Cl_matcont : A continuation toolbox in Matlab

A. Dhooge and W. Govaerts
Department of Applied Mathematics and Computer Science
University of Gent
Krijgslaan 281-S9
B-9000 Gent, Belgium
Annick.Dhooge@rug.ac.be

Yu. A. Kuznetsov, W. Mestrom and A.M.Riet
Mathematisch Instituut
Universiteit Utrecht
Boedapestlaan 6
3584 CD Utrecht, The Netherlands
kuznetsov@math.uu.nl

ABSTRACT

CL_MATCONT is a Matlab continuation package for the numerical study of a range of parameterized nonlinear problems. In the case of ODEs it allows to compute curves of equilibria, limit points, Hopf points, limit cycles and period doubling bifurcation points of limit cycles. All curves are computed by the same function that implements a prediction-correction continuation algorithm based on the Moore - Penrose matrix pseudo-inverse. The continuation of bifurcation points of equilibria and limit cycles is based on bordering methods and minimally extended systems. Hence no additional unknowns such as singular vectors and eigenvectors are used and no artificial sparsity in the systems is created.

The inherent sparsity of the discretized systems for the computation of limit cycles and their bifurcation points is exploited by using the standard Matlab sparse matrix methods.

CL_MATCONT furthermore allows to compute solution branches to underdetermined systems of nonlinear equations and parameterized boundary value problems.

Keywords

continuation, Matlab, bifurcation

1. INTRODUCTION

Numerical continuation is a well - understood subject, see e.g. [1], [2], [4], [5], [9]. The idea is as follows. Consider a smooth function $F : \mathbb{R}^{n+1} \to \mathbb{R}^n$. We want to compute a solution curve of the equation $F(x) = 0$. Numerical continuation is a technique to compute a sequence of points which approximate the desired branch. Like most continuation algorithms, CL_MATCONT implements a predictor-corrector method; for details we refer to the documentation available on the web.

However, the existing software packages such as AUTO [3], CONTENT [6] require the user to rewrite his/her models in a specific format which complicates the export of results, graphical representation etcetera.

The aim of CL_MATCONT is to provide a continuation toolbox which is compatible with the standard Matlab ODE representation of differential equations. This toolbox is developed with the following targets in mind:

- detection of singularities via test functions
- singularity-specific location code
- processing of regular and singular points
- support of adaptive meshes
- support of symbolic derivatives
- support for sparse matrices

Earlier versions of the toolbox are described in [8, 7]. The current version of the package is freely available for download at:

`http://allserv.rug.ac.be/~ajdhooge/research.html`

It requires Matlab 5.3 or 6.* to be installed on your computer. A manual of CL_MATCONT in PostScript format is also available on the web.

In the present paper we concentrate on a technical issue, namely the implementation in Matlab of the computation of bialternate matrix products. Furthermore we provide two examples of the use of cl_matcont; for more details and updates we refer to the above URL.

2. SINGULARITIES AND TEST FUNCTIONS

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

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

The singularity S will then be detected if

$$\phi(x_i)\phi(x_{i+1}) < 0 \tag{2}$$

Having found two points x_i and x_{i+1} one may want to locate the point x^* where $\phi(x)$ vanishes most accurately; 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.

In fact, a singularity may depend on two types of test functions: vanishing (i.e. having a regular zero at the bifurcation point) and non-vanishing (which must be nonzero). To represent all singularities we introduce a *singularity matrix* (as in [6]). This matrix is a compact way to describe the relation between the singularities and all test functions.

In some cases the default location algorithm can have problems to locate a bifurcation point. The default locator may have problem to reach convergence (the branching point in Section 5 is an example). Therefore we provide a possibility to define a specific location algorithm for a particular bifurcation.

3. SOFTWARE

3.1 Continuer

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 (see section 3.2).
`x0` and `v0` are respectively the initial point and the tangent vector of the initial point where the continuation starts.
`options` is a structure as described in section 3.2.3.
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:
`x` and `v` are the points and their tangent vectors of the curve. Each column in `x` and `v` corresponds to a point on the curve, while the rows are the elements.
`s` is an array with structures containing information about the found singularities.
`h` is used for output of the algorithm, currently this is a matrix with for each point a column with the following components :

- The *stepsize* used to calculate this point (zero for initial point and singular points).
- The *number of Newton iterations* is the number of locator iterations for singular points.
- The*test function values* are the values of all active test functions.

`f` can be anything depending on which curve file is used.

3.2 Curve file

The continuer uses a special m-file where the problem is specified and which is coded by the user. This file, further referred to as *curve.m*, contains the following sections (an asterisk indicates that it is a required part of the curve file):

- Problem definition (*)
- Options (*)
- Default processor (*)
- Symbolic derivatives of the problem
- Test functions
- Special processors
- Locators
- Singularity matrix
- User space
- Adaptation

3.2.1 Problem definition

The problem is coded in such a way that a call to `curve(x)` returns $F(x)$ evaluated at point x. Point x is a column vector of size n. Normally the return value must be a vector of size $n-1$. If the return value is empty (`[]`), the continuer considers this as a failure to compute $F(x)$ and tries to make a smaller prediction step.

3.2.2 Symbolic derivatives

To increase the speed and/or improve accuracy of the algorithm one can provide symbolic derivatives of $F(x)$. The option *SymDerivative* indicates to which order the derivatives are provided.

If *SymDerivative*$\geq$ 1, then a call to `curve('jacobian', x)` must return the $n-1 \times n$ Jacobian matrix evaluated at point x.

If *SymDerivative*$\geq$ 2, then a call to `curve('hessians', x)` must return a 3-dimensional $(n \times (n-1) \times (n-1))$ array H such that $H(i,j,k) = \frac{\partial^2 F_i(x)}{\partial x_j \partial x_k}$.

If *SymDerivative*$\geq$ 3, then a call to `curve('der3',x)` must return a 4-dimensional array of $\frac{\partial^3 F_i(x)}{\partial x_j \partial x_k \partial x_l}$.

As with computations of $F(x)$ empty return values of the above calls imply decreasing the step size.

3.2.3 Options

It is possible to specify various options. A call to `curve([],'options')` must return a structure created with `contset`. The command `options = contset` will initialize this structure. Options can then be set using `options = contset(options, optionname, optionvalue)`. Here `optionname` is an option from the following list.

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

MaxStepsize: the maximum stepsize (default: 0.1)

InitStepsize: the initial stepsize (default: 0.01)

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

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

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

MaxCorrIters: maximum number of correction iterations (default: 10)

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

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

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

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

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

Singularities: boolean indicating the presence of test functions and singularity matrix (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).

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

Adapt: number of points indicating when to adapt the problem while computing the curve (default: 0=do not adapt)

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

3.2.4 Summary

In the following table one can see what calls can be made to the problem file and which options are involved.

Syntax of call	What it should do
`curve(x)`	return $F(x)$
`curve('options')`	return option vector
`curve('jacobian',x)`	return Jacobian at `x` (*SymDerivative*≥ 1)
`curve('hessians',x)`	return Hessians at `x` (*SymDerivative*≥ 2)
`curve('der3',x)`	return 3th order derivatives at `x`
`curve('init',x,v)`	initialize user variable space (*WorkSpace*)
`curve('done')`	destroy user variable space (*WorkSpace*)
`curve('singmat')`	return singularity matrix (*Singularities*)
`curve('process',i,x)`	run processor code of singularity `i` at `x`(*Singularities*)
`curve('adapt',x,v)`	run adaptation code of problem (*Adapt*)
`curve('testf',ids,x,v)`	return evaluation of testfunctions `ids` at `x` (*Singularities*)
`curve('locate',i,x1,x2,v1,v2)`	return located singularity and tangent vector (*Locators*)
`curve('defaultprocessor',x,v,s)`	Initialize data for testfunctions and set some general singularity data

4. THE BIALTERNATE PRODUCT

If A, B are $n \times n$ matrices then $A \odot B$ is an $m \times m$ matrix where $m = n(n-1)/2$. Its entries ([4],p. 93) are given by

$$(A \odot B)_{(i,j)(k,l)} = \frac{1}{2}\left\{ \begin{vmatrix} a_{ik} & a_{il} \\ b_{jk} & b_{jl} \end{vmatrix} + \begin{vmatrix} b_{ik} & b_{il} \\ a_{jk} & a_{jl} \end{vmatrix} \right\} \quad (3)$$

where the indices are pairs of variables $(i,j),(k,l)$ with $n \geq i > j \geq 1$ and $n \geq k > l \geq 1$.

The special case of a bialternate product of the form $2A \odot I_n$ is so important that we simply call it the biproduct of A.

From (3) we infer:

$$(2A \odot I_n)_{(i,j)(k,l)} = \begin{cases} -a_{il} & \text{if } k = j, \\ a_{ik} & \text{if } k \neq i \text{ and } l = j, \\ a_{ii} + a_{jj} & \text{if } k = i \text{ and } l = j, \\ a_{jl} & \text{if } k = i \text{ and } l \neq j, \\ -a_{jk} & \text{if } l = i, \\ 0 & \text{else.} \end{cases} \quad (4)$$

4.1 Indexing strategy in $2A \odot I_n$

An important bifurcation on an equilibrium curve $f(u,\alpha) = 0$ of an ODE is the Hopf bifurcation where f_u has a conjugate pair $\pm i\omega$ of pure imaginary eigenvalues. In fact, matrices with zero - sum pairs of eigenvalues are important in several other bifurcation contexts as well.

A test function for a zero - sum pair of eigenvalues is the determinant of $2f_u \odot I_n$, cf. [4], §4.5. This test function covers both the Hopf case and the neutral saddle case (two real eigenvalues with sum zero). We avoid the computation of the eigenvalues because it is well known that they are not analytic functions of the entries of the matrix.

During the computation of some curves (at present equilibrium, limit point and Hopf) we evaluate the determinant of $2f_u \odot I_n$ at each computed point. To avoid the repetition of index computations we build the matrices of index values before actually starting the curve computation. Also, we exploit the sparsity of $2f_u \odot I_n$ which is due to the sparsity of I_n.The computation of $2f_u \odot I_n$ at each point of the curve then merely involves the evaluation of the three index matrices and a matrix addition and subtraction.

We first build an $n \times n$ matrix with entries $A(i) = i$. The Matlab command

```
a=reshape(1:n^2,n,n)}
```

builds such a matrix. For $n = 3$ we get:

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \quad (5)$$

We define a Matlab function `bialt(A)` which computes the indices of the nonzero entries of the biproduct and stores them in 3 square index matrices $A1, A2$ and $A3$ of dimension $n(n-1)/2$. Explicitly

$$(A1)_{(i,j)(k,l)} = \begin{cases} a_{jj} & \text{if } k=i \text{ and } l=j, \\ 0 & \text{else.} \end{cases} \quad (6)$$

$$(A2)_{(i,j)(k,l)} = \begin{cases} a_{il} & \text{if } k=j, \\ a_{jk} & \text{if } l=i, \\ 0 & \text{else.} \end{cases} \quad (7)$$

$$(A3)_{(i,j)(k,l)} = \begin{cases} a_{ik} & \text{if } k \neq i \text{ and } l=j, \\ a_{ii} & \text{if } k=i \text{ and } l=j, \\ a_{jl} & \text{if } k=i \text{ and } l \neq j, \\ 0 & \text{else.} \end{cases} \quad (8)$$

These are full matrices. However the biproduct of f_u with dimension $n(n-1)/2$ has only $n(n-1)(2n-3)/2$ functionally nonzero entries, so for large n it is rather sparse. Therefore exploiting the sparsity is recommended. In Matlab the command `[I,J,V]=find(X)` returns row and column indices of the nonzero entries in the matrix X and returns also a vector containing the nonzero entries in X. This is done for the three matrices $A1, A2, A3$ and the results are saved in the global variable of the computed curve *eds* (=equilibrium description). In the case of an equilibrium curve this is implemented by the commands:

```
[A1,A2,A3] = bialt(A);
[eds.BiAlt_M1_I,eds.BiAlt_M1_J,eds.BiAlt_M1_V]=
find(A1);
[eds.BiAlt_M2_I,eds.BiAlt_M2_J,eds.BiAlt_M2_V]=
  find(A2);
[eds.BiAlt_M3_I,eds.BiAlt_M3_J,eds.BiAlt_M3_V]=
  find(A3);
```

These indices are used to build a sparse matrix. The matlab command `S=sparse(I,J,V)` uses the rows of `[I,J,V]` to generate an max(`I`)$\times max$(`J`) sparse matrix. The two integer index vectors `I` and `J` and the real entries vector `V`, all have the same length, which is the number of the nonzeros in the resulting sparse matrix `S`. The computation of the determinant of $2f_u \odot I_n$ at each point of the curve then involves the evaluation of the three index matrices and a matrix addition and subtraction. It is illustrated by the following code:

```
A=J(:,1:ndim-1);%J(acobian) = f_u
A1=sparse(eds.BiAlt_M1_I,eds.BiAlt_M1_J,
A(eds.BiAlt_M1_V));
A2=sparse(eds.BiAlt_M2_I,eds.BiAlt_M2_J,
A(eds.BiAlt_M2_V));
A3=sparse(eds.BiAlt_M3_I,eds.BiAlt_M3_J,
A(eds.BiAlt_M3_V));
out = det(A1-A2+A3);
```

4.2 Indexing strategy in $A \odot A$

A test function for the Neimark-Sacker bifurcation is the determinant of the bialternate product matrix $M \odot M$(special case of (3)) where M is the monodromy matrix. From (3) we infer that $M \odot M \in R^{m \times m}$ is given by

$$(M \odot M)_{(i,j)(k,l)} = \begin{vmatrix} m_{jl} & m_{jk} \\ m_{il} & m_{ik} \end{vmatrix} = m_{jl}m_{ik} - m_{il}m_{jk} \quad (9)$$

where the indices are pairs of variables $(i,j),(k,l)$ with $n \geq i > j \geq 1$ and $n \geq k > l \geq 1$.

Again, to avoid the repetition of index computations we build the matrices of index values before actually starting the curve computation. The computation of $M \odot M$ at each point of the curve then just involves the evaluation of four index matrices and two entry-by-entry products and one matrix subtraction.

We define the Matlab function `bialtaa(nphase)` where nphase is the dimension of the vector containing the state variables, computes the indices of the nonzero entries of the bialternate product $M \odot M$. The output of `bialtaa(nphase)` consists of 4 full square index matrices $M1$, $M2$, $M3$ and $M4$ of dimension $n(n-1)/2$. $M1$, $M2$, $M3$ and $M4$ contain respectively the indices of the elements m_{jl}, m_{ik}, m_{il} and m_{jk}. Those four matrices are saved in the global variable of the limitcycle `lds` (=limitcycle description). The computation of $M \odot M$ is then given by

```
A = A(lds.bialt_M1).*A(lds.bialt_M2)-
A(lds.bialt_M3).*A(lds.bialt_M4);
```

5. CONTINUATION OF AN ODE EQUILIBRIUM IN A FREE PARAMETER

We 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, in particular the detection of the Hopf bifurcation using the determinant of the biproduct $2f_u \odot I_n$. We note that the standard Matlab `odeget` and `odeset` only support Jacobian matrices w.r.t. phase variables coded in the ode-file. However, we also 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 (Jacobianp) and higher-order derivatives. The new routines are compatible with the ones provided by Matlab.

We consider the differential equation

$$\frac{du}{dt} = f(u,\alpha), \quad u \in \mathbb{R}^n, \alpha \in \mathbb{R} \quad f : \mathbb{R}^{n+1} \to \mathbb{R}^n \quad (10)$$

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

$$F(x) = f(u,\alpha) = 0 \quad (11)$$

with $x = (u,\alpha) \in \mathbb{R}^{n+1}$. We note that the number of state variables and parameters is fixed.

An ODE file is an m-file function to define a differential equation problem. It is expected to respond to the arguments `ODEFILE (t, y, flag, p1, p2, ...)`, where `t` is the integration variable, `y` is a vector containing the values of the state variables, `flag` (ex. 'jacobian', 'jacobianp', 'hessian', 'init', ...) is a string indicating the type of information that the ODE file should return and `p1, p2,...` are additional parameters that the problem requires.

As an example we consider a 4-point discretization of the Bratu-Gelfand BVP(see [4]). This model is defined as follows:

$$\begin{cases} x' &= y - 2x + \alpha e^x \\ y' &= x - 2y + \alpha e^y. \end{cases} \quad (12)$$

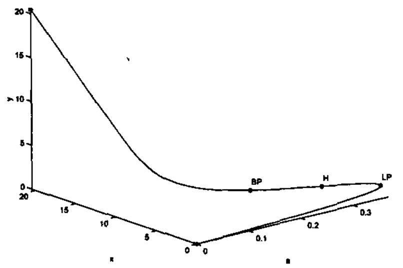

Figure 1: Equilibrium curve of bratu.m

It has 2 state variables x, y and one parameter α. This system has an equilibrium at $(x, y, \alpha) = (0,0,0)$ which we will continue with respect to α. The ODE file *bratu.m* describes this problem. In this description a full Jacobian is defined symbolically. The Hessian is not provided, so the continuer computes the second order derivatives internally by finite differences.

The equilibrium curve file has to 'know' which ode file to use, the values of all state variables, the value of all parameters and which parameter is active. This is provided by the command `[x0,v0]=init_EP_EP('bratu', [0;0], [0],[1])` which stores its information in a global structure `eds` and returns an initial point `x0` and empty tangent vector `v0`.

Now one starts the continuation with the command

```
[x,v,s,h,f]=cont('equilibrium', x0).
```

The equilibrium curve continuation finds three bifurcations: a limit point at $(x, y, \alpha) \approx (1.0; 1.0; 0.37)$, a Hopf (neutral saddle) at $(x, y, \alpha) \approx (2.0; 2.0; 0.27)$ and a branching point at $(x, y, \alpha) \approx (3.0; 3.0; 0.15)$. The resulting curve is plotted in Figure 1. We note that the branching bifurcation shows up as a discretization artifact.

6. CONTINUATION OF A SOLUTION TO A BOUNDARY VALUE PROBLEM IN A FREE PARAMETER

Discretized solutions of PDE's 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'.

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 unknows 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:

$$\begin{aligned} \frac{\partial X}{\partial t} &= \frac{D_x}{L^2}\frac{\partial^2 X}{\partial x^2} + A - (B-1)X + X^2 Y \\ \frac{\partial Y}{\partial t} &= \frac{D_y}{L^2}\frac{\partial^2 Y}{\partial x^2} + BX - X^2 Y \end{aligned} \tag{13}$$

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 will be imposed:

$$\begin{cases} X(0,t) = X(1,t) = A \\ Y(0,t) = Y(1,t) = \frac{B}{A} \end{cases} \tag{14}$$

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:

$$\begin{cases} X(x) &= A + 2\sin(\pi x) \\ Y(x) &= \frac{B}{A} - \frac{1}{2}\sin(\pi x) \end{cases} \tag{15}$$

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 (15) 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 6.1. does not provide sparse structures for 3 - dimensional arrays.

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 the 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 `bruss.m`, using the standard ODE file convention `[t,x0,options] = bruss([],[],'init',20,0.06)`. Within 'bruss.m' it is called with the parameter N ('init(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 equilibrium 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',350);
>opt=contset(opt,'Locators',[]);
```

We start the continuation process by the command
`[x,v,s,h] = cont('pde_1',x1,v1,opt).`

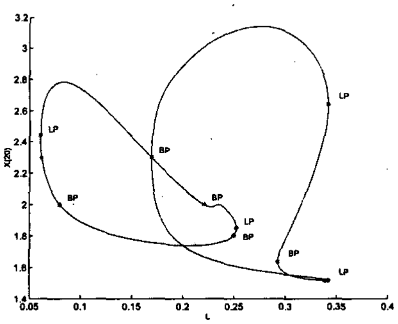

Figure 2: Equilibrium curves of `bruss.m`

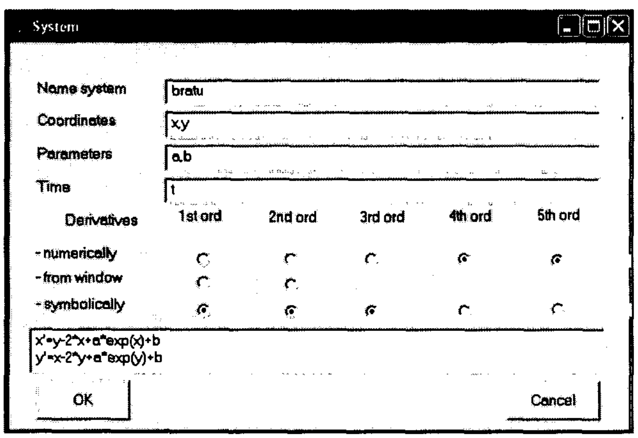

Figure 3: Example of the SelectSystemsEdit window for the system bratu.

In this case the number of state variables can be a parameter and the Jacobian can be sparse.
The routine `cpl` can be used to plot two or three components of the curve. Running the command `testbruss2` adds a new curve, namely the one that branches off the first one at $L \approx 0.17$. and presents it as in Figure 2, where axes labels are added manually.

7. GRAPHICAL USER INTERFACE

An important application of CL_MATCONT is to the bifurcation analysis of ODEs, as we briefly touched upon in section 5. For this case a graphical user interface version of CL_MATCONT is available at

`http://allserv.rug.ac.be/~ajdhooge/research.html.`

It is called MATCONT. The present version of MATCONT works well with Windows version 6.* of Matlab. On a Unix platform it is recommended to use version 6.1 of Matlab since version 6.0 is unable to load the provided examples.

A major advantage of MATCONT is the possibility to generate higher order derivatives. If the Matlab symbolic toolbox is installed, there is an easy to use option (see Figure 3) available that computes the derivatives symbolically and pastes the results in the odefile.

Another major advantage of MATCONT is its filing system. MATCONT builds an archive to store all used dynamical systems with all data specified for their analysis as well as the results of the analysis.

Information on computed objects and curves is stored as mat-files by MATCONT. A curve contains coordinates of points and additional data required to redraw and recompute the curve. MATCONT creates all necessary files automatically. The user can read directly from the archives where he can study the computed results, make plots, print them out etcetera.

8. REFERENCES

[1] E.L. Allgower and K. Georg, Numerical Continuation Methods: An introduction, Springer-Verlag, 1990

[2] 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.

[3] E. J. Doedel, A. R. Champneys, T. F. Fairgrieve, Yu. A. Kuznetsov, B. Sandstede and X. J. Wang, AUTO97 : Continuation and Bifurcation Software for Ordinary Differential Equations (with HomCont), User's Guide, Concordia University, Montreal, Canada 1997. (`http://indy.cs.concordia.ca`).

[4] W.J.F. Govaerts, Numerical Methods for Bifurcations of Dynamical Equilibria, SIAM, 2000

[5] Yu.A. Kuznetsov, Elements of Applied Bifurcation Theory, Springer-Verlag, 1998

[6] Yu.A. Kuznetsov and V.V. Levitin, CONTENT: Integrated Evironment for analysis of dynamical systems. CWI, Amsterdam 1997: `ftp://ftp.cwi.nl/pub/CONTENT`

[7] W. Mestrom, Continuation of limit cycles in MATLAB, Master Thesis, Mathematical Institute, Utrecht University, The Netherlands, 2002.

[8] A. Riet, A Continuation Toolbox in MATLAB, Master Thesis, Mathematical Institute, Utrecht University, The Netherlands, 2000.

[9] D. Roose et al., Aspects of continuation software, in : Continuation and Bifurcations: Numerical Techniques and Applications, (eds. D. Roose, B. De Dier and A. Spence), NATO ASI series, Series C, Vol. 313, Kluwer 1990, pp. 261-268