

**H-GATES FRAMEWORK FOR STRAIGHT-LINE PROGRAMS AND THEIR
APPLICATIONS IN NONLINEAR ALGEBRA**

A Dissertation
Presented to
The Academic Faculty

By

Hannah Mahon

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Mathematics

-

Georgia Institute of Technology

December 2025

© Hannah Mahon 2025

**H-GATES FRAMEWORK FOR STRAIGHT-LINE PROGRAMS AND THEIR
APPLICATIONS IN NONLINEAR ALGEBRA**

Thesis committee:

Dr. Anton Leykin
School of Mathematics
Georgia Institute of Technology

Dr. Vijay Ganesh
School of Computer Science
Georgia Institute of Technology

Dr. Grigoriy Blekherman
School of Mathematics
Georgia Institute of Technology

Date approved: 11/20/2025

To my parents, with all my love

ACKNOWLEDGMENTS

My Master's has been a wonderful journey and it has been with the help of many people that this dissertation has come together.

First, thank you very much to my advisor, Professor Anton Leykin for introducing me to homotopy continuation and guiding me through the concept of this framework and its creative applications. I appreciate the challenging questions as they have sharpened my understanding not only of mathematics, but have also opened up the path to many promising applications. This project sits very nicely in the intersection of algebraic geometry, applied mathematics, computer science, and systems processing, and it has been engaging seeing how these areas overlap and interact.

Next I would like to thank all of my professors for deepening my understanding of mathematics, including Professor Plamen Iliev for engaging conversations around linear algebra, Professor Shahaf Nitzan, Professor Dorian Lubinsky, and Professor Michael Loss for seeing the beauty in analysis, and Professor Grigoriy Blekherman and Professor John Etnyre for teaching rigorous algebraic topics which has grown my love for the area. Thank you to all of my peers in the math community, including Anaheeta Hadjimirzaei for many stimulating and humorous conversations, Vanessa Newsome-Slade for the long, enjoyable study sessions, Colby Fernandez and Ninh Le for interesting algebraic topology discussions, Shane Kosieradzki for collaborating on fully homomorphic encryption research, and many more!

Thank you to all of my colleagues at Georgia Tech Research Institute for their support. In particular, Jay Danner, Lee Lerner, Will Stuckey, and Benjamin Yang have been incredibly supportive and encouraging throughout my experience. I greatly appreciate Nathan Braswell, Stefan Abi-Karam and Spencer Brown for many interesting conversations involving formal verification and systems processing, Jonathan Sanderson for listening to me talk about conics and asking good questions, Shaye Storm for excellent insight and

advice, and the many others who make working enjoyable.

Lastly, thank you to all of family and friends who have been my source of optimism throughout this experience. Thank you Lola, Ariella, Annie, Yana, Vinh, and many others for keeping me well-rounded and sane. Thank you to Manuel for being a rock. Thank you most of all to my parents for their belief and their love.

TABLE OF CONTENTS

Acknowledgments	iv
List of Tables	viii
List of Figures	ix
List of Acronyms	x
Summary	xi
Chapter 1: Introduction	1
1.1 Background	1
1.2 Notation	4
Chapter 2: Straight-line Programs	5
2.1 Polynomial Representations	5
2.2 Arithmetic Operations	7
2.3 SLPs, Circuits, and Auto Differentiation	10
2.4 Complexity Examples	13
Chapter 3: Macaulay2 Code	25
3.1 Types and Methods	25

Chapter 4: Solving Systems of Equations	30
4.1 Newton’s Method	30
4.1.1 Examples	36
4.2 Runge-Kutta Methods	43
4.3 Homotopy Continuation	47
4.3.1 Constructing a Start System	49
4.4 Predictor Gates	54
Chapter 5: Keplerian Orbits	67
5.1 Background	67
5.2 Problem Set up	69
5.3 Building in HGates.m2	77
5.4 Comparing Newton’s Method and Predictor-Corrector Homotopy Continuation	85
5.4.1 Generating A Large Dataset	91
Chapter 6: Conclusion	94
6.0.1 Future Work	94
References	96

LIST OF TABLES

5.1	Average number of unique solutions found for NM with 112 random starting points, PCHC7 tracking 108 paths (2 trials, $\Delta = 0.01$), PCHC2 tracking 1 solution (1 trial, $\Delta = 0.05$), and NAG4M2 solveSystem for the 7-square system.	87
5.2	Approximate Solutions Grouped by 2-norm Difference Less Than $1e - 4$ for the Circle Problem, G i reading “Group i ”	88
5.3	Approximate Solutions Grouped by 2-norm Difference Less Than $1e - 4$ for the Ellipse Problem, G i reading “Group i ”	89
5.4	Approximate Solutions Grouped by 2-norm Difference Less Than $1e - 4$ for the Parabola Problem, G i reading “Group i ”	90
5.5	Approximate Solutions Grouped by 2-norm Difference Less Than $1e - 4$ for the Hyperbola Problem, G i reading “Group i ”	91

LIST OF FIGURES

4.1	Two Iterations of Newton's Operator [23]	32
4.2	Comparison of Different Order Runge-Kutta methods for $y' = y \sin(t^2)y$ [24]	44
4.3	Predictor-Corrector Method for Solving Homotopy Continuation Problems [23]	48
5.1	Different Types of Conic Equations Obtained from Slicing a Right Circular Cone with a Plane[27]	67
5.2	Locations of Foci for a Parabola, Ellipse, and Hyperbola [29]	69
5.3	Corresponding Conic Equation of Grouped Solutions for the Circle Problem	88
5.4	Corresponding Conic Equation of Grouped Solutions for the Ellipse Problem	89
5.5	Corresponding Conic Equation of Grouped Solutions for the Parabola Problem	90
5.6	Corresponding Conic Equation of Grouped Solutions for the Ellipse Problem	91
5.7	Graph of the coefficients of 30 randomly generated conics with focus at the origin.	92
5.8	Graph of first three randomly generated conics and their corresponding eight points plotted in the same color. Each conic has two points through the x-axis and two points through the y-axis and four randomly chosen points.	93

SUMMARY

The massive computations required by artificial intelligence and scientific computing have outpaced general-purpose CPUs, necessitating hardware acceleration via GPUs. Straight-Line Programs (SLPs) are an ideal solution as by definition they are branchless and do not consist of any conditionals, avoiding thread divergence, high memory consumption, and incorrect solutions due to guessing caused by branching on hardware. SLPs can enable an algorithm to be mapped as a predictable data-flow circuit, enabling massive parallelization. Nonlinear systems are foundational to science and engineering, with their solutions studied with algebraic geometry and approximated by numerical algebraic geometry algorithms like homotopy continuation. These solvers rely on automatic (auto) differentiation, and since auto differentiation can be represented as an SLP, it is of great interest to represent the entire solver as an SLP for hardware acceleration.

This thesis introduces `HGates.m2`, a framework that expands on SLPs by defining hyper-level gates to model nonlinear algorithms. We detail how autodifferentiation and an adaptation of the homotopy continuation predictor-corrector method are implemented within this framework. For several geometric problems, we demonstrate how to set up polynomial systems of equations using H-gates.

CHAPTER 1

INTRODUCTION

1.1 Background

Hardware acceleration is critical today because of the massive computations required by artificial intelligence, data analytics, and scientific computing, which have outpaced the capabilities of general-purpose Central Processing Units (CPUs). Utilizing specialized hardware like Graphics Processing Units (GPUs) is essential for achieving higher speeds and lower energy consumption in solving these complex, large-scale problems. GPUs are built on a Single-Instruction, Multiple-Thread (SIMT) architecture which allows for instructions to be executed in parallel. To take advantage of the enormous parallelism capabilities of GPUs, it is of great interest to adapt algorithms so that execution is optimized over the architecture [1]. Straight-line programs (SLPs) are ideal for hardware acceleration because they represent a fixed sequence of arithmetic operations without any conditionals. In computation, *branching* refers to conditional logic such as an if-then-else statement, which forces a process to choose between different instruction paths. This choice can be highly disruptive to parallel hardware, causing issues like thread divergence, high-memory consumption, and sometimes even code execution failures when potentially incorrect guesses are made to speed up the computation. By definition, SLPs do not contain any conditionals or loops, so no branching is necessary and each path is of fixed length, allowing the GPU to fully utilize its SIMT architecture. This type of algorithm is often referred to as *branchless*. The SLP can be mapped to hardware as a pure, completely predictable data-flow circuit, enabling massive parallelization and significantly faster, safer execution. Studies on implementing branchless algorithms on GPUs have shown dramatic speedups, in some cases over $100\times$ efficiency, compared with traditional algorithms using conditionals [2] [3].

Numerically solving systems of equations is a fundamental, multi-disciplinary problem, showing up in areas such as chemistry and economics to identify equilibrium states, robotics and computational geometry to find intersection points, and optimization and machine-learning to identify critical points. The study of solving linear systems, $Ax = b$, is a pillar of numerical analysis with wide-reaching applications in today's technology, and has produced a collection of well-conditioned algorithms such as Gaussian elimination, QR factorization, and Cholesky decomposition [4, 5, 6, 7].

Nonlinear systems are often more difficult to solve numerically than linear systems, making the efficient numerical solving of these systems an important area of study. The theoretical properties of their solution sets are the focus of algebraic geometry, which connects abstract algebra and geometric shapes to study solutions of systems. To compute these solutions in practice, many algorithms in numerical algebraic geometry rely on the homotopy continuation method. Homotopy continuation requires a technique called automatic differentiation, referring to computing exact derivatives of a function via a computer algorithm, which can be represented as an SLP. This provides a natural motivator for studying numerical nonlinear algebra algorithms and their implementation as SLPs to take advantage of modern hardware acceleration. Some of the most well-studied algorithms in homotopy continuation as well as with handling general ordinary differential equations are Newton's method and the Runge-Kutta method. Newton's method is a well-known iterative algorithm used to approximate complex roots for nonlinear functions, offering rapid, quadratic convergence in certain conditions [8, 9]. Similarly, Runge-Kutta methods are a family of iterative algorithms for numerically predicting the next solution in an ordinary differential equation at a particular timestep, which can achieve high accuracy by using a weighted average of multiple derivative evaluations[10, 11].

One significant real-life application where numerical nonlinear algebra is necessary is in astronomy, where optical navigation (OPNAV), a well-studied, vital area in the space domain, relies on navigation and location tracking using data collected from images [12, 13,

14, 15]. A foundational concept in OPNAV is the determination of Keplerian orbits, which describe the motion of a space-body, such as a satellite, under the gravitational influence of a central body [16]. Mathematically, these orbits are conic sections, such as ellipses, parabolas, or hyperbolas, where the central body is positioned at one focus. Determining an orbit from complete or partially complete observational data requires solving the specific nonlinear system of equations that defines this conic, making it an ideal testbed for advanced numerical methods. An example of complete observational data is if we have a set consisting of points with the timestep of a space-body and its coordinates in 3-dimensional space, whereas an example of partially complete data is if we have some fully coordinates and some coordinates relying on indeterminants or variables.

In this thesis, we will introduce a framework called `HGates.m2` designed to build numerical algebraic geometry algorithms in a form that is targeted for hardware acceleration. `HGates.m2` expands on the concept of SLPs with arithmetic gates by introducing high-level gates that can perform more complex arithmetic functions. For example, we will introduce the concept of a solve gate that returns \mathbf{x} in $\mathbf{Ax} = \mathbf{b}$. We will detail how fundamental numerical nonlinear algebra concepts, such as autodifferentiation, are implemented within this framework. A central focus will be the adaptation of the homotopy continuation predictor-corrector method using Newton's method and Runge-Kutta 4. We will explain how to translate this powerhouse algorithm for solving nonlinear systems as an SLP. To validate this approach, we will demonstrate these methods on a collection of benchmark problems derived from the field of optical navigation for Keplerian orbits. By analyzing the performance of `HGates.m2` on these real-world conic problems, we will draw conclusions on the efficacy of SLPs in identifying solutions both quickly and accurately. Finally, we will discuss the broader implications of this work, exploring its connections to the rapidly advancing fields of artificial intelligence and formal verification.

1.2 Notation

Let $[n] = [1, n]$ for any positive integer n , and let \mathbb{F} denote a field.

Two important terms in complexity are big-O and big-Omega notation which provide an upper bound and a lower bound respectively on the growth of a function. Here we are referring to a function describing the complexity of an algorithm, so it is assumed to be nonnegative.

Definition 1.2.1. Big-O notation $f(n) = O(g(n))$ means that there exists a constant $c > 0$ and a number n_0 such that

$$0 \leq f(n) \leq cg(n)$$

for all $n \geq n_0$.

For example, if a function $f(n) = O(\log n)$, then $f(n) = O(n)$. If $f(n) \not\leq O(\log n)$, then f grows faster than $\log n$ and any constant $c > 0$.

Definition 1.2.2. Big-Omega notation $f(n) = \Omega(g(n))$ means that there exists a constant $c > 0$ and a number n_0 such that

$$0 \leq cg(n) \leq f(n)$$

for all $n \geq n_0$.

If a function $f(n) = \Omega(n)$, then also $f(n) = \Omega(\log n)$.

To combine the two, if $f(n) \not\leq O(g(n))$, then for any $c > 0$ and large enough n_0

$$0 \leq c(g(n)) < f(n)$$

for all $n \geq n_0$. Thus $f(n) = \Omega(g(n))$.

CHAPTER 2

STRAIGHT-LINE PROGRAMS

2.1 Polynomial Representations

Polynomials are ubiquitous in scientific computing, data analysis, and engineering softwares, and their internal representation within a computer plays a big role in real-life application performance optimization. We will introduce different representations for computing polynomials and give a notion of “complexity” to compare the efficiency of memory storage among the different representations.

A polynomial is defined as a sum of monomials, where a monomial is the product of a scalar and variables. For example, in $\mathbb{Q}[x, y, z]$, we may consider the polynomial

$$f(x, y, z) = x^6 + x^3y^3 + 10x^3y^3 + 6x^2y^2z + 12xyz^2 + 8z^3 + 25y^6$$

Definition 2.1.1. The *arithmetic formula* or formula for a polynomial is a representation consisting of variables, scalars, parentheses, and the arithmetic operations $+$, \times . It is defined recursively by:

- A scalar or a variable is a formula.
- If F_1, F_2 are formulae, then $F_1 + F_2$ and $F_1 \times F_2$ are formulae.

Then a formula for f is

$$\begin{aligned} F_1 = & x \times x \times x \times x \times x \times x + x \times x \times x \times y \times y \times y + 10 \times x \times x \times x \times y \times y \times y \\ & + 6 \times x \times x \times y \times y \times z + 12 \times x \times y \times z \times z + 8 \times z \times z \times z \\ & + 25 \times y \times y \times y \times y \times y \times y \end{aligned}$$

Definition 2.1.2. The *length* of a formula F is the number of $+$, \times operations in F with notation $|F|$.

In this formula F_1 of f , the length is 40. A polynomial does not necessarily have a unique formula. For example, notice $(xy+2z)^3+(x^3+5y^3)^2=(x^3y^3+6x^2y^2z+12xyz^2+8z^3)+(x^6+10x^3y^3+25y^6)=x^6+x^3y^3+10x^3y^3+6x^2y^2z+12xyz^2+8z^3+25y^6$. In the case of f , we may write the polynomial as

$$f(x, y, z) = (xy + 2z)^3 + (x^3 + 5y^3)^2$$

with formula

$$\begin{aligned} F_2 &= (x \times y + 2 \times z) \times (x \times y + 2 \times z) \times (x \times y + 2 \times z) \\ &\quad + (x \times x \times x + 5 \times y \times y \times y) \\ &\quad \times (x \times x \times x + 5 \times y \times y \times y) \end{aligned}$$

which has length 21.

Definition 2.1.3. The *formula complexity* of a polynomial f is the smallest length of its formulae and is denoted $\mathbf{L}(f)$.

$$\mathbf{L}(f) = \min_F |F|, \text{ any } F \text{ formula of } f$$

The definition of formula complexity presents a good metric for how “easy” a polynomial is to represent. In this example, F_2 is more efficiently represents f than F_1 as it uses fewer operations. Then (one of) the most efficient representation(s) of f has length $\mathbf{L}(f) \leq 21$.

For univariate polynomials, the formula complexity is known up to a linear factor. It can be bounded by $d - 1$ and $2d$ below and above respectively using Horner’s Rule, where d is

the degree of the polynomial. Things get slightly more complicated with multivariate polynomials, and we will need to introduce more ways to represent or compute polynomials, such as in Definition 2.2.1.

2.2 Arithmetic Operations

This subsection will introduce the concept of SLPs and will focus on arithmetic SLPs, which rely on arithmetic operations, $+$, \times , to represent polynomials. We will explore how SLPs offer a more efficient way to represent polynomials compared to arithmetic formulae. A formal definition of SLPs for generic operations will be provided in the following subsection.

Given a polynomial, there is at least one arithmetic circuit representation. An arithmetic circuit is a relation of variables, scalars, and arithmetic operations forming a directed acyclic graph where the variables and scalars are source nodes and the sinks are the outputs of the function it represents. An arithmetic SLP takes an arithmetic circuit and fixes an order of evaluation on the circuit. It can be thought of as a simple compiler where it specifies, line by line, how each computation is to happen in the program, hence the name “straight-line program”. The strength of an SLP is that it can remember data, whereas a formulae forgets and cannot reference any previous calculations. For example, the i th line in an SLP may reference as an input any previous line in the SLP.

An SLP of $f(x) = 3x^2 + 5x$ can be as the following, where $x, 3, 5$ are inputs and R_4 is

the output:

$$R_1 = x \times x,$$

$$R_2 = 3 \times R_1,$$

$$R_3 = 5 \times x,$$

$$R_4 = R_2 + R_3$$

The SLP complexity of f is the minimum number of lines in an SLP over all SLPs representing f and is denoted by $\mathbf{S}(p)$. It can also be referred to as the circuit complexity.

Any formula of a polynomial p can be written as a straight-line program so $\mathbf{S}(p) \leq \mathbf{L}(p)$. It is not always the case that there is equality, and this is nicely shown in the univariate setting with x^d . The formula complexity is $d - 1$, shown in [17], however, the circuit complexity is approximately $2 \log_2 d$. Consider the SLP for x^{2^k}

$$R_1 = x$$

$$R_2 = R_1 \times R_1$$

$$R_3 = R_2 \times R_2$$

$$\vdots$$

$$R_{k+1} = R_k \times R_k$$

where $R_{k+1} = x^{2^k}$ and $R_k = x^{2^{k-1}}$. There are about $\log_2 d \times$ operations, so $\mathbf{S}(x^{2^k})$ is quite a bit smaller than $\mathbf{L}(x^{2^k}) = 2^k - 1$.

This is where the ability of the SLP to remember computations comes in. Notice in the example that the i th line references the previous computation for $x^{2^{i-2}}$. Currently, for f a

univariate polynomial of degree d , we have the bound

$$\log d \leq \mathbf{S}(f) \leq \mathbf{L}(f) \leq 2d$$

Not much more is known, and in particular, it is unknown for which f , $\mathbf{S}(f)$ is not $O(\log d)$. By a result from Hrubeš and Yehudayof, it is believed there exists a univariate polynomial with circuit complexity $\geq c\sqrt{d}$ for some $c > 0$, however no explicit polynomial has been identified [17].

We present another interesting result from Lipton regarding factorial polynomials [18].

Theorem 2.2.1. Given a factorial polynomial $f_d(x) = (x - 1)(x - 2) \cdots (x - d)$, if $\mathbf{S}(f_d) \leq O(\log d)$, then there exists an efficient, or polynomial time, algorithm for integer factorization.

It is widely believed that integer factorization is difficult, which seems to imply that this given is not the case and that $\mathbf{S}(f_d) > O(\log d)$. The RSA cryptosystem is one of the most widely used encryptions and relies on the difficulty of integer factorization.

With the univariate case established, we turn our attention to multivariate polynomials, offering a generalization to the scope of the problem, and often of more interest to the complexity.

Definition 2.2.1. The *matrix multiplication tuple* MM_k is a tuple consisting of k^2 polynomials, which are sums of partial products:

$$\sum_{l=1}^k x_{i,l}y_{l,j} = x_{i,1}y_{1,j} + x_{i,2}y_{2,j} + \cdots + x_{i,k}y_{k,j}$$

for $1 \leq i, j \leq k$. It can also be represented as the entries of XY , where $X = \{x_{i,j}\}_{i,j=1}^k$ and $Y = \{y_{i,j}\}_{i,j=1}^k$.

The first entry of MM_k is $x_{1,1}y_{1,1} + x_{1,2}y_{2,1} + \cdots + x_{1,k}y_{k,1}$ which has circuit complexity $k - 1$. Since there are k^2 polynomials, it would seem to follow that $\mathbf{S}(MM_k) = \Omega(k^3)$,

however due to a long line of research including work by Strassen [19], Coppersmith and Winograd [20], and Williams [21], we know that $\mathbf{S}(MM_k) = O(k^\omega)$ where $\omega \leq 2.371552$.

Theorem 2.2.2. The circuit length of MM_k is in $O(k^{2.371552})$.

It is believed that for any $\epsilon > 0$, $\mathbf{S}(MM_k) = O(2^{k+\epsilon})$.

2.3 SLPs, Circuits, and Auto Differentiation

The generalized concept of circuits and straight-line programs, is an extension of arithmetic circuits and arithmetic SLPs to a more general form where the gates present in the circuit can represent arbitrary functions. As in the arithmetic setting, the explicit implementation of the gate is black-boxed. This generalization mirrors real-world software and hardware design principles where complex systems are built from gates whose internal workings are hidden. For example, a CPU has a floating-point unit whose internal workings are not exposed to the programmer using it. The programmer only needs to interact with the inputs and outputs of the floating-point unit, and the abstraction frees them from the complexities of its internal workings, allowing the implementation to be modified or optimized without impacting how programmers use it. This abstraction design supports smoother heterogeneous systems and smoother hierarchical systems.

Working over an arbitrary field \mathbf{F} , our building block is called a gate which computes a function with a single or multiple inputs in \mathbf{F} and a single or multiple outputs in \mathbf{F} .

Definition 2.3.1. A *gate*, g , represents a function $f : \mathbf{F}^m \rightarrow \mathbf{F}^l$.

Note that a variable or a scalar in \mathbf{F} is also a gate and is often referred to as an *input gate* or simply *input*.

There are different measurable properties of a gate, each providing unique advantages to analyzing efficiency or efficacy depending on the specific application. The most natural metric is to track the fan-out of a gate which can be used to capture both the number of gates in an SLP and the amount of data generated by intermediate gates. This will be called

the length of a gate. We will also introduce complexity for g , which will be dependent on the model of evaluation.

Definition 2.3.2. The *length* of g is $|g| = l$. The *complexity* of g is $\text{complexity}(g)$.

For example, the length of an arithmetic gate is 1, the length of a gate computing the determinant of a square matrix is 1, and the length of a gate solving for \vec{x} in the equation $A\vec{x} = \vec{b}$ is $|\vec{b}|$.

Definition 2.3.3. A *circuit*, representing a function $f : \mathbf{F}^m \rightarrow \mathbf{F}^l$, is a directed acyclic graph $G = (V, E)$ where V is a set of gates and E represents the input-output relations between gates. The set of source nodes contains all inputs to f and any scalars used in representing f and the set of sink nodes contains the outputs of f .

A circuit is a high-level representation of a function's computation by a network of interconnected gates. To move one level of abstraction down to execute the computation on a sequential machine, we will need a list of linear instructions.

A circuit for a function can be used to construct an SLP by fixing an ordering or labeling the vertices of the circuit such that any relation between gates goes from lower ordered or lower indexed vertices to higher ordered or higher indexed vertices. This transforms the circuit into a structured list of lines, each corresponding to a gate in the circuit and relying only on strictly preceding lines.

Definition 2.3.4. Let $G = (V, E)$ be a circuit representing the function $f : \mathbf{F}^m \rightarrow \mathbf{F}^l$. Fix a labeling from $[1, |V|]$ of V such that $(g_i, g_j) \in E$ satisfies $i < j$.

A *straight-line program* (SLP) is given by $P = (I, R, O)$ where

- I is the set of source nodes from V ,
- R is a list of V/I where each element or *line* is of the form:

$$R[i] = g_i(R[j_1], \dots, R[j_s])$$

or equivalently

$$R_i = g_i(R_{j_1}, \dots, R_{j_s})$$

where g_{j_1}, \dots, g_{j_s} is the complete in-coming neighborhood for g_i , and

- $O = \{i : R[i] \text{ a sink} \in V\}$.

Since G is a directed acyclic graph with the condition that the labeling takes lower-indexed vertices to higher-index vertices, it is guaranteed that a line $R[i]$ only depends on previously lines in the list. Given a circuit G , an SLP can be recursively built by specifying the inputs \mathcal{I} and outputs \mathcal{O} from G :

$$(\mathcal{I}, \mathcal{O}) \rightarrow (I, R, O)$$

The length of an SLP quantifies the total amount of output data passing around in the program it computes and inherently acts as an upper bound to the number of gates in the SLP. It is a very useful property to evaluate the memory usage of an SLP.

Definition 2.3.5. The *length* of an SLP $P = (I, R, O)$ is the sum of the length of gates G :

$$|P| = \sum_{g \in R} |g|$$

The *SLP complexity* of a function f is the minimum length of all SLPs representing f and is denoted as $\mathbf{S}(f)$.

While length measures the memory storage, depth is crucial for understanding the potential for parallel computation. It represents the longest chain of dependent operations, indicating the minimum time required to execute the SLP if an unlimited number of processors were available.

Definition 2.3.6. The *depth* of an SLP $P = (I, R, O)$ is the longest path in the corresponding circuit G , excluding source nodes.

The complexity of P is an extension from the complexity of a gate and represents a sum over all the gate complexities in the corresponding circuit.

Definition 2.3.7. The *complexity* of P is $\text{complexity}(P) = \sum_{g \in R} \text{complexity}(g)$.

2.4 Complexity Examples

This section presents illustrative examples of functions where we consider the problem of computing both the function value and its corresponding Jacobian. We will study upper bounds on the straight-line program (SLP) metrics required for each of their representations.

Given a function, if we are interested in taking its Jacobian, it is often computationally less expensive to consider the partial derivatives of smaller, “elementary” operations. In particular, the term we are interested in is *auto differentiation*, which provides a closed form expression or SLP of the Jacobian of the function in question.

Problem 1

Given an SLP with one gate, the determinant gate, going from $\mathbf{F}^{n^2} \rightarrow \mathbf{F}$, where $n \in \mathbb{N}$, return the SLP computing the given SLP of the determinant and the Jacobian of the determinant with respect to the inputs, and state the approximate number of non-input gates needed.

We will denote an abstract $n \times n$ matrix with gate elements as $X = (x_{ij})_{i,j \in [n]}$. The determinant gate, computing the determinant of X , is denoted $D : \mathbf{F}^{n^2} \rightarrow \mathbf{F}$ and is written $D(x_{1,1}, x_{1,2}, \dots, x_{2,1}, \dots, x_{n,n}) = \det(X)$. The matrix of vectors of gates M , given inputs v_1, \dots, v_k , v_i a vector with l gate elements, l, k positive integers, returns a matrix of size $l \times k$ and is written $M(v_1, \dots, v_k) = \begin{pmatrix} v_1 | & \dots & | v_k \end{pmatrix}$. The function $\text{diff}(x, g)$ represents the autodifferentiation of the gate g with respect to the variable x . In the SLP, we will use $d(x, g)$ for shorthand.

There are a few ways to calculate the determinant of an $n \times n$ matrix symbolically. One of the most common computational methods is using Gaussian elimination. The problem with that solution in our case is that it relies on conditionals such as identifying whether an entry is a pivot and which row operations to perform, so, giving the building blocks we have, it is not possible to compose a circuit or SLP mimicking Gaussian elimination unless there is some guarantee that the rows are ordered perfectly and there are never divide-by-zero scenarios. The Laplace expansion is a recursive formula based on summing over the determinants of smaller $(n - 1) \times (n - 1)$ matrices by omitting the i th row and j th column along a fixed row or column:

$$\det(X) = \sum_{j=1}^n (-1)^{i+j} x_{i,j} \det X_{i,j} \quad \text{Fixed along row } i \text{ (1)}$$

$$\det(X) = \sum_{i=1}^n (-1)^{i+j} x_{i,j} \det X_{i,j} \quad \text{Fixed along column } j$$

The notation $X_{i,j}$ denotes the $(n - 1) \times (n - 1)$ matrix obtained from X by removing the i th row and the j th column.

This is a much better formula in this context because all we have to work with are the basic arithmetic operations $+$, \times , the black box gate D returning a determinant, and the black box gate d computing the partial differential. The chain rule gives us that the partial of a determinant with respect to some variable x is the sum of the products of $\frac{\partial}{\partial x_{i,j}} D(X)$ and $\frac{\partial x_{i,j}}{\partial x}$ for all of the entries $x_{i,j}$ in X . Then there's an immediate simplification of $\frac{\partial}{\partial x_{i,j}} D(X)$ using the Laplace expansion (2):

$$\frac{\partial}{\partial x_{i,j}} D(X) \stackrel{(1)}{=} \frac{\partial}{\partial x_{i,j}} \left(\sum_{k=1}^n (-1)^{i+k} x_{i,k} D(X_{i,k}) \right) = (-1)^{i+j} D(X_{i,j})$$

The partial differential of the determinant with respect to x is

$$\begin{aligned}
\frac{\partial}{\partial x} D(X) &= \frac{\partial}{\partial x} D(x_{1,1}, x_{1,2}, \dots, x_{2,1}, \dots, x_{n,n}) \\
&= \sum_{i=1}^n \sum_{j=1}^n \left(\frac{\partial x_{i,j}}{\partial x} \right) \left(\frac{\partial}{\partial x_{i,j}} D(x_{1,1}, x_{1,2}, \dots, x_{2,1}, \dots, x_{n,n}) \right) \\
&\stackrel{(2)}{=} \sum_{i=1}^n \sum_{j=1}^n (-1)^{i+j} \left(\frac{\partial x_{i,j}}{\partial x} \right) D(X_{i,j}) \\
&= \sum_{i=1}^n \left((-1)^{i+1} \frac{\partial x_{i,1}}{\partial x} D(X_{i,1}) + \dots + (-1)^{i+n} \frac{\partial x_{i,n}}{\partial x} D(X_{i,n}) \right) \\
&= \sum_{i=1}^n D \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ (-1)^{i+1} \frac{\partial x_{i,1}}{\partial x} & (-1)^{i+2} \frac{\partial x_{i,2}}{\partial x} & \dots & (-1)^{i+n} \frac{\partial x_{i,n}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,n} \end{bmatrix}
\end{aligned}$$

The last step is recognizing the redundancy in determinant computations over each iteration of i , since the i th row is fixed. This allows the individual determinant gates to be merged into one determinant gate, reducing the total number of determinant gates needed per iteration of i from n to just one.

The following is the SLP computing the determinant of an $n \times n$ matrix X and the partial derivative with respect to x .

We will be assigning a gate to each input in the SLP.

$$R_1 = x_{11}$$

$$R_2 = x_{12}$$

⋮

$$R_{n^2} = nn$$

$$R_{n^2+1} = D(R_1, R_2, \dots, R_{n^2})$$

– differentiate all entries of X

$$R_{n^2+2} = d(x, R_1)$$

⋮

$$R_{2n^2+2} = d(x, R_{n^2})$$

– compute summands for partial of derivative with respect to x

$$R_{2n^2+3} = D(R_{n^2+2}, \dots, R_{n^2+n+2}, R_{n+1}, \dots, R_{n^2})$$

$$R_{2n^2+4} = D(R_1, R_2, \dots, R_n, R_{n^2+n+3}, \dots, R_{n^2+2n+2}, R_{2n+1}, \dots, R_{n^2})$$

⋮

$$R_{2n^2+n+2} = D(R_1, \dots, R_{n(n-1)}, R_{2n^2-n+3}, \dots, R_{2n^2+2})$$

– output

$$R_{2n^2+n+3} = R_{2n^2+3} + \dots + R_{2n^2+n+2}$$

The output O of the SLP is $\{n^2 + 1, 2n^2 + n + 3\}$ where the first element computes the determinant of X and the second element computes the partial derivative of the determinant of X with respect to the variable x . The addition operations in R_{2n^2+n+3} are combined into one gate representing the summation over all inputs.

If there are k variables, then the number of gates necessary to compute all partials is

$O(kn^2)$. Assuming $k = n^2$, the number of gates is in $O(n^4)$.

Problem 2

Given an SLP of length m for computing f , a polynomial map, from $\mathbf{F}^k \rightarrow \mathbf{F}^l$, where $k, l \in \mathbb{N}$, what is the length of an SLP G computing f and the Jacobian of f ?

Theorem 2.4.1. Given an SLP of length m for computing f , a polynomial map from $\mathbf{F}^k \rightarrow \mathbf{F}^l$ \mathbf{F} field, the SLP G computing f and the Jacobian of f has length at most $(3k + 1)m$.

Proof. Let $P = (I, R, O)$ be an arithmetic straight-line program computing $f(x_1, x_2, \dots, x_k)$ using m gates, where gates are the binary operations $+$, \times .

To compute the Jacobian, we want to compute the partial of f with respect to x_i for each $i \in [1, k]$. This can be done by computing the partial of P_1 with respect to x_i , which in turn can be done sequentially computing the partials of R_1 up to $R_{k'+m+1}$ based on the following observation. Each line R_j that is not an input or output involves a $+$ or \times gate and some R_a, R_b for $a, b < j$ not necessarily distinct, so the partial looks like:

$$\begin{aligned} R_j &= R_a + R_b \\ \frac{\partial R_j}{\partial x_i} &= \frac{\partial R_a}{\partial x_i} + \frac{\partial R_b}{\partial x_i} \end{aligned}$$

or using the product rule:

$$\begin{aligned} R_j &= R_a * R_b \\ \frac{\partial R_j}{\partial x_i} &= \frac{\partial R_a}{\partial x_i} * R_b + R_a * \frac{\partial R_b}{\partial x_i} \end{aligned}$$

It is possible that $R_j = R_a$ for some $a < j$, however this would be redundant, so we

may assume that is not the case. In the case that R_j is an input gate, the resulting partial will either be 0 or 1, and in the case that R_j is the last line of the SLP or the output, it will simply reference the previous partials already computed. This methodology is safe because by definition an SLP is a directed acyclic graph and each gates references gates of strictly less than indices.

The SLP of the Jacobian of f given P will use at most

$$|P| + \left| \frac{\partial P}{\partial x_1} \right| + \cdots + \left| \frac{\partial P}{\partial x_k} \right| = m + 3km = (3k + 1)m$$

gates since computing the partial of each gate R_j with respect to $x_i, i \in [1, k]$, uses at most 3 gates.

□

For example, an SLP for $f(x, y) = \begin{pmatrix} 3x^2y \\ xy^4 \\ y \end{pmatrix}$ is $P_f = (I, R, O)$ where the right-hand

side shows the evaluation of each gate:

$$R_1 = 3$$

$$R_2 = x$$

$$R_3 = y$$

$$R_4 = R_1 * R_2 \quad \Longrightarrow \quad 3x$$

$$R_5 = R_4 * R_2 \quad \Longrightarrow \quad 3x^2$$

$$R_6 = R_5 * R_3 \quad \Longrightarrow \quad 3x^2y$$

$$R_7 = R_3 * R_3 \quad \Longrightarrow \quad y^2$$

$$R_8 = R_7 * R_7 \quad \Longrightarrow \quad y^4$$

$$R_9 = R_2 * R_8 \quad \Longrightarrow \quad xy^4$$

$$O = [R_6, R_9, R_3] \quad \Longrightarrow \quad [3x^2y, xy^4, y]$$

The set of arithmetic gates $R = (R_4, \dots, R_9)$, so P_f has length 6.

For example, the partial derivative of $f(x, y)$ with respect to y can be obtained from P_f

as follows:

$$\begin{aligned}
\frac{\partial R_1}{\partial y} &= 0 \\
\frac{\partial R_2}{\partial y} &= 0 \\
\frac{\partial R_3}{\partial y} &= 1 \\
\frac{\partial R_4}{\partial y} &= \frac{\partial R_1}{\partial y} * R_2 + R_1 * \frac{\partial R_2}{\partial y} \\
\frac{\partial R_5}{\partial y} &= \frac{\partial R_4}{\partial y} * R_2 + R_4 * \frac{\partial R_2}{\partial y} \\
\frac{\partial R_6}{\partial y} &= \frac{\partial R_5}{\partial y} * R_3 + R_5 * \frac{\partial R_3}{\partial y} \\
\frac{\partial R_7}{\partial y} &= \frac{\partial R_3}{\partial y} * R_3 + R_3 * \frac{\partial R_3}{\partial y} \\
\frac{\partial R_8}{\partial y} &= \frac{\partial R_7}{\partial y} * R_7 + R_7 * \frac{\partial R_7}{\partial y} \\
\frac{\partial R_9}{\partial y} &= \frac{\partial R_2}{\partial y} * R_8 + R_2 * \frac{\partial R_8}{\partial y} \\
\frac{O}{\partial y} &= \left[\frac{\partial R_6}{\partial y}, \frac{\partial R_9}{\partial y}, \frac{\partial R_3}{\partial y} \right]
\end{aligned}$$

and uses 18 gates, or three times the number of $+$, \times gates in P_f . The partial derivative of $f(x, y)$ with respect to x can be found similarly and will also use 18 gates. An SLP computing f and the Jacobian of f can be obtained by $\left[P_f \quad \frac{\partial P_f}{\partial x} \quad \frac{\partial P_f}{\partial y} \right]$ which uses $6+18+18 = 42$ gates.

Evaluating each gate to check for correctness, we get:

$$\begin{aligned}
\frac{\partial R_1}{\partial y} &= 0 \\
\frac{\partial R_2}{\partial y} &= 0 \\
\frac{\partial R_3}{\partial y} &= 1 \\
\frac{\partial R_4}{\partial y} &= \frac{\partial R_1}{\partial y} * R_2 + R_1 * \frac{\partial R_2}{\partial y} = 0 \\
\frac{\partial R_5}{\partial y} &= \frac{\partial R_4}{\partial y} * R_2 + R_4 * \frac{\partial R_2}{\partial y} = 0 \\
\frac{\partial R_6}{\partial y} &= \frac{\partial R_5}{\partial y} * R_3 + R_5 * \frac{\partial R_3}{\partial y} = 0 + x * 3 * x * 1 && \implies 3x^2 \\
\frac{\partial R_7}{\partial y} &= \frac{\partial R_3}{\partial y} * R_3 + R_3 * \frac{\partial R_3}{\partial y} = 1 * y + y * 1 && \implies 2y \\
\frac{\partial R_8}{\partial y} &= \frac{\partial R_7}{\partial y} * R_7 + R_7 * \frac{\partial R_7}{\partial y} \\
&= (y + y) * (y * y) + (y * y) * (y + y) && \implies 4y^3 \\
\frac{\partial R_9}{\partial y} &= \frac{\partial R_2}{\partial y} * R_8 + R_2 * \frac{\partial R_8}{\partial y} \\
&= 0 + x * [(y + y) * (y * y) + (y * y) * (y + y)] && \implies 4xy^3 \\
\frac{O}{\partial y} &= \left[\frac{\partial R_6}{\partial y}, \frac{\partial R_9}{\partial y}, \frac{\partial R_3}{\partial y} \right] && \implies [3x^2, 4xy^3, 1]
\end{aligned}$$

Problem 3

Given a gate S returning \vec{x} in the expression $A\vec{x} = \vec{b}$ where A is an $n \times n$ matrix of gates and \vec{b} is a vector of n gates such that $S(A, \vec{b}) = \vec{x}$, write an SLP returning S and $\frac{\partial S}{\partial x}$ for some variable x .

First we will compute the partial derivative symbolically to arrive at a closed form

expression only relying on S . The partial derivative for \vec{x} with respect to x is:

$$\begin{aligned}\frac{\partial \vec{x}}{\partial x} &= \frac{\partial}{\partial x}(A^{-1}\vec{b}) \\ &= \left(\frac{\partial}{\partial x}A^{-1}\right)\vec{b} + A^{-1}\frac{\partial}{\partial x}\vec{b}\end{aligned}$$

The function $\frac{\partial}{\partial x}A^{-1}$ can be obtained from computing the partial of I :

$$\begin{aligned}0 &= \frac{\partial}{\partial x}I \\ &= \frac{\partial}{\partial x}(A^{-1}A) \\ &= \left(\frac{\partial}{\partial x}A^{-1}\right)A + A^{-1}\left(\frac{\partial}{\partial x}A\right) \\ \frac{\partial}{\partial x}A^{-1} &= -A^{-1}\left(\frac{\partial}{\partial x}A\right)A^{-1}\end{aligned}$$

Then continuing the calculation of the partial derivative of \vec{x} with respect to x :

$$\begin{aligned}\left(\frac{\partial}{\partial x}A^{-1}\right)\vec{b} + A^{-1}\frac{\partial}{\partial x}\vec{b} &= -A^{-1}\left(\frac{\partial}{\partial x}A\right)A^{-1}\vec{b} + A^{-1}\frac{\partial}{\partial x}\vec{b} \\ &= -M\left(S\left(A, \frac{\partial}{\partial x}A_1\right), \dots, S\left(A, \frac{\partial}{\partial x}A_n\right)\right)S(A, \vec{b}) + S\left(A, \frac{\partial}{\partial x}\vec{b}\right)\end{aligned}$$

where A_1, \dots, A_n are the columns of A .

The following is the SLP computing the solution to $A\vec{x} = \vec{b}$ for an $n \times n$ square matrix $A = (a_{ij})_{ij}$, a_{ij} a gate, and a vector $\vec{b} = (b_i)_i$ of length n and the partial derivative with

respect to the variable x .

$$R_1 = a_{11}$$

⋮

$$R_{n^2} = a_{nn}$$

$$R_{n^2+1} = b_1$$

⋮

$$R_{n^2+n} = b_n$$

$$R_{n^2+n+1} = S(R_1, \dots, R_{n^2}, R_{n^2+1}, \dots, R_{n^2+n})$$

– differentiate all entries of \vec{b}

$$R_{n^2+n+2} = d(x, R_{n^2+1})$$

⋮

$$R_{n^2+2n+1} = d(x, R_{n^2+n})$$

– differentiate all entries of A

$$R_{n^2+2n+2} = d(x, R_1)$$

⋮

$$R_{2n^2+2n+1} = d(x, R_{n^2})$$

– compute $A^{-1} \frac{\partial}{\partial x} \vec{b}$

$$R_{2n^2+2n+2} = S(AR_1, \dots, R_{n^2}, V(R_{n^2+n+2}, \dots, R_{n^2+2n+1}))$$

– compute $(\frac{\partial}{\partial x} A^{-1}) \vec{b}$

$$R_{2n^2+2n+3} = S(R_1, \dots, R_{n^2}, V(R_{n^2+2n+2}, R_{n^2+3n+2}, \dots, R_{2n^2-n+2}))$$

⋮

$$R_{2n^2+3n+2} = S(R_1, \dots, R_{n^2}, V(R_{n^2+3n+1}, R_{n^2+4n+1}, \dots, R_{2n^2+2n+1}))$$

$$R_{2n^2+3n+3} = -1 * M(R_{2n^2+2n+3}, \dots, R_{2n^2+3n+2}) * R_{n^2+n+1} + R_{2n^2+2n+2}$$

This too uses $O(kn^2)$ gates to compute the Jacobian for k variables.

CHAPTER 3

MACAULAY2 CODE

Macaulay2 is a free, open-source software system for research in algebraic geometry and commutative algebra. It is designed to handle complex computations with polynomials and supports ring theoretic structures, making it a primary tool for researchers in these fields [22]. We will write `HGates.m2` in Macaulay2.

This section will outline all of the gates in `HGates.m2` to construct the basis for our framework, enabling the building of functions like Newton’s Method, the Trapezoid Method, and Runge-Kutta, and homotopy continuation. In addition to introducing gates, we will introduce the concept of a map which is like a meta gate, taking a set of gates as inputs and outputting a set of gates. This is a nice data structure which allows for more flexibility in referencing a collection of gates, or a subcircuit, which will become apparent later on with applications in Newton’s Method and homotopy continuation.

3.1 Types and Methods

Definition 3.1.1. The generic data type for a gate emulating a function $f : \mathbb{F}^k \rightarrow \mathbb{F}^l$, where \mathbb{F} is a field and k, l are positive integers, is called an `HGate`¹. Gates defined in `HGates.m2` will be of type `HGate` and will inherit from it. The naming convention for gates in `HGates.m2` is that gates where $l = 1$ end in `HGate` and gates where $l > 1$ end in `HMatrixGate`. Another type of note is `InputValueTable` which is what we will use to evaluate or to specialize gates. The `InputValueTable` type represents a hashtable mapping variables the scalars they will be evaluated on. The following are methods supported for all `HGates`:

- `length HGate` - returns l , the number of outputs in f

¹Pronounced “H-gate”, `HGate` is short for hyper gate, named after a hyperedge in a hypergraph.

- `net HGate` - prints the contents of the gate and its functionality with formatting
- `diff (InputHGate, HGate)` - given a variable x , returns an `HGate` of the differentiation of f with respect to x based on the product and chain rule
- `specialize (HGate, InputValueTable)` - given a table of evaluations L mapping variables to scalars, returns an `HGate` of the evaluation of f with respect to L

The following are the specific gate types inheriting from `HGate`:

- `InputHGate` - either a variable or a scalar. Input gates can be substituted with other gates, not necessarily of type input gate
 - `subGate (InputHGate, HGate, HGate)` - returns the third gate with all references of the first input gate replaced by the second gate
- `SumHGate` - represents the sum of two `HGates` of length 1
- `ProductHGate` - represents the product of two `HGates` of length 1
- `DetHGate` - represents the determinant of an `HMatrixGate` with size n^2
- `ElementHGate` - represents the i th entry of an `HMatrixGate`, provided an appropriate $i \in [\text{length HMatrixGate}]$

Next we introduce `HMatrixGate` which will be the second most abstract gate type, representing the class of gates with more than one output. We list its type specific methods

- `HMatrixGate` - represents an $n \times m$ matrix with attributes `Elements`, `Rows`, `Cols` storing the entries of the matrix as a list, the number of rows, and the number of columns respectively.
 - `length HMatrixGate = Rows * Cols`

- `flatten HMatrixGate` - flattens `Elements` to a one-dimensional list where each entry is a gate of length 1
- `jacobian (HMatrixGate, HMatrixGate)` - given a vector of n variables in the form of input gates and a vector of m functions containing the n variables, returns the $n \times m$ jacobian matrix as an `HMatrixGate`

The following gates are of type `HMatrixGate`:

- `SumHMatrixGate` - represents the sum of two `HMatrixGates` where both matrices have the dimensions
- `ProductHMatrixGate` - represents the product of two `HMatrixGates` with dimensions $n \times m, m \times k$ respectively
- `ScalarProductHMatrixGate` - represents the product of an `HMatrixGate` and an `HGate`
- `SolveHMatrixGate` - represents the solution \vec{x} in the system $A\vec{x} = \vec{b}$ where A an $n \times n$ matrix and \vec{b} of length n are given as `HMatrixGates`

Finally we introduce the `HMap`, which is distinct from a gate:

- `HMap` - represents a function of gates with attributes `InputGates` storing the set of input gates as a list and `OutputGates` storing the set of input gates as a list. It has support for `net` and `specialize`

Definition 3.1.2. A straight-line program of a function $\mathbb{F}^k \rightarrow \mathbb{F}^l$ on length m with K variable and scalar gates satisfying Definition 2.3.4 can be obtained by the method `printSLP`. `printSLP` takes in a list of input gates and output gates and prints an SLP where each line is classified as one of the following:

- `Ixx = X` - denotes an input gate referencing a variable X

- $C_{xx} = C$ - denotes an input gate referencing a scalar C
- $R_{xx} =$ - denotes a line referencing a gate and referencing one or more previous SLP lines as gate inputs
- OUTPUT: - denotes an output of the SLP, referenced by a line number

For example, given an arithmetic circuit recursively defined by `e` with outputs `e`, `a`:

```
declareVariable {x, y}
a = x + x
b = a + y
c = hMatrixGate({b}, 1, 1)
d = inputHMatrixGate 3
e = c + d
printSLP ({x, y}, {e, a})
```

we can print a corresponding SLP with `printSLP (x, y, e, a)`:

```
C0 = `3`
I0 = `x`
I1 = `y`
R0 = I0+I0
R1 = R0+I1
R2 = matrix{R1} (1, 1)
R3 = R2+C0
OUTPUT: R3 = R2+C0
OUTPUT: R0 = I0+I0
```

This is written in `HGates.m2` as `printSLP` which takes in two lists, one representing the inputs to the program and one representing the outputs to the program, and returns an SLP of the gate circuit. As mentioned before, this is particularly useful when calling

specialize to evaluate the program because gates that are referenced multiple times need only be evaluated once in this model. This method will directly print the SLP to terminal, given by lines starting with <<:

```
printSLP = method()
printSLP (List, List) := (I, O) -> (
  if not all(I, (e -> instance(e, HGate))) then error
    "Error, I is not a list of HGates";
  if not all(O, (e -> instance(e, HGate))) then error
    "Error, O is not a list of HGates";
  p := newPrintIndices " = ";
  O / (g -> printHGate(g, p));
  unsortedLines := values p#"gates" / (l -> l#0 |
  p#"assignmentSymbol" | l#1 );
  << "[printSLP]: print sorted lines" << endl;

  sort unsortedLines / (slpLine -> << slpLine << endl);
  O / (g -> << "OUTPUT: "
    << ((p#"gates")#g)#0 << p#"assignmentSymbol"
    << ((p#"gates")#g)#1 << endl);
)
```

CHAPTER 4

SOLVING SYSTEMS OF EQUATIONS

The `HGates.m2` framework is designed to be applicable to a wide range of numeric problems, from solving linear equations to complex nonlinear equations, supporting both real and complex fields. This section will be dedicated to walking through commonly-used types of methods to numerically solve systems of nonlinear equations, how we can leverage linear algebra algorithms with `HGates.m2`, and how we can apply the `HGates.m2` nonlinear algorithms to themselves in an SLP. In particular, we will demonstrate the versatility of `HGates.m2` with a collection of code showcasing how to directly solve nonlinear systems of equations through methods using gates and then how to write these methods as gates and apply them to themselves.

To begin, we will introduce the well-known root-finding algorithm Newton's method and the well-known ordinary differential equation prediction method Runge-Kutta and explain how each can be defined as methods using the gates in `HGates.m2`. Next, we will introduce the concept of homotopy continuation to solve a complex system of nonlinear equations, the predictor-corrector method and define how to compose linear homotopies by defining a known total-degree starting system to path-traverse to solutions in the target system. Finally, we will explain how to write Runge-Kutta and other predictors as gates, discuss their SLP complexities, and give examples with Newton's method applied to the gate to recover initial values.

4.1 Newton's Method

Given a polynomial, it is a natural question to ask what is a zero of the polynomial. There are a couple components to answering this question in the numerical setting. Often the solution cannot be obtained analytically, so a functional model is necessary to approximate

the answer. Then it follows there must be a metric to determine whether or not the answer is good. In particular, suppose that we have an algorithm which claims to produce a root to our polynomial. It may not be possible to return an exact answer, such as with the polynomial $x^2 - 2$ the computer will return an approximation of $\sqrt{2}$. This is where the metric is necessary. With real or complex values, whether an approximation is considered “close” is usually determined by the 2-norm of the difference between the approximation and the true root it is approximating and the 2-norm of the polynomial evaluated on the approximation. This problem can be described formally as follows, where \bar{x} is the true root and x^* the approximation.

Example. Consider a polynomial $f(x) \in \mathbb{F}[x]$. Let \bar{x} be a true zero of f such that $f(\bar{x}) = 0$ and fix any $\delta, \epsilon > 0$. Let $x^* \in \mathbb{F}$ satisfy the following

$$\|x^* - \bar{x}\| < \delta \quad \text{and} \quad \|f(x^*)\| < \epsilon$$

where $\|\cdot\|$ is a norm. If $\mathbb{F} = \mathbb{R}$ or \mathbb{C} , the $\|\cdot\|$ is usually the standard 2-norm.

Newton’s method is an algorithm used to find zeroes and starts with an initial value x_0 and calculates another value x_1 by calculating the intersection of the tangent line at x_0 with the x -axis. In this fashion, it can be used to calculate values x_2, x_3, x_4 and so on. In the univariate case, the tangent line at x_k can be computed by solving for the y -intercept:

$$y = mx + b$$

$$f(x_k) = f'(x_k)x_k + b$$

$$b = f(x_k) - f'(x_k)x_k$$

resulting in $y = f'(x_k)x + f(x_k) - f'(x_k)x_k$. Then the intersection with the x -axis can be

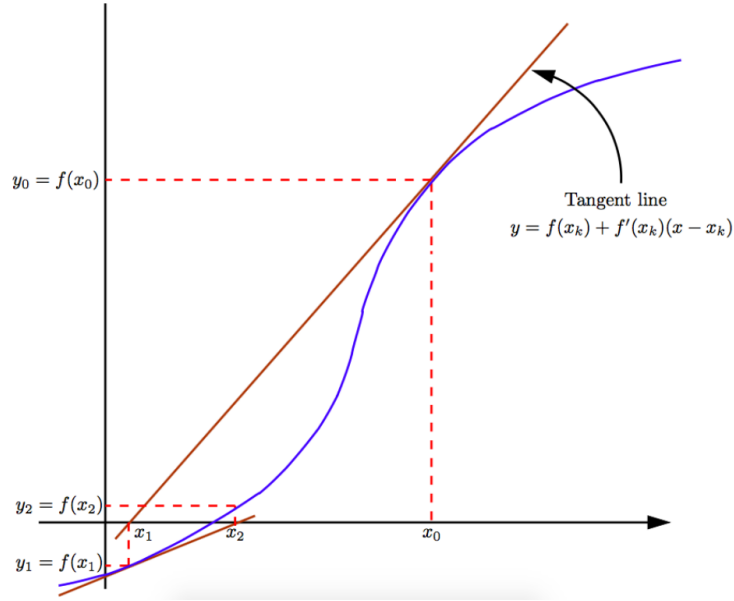


Figure 4.1: Two Iterations of Newton's Operator [23]

found by solving for x_{k+1} at $(x_{k+1}, 0)$:

$$0 = f'(x_k)x_{k+1} + f(x_k) - f'(x_k)x_k \quad (4.1)$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad (4.2)$$

Equation 4.2 is referred to as Newton's operator, representing one iteration of Newton's method.

This process is concisely given by the function called Newton's operator:

Definition 4.1.1 (Univariate). Given a polynomial $f \in \mathbb{F}[x]$, *Newton's operator* is defined as

$$N_f : \mathbb{F} \rightarrow \mathbb{F}, N_f(x) = x - \frac{f(x)}{f'(x)}$$

and is illustrated in Figure 4.1 by two applications of Newton's operator.

It is easy to see Newton's operator is well-defined for $x_{k+1} = N_f(x_k)$ if $f'(x_k) \neq 0$. When $f'(x^*) \neq 0$, x^* is called a regular root. If the initial value x_0 is "good" such that the sequence x_0, x_1, x_2, \dots converges to a regular root of f , then the sequence will converge

quadratically to the regular root. Given a $\delta > 0$ and $\epsilon > 0$, the algorithm terminates and a value x_k is selected when $\|x_k - x_{k-1}\| < \delta$ and $\|f(x_k)\| < \epsilon$. This value δ is called the absolute error tolerance, and the value ϵ is referred to as the absolute residual tolerance.

There are various values that are not good where termination of Newton's method will not occur. For example, x_0 may yield a loop $x_l, x_{l+1}, \dots, x_{l+c}$ such that $f(x_l) = f(x_{l+c})$, x_0 may lead to a steep slope with little variation between subsequent x_k, x_{k+1} , or x_0 may lead to a slope of zero in which case the method fails completely. Another problem occurs when x^* is a multiple root because then no guarantee can be made about the rate of convergence, and in particular, the nice guarantee in the regular root case for quadratic convergence is lost. Yet another problem is when the slope at a point x_k is large because this implies that the slope is near vertical and the subsequent values will be at the same intercept point, leading to the sequence to be stuck.

Remark. Newton's method can be used with an arbitrary initial value, though it is clear that the method is heavily dependent on the initial value as the equation becomes more complex. An algorithm's efficacy can be measured by its backwards stability, which is interested in the approximated answer and the difference between the problem with an exact solution equaling the approximated answer with the actual problem. In linear algebra setting, the merit of an algorithm involves its backward stability guarantee. Although Newton's method will use gates that most likely rely on backwards stable algorithms to solve the linear system of equations over a particular x , it itself is not backwards stable. Newton's method is not backwards stable as a slight perturbation can cause a drastic effect, depending on the complexity of the equation, however it is used in practice due to its computational speed when close to the answer.

Definition 4.1.2 (Multivariate). Given a polynomial $F : \mathbb{F}^k \rightarrow \mathbb{F}^l$, Newton's operator is defined as

$$N_F : \mathbb{F}^k \rightarrow \mathbb{F}^k, N_F(x) = x - \left(\frac{\partial F}{\partial x}(x)\right)^{-1}F(x)$$

where $(\frac{\partial F}{\partial x})^{-1}$ is the inverse of the jacobian matrix of F .

As in the univariate case, Newton's operator relies on the jacobian to be nonsingular at x , and it is standard to assert that x satisfies $\det(\frac{\partial F}{\partial x}(x)) \neq 0$.

For the remainder of this paper, we will assume our polynomial f is of the form $\mathbb{F}^k \rightarrow \mathbb{F}^k$ such that its jacobian is square, and we will call such systems with k variables and k equations a $k \times k$ square system, a k -square system, or a square system.

Observe that the multivariate Newton's operator is equivalent to solving the system of equations for $x - N_F(x)$:

$$\left(\frac{\partial F}{\partial x}(x)\right)(x - N_f(x)) = F(x) \tag{4.3}$$

This observation is essential because it provides a way to solve nonlinear systems with linear methods. Since the jacobian evaluated at an x is a matrix of scalars, the $(x - N_F(x))$ in Equation Equation 4.3 is the solution to a system of linear equations and can be expressed with solve gate.

We define Newton's operator in `HGates.m2` for $F : \mathbb{F}^k \rightarrow \mathbb{F}^k$, $k \geq 1$, by using `SolveHMatrixGate`. Given an `HMap g` representing the polynomial F with one input `HMatrixGate` holding the input vector x and one output `HMatrixGate` also of length k , the method `newtonsOp` returns an `HMap` for Newton's Operator:

```
newtonsOp(HMap) := g -> (
  P := g.OutputGates#0;
  Y := g.InputGates#0;

  J := jacobian(Y, P);
  hMap({Y}, {Y - solveHMatrixGate(J, P)})
)
```

We can study the SLP for this method with an example.

Example. Consider the equations $f_1(x, y) = x^2 + 1$ and $f_2(x, y) = y^3 - 2$. The corresponding SLP for one application of Newton's operator is

```

C0 = -1
C1 = 0
C2 = 1
C3 = 2
I0 = x[0]
I1 = x[1]
R0 = matrix{I0, I1} (2, 1)
R1 = I0+I0
R2 = I1*I1
R3 = I1+I1
R4 = I1*R3
R5 = R2+R4
R6 = matrix{R1, C1, C1, R5} (2, 2)
R7 = I0*I0
R8 = R7+C2
R9 = R2*I1
R10 = C0*C3
R11 = R9+R10
R12 = matrix{R8, R11} (2, 1)
R13 = solve{R6, R12}
R14 = scalarMatrixProduct(C0, R13)
R15 = matrixSum(R0, R14)
OUTPUT: R15 = matrixSum(R0, R14)

```

where lines R1 through R6 represent the jacobian of $(f_1, f_2)^T$, lines R7 to R12 build the

vector $(f_1, f_2)^T$, line R13 solves the linear system of equations, and lines R13 to R15 perform the remaining operations. The complexity based on the type of gates is ten arithmetic gates, 3 matrix instantiations, 2 matrix operations for matrix sum and scalar matrix product, and 1 solve gate.

The SLP complexity for Newton's operator is determined by both variable and fixed components. The variable part, which changes with the specific system, includes the arithmetic gates needed to compute the system equations and perform autodifferentiation, along with the matrix instantiation gates. The fixed part consists of a constant number of matrix operation gates: one matrix sum, one scalar matrix product, and one solve gate. The matrix operation gates will consume more resources in application, especially the solve gate, so it is important that they are distinct from the arithmetic gates.

4.1.1 Examples

Suppose we have a polynomial $F : \mathbb{F}^k \rightarrow \mathbb{F}^k$, and we want to find a value \bar{x} such that $F(\bar{x}) = y_t$ for some given y_t . Then we can define a polynomial $G(x) = F(x) - y_t$ and use Newton's method to identify a root of G which will be a solution to our problem. The following are examples of this in `HGates.m2` :

Example. Let $f_1(x) = x^2 + 2x + 5$. What value $x \in \mathbb{R}$ does $f_1(x) \approx 8$ with a difference of less than 0.01 in the 2-norm?

Answer. First we demonstrate one application of Newton's Operator and then we write a loop to answer the question. Define $f_2(x) = f_1(x) - 8$. Notice that $f_2'(0) = 2 \neq 0$, so we may pick the initial value to be 0. Then evaluating $N_{f_2}(x) = x - (2x + 2)^{-1}(x^2 + 2x - 3)$ at 0, we get $N_{f_2}(0) = -(2)^{-1}(-3) = \frac{3}{2}$. Using `HGates.m2` we define Newton's operator for f_2 :

```
declareVariable \ {x}
f_1 = x*x + (inputHGate 2)*x + (inputHGate 5)
```

```

f_2 = f_1 - (inputHGate 8)

X = hMatrixGate({x}, 1, 1)
F = hMatrixGate({f_2}, 1, 1)

g_1 = hMap({X}, {F})
g_2 = newtonsOp(g_1)

```

We confirm that the derivative is indeed not zero and that one application of Newton's operator yields 1.5:

```

R = RR_53
L = inputValueTable {x => 0_R}
-- confirm f_2'(x) \neq 0
df_2 = diff (x, f_2)
(specialize(df_2, L))#0 != 0
-- expect 1.5
x1 = (specialize (g_2, L))#0;
(x1 == (3/2)_R)

```

Next we extend this method to return a value x answering the question:

```

track = 0 -- track iterations
x0 = 0_R
while (abs(x0-x1) >= 0.1) do (
  x0 = x1;
  L = inputValueTable {x => x0};
  assert ((specialize(df_2, L))#0 != 0);

```

```

x1 = (specialize (g_2, L))#0;
track = track + 1
)

```

Newton's method terminates after two iterations with the value 1.00061. Our function f_1 evaluates to $f(1.00061) = 8.00244$, which has a difference of 0.0024394 satisfying the requirements of the question. \square

Example. Let $F_1 : \mathbb{R}^3 \rightarrow \mathbb{R}^3$, $F_1(X) = \begin{bmatrix} x_1^2 x_2 + 1 \\ x_1 x_3 + 2x_2 \\ x_3^2 + 3 \end{bmatrix}$ where $X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \in \mathbb{R}^3$. What

value X does $F_1(X) \approx \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ such that the difference under the 2-norm is less than 0.01?

Answer. We will compute one application of Newton's operator by hand, then we will present the initial setup and walk through of one application of Newton's operator with `HGates.m2` and verify the answer with the hand-computed result.

Define $F_2(X) = F_1(X) - \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} x_1^2 x_2 \\ x_1 x_3 + 2x_2 - 2 \\ x_3^2 \end{bmatrix}$. The determinant of the jacobian

at $X_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}^T$ is

$$\det \left(\frac{\partial F_2}{\partial X} \left(\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T \right) \right) = \det \begin{bmatrix} 2x_1x_2 & x_1^2 & 0 \\ x_3 & 2 & x_1 \\ 0 & 0 & 2x_3 \end{bmatrix}$$

$$= (2 * 1 * 1) * 2 * (2 * 1) = 8$$

So X_0 is safe to use as an initial point for Newton's operator. The next value X_1 can be computed as follows:

$$N_{F_2}(X_0) = X_0 - \left(\frac{\partial F_2}{\partial X}(X_0) \right)^{-1} F_2(X_0)$$

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

This can be rearranged as a linear system of equations

$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix} (X_0 - N_G(X_0)) = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

where $\begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}^T = \begin{pmatrix} 1/2 & 0 & 1/2 \end{pmatrix}^T$.

In `HGates.m2`, we define $F_2(X)$ ¹, its corresponding `HMap`, and one iteration of Newton's operator:

```
declareVariable \ {x, y, z, x0, x1, y0, y1, z0, z1}
```

¹We will use the variables x, y, z instead of x_1, x_2, x_3 in Macaulay2 for legibility

```

f_1 = x*x*y + oneHGate
f_2 = x*z + (twoHGate*y)
f_3 = z*z + threeHGate
X = hMatrixGate({x,y, z}, 3, 1)
F_1 = hMatrixGate({f_1, f_2, f_3}, 3, 1)
Yt = hMatrixGate({oneHGate, twoHGate, threeHGate}, 3, 1)
F_2 = F_1 - Yt -- F_1 - target value

G_1 = hMap({X}, {F_2})
G_2 = newtonsOp(G_1)

```

We compute the jacobian of F_2 at $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$ and confirm that the determinant is nonzero.

Then we check that one application of Newton's operator with initial value $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$ is

$\begin{pmatrix} 0.5 & 0 & 0.5 \end{pmatrix}^T$:

```
R = RR_53
```

```
X0 = {1_R, 1_R, 1_R}
```

```
L = inputValueTable {x => X0#0, y => X0#1, z => X0#2}
```

```
-- confirm det jac F_2' (1) \neq 0
```

```
dF_2 = jacobian (X, F_2)
```

```
detdF_2 = detHGate dF_2
```

```
assert((specialize(detdF_2, L))#0 != 0)
```

```
X1 = specialize(G_2, L)
```

```
assert(X1 == {0.5, 1, 0.5})
```

Then we extend to Newton's method to return a value X satisfying the question:

```
track = 0 -- track iterations
while (sqrt fold(plus, (X1 - X0)/(i -> i*i)) >= 0.1) do (
  X0 = X1;
  L = inputValueTable {x => X0#0, y => X0#1, z => X0#2};
  assert((specialize(detdF_2, L))#0 != 0);
  X1 = specialize(G_2, L);
  track = track + 1
)
```

After three iterations, the loop terminates and returns the vector $\begin{pmatrix} 0.0625 & 1 & 0.0625 \end{pmatrix}^T$. When this result is input into the function F_1 , the output is $\begin{pmatrix} 1.00391 & 2.00391 & 3.00391 \end{pmatrix}$. The accuracy of this process can be measured by its error under the 2-norm, which is 0.00676582, when compared with the desired target value $\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}^T$. \square

We can write an algorithm for Newton's method using Newton's operator, which intakes an HMap and a list of values in \mathbb{R}_{53} . It then returns the final solution, which is found by iteratively applying Newton's operator until the difference between successive iterations is below the specified threshold of 0.01 under the 2-norm and evaluation of the function is below 0.01.

```
newtonsMethod(HMap, List) := (g, X0) -> (
  G := newtonsOp(g);

  -- initialize values
  R = RR_53;
```

```

X := g.InputGates#0;
Xlist := X.Elements;
n := #Xlist;
L := inputValueTable (toList (0..n-1)/
  (i -> Xlist#i => X0#i));

-- check jacobian is safe
F := g.OutputGates#0;
dF := jacobian (X, F);
detdF := detHGate dF;
assert((specialize(detdF, L))#0 != 0);

-- run Newton's Method
X1 := specialize(G, L);
-- iterate over X_k until we find one satisfying
-- ||X_k - X_{k-1}||_2 < 0.1
track := 0; -- track iterations
while (sqrt fold(plus, (X1 - X0)/(i -> i*i)) >= 0.01)
  do (
    X0 = X1;
    L = inputValueTable (toList (0..n-1)/
      (i -> Xlist#i => X0#i));
    assert((specialize(detdF, L))#0 != 0);
    X1 = specialize(G, L);
    track = track + 1;
  );
X1

```

)

Remark. In `HGates.m2`, Newton's method is not a gate or a map because it intakes an `InputValueTable` and internally evaluates after each call of Newton's operator.

4.2 Runge-Kutta Methods

In differential equations, if one knows that value of a function at a certain time, it is often of interest to ask what the next value will be after a particular timestep. This tool for this question is called a predictor, which at the highest level predicts or approximates the next value of a function after a given timestep based on differentials. In this section, we will introduce Runge-Kutta 4 (RK4), one of the most common predictors of ordinary differential equations as well as solving initial value problems. There are different variants of Runge-Kutta, based on the order of the predictor which describes the bound on the error approximation.

Runge-Kutta, as well as most other predictor methods, relies on taking the slope of F at a particular point. The differential of F with respect to t can be computationally calculated using the chain rule and taking the inverse of the jacobian of F , denoted $c(t, X)$, and the slope can be found at any given point (t, X) . We will assume that the determinant of the jacobian of F is nonzero for any given (t, X) :

$$\begin{aligned}\frac{\partial}{\partial t}(F) &= 0 \\ \frac{\partial F}{\partial X} X'(t) + \frac{\partial F}{\partial t} &= 0 \\ c(t, X) := X'(t) &= -\left(\frac{\partial F}{\partial X}\right)^{-1} \frac{\partial F}{\partial t}\end{aligned}$$

In the code implementation, we make the assumption that the gate F is written in terms of t an `InputHGate` and X an `HMatrixGate` where $X.Elements$ is a list of input gates. By assumption, the jacobian of F at (t, X) is invertible, so we may use `solve gate` to solve for c . The code is:

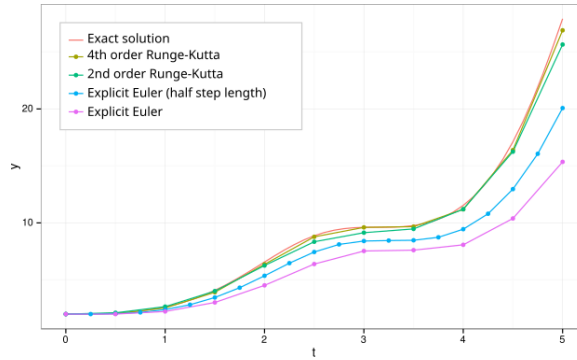


Figure 4.2: Comparison of Different Order Runge-Kutta methods for $y' = y \sin(t^2)y$ [24]

```

c (InputHGate, HMatrixGate, HMatrixGate) := (t, X, F) -> (
  assert (X.Cols == 1 and all(X.Elements,
    (x -> instance(x, InputHGate)))));

dFdx := jacobian (X, F);
dFdt := diff(t, F);
solveHMatrixGate(dFdx, dFdt)
)

```

An m th order scheme means that the error approximation is in $O(\Delta t^{m+1})$ where $\Delta t = t_1 - t_0$ and $\Delta \rightarrow 0$. The most common order is 4 due to its balance between accuracy, ease of programmability, and computation time. RK4 takes four intermediate values with $\Delta = t_1 - t_0$:

$$\begin{aligned}
 R_1 &= c(t_0, X_0)\Delta t \\
 R_2 &= c\left(t_0 + \frac{\Delta t}{2}, X_0 + \frac{R_1}{2}\right)\Delta t \\
 R_3 &= c\left(t_0 + \frac{\Delta t}{2}, X_0 + \frac{R_2}{2}\right)\Delta t \\
 R_4 &= c(t_0 + \Delta t, X_0 + R_3)\Delta t \\
 X_1 &= X_0 + \frac{R_1 + 2R_2 + 2R_3 + R_4}{6}
 \end{aligned}$$

A comparison of the first four orders of Runge-Kutta are given in Figure 4.2 for the differential $y' = y \sin(t^2)$ where it is easy to see that the 4th order Runge-Kutta method performs quite well. In industry, Matlab has a built-in variant of the 4th-order called `ode45` where the 4 and 5 refer to RK4 and RK5 respectively.

We now define the `predictorRK4` method, which intakes an `HMap` and a list of values and returns a list of values for the predicted next value:

```

predictorRK4 (HMap, List) := (g, I) -> (
  -- unpack inputs
  t0 := I#0; -- value
  t1 := I#1; -- value
  X0 := I#2; -- list of values
  t := g.InputGates#0;
  X := g.InputGates#1;
  Xlist := X.Elements;
  F := g.OutputGates#0;

  -- substitute values
  n := #X0;
  c1 := c(t, X, F);
  d := t1 - t0;

  -- compute R_1
  Llist := append(toList (0..n-1)/
    (i -> Xlist#i => X0#i), t => t0);
  L := inputValueTable Llist;
  c11 := specialize(c1, inputValueTable L);
  R1 := c11/(e -> e*d);

```

```

-- compute R_2
tmid := t0 + (0.5*d);
X0plusR1 := toList (0..n-1)/(i -> X0#i + (.5*R1#i));
Llist = append(toList (0..n-1)/
  (i -> Xlist#i => X0plusR1#i), t => tmid);
L = inputValueTable Llist;
c21 := specialize(c1, L);
R2 := c21/(e -> e*d);

-- compute R_3
X0plusR2 := toList (0..n-1)/(i -> X0#i + (.5*R2#i));
Llist = append(toList (0..n-1)/
  (i -> Xlist#i => X0plusR2#i), t => tmid);
L = inputValueTable Llist;
c31 := specialize(c1, L);
R3 := c31/(e -> e*d);

-- compute R_4
tmore := t0 + d;
X0plusR3 := toList (0..n-1)/(i -> X0#i + R3#i);
Llist = append(toList (0..n-1)/
  (i -> Xlist#i => X0plusR3#i), t => tmore);
L = inputValueTable Llist;
c41 := specialize(c1, L);
R4 := c41/(e -> e*d);

```

```

toList ((0..n-1)/
  (i -> X0#i +
    (1/6 * (R1#i + (2. * R2#i) + (2. * R3#i) + R4#i))))
)

```

Remark. As with Newton's method, `predictRK4` is a method and not a gate or a map in `HGates.m2`. It takes in a list of values and internally evaluates each call of c before passing the outputs to the next R_i . As with Newton's operator, RK4 can be written as a gate, as we will show at the end of the section.

4.3 Homotopy Continuation

For a nonlinear system $F(X) = 0$ it is often the case that $F(X)$ is difficult to directly solve. If $F(X)$ is a polynomial system one well-used method is to consider an easier polynomial system to solve, $G(X)$, and build a homotopy from $F(X)$ to $G(X)$:

$$H(X, t) = (1 - t)G(X) + tF(X) = 0 \quad (4.4)$$

where $t \in [0, 1]$. When $t = 0$, then $H(X, T) = G(X) = 0$ and the solution is the known solution. The predictor-correcter method can be used to step along this path, starting with predicting the solution X^* to $H(X, \Delta t) = 0$ from \bar{X} , a known zero of G , correcting the approximation X^* with Newton's method, and repeating this predict-correct process until reaching $t = 1$. An illustration of this process is given in Figure Figure 4.3, where there is a path from the starting solution at time $t = 0$ to the target solution at $t = 1$ and along the path, the purple arrows represent predictor steps, and the green arrows represent corrector steps. In this section, we will focus on polynomial systems of equations and explain how to construct starting systems to solve for all solutions.

The predictor-correcter method is widely used to solve nonlinear differential equations. Consider $F(X(t), t) \in \mathbb{F}^{k+1}$ where t represents time and $F(X(t)) = 0$ is a system of

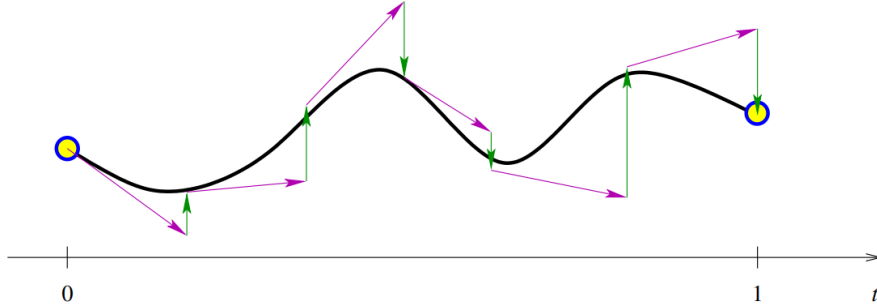


Figure 4.3: Predictor-Corrector Method for Solving Homotopy Continuation Problems [23]

equations, the idea is that from a known location $X(t_0)$, a predictor is used to estimate a solution $\bar{X} = X(t_0 + \Delta t)$ after a certain timestep Δt , and a corrector is used to refine this estimate \bar{X} to get an approximation closer to the true solution to $F(X(t_0 + \Delta))$.

The goal is to have the predictor return a good approximation that is close to the path, so that the corrector doesn't have to correct too much. In practice, Newton's method is often the choice for the corrector because it pairs well with the predictor supplying a close initial value and offers quadratic convergence.

Example. Suppose we have two univariate polynomials, $f_1(x) = x^2 - 5x + 6$ and $f_2(y) = y^3 - 3y^2 - 4y + 12$, and we want to estimate their solutions in the reals. An easier set of equations to solve is $g_1(x) = x^2 - 1$ and $g_2(y) = y^3 - 1$, where $x = 1$ and $y = 1$ are solutions. We can define $F(X) = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$, $G(X) = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}$, $X = \begin{bmatrix} x \\ y \end{bmatrix} \in \mathbb{R}^2$ and construct the homotopy

$$H : (\mathbb{R}^2, [0, 1]) \rightarrow \mathbb{R}^2$$

$$H(X(t), t) = (1 - t)G(X) + tF(X)$$

Then we can start at $H((1, 1)^T, 0) = G((1, 1)^T)$ and step along H with time-step Δt until we get a solution to $H(X, 1) = F(X) = 0$.

The homotopy may provide a continuous path from a solution to $G = 0$ to a solution to

$F = 0$. A set back is sometimes that a path may have a jacobian that is singular, however this can be overcome by the γ trick. The γ trick involves adding in a random scalar $\gamma \in \mathbb{C}$:

$$H(X, t) = (1 - t)G(X) + \gamma tF(X) = 0$$

this changes the path enough that all “bad” points are avoided during the path traversal. Notice in this example that the jacobian of $H((1, 1)^T, 0) = G((1, 1)^T)$ is $\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}$ which is nonsingular as the determinant is 6. This starting system was specially chosen to ensure that all starting solutions are regular, and if the paths are regular for $t \in [0, 1)$, all target solutions are reached.

4.3.1 Constructing a Start System

Often it is of interest to know all solutions to a system F . The starting system G of the homotopy continuation method is a critical component of the homotopy continuation method because it supplies a full set of initial solutions. For the method to be globally convergent, that is to find all solutions to the target system F , the number of isolated solutions of G must be equal to or greater than the number of solutions of F . If we pick a starting system that has too few solutions, then there will be some solutions to F that are unreachable.

Assuming that we are in an algebraically closed field², Bézout’s theorem provides a way to ensure all the solutions in the target system F can be reached. This starting system is sometimes referred to as a total-degree starting system.

Theorem 4.3.1 (Bézout). Let $F : \mathbb{F}^k \rightarrow \mathbb{F}^k$ be a system polynomials $f_1, \dots, f_k : \mathbb{F}^k \rightarrow \mathbb{F}$ where \mathbb{F} is an algebraically closed field. Let d_1, \dots, d_k denote their respective degrees. If $\{X \in \mathbb{F}^k : F(X) = 0\}$ is finite, then it has size at most $\prod_{i \in [k]} d_i$.

Bézout’s Theorem is a powerful result in algebraic geometry. Here we use it as an upper bound on the number of isolated solutions in a square system to construct a reasonable

²For example \mathbb{C} is good, but \mathbb{R} is not algebraically closed.

starting system. In the example given earlier for $\deg(f_1) = 2, \deg(f_2) = 3$, Bézout's theorem says there are at most 6 solutions. We can traverse each path from solutions to $x^2 - 1, y^3 - 1$ to attempt to find all of the isolated solutions of f_1, f_2 .

Building in HGates.m2

We can define a predictor-corrector method with RK4 and Newton's method to solve homotopy continuation problems in `HGates.m2`. First, we will start with the example homotopy defined above:

$$H((x, y), t) = (1 - t)(x^2 - 1, y^3 - 1)^T + \gamma t(x^2 - 5x + 6, y^3 - 3y^2 - 4y + 12)^T$$

We define the homotopy in `HGates.m2` as a map from x, y, t to H and then iterate over time steps of size 0.1 to move from $t = 0$ to $t = 1$ applying RK4 and Newton's method to follow the path.

```
R = RR_53
declareVariable \ {x, y, t}

-- 1. set up problem
f_1 = x*x - (fiveHGate*x) + sixHGate -- roots: 2, 3

-- roots: 2, -2, 3
f_2 = y*y*y - (threeHGate*y*y) - (fourHGate*y) + twelveHGate
g_1 = x*x - oneHGate
g_2 = y*y*y - oneHGate

-- 2. define homotopy
X = hMatrixGate({x, y}, 2, 1)
```

```

n = length X
Xlist = X.Elements;
F = hMatrixGate({f_1, f_2}, 2, 1)
G = hMatrixGate({g_1, g_2}, 2, 1)
H = ((oneHGate - t)*G) + ((inputHGate 0.528)*t*F)
M = hMap({t, X}, {H})

-- 3. traverse homotopy from t = 0 to t = 1
X0literal = {1., 1.}
t0literal = 0.
d = 0.1 -- time step
time while (t0literal < 1. - d) do (
  t1literal = t0literal + d;
  predlist = toList (t0literal, t1literal, X0literal);

  << "Iteration: " << t1literal << endl;
  -- 1. predict
  X1literal = predictorRK4(M, predlist);

  -- 2. correct
  Msinglevar = subMap(t, inputHGate t1literal, M);
  X1literal = newtonsMethod(Msinglevar, X1literal);

  -- 3. update values for next iteration
  t0literal = t1literal;
  X0literal = X1literal;
)

```

This produces a solution of $(x, y) = (2, -2)$ which evaluated over F is $5.76e-8$.

We take this approach and generalize it to the method `predictorCorrector`, based on RK4 and Newton's method, which in takes an `HMap` of F , an `HMap` of G , a list of a known solution to G and a timestep, and returns a list approximating a solution to F . The `predictorCorrector` method is flexible in taking both the maps for F and G , which allows the user more control over choosing different starting systems. For example, if the user is solving something over than a polynomial system of equations, then Bézout's Theorem does not apply for obtaining a reasonable starting system, or if the user is after all solutions to a polynomial system of equations and has a better bound, then `predictorCorrector` supports that.

The method is written as follows:

```
predictorCorrector(HMap, HMap, List, RR) :=
  (Fmap, Gmap, Gsol, d) -> (
    G := Gmap.OutputGates#0;
    F := Fmap.OutputGates#0;
    X := Fmap.InputGates#0;

    -- 1. Define homotopy HMap
    H = ((oneHGate - t)*G) + ((inputHGate 0.5)*t*F);
    M = hMap({t, X}, {H});

    -- 2. traverse homotopy from t = 0 to t = 1
    X0literal = Gsol;
    t0literal = 0.;

    while (t0literal < 1. - d) do (
      t1literal = t0literal + d;
```

```

    predlist = toList (t0literal, t1literal, X0literal);

    -- 1. predict
    X1literal = predictorRK4(M, predlist);

    -- 2. correct
    Msinglevar = subMap(t, inputHGate t1literal, M);
    X1literal = newtonsMethod(Msinglevar, X1literal);

    -- 3. update values for next iteration
    t0literal = t1literal;
    X0literal = X1literal;
  );
  X1literal
)

```

The homotopy continuation defined and solved above can now be succinctly written as:

```

R = RR_53
declareVariable \ {x, y, t}

-- 1. Define HMap of F
f_1 = x*x - (fiveHGate*x) + sixHGate -- roots: 2, 3

-- roots: 2, -2, 3
f_2 = y*y*y - (threeHGate*y*y) - (fourHGate*y) + twelveHGate
X = hMatrixGate({x,y}, 2, 1)
n = length X
Xlist = X.Elements;

```

```

F = hMatrixGate({f_1, f_2}, 2, 1)
MF = hMap({X}, {F})

-- 2. Define HMap of G, List solution
g_1 = x*x - oneHGate
g_2 = y*y*y - oneHGate
G = hMatrixGate({g_1, g_2}, 2, 1)
MG = hMap({X}, {G})
Gsol = {1., 1.}
d= 0.1

-- 3. Return solution to F
Xlliteral = predictorCorrector(MF, MG, Gsol, d)

```

Remark. This is a good starting method and algorithms with higher efficacy will extend beyond constructing a predictor-corrector with RK4 and Newton's method, involving modifications such as taking dynamic time steps, Padé approximants, and certifications.

4.4 Predictor Gates

This section provides a meta look into gates and explores creative applications of methods applied to themselves in the framework. For example, we can backtrack along paths by applying Newton's operator to itself and to predictor gates based on a desired output. The main motivator of the gate form is that it allows for the composition of SLPs, which can then be used for hardware acceleration. In this section we will introduce the tangent predictor gate, the trapezoid predictor gate, and the RK4 gate and give examples of Newton's Method and the RK4 method can be used to backtrack and forward track and compare their results.

The tangent predictor method works to estimate $x(t_1)$ by adding to x_0 the slope of f at

(t_0, x_0) multiplied by the change in t_0 and t_0 :

$$x_1 = x_0 + c(t_0, x_0) * (t_1 - t_0)$$

The corresponding Macaulay2 implementation is straight-forward, intaking an `HMap` g and a list I of initial variable names of the form $I = \{t_0, t_1, X_0\}$ where t_0, t_1 correspond to the initial and subsequent times and X_0 represents the initial value of X :

```

predictorTangHMatrixGate(HMap, List) := (g, I) -> (
  -- unpack variable names
  t0 := I#0;
  t1 := I#1;
  X0 := I#2;
  t := g.InputGates#0;
  X := g.InputGates#1;
  F := g.OutputGates#0;

  -- substitute values
  n := length X0;
  c1 := c(t, X, F);
  c2 := subGate (t, t0, c1);
  (0..n-1) / (i -> c2 =
    subGate((X.Elements)#i, (X0.Elements)#i, c2));

  X0 + c2*(t1 - t0)
)

```

We consider the SLP for the predictor tangent method with an example.

Example. Consider the polynomials $f_1(x, y, t) = x^2t + t$, $f_2(x, y, t) = y^3t - 2t$. The SLP

for predictorTangHMatrixGate to predict the next timestep for this function given initial conditions is

$$C0 = -1$$

$$C1 = 0$$

$$C2 = 1$$

$$C3 = 2$$

$$I0 = x[0]$$

$$I1 = x[1]$$

$$I2 = x[2]$$

$$I3 = x[3]$$

$$R0 = \text{matrix}\{I0, I1\} \ (2, 1)$$

$$R1 = C0 * I3$$

$$R2 = I2 + R1$$

$$R3 = I0 + I0$$

$$R4 = I3 * R3$$

$$R5 = I1 * I1$$

$$R6 = I1 + I1$$

$$R7 = I1 * R6$$

$$R8 = R5 + R7$$

$$R9 = I3 * R8$$

$$R10 = \text{matrix}\{R4, C1, C1, R9\} \ (2, 2)$$

$$R11 = I0 * I0$$

$$R12 = R11 + C2$$

$$R13 = R5 * I1$$

$$R14 = C0 * C3$$

$$R15 = R13 + R14$$

$$R16 = \text{matrix}\{R12, R15\} \ (2, 1)$$

```

R17 = solve{R10, R16}
R18 = scalarMatrixProduct(R2, R17)
R19 = matrixSum(R0, R18)
OUTPUT: R19 = matrixSum(R0, R18)

```

Similar to Newton's method, the first set of lines from R0 to R16 are based on f_1, f_2 and consist of arithmetic gates and matrix instantiation gates. The following lines are the more computationally expensive gates and will be fixed for any call of the predictor gate at one solve gate, one scalar matrix product gate, and one matrix sum gate.

The second method, called the *trapezoid predictor*, builds on the tangent predictor:

$$x_1 = x_0 + \frac{c(t_0, x_0) + c(t_0 + (t_1 - t_0), x_0 + c(t_0, x_0) * (t_1 - t_0))}{2} * (t_1 - t_0)$$

where the second call of c takes in the output of the tangent predictor. The code implementation is:

```

predictorTrapHMatrixGate (HMap, List) := (g, I) -> (
  -- unpack variable names
  t0 := I#0;
  t1 := I#1;
  X0 := I#2;
  t := g.InputGates#0;
  X := g.InputGates#1;
  F := g.OutputGates#0;

  -- substitute values
  n := length X0;
  c1 := c(t, X, F);
  c2 := subGate (t, t0, c1);

```

```

(0..n-1) / (i -> c2 =
    subGate((X.Elements)#i, (X0.Elements)#i, c2));

tDelta := t1 - t0;
cfirst := c2;
h1 := scalarProductHMatrixGate(tDelta, cfirst);
-- Xtang named after X value from tangent predictor
-- X_0 + c(X_0, t_0)*tDelta
Xtang := sumHMatrixGate(X0, h1);

-- substitute t, X with t_0 + tDelta = t_1, Xtang
c4 := subGate (t, t1, c1);
(0..n-1) / (i -> c4 =
    subGate((X.Elements)#i, elementHGate(Xtang, i), c4));
csecond := c4;

h2 := (inputHGate 0.5) * t1; -- t1/2
c5 := sumHMatrixGate(cfirst, csecond);
h3 := scalarProductHMatrixGate(h2, c5);
sumHMatrixGate(X0, h3)
)

```

We consider the SLP for the trapezoid method with the same example as before.

Example. Let $f_1(x, y, t) = x^2t + t$, $f_2(x, y, t) = y^3 - 2t$ as earlier. Then the SLP for one application of `predictorTrapHMatrixGate` is:

```

C0 = .5
...
C4 = 2

```

```

I0 = x[0]
I1 = x[1]
I2 = x[2]
I3 = x[3]
R1 = C0*I2
...
R9 = matrix{R3, C1, C1, R8} (2, 2)
...
R15 = matrix{R11, R14} (2, 1)
R16 = solve{R9, R15}
...
R20 = scalarMatrixProduct (R19, R16)
R21 = matrixSum(R17, R20)
...
R37 = solve{R31, R36}
R38 = matrixSum(R16, R37)
R39 = scalarMatrixProduct (R1, R38)
R40 = matrixSum(R0, R39)
OUTPUT: R40 = matrixSum(R0, R39)

```

where the “...” represents lines consisting of arithmetic or matrix instantiation gates. The key point to note with the complexity is that the trapezoid predictor method requires two solve gate calls and six matrix operation gates compared with the one solve gate and two matrix operation gates for the tangent predictor method.

The RK4 method can be written as a gate as follows:

```

predictorRK4HMatrixGate (HMap, List) := (g, I) -> (
  -- unpack variable names
  t0 := I#0;

```

```

t1 := I#1;
X0 := I#2;
t := g.InputGates#0;
X := g.InputGates#1;
F := g.OutputGates#0;

-- substitute values
n := length X0;
c1 := c(t, X, F);
d := t1 - t0;

-- compute R_1
c11 := subGate (t, t0, c1);
(0..n-1) / (i -> c11
    = subGate((X.Elements)#i, (X0.Elements)#i, c11));
R1 := c11*d;

-- compute R_2
tmid := t0 + ((inputHGate 0.5)*d);
c21 := subGate (t, tmid, c1);
X0plusR1 := X0 + ((inputHGate 0.5)*R1);
(0..n-1) / (i -> c21
    = subGate((X.Elements)#i,
        elementHGate(X0plusR1, i), c21));
R2 := c21*d;

-- compute R_3

```

```

c31 := subGate (t, tmid, c1);
X0plusR2 := X0 + ((inputHGate 0.5)*R2);
(0..n-1) / (i -> c31
    = subGate((X.Elements)#i,
        elementHGate(X0plusR2, i), c31));
R3 := c31*d;

-- compute R_4
tmore := t0 + d;
c41 := subGate (t, tmore, c1);
X0plusR3 := X0 + R3;
(0..n-1) / (i -> c41
    = subGate((X.Elements)#i,
        elementHGate(X0plusR3, i), c41));
R4 := c41*d;

X0 + ((inputHGate (1/6)) * (R1 + (twoHGate * R2)
    + (twoHGate * R3) + R4))
)

```

We consider the SLP for the RK4 method with the same example as before.

Example. Let $f_1(x, y, t) = x^2t + t$, $f_2(x, y, t) = y^3 - 2t$ as earlier. Then the SLP for one application of `predictorRK4HMatrixGate` is:

```

C1 = -1
...
I0 = x[0]
...
R1 = C1*I3

```

```

...
R17 = solve{R10, R16}
R18 = scalarMatrixProduct (R2, R17)
...
R22 = scalarMatrixProduct (C5, R18)
R23 = matrixSum(R21, R22)
...
R39 = solve{R33, R38}
R40 = scalarMatrixProduct (R2, R39)
R41 = scalarMatrixProduct (C4, R40)
R42 = matrixSum(R18, R41)
R44 = scalarMatrixProduct (C5, R40)
R45 = matrixSum(R43, R44)
...
R61 = solve{R55, R60}
R62 = scalarMatrixProduct (R2, R61)
R63 = scalarMatrixProduct (C4, R62)
R64 = matrixSum(R42, R63)
...
R67 = matrixSum(R66, R62)
...
R83 = solve{R77, R82}
R84 = scalarMatrixProduct (R2, R83)
R85 = matrixSum(R64, R84)
R86 = scalarMatrixProduct (C0, R85)
R87 = matrixSum(R0, R86)
OUTPUT: R87 = matrixSum(R0, R86)

```

In addition to the earlier notation, the expression “ . . . ” also represents lines with element gates which access one element of a matrix. The complexity from this is assumed to be on par with the arithmetic gates and the matrix instantiation gates. One application of RK4 requires four solve gates and sixteen matrix operation gates, including matrix products, which were not present in the tangent or the trapezoid predictor methods. Clearly the complexity for this is significantly more costly.

Let us revisit the 2-square polynomial system of equations to see how we can backtrack with Newton’s method from a given solution. Suppose that we know $(0.1, 0.1)^T$ at time t_1 is a good point for Newton’s method to converge or is a good point for RK4 to predict from and we want to find an initial point to get there.

Recall our two target univariate polynomials, $f_1(x_1) = x_1^2 - 5x_1 + 6$ and $f_2(x_2) = x_2^3 - 3x_2^2 - 4x_2 + 12$, and the starting equations $g_1(x_1) = x_1^2 - 1$ and $g_2(x_2) = x_2^3 - 1$. We defined $F(X) = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$, $G(X) = \begin{bmatrix} g_1 \\ g_2 \end{bmatrix}$, $X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2$ and constructed the homotopy

$$H : (\mathbb{R}^2, [0, 1]) \rightarrow \mathbb{R}^2, H(X(t), t) = (1 - t)G(X) + tF(X) \quad (4.5)$$

For what value X does $H(X, 0.2) \approx \begin{pmatrix} 0.1 & 0.1 \end{pmatrix}^T$ with a difference of 0.01 under the 2-norm?

Answer. We initialize the problem in `HGate.m2` by defining $F(X)$, $G(X)$, $H(X, t)$ and Newton’s operator for $H(X, t)$ and set the difference between X_k and X_{k+1} to be 0.1:

```
declareVariable \ {x, y, t}
f_1 = x*x - (fiveHGate*x) + sixHGate -- roots: 2, 3

-- roots: 2, -2, 3
f_2 = y*y*y - (threeHGate*y*y) - (fourHGate*y) + twelveHGate
g_1 = x*x - oneHGate
```

```

g_2 = y*y*y - oneHGate

X = hMatrixGate({x,y}, 2, 1)
F = hMatrixGate({f_1, f_2}, 2, 1)
G = hMatrixGate({g_1, g_2}, 2, 1)
H = ((oneHGate - t)*G) + ((inputHGate 0.528)*t*F)

Yt = hMatrixGate({inputHGate 0.1, inputHGate 0.1}, 2, 1)
t0 = inputHGate 0.2
Ht0 = subGate(t, t0, H)
Hsing = Ht0 - Yt
M_1 = hMap({X}, {Hsing}) -- M for Map
M_2 = newtonsOp(M_1)

```

Notice in the definition for H we multiply the term tF by 0.528. Initially, we ran the program with $H(X, t) = (1 - t)G + tF$ and an initial value of $\begin{pmatrix} 1.1 & 1.1 \end{pmatrix}^T$ which is pretty close to the zero at $t = 0$, however Newton's method did not terminate under the given conditions. Using Newton's method can fail if the function's "steepness" changes too quickly. Since the method relies on using the slope at each point to find the next approximation, if the slope is too extreme, the next guess can overshoot the solution and fail to converge, which is what happened in this case. The change from $G(X)$ to $F(X)$ was too rapid and we never reached a slope that resulted in a converging sequence of X_k .

To account for this, we use the γ trick, where $\gamma \in \mathbb{C}$ is a scaling factor that can dampen the contribution from $F(X)$ to avoid singularities [23]. The modified homotopy is

$$H(X, t) = (1 - t)G(X) + \gamma tF(X)$$

We found that a value of $\gamma = 0.528$ results in a solution and fixed it for consistency.

In practice and for the rest of the paper, γ is randomly generated from \mathbb{C} . The rest of the program is as follows:

```

R = CC_53
X0 = {1.1_R, 1.1_R}
L = inputValueTable {x => X0#0, y => X0#1}
-- confirm det jac Ht0' (X0) \neq 0
dHsing = jacobian (X, Hsing)
detdHsing = detHGate dHsing
assert((specialize(detdHsing, L))#0 != 0)

X1 = specialize(M_2, L)

-- iterate over X_k until we find one satisfying
-- ||X_k - X_{k-1}||_2 < 0.1
track = 0 -- track iterations
xerror = sqrt fold(plus, (X1 - X0)/(i -> i*i));
while (xerror >= 0.1) do (
  X0 = X1;
  L = inputValueTable {x => X0#0, y => X0#1};
  assert((specialize(detdHsing, L))#0 != 0);
  X1 = specialize(M_2, L);
  xerror = sqrt fold(plus, (X1 - X0)/(i -> i*i));
  track = track + 1
)

```

The result is $\left(0.907274, -0.825602\right)^T$, which is found after 12 iterations of Newton's

operator. The evaluation of H at $t = 0.2$ for this value is $\left(0.1, 0.0903743\right)$ with a forward error of 0.00962569 under the 2-norm.

□

We have found a value such that H evaluates to $(0.1, 0.1)^T$ at $t_1 = 0.2$. We can apply this method again to deduce a starting point such that H evaluates to $(0.907274, -0.825602)^T$, and so on with Newton's method.

We can use Newton's method to obtain starting points for predictors for a given time t_1 . For example, consider again

$$f_1(x, y, t) = (x^2 + 5x - 6)t$$

$$f_2(x, y, t) = (y^3 + 2y - 1)t$$

and let $t_0 = 0.2$, $t_1 = 0.3$ and the target solution $(x, y) = (1., 5.)$. Then we use Newton's method to recover starting values for each of the tangent predictor, trapezoid predictor, and RK4 predictor methods. Each call of Newton's method converges to a solution, with starting values being $(0.33, 1.81)$, $(0.60, 3.07)$, $(0.33, 2.08)$ for the tangent predictor, trapezoid predictor, and RK4 predictor respectively. Interestingly, the simple tangent predictor behaves returns a result similar to the more complex RK4 predictor.

CHAPTER 5

KEPLERIAN ORBITS

5.1 Background

Conics have been around for well over a millennium, with the first definition of a conic section is attributed to the Greek mathematician Menaechmus circa 320 BCE [25]. It wasn't until 1604 that Johannes Kelper, for which the Keplerian orbit is named after, defined foci [26]. This section will introduce a class of Keplerian orbits problems. First we will introduce the definition of a conic in both general and standard form and then we will briefly discuss the foci.

Consider the xyz -coordinate system. We can make a double cone by taking a line on the xz plane through the origin and rotating it about the z -axis. Any conic equation can be obtained by slicing this cone with a plane and taking the intersection, as shown in Figure Figure 5.1. If the plane is perpendicular to the xy plane and not through the origin, then the equation is a circle, and if we imagine gradually rotating it at an angle counter clockwise, we can get an ellipse, a parabola, and a hyperbola. If the plane intersects the origin, then the conic is called degenerate and may look like a point or a line.

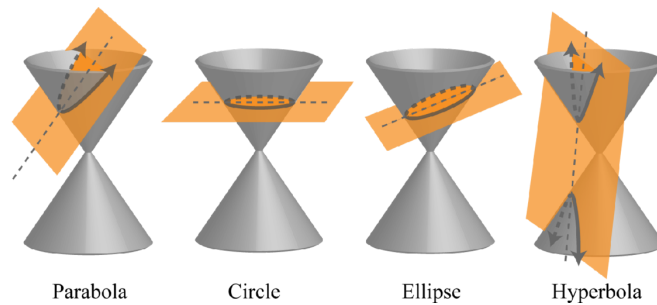


Figure 5.1: Different Types of Conic Equations Obtained from Slicing a Right Circular Cone with a Plane[27]

The general form of a conic is:

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$$

where A, B, C, D, E, F are scalars. Any conic can be expressed in this form, however in the case where the conic is not rotated, it is also common to see the standard equations forms for different types of conics. For example, the standard form of an ellipse centered at (h, k) with major and minor axis a, b depending on $a \leq b$ or $a > b$:

$$\frac{(x - h)^2}{a^2} + \frac{(y - k)^2}{b^2} = 1$$

when $a = b$, this is also called a circle. The other two types of conics are the hyperbola centered at (h, k) :

$$\frac{(x - h)^2}{a^2} - \frac{(y - k)^2}{b^2} = 1$$

and the vertical and horizontal parabolas:

$$y = ax^2 + bx + c, \quad x = ay^2 + by + c$$

The foci of a conic section are fixed points used to define the curve, with their location determining its specific shape and properties. For an ellipse, the sum of the distances from any point on the curve to the two foci is constant. For a hyperbola, the difference in the distances from any point on the curve to the two foci is constant. Parabolas have one focus and are defined by the distance to the focus and a directrix line, while a circle's focus is its center [28]. These are illustrated in Figure 5.2 in \mathbb{R} .

A Keplerian orbit is defined over a two-body problem in space where one relatively small object orbits another relatively large object, and the assumptions are that only the force from gravity is taken into account such that the orbit of the small object is a conic. In two or three dimensional space, the large object is situated at the origin and is also one of

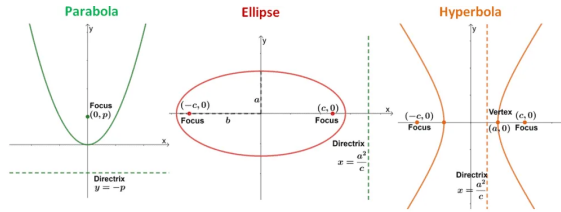


Figure 5.2: Locations of Foci for a Parabola, Ellipse, and Hyperbola [29]

the foci of the small object's conic [30].

5.2 Problem Set up

Suppose that we have two points and three partially known points:

$$(9, 0), (-1, 0), (4x - 8, -\frac{9}{5}), (4 - \frac{5\sqrt{5}}{3}, \frac{5}{2}y - \frac{1}{2}), (2x, -3y)$$

where $x, y \in \mathbb{R}$. We want to know, can we find values for x, y such that there is a conic passing through all points which has a focus at the origin, and if so, what are they?

We can find such points, and interestingly, to do so is a homotopy continuation problem in disguise. It may not be immediately apparent that we have a square system of equations since the general form equation of a conic

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0$$

has six coefficients and we have introduced two indeterminants x, y into the set of points, giving us eight variables and five equations¹.

We begin with the result showing that only three known points are needed to determine the coefficients for a conic with a focus through the origin.

Theorem 5.2.1. Three points in \mathbb{R}^2 determine a conic with a focus at the origin.

¹Each equation is obtained by evaluating the conic equation over one of the given points. For example one equation is $4Ax^2 - 6Bxy + 9Cy^2 + 2Dx - 3Ey + F = 0$.

Proof. A conic is represented by the following polynomial over two variables x, y

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F = 0 \quad (5.1)$$

with coefficients in a field, in this case $A, B, C, D, E, F \in \mathbb{R}$. The coefficients can be scaled, so there are only five degrees of freedom, and the equation for the conic, $Q(x, y)$ or Q for shorthand, is uniquely determined by $A, B, C, D, E \in \mathbb{R}^2$ when $F = 1$. Notice that since we have the restriction that a focus must be at the origin, F cannot be zero, there are five degrees of freedom and F is nonzero.

Let our three given points be of the form (x_i, y_i) for $i \in [3]$, and we have the following three equations

$$f_1 = Ax_1^2 + Bx_1y_1 + Cy_1^2 + Dx_1 + Ey_1 + 1$$

$$f_2 = Ax_2^2 + Bx_2y_2 + Cy_2^2 + Dx_2 + Ey_2 + 1$$

$$f_3 = Ax_3^2 + Bx_3y_3 + Cy_3^2 + Dx_3 + Ey_3 + 1$$

so the five unknown coefficients of the conic A, B, C, D, E .

We will prove the theorem by defining two more equations in terms of A, B, C, D, E which restrict the resulting conic to contain a focus at the origin, based on work by [31]. These five equations are sufficient to solve for the five unknown coefficients of the desired conic.

First, we denote \mathbb{P}^2 as the 2-dimensional projective space consisting of all 1-dimensional subspaces of \mathbb{R}^3 . Then we can represent a conic with Cartesian coordinates $(x, y) \in \mathbb{R}^2$ in \mathbb{P}^2 with homogenous coordinates. Homogenous coordinates extend Cartesian coordinates

²We will only consider conics with coefficients in \mathbb{R} , although they can be generalized to \mathbb{C} .

by adding a non-zero scalar. Formally, a point $(x, y) \in R^2$ can be transitioned to \vec{x} , where

$$\vec{x} \propto \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

This can be visualized as taking the conic in Cartesian coordinates and projecting it directly onto the xy plane at $z = 1$ so that it is an exact copy. Then any point on the conic can be uniquely represented by the line passing through it and the origin. For example, the vector $\begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix}^T$, for $z_0 \neq 0$, recovers the (x, y) coordinate $(\frac{x_0}{z_0}, \frac{y_0}{z_0})$ in the original Cartesian coordinate conic. This representation allows for easier transformation computations such as rotation and translation. We can rewrite the equation for Q in Equation 5.1 as

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} A & B/2 & D/2 \\ B/2 & C & E/2 \\ D/2 & E/2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = Q(x, y) = 0$$

which is proportionate to

$$\vec{x}^T \mathbf{C} \vec{x} = 0 \tag{5.2}$$

where

$$\mathbf{C}_Q \propto \begin{bmatrix} A & B/2 & D/2 \\ B/2 & C & E/2 \\ D/2 & E/2 & 1 \end{bmatrix}$$

We will refer to the coordinates of $\vec{x} = \begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix}^T \in \mathbb{P}^2$ as (x_0, y_0, z_0) and will also use the word *point* when referring to the coordinates in \mathbb{P}^2 . A line l passing through \vec{x} at its coordinates can be represented by $\begin{bmatrix} a & b & c \end{bmatrix}^T$ where $\vec{x}^T l = 0$. A point \vec{x} is conjugate to

another point \vec{y} if the following relation holds:

$$\vec{x}^T \mathbf{C}_Q \vec{y} = 0 \quad (5.3)$$

If the set of all \vec{y} satisfying Equation Equation 5.3 holds and is a line, it is called the *polar line*. In other words, there is an l associated with \vec{x} such that $\vec{y}^T l = 0$ for all y satisfying Equation Equation 5.3. The line l is unique to \vec{v} with respect to the conic, and in the subsequent lines we will give a useful relation between l and \vec{x} . Recall that \mathbf{C}_Q is symmetric, so we can then write

$$\vec{y}^T \mathbf{C}_Q^T \vec{x} = 0$$

$$\vec{y}^T \mathbf{C}_Q \vec{x} = 0$$

$$\vec{y}^T l = 0$$

giving us the following relationships between a point and its polar line:

$$l \propto \mathbf{C}_Q \vec{x}$$

and

$$\vec{x} \propto \mathbf{C}_Q^{-1} l$$

Notice that Equation Equation 5.2 implies that if \vec{x} is on the conic, then it is also on its polar line. It is well-known that the polar line of a point on a conic is the line tangent to the conic at that point. We can use this to define an equation for the *conic envelope*, all the

lines tangent to the conic. Substituting $\vec{x} \propto \mathbf{C}_Q^{-1}l$ into Equation Equation 5.2:

$$(\mathbf{C}_Q^{-1}l)^T \mathbf{C}_Q (\mathbf{C}_Q^{-1}l) = 0$$

$$l^T (\mathbf{C}_Q^T)^{-1} l = 0$$

$$l^T \mathbf{C}_Q^{-1} l = 0$$

Observe that the adjugate or adjoint of \mathbf{C}_Q , denoted \mathbf{C}_Q^* , is proportionate to \mathbf{C}_Q^{-1} . This can be seen by

$$\mathbf{C}_Q \mathbf{C}_Q^* = (\det \mathbf{C}_Q) I \implies \mathbf{C}_Q^{-1} = \frac{1}{\det \mathbf{C}_Q} \mathbf{C}_Q^*$$

assuming that \mathbf{C}_Q is nonsingular. Then the conic envelope is obtained by

$$l^T \mathbf{C}_Q^* l = 0 \tag{5.4}$$

This form is convenient because it does not rely on the inverse of \mathbf{C}_Q and thus holds when \mathbf{C}_Q is singular and the conic is degenerate [31]. Notice that we can write \mathbf{C}_Q^* as

$$\mathbf{C}_Q^* \propto \begin{bmatrix} A^* & B^*/2 & D^*/2 \\ B^*/2 & C^* & E^*/2 \\ D^*/2 & E^*/2 & F^* \end{bmatrix}$$

for values $A^*, B^*, C^*, D^*, E^*, F^* \in \mathbb{R}$ in since \mathbf{C}_Q is symmetric.

The next two definitions will tie everything together. An *isotropic line* of a point \vec{x} is a line in the complex plane with slope $\pm i$ such that it passes through the coordinate \vec{x} and either of the circular points at infinity, also referred to as isotropic points [32]. Since we are translating from the Cartesian coordinate system, we will define the isotropic points I, J

by the coordinates

$$\vec{i} \propto \begin{bmatrix} 1 & i & 0 \end{bmatrix}^T$$

$$\vec{j} \propto \begin{bmatrix} 1 & -i & 0 \end{bmatrix}^T$$

respectively. A point is a *focus* of a conic if its isotropic lines are tangent to the conic. If a focus \vec{f} is at the origin, then we write $\vec{f} \propto \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$. The isotropic lines, l_I, l_J , can then be obtained by taking the cross-product of \vec{f} with \vec{i} and \vec{j} respectively:

$$l_I \propto \vec{f} \times \vec{i} \propto \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \times \begin{bmatrix} 1 & i & 0 \end{bmatrix}^T = \begin{bmatrix} -i & 1 & 0 \end{bmatrix}^T \quad (5.5)$$

$$l_J \propto \vec{f} \times \vec{j} \propto \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T \times \begin{bmatrix} 1 & -i & 0 \end{bmatrix}^T = \begin{bmatrix} i & 1 & 0 \end{bmatrix}^T \quad (5.6)$$

Since l_I, l_J are tangent to the conic, we may combine Equation 5.5 and Equation 5.6 with Equation 5.4:

$$l_I^T \mathbf{C}_Q^* l_I \propto \begin{bmatrix} -i & 1 & 0 \end{bmatrix} \begin{bmatrix} A^* & B^*/2 & D^*/2 \\ B^*/2 & C^* & E^*/2 \\ D^*/2 & E^*/2 & F^* \end{bmatrix} \begin{bmatrix} -i \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -i & 1 & 0 \end{bmatrix} \begin{bmatrix} -A^*i + B^*/2 \\ -B^*i/2 + C^* \\ -D^*i/2 + E^*/2 \end{bmatrix}$$

$$= -A^* - B^*i/2 - B^*i/2 + C^*$$

$$= (C^* - A^*) - B^*i = 0$$

$$l_J^T \mathbf{C}_Q^* l_J \propto \begin{bmatrix} i & 1 & 0 \end{bmatrix} \begin{bmatrix} A^* & B^*/2 & D^*/2 \\ B^*/2 & C^* & E^*/2 \\ D^*/2 & E^*/2 & F^* \end{bmatrix} \begin{bmatrix} i \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} i & 1 & 0 \end{bmatrix} \begin{bmatrix} A^*i + B^*/2 \\ B^*i/2 + C^* \\ D^*i/2 + E^*/2 \end{bmatrix}$$

$$= -A^* + B^*i/2 + B^*i/2 + C^*$$

$$= (C^* - A^*) + B^*i = 0$$

$$(C^* - A^*) - B^*i = (C^* - A^*) + B^*i \implies A^* = C^*, B^* = 0$$

Then

$$\mathbf{C}_Q^* \propto \begin{bmatrix} A^* & 0 & D^*/2 \\ 0 & A^* & E^*/2 \\ D^*/2 & E^*/2 & F^* \end{bmatrix}$$

and we can use the restrictions that the entries $\mathbf{C}_{Q(11)}^* = \mathbf{C}_{Q(22)}^*$ and $\mathbf{C}_{Q(12)}^* = \mathbf{C}_{Q(21)}^* = 0$ to write equations relating A, B, C, D, E from \mathbf{C}_Q . Let M_{ij} be the (i, j) -minor of \mathbf{C}_Q

$$\mathbf{C}_{Q(11)}^* = M_{11} = C - E^2/4$$

$$\mathbf{C}_{Q(22)}^* = M_{22} = A - D^2/4$$

$$\mathbf{C}_{Q(21)}^* = (-1)M_{12} = -B/2 + DE/4$$

Then we can write

$$f_4 = A - C - D^2/4 + E^2/4$$

$$f_5 = B/2 - DE/4$$

where solutions to $f_4, f_5 = 0$ will ensure that the conic has a focus at the origin.

Thus solutions to $f_1, f_2, f_3, f_4, f_5 = 0$ will be the coefficients for the conic Q passing through the given points $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ with a focus at the origin.

□

Now we have at least established a square system of equations. We will initialize the problem stated in the beginning of the subsection in `HGates.m2` and solve it using the predictor-corrector method. Let $Q(x, y) = Ax^2 + Bxy + Cy^2 + Dx + Ey + 1$ be the equation

for our conic, then the target system $F = (f_1, \dots, f_7)$ for $X = (A, B, C, D, E, x, y)^T$ is:

$$f_1(X) = Q(9, 0)$$

$$f_2(X) = Q(-1, -)$$

$$f_3(X) = Q(4x - 8, -\frac{9}{5})$$

$$f_4(X) = (4 - \frac{5\sqrt{5}}{3}, \frac{5}{2}y - \frac{1}{2})$$

$$f_5(X) = (2x, -3y)$$

$$f_6(X) = A - C - D^2/4 + E^2/4$$

$$f_7(X) = B/2 - DE/4$$

Observe that the degrees for $f_1, f_2, f_3, f_4, f_5, f_6$ and f_7 are $d_1 = 1, d_2 = 1, d_3 = 3, d_4 = 3, d_5 = 3, d_6 = 2$ and $d_7 = 2$ respectively. We choose our total-degree starting system $G = (g_1, \dots, g_7)$ such that g_i are univariate of the form $z^{d_i} - 1$, for $i \in [7]$ and $z \in \{A, B, C, D, E, x, y\}$:

$$g_1(A) = A - 1$$

$$g_2(B) = B - 1$$

$$g_3(C) = C^3 - 1$$

$$g_4(x) = x^3 - 1$$

$$g_5(y) = y^3 - 1$$

$$g_6(D) = D^2 - 1$$

$$g_7(E) = E^2 - 1$$

By Bézout's Theorem, we have at most $3^3 2^2 = 108$ solutions. To solve for one solution, we can take $X_0 = (1, 1, 1, 1, 1, 1, 1)$, where $\det \frac{\partial G}{\partial X}(X_0) \neq 0$. Later on we will use this starting system to solve for all solutions.

5.3 Building in HGates.m2

We can initialize our reformulated homotopy problem in HGates.m2 as:

```
R = RR_53
declareVariable \ {A, B, C, D, E, x, y, t}

-- 1. Given points
x1 = inputHGate 9
y1 = zeroHGate
x2 = minusOneHGate
y2 = zeroHGate
x3 = fourHGate * x - inputHGate 8
y3 = inputHGate (-9/5)
x4 = inputHGate (4 - sqrt(5) * 5/3)
y4 = y * inputHGate (5/2) - inputHGate 0.5
x5 = twoHGate * x
y5 = inputHGate (-3) * y

-- 2. For Predictor-Corrector Homotopy Continuation
f0 = method()
f0(HGate, HGate) := (X, Y) -> (
  A*X*X + B*X*Y + C*Y*Y + D*X + E*Y + oneHGate
)
f1 = f0(x1, y1)
f2 = f0(x2, y2)
f3 = f0(x3, y3)
f4 = f0(x4, y4)
```

```

f5 = f0(x5, y5)
f6 = A - C - (inputHGate 0.25 * D * D)
      + (inputHGate 0.25 * E * E)
f7 = (inputHGate 0.5 * B) - (inputHGate 0.25 * D * E)

X = hMatrixGate({A, B, C, D, E, x, y}, 7, 1)
n = length X
Xlist = X.Elements;
F = hMatrixGate({f1, f2, f3, f4, f5, f6, f7}, 7, 1)
MF = hMap({X}, {F})

-- starting system
g1 = A - oneHGate
g2 = B - oneHGate
g3 = C*C*C - oneHGate
g4 = x*x*x - oneHGate
g5 = y*y*y - oneHGate
g6 = D*D - oneHGate
g7 = E*E - oneHGate

G = hMatrixGate({g1, g2, g3, g4, g5, g6, g7}, 7, 1)
MG = hMap({X}, {G})
Gsol = {1, 1, 1, 1, 1, 1, 1} -- known solution of G
d = 0.01 -- time-step

```

It can be executed with `predictorCorrector` with inputs `MF`, `MG`, `Gsol`, `d`.

One might be interested in exploring compact systems. We can reduce this 7-square system into a 2-square system with `SolveHMatrixGate`. This system can be solved

with a direct application of Newton's method or by setting up a homotopy and traversing it with the predictor-corrector method from above. The key observation is that the conic coefficients A, B, C, D, E can be solved for by a linear system of equations if the five given points are known. Fix x, y and let $(x_1, y_1), \dots, (x_5, y_5)$ denote the five given points determined by x, y . Then we can solve:

$$\begin{bmatrix} x_1^2 & x_1 y_1 & y_1^2 & x_1 & y_1 \\ x_2^2 & x_2 y_2 & y_2^2 & x_2 & y_2 \\ x_3^2 & x_3 y_3 & y_3^2 & x_3 & y_3 \\ x_4^2 & x_4 y_4 & y_4^2 & x_4 & y_4 \\ x_5^2 & x_5 y_5 & y_5^2 & x_5 & y_5 \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \\ E \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

This is a function in terms of x, y . We can map the resulting A, B, C, D, E to a length two vector containing the focus-at-the-origin constraints to get the desired 2-square system.

In `HGates.m2` this is initialized for a direct application of Newton's method as:

```
R = RR_53
declareVariable \ {x, y}

-- 1. Given points
x1 = inputHGate 9
y1 = zeroHGate
x2 = minusOneHGate
y2 = zeroHGate
x3 = fourHGate * x - inputHGate 8
y3 = inputHGate (-9/5)
x4 = inputHGate (4 - sqrt(5) * 5/3)
y4 = y * inputHGate (5/2) - inputHGate 0.5
x5 = twoHGate * x
```

```

y5 = inputHGate (-3) * y

-- 2. For Newton's Method
AA = hMatrixGate({
  x1*x1, x1*y1, y1*y1, x1, y1,
  x2*x2, x2*y2, y2*y2, x2, y2,
  x3*x3, x3*y3, y3*y3, x3, y3,
  x4*x4, x4*y4, y4*y4, x4, y4,
  x5*x5, x5*y5, y5*y5, x5, y5},
  5, 5)
bb = hMatrixGate({minusOneHGate, minusOneHGate,
  minusOneHGate, minusOneHGate, minusOneHGate}, 5, 1)
Y = solveHMatrixGate(AA, bb)

(A, B, C, D, E) = apply(0..4, i->elementHGate(Y, i))
F = hMap({hMatrixGate({x, y}, 2, 1)}, {
  hMatrixGate({A - C - D*D * inputHGate 0.25 + E * E
    * inputHGate 0.25, B * inputHGate 0.5 - D * E
    * inputHGate 0.25}, 2, 1)})

```

The construction of the HMap for Newton's Method, enables us to define a homotopy for a 2-square system. The homotopy continuation set-up is more complex than the 7-square case because we are not dealing with polynomials anymore. We will take the following to consider the starting system and the best values d_1, d_2 to use for $x^{d_1} - 1$ and $y^{d_2} - 1$.

Theorem 5.3.1. Let $\mathbf{A} = (a_{ij})_{i,j \in [5]}$ be a 5×5 matrix with $a_{11}, \dots, a_{15}, a_{21}, \dots, a_{25}$ and $a_{33}, a_{35}, a_{41}, a_{44} \in \mathbb{R}$, $a_{32}, a_{33}, a_{35}, a_{41}, a_{42}, a_{43} = 0$, a_{31}, a_{51} quadratic in x^2 , a_{43}, a_{53} quadratic in y^2 , a_{52} quadratic in xy , a_{34}, a_{54} linear in x and a_{45}, a_{55} linear in y . The matrix

\mathbf{A} , with each entry's leading term divided by its leading coefficient is:

$$\begin{bmatrix} \alpha & \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ x^2 & 0 & 0 & x & 0 \\ 0 & 0 & y^2 & 0 & y \\ x^2 & xy & y^2 & x & y \end{bmatrix}$$

where $\alpha \in \{0, 1\}$.

Let $\mathbf{b} = (-1, -1, -1, -1, -1)^T$. Define $(A, B, C, D, E)^T = \mathbf{A}^{-1}\mathbf{b}$ and $f_1 = A - C - D^2/4 + E/4$, $f_2 = B/2 - DE/4$. Then f_1 and f_2 are rational functions in x, y of the form

$$f_1(x, y) = \frac{r_1(x, y)}{p_1(x, y)} = 0$$

$$f_2(x, y) = \frac{r_2(x, y)}{p_2(x, y)} = 0$$

where $r_1, r_2, p_1, p_2 \in \mathbb{R}[x, y]$ satisfy $\max(\deg p_1, \deg p_2) \leq 12$ and $\max(\deg r_1, \deg r_2) \leq 12$.

Proof. We may observe that each coefficient A, B, C, D, E is a rational function of x, y . Let \mathbf{A} denote the matrix `AA` from the code, \mathbf{b} the solution vector `bb`, and \mathbf{x} the vector of coefficients ordered lexicographically. By Cramer's Rule for $\mathbf{A}\mathbf{x} = \mathbf{b}$ in the case that there

is a unique solution in the system,

$$\begin{aligned} A &= \frac{\det \mathbf{A}_1}{\det \mathbf{A}} \\ B &= \frac{\det \mathbf{A}_2}{\det \mathbf{A}} \\ C &= \frac{\det \mathbf{A}_3}{\det \mathbf{A}} \\ D &= \frac{\det \mathbf{A}_4}{\det \mathbf{A}} \\ E &= \frac{\det \mathbf{A}_5}{\det \mathbf{A}} \end{aligned}$$

where \mathbf{A}_i is the matrix \mathbf{A} with the i th column replaced by \mathbf{b} .

Then f_1, f_2 are also rational functions of x, y , and we want to find p_1, p_2 to minimally represent the denominator. We can substitute the functions for A, B, C, D, E into f_1, f_2 :

$$\begin{aligned} f_1 &= A - C - D^2/4 + E^2/4 = \frac{\det \mathbf{A}_1}{\det \mathbf{A}} - \frac{\det \mathbf{A}_3}{\det \mathbf{A}} - \left(\frac{\det \mathbf{A}_4}{2 \det \mathbf{A}}\right)^2 + \left(\frac{\det \mathbf{A}_5}{2 \det \mathbf{A}}\right)^2 \\ f_2 &= B/2 - DE/4 = \left(\frac{\det \mathbf{A}_2}{2 \det \mathbf{A}}\right) - \left(\frac{\det \mathbf{A}_4 \det \mathbf{A}_5}{4(\det \mathbf{A})^2}\right) \end{aligned}$$

Then we choose the polynomials $p_1, p_2 = p^2 = (\det \mathbf{A})^2 \in \mathbb{R}[x, y]$ and the polynomials $q_j = \det \mathbf{A}_j \in \mathbb{R}[x, y]$ for $j \in [5]$ to write

$$\begin{aligned} f_1 &= \frac{r_1}{p_1} = \frac{pq_1 - pq_3 - q_4^2/4 + q_5^2/4}{p^2} \\ f_2 &= \frac{r_2}{p_2} = \frac{pq_2/2 - q_4q_5/4}{p^2} \end{aligned}$$

The order of p^2 is determined by considering the Leibniz formula with the degree matrix of \mathbf{A} , $\deg(\mathbf{A})$, where the (i, j) -entry of $\deg(\mathbf{A})$ corresponds to the total degree of the polynomial $\mathbf{A}_{(i,j)} \in \mathbb{R}[x, y]$, where $\mathbf{A}_{(i,j)}$ is the (i, j) -entry of \mathbf{A} , $i, j \in [5]$. The first two rows are all zero since $(x_1, y_1), (x_2, y_2)$ are scalars, and the next three contain nonzero

entries depending on x, y :

$$\deg \mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 2 & 2 & 2 & 1 & 1 \end{bmatrix}$$

The Leibniz formula, using the usual σ notation for a permutation in the symmetric group size n , S_n , and the signature function sgn is

$$\sum_{\sigma \in S_n} (\text{sgn}(\sigma) \prod_{i=1}^n \mathbf{A}_{(i, \sigma(i))})$$

The largest degree element in the summation is given by

$$\deg(\mathbf{A}_{(3,1)} \mathbf{A}_{(5,2)} \mathbf{A}_{(4,3)} \mathbf{A}_{(1,4)} \mathbf{A}_{(2,5)}) = 2 + 2 + 2 + 0 + 0 = 6$$

So $\deg(p^2) = 2 \deg p = 12$. The orders of the q_j can be computed similarly. We see that q_1, q_2, q_3 will have the same order and q_4, q_5 will have the same order by symmetry of their

corresponding degree matrices, so it is sufficient to look only at $\det \mathbf{A}_1, \det \mathbf{A}_4$

$$\deg \mathbf{A}_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 0 & 2 & 2 & 1 & 1 \end{bmatrix} \implies \max_{\sigma \in S_n} \left(\deg \prod_{i=1}^n \mathbf{A}_{1:(i, \sigma(i))} \right) = 2 + 2 + 1 = 5$$

$$\deg \mathbf{A}_4 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 1 \\ 2 & 2 & 2 & 0 & 1 \end{bmatrix} \implies \max_{\sigma \in S_n} \left(\deg \prod_{i=1}^n \mathbf{A}_{4:(i, \sigma(i))} \right) = 2 + 2 + 2 = 6$$

where we use the notation $\mathbf{A}_{j:(i, \sigma(i))}$ for the $(i, \sigma(i))$ -entry of \mathbf{A}_j . Thus $\deg q_1 = \deg q_2 = \deg q_3 = 5$ and $\deg q_4 = \deg q_5 = 6$, and we can compute the degrees of the numerators of f_1, f_2 :

$$\deg f_1 p^2 = \max(\deg p + \deg q_1, \deg p + \deg q_3, 2 \deg q_4, 2 \deg q_5) = 12$$

$$\deg f_2 p^2 = \max(\deg p + \deg q_2, \deg q_4 + \deg q_5) = \max(12, 12) = 12$$

□

So the total-degree starting system is

$$g_1(x) = x^{12} - 1$$

$$g_2(y) = y^{12} - 1$$

Since the 7×7 system has a starting system of 108, this means there are at least 36 extraneous solutions that have been added by this elimination model. In particular, these arise for $(x, y) \in \mathbb{V}(r_1(x, y), r_2(x, y)) \cap \mathbb{V}(p(x, y))$, or equivalently when (x, y) is in $\mathbb{V}(r_1(x, y), r_2(x, y), p(x, y))$.

5.4 Comparing Newton's Method and Predictor-Corrector Homotopy Continuation

For this section we will refer to Newton's Method as NM and the predictor-corrector method under `predictorCorrector` for the homotopy continuation set-up on $k \times k$ square systems as `PCHCk`.

We construct four examples of problems with at least two known solution over a circle, ellipse, parabola, and hyperbola, and compare three different methods for solving for one solution. We explain how to directly apply Newton's Method, set up a homotopy continuation problem over A, B, C, D, E, x, y , and set up a homotopy continuation problem for x, y .

We will begin with the problem given in section 5.2, which was fabricated by carefully selecting points on an ellipse. Here is it again for convenience:

$$(9, 0), (-1, 0), (4x - 8, -\frac{9}{5}), (4 - \frac{5\sqrt{5}}{3}, \frac{5}{2}y - \frac{1}{2}), (2x, -3y)$$

The target coefficients for the ellipse equation in general form are $-.111, 0, -.309, .889, 0$ and both $(x, y) = (2, 1)$ and $(x, y) = (4, -.6)$ satisfy the constraints. This was derived from the ellipse:

$$\frac{(x - 4)^2}{25} + \frac{y^2}{9} = 1$$

which is centered at $(4, 0)$ with foci $(0, 0)$ and $(8, 0)$.

We introduce three more problems with at least two known solutions for the circle, the parabola, and the hyperbola. The known conic equations written in standard form are as follows, where the parabola is written in standard form and then rotated by 45 degrees to

satisfy the focus at the origin condition:

$$\text{Circle: } x^2 + y^2 = 4$$

$$\text{Hyperbola: } \frac{(x - 3)^3}{4} - \frac{y}{5} = 1$$

$$\text{Parabola: } y^2 = -4(x - 1) - \textit{rotated 45 deg. about the origin}$$

The circle problem is designed to yield at least four possible answers, and the hyperbola and parabola problems are designed to yield at least two correct answers.

For Newton's method, we chose a random starting point in \mathbb{C}^2 and selected 112 trials based on the starting solution set from PCHC2 which operates over x, y . We set the termination conditions to a backward error of less than $1e - 6$ or 50 iterations and determined whether a solution was found based on a forward error of less than $1e - 6$ and the condition number of the jacobian being less than $1e3$.

In PCHC7, we take a timestep of 0.01 and stop Newton's method if the backward error is less than $1e - 6$ or reaches 25 iterations. For the final value $t = 1$, we let Newton's method run for up to 50 iterations. A solution is identified and recorded if its forward error is less than $1e - 6$ and the condition number of its jacobian is less than 1000. If the condition number is greater than or equal to 1000, we record the solution as being ill-conditioned. Similarly, in PCHC2, we follow the same set up as PCHC7.

We use the Macaulay2 package Numerical Algebraic Geometry (NAG4M2) as a ground truth to determine the total number of unique solutions for each problem. NAG4M2 uses homotopy continuation methods to solve for systems of polynomial equations. In our solution recovery, we use the function `solveSystem`, which returns the set of solutions to the problem with default settings to use predictor-corrector method with Newton's method and Runge-Kutta 4 [33]. The method `solveSystem` varies its timestep depending on the slope of the function to improve performance and accuracy, whereas our methods in PCHC7 and PCHC2 use a fixed timestep to align with SLP conversion.

Table 5.1: Average number of unique solutions found for NM with 112 random starting points, PCHC7 tracking 108 paths (2 trials, $\Delta = 0.01$), PCHC2 tracking 1 solution (1 trial, $\Delta = 0.05$), and NAG4M2 solveSystem for the 7-square system.

	Circle	Ellipse	Parabola	Hyperbola
NM	0	0	0	0
PCHC7	7.5	9.5	17.5	15
PCHC2	-	0	-	-
NAG4M2	8	12	28	32

The results for NM, PCHC7, PCHC2, and NAGM2 in identifying all unique solutions to the four conic problems are shown in Table 5.1. Since we are approximating solutions, for NM, PCHC7, and PCHC2 we grouped the solutions based on their difference under the 2-norm to give an estimate of the number of unique solutions that were discovered. In all cases, the number of unique solutions for NM, PCH7 and PCHC2 will be upper bounded by the number of solutions found by NAGM2. We remark that both the methods for NM and for PCHC2 performed poorly in accuracy. NM started with 112 random points in \mathbb{C}^2 and did not converge for a single one to produce a solution, over all runs. PCHC2 ran for an average of 50 minutes *per path* for a select few of the ellipse homotopies and failed to produce any solutions. Given its high computational cost and poor performance, we opted not to test PCHC2 over all 112 paths for the conic problems. Instead, we focused on PCHC7 to demonstrate a more efficient, SLP-compatible technique.

Two trials were conducted for each conic problem using PCHC7 to find solutions. The grouped results for each trial of the circle problem are shown in Table 5.2 where 52.5 solutions were identified of the 108 paths tracked with good condition number on the jacobian and grouped by a 2-norm difference of $1e - 4$. All four fabricated solutions were found for both trials, and the four additional solutions known to exist by NAG4M2 were found in Trial 1. Trial 2 only located 3 of these solutions. A representative from each group is selected and the coefficients are used to graphed the corresponding conic in, with the results for Trial 1 given in Figure 5.3. The target conic equation for the circle is given in red, where solutions containing coefficients matching the conic are hidden by the red curve. We

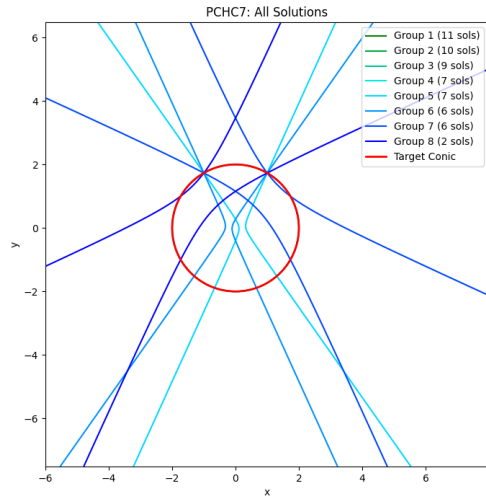


Figure 5.3: Corresponding Conic Equation of Grouped Solutions for the Circle Problem

can see that conics representing the other four groups, those not matching our fabricated solutions, are all hyperbolas.

Table 5.2: Approximate Solutions Grouped by 2-norm Difference Less Than $1e - 4$ for the Circle Problem, G_i reading “Group i ”

	G 1	G 2	G 3	G 4	G 5	G 6	G 7	G 8
Trial 1	11	10	9	7	7	6	6	2
Trial 2	11	9	8	6	5	5	3	

For the ellipse trials, neither was able to find all twelve solutions identified by NAG4M2, however compared with the circle trials, more paths were successfully tracked to a solution. The breakdown of the groupings is shown in Table 5.3. The average number of paths tracked to a solution was 108 for both with each finding one and four solutions respectively with a poor condition number on the jacobian. Given the number of solutions found from tracking each path with the overall low condition number, it is reasonable to conclude that there must have been some path jumping involved for both methods to miss at least two solutions. We graph the results from Trial one in Figure 5.4. In addition to the groups matching the target conic in red, we can see two more ellipses that are slightly rotated. The rest of the groups have conics that are hyperbolas.

Table 5.3: Approximate Solutions Grouped by 2-norm Difference Less Than $1e - 4$ for the Ellipse Problem, G_i reading “Group i ”

	G 1	G 2	G 3	G 4	G 5	G 6	G 7	G 8	G 9	G 10
Trial 1	44	13	10	9	8	7	6	5	5	1
Trial 2	40	14	11	10	10	9	9	3	2	

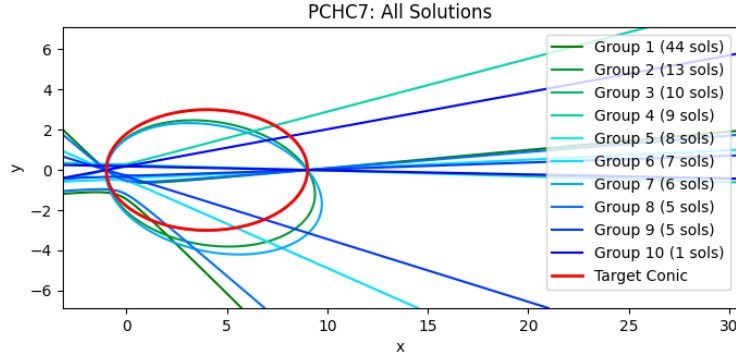


Figure 5.4: Corresponding Conic Equation of Grouped Solutions for the Ellipse Problem

For the parabola problem, in Trial 1 there were 84 solutions found with 12 solutions giving a poor jacobian condition number or about 14%. Of the 72 solutions with a well-conditioned jacobian, we found 18 groups. In Trial 2, there were 89 solutions found with 13 solutions giving a poor jacobian condition number or about 14.5%. We found 17 groups for this trial with a similar distribution to the first trial where most of the solutions were split between the first five groups and the rest of the groups had 1-3 solutions. The division of solutions for Trial 1 and Trial 2 are given in Table 5.4 for the first ten groups. The rest of the groups have between 1-2 solutions. Compare with the results from NAG4M2, we expected to find closer to 28 solutions, however we only recovered 62.5% of the total solutions with PCHC7. Since the solutions with poorly conditioned jacobians are still short of the missing solutions, 14% vs 37.5%, it is possible that there is path jumping occurring in addition to some of the solutions being singular. The results from Trial 1 are graphed in Figure 5.5. Most of the conics are clustered close to their focus at the origin, slightly shifted above and below the red target conic equation. It is noteworthy that many of them intersect at one point on the left of the graph. One equation in light green that looks distinct

Table 5.4: Approximate Solutions Grouped by 2-norm Difference Less Than $1e - 4$ for the Parabola Problem, G_i reading “Group i ”

	G 1	G 2	G 3	G 4	G 5	G 6	G 7	G 8	G 9	10
Trial 1	12	7	7	7	7	5	4	3	3	3
Trial 2	12	10	10	8	4	4	4	4	4	3

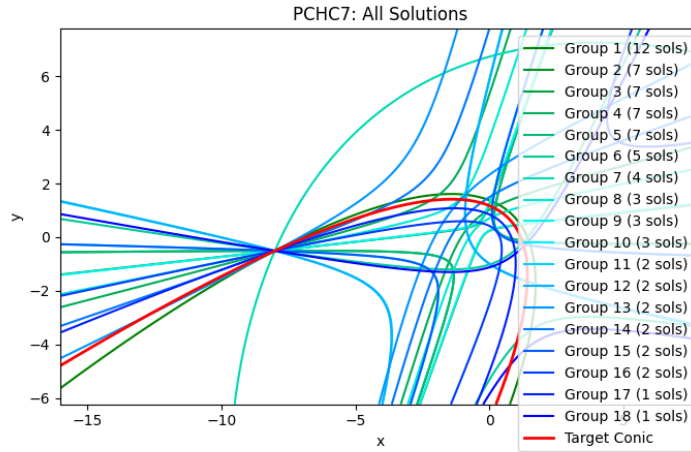


Figure 5.5: Corresponding Conic Equation of Grouped Solutions for the Parabola Problem

from the rest such that it doesn't look like a slight perturbation of another group's conic.

The results for the hyperbola problem are less numerically stable. All 108 paths were successfully tracked from the starting solutions such that the forward error was less than $1e - 6$, however 62 and 57 of the solutions respectively had poor condition numbers for the jacobian. That means that over half of the solutions found, 55%, have a poor condition number. In the first trial, of the 46 good solutions with condition number less than 1000, we identified 16 groups, with the group division given in Table 5.5. In the second trial, 51 solutions had a good condition number for the jacobian and produced 14 groups. For both trials, this about half of the expected number of 32 unique solutions identified with NAG4M2. Given both the number of high-condition number solutions and the results from NAG4M2, it is highly likely that the other half of the solutions are singular, since we discard the results with low forward error when the condition number of the jacobian is over 1000. The conic equation with each group is graphed in Figure 5.6 where the conics appear far more scattered than in the ellipse graph in Figure 5.4. A few of the groups look to be close

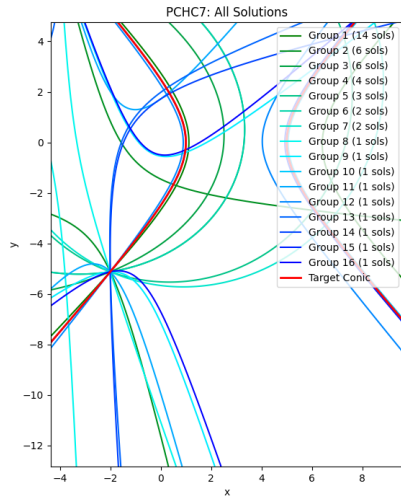


Figure 5.6: Corresponding Conic Equation of Grouped Solutions for the Ellipse Problem

to the red equation, however the rest of the conics are fairly spread out. A similar note to the results from the parabola problem shown in Figure 5.5 are that about half the conics for the hyperbola problem appear to intersect close to $(-2, -5)$ on the lower left of the graph.

Table 5.5: Approximate Solutions Grouped by 2-norm Difference Less Than $1e - 4$ for the Hyperbola Problem, G_i reading “Group i ”

	G 1	G 2	G 3	G 4	G 5	G 6	G 7	G 8 - G 9	G 10 - G 16
Trial 1	14	6	6	4	3	2	2	1	1
Trial 2	9	9	9	5	4	3	3	2	1

One remark is that each trial of PCHC7, tracking all 108 paths from the starting solution set, took about 3000 seconds or 50 minutes on average vs NAG4M2 which produced results in less than ten seconds. If PCHC7 is mapped to hardware as an SLP and optimized, it will be significantly faster and is believed to outperform methods like `solveSystem` in NAG4M2 which does not use SLPs.

5.4.1 Generating A Large Dataset

Manually determining conic problems is time consuming, and to gather more data on our method, it is of interest to have a larger dataset. We conclude this chapter with a technique

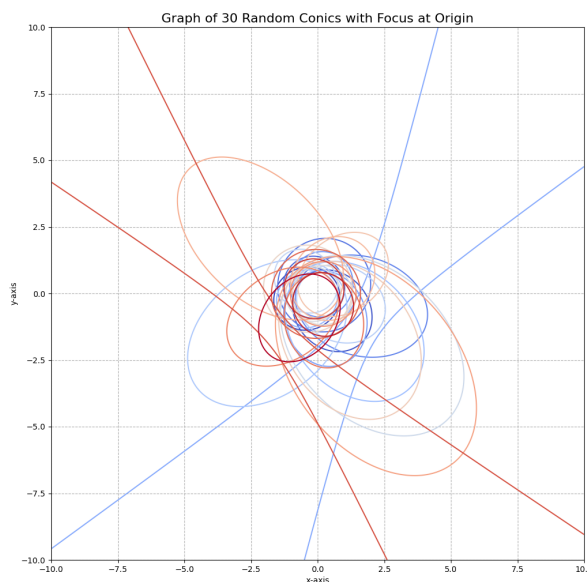


Figure 5.7: Graph of the coefficients of 30 randomly generated conics with focus at the origin.

to generate synthetic data. We write a program to generate random conic problems of a similar type where two coordinates are known points, two coordinates are 0 in each of x, y and contain a linear equation for x, y , and one coordinate is (x, y) :

$$(\alpha_1, \beta_1), (\alpha_2, \beta_2), (L_1(x), 0), (0, L_2(y)), (x, y)$$

for $\alpha_i, \beta_i \in \mathbb{R}$ and L_1, L_2 linear. This is done by generating random coefficients for the conic with the restriction that a focus is at the origin. Then eight points are selected on the conic, two passing through the x -axis, two passing through the y -axis, and four chosen randomly. Two points are selected to be (x, y) , the x, y solutions to the problem, and then the respective equations are selected for the points passing through the x -axis and y -axis. We generate thirty random conics with focus at the origin, graphed in Figure 5.7 by selecting values for C, D, E randomly from $(-1, 1)$, solving for A, B based on the focus-at-the-origin restriction, and then discarding any coordinates that will lead to an invalid

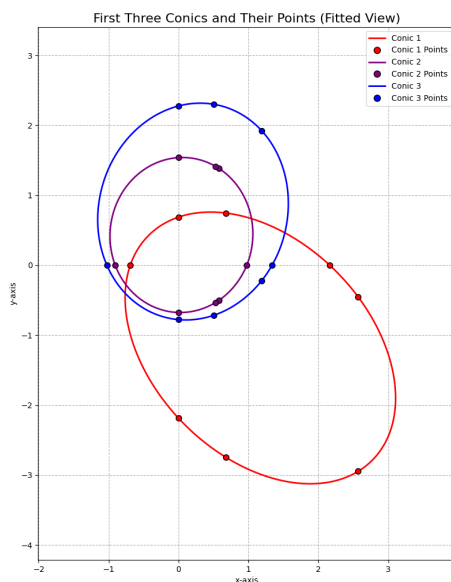


Figure 5.8: Graph of first three randomly generated conics and their corresponding eight points plotted in the same color. Each conic has two points through the x -axis and two points through the y -axis and four randomly chosen points.

conic or one that doesn't pass through both the x -axis and y -axis.

The first three conics we randomly generated are shown in Figure 5.8, with the selected points plotted on each conic in the same color.

CHAPTER 6

CONCLUSION

This thesis introduced `HGates.m2`, a computational framework written in Macaulay2 designed to model numerical algebraic geometry algorithms as Straight-line Programs (SLPs) for hardware acceleration. We have defined all the fundamental building blocks or gates necessary to begin solving nonlinear algebra problems and have demonstrated the application of a collection of well-used algorithms for solving nonlinear problems as straight-line programs. We validated this framework by implementing and comparing these algorithms to solve a nonlinear system derived from a class of Keplerian orbit optical navigation problems.

In summary, this work demonstrates that `HGates.m2` can be used to build SLPs to solve nonlinear problems. Our results show that PCHC7 is a viable and efficient method for a class of Keplerian orbit problems, and that the choice of system formulation and solver parameters has a significant impact on performance and should be tested for.

6.0.1 Future Work

The modular and abstract design of this framework opens several compelling paths for future research and practical applications. In addition to providing the building blocks for more advanced techniques such as certification and Padé approximants, the `HGates.m2` framework is well-suited for integration with external, high-performance libraries, such as LAPACK[34], allowing this work to leverage state-of-the-art, fast-performing linear algebra algorithms. This ensures a separation with the functional model and the low-level implementation to support conceptual simplicity in working with high-level systems in nonlinear algebra.

Another equally significant extension is that `HGates.m2` is designed to be easily for-

mally verified due to its modularity. Because the framework is built on a finite number of gates, the proof verification will be focused on the gate relations in the functional model, separating from lower-language specific constructs such as valid memory referencing and overflows that are off-loaded to an external library. This will provide a much more simplified verification process with an interactive theorem prover, such as Rocq, Lean or Isabelle. Furthermore, this integration with formal verification offers an opportunity to pair `HGates.m2` with AI generation for correct-by-construction code. By verifying the gates and relations between gates, AI models can be used to autogenerate corresponding, provably correct algorithms in `HGates.m2` .

REFERENCES

- [1] T. D. Han and T. S. Abdelrahman, “Reducing branch divergence in gpu programs,” in *Proceedings of the 4th International Workshop on General-Purpose Computation on Graphics Processing Units*, ACM, 2011, pp. 35–42.
- [2] F. Liu *et al.*, “Gpu-based branchless distance-driven projection and backprojection,” *IEEE Transactions on Medical Imaging*, vol. 36, no. 3, pp. 754–764, 2017.
- [3] A. Mitra, D. G. Politte, B. R. Whiting, J. F. Williamson, and J. A. O’Sullivan, “Multi-gpu acceleration of branchless distance driven projection and backprojection for clinical helical CT,” *Journal of Imaging Science and Technology*, vol. 61, no. 1, p. 010405, Jan. 2017.
- [4] J. W. Demmel, *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), 1997.
- [5] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th. Johns Hopkins University Press, 2013.
- [6] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd. Society for Industrial and Applied Mathematics (SIAM), 2003.
- [7] L. N. Trefethen and D. Bau III, *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), 1997.
- [8] R. L. Burden and J. D. Faires, *Numerical Analysis*, 9th. Brooks/Cole, 2011.
- [9] J. Stoer and R. Bulirsch, *Introduction to Numerical Analysis*, 3rd. Springer, 2002.
- [10] U. M. Ascher and L. R. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. Society for Industrial and Applied Mathematics (SIAM), 1998.
- [11] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems*, 2nd. Springer-Verlag Berlin Heidelberg, 1993.
- [12] S. D’Amico, J.-S. Ardaens, G. Gaias, H. Benninghoff, B. Schlepp, and J. L. Jørgensen, “Noncooperative rendezvous using angles-only optical navigation: System design and flight results,” *Journal of Guidance, Control, and Dynamics*, vol. 36, no. 6, pp. 1576–1595, 2013.
- [13] V. Franzese, F. Topputo, and F. Ankersen, “Deep-space optical navigation for m-argo mission,” *Journal of the Astronautical Sciences*, vol. 68, pp. 1034–1055, Dec. 2021.

- [14] S. Li, R. Lu, L. Zhang, and Y. Peng, “Image processing algorithms for deep-space autonomous optical navigation,” *Journal of Navigation*, vol. 66, no. 4, pp. 605–623, 2013.
- [15] G. Gaias, S. D’Amico, and J.-S. Ardaens, “Angles-only navigation to a noncooperative satellite using relative orbital elements,” *Journal of Guidance, Control, and Dynamics*, vol. 37, no. 2, pp. 439–451, 2014.
- [16] A. G. Butkevich, “Proper motion and secular variations of keplerian orbital elements,” *Frontiers in Astronomy and Space Sciences*, vol. 5, p. 18, 2018.
- [17] Y. Wigderson, *Algebraic complexity*, Mathcamp 2024 Notes, 2024.
- [18] R. J. Lipton, “Straight-line complexity and integer factorization,” in *Proceedings of the Second International Symposium on Parallel and Distributed Computing (ISPDC ’94)*, Princeton, NJ: IEEE Computer Society Press, 1994, pp. 71–79.
- [19] V. Strassen, “Gaussian elimination is not optimal,” *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, 1969.
- [20] D. Coppersmith and S. Winograd, “Matrix multiplication via arithmetic progressions,” *Journal of Symbolic Computation*, vol. 9, no. 3, pp. 251–280, 1990.
- [21] V. Vassilevska Williams, Y. Xu, Z. Xu, and R. Zhou, “New bounds for matrix multiplication: From alpha to omega,” *ACM Transactions on Computation Theory*, vol. 16, no. 2, 8:1–8:29, 2024.
- [22] D. R. Grayson and M. E. Stillman, *Macaulay2, a software system for research in algebraic geometry*, <http://www.math.uiuc.edu/Macaulay2/>.
- [23] A. Leykin, *Introduction to algebraic computation*, Online class text, 2024.
- [24] Wikipedia contributors, *Runge–kutta methods — Wikipedia, the free encyclopedia*, [Online; accessed 2-November-2025], 2025.
- [25] C. B. Boyer, *History of Analytic Geometry*. Mineola, NY: Dover Publications, 2004, ISBN: 9780486438320.
- [26] V. J. Katz, *A history of mathematics*. Addison-Wesley, 1998.
- [27] Saylor Academy, *Conic sections*, https://saylordotorg.github.io/text_intermediate-algebra/s11-conic-sections.html, Accessed: 2025-11-01.
- [28] G. Strang and E. ”. Herman. “11.5: Conic sections,” Mathematics LibreTexts. (2025), (visited on 11/06/2025).

- [29] “Conic sections – types, properties, and examples,” SOM – Story of Mathematics. (2025), (visited on 11/06/2025).
- [30] R. R. Bate, D. D. Mueller, and J. E. White, *Fundamentals of Astrodynamics*. New York: Dover Publications, 1971, ISBN: 978-0-486-60061-1.
- [31] M. Mancini, *ALGEBRAIC METHODS IN SPACECRAFT NAVIGATION*, Ph.D. Dissertation, May 2025.
- [32] Wolfram, *Isotropic Line*, From MathWorld—A Wolfram Web Resource, Accessed: 2025-09-16.
- [33] A. Leykin and R. Krone, *NumericalAlgebraicGeometry: A Macaulay2 package. Version 1.24*.
- [34] E. Anderson *et al.*, *LAPACK Users’ Guide*, Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999, ISBN: 0-89871-447-8.