ABSTRACT

Title of Dissertation: MATHEMATICAL VISUALIZATION, PARAMETER CONTINUATION, AND STEERED COMPUTATIONS

Randy Clinton Paffenroth, Doctor of Philosophy, 1999

Dissertation directed by: Professor J. Maddocks
Department of Mathematics

In this dissertation we study aspects of visualization, interactively steered computation, and efficient parallel implementations as they pertain to continuation algorithms for systems of nonlinear equations. Interactive steering of computations allow the user to be an active participant in the construction of the solution set, instead of being only a passive observer after the fact.

The first objective is to use visualization and special projections of part of the solution set of the nonlinear equations to facilitate efficient computation of more of the solution set, to organize and allow easy access to the large data sets that arise, and most of all, to generate mathematical understanding of the features and properties of the solution set. The second objective is that we want the design of the algorithms and implementations to be modular and extensible, allowing the easy incorporation of pre-existing software packages, especially numerical parameter continuation algorithms with efficient parallel implementations.

We also present visualization methods for the solution set, including various methods utilizing post-processing and projection, as well as a multi-parameter numerical continuation algorithm, closely based on one-parameter algorithms, called Single Parameter With Switching (SPWS). The SPWS algorithm has apparently not previously been discussed in the literature and may easily be parallelized, at

multiple levels, in a straightforward way. The SPWS algorithm does possess several deficiencies, but they are not critical when combined with interactive steering via visualization.

Two software packages are presented which implement the necessary infrastructure for such calculations, Multi-parameter Computation and Continuation Code (MCCC), and Visualization for Bifurcation Manifolds (VBM). In MCCC we utilize a novel decomposition of numerical parameter continuation algorithms to allow the easy construction and parallelization of parameter continuation algorithms. The MCCC framework is designed to allow the user to easily experiment with various continuation algorithms and solution correction procedures in a modular, object oriented, way. On the other hand, VBM is a visualization package with associated Application Programming Interfaces (APIs) that allows the interactive control and visualization of continuation algorithms.

MATHEMATICAL VISUALIZATION,
PARAMETER CONTINUATION,
AND STEERED COMPUTATIONS

by

Randy Clinton Paffenroth

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1999

Advisory Committee:

Professor J. Maddocks, Chairman/Advisor
Professor J. Alexander
Professor D. Baden
Professor R. Kellogg
Professor D. Mount

DEDICATION

For Christine

ACKNOWLEDGEMENTS

This thesis has benefited from the advice and support from more people than I could possibly list here. I am indebted to my advisor, Prof. John H. Maddocks, who guided my dissertation research. The freedom which he provided me throughout my dissertation research allowed me to explore the numerous areas of research which appear in this thesis, while his comments and ideas taught me the nuances of scientific research. I acknowledge Prof. Glenn Stevens, my undergraduate advisor and mentor, for first introducing me to mathematical research.

I am grateful to the past and present members of the Laboratory for Computation and Visualization in Mechanics (LCVM) for their help and support. Special thanks go to Dr. Robert Manning, for his work in developing VBM and our conversations on many topics; Angela Ahearn, for her work in GUI design as part of DataViewer; Viginia Kalb, for her development of the AvA (a VBM progenitor) front end; Charles Meztenyi, for his development of the AvA backend as well as long conversations on numerical methods and Fortran development; Dr. Gabor Domokos, for many of the ideas behind PCR (another VBM progenitor); Dana Vrajitoru, for her continuing development of DataViewer; and an extra special thanks to Dr. Thomas Stone, for the many hours of discussion on the fine points of object oriented programming, DataViewer, scene graphs, and many other topics.

I would also like to give a special thanks to David McNabb and Charles Goodrich, the manager and director of the Advanced Visualization Lab at the University of Maryland, for introducing me to scientific visualization, providing me employment at a time in my career when I desperately needed it, and initially introducing me to Dr. Maddocks.

I would like to thank my mother Mary Jane Paffenroth for her support and I cannot even begin to express my gratitude to Mary and Clinton House, my foster parents, who raised me during my formative years and were the best parents a person could ever have. My only regret is they are not alive today to see me finish what they encouraged me to start so many years ago. I would also like to thank my wife Christine for her unwavering support and love, without which this thesis would never have been possible. This dissertation is for them and because of them.

This research was supported by an AASERT Award from AFOSR.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In this dissertation we study aspects of visualization, interactively steered computation, and efficient parallel implementations as they pertain to continuation algorithms for systems of nonlinear equations of the form

$$F(\mathbf{x}) = \mathbf{0},\ \ F : \mathbb{R}^{n+m} \to \mathbb{R}^{n},\ \ n, m \in \mathbb{Z}, n, m > 0. \tag{1.1}$$

Such systems are known as *parameter dependent* or *under-determined*, so that solutions can be expected locally to form m-dimensional manifolds. We call any representation of their solution set a *bifurcation diagram*. In this dissertation we present a novel approach to such problems. We implement methods which not only find solutions to such systems, but do so interactively. Interactive methods allow the user to be an active participant in the construction of the solution set, instead of being only a passive observer after the fact.

Canonically, parameter dependent problems do not give rise to just a single solution, but rather to a continuous family of solutions. One class of methods for the numerical construction of such a family of solutions is *parameter continuation algorithms*. In such algorithms, known solutions to the equations are used to construct nearby, previously unknown, solutions. Parameter continuation algorithms are especially useful in that they are well suited to user interaction and computational steering, and they will therefore comprise our main computational tool.

On the other hand, even the best numerical algorithms do not suffice, in of themselves, to allow the user to interactively steer the computations. The data computed by the algorithm must be presented to the user, in some easily understandable form, before they are able to intelligently direct the computations. The process of retrieving the raw data from the numerical algorithm and presenting it to the user in a useful and concise form falls within the realm of scientific visualization. To realize our goal of interactively steered computations we will tightly couple visualization tools to our computational algorithms.

Figure 1.1 shows an example of the graphical user interface utilized within the

VBM program, which was developed as part of this dissertation research. This interface is used to interactively construct and visualize bifurcation diagrams for problems including equations of the form of equation (1.1). An example of the type of visualization that can be performed within VBM is shown in Figure 1.2. In our work we have studied discretizations of ordinary differential equations in which $n \approx 1000$, but in the bifurcation diagram shown in Figure 1.2 we are restricted to displaying only a hand-full of these variables at each point. In fact, in the bifurcation diagram in Figure 1.2, we show only four: three as spatial coordinates, and a fourth as color. As shown in the right hand panes of Figure 1.2, VBM addresses this issue by allowing the user to select individual points of the bifurcation diagram, here marked by red and white balls, and perform augmented visualizations of the selected points. Finally, as shown in Figure 1.3, the bifurcation diagrams in VBM are not static entities. They may be easily and interactively modified by the addition of data computed by parameter continuation algorithms driven through a compute engine interface and by the choice of different projections and views using the interactive graphics.

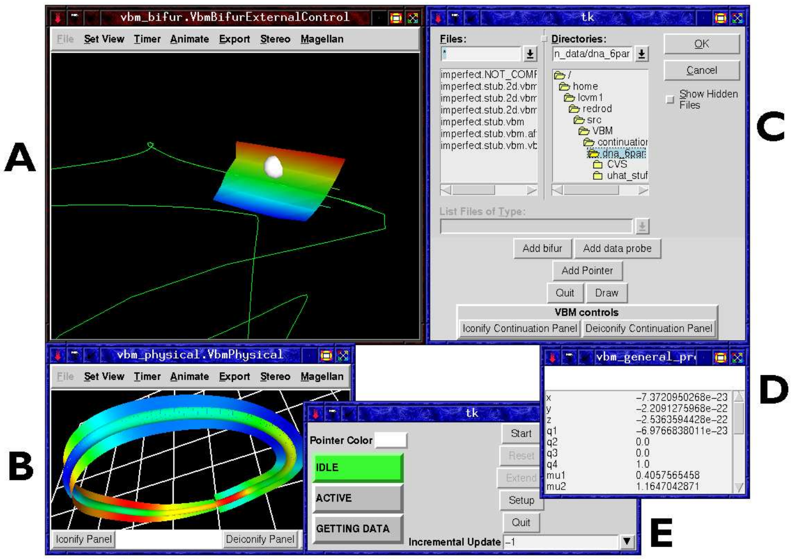

Figure 1.1: This figure shows some of the elements which make up VBM. The window marked A is a bifurcation visualization module written in DataViewer (see Appendix A for details). The window marked B is a data probe showing the solution corresponding to the point in the bifurcation visualization window marked with the white ball. The window marked C is the main VBM control panel. The window marked D is a data probe displaying numerical data and the window marked E is an interface to an compute engine, both of which are also attached to the white marker.

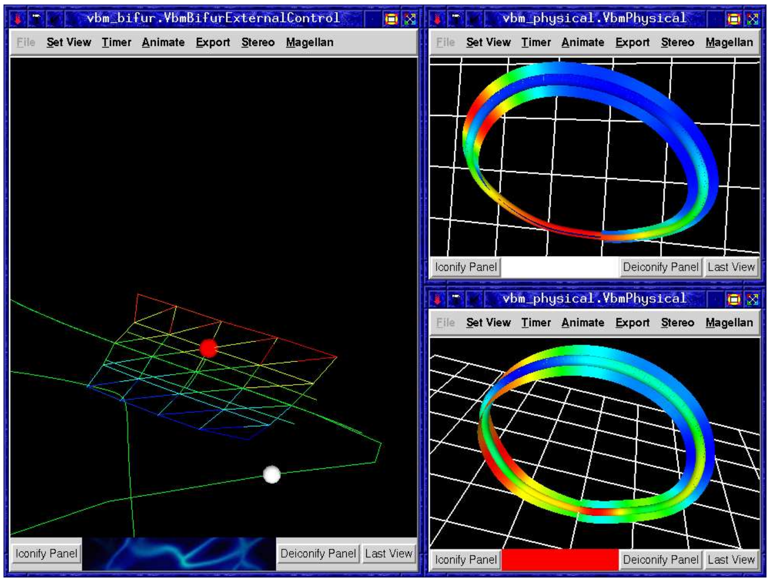

Figure 1.2: This figure shows a bifurcation diagram on the left, with a white sphere, and a red sphere each marking a single solution. Each vertex of the bifurcation diagram corresponds to a solution of equation (1.1) (the two dimensional manifold has been rendered using line segments to emphasize the position of the vertices). The windows on the right show the full solutions that correspond to the points marked with the balls. Each window displaying a full solution has a colored bar on the bottom which matches the color of the corresponding sphere.

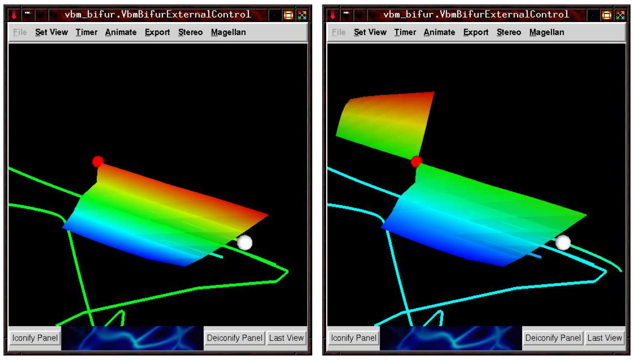

Figure 1.3: This figure shows two bifurcation diagrams. The bifurcation diagram on the left has two points marked with red and white spheres. The bifurcation diagram on the left shows the same bifurcation diagram, except that a new one dimensional portion of the bifurcation diagram has been computed starting at the white sphere, and a new two dimensional portion has been computed starting at the red sphere.

The core of the thesis is formed by Section 3.3, and Chapters 5, 6, and 7. The starting point of the analysis will be a description of the properties of the solution set of a special case of equation (1.1), namely

$$F(\mathbf{x}) = \mathbf{0}, \;\; F : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^n, \;\; n \in \mathbb{Z}, n > 0. \tag{1.2}$$

In other words, there is a single active parameter. Such systems have been widely studied in the literature, and the first part of Chapter 2 presents an overview of associated theory. Under mild assumptions, the solution set of the above equations comprises a set of smooth curves $\mathbf{c} : \mathbb{R} \rightarrow \mathbb{R}^{n+1}$ together with various types of singular points. We call such a collection of curves and singular points a bifurcation manifold. There is freedom in precisely how one defines $\mathbf{c}$. The most important decision one must make in the definition of $\mathbf{c}$ is its *parameterization.* Chapter 2 will present several possible parameterizations of $\mathbf{c}$ including the *arclength parameterization* and the *pseudo-arclength parameterization* [14, 15]. Arclength parameterization is important from a theoretical standpoint, e.g. for the derivation of uniqueness. Pseudo-arclength parameterization is a modification of arclength parameterization that is useful as a computational tool. Parameter continuation algorithms for the computation of such curves and singular points are well known and fall into two main categories, *predictor corrector methods* and *simplicial methods* [4, 2]. In brief, predictor corrector methods compute an approximation of $\mathbf{c}$ directly, while simplicial methods define a new problem, which approximates $\mathbf{c}$, and then solves this new problem exactly. The second part of Chapter 2 will present an overview of such methods and compare the two approaches.

Once the case $m = 1$ has been discussed, Chapter 3 will broaden our attention to $m > 1$. Our main goal is the construction of algorithms which may be interactively steered, and to that end, in Section 3.3, we present an algorithm for the case $m > 1$, whose implementation is strongly based on the methods used for $m = 1$. We call this algorithm the Single Parameter With Switching (SPWS) algorithm. The SPWS algorithm first decomposes a problem with $m > 1$ into a set of different problems with $m = 1$. It then solves this new set of problems, in a prescribed order, to approximate the solution of the original problem with $m > 1$. The SPWS algorithm is a simple and obvious generalization of algorithms for $m = 1$ which, to our knowledge, has not been pursued in the literature. The previous lack of interest in the SPWS algorithm is perhaps due to the fact that it is unsuitable for many purposes. However, we remark that, when combined with interactive control, the SPWS algorithm is in fact quite practical. Two other methods for the case $m > 1$ taken from the literature, one due to Rheinboldt [43] and one due to Henderson [22], will also be presented and their strengths and weaknesses compared to the SPWS algorithm.

To achieve our goal of interactively steered computations, a purely theoretical discussion of continuation algorithms does not suffice. We also need efficient computer implementations. Modern computers are exceedingly complex, with many particularities which can profoundly affect the performance of both parallel and serial numerical algorithms. The first part of Chapter 4 will review *memory hierarchy* and *pipelines*, and show how standard software packages may be used to achieve a high level of performance, particularly with regards to the goal of developing methods and an implementation that can be used for many numerical methods and for many different problems. Beyond the low-level numerical routines, the choice of programming model can also have a significant effect on performance and ease of use. Accordingly, we have chosen to develop our software packages using an *object oriented* approach in as many parts as possible. Chapter 4 will present a summary of basic object oriented nomenclature as well descriptions of selected advanced topics which will become important in later chapters, such as *dynamic method lookup* and *runtime typing.*

The object oriented and parallel computation ideas presented in Chapters 2, 3, and 4 have been combined, in the specific context of parameter continuation algorithms, to create a new software package developed as a part of this dissertation called the Multi-parameter Computation and Continuation Code (MCCC). MCCC uses a novel, object oriented, decomposition of parameter continuation methods, both for the case $m = 1$ and $m > 1$. This decomposition allows the easy development of efficient algorithms, and encourages code reuse and encapsulation. We also note that parameter continuation algorithms, especially those for which $m > 1$, have three distinct levels of parallelism, each of which has different communication characteristics. Our object oriented decomposition allows predictor corrector algorithms to be implemented to use these multiple levels of parallism in a straightforward, efficient, way. Chapter 5 describes the design of MCCC and specific details of its implementation.

Just as the computational algorithms are important, so are the visualization tools that one utilizes to understand the bifurcation manifolds that are generated by the numerics. There is of course a vast literature concerned with scientific visualization, some of which is directly applicable to the data sets generated by parameter continuation algorithms. On the other hand, some aspects of data sets arising in continuation algorithms must be attacked using more specialized methods. For example, much is known about the visualization of curves and surfaces in three dimensional space, especially when the curves and surfaces represent something which is physically realizable (e.g. a wing of an airplane). Unfortunately, our main concern is problems where $n + m \approx 1000$ and the curves and surfaces represent some mathematical entity, not something that necessarily has a physical analog. Canonically, we must first apply some specialized method to reduce the size of the problem into the regime where standard methods may be used, and in Section 6.2 we discuss novel algorithms for accomplishing this reduction. Chapter 6 will review standard visualization methods and introduce new methods which are applicable to the high dimensional case.

The ideas presented in Chapters 2, 3, 4, 5, and 6 have been implemented in a new software package called Visualization for Bifurcation Manifolds (VBM). VBM is an extensible, coherent environment which allows the user to interactively control a range of continuation algorithms as well as to utilize both general and problem-specific visualization tools. VBM accomplishes these feats by defining three application programming interfaces (API). The first interface, called the Bifurcation Visualization API, allows the definition of visualization modules which exhibit representations of bifurcation manifolds. The second interface, called the Data Probe API, allows the definition of visualization modules which exhibit detailed representations of individual solutions in the bifurcation manifold. Finally, the Compute Engine API defines a single interface through which a variety of parameter continuation algorithms may be used. The flexibility included in the definition of the various APIs is of special importance because we wish to utilize standard packages, such as the widely used parameter continuation package AUTO [16] within the VBM framework, as well as software modules which we have developed. The APIs inherent in VBM allow many software packages developed by others to be used within our interactive computation and visualization packages, in a way which is transparent to the user.

Chapter 8 will survey several examples and case studies of problems in which VBM has been applied. The first example will be a problem in which $n = 6$ and $m = 2$, whose solutions represent equilibrium configurations of a spinning heavy rigid-body [35]. The other examples leave the realm of explicit algebraic systems and enter that of continuous systems to which some discretization process has been applied. The first such example will be a boundary value problem (BVP) in ordinary differential equations (ODE) which may be interpreted as modeling optimal paths followed by three wheeled robots [1]. In this case $n \approx 500$ and $m = 1$. The next example will also show solutions to a BVP in ODEs, but in this case the problem models the statics of an elastic rod formed into a circle. This set of equations has been used extensively within the Laboratory for Computation and Visualization in Mechanics and Mathematics as a model for DNA mini-circles [36]. For this problem $n \approx 1000$ and $m = 1$. The final example will be a BVP in ODEs which models an elastic strut. In this case $n \approx 1000$ and $m = 2$.

A detailed summary of results is presented in Chapter 9.

Chapter 2

Overview Of Single Parameter Continuation Methods

2.1 Introduction

In this chapter we begin by considering equations of the form

$$F(\mathbf{x}) = \mathbf{0},\ \ F : \mathbb{R}^{n+1} \to \mathbb{R}^n,\ \ n \in \mathbb{Z}, n > 0. \tag{2.1}$$

Specifically, we are interested in knowing under what conditions $F^{-1}(\mathbf{0}) = \{\mathbf{x} \in \mathbb{R}^{n+1} | F(\mathbf{x}) = \mathbf{0}\}$ is non-empty and how one can approximate it numerically. In the first section we will describe some features of the solution set and conditions under which it can be shown to exist. The next section will survey some of the standard continuation methods for one parameter systems. The presentation of material here will be very cursory and merely serves to motivate what follows. For more detail one may look to the large body of literature regarding such problems, e.g. [3, 14, 15]. Later chapters will be concerned with the case where $n \gg 1$, and we will emphasize those places where scale has significant affects on the exposition.

2.2 Solution set

Given a problem of the form of equation (2.1) we want to characterize its solution set. The first step is to define a *regular zero point* of equation (2.1). Note that, for compactness of exposition, we will assume throughout that F is as smooth as necessary.

Definition 2.2.1. *A solution* $\mathbf{x}_0$ *of* $F(\mathbf{x}_0) = \mathbf{0}$ *is called a regular zero point of* $F(\mathbf{x})$ *if the Jacobian* $F'(\mathbf{x}_0)$ *has full rank.*

At a regular zero point of equation (2.1) we may apply the Inverse Function Theorem to yield:

Theorem 2.2.1. *If* $\mathbf{x}_0$ *is a regular zero point of equation (2.1) then there exists a smooth curve* $\mathbf{c} : \mathbb{R} \to \mathbb{R}^{n+1}$, $\mathbf{c}(0) = \mathbf{x}_0$ *with an open interval* $I \subset \mathbb{R}$, $0 \in I$ *s.t.*

$$F(\mathbf{c}(t)) = 0, \ \forall t \in I.$$

There is freedom in the choice of parameterization t. Of the several options for this parameterization, one may be tempted to parameterize $\mathbf{c}(t)$ by some *component* x^i of $\mathbf{x}$ where

$$x^i = \mathbf{x} \cdot \mathbf{e}^i$$

and $\mathbf{e}^i$ is a standard basis vector of $\mathbb{R}^{n+1}$ (this parameterization is sometimes called the *natural parameterization*).

Unfortunately, the natural parameterization often does not suffice for parameterizing $\mathbf{c}$. The problem can be seen in the following simple example. Consider the equation

$$F(x, y) = x^2 - y \tag{2.2}$$

with $n = 1$. If we parameterize by y we get $\mathbf{c}(y) = (\sqrt{y}, y)$. For notational convenience we will denote $\frac{\partial \mathbf{c}}{\partial y}$ by $\dot{\mathbf{c}}(y)$ and in this case we have that

$$\dot{\mathbf{c}}(y) = (\frac{1}{\sqrt{y}}, y) \tag{2.3}$$

In this case the tangent (2.3) is not defined at $y = 0$, and the solution set of equation (2.2) is not locally parameterizable by y there.

Let us examine more closely what happened in equation (2.2) by generalizing to the case where $n \geq 1$, and decomposing $\mathbf{x}_0$ into $\mathbf{x}_0 = (\mathbf{y}_0, \lambda_0)$ with $\mathbf{y}_0 \in \mathbb{R}^n$, and $\lambda_0 \in \mathbb{R}$. If $F'(\mathbf{y}_0, \lambda_0)$ has full rank then either

$$\det \left(\frac{\partial F(\mathbf{y}_0, \lambda_0)}{\partial \mathbf{y}_0} \right) \neq 0 \tag{2.4}$$

or

$$\det \left(\frac{\partial F(\mathbf{y}_0, \lambda_0)}{\partial \mathbf{y}_0} \right) = 0, \ \frac{\partial F(\mathbf{y}_0, \lambda_0)}{\partial \lambda_0} \notin \text{range} \left(\frac{\partial F(\mathbf{y}_0, \lambda_0)}{\partial \mathbf{y}_0} \right). \tag{2.5}$$

In the case of equation (2.4) we can locally parameterize by the Implicit Function theorem. In the case of equation (2.5) we call $(\mathbf{y}_0, \lambda_0)$ a *fold point* in the component λ_0, and, at a fold point, the component λ_0 cannot be guaranteed to be used as a local parameterization of $\mathbf{c}$.

Another, often better, choice is to parameterize by *arclength*. Given any parameterization $\mathbf{c}(t)$ we replace t with an arclength parameter s such that

$$ds = \| \mathbf{c}(t) \|_2 dt \tag{2.6}$$

where $\|\cdot\|_2$ denotes the Euclidean norm. In such a case we avoid the problems with natural parameterization, since arclength is a monotonic quantity and hence cannot fold. In addition there is the added convenience that

$$\| \dot{\mathbf{c}}(s) \|_2 = 1. \tag{2.7}$$

Given equation (2.7), $\dot{\mathbf{c}}(s)$ is uniquely defined modulo sign (which corresponds to the direction of traversal of $\mathbf{c}$). If we compute the derivative of the equation

$$F(\mathbf{c}(s)) = \mathbf{0}, \tag{2.8}$$

we get

$$F'(\mathbf{c}(s))\dot{\mathbf{c}}(s) = 0, \tag{2.9}$$

and hence, assuming full rank, $\dot{\mathbf{c}}(s)$ spans the null space of $F'(\mathbf{c}(s))$. Since, by equation (2.7), $\dot{\mathbf{c}}(s) \neq \mathbf{0}$ the $(m+1) \times (m+1)$ matrix

$$\begin{pmatrix} F'(\mathbf{c}(s)) \\ \dot{\mathbf{c}}(s) \end{pmatrix}, \tag{2.10}$$

is non-singular for all $s \in I$. If we choose our parameterization s such that

$$\det \begin{pmatrix} F'(\mathbf{c}(s)) \\ \dot{\mathbf{c}}(s) \end{pmatrix} > 0, \tag{2.11}$$

then $\mathbf{c}(s)$ is uniquely defined.

We can make one generalization to the previous discussion by considering a case where $\mathbf{c}(\mathbf{0}) = \mathbf{x}_0$ satisfies equation (2.1), but

$$\operatorname{rank}(F'(\mathbf{x}_0)) = n - 1. \tag{2.12}$$

In this case $\mathbf{x}_0$ is called a *simple singular point* or *simple bifurcation point* of equation (2.1) [14]. Canonically, bifurcation points are points of special interest, where, for example, two branches intersect. Their computation is not strictly part of continuation methods, as presented here, but it is a common addition to many implementations.

Generically, the solution set of equation (2.1) is a collection of one dimensional manifolds and their singular points, e.g. folds, bifurcation points, etc., embedded in $n+1$ dimensional space. We call this set of manifolds and singular points a *bifurcation manifold*, and any representation or visualization thereof we call a *bifurcation diagram*. Often, some of the components of the argument of F are specially designated as *parameters* based upon the structure of the underlying problem. In the case where $F : \mathbb{R}^{n+1} \to \mathbb{R}^n$ we normally say we have a single parameter.

2.3 Survey of Standard Single Parameter Numerical Continuation Methods

Single parameter continuation methods are roughly divided into two categories. First are *Predictor Corrector* (P-C) methods in which previous estimates of the solutions $[\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_n]$ to equation (2.1) are used to generate a new approximate solution $\mathbf{x}^*_{n+1}$. This approximate solution is then corrected using a root finding algorithm (such as Newton's method) with a given tolerance until we find a new solution $\mathbf{x}_{n+1}$. Second are *piecewise linear* or *simplicial* methods. In these methods one exactly follows a polygonal path which approximates $\mathbf{c}(s)$. The polygonal path is generated relative to an underlying *triangulation* of $\mathbb{R}^{n+1}$. In both of the following subsections we will somewhat follow the presentation in [2, 4].

2.3.1 Predictor Corrector Methods

Assume that we have a regular point $\mathbf{x}_0$ of $F(\mathbf{x})$. Let $\mathbf{c}(0) = \mathbf{x}_0$, then, by Theorem 2.2.1 there exists an open interval I with $0 \in I$ such that $\forall t \in I$ we have $F(\mathbf{c}(t)) = \mathbf{0}$, and $F'(\mathbf{c}(t))$ has rank n.

As we know from equation (2.8) the *tangent* $\dot{\mathbf{c}}(t)$ spans the null space of $F'(\mathbf{c}(t))$ and if $\mathbf{c}(t)$ is parameterized by arclength we have $\|\dot{\mathbf{c}}(s)\| = 1$. The above has all been for the case where $F(\mathbf{c}(s)) = \mathbf{0}$, but these ideas may be extended to a more general setting as follows.

Definition 2.3.1. *Let $A \in \mathbb{R}^{n\times(n+1)}$, and $\operatorname{rank}(A) = n$. Then the null space of A is one dimensional, and there is a unique vector $\boldsymbol{\tau}(A)$ such that*

- $A\boldsymbol{\tau} = 0$
- $\|\boldsymbol{\tau}\| = 1$
- $\det \begin{pmatrix} A \\ \boldsymbol{\tau}^T \end{pmatrix} > 0$

The vector $\boldsymbol{\tau}(A)$ is called the tangent vector induced by A.

We can now state the basic idea of P-C methods. Let $\mathbf{x}_i$ be a point such that $\|F(\mathbf{x}_i)\| < \epsilon$ for some given tolerance ϵ. From Theorem 2.2.1, if $\mathbf{x}_i$ is a regular value of $F(\mathbf{x})$ then there is a unique solution curve $\mathbf{c}(s)$ with $\mathbf{c}(0) = \mathbf{x}_i$ defined on some open interval $0 \in I$ which is a solution of the initial value problem

$$\dot{\mathbf{u}} = \boldsymbol{\tau}(F'(\mathbf{u})),\ \mathbf{u}(0) = \mathbf{x}_i. \tag{2.13}$$

Accordingly, to compute an approximation $\mathbf{x}_{i+1}^*$ to a new point along $\mathbf{c}(s)$ one can use methods from initial value problems in ODEs, such as Euler's method.

$$\mathbf{x}_{i+1}^* = \mathbf{x}_i + h\boldsymbol{\tau}(F'(\mathbf{x}_i)). \tag{2.14}$$

We call the procedure presented in the previous paragraph the *Predictor Step*. Bear in mind that $F(\mathbf{c}(s)) = \mathbf{0}$ and that we can use the contractive properties of $\mathbf{c}(s)$ to correct $\mathbf{x}_{i+1}^*$ e.g. by using Newton's method, to within a given tolerance. After this *Corrector Step* we have a new solution $\mathbf{x}_{i+1}$ such that $\| F(\mathbf{x}_{i+1}) \| < \epsilon$ and we may iterate the procedure. We also note that for $n \gg 1$ the corrector step can be very expensive computationally, and this computational expense will be the dominant factor in an efficient implementation of these algorithms.

We can now present several instructive examples of P-C methods which will display several of the issues involved. We start with one of the simpler continuation methods *natural parameter continuation*, which is closely related to the natural parameterization mentioned before. As in the natural parameterization we decompose the solution into two parts, $\mathbf{x}_0 = (\mathbf{y}_0, \lambda_0)$. Given $\mathbf{x}_0$ we construct a new point

$$\mathbf{x}_1^* = (\mathbf{y}_0, \lambda_0 + \delta s), \tag{2.15}$$

where δs is some given step-size. In general, there is no reason to believe that $F(\mathbf{x}_1^*) = 0$, but $\mathbf{x}_1^*$ may be used as a starting guess for some correction procedure which finds an approximate solution to equation (2.1). The correction procedure might solve the underdetermined equation (2.1) directly, for example by using the Moore-Penrose Pseudo-Inverse [21]. On the other hand, a more interesting approach is to construct a new function $G(\mathbf{x}) = (F(\mathbf{x}),\ \lambda - (\lambda_0 + \delta s))$ with the following properties. $G(\mathbf{x}) = \mathbf{0}$ is a square system with the same solution set as $F(\mathbf{x}) = \mathbf{0}$ except that $G(\mathbf{x}) = \mathbf{0}$ requires that $\lambda = (\lambda_0 + \delta s)$. We may apply any appropriate algorithm, such as Newton's Method, to solve it, but with the added advantage that $\mathbf{x}_1^*$ is a good starting guess. Solving $G(\mathbf{x}) = \mathbf{0}$ creates a new point $\mathbf{x}_1$ such that $F(\mathbf{x}_1) = \mathbf{0}$ and $\lambda_1 = \lambda_0 + \delta s$. This algorithm is illustrated schematically in Figure 2.1.

One can easily construct a better algorithm than the above. For example, instead of making the guess as in equation (2.15), we can make a higher order approximation, for example, we can try

$$\mathbf{x}_1^* = \mathbf{x}_0 + \frac{\dot{F}(\mathbf{x}_0)}{||F(\mathbf{x}_0)||}\delta s \tag{2.16}$$

and this method, with the same G as above, is illustrated in Figure 2.2. If we do not wish to compute $\dot{F}(\mathbf{x}_0)$, and we have two solutions $\mathbf{x}_0$ and $\mathbf{x}_1$, then we can

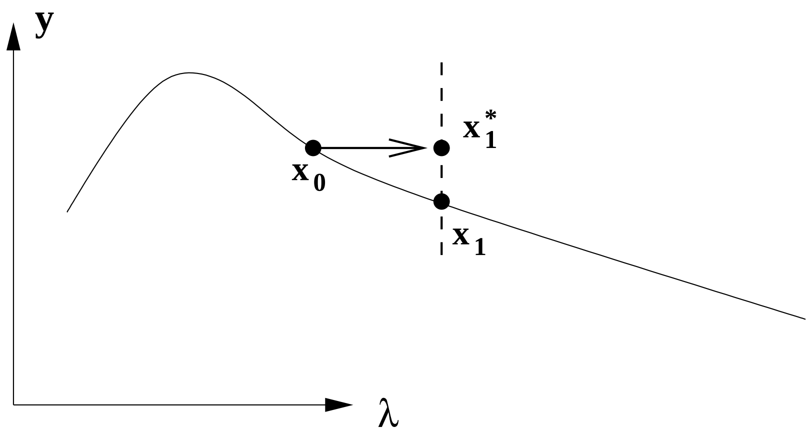

Figure 2.1: This figure shows an illustration of natural parameter continuation. The point on the left represents $\mathbf{x}_0$. The dotted line shows those points which have the property that $\lambda_1 = \lambda_0 + \delta s$. The point marked $\mathbf{x}_0^*$ is the point at which the correction procedure will start and the point marked $\mathbf{x}_1$ is the desired new solution to $G(\mathbf{x}) = \mathbf{0}$.

choose a secant approximation

$$\mathbf{x}_1^* = \mathbf{x}_0 + \frac{F(\mathbf{x}_1) - F(\mathbf{x}_0)}{\|\mathbf{x}_1 - \mathbf{x}_0\|}\delta s. \tag{2.17}$$

Similarly, if we have a set of solutions $\mathbf{x}_0, ..., \mathbf{x}_n$ we can construct an even higher order approximation.

At this point, we note that natural parameter continuation suffers from the same problem as the natural parameterization; it fails at fold points. We can construct an algorithm which does not fail at fold points by choosing a better *constraint function* instead of $\lambda - (\lambda_0 - \delta s)$. One possibility is to use what is called *arclength continuation*. In this case we choose G such that $G(\mathbf{x}) = (F(\mathbf{x}),\ \|\mathbf{x}_0 - \mathbf{x}\| - \delta s)$. This method is shown in Figure 2.3. Such an algorithm allows us to continue around folds, but has the disadvantage that there are always two points on the constraint manifold to which the root finding algorithm may converge.

Our last example has all of the advantages of arclength continuation, but removes the problem of multiple solutions by choosing G as $G(\mathbf{x}) = (F(\mathbf{x}),\ (\dot{F}(\mathbf{x}_0) - \mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0 - \delta s)$. This method is shown in Figure 2.4 and is known as *pseudo-arclength continuation.*

As can be seen, there is much freedom in the construction of a one dimensional continuation algorithm and many of these issues arise again when we consider multi-parameter continuation algorithms. The pseudo-arclength algorithm with

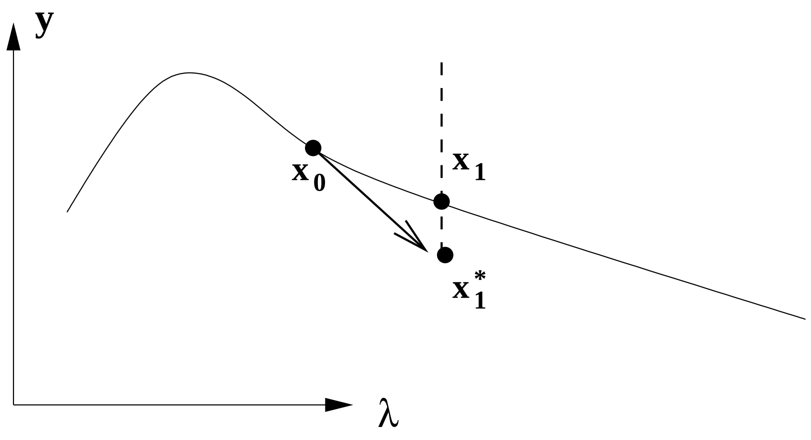

Figure 2.2: This figure shows natural parameter continuation using a higher order initial guess. Just as in Figure 2.1, the point on the left represents $\mathbf{x}_0$. The dotted line shows those points which have the property that $\lambda_1 = \lambda_0 + \delta s$. The point marked $\mathbf{x}_0^*$ is the point at which the correction procedure will start and the point marked $\mathbf{x}_1$ is the desired new solution to $G(\mathbf{x}) = \mathbf{0}$. The difference between this method and the one shown in Figure 2.1 is that the starting point for the correction procedure is generally closer to the solution, because of the higher order guess.

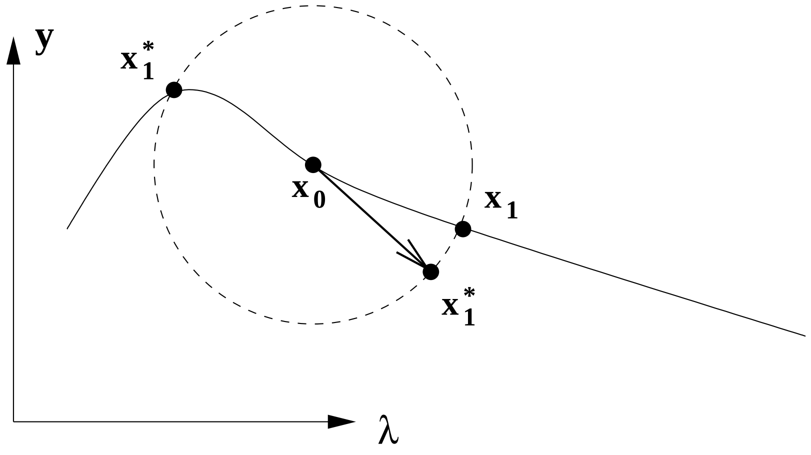

Figure 2.3: This figure illustrates a more complex constraint function, shown by the dotted circle, than that presented in Figure 2.1 and Figure 2.2. The advantage of this constraint function is that it works in the presence of folds. The disadvantage is that, generally, there are two candidate solutions to $G(\mathbf{x}) = \mathbf{0}$, here both designated by $\mathbf{x}_1^*$.

either a tangent, secant, or higher-order approximations has been used successfully in several codes [16, 50, 7, 44].

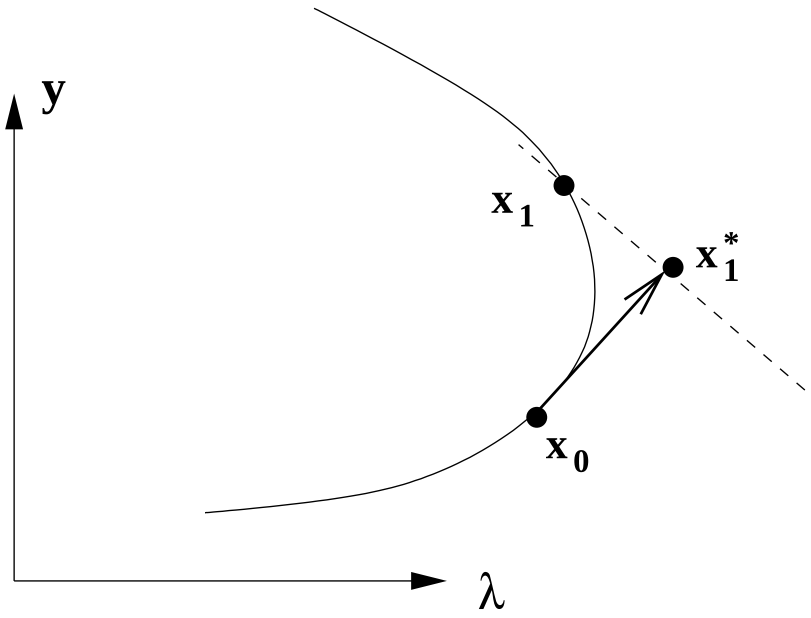

Figure 2.4: This figure illustrates the pseudo-arclength continuation algorithm. The difference between this algorithm and the algorithms shown in Figure 2.1 and Figure 2.2 is that restriction function changes based upon the initial guess to always be orthogonal to the guess vector, shown by the arrow. As can be seen schematically here, this algorithm is able to proceed past folds in λ.

2.3.2 Simplicial Methods

We provide here a brief description of *simplicial* methods for problems of the general type of equation (2.1). The presentation will be very cursory since, as will be seen, these methods are not appropriate for the class of problems that mainly concern us, namely discretized differential equations. These methods are described for completeness and to introduce the idea of simplices, which will be used in later chapters. It is also a useful exercise to compare the simplicial methods to the P-C methods subject to the criterion of being efficient for large systems.

We begin with the following definition

Definition 2.3.2. *A set of points $\mathbf{x}_0, ..., \mathbf{x}_k$ in $\mathbb{R}^n$ are said to be affinely independent if the differences $\mathbf{x}_1 - \mathbf{x}_0, ..., \mathbf{x}_k - \mathbf{x}_0$ are linearly independent vectors in $\mathbb{R}^k$. Obviously, $k \leq n$.*

The convex hull of a set of affinely independent points is the main computational element in simplical methods.

Definition 2.3.3. *Given a set of $k+1$ affinely independent points $\mathbf{x}_0, ..., \mathbf{x}_k$ we* *define the k dimensional simplex $\boldsymbol{\alpha}$ with vertices $\mathbf{x}_0, ..., \mathbf{x}_k$ to be the set*

$$\left\{\mathbf{x} = \sum_{i=0}^{k} \alpha_i \mathbf{x_i} \mid \alpha_i \geq 0 \ \ , \forall i = 0, ..., k, \sum_{i=0}^{k} \alpha_i = 1\right\} \tag{2.18}$$

The α_i are called the barycentric coordinates of $\mathbf{x}$.

Given a simplex $\boldsymbol{\alpha}$ defined by $S = \{\mathbf{x}_0, ..., \mathbf{x}_k\}$ we note that any subset $S' \subseteq S$ also defines a simplex (of lower dimension). Several such sub-simplices are important and are given names.

Definition 2.3.4. *Let $\boldsymbol{\alpha}$ be the simplex defined by $\mathbf{x}_0, ..., \mathbf{x}_k$. If $\mathbf{y}_0, ..., \mathbf{y}_l \subset \mathbf{x}_0, ..., \mathbf{x}_k$ then we call the simplex defined by $\mathbf{y}_0, ..., \mathbf{y}_l$ a l-face of $\boldsymbol{\alpha}$. A $(k-1)$-face of $\boldsymbol{\alpha}$ is called a facet. A 0-face of $\boldsymbol{\alpha}$ is called a vertex.*

We turn our attention to special sets of simplices, specifically those which form a triangulation of $\mathbb{R}^{n+1}$.

Definition 2.3.5. *A triangulation of $\mathbb{R}^{n+1}$ is a set T of $n+1$ dimensional simplices satisfying*

- $\bigcup_{\boldsymbol{\alpha} \in T} \boldsymbol{\alpha} = \mathbb{R}^{n+1}$
- *If $\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in T$ then $\boldsymbol{\alpha}_1 \cap \boldsymbol{\alpha}_2$ is empty, or a facet.*
- *Any compact subset of $\mathbb{R}^{n+1}$ meets only finitely many simplices of T.*

Accordingly, let T be a triangulation of $\mathbb{R}^{n+1}$, and let F be as in equation (2.1). We denote by F_T the affine map which agrees with F on the vertices of T,

$$F_T(\mathbf{x}) = \sum_{i=1}^{n+1} \alpha_i F(\mathbf{x}_i) \ \ for \ \ \mathbf{x} = \sum_{i=1}^{n+1} \alpha_i \mathbf{x}_i. \tag{2.19}$$

The idea is to use the approximation F_T to compute zeros of F. The correspondence between solutions of $F(\mathbf{x}) = \mathbf{0}$, and $F_T(\mathbf{x}) = \mathbf{0}$ is given by the following definition and theorem.

Definition 2.3.6. *A facet of a simplex in a triangulation T is called completely labeled if and only if $\forall \epsilon > 0$ it contains solutions $\mathbf{x}_\epsilon$ for the equation $F_T(\mathbf{x}) = \mathbf{x}_\epsilon$ where $\|\mathbf{x}_\epsilon\| < \epsilon$.*

Basically, this definition ensures that there is a solution $\mathbf{x}_0$ to $F_T(\mathbf{x}_0) = \mathbf{0}$ under a certain class of perturbations (see [2] for details).

Theorem 2.3.1. *Let T be a triangulation of $\mathbb{R}^{n+1}$, $\boldsymbol{\alpha} = \mathbf{x}_0, ..., \mathbf{x}_n$ be a facet of a simplex of T, and consider the labeling matrix*

$$L(\boldsymbol{\alpha}) = \begin{pmatrix} 1 & \cdots & 1 \\ F(\mathbf{x}_0) & \cdots & F(\mathbf{x}_n) \end{pmatrix}. \tag{2.20}$$

Then the facet $\boldsymbol{\alpha}$ is completely labeled if and only if both of the following two conditions hold:

- *$L(\boldsymbol{\alpha})$ is non-singular*
- *The first nonzero entry of any row of $L(\boldsymbol{\alpha})^{-1}$ is positive*

The simplicial continuation process consists of finding sets of completely labeled facets, and this process is made tractable by the following observation.

Observation 2.3.1. *A $n+1$ simplex has either no, or exactly two completely labeled facets.*

Accordingly, the simplicial method can be outlined as follows. Begin with a simplex $\boldsymbol{\alpha}_i \in T$ with a completely labeled facet β_i. Find another simplex $\boldsymbol{\alpha}_{i+1}$ such that $\boldsymbol{\alpha}_i \cap \boldsymbol{\alpha}_{i+1} = \beta_i$. Find the second completely labeled facet $\beta_{i+1} \subset \boldsymbol{\alpha}_{i+1}$ and iterate.

Simplicial continuation will be revisited briefly in chapter Chapter 5 to look more closely at its efficiency as compared to P-C methods.

Chapter 3

Multi Parameter Continuation Methods

3.1 Introduction

Now that we have developed the single parameter case of equation (2.1), we broaden our attention to the multi-parameter case where

$$F(\mathbf{x}) = \mathbf{0}, \ F : \mathbb{R}^{n+m} \to \mathbb{R}^n, \ n, m \in \mathbb{Z}, n \geq 1, m > 1. \tag{3.1}$$

The properties of the solution set of this equation are similar to the case where $m = 1$, but some significant differences will be pointed out in Section 3.2. In Section 3.3 we will derive a simple method for computing such solution sets, using the ideas from the one parameter methods in Chapter 2. To our knowledge, this method has not been previously discussed in the literature. Conceptually, the method is very simple, and the only difficulties are implementation issues, which we will discuss in detail. The method will be shown to have several drawbacks, but we will see that they are not critical when interactive control of the computation is envisioned. In Section 3.4 we will survey two other multi-parameter continuation methods, one due to Rheinboldt [43] and one to Henderson [22], and discuss their advantages and disadvantages.

3.2 The Solution Set

To begin, we may use Definition 2.2.1, without modification, to define a regular zero point in the multi-parameter equation (3.1). On the other hand, Theorem 2.2.1 must be modified somewhat for the more general case.

Theorem 3.2.1. *If $\mathbf{x}_0$ is a regular zero point of equation (3.1) then there exists a smooth function $\mathbf{c}(\mathbf{0}) = \mathbf{x}_0$, $\mathbf{c} : \mathbb{R}^m \to \mathbb{R}^{n+m}$ and an open neighborhood $U \subset \mathbb{R}^m$, $\mathbf{0} \in U$ s.t.*

$$F(\mathbf{c}(\mathbf{t})) = \mathbf{0}, \ \forall \mathbf{t} \in U,$$

where $\mathbf{c}(\mathbf{t})$ now defines a smooth m-manifold embedded in $\mathbb{R}^{n+m}$.

There is an appropriate generalization of the tangent vector defined in Definition 2.3.1.

Definition 3.2.1. *If $A \in \mathbb{R}^{n\times(n+m)}$ and $\operatorname{rank}(A) = n$ then the null space of A is m-dimensional. This space is called the tangent space induced by A and is denoted by $\boldsymbol{\tau}(A)$.*

Fold points can be treated similarly to the case where $m = 1$. Consider a decomposition, as before, with $\mathbf{x}_0 = (\mathbf{y}_0, \boldsymbol{\lambda}_0)$, $\mathbf{y}_0 \in \mathbb{R}^n$, and $\boldsymbol{\lambda}_0 \in \mathbb{R}^m$. If $F'(\mathbf{y}_0, \boldsymbol{\lambda}_0)$ has full rank then either

$$\det\left(\frac{\partial F(\mathbf{y}_0, \boldsymbol{\lambda}_0)}{\partial \mathbf{y}_0}\right) \neq 0 \tag{3.2}$$

or

$$\operatorname{rank}\left(\frac{\partial F(\mathbf{y}_0, \boldsymbol{\lambda}_0)}{\partial \mathbf{y}_0}\right) = n - p,\ p \leq m, \tag{3.3}$$

but

$$\frac{\partial F(\mathbf{y}_0, \boldsymbol{\lambda}_0)}{\partial \boldsymbol{\lambda}_0} \tag{3.4}$$

has p linearly independent columns $\mathbf{v_1}, ..., \mathbf{v_p}$ such that

$$\nexists c_1, ..., c_p \in \mathbb{R} \tag{3.5}$$

with

$$\sum_{i=1}^{p} c_i \mathbf{v_i} \in \operatorname{range}\left(\frac{\partial F(\mathbf{y}_0, \boldsymbol{\lambda}_0)}{\partial \mathbf{y}_0}\right). \tag{3.6}$$

In the case where $p > 1$, we say that p components are folding simultaneously.

3.3 The Single Parameter With Switching (SPWS) Method

The first multi-parameter continuation algorithm we will present is a simple generalization of the one parameter P-C methods we saw in Chapter 2 which we will call the Single Parameter With Switching (SPWS) method. To our knowledge, the SPWS method is not explicitly discussed in the literature, perhaps because it is both rather simple and has several obvious deficiencies. On the other hand, in the domain of interactive computations, it is quite practical. We believe it to be especially useful in two main areas. First, since it utilizes one-parameter algorithms as the basis for its computations, it may make use of the one-parameter algorithms and software which already exist in the literature. Second, one may implement it efficiently using three distinct levels of parallelism (see Section 5.3).

Assume that $\mathbf{x}_0$ is a solution to equation (3.1). Arbitrarily decompose $\mathbf{x}_0$ into $(\mathbf{y}_0, \lambda_0^1, ..., \lambda_0^m)$ with $\mathbf{y}_0 \in \mathbb{R}^n$, and $\lambda_0^i \in \mathbb{R}$, $i = 1, ..., m$. Define $\boldsymbol{\lambda}_0^i(\alpha)$ to be

$$\boldsymbol{\lambda}_0^i(\alpha) = \{\lambda_0^1, ..., \lambda_0^{i-1}, \alpha, \lambda_0^{i+1}, ..., \lambda_0^m\}. \tag{3.7}$$

We now may define m functions

$$G_{\mathbf{x}_0}^i : \mathbb{R}^{n+1} \to \mathbb{R}^n \tag{3.8}$$

by

$$G_{\mathbf{x}_0}^i(\mathbf{y}, \alpha) = F(\mathbf{y}, \boldsymbol{\lambda}_0^i(\alpha)). \tag{3.9}$$

We have chosen the notation $G_{\mathbf{x}_0}^i$ to emphasize that the definition of $G_{\mathbf{x}_0}^i$ depends both on F and the given solution $\mathbf{x}_0$. In other words, we create each function $G_{\mathbf{x}_0}^i$ from F by fixing the parameters $\lambda_0^1, ..., \lambda_0^{i-1}, \lambda_0^{i+1}, ..., \lambda_0^m$ and only allowing λ_0^i to vary.

The idea of the SPWS algorithm is to use the one parameter P-C algorithms described in Chapter 2 to solve multiple problems of the form of equation (3.9) to approximate the m-dimensional bifurcation manifold. The main problem which arises is an appropriate definition of the connectivity of the computed points. We first describe an algorithm for the case $m = 2$. The algorithm for $m > 2$ will then be a straightforward generalization.

Assume we have a solution $\mathbf{x}_0$ to equation (3.1), with $m = 2$, and let us define $G_{\mathbf{x}_0}^1$ and $G_{\mathbf{x}_0}^2$ as in equation (3.9). If we assume that our one parameter algorithm is parameterized by arclength (see equation (2.6)), then there are four problems to which we may apply our one parameter algorithm. We may continue both $G_{\mathbf{x}_0}^1$ and $G_{\mathbf{x}_0}^2$, in both the positive and negative directions (as defined by equation (2.11)). As shown in Figure 3.1, we may use this procedure to compute four additional points on the bifurcation diagram.

Under mild assumptions (which are discussed in Section 3.3.1), these five points can now be considered as an approximation of a small patch of the two dimensional bifurcation manifold of equation (3.1). Suppose we now wish to extend this small patch and create an approximation of a larger portion of the bifurcation manifold. Obviously, one possibility is to apply the same procedure to each of the new points $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$, and $\mathbf{x}_4$ as we did to point $\mathbf{x}_0$. We may compute G^1 and G^2 for each of the points $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$, and $\mathbf{x}_4$, and then perform

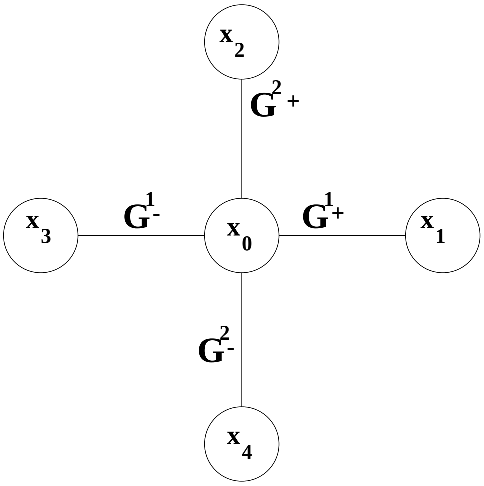

Figure 3.1: This figure shows a schematic of the first step of the SPWS algorithm. The point marked $\mathbf{x}_0$ is the initial point at which the algorithm is started. Two continuation steps were performed on $G^1_{\mathbf{x}_0}$, one in the positive direction and one in the negative direction, to compute points $\mathbf{x}_1$ and $\mathbf{x}_3$. Similarly two continuation steps were performed on $G^2_{\mathbf{x}_0}$ to compute points $\mathbf{x}_2$ and $\mathbf{x}_4$. For consistency, all figures which show the SPWS algorithm will show continuation steps using $G^1_{\mathbf{x}_0}$ as horizontal lines and those using $G^2_{\mathbf{x}_0}$ vertical lines. Similarly, the positive directions will be right and up, and the negative direction will be left and down.

sixteen new continuation steps, four at each point. Unfortunately, the above procedure is somewhat inefficient in that it generally produces some points that are very close together. For example, consider the point $\mathbf{x}_1$ in Figure 3.1. It was computed by starting at $\mathbf{x}_0$, using $G^1_{\mathbf{x}_0}$, and taking a continuation step in the positive direction. If we now start at $\mathbf{x}_1$, use the function $G^1_{\mathbf{x}_1}$, and take a negative step, we proceed back to $\mathbf{x}_0$. We point out that, even though $G^1_{\mathbf{x}_0}$ and $G^1_{\mathbf{x}_1}$ are identical (since we have not varied λ^2_0), we may not come exactly back to $\mathbf{x}_0$. Depending on the implementation of the algorithm, it may make different step-size choices when starting at $\mathbf{x}_0$ and $\mathbf{x}_1$. Nevertheless, for most well behaved one parameter algorithms, the point generated by starting at $\mathbf{x}_1$ and taking a negative step using $G^1_{\mathbf{x}_1}$ is close to $\mathbf{x}_0$ and we do not wish to compute it.

We now consider a two dimensional cubic lattice, as shown in Figure 3.2, which we will utilize to avoid calculating points which are close to points we have already calculated. The idea is to use this lattice as a template for our calculation by performing a continuation step for problem G^1 for each horizontal edge, and a continuation step for problem G^2 for each vertical edge. Each vertex of the lattice represents a point on the bifurcation manifold, and each edge represents a possible continuation calculation. Figure 3.2 shows how this procedure is used to avoid unnecessary calculations. Basically, when the vertices at both ends of an edge have already been computed by other continuation calculations, the continuation calculation represented by the edge need not be performed. The above process may be implemented in several interesting ways. For example, see Figure 3.3. We wish to emphasize that the cubic lattice shown in Figure 3.2 denotes topology; it can only be considered as a cubic lattice in parameter space[1] if one is utilizing natural parameter continuation with constant step sizes in both λ^1 and λ^2. Otherwise, the difference in λ^i between two points in the cubic lattice, which share an edge, may not be constant. In fact, in the presence of folds, the sign of the difference may well change.

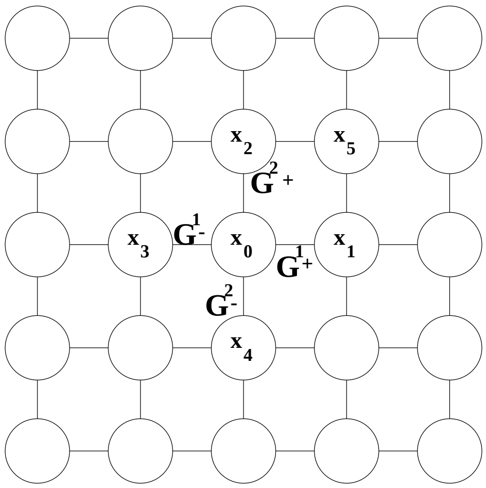

Figure 3.2: This figure shows a two dimensional cubic lattice into which the calculation from Figure 3.1 has been embedded. Suppose we continue the procedure by calculating the point $\mathbf{x}_5$. Using the convention described in Figure 3.1, we may start at point $\mathbf{x}_2$ and perform a continuation step using $G^1_{\mathbf{x}_2}$ in the positive direction. After performing such a step, there is no need to perform a continuation step, starting at $\mathbf{x}_1$, and using $G^2_{\mathbf{x}_1}$ in the positive direction, since that calculation will result in a point close to $\mathbf{x}_5$.

3.3.1 Advantages and Disadvantages of the SPWS Method

There are a number of technical issues which arise with the SPWS method. First, if any of the m parameters fold, or have higher order singularities, then the underlying one parameter algorithm might fail. We can solve this problem by utilizing a sufficiently robust one parameter algorithm.

[1] In this case, we denote by *parameter space* the space $\mathbb{R}^2$ where the standard basis vector $\mathbf{e}^1$ represents the values of λ^1_0 and the standard basis vector $\mathbf{e}^2$ represents the values of λ^2_0.

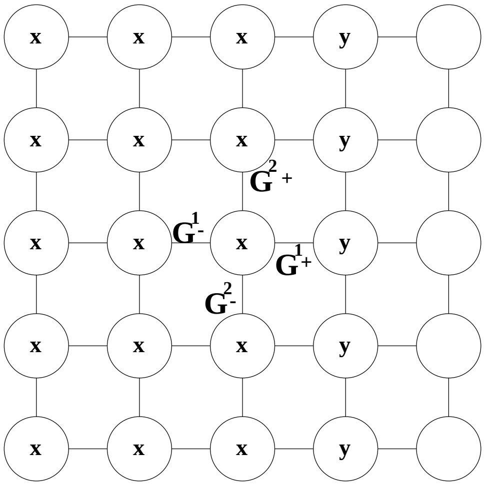

Figure 3.3: This figure shows an example of one possible implementation of the SPWS algorithm. Assume that all of the points marked by $\mathbf{x}$ have already been computed. The points marked with $\mathbf{y}$ may then be computed by starting at each of the rightmost $\mathbf{x}$ points, and then performing a positive step using $G^1_{\mathbf{x}}$. Using this method the computed region will always retain a rectangular topology.

Second, the assumption in Figure 3.2, that two different continuation calculations result in new solutions, which are in some sense close, may fail. For every point on the bifurcation manifold that we compute there is some *path* or sequence of continuation steps, which were performed to create that point. For example, in Figure 3.4 we see that two different sequences of continuation computations may be used to create $\mathbf{x}_5$. If the bifurcation manifold is not sufficiently smooth, the two values computed for $\mathbf{x}_5$ can be arbitrarily far apart (see caption of Figure 3.4).

Finally, the SPWS method may not actually compute the desired bifurcation manifold, but some sub-manifold thereof, for example if $\mathbf{x}_0, \mathbf{x}_1$, and $\mathbf{x}_2$ from Figure 3.1, have the property that the two vectors $\mathbf{x}_0 - \mathbf{x}_1$ and $\mathbf{x}_0 - \mathbf{x}_2$ were not linearly independent. Fortunately, one can test for this problem and report any degeneracies.

As an aside, we note that these degeneracies are an example of a more general class of problems which arise in multidimensional algorithms. In these algorithms the mesh on which the bifurcation manifold is computed may become too coarse or too fine unintentionally. The mesh size changes not because of any numerical or visualization concern, but purely because of the structure of the equations and the specific algorithm chosen to approximate the mesh. The main point is that we wish to detect and avoid the construction of a triangulation in which the triangles have small interior angles, since these can often be caused by problems

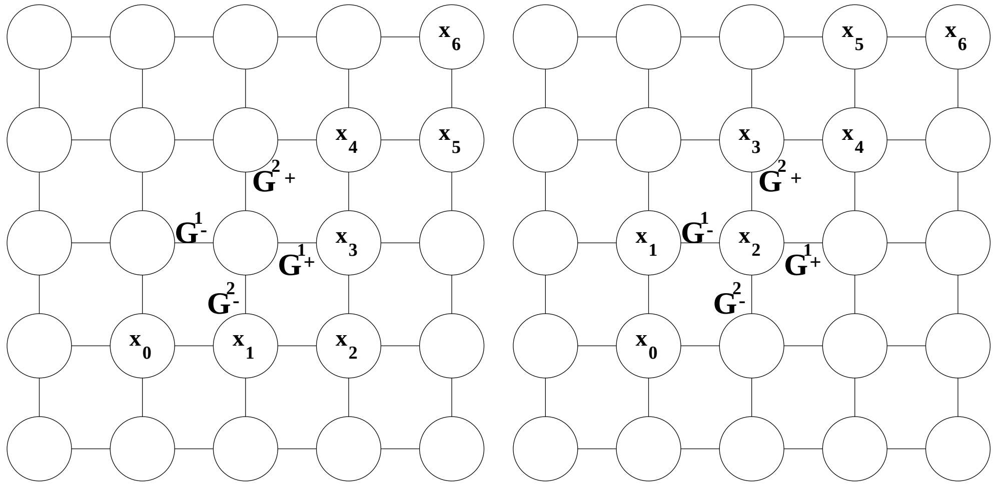

Figure 3.4: This figure shows two different sequences of continuation computations, or paths, each starting at point $\mathbf{x}_0$ and ending at point $\mathbf{x}_6$. If the bifurcation manifold is not sufficiently smooth, the actual value of parameters and the solution at the point $\mathbf{x}_6$ may be arbitrarily different depending on the path that was taken.

within the manifold approximation algorithm itself (e.g. non-commensurate step-size selection for each of the parameters or a non-orthogonal basis for the tangent space). For example, consider $\mathbf{x}_0, \mathbf{x}_1$, and $\mathbf{x}_2$ from Figure 3.1, and the two vectors $\mathbf{x}_0 - \mathbf{x}_1$ and $\mathbf{x}_0 - \mathbf{x}_2$. We call two such vectors the *basis of the mesh*. If the basis of the mesh is far from orthonormal, then we call the mesh *ill-conditioned*. If the vectors which make up the basis of the mesh are close to collinear, then the triangles which make up the mesh of the bifurcation diagram will have two very small angles and one very obtuse angle (for example, see Figure 3.5). Similarly, if the vectors which make up the basis of the mesh have very different lengths, then the triangles which make up the mesh will have two angles very close to ninety degrees and one angle which is very small (for example, see Figure 3.5).

On the other hand, there are several advantages of the SPWS algorithm. First, step-size control is taken care of by the underlying one parameter algorithm, so it may vary over the mesh. Second, folds and bifurcations (of the one parameter problem) may be detected using the one parameter code. Finally, it is quite easy to implement. In fact, we have implemented it in such a way that a variety of one parameter P-C algorithms may easily be used. We also note that the use of a cubic lattices allows us to easily generate a triangulation of the computed points which may be used in the graphical rendering of the bifurcation diagram.

The deficiencies of the SPWS algorithm make it unsuitable for any application in which it must run in batch mode. On the other hand, in interactive computa-

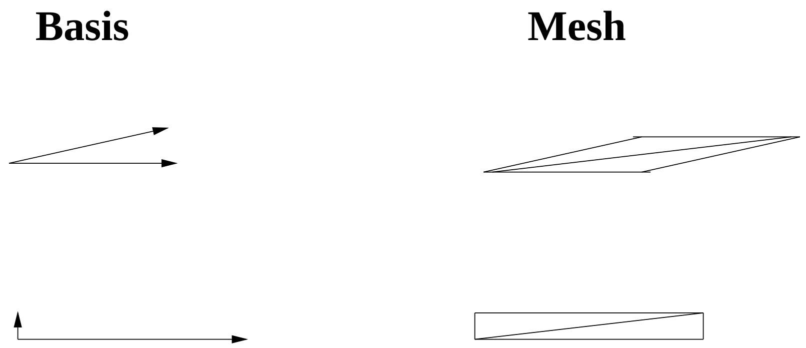

Figure 3.5: This figure shows two ill-conditioned meshes. On the left are basis vectors for the mesh and on the right are small example patches of the produced meshes. The top basis vectors are close to collinear and the bottom basis vectors have very different lengths. In each case, the produced triangulation has triangles with small angles.

tions the speed and flexibility of the above algorithm are quite useful while any of the problems it has are easily observable and correctable by user interactive, e.g. by starting a new continuation calculation. The use of the interactive software discussed in Chapter 7 allows the SPWS algorithm, which was apparently regarded as impractical, to be applied to a wide class of problems (e.g. see Chapter 8).

3.4 Survey of Multi-parameter Algorithms

Several multi-parameter continuation algorithms exist in the literature which are superior to the SPWS algorithm in various ways. We present here two algorithms from the literature, namely Rheinboldt's Moving Frame algorithm [43] and Henderson's Overlapping Disk algorithm [22].

3.4.1 Rheinboldt Moving Frame

We first examine an algorithm due to Rheinboldt [43]. Given a system of the form of equation (3.1) we compute a basis for the null space of $F'(\mathbf{x})$, which is the subspace tangent to the bifurcation manifold of F at $\mathbf{x}$. We will denote by $U(\mathbf{x})$ a matrix whose columns are basis vectors which span the null space of $F'(\mathbf{x})$. In addition, we will denote by $U(\mathbf{x})_i$ the ith column of $U(\mathbf{x})$.

Suppose that we had a method for producing a basis that varied continuously with $\mathbf{x}$. We could then construct an m-dimensional subspace T of $\mathbb{R}^{n+m}$ such that $T \cap \boldsymbol{\tau}(F'(\mathbf{x}_0))^{\perp} = \mathbf{0}$ where $\boldsymbol{\tau}(F'(\mathbf{x}_0))^{\perp}$ represents the orthogonal complement of the tangent space $\boldsymbol{\tau}(F'(\mathbf{x}_0))$. T is then said to induce a local coordinate system on an open subset M of $F^{-1}(\mathbf{0})$. An obvious example of such a subspace is $\boldsymbol{\tau}(F'(\mathbf{x}_0))$ with M being some open neighborhood of $\mathbf{x}_0$. We could then triangulate T, using some type of reference triangulation, and then map this triangulation onto the bifurcation manifold, using some type of correction procedure, such as Newton's method. As an example, consider the function

$$F(x,y,z) = x^2 + y^2 + z^2 - 1,\ x,y,z \in \mathbb{R} \tag{3.10}$$

(i.e. the unit sphere) and choose

$$T = \{(x,y,z)|z = 0\} \tag{3.11}$$

In this case, T induces a local coordinate system on all of equation (3.10) except for the great circle where $z = 0$.

There exist many ways in which one may compute such a basis. For example, one can compute the QR factorization of $F'(\mathbf{x})$. The QR factorization of a matrix A consists of constructing an orthogonal matrix Q (i.e. $QQ^T = I$) and an upper-triangular matrix R such that $A = QR$. If the matrix R has positive diagonal entries then the factorization is unique [21]. If one decomposes Q into two pieces $Q = (Q_1, Q_2)$ with Q_1 having n columns, then the columns of Q_2 then form the desired basis. Unfortunately, for this algorithm and many similar ones, the generated basis does not depend continuously on $\mathbf{x}$.

The idea of Rheinboldt's Moving Frame algorithm is to compute a basis for the null space of $F'(\mathbf{x})$ using an algorithm such as the QR factorization, and then to "rotate" each basis so that they vary continuously with respect to $\mathbf{x}$. The foundation of the Moving Frame algorithm can then be found in the following theorem [43].

Theorem 3.4.1. *Let M be an open subset of $F^{-1}(\mathbf{0})$ such that a given m-dimensional subspace T of $\mathbb{R}^{n+m}$ induces a local coordinate system on it. For any $\mathbf{x}_0 \in M$ let $U(\mathbf{x}_0)$ be any orthonormal basis of $\boldsymbol{\tau}(F'(\mathbf{x}_0))$. If one computes the singular value decomposition of the matrix $U(\mathbf{x}_0)^T T = A\Sigma B^T$ then the mapping $\mathbf{x}_0 \in M \to U(\mathbf{x}_0)AB^T$ is of class C^{r-1} whenever F is of class C^r.*

Since the above theorem gives us a smoothly varying set of bases over a subset $M \subseteq F^{-1}(\mathbf{0})$ we may compute this subset using the following algorithm. First, generate a reference triangulation of $\mathbb{R}^m$. Second, use the bases provided by Theorem 3.4.1 to map segments of it onto $\mathbf{x}_0 + \boldsymbol{\tau}(F'(\mathbf{x}_0))$, for a point $\mathbf{x}_0 \in M$. Some procedure, such as Newton's method, can then be used to map the segments onto M.

The advantages of this algorithm are threefold. First, it does not rely on any underlying one parameter algorithm, hence we do not have to be concerned with folds in individual parameters or solution components. Second, there is a large freedom in the selection of T, the reference triangulation of $\mathbb{R}^m$, which we may use to implement the algorithm very efficiently. Third, this algorithm may easily be used for any $m > 1$.

The disadvantage of this algorithm is that Theorem 3.4.1 only holds as long as T induces a local coordinate system. On areas of $F^{-1}(\mathbf{0})$ where this does not hold, the algorithm may fail. Also, since a fixed triangulation of T is chosen at the onset of the algorithm, step-size control is difficult.

As an example of a bifurcation manifold where the Rheinboldt algorithm fails we refer back to equation (3.10). No matter what point $\mathbf{x}_0$ is chosen as the starting point of the algorithm, no single space T induces a local coordinate system everywhere on the sphere. There is always a circle of points x at the "equator" where $T \cap \boldsymbol{\tau}(F'(\mathbf{x}))^{\perp} \neq \mathbf{0}$. Accordingly, the current continuation must be halted there and a new one begun by the definition of a new T, chosen, if possible, to maintain connectivity.

3.4.2 Henderson Overlapping Disks

The last algorithm which we will describe is one due to Henderson [22]. This algorithm is interesting in that it approaches the computation of the bifurcation manifold differently than the two other algorithms presented in this chapter. The key issue in multi-parameter algorithms is determining "where to compute next". The two previous algorithms attempt to compute sets of bases that are in some way continuous, and this continuity of the bases enables the algorithm to determine where to compute at each step of the algorithm. Henderson's Overlapping Disk algorithm, on the other hand, uses the geometry of the computed manifold itself to determine where to compute next.

In the presentation of this algorithm we will restrict our attention to the case where $m = 2$. As presented in [22] the algorithm does not have an easy generalization to $m > 2$, though an extension is under development [23].

Suppose we have a point $\mathbf{x}_0$ such that $F(\mathbf{x}_0) = 0$. We then compute $U(\mathbf{x}_0)$ (as defined in Section 3.4.1) by any convenient procedure. As pointed out before, there is freedom in the construction of $U(\mathbf{x}_0)$, but any definition in which the basis vectors are orthonormal will suffice for our needs. Constants r_s and r_t can now be chosen, for example using Taylor's theorem, such that if

$$\mathbf{x} = \mathbf{x}_0 + sU(\mathbf{x}_0)_1 + tU(\mathbf{x}_0)_2, \ |\, s \,| < r_s, \ |\, t \,| < r_t \tag{3.12}$$

then $|F(\mathbf{x})| < \epsilon$ for some given ϵ (see Figure 3.6).

Consider now the ellipse $E_{\mathbf{x}_0} \subset \boldsymbol{\tau}(F'(\mathbf{x}_0))$ around $\mathbf{x}_0$ defined by equation (3.12), and take any point on its boundary $\mathbf{x}_1^*$. Now, project this point onto $F^{-1}(\mathbf{0})$, using any convenient method, and compute the ellipse $E_{\mathbf{x}_1}$ around this point as well.

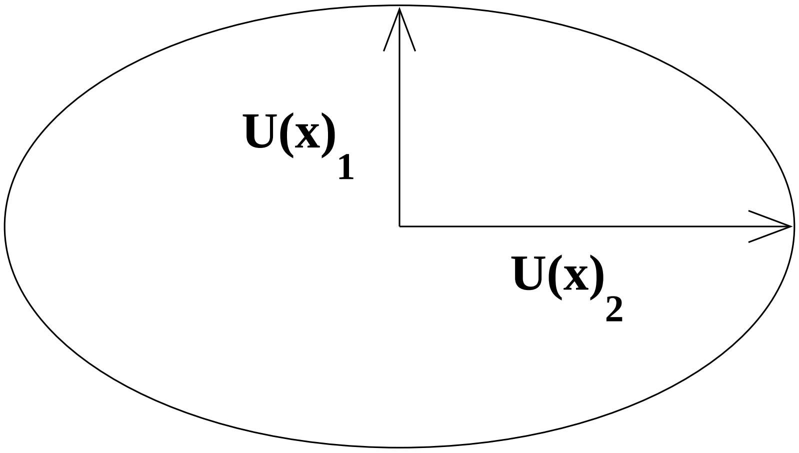

Figure 3.6: This figure shows the ellipse defined by equation (3.12). The length of $U(\mathbf{x})_1$ is r_s and the length of $U(\mathbf{x})_2$ is r_t. The numbers r_s and r_t are chosen such that every point in the ellipse is within a given tolerance of $F^{-1}(\mathbf{0})$.

We now combine the two ellipses into an approximation of a portion of $F^{-1}(\mathbf{0})$. The outline of this process is shown in Figure 3.7. The actual implementation of this procedure is somewhat messy, and the details will not be given here. However, the basic point of the implementation is that two ellipses are combined by taking their borders and marking the sections which are *overlapped* by other ellipses, and those which are not overlapped. The sections of the border which are overlapped by another ellipse are ignored, while those which are not overlapped are called *active.*

Definition 3.4.1. *Given two points $\mathbf{x}_0$ and $\mathbf{x}_1$, a point $\mathbf{x}$ on ellipse $E_{\mathbf{x}_0}$ overlaps $E_{\mathbf{x}_1}$ if there exists a point $\mathbf{y} \in E_{\mathbf{x}_1}$ such that the orthogonal projection of $\mathbf{x}$ and $\mathbf{y}$ onto $F^{-1}(\mathbf{0})$ are the same.*

The algorithm can now be described as follows. Assume that we have a list of the active borders of all of the current ellipses. Choose a point on an active border, and project that point onto $F^{-1}(\mathbf{0})$ to create a new ellipse. Intersect the new ellipse with all other ellipses. Remove from the list of active borders any section of an active border which overlaps the new ellipse. Add to the list of active borders any section of the border of the new ellipse which remains active after all of the intersections have occurred. Repeat this process until no active borders remain (see Figure 3.8 for an example).

There are several advantages to this algorithm. First, step-size control is an integral part of the algorithm, due to the computation of r_s and r_t which define the sizes of the ellipses (see Figure 3.6). Second, the algorithm does not have to worry about computing a continuous set of bases, since each basis is only used to

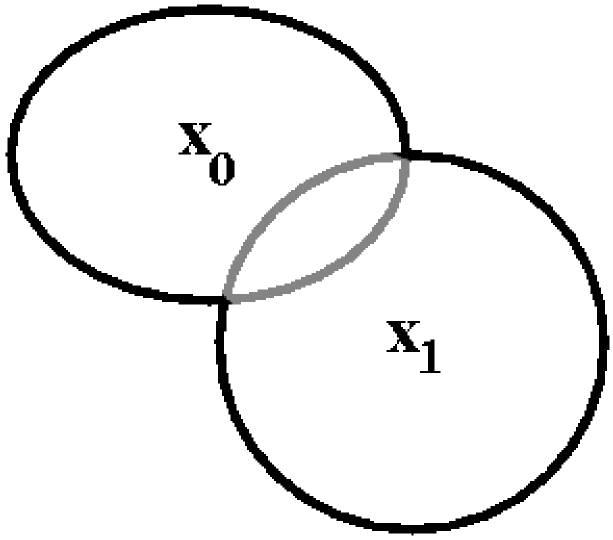

Figure 3.7: This figure shows the first two ellipses after they have been combined. The sections of their boundaries shown in black lines are active while those which are shown in grey overlap and have been discarded.

compute the specific ellipse. Third, because it keeps track of the active borders of the bifurcation manifold it can efficiently compute manifolds with more complex non-simply connected topology (e.g. torii) without recomputing portions of the manifold.

The disadvantage of Henderson's algorithm is that it is somewhat complex to implement. A naive implementation which intersects a new ellipse with all previous ellipses, would be computationally expensive. Accordingly, the algorithm needs to be implemented so that some set of disks can be quickly discarded as possible candidates for intersection. For example, a more specialized data structure for spatial decompositions, such as a quadtree [47], could be utilized to store the ellipses. Unfortunately, the available implementations of which we are currently aware do not utilize any such special data structures.

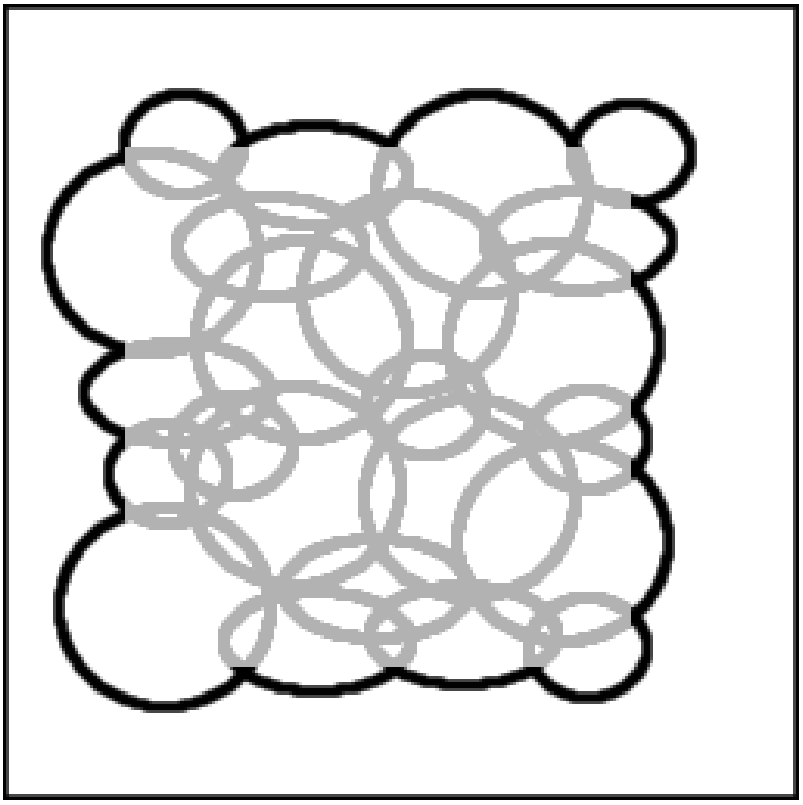

Figure 3.8: A set of ellipses at some point of the Henderson Overlapping Disk algorithm. Those parts of the ellipses shown in grey overlap and have been discarded. The set of active borders are shown in black.

Chapter 4

Overview of Efficient Numerical Implementations and Object Oriented Methods

4.1 Introduction

The previous chapters have described the mathematical theory of parameter continuation algorithms (both 1D and multi-D). This chapter will explore how one can write efficient computer implementations of such algorithms. Hardware design will have a very significant impact on efficient implementations of continuation computations. Accordingly, we briefly review some basic concepts of high performance computing architectures. Section 4.2 will give an overview of some basic aspects of computer hardware and how they affect numerical algorithms. We will touch briefly on general numerical software as well as the concept of non-uniform memory access (NUMA) computers because in Chapters 5 and 7 there will be a need for various concepts in distributed computing. In Section 4.3 we begin with a definition of basic terms used in object oriented design. We then explore more advanced object oriented topics which will be used extensively in Chapter 5.

4.2 Computer Architecture

We begin with a brief discussion of computer hardware and how its design affects numerical algorithms. This subject has received wide attention in the literature (see, for example [24, 41, 42, 49, 31, 21]) so we will focus our attention on those parts which strongly affect the performance of numerical continuation algorithms.

4.2.1 Memory Hierarchy

The single most important efficiency issue in modern computer systems is the use of the *memory hierarchy*. The modern CPU has various levels of memory and

Type	Latencies (in clock cycles)	Size
Registers	1 cycle	256 bytes
Level 1 Cache	3-4 cycles	16K bytes
Level 2 Cache	6-12 cycles	4M bytes
Main memory (DRAM)	60 cycles	up to 4G bytes
Local Disk	O(1,000,000) cycles	Terabytes possible

Table 4.1: This figure shows the latencies for accessing various parts of the UltraSPARC memory hierarchy as well as their storage capacities. Latency is measured in CPU clock cycles (i.e. the rate at which commands are issued to the CPU). For example, six to twelve commands could be executed in the amount of time it takes the CPU to access the level 2 cache. So, in the absence of optimizations, a command whose operands are in level 2 cache, as opposed to registers, takes six to twelve times longer to execute.

storage media associated with it, and the size and performance of these levels varies by several orders of magnitude. Let us take as an illustrative example the UltraSPARC by Sun Microsystems. The memory hierarchy for this processor is shown in Table 4.1 [8].

Table 4.1 shows the *latencies* in communication between the CPU and the various levels of the memory hierarchy. Latency is the amount of time it takes for a single bit to travel from where it is stored to the CPU. It is important to distinguish this from *bandwidth*, which is the total number of bits that arrive within a given time. Bandwidth is not shown in Table 4.1 because, for most of the levels shown in that figure, latency is the limiting factor. For completeness we note two exceptions to this rule. First, main memory bandwidth becomes important for *symmetric multi-processors*, which we will discuss further later in this chapter. Second, disk bandwidth is important for some problem classes, but not those which arise here. The algorithms we discuss here are limited more by the amount of computation they require rather than by their input-output (I/O) requirements.

As can be seen in Table 4.1, the latencies between the various layers range from one clock cycle to millions of clock cycles. To emphasize the implication of this difference in latencies, consider the case where the CPU is performing an addition. If the operands are in registers this operation takes only one clock cycle. On the other hand, if the operands are on the disk, this operation will take millions of times longer. So, to be efficient, it is very important that a program does not frequently retrieve data from the disk. In fact, this is exactly the reason that the memory hierarchy exists, to minimize the number of access to the slow levels of the memory hierarchy.

Optimization of the memory hierarchy is based on the idea of *locality of ref-*

erence. A given program is assumed not to access memory in a random manner, but to work on a small area for some time before moving on. This assumption is used to amortize the cost of an access to a higher level of the memory hierarchy by retrieving not just the requested memory location, but nearby ones as well. The number of memory locations that are retrieved at once is sometimes called the *length of the cache line*, and is another important factor in efficiency. Bandwidth is not the limiting factor. The incremental cost of retrieving several memory locations, rather than one, is small. The nearby memory locations can then be stored, or cached, in lower levels of the memory hierarchy with the assumption that they will soon be requested by the CPU as well. There are a myriad of details one needs to address in memory hierarchy design, such as how one decides what parts of a cache are discarded if it is full and new data needs to be entered. Fortunately, there is a well developed theory of memory hierarchy design which, in practice, works quite well at solving these problems. On the other hand, memory hierarchy designs make assumptions about programs, such as locality of reference, and it is important that numerical codes obey these assumptions if they are to use the hardware efficiently.

For completeness we now mention one other important hardware structure which affects performance, the concept of *pipelining.* There are several stages a single command goes through during its execution, for example fetching the instruction from memory, decoding the instruction, address calculation, etc. Likewise, modern CPUs are divided into sections, each of which performs some part of the execution of a command. The important thing to note is that, in general, one command only uses a single section at a time. A naive approach to command execution is to start one command and wait until it is finished before starting the next command. Unfortunately, using this approach leaves all of the sections of the CPU idle, except for the single one in which the command currently resides. The solution is to pipeline the commands, i.e. begin the new command as soon as the previous command has finished with the first section of the CPU (see Figure 4.1).

Unfortunately, pipelining is somewhat difficult to implement. Consider the following scenario. We have two commands, A and B. Command B uses the result of command A. If we pipeline the two commands it may very well happen that the result of command A will be required by command B before command A has finished executing (see Figure 4.2). This situation is called a pipeline *hazard.* Modern processors, with their many stages, can perform poorly if commands are not carefully ordered to avoid hazards.

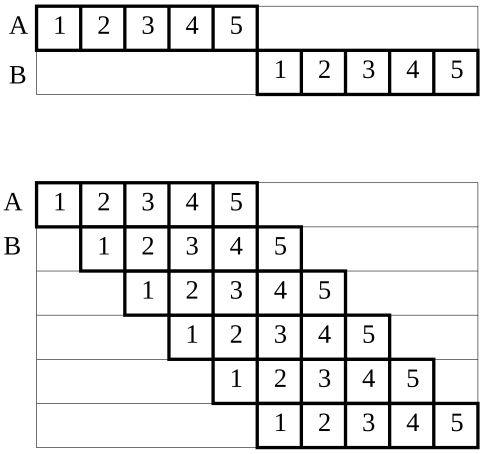

Figure 4.1: This figure shows a schematic of pipelining on a computer with a five stage pipeline. The top diagram shows an example of naive command execution. Two commands, A and B, are divided into sections. Command B does not start executing until command A is totally finished. Each part of the CPU is active only one-fifth of the time and idle the rest. The bottom figure shows pipelining. As soon as command A is finished with the first section of the CPU command B starts using it. Six commands are executed in the time that it took the naive approach to do two.

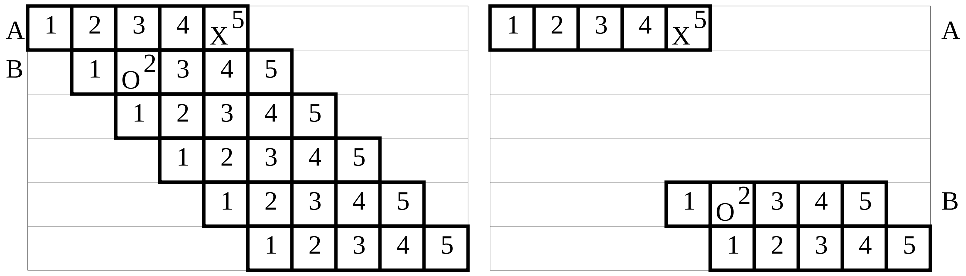

Figure 4.2: An example of a pipeline hazard. The section of command B, marked with the O, cannot execute because it requires the result of command A, which will not be available until the point in time marked by the X. The pipeline has to be stalled until the result is available, losing several clock cycles of computation.

4.2.2 General Numerical Software

It would now seem that we have a serious problem. To write efficient numerical software we have to have detailed knowledge of the hardware on which we wish to run. Not only that, but each hardware manufacturer makes different assumptions in their design. We must have custom versions of our software for each machine we want to run on, since satisfying the requirements of one machine may, in fact, break the requirements of another.

Fortunately, these problems can be alleviated by compilers and software libraries. Modern compilers are quite good at solving the pipeline hazard problem on all but the most pathological codes by command reordering and other similar methods. On the other hand, the efficient use of the memory hierarchy is more difficult. The order of memory access in a program can be very complex and can depend subtly on its global structure, e.g. whether matrices are stored in column major versus row major form. Software libraries, such as the Basic Linear Algebra Subroutines (BLAS), solve this problem. BLAS is a collection of routines which perform standard linear operations, such as matrix addition and multiplication. Computer vendors provide hand-tuned implementations of these libraries for their hardware.

The idea is that if one utilizes the BLAS routines then at least those functions in your code are efficient. If calls to these routines make up a significant portion of the total run time of your program, which is very common in practice, this portion of your program will use the memory hierarchy efficiently. There are even libraries written on top of BLAS, such as LAPACK [5], PETSc [9], and SCALAPACK [10], for performing more complex functions, such as matrix factorizations. Often, such procedures make up 90% or more of the total run time of a program, and using standard optimized libraries for these computations can give us very efficient utilization of hardware.

4.2.3 Non-Uniform Memory Access Computers

We will now make a brief divergence to look at some classes of parallel computers. There are many details in the design of parallel computers which do not directly concern us, but we note that two major classes of parallel computers today are symmetric multi-processors (SMP) and clusters. In SMPs every processor has access to the same memory pool. Communication between processors is normally fast (low latency and high bandwidth). Clusters, on the other hand, are groups of processors, each with their own memory. Clusters normally communicate with each other using some type of networking media, such as Ethernet, ATM, Myrinet, etc., which have a wide range of performance, but are slower than direct memory access.

There are two programming models associated with these types of computers as well. In the *shared memory* model, which is associated with SMPs, a program assumes that it can communicate with another processor by directly writing into its memory. In the *message passing* model, which is associated with clusters, a program assumes that it can only communicate with another processor by creating a message and transferring that message to some agent which transports it to the other processor. To be precise, one should note that the association between the hardware and the programming model is quite flexible. For example, one can certainly program a SMP using the message passing model.

A third class of hardware is the non-uniform memory access computer (NUMA). In the standard nomenclature, a computer is called a NUMA if every processor has direct access, through basic loads and stores, to the memory of every other processor, but different pairs of processors can have different communication characteristics. For example, one could have three processors, A, B, and C, with the communication between A and B being much faster than that between A and C. Canonically, there is a hierarchy of processors, in which small groups of processors have fast communication within their group, but slower communication to processors in other groups. For our purposes, we will expand the definition slightly to allow a computer where a given processor does not have direct load-store access to the memory of all other processors. Some processors are only accessible using message passing over a network. An example would be a group of SMPs together on a network.

4.3 Object Oriented Methods

We begin our discussion of object oriented methods by defining a few basic terms. *Objects* are collections of data and functions which are intimately tied together. The functions which comprise objects are often called *methods*. The methods are incorporated with the data and are not separate entities, as one finds in some other types of programming models. *Interfaces* are the external structure of objects. An interface describes how an object appears to other parts of the program and how it may be used. The distinction between objects and interfaces is important. Often it occurs that two objects implement the same interface but have very different internal structure. To other parts of the program they appear identical and may be used interchangeably, but, internally, they each utilize different algorithms, data structures, etc. Another case is also possible, where one object implements several different interfaces simultaneously. An object which implements more then one interface may be used in any place where any one of its interfaces are appropriate. Finally, given an interface Snarf, we say an object which implements the Snarf interface is a Snarf object[1].

[1]We have abused slightly the common usage of the term interface here. A distinction is often made between two types of interfaces, those that can be used directly to create objects and those which cannot. For example, in C++, interfaces which can be used to create objects are called *Classes*, and those that cannot are called *Abstract Base Classes*. In Java they are called *Classes* and *Interfaces* respectively. We choose to use the word interface because these differences are not germane to our discussion, and we feel that the intuition provided by the

4.3.1 Object Oriented Methods and Numerical Efficiency

Object oriented programming languages are a wide and diverse topic and we will not present here an exhaustive discussion of the subject. On the other hand, a few ideas from object oriented languages, such as *dynamic method lookup* and *runtime typing*, are key to the development of MCCC discussed in Chapter 5 and we will describe them here.

Normally, the interfaces which exist in a program are not independent of each other but rather exist in a hierarchy (e.g. see Figure 4.3). Higher level interfaces can be created which are supersets of the lower level interfaces. This process is called *inheritance*. Given two interfaces, A and B, if B is a superset of A we say that B is a *derived interface* of A and that A is a *base interface* of B.

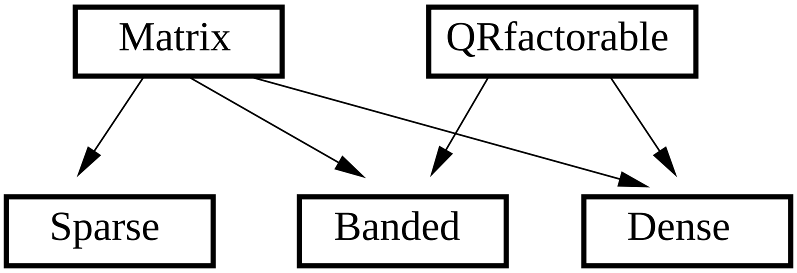

Figure 4.3: This figure shows an example of an interface hierarchy for matrix objects. The two top level interfaces are Matrix and QRfactorable. The matrix interface provides methods for basic linear algebra (such as matrix addition, etc). The QRfactorable interface provides a method for constructing the QR factorization. The next level shows interfaces for specific kinds of matrices. The Sparse interface is a superset of the Matrix interface. The Banded and Dense are supersets of both Matrix and QRfactorable. Any section of the code which expects a Matrix object can use any of the three second level objects, while any section which needs a QRfactorable object can only use a Banded or Dense object.

It is easy to see that, to maintain generality, it is desirable to write code which uses the lowest level interface possible. For example, if you write a subroutine which explicitly uses a Banded interface, it will not work for an object which implements the Sparse interface. On the other hand, if the subroutine could have been written using the Matrix interface instead, it would work for objects which implement the Sparse, Dense, or Banded derived interfaces as well (an object which implements the Sparse interface must, by definition, implement the Matrix interface).

term is the correct one for our purposes.

A problem now arises which is best explained with an example. Suppose that the Matrix interface has a method `compute determinant` and that we have two objects, one which implements the Sparse interface and one which implements the Dense interface. The actual implementation of this method could be very different between the two objects, e.g. the Sparse object could implement this method in a way which uses the sparsity of the matrix, and which can be much faster. Further, suppose that we have written a function which takes as input an object which implements the Matrix interface and then calls the `compute determinant` function. This function will work on both the Sparse and Dense objects defined above, but it does not know that they have special implementations of this function. It only knows about Matrix objects. Dynamic method lookup is used to solve this problem. In dynamic method lookup the system itself records which methods are associated with an object, regardless of which interface the current function thinks it implements. Accordingly, in our example above, the function can call what it thinks is a generic `compute determinant`, but what actually gets called is the special `compute determinant` for the sparse matrix, and, most importantly, the function does not have to be cognizant of the difference.

Dynamic method lookup works very well for simple interface hierarchies, but for more complex hierarchies different methods are needed. For example, consider the interface hierarchy in Figure 4.3. Suppose we have a function which has as an input a Matrix object, and as part of its work it needs to compute the QR factorization of this matrix. There are two choices. It can do a brute force calculation of the QR factorization itself, or it can call the QRfactorable method of the QR interface. The QRfactorable method would be the better choice, since it can use the specific structure of the matrix to accelerate the calculation, but not all Matrix Objects implement the QR interface. It would be useful to be able to query an object and find out if it implements just the Matrix interface or both the Matrix and QR interfaces. Runtime typing is used to solve this problem by allowing a function to query an object as to which interfaces it implements.

It might appear that runtime typing is strictly better than dynamic method lookup since dynamic method lookup can only handle simple class hierarchies, while runtime typing can handle all cases. In practice, this is not the case. Dynamic method lookup requires no code changes in order to handle different interfaces, accordingly, when a new interface is added the code does not need to be modified. Runtime typing depends on knowledge of the given interface, so when new interfaces are added the code needs to be modified everywhere the new interfaces are to be used. It is important to strike a careful balance between dynamic method lookup and runtime typing when one defines an interface hierarchy, and to attempt to use the former as much as possible.

Chapter 5

Multi-parameter Computation and Continuation Code (MCCC)

5.1 Introduction

In Chapter 4 we studied some of the basic theory behind the efficient implementation of numerical algorithms. In this chapter we will utilize this theory by constructing a new framework for the implementation of parameter continuation methods called the Multi-parameter Continuation and Computation Code (MCCC). The unifying themes in MCCC are to use an object oriented (OO) approach to encapsulate the interrelationship between the various components of a numerical continuation algorithm, and to make the implementation of such algorithms easy and efficient. We will begin, in Section 5.2, with a description of a novel OOdecomposition of parameter continuation algorithms. Section 5.3 will describe a three tiered parallel decomposition of parameter algorithms, and describe how this decomposition may be used on a wide range of parallel architectures, including NUMAs. Section 5.4 will describe the specific OOstructure utilized by MCCC, and Section 5.5 will describe some specific optimizations that may be used within MCCC.

5.2 Parameter Continuation Efficiency

In Chapter 2 and Chapter 3 we described the details of parameter continuation algorithms. We will now step back and look more closely at their overall structure. We will describe how parameter continuation algorithms (both 1D and multi-D) can be broken up into pieces and how those pieces interrelate. MCCC breaks a numerical continuation algorithm into the three main pieces that are shown schematically in Figure 5.1.

Often the problem of interest is not explicitly in the form of equation (2.1), e.g. it could be in the form of an ODE BVP. The first step, then, is to recast

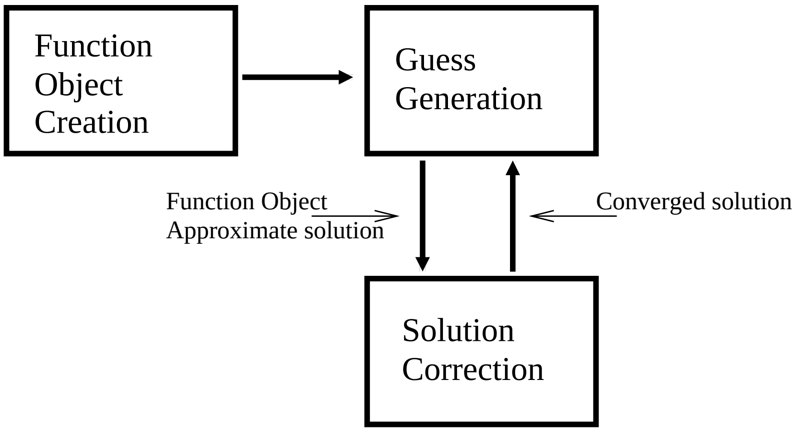

Figure 5.1: This figure shows a conceptual perspective class diagram for P-C continuation algorithms. To start the process a given problem must be placed into the form of equation (2.1). This algebraic non-linear function, and some set of solutions to the problem, are then passed to a guess generation procedure which constructs a new approximate solution to the problem. This approximation, and the original function, are passed to a procedure which projects the guess onto the bifurcation manifold, and returns the converged solution. The process can then be repeated by utilizing the new solution to generate more approximate solutions.

the problem in the form of equation (2.1). For example, in the above case a discretization process, such as finite differences or orthogonal collocation, is applied to the problem, changing it into the form of equation (2.1). The second step is to compute an approximation to a solution of equation (2.1) using some set of previous solutions. How exactly this is done is the heart of the continuation procedure. There are a wide range of possible algorithms, as seen in the previous chapters. The final step is to take the approximation and correct it to within a given tolerance of the bifurcation manifold.

Let us look at the various pieces of a continuation algorithm and consider their computational cost, in very general terms, for both P-C and simplicial algorithms. For both simplicial and P-C methods the first step is equivalent, though there exist cases where it can be quite expensive, for example, if one needs to generate meshes for very complex geometries. Fortunately, the recasting is generally not performed often, though, we may not assume that it only happens once. There exist many cases where discretizations need to be changed at different parameter values, and we want to allow for this possibility. The second step normally involves the computation of a tangent space, or an approximation thereof, for predictor corrector methods and the solution of the labeling matrix (see Theorem 2.3.1) for simplicial methods. Again, the specific problem or implementation affects how the approximation is computed. The third step is trivial for simplicial methods and canonically involves some iterative procedure for P-C methods. We note here that specific implementations of these algorithms and specific problems may not exhibit these properties, but that all of the algorithms with which we are familiar do.

Accordingly, if we assess the relative computational merits of the two methods, it seems that they are roughly equal. Generally, the second step for the simplicial methods involves solving linear equations involving the $n \times n$ labeling matrix. The second step for the P-C method normally consists of solving linear equations involving the $n \times n$ Jacobian of equation (2.1), as does the third step. In each case the main computational expense is in solving an $n \times n$ system of equations.

On the other hand, if we consider the case where the problem is not inherently of the form of equation (2.1), we may come to a very different conclusion. For example, if the problem we wish to solve is a discretization of some continuous problem, the set of equations we wish to solve often has some special structure, such as having a Jacobian which is sparse. In the case of simplicial methods, the labeling matrix (see Theorem 2.3.1) is dense even if the equations are derived from a discretization process. The main point to observe is that the P-C algorithms may easily be implemented to take advantage of the structure of the equations, while the simplicial methods do not facilitate such a specialization. Hence, for large, structured problems P-C methods are generally more efficient because they can more easily use the structure of the problem to accelerate calculations. For this reason we have concentrated our attention on P-C algorithms for MCCC.

5.3 Parallelism

In this section we will discuss the implications of hardware and network design on efficient interactive continuation computations, and how they may be used in a structure such as MCCC to accelerate calculations. Once one leaves the realm of using only a single processor for calculation and visualization, the exact topology and memory structure of a *compute engine* must affect the design of the computation, if one is to achieve a high level of efficiency.

5.3.1 Multilevel Parallelism

For the algorithms and methods discussed here there is a natural categorization of the various areas where parallelism can be utilized.

- Parallelism in linear algebra
- Parallelism in adjacent parameter values

- Parallelism in non-adjacent parameter value regions

We will now examine each of these levels in turn.

In many, if not all, of the algorithms described here there are various linear algebra operations that are carried out. This linear algebra is a place where parallelism is very commonly used. Normally such calculations require low latency and high bandwidth, and their optimization can be quite difficult. Fortunately, we may take the same approach as used to optimize the memory hierarchy and utilize the large body of work already done on this subject. There exist software packages, such as SCALAPACK [10], and PETSc [9], which are highly optimized and solve many common problems in linear algebra. In practice, these packages may be used separately in each of the blocks shown in Figure 5.1 to parallelize each of them easily. Modern compilers can even be used to affect some parallelization automatically, though currently these compilers are often not as effective as one would hope. Automatic parallelization is an active area of research in compiler design. An example of compiler based parallelization can be found in MCCC. One of the solution correction procedures implemented in MCCC is a Powell hybrid method from MINPACK [37]. The Sun Microsystem high performance computing compilers have been used to create a multi-threaded version of this routine for use in MCCC. The multi-threaded version of this routine is also used as part of the example in Section 8.3.

For algorithms with $m > 1$, one can often compute on several parameter values concurrently. Such algorithms can often be implemented by having the guess generation procedure compute several (approximate) new points at one time. These new points can then be corrected in parallel. The advantage of this level of parallelism is that even though it may require high bandwidth (if n is very large) it does not require very low latency. Each of the individual calculations is of a long duration, and any latency can be amortized over the length of the calculation.

Finally, for $m \geq 1$ several regions of a bifurcation manifold may usefully be computed in parallel. This is by far the easiest of the levels of parallelism to implement. It is what is sometimes known as "embarrassingly parallel", although it is still important to exploit its structure whenever possible. The calculations are totally independent. In fact, each of the calculations may be running totally different algorithms and computing with different values of m.

A schematic of all of these levels of parallelism may be seen in Figure 5.2. One advantage of such multilevel parallelism is that it works well on NUMAs. It is notoriously hard to develop efficient code for NUMAs, because of their various communication speeds. In contrast, continuation algorithms with a decomposition as shown in Figure 5.1 fit very well on this type of machine, because the lowest level parallelism requires fast communication (which NUMAs often have for small groups of processors), but the higher levels have lower requirements in communication speed. Another advantage of this multilevel parallelism is that large numbers of processors may be easily and efficiently utilized. For example, using four-way parallelism at each level utilizes a total of $4 \times 4 \times 4 = 64$ processors. MCCC has been designed to implement the first two levels, while the third level is implemented by the software VBM discussed in Chapter 7.

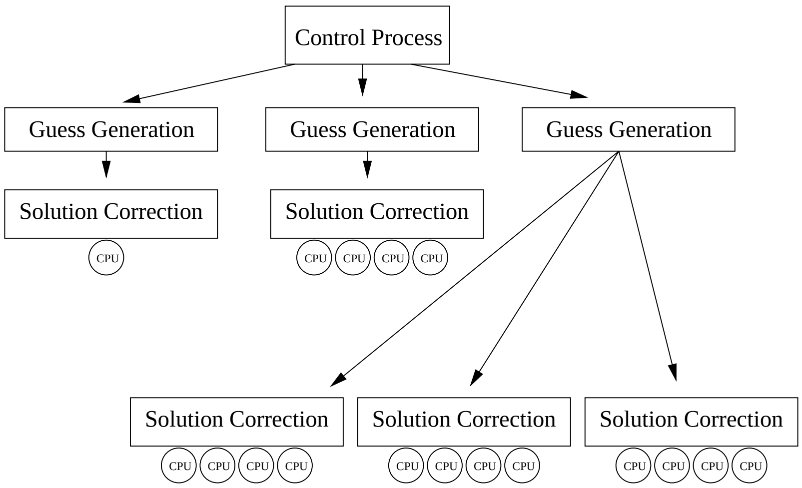

Figure 5.2: This figure is a schematic of multi-level parallelism. The top box is a control process which is controlling three continuation algorithms. The left algorithm is running on a single processor. The middle algorithm is only performing one correction at a time, but the linear algebra inside the correction algorithm is running on several processors. The right algorithm uses several correction procedures at once, each of which is using several processors to perform its linear algebra.

5.3.2 Separation of visualization and computation

We finish this section with a brief discussion of the *control process* from Figure 5.2. While not directly a part of MCCC this module is important and will be the subject of Chapters 6 and 7. The problem is that the various computational engines can generate vast amounts of data and the ability to quickly understand this data in an interactive way is one of our key goals.

In our implementation of the structure shown in Figure 5.2 the control process does not only just control the continuation algorithms. It also provides visualization services. The idea is that we want the computations to be interactive. The computations provide data which is visualized by the operator. The operator uses the insight gained by the visualization to steer the computation, providing more data, etc. Accordingly, MCCC has been designed to run either stand-alone or as a part of a larger system.

There are a few issues that one must address in such a structure. First, since the control process also provides visualization services, it must run on appropriate hardware (with accelerated graphics, specialized input devices, etc.). The requirements of specialized visualization hardware and computational hardware are often contradictory. This problem can be solved by having separate visualization and computation hardware and having them communicate over a network.

Unfortunately, with a configuration as shown in Figure 5.2, the amount of data that must be transmitted to the control process may be very large, especially when several compute engines are running in parallel. It would therefore be desirable to appropriately filter data via preprocessing before it is sent to the control process. An example of this procedure will be discussed in Chapter 8.

5.4 Object Oriented Structure of MCCC

MCCC is designed as a set of interfaces which codify the conceptual class diagram introduced in Figure 5.1. The objects and interfaces in MCCC specify the relationship between the blocks and their modes of communication. The class hierarchy of MCCC is shown in Table 5.1 and Table 5.2 (which may be found at the end of this section).

The lowest level interface is the Function interface. An object which implements this interface comprises all that is required of a general mapping of $\mathbb{R}^{n+m} \to \mathbb{R}^n$. This interface provides three methods.

- One method returns the dimension of the domain.
- One method returns the dimension of the range.
- One method takes as an argument a vector of dimension $n+m$ and returns one of dimension n.

For generality, this set of methods is all that can be assumed about a Function object. Unfortunately, such a structure, while very general, is very inefficient. Function objects often have structure, such as sparsity of the Jacobian, that can be used to greatly accelerate computations. This problem will be addressed further in Section 5.5.

The second major interface is the SolutionCorrection interface. Normally, when one of these objects is created it contains a Function object. This interface provides two methods. One allows the user to set a tolerance for the calculation, and the other allows the user to find a zero of the Function object (to within the given tolerance) given an initial guess. SolutionCorrection objects can be implemented using a wide range of algorithms, such as Newton's Method or Conjugate Gradients. For large systems, these algorithms are often the main computational expense of the continuation algorithm.

The final major class of interfaces are the GuessGeneration interfaces. These interfaces are the highest level interfaces in the MCCC library, and each GuessGeneration object encapsulates some P-C algorithm. When a GuessGeneration object is created it is given a Function object whose solution manifold the GuessGeneration object will attempt to compute. SolutionCorrection objects may also be given to the GuessGeneration object when it is created, or the GuessGeneration object may utilize some default SolutionCorrection object. The idea is to allow the user to give a GuessGeneration object a special Solution Correction object which works well for the given Function object. The methods of these objects may vary from one P-C algorithm to another, though standard interfaces are provided for several classes of GuessGeneration objects, such as those which perform continuation based on active edges (e.g. Rheinboldt's Moving Frame in Section 3.4.1 and most one parameter algorithms in Section 2.3) and those which perform continuation based upon bounding boxes (e.g. Henderson's Overlapping Disk method in Section 3.4.2).

The above interfaces are intentionally simplistic. They are the basic building blocks of MCCC and are designed to encompass as many different algorithms as possible. They make only the most basic assumptions about the algorithms which they encapsulate, enabling the widest range of algorithms to be used. The advantage that they provide is code modularity. For example, to change the GuessGeneration object or SolutionCorrection object in an application written in MCCC requires the change of only one line of code. The ease with which different algorithms may be tried encourages experimentation. Unfortunately, as was seen in Chapter 4, it is the structure of the specific calculations that must be used to achieve efficiency.

We close this section with a brief discussion of two other important interfaces in MCCC, namely Matrix and Discretization. The Matrix interface provides a wrapper over different possible implementations of matrix storage, such as banded, sparse, or dense. The Matrix interface demands that any underlying storage scheme provide services such as scalar and matrix multiplication, and the assumption is that these services are optimized for the storage scheme. More specialized functions, such as QR factorization, are separate interfaces, since not all matrix storage types may have good algorithms for all functions. MCCC currently only implements dense matrices because good Java libraries for sparse matrices do not yet exist. Fortunately, once Java libraries for sparse matrices become available they may trivially be utilized in MCCC.

The Discretization interface is used to transform problems which are not inherently of form of equation (3.1) into that form. All Discretization objects implement the Function interface, so that they appear to other parts of the program as normal Function objects. The main difference between Discretization objects and the canonical Function object is that Discretization objects are normally structured in some way. As will be seen in the next section, MCCC has been designed to access this structure in a modular way.

- class java.lang.Object
    - class CH.epfl.lcvm.mccc.ContinuationNatural (implements CH.epfl.lcvm.mccc.GuessGeneration1D)
    - class CH.epfl.lcvm.mccc.ContinuationPseudoArclength (implements CH.epfl.lcvm.mccc.GuessGeneration1D)
    - class CH.epfl.lcvm.mccc.ContinuationRheinbolt2D
    - class CH.epfl.lcvm.mccc.FDDiscretizationContact (implements CH.epfl.lcvm.mccc.Function)
    - interface CH.epfl.lcvm.mccc.Function
    - class CH.epfl.lcvm.mccc.FunctionBase (implements CH.epfl.lcvm.mccc.Function)
        - class CH.epfl.lcvm.mccc.FunctionConcatenation
        - class CH.epfl.lcvm.mccc.FunctionFixed
        - class CH.epfl.lcvm.mccc.FunctionSphere (implements CH.epfl.lcvm.mccc.HasJacobian)
    - interface CH.epfl.lcvm.mccc.GuessGeneration1D
    - interface CH.epfl.lcvm.mccc.GuessGeneration2D
    - interface CH.epfl.lcvm.mccc.HasJacobian
    - interface CH.epfl.lcvm.mccc.IVPSolver
    - class CH.epfl.lcvm.mccc.IVPSolverBase (implements CH.epfl.lcvm.mccc.IVPSolver)
        - class CH.epfl.lcvm.mccc.IVPEuler
        - class CH.epfl.lcvm.mccc.IVPRungeKutta2
    - interface CH.epfl.lcvm.mccc.Matrix
    - class CH.epfl.lcvm.mccc.MatrixArray (implements CH.epfl.lcvm.mccc.Matrix, java.lang.Cloneable)
    - class CH.epfl.lcvm.mccc.NewtonRaphson (implements CH.epfl.lcvm.mccc.SolutionCorrection)
    - class CH.epfl.lcvm.mccc.NumericalJacobian (implements CH.epfl.lcvm.mccc.HasJacobian)
    - interface CH.epfl.lcvm.mccc.ODEFunction

Table 5.1: This figure shows the first part of the MCCC class hierarchy. The main interfaces are Function, SolutionCorrection, and GuessGeneration. Two other interfaces, which are not core to MCCC but which are used extensively are Matrix and Discretization. Matrix, shown here, provides a general interface to different matrix storage schemes.

- class java.lang.Object
  - class CH.epfl.lcvm.mccc.ODEFunctionBase (implements CH.epfl.lcvm.mccc.ODEFunction)
    - class CH.epfl.lcvm.mccc.ODEFunctionCircle (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
    - class CH.epfl.lcvm.mccc.ODEFunctionDNA1ParNative (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
    - class CH.epfl.lcvm.mccc.ODEFunctionDNA6ParNative (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
    - class CH.epfl.lcvm.mccc.ODEFunctionPlanarStrut (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
    - class CH.epfl.lcvm.mccc.ODEFunctionRod (implements CH.epfl.lcvm.mccc.ODEFunctionWithBC)
  - interface CH.epfl.lcvm.mccc.ODEFunctionWithBC (extends CH.epfl.lcvm.mccc.ODEFunction)
  - class CH.epfl.lcvm.mccc.OneDDiscretization (implements CH.epfl.lcvm.mccc.Function)
    - class CH.epfl.lcvm.mccc.CollocationDiscretization
    - class CH.epfl.lcvm.mccc.FDDiscretization
  - interface CH.epfl.lcvm.mccc.OneDMesh
  - class CH.epfl.lcvm.mccc.OneDMeshUniform (implements CH.epfl.lcvm.mccc.OneDMesh)
  - class CH.epfl.lcvm.mccc.PowellHybridNative (implements CH.epfl.lcvm.mccc.SolutionCorrection)
  - interface CH.epfl.lcvm.mccc.QRfactorable
  - interface CH.epfl.lcvm.mccc.SolutionCorrection
  - class java.lang.Throwable (implements java.io.Serializable)
    - class java.lang.Exception
      - class CH.epfl.lcvm.mccc.MCCCException
        - class CH.epfl.lcvm.mccc.NoConvergenceException
        - class CH.epfl.lcvm.mccc.SolverException
  - class demo

Table 5.2: This figure shows the second part of the MCCC class hierarchy. The Discretization interface is used to transform a variety of problems into the form of equation (3.1).

5.5 Efficient Implementation

We will now look at the specifics of how one implements Parameter Continuation algorithms efficiently using the above OOstructure while still maintaining modularity. The key issue is that, of all the objects in MCCC, only a Function object can know the details of its own implementation. The SolutionCorrection and GuessGeneration objects should be able to perform their functions on any Function object, and they should not need to be concerned with any special structure the Function object may have.

The solution is to add additional interfaces that a Function object may implement. For example, consider an interface such as SolutionCorrection. The SolutionCorrection interface indicates that the Function object has some efficient way of finding solutions to equation (2.1) given an initial guess. For example, it may use Newton's Method on a sparse Jacobian because the Function object was created using a finite difference approximation of an ODE. The point is that the Function object knows what its structure is, and that the above algorithm is efficient for this structure. The other objects only know that it is a Function object which implements SolutionCorrection. In such a case, a SolutionCorrection object which has such a Function object merely calls the appropriate method of the Function object, and then returns. This procedure obviously requires some way for the SolutionCorrection object to query the Function object, and performing this query is where MCCC uses runtime typing. As another example, consider a Function object which has some way to compute a basis for its tangent space. This procedure is often a part of the GuessGeneration step, and a GuessGeneration object could either compute the basis itself (which can be slow for large systems) or query the Function object to see if it has a better method.

In general, each section of the program which uses a Function object will query the Function object to see if it offers any services that it finds useful. If it finds any needed services it must then decide whether to use its own algorithms or those provided by the Function object. Querying the Function object maintains modularity in that any Function object may be used in any section of the code, but increases efficiency by using the structure of the Function object where applicable. The one failing of this method is that there has to be a global list of interfaces which Function objects may implement. When a new interface is added the old sections of code will not be able to use the new interface unless they are each modified as well.

5.5.1 Specific Optimizations of the MCCC Implementation

The previous sections of this chapter have discussed MCCC in a theoretical, language independent fashion. In this section we will describe some of the specifics of our implementation and how they affect optimization.

We have implemented MCCC in Java. There are several reasons for this choice of language. First, dynamic method lookup is the default in Java. In languages such as C++, dynamic method lookup must be explicitly requested when an interface is created (using the "virtual" keyword). Second, runtime typing is implemented in Java in a consistent and robust way. Runtime typing is a relatively new construct in OOlanguages, and while it is a part of the C++ language definition, not many compilers have good implementations of it. Third, Java uses garbage collection in lieu of explicit memory management. While there is a performance cost associated with this feature, we find that the robustness that it affords the program is well worth the added computational cost (especially with the use of native methods). Finally, Java is a strongly typed language (as opposed to languages such as Python, Perl, and Tcl). This allows many classes of errors to be caught at compile time, again adding to the robustness of the code.

One aspect of Java which is normally seen as inadequate for numerical computations is the fact that it is currently slower than languages such as C and Fortran. If we examine Table 5.3 we notice that unoptimized Java is approximately thirty times slower than C or Fortran. Utilizing a "just in time" compiler improves the performance to approximately three and a half times slower than C or Fortran. On the other hand, Java's native language interface is only approximately five percent slower than a pure natively compiled language.

Machine	**Language**	**Runtime**
Sun	Direct call of Lapack SGEQRF	0.155 seconds
Sun	Visual Numerics Java library (with JIT)	0.530 seconds
Sun	Visual Numerics Java library (w/o JIT)	4.540 seconds
Intel	Direct call of Lapack SGEQRF	0.42 seconds
Intel	Java Native call of Lapack SGEQRF	0.441 seconds
Intel	Visual Numerics Java library (w/o JIT)	4.98 seconds

Table 5.3: This table displays a comparison of two computers running a QR factorization algorithm, on a two hundred by two hundred matrix, in various languages. The machine marked "Sun" uses an UltraSPARC II 300Mhz, and the machine marked "Intel" uses a Pentium II 266Mhz. On the UltraSPARC, one can observe that interpreted Java is about thirty times slower than the Lapack routine, but that using the "just in time" compiler improves this to only three and a half times slower. For the Pentium II, the direct call of the Lapack routine and the call of the Lapack routine using the Java native interface are almost identical. The import of the good performance of the Java native interface is that one can use Java for high level data processing and use native languages for numeric algorithms with little loss of performance.

Java's native language interface bears closer inspection, since the idea of calling low level languages from high level languages will occur again in later chapters. Java provides an interface through which routines may be called which are written in a language different from Java. The advantage of this interface is that Java may utilize the features of the other language, such as speed. The disadvantage is that Java will also suffer from the problems of the other language, such as lack of portability. The approach used in MCCC is to use Java for as much of the program as possible, and then only to rewrite those sections which are computationally expensive. In this way, the vast majority of the code is portable while only small sections are not. Table 5.4 shows a selection of interfaces from MCCC and notes which are native language routines and which are written in Java.

Routine	Language
CollocationDiscretization	Java
ContinuationNatural	Java
ContinuationPseudoArclength	Java
ContinuationRheinboldt2D	Java
FDDiscretization	Java
IVPEuler	Java
IVPRungeKutta2	Java
NewtonRaphson	Java
ODEFunctionRod	Java
ODEFunctionDNA1ParNative	Fortran
ODEFunctionDNA6ParNative	Fortran
ODEFunctionPlanarStrut	Java
PowellHybridNative	Fortran

Table 5.4: This figure shows some of the interfaces in MCCC. The bulk of the routines are written in Java. ODEFunctionDNA1ParNative and ODEFunctionDNA6ParNative are "right hand sides" for ordinary differential equations which are written in a native language because they are called frequently. PowellHybridNative uses a Powell hybrid scheme from the MINPACK [37] library for solving systems of nonlinear equations. Such a SolutionCorrection procedure is often a large part of the computational expense of numerical continuation algorithms, so we provid an optimized native language version. It should be noted that, for each native language routine, there is always a Java replacement which performs the same function. We have implemented MCCC in this manner so that we may maintain portability.

We now enter the realm of parallelism by noting that the conceptual class diagram in Figure 5.1 is slightly misleading in that it implies that there is only one SolutionCorrection object in use by a GuessGeneration object. In a multi-threading, or more generally multi-processor environment, GuessGeneration objects may use multiple SolutionCorrection objects, especially in the case where $m > 1$. GuessGeneration objects, in the course of their computations, can often have several pending guesses at one time. If a small amount of additional intelligence is added to the GuessGeneration object, so that it can coordinate several SolutionCorrection objects, we can actually implement the three tiered parallelism described in Section 5.3. Currently, no GuessGeneration object has been implemented to use multiple SolutionCorrection objects, though, we have implemented a version of the SPWS algorithm, described in Section 3.3, which does allow all three tiers of parallelism to be used.

The first tier of parallelism is to have more than one GuessGeneration object working at once. Canonically, there will be no communication between the GuessGeneration objects. The GuessGeneration objects will only communicate with the Control Process from Figure 5.2. The second tier of parallelism is the ability of one GuessGeneration object to have several SolutionCorrection objects working at once. For example, a multi-parameter GuessGeneration object could, at any one time, have several guesses pending which require correction. It could utilize several SolutionCorrection objects to correct these guesses simultaneously. The communication requirements are again small. The final tier of parallelism is to actually parallelize the SolutionCorrection object itself. In this case, the communication overhead can be very high. For this part of the code, we can use a third party software (be it in Java or a native language such as C or Fortran), since a great deal of work has been done in these areas that we have no reason to reproduce. Examples of such software include PETSc [9] and SCALAPACK [10].

MCCC implements the ideas discussed in Chapter 4 in an efficient and portable way. Routines from standard libraries can be used through the Java native language interface to efficiently utilize the underlying hardware, while the higher level routines can be written in a modular way by using dynamic method lookup and runtime typing.

5.6 Future Work

MCCC is designed to allow the various pieces which make up a continuation algorithm to be easily assembled, allowing different combinations to be experimented with. The modularity provided by Java, as well as the ability to use native language libraries, even allows us to implement all three tiers of parallelism in a straight forward way. While the structure of MCCC is designed to be very efficient, the implementation of some of the individual components are experimental. Also, the lack of a good Java implementation of sparse matrices makes some of the computations for large systems slow. On the other hand, MCCC has been used for non-trivial calculations, and two example problems on which MCCC has been used can be found in Chapter 8, in Section 8.2 and Section 8.3.

Chapter 6

Visualization Methods

6.1 Introduction

The previous chapters have discussed the computation of bifurcation manifolds. This chapter will discuss the exploration of the data, after it has been computed, using ideas from scientific visualization. Visualization is important because each solution to equation (2.1) can be very complex in of itself, to speak nothing of the complexity of the bifurcation manifolds. The sections of this chapter will cover the cases where $m = 1, 2, 3$, mentioning both general and problem specific methods. Many of the ideas used throughout this chapter will initially be discussed in Section 6.2 and elaborated on in later sections as necessary.

The general visualization problem we have can be described as follows. Consider a single solution to an equation of the form of equation (2.1) or equation (3.1). The dimension of the system can be quite large, especially when the system is constructed as a discretization of a continuous problem. For example, we routinely consider the case where $n \approx 1000$, and we have worked with problems where $n \approx 10000$. We are not only interested in single solutions to equation (2.1) or equation (3.1), but families of solutions, possibly involving 10^6 solutions. Continuation methods that we utilize generate large numbers of solutions to these problems, and we wish to explore the relationship between these solutions and the various parameters on which they depend.

6.2 1D Manifolds

6.2.1 Example problem

We will begin by examining the case where $m = 1$. An example is shown in Figure 6.1. In general, visualizing such structures for $n \leq 4$ is trivial since one may plot three coordinates and utilize a fourth for color. Unfortunately, as stated before, we often have $n \gg 1$ and more specialized procedures are required. We will use as an example problem throughout this section a discretization of two-point boundary value problems for the set of ordinary differential equations (ODE) shown in equation (6.1).

$$\begin{pmatrix} \dot{\mathbf{r}} \\ \dot{\mathbf{q}} \\ \dot{\mathbf{n}} \\ \dot{\boldsymbol{\mu}} \end{pmatrix} = \begin{pmatrix} \mathbf{d}_3(\mathbf{q}) \\ \frac{1}{2}\sum_{i=3}^{3} u_i B_i \mathbf{q} \\ \mathbf{0} \\ \frac{1}{2}\sum_{i=3}^{3} u_i B_i \boldsymbol{\mu} - \left(\frac{\partial \mathbf{d}_3(\mathbf{q})}{\partial \mathbf{q}}\right)^T \mathbf{n} \end{pmatrix} \tag{6.1}$$

Here $\mathbf{r}(s) \in \mathbb{R}^3$, $\mathbf{q}(s) \in \mathbb{R}^4$, $\mathbf{n}(s) \in \mathbb{R}^3$, and $\boldsymbol{\mu}(s) \in \mathbb{R}^4$ for a fourteen dimensional system. In this equation $\mathbf{d}_i$ is a quadratic function of $\mathbf{q}$, B_i is a constant matrix, and $u_i = \frac{2}{\|\mathbf{q}\|^2}\mathbf{q}' \cdot B_i\mathbf{q}$. This system has been used extensively in our group as a model for DNA mini-circles. We commonly utilize the method of orthogonal collocation [15] to discretize the ODEs, which results in a system of equations where $n \approx 1800$. A bifurcation diagram computed for equation (6.1) (subject to certain boundary conditions) is shown in Figure 6.1.

6.2.2 Post-processing and Bifurcation Diagram Visualization

Given a problem as described above, there are several steps one goes through to produce a picture such as shown in Figure 6.1. The first step is to perform some type of post-processing. Some set of values, hopefully small, can be chosen or computed from the full solution at each point of the bifurcation manifold. We call these values *representatives* of the bifurcation manifold. This process, by definition, is problem dependent. For example, if we are working with the elastic rod problem shown in equation (6.1), it is often useful to choose as representatives those variables in the discretization of the problem which correspond to the boundary values of the continuous problem. More complex calculations may also be performed. For example, we may compute important functionals. There are several integrals over solutions of equation (6.1), such as elastic energy or a geometric quantity called writhe [36], which are useful for visualization purposes. These quantities can be computed at each point of the bifurcation manifold and then used for visualization.

As an example of the usefulness of post-processing, we note that a bifurcation diagram is a large set of solutions to a given problem. For example, we have generated data sets with thirteen thousand solutions to the problem shown in equation (6.1). If each solution consists of approximately two thousand double precision numbers the total size of this single bifurcation diagram, without any post-processing, is approximately two hundred megabytes. After post-processing,

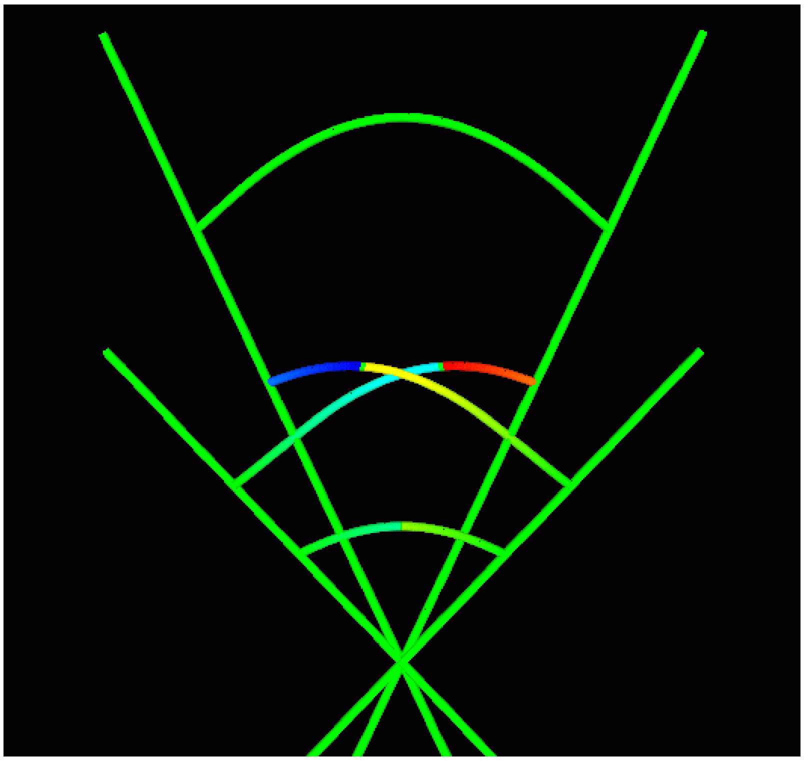

Figure 6.1: This figure shows a collection of one dimensional manifolds which comprise a bifurcation diagram for equation (6.1). In this case, $n \approx 2000$. The data has been post-processed to choose eleven representatives, of which two are plotted here and a third is used to define the color. Of the eleven representatives in this bifurcation manifold, three are moments computed at the starting boundary point using the μ equations, one is the only nonzero component of $\mathbf{n}$ (which is constant over the solution), one is an integral over the centerline $\mathbf{r}$ called writhe, one is the maximum value of the first component of $\mathbf{r}$ over the solution, two are the minimum and maximum radius of the solution, two are the minimum and maximum of the norm of the moments computed from the $\boldsymbol{\mu}$ equations over the solution, and the last is an angle computed from the values of the $\mathbf{q}$ equations at both endpoints. The two quantities which are plotted here are the first two moments, and the color is based upon writhe.

and selection of representatives, the size is just over one megabyte, with no effective data loss for the user. We call this procedure *data reduction*[1]. A data reduction algorithm is problem dependent. It is possible that effectively no use-

[1]As opposed to data compression such as is performed by the Unix utility gzip and other similar software, and does not depend on the specific structure of the data.

ful information is lost in a good data reduction strategy, either because the full solution set is itself too large to comprehend, and all information of use to the user is available, or enough information is retained so that all useful information can be recomputed efficiently on demand (e.g. see Section 6.2.3).

If the number of quantities that we wish to have in the bifurcation diagram is still greater than three, there are several actions we can perform. First, we may arbitrarily choose multiple sets of three of the quantities and use standard methods to visualize them simultaneously as curves embedded in different spaces. Such interactive three dimensional visualizations are more effective than static two dimensional images because they utilize the innate human ability to process three dimensional data. Unfortunately, the usefulness of this method decreases when the number of such three dimensional visualizations is large, and an understanding of the inter-relationship between many quantities is desired. In such a situation, one might utilize other tools of scientific visualization to go beyond three quantities. For example, one may certainly utilize color to include more information. In Figure 6.1, the color shows an additional quantity, where blue represents low values and red represents high values. One may even use the thickness of the lines with which the diagram is drawn to represent information, giving a total of two additional quantities which may be displayed.

6.2.3 Solution Reconstruction

Much of the data is discarded in the post-processing step when moving from the full solution to the representatives used in the bifurcation diagram. While it is not possible to visualize all of the full solutions to the problem for every point of the bifurcation diagram (since that was the reason we did the post-processing in the first place), it is often useful to be able to reconstruct the full solution at certain selected points of the bifurcation diagram. For example, see Figure 6.2.

This idea of solution reconstruction bears closer inspection. Post-processing, if performed intelligently, can sometimes enable us to achieve the large data reductions discussed previously, while still maintaining access to full solutions. The point is that the computation of a solution to equation (2.1) may be quite expensive, but, depending on the structure of the problem, knowledge of a part of the solution may allow the fast reconstruction of the rest of the solution. For example, if the problem of interest is a discretization of a continuous problem, then one may be able to store the solution at a coarser discretization than was actually computed. Visualization may then be performed on the coarse discretization utilizing interpolation. Another example is the process we utilize for the ODE boundary value problem shown in equation (6.1), which is not stiff. For this type of problem, it suffices to store the values of the solution solely at one endpoint. Solving the ODE two point boundary value problem is (somewhat) expensive, but

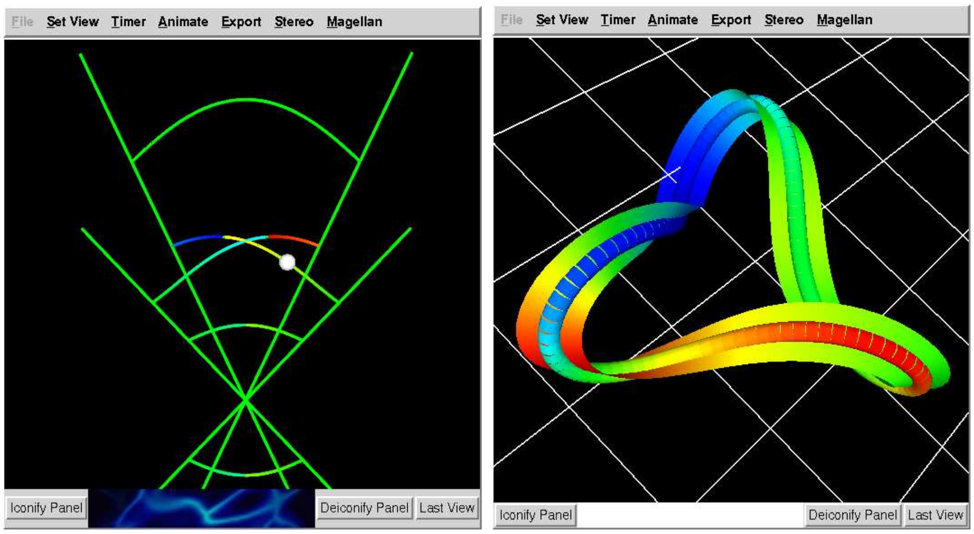

Figure 6.2: On the left we see a bifurcation manifold with a single point marked by a ball, and on the right we have a visualization of the whole solution which corresponds to that point. This data set is generated from the ODEs in equation (6.1). In this case we utilize a tube, which shows the centerline $\mathbf{r}$, and a ribbon, which shows a director computed from the $\mathbf{q}$ equations. In this way one can visualize the overall structure of the solution set while having interactive access to visualizations of the full solutions.

once the solution is known, and the values at one endpoint recorded, the solution may be trivially reconstructed by solving an appropriate initial value problem. Basically, by recording a subset of the solution we have transformed the problem in question from a computationally expensive one, into a computationally inexpensive one.

6.2.4 Separation of Computation and Visualization Domains

We take this opportunity to briefly discuss a difficulty of steered computations, of which the above discussion is an example. Often it occurs that the form of the problem that one solves numerically is not the same as the form which one wishes to visualize. Several edifying examples of this problem can be found in our treatment of equation (6.1). As previously noted, the visualization and numerical discretization requirements may not be the same. For example, the $\mathbf{r}$ in equation (6.1) represents the centerline of the rod (e.g. see Figure 6.3), so this part of the solution may be visualized "directly". We say "directly" because the numerical discretization necessary for accurate numerics is often much finer than the discretization useful for visualization. On the other hand, the ribbon shown in Figure 6.3, while being a very useful visualization tool, is not an inherent part of equation (6.1). It is, in fact, a function of the values of the unknowns $\mathbf{q}(s)$ (which are, in turn, a quaternion parameterization of a rotation matrix, see [29] for more details). The $\mathbf{q}(s)$ were chosen because they are easier to treat numerically than the values which directly comprise the ribbon [29]. The previous two points show the difficulty in combining computation and visualization for single solutions of equation (6.1). On the other hand, as mentioned in Figure 6.1, the construction of the bifurcation diagram may be more difficult. For example, of the eleven quantities used to construct the bifurcation diagram shown in Figure 6.1, only one quantity is directly available from the solution of equation (2.1). The rest require computations, some of which may be quite expensive. The point which we wish to emphasize is that the discretization and data used for computation and visualization may be quite disparate, and a system which wishes to combine them together must be flexible enough to handle the differences between the two domains.

6.2.5 Special Projections

We will now consider other properties of bifurcation diagrams, which are important for visualization. We will discuss two examples of special bifurcation diagrams which provide additional information and are especially suitable for visualization. The first idea is that of *topological correctness* [17]. A topologically correct bifurcation diagram is one which has the same topology as the bifurcation manifold from which it is derived. We wish to use visualizations of the bifurcation diagram which maintain this property. For example, the visualization of the bifurcation diagram shown in Figure 6.1 is not topologically correct because the transverse branches are double covered, even though the bifurcation diagram itself is topologically correct. Topological correctness in visualizations can sometimes be recovered by a small perturbation of the bifurcation diagram. Instead of plotting values directly from the bifurcation diagram one may perform a simple post-processing step. Take two quantities from the bifurcation diagram and compute a third quantity which is a combination of the two. For example, one could compute $F(x, y) = x\cos(\theta) + y\sin(\theta)$ where θ is a given angle. This corresponds to a rotation in the plane spanned by other components x and y. An example of a topologically correct visualization of a bifurcation diagram can be seen in Figure 6.4. This visualization is created from the diagram in Figure 6.1 by an application of the above perturbation. The second idea is that of a *distinguished diagram* [33] in which the direction of stability exchange at a fold point may be discerned by examining specific bifurcation diagrams. An example of a distinguished diagram, taken from [45], is shown in Figure 6.5. The impor-

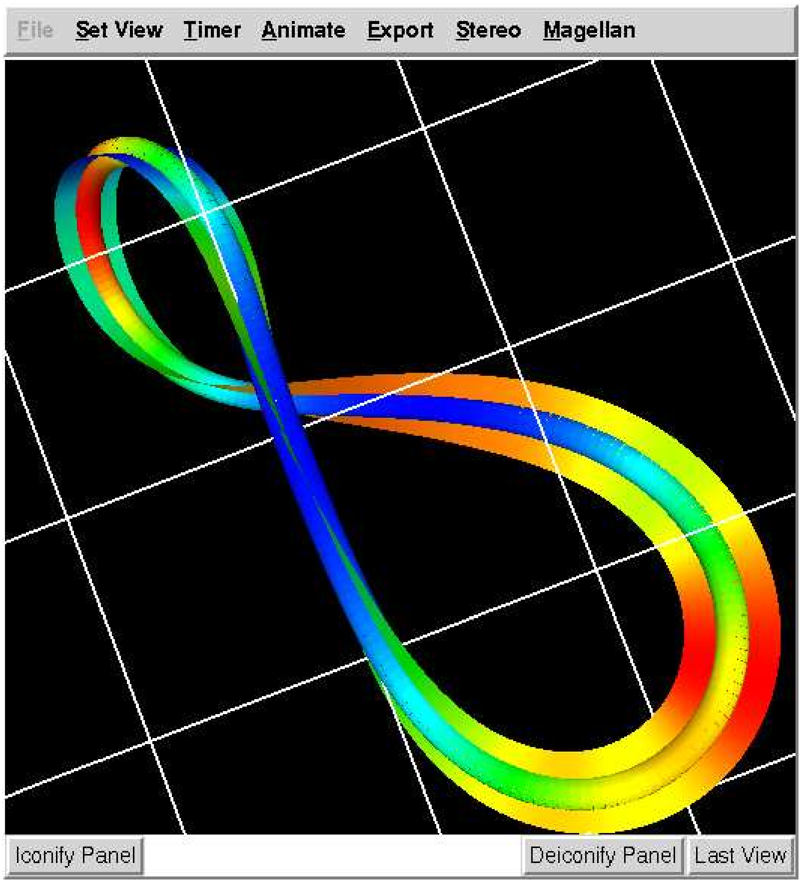

Figure 6.3: This figure shows an example of a visualization of a solution of equation (6.1). The tube shows the centerline defined by the **r** equations, while the ribbon shows the "twist" of the solution as defined by a function of the **q** equations. The color of the tube is derived from the bending energy, and the color of the ribbon is derived from the twisting energy.

tant point here is that different views are required for different purposes, and any visualization tool which you wish to use for these problems must be flexible.

6.3 2D Manifolds

We now enter the realm $m = 2$. An example of such a bifurcation diagram is shown in Figure 6.6. All of the comments from the previous section also apply to this case, but because of the added complexity of two-dimensional manifolds, there are a few additional points which we must address.

We first note that many of the problems discussed in the previous section

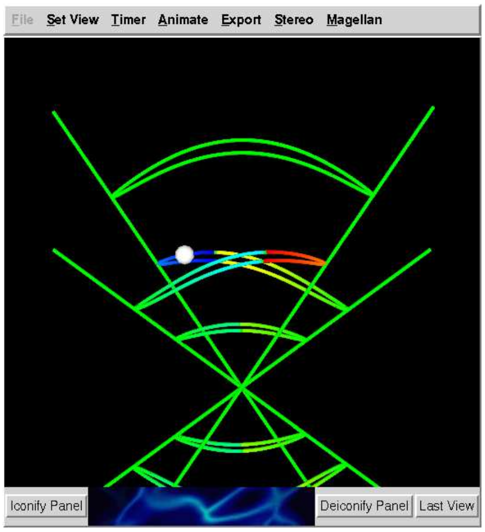

Figure 6.4: This figure shows the same bifurcation diagram as seen in Figure 6.1, except that this visualization has undergone a small perturbation to make it topologically correct. The curved transverse branches in Figure 6.1 are actually double covered, and the perturbation has been applied in this figure to allow both branches to be seen. Of the quantities plotted here, one is taken directly from the data which defines the bifurcation diagram, and the second is derived from a function of two other quantities.

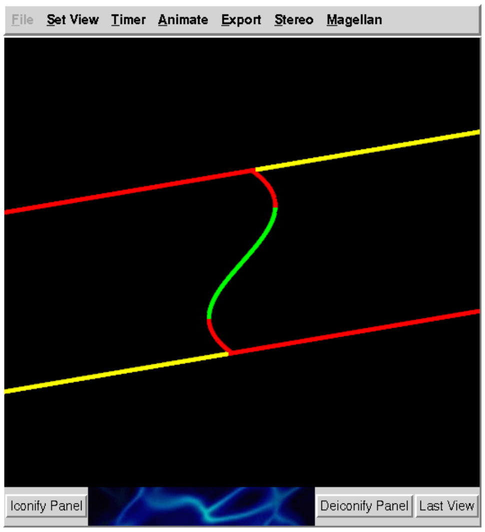

Figure 6.5: This figure shows an example of a distinguished diagram from [45]. The color of a point denotes the stability of the solution at that point. Stability exchanges occur in a predictable way exactly at folds in the diagram.

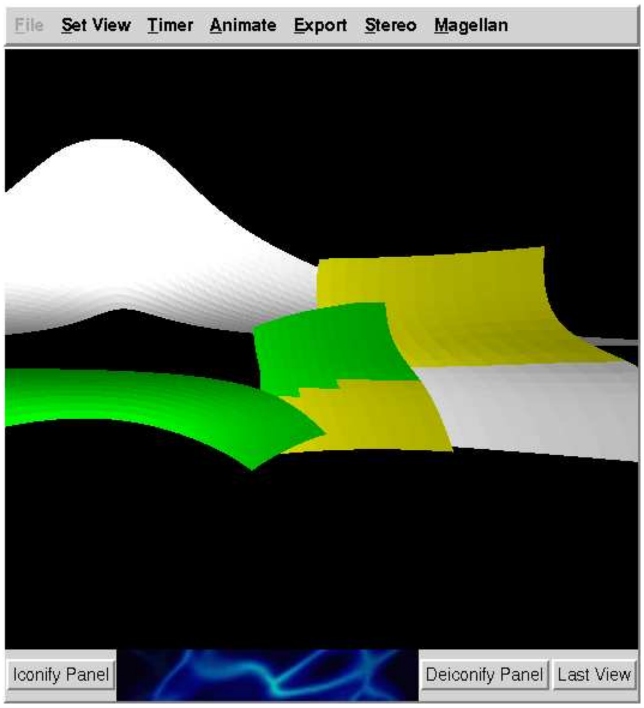

Figure 6.6: This figure shows a collection of manifolds with $m = 2$ for the problem discussed in Section 8.2. In this case $n = 6$. Two of the displayed quantities are components of the solution to the equations, and one is a computed quantity.

are exacerbated in the $m = 2$ case. For example, the number of solutions which comprise a two dimensional manifold are far greater than that which comprise a one dimensional manifold, with commensurate storage requirements and needs for post-processing.

An important difference between the one and two dimensional cases is in the actual construction of the manifold for visualization. In the one dimensional case, it is obvious how one should break the manifold into simplices. Each point is connected to the point which follows it in the continuation procedure. In the two dimensional case, it is far less obvious. Algorithms such as Rheinboldt's Moving Frame method implicitly imply a triangulation, while Henderson's Overlapping Disk method does not. For the actual graphical generation of the surface we often require a triangulation and it must be delivered, at a nontrivial computational expense, for those algorithms which do not themselves provide it. Even when the computational triangulation exists, it may be too fine (or coarse) for visualization purposes and must be modified by procedures such as splining or vertex deletion.

Given that we have a triangulation of the manifold we must still construct useful visualizations of it. One useful tool is to compute contour lines on the surface. In such a way we may reduce the dimension of the manifold to one and use the tools described in the previous chapter. On the other hand, we might want to visualize the surface itself. For example, the surface may be shaded using explicit lighting calculations, such as ray-tracing, or approximations thereof, such as Gouraud shading [19]. A selection of rendering methods for two dimensional manifolds are shown in Figure 6.7.

6.4 Future Work: 3D Manifolds

Finally, we enter the realm of $m = 3$. An example of such a bifurcation manifold is shown in Figure 6.8. Again, the comments from Section 6.3 apply to this case, but there are more complexities that must be addressed. While several of the algorithms which we have discussed, such as the SPWS algorithm from Section 3.3 and Rheinboldt's algorithm from Section 3.4.1, may be utilized to compute three, or even higher, dimensional manifolds, the visualization of such manifolds is significantly more difficult. We have only performed preliminary experimentation in this area, and we present in this section a brief survey of some tools that might be used to visualize such manifolds.

The number of solutions which comprise a three dimensional manifold are greater than that which comprise a one or two dimensional manifold. Generally speaking, the number of solutions in a three dimensional manifold is the cube of the number of solutions in an equivalent one dimensional manifold, with commensurate additional storage and number of simplices. The most important difference, though, between the three dimensional case and the lower dimensional cases is that the three dimensional case is *co-dimension zero* or *space filling* in the world our brains are used to. In other words, when the manifold is viewed in three dimensions there are portions which are "inside" and are occluded by other portions of the manifold, as seen in Figure 6.8.

Several useful algorithms exist for rendering a three dimensional manifold. First, isosurfaces may be computed, reducing the visualization problem to the two dimensional case. Such procedures are nontrivial, especially on complex meshes, but algorithms, such as marching cubes [30], have been developed to generate isosurfaces of volumetric data. Similarly, the surface may be sliced (i.e. two dimensional subsets can be constructed) and the slices visualized. One could also render the three dimensional manifold as a solid object, using ray-tracing or Gouraud shading of the external surface, accepting that all of the internal detail will be occluded. On the other hand, one could only draw the one dimensional edges which make up the simplices of the manifold. One can then view internal structure, but visualization given by this method is somewhat confusing and non-intuitive. Examples of some of these methods may be seen in Figure 6.9.

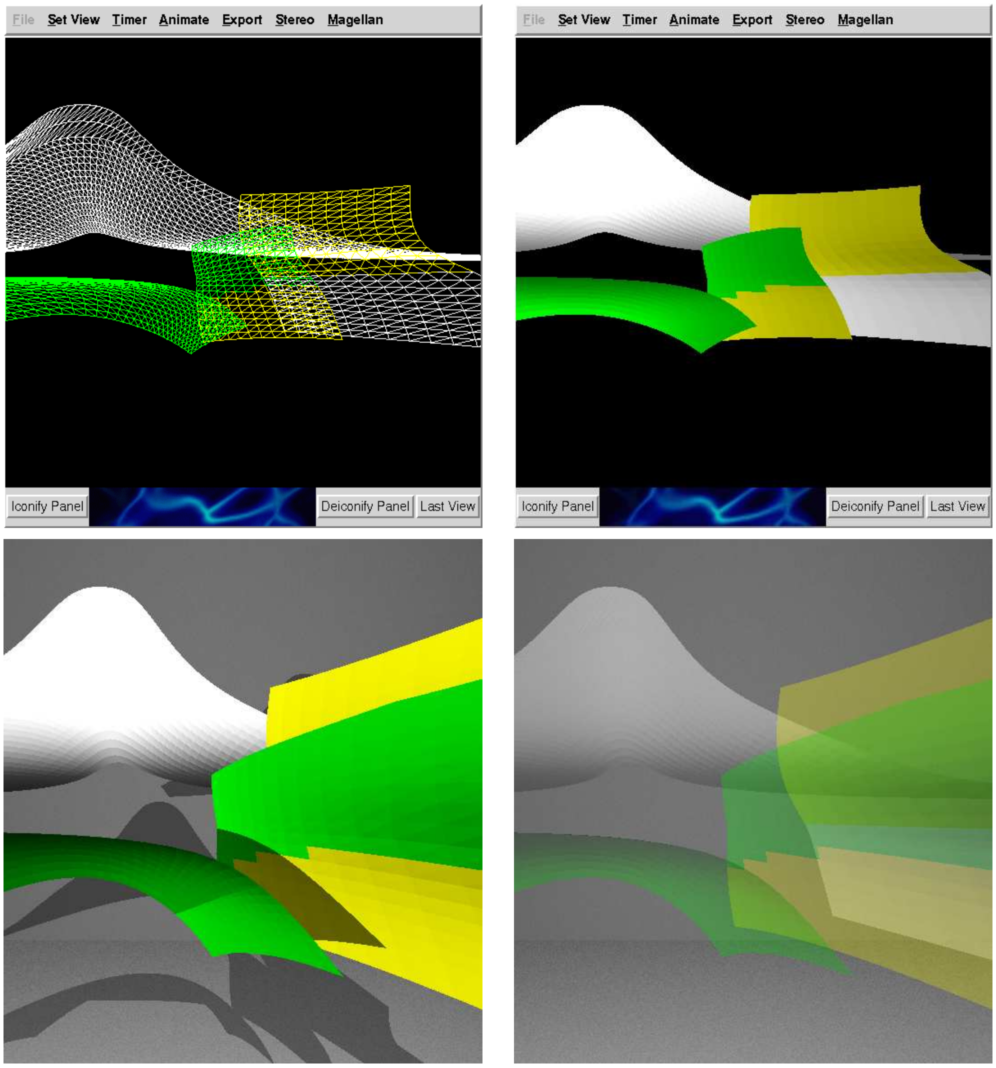

Figure 6.7: This figure shows a collection of manifolds with $m = 2$. The top left window shows a two dimensional manifold in which only the edges of the triangulation have been drawn. The top right window shows a surface rendered using Gouraud shading in OpenGL [39]. The bottom left window is a ray traced example using POV-ray [11]. The bottom right window is a ray traced example using POV-ray with the addition of transparency (we have removed the shadows to make the image easier to understand). The effectiveness of different visualization techniques is very different between a workstation, where interactive three dimensional rotations are available, and static two dimensional images. Moreover, in two dimensional images the quality of the printing can be crucial (e.g. when using transparency).

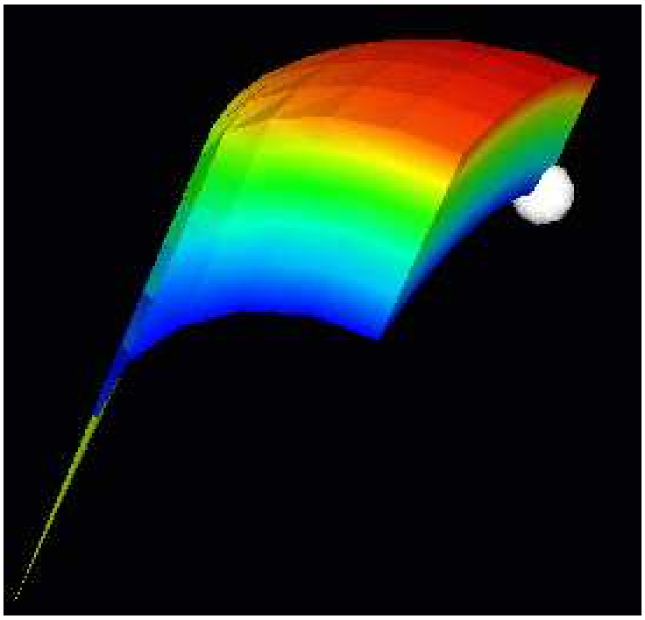

Figure 6.8: This figure shows a manifold with $m = 3$. The main problem with the visualization of such manifolds can clearly be seen in this figure. Most of the data is occluded by other parts of the manifold, and we can only see the portion of it that lies on the outside facets.

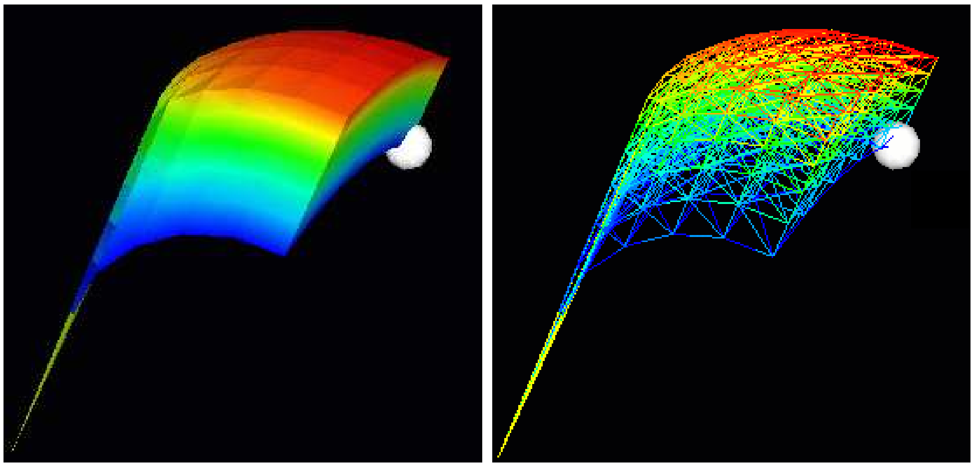

Figure 6.9: This figure shows two manifolds with $m = 3$ draw with various rendering algorithms. The left window shows a three dimensional manifold drawn as a solid object. All internal detail is lost. The right window show a three dimensional manifold drawn as a collection of edges. Internal detail is visible, but overall structure is difficult to discern.

A final class of rendering methods which bears closer examination is the utilization transparency. In general, such methods render the three dimensional manifold by treating it as a translucent object which only partially blocks the transmission of light. Different parts of the object can be assigned *transfer functions* which define how much light is allowed to pass through the object and how much is absorbed [19]. If the transfer functions are intelligently chosen, the generated visualizations can be quite useful. Utilizing such methods, a three dimensional manifold can be rendered in such a way that both the overall shape, as well as the internal structure, can be visualized simultaneously. An example of a three dimensional manifold rendered with such an algorithm is shown in Figure 6.10. Unfortunately, many such methods are quite slow and are not suitable for interactive visualization, though work is currently underway to make such visualization methods interactive using texture mapping hardware [53].

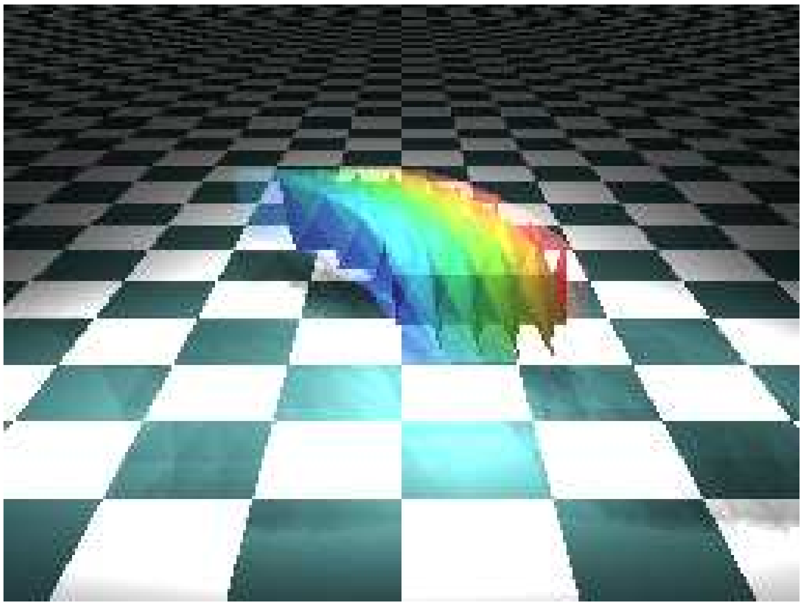

Figure 6.10: This figure shows a three dimensional manifold which, as an experiment, has been rendered using transparency. Both overall structure and internal detail are visible, though work remains to be performed in this area. The checkerboard background is necessary to see those parts of the manifold which are particularly transparent.

In Section 8.3 we will display a three dimensional manifold which has been computed using the SPWS algorithm from Section 3.3. Utilizing the SPWS method, the calculation of such a manifold is straightforward, but we lack visualization tools which allow the user to efficiently steer the computation. We consider the development of visualization tools for three dimensional manifolds which allow interactive steering of parameter continuation computations to be an important open topic for future research.

Chapter 7

Visualization for Bifurcation Manifolds (VBM)

7.1 Introduction

The previous chapters have discussed many ideas regarding the computation and visualization of bifurcation manifolds. VBM is an attempt to meld all of these ideas into one software package, combining visualization and computation into one coherent whole. Several goals have driven the design of VBM. First, we want to use visualization and special projections of the bifurcation manifold to facilitate efficient computation of the solution set, to organize and allow easy access to the large data sets that arise, and most of all, to generate mathematical understanding of the features and properties of the solution set. Second, we want the design to be very modular and extensible, allowing the easy incorporation of pre-existing software packages, especially compute engines based on existing implementations of continuation algorithms.

Section 7.2 will describe the various Application Programmer Interfaces (APIs) that have been developed for the visualization and computation components of VBM. In this section the description will not involve a specific programming language and will be purely concept based. Section 7.3 will discuss some of the specific implementation details of VBM. Further documentation and a tutorial can be found at `http://lcvmwww.epfl.ch/VBM`.

7.2 API for Multi-parameter Continuation Methods

The most important part of designing a modular object oriented software such as VBM is API design. Basically, APIs prescribe how different parts of the program communicate and interact, by defining a contract which each part of the program is expected to fulfill. The importance of good APIs cannot be over-emphasized. If the APIs are correct for the problem you wish to solve, but some specific code module is found to be inadequate, then only that code module needs to be fixed. On the other hand, if an API is found to be inadequate, then all code modules which use that API will have to be modified, a much larger task.

The first step in the design of any set of APIs is an examination of the problem domain that one wishes to solve. For VBM, the problem domain is exactly that which was discussed in Chapters 2, 3, 4, and 6, namely, we wish to

- compute bifurcation manifolds for a variety of problems,
- perform post-processing of data,
- visualize the results at various levels of detail.

To this end, we decided that our main computational and visualization entity would be the bifurcation diagram.

The first goal was to be able to study the bifurcation manifold itself in its entirety (or at least that subset which has already been computed). Accordingly, the first API defines how we represent bifurcation diagrams and it is called the *Bifurcation Visualization API*. Note that this interface does not define how the visualization occurs, only how the data is stored and interacted with. This freedom allows the modules which actually implement the interface to present the data in any way that is appropriate for the problem at hand.

On the other hand, each point of a bifurcation manifold can itself be a very complex object. As discussed in Chapter 6 there is often no way to encode all of the information contained in a bifurcation manifold in a single bifurcation diagram. In this case, we need to be able to choose individual points of the bifurcation diagram, and look more closely at the solutions which they represent. Accordingly, the second API defines how to extract points from the bifurcation diagram and examine them. This process is defined by the *Data Probe API*. Again, the specifics of the visualization are controlled by the individual data probe modules and not explicitly defined by the API.

The previous two interfaces define how one can explore static subsets of a bifurcation manifold, but often the the question of interest can only be answered by extending the manifold through further computation. In such a case, continuation methods can be applied to the bifurcation diagram to extend it. Accordingly, the *Compute Engine API* defines how one can extract data from the bifurcation diagram, send that data to a continuation algorithm, and add the newly generated data back to the bifurcation diagram. The Compute Engine API may also be used to post-process the raw data in the bifurcation manifold to create a bifurcation diagram.

All three of the interfaces and their interrelationship can be seen schematically in Figure 7.1.

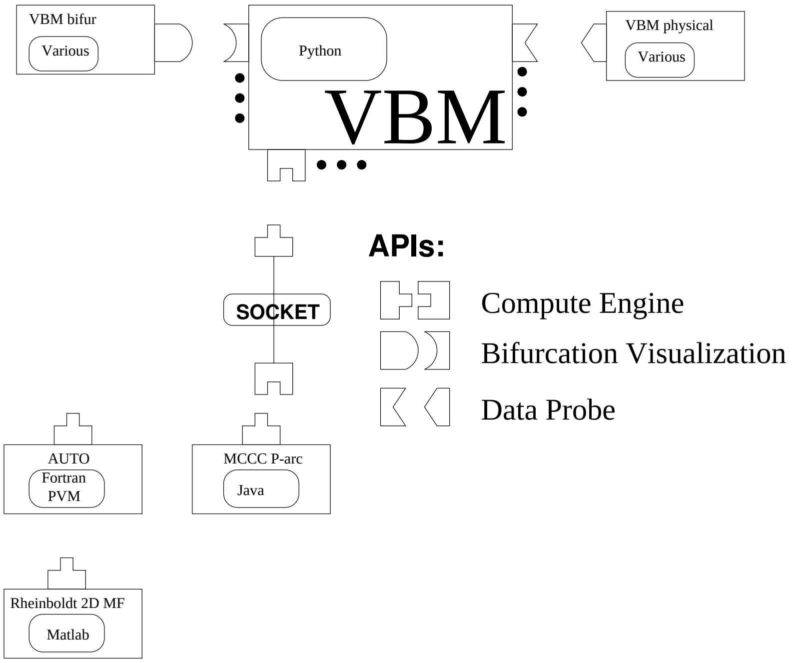

Figure 7.1: This figure shows a block diagram of the VBM APIs. The API symbols show where external programs may be used with VBM. Of special interest are the compute engines. The names of the example modules are shown in the top of each box, while the implementation language is shown in the rounded box inside each module. The line with compute engine interfaces at each end represent a networking interface, which allows any compute engine to be executed on a computer other than the one on which VBM itself is running (e.g. a remote supercomputer). The module named "AUTO" uses [16], "MCCC P-arc" is based on [14, 15]. and "Rheinboldt 2D MF" is based on [43].

7.2.1 Data Structures

There are many data structures used in VBM, but two in particular bear closer inspection because they are essential to the definition of the various APIs which we will discuss.

The first data structure, and one of the single most important parts of VBM, is the bifurcation manifold object. Geometrically speaking, the bifurcation diagram object is a list of points in $\mathbb{R}^p$, with p being the number of representatives chosen in the construction of this bifurcation diagram, along with connectivity (e.g. see Example 5 in this Section). Normally, the vectors in the bifurcation diagram object are not a solution of equation (2.1). Often, some representatives are chosen, or quantities computed, which somehow capture the overall structure of the solution (see for example Section 8.4 and Section 8.5). The user is totally free in this regard and may include in the bifurcation diagram object any data that they wish. The user is not even restricted to numeric data, since each point can also contain *tags*. These tags provide additional information about the point. For example, one can store information such as how one starts a computation at that point, or how one can reconstruct the entire solution from the given representatives. As currently implemented, the VBM core program only uses a few tags, the vast majority are user (or more precisely module) defined.

As the bifurcation diagram object is of such importance we will give an example. For ease of use we have defined a file format in which a bifurcation diagram object may be stored. The file format allows bifurcation diagram objects to be created using a wide variety of tools, not just VBM. An example of a bifurcation diagram object, in this file format, is shown in Examples 1, 2, 3, 4, and 5 which can be found at the end of this section.

The second data structure which VBM uses is the pointer object. The pointer object encapsulates an address of one point of the bifurcation diagram. These are used to designate points that are either being visualized by the data probes or serve as the starting point of a compute engine. The pointer object is a very simple data structure which we will not further elaborate on here.

```
# AUTO_EQ 0 4
# AUTO_PAR 5 6
# AUTO_EXTRA_DATA_REPORT 7 11 1.0
# VBM_Default_Plot_Coordinates 7 4 5 1
N3;phi;m2;r1;r3;lambda;nu;eN3;ephi;em2;er1;er3
12
7 2
3
# AUTOFort8_file /tmp/fort.8 1
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
# AUTOFort8_file /tmp/fort.8 5
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
# AUTOFort8_file /tmp/fort.8 7
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
4
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
##Begin VBM_Connectivity
line 1 0 1 1
line 1 1 1 2
triangle 2 0 2 1 2 2
line 2 2 2 3
##End
##Begin VBM_Compute_Module
NULL
./r.strut
./strut_ivp.f
./work
VBM_1D_AUTO
Continuation with IVP solver
##End
```

Example 1: This example shows the overall structure of a bifurcation manifold object as it can be represented in an ASCII file. The individual sections of this data file are described in greater detail in Examples 2, 3, 4, and 5.

```
# AUTO_EQ 0 5
# AUTO_PAR 6 9
# AUTO_EXTRA_DATA_REPORT 10 15 1.0
# VBM_Default_Plot_Coordinates 7 4 5 1
```

Example 2: This section of the bifurcation diagram object file gives a list of tags that apply to all points in the bifurcation diagram object, which our interface calls *global tags*. A tag is defined as a # followed by a tag name. The arguments which follow the tag name are user definable. In the example shown here the "AUTO_EQ" and "AUTO_PAR" tags are intended for a compute engine based upon AUTO [16]. These tags tell the compute engine what columns of the data are the ones it should use for its computation. The "AUTO_EXTRA_DATA_REPORT" tag is an optional tag for the above compute engine which asks it to compute additional data. The "VBM_Default_Plot_Coordinates" is a general tag for VBM which various modules use to determine what graphics to draw by default. The tags are quite flexible. We also note that all of these tags are optional. A bifurcation diagram object file is legal even if it contains no global tags. On the other hand, specific modules may check for the existence of certain tags, and not function if the expected tags are not present.

```
N3;phi;m2;r1;r3;lambda;nu;eN3;ephi;em2;er1;er3
12
7 2
```

Example 3: This example shows the header information which is required of all bifurcation diagram object files. The first line is a list of names by which the representatives in the file are known. They can be used to denote the representatives in various modules. The second line denotes the number of columns that each data point in the file should contain. This entry is mainly used for error checking. The third line contains two numbers. The first is the total number of data points in the file and the second is the number of sections of data in the file. The first number is not required and only exists for backward compatibility with former versions of the software. In current versions it may just be set to zero.

```
3
# AUTOFort8_file /tmp/fort.8 1
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
# AUTOFort8_file /tmp/fort.8 5
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
# AUTOFort8_file /tmp/fort.8 7
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
4
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0 12.0
```

Example 4: This example shows the actual body of the bifurcation diagram object file. Each section of data begins with an integer which represents the number of data lines in the section. Each data line is a list of floating point numbers which are the representatives of a point of the bifurcation manifold. In addition, a line may be proceeded by any number of *point-wise* tags. Each point-wise tag applies to the point which follows it. For example, the point-wise tags in the first section, namely "AUTOFort8_file", inform various modules that additional data about this point is available in the given filename. In this example, the additional data can be used for full solution reconstruction.

```
##Begin VBM_Connectivity
line 1 0 1 1
line 1 1 1 2
triangle 2 0 2 1 2 2
line 2 2 2 3
##End
##Begin VBM_Compute_Module
NULL
./r.strut
./strut_ivp.f
./work
VBM_1D_AUTO
Continuation with IVP solver
##End
```

Example 5: This example shows the final section of the bifurcation diagram object file. This part of the file is very similar to the global tag section, except, instead of having all of the data for the tag on a single line, the data may be spread over several lines. Such data blocks are normally used to give additional information which is required for continuation or solution reconstruction, and each data block begins with ## followed by a block name. The two examples shown here have more specialized purposes. The first block, "VBM_Connectivity", is a general block which allows the bifurcation diagram to have arbitrary connectivity. Simplices may be defined here which use as vertices the data from the previous section. If this block is not defined, the bifurcation visualization modules may prescribe the connectivity as they wish. Canonically, they will draw the sections of data as a set of line segments which connect the data points in that section sequentially. The second section, "VBM_Compute_Module", gives a description of how to create a compute engine (see Section 7.2.5), which may be used to extend this data set.

7.2.2 VBM Core

The core of VBM is a very simple program. It is basically only a backplane onto which modules can be plugged, providing no services other than basic communication. Many different modules may be used simultaneously. For example, several bifurcation visualization modules, data probe modules, and compute engine modules may all be used together. The idea is that we want to be able to use as wide a range of computation and visualization tools as possible. Accordingly the VBM Core program makes very few assumptions about the form of the problem. In fact, if it were desirable to use VBM APIs and modules inside of some larger framework, it would be very easy to discard the VBM Core and replace it with whatever would be desired. Examples of programs which are commonly used in place of the normal VBM Core are the suite of test programs. One simply plugs a given module into a tester, instead of the normal VBM Core program, and the tester exercises the given interface.

7.2.3 Bifurcation Visualization API

The bifurcation visualization API allows the user to use various tools to interactively visualize bifurcation diagrams. The two main functions of this interface are to allow visualization of the bifurcation diagram and interactive selection of specific points. Table 7.1 shows the functions which make up this API.

Table 7.1: The Bifurcation Visualization API.

`setUpdateFunc`	Registers a callback to use if the data is modified
`Draw`	Redraws the bifurcation diagram
`setFileData`	Sends a copy of the current bifurcation diagram object
`setPointer`	Sends the list of currently defined pointer objects

The `setFileData` and `setPointer` methods allow the sending of bifurcation diagram and pointer objects to the module. It is important that these two data structures be set independently for efficiency. The bifurcation diagram object may be large and require a nontrivial amount of computation each time it is reset. The pointer objects, on the other hand, are normally small and change frequently.

The bifurcation visualization API is the one bi-directional API in VBM. Through this API the user can move the markers on the bifurcation diagram to designate new points of interest. This functionality is implemented with the `setUpdateFunc` method. This method allows the bifurcation diagram module to register a callback which it may use to inform the VBM Core when it modifies either the bifurcation manifold object or the pointer object.

The `Draw` method informs the module that it should update its output based upon any new data that has been given to it. The `Draw` method seems to be redundant at first glance, since the module knows that the data is updated whenever `setFileData` or `setPointer` are called, but it is needed for efficiency. The `Draw` method can be very expensive in graphical bifurcation visualization modules, so its functionality was extracted from the `setFileData` and `setPointer` routines so that it can be called only when necessary. For example, if both the bifurcation manifold object and the pointer object need to be updated, the `Draw` function can be called only once. If its functionality was part of `setFileData` and `setPointer` it would have been called twice.

7.2.4 Data Probe API

The data probe API is designed to allow the visualization of solutions of a problem of the form of equation (2.1) in greater detail than is possible in a bifurcation diagram. As equation (2.1) is very general, there are a wide range of visualization tools and methods that can be applied for specific problems. This interface is specifically designed to make almost no assumptions about how this visualization will occur. Basically, this interface transmits to the data probe module the numeric data of a points of the bifurcation diagram along with all of its corresponding tags. The data probe module is then free to interpret this data as it wishes. The data probe API is shown in Figure 7.2.

Table 7.2: The Data Probe API.

`setWhichPointer`	Sets the pointer that this data probe should follow
`setFileData`	Sends a copy of the current bifurcation manifold object
`setPointer`	Sends the list of currently defined pointer objects
`Draw`	Redraws the solution in the Data Probe

Again, as in the bifurcation visualization API, the `setFileData` and `setPointer` functions are straight forward. The one point to emphasize is that the `setFileData` routine sends the data probe all of the data for the bifurcation diagram, not just the single point defined by the `setWhichPointer` routine. The reason for this is to give the data probe freedom in the visualization that it implements. For example, all of the data points in the bifurcation diagram may not have the proper tags for a given data probe to function. If the data probe is given access to the total bifurcation diagram, it can perhaps choose a nearby point which does have the proper tags and visualize that point instead. This functionality is another example of the need to separate the visualization and computation domains. Perhaps the data needed by the data probe is large and the computational discretization is fine. In such a case, a large amount of data would be required if every point of the bifurcation diagram was to be valid for this data probe. Given the above described freedom, we can store the large data sets at only a small number of the points.

The `setWhichPointer` routine is, as discussed above, only a recommendation. There is no guarantee as to how a given module will interpret the arguments of this method, but well designed modules should give feedback if they stray too far from the point specified in this method.

7.2.5 Compute Engine API

By far the most complex API in VBM is the compute engine API. This API is our attempt to encapsulate the interactivity required to steer continuation algorithms, in a very general sense. It consists not only of a functional API, but also a set of states. Certain functions are only valid in certain states, and functions can cause the state of the computation to change. The interface methods are shown in Table 7.3, and the states are shown in Figure 7.2. This figure also shows how the methods change the state.

Table 7.3: The Compute Engine API.

`powerup`	Creates the compute engine
`start`	Starts a new continuation
`extend`	Extends an already started continuation
`reset`	Stops the current continuation and prepares to start a new one
`setup`	Modifies the computational parameters
`quit`	Destroys the computation and frees its resources

We will begin by describing the states shown in Figure 7.2. The first state, namely the *Off* state, is simply when no work has yet been done in the process of creating the compute module. In general, its code has not yet been fully compiled. The *Idle* state is when enough is known about the problem to do some preliminary setup. For example, the equations of interest have been made available, so they may be compiled into an executable. The module achieves the *Active* state when the continuation problem has been fully defined. A starting point has been chosen and the algebraic non-linear equations are fixed (e.g. a discretization has been chosen). The final state, *Sending Data*, is when the compute engine is actually running and producing data.

The state transitions occur based upon the methods in the API. The `powerup` method causes the transition from Off to Idle. It takes as arguments one string, three streams, and one array of strings (see Table 7.4). When we say "stream" we mean any arbitrary data structure which may be read sequentially (e.g. string, files, etc.). The most important arguments are the three streams which we call the *defaults* stream, the *problem* stream, and the *data* stream. The defaults stream contains data which might be used for several different problems, such as discretization details. The problem stream defines the current problem. The data stream gives access to the current bifurcation diagram object. We should emphasize at this point that the interpretation of these streams is up to the compute engine; an example will be given in Section 7.3.4. The above division is purely a convenience (i.e. if the module wished, it could have all of its data in

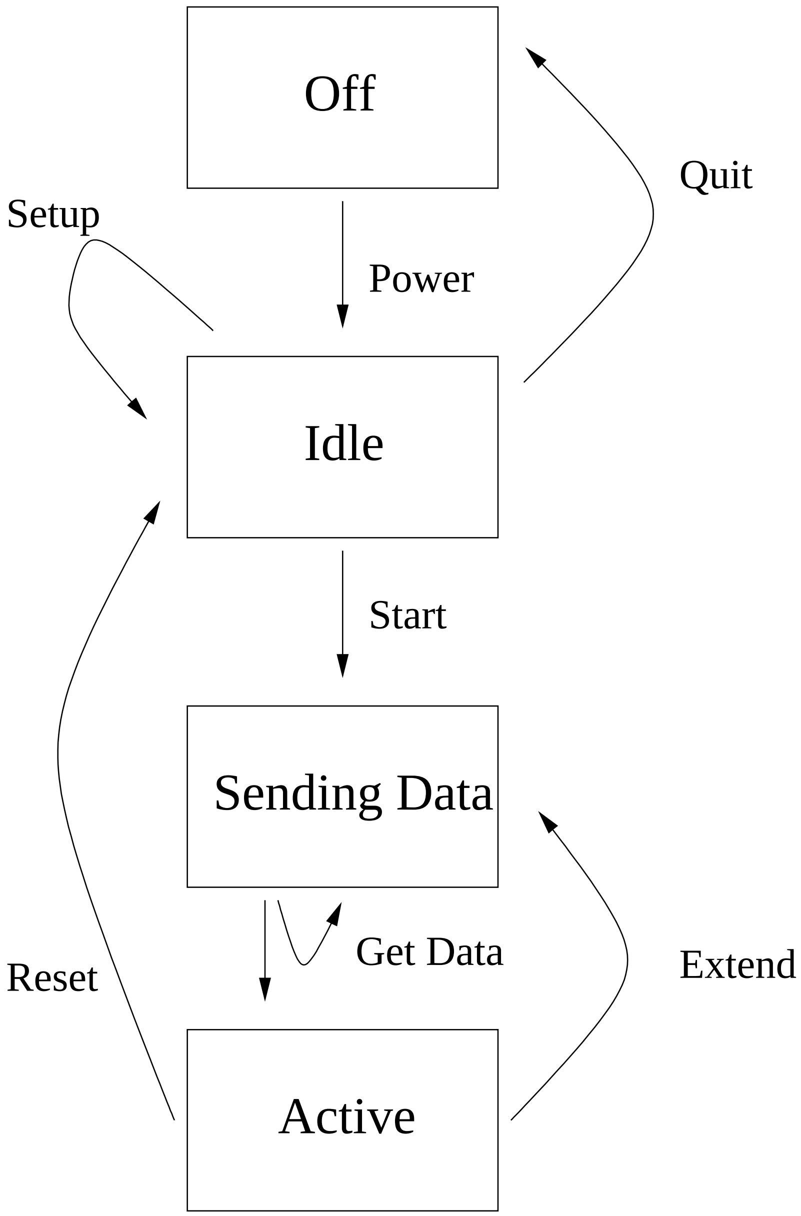

Figure 7.2: The state flow diagram for VBM compute engines. The Off state is simply when no work has yet been done in the process of creating the compute module. The Idle state is when enough is known about the problem to do some preliminary setup. The Active state is when the continuation problem has been fully defined. The Sending Data state is when the compute engine is actually running and producing data.

one stream and ignore the rest). The `quit` method is very simple and returns the module to the Off state, allowing the `powerup` method to be called again.

Having the various states, as shown in Figure 7.2, is not functionally necessary, i.e. one could implement the control of continuation algorithms without separating these various pieces of functionality, as we have done here. Pragmatically, one could control continuation algorithms with only a single function call, which would have as arguments the starting points, how many steps to compute, and perhaps some setup parameters. Unfortunately, this mode of operation would be very inefficient, because we would not be able to distinguish between those computational parameters which may be easily changed and those which are expensive to change.

Table 7.4: Arguments of the `powerup` method of the compute engine API.

working_dir	A directory in which the compute engine may store temporary files. If this argument is set to "NULL" the compute engine is free to create a directory itself.
Hostnames	An array of strings, each of which is an address of a machine on which to run a compute engine. Again, if the first entry in this array is "NULL" then the compute engine is free to select the host on which it will run.
defaults	A stream which the compute engine may read to get computational defaults, e.g. discretization parameters.
problem	A stream which defines the problem which the compute engine will solve. It may be a string, data file, or an executable program.
data	A stream from which the bifurcation diagram object may be accessed, which is canonically in the format shown in Examples 1, 2, 3, 4, and 5.

The values of the problem and data streams are fixed after `powerup` is called. On the other hand, the value of the defaults stream may be modified, and this action is performed through the `setup` method. We make this distinction for efficiency. We allow the compute engine to place options which are slow to change into the problem stream and those which are fast to change into the defaults stream.

The `start` method affects the transition from the Idle state to the Active state. It takes as arguments the numeric data of the desired starting point, the tags at that point, a vector of integers which describe what parameters to continue in, and a vector of directions (see Table 7.5). The interpretation of the direction vector is up to the compute engine, but normally positive values correspond to the increase of some parameter. The `reset` function returns the module to the Off state.

Table 7.5: Arguments of the `start` method of the compute engine API.

data	An array which contains the numeric data from the bifurcation diagram object, as shown in Example 4.
tags	An array which contains all of the tags, both point-wise and global, which apply to the point in the data array. These tags and the numeric data must provide enough information for the compute engine to start a new computation.
param	An array of m integers which define what parameters the compute engine should continue in. The values which are contained in this array are compute engine dependent.
directions	An array of m integers whose sign is used to determine the direction of continuation. The compute engine is free to interpret these integers as it wishes, though canonically a positive value should imply that the appropriate parameter is increasing and a negative value should imply the parameter is decreasing.

The `extend` function allows more data to be computed as part of the current continuation. It takes two arguments, the first of which is an array of numeric values and the second of which is a string (see Table 7.6). The string defines what type of continuation the compute engine should perform. The currently defined types are *edge* and *bounding box*, which are explained in Table 7.6. The array of numerical values defines the stopping conditions for the continuation algorithm and their interpretation is defined by the type string.

We now refer back to Chapter 5 and note that MCCC was specifically designed to allow the easy construction of compute engines. For example, the “MCCC P-arc” box in Figure 7.1 contains an implementation of the class diagram shown in Figure 5.1. In some sense, MCCC may be considered as a set of finer grained APIs which exist inside the ComputeEngine API, as well as being a stand-alone software package.

Table 7.6: Arguments of the `extend` method of the compute engine API.

type	A string which defines how the stopping conditions for the continuation algorithm are implemented. For example, the "edge" type allows the values of the params array to be used as the number of steps to add to a continuation. Another example is the "bounding box" type, in which the params array are interpreted as maximum or minimum values of some parameter to compute.
params	An array of numerical values which are used as stopping conditions for the compute engine. Their interpretation is determined by the type variable.

7.2.6 Layering of Modules

One final implication of the above interface structure bears closer inspection. As shown in Figure 7.1 modules do not have to be directly connected to the VBM Core to function. Modules may be layered and interconnected in complex ways to create various visualization and computation structures. The flexibility provided by layered modules allows functionality to be segregated into specialized modules. These specialized modules may then be attached to other modules, allowing them to use the new functionality as well. For example, Figure 7.1 shows various compute engines and networking interfaces which may be used with them. The networking modules use some protocol to transmit the arguments from the VBM Core to the various compute engines, allowing the compute engines to be run on computers different than the one on which the VBM Core runs. For example, a talk was presented on VBM at the 1998 International Congress of Mathematicians. The VBM Core ran on a laptop running Linux at the conference in Berlin, Germany, while the compute engines ran on a multi-processor Sun SMP workstation in Lausanne, Switzerland.

7.3 Examples of Specific Module Implementations

The previous sections discussed VBM in very theoretical terms. In this section we will discuss some of the specific implementation details of a selection of modules which implement the above interfaces. We intend for these examples to further elucidate the above interfaces as well as to explore some of the implementation decisions we made in the design of VBM.

7.3.1 Core

The VBM Core, as well as a significant portion of all of the other modules, is written in an object oriented scripting language called Python [32]. There are several reasons for this choice of language.

First, Python has a very clean syntax and provides many powerful data structures as an inherent part of the language. Also, Python does not have explicit memory management, i.e. all memory allocation and deallocation is handled internally. These facts combine to make Python programs very easy to write and to debug.

Next, Python is object oriented. Accordingly, the above interfaces can easily be implemented in terms of inheritance. A class can be constructed which defines the functions which comprise the interface. Individual modules can then be defined to inherit from this base class and override its methods appropriately. This structure allows us to use runtime typing to check any object we are given to see which interface it implements.

The above feature is important for the full utilization of Python's next major strength, namely that it is a scripting language. It is not a compiled language, such as C or Fortran, but source code is interpreted as it is loaded [1]. This feature allows modules to be loaded easily, or even modified, at runtime. One must be careful that any module which is loaded implements the appropriate interface, but this fact is easily checked using runtime typing and the above interface hierarchy.

All of the above advantages of Python come with a cost. Python is significantly slower than compiled languages. Python ameliorates this problem by providing well-defined interfaces to other languages such as C and C++. This feature turns a weakness into a strength, by allowing us to write the major portion of VBM in Python and then access native language programs for sections involving intense numerical computation, or graphical rendering. This idea harkens back to Chapter 4 and Chapter 5, except now, instead of numerical efficiency, our main concern is access to visualization tools.

[1] To be precise, when Python reads a source file it converts it into a compiled or "byte-code" form, which is then interpreted. The byte-code is then saved in a file, with a name which is similar to that of the original file. Fortunately, this whole process is transparent to the user. If the user makes a change to the source file, Python detects this change the next time the file is loaded, and recompiles it appropriately.

7.3.2 Bifurcation Visualization

We will now describe a bifurcation visualization module as an example of how one might implement the interface in Section 7.2.3. This module has been used extensively in our work for exploring a wide range of bifurcation diagrams. An example of the visualization provided by this module is shown in Figure 7.3, which can be found at the end of this section.

This module has been implemented using a three dimensional geometry visualization library called DataViewer, described in Appendix A. DataViewer consists of a set of C++ classes developed by Paffenroth, Stone and Ahearn. DataViewer is written on top of the OpenGL API [39] and is designed to make the development of efficient special purpose visualization software easy for those who are not experts in OpenGL.

Accordingly, the graphical user interface of this module is written using Python, and we use Pythons interface to C++ to access DataViewer for the graphics rendering. In this way the majority of the code is written in a high level language, like Python, and the sections that must be fast are written in a low level language using a previously developed library.

Basically, when this module is used, four representatives from the bifurcation diagram are chosen. Three are used as the coordinates of a three dimensional object, and the fourth is used to define a color, as discussed in Chapter 6. The three dimensional object can then be rotated, translated, and scaled using the mouse or the keyboard. The color of each point is derived by taking the value of the specified representative at that point and applying a color-map. For example, in Figure 7.3 the color-map is a standard rainbow where blue corresponds to the minimum value of the selected representative and red corresponds to the maximum value of the selected representative.

As shown in Figure 7.3, the contents of the pointer objects are displayed as small spheres. The pointer objects may be interactively modified by using the mouse and clicking on the graphical representation of the bifurcation diagram. An active pointer may be chosen, and when the mouse is clicked over a point of the bifurcation diagram, the pointer object is modified to designate that point. In this way the user may explore the data set as well as select points for further computation.

One other feature of this module deserves description here. Since the module has access to the whole bifurcation diagram, not just the four representatives which it displays, it is able to perform visualization procedures which involve additional representatives. As discussed in Chapter 6, several representatives may be combined to compute a new value which is desirable for visualization, for example a new value may be computed by performing a small rotation $\sin(\theta)a + \cos(\theta)b$ using two representatives a and b. This feature is implemented in this module by allowing the user to enter a small Python program at runtime. This program has access to the entire bifurcation diagram, and may perform arbitrary calculations on it.

As can be seen, in this DataViewer based bifurcation visualization module we have combined many of the ideas discussed throughout this thesis to create a flexible and efficient visualization tool. It can be used on a wide variety of problems, but more importantly, it can easily be used with other, more specialized, modules. If at some time more specialized functionality is required this module may even be used as a template for the design of a new module.

7.3.3 Data Probe

The VBMPhysical module is an example of a data probe module and exemplifies many of the important details of the interface discussed in Section 7.2.4 and its implementation. This module has been used extensively in our work for the visualization of the rod problem shown in equation (6.1). An example of the visualization provided by this module is shown in Figure 6.3 in Section 6.2.4.

This module, just as the previous one, has been implemented in DataViewer. Accordingly, it automatically inherits all of the advantages of this library. In this case, the module provides a visualization of a single, full solution to equation (6.1). Given a bifurcation diagram object and a pointer object, this module reconstructs the full solution from a small subset of representatives, using the ideas from Chapter 6. Basically, it assumes that a subset of the representatives chosen for the bifurcation diagram are actually the values of the continuous system equation (6.1) at the point $t = 0$. We remind the reader that equation (6.1) is solved, in this case, as a boundary value problem. Once the boundary value problem has been solved, and the solution recorded at one endpoint, the rest of the solution may be reconstructed trivially using an initial value problem solver, because in this example the equations are not too stiff. One important implication of this method is that the visualization domain and computation domain can be fully decoupled. The discretization at which the solution of the boundary value problem is performed is independent of the discretization used in the initial value problem solver and the visualization.

We also note that the visualization this module provides is quite specialized. As shown in Figure 6.3 in Section 6.2.4 the solution to equation (6.1) is represented as a tube and a ribbon. This visualization metaphor is chosen because of the specific structure of these equations, which represent the statics of an elastic rod. The two basic properties of such an entity are a centerline, defined by the $\mathbf{r}$ variables and their conjugates $\mathbf{n}$, and a set of directors, defined by the $\mathbf{q}$ variables and their conjugates $\boldsymbol{\mu}$. To capture this information in a single visualization we represent the centerline by a tube and the directors by a ribbon. In some sense, we display a representation of the solution as it would appear if it were physically realizable. We are, of course, not bound by physical realizability, and may add whatever other cues we wish to the visualization, such as the ribbon. Other information may be encoded, as well, for example utilizing color of the tube and ribbon (using the color-mapping ideas from Section 7.3.2). Other cues are also possible, for example the radius of the tube, or the width of the ribbon.

We finish the discussion of this DataViewer based data probe module with a few words on the interaction between this module and the bifurcation visualization module shown in Section 7.3.2. Consider Figure 7.4, which can be found at the end of this section. This figure shows several bifurcation visualization modules and data probe modules all intercommunicating. The bifurcation visualization modules are designed to be very general and the data probe modules are very specific to the problem shown in equation (6.1), but they are still able to work together easily. The APIs discussed in this chapter are the central reason that this close coupling of modules with different purposes is possible.

7.3.4 Compute Engine

We finish this chapter with a discussion of two closely related compute engine modules, namely VBM_1D_AUTO and VBM_1D_AUTO_FORT8. Both of these modules are based on AUTO [16], a software developed by Doedel, et. al.. AUTO is a large program, written in Fortran, which implements parameter continuation methods for solving boundary value problems in ordinary differential equations. It has proven useful in our work, and we have no desire to reinvent the functionality that it provides. Our desire, then, is to take AUTO and use it within VBM with as little modification as possible.

The main hurdle that we must overcome is to make AUTO function within the compute engine API described in Section 7.2.5. Fortunately, Python makes this procedure quite straight forward. A small Python program may be written which acts as an adapter, changing AUTO's native interface into the interface defined for VBM compute engines. Basically, there are two ways in which one may proceed. First, AUTO may be called directly from Python. Second, AUTO may be run as a separate process and communicate with VBM using standard inter-process communication protocols. We have chosen the second method because, even though it is somewhat slower, it is more portable than the first method. Python can only call C or C++ directly and calling Fortran from either of these languages is notoriously difficult to do, in a portable way. We also note that, for most applications, the added cost of the inter-process communication is negligible compared to the amount of time AUTO spends performing its computations.

The difference between the VBM_1D_AUTO and VBM_1D_AUTO_FORT8 modules is in how they handle solution reconstruction and the starting of continuation computations. VBM_1D_AUTO uses the same idea as the data probe module described in Section 7.3.3. Whenever this module is used some number of representatives in the bifurcation diagram must be reserved to represent the values of the equations at some fixed s value, usually one end-point. This module then uses these values to reconstruct an initial guess, using an initial value problem solver, so that it may start its continuation procedure (as described in Chapter 2). VBM_1D_AUTO_FORT8, on the other hand, uses the tag structure to start its continuation computations. For a point of the bifurcation diagram to be a valid starting point for this compute engine, it must have the tag "AUTOFort8_file". This tag informs the compute engine of the name of a file where the full solution, as originally generated by AUTO, is stored. This solution can then be used as the starting point for further continuations. We developed two modules because the data stored by VBM_1D_AUTO is smaller than that which must be stored by VBM_1D_AUTO_FORT8, but this module only functions on a subset of problems which AUTO can solve. VBM_1D_AUTO_FORT8, on the other hand stores a significant amount of data, but works on a wider selection of problems. We also note that all three levels of parallelism discussed in Section 5.3 have been implemented using these modules within the SPWS method discussed in Section 3.3), on Sun SMP workstations, using Sun's parallelizing compilers and threads to implement the linear algebra parallelism.

One interesting feature of these two compute engines is the method they use to represent bifurcation points. When these modules reach a bifurcation point they computes two stubs, one in each of the bifurcating directions. In this way, the user has a cue as to where a bifurcation point is in the bifurcation diagram, and the user may select the stub for further computation.

All compute engines used within VBM utilize a single, coherent, interface. The actual details of the implementation are hidden, and, hence, the user does not need to learn the specifics of the control of the underlying algorithm. They only need to learn the VBM interface, and they are then able to use any algorithm which has been implemented as a compute engine. It is important that these modules are totally independent of the underlying implementation of the compute engine. Accordingly, compute engines and bifurcation visualization modules may be freely interconnected. Specialized data probes may also be used, as long as the proper data and tags are in the bifurcation diagram object. As can be seen, VBM provides a wide range of freedom in the computation and visualization of bifurcation manifolds and it makes the interaction with many different classes of problems easy.

At this point we return to Figure 1.1 from the introduction, which we have reproduced in Figure 7.5. This figure can now be discussed, in detail, with regard to the various elements that make up this snapshot of a VBM session. The window marked A is a bifurcation visualization module written in DataViewer (see Appendix A for details) which implements the Bifurcation Visualization API. The window marked B is a data probe, also implemented in DataViewer, showing the solution corresponding to the point in the bifurcation visualization window marked with the white ball. The data probe in window B implements the Data Probe API The window marked C is the main VBM control panel, which is a GUI on top of the VBM Core. The window marked D is a data probe displaying numerical data. The data probe is implemented purely in Python and implements the DataProbe API. The window marked E is a compute engine control panel, attached to the white marker. This compute engine is an implementation of the SPWS algorithm using the Compute Engine API. This implementation of the SPWS algorithm is an example of module layering. It can use any one parameter code which implements the Compute Engine API as its underlying algorithm.

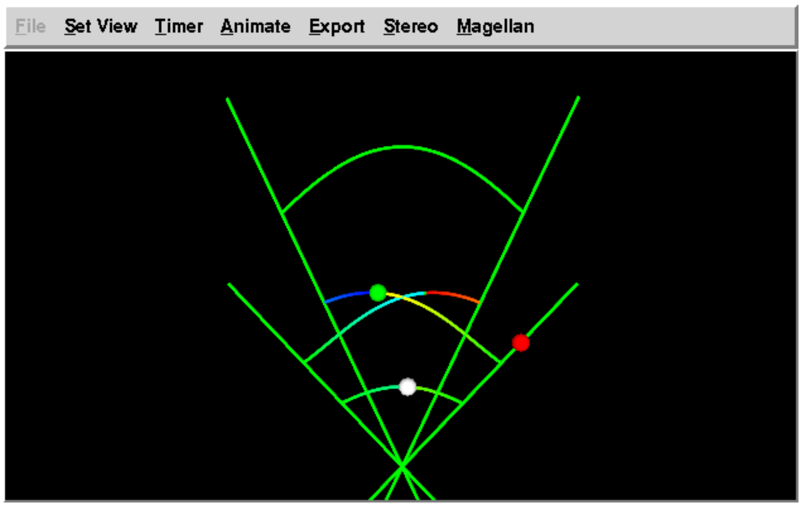

Figure 7.3: This figure depicts a bifurcation visualization object implemented using the DataViewer library. The colored lines represent the bifurcation diagram and the colored balls represent points which have been designated by pointer objects.

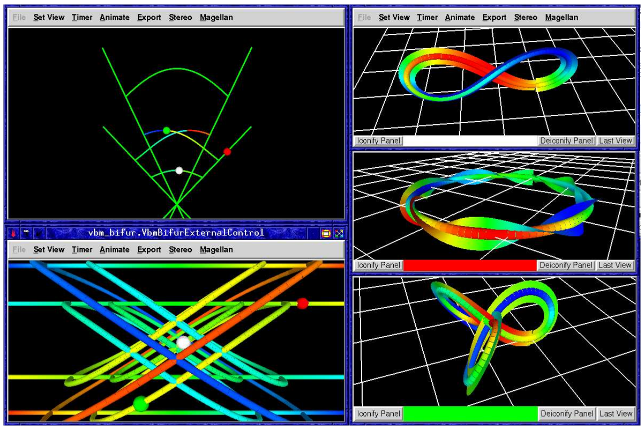

Figure 7.4: This figure shows two bifurcation visualization modules and three data probe modules. The two bifurcation visualization modules are displaying the same bifurcation diagram, except that different sets of representatives are shown in each. Each bifurcation visualization module also shows three colored spheres marking individual points. The three data probe modules each show a full solution and a colored bar at the bottom. The solution shown in each data probe corresponds to the sphere whose color matches that of the bar.

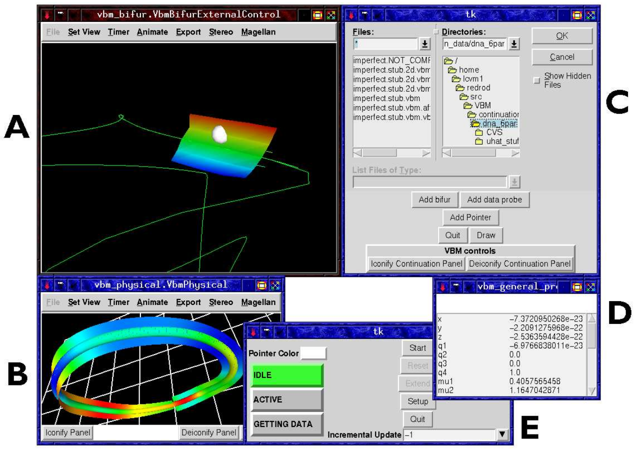

Figure 7.5: This figure shows some of the elements which make up VBM. The window marked A is a bifurcation visualization module written in DataViewer (see Appendix A for details). The window marked B is a data probe showing the solution corresponding to the point in the bifurcation visualization window marked with the white ball. The window marked C is the main VBM control panel. The window marked D is a data probe displaying numerical data and the window marked E is an interface to an compute engine, both of which are also attached to the white marker.

Chapter 8

A Menagerie of Examples

8.1 Introduction

In this chapter we will describe a set of examples to which the software and methods described in this dissertation have been applied. The first example, taken from the work of Maddocks [35], is a problem which is explicitly of the form of equation (3.1), with $n = 6$ and $m = 2$. The system models the equilibrium configurations of a heavy rigid body. The second example, taken from the work of Alexander, Maddocks, and Michalowski [1], is a six dimensional ODE BVP, which we discretize to create a system of the form of equation (3.1) with $n \approx 600$ and $m = 4$. One of the parameters in this problem is a *homotopy* parameter, and we will discuss how such parameters are used to transform known solutions of simple problems into unknown solutions of complex problems. We will present an example of how the homotopy parameter is used, as well as bifurcation diagrams with $m = 1$ and $m = 3$, which involve the other three parameters. The third example is a fourteen dimensional Hamiltonian ODE BVP which has been used to model the equilibrium configurations of small DNA mini-circles. In this example VBM (or its predecessors) was used as part of the original research, and results have been published in [36, 34, 20, 45]. After discretization we will have a system with $n \approx 1200$ and $m = 6$. We will not explicitly compute any manifolds with $m = 6$, but we will show various sub-bifurcation diagrams with $m = 1$. The final example will be the same fourteen dimensional Hamiltonian ODE BVP which was used in Section 8.4, except that we will utilize different boundary conditions to model a strut to which a force and a twist are being applied at the end. Again, in this example VBM has been used as part of the original research. After discretization we will have a system with $n \approx 1200$ and $m = 2$, and we will show a bifurcation diagram which is original to this thesis.

8.2 Equilibria of a Heavy Rigid Body

Our first example is a problem which models the steady spins of a heavy rigid body characterized by the constrained stationary points of the Hamiltonian

$$H = \sum_{i=1}^{3} \frac{1}{2} a_i m_i^2 + b_i d_i \tag{8.1}$$

subject to the constraints

$$k_1 = m_1 d_1 + m_2 d_2 + m_3 d_3, \tag{8.2}$$

and

$$k_2 = \frac{1}{2}(d_1^2 + d_2^2 + d_3^2). \tag{8.3}$$

Here there are six unknowns m_i and d_i, and six fixed constants a_i and b_i. The unknowns m_i are momentas, and the unknowns d_i are orientations. The computation of constrained stationary points is then a quadratic programming problem leading to a set of linear equations

$$\begin{aligned}
a_1 m_1 + p_1 d_1 &= 0, \\
a_2 m_2 + p_1 d_2 &= 0, \\
a_3 m_3 + p_1 d_3 &= 0, \\
p_1 m_1 + p_2 d_1 + b_1 &= 0, \\
p_1 m_2 + p_2 d_2 + b_2 &= 0, \\
p_1 m_3 + p_2 d_3 + b_3 &= 0,
\end{aligned} \tag{8.4}$$

and

$$F = H + p_1 k_1 + p_2 k_2, \tag{8.5}$$

where the p_i are Lagrange multipliers for the constraints. We note that a closed form solution is available for the linear equations (8.4) and (8.5), but for this example we will compute the bifurcation manifold with parameter continuation methods.

Post-processing is necessary for this problem because the actual quantities of interest do not explicitly appear in equation (8.4). One desired bifurcation diagram for this problem consists of the two p_i plus the Hamiltonian (8.1), and the quantities shown in equations (8.2), (8.3), and (8.5). Hence, we wish to compute the bifurcation manifold, and then to visualize a bifurcation diagram using VBM, consisting of the six quantities p_1, p_2, k_1, k_2, H and F. There are several ways in which one can construct this bifurcation diagram in VBM. For this example we have chosen to store, in the bifurcation diagram file, the m_i, d_i and p_i (e.g. see Figure 1). Since equations (8.2), (8.3), (8.1), and (8.5) are all quite small we may use a feature of the bifurcation visualization module discussed in Section 7.3.2 to compute the k_i, H and F, even though they do not explicitly appear in the bifurcation manifold. The bifurcation visualization module discussed in Section 7.3.2 allows the user to input a Python function which utilizes the quantities in the bifurcation diagram and then allows the user to plot the output of this function.

To start VBM we will assume that we have an initial section of the bifurcation diagram, as shown in Figure 8.1 (all figures appear at the end of the section). In Figure 8.2 there are two small balls, a white one and a red one, which mark two points of the bifurcation diagram. To demonstrate the Compute Engine API we will attach a compute engine based upon AUTO to the white ball and a compute engine based upon MCCC to the red ball, and we will use them to compute two new one-dimensional pieces of the bifurcation diagram, as shown in Figure 8.2. It is important to note that two compute engines, based upon two different algorithms and written in two different computer languages, have been able to contribute to the construction of a single bifurcation diagram. AUTO was implemented before VBM was created, yet it is able to communicate with VBM, and the visualization tools in VBM may be used to visualize its output.

As can also be seen in Figure 8.2, both compute engines have found a fold. We now move the white marker onto the fold and attach to it a third compute engine, namely a Matlab implementation of Rheinboldt's Moving Frame algorithm (see Section 3.4.1). This algorithm may be used to compute a two dimensional section of the bifurcation diagram, and the result of using it from within VBM is shown in Figure 8.3.

Figure 8.3 now contains pieces of the bifurcation diagram computed using three different algorithms in three different languages. To see what type of information can be garnered from such computations and visualizations we refer to Figure 8.4 and Figure 8.5. Figure 8.4 shows a collection of sheets which make up a bifurcation diagram for equation (8.4). Each sheet is colored by stability. Figure 8.5 shows two different projections of this bifurcation diagram, from which a conjecture was derived and later proven (see the caption of Figure 8.5 for further detail).

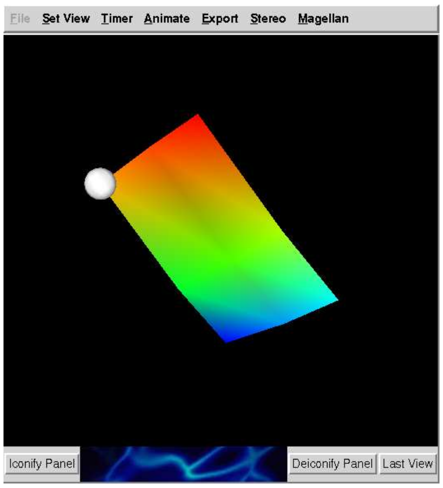

Figure 8.1: This figure shows a small two dimensional patch of the solution set of equation (8.4). The quantities being shown here are p_1, p_2 and H, with the color being derived from F. Red represents large values of F, and blue represents small values of F.

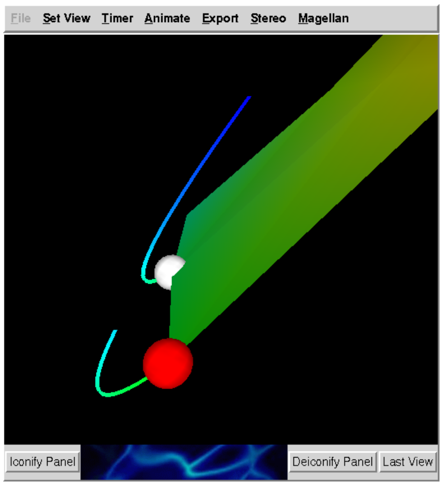

Figure 8.2: This figure shows the same two dimensional patch of the solution set of equation (8.4), as shown in Figure 8.1, with the addition of two one dimensional manifolds. The one dimensional manifold which starts at the white sphere was computed using AUTO, and the one which starts at the red sphere was computed using MCCC. Note the fold in the manifold computed by both packages.

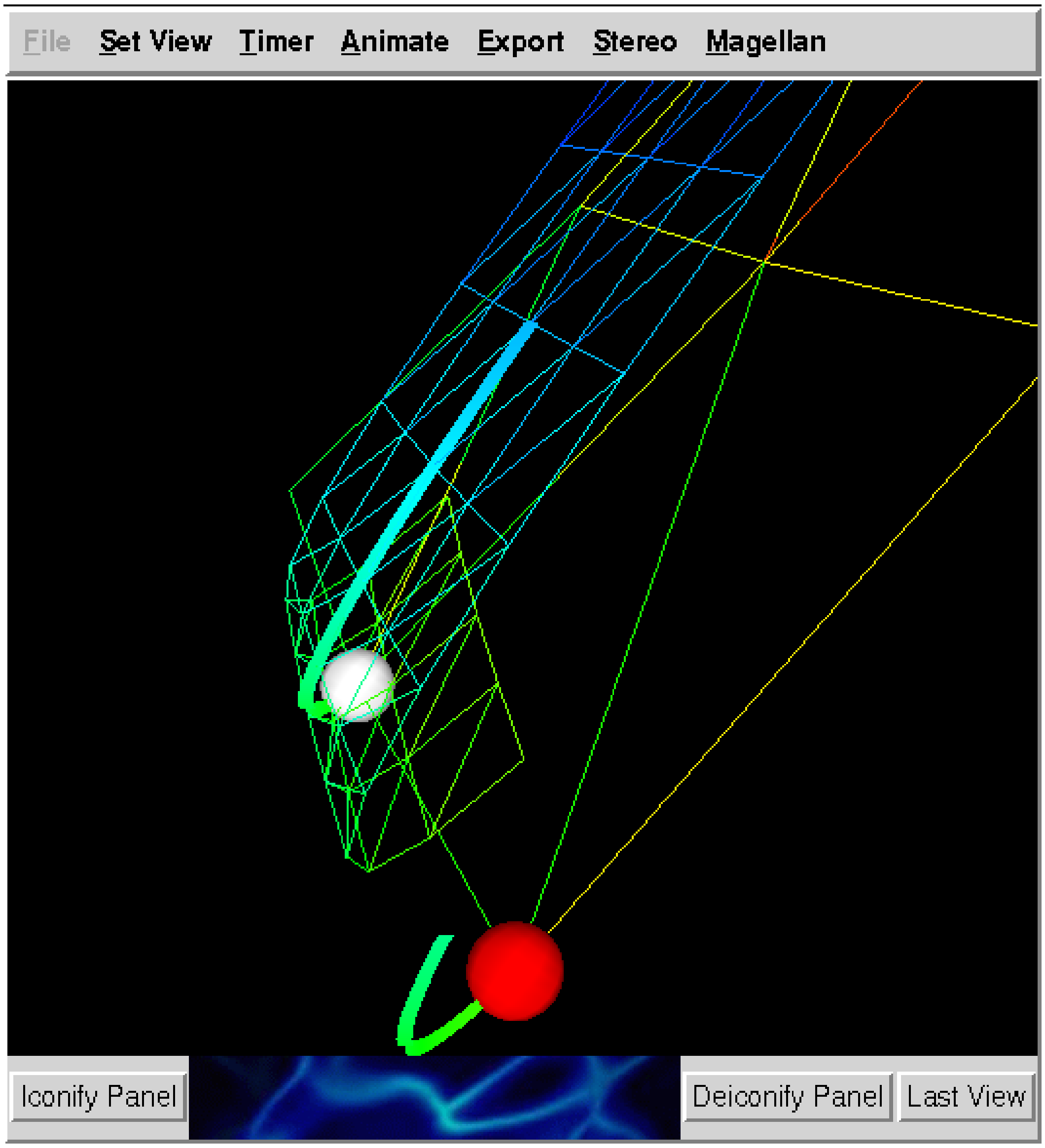

Figure 8.3: This figure shows several sections of the bifurcation diagram for the set of equation (8.4). The two dimensional manifold with the white sphere was computed using Rheinboldt's Moving Frame algorithm. We have rendered this diagram in wire-frame to show that the new surface was computed using a different step-size then the old surface.

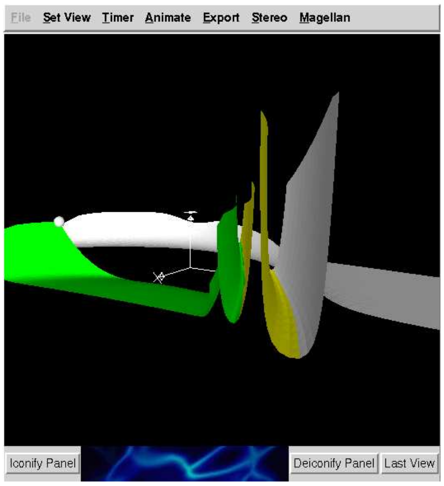

Figure 8.4: This figure shows several sheets which make up a bifurcation diagram for equation (8.4). In this visualization the bifurcation diagram is colored by stability. White is stable, and other colors represent different numbers of negative eigenvalues of a certain Hessian matrix. The surface which was computed in Figure 8.3 is part of the yellow-green surface.

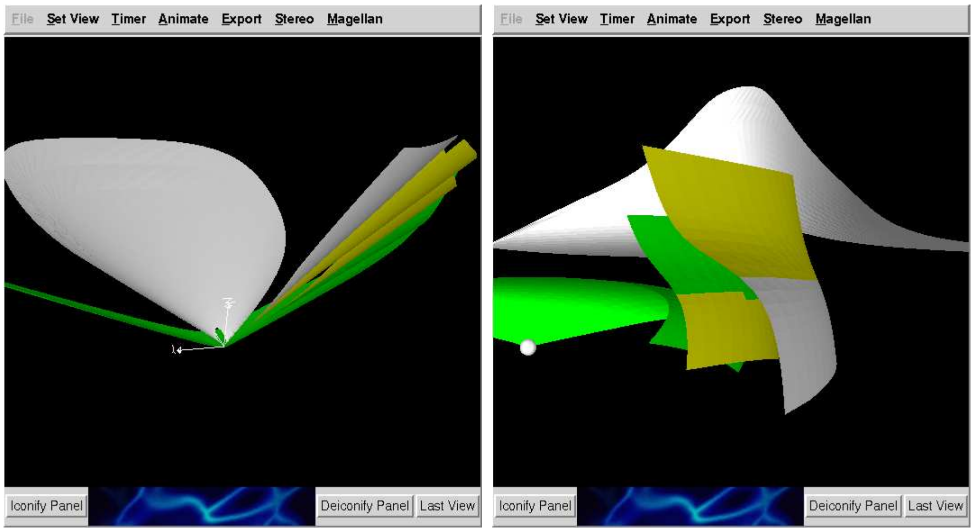

Figure 8.5: This figure shows several sheets which make up a bifurcation diagram for equation (8.4). The left image shows k_1, k_2 versus H, and the right image shows p_1, p_2 versus F. As can be seen, an inflection point in the white-green surface in the right image corresponds to the cusp in the white-green surface in the left image. This fact was only proven after being visualized.

8.3 Three Wheeled Robots

In this section we apply VBM to a problem of the computation of minimum distance paths for three wheeled robots as described fully by Alexander, Maddocks, and Michalowski [1]. Therein this problem was cast as the Hamiltonian ODE BVP

$$\begin{pmatrix} \dot{u} \\ \dot{v} \\ \dot{\xi} \\ \dot{\lambda}_1 \\ \dot{\lambda}_2 \\ \dot{\lambda}_3 \end{pmatrix} = -K \begin{pmatrix} \Lambda \cos\xi \\ \Lambda \sin\xi \\ \lambda_3 \\ 0 \\ 0 \\ \Lambda(\lambda_1 \sin\xi - \lambda_2 \cos\xi) \end{pmatrix} \tag{8.6}$$

for the basic unknowns u, v, and ξ, and controls $\lambda_1(s)$, $\lambda_2(s)$, and $\lambda_3(s)$. In equation (8.6), $K = \sqrt{\lambda_1^2(0) + \lambda_3^2(0)}$ and $\Lambda = \lambda_1 \cos\xi + \lambda_2 \sin\xi$.

To derive a system of the form of equation (3.1), equation (8.6) is discretized subject to the boundary conditions

$$\begin{aligned} u(0) &= 0, \\ u(1) &= u_f, \\ v(0) &= 0, \\ v(1) &= v_f, \\ \xi(0) &= (1-p_1)\mathtt{atan2}(v_f, u_f) + p_1(2\pi N), \\ \xi(1) &= (1-p_1)\mathtt{atan2}(v_f, u_f) + p_1(2\pi M + \xi_f). \end{aligned} \tag{8.7}$$

The four parameters u_f, v_f, ξ_f and p_1 in this problem appear in the boundary conditions. The system is obtained after applying the Pontryagin maximum principle, and any solution is critical, or stationary, amongst paths which link the initial and final positions delivered by u_f, v_f, and ξ_f. Many methods may be used to construct the discretized equations, and for this example we will utilize *orthogonal collocation* as implemented in AUTO94 [14, 15, 16].

After discretization we have a system of equations where $n \approx 600$ and $m = 4$. The three parameters u_f, v_f and ξ_f encode the desired final position of the robot, and parameter p_1 is a *homotopy parameter*. The variables u and v represent the coordinates of the center of the axle of the back two wheels, and ξ represents the angle of the turnable front wheel. In the context of numerical parameter continuation methods, a homotopy parameter is used to transform a known solution of one problem into an unknown solution of another problem. For example, consider the boundary conditions shown in equation (8.7). When $p_1 = 1$ we have the actual problem of interest,

$$\begin{aligned}
u(0) &= 0, \\
u(1) &= u_f, \\
v(0) &= 0, \\
v(1) &= v_f, \\
\xi(0) &= 2\pi N, \\
\xi(1) &= 2\pi M + \xi_f,
\end{aligned} \tag{8.8}$$

while when $p_1 = 0$ we have the simpler problem, to which explicit solutions are known,

$$\begin{aligned}
u(0) &= 0, \\
u(1) &= u_f, \\
v(0) &= 0, \\
v(1) &= v_f, \\
\xi(0) &= \texttt{atan2}(v_f, u_f), \\
\xi(1) &= \texttt{atan2}(v_f, u_f).
\end{aligned} \tag{8.9}$$

To solve such a system we begin with a known solution for the problem with $p_1 = 0$. We then perform numerical parameter continuation, in the p_1 parameter, until $p_1 = 1$, and we have a solution to the desired discretized problem.

Figure 8.6 shows a visualization of the bifurcation diagram after the homotopy parameter p_1 has been increased from 0 to 1, using a compute engine base upon AUTO (all figures appear at the end of the section). Once we have a solution for the case where $p_1 = 1$, we may use that solution as the starting point for varying the other three parameters. Figure 8.7 shows several one dimensional manifolds computed using AUTO and MCCC by varying each of the parameters u_f, v_f, and ξ_f. Because of the size of the system, the portion of the manifolds computed by MCCC used a multi-threaded version of the Powell hybrid method for solution correction, compiled using the Sun Workshop Pro parallelizing compilers and run on a four processor Sun 450. While these one dimensional portions of the bifurcation diagram are useful, we may also vary all the parameters simultaneously to compute a three dimensional bifurcation diagram. Figure 8.8 shows an example of such a three dimensional manifold computed using the SPWS method. The computational algorithms we have described can compute such a manifold quite easily, but our current graphical tools do not suffice for its effective visualization.

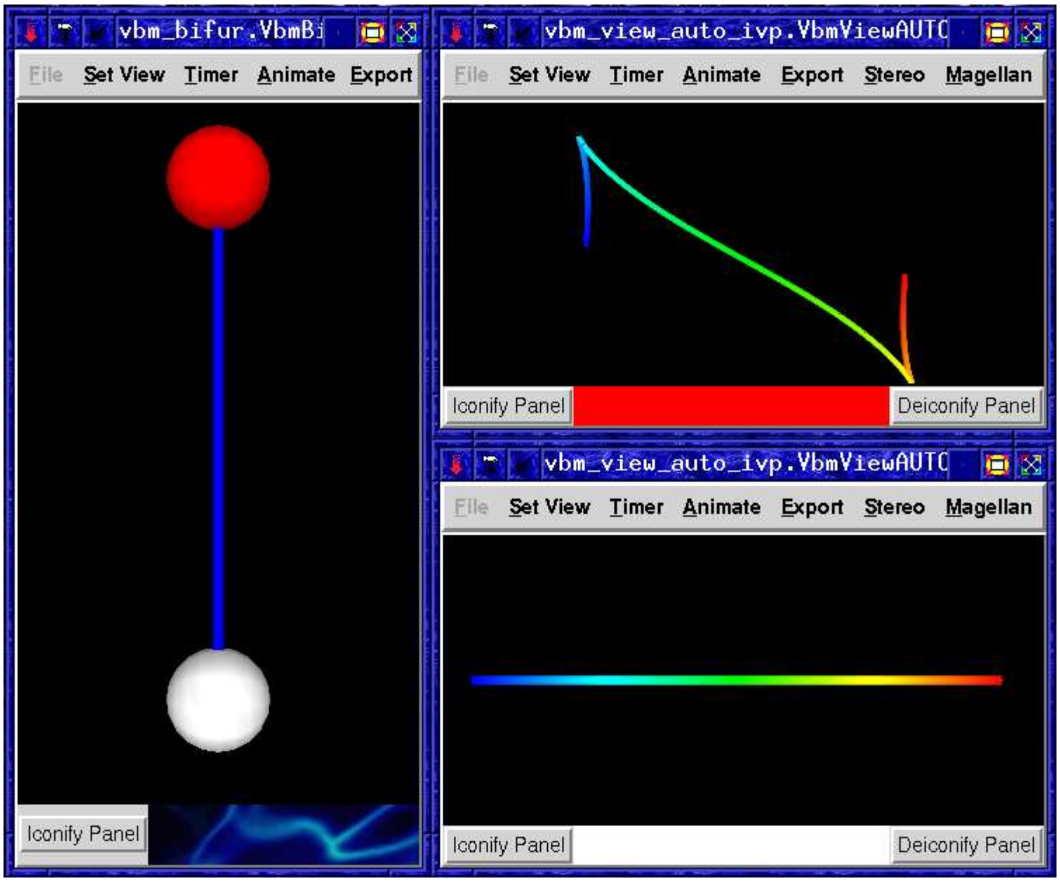

Figure 8.6: On the left this figure shows the portion of the bifurcation diagram for equation (8.6) which corresponds to the increase of the homotopy parameter p_1 from 0 to 1. p_1 is plotted on the vertical axis. The white ball marks the point at which $p_1 = 0$, and the red ball marks the point at which $p_1 = 1$. On the right are two data probes which show the full solution at the marked points. The top window shows the solution at $p_1 = 1$ and the bottom window show the solution at $p_1 = 0$. Note that the bar on the bottom of each data probe is colored the same as the corresponding marker sphere. Both data probes show u versus v, i.e. the physical path followed by a part of the robot.

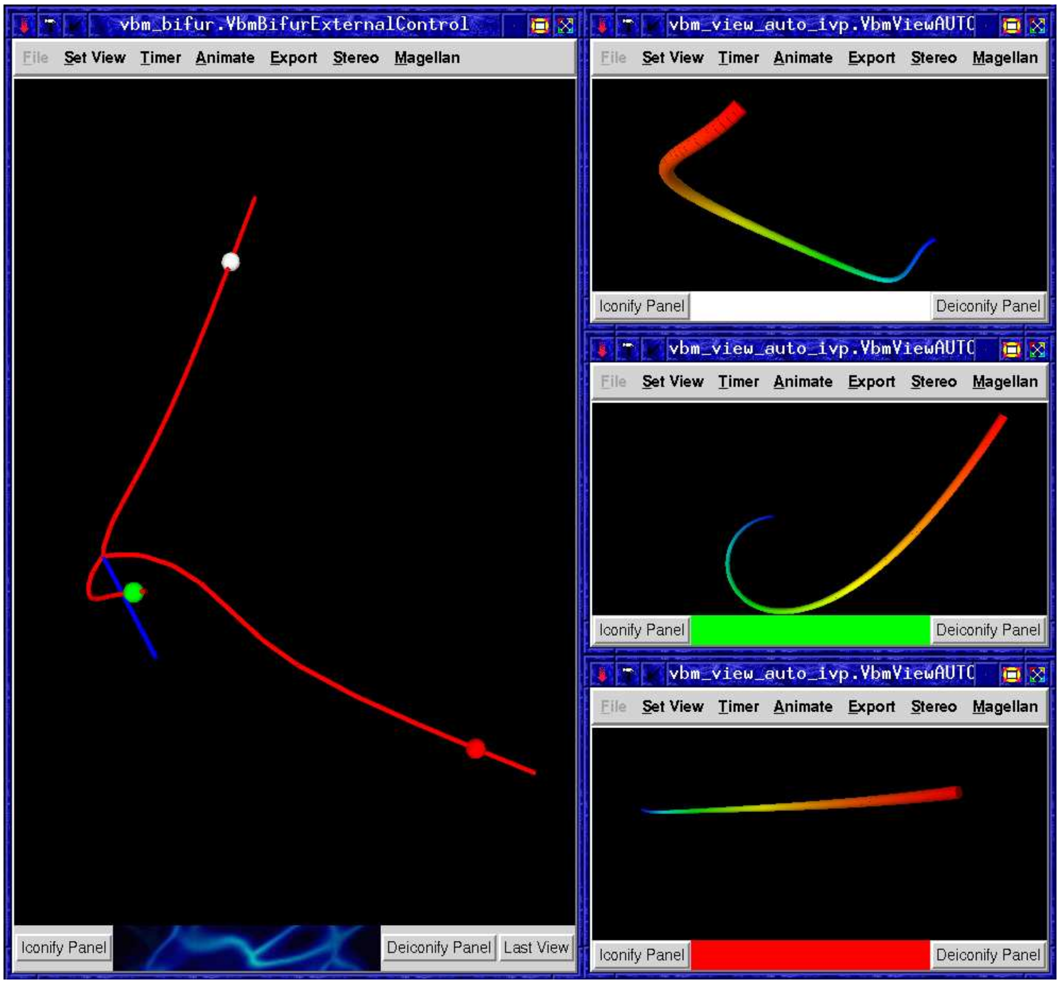

Figure 8.7: The window on the left shows several one dimensional pieces of the bifurcation diagram which have been computed using AUTO and MCCC, with $\lambda_1(0), \lambda_2(0)$ and $\lambda_3(0)$ plotted. Three points are marked with white, red, and green spheres, and the full solutions at the marked points are shown on the right. In each data probe we have plotted a three dimensional view of u, v, and ξ. The branch marked with the green ball was computed by varying ξ_f, the branch marked with the white ball was computed by varying v_f, and the branch marked with the red ball was computed by varying u_f.

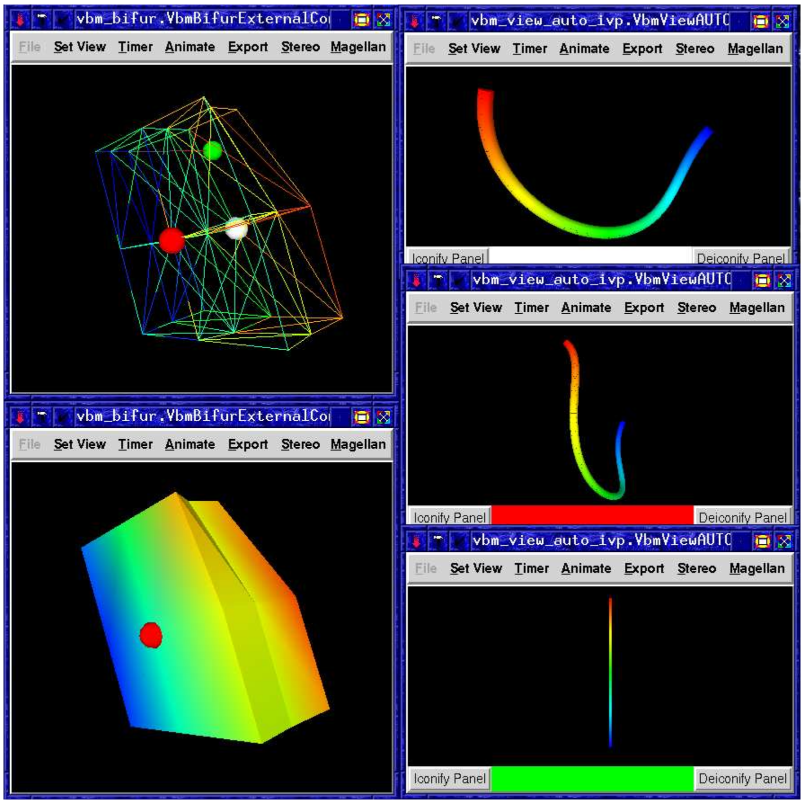

Figure 8.8: On the left this figure shows two three-dimensional bifurcation diagrams. The bottom bifurcation visualization module draws the bifurcation diagram as a solid object, and the top bifurcation visualization module draws the bifurcation diagram as a collection of line segments. Plotted in this diagram are $\lambda_1(0), \lambda_2(0)$ and $\lambda_3(0)$. As before, three points are marked with white, red, and green spheres and the full solutions at the marked points are shown on the right. In each data probe we have plotted a three dimensional view of u, v, and ξ. The bifurcation diagram in this example was computed using the SPWS method. The computational algorithms we have described can compute such a manifold quite easily, but our current graphical tools do not suffice for an effective visualization.

8.4 Elastic Rod With Ring Boundary Conditions

In this section we apply VBM to a problem of the computation of equilibrium configurations of an elastic rod. This problem has been cast [13] as a Hamiltonian ODE BVP, as shown in equation (8.10). These equations have been used extensively as a model of small DNA mini-circles in [36] and other articles.

$$\begin{pmatrix} \dot{\mathbf{r}} \\ \dot{\mathbf{q}} \\ \dot{\mathbf{n}} \\ \dot{\boldsymbol{\mu}} \end{pmatrix} = \begin{pmatrix} \mathbf{d}_3(\mathbf{q}) \\ \sum_{i=3}^{3} \left(\frac{\boldsymbol{\mu}^T B_i \mathbf{q}}{2K_i} + \hat{u}_i \right) \frac{1}{2} B_i \mathbf{q} \\ \mathbf{0} \\ \sum_{i=3}^{3} \left(\frac{\boldsymbol{\mu}^T B_i \mathbf{q}}{2K_i} + \hat{u}_i \right) \frac{1}{2} B_i \boldsymbol{\mu} - \left(\frac{\partial \mathbf{d}_3(\mathbf{q})}{\partial \mathbf{q}} \right)^T \mathbf{n} \end{pmatrix} \tag{8.10}$$

Here $\mathbf{r}(s) \in \mathbb{R}^3$, $\mathbf{q}(s) \in \mathbb{R}^4$, $\mathbf{n}(s) \in \mathbb{R}^3$, and $\boldsymbol{\mu}(s) \in \mathbb{R}^4$ for a fourteen dimensional system. In this equation $\mathbf{d}_i$ is a quadratic function of $\mathbf{q}$, B_i is a constant matrix, $u_i = \frac{2}{\|\mathbf{q}\|^2}\mathbf{q}' \cdot B_i\mathbf{q}$, K_i is a bending stiffness, and $\hat{u}_i$ is the initial configuration of the rod. We shall not further discuss the details of these equations here, since a full discussion may be found in [13]. For our purposes it suffices to say that to derive a system of the form of equation (3.1), equation (8.10) is discretized, and then some set of boundary conditions are applied. We will consider two cases.

The first case of equation (8.10) that we will consider is the so called *perfect problem* in which the rod is assumed to be initially straight and isotropic (i.e. the rod bends equally in all directions). For this problem we use the boundary conditions shown in equation (8.11).

$$\begin{array}{lll} r_1(0) = 0, & r_2(0) = 0, & r_3(0) = 0, \\ q_1(0) = 0, & q_2(0) = 0, & q_3(0) = 0, \\ \mu_4(0) = 0, & r_1(1) = 0, & r_2(1) = 0, \\ r_3(1) = 0, & q_3(1) = \sin p_1, & q_4(1) = \cos p_1, \\ n_2(0) = 0, & n_3(0) = 0. & \end{array} \tag{8.11}$$

A schematic of these boundary conditions may be seen in Figure 8.9 (all figures appear at the end of the section).

The bifurcation diagram for the perfect problem is particularly interesting in that only one of the values from equation (8.10) is actually included in it. The bifurcation diagram for the perfect problem includes

- $n_1(0)$ from equation (8.10),
- three moments (two of bending and one of twisting),
- the elastic energy of the rod configuration,
- the writhe[1] of the rod configuration.

[1] Writhe is an integral over a space curve which is related to non-planarity [36].

Figure 8.10 and Figure 8.11 show two three dimensional projections of a bifurcation diagram for the perfect problem. Figure 8.10 shows an orthogonal view of the bifurcation diagram, while Figure 8.11 shows a skew view of the bifurcation diagram. Figure 8.12 show various data probes that have been added to the bifurcation diagrams shown in Figure 8.10. Figure 8.12 is important in that it displays several of the features that lie at the heart of VBM. First, the values stored in the bifurcation diagram are almost totally distinct from those which comprise a solution of equation (8.10). Hence, we have achieved almost total decoupling of the visualization and computation domains. Second, several projections of the bifurcation manifold are being visualized simultaneously. Spheres have been used to mark several points of the bifurcation diagram, and a data probe has been attached to each marker. Even though the bifurcation visualization module is general, the data probe modules, which are specific to equation (8.10), may easily be used with it. Finally, the data probes use many of the ideas from Chapter 7. For example, the full solution to equation (8.10) is not stored at any point of the bifurcation diagram. Instead, the data probe module reconstructs the solution to equation (8.10) by using the three moments, the value of $n_1(0)$ and an IVP solver.

The second case of equation (8.10) that we will consider is the so-called *imperfect problem*, which uses the boundary conditions shown in equation (8.12). The imperfect case differs from the perfect case in that the rod is neither initially straight nor isotropic.

$$\begin{array}{rrr} r_1(0) = 0, & r_2(0) = 0, & r_3(0) = 0, \\ q_1(0) = 0, & q_2(0) = 0, & q_3(0) = 0, \\ r_1(1) = 0, & r_2(1) = 0, & r_3(1) = 0, \\ \mu_4(0) = 0, & q_1(1) = 0, & q_2(1) = 0, \\ q_3(1) = \sin p_1, & q_4(1) = \cos p_1. & \end{array} \tag{8.12}$$

For the perfect problem we emphasized the visualization aspects of VBM. For the imperfect problem we will emphasize the computational aspects of VBM. In the imperfect problem we have $m = 6$. We will not compute a bifurcation diagram which uses all six parameters, but instead we will concentrate on two parameters. We do not wish to compute the whole bifurcation diagram, but subsets which form isolas (i.e. a closed curve) as p_1 is varied. The exact isola that is formed is dependent on another parameter, namely the ratio of twisting to bending stiffness which we will denote by K. Our goal is to construct a bifurcation diagram which consists of several isolas for different values of K.

Figure 8.13 shows an isola for a single value of K. Figure 8.14 shows the isola from Figure 8.13, but with the addition of small stubs which have been computed for two new values of K. Figure 8.15 shows the three completed isolas along with data probes showing full solutions.

The data set shown in Figures 8.13, 8.14, and 8.15 is taken from [20], where the data set is derived from calculations on a set of parameters which models a small segment of DNA. Figure 8.16 shows a visualization of the data set which includes aspects which are of special interest in the DNA problem.

We close this section with a brief aside by examining a visualization for a problem which is not strictly of the form of equation (8.10), but is a related two-parameter problem for certain elastic rod equilibrium problems [28] (see Figure 8.17). For this example the desired visualization is a surface of revolution whose cross section is the solution to the discretized equations. This data probe which implements the visualization of a surface of revolution performs significant computation in addition to that required to reconstruct the solution to the discretized equations. The requirement of such additional computation exemplifies the need to separate visualization and computation domains.

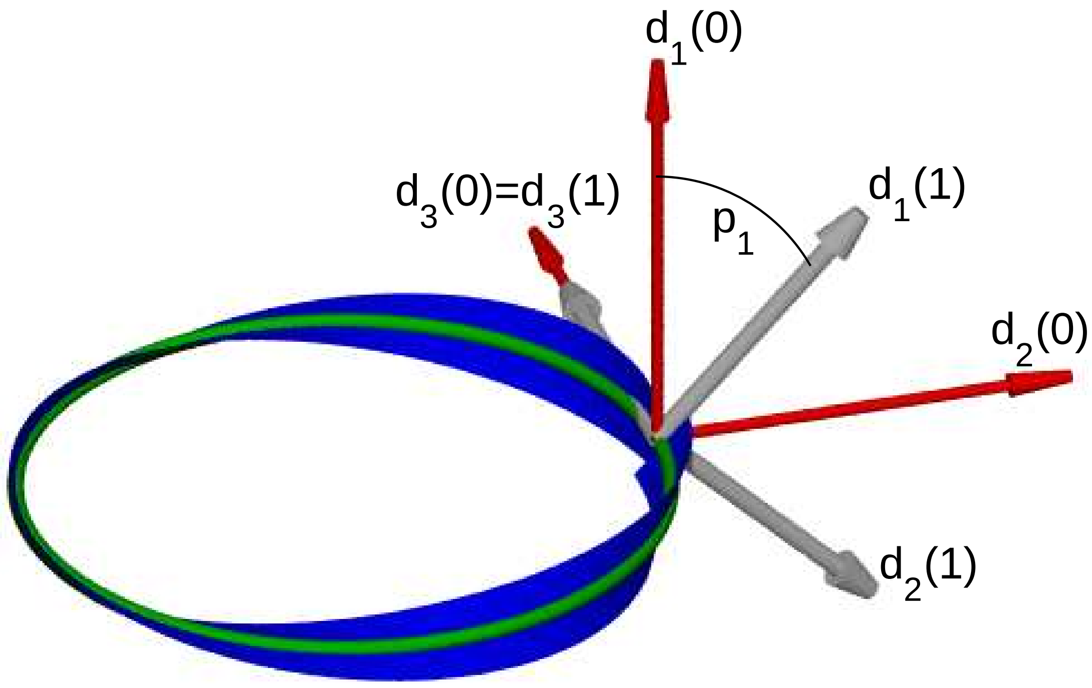

Figure 8.9: This figure shows a schematic of the boundary conditions shown in equation (8.11) and equation (8.12). These boundary conditions require the rod to assume a ring shape, and the parameter p_1 is the imposed angle between the director frames at the two endpoints.

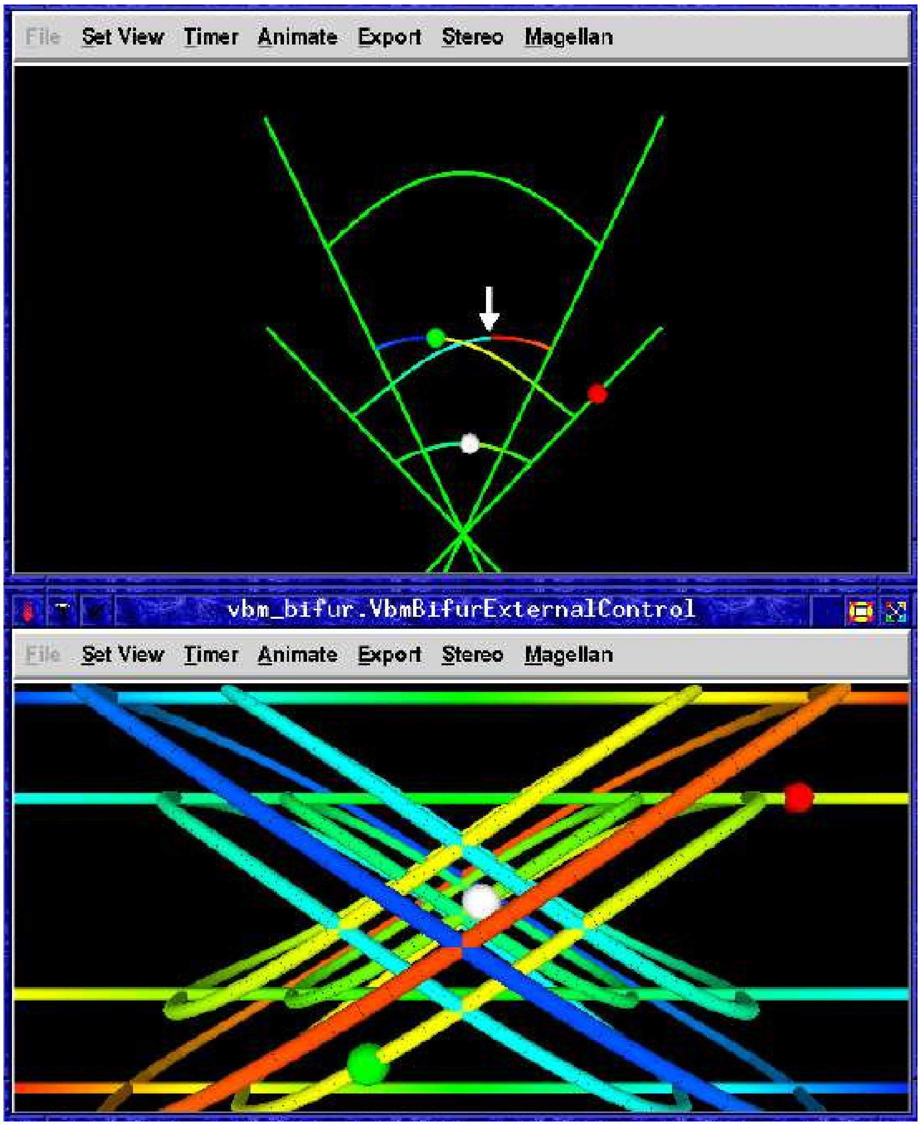

Figure 8.10: This figure shows two projections of a bifurcation diagram for the perfect problem. The projection on the bottom is created by plotting the three moments and coloring by $n_1(0)$. The projection on the top is created by plotting $n_1(0)$, the elastic energy, and the twisting moment, and coloring by writhe. One feature which we wish to emphasize is the color discontinuity highlighted by the arrow in the top bifurcation diagram. The rapid change from blue to red shows that the writhe is discontinuous at that point of the bifurcation diagram which is a cue for special behavior, in this case self-intersection as shown in Figure 8.12. We also wish to emphasize that we have chosen to display two different projections of the same bifurcation diagram, and that the spheres mark the same solution points in both visualizations.

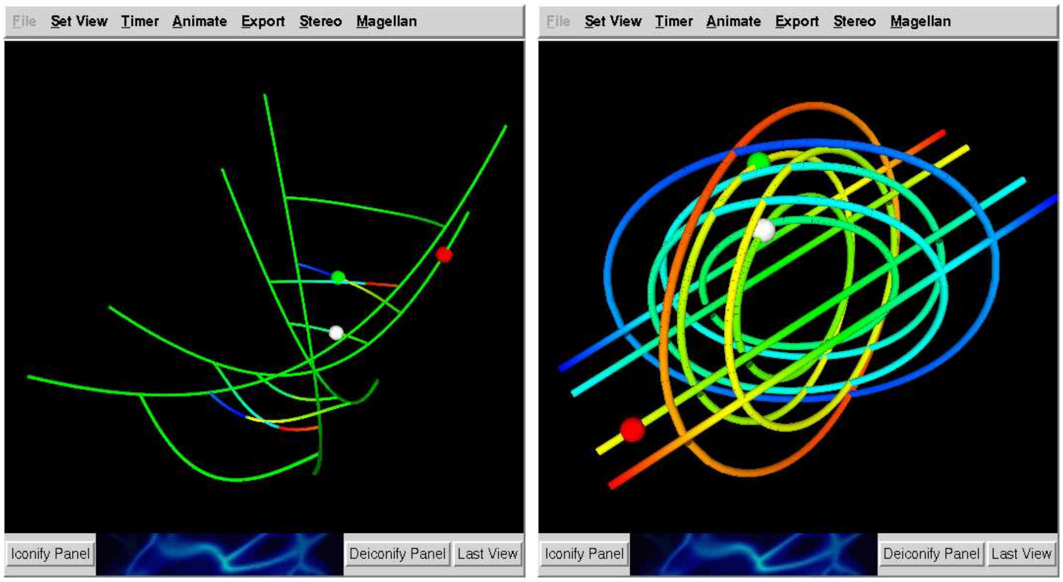

Figure 8.11: This figure shows the same two three-dimensional projections of the bifurcation diagram as shown in Figure 8.10, except that they are shown in a skew perspective instead of an orthogonal perspective.

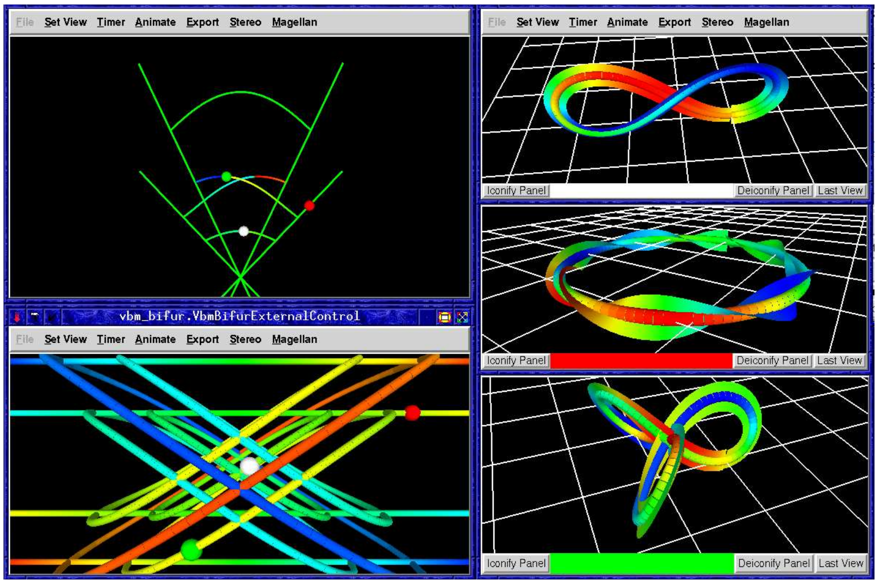

Figure 8.12: On the left this figure shows the same two projections of a bifurcation diagram for the perfect problem as shown in Figure 8.10. On the right are three data probes which show a representation of the solution to equation (8.10) at each marked point. As can be seen in the bottom Data Probe, the green sphere marks the point of the bifurcation diagram at which the solution has a self-intersection.

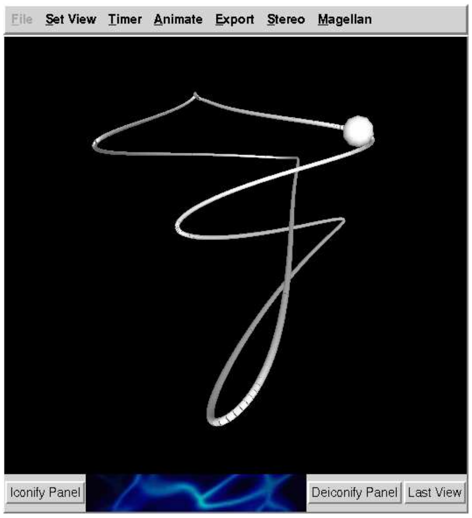

Figure 8.13: This figure shows an isola for a fixed value of K. The quantities $\mu_1(0), \mu_2(0)$ and $n_1(0)$ have been plotted in this diagram, and the color is derived from the value of K (as can be seen, it is constant).

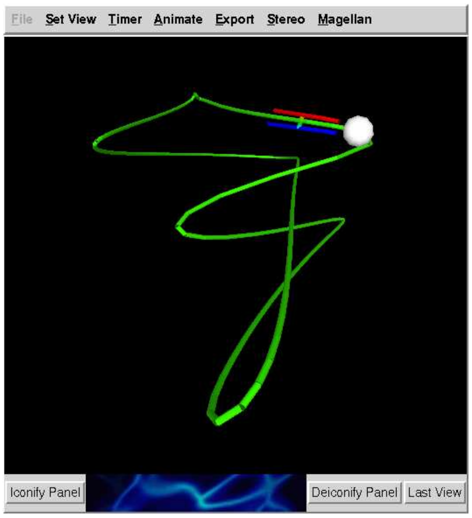

Figure 8.14: This figure shows an isola for a fixed value of K with the addition of two "T" shaped stubs which have been computed for two new values of K. Again $\mu_1(0), \mu_2(0)$ and $n_1(0)$ have been plotted, and the color is derived from the value of K. The short vertical lines where the color varies show parts of the bifurcation diagram which have been computed by fixing p_1 and varying K. The short horizontal lines where the color is solid show parts of the bifurcation diagram which have been computed by fixing K and varying p_1.

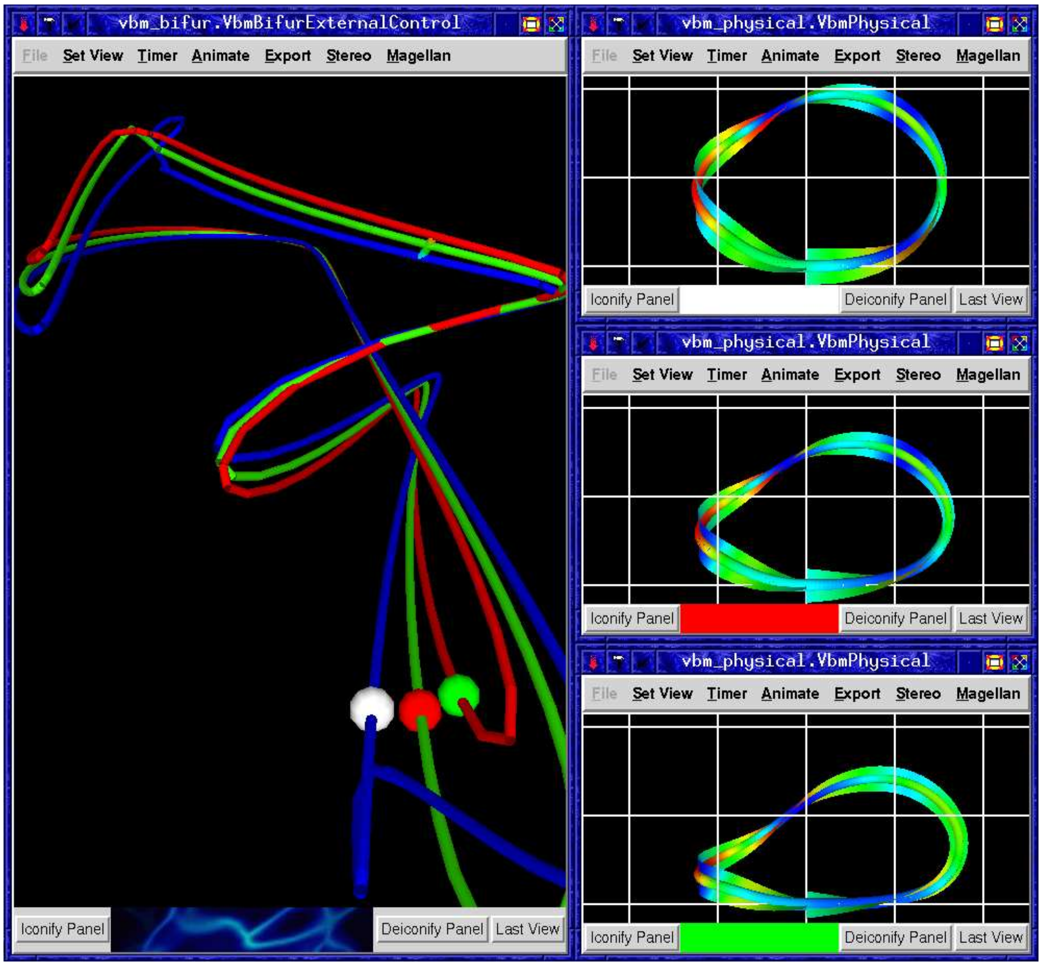

Figure 8.15: On the left this figure shows three isolas for three different values of K, along with three marker spheres, one on each isola. On the right are three data probes which show the full solution at each of the marked points.

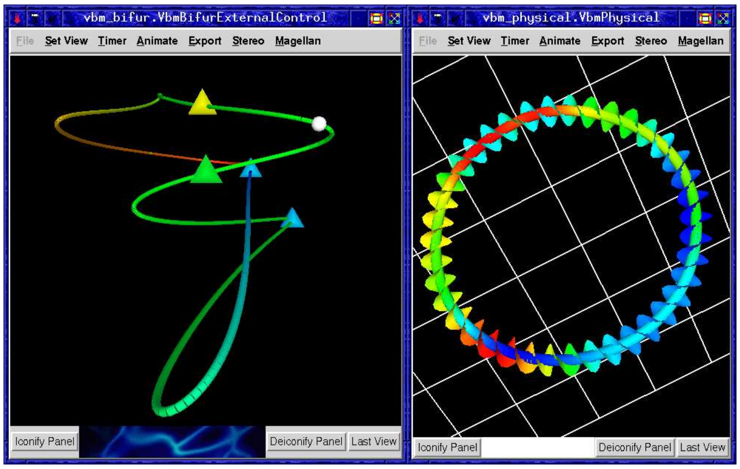

Figure 8.16: These figures shows several visualization tools which are useful for the DNA problem. The data probe shows a ribbon which tracks the backbone of the DNA, which is colored by bending energy and twisting energy. The bifurcation diagram is colored by stability (those solutions in blue are stable while those in red are not). Each colored triangle marks a point at which the DNA double helix is closed, where the solution is of particular interest. The triangles are colored by a quantity called *Link*.

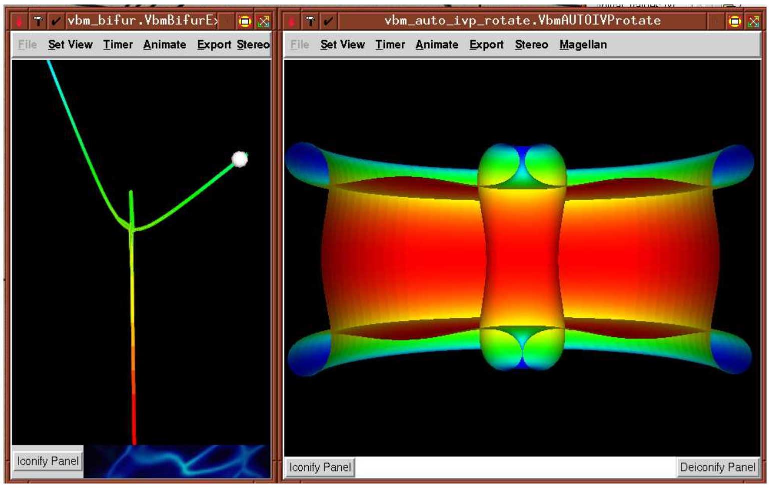

Figure 8.17: This figure shows a specialized data probe whose visualization is far removed from the computation of the bifurcation diagram. The solution to the discretized equations is the four lobed cross section, shown in this figure, which we wish to visualize as a surface of revolution. In such a context the data probe must perform significant computation, beyond that required to reconstruct the solution to the discretized equations.

8.5 Elastic Rod With Strut Boundary Conditions

In this section we apply VBM to a problem of the computation of equilibrium configurations of the same elastic rod equation (8.10), except that we now utilize the different set of boundary conditions.

$$\begin{array}{ccc} r_1(0) = 0, & r_2(0) = 0, & r_3(0) = 0, \\ r_1(1) = 0, & r_2(1) = 0, & q_2(0) = 0, \\ q_3(0) = 0, & q_1(1) = 0, & q_2(1) = 0, \\ q_3(1) = \sin(p_2), & q_4(1) = \cos(p_2), & \mu_4(0) = 0, \\ \multicolumn{3}{c}{\mu_1(0)q_4(0) - \mu_2(0)q_3(0) + \mu_3(0)q_2(0) - \mu_4(0)q_1(0) = 0,} \\ \multicolumn{3}{c}{n_1(1) = -p_1.} \end{array} \tag{8.13}$$

These boundary conditions represent a rod, both of whose endpoints are constrained to lie on the line $(x, y) = \mathbf{0}$. A schematic of these boundary conditions may be seen in Figure 8.18 (all figures appear at the end of the section).

The boundary conditions shown in equation (8.13) make the problem shown in equation (8.10) an inherently two parameter problem. Accordingly, we utilize VBM to compute examples of two dimensional manifolds. Figure 8.19 shows a small collection of one dimensional manifolds which we will use as the starting point for our calculations. Figure 8.20 shows a larger set of one dimensional manifolds which have been computed to provide an outline of the general shape and complexity of the underlying two dimensional manifold. We note that the area of the bifurcation diagram near the white sphere appears especially complex, so in Figures 8.21 and 8.22 we have performed two separate computations using the two parameter version of the SPWS algorithm. One SPWS computation has been used to generate the large portion of the bifurcation diagram, at a coarse discretization, while the other SPWS computation has been used to generate the small, more complex, portion at a slightly finer discretization. In Figure 8.23, we have added a second two-dimensional manifold which contains a line of bifurcation points where it intersects the first two dimensional manifold. Figure 8.24 adds a one parameter continuation of a branch which bifurcates from the p_1 branch in Figure 8.19, and Figure 8.25 shows a continuation of a secondary bifurcation on the first bifurcating branch. The continuation calculations lead one to believe that the secondary bifurcation branch intersects the two dimensional sheet computed in Figure 8.21, and Figure 8.26 demonstrated that a two dimensional sheet computed by starting on the secondary bifurcation branch coincides with the sheet computed in Figure 8.21.

We now begin a new set of computations to further explore the bifurcation point shown in Figure 8.23. We have removed the previously computed manifolds and started a new one dimensional parameter continuation at this bifurcation point. Figure 8.27 shows the secondary bifurcation point which is a result of the above calculation. We expand the secondary bifurcation point by using the SPWS algorithm to generate a two dimensional manifold starting on the bifurcating branch, as shown in Figure 8.28. We now change the projection of the bifurcation diagram, as shown in Figure 8.29, and perform a new two dimensional SPWS calculation starting on the original branch. Somewhat surprisingly, the two manifolds do not intersect in a line of bifurcation points, as occurred in Figure 8.22, but exactly overlap as shown in Figure 8.30. We change the projection again, and Figure 8.31 shows that we have computed a cusp catastrophe.

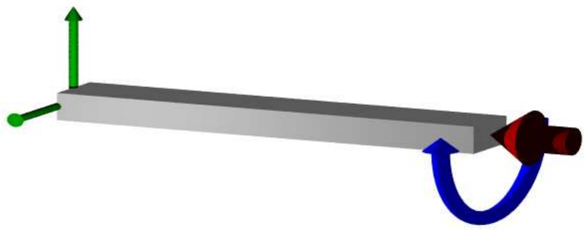

Figure 8.18: This figure shows a schematic of the boundary conditions shown in equation (8.13). The two parameters in this system are a force p_1 represented by the red arrow, and an imposed angle p_2 represented by the blue arrow.

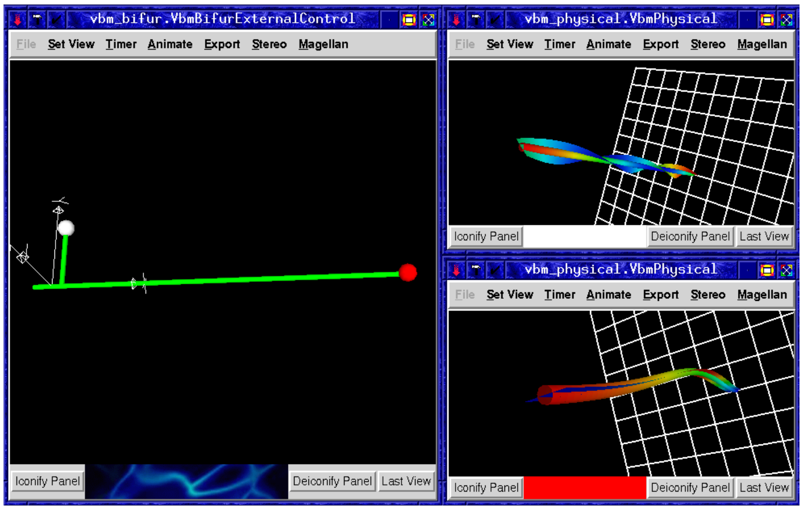

Figure 8.19: This figure shows a bifurcation diagram for equation (8.10) with the boundary conditions shown in equation (8.13). The horizontal branch with the red ball was computed by a continuation in p_1, and the vertical branch with the white ball was computed by a continuation in p_2. In this diagram we have plotted p_1, p_2, and $y(0.5)$, and the color is based on $x(0.5)$. The corresponding full solutions are shown on the right. The spatial coordinates for the full solutions are x, y, and z, the color of the tube is based on μ_1, and the color of the ribbon is based on q_4.

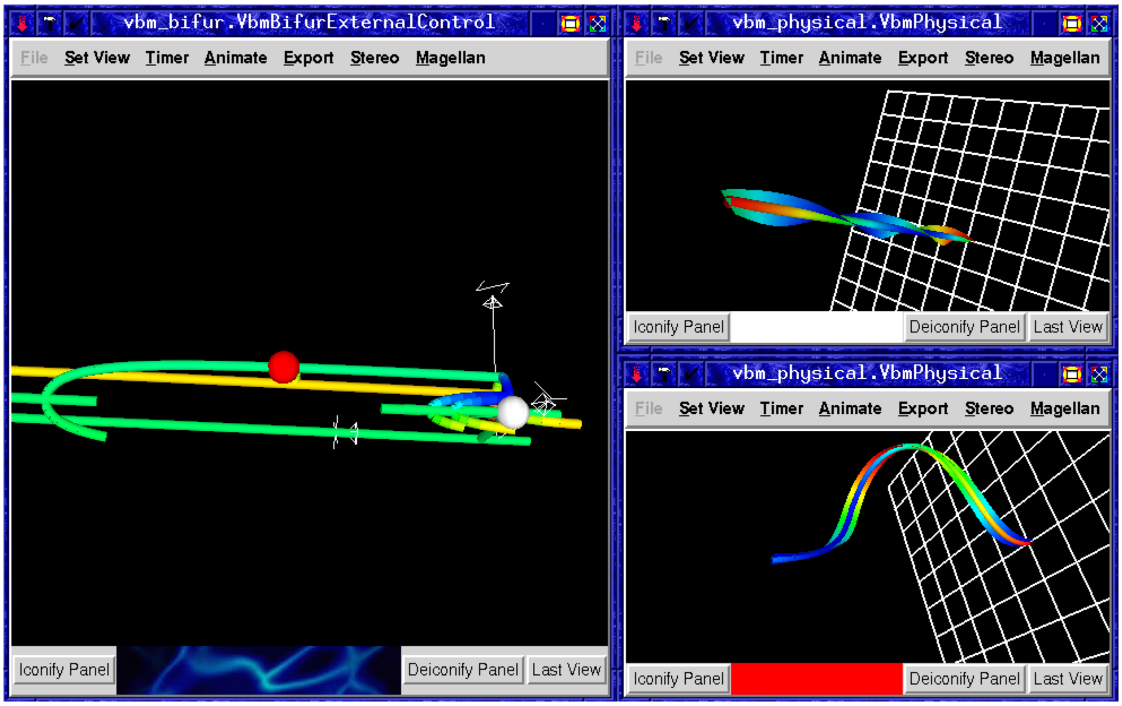

Figure 8.20: This figure shows several additional one parameter continuations. The components plotted in the bifurcation diagram and the data probes are the same as Figure 8.19. The bifurcation diagram in the area of the white sphere is quite complex, with several nearby bifurcation points and folds. The bifurcation diagram in the area of the red ball appears to be smoother and less complex. Based upon this preliminary computation we divide our two dimensional computation into two parts. The first will be applied to the smooth region and the second will be applied to the complex region.

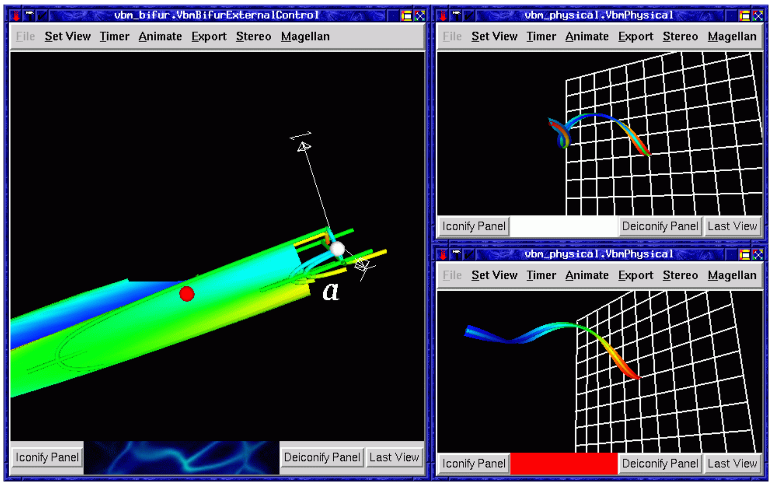

Figure 8.21: This figure shows several one parameter continuations and a two dimensional manifold. The components plotted in the bifurcation diagram and the data probes are the same as Figure 8.19. The computation of the two dimensional manifold was initiated at the point marked by the red sphere. The "a" marks a complex region of the manifold, and it will appear in some of the following images as a cue to the orientation of the bifurcation diagram.

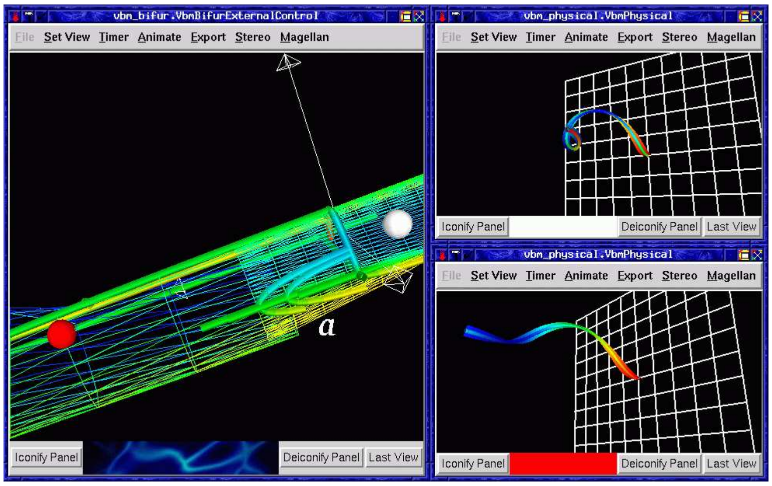

Figure 8.22: This figure shows several one parameter continuations and multiple two-dimensional manifolds. The components plotted in the bifurcation diagram and the data probes are the same as Figure 8.19. The red sphere is on a section which was computed with a course discretization, and the white sphere is on a section which includes a complex area and was computed with a finer discretization. The two two-dimensional manifolds have been drawn in wire-frame to emphasize the different discretizations and to show that they overlap slightly.

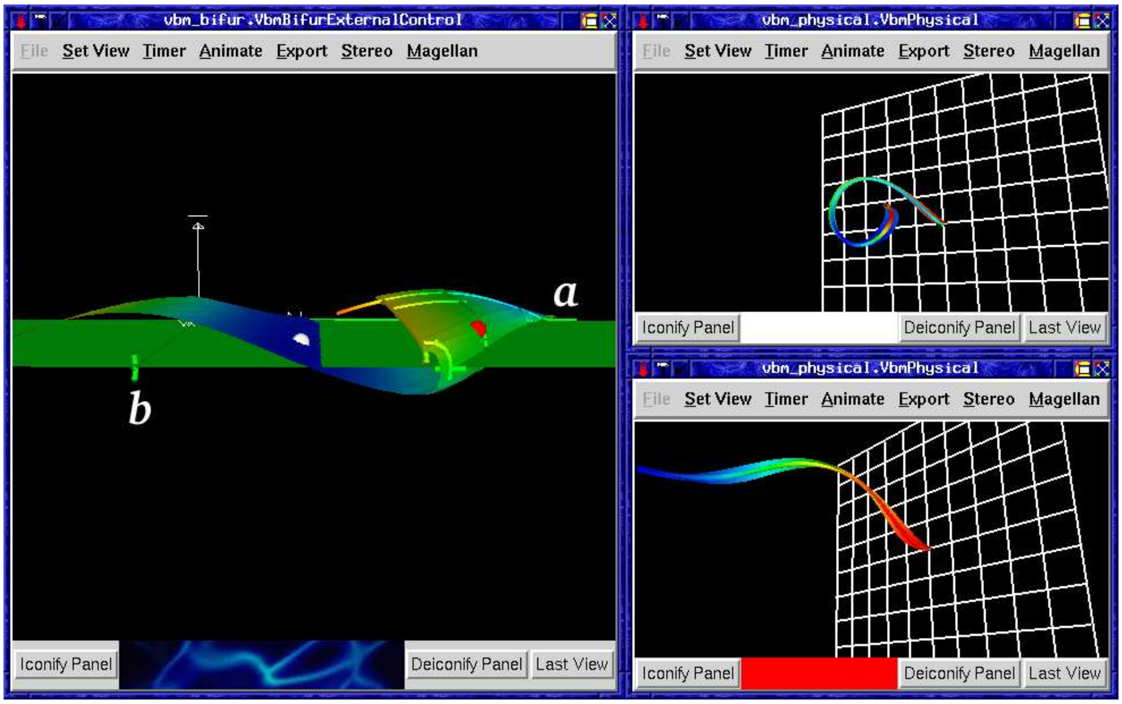

Figure 8.23: This figure shows several continuations, including the flat "trivial" manifold, in which all solutions are straight. The components plotted are the same as Figure 8.19, except p_2 has been scaled by five to make certain features easier to see. There are two lines where the "trivial" manifold intersects the manifold shown in Figure 8.22. The intersection near the red sphere is a line of bifurcation points, while the intersection near the white sphere is not a true intersection. Near the white sphere the color of the two surfaces is not the same; one surface is blue and the other is green. Hence, this intersection is only an artifact of the chosen projection. This diagram has been computed using compute engines based on VBM_1D_AUTO_FORT8 discussed in Section 7.3.4, and the bifurcation point marked with "b" is an example of the method this module uses to graphically represent bifurcation points. When this module reaches a bifurcation point it computes two stubs, one in each of the bifurcating directions. The point marked with "b" will be explored in Figures 8.24 and 8.27.

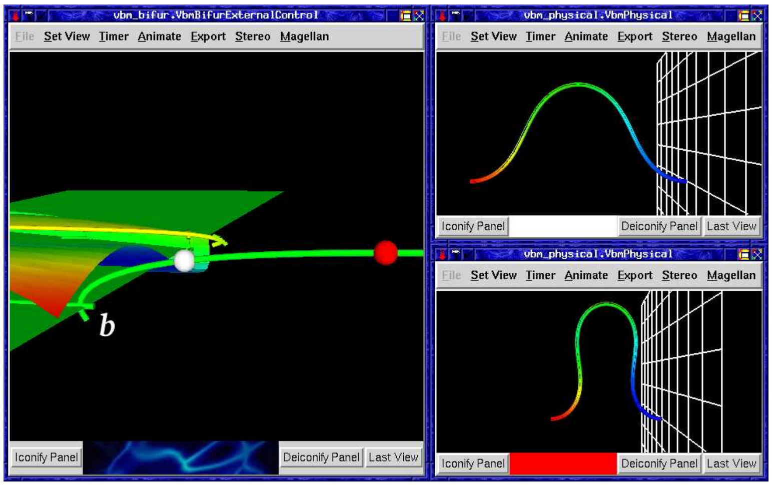

Figure 8.24: This figure shows several one parameter continuations and multiple two-dimensional manifolds. The components plotted in the bifurcation diagram and the data probes are the same as Figure 8.23. The bifurcation point marked with “b” has been extended by a parameter continuation calculation in p_1. The red and white spheres mark two points whose full solutions are shown in the data probes on the right.

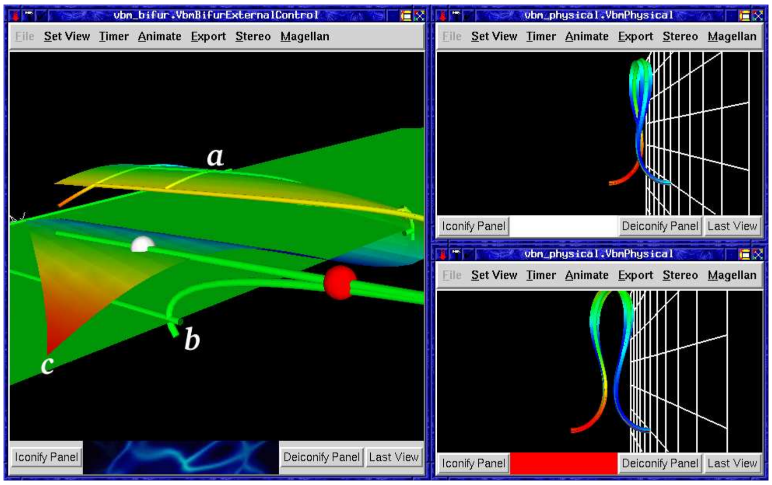

Figure 8.25: This figure shows several one parameter continuations and multiple two-dimensional manifolds. The components plotted in the bifurcation diagram and the data probes are the same as Figure 8.23. A secondary bifurcation point has been found on the bifurcating branch started at “b”. The secondary bifurcation point has been extended by a parameter continuation calculation in p_1 as well, and the secondary bifurcation branch intersects the sheet calculated in Figure 8.21 (which has been marked with a “c”).

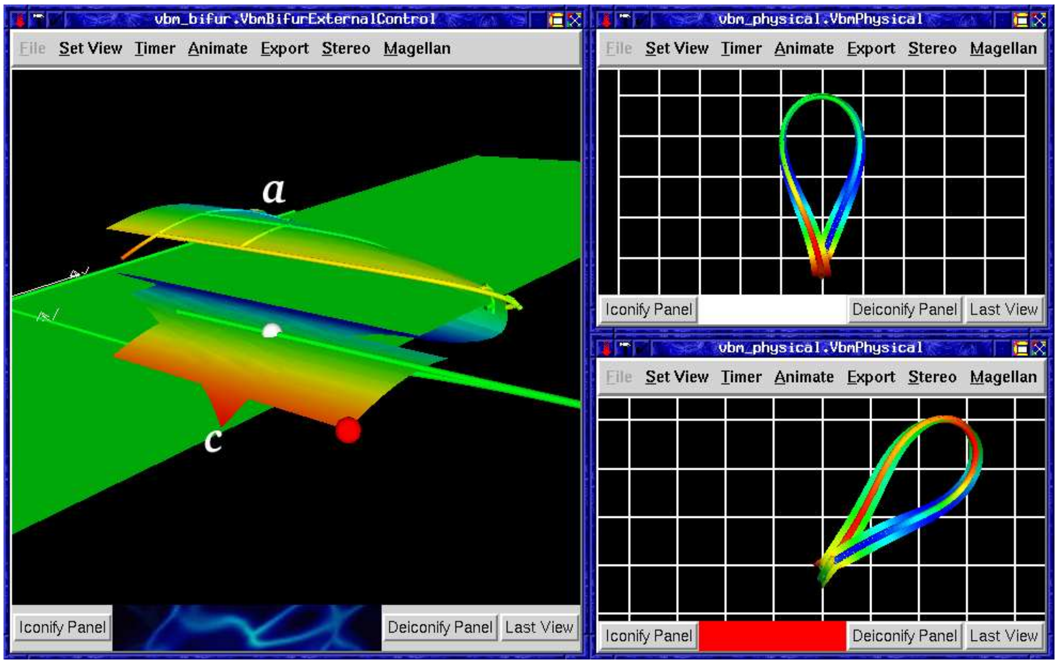

Figure 8.26: This figure shows several one parameter continuations and multiple two-dimensional manifolds. The components plotted in the bifurcation diagram and the data probes are the same as Figure 8.23. A new two dimensional manifold was computed, starting at the white sphere. The original two dimensional manifold (marked with the "c") and the newly computed manifold coincide exactly. Hence, the original manifold, started at a bifurcation point in p_2, and the new manifold, started at a secondary bifurcation point in p_1, are numerically demonstrated to be two parts of the same manifold.

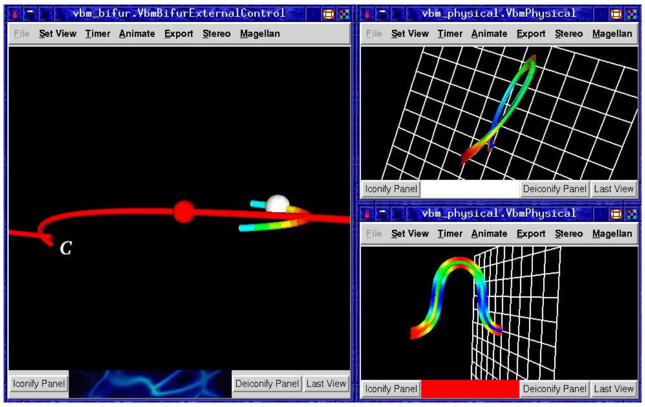

Figure 8.27: This figure shows a one dimensional continuation, in p_1, starting at the point marked "b" in Figure 8.22. In this diagram we have plotted $p_1, x(0.5)$, and $y(0.5)$, and the color is based on $q_4(0.5)$. The corresponding full solutions are shown on the right. The spatial coordinates for the full solutions are x, y, and z, the color of the tube is based on μ_1, and the color of the ribbon is based on q_4. The red sphere has been placed on the initial bifurcating branch and the white sphere has been placed on the secondary bifurcating branch.

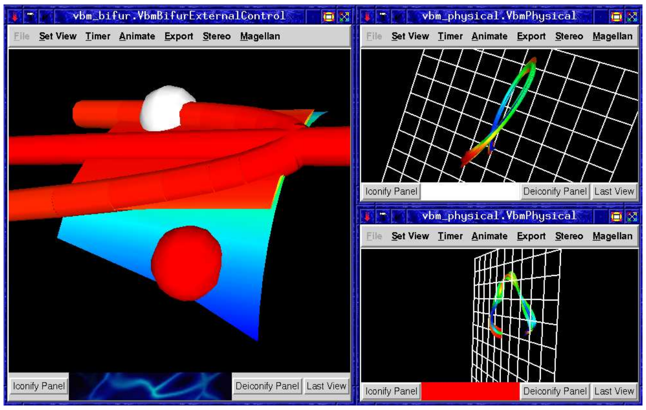

Figure 8.28: This figure shows a small two dimensional manifold in addition to the one dimensional manifolds shown in Figure 8.27. The components plotted in the bifurcation diagram and the data probes are the same as Figure 8.27. The two dimensional manifold is quite small and it is difficult to discern its exact shape. For example, it is difficult to see if it is double covered. We change the projection to create Figure 8.29.

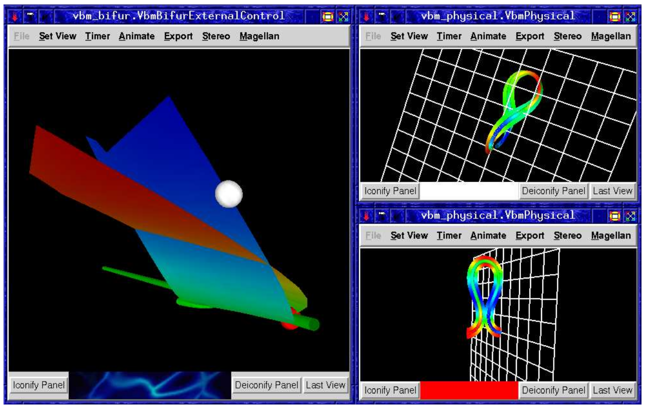

Figure 8.29: This figure shows the same manifolds as shown in Figure 8.28 except we have now chosen to plot $p_1, y(0.5)$, and $q_4(0.5)$, and base the color on $q_3(0.5)$. The structure of the two dimensional portion of the bifurcation diagram is now easier to discern. The self-intersection of the two dimensional manifold is not a true intersection because the colors do not match (i.e. the values of $q_3(0.5)$ are not the same at the two places in the manifold).

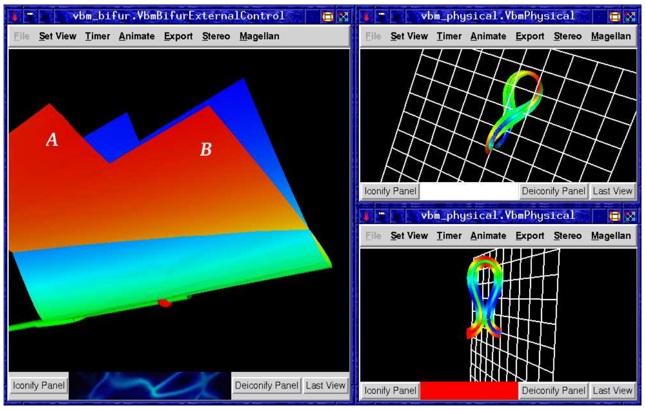

Figure 8.30: This figure shows the same manifolds as shown in Figure 8.29 except we have added an additional two dimensional manifold. The components plotted in the bifurcation diagram and the data probes are the same as Figure 8.29. The two dimensional manifold marked by "A" was computed by starting on the secondary bifurcating branch. The two dimensional manifold marked by "B" was computed by starting on the original bifurcating branch. Somewhat surprisingly, the two manifolds exactly coincide, instead of intersecting in a line of bifurcation points, such as in Figure 8.22.

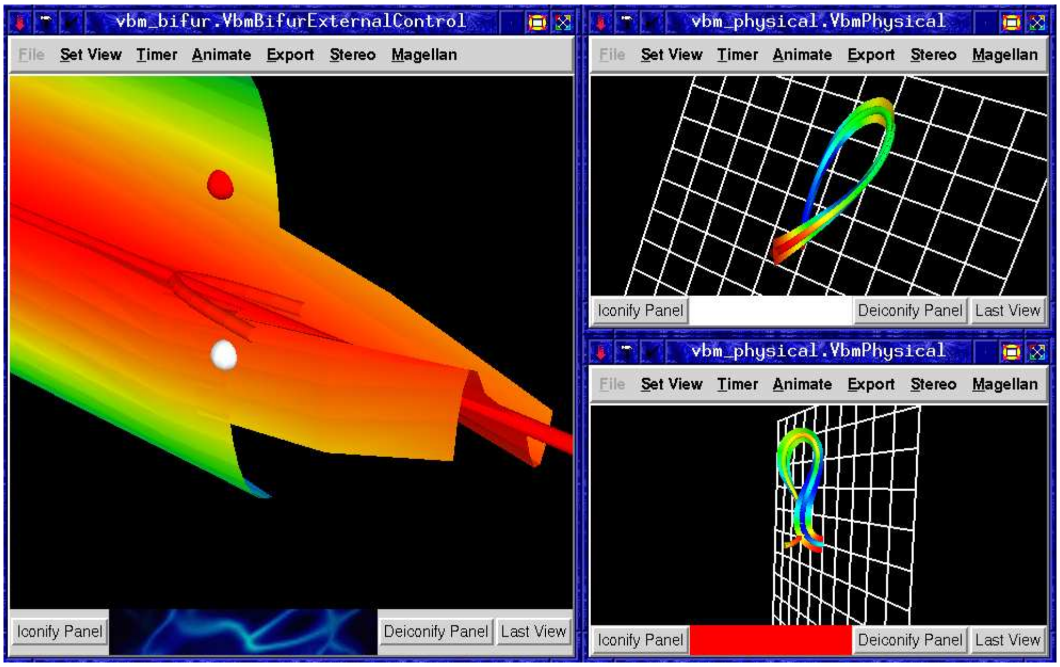

Figure 8.31: This figure shows the same manifolds as shown in Figure 8.30 except we have now chosen to plot $p_1, y(0.5)$, and $q_3(0.5)$, and base the color on $q_4(0.5)$. In this projection we can see that we have computed a cusp catastrophe.

Chapter 9

Conclusion and Discussion

In this dissertation we have described methods and software for the computation and visualization of the solutions set, or bifurcation manifolds, of systems of nonlinear equations of the form

$$F(\mathbf{x}) = \mathbf{0},\ F : \mathbb{R}^{n+m} \to \mathbb{R}^n,\ n, m \in \mathbb{Z}, n \geq 1, m \geq 1. \qquad (9.1)$$

While the solution of such equations is a mature field of research, this dissertation has emphasized algorithms allowing efficient computation, modular decomposition of the problem domain, parallelization at several levels, and tight coupling to scientific visualization, all of which have been utilized to allow the computations to be interactively steered.

The solution set of equation (9.1) is normally comprised of a set of smooth manifolds of dimension m, along with singular points at which the Jacobian of equation (9.1) does not have full rank. Numerical parameter continuation methods, both predictor corrector (P-C) and simplicial, are a class of methods which are highly effective for the computation of such solution manifolds, and Chapter 2 examined the methods in the case where $m = 1$. While both types of methods can effectively be used to compute the desired solution manifolds, P-C methods are preferred because of their flexibility, and their efficiency in the case of primary interest where $n \gg 1$, and equation (9.1) has some structure, for example if it arises as the discretization of a set of differential equations. The flexibility of P-C methods manifests itself in two major areas. First, a variety of methods may be used to generate an initial guess for the new solution of the system. These methods range from the very simple, as exemplified by natural parameter continuation, to arbitrarily high order methods. Second, P-C methods may utilize a variety of correction procedures, which may either solve the underdetermined system directly, or involve the addition of various constraint functions. These freedoms allow the P-C methods to be implemented efficiently, and Chapter 4 and Chapter 5 demonstrate how such efficient implementations may be carried out.

In this dissertation we also discussed generalizations of P-C methods to the case $m > 1$. Examples of such methods can be found in the literature, and we discussed two of them, one due to Rheinboldt [43] and one due to Henderson [22]. Chapter 3 discussed and compared these two algorithms with a method with novel aspects that we have called the Single Parameter With Switching (SPWS) method. SPWS decomposes a problem where $m > 1$ into several problems with $m = 1$. SPWS then uses standard methods on the $m = 1$ problems, combined with a method to generate connectivity, to construct an approximation to the solution manifold of equation (9.1). Indeed, the main issue with SPWS is how to generate connectivity and to control step sizes in different parameters. We stressed that the SPWS method has several serious deficiencies, which make it unsuitable for many purposes (which is probably the reason that its use has not been previously discussed in the literature). On the other hand, it is particularly suited to steered computations; in the context of steered computations the deficiencies of SPWS may easily be overcome, and its strengths may be used to good effect.

To generate modular and efficient computer implementations of the methods from Chapter 2 and Chapter 3, we have carefully considered the available tools and methodologies. Computers are complex, and their subtle nuances can have profound effects on the efficiency of numerical algorithms. Fortunately, one does not need to be cognizant of all of the possible pitfalls if one utilizes standard numerical packages such as BLAS, LAPACK, PETSc, or SCALAPACK, to name but a few, wherever appropriate. These libraries have been optimized by experts in the field and we do not wish to recreate their functionality. Accordingly, our numerical implementation of the algorithms from Chapter 2 and Chapter 3 use standard libraries wherever possible. Modularity of the implementation was accomplished by using an object oriented approach. Chapter 4 gave definitions of basic object oriented concepts as well as more advanced features that are used throughout our implementations.

MCCC, our framework for the implementation of multi-parameter P-C methods, was described in Chapter 5. It utilizes a novel object oriented decomposition of P-C methods into three pieces, namely GuessGeneration Objects, Solution Correction Objects, and Function Objects. MCCC has been implemented in Java. The main advantage of this object oriented approach is that it allows the various pieces of the algorithm to be changed independently. MCCC was designed to encourage experimentation by allowing the user easily to try arbitrary combinations of algorithms. On the other hand, MCCC was also designed for efficiency. Function Objects often have some type of structure which may be used to accelerate calculations, for example specific sparsity patterns of an associated Jacobian. Accordingly, a hierarchy of interfaces have been defined which Function Objects may implement. These interfaces are used to inform the other objects in MCCC that this Function Object provides additional services, which have efficient implementations (e.g. utilizing a specific sparsity pattern). By encapsulating all knowledge of the structure of the Function Object inside the Function Object, we maintain modularity. Finally, the object oriented decomposition of MCCC allows a straightforward implementation of a novel three tiered parallelism structure for P-C algorithms which allows efficient use of a large number of processors (see below).

Java was a particularly good choice for the implementation of MCCC for three reasons.

- Java provides all of the object oriented services that MCCC requires in a robust way.
- Java is portable.
- Java has a well defined interface to C.

The interface to C is mainly used to enhance efficiency. As described in Chapter 4 we require access to standard numerical libraries to achieve efficiency, and the C interface allows access to a vast array of such libraries. Because of these points, Java has begun to make progress in the high performance computing community. For example, there is the JavaGrande forum [27], which includes participants from Sun Microsystems, NIST, and IBM. JavaGrande is dedicated to the advancement of Java for numerical computations. For the purposes of MCCC, the main addition that is required to the current Java functionality is sparse matrix computations, either written in Java or as native routines accessible from within Java. Once such libraries are available, they may easily be incorporated into MCCC.

Good numerical algorithms and implementations do not, by themselves, suffice to steer computations. The data generated by the algorithms must be presented to the user in an easily understandable form, so that the user may quickly scientifically process the information and give feedback to the computations. Chapter 6 introduced both standard methods from scientific visualization, as well as specialized methods for parameter continuation problems of the form (9.1). For example, one cannot dismiss the importance of the ability to interactively select different projections, etc. Visualization aspects of one and two dimensional bifurcation diagrams are well understood (shading, etc.), and have been implemented in modules in VBM. We believe the interactive use of these standard visualization techniques to be highly effective in the steering and visualziation of parameter continuation computations.

For the purposes of steering parameter continuation algorithms, the most important single step in visualization is the post-processing of the data to construct the bifurcation diagram. This post-processing is obviously very problem dependent, and any software package which combines computation and visualization must be quite flexible in its approach to problems of the general form of equation (9.1). VBM is the software package which we have designed to incorporate general and problem specific visualization tools, with the interactive steering of parallel numerical continuation algorithms. The most important parts of VBM are three APIs designed to allow a wide range of continuation algorithms and visualization tools to be used together transparently. The Bifurcation Visualization API allows the complete bifurcation diagram to be visualized, and the Data Probe API allows each individual solution to be visualized in detail. The new DataViewer visualization library, discussed in Appendix A, has been used extensively in the development of modules which implement both of these VBM APIs. The Compute Engine API, on the other hand, allows the control of continuation algorithms for the augmentation of bifurcation diagrams. The main advantage of these APIs is that any module which implements one of them may be used with all other modules that implement the APIs. For, example, if a new Compute Engine is created, it may immediately access all of the visualization tools which have already been developed. The Compute Engine API is especially important is this regard in that all compute engines may be controlled using a single coherent GUI. The user does not need to be cognizant of how the compute engine is used, but only needs to know how to use the GUI for the Compute Engine API.

Chapter 8 contained a variety of examples implemented in VBM, using both locally developed continuation algorithms as well as those found in standard packages. The current implementation of VBM has been validated in a variety of different pre-existing examples [1, 35], and also used as part of the research leading to several publications [36, 34, 20, 45, 13, 29, 51], some of which are described in Chapter 8. The utility of interactive visualization of bifurcation diagrams through the Bifurcation Visualization API and full solutions through the Data Probe API, in both one-dimensional and two-dimensional problems is, we believe, clearly demonstrated in these examples.

The main conclusion which we have drawn is that well designed APIs are the most important facet of software packages such as VBM and MCCC which combine interactive visualization and computation. The APIs are a distillation of the interactions between bifurcation diagrams, full solutions, and parameter continuation algorithms that are necessary in useful software. In VBM, the various APIs have been used to combine computational algorithms, both of our own design, and from the mathematical community, with an array of visualization tools to allow the interactive control of parameter continuation computations. One area where the APIs in VBM and MCCC are especially useful is in the realm of parallelism. Within the three levels of parallelism described in Section 5.3, a large number of processors may be used very efficiently. First, several GuessGeneration Objects may function simultaneously. Second, each GuessGeneration Object may have several SolutionCorrection Objects which function simultaneously. Finally, each SolutionCorrection Object may be be parallelized used standard methods for numerical algorithms. For example, one may quite easily utilize $4 \times 8 \times 4 = 128$ processors within a conservative configuration such as

- four instantiations of a GuessGeneration Object,
- eight instantiations of a SolutionCorrection Object for each GuessGeneration Object,
- four processors used within each SolutionCorrection Object.

We also note that such a configuration is perfectly suited to a NUMA, a distributed architecture machine that is notoriously hard to program. High performance communication is only required within each group of four processors in the first level and the communication between other groups is rather small.

We close with a description of several open problems and issues which have been raised. First, for the efficiency goals of MCCC to be fully realized, low level linear algebra routines for sparse matrices are required. While it is straightforward to call C and C++ routines from Java, good Java wrappers still need to be developed to make their use easy and efficient. While MCCC is currently useful for the development of, and experimentation with, parameter continuation algorithms, and has been used on several non-trivial problems, it cannot currently compete on the basis of speed with parameter continuation algorithms implemented in a native language. Fortunately, the JavaGrande forum [27] was created to address some of these numerical issues with regards to Java. Second, as was demonstrated in Section 8.3 and Section 6.4, the computation of three or higher dimensional manifolds, utilizing algorithms such as SPWS, is straightforward. Unfortunately, the required visualization of such bifurcation manifolds for interactive control of the parameter continuation algorithms requires extensive further research. While visualization methods exist in the literature for three dimensional manifolds which have some analog to physical entities, we are unaware of effective methods for manifolds which are purely mathematical in nature. We also point out that there are visualization methods found in the literature for the one and two-dimensional cases which are currently not implemented in VBM, but which might prove useful (e.g. texture mapping). Finally, the most computationally intensive problems addressed in this dissertation have been discretizations of ODE BVPs. Even in the ODE case the problems can have large I/O and data storage requirements. Some of these issues have been addressed in this thesis, but there is still room for further research and development. On the other hand, one may wish to extend our methods to larger problems, such as discretized Partial

Differential Equations (PDEs), with a commensurate increase in computational and visualization complexity. For example, discretizations of PDEs often have at least 10^3 equations, and they can be much larger. New data reduction techniques will be required, similar to our use of IVP reconstruction of ODE BVP solutions.

More information, including the downloading of source code, can be found on the following web pages.

Table 9.1: Web pages for more information.

VBM	`http://lcvmwww.epfl.ch/VBM`
MCCC	`http://lcvmwww.epfl.ch/MCCC`
DataViewer	`http://lcvmwww.epfl.ch/DV`

Appendix A

DataViewer

A.1 Introduction

DataViewer is a scientific visualization library created to allow the easy development of efficient visualization programs for those who are not graphics programming experts. It consists of a high-level set of routines for rendering geometric objects and utilizes the OpenGL graphics API [39] for its low level rendering. DataViewer has been used for several different visualization tasks, e.g. most of VBM's visualization modules have been developed using it. The goals that have driven the design of DataViewer are to be efficient, easy to use, portable, closely connectable to numerical computations, and available free of charge to the mathematical and scientific community.

The design decisions that were made in the development of DataViewer were profoundly affected by the historical software context in which they were made. Accordingly, Section A.2 will describe some of the history of DataViewer, and other software to which it may be compared. The core of DataViewer is its scene graph representation of geometrical objects and their properties. Section A.3 will detail DataViewer's implementation of a scene graph and give examples of geometrical objects that DataViewer provides.

A.2 History

Many software packages and algorithms have been developed for the rendering of three dimensional graphics on modern computers (for a thorough discussion of graphical rendering techniques one may see, for example, [19]). Only certain of these algorithms are of interest in our work because of our interactivity requirement. For example, ray-tracing performs well under many circumstances, but is too slow for our purposes. Accordingly, we will restrict our attention to packages that allow interactive control.

Commonly, interactive graphics require some type of hardware assistance for the rendering process. Geometry rendering is notoriously slow, but since it is a repetitive process it can be efficiently implemented in hardware. Many companies produce hardware accelerators for graphical rendering (for example, "Silicon Graphics" [12] and "Evans and Sutherland" [18], to name but a few), but, unfortunately, each accelerator tends to use its own proprietary library for accessing the hardware. Obviously, none of these libraries are appropriate for the development of general visualization software, since the software would only be usable on the specific hardware platform.

Many attempts have been made to place software layers on top of the specific hardware implementations to make applications portable. These libraries define an API, and software vendors are expected to provide implementations of this API for their hardware. Examples include PEX, PHIGS, GL, OpenGL, and Direct3D. Today, the situation has become clearer, and it appears that OpenGL is the preferred solution on Unix systems and either OpenGL or Direct3D is the preferred solution on Microsoft Windows. These libraries offer the advantage that an application may be developed using them, and this application will then be portable to any hardware which has an implementation of the appropriate API.

To achieve portability and maintain efficient access to hardware renderers, Paffenroth implemented the first version of DataViewer (DataViewer1.0) using OpenGL. Several useful applications were developed using DataViewer1.0, but a few problems were noticed with this approach. First, the development of DataViewer1.0 applications required the user to develop OpenGL code themselves. OpenGL is very general, but quite low level. It only provides simple geometrical primitives (such as lines and triangles), and more high level objects must be implemented separately. Accordingly, writing applications directly in OpenGL can be complex and time consuming. Second, the interface in DataViewer1.0 was written in Motif, a graphical user interface (GUI) library for Unix. Motif, as OpenGL, is very general but low level. Thus, it was not easy to develop custom GUIs in DataViewer. Also, Motif is proprietary software, and even though it is widely available, there exist Unix based systems for which it is either not available or only available at a significant cost.

We decided that our needs called for some type of layer on top of OpenGL, or some similar library, to provide a higher level of abstraction and which would be easier to use. We examined the libraries and packages that were available in late 1996, and found several candidates, for example OpenInventor [52], GeomView [38], and Java3D [46]. Unfortunately, none of the available software exactly met our needs. OpenInventor is a product developed by Silicon Graphics as an evolution of their Inventor library. It is a layer written on top of OpenGL which provides high level geometrical objects, manipulation tools, and basic GUI elements. OpenInventor provided all of the functionality that we required, but was only ubiquitous on computers from Silicon Graphics. GeomView is a software package developed at the Geometry Center of the University of Minnesota. It is a visualization system, as opposed to a library such as OpenInventor. It has many useful features, but being a complete system we found it somewhat difficult to closely couple to numerical computations. Java3D is an extension to the Java [6] programming language which allows geometrical rendering to be efficiently performed in Java programs. It can use a variety of low level rendering libraries, including OpenGL and Direct3D. Java3D seemed perfect for our needs. It provided efficient, easy to use geometry rendering, can be tightly coupled to numerical computations, and can be used on any computer with a Java Virtual Machine. Unfortunately, at the time DataViewer was being developed, Java3D was only a very preliminary specification, and a fully functional version was not released until November 1998.

Hence we were left with the prospect of creating our own software library. DataViewer2.0 was developed by R. Paffenroth, T. Stone, and A. Ahearn; it is our attempt to create a library which encompasses all of our needs, based on our experiences in the development of DataViewer1.0.

A.3 DataViewer2.0 Design

DataViewer2.0 differs from DataViewer1.0 in two main respects. First, DataViewer2.0 uses Python to create GUIs, instead of Motif, allowing GUIs to be implemented much more easily. Second, DataViewer2.0 encapsulates all OpenGL calls. The user does not need to know anything about OpenGL to use DataViewer2.0.

A.3.1 Implementation

DataViewer2.0 is written entirely in C++, and uses many of the language's object oriented features, such as templates and virtual functions [48]. Each class in DataViewer2.0 inherits from a single base class called DVobject. Each DVobject defines a *draw* routine which canonically contains a set of OpenGL commands that create the graphical representation of the object. In addition, each DVobject contains a set of properties, such as color, translation, and rotation. These properties will be discussed in greater detail below.

Python gives access to the Tk [40] widget library, which DataViewer utilizes for the design of GUIs. Tk is a library of widgets which can be accessed from a wide variety of languages, including Tcl, Python, C/C++, and Perl. It is a robust and mature code which has implementations on all Unix systems as well as Microsoft Windows. In our opinion, Tk is much easier to use than Motif and makes the GUI more flexible. With Tk and Python it is quite easy for the user of DataViewer to create custom GUIs for their problems.

Finally, DataViewer2.0 provides several features which are accessible in any application which uses the library. For example, DataViewer provides stereo viewing capabilities, using hardware such as Crystal Eyes [26] and access to six degree-of-freedom controls, such as Logitech's Magellan [25].

A.3.2 Scene Graph and Properties

The most important feature of DataViewer2.0 is its scene graph. In a scene graph, a geometrical object is represented in a hierarchical fashion using a tree. A given object is divided into pieces, each of which is either a *leaf node* (i.e. a geometrical object which can be represented as an atomic data type) or a *group node* which itself contains other nodes. DataViewer2.0 implements a scene graph by using two main types of nodes.

The first type of node in DataViewer2.0 is the *geometry node*. Geometry nodes are the leaf nodes of DataViewer2.0 and they represent geometrical objects. DataViewer2.0 differs from most libraries that use scene graphs in that DataViewer2.0 not only provides simple leaf nodes, but also leaf nodes which are quite complex geometric objects. For example, the object shown at the end of this section in Figure A.1 is a single leaf node of DataViewer2.0.

Even though complex leaf nodes reduce the flexibility of a library, DataViewer2.0 includes complex objects for two reasons. First, certain types of complex objects are commonly used as metaphors for scientific visualization in our work (for example see Chapter 6 and Chapter 7), and we wish these complex objects to be easy to use. Second, each node in DataViewer2.0 has a certain amount of computational overhead which is required for the processing of the scene graph. For complex objects this overhead needs to be paid only once, as opposed to conglomerations of simpler objects where the overhead must be paid for each object.

The second type of node in DataViewer2.0 is the *container* node. Container nodes group together other nodes into a set, which may be used as a single node. In a container node, the other nodes which are contained within the container node are known as its *children*. Similarly, the container node itself is known as the *parent* of the nodes which it contains. DataViewer divides container nodes into two classes. The first class is the *normal containers*, with the property that each time the draw function of the parent is called the draw functions of all of the children are called. The second class is the *selection containers*, with the property that each time the draw function of the parent is called only some of the children's draw functions are called. Generally, the selection containers are used for special affects, such as animation and level-of-detail [1].

A scene graph also requires some way to handle *properties*, such as color. For example, OpenInventor chose to implement properties as leaf nodes themselves. In other words, leaf nodes are not just geometrical objects, they may also be properties. A color node may appear in the scene graph, and each geometric object is drawn in the color of the color node which is closest to it (depending on how the scene graph tree is traversed). Unfortunately, while this approach is quite flexible it introduces an order dependence in the scene graph. Depending on the order in which the scene graph is rendered different results may occur. DataViewer took a different approach by including the properties into the geometry nodes themselves. Using these types of properties each node may either set a given property itself, in which case it is drawn using its own value of the property, or it may inherit the property from its parent. An example of a group of spheres and a set of property definitions is shown in Figure A.2.

Using this type of property setup the scene graph is not order dependent, at only a small lose of flexibility. We also note that, subsequent to our development of DataViewer2.0, Java3D has adopted this property structure as well. Further detail on the DataViewer2.0 software library may be found at `http://lcvmwww.epfl.ch/DV`.

[1] Level-of-detail is a method by which a given object is rendered differently based upon its relationship to the viewer. For example, an object may be rendered at lower detail when it is far from the viewer.

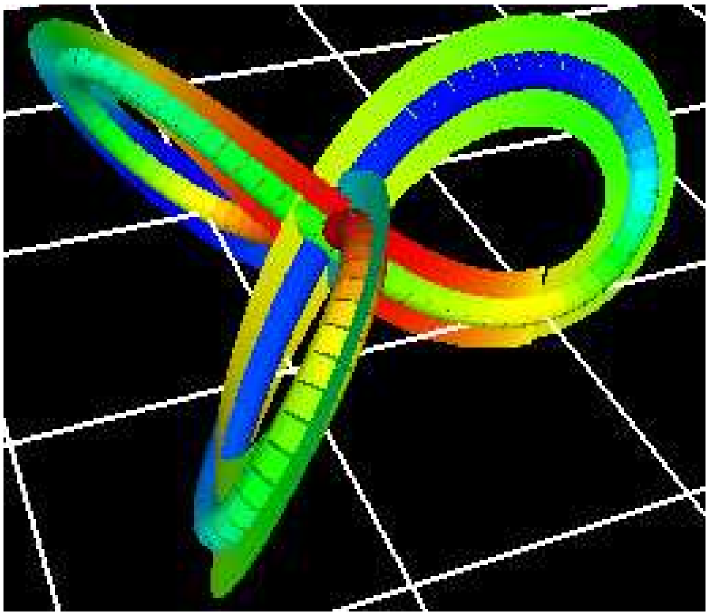

Figure A.1: This figure shows a complex geometric object which can be represented in DataViewer2.0 by a single leaf node. The tube and the ribbon are both included in the definition of the leaf node.

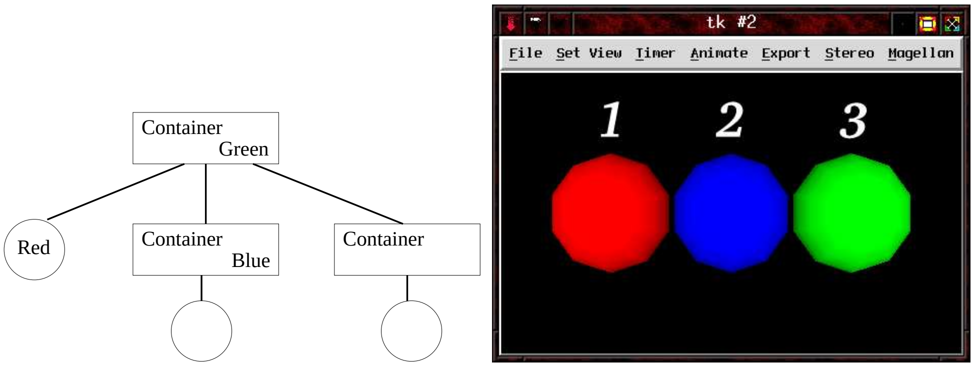

Figure A.2: This figure shows a scene graph on the left and the graphical result of the scene graph on the right. The top container sets the color property to be green. The sphere attached to the top container sets the color property to red, and the first container attached to the top container sets the color property to blue. No other nodes set the color property. The sphere marked with 1 is drawn in red since it sets that property itself. The sphere marked with 2 inherits it blue color from the container directly above it. The sphere marked with 3 inherits is green color from the top container.

BIBLIOGRAPHY

[1] J. C. Alexander, J. H. Maddocks, and B. A. Michalowski. Shortest distance paths for wheeled mobile robots. *IEEE Trans. on Robotics and Automation*, 14(5):657–662, Oct. 1998.

[2] E. L. Allgower and K. Georg. *Numerical Continuation Methods.* Springer-Verlag, 1990.

[3] E. L. Allgower and K. Georg. Continuation and path following. *Acta Numerica*, 22:1–64, 1992.

[4] E. L. Allgower and K. Georg. Numerical path following. In P. G. Ciarlet and J. L. Lions, editors, *Handbook of Numerical Analysis*, volume 5, pages 3–208. Elsevier Science B. V., 1997.

[5] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users Guide.* SIAM, second edition, 1995.

[6] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[7] U. Ascher, J. Christiansen, and R. D. Russell. A collocation solver for mixed order systems of boundary value problems. *Math. Comp.*, (33):659–679, 1979.

[8] C. A. Atwood. Technical computing software on Sun UltraSPARC servers and desktops. Technical report, SuperG, 1998.

[9] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. PETSc home page. http://www.mcs.anl.gov/petsc, 1998.

[10] L. S. Blackford, J. Choi, E. D'Azevedo A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. *ScaLAPACK Users Guide.* SIAM, second edition, 1997.

[11] C. Cason. *Persistence of Vision Ray-Tracer User Documentation*, 1994. `http://www.povray.org`.

[12] Silicon Graphics Corporation. WWW page. `http://ww.sgi.com`.

[13] D. J. Dichmann, Y. Li, and J. H. Maddocks. Hamiltonian formulations and symmetries in rod mechanics. In J. P. Mesirov, K. Schulten, and D. W. Sumners, editors, *Mathematical Approaches to Biomolecular Structure and Dynamics*, volume 82 of *The IMA Volumes in Mathematics and Its Applications*, pages 71–113. 1996.

[14] E. J. Doedel, H. B. Keller, and J. P. Kernévez. Numerical analysis and control of bifurcation problems, part I. *Int. J. of Bif. and Chaos*, 3, 4:493–520, 745–772, 1991.

[15] E. J. Doedel, H. B. Keller, and J. P. Kernévez. Numerical analysis and control of bifurcation problems, part II. *International Journal of Bifuration and Chaos*, 4:745–772, 1991.

[16] E. J. Doedel, X. Wang, and T. Fairgrieve. *AUTO 94, Software for Continuation and Bifurcation Problems in Ordinary Differential Equations.* Caltech, Dept of Applied Mathematics, 1994.

[17] G. Domokos and R. C. Paffenroth. Case study: Visualization for boundary value problems. In R. D. Bergeron and A. E. Kaufman, editors, *IEEE Visualization '94 Conference Proceedings*, pages 345–348. IEEE Computer Society Press, 1994.

[18] Evans and Sutherland Computer Corporation. WWW page. `http://www.es.com/`.

[19] J. Foley, A. vanDam, S. Feiner, and J. Hughes. *Computer Graphics Principles and Practice.* Addison-Wesley, 1996.

[20] P. Furrer, R. Manning, and J. H. Maddocks. Multiple equilibria in DNA rings. In preparation, 1999.

[21] G. H. Golub and C. F. Van Loan. *Matrix Compuations.* The Johns Hhopkins University Press, 2nd edition, 1989.

[22] M. E. Henderson. Computing implicitly defined surfaces: Two parameter continuation. Technical report, IBM Research Division, T. J Watson Research Center, 1993.

[23] M. E. Henderson. personal communication, October 1998.

[24] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[25] Logitech Inc. Magellan 6d mouse. WWW page. `http://www.logitech.com`.

[26] StereoGraphics Inc. CrystalEyes stereo3D visualization hardware. WWW page. `http://www.stereographics.com`.

[27] JavaGrande. WWW page. `http://www.javagrande.org`.

[28] J. Langer, D. Vrajitoru, and J. H. Maddocks. Personal communication during visit to LCVM, November 1998.

[29] Y. Li and J. H. Maddocks. On the computation of equilibria of elastic rods, Part I: Integrals, symmetry and a Hamiltonian formulation. *J. Comp. Phys.*, to appear.

[30] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH '87. Proceedings of the 14th annual conference on Computer Graphics.*, pages 163–169, 1987.

[31] M. Loukides. *System Performance Tuning.* O'Reilly and Associates, 1992.

[32] M. Lutz. *Programming Python.* O'Reilly and Associates, 1996.

[33] J. H. Maddocks. Stability and folds. *Arch. for Rat. Mech. and Anal.*, 99(4):301–328, 1987.

[34] J. H. Maddocks, R. Manning, R. Paffenroth, K. Rogers, and J. Warner. Interactive computation, parameter continuation, and visualization. *International Journal of Bifurcation and Chaos*, (8):1699–1715, 1997.

[35] J. H. Maddocks and R. L. Sachs. Constrained variational principles and stability in Hamiltonian systems. In Dumas, Meyer, and Schmidt, editors, *Hamiltonian Dynamical Systems*, volume 63 of *IMA Volume in Mathematics and Its Applications.* Springer Verlag, 1995.

[36] R. S. Manning, J. H. Maddocks, and J. D. Kahn. A continuum rod model of sequence-dependent DNA structure. *J. Chem. Phys.*, 105:5626–5646, 1996.

[37] J. Moré, B. Garbow, and K. Hillstrom. *Minpack.* Argonne National Laboratory.

[38] T. Munzner, S. Levy, and M. Phillips. *The Geomview Manual.* The Geometry Center, University of Minnesota. `http://www.geom.umn.edu/software/download/geomview.html`.

[39] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide.* Addison-Wesley, 1993.

[40] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[41] L. L. Peterson, B. S. Davie, and D. Clark. *Computer Networks: A Systems Approach.* Morgran Kaufman, 1996.

[42] G. F. Pfister. *In Search of Clusters: The Coming Battle in Lowly Parallel Computing.* Prentice Hall, 1995.

[43] W. C. Rheinboldt. On the computation of multi-dimensional solution manifolds of parameterized equations. *Numer. Math.*, 53:165–181, 1988.

[44] W. C. Rheinboldt and J. V. Burkardt. A locally-parameterized continuation process. *ACM Trans. Math. Software*, 9:215–235, 1983.

[45] K. A. Rogers. *Stability of Twisted Elastic Rods.* PhD thesis, University of Maryland, College Park, 1997.

[46] K. Rushforth, M. Deering, and H. Sowizra. *The Java 3D API Specification.* Addison-Wesley, 1997.

[47] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[48] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, third edition, 1997.

[49] P. R. Turner, editor. *Numerical Analysis and Parallel processing.* Lancaster Numerical Analysis Summer school 1987, Springer-Verlag, 1989.

[50] X. Wang. Report on parallelization of AUTO 94. Master's thesis, Caltech, Dept of Applied Mathematics, July 21 1994.

[51] J. A. Warner. Numerical simulations in rod mechanics. Master's thesis, University of Maryland, College Park, 1996.

[52] J. Wernecke. *The Inventor Mentor : Programming Object-Oriented 3d Graphics With Open Inventor.* Addison-Wesley, second edition, 1994.

[53] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98. Proceedings of the 25th annual conference on Computer Graphics.*, pages 169–177, 1998.