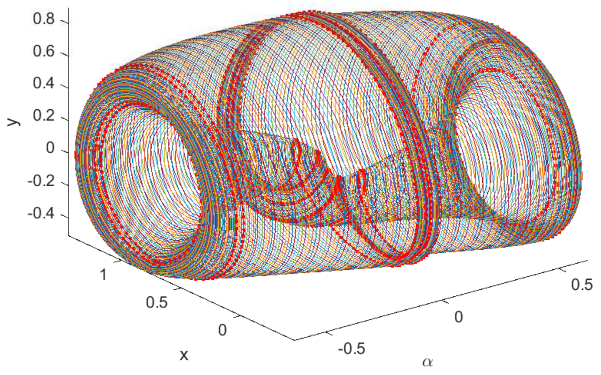
MATCONT:
Continuation toolbox for ODEs in Matlab

W. Govaerts, Yu. A. Kuznetsov, H.G.E. Meijer,

B. Al-Hdaibat, V. De Witte, A. Dhooge, W. Mestrom, N. Neirynck, A.M. Riet and B. Sautois

August 2019, adapted for version MatCont 7.1.



Belgium

Utrecht University
The Netherlands

UNIVERSITY OF TWENTE. The Netherlands

Contents

1 Introduction

The aim of MatCont is to provide a numerical tool for the study of parameterized dynamical systems, in particular bifurcation studies. The first MatCont - related work was done in the master theses [35] and [32]. The package was first announced in [10] and [11]. The present (2019) GUI was described in [34].

The present manual on MatCont is based on version 7.1 of MatCont and the runs were tested on Matlab[27] version 9.5 (R2018b). It is meant to be a practical guide for MatCont users without undue discussion of the details of the used algorithms. Parts of this manual can be used as tutorials since many explicit command line runs are discussed which are also available in the directory `Testruns` of MatCont, see §1.2. There are separate tutorials for the GUI version of MatCont.

MatCont is a GUI-matlab package that builds upon a collection of routines which can also be used independently from the command line of matlab; we refer to this use as the command-line package CL_MatCont. The aim of MatCont and CL_MatCont is to provide a continuation and bifurcation toolbox which is compatible with the standard matlab ODE representation of differential equations. The user can easily use his/her models without rewriting them to a specific package. The matlab programming language makes the use and extensions of the toolbox relatively easy.

This document is structured as follows. In Chapter 2 the underlying mathematics of continuation is treated. Section 2.4 introduces how singularities (usually, but not necessarily, bifurcations) are handled. The toolbox specification in Chapter 3 describes the general aspects of numerically continuing a curve in MatCont, format of the output data etcetera.

In Chapter 4 the *odefile* of a dynamical system is introduced. It is the handle through which MatCont accesses the dynamical system that is being studied. The user can also access the dynamical system directly by calling the odefile from the Matlab command line. Chapter 5 provides more details on time integration and, more specifically, on the computation of Poincaré sections.

The first continuation application of the toolbox, namely the initialization and continuation of equilibria, with the detection of bifurcations and the computation of their normal form coefficients, is given in Chapter 6. Chapter 7 describes the initialization and continuation of limit cycles with the detection of bifurcations and their normal form coefficients. A special feature is the computation of the phase response curve and its derivative.

Chapter 8 describes the continuation of codim 1 bifurcations, i.e. limit points of equilibria, Hopf bifurcation points of equilibria, period doubling bifurcation points of limit cycles, fold bifurcation points of limit cycles and Neimark-Sacker bifurcation points of limit cycles. Their bifurcations are detected and normal form coefficients are computed.

Chapter 9 describes the continuation of two codim 2 bifurcations, namely branch points of equilibria and branch points of limit cycles.

Chapter 10 deals (rather briefly) with the initialization and continuation of homoclinic and heteroclinic orbits.

The continuation routines can also be used outside the field of parameterized dynamical systems. We provide two examples in the appendices. A simple example of drawing a geometric curve is given in Appendix A. The continuation of a solution to a boundary value problem in a free parameter is described in Appendix B.

1.1 Survey of functionalities supported by MatCOnt

Upon the development of MatCont and CL_MatCont, there were multiple objectives:

- Cover as many bifurcations in ODEs as possible (all standard bifurcations with two control parameters are now covered).
- Allow easier data exchange between programs and with matlab's standard ODE solvers.
- Implement robust and efficient numerical methods for all computations, using minimally extended systems where possible.
- Represent results in a form suitable for standard control, identification, and visualization.
- Allow for easy extendibility.

In Table 1 we give a general comparison of the available features during computations for ODEs currently supported by the most widely used software packages auto97/2000 [12], content 1.5 [26] and MatCont/CL_MatCont.

Relationships between objects of codimension $0, 1$ and 2 computed by MatCont and CL_MatCont are presented in Figures 1 and 2, while the symbols and their meaning are summarized in Tables 2 and 3, where the standard terminology is used, see [25].

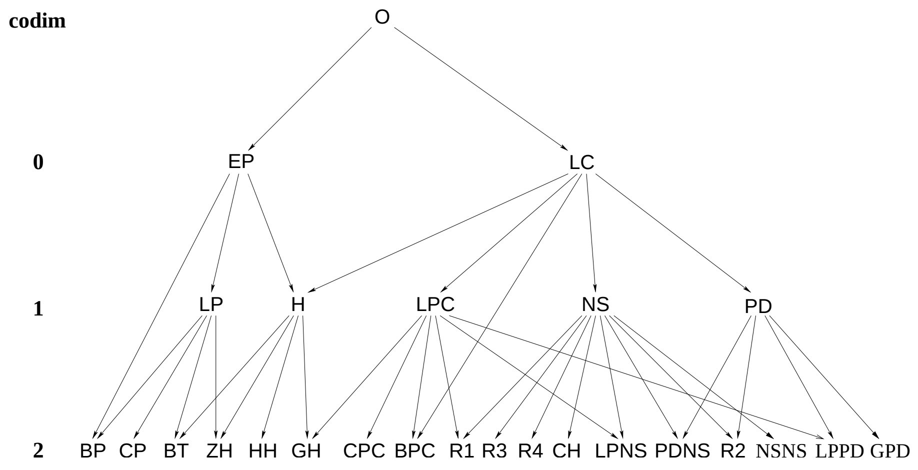


Figure 1: The graph of adjacency for equilibrium and limit cycle bifurcations in MatCont

An arrow in Figure 1 from O to EP or LC means that by starting time integration from a given point we can converge to a stable equilibrium or a stable limit cycle, respectively. In general, an arrow from an object of type A to an object of type B means that the object of type B can be detected (either automatically or by inspecting the output) during the computation of a curve of objects of type A. For example, the arrows from EP to H, LP, and BP mean that we can detect H, LP and BP during the equilibrium continuation. Moreover, for each arrow traced in the reversed direction, i.e. from B to A, there is a possibility to start the computation of the solution of type A starting from a given object B. For example, starting from a BT point, one can initialize the continuation of both LP and H curves. Of course, each object of codim 0 and 1 can be continued in one or two system parameters, respectively.

Table 1: Supported functionalities for ODEs in AUTO (A), CONTENT (C) and MatCont (M).

	A	C	M
time-integration		+	+
Poincaré maps			+
monitoring user functions along curves computed by continuation	+	+	+
continuation of equilibria	+	+	+
detection of branch points and codim 1 bifurcations (limit and Hopf points) of equilibria	+	+	+
computation of normal forms for codim 1 bifurcations of equilibria		+	+
continuation of codim 1 bifurcations of equilibria	+	+	+
detection of codim 2 equilibrium bifurcations (cusp, Bogdanov-Takens, fold-Hopf, generalized and double Hopf)		+	+
computation of normal forms for codim 2 bifurcations of equilibria			+
continuation of codim 2 equilibrium bifurcations in three parameters		+	
continuation of limit cycles	+	+	+
computation of phase response curves and their derivatives			+
detection of branch points and codim 1 bifurcations (limit points, flip and Neimark-Sacker (torus)) of cycles	+	+	+
continuation of codim 1 bifurcations of cycles	+		+
branch switching at equilibrium and cycle bifurcations	+	+	+
continuation of branch points of equilibria and cycles			+
computation of normal forms for codim 1 bifurcations of cycles			+
detection of codim 2 bifurcations of cycles			+
computation of normal forms for codim 2 bifurcations of cycles			+
continuation of orbits homoclinic to equilibria	+		+
continuation of orbits heteroclinic to equilibria	+		+

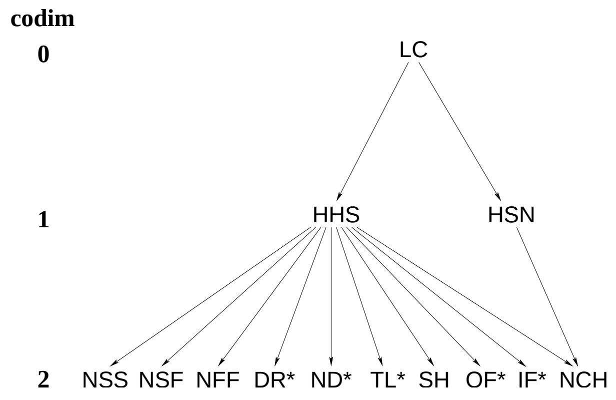


Figure 2: The graph of adjacency for homoclinic bifurcations in MatCont; here * stands for S or U.

The same interpretation applies to the arrows in Figure 2, where '*' stands for either S or U, depending on whether a stable or an unstable invariant manifold is involved.

In principle, the graphs presented in Figures 1 and 2 are connected. Indeed, it is known that curves of codim 1 homoclinic bifurcations emanate from the BT, ZH, and HH codim 2 points. The current version of MatCont fully supports, however, only one such connection: BT $\rightarrow$ HHS.

1.2 Testruns and how to start with Cl_MatCont

Cl_MatCont has a large number of testruns which are collected in the directory Testruns. One reason for that is to check that the computational routines work well in a particular environment. But there is also a paedagogical reason.

The easiest way to start with MatCont is by using the GUI-version and going through the tutorials. However, the command line version Cl_MatCont has more functionalities and is more flexible for advanced use.

A reasonable way to start using Cl_MatCont is to first have a quick reading of the most important (sub)sections of this manual and then go through the testruns provided in the directory `Testruns` in increasing order of complexity.

As the most important (sub)sections we recommend, in that order, §1.1, §1.5, §2, §3 with the exception of §3.6, §3.7 and §3.8, §4 with the exception of §4.3 and §4.4.

We then recommend to go through the testruns in the order suggested below and to compare their content and their output with the discussion in the manual. Of course, it will often be necessary to turn back to the manual for more information.

- `testodefile.m`, discussed in §4.3 (basic use of the system definition file)
- `testtimeinteg.m` and `testtimeintegJacobian`, discussed in §5.1.3 (time integration)
- `testPoincare.m`, discussed in §5.2.1 (Poincaré map)
- `testmymlPRC.m`, discussed in §6.2 and §7.8 (finding a stable equilibrium by time integration, continuation of equilibria, detecting a Hopf point, starting a continuation of limit cycles, computing the phase response curve and its derivative).

Type of object	Label
Point	P
Orbit	O
Equilibrium	EP
Limit cycle	LC
Limit Point (fold) bifurcation	LP
Hopf bifurcation	H
Limit Point bifurcation of cycles	LPC
Neimark-Sacker (torus) bifurcation	NS
Period Doubling (flip) bifurcation	PD
Branch Point	BP
Cusp bifurcation	CP
Bogdanov-Takens bifurcation	BT
Zero-Hopf bifurcation	ZH
Double Hopf bifurcation	HH
Generalized Hopf (Bautin) bifurcation	GH
Branch Point of Cycles	BPC
Cusp bifurcation of Cycles	CPC
1:1 Resonance	R1
1:2 Resonance	R2
1:3 Resonance	R3
1:4 Resonance	R4
Chenciner (generalized Neimark-Sacker) bifurcation	CH
Fold–Neimark-Sacker bifurcation	LPNS
Flip–Neimark-Sacker bifurcation	PDNS
Fold-flip	LPPD
Double Neimark-Sacker	NSNS
Generalized Period Doubling	GPD

Table 2: Equilibrium- and cycle-related objects and their labels within the GUI

- `testadaptPRC.m`, discussed in §7.8 (continuation of equilibria, detection of a Hopf point, continuation of limit cycles, starting a curve of limit cycles by starting from a limit cycle, computation of the phase response curve and its derivative)
- `testselectcycle.m`, discussed in §7.4 (starting the continuation of limit cycles from an orbit obtained by time integration).
- `testbratu.m` and `testbratu2.m`, discussed in §6.5 (user function, equilibrium continuation, neutral equilibrium (H), branch point, branch switching in a branch point of equilibria)
- `testequilcataloscill.m` and `testLPcataloscill.m`, discussed in §8.1.5 (equilibrium continuation, detection of LP points, continuation of an LP-curve, detection of BT and Cusp points)
- `testLPHopfcataloscill.m`, discussed in §8.2.5 (equilibrium continuation, detection of Hopf points, continuation of a Hopf curve, detection of BT points, Hopf points versus

Type of object	Label
Limit cycle	LC
Homoclinic to Hyperbolic Saddle	HHS
Homoclinic to Saddle-Node	HSN
Neutral saddle	NSS
Neutral saddle-focus	NSF
Neutral Bi-Focus	NFF
Shilnikov-Hopf	SH
Double Real Stable leading eigenvalue	DRS
Double Real Unstable leading eigenvalue	DRU
Neutrally-Divergent saddle-focus (Stable)	NDS
Neutrally-Divergent saddle-focus (Unstable)	NDU
Three Leading eigenvalues (Stable)	TLS
Three Leading eigenvalues (Unstable)	TLU
Orbit-Flip with respect to the Stable manifold	OFS
Orbit-Flip with respect to the Unstable manifold	OFU
Inclination-Flip with respect to the Stable manifold	IFS
Inclination-Flip with respect to the Unstable manifold	IFU
Non-Central Homoclinic to saddle-node	NCH

Table 3: Objects related to homoclinics to equilibria and their labels within the GUI

neutral equilibria, detection of GH points)

- `testadapt.m`, discussed in §7.7 (continuation of equilibria, detection of Hopf points)
- `testadapt1.m`, discussed in §7.7 (starting a curve of limit cycles from a Hopf point, continuation of limit cycles, detection of fold and period doubling point of cycles)
- `testadapt2.m`, discussed in §7.7 (starting a period doubled orbit from a PD point; detection of secondary PD points; producing the cover picture of the manual)
- `testadapt3.m`, discussed in §8.3.5 (starting a curve of period doubling bifurcations from a PD point; detection of Resonance 1:2 (R2) bifurcation points)
- `testEquilMLfast.m`, discussed in §8.4.4 (continuation of equilibria, detection of limit points, Hopf points and a neutral equilibrium (H))
- `testLCMLfast.m`, discussed in §8.4.4 (continuation of limit cycles and detection of an LPC (limit point of cycles) point)
- `testLPCMLfast.m`, discussed in §8.4.4 (continuation of an LPC curve with two free parameters).
- `testtorBPC1.m` and `testtorBPC2.m`, discussed in §8.5.4 (detection of a Neimark-Sacker point on a curve of periodic orbits and preceding computations; in `testtorBPC2.m` also the use of a user function).
- `testtorBPC3.m`, discussed in §8.5.4 (continuation of Neimark-Sacker bifurcations in two free parameters)

- `cstr1.m`, `cstr2` and `cstr3`, discussed in §9.1.4 (continuation of equilibria, detection of LP points, continuation of an LP curve, detection of branch points with respect to two different parameters, continuation of a BP curve with three free parameters)
- `testtorBPC4.m`, discussed in §9.2.4 (continuation of equilibria, detection of a Hopf point and a branch point)
- `testtorBPC5.m`, discussed in §9.2.4 (continuation of limit cycles, detection of one LPC and two BPC (branch points of cycles))
- `testtorBPC6.m`, discussed in §9.2.4 (branching off in a BPC, detection of a NS and a PD bifurcation point)
- `testtorBPC7.m`, discussed in §9.2.4 (continuation of a BPC with three free parameters)
- `testmyml.m` and `homoc1.m`, discussed in §10.5 (computation of a curve of equilibria from a starting point that in the first case was obtained by time integration, detection of a Hopf point and starting a curve of limit cycles, starting a continuation of orbits homoclinic to saddle from a limit cycle with large period)
- `testdrawcurve.m`, discussed in Appendix A (drawing a curve by using a continuation algorithm)
- `testbrusselator.m`, discussed in Appendix B (continuation of an equilibrium solution to a 1-dimensional PDE)

1.3 Computational routines in MatCont

MatCont is organized around the continuation of curves of 12 different types, i.e equilibrium, limit cycle, limit point, Hopf, limit point of cycles, period doubling of cycles, Neimark-Sacker bifurcation of cycles, branch point, branch point of cycles, orbits homoclinic to saddle, orbits homoclinic to saddle node and heteroclinic orbits. Each of these curve types has its own dedicated directory in MatCont, cf. §3.7.

Each curve type is related to a parameterized dynamical system and time integration (simulation) of that system is often either desirable to confirm the results obtained from continuation studies or needed for the initialization of the continuation curves. Therefore, MatCont provides access to the standard Matlab integrators and two additional high-order integrators called `ode78.m` and `ode87.m`.

Next to continuation, the main computational contribution of MatContM is in the initializers, which come in various forms. We mention

- Some consist of a single initialization routine which can be very simple, e.g. `init_H_EP` (starting an equilibrium curve from a Hopf point) or more complicated, e.g. `init_BT_Hom` (starting a curve a homoclinic orbits from a Bogdanov-Takens point). We mention in particular the quite useful initializers `init_LC_Hom` and `init_LC_HSN` in §10.3 to start homoclinic orbits to saddle or to saddle node from a limit cycle with a long period.
- Two of the most useful initializers rely on the use of the integrator routines to find either equilibrium points (usually stable, see §6.2) or limit cycles (usually stable, see §7.4).

- Some initializers require a specific procedure, namely these for orbits homoclinic to saddle, orbits homoclinic to saddle node and heteroclinic orbits, cf. §10.3. These have their own dedicated directories in MatCont, cf §3.7.

Computation of the normal form coefficients of bifurcations is another important task. Some of the involved routines are in the directories dedicated to curves on which the bifurcations are naturally detected. Two directories, namely MultilinearForms and LimitCycle-Codim2 are dedicated to a more systematic handling of this type of computation.

In §5.2 we discuss the computation of Poincaré sections, based on the integrator routines.

Finally, the computation of phase response curves and their derivatives in §7.8 is a specific feature of MatCont.

1.4 Availability

This package is freely available for download at:

`http://www.sourceforge.net/`

Search for 'matcont' and then preferably go to the latest release in the directory 'matcont'. Unzipping the downloaded file creates a directory `matcont` with all necessary files (see section 3.7). Download also the `readme.pdf` file which contains information about the release such as where to find tutorials and the manual.

1.5 Software requirements

The present manual on MatCont is based on version 7.1 of MatCont and the runs were tested on Matlab 9.5 (R2018b).

In principle no special matlab packages or toolboxes are necessary but the Symbolic Toolbox will be used if it is available. We note that in the computation of normal form coefficients derivatives up to order 5 are used and these may contain big errors when computed by the finite difference approximations which are used if no symbolic derivatives are available.

It is important to know that the directory LimitCycle contains 7 c-files that have to be compiled to MEX files in a platform-dependent way. The present version is expected to work on all Windows, Unix and Mac 32-bit and 64-bit platforms when using the default c-compiler to compile the c-files in the LimitCycle directory into MEX-files (when matcont is called for the first time on any platform). This is no problem under Windows 32, Linux 32 and Linux 64 where the c-compiler is present by default. For Windows 64 and Mac it may be necessary to download the compiler separately. However, for Windows32, Windows 64 and Mac64 the MEX files are also available in the directory "Auxiliaries" on the root of the MatCont website.

In general, compilation can depend on the Matlab version and operating system of the computer. In case of problems, check for details provided with the MatCont release

1.6 Disclaimer

The packages MatCont and CL_MatCont are freely available for non-commercial use on an "as is" basis. In no circumstances can the authors be held liable for any deficiency, fault or other mishappening with regard to the use or performance of MatCont and/or CL_MatCont.

For the best understanding of dynamical systems and bifurcation theory we refer to [25].

2 Mathematical aspects of numerical continuation and handling of singularities

Consider a smooth function $F : \mathbf{R}^{n+1} \to \mathbf{R}^n$. We want to compute a solution curve of the equation $F(x) = 0$. Numerical continuation is a technique to compute a consecutive sequence of points which approximate the desired branch. Most continuation algorithms implement a predictor-corrector method. The idea behind this method is to generate a sequence of points x_i, $i = 1, 2, \ldots$ along the curve, satisfying a chosen tolerance criterion: $||F(x_i)|| \leq \epsilon$ for some $\epsilon > 0$ and an additional accuracy condition $||\delta x_i|| \leq \epsilon'$ where $\epsilon' > 0$ and δx_i is the last Newton correction.

To show how the points are generated, suppose we have found a point x_i on the curve. Also suppose we have a normalized tangent vector v_i at x_i, i.e. $F_x(x_i)v_i = 0$, $\langle v_i, v_i \rangle = 1$.

The computation of the next point x_{i+1} consists of 2 steps:

- prediction of a new point
- correction of the predicted point

2.1 Prediction

Suppose $h > 0$ which will represent a stepsize. A commonly used predictor is *tangent prediction*:

$$X^0 = x_i + hv_i. \tag{1}$$

The choice of the stepsize is discussed in section 2.3.

2.2 Correction

We assume that X^0 is close to the curve. To find the point x_{i+1} on the curve we use a Newton-like procedure. Since the standard Newton iterations can only be applied to systems with the same number of equations as unknowns, we have to append an extra scalar condition:

$$\left\{ \begin{array}{lcl} F(x) & = & 0, \\ g(x) & = & 0. \end{array} \right. \tag{2}$$

The question is how to choose the function $g(x)$.

2.2.1 Pseudo-arclength continuation

One option for choosing $g(x)$ is to select a hyperplane passing through X^0 that is orthogonal to the vector v_i:

$$g(x) = \langle x - X^0, v_i \rangle \ . \tag{3}$$

So, the Newton iteration becomes:

$$X^{k+1} = X^k - H_x^{-1}(X^k)H(X^k) \tag{4}$$

$$H(X) = \left(\begin{array}{c} F(X) \\ 0 \end{array} \right), \quad H_x(X) = \left(\begin{array}{c} F_x(X) \\ v_i^T \end{array} \right) \ . \tag{5}$$

Then one can prove that the Newton iteration for (2) will converge to a point x_{i+1} on the curve from X^0 provided that the stepsize h is sufficiently small and that the curve is regular

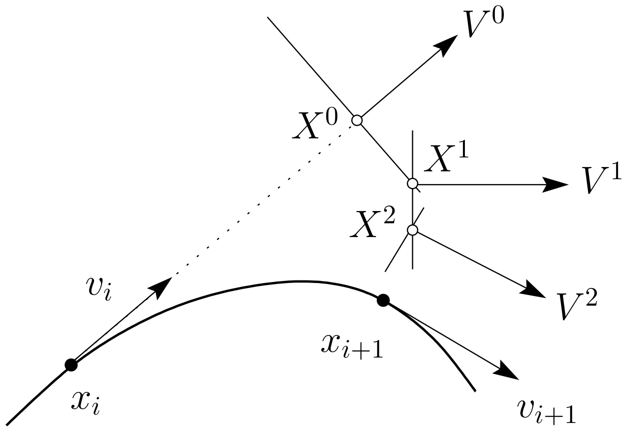


Figure 3: Moore-Penrose continuation

(rank $F_x(x) = n$). Having found the new point x_{i+1} on the curve we need to compute the tangent vector at that point:

$$F_x(x_{i+1})v_{i+1} = 0\ . \tag{6}$$

Furthermore the direction along the curve must be preserved: $\langle v_i, v_{i+1}\rangle = 1$, so we get the $(n+1)$-dimensional appended system

$$\begin{pmatrix} F_x(x_{i+1}) \\ v_i^T \end{pmatrix} v_{i+1} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{7}$$

Upon solving this system, v_{i+1} must be normalized.

2.2.2 Moore-Penrose continuation

CL_MatCont implements a continuation method that is slightly different from the pseudo-arclength continuation.

Definition 1 *Let A be an $N \times (N+1)$ matrix with maximal rank. Then the* Moore-Penrose inverse *of A is defined by $A^+ = A^T(AA^T)^{-1}$.*

Let A be an $N \times (N+1)$ matrix with maximal rank. Consider the following linear system with $x, v \in \mathbf{R}^{N+1}, b \in \mathbf{R}^N$:

$$Ax = b \tag{8}$$
$$v^T x = 0 \tag{9}$$

where x is a point on the curve and v its tangent vector with respect to A, i.e. $Av = 0$. Since $AA^+b = b$ and $v^T A^+ b = \langle Av, (AA^T)^{-1}b\rangle = 0$, a solution of this system is

$$x = A^+ b. \tag{10}$$

Suppose we have a predicted point X^0 using (1). We want to find the point x on the curve which is nearest to X^0, i.e. we are trying to solve the optimization problem:

$$\min_x \{||x - X^0|| \mid F(x) = 0\} \tag{11}$$

So, the system we need to solve is:

$$\begin{aligned} F(x) &= 0 && (12)\\ w^T(x - X^0) &= 0 && (13)\end{aligned}$$

where w is the tangent vector at point x. In Newton's method this system is solved using a linearization about X^0. Taylor expansion about X^0 gives:

$$\begin{aligned} F(x) &= F(X^0) + F_x(X^0)(x - X^0) + \mathcal{O}(||x - X^0||^2) && (14)\\ w^T(x - X^0) &= v^T(x - X^0) + \mathcal{O}(||x - X^0||^2)\ . && (15)\end{aligned}$$

So when we discard the higher order terms we can see using (8) and (10) that the solution of this system is:

$$x = X^0 - F_x^+(X^0)F(X^0)\ . \qquad (16)$$

However, the null vector of $F_x(X^0)$ is not known, therefore we approximate it by $V^0 = v_i$, the tangent vector at x_i. Geometrically this means we are solving $F(x) = 0$ in a hyperplane perpendicular to the previous tangent vector. This is illustrated in Figure 3. In other words, the extra function $g(x)$ in (2) becomes:

$$g_k(x) = \langle x - X^k, V^k\rangle, \qquad (17)$$

where $F_x(X^{k-1})V^k = 0$ for $k = 1, 2, \ldots$. Thus, the Newton iteration we are doing is:

$$\begin{aligned} X^{k+1} &= X^k - H_x^{-1}(X^k, V^k)H(X^k, V^k) && (18)\\ V^{k+1} &= V^k - H_x^{-1}(X^k, V^k)R(X^k, V^k) && (19)\end{aligned}$$

$$H(X, V) = \begin{pmatrix} F(X) \\ 0 \end{pmatrix},\quad H_x(X, V) = \begin{pmatrix} F_x(X) \\ V^T \end{pmatrix} \qquad (20)$$

$$R(X, V) = \begin{pmatrix} F_x(X)V \\ 0 \end{pmatrix}\ . \qquad (21)$$

One can prove that under the same conditions as for the pseudo-arclength continuation, the Newton iterations (18) and (19) converge to a point on the curve x_{i+1} and the corresponding tangent vector v_{i+1}, respectively. In the pseudo-arclength continuation, we had to compute a tangent vector when a new point was found. In this case however, we already compute the tangent vectors V^k at each iterate (19), so we only need to normalize the computed tangent vectors.

2.3 Stepsize control

Stepsize control is an important issue in these algorithms. Too small stepsizes lead to unnecessary work being done, while too big stepsizes can lead to losing details of the curve. An easily implementable and proven to be reliable method is convergence-dependent control.

Consider the computation of a next point using step size h_i. If the computation converged, let n denote the number of Newton iterations needed. Then the new step size h_{i+1} will be selected as follows:

$$h_{i+1} = \begin{cases} h_i \cdot h_{dec} & \text{if not converged} \\ h_i \cdot h_{inc} & \text{if converged and } n < n_{thr} \\ h_i & \text{otherwise} \end{cases} \qquad (22)$$

where $h_{dec} < 1$, $h_{inc} > 1$ and n_{thr} are constants which are experimentally determined.

2.4 Singularity handling

This section explains the idea of singularities which can occur on a solution branch.

2.4.1 Test functions

The idea to detect singularities is to define smooth scalar functions which have regular zeros at the singularity points. These functions are called *test functions*. Suppose we have a singularity S which is detectable by a test function $\phi : \mathbf{R}^{n+1} \to \mathbf{R}$. Also assume we have found two consecutive points x_i and x_{i+1} on the curve

$$F(x) = 0, \quad F : \mathbf{R}^{n+1} \to \mathbf{R}^n \ . \tag{23}$$

The singularity S will then be detected if

$$\phi(x_i)\phi(x_{i+1}) < 0 \ . \tag{24}$$

Having found two points x_i and x_{i+1} one may want to locate the point x^* where $\phi(x)$ vanishes. A logical solution is to solve the following system

$$F(x) = 0 \tag{25}$$
$$\phi(x) = 0 \tag{26}$$

using Newton iterations starting at x_i. However, to use this method, one should be able to compute the derivatives of $\phi(x)$ with respect to x, which is not always easy. To avoid this difficulty we implemented by default a one-dimensional secant method to locate $\phi(x) = 0$ along the curve. Notice that this involves Newton corrections at each intermediate point.

2.4.2 Multiple test functions

The above is a general way to detect and locate singularities depending on one test function. However, it may happen that it is not possible to represent a singularity with only one test function.

Suppose we have a singularity S which depends on n_t test functions. Also assume we have found two consecutive points x_i and x_{i+1} and all n_t test functions change sign:

$$\forall j \in [1, n_t] : \phi_j(x_i)\phi_j(x_{i+1}) < 0 \tag{27}$$

Also assume we have found, using a one-dimensional secant method, all zeros x_j^* of the test functions. In the ideal (exact) case all these zeros will coincide:

$$\forall j \in [1, n_t] : x^* = x_j^* \quad \text{and} \quad \phi_j(x_j^*) = 0 \tag{28}$$

Since the continuation is not exact but numerical, we cannot assume this. However, the locations of x_j^* probably will be clustered around some center point x^c. In this case we will *glue* the points x_j^* to $x^* = x^c$.

A cluster will be detected if $\forall i, j \in [1, n_t] : ||x_i^* - x_j^*|| \leq \epsilon$ for some small value ϵ. In this case we define x^* as the mean of all located zeroes:

$$x^* = \frac{1}{n_t} \sum_{j=1}^{n_t} x_j^* \tag{29}$$

2.4.3 Singularity matrix

Until now we have discussed singularities depending only on test functions which vanish. Suppose we have two singularities S_1 and S_2, depending respectively on test functions ϕ_1 and ϕ_2. Namely, assume that ϕ_1 vanishes at both S_1 and S_2, while ϕ_2 vanishes at only S_2. Therefore we need a possibility to represent singularities using non-vanishing test functions.

To represent all singularities we will introduce a *singularity matrix* (as in [26]). This matrix is a compact way to describe the relation between the singularities and all test functions.

Suppose we are interested in n_s singularities and n_t test functions which are needed to detect and locate the singularities. Then let S be the $n_s \times n_t$ matrix, such that:

$$S_{ij} = \begin{cases} 0 & \text{singularity } i\text{: test function } j \text{ must vanish,} \\ 1 & \text{singularity } i\text{: test function } j \text{ must not vanish,} \\ \text{otherwise} & \text{singularity } i\text{: ignore test function } j. \end{cases} \tag{30}$$

2.4.4 User location

In some cases the default location algorithm can have problems to locate a bifurcation point. Therefore we provide the possibility to define a specific location algorithm for a particular bifurcation. In fact, this is done for the location of branch points of equilibrium curves and curves of limitcycles.

3 General software aspects of MatCont

3.1 System definition

The user defines his dynamical system in an *odefile*, using the framework as in the file `standard.m` in the Systems directory.

In the function `fun_eval`, the dynamical system is to be given, where the parameters should be listed individually. Under `init`, the user can define some initialization parameters, as the phase variable values, the timespan, etc. All phase variables and parameters are expected to be scalar variables, not vectors or matrices. In the further functions, it is possible to supply the symbolic derivatives of the system to various orders to increase the speed and/or improve accuracy of the algorithm. Note that for the state variables derivatives up to fifth order can be provided. For the parameters first order derivatives of `fun_eval` and of its state Jacobian can be provided. No other derivatives can be supplied through the system definition file, since they are never used in (CL_)MatCont. We recall that the presence of symbolic derivatives, whenever necessary, typically will be stored in the subfields `cds.options.SymDerivate` and `cds.options.SymDerivativeP` of `cds.options`. This is done by the initializers to the curve description files. We note that $0 \leq$ SymDerivative ≤ 5 denotes the order of the highest symbolically available derivative with respect to state variables; similarly $0 \leq$ SymDerivativeP ≤ 2 denotes the order of the highest symbolically available derivative in which a derivative with respect to a parameter is involved. It is always assumed that this implies the presence of the lower order derivatives.

Finally, the *odefile* can contain the description of any number of user functions, i.e. functions that can be monitored along computed curves and whose zeros can be detected and located. User functions can serve many purposes; an example of a sophisticated use of user functions is given in §8.5.4.

Details and examples on the construction of the *odefile* of a dynamical system are given in Chapter 4.

3.2 Continuation and output

The syntax of the continuer is:

```
[x,v,s,h,f] = cont(@curve, x0, v0, options);
```

`curve` is a MATLAB m-file where the problem is specified (cf. section 3.3). Evaluating a function by means of a function handle replaces the earlier MATLAB mechanism of evaluating a function through a string containing the function name.
`x0` and `v0` are respectively the initial point and the tangent vector at the initial point where the continuation starts.
`options` is a structure as described in section 3.4.
The arguments `v0` and `options` can be omitted. In this case the tangent vector at `x0` is computed internally and default options are used.

The function returns a series of matrices:

`x` and `v` are the points and their tangent vectors along the curve. Each column in `x` and `v` corresponds to a point on the curve.

`s` is an array with structures containing information about the found singularities. Its first and last elements always refer to the first and last points of the curve, respectively, since for convenience these are also considered "special".

Each element of this structure array `s` has the following fields:

`s.index`	index of the singularity point in `x`, so `s(1).index` is always equal to 1 and `s(end).index` is the number of computed points.
`s.label`	label of the singularity; by convention `s(1).label` is "00" and `s(end).label` is "99".
`s.msg`	a string that may contain any information which is useful for the user, for example the full name of the detected special point.
`s.data`	For each special point it contains fields with additional information, depending on the type of point, and accumulating with increasing codimension:

- Equilibrium: s.data.v = tangent vector at the bifurcation
  - Hopf point: s.data.lyapunov = first Lyapunov coefficient
  - Limit point: s.data.a = normal form coefficient
    - ∗ Bogdanov-Takens / Zero Hopf / Double Hopf / Generalized Hopf / Cusp : s.data.c = normal form coefficient
- Limit cycle:

  s.data.multipliers = multipliers at the bifurcation

  s.data.timemesh = time mesh of the orbit at the bifurcation

  s.data.ntst = number of test intervals

  s.data.ncol = number of collocation points

  s.data.parametervalues = parameter values at the bifurcation

  s.data.T = period of the orbit at the bifurcation

  s.data.phi = bordering vector for locating PD bifurcations
  - Period-doubling point: s.data.pdcoefficient = normal form coefficient
  - Limit point of cycles: s.data.lpccoefficient = normal form coefficient
  - Neimark-Sacker point: s.data.nscoefficient = normal form coefficient
- Homoclinic to hyperbolic saddle / Homoclinic to saddle-node:

  s.data.timemesh = time mesh of the orbit at the bifurcation

  s.data.ntst = number of test intervals

  s.data.ncol = number of collocation points

  s.data.parametervalues = parameter values at the bifurcation

  s.data.T = period of the orbit at the bifurcation

`h` contains some information on the continuation process. Its columns are related to the computed points, and have the following components:

Stepsize	Stepsize used to calculate this point (zero for initial point and singular points)
Half the number of correction iterations, rounded up to the next integer	For singular points this is the number of locator iterations
User function values	The values of all active user functions
Test function values	The values of all active test functions

`f` contains different information, depending on the continuation run. For noncycle-related continuations, the `f`-vector just contains the eigenvalues, if asked for. For limit cycle continuations, it begins with the mesh points of the time- discretization, followed by, if they were asked for, the PRC- and dPRC-values in all points of the periodic orbit (cf. section 7.8). Then, if required, follow the multipliers.

It is also possible to extend the most recently computed curve with the same options (also same number of points) as it was first computed. The syntax to extend this curve is:

```
[x, v, s, h, f] = cont( x, v, s, h, f, cds);
```

`x, v, s, h` and `f` are the results of the previous call to the continuer and `cds` is the global variable that contains the curve description of the most recently computed curve (note that this variable has to be defined as `global cds` in the calling command). The function returns the same output as before, extended with the new results.

In MatCont, all curves that have been computed using a specific system are stored in separate `.mat`-files, in a directory called *diagram*, under a subdirectory named after the system. For example, curves of the *Connor* system will be kept in `.mat`-files under the subdirectory `Systems/Connor/diagram/`. For continuation runs, each such `mat`-file contains the computed `x,v,s,h,f` arrays, plus the `cds` structure and a structure related to the curve type. Also, it contains the variables *point*, *ctype* and *num*. To understand their meaning, suppose that we are computing curves of limit cycles that we start from Hopf points. The first such computed curve then gets the name "H_LC(1)", *point* stores the string "H" and *ctype* stores the string "LC". Furthermore, *num* stores the index in `s` of the last selected point of the curve (the default is 1). The second curve of the same type is called "H_LC(2)" and so on. In fact, to save storage space, only a limited number of curves of a certain type is stored. This number can be set by the user and the default is 2. To save a computed curve permanently, the user must change its name.

For time integration runs, cf. §5.1 (Curve Type O), the `mat`-file contains `ctype`, `option`, `param`, `point`, `s`, `t`, `x`. Here `t` is the vector of time points and `x` is the corresponding array of computed points. `s` contains data on the first and last computed points. The meaning of `point,ctype` is similar to the case of continuation curves. Finally, `param` is the vector of parameters of the ODE (constant during time integration) and `option` is a structure that contains optional settings for time integration.

To export the computed results of a system to a different installation of MatCont one has to copy the corresponding `m`-file, the `mat`-file and the directory of the system.

These files also contain all information needed to export the computed results to the general MATLAB environment, so MatCont is really an open system.

MatCont also produces graphical output. 2D and 3D graphs are plotted in matlab figure windows. Such a graph can be handled as any other graph that is produced in matlab. It can be selected using the arrow-function of the matlab figure, and the line width, line style and colour can be altered. Markers can be set on the curve. It can be copy-pasted into another matlab figure. In a figure, textboxes can be inserted and axes labels can be added. Thus the user has a plethora of possibilities to combine different MatCont output graphs into one figure.

Finally, we note that users often want to introduce new systems that are modifications of existing systems, but with slightly different sets of state variables and/or parameters. The best strategy to do this in MatCont is first to edit the existing system, change its name to a new one and click "OK" to build an m-file with a different name and no associated directory of computed curves. Afterwards, one can edit the newly created system, make all desired changes and click "OK" again. For Cl_MatCont see §4

3.3 Curve file

The continuer uses a special m-file where the type of solution branch is defined. This file, further referred to as `curve.m`, contains the following sections:

- `curve_func`: contains the evaluation of the right-hand side of that type of solution branch.
- `defaultprocessor`: is called and executed after each point computed during a continuation experiment.
- `options`: sets the default setting for the options-structure for this type of solution branch (more details are given in section 3.4).
- `jacobian`: contains the evaluation of the jacobian of that type of solution branch.
- `hessians`: contains the evaluation of the hessians of that type of solution branch.
- `testf`: contains the test functions for detecting bifurcations along the branch.
- `userf`: calls the user-defined functions if there are any.
- `process`: is called when a bifurcation point is detected, to handle any necessary output messages and storage.
- `singmat`: defines the singularity matrix of the solution branch (cf. section 2.4.3).
- `locate`: here specific localisation functions can be defined for bifurcations (cf. section 2.4.4)
- `init`: handles any special initialisations needed in the parameters or workspace.
- `done`: handles any special actions needed at the end of continuing the solution branch.
- `adapt`: this is called after every n steps, where n is user-defined. It handles any adaptation of parameters, subspaces, etc.

3.4 Options

3.4.1 The options-structure

It is possible to specify various options for the continuation run. In the continuation we use the options structure which is initially created with contset:

```
options = contset;
```

will initialize the structure. The continuer stores the handle to the options in the variable `cds.options`. Options can then be set using

```
options = contset(options, optionname, optionvalue);
```

where `optionname` is an option from the following list:

InitStepsize the initial stepsize (default: 0.01)

MinStepsize the minimum stepsize to compute the next point on the curve (default: 10^{-5})

MaxStepsize the maximum stepsize (default: 0.1)

MaxCorrIters maximum number of correction iterations (default: 10)

MaxNewtonIters maximum number of Newton-Raphson iterations before switching to Newton-Chords in the corrector iterations (default: 3)

MaxTestIters maximum number of iterations to locate a zero of a testfunction (default: 10)

Increment the increment to compute first order derivatives numerically (default: 10^{-5})

MoorePenrose boolean indicating the use of the Moore-Penrose continuation as the Newton-like corrector procedure (default: 1)

FunTolerance tolerance of function values: $||F(x)|| \leq$ `FunTolerance` is the first convergence criterium of the Newton iteration (default: 10^{-6})

VarTolerance tolerance of coordinates: $||\delta x|| \leq$ `VarTolerance` is the second convergence criterium of the Newton iteration (default: 10^{-6})

TestTolerance tolerance of test functions (default: 10^{-5})

Singularities boolean indicating the presence of a singularity matrix (default: 0)

MaxNumPoints maximum number of points on the curve (default: 300)

Backward boolean indicating the direction of the continuation (sign of the initial tangent vector) v_0 (default: 0)

CheckClosed number of points indicating when to start to check if the curve is closed (0 = do not check) (default: 50)

Adapt number of points after which to call the adapt-function while computing the curve (default: 1=adapt always)

IgnoreSingularity vector containing indices of singularities which are to be ignored (default: empty)

Multipliers boolean indicating the computation of the multipliers (default: 0)

Eigenvalues boolean indicating the computation of the eigenvalues (default: 0)

TSearchOrder numerical value that indicates if unit vectors are cycled in increasing order of index (default: 1, increasing) or decreasing (set to a value different from 1), see §3.4.10.

Userfunctions boolean indicating the presence of user functions (default: 0)

UserfunctionsInfo is an array with structures containing information about the userfunctions. This structure has the following fields:

`.label`	label of the userfunction
`.name`	name of this particular userfunction
`.state`	boolean indicating whether the userfunction has to be evaluated or not

PRC variable indicating the computation of the phase response curve (default: empty)

dPRC variable indicating the computation of the derivative of the phase response curve (default: empty)

Input vector representing the input given to the system for the computation of the phase response curve (default: 0)

This list is stored in the file `contidx.m` in the directory `Continuer`. However, `options` also contains fields which are not set by the user but frozen or filled by calls to the curvefile, namely:

SymDerivative the highest order symbolic derivative which is present (default: 0)

SymDerivativeP the highest order symbolic derivative with respect to the free parameter(s) which is present (default: 0)

Testfunctions boolean indicating the presence of test functions (default: 0)

WorkSpace boolean indicating to initialize and clean up user variable space (default: 0)

Locators boolean vector indicating the user has provided his own locator code to locate zeroes of test functions. Otherwise the default locator will be used (default: empty)

ActiveParams vector containing indices of the active parameter(s) (default: empty)

ActiveUParams

ActiveSParams

ActiveSParam

The last three fields are used only in the homotopy methods for the initialzation of connecting orbits.

Some more details follow now.

3.4.2 Derivatives of the defining system of the curve

In the defining system of the object that is to be continued, the derivates can be provided that are needed for the continuation algorithm or other computations. The continuer stores the handle to the derivatives in the variables `cds.curve_jacobian`,`cds.curve_hessians`.

If `cds.symjac`$= 1$, then a call to `feval(cds.curve_jacobian, x)` must return the $(n-1) \times n$ Jacobian matrix evaluated at point x.

If `cds.symhess`$= 1$, then a call to `feval(cds.curve_hessians, x)` must return a 3-dimensional $(n-1 \times n \times n)$ matrix H such that $H(i,j,k) = \frac{\partial^2 F_i(x)}{\partial x_j \partial x_k}$.

In the present implementation in most cases `cds.symhess`$= 0$, so the ODE-file does not provide second order derivatives, since they are not needed in the algorithms used.

3.4.3 Singularities and test functions

To detect singularities on the curve one must set the option *Singularities* on. Singularities are defined using the singularity matrix, as described in section 2.4.3. The continuer stores the handles to the singularities, the testfunctions and the processing of the singularities respectively in the variables `cds.curve_singmat`,`cds.curve_testf` and `cds.curve_process`.

A call to `[S,L] = feval(cds.curve_singmat)` gets the singularity matrix S and a vector of 2-character strings which are abbreviations of the singularities.

A call to `feval(cds.curve_testf, ids, x, v)` then must return the evaluation of all testfunctions, whose indices are in the integer vector `ids`, at `x` (`v` is the tangent vector at `x`). As a second return argument it should return an array of all testfunction id's which could not be evaluated. If this array is not empty the stepsize will be decreased.

When a singularity is found, a call to `[failed,s] = feval(cds.curve_process,i,x,v,s)` will be made to process singularity `i` at `x`. This is the point where computations can be done, like computing normal forms, eigenvalues, etc. of the singularity. These results can then be saved in the structure `s.data` which can be reused for further analysis. Note that the first and last point of the curve are also treated as singular.

3.4.4 Locators

It may be useful to have a specific locator code for locating certain singularities (cf. section 2.4.4). To use a specific locator you must set the option *Locators*. This is a vector in which the index of an element corresponds to the index of a singularity. Setting the entry to 1 means the presence of a user-defined locator. The continuer has stored the handles to the locators in the variable `cds.curve_locator` and will then make a call to
`[x,v]=feval(cds.curve_locate,i,x1,v1,x2,v2)`
to locate singularity i which was detected between `x1` and `x2` with their corresponding tangent vectors `v1` and `v2`. It must return the located point and the tangent vector at that point. If the locator was unable to find a point it should return `x = []`.

3.4.5 User functions

To detect zeros of userfunctions on the curve one must set the option *Userfunctions* on. The continuer has stored the handles to the userfunctions `cds.curve_userf`. First a call to `UserInfo = contget(cds.options, 'UserfunctionsInfo', [])` is made to get information on the userfunctions. A call to `feval(cds.curve_userf, UserInfo, ids, x, v)` then must return the evaluation of all userfunctions ids, whose information is in the structure `UserInfo`, at `x` (`v` is the tangent vector at `x`). As a second return argument it should return an array of all user function id's which could not be evaluated. If this array is not empty the stepsize will be decreased.

A special point on a bifurcation curve that is specified by a user function has a structure as follows:

`s.index`	index of the detected singular point defined by the user function.
`s.label`	a string that is in `UserInfo.label`, label of the singularity.
`s.msg`	a string that is set in `UserInfo.name`.
`s.data`	an empty tangent vector or values of the user functions in the singular point.

When a change of sign of a userfunction is detected, the userfunction `i` is processed at `x`. This is the point where the results (values of the userfunction) can be saved in the structure `s.data` which can be reused for further analysis.

3.4.6 Defaultprocessor

In many cases it is useful to do some general computations for every calculated point on the curve. The results of these computations can then be used by for example the test-functions. The continuer has stored the handle to the defaultprocessor in the variable `cds.curve_defaultprocessor`.

The defaultprocessor is called as
`[failed,f,s] = feval(cds.curve_defaultprocressor,x,v,s)`.
`x` and `v` are the point on the curve and it's tangent vector. The argument `s` is only supplied if the point is a singular point, in that case the defaultprocessor may also add some data to the `s.data` field. If for some reason the default processor fails it should set `failed` to 1. This will result in a reduction of the stepsize and a retry which should solve the problem. Any information that is to be preserved, should be put in `f`. `f` must be a column vector and must be of equal size for every call to the default processor.

3.4.7 Special processors

After a singular point has been detected and located a singular point data structure will be created and initialized as described in section 3.2. If there are some special data (like eigenvalues) which may be of interest for a particular singular point then a call to `[failed,s] = feval(cds.curve_process,i,x,v,s)` should store this data in the `s.data` field. Here `i` indicates which singularity was detected and `x` and `v` are the point and tangent vector where this singularity was detected.

3.4.8 Workspace

During the computation of a curve it is sometimes necessary to introduce variables and do additional computations that are common to all points of the curve. The continuer has stored the handle to the initialization and cleaning of the workspace in the variables `cds.curve_init` and `cds.curve_done`. These can be relegated to a call of the type

```
feval(cds.curve_init,x,v).
```

This option has to be provided only if the variable `WorkSpace` in `cds.options` is switched on. In this case a call

```
feval(cds.curve_done,x,v)
```

must clear the workspace. Variables in the workspace must be set global.

3.4.9 Adaptation

It is possible to adapt the problem while generating the curve. If *Adapt* has a value, say 5, then after 5 computed points a call to `[reeval,x,v]=feval(cds.curve_adapt,x,v)` will be made where the user can program to change the system.

For some applications it is useful to change or modify the used test functions while computing the curve (like in bordering techniques). In order to preserve the correct signs of the test functions it is sometimes necessary to reevaluate the test functions after adaptation. To do this `reeval` should be one, otherwise zero. The return variables `x` and `v` should be the updated `x` and `v` which may have changed because of the changes made to the system.

3.4.10 Tangent search order

To start a continuation, an initial point x_0 and a tangent vector v_0 are needed in general. Often, only x_0 is available. In this case, MatCont successively tries all unit vectors as candidate tangent vectors. By default, this is done in increasing order of index (cds.options.TSearchOrder = 1). If cds.options.TsearchOrder is set to a value different from 1 then the cycling is done in decreasing order of index. See §7.4 for an example.

In cases where the number of continuation variables is large (e.g. when computing limit cycles) the choice of cds.options.TSearchOrder can substantially change the speed of the computation.

3.4.11 Summary

In the following table we list calls that can be made to the continuation curve description `cds` and which options are involved.

Syntax of call	What it should do (options involved)
`feval(cds.curve_func,x)`	return F(x)
`feval(cds.curve_options)`	return option vector
`feval(cds.curve_jacobian,x)`	return Jacobian at `x` (*SymDerivative*≥ 1)
`feval(cds.curve_hessians,x)`	return Hessians at `x` (*SymDerivative*≥ 2)
`feval(cds.curve_init,x,v)`	initialize user variable space (*WorkSpace*)
`feval(cds.curve_done)`	destroy user variable space (*WorkSpace*)
`feval(cds.curve_defaultprocessor,x,v,s)`	initialize data for testfunctions and set some general singularity data
`feval(cds.curve_testf,ids,x,v)`	return evaluation of testfunctions `ids` at `x` (*Singularities*)
`feval(cds.curve_locate,i,x1,x2,v1,v2)`	return located singularity and tangent vector(*Locators*)
`feval(cds.curve_userf,UserInfo,ids,x,v)`	return evaluation of userfunctions ids with `UserInfo` at `x` (*Userfunctions*)
`feval(cds.curve_singmat)`	return singularity matrix (*Singularities*)
`feval(cds.curve_process,i,x)`	run processor code of singularity `i` at `x`(*Singularities*)
`feval(cds.curve_adapt,x,v)`	run adaptation code of problem (*Adapt*)

3.5 Failure handling

During the continuation numerical problems may arise. For example a linear system in the Newton corrections could be ill conditioned. In such cases the continuer checks for the last warning issued by MATLAB using `lastwarn()` and decreases the step size along the curve. The same mechanism is used when a test function or a user function cannot be computed.

3.6 General remarks on the data flow

At this point we have discussed two components of a continuation process, the continuer itself and the curve definition. In Figure 4 the complete structure is visualized. The arrows show the flow of information between the objects, where an arrow from object A to object B indicates that information present in A is sent to B, typically by a call from B to A. The information is sometimes passed via a function call but in many cases via a global structure. Global structures are discussed in §3.8.

As one can see, two extra components are included: the *curve initializer* and some external ODE file.

Continuation of curves with complicated curve definitions often needs to be initialized. Since the continuer is called only with the start point x_0 and an options structure (and sometimes, but not always v_0) there must be some way to initialize other parameters. Calling an initializer from a GUI or command prompt solves this problem. The interaction between initializer and continuer is “invisible” since it passes through a global structure called `cds` (continuation descriptor structure), see §3.8. One important field is `cds.symjac` which informs the continuer whether or not the curve definition file includes the Jacobian of the curve definition function. See also the note at the end of §A.

The standard MATLAB `odeget` and `odeset` only support Jacobian matrices coded in the

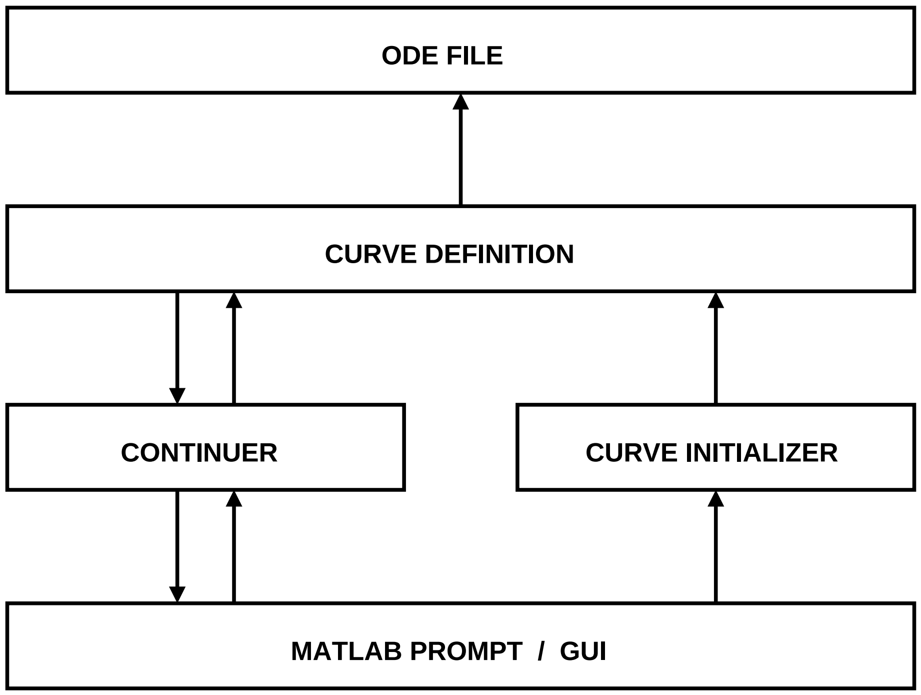


Figure 4: Structure of continuation process

ode-file. However, we do need the derivatives with respect to the parameters. It is also useful to have higher-order symbolic derivatives available.

To overcome this problem, the package contains new versions of `odeget` and `odeset` which support Jacobians with respect to parameters and higher-order derivatives. The new routines are compatible with the ones provided by MATLAB.

To include the Jacobian with respect to parameters, the option *JacobianP* should contain the handle of the subfunction jacobianp *@jacobianp*. A call to `feval(@jacobianp, 0, x, p1, p2, ...)` should then return the Jacobian with respect to to parameter $p_1, p_2, \ldots$.

To include Hessians in the ode-file the option *Hessians* should contain the handle of the subfunction hessians *@hessians*. The software then assumes that a call to `feval(@hessians, 0, x, p1, p2, ...)` will return all Hessians in the same way as mentioned above. Setting the option to *[]* indicates that there are no Hessians available from the ode-file (default behaviour).

To include Hessians with respect to parameters in your ode-file the option *HessiansP* should contain the handle of the subfunction hessiansp *@hessiansp*. The software then assumes that a call to `feval(@hessiansp, 0, x, p1, p2, ...)` will return all Hessians with respect to parameters in the same way as mentioned above. Setting the option to *[]* indicates that there are no Hessians with respect to parameters available from the ode-file (default behaviour).

To include the third order derivatives in your ode-file the option *Der3* should contain the handle of the subfunction der3 *@der3*. The software then assumes that a call to `feval(@der3, 0, x, p1, p2, ...)` will return all third order derivatives in the same way as mentioned above. Setting the option to *[]* indicates that they are not available from the ode-file (default behaviour)

Der4 and *Der5* are values indicating the 4th and 5th order symbolic derivative, available in the ode-file.

3.7 Directories

The files of the toolbox are organized in the following 23 directories

- BranchPoint
  Here are all files stored needed to do a branch point continuation. This includes the initializers and a branch point curve definition file. BranchPoint and BranchPointCycle are the only continuation curve types with three free system parameters in MatCont.

- BranchPointCycle
  Here are all files stored needed to do a branch point of cycles continuation. This inluded the initializers and branch point of cycles definition files. BranchPoint and BranchPointCycle are the only continuation curve types with three free system parameters in MatCont.

- Continuer
  Here are all the main files stored for the continuer, which are needed to calculate and plot any curve.

- Equilibrium
  Here are all files stored needed to do an equilibrium continuation. This includes the initializers and the equilibrium curve definition file.

- GUI
  This directory contains all GUI-related files. However, some files in this directory are also used in Cl_MatCont.

- Help
  Contains the help-files.

- Heteroclinc
  Here are all files stored needed to do a continuation of heteroclinic orbits. This includes the initializers and the curve definition file.

- Homoclinic
  Here are all files stored needed to do a homoclinic-to-hyperbolic-saddle continuation. This includes the initializers and the curve definition file.

- HomoclinicSaddleNode
  Here are all files stored needed to do a homoclinic-to-saddle-node continuation. This includes the initializers and the curve definition file.

- HomtopyHet
  Here are all files stored needed to find a heteroclinic connection by the homotopy method.

- HomotopySaddle
  Here are all files stored needed to find an orbit homoclinic to saddle by the homotopy method.

- HomotopySaddlenode
  Here are all files stored needed to find an orbit homoclinic to saddle-node by the homotopy method.

- Hopf
  Here are all files stored needed to do a Hopf point continuation. This includes the initializers and the Hopf point curve definition file.

- LimitCycle
  Here are all files stored needed to do a limit cycle continuation. This includes the initializers and the limitcycle curve definition file.

- LimitCycleCodim2
  This directory contains the routines that compute the normal form coefficients of codim2 bifurcations of limit cycles. Computing these coefficients reliably requires that computations are done to high precision and symbolic derivatives up to fifth order are used.

- LimitPoint
  Here are all files stored needed to do a limitpoint continuation. This includes the initializers and the limitpoint curve definition file.

- LimitPointCycle
  Here are all files stored needed to do a fold bifurcation of limit cycles continuation. This includes the initializers and limitpoint of cycles curve definition files.

- MultiLinearForms
  Here are files stored needed to compute high-order derivatives of systems and normal-form coefficients of bifurcations.

- NeimarkSacker
  Here are all files stored needed to do a torus bifurcation of limit cycles continuation. This includes the initializers and torus curve definition files.

- PeriodDoubling
  Here are all files stored needed to do a period doubling bifurcation continuation. This includes the initializers and perioddoubling curve definition files.

- SBML
  This directory contains the routines that allow to import SBML (Systems Biology Markup Language) systems into MatCont. These routines require an update and/or revision.

- Systems
  Here all system definition files and files related to computed curves are stored, except those odefiles which are hard-stored in the directory Testruns/TestSystems because these are used in the testruns. To avoid confusion, it is not recommended to use the names of the odefiles that are present in Testruns/TestSystems.

- Testruns
  Here are some example testruns stored which can be executed from the MATLAB command line. They can be used to run the examples described in this manual and to test if everything is working correctly. The necessary odefiles are stored in the subdirectory TestSystems.

We note that 12 directories are dedicated to a specific curve type, namely:
`Equilibrium`, `LimitCycle`, `LimitPoint`, `Hopf`,
`LimitPointCycle`, `PeriodDoubling`, `NeimarkSacker`, `BranchPoint`,
`BranchPointCycle`, `Homoclinic`, `Heteroclinic` and `HomoclinicSaddleNode`.

Three other directories, namely `HomotopyHet`, `HomotopySaddle` and `HomotopySaddleNode` are initialization directories to a specific curve type.

The remaining 8 directories are not dedicated to a specific curve type.

The only files which are not in any of these directories are `init.m`, `cpl.m`, `matcont.fig` and `matcont.m`. The function `init` must be called before using the command-line toolbox so that MATLAB can find all the needed functions. `cpl` is used to plot the results of the continuation in CL_MatCont (for details see §3.9). `matcont.m` is the start-up file of the GUI version MatCont, and `matcont.fig` is the related figure-file.

3.8 Global Structures

In general the user does not need to know much about the use of global structures in MatCont but advanced users can find it useful since a lot of additional information can be hidden in these structures. MatCont uses a structure `cds` (continuation descriptor structure) which is global in the continuer routine `cont.m` and in all (12) curve definition files and it carries information back and forth between them (cf. Figure 4). Also, `cds` has a field `cds.options` with the fields described in §3.4. It contains default values for these options but overrides them with the values in the options-structure that is passed with the call of `cont.m`. Other fields of `cds.options` are filled via the interaction with the curve definition file which is also passed with the call to `cont.m`

Next, MatCont uses more specific global structures, namely:

- `eds` (equilibrium descriptor structure): this structure is global in the curve definition file `equilibrium.m` and in the initializers for the continuation of equilibria.
- `lpds` (limit point descriptor structure): this structure is global in the curve definition file `limitpoint.m` and in the initializers for the continuation of limit points.
- `hds` (Hopf descriptor structure): this structure is global in the curve definition file `hopf.m` and in the initializers for the continuation of Hopf points.
- `bpds` (branch point descriptor structure): this structure is global in the curve definition file `branchpoint.m` and in the initializers for the continuation of branch points.
- `lds` (limitcycle descriptor structure): this structure is global in the curve definition files `limitcycle.m`, `limitpointcycle.m`, `perioddoubling.m`, `neimarksacker.m`, and also in `branchpointcycle.m`, in the initializers for the continuation of limit cycles, their codimension 1 bifurcations and branch points.

- `homds` (homoclinc descriptor structure): this structure is global in the curve definition files `homoclinic.m`, `homoclinicsaddlenode.m` and in the initializers for the continuation of orbits to saddle or to saddle-node.
- `hetds` (heteroclinc descriptor structure): this structure is global in the curve definition file `heteroclinic.m` and in the initializers for the continuation of heteroclinic orbits.

These specific structures carry information that is collected or computed in the initializers to the curve definition files in which they are global. In the initializers themselves other specific structures can also be global, depending on the data that are used in the initializer. For example, in the initializer `init_H_LC` not only `lds` and `cds` are global but also `eds` and `hds`. Indeed, Hopf points can be detected on equilibrium curves or be taken from Hopf curves.

3.9 Plots in MatCont

When MatCont is handled from the command line, then of course all MATLAB plot functionalities can be used. However, MatCont provides two specific plot functions, namely `cpl.m` and `plotcycle.m`.

`cpl(x, v, s, e)` makes a two or three dimensional plot. This routine automatically places labels at the singularity points.The first three arguments must be the output of a continuation run. The fourth argument `e` is optional. `e` is an array whose elements define which coordinates of the system are used. Therefore `e` must have either 2 components (2D-plot) or 3 components (3D-plot). If e is not given and `x` has 2 (respectively 3) components, then a 2D-plot (3D-plot,respectively) is drawn. In all other cases an error message will be generated.

The use of `plotcycle` is discussed in §7.3.

4 The odefile of a dynamical system

4.1 Structure and construction of an odefile

A solution curve must be initialized before doing a continuation. Each curve file has its own initializers and in the case of dynamical systems the initializers typically use an *odefile* where the ode is defined. An *odefile* (short for *system definition file*) contains at least the following sections:
init, *fun_eval*, *jacobian*, *jacobianp*, *hessians*, *hessiansp*, *der3*, *der4* , *der5*.
An *odefile* may also contain one or more sections that describe user functions.

There is a variety of options to create such odefiles. In all cases users should avoid using loops, conditional statements, or other specific programming constructions in the system definition.

First, an *odefile* can be defined by simply using the MATLAB editor (or, in fact, any text editor). This is likely to lead to errors and therefore not encouraged if the MATLAB symbolic toolbox is available and symbolic derivatives are desirable.

A second option is to use the GUI version of MATCONT by defining the problem in the 'System' window and then choosing the options "symbolically" or "numerically" to use symbolic or numeric (= finite difference) derivatives, respectively. The user functions can also be added in the GUI of MATCONT, using the menu 'User function'.

From version 6.7 on MATCONT provides another possibility to create odefiles. It is in fact a shortcut to using the GUI version of MATCONT. As an example we will study the *Rössler chaotic system*:

$$\begin{cases} \dot{x} &= -y - z \\ \dot{y} &= x + Ay \\ \dot{z} &= Bx - Cz + xz, \end{cases}$$

where (x, y, z) are the phase variables, and (A, B, C) are the parameters.

An *odefile* can be created by calling `SysGUI.new`.

This opens a **System** window, which contains several fields and buttons. To identify the system, type for example

```
Roessl
```

in the **Name** field (it must be one word).

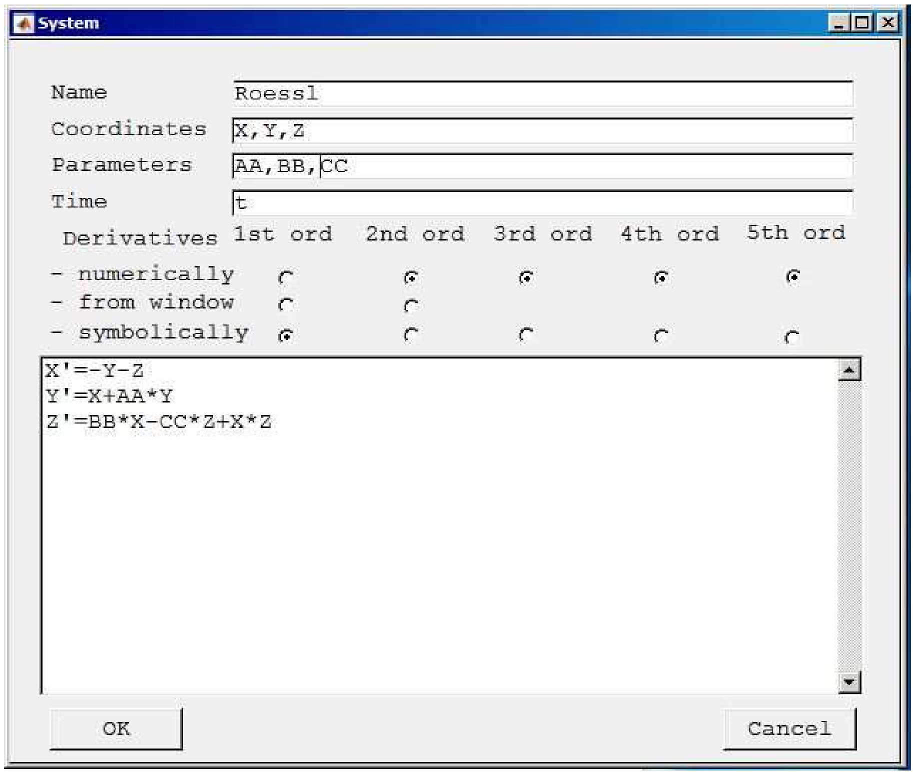
Input names of the **Coordinates**: `X,Y,Z`, and the **Parameters**: `AA,BB,CC`.

If shown, select symbolic generation of the 1st order derivatives by pressing the corresponding radio-button [1].

Finally, in the large input field, type the RHS of the truncated normal form map as

```
X'=-Y-Z
Y'=X+AA*Y
Z'=BB*X-CC*Z+X*Z
```

[1]If the MATLAB Symbolic Toolbox is present, there will be buttons indicated 'symbolically'. The first-order derivatives are used in some of the integration algorithms, the first- and second-order derivatives are used in the continuation, while the third-order derivatives are employed in the normal form computations. The derivatives of fourth and fifth order are only used in the normal form computations of some codimension 2 bifurcations.



Name: Roessl

Coordinates: X,Y,Z

Parameters: AA,BB,CC

Time: t

Derivatives	1st ord	2nd ord	3rd ord	4th ord	5th ord
- numerically	○	●	●	●	●
- from window	○	○			
- symbolically	●	○	○	○	○

```
X'=-Y-Z
Y'=X+AA*Y
Z'=BB*X-CC*Z+X*Z
```

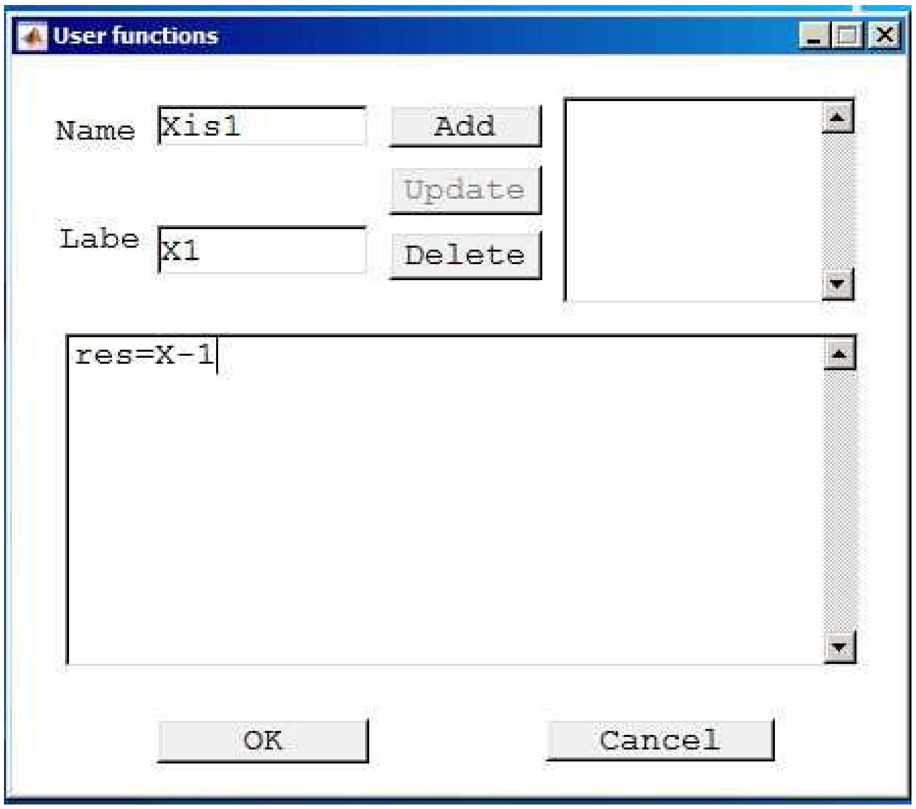
Figure 5: Specifying a new model.

Avoid typical mistakes:

- Make sure the multiplication is written explicitly with $*$.
- Specify the right hand sides in the same order as the coordinates.

It is best not to add comma's or semicolons after the equations. Now the **System** window should look like in Figure 5, and you can press the **OK** button. Two new files will be created in the `Systems` directory of MatCont, namely the *odefile* `Roessl.m` and a `mat`-file `Roessl.mat`.

The *odefile* can be edited later on by calling `SysGUI.edit(@name)` where `name` is the name of an existing *odefile*. User functions can be added by calling `SysGUI.userfunctions(@name).` See Figure 6.



Name: Xis1
Labe: X1

```
res=X-1
```

Figure 6: Adding a user function.

We note that if state variables, parameters or user functions are added or deleted then this constitutes another dynamical system. So either all computed data should be deleted or ignored, or the name of the system should be changed. The last option is recommended, in particular when the GUI is used.

A special point on a bifurcation curve that is specified by a user function has a structure as follows:

s.index	index of the detected singular point defined by the user function.
s.label	a string that is in `UserInfo.label`, label of the singularity.
s.data	an empty tangent vector, values of the test and user functions in the singular point.
s.msg	a string that is set in `UserInfo.name`.

We now give an *odefile* for the *Rössler* system using symbolic derivatives of all orders up to three and including the userfunction $x - 1$. Note that we have returned to state variables denoted by x, y, z and parameters A, B, C.

```
function out = Roessl
out{1} = @init;
out{2} = @fun_eval;
out{3} = @jacobian;
out{4} = @jacobianp;
out{5} = @hessians;
out{6} = @hessiansp;
out{7} = @der3;
out{8} = [];
out{9} = [];
out{10}= @xis1;

% --------------------------------------------------------------------------
function dydt = fun_eval(t,kmrgd,A,B,C)
dydt=[-kmrgd(2)-kmrgd(3);
kmrgd(1)+A*kmrgd(2);
B*kmrgd(1)-C*kmrgd(3)+kmrgd(1)*kmrgd(3);];

% --------------------------------------------------------------------------
function [tspan,y0,options] = init
y0=[0,0,0];
options = odeset('Jacobian',handles(3),'JacobianP',handles(4),...
                 'Hessians',handles(5),'HessiansP',handles(6));
handles = feval(Roessl);
tspan = [0 10];

% --------------------------------------------------------------------------
function jac = jacobian(t,kmrgd,A,B,C)
jac=[ 0 , -1 , -1 ; 1 , A , 0 ; B + kmrgd(3) , 0 , kmrgd(1) - C ];
% --------------------------------------------------------------------------
function jacp = jacobianp(t,kmrgd,A,B,C)
jacp=[ 0 , 0 , 0 ; kmrgd(2) , 0 , 0 ; 0 , kmrgd(1) , -kmrgd(3) ];
% --------------------------------------------------------------------------
function hess = hessians(t,kmrgd,A,B,C)
hess1=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 1 ];
hess2=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
hess3=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 1 , 0 , 0 ];
hess(:,:,1) =hess1;
hess(:,:,2) =hess2;
hess(:,:,3) =hess3;
% --------------------------------------------------------------------------
function hessp = hessiansp(t,kmrgd,A,B,C)
hessp1=[ 0 , 0 , 0 ; 0 , 1 , 0 ; 0 , 0 , 0 ];
hessp2=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 1 , 0 , 0 ];
hessp3=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , -1 ];
hessp(:,:,1) =hessp1;
hessp(:,:,2) =hessp2;
hessp(:,:,3) =hessp3;
%---------------------------------------------------------------------------
function tens3  = der3(t,kmrgd,A,B,C)
tens31=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens32=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens33=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens34=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens35=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens36=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens37=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens38=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens39=[ 0 , 0 , 0 ; 0 , 0 , 0 ; 0 , 0 , 0 ];
tens3(:,:,1,1) =tens31;
tens3(:,:,1,2) =tens32;
tens3(:,:,1,3) =tens33;
tens3(:,:,2,1) =tens34;
tens3(:,:,2,2) =tens35;
tens3(:,:,2,3) =tens36;
tens3(:,:,3,1) =tens37;
tens3(:,:,3,2) =tens38;
tens3(:,:,3,3) =tens39;
%---------------------------------------------------------------------------
function tens4  = der4(t,kmrgd,A,B,C)
%---------------------------------------------------------------------------
function tens5  = der5(t,kmrgd,A,B,C)
function userfun1=xis1(t,kmrgd,A,B,C)
userfun1=kmrgd(1)-1;
```

We observe the following:

- The state variables are collected in a vector called `kmrgd`.
- Internally the names of the parameters are extended so that, e.g., "C" is replaced by "par_C". This is done to avoid clashes with the symbolic toolbox in which certain names are protected. This does not affect the use of the odefile since the parameters are not referred to by name. If the user writes his own odefile then this precaution is not necessary.

- The init function is called when performing a time integration in the GUI mode of MatCont. It is of no use in the command line version. See §5.1.3 for a command line example.

4.2 Handling auxiliary functions in the construction of the odefile

If complicated expressions arise several times in the definition of a dynamical system then is often useful to give them a name. A simple example is the following system (a catalytic oscillator, (60):

$$\begin{cases} \dot{x} &=& 2q_1z^2 - 2q_5x^2 - q_3xy \\ \dot{y} &=& q_2z - q_6y - q_3xy \\ \dot{s} &=& q_4z - kq_4s \end{cases} \tag{31}$$

where $z = 1 - x - y - s$. When `SysGUI.new` is used to create the *odefile* of the system then is not necessary to "expand" the auxiliary function for z. The system can be introduced as

```
z=1-x-y-s
x'=2*q1*z^2-2*q5*x^2-q3*x*y
y'=q2*z-q6*y-q3*x*y
z'=q4*z-k*q4*s
```

We note that the definition of the auxiliary functions must precede the definiton of the right hand sides.

The corresponding odefile is `cataloscill.m` in the directory `Testruns/TestSystems.`

4.3 Access to function and Jacobian values

Once an odefile is created, it can be used to compute the values of the odefunction and its derivatives with respect to state variables and parameters on the command line. As an example, consider the Morris-Lecar system `MyML` in the directory `Testruns/TestSystems`. This system has two state variables and two parameters. We compute the odefunction values and derivatives with respect to the state vaariables and parameters for the state vector $[0; 0]$ and parameters (30,10) by executing the script `testodefile`, available in the directory `Testruns`:

```
MyMLSystem=MyML
MyMLfunction=MyMLSystem{2}
MyMLjacobian=MyMLSystem{3}
MyMLjacobianp=MyMLSystem{4}
MyMLfunction(0,[0;0],30,10)
MyMLjacobian(0,[0;0],30,10)
MyMLjacobianp(0,[0;0],30,10)
```

The output is as follows:

```
MyMLSystem =

   1 x 9 cell array
```

```
{@init} {@fun_eval}  {@jacobian} {@jacobianp} {@hessians}
{@hessiansp} {0x0 double} {0x0 double} {0x0 double}

MyMLfunction =

  function_handle with value:

    @fun_eval


MyMLjacobian =

  function_handle with value:

    @jacobian


MyMLjacobianp =

  function_handle with value:

    @jacobianp


ans =

   33.1953
    0.0167


ans =

    1.8282 -128.0000
    0.0013   -0.0694


ans =

    0.2000         0
         0   -0.0013
```

4.4 Time integration using the odefile

Time integration (= simulation) of the dynamical system can be done on the command line as in the following example:

```
OPTIONS=[];
```

```
hls = MyML;
[t,y] = ode45(hls{2},[0 1000],[0 0],OPTIONS,30,10);
x0 = y(end,:)';
```

In this example the system MyML is integrated over the interval $[0, 1000]$ starting with the input vector $[0, 0]$ and with parameters $30, 10$ in that order. The integration is performed with the integration routine ode45 with the defaults (OPTIONS=[]) in MATLAB. The output vector is placed in x0. More details are given in Chapter 5.

5 Time integration and Poincaré maps

MATLAB provides a suite of ODE-solvers. To create a combined integration-continuation environment, we have made them all accessible in MatCont and added two new ones, `ode78` and `ode87`. `ode78` is an explicit Runge-Kutta method that uses 7th order Fehlberg formulas [Fehlberg 1969]. This is a 7-8th-order accurate integrator, therefore the local error normally expected is $O(h^9)$. `ode87` is a Runge-Kutta method that uses 8-7th order Dorman and Prince formulas. See [Prince and Dorman 1981]. This is a 8th-order accurate integrator therefore the local error normally expected is $O(h^9)$. In MatCont the choice of the solver is made via the Integrator window that is opened automatically when the curve type O (Orbit) is chosen. We note that the Starter window has a SelectCycle button that allows to start the continuation of periodic orbits from curves computed by time integration.

5.1 Time integration

MatCont uses the MATLAB representation of ODEs, which can also be used by the MATLAB ODE solvers. The user of Cl_MatCont is therefore also able to use his/her model within the standard ODE solver without rewriting the code. In MatCont, also backward time integration is possible using the standard ODE solvers. To make them accessible in MatCont, some output functions and properties were created.

5.1.1 Solver output properties

The solver output properties allow one to control the output that the solvers generate. MatCont detects whether extra output is needed by looking at the number of open output windows (2D, 3D or numeric). If extra output is required, MatCont sets the OutputFcn property to the output function, `integplot` which is passed to an ODE solver by `options = odeset('OutputFcn', @integplot)`, otherwise it is set to the function `integ_prs`. The output function must be of the form `status = integplot(t, y, flag,` p_1`,` p_2`,...)`. The solver calls this function after every successful integration step with the following flags. Note that the syntax of the call differs with the flag. The function must respond appropriately:

- `init`: The solver calls `integplot(tspan,y0,'init')` before beginning the integration, to allow the output function to initialize. This part initializes the output windows and does some initializations to speed up the further processing of the output. It also launches the window that makes it possible to interactively "stop/pause/resume" the computations.

- within MatCont it is possible to define the number of points (*npoints*) after which output is needed. Because the solver calls `status = integplot (t,y)` after each integration step, the number of calls to `integplot` and *npoints* does not correspond. Therefore this part is divided into two parts. `t` contains points where output was generated during the step, and `y` is the numerical solution at the points in `t`. If t is a vector, the *i*-th column of `y` corresponds to the *i*-th element of `t`. The output is produced according to the Refine option. `integplot` must return a status output value of 0 or 1. If *status* $= 1$, the solver halts integration. This part also handles the "stop/pause/resume" interactions.

- `done`: The solver calls `integplot([],[],'done')` when integration is completed to allow the output function to perform any cleanup chores. The stop-window is also deleted.

Setting the OutputFcn property to the output function, `integ_prs`, reduces the output time. It only allows to interactively pause, resume and stop the integration.

5.1.2 Jacobian matrices

Stiff ODE solvers often execute faster if the Jacobian matrix, i.e. the matrix of partial derivatives of the RHS that defines the differential equations, is provided. The Jacobian matrix pertains only to those solvers for stiff problems (`ode15s, ode23s, ode23t, ode23tb`), for which it can be critical for reliability and efficiency. If the user does not provide a code to calculate the Jacobian matrix, these solvers approximate it numerically using finite differences. Supplying an analytical Jacobian matrix often increases the speed and reliability of the solution for stiff problems. If the Jacobian matrix is provided (symbolically) in the odefile, MatCont stores as property the function handle of the Jacobian, otherwise this handle is set empty.

The standard matlab `odeget` and `odeset` only support Jacobian matrices of the RHS w.r.t. the state variables. However, we do need derivatives with respect to the parameters for the continuation. To compute normal form coefficients, it is also useful to have higher-order symbolic derivatives. To overcome this problem, MatCont contains new versions of `odeget` and `odeset`, which support Jacobian matrices with respect to parameters and higher-order partial derivatives w.r.t state variables. The new routines are compatible with the ones provided by matlab.

5.1.3 The Steinmetz-Larter example

In the peroxidase-oxidase reaction a peroxidase catalyzes an aerobic oxidation:

$$2YH_2 + O_2 + 2H^+ \rightarrow 2YH^+ + 2H_2O,$$

where YH_2 ($NADH$) is a general electron donor. Under the proper conditions this reaction yields oscillations in the concentrations of YH_2, O_2 and various reaction intermediates. Steinmetz and Larter [Steinmetz and Larter 1991] derived the following system of four coupled nonlinear differential rate equations:

$$\begin{cases} \dot{A} &= -k_1ABX - k_3ABY + k_7 - k_{-7}A, \\ \dot{B} &= -k_1ABX - k_3ABY + k_8, \\ \dot{X} &= k_1ABX - 2k_2X^2 + 2k_3ABY - k_4X + k_6, \\ \dot{Y} &= -k_3ABY + 2k_2X^2 - k_5Y, \end{cases} \tag{32}$$

where A, B, X, Y are state variables and k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8, and k_{-7} are parameters. Although highly simplified, the model is able to reproduce the three modes of simple, chaotic, and bursting oscillations found in experiments. State and parameter values of a point on a NS cycle in (32) are given in Table 4. The normal form coefficient of the NS cycle is $-1.406017e-006$. Since it is negative, we expect stable tori nearby.

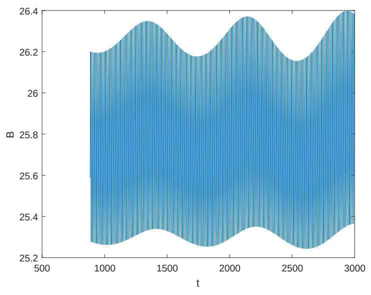
To start a time integration from this NS point, with a slightly supercritical parameter value, namely $k_7 = 0.7167$, we can use the commands

Variable	*Value*	*Parameter*	*Value*	*Parameter*	*Value*
A	1.8609653	k_1	0.1631021	k_5	1.104
B	25.678306	k_2	1250	k_6	0.001
X	0.010838258	k_3	0.046875	k_7	0.71643356
Y	0.094707061	k_4	20	k_{-7}	0.1175
				k_8	0.5

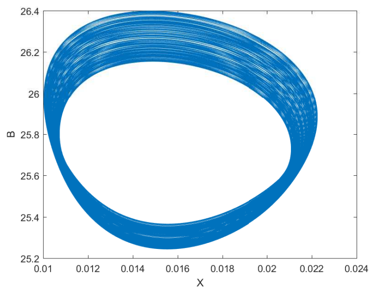
Table 4: State and parameter values of a point on an NS- cycle in the Steinmetz-Larter model.

```
X0=[1.8609653;25.678306;0.010838258;0.094707061];
P0=[0.1631021;1250;0.046875;20;1.104;0.001;0.71643356;0.1175;0.5];
P0(7)=0.7167;
hls=feval(@SteLar);
options=odeset('RelTol',1e-8);
[t,y]=ode45(hls{2},[0,3000],X0,options,0.1631021,1250,0.046875,20,1.104,
0.001,0.7167,0.1175,0.5);
size(t)
size(y)
plot(t(100000:339000),y(100000:339000,2));
'press any key'
pause
plot(y(100000:339000,3),y(100000:339000,2));
```

After a transient the time-series exhibits modulated oscillations with two frequencies near the original limit cycle (see Fig. 7). This is a motion on a stable two-dimensional torus that arises from the Neimark-Sacker bifurcation. The commands can be executed by running the testrun `testtimeinteg.m`. The total number of computed points is 339161.



(a) Modulated oscillations.



(b) Orbits on a stable 2-torus.

Figure 7: Movement on a stable torus in a 4-dimensional space as a function of time in the $B-$ direction and as projected on the (B, X) phase plane.

The testrun `testtimeintegJacobian.m`

```
X0=[1.8609653;25.678306;0.010838258;0.094707061];
P0=[0.1631021;1250;0.046875;20;1.104;0.001;0.71643356;0.1175;0.5];
P0(7)=0.7167;
hls=feval(@SteLar);
```

```
options=odeset('RelTol',1e-8);
options = odeset('Jacobian',hls(3),'JacobianP',hls(4),
'Hessians',hls(5),'HessiansP',hls(6));
[t,y]=ode45(hls{2},[0,3000],X0,options,0.1631021,1250,0.046875,20,
1.104,0.001,0.7167,0.1175,0.5);
size(t)
size(y)
plot(t(100000:325000),y(100000:325000,2));
'press any key'
pause
plot(y(100000:325000,3),y(100000:325000,2));
```

is a small modification of `testtimeinteg.m` in which the solver has access to the Jacobian matrix. The graphical output is similar, but the number of computed points is now 328197.

5.2 Poincaré section and Poincaré map

A Poincaré section is a surface in phase space that cuts across the flow of a dynamical system. It is a carefully chosen (in general, curved) surface in the phase space that is crossed by almost all orbits. It is a tool developed by Poincaré for visualization of the flow in more than two dimensions. The Poincaré section has one dimension less than the phase space and the Poincaré map transforms the Poincaré section onto itself by relating two consecutive intersection points, say u_k and u_{k+1}. We note that only those intersection points count, which come from the same side of the section. The Poincaré map is invertible because one gets u_n from u_{n+1} by following the orbit backwards. A Poincaré map turns a continuous-time dynamical system into a discrete-time one. If the Poincaré section is carefully chosen no information is lost concerning the qualitative behaviour of the dynamics. For example, if the system is being attracted to a limit cycle, one observes dots converging to a fixed point in the Poincaré section.

5.2.1 Poincaré maps in Cl_MatCont

When computing an orbit $(t, y(t))$ in Matlab an event can be defined as going through a zero of a given scalar function $G(t, y)$. If G does not explicitly depend on time in an autonomous dynamical system, this feature can be used to detect Poincaré intersections. One does this by setting the Events property to a function handle, e.g. `@events`, creating a function `[value,isterminal,direction] = events(t,y)` and calling

```
   [t,Y,TE,YE,IE] = solver(odefun,tspan,y0,options).
```

For the i-th event function:

- `value(i)` is the value of the function.
- `isterminal(i)` = 1 if the integration is to terminate at a zero of this event function and 0 otherwise.
- `direction(i)` = 0 if all zeros are to be computed (the default), +1 if only the zeros are needed where the event function increases, and -1 if only the zeros where the event function decreases.

Corresponding entries in `TE, YE`, and `IE` return, respectively, the time at which an event occurs, the solution at the time of the event, and the index i of the event function that vanishes.

Example

Let a set of two event functions be introduced by defining the function `testEV`:

```
function [value,isterminal,direction]= testEV(t,y,varargin)
value=[y(1)-0.2;y(2)-0.3];
isterminal=zeros(2,1);
direction=ones(2,1);
end
```

This event function requires that the system is at least two-dimensional and defines two events, namely $y(1) = 0.2$ and $y(2) = 0.3$. The integration will not be terminated if an event is detected and all zeros will be detected regardless of the direction of y (increasing or decreasing).

We now consider the system `adaptx` in the directory Testruns/TestSystems of MatCont with three state variables and two parameters. By runnning the script `testPoincare`:

```
TSTP=@testEV;
OPTIONS = odeset('RelTol',1e-8,'Events',TSTP);
hls = adaptx;
[t,y,TE,YE,IE] = ode45(hls{2},[0 300],[0.3 0.5 -0.1],OPTIONS,1,0.8);
x0 = y(end,:);
[t,y,TE,YE,IE] = ode45(hls{2},[0 10],x0,OPTIONS,1,0.8);
```

we integrate the system `adaptx` from 0 to 300 and then further from 300 to 310 starting from the point $[0.3; 0.5; -0.1]$ with parameter values $1, 0.8$ and we detect the points in the second run where $y(1) = 0.2$ or $y(2) = 0.3$. The output is given by

```
TE =

    5.6334
    6.6979

YE =

   -0.2247    0.3000    0.3131
    0.2000    0.4213   -0.1059

IE =

     2
     1
```

The test is run by typing `testPoincare` in the command line.

6 Equilibrium continuation

This example will show how to continue an equilibrium of a differential equation defined in a standard MATLAB ODE file. Furthermore, this example illustrates the detection, location, and processing of singularities along an equilibrium curve.

6.1 Mathematical definition

Consider a differential equation

$$\frac{du}{dt} = f(u,\alpha), \quad u \in \mathbf{R}^n, \alpha \in \mathbf{R} \quad f : \mathbf{R}^{n+1} \to \mathbf{R}^n \ . \tag{33}$$

We are interested in an equilibrium curve, i.e. $f(u,\alpha) = 0$. The defining function is therefore:

$$F(x) = f(u,\alpha) = 0 \tag{34}$$

with $x = (u,\alpha) \in \mathbf{R}^{n+1}$. Denote by $v \in \mathbf{R}^{n+1}$ the tangent vector to the equilibrium curve at x.

6.2 Initialization by time integration

If not much previous knowledge is available on a given dynamical system then a common way to start the study is to compute orbits and see whether some of them converge to (stable) equilibria or periodic orbits. An example for the case of equilibria is given in the testrun `testmymlPRC.m`.

6.3 Bifurcations and their normal form coefficients

In continuous-time systems there are two generic codim 1 bifurcations that can be detected along the equilibrium curve (no derivations will be done here; for more detailed information see [25]):

- *fold*, also known as *limit point*. We will denote this bifurcation by `LP`
- *Hopf*-point, denoted by `H`

The equilibrium curve can also have *branch points*. These are denoted with `BP`. To detect these singularities, we first define 3 test functions:

$$\begin{aligned}
\phi_1(u,\alpha) &= \det\begin{pmatrix} F_x \\ v^T \end{pmatrix}, & (35)\\
\phi_2(u,\alpha) &= \left(\begin{bmatrix} (2f_u(u,\alpha)\odot I_n) & w_1 \\ v_1^T & d \end{bmatrix} \backslash \begin{pmatrix} 0 \\ \dots \\ 0 \\ 1 \end{pmatrix}\right)_{n+1}, & (36)\\
\phi_3(u,\alpha) &= v_{n+1}, & (37)
\end{aligned}$$

where $\odot$ is the bialternate matrix product and v_1, w_1 are $\frac{n(n-1)}{2}$ vectors chosen so that the square matrix in (36) is non-singular. (the bialternate matrix product was introduced by C. Stéphanos [36]; see [22] for details). Using these test functions we can define the singularities:

- `BP`: $\phi_1 = 0$
- `H`: $\phi_2 = 0$
- `LP`: $\phi_3 = 0,\ \phi_1 \neq 0$

A proof that these test functions correctly detect the mentioned singularities can be found in [25]. Here we only notice that $\phi_2 = 0$ not only at Hopf points but also at *neutral saddles*, i.e. points where f_x has two real eigenvalues with sum zero. So, the singularity matrix is:

$$S = \begin{pmatrix} 0 & - & - \\ - & 0 & - \\ 1 & - & 0 \end{pmatrix} \tag{38}$$

For each detected limit point, the corresponding *quadratic normal form coefficient* is computed:

$$a = \frac{1}{2} p^T f_{uu}[q,q], \tag{39}$$

where $f_u q = f_u^T p = 0, q^T q = 1, p^T q = 1$. Mathematically, the limit point is nondegenerate (i.e. the equilibrium branch looks locally like a parabola) if and only if $a \neq 0$. In practice, because of round-off errors the computed a is always nonzero. So it is more important to check if the testfunction ϕ_3 changes sign. (In a *cusp point* (CP) $a = 0$ but this will not be detected on a branch of equilibria since it is a codimension 2 phenomenon, see §8.1)

At a Hopf bifurcation point, the *first Lyapunov coefficient* is computed by the formula

$$l_1 = \frac{1}{2} \mathrm{Re} \left\{ p^T \left(f_{uuu}[q,q,\bar{q}] - 2 f_{uu}[q, F_u^{-1} f_{uu}[q,\bar{q}]] + f_{uu}[\bar{q}, (2i\omega I_n - f_u)^{-1} f_{uu}[q,q]] \right) \right\}, \tag{40}$$

where $f_u q = i\omega q,\ q^T q = 1,\ f_u^T p = -i\omega p,\ \bar{p}^T q = 1$.

The *first Lyapunov coefficient* is quite important. If $l_1 < 0$ then the Hopf bifurcation is *supercritical,* i.e. within the center manifold on one side of the bifurcation only stable equilibria exist, on the other side unstable equilibria coexist with stable periodic orbits. If $l_1 > 0$ then the Hopf bifurcation is *subcritical,* i.e. within the center manifold on one side of the bifurcation stable equilibria coexist with unstable periodic orbits, on the other side only unstable equilibria exist. (In a *Generalized Hopf point* (GH) $l_1 = 0$ but this will not be detected on a branch of equilibria since it is a codimension 2 phenomenon, see §8.2)

6.3.1 Branch point locator

The location of Hopf and limit points usually does not cause problems. However, the location of branch points can give problems. The region of attraction of the Newton type continuation method which is used, has the shape of a cone (see [2]). In the localisation process we cannot assume to stay in this cone. This difficulty can be avoided by introducing $p \in \mathbf{R}^n$ and $\beta \in \mathbf{R}$ and considering the *extended system*:

$$\left\{ \begin{array}{rcl} f(u,\alpha) + \beta p &=& 0 \\ f_u^T(u,\alpha) p &=& 0 \\ p^T f_\alpha(u,\alpha) &=& 0 \\ p^T p - 1 &=& 0 \end{array} \right. \tag{41}$$

We solve this system by Newton's method with initial data $\beta = 0$ and $p : f_u^T p = \mu p$ where μ is the real eigenvalue with smallest norm. A branch point (u, α) corresponds to a regular solution $(u, \alpha, 0, p)$ of system (41) (see [3],p. 165). We note that the second order partial derivatives (Hessian) of f with respect to u and α are required.

The tangent vector at the singularity is also computed here. This is related to the processing of the branch point (computing the direction of the secondary branch).

6.4 Equilibrium initialization

Naively, one would start the continuation immediately with:

```
[x,v,s,h,f]=cont(@equilibrium, x0, v0, opt)
```

However, the equilibrium curve file has to know :

- which ode file to use,
- the values of all state variables,
- the values of all parameters,
- which parameter is active.

All this information can be supplied by one of the following two starting functions. Both functions return an initial point `x0` as well as its tangent vector `v0`.

- `[x0,v0]=init_EP_EP(@odefile, x, p, ap)`

  This routine stores its information in a global structure `eds` (see also Figure 4). The result of `init_EP_EP` is a vector `x0` with the state variables and the active parameter and a vector `v0` that is empty. Here `odefile` is the ode-file (system-definition file) to be used, `x` is a vector containing the values of the state variables. `p` is the vector containing the current values of the parameters and `ap` is the active parameter. The full listing of the equilibrium initializer can be found in the file 'Equilibrium/init_EP_EP.m' of the toolbox.

- `[x0,v0]=init_BP_EP(@odefile, x, p, s, h)`

  Calculates an initial point for starting a new branch from a branch point detected on an equilibrium curve. This routine stores its information in a global structure `eds` (see also Figure 4). Here `odefile` is the ode-file (system-definition file) to be used, `x` is a vector containing the values of the state variables returned by a previous equilibrium curve continuation. `p` is the vector containing the current values of the parameters and `h` contains the value of the initial amplitude. The full listing of the equilibrium initializer and the curve definition can be found in the files 'Equilibrium/init_BP_EP.m' and 'equilibrium.m' of the toolbox, respectively.

6.5 Bratu example

The first example we will look at is a 4-point discretization of the Bratu-Gelfand BVP [22]. This model is defined as follows:

$$x' = y - 2x + ae^x \qquad (42)$$
$$y' = x - 2y + ae^y \qquad (43)$$

The system is specified as

bratu.m

```
function out = bratu
out{1} = @init;
out{2} = @fun_eval;
out{3} = @jacobian;
out{4} = @jacobianp;
out{5} = @hessians;
out{6} = @hessiansp;
out{7} = [];
out{8} = [];
out{9} = [];
out{10}= @userf1;

end

% --------------------------------------------------------------------------
function dydt = fun_eval(t,kmrgd,a)
dydt =  [ -2*kmrgd(1)+kmrgd(2)+a*exp(kmrgd(1));
           kmrgd(1)-2*kmrgd(2)+a*exp(kmrgd(2)) ];

% --------------------------------------------------------------------------
function [tspan,y0,options] = init
tspan = [0; 10];
y0 = [0;0];handles = feval(@bratu)
options = odeset('Jacobian',handles(3),'JacobianP', 'handles(4)',...
...,'Hessians',handles(5), 'Hessiansp',handles(6));
% --------------------------------------------------------------------------
function jac = jacobian(t,kmrgd,a)
jac  = [ -2+a*exp(kmrgd(1))   1
          1              -2+a*exp(kmrgd(2)) ];

% --------------------------------------------------------------------------
function jacp = jacobianp(t,kmrgd,a)

jacp = [ exp(kmrgd(1))
         exp(kmrgd(2)) ];

% --------------------------------------------------------------------------
function hess = hessians(t,kmrgd,a)
hess1=[[a*exp(kmrgd(1)),0];[0,0]];
hess2=[[0,0];[0,a*exp(kmrgd(2))]];
hess(:,:,1) = hess1;
hess(:,:,2) = hess2;

% --------------------------------------------------------------------------
function hessp = hessiansp(t,kmrgd,a)
hessp1=[[exp(kmrgd(1)),0];[0,exp(kmrgd(2))]];
hessp(:,:,1) = hessp1;

%---------------------------------------------------------------------------
function userfun1 = userf1(t,kmrgd,a)
userfun1 = a-0.2;

```

bratu.m

As seen above, a user function is defined to detect all points where $a = 0.2$. This system has an equilibrium at $(x, y, a) = (0, 0, 0)$ which we will continue with respect to a. We first compute 50 points and then extend the curve with another 50 points.

```
global cds
p=[0];ap=[1];
[x0,v0]=init_EP_EP(@bratu,[0;0],p,ap);
opt=contset;
opt=contset(opt,'MaxNumPoints',50);
opt=contset(opt,'Singularities',1);
opt=contset(opt,'Userfunctions',1);
UserInfo.name='userf1';
UserInfo.state=1;
UserInfo.label='u1';
opt=contset(opt,'UserfunctionsInfo',UserInfo);
[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
[x,v,s,h,f]=cont(x,v,s,h,f,cds);
cpl(x,v,s,[3 1 2]);
```

The above computations can be done by running `testbratu.m` in the directory `Testruns`. The output in the command window is as follows:

```
>> testbratu
first point found
tangent vector to first point found
label = u1, x = ( 0.259174 0.259174 0.200002 )
label = LP, x = ( 1.000001 1.000001 0.367879 )
a=3.535537e-01
Neutral saddle
label = H , x = ( 2.000000 2.000000 0.270671 )
label = u1, x = ( 2.542639 2.542639 0.200000 )
elapsed time  = 0.4 secs
npoints curve = 50
start computing extended curve
label = BP, x = ( 3.000000 3.000000 0.149361 )
elapsed time  = 0.1 secs
npoints curve = 100
```

We note that in the first continuation two zeros of the user function (label u1) were detected, as well as a limit point (label LP) and a neutral equilibrium (label H). A neutral equilibrium is an equilibrium with two real eigenvalues with sum zero.

In the extension of the run at $(x, y, a) \approx (3.0; 3.0; 0.15)$ the system has a branch point (label BP).

`cpl(x,v,s,[3 1 2])` plots a 3D-plot with the parameter a on the x-axis and the first and second state variable on the y- and z-axes, respectively. The labels of the plot are changed manually by the following commands:

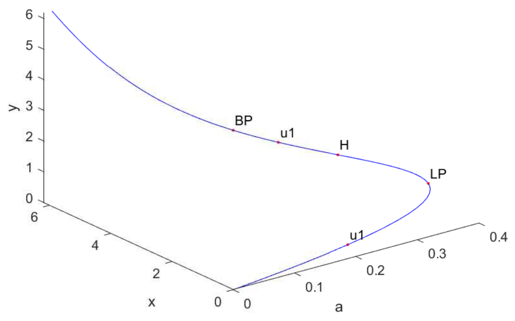


Figure 8: Equilibrium curve of `bratu.m` with a branch point

- press ‘Insert’ → ‘XLabel’ →, ‘a’
- press ‘Insert’ → ‘YLabel’ →, ‘x’
- press ‘Insert’ → ‘ZLabel’ →, ‘y’

The resulting curve is plotted in Figure 8.

To select the branch point the output s is used. Since the first and last points are also treated as singular, the array of structures `s` has 7 components and the data concerning the branch point are at $s(6)$. To switch to another branch at the detected branch point, we select that branch point and we use the starter `init_BP_EP`. We start a forward and backward continuation from this point:

```
testbratu;
x1=x(1:2,s(6).index);
p(ap)=x(3,s(6).index);
[x0,v0]=init_BP_EP(@bratu,x1,p,s(6),0.01);
opt=contset(opt,'InitStepsize',0.0001);
[x1,v1,s1,h1,f1]=cont(@equilibrium,x0,v0,opt);
cpl(x1,v1,s1,[3 1 2]);
opt=contset(opt,'Backward',1);
[x2,v2,s2,h2,f2]=cont(@equilibrium,x0,v0,opt);
cpl(x2,v2,s2,[3 1 2]);
```

Note that p is a vector containing the initial values of all parameters. The above computations can be done by running `testbratu2.m` in the directory `Testruns`. The output in the command window is as follows:

```
>> testbratu2
first point found
tangent vector to first point found
```

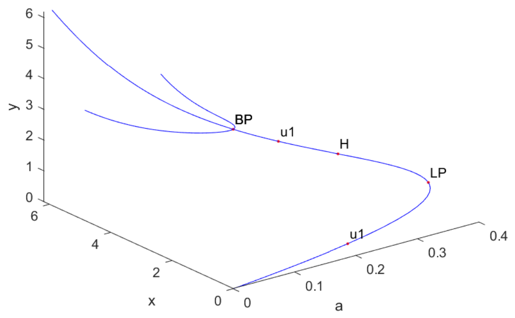


Figure 9: Equilibrium curve of `bratu.m` with new branches rooted in the branch point

```
label = u1, x = ( 0.259174 0.259174 0.200002 )
label = LP, x = ( 1.000001 1.000001 0.367879 )
a=3.535537e-01
label = H , x = ( 2.000000 2.000000 0.270671 )
label = u1, x = ( 2.542639 2.542639 0.200000 )

elapsed time  = 0.1 secs
npoints curve = 50
start computing extended curve
label = BP, x = ( 3.000000 3.000000 0.149361 )

elapsed time  = 0.1 secs
npoints curve = 100
first point found
tangent vector to first point found

elapsed time  = 0.1 secs
npoints curve = 50
first point found
tangent vector to first point found
label = BP, x = ( 3.000000 3.000000 0.149361 )

elapsed time  = 0.1 secs
npoints curve = 50
```

We note that the branch point is "discovered" a second time during the backward continuation of the secondary branch that is rooted at the branch point. All computed curves are plotted together in Figure 9.

7 Continuation of limit cycles

7.1 Mathematical definition

Consider the following differential equation

$$\frac{du}{dt} = f(u, \alpha) \tag{44}$$

with $u \in \mathbf{R}^n$ and $\alpha \in \mathbf{R}$. A periodic solution with period T satisfies the following system

$$\left\{ \begin{array}{l} \frac{du}{dt} = f(u, \alpha) \\ u(0) = u(T) \end{array} \right. . \tag{45}$$

For simplicity the period T is treated as a parameter resulting in the system

$$\left\{ \begin{array}{l} \frac{du}{d\tau} = T\, f(u, \alpha) \\ u(0) = u(1) \end{array} \right. . \tag{46}$$

If $u(\tau)$ is its solution then the shifted solution $u(\tau + s)$ is also a solution to (46) for any value of s. To select one solution, a phase condition is added to the system. The complete BVP (boundary value problem) is

$$\left\{ \begin{array}{rl} \frac{du}{d\tau} - Tf(u, \alpha) & = 0 \\ u(0) - u(1) & = 0 \\ \int_0^1 \langle u(t), \dot{u}_{old}(t) \rangle dt & = 0 \end{array} \right. \tag{47}$$

where $\dot{u}_{old}$ is the derivative of a previous solution. A limit cycle is a closed phase orbit corresponding to this periodic solution.

7.2 Discretization of a limit cycle

In MatCont limit cycles are discretized using orthogonal collocation [6], the same way as it was done in AUTO [12]; the left hand side of the resulting system is the defining function $F(u, T, \alpha)$ for limit cycles.

In practice this means that the normalized time interval $[0, 1]$ is divided in a number *ntst* (number of test intervals) of intervals with variable lengths; the (*ntst*+1) endpoints of these intervals form the *coarse mesh*. Each interval is further subdivided in a number *ncol* (number of collocation points) of subintervals with equal lengths. So altogether there are (*ntst* $\times$ *ncol* + 1) *fine mesh* points. The state variable values are collected in the points of the *fine mesh*. The limit cycle itself is then approximated by a continuous piecewise polynomial which is a polynomial of degree *ncol* in each of the *ntst* coarse mesh intervals.

If the number of phase variables is denoted by *nphase* then altogether

(*ntst* $\times$ *ncol* $\times$ *nphase*)

state variable values are stored, since by periodicity the values in the endpoints 0 and 1 must be identical. But the number of continuation variables is

((*ntst* $\times$ *ncol* + 1) $\times$ *nphase*)+2

since the state variable values in both 0 and 1 are stored and the period T and a free system parameter must be included. They are stored in the output vector x of a limit cycle continuation in that order, after all state variable values.

The coarse mesh is adapted after each number of *Adapt* continuation points, cf §3.4.9. We note that *ntst* and *ncol* are input arguments of the routines that initialize the continuation of limit cycles but are not explicitly found in the output of the continuation; however, they are preserved as fields in the global structure `lds`.

7.3 Plotting the output of a continuation of limit cycles

A continuation run of limit cycles has the standard form

```
[x,v,s,h,f]=cont(@limitcycle,x0,v0,opt);
```

Here `x` is a matrix in which each column corresponds to a computed orbit. The last element of each column contains the value of the active parameter. CL_MatCont provides only limited possibilities to visualise the output. Namely, it provides the routine `plotcycle.m` which is to be called in the form

```
plotcycle(x,v,s,e)
```

where `e` is either a two- or a three-dimensional row vector.

- If `e=[e1,e2]` one must have $1 \leq e1, e2 \leq nphase$ and a 2D plot will be generated in which the projections of the orbits on the $(e1, e2)$ coordinate plane are shown. The first and last curves, and the curves with singulaties are in red.
- If `e=[e1,e2,e3]` then `e1` must be the index of the active parameter, i.e. `e1=size(x,1)` and one must have $1 \leq e2, e3 \leq nphase$. Then a 3D plot will be generated in which the projections of the orbits on the $(e2, e3)$ coordinate plane are shown versus the active parameter. The first and last curves, and the curves with singulaties are in red.

7.4 Initialization by time integration

The initialization of a continuation of limit cycles is a nontrivial issue. Probably the most often used method is to start from a Hopf point that is detected on a curve of equilibria, see §7.6. Another powerful method is to compute orbits and see whether some of them converge to (stable) periodic orbits. If such an orbit is found (the easiest way is by graphical inspection) then one uses an (approximation to) a point of the orbit and integrates again over a time interval that is somewhat larger than the period of the orbit but smaller that twice the period. The routine `initOrbLC.m` in the directory `LimitCycle` then can initialize the continuation of periodic orbits with a user-chosen free parameter, with the period of orbit as the second free parameter. An example is provided in the testrun `testselectcycle.m` in the directory `Testruns`:

```
OPTIONS = [];
hls = adaptx;
OPTIONS=odeset('RelTol',1e-8);
[t,y] = ode45(hls{2},[0 300],[0.3 0.5 -0.1],OPTIONS,1,0.8);

x1 = y(end,:);

[t,y] = ode45(hls{2},[0 10],x1,OPTIONS,1,0.8);

figure
```

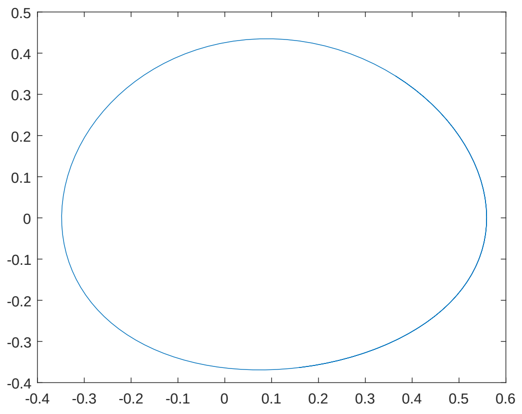


Figure 10: A closed curve in the adaptx - system.

```
plot(y(:,1),y(:,2))

p=[1;0.8];
ap=[2];

tolerance=1e-2;
[x0,v0]=initOrbLC(@adaptx,t,y,p,ap,20,4,tolerance);
opt=contset;
opt=contset(opt,'MaxNumPoints',50);
%opt=contset(opt,'TSearchOrder',0);
%opt=contset(opt,'Backward',1);
[xlcc,vlcc,slcc,hlcc,flcc]=cont(@limitcycle,x0,v0,opt);

figure
axes
plotcycle(xlcc,vlcc,slcc,[size(xlcc,1) 1 2]);
```

In this script `adaptx` is the dynamical system (54) in §7.7 whose *odefile* `adaptx.m` is stored in the directory `Testruns/TestSystems`. It has three state variables and two parameters. Starting from the point $(0.3; 0.5; -0.1)$ and with parameter values $(1, 0.8)$ it is integrated over the time span 300. The endpoint is called $x1$ and from this point another time integration over the shorter span 10 is performed. It is then checked graphically that this new integration contains a closed circle, see Figure 10 for a plot in a two-dimensional space.

The call

```
[x0,v0]=initOrbLC(@adaptx,t,y,p,ap,20,4,tolerance);
```

now initializes the continuation of limit cycles. Here t is a column vector whose entries are the time points (between 0 and 10) and y is a matrix whose rows contain the coordinates of the points computed along the orbit. Also, $p = (1; 0.8)$ is the parameter vector and $ap = [2]$ contains the indices of the free parameters, so in this case the second parameter is free. Also, the number of test functions is 20 and the number of collocation points in the discretization

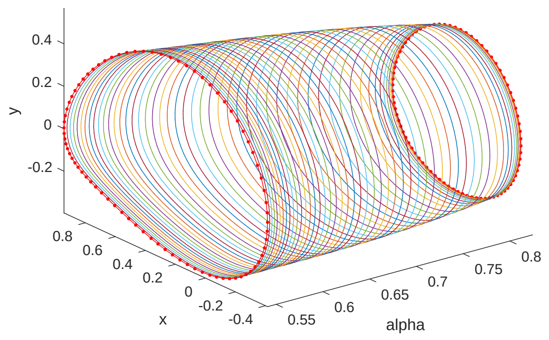


Figure 11: 50 limit cycles in the adaptx - system.

of the limit cycle is 4. Finally, the tolerance is a threshold for accepting an initial part of the computed orbit as an approximation to a limit cycle.

The last command plots the continuation of a branch of limit cycles with 50 computed limit cycles in the state space, see Figure 11 (the axis labels were added manually).

Note: when the lines

```
%opt=contset(opt,'TSearchOrder',0);
%opt=contset(opt,'Backward',1);
```

in `testselectcycle` are made active, then the initialization of the continuation of limit cycles involves a decreasing order of the unit vectors in continuation space instead of an increasing order. This accidentally leads to a continuation in the opposite direction; therefore cds.options.Backward is set to reverse the continuation.

7.5 Bifurcations of limit cycles

On a limit cycle curve the following bifurcations can occur

- *Branch Point of Cycles*, this will be denoted as `BPC`
- *Period Doubling*, denoted as `PD`
- *Fold*, also known as *Limit Point of Cycles*, this will be denoted as `LPC`
- *Neimark-Sacker*, this will be denoted as `NS`

The test function for the Period Doubling bifurcation is defined by the following system

$$\left\{\begin{array}{rl} \dot{v}(\tau) - Tf_u(u,\alpha)v(\tau) + G\varphi(\tau) & = 0 \\ v(0) + v(1) & = 0 \\ \int_0^1 \langle \psi(\tau), v(\tau) \rangle d\tau & = 1 \end{array}\right. \tag{48}$$

here φ and ψ are so-called bordering vector-functions [25], see [15] for details on the implementation. The system is discretized using orthogonal collocation and solved using the standard MATLAB sparse system solver. The solution component $G \in \mathbf{R}$ of this system is the test function and equals zero when there is a Period Doubling bifurcation.

The Fold bifurcation is detected in the same way as the Fold bifurcation of equilibria, the last component of the tangent vector (the α component) is used as the test function.

The Neimark-Sacker bifurcation is detected by monitoring the eigenvalues of the monodromy matrix for the cycle. The monodromy matrix is computed like in AUTO by a block elimination in the discretized form of the Jacobian of (47).

BPC cycles are not generic in families of limit cycles, but they are common in the case of symmetries, if the branch parameter is also the continuation parameter. CL_MatCont uses a strategy that requires only the solution of linear systems; it is based on the fact that in a symmetry-breaking BPC cycle M_D has rank defect two, where M_D is the square matrix M_D, obtained from the discretized form of the Jacobian of (47). To be precise, if $h \in \mathcal{C}^1([0,1], \mathbb{R}^n)$, then

$$Mh = \begin{bmatrix} \dot{h} - Tf_x(x(t), \alpha)h \\ h(0) - h(1) \end{bmatrix},$$

and

$$M_D(h)_{dm} = \begin{bmatrix} (\dot{h} - Tf_x(x(t), \alpha)h)_{dc} \\ h(0) - h(1) \end{bmatrix},$$

where $()_{dm}$ and $()_{dc}$ denote discretization in mesh points and in collocation points, respectively. Therefore we border M_D with two additional rows and columns to obtain

$$M_{Dbb} = \begin{bmatrix} M_D & w_1 & w_2 \\ v_1^* & 0 & 0 \\ v_2^* & 0 & 0 \end{bmatrix},$$

so that M_{Dbb} is nonsingular in the BPC cycle. Then we solve the systems

$$M_{Dbb} \begin{bmatrix} \psi_{11} & \psi_{12} \\ g_{BPC11} & g_{BPC12} \\ g_{BPC21} & g_{BPC22} \end{bmatrix} = \begin{bmatrix} 0_{(Nm+1)n} & 0_{(Nm+1)n} \\ 1 & 0 \\ 0 & 1 \end{bmatrix},$$

where ψ_{11}, ψ_{12} have $(Nm+1)n$ components, and g_{BPC11}, g_{BPC12}, g_{BPC21}, and g_{BPC22}, are scalar test functions for the BPC. In the BPC cycle they all vanish.

The singularity matrix is

$$S = \begin{pmatrix} 0 & 0 & 0 & 0 & - & - & - & - \\ - & - & - & - & - & 0 & - & - \\ - & - & - & - & - & - & 0 & - \\ - & - & - & - & 1 & - & 1 & 0 \end{pmatrix} \tag{49}$$

The first row corresponds to the BPC. It contains 4 zeros which indicates that g_{BPC11}, g_{BPC12}, g_{BPC21}, and g_{BPC22} should vanish. The last row corresponds to the NS. Because we have to exclude that all four testfunctions of the BPC are zeros, we introduce an extra testfunction which corresponds to the norm of these four testfunctions. A NS is detected if this norm is nonzero,the testfunction for the fold is nonzero and the testfunction for the NS is equal to zero.

7.5.1 Branch Point Locator

The location of BPC points in the non-generic situation (i.e. where some symmetry is present) as zeros of the test functions is numerically suspect because no local quadratic convergence can be guaranteed. This difficulty can be avoided by introducing an additional unknown $\beta \in \mathbf{R}$ and considering the minimally extended system:

$$\begin{cases} \frac{dx}{dt} - Tf(x,\alpha) + \beta p_1 & = 0 \\ x(0) - x(1) + \beta p_2 & = 0 \\ \int_0^1 \langle x(t), \dot{x}_{old}(t) \rangle dt + \beta p_3 & = 0 \\ G[x,T,\alpha] & = 0 \end{cases} \tag{50}$$

where G is defined as in (84) and $[p_1^T p_2^T p_3]^T$ is the bordering vector $[w_{01}; w_{02}; w_{03}]^T$ in (86). We solve this system with respect to x, T, α and β by Newton's method with initial $\beta = 0$. A branch point (x, T, α) corresponds to a regular solution $(x, T, \alpha, 0)$ of system (50) (see [3],p. 165). We note, however that the second order partial derivatives (Hessian) of f with respect to x and α are required. The tangent vector v_{1st} at the BPC singularity is approximated as $v_{1st} = \frac{v_1+v_2}{2}$ where v_1 is the tangent vector in the continuation point previous to the BPC and v_2 is the one in the next point.

7.5.2 Normal form coefficients

The numerical methods used in MatCont to compute normal form coefficients of codimension 1 bifurcations of limit cycles are discussed in [28].

The periodic normal form at the Limit Point of Cycles (LPC) bifurcation is

$$\begin{cases} \frac{d\tau}{dt} & = & 1 - \xi + a\xi^2 + \cdots, \\ \frac{d\xi}{dt} & = & b\xi^2 + \cdots, \end{cases} \tag{51}$$

where $\tau \in [0, T]$, ξ is a real coordinate on the center manifold that is transverse to the limit-cycle, $a, b \in \mathbb{R}$, and dots denote nonautonomous T-periodic $O(\xi^3)$-terms. For each detected LPC point the normal form coefficient b is computed. If $b \neq 0$ then the LPC bifurcation is *nondegenerate,* i.e., the cycle manifold is quadratically tangential to the hyperplane orthogonal to the parameter direction. In practice, by round-off errors b is never zero, so it is better to check if the testfunction changes sign. (for $b = 0$ the LPC degenerates to a *cusp of cycles* (CP) but this will not be detected on a branch of limit cycles since it is a codimension 2 phenomenon.)

The periodic normal form at the Period Doubling (PD) bifurcation is

$$\begin{cases} \frac{d\tau}{dt} & = & 1 + a\xi^2 + \cdots, \\ \frac{d\xi}{dt} & = & c\xi^3 + \cdots, \end{cases} \tag{52}$$

where $\tau \in [0, 2T]$, ξ is a real coordinate on the center manifold that is transverse to the limit cycle, $a, c \in \mathbb{R}$, and dots denote nonautonomous $2T$-periodic $O(\xi^4)$-terms. The coefficient c determines the stability of the period doubled cycle in the center manifold and is computed during the processing of each PD point.

If $c < 0$ then the PD bifurcation is supercritical, i.e., within the center manifold on one side of the bifurcation only stable cycles exist and on the other side unstable cycles coexist with stable period-doubled cycles. If $c > 0$ the PD bifurcation is subcritical, i.e., within the center manifold on one side of the bifurcation stable cycles coexist with unstable period-doubled cycles and on the other side only unstable cycles exist. (for $c = 0$ the PD degenerates to *Generalized Period Doubling bifurcation* (GPD) but this will not be detected on a branch of limit cycles since this is a codimension 2 phenomenon).

The periodic normal form at the Neimark-Sacker (NS) bifurcation is

$$\left\{\begin{array}{rcl} \frac{d\tau}{dt} & = & 1 + a|\xi|^2 + \cdots, \\ \frac{d\xi}{dt} & = & \frac{i\theta}{T}\xi + d\xi|\xi|^2 + \cdots, \end{array}\right. \tag{53}$$

where $\tau \in [0, T]$, ξ is a complex coordinate on the center manifold that is complementary to τ, $a \in \mathbb{R}, d \in \mathbb{C}$, and dots denote nonautonomous T-periodic $O(|\xi|^4)$-terms. The critical coefficient d in the periodic normal form for the NS bifurcation is computed during the processing of a NS point.

If Re $d < 0$ then the bifurcation is supercritical, i.e. in the center manifold the limit cycle is stable on one side of the bifurcation point and unstable on the other side and the unstable cycles coexist with stable invariant tori. If Re $d > 0$ then the bifurcation is subcritical, i.e. in the center manifold the limit cycle is stable on one side of the bifurcation point and unstable on the other side and the stable cycles coexist with unstable invariant tori. (for Re $d = 0$ the NS bifurcation degenerates to a *Chenciner bifurcation* (CH) but this will not be detected on a branch of limit cycles since it is a codimension 2 phenomenon).

7.6 Limitcycle initialization

For limit cycles the same problems occur as with equilibria, a limit cycle continuation can't be done by just calling the continuer as

```
[x,v,s,h,f]=cont(@limitcycle, x0, v0, opt)
```

The limit cycle curve file has to know

- which ode file to use,
- which parameter is active,
- the values of all parameters,
- the number of mesh and collocation points to use for the discretization.

Also an initial cycle `x0` has to be known. All this information can be supplied using any of the following three starting functions. All three return an initial cycle `x0` as well as its tangent vector `v0`.

- `[x0,v0]=init_H_LC(@odefile, x, p, ap, h, ntst, ncol)`
  Calculates an initial cycle from a Hopf point detected on an equilibrium curve. Here `odefile` is the ode-file to be used. `x` is a vector containing the values of the state variables returned by a previous equilibrium curve continuation. `p` is the vector containing the current values of the parameters. `ap` is the active parameter, `h` contains the value of the initial amplitude and `ntst` and `ncol` are the number of mesh and collocation points to be used for the discretization.

- `[x0,v0]=init_BPC_LC(@odefile, x, v, s, ap, ntst, ncol,h)`
  Calculates an initial cycle for starting a secondary cycle from a BPC detected on a previous calculated limit cycle. `odefile`, `ap`, `ntst` and `ncol` are the same as for `init_H_LC`. `x`, `v` and `s` are here the `x`, `v` and `s` as returned by a previous limit cycle continuation and `h` contains the value of the initial amplitude.

- `[x0,v0]=init_LC_LC(@odefile, x, v, s, par, ap, ntst, ncol)`
  This starter can be used to start a limit cycle continuation from a previous limit cycle continuation. `odefile`, `ap`, `ntst` and `ncol` are the same as for `init_H_LC`. `x` and `v` are here the `x` and `v` as returned by a previous limit cycle continuation. `s` is the special point structure for the point from where to start the new continuation. `par` is the parameter vector that is to be used for the new continuation. If this is the same as in the cycle returned by the continuer (arguably the natural situation), then this is the same as `s.data.parametervalues`

- `[x0,v0]=init_PD_LC(@odefile, x, s, ntst, ncol,h)`
  This starter calculates an initial double period cycle from a period doubling bifurcation point. Here `h` is a distance parameter which can be set to zero. We note also that there is no active parameter `ap` input argument.

7.7 Adaptive control example

For this example the following system from adaptive control was used in a feedback control system, as described in [20], [21] and further used in [25] (Example 5.4, p. 178):

$$\begin{cases} \dot{x} &= y \\ \dot{y} &= z \\ \dot{z} &= -\alpha z - \beta y - x + x^2 \end{cases} \tag{54}$$

This system is introduced in the testruns of MatCont under the name `adaptx`. It has a Hopf point at the origin for $\alpha = 1$ and $\beta = 1$ as detected in the testrun `testadapt.m`.

From this Hopf point an initial cycle is calculated using the starter `init_H_LC`. The results of the continuation are plotted using the plot function `plotcycle(x,v,s,e)` (see Figure 12). This function plots the cycles `x`. `e` is an array whith either 2 or 3 elements for 2-dimensional and 3-dimensional plotting respectively. Its entries must be indices of state variables or active parameters in x. The index of the active parameter is `size(x,1)`

```
>> init;
>> [x0,v0]=init_EP_EP(@adaptx,[0;0;0],[-10;1],[1]);
>> opt = contset; opt = contset(opt,'Singularities',1);
>> [x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
first point found
tangent vector to first point found
label = H , x = ( 0.000000 0.000000 0.000000 1.000002 )
First Lyapunov coefficient = -3.000001e-001

elapsed time  = 0.3 secs
npoints curve = 300
>> x1=x(1:3,s(2).index);p=[x(end,s(2).index);1];
>> [x0,v0]=init_H_LC(@adaptx,x1,p,[1],1e-6,20,4);
>> opt = contset(opt,'MaxNumPoints',200);
>> opt = contset(opt,'Multipliers',1);
>> opt = contset(opt,'Adapt',1);
>> [xlc,vlc,slc,hlc,flc]=cont(@limitcycle,x0,v0,opt);
first point found
tangent vector to first point found
Limit point cycle (period = 6.283185e+000, parameter = 1.000000e+000)
Normal form coefficient = -1.306379e+000
Branch Point cycle(period = 6.283185e+000, parameter = 9.999996e-001)
Period Doubling (period = 6.364071e+000, parameter = 6.303020e-001)
Normal form coefficient = -4.267675e-002
Neimark-Sacker (period = 6.433818e+000, parameter = 1.895460e-008)
Neutral saddle
Period Doubling (period = 6.364071e+000, parameter = -6.303020e-001)
Normal form coefficient = 4.268472e-002

elapsed time  = 27.6 secs
npoints curve = 200
>> plotcycle(xlc,vlc,slc,[size(xlc,1) 1 2]);
```

We note that `xlc` is a 245×200 matrix; each column corresponds to a computed limit cycle and gives the coordinates of all points of the fine mesh, i.e. $243 = (20 \times 4 + 1) \times 3$ values, plus the period T as the $244 - th$ component and the value of the active parameter α as the $245 - th$ component.

The x-axis contains the active parameter, the y-axis the first state variable x and the z-axis the second state variable y. This run can be tested by the statement `testadapt1` (the axis labels have to be set manually). If you run only this example, do not forget to execute init `init` statement first.

We note that the Limit point cycle and Branch point cycle detected in this run are degenerate: they reduce to the Hopf point itself. The Neimark-Sacker bifurcation is, in reality, a Neutral Saddle, i.e. it is an unstable periodic orbit with two real multipliers whose product is 1.

From the first Period Doubling bifurcation detected a limit cycle continuation of the nearby double period cycle is started. First, an initial cycle and its tangent vector are calculated using the starter `init_PD_LC`. The continuation is done using the standard continuer and the result is plotted using the `plotcycle` function (see figure 13).

```
>> [x1,v1]=init_PD_LC(@adaptx,xlc,slc(4),40,4,1e-6);
>> opt=contset(opt,'MaxNumPoints',250);
>> [xlc2,vlc2,slc2,hlc2,flc2]=cont(@limitcycle,x1,v1,opt);
first point found
tangent vector to first point found
Branch Point cycle(period = 1.272814e+001, parameter = 6.303020e-001)
Period Doubling (period = 1.273437e+001, parameter = 5.796299e-001)
Normal form coefficient = -5.579636e-002
Neimark-Sacker (period = 1.154609e+001, parameter = 2.806142e-010)
```

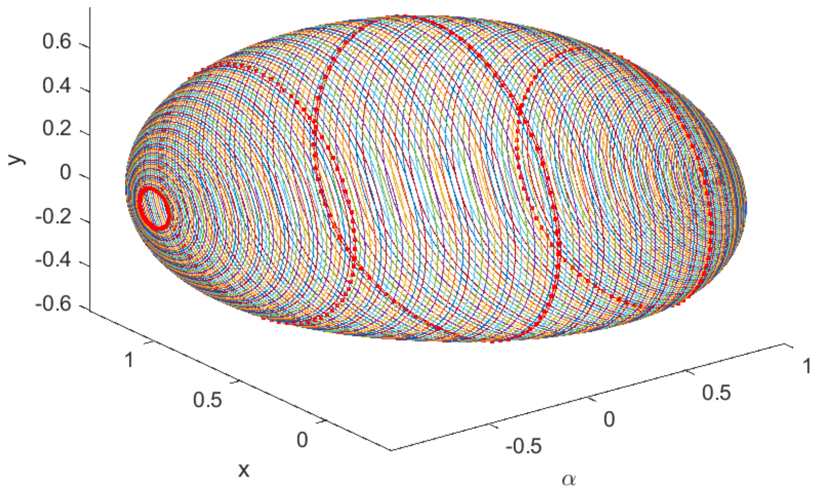


Figure 12: Computed limit cycle curve

```
Neutral saddle
Period Doubling (period = 1.106284e+001, parameter = -4.471966e-002)
Normal form coefficient = 6.970442e-003
Limit point cycle (period = 1.103168e+001, parameter = -4.494912e-002)
Normal form coefficient = 1.311327e+002
Neimark-Sacker (period = 1.076785e+001, parameter = -1.076152e-009)
Neutral saddle
Limit point cycle (period = 1.103169e+001, parameter = 4.494912e-002)
Normal form coefficient = -1.310582e+002
Period Doubling (period = 1.106284e+001, parameter = 4.471966e-002)
Normal form coefficient = -6.973392e-003
Neimark-Sacker (period = 1.154609e+001, parameter = 5.372279e-010)
Neutral saddle
Period Doubling (period = 1.273437e+001, parameter = -5.796298e-001)
Normal form coefficient = 5.580465e-002

elapsed time  = 62.3 secs
npoints curve = 250
>> plotcycle(xlc2,vlc2,slc2,[size(xlc2,1) 1 2]);
```

This run can be tested by the statement `testadapt2`.

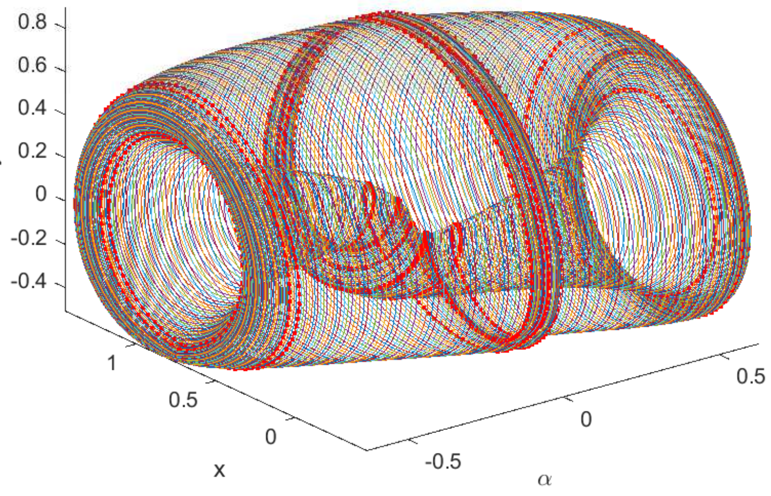


Figure 13: Computed limit cycle curve started from a Period Doubling bifurcation

7.8 The phase response curve

The phase response curve of a limit cycle, or PRC, is a curve, defined over the period of the cycle, that expresses, at each time of that period, the effect of a small input vector on the cycle. In experimental circumstances, this may correspond to injected current, to the addition of more chemical agents, etc. A positive value means that the current cycle is shortened in time, a negative value means that the period is prolonged.

The PRC, as it is generally computed, is exact for infinitesimally small input vectors. In practice the maximum norm of the input vector would depend on the needed accuracy and the values of the system's state variables.

The derivative phase response curve or dPRC also has some very important applications. For the concrete use of PRC and dPRC in synchronization studies in neural modeling, we refer to [23].

Cl_MatCont supports the computation of the PRC and dPRC of limit cycles during continuation, using the method described in [24]. The standard method, which uses numerical integration of the adjoint system, was implemented in XPPAUT [17].

The use in MatCont is easy: before starting the actual limit cycle continuation, the user can specify whether he wants to compute the PRC, dPRC or both, and he needs to indicate the input vector used. When a scalar is given as input, then the vector has this scalar as first entry and all other entries are zero. Then in separate plotting windows, for each computed step in limit cycle continuation, the PRC and/or dPRC are computed and plotted. The computed values are saved in the output f-array of the continuation.

To illustrate the use in CL_MatCont, we here supply code that does a LC continuation experiment in the Morris Lecar system (the odefile is called `MyML.m` and is located in the directory `Testruns/TestSystems`), and also computes the PRC and dPRC of the system:

```
clear
global x v s h f opt
OPTIONS=[];
hls = MyML;
[t,y] = ode45(hls{2},[0 1000],[0 0],OPTIONS,30,10);
x0 = y(end,:)';

opt=contset;opt=contset(opt,'Singularities',1);
opt = contset(opt,'MaxStepsize',10);
opt=contset(opt,'MaxNumpoints',2500);
[x1,v1]=init_EP_EP(@MyML,x0,[30;6],[1]);
[x,v,s,h,f]=cont(@equilibrium,x1,[],opt);

x1=x(1:2,s(5).index);p=[x(end,s(5).index);6];
[x0,v0]=init_H_LC(@MyML,x1,p,[1],1e-6,40,4);
opt = contset(opt,'Multipliers',0);
opt = contset(opt,'Adapt',1);
opt = contset(opt,'MaxStepsize',5);
opt = contset(opt,'FunTolerance',1e-6);
opt = contset(opt,'VarTolerance',1e-6);
opt = contset(opt,'PRC',1);
opt = contset(opt,'dPRC',1);
opt = contset(opt,'Input',1);
opt = contset(opt,'MaxNumPoints',100);
[xlc2,vlc2,slc2,hlc2,flc2]=cont(@limitcycle,x0,v0,opt);
fvector=flc2(:,100);
PRC10=fvector(42:202);
dPRC10=fvector(203:363);
plot(PRC10,'r');
hold on
plot(dPRC10,'b');
```

We note that the first 41 components (number of test intervals plus one) of the *fvector* contain the coarse mesh points of the discretization. The next 161 components (number of fine mesh points, i.e. number of test intervals times number of collocation points plus 1) contain the values of the PRC in the fine mesh points. The next 161 components contain the values of the dPRC in the fine mesh points.

This script is in the MatCont directory `Testruns` under the name `testmymlPRC`. The output is:

```
>> testmymlPRC
first point found
tangent vector to first point found
```

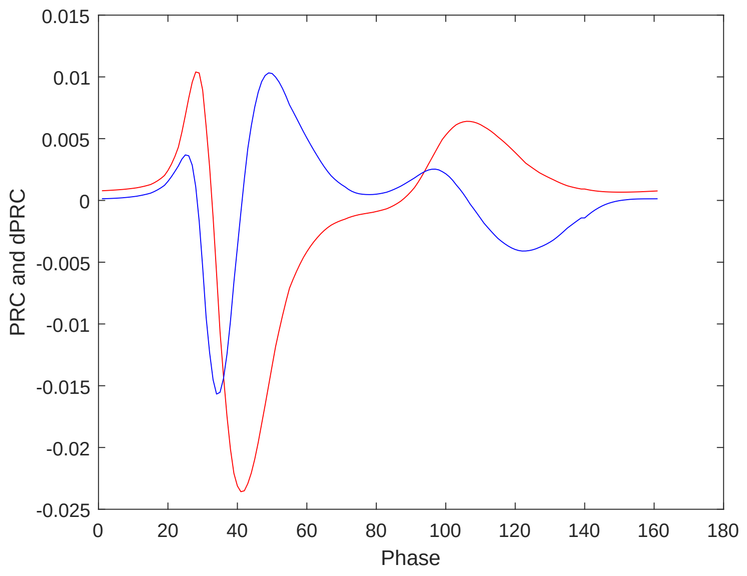


Figure 14: Phase response curve in the Morris-Lecar system: red=PRC, blue=dPRC

```
label = H , x = ( -28.700773 0.018189 43.312018 )
First Lyapunov coefficient = 6.461827e-03
label = LP, x = ( -26.127744 0.024296 43.740592 )
a=9.306236e-03
label = LP, x = ( -10.804133 0.126584 34.546930 )
a=-2.105189e-03
label = H , x = ( 7.947868 0.555741 197.796289 )
First Lyapunov coefficient = 8.882425e-04
Current step size too small (point 1134)
elapsed time  = 2.6 secs
npoints curve = 1134
first point found
tangent vector to first point found
Limit point cycle (period = 2.254127e+01, parameter = 2.005306e+02)
Normal form coefficient = -3.155002e-01

elapsed time  = 79.9 secs
npoints curve = 100
```

The code also produces Figure 14.

To further illustrate the computation of the PRC and the dPRC in CL_MatCont, we here supply code that does a LC continuation experiment in the `adaptx` system (the odefile is called `adaptx.m` and is located in the directory `Testruns/TestSystems`), and also computes the PRC and dPRC of the system:

```
[x0,v0]=init_EP_EP(@adaptx,[0;0;0],[-10;1],[1]);
opt=contset;opt=contset(opt,'Singularities',1);
```

```
[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);

x1=x(1:3,s(2).index);p=[x(end,s(2).index);1];
[x0,v0]=init_H_LC(@adaptx,x1,p,[1],1e-6,20,4);
opt = contset(opt,'MaxNumPoints',10);
opt = contset(opt,'Multipliers',1);
opt = contset(opt,'Adapt',1);
[xlc,vlc,slc,hlc,flc]=cont(@limitcycle,x0,v0,opt);
par=slc(end).data.parametervalues;
[x1,v1] = init_LC_LC(@adaptx, xlc, vlc, slc(end), par, 1, 20, 4);
opt = contset(opt,'PRC',1);
opt = contset(opt,'dPRC',1);
opt = contset(opt,'Input',1);
opt = contset(opt,'MaxNumPoints',20);
[xlc1,vlc1,slc1,hlc1,flc1]=cont(@limitcycle,x1,v1,opt);

fvector=flc1(:,20);
plot(fvector(22:102),'r')
hold on
plot(fvector(103:183),'b')
```

We note that the first 21 components (number of test intervals plus one) of the *fvector* contain the coarse mesh points of the discretization. The next 81 components (number of fine mesh points, i.e. number of test intervals times number of collocation points plus 1) contain the values of the PRC in the fine mesh points. The next 81 components contain the values of the dPRC in the fine mesh points.

This script is in the MatCont directory `Testruns` under the name `testadaptPRC`. The output is:

```
>> testadaptPRC
first point found
tangent vector to first point found
label = H , x = ( 0.000000 0.000000 0.000000 1.000002 )
First Lyapunov coefficient = -3.000001e-01

elapsed time  = 0.5 secs
npoints curve = 300
first point found
tangent vector to first point found
Limit point cycle (period = 6.283185e+00, parameter = 1.000000e+00)
Normal form coefficient = -1.306303e+00
Branch Point cycle(period = 6.283185e+00, parameter = 9.999996e-01)

elapsed time  = 13.3 secs
npoints curve = 10
first point found
tangent vector to first point found
```

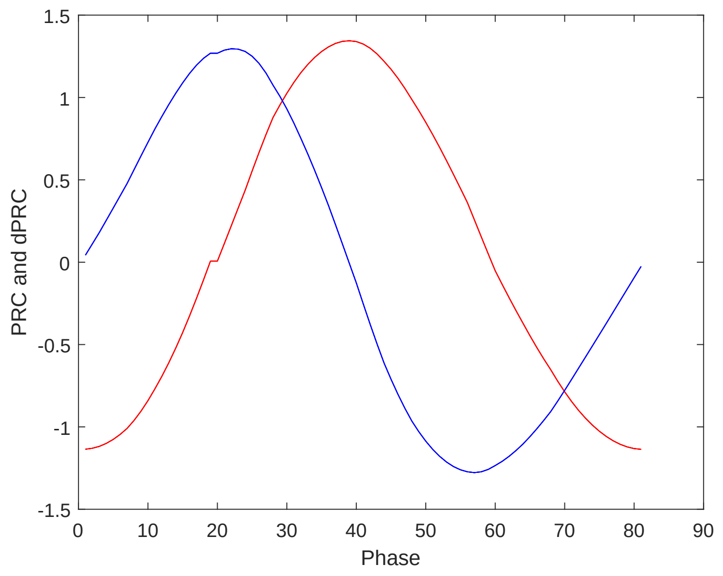


Figure 15: Phase response curve in the adaptx system: red=PRC, blue=dPRC

```
elapsed time  = 6.5 secs
npoints curve = 20
```

The code also produces Figure 15.

8 Continuation of codim 1 bifurcations

MatCont computes branches of codim 1 bifurcations of equilibria (namely, fold LP and Hopf H) and of limit cycles (fold of cycles LPC, period doubling PD and Neimark-Sacker NS). On these branches codimension 2 bifurcations of equilibria (CP,BT,ZH,HH,GH) and of limit cycles (CPC,R1,R3,R4,CH,LPNS,PDNS,R2, NSNS,LPPD and GPD) are detected and located, cf. the graph of adjacency in Figure 1.1. For the numerical methods used in the computation of the normal form coefficients, but also for the meaning and use of these coefficients we refer to [25] in the equilibrium case and to [9] in the limit cycle case. We note that the MatCont directory `LimitCycleCodim2` is exclusively devoted to the computation of normal form coefficients of codim 2 bifurcations of limit cycles.

8.1 Fold Continuation

8.1.1 Mathematical definition

In the toolbox fold curves are computed by *minimally extended defining systems* cf. [22], §4.1.2. The fold curve is defined by the following system

$$\left\{\begin{array}{lcl} f(u,\alpha) & = & 0, \\ g(u,\alpha) & = & 0, \end{array}\right. \tag{55}$$

where $(u,\alpha) \in \mathbf{R}^{n+2}$, while g is obtained by solving

$$\begin{pmatrix} f_u(u,\alpha) & w_{bor} \\ v_{bor}^T & 0 \end{pmatrix} \begin{pmatrix} v \\ g \end{pmatrix} = \begin{pmatrix} 0_n \\ 1 \end{pmatrix}, \tag{56}$$

and $w_{bor}, v_{bor} \in \mathbf{R}^n$ are chosen such that the matrix in (56) is nonsingular. An advantage of this method is that the derivatives of g can be obtained easily from the derivatives of $f_u(u,\alpha)$:

$$g_z = -w^T (f_u)_z v$$

where z is a state variable or an active parameter and w is obtained by solving

$$\begin{pmatrix} f_u^T(u,\alpha) & v_{bor} \\ w_{bor}^T & 0 \end{pmatrix} \begin{pmatrix} w \\ g \end{pmatrix} = \begin{pmatrix} 0_n \\ 1 \end{pmatrix}, \tag{57}$$

This method is implemented in the curve definition file `limitpoint.m`.

8.1.2 Bifurcations along a fold curve

In continuous-time systems there are four generic codim 2 bifurcations that can be detected along a fold curve:

- *Bogdanov - Takens*. We will denote this bifurcation by `BT`
- *Zero - Hopf* point, denoted by `ZH`
- *Cusp* point, denoted by `CP`
- *Branch* point, denoted by `BP`

To detect these singularities, we first define $bp+3$ test functions, where bp is the number of branch parameters:

- $\phi_1 = w^T v$ (cf. formula (39))
- $$\phi_2(u,\alpha) = \left(\begin{bmatrix} (2f_u(u,\alpha)\odot I_n) & w_1 \\ v_1^T & d \end{bmatrix} \backslash \begin{pmatrix} 0 \\ \ldots \\ 0 \\ 1 \end{pmatrix} \right)_{n+1} \tag{58}$$
- $\phi_3 = w^T f_{uu}[v,v]$
- $\phi_i = w^T f_{\beta_i}(u,\alpha)$

In these expressions v, w are the vectors computed in (56) and(57) respectively, $\odot$ is the bialternate matrix product, v_1, w_1 are $\frac{n(n-1)}{2}$ vectors chosen so that the square matrix in (58) is non-singular, and β_i (branch parameters) are components of α. The singularity matrix for $bp = 0$ is:

$$S = \begin{pmatrix} 0 & 0 & - \\ 1 & 0 & - \\ - & - & 0 \end{pmatrix} \tag{59}$$

The number of branch parameters is not fixed. If the number of branch parameters is 2 then this matrix has two more rows and columns. This singularity matrix is automatically extended:

$$S = \begin{pmatrix} 0 & 0 & - & - & - \\ 1 & 0 & - & - & - \\ - & - & 0 & - & - \\ - & - & - & 0 & - \\ - & - & - & - & 0 \end{pmatrix}$$

8.1.3 Fold initialization

The most natural way to start a fold curve continuation in the toolbox is from a limit point. As in the case of equilibria, a fold curve continuation cannot be done by just calling the continuer as:
`[x,v,s,h,f]=cont(@limitpoint, x0, v0, opt)`.
The fold curve file has to know

- which odefile to use,
- which parameters are active,
- the values of all parameters.

To initialize the continuation one first gives the following command:
`[x0,v0]=init_LP_LP(@odefile, xnew, p, ap` $(,bp)_{optional}$`)`.

In this command `xnew` must be a vector that contains the values of the state variables. `p` must contain the current values of all the parameters and `ap` must be the indices of the 2 active parameters. In the most natural situation where `x` is the matrix returned by the previous equilibrium curve continuation one starts to build x_{new} by the command `xnew=x(1:nphase,s(i).index)`, where `nphase` is the number of state variables and `s(i)` is the special point structure of the detected fold point on the equilibrium curve continuation. Next, the command `p(ap_old)=x(end,s(i).index);` replaces the old value of the free parameter in the previous run by the newly found parameter `p`. `odefile` specifies the ode-file to be used. `bp` are the optional indices of the branch parameters. It also works without entering a value for this field.

MatCont provides four other initializers which allow to continue a fold curve from a codim 2 equilibrium bifurcation, namely `init_BP_LP.m, init_BT_LP.m, init_CP_LP.m` and `init_ZH_LP.m`. In fact, these initializers are only added for ease of use: they refer back to `init_LP_LP.m`.

8.1.4 Adaptation

It is possible to adapt the problem while generating the fold curve. This call updates the auxiliary variables used in the defining system of the computed branch. The bordering vectors v_{bor} and w_{bor} may require updating since they must at least be such that the matrices in (56) and (57) are nonsingular. Updating is done by replacing v_{bor} and w_{bor} by the normalized vectors v, w computed in (56) and (57) respectively.

8.1.5 Example: a catalytic oscillator

For this example the following system (a catalytic oscillator) is used

$$\left\{\begin{array}{rcl} \dot{x} &=& 2q_1z^2 - 2q_5x^2 - q_3xy \\ \dot{y} &=& q_2z - q_6y - q_3xy \\ \dot{s} &=& q_4z - kq_4s \end{array}\right. \tag{60}$$

where $z = 1 - x - y - s$ and the parameters $q_1, q_2, q_3, q_4, q_5, q_6, k$ are introduced in that order in the odefile `cataloscill` in the MatCont directory `Testruns/TestSystems.`

The starting vector `x0` and its tangent vector `v0` are calculated from the following equilibrium curve continuation (q_2 is free).

```
p=[2.5;2.204678;10;0.0675;1;0.1;0.4];
ap1=[2];
[x0,v0]=init_EP_EP(@cataloscill,[0.001137;0.891483;0.062345],p,ap1);
opt=contset;
opt=contset(opt,'MaxStepSize',0.025);
opt=contset(opt,'MaxNumPoints',78);
opt=contset(opt,'Singularities',1);
[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
cpl(x,v,s,[4,1]);
```

This set of commands is run by executing the file `testequilcataloscill` in the MatCont directory `Testruns.` The command line output is the following:

```
>> testequilcataloscill
first point found
```

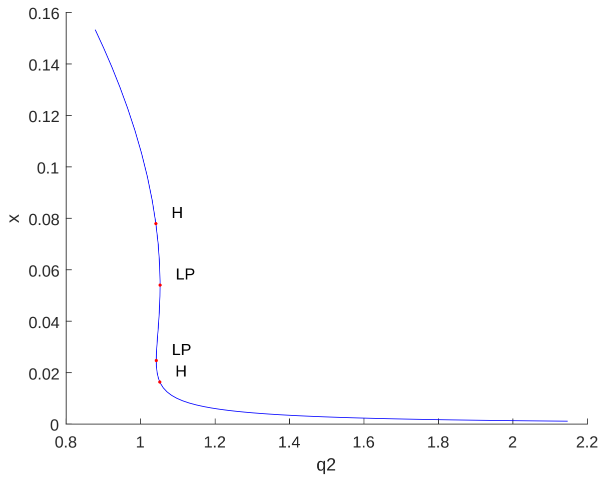


Figure 16: Computed equilibrium curve in the catalytic oscillar system (60)

```
tangent vector to first point found
label = H , x = ( 0.016357 0.523973 0.328336 1.051558 )
First Lyapunov coefficient = 1.070259e+01
label = LP, x = ( 0.024717 0.450257 0.375018 1.042049 )
a=-1.166509e-01
label = LP, x = ( 0.054030 0.302241 0.459807 1.052200 )
a=1.346534e-01
label = H , x = ( 0.077929 0.233063 0.492149 1.040991 )
First Lyapunov coefficient = 4.332247e+00

elapsed time  = 1.2 secs
npoints curve = 78
```

The results are plotted using the plot function `cpl` where the fourth argument selects the fourth and first components of the solution which are the parameter q_2 and the coordinate x. The results can be seen in Figure 16 where the axes labels are added manually.

We now add a forward and a backward fold continuation from the first LP detected on the previous equilibrium curve; q_2 and k are free in both runs.

```
p=[2.5;2.204678;10;0.0675;1;0.1;0.4];
ap1=[2];
[x0,v0]=init_EP_EP(@cataloscill,[0.001137;0.891483;0.062345],p,ap1);
opt=contset;
opt=contset(opt,'MaxStepSize',0.025);
opt=contset(opt,'MaxNumPoints',78);
opt=contset(opt,'Singularities',1);
[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
cpl(x,v,s,[4,1]);
'press any key'
```

```
pause
x1=x(1:3,s(3).index);
p(ap1)=x(end,s(3).index);
[x0,v0]=init_LP_LP(@cataloscill,x1,p,[2,7]);
[x2,v2,s2,h2,f2]=cont(@limitpoint,x0,v0,opt);
hold on;
cpl(x2,v2,s2,[4,1]);
'press any key'
pause
opt=contset(opt,'Backward',1);
[x3,v3,s3,h3,f3]=cont(@limitpoint,x0,v0,opt);
hold on
cpl(x3,v3,s3,[4,1]);
```

This set of commands is run by executing the file `testLPcataloscill` in the MatCont directory `Testruns`. The command line output is the following:

```
>> testLPcataloscill
first point found
tangent vector to first point found
label = H , x = ( 0.016357 0.523973 0.328336 1.051558 )
First Lyapunov coefficient = 1.070259e+01
label = LP, x = ( 0.024717 0.450257 0.375018 1.042049 )
a=-1.166509e-01
label = LP, x = ( 0.054030 0.302241 0.459807 1.052200 )
a=1.346534e-01
label = H , x = ( 0.077929 0.233063 0.492149 1.040991 )
First Lyapunov coefficient = 4.332247e+00

elapsed time  = 0.4 secs
npoints curve = 78

ans =

    'press any key'

first point found
tangent vector to first point found
label = CP , x = ( 0.035941 0.352004 0.451371 1.006408 0.355991 )
c=3.627908e-01
label = BT , x = ( 0.115909 0.315467 0.288437 1.417627 0.971397 )
(a,b)=(8.378438e-02, 2.136279e+00)

elapsed time  = 0.9 secs
npoints curve = 78

ans =
```

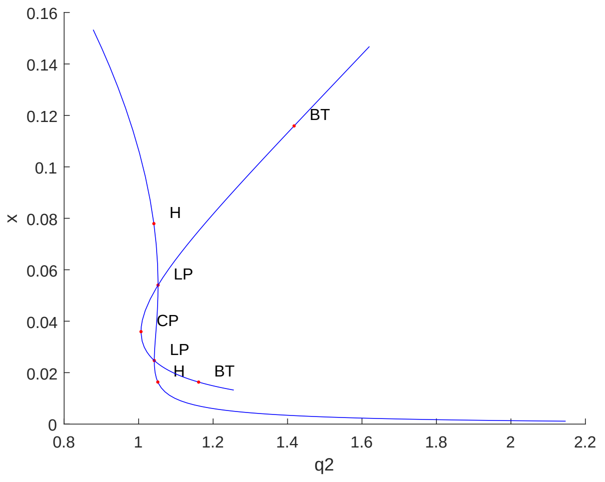


Figure 17: Computed equilibrium and fold curves in the catalytic oscillator system (60)

```
    'press any key'

first point found
tangent vector to first point found
label = BT , x = ( 0.016337 0.638410 0.200456 1.161199 0.722340 )
(a,b)=(-4.822577e-02, -1.937637e+00)

elapsed time  = 0.5 secs
npoints curve = 78
```

The results are plotted in Figure 17, where, as usual, the axis labels are added manually.

8.2 Hopf Continuation

8.2.1 Mathematical definition

In the MatCont / CL_MatCont toolbox Hopf curves are computed by *minimally extended defining systems*, cf. [22] §4.3.4. The Hopf curve is defined by the following system

$$\left\{ \begin{array}{rcl} f(u,\alpha) & = & 0, \\ g_{i_1 j_1}(u,\alpha,k) & = & 0 \\ g_{i_2 j_2}(u,\alpha,k) & = & 0 \end{array} \right. \tag{61}$$

with the unknowns $u, \alpha, k, (i_1, j_1, i_2, j_2) \in \{1,2\}$ and where $g = \begin{pmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{pmatrix}$ is obtained by solving

$$\begin{pmatrix} f_u^2 + kI_n & W_{bor} \\ V_{bor}^T & O \end{pmatrix} \begin{pmatrix} V \\ G \end{pmatrix} = \begin{pmatrix} 0_{n,2} \\ I_2 \end{pmatrix}, \tag{62}$$

where f_u has eigenvalues $\pm i\omega, \omega > 0$, $k = \omega^2$ and $V_{bor}, W_{bor} \in \mathbf{R}^{n\times 2}$ are chosen such that the matrix in (62) is nonsingular. $i_1, j_1, i_2, j_2, V_{bor}$ and W_{bor} are auxiliary variables that can be adapted. This method is implemented in the curve definition file `hopf.m`.

8.2.2 Bifurcations along a Hopf curve

In continuous-time systems there are four generic codim 2 bifurcations that can be detected along a Hopf curve:

- *Bogdanov - Takens.* We will denote this bifurcation by `BT`
- *Zero - Hopf* point, denoted by `ZH`
- *Double - Hopf* point, denoted by `HH`
- *Generalized Hopf* point, denoted by `GH`

To detect these singularities, we first define 4 test functions:

- $\phi_1 = k$
- $\phi_2 = \det(f_u)$
- $\phi_3 = \left(\begin{bmatrix} (2f_u(u,\alpha) \odot I_n) & w_1 \\ v_1^T & d \end{bmatrix} \backslash \begin{pmatrix} 0 \\ ... \\ 0 \\ 1 \end{pmatrix} \right)_{n+1}$
- $\phi_4 = l_1$ (first Lyapunov coefficient, see (40)),

where v_1, w_1 are carefully constructed and updated $\frac{n(n-1)}{2} \times 2$ matrices.

In this case the singularity matrix is:

$$S = \begin{pmatrix} 0 & 0 & - & - \\ 1 & 0 & - & - \\ - & - & 0 & - \\ - & - & - & 0 \end{pmatrix} \tag{63}$$

8.2.3 Hopf initialization

The most natural way to start the continuation of a Hopf bifurcation curve is to start it from a Hopf point, typically found on an equilibrium curve. The continuation of Hopf points cannot simply be started by using the following command:

```
[x,v,s,h,f]=cont(@hopf, x0, v0, opt)
```

The Hopf curve file has to know

- the values of all state variables,
- which ode file to use,
- which parameters are active,
- the values of all parameters.

Also an initial point `x0` has to be known. All this information can be supplied using the following command:
`[x0,v0]=init_H_H(@odefile, xnew, p, ap)`.

The input arguments `odefile, xnew, p, ap` are built exactly as in `init_LP_LP`. The output vector `x0` consists of x_{new} extended with the free parameters and paramter k from (62).

MatCont provides four other initializers which allow to continue a Hopf curve by starting from a codim 2 equilibrium point. These are `init_BT_H.m, init_GH_H.m`, `init_HH_H.m` and `init_ZH_H.m` These initializers are added merely for ease of use since they refer back to `init_H_H.m`. However, we note that two Hopf curves pass through a Hopf point. So it can be necessary to specify the tangent vector to the desired Hopf curve.

8.2.4 Adaptation

It is possible to adapt the problem while generating the Hopf curve. This call updates the auxiliary variables used in the defining system of the computed branch. The bordering matrices V_{bor} and W_{bor} may require updating since they must at least be such that the matrix in (62) is nonsingular. Updating of V_{bor} and W_{bor} is done exactly as in the LP case. Updating of i_1, j_1, i_2, j_2 is done in such a way that the linearized system of (61) is as well-conditioned as possible.

8.2.5 Example

For this example we again use the system (60). The starting vector is calculated from the first Hopf bifurcation point detected in the equilibrium continuation example (see 8.1.5). q_2 and k are free.

```
p=[2.5;2.204678;10;0.0675;1;0.1;0.4];
ap1=[2];
[x0,v0]=init_EP_EP(@cataloscill,[0.001137;0.891483;0.062345],p,ap1);
opt=contset;
opt=contset(opt,'MaxStepSize',0.025);
opt=contset(opt,'MaxNumPoints',78);
opt=contset(opt,'Singularities',1);
[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
cpl(x,v,s,[4,1]);
'press any key'
pause
x1=x(1:3,s(3).index);
p(ap1)=x(end,s(3).index);
[x0,v0]=init_LP_LP(@cataloscill,x1,p,[2,7]);
[x2,v2,s2,h2,f2]=cont(@limitpoint,x0,v0,opt);
hold on;
cpl(x2,v2,s2,[4,1]);
'press any key'
pause
opt=contset(opt,'Backward',1);
[x3,v3,s3,h3,f3]=cont(@limitpoint,x0,v0,opt);
```

```
hold on
cpl(x3,v3,s3,[4,1]);
'press any key'
pause
x1=x(1:3,s(2).index);
p(ap1)=x(end,s(2).index);
[x0,v0]=init_H_H(@cataloscill,x1,p,[2 7]);
opt=contset;
opt=contset(opt,'Singularities',1);
opt=contset(opt,'MaxStepsize',0.1);
opt=contset(opt,'backward',0);
[x4,v4,s4,h4,f4]=cont(@hopf,x0,v0,opt);

hold on;
cpl(x4,v4,s4,[4 1]);
```

This set of commands is executed by the file `testLPHopfcataloscill.m` in the directory `Testruns`. The command line output is

```
>> testLPHopfcataloscill
first point found
tangent vector to first point found
label = H , x = ( 0.016357 0.523973 0.328336 1.051558 )
First Lyapunov coefficient = 1.070259e+01
label = LP, x = ( 0.024717 0.450257 0.375018 1.042049 )
a=-1.166509e-01
label = LP, x = ( 0.054030 0.302241 0.459807 1.052200 )
a=1.346534e-01
label = H , x = ( 0.077929 0.233063 0.492149 1.040991 )
First Lyapunov coefficient = 4.332247e+00

elapsed time  = 0.3 secs
npoints curve = 78

ans =

    'press any key'

first point found
tangent vector to first point found
label = CP , x = ( 0.035941 0.352004 0.451371 1.006408 0.355991 )
c=3.627908e-01
label = BT , x = ( 0.115909 0.315467 0.288437 1.417627 0.971397 )
(a,b)=(8.378438e-02, 2.136279e+00)

elapsed time  = 0.5 secs
npoints curve = 78
```

```
ans =

    'press any key'

first point found
tangent vector to first point found
label = BT , x = ( 0.016337 0.638410 0.200456 1.161199 0.722340 )
(a,b)=(-4.822577e-02, -1.937637e+00)

elapsed time  = 0.4 secs
npoints curve = 78

ans =

    'press any key'

first point found
tangent vector to first point found
label = GH, x = ( 0.018022 0.368238 0.497968 0.891319 0.232487 0.003324 )
l2=-7.768956e+02
label = GH, x = ( 0.064311 0.211095 0.554870 0.924255 0.305879 0.003512 )
l2=-2.401240e+02
label = BT, x = ( 0.115909 0.315467 0.288437 1.417627 0.971396 0.000000 )
(a,b)=(8.378437e-02, 2.136280e+00)
label = BT, x = ( 0.016337 0.638410 0.200456 1.161199 0.722339 0.000000 )
(a,b)=(-4.822564e-02, -1.937633e+00)
Closed curve detected at step 159

elapsed time  = 1.6 secs
npoints curve = 159
>>
```

The results are plotted in Figure 18 using the standard plot function `cpl` where the fourth argument is used to select the fourth and first components of the solution which are the parameter q_2 and the coordinate x.
We not that the BT points are detected twice, namely on both the fold curve and the Hopf curve. Also, the Hopf curve is a closed curve but only the part between the two BT points that intersects the equilibrium curve consists of Hopf points. The other part consists of neutral saddle equilibria which have two real eigenvalues with sum zero.

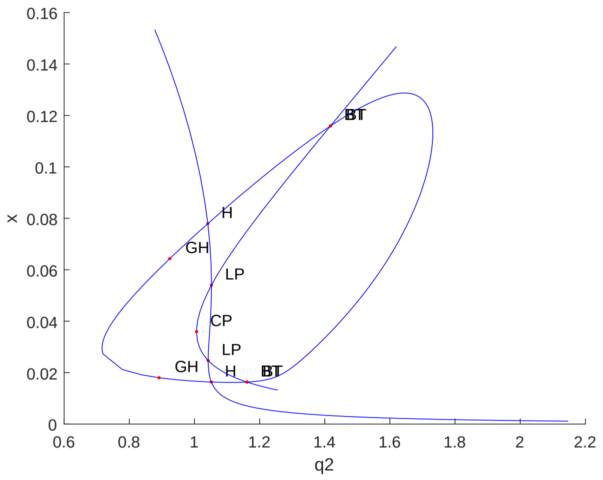


Figure 18: Computed equilibrium, fold and Hopf curves in the catalytic oscillator model (60)

8.3 Period Doubling

8.3.1 Mathematical definition

The Period Doubling bifurcation curve is defined by the following system

$$\begin{cases} \frac{du}{d\tau} - Tf(u,\alpha) & = 0 \\ u(0) - u(1) & = 0 \\ \int_0^1 \langle u(t), \dot{u}_{old}(t) \rangle dt & = 0 \\ G(u,T,\alpha) & = 0 \end{cases} \tag{64}$$

this is exactly the system defining limit cycles but with one extra constraint $G(u,T,\alpha) = 0$ where $G(u,T,\alpha)$ is defined as the solution component G of the system

$$\begin{cases} \dot{v}(\tau) - Tf_u(u,\alpha)v(\tau) + G\varphi(\tau) & = 0 \\ v(0) + v(1) & = 0 \\ \int_0^1 \langle \psi(\tau), v(\tau) \rangle d\tau & = 1 \end{cases} \tag{65}$$

which is exactly the same system as was used to detect the Period Doubling bifurcation.

Instead of using this functional G both systems can be combined in one larger system

$$\begin{cases} \frac{du}{d\tau} - Tf(u,\alpha) & = 0 \\ u(0) - u(1) & = 0 \\ \int_0^1 \langle u(t), \dot{u}_{old}(t) \rangle dt & = 0 \\ \dot{v}(\tau) - Tf_u(u,\alpha)v(\tau) & = 0 \\ v(0) + v(1) & = 0 \\ \int_0^1 \langle v_{old}(\tau), v(\tau) \rangle d\tau - 1 & = 0 \end{cases} \tag{66}$$

The first method (using system (64) and (65)) is implemented in the curve definition file `perioddoubling`. The other method is implemented in `perioddoubling2`. We will not use it further and it should not be confused with the continuation of the period doubled curve that branches off in a PD point on a limit cycle curve.

8.3.2 Output of a continuation of period doubling bifurcation points

The $x-$ output is in this case a matrix and each column corresponds to a computed period doubling limit cycle. The last three components of each column contain the value of the period T and the values of the two active parameters, in that order.

8.3.3 Bifurcations along a flip curve

In MatCont / CL_MatCont there are four generic codim 2 bifurcations that can be detected along a flip curve:

- *Strong 1:2 resonance.* We will denote this bifurcation with `R2`
- *Fold - flip*, *Limit Point - Period Doubling*or We will denote this bifurcation with `LPPD`
- *Flip - Neimark-Sacker*, denoted as `PDNS`
- *Generalized period doubling* point, denoted as `GPD`

To detect these singularities, we define 4 test functions:

- $\phi_1 = (v_1^*)_{W_1}^T L_{C\times M} v_{1M}$
- $\phi_2 = (\psi_1^*)_{W_1}^T (F(u_{0,1}(t)))_C$
- $\phi_3 = det\,((M \odot M) - I_n)$
- $\phi_4 = c$

where c is the coefficient defined in (52), M is the monodromy matrix and $\odot$ is the bialternate product.

The v_1's and ψ_1's are obtained as follows. For a given $\zeta \in \mathcal{C}^1([0,1],\mathbb{R}^n)$ we consider three different discretizations:

- $\zeta_M \in \mathbb{R}^{(Nm+1)n}$ the vector of values in the mesh points,
- $\zeta_C \in \mathbb{R}^{Nmn}$ the vector of values in the collocation points,
- $\zeta_W = \begin{bmatrix} \zeta_{W_1} \\ \zeta_{W_2} \end{bmatrix} \in \mathbb{R}^{Nmn} \times \mathbb{R}^n$ where ζ_{W_1} is the vector of values in the collocation points multiplied with the Gauss - Legendre weights and the lengths of the mesh intervals, and $\zeta_{W_2} = \zeta(0)$.

Formally we further introduce $L_{C\times M}$ which is a structured sparse matrix that converts a vector ζ_M of values in the mesh points into a vector ζ_C of values in the collocation points by $\zeta_C = L_{C\times M}\zeta_M$. We compute v_{1M} by solving

$$\begin{bmatrix} D - TA(t) \\ \delta_0 + \delta_1 \end{bmatrix}_{disc} v_{1M} = 0. \tag{67}$$

The normalization of v_{1M} is done by requiring $\sum_{i=1}^{N}\sum_{j=0}^{m} \sigma_j \langle (v_{1M})_{(i-1)m+j}, (v_{1M})_{(i-1)m+j} \rangle \Delta_i = 1$ where σ_j is the Gauss-Lagrange quadrature coefficient and Δ_i is the length of the i-th interval. By discretization we obtain

$$(v_{1W}^*)^T \begin{bmatrix} D - TA(t) \\ \delta_0 + \delta_1 \end{bmatrix}_{disc} = 0.$$

To normalize $(v_1^*)_{W_1}$ we require $\sum_{i=1}^N \sum_{j=1}^m |((v_1^*)_{W_1})_{ij}|_1 = 1$. Then $\int_0^1 \langle v_1^*(t), v_1(t)\rangle dt$ is approximated by $(v_1^*)_{W_1}^T L_{C\times M} v_{1M}$ and if this quantity is nonzero, v_{1W}^* is rescaled so that $\int_0^1 \langle v_1^*(t), v_1(t)\rangle dt = \frac{1}{2}$. We compute ψ_{1W}^* by solving

$$(\psi_{1W}^*)^T \left[\begin{array}{c} D - TA(t) \\ \delta_0 - \delta_1 \end{array} \right]_{disc} = 0$$

and normalize $\psi_{1W_1}^*$ by requiring $\sum_{i=1}^N \sum_{j=1}^m |((\psi_1^*)_{W_1})_{ij}|_1 = 1$. Then $\int_0^1 \langle \psi_1^*(t), F(u_{0,1}(t))\rangle dt$ is approximated by $(\psi_1^*)_{W_1}^T (F(u_{0,1}(t)))_C$ and if this quantity is nonzero, ψ_{1W}^* is rescaled so that $\int_0^1 \langle \psi_1^*(t), F(u_{0,1}(t))\rangle dt = 1$. a_1 can be computed as $(\psi_{W_1}^*)^T (B(t, v_{1M}, v_{1M}))_C$. The computation of $(h_{2,1})_M$ is done by solving

$$\left\{ \begin{array}{rcl} (D - TA(t))_{C\times M}(h_{2,1})_M &=& (B(t; v_{1M}, v_{1M}))_C + 2a_1(F(u_{0,1}(t)))_C,\ t \in [0,1] \\ (\delta(0) - \delta(1))(h_{2,1})_M &=& 0, \\ (\psi_{W_1}^*)^T L_{C\times M}(h_{2,1})_M &=& 0. \end{array} \right.$$

The expression for the normal form coefficient c becomes

$$\begin{array}{c} c = \frac{1}{3}((v_{1W_1}^*)^T (C(t; v_{1M}, \frac{1}{T} v_{1M}, v_{1M})_C + 3(B(t; v_{1M}, (h_{2,1})_M)_C \\ -6\frac{a_1}{T}(v_{1W_1}^*)^T (A(t)v_1(t))_C). \end{array}$$

The singularity matrix is:

$$S = \left(\begin{array}{ccc} 0 & - & - \\ - & 0 & - \\ 1 & 1 & 0 \end{array} \right). \tag{68}$$

8.3.4 Period doubling initialization

The natural way to start a Period Doubling bifurcation curve continuation is to start it from a Period Doubling bifurcation point on a limit cycle curve. This can be done using the following command: `[x0,v0]=init_PD_PD(@odefile, x, s, ap, ntst, ncol)`. `x` should be the `x` as returned by the previous limit cycle continuation. `s` is the special point structure of the detected Period Doubling point on the limit cycle curve. `odefile` specifies the ode-file to be used. `ap` is the active parameter and `ntst` and `ncol` are again the number of mesh and collocation points for the discretization.

MatCont provides four other initializers to start the continuation of PD curves from a codim2 bifurcation of limit cycles, namely `init_GPD_PD.m`, `init_LPPD_PD.m`, `init_PDNS_PD.m` and `init_R2_PD.m`. These initializers are added for ease of use; they only refer back to `init_PD_PD.m`.

We note that MatCont also provides an initializer `init_PD_PD2.m` which is merely an alternative to `init_PD_PD.m`; it is not to be confused with the initializer `init_PD_LC.m` which initializes the computation of the period doubled curve from a PD point.

8.3.5 Example

For this example the adaptive control system

$$\left\{ \begin{array}{rcl} \dot{x} &=& y \\ \dot{y} &=& z \\ \dot{z} &=& -\alpha z - \beta y - x + x^2 \end{array} \right. \tag{69}$$

from §7.7 is used again. The starting vector `x0` is calculated from the Period Doubling bifurcation detected in the limit cycle example (section 7.7) using `init_PD_PD`. Continuation is done using a call to the standard continuer with `perioddoubling` as curve definition file. The results are plotted using the standard plot function `cpl` where the fourth argument is used to select the 245th and 246th component of the solution which are the parameters α and β. The results can be seen in Figure 19. The labels of the plot are added manually. It can be tested by the command `testadapt3`.

```
>> [x0,v0]=init_EP_EP(@adaptx,[0;0;0],[-10;1],[1]);
>> opt = contset; opt = contset(opt,'Singularities',1);
>> [x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
first point found
tangent vector to first point found
label = H , x = ( 0.000000 0.000000 0.000000 1.000002 )
First Lyapunov coefficient = -3.000001e-01

elapsed time  = 0.7 secs
npoints curve = 300
>> x1=x(1:3,s(2).index);p=[x(end,s(2).index);1];
>> [x0,v0]=init_H_LC(@adaptx,x1,p,[1],1e-6,20,4);
>> opt = contset(opt,'MaxNumPoints',200);
>> opt = contset(opt,'Multipliers',1);
>> opt = contset(opt,'Adapt',1);
>> [xlc,vlc,slc,hlc,flc]=cont(@limitcycle,x0,v0,opt);
first point found
tangent vector to first point found
Limit point cycle (period = 6.283185e+00, parameter = 1.000000e+00)
Normal form coefficient = -1.306201e+00
Branch Point cycle(period = 6.283185e+00, parameter = 9.999996e-01)
Period Doubling (period = 6.364071e+00, parameter = 6.303020e-01)
Normal form coefficient = -4.267675e-02
Neutral Saddle Cycle (period = 6.433818e+00, parameter = 1.895459e-08)
Period Doubling (period = 6.364071e+00, parameter = -6.303020e-01)
Normal form coefficient = 4.268472e-02

elapsed time  = 58.3 secs
npoints curve = 200
>> plotcycle(xlc,vlc,slc,[size(xlc,1) 1 2]);
>> [x0,v0]=init_PD_PD(@adaptx,xlc,slc(4),[1 2],20,4);
>> opt = contset; opt = contset(opt,'Singularities',1);
>> [xpd,vpd,spd,hpd,fpd]=cont(@perioddoubling,x0,v0,opt);
first point found
tangent vector to first point found
Resonance 1:2 (period = 4.841835e+00, parameters = 5.317604e-09, 1.698711e+00)
(a,b)=(-7.330657e-02, 5.220090e-09)
Resonance 1:2 (period = 9.058318e+00, parameters = -3.045539e-07, 6.782783e-01)
(a,b)=(-1.060571e+01, -3.537444e-04)
```

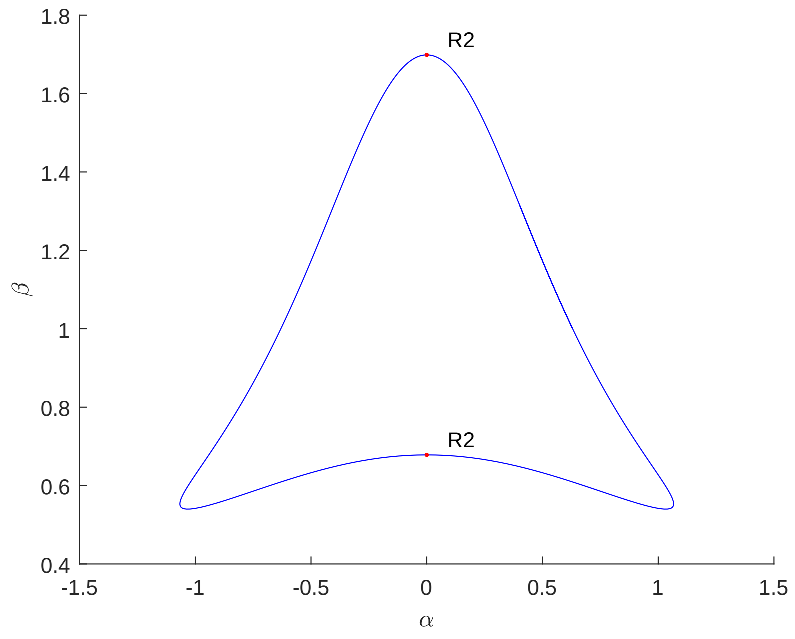


Figure 19: Computed Period Doubling curve

```
elapsed time  = 195.4 secs
npoints curve = 300
>> cpl(xpd,vpd,spd,[245 246]);
```

We note that `xpd` is a 246×300 matrix; each column corresponds to a computed period doubling limit cycle and gives the coordinates of all points of the fine mesh, i.e. $243 = (20 \times 4 + 1) \times 3$ values, plus the period T as the $244 - th$ component and the values of the two active parameter α, β as the 245−th and 246−the components.

8.4 Continuation of fold bifurcation of limit cycles

8.4.1 Mathematical definition

A Fold bifurcation of limit cycles (Limit Point of Cycles, LPC) generically corresponds to a turning point of a curve of limit cycles. It can be characterized by adding an extra constraint $G = 0$ to (47) where G is the Fold test function. The complete BVP defining a LPC point using the minimal extended system is

$$\begin{cases} \frac{du}{dt} - Tf(u,\alpha) & = 0 \\ u(0) - u(1) & = 0 \\ \int_0^1 \langle u(t), \dot{u}_{old}(t) \rangle dt & = 0 \\ G[u,T,\alpha] & = 0 \end{cases} \tag{70}$$

where G is defined by requiring

$$N^1 \begin{pmatrix} v \\ S \\ G \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \tag{71}$$

Here v is a function, S and G are scalars and

$$N^1 = \begin{bmatrix} D - Tf_u(u(t),\alpha) & -f(u(t),\alpha) & w_{01} \\ \delta_1 - \delta_0 & 0 & w_{02} \\ Int_{f(u(\cdot),\alpha)} & 0 & w_{03} \\ Int_{v_{01}} & v_{02} & 0 \end{bmatrix} \tag{72}$$

where the bordering functions v_{01}, w_{01}, vector w_{02} and scalars v_{02} and w_{03} are chosen so that N^1 is nonsingular [15]. This method (using system (71) and (72)) is implemented in the curve definition file `limitpointcycle`. The discretization is done using orthogonal collocation in the same way as it was done for limit cycles.

8.4.2 Bifurcations along a fold of cycles curve

In CL_MatCont there are five generic codim 2 bifurcations that can be detected along a fold of cycles curve:

- *Branch point.* We will denote this bifurcation with `BPC`
- *Strong 1:1 resonance.* We will denote this bifurcation with `R1`
- *Cusp* point, denoted by `CPC`
- *Fold - flip*, *Limit Point - Period Doubling*or We will denote this bifurcation by `LPPD`
- *Fold - Neimark-Sacker*, denoted by `LPNS`

A *Generalized Hopf* (`GH`) marks the end (or start) of an LPC curve.
To detect the generic singularities, we first define $bp+5$ test functions, where bp is the number of branch parameters:

- $\psi_i = w^T f_{\beta_i}(u,\alpha), i \in (1,...,bp)$
- $\psi_{bp+1} = (\varphi_1^*)^T_{W_1} L_{C\times M} v_{1M}$
- $\psi_{bp+2} = b$ (normal form coefficient)
- $\phi_{bp+3} = det\,(M + I_n)$
- $\psi_{bp+4} = det\,((M2 \odot M2) - I_n)$

In the ψ_i expressions w is the vector computed in (57) and β_i (branch parameter) is a component of α.

In the second expression ψ_{bp+1}, we compute v_{1M} by solving

$$\begin{bmatrix} D - TA(t) \\ \delta_0 - \delta_1 \\ \int_0^1 F(u_{0,1}(t)) L_{C\times M} \end{bmatrix}_{disc} v_{1M} = \begin{bmatrix} TF(u_{0,1}(t)) \\ 0 \\ 0 \end{bmatrix}_{disc}. \tag{73}$$

By discretization we obtain

$$(\varphi_{1W}^*)^T \begin{bmatrix} D - TA(t) \\ \delta_0 - \delta_1 \end{bmatrix}_{disc} = 0.$$

To normalize $(\varphi_1^*)_{W_1}$ we require $\sum_{i=1}^N \sum_{j=1}^m \left|((v_1^*)_{W_1})_{(i-1)m+j}\right|_1 = 1$. Then $\int_0^1 \langle \varphi_1^*(t), v_1(t)\rangle dt$ is approximated by $(\varphi_1^*)_{W_1}^T L_{C\times M} v_{1M}$ and if this quantity is nonzero, φ_{1W}^* is rescaled so that $\int_0^1 \langle \varphi_1^*(t), v_1(t)\rangle dt = 1$.

So the third expression for the normal form coefficient b becomes

$$b = \frac{1}{2}((\varphi_{1W_1}^*)^T((B(t; v_{1M}, v_{1M})_C + 2(A(t)v_1(t))_C)).$$

In the fourth expression, M is the monodromy matrix.

In the fifth expression, $M2$ is the $(n-2)\times(n-2)$ matrix that restricts the $n \times n$ matrix M to the space orthogonal to the two-dimensional left eigenspace of the two multipliers that are closest to 1.

The number of branch parameters is not fixed. If the number of branch parameters is 3 then this matrix has three more rows and columns. This singularity matrix is automatically extended:

$$S = \begin{pmatrix} 0 & - & - & - & - \\ - & 0 & - & - & - \\ - & - & 0 & - & - \\ - & - & - & 0 & - \\ - & - & - & 1 & 0 \end{pmatrix}.$$

8.4.3 Fold of cycles initialization

One way to start a Fold bifurcation of cycles continuation supported in the current version is to start it from a fold bifurcation point (LPC) on a limit cycle curve. This can be done using the following command: `[x0,v0]=init_LPC_LPC(@odefile, x, s, ap, ntst, ncol`$(, bp)_{optional}$`)`. `x` should be the `x` as returned by the previous limit cycle continuation. `s` is the special point structure of the detected Fold bifurcation point on the limit cycle curve. `odefile` specifies the ode-file to be used. `ap` is the active parameter and `ntst` and `ncol` are again the number of mesh and collocation points for the discretization. `bp` are the optional indices of the branch parameters. It also works without entering a value for this field: `[x0,v0]=init_LPC_LPC(@odefile, x, s, ap, ntst, ncol)`.

MatCont provides four other initializers which allow to start a continuation of folds of limit cycles by starting from an codim2 bifurcation of limit cycles. These are `init_CPC_LPC.m`, `init_LPNS_LPC.m`, `init_LPPD_LPC.m` and `init_R1_LPC.m`. These initializers are introduced for ease of use since they refer back to `init_LPC_LPC.m`.

A more interesting and indeed nontrivial initializer is `init_GH_LPC.m`. Computational methods to switch to nonhyperbolic cycles from codim 2 bifurcations of equilibria are discussed in [29].

8.4.4 Example: the fast Morris-Lecar equations

We consider the following system

$$\begin{cases} \dot{v} &= y - 0.5(v+0.5) - 2w(v+0.7) - m_\infty(v-1) \\ \dot{w} &= 1.15(w_\infty - w)\tau \end{cases} \tag{74}$$

where $m_\infty(v) = (1+\tanh((v+0.01)/0.15))/2$, $w_\infty(v) = (1+\tanh((v-z)/0.145))/2$ and $\tau = \cosh((v-0.1)/0.29)$. Here v and w are the state variables and y and z are the parameters. This is a modification of the fast subsystem of the Morris-Lecar equations studied in [37],[38]; the Morris-Lecar equations were introduced in [33] as a model for the electrical activity in the barnacle giant muscle fiber. In our model y corresponds to the slow variable in the fast Morris-Lecar equations; z is the potential for which $w_\infty(z) = \frac{1}{2}$.

We find a stable equilibrium (EP) for $y = 0.110472$ and $z = 0.1$ at $(0.04722, 0.32564)$ by time integration. We continue this equilibrium with free parameter y for decreasing values of y by running `testEquilMLfast.m`:

```
p=[0.11047;0.1];ap1=[1];
[x0,v0]=init_EP_EP(@MLfast,[0.047222;0.32564],p,ap1);
opt=contset;
opt=contset(opt,'Singularities',1);
opt=contset(opt,'MaxNumPoints',65);
opt=contset(opt,'MinStepsize',0.00001);
opt=contset(opt,'MaxStepsize',0.01);
opt=contset(opt,'Backward',1);

[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
cpl(x,v,s,[3 1]);
```

The output is as follows:

```
>> testEquilMLfast
first point found
tangent vector to first point found
label = H , x = ( 0.036756 0.294770 0.075658 )
First Lyapunov coefficient = 8.238955e+00
label = LP, x = ( -0.033738 0.136501 -0.020727 )
a=-1.036700e+01
Neutral saddle
label = H , x = ( -0.119894 0.045956 0.033207 )
label = LP, x = ( -0.244914 0.008514 0.083257 )
a=-2.697425e+00

elapsed time  = 0.1 secs
npoints curve = 65
```

We find a Hopf (H) bifurcation point at $y = 0.075659$, two limit points (LP) at $y = -0.020727$ and $y = 0.083257$ and a neutral saddle (H) at $y = 0.033207$. There are stable equilibria before the first H point and after the second LP point and unstable equilibria

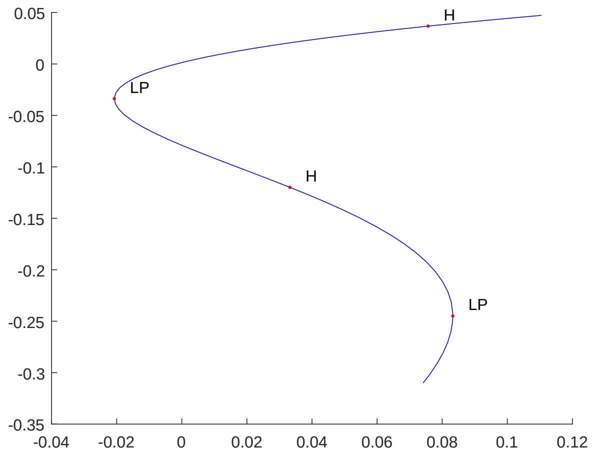


Figure 20: Computed equilibrium curve in the `MLfast` model

between the first H point and the second LP point. The Lyapunov coefficient in the first Hopf point $l_1 = 8.234573$ is positive which means that the periodic orbits are born unstable. The results are plotted using the plot function `cpl`, cf. §3.9 where the fourth argument is used to select the third and first component of the solution which are the parameter y and the coordinate v. The results can be seen in Figure 20.

The detected Hopf point is used to start a limit cycle continuation. We choose $N = 30$ test intervals and $m = 4$ collocation points for the discretization.

```
testEquilMLfast
x1=x(1:2,s(2).index);p=[x(end,s(2).index);0.1];
[x0,v0]=init_H_LC(@MLfast,x1,p,ap1,0.0001,30,4);
opt=contset;
opt=contset(opt,'IgnoreSingularity',1);
opt=contset(opt,'Singularities',1);
opt=contset(opt,'MaxNumPoints',50);
[x2,v2,s2,h2,f2]=cont(@limitcycle,x0,v0,opt);

plotcycle(x2,v2,s2,[1 2]);
```

The output is as follows:

```
>> testLCMLfast
first point found
tangent vector to first point found
label = H , x = ( 0.036756 0.294770 0.075658 )
First Lyapunov coefficient = 8.238955e+00
label = LP, x = ( -0.033738 0.136501 -0.020727 )
a=-1.036700e+01
label = H , x = ( -0.119894 0.045956 0.033207 )
label = LP, x = ( -0.244914 0.008514 0.083257 )
```

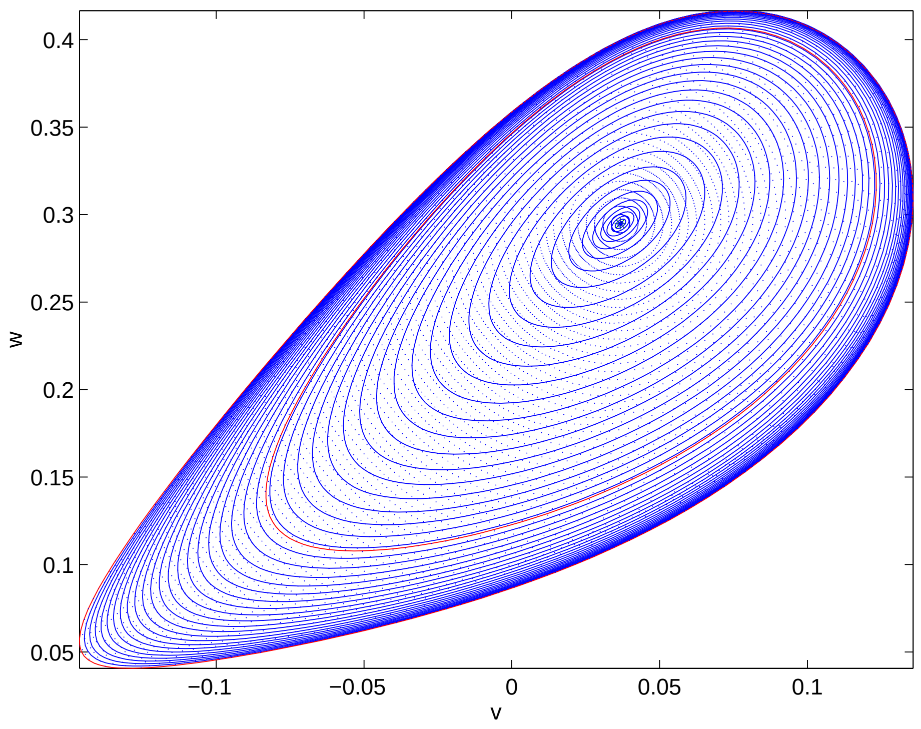


Figure 21: Computed limit cycle curve started from a Hopf bifurcation

```
a=-2.697425e+00

elapsed time  = 0.1 secs
npoints curve = 65
first point found
tangent vector to first point found
Limit point cycle (period = 4.222012e+00, parameter = 8.456948e-02)
Normal form coefficient = -2.334578e-01

elapsed time  = 4.0 secs
npoints curve = 50
```

The periodic orbit is initially unstable. We detect a limit point of cycles LPC at $y = 0.084569$. At this point the stability is gained. Afterwards the stability is preserved but the period tends to infinity and the periodic orbits end in a homoclinic orbit. The results can be seen in Figure 21.

We now compute a curve of fold bifurcations of limit cycles. The starting vector `x0` is calculated from the LPC on the previously computed curve of limit cycles, using `init_LPC_LPC`. Continuation is done using a call to the standard continuer with `limitpointcycle` as curve definition file. We free both y and z to continue the LPC curve through this LPC point. The computations are done by executing the script `testLPCMLfast.m`:

```
testEquilMLfast
```

```
x1=x(1:2,s(2).index);p=[x(end,s(2).index);0.1];
[x0,v0]=init_H_LC(@MLfast,x1,p,ap1,0.0001,30,4);
opt=contset;
opt=contset(opt,'IgnoreSingularity',1);
opt=contset(opt,'Singularities',1);
opt=contset(opt,'MaxNumPoints',50);
opt=contset(opt,'FunTolerance',0.0000001);
opt=contset(opt,'VarTolerance',0.0000001);
[x2,v2,s2,h2,f2]=cont(@limitcycle,x0,v0,opt);
[x0,v0]=init_LPC_LPC(@MLfast,x2,s2(2),[1 2],30,4);
opt=contset;
opt=contset(opt,'FunTolerance',0.0001);
opt=contset(opt,'VarTolerance',0.0001);
opt=contset(opt,'MaxNumPoints',30);
%opt=contset(opt,'Backward',1);
opt=contset(opt,'Singularities',1);
[x3,v3,s3,h3,f3]=cont(@limitpointcycle,x0,v0,opt);
plotcycle(x3,v3,s3,[1 2]);
```

The output is as follows:

```
>> testLPCMLfast
first point found
tangent vector to first point found
label = H , x = ( 0.036756 0.294770 0.075658 )
First Lyapunov coefficient = 8.238955e+00
label = LP, x = ( -0.033738 0.136501 -0.020727 )
a=-1.036700e+01
Neutral saddle
label = H , x = ( -0.119894 0.045956 0.033207 )
label = LP, x = ( -0.244914 0.008514 0.083257 )
a=-2.697425e+00

elapsed time  = 0.4 secs
npoints curve = 65
first point found
tangent vector to first point found
Limit point cycle (period = 4.222012e+00, parameter = 8.456948e-02)
Normal form coefficient = -2.334578e-01

elapsed time  = 3.9 secs
npoints curve = 50
first point found
tangent vector to first point found

elapsed time  = 7.1 secs
npoints curve = 30
```

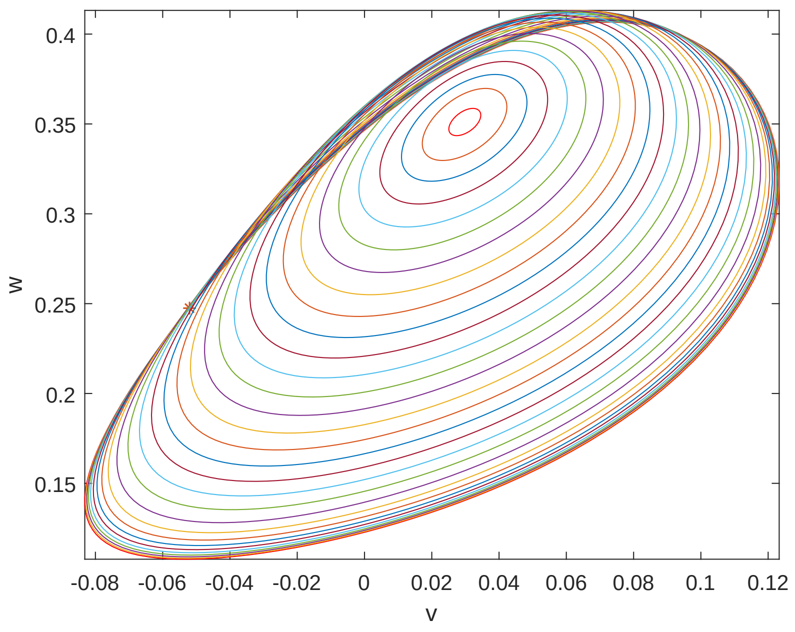


Figure 22: Computed fold of limit cycles curve started from a fold bifurcation of limitcycles

The results are plotted using the standard plot function `plotcycle` where the fourth argument is used to select the coordinates. The results can be seen in Figure 22. We note that it shrinks to a single point. The labels of the plot are added manually .

8.5 Continuation of torus bifurcation of limit cycles

8.5.1 Mathematical definition

A torus bifurcation of limit cycles (Neimark-Sacker, NS) generically corresponds to a bifurcation to an invariant torus, on which the flow contains periodic or quasi-periodic motion. It can be characterized by adding an extra constraint $G = 0$ to (47) where G is the torus test function which has four components from which two are selected. The complete BVP defining a NS point using a minimally extended system is

$$\left\{\begin{array}{ll} \frac{dx}{dt} - Tf(x,\alpha) & = 0 \\ x(0) - x(1) & = 0 \\ \int_0^1 \langle x(t), \dot{x}_{old}(t)\rangle dt & = 0 \\ G[x,T,\alpha,\kappa] & = 0 \end{array}\right. \tag{75}$$

where

$$G = \begin{pmatrix} G^{11} & G^{12} \\ G^{21} & G^{22} \end{pmatrix}$$

is defined by requiring

$$N^3 \begin{pmatrix} v^1 & v^2 \\ G^{11} & G^{12} \\ G^{21} & G^{22} \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{76}$$

Here v^1 and v^2 are functions and G^{11}, G^{12}, G^{21} and G^{22} are scalars and

$$N^3 = \begin{bmatrix} D - Tf_x(x(\cdot),\alpha) & w_{11} & w_{12} \\ \delta_0 - 2\kappa\delta_1 + \delta_2 & w_{21} & w_{22} \\ Int_{v_{01}} & 0 & 0 \\ Int_{v_{02}} & 0 & 0 \end{bmatrix} \tag{77}$$

where the bordering functions $v_{01}, v_{02}, w_{11}, w_{12}$, vectors $w_{21}$and w_{22} are chosen so that N^3 is nonsingular [15]. We note that an additional variable κ is introduced in (75). This method (using system (76) and (77)) is implemented in the curve definition file `neimarksacker.m`. The discretization is done using orthogonal collocation over the interval [0 2]. The additional variable κ is introduced as the last continuation variable of `neimarksacker.m`, after the state variables, the period and the two active system parameters.

8.5.2 Bifurcations along a Neimark-Sacker curve

In continuous-time systems there are eight generic codim 2 bifurcations that can be detected along a torus curve:

- *1:1 resonance*. We will denote this bifurcation by `R1`
- *2:1 resonance* point, denoted by `R2`
- *3:1 resonance* point, denoted by `R3`
- *4:1 resonance* point, denoted by `R4`
- *Fold-Neimarksacker* point, denoted by `LPNS`
- *Chenciner* point, denoted by `CH`.
- *Flip-Neimarksacker* point, denoted by `PDNS`
- *Double Neimarksacker bifurcation* point, denoted by `NSNS`

To detect these singularities, we first define 6 test functions:

- $\psi_1 = \kappa - 1$ (cf. formula (39))
- $\psi_2 = \kappa + 1$
- $\psi_3 = \kappa - 1/2$
- $\psi_4 = \kappa$
- $\psi_5 = (v_1^*)_{W_1}^T L_{C\times M} v_{1M}$
- $\psi_6 = Re(d)$

- $\psi_7 = det\,(M + I_n)$
- $\psi_8 = det\,((M2 \odot M2) - I_n)$

where v_{1M} is computed by solving

$$\left[\begin{array}{c} D - TA(t) + i\theta I \\ \delta_0 - \delta_1 \end{array} \right]_D v_{1M} = 0. \tag{78}$$

The normalization of v_{1M} is done by requiring $\sum_{i=1}^{N} \sum_{j=0}^{m} \sigma_j \langle (v_{1M})_{ij}, (v_{1M})_{ij} \rangle t_i = 1$ where σ_j is the Gauss-Lagrange quadrature coefficient. By discretization we obtain

$$(v_{1W}^*)^H \left[\begin{array}{c} D - TA(t) + i\theta \\ \delta_0 + \delta_1 \end{array} \right]_{disc} = 0.$$

To normalize $(v_1^*)_{W_1}$ we require $\sum_{i=1}^{N} \sum_{j=1}^{m} |((v_1^*)_{W_1})_{ij}|_1 = 1$. Then $\int_0^1 \langle v_1^*(t), v_1(t) \rangle dt$ is approximated by $(v_1^*)_{W_1}^T L_{C\times M} v_{1M}$ and if this quantity is nonzero, v_{1W}^* is rescaled so that $\int_0^1 \langle v_1^*(t), v_1(t) \rangle dt = 1$. We compute φ_{1W}^* by solving

$$(\varphi_{1W}^*)^T \left[\begin{array}{c} D - TA(t) \\ \delta_0 - \delta_1 \end{array} \right]_{disc} = 0$$

and normalize $\varphi_{1W_1}^*$ by requiring $\sum_{i=1}^{N} \sum_{j=1}^{m} |((\varphi_1^*)_{W_1})_{ij}|_1 = 1$. Then $\int_0^1 \langle \varphi_1^*(t), F(u_{0,1}(t)) \rangle dt$ is approximated by $(\varphi_1^*)_{W_1}^T (F(u_{0,1}(t)))_C$ and if this quantity is nonzero, φ_{1W}^* is rescaled so that $\int_0^1 \langle \varphi_1^*(t), F(u_{0,1}(t)) \rangle dt = 1$. We compute $h_{20,1M}$ by solving

$$\left[\begin{array}{c} D - TA(t) + 2i\theta I \\ \delta_0 - \delta_1 \end{array} \right]_{disc} h_{20,1M} = \left[\begin{array}{c} B(t, v_{1M}(t), v_{1M}(t)) \\ 0 \end{array} \right].$$

a_1 can be computed as $(\varphi_{W_1}^*)^T (B(t, v_{1M}, \overline{v}_{1M}))_C$.

The computation of $(h_{11,1})_M$ is done by solving

$$\left[\begin{array}{c} (D - TA(t))_{C\times M} \\ \delta(0) - \delta(1) \\ (\varphi_{W_1}^*)^T L_{C\times M} \end{array} \right] (h_{11,1})_M = \left[\begin{array}{c} B(t; v_{1M}, \overline{v}_{1M}))_C - a_1 (F(u_{0,1}(t)))_C \\ 0 \\ 0 \end{array} \right]$$

The expression for the normal form coefficient d becomes

$$\begin{array}{c} d = \frac{1}{2}((v_{1W_1}^*)^T, (B(t; h_{11,1M}, v_{1M}))_C + (B(t; h_{20,1M}, \overline{v}_{1M}))_C + \frac{1}{T}(C(t; v_{1M}, v_{1M}, \overline{v}_{1M}))_C) \\ -\frac{a_1}{T}(v_{1W_1}^*)^T (A(t) v_1(t))_C + \frac{i a_1 \theta}{T^2}. \end{array}$$

In the 7th test function, M is the monodromy matrix.

In the 8th test function, $M2$ is the $(n-2)\times(n-2)$ matrix which restricts the $n \times n$ matrix M to the subspace orthogonal to the two-dimensional left eigenspace of the Neimark-Sacker eigenvalues.

The singularity matrix is:

$$S = \left(\begin{array}{cccccc} 0 & - & - & - & - & - \\ - & 0 & - & - & - & - \\ - & - & 0 & - & - & - \\ - & - & - & 0 & - & - \\ - & - & - & - & 0 & - \\ - & - & - & - & 1 & 0 \end{array} \right). \tag{79}$$

8.5.3 Torus bifurcation initialization

One way to start a continuation of torus bifurcations of cycles supported in the current version is to start it from a torus bifurcation point or neimarksacker point (NS) on a limit cycle curve. This can be done using the following command: `[x0,v0]=init_NS_NS(@odefile, x, s, ap, ntst, ncol)`. `x` should be the `x` as returned by the previous limit cycle continuation. `s` is the special point structure of the detected torus bifurcation point on the limit cycle curve. `odefile` specifies the ode-file to be used. `ap` is the array containing the two active parameters and `ntst` and `ncol` are again the number of mesh and collocation points for the discretization.

MatCont provides seven other initializers to start the continuation of torus bifurcations from codim2 bifurcations of limit cycles. These all have the form `init_XYX_NS` where `XYX` is one of {CH,LP,PD,R1,R2,R3,R4}. They are introduced for ease of use since they all refer back to `init_NS_NS.m`.

More interesting and indeed nontrivial initializers are `init_HH_NS1.m, init_HH_NS2.m` and `init_ZH_NS`. Indeed, two different torus bifurcation curves can generically cross a HH point. Computational methods to switch to nonhyperbolic cycles from codim 2 bifurcations of equilibria are discussed in [29].

8.5.4 Example: an autonomous electronic circuit

We consider the following model of an autonomous electronic circuit [18] where x, y and z are state variables and $\beta, \gamma, r, a_3, b_3, \nu$ are parameters :

$$\left\{\begin{array}{lcl} \dot{x} & = & (-(\beta+\nu)x + \beta y - a_3x^3 + b_3(y-x)^3)/r \\ \dot{y} & = & \beta x - (\beta+\gamma)y - z - b_3(y-x)^3 \\ \dot{z} & = & y \end{array}\right. \tag{80}$$

A torus bifurcation in this system is described in the manual [14]. It is found by starting an equilibrium curve from the trival equilibrium point ($x = y = z = 0$) at $\beta = 0.5$, $\gamma = -0.6$, $r = 0.6$, $a_3 = 0.32858$, $b_3 = 0.93358$, $\nu = -0.9$. The free parameter is ν and the branch is the trivial one ($x = y = z = 0$). On this branch a Hopf bifurcation is detected at $\nu = -0.58934$. On the emerging branch of limit cycles a branch point of limit cycles is found; by continuing the newly found branch one detects a torus bifurcation of periodic orbits.

We proceed in a somewhat different way; to avoid the branch point of periodic orbits we start with a slightly perturbed system where the last equation of (80) is replaced by

$$\dot{z} = y + \epsilon$$

where ϵ is a new (artificial) parameter with initially $\epsilon = 0.001$. The perturbed system is introduced in MatCont in the odefile `torBPC.m` in the directory `Testruns/TestSystems`. It contains a user function ϵ so that the zeroes of the user function correspond to solutions of the unperturbed system (80).

The trivial solution is replaced by the equilibrium solution $x = 0.00125$, $y = -0.001$, $z = 0.00052502$ of the perturbed system `torBPC.m` and we compute a branch of equilibria with free parameter ν in the now standard way (note that ν is the sixth parameter, ϵ is the seventh):

```
>> p=[0.5;-0.6;0.6;0.32858;0.93358;-0.9;0.001];
>> [x0,v0]=init_EP_EP(@torBPC,[0.00125;-0.001;0.00052502],p,[6]);
>> opt=contset; opt= contset(opt,'Singularities',1);
>> opt=contset(opt,'MaxNumPoints',10);
>> [x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
first point found
tangent vector to first point found
label = H , x = ( 0.005604 -0.001000 0.002702 -0.589286 )
First Lyapunov coefficient = -4.548985e-001

elapsed time  = 0.1 secs
npoints curve = 10
```

A Hopf point is found for $\nu = -0.58928$ for the state values $x = 0.0056037$, $y = -0.001$, $z = 0.0027021$. We start a curve of periodic orbits from this Hopf point, using $N = 25$ test intervals and $m = 4$ collocation points and choosing ν as the free parameter. The results can be seen in Figure 23.

```
>> x1=x(1:3,s(2).index);p(6)= x(end,s(2).index);
>> [x0,v0]=init_H_LC(@torBPC,x1,p,[6],0.0001,25,4);
>> opt=contset; opt= contset(opt,'Singularities',1);
>> opt=contset(opt,'MaxNumPoints',50);
>> opt=contset(opt,'Multipliers',1);
>> [x,v,s,h,f]=cont(@limitcycle,x0,v0,opt);
first point found
tangent vector to first point found
Limit point cycle (period = 8.411855e+000, parameter = -5.844928e-001)
Normal form coefficient = 1.788080e-001
Neimark-Sacker (period = 8.861103e+000, parameter = -5.957506e-001)
Normal form coefficient = 2.674115e-003
Period Doubling (period = 9.256846e+000, parameter = -6.146817e-001)
Normal form coefficient = -6.068973e-003

elapsed time  = 28.3 secs
npoints curve = 50
>> plotcycle(x,v,s,[1 2]);
```

The previous computations are done by running the script testtorBPC1, including drawing Figure 23 in which the Hopf point and the three bifurcations of limit cycles are clearly visible. The axis labels were added manually.

In particular, we detect a torus bifurcation point at $\nu = -0.59575$. To recover the torus bifurcation point of (80) we continue the torus bifurcation in two parameters ν, ϵ. With the aid of the user function ϵ we will now locate a NS bifurcation of (80). The starting vector `x1` is calculated from the NS on this branch using `init_NS_NS`. Continuation is done using a call to the standard continuer with `neimarksacker` as curve definition file:

```
p=[0.5;-0.6;0.6;0.32858;0.93358;-0.9;0.001];
x=[0.00125;-0.001;0.00052502];
[x0,v0]=init_EP_EP(@torBPC,x,p,[6]);
opt=contset;
```

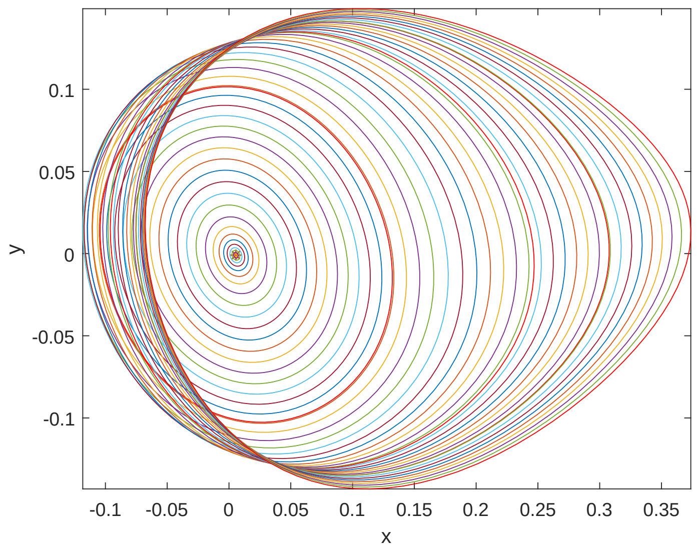


Figure 23: Computed limit cycle curve started from a Hopf bifurcation

```
opt=contset(opt,'Singularities',1);
opt=contset(opt,'MaxNumPoints',10);
[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
x1=x(1:3,s(2).index);
p(6)=x(end,s(2).index);
[x0,v0]=init_H_LC(@torBPC,x1,p,[6],0.0001,25,4);
opt=contset;
opt=contset(opt,'Singularities',1);
opt=contset(opt,'Multipliers',1);
opt=contset(opt,'MaxNumPoints',50);
[x,v,s,h,f]=cont(@limitcycle,x0,v0,opt);
[x1,v1]=init_NS_NS(@torBPC,x,s(3),[6 7],25,4);
opt=contset;
opt=contset(opt,'VarTolerance',1e-4);
opt=contset(opt,'FunTolerance',1e-4);
opt=contset(opt,'Userfunctions',1);
UserInfo.name='epsilon0';
UserInfo.state=1;
UserInfo.label='E0';
opt=contset(opt,'UserfunctionsInfo',UserInfo);
opt=contset(opt,'Backward',1);
opt=contset(opt,'MaxNumPoints',16);
[xns1,vns1,sns1,hns1,fns1]=cont(@neimarksacker,x1,v1,opt);
plotcycle(xns1,vns1,sns1,[1 2]);
```

This continuation is done by running the script `testtorBPC2`. The output is the following:

```
>> testtorBPC2
first point found
tangent vector to first point found
label = H , x = ( 0.005604 -0.001000 0.002702 -0.589286 )
First Lyapunov coefficient = -4.549030e-01

elapsed time  = 0.7 secs
npoints curve = 10
first point found
tangent vector to first point found
Limit point cycle (period = 8.411870e+00, parameter = -5.844928e-01)
Normal form coefficient = 1.788366e-01
Neimark-Sacker (period = 8.861100e+00, parameter = -5.957504e-01)
Normal form coefficient = 2.674115e-03
Period Doubling (period = 9.256846e+00, parameter = -6.146817e-01)
Normal form coefficient = -6.068982e-03

elapsed time  = 11.5 secs
npoints curve = 50
first point found
tangent vector to first point found
label = E0, x = ( 0.046835 0.141698 0.046209 ...
... 0.141698 0.046209 8.793572 -0.591612 -0.000006 0.822406 )

elapsed time  = 9.1 secs
npoints curve = 16
```

We note that the point with label E0 is a torus bifurcation point of (80) with $\nu = -0.591612$, $\epsilon = -0.000006 \approx 0$, and period 0.822406. The orbits are shown in Figure 24. The axis labels were added manually.

Finally, we continue numerically the NS orbit with two free parameters β, ν and find, interestingly, that the NS orbit shrinks to a single point. The results are plotted using the standard plot function `plotcycle` where the fourth argument is used to select the coordinates. The plot is shown in Figure 25 (the axis labels were added manually). This computation is executed by running `testtorBPC3`:

```
p=[0.5;-0.6;0.6;0.32858;0.93358;-0.9;0.001];
x=[0.00125;-0.001;0.00052502];
[x0,v0]=init_EP_EP(@torBPC,x,p,[6]);
opt=contset;
opt=contset(opt,'Singularities',1);
opt=contset(opt,'MaxNumPoints',10);
[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
x1=x(1:3,s(2).index);
p(6)=x(end,s(2).index);
[x0,v0]=init_H_LC(@torBPC,x1,p,[6],0.0001,25,4);
opt=contset;
opt=contset(opt,'Singularities',1);
```

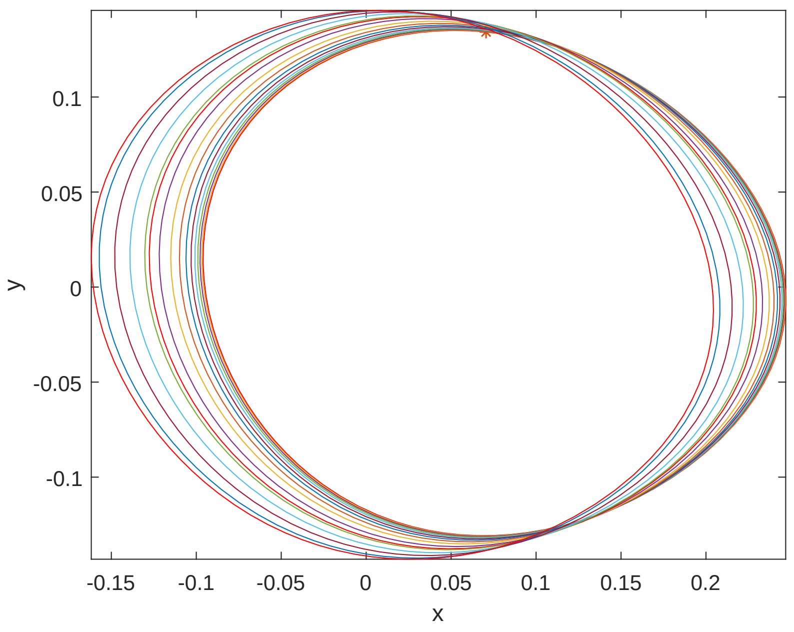


Figure 24: Computed torus bifurcation curve started from a torus bifurcation of limit cycles

```
opt=contset(opt,'Multipliers',1);
opt=contset(opt,'MaxNumPoints',50);
[x,v,s,h,f]=cont(@limitcycle,x0,v0,opt);
[x2,v2]=init_NS_NS(@torBPC,x,s(3),[1 6],50,4);
opt=contset; opt=contset(opt,'VarTolerance',1e-4);
opt=contset(opt,'FunTolerance',1e-4);
opt=contset(opt,'MaxNumPoints',50);
[xns2,vns2,sns2,hns,fns]=cont(@neimarksacker,x2,v2,opt);
plotcycle(xns2,vns2,sns2,[1 2]);
```

The output is as follows:

```
>> testtorBPC3
first point found
tangent vector to first point found
label = H , x = ( 0.005604 -0.001000 0.002702 -0.589286 )
First Lyapunov coefficient = -4.549030e-01

elapsed time  = 0.0 secs
npoints curve = 10
first point found
tangent vector to first point found
Limit point cycle (period = 8.411870e+00, parameter = -5.844928e-01)
Normal form coefficient = 1.788366e-01
Neimark-Sacker (period = 8.861100e+00, parameter = -5.957504e-01)
```

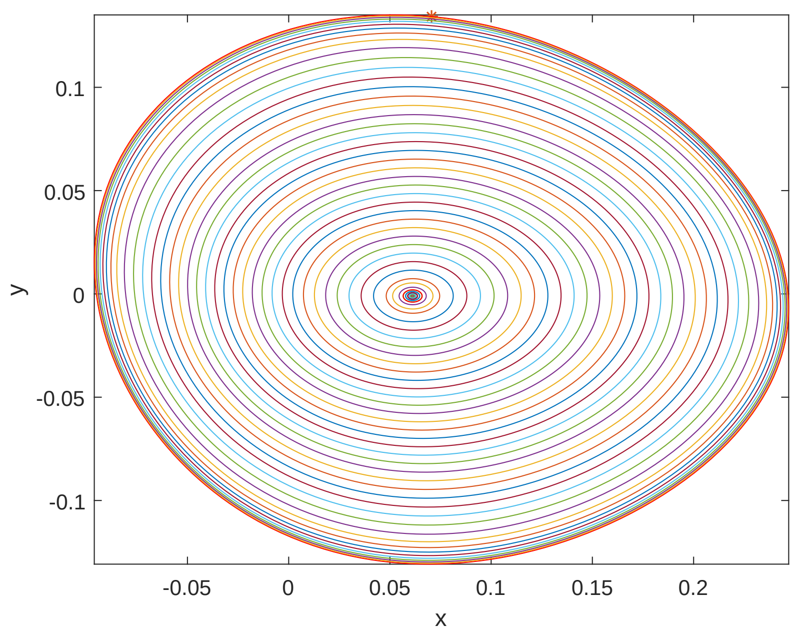


Figure 25: Computed torus bifurcation curve that shrinks to a single point

```
Normal form coefficient = 2.674115e-03
Period Doubling (period = 9.256846e+00, parameter = -6.146817e-01)
Normal form coefficient = -6.068982e-03

elapsed time  = 14.1 secs
npoints curve = 50
first point found
tangent vector to first point found

elapsed time  = 124.5 secs
npoints curve = 50
```

Here $xns2$ is a 607×50 matrix. The last four components of each column are, in that order, the period of the orbit, the values of the two free parameters, and the value of the additional variable κ that is introduced in (75).

9 Continuation of codim 2 bifurcations

9.1 Branch Point Continuation

Although the predecessor CONTENT [26] of MatCont included the continuation of codimension 2 bifurcations of equilibria in three free parameters, this is so far not included in MatCont. Though the initialization and continuation itself would not be too difficult, the computation and interpretation of the normal form coefficients of the codim3 bifurcations would be unfeasible. Nevertheless, MatCont includes the continuation of branch points of equilibria and limit cycles in three parameters since this appears to be useful in some studies of models with equivariance.

9.1.1 Mathematical definition

In the toolbox branch point curves are computed by *minimally extended defining systems*, cf. [22], §4.1.2. The branch point curve is defined by the following system

$$\left\{\begin{array}{rcl} f(u,\alpha) & = & 0, \\ g_1(u,\alpha) & = & 0, \\ g_2(u,\alpha) & = & 0, \end{array}\right. \tag{81}$$

where $(u,\alpha) \in \mathbf{R}^{n+2}$, while g_1 and g_2 are obtained by solving

$$N^4 \begin{pmatrix} v_{11} & v_{21} \\ v_{12} & v_{22} \\ g_1 & g_2 \end{pmatrix} = \begin{pmatrix} 0_n & 0_n \\ 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{82}$$

Here v_{11} and v_{21} are functions and v_{12}, v_{22}, g_1 and g_2 are scalars and

$$N^4 = \begin{bmatrix} f_u(u,\alpha) & f_\beta(u,\alpha) & w_{01} \\ v_0^{11T} & v_0^{12T} & 0 \\ v_0^{21T} & v_0^{22T} & 0 \end{bmatrix}$$

where the bordering functions $v_0^{11}, v_0^{21}, w_{01}$ and scalars v_0^{12}, v_0^{22} are chosen so that N^4 is nonsingular. This method is implemented in the curve definition file `branchpoint.m`.

9.1.2 Bifurcations

In the current version no bifurcations are detected.

9.1.3 Branch Point initialization

The only way to start a continuation of branch points supported in the current version is to start it from a Branch point detected on an equilibrium curve or on a fold curve. This can be done using the following statement: `[x0,v0]=init_BP_BP(@odefile, x, p, ap, bp)`. This routine stores its information in a global stucture `bpds`. The result of `init_BP_BP` contains a vector `x0` with the state variables and the three active parameters and a vector `v0` that is empty. Here `odefile` is the ode-file to be used, `x` is a vector of state variables containing the values of the state variables returned by a previous equilibrium curve or fold curve continuation, `p` is the vector containing the current values of the parameters, `ap` is the vector containing the indices of the 3 active parameters and `bp` is the index of the branch parameter.

9.1.4 Example

For this example we use a model of a continuous stirred tank reactor (`cstr.m`):

$$\left\{ \begin{array}{rcl} \dot{x} & = & \alpha_3 - (1+\lambda)x + \lambda\alpha_1/(1+\lambda\alpha_2 * e^{-\alpha_4 x/(1+x)}) \end{array} \right. \qquad (83)$$

where $\alpha_1 = 10 - 9*\beta + \gamma, \alpha_2 = 10 - 9*\beta$ and $\alpha_3 = -0.9 + 0.4*\beta$. The model is coded in such a way that the parameters are $\lambda, \beta, \gamma, \alpha_4$, in that order.

It is easily seen that $x = -0.9$ is an equilibrium point of the system for the choice $(0, 0, 0, 3)$ of the parameters. From this we can start an equilbrium continuation with λ (the first parameter) free.

```
p=[0;0;0;3];ap1=[1];
[x0,v0]=init_EP_EP(@cstr,[-0.9],p,ap1);
opt=contset;
opt=contset(opt,'VarTolerance',1e-3);
opt=contset(opt,'FunTolerance',1e-3);
opt=contset(opt,'MaxNumPoints',50);
opt=contset(opt,'Singularities',1);
[x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
first point found
tangent vector to first point found
label = LP, x = ( -0.143564 1.250669 )
a=1.550147e+000
label = LP, x = ( 0.393180 0.377651 )
a=-7.370472e-001
cpl(x,v,s,[2,1]);
```

The results are plotted using the plot function `cpl` where the fourth argument is used to select the second and first components of the solution which are the parameter λ and the coordinate x. The resulting curve is a part of Figure 26. These computations can be done by running the script cstr1.

We start a fold continuation from the second LP detected on the previous equilibrium curve; λ and β are free in this run.

```
x1=x(1,s(3).index);
p(ap1)=x(end,s(3).index);
[x0,v0]=init_LP_LP(@cstr,x1,p,[1 2],[1 2 3 4]);
opt=contset(opt,'MaxNumPoints',300);
[x2,v2,s2,h2,f2]=cont(@limitpoint,x0,v0,opt);
first point found
tangent vector to first point found
label = BP1, x = ( 2.018621 0.581081 -4.709219 )
label = CP , x = ( 0.259553 1.968966 -0.090655 )
c=-8.847089e-001
label = BP1, x = ( 0.030643 1.772454 -0.127542 )
label = BP4, x = ( -0.000009 1.707401 -0.124964 )
label = CP , x = ( -0.173872 0.405524 0.608093 )
```

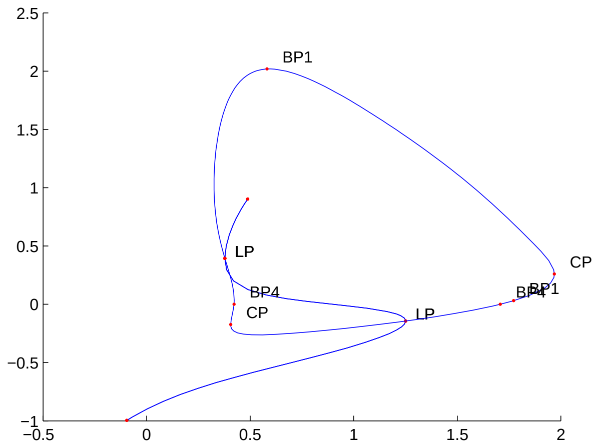


Figure 26: Computed fold curve

```
c=-2.263137e+000
label = BP4, x = ( -0.000000 0.421692 0.528995 )
Closed curve detected at step 164

elapsed time  = 0.5 secs
npoints curve = 164
hold on;
cpl(x2,v2,s2,[2,1]);
```

These computations can be done by running the script cstr2. The results are plotted using the standard plot function `cpl` where the fourth argument is used to select the second and first components of the solution which are the parameter λ and the coordinate x. The results can be seen in Figure 26.

Finally, we continue numerically the BP curves with three free parameters λ, β and γ. The BP curves are started respectively from the first BP1 point (λ is the branch parameter) and the first BP4 point (α_4 is the branch parameter) detected on the previous fold curve. The results are plotted using the standard plot function `cpl` where the fourth argument is used to select the coordinates. A graphical representation of this phenomenon is shown in Figure 27. In the latter λ is plotted versus x. The labels of the plot are added manually .

```
x1=x2(1,s2(2).index);
p([1 2])=x2(end-1:end,s2(2).index);
[x0,v0]=init_BP_BP(@cstr,x1,p,[1 2 3],1);
opt=contset(opt,'Backward',1);
[x3,v3,s3,h3,f3]=cont(@branchpoint,x0,[],opt);
first point found
tangent vector to first point found

elapsed time  = 0.8 secs
npoints curve = 300
```

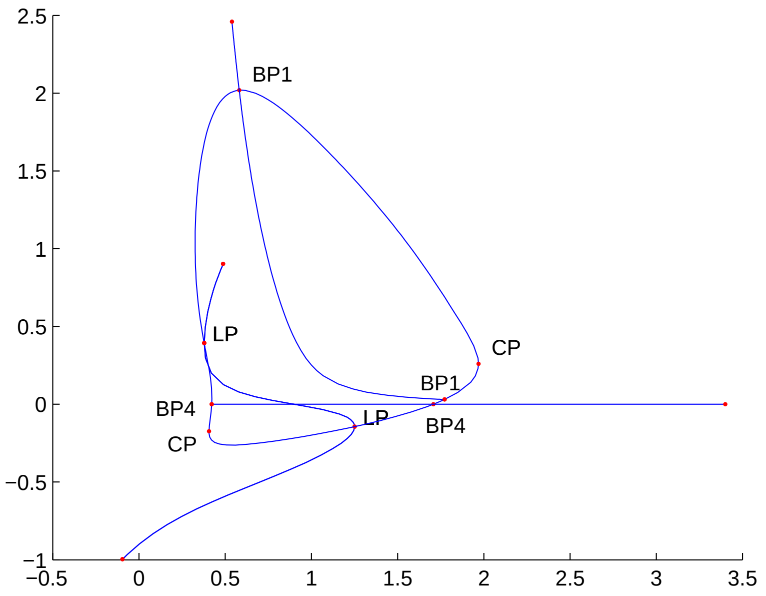


Figure 27: Computed BP curves started from Branch Points detected on a fold curve.

```
hold on;
cpl(x3,v3,s3,[2,1]);
x1=x2(1,s2(5).index);
p([1 2])=x2(end-1:end,s2(5).index);
[x0,v0]=init_BP_BP(@cstr,x1,p,[1 2 3],4);
opt=contset(opt,'Backward',1);
[x3,v3,s3,h3,f3]=cont(@branchpoint,x0,[],opt);
first point found
tangent vector to first point found

elapsed time  = 0.8 secs
npoints curve = 300
hold on;
cpl(x3,v3,s3,[2,1]);
```

These computations can be done by running the script cstr3.

9.2 Branch Point of Cycles Continuation

9.2.1 Mathematical Definition

A BPC can be characterized by adding two extra constraints $G_1 = 0$ and $G_2 = 0$ to (47) where G_1 and G_2 are the Branch Point test functions. The complete BVP defining a BPC point using the minimal extended system is

$$\begin{cases} \frac{dx}{dt} - Tf(x,\alpha) & = 0 \\ x(0) - x(1) & = 0 \\ \int_0^1 \langle x(t), \dot{x}_{old}(t) \rangle dt & = 0 \\ G[x,T,\alpha] & = 0 \end{cases} \tag{84}$$

where

$$G = \begin{pmatrix} G_1 \\ G_2 \end{pmatrix}$$

is defined by requiring

$$N \begin{pmatrix} v_1 & v_2 \\ G_1 & G_2 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{85}$$

Here v_1 and v_2 are functions, G_1 and G_2 are scalars and

$$N = \begin{bmatrix} D - Tf_x(x(\cdot),\alpha) & -f(x(\cdot),\alpha) & -Tf_\beta(x(\cdot),\alpha) & w_{01} \\ \delta_1 - \delta_0 & 0 & 0 & w_{02} \\ Int_{\dot{x}_{old}(\cdot)} & 0 & 0 & w_{03} \\ v_{11} & v_{12} & v_{13} & 0 \\ v_{21} & v_{22} & v_{23} & 0 \end{bmatrix} \tag{86}$$

where the bordering operators v_{11}, v_{21}, function w_{01}, vector w_{02} and scalars $v_{12}, v_{22}, v_{13}, v_{23}$ and w_{03} are chosen so that N is nonsingular [15][16].

9.2.2 Bifurcations

In the current version no bifurcations are detected.

9.2.3 Branch Point of Cycles initialization

The only way to start a continuation of branch points supported in the current version is to start it from a BPC detected on an limitcycle curve or on an LPC curve. This can be done using the following statement: `[x0,v0]=init_BPC_BPC(@odefile, x, s, ap, ntst, ncol, bp)`. This routine stores its information in a global stucture `lds`. The result of `init_BP_BP` contains a vector `x0` with the state variables and the three active parameters and a vector `v0` that is empty. Here `odefile` is the ode-file to be used, `x` is a vector of state variables containing the values of the state variables returned by a previous limit cycle curve or fold of cycles curve continuation, `s` is a structure containing the BPC point values returned by a previous limit cycle curve or fold of cycles curve continuation, `ap` is the vector containing the indices of the 3 active parameters and `bp` is the index of the branch parameter. `ntst` and `ncol` are again the number of mesh and collocation points for the discretization.

9.2.4 Example

In this section we discuss a non-generic situation, i.e. a case with a symmetry and a continuation of BPC points that involves two effective parameters and one artificial parameter.

For this example the following model is used:

$$\left\{\begin{array}{lcl} \dot{x} & = & (-(\beta+\nu)x+\beta y-a_3x^3+b_3(y-x)^3)/r \\ \dot{y} & = & \beta x-(\beta+\gamma)y-z-b_3(y-x)^3 \\ \dot{z} & =y & \end{array}\right. \tag{87}$$

which is the same system as in the torus of cycles continuation. It has a trivial solution branch $x = y = z = 0$ for all parameter values. Moreover, it has the Z_2 - symmetry $x \to -x$,$y \to -y$. To compute the branch of BPC points with respect to ν through the BPC point that we will detect on a limitcycle continued with free parameters ν, β, we need to introduce an additional free parameter that breaks the symmetry. There are many choices for this; we choose to introduce a parameter ϵ and extend the system (87) by simply adding a term $+\epsilon$ to the first right-hand-side. For $\epsilon = 0$ this reduces to (87) while for $\epsilon \neq 0$ the symmetry is broken.

We start by computing the trivial branch with fixed parameters $\gamma = -0.6$, $r = 0.6$, $a_3 = 0.328578$, $b_3 = 0.933578$, $\beta = 0.5$,$\epsilon = 0$ and free parameter ν with initially $\nu = -0.9$. On this branch a Hopf point is detected for $\nu = -0.58933644$ and a branch point of equilibria for $\nu = -0.5$.

```
>> p=[0.5;-0.6;0.6;0.32858;0.93358;-0.9;0];
>> [x0,v0]=init_EP_EP(@torBPC,[0;0;0],p,[6]);
>> opt=contset; opt= contset(opt,'Singularities',1);
>> opt=contset(opt,'MaxNumPoints',50);
>> [x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
first point found
tangent vector to first point found
label = H , x = ( 0.000000 0.000000 0.000000 -0.589336 )
First Lyapunov coefficient = -4.563631e-001
label = BP, x = ( 0.000000 0.000000 0.000000 -0.500000 )

elapsed time  = 0.3 secs
npoints curve = 50
```

These computations can be done by running the script testtorBPC4.

From the Hopf point we start the computation of a curve of limit cycles, using 25 test intervals and 4 collocation points. This is clearly a branch of symmetric solutions of (87); we detect one LPC and two BPC, see Fig. 28.

```
>> x1=x(1:3,s(2).index);p(6)= x(end,s(2).index);ap = 6;
>> [x0,v0]=init_H_LC(@torBPC,x1,p,ap,0.0001,25,4);
>> opt=contset; opt= contset(opt,'Singularities',1);
>> opt=contset(opt,'Multipliers',1);
>> opt=contset(opt,'MaxNumPoints',150);
>> opt=contset(opt,'Adapt',5);
>> [xlc,vlc,slc,hlc,flc]=cont(@limitcycle,x0,v0,opt);
first point found
```

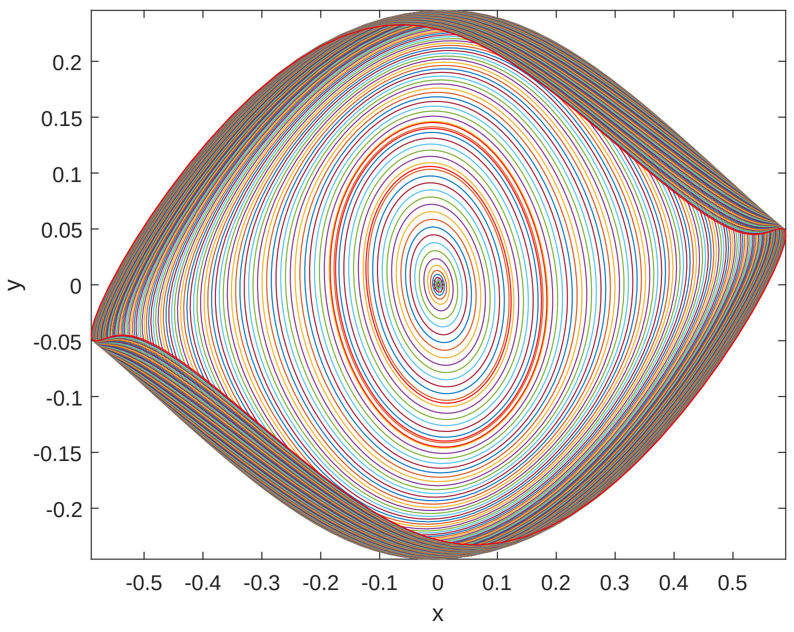


Figure 28: Curve of limit cycles with LPC and branch points in the circuit example.

```
tangent vector to first point found
Limit point cycle (period = 8.426472e+000, parameter = -5.843348e-001)
Normal form coefficient = 1.553595e-001
Branch Point cycle(period = 8.689669e+000, parameter = -5.870290e-001)
Neimark-Sacker (period = 8.743033e+000, parameter = -5.881194e-001)
Neutral saddle

elapsed time  = 30.2 secs
npoints curve = 150
>> plotcycle(xlc,vlc,slc,[1 2]);
```

These computations can be done by running the script testtorBPC5.

We continue the secondary cycle branch passing through the BPC point. From Fig. 29 it is clear that in the secondary cycle the symmetry is broken.

```
>> [x1,v1]=init_BPC_LC(@torBPC,xlc,vlc,slc(3),25,4,1e-6);
>> opt=contset(opt,'MaxNumPoints',50);
>> opt=contset(opt,'Backward',1);
>> [xlc1,vlc1,slc1,hlc1,flc1]=cont(@limitcycle,x1,v1,opt);
first point found
tangent vector to first point found
Neimark-Sacker (period = 8.794152e+000, parameter = -5.916502e-001)
Normal form coefficient = -8.661266e-003
Period Doubling (period = 9.266303e+000, parameter = -6.149552e-001)
Normal form coefficient = -6.374237e-003
```

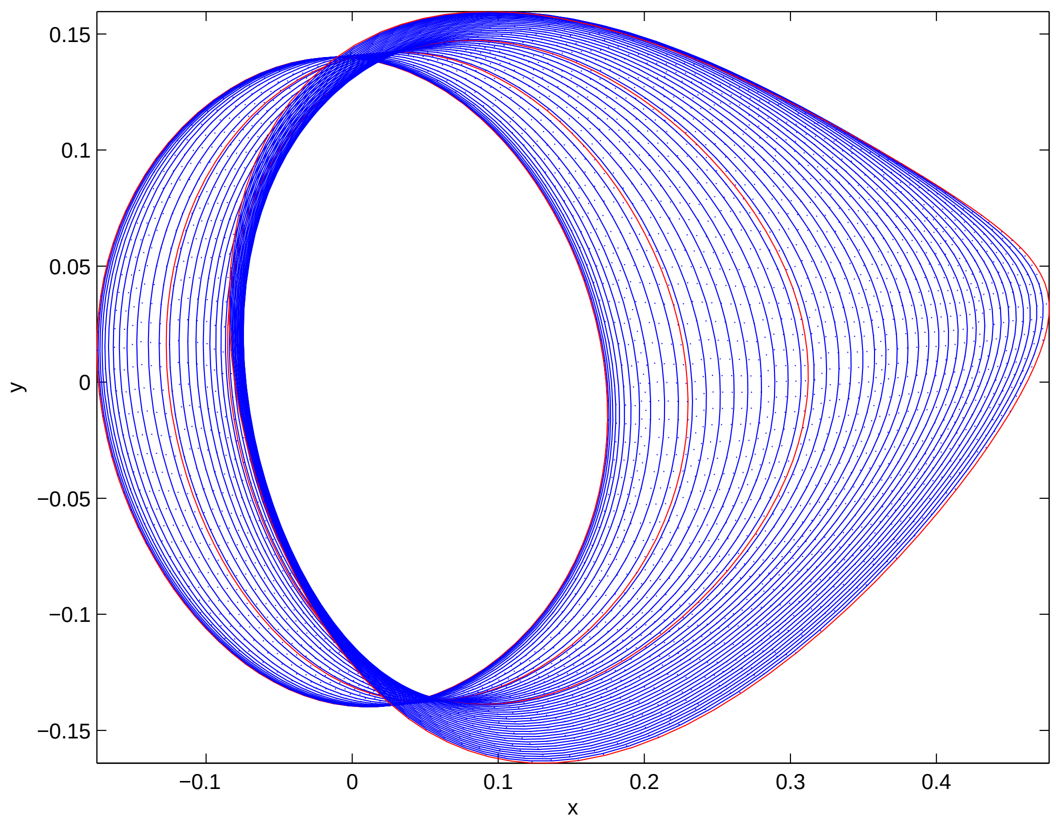


Figure 29: Asymmetric curve of limit cycles in the circuit example.

```
elapsed time  = 13.6 secs
npoints curve = 50
>> plotcycle(xlc1,vlc1,slc1,[1 2]);
```

These computations can be done by running the script testtorBPC6.

Using the code for the continuation of generic BPC points with three free parameters ν, β, ϵ we continue the curve of non-generic BPC points, where ϵ remains close to zero. The picture in Fig. 30 clearly shows that the symmetry is preserved (the axis labels were added manually).

```
>> [x1,v1]=init_BPC_BPC(@torBPC,xlc,slc(3),[1 6 7],25,4,ap);
>> opt=contset(opt,'MaxNumPoints',200);
>> [xbpc,vbpc,sbpc,hbpc,fbpc]=cont(@branchpointcycle,x1,v1,opt);
first point found
tangent vector to first point found

elapsed time  = 158.1 secs
npoints curve = 200
>> plotcycle(xbpc,vbpc,sbpc,[1 2]);
```

These computations can be done by running the script testtorBPC7.

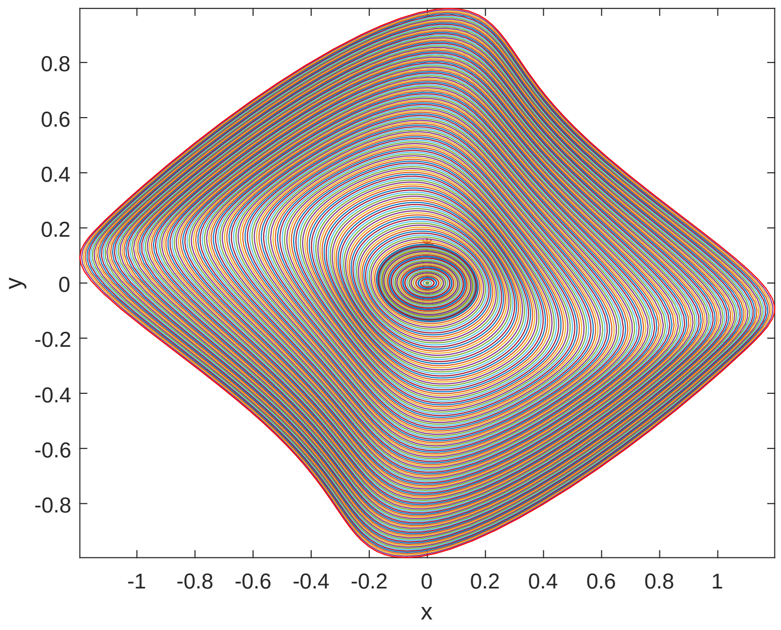


Figure 30: Curve of BPC points in the circuit example.

10 Continuation of homoclinic and heteroclinic orbits

10.1 Homoclinic orbits: Mathematical definition

In dynamical systems theory, an orbit corresponding to a solution $u(t)$ is called homoclinic to the equilibrium point u^0 of the dynamical system if $u(t) \to u^0$ as $t \to \pm\infty$. There are two types of homoclinic orbits with codimension 1, namely homoclinic-to-hyperbolic-saddle (HHS), whereby u^0 is a saddle, saddle-focus or bi-focus, and homoclinic-to-saddle-node (HSN), whereby u^0 is a saddle-node (i.e., exhibits a limit point bifurcation). We recall that AUTO has a toolbox for homoclinic continuation, named HomCont [4], [5].

During the continuation, it is necessary to keep track of several eigenspaces of the equilibrium in each step. To do this in an efficient way, MatCont incorporates the continuation of invariant subspaces [7] into the defining system.

10.1.1 Homoclinic-to-Hyperbolic-Saddle Orbits

To continue HHS orbits in two free parameters, we use an extended defining system that consists of several parts.

First, the infinite time interval is truncated, so that instead of $[-\infty, +\infty]$ we use $[-T, +T]$, which is scaled to $[0, 1]$ and divided into mesh-intervals. The mesh is nonuniform and adaptive. Each mesh interval is further subdivided by equidistant fine mesh points. Also, each mesh interval contains a number of collocation points. (This discretization is the same as that in AUTO for boundary value problems.) The equation

$$\dot{x}(t) - 2Tf(x(t), \alpha) = 0, \tag{88}$$

must be satisfied in each collocation point.

The second part is the equilibrium condition

$$f(x_0, \alpha) = 0. \tag{89}$$

Third, there is a so-called phase condition needed for the homoclinic solution, similar to periodic solutions

$$\int_0^1 \dot{\widetilde{x}}^*(t)[x(t) - \widetilde{x}(t)]dt = 0. \tag{90}$$

Here $\widetilde{x}(t)$ is some initial guess for the solution, typically obtained from the previous continuation step. We note that in the literature another phase condition is also used, see, for example [13]. However, in the present implementation we employ the condition (90).

Fourth, there are the homoclinic-specific constraints to the solution. For these we need access to the stable and unstable eigenspaces of the system in the equilibrium point after each step. It is not efficient to recompute the spaces from scratch in each continuation-step. Instead, we use the algorithm for continuing invariant subspaces, as described in [7]. This method adds two small-sized vectors (Y_S and Y_U) to the system variables, from which the necessary eigenspaces (stable and unstable, respectively) can easily be computed in each step.

If $Q(0)$ is an orthogonal matrix whose first m columns form a basis for the invariant subspace under consideration in the previous step, and $A = f_x(x_0, \alpha)$ is the Jacobian at the new equilibrium point, then we first compute the so-called Ricatti-blocks, T_{ij}, by the formula

$$\left[\begin{array}{cc} T_{11} & T_{12} \\ T_{21} & T_{22} \end{array} \right] = Q(0)^T A\, Q(0). \tag{91}$$

If n is the number of state variables, then T_{11} is of size $m \times m$ and T_{22} is $(n-m) \times (n-m)$. This is done for the stable and unstable eigenspaces separately. Now Y_S and Y_U are obtained from the Ricatti equations

$$\begin{array}{lllllllll} T_{22U} Y_U & - & Y_U T_{11U} & + & T_{21U} & - & Y_U T_{12U} Y_U & = & 0, \\ T_{22S} Y_S & - & Y_S T_{11S} & + & T_{21S} & - & Y_S T_{12S} Y_S & = & 0. \end{array} \tag{92}$$

Now we can formulate constraints on the behavior of the solution close to the equilibrium x_0. The initial vector of the orbit, $(x(0) - x_0)$, is placed in the unstable eigenspace of the system in the equilibrium. We express that by the requirement that it is orthogonal to the orthogonal complement of the unstable eigenspace. Using Y_U, we can compute the orthogonal complement of the unstable eigenspace. If $Q_U(0)$ is the orthogonal matrix from the previous step, related to the unstable invariant subspace, then a basis for the orthogonal complement in the new step $Q_U^{\perp}(s)$ is

$$Q_U^{\perp}(s) = Q_U(0) \left[\begin{array}{c} -Y_U^T \\ I \end{array} \right].$$

Note that $Q_U^{\perp}(s)$ is not orthogonal. The full orthogonal matrix Q_{1U} needed for the next step, is computed separately after each step. The equations to be added to the system are (after analogous preparatory computations for the stable eigenspace)

$$\begin{array}{l} Q_U^{\perp}(s)^T (x(0) - x_0) = 0, \\ Q_S^{\perp}(s)^T (x(1) - x_0) = 0. \end{array} \tag{93}$$

Finally, the distances between $x(0)$ (resp., $x(1)$) and x_0 must be small enough, so that

$$\begin{array}{l} \|x(0) - x_0\| - \epsilon_0 = 0, \\ \|x(1) - x_0\| - \epsilon_1 = 0. \end{array} \tag{94}$$

A system consisting of all equations (88), (89), (90), (92), (93) and (94), is overdetermined. The basic defining system for the continuation of a HHS orbit in two free parameters consists of (88), (89), (92), (93), and (94) with fixed $\epsilon_{0,1}$, so that the phase condition (90) is not used. The variables in this system are stored in one vector. It contains the values of $x(t)$ in the fine mesh points including $x(0)$ and $x(1)$, the truncation time T, two free system parameters, the coordinates of the saddle x_0, and the elements of the matrices Y_S and Y_U. Alternatively, the phase condition (90) can be added if T is kept fixed but ϵ_0 and ϵ_1 are allowed to vary. It is also possible to fix T and ϵ_0, say, and allow ϵ_1 to vary, again with no phase condition. Other combinations are also possible, in particular, when the homotopy method [7] is used to compute a starting homoclinic solution. For more details on the implementation of the homoclinic continuation we refer to [19].

10.1.2 Homoclinic-to-Saddle-Node Orbits

For a homoclinic orbit to a saddle-node equilibrium, the extended defining system undergoes some small changes. Now $(x(0) - x_0)$ has to be placed in the center-unstable subspace. Analogously, $(x(1) - x_0)$ must be in the center-stable subspace. This again is implemented by requiring that the vector is orthogonal to the orthogonal complement of the corresponding space. So the equations (93) themselves do not really change; the changes happen in the computation of the matrices Q. The defining system now has one equation less than in the HHS case ($n_s + n_u < n$, with n_s the dimension of the stable, and n_u of the unstable eigenspace); the number of equations is restored however, by adding the constraint that the equilibrium must be a saddle-node. For this we use the bordering technique, as described in section 4.2.1 of [22].

10.2 Bifurcations

During HSN continuation, only one bifurcation is tested for, namely the non-central homoclinic-to-saddle-node orbit or NCH. This orbit forms the transition between HHS and HSN curves. The strategy used for detection is taken from HomCont [5].

During HHS continuation, all bifurcations detected in HomCont are also detected in our implementation. For this, mostly test functions from [5] are used.

Suppose that the eigenvalues of $f_x(x_0, \alpha_0)$ can be ordered according to

$$\Re(\mu_{ns}) \leq ... \leq \Re(\mu_1) < 0 < \Re(\lambda_1) \leq ... \leq \Re(\lambda_{nu}), \tag{95}$$

where $\Re()$ stands for 'real part of', ns is the number of stable, and nu the number of unstable eigenvalues. The test functions for the bifurcations are

- Neutral saddle, saddle-focus or bi-focus

  $$\psi = \Re(\mu_1) + \Re(\lambda_1)$$

  If both μ_1 and λ_1 are real, then it is a neutral saddle, if one is real and one consists of a pair of complex conjugates, the bifurcation is a saddle-focus, and it is a bi-focus when both eigenvalues consist of a pair of complex conjugates.

- Double real stable leading eigenvalue

  $$\psi = \begin{cases} (\Re(\mu_1) - \Re(\mu_2))^2, & \Im(\mu_1) = 0 \\ -(\Im(\mu_1) - \Im(\mu_2))^2, & \Im(\mu_1) \neq 0 \end{cases}$$

- Double real unstable leading eigenvalue

  $$\psi = \begin{cases} (\Re(\lambda_1) - \Re(\lambda_2))^2, & \Im(\lambda_1) = 0 \\ -(\Im(\lambda_1) - \Im(\lambda_2))^2, & \Im(\lambda_1) \neq 0 \end{cases}$$

- Neutrally-divergent saddle-focus (stable)

  $$\psi = \Re(\mu_1) + \Re(\mu_2) + \Re(\lambda_1)$$

- Neutrally-divergent saddle-focus (unstable)

$$\psi = \Re(\mu_1) + \Re(\lambda_2) + \Re(\lambda_1)$$

- Three leading eigenvalues (stable)

$$\psi = \Re(\mu_1) - \Re(\mu_3)$$

- Three leading eigenvalues (unstable)

$$\psi = \Re(\lambda_1) + \Re(\lambda_3)$$

- Non-central homoclinic-to-saddle-node

$$\psi = \Re(\mu_1)$$

- Shil'nikov-Hopf

$$\psi = \Re(\lambda_1)$$

- Bogdanov-Takens point

$$\psi = \begin{cases} \Re(\mu_1) \\ \Re(\lambda_1) \end{cases}$$

For orbit- and inclination-flip bifurcations, we assume the same ordering of the eigenvalues of $f_x(x_0, \alpha_0) = A(x_0, \alpha_0)$ as shown in (95), but also that the leading eigenvalues μ_1 and λ_1 are unique and real:

$$\Re(\mu_{ns}) \leq ... \leq \Re(\mu_2) < \mu_1 < 0 < \lambda_1 < \Re(\lambda_2) \leq ... \leq \Re(\lambda_{nu}) \ .$$

Then it is possible to choose normalised eigenvectors p_1^s and p_1^u of $A^T(x_0, \alpha_0)$ and q_1^s and q_1^u of $A(x_0, \alpha_0)$ depending smoothly on (x_0, α_0), which satisfy

$$\begin{aligned} A^T(x_0, \alpha_0) \ p_1^s &= \mu_1 \ p_1^s \qquad & A^T(x_0, \alpha_0) \ p_1^u &= \lambda_1 \ p_1^u \\ A(x_0, \alpha_0) \ q_1^s &= \mu_1 \ q_1^s \qquad & A(x_0, \alpha_0) \ q_1^u &= \lambda_1 \ q_1^u \ . \end{aligned}$$

The test functions for the orbit-flip bifurcations are then:

- Orbit-flip with respect to the stable manifold

$$\psi = e^{-\mu_1 T} < p_1^s, x(1) - x_0 >$$

- Orbit-flip with respect to the unstable manifold

$$\psi = e^{\lambda_1 T} < p_1^u, x(0) - x_0 >$$

For the inclination-flip bifurcations, in [5] the following test functions are introduced:

- Inclination-flip with respect to the stable manifold

$$\psi = e^{-\mu_1 T} < q_1^s, \phi(0) >$$

- Inclination-flip with respect to the unstable manifold

$$\psi = e^{\lambda_1 T} < q_1^u, \phi(1) >$$

where ϕ ($\phi \in \mathcal{C}^1([0,1], \mathbb{R}^n)$) is the solution to the adjoint system, which can be written as

$$\begin{cases} \dot{\phi}(t) + 2\,T\,A^T(x(t), \alpha_0)\,\phi(t) = 0 \\ (L_s)^T \phi(1) = 0 \\ (L_u)^T \phi(0) = 0 \\ \int_0^1 \widetilde{\phi}^T(t)[\phi(t) - \widetilde{\phi}(t)]dt = 0 \end{cases} \tag{96}$$

where L_s and L_u are matrices whose columns form bases for the stable and unstable eigenspaces of $A(x_0, \alpha_0)$, respectively, and the last condition selects one solution out of the family $c\phi(t)$ for $c \in \mathbb{R}$. L_u is equivalent to Q_U from the mathematical definition of the system, and L_s to Q_S. In the homoclinic defining system the orthogonal complements of Q_S and Q_U are used; in the adjoint system for the inclination-flip bifurcation, we use the matrices themselves (or at least, their transposed versions).

10.3 Homoclinic initialization (HHS)

For homoclinics the same problems occur as with limit cycles, a homoclinic continuation cannot be done by just calling the continuer as

```
[x,v,s,h,f]=cont(@homoclinic, x0, v0, opt)
```

The homoclinic curve file has to know

- which ode file to use,
- which (two) system parameters are active,
- the values of all parameters,
- the number of mesh and collocation points to use for the discretization.

Also, some initial information about the state variables is necessary. In the simplest cases this is an initial cycle `x0` with a large period and close to a homoclic orbit, or an already known homoclinic orbit. This information can be supplied using an initializer:

- `[x0,v0]=init_LC_Hom(@odefile, x, s, p, ap, ntst, ncol, extravec, T, eps0, eps1)`
  Calculates an initial homoclinic orbit from a limit cycle with large period. Here `odefile` is the ode-file to be used. `x` and `s` are here the `x` and `s` belonging to the limit cycle with large period, obtained in a previous continuation. `p` is the vector containing the current values of the parameters. `ap` is the active parameter and `ntst` and `ncol` are the number of mesh and collocation points to be used for the discretization. `extravec` is a vector of 3 integers, which are either `0` or `1`, and which indicate which of `T, eps0, eps1` are to be variable during the continuation. The vector is `1` for those that should be variable. This can either be 1 or 2 or the three parameters. `T, eps0` and `eps1` are values for these parameters.

- `[x0,v0]=init_Hom_Hom(@odefile, x, s, p, ap, ntst, ncol, extravec, T, eps0, eps1)`
  Calculates an initial homoclinic orbit from a homoclinic orbit obtained during a previous continuation. All parameters are similar to the initialisations above.

These initializers return an initial homoclinic orbit `x0` as well as its tangent vector `v0`.

It is also possible to start a homoclinic orbit from a Bogdanov-Takens equilibrium point. The initialization of homoclinic orbits from a BT point has a long history. We refer in particular to [30], [31] and [1] where further references can be found. This case is relegated to the MatCont GUI tutorials.

The homotopy method even allows a start from any equilibrium point. For details and examples we refer to [8]. This case is also relegated to the MatCont GUI tutorials.

10.4 Homoclinic-to-Saddle-Node initialization (HSN)

For homoclinic-to-saddle-node orbits, the user needs to replace `Hom` with `HSN` in the above two initializers. The homotopy method allows a start from any limit point. For details and examples we refer to [8]. This case is relegated to the MatCont GUI tutorials.

10.5 CL_MatCont: the MLFast example

In §8.4.4 we studied a continuation of limit cycles in the `MLfast` Morris-Lecar model and noted that the limit cycles were approaching a homoclinic orbit. We will now approach this homoclinic even closer, and then start its continuation from the large limit cycle. The result is shown in Figure 31.

```
>> init;
>> p=[0.11047;0.1];ap1=[1];
>> [x0,v0]=init_EP_EP(@MLfast,[0.047222;0.32564],p,ap1);
>> opt=contset;opt=contset(opt,'Singularities',1);
>> opt=contset(opt,'MaxNumPoints',65);
>> opt=contset(opt,'MinStepSize',0.00001);
>> opt=contset(opt,'MaxStepSize',0.01);
>> opt=contset(opt,'Backward',1);
>> [x,v,s,h,f]=cont(@equilibrium,x0,[],opt);
first point found
tangent vector to first point found
label = H , x = ( 0.036756 0.294770 0.075659 )
First Lyapunov coefficient = 8.234573e+000
label = LP, x = ( -0.033738 0.136501 -0.020727 )
a=-1.036706e+001
label = H , x = ( -0.119894 0.045956 0.033207 )
Neutral saddle
label = LP, x = ( -0.244915 0.008514 0.083257 )
a=2.697414e+000

elapsed time  = 0.4 secs
npoints curve = 65
```

```
>> x1=x(1:2,s(2).index);p=[x(end,s(2).index);0.1];
>> [x0,v0]=init_H_LC(@MLfast,x1,p,ap1,0.0001,30,4);
>> opt=contset;
>> opt=contset(opt,'MaxStepSize',1);
>> opt=contset(opt,'IgnoreSingularity',1);
>> opt=contset(opt,'Singularities',1);
>> opt=contset(opt,'MaxNumPoints',200);
>> [x2,v2,s2,h2,f2]=cont(@limitcycle,x0,v0,opt);
first point found
tangent vector to first point found
Limit point cycle (period = 4.222011e+000, parameter = 8.456948e-002)
Normal form coefficient = -2.334576e-001
Limit point cycle (period = 5.653399e+001, parameter = 7.293070e-002)
Normal form coefficient = 1.132235e+000
Limit point cycle (period = 5.739877e+001, parameter = 7.293070e-002)
Normal form coefficient = 3.266287e+000
Limit point cycle (period = 8.938964e+001, parameter = 7.293071e-002)
Normal form coefficient = -1.537206e-001

elapsed time  = 86.6 secs
npoints curve = 200
>> p(ap1) = x2(end,end);
>> T = x2(end-1,end)/2;
>> [x0,v0]=init_LC_Hom(@MLfast, x2(:,end), s2(:,end), p, [1 2], 40, 4,...
>>          [0 1 1], T, 0.01, 0.01);
>> opt=contset(opt,'MaxNumPoints',15);
>> [xh,vh,sh,hh,fh] = cont(@homoclinic,x0,v0,opt);
first point found
tangent vector to first point found
elapsed time  = 4.4 secs
npoints curve = 15
>> plotcycle(xh,vh,sh,[1 2]);
```

The above computations can be done by running the script `homoc1`. The picture is presented in Figure 31. Similar tests can be done by using the testrun `testmyml`.

10.6 Heteroclinic orbits (Het)

An orbit corresponding to a solution $u(t)$ is called heteroclinic if there exist equilibria $u_0, u_1, u_0 \neq u_1$ so that $u(t) \to u_0$ as $t \to -\infty$ and $u(t) \to u_1$ as $t \to \infty$.

Details on the continuation of heteroclinic orbits can be found in [19] and [8]. The routines for dealing with heteroclinic orbits are in the directory `Heteroclinic`.

The directories `HomotopyHet`, `HomotopySaddle` and `HomotopySaddleNode` are initialization directories in which homotopy methods are provided to initialize heteroclinic orbits, orbits homoclinic to saddle and orbits homoclinicto saddle-node, respectively. Eamples of their use are relegated to the MatCont GUI tutorials.

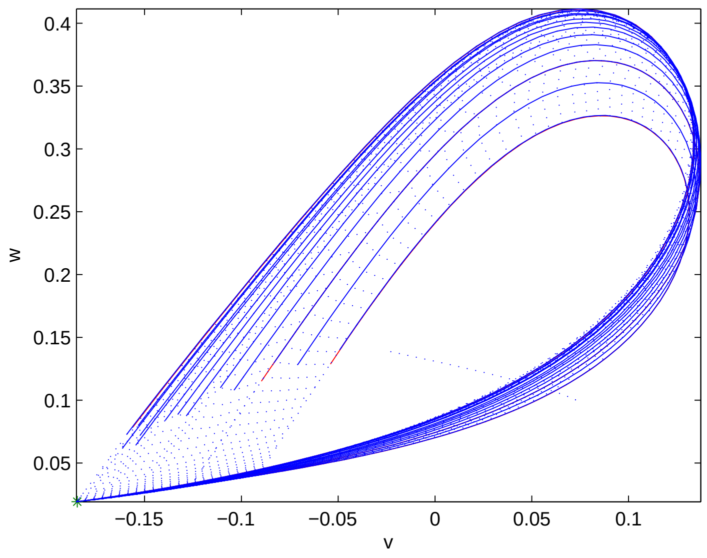


Figure 31: Computed curve of homoclinic-to-saddle orbits started from a limit cycle with large period.

A Continuer example: a curve object

In this section a simple example is presented to illustrate the basic use of the continuer. This example generates a curve g in the (x, y)-plane such that $x^2 + y^2 = 1$. So if the user specifies a point reasonably close to this curve we get the unit circle. The defining function is

$$F(x, y) = x^2 + y^2 - 1 \tag{97}$$

In the following listing this curve is implemented. Do note that while there are no options needed, the curve file *must* return an option structure (see section 3.3).

curve.m

```
function out = curve
%
% Curve file of circle
%

 out{1}  = @curve_func;
 out{2}  = @defaultprocessor;
 out{3}  = @options;
 out{4}  = []; %@jacobian;
 out{5}  = []; %@hessians;
 out{6}  = []; %@testf;
 out{7}  = []; %@userf;
 out{8}  = []; %@process;
 out{9}  = []; %@singmat;
 out{10} = []; %@locate;
 out{11} = []; %@init;
 out{12} = []; %@done;
 out{13} = @adapt;
function f = curve_func(arg)
  x = arg;
  f = x(1)^2+x(2)^2-1;

function varargout = defaultprocessor(varargin)
  if nargin > 2
    s = varargin{3};
    varargout{3} = s;
  end
  % no special data
  varargout{2} = [];
  % all done succesfully
  varargout{1} = 0;

function option = options
  option = contset;

function [res,x,v] = adapt(x,v)
  res=[];

```

curve.m

The file `curve.m` is stored in the directory `Testruns/TestSystems`. Starting computations

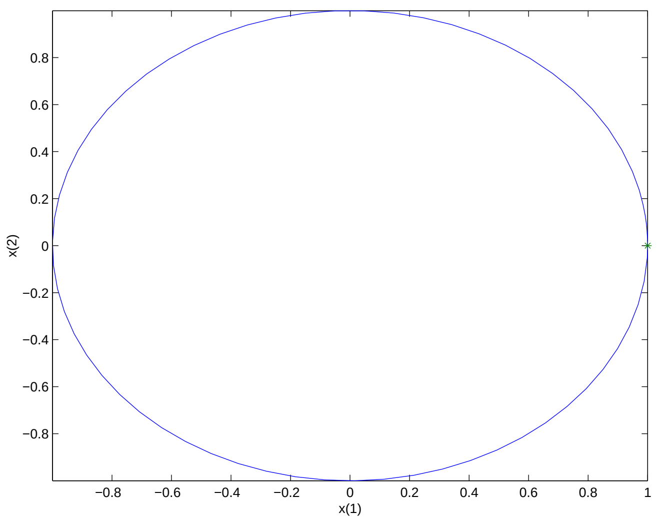


Figure 32: Computed curve of `curve.m`

at $(x, y) = (1, 0)$, the output in MATLAB looks like:

```
>> init;
>> [x,v,s]=cont(@curve,[1;0]);
first point found
tangent vector to first point found
Closed curve detected at step 70

elapsed time  = 0.1 secs
npoints curve = 70
```

The generated curve is plotted in Figure 32 with the command:

```
>> cpl(x,v,s)
```

In this case `x` has dimension 2, so a 2D-plot is drawn with the first component of `x` (value of the state variable x) on the x-axis and the second component of `x` (value of the state variable y) on the y-axis. The above commands are also executed by running `testdrawcurve`; the file `testdrawcurve.m` is in the directory `Testruns`.

B The Brusselator example: Continuation of a solution to a boundary value problem in a free parameter

Discretized solutions of PDEs can also be continued in CL_MatCont. We illustrate this by continuing the equilibrium solution to a one-dimensional PDE. The curve type is called 'pde_1'. It is defined in the file `pdf_1.m` in the directory `Equilibrium`.

The Brusselator is a system of equations intended to model the Belusov-Zhabotinsky reaction. This is a system of reaction-diffusion equations that is known to exhibit oscillatory behavior. The unknowns are the concentrations $X(x,t), Y(x,t), A(x,t)$ and $B(x,t)$ of four reactants. Here t denotes time and x is a one-dimensional space variable normalized so that $x \in [0,1]$. The length L of the reactor is a parameter of the problem. In our simplified setting A and B are constants.

The system is described by two partial differential equations:

$$\begin{array}{rcl} \frac{\partial X}{\partial t} & = & \frac{D_x}{L^2}\frac{\partial^2 X}{\partial x^2} + A - (B-1)X + X^2Y \\ \frac{\partial Y}{\partial t} & = & \frac{D_y}{L^2}\frac{\partial^2 Y}{\partial x^2} + BX - X^2Y \end{array} \tag{98}$$

with $x \in [0,1],\ t \geq 0$. Here D_x, D_y are the diffusion coefficients of X and Y. At the boundaries $x = 0$ and $x = 1$ Dirichlet conditions are imposed:

$$\begin{cases} X(0,t) = X(1,t) = A \\ Y(0,t) = Y(1,t) = \frac{B}{A} \end{cases} \tag{99}$$

We are interested in equilibrium solutions $X(x)$ and $Y(x)$ to the system and their dependence on the parameter L.

The approximate equilibrium solution is:

$$\begin{cases} X(x) & = & A + 2\sin(\pi x) \\ Y(x) & = & \frac{B}{A} - \frac{1}{2}\sin(\pi x) \end{cases} \tag{100}$$

The initial values of the parameters are: $A = 2$, $B = 4.6$, $D_x = 0.0016$, $D_y = 0.08$ and $L = 0.06$. The initial solution (100) is not an equilibrium, but the continuer will try to converge to an equilibrium close to the initial solution. We use equidistant meshes. To avoid spurious solutions (solutions that are induced by the discretization but do not actually correspond to solutions of the undiscretized problem) one can vary the number of mesh points by setting the parameter N. If the same solution is found for several discretizations, then we can assume that they correspond to solutions of the continuous problem.

The second order space derivative is approximated using the well-known three-points difference formula: $\frac{\partial^2 f}{\partial x^2} = \frac{1}{h^2}(f_{i-1} - 2f_i + f_{i+1})$, where $h = \frac{1}{N+1}$, where N is the number of grid points on which we discretize X and Y. So N is a parameter of the problem and $2N$ is the number of state variables (which is not fixed in this case).

The Jacobian is a sparse 5-band matrix. In the ode-file describing the problem the Jacobian is introduced as a sparse matrix. The Hessian is never computed as such but second order derivatives are computed by finite differences whenever needed. We note that MATLAB 6.5 or 7 does not provide sparse structures for 3-dimensional arrays.

bruss.m

```
function out = bruss
%
% Odefile of 1-d Brusselator model
%

out{1} = @init;
out{2} = @fun_eval;
out{3} = @jacobian;
out{4} = @jacobianp;
out{5} = [];%@hessians;
out{6} = [];%@hessiansp;
out{7} = [];
out{8} = [];
out{9} = [];



% --------------------------------------------------------------------------


function dfdt = fun_eval(t,y,N,L)

x = y(1:N);
y = y(N+1:2*N);

A  = 2;
B  = 4.6;
Dx = 0.0016;
Dy = 0.008;
x0 = A; x1 = A;
y0 = B/A; y1 = B/A;
L2 = L^2;
h  = 1/(N+1);
cx = (Dx/L2)/(h*h);
cy = (Dy/L2)/(h*h);

dxdt = zeros(N,1);
dydt = zeros(N,1);

dxdt(1) = (x0-2*x(1)+x(2))*cx + A - (B+1)*x(1) + x(1)*x(1)*y(1);
dxdt(N) = (x(N-1)-2*x(N)+x1)*cx + A - (B+1)*x(N) + x(N)*x(N)*y(N);

dydt(1) = (y0-2*y(1)+y(2))*cy + B*x(1) - x(1)*x(1)*y(1);
dydt(N) = (y(N-1)-2*y(N)+y1)*cy + B*x(N) - x(N)*x(N)*y(N);

for i=2:N-1
  dxdt(i) = (x(i-1)-2*x(i)+x(i+1))*cx + A - (B+1)*x(i) + x(i)*x(i)*y(i);
  dydt(i) = (y(i-1)-2*y(i)+y(i+1))*cy + B*x(i) - x(i)*x(i)*y(i);
end

dfdt = [dxdt; dydt];

% --------------------------------------------------------------------------

function [tspan,y0,options] = init(N)
tspan = [0; 10];
A  = 2;
```

```
B  = 4.6;

y0 = zeros(2*N,1);

for i=1:N
  y0(i)   = A + 2*sin(pi*i/(N+1));
  y0(N+i) = B/A - 0.5*sin(pi*i/(N+1));
end
handles = feval(@bruss);
options = odeset('Vectorized','on', 'Jacobian', handles(3), 'JacobianP', handles(4));

% --------------------------------------------------------------------------

function dfdxy = jacobian(t,y,N,L)
x = y(1:N);
y = y(N+1:2*N);
A  = 2;
B  = 4.6;
Dx = 0.0016;
Dy = 0.008;
x0 = A; x1 = A;
y0 = B/A; y1 = B/A;
L2 = L^2;
h  = 1/(N+1);
cx = (Dx/L2)/(h*h);
cy = (Dy/L2)/(h*h);


%
% Sparse jacobian
%
A=zeros(2*N,3);
A(1:N-1,2)=cx;
A(1:N,3)=-2*cx -(B+1) + 2*x(1:N).*y(1:N);
A(1:N,4)=cx;

A(N+1:2*N,2) = cy;
A(N+1:2*N,3) = -2*cy -x(:).*x(:);
A(N+2:2*N,4) = cy;


A(1:N,1) = B - 2*x(:).*y(:);
A(N+1:2*N,5) = x(:).*x(:);

dfdxy = spdiags(A, [-N,-1:1,N] , 2*N, 2*N);
return

%
% Full matrix
%
dfxdx = zeros(N,N);
dfydy = zeros(N,N);
dfxdy = zeros(N,N);
dfydx = zeros(N,N);

for i=1:N
  if i>1, dfxdx(i,i-1) = cx; end;
```

```
          dfxdx(i,i)   = -2*cx -(B+1) + 2*x(i)*y(i);
  if i<N, dfxdx(i,i+1) = cx; end;
  dfxdy(i,i) = x(i)*x(i);

  if i>1, dfydy(i,i-1) = cy; end;
          dfydy(i,i)   = -2*cy -x(i)*x(i);
  if i<N, dfydy(i,i+1) = cy; end;
  dfydx(i,i) = B - 2*x(i)*y(i);
end

dfdxy = [ dfxdx, dfxdy; dfydx, dfydy ];

% ----------------------------------------------------------------------------

function dfdp = jacobianp(t,y,N,L)
x = y(1:N);
y = y(N+1:2*N);
A  = 2;
B  = 4.6;
Dx = 0.0016;
Dy = 0.008;
x0 = A; x1 = A;
y0 = B/A; y1 = B/A;
L2 = L^2;
h  = 1/(N+1);
cx = (Dx/L2)/(h*h);
cy = (Dy/L2)/(h*h);
kx = (-2/L)*cx;
ky = (-2/L)*cy;

Sx = zeros(N,1);
Sy = zeros(N,1);

Sx(1) = kx*(x0-2*x(1)+x(2));
Sy(1) = ky*(y0-2*y(1)+y(2));

Sx(N) = kx*(x(N-1)-2*x(N)+x1);
Sy(N) = ky*(y(N-1)-2*y(N)+y1);

i=2:N-1;
Sx(i)   = kx*(x(i-1)-2*x(i)+x(i+1));
Sy(i) = ky*(y(i-1)-2*y(i)+y(i+1));

dfdp = [ zeros(2*N,1) [Sx;Sy] ];

% ----------------------------------------------------------------------------
```
bruss.m

The model is implemented with 2 parameters: N and L; the values of A, B, D_x, D_y are hard-coded. Note that N is a parameter that cannot vary during the continuation. Therefore it does not have entries in Jacobianp. We should let the pde_1 curve know that *bruss.m* is the active file, the initial values of the parameters N and L are respectively 20 and 0.06 and the active parameter is L, i.e. the second parameter of `bruss.m`. So, first of all we have to get the approximate equilibrium solution which is provided in a subfunction 'init' of 'bruss.m'.

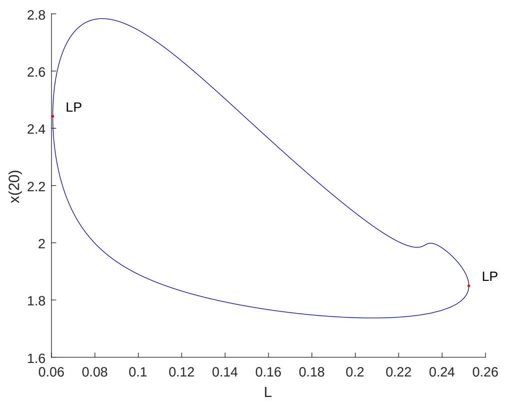


Figure 33: An quilibrium curve of `bruss.m`

We first locate the handle to the subfunction init and call it.

```
>N = 20; L = 0.06;
>handles = feval(@bruss);
>[t,x0,options] = feval(handles{1},N);
```

It sets the number of state variables to $2N$ and makes an initial vector $x0$ of length $2N$ containing the values of the approximate equilibrium solution. Now we inform the pde_1 curve that the second parameter of `bruss.m` is the active parameter and what the default values of the other parameters are. We also set some options.

```
>[x1,v1] = init_EP_EP(@bruss,x0,[N;L], [2]);
>opt = contset;opt=contset(opt,'MinStepsize', 1e-5);
>opt=contset(opt,'MaxCorrIters', 10);
>opt=contset(opt,'MaxNewtonIters', 20);
>opt=contset(opt,'FunTolerance', 1e-3);
>opt=contset(opt,'Singularities',1);
>opt=contset(opt,'MaxNumPoints',500);
>opt=contset(opt,'Locators',[]);
```

We start the continuation process by the statement
`[x,v,s,h] = cont(@pde_1,x1,v1,opt).`
In this case the number of state variables can be a parameter and the Jacobian can be sparse. Using the command

```
cpl(x,v,s,[41;20]);
```

we can plot the 20th component of x as a function of L (which is the 41th continuation variable), see Figure 33.

The above test can be executed by running `testbrusselator`; this file is in the directory `Testruns`.

References

[1] Bashir Al-Hdaibat, Willy Govaerts, Yuri A. Kuznetsov and Hil G.E. Meijer, Initialization of homoclinic solutions near Bogdanov-Takens points: Lindstedt-Poincaré compared with regular perturbation method, SIAM J. Appl. Dyn. Syst. 15(2). p.952-980 (2016).

[2] E.L. Allgower and K. Georg, Numerical Continuation Methods: An introduction, Springer-Verlag, 1990 .

[3] W.J. Beyn, A. Champneys, E. Doedel, W. Govaerts, Yu.A. Kuznetsov, and B. Sandstede, Numerical continuation and computation of normal forms. In: B. Fiedler, G. Iooss, and N. Kopell (eds.) "Handbook of Dynamical Systems : Vol 2", Elsevier 2002, pp 149 - 219.

[4] Champneys, A.R. and Kuznetsov Yu.A. 1994. Numerical detection and continuation of codimension-two homoclinic orbits. Int. J. Bifurcation Chaos, 4(4), 785-822.

[5] Champneys, A.R., Kuznetsov Yu.A. and Sandstede B. 1996. A numerical toolbox for homoclinic bifurcation analysis. Int. J. Bifurcation Chaos, 6(5), 867-887.

[6] C. De Boor and B. Swartz, Collocation at Gaussian points, SIAM Journal on Numerical Analysis 10 (1973), pp. 582-606.

[7] Demmel, J.W., Dieci, L. and Friedman, M.J. 2001. Computing connecting orbits via an improved algorithm for continuing invariant subspaces. SIAM J. Sci. Comput., 22(1), 81-94.

[8] V. De Witte, W. Govaerts, Yu. A. Kuznetsov and M. Friedman, Interactive Initialization and Continuation of Homoclinic and Heteroclinic Orbits in MATLAB, ACM Transactions on Mathematical Software. Volume 38, Issue 3, Article Number: 18, DOI: 10.1145/2168773.2168776 Published: APR 2012

[9] V. De Witte, F. Della Rossa, W.Govaerts and Yu.A. Kuznetsov, Numerical Periodic Normalization for Codim2 Bifurcations of Limit Cycles: Computational Formulas, Numerical Implementation, and Examples, SIAM J. Applied Dynamical Systems 12,2 (2013) 722-788. DOI: 10.1137/120874904

[10] A. Dhooge, W. Govaerts and Yu. A. Kuznetsov, MATCONT : A matlab package for numerical bifurcation analysis of ODEs, ACM Transactions on Mathematical Software 29(2) (2003), pp. 141-164.

[11] A. Dhooge, W.Govaerts, Yu. A. Kuznetsov, H. G. E. Meijer and B. Sautois: New features of the software MatCont for bifurcation analysis of dynamical systems, Mathematical and Computer Modelling of Dynamical Systems, Vol. 14(2), pp. 147-175, Published: 2008.
https://doi.org/10.1080/13873950701742754.

[12] E. Doedel and J Kernévez, AUTO: Software for continuation problems in ordinary differential equations with applications, California Institute of Technology, Applied Mathematics, 1986.

[13] Doedel, E.J. and Friedman, M.J.: Numerical computation of heteroclinic orbits, J. Comp. Appl. Math. 26 (1989) 155-170.

[14] E.J. Doedel, A.R. Champneys, T.F. Fairgrieve, Yu.A. Kuznetsov, B. Sandstede and X.J. Wang, auto97-00 : Continuation and Bifurcation Software for Ordinary Differential Equations (with HomCont), User's Guide, Concordia University, Montreal, Canada (1997-2000). (`http://indy.cs.concordia.ca`).

[15] Doedel, E.J., Govaerts W., Kuznetsov, Yu.A.: Computation of Periodic Solution Bifurcations in ODEs using Bordered Systems, SIAM Journal on Numerical Analysis 41,2(2003) 401-435.

[16] Doedel, E.J., Govaerts, W., Kuznetsov, Yu.A., Dhooge, A.: Numerical continuation of branch points of equilibria and periodic orbits, Int. J. Bifurcation and Chaos, 15(3) (2005), 841-860.

[17] Ermentrout, B.: Simulating, Analyzing, and Animating Dynamical Systems. Siam Publications, Philadelphia, 2002.

[18] Freire, E., Rodriguez-Luis, A., Gamero E. and Ponce, E., A case study for homoclinic chaos in an autonomous electronic circuit: A trip form Takens-Bogdanov to Hopf- Shilnikov, Physica D 62 (1993) 230–253.

[19] Friedman, M., Govaerts, W., Kuznetsov, Yu.A. and Sautois, B. 2005. Continuation of homoclinic orbits in matlab. Lecture Notes in Computer Science, 3514, 263-270.

[20] Genesio, R. and Tesi, A. Harmonic balance methods for the analysis of chaotic dynamics in nonlinear systems. Automatica 28 (1992), 531-548.

[21] Genesio, R., Tesi, A., and Villoresi, F., Models of complex dynamics in nonlinear systems. Systems Control Lett. 25 (1995), 185-192.

[22] W.J.F. Govaerts, Numerical Methods for Bifurcations of Dynamical Equilibria, SIAM, 2000.

[23] Govaerts, W. and Sautois, B.: Phase response curves, delays and synchronization in matlab. Lecture Notes in Computer Science, 3992 (2006), 391-398.

[24] Govaerts, W. and Sautois, B.: Computation of the phase response curve: a direct numerical approach. Neural Comput. 18(4) (2006), 817-847.

[25] Yu. A. Kuznetsov, Elements of Applied Bifurcation Theory, Springer-Verlag, 1998. (third edition 2004).

[26] Yu. A. Kuznetsov and V.V. Levitin, CONTENT: Integrated Environment for analysis of dynamical systems. CWI, Amsterdam 1997: `ftp://ftp.cwi.nl/pub/CONTENT`

[27] matlab, The Mathworks Inc., `http://www.mathworks.com`.

[28] Yu. A. Kuznetsov, W. Govaerts, E.J. Doedel and A. Dhooge, Numerical periodic normalization for codim 1 bifurcations of limit cycles, SIAM J. Numer. Anal. 43 (2005) 1407-1435.

[29] Yu.A. Kuznetsov, H.G.E. Meijer, W. Govaerts and B. Sautois, Switching to nonhyperbolic cycles from codim 2 bifurcations of equilibria in ODEs, Physica D 237 No. 23 (2008) 3061-3068 (ISSN 0167-2789).

[30] Kuznetsov, Yu.A., Meijer H.G.E., Al Hdaibat, B. and Govaerts, W., Improved homoclinic predictor for Bogdanov-Takens Bifurcation, International Journal of Bifurcations and Chaos, 24(4) (2014). Article Number: 1450057. DOI: 10.1142/S0218127414500576

[31] Yu.A. Kuznetsov, H.G.E. Meijer, B. Al-Hdaibat and W. Govaerts, Accurate Approximation of Homoclinic Solutions in Gray-Scott Kinetic Model. International Journal of Bifurcation and Chaos, Volume: 25(9) August 2015. Article Number: 1550125
DOI: 10.1142/S0218127415501254

[32] W. Mestrom, Continuation of limit cycles in matlab, Master Thesis, Mathematical Institute, Utrecht University, The Netherlands, 2002.

[33] Morris, C., Lecar,H., Voltage oscillations in the barnacle giant muscle fiber,Biophys J. 35 (1981) 193–213.

[34] N. Neirynck, Advances in numerical bifurcation software: MatCont. PhD thesis, Ghent University, Belgium 2019. https://biblio.ugent.be/publication/8615817.

[35] A. Riet, A Continuation Toolbox in matlab, Master Thesis, Mathematical Institute, Utrecht University, The Netherlands, 2000.

[36] C. Stéphanos, Sur une extension du calcul des substitutions linéaires, J. Math. Pures Appl. 6 (1900) 73-128.

[37] Terman, D., Chaotic spikes arising from a model of bursting in excitable membranes, Siam J. Appl. Math. 51 (1991) 1418–1450.

[38] Terman, D., The transition from bursting to continuous spiking in excitable membrane models, J. Nonlinear Sci. 2, (1992) 135–182.