Computing critical points of polynomial systems using PHCpack and Python

BY

KATHERINE PIRET
B.S., University of Illinois at Chicago, 2003
M.S., University of Illinois at Chicago, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Mathematics
in the Graduate College of the
University of Illinois at Chicago, 2008

Chicago, Illinois

This thesis is dedicated to my husband Chris Piret, my mother Meiying Qi, my deceased father Zhongyuan Wang, my deceased maternal grandfather Yusheng Qi, my uncle Honghao Qi, my father in-law Dale Piret, my cousin Paul Piret, my best friend Dave Turkington, my best officemate Dimitrios Diochnos, and my little friend Scotchy. Without their support, I would not have accomplished my goal of finishing my graduate studies at UIC.

ACKNOWLEDGMENTS

I would like to thank everyone who has been involved in this work and provided me with help and encouragement.

I would like to express my deep gratitude towards my thesis advisor, Prof. Jan Verschelde, for giving me tremendous guidance in my research. Without his inspiration and support, I wouldn't have made it this far with my PhD studies.

I would also like to thank all the committee members, Prof. Jan Verschelde, Prof. Shmuel Friedland, Prof. David Nicholls , and Prof. Gyorgy Turan for taking the time to review my thesis and provide me with their valuable insight and comments.

I would also like to thank William Stein for his generous support with the development of the Python interface to PHCpack.

I would also like to thank the NSF for supporting me as a research assistant under the grant numbers, 0105739, 0134611, 0410036, and 0713018.

I would also like to thank the Department of Mathematics, Statistics, and Computer Science at UIC for supporting me through my years as a graduate student.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF MATHEMATICS SYMBOLS

The symbols used in the thesis with their explanations are listed below. Different meanings for the same symbol occur when there is no confusion from context.

N	Number of Equations
n	Number of Unknowns
m	Multiplicity
$\mathbf{x}$	Vector $[x_1, x_2 \ldots x_n]$
$J(\mathbf{x})$	Jacobian Matrix
f	Polynomial system
λ	Natural parameter
t	Artificial parameter
g	Start system
$\mathbb{R}$	Real plane
$\mathbb{C}$	Complex plane
API	Application Programmers Interface
$DBAPI$	Python Database API
MPI	Message Passing Interface
$MySQL$	The world's most popular open source database

LIST OF MATHEMATICS SYMBOLS (Continued)

TCP/IP	Transmission Control Protocol/Internet Protocol
SVD	Singular Value Decomposition
r	Numerical Rank
ϵ	Tolerance for Numerical Rank

SUMMARY

Our first focus of this thesis is to build a Python interface to PHCpack, which is a software package to solve polynomial systems. As a powerful scripting and programming language, Python is robust and flexible with convenient structures to store data and features that can let one build extension modules such as extending C/C++ to Python. We decided to experiment with Python and were able to extend the C code to Python via a flexible Python interface, which contains function calls including a blackbox routine, a track routine, and a mixed-volume routine.

To add to the Python interface, we then created a client/server program to solve many polynomials concurrently. The idea is that a server sends multiple polynomial systems to available clients one at a time and each client solves the received system and sends the solution set back to the server. The client/server program serves as a flexible model for coarse grained distributed tasks.

Our second focus is on detecting and locate the critical points for a polynomial system, which occurs often in Mechanical Engineering, for instance, finding all bifurcation points of a Stewart-Gough platform. We can apply homotopy continuation methods locally to systems occurring in this kind of application domain to find critical points. Considering a polynomial system with one parameter, our interest is to find all critical points of the system for a range of parameter values via an efficient local method using homotopy continuation. In this thesis, we present a new algorithm to compute critical points.

CHAPTER 1

INTRODUCTION

Polynomial systems often arise in many areas of science and engineering, as seen in the case studies in [80]. To approximate all isolated solutions of polynomial systems, numerical path following techniques have been proven to be reliable and efficient during the past two decades. The general strategy of the numerical path following techniques is setting up a collection of implicitly defined paths that can be traced to the solutions of the original system, see [2] for more details. In addition, homotopy continuation methods used to compute numerical approximations to all isolated solutions of a polynomial system are known as "pleasingly parallel" because of their low communication overhead. These methods scale well for a large number of processors. Moreover, homotopy continuation methods can be applied locally to observe singularities and find critical points. PHCpack [87] is a software package based on homotopy continuation methods to solve polynomial systems. Other available software packages to solve polynomial systems include HOMPACK [91], Bertini [3], HOM4PS [47] and PHoM [34].

1.1 Related notations and definitions

Since this thesis is focused on our studies of polynomials, let us begin our introduction by defining polynomials, describing what the solutions of polynomial systems are like, and how we might represent these solutions numerically.

The formal definition of a multivariate polynomial, including the single-variable case, is defined as follows.

Definition 1.1.1 A function $f(\mathbf{x}) : \mathbb{C} \to \mathbb{C}$ in n variables $\mathbf{x} = (x_1, \ldots, x_n)$ is a *multivariate polynomial* if it can be expressed as a sum of terms each being the product of a coefficient and a product monomial of variables raised to nonnegative integer powers. A polynomial f is a function that is defined as

$$f(\mathbf{x}) = \sum_{d \in I} c_d x^d, \qquad x^d = \prod_{i=1}^{n} x_i^{d_i}, \tag{1.1}$$

where I is a finite index set and $c_d \in \mathbb{C}$.

Definition 1.1.2 *A system of polynomials* can be conveniently defined as :

$$f(\mathbf{x}) = \begin{bmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_N(x_1, \ldots, x_n) \end{bmatrix} = 0$$

The polynomial system $f(\mathbf{x})$ consists of N polynomials, namely $f_i(x_1, \ldots, x_N)$ on $\mathbb{C}^n$ contained in the ring $\mathbb{C}[x_1, \ldots, x_n]$ of polynomials in the variables with complex coefficients.

Let us define the solution set for a system of multivariate polynomials.

Definition 1.1.3 The *solution set* for a *multivariate polynomial* can be defined as

$$V(f) = f^{-1}(0) = \{x \in \mathbb{C}^n | f(x) = 0\}, \tag{1.2}$$

$V(f)$ is the *algebraic set of* f or the *variety associated to* f.

The solution set of a single-variable polynomial can be represented numerically by a list of approximated solution points. For a system of multivariate polynomials, the solution set is more complicated than that for a single-variable polynomial. Such a system may have solution sets of several different dimensions. A solution is isolated if there exists a ball with nonzero radius centered at the solution, which contains only one solution.

Definition 1.1.4 The *Jacobian matrix* of a polynomial system f is defined as

$$J(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & \cdots & \frac{\partial f_N}{\partial x_n} \end{bmatrix} = D_{\mathbf{x}} f$$

Just as a univariate polynomial may have multiple roots, a multivariate system may have solution sets with multiplicity more than one. Singularities of any polynomial system rely on the rank of the Jacobian matrix J. We define a point x to be a regular point if the Jacobian matrix has full rank at x. Otherwise, we define a point x to be a singular point if the Jacobian matrix has rank deficiency at x.

Definition 1.1.5 A solution z to $f(\mathbf{x}) = 0$, $f = (f_1, \ldots, f_N)$, $\mathbf{x} = (x_1, \ldots, x_n)$, $N \geq n$, is *singular* if $J(f)$ has rank $r < n$ at z.

We distinguish two groups of singularities [80], isolated singularities and nonisolated singularities. Isolated singularities are isolated solutions that are singular. Nonisolated singularities are the singular points on curves of various degrees. In this thesis, we assume that singularities are isolated. For more information on local methods to compute nonisolated singularities, see T.Y. Li and Y. Kuo's paper [42]. What we mean by computing critical points in this thesis is finding isolated singularities.

1.2 Homotopy continuation methods

Homotopy continuation methods [54], [55], [80] are globally convergent and exhaustive solvers for computing numerical approximations to all isolated solutions of polynomial systems, except, perhaps, at the very end of the paths if a polynomial system to be solved has singular solutions.

Definition 1.2.1 In order to solve a *target system* $f(\mathbf{x}) = \mathbf{0}$, we construct a *start system* $g(\mathbf{x}) = \mathbf{0}$ and define the *artificial parameter homotopy* as follows,

$$h(\mathbf{x}, t) = \gamma(1-t)g(\mathbf{x}) + tf(\mathbf{x}) = \mathbf{0}, \quad \gamma \in \mathbb{C}, \quad t \in [0, 1]. \tag{1.3}$$

The constant γ is a random number that can be called the *accessibility constant* [80]. The t is the *continuation parameter* and varies from 0 to 1, deforming the *start system* g into the *target system* f.

Given that the input of a problem is a polynomial system with one natural parameter, we define the following homotopy as a natural parameter homotopy, denoted as $f(\mathbf{x}, \lambda) = \mathbf{0}$.

Let's look at a simple demonstration of a *natural parameter homotopy* on a circle.

Example 1.2.2 The equation for a unit circle is defined as follows:

$$\lambda^2 + x^2 - 1 = 0. \tag{1.4}$$

We let λ be a parameter of the polynomial function $f(x, \lambda)$.

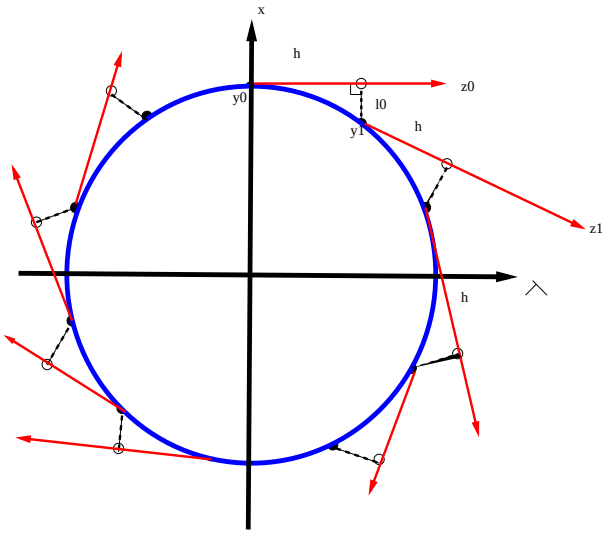

Figure 1. A natural parameter homotopy for a circle

In Figure 1, let h be the step size of a predictor step from iterations of pseudo-arclength continuation and h $\ll$ 1. In order to find all turning points, where the tangent vector keeps

changing, on the circle, we first set a starting point y_0 on the circle, and draw a tangent line z_0 through y_0. Starting from y_0, we take a step h on the tangent z_0 and then stop. At the point $y_0 + hz_0$, we draw a line l_0 perpendicular to the tangent z_0 and make the line intersect the circle at the point y_1. We repeat the step and go around the circle in this fashion. This is also called the predictor-corrector method for finding turning points on the circle according to the step size without any limitation on the natural parameter λ. Every point on the circle has the potential to be a turning point with a small step size.

Natural parameterizations with pseudo-arclength continuation are described in [26] to compute a solution branch of a polynomial equation. Also in a natural parameter homotopy setting, general numerical methods are proposed in [17] to find turning points for planar polynomial systems.

1.2.1 Related work on global methods

Safey El Din and Schost described an algorithm called "ConnectedComponents" to compute at least one point in each connected component of a real algebraic variety in [75] and [74]. One of the subroutines of "ConnectedComponents" detects critical points using a strongly normalized triangular set. Their methods are mainly symbolic. Lazard and Rouillier presented a new algorithm [45] for finding critical points by using global symbolic methods:

$$C = \{x \in \mathbb{C}^n, p_1(x) = 0, \ldots, p_s(x) = 0, f_1(x) \neq 0, \ldots, f_l(x) \neq 0\} \tag{1.5}$$

or

$$S = \{x \in \mathbb{R}^n, p_1(x) = 0, \ldots, p_s(x) = 0, f_1(x) > 0, \ldots, f_l(x) > 0\}, \tag{1.6}$$

where p_i, $f_i \in Q[U, X]$, Q is a polynomial ring, $U = [U_1, \ldots, U_d]$ is the set of parameters, and $X = [X_{d+1}, \ldots, X_n]$ is the set of unknowns.

Lu, Bates, and Sommese [61] proposed an algorithm based on homotopy continuation and numerical irreducible decomposition to compute the real points of the irreducible one-dimensional complex components of the solution sets for polynomial systems with real coefficients. For each pair of distinct complex conjugate components, the isolated points in the intersection of the pair can be found by using a diagonal homotopy [80] [78]. Dealing with components of multiplicity greater than one, they use a deflation method to effectively replace such components with one-dimensional components of multiplicity-one .

1.2.2 Related work on local methods

General methods for finding critical points using path following methods can be found in [26], [63], [2], [43], and [29]. These methods are often numeric.

The idea of characterizing a singular point by an extended regular system, which can be treated numerically by a Newton type method, was proposed in [1], [2], [41], and [43]. This allows one to compute simple bifurcation points by choosing optimal parameters. The precondition of a minimally extended system is defined as follows:

Definition 1.2.3 Suppose $f(\mathbf{x}, \lambda) = 0$ is a multivariate polynomial system, $\mathbf{x} \in \mathbb{R}^n$ is a vector of n variables, $\lambda \in \mathbb{R}$ is a specific *control variable* denoted as a *bifurcation parameter.* Of special interest are *singular points* $(\mathbf{x}^*, \lambda^*)$ of the solution manifold S,

$$S = \{(x, \lambda) | f(\mathbf{x}, \lambda) = 0\}, \tag{1.7}$$

where the rank of the $n \times (n+1)$ Jacobian matrix is $< n$.

Definition 1.2.4 *Simple bifurcation points* can be characterized by a *minimally extended system* G, as follows:

$$G(\mathbf{x}, \lambda, \mu) = \left\{ \frac{f(\mathbf{x}, \lambda) + \mu d}{f(\mathbf{x}, \lambda)} = 0 \right\}, \tag{1.8}$$

where $d \in \mathbb{R}^n$ and f: $\mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}^2$ are supposed to be chosen such that $Z^* = (\mathbf{x}^*, \lambda^*, 0)$ is a *regular solution* of G.

This method works well under the condition that the starting point is close enough to the simple bifurcation point to be found. If the starting point is far away from the simple bifurcation point to be found, then there is no guarantee that the Newton type method will converge to the simple bifurcation point.

Li and Wang's paper [57] contains a discussion on real homotopy methods to compute quadratic turning points numerically. When a real homotopy method is used for solving a polynomial system with real coefficients, some homotopy paths may inevitably bifurcate at singular points. Li and Wang showed that the solution set of a real homotopy contains no singular points other than quadratic turning points. At a quadratic turning point, the bifurcation is simple:

the tangent vectors of the two bifurcation branches are perpendicular to each other. In [60], Li, Zeng, and Cong proposed efficient algorithms to calculate quadratic turning points locally. Later on, Li and Wang introduced the notion of a kth order turning point with k bifurcation branches [58] as follows:

Definition 1.2.5 A point $(\mathbf{x}^0,\, t^0) \in \mathbb{C} \times \mathbb{R}^1$ is a *kth order turning point* of $f(\mathbf{x}, t) = 0$, if

(i) $f(\mathbf{x}^0, t^0) = 0$;

(ii) rank $D_{\mathbf{x}} f(\mathbf{x}^0, t^0) = n - 1$;

(iii)$D_t f(\mathbf{x}^0, t^0)$ is not in the range of $D_{\mathbf{x}} f(\mathbf{x}^0, t^0)$ such that $Df \equiv [D_{\mathbf{x}} f, D_t f]$ is of real rank $2n - 1$ at $(\mathbf{x}^0, t^0)$;

(iv) there are exactly k solution curves of $f(\mathbf{x}, t) = 0$ passing through $(\mathbf{x}^0, t^0)$.

t is the continuation parameter. D stands for a partial derivative.

1.3 Problem statement

Our focus in this problem is not on natural parameter homotopies but on artificial parameter ones. Let's look at how an artificial homotopy works on the same circle as that in Figure 1.

In Figure 2, we apply an artificial parameter homotopy to conduct what we call a "sweep." In order to conduct a "sweep" on a polynomial system, we add the artificial-parameter homotopy equation to it. The idea is to force the artificial parameter to sweep critical points in a certain

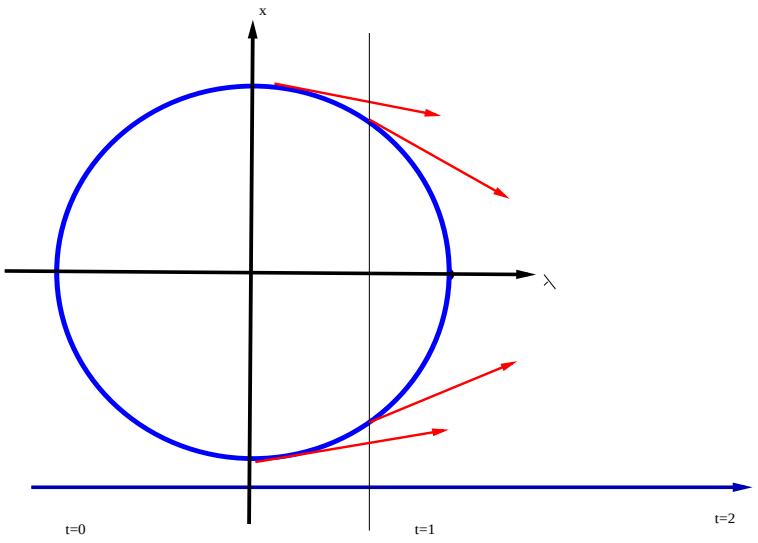

Figure 2. An artificial parameter homotopy finding quadratic turning points for a circle: vertical line sweeps

range. For instance, the homotopy to execute the sweep for t going from 0 to 1 and λ going from -1 to 1 on a unit circle is defined as

$$h(\mathbf{x}, \lambda, t) = \begin{cases} x^2 + \lambda^2 - 1 = 0 \\ (1-t)(\lambda+1) + t(\lambda-1) = 0. \end{cases} \tag{1.9}$$

In Figure 2, we can use the vertical line sweep to move t from 0 to 1, as the parameter λ traces from -1 to 1. The last equation simplifies to $\lambda - 2t + 1 = 0$. At $t = 1$, the sweep meets a quadratic turning point at (1,0).

In general, the homotopy to execute a sweep for t going from 0 to 1 and λ going from λ_0 to λ_1 is defined as

$$h(\mathbf{x}, \lambda, t) = \begin{cases} f(\mathbf{x}, \lambda) = \mathbf{0} \\ (1-t)\lambda_0 + t\lambda_1 = 0. \end{cases} \tag{1.10}$$

With the predictor-corrector method we can track any solution paths $(\mathbf{x}(t), \lambda(t))$, defined by $h(\mathbf{x}, \lambda, t) = \mathbf{0}$, accordingly. The sweep starts at any regular real or complex solution of $f(\mathbf{x}, \lambda_0) = \mathbf{0}$ and stops at $t = 1$, at any (possibly singular) solution of $f(\mathbf{x}, \lambda_1) = \mathbf{0}$, or at a value $t = t^*$ for which the corresponding λ^* leads to a corresponding singular solution of $f(\mathbf{x}, \lambda = \lambda^*) = \mathbf{0}$, i.e., for $\lambda = \lambda^*$, there is a corresponding $\mathbf{x}^*$ for which the Jacobian matrix is singular.

To compute the tangent vector τ to a solution path of $f(\mathbf{x}, \lambda)$ at a given point, we first solve a linear system defined by

$$\begin{bmatrix} \frac{\partial h}{\partial \mathbf{x}} & \frac{\partial h}{\partial \lambda} & \frac{\partial h}{\partial t} \end{bmatrix} \begin{bmatrix} \tau_{\mathbf{x}} \\ \tau_{\lambda} \\ 1 \end{bmatrix} = \mathbf{0}. \tag{1.11}$$

As in general, this is a system of $n + 1$ equations in $n + 2$ variables. We first fix the last component of τ to one. The solution τ to this linear system is then divided by its length, i.e., $\tau := \tau / ||\tau||_2$, using the 2-norm $||\cdot||_2$. Lastly, we determine the orientation of τ, computing the inner product of τ with the previous tangent vector. If that inner product is less than zero, the orientation of the tangent vector flipped and we passed a quadratic turning point [60].

Once we have detected that we passed a quadratic turning point, we return to the previously computed point along the path and decrease the step size. Once an optimal step size is found either by bisection or extrapolation formulas using shooting – a correction method using an extra hyperplane perpendicular to the tangent vector. The correction method has no numerical difficulties computing the quadratic turning point accurately. However, if we take big step sizes and general singularities into consideration when trying to find multiple critical points, things may have become more complicated.

In order to detect higher-order singularities, we use the method of "Parabolic Interpolation of Determinants" to detect singularities and Puiseux series [16] [89] to iterate close to them when Newton's method fails. We will discuss how to detect higher-order singularities in detail in chapter 3.

After detecting singularities, we can use deflation [51] to localize and recondition them. The hard part is that the starting points have to be close enough for deflation to work.

1.4 Building an interface to PHCpack in Python

Our first encounter with Sage [82] during the IMA workshop on Software for Algebraic Geometry (October 23-27, 2006) got us interested in experimenting with Python. The primary implementation language of Sage is Python. Sage is free, open-source math software that supports research and teaching in algebra, geometry, number theory, cryptography, numerical computation, and related areas. Both the development model and the technology in Sage are distinguished by "an extremely strong emphasis on openness, community, cooperation, and collaboration: we are building the car, not reinventing the wheel." [82] The overall goal of Sage

is to create a viable, free, open-source alternative to Maple, Mathematica, Magma, MATLAB, and many other mathematical packages. PHCpack [87] is now an experimental interface in Sage.

In order to build an efficient, yet flexible, interface to PHCpack, we decided to extend the C library routines in PHCpack to Python. The C library routines serve as a programming interface to PHCpack and a gateway to the Ada routines. Based on the layout, we wrote a Python extension by building a dynamic library. Since the support modules are written in C and hidden under a thin interpreted Python layer, applications respond well because more CPU cycles are spent in fast C code and fewer are spent in spinning the Python interpreter's wheel.

We also created a Python program (module) to call the dynamic library. We now show basic commands to make use of the dynamic library. Let's first take a look at how to call the blackbox solver of PHCpack in Python. For example, consider the system $f(\mathbf{x}) = 0$

$$f(x_1, x_2) = \begin{cases} 1.2x_1^2 + 3.9x_2^2 - 8.3 + 5.9i = 0 \\ 3.1x_2^2 - 3.9x_1 = 0. \end{cases} \tag{1.12}$$

In the Python shell, the input to the blackbox solver is a list of strings of multivariate polynomial equations with real or complex coefficients. The output is a list of solution strings in Python.

```
>>> from phcpy import *
>>> sol = solve([‘‘1.2*x1^2+3.9*x2^2-8.3+5.9*i;’’,‘‘3.1*x2^2-3.9*x1;’’])
>>> n = len(sol)
```

```
>>> print n

4

>>> print sol[0]

t :  1.00000000000000E+00   0.00000000000000E+00

m : 1

the solution for t :

x1 : -5.45262494233056E+00   7.21284774561043E-01

x2 : -1.72854880214128E-01  -2.62481099687578E+00

== err :  7.953E-16 = rco :  1.543E-01 = res :  1.221E-15 =
```

In the above example, before invoking the blackbox solver, we use the Python command "from phcpy import *" to import the Python module "phcpy.py." The solve() command calling the blackbox solver is a function definition in "phcpy.py." The solve() function returns a list of solution strings stored in sol. The number of solutions is thus the length of the list of solution strings. We use the Python built-in command len() to get the length of sol. In this example, there are four solutions in the list of solutions strings. We can show the first solution in sol by printing out the first element in the list, that is, sol[0].

Unlike some other interpreted languages, Python comes with an extensive library of useful modules such as the socket module, the threading module, and the database module. To utilize the sockets and the threads supported by Python, we created a multithreaded client/server application to solve many polynomial systems concurrently from a server to multiple clients.

Since we have to solve large polynomial systems, we inevitably have to deal with a huge collection of data, i.e., millions of solution sets for one system, multiple instances of one system, or solution paths that are very hard to organize in just one file. All these difficulties could cause an overflow of files or cause the files to lose structure. A possible solution is to create tables to store data in a mySQL database. The mysqldb module, which comes with Python, helps programmers write MySQL scripts with Python DB-API.

1.5 Applications: random walks

Numerical homotopy continuation methods are powerful tools to find solutions for a polynomial system. These methods can also be useful in pure Mathematical research. For instance, numerical homotopy methods can be used to construct monodromy groups, see [76] and [49]. Leykin and Verschelde predict a monodromy breakup by generating loops around singularities to exploit the monodromy group action [49]. Generating loops is done by homotopies. They compute more samples of a k-dimensional solution set, starting from a list of well-conditioned solutions for $f(x)$ in the witness set W_L and moving L to another set of k random hyperplanes. They then join points on the same irreducible components in the monodromy loops. To certify whether they have found enough points, they use the linear trace test [49] as the stopping criterion for the monodromy breakup algorithm.

Stewart-Gough platform is a mechanism that is used as flight simulators. It consists of a top platform supported from a base platform by six extensible "legs," see Figure 9 in chapter 4 as a special case. The forward problem is to determine the position of the top platform if the location of the base platform and the lengths of the legs are given. Dietmaier's numerical

method [15], which systematically changes the parameters of a given general Stewart-Gough platform to obtain 40 real positions, also involves random walks. To start the numerical procedure, Dietmaier presents an arbitrary chosen set of 29 parameters of the polynomial system for a general Stewart-Gough platform [15]. He first computes all real and complex solution vectors, which satisfy the 9 equations defining the platform. Secondly, he calculates each solution vector by picking a vector with 9 random complex elements and then improving this estimate by making use of a combination of the Newton-Raphson and the steepest decent methods [15]. This task is repeated until all 40 real solutions are found.

The concept of random walks links the Python interface to the "sweep" algorithm. We are in the process of implementing the "sweep" function in the Python interface. After the interface is built, we will be able to "sweep" different paths concurrently using the client/server routine.

1.6 Contributions of this thesis

This thesis makes several contributions. First, we extended PHCpack to Python by building a dynamic library and making good use of the existing socket module, threading module, and database module in Python. Secondly, in Chapter 2 we will describe the flexibility and capability of the python interface, such as a client/server application to concurrently solve many polynomial systems. Thirdly, in Chapter 3 we will explain the methods used to find critical points, including higher-order singularities, in great detail. In the last chapter, we will show all the numerical results.

CHAPTER 2

A PYTHON INTERFACE TO PHCPACK

2.1 Why Python?

Python was conceived in the late 1980s [86] by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in the Netherlands. Python was first designed as a successor to the ABC programming language, capable of exception handling and interfacing with the Amoeba operating system [73]. Python is an interpreted and object-oriented programming language and comes with a large standard library that covers areas such as list expressions, string processing, abstraction, database management, software engineering, and operating system interfaces like TCP/IP sockets.

Rather than forcing programmers to adopt a particular style of programming, Python permits several styles, such as object-oriented and structured programming. Python uses dynamic typing, a combination of reference counting, and a cycle detecting garbage collector for memory management. An important feature of Python is dynamic name resolution, which binds methods and variable names during program execution.

Another target of the language's design is ease of extensibility, rather than having everything built into the language core. New built-in modules can easily be written in C or C++. Python can also be used as an extension language for existing modules and applications that need a

programming interface. This design, a small core language with a large standard library and an easily-extensible interpreter, was intended by Guido van Rossum from the very start [86].

The main reason why we chose to build a Python interface to PHCpack is because Python has the capability of extending C/C++, represent lists of strings nicely, communicate between a remote server and its clients, and organize large data into a database. Not to mention that Python is an excellent scripting language and very user-friendly.

2.2 The layout of PHCpack and the C library

PHCpack was written in Ada, a structured, statically typed, imperative, and object-oriented high-level computer programming language. Ada is strongly typed and compilers are validated for reliability in mission-critical applications, such as avionics software. Another feature of Ada is the use of the command "pragma" to make use of mixed language programming.

All Ada source code is in the directory called Ada as a backbone of PHCpack. The C library routines in the Lib directory serve as a gateway to call Ada functions. It was first developed as an MPI library by Prof. Jan Verschelde. MPI, Maple, and Python are directories, which represent components of the outmost layer of PHCpack, containing an MPI library, a Maple interface, and a Python interface using C library routines. PHClab [32] is a MATLAB/Octave interface to PHCpack. PHCmaple [48] is a Maple interface to the numerical Homotopy algorithms in PHCpack. PHCpy is a Python interface to PHCpack. The C library is like a bridge that connects the above interfaces to the inner layer of Ada code and hides the Ada code from users. See Figure 3 for the layout of PHCpack. "Ada," "Lib," "MPI," "Maple," and "Python"

are major components of PHCpack with "Ada" being the core of PHCpack, "Lib" being the gateway, and the rest of the components calling Ada functions through "Lib."

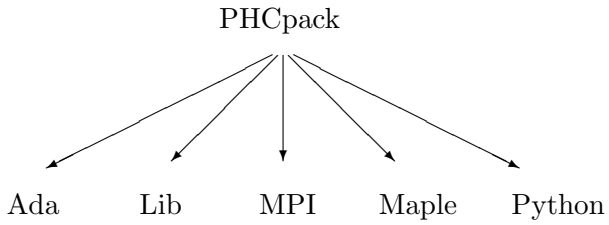

Figure 3. The layout (directories) in PHCpack

The C library interface uses PHCpack as a state machine described in [92], [50]. It feeds data into the state machine and selects methods to proceed; it computes results with user-given data and methods; and it extracts the results from the state machine. The C library contains modules and containers as abstraction of data types.

The C library acts like a gateway to call Ada routines from C. The function calls "pragma import" and "pragma export" can be used to associate function names in Ada with those in C and provide support with C types [92]. This type of interface is efficient, since it lets C programs prepare an input and call the Ada routines. It also hides the Ada code from users so that they do not need to have any knowledge of Ada. In order to connect C with Ada, a procedure called "use_c2phc" was constructed by Prof. Jan Verschelde. The Ada procedure consists of the following major routines [92]:

1. use_syscon : This provides a gateway to a system container. It reads a polynomial system, puts it into a system container, gets the dimension of the polynomial system, and finally returns the number of the terms of the i-th polynomial, the complex coefficients, the exponents vectors of the polynomial system, and so on.

2. use_solcon : This provides a gateway to a solution container. It reads the solutions from a file, puts the start solutions in a solution container, returns the dimension and the multiplicity of the target solutions and the target solution arrays, and finally appends the target solutions to the solution container.

3. use_celcon : This provides a gateway to a cell container that contains data structures for mixed-volume computations.

4. c_to_phcpack : This provides a gateway to the state machine. It reads and writes a start polynomial system and a target polynomial system, reads the solutions for the start system, solves the system by using homotopy continuation, computes and writes the target solutions for the target system, and finally defines an output file to store the solutions and other information.

We can call the C library functions by adding a gateway routine to the beginning of a C program as follows:

```
extern void adainit(void);

extern int _ada_use_c2phc ( int task, int *a, int *b, double *c );

extern void adafinal( void );
```

2.3 Building a dynamic library

It was a design problem at first regarding how to create an efficient Python interface to PHCpack. For the sake of manageability, we decided to create an interface in Python using PHCpack as a state machine. The C library in PHCpack can be extended by Python and built into a dynamic library.

We will describe in this section how we wrote a module in C to extend C programs with Python in a new module. The module can define not only new functions but also new object types and new methods. We will also describe how to link Ada routines to the extension module. Finally, we will show how to compile and link the extension module so that it can be loaded dynamically (at run time) into the interpreter on Unix.

Since the C library in PHCpack serves as a gateway to the main Ada routines, we managed to extend the major C library routines to Python by writing an extension module called "phcpy2c.c." Such an extension module can do two things that can't be done directly in Python: they can implement new built-in object types and they can make C library function calls and system calls. To support such an extension, the Python API (Application Programmers Inter-

face) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system.

The Python API is incorporated into "phcpy2c.c" by including the header "Python.h." "Python.h" includes a few standard header files: <stdio.h>, <string.h>, <errno.h>, and <stdlib.h>. We then add C functions that will be called from Python. Ada routines also need to be initialized in the extension module by calling adainit().

An example of a fragment of "phcpy2c.c" to extend C with Python is shown as follows:

```
A routine in the C library:
int syscon_store_polynomial ( int nc,int n, int k, char *p)
{ ...
    fail = _ada_use_c2phc(76,a,b,c);
    return fail;
}

A Python wrapper over the C routine:
static PyObject *py2c_syscon_store_polynomial( PyObject*self, PyObject *args )
{...
    if(!PyArg_ParseTuple(args,‘‘iiis’’, &nc, &n, &k, &p))
       return NULL;
    fail = syscon_store_polynomial(nc,n,k,p);
    return Py_BuildValue(‘‘i’’,fail);
}
```

The function "syscon_store_polynomial (int nc, int n, int k, char *p)" calls an Ada routine to store polynomials in a system container. It reads three integers and a pointer to a character as parameters and returns a boolean integer to indicate whether the function is executed correctly or not. "py2c_syscon_store_polynomial(PyObject *self, PyObject *args)" is a Python wrapper that flexibly "wraps" around "syscon_store_polynomial (int nc, int n, int k, char *p)." If there is any future change made to "syscon_store_polynomial (int nc, int n, int k, char *p)," there will be little change necessary to modify the Python interface.

After doing all the necessary steps to build an extension to the C library, we need to compile the source code and then link it with the Python system. We can create a dynamic library by compiling and linking the C library files, the Python extension file, and the Ada gateway routines by using the command "gnatlink -shared ... ". The gnatlink command links all the object files together to build a shared library called "phcpy2c.so."

2.4 Solving polynomial systems

In this section, we are going to call the dynamic library we built in the previous section. In order to organize the calling (testing) routines to call the dynamic library, we created a Python program (module) called "phcpy.py". Just like a subdirectory of PHCpack called MPI, which has a collection of MPI routines calling the C library, we created a subdirectory called Python, which has a collection of Python routines calling the C library. The "phcpy.py" module has several function definitions such as the blackbox solver to solve a polynomial system, the mixed volume calculator [22] to find the number of solutions for a polynomial system without solving it, and the track function to track solution paths for a polynomial system.

The blackbox solver is the most frequently used program in PHCpack and is called by the solve(L) function in the Python module "phcpy.py." The solve function takes a list of polynomial strings as the input polynomial system and outputs a list of solution strings as the solution sets returned by the blackbox solver. We have already shown how to run solve(L) in a Python shell in Chapter 1.

Now let's look at the track function in "phcpy.py." Path tracking usually starts from a generic system (without any singular solutions) to a more specific system. We use the system that we solved by using the blackbox solver as a start system. We also use the list of solution strings returned by calling the blackbox solver as the list of the start solution strings. Before calling the function "track(target,start,sols)," we must set the value of t (time) in every solution to zero so that the start solution strings contain the proper start solutions. The function "track(target,start,sols)" takes a list of strings representing a target system, a list of strings representing a start system, and a list of strings representing the solution sets.

Definition 2.4.1 The *condition number* of a solution can be defined as the ratio of the relative change in the solution to the relative change in the input. Let $\hat{x}$ be a solution in the solution set and let C denote the *condition number.*

$$C = \frac{|(f(\hat{x}) - f(x))/f(x)|}{|(\hat{x} - x)/x|} \tag{2.1}$$

Let's call "track(target,start,sols)" in a Python shell using a similar example we have shown for solve(L) in Chapter 1. This time we are going to track the solution paths from $x = 1$ to $x = 2$. Let "target" be the target system as follows:

$$f(x_1, x_2) = \begin{cases} 1.2x_1^2 + 3.9x_2^2 - 8.3 + 5.9i = 0 \\ \\ x_1 - 2; \end{cases} \tag{2.2}$$

Let "start" be the start system as follows:

$$f(x_1, x_2) = \begin{cases} 1.2x_1^2 + 3.9x_2^2 - 8.3 + 5.9i = 0 \\ \\ x_1 - 1; \end{cases} \tag{2.3}$$

Let sol be the start solutions, which can be obtained by calling solve(L).

In a Python shell, we call track(target, start, sols) and display the first solution as follows:

```
>>> from phcpy import *
>>>  ...enter here the target system, the start system,
>>>  and the solution strings before calling track()...
>>> T = track(target, start, sols)
>>> n = len(T)
2
>>> T[0]
t :  1.00000000000000E+00   0.00000000000000E+00
m : 1
```

```
the solution for t :
 x1 :  2.00000000000000E+00   0.00000000000000E+00
 x2 :  1.15247944274769E+00  -6.56332970770270E-01
== err :  1.171E-16 = rco :  4.551E-02 = res :  8.882E-16 =
```

In the above example, track(target, start, sols) takes the list of the target system, the list of the start system, and the list of the solutions for the start system and returns a list of solution strings stored in T. The length of T is 2, which means that the target system has two solutions. We show the first solution in T by printing out the first element in the list, that is, T[0].

The diagnostics shown for T[0] are explained as below [32]:

t is the end value of a continuation parameter. If this value does not equal one, then it means that the path tracker did not manage to reach the end of the path. This might happen to a path diverging to infinity or highly singular solutions.

m is the multiplicity of a solution. A solution is regular when the multiplicity is one. Even when the approximation for a solution is not accurate enough, the multiplicity might still be one, possibly with the value for ***rco*** approaching a threshold.

err is the magnitude of the last updated Newton's method evaluated for a solution. At singularities, polynomial functions exhibit "flat" behavior. Although the residual for a singular solution may be small, the value for ***err*** can still be big.

rco is an estimate for the inverse of a condition number of the Jacobian matrix evaluated at an approximate solution. A solution becomes singular when ***rco*** drops below the threshold value 10^{-8}, because it has lost half of precision of a typical tolerance, 10^{-16}. The condition

number C of the Jacobian matrix measures a forward error, i.e., if the coefficients are given with a precision of D digits, then the forward error on the approximated solution can be as large as C $\times 10^{-D}$.

res is the residual, or the magnitude of a polynomial system evaluated at an approximated solution. The residual measures a backward error: how much we should change the coefficients of a given system to approximate a computed solution to an exact solution.

The function mixed_volume(L) in "phcpy.py" calculates the mixed volume [22], which is a bound for the number of solutions for a polynomial system. It takes a list of polynomial strings as the input and outputs an integer to represent the mixed volume without solving the polynomial system. The mixed volume equals the number of roots without zero components of a polynomial system with generic coefficients [56]. In a Python shell, the input to the mixed volume calculator is also a list of strings of multivariate polynomial equations with real or complex coefficients. And the output of the mixed volume calculator is an integer, which represents the mixed volume of the input polynomial system. We call the "mixed_volume" function from Python as follows:

```
>>> mixed_volume([‘‘1.2*x1^2+3.9*x2^2-8.3+5.9*i;’’,‘‘3.1*x2^2-3.9*x1;’’])
>>> 4
```

2.5 Solving many polynomial systems

We have described how to use the blackbox solver to solve one polynomial at a time in Chapter 1. But what if we have many polynomial systems to solve concurrently? Having a

Python interface to PHCpack provides us with a suitable solution because Python is a powerful tool for network programming.

Related work include the use of Dsage, distributed Sage, in Sage. Dsage is a distributed computing framework suitable for coarse distributed computations. DSage works well on the problems that can take advantage of coarse grained distributed computing.

There are many preexisting libraries for common network protocols with various layers of abstractions on top of them. There are plenty of networking modules available in the standard library. A basic component in network programming is the socket. A socket is basically an "information channel" with a program on both ends - the analogy is to a wire (the network data connection) being plugged into a socket. The programs may be on different computers connected through a network and may send information to each other through sockets. Most network programming in Python hides the basic workings of the socket module and does not interact with the sockets directly.

Sockets come in two varieties: server sockets and client sockets. After a server socket is created, it can be told to wait for connections from client sockets. A client socket can be connected to a remote server socket via an open data connection. Closing the connection destroys both sockets at each endpoint. A socket is not a port, though there is a close relationship between them. A socket is associated with a port, though this is a many-to-one relationship. Each port can have a single server socket, awaiting incoming connections, and multiple client sockets, each corresponding to an open connection on the same port. In Python, a socket is an instance of the socket class from the socket module.

Based on the "one server and many clients" strategy, we created a network application called "phcwulf.py." After a server is connected by calling "phcwulf.py," it will then listen at a certain network address, which is a combination of an IP (Internet Protocol) address and a port number, until a client socket connects. The two can then communicate. The main program in "phcwulf.py" provides the user with a choice to start a server or a client. Dealing with the client sockets is usually quite a bit easier than dealing with the server side because the server has to be ready to deal with clients whenever they connect or may deal with multiple clients, while the client simple connects, does some work, and disconnects.

"phcwulf.py" includes code for the server side and the client side. For transmitting data, sockets have two methods, ***send*** and ***recv***(for "receive"). We can call ***send*** with a string argument to send data and ***recv*** with a desired number of bytes to receive data.

On the server side, the server script must import the network-oriented modules such as the socket module to prepare sockets and the threading module to create multiple threads. The server socket uses its ***bind*** method followed by a call to ***listen*** to a given address. The address is just a tuple of the form (host,port), where host is a host name and port is a port number (an integer). Once a server is listening, it can start accepting clients. This is done using the ***accept*** method. The server generates random polynomial systems, sends the list of polynomial systems to available clients, and gets a list of solutions back from each client.

On the client side, the client can connect to the server by using its ***connect*** method with the same address as used in ***bind*** on the server side. We need to import the "phcpy.py" module

to use the blackbox solver. A client accepts a list of polynomial strings, invokes solve(L) in "phcpy.py," and then returns the list of solution strings to the server.

To sum up, the client/server module, "phcwulf.py," accomplishes the goal of solving many polynomial systems concurrently via heterogenous distributed computing.

2.6 The potential of the Python interface

The future of the Python interface looks bright. Based on our existing interface, we will be able to include more functions such as using sockets as an alternative to MPI to meet the goal of heterogeneous computing, improving the user interface on the client side with a GUI (Graphical User Interface) and on the server side with a GUI load distribution, and making use of the Python DB API to create a MySQL database to store collections of solutions.

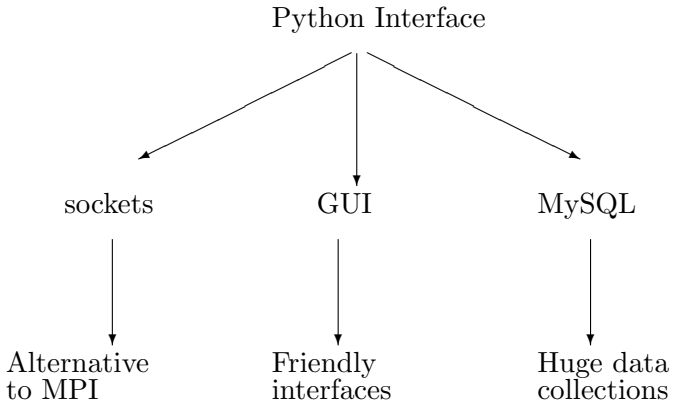

Figure 4. The potential of the Python interface

CHAPTER 3

COMPUTING CRITICAL POINTS BY CONTINUATION

One of the most important and recurring problems in applications is finding critical points of polynomial systems with parameters. In order to compute critical points of polynomial systems, one may apply global or local methods. As for our research, we developed a new algorithm to detect singularities along paths. We will first offer a quick introduction to several available global and local methods and then an extensive discussion on our new algorithm for locally detecting and locating singularities including higher-order singularities.

3.1 Detection and computation of singular points

A global approach to our problem is to apply the so-called Jacobian criterion. Introducing n auxiliary multiplier variables $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, the condition that a solution is singular can then be expressed as augmenting the original system with $n + 1$ additional equations:

$$F(\mathbf{x},\lambda,\mathbf{y}) = \begin{cases} f(\mathbf{x},\lambda) = \mathbf{0} \\ \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \mathbf{0} \\ c_1 y_1 + c_2 y_2 + \cdots + c_n y_n = 1. \end{cases}$$

The constants $c_1, c_2, \ldots, c_n$ in the last linear equation are randomly chosen complex numbers. When applied locally, the Jacobian criterion works well to find isolated singularities. The Jacobian criterion (applied in [61]) offers a satisfactory global solution for modest dimensions. However, as the dimension of the problem roughly doubles, the complexity of solving $F(\mathbf{x},\lambda,\mathbf{y}) = \mathbf{0}$ limits the practical applicability of this approach.

Our problem can be solved globally via the Jacobian criterion by augmenting the system with the information in the Jacobian matrix. However, any global method will suffer from the complexity of all singularities. Before we switch our attention to a local approach, we point out that the systems that arise in applications of the Jacobian criterion are excellent test problems for polynomial system solvers. Exploiting their structure is a very interesting line of research. Some symbolic methods are discussed in [75] [45].

The aim of our research is to detect and locate critical points efficiently by using a local method. Considering a polynomial system with a parameter, we use an artificial parameter homotopy to "sweep" critical points, which is appropriate for the problems we are interested in. We would like to know for which values of the parameter singularities happen.

The local approach is to start at some point along a solution path and track the path, monitoring for occurrences of singular solutions and then adjusting step sizes accordingly in order not to miss any singular solutions.

The input to our problem is a polynomial system with a parameter. The coefficients of this system will more likely be real than complex. But in general we consider them as approximated coefficients, given only with limited precision. Furthermore we assume that we can compute solutions for generic instances of the parameter. We are looking for a general algorithm to detect and locate critical values for the parameter, i.e., for which values of the parameter do the solutions become singular?

At a quadratic turning point, the two real solution paths become complex conjugate. Viewed from the opposite direction, the two complex conjugate solution paths join at a double solution and then become real. Quadratic turning points are detected by monitoring the direction of the tangent vector and computed via a shooting method. The useful description in [60] guided our implementation.

However, when we consider polynomial systems depending on only one parameter, the only random coefficients that may occur in the system stemmed from making random choices of the parameter. In this case, for the systems we tested, we found that quadratic turning points are

not the only occurring singularities. Studying the polynomial systems with real parameters, we encountered the following two difficulties:

1. Straight solution paths pass swiftly through a multiple point so that the value of the parameter for which the multiple solution occurs is not detected.
2. Higher-order singularities and more than corank one deficiencies of the Jacobian matrix prevent the predictor-corrector method from marching close enough to the singularity.

For the second type of difficulty, we can apply the so-called *endgames*, see [80], by increasing the working precision. The first difficulty is called *jumping over a singularity*, see [63, page 79]. This should not be confused with *path jumping*. *Path jumping* occurs when a path tracker loses track of the current path and computes approximations belonging to another path. The problem of path jumping is covered in [37].

The focus of our research is to experiment with particular examples of polynomial systems with parameters arising in Science and Engineering from which we have encountered the problem of jumping over a singularity. In order to solve this problem, we have mainly two tasks to do:

1. Detection mechanism to detect singularities.
2. Location mechanism to locate singular points and compute them accurately.

3.1.1 Singularities along paths

Before we try to detect and compute singular points for a polynomial system with a parameter, let's take a look at general singularities along paths.

Based on our experiments, we roughly classify general singularities along paths as follows:

1. Singularities that can be detected via the sign change of consecutive tangent vectors, the determinant of the Jacobian matrix, or the eigenvalues of the Jacobian matrix.

2. Singularities that cannot be detected by the methods in 1, see the schematic plot in Figure 5 and the actual Maple plot in Figure 6.

Jumping over Hopf bifurcation points [33] is a good representation for detectable jumping-over singularities. Hopf bifurcation points are detectable via the sign change of the eigenvalues of the Jacobian matrix of a polynomial system. If the Jacobian matrix has an eigenvalue with zero real part and if the eigenvalue is non-zero but purely imaginary, then this is a Hopf bifurcation.

Regarding the case of undetectable jumping-over singularities, as we mentioned before, we only consider isolated singularities. We concentrate on how to detect and locate those singularities. Before discussing how to detect those singularities, let's look at jumping-over singularities in more detail.

A Newton's iterator is defined as:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - [J(\mathbf{x}_n)]^{-1} f(\mathbf{x}_n) \tag{3.1}$$

where $J(\mathbf{x}_n)$ is the Jacobian matrix of a polynomial system $f(\mathbf{x}_n)$.

A Case of jumping over a bifurcation point is shown as follows [63]:

- Due to straight curves, large step sizes, or general singularities, jumping over a bifurcation point can occur in many cases.

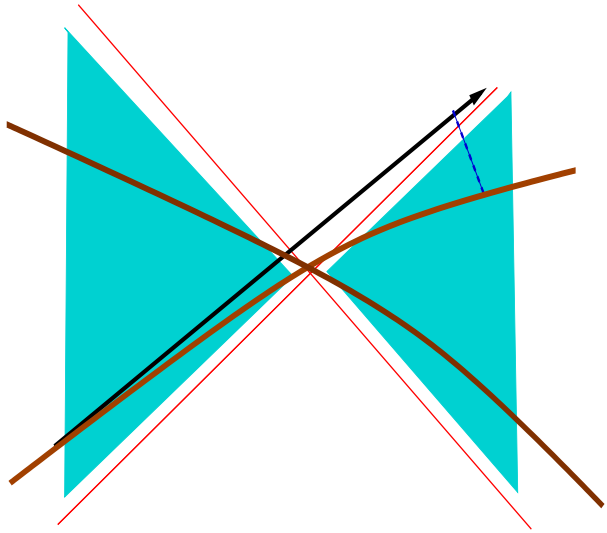

Figure 5. A schematic plot of jumping over a bifurcation point

- The shaded areas in Figure 5 are the regions where Newton's method converges. On a curve that is nearly straight, the path tracker will never cut back in size. It results in missing the bifurcation point in the center of the graph.

We also observed other occurrences of jumping over a critical point in our applications, e.g., see the plot of solution paths for Neural Networks [69] in Figure 6. The system of Neural Networks in the Noonberg [69] family of polynomial systems is defined as follows: for $n = 3$, the system is

$$f(\mathbf{x}, \lambda) = \begin{cases} x_1x_2^2 + x_1x_3^2 - \lambda x_1 + 1 = 0 \\ x_2x_1^2 + x_2x_3^2 - \lambda x_2 + 1 = 0 \\ x_3x_1^2 + x_3x_2^2 - \lambda x_3 + 1 = 0. \end{cases} \tag{3.2}$$

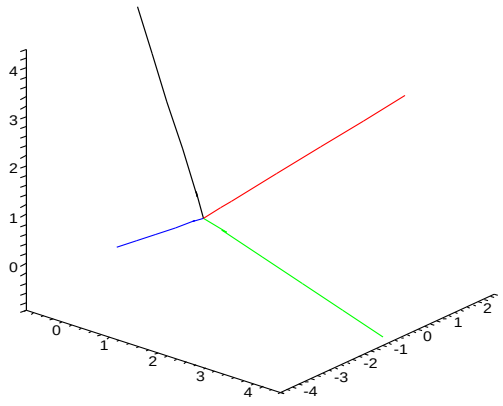

Figure 6. An actual Maple plot of a neural network of dimension three

In Figure 6, since the four solution paths are nearly straight lines, we will not be able to get any closer to the multiple point at the origin. Since the Newton's iterator fails to converge near the origin, deflation methods based on Newton's method will fail. In order to find isolated singularities of this kind, we need to develop an interpolation method to detect these isolated singularities. We will show how to use numerical methods to detect isolated singularities.

3.1.2 Detecting singularities

We assume that the step size h of a predictor step is sufficiently smaller than the distance δ between two singular points, which can be denoted as: $h \ll \delta$. The distance δ can also be called a cluster radius if one singular point is in the proximity of another singular point. If the

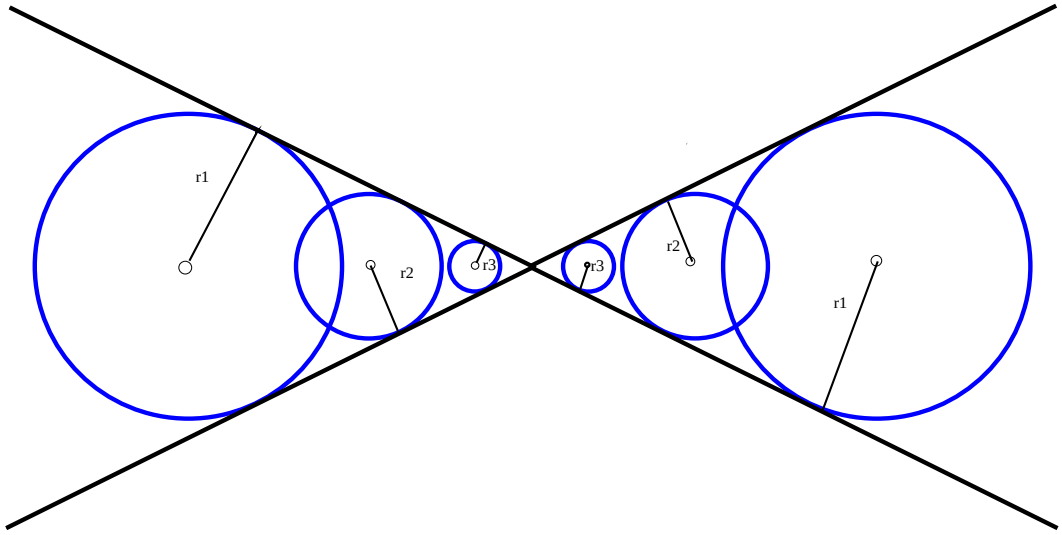

Figure 7. A schematic plot of radius of convergence of Newton's method along a path

cluster radius $\delta < 10^{-8}$, then we treat the two singular points as one singularity related to the working precision of a standard double float.

In Figure 7, we can visualize the situation of radius of convergence [5] of Newton's method. If the curve is too steep, the point of radius r_1 on the left might miss the singularity at the origin and jump right to the point of radius r_2 on the right.

There is research done to control the step size in order to prevent jumping over a critical point, see [37].

Monitoring the behavior of the eigenvalues and eigenvectors of the Jacobian matrices, as the path tracker gradually approaches a singularity, should provide a lot of information. But for our applications (described below), the extra information is not necessary to slow down the path trackers and prevent jumping over a singularity. So our assumption is monotonicity in the

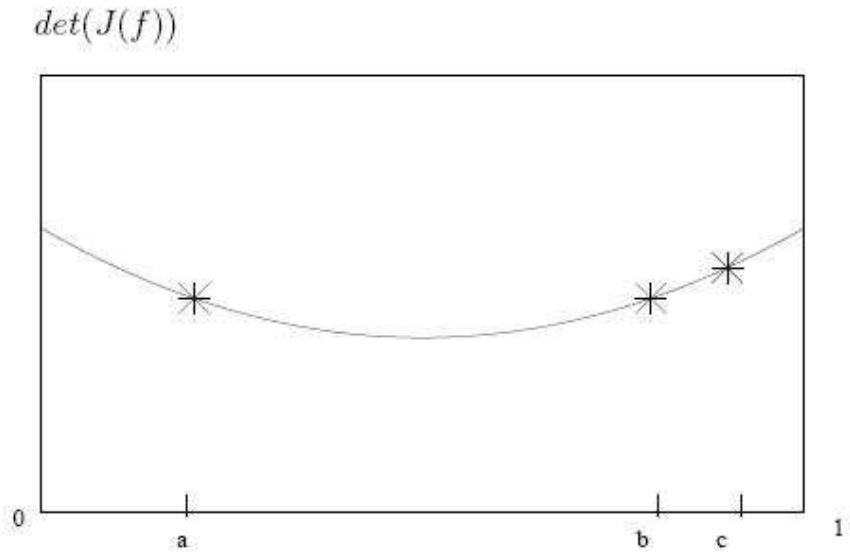

Figure 8. Parabolic interpolation of the determinant of the Jacobian matrix along a path.

sense of jumping over a critical point. We have the need to design an algorithm to deal with monotone determinants.

When the determinant of the Jacobian matrix does not change sign, we use the following two steps to detect and locate singularities along paths:

1. Detection: use the change of concavities of the parabolic interpolation of the determinant of the Jacobian matrix to detect existing singularities. If the concavities do not change, then continue tracking the path.

2. Location: after a singularity is detected in 1, we use the Golden Section Search method [68] to minimize the parabolic interpolation to locate the singular point.

The detecting criterion is efficient to detect singularities. If there are singularities lying in the parabola of the determinant of the Jacobian matrix, the concavity of the curve changes. As a result, we predict that there is a singular or a multiple point after we minimize the parabolic interpolation. A change in the observed concavity signals a reduction in the step size of the path tracker and a finer resolution of the range of the parameter values. In Figure 8, if point a and point c have the same concavity and if the concavity changes at point b, then we say that we have minimized the parabolic interpolation at point b. Our conclusion is that a singularity occurs somewhere near point b.

To approximate a local minimum point (the singular point we want), we can apply parabolic interpolation of the Jacobian matrix with the Golden Sections Search method. When the determinant of the Jacobian matrix approaches approximately 10^{-3}, we take the current point together with the next two consecutive points to form a parabolic and continuous downward and then upward trajectory. Our task is to find the minimum of the trajectory.

One possible solution is to use the Golden Section Search method [68]to locate the minimum. This method has the following advantages:

- The function of a Golden Section Search method is unimodal, meaning that the function monotonically increases or decreases on one interval or one subinterval. Consequently, the function has a minimum or a maximum inside the interval or the subinterval if the boundary points are not the minimum or the maximum.

- In contrast to a derivative method like a Newton's method, the starting iterate for the Golden Section Search method does not have to be close enough to the local maximum or minimum for convergence without interference.
- The Golden Section Search method requires only one new function evaluation per iteration.
- The Golden Section Search method always keeps the current approximated minimum or maximum trapped in an interval or a subinterval.
- If the resolution of a step size is small enough, the Golden Section Search method will give reliable results in trapping minima or maxima.

Require: $f = \{f_1, \ldots, f_N\}$, a finite set of polynomials in R with parameters, such that the determinant of the Jacobian matrix $J(f)$ at some point approaches 10^{-3} when a sweep is applied to find all critical values.

Ensure: Get the next point p_2, as soon as $J(f)$ approaches 10^{-3} at the first point p_1. Use the Golden Section Search method to find a local minimum between those two points in the parameter range [a,b].

Evaluate f at a fraction c (i.e., the Golden Ratio) of the end points:

$p_1 = b - c(b-a) = a + (1-c)(b-a)$

$p_2 = a + c(b-a) = b - (1-c)(b-a)$

Eliminate and reduce. Only one new function evaluation is needed at subsequent iterations. The search is terminated when the interval has reduced below a given tolerance, tol $< 10^{-8}$.

if $f(p_1) < f(p_2)$, then [a,b] => [a,p_2], with size reduction:

p_2 = (1-c)a + cb

if $f(p_1) > f(p_2)$, then [a,b] => [p_1,b], with size reduction:

p_1 = ca + (1-c)b

3.1.3 Reconditioning singularities

After locally detecting a singularity, we need to restore quadratic convergence for isolated singular solutions of polynomial systems. The validation of the results from our experiments consists of sufficiently accurate approximations for the critical values of the parameters so that deflation [51] can recondition the singularities. In [51], Leykin, Verschelde, and Zhao proposed a symbolic-numeric method to produce a new polynomial system, which has the original multiple solution as a regular root.

A singular root $\mathbf{x}^*$ of a square (i.e.: $N = n$) system $f(\mathbf{x}) = \mathbf{0}$ with Jacobian matrix $J(\mathbf{x})$ satisfies

$$\begin{cases} f(\mathbf{x}) = \mathbf{0} \\ \det(J(\mathbf{x})) = 0. \end{cases} \tag{3.3}$$

The augmented system (Equation 3.3) forms the basic idea for deflation. If $\mathbf{x}^*$ is isolated and corank$(J(\mathbf{x}^*)) = 1$, then $\mathbf{x}^*$ as a root for Equation 3.3 has a lower multiplicity.

In theory, $\det(J(\mathbf{x})) = 0$ (or maximal minors) could be used to form new equations. But this is neither good symbolically because the determinant is usually of high degree and leads to expression swell, nor numerically, as evaluating polynomials of high degree is numerically unstable.

Instead of using the determinant, on a system f of N equations in n variables, Leykin, Verschelde, and Zhao proceeded along the following three steps to form new equations [51]:

1. Let $r = \text{rank}(J(\mathbf{x}_0), \epsilon)$ for $\mathbf{x}_0 \approx \mathbf{x}^*$ and tolerance ϵ, $0 < \epsilon \ll 1$. For numerical stability, they compute the rank via a singular value decomposition (SVD) of the matrix $J = J(\mathbf{x}_0)$. The numerical rank r equals the number of singular values larger than the tolerance ϵ.

2. Let $\mathbf{h} \in \mathbb{C}^{r+1}$ be a random vector. For numerical stability, they generate random numbers on the complex unit circle. They use $\mathbf{h}$ as a scaling equation to obtain a unique vector in the kernel of the Jacobian matrix.

3. Let $B \in \mathbb{C}^{n\times(r+1)}$ be a random matrix with numbers on the complex unit circle. Using B, they form $C(\mathbf{x}) = J(\mathbf{x})B$. Notice that $C = [\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{r+1}]$ is an $N \times (r+1)$ matrix with polynomial entries.

In one deflation step, Leykin, Verschelde, and Zhao added the equations of $C(\mathbf{x}, \boldsymbol{\lambda})$ instead of $\det(J(\mathbf{x})) = 0$ to the system $f(\mathbf{x}) = \mathbf{0}$. They also added $r + 1$ extra variables λ_1, λ_2, ..., λ_{r+1}.

3.2 Puiseux series

The problem is that we need to get close enough to any singularities in order for deflation to work. We need to use a reliable method to get close enough to those singularities.

Definition 3.2.1 *A formal series* of the form $\sum_{n=m}^{\infty} a_n z^{n/k}$ where m and k are integers such that $k \geq 1$ is is called a *Puiseux series* or a *fractional power series,* see [16] and [89]. The k here is also called *the winding number* of *a Puiseux series.* Note that if $k > 1$, then $z^{n/k}$ could be

multivalued. One example of the use of such a power series is the *Puiseux parameterization* of one-dimensional complex analytic varieties.

There are two major uses of Puiseux series for our applications:

- higher order predictor at a regular point.
- verification of winding numbers at singular points.

The main reason to introduce Puiseux series is that we need it to do post-processing when the deflation method fails. In order to use Puiseux series to iterate through the multiple point for the Noonberg example in Figure 6, we created a Maple function called "myInterpolation(a,x,winding,shift)". On a solution path, "a" is a set of parameter values; "x" is a set of solutions; "winding" stands for the winding number of the Puiseux series for the Noonberg system; and "shift" is the shifting parameter for x. Let's increase the constant term from 1 to 100 in the Noonberg example. For "a = 5.00000000000000E-11", a Newton's iterator fails but the function "myInterpolation(a,x,winding,shift)" returns the value, "x = -3.68403149433520020633200," which is extremely close to the higher-order multiple point that we are interested in.

3.3 Conclusion: implementation of the algorithm

In summary, we implemented several algorithms to "sweep" critical points for polynomial systems with parameters. The "sweep" routine includes Ada programs to calculate quadratic turning points and display the determinant values and eigenvalues of the polynomial system at every step. When the determinant values or the eigenvalues change signs, the "detectable"

singularities, including "detectable " higher-order singularities, will be detected. For the case of jumping-over singularities, we make good use of Parabolic Interpolation of Determinants to detect singularities and Puiseux series to iterate close to these singularities. We will talk about the implementation and applications in great detail in the next chapter.

CHAPTER 4

APPLICATIONS

In this chapter, we are going to show the "sweep" routine in the Ada directory of PHCpack, how to use the "sweep" routine to sweep the critical points for polynomial systems with parameters, and apply Puiseux series to approximate "jump-over" singularities when necessary. We will also show the results of our experiments.

4.1 The "sweep" routine in Ada

Our motivation is to compute critical points for a system of polynomials with one parameter. These critical points are always meaningful in mechanical design, such as finding all singular positions of the movement of a robotic object. In chapter one, we introduced the concept of a "sweep."

The Ada routine to call a "sweep" has two packages in the Ada directory of PHCpack. The Ada package "Standard_Quad_Turn_Points" offers some basic utilities to locate quadratic turning points of real polynomial systems, starting from a real or a complex start solution. The Ada package "Standard_Quad_Turn_Points_io" offers some basic i/o for the computation of quadratic turning points in real homotopies. The Ada package "ts_realcont" is a test function to call the "sweep" routine. The "sweep" routine is capable of finding all the quadratic turning points for polynomial systems with parameters.

Those two packages have several Ada procedures. These procedures perform a "sweep" starting at a given solution, either complex or real, until either a singularity is encountered, the target value for the parameter is reached, or the number of steps is exhausted. The user has an option to save the steps of the "sweep" in a file or use an interactive version to closely monitor the steps on screen. The algorithm of applying a simple "sweep" on a unit circle was mentioned in chapter one. In order to locate the turning point more accurately, the application of a first-order shooting method is also built into the package "Standard_Quad_Turn_Points."

4.2 Polynomial systems

We have selected three polynomial systems from the literature to use as examples of running the "sweep" routine. The first system [19] is small enough that global methods apply. The second system belongs to a family of polynomial systems [69], whose dimension can be scaled up to make it an intractable problem for global methods. The last example comes from mechanical design [90]. We are able to reproduce the results reported in the literature using "sweep."

4.2.1 Molecular configurations

The following system occurs in [19]:

$$f(\mathbf{x},\lambda) = \begin{cases} \frac{1}{2}(x_2^2 + 4x_2x_3 + x_3^2) + \lambda(x_2^2x_3^2 - 1) = 0 \\ \\ \frac{1}{2}(x_3^2 + 4x_3x_1 + x_1^2) + \lambda(x_3^2x_1^2 - 1) = 0 \\ \\ \frac{1}{2}(x_1^2 + 4x_1x_2 + x_2^2) + \lambda(x_1^2x_2^2 - 1) = 0. \end{cases} \tag{4.1}$$

The system is listed as a nontrivial example in [83, pages 391-392]. Since the system is small enough to be handled with resultant/symbolic methods and global methods, Emiris and Mourrain's symbolic methods work well on the system. The Jacobian criterion produces a system that can be solved by the blackbox solver of PHCpack. The system has 54 generic solutions which can be divided into five groups with the same x_1, x_2, x_3 and λ values. The first four groups have the same absolute values of x_1, x_2 and x_3 with the natural parameter λ being either +1.5i or -1.5i; there are exactly twelve solutions in each group. The last group has 6 solutions with the real value $\pm$ 0.866025403780023 as the natural parameter λ. In this example, all the special values of λ can be found by globally applying the Jacobian criterion.

Plugging the special value $\lambda = \pm$ 0.866025403780023 back into the system and then running the cascade of homotopy, we find two lists of solutions. One list contains 9 isolated solutions of dimension 0 and the other list is a curve of degree 6 of dimension 1. After filtering the list of solutions of dimension 0, we find 7 points (out of 9 points) lying on the 1-dimensional solution set. Therefore, a curve of degree 6 and 2 isolated solutions are found.

As λ approaches the origin, the system becomes singular. At the origin, there is one solution of multiplicity 8 for the system when the deflation method in PHCpack is applied. Considering

a sweep over the origin is very natural to follow the different paths at the origin. The sweep can be defined by the homotopy:

$$f(\mathbf{x},\lambda) = \begin{cases} \frac{1}{2}(x_2^2 + 4x_2x_3 + x_3^2) + \lambda(x_2^2x_3^2 - 1) = 0 \\ \frac{1}{2}(x_3^2 + 4x_3x_1 + x_1^2) + \lambda(x_3^2x_1^2 - 1) = 0 \\ \frac{1}{2}(x_1^2 + 4x_1x_2 + x_2^2) + \lambda(x_1^2x_2^2 - 1) = 0 \\ (\lambda - 1)(1 - t) + (\lambda + 1)t = 0. \end{cases} \tag{4.2}$$

As the artificial parameter t goes from 0 to 1, the natural parameter λ is swept from 1 to -1. According to the multihomogeneous Bézout bound [81], the permanent [81] of the degree matrix of the system is 16 when λ is set to 1. If we use Maple to solve the system in lexicographic order, we will also find 16 isolated solutions. So there are sixteen generic start solutions for the system at t = 0. Among them, there are four symmetrical complex conjugate solution pairs and four symmetrical real solution pairs, where "symmetrical" means that all eight real solutions and all eight complex solutions can be paired as $x_1 = x_2$, $x_2 = x_3$, $x_1 = x_3$ and $x_1 = x_2 = x_3$. As λ is swept from 1 to -1, starting with start solutions at t = 0, four real solution paths converge around the origin and the four complex solution paths diverge. Conversely, if we would like to track the converging complex solutions paths, we would set the homotopy to $(\lambda + 1)(1 - t) + (\lambda - 1)t = 0$ such that the four complex solution paths converge around the origin and the four real solution paths diverge. The special value zero for the natural parameter

λ is found by using a "sweep" as the tangent flips. There is a point of multiplicity 8 found at the original.

Applying parabolic interpolation of determinants when sweeping, we discover singularities at $\lambda = \pm 0.866025403780023$ on symmetrical curves of degree 6. Due to the "big" step size, the tangent does not flip at the singular point.

4.2.2 Neural networks

The following problem occurs as a family of polynomial systems in [69]. For $n = 3$, the system is

$$f(\mathbf{x}, \lambda) = \begin{cases} x_1 x_2^2 + x_1 x_3^2 - \lambda x_1 + 1 = 0 \\ x_2 x_1^2 + x_2 x_3^2 - \lambda x_2 + 1 = 0 \\ x_3 x_1^2 + x_3 x_2^2 - \lambda x_3 + 1 = 0. \end{cases} \tag{4.3}$$

As this system can be written for any number n, it is obvious that a plain application of the Jacobian criterion leads to an intractable problem, whereas tracking one solution path at a time is much less complex. Using the Jacobian criterion, the 7-by-7 system has 54 generic solutions. Special real values for the natural parameter λ found among these solutions are shown as follows:

$\lambda = 0$,

$\lambda = 1.88988157484231$,

$\lambda = 3.61703146124952$,

$\lambda = 2.38110157795230$,

λ = -0.414704714645147.

If we apply a "sweep," we can monitor the convergence of a solution path closely. The "sweep" can be defined by the Homotopy: For $n = 3$, the system is

$$f(\mathbf{x}, \lambda) = \begin{cases} x_1x_2^2 + x_1x_3^2 - \lambda x_1 + 1 = 0 \\ x_2x_1^2 + x_2x_3^2 - \lambda x_2 + 1 = 0 \\ x_3x_1^2 + x_3x_2^2 - \lambda x_3 + 1 = 0. \\ (\lambda + 0.1)(1 - t) + (\lambda - 0.1)t = 0. \end{cases} \tag{4.4}$$

As the artificial parameter t goes from 0 to 1, the natural parameter λ is swept from -0.1 to 0.1. The "sweep" finds all the special parameters values found by the Jacobian criterion except the special value 0. Solving the start system provides 21 start solutions. Applying the sweep, we found that 6 of the start solutions converge to 1.88988157484231; 6 of the start solutions converge to 3.61703146124952; 3 of the start solutions converge to -0.414704714645147; 2 of the start solutions converge to 2.38110157795230; and the remaining 4 start solutions, which form an almost straight solution path, will go through the origin without being detected. Applying parabolic interpolation of determinants, we discover a critical point at the origin. The tangent does not flip at the origin due to possibly jumping over the critical point.

4.2.3 Stewart-Gough platforms

In a highly symmetrical Stewart-Gough platform, the controllability of the platform at singular positions is important to Engineering fields such as aviation, space exploration, and

robotics. The symmetrical Stewart-Gough platform with six degrees-of-freedom [90] is shown in Figure 9. In such a symmetrical Stewart-Gough platform, two rigid bodies are connected by six rods attached by spherical joints. The two rigid bodies are in relative position to each other.

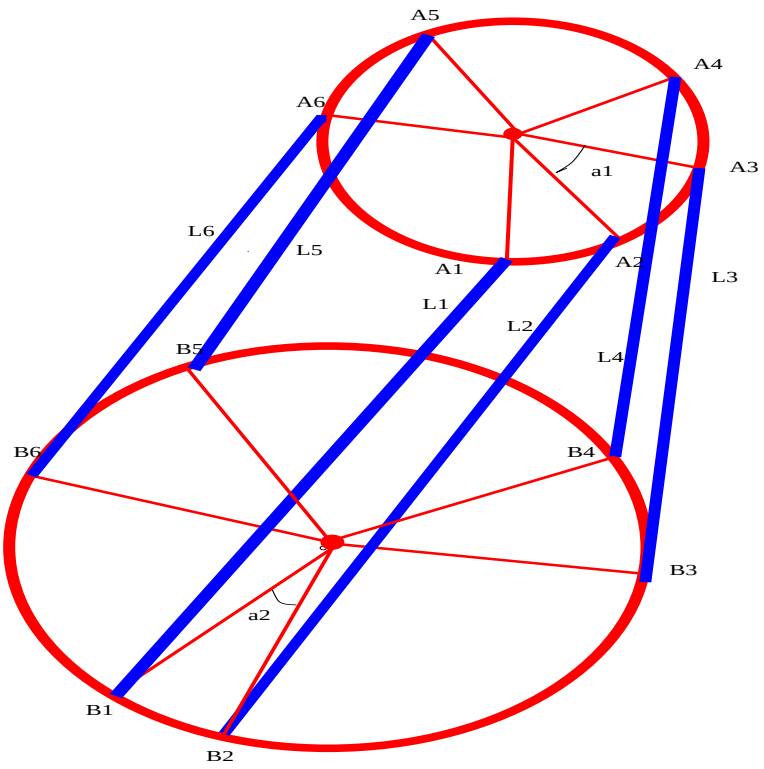

Figure 9. A symmetrical Stewart-Gough platform with six degrees-of-freedom

The following system can be used to determine all the configuration branches and bifurcation points of six degrees-of-freedom symmetrical Stewart-Gough platforms, described in [90].

$$f(\mathbf{x}, \mathrm{L}_1) = \begin{cases} f_i = (x_i - x_{i0})^2 + (y_i - y_{i0})^2 + z_i^2 - l_i^2 (i = 1, 2, ..., 6) \\ f_7 = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2 - 2R_1^2(1 - \cos(\alpha_1)) \\ f_8 = (x_1 - x_0)^2 + (y_1 - y_0)^2 + (z_1 - z_0)^2 - R_1^2 \\ f_9 = (x_2 - x_0)^2 + (y_2 - y_0)^2 + (z_2 - z_0)^2 - R_1^2 \end{cases} \tag{4.5}$$

for

$$\begin{cases} x_i = w_1 x_0 + w_2^{m_1} w_3^{m_2} x_1 + w_2^{m_2} w_3^{m_1} x_2 \\ y_i = w_1 y_0 + w_2^{m_1} w_3^{m_2} y_1 + w_2^{m_2} w_3^{m_1} y_2 \\ z_i = w_1 z_0 + w_2^{m_1} w_3^{m_2} z_1 + w_2^{m_2} w_3^{m_1} z_2 \end{cases} \tag{4.6}$$

where

$$\begin{cases} w1 = \frac{3sin\alpha_1 + (-1)^m \sqrt{3}(cos\alpha_1 - 1)}{2sin\alpha_1} \\ w2 = \frac{-sin\alpha_1 - (-1)^m \sqrt{3}(cos\alpha_1)}{2sin\alpha_1} \\ w3 = \frac{(-1)^m \sqrt{3}}{2sin\alpha_1}. \end{cases} \tag{4.7}$$

$$\begin{cases} m = 0, \ when \ i = 3, 6, \\ m = 1, \ when \ i = 4, 5 \end{cases} \tag{4.8}$$

$$\begin{cases} m_1 = 0, m_2 = 1, \ when \ i = 3, 5, \\ m_1 = 0, m_2 = 1, \ when \ i = 4, 6 \end{cases} \tag{4.9}$$

α_1 is the relative angle between two joint-triangles in the movable platform (the top platform), whereas α_2 is the relative angle between two joint-triangles in the base platform (the fixed platform on the bottom). R_1 is the radius of joints on the movable (top) platform. Although this system could be considered as a two-parameter problem of L_1 and L_i for $(i = 2, \ldots, 6)$, for the purpose of simplicity, we fix L_i to a real length and treat L_1 as a parameter. Integrating Eq.(8) to Eq.(12), we obtain the symmetrical platform with nine dimensions: x = $[x_0, y_0, z_0, x_1, y_1, z_1, x_2, y_2, z_2]$.

This system is much bigger compared to those in the previous two examples. Therefore, applying the Jacobian criterion is rather expensive, since solving a 19-by-19 system is required. It takes 8 hours and 55 minutes to find all 256 regular solutions for the 19-by-19 system using the blackbox solver in PHCpack on a 2.4 Ghz Linux machine. Since the mixed volume for the system is 4,608, the same number of paths needs to be tracked. Sweeping locally to find all quadratic turning points makes more sense in this case. By fixing L_i to 1.5, 2.0, and 3.0, we are able to find four special values for the natural parameter L_1 for each L_i with higher precision.

To sum up from the data in section 4.3, applying local sweeps help us find bifurcation points with higher precisions from 10^{-7} to 10^{-16}, compared to the data posted in [90]. In addition, we are able to see that z_0 can be either positive or negative.

When L_i is around 1.003, a multiple singular point occurs at the origin and L_i approximates the value of L_1, the system becomes a two-parameter problem and requires special care. For the time being, we only concentrate on the one parameter cases of this problem.

4.3 Experimental data

l_i	M_i	l_1	x_0	y_0	z_0
1.5	M_1	1.25545484315541	0.0457977771113300	-0.137910957471223	± 0.0171002602689638
1.5	M_2	1.47275687143534	-0.0794313388992136	0.177107430690531	± 1.07337778280609
1.5	M_3	1.52346902551850	-0.0506601244396293	-0.200728310822230	± 1.08364609883707
1.5	M_4	1.73368658400808	-0.119437254511978	0.0684036180374109	± 0.00340374313034239
2.0	M_1	1.73038986250179	0.0268915761641325	-0.149915020289956	± 0.0231273268048275
2.0	M_2	1.95798464537013	-0.142619775907013	0.374975885091748	± 1.64513582559604
2.0	M_3	2.03742087120587	-0.081535244565704	-0.416587787322360	± 1.65744848003223
2.0	M_4	2.24577649834005	-0.155654193080636	0.0408077764933302	± 0.111476101296520
3.0	M_1	2.81886485175146	-0.624218435401017	-0.110705353697865	± 1.80704550482076
3.0	M_2	2.94449339468704	-0.211303161911459	0.781662747068259	± 2.65003274759981
3.0	M_3	3.05234178015748	-0.0788925946355437	-0.844008017385622	± 2.66107026366929
3.0	M_4	3.17719187710019	-0.636400882693620	-0.295290561100177	± 1.90479382143718

TABLE I

POSITIONS OF THE MOVABLE PLATFORM AT BIFURCATION POINTS OF DIFFERENT LENGTH OF INPUT PARAMETERS L_I

APPENDICES

.1 The general system of a symmetrical Stewart-Gough platform

9

$x[1]^2$-4*x[1]+4+$y[1]^2$+$z[1]^2$-$L1^2$;

$x[2]^2$-3.708735418267150*x[2]+1.750000000000000+$y[2]^2$-1.498426373663648*y[2]+$z[2]^2$;

2.568151231325932*x[0]-4.257513793119272*x[1]+3.689362561793340*x[2]-4.448168414177086*y[0]

+7.374230203607868*y[1]-6.390163404568536*y[2]+3.402849028090579*$x[2]^2$

+3.402849028090579*$y[2]^2$+1.648850186740225*$x[0]^2$-5.466969645093198*x[0]*x[1]

+4.737420502938680*x[0]*x[2]-7.853755997326498*x[1]*x[2]+1.648850186740225*$y[0]^2$

-5.466969645093198*y[0]*y[1]+4.737420502938680*y[0]*y[2]-7.853755997326498*y[1]*y[2]

+1.648850186740225*$z[0]^2$-5.466969645093198*z[0]*z[1]+4.737420502938680*z[0]*z[2]

-7.853755997326498*z[1]*z[2]+4.531605924650213*$x[1]^2$+4.531605924650213*$y[1]^2$

+4.531605924650213*$z[1]^2$+3.402849028090579*$z[2]^2$+1.749999999999999;

5.408667468934306*x[0]-5.814514745294392*x[1]+3.557890290786974*x[2]-4.225714152032840*y[0]

+4.542796795609874*y[1]-2.779728544879666*y[2]+1.274092131530941*$x[2]^2$

+1.274092131530941*$y[2]^2$+2.944396492762429*$x[0]^2$-6.330667182441340*x[0]*x[1]

+3.873722965590550*x[0]*x[2]-4.164393435533158*x[1]*x[2]+2.944396492762429*$y[0]^2$

-6.330667182441340*y[0]*y[1]+3.873722965590550*y[0]*y[2]-4.164393435533158*y[1]*y[2]

+2.944396492762429*$z[0]^2$-6.330667182441340*z[0]*z[1]+3.873722965590550*z[0]*z[2]

-4.164393435533158*z[1]*z[2]+3.402849028090579*$x[1]^2$+3.402849028090579*$y[1]^2$

+3.402849028090579*$z[1]^2$+1.274092131530941*$z[2]^2$+1.749999999999999;

3.431848768674068*x[0]+2.257513793119272*x[1]-3.689362561793340*x[2]+5.944136431236176*y[0]

+3.910128588470114*y[1]-6.390163404568536*y[2]+3.402849028090579*$x[2]^2$+3.402849028090579*$y[2]^2$

+2.944396492762429*$x[0]^2$+3.873722965590550*x[0]*x[1]-6.330667182441340*x[0]*x[2]

-4.164393435533158*x[1]*x[2]+2.944396492762429*$y[0]^2$+3.873722965590550*y[0]*y[1]

-6.330667182441340*y[0]*y[2]-4.164393435533158*y[1]*y[2]+2.944396492762429*$z[0]^2$

+3.873722965590550*z[0]*z[1]-6.330667182441340*z[0]*z[2]-4.164393435533158*z[1]*z[2]

+1.274092131530941*$x[1]^2$+1.274092131530941*$y[1]^2$+1.274092131530941*$z[1]^2$

+3.402849028090579*$z[2]^2$+1.749999999999999;

.7148351411960820*x[0]+1.026920056581502*x[1]-1.185062793937321*x[2]+5.086316320162834*y[0]

+7.306915877909088*y[1]-8.432159923105640*y[2]+4.531605924650213*$x[2]^2$+4.531605924650213*$y[2]^2$

+1.648850186740225*$x[0]^2$+4.737420502938680*x[0]*x[1]-5.466969645093198*x[0]*x[2]-7.853755997326498

+1.648850186740225*$y[0]^2$+4.737420502938680*y[0]*y[1]-5.466969645093198*y[0]*y[2]-7.853755997326498

+1.648850186740225*$z[0]^2$+4.737420502938680*z[0]*z[1]-5.466969645093198*z[0]*z[2]

-7.853755997326498*z[1]*z[2]+3.402849028090579*$x[1]^2$ +3.402849028090579*$y[1]^2$

+3.402849028090579*$z[1]^2$+4.531605924650213*$z[2]^2$+1.750000000000002;

$x[2]^2$-2*x[1]*x[2]+$x[1]^2$+$y[2]^2$-2*y[1]*y[2]+$y[1]^2$+$z[2]^2$-2*z[1]*z[2]+$z[1]^2$ -.2341048142821462;

$x[1]^2$-2*x[0]*x[1]+$x[0]^2$+$y[1]^2$-2*y[0]*y[1]+$y[0]^2$+$z[1]^2$-2*z[0]*z[1]+$z[0]^2$-1;

$x[2]^2$-2*x[0]*x[2]+$x[0]^2$+$y[2]^2$-2*y[0]*y[2]+$y[0]^2$+$z[2]^2$-2*z[0]*z[2]+$z[0]^2$-1;

.2 Running a "sweep" on the symmetrical Stewart-Gough platform

kathy@venus: /Desktop/phc/bin $ ./ts_realcont

Parameter continuation on real polynomial systems.

Reading the name of the input file.

Give a string of characters : sg40

number of equations : 9

number of variables : 10

MENU for quadratic turning point computation :

1. interactive sweep from real solution;

2. interactive sweep from complex solution;

3. real sweep till target or singularity;

4. complex sweep till target or singularity.

Type 1, 2, 3, or 4 to choose : 3

Reading the name of the output file

Give a string of characters : sg40_output

Give end target value for t : 1

Give maximal number of steps along a path : 100

Monitor determinants and eigenvalues ? (y/n) y

CITED LITERATURE

1. Allgower, E. L. and Schwetlick, H.: A general approach to minimally extended systems for simple bifurcation points. Z. Angew. Math. Mech., 75:611–612, 1995.

2. Allgower, E. and Georg, K.: Introduction to Numerical Continuation Methods, volume 45 of Classics in Applied Mathematics. SIAM, 2003.

3. Bates, D. J., Hauenstein, J. D., Sommese, A., and Wampler, C.: Bertini:Software for solving polynomial systems. The University of Notre Dame, 2006. `http://www.nd.edu/ sommese/bertini/`.

4. Bates, D., Peterson, C., and Sommese, A.: A numerical-symbolic algorithm for computing the multiplicity of a component of an algebraic set. J. Complexity, 22(4):475–489, 2006.

5. Blum, L., Cucker, F., Shub, M., and Smale, S.: Complexity and Real Computation. Springer–Verlag, 1998.

6. Cox, D., Little, J., and O'Shea, D.: Using Algebraic Geometry, volume 185 of Graduate Texts in Mathematics. Springer-Verlag, 1998.

7. Dayton, B. and Zeng, Z.: Computing the multiplicity structure in solving polynomial systems. In Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation. ACM, 2005., ed. M. Kauers, pages 116–123, 2005.

8. Decker, D., Keller, H., and Kelley, C.: Convergence rates for Newton's method at singular points. SIAM J. Numer. Anal., 20(2):296–314, 1983.

9. Decker, D., Keller, H., and Kelley, C.: Broyden's method for a class of problems having singular Jacobian at the root. SIAM J. Numer. Anal., 22:566–574, 1985.

10. Decker, D. and Kelley, C.: Newton's method at singular points: II. SIAM J. Numer. Anal., 17(3):465–471, 1980.

11. Dedieu, J. and Shub, M.: Newton's method for overdetermined systems of equations. Math. Comp., 69(231):1099–1115, 1999.

12. Demmel, J.: Applied Numerical Linear Algebra, volume 45 of Classics in Applied Mathematics. SIAM, 2003.

13. Deuflard, P., Friedler, B., and Kunkel, P.: Efficient numerical path folowing beyond critical points. SIAM J. Numer. Anal., 24:912–927, 1987.

14. Deuflhard, P.: Newton Methods for Nonlinear Problems. Affine Invariance and Adaptive Algorithms. Springer-Verlag, 2004.

15. Dietmaier, P.: The Stewart-Gough platform of general geometry can have 40 real postures. In Advances in Robot Kinematics: Analysis and Control, eds, J. Lenarčič and M. Husty, pages 7–16. Kluwer Academic Publishers, 1998.

16. Dimca, A.: Topics on Real and Complex Singularities. Vieweg, 1987.

17. Dumortier, F., Llibre, J., and Artes, J.: Qualitative Theory of Planar Differential systems. Springer, 2006.

18. Durand, C.: Symbolic and Numerical Techniques for Constraints Solving. Doctoral dissertation, Purdue University, Department of Computer Science, 1998.

19. Emiris, I. and Mourrain, B.: Computer algebra methods for studying and computing molecular conformations. Algorithmica, 25(2-3):372–402, 1999. Special issue on algorithmic research in Computational Biology, edited by D. Gusfield and M.-Y. Kao.

20. Fierro, R. and Hansen, P.: Utv expansion pack: Special-purpose rank-revealing algorithms. Numerical Algorithms, 40(1):47–66, 2005.

21. Gao, T. and Li, T.: Mixed volume computation for semi-mixed systems. Discrete Comput. Geom., 29(2):257–277, 2003.

22. Gao, T. and Li, T.: Algorithm 846: Mixedvol: a software package for mixed-volume computation. ACM Transactions on Mathematical Software, 31(4):555–560, 2005.

23. Gerald, C. F. and Wheatley, P. O.: Applied Numerical Analysis. Addison-Wesley, 2003.

24. Giusti, M., Lecerf, G., Salvy, B., and Yakoubsohn, J.: On location and approximation of clusters of zeroes of analytic functions. Found. Comput. Math., 5(3):257–311, 2005.

25. Giusti, M., Lecerf, G., Salvy, B., and Yakoubsohn, J.: On location and approximation of clusters of zeroes: case of embedding dimension one. Found. Comput. Math., 7(1):1–58, 2007.

26. Govaerts, W.: Numerical Methods for Bifurcations of Dynamical Equilibria. SIAM, 2000.

27. Greuel, G.-M., Pfister, G., and Schönemann, H.: SINGULAR 2.0. A Computer Algebra System for Polynomial Computations, Centre for Computer Algebra, University of Kaiserslautern, 2001. `http://www.singular.uni-kl.de`.

28. Greuel, G. and Pfister, G.: Advances and improvements in the theory of standard bases and syzygies. Arch. Math., 66:163–196, 1996.

29. Greuel, G. and Pfister, G.: A Singular Introduction to Commutative Algebra. Springer-Verlag, 2002.

30. Griewank, A.: On solving nonlinear equations with simple singularities or nearly singular solutions. SIAM Review, 27(4):537–563, 1985.

31. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, 2000.

32. Guan, Y. and Verschelde, J.: PHClab: A MATLAB/Octave interface to PHCpack. 148:15–32, 2008. IMA Volume.

33. Guckenheimer, J. Myers, M. and Sturmfels, B.: Computing hopf bifurcations. ii: Three examples from neurophysiology. SIAM J. Sci. Comput., 17(6):1275–1301, 1996.

34. Gunji, T., Kim, S., Kojima, M., Takeda, A., Fujisawa, K., and Mizutani, T.: PHoM – a polyhedral homotopy continuation method for polynomial systems. Computing, 73(1):57–77, 2004.

35. Heath, M.: Scientific Computing: an introductory survey. Data Processing. Springer-Verlag, second edition, 2002.

36. Kearfott, R. and Dian, J.: Existence verification for higher degree singular zeros of nonlinear systems. SIAM J. Number. Anal., 41(6):2350–2373, 2003.

37. Kearfott, R. and Xing, Z.: An interval step control for continuation methods. SIAM J. Numer. Anal., 31(3):892–914, 1994.

38. Kobayashi, H., Suzuki, H., and Sakai, Y.: Numerical calculation of the multiplicity of a solution to algebraic equations. Math. Comp., 67(221):257–270, 1998.

39. Kress, R.: Numerical Analysis, volume 181 of Graduate Texts in Mathematics. Springer, 1998.

40. Kunkel, P.: Efficient computation of singular points. IMA J. Numer. Anal., 9:421–433, 1989.

41. Kunkel, P.: A tree-based analysis of a family of augmented systems for the computation of singular points. IMA J. Numer. Anal., 16:501–527, 1996.

42. Kuo, Y. and Li, T.: Determine the dimension of the solution component that contains a computed zero of a polynomial system. Math.Anal.Appl, 338:840–851, 2008.

43. Kuznetsov, Y.: Elements of Applied Bifurcation Theory, volume 112 of Applied Mathematical Sciences. Springer-Verlag, third edition, 2004.

44. Labs, O.: Hypersurfaces with Many Singularities - History, Constructions, Algorithms, Visualization. Doctoral dissertation, Johannes Gutenberg University of Mainz, Department of Physics, Mathematics and Computer Science, 2005.

45. Lazard, D. and Rouillier, F.: Solving parametric polynomial systems. Journal of Symbolic Computation, 42:636–667, 2007.

46. Lecerf, G.: Quadratic Newton iteration for systems with multiplicity. Found. Comput. Math., 2:247–293, 2002.

47. Lee, T., Li, T., and Tsai, C.: Software pakage which implements the polyhedral homotopy continuation method. Michigan state University, 2008. `http://www.msu.edu/ leetsung/HOM4PSsoft.htm`.

48. Leykin, A. and Verschelde, J.: A Maple interface to the numerical homotopy algorithms in PHCpack. 2:139–147, 2004. In Quoc-Nam Tran, editor, *Proceedings of the Tenth International Conference on Applications of Computer Algebra (ACA'2004)*,.

49. Leykin, A. and Verschelde, J.: Factoring Solution Sets of Polynomial Systems in Parallel. pages 173–180, 2005. Proceedings of the 2005 International Conference on Parallel Processing Workshops.

50. Leykin, A. and Verschelde, J.: Interfacing with the Numerical Homotopy Algorithms in PHCpack. pages 354–360, 2006. Proceedings of ICMS 2006, LNCS 4151.

51. Leykin, A., Verschelde, J., and Zhao, A.: Newton's method with deflation for isolated singularities of polynomial systems. Theoretical Computer Science, 359(1-3):111–122, 2006.

52. Leykin, A., Verschelde, J., and Zhao, A.: Evaluation of Jacobian matrices for Newton's method with deflation to approximate isolated singular solutions of polynomial systems. pages 269–278, 2007.

53. Leykin, A., Verschelde, J., and Zhao, A.: Higher-order deflation for polynomial systems with isolated singular solutions. 2007.

54. Li, T.: Numerical solution of multivariate polynomial systems by homotopy continuation methods. Acta Numerica, 6:399–436, 1997.

55. Li, T.: Numerical solution of polynomial systems by homotopy continuation methods. In Handbook of Numerical Analysis. Volume XI. Special Volume: Foundations of Computational Mathematics, ed. F. Cucker, pages 209–304. North-Holland, 2003.

56. Li, T. and Li, X.: Finding mixed cells in the mixed volume computation. Found. Comput. Math., 1(2):161–181, 2001.

57. Li, T. and Wang, X.: Solving real polynomial systems with real homotopies. Math. Comp., 60:669–680, 1993.

58. Li, T. and Wang, X.: Higher order turning points. Appl. Math. Comput., 64:155–166, 1994.

59. Li, T. and Zeng, Z.: A rank-revealing method with updating, downdating and applications. SIAM J. Matrix Anal. Appl., 26:918–946, 2005.

60. Li, T., Zeng, Z., and Cong, L.: Solving eigenvalue problems of real nonsymmetric matrices with real homotopies. SIAM J. Numer. Anal., 29(1):229–248, 1992.

61. Lu, Y., Bates, D., Sommese, A., and Wampler, C.: Finding all real points of a complex curve. Contemporary Mathematics, 448:183–206, 2007.

62. Marinari, M., Moeller, H., and Mora, T.: On multiplicities in polynomial system solving. Trans. AMS, 348(8):3283–3321, 1996.

63. Mei, Z.: Numerical Bifurcation Analysis for Reaction-Diffusion Equations. Springer–Verlag, 2000.

64. Möller, H. M. and Stetter, H. J.: Multivariate polynomial equations with multiple zeros solved by matrix eigenproblems. Numer. Math., 70:311–329, 1995.

65. Mora, T.: Gröbner duality and multiple points in linearly general position. Proceedings of the AMS, 125(5):1273–1282, 1997.

66. Moritsugu, S. and Kuriyama, K.: On multiple zeros of systems of algebraic equations. In Proceedings of the 1999 International Symposium on Symbolic and Algebraic Computation (ISSAC 1999), ed. S. Dooley, pages 23–30. ACM, 1999.

67. Mourrain, B.: Isolated points, duality and residues. Journal of Pure and Applied Algebra, 117/118:469–493, 1997.

68. Nocedal, J. and Wright, S.: Numerical Optimization. Springer, 1999.

69. Noonburg, V.: A neural network modeled by an adaptive Lotka-Volterra system. SIAM J. Appl. Math., 49(6):1779–1792, 1989.

70. Ojika, T.: Modified deflation algorithm for the solution of singular problems. I. A system of nonlinear algebraic equations. J. Math. Anal. Appl., 123:199–221, 1987.

71. Ojika, T.: A numerical method for branch points of a system of nonlinear algebraic equations. Applied Numerical Mathematics, 4:419–430, 1988.

72. Ojika, T., Watanabe, S., and Mitsui, T.: Deflation algorithm for the multiple roots of a system of nonlinear equations. J. Math. Anal. Appl., 96:463–479, 1983.

73. Rossum, G. v.: Why was Python created in the first place? 2002.

74. Rouillier, F., Roy, M., and Safey El Din, M.: Finding at least one point in each connected component of a real algebraic set defined by a single equation. Journal of Complexity, 16:716–750, 2000.

75. Safey El Din, M. and Schost, E.: Properness defects of projections and computation of one point in each connected component of a real algebraic set. Journal of Discrete and Computational Geometry, 32(3):417–430, 2004.

76. Sommese, A., Verschelde, J., and Wampler, C.: Using monodromy to decompose solution sets of polynomial systems into irreducible components. Application of Algebraic Geometry to Coding Theory, Physics, and Computation, pages 297–315, 2001.

77. Sommese, A., Verschelde, J., and Wampler, C.: A method for tracking singular paths with application to the numerical irreducible decomposition. pages 329–345, 2002. In *Algebraic Geometry, a Volume in Memory of Paolo Francia*, edited by M.C. Beltrametti, F. Catanese, C. Ciliberto, A. Lanteri, C. Pedrini. W. de Gruyter.

78. Sommese, A., Verschelde, J., and Wampler, C.: An intrinsic homotopy for intersecting algebraic varieties. J. of Complexity, 42:1552–1571, 2005.

79. Sommese, A. and Wampler, C.: Numerical algebraic geometry. In The Mathematics of Numerical Analysis, eds, J. Renegar, M. Shub, and S. Smale, volume 32 of Lectures in Applied Mathematics, pages 749–763. AMS, 1996. Proceedings of the AMS-SIAM Summer Seminar in Applied Mathematics. Park City, Utah, July 17-August 11, 1995, Park City, Utah.

80. Sommese, A. and Wampler, C.: The Numerical Solution of Systems of Polynomials arising in Engineering and Science. World Scientific Press, 2005.

81. Sommese, A.J.and Verschelde, J.: Numerical homotopies to compute generic points on positive dimensional algebraic sets. J. of Complexity, 16(3):572–602, 2000.

82. Stein, W.: Sage Mathematics Software (Version 3.0). The Sage Group, 2008. `http://www.sagemath.org`.

83. Stetter, H.: Numerical Polynomial Algebra. SIAM, 2004.

84. Stetter, H. and G.T., T.: Singular systems of polynomials. pages 9–16, 1998. In *Proceedings of ISSAC 1998*, edited by O. Gloor.

85. Sturmfels, B.: Solving Systems of Polynomial Equations. Number 97 in CBMS Regional Conference Series in Mathematics. AMS, 2002.

86. Venners, B.: The making of Python. Artima Developer, 1, 2003.

87. Verschelde, J.: Algorithm 795: PHCpack: A general-purpose solver for polynomial systems by homotopy continuation. ACM Trans. Math. Softw., 25(2):251–276, 1999. Software available at `http://www.math.uic.edu/~jan`.

88. Verschelde, J. and Zhao, A.: Newton's method with deflation for isolated singularities. Poster presented at ISSAC'04, 6 July 2004, Santander, Spain. Available at `http://www.math.uic.edu/~jan/poster.pdf` and at `http://www.math.uic.edu/~azhao1/poster.pdf`.

89. Walker, R.: Algebraic Curves. Princeton University Press, 1950.

90. Wang, Y. and Wang, Y.: Configuration bifurcations analysis of six degree-of-freedom symmetrical Stewart parallel mechanism. ASME, 127(2):70–77, 2005.

91. Watson, L., Billups, S., and Morgan, A.: Algorithm 652: Hompack: a suite of codes for globally convergent homotopy algorithms. ACM Trans.Math.Softw., 13(3):281–310, 1987.

92. Zhuang, Y.: Parallel implementation of polyhedral homotopy methods. 2008. PHD Thesis, UIC.

VITA

Katherine W. Piret

Department of Mathematics, Statistics, and Computer Science,

University of Illinois at Chicago.

wpiret1@math.uic.edu (312)-413-8263

Education

Ph.D. in Mathematical Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2008

Master of Science in Mathematical Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2004

Bachelor of Science in Mathematical Computer Science, University of Illinois at Chicago, Chicago, Illinois, 2003

Skills

Problem Solving

Excellent analytical and logical reasoning skills. Able to multi-task. Can learn new skills quickly. Able to lead or work within a group environment.

Computer Languages

Some experience with C/C++, HTML, Java, Python, SQL, and SAS. Can become proficient

in any of these (or other) languages upon request.

Other

Creative, motivated, and innovative mathematical ability. Teaching skills.

Employment, University of Illinois at Chicago

Teaching Assistant *8/2006 - 1/2008*

Ran discussion/problem/review sessions twice a week that augmented the basic first-year Calculus course and Computer Science course, graded projects and quizzes, some guest lecturing.

Research Assistant *5/2005 to 5/2008*

Worked with Professor Jan Verschelde; constructed a Python interface to PHCpack and Homotopy methods to compute critical points by continuation.

Employment, Harold Washington College

Part-Time Instructor *8/2006 to present*

Preparation and presentation of lectures, supervision of group work, creating my own syllabus and homework problems, writing and grading tests and quizzes, grading projects and papers, preparing, and grading the final exams for the courses as follows:

Math 098 College Algebra Fall 2006, Spring 2007

Math 099 Intermediate Algebra Fall 2007, Spring 2008

Presentations

Polynomial Continuation for Singular Solutions, ICCAM 2008 International Congress on Computational and Applied Mathematics, 7-11 July 2008, University of Ghent, Belgium.

PHCpython: The Python interface to PHCpack, SIAM Student Chapter Meeting, May

5, 2008, Chicago, IL.

Computing Critical Points by Continuation, Mathematical Aspects of Computer and Information Sciences, December 5-7, 2007, Paris, France.

Computing Critical Points by Continuation, AMS Special Session: Numerical and Symbolic Techniques in Algebraic Geometry and its Applications, Oct 5-6, 2007, Depaul University, Chicago, IL.

Publications

Xun Luo, Yun Guan and Katherine Piret, "A Service Framework for Temporal Link Analysis with Historical Segments Integration," 2007 IEEE International Conference on Systems, Man and Cybernetics(SMC'07), Montreal, Canada, Oct. 2007.

Awards and Honors

SKIT Fellowship Award *8/2005 - 5/2006*

One of the three recipients of the competative fellowship award given by the Department of Education to work with high school teachers and students.

GK-12 Fellowship Award *8/2004 - 5/2005*

Recipient of the fellowship award given by the Department of Mathematics, Statistics, and Computer Science to facilitate teaching in nearby high schools.

UIC Tuition Waiver Award *8/2001-5/2003*

Acadamic Acheivement Award given by the University of Illinois at Chicago to finish my bachelor's degree.