



Homotopy gets real

Juan Camilo González Cabrera

Universidad de los Andes
Facultad de Ingenieria, Departamento de Ingenieria de Sistemas y Computacion
Bogotá, Colombia
2024

Homotopy gets real

Juan Camilo González Cabrera

Trabajo de grado presentado como requisito parcial para optar al título de:
Ingeniero de Sistemas y Computacion

Director:
Nicolas Cardozo Alvarez

Asistente de tesis:
Daniel Ricardo Barrero Rosero

Universidad de los Andes
Facultad de Ingenieria, Departamento de Ingenieria de Sistemas y Computacion
Bogotá, Colombia
2024

Acknowledgments

I would like to express my deepest gratitude to my thesis advisor, Nicolas Cardozo Alvarez, and to Daniel Ricardo Barrero Rosero. Without their invaluable ideas and corrections, this work would not have been possible. I extend my heartfelt thanks to my family and my cat for their unconditional support throughout this journey. Special thanks to Dr. Silviana Amethyst for her guidance and for creating BertiniReal, and to Dr. Paul Breiding for his orientation and for HomotopyContinuation.jl. Lastly, I am inspired by the words of Sir Winston Churchill:

This is the lesson: never give in, never give in, never, never, never,
never in nothing, great or small, large or petty never give in except to
convictions of honour and good sense.

Thank you all for your support and encouragement.

Abstract

This thesis presents the development of a new module for the Julia programming language, designed to decompose the real part of complex curves within the field of computational algebraic geometry. The work leverages homotopic approaches to bridge gaps in existing tools. The core objective is to implement a Julia package inspired by BertiniReal, aiming to efficiently handle the decomposition of algebraic sets into their real components. Julia's robust features and growing prominence in scientific computing motivate the choice of the language. The thesis details the foundational concepts, the implementation of the new module, and its validation through comparative analysis, demonstrating the reliability and utility.

Contents

1	Introduction	1
1.1	Context	1
1.2	Thesis Structure	1
2	Foundational Concepts and Tools	3
2.1	Homotopy methods	3
2.2	Witness Sets and NID	4
2.3	Julia	4
2.4	Bertini	6
2.4.1	Input Specifications	6
2.4.2	Computation Outputs	7
2.5	BertiniReal	7
2.5.1	Computation Outputs	8
3	Decomposition of Algebraic Curves	10
3.1	A Homotopy Continuation Method	10
3.1.1	Compute critical points	11
3.1.2	Intersect with bounding object	12
3.1.3	Slice at projection interval midpoints	12
3.1.4	Connect the dots	13
3.1.5	Merge	13
3.1.6	Sample	13
3.2	Implementation in Julia	14
3.2.1	Parser	14
3.2.2	Six Steps	15
3.2.3	Compute Critical Points	15
3.2.4	Intersect with Bounding Object	16
3.2.5	Slice at Projection Interval Midpoints	17
3.2.6	Connect the Dots	18
3.2.7	Merge	20
3.2.8	Sample	21
3.2.9	Graphical Plotting	23
3.3	The Julia Package	24
3.3.1	Installation	24
3.3.2	Usage	25

4	Validation	26
4.1	Validating Results	26
4.2	Visualization of Results	28
5	Conclusions and Future Work	30
5.1	Conclusions	30
5.2	Future Work	30

1 Introduction

1.1. Context

This thesis presents the development of a new module for the Julia programming language, designed to decompose the real part of complex curves or surfaces within the field of computational algebraic geometry. While existing tools in this field are often specialized and limited in scope, this work seeks to bridge these gaps by leveraging homotopic approaches, which are crucial for solving intricate algebraic geometry problems more universally.

The core objective of this project is to implement a Julia package, inspired by and aiming to complement the capabilities of BertiniReal, that efficiently handles the decomposition of algebraic sets into their real components. Julia robust features, coupled with the language's growing prominence in scientific computing, motivates the language choice.

In the initial phase of this project, We engaged with the developers of HomotopyContinuation.jl [BT24], a Julia package that employs homotopy continuation methods to track solution paths of polynomial systems. This interaction revealed a significant gap in the Julia ecosystem: the lack of a tool capable of performing real decomposition of algebraic sets comparable to BertiniReal. Inspired by this gap, I embarked on creating a package that extends the functionalities of Julia in computational algebraic geometry.

Through guidance from the creators of BertiniReal, along with a study of their software's architecture and underlying algorithms, this project has been meticulously crafted to offer a powerful tool that enhances the computational capabilities for researchers and practitioners in the field, focusing specifically on the real decomposition of curves and surfaces.

1.2. Thesis Structure

The thesis is structured to provide a coherent and detailed account of the project from conceptualization to validation. The organization is as follows:

Chapter 2: Foundational Concepts and Tools

Newly introduced, this chapter provides a comprehensive overview of the methodologies and tools that form the basis of this thesis. It covers:

- Homotopy Methods: Explaining the concept and application of homotopy in solving polynomial systems.
- Julia Programming Language: Discussing the features that make Julia ideal for computational tasks in algebraic geometry.
- Bertini Software: Detailing Bertini's core functionalities and its use in numerical homotopy continuation methods.
- BertiniReal Software: Describing how BertiniReal extends Bertini's functionalities to focus on real algebraic sets.

Chapter 3: Decomposition of Algebraic Curves

Building upon the theoretical groundwork laid in the previous chapters, this chapter delves into the practical aspects of the project. It discusses the development of the new Julia module, inspired by the capabilities of Bertini and BertiniReal. The implementation details are outlined, focusing on how the specific problems aimed to be solved influenced the development process and the choice of tools.

Chapter 4: Validation

In order to assess the effectiveness and accuracy of our Julia module, this chapter compares its performance against BertiniReal. Through a series of tests and evaluations, we demonstrate how our project aligns with, and potentially enhances, the existing solutions provided by BertiniReal. This validation is crucial for establishing the reliability and utility of the new tool within the computational algebraic geometry landscape.

Chapter 5: Conclusions and Future Work

The final chapter synthesizes the findings of the thesis, highlighting the contributions to the field and the implications of this work for future research. It outlines potential enhancements to the Julia module can be further exploited and improved.

2 Foundational Concepts and Tools

2.1. Homotopy methods

These methods rely on the concept of homotopy to solve systems of polynomial equations. These methods provide a systematic approach to track the solutions of these equations as they evolve from a simple, well-understood system to the complex system of interest.

Definition of Homotopy

Intuitively, a homotopy is a continuous deformation between two functions or maps. Formally, consider two continuous functions $f, g : X \rightarrow Y$ between spaces X and Y . A homotopy H from f to g is a continuous function

$$H : X \times [0, 1] \rightarrow Y \text{ such that}$$

$$H(x, 0) = f(x) \text{ and } H(x, 1) = g(x) \text{ for all } x \in X \text{ [Bre93].}$$

This definition can be adapted to the context of polynomial systems, where f and g represent different polynomial systems, and H represents the continuum of systems interpolating between them.

In the context of solving polynomial equations, a homotopy connects a system with known solutions (the start system) to a target system whose solutions are sought. The homotopy thus defined provides a path along which solutions can evolve, and these paths are tracked numerically.

Tracking the paths of solutions involves numerical integration techniques paired with predictor-corrector methods to accurately follow the solution paths from the start system to the target system. This method ensures that each path leads from a known solution of the start system to a solution of the target system, uncovering all potential solutions.

Depending on the structure of the polynomial systems involved, different types of homotopies may be employed:

- **Linear Homotopy:** This straightforward approach uses a linear path connecting the coefficients of the start and target systems, but may encounter difficulties such as undefined or infinite solutions.

- **Polyhedral Homotopy:** Utilizes a transformation based on the specific geometry of the polynomial systems, minimizing the number of tracked paths and increasing efficiency.
- **Projective Homotopy:** Adapts the system into projective space to handle complications like solutions at infinity, providing robustness in continuation.

Example : Linear Homotopy

Consider the simple case of transitioning between two quadratic polynomials:

$$f(x) = x^2 - 4 \text{ and } g(x) = x^2 - 9$$

A linear homotopy connecting these systems could be:

$$H(x, t) = x^2 - (4 + 5t)$$

At $t = 0$, $H(x, 0) = f(x)$ and at $t = 1$, $H(x, 1) = g(x)$. This homotopy linearly changes the constant term from -4 to -9, tracing the solution paths from $x = \pm 2$ to $x = \pm 3$.

2.2. Witness Sets and NID

Algebraic sets, which are the solution sets of polynomial systems, may have structures with many irreducible components each having distinct dimensions, degrees, and multiplicity structures. It is possible to decompose an algebraic set into its irreducible components, obtaining a numerical description of each component via a witness set. This process is known as numerical irreducible decomposition (NID) [ABH⁺17].

For an irreducible component $C \subset \mathbb{C}^N$, a witness set for C includes information about the dimension and degree of C as well as the defining polynomial system f such that C is an irreducible component of the variety defined by f , denoted as:

$$V(f) = \{x \in \mathbb{C}^N \mid f(x) = 0\}$$

Concisely, a witness set for the irreducible component C is represented by a triple $W = \langle f, L, W \rangle$, where $L \subset \mathbb{C}^N$ is a general linear space of codimension equal to the dimension of C , and $W = C \cap L$ with $\deg(C) = |W|$. The components f , L , and the set of points W are the witness system, witness slice, and witness point set for C , respectively.

2.3. Julia

This section sets the stage for later discussions in the thesis on the specific implementation details of the new Julia module, its integration with existing computational

tools, and its performance evaluation against other established software in the field. By highlighting Julia's strengths and its relevance to the project, the thesis underscores the rationale behind the choice of programming language and its potential impact on the field of computational algebraic geometry.

Julia is a high-performance, high-level programming language that is increasingly popular among scientists, engineers, and mathematicians for its powerful capabilities in numerical and computational applications. Designed to address the needs of high-performance numerical and scientific computing, Julia combines the speed of compiled languages like C with the simplicity and interactivity of scripting languages like Python.

Key Features

- **Speed:** Julia is designed with performance in mind. Its just-in-time (JIT) compilation ensures that it can run at speeds comparable to statically-typed, compiled languages.
- **Dynamic Typing:** Julia offers the flexibility of dynamic types without sacrificing performance, making it ideal for exploratory programming and rapid prototyping in research.
- **Multiple Dispatch:** This allows Julia to choose which version of a function to execute based on the types of all arguments, making it highly effective for defining function behaviors across different types of inputs. This is particularly useful in computational algebraic geometry, where operations may vary significantly between different types of mathematical objects.
- **Metaprogramming:** Julia's powerful metaprogramming capabilities allow for generating and including pieces of code programmatically, which is advantageous when developing complex mathematical models and algorithms.
- **Rich Ecosystem:** Julia has a vibrant ecosystem of packages, including numerous libraries specifically for mathematical, statistical, and engineering functions. Notably, the Julia package for homotopy continuation methods [BT24] is specifically designed to leverage Julia's strengths in handling complex numerical computations.

In the realm of computational algebraic geometry, Julia's capabilities allow for the efficient implementation of complex mathematical models and methods, such as those involving homotopies. The language's ability to handle high-level abstract mathematical coding while ensuring that the computational performance remains optimal is crucial for dealing with the large-scale polynomial systems typical in algebraic geometry.

2.4. Bertini

Before delving into the implementation of a Julia package that mirrors the capabilities of BertiniReal, it is crucial to first understand Bertini [BHSW23] itself, as BertiniReal fundamentally builds upon it. Bertini is a well regarded software package used for solving systems of polynomial equations via homotopy continuation methods. By comprehending Bertini’s core functionalities and its application in computational algebraic geometry, we can better appreciate the foundations upon which BertiniReal and by extension, our Julia implementation is constructed.

Bertini employs numerical homotopy continuation methods to find all solutions to a system of polynomial equations. This approach involves defining a start system with known solutions and continuously transforming it into the target system whose solutions are sought. Bertini effectively tracks these solution paths through the parameter space, a process that is essential for ensuring the accuracy and completeness of the solution set.

The software performs the following tasks:

- **Numerical Irreducible Decomposition (NID):** Bertini decomposes any given polynomial system into its irreducible components, providing information about each component such as its dimension and degree.
- **Regeneration:** A technique used to refine the approximation of the solution set by incrementally adding equations to the system and tracking the evolving solutions.
- **Deflation:** To handle singular solutions or solutions at which the Jacobian of the system degenerates, Bertini uses deflation techniques to adjust the system and continue tracking solutions accurately.

2.4.1 Input Specifications

Bertini process inputs that define polynomial systems, typically provided in a text file using a specific syntax. The input file specifies the polynomial equations, the configuration settings for the homotopy continuation process, and any additional parameters that controls the behavior of the solver. We now list the key components of a Bertini input file:

- **Polynomial System:** The set of polynomial equations that need to be solved.
- **Configuration Settings:** Options that determine how the homotopy continuation methods are applied, including the choice of homotopy, path tracking options, and convergence criteria.
- **Parameter Homotopy:** For certain complex systems, users may specify parameter homotopies where the coefficients of the polynomials are themselves functions of parameters being continued.

2.4.2 Computation Outputs

Bertini produces several types of outputs, which are essential for further computations and analyses:

- **Solution Files:** These files contain the numerical solutions to the polynomial system, including real and complex solutions, their multiplicities, and precision metrics.
- **Decomposition Files:** For systems that have a numerical irreducible decomposition, Bertini outputs information about the irreducible components, such as their dimension and degree.
- **Witness Sets:** Perhaps the most crucial output for the functionality of BertiniReal is the witness set. A witness set comprises a numerical representation of the solution sets of a polynomial system. Each witness set includes:
 - **Dimension Information:** The dimension of the component the witness set represents.
 - **Degree Information:** The number of points in the witness set.
 - **Points:** The actual points that forms the witness set.

Understanding Bertini is fundamental for our project because BertiniReal uses Bertini’s capabilities to focus specifically on the real components of algebraic sets. BertiniReal uses the outputs of Bertini, particularly the witness sets, to construct real algebraic sets and then decomposes them into their irreducible components. Thus, any implementation aiming to replicate or enhance BertiniReal’s functionality in Julia must first incorporate the homotopy continuation methods and other techniques employed by Bertini. In the subsequent sections, we will explore how these methodologies have been adapted and integrated into our Julia package.

2.5. BertiniReal

BertiniReal [ABH⁺24] is designed specifically to handle the real algebraic sets defined by systems of polynomial equations solved by Bertini. By utilizing the complex solutions provided by Bertini, BertiniReal identifies and decomposes the real part of a complex curve or surface. This section details how BertiniReal operates, its unique features, and its integration with Bertini, providing a solid foundation for understanding the adaptations needed for the Julia implementation.

Here are the primary functionalities of BertiniReal:

- **Real Solution Decomposition:** BertiniReal takes the witness sets generated by Bertini, which include solutions to the polynomial system, and uses these to identify and isolate real solutions.

- **Topological Decomposition:** Once real solutions are identified, BertiniReal performs a further decomposition to understand and categorize the topology of these real components. This involves distinguishing between different types of real structures, such as isolated points, curves, and surfaces.
- **Visualization and Analysis:** BertiniReal provides tools for visualizing the decomposed real algebraic sets, allowing for a graphical interpretation of the solutions. This is crucial for both theoretical research and practical applications where visual assessments are necessary.

2.5.1 Computation Outputs

BertiniReal enhances the capabilities of Bertini by focusing specifically on the real components of algebraic sets and provides detailed outputs that are indispensable for further analyses and applications in computational algebraic geometry. Understanding these outputs is crucial for ensuring that our Julia implementation can effectively replicate and perhaps enhance these functionalities.

The outputs produced by BertiniReal are particularly tailored to provide a comprehensive understanding of the real algebraic geometry involved. These outputs include:

- **Decomposed Components:** BertiniReal outputs detailed information about the real decomposed components of the solution set. This includes data on isolated points, curves, and surfaces, which are classified based on their topological characteristics.
- **Connectivity Data:** For complex structures like curves and surfaces, BertiniReal provides connectivity data that describes how various parts of these structures are linked. This is crucial for understanding the underlying topology of the algebraic sets.
- **Numerical Data:** Each component is accompanied by precise numerical data, which includes coordinates of points, parameterization of curves, and descriptions of surfaces. This data is essential for accurate visualization and further numerical analysis.
- **Visualization Files:** BertiniReal generates files that are specifically formatted for use with visualization tools. These files allow users to graphically explore the real algebraic sets, providing a visual understanding of the complex geometries.

The functionality of BertiniReal directly informs the development of our Julia package. By understanding how BertiniReal processes witness sets to extract and decompose real algebraic sets, we can tailor our Julia implementation to replicate these processes effectively. Furthermore, adapting these functionalities into Julia involves

leveraging Julia's strengths in numerical computing and ease of integration with visualization libraries to enhance the user experience and computational efficiency.

3 Decomposition of Algebraic Curves

Currently, our Julia module specifically implements the decomposition of algebraic curves. While the methodology is robust enough to eventually support the decomposition of more complex surfaces, this initial phase focuses on perfecting the techniques and algorithms necessary for accurately decomposing and visualizing real curves defined by systems of polynomial equations. This targeted approach allows us to refine the functionality and ensure high precision in the curve decomposition processes before extending the module to handle higher-dimensional algebraic sets. In our Julia module, having the option to handle some of the inputs and outputs from Bertini allows for a seamless integration of existing methodologies and enables compatibility with users already familiar with Bertini and BertiniReal. Moreover, by understanding these inputs and outputs, our implementation can be designed to enhance user experience, provide more intuitive interaction, and possibly extend the capabilities to include better analyses or visualizations of the solution sets and their geometric properties.

For our Julia package, handling outputs involves several considerations, particularly focusing on data management. This encompasses the efficient management and storage of the extensive numerical and topological data generated during the decomposition process. Ensuring that this data is accurately and systematically organized is crucial for maintaining the integrity and usability of the outputs from the computational process.

3.1. A Homotopy Continuation Method

To effectively implement the decomposition of algebraic curves in our Julia module, it is essential to understand how BertiniReal accomplishes this task. BertiniReal's method for decomposing curves is serving as the foundation upon which our implementation is built. The following section outlines the specific steps BertiniReal uses to decompose curves.

Consider $C \subset \mathbb{C}^N$ as a complex algebraic curve, defined as a one-dimensional component of $V(f)$, where $V(f)$ represents the complex solution set of the polynomial system $f = 0$. We focus on decomposing the real points of C , denoted as $C \cap \mathbb{R}^N$, utilizing a suitably general real linear projection $\pi_0 : \mathbb{R}^N \rightarrow \mathbb{R}$. Through this projection, $C \cap \mathbb{R}^N$ is segmented into cells, each parametrized over a real line segment by

π_0 . The separation of these segments is achieved by the projection of critical points via π_0 .

3.1.1 Compute critical points

The fundamental step is the computation of the critical points of $C \cap \mathbb{R}^N$ with respect to the projection π_0 . These critical points are crucial as they indicate where changes in parameterization are necessary; away from these points, the curve C is smooth and parameterized by π_0 . They are given by points on $C \cap \mathbb{R}^N$ for which the Jacobian matrix of the witness system f for C , concatenated with the direction of π_0 , is rank-deficient. These points are exactly the solutions to the equation:

$$\begin{bmatrix} f(x) \\ \det \begin{pmatrix} Jf(x) \\ J\pi_0 \end{pmatrix} \end{bmatrix} = 0 \quad (1)$$

To obtain the critical points, regeneration is performed starting from a witness set for C computed by Bertini. This involves introducing new variables v , which exist in the left null space of the matrix from equation (1). The resulting homotopy for computing the critical points is formulated as:

$$(1-t) \begin{bmatrix} f(x) \\ v^T \begin{bmatrix} Jf(x) \\ J\pi_0 \end{bmatrix} \\ \text{patch}_v(v) \end{bmatrix} + t \begin{bmatrix} f(x) \\ M_1(v) \prod_{i=1}^{d-1} L_{1,i}(x) \\ \vdots \\ M_N(v) \prod_{i=1}^{d-1} L_{N,i}(x) \\ \text{patch}_v(v) \end{bmatrix} = 0 \quad (2)$$

Here, d represents the maximum degree of any polynomial in f , and the functions $M_j(v)$ and $L_{j,i}$ are random complex linear functions of v and x , respectively. Since v represents a null vector, it naturally resides in the projective space, where projective points correspond to lines through the origin in $\mathbb{C}^n$. We ensure a unique representation in $\mathbb{C}^n$ by intersecting with a random affine hyperplane, appending $\text{patch}_v(v)$ a random affine linear equation in v .

The starting points for the homotopy in equation (2) are obtained by solving two systems: one in x and one in v . The x coordinates are determined by the witness point set of C , and the v coordinates are found using numerical linear algebra to solve the system formed by the matrix of M linear equations and the patch_v linear, inverted as in equation (3).

$$v_i = \begin{bmatrix} M_1 \\ \vdots \\ M_{k \neq i} \\ \vdots \\ M_N \\ \text{patch}_v \end{bmatrix}^{-1} \cdot \begin{bmatrix} 0 \\ \vdots \\ 1 \end{bmatrix} \quad (3)$$

Once the start variables x and v are established, they are concatenated to form start points $s_{i,j} = (x_{i,j}, v_i)$, and the homotopy movement is performed to solve the left nullspace system and obtain the critical points. The result is a set of critical points χ , as we are performing a decomposition of the real portion of the component we discard the complex solutions.

3.1.2 Intersect with bounding object

We acknowledge that the critical regions of the curve C might extend towards infinity. To maintain all arcs finite and manageably within the computational realm, we intersect C with a sphere of radius r and center c .

By default, the bounding sphere is centered at the centroid of the critical points computed in the first step, with a radius three times the maximum distance from the centroid to any critical point. This configuration ensures that all branches, isolated, singular, and critical points with respect to π_0 are encompassed within the sphere, providing ample space for clear visualization. If C is part of a larger decomposition, it inherits c and r from that overarching decomposition.

The intersection points of C with the sphere are computed through a simple two-step regeneration similar to the homotopy discussed previously. Initially, we track the general witness linear L to two other general linear equations L_1, L_2 due to the quadratic nature of the sphere function, then resolve the homotopy in equation (4).

$$(1-t) \left[\frac{f(x)}{\|x-c\|_2^2 - r^2} \right] + t \left[\frac{f(x)}{L_1(x) \cdot L_2(x)} \right] = 0 \quad (4)$$

Any complex solutions found are discarded.

3.1.3 Slice at projection interval midpoints

Having the points of χ , the set of points where notable geometric phenomena occur, the subsequent step involves determining midpoints for the intervals between these critical points. Therefore, we organize the projection values of the critical points as $\pi_0(\chi) = \{p_{cj}\}$, ensuring uniqueness up to a specified numerical tolerance. Corresponding midpoints for these projection value intervals are defined as $\{p_{mi}\} = \{(p_{ci} + p_{ci+1})/2\}$.

To locate points above the interval bisector in terms of projection value, for each p_{mi} , we shift the general witness line L (input from the witness set) to the linear projection $\pi_0(\mathbf{x}) - p_{mi}$. This adjustment aims to obtain midpoints in the fiber of projection value p_{mi} through the homotopy in equation (5).

$$(1-t) \left[\frac{f(x)}{\pi_0(x) - p_{mi}} \right] + t \left[\frac{f(x)}{L(x)} \right] = 0 \quad (5)$$

Again, any complex solutions found are discarded.

3.1.4 Connect the dots

In this stage, the midpoints are tracked to the critical points to establish the edges of the decomposition structure. For each midpoint, we aim to identify its immediate neighbors to the left and right. This is accomplished by executing another homotopy, focused solely on projection value adjustments. For $j = i, i + 1$, we transition the projection $\pi_0(\mathbf{x}) - p_{mi}$ to $\pi_0(\mathbf{x}) - p_{cj}$ through the homotopy in equation (6).

$$(1 - t) \begin{bmatrix} f(x) \\ \pi_0(x) - p_{cj} \end{bmatrix} + t \begin{bmatrix} f(x) \\ \pi_0(x) - p_{mi} \end{bmatrix} = 0 \quad (6)$$

This procedure ensures that each midpoint is associated with its respective segment of the curve by tracing paths between critical and midpoint projections.

All the singularities of this homotopy are confined to the fibers over p_{cj} , thus the paths are typically smooth, free from singularities except possibly at the endpoints. It may occur that a midpoint does not connect to a pre-established point if the segment it represents does not reach a critical state at the expected projection value but another segment does. In such cases, new points may be identified and labeled as 'new' for potential removal during the merging phase.

3.1.5 Merge

As an optional component of the curve decomposition algorithm, edge merging plays a crucial role. The candidates for merging are those edges identified in the connection step as having 'new' endpoints. An optimized merging method is employed

Initially, we identify merge candidates among the decomposed edges, specifically looking for edges that terminate with a 'new' right point. The process involves iterating to the right along connecting edges, compiling an ordered list of edges to be merged, until an edge without a new endpoint is encountered. We then engage a homotopy, similar to the one described in equation (6), originating from the midpoint among the candidates to a new midpoint. This new midpoint is calculated based on the average projection values of the outer endpoints of the edges involved. This merging procedure is iterated until all eligible edges are merged. The strategy aims to merge as many edges as simultaneously possible, rather than sequentially, optimizing the overall merging process.

3.1.6 Sample

While it would be feasible to conclude the decomposition process at this point, having achieved a complete characterization of C , including all critical points and detailed topological information on how branches of the curve converge at these points and extend towards infinity, further refinement might be desired for specific applications. Such applications could include integration over the curve or enhanced visualization, which necessitates further refinement of the data.

Each midpoint in the decomposition is not only defined by its coordinates but also possesses a domain of validity defined by the π_0 values of the adjoining points and is connected by a clearly defined, singularity-free homotopy, facilitating straightforward refinement. The refinement process involves adjusting the linear equations defined by the projection π_0 , capitalizing on the computational simplicity of linear movements within the projection space.

Bertini real offers two refinement routines through its supplementary Sampler program: a fixed-number method and an adaptive sampling method. The fixed-number method assigns a uniform number of samples across each edge, evenly spaced in terms of projection values. In contrast, the adaptive method incrementally adds samples to reduce the length of adjoining line segments until they fall below a predetermined threshold.

3.2. Implementation in Julia

We now turn our attention to the Julia implementation of the real algebraic geometry supported by homotopy. The module developed encapsulates several critical components that are instrumental for its operation. These components include a Parser, which is responsible for interpreting and organizing input polynomial systems into a format that the subsequent processes can utilize efficiently. The Six Steps module, inspired by the Bertini real method, meticulously handles the decomposition of curves by applying the six fundamental steps of numerical algebraic geometry. Finally, the Graphical Plotting component is designed to visualize the results, providing graphical representations of the algebraic sets as decomposed by the algorithm.

3.2.1 Parser

The Parser component of our Julia module serves a critical dual function, essential for the smooth operation of the entire decomposition process. First, it is tasked with interpreting the outputs generated by Bertini. This involves parsing complex data files that contain detailed information about the polynomial systems' solutions, including witness sets and their properties. Once the data is successfully parsed, the Parser's second crucial role comes into play: organizing this information in a structured format that is optimally suited for use in the subsequent six steps of the curve decomposition process. This organization ensures that all necessary data is accessible and formatted correctly, enabling efficient processing and accurate results in the decomposition steps that follow.

3.2.2 Six Steps

Our Julia module processes algebraic curves through six meticulously defined steps, described in section 3.1, to handle the complexity of curve decomposition efficiently. Below, we outline each step, focusing on the essential functionalities. This section describes these steps, each followed by an illustrative figure showing the results of applying the steps to the example curve $(x^3 - xy^2 + y + 1)^2 \cdot (x^2 + y^2 - 1) + y^2 - 5 = 0$. The complete Julia code implementations for each step are available in the URL. <https://github.com/JuanCaGC/HomotopyGetsReal.jl>

3.2.3 Compute Critical Points

The process begins with the computation of critical points where the curve's behavior changes, such as cusps or intersections, using the Jacobian matrix to identify rank deficiencies.

```
function compute_critical_points(W_curve, projections)
    solve_out = compute_crit_nullspace(W_curve, randomizer
    , projections, 1, 1, 1)

    W_crit_real = get_noninfinite_w_mult_full(solve_out)
    W_singular = get_sing(solve_out)

    sort_for_real!(W_crit_real)
    sort_for_unique!(W_crit_real)

    sort_for_real!(W_singular)
    sort_for_unique!(W_singular)
end
```

The *compute_critical_points* function calculates critical points where the behavior of the curve changes, such as cusps or intersections. It uses the Jacobian matrix to identify points of rank deficiency and then solves the system to find these critical points. The results are then sorted and filtered to retain only real and unique points, which are essential for the next steps in the decomposition process.

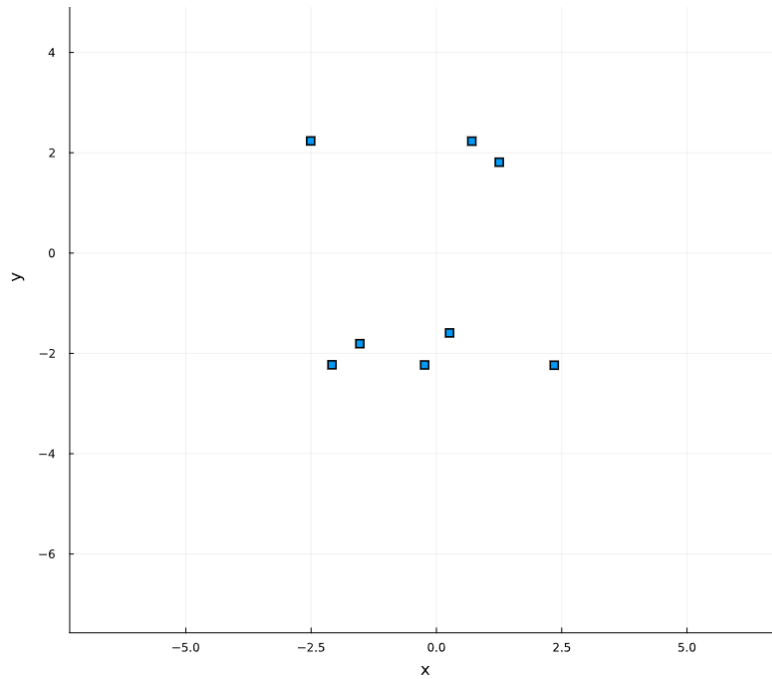


Figure 1: Critical Points illustrates the identified critical points on the curve, represented by squares.

3.2.4 Intersect with Bounding Object

We intersect the curve with a bounding object, typically a sphere, to manage unbounded components and focus the analysis on a finite region.

```
function get_sphere_intersection_pts(W_additional, W_curve)
    fillme = multilin_solver_master_entry_point(W_curve,
        multilin_linears)
    W_temp = get_noninfinite_w_mult(fillme)

    merge!(W_sphere, W_temp)

    fillme = sphere_solver_master_entry_point(W_sphere, sp_config)
    get_noninfinite_w_mult_full!(W_additional, fillme)
end
```

The *get_sphere_intersection_pts* function intersects the curve with a bounding object, typically a sphere, to manage unbounded components and focus the analysis on a finite region. It solves the system for the intersection points with the sphere, then sorts and filters these points to ensure they are real and unique, which helps in maintaining a manageable and bounded analysis space.

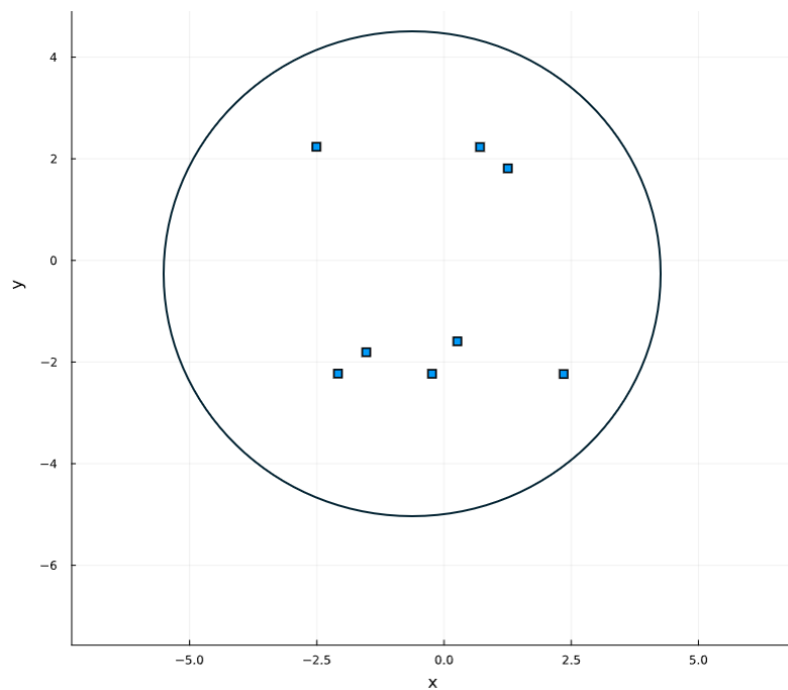


Figure 2: Intersections with Bounding Sphere shows how the curve intersects with a sphere, highlighting the bounded sections.

3.2.5 Slice at Projection Interval Midpoints

Slicing the curve at midpoints between critical points allows us to further segment the curve into manageable parts for detailed analysis.

```
function interslice(W_curve, W_crit_real, projections, V)
    compute_downstairs_crit_midpts!(V, W_canonicalized,
        crit_downstairs, mid_downstairs, projections[1])

    MidSlice!(edge_counter, midpoint_witness_sets, W_curve,
        particular_projection, mid_downstairs)

    ConnectTheDots!(indices_crit, found_indices_mid,
        found_indices_left, found_indices_right,
        crit_point_counter, V, crit_downstairs, mid_downstairs,
        particular_projection, midpoint_witness_sets)
end

function MidSlice!(edge_counter, midpoint_witness_sets, W_curve,
    particular_projection, mid_downstairs)
    for ii in 1:length(mid_downstairs)
        neg_mp!(particular_projection[1], mid_downstairs[ii])

        fillme = multilin_solver_master_entry_point(W_curve,
            particular_projection)
        midpoint_witness_sets[ii] = get_noninfinite_w(fillme)
        sort_for_real!(midpoint_witness_sets[ii])
        sort_for_unique!(midpoint_witness_sets[ii])
    end
end
```

The *interslice* function handles the intersection of critical points and projection midpoints, further segmenting the curve into manageable parts. It calculates the downstairs critical midpoints and uses a solver to find these midpoints on the curve. The function then connects these points using another solver, ensuring accurate and comprehensive segmentation of the curve. The points are sorted and filtered to ensure they are real and unique.

The *MidSlice* function slices the curve at the midpoints between critical points, allowing further segmentation for detailed analysis. It iterates over each midpoint, adjusts the projection values, and uses a solver to find new points at these midpoints on the curve. The resulting points are then sorted and filtered to retain only the real and unique midpoints, which are crucial for constructing the curve's segments.

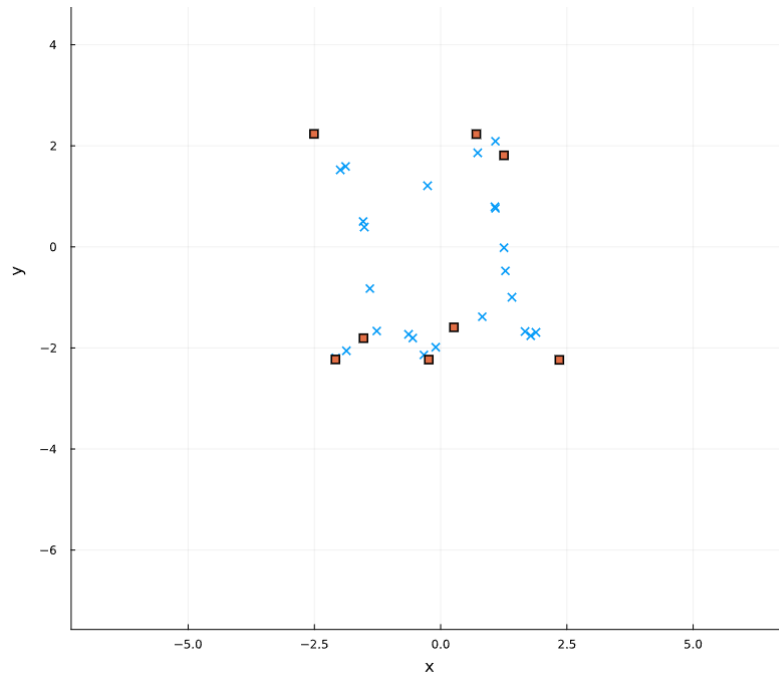


Figure 3: Curve Slicing displays the calculated midpoints, represented by x each.

3.2.6 Connect the Dots

This step involves drawing connections between the calculated points to form a preliminary geometric sketch of the curve.

```
function ConnectTheDots!(indices_crit, found_indices_mid,
found_indices_left, found_indices_right, crit_point_counter, V,
crit_downstairs, mid_downstairs, particular_projection,
midpoint_sets)
    for ii in 1:length(midpoint_witness_sets)
        neg_mp!(particular_projection[1], crit_downstairs[ii])

        fillme = multilin_solver_entry_point(midpoint_sets[ii],
        particular_projection)
        Wleft = get_noninfinite_w_mult_full(fillme)
```

```

neg_mp!(particular_projection[1],
crit_downstairs[ii + 1])
fillme = multilin_solver_master_point(midpoint_sets[ii],
particular_projection)
Wright = get_noninfinite_w_mult_full(fillme)

indices_crit[ii] = insert!(indices_crit[ii],
temp_edge.left)
indices_crit[ii+1]=insert!(indices_crit[ii+1],
temp_edge.right)
indices_mid[ii] = insert!(indices_mid[ii],
temp_edge.midpt)
end
end

```

The *ConnectTheDots* function involves drawing connections between the calculated points to form a preliminary geometric sketch of the curve. This function iterates through the midpoints and critical points, using a solver to connect these points and form the segments of the curve. The connections are verified and adjusted to ensure accuracy and completeness.

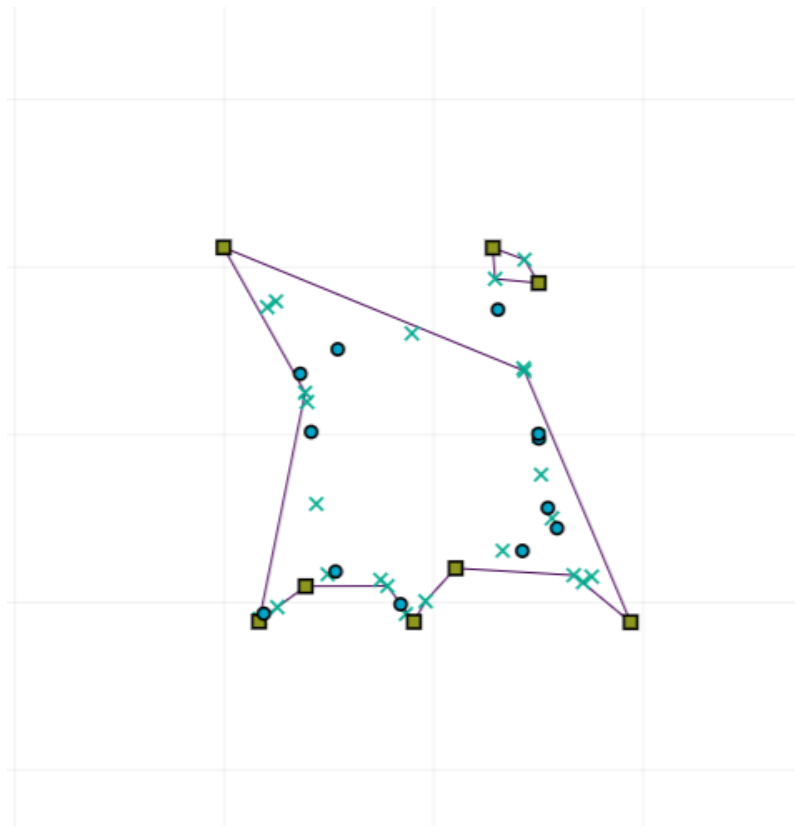


Figure 4: Connecting Points visually represents the connections between mid-points and critical points.

3.2.7 Merge

We simplify the curve by merging closely aligned or overlapping segments, reducing the overall complexity.

```

function GetMergeCandidates(V)
    default_found_edges = [-1]

    for tentative_right_edge in 1:length(edges)
        if V[edges[tentative_right_edge].left].type ==
            :New && V[edges[tentative_right_edge].right].type !=
            :New tentative_edge_list = [tentative_right_edge]

            while true
                tentative_left_edge =
                    nondegenerate_edge_w_right(
                        edges[tentative_edge_list[end]].left)
                push!(tentative_edge_list, tentative_left_edge)
                break if V[edges[tentative_left_edge].left].type !=
                    :New
            end
            if length(tentative_edge_list) > 1
                return tentative_edge_list
            end
        end
    end
    return default_found_edges
end

function Merge!(W_midpt, V, projections)
    edges_to_merge = GetMergeCandidates(V)

    while edges_to_merge[end] != -1
        rightmost_edge = edges_to_merge[1]
        leftmost_edge = edges_to_merge[end]

        W_temp = WitnessSet()
        projection_value_homogeneous_input!(particular_projection[1],
            V[edges[moving_edge].midpt].point, projections[1])
        neg_mp!(particular_projection[1], particular_projection[1])

        W_midpt.add_point(V[edges[moving_edge].midpt].point)
        fillme = multilin_solver_master_entry_point(W_midpt,
            particular_projection)
        W_temp = get_noninfinite_w_mult_full(fillme)

        temp_vertex = Vertex(W_temp.point[1], :Midpoint)

        temp_edge = Edge()
        temp_edge.left = edges[leftmost_edge].left
        temp_edge.midpt = index_in_vertices_with_add!(V,
            temp_vertex)
        temp_edge.right = edges[rightmost_edge].right

        AddEdge!(temp_edge, EdgeMetaData(edges_to_merge, md))

        edges_to_merge = GetMergeCandidates(V)
    end
end

```

The *GetMergeCandidates* function identifies candidates for merging by looking for edges that have points of type New on the left and points of a different type on the right. It iterates through the edges to find such candidates, creating a list of edges that can potentially be merged. If a suitable list of edges is found, it returns this list; otherwise, it returns a default value indicating no candidates were found.

The *Merge* function simplifies the curve by merging closely aligned or overlapping segments. For each candidate, it adjusts the projection values and uses a solver to find new points that represent the merged edges. A new edge is created from the merged points, and the original edges are updated or removed as necessary. This process continues until no more merge candidates are found, ensuring that the curve is simplified, and the overall complexity is reduced.

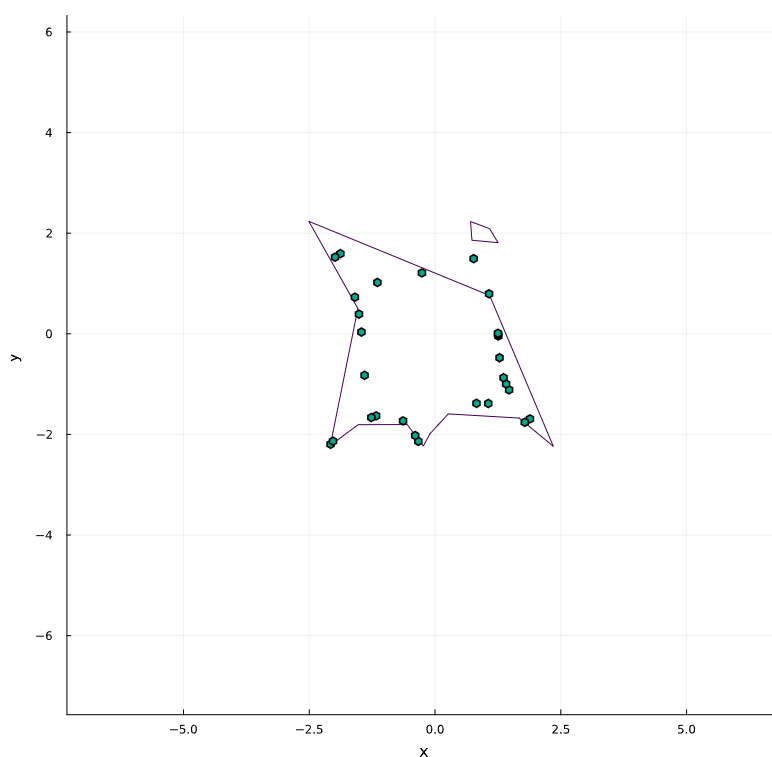


Figure 5: Merged Curve Segments demonstrates the curve after merging operations, showing a simplified structure, the removed points are represented by circles.

3.2.8 Sample

The final refinement involves sampling points along the curve to prepare for further analysis or visualization.

```
function initialize_projection(witness_set, pi, num_variables,
    num_natural_variables)
    target_projection = zeros(Complex{BigFloat}, num_variables)
    target_projection[1] = pi[1]

    # Adjust the size of the projection if necessary
```

```

if length(target_projection) < num_variables
    target_projection = vcat(target_projection,
        zeros(Complex{BigFloat}, num_variables -
            length(target_projection)))
end

# Add the projection to WitnessSet
add_linear!(witness_set, target_projection)
end

function sample_edge(ii, V, sampler_options, target_num_samples)
    interval_width = V[get_edge(ii).right()].projection_values()
    [V.curr_projection()] -
        V[get_edge(ii).left()].projection_values()
    [V.curr_projection()]

    pi_left =
        V[get_edge(ii).left()].projection_values() [V.curr_projection()]
    unit_width = 1.0 / (target_num_samples - 1)
    u_proj_values = [i * unit_width for i in 0:target_num_samples-1]

    for jj in 1:target_num_samples-2
        if jj * 2 == target_num_samples - 1
            sample_indices[ii][jj] = get_edge(ii).midpt()
            continue
        end

        left_cycle_num, right_cycle_num = get_cycle_nums(
            sampler_options, ii)
        target_projection = scale_by_cycle_num(u_proj_values[jj],
            left_cycle_num, right_cycle_num) * interval_width + pi_left

        fillme = SolverOutput()
        multilin_solver_master_entry_point(W, fillme,
            target_projection, ml_config)

        Wnew = get_noninfinite_w_mult_full(fillme)

        temp_vertex = Vertex(Wnew.points[1], Curve_sample_point)

        if sampler_options.no_duplicates
            sample_indices[ii][jj] = index_in_vertices_with_add(V,
                temp_vertex)
        else
            sample_indices[ii][jj] = add_vertex!(V, temp_vertex)
        end

        reset!(Wnew)
    end
end

```

The *initialize_projection* function initializes the target projection for a given witness set. It sets the initial projection based on the provided projection array `pi` and adjusts the size if necessary by filling with zeros to match the number of variables. This adjusted projection is then added to the witness set.

The *sample_edge* function samples points along an edge of the curve. It calculates the interval width and sets the initial projection values. Then, it iterates over each intermediate sample, adjusting the projection based on the cycle numbers. Using a solver, it finds new points and adds them to the Vertex Set. This process ensures

that the curve is adequately sampled for further analysis or visualization.

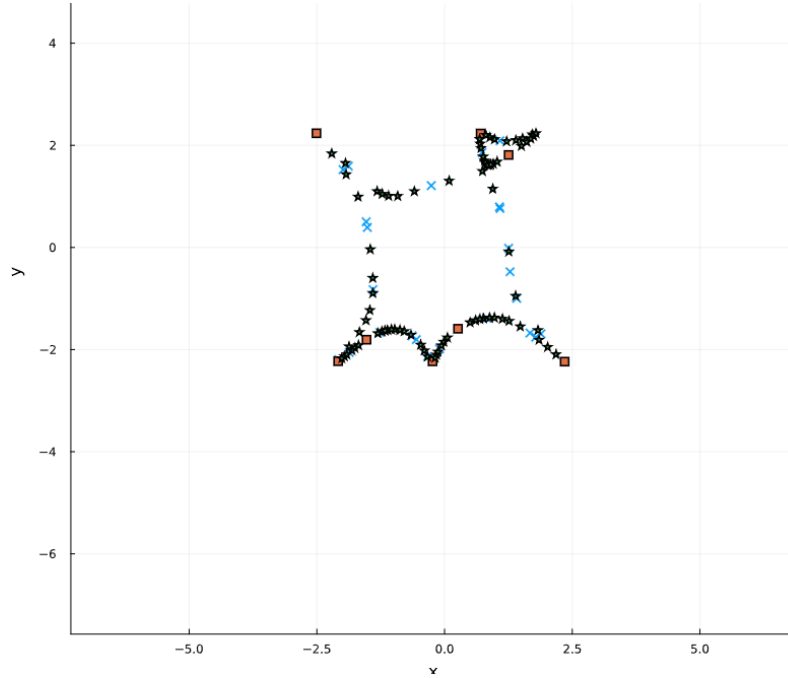


Figure 6: Sampled Curve shows the curve with sampled points, ready for detailed analysis or rendering. the sampled points are represented by stars.

3.2.9 Graphical Plotting

The Graphical Plotting component of our Julia module is specifically designed to visually present the results of the decomposition process. This feature enables users to see clear and insightful visual representations of algebraic curves. It can be utilized effectively to display outputs immediately following the execution of the six steps implemented in Julia, as well as to visualize results directly from BertiniReal outputs. Continuing with the example discussed in the previous section, the graphical representation of the example curve, after undergoing the decomposition process, would appear as follows:



Figure 7: Final Decomposed Curve showcases the fully processed curve, each color illustrating each segment found in the curve decomposition.

3.3. The Julia Package

This section provides a comprehensive guide on how to effectively utilize the Julia module for decomposing algebraic curves. It includes details on installation, configuration, and execution of the module, ensuring users can leverage its capabilities.

3.3.1 Installation

To install the Julia package once it is published, it can be added like any standard Julia package via the Julia package manager. This can be done using the following commands in the Julia REPL:

```
using Pkg
Pkg.add("HomotopyGetsReal")
```

Alternatively, if you prefer to use the development version, you can download it from the repository and include it manually in your Julia environment:

```
include("HomotopyGetsReal/src/HomotopyGetsReal.jl")
using .HomotopyGetsRealModule
```


This method allows you to use the latest features and updates that might not be available in the released version.

3.3.2 Usage

To utilize the module for decomposing algebraic curves, follow these steps:

1. **Set Up the Environment:** Specify the path where Bertini input files are located and where witness data generated by Bertini will be stored. This is also where output files such as graphs can be saved.

```
path = "YourFolderPath/Dir_Name"  
directory, _, _ = parse_directory_name(path)
```

2. **Define the Curve:** Use the module to define the curve, which will process the input files and execute the six steps previously described to create a decomposed curve object.

```
curve = Curve(directory)
```

3. **Visualize the Curve:** If you wish to graphically display the decomposed curve, use the plotting function provided by the module.

```
plotbr(curve)
```

This function will generate a visualization of the curve based on the decompositions performed, providing a graphical representation of the results.

4 Validation

We focus on validating the results produced by the Julia package for decomposing algebraic curves, ensuring accuracy and reliability by comparing these results with those obtained by BertiniReal. We have selected three specific curves for thorough testing and comparison.

The three curves selected for validation represent a range of complexities and characteristics that are typical in algebraic curve analysis. These specific curves were chosen for the following reasons:

- **Curve 1:** $(x^2 + y^2 - 1)^3 - 27x^2y^2 = 0$

This curve represents a relatively simple algebraic structure with a moderate number of critical points. It is useful for testing the basic functionality of the decomposition algorithms and ensuring that the package can handle standard algebraic curves.

- **Curve 2:** $(x^3 - xy^2 + y + 1)^2 \cdot (x^2 + y^2 - 1) + y^2 - 5 = 0$

This curve introduces additional complexity with multiple intersections and cusps. It is selected to test the robustness of the package in handling more intricate structures and to verify that the critical points and edges are accurately identified.

- **Curve 3:** $x^8 - 28x^6y^2 + 70x^4y^4 - 28x^2y^6 + y^8 + 15x^4y^2 - 15x^2y^4 = 0$

This highly complex curve, with numerous critical points and vertices, represents the upper limit of the package's capabilities. It is chosen to evaluate the performance and precision of the Julia package in handling large and complicated algebraic curves.

These curves collectively cover a spectrum from simple to highly complex, providing a comprehensive validation of the package's capabilities.

4.1. Validating Results

We will present the results in tables that contrast the outcomes from the Julia package with those from BertiniReal, providing a clear, side-by-side numerical comparison for each curve.

The following metrics will be compared for each curve:

- **Number of Critical Points:** Counts how many critical points each system identifies.
- **Average Distance Between Corresponding Critical Points:** Calculates the average Euclidean distance between pairs of corresponding critical points identified by each system.
- **Number of Vertices:** Indicates the total vertices generated by the decomposition.
- **Number of Edges:** Reflects the total edges formed in the decomposed structure.

Before diving into the comparative analysis, it is essential to note the computational environments and numerical precision capabilities of the tools being compared. BertiniReal is capable of handling calculations with up to 190 decimal places of precision, which allows for extremely fine-grained analysis and highly accurate results. In contrast, the Julia package, optimized for speed and general use, performs calculations with up to 12 decimal places. This difference in precision levels should be considered when evaluating the results.

Curve 1: The distance between corresponding critical points is approximately $7.993605777301127 \times 10^{-12}$ in precision.

Metric	BertiniReal	Julia Package
Number of Critical Points	6	6
Number of Vertices	32	30
Number of Edges	20	20

Table 4-1: Comparison of numerical results for Curve 1

Curve 2: The distance between corresponding critical points is approximately $9.331203109204507 \times 10^{-12}$ in precision.

Metric	BertiniReal	Julia Package
Number of Critical Points	8	8
Number of Vertices	44	41
Number of Edges	28	28

Table 4-2: Comparison of numerical results for Curve 2

Curve 3: The distance between corresponding critical points is approximately $1.24483884524882432 \times 10^{-10}$ in precision.

Metric	BertiniReal	Julia Package
Number of Critical Points	25	25
Number of Vertices	223	219
Number of Edges	125	125

Table 4-3: Comparison of numerical results for Curve 3

The tables presented above provide a comprehensive quantitative assessment of the decomposition capabilities of the Julia package compared to BertiniReal. The close match in the number of critical points identified and the minimal average distance between corresponding critical points underscore the precision and reliability of the Julia package. These results demonstrate that, despite the difference in numerical precision capabilities, the Julia package performs robustly and accurately in decomposing algebraic curves.

4.2. Visualization of Results

For each of the three curves tested, we provide graphical representations illustrating the results from the Julia package before and after the process of sampling. In these graphics, squares represent the critical points found during the decomposition process. Different colors are used to distinguish between the various edges, aiding in the visual differentiation of the geometric structures. Through all graphs, the curve in the left corresponds to the curve calculated with BertiniReal, while the curve in the right is the curve calculated with our Julia implementation

Curve 1:

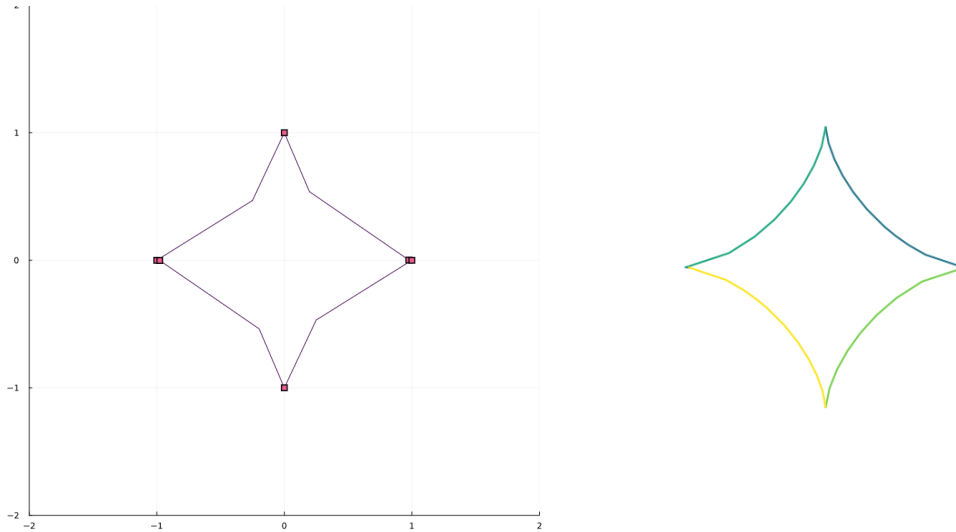
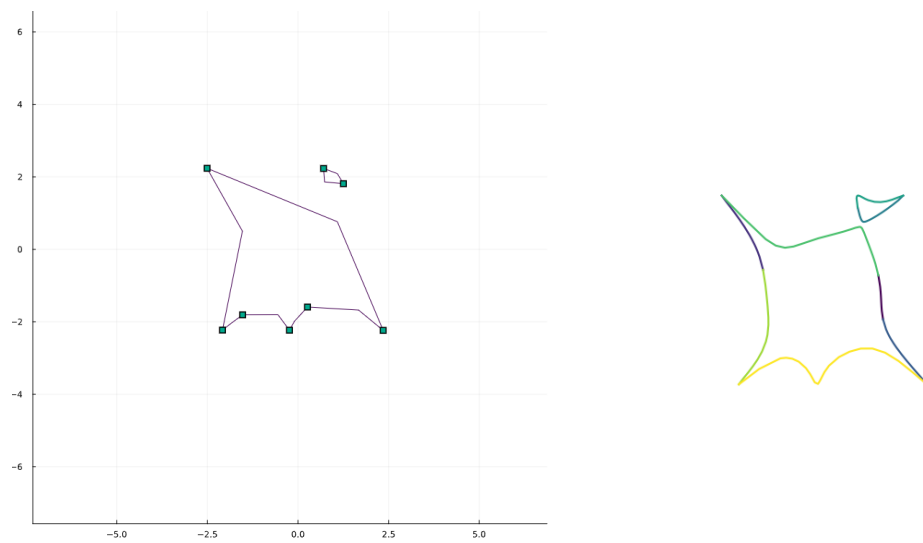
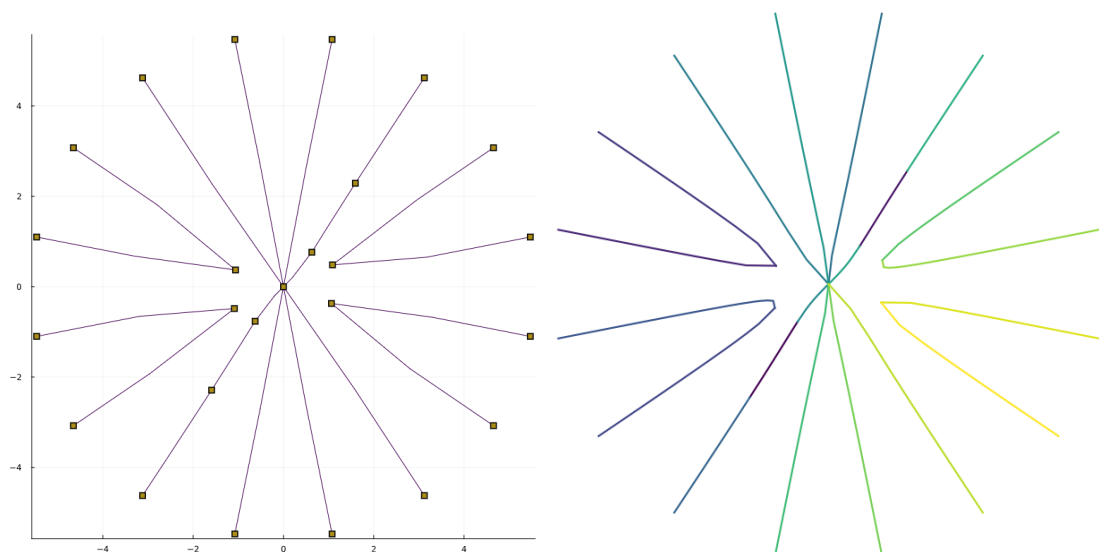


Figure 8: Decomposed Curve 1 The graphics of the curve 1.

Curve 2:**Figure 9: Decomposed Curve 2** The graphics of the curve 2.**Curve 3:****Figure 9: Decomposed Curve 3** The graphics of the curve 3.

5 Conclusions and Future Work

5.1. Conclusions

This thesis has successfully implemented a Julia package for the decomposition of algebraic curves, demonstrating the utility of Julia for such tasks. The project showcased Julia's capabilities for performing precise and rapid calculations, which are complemented by its functionalities for graphical representations. This makes the developed package not only a powerful computational tool but also a useful resource for visualizing and understanding algebraic curves.

Several challenges were encountered during the integration of high-level mathematical theories with practical software applications. Overcoming these challenges not only enhanced the tool's functionality but also deepened the comprehension of computational geometry's theoretical and applied aspects. Moreover, this thesis provides a comparative analysis with existing software solutions, acknowledging and drawing inspiration from the pioneering work of other developers. This acknowledgment extends to the generous guidance provided by these developers, which significantly influenced the project's trajectory and success.

Additionally, the thesis enhanced understanding of homotopies and homotopic methods, providing insights into their application in computational mathematics. It achieved a practical balance between abstract mathematical theories and their application in software, demonstrating how mathematical concepts can be effectively implemented and utilized in computational tools.

5.2. Future Work

The development and validation of the Julia package for decomposing algebraic curves contributes to the computational algebraic geometric environment in Julia. However, there remains substantial work to be done to enhance the utility and robustness of this tool. Future directions for this project include:

- **Code Documentation and Publication:** An immediate step towards enhancing the usability and accessibility of the Julia package involves thoroughly commenting the codebase. Proper documentation is essential for future maintenance, user support, and eventual publication. This process will ensure that the package is understandable and usable by others within the community, facilitating contributions and improvements.

- **Integration with HomotopyContinuation.jl:** Currently, the package operates independently. A key development goal is to adjust its architecture to function seamlessly as a part of the existing HomotopyContinuation.jl ecosystem.
- **Pull Request to HomotopyContinuation.jl:** In line with the integration efforts, submitting a pull request to the HomotopyContinuation.jl repository will be pursued. This contribution aims to merge the developed functionalities with the main project, thereby enhancing the overall feature set of HomotopyContinuation.jl and ensuring that it benefits a broader user base.
- **Extension to Three-Dimensional Surface Decomposition:** The principal future work involves expanding the capabilities of the package to support the decomposition of three-dimensional surfaces. This expansion is anticipated to follow a methodological approach similar to the decomposition of curves, utilizing insights and methods within the literature and the proven implementations from Bertini Real. We aim to develop robust algorithms capable of handling more complex geometrical entities such as surfaces.

Bibliography

- [ABH⁺17] Silviana Amethyst, Daniel J. Bates, Wenrui Hao, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler, *Bertini real: Numerical decomposition of real algebraic curves and surfaces*.
- [ABH⁺24] Silviana Amethyst, Daniel J. Bates, Wenrui Hao, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler, *Bertinireal: Software for numerical decomposition of real algebraic sets*, 2024, Accessed: fecha de acceso.
- [BHSW23] Daniel J. Bates, Jonathan D. Hauenstein, Andrew J. Sommese, and Charles W. Wampler, *Bertini: Software for numerical algebraic geometry*, 2023, Version actualizada.
- [Bre93] G.E. Bredon, *Topology and geometry*, Graduate texts in mathematics, Springer Verlag, 1993.
- [BT24] Paul Breiding and Sascha Timme, *Homotopycontinuation.jl*, Julia Package, 2024.