UNIVERSITY OF MIAMI

ROBUST EXPLICIT CONSTRUCTION OF 3D CONFIGURATION SPACES USING CONTROLLED LINEAR PERTURBATION

By

Steven Cy Trac

A DISSERTATION

Submitted to the Faculty
of the University of Miami
in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

Coral Gables, Florida

December 2008

UNIVERSITY OF MIAMI

A dissertation submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

ROBUST EXPLICIT CONSTRUCTION OF 3D CONFIGURATION SPACES USING CONTROLLED LINEAR PERTURBATION

Steven Cy Trac

Approved:

Victor Milenkovic, Ph.D. Associate Professor of Computer Science	Terri A. Scandura, Ph.D. Dean of the Graduate School
Geoff Sutcliffe, Ph.D. Associate Professor of Computer Science	Elisha Sacks, Ph.D. Professor of Computer Science Purdue University
Brian Coomes, Ph.D. Associate Professor of Mathematics	Hongtan Liu, Ph.D. Associate Professor of Mechanical and Aerospace Engineering

Vita

Steven Cy Trac was born in Baltimore, Maryland, on July 12, 1981. His parents are Tan Ba Trac and Judy Trac. He has one younger brother, Ivan Cy Trac. He received his elementary education at Sabal Palm Elementary School and his secondary education at Charles W. Flanagan High School. In September 1999 he entered the University of Miami from which he was graduated with a BS degree *cum laude*, double majoring in Computer Science and Mathematics, in May 2003.

In August 2003 he was admitted to the Graduate School of the University of Miami, where he was granted a Master's degree in Computer Science in December 2005.

In Spring 2006 he was admitted to the Graduate School of the University of Miami, where he was granted a Doctor of Philosophy degree in Computer Science in December 2008.

Permanent Address: 201 SW 98 Terrace, Pembroke Pines, Florida 33025

TRAC, STEVEN (Ph.D., Computer Science)

Robust Explicit Construction of 3D Configuration Spaces Using Controlled Linear Perturbation (December 2008)

Abstract of a dissertation at the University of Miami.

Dissertation supervised by Professor Victor Milenkovic.
No. of pages in text. (103)

We present robust explicit construction of $3D$ configuration spaces using controlled linear perturbation. The input is two planar parts: a fixed set and a moving set, where each set is bounded by circle segments. The configuration space is the three-dimensional space of Euclidean transformation (translations plus rotations) of the moving set relative to the fixed set. The goal of constructing the $3D$ configuration space is to determine the boundary representation of the free space where the intersection of the moving set and fixed set is empty. To construct the configuration space, we use the controlled linear perturbation algorithm.

The controlled linear perturbation algorithm assigns function signs that are correct for a nearly minimal input perturbation. The output of the algorithm is a consistent set of function signs. This approach is algorithm-independent, and the overhead over traditional floating point methods is reasonable.

If the fixed and moving sets are computer representations of physical objects, then computing the configuration space greatly aids in many computational geometry problems. The main focus of computing the configuration space is for the path planning problem. We must find if a path exists from the start to the goal, where the fixed set is the obstacle, and the moving set is the object trying to reach the goal.

Dedication

This thesis is dedicated to my parents, Judy Trac and Tan Trac, and my brother, Ivan Trac. This is also dedicated to my late grandparents who passed away recently in my life, as well as my late Uncle Sing.

Acknowledgements

I would like to thank Dr. Victor Milenkovic, my advisor, for giving me the chance and opportunity to work under him. It has been a privilege to learn so much about computational geometry, about his research, and just general knowledge of everything with science. I want to thank Dr. Milenkovic also for funding me through out my years as a graduate student through his own grants from the National Science Foundation.

I would like to thank Adam Pease for funding me in the 2007-2008 school year. I want to thank the Dr. Hüseyin Koçak and Dr. Dilip Sarkar for funding me as a Teaching Assistant during my last semester of Fall 2008 as a Computer Science Ph.D. student. I would like to thank the generous donors to the Department of Computer Science for funding me as well.

I would like to thank Dr. Geoff Sutcliffe, Dr. Elisha Sacks, Dr. Brian Coomes, and Dr. Hongtan Liu for serving on my dissertation committee and for their comments and corrections. I would also like to thank Dr. Sutcliffe for allowing me to join his research group here at the University of Miami. I want to also thank my friends here at the University of Miami and my friends abroad who have kept me sane through out these past years. Special thanks to Maurice Duque for helping me with my dissertation defense and to Adam McMahon for creating some wonderful $3D$ images for me to use in the dissertation.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the field of robotics, the *configuration space* (or C-Space) is an important concept for aiding with *path planning*. The path planning problem, also known as the "piano movers problem", is the task of finding a collision free path for a robot around a fixed rigid obstacle. For example, consider parking a robot inside a garage. The starting position of the robot is outside the garage, and naturally the goal position is inside the garage. If there is a collision free series of motions (e.g., translations, rotations, and simultaneous translations and rotations) of the robot from start to goal, then the path exists; path existence means the robot fits inside the garage. If no path is possible then the robot does not fit inside the garage.

The current approach to solve the path planning problem is by using sampling-based algorithms. Sampling-based algorithms use the *pose* of the robot to generate the path, where a pose is the position and orientation of the robot. Sampling algorithms generate many poses of the robot, and the poses that overlap the garage are discarded,

see Figure 1.1. From the poses that do not overlap the garage, a path is generated from connecting poses from the start to the goal, see Figure 1.2. The connection of poses is done using *interpolation*, a method of constructing new poses between the range of two given poses by fitting a function which closely fits the two poses. The number of sampling poses is important because as the number of poses grow, the probability of finding a path approaches 1. This property of sampling-based approaches is called *probabilistic complete*.

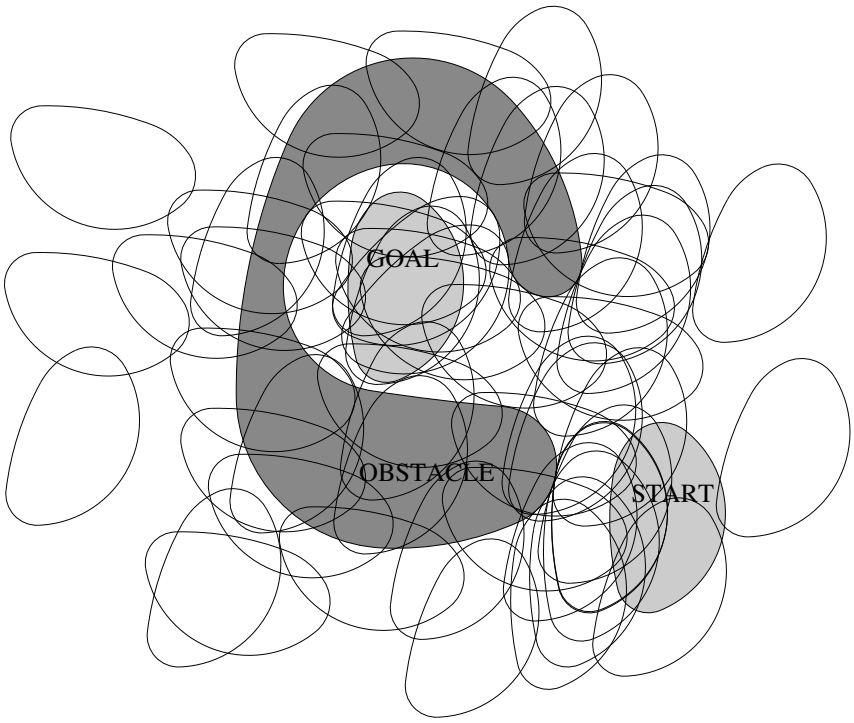


Figure 1.1: Sampling many poses of the robot. The obstacle region represents the garage. The start and goal regions represent the start and goal pose of the robot.

The main problem with sampling-based algorithms is narrow channels in the obstacle. If a path exists through a narrow channel, the narrow channel forces the robot

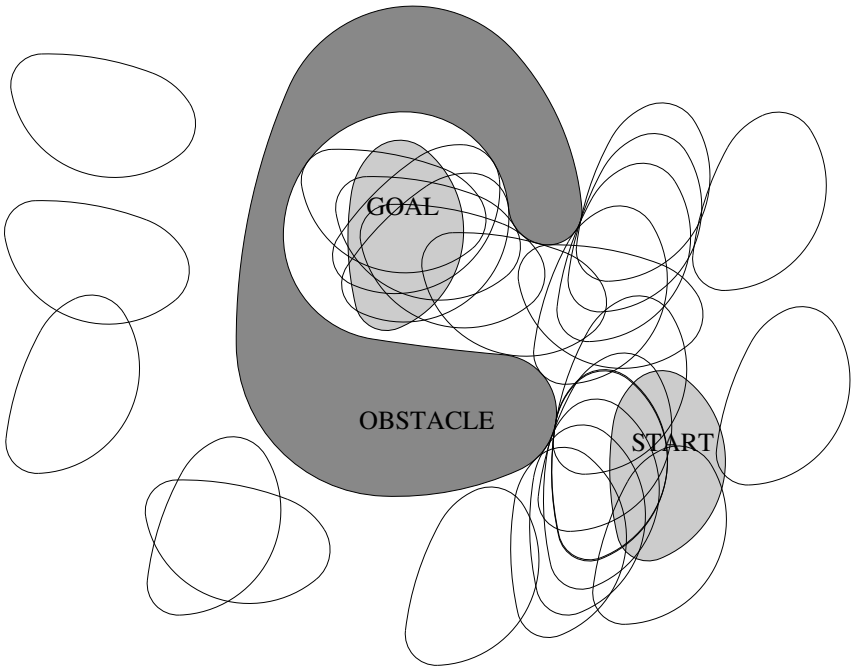


Figure 1.2: Sampling and keeping non-overlapping poses of the robot. The obstacle region represents the garage. The start and goal regions represent the start and goal pose of the robot.

to come very close to or otherwise touch the obstacle. Finding a path through a narrow channel requires many sample poses near the narrow channel. An insufficient number of sample poses might result in missing a solution that gets the robot into the garage. Another problem with sampling-based algorithms is that the interpolation of two poses might overlap with the garage. That is, an interpolated pose overlaps with the garage. A path with such an interpolated pose is an incorrect path.

The problems with sampling and interpolation stem from the difficulty of computing with computer arithmetic. For example, consider finding the square root of 2. We can sample many numbers (1.3, 1.4, 1.5, etc.) to find the square root of 2, but

no one would find the square root in such a fashion. Using computer arithmetic, we would call the standard square root function in LibM math library in C. The result of calling it as a standard function is machine accuracy of up to 16 digits by IEEE 754 standard [24].

What this dissertation offers is the first solution to the problem of the same nature for constructing the configuration space to perform path planning. Instead of sampling, we want a solution to use standard numerical libraries such as the LibM math library in C, but also more complex libraries like LAPACK for solving linear algebra problems or CPLEX for solving Linear Programs (LP). We would solve the problems by calling the standard libraries and not by sampling. This gives an answer with *floating point accuracy*. The algorithm can guarantee if a solution exists or not. Such algorithms are called *resolution complete*.

Floating point accuracy means that if the algorithm says "yes" a path exists or "no" a path does not exist, then the answer is true for a small perturbation of the input. This is called *standard backward analysis*. Another example of standard backward analysis is computing the eigenvalues of a matrix using LAPACK. The eigenvalues returned are not really the eigenvalues for input matrix, but instead the eigenvalues returned are the eigenvalues for a matrix "very close" to the input matrix. What we mean by "very close" is floating point accuracy. So if there is a path, then it is true for a tiny perturbation of the robot or garage. We would "buff" the robot slightly against the garage, but that is acceptable since in most manufacturing such as Computer Aided Design (CAD) or Computer Aided Manufacturing (CAM), only

up to 6 digits of accuracy is needed. If we can get up to 16 digits of accuracy, that is more than acceptable.

The difficulty of such an approach, as we explain through out this dissertation, is that we have a set of functions, and we need to make sure the perturbation maintains the set of signs of the functions. The signs of a set of functions are important, because computational geometry algorithms need to make decisions based on the signs of the functions on the input parameters. For the robot/garage example, the input parameters are the values that describe the shape of the robot and the garage.

The fundamental robustness problem in computational geometry is when the actual value of a function $f(x)$ is at or near zero, $|f(x)| < \epsilon$. Functions at or near zero are called *non-robust* functions. If there are non-robust functions, we need more than machine accuracy to determine the sign of the function. If there are many these non-robust functions, then it is hard to determine the signs of the functions for a perturbation, because a perturbation might flip the sign of a function.

The rest of the dissertation is structured as follows. In the rest of this chapter we explain prior work, motivation and goals in Sections 1.1 and 1.2, respectively. In Chapter 2 we present the C-Space construction. In Chapter 3 we illustrate how to construct arrangements that represent the C-Space. We discuss the path planning algorithm in Chapter 4. In Chapter 5 we explain the CLP algorithm. In Chapter 6 we describe the implementation and explain experimental results. Finally we conclude and discuss future work in Chapter 7.

1.1 Prior work

In computational geometry, the key issue in geometric computations is to achieve consistent evaluation of the signs of a set of functions [23]. Functions that return signs are called *primitives*. Primitives answer questions of "yes" and "no" consistently to achieve robustness. There are several approaches for robust geometric computations when it comes to computer arithmetic. The three main approaches that we discuss are the exact approach, inconsistency sensitive approach, and controlled perturbation approach. We discuss each approach in turn. We then discuss some prior work that has been done in path planning with configuration spaces.

1.1.1 Exact methods

Exact methods are robust because they compute the combinatorial part exactly. Using exact methods, the answer returned is true for the exact input. Exact methods employ custom geometric algorithms, constructive root bounds, and floating point filters to compute correct combinatorial structures. For exact methods, there exists algorithm libraries that provide users access to many efficient and reliable geometric algorithms such as LEDA [31, 29], CGAL [17, 8], and EXACUS [6, 16]. LEDA and CGAL, for example, can compute arrangements of line segments exactly via generalizations of Bentley-Ottmann's line sweep algorithm [5] that employ filtered rational arithmetic. EXACUS deals mainly with curves and surfaces, such as computing arrangements of curves in the plane. These libraries are easy to use and the numerically exact geometric algorithms are flexible and extendible to suit a users needs.

Wein [47] applies exact methods to Minkowski sums of polygons. Yap et al. [49, 50] survey exact approaches on arrangements [7, 12, 19, 26, 36, 46, 48]. The latest exact arrangement algorithm, by Eigenwillig and Kerber, computes exact and efficient two dimensional arrangements of arbitrary algebraic curves [11].

Exact methods give exact answers, but there are problems with this approach. The arithmetic operations are not constant time; the cost increases with the algebraic degree of expression and the cascading leads to rapid growth. The exact answers given by exact methods give hundreds and hundreds of digits of machine accuracy. For manufacturing, we need only 6 digits of accuracy, so hundreds of digits of accuracy are excessive. For the purposes of configuration space construction, we need a faster approach and not necessarily so many digits of accuracy.

Degenerate primitives also pose a problem. A degeneracy is when a primitive evaluates to zero, so there is no sign for the function. Computational geometry algorithms often assume general position of the input, e.g., no three points are co-linear, no four points are co-circular, or no three lines intersect in a common point. An exact approach must handle the excluded cases, thus leading to tedious special cases for each specific algorithm. Because of these reasons exact methods are not considered.

1.1.2 Inconsistency sensitive algorithms

Milenkovic and Sacks expressed the running time and error in terms of the number of *inconsistencies*, and they developed an inconsistency-sensitive algorithm for the construction of an arrangement with semi-algebraic curves [34]. This work was extended to include robust topologically invariant set operations [43].

The problem with this approach is that the algorithm is very algorithm specific. That is, we must deal with degeneracies case-by-case for each algorithm, and we must determine all potential degeneracies ahead of time. Inaccurate computation can create degeneracies where none existed. They can also eliminate degeneracies that were there. Consequently, more stringent error analysis must be performed to ensure robustness. The first attempt to construct the configuration space was to use an inconsistency sensitive approach, but we had no hope of proving the error bound. Error analysis became daunting.

1.1.3 Controlled perturbation

Perturbation methods assume that the input is not precise (due to measuring errors or approximate modeling), so therefore the output need not be precise. Perturbation methods redefine the problem so that the troublesome inputs (degeneracies) are changed to less troublesome ones. Another perturbation approach is to redefine what the result should be, so that bad outputs seem "correct". Seidel gives an overview on some perturbation approaches [41]: a simulation of simplicity scheme by Edelsbrunner and Mucke [10], a symbolic scheme by Yap [51, 52], a linear scheme by Emiris

and Canny [13, 14, 15], and a random scheme by Michelucci [33]. However, all these techniques require exact arithmetic, which we avoid for reasons previously stated.

Halperin and Leiserowitz [21] compute arrangements of circles by a perturbation method that does not need to compute the correct combinatorial structure. They call it *controlled perturbation*. To solve precision problem and degeneracy problems they perturb the input so that each floating point computation is guaranteed to be correct with respect to the perturbed circles. They perturb each input parameter uniformly by a δ interval. Their goal is to cause all primitives that are evaluated to be sufficiently far away from zero so that they can safely determine a "yes" or "no" answer (to achieve robustness). Their method is useful when the perturbation is much less than the manufacturing accuracy, although the output may be incorrect for the input circles.

The controlled perturbation scheme is applied to some computational geometry problems [18, 22, 38]. Mehlhorn et al. extend the controlled perturbation approach further [32]. They prove that controlled perturbation and guarded tests are a general conversion strategy for a wide class of geometric algorithms. They analyze by deriving quantitative relations between the amount of perturbation and the precision of the approximate arithmetic.

Although controlled perturbation achieves polynomial running time, the amount of random perturbation is unacceptable. That is, the robot and garage in the original example would deform greatly. The problem is it needs to pick a perturbation size $\delta = O(\sqrt[d]{\mu})$ for μ the floating point accuracy and d the highest degree primitive. The

method must use the worst-case δ to ensure correctness. In order to guarantee the shape of the robot and garage, there would be much perturbation required. This approach might give up to 3 digits of accuracy, which is unsuitable for most manufacturing. Another problem is that random perturbation invalidates equality constraints. Because we use equality constraints in the construction of the configuration space, invalidating equality constraints can pose problems.

1.1.4 Path planning with configuration spaces

Configuration space computation is a general approach for the classical problem of path planning a robot through an obstacle. One of the main approaches for computing configuration spaces is with arrangement construction [3, 39, 40]. Lazano [30] uses configuration spaces with spatial planning when objects and obstacles are polygons or polyhedra. The problem is divided into finding a safe position (find-space) and finding a safe path (find-path).

Avnaim et al. explicitly compute the configuration space of polygonal objects exactly [4]. The representation of the *free space* is a connectivity graph, whose nodes are faces on the boundary of the free space and edges connect adjacent faces. They studied two types of motion in detail: motion in contact and motion in free space. Motion in contact, meaning the robot will always be in contact with the obstacle, uses the connectivity graph to find a path from start to goal. Motion in free space, meaning the robot is not in contact or colliding with the obstacle, is done by first decomposing the free space into a volumetric representation. After the decomposition

of the free space, the nodes in the connectivity graph are the volumed cells and edges connect adjacent volumed cells.

Varadhan et al. compute the configuration space by approximation and not arrangement building [44]. They argue that the arrangement construction is the major bottleneck, because the computation is prone to problems in accuracy and robustness [23]. Their approach of approximation is done by sampling and reconstruction, where accuracy of the representation depends on the rate of sampling, as we described earlier in the chapter. The limitations in this approach is that free space must not have any tangential contacts. This happens when a robot must touch the obstacle in order to find a path to the goal. They have difficulty dealing with tangential contacts if they do occur.

For path planning algorithms, one sampling approach by Kuffner is to incrementally build two rapidly-exploring random trees (RRTs) rooted at the start and the goal configurations [27]. The trees explore the immediate space around and advance with a greedy heuristic. They can handle rigid objects in $2D$ and $3D$.

Another sampling approach is using classical grid search with probabilistic road maps (PRMs) [28]. The classical grid search is extended using subsampling for collision detection. Kavarki et al. also uses PRMs, and introduces a learning phase and a query phase [45]. They construct a graph whose nodes correspond to collision-free configurations and whose edges correspond to feasible paths between these configurations.

1.2 Motivation and goals

For computing the C-Space, we want an approach that is fast and consistent, so we did not choose exact methods. We want to introduce implicit parameters and equality constraints and more than 3 digits of accuracy, so we did not use the controlled perturbation method. We did not choose inconsistency sensitive approach because error analysis proved too troublesome.

Instead of these approaches, we use a technique developed by Milenkovic and Sacks called *controlled linear perturbation* (CLP), which an earlier version of the technique was tested by Puzis in 2007 [37]. Puzis applied the CLP technique to computing an arrangement for univariate polynomials. Since then, Milenkovic and Sacks have continued to improve the technique, and we are currently using the latest version.

With the use of CLP, we can implement the configuration space algorithm without considering degeneracies. The CLP receives as input the set of functions and input parameters used in the C-Space construction. In return, the CLP returns a set of *consistent* signs for those functions. Consistent means the signs are correct for some perturbation of the input. The CLP algorithm is fast and its overhead over the floating point approach is very reasonable. It also allows for implicit parameters and equality constraints, something not possible in the controlled perturbation scheme. We give a detailed description of this technique in Chapter 5.

For path planning, CLP is helpful for it eliminates the need for sampling. It addresses the problems of accuracy and robustness brought up by Varadhan et al. [44]. It also addresses the shortcomings of their approach, in that we can handle tangential

contacts. Therefore we can handle obstacles with narrow passages that lead to the goal, forcing the robot to be in contact with the obstacle.

For representation of the obstacle and robot region, we want the representation of the regions to include circular arcs, while Avnaim et al. handle only polygonal regions [4]. Instead of a complicated volumetric representation like Avnaim et al., we want a simpler approach to finding a path in free space. We describe such an approach in Chapter 6.

Chapter 2

Configuration Space

In this chapter, we describe the configuration space algorithm. First we present some terminology in Section 2.1. As mentioned in the Chapter 1, the CLP requires the input parameters and the set of functions of an algorithm. In return, the CLP gives back a set of signs from those functions so that the algorithm can make decisions based on those signs. To satisfy the CLP requirements, we list the basic functions that are used through out the C-Space algorithm in Section 2.2, and we describe the input parameters in Section 2.3. We introduce the rest of the functions through out the chapter since they are specific to certain parts of the C-Space algorithm.

We explicitly compute the C-Space by computing the evolving convolution, and we describe this process in Sections 2.4 through 2.6. We discuss the necessary floating calculations that are used through out the C-Space algorithm in Section 2.7. We explain the one caveat of the CLP algorithm and how it affects the C-Space algorithm in Section 2.8. The algorithm pseudocode and complexity is given in Section 2.9.

2.1 Terminology

In order to explain the configuration space more clearly, we introduce some terminology. The *configuration* of a region is a triple $\langle x, y, \theta \rangle$, where $\langle x, y \rangle$ represent its translation in the $2D$ space, and angle θ represents its rotation (orientation), where $\theta \in [0, 2\pi)$. In robotics, this triplet is often referred to as the *pose* of the robot on a plane [9]. Let A and B be regions. The *configuration space* (or C-Space) of A is the set of all possible configurations (i.e. all possible triplets). Given an obstacle B with a fixed pose, the *free space* of A, with respect to B, is the set of configurations such that A does not overlap B. Its complement is the *blocked space* of A, and its boundary is the set of contacting configurations, see Figure 2.1.

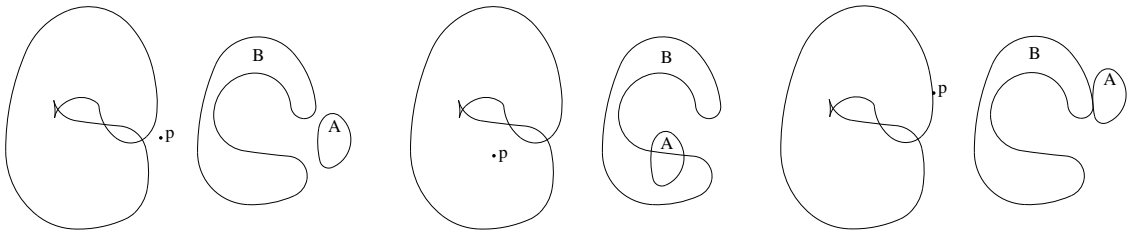


Figure 2.1: Configuration space of part A with respect to part B. The left figures in each case represent the free/blocked space at some θ, and point p represents the pose of A. The right figure in each case shows the actual location of A and B. (a) A residing in free space. (b) A residing in blocked space. (c) A resting on boundary between free/blocked space.

Since A has three *degrees of freedom* (DOF), the configuration space is a three-dimensional ($3D$) manifold:

$$\mathbb{R}^2 \times [0, 2\pi).$$

A degree of freedom (DOF) is either a translational displacement or rotation that specify the position and orientation of the body.

The representation of A and B with line segments and circular arcs is sufficiently rich for many purposes. In this dissertation, we focus on representing A and B with only circular arcs. Line segments are harder to handle than circular arcs because they require a more difficult set of changes to the evolution than the circular arcs. The changes include two parallel line segments *bumping*, or a *triangle flip* of line segments. We explain bumps and triangle flips of circular arcs in Section 2.6. The hope is to construct the C-Space algorithm first for circular arcs, and in future work generalize to line segments.

Given two regions A and a fixed pose B, we want to generate C-Space by computing the boundary of the free and blocked space. We allow for region A to be able to translate by a $2D$ *vector* $t = \langle t_x, t_y \rangle$: $A' = A + t = \{a + t \,|\, a \in A\}$. That is, a region can translate from pose $\langle x, y, \theta \rangle$ to $\langle x', y', \theta \rangle$, where $x' = x + t_x$ and $y' = y + t_y$.

We generate the C-Space of two regions A and a fixed pose B by computing the *Minkowski sum* as θ varies. The Minkowski sum of two sets M_1 and M_2 in Euclidean space is the result of adding every element of M_1 to M_2, i.e., the set:

$$M_1 \oplus M_2 = \{m_1 + m_2 \,|\, m_1 \in M_1, m_2 \in M_2\}. \tag{2.1}$$

Using the translation vector t, we describe the blocked space of A and fixed pose B as follows. To reside in the blocked space, the intersection $(A+t) \cap B$ is not empty:

$$
\begin{array}{ll}
(A+t)\cap B\neq\emptyset & \\
\Updownarrow & \\
a+t=b, & \text{for } a\in A, b\in B \\
\Updownarrow & \\
t=b-a, & \text{for } a\in A, b\in B \\
\Updownarrow & \\
t=b+(-a), & \text{for } a\in A, b\in B \\
\Updownarrow & \\
t\in B\oplus -A, & \text{by Equation 2.1 with } -A=\{-a\,|\,a\in A\}.
\end{array}
$$

We have shown that we can describe the blocked space as a non-empty intersection of $(A+t)$ and a fixed B or t belonging to $(B\oplus -A)$:

$$(A+t)\cap B\neq\emptyset \;\Leftrightarrow\; t\in B\oplus -A. \tag{2.2}$$

We can describe the free space of A and a fixed B by the contrapositive of Equation 2.2:

$$(A+t)\cap B=\emptyset \;\Leftrightarrow\; t\in \overline{B\oplus -A}. \tag{2.3}$$

So the boundary of the free/blocked space is the boundary of the Minkowski sum of the two regions. We have essentially converted the problem of comparing when two regions overlap each other to the problem of comparing when a point lies inside a Minkowski sum region. That is, the point inside the Minkowski sum region is the blocked space, and the point outside the region is the free space.

In order to generate the C-Space of A and fixed pose B we keep track of the Minkowski sum of $B\oplus\theta(-A)$ for all $\theta\in[0,2\pi)$. We denote $\theta(p)$ to be the rotation of the input p by an angle θ.

To construct the Minkowski sum we follow Milenkovic and Sacks' generalization of the convolution-based Minkowski sum algorithm by Guibas et al. [20] to circular arcs [35]. Given two regions $F = \langle V_f, E_f \rangle$ and $M = \langle V_m, E_m \rangle$, the algorithm is as follows. The algorithm uses *smoothing* arcs so that edges of the region meet at continuous angles. That is, the algorithm adds zero-radius arcs to incident edges that have unequal outward normal angles. As a result, the angles are then continuous, because the zero-radius arcs have equal outward normal angles to the incident edges. Unfortunately, smoothing arcs raise CLP issues that exceed the scope of the dissertation.

The main difficulty with smoothing arcs (or zero-radius arcs) is that each circular arc has a particular concavity, but the CLP can't determine the concavity of an arc with zero radius. So we either treat zero-radius arcs as special "zero" cases, or we modify the CLP to give the correct concavity to the arcs. Special "zero" cases lead to difficulties in the algorithm, and modification of the CLP is an issue because we treat the CLP as a "black box" library.

Because of the difficulties of smoothing arcs, the input regions F and M are restricted to not only being circular arcs, but each arc in the region must meet smoothly. Also, for simplicity of the algorithm, we limit each circular arc with radius r to be less than $r\pi$ in arc length (less than half a circle's arc length). Next we determine all pairs of edges $\langle e_1, e_2 \rangle$ where $e_1 \in E_f, e_2 \in E_m$ with overlapping angle intervals. Arcs with overlapping angle intervals are summed together over those intervals, and these sum pairs of arcs generate a convolution $M \otimes F$.

Finally we calculate the *arrangement* of $M \otimes F$, assigning *winding numbers* to its *cells*, and select *subedges* of the arrangement that separate cells with zero or non-zero winding numbers. These subedges bound $M \oplus F$. If we allow $F = B$ and $M = -A$, generating these subedges of the arrangement creates the boundary between free/blocked space of A and B for some fixed θ. In Section 2.4 we explain how we sum edges of fixed set $F = B$ and moving set $M = -A$, and how we update the summed edges as we vary θ. We save the discussion of arrangements, subedges, cells, and winding numbers for Chapter 3.

2.2 Configuration space functions

In this section, we list the basic set of functions that are needed for the entire C-Space algorithm. For each function, a parameter of it is called a *coordinate*. As discussed in Chapter 1, a *primitive* is a function that returns a sign ($f(x) > 0$), such as the *difference* primitive and its *gradient* defined by

$$f_{di}(x) = f_{di}(x_1, x_2) = x_1 - x_2, \tag{2.4}$$

$$\nabla f_{di} = \left(\frac{\partial f_{di}}{\partial x_1}, \frac{\partial f_{di}}{\partial x_2}\right) = \Big(1, -1\Big). \tag{2.5}$$

If $x_1 - x_2 > 0$, then $x_1 > x_2$, else $x_1 < x_2$. In the difference primitive, x_1 and x_2 are example of coordinates. Another way to find the sign of the difference primitive is to evaluate the primitive explicitly as a function with $x_3 = x_1 - x_2$, and then find the

sign using the *identity* primitive and its gradient defined by

$$f_{id}(x) = f_{id}(x_3) = x_3, \tag{2.6}$$

$$\nabla f_{id} = \left(\frac{\partial f_{id}}{\partial x_3}\right) = \Big(1\Big). \tag{2.7}$$

If coordinate $x_3 > 0$ then $x_1 > x_2$, else $x_3 < 0$ and $x_1 < x_2$.

Here are a list of other basic primitives that are required for C-Space computation:

- The *Positive-sum* primitive and its gradient are

$$f_{ps}(x) = f_{ps}(r_1, r_2) = r_1 + r_2, \tag{2.8}$$

$$\nabla f_{ps} = \left(\frac{\partial f_{ps}}{\partial r_1}, \frac{\partial f_{ps}}{\partial r_2}\right) = \Big(1, 1\Big). \tag{2.9}$$

  This primitive is necessary to test for concavity of circular arcs when we sum their radii r_1 and r_2. The summed arc is convex if $r_1 + r_2 > 0$. Otherwise it is concave.

- The *Positive-angle* primitive and its gradient are

$$f_{pa}(x) = f_{pa}(\theta) = \sin(\theta), \tag{2.10}$$

$$\nabla f_{pa} = \left(\frac{\partial f_{pa}}{\partial \theta}\right) = \Big(\cos\theta\Big). \tag{2.11}$$

  If $\sin(\theta) > 0$ then $0 < \theta < \pi$. Otherwise $\pi < \theta < 2\pi$.

- The *Angle-difference* primitive and its gradient are

$$f_{ad}(x) = f_{ad}(\theta_1, \theta_2) = \sin(\theta_1 - \theta_2), \tag{2.12}$$

$$\nabla f_{ad} = \left(\frac{\partial f_{ad}}{\partial \theta_1}, \frac{\partial f_{ad}}{\partial \theta_2}\right) = \Big(\cos(\theta_1 - \theta_2), -\cos(\theta_1 - \theta_2)\Big). \tag{2.13}$$

If $\sin(\theta_1 - \theta_2) > 0$ then $\theta_1 > \theta_2$ trigonometrically, else $\theta_1 < \theta_2$.

The positive-angle primitive and the angle-difference primitive shows that functions can also be trigonometric, and we need trigonometric functions to deal with angle θ. For example, the combination of the positive-angle primitive with the angle-difference primitive provides the ability to order angles from 0 to 2π. We break the set of all angles into two sets: $(0, \pi)$ and $(\pi, 2\pi)$ using the positive-angle primitive. Then we order within the two sets separately with the angle-difference primitive.

As we showed in the difference function, we can treat the function as a primitive by evaluating its sign or we can evaluate the function explicitly and return a value. Evaluating the function explicitly is necessary for ordinary geometric computations and for reducing the number of overall primitives. Here are some of the basic explicit functions:

- The *Negation* function returns the negation of a coordinate. The function and its gradient are

$$f_{ne}(x) = f_{ne}(x_1) = -x_1, \tag{2.14}$$

$$\nabla f_{ne} = \left(\frac{\partial f_{ne}}{\partial x_1}\right) = \Big(-1\Big). \tag{2.15}$$

- The *Opposite-angle* function returns the angle π (or 180°) away from an angle coordinate. The function and its gradient are

$$f_{oa}(x) = f_{oa}(\theta_1) = \theta_1 + \pi, \tag{2.16}$$

$$\nabla f_{oa} = \left(\frac{\partial f_{oa}}{\partial \theta_1}\right) = \Big(1\Big). \tag{2.17}$$

- *Rotate-vertex* function returns a vertex rotated by angle θ about the origin. Rotation is achieved by applying the following rotation matrix on the vector form of a vertex (x_1, y_1):

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}. \tag{2.18}$$

From the 2×2 rotation matrix it is apparent that the rotate-vertex function is actually comprised of two functions. Here are the functions and their respective gradients:

$$f_{rx}(x) = \quad f_{rx}(x_1, y_1, \theta) \quad = \; x_1 \cos\theta - y_1 \sin\theta, \tag{2.19}$$

$$f_{ry}(x) = \quad f_{ry}(x_1, y_1, \theta) \quad = \; x_1 \sin\theta + y_1 \cos\theta, \tag{2.20}$$

$$\nabla f_{rx} = \left(\tfrac{\partial f_{rx}}{\partial x_1}, \tfrac{\partial f_{rx}}{\partial y_1}, \tfrac{\partial f_{rx}}{\partial \theta}\right) = \Big(\cos\theta, -\sin\theta, -x_1 \sin\theta - y_1 \cos\theta\Big), \tag{2.21}$$

$$\nabla f_{ry} = \left(\tfrac{\partial f_{ry}}{\partial x_1}, \tfrac{\partial f_{ry}}{\partial y_1}, \tfrac{\partial f_{ry}}{\partial \theta}\right) = \Big(\sin\theta, \quad \cos\theta, \quad x_1 \cos\theta - y_1 \sin\theta\Big). \tag{2.22}$$

There are three types of *coordinates* in the functions: input, explicit, implicit. For n implicit coordinates, there must be n *implicit constraints* to bound those coordinates. As explained in Chapter 1, an implicit constraint is a function constrained to equal zero: $g(x) = 0$.

2.3 Configuration space input

A region is bounded by a circular list of arcs each with endpoint vertex a, normal angle α at a, center c, and radius r, see Figure 2.2. Because of the smoothness restriction of the circular arcs, from Section 2.1, the other endpoint b equals the a of the next edge, and the normal angle β at b equals the normal angle α at a of the next edge.

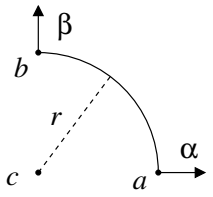


Figure 2.2: Circular arc edge with oriented end points.

For a convex edge, $\alpha < \beta$ (angle-difference primitive) and $r > 0$ (identity primitive). For a concave edge, $\alpha > \beta$ and $r < 0$. Each endpoint a is represented by two input coordinates $\langle a_x, a_y \rangle$. The remaining, α, c, r are implicit coordinates. The implicit constraints are $c+u(\alpha)r-a = 0$ and $c+u(\beta)r-b = 0$, where $u(\alpha) = (\cos\alpha, \sin\alpha)$ is the unit vector in the direction α. Since these are vector equations, we have four implicit constraints on four implicit coordinates.

The general forms of the constraints are

$$c_x + r\cos\theta - p_x = 0, \tag{2.23}$$

$$c_y + r\sin\theta - p_y = 0. \tag{2.24}$$

Their respective gradients are

$$\nabla g_{cx} = (\tfrac{\partial g_{cx}}{\partial c_x}, \tfrac{\partial g_{cx}}{\partial r}, \tfrac{\partial g_{cx}}{\partial \theta}, \tfrac{\partial g_{cx}}{\partial p_x}) = (1, \cos\theta, -r\sin\theta, -1) \quad (2.25)$$

$$\nabla g_{cy} = (\tfrac{\partial g_{cy}}{\partial c_y}, \tfrac{\partial g_{cy}}{\partial r}, \tfrac{\partial g_{cy}}{\partial \theta}, \tfrac{\partial g_{cy}}{\partial p_y}) = (1, \sin\theta, \quad r\cos\theta, -1). \quad (2.26)$$

2.4 SEEs and SVEs

There are two input regions, one fixed $F = \langle V_f, E_f \rangle$ and one moving $M = \langle V_m, E_m \rangle$. Following the Minkowski sum algorithm we described in Section 2.1, we sum edges with overlapping angle intervals at those intervals. This creates the convolution at a fixed θ, and we compute the configuration space by computing the arrangement of the convolution. We account for an evolving θ as follows.

A moving edge $\overline{a_m b_m}$ starts overlapping a fixed edge $\overline{a_f b_f}$ (in orientation) when $\beta_m + \theta = \alpha_f$ or $\theta = \alpha_f - \beta_m$, assuming both are convex. It finishes overlapping when $\theta = \beta_f - \alpha_m$. Let $\overline{ab}$ be that summed edge-edge (SEE) with angles α and β, center $c_f + \theta(c_m)$, and radius $r_f + r_m$, see Figure 2.3.

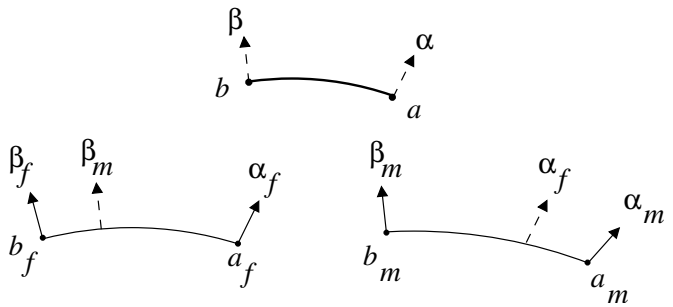


Figure 2.3: Moving edge overlapping a fixed edge.

For each SEE, there are four different ways of summing a vertex with an edge. We call the sum of a vertex with an edge a summed vertex-edge (SVE). We calculate

the four SVEs of SEE $\overline{ab}$ as follows. For $\theta < \alpha_f - \alpha_m$, $\alpha = \alpha_f$ and

$$a = a_1 = a_f + \theta(c_m) + r_m u(\alpha_f), \tag{2.27}$$

where $\theta(c_m)$ means c_m rotated by θ about the origin (rotate-vertex function). For $\theta > \alpha_f - \alpha_m$, $\alpha = \alpha_m + \theta$ and

$$a = a_2 = c_f + r_f\theta(u(\alpha_m)) + \theta(a_m). \tag{2.28}$$

For $\theta < \beta_f - \beta_m$, $\beta = \beta_m + \theta$ and

$$b = b_1 = c_f + r_f\theta(u(\beta_m)) + \theta(b_m). \tag{2.29}$$

For $\theta > \beta_f - \beta_m$, $\beta = \beta_f$ and

$$b = b_2 = b_f + \theta(c_m) + r_m u(\beta_f). \tag{2.30}$$

The four SVEs are a_1, a_2, b_1, and b_2. SVEs a_1 and b_2 are each created from the sum of a fixed vertex (a_f or b_f) and a moving edge ($\overline{a_m b_m}$). SVEs a_2 and b_1 are each created from the sum of a moving vertex (a_m or b_m) and a fixed edge ($\overline{a_f b_f}$).

The evolution of SEEs and SVEs is tracked as θ ranges from 0 to 2π. As we vary θ, SVEs are either valid and exist in the current convolution, or they are not valid and do not exist in the current convolution. We call SVEs that are valid at the current θ *live*, and we call non-valid ones *dead*. When an SEE has one live a and one live b, then the SEE is live. Also every SVE is incident to two SEEs, so when an SVE is live, both incident SEEs are also live. The following uses the angle-difference primitives to describe when the four SVEs of an SEE are live.

- SVE a_1 is live when $\alpha_f - \beta_m < \theta < \alpha_f - \alpha_m$;
- SVE a_2 is live when $\alpha_f - \alpha_m < \theta < \beta_f - \alpha_m$;
- SVE b_1 is live when $\alpha_f - \beta_m < \theta < \beta_f - \beta_m$;
- SVE b_2 is live when $\beta_f - \beta_m < \theta < \beta_f - \alpha_m$.

The *direction* of an SEE is important for computing the winding numbers for the kinetic framework [20, 35]. The direction determines how we traverse that SEE in the convolution, and it depends on the concavity of edges $\overline{a_m b_m}$ and $\overline{a_f b_f}$. If both edges are convex or concave, the direction is forward and we traverse the SEE from a to b. Otherwise if one edge is convex and the other is concave, the direction is flipped and we traverse the SEE from b to a.

2.5 Initial crossing lists and edge lists

We call the intersection of two SEEs a *crossing*, and we call the crossings of the two SEEs a *crossing list*. Because each SEE is represented by a circular arc, each pair of SEEs has zero, one, or two crossings in their crossing list. To determine the initial arrangement, we compute the initial crossing lists for each pair of SEEs at $\theta = 0$. Once we identify initial crossings in the convolution, we can then create the initial *edge lists*. The edge list of an SEE is the list of its crossings.

We construct the initial crossing lists as follows. Let $\overline{a_1 b_1}$ and $\overline{a_2 b_2}$ be two SEEs. To test if they intersect, we first test if their corresponding circles intersect, where

$\overline{a_1 b_1}$ is on circle $\langle c_1, r_1 \rangle$ and $\overline{a_2 b_2}$ is on circle $\langle c_2, r_2 \rangle$, where c_i is the center and r_i is the radius of the respective circles, see Figure 2.4. If the circles do intersect and a crossing lies between the endpoints of the SEE, then the SEEs also have a crossing.

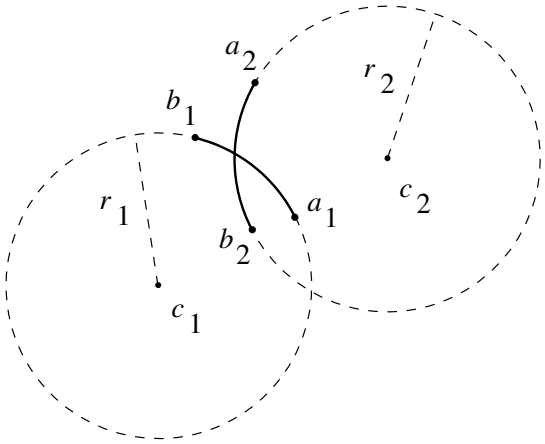


Figure 2.4: Two circles, representing two SEEs, crossing each other.

We first create explicit coordinates $w = (w_x, w_y) = c_2 - c_1$. For ease of computation, we create implicit coordinates $\phi(w)$ and $r(w)$, the polar coordinate representation of w (ϕ and r for short). The implicit constraints are

$$w_x - r\cos\phi = 0, \tag{2.31}$$

$$w_y - r\sin\phi = 0, \tag{2.32}$$

and their respective gradients are

$$\nabla g_{px} = \left(\frac{\partial g_{px}}{\partial w_x}, \frac{\partial g_{px}}{\partial r}, \frac{\partial g_{px}}{\partial \phi}\right) = \Big(1, -\cos\phi, \quad r\sin\phi\Big), \tag{2.33}$$

$$\nabla g_{py} = \left(\frac{\partial g_{py}}{\partial w_y}, \frac{\partial g_{py}}{\partial r}, \frac{\partial g_{py}}{\partial \phi}\right) = \Big(1, -\sin\phi, -r\cos\phi\Big). \tag{2.34}$$

This gives two implicit constraints for two implicit coordinates.

The circles $\langle c_1, r_1 \rangle$ and $\langle c_2, r_2 \rangle$ have a pair of crossings if $r_1 + r_2 > r$ and $r_1 - r_2 < r$ and $r_2 - r_1 < r$. The first condition makes sure the circles are not too far apart for intersection. The last two conditions make sure the circles are not too close for intersection (one circle encompassing the other). To limit the number of primitives, we can create explicit coordinates $r_s = r_1 + r_2$, $r_d^+ = r_1 - r_2$, and $r_d^- = -r_d^+$. We can then use the difference primitive and compare r_s, r_d^+, and r_d^- to r. The signs returned from these difference primitives correspond to a passing or failing of a condition.

If any of the conditions are not met, then there are no crossings. Otherwise all three conditions are met and there are two crossings. We describe why it is not possible to have only one crossing in Chapter 5.

To compute the two crossings of two circles we follow the mathematics involved in the circle-circle intersection algorithm, which we discuss in Section 2.7.1. For each crossing p, we generate its implicit coordinates $\phi(c_1 p)$ and $\phi(c_2 p)$. The angle $\phi(c_1 p)$ is the outward normal direction on the circle $\langle c_1, r_1 \rangle$ at p; similarly for $\phi(c_2 p)$. The implicit constraints are

$$-p + r_i u(\phi(c_i p)) + c_i = 0 \quad \text{for } i = 1, 2. \tag{2.35}$$

Since these are vector equations, this gives four implicit constraints for four implicit coordinates: $p_x, p_y, \phi(c_1 p), \phi(c_2 p)$. The general form for these constraints were given in Section 2.3.

By comparing $\phi(c_1 p)$ and $\phi(c_2 p)$ to the endpoint orientations (α and β) of the SEE, we can determine if p lies on each SEE. Using the angle-difference primitive: if

$\sin(\phi(c_i p) - \alpha) > 0$ and $\sin(\beta - \phi(c_i p)) > 0$, then $\phi(c_i p)$ lies on the SEE. If p lies on each SEE, that means the SEEs cross each other at p. If p does not lie on both SEEs, that means the circles representing the SEEs cross at p, but the SEEs do not cross at p.

We construct the initial edge lists as follows. Given that each crossing's outward normal angle ϕ is between angle α and β of the SEE, we create the initial edge list by inserting each crossing using standard binary insertion on each crossing's ϕ. The binary insertion uses the angle-difference primitive as the comparison function. The set of SEEs, SVEs, and edge lists are enough information to generate the initial arrangement, which we discuss in Chapter 3.

Given n fixed edges and m moving edges, there are nm SEEs and n^2m^2 pair of SEEs to check for potential initial crossing lists. Even though a standard Bentley-Ottmann line sweep [5] is more efficient, the running time of the overall C-Space computation is not affected. We discuss the complexity of the C-Space algorithm in Section 2.9.

2.6 Configuration space events

In Section 2.5 we created the initial convolution; however, as angle θ varies, the convolution evolves. We call each type of change in the evolving convolution an *event*. There are 3 main type of events: Type 1, Type 2, and Type 3.

- Type 1 events are when SEEs and SVEs appear or disappear from the convolution, and we explain them in Section 2.6.1.

- Type 2 events involve a pair of SEEs. There are three Type 2 events: bump, stab, and special events. We illustrate them in Section 2.6.2 through 2.6.4. We keep track of how the Type 2 events affect each pair of SEEs in Section 2.6.5.

- Type 3 triangle flip events are when 3 SEEs meet at a single point, and we describe them in Section 2.6.7.

As the evolution evolves, Type 2 and Type 3 events *add*, *remove*, or *swap* crossings from the edge lists of the SEEs. We discuss how we update the edge lists at these type of events in Section 2.6.6.

2.6.1 Type 1 events

Given the two input regions, fixed $F = \langle V_f, E_f \rangle$ and moving $M = \langle V_m, E_m \rangle$, we create an explicit coordinate for each Type 1 event:

$$\theta_{ij} = \alpha_{f_i} - \alpha_{m_j}. \tag{2.36}$$

The angle θ_{ij} is when vertex a_{m_j} "passes" vertex a_{f_i}, where $a_{m_j} \in V_m$ and $a_{f_i} \in V_f$.

By keeping track of Type 1 events, we know when SVEs are newly live or newly dead, and consequently we know when SEEs come live or go dead. By keeping track of the live SVEs and SEEs, we can keep track of the current convolution at θ.

2.6.2 Type 2 bump events

As angle θ varies, a *bump* event is when two live SEEs come into contact at two crossing points. See Figure 2.5 for an example of a bump event. An *unbump* event is when two live SEEs come out of contact of two crossing points, which is a reverse of the figure.

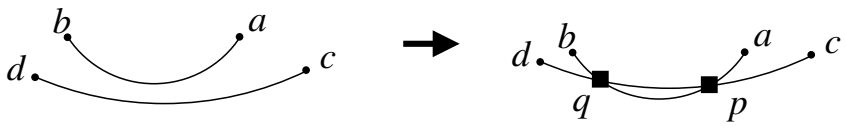


Figure 2.5: Type 2 inner bump

We do the following to detect bump/unbump events. An SEE has center $c_f+\theta(c_m)$ and radius $r_f + r_m$. Two SEEs $c_{f_1} + \theta(c_{m_1}), r_1$ and $c_{f_2} + \theta(c_{m_2}), r_2$ potentially bump or unbump each other when

$$|(c_{f_2} + \theta(c_{m_2})) - (c_{f_1} + \theta(c_{m_1}))| = r_1 + r_2, \tag{2.37}$$

and when

$$|(c_{f_2} + \theta(c_{m_2})) - (c_{f_1} + \theta(c_{m_1}))| = |r_1 - r_2|. \tag{2.38}$$

If Equation (2.37) is true at θ, then the bump is an *outer* bump, see Figure 2.6. If Equation (2.38) is true at θ, then it is an *inner* bump, see Figure 2.5. "Outer" means the moving circles, representing the SEEs, do not encompass one another, while "inner" means one circle encompasses the other.

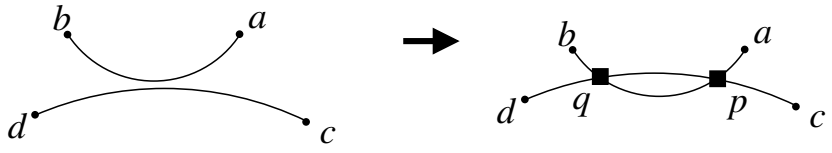


Figure 2.6: Type 2 outer bump

We let $|c_f + \theta(c_m)| = r$ for $c_f = c_{f_2} - c_{f_1}$, $c_m = c_{m_2} - c_{m_1}$, and $r = r_1 + r_2$ or $r = r_1 - r_2$. We calculate the polar coordinates: $r(c_f)$, $\phi(c_f)$, $r(c_m)$ and $\phi(c_m)$. We need to determine the points of a circle centered at c_f with radius $r(c_m)$ at which it intersects the circle of radius r centered at the origin.

There is an intersection (in a pair of crossings) if $r(c_f) - (r + r(c_m)) < 0$ and $r(c_f) - (r - r(c_m)) > 0$ and $r(c_f) - (r(c_m) - r) > 0$. This can be rewritten to $r(c_f)+r(c_m) > r$ and $r(c_f)-r(c_m) < r$ and $r(c_m)-r(c_f) < r$, to match the established conditions for intersection from Section 2.5. If there are a pair of crossings we use the circle-circle intersection algorithm from Section 2.7.1 to find them.

Let p be a crossing. The constraints on p, $\phi(p)$, and $\phi(c_f p)$ are

$$-p + ru(\phi(p)) = 0 \tag{2.39}$$

$$-p + r(c_m)u(\phi(c_f p)) + c_f = 0. \tag{2.40}$$

With some rewriting the general form for the constraints in Equation (2.39) matches the polar coordinate constraints given in Section 2.5. Similarly, with some rewriting

the general form for the constraints in Equation (2.40) matches the constraints given in Section 2.3.

Angle $\phi(p)$ is the outward normal direction on the circle centered at the origin at p, and similarly angle $\phi(c_f p)$ is the outward normal angle on circle $\langle c_f, r(c_m)\rangle$ at p. Since these are vector constraints, there are four implicit constraints on four implicit coordinates: two from p and one from each ϕ.

The angular position of the bump point is $\phi(p)$ or $\phi(p)^-$, depending on which segment, where $\phi(p)^- = \phi(p) + \pi$ (opposite-angle primitive). If both bump SEEs are convex, it is at $\phi(p)$ on the first segment and $\phi(p)^-$ on the second. The bump (or unbump) occurs at

$$\theta = \phi(c_f p) - \phi(c_m). \tag{2.41}$$

This θ can be plugged into the formulas for the SVE orientations. The SVE orientations, α and β, are then compared to $\phi(p)$ or $\phi(p)^-$ to make sure the bump occurs on both segments at θ, see Figure 2.7.

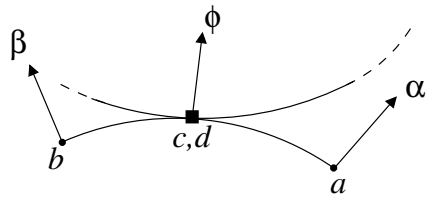


Figure 2.7: Bump event: Another SEE bumps SEE $\overline{ab}$ with crossings c and d with outward normal angle ϕ. Because ϕ is between α and β (using the angle-difference primitive), the bump is valid.

We have shown how to detect bump events at θ; however, we do not know if the two SEEs are meeting for the first time (bump) or meeting for the last time (unbump). We determine if the event is a bump or an unbump in Section 2.6.5.

2.6.3 Type 2 stab events

As angle θ varies, a *stab* event is when a live SVE crosses a live SEE in the convolution, and a new crossing is introduced to both the incident SEE of the SVE and the "stabbed" SEE. An *unstab* event is when a live SVE crosses a live SEE, but an old crossing is removed from both the incident SEE of the SVE and the "unstabbed" SEE. We do the following to detect stab/unstab events. An SEE, represented by circle $\langle c_f + \theta(c_m), r\rangle$, is potentially stabbed or unstabbed by an SVE $A_2 + \theta(B_2)$ when

$$|A + \theta(B)| = r, \tag{2.42}$$

where $A = A_2 - c_f$ and $B = B_2 - c_m$. We create polar coordinates $r(A)$, $\phi(A)$, $r(B)$, and $\phi(B)$. This is analogous to solving $|c_f + \theta(c_m)| = r$ in the Section 2.6.2, and again we use the circle-circle intersection algorithm from Section 2.7.1 to find the pair of crossings (if they exist). The following is how A_2 and B_2 are broken down in the four different SVEs of an SEE from Section 2.4:

- a_1: $A_2 = a_f + r_m u(\alpha_f)$, $B_2 = c_m$;
- a_2: $A_2 = c_f$, $B_2 = r_f u(\alpha_m) + a_m$;
- b_1: $A_2 = c_f$, $B_2 = r_f u(\beta_m) + b_m$;

- b_2: $A_2 = b_f + r_m u(\beta_f)$, $B_2 = c_m$.

Let p be a crossing. The circle is stabbed at p with orientation $\phi(p)$. The stab (or unstab) occurs when

$$\theta = \phi(Ap) - \phi(B). \tag{2.43}$$

Similar to Section 2.6.2, angle $\phi(Ap)$ is the outward normal direction on the circle $\langle A, r(B)\rangle$ at p. Analogous to what was done in Section 2.6.2, we plug θ into the formula for the SVE orientations of the SEE. The SVE orientations, α and β, are compared to $\phi(p)$ to make sure the stab occurs on the SEE at θ, see Figure 2.8. We also make sure the SVE is live at this stab θ. To test if an SVE is live, we use the angle-difference primitives established in Section 2.4.

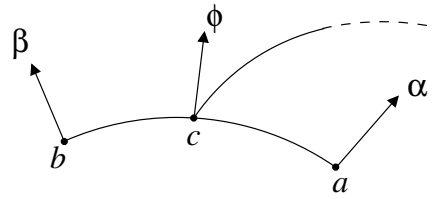


Figure 2.8: Stab event: SVE c stabbing SEE $\overline{ab}$. The point of stab has outward normal angle ϕ, and because ϕ is between α and β (using the angle-difference primitive), the stab is valid.

2.6.4 Type 2 special events

As angle θ varies, *special* events occur when crossings are introduced because of Type 1 events. Recall from Section 2.6.1 that Type 1 events add or remove SEEs and SVEs from the evolving convolution. If the addition of newly live SEEs to the convolution introduces a new crossing, this is a *special stab* event, see Figure 2.9. If the removal

of newly dead SEEs from the convolution removes a crossing, this is a *special unstab* event.

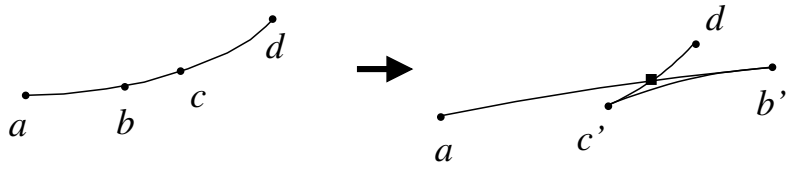


Figure 2.9: Type 2 special stab: Before special stab, SVE b and c and SEE $\overline{bc}$ are about to go dead. At the special stab, b and c are newly dead and are replaced by newly live b' and c', and $\overline{ab'}$ and $\overline{c'd}$ has a new crossing immediately at c' and b'.

For Type 2 stab and bump events, we allow only for crossings to occur between the SVEs (with the angle-difference primitive), and not on them. For Type 2 special events, we have to allow for crossings to occur right at the SVEs.

To determine when special events occur, we use the following primitive checks. The checks determine if the special event is a special stab or a special unstab event, and also it determines the *crossing sign*. Because we are dealing with only circular arcs, there are at most two crossings and each crossing has a unique crossing sign, see Figure 2.10. As the convolution evolves, one of the two crossings is always positive, while the other crossing is always negative. The unique crossing sign is determined by the cross-product of the two outward normal angles at the crossing of the two SEEs: $u(\phi_1) \times u(\phi_2)$. To determine the cross-product sign without actually evaluating the cross-product, we use the angle-difference primitive. If $\sin(\phi_1 - \phi_2) < 0$, then $u(\phi_1) \times u(\phi_2) > 0$. Otherwise $\sin(\theta_1 - \theta_2) > 0$ and $u(\phi_1) \times u(\phi_2) < 0$.

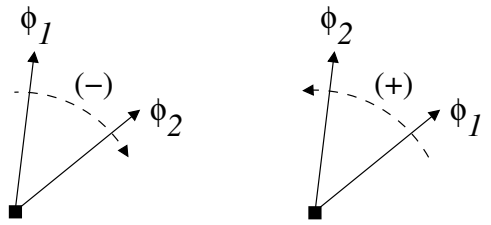


Figure 2.10: Unique crossing sign for each crossing determined by the cross-product.

We follow Algorithm 2.1 to determine when Type 2 special events occur. This looks into every possible case of summation of edges based on their concavity and radii. Let s_1 and s_2 be segments of the fixed region in counter-clockwise order. Let a_{12} be the incident vertex on both segments with an outward normal direction of $\pi/2$: s_2 is to the left of s_1. Similarly let s_3 and s_4 be segments of the moving region with a_{34} as the incident vertex on both with the same outward normal direction. For Algorithm 2.1 let r_i be the radius of segment s_i, and suppose the moving body has rotated counter-clockwise past the Type 1 event ($\sin(\alpha_{34} - \alpha_{12}) > 0$).

Algorithm 2.1 does not cover all the cases; however, the following rules from Algorithm 2.2 show how to transform the input (r_1 r_2 r_3 r_4) by applying a permutation. Let $r_i \Leftrightarrow r_j$ mean we are changing the permutation by swapping r_i with r_j. We apply the swap rules from Algorithm 2.2 before we call the Algorithm 2.1.

Given the input (r_1 r_2 r_3 r_4), we apply the rules in order, and keep track of the current permutation. We then send the permutation as input to Algorithm 2.1. Let $\overline{s_i s_j}$ be an SEE with fixed edge s_i and moving edge s_j. If $\overline{s_1 s_4}$ intersects $\overline{s_2 s_3}$, we apply the inverse of the combined permutation from Algorithm 2.2 to determine which two original SEEs intersected.

Algorithm 2.1 Detecting Type 2 special events

```
1: procedure DETECT-TYPE2S-SINGULAR(r_1, r_2, r_3, r_4)
2:     if (r_1 < 0 and r_2 < 0 and r_3 > 0 and r_4 > 0) then
3:         if (r_1 + r_4 < 0 and r_1 + r_3 > 0 and r_2 + r_3 < 0) then
4:             ▷ s_1s_4 intersects s_2s_3, crossing sign: (−)
5:         else if (r_1 + r_4 > 0 and r_1 + r_3 < 0 and r_2 + r_3 > 0) then
6:             ▷ s_1s_4 intersects s_2s_3, crossing sign: (−)
7:         end if
8:     else if (r_1 < 0 and r_2 > 0 and r_3 > 0 and r_4 > 0) then
9:         if (r_1 + r_3 > 0 and r_1 + r_4 < 0) then
10:            ▷ s_1s_4 intersects s_2s_3, crossing sign: (−)
11:        end if
12:    else if (r_1 < 0 and r_2 > 0 and r_3 < 0 and r_4 > 0) then
13:        if (r_1 + r_4 > 0 and r_2 + r_3 > 0 and r_1 + r_4 > r_2 + r_3) then
14:            ▷ s_1s_4 intersects s_2s_3, crossing sign: (+)
15:        else if (r_1 + r_4 < 0 and r_2 + r_3 < 0 and r_1 + r_4 > r_2 + r_3) then
16:            ▷ s_1s_4 intersects s_2s_3, crossing sign: (−)
17:        end if
18:    end if
19: end procedure
```

If the crossing exists, the algorithm also gives a crossing sign for the crossing. If the crossing sign returned from the algorithm is positive (+), then the cross-product $\alpha_{12} \times \alpha_{34}$ is positive. For each of the four rules in Algorithm 2.2 that were applied, we flip the sign once. This final sign is the crossing sign of the crossing from the Type 2 special event.

To determine if a special event is a special stab or a special unstab, we check if rule 1 from Algorithm 2.2 was invoked ($\sin(\alpha_{34} - \alpha_{12}) < 0$). If rule 1 was invoked, then the special event is a special unstab event, and it happens immediately before the Type 1 event. Otherwise, this event is a special stab and it happens immediately after the Type 1 event. Thus each special event is tightly coupled with a Type 1

Algorithm 2.2 Rules for Type 2 special events

1. If prior to event ($\sin(\alpha_{34} - \alpha_{12}) < 0$), then swap:
   $r_1 \Leftrightarrow r_3$,
   $r_2 \Leftrightarrow r_4$.
2. If $r_3 r_4 < 0$ (i.e., r_3 and r_4 don't have the same sign) and $r_1 r_2 > 0$, then swap:
   $r_1 \Leftrightarrow r_4$,
   $r_2 \Leftrightarrow r_3$.
3. If $r_4 < 0$ and $r_3 < 0$, then swap:
   $r_1 \Leftrightarrow r_2$,
   $r_3 \Leftrightarrow r_4$.
   Also, negate all r_i.
4. If $r_4 < 0$ and $r_1 > 0$, then swap:
   $r_1 \Leftrightarrow r_4$,
   $r_2 \Leftrightarrow r_3$.

event, and we treat that pair of event as one event in order for a correct explicit construction of the evolving convolution.

2.6.5 Evolution of crossing lists due to Type 2 events

For each pair of SEEs we have determined the following. From Section 2.6.2, we found all possible bump events (bump/unbump) between a pair of SEEs but we are unsure if each event is a bump or unbump. Similarly from Section 2.6.3, we found all possible stab events (stab/unstab) between a pair of SEEs but are unsure if each event is a stab or unstab. In order to determine which event is which, we need to keep track of the evolution of the crossing list for each pair of SEEs using each crossing's unique crossing sign.

We sort the bump and stab events for a pair of SEEs by θ. There are four possible bump events (two outer bumps, two inner bumps), and sixteen possible stab events (two from each SVE/SEE pairing and there are eight pairs). The crossing list starts out empty. We process each event in sorted θ order, and update the current crossing list as follows.

If the event is a bump, then it is trivial to determine if the bump is a bump event or an unbump event. If current crossing list has zero crossings, then this is a bump, otherwise current crossing list has two crossings, and this is an unbump event. We describe why it is not possible to have only one current crossing in the crossing list when the event is a bump or unbump in Chapter 5.

If the event is a stab, it is trivial to determine that the event is really a stab (zero crossings) or unstab (two crossings). If there is only one crossing in the current crossing list, then we look at the current crossing sign. If the sign of the current crossing matches the sign of the event's crossing, then this event is an unstab event, and we remove the crossing from the current crossing list. Otherwise, the signs do not match and this is a stab event, and we add the event's crossing to the current crossing list.

Because all SVEs are incident on two SEEs, each SVE is involved in a pair of stab/unstab events: either unstab-stab, double stab, or double unstab. A double stab event introduces a pair of crossings to the convolution, see Figure 2.11. A double unstab removes a pair of crossings from the convolution, which is the reverse of the figure.

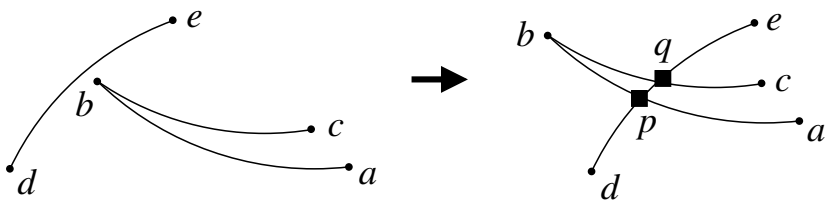


Figure 2.11: Type 2 double stab: Before the double stab event, SEE $\overline{ab}$ and $\overline{bc}$ are about to stab SEE $\overline{de}$ at b. After the double stab event, b has stabbed through de, creating new crossings p and q.

An unstab-stab event is when we remove a crossing from one SEE, and add a new crossing to another, essentially "moving" a crossing past an SVE, see Figure 2.12.

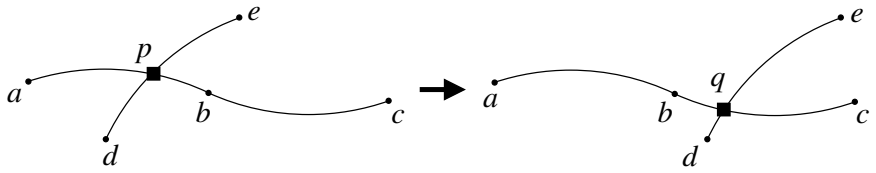


Figure 2.12: Type 2 unstab-stab event: Before the unstab-stab event, SEE $\overline{ab}$ has crossing p with SEE $\overline{de}$. After the unstab-stab event, $\overline{de}$ has crossing q with $\overline{bc}$.

2.6.6 Evolution of edge lists due to Type 2 events

As we vary θ, a bump, stab, or special stab event adds a crossing or pair of crossings to a pair of edge lists. For each new crossing added to an edge list, we know the orientation ϕ and angle θ at which it appears. We can instantiate the ϕs of the other crossings on the edge list at that θ. We then can insert the new crossing in the binary fashion explained in Section 2.5. For an unbump, unstab or a special unstab event,

we remove the crossing(s) from a pair of edge lists. For stab events, the crossing is added at the beginning or end of the edge list on the stabbing SVE's incident SEE.

2.6.7 Type 3 triangle flip events

When two crossings are newly adjacent on an edge list, we determine if they swap order on the edge list. There are three ways two crossings are newly adjacent: one of them is a new crossing, an old crossing was removed, or a *swap* occurred. A swap event is when two crossings exchange order on an edge list and the swapping occurs simultaneously on a triplet of SEEs, see Figure 2.13. We call the three simultaneous swap events a single Type 3 event, or simply a *triangle flip*. As we can see from the figure, a triangle region of three sides flips around on itself when all three edges flip their crossing orders. We must identify all triangle flips that occur so that we can update the arrangement accordingly.

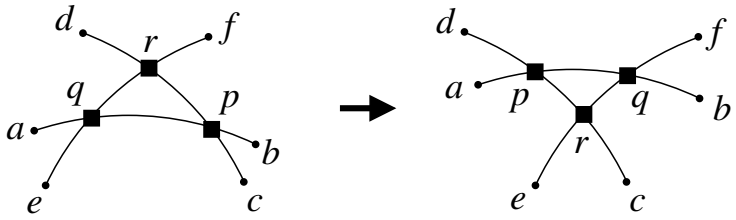


Figure 2.13: Type 3 triangle flip event: Before the triangle flip event, the crossing orders are $q < r$ on $\overline{ef}$, $q < p$ on $\overline{ab}$, and $r < p$ on $\overline{dc}$. After the event, the crossing orders are flipped: $r < q$ on $\overline{ef}$, $p < q$ on $\overline{ab}$, and $p < r$ on $\overline{dc}$.

To determine when adjacent crossings swap on the edge list, we need to calculate three-way circle intersections. The calculations for the three-way circle intersections

are explained in Section 2.7.3. The calculations return pairs of points and angles: $\langle p, \theta \rangle$. The implicit constraints for each pair are

$$|p - (c_{fi} + \theta(c_{mi})| - (r_i)^2 = 0 \text{ for } i = 1, 2, 3. \tag{2.44}$$

This gives three implicit constraints for three implicit coordinates: two from p and one from θ. We can compute each ϕ for each SEE by looking at the outward normal angle at $\langle p, \theta \rangle$ on the SEE.

Given $\langle p, \theta \rangle$, we need to determine which pairs of crossings are actually swapping on each edge list. Because each crossing has a unique crossing sign, we can match the correct crossings by comparing with ϕ_i on all three SEEs.

If a pair of adjacent crossings are from the same bump event or from the same double stab event, we do not check for a swap of that pair. We do not check for a swap, because those crossings can not swap in order.

2.7 Floating point calculations

In this section we explain some calculations needed for the C-Space construction. Floating point calculations are used in this section for all of the computations. After the computations, the CLP method implicitly constrains all of the returned coefficients accordingly.

2.7.1 Circle-circle intersection

The circle-circle intersection algorithm is used in finding the initial crossing list, the bump events, and the stab events, Sections 2.5, 2.6.2, and 2.6.3, respectively. Let circles $\langle c_1, r_1 \rangle$ and $\langle c_2, r_2 \rangle$ intersect in two crossing points p_1 and p_2. This algorithm is only invoked if the signs of the primitives determined that two crossing points do exist on these two circles. The following intersection calculations are illustrated in Figure 2.14.

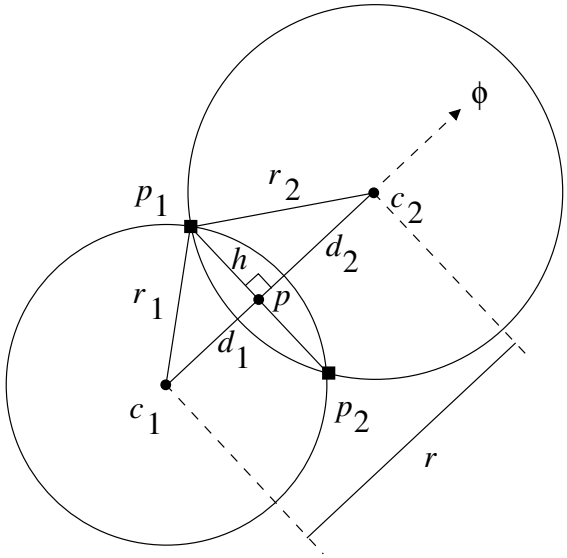


Figure 2.14: Crossing calculations of two circles

Let $w = c_2 - c_1$, and $\phi(w)$, $r(w)$ are its polar coordinates (ϕ and r for short). Let

$$d_1^2 + h^2 = r_1^2 \tag{2.45}$$

$$d_2^2 + h^2 = r_2^2. \tag{2.46}$$

Since $r = d_1 + d_2$, solving for d_1 yields

$$d_1 = \frac{1}{2r}(r_1^2 - r_2^2 + r^2). \tag{2.47}$$

Point p, the halfway point between p_1 and p_2, is computed by

$$p = c_1 + d_1 u(\phi), \tag{2.48}$$

where $u(\phi)$ is the unit vector in the direction of ϕ. The intersection points are then

$$p_1 = p + hu(\phi + \frac{\pi}{2}), \tag{2.49}$$

$$p_2 = p + hu(\phi - \frac{\pi}{2}), \tag{2.50}$$

where h is computed by solving Equation (2.45) for h:

$$h = \sqrt{r_1^2 - d_1^2}. \tag{2.51}$$

2.7.2 Ray-circle intersection

For arrangement construction in Chapter 3 and path planning computations in Chapter 4, we need ray-circle intersection calculations for *ray casting*. Recall that in Section 2.5 each SEE has a corresponding circle $\langle c, r \rangle$. Given the ray $p + tv$ where t is a scalar, we want to know at what t does the ray intersect the circle. Figure 2.15 illustrates this example of ray casting.

The ray intersects the circle when

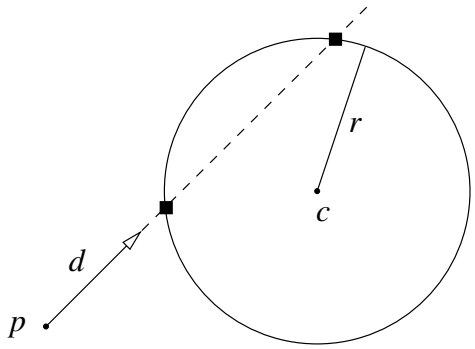


Figure 2.15: Ray crossing a circle

$$|p + tv - c| = r \tag{2.52}$$

$$\Updownarrow$$

$$|(p - c) + tv|^2 = r^2 \tag{2.53}$$

$$\Updownarrow$$

$$t^2|v|^2 + 2t(v \cdot w) + |w|^2 - r^2 = 0, \quad \text{with } w = p - c. \tag{2.54}$$

Solving for t yields

$$t = \frac{-v \cdot w \pm \sqrt{(v \cdot w)^2 - |v|^2(|w|^2 - r^2)}}{|v|^2}. \tag{2.55}$$

Equation (2.55) gives t_1 and t_2. Let $\delta = (v \cdot w)^2 - |v|^2(|w|^2 - r^2)$. If $\delta > 0$ then the ray intersects the circle at two points $p + t_1 v$ and $p + t_2 v$. If $\delta < 0$ then the ray never intersects the circle. If $\delta = 0$ then the ray is tangent to the circle and this is a floating point degeneracy. We explain how we deal with these ray tracing degeneracies in Chapter 3.

2.7.3 Three-way circle intersections

In Section 2.6.7, Type 3 swap events correspond to triangle flips in the convolution as θ is varies from 0 to 2π. The triangle flip corresponds to three SEEs crossing at the same point p at the swap event's θ. We call this pair $\langle p, \theta \rangle$ a *criticality*. To determine all criticalities of a triplet of SEEs we look at their corresponding circles $\langle c_1, r_1 \rangle$, $\langle c_2, r_2 \rangle$, and $\langle c_3, r_3 \rangle$. Recall from Section 2.4, each SEE center $c_i = c_{f_i} + \theta(c_{m_i})$.

Since p is a critical point, it lies on all three circles: $|c_i - p| = r_i$ for $i = 1, 2, 3$, see Figure 2.16.

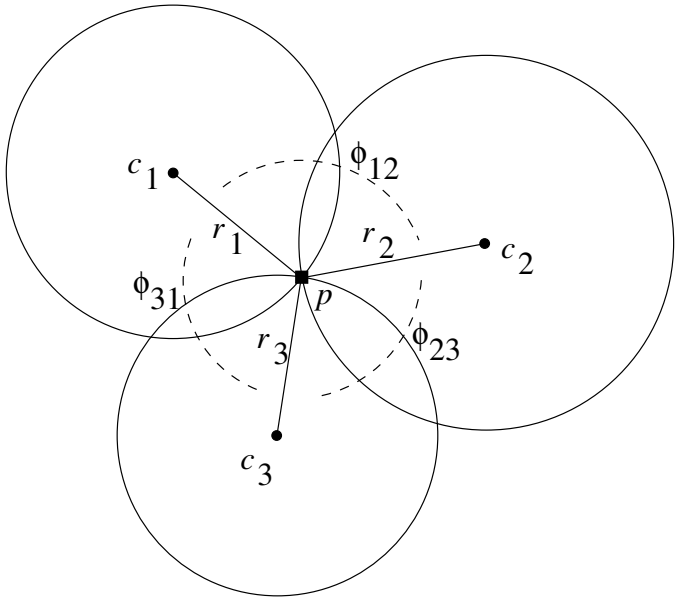


Figure 2.16: Three circles meeting at a critical point p.

Let ϕ_{12} be the angle of c_1, p, c_2 from the figure. By the law of cosines ($c^2 = a^2 + b^2 - 2ab\cos\gamma$),

$$(c_1 - c_2)^2 = r_1^2 + r_2^2 - 2r_1 r_2 \cos\phi_{12} \;\Leftrightarrow\; \cos\phi_{12} = \frac{r_1^2 + r_2^2 - (c_1 - c_2)^2}{2r_1 r_2}. \qquad (2.56)$$

We expand $(c_1 - c_2)^2$ from Eq.(2.56):

$$\begin{aligned}(c_1 - c_2)^2 &= ((c_{f_1} - c_{f_2}) + \theta(c_{m_1} - c_{m_2}))^2 \\ &= (c_{f_1} - c_{f_2})^2 + (c_{m_1} - c_{m_2})^2 + 2((c_{f_1} - c_{f_2}) \cdot \theta(c_{m_1} - c_{m_2})), \end{aligned} \quad (2.57)$$

where $(\cdot)$ is the dot product, and $\theta(c_{m_1} - c_{m_2})$ is the rotation matrix from Eq. (2.18). Therefore $\cos\phi_{12}$, from Eq. (2.56), is a linear function of $\cos\theta$ and $\sin\theta$. Similarly we can do the same for $\cos\phi_{23}$ and $\cos\phi_{31}$, where ϕ_{23} and ϕ_{31} are the respective angles of c_2, p, c_3 and c_3, p, c_1. From the figure we know that

$$\phi_{12} + \phi_{23} + \phi_{31} = 2\pi. \quad (2.58)$$

From Eq.(2.58) angles ϕ_{12}, ϕ_{23}, and ϕ_{31} satisfy the following trigonometric identity:

$$\cos^2\phi_{12} + \cos^2\phi_{23} + \cos^2\phi_{31} - 2\cos\phi_{12}\cos\phi_{23}\cos\phi_{31} - 1 = 0 \quad (2.59)$$

We substitute the formulas from Eq.(2.56) and Eq.(2.57) for $\cos\phi_{12}$, $\cos\phi_{23}$, and $\cos\phi_{31}$ in Eq.(2.59), resulting in a cubic polynomial in $\cos\theta$ and $\sin(\theta)$. To constrain θ, we also know from trigonometry the Pythagorean identity:

$$\cos^2\theta + \sin^2\theta = 1 \quad (2.60)$$

This gives us two multivariate equations for two variables $\cos\theta$ and $\sin\theta$. To solve the two simultaneous equations, we follow the *eigenvector* method from Auzinger and Stetter [1, 2, 42]. The input to the method is a system of multivariate polynomial equations $F(x) := \{f_1(x), .., f_n(x)\}$, where $f_i(x) = f_i(x_1, ...x_n) = 0$ for $i = 1, ..., n$. The number of roots in the system $F(x)$ is the Bezout's number $\mu = \prod_{i=1}^{n} \mu_i$, where μ_i is the degree of polynomial f_i. The method uses standard Gauss elimination for

a large, sparse matrix, and eigenvector computation for a matrix of dimension μ. The eigenvector method returns μ real and complex roots. For our purposes we keep only the real roots $\langle x_1, x_2\rangle = \langle \cos\theta, \sin\theta\rangle$. There are six possible real roots, since $\mu_1 = 3, \mu_2 = 2$, and $\mu = \mu_1\mu_2 = 6$, with f_1 from Eq.(2.59) and f_2 from Eq.(2.60). Each real root $\langle x_1, x_2\rangle$ is converted to $\theta = \tan^{-1}\left(\frac{x_2}{x_1}\right)$, where each θ is the critical angle representing where the three circles meet at the a critical point p. To find p, let (c_{1_x}, c_{1_y}), (c_{2_x}, c_{2_y}), (c_{3_x}, c_{3_y}) be the centers of the circles at critical angle θ. The three circle equations are

$$(x - c_{1_x})^2 - (y - c_{1_y})^2 - r_1^2 = 0 \tag{2.61}$$

$$(x - c_{2_x})^2 - (y - c_{2_y})^2 - r_2^2 = 0 \tag{2.62}$$

$$(x - c_{3_x})^2 - (y - c_{3_y})^2 - r_3^2 = 0. \tag{2.63}$$

Subtracting Eq.(2.62) from Eq.(2.61) and subtracting Eq.(2.63) from Eq.(2.62) respectively yields

$$c_{1_x}^2 - c_{2_x}^2 + c_{1_y}^2 - c_{2_y}^2 - r_1^2 + r_2^2 - 2(c_{1_x}x + c_{1_y}y - c_{2_x}x - c_{2_y}y) = 0 \tag{2.64}$$

$$c_{2_x}^2 - c_{3_x}^2 + c_{2_y}^2 - c_{3_y}^2 - r_2^2 + r_3^2 - 2(c_{2_x}x + c_{2_y}y - c_{3_x}x - c_{3_y}y) = 0. \tag{2.65}$$

Solving the two linear equations Eq.(2.64) and Eq.(2.65) for the two unknowns x and y results in critical point $p = \langle p_x, p_y\rangle = \langle x, y\rangle$. So for every real root $\langle \cos\theta, \sin\theta\rangle$, we can find the criticalities $\langle p_i, \theta_i\rangle$ for $1 \leq i \leq \mu$.

Proof of the trigonometric identity in Eq.(2.59). From Eq.(2.58), let

$$\cos\phi_1 = \cos(2\pi - (\phi_2 + \phi_3)) = \cos(-(\phi_2 + \phi_3)) = \cos(\phi_2 + \phi_3), \tag{2.66}$$

where angles ϕ_1, ϕ_2, and ϕ_3 replaced ϕ_{12}, ϕ_{23}, and ϕ_{31} respectively. The trigonometric sum formula is

$$\cos(\phi_2 + \phi_3) = \cos\phi_2 \cos\phi_3 - \sin\phi_2 \sin\phi_3. \tag{2.67}$$

The trigonometric identity Eq.(2.59) can then be rewritten as follows using Eq.(2.66) and Eq.(2.67).

$$\begin{aligned}
\text{Eq.(2.59)} &= \cos^2\phi_2 + \cos^2\phi_3 + \cos^2\phi_1 - 2\cos\phi_2\cos\phi_3\cos\phi_1 - 1 \\
&= \cos^2\phi_2 + \cos^2\phi_3 + (\cos\phi_2\cos\phi_3 - \sin\phi_2\sin\phi_3)^2 \\
&\quad - 2\cos\phi_2\cos\phi_3(\cos\phi_2\cos\phi_3 - \sin\phi_2\sin\phi_3) - 1 \\
&= \cos^2\phi_2 + \cos^2\phi_3 + \cos^2\phi_2\cos^2\phi_3 - 2\cos\phi_2\cos\phi_3\sin\phi_2\sin\phi_3 \\
&\quad + \sin^2\phi_2\sin^2\phi_3 - 2\cos\phi_2^2\cos^2\phi_3 + 2\cos\phi_2\cos\phi_3\sin\phi_2\sin\phi_3 - 1 \\
&= \cos^2\phi_2 + \cos^2\phi_3 - \cos^2\phi_2\cos^2\phi_3 + \sin^2\phi_2\sin^2\phi_3 - 1 \\
&= \cos^2\phi_2 + \cos^2\phi_3 - \cos^2\phi_2\cos^2\phi_3 + (1 - \cos^2\phi_2)(1 - \cos^2\phi_3) - 1 \\
&= \cos^2\phi_2 + \cos^2\phi_3 - \cos^2\phi_2\cos^2\phi_3 + 1 - \cos^2\phi_2 - \cos^2\phi_3 \\
&\quad + \cos^2\phi_2\cos^2\phi_3 - 1 \\
&= 0. \qquad \square
\end{aligned}$$

2.8 Caveat of the CLP algorithm

Recall from Section 2.5 that we evaluate three difference primitives to determine if there are a pair of crossings or no crossings. In the case of a near tangency of two circles, if the three difference primitives determined there are a pair of crossings, then creating the implicit coordinates for these crossings "freezes" a difference primitive

at zero. To freeze a primitive at zero means the CLP can not determine a sign for the primitive. That is, we are unable to perturb the primitive away from zero unless we add a great amount of perturbation. Consequently, the C-Space algorithm fails since it needs the CLP to determine the primitive's sign.

Rather than having to reformulate the primitives since implicit constraints are convenient, Milenkovic added to the CLP the ability to request a sign of a primitive. We examine the three difference primitives that determined the pair of crossings. The near tangency of two circles corresponds to one of the three difference primitives being non-robust. If any of the three difference primitives for the circle-circle intersection test is non-robust, then we request the sign of the primitive that represents no crossings. Since that one condition is not met, the C-Space algorithm reports there are no crossings. We discuss in detail why this caveat in the CLP occurs and how an algorithm can request a sign of a primitive to the CLP in Chapter 5.

This near tangency of two circles affects parts of the algorithm that uses the circle-circle intersection tests: creating initial crossing lists, bump tests, and stab tests. For example, if a bump test reveals such a near tangency in one of its difference primitive checks, that means either two SEEs are close to bumping or close to unbumping, and we force the bump or unbump to not occur, respectively.

2.9 Algorithm pseudocode and complexity

In this section, we illustrate the algorithm pseudocode and complexity.

2.9.1 Pseudocode

Here is the pseudocode for the C-Space algorithm:

- Read Input: input a, and implicit constraints c, r, α on each arc.
- Create all Type 1 events and add to event queue Q, sorted by the positive-angle and angle-difference primitive on θ, from 0 to 2π.
- Create all SVE, SEEs, and determine the live SVEs and SEEs at initial $\theta = 0$.
- Create initial crossing lists at initial $\theta = 0$.
- Create Type 2 bump, stab, and special events and identify which are stab-unstabs, double-stabs, double-unstabs, bumps, or unbumps. Insert these events into Q.
- Create initial edge lists using the initial crossing lists. For adjacent crossings check for potential swaps and add those Type 3 events to Q.
- Create initial arrangement.
- Handle each event from Q starting at initial $\theta = 0$:
    - If Type 1: update live SVE/SEEs, and check if there was a Type 2 special event associated with this Type 1. If so, add/or remove that crossing from the edge lists on affected SEEs.
    - If Type 2 bump or stab: update edge list on affected SEEs. Check for new swaps on newly adjacent crossings, and add new swaps to Q.

- If Type 3: swap crossings, check for new swaps, add new swaps to Q.
- Update arrangement

- Finalize last arrangement

We describe how we create and update the arrangement in Sections 3.1 and 3.2, respectively. The arrangement construction is necessary for the path planning algorithm in Chapter 4.

2.9.2 Complexity

The running time of the algorithm is as follows. Given n input fixed arcs and m input moving arcs, there are nm SEEs and $2nm$ SVEs. Reading input and creating all Type 1 events, SVEs, and SEEs takes $O(nm)$ time. Creating initial crossing lists takes $O(n^2m^2)$ time. Creating Type 2 bump events takes $O(n^2m^2)$ and Type 2 stab events takes $O(4n^2m^2) = O(n^2m^2)$ time. Creating Type 2 special events takes $O(nm)$ time, since at most one special event per Type 1 event.

We can improve upon running time for bump and stab events with a bounded box approach, but we would still have to compare adjacent cell regions for bumps and stabs, and in worst case each cell is still $O(nm)$ in size, so it would still take $O(n^2m^2)$. However in practice this should speed up the running time.

The bump/stab/special events and Type 1 events are added to event queue Q, as explained in Section 2.9.1. At the end of the main loop, Q also includes all triangle flip events. Let k be the number of events in Q. The cost of inserting/removing an

event to/from Q is $O(k \log(nm))$. The cost of inserting a crossing into an SEE's edge list, because of bumps/stabs, is also $O(k \log(nm))$. The worst case of k is $O(n^3 m^3)$, where we have to check all triangle flips, meaning checking every triplet of SEEs. However, in most practical situations, k is much smaller.

The number of events, k, is *output sensitive*. An output sensitive algorithm is an algorithm whose running time depends on the size of the input and output, so the size of k depends on the size of the input and output. As we shall discuss in Chapter 3, each arrangement output of the C-Space algorithm includes a set of subedges, and each subedge in $3D$ is a patch of a face of the contact surface of the C-Space. If we consider the output to be the trapezoidalization of the contact surface, then the complexity of the output is the number of subedges: k. We perform at most one triangle flip per subedge, so the cost of checking for triangle flips is k. For bump/stabs, we perform at most one bump/stab per subedge, so the cost is also k. Thus the complexity is $O(n^2 m^2 + k \log(nm) + k \log(nm)) = O(n^2 m^2 + k \log(nm))$. In Section 3.4, we explain why the complexity gets an extra cost of $O(k^2 c)$ where c is the number of *disconnected components*.

Chapter 3

Arrangements

In this chapter we describe the arrangement algorithm. An arrangement is a subdivision of the plane into subvertices, subedges, and cells, as illustrated in Figure 3.1. A cell is just a planar region of the arrangement made up of zero or one outer *loop*, and zero or more holes. A loop/hole is generated by traversing the subedges/subvertices in a clockwise fashion.

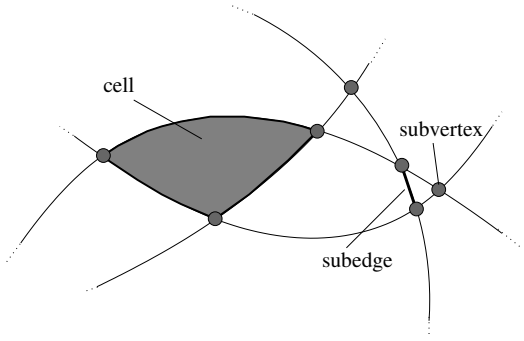


Figure 3.1: Arrangement structure

Each cell has a winding number that determines if the cell is an interior cell or exterior cell of the Minkowski sum. The behavior of this winding number, as Guibas et al. [20] prove, is that if vector t is in a cell of the arrangement of convolution $B \otimes -A$ with winding number w, then $(A + t) \cap B$ has w connected components.

In the rest of this chapter we discuss the following. We show how to create the initial arrangement that represents the C-Space in Section 3.1. Then we explain how to update the arrangement at each C-Space event in Section 3.2. If we were to output the arrangement after each C-Space event, then each arrangement explicitly defines the C-Space at some θ. However, these separate arrangements do not explicitly define the entire C-Space but only slices of it. Since we are updating the arrangement after each C-Space event, we can "connect" neighboring arrangements together, so that we can explicitly define the entire C-Space. We illustrate how a graph representation "connects" neighboring arrangements in Section 3.3, in order to define the entire C-Space.

3.1 Creating the initial arrangement

To create an arrangement, we first subdivide each SEE into *subedges* at the crossings on the SEE, see Figure 3.2. A subedge inherits its direction from the subdivided SEE, and it has two unique *subvertices*. Since each SVE is incident on two SEEs, each subvertex is either created from an SVE with degree two or created from a crossing with degree four. We call them *subvertices* because the SVE or crossing

is split into different subvertices at either Type 2 stab events for SVEs or Type 3 events for crossings. We explain the splitting of subvertices of SVEs and crossings in Section 3.2.

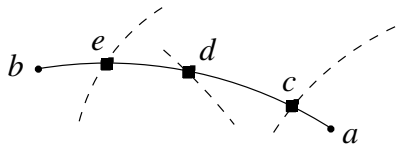


Figure 3.2: An SEE $\overline{ab}$ broken into subedges $\overline{ac}$, $\overline{cd}$, $\overline{de}$, and $\overline{eb}$ by crossings c, d, and e. The subvertices a and b have degree two, while c, d, and e each have degree four.

To create the initial arrangement, we subdivide each initially live SEE at the crossings in its initial edge list into subedges. We create subvertices for the initially live SVEs and initial crossings. We associate each *side* of each subedge with a loop by traversing the subedges in a counterclockwise fashion at each subvertex, as done in [34, 43]. A subedge has two sides based on the direction, see Figure 3.3.

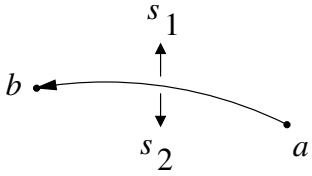


Figure 3.3: A subedge with sides s_1 and s_2, where s_1 is the right side of the subedge and s_2 is the left with respect to the subedge's direction.

After the are loops constructed, we generate the cells using the loops. However since we do not use a line sweep approach in the arrangement construction [34, 43], we need to correctly associate *disconnected components* to generate the cells. We call a set of *connected* loops a component. A set of loops G is connected if there exists a *path* leading from subvertices u to v for $u, v \in G$. A path is a sequence of subvertices

connected by subedges. Two components G_1, G_2 are disconnected if there does not exist a path from u to v where $u \in G_1$ and $v \in G_2$.

Recall that each cell is made up of zero or one outer loop and zero or more holes. If there are no disconnected components then generating the cells is trivial, for each loop, besides the outermost loop, is the outer loop of its own cell. The outermost loop of a component is always a hole in the *outermost cell*. The outermost cell is the unbounded outer cell of the arrangement.

To find the outermost loop, we use *randomized ray tracing* as follows. We pick a subvertex p and shoot a ray in a random direction ($p + tv$ with scalar t and vector direction v) to see if the ray crosses any subedges. Because all subedges are just pieces of SEEs, and SEEs have corresponding circles, we perform ray tracing by computing the intersections of the ray with the corresponding circles using the computations from Section 2.7.2.

We have the freedom of not using the CLP for ray tracing calculations. Ray tracing is a simple task, so there is no need to hinder performance by adding more primitives to the CLP. If the random direction of the ray is degenerate with any of the SEE's corresponding circles, we choose a new random direction for the ray. The primitives used for ray tracing can be checked using floating point methods.

The ray-circle computations return two scalar values t_1, t_2 representing where the ray crosses each circle; the crossings are $(p + t_1 v)$ are $(p + t_2 v)$. To check if the crossings lie on the subedge, we compare outward normal directions on the crossings

and the subvertices. The subedge with the highest scalar $t_{\max}$ crossing with the ray is the outermost subedge, see Figure 3.4.

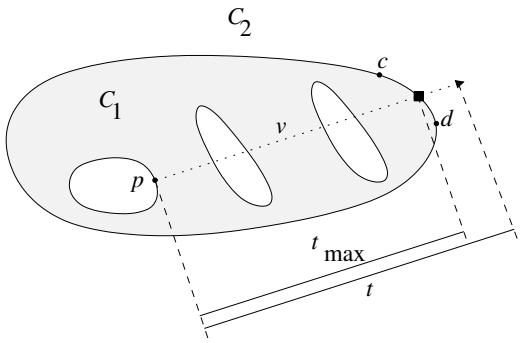


Figure 3.4: Ray tracing to find outermost subedge. The ray $p + tv$ crosses the outermost subedge $\overline{cd}$ at $p + t_{\max}v$, and the outer side of that subedge belongs to a hole of the outermost cell C_2.

The outer side of the outermost subedge is part of the outermost loop and consequently is a hole in the outermost cell. If the ray does not cross any subedge, then the outer side of the incident subedges of subvertex p are part of the outermost loop and is a hole in the outermost cell.

If there are disconnected components, then we have to find the outermost subedge of all disconnected components. From that outermost subedge, we again shoot a random ray. However, this time we find the inner-most subedge of all the cells constructed so far, to determine within which cell the disconnected component lies. Initially there is only the outermost cell, but as we process more disconnected components, more cells are generated and considered. If an inner-most subedge is found with lowest scalar $t_{\min}$ from the ray-circle computations, that means this disconnected component is a hole in that inner-most subedge's cell, see Figure 3.5.

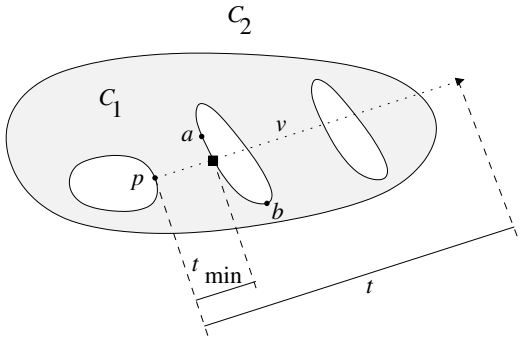


Figure 3.5: Ray tracing to find inner-most subedge. The ray $p + tv$ crosses the inner-most subedge $\overline{ab}$ at $p + t_{\min}v$, and the inner side of that subedge belongs to a hole in cell C_1. Consequently the hole containing subvertex p is also a hole in cell C_1.

If no such subedge exists, meaning the ray did not intersect any subedge, then this disconnected component is a hole in the outermost cell. The rest of the loops in the disconnected component will be outer loops in their own cells. We process all disconnected components in the same fashion, until each component is part of a cell. This procedure generates cells correctly for the disconnected components without the use of a line sweep.

The last step in the arrangement construction is adding winding numbers. As mentioned in Chapter 1, winding numbers help us determine which cells are in free space and which cells are in blocked space. Since our goal is to apply C-Space construction to path planning, finding the cells that represent free space is the whole intention. Let the outermost cell C_0 have winding number $w_0 = 0$, and let it also be the current cell $C_i = C_0$. Again, we follow the approach from [34, 43] for winding number assignment using a standard breadth-first search (BFS) algorithm. Let C_{i+1} be an *adjacent* cell to the current cell C_i. For C_{i+1} to be adjacent to C_i, both cells

have a common incident subedge e where C_{i+1} resides on one side, and C_i resides on the other. Recall each subedge has an inherited direction: one side is left and the other is right (Figure 3.3). If C_i resides on the right side of e, then $w_{i+1} = w_i + 1$. Otherwise C_i resides on the left side of e and $w_{i+1} = w_i - 1$. We visit adjacent cells in BFS fashion and assign winding numbers until all cells have been visited, resulting in each cell having a winding number. We know from [20, 35] that $w_i \geq 0$ for all C_i. So if $w_i > 0$, then C_i is part of blocked space. Otherwise C_i is part of free space.

3.2 Updating the current arrangement

In this section we show how to update the current arrangement as each C-Space event is processed. Initially the current arrangement is the initial arrangement, from Section 3.1. As described in the C-Space algorithm in Section 2.9.1, events are sorted in an event queue Q from 0 to 2π.

In the arrangement every cell has a unique set of subedges, so no two cells have the same set of subedges. Each event introduces new subedges and/or removes old subedges. Therefore each event also introduces new cells and removes old cells. Here are the different subedges affected by each event.

- Type 1 events. Each Type 1 event affects four SVEs and four SEEs. Therefore, four subvertices and four subedges are affected.

- Type 2 bump events. A bump gives two newly dead and six newly live subedges from the two bump crossings. An unbump is the converse.

- Type 2 stab events. In a unstab-stab event, referring back to Figure 2.12 in Section 2.6.5, there are a total of five newly dead subedges. Four are from the removed crossing and one is from the SEE being stabbed. There are also a total of five newly live subedges. Four are from the added crossing and one is from the SEE being unstabbed. For a double stab event, referring back to Figure 2.11, there are a total of three newly dead subedges and seven newly live subedges. A double unstab event is the converse: three newly live subedges and seven newly dead subedges.

- Type 2 special events. A special stab event involves two newly dead subedges and four newly live subedges, and the special unstab event is the converse.

- Type 3 events. A total of nine subedges are newly dead and nine subedges are newly live.

We update the arrangement by performing new loop traversal around the newly live subedges. If any loop is part of a new disconnected component, we place the outermost loop of the component as a hole in the correct cell. The procedure for connecting disconnected components is explained in Section 3.1. Every newly dead subedge is part of a newly dead cell, so each loop of the dead cell must be processed again in case the loop still has live subedges. These loops are again disconnected components, and we deal with them in the same fashion as the disconnected components from the newly live subedges.

Each subedge has two unique subvertices; however, each SVE or crossing can have one or more subvertices that represent it. To maintain the topology of the arrangement, we cannot allow a subvertex to ever be passed by a subedge. Once the subedge passes the subvertex, we consider it a new subvertex. A subedge passing a subvertex is equivalent to a Type 2 stab event. It is then necessary to split the SVE into multiple subvertices, one for each Type 2 stab event that it is involved in. Splitting is done at each event's angle θ.

For a crossing, we maintain the topology of the arrangement by never allowing a subvertex to ever be swapped. Once a swap occurs between two adjacent subvertices, we consider them to be new subvertices. We split the crossing into multiple subvertices, one for each Type 3 event it is involved in at the event's θ. In Section 3.3 we show the importance of splitting an SVE or crossing into multiple subvertices at events in the graph representation.

3.3 Graph representation

This section explains how we convert the arrangements from Section 3.1 and Section 3.2 to a graph representation. We relate cells to subedges as follows. Recall from Section 3.2 that each cell of the arrangement has a unique set of subedges. So when a C-Space event introduces or removes cells from the current arrangement, it is possible that the event did not remove all subedges of a dead cell from the arrangement. That is, the still-live subedges from the dead cell are part of some new cell in the

current arrangement. So to "connect" the arrangements before and after the current C-Space event, we link the dead cells to the new cells through the still-live subedges. For every C-Space event, we can link neighboring arrangements by linking cells to subedges and vice-versa.

We relate subedges to subvertices as follows. Recall from Section 3.1 that every subedge has two unique subvertices. So when a C-Space event introduces or removes subedges from the current arrangement, it is possible that the event did not remove both unique subvertices from the arrangement. So to "connect" the arrangements before and after the current C-Space event, we link dead subedges to new subedges through still-live subvertices. For every C-Space event, we can link neighboring arrangements by linking subedges to subvertices and vice-versa.

For the graph representation of the arrangement, we introduce *side* to subedges and subvertices, see Figure 3.6. A subedge bounds two cells, and it has two sides. We call each side of a subedge a *subedge-side*. We link each subedge-side to a cell. A subvertex has two sides if its from an SVE and four sides if its from a crossing. Each side of a subvertex is linked to two subedge-sides, as shown in the figure. Figure 3.7 illustrates an example of a subvertex with four sides. We call each side of a subvertex a *subvertex-side*.

By creating sides for subedges and subvertices, we can do the following with the graph representation.

- At each cell we can return a list of subedge-sides.

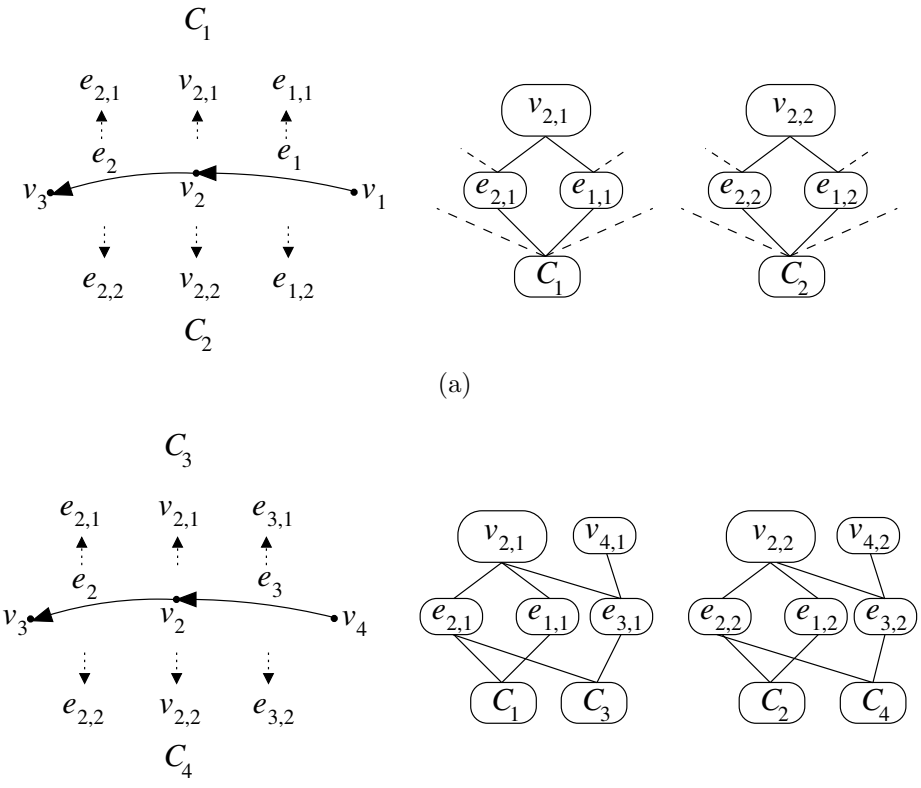


Figure 3.6: Sides of a subvertex: (a) Two sides of a subvertex are represented in the arrangement graph. If we let cell C_1 be in free space, then the subvertex-side $v_{2,1}$ and subedge-side" $e_{1,1}$ and $e_{2,1}$ are also in free space. The graph on the right hand side shows how the elements link to each other. (b) Subvertex v_1 is replaced by v_4, and cells C_1, C_2 are replaced by C_3, C_4 respectively. The graph representation is updated to include the new elements. C_1 now linked to C_3, so C_3 is also free space now.

- At each subedge, given a side we can return a list of cells on that side. Also we can return the subvertex-side its two subvertices on that side of the subedge.

- At each subvertex, given a side we can return the two subedge-sides on that side of the subvertex.

At every arrangement update, we add/remove cells, we keep a list of cells and subvertex-sides at both sides of a subedge, and we keep a list of subedge-sides at

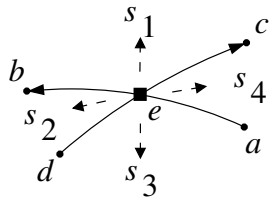


Figure 3.7: Sides of a subvertex from a crossing. Side example: Side s_1 has the left side of $\overline{ec}$ and the right side of $\overline{eb}$.

either the two or four sides of a subvertex. The elements of our graph are cells, subedge-sides, and subvertex-sides, and the links to elements in our graph are these lists. So each cell is one element, each subedge is two elements, and each subvertex is two or four elements of the graph.

Recall from Section 3.2 that we did not allow a subedge to sweep pass a subvertex or two subvertices to swap each other. Instead we create multiple subvertices that represents the SVE or crossing. This requirement allows a correct graph representation. If we do not create multiple subvertices, then this results in an incorrect graph representation, see Figure 3.8.

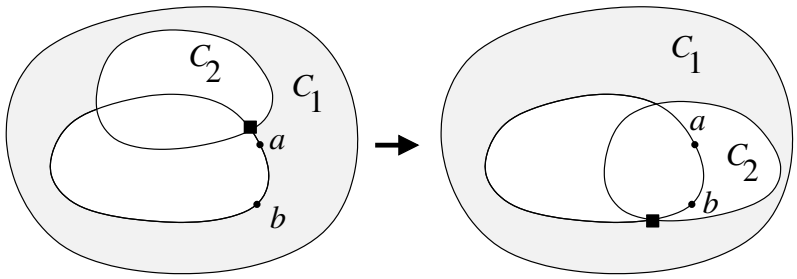


Figure 3.8: Incorrect graph representation due to stab events. Before the stab events on SVE a and b, the outer side of subedge $\overline{ab}$ belongs in cell C_1. After the stab events, the outer side of $\overline{ab}$ belongs in cell C_2. Consequently it is possible we linked a free cell to a blocked cell since C_1 and C_2 have different winding numbers, thus creating an incorrect graph representation.

The last information required for the graph representation is the *life span* of each element: cells, subedges, subvertices. The life span is the angle interval $[\alpha, \beta]$, where α is the angle at which the element is newly live, and β is the angle at which it is newly dead. Cells and subvertices have only one life span. It is possible for subedges to have more than one life span, because two SEEs can bump and unbump each other. Consequently the subedge before the bump and after the unbump is equivalent.

3.4 Arrangement complexity

From Section 2.9.2, we stated that the complexity of the C-Space algorithm is $O(n^2m^2 + k\log(nm) + k^2c)$. As we have explained in this chapter, we need ray casting to construct the arrangement at each event. Ray casting is used to "connect" disconnected components to each other. Let c be the number of disconnected components. Then each ray cast costs $O(kc)$, since we check at the worst case k subedges for each ray we cast, and we do this c times. The worst case for c is $O(k)$; however, in most practical situations c is much smaller.

The arrangement complexity is as follows. In the worst case every event is a bump event. If every event is a bump event, then in the worst case the bump event splits a cell into two cells, where the cell that split has $O(c)$ disconnected holes in it. So the ray cast cost at each arrangement is $O(kc)$ and there are k arrangements due to k events. So complexity of creating arrangements is $O(k^2c)$.

Chapter 4

Path Planning and Plotting

In Section 4.1 we describe how to perform path planning with the aid of the C-Space construction described in Chapters 2 and 3. We then detail how we plot paths in in Section 4.2.

4.1 Planning the path

As discussed in Chapter 1, the goal for path planning is to find if a path exists for a robot from a start to a goal, while also avoiding any collisions with an obstacle. The C-Space construction takes the fixed region (the obstacle) and the moving region (the robot) and provides the arrangements necessary to build the graph representation from Section 3.3.

Given the start pose $\langle x_0, y_0, \theta_0 \rangle$ and goal pose $\langle x_1, y_1, \theta_1 \rangle$ as input, we first identify in which cells the start and goal poses reside. This is done with ray tracing, similar to the procedure in Section 3.1. We take a pose $\langle x_i, y_i, \theta_i \rangle$, and retrieve the live

SEEs and its subedges at θ_i. At point $\langle x_i, y_i \rangle$ we shoot a ray in a random direction and see if the ray crosses any subedge and keep only the innermost subedge, using the computations from Section 2.7.2. If the ray does cross subedges, then the robot resides in the live cell on the inner side of the innermost subedge. If the ray does not cross any subedge, then the outermost cell at θ_i is the live cell we are trying to find.

We find such a cell for the start pose and goal pose; call them cells C_0 and C_1 respectively. If either cell is not in free space, then there is no path; that is, one of the robot poses is already colliding with the obstacle. Both cells are required to be in free space before we can proceed. With both cells in free space, the robot can reach the goal pose from the start pose if and only if there exists a path from C_0 to C_1 in the graph representation described in Section 3.3. We use a standard BFS algorithm on the graph to find the path from the C_0 to C_1. We illustrate how to plot the path in Section 4.2.

4.2 Plotting the path

In this section we show how to *plot* a path found from the graph representation. Each plot is a position $\langle x_i, y_i \rangle$ with orientation θ_i. That is, we rotate the robot to the plot orientation and then translate the robot by the plot position. Initially let $\theta_i = \theta_0$, which is the orientation of the starting pose. The first plot is the starting pose.

For plotting it is not necessary to keep the side information for subvertices/subedges attained from each element in the graph, so we drop that information. For the path

we care only for which subvertices, subedges, and cells are on the path. The path has the pattern

$$(C_0 \rightarrow e_j \rightarrow ... \rightarrow e_k \rightarrow C_1).$$

We add to the path the innermost subedges from the earlier ray tracing that found the start/goal cells, for we need those subedges to plot correctly from the start/goal poses. We add the start cell's innermost subedge at the beginning and goal cell's innermost subedge at the end of the path. This results in a path where every other element is a subedge with the pattern

$$(e_0 \rightarrow C_0 \rightarrow e_j \rightarrow ... \rightarrow e_k \rightarrow C_1 \rightarrow e_1).$$

Therefore every odd element in the path plus the next two successive elements equals one of the two following triples.

- Subedge e_j to cell C to subedge e_k $(e_j \rightarrow C \rightarrow e_k)$;
- Subedge e_j to subvertex v to subedge e_k $(e_j \rightarrow v \rightarrow e_k)$.

We can think of each subedge element in the path as key frames in the animation of the plot. However, for a better visual we provide more detailed plots between each key frame element from our path by using the middle element between the key subedge elements to smooth out the plots.

Suppose the middle element between key frames e_j and e_k is a cell C $(e_j \rightarrow C \rightarrow e_k)$. That means we are trying to plot from subedge e_j to e_k while staying inside a cell C. Since e_j and e_k are adjacent elements to C, we know from the graph representation

that e_j and e_k both belong to loops that are part of C. Since we need to plot inside C, we make sure we are at an orientation where all the subedges are live on C. The orientation that satisfies this condition is $\theta_i = \alpha$, where α is the starting life span of C.

If e_j and e_k reside on the same loop of C, we can simply traverse the loop starting at e_j and plot every subvertex on the loop until we reach e_k. If e_j and e_k reside on different loops, we use ray tracing to traverse from one loop to the next, see Figure 4.1.

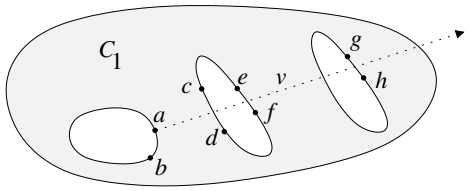


Figure 4.1: Plotting with ray tracing. To plot from $\overline{ab}$ to $\overline{gh}$ in cell C_1, we use ray tracing since those two subedges do not belong on the same loop in C_1. The ray enters another loop at subedge $\overline{cd}$ and exits at subedge $\overline{ef}$, so the plot has to traverse around that loop from $\overline{cd}$ to $\overline{ef}$ before it can continue to the loop containing $\overline{gh}$. Once it enters the loop of $\overline{gh}$, a simple traversal around the loop will give us the path.

From the loop of e_j we shoot a ray in the direction of the loop of e_k. If the ray crosses the loop of e_j, we have to traverse around the loop until we reach the subedge where the ray exits the loop. From that subedge we follow along the ray until it enters another loop. If this loop is not the loop of e_k, we follow the same procedure and traverse this loop until we find the subedge where the ray exits. We repeat this procedure until we enter the loop of e_k where we traverse this final loop to reach e_k. As we traverse this path, we plot along each subvertex of the path.

For finer plotting of the path, we can plot more points at each subedge instead of just the points at the subvertices. That is, we can plot points along the circular arc of the subedge, because the correct path travels along the arc. That path will guarantee no collision. Plotting points along the circular arc is not like a sampling-based path planning method. Unlike those algorithms, we know the entire path that is guarantee to avoid collision.

Suppose the middle element between key frames e_j and e_k is a vertex v ($e_j \rightarrow v \rightarrow e_k$). Instead of plotting at a fixed orientation on different subvertices, we have to plot at different orientations at the same subvertex. We are essentially "riding" the subvertex from one orientation to another, since the subvertex is in different positions as the orientation changes. To plot from subedge e_j to e_k while traveling along a subvertex v, we just update the orientation from the current orientation θ_i to an orientation where e_k is live. We need not worry about the life span of v, since its life span encompasses both the life spans of e_j and e_k, or else it wouldn't have been the subedge's subvertex. The goal orientation is then $\theta_i = \alpha$, where α is one of the starting life spans of subedge e_k. So we plot v once at θ_i and then once at α. We then update the current orientation θ_i to α: $\theta_i = \alpha$.

For finer plotting of the path, we can plot at more orientations at subvertex v. That is, we can plot points along v at equal increments of the angle from $\theta_i \rightarrow \alpha$ that can be user specified.

When all elements of the path are plotted we plot the final plot which is the robot in the final pose. By following the path along each plot, we visually see how the robot travels from the start pose to the goal pose.

Chapter 5

Controlled Linear Perturbation

The input to the CLP algorithm is the configuration space functions and the configuration space input from Chapter 2. The input is defined by $x = (x_i, ..., x_n)$. There are three types of functions:

- A function can return a sign (primitive): $f_j(x) > 0$.
- A function can be constrained to zero: $g_k(x) = 0$. These are called implicit constraints.
- A function can return an explicit value: $y = h_l(x)$. These are called explicit functions.

To allow for perturbation, we assign *velocities* to the parameters and the functions. The velocity is the direction to perturb. Velocity of the input x is denoted $v = (v_i, ..., v_n)$. The perturbation x_i' of each parameter x_i is denoted $x_i' = x_i + v_i t$ for some time $t \geq 0$. For each f_j, the *safety margin* is written as $k_j \mu$ with μ the floating point

rounding unit and k_j the error estimation for f_j. The CLP computes $x' = (x'_i, ..., x'_n)$ that makes every f_j *safe* with sign s_j: $s_j f_j(x') \geq k_j \mu$ where $s_j = \pm 1$. Safety margin is the amount of perturbation needed for a non-robust primitive to be safe, and safe means the non-robust primitive's sign is consistent with the rest of the primitives after the small perturbation.

The velocity of a function is computed by the dot product $\nabla f_j \cdot v$ with $\nabla f_j = (\frac{\partial f_j}{\partial x_1}, ..., \frac{\partial f_j}{\partial x_n})$, where ∇f_j is the gradient vector of $f_j(x)$ at x. Each component of the gradient, $\frac{\partial f_j}{\partial x_i}$, is the partial derivative of f_j with respect to parameter x_i. If the function is an implicit constraint we constrain the function and its velocity to equal zero: $\nabla g_k \cdot v = 0$. The velocity of an explicit valued function is $\nabla h_l \cdot v$.

Each input parameter receives a random initial velocity: $|v_i| \leq 1$. Each implicit parameter receives an initial velocity constrained by the gradient of the implicit constraint. Each explicit parameter receives the velocity of the function that computed them to it.

In the CLP method, if a primitive is near zero, we can't tell the sign of the primitive. For these non-robust primitives, we return the sign of its velocity. So if the value of the function is small and the velocity is large, then the sign is true for a small value of t. Hence the perturbation is small. Because we assigned random velocities to the input parameters, there is high probability that the velocities of the primitives are large.

For the non-robust primitives, the CLP figures out the minimum value of t that will make it safe, according to a linear estimation of the value of the function. It

keeps track of $t_{\min}$, the smallest value of t sufficient to make all unsafe primitives safe. For each new primitive the CLP encounters, it estimates the value at $t = t_{\min}$, and if that value is unsafe, $t_{\min}$ is increased. For each safe primitive, we also estimate a value $t_{\max}$ at which it would become unsafe. If $t_{\min}$ ever exceeds $t_{\max}$, we restart the algorithm with the current value of $t_{\min}$. When the algorithm runs again, the previously safe primitive with $t_{\max}$ is unsafe and is given the opposite sign. The CLP algorithm also needs a restart if $s_j f_j < k_j \mu$ for some j, which means the perturbation along v has too large a linearization error. The CLP picks a new random v for x, and the algorithm restarts with $t_{\min} = 0$.

To find the smallest possible perturbation, the CLP calls CPLEX to solve a linear program (LP). Each LP that is solved is costly. We show how costly each LP call is in our experimental data in Chapter 6. Recall from Section 2.8 that the CLP has the ability to request a sign of a primitive. Because we can't count on random velocities to enforce this sign, the CLP is forced to call an LP for each set sign request.

Chapter 6

Testing Results

The experiments were executed on a 2.40 GHz Intel(R) Core 2 CPU machine with 2.0 GB of RAM. Linear Programs were solved using the ILOG CPLEX 11.0 mathematical programming library [25]. The source code was written in GNU C++ for Linux.

Each experiment contains one fixed obstacle region and one moving robot region. We supplied start/goal poses for each experiment. To scale we simply multiply the input vertices of the robot by the scale factor r. Assuming the initial set does have a path from start to goal, we find a suitable initial scaling factors at which the minimum scale $r_{\min}$ is feasible, while the maximum scale $r_{\max}$ is infeasible. We subdivide the scales until we reach a terminating point, where we attain two adjacent floating point scale values (r_i, r_j) such that the lower of the two values, r_i, gives a path and the latter, r_j does not. This procedure terminates because our implementation is in double precision, and there are only 52 bits in the mantissa by the IEEE 754 standard [24].

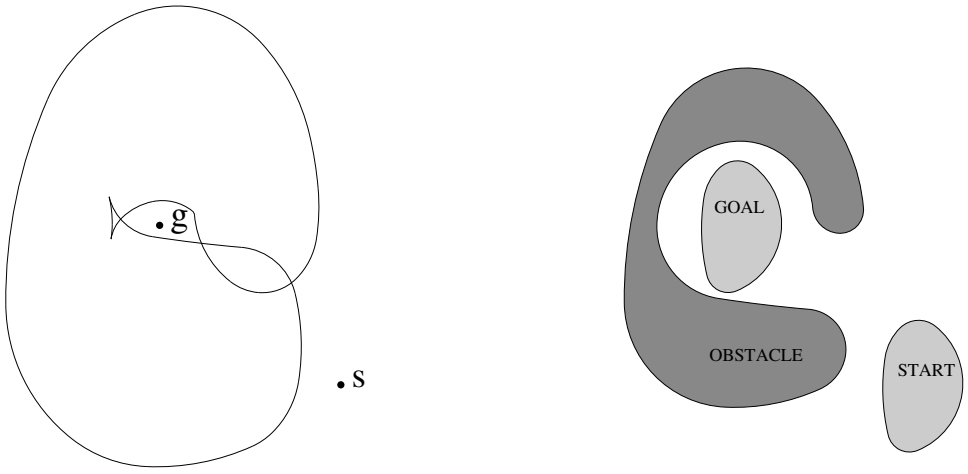
Also, since we subdivide in a binary fashion, the procedure takes no more than 52 iterations.

To scale by a factor of r, we are not truly dilating the robot. To dilate, we simply compute the Minkowski sum of the object with a circle of radius r. However, our goal in scaling is to put a stressing example through the system. We also choose not to dilate the robot because the Minkowski sum of the object with a circle is not guaranteed to be smoothed. This would force some changes in how we handle input. That is, we have to take into account the smoothing arcs mentioned in Section 2.1.

For path planning, as described in Chapter 1, the method of choice is to sample the immediate space and branch off to find the goal through greedy heuristics described in Section 1.1.4. By scaling the robot, there is a point in the chosen path that the object can no longer fit through. We have created such examples, and call these narrow channels in the path *bottlenecks*. At the bottleneck, normal sampling methods require a lot of sampling to fit through. In our case we were able to fit through the obstacle to the last floating point digit.

The first experiment, Experiment 1, is illustrated in Figure 6.1. The experiment has an obstacle made from 15 circular arc segments, and a moving robot with 5 circular arc segments. Using the subedges and subvertices from the arrangement computation from Chapter 3, we illustrate the 3D manifold for Experiment 1 in Figure 6.2.[1]

[1]**Acknowledgment**: The 3D manifold image for Experiment 1 was created by Adam McMahon, at the University of Miami, FL.



(a) (b)

Figure 6.1: Configuration Experiment 1. (a) The convolution at $\theta = 0$ has start and goal positions of the robot located at points s and g. (b) The start and goal poses of the robot region with the obstacle region.

In Experiment 1, $r_{\min} = 1.0$ and $r_{\max} = 1.2$. From the experiment we find two adjacent floating point numbers where a path is found (r_i) and a path is not found (r_j):

$$r_i \approx 1.1930753191765102...$$

$$r_j \approx 1.1930753191765104...$$

This difference between r_i and r_j is $|r_i - r_j| < 2.3 \times 10^{-16}$. Because of round off error in IEEE 754 double precision, there is no floating point number between r_i and r_j, so Experiment 1 terminates after 50 iterations. Figures 6.3 and 6.4 illustrate the first and last iteration where a path is found, respectively.

Table 6.1 records the C-Space information for Experiment 1. The C-Space information we record in each experiment is as follows.

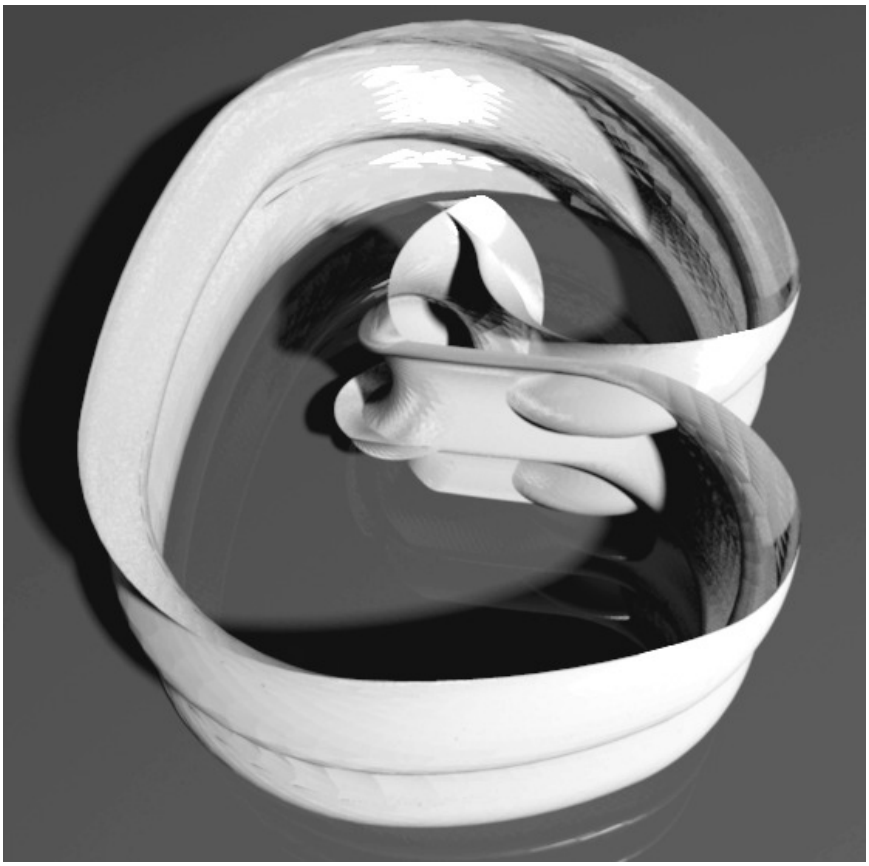


Figure 6.2: A top down view of the $3D$ manifold for Experiment 1. The height of the $3D$ manifold represents the change in angle θ.

- i: stands for the i-th iteration, where $i = 0$ is the first;
- Path: "yes" means a path exists, otherwise "no";
- T1: Number of Type 1 events;
- T2s: Number of Type 2 stab events (unstab-stabs, double stabs, double unstabs);
- T2S: Number of Type 2 special stab and unstab events;

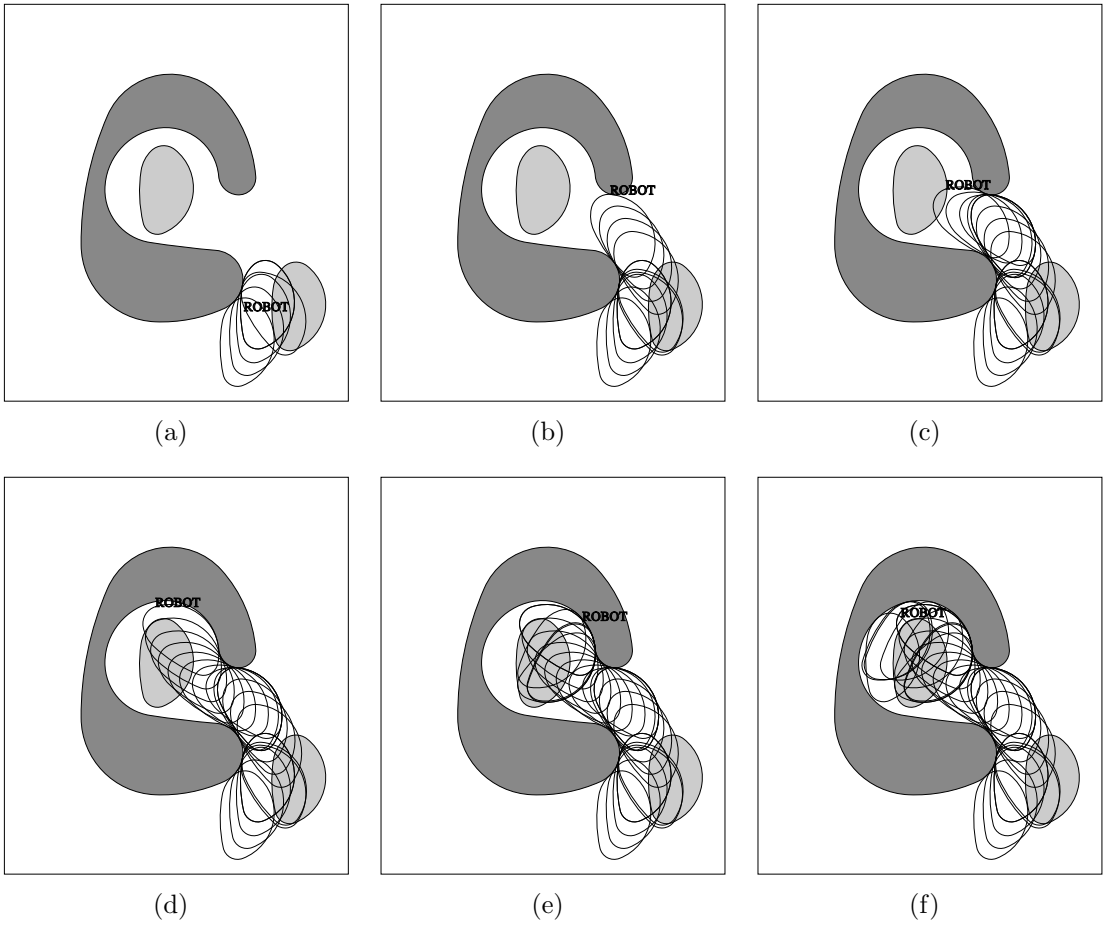


Figure 6.3: Experiment 1: First iteration with a path. First 5 figures are snapshots in increments of 5 plots. Last figure is the snapshot of the entire path. Last figure has 31 plots of the moving robot.

- T2b: Number of Type 2 bump and unbump events;
- T3: Number of Type 3 triangle flip events;
- Total: Total number of events;
- SV: Total number of subvertices created;
- SE: Total number of subedges created;
- C: Total number of cells created.

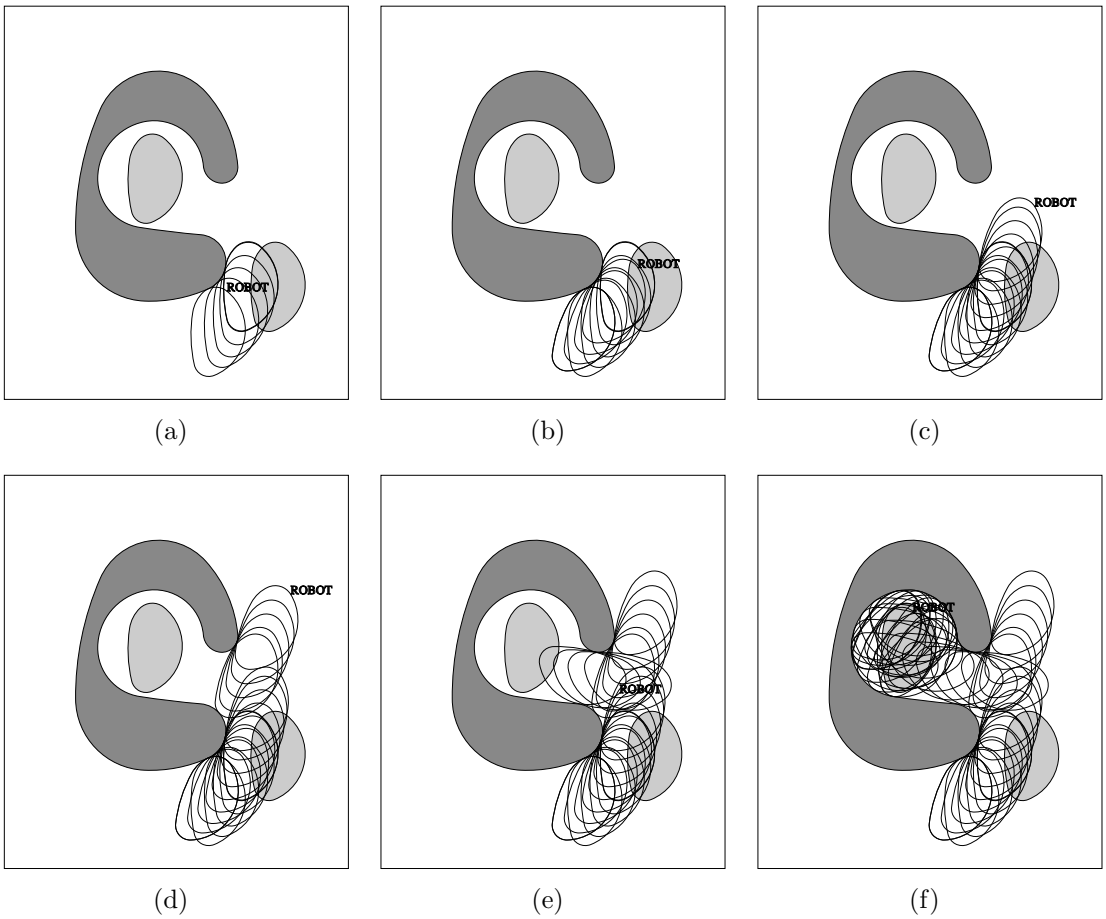


Figure 6.4: Experiment 1: Last iteration with a path. First 5 figures are snapshots in increments of 5 plots. Last figure is the snapshot of the entire path. Last figure has 48 plots of the moving robot.

From Table 6.1, we can see that the number of subvertices, subedges, and cells stabilize as we reach the terminating iterations. The actual scaling factors at each iteration are not important, so we omit them from the table. The number of Type 1 events is fixed: $13 \times 5 = 65$ with 13 edges from the obstacle and 5 edges from the robot. The number of Type 2 special events and number of Type 3 events are small in all iterations.

From Figures 6.3 and 6.4, it is apparent that the plotted paths lie on the contact boundary at each plot, i.e. the robot is always in contact with the boundary. This occurs because the path plotting from Chapter 4 explicitly plots along the contact boundary. One of the main goals of this dissertation is to find if a path exists in obstacles with narrow channels. In order to find a path through a narrow channel, the path consequently resides on the contact boundary around the narrow channel. One solution to find a path that lies entirely in free space is to apply the path of the largest scaled robot from Figure 6.4 to the initial robot from Figure 6.3. The result is Figure 6.5.

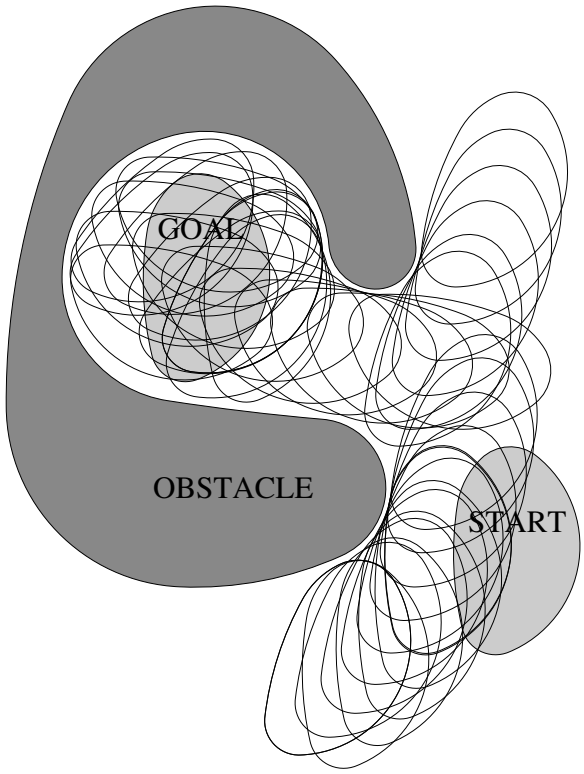


Figure 6.5: Plot of robot that lies entirely within free space.

i	Path	T1	T2s	T2S	T2b	T3	Total		SV	SE	C
0	yes	65	40	2	4	1	112		218	422	274
1	yes	65	42	2	4	1	114		218	423	282
2	yes	65	40	2	4	1	112		213	411	276
3	yes	65	40	2	4	1	112		213	409	276
4	no	65	44	4	0	1	114		222	426	278
5	yes	65	40	2	4	1	112		213	408	276
6	yes	65	40	2	4	1	112		213	408	276
7	yes	65	40	2	4	1	112		213	408	276
8	no	65	40	2	0	1	108		213	404	264
9	no	65	40	2	0	1	108		213	404	264
					...						
39	yes	65	40	2	4	1	112		213	408	276
40	yes	65	40	2	4	1	112		213	408	276
41	no	65	40	2	0	1	108		213	404	264
42	yes	65	40	2	4	1	112		213	408	276
43	no	65	40	2	0	1	108		213	404	264
44	yes	65	40	2	4	1	112		213	408	276
45	yes	65	40	2	4	1	112		213	408	276
46	no	65	40	2	0	1	108		213	404	264
47	yes	65	40	2	4	1	112		213	408	276
48	no	65	40	2	0	1	108		213	404	264

Table 6.1: Configuration space information on Experiment 1: Number of events and graph elements in each iteration. We list the first 10 and last 10 iterations.

Table 6.2 records the CLP information for Experiment 1. For each iteration we again keep track of the i-th iteration and if the path exists with a "yes" or "no" answer. The rest of the table is recorded as follows.

- EXP: Number of explicit coordinates;
- IMP: Number of implicit coordinates;
- TC: Total number of coordinates;
- F: Total number of functions (implicit constraints, explicit functions, and primitives);

- P: Number of primitives;
- EP: Number of non-robust primitives (value less than ϵ);
- CLP Time: Time for the CLP algorithm (mainly to call the Linear Program);
- Time: Total time taken for the C-Space algorithm, including CLP time;
- setSign: Number of primitives sign requests;
- B and A: We keep track of the rounding units of ϵ needed to be perturbed for the input before (B) and after (A) a final LP call for error estimation. This value is achieved by $\frac{v_{\max}}{v_{\min}}$, where $v_{\max}$ is the maximum absolute value of input coordinate velocities and $v_{\min}$ is the minimum absolute value of non-robust primitive velocities. If the number of rounding units is reasonably small, we do not make a final LP call for error estimation (for speed-up). So when column A equals column B, then there is no final LP call. If the value is 0, then no perturbation is needed because all primitives are robust.

From Table 6.2, there are an average of about 190,000 explicit coordinates and 119,000 implicit coordinates. There are about 148,000 primitives with signs, and only a handful of those are non-robust primitives. Non-robust primitives occurred when no path existed. These were caused by forced primitive signs. Only in the cases where we forced primitive signs did we call a Linear Program, which affected running time. For example, in iteration $i = 41$, it took 110 total seconds to call the LP twice; the

time it takes for the configuration space is negligible. The number of rounding units of perturbation were reasonable, so we did not force final LP calls to measure error.

i	Path	EXP	IMP	TC	TP	SP	EP	CLP Time	Time	CLP	setSign	B	A
0	yes	189854	118599	308494	455906	147453	1	234	267	0	0	0	0
1	yes	191065	118983	310089	458228	148180	1	247	289	0	0	0	0
2	yes	190841	118993	309875	457995	148161	1	262	298	0	0	0	0
3	yes	190941	119009	309991	458142	148192	1	252	300	0	0	0	0
4	no	191397	119305	310743	459539	148837	1	270	309	0	0	0	0
5	yes	190962	119031	310034	458252	148259	1	267	310	0	0	0	0
6	yes	191013	119047	310101	458334	148274	1	251	308	0	0	0	0
7	yes	191048	119063	310152	458411	148300	1	273	306	0	0	0	0
8	no	190428	119031	309500	457598	148139	1	254	298	0	0	0	0
9	no	190420	119031	309492	457598	148147	1	266	298	0	0	0	0
						...							
39	yes	191034	119047	310122	458364	148283	1	261	306	0	0	0	0
40	yes	191024	119047	310112	458357	148286	1	269	305	0	0	0	0
41	no	190436	119031	309508	457605	148138	5	11138	11171	2	2	36.35	36.35
42	yes	191000	119047	310088	458341	148294	1	261	309	0	0	0	0
43	no	190420	119031	309492	457591	148140	5	11267	11309	2	2	35.01	35.01
44	yes	191014	119047	310102	458344	148283	1	270	304	0	0	0	0
45	yes	190994	119047	310082	458320	148279	1	264	304	0	0	0	0
46	no	190464	119031	309536	457634	148139	5	11490	11524	2	2	53.87	53.87
47	yes	191012	119047	310100	458362	148303	1	277	313	0	0	0	0
48	no	190448	119031	309520	457618	148139	5	13048	13081	2	2	57.97	57.97

Table 6.2: CLP Experiment 1. We list the first 10 and last 10 iterations.

Experiment 2 uses the same robot region as in Experiment 1, but obstacle region in Experiment 2 is more complicated than in Experiment 1. The obstacle in Experiment 2 is made up of 25 circular arc segments, and the same robot from Experiment 1 has 5 circular arc segments. The first and last iteration where a path is found is displayed in Figure 6.6 and Figure 6.7, respectively. For Experiment 2, the C-Space data is recorded in Table 6.3, and the CLP data is recorded in Table 6.4.

Because Experiment 2 has more input segments from the obstacle, there are more C-Space events in Table 6.3 than in Table 6.1. On average, there are about 66 stab events, 10 special events, 4 to 8 bump events and 1 triangle flip event. There are about 393 subvertices, 743 subedges, and 490 cells. The increase in size of the obstacle region from Experiment 1 to Experiment 2 has almost doubled the size of the subvertices, subedges, and cells. Table 6.4 shows that Experiment 2 has more coordinates and functions than Experiment 1 (Table 6.2), and as a consequence the running times are larger.

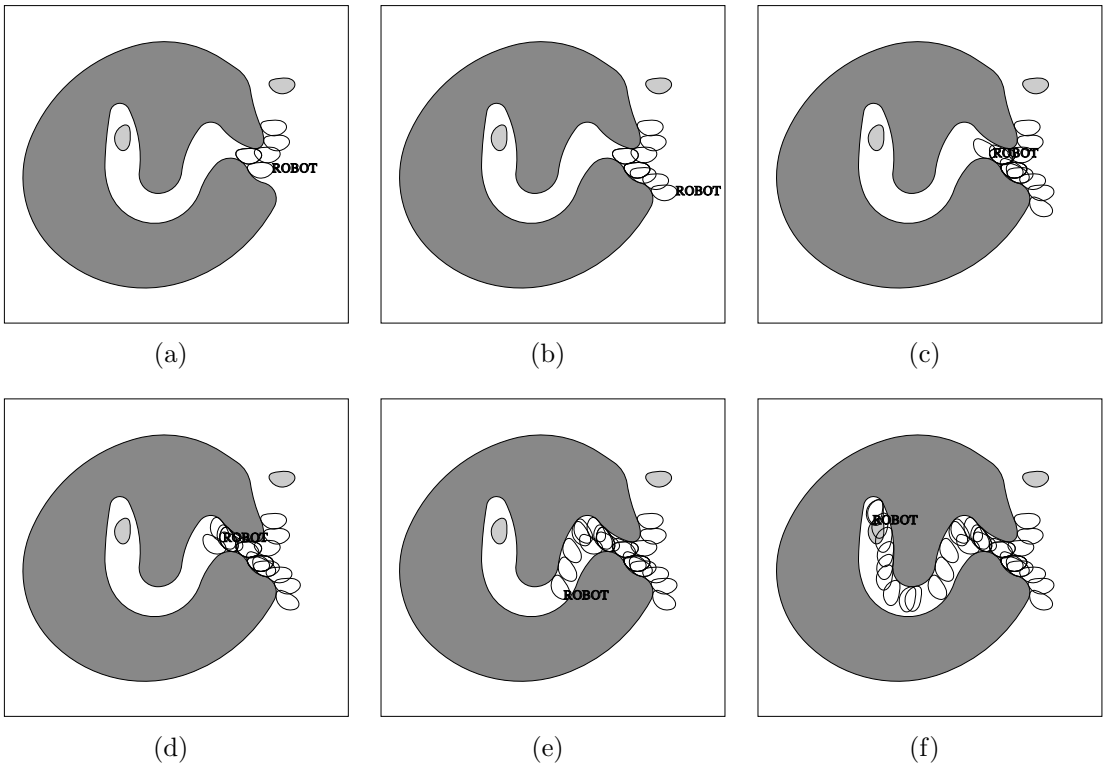


Figure 6.6: Experiment 2: First iteration with a path. First 5 figures are snapshots in increments of 5 plots. Last figure is the snapshot of the entire path. Last figure has 36 plots of the moving robot.

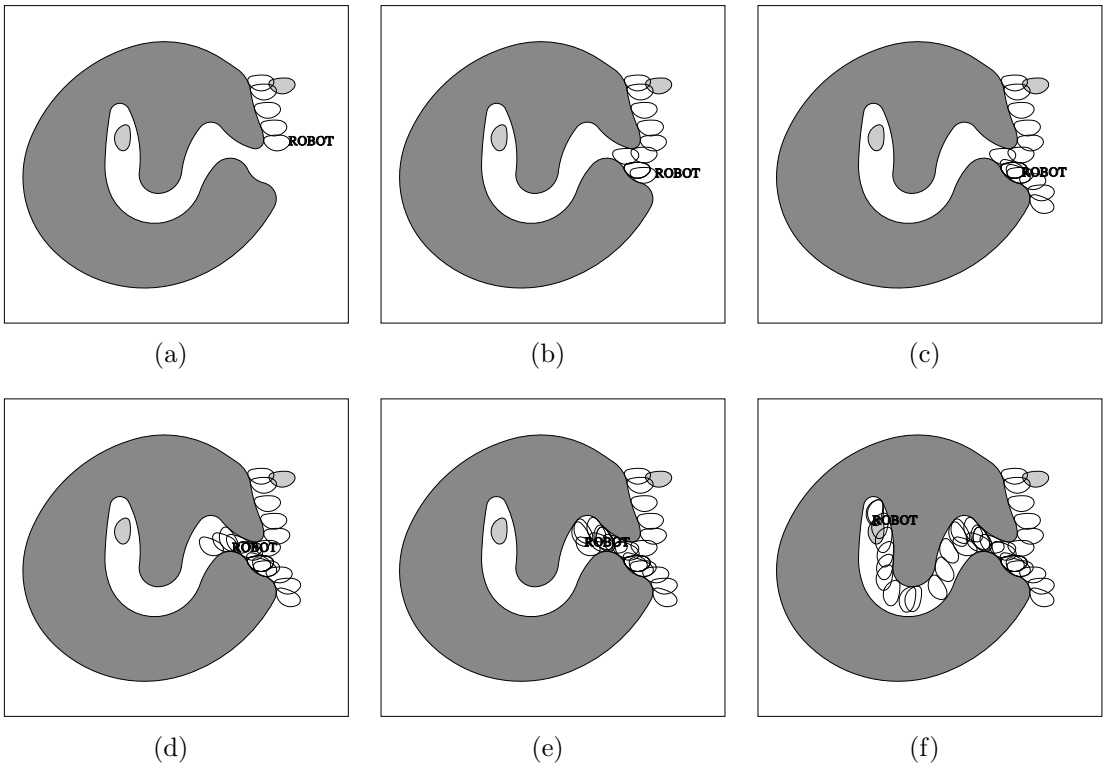


Figure 6.7: Experiment 2: Last iteration with a path. First 5 figures are snapshots in increments of 5 plots. Last figure is the snapshot of the entire path. Last figure has 38 plots of the moving robot.

i	Path	T1	T2s	T2S	T2b	T3	Total		SV	SE	C
0	yes	125	66	10	8	1	211		393	744	493
1	no	125	110	12	4	4	256		489	985	644
2	no	125	84	10	4	2	226		433	839	545
3	no	125	70	10	4	2	212		405	769	501
4	no	125	66	10	4	1	207		394	740	481
5	no	125	66	10	4	1	207		394	740	481
6	yes	125	66	10	8	1	211		393	743	493
7	no	125	66	10	4	1	207		394	740	481
8	no	125	66	10	4	1	207		394	740	481
9	no	125	66	10	4	1	207		393	739	481
					...						
43	no	125	66	10	4	1	207		393	739	481
44	no	125	66	10	4	1	207		393	739	481
45	yes	125	66	10	8	1	211		393	743	493
46	yes	125	66	10	8	1	211		393	743	493
47	no	125	66	10	4	1	207		393	739	481
48	no	125	66	10	4	1	207		393	739	481
49	yes	125	66	10	6	1	209		393	741	487
50	yes	125	66	10	8	1	211		393	743	493
51	yes	125	66	10	8	1	211		393	743	493
52	yes	125	66	10	8	1	211		393	743	493

Table 6.3: Configuration space information on Experiment 2: Number of events and graph elements in each iteration. We list the first 10 and last 10 iterations.

i	Path	EXP	IMP	TC	TP	SP	EP	CLP Time	Time	CLP	setSign	B
0	yes	537985	372581	910631	1348872	438306	3	744	865	0	0	1.00
1	no	583630	391855	975550	1444333	468848	3	893	1028	0	0	3.09
2	no	557048	381127	938240	1389954	451779	3	879	1017	0	0	4.41
3	no	546563	377091	923719	1369083	445429	3	873	1009	0	0	5.24
4	no	542216	375321	917602	1360030	442493	3	846	997	0	0	2.25
5	no	538847	373553	912465	1352175	439775	3	880	1011	0	0	1.84
6	yes	538767	373043	911875	1350929	439119	3	895	1020	0	0	2.37
7	no	538343	373305	911713	1350971	439323	3	876	1003	0	0	3.82
8	no	538227	373237	911529	1350735	439271	3	877	1001	0	0	33.01
9	no	537843	373041	910949	1349871	438987	3	897	1011	0	0	22.89
						...						
43	no	537805	373025	910895	1349773	438943	7	8312	8406	2	2	8.13
44	no	537779	373025	910869	1349753	438949	7	8279	8396	2	2	3.94
45	yes	538709	373051	911825	1350868	439108	3	860	975	0	0	13.01
46	yes	538689	373051	911805	1350858	439118	3	855	979	0	0	1.00
47	no	537801	373025	910891	1349806	438980	7	8316	8412	2	2	5.04
48	no	537751	373025	910841	1349725	438949	7	8582	8710	2	2	3.91
49	yes	538235	373043	911343	1350311	439033	5	4425	4553	1	1	5.73
50	yes	538759	373051	911875	1350922	439112	3	861	967	0	0	1.72
51	yes	538731	373051	911847	1350900	439118	3	871	979	0	0	2.86
52	yes	538731	373051	911847	1350890	439108	3	868	978	0	0	3.64

Table 6.4: CLP Experiment 2. We list the first 10 and last 10 iterations.

Experiment 3 uses the same obstacle as Experiment 2, but the moving region in Experiment 3 is more difficult than in Experiment 2. There are 25 circular arc segments from the obstacle and 5 circular arc segments from the moving region. The moving region is more difficult because it is longer in length than the moving region from Experiments 1 and 2. A longer region can require a more complex path, and this proves to be true in Experiment 3.

The first and last iteration where a path is found for Experiment 3 is displayed in Figure 6.8 and Figure 6.9, respectively. For Experiment 3, the C-Space data is recorded in Table 6.5, and the CLP data is recorded in Table 6.6.

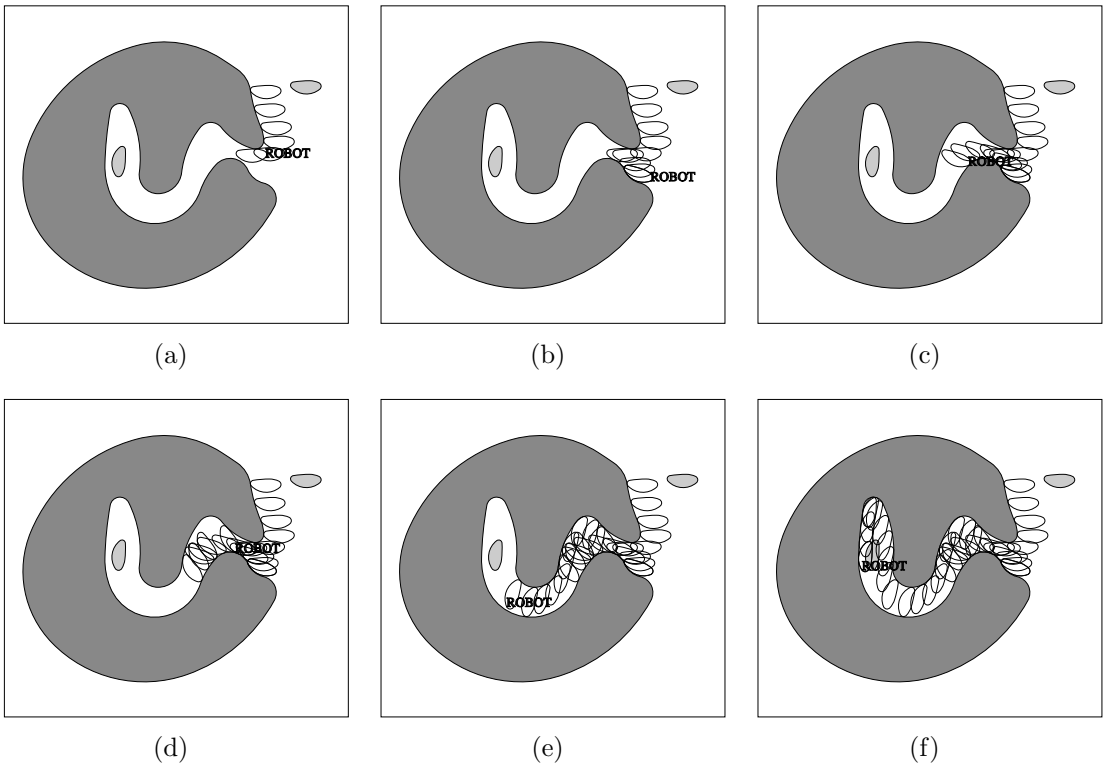


Figure 6.8: Path of first iteration for Experiment 3. First 5 figures are snapshots of the 5th, 10th, 15th, 20th, and 30th plot, and the last figure is the snapshot of the very first iteration. There are 36 total plots in the first iteration.

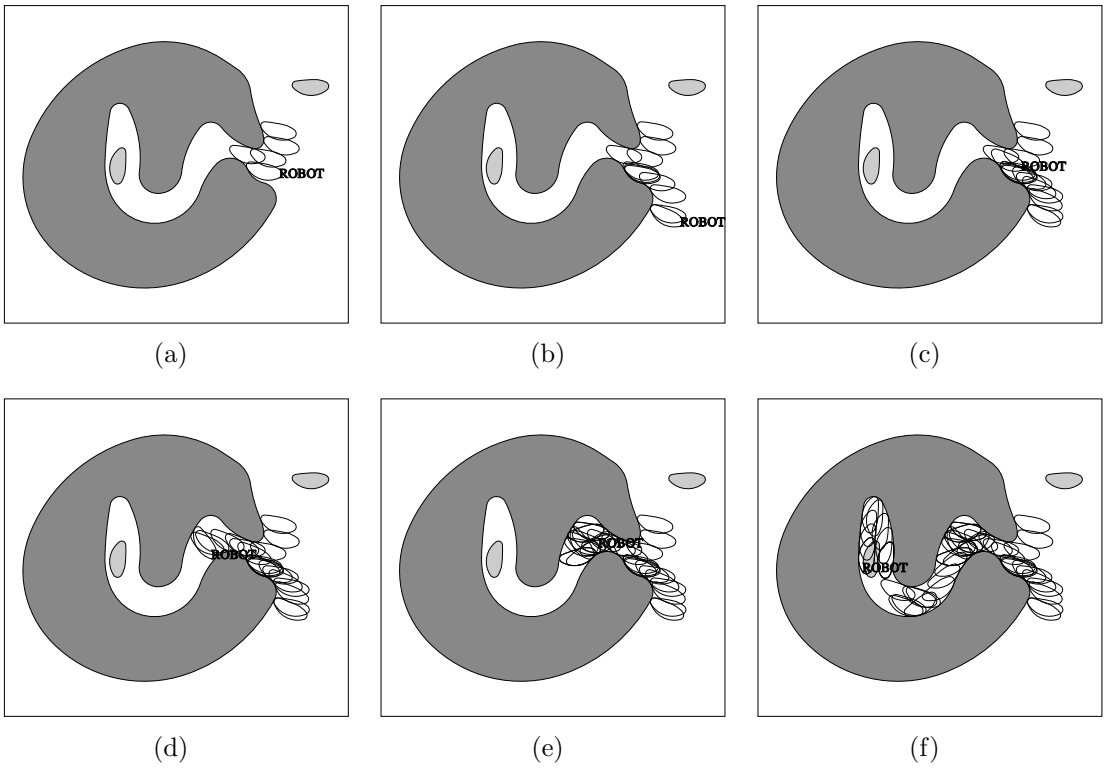


Figure 6.9: Path of last iteration for Experiment 3. First 5 figures are snapshots in increments of 5 plots, last figure is the snapshot of the very last iteration. There are 38 total plots in the last iteration.

i	Path	T1	T2s	T2S	T2b	T3	Total		SV	SE	C
0	yes	125	84	14	8	2	234		438	856	559
1	no	125	130	16	12	5	289		542	1128	738
2	yes	125	112	16	8	3	265		495	1004	658
3	yes	125	128	16	12	4	286		529	1093	725
4	no	125	128	16	12	6	288		539	1128	737
5	no	125	128	16	8	4	282		529	1091	713
6	yes	125	128	16	12	4	286		529	1095	725
7	no	125	128	16	8	4	282		529	1091	713
8	no	125	128	16	8	4	282		529	1091	713
9	no	125	128	16	8	4	282		529	1091	713
					...						
39	no	125	128	16	8	4	282		529	1090	713
40	yes	125	128	16	12	4	286		529	1094	725
41	no	125	128	16	8	4	282		529	1090	713
42	yes	125	128	16	12	4	286		529	1094	725
43	no	125	128	16	8	4	282		529	1090	713
44	no	125	128	16	8	4	282		529	1090	713
45	no	125	128	16	8	4	282		529	1090	713
46	no	125	128	16	8	4	282		529	1090	713
47	yes	125	128	16	12	4	286		529	1094	725
48	no	125	128	16	8	4	282		529	1090	713

Table 6.5: Configuration space information on Experiment 3: Number of events and graph elements in each iteration. We list the first 10 and last 10 iterations.

i	Path	EXP	IMP	TC	TP	SP	EP	CLP Time	Time	CLP	setSign	B
0	yes	630010	419329	1049404	1558652	509313	3	836	955	0	0	1.16
1	no	667210	431851	1099126	1630215	531154	3	968	1112	0	0	1.25
2	yes	647015	424833	1071913	1591248	519400	3	1016	1135	0	0	8.50
3	yes	659841	428545	1088451	1614308	525922	3	1042	1188	0	0	1.29
4	no	664355	430499	1094919	1623730	528876	3	1045	1204	0	0	2.56
5	no	661162	429509	1090736	1618037	527366	3	1050	1205	0	0	2.12
6	yes	660907	429007	1089979	1616541	526627	3	1056	1202	0	0	2.65
7	no	660605	429241	1089911	1616719	526873	3	1052	1197	0	0	1.16
8	no	660098	429051	1089214	1615703	526554	3	1038	1194	0	0	2.73
9	no	660066	429027	1089158	1615626	526533	3	1048	1194	0	0	7.74
						...						
39	no	659998	428971	1089034	1615393	526424	7	12317	12470	2	2	42.61
40	yes	661002	429007	1090074	1616611	526602	3	1083	1231	0	0	3.03
41	no	659960	428971	1088996	1615415	526484	7	13900	14045	2	2	9.79
42	yes	660970	429007	1090042	1616585	526608	3	1080	1239	0	0	14.31
43	no	659986	428971	1089022	1615436	526479	7	13570	13713	2	2	37.05
44	no	659938	428971	1088974	1615351	526442	7	15234	15394	2	2	8.86
45	no	659940	428971	1088976	1615339	526428	7	13219	13365	2	2	13.44
46	no	659944	428971	1088980	1615339	526424	7	13439	13560	2	2	7.00
47	yes	660960	429007	1090032	1616626	526659	3	1072	1229	0	0	1.10
48	no	659940	428971	1088976	1615402	526491	7	13695	13826	2	2	8.96

Table 6.6: CLP Experiment 3. We list the first 10 and last 10 iterations.

Chapter 7

Conclusion

We successfully computed the configuration space of a moving robot region with a fixed obstacle region, and showed how the configuration space aided in path planning. We were able to find if a path exists for a robot from a start to a goal all while avoiding collision with the obstacle. We verified the robustness in the CLP method by performing a binary search on the scale factor of the robot that terminated with two consecutive floating point numbers.

For future work, we would like to create more tests to fine tune the algorithm, such as allowing true dilating of the robot with a circle of radius r. This can provide a means of buffer for a real robot traveling through an obstacle, rather than an approximate buffer from scaling. As mentioned in Chapter 2, we would like to generalize the algorithm to line segments and smoothing arcs. This would generalize each type of event to handle line segments with curves segments or line segments with

line segments. For smoothing arcs, the algorithm would have to handle arcs with zero radius.

Currently there are only three degrees of freedom: $\langle x, y, \theta \rangle$. We would like to extend this algorithm more degrees of freedom. Extending to 6 degrees of freedom would allow translation along the z axis and allow rotation about the x, y axis for three dimensional objects.

Through the experiments we found that the CLP algorithm greatly aided in implementation. For example, in the configuration space algorithm the CLP discovered the Type 2 special events discussed in Section 2.6.4. Without these events, the explicit construction of the configuration space would be incorrect. As a side effect of the experiments, the configuration space algorithm aided in the improvement of the CLP algorithm. From our experimental data, we realize that the Linear Program is a real bottleneck for the algorithm, even if there are not many non-robust primitives. The reason for the large running times is the amount of implicit equality constraints that need to be satisfied in every LP call. Milenkovic and Sacks are already making improvements to the current CLP algorithm based on these findings, and in time the LP calls will be more efficient.

References

[1] W. Auzinger and H. Stetter. An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations. In International Series on Numerical Math, pages 11–30, 1988.

[2] W. Auzinger and H. Stetter. A study of numerical elimination for the solution of multivariate polynomial systems. Unpublished report, 1990.

[3] F. Avnaim and J.-D. Boissonnat. Practical exact motion planning of a class of robots with three degrees of freedom. In Proceedings of Canadian Conference of Computational Geometry, page 19, 1989.

[4] F. Avnaim, J.-D. Boissonnat, and B. Faverjon. A practical exact motion planning algorithm for polygonal object amidst polygonal obstacles. In Proceedings of the Workshop on Geometry and Robotics, pages 67–86, London, UK, 1989. Springer-Verlag.

[5] J. Bentley and T. Ottmann. Algorithms for reporting and counting geometric intersections. Computers, IEEE Transactions on, C-28(9):643–647, Sept. 1979.

[6] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, L. Kettner, K. Mehlhorn, J. Reichel, S. Schmitt, E. Schmer, and N. Wolpert. Exacus: Efficient and exact algorithms for curves and surfaces. In European Symposium on Algorithms, volume 3669 of LNCS, pages 155–166. Springer, 2005.

[7] E. Berberich, A. Eigenwillig, M. Hemmer, S. Hert, K. Mehlhorn, and E. Schömer. A computational basis for conic arcs and boolean operations on conic polygons. In European Symposium on Algorithms, pages 174–186. Springer-Verlag, 2002.

[8] CGAL: Computational Geometry Algorithms Library. `http://www.cgal.org`.

[9] G. Dudek and M. Jenkin. Computational Principles of Mobile Robotics. Cambridge University Press, New York, 2000.

[10] H. Edelsbrunner and E. P. Mucke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. ACM Trans. Graph, 9:66–104, 1990.

[11] A. Eigenwillig and M. Kerber. Exact and efficient 2d-arrangements of arbitrary algebraic curves. In Proceedings of the nineteenth annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 122–131, Philadelphia, USA, 2008. Society for Industrial and Applied Mathematics.

[12] A. Eigenwillig, L. Kettner, E. Schömer, and N. Wolpert. Complete, exact, and efficient computations with cubic curves. In Symposium on Computational Geometry, pages 409–418, 2004.

[13] I. Emiris and J. Canny. A general approach to removing degeneracies. In SFCS '91: Proceedings of the 32nd annual symposium on Foundations of computer science, pages 405–413, Washington, USA, 1991. IEEE Computer Society.

[14] I. Emiris and J. Canny. An efficient approach to removing geometric degeneracies. In SCG '92: Proceedings of the eighth annual symposium on Computational geometry, pages 74–82, New York, USA, 1992. ACM.

[15] I. Z. Emiris, J. F. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. Algorithmica, 19:219–242, 1997.

[16] EXACUS: Efficient and Exact Algorithms for Curves and Surfaces. `http://www.mpi-inf.mpg.de/projects/EXACUS/`.

[17] E. Flato, D. Halperin, I. Hanniel, O. Nechushtan, and E. Ezra. The design and implementation of planar maps in cgal. ACM Journal of Experimental Algorithms, 5:13, 2000.

[18] S. Funke, C. Klein, K. Mehlhorn, and S. Schmitt. Controlled perturbation for delaunay triangulations. In SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pages 1047–1056, Philadelphia, USA, 2005. Society for Industrial and Applied Mathematics.

[19] N. Geismann, M. Hemmer, and E. Schömer. The convex hull of ellipsoids. In Symposium on Computational Geometry, pages 321–322, 2001.

[20] L. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, pages 110–111, 1983.

[21] D. Halperin and E. Leiserowitz. Controlled perturbation for arrangements of circles. In Symposium on Computational Geometry, pages 264–273, 2003.

[22] D. Halperin and C. R. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. In Symposium on Computational Geometry, pages 183–192, 1997.

[23] C. Hoffmann. Robustness in geometric computations. Journal of Computing and Information Science in Engineering, 1:143–156, 2001.

[24] IEEE Standard for floating-point arithmetic. IEEE Std. 754-2008, pages 1–58, 2008.

[25] ILOG CPLEX. http://www.ilog.com/products/cplex/.

[26] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. Efficient and exact manipulation of algebraic points and curves. Computer-Aided Design, 32(11):649–662, 2000.

[27] J. Kuffner, J.J. and S. LaValle. Rrt-connect: An efficient approach to single-query path planning. IEEE International Conference on Robotics and Automation, 2:995–1001, 2000.

[28] S. M. Lavalle, M. S. Branicky, and S. R. Lindemann. On the relationship between classical grid search and probabilistic roadmaps. International Journal of Robotics Research, 2004.

[29] LEDA: Library of Efcient Data Types and Algorithms. http://www.algorithmic-solutions.com/leda/.

[30] T. Lozano-Perez. Spatial planning: A configuration space approach. Computers, IEEE Transactions on, C-32(2):108–120, 1983.

[31] K. Mehlhorn and S. Näher. LEDA: A Platform for Combinatorial and Geometric Computing, volume 38. 1999.

[32] K. Mehlhorn, R. Osbild, and M. Sagraloff. Reliable and efficient computational geometry via controlled perturbation. In Automata, Languages and Programming, 33rd Internat. Colloquium (ICALP06), Part I, volume 4051 of LNCS, pages 299–310. Springer, 2006.

[33] D. Michelucci. An ϵ-arithmetic for removing degeneracies. In IEEE Symposium On Computer Arithmetic, pages 230–. IEEE Computer Society, 1995.

[34] V. Milenkovic and E. Sacks. An approximate arrangement algorithm for semi-algebraic curves. In SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry, pages 237–246, New York, USA, 2006. ACM.

[35] V. Milenkovic and E. Sacks. A monotonic convolution for minkowski sums. International Journal of Computational Geometry and Applications, 17(4):383–396, 2007.

[36] B. Mourrain, J.-P. Técourt, and M. Teillaud. Sweeping an arrangement of quadrics in 3d. In Proceedings 19th European Workshop on Computational Geometry, pages 31–34, 2003.

[37] Y. L. Puzis. Robust arrangement of polynomials using linear programming. Master's thesis, Department of Computer Science, University of Miami, USA, August 2007.

[38] S. Raab. Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes. In Proceedings 15th ACM Symposium on Computational Geometry, pages 163–172, 1999.

[39] E. Sacks. Path planning for planar articulated robots using configuration spaces and compliant motion. Robotics and Automation, IEEE Transactions on, 19(3):381–390, 2003.

[40] E. Sacks and R. Bajaj. Sliced configuration spaces for curved planar bodies. International Journal of Robotics Research, 17:639–651, 1998.

[41] R. Seidel. The nature and meaning of perturbations in geometric computing. In STACS '94: Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science, pages 3–17, London, UK, 1994. Springer-Verlag.

[42] H. Stetter. Multivariate polynomial equations as matrix eigenproblems. In World Scientific Series in Applicable Analysis, pages 355–371, 1993.

[43] S. C. Trac. Robust topologically invariant set operations on 2d semi-algebraic sets. Master's thesis, Department of Computer Science, University of Miami, USA, December 2005.

[44] G. Varadhan, Y. Kim, S. Krishnan, and D. Manocha. Topology preserving approximation of free configuration space. Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on, pages 3041–3048, 2006.

[45] L. E. K. P. Vestka, J. claude Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Transactions on Robotics and Automation, 12:566–580, 1996.

[46] R. Wein. High-level filtering for arrangements of conic arcs. In Proceedings European Symposium on Algorithms 2002, pages 884–895. Springer-Verlag, 2002.

[47] R. Wein. Exact and efficient construction of planar minkowski sums using the convolution method. In Proceedings of the 14th conference on Annual European Symposium on Algorithms, pages 829–840, London, UK, 2006. Springer-Verlag.

[48] N. Wolpert. Jacobi curves: Computing the exact topology of arrangements of non-singular algebraic curves. In European Symposium on Algorithms, pages 532–543, 2003.

[49] C. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, Handbook of discrete and computational geometry. CRC Press, second edition, 2004.

[50] C. Yap and T. Dubé. The exact computation paradigm. In D. Du and F. Hwang, editors, Computing in Euclidean Geometry, pages 452–492. World Scientific Press, second edition, 1995.

[51] C. K. Yap. A geometric consistency theorem for a symbolic perturbation scheme. In SCG '88: Proceedings of the fourth annual symposium on Computational geometry, pages 134–142, New York, USA, 1988. ACM.

[52] C.-K. Yap. Symbolic treatment of geometric degeneracies. J. Symb. Comput., 10(3-4):349–370, 1990.