2017-12-20

Encasement: A Robust Method for Finding Intersections of Semi-algebraic Curves

Joseph Masterjohn
University of Miami, j.masterjohn@umiami.edu

UNIVERSITY OF MIAMI

ENCASEMENT: A ROBUST METHOD FOR FINDING INTERSECTIONS OF SEMI-ALGEBRAIC CURVES

By

Joseph Masterjohn

A THESIS

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Master of Science

Coral Gables, Florida

December 2017

UNIVERSITY OF MIAMI

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science

ENCASEMENT: A ROBUST METHOD FOR FINDING INTERSECTIONS OF SEMI-ALGEBRAIC CURVES

Joseph Masterjohn

Approved:

MASTERJOHN, JOSEPH (M.S., Computer Science)

(December 2017)

Encasement: A Robust Method for Finding Intersections of Semi-algebraic Curves

Abstract of a thesis at the University of Miami.

Thesis supervised by Professor Victor Milenkovic.
No. of pages in text. (118)

One of the fundamental concepts in computational geometry is deducing the combinatorial structure, or interactions, of a group of static geometric objects. In two dimensions, the objects in question include, but are not limited to: points, lines, line segments, polygons, and non-linear curves. There are various properties of interest describing a collection of such objects; examples include: distances, adjacencies, and most notably, intersections of these objects. Well studied, robust, and highly efficient algorithms exist for linear geometry and parametric curves. Problems involving non-linear, implicit, and high-dimensional objects however are an active area of research. Algebraic curves and algebraic surfaces arise frequently in numerous applications: GIS software, CAD software, VLSI design, computational chemistry and biology, dynamics, and robotics. We present a novel algorithm for finding all intersections of two semi-algebraic curves in a convex polygonal region, and describe its prospective analog in 3 dimensions. We "encase" the curves in the con-

vex region by repeatedly splitting the region until each cell contains at most two intersecting segments, thus detecting and isolating all of the intersections. The advantage of using encasement is that the running time is proportional to the size of the convex region when it is small and yet comparable to existing techniques when it is large.

Acknowledgements

I would like to thank my advisor Dr. Victor Milenkovic for his guidance and support in the past few years. Without his generous help and wisdom, this work would not be possible. I would also like to thank the rest of my committee Dr. Huseyin Kocak and Dr. Elisha Sacks for their support and advice throughout the thesis process.

Joseph Masterjohn

University of Miami

December 2017

Table of Contents

LIST OF FIGURES **viii**

1 INTRODUCTION **1**

1.1 Related Work 4

1.2 Outline 6

2 ENCASEMENT DEFINITION **7**

2.1 Algorithm Overview 8

3 ENCASEMENT ALGORITHM **17**

3.1 Loop Elimination 17

3.1.1 Efficient Critical Region Subdivision 19

3.1.2 Connected Component Finding 23

3.2 Curve Self Isolation 25

3.3 Intersection Isolation and Encasement 29

3.3.1 Intersection Isolation 31

3.3.1.1 Subdivision 33

3.3.1.2 Interval Newton's Method 35

3.3.2 Intersection Encasement 37

3.4 Curve-Curve Isolation 40

3.4.1 Splitting Line Generation 43

3.5 Subdivision and Refinement 49

4 IMPLEMENTATION **52**

4.1 ACP 52

4.2 Encasement Data Structure 56

4.3 Primitives 57

4.4 Software and Robust Curve Tracing 59

4.5 Resultant Calculation 63

4.6 Verification 81

4.7 Line Rounding 83

5 EXPERIMENTS AND RESULTS 86

5.1 Experimental Setup 88

5.2 Results Large Domain 91

5.3 Results Smaller Domains 96

6 FUTURE WORK 109

6.1 Improved Splitting Lines 109

6.2 Individual Encasements 110

6.3 Improved Intersection Finding 112

6.4 Arrangement Construction 113

6.5 3D Analogue 113

BIBLIOGRAPHY **115**

List of Figures

2.1 (a) The starting state of the algorithm, curves are traced based on their boundary intersections. (b) The critical regions of each curve are computed. 11

2.2 (a) The critical region in the center is contained entirely inside a face, and thus needs to be split. (b) Splitting through a vertex of the region discovers a loop of the green curve. 11

2.3 (a) The top cell contains two segments of the red curve (> 2 intersections along the boundary). (b) Self separation is performed and the two segments are isolated to different cells. . . 12

2.4 (a) f and g alternate along the boundary and an intersection is isolated. (b) The intersection cannot be verified to be the only one in the cell, thus the intersection is encased into a sub-cell where it can be verified. 13

2.5 (a) f and g intersect in a non-alternating manner along the boundary, we assume no intersections and attempt to split between the curves. (b) f and g are successfully confined to separate cells by performing a split. 14

2.6 (a) f and g do not alternate along the boundary, however they contain 2 intersections and cannot be separated by a splitting line. (b) Subdivision splits between the two intersections, confining each to individual cells. 15

2.7 A view of the final encasement. 16

3.1 (a) A loop of an algebraic curve f. (b) Alternating turning points along the loop. (c) By continuity, the curves f_x and f_y must intersect. (d) A "critical region" contained inside a loop surrounding a critical point. 20

3.2 Initial state of connected components. Box j is the first region whose right x boundary is x_0. Box i is the first region whose left x boundary is x_0 (a). Box i and j are adjacent, and are therefore added to the same connected component set (b). Moving i forward to the next box with left x boundary equal to x_0. It is also adjacent, and added to the current common set (c). Moving i forward results in disjoint boxes, so we start a new current set (d). We move j to the next box and see an adjacency. These two boxes start the next connected component set (e). We move i forward to a box that is disjoint from the previous i, but adjacent to box j. It is also added to the common connected component set (f). 26

3.3 Both red and green segments are of the curve f. The green segment shows one way of connecting two points with the same parity around the boundary. Because the curve is continuous, this would cause a self intersection within the cell with any other way of connecting the other boundary intersections 27

3.4 (a) and (b) show two possible ways of f to be connected if we only know about intersections along the boundary. By the argument in section 3.2, we can only make the assumption that intersections with the same parity aren't on the same component of the curve. (c) Find the midpoint of two points known not to be on the same segment. (d) Split in the direction of the tangent (gradient rotated by $\frac{\pi}{2}$) at the midpoint. 28

3.5 (a) Along the boundary R_i, f and g alternate. (b) A Newton step fails to shrink the interval (c) R_i is subdivided into two sub-intervals. 36

3.6 (a) A Newton step will be attempted in R_i. (b) The resulting interval and its intersection with R_i. (c) Assign R_{i+1} to the intersection of R_i and the resulting interval. 38

3.7 (a) A Newton step will be attempted in R_i. (b) The resulting interval and its intersection with R_i. It is entirely contained within R_i. (c) Assign R_{i+1} to the intersection of R_i and the resulting interval. 39

3.8 The intersection has been isolated, and the bisectors do not intersect the curves between the intersection and the boundary. No intersection encasement is needed, the intersection is the only one in the cell. 41

3.9 The left bisector has intersected the red curve. To ensure a cell where the bisectors do not intersect the curves before the cell boundary, split between the intersection of f and g and the first intersection of the bisector and the red curve. 42

3.10 Case 1: f_t and g_t converge 48

3.11 Case 2: f_t and g_t diverge 48

3.12 Case 3: f_t and g_t are skew 49

3.13 (a) Frame of reference for generating splitting lines for refinement. (b) Resulting cell after refinement. 51

4.1 The encasement software in initial state. The initial convex region B is shown. 64

4.2 Critical regions have been identified. Critical regions of f are red and g are green. 65

4.3 Current knowledge of all curves after intersecting with the boundary of B 66

4.4 Currently splitting through critical regions of f and g 67

4.5 One step after figure 4.4. Splitting through the critical region in the bottom cell discovered a loop of f 68

4.6 After all of the critical regions have been split, the separating splitting line is constructed. 69

4.7 A completed and validated encasement of f and g. 70

4.8 Intersections of f and g identified. 71

4.9 Zoomed in view of a Root2. The 2D interval is represented by the square box of width around 10^{-14}. The reason for the jagged curve tracing is because he precision of Newton's method in double at this scale is the same magnitude of rounding error. 72

4.10 Another Root2. 73

4.11 A third Root2. 74

4.12 Interactive selection and tracing of arrangement faces. 75

4.13 Arrangement face of f and g. 76

4.14 Arrangment face induced by a loop of f 77

4.15 Arrangement face containing a hole whose boundary is a loop of f 78

5.1 Analogous first steps in each of the n^2 CGAL and batch CGAL algorithms. Both demonstrate linear behavior asymptotically in the number of curves. 92

5.2 The cost of arrangement of the n^2 CGAL and batch mode CGAL. Both demonstrate quadratic behavior asymptotically in the number of curves, with the n^2 showing a smaller constant factor. 93

5.3 (Individual Curve Isolation) Total running time of encasement vs CGAL on pairs of curves of varying degrees. 97

5.4 (Individual Curve Isolation) Ratio of running time CGAL over encasement on pairs of curves of varying degrees. 98

5.5 (Root Isolation) Total running time of encasement vs CGAL on pairs of curves of varying degrees. 99

5.6 (Root Isolation) Ratio of running time CGAL over encasement on pairs of curves of varying degrees. 100

5.7 (Root Approx Double) Total running time of encasement vs CGAL on pairs of curves of varying degrees. 101

5.8 (Root Approx Double) Ratio of running time CGAL over encasement on pairs of curves of varying degrees. 102

5.9 (Total Running Time) Sum of all three stages of each encasement and CGAL on pairs of curves of varying degrees. 103

5.10 (Total Running Time) Ratio of running time CGAL over encasement on pairs of curves of varying degrees. 104

5.11 Running time of CGAL on pairs of varying degree curves with varying bounding box sizes. No marked improvement on running time with smaller bounding boxes. 107

5.12 Running time of encasement on pairs of varying degree curves with varying bounding box sizes. Curves of all degrees see improvement from smaller starting box. 108

CHAPTER 1

Introduction

One of the fundamental concepts in computational geometry is deducing the combinatorial structure or interactions of a group of static geometric objects. In two dimensions, the objects in question are points, lines, line segments, non-linear curves, and other objects whose boundaries are defined by the previously listed objects. There are various properties of interest describing a collection of such objects; examples include: distances, adjacencies, and most notably, intersections of these objects. In fact, one of the first problems studied in most computational geometry classes is the arrangement of line segments, a description of the intersections, adjacencies, and in general the entire combinatorial structure of a set of line segments in the plane. Well studied, robust, and highly efficient algorithms exist for linear geometry and parametric curves. Most research for problems limited to low degree objects concern finding lower bounds on algorithms for said problems. That said, problems involving non-linear, implicit, and high-dimensional objects are an active area of research.

In the case of two dimensions, this includes finding the intersections, and in general the arrangement, of a set of algebraic or semi-algebraic curves in the plane in a robust manner. Algebraic curves and algebraic surfaces arise frequently in numerous applications: GIS software requiring the analysis of curves defined on spheres [BT07], CAD programs for engineering design problems including VLSI design [TOG04], computational chemistry and biology for understanding the geometric structure of molecules, DNA, proteins, and other such entities [HS97], and in robotics for describing feasible configurations of links determining the workspace of a robot, as well as feasible configurations of poses determining the operational space [TOG04].

The problem we choose to focus our attention on is the problem of finding all intersections of two semi-algebraic curves in a convex polygonal region. This problem is equivalent to finding the real solutions of a system of bivariate polynomials:

$$f(x,y) = \sum_{i+j\leq d} f_{i,j}x^i y^j = 0$$

$$g(x,y) = \sum_{i+j\leq e} g_{i,j}x^i y^j = 0$$

Subject to:

$$Q_i(x,y) > 0$$

for a set of linear functions $Q_0, ..., Q_n$

Some work has been done in determining the bit complexity of isolating the real solutions to such a system [ES12], but in general it remains a studied problem.

The most widely used and current state of the art solutions to this problem come mainly from the area of computational algebra. Some work has been done on realizing solutions that can take advantage of the sparsity of intersections in the region of interest [BEKS13] however, most of these solutions however require solving the entire unconstrained system of bivariates on the entire real plane and then reporting a subset of these solutions that satisfy the constraints of the semi-algebraic curves. These techniques involve computing structures that have a fixed cost, regardless of the area of interest, and also have proven difficult to scale to higher dimensions [GCL92]. Thus, we seek to address these problems in a technique we call encasement. The running time of our technique is directly proportional to the size of the area of interest and is comparable in running time to the algebraic techniques for larger regions that contain all intersections of the curves involved. As well, we propose that the technique can be applied to the analogous problem in three dimensions, which seeks to address the problem of finding intersections of algebraic surfaces. This is a problem which, up to this point in time, has no well known solutions that are both efficient and robust for generic algebraic surfaces of arbitrary dimension. We leverage well known and efficient techniques involving linear geometry and finding roots of univariate polynomials to implement our algorithm.

1.1 Related Work

This work follows the strategy of the Adaptive Controlled Precision (ACP) library by Milenkovic and Sacks [MST13]. Any algorithm within this paradigm offers the guarantee of a bound on the backwards error of the solution: the result is correct solution for some small perturbation of of the original input. In this case the perturbation is made explicit: an an exact computation is carried out on the perturbed input. It is of note, however, to mention that if the input is already known to be generic, then ACP can abstain from perturbation and provide an exact solution in the same running time as the approximate solution. Milenkovic and Sacks provide an approximate arrangement for continuous and compact x-monotone curves in the plane [MS07]. Their output is realizable for a set of curves that are $O(\epsilon + k^2\epsilon)$ close to the original curves. In their case k being the number of inconsistencies (see their work for details). Also in the realm of approximate arrangements is a method of *controlled perturbation* by Halperin and Shelton for the approximate spherical arrangements and circular arrangements [HS97, HL04].

The above strategies are in response to the leading strategy in the realm of computational geometry, the Exact Geometric Computation (ECG) paradigm of Yap [Yap00]. This is the strategy behind the most well known computational geometry libraries CGAL [The17] and LEDA [MN99]. These systems use both numerical and symbolic computations to ensure the exact result regardless of degenerate input. CGAL provides a package for 2D arrangements of curves [WBF$^+$17]. This package has support for computation of the exact arrangement of arbitrary algebraic curves in the plane, described by polynomials with integer coefficients. Significant work has been put into ECG algorithms for non-linear arrangements that limit the input to circular arcs, conics, cubics, quadrics, and Bézier curves [BHK$^+$05, BEH$^+$02, DFMT00, EKSW06, EKP$^+$04, HW07, Wei02, WZ06]. Eigenwillig and Kerber provide an exact method for the arrangement of algebraic curves [EK08]. This method is based on the well known Bentley-Ottoman sweep-line algorithm [BO79]. They obtain efficiency by performing a only a small number of symbolic computations and relying heavily on an efficient adaptive-precision root finding algorithm [EKK$^+$05]. Their method however is held back by its dependency on having to compute and factor the resultant of two bivariates [GCL92]. While efficient algorithms exist, especially for integer and rational coefficients, they all have a running time that is at least $O(d^2)$ in the degree d of the resultant polynomial. Berberich et al. have implemented algorithms for computing arrangements of real planar algebraic curves and isolating real solutions of zero-dimensional bivariate systems [BEKS13]. They takes advantage of graphics hardware to expedite symbolic computations. Their results outperform reference implementations in Maple and CGAL.

1.2 Outline

Section 2 defines the notion of an encasement and how it leads to finding the intersections of a set of Semi-algebraic curves. Section 1.1 discusses some of the previously used techniques for solving systems of bivariate polynomials, mostly for their use in 2D arrangements, and current state of the art software implementing these algorithms. Section 2.1 discussed an overview of the 2D encasement algorithm, providing minimal details of the implementation. Section 4.1 gives and overview of the ACP software, the underlying library for robust geometric computation used for the encasement algorithm. Section 4.2 gives an overview of the data structures used in the algorithm. Section 4.3 details the geometric primitives used in the algorithm. Section 4 gives low level details of the software implementation. Section 5.2 gives the results of testing the implementation against state of the art software (CGAL [The17]) for computing 2D arrangements. Section 6 discusses possible future improvement to the algorithm as well as plans for a 3D implementation.

CHAPTER 2

Encasement Definition

Given two generic curves f and g, defined by bivariate polynomial equations $f(x, y) = 0$ and $g(x, y) = 0$, and a convex region $B \subset \mathbb{R}^2$ we define an encasement as a partitioning of B into a set of convex polygons $\{C_1, ..., C_n\}$ called cells with the one of following constraints:

1. **The cell is empty**. That is, both f and g do not exist anywhere on the interior of the cell, nor do they intersect the boundary of the cell.
2. **The cell contains exactly one of f or g.** If a cell contains a given curve, that curve intersects the cell in a single connected segment on the interior of the cell with a and b being the points on the boundary.
3. **The cell contains both f and g.** If this is true, each curve intersects the cell in a single segment as in the previous case. In addition, f and g intersect each other transversely at exactly one point on the interior of the cell.

And the edges of each cell satisfy one of two constraints:

1. **Edge has no intersection** No curve intersects the line segment defining the edge.
2. **Edge has one intersection** The line segment intersects one of either f or g at a single point on the interior of the segment.

All cells must have positive area but may have collinear edges.

Each intersection is isolated by a cell. We show how this structure can be used to isolate the intersection to an arbitrarily small region. The incidence of curves on the cells determines the structure of the arrangement of f and g. With the additional constraint that each curve segment is monotonic with respect to the line ab, the pairwise encasements are also be a means to construct the arrangement of a set of n curves (Sec. 6.4).

2.1 Algorithm Overview

The partition of B is generated by starting with B as the sole cell, and then splitting each edge or cell that violates all the constraints. An edge is split at a point. A cell C is split into two smaller cells (C_1 and C_2) with a splitting line $L = (p, v) = \{p + tv : t \in \mathbb{R} \text{ and } p, v \in \mathbb{R}^2\}$. The two resulting cells correspond to the subset of C to the left of L and the subset of C to the right of L respectively. That is to say:

$$C_1 = \{q \in C : (q - p) \times (v) < 0\}$$

$$C_2 = \{q \in C : (q - p) \times (v) > 0\}$$

The line segment of L that intersects C becomes an edge of each new cell. Each step of the algorithm is characterized by choosing a splitting line, given a cell and the curves that it intersects. Each splitting line seeks to bring the sub-cells closer to satisfying the constraints of a valid encasement.

The encasement algorithm visits each cell, initially just B, and attempts to certify it as simple, meaning intersecting f and g simply. If it cannot, it splits the cell by a line L in a manner that makes the two resulting cells either simple or closer to simple. The following is a summary. Details appear in the indicated sections.

Loop splitting (Sec. 3.1) Construct a *critical set* S for f: S does not intersect f and contains all local minima and maxima of $f(x, y)$. Since a loop of f must surround an extrema, it must surround a connected component of S. If a cell contains a connected component of S, L intersects it and hence splits any loop surrounding it. Likewise g. See Figure 2.2.

Self-separation (Sec. 3.2) If C contains more than one segment of f or g, L separates a maximal subset of one segment from another. See Figure 2.3.

Intersection isolation and encasement (Sec. 3.3) If f and g intersect an odd number of times, isolate a single intersection in the cell to a rectangle containing no other intersections. Split C by up to four lines to encase the intersection in a cell excluding all other intersections. See Figure 2.4.

Curve separation (Sec. 3.4) If both f and g have a single segment in C that have an even number of intersections, construct a splitting line L that seeks to separate f from g. See Figure 2.5.

Subdivide and Refine (Sec. 3.5) If curve separation or self separation fails to separate after a number of tries, we fall back to a binary subdivision of the cell to ensure termination. Subdivide the cell into two sub-cells, splitting by the perpendicular bisector of the intersections of f and the boundary. If that fails to separate the cell into two cells each with an odd number of intersections, refine the boundary of the cells containing f to roughly the mean tangent direction of f in the cell. See Figure 2.6.

A loop or pair of intersections or intersection can be split or separated or isolated only once. The number of splits required is $O(d^2)$ for d the maximum degree of f and g. Each split can generate a constant number of cells requiring self-separation of f. A split might create $O(d)$ segments to be self-separated and more than one split may be required to separate even a pair of segments, but empirically, self-separation is not a majority of the cost.

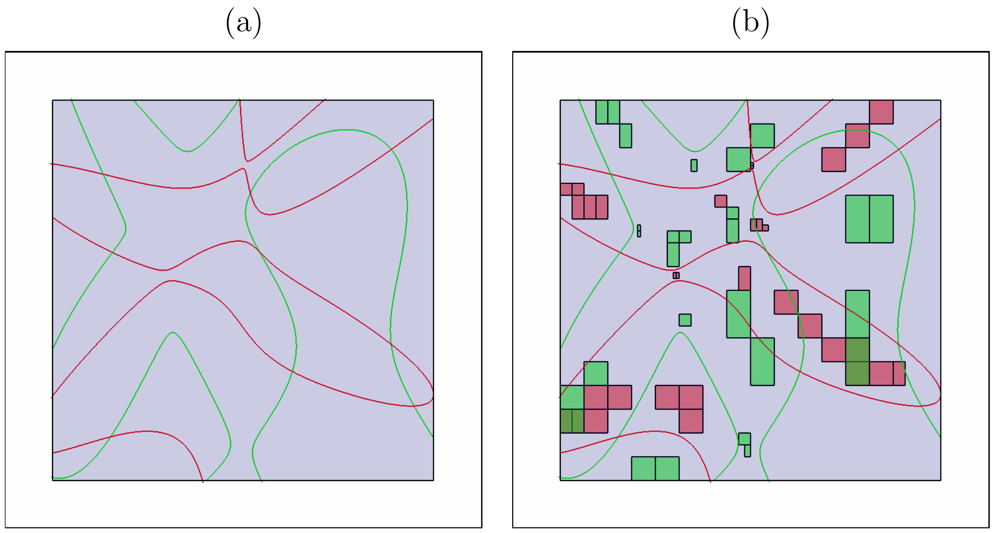

Figure 2.1: (a) The starting state of the algorithm, curves are traced based on their boundary intersections. (b) The critical regions of each curve are computed.

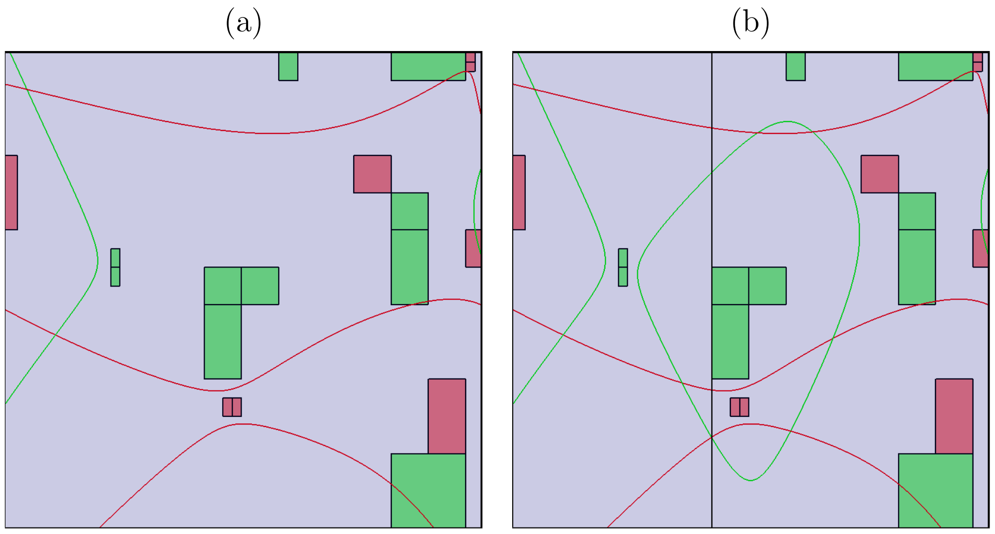

Figure 2.2: (a) The critical region in the center is contained entirely inside a face, and thus needs to be split. (b) Splitting through a vertex of the region discovers a loop of the green curve.

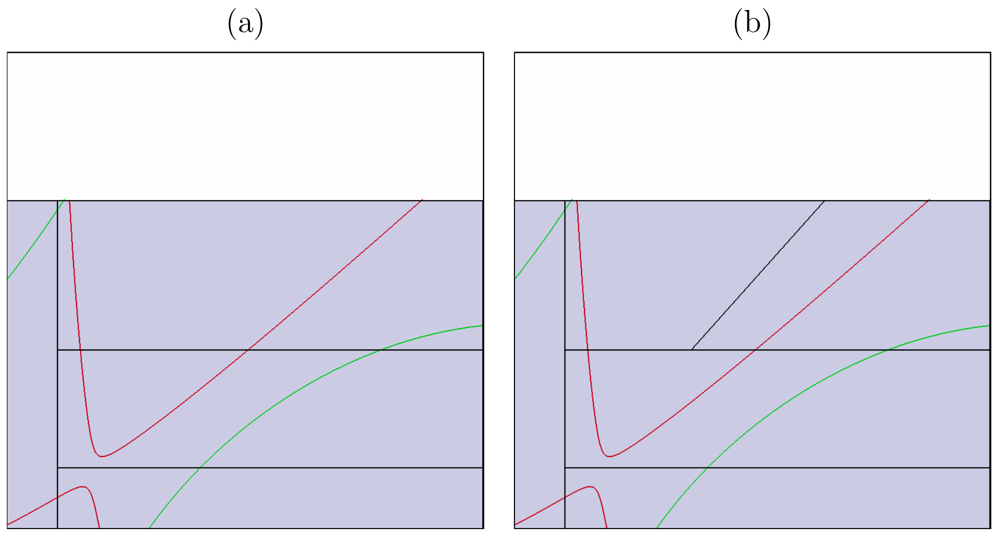

Figure 2.3: (a) The top cell contains two segments of the red curve ($>$ 2 intersections along the boundary). (b) Self separation is performed and the two segments are isolated to different cells.

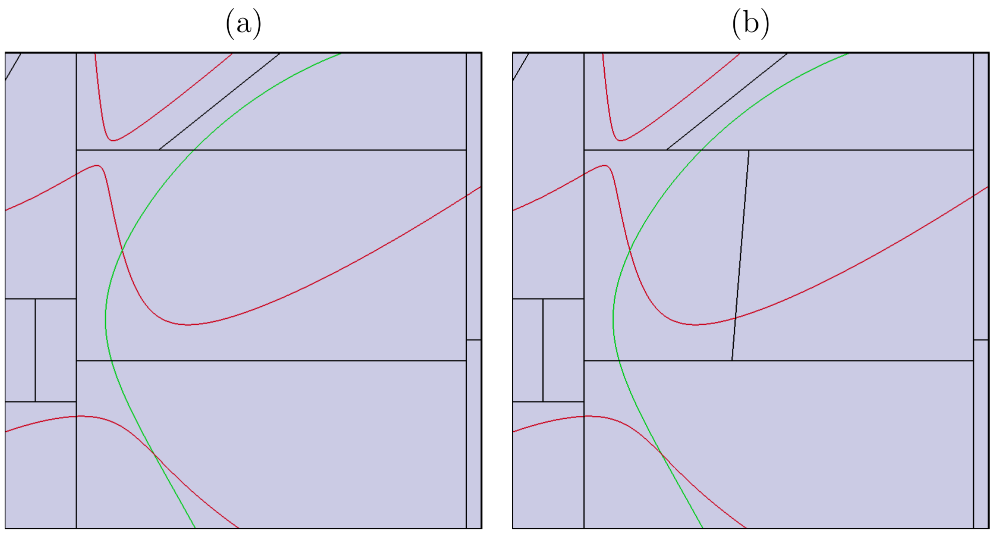

Figure 2.4: (a) f and g alternate along the boundary and an intersection is isolated. (b) The intersection cannot be verified to be the only one in the cell, thus the intersection is encased into a sub-cell where it can be verified.

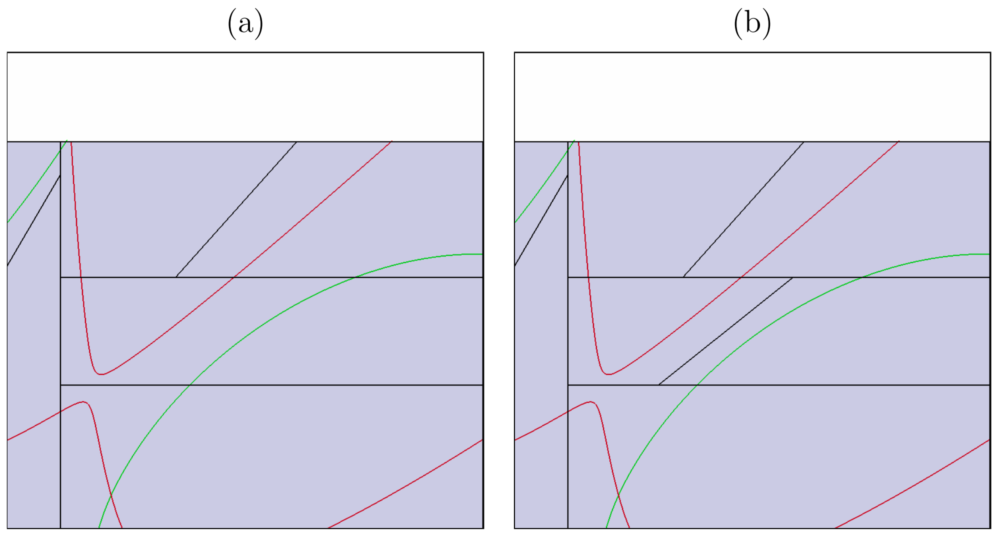

Figure 2.5: (a) f and g intersect in a non-alternating manner along the boundary, we assume no intersections and attempt to split between the curves. (b) f and g are successfully confined to separate cells by performing a split.

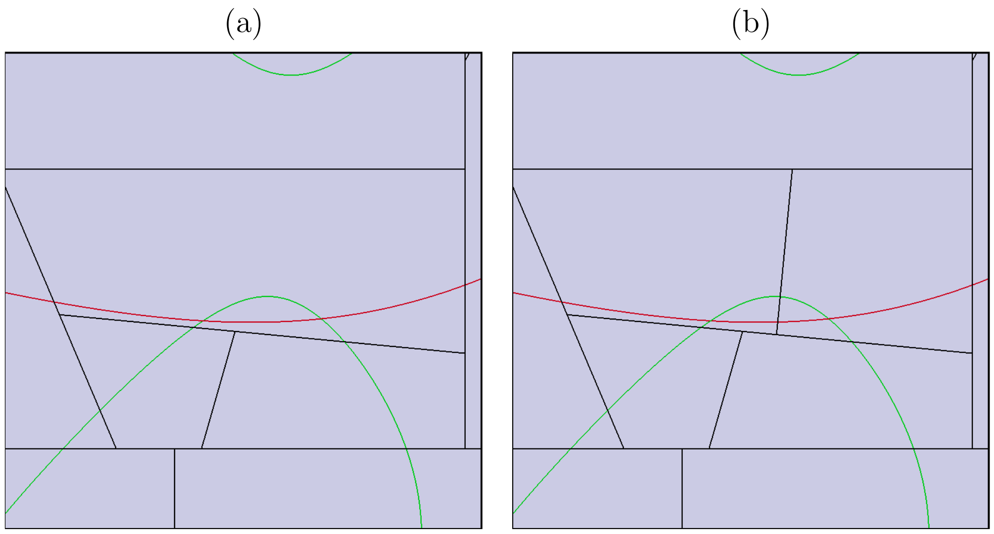

Figure 2.6: (a) f and g do not alternate along the boundary, however they contain 2 intersections and cannot be separated by a splitting line. (b) Subdivision splits between the two intersections, confining each to individual cells.

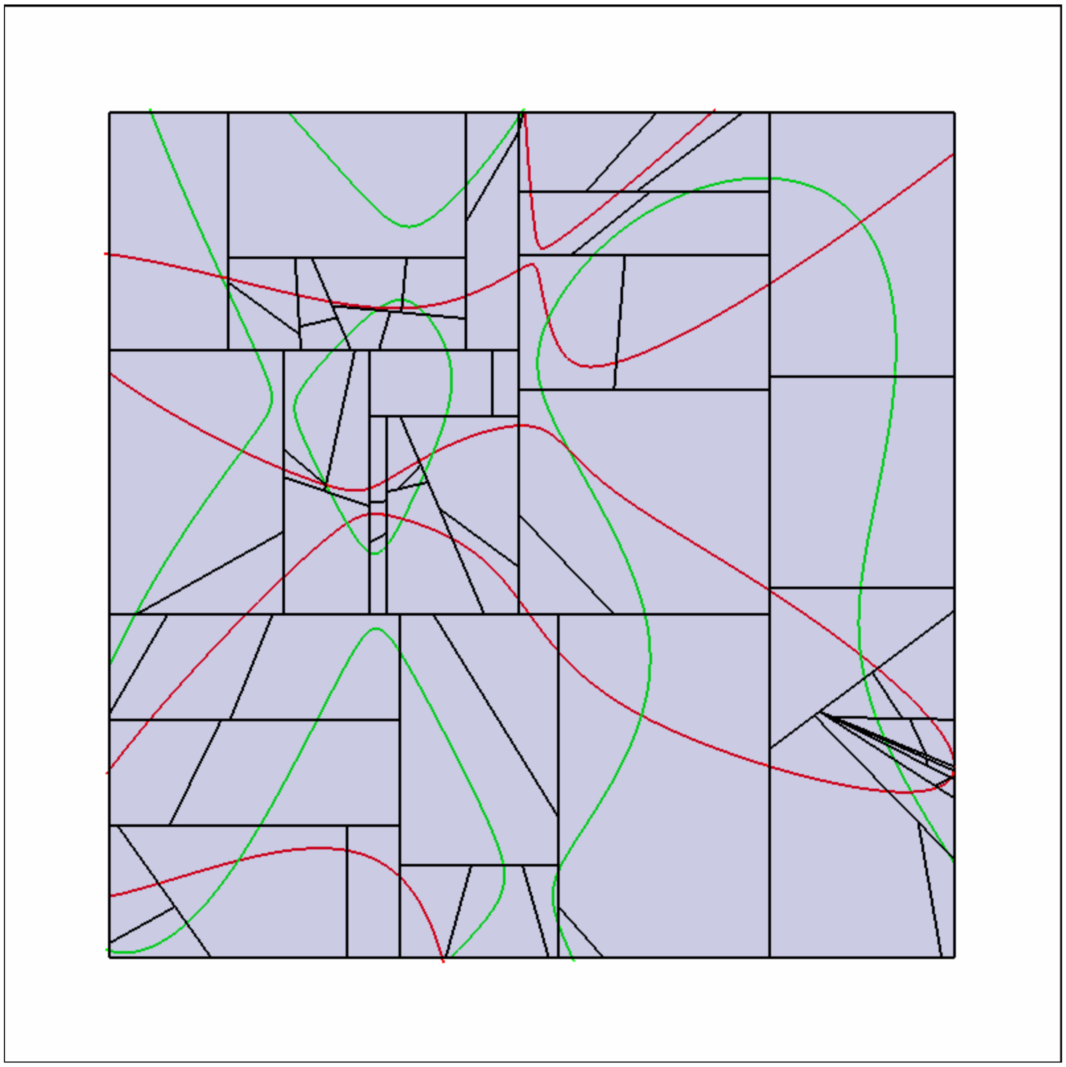

Figure 2.7: A view of the final encasement.

CHAPTER 3

Encasement Algorithm

3.1 Loop Elimination

Because generic algebraic curves can have closed loops within B, we need to detect any closed loops of f within B and perform splits that intersect them. If $f = \{(x, y) : f(x, y) = 0\}$ has a closed loop entirely contained in B, then the interior of the loop must contain a critical point of $f(x, y)$: (x_0, y_0) such that $f_x(x_0, y_0) = 0$ and $f_y(x_0, y_0) = 0$. This is equivalent to saying that the algebraic curves $f_x(x, y)$ and $f_y(x, y)$ intersect on the interior of the cell. By the extreme value theorem, f has an extremum in the interior because it is zero on the boundary, see Figure 3.1. To ensure all closed loops of f are split, we generate $S \subset B$ that contains all critical points of f and does not intersect f. For any connected component of S that contains a critical point, any loop associated with that critical point cannot intersect the connected component, thus it must completely surround the connected component. For each component S_i of S, if S_i is entirely contained within a cell C, we split C by a line that intersects S_i. Each S_i is considered with respect to the current, not original, set of cells, so a single line split might intersect multiple components hence split multiple loops.

S is determined with a planar decomposition algorithm. Start with an axis-aligned rectangle R (2D interval) that is the bounding box of B and evaluate f, f_x, and f_y on R, using interval arithmetic. If the resulting interval contains 0, the sign of the polynomial is ambiguous on R. If the sign of f, f_x, and f_y are all ambiguous, split the rectangle either vertically or horizontally across its longer axis and recurse. If the sign of f_x or f_y is known, this rectangle cannot contain a critical point, and the recursion terminates and this rectangle is ignored. If the sign of f is known, but the sign of f_x and f_y are still ambiguous, then this is a candidate rectangle. If a critical point of f exists, it lies inside a candidate rectangle. Also, since the sign of f is known, no loop of f intersects this rectangle. The recursion terminates and this rectangle is added to the set of candidate axis-aligned rectangles.

At this point, S is defined as the union of all candidate rectangles. We want to identify the connected components of S in order to split the critical regions in the most efficient way. This is accomplished with two merge operations, detailed in (Sec. 3.1.2), that organize the unordered set of candidate rectangles into connected components, see Figure 3.2.

3.1.1 Efficient Critical Region Subdivision

The first step in the algorithm (Loop Splitting) requires splitting of any loop of curve f that does not intersect the initial bounding box. Doing so is required to ensure that the algorithm accounts for all branches of f. We find sub-regions of the bounding box of constant sign that enclose all extrema of f. We evaluate f, f_x, f_y on this sub-region using interval arithmetic and verify that the resulting interval does not contain 0. If the sign of the value of f or its partials is ambiguous, we split the interval and recurse. The naive implementation of the splitting results in an undesirable recursive depth. The nature of this depth is a basic consequence of the weakness of interval arithmetic. Evaluating f on the interval is equivalent to detecting whether a 0^{th} order approximation to f contains 0 on $[x_l, x_u] \times [y_l, y_u]$. To reduce the depth of the recursion, we use the lower bound of a 2^{nd} order approximation of the curve and evaluate whether this quadratic approximation has zeros on the interval. Given a two-dimensional interval $R = [x_l, x_u] \times [y_l, y_u]$ and a curve $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, Taylor's theorem states that for any point $(x, y) \in R$: $f(x, y) \in Q_f(x, y) =$

$$f(x_l, y_l) + (x - x_l) f_x(x_l, y_l) + (y - y_l) f_y(x_l, y_l) +$$
$$\tfrac{1}{2}(x - x_l)^2 f_{xx}(R_x, y_l) + (x - x_l)(y - y_l) f_{xy}(R_x, y_l) + \tfrac{1}{2}(y - y_l)^2 f_{yy}(R)$$

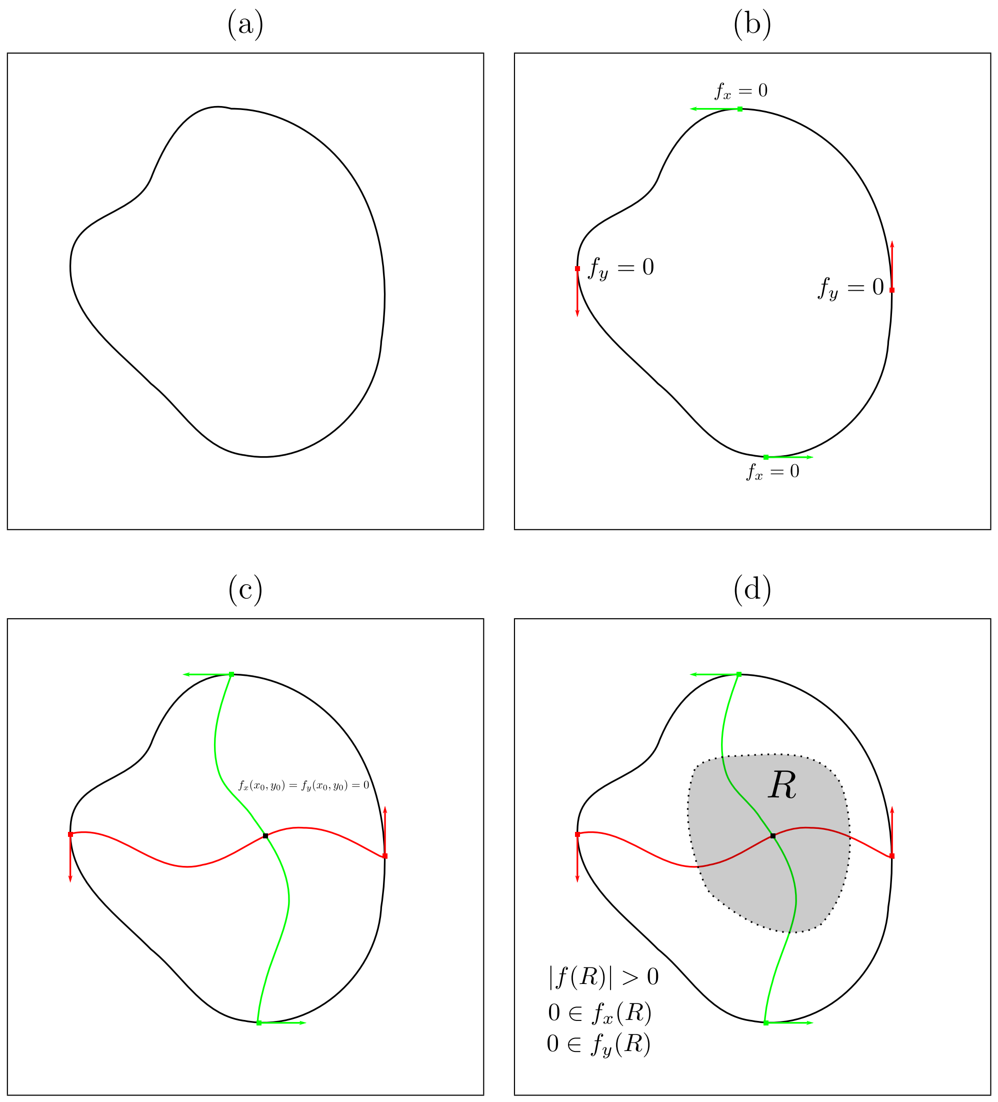

Figure 3.1: (a) A loop of an algebraic curve f. (b) Alternating turning points along the loop. (c) By continuity, the curves f_x and f_y must intersect. (d) A "critical region" contained inside a loop surrounding a critical point.

The function is first expanded in x about a, and then subsequently expanded in y about b. This leads to less evaluations on large intervals, and thus a smaller resulting interval. We choose to expand about the point (x_l, y_l) to ensure that when evaluated, $(x - x_l)$ and $(y - y_l)$ are positive. We compute the quadratic approximation on the interval, and then set the coefficients to their lower bounds in the resulting bivariate. So if $0 < Q_f(x, y)$ then $0 < f(x, y)$ on the interval R. If f is initially negative on the region, we work with an approximation of $-f$ for generality. We verify that $Q_f(x, y)$ is positive on R by

1. Evaluating $Q_f(x, y)$ at the four corners of the 2D interval:

   $(0, 0), (0, y_u - y_l), (x_u - x_l, 0), (x_u - x_l, y_u - y_l)$
2. Solving for intersections of $Q_f(x, y)$ and the boundary of R
3. Evaluating $Q_f(x, y)$ at its single critical point:

   $\{(x_0, y_0) : Q_{fx}(x_0, y_0) = Q_{fy}(x_0, y_0) = 0\}$ (if $(x_0, y_0) \in R$)

If $Q_f(x, y)$ has constant and consistent sign at all corners and its critical point, and has no intersections with the boundary, then it has constant sign on all of R. This also implies that $f(x, y)$ itself has constant sign on all of R. We compute and verify a lower bound quadratic approximation for f, f_x, f_y at each subdivision, rather than evaluating the functions themselves on the interval. This leads to a considerable decrease in the depth of the recursion as well as the amount of computation for each 2D interval.

The possibility exists to choose a higher order approximation of the function as our lower bound by expanding the Taylor series further. The benefit of this would be a more accurate approximation to the original function, and presumably a decrease in the depth of the recursion. One problem with this approach is that the evaluation of a higher degree function is more computationally expensive. For an approximation of degree d, $\frac{d(d+1)}{2}$ partial derivatives must be computed. The higher degree function must be evaluated on maxima/minima of the 2D interval. And we must do linear substitution and univariate root finding to compute intersections with the boundary. But, the primary disadvantage of this method is the computation of the critical points of the approximation. In the quadratic case, the critical points of the function can be found by solving a linear system. We don't have that advantage with higher degree approximations.

We chose to experiment with various degrees of approximation and ways of computing the critical points. We compared using resultants and recursive encasement in order to solve for the intersections of $f_x(x, y) = 0$ and $f_y(x, y) = 0$. In most cases recursive encasement was decidedly more efficient than our own resultant computation, and thus our choice for finding the critical points of higher degree approximations. For most inputs with degree ≤ 9 the speedups of using a higher degree approximation were negligible compared to using the quadratic approximation. However inputs with degree > 9 saw considerable improvement using higher order approximations. In general, the speedup of using a higher order approximation for a polynomial of degree d did not seem to benefit more from any approximation of degree greater than $\frac{d}{2}$. With these observations in mind, we determined the following heuristic for choosing an approximating polynomial:

- If the degree of f is ≤ 9, use a quadratic approximation.
- Else use a Taylor expansion to degree $\frac{d}{2}$.

One benefit of using this recursive strategy is that any speedups to the main encasement algorithm will be reflected in the speed of the critical region finding and vice versa.

3.1.2 Connected Component Finding

The termination of the recursive critical region subdivision results in a set of 2D intervals $r = \{[x_{l_0}, x_{u_0}] \times [y_{l_0}, y_{u_0}], ..., [x_{l_n}, x_{u_n}] \times [y_{l_n}, y_{u_n}]\}$ where on each interval r_i, $0 \notin f(r_i)$ and $0 \in f_x(r_i), 0 \in f_y(r_i)$. In other words all regions potentially inside of a loop of f and covering all critical points of f. These regions give us a way to complete the loop splitting procedure as any critical point that exists is completely contained in one of these regions. Splitting through each 2D interval would result in a considerable number of splits however. The observation that a loop cannot intersect any connected component of the 2D intervals. This leads us to the more efficient procedure of finding all connected components of the intervals and splitting through them.

If $r_i \cap r_j \neq \emptyset$, then r_i and r_j share a boundary and can be thought of as part of a connected component that covers the same critical region inside of a loop of f. It is only necessary to intersect a connected component of the regions once to split the loop surrounding the connected component. Therefore we seek to find all of the connected components of r, in order to minimize the number of splitting lines required to eliminate a closed loop of f. We calculate the connected components using a Union-Find data structure as follows (Fig. 3.2):

- Create lists for each unique x and y lower and upper bounds appearing in r. For instance $x_{lb}[0.5]$ is a list containing all $r_i \in r$, such that the lower bound of the x coordinate of r_i is 0.5. Keep each list sorted by the lower bound of the opposite dimension.
- For each 2D interval, insert it into its corresponding lists for each x and y lower and upper bound.
- For each x upper bound list, find the corresponding x lower bound list. In linear time in the size of the lists we can find all adjacent intervals that overlap in y. For each overlapping interval call $union(r_i, r_j)$, to make them belong to the same set.
- Do the same for each y upper bound list and its corresponding y lower bound list.
- At the end, each r_i should be a member of a set representing its connected component. If all intervals of this set are wholly contained inside the same encasement cell, then the connected component needs to be split.

Choose a representative interval from that set and a random point in its interior. Construct a horizontal or vertical line (chosen randomly) at that point, and split the encasement cell that contains that point.

3.2 Curve Self Isolation

After all loops of the curves have been successfully split, we are left with cells that only contain open curve segments of f and g. But a cell C may contain multiple disjoint open curve segments of a curve. Assume curve f intersects the boundary of cell C at $2k$ points $(a_1, a_2, ..., a_{2k})$ ordered clockwise. Because the curves are generic, curves contain no self intersections. Thus, any two distinct points with the same parity (a_i, a_{i+2j}) cannot be on the same open component of f. If they were, it would contradict the assumption that the curves do not contain self intersections. An example of this violation can be seen in Figure 3.3.

To eliminate multiple disjoint segments of the same curve, we choose the closest pair (a, b) of boundary intersections with the same parity measured by distance around the boundary, and construct a splitting line between them. Given a pair of boundary intersections that are on different curve segments, we construct the splitting line as such:

1. $p = 0.5 * (a + b)$ (i.e. midpoint of a and b)

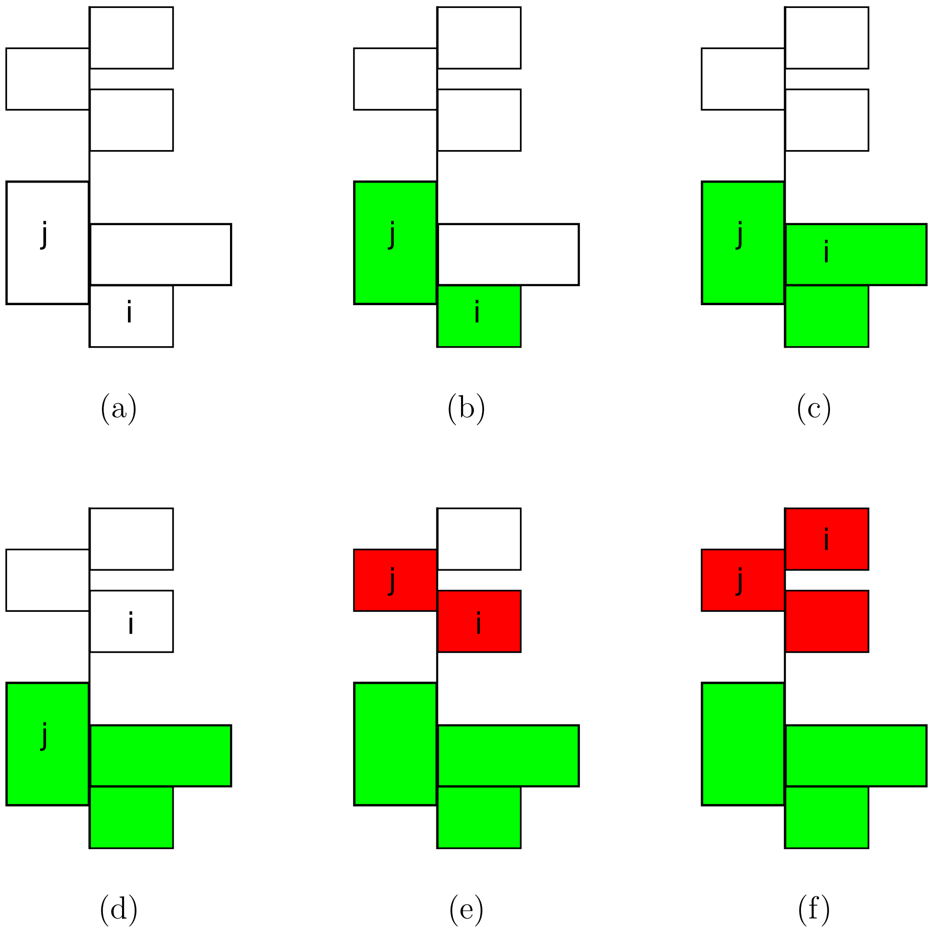

Figure 3.2: Initial state of connected components. Box j is the first region whose right x boundary is x_0. Box i is the first region whose left x boundary is x_0 (a). Box i and j are adjacent, and are therefore added to the same connected component set (b). Moving i forward to the next box with left x boundary equal to x_0. It is also adjacent, and added to the current common set (c). Moving i forward results in disjoint boxes, so we start a new current set (d). We move j to the next box and see an adjacency. These two boxes start the next connected component set (e). We move i forward to a box that is disjoint from the previous i, but adjacent to box j. It is also added to the common connected component set (f).

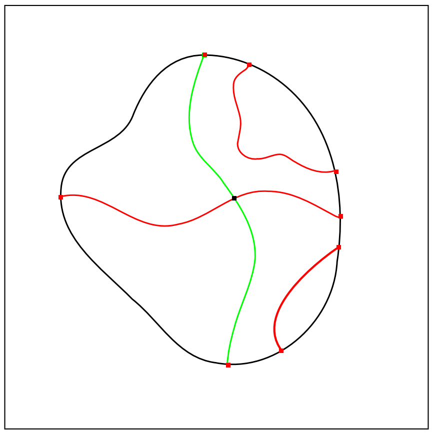

Figure 3.3: Both red and green segments are of the curve f. The green segment shows one way of connecting two points with the same parity around the boundary. Because the curve is continuous, this would cause a self intersection within the cell with any other way of connecting the other boundary intersections

2. $u = \nabla f(p)^{\perp}/|\nabla f(p)|$ (i.e. unit tangent to f at point p)
3. $L = (p, u) = \{p + ru : r \in \mathbb{R}\}$

We then split cell C with the generated line L and recurse into the two generated cells . If the splitting line fails to separate the curve segments, the sub-cells it generates will contain shorter segments which are closer to linear. For a linear f, this splitting line corresponds to the perpendicular bisector of the line segment connecting their closest points, an optimal separator.

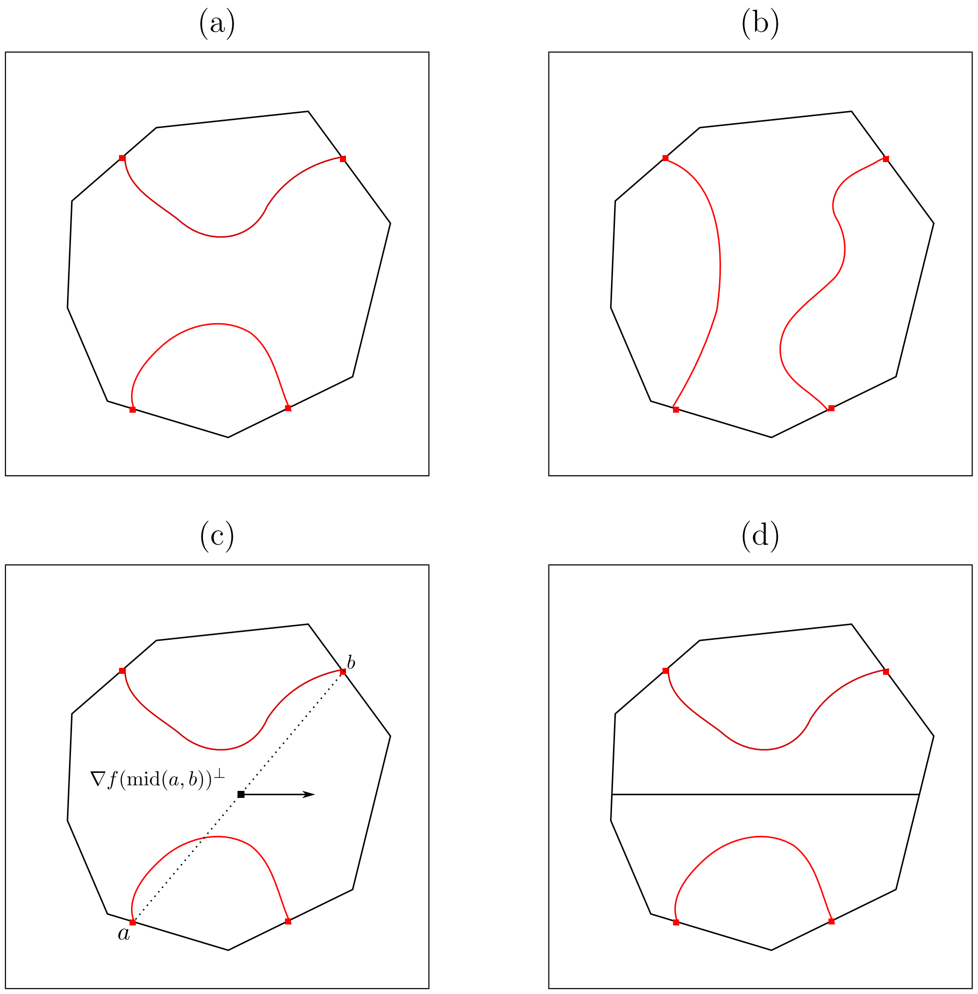

Figure 3.4: (a) and (b) show two possible ways of f to be connected if we only know about intersections along the boundary. By the argument in section 3.2, we can only make the assumption that intersections with the same parity aren't on the same component of the curve. (c) Find the midpoint of two points known not to be on the same segment. (d) Split in the direction of the tangent (gradient rotated by $\frac{\pi}{2}$) at the midpoint.

3.3 Intersection Isolation and Encasement

After self separation, some cells contain curve segments of both f and g. The parity of the number of intersections of two curves f and g can be calculated just by knowledge of the ordering of their intersections along the boundary (Sec. 3.3.1.1). If the parity is odd, f and g must have at least one intersection in the cell. In this case we isolate an intersection within the cell to a 2D interval and encase it by up to 4 line splits to a cell that excludes all other intersections.

Given a cell C with an odd number of intersections, we need to find an initial 2D interval R that isolates some intersection in C. Let R be the bounding box of C. If the interval $\nabla f(R) \times \nabla g(R)$ does not contain 0, then $\nabla f \times \nabla g$ has constant sign on R, and there is a single intersection of f and g inside R and more importantly inside C.

Lemma 1 *Given a 2D interval $R = [xl, xu] \times [yl, yu]$ and two continuous functions $f(x, y)$ and $g(x, y)$. If $\nabla f \times \nabla g(x, y)$ evaluated on the entire interval R has constant sign, then f and g intersect at most once inside R.*

Proof: Suppose there are at least two intersections of f and g in a cell R. Call an arbitrary pair of those intersections a and b. Also suppose that $\nabla f \times \nabla g$ has constant sign on R. Because $f(a) = 0$ and $f(b) = 0$, f must achieve a minimum or maximum along the interior of line segment ab. Call that point p. Consequently at this point p, the directional derivative in the direction ab will be zero. $\nabla f(p) \cdot ab = 0$. Likewise there exists a point q for g (q and p are not necessarily the same point). Thus $\nabla f(p)$ and $\nabla g(q)$ are parallel vectors. Since evaluating the sign of the cross products on the entire cell means considering every pair of points in the cell and $\nabla f(p) \times \nabla g(q) = 0$, the sign is not constant on the entire cell. This is a contradiction, and thus if $\nabla f \times \nabla g$ is constant on R, then f and g can intersect at most once inside R.

■

If $\nabla f(R) \times \nabla g(R)$ contains 0, split C with an axis-parallel line that bisects its longer dimension. One of the resulting cells will intersect contain an odd number of intersections of f and g. Recurse on that sub-cell. The splits and resulting sub-cells will not become part of the encasement topology: they are just auxiliary cells used to find a suitable R.

The 2D interval defining R, $[x_0, x_1] \times [y_0, y_1]$, now represents an intersection of f and g, although not very accurately. At this point we minimize the size of the interval by alternating iterative Newton's method and subdivision until they both fail in double precision or the current level of precision if a higher level of precision is being used.

Newton's method can fail if R is not within the interval of convergence. The details of this failure are described later in (Sec. 3.3.1). In the general case, after some number of subdivisions, Newton's method will succeed and iterate until it fails to shrink the interval, because the width of the interval is on the magnitude of roundoff error.

Subdivision can fail if we are unable to order the intersections along the boundary. This happens when the intervals representing the univariate roots corresponding to the boundary intersections overlap.

Given an minimal R, we can interpret R as a point in ACP because the fundamental representation of a point in ACP is a 2D interval whose precision can be adjusted when necessary. Let the point represented by the interval R be called q. If q is in cell C, we need to verify that it is the only intersection in that cell. C isolates an intersection q if the angle bisectors of the tangents lines to f and g at q intersect f and g only once inside C. Otherwise C is shrunk by subdivision to a sub-cell for which this is true (Sec. 3.3.2).

3.3.1 Intersection Isolation

Intersections of two bivariate curves act as vertices of the resulting arrangement, and thus it is necessary to be able to compute them to arbitrary precision. We devised a strategy that utilizes a two dimensional interval Newton's method along with a subdivision method in order to take a given interval $R = [x_l, x_u] \times [y_l, y_u]$ that definitely contains an intersection point of f and g, $(x_0, y_0) \in R : f(x_0, y_0) = g(x_0, y_0) = 0$, and refine it to the tightest bound at the desired precision.

We can determine the parity of the number of intersections of f and g inside a cell C only knowing the intersections of f and g with the boundary of C. This leads to a natural subdivision algorithm to refine a cell containing an intersection. Given a cell that contains an odd number of intersections, split this cell by any line going through the cell. One of the resulting sub-cells contains an odd number of intersections. Recursively subdivide that sub-cell until the desired level of precision. While being relatively simple to compute, a subdivision step is expensive because it requires multiple calls to the univariate root finder in order to compute new boundary intersections when computing the parity of a sub-cell. Also, it does not give any guarantee that the sub-cell in question contains a unique intersection. To alleviate this problem, we developed a 2 dimensional Newton like method to refine an interval containing a unique intersection.

2D Newton's method can fail to successfully refine the interval for numerous reasons:

- There is not a unique intersection in the interval, thus $\nabla f(R) \times \nabla g(R)$ does not have constant sign.

- R is not within the interval of convergence, so the arithmetic succeeds, but the interval produced is not smaller than the previous interval.
- The dimensions of the interval are on the order of magnitude of the roundoff error, and a successful step results in an interval no smaller than the previous.

The first case is easy to verify, and when it occurs we fall back on a subdivision step. We distinguish the second and third case by also doing a subdivision step. If the interval is too large to be within the interval of convergence, then a subdivision is likely to succeed. So when Newton's method does not shrink the interval, subdivision should help to proceed. However, if the interval did not shrink because of the third case, the subdivision step will fail when applied, because it won't be able to distinguish various univariate roots along the boundary because their intervals overlap. When a subdivision step fails after successfully iterating Newton's method, we terminate isolation.

3.3.1.1 Subdivision

A subdivision step requires detecting the parity of the number of intersections of f and g within a cell C. This is easily computed with only knowledge of the intersections of the curves along the boundary of the cell [BL04].

Lemma 2 *Consider a convex polygonal cell C and two algebraic curves f and g. Let $r_1, ..., r_n$ be the intersections of f and the boundary of C, in clockwise* *order, and let x_i be the number of intersections of g and the boundary of C between r_i and r_{i+1}. Then the parity of the number of intersections of f and g inside C is equivalent to the parity of $x_2 + x_4 + x_6 + ... + x_k$.*

Given a 2D interval, $R = [x_0, x_1] \times [y_0, y_1]$, we can find the intersections of f and g with the boundary of R with four substitutions and invocations of the univariate root solver. For instance, for intersections along the lower boundary of the interval we would substitute $\mathbf{y} = y_o$ into $f(\mathbf{x}, \mathbf{y})$, to create $f(\mathbf{x}, y_o)$ a univariate polynomial in x. We then query for the roots of that univariate in the interval $[x_0, x_1]$, call them $r_0, ..., r_n$. Each root represents an intersection with the boundary (r_i, y_0). We can do the same for the curve g, to create roots $s_0, ..., s_m$ and corresponding boundary intersections (s_i, y_0). These two lists of curves can then be compared by their x coordinate to give them an ascending ordering along the lower boundary from x_0 to x_1 (e.g. $(r_0, s_0, s_1, s_2, r_1, ...)$). An ordering of the roots along each of the four boundary intervals of R give an ordering of all intersections along the boundary, and thus a way to compute the parity of the number of intersections of f and g inside R.

The subdivision step determines the major axis of R, and splits at the midpoint of the interval on that axis. WLOG assume the x dimension of $R = [x_0, x_1] \times [y_0, y_1]$ has the greater magnitude. Let $x_m = \frac{(x_0+x_1)}{2}$, then R is split into $R_l = [x_0, x_m] \times [y_0, y_1]$ and $R_r = [x_m, x_1] \times [y_0, y_1]$. The parity of the number of intersections of f and g in these two sub-intervals is computed and the one with odd parity becomes the new R. Only one substitution for each curve, $f(x_m, [y_0, y_1])$ etc., is required to split the cell, and the previous boundary intersections along R can be re-purposed for the boundaries shared with R_r and R_l.

3.3.1.2 Interval Newton's Method

Newton's method is a well known numerical technique for finding roots of equations. However there is always inherent imprecision in any numerical technique implemented in floating point arithmetic, and no robust guarantees are given for the standard Newton algorithm. Interval Newton's method is a version of the standard Newton method that uses interval arithmetic to isolate the root within an interval. This provides the robustness necessary for doing exact computations [HW03].

Consider the truncated Taylor series of a function f about a point x in Lagrange form:

$$f(y) = f(x) + (y - x)f'(x) + ... + \frac{(y - x)^m}{m!} f^{(m)}(x) + R_m(x, y, \xi)$$

where:

$$R_m(x, y, \xi) = \frac{(y - x)^{m+1}}{(m + 1)!} f^{(m+1)}(\xi)$$

For a point $\xi \in X = [x, y]$ (assuming $x < y$). Thus:

$$f^{(m+1)}(\xi) \in f^{(m+1)}(X)$$

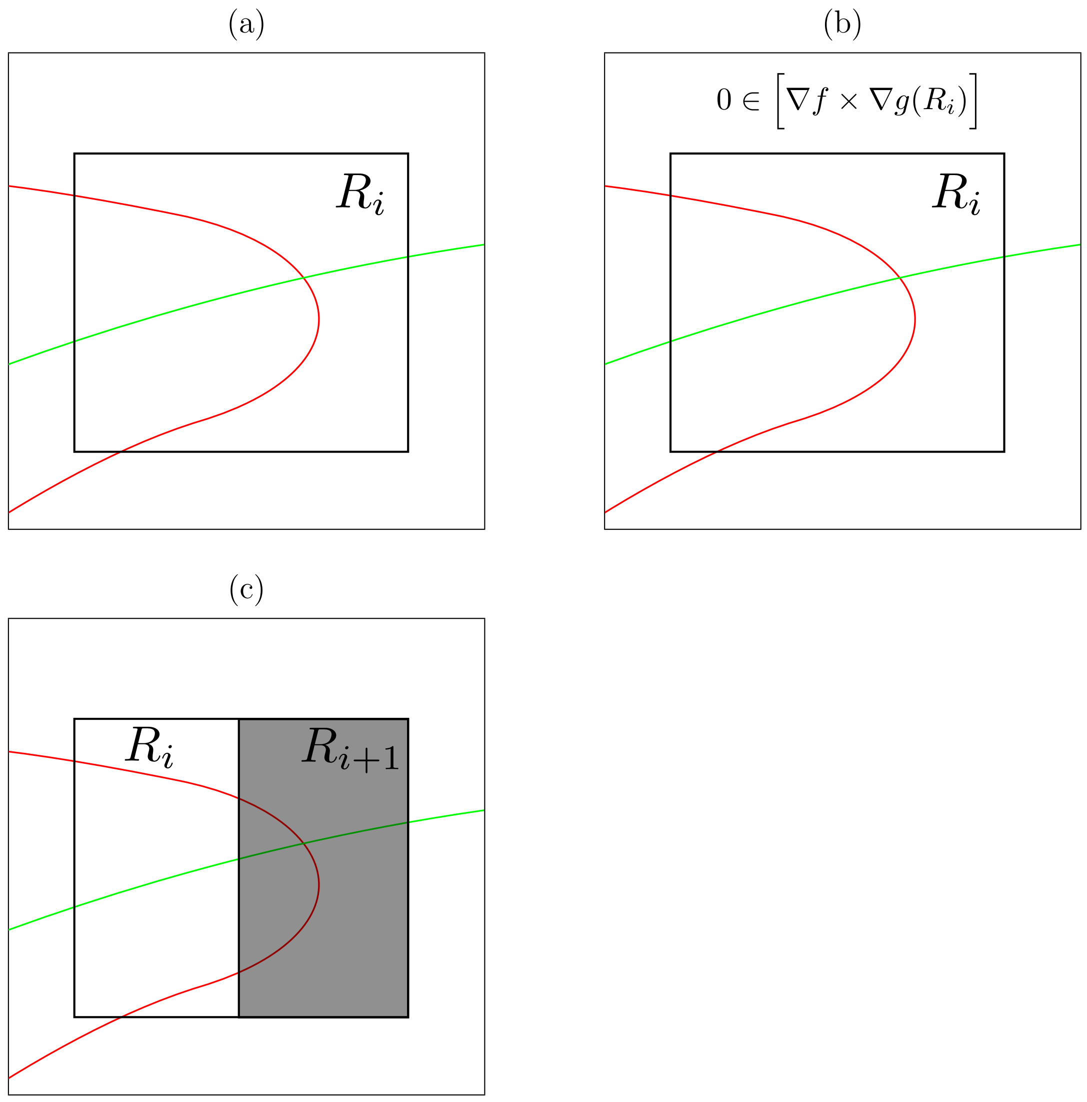

Figure 3.5: (a) Along the boundary R_i, f and g alternate. (b) A Newton step fails to shrink the interval (c) R_i is subdivided into two sub-intervals.

This bound on the remainder of the truncated series leads to the fact that for any $x, y \in X$:

$$f(y) \in f(x) + (x - y)f'(X)$$

The same holds in the multidimensional case $z, y \in R = [x_l, x_u][y_l, y_u]$:

$$f(z) \in f(y) + J_f(R) \cdot (z - y)$$

For a zero in the interval $\{z : f(z) = 0\}$

$$f(y) + J_f(R) \cdot (z - y) = 0$$

$$z \in y - J_f^{-1}(R) \cdot f(y)$$

Given our starting interval R we iteratively improve the interval by computing $R_i = R_{i-1} \cap \left(y - J_f^{-1}(R_{i-1}) \cdot f(y) \right)$. In order to compute $J_f^{-1}(R_{i-1}) \cdot f(y)$, we solve the linear system:

$$J_f(R) \cdot \mathbf{x} = f(y)$$

for $\mathbf{x}$ using Gaussian elimination. We repeat this process until $R_i - R_{i-1} = \emptyset$, at which time we have reached the limits of the current level of precision and the magnitude of the interval is on the order of roundoff error.

3.3.2 Intersection Encasement

Once an intersection, q, of f and g is found within a cell C, we need to verify that it is the only intersection inside that cell, and encase it if not. Compute

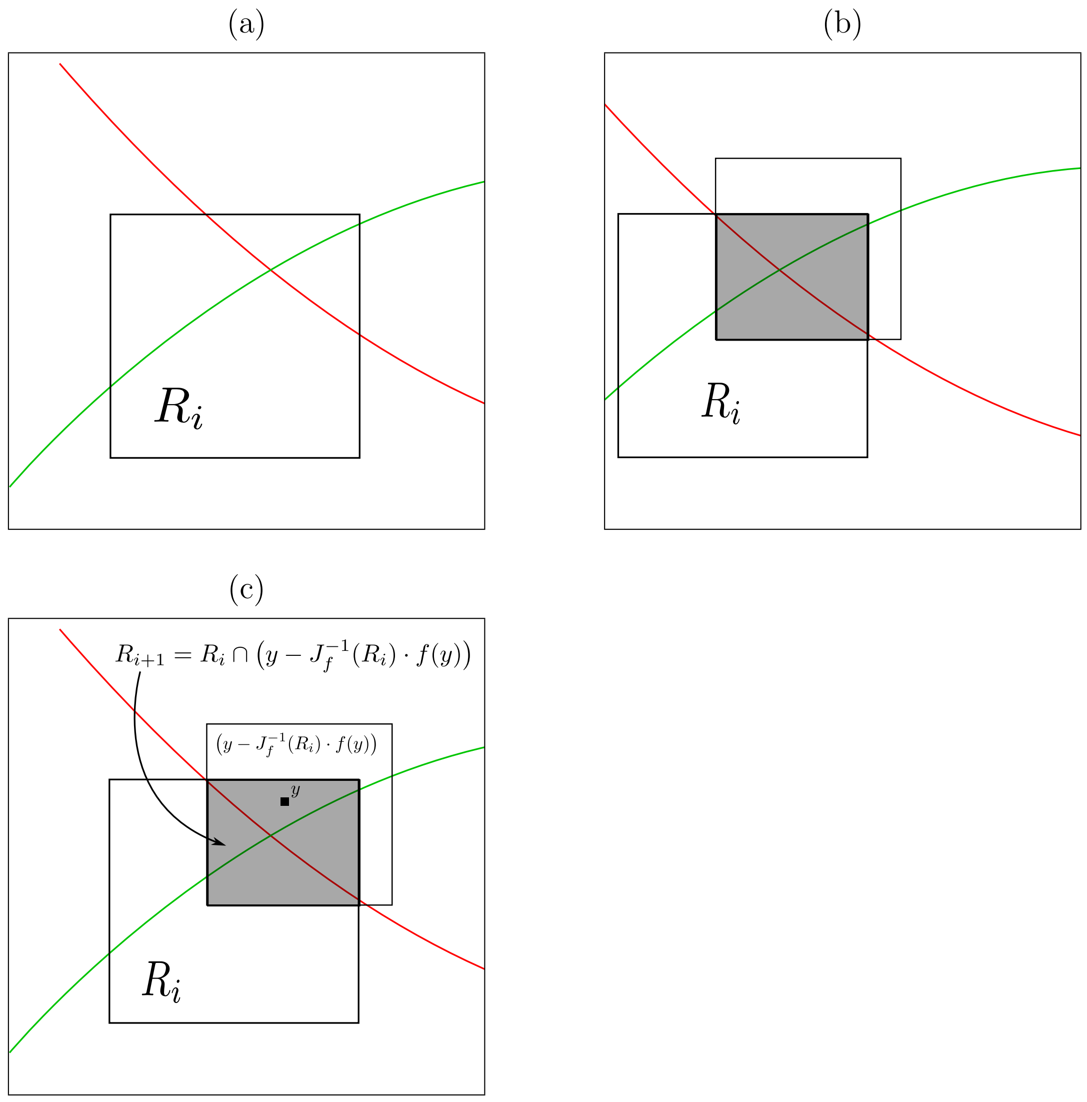
(a)
R_i
(b)
R_i
(c)
$R_{i+1} = R_i \cap \left(y - J_f^{-1}(R_i) \cdot f(y)\right)$
$\left(y - J_f^{-1}(R_i) \cdot f(y)\right)$
y
R_i

Figure 3.6: (a) A Newton step will be attempted in R_i. (b) The resulting interval and its intersection with R_i. (c) Assign R_{i+1} to the intersection of R_i and the resulting interval.

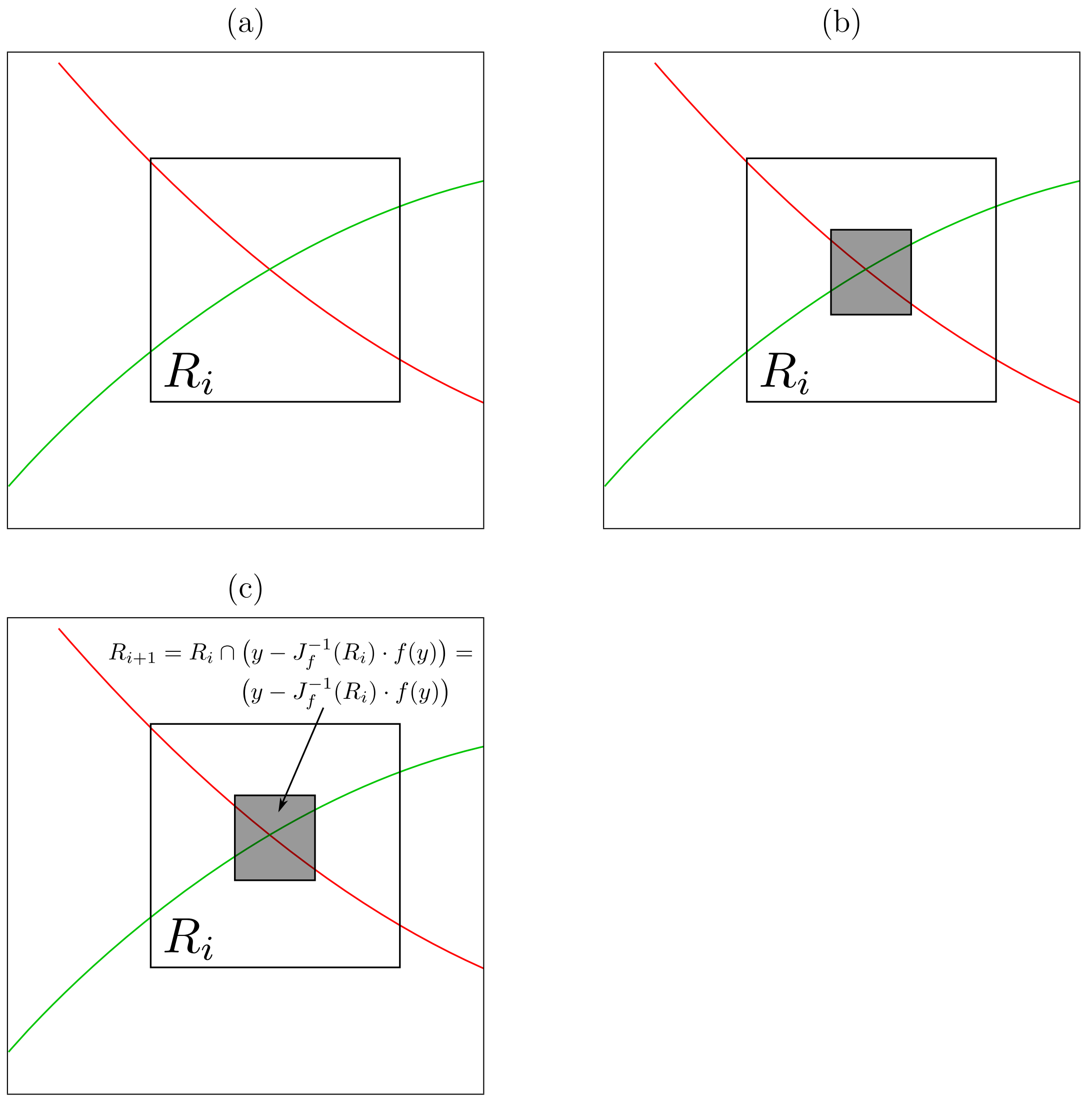
(a)
R_i
(b)
R_i
(c)
$R_{i+1} = R_i \cap \left(y - J_f^{-1}(R_i) \cdot f(y)\right) =$
$\left(y - J_f^{-1}(R_i) \cdot f(y)\right)$
R_i

Figure 3.7: (a) A Newton step will be attempted in R_i. (b) The resulting interval and its intersection with R_i. It is entirely contained within R_i. (c) Assign R_{i+1} to the intersection of R_i and the resulting interval.

the angle bisectors of the tangents of f and g at q. Call those bisectors u and v. The lines $q + tu$ and $q + tv$ pass through the intersection of f and g at $t = 0$, and we want that to be the only place they intersect f or g inside C. Substitute the lines into f and g to find any other intersections. We know there is a root at q ($f(q) = g(q) = 0$) so we algebraically remove the known root by eliminating the constant term and dividing by t to give $F_u(t)$ and $G_u(t)$. Solve $F_u(t) = 0$ and $G_u(t) = 0$ using univariate root isolation, along the interval of t where $F_u(t)$ and $G_u(t)$ lie inside C. Do so similarly with v. If there are no roots, then the cell has been verified and q is the only intersection in the cell. If not, take the smallest positive root and largest negative root of each polynomial r_p, r_n. Split the cell with the perpendicular bisector of $(q + r_p u)$ and q and also with the perpendicular bisector of $q + r_n u$ and q. Do so similarly with the other bisector v. In the resulting cell containing q, the angle bisectors of the tangents to f and g at q will intersect the curves only at q.

3.4 Curve-Curve Isolation

If the parity of the number of intersections of f and g within cell C is even, the curves either don't intersect or intersect an even number of times. In the first case we want to separate the segments of f and g into different sub-cells. In the second case we want to separate intersections into different sub-cells,

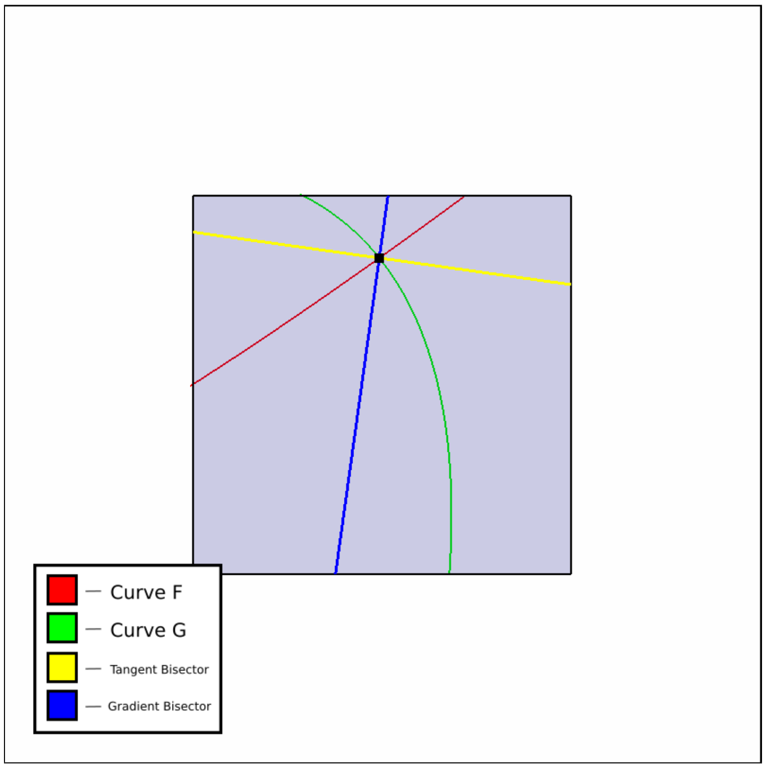

Figure 3.8: The intersection has been isolated, and the bisectors do not intersect the curves between the intersection and the boundary. No intersection encasement is needed, the intersection is the only one in the cell.

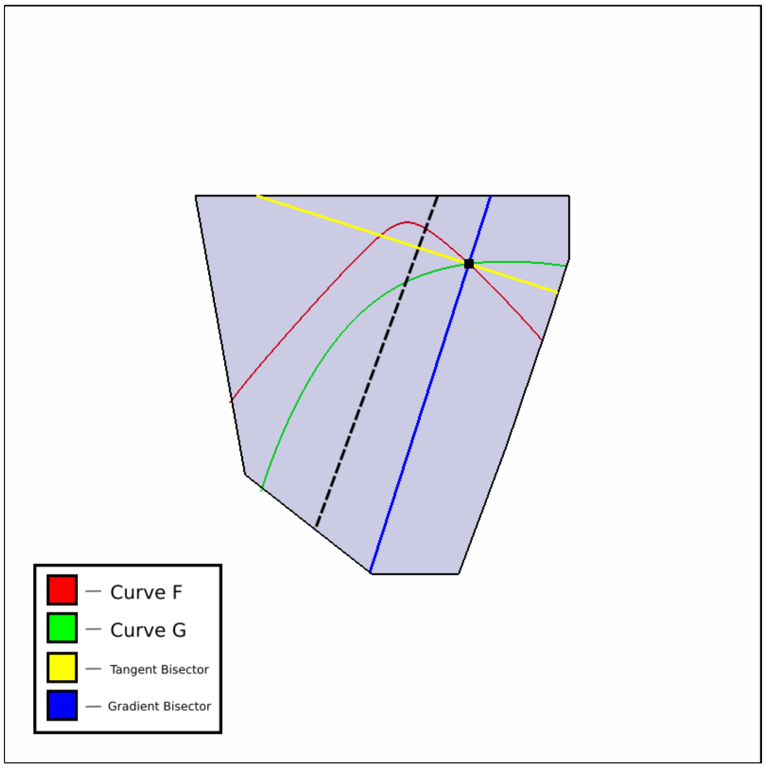

Figure 3.9: The left bisector has intersected the red curve. To ensure a cell where the bisectors do not intersect the curves before the cell boundary, split between the intersection of f and g and the first intersection of the bisector and the red curve.

until some sub-cell has odd parity. We generate the splitting line for the first case in the following manner:

1. Let (a, b) be the closest pair of boundary intersection points $a \in f$ and $b \in g$ and (c, d) be the other pair.
2. Let p be a point calculated to be between c and d and equidistant from f and g. The notion of equidistant from f and g is an approximation and depends on the orientation of the curve segments at the boundary, but we leave the details of those calculations for later (Sec. 3.4.1).
3. Solve for the intersection of the linear approximations of f and g at p: $f(p) + \nabla f(p) \cdot v = g(p) + \nabla g(p) \cdot v = 0$, for v.
4. $L = (p, v) = \{p + tv : t \in \mathbb{R}\}$

If this splitting line results in multiple components of the same curve, then apply self isolation. If the segments have a two or more intersections, the splitting lines converge on the farthest pair. If the splitting lines fail to separate the two curves after a fixed number of recursive calls, the cell is passed to Refinement and Subdivision.

3.4.1 Splitting Line Generation

When a cell contains two curves that do not alternate along the boundary, we know that the parity of the number of intersections in that cell is even. A valid encasement has no cells containing two curves, with even intersection parity, therefore these cells are in violation of the encasement constraints and must be subdivided. This leads to the question of how to subdivide the cell in the most efficient manner. Because the inputs are assumed to be generic, we can infer that the likelihood of there being 2, 4 or more intersections in a cell that is relatively refined to be low. This drives the effort towards trying to generate a splitting line that most effectively separates two curve segments that do not intersect inside the cell. Our strategy for generating splitting lines tries to achieve this. To separate segments of f and g, split the cell with a line through their closest point in the cell and whose normal is the average of their gradients at this point. Rather than computing the closest pair of points of f and g in the cell, we make an approximation near the boundary intersections.

Given a cell containing two curve segments whose intersection parity is even, we generate splitting as such: In general you can give an ordering to the intersections of f and g with the boundary that is equivalent to $(f_0\ f_1\ g_0\ g_1)$ where f_i is an intersection of f and g_i is an intersection of g. We consider adjacent pairs, with one point from each curve, either $(f_1,\ g_0)$ or $(f_0,\ g_1)$. Which ever has the shorter distance from each other around the boundary will be the points we work with. From the chosen pair call the point on f: p and call the point on g: q. For the purpose of the approximation we are about to make, we need to ensure that $f(q) > 0$ and $g(p) > 0$: both functions are positive on the region between the curves inside the cell. We flip the signs of f and g accordingly. We then compute the normalized inward tangents to the curves at their respective points: $f_t(p)$ and $g_t(q)$. Inward meaning in the direction entering the cell. We recognize 4 cases with respect to the projection of the tangents on the line segment pq:

- Case 1: $f_t(p) \cdot \vec{pq} > 0$ and $g_t(q) \cdot \vec{pq} < 0$ (a.k.a. the curves are converging towards each other)
- Case 2: $f_t(p) \cdot \vec{pq} < 0$ and $g_t(q) \cdot \vec{pq} > 0$ (a.k.a. the curves are diverging away from each other)
- Case 3: $f_t(p) \cdot \vec{pq} > 0$ and $g_t(q) \cdot \vec{pq} > 0$ (a.k.a. the curves are skew, and g projects onto f inside the cell)
- Case 4: $f_t(p) \cdot \vec{pq} < 0$ and $g_t(q) \cdot \vec{pq} < 0$ (a.k.a. the curves are skew, and f projects onto g inside the cell)

We view the skew case as a sort of symmetry, so long as we know which curve projects onto the other in the cell, the way we handle this case is the same. Assume f projects onto g, by project we mean that the ray from p along the direction of the normal to the curve, $\nabla f(p)$, intersects g inside the cell. Case 1 and 2 (a.k.a diverging and converging) also have somewhat of a symmetry. In either case neither curve projects onto each other along their normal direction. In this case we consider the direction of projection to be $\vec{pq}$

In either case we have one of the boundary points on the curve, call it u, and the direction of projection, call it v. We are looking for a point p on the line $u+tv$ that is approximately equidistant from f and g. The assumption is that this point lies on a line that separates f and g in roughly equal proportion.

To find p we construct linear approximations of f and g about the point u and solve for the point on line $u + tv$ equidistant from those two lines.

The linear approximations about the point u are given by:

$$f^u(p) = f(u) + \nabla f \cdot (u - p)$$

$$g^u(p) = g(u) + \nabla g \cdot (u - p)$$

Where the distance to the linear functions $f^u(p)$ and $g^u(p)$ are:

$$\text{dist}(p, f) = \frac{f^u(p)}{|\nabla f(p)|} \qquad \text{dist}(p, g) = \frac{g^u(p)}{|\nabla g(p)|}$$

Setting these two distances equal to each other we can solve for t in closed form.

$$\text{dist}(p, f) = \text{dist}(p, g) \implies$$

$$\frac{f(u) + \nabla f(u) \cdot tv}{|\nabla f(u)|} = \frac{g(u) + t\nabla g(u) \cdot tv}{|\nabla g(u)|}$$

One caveat is that the point generated may not be within the given cell. If this happens we choose the point on the line closest to that point that is also inside the cell.

At the point $p = u + tv$ we need to find the direction of the splitting line, call it v. The strategy here is to again linearize the two functions, but now about the point p. Find the intersection of these two linear approximations, and the direction from p to this intersection is the direction of the splitting line. If the two curves truly intersect transversely in this cell, this approximation of the direction of the intersection becomes more accurate as the area of the cell decreases, f and g appear linear in a neighbourhood of a transverse intersection. If the curves do not intersect in this cell, then a sequence of splitting lines generated where at least one point on each line is between curves f and g will eventually isolate the curves to separate cells. The linearization of the functions about p are given by:

$$f^p(q) = f(p) + \nabla f \cdot (p - q)$$

$$g^p(q) = g(p) + \nabla g \cdot (p - q)$$

The intersection of these lines lies on some line $p + tv$, we can arbitrarily solve for the representation where $t = 0$, so we solve for v:

$$\begin{aligned} f^p(p+v) &= 0 \\ g^p(p+v) &= 0 \end{aligned}$$

And thus:

$$\begin{aligned} f^p(p) + \nabla f \cdot v &= 0 \\ g^p(p) + \nabla g \cdot v &= 0 \end{aligned}$$

This is a very simple linear system, and we can solve it with direct methods. Once a valid p and v are found, we construct the line $p + tv$ and split the cell by this line.

The method described above is a latter iteration of several heuristically tested ways of generating splitting lines and we in no way claim that it is optimal.

Several other strategies have been considered and are planned for future work (Sec. 6.1). Much effort and thought has gone into the generation of splitting lines, because it is arguably the most frequent operation in the encasement algorithm, both from intuition and profiling of the algorithm.

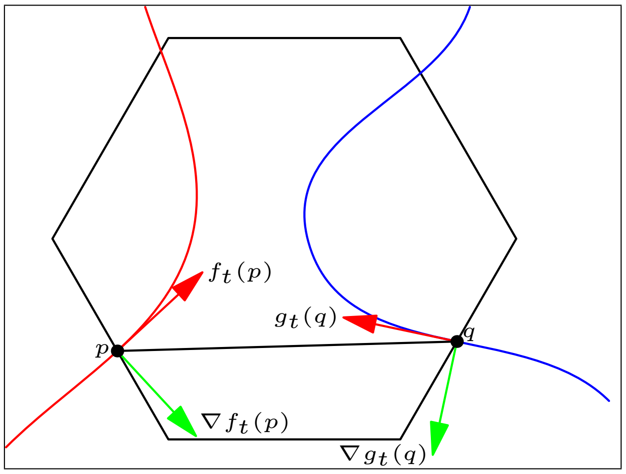

Figure 3.10: Case 1: f_t and g_t converge

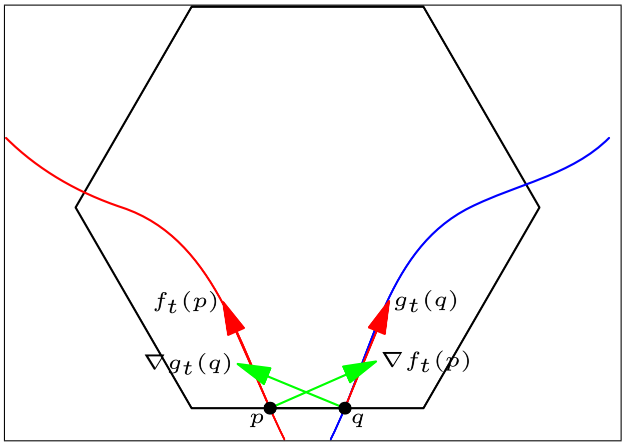

Figure 3.11: Case 2: f_t and g_t diverge

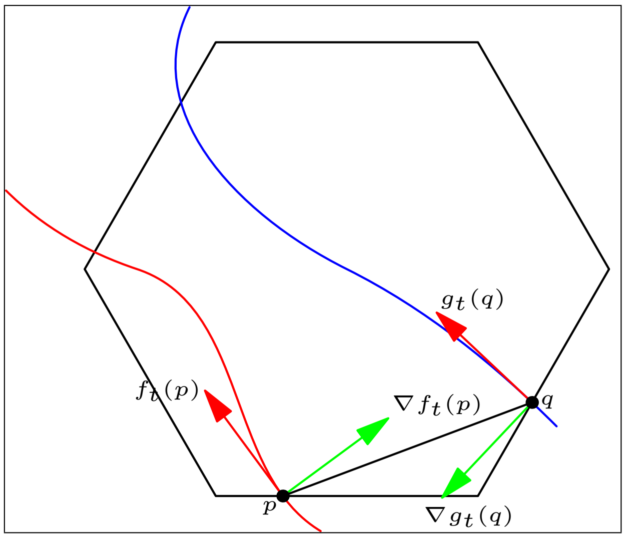

Figure 3.12: Case 3: f_t and g_t are skew

3.5 Subdivision and Refinement

If curve-curve isolation fails to separate two curves that seemingly don't intersect, procedures we refer to as refinement and subdivision are invoked to ensure termination of the algorithm. We want to refine a cell C containing f to a smaller cell that more tightly surrounds the curve. That is to say, the minimum distance from any point on f to the boundary of the resulting cell is smaller than its minimum distance to C.

Subdivision: Curve f intersects cell C at points a and b along the boundary. We wish to subdivide this curve by splitting the cell so that the segments in the resulting cells have roughly the same length. We split the cell C with the perpendicular bisector of ab. Repeatedly subdividing the resulting cells results in a finer and finer linear approximation of f by line segments ab, where a and b are the intersection of f with the boundary of the new cell.

Refinement: We identify intersections with the boundary, a and b and their midpoint $m = (a+b)/2$. Let $L = (m, (a-b)^{\perp}) = \{m + t(a-b)^{\perp} : t \in \mathbb{R}\}$ be the perpendicular bisector of ab. Compute its intersections with the boundary, c and d at $t = t_c$ and $t = t_d$. Let c be the endpoint of the intersection of L and the boundary such that mc intersects f. Compute the intersections of L and f on the segment mc and choose its smallest root r in its interval $[0, t_c]$. Let $p = m + (0.9r + 0.1t_c)(b-a)^{\perp}$ and v be $\nabla f(p)^{\perp}$. Split the cell with line $L_{\text{top}} = (p, v)$. Split the resulting cell with $L_{\text{bottom}} = (0.9m + 0.1d, (a-b))$. The splits result in cutting in the tangent direction to f at p just above the curve f and cutting in the mean tangent direction $(a-b)$ just below the curve.

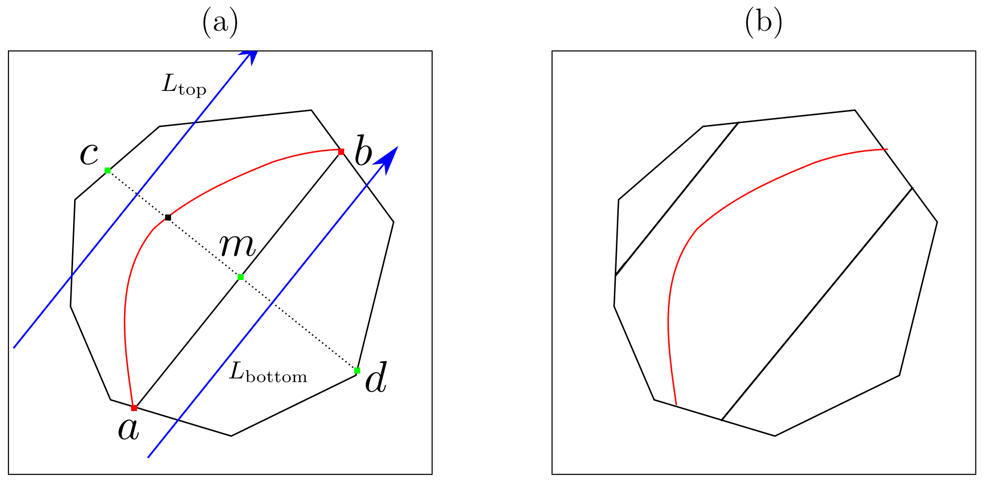

Figure 3.13: (a) Frame of reference for generating splitting lines for refinement. (b) Resulting cell after refinement.

CHAPTER 4

Implementation

The following are various in-depth descriptions and discussions of techniques developed in the process of implementing the encasement algorithm. Most techniques are the end result of a few attempts to solve the underlying problem. These techniques are not only relevant to the encasement algorithm, but are generic techniques that can be applied in any future algorithms developed in ACP 4.1.

4.1 ACP

The **ACP** library [MST13] is designed to provide efficient and exact evaluation of geometric primitives on perturbed inputs. The fundamental object of arithmetic in the ACP library is called a **Parameter**. A parameter represents a real number using a floating point interval that contains it. The default is double-float, but it can use higher precision floats (MPFR) [FHL+07] if needed. All inputs to any program using ACP are stated at double precision numbers $d_0, ..., d_n$. Each one is given a perturbation $d_i^* = d_i + \text{rand}(-\delta, \delta)(1 + |\text{d}_\text{i}|)$ (where $\delta = 10^{-8}$ for this particular application) and its initial representation is a trivial interval $[d_i^*, d_i^*]$. Beyond the initial perturbation, the input parameters $d_0^*, ..., d_n^*$ are treated as exact input. The endpoints of the interval are initially double precision numbers, but they can be extended to higher precision when needed. All arithmetic on parameters is computed using interval arithmetic, implemented in the ACP library.

The next fundamental object in ACP is known as a **Geometric Object**. A geometric object is specified by a list of parameters. For instance, points in $\mathbb{R}^n$ are specified by their coordinates $(x_0, ..., x_n)$. A line in $\mathbb{R}^2$ is represented by $(p, v) = ((p_x, p_y), (v_x, v_y)) = p + vt$ for $t \in \mathbb{R}$. A polynomial in $\mathbb{R}[t]$ is represented by its coefficients $(c_0, ..., c_n) = c_0 + c_1 t + c_2 t^2 + ... + c_n t^n$. These parameters may be inputs to the program, wherein the object is referred to as an **Input Object**. The parameters of the geometric object may also be represented in terms of parameters of other geometric objects. For example, a point may be represented by a the intersection of a bivariate polynomial and a line, where the parameters of the line are substituted into the coefficients of the polynomial.. Its coordinate is a single parameter t on a parametric line L where t is a root of the polynomial that is the result of substituting the line equation into the bivariate polynomial. As well, a point $p \in \mathbb{R}^2$ can be represented by the common zero $x = r, y = s$ of two bivariate polynomials with parameter coefficients $f(x, y), g(x, y)$ where $f(s, r) = g(s, r) = 0$. These points are the fundamental point of investigation for the encasement calculation. We refer to this objects as **Derived Objects**. The objects that are required for the computation of a derived geometric object are referred to as the **antecedents** of the derived object. In general the antecedents of a geometric object may be derived objects themselves, resulting in a tree of antecedents involved in the computation, with the eldest antecedents of any derived object being input objects. Every object provides a method to **calculate** its parameters from the parameters of its antecedents at the prevailing level of precision. For an input object, the calculation is simple: just provide the input parameters padded with enough digits of precision to match the prevailing the precision. For a derived object, first it must ensure that its antecedents are calculated at the prevailing precision, then it can proceed with its specified logic for computing its coordinates.

Branching of logic in a computational geometry program is determined by the sign of **Primitives** (interpreted as truth values), whose inputs are geometric objects. For instance, a primitive **Orient2D** is used to determine whether a list of 2D points, (a, b, c), is oriented clockwise or counterclockwise in the plane. It is defined to be: $\mathrm{sign}((\mathrm{b} - \mathrm{a}) \times (\mathrm{c} - \mathrm{a}))$. When a primitive is invoked, it performs the intended operation on the geometric objects and evaluates the sign of the result. ACP uses a strategy of lazy evaluation, in which a derived geometric object is not computed until a primitive involving that object is invoked. If the sign of a primitive is ambiguous (i.e. the interval of the resulting parameter contains 0), it is recomputed at a higher level of precision. The number of bits of precision begins at 53-bit double precision and doubles at each primitive evaluation failure, until it fails with 848 bits of precision. At this time the computation gives up, and is unable to determine the sign of the primitive. The primitive may just require more precision to finish the computation, or the inputs to the primitive might be degenerate. We avoid degenerate input by perturbation, but there is still the possibility that perturbed inputs contain degeneracies. However, if the same primitive fails with different different perturbation seeds, the probability of the perturbed input being degenerate gets exponentially smaller. A complete solution would restart the computation with twice the number of initial random bits to each input, as well as twice the maximum precision allowed. We do not currently implement this approach.

A primitive may be degenerate because of an **Identity**. An identity results when the symbolic expression of the primitive is identical to the zero polynomial, due to special relationships between the inputs. For instance: let a and b be two points in the plane, and let $c = (a + b)/2$ a derived geometric object that is the midpoint of a and b. If we evaluate **Orient2D** on points a, b, c the result will be identically zero ANY level of precision because the point c

is derived to lie on the line going through a and b. Within the encasement algorithm, we currently handle identities with specialized logic.

4.2 Encasement Data Structure

The structure of the encasement is represented through a standard half-edge Doubly Connected Edge List (DCEL) structure. Each directed half-edge is defined by a parametric line $L = p + tv$ and interval $[t_0, t_1]$ defining the segment of the line where this edge exists. The two vertices of an edge are the points $L(t_0)$ and $L(t_1)$. Each vertex is the intersection of two lines that define two edges. Each line is either a splitting line or a line induced by the original constraints of the initial convex polygonal region. A face is the interior of a closed chain of edges. One face exists at the onset of the algorithm, the interior region, whose boundary is the initial convex polygon. Each subsequent face generated by a splitting line, and the resulting faces are a partition of the interior of the parent face. Thus the DCEL induced by encasement operations is a binary space partition of the original convex polygonal region. Storing parent and child pointer in each face gives rise to a natural way to trace the history of the algorithm.

4.3 Primitives

As in any algorithm in computational geometry, the branching of control logic in the encasement algorithm is determined by the signs of geometric predicates, which we refer to as “Primitives” within the software and from hereon. This section presents the necessary geometric primitives used in the encasement algorithm.

Because ACP uses a strategy of lazy evaluation combined with adaptively increasing the precision of derived objects, it is necessary to encapsulate all primitives into a functional object structure. The value of an object is not calculated at any precision until the first primitive that depends on the object is invoked. Thus even the simplest of primitives need to be encapsulated in this structure. The following is a listing of all primitives used in the algorithm as well as their derivations.

Orient2D (PV2 a, PV2 b, PV2 c): Determines if the vertices of triangle abc are given in clockwise or counterclockwise order. Given by the sign of $((b-a)\times(c-a))$. A positive sign implies clockwise order, while negative implies counterclockwise.

LeftOf (Line l, PV2 o): Determines if point o is to the left of the line given by $p+tv : t\in\mathbb{R}$. Given by the sign of $(o-p)\times v$. A positive sign indicates that the point o is to the right of the line, while negative indicates left of the line.

YOrder (PV2 a, PV2 b): Compares points a and b by their y coordinate. Given by the sign of $a.y - b.y$. Negative sign indicates that the y coordinate of a is ordered before the y coordinate of b in ascending order.

XOrder (PV2 a, PV2 b): Compares points a and b by their x coordinate. Given by the sign of $a.x - b.x$. Negative sign indicates that the x coordinate of a is ordered before the x coordinate of b in ascending order.

LessThan (Parameter a, Parameter b): Determines if $a < b$. Given by the sign of $a - b$. Negative sign indicates that $a < b$.

Dot (PV2 a, PV2 b): Determines the sign of the dot product of vectors a and b. Given by the sign of $a.xb.x + a.yb.y$.

Side (PV2 a, PV2 b): Determines the sign of the cross product of vectors a and b. Given by the sign of $a.xb.y - a.yb.x$.

PolySign (Poly f, PV2 p): Determines the sign of the value of the polynomial f evaluated at p. Given by the sign of the parameter that is the result of evaluating $c_{00} + c_{10}(p.x) + c_{01}(p.y) + c_{11}(p.x)(p.y) + ...+$ etc. Where $(p.x, p.y)$ are the coordinates of p and $c_{00}, c_{10}, c_{01}, c_{11}, ...$ are the coefficients of f.

Sign (Parameter x): Determines the sign of a single parameter, x. Simply -1 if $x < 0$ or 1 if $x > 0$.

Shorter (PV2 a, PV2 b, PV2 c, PV2 d): Determines if the line segment between a and b is shorter than the line segments between c and d. Given by the sign of $((b-a) \cdot (b-a)) - ((d-c) \cdot (c-d))$, where negative sign indicates that ab is shorter than cd.

CCWOrder (PV2 a, PV2 b): Determines the ordering of vectors a and b when sorted about the origin considered by the angle they make with the vector $(1,0)$ measured counterclockwise from $(1,0)$ to a and similarly for b. If the sign of $a.y$ and $b.y$ differ, then the order is given by the sign of $a.y - b.y$, where positive means a is ordered before b. If the sign of $a.y$ and $b.y$ are the same, then the order is given by the sign of $a \times b$, where positive means a is ordered before b.

4.4 Software and Robust Curve Tracing

We have provided a companion software for visualization of the algorithm, Along with the C++ implementation of the algorithm itself. Visualization of general algebraic curves and even more so, **robust** visualizations of algebraic curves is of some interest. Our visualization is efficient at a variety of scales, but not necessarily robust at this point. This is due to the nature of the our curve tracing implementation. However, we propose a framework for constructing robust visualizations of curves given a small modification to the encasement algorithm.

Given a curve f, a cell C and the two intersections of f on the boundary a and b, the most simple visual approximation of the curve in this cell is a line segment from a to b. One might wish to see the more complicated and nuanced behavior of the curve in this cell, as well as its interactions with possibly another curve in the cell. We could accomplish this by sampling the curve at intersections with a line perpendicular to ab in the space between a and b inside the cell. The first caveat is constructing an exact point on the curve requires calling the univariate solver. This can become rather expensive depending on the number of samples requested and the total number of cells in the visualization. Although the arrangement is static, visualizations of the solution can be a very dynamic if one wishes to zoom in and produce a more refined picture of a smaller region. The second caveat is that when intersecting one of these perpendicular lines between a and b, you might possibly encounter more than one intersection along the interval. This presents a problem because ostensibly when you have points on the curve a, b, and mid (a point between a and b), you would visualize the curve with a line segment between a and mid and then one between mid and b. But there is no obvious way to generalize constructing line segments in a meaningful way when you have multiple intersections along this line. We propose an approach to deal with these two problems and create a method for robust and efficient visualizations. Given the curve and cell described above, assume f is monotonic with respect to some direction v inside the cell. That is to say either $\nabla f \times v > 0$ or $\nabla f \times v < 0$ on the entire cell. If this is true than any line parallel to $v^{\perp}$ that intersects the cell and separates a and b on either side will intersect f exactly once. Thus if we can further constrain the segments of curves in encasement cells to be monotonic to some direction, we can construct a point on a perpendicular between a and b without the burden of not knowing the topology of the curve along this line. Doing this also gives rise to a solution to the first caveat. Because we are guaranteed one intersection on this line in the cell, we can construct the two intersections of the perpendicular line and the cell, c and d at t_0 and t_1 and we know the interval $[t_0, t_0]$ contains and isolated the t where the line intersects the curve. Thus we are not required to call the univariate solver, and can use direct numerical methods to refine this interval to the required precision of the visualization. This proposed solution is intended to be included in future versions of the software.

In the current software, we implement curve tracing as such: Starting at a point on the endpoint p_0 and a time step t we do the following at iteration i:

1. Compute $v = \nabla f(p_i)^{\perp}$, the tangent to the curve at p_i. If it is not pointing inward to the cell, flip its sign.
2. Compute $q_0^* = p_i + tv$

3. Iterate Newton's method in double precision for n iterations:

   $f_L(q_i + \Delta q) = f(q_i) + \nabla f(q_i) \cdot \Delta q = 0$

   $q_{i+1} = q_i + \Delta q$

4. $p_{i+1} = q_n$

5. Repeat until p_{i+1} falls out of the cell or a fixed maximum number of iterations have been completed

This is a standard curve tracing (homotopy / continuation) method [AY78]. It makes no guarantees that the displayed topology matches the true topology of the curve nor that tracing from intersection a on the boundary will converge to intersection b. It is however a very efficient algorithm, and for most generic scenarios give a very good approximation to the features of the curves. We vary the time step t based on the level of detail of the viewing area and adaptively cull any faces that do not intersect the viewing area. The end result is an effective visualization, that aides in inspection of inputs as well as debugging the algorithm itself.

We provide a small set of features in the prototype software. A user may input a file description of a set of curves by providing the coefficients of their polynomials as doubles. As well, we provide the option to generate a random curve of arbitrary degree. Once input is determined, the user may query for the entire encasement to be computed, or to step through each splitting line generated from the algorithm, with the option to step backwards if so desired. At any time, the user may query to iterate through the current set of faces, edges, and vertices of the DCEL structure, to inspect them without the burden of the rest of the encasement scaffolding in the way. The user may query to visualize the connected components of the critical regions if so desired, and they will be displayed alongside the current encasement structure. Using the keyboard the user can translate the center of the viewport along the plane and zoom in on areas of interest in real time. Once the entire arrangement is computed, the user may query to iterate through the intersections detected and display their locations and automatically relocate the camera to a view centered on the intersection. As well, the user may click within the initial boundary and a point location query will be performed. The arrangement cell containing the point will be identified by tracing its boundary in black. While not a complete production ready software, we provide a small but rich set of features for inspecting the interaction of two semi-algebraic curves.

4.5 Resultant Calculation

In the midst of our investigation into state of the art techniques for solving systems of bivariate equations, we discovered some methods for computing resultants that we thought helpful to implement within ACP both for the purpose of understanding the algorithm to a finer degree as well as for use in the testing and verification of our own implementation. It also gave insight

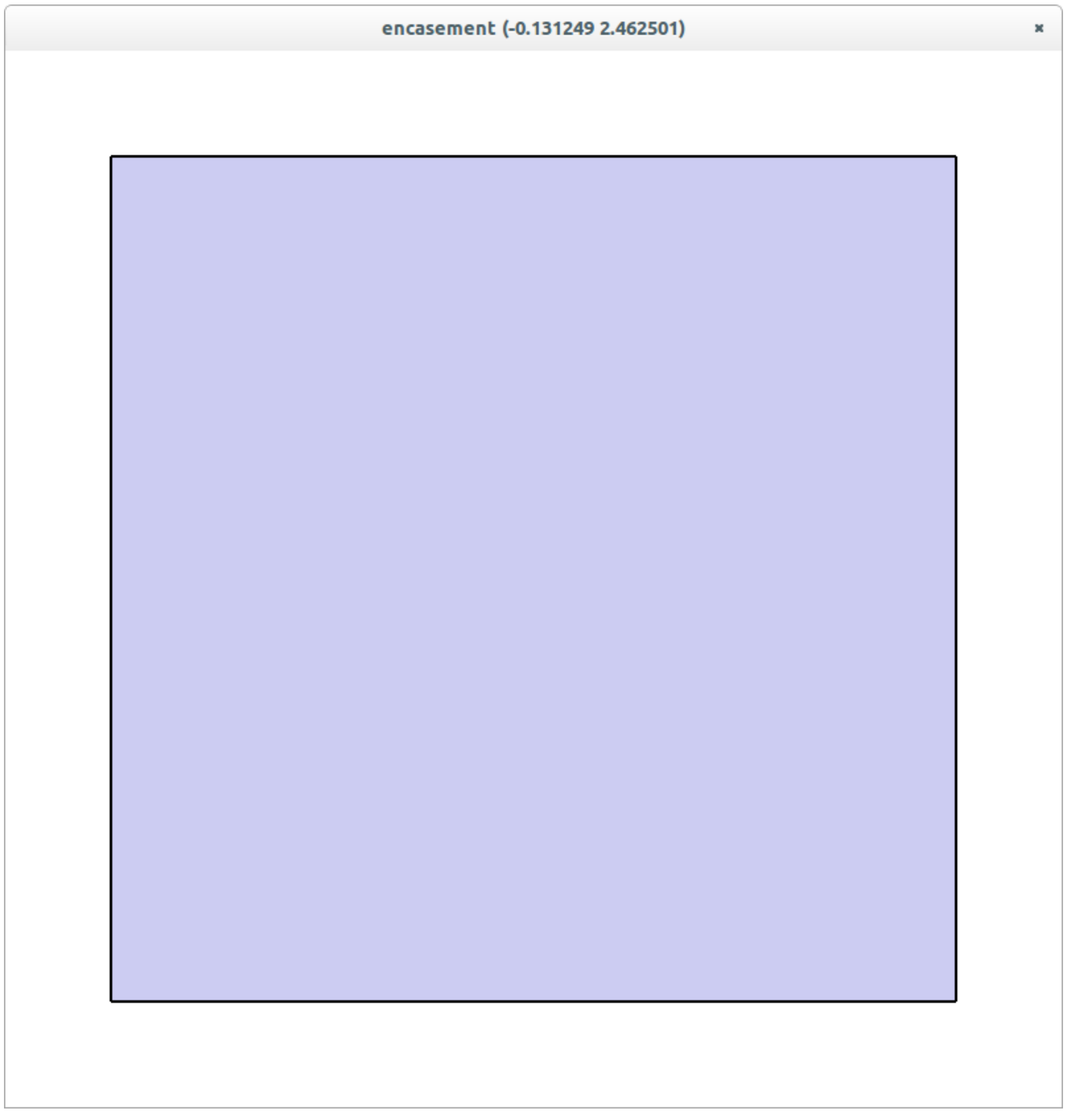

Figure 4.1: The encasement software in initial state. The initial convex region B is shown.

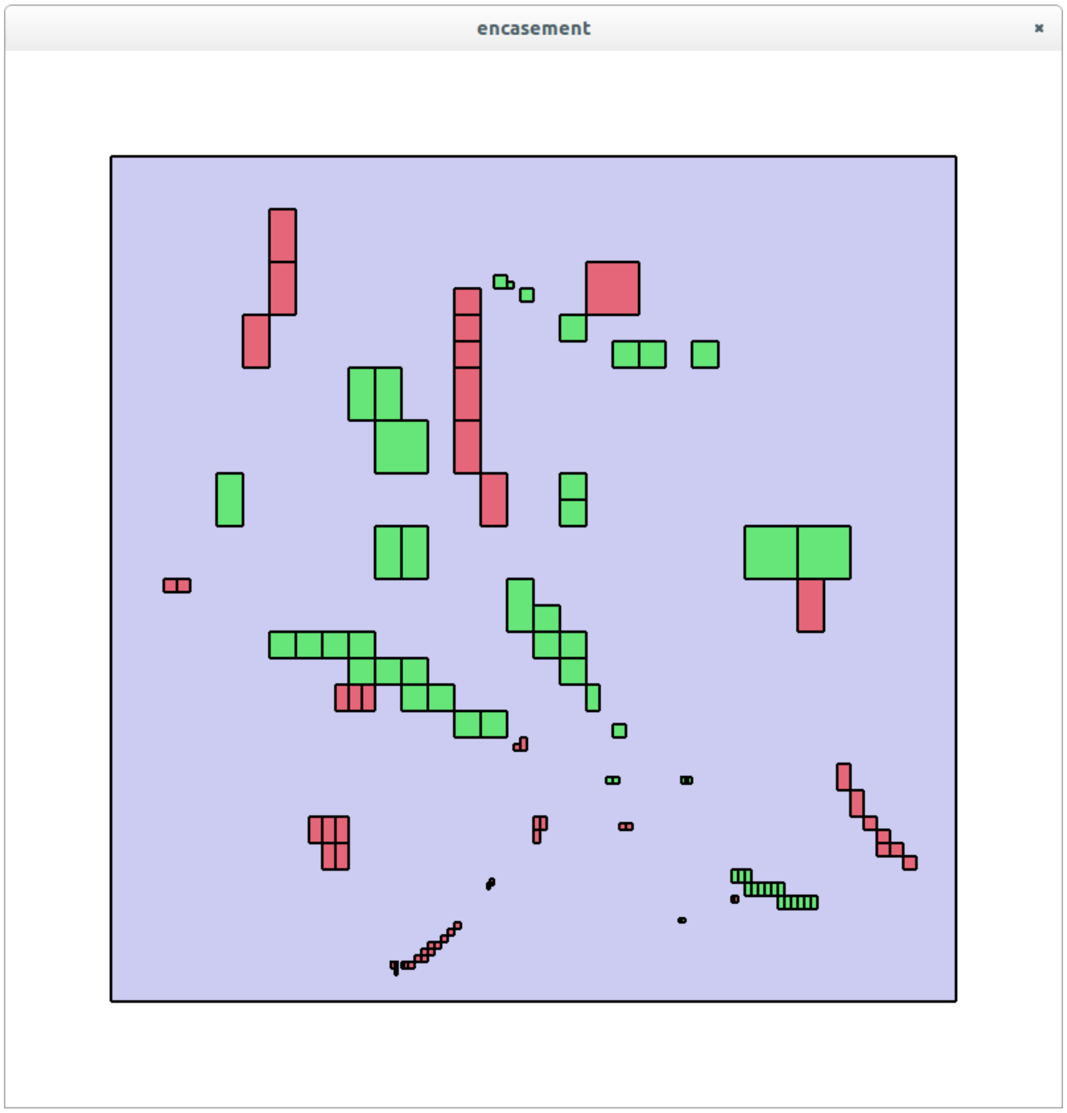

Figure 4.2: Critical regions have been identified. Critical regions of f are red and g are green.

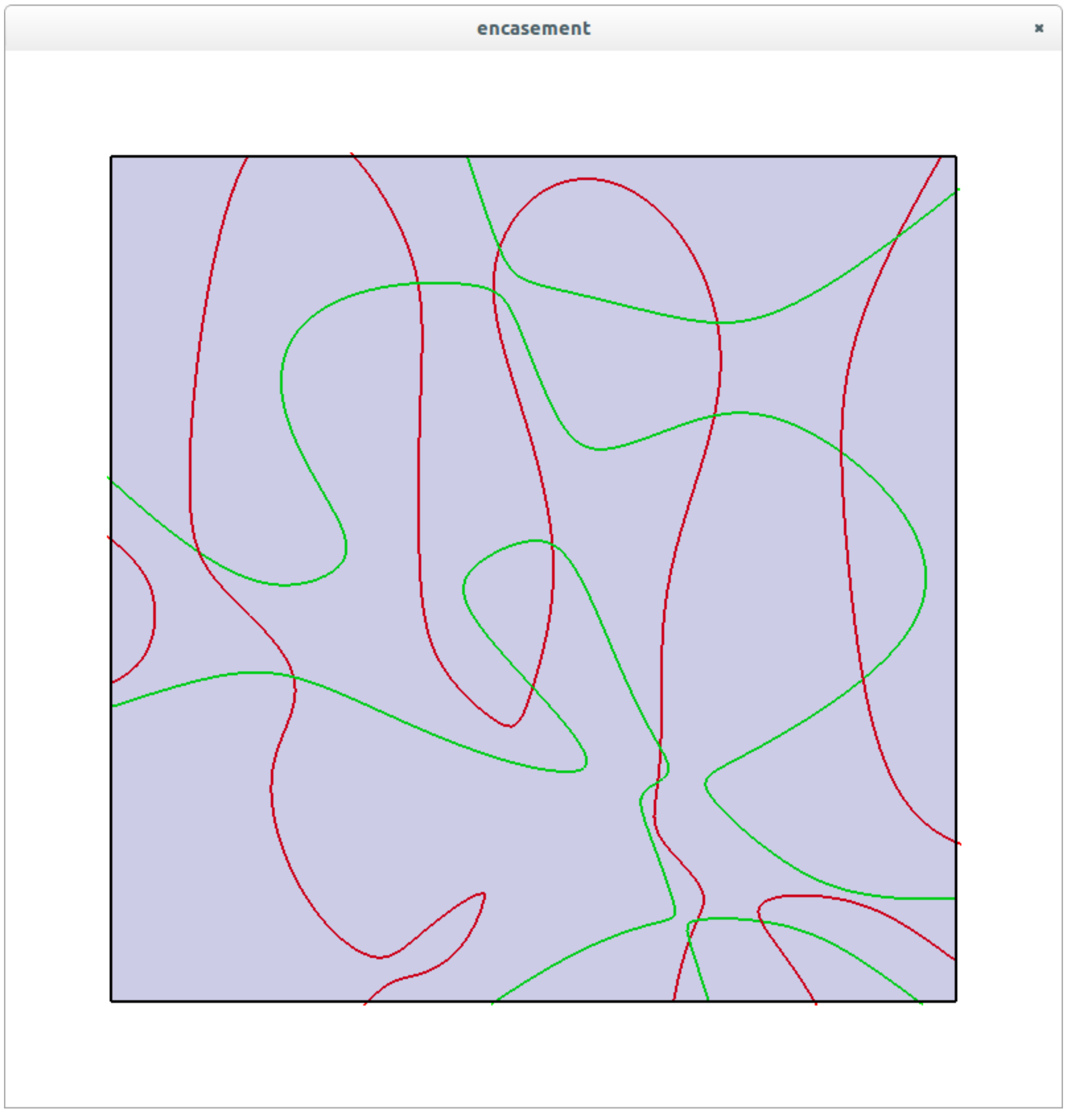

Figure 4.3: Current knowledge of all curves after intersecting with the boundary of B

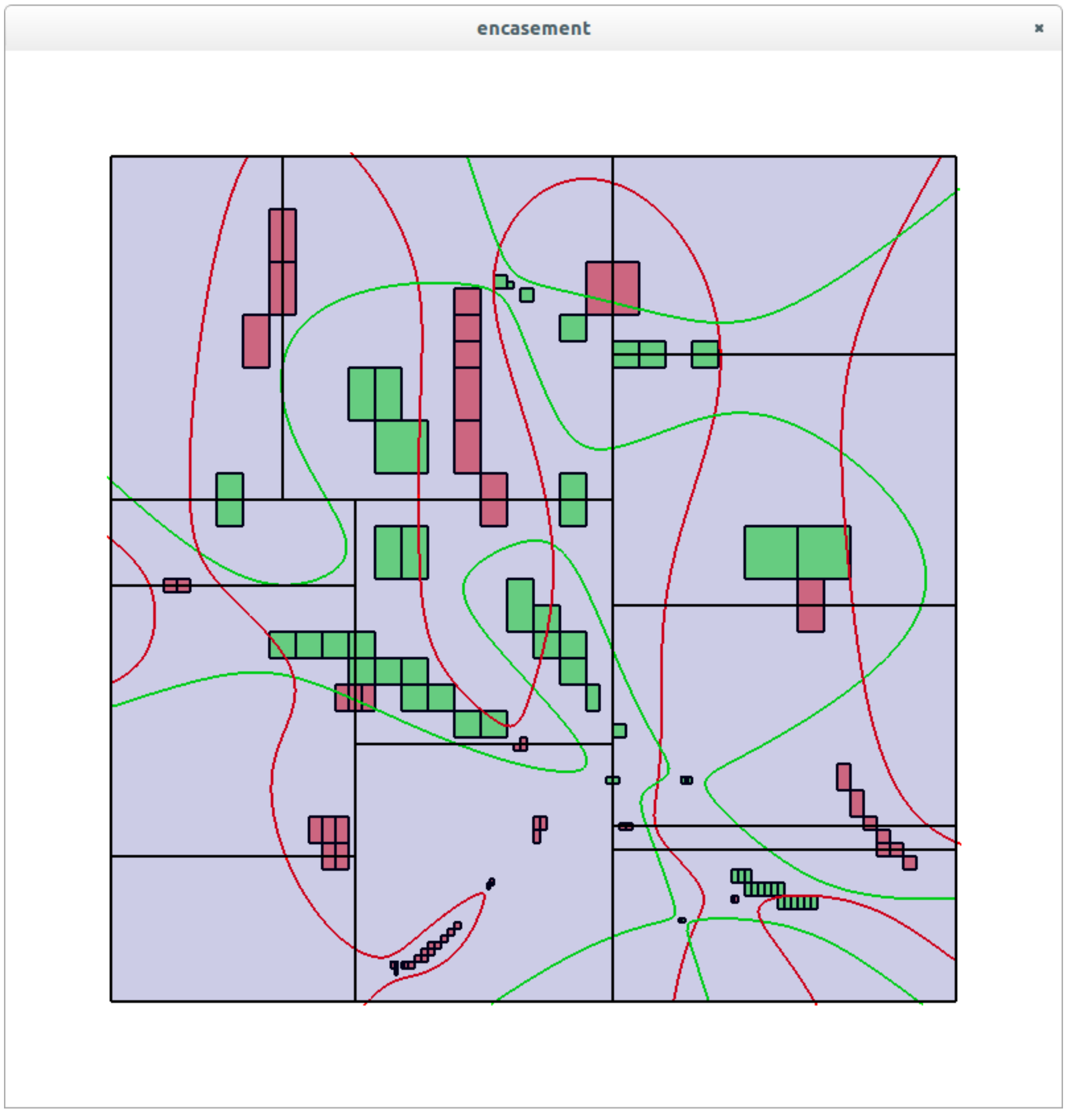

Figure 4.4: Currently splitting through critical regions of f and g

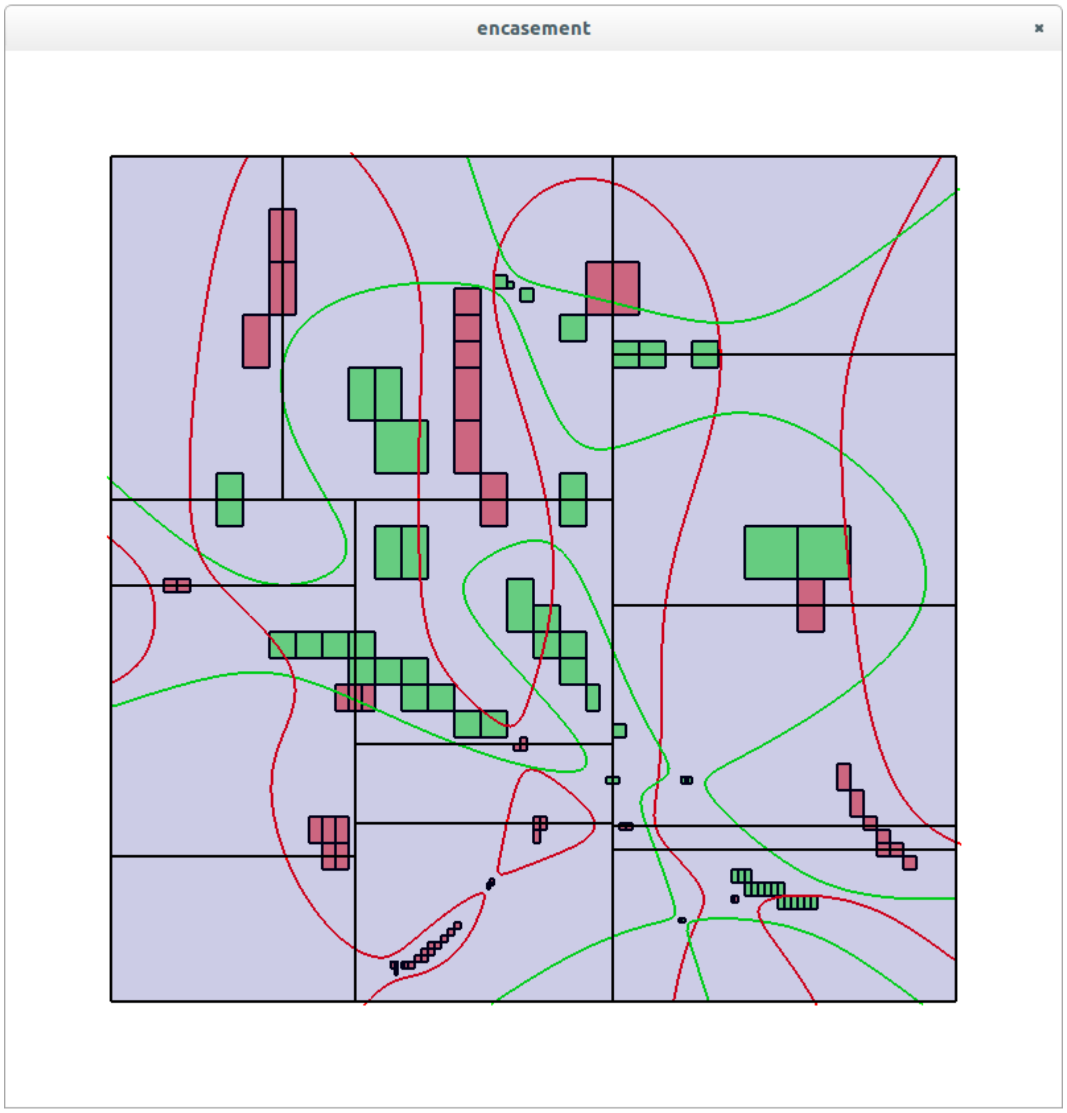

Figure 4.5: One step after figure 4.4. Splitting through the critical region in the bottom cell discovered a loop of f

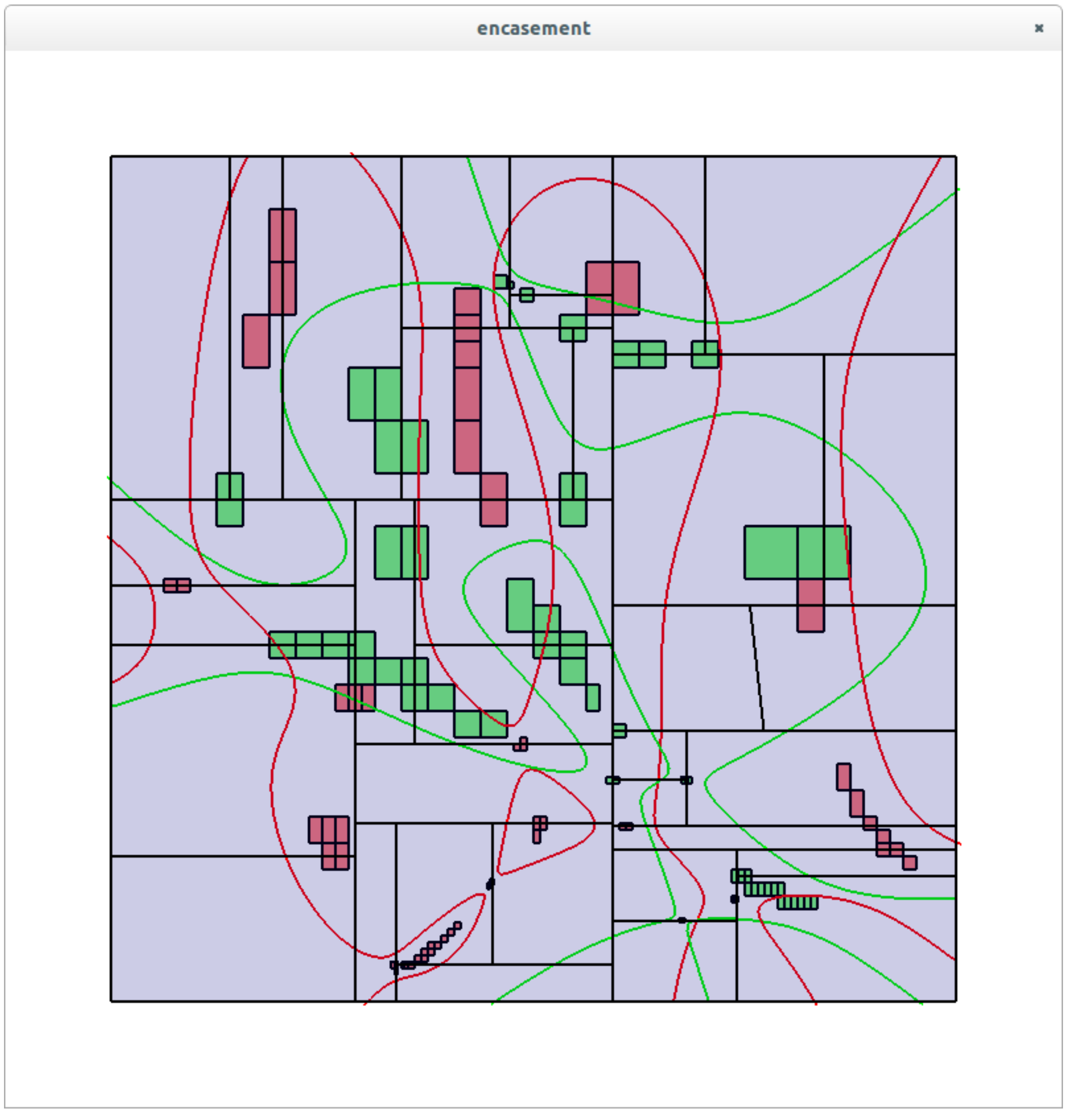

Figure 4.6: After all of the critical regions have been split, the separating splitting line is constructed.

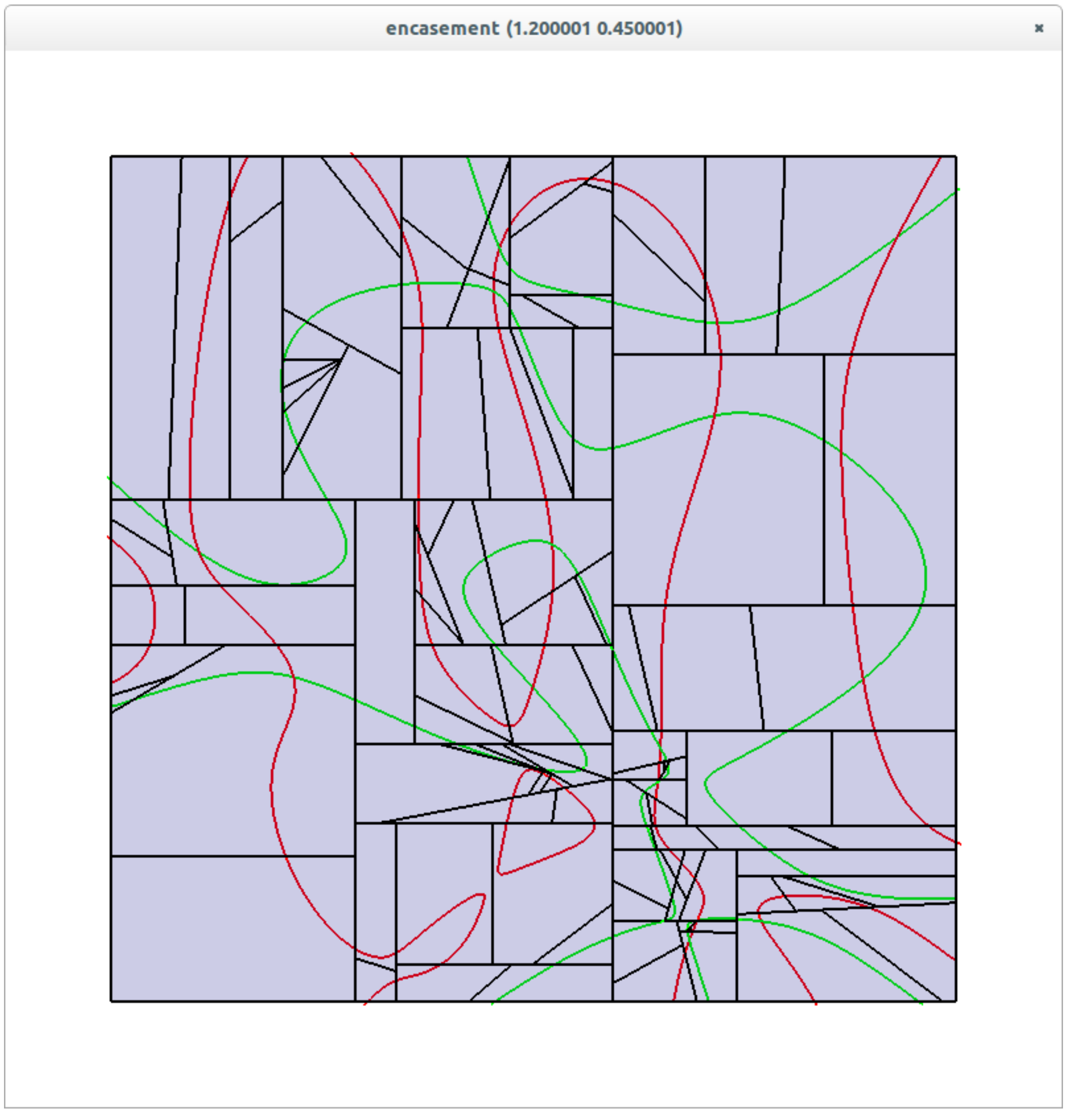

Figure 4.7: A completed and validated encasement of f and g.

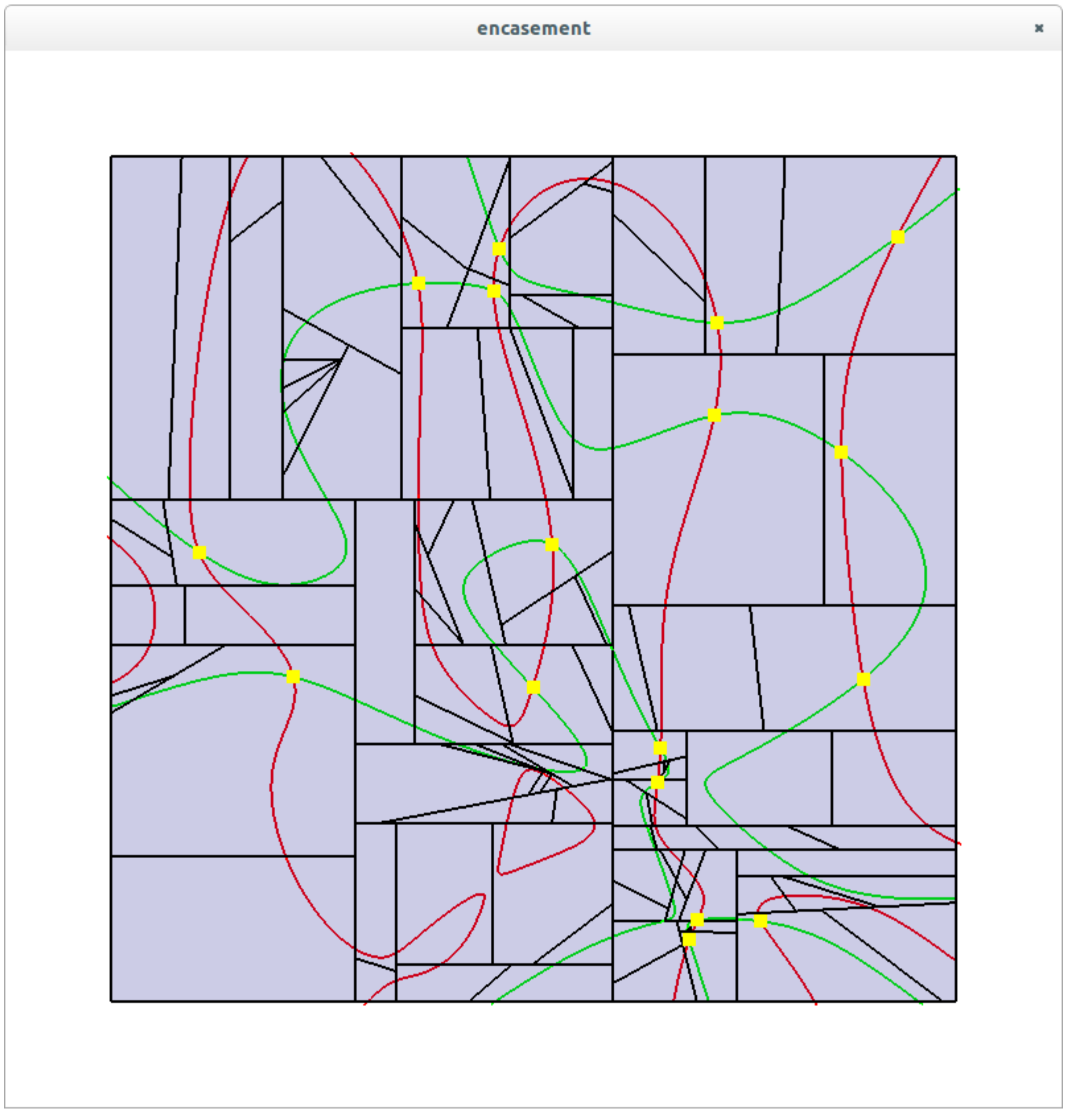

Figure 4.8: Intersections of f and g identified.

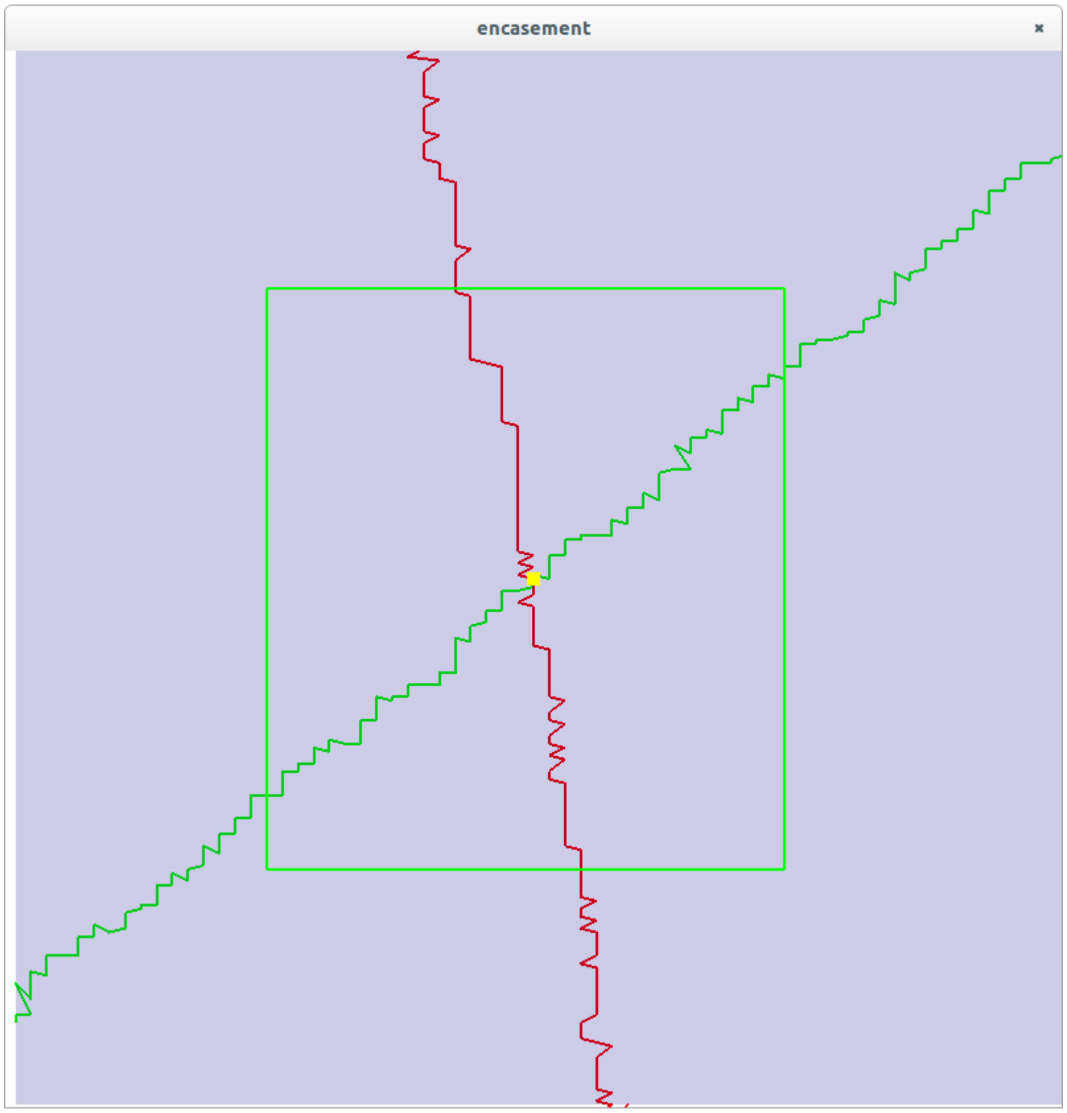

Figure 4.9: Zoomed in view of a Root2. The 2D interval is represented by the square box of width around 10^{-14}. The reason for the jagged curve tracing is because he precision of Newton's method in double at this scale is the same magnitude of rounding error.

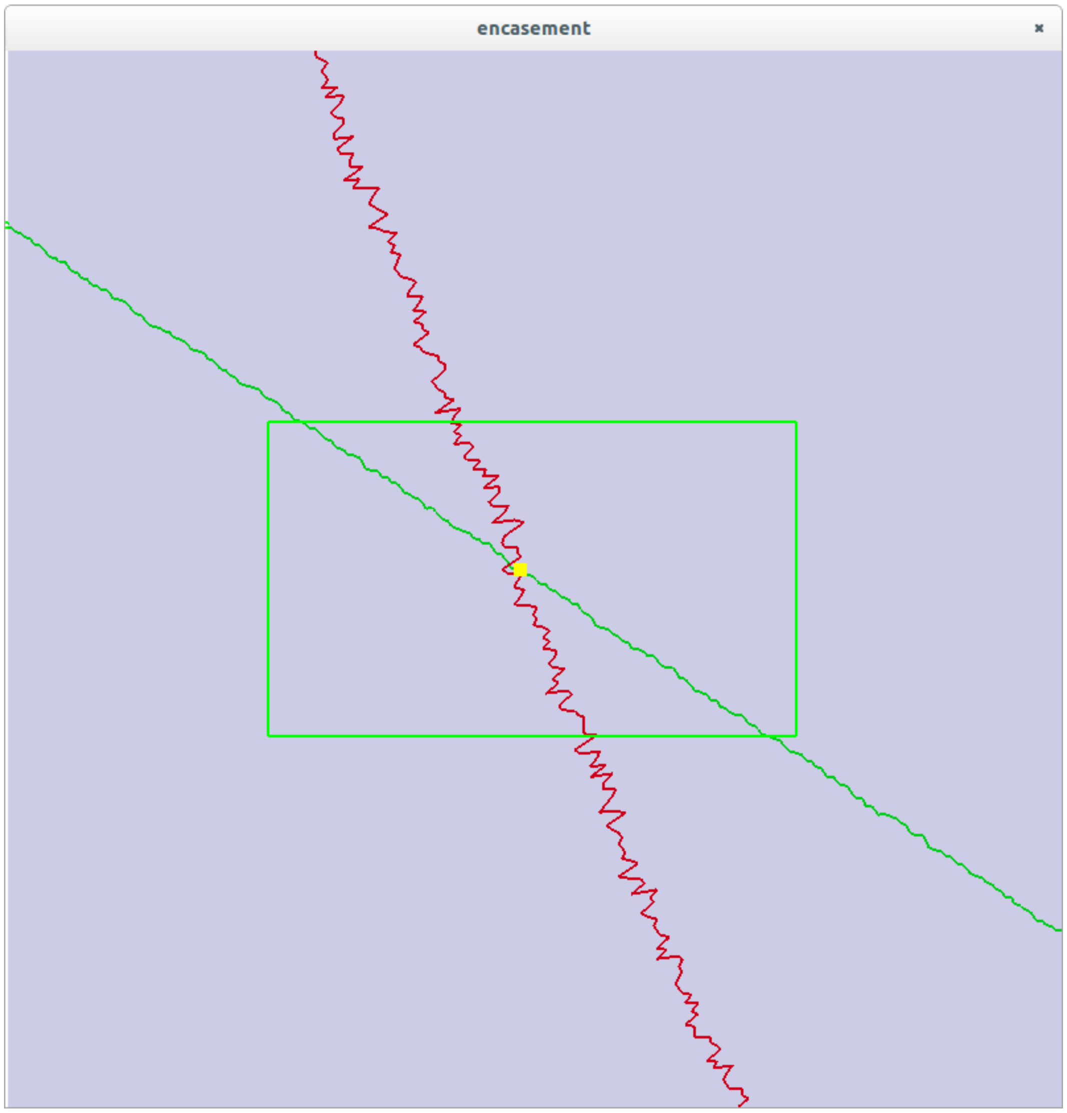

Figure 4.10: Another Root2.

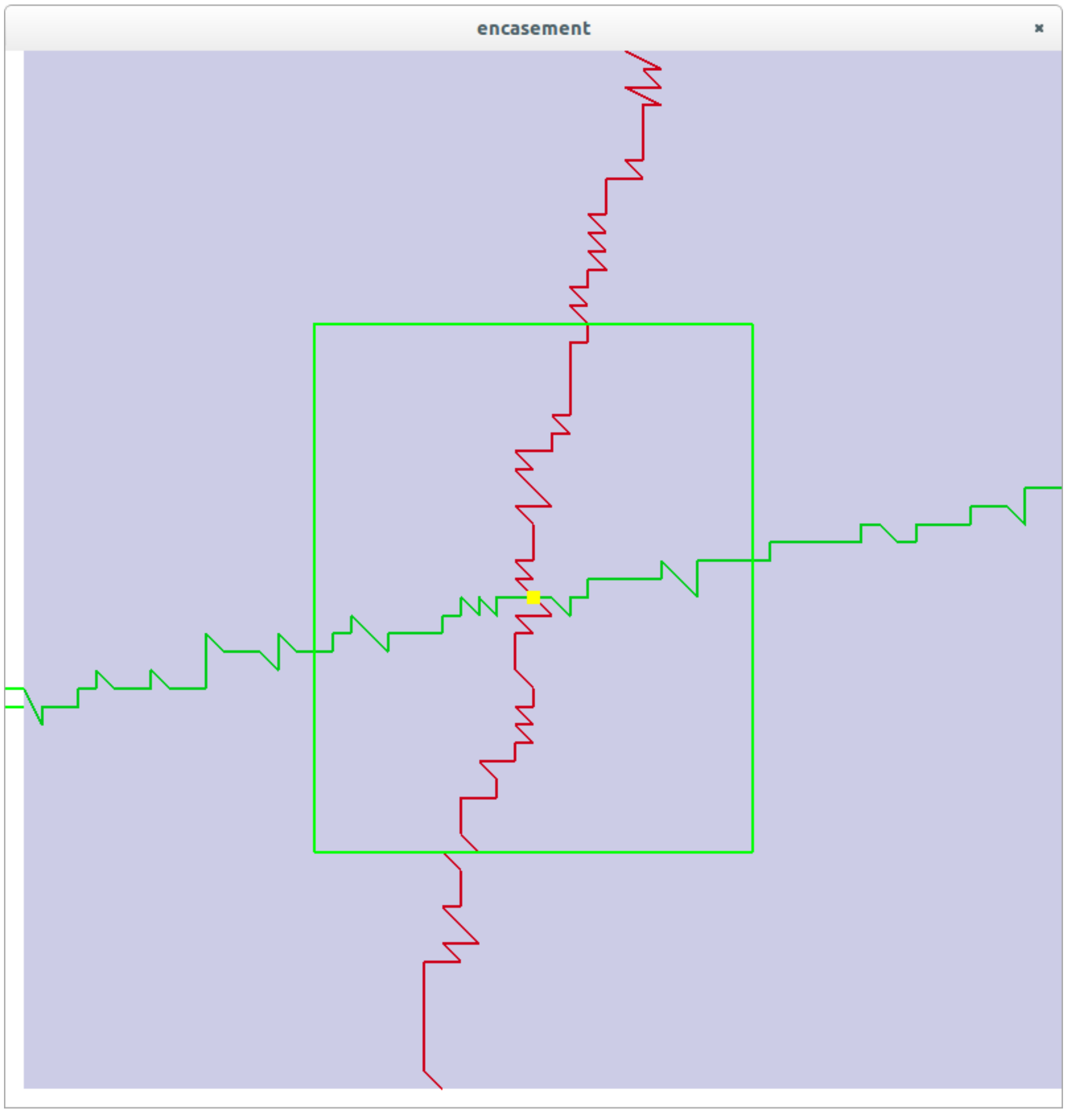

Figure 4.11: A third Root2.

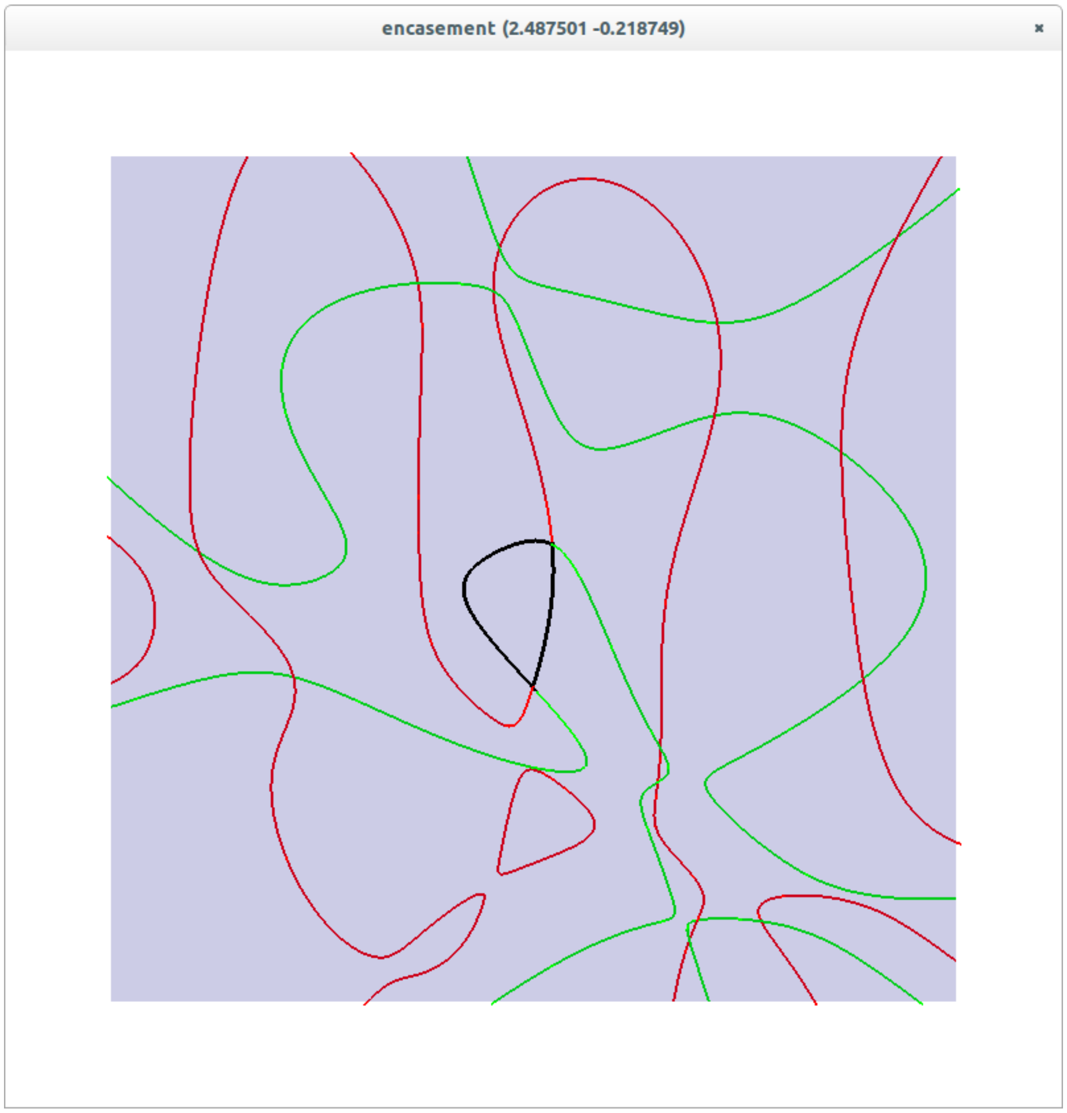

Figure 4.12: Interactive selection and tracing of arrangement faces.

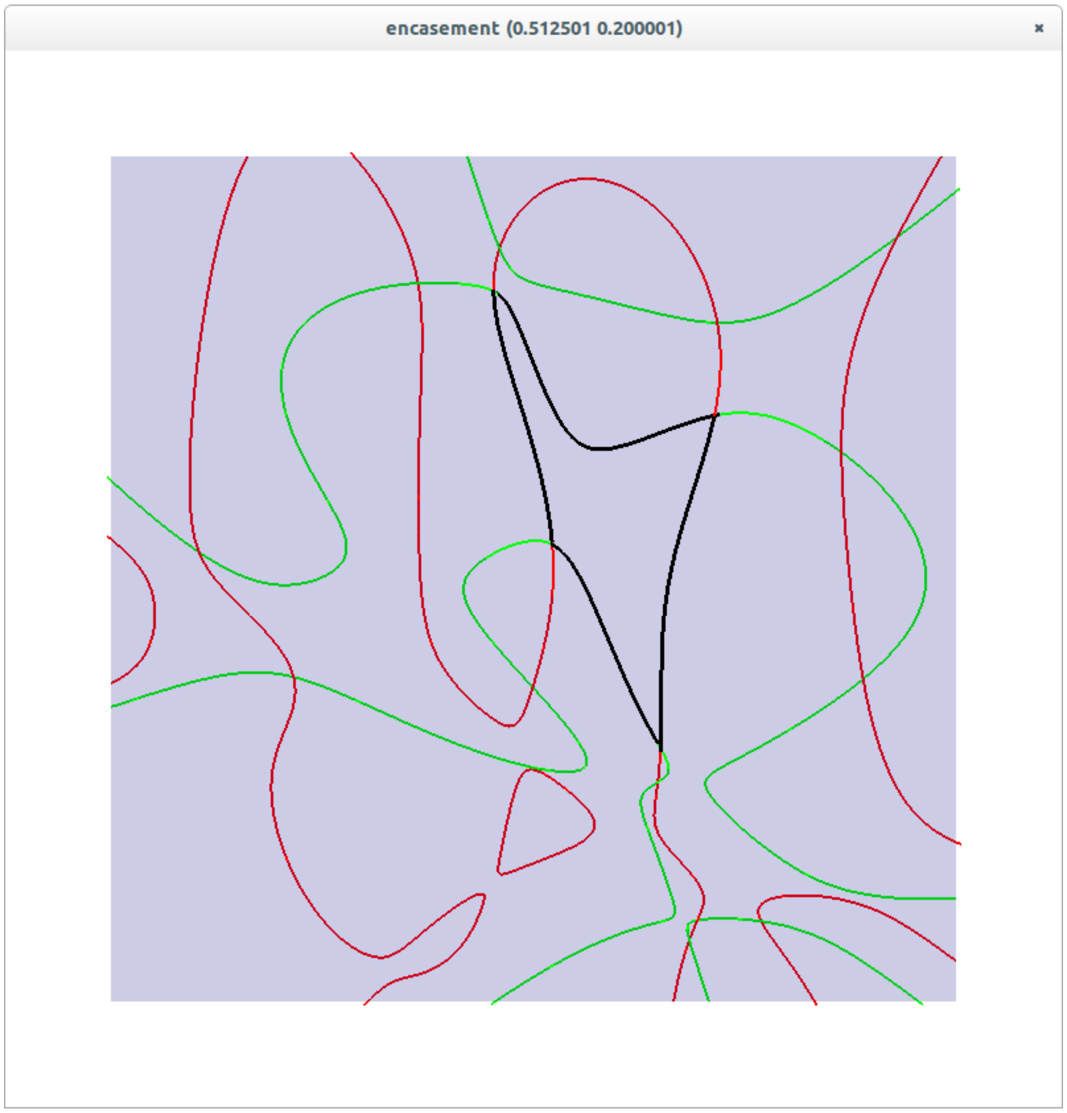

Figure 4.13: Arrangement face of f and g.

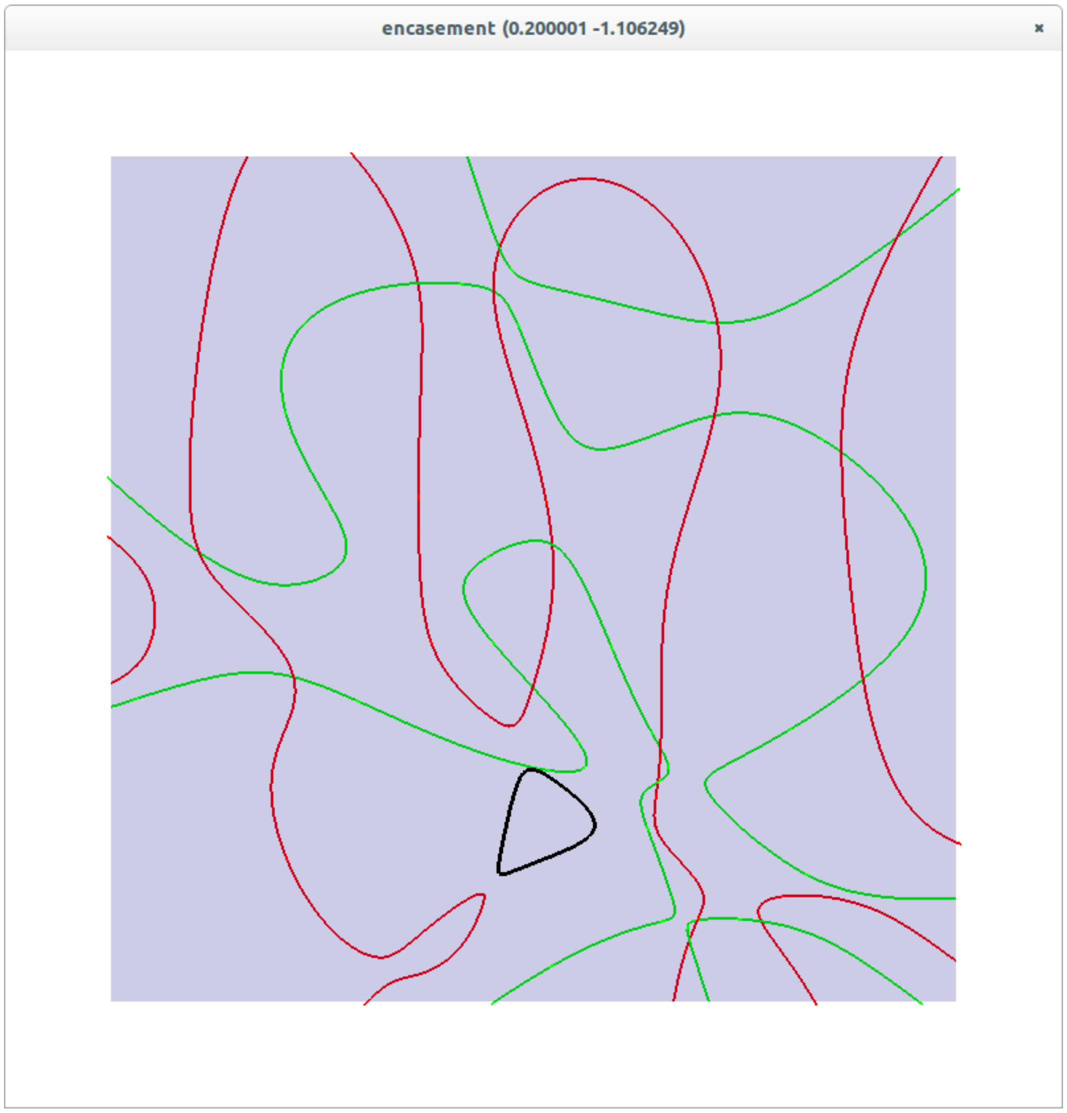

Figure 4.14: Arrangment face induced by a loop of f

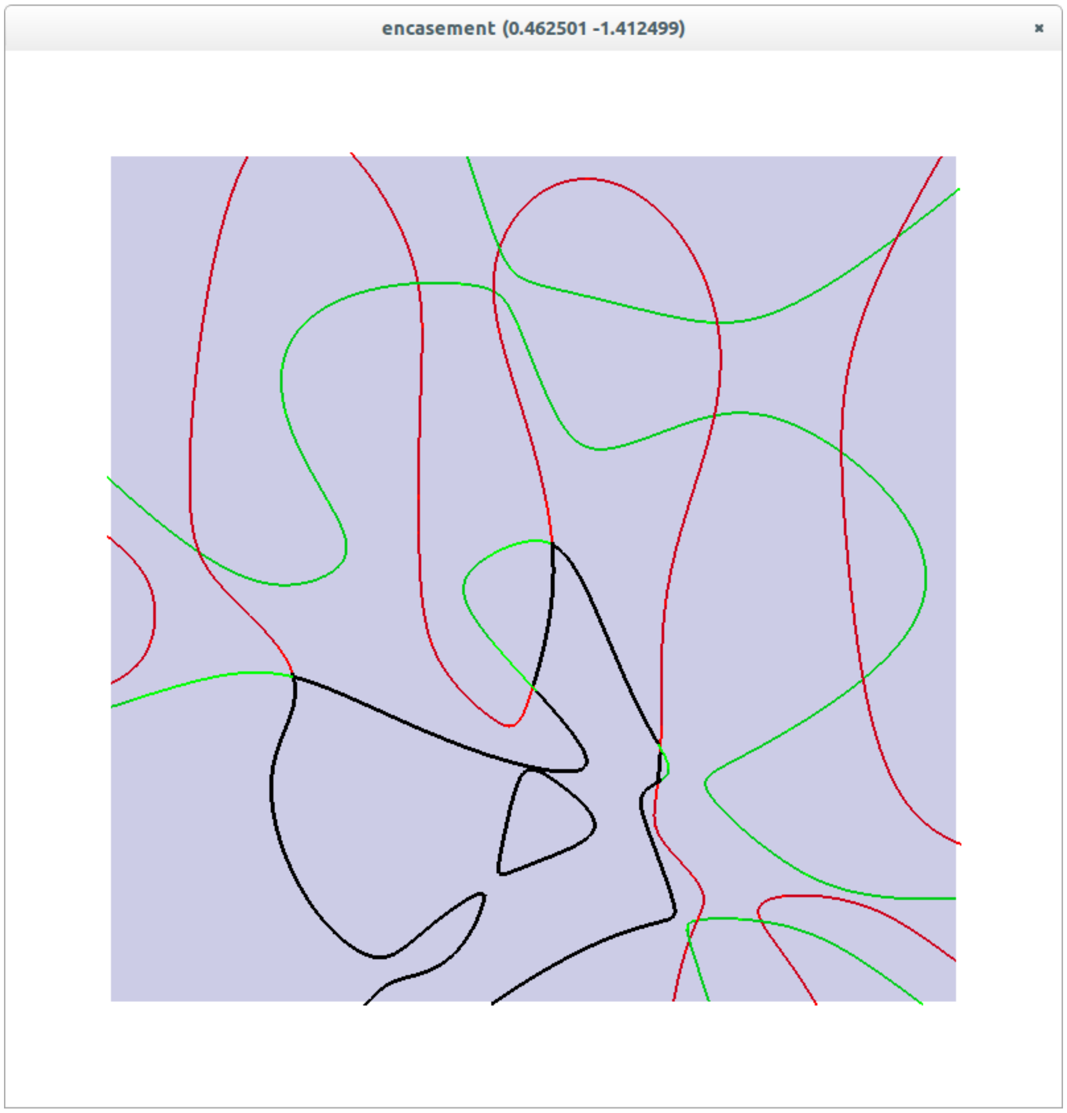

Figure 4.15: Arrangement face containing a hole whose boundary is a loop of f

into the algorithms used by CGAL and the efforts undertaken to optimize their own implementations.

The resultant of two polynomials is a polynomial expression of their coefficients, which is equal to zero if and only if the polynomials have a common root (possibly in a field extension), or, equivalently, a common factor (over their field of coefficients) [Sal85]. Any intersection of f and g occurs at a common root x of f and g considered as polynomials in y with coefficients in the ring $\mathbb{R}[x]$. These values x are the roots of the resultant $res_y(f, g)$. The subscript indicating that y is the indeterminate over which we define and compute the resultant. For each root of the resultant α one may compute the common roots of $f(\alpha, y)$ and $g(\alpha, y)$ to generate the solutions to the system: $f(x, y) = 0$ and $g(x, y) = 0$. This involves solving for the roots of univariate polynomials, something already possible and efficient in ACP. We're left to compute $res_y(f, g)$ in ACP. Let,

$$A = a_0x^d + a_1x^{d-1} + \cdots + a_d$$

$$B = b_0x^e + b_1x^{e-1} + \cdots + b_e$$

The resultant of A and B is given by the determinant of the so called Sylvester matrix of A and B:

$$\begin{vmatrix} a_0 & 0 & \cdots & 0 & b_0 & 0 & \cdots & 0 \\ a_1 & a_0 & \cdots & 0 & b_1 & b_0 & \cdots & 0 \\ a_2 & a_1 & \ddots & 0 & b_2 & b_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & a_0 & \vdots & \vdots & \ddots & b_0 \\ \vdots & \vdots & \cdots & a_1 & \vdots & \vdots & \cdots & b_1 \\ a_d & a_{d-1} & \cdots & \vdots & b_e & b_{e-1} & \cdots & \vdots \\ 0 & a_d & \ddots & \vdots & 0 & b_e & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{d-1} & \vdots & \vdots & \ddots & b_{e-1} \\ 0 & 0 & \cdots & a_d & 0 & 0 & \cdots & b_e \end{vmatrix}$$

This provides a very direct way of computing the resultant of f and g of degree d and e respectively: construct the Sylvester matrix of f and g and compute its determinant. This, however, turns out to be a very inefficient way of computing the resultant. As it turns out through the use of the Euclidean algorithm for polynomials, one may efficiently compute values of the resultant polynomial evaluated at given inputs without explicitly computing each coefficient of the resultant polynomial. This gives rise to an alternative algorithm for computation: evaluate $res_y(f, g)$ at $d \cdot e$ unique points, then interpolate the coefficients from this point-value representation of the polynomial. Evaluating the value of the resultant at a single point $x = \alpha$ takes $O(de)$ arithmetic operations on the coefficients of $f(\alpha, y)$ and $f(\alpha, y)$, which are now real valued intervals.

Standard Newton interpolation of a polynomial of degree d costs $O(d^2)$ arithmetic operations in the field of its coefficients. This makes the interpolation of $res_y(f,g)$ cost $O((de)^2)$ operations. There exist faster methods for resultants of polynomials with integer, rational, and polynomial coefficients using subresultants and even faster methods using the Chinese Remainder Theorem are known for polynomials with integer coefficients [GCL92].

4.6 Verification

The correctness of the encasement algorithm can be shown mathematically, but in order to validate the correctness of the implementation as well as strengthen the result we devised a scheme to verify the output of the encasement algorithm. Of primary concern is the correct detection of all intersections in the starting cell. A secondary but equally important concern is correct topology of the curve away from intersections, this primarily concerns the detection of loops.

To verify that all intersections have been found and that the interval representations of the enclosure of an intersection truly contain a root we used CGAL as well as our own resultant calculation. For our own resultant calculation we first find a perfect matching between the set of 2D intervals from encasement and the set of 2D intervals from resultant, where the binary relation is non-empty intersection of the two intervals. If a valid matching is found we take the intersection of each pair of roots in the matching and verify that the curves indeed alternate along the boundary of this interval. If all root pairs succeed, then the intersection set from encasement is deemed valid. For validation with CGAL we also try to construct a perfect matching between the output sets, but now the relation is the root approximated at double from CGAL is contained in a 2D interval from encasement. If the output passes both of these tests we deem the intersection set valid.

The first test validates whether we found loops of curves that intersect some branch of the other curve, but we are still not sure whether we have intersected all loops of the curve that are isolated and do not intersect the original boundary or any branch of the other curve. To validate this we perform the encasement algorithm, but replace the critical region subdivision step with a slow yet precise calculation of all of the critical point of the curves via resultants. At each critical point we generate a splitting line and split the current cell containing the critical point. This ensures that no loop of a curve exists entirely in the interior of a single encasement cell. After both encasements are computed, we count the number of connected components of each curve (branches) in the arrangement. Furthermore we also count all intersections of f with g along each branch and verify their ordering in terms of branches of g and vice versa for g. If the encasement satisfies each of these tests, we consider the entire encasement valid and its output to be correct.

4.7 Line Rounding

As stated, the primary step in the encasement algorithm is a binary subdivision of a cell, whose split is determined by a splitting line that crosses the cell. All intermediate calculations are performed for the sole task of determining a more optimal line to split a given cell, until the resulting cells satisfy the requirements of a valid encasement cell. As such, splitting lines, which end up as the boundaries of resulting cells, are first class citizens in the encasement algorithm. Splitting lines are the objects that appear most frequent in primitive evaluations, as well as the most prevalent parents of other derived objects (including other splitting lines themselves). For instance:

- Splitting line L is constructed to split between points p and q
- Point p lies on an edge of an encasement cell and is the result of substituting the line that the edge is a segment of (L_2) into a curve f and solving for roots of the resulting univariate polynomial
- The line L_2 itself is a splitting line, that resulted in the subdivision of the parent of the cell that contains it.
- Splitting line L_2 depends on splitting between a different pair of points p_2 and q_2
- etc...

It is apparent that a splitting line involved in a cell that is a result of d subdivisions may have $\Omega(2^d)$ antecedents. If a primitive fails to resolve its sign, then precision of all objects involved will need to be increased including all antecedents of any derived object in the computation. Thus, failed primitives involving splitting lines would consume the bulk of computation. If the derivation tree of a splitting line gets prohibitively large, it makes the evaluation of any failed primitive inefficient. For that reason, it was necessary to develop for any splitting line L and a set of constraints $\{c_0, c_1, ..., c_n\}$ that L satisfies, a way to construct an alternative splitting line, L^*, which satisfies all constraints, but whose construction depends on randomized and generic input with empty derivation trees. To do so, we implement an approximation function approx(L, a, b). The constraints of this function are that line $L^* = (p, v)$ separates points a and b, and that L^* is close to the exact splitting line L. That is to say: $\text{sign}((a - p) \cdot v)) \neq \text{sign}((b - p) \cdot v)$. The function returns a randomized splitting line that also separates points a and b.

We achieve the alternative splitting line, L^* as follows: First try approximating each coordinate of p and v by the midpoint of its interval, e.g. $(p.x_{lb} + p.x_{ub})/2$, and then perturbing in the same manner as we perturb inputs: add $\text{rand}(-10^{-8}, 10^{-8})(1 + |x|)$ to x. If this line fails to separate a and b, do the following:

Recompute the intervals for the coordinates until they are small enough so that $p + vt$ does separate a and b in interval arithmetic. This may require higher precision arithmetic. Then replace each coordinate of p and v by a random double-float element of its interval. This is complicated by the fact that the interval may be very small and possibly have no double-floats in it at all. Select a random element as follows:

- Let S be a list of double-float numbers, initially empty. Let $[l, u]$ be the interval.
- If $|u - l| < (10^{-8})\ |l|$, then the interval contains fewer than 10^8 double-floats, which we consider too few for a random choice.
- In that case, let d be the double-float resulting from rounding l towards zero. Add d to S and subtract d from l and u.
- Repeat until $|u - l| \geq (10^{-8})\ |l|$. Select a random double-float r in the interval $[l, u]$ and add r to S. The approximation is defined as the sum of the elements of S.

We use the sum of doubles representation presented by Shewchuk [She97]. Using this technique, each coordinate of p and v are expressed as a set S of double-floats whose largest element is an accurate approximation and whose sum, carried out in extended precision, is the exact value.

CHAPTER 5

Experiments and Results

The encasement algorithm isolates the connected components of a semi-algebraic curve inside of a convex region, but its primary focus is finding all intersections between a pair of semi-algebraic curves (f and g) in that convex region. The problem of finding all intersections of a pair of bivariate polynomial equations in a 2D region can be reduced to the problem of finding the roots of univariate polynomials on an interval. This is already shown with various methods using resultants, but encasement provides an alternative reduction and method of computation. Therefore to show the merits of encasement, we test its performance computing the intersections on a gamut of different curves and compare it to the performance of the leading computational geometry software, CGAL [The17]. Specifically, we use the Arrangement_on_surface_2 [WBF^{+}17] package which provides functionality for arrangements of planar algebraic curves. Most of the algorithms computing the algebraic objects and univariate roots required by the techniques CGAL uses are contained in the Curved_kernel_via_analysis_2 [BHK$^+$17] package, which is a model for ArrangementTraits_2 concept. The methods of this model that analyze the geometric properties of single curves and pairs of curves are contained in the Curve_analysis_2 and Curve_pair_analysis_2 classes respectively. This package makes use of state of the art algebraic methods for solving bivariate systems, cited in the code as [EK08] and [EKK$^+$05] (which were discussed in the related work section).

One caveat to make mention of is that CGAL and ACP operate under different paradigms to provide robustness. ACP's paradigm is to handle only generic input, for us curves in general position, and provide exact solutions for a small perturbation of the input with backwards error gaurantees. CGAL on the other hand operates under the Exact Computation Paradigm of Yap [Yap00]. CGAL tracks error bounds and uses extended precision only when it is truly needed, much like ACP, but with different representations of the underlying number types. So not only are we comparing different algorithms for the solutions of bivariate equations, but at the same time comparing the two paradigms. Another thing to mention is that the coefficients of the polynomials for CGAL's 2D Arrangement package are specified to be integers. Because we generate the random input numerically in double, we must transform the polynomials to equivalent polynomials with integer coefficients. This is done in such a manner that the number of digits of precision in the coefficients (namely 16 digits) is preserved.

5.1 Experimental Setup

We generate random curves of varying degree in general position that intersect at common points. To generate a curve f of total degree n (having $m = (n(n+1)/2)$ coefficients), generate $m-1$ random points in the region of interest. We want these $m-1$ points to lie on the curve (i.e. $f(p_i) = 0$). Thus for $p_i = (x_i, y_i)$ we can infer

$$c_m x_i^n + c_{m-1} x_i^{n-1} y_i + ... + c_2 x_i + c_1 y_i + c_0 = 0$$

We can assume WLOG that $c_0 = 1$. From this we can generate a non-homogeneous linear system:

$$\begin{bmatrix} x_1^n & x_1^{n-1}y_1 & \dots & x_1 & y_1 \\ x_2^n & x_2^{n-1}y_2 & \dots & x_2 & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m-1}^n & x_{m-1}^{n-1}y_{m-1} & \dots & x_{m-1} & y_{m-1} \end{bmatrix} \begin{bmatrix} c_m \\ c_{m-1} \\ \vdots \\ c_1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ \vdots \\ -1 \end{bmatrix}$$

We then solve this system using efficient subroutines from the BLAS/LAPACK software package [ABB+99]. Because each curve is generated from its own unique set of interpolating points, the probability of these curves being degenerate is very low. Thus we assume each intersection point is transverse and involves only two curves.

We generate curves ranging from degree 2 to degree 20. As a baseline for the performance of the various algorithms with respect to the degree of the input, we test each algorithm against pairs of curves of the same degree. We then tested each algorithm's ability to process the arrangement of a batch set of curves ranging from groups of 3 to 20 curves. All curves within a batch have the same total degree.

To exemplify the ability of encasement to perform when smaller regions of interest are selected, we isolated smaller convex regions surrounding a known intersection of a pair of curves. We then ran the encasement algorithm again, changing the starting region of interest to $[r_x - d, r_x + d] \times [r_y - d, r_y + d]$, for various values of d.

We compare the timing results of our encasement algorithm against the leading computational geometry software, CGAL [The17] All experiments were performed on a server with an Intel(R) Xeon(R) CPU E5-2650 v3 @ 2.30GHz (10 cores) with 64 GB of RAM.

Because the focus of the encasement software as it stands is on computing all pairwise intersections of each of the curves, we would like to focus our timing comparisons on pairs of curves. The encasement algorithm handles only two curves at a time computing their indices, and only after that would provide relevant arrangement information. Thus we want to compare this performance against CGAL computing the arrangements of only a pair of curves at a time. One might argue that this is a handicap for CGAL, and that neglecting its batch processing capabilities would lead to suffering performance, but we intend to show that the extra complexity of computing the arrangement of n curves supplants the complexity of doing $O(n^2)$ arrangements of pairs of curves. To show this we have given the comparison of the performance of CGAL on various numbers of curves. The monotonization process of either method remains the same (Fig. 5.1). Both must monotonize n curves individually resulting in a curve reflecting the $O(n)$ nature of the monotonization for both.Clearly CGAL receives no benefit from computing all of the intersections of the curves from the batch arrangement (Fig. 5.2). If the intersections of the curves (vertices of the arrangements in CGALs case) are the main focus, then using CGAL to compute $O(n^2)$ individual arrangements of the pairs of curves yeilds better results. Therefore we believe it is justified to compare encasement against CGAL Arrangements by just looking at running time on a single pair of curves. The asymptotic behavior of both on n curves can be reasonably predicted.

Both the encasement and Resultant Based CGAL Arrangement algorithms can be broken down in to 3 stages: sub- algorithms that are analogous. The encasement algorithm must first isolate all features of the curves that are unknown from the bounding box stage. This corresponds to loops of the curves. Likewise CGAL must find x-monotone segments of each curve (which includes the price of finding loops) in order for it to proceed. Second each algorithm must isolate the common roots of the pair of curves and robustly ensure their existence and separation. Thirdly we require that each algorithm provide representations of the intersections at at least double precision. CGAL provides a double representation of the algebraic numbers representing their coordinates if queried and encasement can provide approximations of the coordinates of the intersections to arbitrary degree.

5.2 Results Large Domain

The first result compares the running time of the first stage of each algorithm (Fig. 5.3). X-Monotonization of the curves for CGAL and splitting of loops for encasement. The two methods are comparable for low degrees but show a significant divergence around degree 12.

The encasement loop splitting technique computes a lower degree lower bound Taylor approximation of the curve on each sub-cell until the separation is achieved. The combination of adapting the degree of this approximation along with using recursive encasements at optimal depths keeps the running time modest with respect to the degree.

We show the ratio $T_{\text{CGAL}}(d)/T_{\text{encasement}}(d)$, of the first stage, where T denotes the average running time of the algorithm on a random curve of the specified degree (Fig. 5.4). The data shows a definite increasing trend, indicating that the monotonization process for CGAL is asymptotically more complex than loop splitting for encasement.

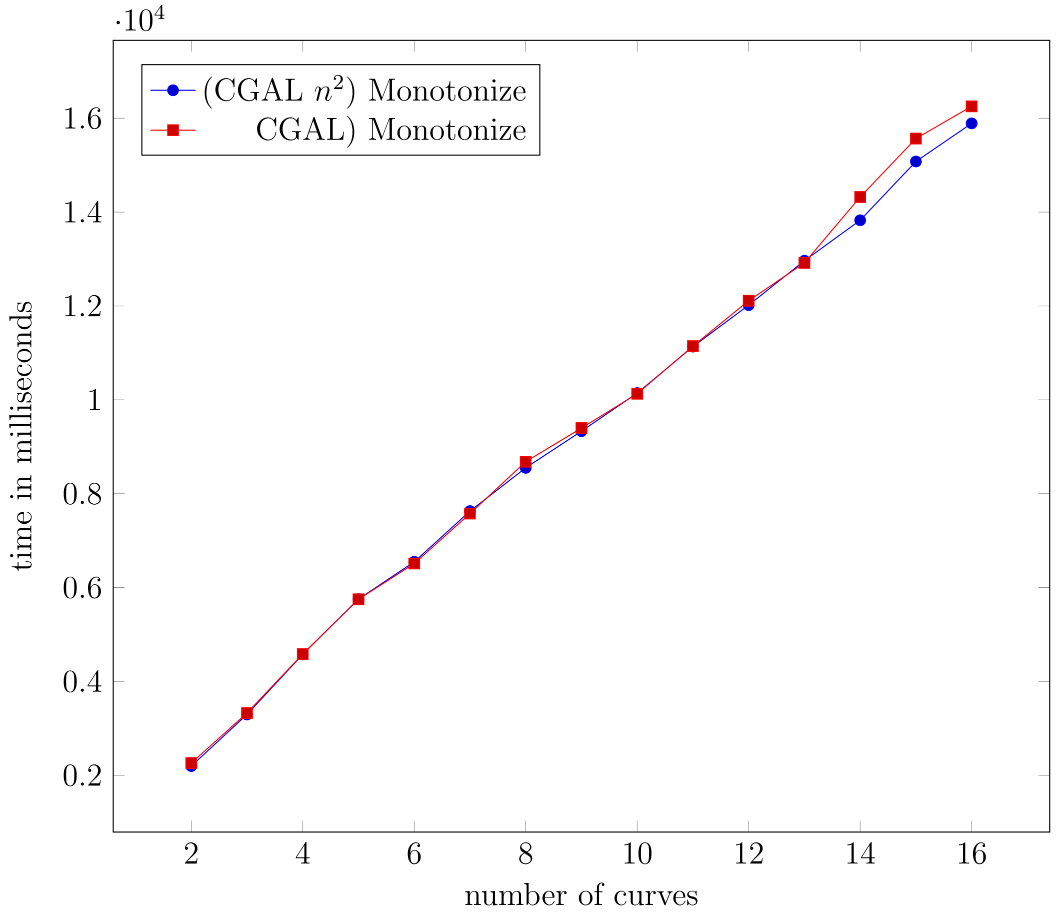

Figure 5.1: Analogous first steps in each of the n^2 CGAL and batch CGAL algorithms. Both demonstrate linear behavior asymptotically in the number of curves.

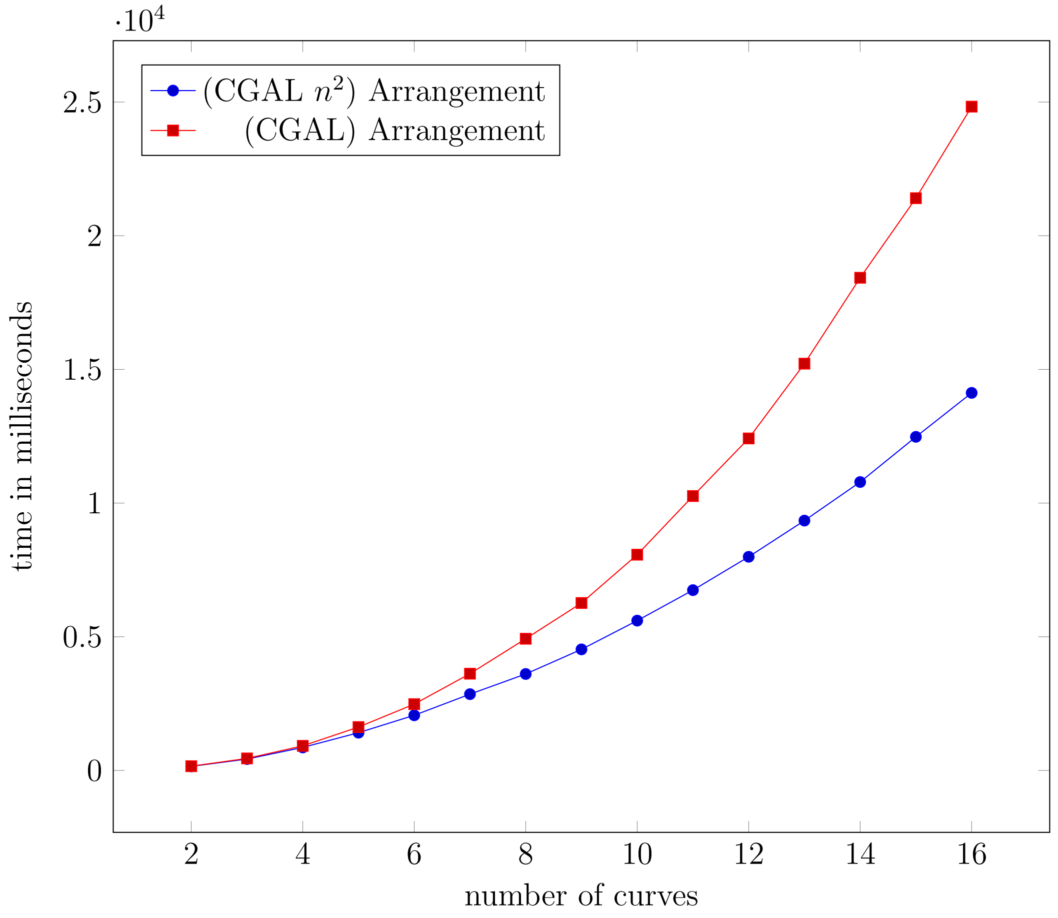

Figure 5.2: The cost of arrangement of the n^2 CGAL and batch mode CGAL. Both demonstrate quadratic behavior asymptotically in the number of curves, with the n^2 showing a smaller constant factor.

The second result demonstrates the comparative running time between the second stage of each algorithm (Fig. 5.5). Face splitting to isolate intersections for encasement and arrangement insertion for CGAL. Although encasement seems to have the disadvantage here, the scale of the time axis is a factor of 10 smaller than the other two figures. This second stage of each algorithm seems to affect their respective running times the least. The main cost encasement endures during this phase of the algorithm is substituting univariate line equations into bivariate polynomials and the isolation of the subsequent single univariate polynomial's roots. Substitution is either done in a direct manner or by interpolation with Newton Polynomials. Both of these depend on the number of terms in the bivariate polynomial, which is quadratic in the total degree. Thus the non linear nature of the cost. At larger degrees, ACP may not be able to isolate the roots of polynomial without pushing to higher precision arithmetic.

We show the ratio $T_{\mathrm{CGAL}}(d)/T_{\mathrm{encasement}}(d)$ of the second stage, where T denotes the average running time of the algorithm on a random curve of the specified degree (Fig. 5.6). The ratio quickly decreases and approaches a constant rate of around 0.5. This indicates that the two algorithms only differ by a constant factor when runtime is analyzed with respect to the degree of the input.

The third result demonstrates the running time of computing double precision approximation of the coordinates of the intersections of the two curves

(Fig. 5.7). For encasement this cost is relatively low. The intersection isolation portion already requires computing the coordinates of the roots in double precision to the best ability of Interval Newton's method and subdivision. Thus, all that is required for coordinates whose intervals are not yet already consecutive double precision floats is to perform Interval Newton's method and subdivision (a.k.a. Root2 polishing) in 212-bit precision. The quadratic convergence of Newton's method is key in this procedure's efficiency. On the other hand, this task seems to be difficult for CGAL. Even the documentation for the arrangements of 2D curves package mentions that requesting double approximations of coordinates of points can be a costly procedure [WBF$^+$17]. This is clearly exemplified by the steep increase in CGAL's cost, again around degree 11-12. Having double precision approximations of these coordinates is essential for use of the results as inputs in additional CAD and geometry software, so we believe the addition this runtime to the total timing results to be fair.

We show the ratio $T_{\text{CGAL}}(d)/T_{\text{encasement}}(d)$ of the third stage, where T denotes the average running time of the algorithm on a random curve of the specified degree (Fig. 5.8). The ratio is increasing as the ratio for the first stage. Yet, the ratio is a order of magnitude larger than the ratio for the first stage. It trends in a close to linear fashion, indicating that CGAL's approximation for intersection coordinates may be a factor of d larger than encasement's intersection approximation.

The final result compares the total running time of the three stages of each algorithm added together (Fig. 5.9). Clearly the pitfalls of the encasement algorithm are undoubtedly overshadowed by the pitfalls of CGAL's algorithm. Even though CGAL outperforms encasement in the second stage, the scale of this improvement is much smaller than the out-performance of encasement in the other stages.

We show the complete ratio $T_{\text{CGAL}}(d)/T_{\text{encasement}}(d)$ of the total running time of all three stages, comprising the totality of the algorithms (Fig. 5.8). The ratio is dominated by the contributions from the first and third stages of each algorithm, where most of the expense of each is contained. Thus it is steadily increasing at a rate that seems to be well approximated linearly, at this range of degrees. Therefore we can confidently state that the asymptotic complexity of encasement improves on the complexity of CGAL's arrangements by a factor of at least d.

5.3 Results Smaller Domains

The second set of results exemplifies encasement's stellar performance on progressively smaller input domains. The interest in computing relevant topological information on restricted input domains is important. Many applications require locally refined information and have no need for a global arrangement.

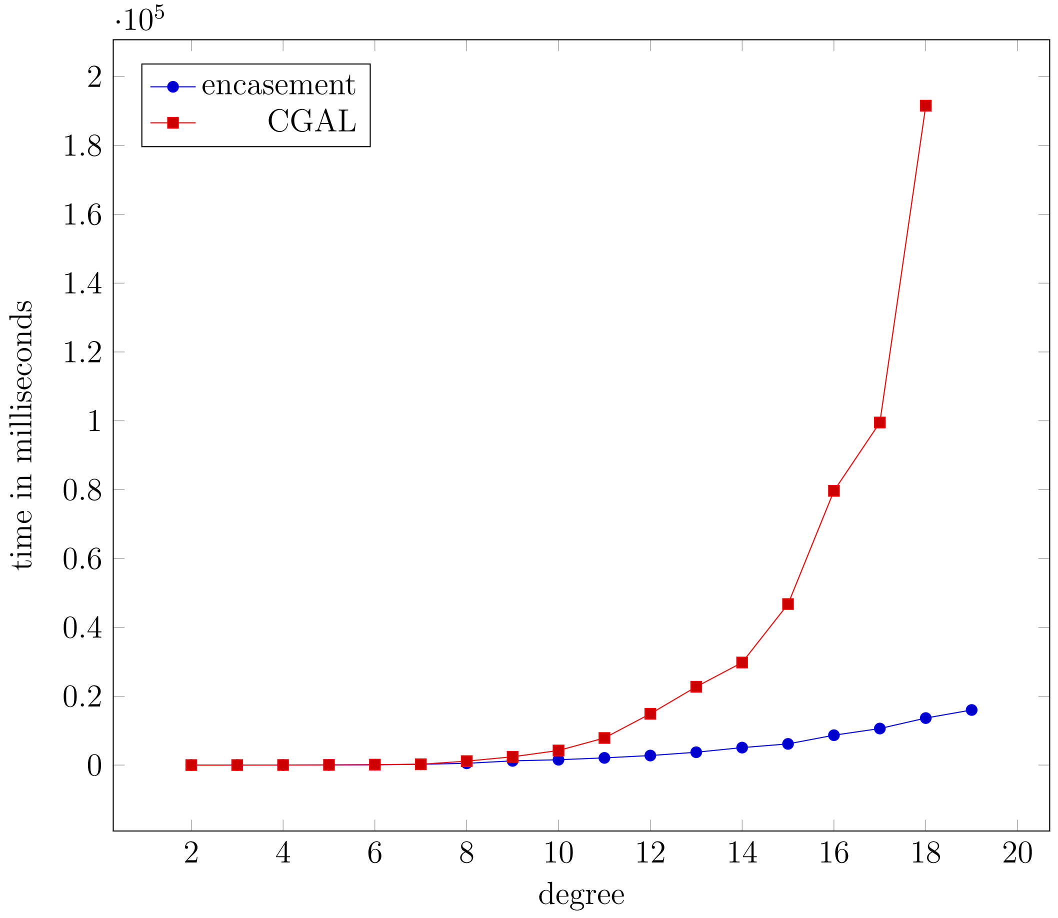

Figure 5.3: (Individual Curve Isolation) Total running time of encasement vs CGAL on pairs of curves of varying degrees.

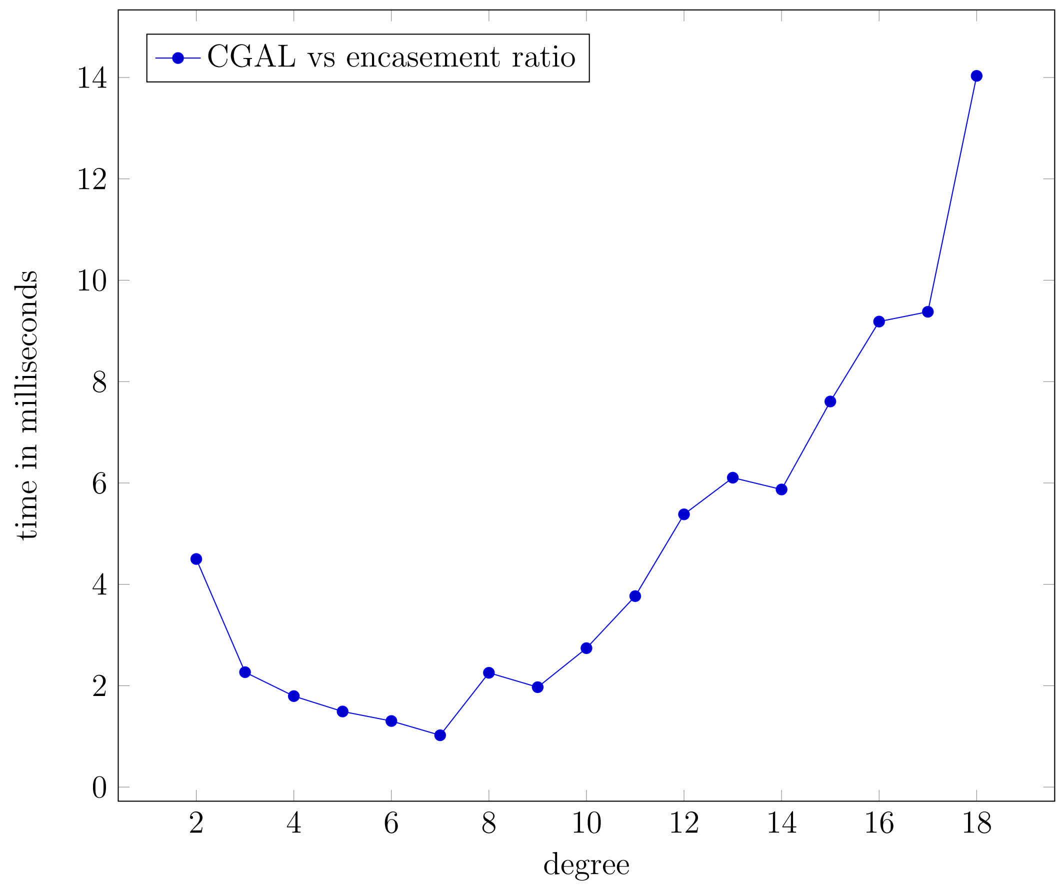

Figure 5.4: (Individual Curve Isolation) Ratio of running time CGAL over encasement on pairs of curves of varying degrees.

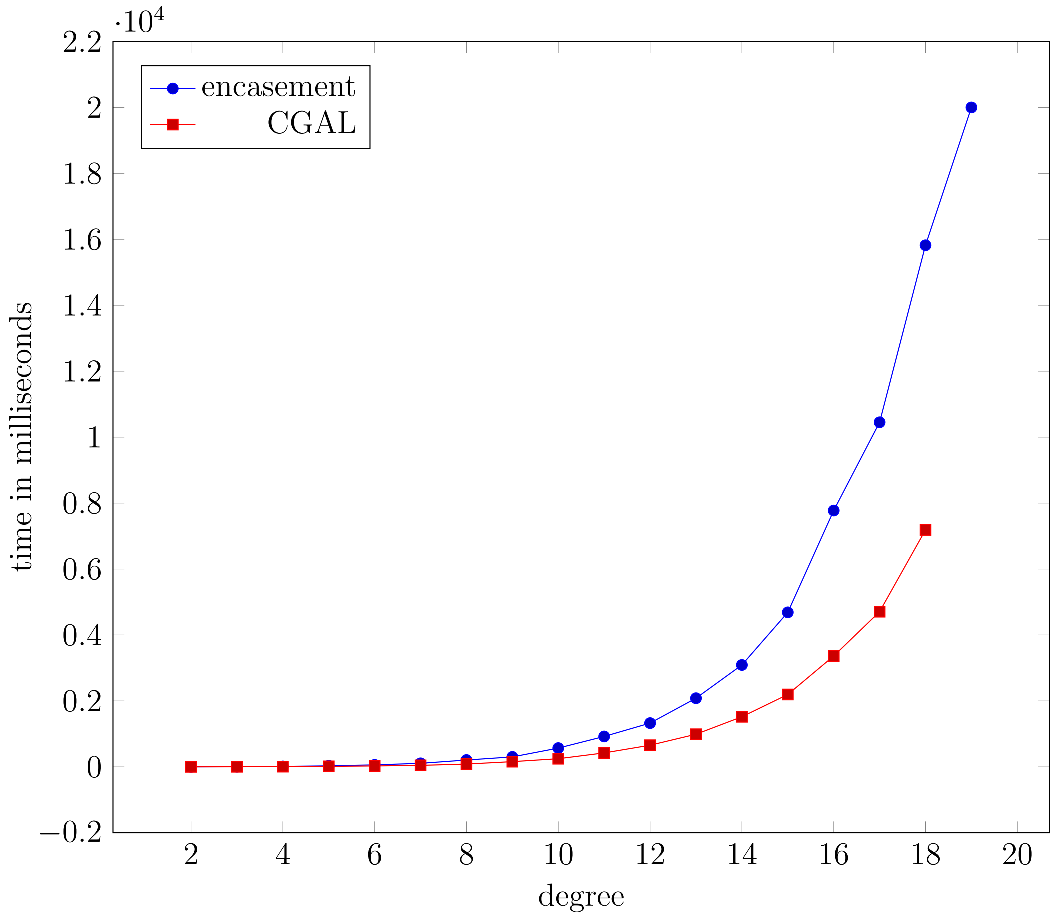

Figure 5.5: (Root Isolation) Total running time of encasement vs CGAL on pairs of curves of varying degrees.

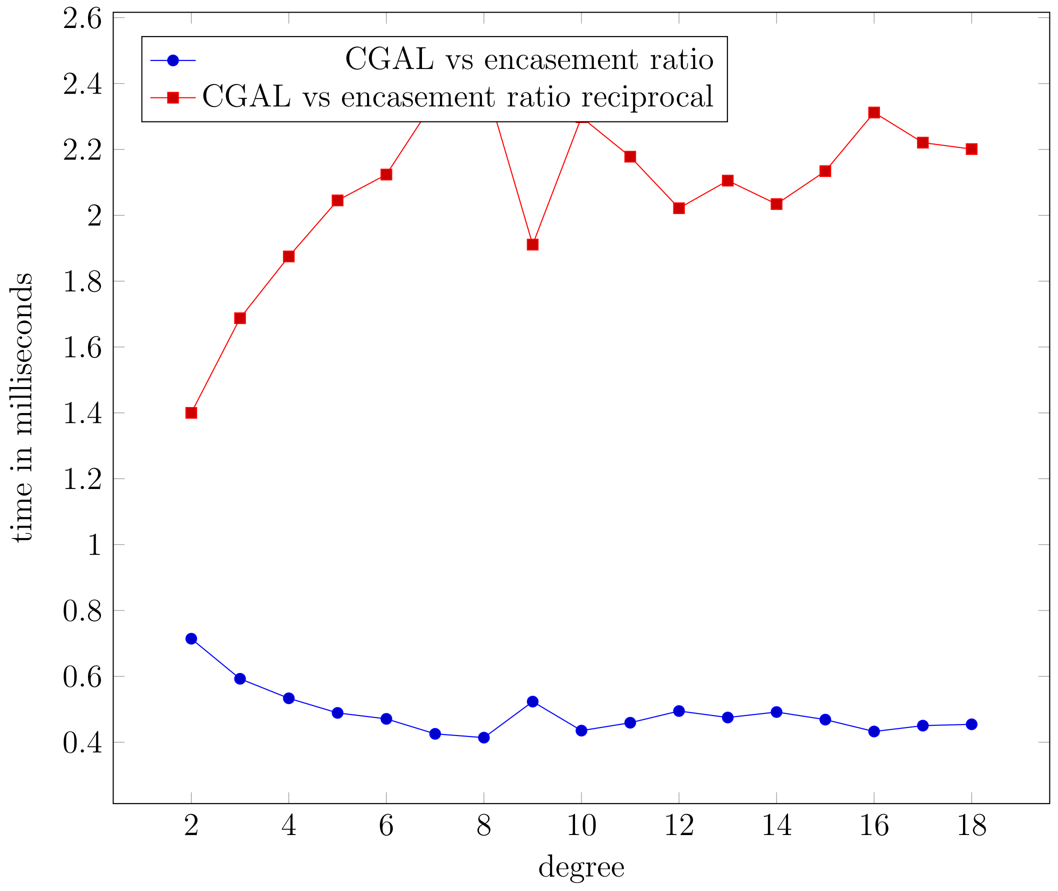

Figure 5.6: (Root Isolation) Ratio of running time CGAL over encasement on pairs of curves of varying degrees.

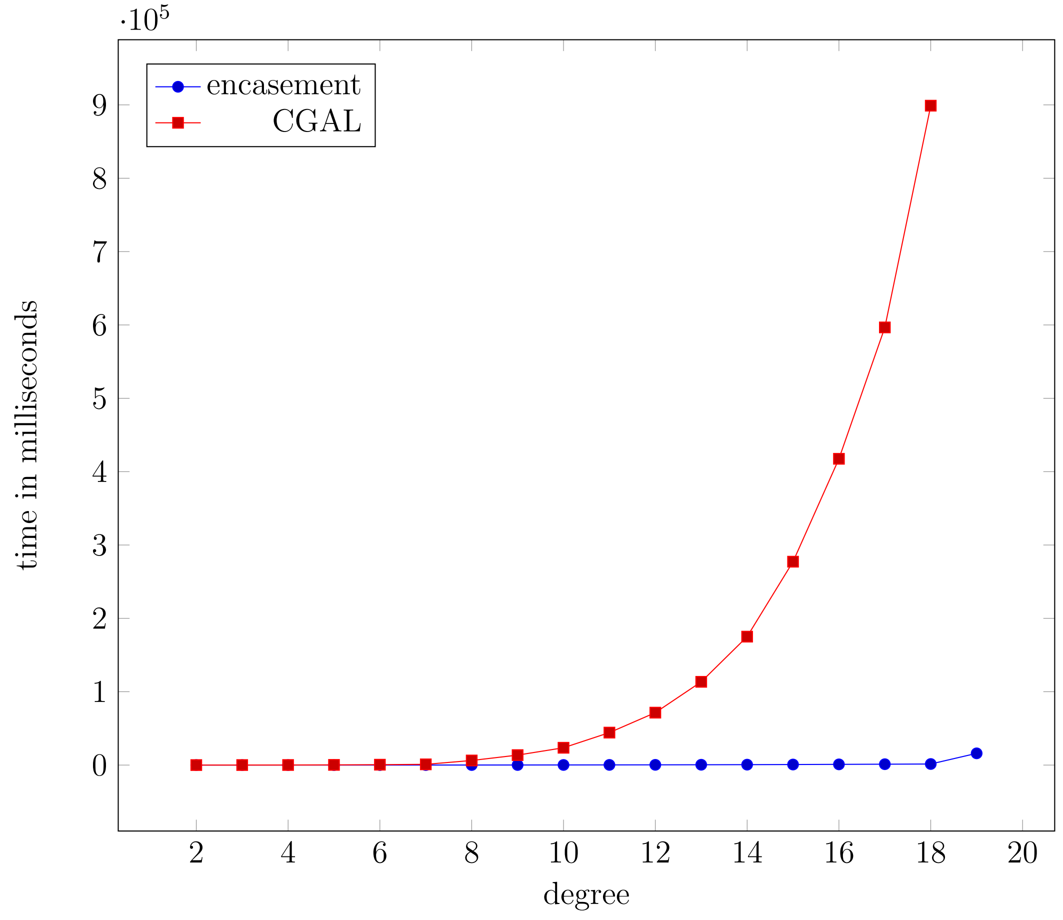

Figure 5.7: (Root Approx Double) Total running time of encasement vs CGAL on pairs of curves of varying degrees.

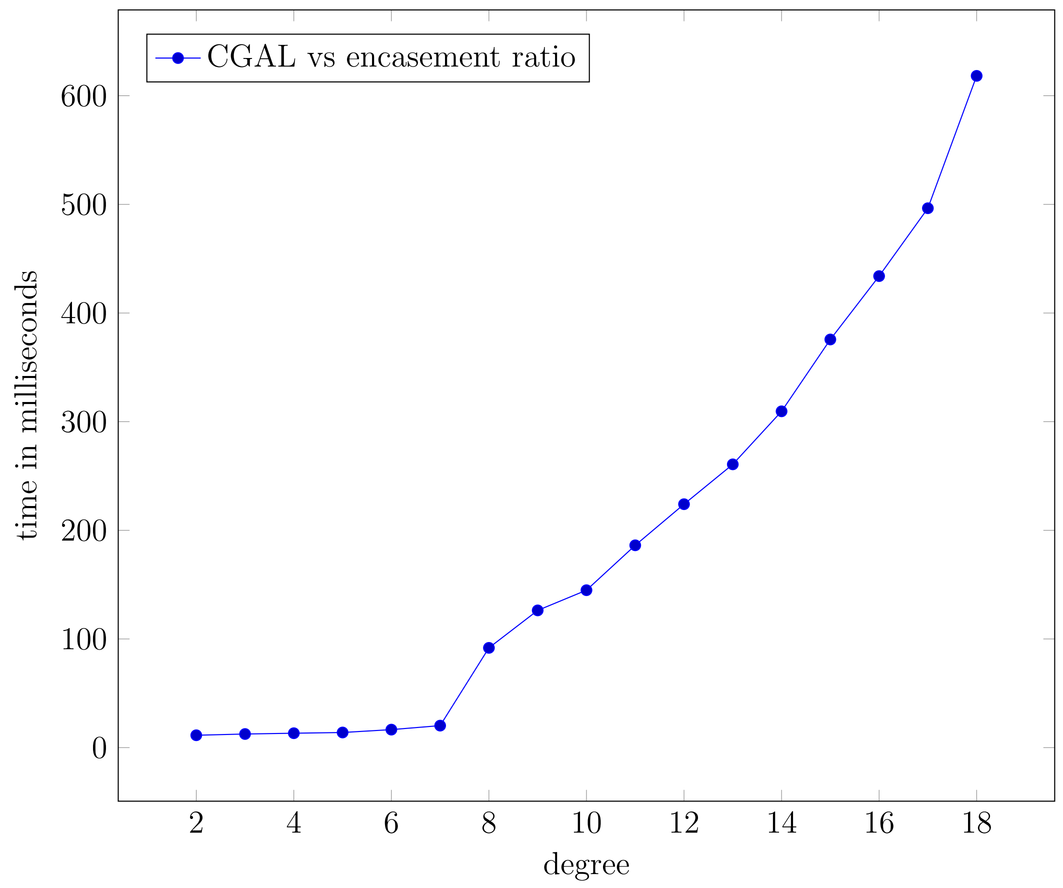

Figure 5.8: (Root Approx Double) Ratio of running time CGAL over encasement on pairs of curves of varying degrees.

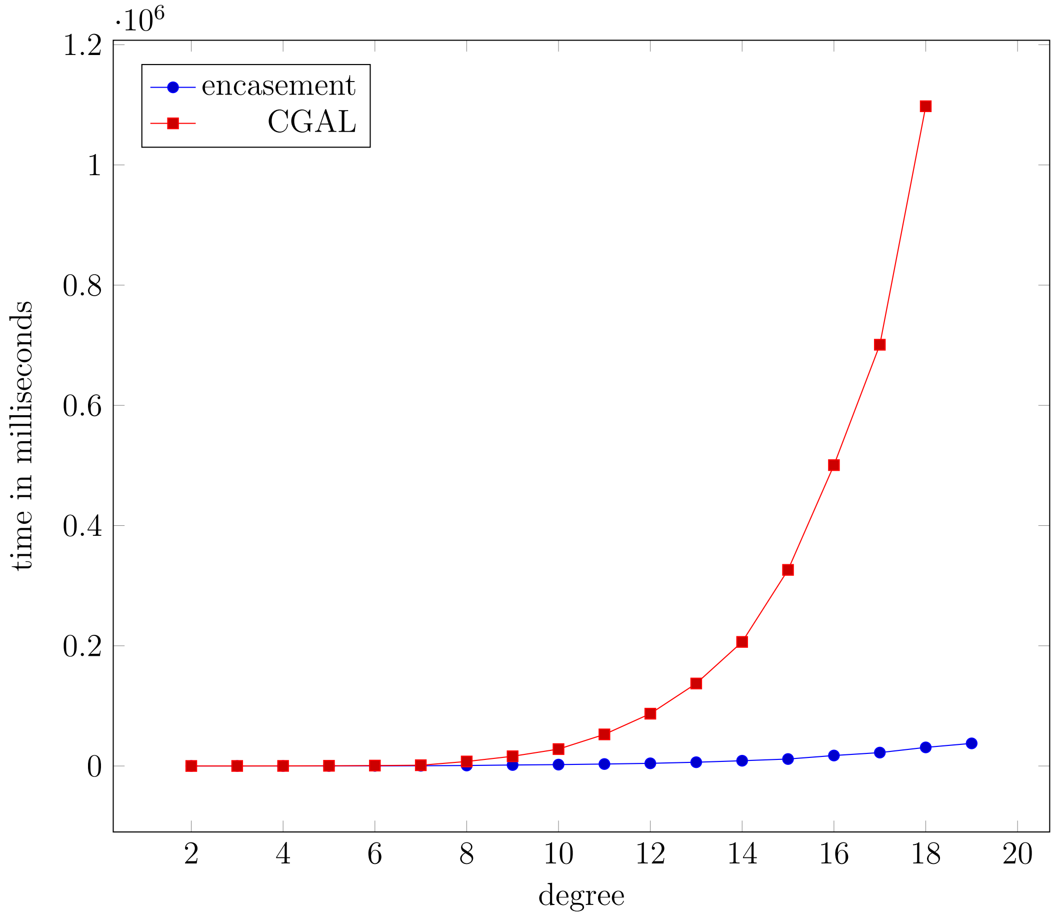

Figure 5.9: (Total Running Time) Sum of all three stages of each encasement and CGAL on pairs of curves of varying degrees.

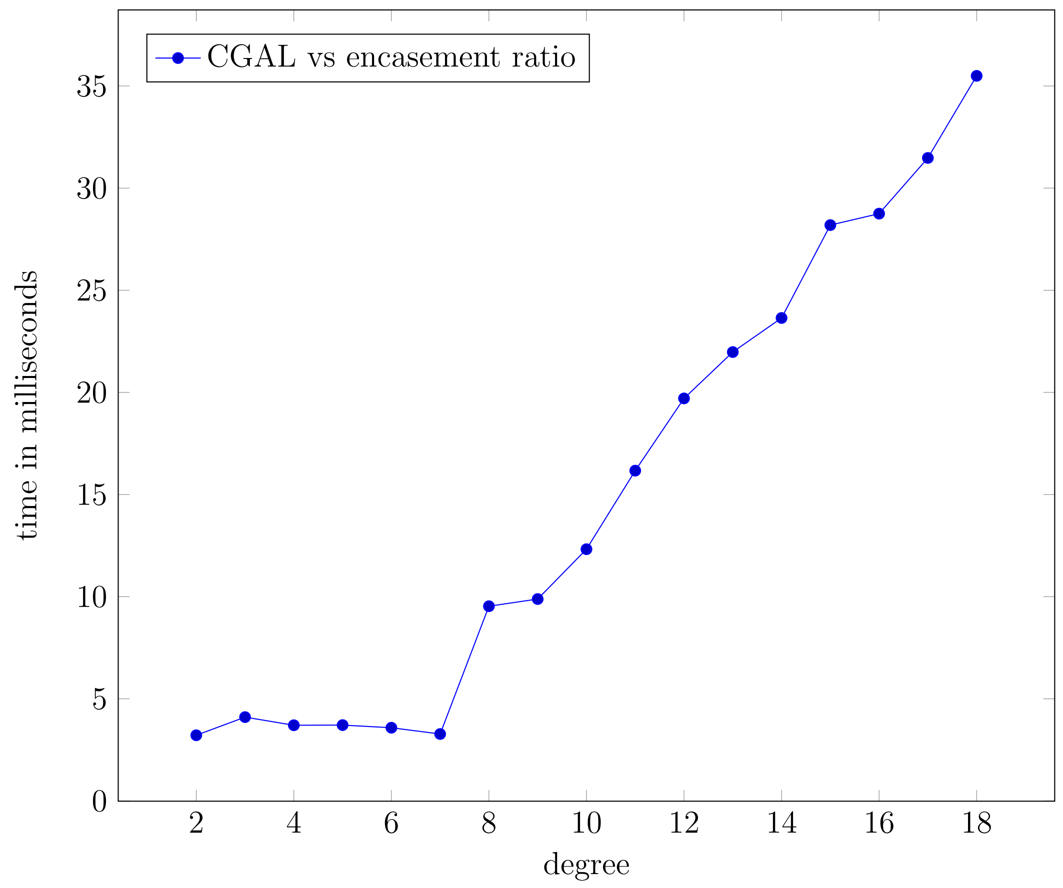

Figure 5.10: (Total Running Time) Ratio of running time CGAL over encasement on pairs of curves of varying degrees.

We show a comparison between the total running time of encasement and CGAL on progressively smaller and smaller input domains (Fig. 5.12 and Fig. 5.11). Each curve in the figure represents the running time of an algorithm with a fixed-size input box on curves of varying degrees. The input domains were chosen to be non-trivial and include at least a single intersection of the two input curves. Each input domain is square and the number next to the title of each curve in the legend represents the side length.

CGAL does not have a natural way to restrict the domain of the input, so what is required is to compute sub-segments of the X-monotone segments of the original curve who intersect the left and right x values of the input domain. These become the inputs to the arrangement, and all other segments that do not intersect that interval are discarded. Clearly from the figure, CGAL sees no benefit from decreasing the size of the input bounding box. Their techniques are mostly agnostic from the region of interest specified. It must monotonize the input and compute a resultant regardless. It may benefit slightly when isolating roots of its resultant polynomial, but those gains are insignificant in comparison to the cost of everything else.

encasement on the other hand is, by design, handled to reap the benefits of smaller input domains. Each factor of two smaller the input domain removes a recursive depth from the loop cutting algorithm. For generic inputs the complexity and distribution of splitting lines is proportional to the complexity of the arrangement and distribution of its features. Therefore whatever factor of the complexity of the arrangement is removed from the input domain will be the same factor of improvement in the face splitting for intersection isolation. The same applies to refining the coordinates of the intersections to consecutive double floats. Clearly each factor smaller in the input box pushes the running time down for all degrees (Fig. 5.12).

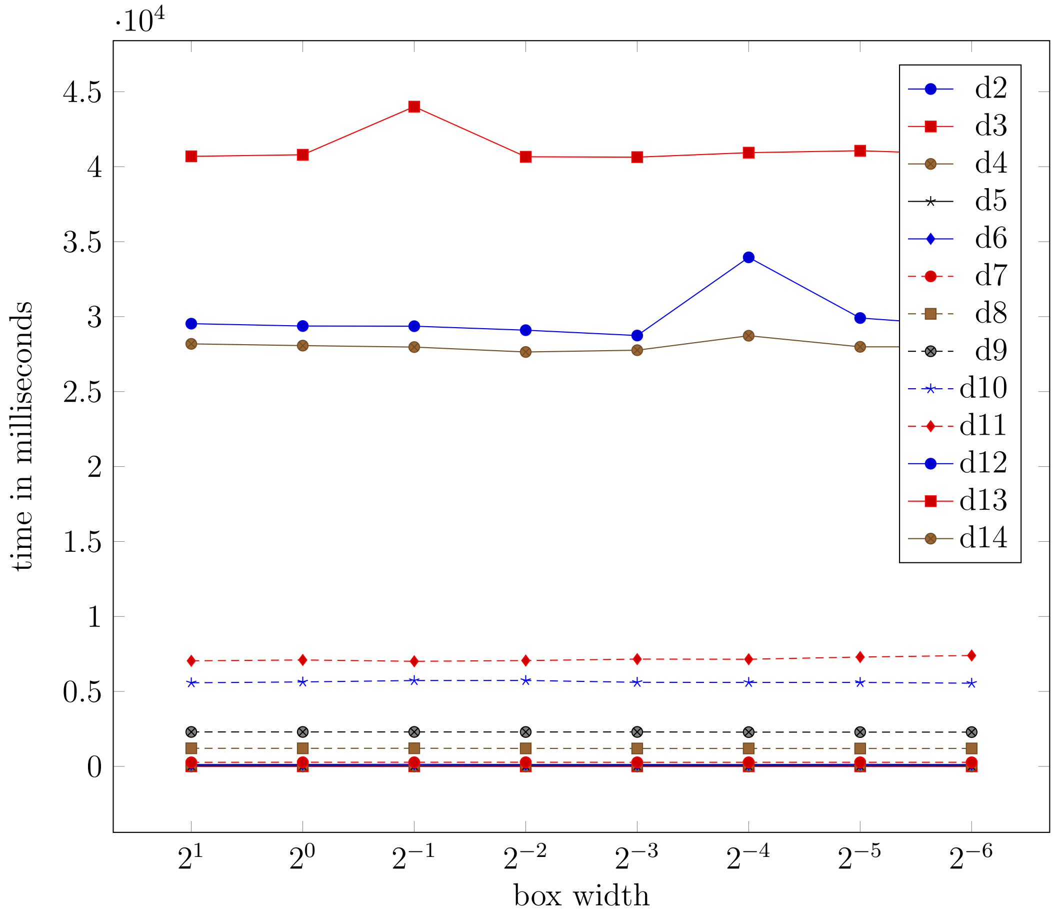

Figure 5.11: Running time of CGAL on pairs of varying degree curves with varying bounding box sizes. No marked improvement on running time with smaller bounding boxes.

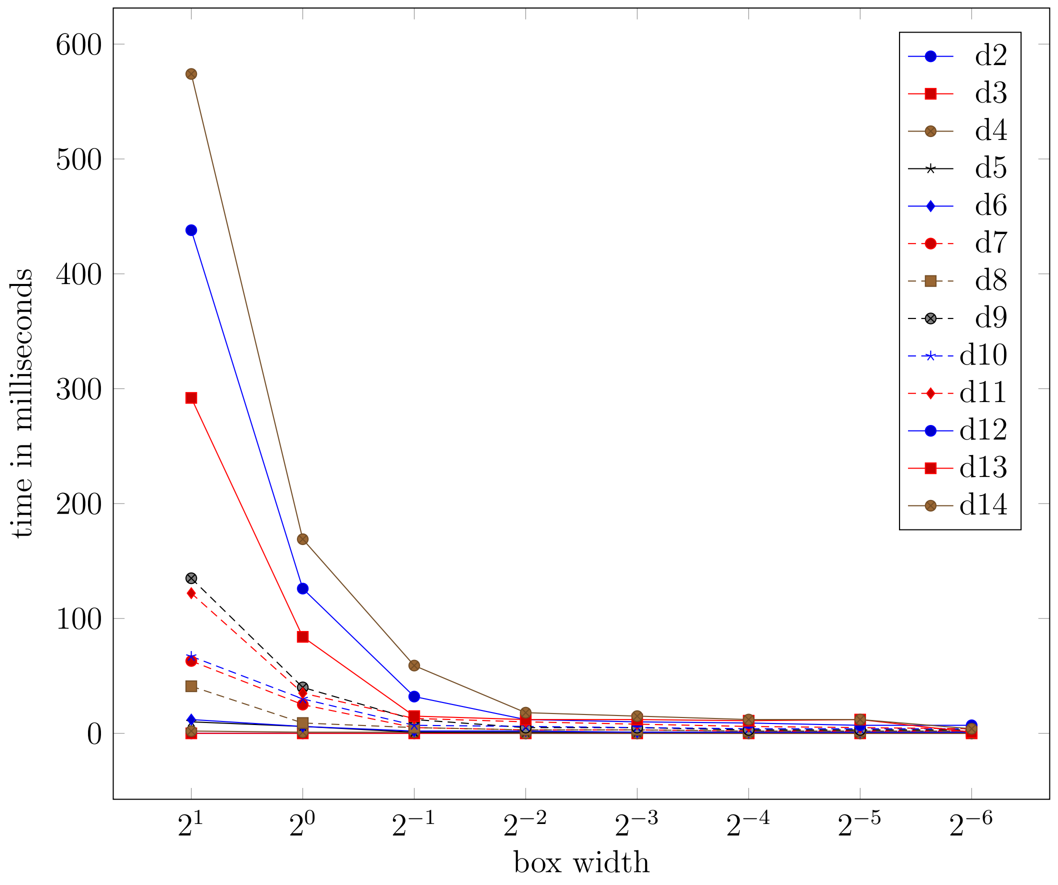

Figure 5.12: Running time of encasement on pairs of varying degree curves with varying bounding box sizes. Curves of all degrees see improvement from smaller starting box.

CHAPTER 6

Future Work

The development of the techniques for the algorithm have been very incremental, with new improvements being based on ideas that were themselves iterations of previous innovations. The results in this document are based on the current state of the algorithm and implementation, but planned improvements are already being considered.

6.1 Improved Splitting Lines

Arguably the most important decision the algorithm must make is the construction of the next splitting line. This is the only operation that modifies the current encasement, and the optimality of the line chosen is directly responsible for the efficiency of the algorithm. We in no way claim that our current strategy chooses the most optimal line. The method for choosing is based purely on heuristics and appealing to low order approximations of the curves within the cell. Given two curves f and g that do not intersect in the cell, one can qualitatively describe an optimal splitting line by its behavior around the area of the curves closest approach within the cell. Let point a on curve f and point b on curve g be the points where the distance from f to g is minimal. There is not necessarily a unique a nor b but any such points will do. At these points $\nabla f(a)$ is parallel to $\nabla g(b)$. A line at point p in the direction $\nabla f(a)^{\perp}$ separates f and g with equal distance at a and b. It is the optimal splitting line to the best linear approximation to the curves at their respective points. We hypothesize that the generation of such splitting lines would produce a more efficient encasement algorithm than the current strategy for generating splitting lines. We propose using numerical methods to find a on f such that $\nabla f(a) \times \nabla g(a) = 0$ to find the approximate region of closest approach of the curves, and construct a splitting line approximate to the splitting line described above.

6.2 Individual Encasements

An encasement that constrains the curves to be monotonic with respect to some direction has already been described to be effective in constructing robust visualizations, but we are of the opinion that constraining the curves in this way can also lead to a more efficient encasement algorithm. Our proposed idea is to perform individual encasements of each curve with the restriction that the curve is monotonic in either x or y within the cell. This amounts to separating f from both f_x and f_y within the cell. Thus we can use a relaxed encasement to achieve the separation. Create an encasement of curve f and the curves induced by $f_x = 0$, and $f_y = 0$. However we relax the constraint that two curves in a common cell must intersect. This results in an encasement where each cell contains either zero, one, or two curves. We are interested in the cells that contain f. Such a cell possibly intersects either f_x or f_y, but never both. Thus if such a cell excludes $f_x = 0$ then we know either $f_x > 0$ or $f_x < 0$ and either way, f is monotonic with respect to x inside the cell. The same applies to f_y. The result is that each cell containing f contains a single segment that is either monotonic in x or y. This provides the means for our proposed robust visualization. We also seek to use this procedure as a preprocessing step. To encase f and g, first compute their individual encasements. Then overlay the individual encasements and intersect edges of both to construct a decomposition of the cell. This is not necessarily a valid encasement of f and g yet. However, now any cell containing both f and g contain a single segment of each monotonic with respect to some direction. This can aid in constructing more optimal splitting lines and alternative ways of finding their possible intersection within the cell.

6.3 Improved Intersection Finding

We believe our algorithm for for isolating the intersection of f and g once a cell with an alternating boundary is found can be significantly improved. As it stands, the algorithm must subdivide the cell until a verifiable 2D interval that is entirely contained in the cell is found. At that point the relatively large 2D interval must be subdivided until Newton's method can succeed. Upon initial analysis, a significant portion of the cost of this operation goes into the linear substitution and root finding involved in subdivision, even thought the substitution of vertical and horizontal lines is slightly more efficient than general lines. We propose then to use numerical method in double precision to converge to a point close to an intersection within the cell and then verify that a 2D region around that point contains an intersection. This method would have the benefit of the high efficiency of numerical algorithms in double precision, but is complicated by the fact that we leave the realm of verifiable intersection using interval arithmetic. Care must be taken to ensure that once a region of interest is found in double, that we can construct a verifiable intersection interval around such a point.

6.4 Arrangement Construction

As it stands, the current software is primarily focused on providing the intersections of f and g verifiably. The intention of the method however is to be able to produce the entire arrangement of f and g and to have the ability to answer any associated queries about such an arrangement (point location, adjacency of cells, nesting of cells, etc). Preliminary work has been investigated into providing such queries, but are at this point incomplete. The current software can report the vertices of all arrangement cells accurately, the adjacency of each cell, and an approximation of its boundary. To accurately provide points on the boundary to an arbitrary degree of precision would require some modifications to the way that single curve segments are represented in the encasement (Sec. 4.4).

6.5 3D Analogue

The eventual goal of the technique of encasement is to be able to provide the arrangement of a set of algebraic surfaces embedded in $\mathbb{R}^3$. We believe that most of the strategies involved in computing the 2D encasement can be translated into direct analogues in 3D. Critical point finding remains very similar, the difference being the regions are now connected components of leaves of an octtree rather than a quadtree. The use of Taylor's theorem for lower bound approximations of the functions applies directly with small modification to the code. The central idea of splitting lines translates into the construction of splitting planes for 3D cells. Intersections of trivariates $f(x, y, z) = 0$ and $g(x, y, z) = 0$ on a plane with parametric equation $\vec{p} + \alpha\vec{u} + \beta\vec{v}$ results in the subproblem of finding the intersections of two bivariates $f_{\text{plane}}(\alpha, \beta)$ and $g_{\text{plane}}(\alpha, \beta)$ in a convex region, a problem we have shown in this work to be efficiently solved by encasement. Other details have to be worked out, but we believe the translation to be plausible.

Bibliography

[ABB$^+$99] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al., *Lapack users' guide*, SIAM, 1999.

[AY78] JC Alexander and James A Yorke, *The homotopy continuation method: numerically implementable topological procedures*, Transactions of the American Mathematical Society **242** (1978), 271–284.

[BEH$^+$02] Eric Berberich, Arno Eigenwillig, Michael Hemmer, Susan Hert, Kurt Mehlhorn, and Elmar Schömer, *A computational basis for conic arcs and boolean operations on conic polygons*, European Symposium on Algorithms, Springer, 2002, pp. 174–186.

[BEKS13] Eric Berberich, Pavel Emeliyanenko, Alexander Kobel, and Michael Sagraloff, *Exact symbolic–numeric computation of planar algebraic curves*, Theoretical Computer Science **491** (2013), 1–32.

[BHK$^+$05] Eric Berberich, Michael Hemmer, Lutz Kettner, Elmar Schömer, and Nicola Wolpert, *An exact, complete and efficient implementation for computing planar maps of quadric intersection curves*, Proceedings of the twenty-first annual symposium on Computational geometry, ACM, 2005, pp. 99–106.

[BHK$^+$17] Eric Berberich, Michael Hemmer, Michael Kerber, Sylvain Lazard, Luis Peñaranda, and Monique Teillaud, *Algebraic kernel*, CGAL User and Reference Manual, CGAL Editorial Board, 4.11 ed., 2017.

[BL04] Prosenjit Bose and Stefan Langerman, *Weighted ham-sandwich cuts*, Japanese Conference on Discrete and Computational Geometry, Springer, 2004, pp. 48–53.

[BO79] Jon Louis Bentley and Thomas A Ottmann, *Algorithms for reporting and counting geometric intersections*, IEEE Transactions on computers (1979), no. 9, 643–647.

[BT07] Jean-Daniel Boissonnat and Monique Teillaud, *Effective computational geometry for curves and surfaces*, Springer, 2007.

[DFMT00] Olivier Devillers, Alexandra Fronville, Bernard Mourrain, and Monique Teillaud, *Algebraic methods and arithmetic filtering for exact predicates on circle arcs*, Proceedings of the sixteenth annual symposium on Computational geometry, ACM, 2000, pp. 139–147.

[EK08] Arno Eigenwillig and Michael Kerber, *Exact and efficient 2d-arrangements of arbitrary algebraic curves*, Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2008, pp. 122–131.

[EKK$^+$05] Arno Eigenwillig, Lutz Kettner, Werner Krandick, Kurt Mehlhorn, Susanne Schmitt, and Nicola Wolpert, *A descartes algorithm for polynomials with bit-stream coefficients*, CASC, vol. 3718, Springer, 2005, pp. 138–149.

[EKP$^+$04] Ioannis Z Emiris, Athanasios Kakargias, Sylvain Pion, Monique Teillaud, and Elias P Tsigaridas, *Towards and open curved kernel*, Proceedings of the twentieth annual symposium on Computational geometry, ACM, 2004, pp. 438–446.

[EKSW06] Arno Eigenwillig, Lutz Kettner, Elmar Schömer, and Nicola Wolpert, *Exact, efficient, and complete arrangement computation for cubic curves*, Computational Geometry **35** (2006), no. 1-2, 36–73.

[ES12] Pavel Emeliyanenko and Michael Sagraloff, *On the complexity of solving a bivariate polynomial system*, Proceedings of the 37th International Symposium on Symbolic and Algebraic Computation, ACM, 2012, pp. 154–161.

[FHL$^+$07] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann, *Mpfr: A multiple-precision binary floating-point library with correct rounding*, ACM Transactions on Mathematical Software (TOMS) **33** (2007), no. 2, 13.

[GCL92] Keith O Geddes, Stephen R Czapor, and George Labahn, *Algorithms for computer algebra*, Springer Science & Business Media, 1992.

[HL04] Dan Halperin and Eran Leiserowitz, *Controlled perturbation for arrangements of circles*, International Journal of Computational Geometry & Applications **14** (2004), no. 04n05, 277–310.

[HS97] Dan Halperin and Christian R Shelton, *A perturbation scheme for spherical arrangements with application to molecular modeling*, Proceedings of the thirteenth annual symposium on Computational geometry, ACM, 1997, pp. 183–192.

[HW03] Eldon Hansen and G William Walster, *Global optimization using interval analysis: revised and expanded*, vol. 264, CRC Press, 2003.

[HW07] Iddo Hanniel and Ron Wein, *An exact, complete and efficient computation of arrangements of bézier curves*, Proceedings of the 2007 ACM symposium on Solid and physical modeling, ACM, 2007, pp. 253–263.

[MN99] Kurt Mehlhorn and Stefan Näher, *Leda: A platform for combinatorial and geometric computing*, Cambridge University Press, 1999.

[MS07] Victor Milenkovic and Elisha Sacks, *An approximate arrangement algorithm for semi-algebraic curves*, International Journal of Computational Geometry & Applications **17** (2007), no. 02, 175–198.

[MST13] Victor Milenkovic, Elisha Sacks, and Steven Trac, *Robust complete path planning in the plane*, Algorithmic Foundations of Robotics X, Springer, 2013, pp. 37–52.

[Sal85] George Salmon, *Lessons introductory to the modern higher algebra*, Hodges, Figgis, and Company, 1885.

[She97] Jonathan Richard Shewchuk, *Adaptive precision floating-point arithmetic and fast robust geometric predicates*, Discrete & Computational Geometry **18** (1997), no. 3, 305–363.

[The17] The CGAL Project, *CGAL user and reference manual*, 4.11 ed., CGAL Editorial Board, 2017.

[TOG04] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman, *Handbook of discrete and computational geometry*, CRC Press, 2004.

[WBF$^+$17] Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman, *2D arrangements*, CGAL User and Reference Manual, CGAL Editorial Board, 4.11 ed., 2017.

[Wei02] Ron Wein, *High-level filtering for arrangements of conic arcs*, AlgorithmsESA 2002 (2002), 147–154.

[WZ06] Ron Wein and Baruch Zukerman, *Exact and efficient construction of planar arrangements of circular arcs and line segments with applications*, Tech. report, Tech. Rep. ACS-TR-121200-01, Tel Aviv University, Israel, 2006.

[Yap00] Chee-Keng Yap, *Fundamental problems of algorithmic algebra*, vol. 49, Oxford University Press Oxford, 2000.