Bifurcation Diagrams based on NichePSO and Its Fixed Points Qualification

A DISSERTATION PRESENTED
BY
OSCAR VARGAS TORRES
TO
THE DIVISIÓN DE ESTUDIOS DE POSGRADO DE LA FACULTAD DE INGENIERÍA ELÉCTRICA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER IN SCIENCE
IN THE SUBJECT OF
COMPUTER SCIENCE

DR. JUAN JOSÉ FLORES ROMERO
THESIS ADVISOR

DR. JAIME CERDA JACOBO
THESIS COADVISOR

UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO
MORELIA, MICHOACÁN
AUGUST 2013

Thesis advisor: Dr. Juan José Flores Romero Oscar Vargas Torres

Bifurcation Diagrams based on NichePSO and Its Fixed Points Qualification

Abstract

An approach to find fixed points of dynamical systems based on Niche PSO is studied. Important parameters of the Niche PSO algorithms are highlighted: 1) the number of particles in the main swarm, 2) the merging strategy for subswarms and 3) the stopping condition.

Stability qualification of fixed points is done with linearization whenever it is appropriate (using eigenvalues). For hyperbolic fixed points, additional information is available: the nonlinear flow resembles that of its conjugate linear flow.

A description of Bidiatool, a Bifurcation diagram tool for bifurcation diagram plotting is given in broad terms, highlighting examples of testing (even in the Graphical User Interface context) and high-level of abstraction to get a better overall design.

A comparison between more traditional approaches and this (based on computational intelligence algorithms) is made in the Results chapter.

Contents

List of Figures

List of Algorithms

To my son Alex

Acknowledgments

I want to thank Jehovah God, for allowing me to study this far.

I am grateful to my teachers, specially Dr. Juan José Flores Romero and Dr. Jaime Cerda Jacobo for supervision.

I thank my family for their patience and unconditional support.

The advanced reader who skips parts that appear too elementary may miss more than less advanced reader who skips parts that appear too complex.

G. Pólya

1 Introduction

Many research areas are interested in bifurcation analysis of dynamical systems. Bifurcation diagrams plotting is a very important task in this kind of analysis because it helps to qualitatively predict complex behaviors in the structure of a system where there is variation in its parameters.

Common bifurcation diagrams plotting methods, require of initial values and parameter adjustment for its correct operation. These values are frequently unknown, and requiere a deep knowledge of the system being analyzed or a non-systematic search of these parameters.

As an alternative to these methods, metaheuristic methods are used as a complementary tool in bifurcation diagrams plotting. This kind of methods present a number of advantages and disadvantages over traditional methods.

The intention of this project is the implementation of software that produces bifurcation diagrams using Niche PSO, as well as the qualification of fixed points.

1.1 Justification

Advantages:

- Drawing bifurcation diagrams with little previous knowledge of the behavior of the dynamical system.
- Generation of bifurcation diagrams in a unsupervised way.
- The ability to work with non-differentiable functions.
- Great portability because of the usage of the Java Virtual Machine (JVM).

1.2 State of the Art

[16] gives a (necessarily) incomplete list of software devoted to the study of dynamical systems and bifurcation problems. Amongst the most prominent is AUTO [4].

AUTO has become a standard package in bifurcation analysis. A complete and up-to-date installation of AUTO (at the moment of writing) uses Fortran, Python and LaTeX and transfig (for the documentation). Even when AUTO has a lot of features, the user of this software has the following challenges:

- It requires a decent amount of knowledge to install everything properly in different platforms (portability is an issue here).
- The user can write some dynamical system specification using Fortran or C, and then it is possible to use Python to get some scripting functionality. Interactivity is gained through the AUTO CLUI (based on the Python REPL) and the Unix command line. The user then has to learn the basics of Fortran/C to describe the dynamical system, and choose between Python or Unix interactivity (and therefore learn additional programming or a lot of custom unix AUTO commands). Usability is an issue here.
- Documentation is distributed in PDF form, but some of them requires an installation of a TeX distribution *and* transfig. HTML documentation is non-existent. Again, portability and usability concerns.
- A quite esoteric file naming convention is used.

XPP/XPPAUTO contains the code for AUTO, and makes its usage easier. However, it doesn't expose all the features AUTO has (it targets more platforms and has to keep up with the latest developments of AUTO). Someone has to compile the source code for every platform to distribute it in binary form.

PyDSTool provides another alternative for bifurcation and stability analysis. Again, if the user wants a feature-complete installation, he/she will face similar challenges as mentioned before for AUTO (to use the C–based integrator and the AUTO support). Furthermore, the user has to deal with a 32 bit installation, even when using 64 bit systems. For example, to install it on a 64 bit Mac, one possibility is:

- Partition the hard disk to make a dual installation of Mac OS X and a 64 bit Linux distribution (e.g. Ubuntu, to mention a common distribution).

- Use debootstrap and schroot (with the proper configuration) to install *every* dependency (gfortran, gcc, python, numpy, scipy, matplotlib, etc.) in a 32 bit flavor.

Although installing PyDSTool is harder than installing AUTO, it provides a more uniform programming environment (Python scripting only) and is therefore more usable. The pain of a correct installation is a price the user has to pay to get all the features promised by the aforementioned software.

PyDSTool has been used to generate the same bifurcation diagrams that are reported in this work with Niche PSO. Therefore, we have some way to compare our proposal.

1.3 Hypothesis

Generating bifurcation diagrams with traditional methods requires initial values and suitable configuration of parameters. Using Niche PSO (and other niching or speciation techniques) can guide the researcher to provide better initial values in a more systematic way. Furthermore, Niche PSO is an alternative to generate these bifurcation diagrams.

1.4 Objectives

The purpose of this thesis is the implementation of a system for drawing bifurcation diagrams using Niche PSO. The developed system must meet the following minimum requirements:

- To allow the definition of dynamical systems.

- Generate bifurcation diagrams in 2D and 3D using Niche PSO.

- To be able to qualify the stability of the fixed points found.

- Run on the Java Virtual Machine

1.5 Thesis Contents

The rest of this thesis is organized as follows:

- Chapter 2 introduces dynamical systems terminology and stablishes the criteria used for classification of fixed points. It also describes a grammar to parse dynamical systems with certain characteristics.
- Chapter 3 introduces PSO and then introduces the special variant Niche PSO that was developed to locate and maitain multiple solutions. A description of the configuration of the algorithm is given. The merge strategy is of crucial importance to avoid excessive merging of niches (with the consequent loss of potential solutions). Next, it explains how Niche PSO can be used to produce bifurcation diagrams.
- Chapter 4 describes the process of plotting a bifurcation with Bidiatool (a bifurcation diagram tool, the product of this work). It also explains some non-trivial software techniques that were exploited in this work.
- Chapter 5 presents some results of this work. It also compares bifurcation diagrams produced with Bidiatool and PyDSTool.
- Chapter 6 summarizes the main achievements of this thesis and suggests further work.

Mathematics is the language in which God has written the universe.

Galileo Galilei

2
Dynamical Systems and its Parsing

In this chapter basic dynamical system terminology (Sections 2.1–2.4) and the qualification criteria of fixed points are stablished (Section 2.7).

- Section 2.5 states the noteworthy Theorem 1 that constitutes the basis of our qualification of fixed points criteria; it also makes a summary of the different linear systems that could arise after linearization (this "catalog" serves as a basis to test the implementation of the qualification criteria).
- Section 2.6 is an illustration of the application of some concepts given in Section 2.5 in a classical problem.
- Section 2.8 summarizes two numerical algorithms used for qualification of fixed points.

Finally, Section 2.9 explains the basic theory behind a proposed implementation of a parser (based on Parsing Expression Grammars and Combinator Parsers) for dynamical systems. The clear separation between the Abstract Syntax Tree construction and it's evaluation means the parser as a whole is a high-level interpreter for dynamical systems.

2.1 Systems of Differential Equations

A *system of differential equations* is a collection of n interrelated differential equations:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(t, \mathbf{x}(t))$$

This equation stands for a system consisting of n scalar components,

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} f_1(t, x_1, \ldots .x_n) \\ f_2(t, x_1, \ldots .x_n) \\ \vdots \\ f_n(t, x_1, \ldots .x_n) \end{bmatrix}$$

An equation system of the type,

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$$

in which the independent variable t does not occur explicitly, is called *autonomous* [16] . Autonomous systems have the feature that whenever $\mathbf{x}(t)$ is a solution then $\mathbf{x}(t+c)$ is also a solution for all c. Consequently initial values

$$\mathbf{x}(t_0) = \mathbf{x}_0$$

can be assumed for $t = 0$. A non–autonomous equation can be transformed into an extended autonomous system by means of

$$x_{n+1} = t$$

This leads to a system with $n + 1$ components,

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \\ \dot{x}_{n+1} \end{bmatrix} = \begin{bmatrix} f_1(t, x_1, \ldots .x_n) \\ f_2(t, x_1, \ldots .x_n) \\ \vdots \\ f_n(t, x_1, \ldots .x_n) \\ 1 \end{bmatrix}$$

A vector $\mathbf{x}_0$ for which $\mathbf{f}(\mathbf{x}_0) = 0$ is called *equilibrium point*. In equilibrium points, the system is at rest; equilibrium solutions are constant solutions. These points are also called stationary solutions, and sometimes singular points, critical points, rest points, or fixed points.

2.2 Dynamical Systems: basic terminology

Let's use the one–dimensional system

$$\dot{x} = f(x, \lambda) \tag{2.1}$$

and

$$0 = f(x, \lambda) \tag{2.2}$$

as tools to introduce some terminology. We will assume throughout that λ is a real parameter. Clearly, solutions $\mathbf{x}$ of equation (2.1) or (2.2) in general vary with λ. For more information about the used terms, see [18], [17], [16].

The graph of $f(x, \lambda)$ (for a fixed λ) versus x can be used in order to understand the *vector field* on the line. The *trajectory* $x(t)$ of *phase point* based at x_0 represents the solution of the differential equation starting from the initial condition x_0. A picture that shows the different trajectories of the system is called *phase portrait*.

Let's take as an example the one–dimensional system

$$\dot{x} = x^2 + \lambda \tag{2.3}$$

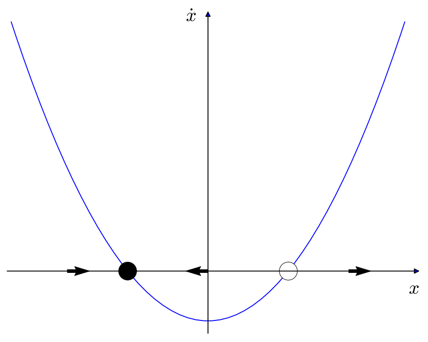
Figure 2.2.1 shows the phase portraits corresponding to several values of λ. Solid black dots represent *stable* equilibrium points (often called attractors or sinks, because the flow is toward them); open circles represent *unstable* equilibrium points (also known as repellers or sources); half–filled circles represent *half–stable* equilibrium points.

2.3 Eigenvalues and Eigenvectors

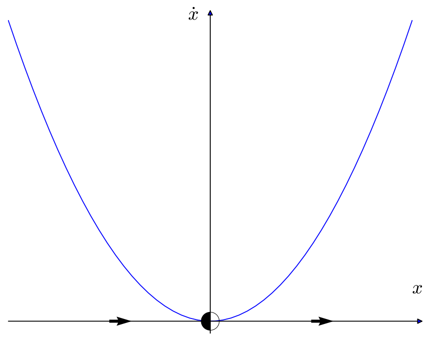
Suppose A is a square matrix of size 2×2, $\mathbf{x} \neq 0$ is a vector in $\mathbb{R}^2$. Then the equation $\mathbf{y} = A\mathbf{x}$ represents a pair of scalar equations

$$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 \\ y_2 &= a_{21}x_1 + a_{22}x_2 \end{aligned}$$

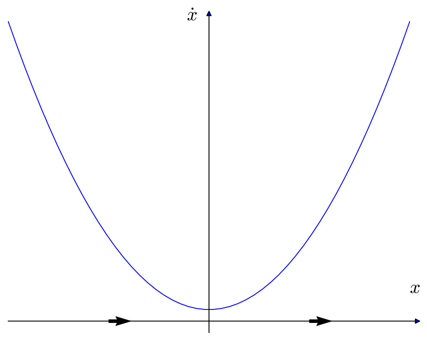
As these equations show, generally the matrix A mixes up the components of $\mathbf{x}$. In more complex situa-



(a) $\lambda < 0$



(b) $\lambda = 0$



(c) $\lambda > 0$

Figure 2.2.1: Phase portrait for Equation (2.3)

tions (for example, in higher dimensions), this mixture can be quite involved. However, it is often possible to find a new basis $\mathcal{S} = \{\mathbf{s}_1, \mathbf{s}_2\}$ for $\mathbb{R}^2$ in which such mixing does not occur. That is, a basis in which the components are decoupled and develop separately as

$$y_{S1} = \lambda_1 x_{S1}$$
$$y_{S2} = \lambda_2 x_{S2}$$

where λ_1 and λ_2 are appropiate scalars depending on the matrix A (note that the ordered tuple $S = (\mathbf{s}_1, \mathbf{s}_2)$ constitutes an *ordered basis* for $\mathbb{R}^2$, and subindices $S1$ and $S2$ refer to this order). This could lead to greatly simplified computations.

Definition 1 (Eigenvalues and eigenvectors) *Given any $n \times n$ matrix A, if for a scalar λ and a* nonzero *vector* $\mathbf{v}$ *the equation*

$$A\mathbf{v} = \lambda\mathbf{v} \tag{2.4}$$

holds, then λ is called an eigenvalue *of the matrix A and* $\mathbf{v}$ *an* eigenvector *of A corresponding or belonging to λ. For any eigenvalue λ the zero vector is always a solution of (2.4) and is called the trivial eigenvector of A belonging to λ.*

Equation (2.4) can be rewritten as

$$(A - \lambda I)\mathbf{v} = 0 \tag{2.5}$$

For any fixed λ, a homogeneous equation like this has nontrivial solutions if and only if its matrix is singular. It follows that λ is an eigenvalue of A if and only if λ is a root of the *characteristic equation*

$$\det(A - \lambda I) = 0 \tag{2.6}$$

To find the eigenvectors we substitute the eigenvalues resulting from (2.6), one after the other, into (2.5), and solve for the unknown vector $\mathbf{v}$.

2.4 Stability

Suppose $\mathbf{f}(\mathbf{y})\colon \mathbf{R}^n \to \mathbf{R}^n$, and $\mathbf{f} \in \mathcal{C}^1$ (it is continuously differentiable in all $\mathbf{y}$–space). The usual stability definitions given next are for the initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \mathbf{y}(0) = \mathbf{p}$.

The system is *stable* at an equilibrium point $\mathbf{y}_0$, if for each positive number ε there is a positive number δ such that for all solutions $\mathbf{y}(t, \mathbf{p})$, if $\|\mathbf{p} - \mathbf{y}_0\|_2 < \delta$, then $\|\mathbf{y}(t, \mathbf{p}) - \mathbf{y}_0\|_2 < \varepsilon$, for all $t \geq 0$.

The system is *asymptotically stable* at the equilibrium point if it is stable *and* if $\|\mathbf{y}(t,\mathbf{p}) - \mathbf{y}_0\| \to 0$ as $t \to +\infty$ for all points $\mathbf{p}$ near $\mathbf{y}_0$.

The system is *neutrally stable* at $\mathbf{y}_0$ if it is stable at $\mathbf{y}_0$, but not asymptotically stable. Finally, the system is *unstable* at $\mathbf{y}_0$ if it is not stable.

The stability properties of a autonomous linear system $\dot{\mathbf{y}} = A\mathbf{y}$ can be completely determined by a study of the eigenvalues of A and its corresponding eigenspaces. For nonlinear autonomous systems, we could determine the stability of the dynamical system at the fixed points via linearization or with Liapunov functions [1]. In the present work, we classify the stability of fixed points only if linearization is suitable. Then, if $\mathbf{f} \in C^2$, $\mathbf{y}_0$ is a fixed point of $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, and $\mathbf{f}^0_{\mathbf{y}}$ is the Jacobian of $\mathbf{f}$ evaluated at $\mathbf{y}_0$, then the system $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$,

1. is *asymptotically stable* at $\mathbf{y}_0$ if all eigenvalues of $\mathbf{f}^0_{\mathbf{y}}$ have negative real parts.

2. is unstable at $\mathbf{y}_0$ if $\mathbf{f}^0_{\mathbf{y}}$ has at least one eigenvalue with a positive real part.

We cannot draw any conclusion about stability (using linearization) if $\mathbf{f}^0_{\mathbf{y}}$ has an eigenvalue with a zero real part, but no eigenvalue with a positive real part (non–linear order terms determine the stability).

However, if $\mathbf{y}_0$ is hyperbolic (none of its eigenvalues has real part zero), the nonlinear flow near this fixed point is conjugate to the flow of the linearized system (the nonlinear flow resembles that of the linearized system near $\mathbf{y}_0$).

A sink is asymptotically stable and therefore stable. Sources and saddles are examples of unstable equilibria.

2.5 Linear Systems and its relation to Nonlinear Systems

Definition 2 *A matrix A is hyperbolic if none of its eigenvalues has real part 0. We also say that the system $\dot{\mathbf{x}} = A\mathbf{x}$ is hyperbolic.*

Let $\mathbf{f}^0_{\mathbf{x}}$ denote the *Jacobian* matrix of $\mathbf{f}$ evaluated at the fixed point $\mathbf{x}_0$. Then the system

$$\dot{\mathbf{y}} = \mathbf{f}^0_{\mathbf{x}}\mathbf{y} \tag{2.7}$$

where $\mathbf{y} = \mathbf{x} - \mathbf{x}_0$, is called the *linearized system* near $\mathbf{x}_0$.

These concepts are important because of [17]:

[1]Liapunov defined classes of scalar functions to test for the stability, asymptotic stability, or instability of any system, linear or nonlinear, autonomous or nonautonomous.

Theorem 1 (The Linearization Theorem) *Suppose the n–dimensional system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ has an equilibrium point at $\mathbf{x}_0$ that is hyperbolic. Then the nonlinear flow is conjugate to the flow of the linearized system in a neighborhood of $\mathbf{x}_0$.*

In a more intuitive way, this means that near the hyperbolic equilibrium point, the flow of the nonlinear system has the same fate than the linear one. Then, an understanding of linear systems is useful to study nonlinear phenomena.

2.5.1 Linear Planar Systems (autonomous)

In the case of an autonomous linear system $\dot{\mathbf{x}} = A\mathbf{x}$, the coefficient matrix is

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

where a, b, c and d are constants. A can be classified according to its eigenvalues. There are three possibilities:

- Real distinct eigenvalues
- Complex eigenvalues
- Repeated eigenvalues

Real distinct eigenvalues

Suppose the matrix A has two real, distinct eigenvalues λ_1 and λ_2 with associated eigenvectors $\mathbf{s}_1$ and $\mathbf{s}_2$. Let T be the matrix whose columns are $\mathbf{s}_1$ and $\mathbf{s}_2$. Then, instead of considering the linear system $\dot{\mathbf{x}} = A\mathbf{x}$, we consider the system

$$\dot{\mathbf{y}} = (T^{-1}AT)Y$$

that assumes the canonical form

$$\dot{\mathbf{y}} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \mathbf{y} \tag{2.8}$$

Note that if $\mathbf{y}(t)$ is a solution of this new system, then $\mathbf{x}(t) = T\mathbf{y}(t)$ solves $\dot{\mathbf{x}} = A\mathbf{x}$. Equation (2.8) has the general solution

$$\mathbf{y}(t) = \alpha e^{\lambda_1 t} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta e^{\lambda_2 t} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Assuming that $\lambda_i \neq 0$, there are three cases to consider:

1. $\lambda_1 < 0 < \lambda_2$. The equilibrium point is called a *saddle*.
2. $\lambda_1 < \lambda_2 < 0$. The equilibrium point is called a *sink*.
3. $0 < \lambda_1 < \lambda_2$. The equilibrium point is called a *source*.

Complex eigenvalues

Now suppose that the matrix A has complex eigenvalues $\alpha \pm i\beta$ with $\beta \neq 0$. Then we may find a complex eigenvector $\mathbf{v}_1 + i\mathbf{v}_2$ corresponding to $\alpha + i\beta$, where both $\mathbf{v}_1$ and $\mathbf{v}_2$ are real vectors. Let T be the matrix whose columns are $\mathbf{v}_1$ and $\mathbf{v}_2$. The system

$$\dot{\mathbf{y}} = (T^{-1}AT)Y$$

assumes the canonical form

$$\dot{\mathbf{y}} = \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix} \mathbf{y} \tag{2.9}$$

and has a phase portrait corresponding to a *spiral sink*, *center*, or *spiral source* depending on whether $\alpha < 0$, $\alpha = 0$, or $\alpha > 0$

Repeated eigenvalues

Suppose A has a single real eigenvalue λ. If there exist a pair of linearly independent eigenvectors, then in fact A must be

$$A = \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix} \tag{2.10}$$

and solutions are of the form

$$\mathbf{x}(t) = \alpha e^{\lambda t}\mathbf{v}$$

where $\mathbf{v}$ is any non–zero vector in $\mathbb{R}^2$.

On the other hand, suppose $\mathbf{v}$ is an eigenvector and that every other eigenvector is a multiple of $\mathbf{v}$. Let $\mathbf{w}$ be any vector for which $\mathbf{v}$ and $\mathbf{w}$ are linearly independent. Then

$$A\mathbf{w} = \mu\mathbf{v} + \lambda\mathbf{w}$$

for some $\mu \neq 0$. Let $\mathbf{u} = (1/\mu)\mathbf{w}$. Then

$$A\mathbf{u} = \mathbf{v} + \lambda\mathbf{u}$$

Let T be the matrix whose columns are $\mathbf{v}$ and $\mathbf{u}$. The system

$$\dot{\mathbf{y}} = (T^{-1}AT)Y$$

assumes the canonical form

$$\dot{\mathbf{y}} = \begin{bmatrix} \lambda & 1 \\ 0 & \lambda \end{bmatrix} \mathbf{y} \tag{2.11}$$

The general solution of (2.11) may be written as

$$\mathbf{y}(t) = \alpha e^{\lambda t} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta e^{\lambda t} \begin{bmatrix} t \\ 1 \end{bmatrix}$$

2.5.2 Higher Dimensional Linear Systems

Real Distinct Eigenvalues

Suppose A is an $n \times n$ matrix with *real,* distinct eigenvalues. Then there is a matrix T such that

$$TA^{-1}T = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \tag{2.12}$$

where all of the entries off the diagonal are 0. T is the matrix whose columns are the eigenvectors associated to the eigenvalues of A.

The system $\dot{\mathbf{y}} = T^{-1}AT\mathbf{y}$ assumes the simple form

$$\begin{aligned} \dot{y}_1 &= \lambda_1 y_1 \\ &\vdots \\ \dot{y}_n &= \lambda_n y_n \end{aligned} \tag{2.13}$$

A function of the form

$$\mathbf{y}(t) = \begin{bmatrix} c_1 e^{\lambda_1 t} \\ \vdots \\ c_n e^{\lambda_n t} \end{bmatrix} \tag{2.14}$$

is a solution of $\dot{\mathbf{y}} = T^{-1}AT\mathbf{y}$ that satisfies the initial condition $\mathbf{y}(0) = [c_1, \dots, c_n]^T$. $\mathbf{x} = T\mathbf{y}$ is the general solution of $\dot{\mathbf{x}} = A\mathbf{x}$, that can be written in the form

$$\mathbf{x}(t) = \sum_{j=1}^{n} c_j e^{\lambda_j t}\mathbf{v}_j \tag{2.15}$$

where $\mathbf{v}_j$ is the eigenvector associated to λ_j

Now suppose the system is hyperbolic (none of the real eigenvalues is zero).

- If every eigenvalue is negative, we have a higher dimensional *sink*.
- If some eigenvalues are negatives and some others are positive, we have a higher dimensional *saddle*.
- If every eigenvalue is positive, we have a higher dimensional *source*.

Distinct eigenvalues (real and complex)

Suppose that the $n \times n$ matrix A has n distinct eigenvalues, of which k_1 are real and $2k_2$ are complex, so that $n = k_1 + 2k_2$. Then there exist a linear map T so that

$$T^{-1}AT = \begin{bmatrix} \lambda_1 & & & & & \\ & \ddots & & & & \\ & & \lambda_{k_1} & & & \\ & & & D_1 & & \\ & & & & \ddots & \\ & & & & & D_{k_2} \end{bmatrix} \tag{2.16}$$

where the D_j are 2×2 matrices of the form

$$D_j = \begin{bmatrix} \alpha_j & \beta_j \\ -\beta_j & \alpha_j \end{bmatrix} \tag{2.17}$$

For the k_1 real eigenvalues, let $\mathbf{v}_j$ be an eigenvector associated to λ_j. The first columns of T are formed with $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_{k_1}$. For the $2k_2$ complex eigenvalues, let $\mathbf{v}_l$ and $\overline{\mathbf{v}}_l$, $1 \le l \le k_2$, be the associated eigenvectors associated to $\alpha_l \pm \beta_l$. Let

$$\begin{aligned} \mathbf{w}_{2l-1} &= \frac{1}{2}(\mathbf{v}_l + \overline{\mathbf{v}}_l) \quad \text{a \textit{real} vector, the real part of } \mathbf{v}_l \\ \mathbf{w}_{2l} &= -\frac{i}{2}(\mathbf{v}_l - \overline{\mathbf{v}}_l) \quad \text{a \textit{real} vector, the imaginary part of } \mathbf{v}_l \end{aligned}$$

The rest of the columns of T are $\mathbf{w}_1, \ldots, \mathbf{w}_{2k_2}$

Repeated Eigenvalues

Let A be an $n \times n$ matrix with repeated eigenvalues. Then there is a change of coordinates T for which

$$T^{-1}AT = \begin{bmatrix} B_1 & & \\ & \ddots & \\ & & B_k \end{bmatrix} \tag{2.18}$$

where each of the B_j's is a square matrix (and all other entries are zero) of one of the following forms:

$$\begin{bmatrix} \lambda & 1 & & & \\ & \lambda & 1 & & \\ & & \ddots & \ddots & \\ & & & \ddots & 1 \\ & & & & \lambda \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} C_2 & I_2 & & & \\ & C_2 & I_2 & & \\ & & \ddots & \ddots & \\ & & & \ddots & I_2 \\ & & & & C_2 \end{bmatrix} \tag{2.19}$$

where

$$C_2 = \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix} \quad \text{and} \quad I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

where $\alpha, \beta, \lambda \in \mathbb{R}$ with $\beta \neq 0$. The special cases where $B_j = (\lambda)$ or

$$B_j = \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix}$$

are, of course, allowed.

2.6 A damped, undriven Hooke's Law Spring

As an example of a linear sytem we will analyze the vertical motion of a weight attached to a driven Hooke's Law spring. Out model is the linear ODE

$$a_2\ddot{x} + a_1\dot{x} + a_0x = f(t) \tag{2.20}$$

where a_0 and a_2 are positive constants, a_1 is nonnegative, and $f(t)$ is the driving force. The voltage and current in an electrical RLC circuit are modeled by an ODE that looks like (2.20). For the undriven ODE, $f(t) = 0$. Dividing (2.20) by a_2, we have:

$$\ddot{x} + 2c\dot{x} + \omega_0^2 x = 0 \tag{2.21}$$

where $a_1/a_2 = 2c$, $a_0/a_2 = \omega_0^2$. ODE (2.21) can be rewritten as a system of first–order ODEs in normal form:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -\omega_0^2 x - 2cv \end{aligned} \tag{2.22}$$

The system can be written in matrix form as $\dot{\mathbf{x}} = A\mathbf{x}$, where $\mathbf{x} = [x, v]^T$ and

$$A = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2c \end{bmatrix} \tag{2.23}$$

The characteristic polynomial of A is

$$\lambda^2 + 2c\lambda + \omega_0^2 = 0 \tag{2.24}$$

with roots:

$$\begin{aligned} \lambda_1 &= -c - \sqrt{c^2 - \omega_0^2} \\ \lambda_2 &= -c + \sqrt{c^2 - \omega_0^2} \end{aligned} \tag{2.25}$$

Since c and ω_0 are positive real numbers, the real parts of λ_1 and λ_2 are always negative, whatever the actual values of c and ω_0 are. If $c < \omega_0$, then λ_1 and λ_2 are complex conjugates with negative real part $-c$. If $c \geq \omega_0$, then $0 \leq c^2 - \omega_0^2 < c^2$, so $0 \leq \sqrt{c^2 - \omega_0^2} < c$, or equivalently, $-c \leq -c + \sqrt{c^2 + \omega_0^2} < 0$ and λ_1 and λ_2 are real and negative.

The exact nature of the free solutions depends on the relative sizes of the constants c and ω_0. There are three cases:

- $c > \omega_0$. Then λ_1 and λ_2 are real, negative, and distinct.

- $c = \omega_0$. Then $\lambda_1 = \lambda_2 = -c < 0$

- $c < \omega_0$. Then $\lambda_1 = \alpha + i\beta$, $\lambda_2 = \bar{\lambda}_1 = \alpha - i\beta$.

In order to find the eigenvalues corresponding to $\lambda_{1,2}$, we can substitute into $(A - \lambda I)\mathbf{v} = 0$.

2.6.1 Overdamped system: $c > \omega_0$

For λ_1, $(A - \lambda_1 I)\mathbf{v} = 0$ can be written as

$$\begin{bmatrix} c + \sqrt{c^2 - \omega_0^2} & -1 \\ -\omega_0^2 & -c + \sqrt{c^2 - \omega_0^2} \end{bmatrix} \mathbf{v} = 0$$

and solutions are of the form

$$\mathbf{v}_1 = s \begin{bmatrix} 1 \\ -c - \sqrt{c^2 - \omega_0^2} \end{bmatrix} \quad s \neq 0 \tag{2.26}$$

For λ_2, we get

$$\begin{bmatrix} c - \sqrt{c^2 - \omega_0^2} & -1 \\ -\omega_0^2 & -c - \sqrt{c^2 - \omega_0^2} \end{bmatrix} \mathbf{v} = 0$$

and solutions are of the form

$$\mathbf{v}_2 = t \begin{bmatrix} 1 \\ -c + \sqrt{c^2 - \omega_0^2} \end{bmatrix} \quad t \neq 0 \tag{2.27}$$

The matrix T whose columns are the eigenvectors of A is:

$$T = \begin{bmatrix} 1 & 1 \\ -c - \sqrt{c^2 - \omega_0^2} & -c + \sqrt{c^2 - \omega_0^2} \end{bmatrix} \tag{2.28}$$

with inverse

$$T^{-1} = \frac{1}{2\sqrt{c^2 - \omega_0^2}} \begin{bmatrix} -c + \sqrt{c^2 - \omega_0^2} & -1 \\ c + \sqrt{c^2 - \omega_0^2} & 1 \end{bmatrix} \tag{2.29}$$

Then, the system $\dot{\mathbf{y}} = T^{-1}AT\mathbf{y}$ is

$$\dot{\mathbf{y}} = \begin{bmatrix} -c - \sqrt{c^2 - \omega_0^2} & 0 \\ 0 & -c + \sqrt{c^2 - \omega_0^2} \end{bmatrix} \mathbf{y} \tag{2.30}$$

with solution:

$$\mathbf{y}(t) = \alpha e^{(-c-\sqrt{c^2-\omega_0^2})t} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta e^{(-c+\sqrt{c^2-\omega_0^2})t} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.31}$$

The solution to $\dot{\mathbf{x}} = A\mathbf{x}$ is given by $T\mathbf{y}$:

$$\mathbf{x}(t) = \alpha e^{(-c-\sqrt{c^2-\omega_0^2})t} \begin{bmatrix} 1 \\ -c - \sqrt{c^2 - \omega_0^2} \end{bmatrix} + \beta e^{(-c+\sqrt{c^2-\omega_0^2})t} \begin{bmatrix} 1 \\ -c + \sqrt{c^2 - \omega_0^2} \end{bmatrix} \tag{2.32}$$

or in a more compact way:

$$\mathbf{x}(t) = \alpha e^{\lambda_1 t} \begin{bmatrix} 1 \\ \lambda_1 \end{bmatrix} + \beta e^{\lambda_2 t} \begin{bmatrix} 1 \\ \lambda_2 \end{bmatrix} \tag{2.33}$$

The only equilibrium point $\mathbf{x}(t) = [0,0]^T$ is a *sink*, because both eigenvalues are real, negative and distinct.

Given the initial condition $\mathbf{x}(0) = \begin{bmatrix} x_0 \\ v_0 \end{bmatrix}$, α and β can be computed with

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = T^{-1}\mathbf{x}(0) \tag{2.34}$$

Figure 2.6.1, was drawn using $c = 2.0$, $\omega_0 = 1.0$, and initial conditions from the sequence:

$$\left(\begin{bmatrix} -3.0 \\ -4.0 \end{bmatrix}, \begin{bmatrix} -3.0 \\ 4.0 \end{bmatrix}, \begin{bmatrix} -2.0 \\ -4.0 \end{bmatrix}, \begin{bmatrix} -2.0 \\ 4.0 \end{bmatrix}, \ldots, \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix} \right) \tag{2.35}$$

2.6.2 Critically damped: $c = \omega_0$

In this case, $\lambda_1 = \lambda_2 = -c < 0$ (a double eigenvalue). Besides

$$A = \begin{bmatrix} 0 & 1 \\ -c^2 & -2c \end{bmatrix} \tag{2.36}$$

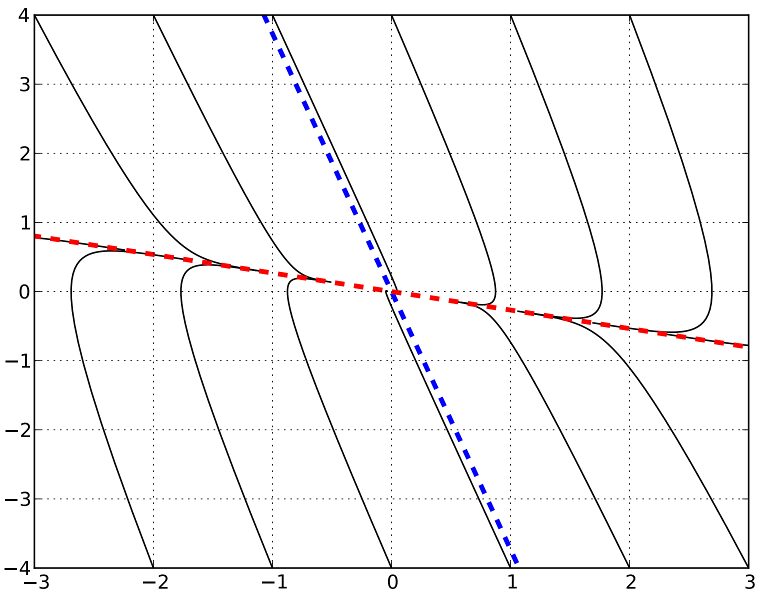


Figure 2.6.1: Sink in phase space (eigenspaces are indicated with dashed lines)

The characteristic polynomial is $\lambda^2 + 2c\lambda + c^2 = 0$, with eigenvalues $\lambda_1 = \lambda_2 = -c$. If we substitute these in $(A - \lambda I)\mathbf{v} = 0$ we get

$$\begin{bmatrix} c & 1 \\ -c^2 & -c \end{bmatrix} \mathbf{v} = 0$$

and solutions are multiple (non–zero) of $\mathbf{v} = [1, -c]^T$. The dimension of the eigenspace in this case is less than the multiplicity of the eigenvalue, and therefore, the eigenspace is *deficient*. Let $\mathbf{w} = [c, 1]^T$, so that $\mathbf{v}$ and $\mathbf{w}$ are linearly independent. Then $A\mathbf{w} = \mu\mathbf{v} - c\mathbf{w}$ for some $\mu \neq 0$. To be precise, $\mu = c^2 + 1$. Let

$$\mathbf{u} = \frac{1}{\mu}\mathbf{w} = \frac{1}{c^2+1}\begin{bmatrix} c \\ 1 \end{bmatrix}$$

Let T be the matrix whose columns are $\mathbf{v}$ and $\mathbf{u}$

$$T = \begin{bmatrix} 1 & c/(c^2+1) \\ -c & 1/(c^2+1) \end{bmatrix} [r] = \frac{1}{c^2+1}\begin{bmatrix} c^2+1 & c \\ -c(c^2+1) & 1 \end{bmatrix} \qquad (2.37)$$

with inverse

$$T^{-1} = \frac{1}{c^2+1}\begin{bmatrix} 1 & -c \\ c(c^2+1) & c^2+1 \end{bmatrix} \tag{2.38}$$

Then, the similarity transformation $T^{-1}AT$ makes possible to express the system in a canonical form

$$\dot{\mathbf{y}} = \begin{bmatrix} -c & 1 \\ 0 & -c \end{bmatrix} \mathbf{y} \tag{2.39}$$

The general solution may be written as

$$\mathbf{y}(t) = \alpha e^{-ct}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + \beta e^{-ct}\begin{bmatrix} t \\ 1 \end{bmatrix}$$

that corresponds to $\mathbf{x} = T\mathbf{y}$

$$\mathbf{x}(t) = \alpha e^{-ct}\begin{bmatrix} 1 \\ -c \end{bmatrix} + \beta e^{-ct}\begin{bmatrix} t + c/(c^2+1) \\ -tc + 1/(c^2+1) \end{bmatrix} \tag{2.40}$$

The only equilibrium point $\mathbf{x}(t) = [0,0]^T$ is a *deficient node*, because there is only a double eigenvalue (algebraic multiplicity is two), but it's eigenspace has dimension one.

Figure 2.6.2 was drawn using $c = \omega_0 = 1.0$, and initial conditions from sequence 2.35

2.6.3 Underdamped: $c < \omega_0$

If $c < \omega_0$, then $\lambda_1 = \alpha + i\beta$, $\lambda_2 = \overline{\lambda}_1 = \alpha - i\beta$, where $\alpha < 0$ and $\beta = \sqrt{\omega_0^2 - c^2}$. The characteristic polynomial is exactly as (2.24), but now the roots are complex conjugates:

$$\begin{aligned} \lambda_1 &= -c + i\sqrt{\omega_0^2 - c^2} \\ \lambda_2 &= -c - i\sqrt{\omega_0^2 - c^2} \end{aligned} \tag{2.41}$$

$(A - \lambda I)\mathbf{v} = 0$ is

$$\begin{bmatrix} c - i\sqrt{\omega_0^2 - c^2} & 1 \\ -\omega_0^2 & -c - i\sqrt{\omega_0^2 - c^2} \end{bmatrix}\mathbf{v} = 0$$

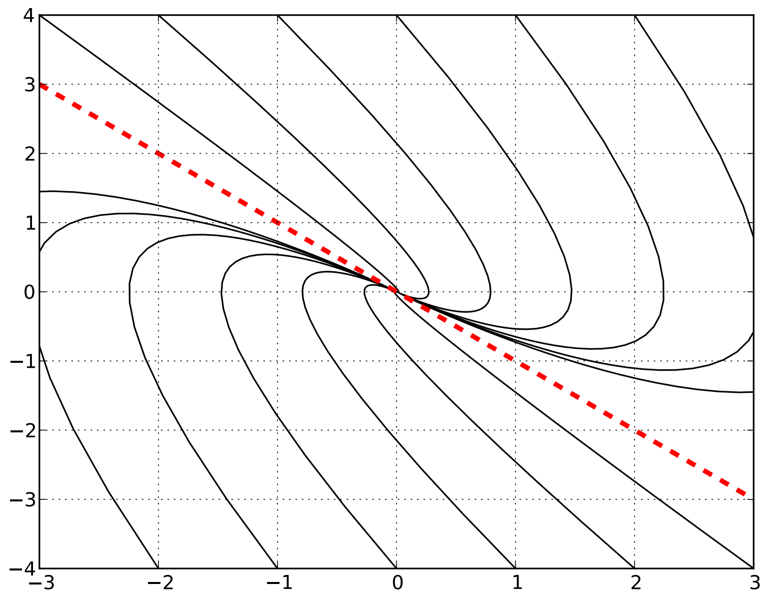


Figure 2.6.2: Deficient node in phase space (the eigenspace is indicated with a dashed line)

and solutions are multiple (non–zero) of

$$\mathbf{v} = \begin{bmatrix} 1 \\ -c + i\sqrt{\omega_0^2 - c^2} \end{bmatrix} \tag{2.42}$$

Let

$$\begin{aligned} \mathbf{w}_1 &= \frac{1}{2}(\mathbf{v}_1 + \bar{\mathbf{v}}_2) = \begin{bmatrix} 1 \\ -c \end{bmatrix} \\ \mathbf{w}_2 &= -\frac{i}{2}(\mathbf{v}_1 - \bar{\mathbf{v}}_2) = \begin{bmatrix} 0 \\ \sqrt{\omega_0^2 - c^2} \end{bmatrix} \end{aligned}$$

These vectors are linearly independent, and we can build a similarity transformation $T^{-1}AT$ with

$$T = \begin{bmatrix} 1 & 0 \\ -c & \sqrt{\omega_0^2 - c^2} \end{bmatrix} \tag{2.43}$$

that has inverse

$$T^{-1} = \frac{1}{\sqrt{\omega_0^2 - c^2}} \begin{bmatrix} \sqrt{\omega_0^2 - c^2} & 0 \\ c & 1 \end{bmatrix} \tag{2.44}$$

Hence we have solutions of the form $\mathbf{x} = T\mathbf{y}$ where

$$\mathbf{y}(t) = k_1 e^{-ct} \begin{bmatrix} \cos\sqrt{\omega_0^2 - c^2}t \\ -\sin\sqrt{\omega_0^2 - c^2}t \end{bmatrix} + k_2 e^{-ct} \begin{bmatrix} \sin\sqrt{\omega_0^2 - c^2}t \\ \cos\sqrt{\omega_0^2 - c^2}t \end{bmatrix} \tag{2.45}$$

$\mathbf{x}(t)$ can be written as:

$$\mathbf{x}(t) = e^{-ct} \begin{bmatrix} \cos\beta t & \sin\beta t \\ -c\cos\beta t - \beta\sin\beta t & -c\sin\beta t + \beta\cos\beta t \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix} \tag{2.46}$$

The phase portrait corresponds to a spiral sink. Figure 2.6.3 was draw using $c = 0.5$, $\omega_0 = 1.0$, and initial conditions from sequence 2.35.

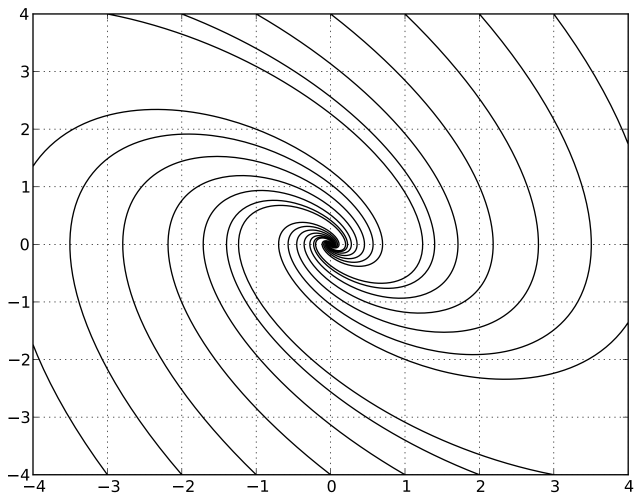


Figure 2.6.3: Spiral sink in phase space

2.7 Criteria for qualification of fixed points

The linear system (2.22) can be completely described with the eigenvalues and eigenvectors (besides stability, we can say how the linear flow is in the neighbourhood of the fixed point). This is not the case for nonlinear systems. For the latter, we could summarize the process followed in this work in a few simple steps:

- Find the fixed points of the system (we will analyze one alternative to do this in the next chapter).
- Evaluate the jacobian $\mathbf{f}^0_\mathbf{y}$ in the fixed point (we are using linearization here).
- Compute the eigenvalues of $\mathbf{f}^0_\mathbf{y}$.
  - If the fixed point is hyperbolic, we can use the linear conjugate flow to describe the behaviour in it's neighbourhood (see table 2.7.1). By consequence, we are able to classify the fixed point as "Asymptotically Stable" or "Unstable".
  - If the fixed point is not hyperbolic, acording to Section 2.4, we can only say if it is "Unstable", or that we don't have enough information from linearization ("Undefined").

2.8 Numerical computing of derivatives and jacobians

Acording to 2.7, we need to evaluate $\mathbf{f}^0_\mathbf{y}$. For dynamical "systems" consisting of one scalar equation, it is enough to check the sign of the derivative evaluated at the fixed point. Next, we'll describe the algorithms used for this purpose.

2.8.1 Numerical differentiation: Ridders Method

The derivative of $f(x)$ is

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \tag{2.47}$$

This formula suggest a naive approach to compute a numerical derivative: pick a small value h and apply (2.47). However, applied uncritically, this procedure is almost guaranteed to produce inaccurate results.

Ridders [15] applied Romberg's method to improve the accuracy in the computation of the first and second derivatives of a real function. Using the Taylor expansion in the vicinity of x:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{2.48}$$

with a truncation error of $e_t \sim h^2 f^{(3)}$. As a consequence, e_t decreases quadratically with decreasing h. When we repeatedly halve the value of h, a series of corresponding values of $\frac{f(x+h)-f(x-h)}{2h}$ is obtained, which we denote by $(A_1, A_2, A_3, \ldots)$.

Now $f'(x) \sim A_1 + e_{t_1} \sim A_2 + e_{t_2}$, furthermore $e_{t_1}/e_{t_2} = 4$, so we get a better approximation:

$$f'(x) \approx \frac{4A_2 - A_1}{4-1} \tag{2.49}$$

which we denote by B_1.

This procedure leads us to the well known Romberg method: with

$$\begin{aligned} B_n &= \frac{A_{n+1} \cdot 4^m - A_n}{4^m - 1}, \quad m = 1 \\ C_n &= \frac{B_{n+1} \cdot 4^m - B_n}{4^m - 1}, \quad m = 2 \end{aligned} \tag{2.50}$$

and so on.

2.8.2 Jacobians with forward differences

The implementation used in this work is based on [3]. We use a forward difference approximation to $\mathbf{f}^0_{\mathbf{y}}$ (the jacobian matrix of $\mathbf{f}(\mathbf{y})$ at $\mathbf{y}_0$), using values of $\mathbf{f}(\mathbf{y})$.

Column j of $J(\mathbf{y}_c)$ is approximated by $\mathbf{f}(\mathbf{y}_c + h_j \mathbf{e}_j)$, where $\mathbf{e_j}$ is the j–th unit vector, and $h_j = \eta^{1/2} \max\{|\mathbf{y}_c[j]|, 1/\mathbf{S}_{\mathbf{y}}[j]\} \operatorname{sgn}(\mathbf{y}_c[j])$. $1/\mathbf{S}_{\mathbf{y}}[j]$ is the typical size of $|\mathbf{y}_c[j]|$ by the user, and $\eta = 10^{\text{-DIGITS}}$, where DIGITS is the number of reliable base 10 digits in $\mathbf{f}(\mathbf{x})$. The corresponding elements of $J(\mathbf{y}_c)$ and J typically will agree in about their first DIGITS/2 base 10 digits[3].

2.9 Parsing of a System of ODE's

The process of finding the structure in the program (a flat stream —or sequence— of tokens) is called *parsing*, and a module that performs this task is a parser [8].

Following the approach proposed by [11], we use PEGs instead of context–free grammars for the definition of our language, and combinator parsers instead of parser generators.

Algorithm 1: Finite difference Jacobian approximation

```
Input  : $n \in \mathbb{Z}$, $\mathbf{y_c} \in \mathbb{R}^n$, $\mathbf{f}(\mathbf{y}_c) \in \mathbb{R}^n$, $\mathbf{S_y} \in \mathbb{R}^n$, $\eta \in \mathbb{R}$
Output : $J \in \mathbb{R}^{n \times n} \approx \mathbf{f_y}(\mathbf{y}_c)$
sqrtEta $\leftarrow \eta^{1/2}$
for $j \leftarrow 1$ to $n$ do
    // calculate column j of $J$
    stepSizej $\leftarrow$ sqrtEta * $\max\{|\mathbf{y}_c[j]|, 1/\mathbf{S_y}[j]\} * \mathrm{sgn}(\mathbf{y}_c[j])$
    tempj $\leftarrow \mathbf{y}_c[j]$
    $\mathbf{y}_c[j] \leftarrow \mathbf{y}_c[j] +$ stepSizej
    stepSizej $\leftarrow \mathbf{y}_c[j] -$ tempj
    fj $\leftarrow \mathbf{f}(\mathbf{y}_c +$ stepSizej $* \mathbf{e}_j)$
    for $i \leftarrow 1$ to $n$ do
        $J[i, j] \leftarrow$ (fj$[i] - \mathbf{f}(\mathbf{y}_c)[i]$)/stepSizej
    end
    $\mathbf{y}_c[j] \leftarrow$ tempj
end
```

2.9.1 Grammars

Most language syntax theory and practice is based on generative systems, particularly context–free grammars (CFGs) and regular expressions (REs) [6]. Chomsky's generative system of grammars allow for ambiguities, which is useful for modelling natural languages, but this power makes difficult to express and parse machine–oriented languages.

Parsing Expression Grammars (PEGs) provide an alternative, *recognition–based* formal foundation for describing machine-oriented syntax. PEGs "solve the ambiguity problem by not introducing ambiguity in the first place" [6] using *prioritized choice*. PEGs provide operators for constructing grammars. The combinator parsing systems from the Scala standard library, use PEG semantics for recognizing. Table 2.9.1 shows how PEG operators are implemented in Scala combinator parsing library.

2.9.2 Combinators parsers Vs. parser generators

Parser generators, such as ANTLR, achieve their goal (generate parsers in a target language, e.g. Java) using a particular grammar notation system (different from the target language). Besides, auxiliary structures and routines (such as actions) have to be programmed in the target language. The programmer has therefore to deal with *two languages*.

In contrast, combinator parsers are implemented in the host language as a library. The parsing rules as well as auxiliary routines are both written in the host language.

But the combinator parsing approach have some disadvantages with respect to specialized parsing systems. The latter have full control on the parser code generation and can apply arbitrary optimizations to the code. Combinators are limited in this area. The obvious consequence is that combinators parsers, in general, are slower than generated parsers.

The main advantage of combinator parsers over parsing generators in our project is the hability to change the grammar more easily.

2.9.3 A grammar for recognizing a dynamical system

A PEG grammar for our problem can be defined as next.

```
system ← sentence+ normDefinition normDefinition?
sentence ← equation / constantDefinition
equation ← dotStateVar "=" expr
constantDefinition ← constant "=" floatingPointNumber
normDefinition ← ("Norm1" / "Norm2") "=" expr
dotStateVar ← "[a-z]\w*" "'"
stateVar ← "[a-z]\w*" !"'"
constant ← "[A-Z][A-Z0-9_]*" !"'"
parameter ← "_\w+" !"'"
expr ← prod ("+" prod / "-" prod)
prod ← signExp ("*" signExp / "/" signExp)
signExp ← "-"? power
power ← (appExpr "^") appExpr
appExpr ← fun "[" expr "]" / simpleExpr
fun ← "Cos" / "Sin"
simpleExpr ← stateVar / constant / parameter / floatingPointNumber / "(" expr ")"
```

For brevity, the definition of a `floatingPointNumber` is taken for granted (see the definition from `scala.util.parsing.combinator.JavaTokenParsers.floatingPointNumber`). The syntax from regular expressions in the JDK is also exploited. For example, `[a-zA-Z]` matches any letter from the alphabet (a through z, or A through Z); `\w` is a word character: `[a-zA-Z_0-9]`.

Note that defining the grammar in a top–down decomposition fashion allows to easily *encode precedences of arithmetical operations directly in the grammar rules*. This grammar can be translated (almost) directly to code in Scala to build a parser. The only parser's task is to build an AST for the given input.

Evaluation of the AST (which is an intermediate representation that has discarded semantically irrelevant parts from the concrete representation in the input) is done (at run–time, and separately from parsing) by an object of the class `DynamicalSystem`. Thus, the class `DynamicalSystem` represents target–machines that can evaluate "programs" expressed in the specific intermediate representation language. Then we have built an *interpreter* for simple dynamical systems.

2.9.4 AST definition and evaluation

Section 2.9.3 stated that the only parser's task is to build an AST. We will use some excerpts from our code to explain how this is done, and also, how the dynamical system is evaluated.

```
1  def sentence = equation | constantDefinition
2  def equation = dotStateVar ~ ("=" ~> expr) ^^ Equation
3  def stateVar = """[a-z]\w*""".r <~ not("'") ^^ StateVar
4  def expr: Parser[Expr] = chainl1(prod, "+" ^^^ Add | "-" ^^^ Sub)
```

- Line 1 is almost identical to the corresponding PEG expression. Note that the vertical bar `|` means prioritized–choice.
- Line 2 uses the sequence operator ~, the ~> operator, and ^^.
  - The ~> and <~ operators are used to match and discard a token. For example, the result of `"=" ~> expr` is just the result of `expr`, not a value of the form `"=" ~ expr`.
  - `expr ^^ f` applies `f` to the result of `expr`. For example, line 2 calls `Equation(dotStateVar, expr)`. `Equation is a case class`: an `apply` method is provided automatically for the companion object that lets the user of the case class to construct objects without the `new` keyword. Besides, the call `Equation(dotStateVar, expr)` is a shortcut to `Equation.apply(dotStateVar, expr)`.
- Line 4 uses ^^^ combinator. `p^^v` replaces the result of `p` by the constant `v`. `chainl1(p, s)` combinator matches 1 or more repetitions of `p` (with type `P`), separated by matches of `s` (`s` must, upon matching each separator, produce a binary function that is used to combine neighboring values). For example if `p` produces values `prod1`, `prod2`, `prod3`, and `s` produces `Add`, `Sub`, then the result is `(prod1 Add prod2) Sub prod3`. Note the left associative grouping. There is an analog `chainr1` combinator we are using for the exponentiation operation (that is right associative).
- By using ^^, ^^^, and case classes (like `Equation`, `StateVar`, `Add`, `Sub` and others) we are building the nodes of a parse tree (the AST). The nodes of the AST know how to evaluate themselves using a

simple idea: each node is decomposed into its parts, the parts are evaluated, and then merged again with the operation that corresponds to the node type.

2.10 Final Remarks

The study of similarity transformations, eigenvalues, eigenvectors, linear flows in Section 2.5, has been practical during software development. For example, understanding the patterns of some canonical forms for matrices has been valuable to design tests for eigenvalue computation, as well as to detect special cases that are hard in terms of numerical computation.

Table 2.7.1 summarizes the criteria for qualification of fixed points.

An interpreter for dynamical systems has been developed, using PEGs and Combinator Parsers from Scala.

Table 2.7.1: Criteria used for qualification of fixed points

Case Number	isHyperbolic	negativeFound	positiveFound	complexFound	Linear Conjugate Flow	Stability	Comments
1	false	false	false	X	–	Undefined	Every eigenvalue has zero real part
2	false	false	true	X	–	Unstable	Every real part is positive of zero
3	false	true	false	X	–	Undefined	Every real part is negative or zero
4	false	true	true	X	–	Unstable	Negative, zero and positive real parts
–	true	false	false	X	–	–	This case is impossible
5	true	false	true	false	Source	Unstable	
6	true	false	true	true	Spiral Source	Unstable	
7	true	true	false	false	Sink	Asymptotically Stable	
8	true	true	false	true	Spiral Sink	Asymptotically Stable	
9	true	true	true	false	Saddle	Unstable	
10	true	true	true	true	Spiral Saddle	Unstable	

	A_1	A_2	A_3	A_4	$\cdots$
$m = 1$		B_1	B_2	B_3	$\cdots$
$m = 2$			C_1	C_2	$\cdots$
$m = 3$				D_1	$\cdots$
$\vdots$					

Table 2.8.1: Table resulting from Ridders method

Description	PEG notation	Scala notation
Literal string	`' '`	`" "`
Literal string	`" "`	`" "`
Character class	`[ ]`	`"[ ]".r`
Any character	`.`	`".".r`
Grouping	`(e)`	`(e)`
Optional	`e?`	`(e?)` or `opt(e)`
Zero–or–more	`e*`	`(e*)` or `rep(e)`
One–or–more	`e+`	`(e+)` or `rep1(e)`
And–predicate	`&e`	`guard(e)`
Not–predicate	`!e`	`not(e)`
Sequence	$e_1 e_2$	`e1` $\sim$ `e2`
Prioritized choice	e_1 / e_2	`e1 | e2`

Table 2.9.1: Implementation of PEG operators in Scala

Observe intently the birds of heaven, ...

Jesus Christ

3 Bifurcation Diagrams based on Niche PSO

In this chapter the basics of PSO and Niche PSO are studied (pointing out some important configuration chosen for the algorithms involved). Section 3.3 defines what a bifurcation diagram is and Subsection 3.3.1 explains how Niche PSO is exploited to find the (possibly multiple) fixed points for a given set of values of the parameters.

3.1 PSO

If $\mathbf{x}_i(t) \in \mathbb{R}^{n_x}$ (a vector with n_x real components) is the position of the i–th particle of the swarm (which has size n_s) in the search space at time t (which denotes discrete time steps), then after one time step the new position is given by:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{3.1}$$

where $\mathbf{v}_i(t)$ denotes the velocity vector of the i–th particle. The velocity has a *social* and a *cognitive* component, that drive the optimization process.

For the basic PSO algorithm, the velocity update of the i–particle has the form:

$$\mathbf{v}_{ij}(t+1) = \mathbf{v}_{ij}(t) + c_1 r_{1j}(t)\mathbf{p}_{ij}(t) + c_2 r_{2j}(t)\mathbf{s}_{ij}(t) \tag{3.2}$$

where the subindex ij denotes the j–th entry corresponding to i–th particle; $\mathbf{p}(t)$, the cognitive component and $\mathbf{s}(t)$ the social component. c_1, and c_2 are positive acceleration constants used to scale the contribution of the cognitive and social components respectively. r_{1j} and r_{2j} are random values in the range $[0, 1]$, sampled from a uniform distribution.

$\mathbf{p}(t)$ has the same definition for GBest and LBest topologies, but $\mathbf{s}(t)$ is defined differently for the same topologies (the "global" best component is computed differently taking into account the concept of *neighbourhood* in the LBest topology).

3.2 Niche PSO

Niching algorithms are used for locating and maintaining multiple solutions [13]. The PSO (Particle Swarm Optimization) algorithm has poor habilities to achieve this goal, as summarized by [2]. The NichePSO was developed to enhance these habilities of PSO.

Niche PSO starts with a main swarm that contains all particles. When a particle seems to have converged on a solution, a subswarm is formed with that particle and its closest neighbour (the subswarm particles are no longer part of the main swarm). Each subswarm refines and maintain their own solution. To maintain niches, it is important that every subswarm is independent from the others. See Algorithm 2. In the following sections we are going to describe this algorithm further (using our particular configuration). We are using an implementation from the excellent CILib library.

3.2.1 Main Swarm Training

The main swarm is a normal PSO swarm trained using a cognition–only model (social component is zero) to promote exploration. This is configured with:

```
val velProv = new StandardVelocityProvider()
velProv.setCognitiveAcceleration(ConstantControlParameter.of(1.2))
```

Using (3.2), that means $c_1 = 1.2$, $c_2 = 0.0$.

Algorithm 2: NichePSO Algorithm

```
Input  : $n_x$ (the dimension of the search space), $n_S$ (the size of the swarm)
Output : $S_k.\hat{\mathbf{y}}$ (the best solution of each subswarm)
Create and initialize a $n_x$–dimensional *main* swarm, $S$
repeat
    Train the main swarm, $S$, for one iteration using the *cognition–only* model
    Update the fitness of each main swarm particle, $S.\mathbf{x}_i$
    for each sub–swarm $S_k$ do
        Train sub–swarm particles, $S_k.\mathbf{x}_i$, using a *full model* PSO
        Update each particle's fitness
        Update the swarm radius $S_k.R$
    end
    If possible, merge sub–swarms
    Allow sub–swarms to absorb any particles from the main swarm that moved into the sub–swarm
    If possible, create new sub–swarms
until *stopping condition is true*
Return $S_k.\hat{\mathbf{y}}$ for each sub–swarm $S_k$ as a solution
```

3.2.2 Sub–Swarm Training

In turn, each subswarm is trained using GCPSO (Guaranteed Convergence PSO) as suggested by [5], because "it has guaranteed convergence to a local minimum [19], and because the GCPSO has been shown to perform well on extremely small swarms [19]".

The GCPSO changes the position and velocity update *of the global best particle*, at the same time that uses a "conventional" PSO for the rest of the particles. If τ is the index of the global best particle ($\mathbf{y}_\tau = \hat{\mathbf{y}}$), GCPSO changes the position update to

$$\mathbf{x}_{\tau j}(t+1) = \hat{\mathbf{y}}_j(t) + w\mathbf{v}_{\tau j}(t) + \rho(t)(1 - 2r_2(t)) \tag{3.3}$$

This is obtained if the velocity update of the global best particle changes to

$$\mathbf{v}_{\tau j}(t+1) = -\mathbf{x}_{\tau j}(t) + \hat{\mathbf{y}}_j(t) + w\mathbf{v}_{\tau j}(t) + \rho(t)(1 - 2r_2(t)) \tag{3.4}$$

where $\rho(t)$ is a scaling factor defined next:

$$\rho(t) = \begin{cases} 2\rho(t) & \text{if } \#\text{sucesses}(t) > \varepsilon_s \\ 0.5\rho(t) & \text{if } \#\text{failures}(t) > \varepsilon_f \\ \rho(t) & \text{otherwise} \end{cases} \tag{3.5}$$

where #successes and #failures denote the number of *consecutive* successes and failures, respectively.

For the rest of the particles, the "conventional" PSO that we have chosen uses a `ConstrictionVelocityProvider`. Therefore the velocity update changes to:

$$\mathbf{v}_{ij}(t+1) = \chi[\mathbf{v}_{ij}(t) + \varphi_1(\mathbf{y}_{ij}(t) - \mathbf{x}_{ij}(t)) + \varphi_2(\hat{\mathbf{y}}_j(t) - \mathbf{x}_{ij}(t))] \tag{3.6}$$

where

$$\chi = \frac{2\kappa}{|2 - \varphi\sqrt{\varphi(\varphi-4)}|} \tag{3.7}$$

with $\varphi = \varphi_1 + \varphi_2$, $\varphi_1 = c_1 r_1$, and $\varphi_2 = c_2 r_2$. Under the conditions that $\varphi \geq 4$ and $\kappa \in [0, 1]$, the swarm is guaranteed to converge [5].

The `ConstrictionVelocityProvider` is used together with a `ClampingVelocityProvider`. Next we show how this has been configured in our code:

```
// velocityProvider
val subSwarmVelProv = new GCVelocityProvider()
subSwarmVelProv.setRho(ConstantControlParameter.of(1.0))
// epsilons y epsilonf
subSwarmVelProv.setSuccessCountThreshold(15)
subSwarmVelProv.setFailureCountThreshold(5)
val constrictionVelProv = new ConstrictionVelocityProvider()
constrictionVelProv.setSocialAcceleration(ConstantControlParameter.of(2.05))
constrictionVelProv.setCognitiveAcceleration(ConstantControlParameter.of(2.05))
constrictionVelProv.setKappa(ConstantControlParameter.of(1.0))
val clampingVelProv = new ClampingVelocityProvider(
  ConstantControlParameter.of(1.0), constrictionVelProv)
subSwarmVelProv.setDelegate(clampingVelProv)
```

To provide better exploration habilities, we have chosen a `LBestTopology` for the subswarm.

3.2.3 Creation and Merging of Niches

The basic ideas are:

- A sub–swarm is formed when a particle seems to have converged on a solution. If the standard deviation (over several iterations) of the fitness of a partice is below a threshold, the particle has converged to a solution.

- A niche is formed with the closest neighbour (euclidean distance is used for measuring distances).
- Particles leaving the main swarm and are added to a suitable niche.
- The merge strategy for subswarms needs special care. If we choose not to merge subswarms to maintain every niche, then multiple subswarms might be refining the same solution. On the other hand, if subswarms merge too easily, we could lose some niches. We have used a merge detection strategy based on diversity.

We can configure these behaviour with something like:

```
// Niche Detector
this.nicheDetector = new MaintainedFitnessNicheDetection()
this.nicheDetector.asInstanceOf[MaintainedFitnessNicheDetection].
  setThreshold(ConstantControlParameter.of(1.0E-12))
this.nicheDetector.asInstanceOf[MaintainedFitnessNicheDetection].
  setStationaryCounter(ConstantControlParameter.of(3.0))

// Merge Detector
this.mergeDetector = new DiversityBasedMergeDetection()
this.mergeDetector.asInstanceOf[DiversityBasedMergeDetection]
  .setThreshold(ConstantControlParameter.of(1.0e-12))
```

3.3 Bifurcation diagrams

The summary provided in this section is based on [16]. For a system of ODEs

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, \lambda) \tag{3.8}$$

a bifurcation diagram depicts a scalar measure $[\mathbf{y}]$ versus the real parameter λ, where $(\mathbf{y}, \lambda)$ solves (3.8). For example, $[\mathbf{y}]$ could be y_k (one of the components of the n-vector $\mathbf{y}$), or some convenient vector norm (like $\|\mathbf{y}\|_2$, $\|\mathbf{y}\|_\infty$).

As a consequence of the existence and uniqueness theorems of ODEs, there are three kind of *trajectories*, namely:

1. Stationary solutions $\mathbf{y}(t) \equiv \mathbf{y}^s$, $\mathbf{f}(\mathbf{y}^s) = 0$ (e.g. nodes, saddle points, foci, and degenerate cases like centers) and turning points.
2. Periodic solutions $\mathbf{y}(t+T) = \mathbf{y}(t)$ (e.g. limit cycles, heteroclinic and homoclinic orbits).
3. One–to–one solutions $\mathbf{y}(t_1) \neq \mathbf{y}(t_2)$ for $t_1 \neq t_2$

Some stationary solutions can be at the same time *bifurcation points*. Informally, a bifurcation point (with respect to λ) is a solution $(\mathbf{y}_0, \lambda_0)$ to (3.8) where the number of solutions changes when λ passes λ_0. Examples of this kind of bifurcations are turning points, transcritical and pitchfork bifurcations.

A bifurcation from a branch of equilibria to a branch of periodic oscillations is called Hopf bifurcation.

Local bifurcations can be characterized by locally defined eigenvalues crossing some line. For stationary bifurcations and Hopf bifurcations, these are the eigenvalues $\mu(\lambda)$ of the Jacobian $\mathbf{f}_\mathbf{y}(\mathbf{y}, \lambda)$ of stationary solutions $(\mathbf{y}, \lambda)$ and the line is the imaginary axis in the complex plane.

Global bifurcations (e.g. homoclinic bifurcation) cannot be analized based on locally defined eigenvalues. This work only adresses *local bifurcations*.

3.3.1 Using NichePSO to find fixed points

Traditionally, finding fixed points of (3.8) is done with techniques such as *homotopy* and *continuation*. However, we have formulated this problem as the optimization problem: Find a set of solutions $\mathcal{Y}_\lambda = \{\mathbf{y}_1^s, \mathbf{y}_2^s, \ldots, \mathbf{y}_{n_{\mathcal{Y}_\lambda}}^s\}$, such that each $\mathbf{y}^s \in \mathcal{Y}_\lambda$ is a minimum of $\|\mathbf{f}(\mathbf{y}, \lambda)\|$. Because, in general, we can expect several solutions for each λ in a multi–parameter problem, we need an optimization method than can find multiple solutions. This is where NichePSO is used in our work.

3.4 Final Remarks

Training of subswarms takes advantage of several CIlib implementations: `ConstrictionVelocityProvider`, `GCVelocityProvider`, `ClampingVelocityProvider`, `LBestTopology`, `DiversityBasedMergeCriterion`, among others. This chapter has shown a configuration of Niche PSO adapted to find fixed points of dynamical systems (a problem reformulated as an optimization problem).

The following main parameters of Niche PSO has to be tuned for best results:

- The size of the main swarm.
- The value of the threshold for the `DiversityBasedMergeCriterion` to avoid excessive merging of niches at the same time that some merging is allowed in order to get benefits from the social information and experience of merged subswarms.
- The stopping condition of the algorithm. Running Niche PSO for too long can lead to loss of solutions. On the other hand, stopping too quickly could give wrong solutions.

When I am working on a problem I never think about beauty. I only think about how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong.

Buckminster Fuller

4 A tool for Fixed Point Qualification of Bifurcation Diagrams based on Niche PSO

In previous chapters consideration has given to theoretical foundations of qualification of fixed points of dynamical systems. In this chapter, a description of important implementation issues will be considered.

Next, a simplified description of the process of plotting a bifurcation diagram with *Bidiatool* (Bifurcation diagram tool) follows.

1. Parse a description of the dynamical system. To be successfully parsed, such description must be recognized by the previously defined PEG in Section

2. We build a function $\mathbf{f}(\mathbf{y})$ that may depend on at most two parameters λ_1, λ_2, using `Dynamical-System`, and *closing over* some of the parameters (the λs) to fix their values (this is done by the user), so that we have at most two parameters varying quasi-statically. In other words, if there are more than two parameters, then we require the user to fix the additional ones to specific

values (or even all of them but one).

3. Using **f** from the previous step, NichePSO minimizes $||\mathbf{f}||$ and tries to find *every* global minima (which should correspond to fixed points of the dynamical system).

4. Given that we have no absolute guarantee that NichePSO has converged to fixed points only (it may have stagnated somewhere else), we discard solutions with fitness above a given tolerance.

5. Classify the stability of every fixed point using the criteria described in

6. If the fixed point is hyperbolic, we try to classify the conjugate linear flow near it.

7. Plot the bifurcation diagram.

To actually execute the process above, bidiatool must have modules for:

- Parsing and evaluating dynamical systems.
- Minimizing **f** using NichePSO.
- Computing derivatives and jacobians for the stability quatilification; for hyperbolic fixed points, classifying the conjugate linear flow in their neighbourhood.
- Plotting bifurcation diagrams.
- Interacting with the user (via a GUI).

The first three bullets have been addressed in previous chapters. In this chapter we will describe describe the last two bullets in the previous list and some ideas and tools used in the aforementioned modules.

Section 4.1 remarks that the implementation of Bidiatool is using the JVM, taking advantage of both Java and Scala. Section 4.2 just states some choices made during development of Bidiatool, related to 2D and 3D plotting. Section 4.3 discusses an important part of the application: the Graphical User Interface (GUI). Then, taking as an example a small part of the GUI (see Fig. 4.3.1) the reader can have a glimpse of how testing has been important in this work, and how non–trivial programming techniques have been exploited to get a better design (and a higher-level of abstraction). Listing 4.5 shows a pattern that takes advantage of delimited continuations to undo inversion of control in GUIs, to get a more "imperative" style of programming user interfaces.

Finally we conclude this chapter with some remarks.

4.1 The JVM, Java and Scala

In previous chapters we have stated the theoretical foundations of our work, as well as important algorithms used in our application. However, we have barely touched implementation issues (that have needed careful thought, and a lot of time and effort).

One of the requirements was working on top of the Java Virtual Machine (JVM). At this point, we should make a distinction here:

- *The Java language*. The Java programming language is a general–purpose, concurrent, class–based, object–oriented language. It is a strongly and statically typed language. It is a relatively high level language, in that details of the machine representation are not available through the language. It includes automatic storage management, typically using a garbage collector. The Java programming language is normally compiled to the bytecoded instruction and binary format defined in the Java Virtual Machine Specification [7].

- *The Java Virtual Machine (JVM)*. It is the cornerstone of the Java Platform. It is the component of the technology responsible for its hardware– and operating system–independence, the small size of its compiled code, and its ability to protect its users from malicious programs. It is an abstract computing machine that, like a real computing machine, has an instruction set and manipulates various memory areas at run time. *The JVM knows nothing of the Java programming language, only of a particular binary format*, the `class` file format [12]

We chose a somewhat conservative approach: working on top of the JVM (mature and well stablished) but using Java's type system *and* the more powerful type system from Scala.

4.2 Plotting in 2D and 3D

Virtually every bifurcation diagram is drawn in 2D. Therefore, we needed to make a choice of the library that would serve this purpose. This had to take into account the Java GUI toolkit that was going to be used to build the user interface of bidiatool (some choices were Java Swing and the SWT from Eclipse).

Another "compatibility" issue we had to address was the choice of the 3D library. The graphic output must be embedded in a container from the same Java GUI toolkit of the user interface.

The final choice was this:

- Java Swing for the GUI Toolkit.

- JFreeChart for 2D plotting.

- Jzy3d for 3D plotting.

The "backeend" used for 2D plotting is JFreeChart, but we are using a convenient wrapper for this library: scala-chart (there is less boilerplate to write compared to plain java programming with the JFreeChart API).

Jzy3d depends on JOGL (Java Bindings for OpenGL) and that means Jzy3d depends on native libraries. The good news is that those dependencies are distributed in a convenient way to be used by the Java platform. Some of the advantages are: good performance and interactive 3D graphics. As a sample of beautiful 3D interactive plots (illustrating some common functions used for optimization) produced with jzy3d and scala code, see `https://github.com/oscarvarto/benchmarkPlots/wiki`.

4.3 Graphical User Interface

A simple part of the GUI will illustrate some of the non–trivial techniques used in bidiatool. During the process of plotting the bifurcation diagram, the user is required to configure the parameters for further processing of the dynamical system. For example, suppose the user wants to plot a diagram for parameter `_w`, then he must choose the range [`From`, `To`] for that parameter and the `Step` used in that range. See Fig. 4.3.1, that corresponds to an instance of class `ConfParamFrame`.

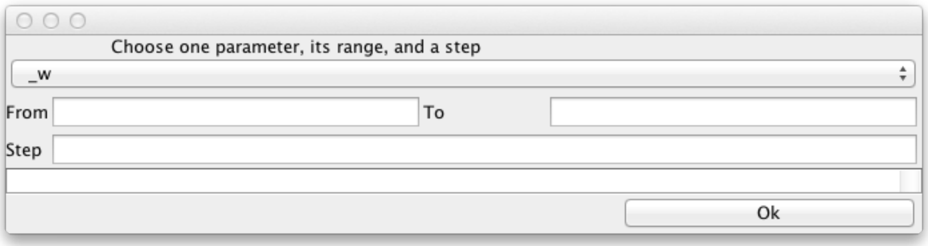


Figure 4.3.1: Configuration of parameters

Bidiatool has been constructed trying to follow important principles like correctness and modularity.

In order to increase the chances that our implementation is correct, we have strived to test as many parts of it as possible. There are tests for numerical algorithms (derivatives, jacobians, eigenvalues), Niche PSO, parsing, stability qualification, etc., and also some tests that integrate those smaller blocks.

GUI code is a substantial part of a lot of applications (roughly 45–60 % of an average codebase [14]), but nonetheless, GUI tests remain relatively unused in practice and remain an unexplored area of research [14].

To test a GUI we need to to be able to click, type and have the ability to perform a multitude of other actions, so the needs for these kind of tests are unique. We chose FEST-Swing as a tool to test the GUI-functionality. FEST simulates (using a software robot) actual user gestures at the operating system level, ensuring that the application will behave correctly in front of the user.

The package summary for `javax.swing` states:

> In general Swing is not thread safe. All Swing components and related classes, unless otherwise documented, must be accessed on the event dispatching thread.

Therefore, as stated by the Fest-Swing documentation: "the cardinal rule is: creation and access of Swing components should be done in the Event Dispatch Thread (EDT.)"

See Listing 4.1 as an example of some of the implemented tests for GUI's.

Listing 4.1: Excerpts of GUI Tests implemented for frame in Fig. 4.3.1

```
 1 class ConfParamFrameTest1 extends FestiveFunSuite with Matchers {
 2   val dummyProject = Project("DummyProject", DynSys1.dynSys)
 3 
 4   override def beforeEach() {
 5     val frame = GuiActionRunner.execute(
 6       new GuiQuery[ConfParamFrame]() {
 7         protected def executeInEDT() = new ConfParamFrame(dummyProject)
 8       })
 9     window = new FrameFixture(frame)
10     window.show()
11   }
12 
13   test("From, To and Step entries must be numbers.", GUITest) {
14     window.button("ConfirmationOrSkipPanel.okButton").click()
15     window.textBox("ConfParamFr.logArea").text() should be(
16       """|From, To and Step entries must be numbers
17          |empty String
18          |empty String
19          |empty String""".stripMargin)
20   }
21 
22   test("No error message if param confiration is Ok", GUITest) {
23     window.textBox("ConfParamFr.paramInfo.rangeFrom").setText("0.0")
24     window.textBox("ConfParamFr.paramInfo.rangeTo").setText("1.0")
25     window.textBox("ConfParamFr.paramInfo.step").setText("0.1")
26     window.button("ConfirmationOrSkipPanel.okButton").click()
27     window.textBox("ConfParamFr.logArea").text() should be("")
28   }
29 }
```

Test `ConfParamFrameTest1` is written using ScalaTest (as every test implemented in our work so far). Next we will explain some details of Listing 4.1.

- Line 1 shows `ConfParamFrameTest1` extends `FestiveFunSuite`. The latter was created to simplify GUI tests, refactoring some important commonalities:

    - Declaration of a mutable member of `FestiveFunSuite` class: `var window: FrameFixture`. The frame fixture that will handle the tested window in a EDT–safe way is bound to `window`.
    - Installation of `FailOnThreadViolationRepaintManager` to catch EDT-access violations. This is done before every test in the class.
    - We must "clean up resources" after each test, as explained in the documentation of Fest-Swing:

        > FEST-Swing forces sequential test execution, regardless of the testing framework. To do so, it uses a semaphore to give access to the keyboard and mouse to a single test. Cleaning up resources after running each test method releases the lock on such semaphore. To clean up resources simply call the method `cleanUp` in the FEST-Swing fixture inside.

- Method `beforeEach`, lines 4–11, shows the right way to create Swing components using a convenient mechanism to access those in the EDT from test code. Quoting the Fest-Swing documentation:

    > This mechanism involves three classes.
    >
    > - `GuiQuery`, for performing actions in the EDT that return a value
    > - `GuiTask`, for performing actions in the EDT that do not return a value
    > - `GuiActionRunner`, executes a `GuiQuery` or `GuiTask` in the EDT, re-throwing any exceptions thrown when executing any GUI action in the EDT.

- Every test for the GUI module is *tagged* `GUITest` as shown in lines 13 and 22. This tag serves a special purpose: to identify tests that take control over the mouse and keyboard while running.
    - The build file for the project (written for `sbt`) was configured to be able to run *only* this kind of tests using the `-n` option as shown in line 12 of Listing 4.2. ScalaTest also provides the `-l` option to exclude any test by tag (or a list of names of tags surrounded by double quotes) [10].
    - According to the documentation for `org.scalatest.Tag`,

        > The tag annotation must be written in Java, not Scala, because annotations written in Scala are not accessible at runtime.

      The Java implementation for `GUITest` annotation is

```
package umich.gui.tags;

import java.lang.annotation.*;
import org.scalatest.TagAnnotation;

@TagAnnotation
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.TYPE})
public @interface GUITest {}
```

  – Now we can run only tests tagged `GUITest` from sbt (0.12.4) with `gui:test`

- The first test, lines 13–20, verifies that clicking the "Ok" button (see Fig. 4.3.1) without entering any numbers in the From, To, and Step fields gives a specific error message in the "log area" of the frame.

- The second test, lines 22–28, verifies that appropiate input is accepted without problems.

Listing 4.2: Special configuration for tests tagged `GUITest`

```
lazy val bidiatool = Project(
  "bidiatool",
  file("."),
  settings = commonSettings ++ Seq(
    libraryDependencies ++= Seq(
    )
  )
).configs( GUITests )
.settings( inConfig(GUITests)(Defaults.testTasks): _*)
.settings(
  testOptions in GUITests := Seq(
    Tests.Argument("-n", "umich.gui.tags.GUITest")
  )
)

lazy val GUITests = config("gui") extend(Test)
```

Now let's discuss a little bit more abstract part of the same example (see Fig. 4.3.1): validation of input (numeric fields From, To, and Step) using *Applicative Functors*. The following comments are for Listing 4.3.

- Class `Range` and its object companion are auxiliary to class `ParameterConfig` and its object companion.

- Line 1 and 7 show both constructors for `Range` and `ParameterConfig` are `private` and that is intentional, because the provided way to (indirectly) instantiate this classes is through respective `apply` methods in their companion objects (see Lines 3 and 9).

- Range.apply has return type Validation[String, Range]. A Validation[E, A] represents either a Success(a) or a Failure(e) and it's motivation is to provide the instance of Applicative[λ[_]] (where λ[α] = Validation[E, α]) that accumulates errors through semigroup[1] E.
- Line 11 call Range.apply and right after that transform its output value to a value of type ValidationNel[String, Range] (which is just a convenient way to abbreviate type Validation[NonEmptyList[String], Range]). Something analog is done in lines 12–13.

Lines 14–15 are the reason of Listing 4.3. We are "applying" the function { (r, s) ⇒ new ParameterConfig(r.from, r.to, s.from) } to (vnelRange ⊛ vnelStep). The aforementioned function knows nothing about the possibility of failure: we are letting Validation handle this automatically for us. In case of success we have a tuple (r, s) representing a range and a step, and we can build a ParameterConfig from them. What happens in case of failure? In this example that could come from vnelRange or vnelStep (remember that both are applicative values, instances of Applicative[λ[_]] where λ[Range] = Validation[NonEmptyList[String], Range]).

- Suppose

  ```
  vnelRange = Failure(NonEmptyList(cond1Msg))
  vnelStep = Success(Range(0.0, 1.0))
  ```

  In that case, lines 14–15 would return a Failure(NonEmptyList(cond1Msg)).

- If vnelStep = Failure(NonEmptyList(cond2Msg)), then lines 14–15 would return the accumulated failures Failure(NonEmptyList(cond1Msg, cond2Msg)).

We have used a very succint and powerful way to manage every possibility thanks to Applicative Functors. Please note that Listing 4.3 has nothing to do with GUI code.

Additional validation is used in the code corresponding to Fig. 4.3.1 to make the application more robust: using functional programming (Validation, amongst other *Scalaz* facilities) we accept only suitable numerical input that corresponds to an actual ParameterConfig. In case of failures, the accumulated error messages are used to give the user some feedback to correct his input.

[1] In the *abstract algebra* sense, a semigroup is a set together with a binary operation on that set that satisfy a closure and an associative law. Unlike a Monoid, there is not necessarily a *zero*.

Listing 4.3: Validation of Parameter Configuration using Applicative Functors

```scala
class Range private(val from: Double, val to: Double)
object Range {
  def apply(from: Double, to: Double, errorMsg: String):
      Validation[String, Range] = if (from <= to) new Range(from, to).success
                                  else errorMsg.fail
}
class ParameterConfig private(val from: Double, val to: Double, val step: Double)
object ParameterConfig {
  def apply(from: Double, to: Double, step: Double):
      ValidationNel[String, ParameterConfig] = {
    val vnelRange = Range(from, to, cond1Msg).toValidationNel[String, Range]
    val vnelStep = Range(step, (to - from).abs, cond2Msg).
      toValidationNel[String, Range]
    (vnelRange ⊛ vnelStep) { (r, s) ⇒
      new ParameterConfig(r.from, r.to, s.from) }
  }
  val cond1Msg = "From must be less or equal than To"
  val cond2Msg = "Step must be less or equal than given range"
}
```

The same example would be useful to explain how delimited continuations (an advanced Scala construct) are exploited to handle GUI events code in a suitable way. Next comments are for Listing 4.4.

- If validation of input fails, then we get a `NonEmptyList(String)` where every string of this non–empty list is an error message that should be used as feedback for the user to correct his input. That is shown in lines 4–10.

- If validation succeeds we get a `ParameterConfig` and lines 12–17 manages two different situations: code for `ConfParamFrame` is used 1) during normal operation of bidiatool *or* 2) during unit-testing (using Fest-Swing). During 1) we call the continuation with `c(())` and `ConfParamFrame` loses control of the application: we are using delimited continuations to undo the inversion of control that is usual in GUI code.

Listing 4.4: Clicking “Ok” (see Fig. 4.3.1) calls this code

```scala
1  val okAction = new AbstractAction("Ok") {
2    def actionPerformed(event: ActionEvent) {
3      getErrorsOrParamConfig().fold(
4        errorsNel ⇒
5          {
6            // Show errors to user
7            val errorMessages: List[String] =
8              EntriesMustBeNumbers :: errorsNel.list
9            logPane.textArea.setText(errorMessages.mkString("\n"))
10         },
11       parConf ⇒
12         {
13           import scalaz.std.option._
14           import scalaz.syntax.std.option._
15           cont.cata(
            c ⇒ c(()), println("This should happen during testing only"))
        })
  }
}
```

Fig. 4.3.1 is just one of several dialogs that the user interacts with (one after the other) to input information for plotting a bifurcation diagram. Undoing inversion of control is useful here because once we recover control of the application, we can write code in only one place, instead of having logic for managing GUI spread in several files. The purpose of using this programming style is to write code that is more maintainable and easier to change.

As an example, lets review an excerpt of the code that creates several dialogs (one after the other) that expect input from the user to create a bifurcation diagram (see Listing 4.5).

Listing 4.5: Usage of delimited continuations to undo inversion of control in GUIs.

```
1  class NewProjectAction() extends AbstractAction("New Project") {
2    var cont: (Unit ⇒ Unit) = null
3    def actionPerformed(event: ActionEvent) = reset {
4      val projName = getProjectName()
5      val dynSys = getDynamicalSystem()
6      val proj1 = addParamConfig(Project(projName, dynSys))
7      val proj2 = if (dynSys.maybeTwoParameterSimulation)
8                  addParamConfig(proj1) else proj1
9      processProject(proj2)
10   }
11
12   // code for additional dialogs
13
14   def addParamConfig(proj: Project): Project @cps[Unit] = {
15     val cpf = new ConfParamFrame(proj)
16     cpf.pack()
17     cpf.setVisible(true)
18     shift {
19       k: (Unit ⇒ Unit) ⇒
20         {
21           cont = k
22           cpf.cont = Some(cont)
23         }
24     }
25     cpf.setVisible(false)
26     cpf.getParamConfig().cata(t ⇒ proj.addParamConf(t), proj)
27   }
```

- Lines 3–10 show calls to methods that create dialogs and return output produced from user input. This methods are using delimited continuations to 1) wait until the user has introduced suitable input (remember we are using validation for this) so that the next dialog can be created and shown *and* 2) undo inversion of control.

- Note lines 4–9 show how we are able to program logic (with a convenient imperative programming style) for GUI in *one place* once we have recovered control.

- Method `addParamConfig` is called in line 6. Lines 14–27 show the corresponding signature and body. Note that line 15 shows the creation of a `ParamConfFrame` from previous examples (see Fig. 4.3.1).
- Once `addParamConfig` has been called, lines 15–17 run normally, execution enters the `shift`, *captures* the continuation, and returns to the end of the enclosing `reset`, exiting the `actionPerformed` method (line 10).
- Note that lines 21–22 save the captured continuation `k` inside the member `cont` of `cpf` which in turn has type `ConfParamFrame`. Remember that line 14 of Listing 4.4 calls this continuation only after proper input has been introduced.
- Once the continuation has been called, execution continues at line 25. Line 26 returns a `Project` and that should be annotated in the signature of `addParamConfig` (see line 14). However, because we are using a `shift`, `addParamConfig` is also annotated with `@cps`.
- The returned `Project` is bound to `proj1` in line 6, and execution continues *inside* `actionPerformed` with the rest of method calls (some of which are also `@cps` annotated).

4.4 Final Remarks

This chapter puts emphasis on implementation details of Bidiatool, taking graphical examples to demonstrate important techniques used: testing, functional programming and delimited continuations.

Testing has received a lot of attention in this work, not only in the GUI. However, GUI testing requires additional considerations that have been addressed correctly, and two specific examples have been explained thoroughly.

Next chapter will focus on some results of this work.

The most important property of a program is whether it accomplishes the intention of its user.

C. A. R. Hoare

5 Results

Previous chapters considered theoretical foundations as well as practical glimpses of Bidiatool. In this chapter focus is on specific problems solved with Bidiatool.

Section 5.1 focus on a problem that even simple (from a "differential equations" point of view), represents a challenge for Niche PSO: branches of the bifurcation diagram get closer near the bifurcation at the origin, and niches should not merge more than necessary (*all* solutions should be given!).

Section **??** shows a bifurcation diagram obtained with Bidiatool where two parameters are varied at the same time.

Both examples are compared with corresponding bifurcation diagrams obtained with PyDSTool (a more "traditional" tool for bifurcation diagram plotting). Note that Bidiatool needs less input from the user (PyDSTool requires initial conditions).

Given the mathematical simplicity of the models chosen, some analytical solutions are given using Sage (an open source Computer Algebra System that uses Python as a scripting language).

5.1 Subcritical Pitchfork Bifurcation Diagram

The canonical example of a subcritical pitchfork bifurcation diagram is given by

$$\dot{x} = rx + x^3 - x^5 \tag{5.1}$$

The scalar measure used for the bifurcation diagram is simply x. Fixed points for (5.1) can be found in closed form. Using Sage Mathematics,

```
var('r')
g(x, r) = x*(r + x^2 - x^4)
roots = g.roots(x)
```

gives the following roots:

$$\begin{aligned} x &= \pm\sqrt{\frac{1}{2}\sqrt{4r+1}+\frac{1}{2}} \\ x &= \pm\sqrt{-\frac{1}{2}\sqrt{4r+1}+\frac{1}{2}} \\ x &= 0 \end{aligned} \tag{5.2}$$

Which, should give a bifurcation diagram like Fig. 5.1.1 (Red lines indicate unstable fixed points, and blue lines, stable ones).

Fig. 5.1.2 shows the corresponding bifurcation diagram obtained with Bidiatool. Black dots indicate points where the stability criteria used (the sign of the derivative of the right hand side of Eq. 5.1) is not enough to classify it (nothing can be said in general when $f'(x_0) = 0$ for the fixed point x_0, and a graphical analysis is required [18]).

From the bifurcation diagram, $(r, x) = (0, 0)$ is a subcritical pitchfork bifurcation (and the black dot gives us a hint that something special might be happening there). The other two black dots correspond to turning points (or saddle-node bifurcations).

The corresponding bifurcation diagram can be obtained with PyDSTool with Listing 5.1

Listing 5.1: Python Script for PyDSTool to get Fig. 5.1.3

```python
import PyDSTool as dst
import matplotlib.pyplot as plt

DSargs = dst.args(name='Canonical example of a subcritical pitchfork bifurcation')
DSargs.pars = {'r': -0.25}
```

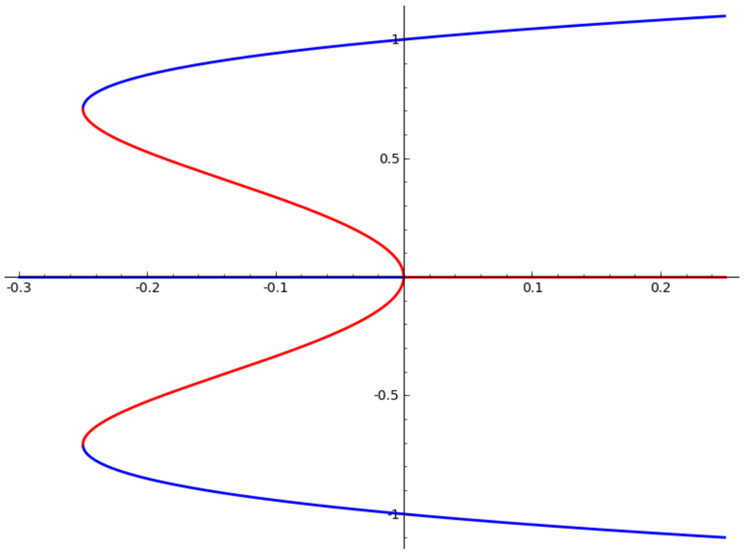


Figure 5.1.1: Expected bifurcation diagram for Eq. (5.1).

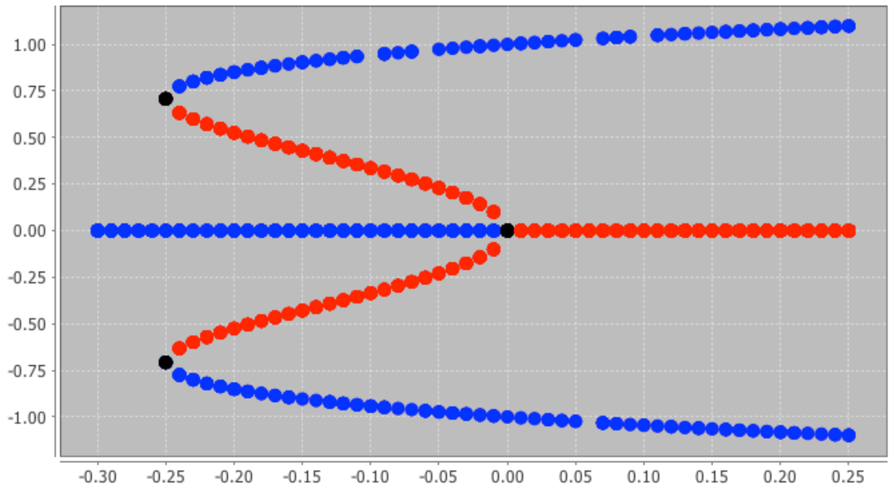


Figure 5.1.2: Bifurcation diagram for Eq. (5.1) obtained with Bidiatool.

```
DSargs.varspecs = {'x': 'r*x + x**3 - x**5', 'w': 'x - w'}
DSargs.ics = {'x': 0.5, 'w': 0}
ode = dst.Generator.Vode_ODEsystem(DSargs)
PC = dst.ContClass(ode)
PCargs = dst.args(name='SubcritPitchforkBif', type='EP-C')
PCargs.freepars = ['r']
PCargs.MaxNumPoints = 40
PCargs.MaxStepSize = 5e-2
```

```
14 PCargs.MinStepSize = 1e-3
15 PCargs.StepSize = 5e-3
16 PCargs.LocBifPoints = ['BP', 'LP']
17 PCargs.StopAtPoints = 'BP'
18 PCargs.SaveEigen = True
19
20 plt.clf()
21 plt.hold("on")
22 PC.newCurve(PCargs)
23 PC['SubcritPitchforkBif'].forward()
24 PC['SubcritPitchforkBif'].backward()
25 PC.display(['r', 'x'], stability=True)
26
27 ode.set(ics = {'x': -0.5, 'w': 0}, pars = {'r': -0.25})
28 PC2 = dst.ContClass(ode)
29 PC2.newCurve(PCargs)
30 PC2['SubcritPitchforkBif'].forward()
31 PC2['SubcritPitchforkBif'].backward()
32 PC2.display(['r', 'x'], stability=True)
33
34 ode.set(ics = {'x': 0.0, 'w': 0.0}, pars = {'r': -0.30})
35 PC3 = dst.ContClass(ode)
36 PCargs.MaxNumPoints = 20
37 PCargs.StopAtPoints = []
38 PC3.newCurve(PCargs)
39 PC3['SubcritPitchforkBif'].forward()
40 PC3.display(['r', 'x'], stability=True)
41 plt.grid('on')
42 plt.show()
```

Next we will highlight some important details of Listing 5.1 and make some comparison between the corresponding bifurcation diagram and the “methodology” to produce them.

- Line 6 describes the system given by Eq. (5.1). Note that a *dummy* variable `w` is needed for one-dimensional “systems”.

- Lines 5, 7, 27, 34 show a very important detail: in order to produce the “complete” bifurcation diagram of Fig. 5.1.3, PyDSTool needs to be provided with the suitable initial conditions corresponding to specific values of parameters. If this necessary guidance is somewhat hard for a relatively simple dynamical system, the harder it is for more complicated dynamical systems. This is where computational intelligence algorithms can help to simplify the required intervention of the user: “every” branch of the bifurcation diagram is found (for a given parameter value) using a niching or speciation algorithm.

- Continuation methods allow us to keep track of changes in the eigenvalues of fixed points along a curve, therefore giving us additional information about the kind of bifurcations points found (see the two turning points —or *limit points*, or saddle node bifurcations— and the *branch point* in Fig. 5.1.3). In contrast, Fig. 5.1.2 does not try to classify the fixed point based only in linearization in the neighbourhood of the equilibria. However, from the graphical analysis we can get the same conclusions. Here, the user must interpret the graphical output.

Both approaches have advantages and disadvantages, and the last observations suggest that instead of competing, they could be complementary and might even be fused in a hybrid algorithm that takes advantage of the strenghts of each method.

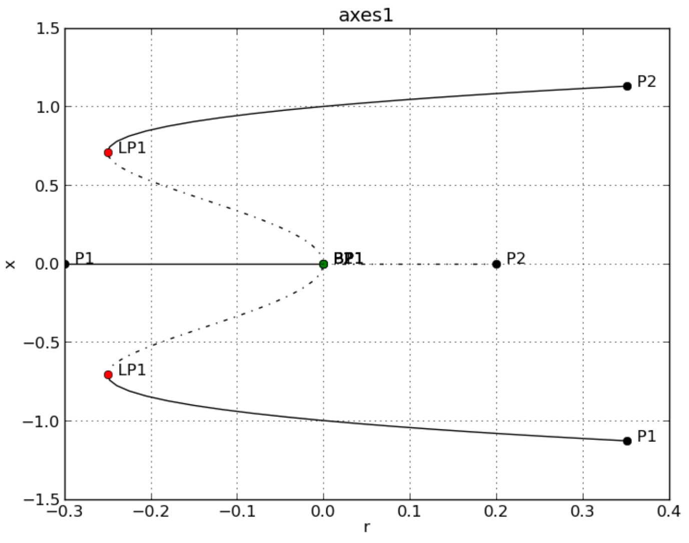


Figure 5.1.3: Bifurcation diagram for Eq. (5.1) obtained with PyDSTool.

5.2 Insect Outbreak

We used a dimensionless formulation of the insect outbreak model studied in [18]. The dynamical system is given by the scalar equation

$$\dot{x} = rx\left(1 - \frac{x}{k}\right) - \frac{x^2}{1 + x^2} \tag{5.3}$$

where r (dimensionless growth rate) and k (dimensionless carrying capacity) are parameters of the system. Again we are using x as the scalar measure for the bifurcation diagram (x is the dimensionless size of the insect population).

Again, in this relatively simple problem, we can find some fixed points using a CAS system, like Sage. Listing 5.2 was used to help testing the criteria of stability (we generated only real solutions and verified our stability qualification agreed with this computations)

Listing 5.2: Generation of fixed points for Eq. (5.3) using Sage

```
var('r k')
eq = r*(k - x)*(1 + x^2) - k*x
sols = eq.roots(x,multiplicities=false)
eq2 = r*x*(1 - x/k) - x^2/(1 + x^2)
deq2 = diff(eq2,x)
total, total_real, good_count = 0, 0, 0
for sol in sols:
    ss = [[r0, k0, sol.subs(r = r0, k = k0).N()]
           for r0 in [0.05*i for i in range(1, 11)]
           for k0 in [10.0*j for j in range(1, 4)]]
    print "*"*50
    total += len(ss)
    for i, s in enumerate(ss):
        if s[2].is_real():
            total_real += 1
            error = eq2.subs(r = s[0], k = s[1], x = s[2])
            if abs(error) < 1.0e-6:
                good_count += 1
                print (s[0].N(digits=2),
                       s[1].N(digits=2),
                       s[2],
                       deq2.subs(r = s[0], k = s[1], x = s[2]).N())

print "="*50
print "Total", total
print "Total real", total_real
print "Good count", good_count
```

The bifurcation diagram shown in Fig. 5.2.1 was obtained with Bidiatool, where axis x, y, z represent k, r and x respectively. From the diagram, we can see there are two saddle node bifurcation for each k from 10.0 to 40 (even when there are no "black dots").

A corresponding *part* of the latter diagram is obtained with PyDSTool in Listing 5.3. Note that only one parameter is varied and only one initial condition is given (see line 7).

Listing 5.3: Python script for PyDSTool to get Fig. 5.2.2

```
1  import PyDSTool as dst
2  import matplotlib.pyplot as plt
3
4  DSargs = dst.args(name='Insect Outbreak problem')
5  DSargs.pars = {'r': 0.6}
6  DSargs.varspecs = {'x': 'r*x*(1 - x/30.0) - x**2/(1 + x**2)', 'w': 'x - w'}
7  DSargs.ics = {'x': 15.0, 'w': 0.0}
8  ode = dst.Generator.Vode_ODEsystem(DSargs)
9  PC = dst.ContClass(ode)
10 PCargs = dst.args(name='InsectOutbreak', type='EP-C')
11 PCargs.freepars = ['r']
12 PCargs.MaxNumPoints = 40
13 PCargs.MaxStepSize = 1.0
14 PCargs.MinStepSize = 0.01
```

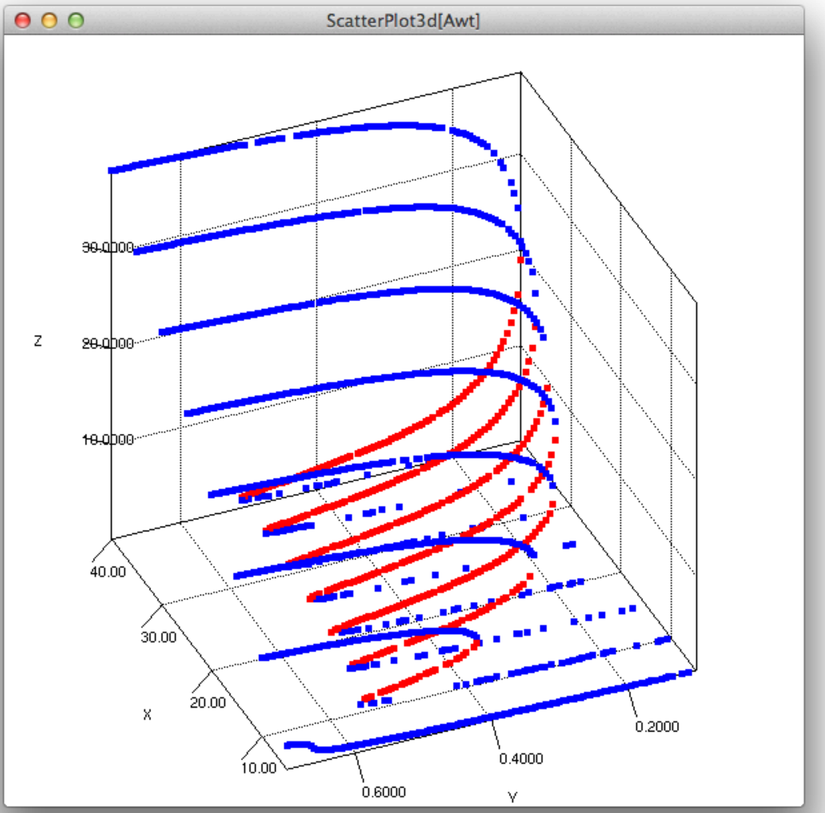


Figure 5.2.1: Bifurcation diagram with two parameters for Eq. (5.3).

```
PCargs.StepSize = 0.1
PCargs.LocBifPoints = ['LP', 'BP']
PCargs.StopAtPoints = 'BP'
PCargs.SaveEigen = True

plt.clf()
plt.grid("on")
PC.newCurve(PCargs)
PC['InsectOutbreak'].forward()
PC['InsectOutbreak'].backward()
PC.display(['r', 'x'], stability=True)

plt.show()
```

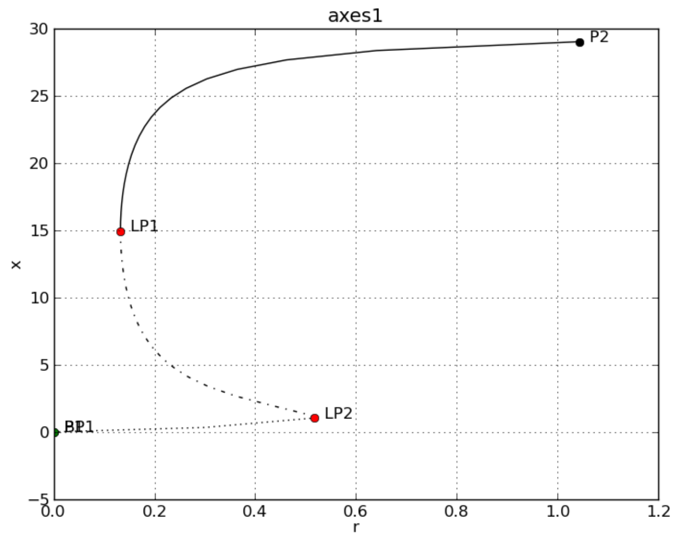


Figure 5.2.2: Bifurcation diagram for Eq. (5.3) obtained with PyDSTool. Note that only one parameter is varied.

5.3 Final Remarks

Two case studies have been used to compare the output given by PyDSTool (that can use continuation to find bifurcations) and Bidiatool. Whereas PyDSTool requires suitable initial conditions provided by the user, computational intelligence algorithms (such as Niche PSO) provide Bidiatool with the ability to get similar output with less guidance (studying only local bifurcation problems).

Both approaches have advantages and disadvantages, and this suggests that an hybrid solution should be better.

Niche PSO has been used successfully to solve a particular application (find fixed points) that requires the ability to search *and maintain* multiple solutions.

Education is a progressive discovery of our own ignorance.

Will Durant

6 Conclusions and further work

In this final chapter we address two main points: general conclusions (Section 6.1) and further work (Section 6.2).

Section 6.1 emphasizes the more important parameters identified for fine tuning of Niche PSO in the context of bifurcation diagrams. It also explains what has been used to successfully achieve several goals: parsing of dynamical systems, finding fixed points with computational algorithms, testing for correctness, and having a suitable build script for the application, among others.

Section 6.2 touches several possibilities for additional work. A modular design over the Java Platform can take advantage of the well stablished OSGi specification. On a very different direction, dynamic PSO variants are suggested to guide the search of fixed points once parameters are varied. Parameter variation is done in fixed steps, but a more sophisticated approach (a dynamical variation) could also be investigated.

6.1 General Conclusions

The field of dynamical systems is highly complex, but also fascinating, because it helps us understand a lot of phenomena around us.

In this work we have applied Niche PSO as an alternative way to find fixed points of dynamical systems. The important decisions the user of this algorithm has to make to get better bifurcation diagrams are:

- Choose an appropriate size for the main swarm.
- Choose a specific value for the threshold used in the `DiversityBasedMergeCriterion` so that we avoid losing solutions by excessive niche merging, at the same time that we allow some merging so that the new, larger subswarms benefit from the social information and experience of merged subswarms.
- Choose a suitable stopping condition for the algorithm. If we let the algorithm run "more than necessary", we lose some of the niches. On the other hand, if stopped prematurely, solutions have not been refined enough, and some of them may be "wrong".

Qualification of fixed points consists of two tasks

1. If linearization is suitable, each equilibrium point is classified as Asymptotically Stable or Unstable (if linearization is not appropriate, sometimes it is possible to classify it as Unstable, and for the rest we merely say it is Undefined).
2. For a hyperbolic fixed point, we also try to classify the linear conjugate flow near it.

This was done with traditional numerical eigenvalue methods.

We used Scala and Java (both running on top of the JVM) for the implementation:

1. Combinators Parsers and a PEG grammar to parse dynamical systems instead of the more traditional approach based on parser generators, CFGs and REs.
2. CIlib (written in Java) for implementations of swarm algorithms (PSO, Niche PSO, Guaranteed Convergence PSO) and some other facilities used in this work: velocity providers (contriction and clamping velocity providers), topologies (local best topology), niche detection and merging strategies (like a diversity based merging strategy), etc. At the moment, CIlib is mostly used as a framework, together with a simulator that is configured via XML. However we have done all configuration of the algorithms using CIlib from Scala code.
3. Numerical differentiation and jacobians were implemented in Scala code, but we are relying on EJML Java library for implementations of eigenvalue algorithms.

4. We took advantage of Scalaz, Shapeless and Spire (all of them advanced Scala libraries) in this work. Applicative functors, the reader monad, lenses, heterogenous lists, polymorphic functions, and numerical generic programming are some of the concepts we learned and applied. We strived for a functional programming style whenever possible, but we also relied on object oriented programming as well as a more imperative style when convenient. Maintainance and further development can be done using the paradigm the programmer feels more comfortable with.

5. Testing for correctness (using ScalaTest) has an important place in this work. Fest-Swing allowed us to test the GUI.

6. We rely on excellent libraries for plotting: JFreeChart (using scala-chart as a convenient wrapper) and Jzy3d.

7. A SBT build was programmed to be able to compile, package and test our code. Dependencies are handled automatically and downloaded from internet. The end user will need Java 1.7 installed in his/her machine, and nothing else.

6.2 Further Work

The Java language lacks advanced modularization support, as witnessed by the existance of the Jigsaw project. Some limitations are [9]:

- Low-level code visibility control.
- Error-prone class path concept.
- Limited deployment and management support

For the moment, the only realistic (and solid) alternative for true modularity for the Java platform is OSGi. A good architecture for Bidiatool could use OSGi bundles as units of modularisation. The OSGi framework provides functionality in the following layers [1]:

- Security layer.
- Module layer.
- Life cycle layer.
- Service layer.

- Actual services.

As a proof of concept, further work could be done exercising the module and life cicle layers, and then taking advantage of the service layer to provide different computational intelligence algorithms to find and maintain multiple solutions of optimization problems. This approach is far better than resorting to Java reflection (and we also tackle the aforementioned modularity limitations).

For the current implementation, Niche PSO is "restarted" every time we modify a parameter. This approach has advantages and disadvantages. We are not relying on (possibly innacurate) previous results to guide the search, so each time we give the algorithm the chance to search for *every* solution. For example, sometimes the algorithm doesn't find *all* solutions, but it could do it for the next value of parameters (that are generally close). However, in most situations, new solutions obtained with a small variation of parameters remain close to previous solutions, and we could take advantage of this to guide the search.

CIlib provides implementations for *dynamic* versions of PSO, that respond to "environment" changes. Parameter variation could correspond to environment changes and trigger adjustments in the best solutions. Such adjustment would be faster than starting the search without any guidance. A simple approach to mantain the hability to find multiple solutions could be to interleave normal niching iterations (starting without any guidance) with dynamic pso searches.

Currently, parameter are varied using a fixed step. That approach makes easier to miss a bifurcation point. A more elaborated solution could dynamically vary the step so that 1) bifurcations are not overlooked, 2) the step size is not chosen too small or too big.

A more traditional approach to bifurcation and stability analysis could be implemented on the JVM to make comparisons with our work.

Bibliography

[1] OSGi Alliance. Osgi service platform, core specification, release 5, version 5.0. *OSGi Specification*, 2012.

[2] R. Brits, A. P. Engelbrecht, and F. Van den Bergh. Locating multiple optima using particle swarm optimization. 2007. ISSN 0096-3003. URL http://researchspace.csir.co.za/dspace/handle/10204/801. http://www.sciencedirect.com/science/journal/00963003.

[3] J. E. Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Society for Industrial Mathematics, January 1987. ISBN 0898713641.

[4] Eusebius J. Doedel. AUTO: a program for the automatic bifurcation analysis of autonomous systems. *Congr. Numer*, 30:265–284, 1981. URL http://cmvl.cs.concordia.ca/publications/CongNum30.pdf.

[5] Andries P. Engelbrecht. *Computational intelligence: An introduction*. Wiley, 2007.

[6] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, January 2004. ISSN 0362-1340. doi: 10.1145/982962.964011. URL http://doi.acm.org/10.1145/982962.964011.

[7] James Gosling et al. *The Java® Language Specification, Java SE 7 Edition*. 2012.

[8] Dick Grune. *Modern compiler design*. Springer, New York, NY, 2012. ISBN 9781461446996 1461446996. URL http://dx.doi.org/10.1007/978-1-4614-4699-6.

[9] Richard Hall, Karl Pauls, Stuart McCulloch, and David Savage. *OSGi in action: Creating modular applications in Java*. Manning Publications Co., 2011.

[10] Daniel Hinojosa. *Testing in Scala*. O'Reilly Media, Inc., 2012. URL http://books.google.com.mx/books?hl=en&lr=&id=Ve2iwCbvKCEC&oi=fnd&pg=PR2&dq=Testing+in+Scala&ots=c68QQTnNYl&sig=67KOdBNOlQ82MKvhQGSbZhyFlrI.

[11] Eugen Labun. *Combinator Parsing in Scala*. Master's thesis, April 2012.

[12] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification, Java SE 7 Edition*. Technical Report JSR-000924, Oracle, 2012.

[13] Samir W Mahfoud. Niching methods for genetic algorithms. *Urbana*, 51(95001), 1995.

[14] Atif M. Memon. GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88, 2002. URL http://cvs.cs.umd.edu/~atif/papers/MemonIEEEComputer2002.pdf.

[15] C.J.F. Ridders. Accurate computation of f'(x) and f'(x) f"(x). *Advances in Engineering Software (1978)*, 4(2):75–76, April 1982. ISSN 0141-1195. doi: 10.1016/S0141-1195(82)80057-0. URL http://www.sciencedirect.com/science/article/pii/S0141119582800570.

[16] Rüdiger U. Seydel. *Practical Bifurcation and Stability Analysis*. Springer, December 2009. ISBN 9781441917393.

[17] Stephen Smale, Morris W. Hirsch, and Robert L. Devaney. *Differential Equations, Dynamical Systems, and an Introduction to Chaos, Second Edition*. Academic Press, 2 edition, November 2003. ISBN 0123497035.

[18] Steven H. Strogatz. *Nonlinear Dynamics And Chaos: With Applications To Physics, Biology, Chemistry And Engineering*. Westview Press, April 1994. ISBN 0201543443.

[19] Frans Van Den Bergh. *An analysis of particle swarm optimizers*. PhD thesis, University of Pretoria, 2006.

Colophon

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Arno Pro, designed by Robert Slimbach in the style of book types from the Aldine Press in Venice, and issued by Adobe in 2007. A template, which can be used to format a PhD thesis with this look and feel, has been released under the permissive MIT (X11) license, and can be found online at github.com/suchow/ or from the author at suchow@post.harvard.edu.