CompDTIMe: Computing one-dimensional invariant manifolds for saddle points of discrete time dynamical systems

A. Panchuk

February 20, 2015

Abstract

This paper describes briefly main functionalities and exploited numerical methods of the package CompDTIMe which consists of several Matlab routines. This package allows one to calculate two-dimensional bifurcation diagrams, to find periodic points not depending on whether they are attracting, to compute one-dimensional stable and unstable manifolds of saddle points. Certain functions are also provided for plotting the numerical outcome by means of Matlab.

1 Introduction

Dynamical systems with discrete time arise as models to describe various economic, engineering, or physical phenomena. Describing certain dynamical aspects of these models usually gives prediction about real system behaviour. Direct simulation is a powerful tool, however it can be used for quite restricted number of tasks. For instance, in case of slow convergence to an attractor the computation time can become rather large. Moreover, using direct simulation one cannot find unstable periodic points or compute invariant manifolds. Bifurcation theory provides a better framework for such investigation, but most analytical techniques are limited to simple systems.

Concerning studies of asymptotic behaviour and bifurcation phenomena in discrete and continuous time dynamical systems, there is a large number of numerical methods implemented by different software tools and routines, such as Dynamics[**?**], AUTO[**?**], CONTENT[**?**], DsTool[**?**], Numerical

Recepies[**?**], and many others. However, some of these programmes run only on Linux/Unix systems which makes them impossible to be spread among Windows users. Furthermore, certain pieces were developed some time ago and are not supported currently, therefore they cannot be run on newer operation systems. In addition, not many of known tools are able to compute stable and unstable invariant manifolds of saddle periodic points.

This paper describes a package CompDTIMe meant to be run in Matlab (compatible with the version R2010b) which is consisted of several routines for analysing certain aspects of dynamical behaviour of discrete time dynamical systems. In particular, it can calculate 2D bifurcation diagrams in the parameter space of a map, find periodic points and determining their type, compute invariant 1D stable and unstable manifolds.

The rest of the paper is organised as follows. In Sec. 2 we present main numerical methods used by CompDTIMe. Sec. 3 describes briefly basic principles of the package usage together with simple illustrative examples. Sec. 4 concludes.

2 Description of the Methods

Let us consider a map $\mathbf{F} : \mathbb{R}^{n+m} \to \mathbb{R}^n$ written in general form as

$$\mathbf{F} : \mathbf{x} \mapsto \mathbf{F}(\mathbf{x}, \mu) = \begin{cases} F_1(\mathbf{x}, \mu) \\ \dots \\ F_n(\mathbf{x}, \mu) \end{cases} \tag{1}$$

where $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{R}^n$ is the variable vector, $\mu = (\mu_1, \dots, \mu_m) \in \mathbb{R}^m$ is the parameter vector, and $F_i : \mathbb{R}^{n+m} \to \mathbb{R}$, $i = 1, \dots, n$. The related discrete time dynamical system is then defined as

$$\mathbf{x}_{t+1} = \mathbf{F}(\mathbf{x}_t, \mu), \quad t = 0, 1, \dots. \tag{2}$$

Starting with an arbitrary initial condition $\mathbf{x}_0 = (x_1^0, \dots, x_n^0)$ a particular orbit $\{\mathbf{x}_t\}_{t=0}^{\infty}$ of (2) is defined. Asymptotically the behaviour of this orbit can be regular (a fixed point, an n-cycle) or non-regular (chaos, divergence). Clearly, in case of coexistence of several attractors asymptotic dynamics of two distinct orbits may be different.

2.1 Finding periodic points: Newton's method

One of the basic tasks in numerical analysing of a dynamical system (2) is to find its periodic points, namely, the points $\mathbf{x} \in \mathbb{R}^n$ which satisfy $\mathbf{F}^T(\mathbf{x}, \mu) = \mathbf{x}$ for some integer $T \geq 1$. Technically the problem of finding periodic points is reduced to finding roots of the equation

$$\mathbf{G}_\mu(\mathbf{x}) \overset{def}{=} \mathbf{F}^T(\mathbf{x}, \mu) - \mathbf{x} = 0 \tag{3}$$

with $\mathbf{G}_\mu = (G_1, \ldots, G_n) : \mathbb{R}^n \to \mathbb{R}^n$ with some fixed parameter vector μ.

2.1.1 One-dimensional problems

In one-dimensional case

$$G_\mu(x) = 0, \tag{4}$$

$G_\mu : \mathbb{R} \to \mathbb{R}$, $x \in \mathbb{R}$, there exists a variety of iterative methods for deriving solutions of (4). All these methods use the same principle: one starts from an approximate trial solution which is then improved until some predetermined convergence criterion is satisfied. However, it is often very important to have a good first guess for a solution.

One of the most popular among one-dimensional root finding iterative methods is Newton's method, called also Newton-Raphson method (see, *e. g.*, [**?**]). This method is distinguished from other methods because it requires the evaluation of both the function $G_\mu(x)$, and its derivative $G'_\mu(x)$. The latter can be either given explicitly or approximated by a numerical difference

$$G'_\mu(x) \approx \frac{G_\mu(x + dx) - G_\mu(x)}{dx}, \quad dx \ll 1.$$

The reason is that Newton's method iterative formula is based on the familiar Taylor series expansion of a function in the neighbourhood of a point:

$$G_\mu(x + \delta) \approx G_\mu(x) + G'_\mu(x)\delta + \frac{G''_\mu(x)}{2}\delta^2 + \ldots \tag{5}$$

If δ is small enough and G_μ is well-defined, then higher-order terms are negligible, thus, one constructs an iterative law

$$x_{k+1} = x_k - \frac{G_\mu(x_k)}{G'_\mu(x_k)}, \quad k \geq 0. \tag{6}$$

After choosing a trial guess x_0 expression (6) is iterated until a desired accuracy is reached, that is, when $|x_{k+1} - x_k| < \varepsilon$ with a certain predefined $0 < \varepsilon \ll 1$.

Newton's method is considered to be rather efficient. Indeed, when the initial guess is appropriate (close enough to a root) then the sequence $\{x_0, x_1, \ldots\}$ converges to the root *quadratically.* However, if the initial guess is far from a root then the higher-order terms in the series are important, which can lead to meaningless x_{k+1}. For instance, when x_k gets close to a local extremum of G_μ the correction term in (6) becomes inaccurately large because G'_μ nearly vanishes.

One should also keep in mind that if the derivative G'_μ is not given explicitly but approximated then the convergence rate is only $\sqrt{2}$. In case when G'_μ cannot be derived explicitly, it is advised to use other methods for finding roots of (4), for instance, secant method (see, *e. g.*, [**?**]).

2.1.2 Multi-dimensional problems

In multidimensional case with $n > 1$ solving (3) is more tricky, and there are no good general root finding methods. Indeed, one has to find points mutually common to n unrelated zero-contour hypersurfaces, each of dimension $n - 1$. Usually one has to use additional information, specific to every particular problem.

Similarly to the case $n = 1$ Newton's method again appears to be rather powerful (also being the simplest), provided that an initial guess is sufficiently good. On the other hand, the method can fail to find the desired root, because this root does not exist near the initial guess solution. To improve such a situation one can combine Newton's method with a globally convergent strategy that will guarantee some progress towards the solution at each iteration (as suggested, for example, in [**?**]).

First we obtain the iterative formula of Newton's method similarly to the one-dimensional case. In the neighbourhood of an arbitrary point $\mathbf{x}$ each of the functions G_i, $i = \overline{1, n}$, can be expanded in Taylor series

$$G_i(\mathbf{x} + \delta\mathbf{x}) = G_i(\mathbf{x}) + \sum_{j=1}^{n} \frac{\partial G_i(\mathbf{x})}{\partial x_j} \delta\mathbf{x} + O(\delta\mathbf{x}^2), \quad \|\delta\mathbf{x}\| \ll 1. \tag{7}$$

Or, equivalently, in matrix notation

$$\mathbf{G}_\mu(\mathbf{x} + \delta\mathbf{x}) = \mathbf{G}_\mu(\mathbf{x}) + \mathbf{J}(\mathbf{x}) \cdot \delta\mathbf{x} + O(\delta\mathbf{x}^2), \tag{8}$$

where $\mathbf{J}$ is a Jacobian matrix of $\mathbf{G}_\mu$. From (8) (omitting higher-order terms) one obtains

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \delta\mathbf{x}_k = \mathbf{x}_k - \mathbf{J}^{-1}(\mathbf{x}_k) \cdot \mathbf{G}_\mu(\mathbf{x}_k). \tag{9}$$

Next we notice that the problem of finding roots of (3) is closely related to the problem of minimising the functional

$$f = \frac{1}{2}|\mathbf{G}_\mu|^2 = \frac{1}{2}\mathbf{G}_\mu \cdot \mathbf{G}_\mu. \tag{10}$$

More precisely, every solution of (3) also minimises (10). The opposite is not true though, thus, simply solving minimisation problem for (10) will not necessarily gives a root of (3). However, the Newton step

$$\delta\mathbf{x} = \mathbf{J}^{-1}(\mathbf{x}_k) \cdot \mathbf{G}_\mu(\mathbf{x}_k) \tag{11}$$

definitely gives a direction along which the functional f decreases. Indeed,

$$\nabla f(\mathbf{x}_k)\cdot\delta\mathbf{x}_k = (\mathbf{G}_\mu(\mathbf{x}_k)\cdot\mathbf{J}(\mathbf{x}_k))\cdot(-\mathbf{J}^{-1}(\mathbf{x}_k)\cdot\mathbf{G}_\mu(\mathbf{x}_k)) = -\mathbf{G}_\mu(\mathbf{x}_k)\cdot\mathbf{G}_\mu(\mathbf{x}_k) < 0.$$

In such a way one can derive the improved Newton method which consists of the following. First the full Newton step is tried, since we have quadratic convergence in the neighbourhood of the root. Then we check whether the computed $\delta\mathbf{x}_k$ also reduces the functional f (10). If it is not the case, we backtrack along the Newton direction $\delta\mathbf{x}_k$ until we have an acceptable step. One of possible backtracking algorithms is described below.

Suppose that moving the full Newton step $\delta\mathbf{x}_k$ does not decrease f. Let us find a certain $0 < \lambda < 1$ which guarantees that $f(\mathbf{x}_k + \lambda \cdot \delta\mathbf{x}_k)$ decreases with respect to $f(\mathbf{x}_k)$. For that we define

$$g(\lambda) \overset{def}{=} f(\mathbf{x}_k + \lambda \cdot \delta\mathbf{x}_k) \tag{12}$$

so that

$$g'(\lambda) = \nabla f(\mathbf{x}_k + \lambda \cdot \delta\mathbf{x}_k) \cdot \delta\mathbf{x}_k. \tag{13}$$

We then approximate g with the most current information we have, and choose λ to minimise this approximated expression. In such a way we iteratively construct a sequence $\{\lambda_{kl}\}_{l=0}^N$ until λ_{kN} satisfies certain criteria. The first criteria is, obviously,

$$\mathbf{G}_\mu(\mathbf{x}_k + \lambda_{kN} \cdot \delta\mathbf{x}_k) < \mathbf{G}_\mu(\mathbf{x}_k). \tag{14}$$

Then we need to require that $f(\mathbf{x}_k + \lambda_{kN} \cdot \delta\mathbf{x}_k) < f(\mathbf{x}_k)$. However, it might happen that this mere condition is not sufficient, in particular, when f decreases too slowly with respect to the step lengths $\lambda_{kN}\delta\mathbf{x}_k$ (for example, see [?], p. 117). To avoid such a situation it is enough to require

$$f(\mathbf{x}_k + \lambda_{kN} \cdot \delta\mathbf{x}_k) \le f(\mathbf{x}_k) + \alpha\nabla f(\mathbf{x}_k)\lambda_{kN}\delta\mathbf{x} \qquad (15)$$

with a fixed positive $\alpha < 1$ (choosing $\alpha = 10^{-4}$ serves fine). The third criteria is that the steps should not be too small, and we require that $\lambda_{k(l+1)} \ge 0.1\lambda_{kl}$.

We always start from the full Newton step, and thus $\lambda_{k0} = 1$. At this time we know the values $g(0)$, $g'(0)$, and $g(1)$. If λ_{k0} is not acceptable then we can approximate $g(\lambda)$ by a quadratic expression:

$$g(\lambda) \approx \big(g(1) - g(0) - g'(0)\big)\lambda^2 + g'(0)\lambda + g(0). \qquad (16)$$

The minimum of (16) is attained at

$$\lambda = \lambda_{k1} \overset{def}{=} \frac{g'(0)}{2\big(g(1) - g(0) - g'(0)\big)}. \qquad (17)$$

Here it can be also shown that $\lambda_{k1} \lesssim 0.5$ for small α. If λ_{k1} is not acceptable neither, we should proceed.

On the second and subsequent backtrack iterations, we approximate g by a cubic expression using the two most recent values $g(\lambda_{kl})$ and $g(\lambda_{k(l-1)})$:

$$g(\lambda) = a\lambda^3 + b\lambda^2 + g'(0)\lambda + g(0) \qquad (18)$$

with

$$\begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\lambda_{kl} - \lambda_{k(l-1)}} \begin{bmatrix} 1/\lambda_{kl}^2 & -1/\lambda_{k(l-1)}^2 \\ -\lambda_{k(l-1)}/\lambda_{kl}^2 & \lambda_{kl}/\lambda_{k(l-1)}^2 \end{bmatrix} \cdot \begin{bmatrix} g(\lambda_{kl}) - g'(0)\lambda_{kl} - g(0) \\ g(\lambda_{k(l-1)}) - g'(0)\lambda_{k(l-1)} - g(0) \end{bmatrix}$$

The minimum of the cubic (18) is at

$$\lambda_{k(l+1)} = \frac{-b + \sqrt{b^2 - 3ag'(0)}}{3a} \overset{def}{=} \lambda_{kl}^*.$$

We also enforce $\lambda_{k(l+1)}$ to lie between $0.1\lambda_{kl}$ and $0.5\lambda_{kl}$, that is, we put

$$\lambda_{k(l+1)} = \min\{0.5\lambda_{kl} \max\{0.1\lambda_{kl}\lambda_{kl}^*\}\}.$$

The procedure is repeated till a certain λ_{kN} satisfies both (14) and (15). Then the next Newton step is performed.

2.2 Search circle algorithm

The method used to construct invariant manifolds is the Search Circle (SC) algorithm [**?**, **?**]. It grows iteratively a piecewise linear approximation for a one-dimensional manifold by adding new points according to the local curvature properties of this manifold. The difference of the SC algorithm from other methods is that it *does not need the inverse*.

2.2.1 Stable manifold

The main idea of the method is as follows. Given a fixed point p_0 we construct a piecewise linear approximation of $W^s(p_0)$ by computing successive points $M = \{p_0, p_1, \ldots, p_k, \ldots\}$ at varying distance from each other. The first point p_1 is taken at a small distance $\delta > 0$ from p_0 along the stable eigenspace $E^s(p_0)$. For obtaining each successive point we draw a circle having the centre in the last computed point p_k and the radius Δ_k (the adoptive step value which may vary through the procedure). The new point p_{k+1} of the manifold is searched on this circle by using the fact that p_{k+1} is mapped under $\mathbf{F}$ to a piece of the manifold that was already computed. Namely, there exists a point $\hat{p}$ belonging to some already computed segment (p_i, p_{i+1}) of the manifold, and $\mathbf{F}(p_{k+1}) = \hat{p}$[1].

Suppose that the manifold has been grown up to some point p_k such that $M = \{p_0, p_1, \ldots, p_k\}$. We draw a circle $C(p_k, \Delta_k)$ of the radius Δ_k with a centre in p_k (green colour in Fig. 1). Provided that Δ_k is small enough, the circle $C(p_k, \Delta_k)$ intersects the manifold $W^s(p_0)$ only twice, namely, in the point p^* (which belongs to the already computed segment, and thus, is not the target point) and the target point p_{k+1}. The image $\mathbf{F}(C(p_k, \Delta_k))$ is then a closed curve (magenta colour in Fig. 1), which is located near some segment of the already computed manifold. Obviously, $\mathbf{F}(C(p_k, \Delta_k))$ also has two intersections with $W^s(p_0)$, from which the one $\mathbf{F}(p^*)$ we are not interested in (in Fig. 1 it belongs to (p_{i-2}, p_{i-1})), while the other $\mathbf{F}(p_{k+1})$ (belonging to (p_i, p_{i+1})) is the image of the point p_{k+1} searched for.

The distance Δ_k is chosen so that it is acceptable meaning that the interpolation error is within the desired accuracy which ensures that the approximated manifold is of reasonable resolution. For that the method suggested in [**?**] is used, namely, it is assumed that the angle α_k between the lines

[1]Note, that in certain cases of non-invertible maps, the point p_{k+1} is mapped to the previously calculated part of the manifold not by $\mathbf{F}$, but by some its iterate $\mathbf{F}^j$.

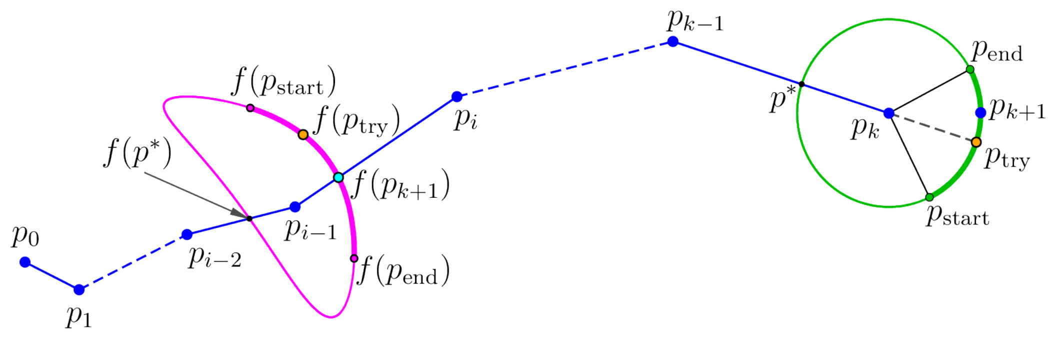

Figure 1: Piecewise linear approximation of the stable manifold by using SC algorithm.

drawn through p_{k-1}, p_k and p_k, p_{k+1} (see Fig. 2) lies within certain bounds $\alpha_{\min} < \alpha_k < \alpha_{\max}$. If $\alpha_k > \alpha_{\max}$, the point p_{k+1} is too far apart from the point p_k, so the step should be reduced, and the procedure is repeated with the decreased step. If $\alpha_k < \alpha_{\min}$, the point p_{k+1} is too close to p_k, however, for making the decision of enlarging the step or not we check also another condition $\Delta_k \alpha_k > (\Delta\alpha)_{\min}$, which is explained below.

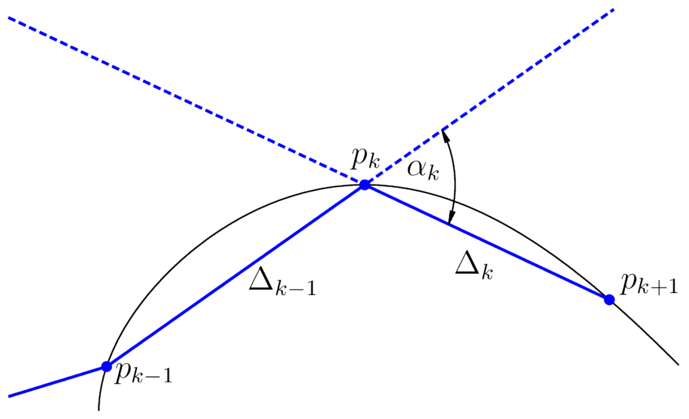

Figure 2: The angle between the lines drawn through p_{k-1}, p_k and p_k, p_{k+1}.

We can ensure immediately that α_k does not exceed $\alpha_{\max}$ by only searching the part of the circle $C(p_k, \Delta_k)$ that satisfies this criterion. This search region is the arc between the points p_{start} and p_{end}, indicated by thicker green curve in Fig. 1, and its image, which is the thicker magenta curve between $\mathbf{F}(p_{\text{start}})$ and $\mathbf{F}(p_{\text{end}})$, intersects the previously computed part of the manifold only once, which automatically ensures that we do not accidentally search for a pre-image of the point p^* (that we are not interested in).

In addition, to control the local interpolation error, we also check that the product $\Delta_k \alpha_k$ lies between $(\Delta\alpha)_{\min}$ and $(\Delta\alpha)_{\max}$. If $\Delta_k \alpha_k > (\Delta\alpha)_{\max}$, the step Δ_k is halved and the procedure of finding p_{k+1} is repeated. If $\Delta_k \alpha_k < (\Delta\alpha)_{\min}$ together with $\alpha_k < \alpha_{\min}$, then for the next iteration the step is enlarged $\Delta_{k+1} = 2\Delta_k$. This ensures that the number of points used to approximate the manifold is in some sense optimised for the required accuracy constraints. Note that, at sharp folds it may be necessary to accept $\alpha_k > \alpha_{\max}$ due to Δ_k becoming very small. In this case we accept the "unacceptable" point if $\Delta_k < \Delta_{\min}$, where $\Delta_{\min}$ is also a predefined parameter.

Finally, in order to find p_{k+1} we need to define which segment of the previously calculated manifold contains the intersection point with $\mathbf{F}(C(p_k, \Delta_k))$. We first try the segment (p_{i-1}, p_i) that was used in the previous step (to find a candidate for p_k). If the image $\mathbf{F}(p_{k+1})$ of the candidate for p_{k+1} lies on the line through p_{i-1} and p_i, but not in the segment (p_{i-1}, p_i), we discard this point and repeat the algorithm with the following segment (p_i, p_{i+1}) to find a new candidate for p_{k+1}. (If the map has multiple pre-images then we may need to search for $\mathbf{F}(p_{k+1})$ on the previous segment (p_{i-2}, p_{i-1})). To find the candidate for p_{k+1} we use the bisection method for the angle α_k, and allow the point $\mathbf{F}(p_{k+1})$ to lie at a maximum distance of some small ε_B (the bisection error) from the detected segment, say, $I_j = (p_j, p_{j+1})$. Note, that in this procedure the points $\mathbf{F}(p_{\text{start}})$ and $\mathbf{F}(p_{\text{end}})$ must lie on the opposite sides of the segment I_j. If they do not (for instance, if there is a sharp fold in the manifold) the search region for the angle α_k is increased, and a warning message is printed.

To sum up, a single run of the algorithm may be briefly described by the following steps:

1. We put $\Delta_k = \Delta_{k-1}$ and draw a circle $C(p_k, \Delta_k)$.

2. We take the image of the arc between p_{start} and p_{end} and check if $\mathbf{F}(p_{\text{start}})$ and $\mathbf{F}(p_{\text{end}})$ lie on the opposite sides of the line drawn through p_{i-1} and p_i. If not, we increase the search region for α_k.

3. We use bisection method for finding the candidate p_{k+1} such that $\mathbf{F}(p_{k+1})$ lies within the distance ε_B from the line drawn through p_{i-1} and p_i.

4. If $\mathbf{F}(p_{k+1})$ lies outside the segment (p_{i-1}, p_i), we discard the point p_{k+1} and restart the procedure with using the segment (p_i, p_{i+1}).

5. If $\mathbf{F}(p_{k+1})$ lies inside the segment (p_{i-1}, p_i), we check the condition $\Delta_k\alpha_k < (\Delta\alpha)_{\max}$, and if it fails, we assign $\Delta_k = \Delta_k/2$ and restart the procedure.

6. If $\Delta_k\alpha_k < (\Delta\alpha)_{\max}$, then we check also that $\Delta_k\alpha_k > (\Delta\alpha)_{\min}$. If not and additionally $\alpha_k < \alpha_{\min}$, we accept the found point, but assign $\Delta_{k+1} = 2\Delta_k$ for the next run of the algorithm.

7. Finally, if the angle search region was enlarged and $\alpha_k > \alpha_{\max}$, we also check if $\Delta_k < \Delta_{\min}$. If yes, we accept the found point ignoring the failed criterion.

2.2.2 Unstable manifold

The procedure for the unstable manifold is quite similar, with the main difference that now mapping is in the reverse direction. So that, having $M = \{p_0, p_1, \ldots, p_k\}$ already computed, we would like to find a point p_{k+1} such that some point q_{k+1}, which belongs to a certain already known segment $I_i = (p_{i-1}, p_i)$, is mapped to p_{k+1}, that is, $\mathbf{F}(q_{k+1}) = p_{k+1}$. We also assume that the point p_{k+1} lies at an approximate distance Δ_k from p_k (see Fig. 3).

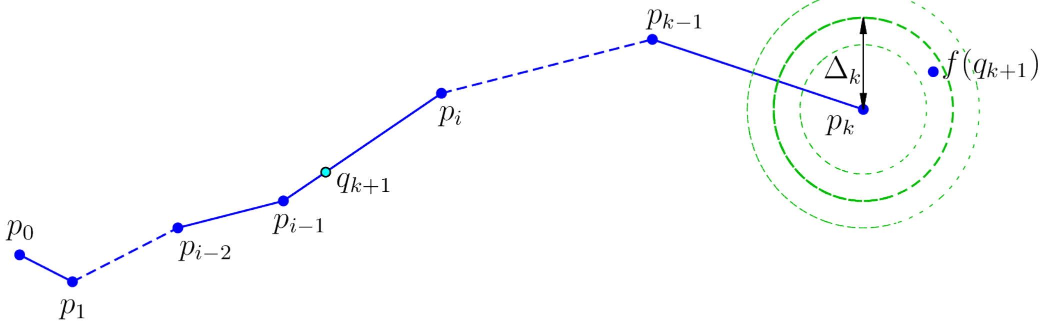

Figure 3: Unstable manifold approximation

Then, to find the candidate for p_{k+1}, we find the images of the points of the segment I_i, which are mapped to some neighbourhood of p_k. We use the bisection to detect the point q_{k+1} whose image is located near the point p_k so that $\Delta_k - \varepsilon_B < \|\mathbf{F}(q_{k+1}) - p_k\| < \Delta_k + \varepsilon_B$. If such a point does not exist, we take the next segment $I_{i+1} = (p_i, p_{i+1})$ and repeat the procedure.

Finally, we check the same accuracy conditions

$$\alpha_{\min} < \alpha_k < \alpha_{\max}, \quad (\Delta\alpha)_{\min} < \Delta_k\alpha_k < (\Delta\alpha)_{\max},$$

adjusting the step as described for the stable manifold case. Similarly, we accept the point p_{k+1} if $\alpha_k > \alpha_{\max}$, but $\Delta_k < \Delta_{\min}$.

3 Description of the Routine

In this section we describe main features of CompDTIMe. The package consists of a set of routines running in Matlab (compatible with the version R2010b) which is a widely used environment for scientific computing [?]. The package can perform the following tasks:

- Plot 2D bifurcation diagrams (period diagrams);
- Find periodic points of a predefined period and determining the type of these points (stable, unstable, or saddle).
- Compute invariant 1D stable and unstable manifolds (for 2D maps only).

CompDTIMe has no graphical user interface, but a number of routines are provided to plot periodic points, invariant manifolds, and stability information.

3.1 Initialisation

Before starting to use CompDTIMe it is necessary to define several constants which control running numerical methods. The easiest way to do this is to load the file `init.m` available in the routine's root directory. This file will add as well necessary directory paths to the Matlab's `PATH` variable.

The default values of all required constants are also stored in the file `data/default.mat`, which can be loaded to restore these values at any time. The full list of constants together with their defaults and descriptions can be found in Appendix A.

3.2 Definition of a new map

A new map of the form (1) can be defined by using a function handle (for more details on function handles see [?]). Namely, a user has to create a new Matlab script file and describe the right-hand side of the map $\mathbf{F}$ as presented in Listing 1.

Listing 1: user_map.m

```
function y = <user_map>(x, varargin)
% a comment and description
{ some optional commands }
y(1) = <equation 1>
....
y(n) = <equation n>
```

Here `varargin` is the variable-length input argument list, which should present the related parameter vector μ of the map $\mathbf{F}$.

A Jacobi matrix $\mathbf{J}(\mathbf{x}_0, \mu)$ at the point $\mathbf{x}_0 = (x_1^0, \ldots, x_n^0)$ for the user defined map $\mathbf{F}$ with the parameter vector μ has the form

$$\mathbf{J}(\mathbf{x}_0, \mu) = \begin{pmatrix} \frac{\partial F_1(\mathbf{x}_0, \mu)}{\partial x_1} & \ldots & \frac{\partial F_1(\mathbf{x}_0, \mu)}{\partial x_n} \\ \vdots & \ldots & \vdots \\ \frac{\partial F_n(\mathbf{x}_0, \mu)}{\partial x_1} & \ldots & \frac{\partial F_n(\mathbf{x}_0, \mu)}{\partial x_n} \end{pmatrix}. \tag{19}$$

This matrix may be given explicitly, which improves calculation speed of methods for finding periodic points and obtaining their eigenvalues and eigenvectors. The function handle for $\mathbf{J}$ should be contained in a separate file, which must have the name `<user_map>_J.m`, where `<user_map>.m` is the name of the file which contains definition of the corresponding user map. Furthermore, the Jacobi matrix function `<user_map>_J` must have the same number of parameters as the map function `<user_map>`. For example, see Listing 2.

Listing 2: user_map_J.m

```
function J = <user_map_J>(x, param1, param2, ....)
% a comment and description
{ some optional commands }
J(1, 1) = <equation 11>
J(1, 2) = <equation 12>
....
J(n, n) = <equation nn>
```

The matrix $\mathbf{J}(\mathbf{x}_0, \mu)$ may also be calculated numerically by using the function derivative approximation scheme:

$$\frac{\partial F_i(\mathbf{x}_0, \mu)}{\partial x_j} = \frac{F_i(\mathbf{x}_1, \mu) - F_i(\mathbf{x}_0, \mu)}{dx},$$

where $\mathbf{x}_1 = (x_1^0, \ldots, x_j^0 + dx, \ldots, x_n^0) \in \mathbb{R}^n$ and the small increment of the argument $dx \ll 1$. This scheme is implemented by using the predefined function `jacob`, which has the following format

```
function J = jacob(fhandle, x0, dx, varargin)
```

Here `fhandle` is the map function handle, `x0` is the target point $\mathbf{x}_0$, `dx` represents the small increment, and `varargin` is the variable-length input argument list, which should present the related parameter vector μ of the map $\mathbf{F}$.

For example, let us consider Hénon map $H : \mathbb{R}^2 \to \mathbb{R}^2$

$$H : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} y + 1 - ax^2 \\ bx \end{pmatrix} = H(x, y) \tag{20}$$

with two parameters a and b. Listings 3 and 4 show files defining both the new map and the related Jacobian matrix.

Listing 3: henon.m

```
function y = henon(x, a, b)
    y(1) = b*x(2) + a - x(1)^2;
    y(2) = x(1);
```

Listing 4: henon_J.m

```
function y = henon_J(x, a, b)
    J = zeros(2, 2);
    J(1, 1) = -2*x(1);
    J(1, 2) = b;
    J(2, 1) = 1;
```

3.3 Period Diagrams

The procedure to produce 2D bifurcation diagrams (period diagrams) consists usually of two steps. First, one obtains the data for the diagram, and then uses this data to make the 2D colour plot.

The method for getting the period data is rather straightforward. For two target function parameters μ_k and μ_l, $1 \leq k \leq n$, $1 \leq l \leq n$, $k \neq l$, one defines a certain $r \times s$ uniform rectangular mesh $M = \{M_{ij}\}_{i=1,j=1}^{r,s}$, $M_{ij} = (\mu_{ki}, \mu_{lj})$ (the remaining parameters stay fixed). For each pair M_{ij}, starting from the given initial orbit point $\mathbf{x}_0$ a target map $\mathbf{F}$ is iterated a certain number of times T (transient time). Then the last obtained orbit point $\mathbf{x}_T$ is checked for being periodic or not with a predefined maximal period value P.

3.3.1 Obtaining period data

For obtaining data for a period diagram the function `period_bd_calc` should be used, which has the format

```
function [param1, param2, periods] = ...
      period_bd_calc(fhandle, x0, prange, dp, pidx, ...
      trans, maxperiod, tolerance, divergence, varargin)
```

The meaning of the parameters is as follows:

- `fhandle` is a related function handle.
- `x0` is an initial point for obtaining the orbit at each system run. If `x0` is a numeric array, then for each parameter pair the initial orbit point is reset to `x0`. However, one may force the initial orbit point to be chosen randomly if necessary. For this `x0` must be set to a string of the form `'rand(<n>)'` where `<n>` is the dimension of the variable vector $\mathbf{x}$.
- `prange` defines ranges for two target parameters μ_k and μ_l. It is a 2×2 matrix $\begin{pmatrix} \mu_{k1} & \mu_{kr} \\ \mu_{l1} & \mu_{ls} \end{pmatrix}$.
- `dp` is a 2-dimensional vector defining increments for two target parameters.
- `pidx` is a 2-dimensional vector defining the parameter indices.
- `trans` is a transient number of iterations T.
- `maxperiod` defines a maximal potential period P of the orbit.
- `tolerance` is an accuracy for checking parity of two orbit points. Namely, if the norm $\|\mathbf{x}_1 - \mathbf{x}_2\|$ is less than `tolerance`, then the points $\mathbf{x}_1$ and $\mathbf{x}_2$ are considered to be equal.
- `divergence` is a limit for detecting divergence to infinity. Namely, if at a certain iteration the norm $\|\mathbf{x}\|$ becomes larger than `divergence`, then the related orbit is considered to diverge.

- `varargin` represents the parameter vector μ of the iterated map $\mathbf{F}$, which can be specified either as a comma separated list, or as a single cell-array parameter. (For the two target parameters arbitrary values can be given).

After finishing the calculation process the function `period_bd_calc` returns three objects:

- `param1` contains the mesh vector for the first target parameter $(\mu_{k1}, \ldots, \mu_{kr})$,
- `param2` contains the mesh vector for the second target parameter $(\mu_{l1}, \ldots, \mu_{ls})$,
- and `periods` is the $r \times s$ numerical matrix, whose cell ij represents the calculated orbit period corresponding to the parameter pair M_{ij}. Note, that the value `-1` is related to diverging orbits, while `0` means either that dynamics is non-regular, or that the orbit period is larger than `maxperiod`.

3.3.2 Plotting period data

The data obtained by `period_bd_calc` can be used to generate a period diagram graph in either an external plotting program, or inside Matlab by invoking the embedded function `pcolor`. For user convenience CompDTIMe contains the predefined function `bd_plot` which calls `pcolor` and makes basic settings of the figure. The format of `bd_plot` is

```
function [hfig, hcbar] = bd_plot(X, Y, C, maxperiod)
```

This function produces a pseudocolour plot[2] of the elements of `C` at the locations specified by `X` and `Y`. That is, `X` and `Y` are vectors of the length r and s, respectively, which determine the grid, while `C` is an $r \times s$ matrix whose elements define the colours to be used. Clearly, to plot the data obtained by `period_bd_calc` one has to replace `X` with `param1`, `Y` with `param2`, and `C` with `period`. The argument `maxperiod` means the highest periodicity to be plotted. Namely, only those elements of the matrix `C` will appear on the graph which are less than or equal to `maxperiod`.

[2]A pseudocolour plot is represented by a rectangular array of cells each being plotted with a certain colour. For details see [?].

The plot will be produced by using the preset colour palette (defined in `data/bd_colors.mat`). It is limited up to maximum of 62 colours, where grey colour corresponds to the value `-1` (divergence), and white colour is related to `0` (higher periodic or non-regular behaviour). Note, that if the matrix `C` contains cells with values being larger than `maxperiod`, then they will be also coloured white in the produced plot.

The return values are

- `hfig` being a handle for the plotted figure window,
- and `hcbar` being a handle for the plotted colour bar.

These handles can be used for further customisation of the figure.

3.3.3 Period diagram example

Let us demonstrate an example for computing 2D bifurcation diagrams by using Hénon map (20). Suppose we want to plot a period diagram of (20) for the parameter range $a \in [0.1, 2]$ and $b \in [-0.5, 1]$. To get nice resolution we choose the step sizes $\delta a = 0.01$, $\delta b = 0.005$ and try to find periodic solutions up to period 30. Listing 5 presents a sequence of commands which produces the plot in Fig. 4(a). Three last lines are related to customising the figure: we increase the font size and change ticks for the colour bar.

Listing 5: A sample command list for plotting period diagram.

```
>> init()
>> [param1, param2, periods] = ...
    period_bd_calc(@henon, [0.9 0.8], [0.1 2; -1 1.5], ...
    [0.01 0.005], [1 2], 2000, 30, tolerance, ...
    divergence, 1, 1);
>> [figbd, cbbd] = bd_plot(param1, param2, periods, 30)
>> axbd = get(figbd, 'CurrentAxes')
>> set(axbd, 'FontSize', 24)
>> set(cbbd, 'YTick', (-1:2:29))
```

Moreover, to plot only periodicity regions related to solutions up to period 5 (see Fig. 4(b)) one should use the following command

```
>> [figbd, cbbd] = bd_plot(param1, param2, periods, 5)
```

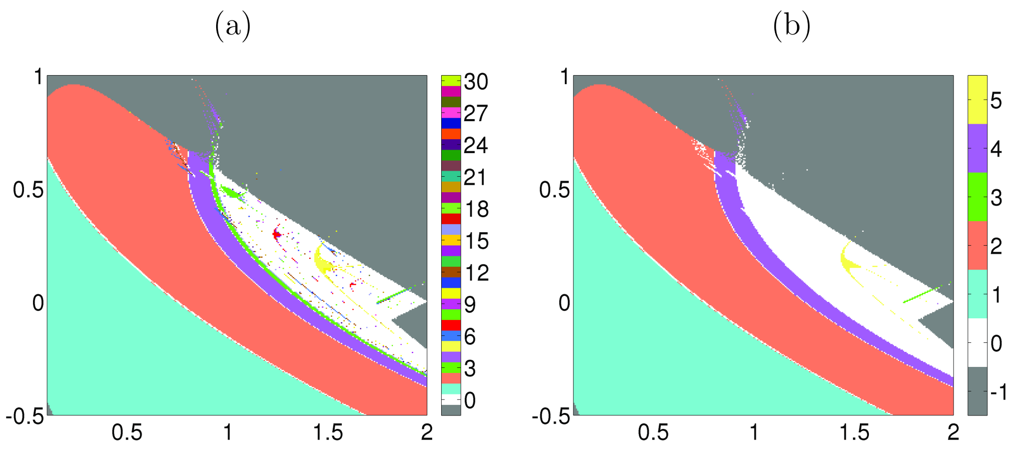

Figure 4: Bifurcation structure of the parameter plane (a, b) of H'enon map (20). Periodicity regions are shown up to (a) period 30; (b) period 5.

3.4 Find periodic points

As it is described in Sec. 2.1, for finding periodic points Newton's method is used. In addition, for multi-dimensional maps a certain backtracking technique is also included to improve global convergence to the root. Therefore, two separate functions `find_periodic_1d` and `find_periodic_md` correspond to finding periodic points of scalar and vector maps, respectively.

3.4.1 One-dimensional maps

The function `find_periodic_1d` for 1D maps has the format

```
function xF = find_periodic_1d(fhandle, period, ...
  interval, init_num, tolerance, max_step, rnd, varargin)
```

The meaning of the parameters is as follows:

- `fhandle` is a function handle corresponding to the desired map.
- `period` is a period of points searched for. Note, that the points with periods being divisors of `period` are also may be found.
- `interval` is usually a two-dimensional vector representing an interval from which the initial conditions are taken. However, when `init_num`

is zero, `interval` should be a single scalar value which is taken as a *predefined* initial condition.

- `init_num` is the number of initial conditions to try. If `init_num` is zero, the initial condition is taken from the parameter `interval`.
- `tolerance` is an accuracy for checking parity of points, namely, if the distance (norm) between the two points x_1 and x_2 is less than `tolerance`, then these points are considered to be equal.
- `max_step` defines the maximal number of Newton steps performed.
- `rnd` defines whether to set initial conditions randomly or not. If being `true` (or `1`), initial conditions are chosen randomly (in the total amount of `init_num`). If being `false` (or `0`), the search interval is divided into `init_num` equal parts, and then a middle point of each subinterval is tried for the initial condition.
- `varargin` represents the parameter vector μ of the iterated map $\mathbf{F}$.

After finishing the calculation process the function `find_periodic_1d` returns the value `xF` which contains a vector of the found (distinct) points sorted in ascending order. This array can be further used as an argument to the function `period_classify` which gives more detailed information about the periodic points found (see Sec. 3.4.3). In particular, it checks whether some of the points belong to the same cycle or not.

3.4.2 Multi-dimensional maps

The function `find_periodic_md` for higher dimensional maps has similar form:

```
function xF = find_periodic_md(fhandle, period, ...
range, init_num, tolerance, max_step, rnd, varargin)
```

We describe only those arguments which are different from the 1D case:

- `range` is similar to `interval` of the function `find_periodic_1d`. However, here it usually is an $n \times 2$ matrix $R = \{r_{ij}\}_{i=1,j=1}^{n,2}$ (n is the dimension of $\mathbf{F}$). Each coordinate x_i^0 of an initial condition vector $\mathbf{x}_0$ is taken from the range $[r_{i1}, r_{i2}]$. However, if `init_num` is zero then `range` should be an n-vector of a *predefined* initial condition.

- Initial conditions can be chosen in three different ways:

    1. Random choice: One should set the parameter `rnd` to be `true` (or `1`) and the parameter `init_num` to be an integer. This integer then defines the total number of initial conditions to try. The initial condition for each coordinate x_i, $i = \overline{1,n}$, is taken randomly from the interval $[r_{i1}, r_{i2}]$ specified by `range`.
    2. Regular grid: One should set `rnd` to be `false` (or `0`) and `init_num` to be an n-dimensional integer vector $\mathbf{N} = (N_1, \ldots, N_n)$. For each coordinate x_i, $i = \overline{1,n}$, the related interval $R_i = [r_{i1}, r_{i2}]$ (specified by `range`) is divided into N_i equally sized subintervals $R_{i1}, \ldots, R_{iN_i}$. Then all possible products $R_{1j_1} \times \ldots \times R_{nj_n}$, $j_k = \overline{1, N_k}$, $k = \overline{1,n}$, are constructed, and centres of these parallelepipeds are taken as initial conditions. So that the total number of initial conditions to be tried is $N_1 \cdot \ldots \cdot N_n$.
    3. Predefined point: One should set `rnd` to be `false` (or `0`) and `init_num` to equal zero. Then `range` should be an n-dimensional real vector $\mathbf{x}_0 = (x_1^0, \ldots, x_n^0)$ which is considered as the initial condition.

The return value of `find_periodic_md` is similar to that of `find_periodic_1d`, however, now it is not a vector but a matrix having n columns corresponding to the coordinates x_i, $i = \overline{1,n}$. The values are sorted in ascending order by the first coordinate (column).

3.4.3 Classifying periodic points

As the functions `find_periodic_1d` and `find_periodic_md` return only a sorted array of points, neither indicating exact periods of them, nor giving the information about their stability, one may need to organise this output. In particular, some of the discovered periodic points may belong to the same cycle, or some cycle points may not appear in the list. For performing further classification and arrangement of the points, one can use the function `period_classify`

```
function points = period_classify(fhandle, xF, ...
  max_period, tolerance, varargin)
```

The function takes as parameters the iterated function handle `fhandle`, the array of points `xF`, the maximal period `max_period` (which usually coincides with `max_period` used in `find_periodic_1d` or `find_periodic_md`). The values `tolerance` and `varargin` are the same as described above. After finishing `period_classify` returns a cell-array `points`. It has the following structure: each row has three cells, where the first cell contains the cycle (or a fixed point) coordinates, the second cell is the exact period of the cycle, and the third cell indicates the stability—`'stable'`, `'unstable'`, or `'saddle'`.

3.4.4 Periodic points example

We demonstrate usage of `find_periodic_md` and `period_classify` by Hénon map (20). Let us find fixed points of H with $a = 1.28$, $b = -0.3$. We set a reasonable search range for periodic points as $\Pi = [-5, 5] \times [-5, 5]$. Calling the command

```
>> xF = find_periodic_md(@henon, 1, [-5 5; -5 5], [10 10],
   tolerance, max_step, false, 1.28, -0.3)
```

we get two fixed points $L_1^{(1)} = (x_1, y_1) \approx (-1.9548, -1.9548)$ and $L_1^{(2)} = (x_2, y_2) \approx (0.6548,\, 0.6548)$. Then we classify these points by

```
>> points = period_classify(@henon, xF, 1, 1e-6, 1.28, -0.3)
```

which announces that both points are saddles. Comparing to the plotted above 2D bifurcation diagram we discover that for these fixed parameter values asymptotic dynamics corresponds to a solution of period 2. Hence we retry finding periodic points but those being of period 2 now:

```
>> xF = find_periodic_md(@henon, 2, [-5 5; -5 5], [10 10],
   tolerance, max_step, false, 1.28, -0.3)
```

Using again `period_classify` (putting `max_period` $= 2$) we get the following result

```
points =
    [1x2 double]    [1]    'saddle'
    [2x2 double]    [2]    'stable'
    [1x2 double]    [1]    'saddle'
```

A new finding here is a 2-cycle $L_2 = \{(x_{c1}, y_{c1}), (x_{c2}, y_{c2})\}$ with $x_{c1} \approx 0.538, y_{c1} \approx 0.762, x_{c2} = y_{c1}, y_{c2} = x_{c1}$. Note that together with a 2-cycle L_2

the routine finds also two saddle fixed points $L_1^{(1)}$ and $L_1^{(2)}$ mentioned above.

3.5 Invariant manifolds

3.5.1 Unstable manifolds

To compute the unstable manifold of a fixed point or a cycle the functions `sc_unstable` or `sc_unstable_cyc` should be used, respectively.

```
function [p, arc_length] = sc_unstable(fhandle, p0, ...
  arc_max, d, a_min, a_max, Da_min, Da_max, D_min, ...
  cB, varargin)
function [p, arc_length] = sc_unstable_cyc(fhandle, ...
  p0, period, arc_max, d, a_min, a_max, Da_min, ...
  Da_max, D_min, cB, varargin)
```

The second one differs from the first one by a single parameter `period` which indicates the exact period of the point `p0`. The other parameters are:

- `fhandle` is an iterated function handle.
- `p0` is a saddle point whose manifold is to be computed.
- `arc_max` is a desired length of the manifold arc (is approximated as a sum of vector norms $\|p_{k+1} - p_k\|$, where $\{p_k\}_{k=0}^{N}$ are computed points of the manifold, see Sec. 2.2.2).
- `d` is an initial step size. Note that if `d` > 0 manifold is grown forward (in the direction of the related eigenvector), while if `d` < 0 it is grown backward (in the direction opposite to the related eigenvector).
- `a_min` is the minimal value for the angle between the two successive manifold segments (see Fig. 2). If the angle α_k between the vectors (p_{k-1}, p_k) and (p_k, p_{k+1}) falls below this value, the step size is increased.
- `a_max` is the maximal angle value. The angle α_k between the vectors (p_{k-1}, p_k) and (p_k, p_{k+1}) is assumed to not exceed this threshold (for possible exceptions see description of the method in Sec. 2.2.2).
- `Da_min` is the minimal value for the product $\Delta_k \alpha_k$.
- `Da_max` is the maximal value for the product $\Delta_k \alpha_k$.

- `D_min` is the minimal step size.
- `cB` is an accuracy coefficient for bisection method, it must be positive and less than one.

The function returns two values: an array of computed manifold points `p` and a computed manifold length `arc_length`.

Note, that only *one-dimensional* unstable manifolds can be calculated.

3.5.2 Stable manifolds

Similarly, to compute the stable manifold of a fixed point or a cycle the functions `sc_stable` or `sc_stable_cyc` should be used, respectively.

```
function [p, arc_length] = sc_stable(fhandle, p0, ...
  arc_max, max_iter, d, a_min, a_max, Da_min, Da_max, ...
  D_min, eB, varargin)
function [p, arc_length] = sc_stable_cyc(fhandle, p0, ...
  period, arc_max, max_iter, d, a_min, a_max, Da_min, ...
  Da_max, D_min, eB, varargin)
```

These two functions take almost the same parameters as those for computing unstable manifolds. We describe only parameters which differ:

- `max_iter` is a maximal number of iterates to try. This parameter is useful in case of a non-invertible map, which may have several preimages. The stable manifold of such a map may be mapped onto itself in a complex way, so that it is required to iterate the map several times until getting the right approximation for the next point.
- `eB` is an error for the bisection method used during search circle procedure.

The return values are the same as in case of unstable manifold computation.

3.5.3 Plotting manifolds

There is one more useful function `plot_manifolds` which allows to plot computed manifolds together with the periodic points found. It has the format

```
function hfig = plot_manifolds(points, umanif, smanif)
```

Its parameters are

- `points` is the cell array of periodic points, which should be of the same structure as one obtained from the functions `find_periodic_1d` and `find_periodic_md`.
- `umanif` is the cell array of unstable manifolds to be plotted (may be empty `{}`). They are plotted in red.
- `smanif` is the cell array of stable manifolds to be plotted (may be empty `{}`). They are plotted in blue.

The return value is a handle `hfig` for the plotted figure window, which may be used for further customisation of the plot.

3.5.4 Invariant manifolds example

As before we use Hénon map (20) for demonstration. We fix parameter values as in Sec. 3.4.3. Recall that there exists at least two saddle fixed points $L_1^{(1)} = (x_1, y_1)$ and $L_1^{(2)} = (x_2, y_2)$ together with a stable 2-cycle $L_2 = \{(x_{c1}, y_{c1}), (x_{c2}, y_{c2})\}$ (see Sec. 3.4.4). We make the assignments

```
>> fp1 = points{1, 1} % fixed point (x_1, y_1)
>> fp2 = points{3, 1} % fixed point (x_2, y_2)
>> c21 = points{2, 1}(1, :) % cycle point (x_{c1}, y_{c1})
>> c22 = points{2, 1}(2, :) % cycle point (x_{c2}, y_{c2})
```

To compute the manifolds of the first fixed point we call two commands (computing forward and backward):

```
>> [um_fp1_f, arc] = sc_unstable(@henon, fp1, arc_length, d,
   a_min, a_max, Da_min, Da_max, D_min, cB, 1.28, -0.3);
>> [um_fp1_b, arc] = sc_unstable(@henon, fp1, arc_length, -d,
    a_min, a_max, Da_min, Da_max, D_min, cB, 1.28, -0.3);
```

Both commands run with success producing two branches of the unstable manifold of $L_1^{(1)}$. However when we do the same for $L_1^{(2)}$ we get the following message

```
WARNING: Unable to find next point, aborting computation.
Manifold is computed up to 5 point, arc length = 0.00110769
```

Similar message we get when trying to compute the manifold backwards. Making some additional investigation we discover that at some point the unstable manifold of $L_1^{(2)}$ becomes rather 'slow', that is, the distance between two successive manifold points becomes very small and less than `D_min`. Replacing `D_min` with `D_min/100` improves things and both branches of the manifold are computed. In Fig. 5 we show computed unstable manifolds of both fixed points, $L_1^{(1)}$ and $L_1^{(2)}$ (cyan triangles), together with a stable 2-cycle L_2 (orange circles). Plot (b) is the enlargement of the box indicated in plot (a), and one can see that unstable manifold of $L_1^{(2)}$ is in fact a part of the stable manifold of the 2-cycle L_2.

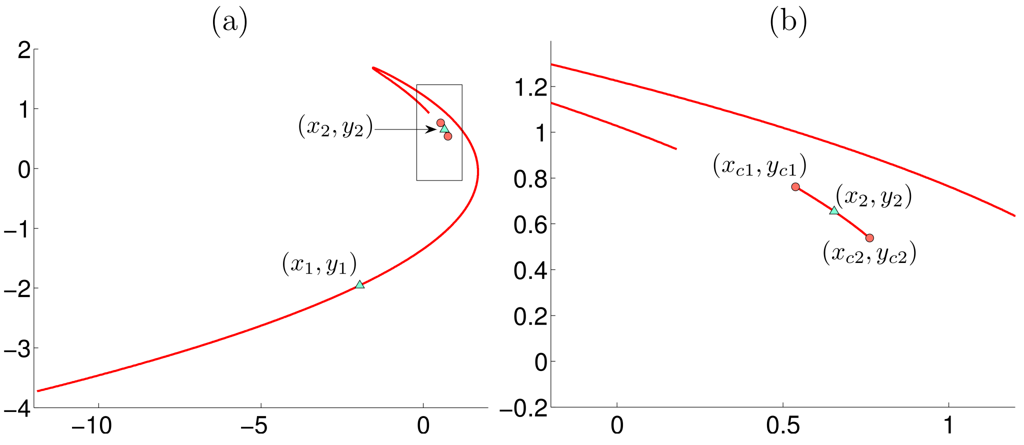

Figure 5: Unstable manifolds of the fixed points (x_1, y_1) and (x_2, y_2) of Hénon map (20). (b) is the enlargement of the window shown in (a).

Let us turn now to stable manifolds of both fixed points. We compute both branches of stable manifold for $L_1^{(1)}$ (similarly for $L_1^{(2)}$) by

```
>> [sm_fp1_f, arc] = sc_stable(@henon, fp1, 30, 1, d, a_min,
   a_max, Da_min, Da_max, D_min, eB, 1.28, -0.3);
>> [sm_fp1_b, arc] = sc_stable(@henon, fp1, 30, 1, -d, a_min,
    a_max, Da_min, Da_max, D_min, eB, 1.28, -0.3);
```

Note that since Hénon map is invertible (provided that $b \neq 0$) it is enough to put `max_iter` $= 1$. A couple of examples for handling non-invertible maps are included in the CompDTIMe distribution.

The resulting plot can be seen in Fig. 6. This plot is obtained by using `plot_manifolds`, namely

```
>> hfig = plot_manifolds(points, {um_fp1_f, um_fp1_b,
   um_fp2_f, um_fp2_b}, {sm_fp1_f, sm_fp1_b, sm_fp2_f,
   sm_fp2_b});
```

The figure handle `hfig` is then used to customise fonts, labels, and arrows, similarly to how it is done in example presented in Sec. 3.3.3.

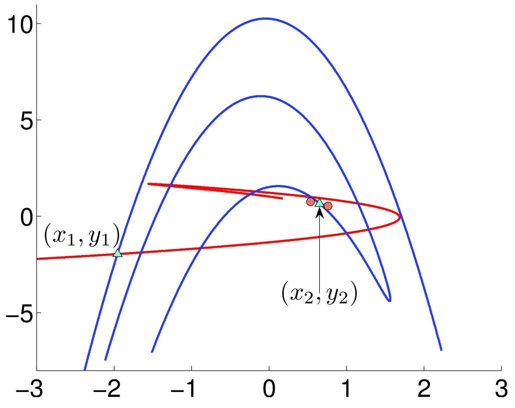

Figure 6: Stable and unstable manifolds of the fixed points $L_1^{(1)}$ and $L_1^{(2)}$ of Hénon map (20).

4 Conclusion

In this paper we described main functionalities of CompDTIMe developed for investigating discrete time dynamical systems. The package consists of several Matlab routines which allow one to calculate data for two-dimensional bifurcation diagrams (period diagrams) in the parameter space of a map; to find periodic points of a predefined period and to classify these points (determine their type, group the points belonging to a single cycle, *etc.*); to compute one-dimensional stable and unstable manifolds of a saddle periodic point. CompDTIMe does not have any graphical user interface, but several routines are provided to plot the numerical outcome (such as period diagrams, periodic points including stability information, invariant manifolds) by means of Matlab.

Numerical methods used by CompDTIMe are also briefly presented. In particular, for finding periodic points Newton's (Newton-Raphson) method is chosen as being one of the most efficient. Computation of one-dimensional invariant manifolds is performed by using the so-called Search Circle algorithm, which differs from other similar methods by the fact that one does not need to calculate the inverse map. This provides an efficient technique for obtaining the stable invariant manifold even in case when the map is not invertible.

For the moment the package CompDTIMe is designed to be run under Matlab environment, but it is planned to translate it to QB64 and C++ (the work is now in progress).

A Global parameters for numerical methods

Variable	Value	Description
`divergence`	`100000`	limit for detecting divergence to infinity
`tolerance`	`1e-8`	accuracy for checking parity of points
`max_step`	`50`	maximal number of Newton steps
`arc_length`	`10`	desired length of the manifold arc
`d`	`0.001`	initial step for manifold calculation
`a_min`	`0.2`	search circle minimal angle value
`a_max`	`0.3`	search circle maximal angle value
`Da_min`	`1e-6`	minimal value for the product $\Delta_k\alpha$
`Da_max`	`1e-5`	maximal value for the product $\Delta_k\alpha$
`D_min`	`1e-4`	search circle minimal step size
`cB`	`0.2`	bisection accuracy coefficient
`eB`	`1e-6`	search circle bisection error

Table 1: List of predefined parameters which control running numerical methods.