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
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
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Box Predicates</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Jacobian Condition in Planar Curve Intersection</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Geometric Consequence of the Jacobian Condition</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>The Miranda Conditions</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>A Miranda-Type Theorem for Space Curves</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Preconditioning</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Algorithm Design</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Curve-Isolating Subdivision Algorithm</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>Reconstruction Algorithm</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>Implementation and Experiments</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Implementation</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>Experiments and Results</td>
<td>34</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Experiment 1</td>
<td>35</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Experiment 2</td>
<td>38</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Experiment 3</td>
<td>41</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Experiment 4</td>
<td>44</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Experiment 5</td>
<td>47</td>
</tr>
<tr>
<td>5</td>
<td>Conclusion</td>
<td>51</td>
</tr>
<tr>
<td>A</td>
<td>Appendix</td>
<td>52</td>
</tr>
<tr>
<td>A.1</td>
<td>Main code for the implemented algorithms</td>
<td>52</td>
</tr>
<tr>
<td>A.2</td>
<td>Code for the cycle finding algorithm used in reconstruction</td>
<td>59</td>
</tr>
<tr>
<td>A.3</td>
<td>Code for the path finding algorithm used in reconstruction</td>
<td>60</td>
</tr>
<tr>
<td>A.4</td>
<td>Code for the implemented tests</td>
<td>61</td>
</tr>
<tr>
<td>A.5</td>
<td>Code defining intervals and interval arithmetic</td>
<td>68</td>
</tr>
</tbody>
</table>
## 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 $f_1, f_2 : \mathbb{R}^3 \rightarrow \mathbb{R}$ are smooth and non-singular in $B_0$. We use the notation $f = (f_1, f_2)$ - for which the traced curve can be defined as the set of points given by $S = f^{-1}(0) = \{ p \in \mathbb{R}^3, f(p) = 0 \} = f_1^{-1}(0) \cap f_2^{-1}(0)$.

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

\[ \forall p \in B_0. \ (f_1(p) = 0 \implies \nabla f_1(p) \neq 0) \land (f_2(p) = 0 \implies \nabla f_2(p) \neq 0) \]

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 $S = f^{-1}(0)$ 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 $\Box f_1, \Box f_2, \Box \nabla f_1, \Box \nabla f_2$. 
3. We assume that the surfaces given by \( f_1 = 0 \) and \( f_2 = 0 \) never intersect tangentially in \( B_0 \).

\[
(\forall p \in B_0, \forall c \in R. \ f_1(p) = f_2(p) = 0 \implies \nabla f_1(p) \neq c\nabla f_2(p))
\]

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 \( p \in \mathbb{R}^m \) satisfying the system of equations \( f(p) = (f_1(p), f_2(p), \ldots, f_n(p)) = 0^n = 0 \) where \( f_1, f_2, \ldots, f_n \) 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

\[
y: [0, 1] \times \mathbb{R}^3 \to \mathbb{R}^3
\]
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. \( \gamma_0 \) is the identity function;

3. \( \gamma_1(S) = S' \).

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

If the continuous map
\[
\gamma : [0, 1] \times S \to \mathbb{R}^3
\]
satisfies (i)-(iii), then we say that \( \gamma \) 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 $f : X \rightarrow \mathbb{R}$ 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 $f : X \rightarrow \mathbb{R}, X \in I$ 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 \(\Box f\) 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 \(\Box f(B_i) \to f(p)\) as \(i \to \infty\). ([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 $F : \mathbb{R}^2 \rightarrow \mathbb{R}$ be an implicit function, and $B \subseteq \mathbb{R}^2$ be a square bounding box. Also, let $□g$ 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 $\nabla F$ 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 ($\langle \Box\nabla F(C), \Box\nabla F(C) \rangle > 0$) 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 $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ 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 $C_0(B) = 0 \not\in f(B)$ with the following algorithm:
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:

\begin{figure}[h]
\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Abstract MK Test} \\
\hline
\textbf{Input:} $f$ and box $B$ \\
\textbf{Output:} true iff $\text{MK}_f(B)$ succeeds \\
1. $C \leftarrow J_f(m(B))$, Jacobian matrix at $m(B)$ \\
   If $C^{-1}$ does not exist, return false. \\
2. Construct a “preconditioned version” $g$: \\
   $g \leftarrow C^{-1}f = (g_1(x), \ldots, g_n(x))$ \\
3. Apply the Simple Miranda Test to $2B$ for $g$: \\
   For $i \leftarrow 1, \ldots, n$: \\
   \begin{align*}
   &\text{If } g_i(2B_i^+) \leq 0 \text{ or } g_i(2B_i^-) \geq 0, \quad (*) \\
   &\text{return false}
   \end{align*}
4. Return true.
\end{tabular}
\end{center}
\end{figure}

\textbf{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 $C_0(B) = C^f_0(B)$ holds iff $C^f_1(B) \lor C^f_2(B)$ where

  $$C^f_0(B) \equiv \left[ 0 \notin f_j(B) \right] \quad (j = 1, 2). \quad (2.1)$$

  If $C^f_0(B)$ holds, then the curve $f^{-1}(0)$ 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 $C_1(B) = C^f_1(B)$ holds iff $C^f_1(B) \land C^f_2(B)$ where

  $$C^f_1(B) \equiv \left[ 0 \notin \sum_{i=1}^{3} (D_i f_j(B))^2 \right] \quad (2.2)$$

  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 $S^2 = \{s^2 : s \in S\}$). 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 $f_1^{-1}(0) \cap f_2^{-1}(0) = f^{-1}(0)$. The next condition will address this.

- **(Jacobian Conditions $JC_i$)**

First let

$$Df(x) := \begin{bmatrix}
(f_1)_x(x) & (f_1)_y(x) & (f_1)_z(x) \\
(f_2)_x(x) & (f_2)_y(x) & (f_2)_z(x)
\end{bmatrix}$$

(2.3)

denote a $2 \times 3$ 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 $p \subseteq \mathbb{R}^3$ is when the matrix $Df$ is full-rank\(^1\) when evaluated at $x := p$.

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

$$\Delta_1f := \det \begin{bmatrix}
(f_1)_y(x) & (f_1)_z(x) \\
(f_2)_y(x) & (f_2)_z(x)
\end{bmatrix}, \quad \Delta_2f := -\det \begin{bmatrix}
(f_1)_x(x) & (f_1)_z(x) \\
(f_2)_x(x) & (f_2)_z(x)
\end{bmatrix}, \quad \Delta_3f := \det \begin{bmatrix}
(f_1)_x(x) & (f_1)_y(x) \\
(f_2)_x(x) & (f_2)_y(x)
\end{bmatrix}.$$  

(2.4)

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

$$JC^f(B) \equiv \left[ \bigwedge_{i=1}^3 JC_i^f(B) \right]$$

(2.5)

where

$$JC_i^f(B) \equiv \left[ 0 \notin \Delta_i f(B) \right] \quad (i = 1, 2, 3).$$

(2.6)

Thus $JC^f(B)$ implies that $Df(p)$ is full-rank for each $p \in B$. 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 \( f = (f, g) : \mathbb{R}^2 \rightarrow \mathbb{R}^2 \). We prove the following lemma (to be used in subsequent sections):

**Lemma 2.1 (2D Jacobian Condition).** Let \( B \subseteq \mathbb{R}^2 \) and \( JC_2^f(B) \) holds, i.e.,

\[
0 \notin Jf(B) \iff \det \begin{bmatrix} D_x f(B) & D_y f(B) \\ D_x g(B) & D_y g(B) \end{bmatrix}.
\]  

(2.7)

Then \(|\text{Zero}(f, g) \cap B| \leq 1|.

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

\[
F'(t) = \frac{dF}{dt} = \nabla f(L(t)) \ast (b - a).
\]  

(2.8)

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

\[
0 = F(0) - F(1) = F'(\xi) = \nabla f(L(\xi)) \ast (b - a).
\]  

(2.9)

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

\[
0 = G(0) - G(1) = G'(\xi) = \nabla g(L(\xi)) \ast (b - a).
\]  

(2.10)

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

$$f = (f_1, ..., f_m) : \Omega \subseteq \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^m$$  \hspace{1cm} (2.11)

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

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

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

**Theorem 2.2** (The Implicit Function Theorem).

*Given $f$ as in 2.11, let $(a; b) \in \Omega \subseteq \mathbb{R}^m \times \mathbb{R}^n$ satisfy

$$f(a; b) = 0$$

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

Then there exists an open set $X \subseteq \mathbb{R}^m$ containing $a$, an open set $Y \subseteq \mathbb{R}^n$ containing $b$, and an (implicit) function $g : Y \to X$ such that:
For each \( y^* \in Y \), there is a unique \( x^* \in X \) such that \( f(x^*, y^*) = 0 \). In fact, \( x^* = g(y^*) \). It follows that \( g(b) = a \).

The function \( g : Y \to X \) is \( C^1 \) with Jacobian matrix \( Jg \) at any \( y^* \in Y \) given by

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

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 \( J\mathcal{C}_3^f(B) \) hold. If \( B \) contains a point \( p = (p_1, p_2, p_3) \) of the curve \( f^{-1}(0) \) then there exists an implicit \( C^1 \) function

\[
g : J_p \to \mathbb{R}^2
\]

where \( J_p \) is an open interval containing \( p_3 \) satisfying

(i) \( g(p_3) = (p_1, p_2) \).

(i) For all \( z \in J_p \),

\[
f(g(z), z) = 0.
\]

**Proof:** The condition \( J\mathcal{C}_3^f(B) \) implies that \( \Delta_3 f = \det \begin{bmatrix} (f_1)_x(x) & (f_1)_y(x) \\ (f_2)_x(x) & (f_2)_y(x) \end{bmatrix} \) is non-zero at \( x = p \) (see 2.4). To apply Theorem 2.2 (IFT Theorem), let \( m = 1, n = 2, f = (f_1, f_2), x = (x, y) \) and \( y = (z) \). If \( (a; b) = (p_1, p_2; p_3) = p \), then the hypothesis of IFT Theorem is satisfied, namely

\[
f(a; b) = 0, \quad \text{and} \quad \frac{\partial f}{\partial x}(p) = \Delta_3 f(p) \neq 0.
\]
We conclude that there exists an implicit function $g : X \to Y$ 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 $y \in Y$, $f(g(y); y) = 0$. The theorem follows if we rename $Y$ to be $J_p$, and view the range of $g$ to be $\mathbb{R}^2$. □

In other words, this lemma tells us that the set

$$\{(g(z), z) : z \in J_p\}$$

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

**Lemma 2.4** (No Loop under the Jacobian Condition).

If $JC^f(B)$ holds, then $B$ does not contain a closed curve (i.e., loop) of $f^{-1}(0)$.

**Proof.** Consider a box $B$ such that $JC^f(B)$ holds. Without loss of generality, assume that $JC^f_3(B)$ holds. Then, Suppose $B$ contains a loop $C \subseteq f^{-1}(0)$. Pick a point $p = (p_1, p_2, p_3) \in C$ where $p_3$ is maximum. Note that such a point exists since $C$ is contained in $B$. By Lemma 2.3, and $JC^f_3(B)$, 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 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 $i = 1, ..., n$) denote the pair of opposite facets\(^2\) that are normal to the $i$th axis. Moreover, if $\Pi_i(x) = x_i$ denote the projection to the $i$th coordinate, then assume that $x^-_i < x^+_i$ where $\Pi_i(B^-_i) = \{x^-_i\}$ and $\Pi_i(B^+_i) = \{x^+_i\}$. If $f = (f_1, ..., f_n) : \mathbb{R}^n \rightarrow \mathbb{R}^n$, the following box predicate

$$MT^f(B) \equiv \left[ \bigwedge_{i=1}^{n} (f_i(B^-_i) < 0 < f_i(B^+_i)) \right]$$

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

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

where $MK^f_i(B) = MK^f_i(B)$ is given by

$$MK^f_i(B) \equiv \left[ \bigwedge_{j=1}^{i-1} (f_j(B^-_j) < 0 < f_j(B^+_j)) \right] \wedge \left[ \bigwedge_{j=i+1}^{n} (f_{j-1}(B^-_j) < 0 < f_{j-1}(B^+_j)) \right].$$

(2.14)

Note that $MK^f_i(B)$ 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 \( f = (f_1, f_2) \), \( MK^f_3(B) \) is simply

\[
MK_3(B) = (f_1(B_1^-) < 0 < f_1(B_1^+)) \land (f_2(B_2^-) < 0 < f_2(B_2^+))
\]

Now, we present the following theorem:

**Theorem 2.5** (Miranda Theorem for Curves).

Let \( B \subseteq \mathbb{R}^3 \) and the following conditions hold:

\[
C_1^f(B) \land JC_3^f(B) \land MK_3^f(B).
\]

Then \( \text{Zero}(f, g) \cap B \) 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 \( C_1^f(B) \land JC_3^f(B) \land MK_3^f(B) \) 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 \( JC_3(B) \land MK_3(B) \), then

there exists a curve \( H : I \to \mathbb{R}^3 \) within \( f^{-1}(0) \), which connects the faces \( B_2^- \) and \( B_2^+ \).

**Proof:** Because of the standard 2-dimensional Miranda theorem, we know that there exists a point \( p \in B_2^+ \) such that \( f(p) = 0 \). By Lemma 2.2, there exists an open interval \( J_p \) containing \( p_3 \), and a differentiable function

\[
h_p : J_p \to \mathbb{R}^2
\]
such that $h_p(p_3) = (p_1, p_2)$ and, for all $z \in J_p$,
\[ f(h_p(z), z) = 0. \]

Among all open intervals $J_p$ with that property, let $J = (a, b)$ be one, which minimizes $a$. Note that the curve $H : J \to \mathbb{R}^3, z \mapsto f(h_p(z), z)$ 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 $q = (h(a), a)$ to derive an open interval $J_q$ together with a differentiable function $h_q$ similar as before. But this is a contradiction to the maximality of interval $J$ because also the longer interval $J \cup J_q$ could have been chosen together with the function $h$:

\[
 h : J \cup J_q, z \mapsto \begin{cases} 
 h_p(z) & \text{if } z > a \\
 h_q(z) & \text{otherwise}
\end{cases}
\]  

(2.17)

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

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 $C^\mathbf{f}_1(B) \land JC^\mathbf{f}_3(B) \land MK_3^\mathbf{f}(B)$.

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 $p_1 = (p_1, p_2, 1) \in B_z^+$ such that $f(p_1) = 0$. 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 $\text{Zero}(f)$ around $p_1$. Then, we can choose a point $p_2 = (g_1(1-\mu), 1-\mu) \in B$.
on the curve. This point is in \( B \) and thus we can apply the theorem again to get another function

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

whose graph is a parameterization of \( \text{Zero}(f) \) around the point \( p_2 \). Continuing in this way, we can construct a curve through the points \( p_1, p_2, ..., p_k \) (for some \( k \geq 1 \)) until the curve reaches the face \( B_{k}^{-} \).

Now, let \( \Gamma \) be such a curve.

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

This implies:

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

Moreover, as the curve \( \Gamma \) connects the faces \( B_2^{+} \) and \( B_2^{-} \), we have:

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

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

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

However, for the same \( z' \), we must also have: \( \exists x, y \in \mathbb{R}. \quad (x, y, z') \in \Gamma \) by (2.19). \((x, y) \neq (x', y') \) is known as \( p' \notin \Gamma \)
This implies that, for the box $B' = [-1, 1] \times [-1, 1] \times z'$ we must have:

$$|\text{Zero}(f, g) \cap B'| \geq 2$$

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

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

However, by Lemma 2.1, we know $\text{Zero}(f, g) \implies |\text{Zero}(f, g) \cap B'| \leq 1$

Thus, we have $JC_3^{f}(B') \land |\text{Zero}(f, g) \cap B'| \geq 2 \land (JC_3^{f}(B') \implies |\text{Zero}(f, g) \cap B'| \leq 1)$, which is a contradiction.

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

This completes the proof of the Miranda Theorem for Curves. □

21
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 $\delta_{ij}$ the Kronecker delta, i.e. $\delta_{ij} = 1$ if and only if $i = j$. Otherwise $\delta_{ij} = 0$. We use $\partial_i$ 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:

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

(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

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

(2.21)

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

22
Proof. The second equation holds to linearity of derivatives:

\[
\partial_i \tilde{f}_j(m) = e_j^T \frac{\partial \tilde{f}_1}{\partial \tilde{f}_2} \left[ \begin{array}{c} \partial_i \tilde{f}_1 \\ \partial_i \tilde{f}_2 \\ \partial_i \tilde{f}_3 \end{array} \right]_{(p)} = e_j^T \frac{\partial f_1}{\partial f_2} \cdot \left[ \begin{array}{c} \partial_i f_1 \\ \partial_i f_2 \\ \partial_i f_3 \end{array} \right]_{(p)}
\]

\[
= \delta_{ij}
\]

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

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

\[
\begin{bmatrix}
\tilde{f}_1 \\
\tilde{f}_2 \\
\tilde{f}_3 \end{bmatrix} = \left( \begin{bmatrix} \nabla f_1^T \\
\nabla f_2^T \\
\nabla f_3^T \end{bmatrix}_{(mb)} \right)^{-1} \cdot \begin{bmatrix} f_1 \\
 f_2 \\
 f_3 \end{bmatrix}_{(p)} = 0 \iff \begin{bmatrix} f_1 \\
 f_2 \\
 f_3 \end{bmatrix}_{(p)} = 0
\]

I.e. a root of \(\tilde{f}\) is also a root of \(f\). \(\square\)

It has been proven that this preconditioning ensures the halting of the MK test in a small enough box around roots if \(f\) in [13]. Thus, this preconditioning can be used before the MK test in a subdivision algorithm searching for the roots of \(f\) (where boxes far from the roots are eventually discarded by \(C_f\)) 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 $f$ (given by information regarding the roots of $\tilde{f}$) is valuable, as it can be seen by its use in our algorithm in the subsequent chapters.
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 $\hat{B}$ of disjoint boxes for which $\bigcup_{B \in \hat{B}} B = B_0$ holds. Let us consider the elements of such a $\hat{B}$ 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 $\hat{B} = \{B_0\}$ and splitting an arbitrary element of $\hat{B}$ 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 $\text{Faces}(B)$ be the set of all 2D faces of $B$.

Now, for arbitrary box or box face $B$ and real number $c \geq 1$, define $B.\text{scale}(c)$ as the box/box face sharing the same mid-point and proportions as $B$ but having all interval widths multiplied by $c$. 
For \( f = (f_1, f_2) \) implicitly defining a space curve and satisfying the presented assumptions on an initial bounded box \( B_0 \subseteq \Box \mathbb{R}^3 \):

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

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

Then, consider the preconditioned test: \( F_{\text{PMK}_c}^f(\text{face}) \equiv MT^f(\text{face}.\text{scale}(c)) \)

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

Such a test can be equivalently written as: \( F_{\text{PMK}_c}^f(\text{face}) \equiv \)

\[
\left[ (\tilde{f}_1((\text{face}_1^-).\text{scale}(c))) < 0 < \tilde{f}_1((\text{face}_1^+).\text{scale}(c))) \land (\tilde{f}_2((\text{face}_2^-).\text{scale}(c))) < 0 < \tilde{f}_2((\text{face}_2^+).\text{scale}(c))) \right]
\]

(3.1)

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

\[
\text{PMK}_c^f(B) \equiv \bigvee_{\text{face} \in \text{faces}(B)} F_{\text{PMK}_c}^f(\text{face}.\text{scale}(c))
\]

(3.2)
Using the test $PMK^f_c(B)$, alongside $C_0, C_1,$ and $JC$ defined in previous sections, we can define
the following algorithm:

**Algorithm 1** Subdivide$^f(B_0, c, \epsilon)$

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

**Output:** A subset $B$ of an octree subdivision of $B_0$

**Ensure:** $\forall B \in B. \ (\neg C^f_0(B)) \land (\text{size}(B) \leq \epsilon) \land C^f_1(B) \land C^f_1(B) \land JC^f_1(B) \land PMK^f_c(B)$

\[
B \leftarrow \emptyset \\
Q \leftarrow \{B_0\}
\]

while $Q \neq \emptyset$ do

\[
B \leftarrow Q.pop()
\]

if $\neg C^f_0(B)$ then \hspace{1cm} \text{\textcircled{If $B$ not excluded}}

if $(\text{size}(B) \leq \epsilon) \land C^f_1(B) \land C^f_1(B) \land JC^f_1(B) \land PMK^f_c(B)$ then

\[
B.push(B) \hspace{1cm} \text{\textcircled{Subdivision halted for box $B$}}
\]

else

\[
Q.push(B.split()) \hspace{1cm} \text{\textcircled{Further subdivision required}}
\]

end if

end if

end while

Where Boolean expressions are evaluated using short-circuit evaluation. This ensures that each test in $(\text{size}(B) \leq \epsilon) \land C^f_1(B) \land C^f_1(B) \land JC^f_1(B) \land PMK^f_c(B)$ is only carried out if the preceding tests are true.

Note that as all $B \in B$ satisfies $PMK^f_c(B)$, 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:

\[ \forall B \in B. \ \exists \text{face} \in \text{Faces}(B.\text{scale}(c)). \ \exists p \in \text{face.\text{scale}}(c). \ f(p) = 0 \]

which means that there is a point \( p \) close to the volume of box \( B \) which lies on the traced curve \( S = B \cap f^{-1}(0) \). 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 \( B \in B \) as all boxes in such volumes must have been discarded by the \( C_0 \) (exclusion) predicate.

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

Unfortunately, however, we have no proof of further properties regarding the traced curve’s behavior in the outputted set of boxes \( B \) 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, \( \text{Subdivide}^f(B_0, c, \epsilon) \) has been observed to output a set of boxes \( B \) whose connected components \( K_1, \ldots, K_m \) each cover exactly one of the connected components of the traced curve in \( B_0 \). (i.e. connected components of \( B \) isolate the connected components of the traced curve in \( B_0 \))
3.2 Reconstruction Algorithm

Let $Q_{\text{curve}}$ be a set of boxes, outputted by $\text{Subdivide}^f(B_0, c, \epsilon)$ 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** $\text{Reconstruct}(Q_{\text{curve}})$

<table>
<thead>
<tr>
<th>Input:</th>
<th>$Q_{\text{curve}} \subseteq \square \mathbb{R}^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>An $\epsilon$-approximation of the curve(s) in $f = 0$</td>
</tr>
</tbody>
</table>

1. Create a directed graph $G$ that contains:
   2. A vertex for each box in $Q_{\text{curve}}$
   3. A directed edge from $B_1$ to $B_2 \iff$ 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 $K_1, K_2, ...$
5. 
6. for each component $K_i$ do
   7. if $\exists$ directed cycle in $K_i$ with length $> 2$ then
   8. Output short(est) directed cycle in $K_i$ with 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 $v, ..., w$ 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 $Q_{\text{curve}}$ 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 possess 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:

```matlab
... %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)
...```
if $B_{\text{par}}.\text{radius} < \text{MAXEPS} \land \text{local\_predicate.C1}(B_{\text{par}}, df, 3) \land \ldots \& \text{local\_predicate.C1\_cross}(B_{\text{par}}, df, dg, 5) \
\text{local\_predicate.Jacobian}(B_{\text{par}}, df, dg, 5)$
    $Q_{\text{Jac\_add}}[i] = B_{\text{par}};$
else
    $\text{children} = B_{\text{par}}.\text{split} ;$
    $Q_{\text{next}}[i] = \text{children} ;$
end

$\text{accepted} = [Q_{\text{Jac\_add}}[:]] ;$

$\text{disp}(['# \text{ accepted boxes} = ', \text{num2str}(\text{length}(\text{accepted}))]);$

$\text{Q} = [Q_{\text{next}}[:]] ;$
$Q_{\text{Jac}} = [Q_{\text{Jac}}, Q_{\text{Jac\_add}}[:]] ;$
$\text{depth} = \text{depth} + 1 ;$
end

if $\neg \text{isempty}(Q) \land \text{depth} = \text{depth\_limit} + 1$
    $\text{disp}('\text{Phase 1 stopped due to depth limit'}');$
end

$\text{disp}(['\text{Time for Phase 1:}', \text{num2str}(\text{toc}(t_{\text{Start}})), 's']);$
$\text{disp}('\text{Phase 1 finalized.}');$
$\text{disp}('\text{Proceeding to phase 2...}');$
$t_{\text{Phase\_2}} = \text{tic};$

$\% \text{ Subdivision Phase 2}$
$Q_{\text{MK}} = Q_{\text{Jac}} ; %\text{Input of the second phase of subdivision}$
$Q_{\text{curve}} = [] ; %\text{Output of second phase of subdivision}$

$\% \text{ Depth for phase 2}$
$numiterMK = 1 ;$

while $\neg \text{isempty}(Q_{\text{MK}}) \land numiterMK \leq numiterMK\_limit$

    $\% \text{ Iteration cell arrays}$
    $Q_{\text{MK\_next}} = \text{cell}(\text{length}(Q_{\text{MK}}), 1) ;$
    $Q_{\text{curve\_add}} = \text{cell}(\text{length}(Q_{\text{MK}}), 1) ;$

    $\text{disp}(['\text{Phase 2: numiterMK = }, \text{num2str(numiterMK)}, ', ' | length(Q_{\text{MK}}) = ', \text{num2str(length(Q_{\text{MK}}))}]);$

    $\% \text{ Parallel Subdivision}$
    $\text{parfor} i = 1: \text{length}(Q_{\text{MK}})$
    $B_{\text{par}} = Q_{\text{MK}}[i] ;$
    if $\neg \text{local\_predicate.C0}(B_{\text{par}}, f, 1) \land \neg \text{local\_predicate.C0}(B_{\text{par}}, g, 2)$
        if $\text{local\_predicate.Jacobian}(B_{\text{par}}, df, dg, 5) \land \text{local\_predicate.MK\_face}(B_{\text{par}}, f, df, g, dg, 7)$
            $\text{Q_{\text{curve\_add}}[i] = B_{\text{par}};}$
        end
    else
        internal call to C0\_faces
    end
end

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 $J_C$ condition and the additional depth of subdivision required to satisfy the $M_K$ test.

Then, the output of this subdivision step $Q_{curve}$ 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 $f = (f_1, f_2)$ 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 $\text{Subdivide}^f(B_0, c, \epsilon)$ 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**

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>224</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>128</td>
<td>4</td>
</tr>
</tbody>
</table>

Time: 0.84525s
Table 4.2: Experiment 1 - Phase 2

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
<td>0</td>
</tr>
</tbody>
</table>

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](image)

Figure 4.1: The output for experiment 1 seen in 3D
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 \( f = (f_1, f_2) \) given by:

\[
\begin{align*}
  f_1(x, y, z) &= x^2 + y^2 - z \\
  f_2(x, y, z) &= x^2 + y^2 + z^2 - 1
\end{align*}
\]

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

\[
\begin{align*}
  B_0 &= \left[ [-1, 1], [-1, 1], [-1, 1] \right] \\
  c &= 1.3 \\
  \epsilon &= 0.05
\end{align*}
\]

presents the following behavior and results:

**Table 4.3: Experiment 2 - Phase 1**

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>224</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>480</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>928</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1664</td>
<td>424</td>
</tr>
</tbody>
</table>

Time: 3.5449s
### Table 4.4: Experiment 2 - Phase 2

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>424</td>
<td>408</td>
</tr>
<tr>
<td>1</td>
<td>128</td>
<td>0</td>
</tr>
</tbody>
</table>

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](image-url)
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 \( f = (f_1, f_2) \) given by:

\[
\begin{align*}
  f_1(x, y, z) &= x^4 + 2x^2y^2 + y^4 - 2(x^2 + y^2) + 1 - z \\
  f_2(x, y, z) &= 0.5 - z
\end{align*}
\]

where inputs for \( \text{Subdivide}^f(B_0, c, \varepsilon) \) are as follows:

\[
B_0 = [[-1.2, 1.2], [-1.2, 1.2], [-1.2, 1.2]]
\]
\[c = 1.3\]
\[\varepsilon = \infty\]

presents the following behavior and results:

**Table 4.5: Experiment 3 - Phase 1**

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>128</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>480</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1920</td>
<td>108</td>
</tr>
<tr>
<td>6</td>
<td>3456</td>
<td>660</td>
</tr>
<tr>
<td>7</td>
<td>96</td>
<td>0</td>
</tr>
</tbody>
</table>

Time: 10.9647s
Table 4.6: Experiment 3 - Phase 2

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>768</td>
<td>364</td>
</tr>
<tr>
<td>1</td>
<td>1824</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>736</td>
<td>0</td>
</tr>
</tbody>
</table>

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
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 \( f = (f_1, f_2) \) given by:

\[
\begin{align*}
    f_1(x, y, z) &= x^2 + y^2 - z^2 - 2 \\
    f_2(x, y, z) &= x^2 - y^2 + z^2 - 1
\end{align*}
\]

where inputs for \( \text{Subdivide}^f(B_0, c, \epsilon) \) 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**

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>384</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>896</td>
<td>200</td>
</tr>
<tr>
<td>5</td>
<td>384</td>
<td>40</td>
</tr>
</tbody>
</table>

Time: 3.1143s
Table 4.8: Experiment 4 - Phase 2

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>256</td>
<td>224</td>
</tr>
<tr>
<td>1</td>
<td>256</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>0</td>
</tr>
</tbody>
</table>

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](image-url)
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 $f = (f_1, f_2)$ given by:

$$f_1(x, y, z) = \left\| \langle qp_1, p_1 - [x; y; z] \rangle \right\| - \left\| \langle qp_2, p_2 - [x; y; z] \rangle \right\|$$

$$f_2(x, y, z) = \left\| \langle qp_1, p_1 - [x; y; z] \rangle \right\| - \left\| \langle qp_3, p_3 - [x; y; z] \rangle \right\|$$

where $p_1, p_2, p_3$ represent coordinates of three points and $qp_1, qp_2, qp_3$ represent 3D vectors, given as follows:

$$p_1 = [0; 0; 4]$$

$$p_2 = [0; 0; -4]$$

$$p_3 = [1; 1; 0]$$

$$qp_1 = [4; 3; 0]$$

$$qp_2 = [4; -3; 0]$$

$$qp_3 = [4; 1; 4]$$

With this, the given $f_1$ function implicitly defines the surface of equal distance to the line defined by point $p_1$ and vector $qp_1$ and the line defined by point $p_2$ and vector $qp_2$. Similarly, $f_2$ implicitly defines the surface of equal distance to the line defined by point $p_1$ and vector $qp_1$ and the line defined by point $p_3$ and vector $qp_3$.

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 $Subdivide^c(B_0, c, \epsilon)$ 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**

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Evaluated Boxes</th>
<th>Number of Accepted Boxes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>368</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>1368</td>
<td>200</td>
</tr>
<tr>
<td>5</td>
<td>4504</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>10424</td>
<td>186</td>
</tr>
<tr>
<td>7</td>
<td>20376</td>
<td>2945</td>
</tr>
<tr>
<td>8</td>
<td>19056</td>
<td>4949</td>
</tr>
</tbody>
</table>

Time: 305.3464s
The outputted curve approximation can then be seen drawn on top of the boxes used in the approximation in the following figures:

![View 1](image1.png) ![View 2](image2.png)

**Figure 4.9:** The output for experiment 5 seen in 3D, from two different angles
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.1 Main code for the implemented algorithms

%Main code for parallelized subdivision

%% Initialization

tic;

% Example 1
syms f_syms g_syms d_syms x y z
assume(x, 'real');
assume(y, 'real');
assume(z, 'real');

% 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 = inf;

% 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([-ones(3,1),ones(3,1)]);
B0 = freecurvebox([-ones(3,1),ones(3,1)]).scale(1.2);
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([-ones(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;
Example 5: trisector

% https://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html
% Implement the sqrt function for intervals
p = [0, 0, 1; 0, 0, 1; 4, 4, 0];
direction = [4, 4, 4; 3, -3, 1; 0, 0, 4];
q = p * direction;

% d_syms(p,q,x,y,z) = norm(cross(q-p,p-[x;y;z]))/norm(q-p);
% 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));
% 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));

qp1 = q(:,1) - p(:,1);
p1 = p(:,1);
qp2 = q(:,2) - p(:,2);
p2 = p(:,2);
qp3 = q(:,3) - p(:,3);
p3 = p(:,3);

f_syms(x,y,z) = norm(cross(qp1,p1-[x;y;z]))/norm(qp1)-norm(cross(qp2,p2-[x;y;z]))/norm(qp2);
g_syms(x,y,z) = norm(cross(qp1,p1-[x;y;z]))/norm(qp1)-norm(cross(qp3,p3-[x;y;z]))/norm(qp3);

B0 = freecurvebox(10*[-ones(3,1),ones(3,1)]);
MAXEPS = inf;

%#################################################

df_syms = gradient(f_syms,[x,y,z]);
dg_syms = gradient(g_syms,[x,y,z]);
f_mat = matlabFunction(f_syms);
g_mat = matlabFunction(g_syms);
df_mat = matlabFunction(df_syms);
dg_mat = matlabFunction(dg_syms);
[f,g,df,dg] = funmanipulation.boxfunction(f_mat,g_mat,df_mat,dg_mat);

B0 = freecurvebox(10*[-ones(3,1),ones(3,1)]);

%% for trisector example
M = cat(3,p+100*(q-p),q+100*(p-q));

%% end trisector part

filename = which('parallel_local_tracecurve');

%% Settings
% Rules for inherited test results
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
B0.inherittestindices = 1:4;

%Depth limit for phase 1
depthlimit = 8;

%Depth limit for phase 2
numiterMKlimit = 6;

tStart = tic;

% 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)
            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)
                QJac_add{i} = B_par;
            else
                children = B_par.split;
                Q_next{i} = children;
            end
        end
    end

    accepted = [QJac_add{:}];
    disp(['# accepted boxes = ', num2str(length(accepted))]);

%Collection of results
Q = [Q_next{:}];
QJac = [QJac, QJac_add{:}];

depth = depth+1;
end

if ~isempty(Q) && depth == depthlimit + 1
    disp("Phase 1 stopped due to depth limit");
end

disp(['Time for Phase 1: ',num2str(toc(tStart)),'s']);
disp("Phase 1 finalized.");
disp("Proceeding to phase 2...");
tPhase2 = tic;

%% Subdivision Phase 2
QMK = QJac; %Input of the second phase of subdivision
Qcurve = [];

%%

%% Depth for phase 2
numiterMK = 1;

while ~isempty(QMK) && numiterMK <= numiterMKlimit
  %Iteration cell arrays
  QMK_next = cell(length(QMK), 1);
  Qcurve_add = cell(length(QMK), 1);

  disp(['Phase 2: numiterMK = ', num2str(numiterMK), ', length(QMK) = ', num2str(length(QMK))]);

  %Parallel Subdivision
  parfor i = 1:length(QMK)
    B_par = QMK(i);
    if ~local_predicate.C0(B_par, f, 1) && ~local_predicate.C0(B_par, g, 2)
      if local_predicate.Jacobian(B_par, df, dg, 5) &&
        local_predicate.MK_face(B_par, f, df, g, dg, 7)
        if any(B_par.testresults{7})
          Qcurve_add{i} = B_par;
        end
      else
        children = B_par.split;
        QMK_next{i} = children;
      end
    end
    accepted = [Qcurve_add{:}];
    disp(['# accepted boxes = ', num2str(length(accepted))]);

  end

  %Collection of results
  QMK = [QMK_next{:}];
  Qcurve = [Qcurve, Qcurve_add{:}];
  numiterMK = numiterMK + 1;
end

if ~isempty(QMK) && numiterMK == numiterMKlimit + 1
  disp('Phase 2 stopped due to depth limit');
end

disp(['Time for Phase 2: ', num2str(toc(tPhase2)),'s']);
disp('Phase 2 finalized.');

%% Outputs

disp(['Total time for subdivision: ',num2str(toc(tStart)),'s']);
disp(['Number of boxes in Q: ',num2str(length(Q))]);

%boxes which were not classified and fail the Jacobian
for i = 1:length(Q)
    Q(i).plotbox(ax,'r');
end

disp(['Number of boxes in QMK: ',num2str(length(QMK))]);
%boxes which were not classified but pass the Jacobian
for i = 1:length(QMK)
    QMK(i).plotbox(ax,'y');
end
disp(['Number of accepted boxes: ',num2str(length(Qcurve))]);
%boxes which pass the MK test and satisfy all the C1/Jacobian requirements
for i = 1:length(Qcurve)
    Qcurve(i).plotbox(ax,'g');
end

%% Curve Construction

leavessub = leaves(B0);
n = length(leavessub);
nodesnearcurvelogical = false(n,1);
centers = zeros(3,n);
parfor i = 1:n
    B = leavessub(i);
    leavessub(i).boxid = i;
    centers(:,i) = leavessub(i).center;
    if any(B.testresults{7}) %all boxes in the final subdivision which pass the MK test also pass Jacobian
        nodesnearcurvelogical(i) = true;
    end
end

nodes = 1:n;
nodesnearcurve = nodes(nodesnearcurvelogical); %list of nodes according to initial IDs
Afull = logical(sparse(n,n));
for i = 1:n %Can not directly use parfor here
    neighbors = leavessub(i).neighbors;
    for j = 1:length(neighbors) %Probably not worth doing with parfor
        Afull(i,neighbors(j).boxid) = true;
    end
end

Afull = Afull | Afull';
Gfull = graph(Afull);
Gnearcurve = subgraph(Gfull,nodesnearcurve);
bins = conncomp(Gnearcurve);
umcomps = numel(unique(bins));
curvepieces = cell(1,numcomps);
disp(['Numcomps: ', num2str(numcomps)])

for comp = 1:numcomps
    nodescomponentlogical = bins == comp;
    nodescomponent = nodesnearcurve(nodescomponentlogical);
    Gcomponent = subgraph(Gnearcurve,nodescomponentlogical);

    ndirected = length(nodescomponent);
    disp(['Comp ', num2str(comp), ': '])
    disp(['Component size:', ndirected])

    if ndirected == 1
        disp('Skipped comp of size 1.')
        disp('-----ooooo-----')
        continue
    end

    Adirected = logical(sparse(ndirected,ndirected));
    edges = table2array(Gcomponent.Edges);
    for i = 1:size(edges,1)
        edge = edges(i,:);
        B1 = leavessub(nodescomponent(edge(1)));
        B2 = leavessub(nodescomponent(edge(2)));
        center1 = B1.center;
        center2 = B2.center;
        direction = (B1.radius+B2.radius)/sqrt(3) - (center2-center1) < 10*eps;
        direction = direction*(-1)^(center1(direction)<center2(direction));%B1 has smaller -> coordinates than B2 in direction
        if all(((direction)'*B1.testresults{5} >= 0) && all(((direction)'*B2.testresults{5} >=0)
            Adirected(edge(1),edge(2)) = true;
        elseif all(((direction)'*B1.testresults{5} <= 0) && all(((direction)'*B2.testresults{5} <=0)
            Adirected(edge(2),edge(1)) = true;
        end
    end

    Gdirected = digraph(Adirected);
    hGdirected = plot(ax,Gdirected,'XData',centers(1,nodescomponent),'YData',centers(2,nodescomponent),'ZData',centers(3,nodescomponent));
    hGdirected.ArrowSize = 5; %7.5;
    hGdirected.LineWidth = 0.5; %0.75;

    %Find cycle or path
    %Use BFS to create a tree with shortest paths
    %Then run DFS to check for paths that have all boxes nearby (neighbors of neighbors)
    %Repeat for all neighbors of the starting vertex and select the shortest path
    [path,foundcycle] = findcycle(Gdirected);
    if ~foundcycle
        disp('Cycle not found!')
        path = findpath(Gdirected);
    else
        disp('Cycle found.')
    end

disp(['Path length: ', num2str(length(path))])
    disp('-----ooooo-----')
    curvepieces(comp) = path;

%Plot curve
    Coordinates = zeros(3,length(path));
    for i = 1:length(path)
Coordinates(:,i) = leavessub(nodescomponent(path(i))).center;
end
plot3(ax,Coordinates(1,:),Coordinates(2,:),Coordinates(3,:),'-k','Linewidth',3);
disp(['time for graph algorithm: ',num2str(toc),'s']);
A.2 Code for the cycle finding algorithm used in reconstruction

```matlab
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

```matlab
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] = shortestpath(Gundirected,'all',v);
    s = find(D==max(D),1);
    [~,D] = shortestpath(Gundirected,'all',s);
    t = find(D==max(D),1);
    TR = shortestpath(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

```matlab
classdef local_predicate < handle
%
%Assuming the following indexing for inherited test results:
%1:C0(f), 2:C0(g), 3:C1(f), 4:C1(g), 5:JC, 6:MK, 7:MK_face, 8:C0_face

properties (Constant)
  JC_scale = 4;
  MK_scale = 2;
  %For the test used on 6 sides:
  MK_box_scale = 1.3; %Scales each box before determining the faces, must be above 1 for current implementation
  MK_face_scale = 1.3; %Scales each face of the box, must be above 1 for current implementation
  MK_jac_ind = 6;
  C0_depth = 0; %0 means just evaluate on the main box, and gives the best performance by far (for C0_depth)
  C0_face_depth = 0; %0 means just evaluate on the main box, and seems to give the best performance (for C0_face_depth)
end

methods
  function this = local_predicate()
    local_tracecurve;
  end
end
methods(Static)
  % ###### C0 ######
  function bool = C0(B,f,ind)
    if nargin > 2
      if isempty(B.testresults{ind})
        bool = local_predicate.C0core(B,f);
        B.testresults{ind} = bool;
      else
        bool = B.testresults{ind};
      end
    else
      bool = local_predicate.C0core(B,f);
    end
  end
  function bool = C0core(B,f)
    %Returns true if the surface f=0 surely does not pass through B
    %bool = 0 \notin f(B)
    bool = funexcludes0(f,B,local_predicate.C0_depth);
    ~ interval.zeros(1).subset(f(B));
  end

  % ###### C1 ######
  function bool = C1(B,df,ind)
    if nargin > 2
      if isempty(B.testresults{ind})
        bool = local_predicate.C1core(B,df);
        B.testresults{ind} = bool;
      else
        bool = B.testresults{ind};
      end
    else
      bool = local_predicate.C1core(B,df);
    end
  end
```

61
else
    bool = B.testresults{ind};
end
else
    bool = local_predicate.C1core(B,df);
end

function bool = C1core(B,df)
% C1(B,df, IND)
% Returns true if the surface f=0 satisfies the C1 condition
% bool = 0 \notin < df(B) , df(B) >
    Bint = B.interval;
    bool = ~interval.zeros(3,1).subset(df(Bint)'*df(Bint));
end

function bool = C1cross(B,df,dg,ind)
if nargin > 2
    if isempty(B.testresults{ind})
        bool = local_predicate.C1crosscore(B,df,dg);
        B.testresults{ind} = bool;
    else
        bool = B.testresults{ind};
    end
else
    bool = local_predicate.C1crosscore(B,df);  
end

function bool = C1crosscore(B,df,dg)
% C1cross(B,df,dg,ind)
% Returns true if the curve defined by f=0 and g=0 satisfies the C1 condition
% bool = 0 \notin < df(B) x dg(B) , df(B) x dg(B)>
    Bint = B.interval;
    cross_fg = cross(df(Bint),dg(Bint));
    bool = ~interval.zeros(3,1).subset(cross_fg'*cross_fg);
end

% ####################
% ##### JC #####
function [bool,v] = Jacobian(B,df,dg,ind)
if nargin > 3
    if isempty(B.testresults{ind})
        [bool,v] = local_predicate.Jacobiancore(B,df,dg);
        B.testresults{ind} = v;
    else
        v = B.testresults{ind};
        bool = any(v);
    end
else
    [bool,v] = local_predicate.Jacobiancore(B,df,dg);
end
B.passesJacobian = B.passesJacobian || bool; % for function markcurve
end

function [bool,v] = Jacobiancore(B,df, dg)
B = B.scale(local_predicate.JC_scale);
v = zeros(3,1);
dB = [df(B),dg(B)];
for i = 1:3
    testinterval = (-1)^i*det(dB(setdiff([1,2,3],i),:));
    if 0 < testinterval
        v(i) = 1;
elseif testinterval < 0
    v(i) = -1;
else
    v(i) = 0; % Jacobian did not succeed
end
bool = any(v);
end

% ######## MK #######

function
 bool = MK(B,f,df,g,dg,ind,ax)
if nargin > 5
    if isempty(B.testresults{ind})
        if nargin > 6
            bool = local_predicate.MKcore(B,f,df,g,dg,ax);
        else
            bool = local_predicate.MKcore(B,f,df,g,dg);
        end
        B.testresults{ind} = bool;
    else
        bool = B.testresults{ind};
    end
else
    bool = local_predicate.MKcore(B,f,df,g,dg);
end
end

function
 bool = MKcore(B,f,df,g,dg,ax)
% Returns true if there are two pairs of opposite faces of B% with f having opposite sign on one pair, and g on the other.% Includes preconditioning around the box centerB = B.scale(local_predicate.MK_scale);
m = B.center;
dfm = df(m); dgm = dg(m);
Jm = [dfm,dgm,cross(dfm,dgm)]';
invJm = Jm^-1;
h = @(varargin) cross(dfm,dgm)'*(funmanipulation.convert2vec(varargin{:})-m);
  %varargin can be x,y,z components or box or interval vectorfprime = @(varargin) invJm(1,:) * [f(varargin{:});g(varargin{:});h(varargin{:})];gprime = @(varargin) invJm(2,:) * [f(varargin{:});g(varargin{:});h(varargin{:})];hprime = @(varargin) invJm(3,:) * [f(varargin{:});g(varargin{:});h(varargin{:})];faces = B.facets;
if ~funsmaller0(fprime,faces(1,1)) || ~funsmaller0(@(B) -fprime(B),faces(1,2)) ||
    ... ~funsmaller0(gprime,faces(2,1)) || ~funsmaller0(@(B) -gprime(B),faces(2,2)) ||
    ... ~funsmaller0(hprime,faces(3,1)) || ~funsmaller0(@(B) -hprime(B),faces(3,2))
bool = false;
else
    bool = true;
end
if nargin>5
    Bint = B.interval;
    plothandle = B.plotbox(ax,'b');
    if ~exist(axorig,'var') || isempty(axorig) || isgraphics(axorig) %#ok<NODEF>
        subplot(1,3,1,ax);
axorig = subplot(1,3,2);
r = groot;
Monitors = r.MonitorPositions;
[~,M] = max(Monitors(:,3));
fig = ax.Parent;
sizex = Monitors(M,3);
originalPos = fig.Position;
newPos = originalPos;

set(fig, 'units', 'pixels', 'position',newPos);
prepareaxes(axorig);
end
if ~exist('axtrans','var') || isempty(axtrans) || ~isgraphics(axtrans)
    axtrans = subplot(1,3,3);
    prepareaxes(axtrans);
end
plothoriginal = local_predicate.plotfunctions(axorig,f,g,h,Bint);
plothtransformed =
    local_predicate.plotfunctions(axtrans,fprime,gprime,hprime,Bint);
ploth = [plothoriginal,plothtransformed];
end
if nargin>5
    if bool
        titletext = 'success';
    else
        titletext = [...] '$ f^{\sim}(B_x^-)$ =
            ['',num2str(fprime(faces(1,1)).bounds),''],newline,...
        '$-f^{\sim}(B_x^+)$ =
            ['',num2str(uminus(fprime(faces(1,2))).bounds),''],newline,...
        '$ g^{\sim}(B_y^-)$ =
            ['',num2str(gprime(faces(2,1)).bounds),''],newline,...
        '$-g^{\sim}(B_y^+)$ =
            ['',num2str(uminus(gprime(faces(2,2))).bounds),''],newline,...
        '$ h^{\sim}(B_y^-)$ =
            ['',num2str(hprime(faces(3,1)).bounds),''],newline,...
        '$-h^{\sim}(B_y^+)$ =
            ['',num2str(uminus(hprime(faces(3,2))).bounds),''],newline];
    end
    title(axtrans,titletext,'interpreter','latex','FontSize',16);
    for i = 1:length(ploth)
        delete(ploth(i));
    end
dealeted(ploithandle);
end

% ####################
% ##### Plot #####
function ploth = plotfunctions(ax,fun1,fun2,fun3,B)
    if nargin == 5
        plotinterval = [B(1).bounds,B(2).bounds,B(3).bounds];
        axis(ax,plotinterval);
    else
        plotinterval = axis(ax);
    end
% Use the function arrayfun for the next 3 lines possibly
fun1_element = @(x,y,z) elementwisefunction(fun1,x,y,z);
fun2_element = @(x,y,z) elementwisefunction(fun2,x,y,z);
fun3_element = @(x,y,z) elementwisefunction(fun3,x,y,z);
ploth = cell(1,3);
ploth{1} = fimplicit3(ax,fun1_element,plotinterval,'b');
ploth{2} = fimplicit3(ax,fun2_element,plotinterval,'r');
ploth{3} = fimplicit3(ax,fun3_element,plotinterval,'g');
alpha(ploth{1},0.5);
alpha(ploth{2},0.5);
alpha(ploth{3},0.5);
end
function ploth = plotfunctions2(ax,fun1,fun2,B)
% plot the functions within the box for testing reasons
if nargin == 5
    plotinterval = [B(1).bounds,B(2).bounds,B(3).bounds];
    axis(ax,plotinterval);
else
    plotinterval = axis(ax);
end
% Use the function arrayfun for the next 3 lines possibly
fun1_element = @(x,y,z) elementwisefunction(fun1,x,y,z);
fun2_element = @(x,y,z) elementwisefunction(fun2,x,y,z);
ploth = cell(1,2);
ploth{1} = fimplicit3(ax,fun1_element,plotinterval,'b');
ploth{2} = fimplicit3(ax,fun2_element,plotinterval,'r');
alpha(ploth{1},0.5);
alpha(ploth{2},0.5);
end
% ####################

% ##### MK_face #####
function [bool,v] = MK_face(B,f,df,g,dg,ind,ax)
if nargin > 5
    if isempty(B.testresults{ind})
        if nargin > 6
            [bool,v] = local_predicate.MK_face_core(B,f,df,g,dg,ax);
        else
            [bool,v] = local_predicate.MK_face_core(B,f,df,g,dg);
        end
        B.testresults{ind} = v;
    else
        v = B.testresults{ind};
        bool = any(v);
    end
else
    [bool,v] = local_predicate.MK_face_core(B,f,df,g,dg);
end
function [bool,v] = MK_face_core(B,f,df,g,dg)
% Matrix showing which surfaces intersect the curve (populated below):
res = zeros(3,2);
% Does not consider intersections in non-parameterizable direction's faces
spandir = zeros(3,1);
% C0 on the faces of the initial box
[C0_face_small, ~] = local_predicate.C0_face_core(B.scale(1),f,g);
if C0_face_small % MK test terminates true but no faces are found, will not be
    bool = true;
v = reshape(res.',1,[]);
return
end

Bscaled = B.scale(local_predicate.MK_box_scale);

%C0 on the faces of the scaled box
[C0_face, exclusions] = local_predicate.C0_face_core(Bscaled,f,g);
if C0_face %MK test terminates true but no faces are found, will not be considered
    bool = true;
    v = reshape(res.',1,[]);
    return
end

[jacbool,jac] = local_predicate.Jacobian(B,df,dg, local_predicate.MK_jac_ind);
if ~jacbool %Jacobian test fails, need to split
    bool = false;
    v = reshape(res.',1,[]);
    return
end

faces = Bscaled.facets;

for i = 1:3
    if jac(i) ~= 0 %Parameterizable in this direction
        for j = 1:2
            if exclusions(2*(i-1)+j) ~= 1 %do not bother if face would be excluded
                face = faces(i,j);
                m = face.center;
                dfm = df(m); dgm = dg(m);

                %Remove the i'th dimension from dfm/dgf to get 2d versions
                dfm(i,:) = [];
                dgm(i,:) = [];

                Jm = [dfm,dgm]';
                invJm = Jm^(-1);

                fprime = @(varargin) invJm(1,:) * [f(varargin{:});g(varargin{:})];
                gprime = @(varargin) invJm(2,:) * [f(varargin{:});g(varargin{:})];

                edges = face.scale(local_predicate.MK_face_scale).facets;

                %facebool
                if ~funsmaller0(fprime,edges(1,1)) || ~funsmaller0(gprime,edges(1,2)) || ...
                    ~funsmaller0(fprime,edges(2,1)) || ~funsmaller0(gprime,edges(2,2))
                    facebool = false;
                else
                    facebool = true;
                end

                res(i,j) = facebool;
        end
    end
dirbool = res(i,1) && res(i,2);
spandir(i) = dirbool;
end
% Flatten result for output
v = reshape(res.',1,[]);
bool = any(v);
end

%####################
% ##### C0_face #####
function [bool,v] = C0_face(B,f,g,ind) %Returns true iff all surfaces can be excluded
if nargin > 3
    if isempty(B.testresults{ind})
        [bool,v] = local_predicate.C0_face_core(B,f,g);
        B.testresults{ind} = v;
    else
        v = B.testresults{ind};
        bool = all(v);
    end
else
    [bool,v] = local_predicate.C0_face_core(B,f,g);
end
end

function [bool,v] = C0_face_core(B,f,g)
% Matrix showing which surfaces can be excluded
res = zeros(3,2);

faces = B.facets;

for i = 1:3
    for j = 1:2
        face = faces(i,j);
        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
    end
end

% Flatten result for output
v = reshape(res.',1,[]);
bool = all(v);
end

%####################
A.5  **Code defining intervals and interval arithmetic**

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

    %Overloading operators for the interval class
    +%a+b
    +%a-b
    +%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
    %%%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)
```
\[ x \in a: \text{ a.element}(x) \]

%scale(factor): If \([a,b]\) is bounded then this doubles the length of the interval, while keeping the midpoint
% if \([a,b]\) is unbounded, then this halves the length of the uncovered number line

%linspace(n): creates a mesh with each interval getting uniformly split into n values properties
bounds \([a,b]\)
  \%if \(a\leq b\): \(I = [a,b]\)
  \%if \(b < a\): \(I = [-\infty, a] \cup [b, \infty]\)
  \%The empty interval is represented by \([\infty, -\infty]\)
  \%Imagine the ends of the number line being connected at infinity
end

methods
function this = interval(lowerbounds, upperbounds)
  \%Accepts two matrices of same size as bounds to create a matrix of intervals
  if nargin > 0
    if nargin == 1
      upperbounds = lowerbounds;
    end
    v = size(lowerbounds);
    if all(size(lowerbounds) == size(upperbounds))
      numv = prod(v);
      if numv > 1
        for ind = numv:-1:1
          this(ind) = interval(lowerbounds(ind), upperbounds(ind));
        end
        if length(v) == 1
          v = [v, 1];
        end
        this = reshape(this, v);
      else
        this.bounds = [lowerbounds, upperbounds];
      end
    else
      warning('Wrong interval definition');
    end
  end
end

% Overloading operators with MATLAB:
% https://ch.mathworks.com/help/matlab/matlab_oop/implementing-operators-for-your-class.html
function result = plus(a, b)
  \%a+b
  fct = @interval.plus_element;
  result = interval.elementwiseoperator(fct, a, b);
end
function result = minus(a, b)
  \%a-b
  result = a + (-b);
end
function result = uminus(a)
  \%-a
  fct = @interval.uminus_element;
  result = interval.elementwiseoperator(fct, a);
end
function result = uplus(a)
  \%+a
result = a;
end

function result = times(a,b)
    %a.*b
    fct = @interval.times_element;
    result = interval.elementwiseoperator(fct,a,b);
end

function result = mtimes(a,b)
    %a*b
    [i,j] = size(a);
    [j2,k] = size(b);
    result = interval(zeros([i,k]));
    if j == j2
        if ~isa(a,'interval')
            a = interval(a);
        end
        if ~isa(b,'interval')
            b = interval(b);
        end
        for iind = 1:i
            for kind = 1:k
                currentsum = interval(0,0);
                for jind = 1:j
                    currentsum = currentsum+interval.times_element(a(iind,jind).bounds,b(jind,kind).bounds);
                end
                result(iind,kind) = currentsum;
            end
        end
    else
        error('The matrix dimensions are not matching');
    end
end

function result = rdivide(a,b)
    %a./b
    if isa(b, 'interval')
        result = a.*inverse(b);
    else
        result = a.*(1/b);
    end
end

function result = inverse(a)
    %1./a
    fct = @interval.inverse_element;
    result = interval.elementwiseoperator(fct,a);
end

function result = power(a,b)
    %a.^b
    fct = @interval.power_element;
    result = interval.elementwiseoperator(fct,a,b);
end

function result = lt(a,b)
    %a < b
    fct = @interval.lt_element;
    result = interval.elementwiseoperator(fct,a,b);
end

function result = gt(a,b)
%a > b
result = b < a;
end
function result = le(a,b)
%a <= b
fct = @interval.le_element;
result = interval.elementwiseoperator(fct,a,b);
end
function result = ge(a,b)
%a >= b
result = b <= a;
end
function result = ne(a,b)
%a ~= b
result = ~(a==b);
end
function result = eq(a,b)
%a == b
fct = @interval.eq_element;
result = interval.elementwiseoperator(fct,a,b);
end

function result = inverse_elementwise(this)

fct = @interval.inverse_element;
result = interval.elementwiseoperator(fct,this);
end

function result = sqrt(a)
%sqrt(a)

fct = @interval.power_element;
result = interval.elementwiseoperator(fct,a,1/2);
end

function result = nthroot(a,n)
%nthroot(a,n)
fct = @interval.power_element;
result = interval.elementwiseoperator(fct,a,n^(-1));
end

function result = abs(a)
%abs(a)
fct = @interval.abs_element;
result = interval.elementwiseoperator(fct,a);
end

function result = norm(a)
%norm(a)
result = norm_p(a,2);
end

function result = norm_p(a,p)
%nthroot(a,b)
if length(size(a)) == 2 && length(size(p)) == 2 && all(size(p) == [1,1])
sizes = size(a);
if sizes(1) == 1 %row vector
result = sum(a.^p,2).^((1/p));
elseif sizes(2) == 1 %column vector
result = sum(a.^p,1).^((1/p));
else %matrix
  error('matrix norm not yet implemented')
end
elseif length(size(a)) ~= 2
error('invalid input a for norm_p(a,p)')
end

else
    error('norm_p(a,b) function requires a single scalar p')
end

function disp(this)
    disp('Interval');
    dispmat = interval.int2mat(this);
    disp(dispmat);
end

function str = int2str(this)
    str = mat2str(interval.int2mat(this));
end

function result = summinmax(this,directions,to_be_evaluated)
    if nargin<2
        directions = 1;
    end
    s = size(this);
    d = length(s);
    snew = s;
    snew(directions) = ones(1,length(directions));
    result = interval(zeros(snew));
    specifieddirections = false(1,d);
    specifieddirections(directions) = true(1,length(directions));
    vLim = s;
    vLim1 = s;
    vLim1(directions) = [];
    v1 = ones(1, length(vLim1));
    ready = false;
    while ~ready
        Index1 = arrayindexing.sub2indV(vLim1, v1);
        vLim2 = s(directions);
        v2 = ones(1, length(vLim2));
        ready = false;
        while ~ready
            v = zeros(1,d);
            v(~specifieddirections) = v1;
            v(specifieddirections) = v2;
            Index = arrayindexing.sub2indV(vLim, v);
            eval(to_be_evaluated);
            % Update the index vector:
            [v2,ready] = arrayindexing.updateindexvec(v2,vLim2);
        end
        % Update the index vector:
        [v1,ready] = arrayindexing.updateindexvec(v1,vLim1);
    end

function result = sum(this,directions)
    to_be_evaluated = 'result(Index1) = result(Index1)+this(Index);';
    result = summinmax(this,directions,to_be_evaluated);
end

function result = min(this,varargin)
    to_be_evaluated = 'result(Index1) = result(Index1)+this(Index);';
    result = summinmax(this,directions,to_be_evaluated);
end
%max
function result = max(this, varargin)
end

function result = cross(a, b)
    \%cross(a, b)
    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])
    \rightarrow
    result = [(a(2)*b(3)) - (a(3)*b(2)), (a(3)*b(1)) - (a(1)*b(3)), (a(1)*b(2)) -
    \rightarrow (a(2)*b(1))];
else
    error('cross function is only implemented for two interval vectors of size 3')
end

function result = det(this)
    \%det
    if length(size(this)) == 2 && all(size(this) == [2, 2])
    result = this(1)*this(4)-this(2)*this(3);
else
    error('det function has to be implemented for non-2x2 matrices')
end

\%cap
\%cup
function result = cup(this, b)
    fct = @interval.cup_element;
    result = interval.elementwiseoperator(fct, this, b);
end

function result = sign(this)
    fct = @interval.sign_element;
    result = interval.elementwiseoperator(fct, this);
end

function result = extractbounds(this)
    \%extractbounds
    s = size(this);
    n = prod(s);
    result = zeros(n, 2);
    for i = 1:n
        result(i, :) = this(i).bounds;
    end
    if s(end) == 1
        s(end) = [];
    end
    result = reshape(result, [s, 2]);
end

function result = subset(a, b)
    fct = @interval.subset_element;
    result = all(interval.elementwiseoperator(fct, a, b));
end

function result = superset(this, bi)
    result = bi.subset(this);
end

function result = scale(this, factor)
    result = interval(zeros(size(this)));
    s = 1+(factor-1)/2;
    for i = 1:numel(this)
        result(i) = interval(s*this(i).bounds(1)+(1-s)*this(i).bounds(2),(1-s)*this(i)
        \rightarrow ).bounds(1)*s*this(i).bounds(2));
function result = linspace(this,n)
    result = zeros([numel(this),n]);
    for i=1:numel(this)
        result(i,:) = linspace(this(i).bounds(1),this(i).bounds(2),n);
    end
    s = size(this);
    if length(s) == 2 && s(2) == 1
        s(2) = [];
    end
    result = reshape(result,[s,n]);
end

% functions which need revision
% function result = mrdivide(a,b)
% %a/b
% %%%%%%%%%%% Be careful if 0 is contained in numerator and denominator
% result = a*b.inverseinterval;
% end
% function result = mpower(a,b)
% % %a^b
% % % res1 =
% % % result = a.interval+b.interval;
% % end
methods(Static)
function result = elementwiseoperator(fct,a,b)
    %Computes elementwise: fct(a,b) or fct(a) if b is not defined
    %Possible outputtupes are: 'interval' or 'logical'
    %allows for arrays a and b to have only 1 element while the other doesn't, similar
    % to 1+[2,3] = [3,4]
    s = size(a);
    if isa(a,'double')
        a = interval(a);
    end
    if nargin >= 3
        if numel(a) == 1
            s = size(b);
        end
        if isa(b,'double')
            b = interval(b);
        end
    end
    snum = prod(s);   %#ok<NASGU>
    for ind = prod(s):-1:1
        if nargin <= 2
            input = {a(ind).bounds};
        else
            if numel(a) == 1
                input = {a.bounds,b(ind).bounds};
            elseif numel(b) == 1
                input = {a(ind).bounds,b.bounds};
            else
                input = {a(ind).bounds,b(ind).bounds};
            end
        end
end
end
result(ind) = fct(input{:});
end
result = reshape(result,s);
end
function result = plus_element(a,b)
%a+b
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)))
bounds = [-inf,inf];
else
    bounds = a+b;
end
result = interval(bounds(1),bounds(2));
end
function result = uminus_element(a)
%-a
result = interval(-a(2),-a(1));
end
function result = times_element(a,b)
%a*b
values = a’*b;
if a(1) <= a(2) && b(1) <= b(2)
bounds = [min(values(:)),max(values(:))];
else
    if interval.subset_element([0,0],b) || (b(2) < b(1) &&
    -> interval.subset_element([0,0],a)) %0 in a or b
        bounds = [-inf,inf];
    elseif b(2) < b(1) && ~interval.subset_element([0,0],a) &&
    ~interval.subset_element([0,0],b) %0 notin a or b
        bounds = [min(values(values>0)),max(values(values<0))];
    else
        if interval.subset_element([0,0],b)
            c = a;
            a = b;
            b = c;
        end
        if ~interval.subset_element([0,0],a)
            if 0 < b(1) %b positive
                bounds = [a(1)*b(1),a(2)*b(1)];
            elseif b(2) %b negative
                bounds = [a(2)*b(2),a(1)*b(2)];
            end
        else
            if 0<a(2) %a < complement(a)
                if 0 < b(1) %b positive
                    bounds = [a(1)*b(1),a(2)*b(1)];
                elseif bounds(2)>bounds(1)
                    bounds = [-inf,inf];
                end
            else %b negative
                bounds = [a(2)*b(1),a(1)*b(2)];
            end
        end
    end
e elseif 0<a(2) %a < complement(a)
        if 0 < b(1) %b positive
            bounds = [a(1)*b(1),a(2)*b(1)];
        elseif bounds(2)>bounds(1)
            bounds = [-inf,inf];
        end
    else %b negative
        bounds = [a(2)*b(1),a(1)*b(2)];
    end
end
end

else %complement(a) < 0
    if 0 < b(1) %b positive
        bounds = [a(1)*b(2),a(2)*b(1)];
        if bounds(2)>bounds(1)
            bounds = [-inf,inf];
        end
    else %b negative
        bounds = [a(1)*b(1),a(2)*b(2)];
        if bounds(2)>bounds(1)
            bounds = [-inf,inf];
        end
    end
end
end
end

result = interval(bounds(1),bounds(2));
end

function result = inverse_element(a)
%Computes the inverse interval a^-1
result = interval(1/a(2),1/a(1));
end

function result = abs_element(a)
%|a|
if a(1) <= a(2) %[a(1), a(2)]
    if a(1) >= 0
        result = interval(a(1), a(2));
    elseif a(2) <= 0
        result = interval(-a(2), -a(1));
    else %a(1) < 0 < a(2)
        result = interval(0, max(-a(1),a(2)));
    end
else [%-inf, a(2)] \cup [a(1), inf]
    if a(2) >= 0 || a(1) <= 0
        result = interval(0, inf);
    else
        result = interval(min(abs(a(1)), abs(a(2))), inf);
    end
end
end

function result = power_element(a,b)
%Computes a^b assuming b is a real number
%and a is non-negative when b is not an integer
if a(2) >= a(1)
    if b(1) == b(2) %b can be treated as a single real number
        b = b(1);
        if rem(b(1),1) == 0 %b is an integer
            if b >= 0
                if rem(b,2) == 0 && a(1) <= 0 && 0 <= a(2)
                    result = interval(0,max(a(1)^b,a(2)^b));
                else
                    result = a(1)^b;
                end
            elseif b < 0
                result = a(1)^b;
            end
        end
    end
end
end
res2 = a(2)^b;
result = interval(min(res1,res2),max(res1,res2));
end
else % b < 0
if rem(b,2) == 0 && a(1) <= 0 && 0 <= a(2)
    result = interval(min(a(1)^b,a(2)^b), inf);
elseif a(1) <= 0 && 0 <= a(2) && rem(b,2) == 1
    result = interval(-inf, inf);
else % a does not include 0
    res1 = a(1)^b;
    res2 = a(2)^b;
    result = interval(min(res1,res2),max(res1,res2));
end
end
elseif a >= 0 % b is not an integer, but a is non-negative
    res1 = a(1)^b;
    res2 = a(2)^b;
    result = interval(min(res1,res2),max(res1,res2));
else
    error("interval method a.^b not implemented for non-integer b and negative a")
end
else
    error("interval method a.^b not implemented for an interval b")
end
else % a(1) > a(2) (a = [-\inf,a(2)] \cup [a(1),\inf])
    result = cup(interval(-inf,a(2)).^2, interval(a(1),inf).^2);
end
end

function result = lt_element(a,b)
% a < b
result = (a(1)<=a(2)) && (b(1)<=b(2)) && a(2)<b(1);
end

function result = le_element(a,b)
% a <= b
result = (a(1)<=a(2)) && (b(1)<=b(2)) && a(2)<=b(1);
end

function result = eq_element(a,b)
% a == b
result = all(abs(a - b) == 3*eps);
end

function result = cup_element(a,b)
if a(2) >= a(1) & b(2) >= b(1)
    % [a(1), a(2)] \cup [b(1), b(2)]
    result = interval(min(a(1),b(1)), max(a(2),b(2)));
elseif a(2) >= a(1) & b(2) < b(1)
    % [a(1), a(2)] \cup [(-inf, b(2)] \cup [b(1), inf])
    if (a(1) <= b(2) & a(2) >= b(1)) || (a(1) >= b(2) & a(2) <= b(1))
        result = interval(-inf, inf);
    elseif a(1) <= b(2) & a(2) < b(1)
        result = interval(b(1), a(2));
    else
        result = interval(a(1), b(2));
    end
elseif a(2) < a(1) & b(2) >= b(1)
    % [(-inf, a(2)] \cup [a(1), inf]) \cup [b(1), b(2)]
    % symmetric to the previous case
    c = a;
a = b;
b = c;
if (a(1) <= b(2) && a(2) >= b(1)) || (a(1) >= b(2) && a(2) <= b(1))
    result = interval(-inf, inf);
elseif a(1) <= b(2) && a(2) < b(1)
    result = interval(b(1), a(2));
else % a(1) > b(2) && a(2) >= b(1)
    result = interval(a(1), b(2));
end

else % a(2) < a(1) && b(2) < b(1)
    result = interval(min(a(1),a(2)), max(a(2), b(2)));
end

%to be checked

function result = sign_element(a)
    result = interval(min(sign(a(1)), sign(a(2))), max(sign(a(1)), sign(a(2))));
end

function result = subset_element(a,b)
    % subset of b
    if a(1)<=a(2) && b(1) == b(2)
        result = b(1) <= a(1) && a(2) <= b(2);
    elseif a(1)<=a(2) && ~b(1) <= b(2))
        result = a(2) <= b(1) || b(2) <= a(1);
    elseif ~a(1)<=a(2)) && b(1) <= b(2)
        result = false;
    elseif (a(1)<a(2)) && ~b(1) <= b(2))
        result = a(1)<=b(1) && b(2)<=a(2);
end
end

function Index = sub2indV(Vlim,X)
    k = [1, cumprod(Vlim)];
    Index = sum(k(1:length(X)) .* (X - 1)) + 1;
end
function v = ind2subV(Vlim, ind)
    ind = ind-1;
    v = zeros(1,0);
    for i = 1:length(Vlim)
        v(i) = 1+mod(ind,Vlim(i));
        ind = (ind-v(i)+1)/Vlim(i);
    end
end
function mat = int2mat(I)
%Gather the interval bounds in a matrix
sizeI = size(I);
if sizeI(end) == 1
    sizeI(end) = [];
end
mat = zeros([sizeI,2]);
num = numel(I);
for i = 1:num
    if ~isempty(I(i).bounds)
        mat([i,i+num]) = I(i).bounds;
    else
        mat = ['empty ',num2str(sizeI),' array of intervals'];
    end
end

function int = zeros(varargin)
sizes = cell2mat(varargin);
int = interval(zeros(sizes));
end

function int = ones(varargin)
sizes = cell2mat(varargin);
int = interval(ones(sizes));
end

function int = unit(varargin)
sizes = cell2mat(varargin);
if length(sizes) == 1
    A = zeros([sizes,1]);
    B = ones([sizes,1]);
else
    A = zeros(sizes);
    B = ones(sizes);
end
int = interval(A,B);
end

function test()
    interval.test1;
end

function test1()
a = interval(0,1);
b = interval(3,4);
disp(a*b);
disp(a-b);
disp(a.*b);
disp(a./b);
end
end
end
end
A.6 Code defining generic boxes and functions on boxes

```matlab
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));
            end
        end
```
vLim = 2*ones(1,d);
v = ones(1, d);
ready = false;
while ~ready
    Index = arrayindexing.sub2indV(vLim, v);
%If v(i) = 1, then the child will get the smaller coordinate in dimension i
vcomplete = ones(1,D);
vcomplete(~this.length0) = v;
childrenarry(Index).boxdimensions =
    [this.boxdimensions(:,1),sum(this.boxdimensions,2)/2]+((vcomplete-1)'.*(j)
    - this.boxdimensions(:,2)-this.boxdimensions(:,1))/2*[1,1];
childrenarry(Index).length0 = this.length0;
childrenarry(Index).depth = this.depth+1;
childrenarry(Index).parent = this;
    
% Update the index vector:
[v,ready] = arrayindexing.updateindexvec(v,vLim);
end
this.children = childrenarry;
this.plotboxhandle = []; 
boxes = childrenarry(:)';
end

function c = center(this)
c = sum(this.boxdimensions,2)/2;
end
function r = radius(this)
r = norm(this.center-this.boxdimensions(:,1));
end
function C = corners(this)
D = size(this.boxdimensions,1);
d = D-sum(this.length0);
C = zeros(D,2^d);
for i = 1:2^d
    v = arrayindexing.ind2subV(2*ones(1,d),i);
    corner = zeros(D,1);
    corner(this.length0) = this.boxdimensions(this.length0,1);
    vind = 0;
    for j = 1:D
        if ~this.length0(j)
            vind = vind+1;
            corner(j) = (v(vind)==1)*this.boxdimensions(j,1)+(v(vind)==2)*this.box
            dimensions(j,2);
        end
    end
    C(:,i) = corner;
end

function f = facets(this)
D = size(this.boxdimensions,1);
d = D-sum(this.length0);
dind = d;
for i = D:-1:1
    if ~this.length0(i)
        for j = 2:-1:1
            B = copy(this);
            B.boxdimensions(i,3-j) = B.boxdimensions(i,j);
        end
    end
end
B.length0 = this.length0;
B.length0(i) = true;
f(j) = B;
dind = dind-1;
end

defunction Bscaled = scale(this,factor)
classype = class(this);
scaledboxdimensions =
interval(this.boxdimensions(:,1),this.boxdimensions(:,2)).scale(factor);
extractedbounds = scaledboxdimensions.extractbounds; %#ok<NASGU>
Bscaled = eval([classype,'(scaledboxdimensions.extractbounds);']);
Bscaled.length0 = this.length0;
end

%Returns a box without the removed dimension
defunction Bnew = removedim(this,dim)
classype = class(this);
dims = this.boxdimensions;
len0 = this.length0;
dims(dim,:) = [];
len0(dim,:) = [];
Bnew = eval([classype,'(dims);']);
Bnew.length0 = len0;
end

%Returns a box with an inserted dimension at the specified location
%Inserts to the first/last position if the requested position is
%out of bounds
defunction Bnew = insertdim(this,dim,rowdims)
classype = class(this);
dims = this.boxdimensions;
len0 = this.length0;
if dim < 1
  dim = 1;
elseif dim > length(this.boxdimensions) + 1
  dim = length(this.boxdimensions) + 1;
end
dims = [dims(1:dim-1,:); rowdims; dims(dim:end,:)];
len0 = [len0(1:dim-1,:); rowdims(1) == rowdims(2); len0(dim:end,:)];
Bnew = eval([classype,'(dims);']);
Bnew.length0 = len0;
end

function Q = leaves(this)
if isempty(this.children)
  Q = this;
else
  Q = [];
  for i = 1:numel(this.children)
    Q = [Q,leaves(this.children(i))]; %#ok<*AGROW>
  end
end

function int = interval(this)
int = interval(this.boxdimensions(:,1),this.boxdimensions(:,2));
function str = box2str(this)
    str = int2str(this.interval);
end

function Bcap = cap(this,B)
    %to be implemented
    Bcap = this;
end

function Bcup = cup(this,B)
    %to be implemented
    Bcup = this;
end

function varargout = coord(this)
    %Splits the box into its coordinate interval components
    %For 3D boxes:
    D = size(this.boxdimensions,1);
    varargout = cell(D,1);
    intvec = this.interval;
    output = mat2cell(intvec(:),ones(D,1))';
    [varargout{:}] = deal(output{:});
end

function plotsubdivision(this,ax,varargin)
    if isempty(this.children)
        this.plotbox(ax,varargin{:});
    else
        for i = 1:numel(this.children)
            this.children(i).plotsubdivision(ax,varargin{:});
        end
    end
end

function h = plotbox(this,ax,varargin)
    D = size(this.boxdimensions,1);
    d = D-sum(this.length0);
    switch d
        case 1
            %1-dimensional box possibly in higher dimensions
            corners = this.corners;
            switch D
                case {1,2}
                    h = scatter(corners(1,:),corners(2,:),varargin);
                case 3
                    h = scatter3(corners(1,:),corners(2,:),corners(3,:),varargin);
                end
        case 2
            %plot the box's edges
            corners = this.corners;
            corners = corners(:,[1,2,4,3,1]);
            switch D
                case 2
                    h = plot(ax,corners(1,:),corners(2,:),varargin{:});
                case 3
                    h = plot3(ax,corners(1,:),corners(2,:),corners(3,:),varargin{:});
                end
        case 3
            %plot the box's edges
            corners = this.corners;
            %collect the vertex indices making up the edges
eind = [1,2;...  
        end
end
```
4,3;
3,1;
5,6;
6,8;
8,7;
7,5;
1,5;
2,6;
3,7;
4,8;

XYZ = zeros(3,36);
for i = 1:12
    XYZ(:,[3*i-2,3*i-1,3*i]) = [corners(:,eind(i,:)),[NaN;NaN;NaN]];
end
h = plot3(ax,XYZ(1,:),XYZ(2,:),XYZ(3,:),varargin{:});
end
this.plotboxhandle = h;
end

function zoom(this,ax)
    plotinterval = this.boxdimensions;
    plotinterval = plotinterval(this.length0,:) = this.boxdimensions(this.length0,:)+0.5*[~ones(sum(this.length0),1),ones(sum(this.length0),1)];
    plotinterval = plotinterval';
    plotinterval = reshape(plotinterval(:),1,
    r1 = boundingbox(1,2)-boundingbox(1,1);
    r2 = boundingbox(2,2)-boundingbox(2,1);
    daspect(ax,[1 1 1]);
end

function fillbox(this,axeshandle,color)
    switch size(this.boxdimensions,1)
        case 2
            X = [this.boxdimensions(1,1);this.boxdimensions(1,2);this.boxdimensions(1,2);this.boxdimensions(1,1)];
            Y = [this.boxdimensions(2,1);this.boxdimensions(2,2);this.boxdimensions(2,2);this.boxdimensions(2,1)];
            this.filling = fill(axeshandle,X,Y,color);
        case 3
end
end

function delete(this)
    if ~isempty(this.children)
        while ~isempty(this.children)
            this.children(1).delete;
            this.children(1) = [];
        end
    end
    if ~isempty(this.parent)
        this.parent = [];
    end
end

function result = plus(a,b)
    [a,b] = this.convert2interval(a,b);
    result = a+b;
end
```
function result = minus(a,b)
    %a-b
    [a,b] = this.convert2interval(a,b);
    result = a-b;
end

function result = uminus(a)
    %-a
    [a] = this.convert2interval(a);
    result = -a;
end

function result = uplus(a)
    +%a
    [a] = this.convert2interval(a);
    result = a;
end

function result = times(a,b)
    %a.*b
    [a,b] = this.convert2interval(a,b);
    result = a.*b;
end

function result = mtimes(a,b)
    %a*b
    [a,b] = this.convert2interval(a,b);
    result = a.*b;
end

% function result = rdivide(a,b)
end

methods(Static)
function varargout = convert2interval(varargin)
    n = length(varargin);
    varargout = cell(1,n);
    for i = 1:n
        if isnumeric(varargin{i}) || isa(varargin{i},"interval")
            varargout{i} = varargin{i};
        else
            varargout{i} = varargin{i}.interval;
        end
    end
end

function [B,fig,ax] = test
    B = box([-1,1;-1,1;0,0],[false,false,true]);
    [fig,ax] = box.testsplit(B);
end

function [fig,ax] = testsplit(B)
    [fig,ax] = createfigure(B);
    B.split;
    B.children(1).split;
    B.children(1).children(2).split;
    B.plotsubdivision(ax,'k');
end

end
BIBLIOGRAPHY


