



RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
THE BLAVATNIK SCHOOL OF COMPUTER SCIENCE

Polygonal Minkowski Sums via Convolution: Theory and Practice

Thesis submitted in partial fulfillment of the requirements for the M.Sc.
degree in the School of Computer Science, Tel-Aviv University

by

Alon Baram

This work has been carried out at Tel-Aviv University
under the supervision of Prof. Dan Halperin

February 2013

Acknowledgements

I wish to thank my advisor, Prof. Dan Halperin, for his guidance, fruitful discussions and advice while preparing this document.

I would like to thank my lab members Oren Salzman, Doron Shaharabani, Kiril Solovei, Omri Perez, Michal Kleinbort and Efi Fogel for their great advice and help in topics concerning this thesis and being good friends.

I would like to give special thanks to Michael Hemmer for his great help in preparing this document, exchanging ideas and teaching me how to transform ideas to coherent text.

Finally, I wish to express my gratitude to my parents, sister and my lovely partner Shiri.

Abstract

This thesis studies theoretical and practical aspects of the computation of planar polygonal Minkowski sums via convolution methods. In particular we prove the “Convolution Theorem”, which is fundamental to convolution based methods, for the case of simple polygons. To the best of our knowledge this is the first complete proof for this case. Moreover, we describe a complete, exact and efficient implementation for the reduced convolution method for simple polygons based on a method proposed by Lien. As part of the “Convolution Theorem” we show that the “convolution cycles” exist and provide an optimal algorithm to extract those. A crucial step of the reduced convolution algorithm is to verify whether a loop of the convolution is on the boundary of the Minkowski sum. For this purpose we study three methods, namely a sweep-line collision detection, a bounding volume hierarchy and a novel ray-shooting method. The ray-shooting provides better theoretical bounds, however in practice, the bounding volume hierarchy proved to be the most efficient method. The thesis contains comprehensive comparisons of our reduced convolution implementation with the full convolution method, which are both implemented in CGAL. Up to pathological cases, our method was more efficient in terms of runtime and memory consumption. Furthermore, the benchmarks indicate that the method is competitive with the inexact implementation of Lien.

Contents



Introduction

The Minkowski sum (Minkowski addition) of two sets A and B in any Euclidean space S is defined as:

$$A \oplus B = \{x + y \mid x \in A, y \in B\}.$$

The Minkowski sum is a fundamental building block in many application fields of computation geometry. In many “real world” applications, the Euclidean space S is typically \mathbb{R}^2 and \mathbb{R}^3 . The Minkowski sum can be used to efficiently answer several types of proximity queries of two polyhedra, such as collision detection, penetration depth and minimum separation distance [?]. Answering such queries efficiently has applications in many fields, including robotic motion planning [?], assembly planning [?] and computer aided design [?]. For example, given two polygons A and B in the plane, computing the Minkowski sum of $-A$ and B results in the set of all points x such that $A + x$ and B intersects. This gives all the possible locations for robot A where it collides with obstacle B in the plane, which in turn is useful for planning collision-free paths for a robot A translating among the obstacles B .

In this thesis we concentrate on the computation of Minkowski sums in the plane. We focus on closed simple polygons A and B . We assume our polygons have n and m vertices, respectively. The Minkowski sum is represented by its border, as the plane is separated into components which are either inside, outside and on the border of the sum.

1.1 Previous Methods and Related Work

During the last four decades many algorithms were introduced to compute the Minkowski sum for polygons and polyhedra. Some methods give approximate solutions, while other are geometrically exact. For the approximate methods see, for example, [?] and [?]. For approximate and exact three-dimensional methods see [?], [?] and [?]. First we review complexity results for this problem.

1.1.1 Basic Complexity

The border of the Minkowski sum is formed from the sum of pairs of edges and vertices, where each summand is from a different polygon. Since there are $O(mn)$ such pairs, the sum induces an arrangement of $O(m^2n^2)$ edges. This bound is tight in the worst case [?]. If both A and B are convex, the Minkowski sum has at most $m + n$ vertices and may be computed efficiently at $O(m + n)$ time [?]. If only one of the polygons is convex, the sum has $O(mn)$ vertices [?] and may be computed at $O(mn \log(mn))$ time [?].

Most of the exact methods can be divided into two main categories: decomposition and convolution. We elaborate on the convolution in more depth below, as the main topic of this thesis are convolution-based methods, but we start with a short description of the decomposition approach.

1.1.2 Decomposition Methods

The decomposition approach entails the decomposition of the complex input geometrical entities into simpler units to start with. Then the pairwise Minkowski sums between them are computed and unified (union) into the full sum. The approach was first proposed by Lozano-Pérez [?]. Flato [?] has implemented the first exact and robust version of the decomposition method, which handles degenerate (low dimensional) features. The non-convex polygons are decomposed into convex polygons. Then, the pairwise Minkowski sums are computed. Finally the intermediate pairwise Minkowski sums are unified. Each step in this process can create various effects on run time efficiency. The choice of the decomposition method determines how many intermediate polygons and how complex the decomposed polygons are. The simplicity of this method comes at a price of computing the decomposition itself. The number of intermediate convex polygons naturally greatly influences the amount of work required during the union step. Flato tried different decomposition methods and heuristics, and an incremental method for computing the union. Hachenberger [?] implemented a robust decomposition method for computing three-dimensional Minkowski sums of polyhedra using CGAL.

1.1.3 Convolution Methods

In 1983 Guibas et al. [?] presented a framework of operations on polygonal tracing (which extends the concept of a polygon). In this framework the authors define an operation on two tracings called “convolution”. They state a theorem, the “Convolution Theorem”, that implies that the Minkowski sum boundary is a subset of this convolution. However, all the theorems in this paper are given without proofs.

Convolution methods rely on the computation of a super set of the Minkowski sum boundary and filtering out the features that are not on it. Wein [?] implemented the convolution method for planar sums of two simple polygons in CGAL. In order to extract the Minkowski sum border, he computes the arrangement of the convolution segments and computes the winding number for each face in the arrangement. According to the stated (but not explicitly proven) theorem in [?], the Minkowski sum interior regions are cells with positive winding numbers. This method outperforms the best decomposition methods in most cases where input polygons cannot be easily decomposed [?]. The implementation of Wein is also exact and robust and is able to compute all the low dimensional features.

Kaul et al. [?] observed that only a subset of the convolution segments, namely “Convex Convolution”, actually contains the Minkowski sum boundary. The convex convolution does not induce cycles, i.e., it does not induce winding numbers on the faces in the arrangement induced by the convex convolution. Therefore, in order to filter out edges which are not on the boundary of the Minkowski sum, collision detection is performed based on the test described in the beginning of this chapter.

Milenkovic and Sacks [?] define the “Monotonic Convolution”, which is another superset of the Minkowski sum boundary. This set defines cycles and induces winding numbers, which although different from those induced by the original convolution, are positive only in the interior of the Minkowski sum. This fact is shown in their paper.

Since Guibas et al. [?] did not provide proofs for the theorems in their paper, it is our belief that a small gap in the theory of the convolution methods exists. In the introduction of the thesis of Ramkumar [?], there are claims that show the basis for the correctness of the “Convolution Theorem”. However, some of these claims are without proof. Ramkumar [?] shows how to implement the convolution method for three-dimensional polyhedra.

Recently, Lien [?] proposed a convex convolution based method, which adds additional filters before using the collision detection test. It shows faster results than Wein’s method, but do not compute the degenerate features. Lien extended this method for three-dimensional polyhedra [?].

Methods based on similar ideas were proposed for two-dimensional and three-dimensional Minkowski sums. Ghosh [?] introduced “slope diagrams”. Fogel used gaussian maps to implement robust and exact three-dimensional Minkowski sums for convex polyhedra using CGAL. There are additional more similar works for three-dimensional Minkowski sums, however there is no robust implementation for general three-dimensional polyhedra [?].

1.2 Contribution and Thesis Outline

This thesis explores both the practical and the theoretical aspects of the convolution method and discusses two main subjects.

The first topic is partially filling the theory gap alluded to before by presenting a proof for the theorem presented in [?], for the special case of two simple polygons in the plane. To the best of our knowledge no proof was published to this theorem.

The second is the exact implementation of the reduced algorithm in CGAL and the provision of a precise proof that the implementation is correct. We chose to implement and study the reduced convolution algorithm since it is extendable for the three-dimensional case [?].

First we organize and review the basic definitions and terms that are used throughout this thesis and explain in a rigorous way why different convolution sets are all super sets of the Minkowski sum boundary. This can be found in Chapter ???. We then define and prove the existence of “convolution cycles”—a concept which is found in the literature but was not rigorously defined, and give an optimal algorithm to trace them (in the first part of Chapter ???). Afterwards, we prove the “convolution theorem” itself, which gives theoretical justification for the convolution method implemented by Ron Wein and for part of the correctness proof given in [?] for his method (in the second part of Chapter ???).

We discuss our robust implementation of the reduced convolution algorithm in CGAL, which handles degenerate cases. This algorithm is based on the fact that the Minkowski

sum boundary does not contain segments that are the sum of a reflex vertex and an edge. This reduces the number of segments that need to be inserted in an arrangement, thus improving runtime and memory consumption. We present the algorithm and its correctness in Chapter ???. We performed extensive benchmarks of our implementation of the reduced convolution algorithm in CGAL, measured it against the full convolution algorithm implemented in CGAL by Ron Wein and found it to be faster in most cases. We present various benchmarks to show the strength and weaknesses of each algorithm in Chapter ???. Convolution based methods require a method to decide whether a hole loop is on the boundary of the Minkowski sum. We discuss several options for implementing this predicate based on either collision detection or winding numbers. We present a ray-shooting method as an alternative to the traditional collision-detection methods. The ray-shooting method has better theoretical bounds for the test, but its implementation is complex and outside the scope of this thesis. The method that proved to be the fastest in practice is collision detection by a bounding volume hierarchy approach. This discussion can be found in Chapter ???. We further investigate the programming aspects of the implementation of the reduced convolution algorithm in CGAL and briefly describe which methods and optimization were done in order to make this implementation more efficient, in Chapter ???.

Finally, we add some concluding remarks and propose further work that can be pursued in Chapter ???.

2

Terminology and Initial Analysis

Definition 2.0.1 (Minkowski sum). The Minkowski sum of two sets A and B in the plane¹ is defined as:

$$A \oplus B = \{x + y \mid x \in A, y \in B\}.$$

The main focus of this thesis is computing the Minkowski sum of two polygons in the plane via convolution-based approaches. The Minkowski sum is defined by its boundary features, and whether the faces of the planar subdivision induced by those features are interior or exterior to the sum. In Section ??, we define polygons as they are represented in this thesis. Section ?? describes three sets of segments for which the Minkowski sum boundary is contained within. In addition, the first set is a superset of the second set, while the second set is a superset of the third set.

2.1 Polygon Representations

Definition 2.1.1 (Polygonal Chain). A polygonal chain of length n is specified by a sequence of vertices (v_1, \dots, v_{n+1}) such that each two consecutive vertices are connected by a segment.

Definition 2.1.2 (Simple Polygonal Chain). A simple polygonal chain is a polygonal chain where the segments composing the chain intersects only at consecutive vertices. Moreover, each vertex is unique.

Definition 2.1.3 (Closed Polygonal Chain). A closed polygonal chain is a polygonal chain where $v_1 = v_{n+1}$.

Definition 2.1.4 (Simple Closed Polygonal Chain). A simple closed polygonal chain is a simple polygonal chain where all vertices are unique with the exception of $v_1 = v_{n+1}$.

Definition 2.1.5 (Interior Disjoint Segments). A set of segments is called interior disjoint if the intersection of the interiors of any pair of segments is empty.

¹This definition applies to any Euclidian space but our discussion in the thesis focuses on the planar case.

Definition 2.1.6 (Weakly Simple Polygonal Chain). A weakly simple polygonal chain is a polygonal chain where the segments are interior disjoint and if a vertex v_i lies on the interior of a segment s , the incoming and outgoing segments to v_i must be on the same side (half-plane) of the segment s .

Definition 2.1.7 (Segment Direction). The direction for a segment $s = (v_i, v_{i+1})$ in a polygonal chain is defined as $\vec{d} = v_{i+1} - v_i$. Notice that this direction is retained even if the segment undergoes translations, or is split into sub-segments. We consider two directions d_1 and d_2 equal if and only if there exists $\alpha > 0$ such that $d_1 = \alpha d_2$.

We refer to v_i as *source*(s) and v_{i+1} as *target*(s).

Definition 2.1.8 (Segment Normal). We denote the normal to a segment s by n_s . The normal is orthogonal to the direction of s namely, by convention, to the right of s .

Assume we place an observer at a vertex v_1 of the polygonal chain facing the next vertex in the chain. When the observer moves from v_1 to v_2 the observer turns (by the minimal angle) to face v_3 . We define this angle to be positive for a right turn and negative for a left turn. By taking a series of such moves from one vertex to the next the observer performs a series of turns that could be summed up. This summation is related to the orientation of closed polygonal loops. By taking the directions of the segments the observer traverses as vectors, we can define the following expression that measures the angle the observer turns while moving from one segment to the next coupled with a sign stating whether the observer makes a left or right turn.

Definition 2.1.9 (Signed angle). Let the skew product of two vectors u and w in the plane be: $[u, w] = u_x \cdot w_y - u_y \cdot w_x$. The signed angle between two vectors u and w is $\angle(u, w) = \min(\alpha, 2\pi - \alpha) \cdot \text{sign}([u, w])$, where α is the angle between the vectors.

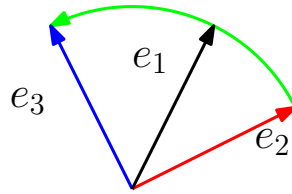
This expression assigns positive values to vectors that are on the right-hand side of u and negative values to vectors on the left-hand side. Note that for close polygonal chains summing this value over all vertices equals a multiplication of 2π by some integer. For closed weakly simple polygonal chains this number is ± 1 and the sign corresponds with the chain's orientation (i.e. negative for counterclockwise chains and positive for clockwise chains).

Definition 2.1.10 (Counterclockwise between). We say that the direction d is counterclockwise (ccw) between directions d_1 and d_2 , if $d \neq d_1$ and while rotating from d_1 to d_2 in counterclockwise fashion, d is encountered before reaching d_2 .

Definition 2.1.11 (CCW predicate). For directions d_1 , d_2 and d_3 , which have direction associated with them, we write $CCW(d_1, d_2, d_3)$ to denote that the direction d_1 is counterclockwise between the direction d_2 and the direction d_3 . If the predicate is written without a value, the value is assumed to be true. Otherwise, we may write $CCW(d_1, d_2, d_3) = \text{true}$ or $CCW(d_1, d_2, d_3) = \text{false}$.

Note that we may sometimes write the predicate with the edges themselves, in which case we mean to take the direction of the edges, as illustrated in Figure ??.

We represent polygons as a collection of closed polygonal chains, which are oriented such that the interior of the polygon is to the left of the segments of the chains. For a polygon edge e , the normal is pointing outwards of the polygon, namely to the right.

Figure 2.1: Illustration of $CCW(e_1, e_2, e_3)$.

Definition 2.1.12 (Simple Polygon). A simple polygon is a single simple closed polygonal chain.

We next define the tangent set for a boundary point of a polygonal chain.

Definition 2.1.13 (Tangent Set). The tangent set T_{v_i} of a chain vertex v_i is the set of directions, which are counterclockwise between $\overrightarrow{v_{i-1}v_i}$ and $\overrightarrow{v_iv_{i+1}}$. The tangent set of a point on a segment contains the direction of the segment only.

Note that for a reflex vertex, that is, a vertex where the angle from the predecessor edge to the successor edge through the polygon's interior is greater than π , the set contains directions which are not tangent in the standard geometrical meaning.

Definition 2.1.14 (Weakly Simple Polygon). A weakly simple polygon is a single weakly simple closed polygonal chain.

Definition 2.1.15 (Planar Subdivision). Given a set C of n planar curves, the arrangement $A(C)$ is the subdivision of the plane induced by the curves in C into maximally connected cells. The cells can be 0-dimensional (vertices), 1-dimensional (edges) or 2-dimensional (faces).

In the planar subdivision induced by a collection of closed polygonal chains, where no two chains have edges crossing each other, we define the following relationship between the chains:

Definition 2.1.16 (Polygonal Chains Relationship). A polygonal chain PC is a child of polygonal chain PC' if it is interior to the face that is enclosed by PC' and PC' 's boundary touches the same face as the boundary of PC .

Definition 2.1.17 (General Polygonal Set). A collection of weakly simple closed polygonal chains defines a non-simple polygon when all chains are interior disjoint and do not cross each other. Furthermore, any polygonal chain PC which touches the outer face is oriented counterclockwise and any child loop of PC is clockwise oriented. This rule is applied recursively for every child with the orientation flipped.

We say that a polygon is open when we consider the interior of the polygon only, i.e., without its boundary, as opposed to a closed polygon, where both the interior and the boundary of the polygon are considered.

2.2 Minkowski Sum Boundary Supersets

2.2.1 Vertex Edge Sum

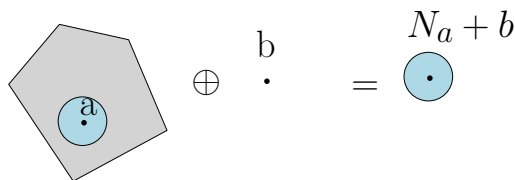
Let P and Q be polygons. We consider the sum of a vertex of one polygon with an edge of the other as the translation of the edge by the radius vector to the vertex. We now define the set of all those sums for P and Q :

Definition 2.2.1 (Vertex-Edge Sum). Let P and Q be two polygons, each composed of a collection of polygonal chains. We take all pairs of polygonal chains from P and Q with vertices (p_0, \dots, p_{n-1}) and (q_0, \dots, q_{m-1}) , respectively. The Vertex-Edge Sum, denoted by \mathcal{S} contains all segments of the form $\overrightarrow{(p_i + q_j)(p_{i+1} + q_j)}$ and $\overrightarrow{(p_i + q_j)(p_i + q_{j+1})}$.

We now show that the set \mathcal{S} is a super set of the Minkowski sum boundary. We first begin by showing that a feature on the boundary of the Minkowski sum must be the sum of two features on the boundary of the polygons. Then we show that for any interior point p of an edge in the Minkowski sum boundary there exists a vertex w.l.o.g. from P and an edge from Q such that their sum contains p . Moreover, the Minkowski sum interior is to the left of the edge. Since the vertices of the Minkowski sum boundary are contained in \mathcal{S} by definition, we can conclude that $\partial(P \oplus Q) \subseteq \mathcal{S}$.

Lemma 2.2.2 (Boundary feature source). Let p be a point on the boundary of the Minkowski sum of two closed sets A and B . For every point $a \in A$ and point $b \in B$ such that $a + b = p$, a is on the boundary of A and b is on the boundary of B .

Proof. Assume by contradiction that (w.l.o.g.) a is not on the boundary of A . Then we can find an $\varepsilon > 0$ environment around a , N_a , where the entire environment is inside A . Thus the entire disk $N_a + b$ is inside the sum, contradicting $a + b = p \in \partial(A \oplus B)$. \square



It immediately follows from Lemma ?? that:

Corollary 2.2.3. Every point on the boundary of the Minkowski sum of two polygons P and Q is the sum of a point on the boundary of P with a point on the boundary of Q .

Observation 2.2.4 (Edge-Point Neighborhood). Let $x = p + q$, such that w.l.o.g. p is edge-interior to the edge e_p . Around x there is an ε -neighborhood, where half of it, namely a half-disk is interior to the Minkowski sum. The disk is oriented such that its base is aligned with e_p , and the disk interior lies to the left of e_p .

Proof. Around p there is an ε -neighborhood such that the half-disk oriented in the manner described is interior to P . By definition the sum of each point in the half disk with q is interior to the Minkowski sum. Therefore, the same half-disk neighborhood is found around x . \square

The Minkowski Sum of Two Segments

We now discuss the Minkowski sum of two segments, as it will help us show the next property of Minkowski sums. Let $s_1 = \overrightarrow{a, b}$ and $s_2 = \overrightarrow{c, d}$. Let $\vec{v}_1 = b - a$ and $\vec{v}_2 = d - c$. By the definition of the Minkowski sum every point x in $s_1 \oplus s_2$ must satisfy:

$$x = a + \lambda_1 \vec{v}_1 + c + \lambda_2 \vec{v}_2, \text{ for some } \lambda_1, \lambda_2 \in [0, 1]. \quad (2.1)$$

Now we assume w.l.o.g. that $a + c$ is at the origin. Thus $x = \lambda_1 \vec{v}_1 + \lambda_2 \vec{v}_2$. Let M be the matrix whose columns are \vec{v}_1 and \vec{v}_2 . We can write the previous expression as:

$$\begin{pmatrix} \vec{v}_1 & \vec{v}_2 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = x.$$

From this expression we see that the image of the Minkowski sum of two points on s_1 and s_2 is equal to the image of the transformation matrix M applied on the vector $\vec{v} = (\lambda_1, \lambda_2)^t$. Since for all x in $s_1 \oplus s_2$ λ_1 and λ_2 are in the range $[0, 1]$, the set of such vectors v is exactly the unit square. Therefore, the Minkowski sum of s_1 and s_2 is the image of the unit square under M .

M is a linear transformation which can be decomposed using a singular value decomposition [?] into a series of transformations which change the base into the singular vector base, perform scaling in the directions of the singular vectors and change the base back to the original base. If M is full rank the change-of-basis transformations are orthogonal matrices which represent rotations or reflections. In this case M transforms the boundary of the unit square into the boundary of the image set, since reflections, rotations and scaling cannot map boundary points to non-boundary points and vice versa. We note that the boundary of the unit square has an extreme value for at least one of λ_1 and λ_2 . Consequently, points that are on the boundary of $s_1 \oplus s_2$ are the image of at least one of the vertices a, b, c, d .

M is not full rank if and only if \vec{v}_1 and \vec{v}_2 are linearly dependent. Note that this case occurs when s_1 and s_2 have the same direction up to a sign. By definition \vec{v}_1 and \vec{v}_2 are linearly dependent if there exist a scalar s such that $s\vec{v}_1 = \vec{v}_2$. Let \hat{v}_1 and \hat{v}_2 be the unit vectors of \vec{v}_1 and \vec{v}_2 , respectively. We can conclude the following:

$$\begin{aligned} \vec{v}_1 &= \hat{v}_1 \|v_1\|, \\ \vec{v}_2 &= \hat{v}_2 \|v_2\|, \\ s\hat{v}_1 \|v_1\| &= \hat{v}_2 \|v_2\|, \\ s &= \pm \frac{\|v_2\|}{\|v_1\|}. \end{aligned}$$

s is positive if \hat{v}_1 and \hat{v}_2 are in the same direction and negative otherwise. We can always flip our choice of direction for one of the segments such that $\hat{v}_1 = \hat{v}_2$. By substituting into Equation ?? we get:

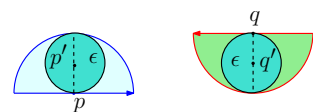
$$x = a + c + (\lambda_1 \|v_1\| + \lambda_2 \|v_2\|)\hat{v}_1.$$

This sum defines a segment that begins at $a + c$ and reaches $a + c + (\|v_1\| + \|v_2\|)\hat{v}_1$. Any intermediate point is in the sum by the correct choice of λ_1 and λ_2 . Thus, the sum is a segment with length $\|v_1\| + \|v_2\|$.

Lemma 2.2.5 (Sum of two points on edges). *Let P and Q be polygons. Let e_p and e_q be edges of P and Q , respectively. Let $p \in e_p$ and $q \in e_q$ be edge-interior points (i.e. not vertices). If $x = p + q$ is on $\partial(P \oplus Q)$ then $T_p = T_q$, where these are the tangent sets of p and q , respectively.*

Proof. By the discussion above if the slope of e_p is different from the slope of e_q , $p + q$ lies interior to the Minkowski sum. It is left then that either $T_p = T_q$ or $T_p = -T_q$.

The case $T_p = -T_q$ occurs when two segments with opposite directions one from each polygon, are summed up. We assume w.l.o.g. that e_p and e_q are horizontal as depicted in the figure to the right. Let $\varepsilon > 0$ be chosen such that there is a half disk neighborhood around p and q which is contained inside P and Q , respectively. Let $p' = p + (0, \frac{\varepsilon}{2})$, $q' = q - (0, \frac{\varepsilon}{2})$. Around p' and q' there are an



$\frac{\varepsilon}{2}$ -neighborhoods that are contained in P and Q respectively. However, $p' + q' = x$, that is x is contained in the sum of the $\frac{\varepsilon}{2}$ -neighborhoods around p' and q' , thus interior to the Minkowski sum and not on the boundary. In conclusion, for x to be on the boundary of the Minkowski sum, it is necessary that $T_p = T_q$. \square

Now we can show that $\partial(P \oplus Q) \subseteq \mathcal{S}$:

Theorem 2.2.6 (\mathcal{S} is a Superset of the Minkowski Sum Boundary). *Let P and Q be polygons. Any point x which belongs to $\partial(P \oplus Q)$ is contained in the sum of a vertex of P and an edge of Q or the sum of vertex of Q and an edge of P .*

Proof. From Lemma ?? we know that all the sources of x lie on the boundary of P and Q . Let $x = p + q$, $p \in P$ and $q \in Q$. Denote by e_p and e_q the edges on which p and q lie, respectively. If either p or q is already a vertex then $x \in p + e_q$ or $x \in e_p + q$, respectively, and we are done.

Thus assume that p and q are edge interior. By Lemma ?? we have that $T_p = T_q$. We saw in the discussion above that the Minkowski sum of two collinear segments is a segment with the length of their sum. We assume w.l.o.g. that e_p is longer than e_q . The union of the sum of e_p with both the source vertex of e_q and the target vertex of e_q is exactly $e_p \oplus e_q$, thus one of these sums in the union contains x . That is, x is contained in either $source(e_q) + e_p$ or $target(e_q) + e_p$. \square

2.2.2 The Convolution Set

We now define a subset of \mathcal{S} that contains only the segments of \mathcal{S} each of which direction is contained in the tangent set of the vertex which was used to create it. This set is called the convolution and was proposed by Guibes et al [?].

In the following definition we have two input polygons P and Q . We treat the directions of Q as being infinitesimally rotated counterclockwise such that none of them equals any direction of P . In particular, for originally equal directions d_p and d_q from P and Q , respectively, d_q is more counterclockwise to d_p with respect to the *CCW* predicate.

Note that this treatment only defines an order on the directions and does not change the resulting convolution set. The choice to make d_p more counterclockwise than d_q will result in the same convolution set, with a different set of original segments (see the discussion about Minkowski sum of two segments above especially for the case of two segments with equal directions).

Definition 2.2.7 (Convolution of two polygons in the plane). Let P and Q be two polygons, each composed of a collection of polygonal chains. We take all pairs of polygonal chains from P and Q with vertices (p_0, \dots, p_{n-1}) and (q_0, \dots, q_{m-1}) , respectively. The convolution is the set of segments that agree with the following rule: Any segment of the form $\overrightarrow{(p_i + q_j)(p_{i+1} + q_j)}$ if the direction d of the vector $\overrightarrow{p_i p_{i+1}}$ lies counterclockwise between the directions d_1 and d_2 of the vectors $\overrightarrow{q_{j-1} q_j}$ and $\overrightarrow{q_j q_{j+1}}$, respectively. Such a segment is the matching of an edge from P and a vertex of Q . Analogously, matches of this fashion between edges from Q and vertices from P are included in the collection. We denote this set as $\mathcal{C} = P \otimes Q$. Obviously, $\mathcal{C} \subseteq \mathcal{S}$.

Definition 2.2.8 (Alternative definition for convolution). Let A and B be polygonal curves in the plane:

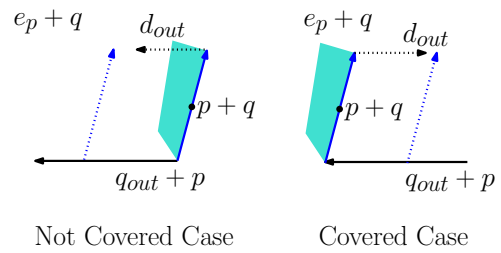
$$A \otimes B = \{x + y \mid x \in \partial A, y \in \partial B, T_x \cap T_y \neq \emptyset\}.$$

Note that the definition for the tangent set of a vertex spans precisely the matching edges of Definition ??; therefore the segments which are included in the convolution by Definition ?? include the same set of points as the set defined in Definition ??.

We now show that the convolution is also a superset of the Minkowski sum boundary. For this purpose we introduce the following lemma which is based on a local criteria that shows cases where we can immediately conclude that the sum of an edge and a vertex is interior to the Minkowski sum.

Lemma 2.2.9 (Sliding Lemma). *Let e_p be an edge w.l.o.g. of P and q a vertex of Q . Let q_{out} be the outgoing edge of q in the polygonal chain Q . The segment $e_p + q$ is interior to the Minkowski sum if the the direction d_{out} of q_{out} faces towards the half space to the right of e_p . The same holds for the incoming edge of q , q_{in} and reversed direction \widetilde{d}_{in} of q_{in} .*

Proof. Let p be a point interior to e_p . Assume that the direction d_{out} , points to the half space to the right of e_p . In the figure to the right (Covered Case) we see that we may “slide” e_p in the direction of d_{out} , which means summing e_p with the corresponding points on q_{out} and obtaining a swept volume which is interior to the Minkowski sum. However, this volume contains $p + q$ which is thus interior to the Minkowski sum. Since this is true for any interior point to e_p , $e_p + q$ is interior to the sum. The same argument holds for “sliding” e_p in the direction of \widetilde{d}_{in} . \square

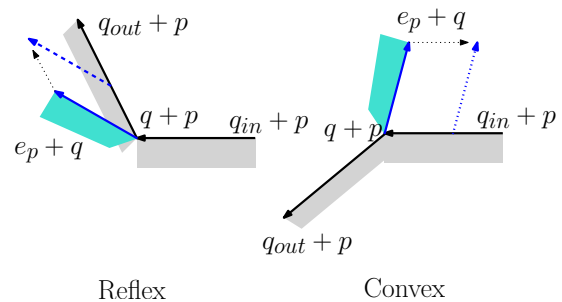


In the Not Covered Case we see that the sliding does not cover $p + q$ and therefore it is not necessarily interior to the Minkowski sum.

Theorem 2.2.10 (Convolution is a superset of the Minkowski sum boundary). $\partial(P \oplus Q) \subseteq P \otimes Q$.

Proof. By Theorem ??, we know that the boundary of the Minkowski sum is contained in the segments of the Vertex-Edge Sum \mathcal{S} . We examine the segments of \mathcal{S} that are not in the convolution. A segment in \mathcal{S} is the sum of an edge e_p w.l.o.g. of P and a vertex q of Q . Let q_{in} and q_{out} be the incoming and outgoing edges to q , respectively. When e_p is not in the tangent set of q , at least one of the edges $-q_{in}$ or q_{out} points to the right half-space of e_p and by Lemma ??, $e_p + q$ is interior to the Minkowski sum.

In the figure to the right we see the two possible cases for q which can be either a convex or reflex vertex. In the Convex example we see that q is convex and e_p is not in the tangent set of q . In this case it is possible to “slide” e_p along q_{in} as shown by the dashed arrows and all the swept regions are interior to the Minkowski sum. When q is a reflex vertex the same argument holds as seen in the Reflex case.



Therefore, edges that are not in the tangent set of their corresponding vertex are not on the Minkowski sum boundary and by Definition ?? not in the convolution as well. Thus, the convolution is a superset of the Minkowski sum boundary. \square

2.3 The Reduced Convolution

Kaul et al. [?] observed that only those segments of \mathcal{S} whose original boundary features sum contains a convex vertex are actually on the Minkowski sum boundary. We denote this set of segments as reduced convolution.

Definition 2.3.1 (Reduced convolution). A subset of the convolution (Definition ??) where each segment originates from the sum of an edge and a convex vertex. This convolution is also referred to as *convex convolution*.

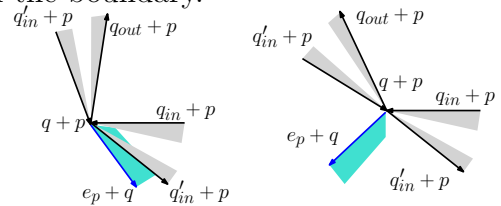
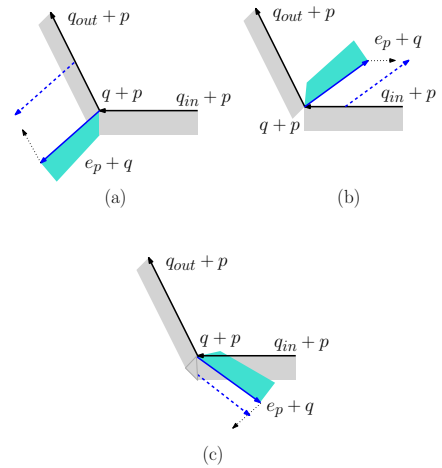
Note that equivalently we may state that the tangent set of a reflex vertex is empty. Thus the tangent sets now get the intuitive definition while using the convex convolution.

Let us show now that indeed the reduced convolution is also a superset of the Minkowski sum boundary. We use the same idea seen in the proof of Theorem ?. Although a proof is given in [?], for the sake of completeness we provide an alternative proof for this observation, and show specifically why Theorem ?? holds for non-simple polygons as well.

Theorem 2.3.2 (Reduced Convolution is a Superset of the Minkowski Sum Boundary). Let P and Q be polygons. Given a segment $s = e_P \oplus q$ of an edge (w.l.o.g.) $e_P \in P$ and a vertex $q \in Q$, if q is a reflex vertex, s is not on the boundary of the Minkowski sum of P and Q .

Proof. Let q be a reflex vertex of Q and e_p an edge of P . Let q_{in} and q_{out} be the incoming and outgoing edges to q , respectively. Let \tilde{d}_{in} and d_{out} be the directions of $-q_{in}$ and q_{out} , respectively. The figure to the right shows that e_p could point either inwards or outwards of the interior of the Minkowski sum locally defined by $q+p$. Assume that e_p is oriented towards the interior of the Minkowski sum. If either one of the directions \tilde{d}_{in} or d_{out} points to the right of e_p we use Lemma ?? (as seen in the figure to the right(a)). Otherwise, around q there is an ε -neighborhood where any direction d which is counterclockwise between d_{out} and \tilde{d}_{in} has a segment of length $\varepsilon \hat{d}$ (where \hat{d} is the unit vector in the direction d) in the direction of d interior to Q (see the figure to the right (c)). Since this is a reflex vertex, the angle while rotating counterclockwise from d_{out} to \tilde{d}_{in} is greater than π , which means that there is a direction d interior to Q that points to the right of e_p . “Sliding” e_p along d will cover e_p and therefore $e_p + q$ is interior to the Minkowski sum. In the same manner, assume that e_p is oriented outwards of the interior of the Minkowski sum. Thus, by sliding e_p along q_{in} the resulting sums are interior to the Minkowski sum and again s is not on the boundary.

Now we examine what happens when the neighborhood of $p+q$ (where p is the source vertex of e_p) is locally not a manifold. In the figure to the right we have two additional edges of Q , q'_{in} and q'_{out} . Note that in this case Q is not a simple polygon. The pair of edges which decide whether e_p might be on the boundary is the pair that directly encompasses e_p . If this pair induces a reflex vertex then by the same arguments seen above, e_p is not on the boundary of the



Minkowski sum. However, if they induce a convex vertex then e_p is a potential boundary segment. \square

3

Reduced Convolution Algorithm

In Chapter ?? we have shown that the Minkowski sum boundary is a subset of the convolution segments (Theorem ??). This fact allows different methods to extract the Minkowski sum boundary out of the convolution segments. The difference between the methods is the manner in which they decide which segments are part of the Minkowski sum boundary. Behar and Lien [?] proposed an algorithm that extracts the edges of the Minkowski sum boundary using a collision detection predicate which is discussed in this chapter. This is opposed to an algorithm proposed by Wein [?] (discussed in Chapter ??), which uses *winding numbers*. We implemented a variant of the reduced convolution algorithm using CGAL. This variant, however limited for the case of two simple polygons, is robust and handles low dimensional boundary features as we show in the sequel. In Section ?? we briefly describe the algorithm by Behar and Lien and provide additional proofs for concepts presented in their paper. Section ?? describes our variant of this algorithm which is implemented in CGAL, together with proofs verifying its correctness. In Section ??, we describe how to use capabilities of CGAL's planar arrangements in order to improve the efficiency of the algorithm. Finally, when we treat polygons as open sets, low dimensional features (edges and vertices) may appear on the boundary of the Minkowski sum. For now we assume that the input polygons are closed sets. In Section ?? we discuss the polygons as open sets case.

3.1 The Algorithm by Behar and Lien

The algorithm proposed by Behar and Lien [?], uses the fact that a point x is interior to the Minkowski sum of P and Q if and only if the intersection of P and $-Q$ translated by x is not empty in its interior; see Chapter 13 [?]. Thus, a point on an edge on the boundary of the Minkowski sum causes P and $-Q^x$ to touch. This is the collision detection predicate. Since only a part of a convolution segment may be on the boundary, the algorithm splits the convolution segments at their intersection points, say by computing the edges of the planar subdivision induced by the convolution segments. The collision detection predicate can then be performed for one point per edge of the refined subdivision to determine the Minkowski sum boundary edges. Performing a collision detection test for

every edge is costly. However, following the observation summated in Theorem ??, Behar and Lien noted that they may disregard all those segments that are the sum of a reflex vertex and an edge, i.e. use the reduced convolution (see Definition ??). Next Behar and Lien observed that the boundary of the Minkowski sum must be composed of loops that are manifold and orientable, where orientable means that all the edges of a loop are directed (see Definition ??) such that the destination vertex of each edge coincides with the source vertex of the subsequent edge. Moreover, loops that reside on the Minkowski sum boundary must obey a “nesting order”: loops that have normals pointing outwards must directly encompass loops that have normals pointing inwards and vice versa. After removing all edges which do not satisfy those filters it remains to test for each loop whether it is on the boundary of the Minkowski sum. Finally, Behar and Lien observed that a single test for each loop is in fact sufficient.

We now describe the algorithm by Behar and Lien in a formal manner. In the first stage the algorithm computes the reduced convolution set. After computing the segments of the reduced convolution, the algorithm computes the edges induced by the planar subdivision of these segments. We note that each edge retains the direction of the original segment it is a subset of. The algorithm then proceeds to use the additional filters mentioned earlier. The filtering is carried out in several steps. Behar and Lien defined three filters for choosing the correct subset of segments: (i) Orientable loops filter, (ii) nesting filter and (iii) the boundary filter. We describe these filters next.

Note that the result of the filtering process is an arrangement containing exactly the Minkowski sum of P and Q . We refer to it as the arrangement of the Minkowski sum.

3.1.1 Orientable Loops

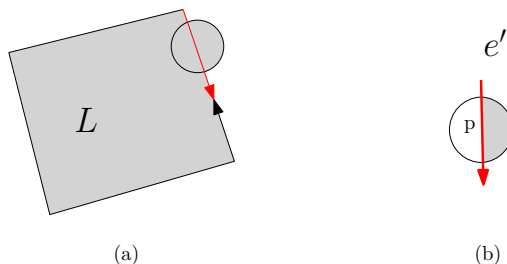
Definition 3.1.1 (Orientable loops). A loop is a closed sequence of interior disjoint segments. The loop is an orientable loop if all the segments are directed such that for every segment its target vertex coincides the source vertex of the following only.

Lemma 3.1.2. *The boundary of the Minkowski sum of two polygons P and Q is a union of orientable loops.*

Proof. By definition boundaries of the Minkowski sum must be a union of connected segments forming cycles, since they are the boundary of a face in an arrangement (which contains only the Minkowski sum). Each loop splits the face it resides in into two open sets: inner and outer (Jordan theorem), a region inside the sum and outside it. Let $L \subset \partial(P \oplus Q)$ be such a loop. Assume by contradiction that L is not an orientable loop. Now assume w.l.o.g. that

the Minkowski sum interior is inside the loop, since L is not orientable there exists at least one edge e with a normal that points inward. Now consider the ϵ -neighborhood of an interior point p on e . e splits the neighborhood into two half-disks. By assumption the half-disk in the interior of the loop is part of the Minkowski sum. Since p is an interior point, we can apply Observation ?. Thus, since the normal is pointing inward, the other part also belongs to the Minkowski sum, which contradicts the fact that the edge is part of $\partial(P \oplus Q)$. \square

We can now conclude:



Corollary 3.1.3. *The boundary of each face interior to the Minkowski sum is a weakly simple polygonal chain.*

Proof. Overlapping opposing edges cannot be on the boundary of the Minkowski sum due to Lemma ???. Since the boundary is an orientable loop surrounding a face, it must be weakly simple. \square

Thus, the algorithm extracts those orientable loops that could be a part of the Minkowski sum boundary by using the following procedure: begin with an arbitrary segment s which has not yet been visited, trace the loop along the direction of the edges, and take the largest clockwise turn from s when more than one outgoing edge appears. Since we are tracing a boundary of the Minkowski sum, we can also see the algorithm as tracing the boundary of a face which does not belong to the Minkowski sum. Then the reason for taking the largest clockwise turn is that this is the only candidate to be on the boundary of the face not containing the Minkowski sum, currently being traced. The algorithm uses this procedure iteratively until all orientable loops are traced.

Unfortunately, not all of the orientable loops are on the boundary of the Minkowski sum. The algorithm could use the collision-detection predicate for all edges of the orientable loops. However, further filtering is possible.

3.1.2 Nesting Loops

We note that there are two types of loops. One type has its normals pointing outwards, thus the Minkowski sum is inside this loop, namely an external loop. The normals of the other type are pointing inwards, which is the case where the loop encloses a hole in the sum, namely a hole loop.

If a polygon is not simple it may contain an external loop directly encompassing hole loops, and so forth. We can consider each loop to be the parent of all the boundary loops it directly contains, that is, there is a simple connected path between the parent loop and the child loop which does not intersect any other boundary loop. For a polygon P it is possible to build a tree containing the relationship for all boundary loops of P . The depth of this tree is called the nesting level of a polygon P .

The Minkowski sum boundary conforms to this nesting order as well:

Observation 3.1.4. *The loops of the boundary of a Minkowski sum must obey the nesting property, an external loop must directly enclose all hole loops, and all hole loops must directly enclose external loops only.*

For a proof see [?]. This is a result of Corollary ?? and Lemma ?? which means that the Minkowski sum boundary is the union of edge-disjoint orientable loops. Moreover, these loops can be seen as curves, which may touch but do not cross each other. Loops of the same type cannot directly encompass each other. The orientable loops which cause this condition to be violated can be discarded using an algorithm which determines the proper order. A loop that touches the outer face must be an external loop with hole loops as children, thus all children which are external loops are discarded. Now each hole loop is the parent of external loops and all children which are hole loops are discarded. The algorithm can proceed in this manner recursively until it reaches loops without children.

3.1.3 Boundary Loop Filter

Finally, Behar and Lien noted that only one collision detection test is actually required per orientable loop:

Observation 3.1.5. *All points on a false loop must make $-P$ collide with Q .*

For a proof see Behar and Lien [?]. We examine a point interior to the face of the planar subdivision induced by the convolution segments instead. Note that a face must be entirely inside or outside the Minkowski sum, since the convolution segments are on the boundary of the Minkowski sum. If this point does not cause $-P$ translated to collide with Q , then this face must be outside the Minkowski sum. Similarly, if it does then the face is interior to the Minkowski sum.

3.1.4 Algorithm Summary

To recap, we give a sketch of the algorithm presented by Behar and Lien [?] for computing the Minkowski sum boundary of two simple polygons from the reduced convolution segments.

1. Compute the reduced convolution.
2. Find the intersection points between all reduced convolution segments, and split them at the intersection points (i.e., by inserting them into an arrangement).
3. Trace all orientable loops.
4. Apply the nesting filter to filter out false loops.
5. Apply the boundary filter to filter out false loops. (Behar and Lien implemented this using a sweep.)
6. Return the remaining loops as the Minkowski sum boundary.

3.2 Implementation of a Robust Variant Using CGAL

We implemented a variant of the algorithm based on CGAL. It utilizes the exact computation paradigm for solving degenerate cases and avoiding numerical errors. This section describes the algorithmic details concerning this implementation. The current CGAL implementation is restricted for the case of two simple polygons. This allows us to simplify the code in a manner described below. Extending the code for handling the non-simple case is possible, though beyond the scope of this work. However, unless stated otherwise the algorithms and proofs presented in this section are for the general case of non-simple polygons. We follow Behar and Lien steps and describe the details of each step of the algorithm providing additional proofs which do not rely on the “convolution theorem”, which is presented in Chapter ???. In Section ??? we describe how to use arrangements for improving the efficiency of the algorithm. Finally, we describe the definition and treatment of the degenerate cases.

3.2.1 Computing the Reduced Convolution

The naive way to compute the segments of the reduced convolution is to iterate over every pair of a vertex of one polygon and an edge of the other polygon, and check whether the conditions of Definition ?? hold for them. If so then the sum of the relevant vertex and edge is added to a list of convolution segments. Wein [?] offered a faster method for this process, which takes $O(n + m + K + n_r m + m_r n)$, where K is the output size and n_r and m_r are the number of reflex vertices of P and Q , respectively. In Chapter ??, we implemented an optimal $O(n + m + K)$ algorithm and use it to compute the (reduced) convolution. An `arrangement of segments with history` [?] is constructed from the convolution segments, hence splitting the segments at intersection points. The `arrangement with history` records the original segment of each part of the split segment (so as to keep the direction).

3.2.2 Tracing Orientable Loops

This section shows how to extract the orientable loops from the arrangement that was computed in the previous step. Recall that we have a direction assigned to each segment of the convolution, and this direction is kept even if the segment is split into several edges (by intersection with other segments). The algorithm by Behar and Lien traces orientable loops by choosing an arbitrary edge as the initial edge. The algorithm traces edges by moving along the direction of the current edge to the next one until it has no outgoing edges to continue to or it closes a loop. If during this movement there is more than one outgoing edge, then the edge that constitutes the largest clockwise turn from the incoming edge is chosen. Let us define this terms more formally and then prove the validity of the loop tracing in our variant of Lien’s algorithm. Recall Definition ?? for a signed angle. By using this expression we can define the “largest clockwise turn” from one segment to the next.

Definition 3.2.1 (Largest clockwise turn). Let E be the set of outgoing edges from edge e . An edge e' is said to make the largest clockwise turn from e if it has the maximal signed angle $\angle(e, e')$ of all edges in E .

Tracing Boundary Loops Algorithm

Recall that faces in the planar subdivision induced by the convolution segments, which are not interior to Minkowski sum have weakly simple polygonal chain boundary as shown in Lemma ?. The following section discusses an algorithm which traces orientable loops that are candidates for being boundaries of these faces.

In order to trace a candidate loop for the Minkowski sum boundary, we start with an arbitrary edge in the convolution arrangement. We traverse the arrangement starting from that edge e . When there are more than one outgoing edges we choose the largest clockwise turn as the next edge to move to. We proceed in this manner until we arrive at the starting edge, at some previously visited edge or we reach a point where there is no outgoing edge. If a loop is closed, then some of the traversed edges compose a potential boundary loop. Algorithm ?? uses this method to trace boundary loops. We next present the algorithm and then prove its correctness.

Algorithm ?? traces all the boundary loops on the arrangement of the convolution. It does the following: It store all edges of the convolution arrangement in a stack (Line

Algorithm 1 Trace Orientable Loops

```

1: Def Stack E
2: Let  $E \leftarrow$  All convolution segments
3: for all  $e \in E$  do
4:   Let  $id(e) \leftarrow -1$ 
5: end for
6: Let  $current\_id \leftarrow 0$ 
7: while  $Not\_Empty(E)$  do
8:   Let  $e \leftarrow Pop(E)$ 
9:   if  $id(e) == -1$  then
10:    Let  $id(e) \leftarrow current\_id$ 
11:    Let  $e' \leftarrow Best\_Direction(e)$ 
12:    while  $e' \neq NULL \ \&\& \ id(e') == -1$  do
13:      Let  $id(e') \leftarrow current\_id$ 
14:      Let  $e' \leftarrow Best\_Direction(e')$ 
15:    end while
16:    if  $id(e') == current\_id$  then
17:       $Record\_Loop(e')$ 
18:    end if
19:    Let  $current\_id \leftarrow current\_id + 1$ 
20:  end if
21: end while

```

??). Each edge stores an id. For every edge in the stack, if it has not yet been visited, the algorithm tries to trace an orientable boundary loop if there exists a path from the starting edge to it (lines ??-??). The tracing is done by choosing an outgoing incident edge from the target vertex of the current edge, changing its id to be the id of the current edge, and moving forth (lines ??-??). If during this move we encounter an edge with the same id as the current edge, it means a loop was closed. Encountering an edge with a different id means that it was already examined at an earlier iteration of the algorithm. This means that if an orientable loop reachable from the starting edge exists, it has already been traced by a previous iteration, since we would have reached the edge with the different id. Therefore, we can begin the process from a new edge. When there are more than one outgoing edges, the algorithm takes the one that makes the largest clockwise turn from the incoming segment. At the end of this process, when the stack is empty the algorithm has traced all the orientable loops, i.e., potential boundary loops.

Next we describe two procedures used by the algorithm. Afterwards, the correctness of the algorithm is proved.

Definition 3.2.2 (Best_Direction(e)). Is a predicate which returns the segment of the arrangement e' that is outgoing from e and makes the largest clockwise turn from it.

Definition 3.2.3 (Record_Loop(e)). Is a method that begins at an edge e of the arrangement. At each step it moves from e to the $Best_Direction(e)$ and outputs the new edge, until it reaches back to e . This process traces a loop which has the same id assigned to its edges. Loops that contain two edges precisely are discarded because we treat the polygons as closed sets thus edges cannot be surrounded by Minkowski sum interior and still be on the boundary.

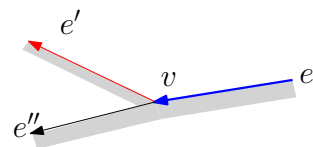
Correctness of Boundary Loops Tracing Algorithm

Now we show that Algorithm ?? traces all candidate loops for the Minkowski sum boundary. We first show that if the algorithm reaches an edge which is on a boundary loop, it traces that loop. Afterwards, we show that the algorithm traces loops which are edge disjoint, and do not cross each other (when seen as curves in the plane). We can then conclude that the output of the algorithm is a collection of edge-disjoint non-crossing orientable loops, which includes all boundary loops. This property allows for using of the nesting filter described in Section ?? to filter out orientable loops that are not on the boundary of the Minkowski sum.

Since the algorithm examines each edge in the arrangement once, we only have to make sure that once the algorithm reaches an edge on a boundary loop, it traces that loop. Recall that the only candidates for boundary loops are orientable loops. We observe next that if the algorithm reaches an edge on a boundary loop, it remains on this loop until it closes:

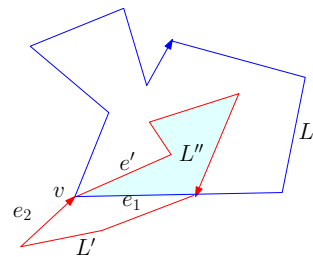
Observation 3.2.4. *Let e be an edge of the convolution arrangement, which is on the boundary of the Minkowski sum. The clockwise-most outgoing segment e' from e is also on the boundary of the Minkowski sum.*

Proof. By assumption, e is on the boundary of the Minkowski sum and therefore on the right hand side of e lies a face that is not in the Minkowski sum. While sweeping directions in counterclockwise order starting from the direction of $-e$, the first outgoing edge from e touches that outer face and therefore is on the boundary of the Minkowski sum as well. By the definition of the signed angle, this edge is e' . \square



Observation 3.2.5. *The output of Algorithm ?? is a union of edge disjoint non-crossing orientable loops.*

Proof. Assume by contradiction that the output of the algorithm contains two crossing orientable loops L and L' . We show the analysis for the case that both loops are clockwise oriented; arguments for the other cases are similar. The loops coincide in a vertex v as shown in the figure to the right. If the algorithm reaches v from e_1 it will take e' as the next segment and trace the loop L'' . Consequently, loop L will not be traced since e_1 is used for L'' . Therefore, the algorithm must reach v from e_2 . Similarly, L'' is now traced, and not L' . Therefore in any case one of the loops L or L' is not traced in contradiction to the assumption. \square



By the previous discussion, the output of the algorithm is a collection of orientable loops which contains all the boundary loops. Note that the algorithm outputs loops which are edge disjoint and the loops do not cross each other. In order to find the actual boundary loops from the collection we can now apply the Nesting filter as described in Section ??, as follows: We begin with the loops that touch the outer face and treat every such loop, which is the boundary of a connected component in the Minkowski sum, separately. For an outer loop L , we iterate the child loops of L and determine which ones are boundary loops. Each child loop is tested for violating the condition defined in the nesting filter and if it does not, we use the collision detection predicate to test if it is

on the boundary of the Minkowski sum. If one of those conditions fails, the loop is not on the boundary and all its child loops become the children of L and undergoes similar tests. For each child loop that is an actual boundary loop, the process is repeated while having the expected orientation in the Nesting filter reversed.

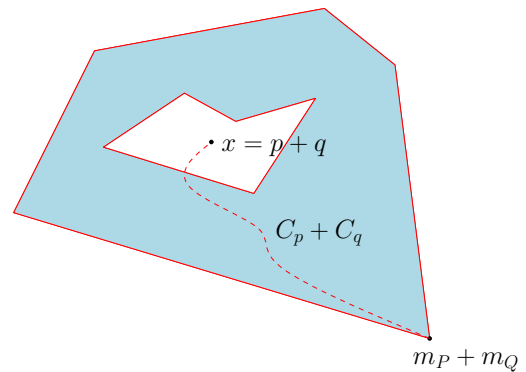
3.2.3 Using Arrangements

We note that Algorithm ?? traces loops one by one. However, we stated earlier that Minkowski sum boundaries form non-crossing loops. Consequently, they form faces in the arrangement induced by the convolution segments. This allows us to replace the algorithm for tracing loops by traversing boundaries of faces of the arrangement. We can check for a boundary of a specific face whether it is orientable, by traversing its edges. During this process we can validate that the loop is a weakly simple polygonal chain, namely that there are no antennas on the boundary (although we allow loops to touch at vertices). Later, we use the collision detection predicate and the nesting filter for the faces that are interior to the previously examined face. The arrangement structure provides quick access to the faces that are interior to a specific face (face holes), eliminating the need for additional sweep as required by Lien's algorithm.

For the simplified case of two simple polygons this method becomes much simpler and efficient. We first note the following:

Observation 3.2.6. *While not considering degeneracies (as defined in section ?? below), if the input polygons P, Q are each a simple polygon, then the resulting output is connected. Namely the output is composed of one external loop and any number of holes.*

Proof. Let P and Q be two simple polygons. Assume by contradiction that we have a region $X \subseteq P \oplus Q$, where X resides inside a hole of $P \oplus Q$. Let $x \in X$ be a point. According to Definition ?? $x = p + q, p \in P, q \in Q$. Let m_P and m_Q be the minimal points (in Y and then X coordinates) of P and Q respectively. Because P and Q are connected there exist two curves C_p and C_q that connect m_P and m_Q to p and q respectively. But the sum $C_p + C_q$ connects the minimal point of the Minkowski sum $P \oplus Q$ (which belongs to the outer boundary of the sum) to the point x , and is contained in it by definition, therefore we have a contradiction. \square



By Observation ??, the output is composed of an outer loop and zero or more hole loops. The outer loop is the outer face of the convolution arrangement. This face can be immediately extracted from the arrangement. Hole loops are clockwise orientable loops that have no convolution segments in their interior. Thus, there is no need for a nesting order check—any loop that is not clockwise or has edges in its interior is not on the boundary.

3.2.4 Degenerate Cases

Low dimensional boundary features are edges or vertices on the boundary of the Minkowski sum that do not touch faces outside the Minkowski sum (in their interior). They may

appear when the polygon boundary is not considered a part of the polygon, i.e., when the input polygons are treated as open sets. By Observation ??, the boundary of the Minkowski sum is the sum of points on the boundary of P and Q . This means that when the polygons are considered as closed, the boundary is always part of the Minkowski sum and no low dimensional boundary features can emerge. These boundary features are also referred to as degenerate features.

We show that when three or more convolution segments intersect in their interiors at the same point, an “isolated vertex” may occur which belongs to the boundary of the Minkowski sum, but its neighborhood is interior to the Minkowski sum. This is the case where one polygon (reflected through origin) fits at some placement inside the other polygon. See Figure ??. When two segments of the convolution overlap, we may get a “dangling edge”, which is an edge that belongs to the Minkowski sum boundary, but to both its left and right there are regions interior to the Minkowski sum. This is the case where one polygon can slide along a one-dimensional tight passage across the other polygon creating an edge of free space between two regions interior to the Minkowski sum; see Figure ??.

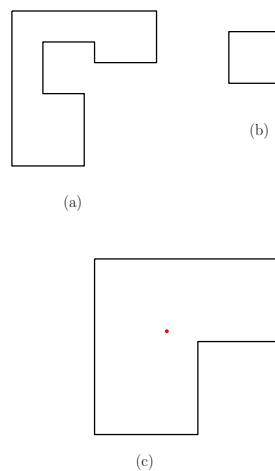


Figure 3.1: Illustration of a degenerate case. (a) and (b) are the two summand polygons. (c) is their Minkowski sum. The red dot is the isolated vertex.

Algorithm for Handling the Low Dimensional Boundaries

1. *dangling edge* - In order to find edges in the convolution that are degenerate (i.e., there are points inside the sum on both sides of the edge) we do the following: We check for each edge whether it lies at the overlap of two or more original convolution segments, whose normals are oriented in opposite directions. Then we perform an intersection test on a single point on the edge to determine whether the edge is on the boundary of the Minkowski sum.
2. *isolated vertex* - In order to find isolated vertices which are on the Minkowski sum boundary we locate all vertices which lie in the intersection of three or more original convolution segments, with three or more different normal directions. Then for each such vertex we perform a collision detection test.

We apply these methods before the step that traces orientable loops. The list of degenerate edges is kept and such edges are ignored during the stage of tracing orientable

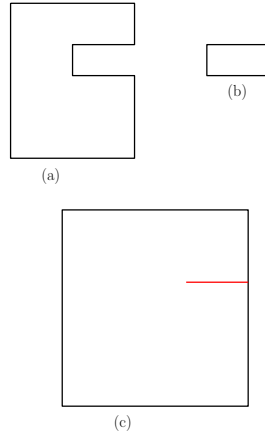


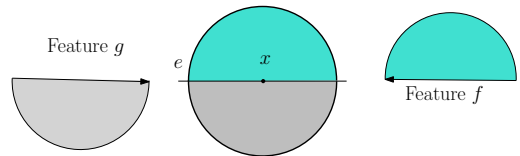
Figure 3.2: Illustration of a degenerate case. (a) and (b) are the two summand polygons. (c) is their Minkowski sum. The red segment is the dangling edge.

loops. Both the degenerate vertices and degenerate edges are kept in lists that can be output once the algorithm finishes. We next prove the correctness of these methods. We show that our filters are necessary conditions for the formation of a dangling edge or an isolated vertex. However, they are not sufficient therefore we need a collision detection test for the remaining vertices and edges.

Definition 3.2.7 (Dangling Edge). A dangling edge is an edge that is on the Minkowski sum boundary where its interior touches the Minkowski sum interior on both sides.

Theorem 3.2.8. *An dangling edge e exists only on the overlap of two or more convolution segments where at least two segments have opposite normals.*

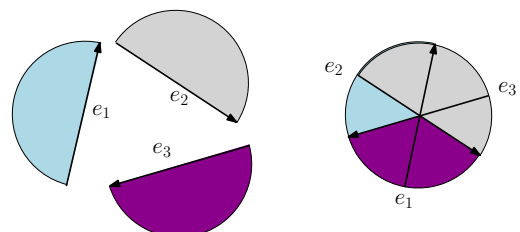
Proof. Let x be a point interior to e . By Theorem ?? and Theorem ??, e is in the (convex) convolution. Let us observe the neighborhood of x . It touches an area of the Minkowski sum on both sides. Each side is locally a half-disk. If we choose one of the sides, By Lemma ?? and Theorem ?? x is a sum of an edge and a vertex, and the edge has the same tangent to be a boundary for that half-disk (i.e., the edge is oriented such that the half-disk is to its right). The same is true for the other side. Therefore, x lies on the sum of two edges with opposing normals. \square



Definition 3.2.9 (Isolated Vertex). An isolated vertex is a vertex that is on the Minkowski sum boundary but has a neighborhood n_v that is entirely inside the Minkowski sum.

Theorem 3.2.10. *An isolated vertex exists only on points of intersection of three or more differently oriented segments of the convolution.*

Proof. Let v be an isolated vertex. Since there exists a neighborhood n_v around v , completely inside the Minkowski-sum interior except v , then v must be the sum of a vertices and edges of P and Q where



by Observation ?? each edge induces around v a half disk neighborhood interior to the Minkowski sum, such that the union of those neighborhoods cover n_v but does not contain v . Note that each boundary feature is either an intersection of two segments (vertex) or an edge. From Theorem ??, it is known that we require at least two segments with opposing normals to create a degenerate edge. It is also clear that if the segments do not have opposite orientations, no degeneracy can occur, as discussed in the proof of Theorem ?. Therefore at least three segments must be incident at a vertex for the creation of an isolated vertex. Another way to see it, is by noting that the minimal (by number of segments) hole loop is a clockwise oriented triangle. If we inflate one of the polygons continuously, the loop shrinks until at some point it is just a single vertex. Thus, this is the minimal number of intersecting features needed to induce a degenerate vertex. \square

4

The Relation between Convolutions and Minkowski Sums

The kinetic framework [?] presented the notion of planar polygonal tracings and their convolution. It provides a general framework for performing various operations on polygonal tracings. An important theorem that was stated in that paper is as follows: The winding number of a particular point x in the plane with respect to the convolution of two polygonal tracings P and Q , represents the number of connected components of the intersection of $P + (-Q)^x$. As a result, a new method to compute the Minkowski sum of two polygons P and Q is made possible. This is due to the fact that all points that have positive winding number with respect to the convolution of P and Q are interior to the Minkowski sum. Wein [?] used this property to implement a convolution-based algorithm, which computes the Minkowski sum of two simple polygons. However the basic facts and theorems that form the basis for this algorithm are given without proof in [?].

In this chapter we provide proofs for the theorems mentioned above for a simplified case, namely for simple polygons. In the first section it is shown that the set of convolution segments can be seen as a directed graph, which can be decomposed into a not necessarily unique set of disjoint cycles. In particular, this implies that it is possible to assign a winding number to each face of the planar subdivision induced by the convolution segments. In the second section it is proved that exactly those faces with positive winding number are indeed part of the Minkowski sum. We remind the reader that in the convolution Definition ??, we rotate Q infinitesimally such that no edge of P has the same direction of an edge of Q .

4.1 Extracting Convolution Cycles

Wein [?] describes an algorithm for tracing a convolution cycle starting from some vertex on this cycle. However, no proof that this cycle ever closes is given. After a formal definition of a cycle, in the first part of the section we describe the algorithm for tracing a cycle. In the second part we show that the algorithm correctly traces a cycle and that

every convolution segment is part of a cycle. Furthermore, we show that repeated invocations of the algorithm decompose the convolution segments into edge-disjoint cycles. Finally, we describe an algorithm which traces all the convolution cycles at optimal time $O(n + m + K)$, where K is the total number of convolution segments and m and n are the number of vertices of P and Q , respectively.

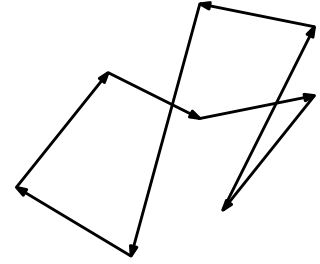
Notice that in the current discussion as opposed to Chapter ??, we refer to the original convolution segments without splitting them at their intersection points.

4.1.1 Algorithm for Tracing Convolution Cycles

We first formally define the convolution cycle:

Definition 4.1.1 (Convolution Cycle). A convolution cycle C is a closed sequence of convolution segments s_1, s_2, \dots, s_k , such that the target vertex of s_i coincides with the source vertex of $s_{(i \bmod k)+1}$.

This definition does not preclude self-intersections, as demonstrated in the figure to the right. Next we define the graph induced by the convolution segments:



Definition 4.1.2 (Convolution Graph). The convolution graph has the convolution segments as directed edges and their vertices as nodes. Vertices which coincide are considered the same node.

Note that this graph does not consider intersection points that are interior to a segment.

The general scheme for tracing convolution cycles is tracing them one by one. Algorithm ??, proposed by Wein [?], is used for tracing a single cycle in the convolution. It receives as an input a vertex on the boundary of the convolution. It then iteratively adds convolution segments to the cycle and stops when the cycle closes. The algorithm then returns the segments that belong to that cycle. We can then repeatedly use the algorithm to find all the cycles in the convolution.

Algorithm ?? keeps a pair of indices one for P and the other for Q . The indices i and j for polygon P and Q , respectively, are references to the polygon vertices p_i and q_j . For each polygon the vertices are ordered in a counterclockwise manner. Each pair of index values (i, j) is a state of the algorithm. There is a correspondence between a state (i, j) and the vertex induced by it $(p_i, q_j) = p_i + q_j$. We may refer to (i, j) either as a state of the algorithm or as a vertex (the induced vertex). The algorithm receives a vertex that is assumed to have unvisited incoming edge (incoming edge is required for the correctness proofs), which is in the convolution. In addition, the convolution graph G stores for each edge whether it was visited by some activation of the algorithm. The graph persists during the tracing of all convolution cycles.

Lines ?? – ?? contain the main loop of the algorithm. Each iteration adds a segment to the cycle, until a cycle is closed. The process at each iteration is as follows: Assume the algorithm is in state (i, j) which corresponds to the vertex v . The two possible successive states $(i + 1, j)$ and $(i, j + 1)$ correspond to two vertices w and w' , respectively. We later show that at least one of the two segments \overrightarrow{vw} , and $\overrightarrow{vw'}$, is in the convolution and possibly both (see Lemma ??). Lines ?? – ?? check which of those segments are in the convolution. If both segments are in the convolution, we just pick the unvisited edge by

inspecting the graph G . For the case that both are unvisited we choose one arbitrarily, specifically in this case we choose the edge from P . Once a segment is chosen it is added to the cycle and the algorithm moves to the state that induces the target vertex of the chosen segment (Lines ?? – ??, ?? – ??). The algorithm terminates when it reaches the starting state. In the next section we show why it is guaranteed to reach the starting state again. We remind the reader that CCW is the same predicate of Definition ??.

Algorithm 2 Compute Convolution Cycle($P, i_0; Q, j_0; G$)

Input: A state representing a vertex in the convolution.

Output: A cycle of the convolution.

```

1: Let  $C \leftarrow \emptyset$ 
2: Let  $i \leftarrow i_0$ . Let  $j \leftarrow j_0$ 
3: Let  $v \leftarrow (p_i + q_j)$ 
4: repeat
5:   Let  $inc\_p = CCW(\overrightarrow{p_i p_{i+1}}, \overrightarrow{q_{j-1} q_j}, \overrightarrow{q_j q_{j+1}})$ 
6:   Let  $inc\_q = CCW(\overrightarrow{q_j q_{j+1}}, \overrightarrow{p_{i-1} p_i}, \overrightarrow{p_i p_{i+1}})$ 
7:   Let  $w \leftarrow (p_{i+1}, q_j)$ 
8:   if  $inc\_p \wedge (\overrightarrow{vw}$  not visited) then
9:     Insert segment  $\overrightarrow{vw}$  to  $C$ 
10:    Let  $v \leftarrow w$ 
11:    Let  $i \leftarrow (i + 1) \bmod size(P)$ 
12:    continue
13:   else
14:     Let  $w \leftarrow (p_i, q_{j+1})$ 
15:     Insert segment  $\overrightarrow{vw}$  to  $C$ 
16:     Let  $v \leftarrow w$ 
17:     Let  $j \leftarrow (j + 1) \bmod size(Q)$ 
18:     continue
19:   end if
20: until  $i == i_0$  and  $j == j_0$ 
21: return  $C$ 

```

4.1.2 Correctness of Cycle Tracing

We now prove that the convolution cycles exist, and that Algorithm ?? traces them. In order to show that the algorithm is correct and the cycles exist, we need to prove two facts:

- For every state (i, j) that the algorithm reaches, which corresponds to a vertex v and the next two possible states $(i + 1, j)$ and $(i, j + 1)$, which correspond to the vertices w and w' , respectively, at least one of the segments \overrightarrow{vw} and $\overrightarrow{vw'}$ is in the convolution.
- The algorithm reaches its starting state (i_0, j_0) after a finite number of steps.

The following Lemma shows why the first fact is correct. But first we introduce some notation that will help formulating the lemma. We say that a state (i, j) is in the convolution if the corresponding vertex of this state, has an outgoing convolution segment.

A transition from state (i, j) to state $(i+1, j)$ or $(i, j+1)$ induces the convolution segment $\overrightarrow{p_i + q_j, p_{i+1} + q_j}$ or $\overrightarrow{p_i + q_j, p_i + q_{j+1}}$, respectively.

Lemma 4.1.3 (Forward Traversal). *While tracing a cycle, for every state (i, j) (with corresponding vertex v) the algorithm is in, the following holds:*

1. *The vertex corresponding to state (i, j) has an outgoing convolution segment.*
2. *The vertex v corresponding to the state (i, j) has two outgoing convolution segments, if and only if both states $(i-1, j)$, $(i, j-1)$ are in the convolution and have outgoing convolution segments to (i, j) .*

Proof. Each step of the algorithm is a result of a transition between two states representing two vertices of some convolution segment. We know that the convolution segment is the sum of a vertex and edge one from P and one from Q . Let us assume w.l.o.g. that the edge originates from P . By the assumption, the convolution segment is the sum of $\vec{p}_{in} = \overrightarrow{p_{i-1}, p_i}$ and q_j . Therefore, the state $(i-1, j)$ is in the convolution.

Since the sum of \vec{p}_{in} and q_j results in a convolution segment, by definition the direction of \vec{p}_{in} is in the tangent set of q_j . Consequently, it holds for the directions of: \vec{p}_{in} , $\vec{q}_{in} = \overrightarrow{q_{j-1}, q_j}$ and $\vec{q}_{out} = \overrightarrow{q_j, q_{j+1}}$, that $CCW(\vec{p}_{in}, \vec{q}_{in}, \vec{q}_{out}) = true$, as seen in the figure to the right. Let $\vec{p}_{out} = \overrightarrow{p_i, p_{i+1}}$. Note that q_{in} , p_{in} and q_{out} partition the unit circle into three regions where p_{out} can be placed. Moving p_{out} within each of those regions does not alter the value of any CCW predicate operating on any choice of three out of the four edges. In the figure to the right we observe the three possible positions for \vec{p}_{out} , as depicted in Figure ??:

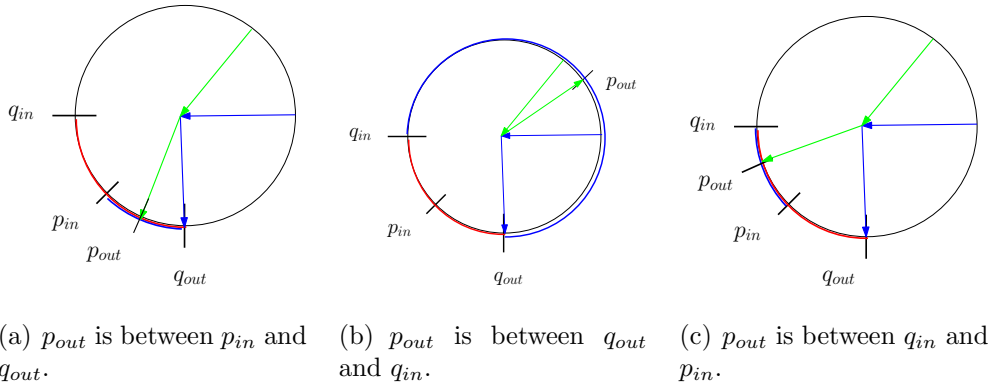
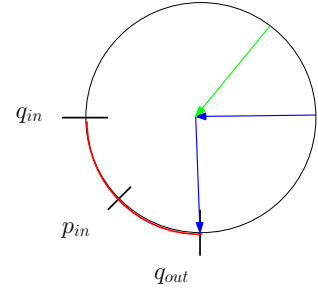


Figure 4.1: Proof of Lemma ??.

- ??: The direction of \vec{p}_{out} is such that $CCW(\vec{p}_{out}, \vec{p}_{in}, \vec{q}_{out}) = true$. Thus, it immediately follows that $CCW(\vec{p}_{out}, \vec{q}_{in}, \vec{q}_{out}) = true$ therefore, $\vec{p}_{out} + q_j$ is in the convolution.
- ??: The direction of \vec{p}_{out} is such that $CCW(\vec{p}_{out}, \vec{q}_{out}, \vec{q}_{in}) = true$. In this case it follows that $CCW(\vec{q}_{out}, \vec{p}_{in}, \vec{p}_{out}) = true$. Consequently, $\vec{q}_{out} + p_i$ is in the convolution.
- ??: The direction of \vec{p}_{out} is such that $CCW(\vec{p}_{out}, \vec{q}_{in}, \vec{p}_{in}) = true$. This imposes that both $CCW(\vec{q}_{out}, \vec{p}_{in}, \vec{p}_{out}) = true$ and $CCW(\vec{p}_{out}, \vec{q}_{in}, \vec{q}_{out}) = true$. Therefore, both $\vec{p}_{out} + q_j$ and $\vec{q}_{out} + p_i$ are in the convolution. Note that the directions also satisfy $CCW(\vec{q}_{in}, \vec{p}_{in}, \vec{p}_{out}) = true$ implying that the segment $p_i + \vec{q}_{in}$ is in the convolution.

Consequently the state $(i, j - 1)$ is in the convolution since $p_i + \vec{q}_{in}$ is outgoing from it.

For all three cases there is at least one outgoing segment in the convolution implying that the state (i, j) is in the convolution as required by the first part of the Lemma. The third case is the only case where we have two incoming convolution segments into (i, j) , and the two states $(i - 1, j)$, $(i, j - 1)$ are in the convolution. In this case there are also two outgoing convolution segments. This verifies the second part of the lemma. \square

From Lemma ?? we can conclude that for every new state that the algorithm reaches this state is also in the convolution. As a result, while tracing a cycle the algorithm always has a subsequent state to move to, without getting “stuck”. The rest of this section shows why the sequence of moves as dictated by the algorithm actually closes a cycle.

Using Lemma ?? we now show that the in-degree and out-degree of any vertex in the convolution graph are equal:

Lemma 4.1.4 (In-Out Degree Equality). *Let G be a convolution graph. For any vertex v , $Indegree(v) = Outdegree(v)$.*

Proof. Let v be a vertex in G . Since the convolution graph consists of vertices of convolution segments, we know there is a finite set S of states of Algorithm ?? which corresponds to the vertex v . By Lemma ?? for each of these states the number of incoming and outgoing segments is equal. Therefore, $Indegree(v) = Outdegree(v)$. \square

This property enables us to show that the convolution graph can be decomposed into edge disjoint directed cycles:

Lemma 4.1.5 (Convolution Graph Decomposition). *The convolution graph can be decomposed into edge disjoint directed cycles.*

Proof. Since for each vertex of the convolution graph the in-degree equals the out-degree, for every connected-component in the graph by Euler’s theorem there is an Euler circuit. Removing this circuit maintains the equality of the in-degree and out-degree for each vertex in the graph. Hence, we may iteratively extract cycles until all edges are removed. Equivalently, the component is decomposable into edge-disjoint cycles. \square

Note that this decomposition is not unique and depends on choice of circuits to be extracted in each iteration.

Now we just need to see that Algorithm ?? traces a cycle:

Theorem 4.1.6 (Algorithm ?? Correctness). *If the initial state (i_0, j_0) is in the convolution and has an outgoing unvisited edge, the algorithm traces a cycle and returns to the state (i_0, j_0) after a finite number of steps.*

Proof. By Lemma ?? we know that (i_0, j_0) belongs to some directed cycle. By Lemma ??, we know that the algorithm does not get stuck. Since the state set is finite, the algorithm must eventually reach (i_0, j_0) again. \square

We note that the algorithm may encounter a sub-cycle (i.e. have some state (i'_0, j'_0) which is traversed twice) in which case it is returned as part of the bigger cycle.

Tracing Convolution Cycles

Now we need to describe how to use Algorithm ?? for finding all the cycles. We assume that P and Q have m and n vertices, respectively. The convolution graph contains the states of the algorithm (vertices) which have unvisited outgoing convolution segments. We could try every possible state in the algorithm as a starting point and check whether the outgoing edges are in the convolution. However this approach generates $2mn$ starting points, as each state can transition to two successive states. We will now show that only $n + m$ starting points need to be examined.

In order to show this we define a tool called the fiber grid:

Definition 4.1.7 (Fiber grid). A fiber grid for polygons P and Q , with m and n vertices respectively, is a periodic grid where each node (i, j) represents being at state (i, j) in the algorithm. A directed edge from a state to the next one will mark that there exists the corresponding edge in the convolution and that this edge is a valid traversal in the algorithm. Note that the only possible outgoing edges from (i, j) are $(i + 1, j)$ and $(i, j + 1)$ since these are the steps allowed in each iteration of the algorithm. From Lemma ??, at least one of them can be traversed by the algorithm, thus in the grid. Similarly, for any state in the convolution, there is at least one incoming edge from state $(i - 1, j)$ or $(i, j - 1)$. The grid is cyclic, the first row is identified with the last, and the first column is identified with the last column. For a simple example for two convex polygons see Figure ??.

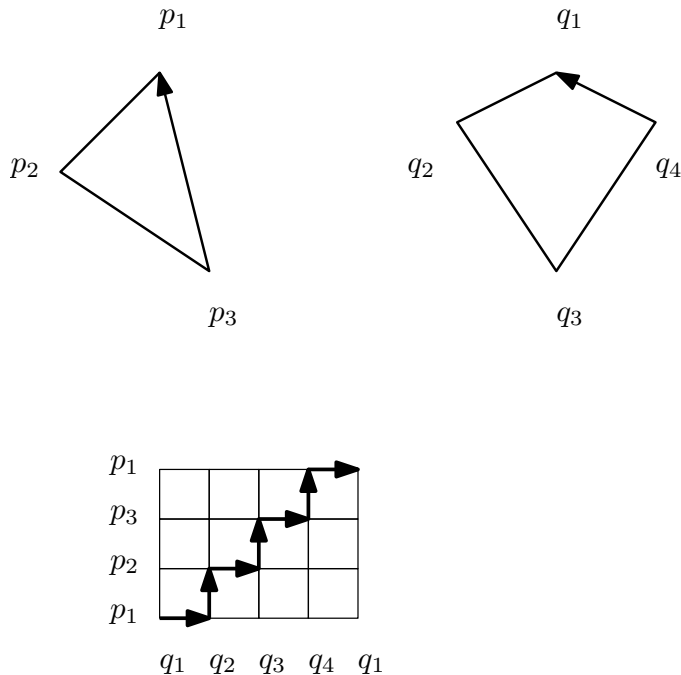


Figure 4.2: Example of fiber grid of two convex polygons P, Q .

Lemma 4.1.8. For every grid row there is at least one state which is the starting vertex of an outgoing vertical edge. Similarly, For every grid column there is at least one state which is the starting vertex of an outgoing horizontal edge.

Proof. Traversing a row in the fiber grid visits all the vertices of the polygon P and matches them with some edge e_q of the polygon Q . However e_q must be in the tangent set of some vertex of P and thus in the convolution, since the union of the tangent spaces

of a polygon covers the unit disk. Therefore, an edge of Q connects the current row to the next one on the fiber grid.

The same holds for traversing a column of a fiber grid which visits all the vertices of Q and matches them with some edge of P . □

The reader should observe that a cycle traverses at least an entire row or an entire column since this is the shortest way to return to the same state twice in order to close a cycle. Thus, every cycle crosses either all of the rows or all of the columns of the fiber grid. Furthermore, we can see the following:

Corollary 4.1.9. *There is at least one cycle of size $m + n$.*

Proof. We have at least one convolution cycle, since a convolution segment can always be found (by taking the minimal points of P and Q as a starting vertex). If the cycle spans a row of size n for example, using Lemma ?? we know that there is at least one vertical edge. To close this cycle the algorithm would have to return to the same row, taking m additional steps. By concatenation we get a cycle of size $m + n$. □

Since each cycle crosses either all of the rows or all of the columns, it is sufficient to start Algorithm ?? at any unvisited out-going convolution edges, starting at vertices of one single row and one single column in the grid. Since all cycles pass through either a row or a column in the grid, we are guaranteed to visit all cycles.

Let us compute the runtime complexity of tracing the convolution cycles using this method. By Corollary ?? there is at least a cycle of size $m + n$. Denote the total size of the convolution cycles, by K . By starting from each starting segment mentioned above, we perform at most $2(m + n)$ invocations of Algorithm ?. Therefore, the total complexity of this method is $O(n + m + K)$. This is the optimal complexity because it is output sensitive, and there is at least one cycle of size $m + n$. We can thus conclude the following:

Theorem 4.1.10. *The complexity of the cycles tracing algorithm is $O(n + m + K)$, which is optimal.*

4.2 The Minkowski-Sum-by-Convolution Theorem

In their paper about the kinetic framework, Guibas et al presented a theorem named the “convolution theorem” [?]. A result from this theorem is that the Minkowski sum is the union of regions in \mathbb{R}^2 where all points interior to a region have positive winding numbers with respect to the convolution. However, to the best of our knowledge no proof for this theorem has been published. In this section we present a proof for a simplified version of this result, i.e., for simple polygons:

Theorem 4.2.1 (The Relation between Minkowski Sum and Convolution). *Let P, Q be two simple polygons. Let $C = P \otimes Q$. Let $\alpha \in \mathbb{R}^2, \alpha \notin C$. Then $wn(\alpha) > 0$ iff $P \cap (-Q^\alpha) \neq \emptyset$.*

As Wein implemented a Minkowski-sum convolution algorithm in CGAL [?] for two simple polygons, this section confirms the correctness of Wein’s implementation.

We begin this section by defining the winding numbers and discussing some of their properties. The second part of this section discusses the events that occur while sweeping

$-Q$ along a path relative to P . Note that for each point α we have a well defined winding number with respect to the convolution segments. If we define α as the translation point of $-Q$, we see that the winding number of α is related to the topology of $P \cap -Q^\alpha$. Apparently, if we were to choose a slightly different set of convolution segments, the winding number will be exactly the number of connected components of $P \cap -Q^\alpha$. In the final part we show that the winding number of α with respect to the convolution segments is an upper bound for the number of connected components of $P \cap -Q^\alpha$, and is zero whenever the number of connected components is zero. This proves Theorem ??.

4.2.1 Winding Numbers

The winding number concept used in many fields of mathematics describes a simple idea. If we stand at a point p and observe a point that travels along a closed curve from the beginning to end, the winding number will count how many times we completed a counterclockwise turn, minus the number of times we completed a clockwise turn. If p is not on the curve, the winding number must be an integer.

Formal Definition

A curve in the plane can be defined by parametric equations, $x = X(t), y = Y(t), t \in [0, 1]$. We demand that $X(t)$ and $Y(t)$ are continuous in t . Since we deal with closed curves (cycles), $X(0) = X(1), Y(0) = Y(1)$. We can convert the curve to polar coordinate system, by translating the observation point p which is not on the curve to the origin. The parametrization becomes $R(t), \theta(t)$.

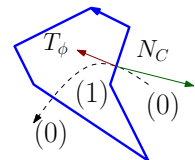
Definition 4.2.2 (Winding number). The winding number for p will be:

$$wn(p) = \frac{\theta(1) - \theta(0)}{2\pi}. \quad (4.1)$$

Another important fact for winding numbers known from complex analysis [?] is the following:

Definition 4.2.3 (Crossing rule). Let C be a closed curve. Let α be a point. If we translate α in some continuous motion, and it crosses C , the winding number is changed by the following amount. Assume the movement is never tangent to C . Therefore, at the crossing point, the dot product of the normal to C and the tangent to the motion is not zero. We take the sign function (who's value is signed one) of this dot product negated as the change in the winding number after the crossing.

In the figure to the right we see an example of a blue curve C and a dashed path. The numbers in parenthesis are the winding numbers along the path. When the path crosses into C , the red arrow denotes the tangent to the path ϕ while the green arrow is the normal to C . At that crossing point the tangent is in opposite direction to the normal which means that the dot product is negative. By taking the sign and negating the result we get an increase of the winding number by one.



We use the crossing rule to define a crossing number for a path. This crossing number is the difference in winding number from the starting point of the path to the target point.

Definition 4.2.4 (Signed crossing number for path). Let C be a closed curve in the plane and α and β two points in the plane. Let ψ be a simple curve from α to β . We assume general position such that ψ is never tangent to C . Furthermore, α and β are not on C . The signed crossing number $SC(\psi)$ of ψ is $SC(\psi) = wn(\beta) - wn(\alpha)$.

We observe that we can compute the crossing number for a path using the crossing rule:

Observation 4.2.5. *We define the multi-set of points*

$$I_\psi = \{C(t') \in \mathbb{R}^2 \mid \exists t \in \mathbb{R}, t' \in \mathbb{R} \text{ s.t. } \psi(t) = C(t')\}$$

as the points in which ψ crosses C . Note that C might overlap itself, thus some points may appear with multiplicity. The sum of the change in the winding number for all the points in I_ψ is exactly $SC(\psi)$.

Note that for the winding numbers to be well defined the signed crossing number for every cyclic path must be zero.

4.2.2 Sweep Events Analysis

From now on C is a piecewise linear curve as we deal with polygons and segments. Let $\alpha \in \mathbb{R}^2, \alpha \notin C$. Let r be the horizontal ray from α to $+\infty$. We now assume general position in the following sense:

Assumption 4.2.6 (General Position).

- *The world coordinate system is rotated such that r does not intersect any of C 's vertices.*
- *We treat the directions of Q as being infinitesimally rotated counterclockwise such that none of them equals any direction of P .*

Each point on r represents a translation for $-Q$ by that point. Intuitively, traversing this ray from plus infinity to α is interpreted as translating polygon $-Q$ horizontally, away from plus infinity until its translation point equals α . Assume we keep P fixed at its original position. Along this sweep we are interested in events that change the topology of connected components of the intersection of P and the translated $-Q$. More specifically, we are interested to find out when the number of connected components of the intersection changes. By tracking these changes we note the time (translation point) during sweeping of $-Q$ that intersection of components appear or disappear. From this we can conclude the times the intersection of P and $-Q$ is empty and the times it is not. Let us impose the additional general position requirement that while sweeping $-Q$ there is only one intersection event between P and the translated $-Q$ at a time.

We define vertex-edge intersection event as:

Definition 4.2.7 (Vertex-Edge Intersection Event). Is an event when a vertex of P intersects an edge of $-Q$ or vice versa. The event can be associated with a translation point α which specifies the translation of $-Q$.

We now show that specific types of vertex and edge intersections occur when the ray intersects a convolution segment of C . A coincidence of a vertex and an edge could change the number of intersecting connected components between $-Q$ and P . We will show below that the direction of a convolution segment (which is pointing upwards or downwards) is related to this increase or decrease in the number of intersecting components. Every such crossing promptly increases or decreases the winding number of the current point on the ray relative to the previous translation.

First we examine how points on convolution segments relate to events which occur while sweeping $-Q$. It appears to be that points on the convolution segments represent translations where P and $-Q$ have an edge and vertex intersection event:

Observation 4.2.8 (Convolution Segment Event). *Let α be a point in the interior of a convolution segment $c \in C$. Then it follows that P and $-Q^\alpha$ have a vertex and an edge intersection.*

Proof. Let p_i and q_j be vertices of P and Q , respectively. By Definition ??, we know that convolution segments are the sum of a vertex from P and an edge from Q or vice versa. By linear interpolation any point, specifically α , on a convolution segment satisfies at least one of the following two equations, for some $\lambda \in [0, 1]$:

$$\alpha = \lambda q_{j+1} + (1 - \lambda)q_j + p_i, p_i \in P, q_j, q_{j+1} \in Q. \quad (4.2)$$

$$\alpha = \lambda p_{i+1} + (1 - \lambda)p_i + q_j, q_j \in Q, p_i, p_{i+1} \in P. \quad (4.3)$$

It is possible to rewrite these equations as :

$$\alpha - (\lambda q_{j+1} + (1 - \lambda)q_j) = p_i, \quad (4.4)$$

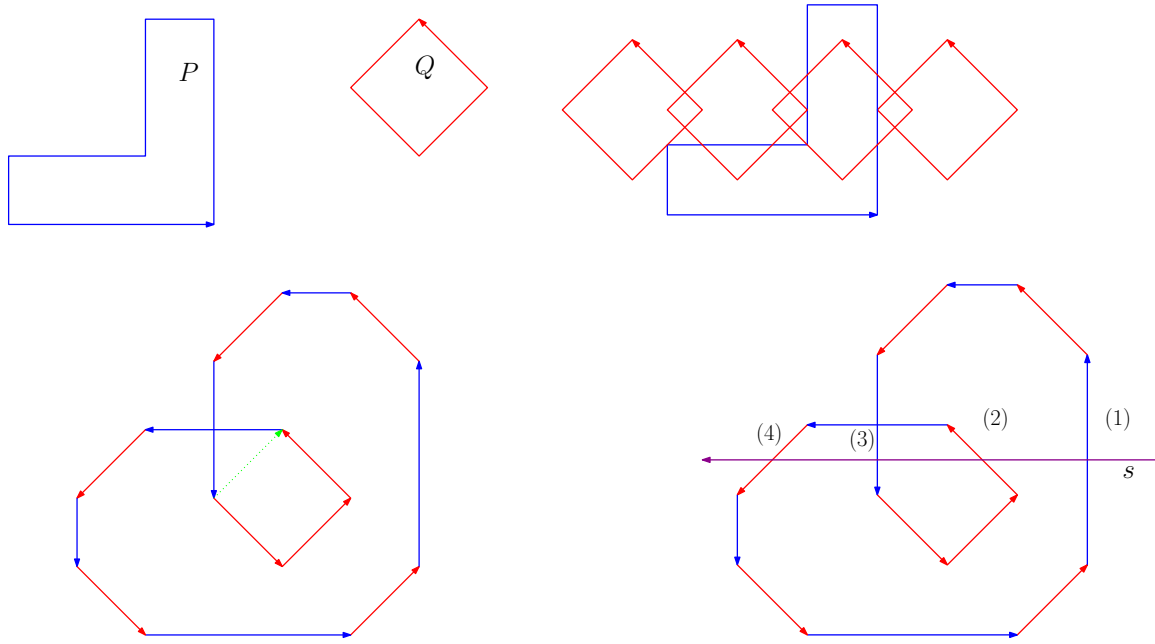
$$\alpha - q_j = \lambda p_{i+1} + (1 - \lambda)p_i. \quad (4.5)$$

Thus, from Equation ?? we get that, if we denote by e the edge obtained by reflecting the segment $\overrightarrow{q_j, q_{j+1}}$ through the origin and translating it by α , then at some point along e lies p_i . In the same manner, from Equation ??, if we reflect q_j through the origin and translate it by α , it will reside on the segment $\overrightarrow{p_i, p_{i+1}}$. But these two cases are exactly the cases where vertex-edge intersection event occurs, at point α . Therefore, points that lie on the convolution represent translations of $-Q$ where a vertex-edge intersection event occurs. \square

Let us more formally describe the sweeping and the events that are of interest during the sweep. While performing the sweep P remains fixed and $-Q$ sweeps along a horizontal line. At the beginning of the sweep $-Q$ is translated to be apart from P such that there are no intersecting components between them. Furthermore, we can choose the starting point of the translation of $-Q$ to be outside of the bounding box of the convolution segments C , thus the winding number of the initial translation point with respect to C is zero. Let s be the (horizontal) segment from this starting point to the point α . Figure ?? shows the sweeping process. In this case we have a symmetrical polygon swept along a segment. The intersection events along this segment are illustrated.

We now examine the events that may occur while sweeping $-Q$ with the translations defined by the points on s . For exploring this, we require the following definitions.

Definition 4.2.9 (Connected Components Counter). For polygons P, Q and a point α let $CCC(\alpha)$ denote the number of connected components of $(-Q^\alpha) \cap P$.



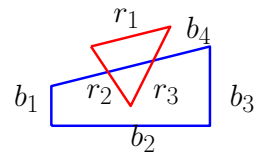
(a) Two polygons (top) and their convolution cycle(bottom).

(b) Example of shooting a ray and the translation it induces on the reflected polygon (in red). The intersection of the ray with the convolution segments each correspond to one of the translations of $-Q$ depicted at the top.

Figure 4.3: The original scenario ?? and the ray-shooting events ??.

Definition 4.2.10 (Active Feature Set). For a connected component $cc(\alpha)$ of $(-Q^\alpha) \cap P$ let $AC_{cc(\alpha)}$ be the set of all the features from P and $-Q$, that lie on the boundary of $cc(\alpha)$.

Refer to the figure to the right. There is a single intersection component between the red and the blue polygons. The active feature set of this component is the set $\{r_2, r_3, b_4\}$, as these are the edges that touch the component.



Now we can define the following events which affect the set of active feature sets:

- A new connected component appears. This means that a new active feature set is created.
- An existing connected component disappears. This means that an active feature set is removed.
- The border of an existing connected component topologically changes. This means that an existing active features set undergoes a change in its members.

By the general position assumption made so far (Assumption ??), during the sweep there are only vertex and edge intersection events (contrary to a vertex of P intersecting a vertex of $-Q$). These are the only events that could cause one of the events described above to occur. By our assumptions only a single event occurs at a time, and this set of events is finite. By Observation ??, some of these events occur when s intersects segments of the convolution. Each such intersection changes the winding number according to the direction of the segment.

Let us analyze the events of intersections of a vertex and an edge from the two polygons during the traversal of s . The goal is to determine which events cause s to intersect segments of the convolution and which do not. Answering this question reveals the relationships between winding numbers, connected components numbers and the convolution. We divide the events into two sets, event points that are on the intersection of s with a segment in the convolution and events that are not. We show that the former events are either events where the number of connected components changes, or they can be “charged” to such events. Finally we show that there is a correlation between a change in the number of connected components and a change in the winding number count at the point where the event occurs.

Events Analysis

We now analyze the different cases that arise when there is a vertex and edge intersection event while sweeping $-Q$. We assume that the sweeping of $-Q$ starts at some far point to the right and proceeds towards the left (i.e. from $+\infty$ to $-\infty$). While examining the events we would like to consider the changes in the number of connected components and the winding number before versus after the event. A change in the winding number occurs when the event is on a convolution segment. The direction of the convolution segment indicates whether the winding number is increased or decreased by one. A change in the number of connected components is determined by the kind of connected component event. Only if a connected component is created, vanishes, splits or is merged, this number changes. We can partition the set of all vertex–edge intersection events into the following subsets, whose union composes the entire set of events:

Vertex/Edge Origin	$P/-Q$	$-Q/P$
Edge direction	Upwards	Downwards
On convolution segment	Yes	No
Vertex type	Convex	Reflex

Table 4.1: Case parameters

The purpose of the current section is, for each of the $2^4 = 16$ combinations that appear in Table ?? to compute the following possible result arguments, as described in Table ??.

Convolution segment direction	Upwards	Downwards	*
Winding–number difference	+1	-1	0
Connected components counter difference	+1	-1	0

Table 4.2: Output parameters

We discern some symmetries in the scenarios of the input arguments of Table ?. First we discuss the symmetry based on the polygon contributing the vertex to the event. Each scenario where the vertex is originated in P is similar to a scenario where the vertex is originated from $-Q$. This role reversal flips the direction of the convolution segment if the event occurs on such segment, but does not change whether the event is on the convolution or not. Since the roles of P and $-Q$ features are now flipped, the direction of

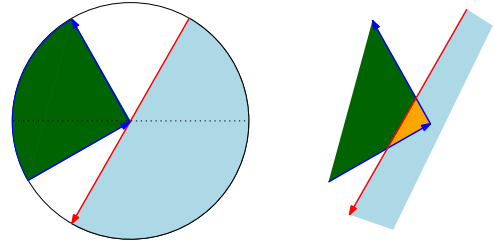
the sweep is also reversed (now $-Q$ is translated to the right) and thus the corresponding change in the intersection number has an opposite sign, namely, every appearance event is now a disappearance event, every split is now merge, and vice versa. Reversing the direction of the convolution segment also means that the change in the winding number has the opposite sign.

The second symmetry we notice is in the edge direction. By rotating the world by 180° and reversing the direction of the sweep, we get the following symmetrical scenario. The property of being on a convolution segment for the event is preserved. Reversing the direction of the sweep means negating the change in the connected-components counter. Additionally, since the edge now has an opposite direction, so is the direction of the convolution segment and thus the change in the winding number. Thus, for the rest of current section, we assume that the vertex p_i originates from P and the downwards directed edge e_Q is from $-Q$, w.l.o.g..

By using both symmetries we only need to examine the effect of the two remaining parameters, and deduce the effects for the entire 16 original combinations. So by reducing to only two parameters, we get four cases to examine.

For the following figures we have a blue polygon P and a red polygon $-Q$. The blue polygon induces the vertex v and the red one the edge e . The green area is interior to the blue polygon. The cyan area is interior to the red polygon. The orange area belongs to the intersection of the red and the blue polygons. The left figure shows the state at the intersection event. The right figure shows the neighborhood before/after a small translation of the polygons relative to each other. For a downwards red edge and a blue vertex there are four configurations. These cases are defined by the ordering of the incoming and outgoing edges of the vertex, relative to the downward edge, according to their angle with the positive x -axis. This ordering defines the tangent space of the vertex participating in the event relative to the edge. Therefore it may affect whether the event occurs on a convolution segment.

Convex Vertex/Convolution Segment The convolution segment this event is on is facing upwards, since the edge originates from $-Q$. In the figure it is easy to see that just after the event, a new component of the intersection of P and $-Q$ appears. Therefore this event increases the connected-components counter by one. Crossing an upwards convolution segment also increases the winding number by one. By using the two symmetries defined earlier we can deduce the following cases:

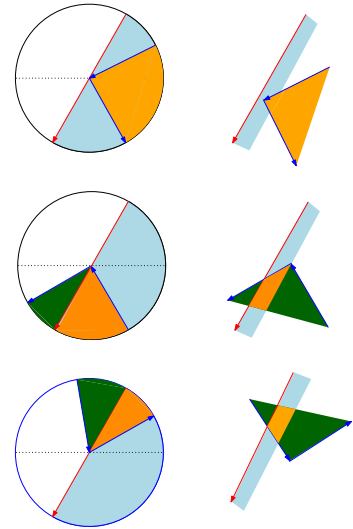


Vertex/ Edge Origin	Edge direction	On con- volution segment	Vertex type	Convolution segment direction	Δ w.n.	Δ CCC
$P/-Q$	Downwards	Yes	Convex	Upwards	+1	+1
$-Q/P$	Downwards	Yes	Convex	Downwards	-1	-1
$P/-Q$	Upwards	Yes	Convex	Downwards	-1	-1
$-Q/P$	Upwards	Yes	Convex	Upwards	+1	+1

Table 4.3: Convex vertex/convolution segment

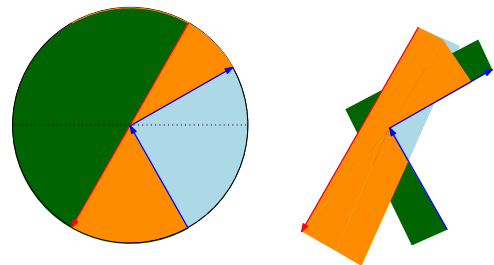
From Table ?? we can see that whenever the winding number increases or decreases so does the connected-components counter accordingly.

Convex Vertex/non-Convolution Segment There are three possible topologies for this case as shown in the diagram to the right. For all of these cases, the analysis and result are similar. Note that before and at the time of the event there is one connected component in a small neighborhood of the intersection event, at the place of the intersection between the vertex and the edge. It is easy to see that just after the event, the border of the connected component of the intersection of P and $-Q$ is changed (AC_{cc} changes). However, no connected components are created or removed, thus the connected-components counter remains the same. Also the winding number remains fixed since no convolution segment is crossed. This is true for all four symmetries of this scenario.



Reflex Vertex/non-Convolution Segment

This event does not occur on a convolution segment. Since the range of directions where this situation occurs is smaller than π and the vertex is reflex, there is only one configuration for this case as shown to the right. Note that before and at the time of the event there are two connected components in a small neighborhood of the intersection event, at the place of the intersection between the



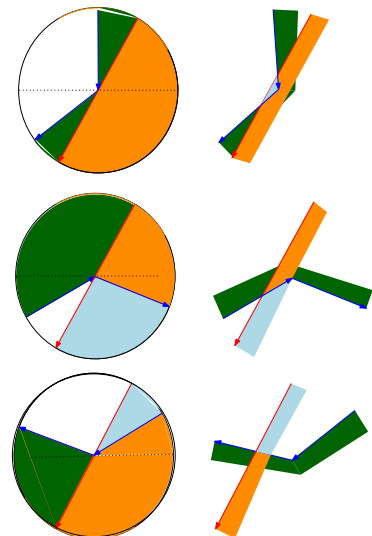
vertex and the edge. It is easy to see that just after the event, the two components have become connected. This decreases the connected-components counter by one. By using the two symmetries defined earlier we can deduce the following cases:

Vertex/ Edge Origin	Edge direction	On con- volution segment	Vertex type	Convolution segment direction	Δ w.n.	Δ CCC
$P/-Q$	Downwards	No	Reflex	*	0	-1
$-Q/P$	Downwards	No	Reflex	*	0	+1
$P/-Q$	Upwards	No	Reflex	*	0	+1
$-Q/P$	Upwards	No	Reflex	*	0	-1

Table 4.4: Reflex vertex/non-convolution segment

From Table ?? we can see that we have a mismatch between the winding number and the connected components counter.

Reflex Vertex/Convolution Segment This event occurs on a convolution segment and increases the winding number. Here we have the three complementary (with respect to the Reflex vertex/non-convolution segment event) cases for the topological order of the edges, as shown in the figure to the right. Again we see that the three cases have similar analysis. Note that before and at the time of the event there is a connected component in a small neighborhood of the intersection event, at the place of the intersection between the vertex and the edge. Just after the event, the border of the connected component of the intersection of P and $-Q$ is changed (AC_{cc} changes); see the figure. However, no connected components appear or disappear, thus the connected-components counter remains the same. By using the two symmetries defined earlier we can deduce the following cases:



Vertex/ Edge Origin	Edge direction	On con- volution segment	Vertex type	Convolution segment direction	Δ w.n.	Δ CCC
$P/-Q$	Downwards	Yes	Reflex	Upwards	+1	0
$-Q/P$	Downwards	Yes	Reflex	Downwards	-1	0
$P/-Q$	Upwards	Yes	Reflex	Downwards	-1	0
$-Q/P$	Upwards	Yes	Reflex	Upwards	+1	0

Table 4.5: Reflex vertex/convolution segment

Again we have a mismatch between the winding number and the connected-components counter; see Table ??.

4.2.3 Winding Numbers and Number of Connected-Components Relation

We now show how to relate the change in the connected-components counter and the winding number when encountering events with a reflex vertex. First we see that we may choose a slightly different set of segments for the convolution, for which the winding number and the connected-components counter exactly match. Note that this is the original set of convolution segments as defined in [?] unlike Wein’s definition which we have used so far. The difference is that in this set of convolution segments, edges that are added to a reflex vertex are complementary to the tangent set of a reflex vertex defined in Definition ?? and are summed with their direction flipped.

We then show that it is possible to transform the set of convolution cycles into the new set by subtracting Q as a cycle from every reflex vertex p_i (we define this subtraction formally as a flip set next). This subtraction is as follows; take $Q + p_i$ as a graph embedded in the plane. By subtracting it we mean taking each edge with reversed orientation and “add” it to the existing graph of convolution cycles. Edges that overlap but with reversed directions are canceled out. Those who intersect but have the same direction increase their multiplicity in the overlapping set. By Definition ?? the winding number for points in the region inside $Q + p_i$ is decreased by one. We show that this subtraction still maintains the directed graph of segments decomposable into cycles.

Now, if begin with the set of segments defined in [?] (where the winding numbers match the connected-components counter) and add Q as a cycle for every reflex vertex, we just increase the winding number in certain regions. The resulting set will be our convolution set in which the winding numbers over count the connected-components counter. However, we show that this occurs only for regions interior to the Minkowski sum to begin with. Therefore, the winding numbers for our convolution set are positive only in points interior to the Minkowski sum. These claims leads to the assertion of Theorem ??.

Let p_i be a reflex vertex (say from P). Examine the set of segments $p_i + e_Q$ where e_Q is an edge of Q . These segments are exactly Q translated by p_i . According to Observation ??, none of those segments are on the Minkowski sum boundary. That is they are all interior to the convolution cycles.

Definition 4.2.11 (Reflex Vertex Segment Sets). Let $S(p_i)$ be the set of segments $\{p_i + e_Q | e_Q \in Q\}$. Let us split these segments into two disjoint sets, the segments that are in the convolution, denoted by $S_c(p_i)$ and the segments that are not in the convolution, denoted by $S_{nc}(p_i)$.

Definition 4.2.12 (Flip Set Directions). For a set of segments S we denote by \widehat{S} the set containing the same segments as S , but with flipped directions.

Let $Q^{p_i} = p_i + Q$. For a given reflex vertex p_i we claim that for every point α inside Q^{p_i} , the winding number induced by the convolution segments, is at least as high as the connected-components counter. As seen in the analysis of the Reflex vertex/non-convolution segment case, events that involve a reflex vertex indicate a split or a merge of components of the intersection of P and $-Q$. These events occur at exactly these segments that are not convolution segments, i.e., $S_{nc}(p_i)$. Since the convolution contains the set of segments $S_c(p_i)$, we have the “wrong” set of segments which does not have the events that change the connected-components counter occurring on its points. But, if we were to replace the set of segments $S_c(p_i)$ with the set $S_{nc}(p_i)$, we have a set of

segments, every point on which represents the event where the connected-components counter changes. Let us consider the effect of each new segment on the connected-components counter. We examine the first row of Table ???. When the vertex is from P and the edge from $-Q$ is directed downward, this means that the original edge is upward. However, the connected-components counter is decreased by one, so we would like the winding number, while crossing this edge, to decrease. In order to achieve this, we flip the segment. By using the two symmetries, we notice that we should flip all edges of the set $S_{nc}(p_i)$. Now for the flipped non-convolution segments we get the following revised table:

Vertex/ Edge Origin	Edge direction	Vertex type	New con- volution segment direction	Δ w.n	Δ CCC
$P/-Q$	Downwards	Reflex	Downwards	-1	-1
$-Q/P$	Downwards	Reflex	Upwards	+1	+1
$P/-Q$	Upwards	Reflex	Upwards	+1	+1
$-Q/P$	Upwards	Reflex	Downwards	-1	-1

Table 4.6: Reflex vertex/flipped convolution segment

Consider Figure ??. This figure shows how the sweep process along the ray encounters different events from both sets of segments. The example scene illustrates the ‘‘charging’’ of events.

Let C' be the set of segments that is created by the following process. Take the set C of convolution segments, iterate over all reflex vertices p_i (w.l.o.g. from P), and replace the set $S_c(p_i)$ with $\widehat{S_{nc}(p_i)}$. By the case analysis we performed, the winding numbers of C' for each point that is not on C' or C , equals the number of connected components. This holds only if the assumption that C' is still composed of cycles is correct. Equivalently, the winding numbers with respect to C' are well defined.

Consequently, in order to complete the proof of the theorem, we need to prove the following claims:

- The new set of segments $C \cup \widehat{S_{nc}(p_i)} \setminus S_c(p_i)$ is composed of cycles.
- The original winding numbers for all points in $\mathbb{R}^2 \setminus C$ with respect to C are greater or equal to the winding numbers for all points in $\mathbb{R}^2 \setminus (C \cup \widehat{S_{nc}(p_i)} \setminus S_c(p_i))$ with respect to $C \cup \widehat{S_{nc}(p_i)} \setminus S_c(p_i)$ for all p_i .
- The winding numbers of areas outside the Minkowski sum are zero before and after the replacement.

Let us prove that for any reflex vertex p_i , by replacing the segments of $S_c(p_i)$ with $\widehat{S_{nc}(p_i)}$ the new set of segments $C \cup \widehat{S_{nc}(p_i)} \setminus S_c(p_i)$ is still composed of cycles. We show this by examining the winding number at all the points α that are not on any of the segments of either C , $S_c(p_i)$ or $\widehat{S_{nc}(p_i)}$. It suffices to show that the winding numbers for all those points are well-defined. The term well-defined here means the following. Fix α , for any β and β' (not in C , $S_c(p_i)$ or $\widehat{S_{nc}(p_i)}$), let γ and γ' be the paths from points

β and β' to α , respectively. Then by Definition ??, $wn(\beta) + SC(\gamma) = wn(\beta') + SC(\gamma')$. Alternatively, we require that every simple cycle that begins and ends with α has zero crossing number.

For the following illustrations, the blue segments represent edges of P , red edges are convolution edges of $S_c(p_i)$, and the green edges are of the set $\widehat{S_{nc}(p_i)}$.

Lemma 4.2.13 (Winding number consistency). *For every point not on any convolution segment, nor on replaced segments, the winding number is well defined.*

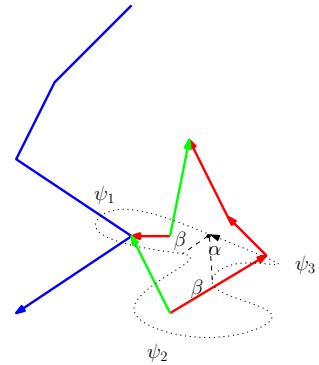
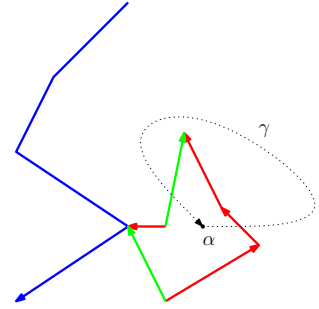
Proof. Let p_i be a reflex vertex of P . We consider what happens at $\alpha \in p_i + Q$ while replacing the set $S_c(p_i)$ with $\widehat{S_{nc}(p_i)}$. Let γ a closed Jordan curve through α , which leaves the interior of the Q^{p_i} and returns to it once. We now examine how the number of crossing of edges changes before and after the sets replacement.

Assume the cycle traverses only segments from $S_c(p_i)$ while moving along γ from α back to itself. Accordingly, the cycle leaves and returns to the interior of Q^{p_i} through a segments s_1 and s_2 of $S_c(p_i)$, respectively. Traversing s_1 reduces the crossing number by one, and traversing s_2 increases it by one. The total change in crossing number while crossing these segments is zero. If we replace the segments $S_c(p_i)$ by $\widehat{S_{nc}(p_i)}$, both segments s_1 and s_2 are removed. By the assumption that γ leaves and returns to the interior of Q^{p_i} only once, the only intersection points of γ and $S_c(p_i) \cup \widehat{S_{nc}(p_i)}$ are on s_1 and s_2 . Therefore, after the removal of s_1 and s_2 the crossing number of γ w.r.t. $\widehat{S_{nc}(p_i)}$ is zero, thus no change occurs for the crossing number. A symmetrical argument can be applied for the case that the cycle traverses only segments from $\widehat{S_{nc}(p_i)}$.

If the path crosses one element from each set, when flipping between the sets, we get the following changes. Assume the path crosses before the flip a segment from $S_c(p_i)$, and departs the region through it. After the flip, it crosses a segment from $\widehat{S_{nc}(p_i)}$ and enters the region through it. Since we have both flip of direction of crossing and direction of edge, there is no change to the crossing number. The other case is symmetrical.

For a point ξ outside Q^{p_i} , there is always a simple path from ξ towards infinity which does not cross Q^{p_i} (since the polygon is simple). The winding number at ξ is determined by any such path, and is unaffected by the replacement of segments. Any cycle from ξ that crosses Q^{p_i} keeps the consistency of the winding numbers by the first and final part of the proof, below.

Any simple path that is entirely inside Q^{p_i} is unaffected by the replacement as well. Any simple cycle ψ from α to itself, which leaves the interior of Q^{p_i} , can be decomposed into a series of simple paths ψ_1, \dots, ψ_n , where each path leaves and returns to the interior of Q^{p_i} exactly once. Every such ψ_i can be extended to a cycle ψ'_i that begins and ends with α and has a consistent winding number. The consistency of the concatenation of paths ψ_1, \dots, ψ_n is achieved by traversing the cycles ψ'_i . We can decompose the concatenation of paths ψ_1, \dots, ψ_n into a movement along each path ψ_i . At each endpoint β of ψ_i , we move to α and then return to β . Moving back and forth along the same path is obviously consistent. The described movement for each ψ_i actually moves along each cycle ψ'_i .



We saw that any cycle that begins and ends with α has a consistent winding number. Consequently the concatenation of paths ψ_1, \dots, ψ_n has a consistent winding number. \square

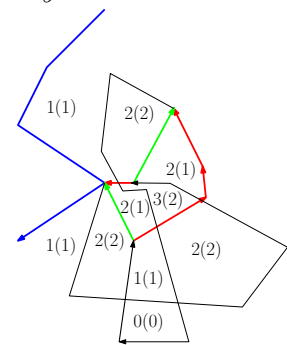
We can now state the theorem which relates the winding number to connected-components counter.

Theorem 4.2.14 (Winding Number/Connected-Components Counter Consistency). *For every point p in $\mathbb{R}^2 \setminus (C \cup C')$, the winding number with respect to C' equals the number of connected components in the intersection of P and $-Q + p$.*

Proof. By Lemma ??, the winding numbers with respect to C' are well defined. The consistency between the winding number and the connected-components counter is verified using the previously shown case analysis. According to it, events which occur on C' are the cases where both the winding number and the connected-components counter change in the same way. \square

Lemma 4.2.15 (Winding number inside the polygon). *Given convolution segments C , for every reflex vertex p_i (w.l.o.g. from P), replacing the segments in the set $S_c(p_i)$ with $\widehat{S_{nc}(p_i)}$, the winding numbers for all points inside Q^{p_i} decreases by one.*

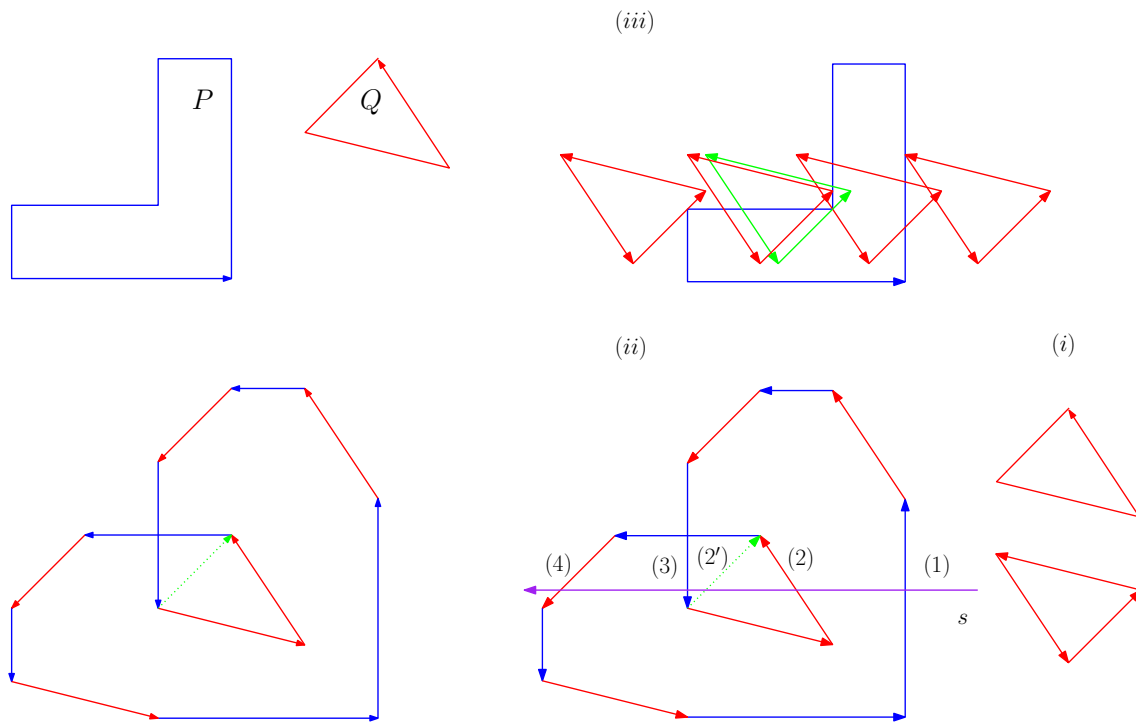
Proof. This follows trivially by noticing the difference of crossing numbers in the neighborhood around the changed segments. Points that belongs to some face of the arrangement which has a convolution segment that is part of Q^{p_i} , after the replacement now share the arrangement same arrangement face as the points interior to Q^{p_i} , which were they neighbours. Thus, decreasing the winding number by one for any point inside the new face. The new segments keep the winding numbers without change for points outside faces interior to Q^{p_i} . \square



In the figure to the right, the black segments are the convolution segments, the numbers are the winding numbers with respect to the convolution, and in parentheses are the winding numbers after replacing the red segments by the green segments.

From Lemma ??, We can see that for every reflex vertex p_i the replacement of segments sets $\widehat{S_{nc}(p_i)}$ with $S_c(p_i)$ causes only the winding number of points interior to Q^{p_i} to increase by one, leaving the rest unchanged.

By Theorem ??, the winding number equals the connected-components counter for all points in each cell in the planar subdivision induced by C' . If we do the reverse process described earlier to transform C' (set of transformed convolution segments) back into C (by replacing non-convolution edges with convolution ones), we now know that we increase the winding numbers in some of the cells interior to Q^{p_i} by one. This increment is for all reflex vertices P_i (w.l.o.g. from P) and the area contained in the respective Q^{p_i} . However, as we noted, by Observation ??, none of the cells whose winding number increased share a border edge with the Minkowski sum. Therefore areas outside the Minkowski sum boundary have zero winding number with respect to both C' and, by Theorem ??, C . In conclusion, the winding number of points outside the Minkowski sum with regard to C is zero, and inside is some number which is at least as high as the winding number with respect for C' , which is positive. Following is that the winding numbers with respect to the convolution segments are positive when there is intersection between P and translated $-Q$. This concludes the proof of Theorem ??.



(a) Two polygons (top) and their convolution (bottom).

(b) Example of shooting a ray and the translation it induces on the reflected polygon (in red). The intersection of the ray with the convolution segments each correspond to one of the translations of $-Q$ depicted at the top. The green segment and event are the event of collision with the green non-convolution edge. This event is “charged” to the red event preceding it to the right. (i) The red polygon and its reflection. (ii) The convolution. (iii) The sliding of the reflected red polygon and the blue polygon from right to left with corresponding events to the traversal of the ray in the convolution.

Figure 4.4: The original scenario ?? and the ray-shooting events ??

4.2.4 Generalization for Non-Simple Polygons

We can generalize the proof for Theorem ?? for the case of non-simple polygons. We now provide a sketch for this generalization; the detailed proof is beyond the scope of this thesis. Recall that non-simple polygons are a collection of loops. Given two input polygons P and Q , we take all the pairs l_p and l_q of loops in P and Q , respectively. For each pair, the connected-components counter and the winding numbers are synchronized as seen in the previous section. The winding number is additive with regard to cycle addition. The connected-components counter measures the intersections of each loop pair independently and thus also additive. Therefore, both counters should remain synchronized and by similar arguments to those in the previous section, Theorem ?? holds for the non-simple case as well.

5

Minkowski Sum Boundary Verification

As discussed in previous chapters, one important sub-routine of the reduced convolution algorithm is deciding whether a given orientable loop lies on the Minkowski sum boundary. In the previous chapters we presented the theoretical basis for two local predicates that can decide this. The collision detection predicate (CDP) decides that a loop is on the boundary of the Minkowski sum if for any point p , and thus for all points, on the loop the polygons P and $-Q + p$ do not collide, where $-Q$ is the polygon Q reflected through the origin. The winding number predicate (WNP) decides that an orientable loop is on the Minkowski sum boundary if the face to the right of the loop (as defined by the loop's direction) has zero winding number. The implementation of a convolution-based algorithm requires to use one of these predicates. In this chapter we show the algorithmic considerations regarding different implementations of these predicates. This implementation can have a profound impact on the runtime as it is invoked for every orientable loop that is suspected to be boundary of the Minkowski sum.

In Section ??, we discuss the implementation options for CDP: We describe two efficient methods for CDP, the first is based on the sweep-line paradigm and the second uses bounding volume hierarchies. In Section ??, we describe the implementation of WNP. We use a ray shooting algorithm to extract the winding number from the full convolution segments. In Section ??, we briefly compare the results for those methods, which were tested using our implementation in CGAL.

5.1 Boundary Checking via Collision Detection

Given two simple polygons P and Q in the plane, CDP has to decide whether P and Q have an intersection point in their interior. To be precise, we do not consider touching of the boundaries to be an intersection. The CGAL Boolean-set operation `intersect`, computes the intersection of two sets in their interior. However, we only need to determine if there is an intersection and not compute it—possibly a much faster operation. The naive test for collision of two polygons requires us to go over $\Theta(mn)$ edges (where m and n are the number of edges of the polygons respectively) for pairwise intersection test, and then perform point-in-polygon test to check if P is contained in Q or vice versa, which

takes $O(n + m)$ time. Thus, the total runtime complexity of this naive intersection-test operation is $\Theta(mn)$, which is not efficient enough for our purpose.

We now describe two methods for implementing CDP. Both these methods detect intersections between segments of P and Q . Note that if no such intersection is found, we still need to test whether P is contained in Q or vice versa, which takes $O(\max\{m, n\})$ time. First we present the sweep line method followed by the bounding volume hierarchy method.

5.1.1 Collision Detection via Sweep Line

The basic **sweep-line** algorithm, as described in [?], allows for computing the intersections between the two polygons in time of $O((m + n) \log(m + n) + k \log(m + n))$ where k is the reported intersection size [?]. CGAL offers an implementation of the **sweep-line** algorithm in a generic framework, which allows extending its functionality using a **visitor** pattern [?]. Furthermore, it uses the exact and robust computation predicates if instantiated with the appropriate kernel. The current implementation of CGAL collision-detection test between two polygons uses the **sweep-line** algorithm to compute the interior intersection and reports whether it is empty. We optimized this process in our implementation to stop the sweep when the first interior intersection between two segments of the two polygons occurs, and then report the found intersection. This strategy reduces the running time to $O((m + n) \log(m + n))$ since we either stop at the first intersection or $k = 0$.

Test for Interior Collision

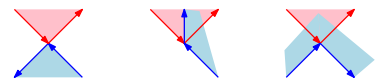
The **sweep-line** algorithm [?] notifies the user when several events occur. Those events include the beginning and ending of the sweeping process and changes in the **sweep status line** caused by the appearance or disappearance of curves and intersections between them. In order to determine if the interiors of the polygons overlap, we need to examine various scenarios where features of the two polygons intersect. We refer to the intersection event of two edges e_P and e_Q from polygons P and Q , respectively, as an **edge-edge intersection**. We further introduce the notion of **weak intersection** if the intersection of the segments results in a point which occurs at a vertex from P or Q . This case requires special attention since these intersections could cause P and Q to touch, but not intersect in their interiors. Finally, if the intersection result is a segment by itself we denote this event as **edge-edge overlap**.

During the **sweep-line** algorithm, we are given the type of the specific intersection event, and have to decide if the polygons intersect in their interior. Since we assume that the polygons are counter-clockwise oriented, for each edge of the polygon, its interior lies on the left side.

In the figure to the right, we see the event of **edge-edge intersection**. In this case it is trivial to see that the polygons will intersect in their interior as the intersection of the half-spaces of the two edges is not empty.



The second event is the **weak intersection event**. The figure to the right illustrates the case that the intersection point is a result of intersection of four non overlapping edges, two from P and two from Q . Notice that each pair of such edges defines an area incident to the intersection point that is interior to the pair's poly-



gon. If the intersection of these areas is not empty then there is an interior intersection. We can decide whether there is an overlap of the areas by examining their order around the intersection point. The **sweep-line** algorithm provides the edges ordered by their angles, with respect to the positive x -axis, around the intersection point. Using this fact we can determine whether there is intersection of P and Q induced by the edges ordering. For an edge of one of the polygons we check if it lies in the area defined as interior by the pair of edges that belong to the other polygon. We perform this test for each of the four edges. We remark that this process does not require the evaluation of geometric predicates. Note for the figure above, the leftmost part shows no intersection, but in the middle and right there is overlap in the interior areas of the polygons.

To deal with the case of **weak intersection** where the intersection point lies in the interior of an edge, we simply split the edge, as shown in the figure to the right. Since the polygons are simple, no more than four edges can meet at a single point.



The third case is when we have two edges that are overlapping. Refer to the figure to the right. If the edges are oriented (with regard to the counter clockwise order of vertices) in the same manner (as shown in the right part of the figure), their overlap causes an interior collision. Otherwise, there is no interior collision, as seen in the left part of the figure. When there are less than four edges at the intersection point, it holds that two of the edges overlap. This case is either handled by the **edge-edge overlap** or by using the test described in the **weak intersection event** while testing the non-overlapping edges.



5.1.2 Collision Detection via Bounding Volume Hierarchy

During a single run of the algorithm for computing the Minkowski sum boundary, the same polygons P and $-Q$ are tested for collision several times but at different relative positions. The difference in the location of $-Q$ relative to P can be expressed by a single translation point α , which changes for each query. Thus each collision detection query performs a collision detection between P and $-Q + \alpha$, where α is the only parameter that changes.

This property can be exploited to speed-up collision detection by constructing a bounding volume hierarchies (BVH) [?] for each polygon, and just by translating one of the hierarchies for each query.

We now briefly describe the bounding volume hierarchy structure:

Bounding Volume Hierarchy

The bounding volume hierarchy (BVH), see [?], is a tree data structure representing a set of geometric objects. The root node represents the entire space. Each internal node in the tree represents a volume in space containing the space covered by its children. These nodes usually represent bounding volumes whose pairwise intersection is simple to compute such as boxes or spheres. The leaves of the tree represent geometric primitives such as triangles, curves, and so forth. The BVH tree is used to answer queries that report primitives that intersect a given region of space.

For estimating the runtime efficiency for BVH traversal, a general cost function is defined as $TC = N_{bv} \times C_{bv}$, where TC is the total cost of the query, N_{bv} is number of bounding volumes traversed by the query and C_{bv} is the cost of computing a single

bounding-volume collision-detection operation. Choosing the type of bounding volumes to use in the hierarchy is a delicate matter. Some bounding volumes have a lower cost for computing intersections amongst themselves, reducing the C_{bv} constant. However, that usually means that they fit the data more loosely, thus increasing the amount of bounding volumes intersections per query, namely increasing N_{bv} . There is no known “recipe” for choosing the correct bounding volume type that works well for all cases.

Our Implementation

We used the axis-aligned bounding box hierarchy provided by CGAL [?] (AABB trees which are discussed in [?], [?]). The primitives for our case are the polygon segments. The bounding box being built is an axis-aligned box around each segment. The root of the hierarchy contains a bounding box of all the segments in a polygon. We build the tree recursively as follows. In every step, the algorithm divides the primitives into two equally sized groups. Each segment is represented by its mid point. The set of mid points (and thus the segments) is partitioned according to the longest axis of the bounding box. Each newly created set is bounded with a bounding box, and we add the bounding box as a child of the root node. Now we proceed with this construction recursively for both child nodes. The recursion ends when the bounding box contains a single segment.

While performing collision detection between two hierarchies, the two trees are traversed simultaneously. Of the options available to traverse the trees simultaneously, our implementation uses the recursive one: Assume we have two hierarchy trees T_1 and T_2 . Let c_1 and c_2 be references for the nodes of the trees T_1 and T_2 , respectively. For each stage of the traversal, the bounding volumes referenced by c_1 and c_2 are tested for intersection. If an intersection occurs, the traversal continues recursively between the following pairs. Children of c_1 are matched with all the children of c_2 . When there is no intersection, the traversal stops exploring this branch. If one of c_1 or c_2 is a leaf, it is matched against the children of the other. If during the traversal c_1 and c_2 refer to two leaves in T_1 and T_2 respectively, and the geometric entities stored therein intersect, then a collision is found. Consequently, the algorithm stops and returns true. If after exploring all branches no collision is found then the algorithm returns false.

The algorithm presented above is designed to answer single CDP query between two stationary objects. For the case of one stationary and one translating polygon, we present a slightly modified traversal process. In each traversal step, we translate the geometric entity (bounding box or segment) of the translating polygon, by the current translation amount, relative to its original location. This is a constant time operation. We build the two hierarchies only once per run of the algorithm, but perform many queries.

The hope with this hierarchy is that if an intersection exists, and the bounding volumes are separated well, the hierarchy will cause the traversal to focus on a single search path for both trees and find the intersection point in logarithmic time for both hierarchies, performing most queries at $O(\log n + \log m)$ time. Our experiments in Chapter ?? shows that the bounding volume hierarchy is indeed the best performing heuristic in practice, thus adding merit that this hope is justified.

5.2 Boundary Checking via Winding Numbers

In Chapter ?? we saw that regions in the plane have a non-zero winding number with respect to the convolution cycles if and only if they are interior to the Minkowski sum.

Observation ?? states that for determining the winding number of a point p , it is sufficient to “follow” a simple path from p to a point on the unbounded face, and count the signed number of edge crossings along this path. These facts give rise to an alternative approach to computing the winding number of a point. It is possible to shoot a ray and count the number of convolution segments it intersects with the correct sign. Of course we have to count the segments of the full convolution rather than of the reduced one. However, this method stills allows to test faces of the reduced convolution arrangement.

In the first part of this section we describe the choice of data structure that returns the crossing number for a given query point and analyze the complexity for a query. The second part discusses how to implement the ray–shooting query and deal with ray queries that are not in general position, namely when the ray intersects a vertex or overlaps a segment.

5.2.1 Ray Shooting Data Structures

The test is done by computing the winding number of an arbitrary point which resides immediately to the right of each edge of an orientable loop. Wein [?], showed a method to compute this number by using the arrangement of the full convolution. We discuss a different approach, which stores the full convolution segments without intersecting them, and uses this information to perform a local query, which computes the winding number for every suspected boundary loop. As seen in Chapter ??, there are $O(mn)$ segments both for the full and the reduced convolution, where m and n are the number of segments in the input polygons. This bound holds as long the intersections between the segments are not computed. Note that computing the arrangement requires the computation of the intersection points between all input segments. Computing the segments of both the reduced and the full convolution has similar running time. The main performance impact occurs while inserting the segments to the arrangement, where the runtime can go up as high as $\Theta((mn)^2)$. Let ρ denote the ratio between the full convolution size and the reduced convolution size. We expect a saving of a factor of ρ^2 in the running time of the reduced convolution algorithm since this is the reduction in the size of the arrangement. However, it is possible to store the segments of the full convolution, and only use them for computing the crossing numbers for specific query points. Note that a naive implementation that goes over all convolution segments and tests which segments are crossed by the ray has a runtime complexity of $O(mn)$, which is identical to the time required for a naive CDP.

For improving the efficiency over the naive implementation a natural approach is to find a data structure that will hold the segments and prepare them for a ray shooting query. Since we may choose the ray however we like, we can decide that the ray has to be axis aligned. In order to choose the appropriate data structure we should first state the query we wish to answer. Assume, for instance, we shoot the ray horizontally to the left. For now assume there are no horizontal edges and the ray does not intersect any vertices. With this assumptions we can split the set of convolution segments into two sets: those with the lexicographically smaller vertex is below the second vertex (upward oriented segments), and those with the smaller vertex above the other one (downward oriented segments). The crossing number for a point p while shooting a horizontal ray towards the left will be the number of segments it intersects from the latter set minus the number of segments it intersects from the former set. This is a counting query [?].

The most trivial data structure to use is the arrangement, however as stated, com-

puting it is too costly. The chosen data structure should not require the computation of all the intersection points between the full convolution segments. Otherwise, it is at least as slow as computing the arrangement.

One candidate data structure is the partition tree. This data structure is a bit complex and its description is given in [?]. Overmars et al. [?] show how to use the partition tree to store general segments in the plane and to report or count intersection with a query segment. It has log linear storage and construction time with regard to the number of segments, and for any choice of $\epsilon > 0$ it is capable of answering a counting query in time $O(n^{0.5+\epsilon})$. It is also possible to improve this data structure by “blending” it with another data structure called cutting tree. With this technique it is possible to build a data structure that for any $n \leq m \leq n^2$ takes $O(m^{1+\epsilon})$ storage, and has a query time of $O(\frac{n^{1+\epsilon}}{m^{0.5}})$. Of course here the storage requirement is also a lower bound on the construction time, and consequently the running time.

Unfortunately, such data structures require the implementation of multi-level trees, which seem to be slower in practice than heuristic alternatives, despite having better guarantees on the worst-case asymptotic run time. We tested the ray shooting approach with the following heuristic instead. We use an interval tree structure, where the intervals contains the y-axis values of each segment. Given a query point p the tree allows to report the list of segments which contains the y-coordinate of p in time $O(\log n + k)$, where n are the number of intervals in the tree and k is the number of reported segments. The heuristic aims to improve runtime for queries where the y-coordinate of p is only contained in a small number of segments (i.e., $k \ll n$).

5.2.2 Ray Shooting Implementation

We now describe the ray shooting algorithm. The algorithm calculates the winding number for the face inside of a suspected hole loop L .

The algorithm gets as an input the segment s , that is assumed to be non-horizontal, which belongs to the suspected hole loop L (as long as L is not degenerate we can always find such a segment). A ray r is shot from the middle of s towards the left. The first step of the algorithm is to compute the crossing number of r . Recall that for a path ξ the crossing number $SC(\xi)$, is the sum of signed crossings with respect to the segment’s direction along ξ . The crossing number of r includes crossing s , i.e., the shot measures the crossing number a little to the right of s . We now have to figure if this point is inside L . If not, the crossing number of r is not the same as the winding number of L and should be corrected to obtain the winding number inside L .

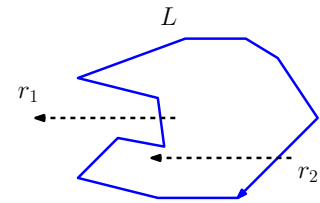
In order to compute the crossing number of r , the algorithm queries the data structure for the crossing number. As stated, the data structure used in the implementation is the interval tree.

After calculating the crossing number for r , we need to determine the winding number inside L . For this purpose, we have to determine if r is shot into or out of L .

Observation 5.2.1. *Let $\Delta_{WN}(r)$ be the difference between the winding number inside L and the crossing number of r . We have:*

1. *If s is upwards oriented then $\Delta_{WN}(r) = 0$*
2. *If s is downwards oriented then $\Delta_{WN}(r) = -1$*

Proof. Since the ray is shot as if it were a little to the right of s , if s is an upwards segment then r is shot from the interior of the hole (see r_1 in the figure to the right). If s is oriented downwards then r is shot into the loop, from the outside (see r_2 in the figure to the right). Therefore, the intersection of r with s increases the crossing number of r by one relative to the winding number inside L . We thus need to decrease the crossing number by one in order to compensate. \square



For handling cases where the ray hits a vertex or a horizontal segment we assume a symbolic translation, which means that the ray is considered to be ϵ higher than its original location. In practice, this means that the ray does not intersect horizontal segments and whenever it hits a vertex, it is assumed to hit only segments that are above it.

5.3 Performance

Experiments on the Lien data set (the benchmark data set, see Chapter ??) have shown that the sweep-line collision detection is the slowest method. The best performance is that of the bounding volume hierarchy which is a widely used method for performing collision-detection queries. It also reuses most of the information for performing queries as compared to the other methods, as the hierarchy is built only once. The ray-shooting method produced intermediate results. We can speculate that the degraded performance is due to the existence of a larger number of false hole loops than true hole loops. False loops are detected efficiently by the BVH since once a collision is found the iteration on the hierarchy stops. The ray-shooting method has no such advantage and always has to count all the segments that are candidates to hit the ray. Therefore, the method chosen for performing the experiments is the bounding volume hierarchy.

6

Experimental results

This chapter describes the experiments that were conducted to assess the validity, robustness and speed of our new implementation of the reduced convolution algorithm in CGAL. We denote our implementation of the reduced convolution algorithm by RCA. We compare the runtime and memory consumption of RCA to the full convolution algorithm implemented in CGAL by Wein [?], which we refer to as CWEIN. We show an example of robustness failure in which the original implementation of the reduced convolution [?] produces incorrect results. We also briefly compare RCA, CWEIN and Lien’s implementation for the data described in [?]. Note that both RCA and CWEIN currently support only simple polygons as inputs.

In Section ??, we describe the different data sets of input polygons we are experimenting on. Recall that for two input polygons with m and n vertices, the complexity of the Minkowski sum ranges from $O(m + n)$ to $O((mn)^2)$. We should use data which represents all of these cases. Section ?? compares the performance of RCA and CWEIN on the data sets described in Section ?. Section ?? shows an example where inexact geometric computing fails on an input. Finally, Section ?? discusses the memory usage of RCA compared to that of CWEIN.

We assume throughout this chapter that we have two input simple polygons P and Q with number of vertices m and n , respectively.

6.1 Test Sets

In this section we describe the input data sets that are used for performing the experiments. We used data from previously tested data sets given by Lien [?] and Wein [?]. Lien’s data set should allow to compare the performance of RCA with Lien’s algorithm. Both data sets contain pairs of polygons whose Minkowski sums complexity ranges from $O(m + n)$ to $O((mn)^2)$. Lien’s set is described in the first part followed by Wein’s set. Additional data sets were generated to allow testing of different polygon families with varying sizes. The polygon families include convex, star shaped and simple polygons. This data set is generated automatically which allows to examine the effect of scaling up in the number of vertices for each polygon family, on the performance of the algorithms.

This set is described in the third part of this section. We produced a data set showing the worst-case behavior for RCA. This set contains a generated “Fork” like polygon matched with an “L” shaped polygon. The Fork is built with n teeth such that its Minkowski sum with L contains n^2 holes. We can control the number of teeth of the fork directly. Finally, we mention a small test set which is used to validate the robustness of the algorithm on degenerate cases.

6.1.1 Lien’s Data Set

The data set used by Lien’s experiments in [?] to compare to Wein’s code spans a set of geometric models. This set is available on the web¹. In Figure ?? we show some of the polygons in the data set. Figure ??, illustrate a sample of their Minkowski sums. The samples in this set are used as a base line performance set as they can compare the performance between RCA, the implementation given by Lien and CWEIN. This set contains diverse real-world models and includes an example where the Minkowski sum contains $O((mn)^2)$ holes.

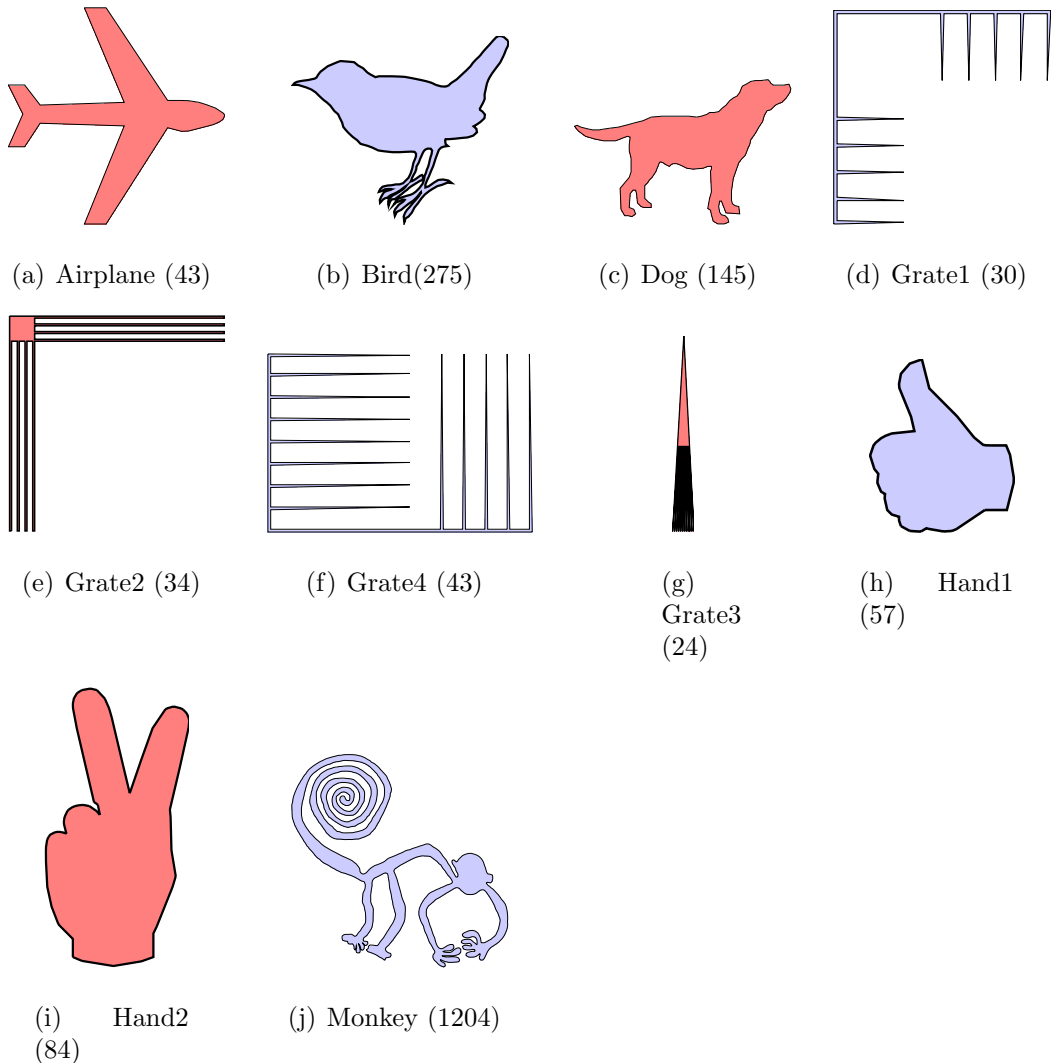


Figure 6.1: Lien’s input polygons; number of vertices in parenthesis.

¹<http://masc.cs.gmu.edu/wiki/ReducedConvolution>.

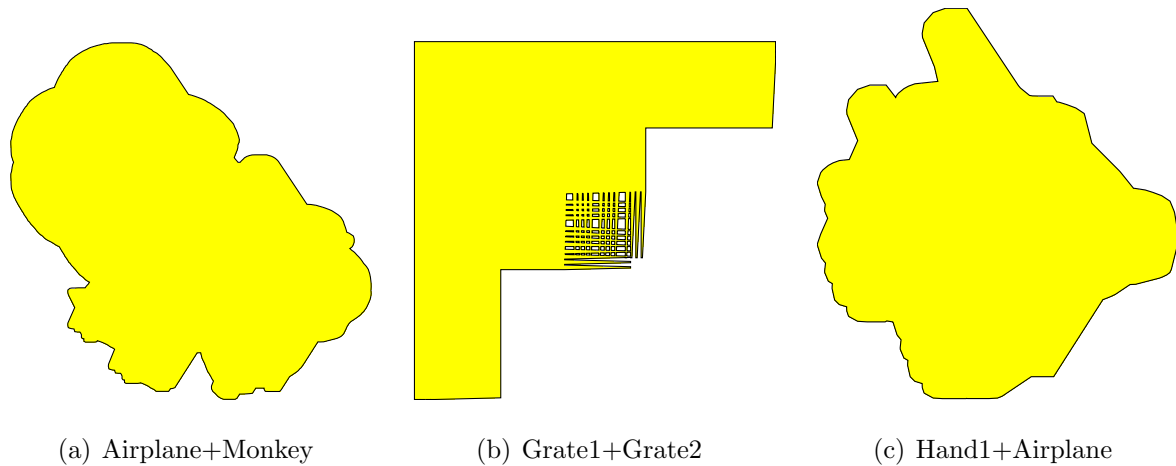


Figure 6.2: Sample Minkowski sum results from Lien's dataset.

6.1.2 Wein's Data Set

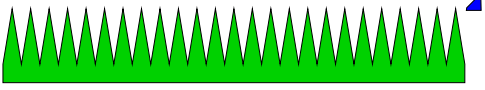
Wein used the data set from [?]. Unfortunately, only a small number of polygons of this set, which is described as a representative set in [?], is available on the web and is used for our benchmarks. Here is the original description of the data set from Chapter 3 of Flato's Thesis [?] and Wein's article [?]:

Input Set	Description	Figure
comb	P is a 'comb' with 25 teeth and 53 vertices, Q is a convex polygon with 22 vertices.	?? (a)
star	P and Q are star shaped polygons with 40 vertices each.	?? (g)
fork	P and Q are a series of vertical and horizontal 'teeth' which together interwind such that their Minkowski sum contains $\Theta((mn)^2)$ edges. P and Q have 34 and 31 vertices, respectively.	?? (b)
mixed chains	P is a polygon in which one of the poly-line is a combination of approximations of a concave or convex semi circles with 82 vertices, Q is star shaped with 30 vertices.	?? (e)
knife	P is a triangle split into teeth, Q is a comb containing vertical and horizontal teeth. P and Q have 64 and 12 vertices, respectively.	?? (d)
random	P and Q are randomly generated polygons.	?? (c) and ?? (f)

Figure ?? and Figure ?? show the data set polygons and their Minkowski sums.

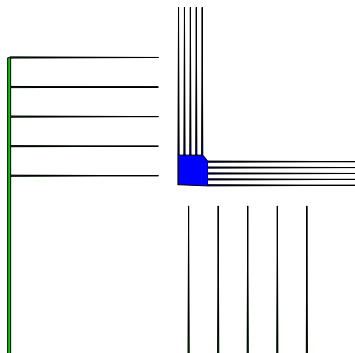
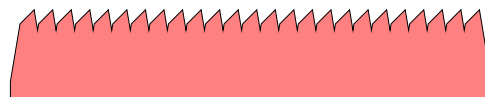
6.1.3 Randomly Generated Families of Polygons

The randomly generated classes of polygons are convex, star-shaped and simple polygons (examples are shown in Figure ??). We begin by choosing the number of vertices n for a polygon. We generate a random convex polygon by sampling n angles uniformly from the interval $[0, 2\pi]$ which determines n vertices on the unit circle. We connect the points to create a convex polygon which is then scaled by a uniformly sampled value of the interval $[0.5, 5]$.



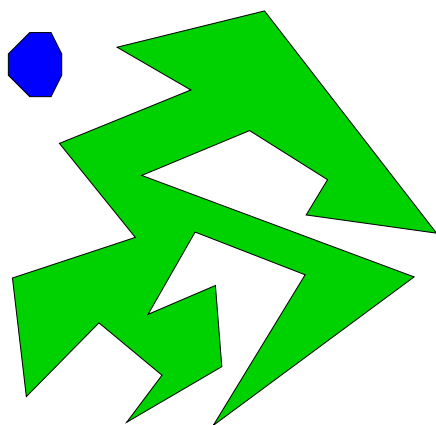
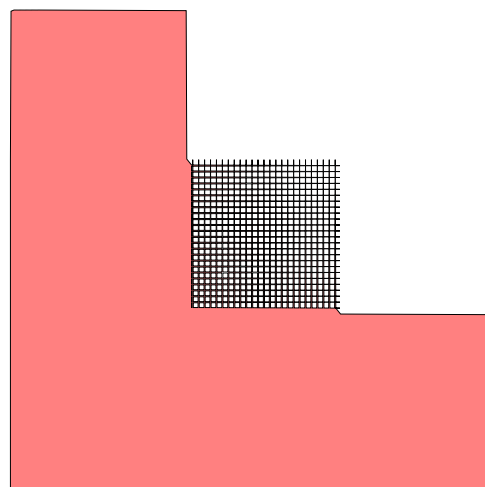
 P (green) has 53 vertices, Q (blue) has 22 vertices

(a)



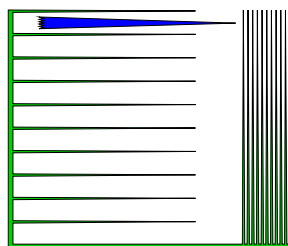
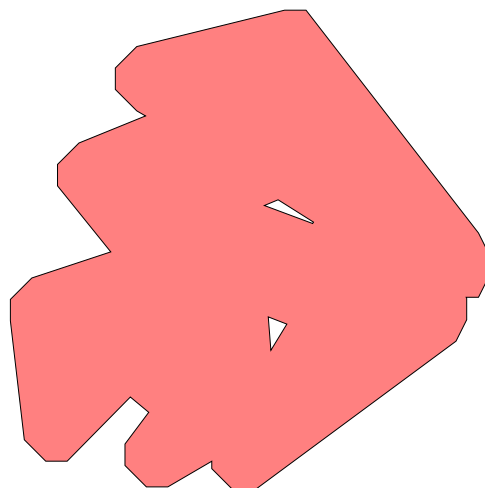
P (green) has 34 vertices, Q (blue) has 31 vertices

(b)



P (green) has 30 vertices, Q (blue) has 15 vertices

(c)



P (green) has 64 vertices, Q (blue) has 12 vertices

(d)

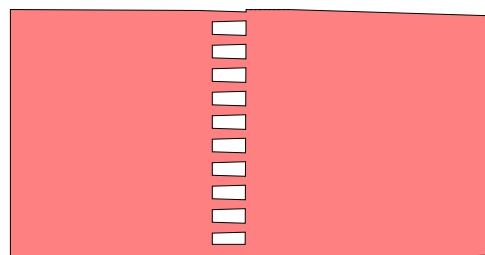
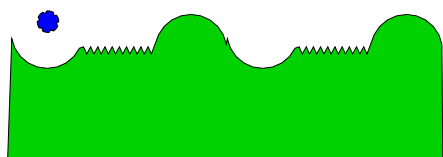
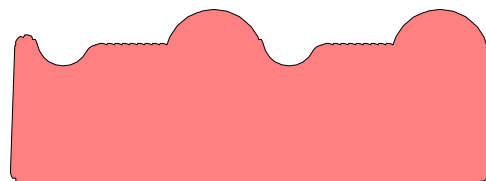


Figure 6.3: Wien's input polygons (left) (a) *comb*, (b) *fork*, (c) *random*, (d) *knife* and their Minkowski sums (right).



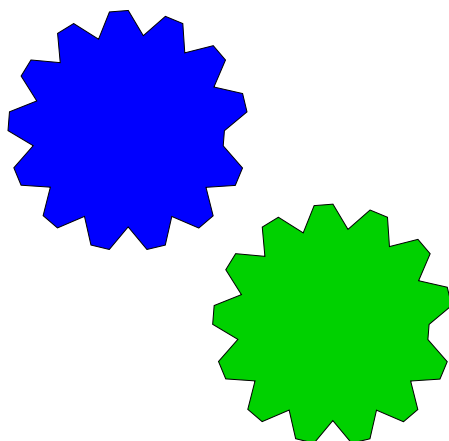
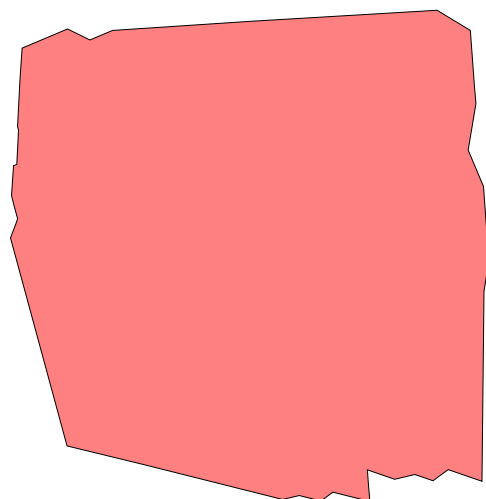
P (green) has 82 vertices, Q (blue) has 30 vertices.

(e)



P (green) has 40 vertices, Q (blue) has 20 vertices.

(f)



P (green) has 40 vertices, Q (blue) has 40 vertices.

(g)

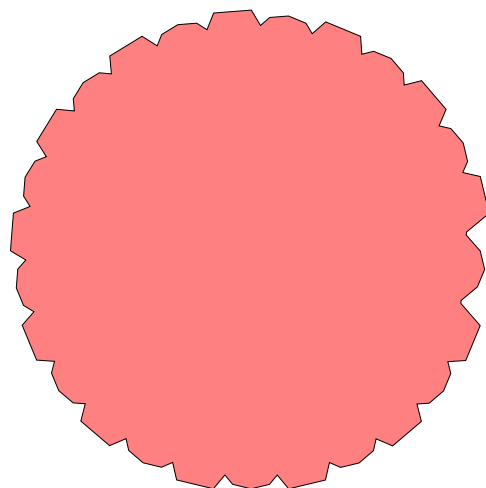


Figure 6.4: Wien's input polygons (left) (e) *chain*, (f) *random*, (g) *stars* and their Minkowski sums (right).

To form a star shaped polygon, we generate a similar sequence of n sorted angles. However, we now pair them with n random radii sampled uniformly from the interval $[0, 1]$. Each pair defines a point in polar coordinates and the paired sequences defines n points winding counter clockwise around the origin, which can be connected to compose a star shaped polygon.

To create a random simple polygon with $n - 1$ edges we uniformly sample $k + n$ points in the plane (for this set we sample from the square $[0, 1] \times [0, 1]$), where $k > 1$ is some constant (we chose $k = \lceil n/10 \rceil$). Then we perform a Delaunay triangulation on the set of points where the boundary of the triangulation is a convex polygon. However, the number of edges on the boundary are not $n - 1$ (almost always). As long as the number of boundary edges is greater than $n - 1$, we find a triangle with two boundary edges and remove it. Consequently, the number of boundary edges decreases by one. If there are less boundary edges than $n - 1$, we remove a triangle with one boundary edge. This increases the number of boundary edges by one. We repeat this process until there are $n - 1$ edges on the boundary of the polygon, in case there are no matching triangles to remove, the process stops and fails.

6.1.4 Fork Data Set

This set contains a Fork polygon created with a variable number of rectangular teeth see Figure ???. The Fork is summed with a fixed L shaped polygon see Figure ??? with 6 vertices scaled in size to create exactly n^2 holes in the sum. This is the worst case for RCA, since the second polygon is very simple and there are many valid holes in the sum. The simplicity of the L polygons causes the difference between the reduced and the full convolution to be small, and the quadratic number of holes causes RCA to perform n^2 expensive collision detection test. Recall that for holes the collision detection test is the most expensive since a point inside the hole is not contained in the Minkowski sum. Thus P and $-Q$ translated do not intersect, and the collision detection test has no option of finding an intersection of a segment of P and a segment of $-Q$ translated by the query point, and stopping the procedure before iterating all options.

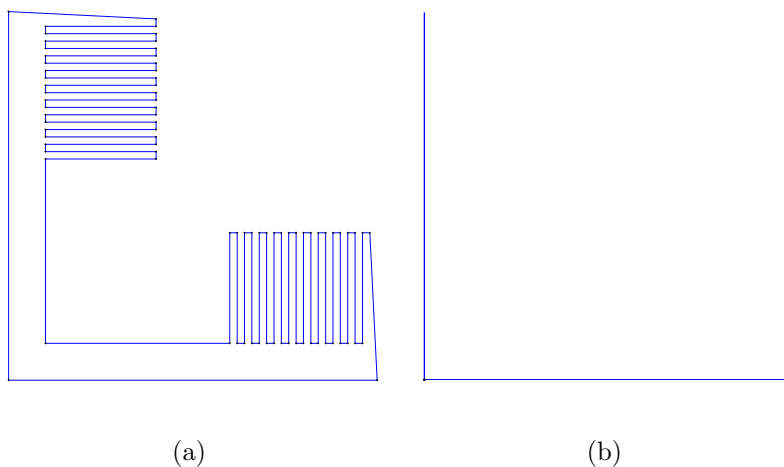


Figure 6.5: Generated fork polygon with 10 teeth ??, generated L-shaped polygon with 6 vertices ??.

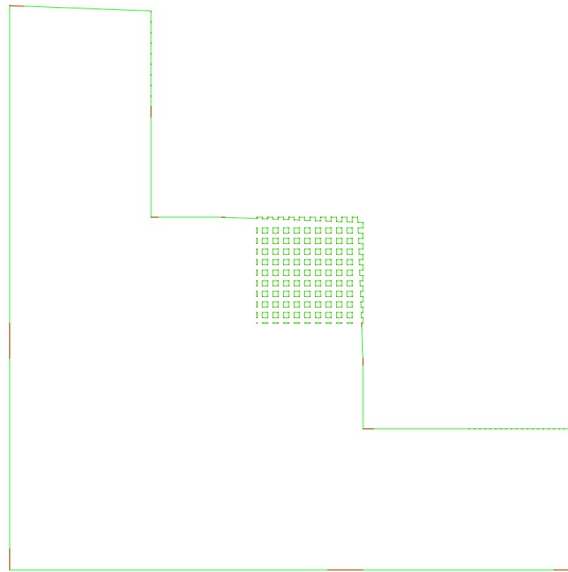


Figure 6.6: The Minkowski sum of the fork and the “L”-shaped polygon.

6.1.5 Degenerate Input

We include some polygons whose Minkowski sum boundary contains one or zero dimensional features. Note that Lien’s code specifies that it does not report such features in the result.

6.2 Performance

In this section we describe the experiments that were used to evaluate the runtime performance of RCA as compared to CWEIN. In Section ?? we briefly review and compare the different steps performed in a convolution-based method. In Section ?? we describe the platform used for the benchmarks. Section ?? shows the results for each experiment. Finally, we analyze the results of all the experiments.

6.2.1 Performance Stages

Let us review the main time consuming elements of the Minkowski sum computation using convolution methods:

1. Compute the convolution cycles (*Convolution Cycles Stage*).
2. Compute the intersection points between the convolution cycles (*Arrangement Build Stage*).
3. Deduce the features which compose the boundary of the Minkowski sum (*Hole Verification Stage*).

We refer to this list as “stages of computation”. We now discuss this stages for RCA and CWEIN.

The first stage of computing the convolution cycles is implemented in RCA with an optimal runtime complexity of $O(m+n+K)$ where K is the complexity of the convolution cycles (see Section ??).

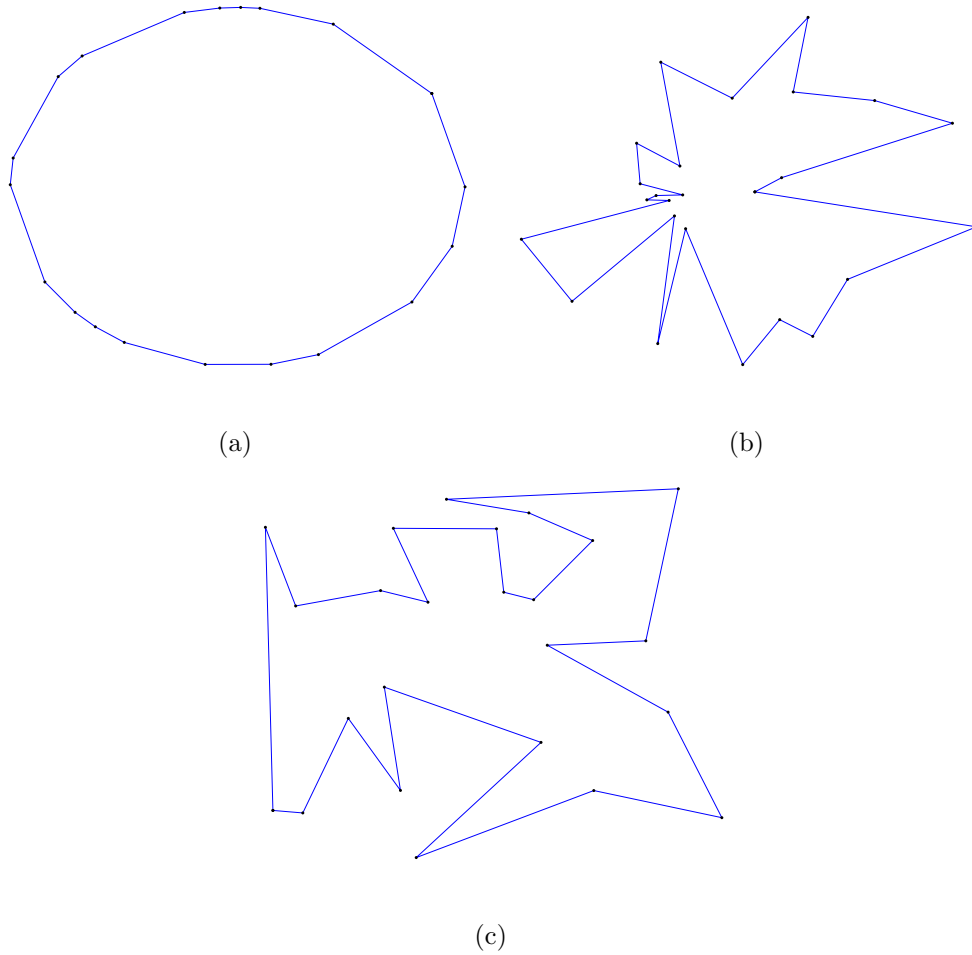


Figure 6.7: Randomly generated convex polygon ??, randomly generated star-shaped polygon ?? and randomly generated simple polygon ??.

Wein [?] implemented an output-sensitive algorithm for computing the full convolution cycles with runtime complexity of $O(m+n+\min\{m+n+n_r m+m_r n\}+K)$, where n_r and m_r are the number of reflex vertices of P and Q respectively. Note that Wein's method produces more segments since the reduced convolution is a subset of the full convolution. However, both methods trace the cycles of the full convolution.

The second stage is implemented in both methods by inserting the convolution segments into an arrangement. Computing the arrangement of n segments and I intersection points, involving a sweep which takes $O(n \log n + I)$. This step is sensitive to the output size, since the number of intersections between convolution segments is at least proportional to the number of output components, which can be as high as $\Omega((mn)^2)$. However in this step RCA is usually more efficient than CWEIN since the size of the reduced convolution can be much smaller than the full convolution [?].

The final stage must decide which features are actually on the boundary of the Minkowski sum. This is the stage where the two methods differ. CWEIN computes the winding numbers of each face by traversing the arrangement in a breath first manner. This approach's runtime complexity is linear in the number of faces, which is as complex as the output size, denoted by η .

For RCA, this stage induces two additional stages. The first is creation of the bounding volume hierarchy performed once per run of the algorithm, which we refer to as *AABB Tree Stage*. The second is reporting the low-dimensional boundary features, which we refer to as *Degenerate Handling Stage*. The latter stage is optional and can be disabled

when the polygons are treated as closed sets (in such case there are no low-dimensional boundary features). Since CWEIN reports low-dimensional features, in order to conduct accurate experiments we enabled this option for RCA as well.

For computing *Hole Verification Stage*, RCA goes over all faces in the arrangement of the reduced convolution, filters out some faces with the orientation check and the nesting filter and finally uses the collision detection predicate for the remaining loops. The first two filters take amortized time proportional to the output size η as they involve iterating the arrangement features once. In contrast, the collision-detection predicate is more expensive, even though it uses the bounding volume hierarchy it still has worst-case runtime complexity of $O(mn)$ per test. Since this test is performed for every actual Minkowski sum boundary feature, the worst time complexity is $O(mn \cdot \eta)$. However, in practice η is usually less than $O((mn)^2)$, and this worst-case scenario is uncommon.

6.2.2 Platform

All the tests but the fork test were performed on Intel core i7 model 870 at clock speed of 2.93Ghz . The machine has 8GB of RAM and operates on Microsoft Windows 7 64-bit. The binaries were compiled with visual studio 2010 64-bit and CGAL version 3.7.

The fork test was run on Intel core i5 model 750 at clock speed of 2.66Ghz . The machine has 4GB of RAM and operates on Microsoft Windows 7 64-bit. The binaries were compiled with visual studio 2010 64-bit and CGAL version 3.7. This second platform was used out of technical reasons and does not carry weight on the quality of the experiments.

6.2.3 Results

Lien's Data Set

In this test we excluded inputs where the number of segments in the convolution arrangement is less than 2000. Below this number there is no preference to either one of the methods since the overhead for different stages of the algorithms dominates the runtime.

Consider Figure ?? . The x -axis is the convolution arrangement size. The y -axis which measures the time, is limited to ten seconds for the smaller times to be visible. The last two examples took for CWEIN over thirty seconds. In the figure we see that RCA is faster than CWEIN for all but two examples. Those examples are the cases where the input is the monkey (1204 vertices) and a small polygon (triangle and rectangle). For this case the stage for building the BVH of the monkey, even though done only once, takes a significant time of the computation causing a delay for RCA. We see that for the smaller examples, Lien's implementation is faster than RCA and CWEIN. However, in some cases, when the input is large, RCA is faster than Lien's implementation. Further investigation is required but is however, beyond the scope of current work.

Wein's Data Set

In Table ??, we provide results for the pairs of polygons used by Wein [?]. Since the input is relatively small for most pairs the computation time is very small. Therefore, we provide the actual time each implementation takes. For inputs with convolution arrangement size of at least 2000, there is one pair where RCA takes longer than CWEIN, the fork example.

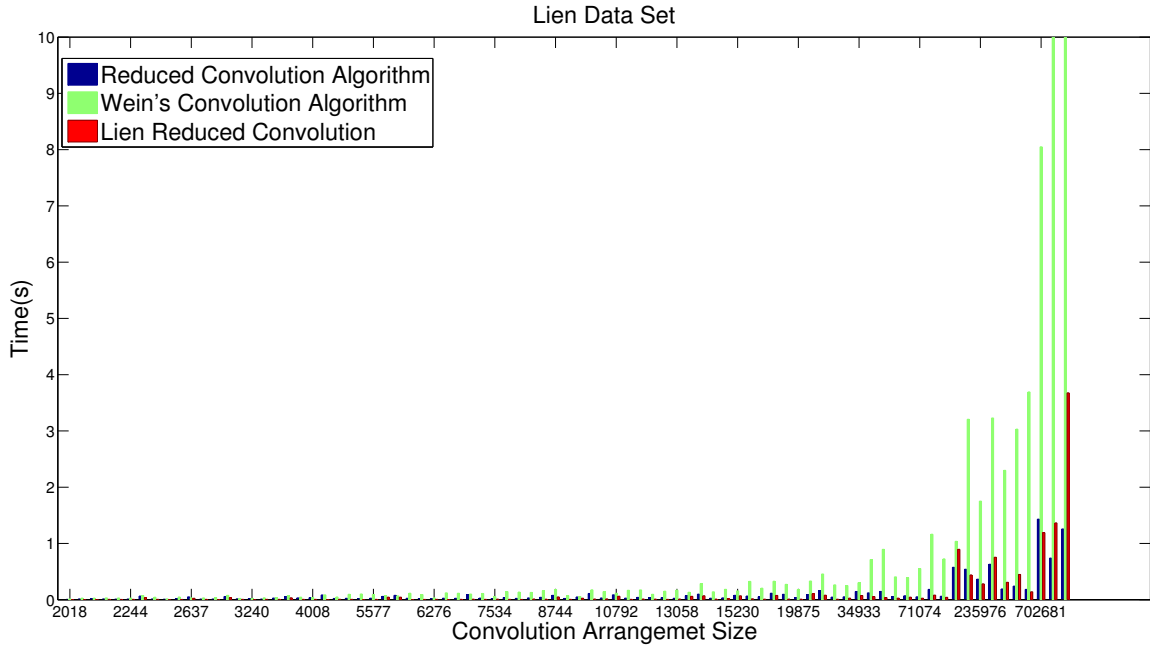


Figure 6.8: Benchmark for Lien's data set. The x -axis is the convolution arrangement size. The y -axis is limited to the range $[0, 10]$ seconds.

This is the case where the Minkowski sum has quadratic number of holes in the number of input vertices, each requires a collision detection test in RCA.

P	Q	m	n	T_W	T_{RC}	A_W	A_{RC}
cavity_part1.dat	cavity_part2.dat	22	8	0.001	0.002	161	81
chain_part1.dat	chain_part2.dat	82	30	0.025	0.013	2868	801
comb_part1.dat	comb_part2.dat	53	22	0.006	0.009	651	579
conc_chain_part1.dat	conc_chain_part2.dat	22	22	0.017	0.004	3543	74
fork_part1.dat	fork_part2.dat	34	31	0.185	0.322	22063	6154
knife_part1.dat	knife_part2.dat	64	12	0.066	0.028	9749	2946
rand3015_part1.dat	rand3015_part2.dat	30	15	0.01	0.005	2309	426
random_part1.dat	random_part2.dat	40	20	0.02	0.01	4698	831
stars_part1.dat	stars_part2.dat	40	40	0.031	0.014	5482	791

Table 6.1: Wein's data set results, T_W and T_{RC} is the runtime of CWEIN and RCA, respectively. A_W and A_{RC} is the arrangement size of CWEIN and RCA, respectively.

6.2.4 Random Data Set

The random data set is given to show how increasing the complexity of P for various polygon families affect the runtime of RCA and CWEIN. Recall that computing the Minkowski sum of two convex polygons has an efficient $O(m + n)$ algorithm, which does not involve the computation of an arrangement. Consequently, for this case this algorithm is preferable over both methods. Thus, we do not consider this case for our benchmarks.

We now describe the various benchmarks that we performed on the randomly generated polygons.

Convex and Star Fixed

For this test P is a convex polygon and Q is a star shaped polygon. We vary the number of vertices of P from 50 to 980 with a step size of 30, each step has 20 tests. Q remains with a fixed size of 100 vertices. Refer to Figure ???. This is the plot of the runtime of both algorithms as a function of the convolution arrangement size. We fit a line using linear regression to RCA and CWEIN runtime obtaining high R^2 scores (above 0.95, which is considered a good fit), which means that the runtime of both algorithms is a linear function of the number of segments in the convolution arrangement. Since both lines approximately touch the origin we can compare the slopes of the two lines to see that for this case RCA is about twice as fast as CWEIN for any input size.

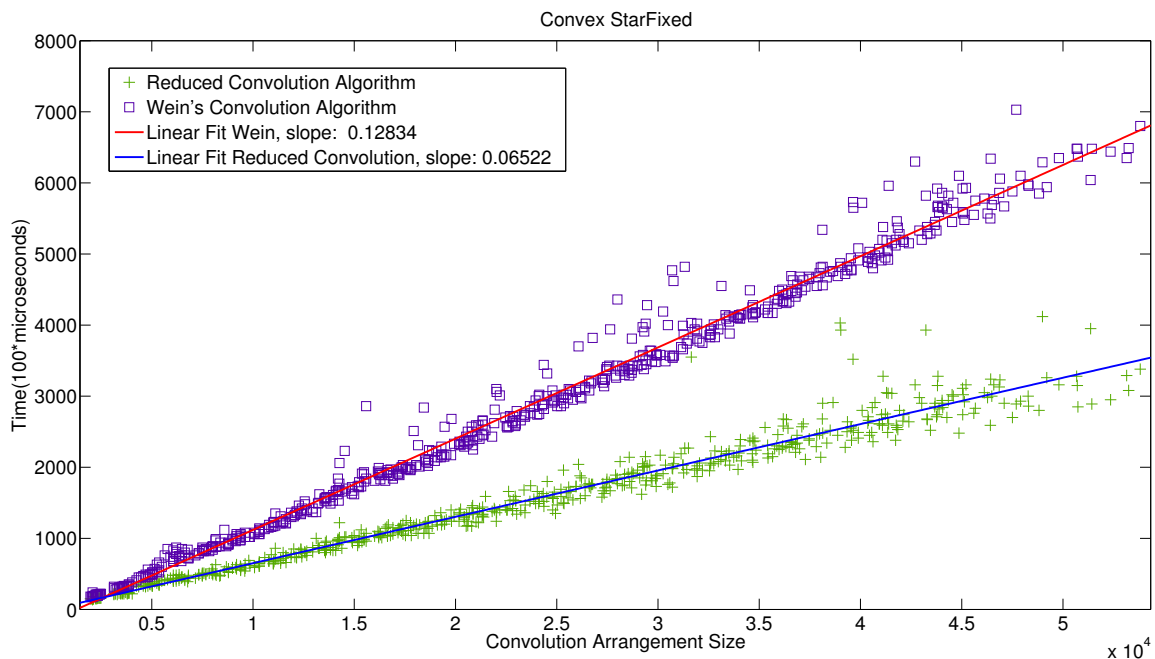


Figure 6.9: Benchmark for Convex and Star Fixed Test.

Star and Star Fixed

For this test both P and Q are star shaped polygons. We vary the number of vertices of P from 25 to 340 with a step size of 3, each step has a single test. Q remains with a fixed size of 100 vertices. Refer to Figure ???. This is the plot of the runtime of both algorithms as a function of the number of vertices of P . We fit a quadratic curve using linear regression to RCA and CWEIN run times obtaining high R^2 scores (above 0.95), which means that the runtime of both algorithms is a quadratic function of the number of vertices of P . We can see in the plot that both algorithms start close to zero meaning they have low constant overhead. We can see that for rather large examples, say where P has 300 vertices, RCA is about four times faster than CWEIN. The time is quadratic in the number of vertices of P since star-shaped polygons have many reflex vertices which creates more convolution cycles. These cycles increase the chance for intersections in the convolution arrangement.

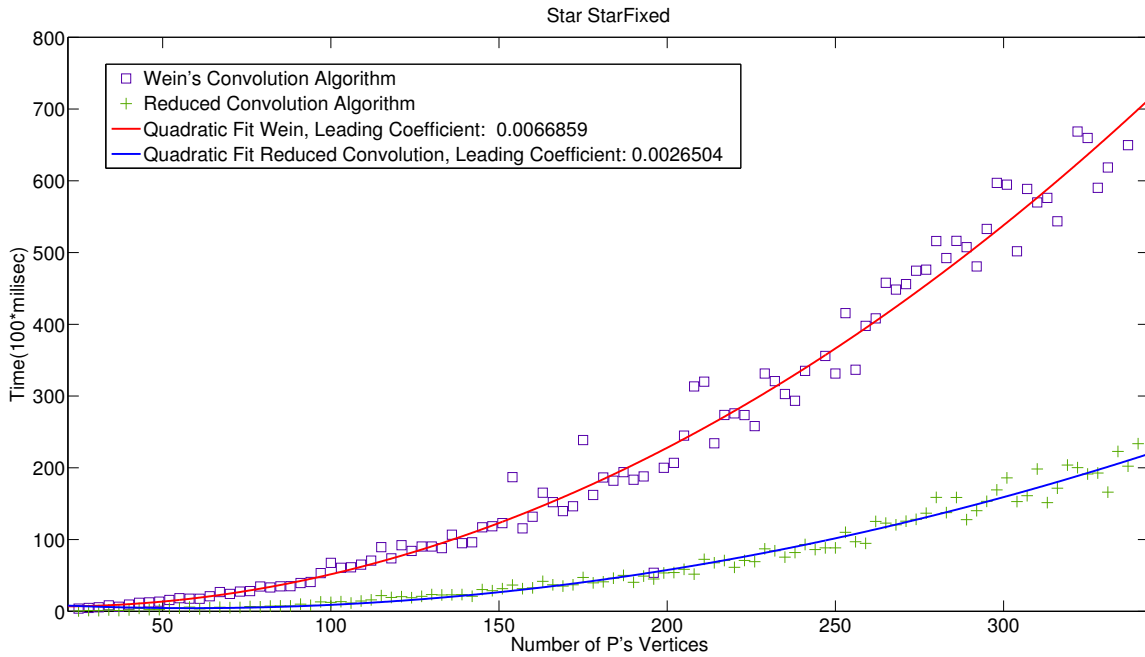


Figure 6.10: Benchmark for Star and Star Fixed Test.

Simple and Simple Fixed

For this test both P and Q are random simple polygons. We vary the number of vertices of P from 30 to 200 with a step size of 5, each step has 5 tests. Q remains with a fixed size of 100 vertices. Refer to Figure ???. This is the plot of the runtime of both algorithms as a function of the number of vertices of P . We fit a quadratic curve using linear regression to RCA and CWEIN run times with high R^2 scores (above 0.95), which means that the run time of both algorithms is a quadratic function of the number of vertices of P .

We further add plots showing the runtime for each different stage of computation for both algorithms:

Figure ?? plots the times spent on each stage of CWEIN. The *Convolution Cycles Stage* refers to tracing the convolution cycles, the *Arrangement Build Stage* measures the construction time of the convolution arrangement and the *Hole Verification Stage* is the calculation time for computing the winding number for each face of the arrangement. We can see that the *Convolution Cycles Stage* is linear in the input size as expected. The *Hole Verification Stage* seems to be linear in the input size, however we know it to be quadratic and as complex as the arrangement size. Thus, the *Hole Verification Stage* has small quadratic coefficients. We observe that *Arrangement Build Stage* is quadratic, as we expected since the example is complex and contain many cycles, thus intersecting segments.

Figure ?? plots the times taken for each stage of RCA. We can see that the *AABB Tree Stage* and the *Convolution Cycles Stage* consume very little time and depend linearly with a relatively small slope on the input size. Therefore, these stages become negligible for complex inputs. The *Degenerate Handling Stage* again seems linear but we know that it should be quadratic as we traverse every feature in the arrangement. The two non-linear factors which contribute to the non-linear curve (quadratic) of the total time are the *Arrangement Build Stage* and the *Hole Verification Stage*. We can see again that a linear addition to the input causes a quadratic growth in the time required to build

the arrangement. It also causes the number of suspected hole loops to rise, increasing the time needed for *Hole Verification Stage* since each hole with the correct orientation induces an expensive collision detection test.

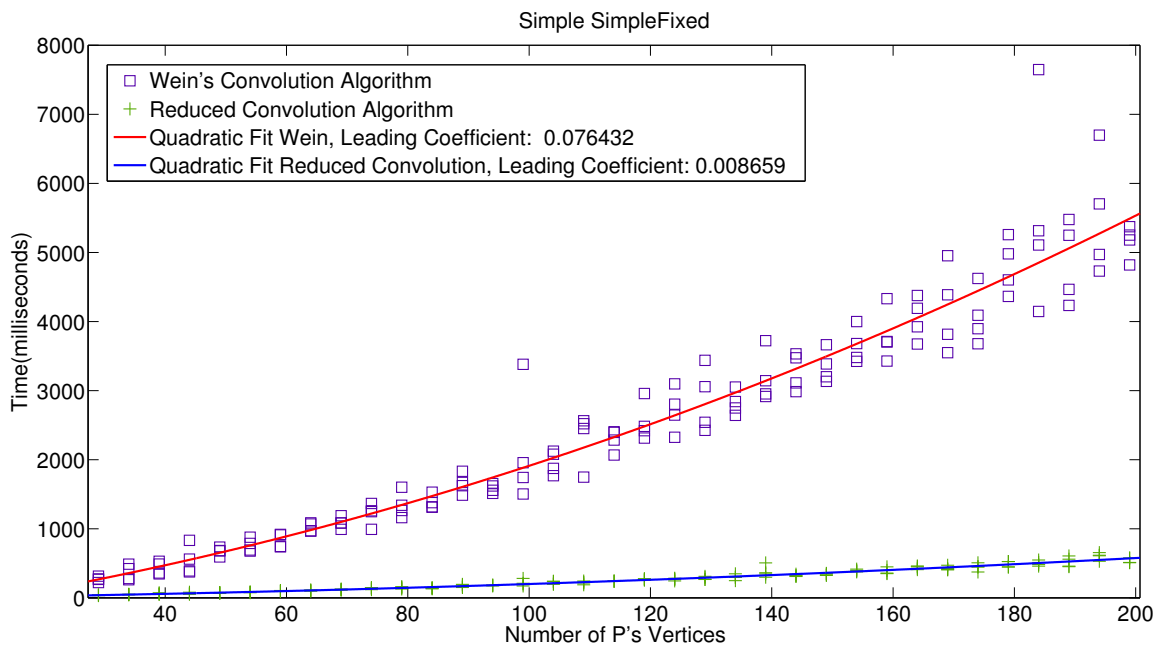


Figure 6.11: Benchmark for Simple and Simple Fixed Test.

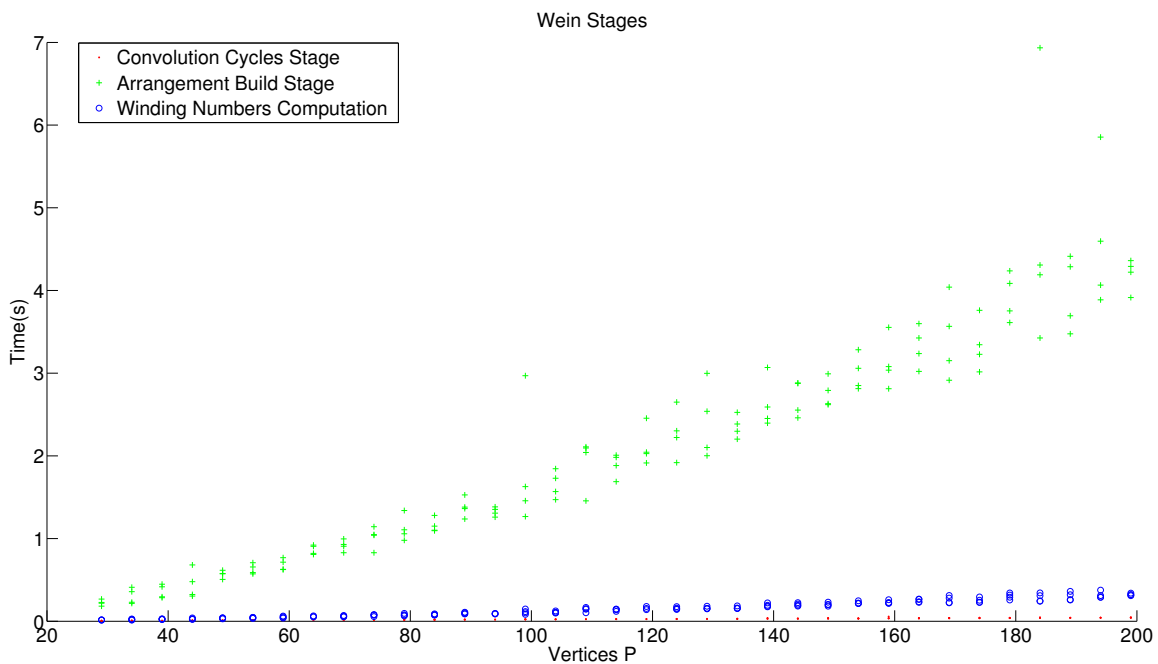


Figure 6.12: Stages of computation of CWEIN for the Simple and Simple Fixed Test.

6.2.5 Fork Data Set

For this test both P is a “fork” and Q is an “L” shaped polygon. We vary the number of teeth of P from 50 to 250 with a step size of 10, each step has a single test (since none

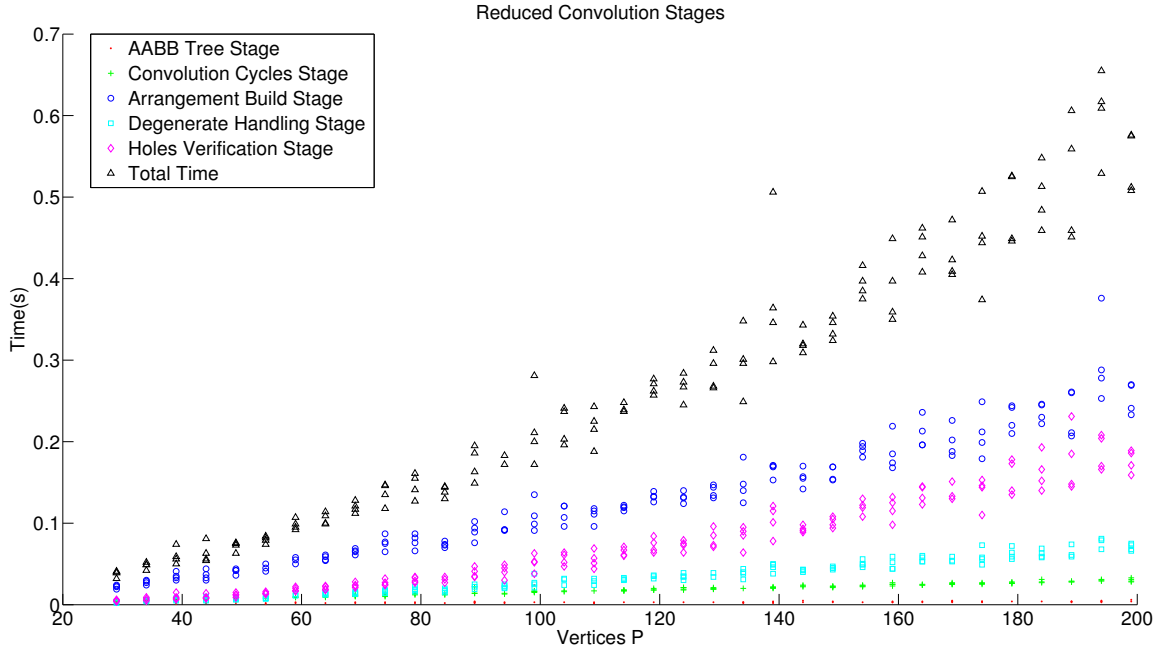


Figure 6.13: Stages of computation of RCA for the Simple and Simple Fixed Test.

of the input is random). Q remains with a fixed size of 6 vertices. Note that the number of vertices of P is about four times the number of teeth.

Figure ?? depicts the runtime of both algorithms as a function of the number of vertices of P . We can see that the runtime for both algorithms is at least quadratic in the number of vertices of P (and thus the number of teeth) as expected. Furthermore, RCA has a steeper curve than CWEIN, as we may expect since this case generates n^2 holes which need to be verified using a collision-detection test. Since the collision test takes at least $\Omega(\log(n))$ (traversing a tree once), the cost for the *Hole Verification Stage* is at least $\Omega(n^2 \cdot \log n)$, the result for this test shows that for the worst-case RCA performance suffers greatly as compared to CWEIN.

We further add plots showing the runtime for each different stage of computation for both algorithms. Figure ?? plots the times taken for each stage of CWEIN. As in the Simple and Simple Fixed example, we can see that *Convolution Cycles Stage* and the *Hole Verification Stage* seen linear and relatively small, although *Hole Verification Stage* should be quadratic. Not surprisingly, the *Arrangement Build Stage* runtime is quadratic in the number of teeth.

Figure ?? plots the times taken for each stage of RCA. As in the Simple and Simple Fixed example, we can see that *Convolution Cycles Stage* and the ABB-Tree building stage consume very little time. The degenerate handling stage seems to have more impact and have a peak about every other example. This could be explained by some artifact in the process of generating the fork which causes an increased amount of suspected degenerate edges (see Section ??). The arrangement build stage is quadratic but very slow as compared to the *Arrangement Build Stage* which performs a collision detection for $\Theta(n^2)$ holes. The collision-detection test runtime for each hole is between $\Omega(\log n)$ and $O(n)$. Thus the *Hole Verification Stage* runtime complexity is between $\Omega(n^2 \cdot \log n)$ and $O(n^3)$, and the plot in the graph fits this range.

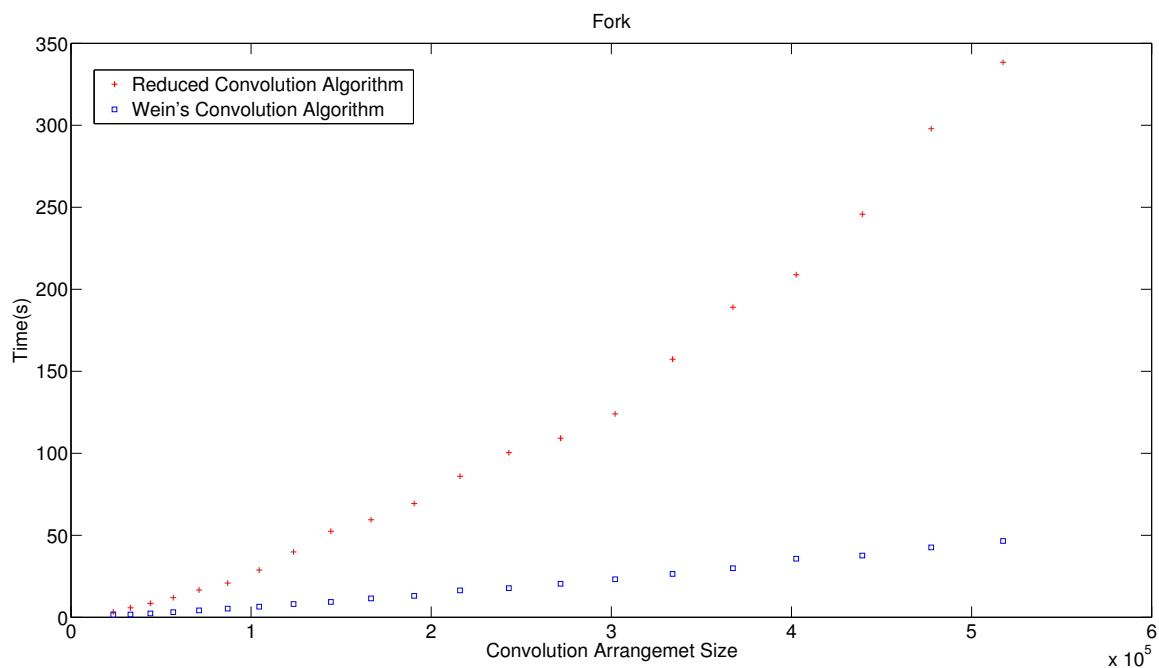


Figure 6.14: Benchmark for Fork Test.

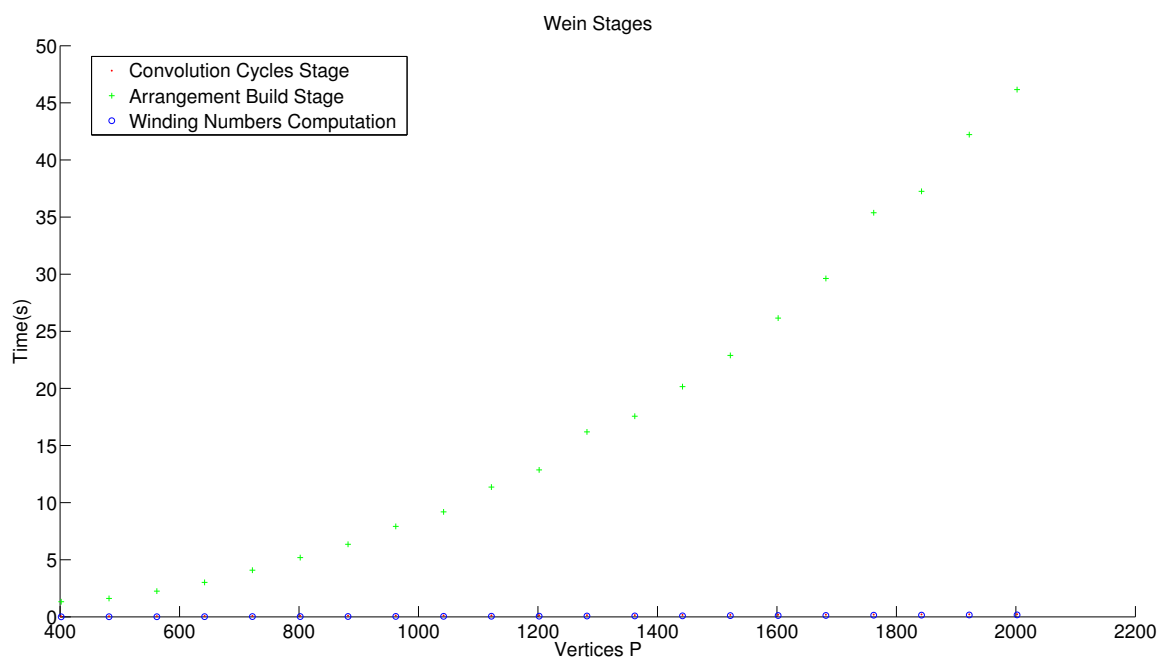


Figure 6.15: Stages of computation of CWEIN for the Fork Test.

6.2.6 Discussion

First we note that as a general behavior, for small input size (when the convolution arrangement has less than 2000 edges), no method is preferable over the other. For larger and random examples, where there are usually no holes, we see a linear dependency for both methods on the size of the full convolution arrangement. However, RCA has a lower slope than CWEIN. If we examine the runtime as a function of the number of vertices of the variable size polygon, we see a quadratic runtime dependency. This is not surprising

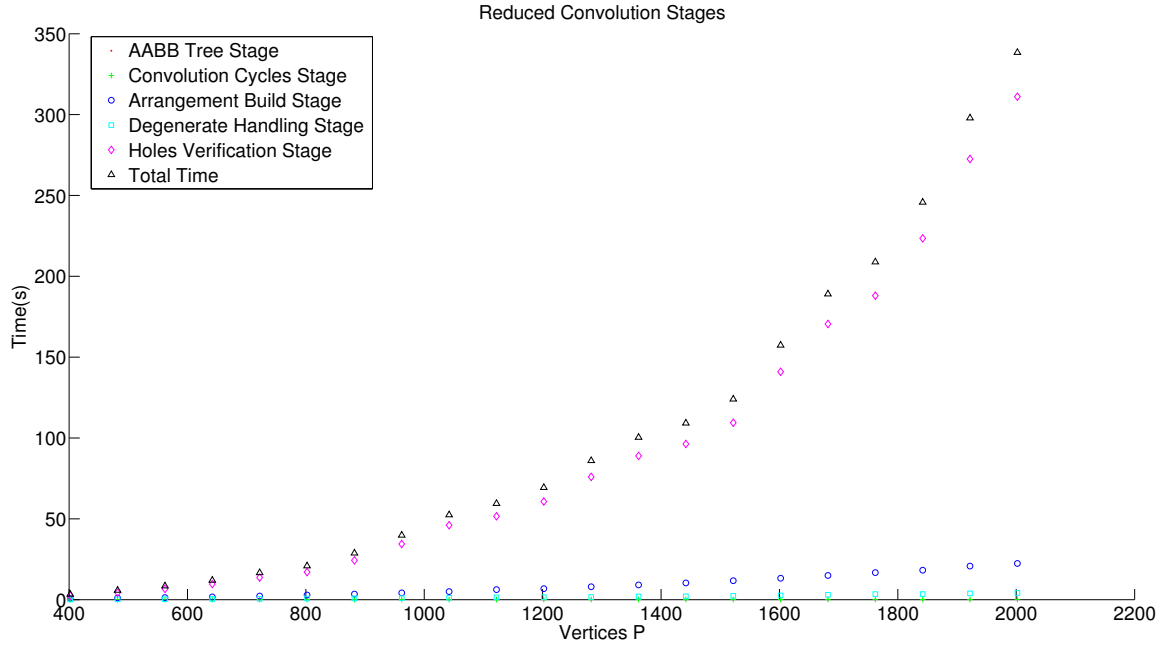


Figure 6.16: Stages of computation of RCA for the Fork Test.

because the size of the full convolution arrangement is usually quadratic in the number of vertices of the variable size polygon. Either way for the random cases RCA exhibits better constants for the runtime. For degenerate cases where we produce holes in the output, CWEIN is preferable since it does not require the costly collision detection.

The *Convolution Cycles Stage* is usually a non-dominant stage for both methods, however RCA is a bit faster for this stage for reasons given in Chapter ???. We have seen that the *AABB Tree Stage* is also negligible for large examples and can only hinder the results of small examples where one polygon is rather large and the other is very simple. This causes overhead for computing a hierarchy for the larger polygon which is then not used. The *Degenerate Handling Stage* is dependant on the topology of the convolution. Suspected degenerate edges and vertices causes expensive collision detection tests only for those features which pass the filters mentioned in Chapter ??? (when two edges in the result overlap, three segments intersects in their interior), but for the random cases, degeneracies usually do not occur. In this case, this stage is linear in the size of the arrangement and is not a dominant stage. The *Arrangement Build Stage* is a dominant step for both methods, and is proportional to the convolution arrangement size. Since the reduced-convolution arrangement is smaller than the full convolution, this step is faster by some factor for RCA over CWEIN. This factor increases as the input gets larger, more complex and has more reflex vertices, which causes the arrangement size to grow. The last stage is the *Hole Verification Stage*. For CWEIN the runtime for this stage is also linear in the convolution arrangement size. However, for RCA the runtime depends on the topology of the result, namely the amount of suspected hole loops in the arrangement. Therefore, for random cases the runtime is also linear in the convolution arrangement size, but for specific cases such as the fork example the runtime is higher.

6.3 Software Issues

During the benchmarks we encountered crashes in CWEIN for some of the examples, which were removed from the reported results. RCA suffered no crashes. Additionally, different bugs were found in the arrangement package. The first performance bug occurs when destructing a large arrangement (of size $2GB$ and up). We measured a quadratic factor in the runtime of the destruction as a function of the arrangement size. For arrangements that occupy less memory than $2GB$ there was no notable slowdown. Since the star shaped test contained large arrangements for CWEIN, we just reported the combined runtime of *Convolution Cycles Stage*, *Arrangement Build Stage* and *Hole Verification Stage*, instead of the total time. Note that this report also does not consider the output build time, which is however small. Another performance bug was found while removing edges from the arrangement, which caused another quadratic factor in the runtime for RCA during *Hole Verification Stage*. To overcome this, instead of removing edges of the arrangement we just mark the valid ones, and copy them to an output data structure. This operation is also not expensive and was done in the fork example where this bug was detected.

6.4 Robustness

In this section we show an example in which exact geometric constructions are mandatory for providing correct results. In this example we show topological errors that arise even when using the implementation given by Lien which uses multi precision arithmetic, but not exact geometric computation as CGAL does.

We compute the Minkowski sum of a diamond shaped polygon with a “U” shaped polygon; see Figure ?? and Figure ?. The “U” shaped polygon has the “roof” part extended inwards. This extension is designed such that the diamond width equals the distance between the two extensions. Consequently, the diamond can “slide” across this hatch and move along the inside upper edges of the “U”. This results in a degenerate edge and a hole in the Minkowski sum. We show how slightly perturbing the relevant vertices of the diamond and the “U” causes topologically incorrect results. Of course, we do not expect Lien’s implementation to find the degenerate edge.

We perturb the input slightly by the following scheme. Let ϵ_1 and ϵ_2 be very small numbers ($\epsilon_1 \ll 1$ and $\epsilon_2 \ll 1$) such that $\epsilon_1 > \epsilon_2$. In the base polygon we perturb the vertex in coordinates $(3, 6)$ to be $(3 - \epsilon_1, 6)$. In the diamond we translate the left vertex by $-\epsilon_2$ to the left extending the width of the diamond to be $1 + \epsilon_2$. The result of this manipulation is that the edges $(3, 6), (3, 5)$ and $(4, 6), (4, 5)$ which were parallel before the modification are no longer parallel. Furthermore, the distance between vertices $(3 - \epsilon_1, 6)$ and $(4, 6)$ is greater than $1 + \epsilon_2$ by construction. The distance between vertices $(3, 5)$ and $(4, 5)$ is 1 which is smaller than $1 + \epsilon_2$. Thus, while “sliding” the diamond across the edge $(3 - \epsilon_1, 6), (3, 5)$ downwards, there is a part where it could pass, at some point the width between the two edges will be exactly $1 + \epsilon_2$ and below this point sliding the diamond downwards will result in a collision. This means that in the Minkowski sum there should be features (two edges) representing the free space where the diamond can still slide. Our implementation correctly computes these features, but the implementation given by Lien does not compute them correctly, and they are missing from the output. This is verified both visually and by counting the number of output features given by the implementation. See Figure ?? for the resulting Minkowski sums of the two implementations.

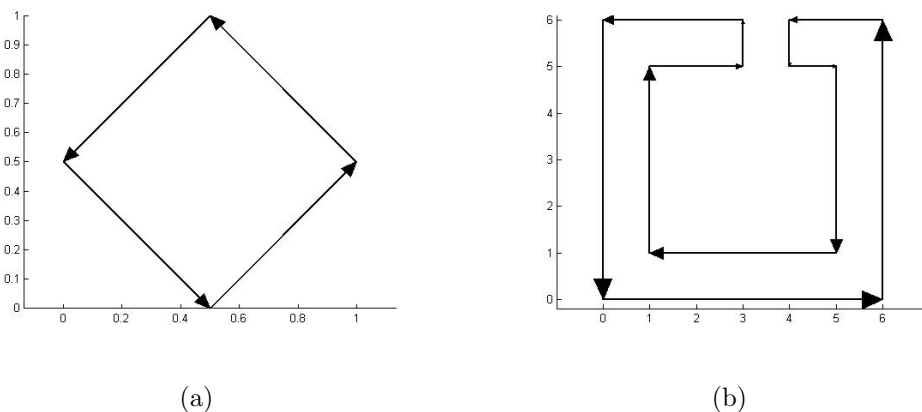


Figure 6.17: The diamond, with width of size 1 ??, and the “U” shaped polygon ??.

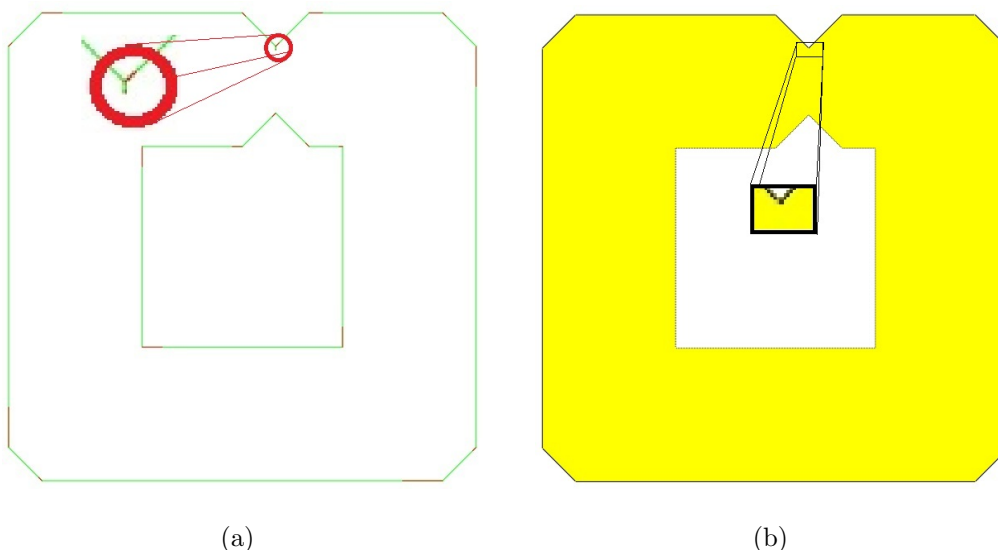


Figure 6.18: The result of running our implementation. The extra feature is marked inside the red circle ??, Lien’s implementation output for the same input ??.

Additionally, we increased ϵ_1 by a few orders of magnitude, but still keeping it relatively small (as compared to a length of 1). The result should be similar to the case described above, but the excluded edge features now grows very long and almost touch the connection points between the hole and the outer loop. This time Lien’s implementation mistakenly omitted the hole loop altogether, as seen in Figure ??.

6.5 Memory Usage

We now turn to discuss the memory consumption of RCA and CWEIN. Memory consumption limits the actual size of polygons these algorithms are capable of operating on, at least while keeping all the work in the computer’s main memory. Exceeding the computer’s available physical memory causes the operating system to use disk paging which involves slow input-output operations on the disks. In practice, this usually causes the system to almost freeze or the process to crash. Therefore, for these two methods, executing the algorithms on examples where the computation requires more memory than the available physical memory is not practical.

We measure the memory consumption of the two implementations at the point where the most memory is required, namely after computing the arrangement of the convo-

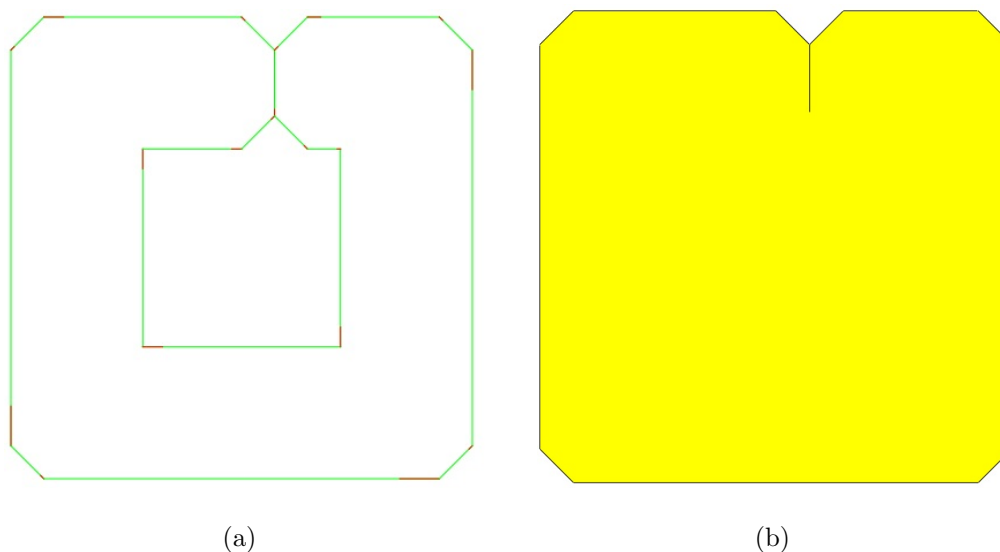
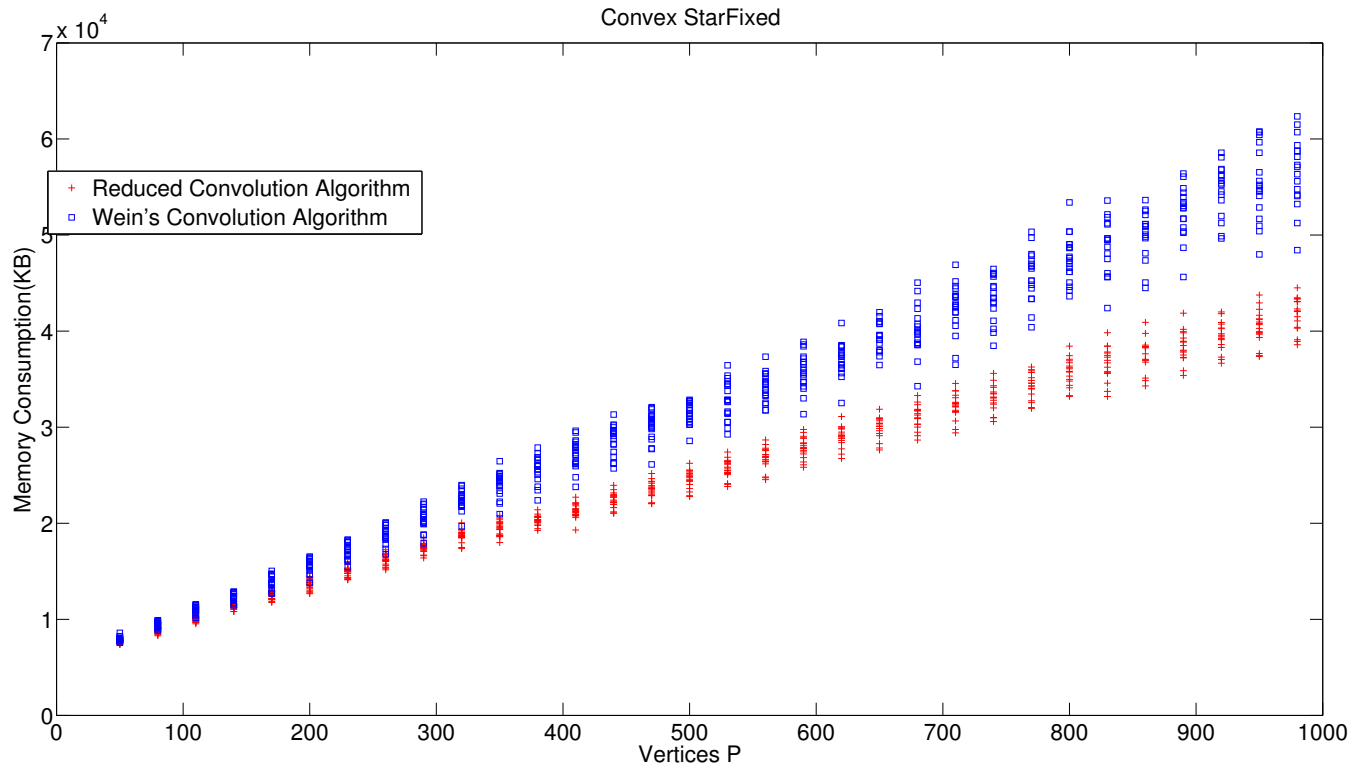


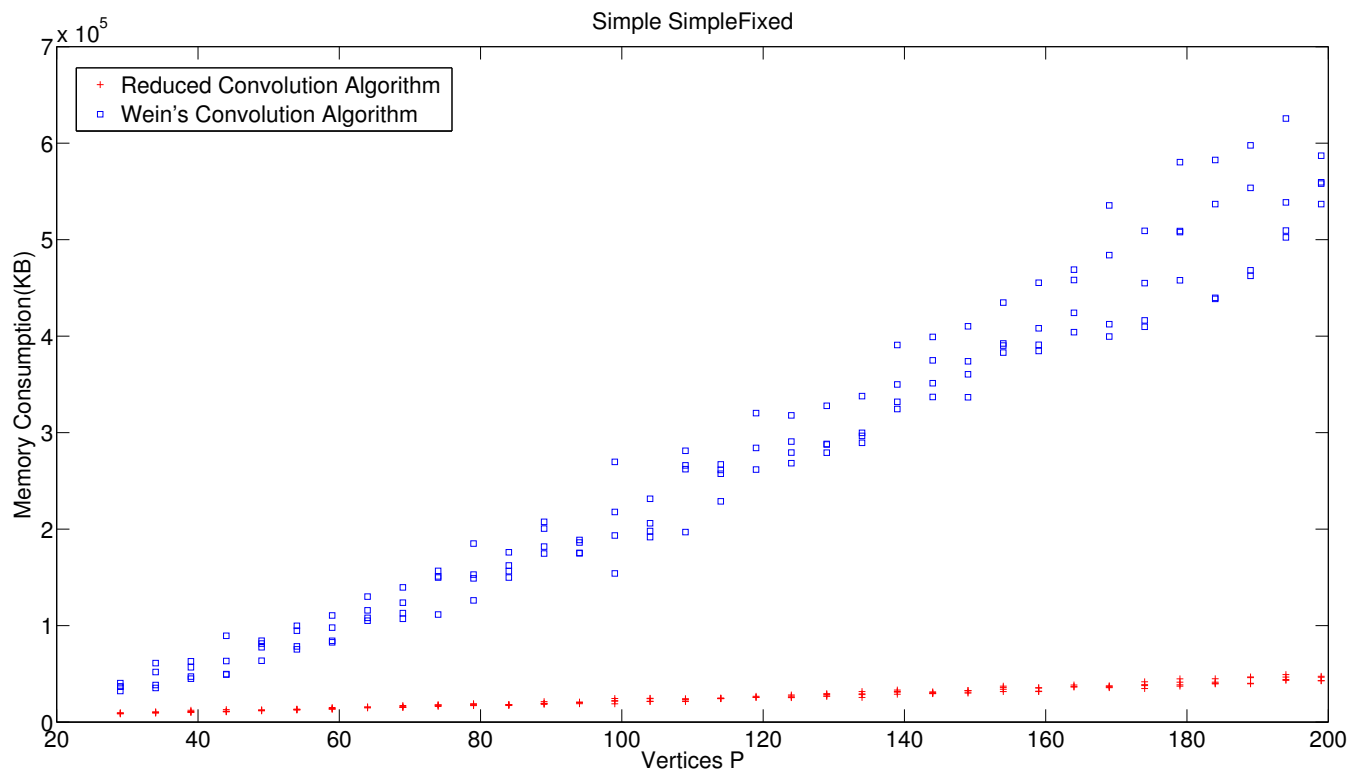
Figure 6.19: The result of running our implementation. The inside face is a hole. ??, Lien's implementation output for the same input. There is no inside face. ??.

lution. Notice that the arrangement complexity (number of segments after performing intersections) is a lower bound on the memory requirement of the arrangement.

We show the memory consumption for the Convex and Star shaped Fixed benchmark and the Simple and Simple Fixed benchmark. Refer to Figure ???. This example produces a relatively simple arrangement for both cases. We could expect that convolution cycles should appear for some of the reflex vertices of the star shaped polygon. Wein optimized his code to remove cycles that are precisely the translation of one of the polygons by some reflex vertex of the other polygon. Because of that, we can see that the memory consumption is quite similar for both methods, though RCA has a little advantage. For the Simple with Simple Fixed polygon test refer to Figure ??. In this case we can see that the memory consumption of RCA increases much more slowly than that of CWEIN. We could expect the increase of CWEIN to be quadratic in the input size, however this is hard to determine due to the variations in the data. We can however see that CWEIN consumes more memory than RCA with a factor that grows with the input complexity.



(a)



(b)

Figure 6.20: Memory consumption after building convolution arrangement: ?? the Convex and Star-shaped Fixed test and ?? the Simple and Simple Fixed test.

7

Implementation Details

In this chapter we briefly describe the implementation details of the reduced convolution algorithm (referred to as RCA) using CGAL. For succinctness, we only provide a survey of our methods, while providing references for further study. In Section ??, we describe the concepts that enable the implementation of RCA in CGAL. Section ?? presents several implementation issues and optimizations.

7.1 Programmatic Background

RCA is implemented in C++ using the CGAL [?] library. CGAL is a library that enables the exact implementation of geometric algorithms and contains a plethora of algorithms by itself. It follows the generic programming paradigm [?] allowing the user to customize different aspects of the algorithms while changing little amount of code. Furthermore, the user of the algorithms may choose to perform the computation using methods that are either exact and usually slower or inexact and usually faster. CGAL is developed in collaboration between several research institutes in Europe and Israel.

Generic programming [?] is a paradigm whose algorithms are described in a general manner operating on any concrete type which implements them. The set of requirements on a specific input type is referred to as *concept*. A type that implements these requirements is a *model* of the *concept*. Generic programming is achieved in C++ using the template mechanism. A trait class [?] is a design pattern, which allows to associate additional information to a type during compile-time. For more information about generic programming can be found in [?]. CGAL's design follows the generic programming paradigm and uses traits classes to act as a wrapper that defines the implementation for concrete *models*. The traits classes are used to separate different aspects of geometrical algorithms, such as separating the topology from the underlying geometric computation. For more details about generic programming in CGAL see [?].

Algorithms in computational geometry usually assume general position and the “real RAM” model of computation [?], where computations always return precise results and the computation time of an arithmetic operation is constant. In practice however, the floating point representation is does not produce precise results and the input may not

be in a general position. This could cause topological errors in the results, resulting in misplaced features, intersections that should not occur and so forth [?]. Of the methods available in the literature for handling these issues, CGAL uses exact geometric computation [?]. This means that geometric predicates (such as asking if a point is above or below a line) are evaluated with the required precision to ensure their correctness. This operations however do not take constant time. Construction of new geometric objects is done in a way that enables the verification of the correctness of predicates operating on them. CGAL uses efficient software libraries such as GMP¹ and MPFR² as building blocks for this mechanism. Finally, it uses a *concept* called *Kernel* to allow the user to decide whether to use exact computing, floating point types and several other optimization features combining the two. Refer to [?] for additional details.

The implementation of RCA and CWEIN relies heavily on the “2D Arrangements” package [?], representing planar subdivisions (see Definition ??) in CGAL. The “2D Arrangements” package allows to build the planar subdivision for different types of curves. The curve type is chosen by using the appropriate traits class. A topology traits class which models the concept of `ArrangementDcel` [?] (which is a data structure representing a planar subdivision) represents the topological relationships between arrangement cells, thus allowing for example, to embed the arrangement on a surface. The “2D Arrangements” package contains algorithms that operate on arrangements such as sweep-line and point-location.

7.2 Implementation of RCA

In this section we discuss the main issues of implementing RCA using CGAL. There are many delicate points affecting the runtime of an algorithm implemented in an exact manner. We describe some of the decisions taken in our implementation and deal with some of the finer details. We show the main design issues and data structures, which allow for an efficient implementation. Specifically, in Section ?? we discuss how we used the arrangement data structure for RCA. Section ?? shows how we implemented collision detection using `AABBTree`.

7.2.1 Using the Cgal Arrangement Data Structure

The most important data structure of RCA is the `Arrangement_with_history_2`. This arrangement is extended in several ways, which we describe below, in order to store additional information, and perform operations more efficiently. This arrangement stores the history of the curves that the arrangement is built from, for our case the convolution segments. By this we can detect which convolutions segments created a specific vertex or edge and the directions of these segments.

We use the arrangement to store the planar subdivision induced by the reduced convolution. We then mark the specific edges which are on the boundary of the Minkowski sum. Isolated vertices are also a feature of the arrangement, which is used to store degenerate vertices (since degenerate vertices are always isolated). In order to mark the features of the arrangement that are part of the Minkowski sum boundary, we extend

¹<http://gmplib.org>.

²<http://www.mpfr.org>.

the DCEL half-edges with additional information stating whether they are part of the boundary or a degenerate feature.

In most cases (see Chapter ??) the most time consuming part of RCA is the insertion of the reduced convolution segments into the arrangement. The arrangement uses the sweep-line algorithm in order to compute the intersections between the input segments and build the DCEL. Naively, one would just insert all the input segments into the sweep algorithm. However, since we deal with segments whose source are translations of segments of two simple polygons, we know immediately that some intersections are not possible. We created a custom traits class, which enables us to use prior information to filter out expensive intersection tests, which we describe next.

In particular recall that a segment of the convolution is represented by a pair of states, for example the pair $(i, j), (i + 1, j)$ describes the segment s_i that lies between the vertices p_i and p_{i+1} of p translated by the vector v_j from the origin to vertex q_j of Q . In this case, since P is simple no other segment of P translated by v_j intersects s_i . We add this information to the arrangement traits class by augmenting the state information for the segments of the arrangement. The traits class checks the state information while performing intersection tests, and does not check for intersection among such segments.

The exact computation mechanism uses many lazy evaluation tricks to perform exact computation only when they are necessary. For the segment s_i and the segment s_{i+1} that lie between the vertices p_{i+1} and p_{i+2} of P , p_{i+1} is a joint vertex. If both segments are translated by the same amount t , a naive computation will detect that they intersect at $p_{i+1} + t$, which is a fact we know in advance, triggering expensive exact computations. Fortunately, the scheme we have just described in solves this issue as well.

7.2.2 Collision Detection via Bounding Volume Hierarchy

In this section we describe the implementation of detecting the interior collision of two polygons P and Q using CGAL's `AABB_Tree` package [?]. For the algorithmic details, and how to use collision detection for deciding CDP, see Section ??.

CGAL's `AABB_Tree` package is originally designed for geometric entities in three-dimensional space. The `AABB_Tree` contains bounding boxes and primitives. The primitive types can represent points, segments, triangles and so forth. The `AABB_Tree` performs queries on an input primitive such as a segment or triangle and detects intersection, reports the intersection points and computes the distance (i.e., closest primitive to the query primitive) with respect to the geometric entities stored in the `AABB_Tree`.

We created a modified version of this package, which can answer whether two polygons' interiors intersect. Since RCA is implemented for planar polygons, we define the package types to be planar. For a given `AABB_Tree` T_1 we define a query, which accepts another `AABB_Tree` T_2 and a translation vector v and decides whether T_1 and $T_2 + v$ intersect in their interior.

Class Design of `AABB_Tree` Package

The `AABB-tree` is represented by a template class `AABB_Tree`. This class defines the queries that can be performed on the tree. It has a member class `AABB_Tree_Node`, which represents a node in the tree, in this case, the root. The nodes contain references to their children and methods that allow the user to traverse their subtree. The nodes hold their bounding boxes as well, which contain the region of space occupied by their

entire subtree. The `AABB_Tree` contains a traits class type, `AABB_Traits`, which is given as a template parameter. The traits class defines the actual geometric entities the tree is composed of and implements the geometric predicates as required by the tree.

RCA Modifications of `AABB_Tree` Package

We now describe how we modified the `AABB_Tree` package to enable it to answer the query described above. First though, we show how we changed the package to work on geometric entities in the plane. Then we describe the features that were added in order to perform our query.

We implemented a primitive class `AABB_segment_2_primitive`, which defines the geometrical entities for a point and segment in the plane. The geometric traits, a template parameter for `AABB_segment_2_primitive`, provides the actual types definitions for those geometrical entities concepts. The class `AABB_traits_2` defines the mapping between the three-dimensional geometrical entities and predicates as defined by the original `AABB_Tree` package to the actual two-dimensional implementation. The `AABB_segment_2_primitive` is given to `AABB_traits_2` as a template parameter, which defines the actual primitive type.

For performing a query that detects an interior intersection between two polygons, one of which is translated, we made the following changes to the package: We introduce a new method to `AABB_Tree`, `join_traversal`. This method accepts a second tree as a parameter, and traverses both trees recursively as described in Section ???. The class `AABB_Node` now includes the actual implementation for this traversal and is invoked when the `join_traversal` method is called.

The modified `AABB_Tree` has to support a query in which one of the polygons is translated. For each query, an instance of `AABB_Traits` is constructed with the translation vector for the query. The second operand to `join_traversal` is the designated tree to be translated by the translation vector. The class `AABB_Traits` is responsible for implementing the intersection tests. It uses the translation vector to translate every geometric entity which originates from the tree which is designated to be translated. There are two types of geometric entities stored in the tree, primitives and bounding boxes. Translating a primitive uses CGAL's exact computation. A translated bounding box is computed using interval arithmetic, such that it contains the correct translated original bounding box.

When computing the intersections of two segments, we may encounter the case of **weak intersection**, as described in Section ???. In order to deal with this case, the information about the neighboring segments with regard to the original polygons has to be retrieved. We store the original polygons in the traits class and keep references from the primitives in the tree to the original segments of each polygon.

8

Conclusions and Future Work

This thesis discussed different aspects of convolution-based methods for computing the Minkowski sum of two polygons in the plane. We examined the definition of the convolution set and convolution cycles. We proved that convolution cycles exist, and provided an optimal algorithm to trace them. Then, we showed that the winding numbers as defined by those cycles are positive only in the faces interior to the Minkowski sum. If we chose a slightly different set of segments for the convolution, the winding number of a point p represents exactly the number of connected components of the intersection of P and $-Q + p$ where P and Q are simple polygons. This result is the “Convolution Theorem” stated in [?], which is the basis for Wein’s convolution-based method correctness.

It is left to generalize our proof for the case of non-simple polygons as we began in the end of Chapter ???. Another possible continuation of this work is linking it to the general framework of constructible functions [?], which should provide an alternative proof of the convolution theorem for higher dimensions.

We implemented a robust algorithm for reduced convolutions, which handles low dimensional features in the output. The algorithm shows preferable results for larger polygons cases both in runtime and memory consumption. We used a bounding volume hierarchy to make multiple queries of collision detection faster, which cause our reduced convolution implementation to perform better than Lien’s implementation for some cases. Further investigation that compares our reduced convolution implementation to Lien’s implementation should be performed to assess how the two methods compare against each other.

We discussed a different way to check whether a point is interior to the Minkowski sum by using a ray shooting and a cutting tree data structure (which stores mn convolution segments). This method guarantees a theoretical sub-linear query time, but hard to implement in practice. A heuristic approximation which uses an interval tree did not prove better than the bounding volume hierarchy. Additional tested method was the sweep-line collision detection, which yielded the worst running time in practice.

We saw that our reduced convolution implementation had problems in the fork example where it needs to perform multiple collision detection tests. A parallelization scheme could be considered as each collision detection test is independent of the other. This

process may be even parallelized using a GPU.

The monotonic convolution [?] should be implemented in CGAL as well and tested against our reduced convolution implementation and the implementation given by Lien. Unfortunately, it is still unknown how to extend the monotonic convolution to three-dimensional space.