UNIVERSIDAD MICHOACANA
DE SAN NICOLÁS DE HIDALGO
FACULTAD DE INGENIERÍA ELÉCTRICA
DIVISIÓN DE ESTUDIOS DE POSGRADO

"BIFURCATION DIAGRAMS CONSTRUCTION BASED ON NICHE PSO AND FIXED POINTS QUALIFICATION"

TESIS

Que para obtener el grado de:
MAESTRO EN CIENCIAS EN INGENIERÍA ELÉCTRICA

Presenta:
Oscar Vargas Torres

Juan José Flores Romero, Ph.D. Computer Science
Director de Tesis

Jaime Cerda Jacobo, Ph.D. Computer Science
Co-director de Tesis

Agosto de 2013

BIFURCATION DIAGRAMS CONSTRUCTION BASED ON NICHE PSO AND FIXED POINTS QUALIFICATION

TESIS

Que para obtener el grado de

MAESTRO EN CIENCIAS EN INGENIERÍA ELÉCTRICA

presenta

Oscar Vargas Torres

Dr. Juan José Flores Romero

Director de Tesis

Dr. Jaime Cerda Jacobo

Co-Director de Tesis

Universidad Michoacana de San Nicolás de Hidalgo

Agosto 2013

To my son Alex Vargas Suárez

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 their supervision.

I thank my family for their patient and unconditional support.

Resumen

En campos tan diversos como la mecánica de fluidos, química, electrónica e ingeniería eléctrica, hay aplicaciones de lo que se conoce como análisis de bifurcación: el análisis de sistemas de ecuaciones diferenciales bajo la variación de parámetros.

Las herramientas tradicionales para el estudio de problemas de bifurcación incluyen métodos de continuación que requieren valores iniciales para su correcto funcionamiento. En el caso de bifurcaciones locales, la linealización en los puntos fijos del sistema proporciona información importante sobre la estabilidad del sistema.

En este trabajo se utiliza la optimización por enjambre de partículas con nichos (Niche PSO), como lo propusieron Brits et al. para encontrar y mantener múltiples puntos fijos correspondientes a los valores de los parámetros de los sistemas dinámicos. Para determinar la estabilidad de los puntos fijos se emplea la técnica de linealización.

El producto de este trabajo es BDT (Bifurcation Diagram Tool), una herramienta para el trazo de diagramas de bifurcación, que utiliza Niche PSO para encontrar los puntos fijos de sistemas dinámicos y la técnica de linealización para calificarlos.

Abstract

In scientific fields as diverse as fluid dynamics, chemistry, electronics, and electrical engineering, there is the application of what is known as bifurcation analysis: the analysis of a system of Ordinary Differential Equations (ODEs) under parameter variation.

Traditional tools required for parameter study in bifurcation problems include continuation methods that require initial values to work correctly. For local bifurcations linearisation at the fixed points of the sytem provides important information about the stability of the system.

This work uses Niche Particle Swarm Optimization as proposed by Brits et al. to locate and maintain multiple fixed points corresponding to parameter values of dynamical systems. To determine the stability of fixed points, linearization is used.

The product of this work is BDT a Bifurcation Diagram Tool, that uses Niche PSO to find the fixed points of dynamical systems and linearization to qualify them.

Contents

List of Figures

List of Tables

List of Algorithms

List of Symbols

θ	bifurcation parameter(s)
x	scalar variable
$\mathbf{x}$	vector of state variables, vector function, solution of an equation
t	time
$\dot{\mathbf{x}}$	derivative of $\mathbf{x}$ with respect to time, $\dot{\mathbf{x}} = d\mathbf{x}/dt$
$\mathbf{f}$	vector function, defines the dynamics of the problem that is to be solved
$\|\cdot\|$	a vector norm
$\mathbf{x}_s$	fixed point such that $\mathbf{f}(\mathbf{x}_s) = \mathbf{0}$
$\mathcal{X}_{\theta_0}$	Set of fixed points corresponding to $\theta = \theta_0$
$\mathbf{x}_0$	the value of $\mathbf{x}$ at $t = 0$, $\mathbf{x}(0) = \mathbf{x}_0$
I	identity matrix
λ	eigenvalue
$\in$	"in", element of a set
C^n	the set of functions with n continuous derivatives
δ, ϵ	real positive numbers
ϕ	solution of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$
$J(\mathbf{x}_s)$	Jacobian of $\mathbf{f}(\mathbf{x})$ evaluated at $\mathbf{x}_s$
T^{-1}	inverse matrix of T
C_1, C_2	constants that depend on initial conditions
$\mathbf{e}_j$	j-th unit vector
sgn	sign function

Chapter 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, heuristic methods are used as an alternative tool in bifurcation diagrams plotting. This kind of methods presents a number of advantages and disadvantages over traditional methods, which will be presented in Section 1.3.

The goal of this project is to implement a software that produces bifurcation diagrams using Niche Particle Swarm Optimization (Niche PSO), as well as the qualification of fixed points.

1.1 Preliminaries

For a first-order system of ordinary differential equations (ODEs)

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \theta) \tag{1.1}$$

A bifurcation diagram depicts a scalar measure $[\mathbf{x}]$ versus the real parameter θ, where $(\mathbf{x}, \theta)$ solves (1.1) [Seydel, 2009]. For example, $[\mathbf{x}]$ could be x_k (one of the components of the n-vector $\mathbf{x}$), or some convenient vector norm like $\|\mathbf{x}\|_2 = \|\mathbf{x}\|$ (the euclidean norm) or $\|\mathbf{x}\|_\infty = \max\limits_{1 \le i \le n} |x_i|$. Fixed points $\mathbf{x}_s$ of Eq. (1.1) are stationary solutions such that $\mathbf{f}(\mathbf{x}_s) = \mathbf{0}$.

Traditional tools required for parameter study in bifurcation problems are 1) continuation methods with devices for detecting bifurcation and checking stability; and 2) methods for switching from one branch to another with or without the option of calculating the bifurcation point itself [Seydel, 2009].

1.1.1 Finding Fixed Points as an Optimization Problem

The problem of finding fixed points of a dynamical system can be formulated as the optimization problem: Find a set of solutions $\mathcal{X}_\theta = \{\mathbf{x}_{s_1}, \mathbf{x}_{s_2}, \ldots, \mathbf{x}_{s_n}\}$, such that each $\mathbf{x}_s \in \mathcal{X}_\theta$ is a minimum of

$$\mathbf{g}_1(\mathbf{x}) = \|\mathbf{f}(\mathbf{x})\| \tag{1.2}$$

for a given value of θ.

The optimization problem can use different objective functions. Instead of a minimization problem, a maximization problem could use

$$\mathbf{g}_2(\mathbf{x}) = \frac{1}{1 + \|\mathbf{f}(\mathbf{x})\|} \tag{1.3}$$

as done in [López Cuevas Villanueva, 2010] and [Barrera et al., 2008]. $\mathbf{g}_1$ is simpler than $\mathbf{g}_2$ (and therefore more convenient in terms of computation time), but $\mathbf{g}_2$ or another function that corresponds to a maximization problem (e.g. $\mathbf{g}_3(\mathbf{x}) = -\|\mathbf{f}(\mathbf{x})\|$) might be necessary when the software used can solve maximization problems only.

$\mathbf{g}_2(\mathbf{x})$ takes values in the range $[0, 1]$. However, in general $\mathbf{f}(\mathbf{x})$ is not a linear function of $\mathbf{x}$. Using $\mathbf{g}_2(\mathbf{x})$ introduces more non-linearities. To illustrate this point, let

$$h(y) = \frac{1}{1+y} \tag{1.4}$$

where $y = \|\mathbf{f}(\mathbf{x})\|$ for some $\mathbf{x}$. Notice this is simply another way of representing Eq. (1.3). Fig. 1.1 shows $h(y)$ and its derivative $h'(y)$.

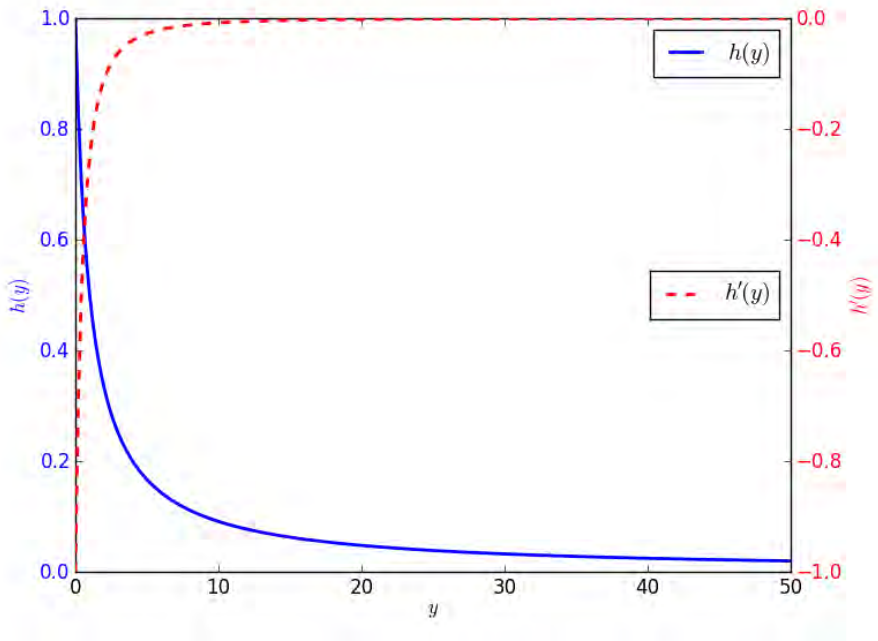

Figure 1.1: Non-linearities introduced by Equation (1.4).

Let $y_1 = \|\mathbf{f}(\mathbf{x}_1)\| = 25$ and $y_2 = \|\mathbf{f}(\mathbf{x}_2)\| = 50$, for some vectors $\mathbf{x}_1$ and $\mathbf{x}_2$. A really small improvement of $h(y_1) - h(y_2) \approx 1.885 \times 10^{-2}$ is obtained for $y_2/y_1 = 2$. Fig. 1.1 also shows that the derivative $h'(y)$ takes values very close to zero for $y \geq 10$. For $y \leq 10$, $|h'(y)| \leq 1.0$.

The linear scale that $\mathbf{g}_1(\mathbf{x}) = \|\mathbf{f}(\mathbf{x})\| = y$ provides makes much more sense: if $y_2/y_1 = 2$, then it is immediately obvious that $\mathbf{x}_2$ (with norm y_2) is twice far away from the origin than $\mathbf{x}_1$ (with norm y_1), *no matter how big* y_1 *and* y_2 *are*.

1.1.2 Finding Multiple Fixed Points for the Same θ

For a given value of θ, $\mathbf{f}(\mathbf{x}_s) = \mathbf{0}$ might have more than one solution (fixed points $\mathbf{x}_s$). As an illustration, Fig. 1.2 shows the bifurcation diagram for the simple system

$$\dot{x} = f(x, \theta) = \theta x + x^3 - x^5 \tag{1.5}$$

For some values of the parameter θ, there are multiple fixed points x_s such that $f(x_s) = 0$. For example, for $\theta = -0.1$, $\mathcal{X}_{-0.1} = \{-0.941965, -0.335711, 0.0, 0.335711, 0.941965\}$ (five fixed points for the same θ).

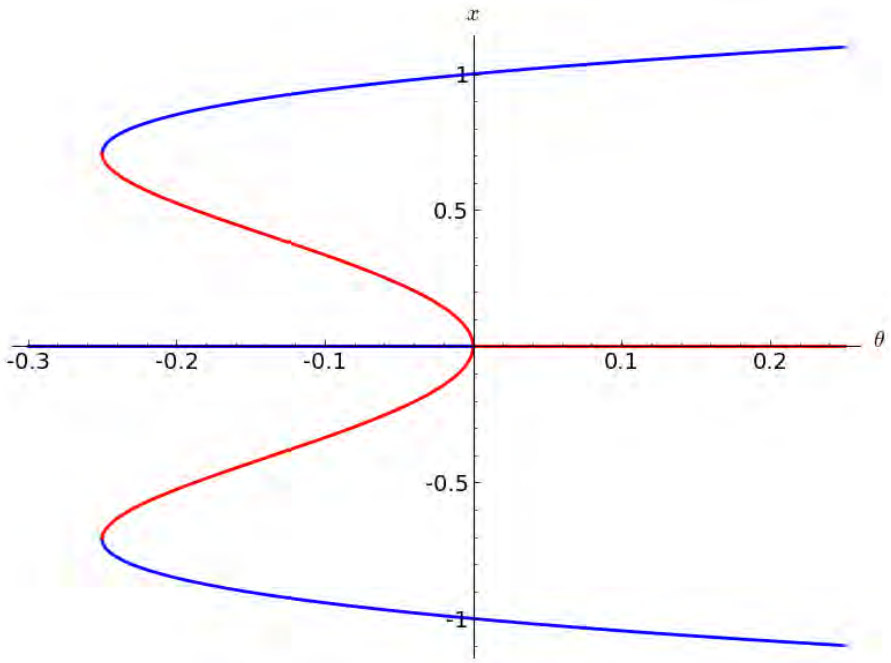

Figure 1.2: Bifurcation diagram for Equation (1.5).

Traditional numerical open-interval methods for finding roots (like Newton-Raphson) require an approximation of *one* of the roots. In general *one value* is returned (the nearest root to the given initial approximation).

Niching methods in genetic algorithms tackle problems that require the location and maintenance of multiple solutions (classification and machine learning, multimodal

function optimization, multiobjective function optimization and simulation of complex and adaptive systems) [Mahfoud, 1995].

This work uses Niche PSO as originally proposed in [Brits et al., 2002] to locate multiple optimal solutions for multimodal optimization problems (such as finding multiple fixed points for the same θ in dynamical systems). Niche PSO can be classified as a *parallel* niching algorithm because the niches are identified and maintained *simultaneously*. In contrast, *sequential* niching methods identifies multiple solutions by adapting the objective function's fitness landscape through the application of a derating function at a position where a potential solution was found.

Niche PSO has the advantage that it does not require of approximations of roots to start the search. This means the researcher of dynamical systems can benefit from this ability to simplify the construction of bifurcation diagrams.

1.2 Problem Definition

The problem that this work solves is the following:

> Construct bifurcation diagrams as complete as possible using Niche PSO to find fixed points of dynamical systems. Information about the stability of fixed points is found using linearization near the fixed point.

1.3 State of the Art

A (necessarily) incomplete list of software devoted to the study of dynamical systems and bifurcation problems is given in [Seydel, 2009]. Among the most prominent are AUTO [Doedel, 1981] and XPP/XPPAUT (XPPAUT provides an interface to AUTO, which is a software for continuation and bifurcation problems in ordinary differential equations) [Ermentrout, 2002].

AUTO has become a standard package in bifurcation analysis. The first distributed version appeared in 1980, and version 0.9.1 of AUTO-07P was released in 2012. AUTO continues to be used and developed up to the moment of writing.

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 Command Line User Interface (CLUI, based on the Python read-eval-print loop interpreter) 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[1] AUTO commands). Usability is an issue here.
- The documentation is distributed in PDF form, but some of it requires an installation of a TeX distribution *and* transfig. HTML documentation is non-existent. Again, portability and usability are concerns.
- A very unusual file naming convention is used.

XPP/XPPAUTO contains the code for AUTO, and makes its usage easier. However, it does not 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.

[1] these commands run both directly in the shell (in Windows the user needs to install MinGW, the "Minimalist GNU for Windows" also) and at the AUTO Python prompt

PyDSTool [Clewley, 2012] provides another alternative for bifurcation and stability analysis (PyDSTool features optional support for AUTO and a C-based integrator). If the user wants a feature-complete installation, he/she will face similar challenges as mentioned before for AUTO. 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 Ubuntu Linux machine, one possibility is to 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.

From the previous summary, the following should be clear by now: AUTO continues to be used (in some way) in a lot of open source tools for bifurcation and stability analysis (e.g. XPPAUT, PyDSTool). This means a lot of Fortran and C legacy code involved, with their respective advantages (e.g. performance) and disadvantages (e.g. portability).

Given the aforementioned usability and portability concerns for AUTO, a tool that is both easier to use and install is proposed: BDT (Bifurcation Diagram Tool). Some of the *advantages* it offers are: the user requires little previous knowledge of the behavior of dynamical systems to construct bifurcation diagrams; bifurcation diagrams can be generated in an unsupervised way; the ability to work with non-continuous functions; great portability provided by the usage of the Java Virtual Machine (JVM).

BDT has one important disadvantage: the computation time required for solving optimization problems with Niche PSO can be large. However, a user of AUTO (or one of its offspring) has to spend a lot of time constructing bifurcation diagrams with continuation methods (initial conditions required).

Harold Abelson from the Massachusetts Institute of Technology (MIT), wrote an article for a "Bifurcation Interpreter" that used artificial intelligence together with the Newton method to study dynamical systems [Abelson, 1990].

An hybrid Particle Swarm Optimization method was used in [Flores et al., 2011] to construct complete bifurcation diagrams to study voltage collapse phenomenon. One of the important contributions of this work is the ability to build complete bifurcation diagrams, varying 2 or more parameters at the same time.

In [López Cuevas Villanueva, 2010], another tool for bifurcation diagram plotting is reported. The default metaheuristic used was developed by Julio Barrera in his doctoral thesis [Barrera Mendoza, 2012]. One of the important contributions of this work is the implementation of a compiler for differential equations with ANTLR. The compiler generates code that is interpreted at run-time using BeanShell scripts.

1.4 Objectives

The purpose of this thesis is the implementation of a tool (BDT) for construction of bifurcation diagrams using Niche PSO. BDT must meet the following requirements:

- To allow the definition of dynamical systems.
- Generate bifurcation diagrams in 2D and 3D using Niche PSO.
- To allow the qualification of the stability of fixed points.
- Run on the Java Virtual Machine

1.5 Description of Chapters

The rest of this thesis is organized as follows:

- Chapter 2 introduces the dynamical systems terminology and stablishes the criteria used for classification of fixed points.

- Chapter 3 introduces PSO and then introduces the special variant, Niche PSO, that was developed to locate and maitain multiple optima. A description of the configuration of the algorithm is given.
- Chapter 4 describes the process of plotting a bifurcation with BDT. 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 BDT and PyDSTool (that can use AUTO for continuation methods).
- Chapter 6 summarizes the main achievements of this thesis and suggests further work.
- Appendix A defines a Parsing Expression Grammar (PEG) to recognize dynamical systems. It mentions how this can be implemented in Scala with combinator parsers.

Chapter 2

Dynamical Systems

In this chapter basic dynamical system terminology is established (Section 2.1). Section 2.2 states the noteworthy Linearization Theorem that constitutes the basis of our qualification of fixed points criteria. Section 2.2.1 is an illustration of the application of some concepts given in Section 2.2 in a *nonlinear* problem. Section 2.3 establishes the qualification criteria of fixed points. Section 2.4 summarizes two numerical algorithms used for qualification of fixed points.

2.1 Basic Terminology

This section introduces the dynamical systems terminology used in this work. Section 2.1.1 shows the kind of differential equations studied in this thesis, whereas Section 2.1.2 reviews the basic definition of eigenvalue and eigenvectors. Section 2.1.3 defines what stability and linearization mean.

2.1.1 Differential Equations and Fixed Points

An *autonomous first-order system of differential equations* is a collection of n interrelated differential equations:

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

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

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

A vector $\mathbf{x}_s$ for which $\mathbf{f}(\mathbf{x}_s, \theta) = \mathbf{0}$ is called *stationary solution*. These solutions are also called *fixed points*, and sometimes equilibrium points, singular points, critical points, or rest points. In general, solutions of Eq. (2.1) vary with the *parameter* θ.

For more information about terminology for differential equations and dynamical systems see [Strogatz, 1994], [Smale et al., 2003] and [Seydel, 2009]).

2.1.2 Eigenvalues and Eigenvectors

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

$$A\mathbf{v} = \lambda\mathbf{v} \tag{2.2}$$

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 (2.2) and is called the trivial eigenvector of A belonging to λ.

Equation (2.2) can be rewritten as

$$(A - \lambda I)\mathbf{v} = \mathbf{0} \tag{2.3}$$

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 polynomial*

$$\det(A - \lambda I) = 0 \tag{2.4}$$

To find the eigenvectors, substitute the eigenvalues resulting from (2.4), one after the other, into (2.3), and solve for the unknown vector $\mathbf{v}$.

2.1.3 Stability and Linearization

Suppose $\mathbf{f}(\mathbf{x})\colon \mathbb{R}^n \to \mathbb{R}^n$, is continuously differentiable in all $\mathbf{x}$-space ($\mathbf{f} \in \mathcal{C}^1$). Then, for any $\mathbf{x}_0$, the initial value problem

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}), \quad \mathbf{x}(0) = \mathbf{x}_0$$

has a unique solution $\mathbf{x}(t, \mathbf{x}_0)$. The usual stability definitions are stated as follows:

- The system is *stable* at an equilibrium point $\mathbf{x}_s$, if for each positive number ϵ there is a positive number δ such that for all solutions $\mathbf{x}(t, \mathbf{x}_0)$:

$$\text{if } \|\mathbf{x}_0 - \mathbf{x}_s\| < \delta, \text{then } \|\mathbf{x}(t, \mathbf{x}_0) - \mathbf{x}_s\| < \epsilon, \text{ for all } t \geq 0$$

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

- The system is *neutrally stable* at $\mathbf{x}_s$ if it is stable at $\mathbf{x}_s$, but not asymptotically stable.

- The system is *unstable* at $\mathbf{x}_s$ if it is not stable.

The stability properties of an autonomous *linear* system $\dot{\mathbf{x}} = A\mathbf{x}$ can be completely determined by a study of the eigenvalues of A and its corresponding eigenspaces. For *nonlinear* autonomous systems, it is possible to determine the stability of the dynamical system at the fixed points via *linearization* or with Liapunov functions[1].

In the present work, the stability of fixed points is classified only if linearization is suitable. If $\mathbf{f}$ is twice continuously differentiable ($\mathbf{f} \in \mathcal{C}^2$), $\mathbf{x}_s$ is a fixed point of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$, and $J(\mathbf{x}_s)$ is the Jacobian of $\mathbf{f}$ evaluated at $\mathbf{x}_s$, then the linear system $\dot{\mathbf{x}} = J(\mathbf{x}_s)(\mathbf{x} - \mathbf{x}_s)$ is the *linearization* of $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$.

- The system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ is *asymptotically stable* at $\mathbf{x}_s$ if all eigenvalues of $J(\mathbf{x}_s)$ have negative real parts.

[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 [Borrelli and Coleman, 2004].

- The system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ is unstable at $\mathbf{x}_s$ if $J(\mathbf{x}_s)$ has at least one eigenvalue with a positive real part.

It is not possible to draw any conclusion about stability (*using linearization*) if $J(\mathbf{x}_s)$ has an eigenvalue with a zero real part, but no eigenvalue with a positive real part (non-linear order terms determine the stability).

2.2 Linear Systems and their Relation to Nonlinear Systems

A matrix A is hyperbolic if none of its eigenvalues has real part 0. In that case, the *linear* system $\dot{\mathbf{x}} = A\mathbf{x}$ is also said to be hyperbolic.

In analogy with linear systems, a fixed point $\mathbf{x}_s$ of a *nonlinear* system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$ is hyperbolic if all of the eigenvalues of $J(\mathbf{x}_s)$ have nonzero real parts.

To emphasize the dependence of solutions on both time and the initial conditions $\mathbf{x}_0$, let $\phi(t, \mathbf{x}_0)$ denote the solution that satisfies the initial condition $\mathbf{x}_0$. That is, $\phi(0, \mathbf{x}_0) = \mathbf{x}_0$. The function $\phi(t, \mathbf{x}_0)$ is called the flow of the differential equation.

Suppose $\dot{\mathbf{x}} = \mathbf{f}_A\mathbf{x}$ and $\dot{\mathbf{x}} = \mathbf{f}_B\mathbf{x}$ have flows ϕ_A and ϕ_B. These two systems are (topologically) conjugate if there exists a homeomorphism[2] $\mathbf{h}\colon \mathbb{R}^n \to \mathbb{R}^n$ that satisfies

$$\phi_B(t, \mathbf{h}(\mathbf{x}_0)) = \mathbf{h}(\phi_A(t, \mathbf{x}_0))$$

The homeomorphism $\mathbf{h}$ is called a conjugacy. Thus a conjugacy takes the solution curves of $\dot{\mathbf{x}} = \mathbf{f}_A\mathbf{x}$ to those of $\dot{\mathbf{x}} = \mathbf{f}_B\mathbf{x}$.

These concepts are important because of the next theorem [Smale et al., 2003]:

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

[2] A homeomorphism is a one-to-one, onto, and continuous function whose inverse is also continuous.

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. As a consequence, an understanding of linear systems is useful to study nonlinear phenomena.

2.2.1 An Illustration of the Linearization Theorem

The horizontal motion of a weight attached to a nonlinear soft spring can be modeled by an ODE that looks like:

$$\ddot{x} + 2c\dot{x} + (\omega_0^2 - d^2x^2)x = 0 \tag{2.5}$$

where x is the horizontal displacement from the equilibrium position, and $x > 0$ corresponds to a compressed spring. Hooke's Law models $S(x)$, the force exerted by the spring, as the linear function $-kx$ that acts opposite to the direction of the displacement. But here the case where the "spring constant" k weakens as the displacement x increases is being considered. Therefore

$$S(x) = -(\omega_0^2 - d^2x^2)x, \quad \omega_0 \text{ and } d \text{ are positive constants}$$

A damping force $-2c\dot{x}$ is exerted over the spring, and it is opposite to the direction of displacement of the weight (c is a positive number also). Equation (2.5) can be rewritten as a first-order system of ODEs in normal form:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -\omega_0^2 x - 2cv + d^2x^3 \end{aligned} \tag{2.6}$$

If $\mathbf{x}$ denotes the state vector $[x, v]^T$, the system (2.6) can be expressed in matrix form as

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2c \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ d^2x^3 \end{bmatrix} \tag{2.7}$$

From Equation (2.6), it follows that fixed points of the nonlinear system are

$$\mathbf{x}_{s_1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \mathbf{x}_{s_2} = \begin{bmatrix} \omega_0/d \\ 0 \end{bmatrix}, \quad \mathbf{x}_{s_3} = \begin{bmatrix} -\omega_0/d \\ 0 \end{bmatrix}$$

For $\mathbf{x}_{s_1}$

$$J(\mathbf{x}_{s_1}) = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2c \end{bmatrix} \tag{2.8}$$

and the *linearization* of the system (2.7) near $\mathbf{x}_{s_1}$ is merely

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2c \end{bmatrix} \mathbf{x} \tag{2.9}$$

The characteristic polynomial (see Equation (2.4)) of $J(\mathbf{x}_{s_1})$ is

$$\lambda^2 + 2c\lambda + \omega_0^2 = 0 \tag{2.10}$$

with roots:

$$\begin{aligned} \lambda_1 &= -c + \sqrt{c^2 - \omega_0^2} \\ \lambda_2 &= -c - \sqrt{c^2 - \omega_0^2} \end{aligned} \tag{2.11}$$

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.

From the previous discussion, it follows that the nonlinear system (2.7) has an equilibrium point at $\mathbf{x}_{s_1}$ that is hyperbolic (neither λ_1 nor λ_2 have real part 0). Therefore, not only it is possible to say that the nonlinear system is asymptotically stable at $\mathbf{x}_{s_1}$ (both λ_1 and λ_2 have negative real part), but because of the Linearization Theorem, it is also possible to say that the nonlinear flow of system (2.7) resembles that of the linearized system near $\mathbf{x}_{s_1}$.

The exact nature of the solutions for system (2.9) 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 different.
- $c = \omega_0$. Then $\lambda_1 = \lambda_2 = -c < 0$

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

In order to find the eigenvectors corresponding to $\lambda_{1,2}$, substitute $\lambda_{1,2}$ (one at a time) into $(J(\mathbf{x}_{s_1}) - \lambda I)\mathbf{v} = \mathbf{0}$, and solve for the unknown $\mathbf{v}$.

Overdamped System: $c > \omega_0$

For λ_1, $(J(\mathbf{x}_{s_1}) - \lambda_1 I)\mathbf{v} = \mathbf{0}$ can be written as

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

and solutions are of the form

$$\mathbf{v}_1 = k_1 \begin{bmatrix} 1 \\ -c + \sqrt{c^2 - \omega_0^2} \end{bmatrix} \quad k_1 \neq 0 \tag{2.12}$$

For λ_2,

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

and solutions are of the form

$$\mathbf{v}_2 = k_2 \begin{bmatrix} 1 \\ -c - \sqrt{c^2 - \omega_0^2} \end{bmatrix} \quad k_2 \neq 0 \tag{2.13}$$

The matrix T whose columns are the eigenvectors of $J(\mathbf{x}_{s_1})$ is:

$$T = \begin{bmatrix} 1 & 1 \\ -c + \sqrt{c^2 - \omega_0^2} & -c - \sqrt{c^2 - \omega_0^2} \end{bmatrix} \tag{2.14}$$

with inverse

$$T^{-1} = \frac{1}{2\sqrt{c^2 - \omega_0^2}} \begin{bmatrix} c + \sqrt{c^2 - \omega_0^2} & 1 \\ -c + \sqrt{c^2 - \omega_0^2} & -1 \end{bmatrix} \tag{2.15}$$

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

$$\dot{\mathbf{y}} = \begin{bmatrix} -c + \sqrt{c^2 - \omega_0^2} & 0 \\ 0 & -c - \sqrt{c^2 - \omega_0^2} \end{bmatrix} \mathbf{y} \tag{2.16}$$

with solution:

$$\mathbf{y}(t) = C_1 e^{(-c+\sqrt{c^2-\omega_0^2})t} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + C_2 e^{(-c-\sqrt{c^2-\omega_0^2})t} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.17}$$

The solution to $\dot{\mathbf{x}} = J(\mathbf{x}_{s_1})\mathbf{x}$ is given by $T\mathbf{y}$:

$$\mathbf{x}(t) = C_1 e^{(-c+\sqrt{c^2-\omega_0^2})t} \begin{bmatrix} 1 \\ -c+\sqrt{c^2-\omega_0^2} \end{bmatrix} + C_2 e^{(-c-\sqrt{c^2-\omega_0^2})t} \begin{bmatrix} 1 \\ -c-\sqrt{c^2-\omega_0^2} \end{bmatrix} \tag{2.18}$$

or in a more compact way:

$$\mathbf{x}(t) = C_1 e^{\lambda_1 t} \begin{bmatrix} 1 \\ \lambda_1 \end{bmatrix} + C_2 e^{\lambda_2 t} \begin{bmatrix} 1 \\ \lambda_2 \end{bmatrix} \tag{2.19}$$

The equilibrium point $\mathbf{x}_{s_1}$ is a *sink*, because both eigenvalues are real, negative and different.

Given the initial condition $\mathbf{x}(0) = \begin{bmatrix} x_0 \\ v_0 \end{bmatrix}$, C_1 and C_2 can be computed with

$$\begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = T^{-1}\mathbf{x}(0) \tag{2.20}$$

Figure 2.1 shows the flow of the linearized system near $\mathbf{x}_{s_1}$. It was drawn using $c = 2.0$, $\omega_0 = 1.0$, and initial conditions from the sequence:

$$\left(\begin{bmatrix} -3.0 \\ -4.0 \end{bmatrix}, \begin{bmatrix} -3.0 \\ 4.0 \end{bmatrix}, \begin{bmatrix} -2.0 \\ -4.0 \end{bmatrix}, \begin{bmatrix} -2.0 \\ 4.0 \end{bmatrix}, \ldots, \begin{bmatrix} 3.0 \\ 4.0 \end{bmatrix} \right) \tag{2.21}$$

According to the Linearization Theorem, the nonlinear flow of the system (2.7) is conjugate to the flow of the linearized system near $\mathbf{x}_{s_1}$.

Critically damped: $c = \omega_0$

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

$$J(\mathbf{x}_{s_1}) = \begin{bmatrix} 0 & 1 \\ -c^2 & -2c \end{bmatrix} \tag{2.22}$$

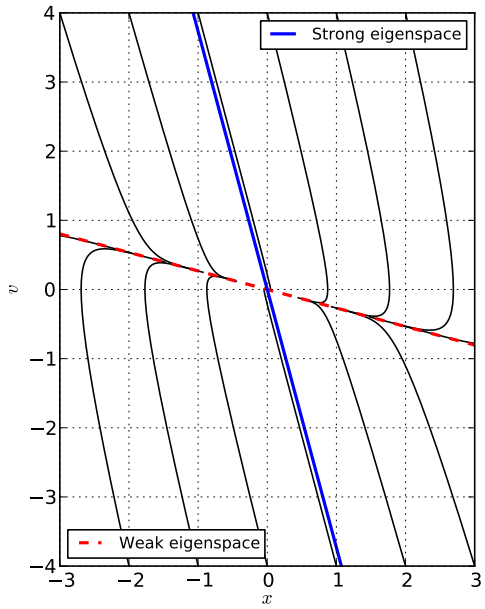

Figure 2.1: Sink in phase space.

The characteristic polynomial is $\lambda^2 + 2c\lambda + c^2 = 0$, with eigenvalues $\lambda_1 = \lambda_2 = -c$. Substituting these in $(J(\mathbf{x}_{s_1}) - \lambda I)\mathbf{v} = \mathbf{0}$ gives

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

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 $J(\mathbf{x}_{s_1})\mathbf{w} = \mu\mathbf{v} - c\mathbf{w}$ 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 $\mathbf{u}$

$$T = \begin{bmatrix} 1 & c/(c^2+1) \\ -c & 1/(c^2+1) \end{bmatrix} = \frac{1}{c^2+1}\begin{bmatrix} c^2+1 & c \\ -c(c^2+1) & 1 \end{bmatrix} \tag{2.23}$$

with inverse

$$T^{-1} = \frac{1}{c^2+1}\begin{bmatrix} 1 & -c \\ c(c^2+1) & c^2+1 \end{bmatrix} \tag{2.24}$$

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

$$\dot{\mathbf{y}} = \begin{bmatrix} -c & 1 \\ 0 & -c \end{bmatrix}\mathbf{y} \tag{2.25}$$

The general solution may be written as

$$\mathbf{y}(t) = C_1 e^{-ct}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + C_2 e^{-ct}\begin{bmatrix} t \\ 1 \end{bmatrix}$$

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

$$\mathbf{x}(t) = C_1 e^{-ct}\begin{bmatrix} 1 \\ -c \end{bmatrix} + C_2 e^{-ct}\begin{bmatrix} t + c/(c^2+1) \\ -tc + 1/(c^2+1) \end{bmatrix} \tag{2.26}$$

The equilibrium point $\mathbf{x}_{s_1}$ is a *deficient node*, because there is only a double eigenvalue (algebraic multiplicity is two), but its eigenspace has dimension one.

Figure 2.2 was drawn using $c = \omega_0 = 1.0$, and initial conditions from sequence (2.21).

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 (2.10), but now the roots are complex conjugates:

$$\begin{aligned} \lambda_1 &= -c + i\sqrt{\omega_0^2 - c^2} \\ \lambda_2 &= -c - i\sqrt{\omega_0^2 - c^2} \end{aligned} \tag{2.27}$$

$(J(\mathbf{x}_{s_1}) - \lambda_1 I)\mathbf{v} = \mathbf{0}$ is

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

and solutions are multiple (non-zero) of

$$\mathbf{v} = \begin{bmatrix} 1 \\ -c + i\sqrt{\omega_0^2 - c^2} \end{bmatrix} \tag{2.28}$$

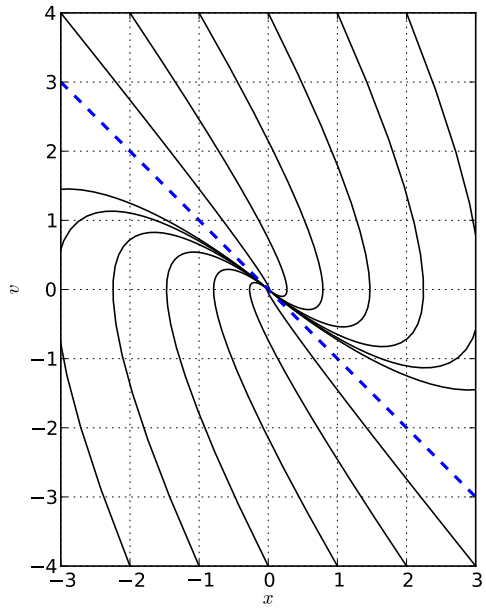

Figure 2.2: Deficient node in phase space.

Let

$$\mathbf{w}_1 = \frac{1}{2}(\mathbf{v}_1 + \overline{\mathbf{v}}_2) = \begin{bmatrix} 1 \\ -c \end{bmatrix}$$

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

These vectors are linearly independent, and a similarity transformation $T^{-1}J(\mathbf{x}_{s_1})T$ can be built with

$$T = \begin{bmatrix} 1 & 0 \\ -c & \sqrt{\omega_0^2 - c^2} \end{bmatrix} \tag{2.29}$$

that has inverse

$$T^{-1} = \frac{1}{\sqrt{\omega_0^2 - c^2}} \begin{bmatrix} \sqrt{\omega_0^2 - c^2} & 0 \\ c & 1 \end{bmatrix} \tag{2.30}$$

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

$$\mathbf{y}(t) = C_1 e^{-ct} \begin{bmatrix} \cos\sqrt{\omega_0^2 - c^2}t \\ -\sin\sqrt{\omega_0^2 - c^2}t \end{bmatrix} + C_2 e^{-ct} \begin{bmatrix} \sin\sqrt{\omega_0^2 - c^2}t \\ \cos\sqrt{\omega_0^2 - c^2}t \end{bmatrix} \tag{2.31}$$

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

$$\mathbf{x}(t) = e^{-ct} \begin{bmatrix} \cos \beta t & \sin \beta t \\ -c \cos \beta t - \beta \sin \beta t & -c \sin \beta t + \beta \cos \beta t \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} \tag{2.32}$$

The phase portrait corresponds to a spiral sink. Figure 2.3 was draw using $c = 0.5$, $\omega_0 = 1.0$, and initial conditions from sequence (2.21).

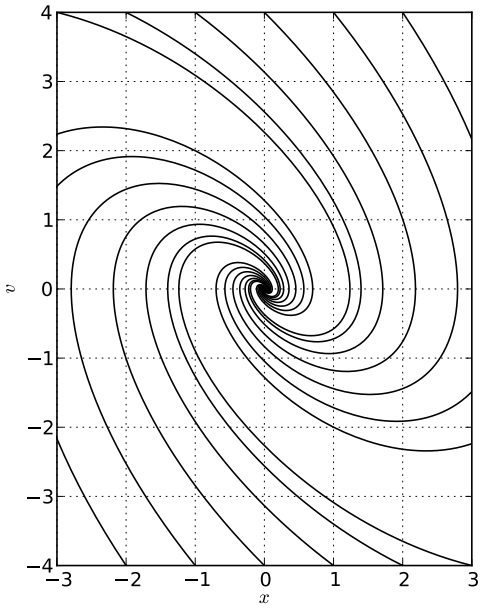

Figure 2.3: Spiral sink in phase space.

2.3 Criteria for Qualification of Fixed Points

A linear system like (2.9) can be completely described with the eigenvalues and eigenvectors (besides stability, it is possible to determine analytically the exact shape of the linear flow in the neighbourhood of the fixed point). This is not the case for nonlinear systems.

For nonlinear systems the process followed in this work consists of a few simple steps:

- Find the fixed points of the system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \theta)$ (an alternative to do this will be studied in the next chapter).
- Evaluate the jacobian of $\mathbf{f}$ in every fixed point $\mathbf{x}_s$: $J(\mathbf{x}_s)$ (linearization is used here).
- Compute the eigenvalues of $J(\mathbf{x}_s)$.
  - If the fixed point $\mathbf{x}_s$ is hyperbolic, the linear conjugate flow can be used to describe the behaviour in its neighbourhood (see Table 2.1). By consequence, the fixed point can be classified as "Asymptotically Stable" or "Unstable".
  - Table 2.1 shows 10 possible cases depending on four boolean flags: `isHyperbolic`, `negativeFound`, `positiveFound`, `complexFound`. The boolean flag `isHyperbolic` is set to `true` if every eigenvalue of $J(\mathbf{x}_s)$ has non-zero real part; `negativeFound` (`positiveFound`) is set to true if there is at least one eigenvalue with negative (positive) real part; `complexFound` is set to `true` if there is at least one eigenvalue that is complex.

    Notice that linearization gives information about the linear conjugate flow only if the fixed point is hyperbolic (column "Linear Conjugate Flow" of the table).
  - If the fixed point $\mathbf{x}_x$ is not hyperbolic, acording to Section 2.1.3, it could be classified as "Unstable" or "Undefined". The latter case occurs when there is not enough information from the linearization process, and stability classification requires an analysis of higher order terms.

2.4 Numerical Computing of Derivatives and Jacobians

Acording to Section 2.3, it is necessary to evaluate $J(\mathbf{x}_s)$. For dynamical systems consisting of one scalar equation, it is enough to check the sign of the derivative evaluated at the fixed point. A description of the algorithms used for this purpose follows.

Table 2.1: Criteria used for qualification of fixed points

Case Number	**isHyperbolic**	**negativeFound**	**positiveFound**	**complexFound**	**Linear Conjugate Flow**	**Stability**
1	false	false	false	X	–	Undefined
2	false	false	true	X	–	Unstable
3	false	true	false	X	–	Undefined
4	false	true	true	X	–	Unstable
–	true	false	false	X	–	–
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

2.4.1 Numerical Differentiation: Ridders Method

The derivative of $f(x)$ is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{2.33}$$

This formula suggests a naive approach to compute a numerical derivative: pick a small value h and apply (2.33). However, applied uncritically, this procedure is almost guaranteed to produce inaccurate results.

Ridders applied Romberg's method to improve the accuracy in the computation of the first and second derivatives of a real function [Ridders, 1982]. Using the Taylor expansion in the vicinity of x:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{2.34}$$

with a truncation error of $e_t \sim h^2 f^{(3)}$. As a consequence, e_t decreases quadratically with decreasing h. Repeatedly halving the value of h produces a series of corresponding values of $\frac{f(x+h) - f(x-h)}{2h}$, which is denoted 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 a better approximation is given by:

$$f'(x) \approx \frac{4A_2 - A_1}{4 - 1} \tag{2.35}$$

which is denoted by B_1.

This procedure leads us to the well known Romberg method:

	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 2.2: Table resulting from Ridders method

with

$$\begin{aligned} B_n &= \frac{A_{n+1} \cdot 4^m - A_n}{4^m - 1}, \quad m = 1 \\ C_n &= \frac{B_{n+1} \cdot 4^m - B_n}{4^m - 1}, \quad m = 2 \end{aligned} \tag{2.36}$$

and so on.

Algorithm 1 computes the terms in Table 2.2 and returns the last element of the diagonal as an approximation to the derivative at the requested point. It also returns an approximation of the error in the computed value. Line 1 assigns a size of 10 to the number of rows, m, in Table 2.2. Line 6 is the first approximation to the derivative, and line 7 assumes the error is big at this point (that is why `err` is assigned the maximum value that `Doubles` can represent). Line 15 corresponds to Equation 2.36. The error strategy is to compare each new extrapolation to one order lower, both at the present stepsize and the previous one. If error is decreased, the improved answer is saved (lines 18–21). If higher order error is worse by a significant factor SAFE, the algorithm quits early (lines 24–26).

2.4.2 Jacobians with Forward Differences

The implementation used in this work is based on [Dennis and Schnabel, 1987]. A forward difference approximation to $J(\mathbf{x}_s)$ (the jacobian matrix of $\mathbf{f}(\mathbf{x})$ evaluated at $\mathbf{x}_s$) is used.

Column j of $J(\mathbf{x}_s)$ is approximated by $\mathbf{f}(\mathbf{x}_s + h_j\mathbf{e}_j)$, where $\mathbf{e_j}$ is the j–th unit vector, and $h_j = \eta^{1/2} \max\{|\mathbf{x}_s[j]|, 1/\mathbf{S_x}[j]\} \operatorname{sgn}(\mathbf{x}_s[j])$. $1/\mathbf{S_x}[j]$ is the typical size of $|\mathbf{x}_s[j]|$ 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 $J(\mathbf{x}_s)$ and J typically will agree in about their first DIGITS/2 base 10 digits[Dennis and Schnabel, 1987].

2.5 Final Remarks

This chapter gives the necessary terminology from dynamical systems theory used in this work. Section 2.2.1 is an illustration of how the Linearization Theorem applies to

Algorithm 1: ridders(f, x, h)

Input : f, the function to be differentiated; x, the value at which the derivative will be computed; h, an initial step from x

Output: ans, the derivative of f at x; err, an estimate of the error in ans

1 ntab $\leftarrow 10$
2 safe $\leftarrow 2.0$
3 Initialize errt, fac, hh and ans to 0.0
4 Initialize bidimensional array a of size ntab $\times$ ntab to zeros
5 hh $\leftarrow$ h
6 a[*0,0*] $\leftarrow \dfrac{\mathsf{f}(\mathsf{x}+\mathsf{hh})-\mathsf{f}(\mathsf{x}-\mathsf{hh})}{2\mathsf{hh}}$
7 err $\leftarrow$ `Double.MaxValue`
8 con $\leftarrow 1.4$
9 con2 $\leftarrow$ con $\times$ con
10 **for** j = *1* **to** a.*numCols* **do**
11 hh $\leftarrow$ hh/con
12 a[*0, j*] $\leftarrow \dfrac{\mathsf{f}(\mathsf{x}+\mathsf{hh})-\mathsf{f}(\mathsf{x}-\mathsf{hh})}{2\mathsf{hh}}$
13 fac $\leftarrow$ con2
14 **for** i = *1* **to** j **do**
15 a[i, j] $\leftarrow \dfrac{\mathsf{a}[i\text{ - }1,\ j]\times\mathsf{fac}-\mathsf{a}[i\text{ - }1,\ j\text{ - }1]}{\mathsf{fac}-1.0}$
16 fac $\leftarrow$ con $\times$ fac
17 errt $\leftarrow \max\left\{\left|\mathsf{a}[i,\ j]-\mathsf{a}[i\text{ - }1,\ j]\right|,\left|\mathsf{a}[i,\ j]-\mathsf{a}[i\text{ - }1,\ j\text{ - }1]\right|\right\}$
18 **if** errt $\leq$ err **then**
19 err $\leftarrow$ *errt*
20 ans $\leftarrow$ a[i, j]
21 **end**
22 **end**
23 higherOrderWorse $\leftarrow |\mathsf{a}[j,\ j]-\mathsf{a}[j\text{ - }1,\ j\text{ - }1]| \geq$ safe $\times$ err
24 **if** higherOrderWorse **then**
25 break
26 **end**
27 **end**
28 **return** ans, err

Algorithm 2: jacobian(n, xs, f, Sx, eta)

Input : $n \in \mathbb{Z}$, $\mathbf{x}_s \in \mathbb{R}^n$, $\mathbf{f}(\mathbf{x})\colon \mathbb{R}^n \to \mathbb{R}^n$, $\mathbf{S}_\mathbf{x} \in \mathbb{R}^n$, $\eta \in \mathbb{R}$

Output: $\mathsf{J} \in \mathbb{R}^{n \times n} \approx J(\mathbf{x}_s)$

$\mathsf{sqrtEta} \leftarrow \eta^{1/2}$

for $j = 1$ **to** n **do**

`// calculate column` j `of` J

$\mathsf{stepSizej} \leftarrow \mathsf{sqrtEta} \times \max\{|\mathbf{x}_s[j]|, 1/\mathbf{S}_\mathbf{x}[j]\} \times \operatorname{sgn}(\mathbf{x}_s[j])$

$\mathsf{tempj} \leftarrow \mathbf{x}_s[j]$

$\mathbf{x}_s[j] \leftarrow \mathbf{x}_s[j] + \mathsf{stepSizej}$

$\mathsf{stepSizej} \leftarrow \mathbf{x}_s[j] - \mathsf{tempj}$

$\mathsf{fj} \leftarrow \mathbf{f}(\mathbf{x}_s + \mathsf{stepSizej} \times \mathbf{e}_j)$

for $i \leftarrow 1$ **to** n **do**

$\mathsf{J}[i, j] \leftarrow \dfrac{\mathsf{fj}[i] - \mathbf{f}(\mathbf{x}_s)[i]}{\mathsf{stepSizej}}$

end

$\mathbf{x}_s[j] \leftarrow \mathsf{tempj}$

end

a nonlinear system, and shows how a similarity transformation can be applied to $J(\mathbf{x}_s)$ in order to express the same matrix in a different basis (given by the eigenvalues in the case of non-defective matrices and by the generalized eigenvalues in the case of defective ones) such that the result is in a Jordan canonical form. The purpose of this process is twofold: it allows to classify the number of possibilities that can arise when computing the eigenvalues of $J(\mathbf{x}_s)$ (the similarity transformation preserves the eigenvalues of the original matrix) *and* it makes easier to solve the linear system of differential equations that arise from the linearization process. The solution in the original basis can be found by a simple linear mapping. Table 2.1 summarizes the different possibilities that can be studied from an eigenvalue analysis of $J(\mathbf{x}_s)$.

Chapter 3

Bifurcation Diagram Construction based on Niche PSO

In this chapter the basics of Particle Swarm Optimization (PSO) and Niche Particle Swarm Optimization (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 Particle Swarm Optimization

Particle swarm optimization (PSO) was originally designed and introduced by Eberhart and Kennedy [Eberhart and Kennedy, 1995]. The PSO is a population based search algorithm based on the simulation of the social behavior of birds, bees or a school of fishes.

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-th 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; $\mathbf{p}(t)$ is 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 sampled from a uniform distribution $U[0, 1]$.

The original form of PSO is the GBest version [Eberhart and Kennedy, 1995]. For the GBest version the neighborhood for each particle is the entire swarm. The social network employed by the GBest PSO uses the star topology (the social component of the particle velocity update reflects information obtained from all the particles in the swarm). In this case, the social information is the best position found by the swarm.

The global information exchange in the GBest version allows a fast convergence of a solution. However, the swarm loses diversity soon and this implies further exploration is not possible and the particles can perform only local search around their convergence point.

Eberhart and Kennedy designed a LBest version of PSO and introduced the concept of neighbourhood of a particle (the group of particles that can exchange social information with the given particle) [Eberhart and Kennedy, 1995]. Each particle assumes a set of other particles to be its neighbors and, at each iteration, it communicates its best position only to these particles, instead of to the whole swarm. Eberhart and Kennedy found that the LBest version has a better ability than the GBest version to avoid being trapped in local optima, at the cost of computation time (it converges slower).

3.2 Niche Particle Swarm Optimization

Niching algorithms are used for locating and maintaining multiple solutions to problems like multimodal function optimization and multiobjective function optimization [Mahfoud, 1995]. The PSO algorithm has poor habilities to achieve this goal, as summarized by [Brits et al., 2007]. The Niche PSO 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 to a solution, a subswarm (niche) 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 3. The following sections describe this algorithm in detail (using a particular configuration). The implementation uses the CILib library (https://github.com/cilib/cilib/).

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

3.2.2 Sub–Swarm Training

Each subswarm is trained using GCPSO (Guaranteed Convergence PSO) as suggested by [Engelbrecht, 2007], because "it has guaranteed convergence to a local minimum [Van Den Bergh, 2006], and because the GCPSO has been shown to perform well on extremely small swarms [Van Den Bergh, 2006]".

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

Algorithm 3: nichePSO(nx, ns)

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)

```
1  Create and initialize a n_x-dimensional main swarm, S
2  repeat
3      Train the main swarm, S, for one iteration using the cognition–only
       model
4      Update the fitness of each particle (S.x_i) of the main swarm
5      for each sub-swarm S_k do
6          Train sub-swarm particles, S_k.x_i, using a full model PSO
7          Update each particle's fitness
8          Update the sub-swarm radius S_k.R
9      end
10     If necessary, merge sub-swarms
11     Allow sub-swarms to absorb any particles from the main swarm that
       moved into the sub-swarm
12     If necessary, create new sub-swarms
13 until stopping condition is true
14 return S_k.ŷ for each sub-swarm S_k as a solution
```

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 as:

$$\rho(t) = \begin{cases} 2\rho(t) & \text{if number of sucesses}(t) > \epsilon_s \\ 0.5\rho(t) & \text{if number of failures}(t) > \epsilon_f \\ \rho(t) & \text{otherwise} \end{cases} \tag{3.5}$$

where number of successes and number of failures denote the number of *consecutive* successes and failures, respectively (ϵ_s and ϵ_f are the chosen thresholds for these). A failure occurs when the new global best position is actually equal or worse than the previous global best position. The optimal values for ϵ_s and ϵ_f are problem dependent, but [van den Bergh and Engelbrecht, 2002] recommends to set $\epsilon_s = 15$ and $\epsilon = 5$ for high-dimensional search spaces.

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

$$\mathbf{v}_{ij}(t+1) = \chi[\mathbf{v}_{ij}(t) + \phi_1(\mathbf{y}_{ij}(t) - \mathbf{x}_{ij}(t)) + \phi_2(\hat{\mathbf{y}}_j(t) - \mathbf{x}_{ij}(t))] \tag{3.6}$$

where

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

with $\phi = \phi_1 + \phi_2$, $\phi_1 = c_1 r_1$, and $\phi_2 = c_2 r_2$. Under the conditions that $\phi \geq 4$ and $\kappa \in [0, 1]$, the swarm is guaranteed to converge [Engelbrecht, 2007].

The `ConstrictionVelocityProvider` is used together with a `ClampingVelocityProvider`. The following code excerpt shows how the velocity providers for each sub-swarm (niche) have been configured:

```
// velocityProvider
val subSwarmVelProv = new GCVelocityProvider()
subSwarmVelProv.setRho(ConstantControlParameter.of(1.0))
subSwarmVelProv.setSuccessCountThreshold(15) // epsilons
subSwarmVelProv.setFailureCountThreshold(5)  // epsilonf

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, `LBestTopology` has been used for the subswarms.

3.2.3 Creation and Merging of Niches

The merge strategy for subswarms needs special care. If merging subswarms is not allowed to maintain every niche, then multiple subswarms might be refining the same solution. On the other hand, if subswarms merge too easily, some niches could be lost. A merge detection strategy based on diversity has been used in this work.

The basic ideas are:

- A sub-swarm is formed when a particle seems to have converged to 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 are added to a suitable niche.

Using CILib, this behaviour can be configured with the following code:

```
// 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 Bifurcations diagrams

As stated in Section 1.1.2, a bifurcation diagram depicts a scalar measure $[\mathbf{x}]$ versus the real parameter θ, where $(\mathbf{x}, \theta)$ solves (1.1).

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

1. Stationary solutions $\mathbf{x}(t) \equiv \mathbf{x}^s$, $\mathbf{f}(\mathbf{x}^s) = \mathbf{0}$ (e.g. nodes, saddle points, foci, and degenerate cases like centers) and turning points.

2. Periodic solutions $\mathbf{x}(t+T) = \mathbf{x}(t)$ (e.g. limit cycles, heteroclinic, and homoclinic orbits).

3. One-to-one solutions $\mathbf{x}(t_1) \neq \mathbf{x}(t_2)$ for $t_1 \neq t_2$.

Some stationary solutions can be at the same time *bifurcation points*. Informally, a bifurcation point (with respect to parameter θ) is a solution $(\mathbf{x}_{s_0}, \theta_{s_0})$ to Equation (1.1) where the number of solutions changes when θ passes θ_{s_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 $\lambda(\theta)$ of the Jacobian $J(\mathbf{x}_s)$ of stationary solutions $(\mathbf{x}_s, \theta)$ 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 Equation (1.1) is done with *continuation*. However, this problem can be formulated as an optimization problem (see Section 1.1.1). In general, several solutions could exist for each θ in a multi-parameter problem, and therefore an optimization method that can find multiple solutions to multimodal optimization problems is needed. Niche PSO is used in this work to find multiple fixed points of dynamical systems for every parameter value.

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 parameters of Niche PSO have 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.

Chapter 4

BDT: Bifurcation Diagram Tool

This chapter describes BDT, a Bifurcation Diagram Tool that uses Niche PSO to construct bifurcation diagrams. In previous chapters, attention has been given to theoretical foundations of qualification of fixed points of dynamical systems. In this chapter, a description of important implementation issues will be considered.

The process of plotting a bifurcation diagram with *BDT* (Bifurcation Diagram Tool) follows.

1. Parse a description of the dynamical system. To be successfully parsed, such description must be recognized by the Parser Expression Grammar (PEG) defined in Appendix A.3.

2. A function $\mathbf{f}(\mathbf{x})$ is built that may depend on at most two parameters θ_1, θ_2, using `DynamicalSystem`, and *closing over* some of the parameters (the other θs) to fix their values (this is done by the user), so that at most two parameters vary quasi-statically. In other words, if there are more than two parameters, the user has to fix the additional ones to specific values (or even all of them but one).

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

4. Since there is no absolute guarantee that NichePSO has converged to fixed points only

(it may have stagnated somewhere else), solutions with fitness above a given tolerance are discarded.

5. Classify the stability of every fixed point using the criteria described in Section 2.3.
6. If the fixed point is hyperbolic, the conjugate linear flow near it is classified.
7. Plot the bifurcation diagram.

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

- Parsing and evaluating dynamical systems.
- Minimizing $\mathbf{f}$ using Niche PSO.
- 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 Graphical User Interface (GUI).

The first three bullets have been addressed in previous chapters. In this chapter a description of the last two bullets in the previous list is given.

Section 4.1 emphasizes that the implementation of BDT is using the Java Virtual Machine (JVM), taking advantage of *both* Java and Scala. Section 4.2 just states some choices made during development of BDT, with respect 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. 4.1) the reader can have a glimpse of how testing has been used for the GUI, 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.

4.1 The JVM, Java, and Scala

In previous chapters the theoretical foundations of this work have been established, as well as important algorithms used in BDT. Implementation issues (that have needed careful thought, and a lot of time and effort) have received almost no attention, up to this chapter.

One of the requirements for BDT is to work on top of the Java Virtual Machine (JVM). At this point, a distinction should be made:

- *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 [Gosling et al., 2012].

- *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 [Lindholm et al., 2012].

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

A *type system* is a tool that helps people to reason about programs. It allows to

classify terms—syntactic phrases—according to the properties of the values that they will compute when executed [Pierce, 2002].

For example, in Java, the value `true` has type `boolean`, while the value `"Hello, world!"` has type `String`. Every boolean variable shares certain properties with other values of the same type. For example, boolean functions (like AND, OR, NOT) can be applied on boolean values and strings can be concatenated; these are properties of those types. That type information allows us to reason about those values without concerning low-level details, and some valuable *abstraction* is gained.

According to [Pierce, 2002], a type system is good at

- Detecting errors.
- Giving programs a higher level of abstraction.
- Documenting code.
- Providing language safety.
- Improving the efficiency of programs.

so there are compelling reasons to harness the power of type systems.

Functions are a noteworthy example of the convenience of the Scala's type system. Functions can be stored in a variable, being called and passed around to other functions as parameters. The type of a function `f` with input type `A` and output type `B` can be written in Scala as `f: A ⇒ B`. For example, the signature of the `ridders` method can be written as

```
1 def ridders(f: Double ⇒ Double, h: Double): Double ⇒ Double
```

The first parameter, `f` (the function that is going to be differentiated), is a function that takes a `Double` and returns a `Double`. The `ridders` method returns a *function* (the first derivative of `f`), which takes a `Double`—the x_0 needed to evaluate $f'(x_0)$—and returns a `Double`—the value of $f'(x_0)$.

Java and C#, two of the most important mainstream languages, have first-order parametric polymorphism, usually called *generics* in these languages. First-order parametric polymorphism has a standard application area in collections. For example, it is possible to abstract over types, to get *type constructors* such as `List[_]`. To get a concrete type like `List[Int]`, the type constructor needs the argument type `Int`.

First-order parametric polymorphism has some limitations though: type constructors cannot be abstracted over. For example, it is not possible to pass a type constructor as a type argument to another type constructor. This generalisation to types that abstract over types that abstract over types ("higher-kinded types") has many practical applications, as reported by [Moors et al., 2008].

A functor is an example of a data type that the Java's type system cannot encode (it lacks the ability to encode higher-kinded types). A functor captures the idea of a data type that implements the `map` operation. In Scala, a functor can be encoded with:

```
1 trait Functor[F[_]] {
2   def map[A,B](fa: F[A])(f: A ⇒ B): F[B]
3 }
```

Lists and a lot of other type constructors (e.g. `Option[_]`) are instances of the data type Functor.

```
1 val listFunctor extends Functor[List] {
2   def map[A,B](as: List[A])(f: A ⇒ B): List[B] = as map f
3 }
```

It is necessary to emphasize that this work takes advantage of several Java libraries. Those can be used from Scala without problems.

4.2 Plotting in 2D and 3D

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

Another compatibility issue that had to be addressed 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[1] for 2D plotting.
- Jzy3d[2] for 3D plotting.

The backend used for 2D plotting is JFreeChart, but a convenient wrapper for this library, scala-chart, is exploited (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 Graphical User Interface (GUI) will illustrate some of the techniques used in BDT. 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. 4.1, that corresponds to an instance of class `ConfParamFrame`.

[1] License: GNU Lesser General Public Licence (LGPL).
[2] License: New BSD

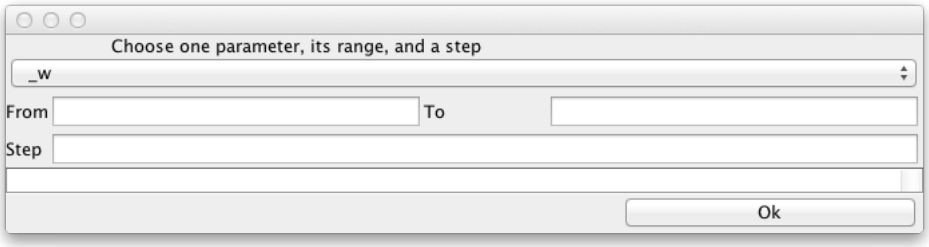

Figure 4.1: Configuration of parameters.

GUI code is a substantial part of most applications (roughly 45–60 % of an average codebase [Memon, 2002]), nonetheless, GUI tests remain relatively unused in practice and remain an unexplored area of research [Memon, 2002].

The needs for GUI tests are unique: tests should be able to click, type and perform a multitude of other actions. FEST-Swing has been chosen to test the GUI-functionality. FEST-Swing 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, according to 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 GUIs. Test `ConfParamFrameTest1` is written using ScalaTest (as every test implemented in this work so far). The most important points of Listing 4.1 are:

- Line 1 shows `ConfParamFrameTest1` extends `FestiveFunSuite`. The latter was implemented to manage the following tasks:
  - Create `var window: FrameFixture` that will handle the tested window in a EDT-

safe way.

- Install `FailOnThreadViolationRepaintManager` to catch EDT-access violations. This is done before every test in the class.
- 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 first test, lines 13–20, verifies that clicking the OK button (see Fig. 4.1) 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 4.1: Excerpts of GUI Tests implemented for frame in Fig. 4.1

```
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 }
```

The build file for the project (written for `sbt`, the Simple Build Tool) 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) [Hinojosa, 2012].

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 the `GUITest` annotation is

```
1 package umich.gui.tags;
2
3 import java.lang.annotation.*;
4 import org.scalatest.TagAnnotation;
5
6 @TagAnnotation
7 @Retention(RetentionPolicy.RUNTIME)
8 @Target({ElementType.METHOD, ElementType.TYPE})
9 public @interface GUITest {}
```

To run only the tests tagged `GUITest`, use the `gui:test` task from the sbt prompt.

Listing 4.2: Special configuration for tests tagged `GUITest`

```
lazy val BDT = Project(
  "BDT",
  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 us discuss a more abstract part of the same example (see Fig. 4.1): validation of input (numeric fields From, To, and Step) using *Applicative Functors*. See Listing 4.3 and the corresponding comments afterwards.

Listing 4.3: Validation of Parameter Configuration using Applicative Functors

```
1  class Range private(val from: Double, val to: Double)
2  object Range {
3    def apply(from: Double, to: Double, errorMsg: String):
4        Validation[String, Range] = if (from <= to) new Range(from, to).success
5                                    else errorMsg.fail
6  }
7
8  class ParameterConfig private(val from: Double, val to: Double, val step: Double)
9  object ParameterConfig {
10   def apply(from: Double, to: Double, step: Double): ValidationNel[String, ParameterConfig] = {
11     val vnelRange = Range(from, to, cond1Msg).toValidationNel[String, Range]
12     val vnelStep = Range(step, (to - from).abs, cond2Msg).toValidationNel[String, Range]
13     (vnelRange ⊛ vnelStep) { (r, s) ⇒ new ParameterConfig(r.from, r.to, s.from) }
14   }
15   val cond1Msg = "From must be less or equal than To"
16   val cond2Msg = "Step must be less or equal than given range"
17 }
```

- Class `Range` and its companion object are auxiliary to class `ParameterConfig` and its companion object.
- Lines 1 and 8 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 10).
  - `Range.apply` has return type `Validation[String, Range]`. A `Validation[E, A]` represents either a `Success(a)` or a `Failure(e)` and its motivation is to provide

an instance of `Applicative[λ[_]]` (where `λ[α] = Validation[E, α]`) that accumulates errors through semigroup[3] `E`.

– Line 11 calls `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 the type `Validation[NonEmptyList[String], Range]`). Something analog is done in lines 12–13.

Line 13 is the reason of Listing 4.3. The function

```
{ (r, s) ⇒ new ParameterConfig(r.from, r.to,  s.from) }
```

is applied to `(vnelRange ⊛ vnelStep)`. The aforementioned function knows nothing about the possibility of failure: `Validation` will handle this automatically. In case of success, a tuple `(r, s)` representing a range and a step will be available, and a `ParameterConfig` can be built from them. What happens in case of failure? In this example that could come from `vnelRange` or `vnelStep` (both are applicative values, instances of `Applicative[λ[_]]` where `λ[Range] = Validation[NonEmptyList[String],Range]`).

Suppose `vnelRange = Failure(NonEmptyList(cond1Msg))`. If `vnelStep` is a `Success`, line 13 would return a `Failure(NonEmptyList(cond1Msg))`. If `vnelStep` is also a `Failure` then line 13 would return the accumulated failures: `Failure(NonEmptyList(cond1Msg, cond2Msg))`. If both `vnelRange` and `vnelStep` are `Success`es, line 13 returns a `ParameterConfig` wrapped in a `Success`. Applicative Functors have provided a very succint and powerful way to manage every possibility.

Additional validation is used in the code corresponding to Fig. 4.1 to make the application more robust: only suitable numerical input that corresponds to an actual `ParameterConfig` is accepted. In case of failures, the accumulated error messages are used to give the user some feedback to correct his/her input.

[3] 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*.

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. The following comments explain Listing 4.4.

- If validation of input fails, then a `NonEmptyList(String)` is returned by `getErrorsOrParamConfig()`, 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. This is shown in lines 4–10.

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

Listing 4.4: Clicking OK (see Fig. 4.1) calls this code

```
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(
16             c ⇒ c(()), println("This should happen during testing only"))
17         })
18   }
19 }
```

Fig. 4.1 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 control of the application has been recovered, code can be written in only one place, instead of having logic for managing GUI events spread in several places. The purpose of using this programming style is to write code that is more

maintainable and easier to change.

As an example, let us 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 (validation is used for this) so that the next dialog can be created and shown *and* 2) undo inversion of control.

- Note lines 4–9 show the program logic (with a convenient imperative programming style) for GUI in *one place* once control of the application has been recovered.

- 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. 4.1).

- 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 a `shift` is being used, `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).

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

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

4.4 Final Remarks

This chapter emphasizes implementation details of BDT, 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.

Chapter 5

Results

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

Section 5.1 focuses on a simple problem that represents a challenge for Niche PSO: the canonical example of a subcritical pitchfork bifurcation.

Section 5.2 shows a bifurcation diagram obtained with BDT where two parameters are varied.

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

5.1 Subcritical Pitchfork Bifurcation Diagram

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

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

The scalar measure used for the bifurcation diagram is simply x. Figure 5.1 shows

the corresponding bifurcation diagram obtained with BDT. Blue dots are stable fixed points, red dots are unstable fixed points and black dots indicate fixed points where the stability criteria—the sign of the derivative of the right hand side of Eq. (5.1)—is not enough to classify them (nothing can be said in general when $f'(x_s) = 0$ for the fixed point x_s, and a graphical analysis is required [Strogatz, 1994]).

From the bifurcation diagram, $(\theta, x) = (0, 0)$ is a subcritical pitchfork bifurcation (and the black dot gives us a hint that something special might be happening there). The other two dots at $\theta = -0.25$ correspond to turning points (or saddle-node bifurcations).

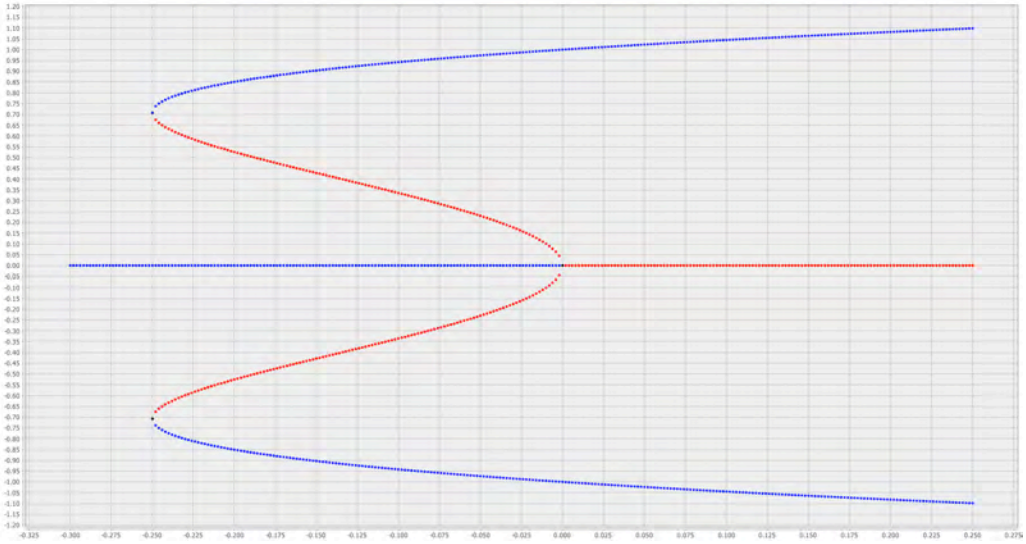

Figure 5.1: Bifurcation diagram for Equation (5.1) obtained with BDT.

To compute the fixed points for Figure 5.1, a user of BDT has to enter the following description (the user selects the parameter range and the step using a graphical user interface):

Listing 5.1: Input for BDT for dynamical system (5.1).

```
x' = _theta*x + x^3 - x^5
measure1 = x
```

Obtaining the corresponding bifurcation diagram with PyDSTool requires the user to give detailed instructions, as shown in Listing 5.2. See Fig. 5.2. The user should know

some Python to describe the dynamical system (AUTO demands some Fortran, C and Python knowledge). BDT is simpler to use, because the user can provide a description of the dynamical system using a *domain specific language* created to avoid the specifics of some programming language (see Appendix A). This lowers the entry point to start using BDT.

Listing 5.2: Python Script for PyDSTool to get Fig. 5.2

```
1  import PyDSTool as dst
2  import matplotlib.pyplot as plt
3
4  DSargs = dst.args(name='Canonical example of a subcritical pitchfork bifurcation')
5  DSargs.pars = {'r': -0.25}
6  DSargs.varspecs = {'x': 'r*x + x**3 - x**5', 'w': 'x - w'}
7  DSargs.ics = {'x': 0.5, 'w': 0}
8  ode = dst.Generator.Vode_ODEsystem(DSargs)
9  PC = dst.ContClass(ode)
10 PCargs = dst.args(name='SubcritPitchforkBif', type='EP-C')
11 PCargs.freepars = ['r']
12 PCargs.MaxNumPoints = 40
13 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()
```

The following comments highlight some important details of Listing 5.2 and make some comparison between the corresponding bifurcation diagrams and the methodology to produce them.

- Line 6 describes the system given by Eq. (5.1). Note that a *dummy* variable `w` is

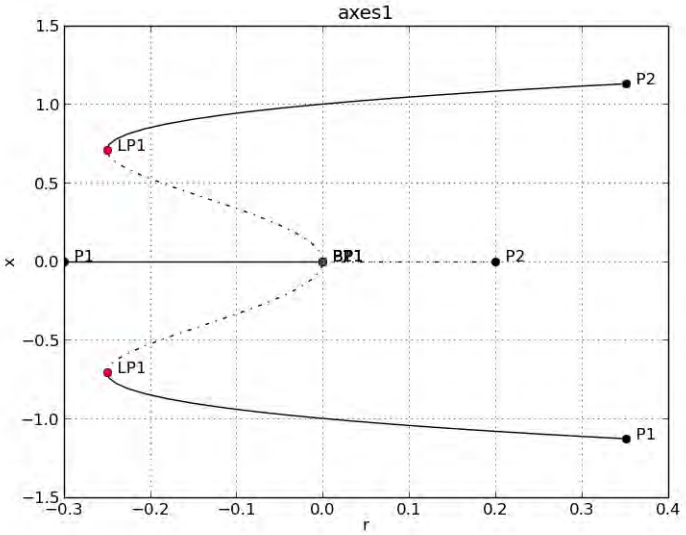

Figure 5.2: Bifurcation diagram for Equation (5.1) obtained with PyDSTool.

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. 5.2, PyDSTool needs to be provided with 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. 5.2). In contrast, Fig. 5.1 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.

- Notice how the bifurcation diagram is drawn in pieces. The user has to give instructions to produce three curves: `PC`, `PC2`, and `PC3`. What if the user has no idea about the number of pieces he/she must draw?

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.

5.2 Insect Outbreak

This section deals with a dimensionless formulation of the insect outbreak model studied in [Strogatz, 1994]. The dynamical system is given by the scalar equation

$$\dot{x} = rx\left(1 - \frac{x}{k}\right) - \frac{x^2}{1 + x^2} \tag{5.2}$$

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

The bifurcation diagram shown in Fig. 5.3 was obtained with BDT, where axis x, y, z represent k, r, and x, respectively. From the diagram, it should be obvious that there are two saddle node bifurcation for each value of k.

Listing 5.3: Input for BDT for dynamical system (5.2).

```
1 x' = _r*x*(1 - x/_k) - x^2/(1 + x^2)
2 measure1 = x
```

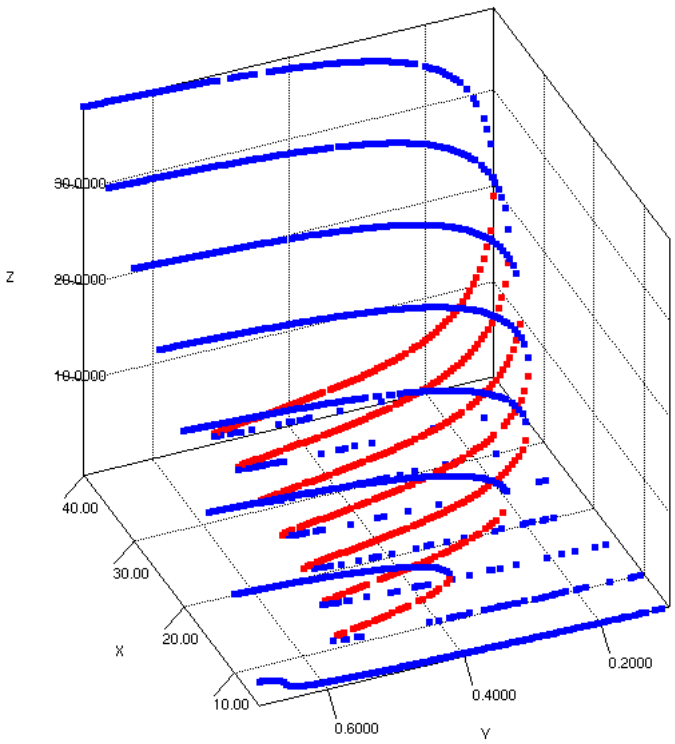

Figure 5.3: Bifurcation diagram for Equation (5.2) obtained with BDT.

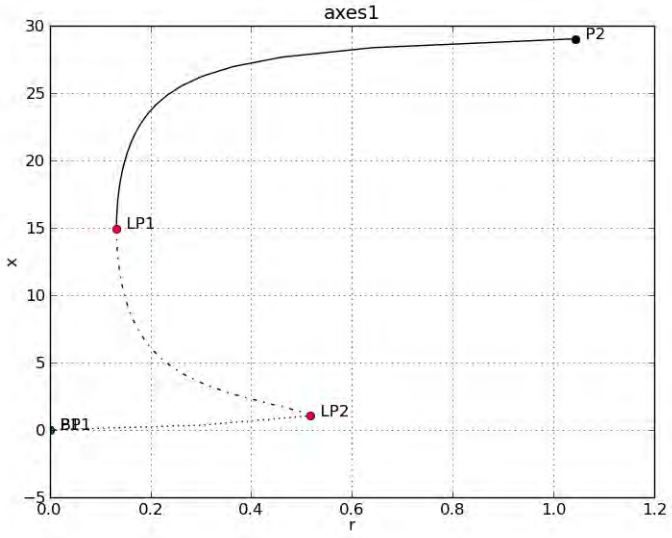

Figure 5.4: Bifurcation diagram for Equation (5.2) obtained with PyDSTool.

A corresponding *part* of the latter diagram was obtained with PyDSTool. See Listing 5.4 and Figure 5.4. Note that only one parameter is varied and only one initial condition is given (see line 7). Writing the Python code (including appropriate initial conditions) to obtain all the slices of Fig. 5.3 is certainly a hard task. To produce a complete bifurcation diagram like the one produced by BDT and shown in Fig. 5.3, this process would need to be repeated for each slice of the bifurcation diagram, then the results should be assembled. This process is performed automatically by BDT.

Listing 5.4: Python script for PyDSTool to get Fig. 5.4

```
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
15 PCargs.StepSize = 0.1
16 PCargs.LocBifPoints = ['LP', 'BP']
17 PCargs.StopAtPoints = 'BP'
18 PCargs.SaveEigen = True
19
20 plt.clf()
21 plt.grid("on")
22 PC.newCurve(PCargs)
23 PC['InsectOutbreak'].forward()
24 PC['InsectOutbreak'].backward()
25 PC.display(['r', 'x'], stability=True)
26
27 plt.show()
```

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 BDT. Whereas PyDSTool requires suitable initial conditions provided by the user, computational intelligence algorithms (such as Niche PSO) provide BDT with the ability to get similar output with less guidance (studying only local bifurcation problems).

Scripts for PyDSTool have been used to illustrate a general characteristic of continuation-based bifurcation tools: the user must provide a lot of guidance to get complete bifurcation diagrams. This guidance may be provided in a more interactive way, using a graphical user interface (e.g. XPPAUT), but *still* has to be provided.

In the other hand, using computational intelligence algorithms (like Niche PSO and others) to solve *different* bifurcation problems, may require some tuning of the parameters of the heuristic involved. For example, in the case of Niche PSO, the number of particles of the main swarm, the number of iterations that the algorithm is going to be executed, etc., may require attention from the researcher. A good understanding of the heuristic involved is expected to get appropriate results. In a typical situation, several experiments may be required before getting acceptable diagrams.

Continuation-based tools require smaller computational time to produce bifurcation diagrams. However, the researcher has to invest a lot of time and intellectual resources to provide the required guidance. 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. Further experimentation and analysis is required though (e.g. different and harder problems, repeated enough times to get data for deeper conclusions). Using different heuristics would provide a better perspective also. Design of good experiments is essential.

AUTO and related tools have a big number of features and can solve a wide variety of problems. Some of them have been under development for years (AUTO has been around for decades) by several people. No doubt researchers will continue to take advantage of them. A serious contender should provide clear advantages and should be robust and easy to use. BDT is still a proof of concept, and requires talented people to use, design, develop, maintain, document, and test it.

Chapter 6

Conclusions

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

Section 6.1 argues that the usage of Niche PSO as a 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. Differential equations remain to be one of the best ways to model our world.

In this work Niche PSO has been applied as an alternative way to find fixed points of dynamical systems. Further experimentation and analysis is required, though.

A proper identification of the most important parameters of the algorithm has to be made. There are a lot of parameters that can be studied: the number of particles; the stopping condition of the algorithm; a proper selection of the creation, merging, and absortion criteria for the subswarms (niches); the selection of the scaling factor ρ and κ for controlling the exploration/exploitation behaviour of subswarms in a intelligent way (according to the problem characteristics); a study of the social structure imposed on subswarms (selection of topologies); just to mention some of them. How each of these paramaters affect the performance of Niche PSO for bifurcation diagram plotting? Which are the most important? A proper answer to these questions deserves thorough investigation and experimentation on its own.

In this work several areas had to be addressed: the conceptual and mathematical background of dynamical systems (what can be learned about fixed points using heuristics instead of continuation? how can stability of fixed points be determined?); the compiler/interpreter theory and implementation to recognize a language targeting systems of ordinary differential equations; the computational intelligence involved in the Niche PSO algorithm, as well as knowing how to take advantage of a complicated (but complete) implementation in the CIlib library; installing, knowing and using traditional bifurcation tools to be able to compare with this work; the implementation details involved in an application that requires graphics, portability, ease of use with a graphical user interface, numerics, artificial intelligence, etc. The task is daunting in extension and complexity. Each one of these areas was studied only in a rather introductory way.

A limited area of bifurcation analysis has been addressed. As pointed out in Section 3.3, only local bifurcations have been studied. A deeper understanding of both dynamical systems and computational algorithms is required to investigate the intersection between global and periodic phenomena with heuristics.

6.2 Future Work

The Java language lacks advanced modularization support, as witnessed by the existance of the Jigsaw project (http://openjdk.java.net/projects/jigsaw/). Some limitations are [Hall et al., 2011]:

- 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 (http://www.osgi.org/Main/HomePage). A good architecture for BDT could use OSGi bundles as units of modularisation. The OSGi framework provides functionality in the following layers [Alliance, 2012]:

- 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 (at the same time that Java modularity limitations are addressed).

For the current implementation, Niche PSO is restarted every time a parameter is modified. This approach has advantages and disadvantages. Previous results (possibly innacurate and/or incomplete) are not trusted blindly (to guide the search), so each time the

algorithm is asked to search for *every* solution. For example, sometimes the algorithm does not 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 this fact can be exploited 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 ability to find multiple solutions could be to interleave normal niching iterations (starting without any guidance) with dynamic PSO searches.

Currently, parameters 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 large.

A more traditional approach to bifurcation and stability analysis could be implemented on the JVM to make comparisons with AUTO and related tools.

Appendix A

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 [Grune, 2012].

Following the approach proposed by [Labun, 2012], this works uses a Parsing Expression Grammar (PEG) instead of Context-Free Frammars for the definition of a language for system of differential equations, and combinator parsers instead of parser generators.

A.1 Grammars

Most language syntax theory and practice is based on generative systems, particularly context-free grammars (CFGs) and regular expressions (REs) [Ford, 2004]. Chomsky's generative system of grammars allows for ambiguities, which is useful for modelling natural languages, but this power makes it 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" [Ford, 2004] using *prioritized choice*. PEGs provide operators for constructing grammars. The combinator parsing systems from the Scala standard library, use PEG semantics for recognizing languages. Table A.1 shows how PEG operators are implemented in Scala combinator parsing library.

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 A.1: Implementation of PEG operators in Scala

A.2 Combinator 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 this project is the hability to change the grammar more easily.

A.3 A Grammar for Recognizing a Dynamical System

A PEG grammar for recognizing first-order systems of ordinary differential equations can be defined as follows.

```
system ← sentence+ scalarMeasure scalarMeasure?

sentence ← equation / constantDefinition

equation ← dotStateVar "=" expr

constantDefinition ← constant "=" floatingPointNumber

scalarMeasure ← ("measure1" / "measure2") "=" 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"/"Ln"/"Log"/"Exp"/"Tan"/"Cot"/"Sec"/"Csc"/"Sqrt"

simpleExpr ← stateVar / constant / parameter / floatingPointNumber / "(" expr ")"
```

For brevity, the definition of a `floatingPointNumber` is taken for granted (definition in `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 arithmetic 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 Abstract Syntax Tree (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 expressions written in the specific intermediate representation language. In practical terms, an *interpreter* for simple dynamical systems has been built.

A.4 AST definition and evaluation

Section A.3 stated that the only parser's task is to build an AST. Some excerpts from the implementation code will be used 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 construct objects without the `new` keyword. Besides, the call `Equation(dotStateVar, expr)` is a shortcut to `Equation.apply(dotStateVar, expr)`.

- Line 4 uses the `^^^` 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 that is used for the exponentiation operation (which is right associative).

- By using `^^`, `^^^`, and case classes (like `Equation`, `StateVar`, `Add`, `Sub`, and others) the nodes of a parse tree (the AST) are being built. 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 (the usual recursive tree evaluation).

Bibliography

[Abelson, 1990] Abelson, H. (1990). The bifurcation interpreter: A step towards the automatic analysis of dynamical systems. *Computers & Mathematics with Applications*, 20(8):13–35.

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

[Barrera et al., 2008] Barrera, J., Flores, J. J., and Fuerte-Esquivel, C. (2008). Generating complete bifurcation diagrams using a dynamic environment particle swarm optimization algorithm. *Journal of Artificial Evolution and Applications*, 2008:1–8.

[Barrera Mendoza, 2012] Barrera Mendoza, J. A. (2012). *Análisis de sistemas dinámicos utilizando herramientas de inteligencia artificial*. Thesis, Universidad Michoacana de San Nicolás de Hidalgo, Morelia, Michoacán.

[Borrelli and Coleman, 2004] Borrelli, R. L. and Coleman, C. S. (2004). *Differential equations: a modeling perspective*. Wiley New York, NY.

[Brits et al., 2002] Brits, R., Engelbrecht, A. P., and Bergh, F. V. D. (2002). A niching particle swarm optimizer. In *In Proceedings of the Conference on Simulated Evolution And Learning*, page 692—696.

[Brits et al., 2007] Brits, R., Engelbrecht, A. P., and Van den Bergh, F. (2007). Locating multiple optima using particle swarm optimization. http://www.sciencedirect.com/science/journal/00963003.

[Clewley, 2012] Clewley, R. (2012). Hybrid models and biological model reduction with PyDSTool. *PLoS computational biology*, 8(8):e1002628.

[Dennis and Schnabel, 1987] Dennis, J. E. and Schnabel, R. B. (1987). *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Society for Industrial Mathematics.

[Doedel, 1981] Doedel, E. J. (1981). AUTO: a program for the automatic bifurcation analysis of autonomous systems. *Congr. Numer*, 30:265–284.

[Eberhart and Kennedy, 1995] Eberhart, R. and Kennedy, J. (1995). A new optimizer using particle swarm theory. In *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, page 39–43.

[Engelbrecht, 2007] Engelbrecht, A. P. (2007). *Computational intelligence: An introduction*. Wiley.

[Ermentrout, 2002] Ermentrout, B. (2002). *Simulating, analyzing, and animating dynamical systems: a guide to XPPAUT for researchers and students*, volume 14. Siam.

[Flores et al., 2011] Flores, J. J., Fuerte-Esquivel, C. R., Barrera, J., and Carvajal, H. R. (2011). Particle swarm optimization method to assess a voltage stability region by multiparameter bifurcation analysis. *International Review of Electrical Engineering*, 6(7).

[Ford, 2004] Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122.

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

[Grune, 2012] Grune, D. (2012). *Modern compiler design*. Springer, New York, NY.

[Hall et al., 2011] Hall, R., Pauls, K., McCulloch, S., and Savage, D. (2011). *OSGi in action: Creating modular applications in Java*. Manning Publications Co.

[Hinojosa, 2012] Hinojosa, D. (2012). *Testing in Scala*. O'Reilly Media, Inc.

[Labun, 2012] Labun, E. (2012). *Combinator Parsing in Scala.* Master's thesis, Institut für SoftwareArchitektur.

[Lindholm et al., 2012] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. (2012). *The Java® Virtual Machine Specification, Java SE 7 Edition.* Technical Report JSR-000924, Oracle.

[López Cuevas Villanueva, 2010] López Cuevas Villanueva, M. (2010). *Herramienta para el análisis de sistemas dinámicos mediante diagramas de bifurcación basado en metaheurísticas.* Master's thesis, Universidad Michoacana de San Nicolás de Hidalgo, Morelia, Michoacán.

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

[Memon, 2002] Memon, A. M. (2002). GUI testing: Pitfalls and process. *IEEE Computer*, 35(8):87–88.

[Moors et al., 2008] Moors, A., Piessens, F., and Odersky, M. (2008). Generics of a higher kind. In *Acm Sigplan Notices*, volume 43, page 423–438.

[Pierce, 2002] Pierce, B. C. (2002). *Types and programming languages.* The MIT Press.

[Ridders, 1982] Ridders, C. (1982). Accurate computation of f'(x) and f'(x) f"(x). *Advances in Engineering Software (1978)*, 4(2):75–76.

[Seydel, 2009] Seydel, R. U. (2009). *Practical Bifurcation and Stability Analysis.* Springer.

[Smale et al., 2003] Smale, S., Hirsch, M. W., and Devaney, R. L. (2003). *Differential Equations, Dynamical Systems, and an Introduction to Chaos, Second Edition.* Academic Press, 2 edition.

[Strogatz, 1994] Strogatz, S. H. (1994). *Nonlinear Dynamics And Chaos: With Applications To Physics, Biology, Chemistry And Engineering.* Westview Press.

[Van Den Bergh, 2006] Van Den Bergh, F. (2006). *An analysis of particle swarm optimizers.* PhD thesis, University of Pretoria.

[van den Bergh and Engelbrecht, 2002] van den Bergh, F. and Engelbrecht, A. P. (2002). A new locally convergent particle swarm optimiser. In *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, volume 3, page 6–pp.