On Certified Isotopic Approximation of Space Curves

by

Caglar Dogan

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Courant Institute of Mathematical Sciences
New York University
May, 2023

____

Professor Chee Yap

© Caglar Dogan

all rights reserved, 2023

Acknowledgements

This thesis would not be possible without the guidance of my thesis advisor, Professor Chee Yap. I am thankful to Professor Chee Yap for introducing me to the field of exact computational geometry under his mentorship, as well as for his continued support and critique through this research project.

I also want to thank Professor Gizem Kayar, who agreed to be my secondary reader for this thesis and has offered me valuable guidance throughout the process.

Moreover, I want to further acknowledge the direct efforts of Martin Suderland in bringing many of the ideas discussed here to reality. He has been an amazing guide and discussion partner throughout this research endeavor.

Lastly, I would like to thank all the current and past members of the NYU Exact Geometric Computation (EGC) Group. The theory and implementation developed by this group have served as a crucial resource for the research presented here.

Abstract

The approximation of implicitly defined curves or surfaces is a problem of interest for many fields. As a result, this problem has been explored using algebraic, geometric, and numerical methods. Amongst these, a numerical method called Marching Cubes Algorithm ([4]) has been the primary choice in implementations because of its efficiency and implementability, even though a guarantee for topological correctness was not generally present.

Research in this area has largely focused on approximations of $n-1$ dimensional manifolds in n dimensional Euclidean space. These are called co-dimension 1 manifolds, defined as the zero sets of single equations in n variables. Plantinga and Vegter (2004) [8] derived the first algorithms with guaranteed topological correctness using interval arithmetic and adaptive subdivision for $n = 2, 3$. Faster variants of such algorithms were described by Yap et al. (2009, 2014) [10] [11]. Galehouse (2008) [9] succeeded in producing such algorithms for all n.

This thesis addresses the problem of computing isotopic approximations of co-dimension 2 manifolds, i.e., $n-2$ dimensional manifolds in n dimensional Euclidean space. Such manifolds are the intersection of the zero sets of two equations in n variables. The first interesting case is $n = 3$, i.e., the problem of computing an isotopic approximation of a space curve in 3D. We work on devising new algorithms by extending the previous interval techniques in co-dimension 1. Moreover, we implement and visualize such algorithms in order to verify their practical efficiency.

Contents

Acknowledgements **iii**

Abstract **iv**

List of Figures **viii**

List of Tables **ix**

1 Introduction **1**

1.1 Problem Statement 1

1.1.1 Assumptions 1

1.2 Relevant Background 2

1.2.1 Implicit Manifolds and Curves 2

1.2.2 Isotopy 2

1.2.3 Interval Arithmetic 3

1.2.4 Boxes and Box Functions: 5

1.3 Related Work 6

1.3.1 The Plantinga-Vegter (PV) Algorithm 6

1.3.2 Miranda Algorithm 7

2 Preliminaries and Theory **10**

2.1 Box Predicates 10
2.2 Jacobian Condition in Planar Curve Intersection 12
2.3 Geometric Consequence of the Jacobian Condition 14
2.4 The Miranda Conditions 17
2.5 A Miranda-Type Theorem for Space Curves 18
2.6 Preconditioning 22

3 Algorithm Design **25**
3.1 Curve-Isolating Subdivision Algorithm 25
3.2 Reconstruction Algorithm 29

4 Implementation and Experiments **32**
4.1 Implementation 32
4.2 Experiments and Results 34
4.2.1 Experiment 1 35
4.2.2 Experiment 2 38
4.2.3 Experiment 3 41
4.2.4 Experiment 4 44
4.2.5 Experiment 5 47

5 Conclusion **51**

A Appendix **52**
A.1 Main code for the implemented algorithms 52
A.2 Code for the cycle finding algorithm used in reconstruction 59
A.3 Code for the path finding algorithm used in reconstruction 60
A.4 Code for the implemented tests 61
A.5 Code defining intervals and interval arithmetic 68

A.6 Code defining generic boxes and functions on boxes 80

List of Figures

1.1 The Miranda algorithm as presented in [13] 8
1.2 The MK test, as presented in [13] 9

4.1 The output for experiment 1 seen in 3D 36
4.2 The output for experiment 1 viewed from the front, the side, and top 37
4.3 The output for experiment 2 seen in 3D 39
4.4 The output for experiment 2 viewed from the front, the side, and top 40
4.5 The output for experiment 3 seen in 3D 42
4.6 The output for experiment 3 viewed from the front, the side, and top 43
4.7 The output for experiment 4 seen in 3D 45
4.8 The output for experiment 4 viewed from the front, the side, and top 46
4.9 The output for experiment 5 seen in 3D, from two different angles 49
4.10 The output for experiment 5 viewed from the front, the side, and top 50

List of Tables

4.1 Experiment 1 - Phase 1 35
4.2 Experiment 1 - Phase 2 36
4.3 Experiment 2 - Phase 1 38
4.4 Experiment 2 - Phase 2 39
4.5 Experiment 3 - Phase 1 41
4.6 Experiment 3 - Phase 2 42
4.7 Experiment 4 - Phase 1 44
4.8 Experiment 4 - Phase 2 45
4.9 Experiment 5 - Phase 1 48
4.10 Experiment 5 - Phase 2 49

1 Introduction

1.1 Problem Statement

We present work on the problem of constructing an ambient isotopic estimation S' to a space curve S defined by two implicit surfaces given by $f_1 = 0$ and $f_2 = 0$ in a bounding box $B_0 \subseteq \mathbb{R}^3$ where functions $f_1, f_2 : \mathbb{R}^3 \to \mathbb{R}$ are smooth and non-singular in B_0. We use the notation $\boldsymbol{f} = (f_1, f_2)$ - for which the traced curve can be defined as the set of points given by $S = \boldsymbol{f}^{-1}(\boldsymbol{0}) = \{\boldsymbol{p} \in \mathbb{R}^3, \boldsymbol{f}(\boldsymbol{p}) = \boldsymbol{0}\} = f_1^{-1}(0) \cap f_2^{-1}(0)$.

An important result of non-singularity in B_0 is the truth of the following statement:

$$\forall \boldsymbol{p} \in B_0.\ \ (f_1(\boldsymbol{p}) = 0 \implies \nabla f_1(\boldsymbol{p}) \neq 0) \wedge (f_2(\boldsymbol{p}) = 0 \implies \nabla f_2(\boldsymbol{p}) \neq 0)$$

This result, alongside the smoothness condition, plays a central role in our arguments for the guaranteed halting of our algorithm steps, as it is presented in part 3 of this report.

1.1.1 Assumptions

1. We assume that $S = \boldsymbol{f}^{-1}(\boldsymbol{0})$ is a 1-dimensional curve.

2. We assume that box functions (point convergent interval functions) for $f_1, f_2, \nabla f_1, \nabla f_2$ exist and are denoted by $\square f_1, \square f_2, \square \nabla f_1, \square \nabla f_2$.

3. We assume that the surfaces given by $f_1 = 0$ and $f_2 = 0$ never intersect tangentially in B_0. $(\forall \boldsymbol{p} \in B_0, \forall c \in R.\ \ f_1(\boldsymbol{p}) = f_2(\boldsymbol{p}) = 0 \implies \nabla f_1(\boldsymbol{p}) \neq c\nabla f_2(\boldsymbol{p}))$

4. We assume that the traced curve and the implicit surfaces given by $f_1 = 0$ and $f_2 = 0$ intersect the bounding box B_0 transversally or not at all.

The softening or removal of our 4th assumption can presumably be achieved by adequately processing the boundary of B_0 in future research. Steps to ensure the prevention of problems of similar origin in the tracing of curves in 2D space have been previously presented by Lin and Yap ([10])

1.2 Relevant Background

Our work on this problem builds upon concepts from differential geometry and interval arithmetic. The following sections (1.2.1-1.2.4) can be seen for the explanation of core constructs and principles utilized by our approach.

1.2.1 Implicit Manifolds and Curves

For arbitrary $m, n \in \mathbb{Z}^+$, an implicit manifold is a manifold defined as the set of points $\boldsymbol{p} \in \mathbb{R}^m$ satisfying the system of equations $\boldsymbol{f}(\boldsymbol{p}) = (f_1(\boldsymbol{p}), f_2(\boldsymbol{p}), ..., f_n(\boldsymbol{p})) = 0^n = \mathbf{0}$ where $f_1, f_2, ..., f_n$ each map $\mathbb{R}^m$ to $\mathbb{R}$. An implicit curve is an implicit manifold with topological dimension 1.

1.2.2 Isotopy

Two manifolds $S, S' \subseteq \mathbb{R}^3$ are **ambient isotopic** if there exists a continuous map

$$\gamma : [0, 1] \times \mathbb{R}^3 \to \mathbb{R}^3$$

such that:

1. For each $t \in [0, 1]$, the map $\gamma_t : \mathbb{R}^3 \to \mathbb{R}^3$ given by $\gamma_t(p) = \gamma(t, p)$ is a homeomorphism;
2. γ_0 is the identity function;
3. $\gamma_1(S) = S'$.

Such a map γ is also called an **ambient isotopy** from S to S'. It is an ε**-ambient isotopy** if, in addition, we have $\|\gamma_0(p) - \gamma_1(p)\| \leqslant \varepsilon$ for all $p \in S$. Note that ε-ambient isotopy of S and S' implies that the Hausdorff distance between S and S' is $\leqslant \varepsilon$.

If the continuous map

$$\gamma : [0, 1] \times S \to \mathbb{R}^3$$

satisfies (i)-(iii), then we say that γ is an **isotopy** from S to S'; we also say S is **isotopic** to S'. Clearly, ambient isotopy implies isotopy, which in turn implies homeomorphism. The difference between ambient isotopy and (plain) isotopy is that the former requires a simultaneous transformation of the complementary space $\mathbb{R}^3 \setminus S$. But Hirsch [[2]] shows that, conversely, an isotopy can be extended to an ambient isotopy in case S is a smooth manifold.

1.2.3 Interval Arithmetic

Interval arithmetic is a tool to derive information about the range of functions over their domain. For this, operations are carried out on intervals rather than single values to achieve bounds on the global maximum and minimum of the function over the evaluated domain.

More formally, interval arithmetic and its methods are described by Ratschek and Ronke [3] as follows:

Let $X \subseteq \mathbb{R}$ be an arbitrary compact interval (i.e. $X = [a, b]$ such that $a, b \in R$ and $a \leq b$) and let $f : X \to \mathbb{R}$ be a **continuous** function. Let us also denote the range of f on its domain as $\bar{f}(X) = \{f(x)|x \in X\}$.

Then, one can define inner ($\Psi(X)$) and outer ($F(X)$) estimations of $\bar{f}(X)$ as estimations satisfying the following criterion:

$$\Psi(X) \subseteq \bar{f}(X) \subseteq F(X)$$

We are particularly interested in functions outputting outer estimations ($F(X)$), which are known to always exist for continuous f as denoted by Moore (1966) [1]. These functions are often referred to as inclusion functions ([3]) and will be referred to as such here.

These inclusion functions are also often directly computable, specifically for rational functions. An outer bound $F(X)$ for $\bar{f}(X)$ (for a continuous rational function $f : X \to \mathbb{R}, X \in I$ where I is the set of compact intervals in R) can be directly calculated by replacing the variables in an arithmetic expression of f with the domains of the respective variables and using interval arithmetic operations defined as follows ([3]):

$$[a, b] + [c, d] = [a + c, b + d]$$

$$[a, b] - [c, d] = [a - d, b - c]$$

$$[a, b][c, d] = [min(ac, ad, bc, bd), max(ac, ad, bc, bd)]$$

$$[a, b]/[c, d] = [a, b][1/b, 1/c] \text{ if } 0 \notin [c, d]$$

This provides a direct way to evaluate functions over intervals in algorithms. However, the exact defining arithmetic expression used in this evaluation affects the tightness and convergence

of the calculated outer bound ($F(X)$), and thus, a good representation should be utilized for the efficient use of interval arithmetic. The standard-centered form and Krawczyk's form are two commonly used expression forms with particularly good convergence rates. The formulations of these forms, as well as the further properties of interval arithmetic operations, can be seen in the work of Ratschek and Ronke [3] but are not explicitly discussed here.

1.2.4 Boxes and Box Functions:

Building upon interval arithmetic but abstracting away the specific evaluated arithmetic expressions of interval functions, the work discussed in this thesis research builds upon the properties of two constructs: **boxes** and **box functions**

Boxes: An n-dimensional box B is defined as a Cartesian product of n compact intervals denoted as $B = I_1 \times I_2 \times ...I_n$. One important point of interest for such a box is the mid-point of the box denoted by $m(B)$, which has the coordinates $(m(I_1), m(I_2), ...m(I_n))$.

The evaluation of a function f with n interval arguments on such a box is then defined with the following equality:

$$f(B) = f(I_1, ..., I_n)$$

Box Functions: A inclusion function $\square f$ is called a box function for f if, in addition to being an inclusion function, it is a point convergent, i.e., for any strictly decreasing sequence $B_0 \subset B_1 \subset ...$ of boxes that converges to a point p, we have $\square f(B_i) \rightarrow f(p)$ as $i \rightarrow \infty$. ([10])

These are the main mathematical constructs of interest in our current research and allow us to discuss methods utilizing arbitrary implementations of box functions.

1.3 Related Work

Our work builds upon previous research on the certified isotopic approximations of co-dimension 1 manifolds and root isolation algorithms. Amongst the algorithms produced by previous studies, the following algorithms serve as the main foundations of our approach:

1.3.1 The Plantinga-Vegter (PV) Algorithm ([8])

Let $F : \mathbb{R}^2 \mapsto \mathbb{R}$ be an implicit function, and $B \subseteq \mathbb{R}^2$ be a square bounding box. Also, let $\square g$ denote a convergent inclusion function for g for all functions g. The 2D PV algorithm defines a procedure that generates a guaranteed topologically correct piecewise linear estimation for $S = F^{-1}(0)$ where S is a regular curve (0 is a regular value of F as the gradient ∇F is non-zero at every point of the curve.).

This procedure defined by the algorithm starts by initializing a quadtree T on this box (initially with only one node), and subdivides the boxes until either a predicate ensuring the discarding of the box ($0 \notin F(C)$) or a stopping condition ($\langle \square\nabla F(C), \square\nabla F(C)\rangle > 0$) is satisfied where C is the two-dimensional interval defining the box. This result is then refined to make the subdivision quadtree balanced (ensuring that boxes that are adjacent are the same size or have a 1/2 ratio in between).

The termination of this process is certain under the assumptions on the inputs (as proven by Long Lin & Chee Yap in [10]). Furthermore, the inner product constraint puts an upper limit of $\pi/2$ radians for the angles between the gradients of F in any terminal box and furthermore implies the local parameterizability of the implicit curve in either the x or y direction.

This local parameterizability implies that S intersects at most two edges of each cell C and

that there cannot be any self-intersections. With this, within any such cell C, if there are two intersection points along the edges, the part of the curve inside C can be seen to be isotopic to a line segment. And thus, an approximation of the implicit curve S can be constructed by linear lines between the centers of box edges with different signs that will be isotopic to S, showing the correctness of the algorithm with the given restrictions. Moreover, even if any given box has more than two intersection points along the edges, it can be shown that the produced result is still globally isotopic to the original curve even though it might not be isotopic to the approximation in each box.

This method is particularly powerful, as it forgoes local isotopy (at each subdivided box) to be able to produce a globally isotopic approximation while being guaranteed to halt.

The 3D PV algorithm builds upon this base, providing a mesh construction method in 3D to give certified isotopic approximations of surfaces in 3D.

1.3.2 Miranda Algorithm ([13])

Given an initial bounding box B_0, the Miranda algorithm provides guaranteed isolation of simple zeros of a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ for some $n \in \mathbb{Z}^+$ in B_0. For this, an existence test EC (proving the existence of at least one root in a box) and a Jacobian test JC (proving the existence of at most one root in a box) are utilized alongside an exclusion predicate defined as $C_0(B) = \mathbf{0} \notin \boldsymbol{f}(B))$ with the following algorithm:

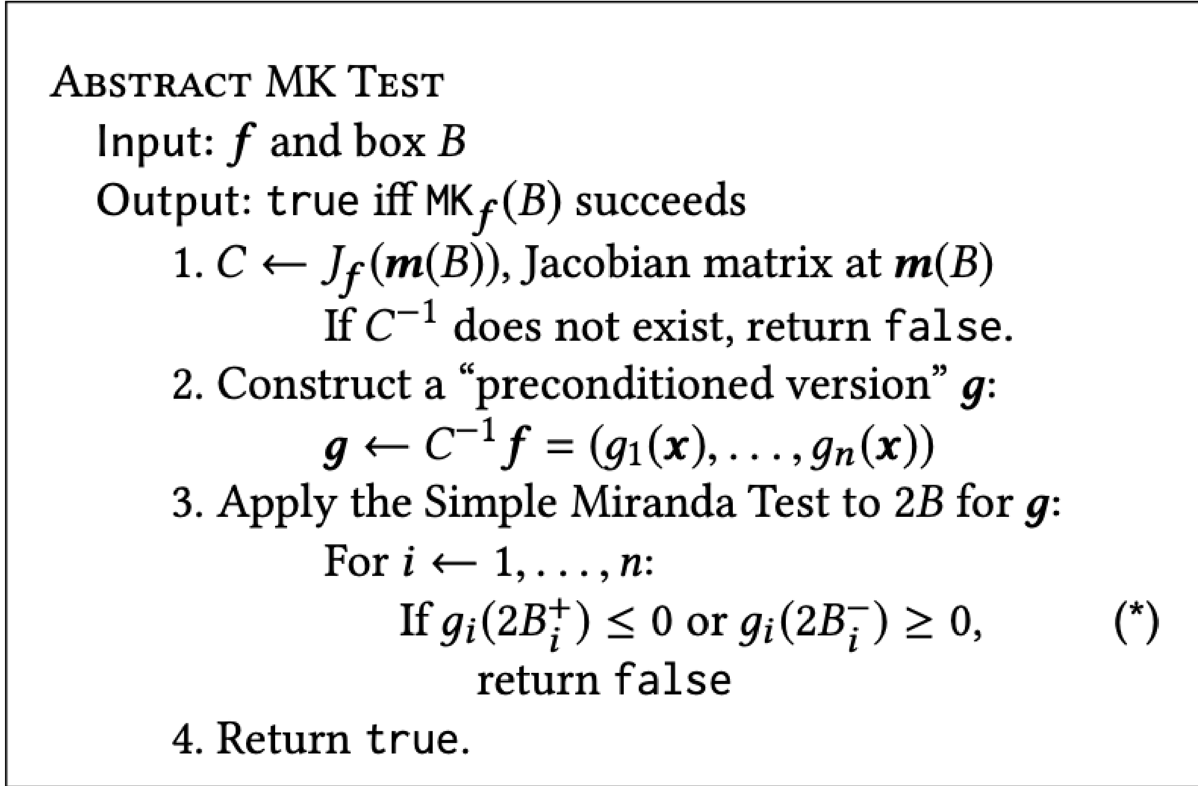
```
SIMPLE ISOLATE(f, B_0)
  Output: sequence of isolating boxes for roots in B_0
  Q ← {B_0}
  While Q ≠ ∅
      B ← Q.pop()
      If C_0(B) continue;          ◂ discard B and repeat loop
      If EC(B) ∧ JC(B)             ◂ B has a unique root
          output B and continue;
      If w(N(B) ∩ B) < w(B)        ◂ if contraction succeeds
          Q.push(B)
      else
          Q.push(subdivide(B))
```

Figure 1.1: The Miranda algorithm as presented in [13]

This algorithm is guaranteed to produce correct outputs and can be shown to always halt for the proposed EC and JC tests.

The proposed EC test (the MK Test) is of particular interest to us, as it is a predicate proving the existence of a root of f when it holds. This test utilizes a preconditioning phase (in which the range of f is multiplied by the inverse Jacobian matrix of f evaluated at the midpoint of B. Then, the value of $J_f(m(B))^{-1}f_i$ is evaluated at the opposite ends of the evaluation box in the i'th dimension. If all such evaluations show that f_i is negative towards the negative direction in the i'th axis and positive on the opposite side, the test is said to succeed as this proves the existence of at least one root in the box (as an extension of the Poincare-Miranda Theorem, as explained in [13]).

This test is formally presented as follows in the [13] paper:



ABSTRACT MK TEST

Input: $\boldsymbol{f}$ and box B

Output: `true` iff $\mathsf{MK}_{\boldsymbol{f}}(B)$ succeeds

1. $C \leftarrow J_{\boldsymbol{f}}(\boldsymbol{m}(B))$, Jacobian matrix at $\boldsymbol{m}(B)$
   If C^{-1} does not exist, return `false`.
2. Construct a "preconditioned version" $\boldsymbol{g}$:
   $\boldsymbol{g} \leftarrow C^{-1}\boldsymbol{f} = (g_1(\boldsymbol{x}), \ldots, g_n(\boldsymbol{x}))$
3. Apply the Simple Miranda Test to $2B$ for $\boldsymbol{g}$:
   For $\boldsymbol{i} \leftarrow 1, \ldots, \boldsymbol{n}$:
   If $g_i(2B_i^+) \leq 0$ or $g_i(2B_i^-) \geq 0$, (*)
   return `false`
4. Return `true`.

Figure 1.2: The MK test, as presented in [13]

A numerical version of this abstract test proven to work under the constraints of finite-precision arithmetic is also provided here, which serves as a foundational tool utilized by our proposed approach.

2 Preliminaries and Theory

2.1 Box Predicates

We introduce several predicates (also called "conditions") on boxes $B \subseteq \mathbb{R}^3$:

- (Exclusion Condition C_0)

  The condition $C_0(B) = C_0^{\mathbf{f}}(B)$ holds iff $C_0^{f_1}(B) \vee C_0^{f_2}(B)$ where

  $$C_0^{f_j}(B) \equiv \left[0 \notin f_j(B)\right] \quad (j = 1, 2). \tag{2.1}$$

  If $C_0^{\mathbf{f}}(B)$ holds, then the curve $\mathbf{f}^{-1}(\mathbf{0})$ does not intersect B (so B can be excluded in our search).

- (Inclusion Condition C_1)

  We write $D_i f$ for the function $\frac{\partial f}{\partial x_i}$ $(i = 1, 2.3)$. The condition $C_1(B) = C_1^{\mathbf{f}}(B)$ holds iff $C_1^{f_1}(B) \wedge C_1^{f_2}(B)$ where

  $$C_1^{f_j}(B) \equiv \left[0 \notin \sum_{i=1}^{3} (D_i f_j(B))^2\right] \tag{2.2}$$

  Note that the expression of the right is the natural set extension of the usual arithmetic operations, with $S^2 := \{st : s, t \in S\}$ (not $S^2 = \left\{s^2 : s \in S\right\}$). Condition $C_1(B)$ implies that the angle between the gradients at any two points in B is at most 90 degrees; this implies that B does not contain any any closed surface of $f_1^{-1}(0)$ or $f_2^{-1}(0)$. However B may still

contain a closed loop of $f_1^{-1}(0) \cap f_2^{-1}(0) = \mathbf{f}^{-1}(\mathbf{0})$. The next condition will address this.

- (Jacobian Conditions JC_i)

  First let

$$D\mathbf{f}(\mathbf{x}) := \begin{bmatrix} (f_1)_x(\mathbf{x}) & (f_1)_y(\mathbf{x}) & (f_1)_z(\mathbf{x}) \\ (f_2)_x(\mathbf{x}) & (f_2)_y(\mathbf{x}) & (f_2)_z(\mathbf{x}) \end{bmatrix} \tag{2.3}$$

  denote a 2×3 matrix where $(f_1)_i = \frac{\partial f_1}{\partial x_i}$ and similarly for $(f_2)_i$ $(i = x, y, z)$. The Jacobian condition at a point $\mathbf{p} \subseteq \mathbb{R}^3$ is when the matrix $D\mathbf{f}$ is full-rank[1] when evaluated at $\mathbf{x} := \mathbf{p}$.

  Next, we define the corresponding condition for a box B. Let $\Delta_i \mathbf{f} : \mathbb{R}^3 \to \mathbb{R}$ $(i = 1, 2, 3)$ denote the determinant of the 2×2 matrix obtained by deleting the ith column of $D\mathbf{f}(\mathbf{x})$, multiplied by $(-1)^{i+1}$. Thus

$$\Delta_1 \mathbf{f} := \det \begin{bmatrix} (f_1)_y(\mathbf{x}) & (f_1)_z(\mathbf{x}) \\ (f_2)_y(\mathbf{x}) & (f_2)_z(\mathbf{x}) \end{bmatrix}, \ \Delta_2 \mathbf{f} := -\det \begin{bmatrix} (f_1)_x(\mathbf{x}) & (f_1)_z(\mathbf{x}) \\ (f_2)_x(\mathbf{x}) & (f_2)_z(\mathbf{x}) \end{bmatrix}, \ \Delta_3 \mathbf{f} := \det \begin{bmatrix} (f_1)_x(\mathbf{x}) & (f_1)_y(\mathbf{x}) \\ (f_2)_x(\mathbf{x}) & (f_2)_y(\mathbf{x}) \end{bmatrix}. \tag{2.4}$$

  The **Jacobian condition** at B is defined by

$$JC^{\mathbf{f}}(B) \equiv \Big[\bigvee_{i=1}^{3} JC_i^{\mathbf{f}}(B) \Big] \tag{2.5}$$

  where

$$JC_i^{\mathbf{f}}(B) \equiv \Big[0 \notin \Delta_i \mathbf{f}(B) \Big] \qquad (i = 1, 2, 3). \tag{2.6}$$

  Thus $JC^{\mathbf{f}}(B)$ implies that $D\mathbf{f}(\mathbf{p})$ is full-rank for each $\mathbf{p} \in B$. But the main geometric conclusion is seen in Lemma 2.3 in the subsequent sections.

[1]For algebraic curves, a birational correspondence between an irreducible algebraic space curve C and an irreducible plane curve $\mathcal{P}$ with the same genus as C depends on a similar full-rank condition (see [5]).

2.2 Jacobian Condition in Planar Curve Intersection

Temporarily, let us define a pair of planar curves $\mathbf{f} = (f, g) : \mathbb{R}^2 \to \mathbb{R}^2$. We prove the following lemma (to be used in subsequent sections):

Lemma 2.1 (2D Jacobian Condition)**.** *Let $B \subseteq \mathbb{R}^2$ and $JC_3^{\mathbf{f}}(B)$ holds, i.e.,*

$$0 \notin J_{\mathbf{f}}(B) := \det \begin{bmatrix} D_x f(B) & D_y f(B) \\ D_x g(B) & D_y g(B) \end{bmatrix}. \tag{2.7}$$

Then $|Zero(f, g) \cap B| \leqslant 1$.

Proof. Let $\mathbf{a}, \mathbf{b} \in B$ be two distinct zeros of $\mathbf{f}$. Define $L : \mathbb{R} \to \mathbb{R}^2$ where $L(t) = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$. Consider the function $F(t) := f(L(t))$: by an application of the Chain Rule, we have

$$F'(t) = \frac{dF}{dt} = \nabla f(L(t) * (\mathbf{b} - \mathbf{a}) \tag{2.8}$$

where $*$ denotes dot product. From $F(0) = f(L(0)) = f(\mathbf{a}) = 0$ and $F(1) = f(L(1)) = f(\mathbf{b}) = 0$, the Mean Value Theorem (MVT) for $F(t)$ implies that there exists $\xi \in [0, 1]$ such that

$$0 = F(0) - F(1) = F'(\xi) = \nabla f(L(\xi) * (\mathbf{b} - \mathbf{a}). \tag{2.9}$$

Similarly, if we define $G(t) := g(L(t))$, there exists $\nu \in [0, 1]$ such that

$$0 = G(0) - G(1) = G'(\xi) = \nabla g(L(\xi) * (\mathbf{b} - \mathbf{a}). \tag{2.10}$$

But 2.9 implies that $\nabla f(L(\xi))$ is perpendicular to $\mathbf{b} - \mathbf{a}$. Similarly, $\nabla g(L(\nu))$ is perpendicular to

$\mathbf{b} - \mathbf{a}$. Thus $\nabla f(L(\xi))$ and $\nabla g(L(\nu))$ are parallel, i.e.,

$$0 = \det \begin{bmatrix} D_x f(L(\xi)) & D_y f(L(\xi)) \\ D_x g(L(\nu)) & D_y g(L(\nu)) \end{bmatrix}.$$

Since $L(\xi), L(\nu) \in B$, this implies

$$0 \in \det \begin{bmatrix} D_x f(B) & D_y f(B) \\ D_x g(B) & D_y g(B) \end{bmatrix},$$

which contradicts 2.7. □

2.3 Geometric Consequence of the Jacobian Condition

The Jacobian condition on a box $B \subseteq \mathbb{R}^3$ has geometric consequences that can be derived from the Inverse Function Theorem (IFT) of mathematical analysis [7]. Here is a version from [12] which is suitable for our needs:

Suppose we have a system of m functions in $m + n$ variables,

$$\mathbf{f} = (f_1, ..., f_m) : \Omega \subseteq \mathbb{R}^m \times \mathbb{R}^n \longrightarrow \mathbb{R}^m \tag{2.11}$$

where Ω is an open set of $\mathbb{R}^m \times \mathbb{R}^n$ and each $f_i = f_i(\mathbf{x}; \mathbf{y})$ is a function in the real variables $\mathbf{x} = (x_1, ..., x_m)$ and $\mathbf{y} = (y_1, ..., y_n)$. Assume $\mathbf{f}$ is C^1, for which we can define:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \left(\frac{\partial f_i}{\partial x_j}\right)_{i=1,j=1}^{m,m} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \cdots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_m} \end{bmatrix}$$

$$\frac{\partial \mathbf{f}}{\partial \mathbf{y}} = \left(\frac{\partial f_i}{\partial y_k}\right)_{i=1,k=1}^{m,n} = \begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \cdots & \frac{\partial f_1}{\partial y_n} \\ \vdots & \cdots & \vdots \\ \frac{\partial f_m}{\partial y_1} & \cdots & \frac{\partial f_m}{\partial y_n} \end{bmatrix}$$

Theorem 2.2 (The Implicit Function Theorem)**.**
Given $\mathbf{f}$ *as in* 2.11*, let* $(\mathbf{a}; \mathbf{b}) \in \Omega \subseteq \mathbb{R}^m \times \mathbb{R}^n$ *satisfy*

$$\mathbf{f}(\mathbf{a}; \mathbf{b}) = \mathbf{0} \textit{ and } \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{a}; \mathbf{b}) \textit{ is invertible.}$$

Then there exists an open set $X \subseteq \mathbb{R}^m$ *containing* $\mathbf{a}$*, an open set* $Y \subseteq \mathbb{R}^n$ *containing* $\mathbf{b}$*, and an (implicit) function* $\mathbf{g} : Y \longrightarrow X$ *such that:*

- *For each* $\mathbf{y}^* \in Y$*, there is a unique* $\mathbf{x}^* \in X$ *such that* $\mathbf{f}(\mathbf{x}^*, \mathbf{y}^*) = \mathbf{0}$*. In fact,* $\mathbf{x}^* = \mathbf{g}(\mathbf{y}^*)$*. It follows that* $\mathbf{g}(\mathbf{b}) = \mathbf{a}$.
- *The function* $\mathbf{g} : Y \longrightarrow X$ *is* C^1 *with Jacobian matrix* $J\mathbf{g}$ *at any* $\mathbf{y}^* \in Y$ *given by*

$$J\mathbf{g}(\mathbf{y}^*) = -\left[\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{g}(\mathbf{y}^*), \mathbf{y}^*)\right]^{-1}_{m\times m} \cdot \left[\frac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{g}(\mathbf{y}^*), \mathbf{y}^*)\right]_{m\times n}$$

See [12, Theorem 2] for a proof.

An immediate consequence of the IFT theorem is this:

Lemma 2.3 (Implicit Function under Jacobian Condition)**.**
Let the Jacobian condition $JC_3^{\mathbf{f}}(B)$ *hold. If* B *contains a point* $\mathbf{p} = (p_1, p_2, p_3)$ *of the curve* $\mathbf{f}^{-1}(\mathbf{0})$ *then there exists an implicit* C^1 *function*

$$\mathbf{g} : J_{\mathbf{p}} \longrightarrow \mathbb{R}^2 \tag{2.12}$$

where $J_{\mathbf{p}}$ *is an open interval containing* p_3 *satisfying*

(i) $\mathbf{g}(p_3) = (p_1, p_2)$.

(i) For all $z \in J_{\mathbf{p}}$,

$$\mathbf{f}(\mathbf{g}(z), z) = \mathbf{0}.$$

Proof. The condition $JC_3^{\mathbf{f}}(B)$ implies that $\Delta_3\mathbf{f} = \det\begin{bmatrix}(f_1)_x(\mathbf{x}) & (f_1)_y(\mathbf{x})\\ (f_2)_x(\mathbf{x}) & (f_2)_y(\mathbf{x})\end{bmatrix}$ is non-zero at $\mathbf{x} = \mathbf{p}$ (see 2.4). To apply Theorem 2.2 (IFT Theorem), let $m = 1$, $n = 2$, $\mathbf{f} = (f_1, f_2)$, $\mathbf{x} = (x, y)$ and $\mathbf{y} = (z)$. If $(\mathbf{a}; \mathbf{b}) = (p_1, p_2; p_3) = \mathbf{p}$, then the hypothesis of IFT Theorem is satisfied, namely

$$\mathbf{f}(\mathbf{a}; \mathbf{b}) = \mathbf{0}, \quad \text{and} \quad \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{p}) = \Delta_3\mathbf{f}(\mathbf{p}) \neq 0.$$

We conclude that there exists an implicit function $\mathbf{g} : X \to Y$ where $X \subseteq \mathbb{R}^2$ is an open set containing (p_1, p_2), $Y \subseteq \mathbb{R}$ is an open interval containing p_3, and for all $\mathbf{y} \in Y$, $\mathbf{f}(\mathbf{g}(\mathbf{y}); \mathbf{y}) = \mathbf{0}$. The theorem follows if we rename Y to be $J_{\mathbf{p}}$, and view the range of $\mathbf{g}$ to be $\mathbb{R}^2$. □

In other words, this lemma tells us that the set

$$\{(\mathbf{g}(z), z) : z \in J_{\mathbf{p}}\}$$

(viewed as the graph of the function $\mathbf{g}$) is a parameterization of the curve $\mathbf{f}^{-1}(\mathbf{0})$ in some neighborhood of $\mathbf{p}$. Next, if $\mathbf{g}_i : J_i \to Y_i$ $(i \in I)$ is a collection of such graphs with the property that $J_* := \cup_{i \in I} J_i$ is a connected interval. Then we can define a unique function $\mathbf{g}_* : J^* \to \mathbb{R}^2$ where $\mathbf{g}_*(\mathbf{p}) = \mathbf{g}_i(\mathbf{p})$ for all $\mathbf{p} \in J_*$.

Lemma 2.4 (No Loop under the Jacobian Condition)**.**
If $JC^{\mathbf{f}}(B)$ holds, then B does not contain a closed curve (i.e., loop) of $\mathbf{f}^{-1}(\mathbf{0})$.

Proof. Consider a box B such that $JC^{\mathbf{f}}(B)$ holds. Without loss of generality, assume that $JC_3^{\mathbf{f}}(B)$ holds. Then, Suppose B contains a loop $C \subseteq \mathbf{f}^{-1}(\mathbf{0})$. Pick a point $\mathbf{p} = (p_1, p_2, p_3) \in C$ where p_3 is maximum. Note that such a point exists since C is contained in B. By Lemma 2.3, and $JC_3^{\mathbf{f}}(B)$, there must be an open interval containing p_3 in which C is parameterizable by the third coordinate axis. This contradicts p_3 being the maximum of the third coordinates amongst the points in C when C is non-singular. □

2.4 The Miranda Conditions

So far, we have given three conditions: $C_0(B), C_1(B), JC(B)$. They all amount to the exclusion of $\mathbf{0}$ from various algebraic expressions evaluated on the box B. The next one is slightly different, and may be called "Miranda conditions".

We temporarily consider the general setting of an n-dimensional axes-parallel box $B \subseteq \mathbb{R}^n$: let B_i^- and B_i^+ (for $i = 1, \ldots, n$) denote the pair of opposite facets[2] that are normal to the ith axis. Moreover, if $\Pi_i(\mathbf{x}) = x_i$ denote the projection to the ith coordinate, then assume that $x_i^- < x_i^+$ where $\Pi_i(B_i^-) = \{x_i^-\}$ and $\Pi_i(B_i^+) = \{x_i^+\}$. If $\mathbf{f} = (f_1, \ldots, f_n) : \mathbb{R}^n \to \mathbb{R}^n$, the following box predicate

$$MT^{\mathbf{f}}(B) \quad \equiv \quad \left[\bigwedge\nolimits_{i=1}^{n} (f_i(B_i^- < 0 < f_i(B_i^+))\right] \tag{2.13}$$

was called the "simple Miranda test" in [13]. The Miranda Theorem (1940) says that if $MT^{\mathbf{f}}(B)$ holds, then $B \cap \mathbf{f}^{-1}(\mathbf{0})$ is non-empty (e.g., [6]). But in our co-dimension one setting, the number of functions is not n, but $n-1$. If $\mathbf{f} = (f_1, \ldots, f_{n-1})$, then we define the **Miranda condition** $MK(B) = MK^{\mathbf{f}}(B)$ as

$$MK^{\mathbf{f}}(B) \equiv \left[\bigvee_{i=1}^{n} MK_i^{\mathbf{f}}(B)\right]$$

where $MK_i(B) = MK_i^{\mathbf{f}}(B)$ is given by

$$MK_i^{\mathbf{f}}(B) \equiv \left[\bigwedge_{j=1}^{i-1} (f_j(B_j^-) < 0 < f_j(B_j^+))\right] \wedge \left[\bigwedge_{j=i+1}^{n} (f_{j-1}(B_j^-) < 0 < f_{j-1}(B_j^+))\right]. \tag{2.14}$$

Note that $MK_i(B)$ places no restrictions on the pair of faces B_i^- and B_i^+.

[2] I.e., $(n-1)$-dimensional faces of B.

2.5 A Miranda-Type Theorem for Space Curves

For $n = 3$, and $\mathbf{f} = (f_1, f_2)$, $MK_3^{\mathbf{f}}(B)$ is simply

$$MK_3(B) = (f_1(B_1^-) < 0 < f_1(B_1^+)) \wedge (f_2(B_2^-) < 0 < f_2(B_2^+))$$

Now, we present the following theorem:

Theorem 2.5 (Miranda Theorem for Curves)**.**
Let $B \subseteq \mathbb{R}^3$ and the following conditions hold:

$$C_1^{\mathbf{f}}(B) \wedge JC_3^{\mathbf{f}}(B) \wedge MK_3^{\mathbf{f}}(B). \tag{2.15}$$

Then $Zero(f, g) \cap B$ is comprised of a single curve component with endpoints in B_3^- and B_3^+, respectively.

If the given theorem is true, the local curves in a set of boxes B in an initial box B_0 satisfying $C_1^{\mathbf{f}}(B) \wedge JC_3^{\mathbf{f}}(B) \wedge MK_3^{\mathbf{f}}(B)$ can be "stitched together" to curve components in B_0, allowing for an approximation of the traced curve components.

Now, let us consider the following lemma:

Lemma 2.6 (Existence of a curve)**.** *Assume that a box B satisfies $JC_3(B) \wedge MK_3(B)$, then there exists a curve $H : I \longrightarrow \mathbb{R}^3$ within $\mathbf{f}^{-1}(\mathbf{0})$, which connects the faces B_z^- and B_z^+.*

Proof. Because of the standard 2-dimensional Miranda theorem, we know that there exists a point $\mathbf{p} \in B_z^+$ such that $\mathbf{f}(\mathbf{p}) = \mathbf{0}$. By Lemma 2.2, there exists an open interval $J_{\mathbf{p}}$ containing p_3, and a differentiable function

$$\mathbf{h}_{\mathbf{p}} : J_{\mathbf{p}} \to \mathbb{R}^2 \tag{2.16}$$

such that $\mathbf{h_p}(p_3) = (p_1, p_2)$ and, for all $z \in J_\mathbf{p}$,

$$\mathbf{f}(\mathbf{h_p}(z), z) = \mathbf{0}.$$

Among all open intervals $J_\mathbf{p}$ with that property, let $J = (a, b)$ be one, which minimizes a. Note that the curve $H : J \to \mathbb{R}^3, z \mapsto \mathbf{f}(\mathbf{h_p}(z), z)$ cannot intersect any of B's faces except for B_z^- and B_z^+ since $MK_3(B)$ is satisfied. If $a < \Pi_3(B_z^-)$ then the curve H connects the opposite faces B_z^- and B_z^+ as desired. We will now show that $a \geq \Pi_3(B_z^-)$ leads to a contradiction. We again use the Lemma 2.2 for $\mathbf{q} = (\mathbf{h}(a), a)$ to derive an open interval $J_\mathbf{q}$ together with a differentiable function $\mathbf{h_q}$ similar as before. But this is a contradiction to the maximality of interval J because also the longer interval $J \cup J_\mathbf{q}$ could have been chosen together with the function $\mathbf{h}$:

$$\mathbf{h} : J \cup J_\mathbf{q}, z \mapsto \begin{cases} \mathbf{h_p}(z) & \text{if } z > a \\ \mathbf{h_q}(z) & \text{otherwise} \end{cases} \tag{2.17}$$

Note that also $\mathbf{h}$ is differentiable because both $\mathbf{h_p}$ and $\mathbf{h_q}$ are so and the domains (z-components) and images overlap. □

Building upon this, we now present a proof of the Miranda Theorem for Curves (Theorem 2.5):

Proof. Without loss of generality, consider a box $B = [-1, 1]^3$ satisfying $C_1^\mathbf{f}(B) \wedge JC_3^\mathbf{f}(B) \wedge MK_3^\mathbf{f}(B)$.

For such a box B, the faces B_z^+ and B_z^- lie in the planes $z = 1$ and $z = -1$.

By the standard Miranda theorem applied to pair of functions $(f(x, y, 1), g(x, y, 1))$, we can see that there is a point $\mathbf{p}_1 = (p_1, p_2, 1) \in B_z^+$ such that $\mathbf{f}(\mathbf{p}_1) = \mathbf{0}$. By applying the quantitative IFT, we can construct the graph of a function $g_1 : [1-\mu, 1+\mu] \to \mathbb{R}^2$ such that $\{(g_1(z), z) : z \in [1 - \mu, 1 + \mu]\}$ is a parameterization of $Zero(\mathbf{f})$ around $\mathbf{p}_1$. Then, we can choose a point $\mathbf{p}_2 = (g_1(1-\mu), 1-\mu) \in B$

on the curve. This point is in B and thus we can apply the theorem again to get another function

$$g_2 : [1-\mu, 1+\mu] \to \mathbb{R}^2$$

whose graph is a parameterization of $Zero(\mathbf{f})$ around the point $\mathbf{p}_2$. Continuing in this way, we can construct a curve through the points $\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_k$ (for some $k \geqslant 1$) until the curve reaches the face B_z^-.

Now, let Γ be such a curve.

As $MK_3(B)$ holds in B, it can trivially be seen that the curve Γ cannot be intersecting the pair of facets $B_1^{\pm}$ or the pair of faces $B_2^{\pm}$. $(\forall i \in \{1,2\}.\ \ f_i(B_1^-) < 0 < f_i(B_1^+) \implies 0 \notin f_i(B_1^-) \wedge 0 \notin f_i(B_1^-))$

This implies:

$$\forall p = (p_1, p_2, p_3) \in \Gamma.\ \ p_1, p_2, p_3 \in [-1, 1] \tag{2.18}$$

Moreover, as the curve Γ connects the faces B_z^+ and B_z^-, we have:

$$\forall z \in [-1,1].\ \ \exists x, y \in [-1,1].\ \ (x, y, z) \in \Gamma \tag{2.19}$$

We now claim that $Zero(\mathbf{f}) \cap B$ is contained in Γ, which proves our theorem. Suppose, for the sake of contradiction, that there is some point $\mathbf{p}' = (x', y', z')' \in (Zero(\mathbf{f}) \cap B) \setminus \Gamma$.

For such a $\mathbf{p}' = (x', y', z')$, we have $x', y', z' \in [-1, 1]$ by (2.18)

However, for the same z', we must also have: $\exists x, y \in R.\ \ (x, y, z') \in \Gamma$ by (2.19).
$((x, y) \neq (x', y')$ is known as $\mathbf{p}' \notin \Gamma)$

This implies that, for the box $B' = [[-1,1],[-1,1],z']$ we must have:

$$|Zero(f,g) \cap B'| \geqslant 2$$

As we have $\{(x,y,z),(x,y,z')\} \subseteq Zero(f,g) \cap B'$ and $(x,y) \neq (x',y')$

Moreover, as $B' \subseteq B$, we have $JC_3^{\mathbf{f}}(B')$ (as a result of $JC_3^{\mathbf{f}}(B)$).

However, by Lemma 2.1, we know $JC_3^{\mathbf{f}}(B') \implies |Zero(f,g) \cap B'| \leqslant 1$

Thus, we have $JC_3^{\mathbf{f}}(B') \wedge |Zero(f,g) \cap B'| \geqslant 2 \wedge (JC_3^{\mathbf{f}}(B') \implies |Zero(f,g) \cap B'| \leqslant 1)$, which is a contradiction.

Hence, our assumption that there exists $\mathbf{p'} = (x',y',z')' \in (Zero(\mathbf{f}) \cap B) \setminus \Gamma$ must be false, proving our claim that $Zero(\mathbf{f}) \cap B$ is contained in Γ.

This completes the proof of the Miranda Theorem for Curves. $\square$

2.6 Preconditioning

Unfortunately, the Miranda conditions discussed in the preceding sections are not guaranteed to hold as we consider smaller and smaller boxes around an arbitrary non-singular space curve. As discussed in [13], a preconditioning operation is needed to provide such guarantees.

To describe an adequate preconditioning operation, let us denote by e_i for $i \in \{1, 2, 3\}$ the unit vectors $(1, 0, 0)^T$, $(0, 1, 0)^T$, and $(0, 0, 1)^T$ respectively. Moreover, let us denote by δ_{ij} the Kronecker delta, i.e. $\delta_{ij} = 1$ if and only if $i = j$. Otherwise $\delta_{ij} = 0$. We use ∂_i for the partial derivative along the i-th coordinate direction. Denote for a matrix $M \in \mathbb{R}^{m \times n}$ by M_{ij} the i-th row and j-th column entry of M.

Then, the standard preconditioning of the MK uses the following transformation:

$$\forall p \in \mathbb{R}^3 : \quad \begin{pmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \end{pmatrix}_{(p)} = \begin{pmatrix} \nabla f_1^T \\ \nabla f_2^T \\ \nabla f_3^T \end{pmatrix}_{(m_B)}^{-1} \cdot \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}_{(p)} \tag{2.20}$$

Lemma 2.7 (Derivatives for the standard preconditioning)**.** *The standard preconditioning aligns the function space of $\tilde{f}_1$, $\tilde{f}_2$ and $\tilde{f}_3$ such that*

$$\forall i \in \{1, 2, 3\} : \quad \nabla \tilde{f}_i = e_i \tag{2.21}$$

Moreover every root of $\tilde{\mathbf{f}}$ is also a root of $\mathbf{f}$.

Proof. The second equation holds to linearity of derivatives:

$$\partial_i \tilde{f}_j(m) = e_j^T \begin{pmatrix} \partial_i \tilde{f}_1 \\ \partial_i \tilde{f}_2 \\ \partial_i \tilde{f}_3 \end{pmatrix}_{(p)}$$

$$= e_j^T \underbrace{\begin{pmatrix} \nabla f_1^T \\ \nabla f_2^T \\ \nabla f_3^T \end{pmatrix}_{(m_B)}^{-1} \cdot \begin{pmatrix} \partial_i f_1 \\ \partial_i f_2 \\ \partial_i f_3 \end{pmatrix}_{(p)}}_{e_i}$$

$$= \delta_{ij}$$

Therefore, $\nabla \tilde{f}_i = e_i$ as claimed.

The transformation matrix is invertible and therefore $\forall p \in \mathbb{R}^3$:

$$\begin{pmatrix} \tilde{f}_1 \\ \tilde{f}_2 \\ \tilde{f}_3 \end{pmatrix}_{(p)} = \begin{pmatrix} \nabla f_1^T \\ \nabla f_2^T \\ \nabla f_3^T \end{pmatrix}_{(m_B)}^{-1} \cdot \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}_{(p)} = 0 \quad \Leftrightarrow \quad \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}_{(p)} = 0$$

I.e. a root of $\tilde{\mathbf{f}}$ is also a root of $\mathbf{f}$. □

It has been proven that this preconditioning ensures the halting of the *MK* test in a small enough box around roots if $\mathbf{f}$ in [13]. Thus, this preconditioning can be used before the *MK* test in a subdivision algorithm searching for the roots of $\mathbf{f}$ (where boxes far from the roots are eventually discarded by $C_0^{\mathbf{f}}$) to guarantee eventual halting of the search.

Unfortunately the curve $\tilde{f}_1^{-1}(0) \cap \tilde{f}_2^{-1}(0)$ in transformed space does not directly tell us anything about the curve $f_1^{-1}(0) \cap f_2^{-1}(0)$ in primal space. As a result, preconditioned Miranda tests

cannot be directly combined with the Miranda Theorem for Curves (Theorem 2.5) for an isotopic approximation of the traced curve.

On the other hand, the information regarding the roots of $\mathbf{f}$ (given by information regarding the roots of $\tilde{\mathbf{f}}$) is valuable, as it can be seen by its use in our algorithm in the subsequent chapters.

3 | Algorithm Design

Here, we present two algorithms which, combined, allow us to approximate space curves. These algorithms build upon the tests and theoretical results provided in Chapter 2.

3.1 Curve-Isolating Subdivision Algorithm

Let us first define a subdivision of a box B_0 as a set $\bar{\mathbf{B}}$ of disjoint boxes for which $\bigcup_{B \in \bar{\mathbf{B}}} B = B_0$ holds. Let us consider the elements of such a $\bar{\mathbf{B}}$ connected if and only if the boxes share a face.

Let us also define the concept of an octree subdivision of a box B_0 as a subdivision of B_0 achievable by starting with the set $\bar{\mathbf{B}} = \{B_0\}$ and splitting an arbitrary element of $\bar{\mathbf{B}}$ to eight boxes sharing the same ratios of dimensions as the original box iteratively for an arbitrary number of iterations.

Moreover, for an arbitrary box $B \subseteq \mathbb{R}^3$, let $Faces(B)$ be the set of all 2D faces of B.

Now, for arbitrary box or box face B and real number $c \geq 1$, define $B.scale(c)$ as the box/box face sharing the same mid-point and proportions as B but having all interval widths multiplied by c.

For $\mathbf{f} = (f_1, f_2)$ implicitly defining a space curve and satisfying the presented assumptions on an initial bounded box $B_0 \subseteq \square\mathbb{R}^3$:

For all $face \in Faces(B)$ for some $B \subseteq \square\mathbb{R}^3$, consider the standard preconditioning defined as follows:

$$\tilde{\mathbf{f}} = \begin{pmatrix} \tilde{f}_1 \\ \tilde{f}_2 \end{pmatrix}_{(p)} = \begin{pmatrix} \nabla f_1^T \\ \nabla f_2^T \end{pmatrix}_{(m_{face})}^{-1} \cdot \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}_{(p)}$$

Then, consider the preconditioned test: $F_PMK_c^{\mathbf{f}}(face) \equiv MT^{\tilde{\mathbf{f}}}(face.scale(c))$

(Where MT is the simple Miranda test defined as in (2.13))

Such a test can be equivalently written as: $F_PMK_c^{\mathbf{f}}(face) \equiv$

$$\left[(\tilde{f}_1((face_1^-).scale(c)) < 0 < \tilde{f}_1((face_1^+).scale(c))) \wedge (\tilde{f}_2((face_2^-).scale(c)) < 0 < \tilde{f}_2((face_2^+).scale(c)))\right] \tag{3.1}$$

Then, define preconditioned MK test for boxes defined as follows:

$$PMK_c^{\mathbf{f}}(B) \equiv \left[\bigvee_{face \in faces(B)} F_PMK_c^{\mathbf{f}}(face.scale(c))\right] \tag{3.2}$$

Using the test $PMK_c^{\mathbf{f}}(B)$, alongside C_0, C_1, and JC defined in previous sections, we can define the following algorithm:

Algorithm 1 $Subdivide^{\mathbf{f}}(B_0, c, \epsilon)$

  Input: $B_0 \subseteq \mathbb{R}^3, c \geq 1, \epsilon \in \mathbb{R}^+ \cup \{+\infty\}$

  Output: A subset $\mathbf{B}$ of an octree subdivision of B_0

Ensure: $\forall B \in \mathbf{B}.\ \ (\neg C_0^{\mathbf{f}}(B)) \wedge (size(B) \leq \epsilon) \wedge C_1^{f_1}(B) \wedge C_1^{f_2}(B) \wedge JC^{\mathbf{f}}(B.scale(4)) \wedge PMK_c^{\mathbf{f}}(B)$

  $\mathbf{B} \leftarrow \emptyset$

  $Q \leftarrow \{B_0\}$

  while $Q \neq \emptyset$ **do**

    $B \leftarrow Q.pop()$

    if $\neg C_0^{\mathbf{f}}(B)$ **then** ▷ If B not excluded

      if $(size(B) \leq \epsilon) \wedge C_1^{f_1}(B) \wedge C_1^{f_2}(B) \wedge JC^{\mathbf{f}}(B.scale(4)) \wedge PMK_c^{\mathbf{f}}(B)$ **then**

        $\mathbf{B}.push(B)$ ▷ Subdivision halted for box B

      else

        $Q.push(B.split())$ ▷ Further subdivision required

      end if

    end if

  end while

Where Boolean expressions are evaluated using short-circuit evaluation. This ensures that each test in $(size(B) \leq \epsilon) \wedge C_1^{f_1}(B) \wedge C_1^{f_2}(B) \wedge JC^{\mathbf{f}}(B.scale(4)) \wedge PMK_c^{\mathbf{f}}(B)$ is only carried out if the preceding tests are true.

Note that as all $B \in \mathbf{B}$ satisfies $PMK_c^{\mathbf{f}}(B)$, we can see that there exists $face \in Faces(B.scale(c))$ such that the 2D MK test on $face.scale(c)$ succeeds.

By the Miranda theorem ([6]), this implies:

$$\forall B \in \mathbf{B}.\ \ \exists face \in Faces(B.scale(c)).\ \ \exists \mathbf{p} \in face.scale(c).\ \ \mathbf{f}(\mathbf{p}) = 0$$

which means that there is a point $\mathbf{p}$ close to the volume of box B which lies on the traced curve $S = B \cap \mathbf{f}^{-1}(\mathbf{0})$. This closeness is determined by c passed to the algorithm.

We also know that for the traced curve cannot be passing through any volume not covered by a $B \in \mathbf{B}$ as all boxes in such volumes must have been discarded by the C_0 (exclusion) predicate.

Moreover, as $B.scale(4)$ for all boxes $B \in \mathbf{B}$, we know by Lemma 2.4 that there are no closed loops of the traced curve in $B.scale(4)$ for all $B \in \mathbf{B}$.

Unfortunately, however, we have no proof of further properties regarding the traced curve's behavior in the outputted set of boxes $\mathbf{B}$ as the results of the Miranda Theorem for Curves (Theorem 2.5) do not directly imply any guarantees when a preconditioning step is used. (As explained in Section 2.6)

However, experimentally, $Subdivide^{\mathbf{f}}(B_0, c, \epsilon)$ has been observed to output a set of boxes $\mathbf{B}$ whose connected components $K_1, ..., K_m$ each cover exactly one of the connected components of the traced curve in B_0. (i.e. connected components of $\mathbf{B}$ isolate the connected components of the traced curve in B_0)

3.2 Reconstruction Algorithm

Let Q_{curve} be a set of boxes, outputted by $\mathit{Subdivide}^{\mathbf{f}}(B_0, c, \epsilon)$ for some $B_0 \subseteq \mathbb{R}^3$, $c > 1$, $\epsilon > 0$. Then, we present the following algorithm to find approximations for the connected components of the traced curve in B_0:

Algorithm 2 $\mathit{Reconstruct}(Q_{\text{curve}})$

Input: $Q_{\mathbf{curve}} \subseteq \square\mathbb{R}^3$

Output: An ϵ-approximation of the curve(s) in $\mathbf{f} = \mathbf{0}$

1: Create a directed graph G that contains:

2: A vertex for each box in Q_{curve}

3: A directed edge from B_1 to $B_2 \Leftrightarrow$ both boxes share a piece of an edge and the direction of the edge conforms with $JC(B_1)$ and $JC(B_2)$

4: Split the graph G into simple connected components $K_1, K_2, \ldots$

5:

6: **for** each component K_i **do**

7: **if** $\exists$ directed cycle in K_i with $\mathit{length} > 2$ **then**

8: Output *short(est)* directed cycle in K_i with $\mathit{length} > 2$

9: **else**

10: Output a path realizing the graph diameter of the undirected version of K_i

11: **end if**

12: **end for**

---

where the graph diameter for the undirected version K_i is determined with the following algorithm:

Algorithm 3 *Find_Diameter*(G)

Input: undirected graph $G = (V, E)$

Output: A path $v, \ldots, w$ realizing the diameter of G

1: Let $u \in V$

2: Use BFS to find a vertex v with maximum distance from u

3: Use BFS to find a vertex w with maximum distance from v

4: Return a shortest path from v to w

In the *Reconstruct* algorithm, A directed edge from B_1 to B_2 is considered to conform with $JC(B_1)$ and $JC(B_2)$ if and only if the vector pointing from the center of B_1 to the center of B_2 does not contradict the parameterization implied by the $JC(B_1)$ and $JC(B_2)$ conditions.

With this, the *Reconstruct* algorithm works by constructing directed graphs for each connected component K_i of the input set Q_{curve} where the direction of each edge represents a potential way the traced curve passes between the boxes in the connected component.

If such a graph for a connected component includes cycles with length greater than two, the *Reconstruct* algorithm returns the shortest of such cycles. While we currently do not posses theoretical guarantees for such an approximation, we have experimentally observed that such cycles have only been detected in the created graphs when the traced curve piece was a closed loop inside the related connected component (for which the smallest of such cycles presented a simple approximation).

If the graph for a connected component does not include any cycles with length greater than two, then we know that the traced curve must not have a cycle of length greater than

two in the connected component as the predicates used are conservative (and thus only ever over-approximate the possibilities for the real behavior of the curve). For such cases, the approximation of the traced curve piece by a path realizing the graph diameter has been experimentally a good approximation.

Here, it must be noted that the limit of two for cycle length is necessary as cycles of length two occur sporadically in the created graphs due to the way directed edges are created. While this has not caused the erroneous approximation of any traced curve components in our experiments, further work is needed in either providing guarantees or improving this method.

4 | Implementation and Experiments

4.1 Implementation

The algorithms described in Chapter 3 have been implemented in Matlab alongside the needed data structures, predicates, and interval arithmetic.

In this process, the *Subdivide* algorithm has been implemented with the *parfor* function of Matlab for parallelizing the subdivision of boxes at each depth as follows:

```
...

%Depth limit for phase 1
depthlimit = 8;
%Depth limit for phase 2
numiterMKlimit = 6;

...

%% Subdivision Phase 1
Q = B0; %Input of the first phase of subdivision
QJac = []; %Output of first phase of subdivision

% Depth for phase 1
depth = 0;

% Create a subdivision of boxes, which all satisfy the predicates untill C1 tests
% and the Jacobian tests hold (where boxes satisfying C0 get excluded at each level)
while ~isempty(Q) && depth <= depthlimit
    Q_next = cell(length(Q),1);
    QJac_add = cell(length(Q),1);

    disp(['Phase 1: depth = ', num2str(depth), ' | length(Q) = ', num2str(length(Q))]);

    %Parallel Subdivision
    parfor i = 1:length(Q)
        B_par = Q(i);
        if ~local_predicate.C0(B_par,f,1) && ~local_predicate.C0(B_par,g,2)
```

```
29              if B_par.radius<MAXEPS && local_predicate.C1(B_par,df,3) &&
                ↪  local_predicate.C1(B_par,dg,4) && ... %&&
                ↪  local_predicate.C1cross(B_par,df,dg,5)
30                 local_predicate.Jacobian(B_par,df,dg,5)
31                  QJac_add{i} = B_par;
32              else
33                  children = B_par.split;
34                  Q_next{i} = children;
35              end
36          end
37      end
38
39      accepted = [QJac_add{:}];
40
41      disp(['# accepted boxes = ', num2str(length(accepted))]);
42
43      %Collection of results
44      Q = [Q_next{:}];
45      QJac = [QJac, QJac_add{:}];
46
47      depth = depth+1;
48  end
49
50  if ~isempty(Q) && depth == depthlimit + 1
51      disp("Phase 1 stopped due to depth limit");
52  end
53
54  disp(['Time for Phase 1: ',num2str(toc(tStart)),'s']);
55
56  disp("Phase 1 finalized.");
57  disp("Proceeding to phase 2...");
58
59  tPhase2 = tic;
60
61  %% Subdivision Phase 2
62  QMK = QJac; %Input of the second phase of subdivision
63  Qcurve = []; %Output of second phase of subdivision
64  %%
65  %Depth for phase 2
66  numiterMK = 1;
67
68  while ~isempty(QMK) && numiterMK <= numiterMKlimit
69      %Iteration cell arrays
70      QMK_next = cell(length(QMK),1);
71      Qcurve_add = cell(length(QMK),1);
72
73      disp(['Phase 2: numiterMK = ', num2str(numiterMK), ' | length(QMK) = ',
        ↪  num2str(length(QMK))]);
74
75      %Parallel Subdivision
76      parfor i = 1:length(QMK)
77          B_par = QMK(i);
78          if ~local_predicate.C0(B_par,f,1) && ~local_predicate.C0(B_par,g,2)
79              if local_predicate.Jacobian(B_par,df,dg,5) &&
                ↪  local_predicate.MK_face(B_par,f,df,g,dg,7)
                ↪  %local_predicate.MK_face(B_par,f,df,g,dg,7)
80                  if any(B_par.testresults{7})
81                      Qcurve_add{i} = B_par;
82                 end %Else: MK test has succeeded not in finding a root, but in excluding all in
                    ↪  internal call to C0_faces
83              else
```

```
84                  children = B_par.split;
85                  QMK_next{i} = children;
86              end
87          end
88      end
89
90      accepted = [Qcurve_add{:}];
91
92      disp(['# accepted boxes = ', num2str(length(accepted))]);
93
94      %Collection of results
95      QMK = [QMK_next{:}];
96      Qcurve = [Qcurve, Qcurve_add{:}];
97
98      numiterMK = numiterMK+1;
99  end
```

As it can be seen, the subdivide algorithm has been split into two steps here to better understand the depth of subdivision required for all boxes to satisfy the JC condition and the additional depth of subdivision required to satisfy the MK test.

Then, the output of this subdivision step $Qcurve$ is used to create the required graph structures, and is used to find approximations for the traced curve pieces. (Appendix A.1 can be seen for the whole main code including these steps for Algorithm 2.)

The code presented in Appendix A can be seen for further details regarding these steps, as well as the implementations of the related functions and the main data structures.

4.2 Experiments and Results

In the following pages, the results for the experimental approximations of several space curves (each implicitly defined by two functions) can be seen.

When discussing the result, the first subdivision loop will be referred as Phase 1 and the second subdivision loop will be referred as Phase 2.

4.2.1 Experiment 1

Approximation of an implicit space curve defined by $\mathbf{f} = (f_1, f_2)$ given by:

$$f_1(x, y, z) = x^2 + y^2 - z$$

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

where inputs for $Subdivide^{\mathbf{f}}(B_0, c, \epsilon)$ are as follows

$$B_0 = [[-1, 1], [-1, 1], [-1, 1]]$$

$$c = 1.3$$

$$\epsilon = \infty$$

presents the following behavior:

Table 4.1: Experiment 1 - Phase 1

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	1	0
1	8	0
2	64	0
3	224	44
4	128	4
Time: 0.84525s		

Table 4.2: Experiment 1 - Phase 2

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	48	36
1	96	0
Time: 1.1794s		

The outputted curve approximation can then be seen drawn on top of the boxes used in the approximation in the following figures:

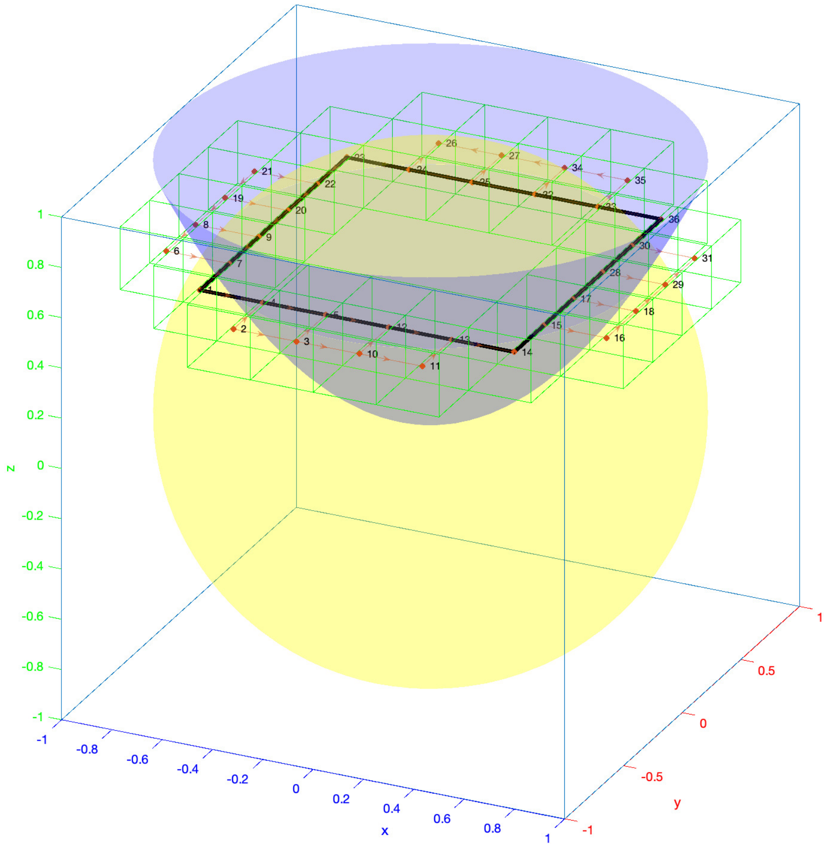
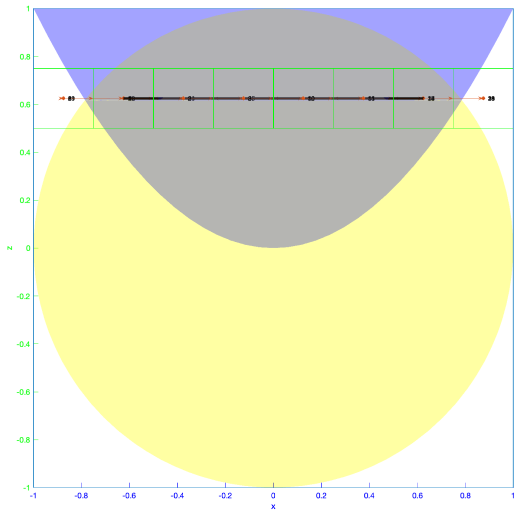
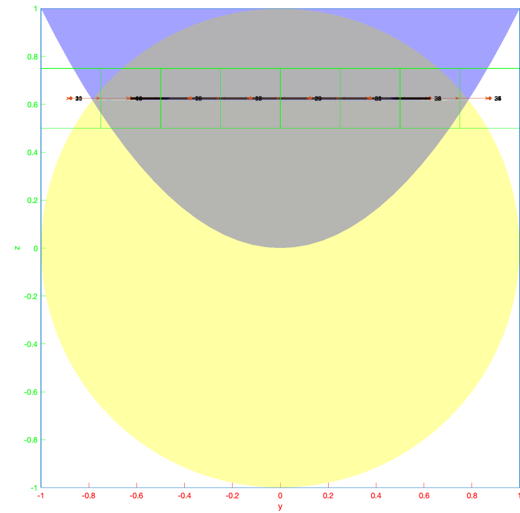


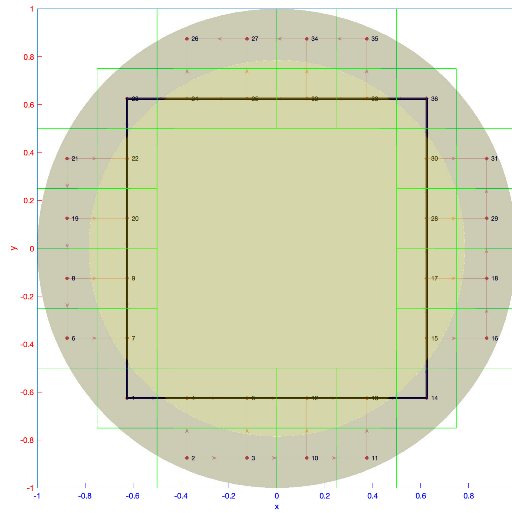
Figure 4.1: The output for experiment 1 seen in 3D



(a) Front view



(b) Side view



(c) Top view

Figure 4.2: The output for experiment 1 viewed from the front, the side, and top

4.2.2 Experiment 2

Approximation of an implicit space curve defined by $\mathbf{f} = (f_1, f_2)$ given by:

$$f_1(x, y, z) = x^2 + y^2 - z$$

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

where inputs for $Subdivide^{\mathbf{f}}(B_0, c, \epsilon)$ are as follows:

$$B_0 = [[-1, 1], [-1, 1], [-1, 1]]$$

$$c = 1.3$$

$$\epsilon = 0.05$$

presents the following behavior and results:

Table 4.3: Experiment 2 - Phase 1

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	1	0
1	8	0
2	64	0
3	224	0
4	480	0
5	928	0
6	1664	424
Time: 3.5449s		

Table 4.4: Experiment 2 - Phase 2

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	424	408
1	128	0
Time: 13.8679s		

The outputted curve approximation can then be seen drawn on top of the boxes used in the approximation in the following figures:

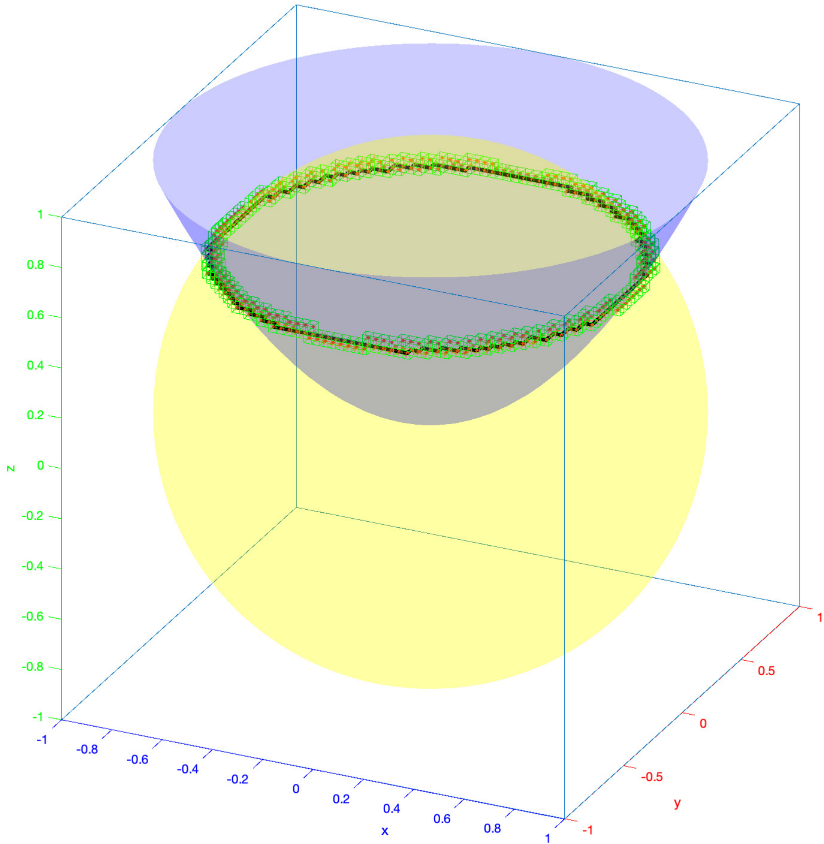
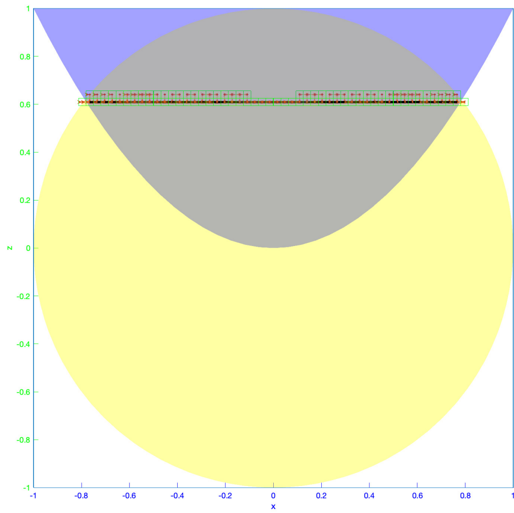
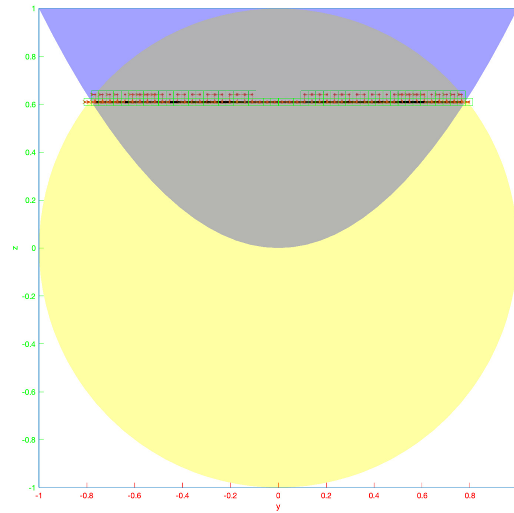


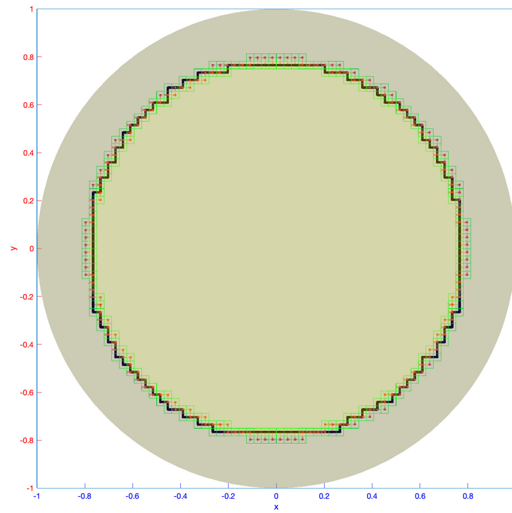
Figure 4.3: The output for experiment 2 seen in 3D



(a) Front view



(b) Side view



(c) Top view

Figure 4.4: The output for experiment 2 viewed from the front, the side, and top

4.2.3 Experiment 3

Approximation of an implicit space curve defined by $\mathbf{f} = (f_1, f_2)$ given by:

$$f_1(x, y, z) = x^4 + 2x^2y^2 + y^4 - 2(x^2 + y^2) + 1 - z$$
$$f_2(x, y, z) = 0.5 - z$$

where inputs for $Subdivide^{\mathbf{f}}(B_0, c, \epsilon)$ are as follows:

$$B_0 = [[-1.2, 1.2], [-1.2, 1.2], [-1.2, 1.2]]$$
$$c = 1.3$$
$$\epsilon = \infty$$

presents the following behavior and results:

Table 4.5: Experiment 3 - Phase 1

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	1	0
1	8	0
2	32	0
3	128	0
4	480	0
5	1920	108
6	3456	660
7	96	0
Time: 10.9647s		

Table 4.6: Experiment 3 - Phase 2

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	768	364
1	1824	0
2	736	0
Time: 31.1397s		

The outputted curve approximation can then be seen drawn on top of the boxes used in the approximation in the following figures:

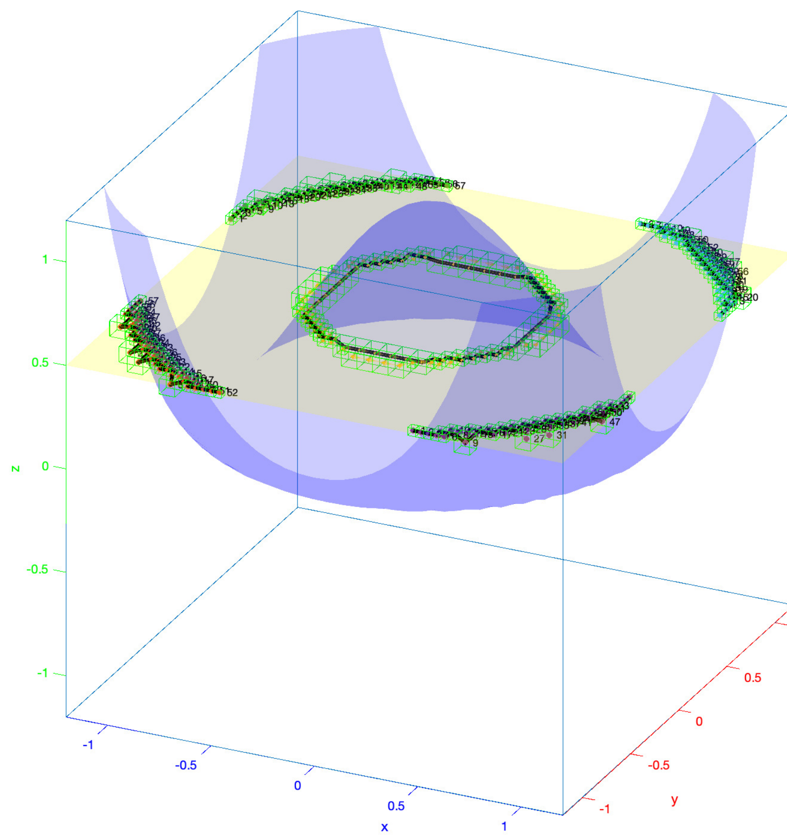
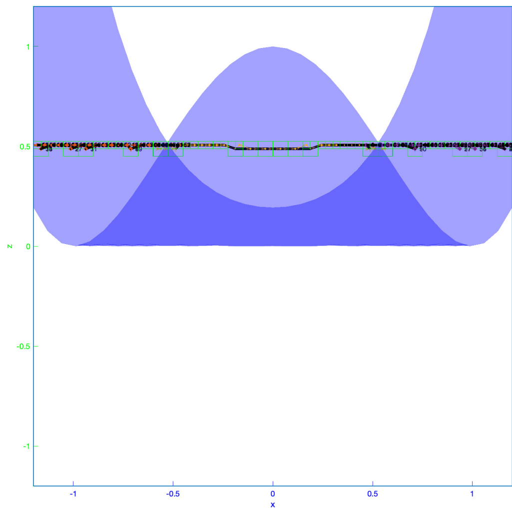
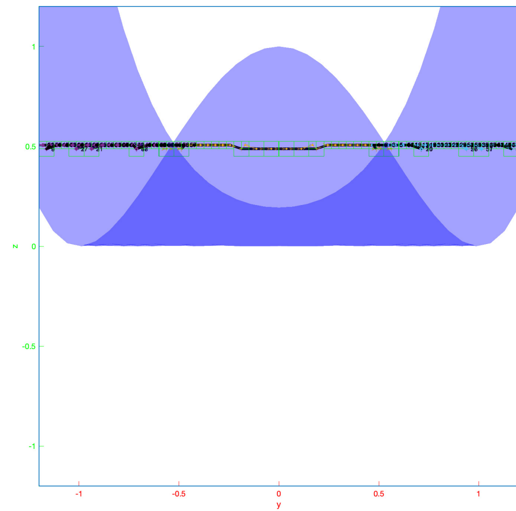


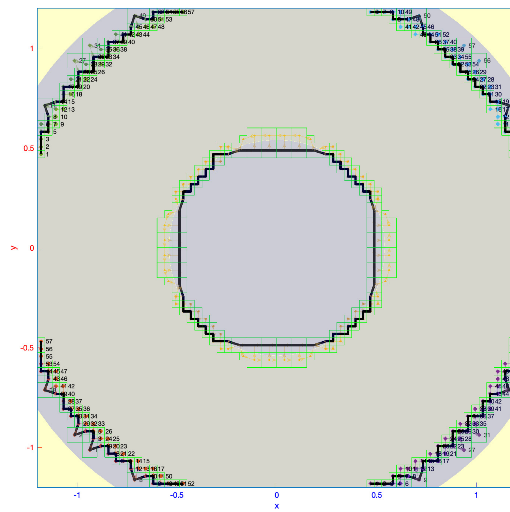
Figure 4.5: The output for experiment 3 seen in 3D



(a) Front view



(b) Side view



(c) Top view

Figure 4.6: The output for experiment 3 viewed from the front, the side, and top

4.2.4 Experiment 4

Approximation of an implicit space curve defined by $\mathbf{f} = (f_1, f_2)$ given by:

$$f_1(x, y, z) = x^2 + y^2 - z^2 - 2$$

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

where inputs for $Subdivide^{\mathbf{f}}(B_0, c, \epsilon)$ are as follows:

$$B_0 = [[-3, 3], [-3, 3], [-3, 3]]$$

$$c = 1.3$$

$$\epsilon = \infty$$

presents the following behavior and results:

Table 4.7: Experiment 4 - Phase 1

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	1	0
1	8	0
2	64	0
3	384	16
4	896	200
5	384	40
Time: 3.1143s		

Table 4.8: Experiment 4 - Phase 2

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	256	224
1	256	0
2	256	0
Time: 23.6716s		

The outputted curve approximation can then be seen drawn on top of the boxes used in the approximation in the following figures:

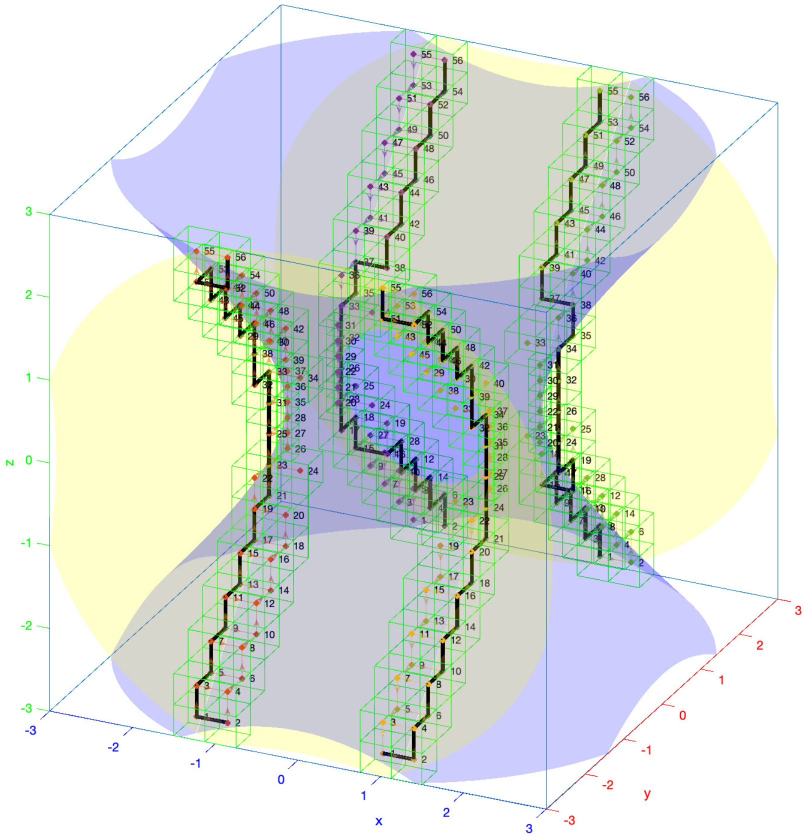
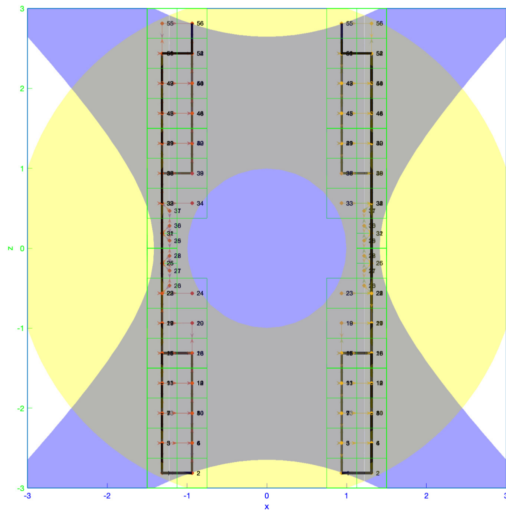
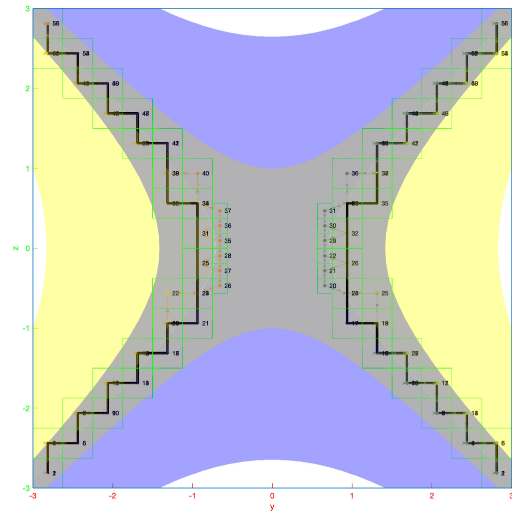


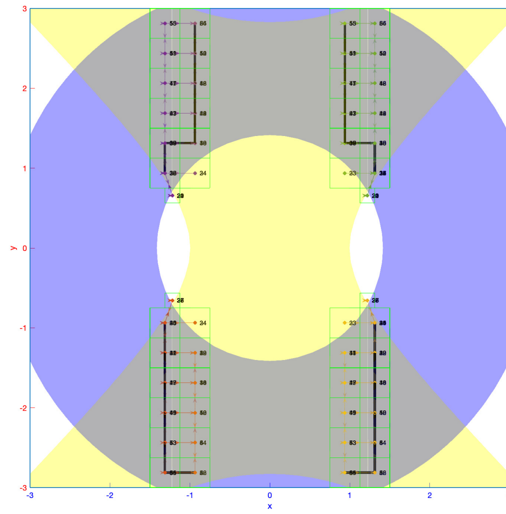
Figure 4.7: The output for experiment 4 seen in 3D



(a) Front view



(b) Side view



(c) Top view

Figure 4.8: The output for experiment 4 viewed from the front, the side, and top

4.2.5 Experiment 5

For this experiment, we look at the approximation of an implicit space curve defined by $\mathbf{f} = (f_1, f_2)$ given by:

$$f_1(x, y, z) = \|\langle qp1, p1 - [x; y; z]\rangle\| / \|qp1\| - \|\langle qp2, p2 - [x; y; z]\rangle\| / \|qp2\|$$

$$f_2(x, y, z) = \|\langle qp1, p1 - [x; y; z]\rangle\| / \|qp1\| - \|\langle qp3, p3 - [x; y; z]\rangle\| / \|qp3\|$$

where $p1, p2, p3$ represent coordinates of three points and $qp1, qp2, qp3$ represent 3D vectors, given as follows:

$$p1 = [0; 0; 4]$$

$$p2 = [0; 0; -4]$$

$$p3 = [1; 1; 0]$$

$$qp1 = [4; 3; 0]$$

$$qp2 = [4; -3; 0]$$

$$qp3 = [4; 1; 4]$$

With this, the given f_1 function implicitly defines the surface of equal distance to the line defined by point $p1$ and vector $qp1$ and the line defined by point $p2$ and vector $qp2$. Similarly, f_2 implicitly defines the surface of equal distance to the line defined by point $p1$ and vector $qp1$ and the line defined by point $p3$ and vector $qp3$.

Thus, the 1D manifold implicitly defined by f_1 and f_2 is the Voronoi diagram of the lines defined by the given points and the corresponding vectors.

For the experiment, inputs for $Subdivide^{f}(B_0, c, \epsilon)$ are given as follows:

$$B_0 = [[-10, 10], [-10, 10], [-10, 10]]$$
$$c = 1.3$$
$$\epsilon = \infty$$

and we observe the following behavior and results:

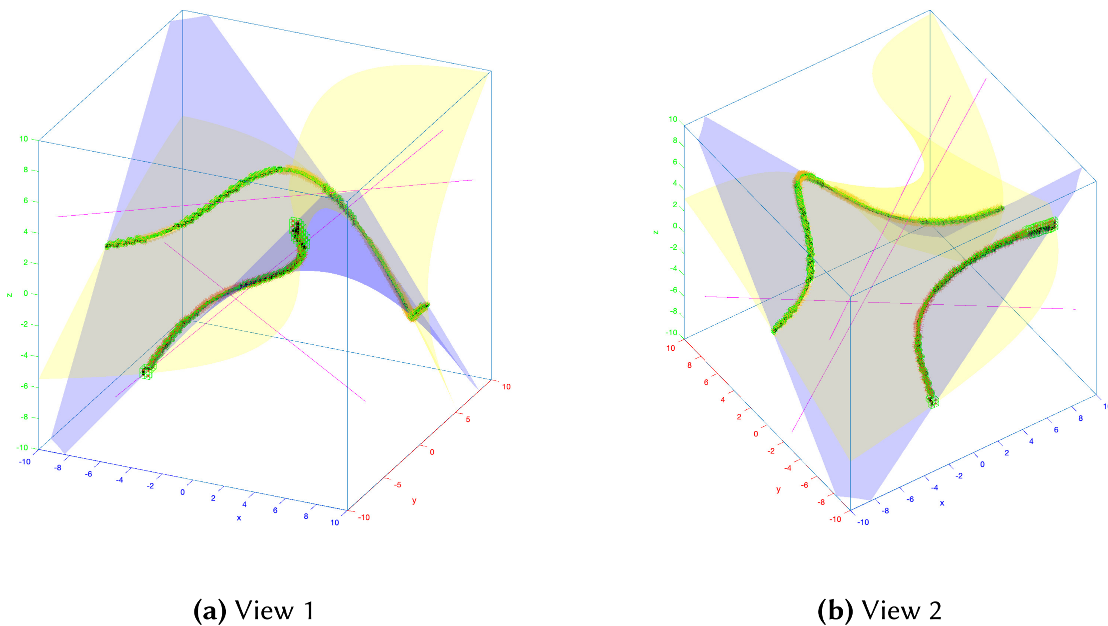
Table 4.9: Experiment 5 - Phase 1

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	1	0
1	8	0
2	64	0
3	368	16
4	1368	200
5	4504	40
6	10424	186
7	20376	2945
8	19056	4949
Time: 305.3464s		

Table 4.10: Experiment 5 - Phase 2

Depth	Number of Evaluated Boxes	Number of Accepted Boxes
0	8080	1551
1	44312	31
2	59488	0
3	45816	0
4	11504	0
5	296	0
Time: 4451.3464s		

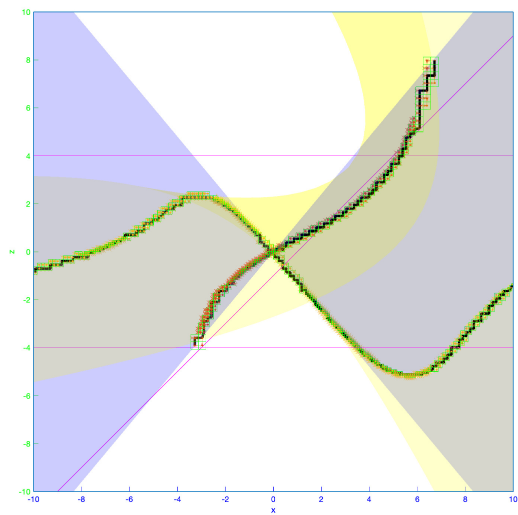
The outputted curve approximation can then be seen drawn on top of the boxes used in the approximation in the following figures:



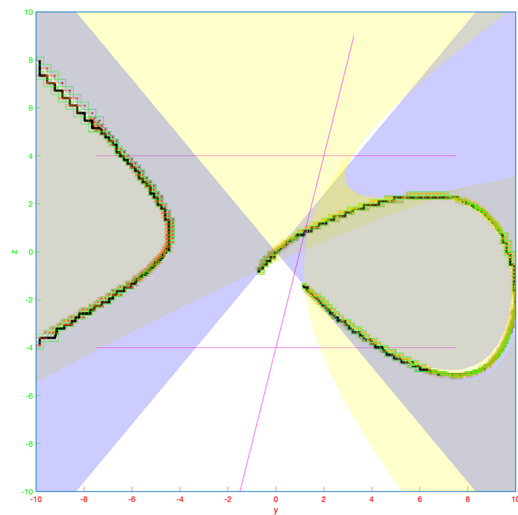
(a) View 1

(b) View 2

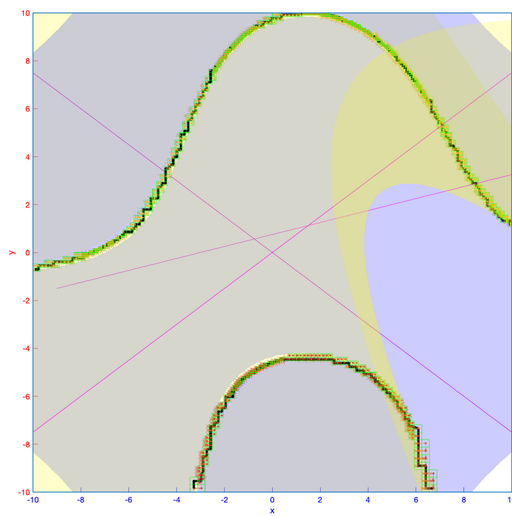
Figure 4.9: The output for experiment 5 seen in 3D, from two different angles



(a) Front view



(b) Side view



(c) Top view

Figure 4.10: The output for experiment 5 viewed from the front, the side, and top

5 | Conclusion

The isotopic approximation of implicitly-defined co-dimension 2 manifolds in n dimensional Euclidean space is an open problem in computational geometry. Building upon previous research on the approximation of co-dimension 1 manifolds, we have approached this problem using interval arithmetic and adaptive subdivision.

We have discussed predicates over interval boxes informative of the behavior of implicit space curves and presented proofs for guarantees provided by the combinations of such predicates.

We further discussed a preconditioning operation, which is currently used to guarantee the eventual detection of non-singular roots with the MK test for boxes containing the roots ([13]) at the cost of the apparent loss of some guarantees provided by our predicates.

Building upon this theory, we designed and implemented a new algorithm for the approximation of space curves and demonstrated the experimental results observed with this algorithm.

While the presented algorithm has not been shown to carry all the theoretical guarantees required for a guarantee for providing isotopic approximations, it has been demonstrated to provide accurate experimental results for the experiments studied in this thesis.

A | Appendix

A.1 Main code for the implemented algorithms

```
%Main code for parallelized subdivision

%% Initialization
tic;
syms f_syms g_syms d_syms x y z
assume(x,'real');
assume(y,'real');
assume(z,'real');

% %Example 1
% f_syms(x,y,z) = x.^2+y.^2-z;
% g_syms(x,y,z) = x.^2+y.^2+z.^2-1;
% B0 = freecurvebox([-ones(3,1),ones(3,1)]);
% MAXEPS = inf;

% %Example 2 - Same as Example 1, but with a limit imposed on box sizes by MAXEPS
% f_syms(x,y,z) = x.^2+y.^2-z;
% g_syms(x,y,z) = x.^2+y.^2+z.^2-1;
% B0 = freecurvebox([-ones(3,1),ones(3,1)]);
% MAXEPS = 0.05;

% %Example 3
% f_syms(x,y,z) = x.^4+2*x.^2*y.^2+y.^4 -2*(x.^2+y.^2) +1-z;
% g_syms(x,y,z) = 0.5-z;
% % B0 = freecurvebox([zeros(3,1),ones(3,1)]);
% B0 = freecurvebox([-ones(3,1),ones(3,1)]).scale(1.2);
% % B0 = freecurvebox([-1,-0.8;-1,-0.8;0.4,0.6]);
% % B0 = freecurvebox([-ones(2,1),ones(2,1);[0.1,2.1]]);
% MAXEPS = inf;

% % %Example 4
% f_syms(x,y,z) = x.^2+y.^2-z.^2 - 2;
% g_syms(x,y,z) = x.^2-y.^2+z.^2 - 1;
% % B0 = freecurvebox([zeros(3,1),ones(3,1)]);
% B0 = freecurvebox([-ones(3,1),ones(3,1)]).scale(3);
% % B0 = freecurvebox([-1,-0.8;-1,-0.8;0.4,0.6]);
% % B0 = freecurvebox([-ones(2,1),ones(2,1);[0.1,2.1]]);
% MAXEPS = inf;

```

```
40  %Example 5: trisector
41  % https://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html
42  %implement the sqrt function for intervals
43  p = [0, 0,1;...
44       0, 0,1;...
45       4,-4,0];
46  direction = [4, 4,4;...
47               3,-3,1;...
48               0, 0,4];
49  q = p+direction;
50  % d_syms(p,q,x,y,z) = norm(cross(q-p,p-[x;y;z]))/norm(q-p);
51  % f_syms(x,y,z) = norm(cross(q(:,1)-p(:,1),p(:,1)-[x;y;z]))/norm(q(:,1)-p(:,1))-norm(cross(q
    ↪  (:,2)-p(:,2),p(:,2)-[x;y;z]))/norm(q(:,2)-p(:,2));
52  % g_syms(x,y,z) = norm(cross(q(:,1)-p(:,1),p(:,1)-[x;y;z]))/norm(q(:,1)-p(:,1))-norm(cross(q
    ↪  (:,3)-p(:,3),p(:,3)-[x;y;z]))/norm(q(:,3)-p(:,3));
53
54  qp1 = q(:,1)-p(:,1);
55  p1 = p(:,1);
56  qp2 = q(:,2)-p(:,2);
57  p2 = p(:,2);
58  qp3 = q(:,3)-p(:,3);
59  p3 = p(:,3);
60  f_syms(x,y,z) = norm(cross(qp1,p1-[x;y;z]))/norm(qp1)-norm(cross(qp2,p2-[x;y;z]))/norm(qp2);
61  g_syms(x,y,z) = norm(cross(qp1,p1-[x;y;z]))/norm(qp1)-norm(cross(qp3,p3-[x;y;z]))/norm(qp3);
62
63  B0 = freecurvebox( 10*[-ones(3,1),ones(3,1)]);
64  MAXEPS = inf;
65
66
67  %##################################################
68  %00000000000000000000000000000000000000000000000000
69  %##################################################
70
71
72  df_syms = gradient(f_syms,[x,y,z]);
73  dg_syms = gradient(g_syms,[x,y,z]);
74  f_mat = matlabFunction(f_syms);
75  g_mat = matlabFunction(g_syms);
76  df_mat = matlabFunction(df_syms);
77  dg_mat = matlabFunction(dg_syms);
78  [f,g,df,dg] = funmanipulation.boxfunction(f_mat,g_mat,df_mat,dg_mat);
79
80  [fig,ax] = createfigure(B0);
81  plotspace = reshape(B0.scale(4).boxdimensions',1,[]);
82  alphavalue = 0.2;
83  hf = fimplicit3(ax,f,plotspace,'EdgeColor','none','FaceAlpha',alphavalue,'FaceColor','b');
84  hg = fimplicit3(ax,g,plotspace,'EdgeColor','none','FaceAlpha',alphavalue,'FaceColor','y');
85  B0.plotbox(ax);
86  drawnow;
87
88  %% for trisector example
89  M = cat(3,p+100*(q-p),q+100*(p-q));
90  hline1 = plot3(ax,squeeze(M(1,:,:))',squeeze(M(2,:,:))',squeeze(M(3,:,:))','m');
91  %% end trisector part
92
93  filename = which('parallel_local_tracecurve');
94  [filepath,name,ext] = fileparts(filename);
95
96  %% Settings
97  %Rules for inherited test results
```

```
98  B0.testresults = cell(1,8);
    ↪  %1:C0(f),2:C0(g),3:C1(f),4:C1(g),5:Jacobian,6:MK,7:MK_face,8:C0_face
99  B0.inherittestindices = 1:4;
100
101 %Depth limit for phase 1
102 depthlimit = 8;
103 %Depth limit for phase 2
104 numiterMKlimit = 6;
105
106 tStart = tic;
107
108 %% Subdivision Phase 1
109 Q = B0; %Input of the first phase of subdivision
110 QJac = []; %Output of first phase of subdivision
111
112 % Depth for phase 1
113 depth = 0;
114
115 % Create a subdivision of boxes, which all satisfy the predicates untill C1 tests
116 % and the Jacobian tests hold (where boxes satisfying C0 get excluded at each level)
117 while ~isempty(Q) && depth <= depthlimit
118     Q_next = cell(length(Q),1);
119     QJac_add = cell(length(Q),1);
120
121     disp(['Phase 1: depth = ', num2str(depth), ' | length(Q) = ', num2str(length(Q))]);
122
123     %Parallel Subdivision
124     parfor i = 1:length(Q)
125         B_par = Q(i);
126         if ~local_predicate.C0(B_par,f,1) && ~local_predicate.C0(B_par,g,2)
127             if B_par.radius<MAXEPS && local_predicate.C1(B_par,df,3) &&
                ↪  local_predicate.C1(B_par,dg,4) && ... %&&
                ↪  local_predicate.C1cross(B_par,df,dg,5)
128                local_predicate.Jacobian(B_par,df,dg,5)
129                 QJac_add{i} = B_par;
130             else
131                 children = B_par.split;
132                 Q_next{i} = children;
133             end
134         end
135     end
136
137     accepted = [QJac_add{:}];
138
139     disp(['# accepted boxes = ', num2str(length(accepted))]);
140
141     %Collection of results
142     Q = [Q_next{:}];
143     QJac = [QJac, QJac_add{:}];
144
145     depth = depth+1;
146 end
147
148 if ~isempty(Q) && depth == depthlimit + 1
149     disp("Phase 1 stopped due to depth limit");
150 end
151
152 disp(['Time for Phase 1: ',num2str(toc(tStart)),'s']);
153
154 disp("Phase 1 finalized.");
155 disp("Proceeding to phase 2...");
```

```
156
157 tPhase2 = tic;
158
159 %% Subdivision Phase 2
160 QMK = QJac; %Input of the second phase of subdivision
161 Qcurve = []; %Output of second phase of subdivision
162 %%
163 %Depth for phase 2
164 numiterMK = 1;
165
166 while ~isempty(QMK) && numiterMK <= numiterMKlimit
167     %Iteration cell arrays
168     QMK_next = cell(length(QMK),1);
169     Qcurve_add = cell(length(QMK),1);
170
171     disp(['Phase 2: numiterMK = ', num2str(numiterMK), ' | length(QMK) = ',
        ↪  num2str(length(QMK))]);
172
173     %Parallel Subdivision
174     parfor i = 1:length(QMK)
175         B_par = QMK(i);
176         if ~local_predicate.C0(B_par,f,1) && ~local_predicate.C0(B_par,g,2)
177             if local_predicate.Jacobian(B_par,df,dg,5) &&
                ↪  local_predicate.MK_face(B_par,f,df,g,dg,7)
                ↪  %local_predicate.MK_face(B_par,f,df,g,dg,7)
178                 if any(B_par.testresults{7})
179                     Qcurve_add{i} = B_par;
180                end %Else: MK test has succeeded not in finding a root, but in excluding all in
                    ↪  internal call to C0_faces
181             else
182                 children = B_par.split;
183                 QMK_next{i} = children;
184             end
185         end
186     end
187
188     accepted = [Qcurve_add{:}];
189
190     disp(['# accepted boxes = ', num2str(length(accepted))]);
191
192     %Collection of results
193     QMK = [QMK_next{:}];
194     Qcurve = [Qcurve, Qcurve_add{:}];
195
196     numiterMK = numiterMK+1;
197 end
198
199 if ~isempty(QMK) && numiterMK == numiterMKlimit + 1
200     disp("Phase 2 stopped due to depth limit");
201 end
202
203 disp(['Time for Phase 2: ',num2str(toc(tPhase2)),'s']);
204
205 disp("Phase 2 finalized.");
206
207 %% Outputs
208 disp(['Total time for subvision: ',num2str(toc(tStart)),'s']);
209
210 disp(['Number of boxes in Q: ',num2str(length(Q))]);
211
212 %boxes which were not classified and fail the Jacobian
```

```
213 for i = 1:length(Q)
214     Q(i).plotbox(ax,'r');
215 end
216
217 disp(['Number of boxes in QMK: ',num2str(length(QMK))]);
218
219 %boxes which were not classified but pass the Jacobian
220 for i = 1:length(QMK)
221     QMK(i).plotbox(ax,'y');
222 end
223
224 disp(['Number of accepted boxes: ',num2str(length(Qcurve))]);
225
226 %boxes which pass the MK test and satisfy all the C1/Jacobian requirements
227 for i = 1:length(Qcurve)
228     Qcurve(i).plotbox(ax,'g');
229 end
230
231 %% ####################
232
233
234
235
236
237 %% Curve Construction
238 tic;
239
240 leavessub = leaves(B0);
241 n = length(leavessub);
242 nodesnearcurvelogical = false(n,1);
243 centers = zeros(3,n);
244
245 parfor i = 1:n
246     B = leavessub(i);
247     leavessub(i).boxid = i;
248     centers(:,i) = leavessub(i).center;
249     if any(B.testresults{7}) %all boxes in the final subdivision which pass the MK test also
    ↪  pass Jacobian
250         nodesnearcurvelogical(i) = true;
251     end
252 end
253 nodes = 1:n;
254 nodesnearcurve = nodes(nodesnearcurvelogical); %list of nodes according to initial IDs
255
256 Afull = logical(sparse(n,n));
257
258 for i = 1:n %Cannor directly use parfor here
259     neighbors = leavessub(i).neighbors;
260     for j = 1:length(neighbors) %probably not worth doing with parfor
261         Afull(i,neighbors(j).boxid) = true;
262     end
263 end
264
265 Afull = Afull | Afull';
266 Gfull = graph(Afull);
267
268 Gnearcurve = subgraph(Gfull,nodesnearcurve);
269
270 bins = conncomp(Gnearcurve);
271 numcomps = numel(unique(bins));
272 curvepieces = cell(1,numcomps);
```

```
273
274  disp(["Numcomps: ", num2str(numcomps)])
275
276  for comp = 1:numcomps
277      nodescomponentlogical = bins == comp;
278      nodescomponent = nodesnearcurve(nodescomponentlogical);
279      Gcomponent = subgraph(Gnearcurve,nodescomponentlogical);
280
281      ndirected = length(nodescomponent);
282
283      disp(["Comp ", num2str(comp), ":"])
284      disp(["Component size:", ndirected])
285
286      if ndirected == 1
287          disp("Skipped comp of size 1.")
288          disp("-----ooooo-----")
289          continue
290      end
291
292      Adirected = logical(sparse(ndirected,ndirected));
293      edges = table2array(Gcomponent.Edges);
294      for i = 1:size(edges,1)
295          edge = edges(i,:);
296          B1 = leavessub(nodescomponent(edge(1)));
297          B2 = leavessub(nodescomponent(edge(2)));
298          center1 = B1.center;
299          center2 = B2.center;
300          direction = (B1.radius+B2.radius)/sqrt(3) - (center2-center1) < 10*eps;
301          direction = direction*(-1)^(center1(direction)<center2(direction));%B1 has smaller
         ↪  coordinates than B2 in direction
302          if all((direction')*B1.testresults{5} >= 0) && all((direction')*B2.testresults{5} >=0)
303              Adirected(edge(1),edge(2)) = true;
304          elseif all((direction')*B1.testresults{5} <= 0) && all((direction')*B2.testresults{5}
         ↪  <=0)
305              Adirected(edge(2),edge(1)) = true;
306          end
307      end
308      Gdirected = digraph(Adirected);
309      hGdirected = plot(ax,Gdirected,'XData',centers(1,nodescomponent),'YData',centers(2,nodesc⌋
     ↪  omponent),'ZData',centers(3,nodescomponent));
310      hGdirected.ArrowSize = 5; %7.5;
311      hGdirected.LineWidth = 0.5; %0.75;
312
313      %%Find cycle or path
314      %Use BFS to create a tree with shortest paths
315      %Then run DFS to check for paths that have all boxes nearby (neighbors of neighbors)
316      %Repeat for all neighbors of the starting vertex and select the shortest path
317      [path,foundcycle] = findcycle(Gdirected);
318      if ~foundcycle
319          disp("Cycle not found!")
320          path = findpath(Gdirected);
321      else
322          disp("Cycle found.")
323      end
324      disp(["Path length:", length(path)])
325      disp("-----ooooo-----")
326      curvepieces{comp} = path;
327
328      %Plot curve
329      Coordinates = zeros(3,length(path));
330      for i = 1:length(path)
```

```
331         Coordinates(:,i) = leavessub(nodescomponent(path(i))).center;
332     end
333     plot3(ax,Coordinates(1,:),Coordinates(2,:),Coordinates(3,:),'-k','Linewidth',3);
334 end
335 disp(['time for graph algorithm: ',num2str(toc),'s']);
```

A.2 Code for the cycle finding algorithm used in reconstruction

```
function [cycle,foundcycle] = findcycle(G)
    if nargin < 1
        s = [1 1 2 3 3 4 4 6 6 7 8 7 5];
        t = [2 3 4 4 5 5 6 1 8 1 3 2 8];
        G = digraph(s,t);
        plot(G);
        hold on;
    end
    %We use node 1 or any of its neighbors as start vertex

    edges = table2array(G.Edges);
    n = size(G.Nodes,1);
    neighbors = (edges(:,1) == 1) | (edges(:,2) == 1); %any(edges == 1,2);
    neighbors = unique(edges(neighbors,:));
    foundcycle = false;
    cycle = [];
    i = 1;
    while ~foundcycle && i <= length(neighbors)
        s = neighbors(i);
        [~,D] = shortestpathtree(G,'all',s);
        maxdist = max(D(~isinf(D)));

        farthestnodesbool = D >= maxdist-2 & ~isinf(D) & D>1;
        farthestnodes = 1:n;
        farthestnodes = farthestnodes(farthestnodesbool);
        j = 1;
        while maxdist < inf && ~foundcycle && j <= length(farthestnodes)
            if findedge(G,farthestnodes(j),s) || findedge(G,s,farthestnodes(j))
                foundcycle = true;
                t = farthestnodes(j);
                TR = shortestpathtree(G,t,s,'OutputForm','cell');
                cycle = [TR{1},t];
            end
            j = j+1;
        end
        i = i+1;
    end
end
```

A.3 Code for the path finding algorithm used in reconstruction

```
function path = findpath(G)
    if nargin < 1
        s = [1 3 3 4 6 6 7 8 7 5];
        t = [2 4 5 5 1 8 1 3 2 8];
        G = digraph(s,t);
        plot(G);
        hold on;
    end
    edges = table2array(G.Edges);
    Gundirected = graph(edges(:,1),edges(:,2));
%     H = flipedge(G);
    v = 1;
    [~,D] = shortestpathtree(Gundirected,'all',v);
    s = find(D==max(D),1);
    [~,D] = shortestpathtree(Gundirected,'all',s);
    t = find(D==max(D),1);
    TR = shortestpathtree(Gundirected,s,t,'OutputForm','cell');
    path = TR{1};

    %Warning: this is not 100% correct, we would also need to additionally check that the first
    ↪ and last boxes are on the domain boundary
end
```

A.4 Code for the implemented tests

```
1  classdef local_predicate < handle
2
3      %Assuming the following indexing for inherited test results:
4      %1:C0(f),2:C0(g),3:C1(f),4:C1(g),5:JC,6:MK,7:MK_face,8:C0_face
5
6      properties (Constant)
7          JC_scale = 4;
8          MK_scale = 2;
9          %For the test used on 6 sides:
10         MK_box_scale = 1.3; %Scales each box before determining the faces, must be above 1 for
           ↪  current implementation
11         MK_face_scale = 1.3; %Scales each face of the box, must be above 1 for current
           ↪  implementation
12         MK_jac_ind = 6;
13        C0_depth = 0; %0 means just evalaute on the main box, and gives the best performence by
           ↪  far (for C0_depth)
14         C0_face_depth = 0; %0 means just evalaute on the main box, and seems to give the best
           ↪  performence (for  C0_face_depth)
15     end
16
17     methods
18         function this = local_predicate()
19             local_tracecurve;
20         end
21     end
22     methods(Static)
23         % ##### C0 #####
24         function bool = C0(B,f,ind)
25             if nargin > 2
26                 if isempty(B.testresults{ind})
27                     bool = local_predicate.C0core(B,f);
28                     B.testresults{ind} = bool;
29                 else
30                     bool = B.testresults{ind};
31                 end
32             else
33                 bool = local_predicate.C0core(B,f);
34             end
35         end
36         function bool = C0core(B,f)
37             %Returns true if the surface f=0 surely does not pass through B
38             %bool = 0 \notin f(B)
39             bool = funexcludes0(f,B,local_predicate.C0_depth);
               ↪  %~interval.zeros(1).subset(f(B));
40         end
41         % ####################
42
43
44
45         % ##### C1 #####
46         function bool = C1(B,df,ind)
47             if nargin > 2
48                 if isempty(B.testresults{ind})
49                     bool = local_predicate.C1core(B,df);
50                     B.testresults{ind} = bool;
```

```
51                  else
52                      bool = B.testresults{ind};
53                  end
54              else
55                  bool = local_predicate.C1core(B,df);
56              end
57          end
58          function bool = C1core(B,df)
59              %C1(B,df, IND)
60              %Returns true if the surface f=0 satisfies the C1 condition
61              %bool = 0 \notin < df(B) , df(B) >
62                  Bint = B.interval;
63                  bool = ~interval.zeros(3,1).subset(df(Bint)'*df(Bint));
64          end
65          function bool = C1cross(B,df,dg,ind)
66              if nargin > 2
67                  if isempty(B.testresults{ind})
68                      bool = local_predicate.C1crosscore(B,df,dg);
69                      B.testresults{ind} = bool;
70                  else
71                      bool = B.testresults{ind};
72                  end
73              else
74                  bool = local_predicate.C1crosscore(B,df);
75              end
76          end
77          function bool = C1crosscore(B,df,dg)
78              %C1cross(B,df,dg,ind)
79              %Returns true if the curve defined by f=0 and g=0 satisfies the C1 condition
80              %bool = 0 \notin < df(B) x dg(B) , df(B) x dg(B)>
81                  Bint = B.interval;
82                  cross_fg = cross(df(Bint),dg(Bint));
83                  bool = ~interval.zeros(3,1).subset(cross_fg'*cross_fg);
84          end
85          % ####################
86
87
88
89          % ##### JC #####
90          function [bool,v] = Jacobian(B,df,dg,ind)
91              if nargin > 3
92                  if isempty(B.testresults{ind})
93                      [bool,v] = local_predicate.Jacobiancore(B,df,dg);
94                      B.testresults{ind} = v;
95                  else
96                      v = B.testresults{ind};
97                      bool = any(v);
98                  end
99              else
100                 [bool,v] = local_predicate.Jacobiancore(B,df,dg);
101             end
102             B.passesJacobian = B.passesJacobian || bool; %for function markcurve
103         end
104         function [bool,v] = Jacobiancore(B,df,dg)
105             B = B.scale(local_predicate.JC_scale);
106             v = zeros(3,1);
107             dB = [df(B),dg(B)];
108             for i = 1:3
109                 testinterval = -(-1)^i*det(dB(setdiff([1,2,3],i),:));
110                 if 0 < testinterval
111                     v(i) = 1;
```

```
112                 elseif testinterval < 0
113                     v(i) = -1;
114                 else
115                     v(i) = 0; %Jacobian did not succeed
116                 end
117             end
118             bool = any(v);
119         end
120         % ####################
121 
122         % ##### MK #####
123         function bool = MK(B,f,df,g,dg,ind,ax)
124             if nargin > 5
125                 if isempty(B.testresults{ind})
126                     if nargin > 6
127                         bool = local_predicate.MKcore(B,f,df,g,dg,ax);
128                     else
129                         bool = local_predicate.MKcore(B,f,df,g,dg);
130                     end
131                     B.testresults{ind} = bool;
132                 else
133                     bool = B.testresults{ind};
134                 end
135             else
136                 bool = local_predicate.MKcore(B,f,df,g,dg);
137             end
138         end
139         function bool = MKcore(B,f,df,g,dg,ax)
140             %Returns true if there are two pairs of opposite faces of B
141             %with f having opposite sign on one pair, and g on the other.
142             %Includes preconditioning around the box center
143             B = B.scale(local_predicate.MK_scale);
144             m = B.center;
145             dfm = df(m); dgm = dg(m);
146 
147                 Jm = [dfm,dgm,cross(dfm,dgm)]';
148             invJm = Jm^(-1);
149             h = @(varargin) cross(dfm,dgm)'*(funmanipulation.convert2vec(varargin{:})-m);
             ↪  %varargin can be x,y,z components or box or interval vector
150             fprime = @(varargin) invJm(1,:) * [f(varargin{:});g(varargin{:});h(varargin{:})];
151             gprime = @(varargin) invJm(2,:) * [f(varargin{:});g(varargin{:});h(varargin{:})];
152             hprime = @(varargin) invJm(3,:) * [f(varargin{:});g(varargin{:});h(varargin{:})];
153 
154 
155             faces = B.facets;
156             if ~funsmaller0(fprime,faces(1,1)) || ~funsmaller0(@(B) -fprime(B),faces(1,2)) ||
             ↪  ...
157                ~funsmaller0(gprime,faces(2,1)) || ~funsmaller0(@(B) -gprime(B),faces(2,2)) ||
                ↪  ...
158                ~funsmaller0(hprime,faces(3,1)) || ~funsmaller0(@(B) -hprime(B),faces(3,2))
159                 bool = false;
160             else
161                 bool = true;
162             end
163 
164 
165             if nargin>5
166                 Bint = B.interval;
167                 plothandle = B.plotbox(ax,'b');
168                if ~exist('axorig','var') || isempty(axorig) || ~isgraphics(axorig) %#ok<NODEF>
169                     subplot(1,3,1,ax);
```

```
170                     axorig = subplot(1,3,2);
171                     r = groot;
172                     Monitors = r.MonitorPositions;
173                     [~,M] = max(Monitors(:,3));
174                     fig = ax.Parent;
175                     sizex = Monitors(M,3);
176                     originalPos = fig.Position;
177                     newPos = originalPos;
178 
179                     set(fig, 'units', 'pixels', 'position',newPos);
180                     prepareaxes(axorig);
181                 end
182                 if ~exist('axtrans','var') || isempty(axtrans) || ~isgraphics(axtrans)
                    ↪   %#ok<NODEF>
183                     axtrans = subplot(1,3,3);
184                     prepareaxes(axtrans);
185                 end
186 
187                 plothoriginal = local_predicate.plotfunctions(axorig,f,g,h,Bint);
188                 plothtransformed =
                    ↪   local_predicate.plotfunctions(axtrans,fprime,gprime,hprime,Bint);
189                 ploth = [plothoriginal,plothtransformed];
190             end
191 
192             if nargin>5
193                 if bool
194                     titletext = 'success';
195                 else
196                     titletext = [...
197                          '$ f^\sim(B_x^-)$ =
                         ↪   [',num2str(fprime(faces(1,1)).bounds),']',newline,...
198                          '$-f^\sim(B_x^+)$ =
                         ↪   [',num2str(uminus(fprime(faces(1,2))).bounds),']',newline,...
199                          '$ g^\sim(B_y^-)$ =
                         ↪   [',num2str(gprime(faces(2,1)).bounds),']',newline,...
200                          '$-g^\sim(B_y^+)$ =
                         ↪   [',num2str(uminus(gprime(faces(2,2))).bounds),']',newline,...
201                          '$ h^\sim(B_y^-)$ =
                         ↪   [',num2str(hprime(faces(3,1)).bounds),']',newline,...
202                          '$-h^\sim(B_y^+)$ =
                         ↪   [',num2str(uminus(hprime(faces(3,2))).bounds),']',newline];
203                 end
204                 title(axtrans,titletext,'interpreter','latex','FontSize',16);
205 
206                 for i = 1:length(ploth)
207                     delete(ploth{i});
208                 end
209                 delete(plothandle);
210             end
211         end
212         % ####################
213 
214         % ##### Plot #####
215         function ploth = plotfunctions(ax,fun1,fun2,fun3,B)
216         %plot the functions within the box for testing reasons
217             if nargin == 5
218                 plotinterval = [B(1).bounds,B(2).bounds,B(3).bounds];
219                 axis(ax,plotinterval);
220             else
221                 plotinterval = axis(ax);
222             end
```

```
223                %Use the function arrayfun for the next 3 lines possibly
224                fun1_element = @(x,y,z) elementwisefunction(fun1,x,y,z);
225                fun2_element = @(x,y,z) elementwisefunction(fun2,x,y,z);
226                fun3_element = @(x,y,z) elementwisefunction(fun3,x,y,z);
227                ploth = cell(1,3);
228                ploth{1} = fimplicit3(ax,fun1_element,plotinterval,'b');
229                ploth{2} = fimplicit3(ax,fun2_element,plotinterval,'r');
230                ploth{3} = fimplicit3(ax,fun3_element,plotinterval,'g');
231                alpha(ploth{1},0.5);
232                alpha(ploth{2},0.5);
233                alpha(ploth{3},0.5);
234            end
235            function ploth = plotfunctions2(ax,fun1,fun2,B)
236            %plot the functions within the box for testing reasons
237                if nargin == 5
238                    plotinterval = [B(1).bounds,B(2).bounds,B(3).bounds];
239                    axis(ax,plotinterval);
240                else
241                    plotinterval = axis(ax);
242                end
243                %Use the function arrayfun for the next 3 lines possibly
244                fun1_element = @(x,y,z) elementwisefunction(fun1,x,y,z);
245                fun2_element = @(x,y,z) elementwisefunction(fun2,x,y,z);
246                ploth = cell(1,2);
247                ploth{1} = fimplicit3(ax,fun1_element,plotinterval,'b');
248                ploth{2} = fimplicit3(ax,fun2_element,plotinterval,'r');
249                alpha(ploth{1},0.5);
250                alpha(ploth{2},0.5);
251            end
252            % ####################
253
254            % ##### MK_face #####
255            function [bool,v] = MK_face(B,f,df,g,dg,ind,ax)
256                if nargin > 5
257                    if isempty(B.testresults{ind})
258                        if nargin > 6
259                            [bool,v] = local_predicate.MK_face_core(B,f,df,g,dg,ax);
260                        else
261                            [bool,v] = local_predicate.MK_face_core(B,f,df,g,dg);
262                        end
263                        B.testresults{ind} = v;
264                    else
265                        v = B.testresults{ind};
266                        bool = any(v);
267                    end
268                else
269                    [bool,v] = local_predicate.MK_face_core(B,f,df,g,dg);
270                end
271            end
272            function [bool,v] = MK_face_core(B,f,df,g,dg)
273                %Matrix showing which surfaces intersect the curve (populated below):
274                res = zeros(3,2);
275                %(Does not consider intersections in non-parameterizable direction's faces)
276
277                spandir = zeros(3,1);
278
279                %C0 on the faces of the initial box
280                [C0_face_small, ~] = local_predicate.C0_face_core(B.scale(1),f,g);
281                if C0_face_small %MK test terminates true but no faces are found, will not be
                   ↪  considered during reconstruction later
282                    bool = true;
```

```
283                 v = reshape(res.',1,[]);
284                 return
285             end
286 
287             Bscaled = B.scale(local_predicate.MK_box_scale);
288 
289             %C0 on the faces of the scaled box
290             [C0_face, exclusions] = local_predicate.C0_face_core(Bscaled,f,g);
291             if C0_face %MK test terminates true but no faces are found, will not be considered
            ↪  during reconstruction later
292                 bool = true;
293                 v = reshape(res.',1,[]);
294                 return
295             end
296 
297             [jacbool,jac] = local_predicate.Jacobian(B,df,dg, local_predicate.MK_jac_ind);
298             if ~jacbool %Jacobian test fails, need to split
299                 bool = false;
300                 v = reshape(res.',1,[]);
301                 return
302             end
303 
304             faces = Bscaled.facets;
305 
306             for i = 1:3
307                 if jac(i) ~= 0 %Parameterizable in this direction
308                     for j = 1:2
309                         if exclusions(2*(i-1)+j) ~= 1 %do not bother if face would be excluded
310                             face = faces(i,j);
311                             m = face.center;
312                             dfm = df(m); dgm = dg(m);
313 
314                             %Remove the i'th dimension from dfm/dfg to get 2d versions
315                             dfm(i,:) = [];
316                             dgm(i,:) = [];
317 
318                             Jm = [dfm,dgm]';
319                             invJm = Jm^(-1);
320 
321                             fprime = @(varargin) invJm(1,:) * [f(varargin{:});g(varargin{:})];
322                             gprime = @(varargin) invJm(2,:) * [f(varargin{:});g(varargin{:})];
323 
324                             edges = face.scale(local_predicate.MK_face_scale).facets;
325 
326                             %facebool
327                             if ~funsmaller0(fprime,edges(1,1)) || ~funsmaller0(@(B)
                            ↪  -fprime(B),edges(1,2)) || ...
328                                ~funsmaller0(gprime,edges(2,1)) || ~funsmaller0(@(B)
                               ↪  -gprime(B),edges(2,2))
329                                 facebool = false;
330                             else
331                                 facebool = true;
332                             end
333 
334                             res(i,j) = facebool;
335                         end
336                     end
337                     dirbool = res(i,1) && res(i,2);
338                     spandir(i) = dirbool;
339                 end
340             end
```

```
341
342                 %Flatten result for output
343                 v = reshape(res.',1,[]);
344                 bool = any(v);
345
346             end
347             %###################
348
349             % ##### C0_face #####
350             function [bool,v] = C0_face(B,f,g,ind) %Returns true iff all surfaces can be excluded
351                 if nargin > 3
352                     if isempty(B.testresults{ind})
353                         [bool,v] = local_predicate.C0_face_core(B,f,g);
354                         B.testresults{ind} = v;
355                     else
356                         v = B.testresults{ind};
357                         bool = all(v);
358                     end
359                 else
360                     [bool,v] = local_predicate.C0_face_core(B,f,g);
361                 end
362             end
363             function [bool,v] = C0_face_core(B,f,g)
364                 %Matrix showing which surfaces can be excluded
365                 res = zeros(3,2);
366
367                 faces = B.facets;
368
369                 for i = 1:3
370                     for j = 1:2
371                         face = faces(i,j);
372
373                         res(i,j) = funexcludes0(f,face,local_predicate.C0_face_depth) ||
                        ↪  funexcludes0(g,face,local_predicate.C0_face_depth); %Can replace by
                        ↪  calls to C0_core
374                     end
375                 end
376
377                 %Flatten result for output
378                 v = reshape(res.',1,[]);
379                 bool = all(v);
380             end
381             %###################
382         end
383     end
```

A.5 Code defining intervals and interval arithmetic

```
classdef interval < handle & matlab.mixin.Copyable
    %Interval arithmetic operations
        %functions

        %Overloading operartors for the interval class
        %a+b
        %a-b
        %-a
        %+a
        %a.*b
        %a*b
        %a./b
        %a.\b
        %a/b
        %a\b
        %a.^b
        %a^b
        %a < b
        %a > b
        %a <=b
        %a >= b
        %a ~= b
        %a == b
            %%%Not included
            %%% a&b
            %%% a|b
            %%% ~a
            %%% a:d:b
            %%% a:b
        %a'
        %a.'
            %%% [a,b]
            %%% [a;b]
            %%% a(s1,s2,...,sn)
            %%% a(s1,s2,...,sn) = b
            %%% b(a)
        %sqrt
        %nthroot
        %abs
        %sign
        %norm %L_2 norm of interval vectors
        %norm_p %L_p norm
        %disp
        %int2str
        %extractbounds: used when working with matrices of intervals
        %sum
        %min
        %max
        %cross(a,b): computes the cross product of two interval vectors of size 3
        %det: computes the determinant of a 2x2 interval matrix
        %a.cap(b):      computes the intersection of intervals a and b
        %a.cup(b):      computes the union of intervals a and b

        %a \subset b:   a.subset(b)
        %a \superset b: a.superset(b)
```

```
56          %x \in a:       a.element(x)
57
58          %scale(factor): if [a,b] is bounded then this doubles the length of the interval, while
            ↪  keeping the midpoint
59          %               if [a,b] is unbounded, then this halves the length of the uncovered
            ↪  number line
60
61          %linspace(n): creates a mesh with each interval getting uniformly split into n values
62      properties
63          bounds %[a,b]
64              %if a<=b: I = [a,b]
65              %if b<a:  I = [-inf,a] \cup [b,inf]
66              %The empty interval is represented by [inf,-inf]
67              %Imagine the ends of the number line being connected at infiniity
68      end
69
70      methods
71          function this = interval(lowerbounds,upperbounds)
72              %Accepts two matrices of same size as bounds to create a matrix of intervals
73              if nargin > 0
74                  if nargin == 1
75                      upperbounds = lowerbounds;
76                  end
77                  v = size(lowerbounds);
78                  if all(size(lowerbounds) == size(upperbounds))
79                      numv = prod(v);
80                      if numv > 1
81                          for ind = numv:-1:1
82                              this(ind) = interval(lowerbounds(ind),upperbounds(ind));
83                          end
84                          if length(v) == 1
85                              v = [v,1];
86                          end
87                          this = reshape(this,v);
88                      else
89                          this.bounds = [lowerbounds,upperbounds];
90                      end
91                  else
92                      warning('wrong interval definition');
93                  end
94              end
95          end
96          %Overloading operators with MATLAB:
97          %https://ch.mathworks.com/help/matlab/matlab_oop/implementing-operators-for-your-clas⌋
            ↪  s.html
98          function result = plus(a,b)
99              %a+b
100             fct = @interval.plus_element;
101             result = interval.elementwiseoperator(fct,a,b);
102         end
103         function result = minus(a,b)
104             %a-b
105             result = a+(-b);
106         end
107         function result = uminus(a)
108             %-a
109             fct = @interval.uminus_element;
110             result = interval.elementwiseoperator(fct,a);
111         end
112         function result = uplus(a)
113             %+a
```

```
114                 result = a;
115             end
116             function result = times(a,b)
117                 %a.*b
118                 fct = @interval.times_element;
119                 result = interval.elementwiseoperator(fct,a,b);
120             end
121             function result = mtimes(a,b)
122                 %a*b
123                 [i,j] = size(a);
124                 [j2,k] = size(b);
125                 result = interval(zeros([i,k]));
126                 if j == j2
127                     if ~isa(a,'interval')
128                         a = interval(a);
129                     end
130                     if ~isa(b,'interval')
131                         b = interval(b);
132                     end
133                     for iind = 1:i
134                         for kind = 1:k
135                             currentsum = interval(0,0);
136                             for jind = 1:j
137                                currentsum = currentsum+interval.times_element(a(iind,jind).bounds
                                  ↪  ,b(jind,kind).bounds);
138                             end
139                             result(iind,kind) = currentsum;
140                         end
141                     end
142                 else
143                     error('The matrix dimensions are not matching');
144                 end
145             end
146             function result = rdivide(a,b)
147                 %a./b
148                 if isa(b, 'interval')
149                     result = a.*inverse(b);
150                 else
151                     result = a.*(1/b);
152                 end
153             end
154             function result = inverse(a)
155                 %1./a
156                 fct = @interval.inverse_element;
157                 result = interval.elementwiseoperator(fct,a);
158             end
159                 %a.\b
160                 %a/b
161                 %a\b
162             function result = power(a,b)
163                 %a.^b
164                 fct = @interval.power_element;
165                 result = interval.elementwiseoperator(fct,a,b);
166             end
167                 %a^b
168             function result = lt(a,b)
169                 %a < b
170                 fct = @interval.lt_element;
171                 result = interval.elementwiseoperator(fct,a,b);
172             end
173             function result = gt(a,b)
```

```
174             %a > b
175             result = b<a;
176         end
177         function result = le(a,b)
178             %a <=b
179             fct = @interval.le_element;
180             result = interval.elementwiseoperator(fct,a,b);
181         end
182         function result = ge(a,b)
183             %a >= b
184             result = b <= a;
185         end
186         function result = ne(a,b)
187             %a ~= b
188             result = ~(a==b);
189         end
190         function result = eq(a,b)
191             %a == b
192             fct = @interval.eq_element;
193             result = interval.elementwiseoperator(fct,a,b);
194         end
195
196         function result = inverse_elementwise(this)
197             fct = @interval.inverse_element;
198             result = interval.elementwiseoperator(fct,this);
199         end
200
201         function result = sqrt(a)
202             %sqrt(a)
203             fct = @interval.power_element;
204             result = interval.elementwiseoperator(fct,a,1/2);
205         end
206         function result = nthroot(a,n)
207             %nthroot(a,n)
208             fct = @interval.power_element;
209             result = interval.elementwiseoperator(fct,a,n^(-1));
210         end
211
212         function result = abs(a)
213             %abs(a)
214             fct = @interval.abs_element;
215             result = interval.elementwiseoperator(fct,a);
216         end
217
218         function result = norm(a)
219             %norm(a)
220             result = norm_p(a,2);
221         end
222         function result = norm_p(a,p)
223             %nthroot(a,b)
224             if length(size(a)) == 2 && length(size(p)) == 2 && all(size(p) == [1,1])
225                 sizes = size(a);
226                 if sizes(1) == 1 %row vector
227                     result = sum(a.^p,2)%^(1/p);
228                 elseif sizes(2) == 1 %column vector
229                     result = sum(a.^p,1)%^(1/p);
230                 else %matrix
231                     error('matrix norm not yet implemented')
232                 end
233             elseif length(size(a)) ~= 2
234                 error('invalid input a for norm_p(a,p)')
```

```
235             else
236                 error('norm_p(a,b) function requires a single scalar p')
237             end
238         end
239 
240         function disp(this)
241             disp('Interval');
242             dispmat = interval.int2mat(this);
243             disp(dispmat);
244         end
245 
246         function str = int2str(this)
247             str = mat2str(interval.int2mat(this));
248         end
249 
250         function result = summinmax(this,directions,to_be_evaluated)
251             if nargin<2
252                 directions = 1;
253             end
254             s = size(this);
255             d = length(s);
256             snew = s;
257             snew(directions) = ones(1,length(directions));
258             result = interval(zeros(snew));
259             specifieddirections = false(1,d);
260             specifieddirections(directions) = true(1,length(directions));
261 
262             vLim = s;
263             vLim1 = s;
264             vLim1(directions) = [];
265 
266             v1    = ones(1, length(vLim1));
267             ready = false;
268             while ~ready
269                 Index1 = arrayindexing.sub2indV(vLim1, v1);
270 
271                 vLim2 = s(directions);
272                 v2    = ones(1, length(vLim2));
273                 ready = false;
274                 while ~ready
275                     v = zeros(1,d);
276                     v(~specifieddirections) = v1;
277                     v(specifieddirections) = v2;
278                     Index = arrayindexing.sub2indV(vLim, v);
279                     eval(to_be_evaluated);
280                     % Update the index vector:
281                     [v2,ready] = arrayindexing.updateindexvec(v2,vLim2);
282                 end
283                 % Update the index vector:
284                 [v1,ready] = arrayindexing.updateindexvec(v1,vLim1);
285             end
286         end
287         function result = sum(this,directions)
288             to_be_evaluated = 'result(Index1) = result(Index1)+this(Index);';
289             result = summinmax(this,directions,to_be_evaluated);
290         end
291         function result = min(this,varargin)
292             to_be_evaluated = 'result(Index1) = result(Index1)+this(Index);';
293             result = summinmax(this,directions,to_be_evaluated);
294         end
295         %max
```

```
296 %           function result = max(this,varargin)
297 %           end
298 
299         function result = cross(a,b)
300             %cross(a,b)
301             if numel(a) == 3 && numel(b) == 3 %length(size(a)) == 2 && length(size(b)) == 2 &&
            ↪  all(size(a) == [1,3]) && all(size(b) == [1,3])
302                 result = [(a(2)*b(3)) - (a(3)*b(2)), (a(3)*b(1)) - (a(1)*b(3)), (a(1)*b(2)) -
                ↪  (a(2)*b(1))];
303             else
304                 error('cross function is only implemented for two interval vectors of size 3')
305             end
306         end
307 
308         %det
309         function result = det(this)
310             if length(size(this)) == 2 && all(size(this) == [2,2])
311                 result = this(1)*this(4)-this(2)*this(3);
312             else
313                 error('det function has to be implemented for non-2x2 matrices')
314             end
315         end
316         %cap
317         %cup
318         function result = cup(this,b)
319             fct = @interval.cup_element;
320             result = interval.elementwiseoperator(fct,this,b);
321         end
322 
323         function result = sign(this)
324             fct = @interval.sign_element;
325             result = interval.elementwiseoperator(fct,this);
326         end
327 
328         function result = extractbounds(this)
329             s = size(this);
330             n = prod(s);
331             result = zeros(n,2);
332             for i = 1:n
333                 result(i,:) = this(i).bounds;
334             end
335             if s(end) == 1
336                 s(end) = [];
337             end
338             result = reshape(result,[s,2]);
339         end
340 
341         function result = subset(a,b)
342             fct = @interval.subset_element;
343             result = all(interval.elementwiseoperator(fct,a,b));
344         end
345         function result = superset(this,bi)
346             %this superset of b
347             result = bi.subset(this);
348         end
349 
350         function result = scale(this,factor)
351             result = interval(zeros(size(this)));
352             s = 1+(factor-1)/2;
353             for i=1:numel(this)
354                 result(i) = interval(s*this(i).bounds(1)+(1-s)*this(i).bounds(2),(1-s)*this(i
                ↪  ).bounds(1)+s*this(i).bounds(2));
```

```
355                 end
356             end
357             function result = linspace(this,n)
358                 result = zeros([numel(this),n]);
359                 for i=1:numel(this)
360                     result(i,:) = linspace(this(i).bounds(1),this(i).bounds(2),n);
361                 end
362                 s = size(this);
363                 if length(s) == 2 && s(2) == 1
364                     s(2) = [];
365                 end
366                 result = reshape(result,[s,n]);
367             end
368 
369             %%%%%%%%%%%%% functions which need revision
370     %
371     %         function result = mrdivide(a,b)
372     %             %a/b
373     %             %%%%%%%%%%% Be careful if 0 is contained in numerator and denominator
374     %             result = a*b.inverseinterval;
375     %         end
376     % %         function result = mpower(a,b)
377     % %             %a^b
378     % % %             res1 =
379     % % %             result = a.interval+b.interval;
380     % %         end
381         end
382 
383 
384 
385         methods(Static)
386             function result = elementwiseoperator(fct,a,b)
387                 %Computes elementwise: fct(a,b) or fct(a) if b is not defined
388                 %Possible outputtupes are: 'interval' or 'logical'
389                 %allows for arrays a and b to have only 1 element while the other doesn't, similar
                    ↪  to 1+[2,3] = [3,4]
390 
391                 s = size(a);
392                 if isa(a,'double')
393                     a = interval(a);
394                 end
395                 if nargin >= 3
396                     if numel(a) == 1
397                         s = size(b);
398                     end
399                     if isa(b,'double')
400                         b = interval(b);
401                     end
402                 end
403                 snum = prod(s); %#ok<NASGU>
404                 for ind = prod(s):-1:1
405                     if nargin <= 2
406                         input = {a(ind).bounds};
407                     else
408                         if numel(a) == 1
409                             input = {a.bounds,b(ind).bounds};
410                         elseif numel(b) == 1
411                             input = {a(ind).bounds,b.bounds};
412                         else
413                             input = {a(ind).bounds,b(ind).bounds};
414                         end
```

```
415                 end
416                 result(ind) = fct(input{:});
417             end
418             result = reshape(result,s);
419         end
420 
421         function result = plus_element(a,b)
422             %a+b
423             if (a(2)<a(1) && b(2)<b(1)) || (a(1)+b(1) <= a(2)+b(2) && (a(2)<a(1) || b(2) <
            ↪  b(1)))
424                 bounds = [-inf,inf];
425             else
426                 bounds = a+b;
427             end
428             result = interval(bounds(1),bounds(2));
429         end
430         function result = uminus_element(a)
431             %-a
432             result = interval(-a(2),-a(1));
433         end
434         function result = times_element(a,b)
435             %a*b
436             values = a'*b;
437             if a(1) <= a(2) && b(1) <= b(2)
438                 bounds = [min(values(:)),max(values(:))];
439             else
440                 %make a contain infinity
441                 if a(1) <= a(2)
442                     c = a;
443                     a = b;
444                     b = c;
445                 end
446                 if interval.subset_element([0,0],b) || (b(2) < b(1) &&
                ↪  interval.subset_element([0,0],a)) %0 in a or b
447                     bounds = [-inf,inf];
448                 elseif b(2) < b(1) && ~interval.subset_element([0,0],a) &&
                ↪  ~interval.subset_element([0,0],b) %0 notin a or b
449                     bounds = [min(values(values>0)),max(values(values<0))];
450                 else
451                     if interval.subset_element([0,0],b)
452                         c = a;
453                         a = b;
454                         b = c;
455                     end
456                     if ~interval.subset_element([0,0],a)
457                         if 0 < b(1) %b positive
458                             bounds = [a(1)*b(1),a(2)*b(1)];
459                         else %b negative
460                             bounds = [a(2)*b(2),a(1)*b(2)];
461                         end
462                     else
463                         if 0<a(2) %0 < complement(a)
464                             if 0 < b(1) %b positive
465                                 bounds = [a(1)*b(1),a(2)*b(2)];
466                                 if bounds(2)>bounds(1)
467                                     bounds = [-inf,inf];
468                                 end
469                             else %b negative
470                                 bounds = [a(2)*b(1),a(1)*b(2)];
471                                 if bounds(2)>bounds(1)
472                                     bounds = [-inf,inf];
```

```
473                                 end
474                             end
475                         else %complement(a) < 0
476                             if 0 < b(1) %b positive
477                                 bounds = [a(1)*b(2),a(2)*b(1)];
478                                 if bounds(2)>bounds(1)
479                                     bounds = [-inf,inf];
480                                 end
481                             else %b negative
482                                 bounds = [a(1)*b(1),a(2)*b(2)];
483                                 if bounds(2)>bounds(1)
484                                     bounds = [-inf,inf];
485                                 end
486                             end
487                         end
488                     end
489                 end
490             end
491             result = interval(bounds(1),bounds(2));
492         end
493 
494 
495 
496 
497 
498         function result = inverse_element(a)
499             %Computes the inverse interval a^-1
500             result = interval(1/a(2),1/a(1));
501         end
502 
503         function result = abs_element(a)
504             %|a|
505             if a(1) <= a(2) %[a(1), a(2)]
506                 if a(1) >= 0
507                     result = interval(a(1), a(2));
508                 elseif a(2) <= 0
509                     result = interval(-a(2), -a(1));
510                 else %a(1) < 0 < a(2)
511                     result = interval(0, max(-a(1),a(2)));
512                 end
513             else %[-inf, a(2)] \cup [a(1), inf]
514                 if a(2) >= 0 || a(1) <= 0
515                     result = interval(0, inf);
516                 else
517                     result = interval(min(abs(a(1)), abs(a(2))), inf);
518                 end
519             end
520         end
521 
522         function result = power_element(a,b)
523             %Computes a^b assuming b is a real number
524             %and a is non-negative when b is not an integer
525             if a(2) >= a(1)
526                 if b(1) == b(2) %b can be treated as a single real number
527                     b = b(1);
528                     if rem(b(1),1) == 0 %b is an integer
529                         if b >= 0
530                             if rem(b,2) == 0 && a(1) <= 0 && 0 <= a(2)
531                                 result = interval(0,max(a(1)^b,a(2)^b));
532                             else
533                                 res1 = a(1)^b;
```

```
534                             res2 = a(2)^b;
535                             result = interval(min(res1,res2),max(res1,res2));
536                         end
537                     else % b < 0
538                         if rem(b,2) == 0 && a(1) <= 0 && 0 <= a(2)
539                             result = interval(min(a(1)^b,a(2)^b), inf);
540                         elseif a(1) <= 0 && 0 <= a(2) %rem(b,2) == 1
541                             result = interval(-inf, inf);
542                         else %a does not include 0
543                             res1 = a(1)^b;
544                             res2 = a(2)^b;
545                             result = interval(min(res1,res2),max(res1,res2));
546                         end
547                     end
548                 elseif a >= 0 %b is not an integer, but a is non-negative
549                     res1 = a(1)^b;
550                     res2 = a(2)^b;
551                     result = interval(min(res1,res2),max(res1,res2));
552                 else
553                     error("interval method a.^b not implemented for non-integer b and
                    ↪  negative a")
554                 end
555             else
556                 error("interval method a.^b not implemented for an interval b")
557             end
558         else %a(1) > a(2) (a = [-\inf,a(2)] \cup [a(1),\inf])
559             result = cup(interval(-inf,a(2)).^2, interval(a(1),inf).^2);
560         end
561     end
562 
563     function result = lt_element(a,b)
564         %a < b
565         result = (a(1)<=a(2)) && (b(1)<=b(2)) && a(2)<b(1);
566     end
567     function result = le_element(a,b)
568         %a <=b
569         result = (a(1)<=a(2)) && (b(1)<=b(2)) && a(2)<=b(1);
570     end
571     function result = eq_element(a,b)
572         %a == b
573         result = all(abs(a - b) == 3*eps);
574     end
575 
576     function result = cup_element(a,b)
577         if a(2) >= a(1) && b(2) >= b(1)
578             % [a(1), a(2)] \cup [b(1), b(2)]
579             result = interval(min(a(1),b(1)), max(a(2),b(2)));
580         elseif a(2) >= a(1) && b(2) < b(1)
581             % [a(1), a(2)] \cup ([-inf, b(2)] \cup [b(1), inf])
582             if (a(1) <= b(2) && a(2) >= b(1)) || (a(1) >= b(2) && a(2) <= b(1))
583                 result = interval(-inf, inf);
584             elseif a(1) <= b(2) &&  a(2) < b(1)
585                 result = interval(b(1), a(2));
586             else % a(1) > b(2) &&  a(2) >= b(1)
587                 result = interval(a(1), b(2));
588             end
589         elseif a(2) < a(1) && b(2) >= b(1)
590             % ([-inf, a(2)] \cup [a(1), inf]) \cup [b(1), b(2)]
591             %symmetric to the previous case
592             c = a;
593             a = b;
```

```
594                 b = c;
595                 if (a(1) <= b(2) && a(2) >= b(1)) || (a(1) >= b(2) && a(2) <= b(1))
596                     result = interval(-inf, inf);
597                 elseif a(1) <= b(2) &&  a(2) < b(1)
598                     result = interval(b(1), a(2));
599                 else % a(1) > b(2) &&  a(2) >= b(1)
600                     result = interval(a(1), b(2));
601                 end
602             else % a(2) < a(1) && b(2) < b(1)
603                 % ([-inf, a(2)] \cup [a(1), inf]) \cup ([-inf, b(2)] \cup [b(1), inf])
604                 if max(a(2), b(2)) >= min(a(1), b(1))
605                     result = interval(-inf, inf);
606                 else
607                     result = interval(min(a(1),b(1)), max(a(2), b(2)));
608                 end
609             end
610             %to be checked
611         end
612 
613         function result = sign_element(a)
614             result = interval(min(sign(a(1)), sign(a(2))), max(sign(a(1)), sign(a(2))));
615         end
616 
617 
618 
619 
620 
621 
622 
623 
624 
625         function result = subset_element(a,b)
626             %a subset of b
627             if a(1)<=a(2) && b(1) <= b(2)
628                 result = b(1) <= a(1) && a(2) <= b(2);
629             elseif a(1)<=a(2) && ~(b(1) <= b(2))
630                 result = a(2) <= b(1) || b(2) <= a(1);
631             elseif ~(a(1)<=a(2)) && b(1) <= b(2)
632                 result = false;
633             elseif ~(a(1)<=a(2)) && ~(b(1) <= b(2))
634                 result = a(1)<=b(1) && b(2)<=a(2);
635             end
636         end
637 
638 
639 
640 
641 
642         function Index = sub2indV(Vlim,X)
643             k     = [1, cumprod(Vlim)];
644             Index = sum(k(1:length(X)) .* (X - 1)) + 1;
645         end
646         function v = ind2subV(Vlim, ind)
647             ind = ind-1;
648             v = zeros(1,0);
649             for i = 1:length(Vlim)
650                 v(i) = 1+mod(ind,Vlim(i));
651                 ind = (ind-v(i)+1)/Vlim(i);
652             end
653         end
654         function mat = int2mat(I)
```

```
655             %Gather the interval bounds in a matrix
656             sizeI = size(I);
657             if sizeI(end) == 1
658                 sizeI(end) = [];
659             end
660             mat = zeros([sizeI,2]);
661             num = numel(I);
662             for i = 1:num
663                 if ~isempty(I(i).bounds)
664                     mat([i,i+num]) = I(i).bounds;
665                 else
666                     mat = ['empty ',num2str(sizeI),' array of intervals'];
667                 end
668             end
669         end
670 
671         function int = zeros(varargin)
672             sizes = cell2mat(varargin);
673             int = interval(zeros(sizes));
674         end
675         function int = ones(varargin)
676             sizes = cell2mat(varargin);
677             int = interval(ones(sizes));
678         end
679         function int = unit(varargin)
680             sizes = cell2mat(varargin);
681             if length(sizes) == 1
682                 A = zeros([sizes,1]);
683                 B = ones([sizes,1]);
684             else
685                 A = zeros(sizes);
686                 B = ones(sizes);
687             end
688             int = interval(A,B);
689         end
690 
691 
692         function test()
693             interval.test1;
694         end
695         function test1()
696             a = interval(0,1);
697             b = interval(3,4);
698             disp(a+b);
699             disp(a-b);
700             disp(a.*b);
701             disp(a./b);
702         end
703     end
704 end
705 
706 
```

A.6 Code defining generic boxes and functions on boxes

```
classdef box < handle & matlab.mixin.Copyable
    %box class for subdivision algorithms

    properties
        boxdimensions %D*2 vector with min and max coordinate of
            %the dimensions in each row[xmin,xmax; ymin,ymax,...]

        length0 %D*1 logical vector keeping track of the directions of 0 length
            %i.e. representing a lower dimensional box / face in D-dimensional space

        depth = 0;
    end
    properties (NonCopyable)
        parent %parent box
        children %2*2*...*2 array containing 2^d many children boxes
            %children(1,1,...,1) corresponds to the child with minimal
            %coordinates in each direction
            %d is the number of directions in which the box does not have 0 length

        plotboxhandle %handle to the plotted box boundary
    end

    methods
        function this = box(boxdimensions,length0)
            %box(boxdimensions)
            if nargin > 0
                this.boxdimensions = boxdimensions;
                if nargin > 1
                    this.length0 = length0;
                else
                    this.length0 = boxdimensions(:,1) == boxdimensions(:,2);
                end
            end
        end
        function disp(this)
            if length(this) == 1
                disp(this.boxdimensions);
            else
                disp([num2str(size(this)),' matrix of boxes']);
            end
        end
        function boxes = split(this)
            if ~isempty(this.children) || isempty(this.boxdimensions)
                boxes = [];
%                 warning("no box split performed");
                return
            end
            D = size(this.boxdimensions,1);
            d = D-sum(this.length0);

            childrenarry(2^d) = copy(this);
            if D == 1 || d == 1
                childrenarry = reshape(childrenarry,[2*ones(1,d),1]);
            else
                childrenarry = reshape(childrenarry,2*ones(1,d));
```

```
56              end
57
58              vLim = 2*ones(1,d);
59              v    = ones(1, d);
60              ready = false;
61              while ~ready
62                  Index = arrayindexing.sub2indV(vLim, v);
63                  %If v(i) = 1, then the child will get the smaller coordinate in dimension i
64                  vcomplete = ones(1,D);
65                  vcomplete(~this.length0) = v;
66                  childrenarry(Index).boxdimensions =
                    ↪  [this.boxdimensions(:,1),sum(this.boxdimensions,2)/2]+((vcomplete-1)'.*(⌋
                    ↪  this.boxdimensions(:,2)-this.boxdimensions(:,1))/2)*[1,1];
67                  childrenarry(Index).length0 = this.length0;
68                  childrenarry(Index).depth = this.depth+1;
69                  childrenarry(Index).parent = this;
70
71                  % Update the index vector:
72                  [v,ready] = arrayindexing.updateindexvec(v,vLim);
73              end
74
75              this.children = childrenarry;
76              this.plotboxhandle = [];
77              boxes = childrenarry(:)';
78          end
79
80          function c = center(this)
81              c = sum(this.boxdimensions,2)/2;
82          end
83          function r = radius(this)
84              r = norm(this.center-this.boxdimensions(:,1));
85          end
86          function C = corners(this)
87              D = size(this.boxdimensions,1);
88              d = D-sum(this.length0);
89              C = zeros(D,2^d);
90
91              for i = 1:2^d
92                  v = arrayindexing.ind2subV(2*ones(1,d),i);
93                  corner = zeros(D,1);
94                  corner(this.length0) = this.boxdimensions(this.length0,1);
95                  vind = 0;
96                  for j = 1:D
97                      if ~this.length0(j)
98                          vind = vind+1;
99                         corner(j) = (v(vind)==1)*this.boxdimensions(j,1)+(v(vind)==2)*this.box⌋
                          ↪  dimensions(j,2);
100                     end
101                 end
102                 C(:,i) = corner;
103             end
104         end
105         function f = facets(this)
106             D = size(this.boxdimensions,1);
107             d = D-sum(this.length0);
108             dind = d;
109             for i = D:-1:1
110                 if ~this.length0(i)
111                     for j = 2:-1:1
112                         B = copy(this);
113                         B.boxdimensions(i,3-j) = B.boxdimensions(i,j);
```

```
114                     B.length0 = this.length0;
115                     B.length0(i) = true;
116                     f(dind,j) = B;
117                 end
118                 dind = dind-1;
119             end
120         end
121     end
122 
123     function Bscaled = scale(this,factor)
124         classtype = class(this);
125         scaledboxdimensions =
        ↪   interval(this.boxdimensions(:,1),this.boxdimensions(:,2)).scale(factor);
126         extractedbounds = scaledboxdimensions.extractbounds; %#ok<NASGU>
127         Bscaled = eval([classtype,'(scaledboxdimensions.extractbounds);']);
128         Bscaled.length0 = this.length0;
129     end
130 
131     %Returns a box without the removed dimension
132     function Bnew = removedim(this,dim)
133         classtype = class(this);
134         dims = this.boxdimensions;
135         len0 = this.length0;
136         dims(dim,:) = [];
137         len0(dim,:) = [];
138         Bnew = eval([classtype,'(dims);']);
139         Bnew.length0 = len0;
140     end
141 
142     %Returns a box with an inserted dimension at the specified location
143     %Inserts to the first/last position if the requested position is
144     %out of bounds
145     function Bnew = insertdim(this,dim,rowdims)
146         classtype = class(this);
147         dims = this.boxdimensions;
148         len0 = this.length0;
149         if dim < 1
150             dim = 1;
151         elseif dim > length(this.boxdimensions) + 1
152             dim = length(this.boxdimensions) + 1;
153         end
154         dims = [dims(1:dim-1,:); rowdims; dims(dim:end,:)];
155         len0 = [len0(1:dim-1,:); rowdims(1) == rowdims(2); len0(dim:end,:)];
156         Bnew = eval([classtype,'(dims);']);
157         Bnew.length0 = len0;
158     end
159 
160     function Q = leaves(this)
161         if isempty(this.children)
162             Q = this;
163         else
164             Q = [];
165             for i = 1:numel(this.children)
166                 Q = [Q,leaves(this.children(i))]; %#ok<*AGROW>
167             end
168         end
169     end
170     function int = interval(this)
171         int = interval(this.boxdimensions(:,1),this.boxdimensions(:,2));
172     end
173 
```

```
174         function str = box2str(this)
175             str = int2str(this.interval);
176         end
177 
178         function Bcap = cap(this,B)
179             %to be implemented
180             Bcap = this;
181         end
182         function Bcup = cup(this,B)
183             %to be implemented
184             Bcup = this;
185         end
186 
187         function varargout = coord(this)
188             %Splits the box into its coordinate interval components
189             %For 3D boxes:
190             %[x,y,z] = B.splitcoordinates
191             D = size(this.boxdimensions,1);
192             varargout = cell(D,1);
193             intvec = this.interval;
194             output = mat2cell(intvec(:),ones(D,1))';
195             [varargout{:}] = deal(output{:});
196         end
197         function plotsubdivision(this,ax,varargin)
198             if isempty(this.children)
199                 this.plotbox(ax,varargin{:});
200             else
201                 for i = 1:numel(this.children)
202                     this.children(i).plotsubdivision(ax,varargin{:});
203                 end
204             end
205         end
206         function h = plotbox(this,ax,varargin)
207             D = size(this.boxdimensions,1);
208             d = D-sum(this.length0);
209             switch d
210                 case 1
211                     %1-dimensional box possibly in higher dimensions
212                     corners = this.corners;
213                     switch D
214                         case {1,2}
215                             h = scatter(corners(1,:),corners(2,:),varargin);
216                         case 3
217                             h = scatter3(corners(1,:),corners(2,:),corners(3,:),varargin);
218                     end
219                 case 2
220                     %plot the box's edges
221                     corners = this.corners;
222                     corners = corners(:,[1,2,4,3,1]);
223                     switch D
224                         case 2
225                             h = plot(ax,corners(1,:),corners(2,:),varargin{:});
226                         case 3
227                             h = plot3(ax,corners(1,:),corners(2,:),corners(3,:),varargin{:});
228                     end
229                 case 3
230                     %plot the box's edges
231                     corners = this.corners;
232                     %collect the vertex indices making up the edges
233                     eind = [1,2;...
234                             2,4;...
```

```
235                                 4,3;...
236                                 3,1;...
237                                 5,6;...
238                                 6,8;...
239                                 8,7;...
240                                 7,5;...
241                                 1,5;...
242                                 2,6;...
243                                 3,7;...
244                                 4,8];
245                         XYZ = zeros(3,36);
246                         for i = 1:12
247                             XYZ(:,[3*i-2,3*i-1,3*i]) = [corners(:,eind(i,:)),[NaN;NaN;NaN]];
248                         end
249                         h = plot3(ax,XYZ(1,:),XYZ(2,:),XYZ(3,:),varargin{:});
250                 end
251                 this.plotboxhandle = h;
252             end
253             function zoom(this,ax)
254                 plotinterval = this.boxdimensions;
255                 plotinterval(this.length0,:) = this.boxdimensions(this.length0,:)+0.5*[-ones(sum(
                ↪  this.length0),1),ones(sum(this.length0),1)];
256                 plotinterval = plotinterval';
257                 plotinterval = reshape(plotinterval(:),1,[]);
258                 axis(ax,plotinterval);
259             %                   r1 = boundingbox(1,2)-boundingbox(1,1);
260             %                   r2 = boundingbox(2,2)-boundingbox(2,1);
261             %                   pbaspect([r1, r2, max(r1,r2)])
262                 daspect(ax,[1 1 1]);
263             end
264     %         function fillbox(this,axeshandle,color)
265     %             switch size(this.boxdimensions,1)
266     %                 case 2
267     %                   X = [this.boxdimensions(1,1);this.boxdimensions(1,2);this.boxdimensions(
        ↪  1,2);this.boxdimensions(1,1);this.boxdimensions(1,1)];
268     %                   Y = [this.boxdimensions(2,1);this.boxdimensions(2,1);this.boxdimensions(
        ↪  2,2);this.boxdimensions(2,2);this.boxdimensions(2,1)];
269     %                     this.filling = fill(axeshandle,X,Y,color);
270     %                 case 3
271     %             end
272     %         end
273
274             function delete(this)
275                 if ~isempty(this.children)
276                     while ~isempty(this.children)
277                         this.children(1).delete;
278                         this.children(1) = [];
279                     end
280                 end
281                 if ~isempty(this.parent)
282                     this.parent = [];
283                 end
284             end
285
286
287             %automatically transform into an interval for interval computations
288             function result = plus(a,b)
289                 %a+b
290                 [a,b] = this.convert2interval(a,b);
291                 result = a+b;
292             end
```

```
293         function result = minus(a,b)
294             %a-b
295             [a,b] = this.convert2interval(a,b);
296             result = a-b;
297         end
298         function result = uminus(a)
299             %-a
300             [a] = this.convert2interval(a);
301             result = -a;
302         end
303         function result = uplus(a)
304             %+a
305             [a] = this.convert2interval(a);
306             result = a;
307         end
308         function result = times(a,b)
309             %a.*b
310             [a,b] = this.convert2interval(a,b);
311             result = a.*b;
312         end
313         function result = mtimes(a,b)
314             %a*b
315             [a,b] = this.convert2interval(a,b);
316             result = a.*b;
317         end
318 %           function result = rdivide(a,b)
319 %           end
320 
321     end
322     methods(Static)
323         function varargout = convert2interval(varargin)
324             n = length(varargin);
325             varargout = cell(1,n);
326             for i = 1:n
327                 if isnumeric(varargin{1}) || isa(varargin{1},"interval")
328                     varargout{i} = varargin{1};
329                 else
330                     varargout{i} = varargin{1}.interval;
331                 end
332             end
333         end
334         function [B,fig,ax] = test
335             B = box([-1,1;-1,1;0,0],[false,false,true]);
336             [fig,ax] = box.testsplit(B);
337         end
338         function [fig,ax] = testsplit(B)
339             [fig,ax] = createfigure(B);
340             B.split;
341             B.children(1).split;
342             B.children(1).children(2).split;
343             B.plotsubdivision(ax,'k');
344         end
345     end
346 end
```

Bibliography

[1] R.E. Moore. *Interval Analysis*. Prentice-Hall series in automatic computation. Prentice-Hall, 1966. URL: https://books.google.com.tr/books?id=csQ-AAAAIAAJ.

[2] Morris Hirsch. *Differential Topology*. New York: Springer-Verlag, 1976. ISBN: 0387901485.

[3] Helmut Ratschek and Jon G. Rokne. "Computer Methods for the Range of Functions". In: 1984.

[4] William E. Lorensen and Harvey E. Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *SIGGRAPH Comput. Graph.* 21.4 (Aug. 1987), pp. 163–169. ISSN: 0097-8930. DOI: 10.1145/37402.37422. URL: https://doi.org/10.1145/37402.37422.

[5] Shreeram Shankar Abhyankar and Chanderjit J Bajaj. "Automatic parameterization of rational curves and surfaces IV: algebraic space curves". In: *ACM Transactions on Graphics (TOG)* 8.4 (1989), pp. 325–334.

[6] Michael N Vrahatis. "A short proof and a generalization of Miranda's existence theorem". In: *Proceedings of the American Mathematical Society* 107.3 (1989), pp. 701–703.

[7] Steven George Krantz and Harold R Parks. *The implicit function theorem: history, theory, and applications*. Springer Science & Business Media, 2002.

[8] Simon Plantinga and Gert Vegter. "Isotopic Approximation of Implicit Curves and Surfaces". In: *Proceedings of the 2004 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*. SGP '04. Nice, France: Association for Computing Machinery, 2004, pp. 245–254. ISBN: 3905673134. DOI: 10.1145/1057432.1057465. URL: https://doi.org/10.1145/1057432.1057465.

[9] Michael Burr et al. "Complete subdivision algorithms, II: isotopic meshing of singular algebraic curves." In: Jan. 2008, pp. 87–94.

[10] Long Lin and Chee Yap. "Adaptive Isotopic Approximation of Nonsingular Curves: the Parametrizability and Non-local Isotopy Approach". In: *Proc. 25th ACM Symp. Computational Geometry*. University of Aarhus, Denmark, Jun 8-10, 2009. Invited for Special Conference Issue of Discrete and Combinatorial Geometry. June 2009, to appear.

[11] Long Lin and Chee Yap. "Adaptive Isotopic Approximation of Nonsingular Curves and Surfaces". In: (June 2014).

[12] Oswaldo de Oliveira. "The Implicit and the Inverse Function Theorems: Easy Proofs". In: *Real Analysis Exchange* 39.1 (2014), p. 207. DOI: 10.14321/realanalexch.39.1.0207. URL: https://doi.org/10.14321%2Frealanalexch.39.1.0207.

[13] Juan Xu and Chee Yap. "Effective Subdivision Algorithm for Isolating Zeros of Real Systems of Equations, with Complexity Analysis". In: arXiv, 2019. DOI: 10.48550/ARXIV.1905.03505. URL: https://arxiv.org/abs/1905.03505.