USER GUIDE MANLAB 1.0

Credits:

Software written by Remi Arquier

Contacts:

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

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

April 2, 2009

Contents

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 sevral 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, $U \in \mathbb{R}^n$ a vector of unknowns and $\lambda \in \mathbb{R}$ a parameter

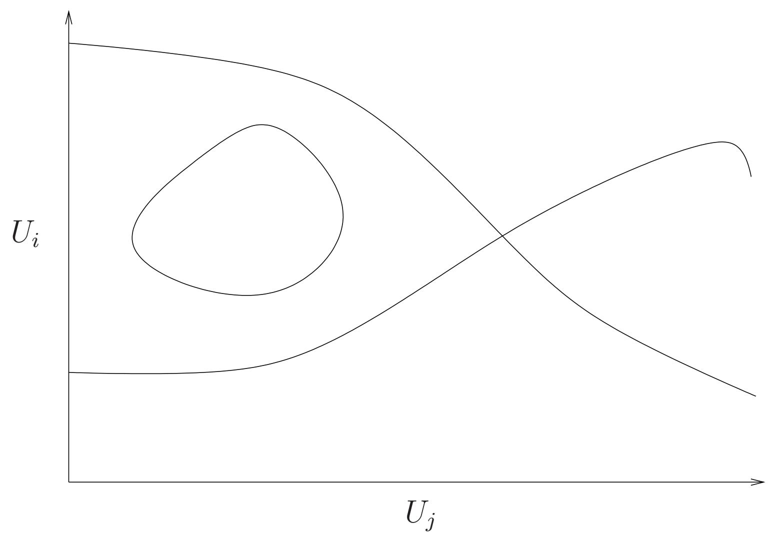


Figure 1.1: *Illustration of a bifurcation diagram showing various branches of solutions.*

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 detction of bifurcation is very robust [3, 4].

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 textbook [2],[5],[6].

1.2.2 Matlab prerequisite

Manlab is intended for version number of **Matlab** larger than (or equal to) 7. The used is supposed to be familiar with basic operations on vectors and structures in **Matlab**. Experience in object-oriented programming is not required.

1.3 HOW TO GIVE THE R VECTOR IN MANLB

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

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

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 genrally the most difficult and unusual task for the beginner with Manlab.

1.4 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/` ← contain the 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.

1.5 Quick start

1.5.1 How to launch examples ?

Matlabă scripts allowing to run the examples can be found in the directory `BASIC-EXAMPLES` of the archive. Make this directory be the working directory of **Matlab**, and type parabole.m in the command line of **Matlab** to launched 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 can be found in the directory `BASIC-EXAMPLES/@PARAB` `L0.m`, `L.m`, `Q.m` and `PARABOLE.m`. 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 argume
objsys = SYS(2);

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

——————End of the file PARABOLE.m

The command `objsys = SYS(2);` creates an object which is a nonlinear system for **Manlab**. The command `objparabole = class(structparabole,'PARABOLE',ob` 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 proagramming, it should be noticed that these few lines allowing the object creation is always the same ! Only the name of the object (here `PARABOLE`) 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;
```

————————File L.m :

```
function L = L(objparabole,U)

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

————————File Q.m :

```
function Q = Q(objparabole,U1,U2)

% Q = - x1 * x2
Q =  -  U1(1) * U2(1);
```

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 appro
> ML_dispvars = [ 1, 2];     % definition of a vector containing the inde
> 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 fuction usually return 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 of File

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 of File

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 allow 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 of File

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 = Translat(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
> Afficher(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, column_vector_U)
function [Q] = Q( instance_yourobj, column_vector_U1, column_vector_U2)
```

Each of these functions should be embeded in a separate file [3]. This is exampli-

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

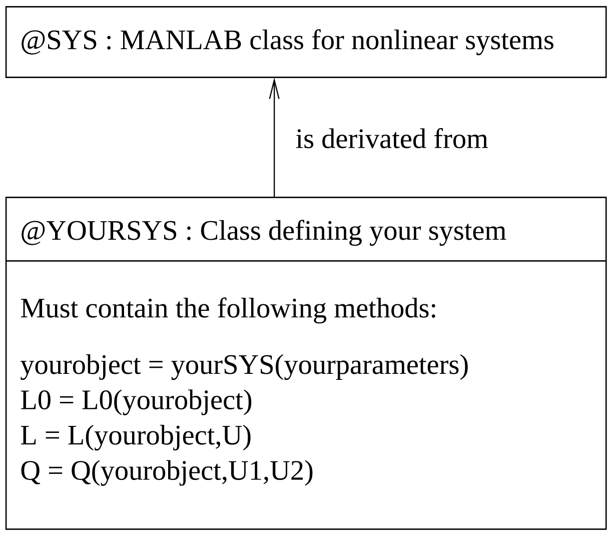

@SYS : MANLAB class for nonlinear systems
is derivated from
@YOURSYS : Class defining your system
Must contain the following methods:
yourobject = yourSYS(yourparameters)
L0 = L0(yourobject)
L = L(yourobject,U)
Q = Q(yourobject,U1,U2)

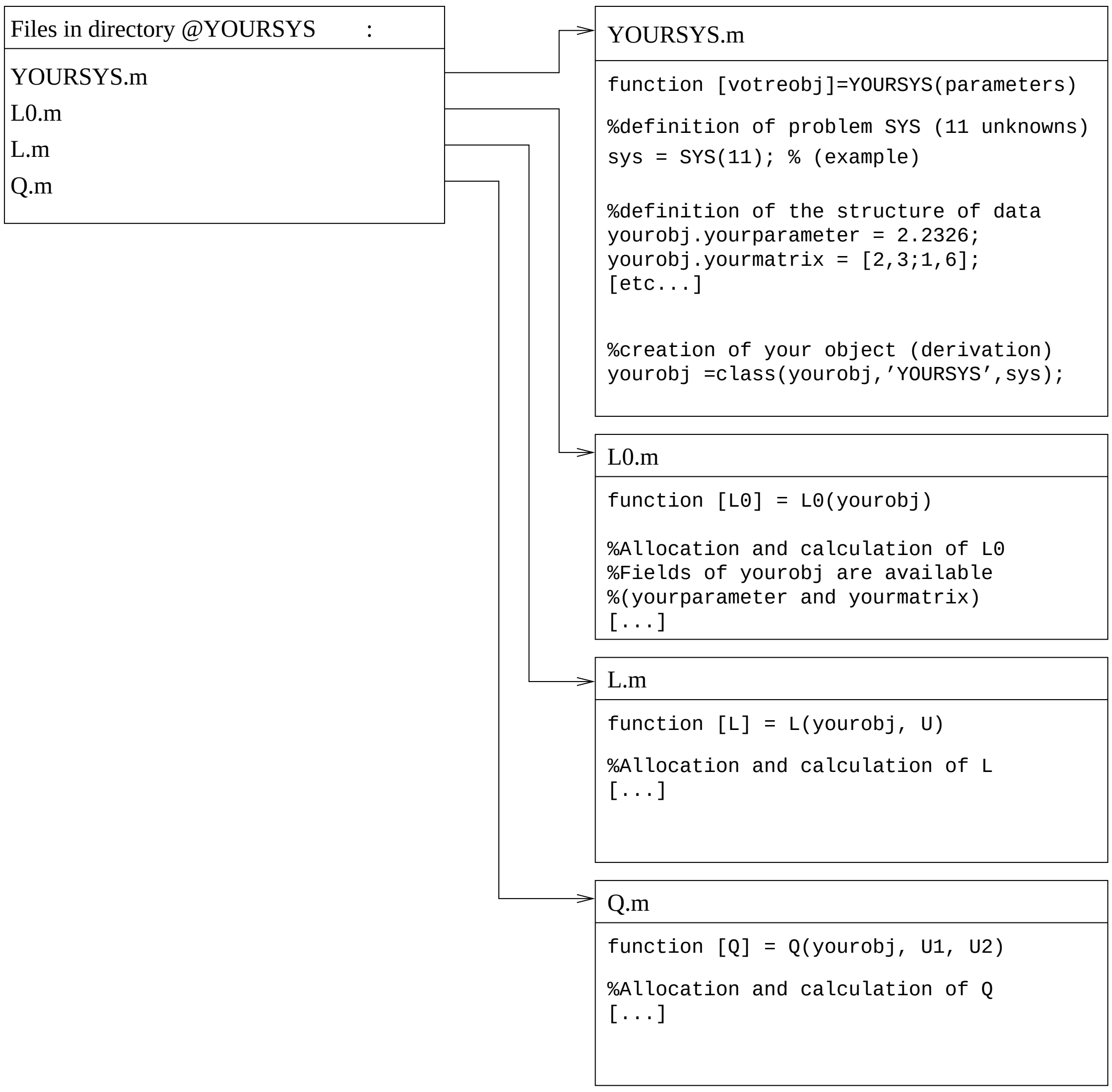
Figure 1.2: *Your class must contain the method required for its creation (the constructor) as well as the methods involved in the calculation of the residue. Moreover, your class should be derivated from a more general class (type @SYS) given by* ***Manlab***

fied in figure (1.3).

1.6.3 Launching

To launch **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` 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`).



YOURSYS.m

```
function [votreobj]=YOURSYS(parameters)

%definition of problem SYS (11 unknowns)
sys = SYS(11); % (example)

%definition of the structure of data
yourobj.yourparameter = 2.2326;
yourobj.yourmatrix = [2,3;1,6];
[etc...]


%creation of your object (derivation)
yourobj =class(yourobj,'YOURSYS',sys);
```

L0.m

```
function [L0] = L0(yourobj)

%Allocation and calculation of L0
%Fields of yourobj are available
%(yourparameter and yourmatrix)
[...]
```

L.m

```
function [L] = L(yourobj, U)

%Allocation and calculation of L
[...]
```

Q.m

```
function [Q] = Q(yourobj, U1, U2)

%Allocation and calculation of Q
[...]
```

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`.*

- 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 of **Manlab** with an object whose type is YOURSYS, an approximate vector and a vector to precise whch variables to plot:

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

When the `manlabstart` is launched, **Manlab** tries to make a correction from the approximate solution `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 indicate by a square box on the bifurcation diagram. A tangent vector at the starting point is also made visible on the diagram. Clicking the + buttom make the continuation toward the direction indicate by the tangent vector. Clicking the - buttom make the continuation in the reverse direction.

1.6.4 The class SYS

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

In particular, the method for the creation of SYS is embedded:

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

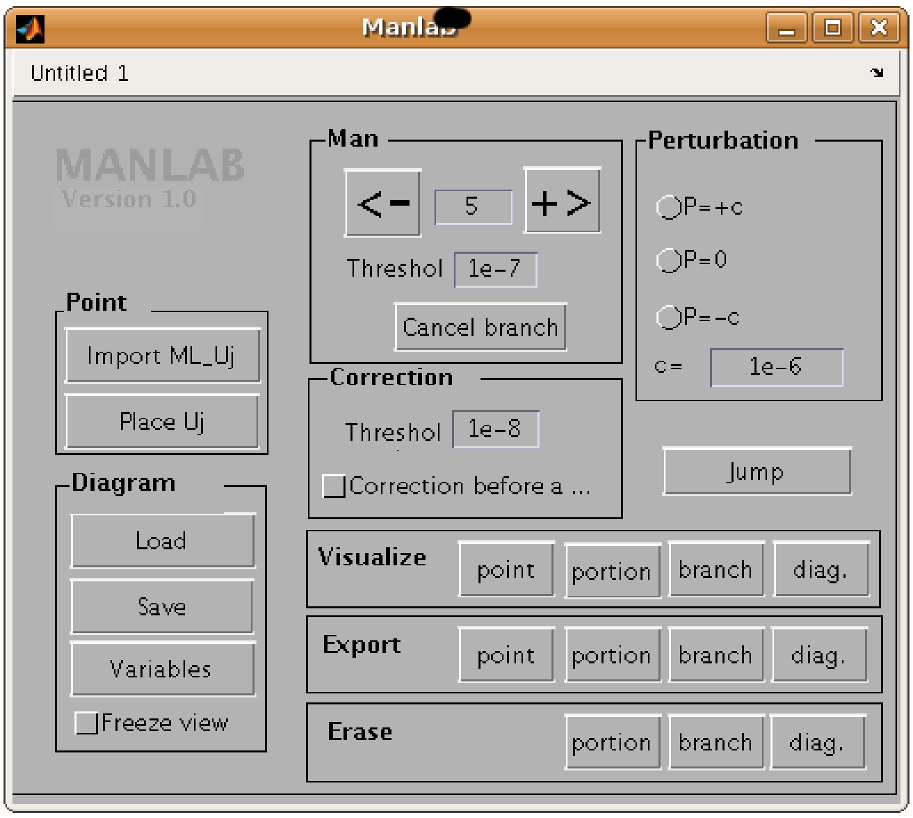
witth `ninc` the number of unknowns of the user problem. The number of equations is automatically set to `ninc-1`.

The class SYS returns an instance of an object (`objsys`) which allow 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.

An example of how to use these functions is given in section 3.2.3 page 43.

1.7 DETAILS OF THE GRAPHICAL INTERFACE



5
Threshol 1e-7
c= 1e-6
Threshol 1e-8

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

1.7.1 "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 indicate 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 indicate on the bifurcation diagram.

Number '5' : Number of sections per branch. '5' is the default value

Threshold : Threshold relative to the residue in the ANM calculation
Erase branch : Erase the latest branch calculated

1.7.2 Frame "Correction"

Threshold : Threshold relative to the correction.
Active at the beginning of a portion : If activated, 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

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 "Visualisation"

point : Launches the visualisation procedure `disp` (local plot) relative to a point in the diagram that you have to select with the mouse. The current active point is not modified.
portion : Launches the visualisation procedure `disp_global` (global plot) relative to a section in the diagram that you have to select with the mouse.
branch : Launches the visualisation procedure `disp_global` (global plot) relative to a branch in the diagram that you have to select with the mouse.
diag. : Launches the visualisation procedure `disp_global` (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.
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.
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.
diag. : Export all the components of vector U into the **Matlab** variable ML_Ud for all points of the diagram.

1.7.6 Frame "Erase"

portion, branch, diag. : Erase the element which has been selected with the mouse.

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 chosed 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"

Freezeview : Freeze/unfreeze the zoom scale of the diagram plot.
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 Jump

This button allows to change the current active point through a jump procedure. This procedure can be seen as a first order predictor. Practically, to perform the jump, the mouse should be placed and left-clicked on the arrival area desired in the diagram. The jump direction is given the tangent to the branch at the current active point. The length of the jump is given by the current active point, the desired arrival area, and the tangent to the branch. Once the jump has been made, a correction step is automatically launched. If this correction step fails, the jump procedure is cancelled.

1.8 USER-DEFINED PLOTS

The continuation graph in **Manlab** only embed the plots of variables whose index are specified in variable `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 at a given point of the continuation graph. To achieve this, a function `disp` should be created and placed in the directory of the class corresponding to the problem studied:

```
function [] = disp(instance_yourobj, U );
```

`instance_yourobj` is an instance of your object and `U` is a column vector containing all the variables of the problem. This function is called each time a continuation point is calculated and each time the button "plot point" is pressed.

Example : of a `disp` function :

```
function  [] = disp(obj, U)
```

```
bar(U); % bar plot of the values of all components of vector U
```

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

Automatic mode : when the global variable `ML_pointdisplay` is set to 1, the `disp` function is called after each new calculation of a portion.

This can be simply achieved in the **Matlab** console:

```
> ML_pointdisplay = 1
```

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 );
```

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

Automatic mode : when the global variable `ML_globaldisplay` is set to 1, the `disp_global` function is called after each new calculation of a portion.

This can be simply achieved in the **Matlab** console:

```
> ML_globaldisplay = 1
```

Chapter 2

Theoretical elements

2.1 CONTINUATION

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

$$\mathbf{R}(\mathbf{U}) = 0 \qquad \text{with } \mathbf{R} \in \mathbb{R}^n \text{ and } \mathbf{U} \in \mathbb{R}^{n+1} \text{ is the vector of unknowns} \tag{2.1}$$

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):

$$\mathbf{U}(a) = \Sigma_{i=0}^{m} \mathbf{U}_i a^i, \qquad \text{where } \mathbf{U}_i \in \mathbb{R}^n,\ a \in \mathbb{R}^+, \tag{2.2}$$

To change the truncation order m, see section 1.6.4 page 22.

More precisely, we only consider quadratic systems R:

$$\mathbf{R} \triangleq \mathbf{L_0} + \mathbf{L}(\mathbf{U}) + \mathbf{Q}(\mathbf{U}, \mathbf{U}) \qquad \text{with } \mathbf{L_0}, \mathbf{L}, \mathbf{Q} \in \mathbb{R}^n \tag{2.3}$$

where $\mathbf{L_0}$ is a constant vector, L is a linear application, and Q is a bilinear form. The ansazt (2.2) is used in equation (2.3), leading to a polynomial expression. Equation

(2.1) 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 U_0 be a solution vector :

- **order 0 :** $\mathbf{L_0} + \mathbf{L}(\mathrm{U}_0) + \mathbf{Q}(\mathrm{U}_0, \mathrm{U}_0) = 0$, which is a trivial system since U_0 is solution of (2.3).
- **order 1 :** $\mathbf{L}(\mathrm{U}_1) + \mathbf{Q}(\mathrm{U}_0, \mathrm{U}_1) + \mathbf{Q}(\mathrm{U}_1, \mathrm{U}_0) = 0$, this system can be rewritten $\mathbf{J}_{\mathbf{U}_0}\mathrm{U}_1 = 0$ where $\mathbf{J}_{\mathbf{U}_0} \in \mathbb{R}^{n\times n+1}$ is the jacobian matrix of $\mathbf{R}$ evaluated at 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 $a \triangleq < \mathbf{U}, \mathrm{U}_1 >$ is generalized to:

$$a \triangleq \mathbf{U}_1^t\mathbf{A}\mathbf{U}, \tag{2.4}$$

where 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 (2.2) in (2.4), and the term-to-term equalization for each power of a leads to;

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

To change the values of A, see section 1.6.4 page 22.

Remarks:

- Once each 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:

$$\mathbf{\Delta U}_{i+1} = -\mathbf{J}^{-1}_{\mathbf{U_{ap}}+\mathbf{\Delta U}_i}\mathbf{R}(\mathbf{U_{ap}} + \mathbf{\Delta U}_i) \tag{2.5}$$

$$\mathbf{\Delta U_{cor}} = \mathbf{\Delta U}_{i+1} \text{ when } \mathbf{\Delta U}_{i+1} - \mathbf{\Delta U}_i \leq \epsilon_c \text{ where } \epsilon_c \text{ is a user-defined threshold} \tag{2.6}$$

  Since $\mathbf{\Delta U_{cor}} \in \mathbb{R}^{n+1}$ and (2.5) is a system of n equations, an additional equation is required. We chose to impose in **Manlab**:

$$\mathbf{U_t}^t \mathbf{\Delta U}_i = 0\ \forall i \text{ where } \mathbf{U_t} \text{ is defined by } \mathbf{J_{U_{ap}}} \mathbf{U_t} = 0 \tag{2.7}$$

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].

2.2 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 2.1). 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.

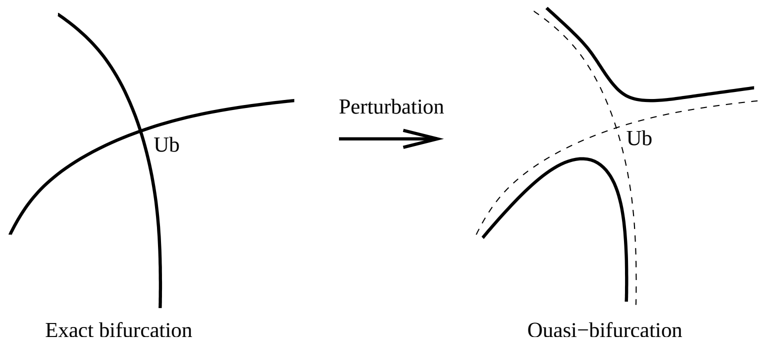


Figure 2.1: *Transformation of an exact bifurcation into a perturbed bifurcation through the addition of a small perturbation.*

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

$$\begin{aligned} R_p(U) &= R(U) + cP \\ &= L0 + L(U) + Q(U,U) + cP \end{aligned} \tag{2.8}$$

where $R_p(U)$ is the perturbated 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 neighborhood 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 2.2.

Finally, in order to go from the original to the perturbated 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.

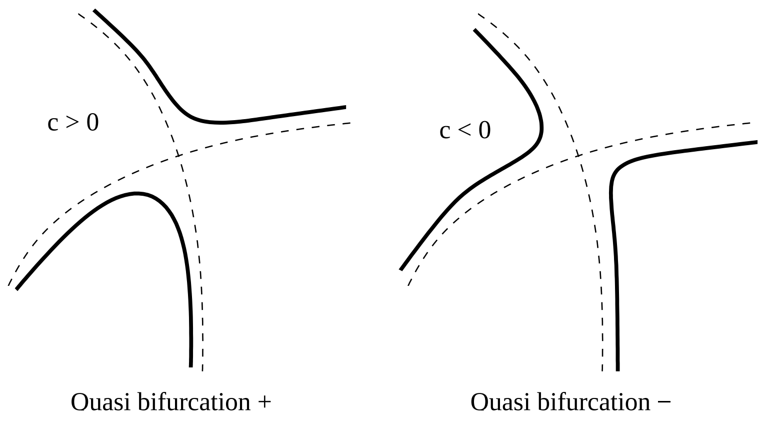


Figure 2.2: *Influence of the sign of* c *on the quasi-bifurcation.*

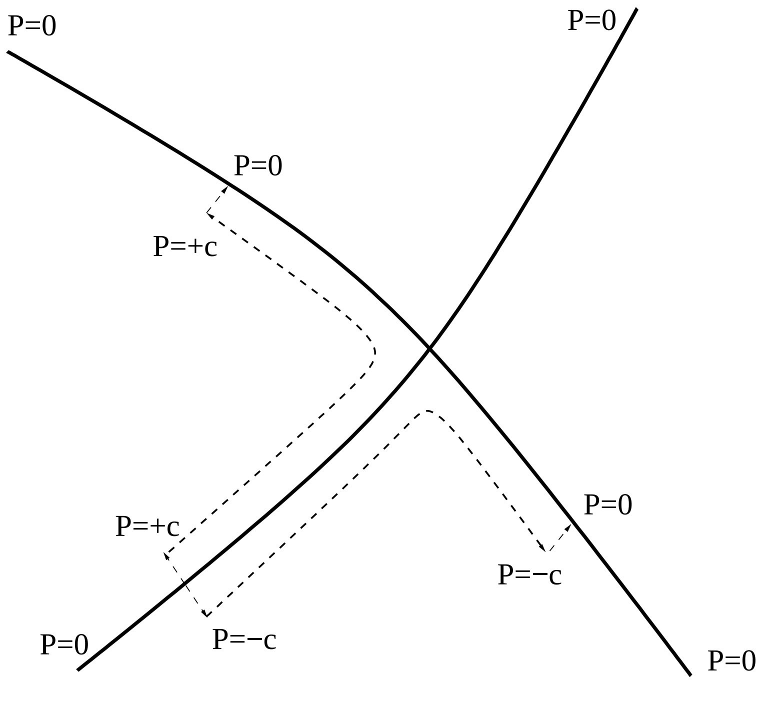


Figure 2.3: *Branch switching through a perturbation method*

several correction steps, **Manlab** allows the user to plot the different branches starting from a bifurcation point, as explained in figure 2.3.

Chapter 3

Examples

3.1 SIMPLE EXAMPLE

3.1.1 Problem statement

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

$$R(U) = R([x, y]^T) = a + bx + cy + dxy + ex^2 + fy^2 \tag{3.1}$$

This problem can be recasted as

$$R(U) = L0 + L(U) + Q(U, U) \tag{3.2}$$

with

$$\begin{cases} L0 & = a \\ L(U) & = bx + cy \\ Q(U_1, U_2) & = dx_1y_2 + ex_1x_2 + fy_1y_2 \end{cases} \tag{3.3}$$

3.1.2 Definition of the user problem

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

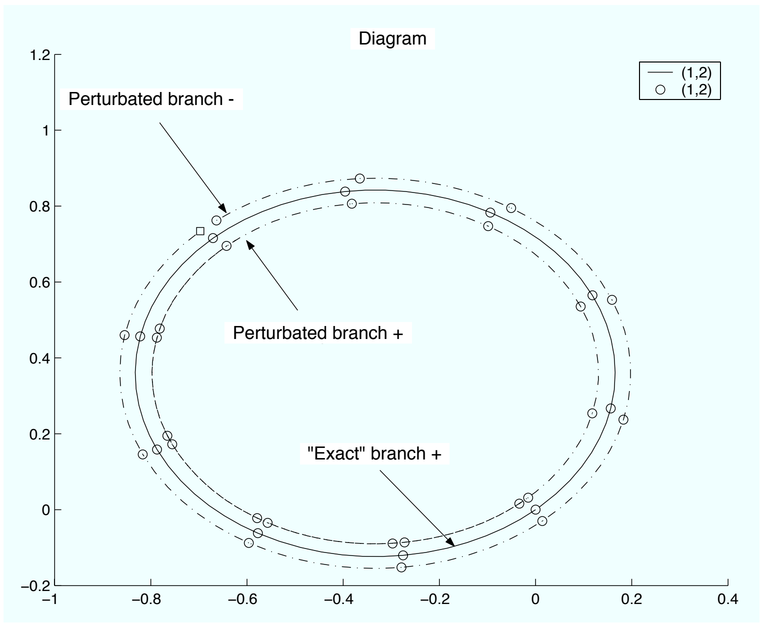


Figure 3.1: *Diagram of the QUADMINI problem when* $a = 0$*,* $b = 2$*,* $c = -2.3$*,* $d = 1$*,* $e = 2$*, and* $f = 3.2$*. The diagram shows the "exact" branch (a circle), as well as two perturbated branches (noted* $+$ *and* $-$*) when the intensity of the perturbation is* $c = 1e - 1$.

————————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
```

3.1.3 Launching the continuation

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

3.2 EXAMPLE WITH BIFURCATION POINTS

3.2.1 Problem statement

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

$$R(U) = R([x,y]^T) = (x - y^2)(y - (x-2)^2) + a \tag{3.4}$$

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:

$$R(U) = R([x,y,v,w]^T) = \begin{cases} vw + a & = 0 \\ x - v - y^2 & = 0 \\ 4 - 4x + w - y + x^2 & = 0 \end{cases} \tag{3.5}$$

which can be recasted quadratically

$$R(U) = L0 + L(U) + Q(U,U) \tag{3.6}$$

with

$$L0 = \begin{cases} a \\ 0 \\ 4 \end{cases}, \quad L(U) = \begin{cases} 0 \\ x - v \\ -4x + w - y \end{cases}, \quad Q(U1,U2) = \begin{cases} v_1 w_2 \\ -y_1 y_2 \\ x_1 x_2 \end{cases} \tag{3.7}$$

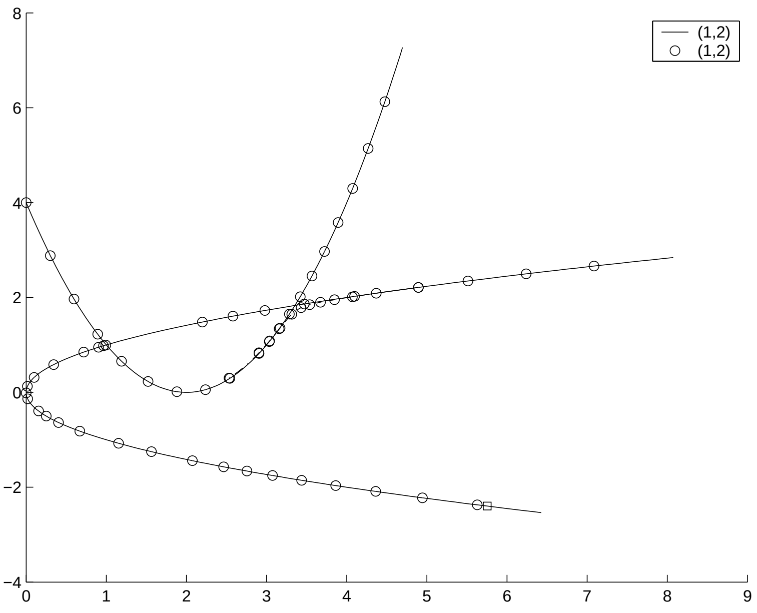


Figure 3.2: *Continuation diagram of the QUADBIF problem, composed of two parabolas intersecting at two bifurcation points.*

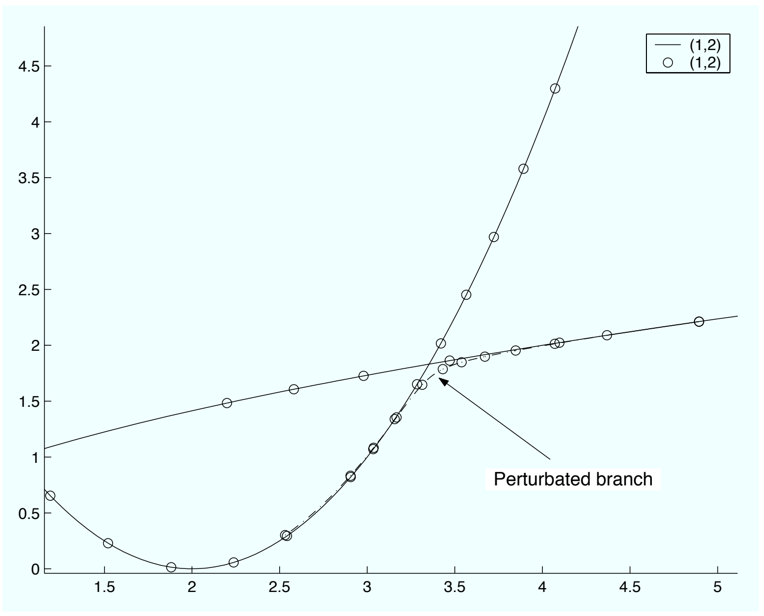


Figure 3.3: *Zoom on the continuation diagram of the QUADBIF problem around one bifurcation point. The use of a perturbated branch allowed the switch between one parabola and the other.*

3.2.2 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);
```

3.2.3 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;
```

Bibliography

[1] E. Doedel, H. Keller, 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 numerique.* Hermes Lavoissier, 2007.

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

[6] E.J. DOEDEL, 2007. *In Numerical Continuation methods for dynamical systems, B. Krauskopf H.M. Osinga J. Galan-Vioque Eds, Springer Verlag,* p1-49. Lecture notes on numerical analysis of nonlinear equations.

[7] M. Potier-Ferry, N. Damil, B. Braikat, J. Descamp, J. M. Cadou, H-L. Cao, and A. El Hage-Hussein. Traitement des fortes non linéarités par la méth-

ode asymptotique numérique. *Comptes Rendus de l'Académie des Sciences de Paris*, t324 Serie II b:171–177, 1997.