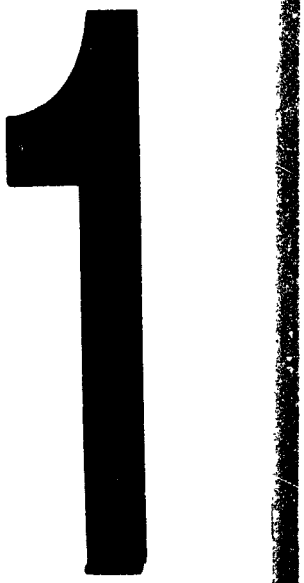


GENERATING GEOMETRIC OBJECTS AT RANDOM

by

Peter Epstein

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Computer Science

Carleton University
School of Computer Science

ISBN 0-315-79783-5

The undersigned recommended to the Faculty of
Graduate Studies and Research acceptance of the thesis

GENERATING GEOMETRIC OBJECTS AT RANDOM

submitted by Peter Epstein, in partial fulfillment of the
requirements for the degree of Master of Computer Science

J.-R. Sack

Thesis Supervisor

Director, School of Computer Science

Carleton University

27 April 1992

ABSTRACT

This thesis deals with the problem of creating a random stream of various types of geometric objects. This research falls in an area known as geometrical probability, and involves both computational geometry and probability theory. We define useful probability distributions for various types of geometric objects, and construct generators satisfying these distributions. Types of geometric objects considered include intervals and simple interval sets; points and point sets; line segment sets; star-shaped, monotone, convex, and arbitrary simple polygons; and triangulations of point sets, convex polygons, and arbitrary simple polygons. We develop the notions of uniformity and of uniform coverage probability. We construct efficient algorithms to generate intervals and polygons with uniform coverage probability, and to generate uniform random triangulations of a simple polygon.

ACKNOWLEDGMENTS

I would like to thank Luc Devroye for his insights into algorithms for intervals and arcs. As well, my supervisor, Jörg-Rüdiger Sack was extremely helpful, both in keeping me working in the right direction and in cooperatively developing new ideas. It was a pleasure working with the two of you.

I would also like to thank my brother Danny Epstein and my friend George Jost for many useful conversations. As well, Jorge Urrutia asked many interesting questions at my seminar, and I would like to thank him for the insights they led to, as well as for his help concerning circle graphs. A particularly helpful conversation with Bruce Richter was also much appreciated.

I would like to thank Lewis Epstein for buying me my first computer, effectively getting me started in computer science.

My final thanks go to those who keep me happy: my parents Bob and Sanni, and especially, Anne-Lise Hassenklover.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

1. INTRODUCTION

This work deals with the problem of generating various types of geometric objects at random. We will precisely define the properties we expect such a stream of geometric objects to exhibit. Uses for geometric objects generated at random include testing of computational geometry algorithms and verification of time complexity.

1.1. RANDOMNESS

In this section we consider the question of what it means for a geometric object to be generated at random. Consider the random generation of an interval on the number line between 0 and 1. Would we expect such a randomly generated interval to cover the point $\frac{1}{10}$ with the same probability as it covers the point $\frac{1}{2}$? The answer may depend upon the intended use of the interval generated. However, in either case, the answer implies a probability distribution that should be used to generate the interval. As [Kendall & Moran 63] point out, failure to recognize that there are many useful probability distributions for any one type of geometric object can lead to paradoxes. We consider several important probability distributions and we develop algorithms to generate random geometric objects with these distributions. For example, if we do expect a randomly generated interval to cover the point $\frac{1}{10}$ with the same probability as it covers the point $\frac{1}{2}$, then we use a uniform coverage probability, as we will define in Section 3.3.4 (page 36). We will see that constructing a generator that provides a specific probability distribution is non-trivial. Surprisingly, it is not possible to construct an algorithm that can produce any possible interval of a given fixed length in a given domain while providing a uniform coverage probability (see Section 7.4.6, page 79).

1.2. MOTIVATION

In this section, we provide some of the motivation for our study on generators of geometric objects at random.

1.2.1. ALGORITHM TESTING

The most direct use for a stream of geometric objects generated at random is for testing computational geometry algorithms. Suppose, for example, that you have implemented a triangulation algorithm for simple polygons. This algorithm can be tested in two basic ways. The first involves the construction of polygons that the implementer considers difficult cases for the algorithm. For example, a triangulation algorithm based on a plane sweep may find horizontal or vertical edges require special case code, making polygons with such edges likely candidates for exposing errors. The second approach to testing an algorithm is to randomly generate a large number of inputs. We expect errors to be exposed if enough different valid inputs are applied to the algorithm. For this purpose, generation of geometric objects at random is useful. Although a uniform distribution is not essential, it is important that any one of the valid inputs may be generated. If there exists a valid input that cannot be generated, then this input may be the only one for which the algorithm fails. A uniform probability distribution is however desirable, since we would like a bound on the expected number of objects generated before any given object is generated. Without such a bound, there is no expected number of objects required to fully test an algorithm, since the only object for which the algorithm fails may be generated with arbitrarily low probability.

Examples of algorithms that would benefit from a generator of geometric objects at random include:

- Algorithms to triangulate simple polygons
- Algorithms to sort Jordan sequences. A random Jordan sequence can be constructed by intersecting a random simple polygon with a line.

- Algorithms to find shortest paths or shortest path trees in a triangulated polygon.
- Algorithms to compute visibility from a point or line segment in a triangulated simple polygon.

Many researchers in computational geometry construct streams of geometric objects generated at random to test their algorithm implementations. Fairly crude algorithms are often used to generate these random objects since their distributions are not important. However, if these algorithms are not carefully constructed, it is possible for the resulting geometric objects to share some properties that the algorithm being tested can take advantage of, leading to inadequate testing. For example, Jordan sequences generated by intersecting polygons with lines will yield very simple sequences if the polygons do not tend to spiral. Since many algorithms for generating random polygons tend to produce polygons with few spirals, these generators will not yield interesting Jordan sequences.

1.2.2. VERIFICATION OF TIME COMPLEXITY

When given an algorithm that is purported to execute in some time complexity such as O(n), how can we verify that this complexity is achieved? One approach is to time the execution of the algorithm for various inputs of different sizes. Although the absolute times are of little value, the relationship between the times will typically follow a curve corresponding to its complexity. For example, if the algorithm is indeed executing in linear time, then the time should be proportional to the input size. What we have overlooked in this discussion is the variety of possible inputs of any given size. The choice is important, since some inputs may take more time than others of the same size.

For example, consider a linear time visibility algorithm that accepts a simple polygon as input. While some polygons have many spirals, making the algorithm perform complex steps, other polygons of the same size are convex, making the algorithm trivial. If we are to get a meaningful graph of time versus polygon size, it is important that the algorithm is tested on many different

random polygons of any given size. In this case, the probability distribution of the polygons is important.

1.2.3. THEORETICAL INTEREST

As in any science, one of the significant motivating factors is simply the advancement of our knowledge. We are interested in developing efficient algorithms for specialized problems even when they do not directly lead to practical applications. It is this interest that allows many of the more difficult practical algorithms to eventually become possible.

1.3. RELATED WORK

This section relates this thesis to existing work in the area. This is by no means a complete list of related work. Much work has been done on very specialized problems in the area. However, little general work has been done on generating geometric objects at random.

1.3.1. COMBINATORIAL OBJECTS

Generation of random combinatorial objects, such as trees and graphs, involves both probability and graph theory, and has been studied in, for example, [Atkinson & Sack 92], [Bollobás 85], and [Reingold et al. 77]. Combinatorial analysis concerns the counting of various non-geometric objects, but does not deal with the *efficient* generation of random non-geometric objects. Of course, some geometric objects are completely combinatorial in nature. For example, there are a finite number of triangulations of a given polygon. In these cases, there are mappings between the geometric objects and some other combinatorial objects. We can take advantage of this mapping to construct efficient generators.

1.3.2. GEOMETRIC OBJECTS

There is little existing work on generation of geometric objects at random. The most significant work is [Kendall & Moran 63], which defines probability distributions for points, lines, planes, and rotations in $\Re^2$ and $\Re^3$. Much of the existing work deals with unbounded objects such as lines rather than bounded objects such as line segments. As well, the domain is typically taken to

be unbounded. The authors motivate their work by giving an example of the paradoxes that can be caused when probability distributions are not explicitly stated. As well, [Kendall & Moran 63] motivate probability distributions that are invariant under Euclidean transformations, as discussed in Section 3.3.2 (page 32), and devise a measure for lines in the plane that is invariant under these transformations, as discussed in Section 6.1.2 (page 48).

5.2.1. POLYGONS

Several papers have been written on the generation of random simple polygons. Most of these algorithms do not meet our requirements, since they cannot generate all simple polygons. For example, [Culberson & Rawlins 85] generate polygons having turn angles that are all multiples of some fundamental angle $\theta = \frac{\pi}{k}$ for some integer $k \geq 2$.

See [O'Rourke & Virmani 91] for an interesting approach to generating random simple polygons. The initial polygon is a regular convex polygon. This polygon is then repeatedly modified as follows. Each vertex is given a random velocity vector. The vertices act as moving particles, and they bounce whenever continuing in their current direction would make the polygon non-simple. After several thousand steps of simulated motion, the vertices tend to be uniformly distributed. Although this technique does not guarantee a uniform distribution, it is capable of producing any simple polygon. This paper also gives several tests to verify the uniformity of a random simple polygon generator.

Generating random simple polygons with other properties, such as convexity, monotonicity, or star-shapedness, has been studied in [Culberson & Rawlins 85], [Devroye 82], [Devroye 89], [Doe & Edwards 84], [Edelsbrunner 87], [O'Rourke et al. 87], and [Sack 84]. Also, [May & Smith 82] consider generation of random unbounded convex polygons in n dimensions.

1.4. SCOPE OF THESIS

We will consider the generation of many types of geometric objects. We consider generation of objects in a bounded portion of the number line or the plane ($\Re^1$ and $\Re^2$).

We will consider generation of combinatorial structures that take a geometric object as input. For example, there are a finite number of triangulations of a given polygon or point set, but generation of a random triangulation is not an entirely combinatorial problem because we can take advantage of geometric properties of the polygon or point set. We therefore consider generation of triangulations in this thesis.

There are many types of geometric objects so we do not attempt to be complete in our coverage. The purpose of this thesis is to break some new ground, providing algorithms for generating some important types of geometric objects that are of theoretical or practical interest.

1.5. NOTATION

This section defines some of the notation we will use in this thesis.

1.5.1. GENERAL

For ease of notation, we say an object generated at random is a **random object**. This does not imply that an object can be said to be random without considering the stream of objects that it is a part of.

Standard set notation is used. The logical **and** and **or** operation is written explicitly for clarity. The logical **not** operation is written as $\bar{a}$. Set subtraction is written as a - b, just as arithmetic subtraction is written. Ambiguities can be resolved from context.

The real numbers are written as $\Re$, the real plane is written as $\Re^2$, and so on. Similarly, the rational numbers are written as Q. Vectors are written as $\bar{v}$ to distinguish them from points. The coordinate system we use in $\Re^2$ is the standard. The x coordinate points right, the y coordinate points up, and angles are measured counter-clockwise from the positive x axis.

Greatest lower bound is written as **inf**{f(N) | <condition on N>}, being the greatest lower bound of f(N) for all N satisfying the given condition. Similarly, least upper bound is written as **sup**{f(N) | <condition on N>}, being the least upper bound of f(N) for all N satisfying the given condition.

1.5.2. PROBABILITY THEORY

The **sample space** is written as Ω. Probability functions are written $Pr(x)$ or $Pr_u(x)$ where u indicates the probability distribution (u standing for uniform, c standing for uniform coverage probability, and so on).

1.5.3. GEOMETRIC OBJECTS

Points in the $\Re^2$ plane are written as (x,y). An **interval** represents a contiguous portion of the number line bounded by a start and an end. Intervals are defined in a domain D which is itself an interval, that may be open, closed, or semi-open.

Definition: The **closed interval** [a,b] in the domain D, where $a,b \in D$, $a < b$, is defined as $[a,b] = \{x \mid x \geq a \text{ and } x \leq b\}$.

Definition: The **open interval** (a,b) in the domain D, where $(a,b) \subseteq D$, $a < b$, is defined as $(a,b) = \{x \mid x > a \text{ and } x < b\}$.

Definition: The **semi-open interval** [a,b) in the domain D, where $[a,b) \subseteq D$, $a < b$, is defined as $[a,b) = \{x \mid x \geq a \text{ and } x < b\}$.

The **semi-open interval** (a,b] in the domain D, where $(a,b] \subseteq D$, $a < b$, is defined as $(a,b] = \{x \mid x > a \text{ and } x \leq b\}$.

Note that the endpoints defining an interval need not be members of the domain. For example, [10,20) is an interval in the domain [0,20).

Definition: The **length** of an interval [a,b), [a,b), (a,b], or (a,b) in the domain D, where $a,b \in D$, $a < b$ is defined as

$$\text{Length}([a,b]) = \text{Length}([a,b)) = \text{Length}((a,b]) = \text{Length}((a,b)) = b - a$$

A **circular arc** or just **arc** represents a contiguous portion of a circular domain. The start and end of the domain are logically equivalent, being an arbitrary break in an otherwise continuous domain. Arcs are written in the same way as intervals, and arcs may be open, closed, or semi-open, just as

intervals are. The domain for arcs must be a semi-open interval. For simplicity, we assume the domain has the form $D = [D_1,D_2)$, although it could equally well be of the form $(D_1,D_2]$. If $a > b$, the arc *wraps around* the ends of the domain:

Definition: The **semi-open arc** $[a,b)$ in the semi-open domain $D = [D_1,D_2)$, where $[a,b) \subseteq D$, $a \neq b$, $D_1 < D_2$ is defined as

$$[a,b) = \begin{cases} \{x | x \geq a \text{ and } x < b\} & \text{if } a < b \\ \{x | (x \geq D_1 \text{ and } x < b) \text{ or } (x \geq a \text{ and } x < D_2)\} & \text{if } a > b \end{cases}$$

Definition: The **length** of an arc $[a,b)$ in the semi-open domain $D = [D_1,D_2)$, where $a,b \in D$, $a \neq b$, $D_1 < D_2$ is defined as

$$\text{Length}([a,b)) = \begin{cases} b - a & \text{if } a < b \\ (D_2 - D_1) - (a - b) & \text{if } a > b \end{cases}$$

The ambiguity between open intervals and points should not lead to confusion, since context will imply whether the domain is $\Re^1$ or $\Re^2$.

Rectangles and line segments are said to be **axis parallel** while polygons are said to be **rectilinear**. Both terms mean that each line segment is parallel to one of the axes (i.e., either the x or y axis in $\Re^2$). Axis parallel rectangles are written out formally as $R = \{(x,y) \mid x \in [R_L,R_R] \text{ and } y \in [R_B,R_T]\}$, where R_L is the x coordinate of the left hand side, R_R is the x coordinate of the right hand side, R_B is the y coordinate of the bottom (i.e., minimum y coordinate), and R_T is the y coordinate of the top. As for intervals, any bounded region in $\Re^2$ is said to be **closed** if its boundary is included while it is said to be **open** if its boundary is excluded. For example, $R = \{(x,y) \mid x \in [R_L,R_R] \text{ and } y \in [R_B,R_T]\}$ is a closed region while $R = \{(x,y) \mid x \in (R_L,R_R) \text{ and } y \in (R_B,R_T)\}$ is an open region. If part of the boundary is included, the region is said to be **semi-open**.

A point set is said to be in **standard form** if all points are distinct. A point set is said to be in **general position** if no three points are collinear and no four points are co-circular.

An **n-gon** is an n vertex polygon. A polygon is said to be in **standard form** if all vertices are distinct and no three consecutive vertices are collinear. A simple polygon P is defined as

$P = (v_1, v_2, \ldots, v_n)$ where $v_1, v_2, \ldots, v_n$ are the n vertices of the polygon listed in clockwise order. The polygon Q produced by removing a single vertex v_i from a polygon P is written as $Q = P - v_i$. The formal definition is $Q = (v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$ if $i > 1$ and $Q = (v_2, \ldots, v_n)$ if $i = 1$, where $n \geq 4$. We typically use modular arithmetic when dealing with vertex indices, so $v_{i-1} = v_n$ if $i = 1$ and $v_{i+1} = v_1$ if $i = n$. An edge of a polygon between vertices v_i and v_j is written as (v_i, v_j). A **θ-monotone polygon** is a polygon which is composed of two θ-monotone chains. A **θ-monotone chain** is a sequence of vertices connected in order along the line in direction θ.

1.6. THESIS OVERVIEW

We start our investigation in Chapter 2 by reviewing the mathematical foundations in measure, probability, and graph theory. Our definition of randomness for geometric objects is based on this theory. Chapter 3 defines the notion of a random geometric object generator, listing the desired properties. We will generate geometric objects at random satisfying these properties. The capabilities we require in order to construct these generators are listed in Chapter 4. In Chapter 5 we describe some general techniques that are useful in generating many types of geometric objects. Chapters 6 through 10 deal with the generation of various types of geometric objects at random. We draw our conclusions in Chapter 11.

Chapter 2

2. FOUNDATIONS

In this chapter we review some of the mathematical foundations of measure theory, probability theory, and graph theory, as relevant to this thesis. The precise meaning of the formal definitions is often important, so we review them carefully here.

2.1. MEASURE THEORY

The following is a review of basic measure theory based on [Reinhardt & Soeder 74] and [Halmos 74]. The fundamental problem that we wish to solve is the problem of defining the area function in $\Re^2$ and its generalizations in $\Re^n$ (for example, volume in $\Re^3$). We will start with the most straightforward definition and progress to more complex but more general definitions.

2.1.1. DEFINITION OF MEASURE

A measure is defined as follows:

Definition: A **measure** is a function $I: M \rightarrow \Re$ satisfying the following requirements:

Property 1: $(M,\cup,-)$ is a **ring** of sets. In other words, M is closed under a finite number of unions and set subtractions. This implies closure under a finite number of intersections.

Property 2: $I(E) \geq 0$ for all $E \in M$. In other words, I is **non-negative**.

Property 3: $I(\varnothing) = 0$.

Property 4: M is **countably additive**. In other words, if $E_i \in M$ for all i, and $\bigcup_{i=1}^{\infty} E_i \in M$, and $E_i \cap E_j = \varnothing$ for all $i \neq j$, then $I\left(\bigcup_{i=1}^{\infty} E_i\right) = \sum_{i=1}^{\infty} I(E_i)$.

2.1.2. PROPERTIES OF MEASURES

Definition: A measure $I: M \rightarrow \Re$ is **monotone** if, for all $A,B \in M$ satisfying $A \subseteq B$, $I(A) \leq I(B)$.

Theorem 2: Any measure I: $M \rightarrow \Re$ is monotone.

Proof:

If $A \subseteq B$, then let $C = B - A$. $C \in M$ by Property 1. We know that $C \cap A = \varnothing$ and that $A \cup C = B$. By Property 4, we therefore know that $I(A \cup C) = I(A) + I(C)$. Therefore, $I(B) = I(A) + I(C)$. Since $I(C) \geq 0$ by Axiom 0, we have the desired result $I(A) \leq I(B)$.□

2.1.3. ELEMENTARY-MEASURE

Our first definition of area will be called the **elementary-measure**. The measure is defined to be a function I_E: $M_E \rightarrow \Re$ where M_E is a set of subsets of $\Re^2$. The following additional requirements must be satisfied:

Property 5: The measure is **invariant under rotation and translation**. In other words, if $A,B \in M_E$ and A can be rotated and/or translated to get B, then $I_E(A) = I_E(B)$.

Property 6: The **unit square** has unit measure. In other words, if S is the unit square, then $I_E(S) = 1$.

Note that Property 5 only requires B to have the same measure as A if B is measurable. Some rotations and/or translations of a measurable set A may not be measurable.

To compute the elementary-measure of a set $A \in M_E$, we simply partition the set into a finite number of pairwise disjoint axis parallel rectangles, and sum the measures of the rectangles. The problem with this measure is that many simple geometric objects, such as triangles and circles, are not elementary-measurable. Therefore, this measure has been generalized as follows.

2.1.4. JORDAN-MEASURE

The second definition of area is **Jordan-measure** (named for Camille Jordan, 1838-1922). This measure is defined to be a function I_J: $M_J \rightarrow \Re$ where M_J is a set of subsets of $\Re^2$.

DEFINITION OF THE JORDAN-MEASURE FUNCTION

We define the function I_J in three steps. We first define the function on axis parallel rectangles. We then use this definition to compute the value of the function for finite unions of axis parallel rectangles. Finally, we use this definition to compute the value of the function for arbitrary bounded regions of $\Re^2$.

Case 1: The Jordan-measure of an $a \times b$ axis parallel rectangle R is defined to be $I_J(R) = ab$, regardless of whether R is an open, semi-open, or closed rectangle.

Case 2: The Jordan-measure for a union of a finite number of rectangles $N = \bigcup_{i=1}^{n} R_i$ is defined as follows. Construct a partitioning of N into m rectangles $R'_1,\ldots,R'_m$. We define the Jordan-measure as $I_J(N) = \sum_{i=1}^{m} I_J(R'_i)$. All such partitionings produce the same sum.

Case 3: The Jordan-measure for an arbitrary bounded region B in $\Re^2$ is defined as follows (see Figure 2.1). Let $m^*(B) = \inf\{I_J(N) \mid N \supseteq B\}$ be the **outer Jordan-measure**, and let $m_*(B) = \sup\{I_J(N) \mid N \subseteq B\}$ be the **inner Jordan-measure**. In other words, $m^*(B)$ is the greatest lower bound of the Jordan-measures of all finite unions of rectangles that contain B (outer approximations), and $m_*(B)$ is the least upper bound of the Jordan-measures of all finite unions of rectangles that are completely contained in B (inner approximations). A set B is defined to be Jordan-measurable if $m^*(B) = m_*(B)$. The Jordan-measure of such a set B is then defined to be $M_J(B) = m^*(B)$.

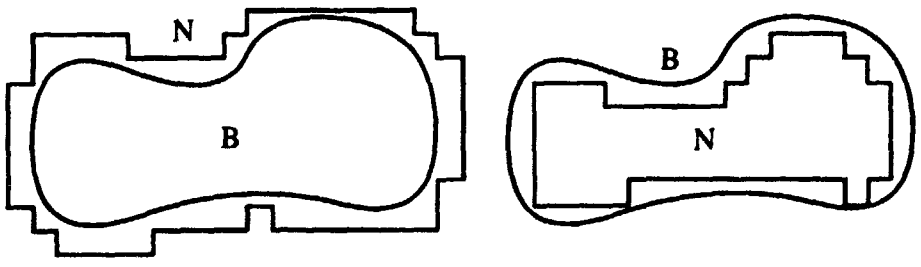


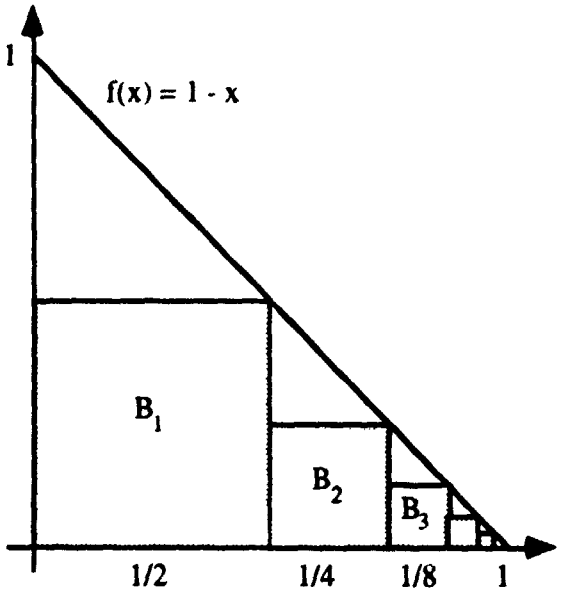
Figure 2.1. A bounded region B and its outer and inner approximations.

Points and axis parallel line segments are degenerate axis parallel rectangles, so their Jordan-measure is zero, regardless of whether none, one, or both of the endpoints are included.

PROPERTIES OF THE JORDAN-MEASURE

Properties 1 to 6 are satisfied by the Jordan-measure, so the Jordan-measure is an Elementary-measure. Many real-life problems can be done using Jordan-measure. For example, rectangles that are not axis parallel, triangles, simple n-vertex polygons, and circles are all Jordan-measurable.

An example of a set that is Jordan-measurable but is not elementary-measurable is shown in Figure 2.2. There are a countable but infinite number of squares, so B is defined as a countable union. Even though we do not explicitly support countable unions, the least upper bound and greatest lower bound operations allow Jordan-measure to work, since the upper and lower approximations can be refined, and their bounds are the same.



B is defined as $B = \bigcup_{i=1}^{\infty} B_i$, where $I_J(B_i) = \left(\frac{1}{4}\right)^i$, so $I_J(B) = \sum_{i=1}^{\infty} I_J(B_i) = \frac{1}{3}$.

Figure 2.2. A Jordan-measurable region.

However, not all bounded subsets of $\Re^2$ are Jordan-measurable. For example, the set $B = \{(x,y) \mid (x,y) \in A \text{ and } x,y \in Q\}$ of all points with rational coordinates in a rectangle A is not Jordan-measurable. To see why this is the case, consider the inner and outer Jordan-measures. The best lower bound we can get is $m_*(B) = 0$ while the best upper bound is $m^*(B) = I_J(A) \neq 0$. So $m_*(B) \neq m^*(B)$, implying that B is not Jordan-measurable.

Definition: A **Jordan-null set** is a subset with Jordan-measure = 0.

Examples of Jordan-null sets are the empty set, a point, a finite set of points, and line segments. Null sets are closed under finite unions, intersections, and subtractions.

Jordan-measurability has been characterized as:

Fact: A bounded subset B of $\Re^2$ is Jordan-measurable if and only if the boundary of B is a Jordan-null set.

It can also be shown that, for a closed bounded set A (includes boundary) with boundary B, any set Z satisfying $A - B \subseteq Z \subseteq A$ implies $I_J(A - B) = I_J(Z) = I_J(A)$. We have seen that not all sets are Jordan-measurable. We therefore further generalize the measure as follows:

2.1.5. LEBESGUE-MEASURE

The third definition of area is called the **Lebesgue-measure** (named for Henri Lebsegue, co-founder of modern probability and measure theory). The measure is defined to be a function $I_L: M_L \rightarrow \Re$ where M_L is a set of subsets of $\Re^2$. Lebesgue-measure is a generalization of Jordan-measure in which possibly infinite countable unions of rectangles are allowed rather than the finite unions of Jordan-measure. For any bounded region B in $\Re^2$, the exterior and interior measure functions are $\mu^*(B)$ and $\mu_*(B)$ respectively, defined as follows.

DEFINITION OF THE LEBESGUE-MEASURE FUNCTION

We define the function I_L in three steps. We first define the function on finite unions of axis parallel rectangles. We then use this definition to compute the value of the function for countably

infinite unions of axis parallel rectangles. Finally, we use this definition to compute the value of the function for arbitrary bounded regions of $\Re^2$.

Case 1: A finite union N of axis parallel rectangles has Lebesgue-measure $I_L(N) = I_J(N)$.

Case 2: The Lebesgue-measure for a bounded region N consisting of a union of a countable number of rectangles $N = \bigcup_{i=1}^{\infty} R_i$ is defined as follows. Construct a partitioning of N into a countable number of rectangles $R'_1,\ldots,R'_i,\ldots$. The Lebesgue-measure of N is defined as $I_L(N) = \sum_{i=1}^{\infty} I_L(R'_i)$. The series always converges. Furthermore, all such partitionings produce the same result.

Case 3: The Lebesgue-measure for an arbitrary bounded region B contained in a rectangle A in $\Re^2$ is defined as follows. Let $\mu^*(B) = \inf\{I_L(N) \mid N \supseteq B\}$ be the **outer Lebesgue-measure**. Similarly, let $\mu_*(B) = \sup\{I_L(N) \mid N \subseteq B\}$ be the **inner Lebesgue-measure**. In other words, $\mu^*(B)$ is the greatest lower bound of the Lebesgue-measures of all outer approximations, and $\mu_*(B)$ is the least upper bound of the Lebesgue-measures of all inner approximations. A set B is defined to be Lebesgue-measurable if $\mu^*(B) = \mu_*(B)$. The Lebesgue-measure of a Lebesgue-measurable set B is then defined to be $I_L(B) = \mu^*(B)$.

An alternative definition for the inner Lebesgue-measure is $\mu_*(B) = I_L(A) - \mu^*(A - B)$, the area of the rectangle less the lower bound of the Lebesgue-measures of all countable unions of rectangles that contain A - B (see [Halmos 74, page 61]).

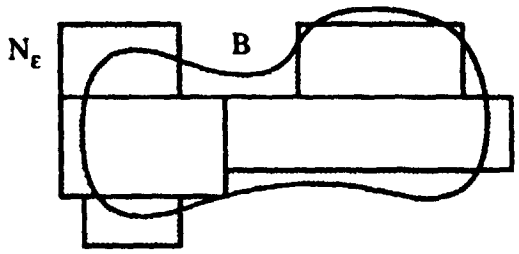
PROPERTIES OF THE LEBESGUE-MEASURE

Properties 1 to 6 are satisfied by the Lebesgue-measure, thus the Lebesgue-measure is also an Elementary-measure. Jordan-measurability implies Lebesgue-measurability, but not all Lebesgue-measurable sets are Jordan-measurable [Reinhardt & Soeder 74].

Lebesgue-measurability has been characterized as:

Fact: A bounded set B of $\Re^2$ is Lebesgue-measurable if and only if, for all real numbers $\varepsilon > 0$, there is a finite set of pairwise disjoint axis parallel rectangles N_ε satisfying $\mu^*((B - N_\varepsilon) \cup (N_\varepsilon - B)) < \varepsilon$.

In other words, a set is Lebesgue-measurable if and only if the set can be approximated arbitrarily well by a finite set of rectangles (see Figure 2.3).



$\square$ $(B - N_\varepsilon) \cup (N_\varepsilon - B)$

The gray areas are the error, which can be arbitrarily small

Figure 2.3. A bounded region and an approximation by a finite set of rectangles.

Definition: A **Lebesgue-null set** is a subset with Lebesgue-measure = 0.

Every countable subset of $\Re^2$ is a Lebesgue-null set, but there are also uncountable Lebesgue-null sets. All bounded open and bounded closed subsets of $\Re^2$ are Lebesgue-measurable, but not all subsets of $\Re^2$ are Lebesgue-measurable. In fact, not even every *bounded* subset of $\Re^2$ is Lebesgue-measurable. We cannot achieve the ability to measure all bounded subsets of $\Re^2$ without sacrificing invariance under rotation and translation.

2.1.6. BOREL SETS

Definition: A **Borel set** (named for Emile Borel, co-founder of probability and measure theory) is a countable union or intersection of open or closed subsets of $\Re^2$.

Let F_σ be the set of all countable unions of closed subsets of $\Re^2$, and let G_σ be the set of all countable intersections of open subsets of $\Re^2$. F_σ and G_σ are Borel sets. For every Lebesgue-

measurable set E, there are two corresponding Borel sets $A \in F_\sigma$ and $B \in G_\sigma$ such that $A \subset E \subset B$ and $l_L(B - A) = 0$.

Bounded Borel sets are Lebesgue-measurable, but not all Lebesgue-measurable sets are bounded Borel sets. Every Lebesgue-measurable set other than a Lebesgue-null set is a Borel set.

2.2. PROBABILITY THEORY

The following is a review of basic probability theory based on [Cormen et al. 90] and [Chung 74]. For more detail, see [Billingsley 86], [Drake 67], or [Feller 68]. Probability theory is a mathematically rigorous approach to describing the possible outcomes of an experiment. For example, querying a random number generator is an experiment in which the possible outcomes are numbers.

2.2.1. DEFINITION OF PROBABILITY SPACE

Each of the possible outcomes of the experiment is an **elementary event**. The **sample space** Ω is the set of all possible outcomes. The sample space may be finite (such as coin flips), countably infinite (such as arbitrary integers) or uncountably infinite (such as real numbers in [0,1]).

Some (typically all) subsets of Ω are **events** (as opposed to elementary events, as defined above). The set E of all events is the **event space**. The event Ω consisting of all the elementary events is the **certain event**, and the event $\varnothing$ consisting of no elementary events is the **null event**. Two events A and B are **mutually exclusive** if $A \cap B = \varnothing$. Elementary events are therefore always mutually exclusive of one another. Only events are **measurable** since only events will be assigned a probability. The set of events must be non-empty, and must be closed under the following operations:

Operation 1: Union of a countable number of events, and

Operation 2: Complement of an event.

Closure under the following operations is implied by the laws of set theory (including De Morgan's law):

Operation 3: Intersection of a countable number of events.

Operation 4: Difference of two events.

A set of subsets of Ω that is closed under operations 1 and 2 is a **Borel field**. All non-empty Borel fields include the certain event and the null event. Since only events are assigned a probability, we want to use as large a Borel field as possible. The set of all subsets of Ω is clearly a Borel field, and when the sample space is countable, this field is used as the event space. However, when the sample space is uncountably infinite, we must use a smaller Borel field, accepting the fact that not all subsets of Ω can be assigned a probability. The reason for this will become clear.

We are now ready to define the notion of probability of an event. A **probability distribution** or **probability measure** Pr is a function which maps events to real numbers (being the probability of the event) such that the following **probability axioms** are satisfied:

Axiom 1: $\Pr(A) \geq 0$ for any event A. Probabilities are non-negative.

Axiom 2: $\Pr(\Omega) = 1$. The certain event has probability 1. Probabilities are normalized.

Axiom 3: $\Pr\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} \Pr(A_i)$ for any countable set of pairwise mutually exclusive events $\{A_1, A_2, \ldots\}$. Probabilities are countably additive.

Note that any measure which satisfies Axiom 2 is a probability measure. As a special case of Axiom 3, we can easily show that $\Pr(A \cup B) = \Pr(A) + \Pr(B)$ for any two mutually exclusive events A and B. However, if the axiom were only to specify finite sets, we could not prove the same for countably infinite sets (see [Chung 74]). Since we want this to hold for all countable sets, we must define Axiom 3 as we did.

When working in a domain that does not have a finite measure, constructing a probability distribution satisfying Axiom 2 is not possible. In such cases, we must satisfy ourselves with a **measure** M(E) for any event E such that $M(\Omega)$ is countably infinite. To define a probability distribution, we must bound the domain such that it has a finite measure, and then normalize it as follows: $Pr(E) = \frac{M(E)}{M(\Omega)}$. For example, we will not be able to define a probability distribution for arbitrary real numbers, but must bound the domain to some interval on the real number line.

We can now formally define a **probability space** as a triple (Ω,E,Pr) consisting of a sample space Ω, an event space E, and a probability distribution Pr. When we refer to the probability that an event occurs, we typically do not explicitly refer to the sample space and event space forming the remainder of the triple. Furthermore, when we refer to a generator producing a value x, we might use the notation $Pr(x \in A)$ rather than the proper Pr(A) to refer to the probability that the event A occurs.

2.2.2. THEOREMS OF PROBABILITY

We now derive some known fundamental properties of probabilities from the probability axioms:

Theorem 1: $Pr(\varnothing) = 0$.

Proof:

By Axiom 3, $Pr(A \cup \varnothing) = Pr(A) + Pr(\varnothing)$ for any event A, since $A \cap \varnothing = \varnothing$. But $A \cup \varnothing = A$ by definition, so $Pr(A \cup \varnothing) = Pr(A)$, implying that $Pr(\varnothing) = 0$.□

Theorem 2: $A \subseteq B \Rightarrow Pr(A) \leq Pr(B)$.

Proof:

Assume $A \subseteq B$. Let $C = B - A$. By definition we know that $A \cap C = \varnothing$. Therefore, by Axiom 3, $Pr(A \cup C) = Pr(A) + Pr(C)$. But $A \cup C = B$, so $Pr(B) = Pr(A) + Pr(C)$. By Axiom 1, $Pr(C) \geq 0$, so $Pr(B) \geq Pr(A)$.□

Theorem 3: $Pr(A) + Pr(\Omega - A) = 1$.

Proof:

Since $A \cap (\Omega - A) = \varnothing$, Axiom 3 implies that $Pr(A \cup (\Omega - A)) = Pr(A) + Pr(\Omega - A)$. But $A \cup (\Omega - A) = \Omega$, and $Pr(\Omega) = 1$ by Axiom 2, so $Pr(A) + Pr(\Omega - A) = 1$.□

Theorem 4: $Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B)$.

Proof:

Let $B' = B - A$ (see Figure 2.4). Therefore $B = B' \cup (A \cap B)$ and $B' \cap (A \cap B) = \varnothing$. By definition $A \cup B = A \cup B'$ and $A \cap B' = \varnothing$. We apply Axiom 3 to get $Pr(B) = Pr(B') + Pr(A \cap B)$. Solving for $Pr(B')$ we get $Pr(B') = Pr(B) - Pr(A \cap B)$. We apply Axiom 3 again to arrive at $Pr(A \cup B) = Pr(A \cup B') = Pr(A) + Pr(B')$. Substituting for $Pr(B')$ we get $Pr(A \cup B) = Pr(A) + Pr(B) - Pr(A \cap B)$.□

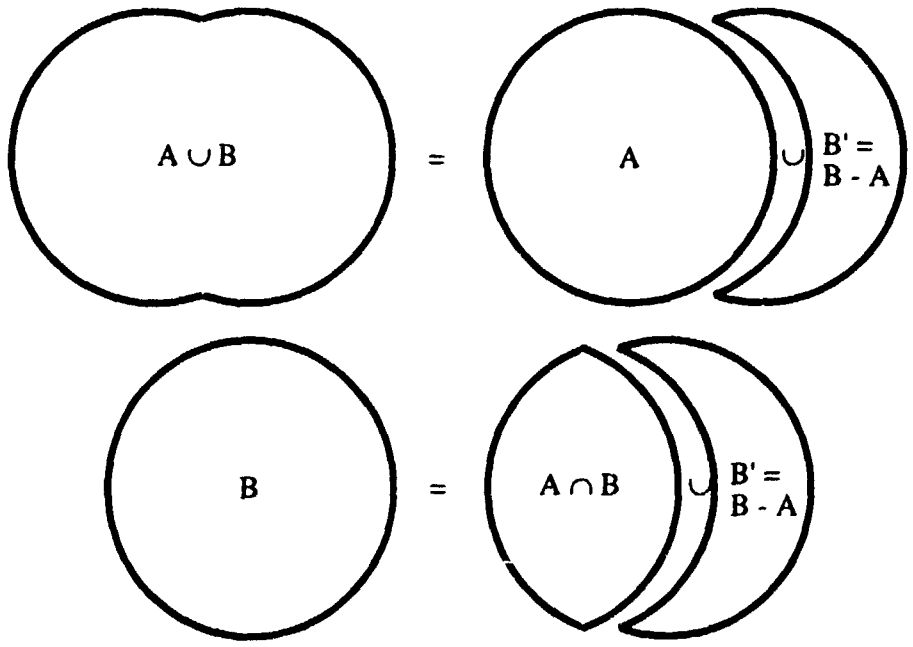


Figure 2.4. Breaking $A \cup B$ and B into non-intersecting parts on which to apply Axiom 3.

Theorem 5: $Pr(A) \leq 1$ for any event A.

Proof:

Since $A \subseteq \Omega$ by definition, Theorem 2 implies that $Pr(A) \leq Pr(\Omega)$. By Axiom 2 we get $Pr(A) \leq 1$.□

A probability distribution is **discrete** if its sample space is countable, and is **continuous** otherwise.

2.2.3. THE UNIFORM PROBABILITY DISTRIBUTIONS

Recall that our event space includes all subsets of the sample space Ω. In any discrete distribution, Axiom 3 directly implies that the probability of any event is always the sum of the probabilities of its elementary events. We can use this to define the **discrete uniform probability distribution** for finite sample spaces as follows:

Definition: Pr_u is **uniform** if $Pr_u(A) = \frac{1}{|\Omega|}$ for all elementary events $A \in \Omega$.

In other words, in a finite sample space, a uniform distribution is one in which each elementary event is equally likely.

To define the continuous uniform probability distribution, we need to use an event space E in which not all subsets of Ω are events. Specifically, the sample space Ω is the set of real numbers in the closed interval $[\Omega_1,\Omega_2]$, where $\Omega_1 < \Omega_2$. We would like to require each number in the interval to be equally likely, as we did above:

$Pr_u(A) = k$ for some fixed non-negative real number k and for all $A \in \Omega$.

However, this is not a complete definition of the function Pr_u. If this fixed probability $k > 0$, then by Axiom 3 we conclude that $Pr_u(\Omega) > 1$, since any event A consisting of more than $\left\lceil \frac{1}{k} \right\rceil$ elementary events would then have probability $Pr_u(A) > 1$, violating Axiom 2. Therefore $k = 0$. Axiom 3 implies that any countable union of elementary events has probability 0. However, the axioms do not imply a probability for events consisting of an uncountable number of elementary

events. For example, the event consisting of the real numbers in some range [a,b] consists of an uncountable number of numbers, and is therefore not assigned a probability by this definition. Since we wish such a set to be an event, we must assign a probability to such an event in order to completely define the function Pr_u. For this reason, we define our events to be only *some* of the subsets of the sample space. Specifically, we define a subset of the sample space to be an event if this subset can be obtained by a countable union of open or closed intervals. As required, this event set is closed under complement, union, and intersection. We then define Pr_u as:

Definition: Pr_u is the **continuous uniform probability distribution** on $[\Omega_1,\Omega_2]$, $\Omega_1 < \Omega_2$, if $Pr_u(A = [A_1, A_2]) = \dfrac{A_2 - A_1}{\Omega_2 - \Omega_1}$ for all A_1,A_2 satisfying $\Omega_1 \le A_1 \le A_2 \le \Omega_2$.

Note that this definition is just the normalization of the measure $I_L(A = [A_1,A_2]) = A_2 - A_1$, since $I_L(\Omega) = \Omega_2 - \Omega_1$. I_L is the Lebesgue-measure in $\Re^1$, as discussed in Section 2.1.5.

In an uncountable sample space, the uniform distribution can only be defined as one in which uncountably large sets of elementary events have specific probabilities. The probability of an event $A = [x,x]$, for any $x \in [\Omega_1,\Omega_2]$ is 0. Furthermore, since

$$[A_1,A_2] = [A_1,A_1] \cup (A_1,A_2) \cup [A_2,A_2],$$

Axiom 3 implies that $Pr_u([A_1,A_2]) = Pr_u((A_1,A_2))$. Since the set $\{x \mid x \in Q \text{ and } x \in \Omega\}$ of all rational numbers in the domain is countable, this set is an event, and, according to our definition of uniform probability, this event has probability 0. The complement of this event, being the set of all irrational numbers in the domain is therefore also an event, and has probability 1.

There is no general definition of the uniform probability distribution. We have only the discrete and continuous uniform distributions rather than one high level definition of uniformity. The only underlying principle is the intuitive notion that each of the elementary events should be equally likely. As we have shown, it is not possible to construct a probability distribution that meets the requirements of the axioms while satisfying $Pr_u(s) = k$ for some real $k > 0$ and for all $s \in \Omega$. Therefore uniformity can not be defined in general based on this notion.

2.2.4. AN ALTERNATIVE DEFINITION OF UNIFORMITY

Some texts, such as [Trivedi 82, page 138], define the continuous uniform probability distribution in terms of the **cumulative distribution function** or **CDF**, F(x), which is the probability that the number generated is less than or equal to x. The **probability density function** or **pdf**, $f(x) = \frac{dF(x)}{dx}$, is the derivative of the cumulative distribution function. Uniformity in the interval (Ω_1,Ω_2) can then be defined as $F(x) = \frac{x-\Omega_1}{\Omega_2-\Omega_1}$ or as $f(x) = \frac{1}{\Omega_2-\Omega_1}$. We can derive this definition from our definition above as follows. We define the event $E_x = \{y \mid y \in (\Omega_1,\Omega_2)$ and $y \le x\}$. Clearly $E_x = (\Omega_1,x)$, so, by our definition above, $Pr_u(E_x) = Pr_u([\Omega_1,x]) = \frac{x-\Omega_1}{\Omega_2-\Omega_1}$, precisely the value of F(x). This alternative definition is therefore equivalent.

2.3. GRAPH THEORY

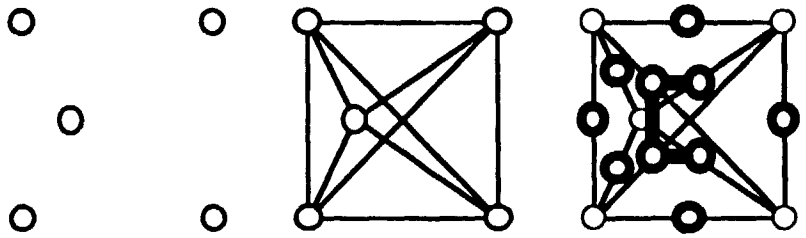
The following is a review of some graph theory relevant to this thesis. This section is based on [Bondy & Murty 76], [Cormen et al. 90], [Garey & Johnson 79], [Gavril 73], and [Urrutia 80].

2.3.1. VISIBILITY AND INTERSECTION GRAPHS

A **visibility graph** for a point set or polygon is a planar straight line graph whose vertices are the points of the point set or the vertices of the polygon and whose edges connect those vertices that are mutually visible in the point set or polygon. In the case of a point set, two points are **mutually visible** if no points lie on the segment joining them. In the case of a polygon, two vertices are mutually visible if the segment joining them lies entirely in the polygon. The visibility graph of a point set is therefore a complete graph if the point set is in general position (see Figure 2.5), since any two points in the set will be connected by an edge in some triangulation. Therefore the graph has $O(n^2)$ edges.

For ease of notation, we say that two edges or curves **cross** if they intersect at a point other than an endpoint. An **intersection graph** for a set of line segments, a set of curves, or a planar straight line graph is a graph whose vertices correspond to the edges or curves and whose edges connect those vertices that correspond to curves which cross one another. A **polygon**

intersection graph is the intersection graph of the visibility graph of the polygon. The **point set intersection graph** is correspondingly defined (see Figure 2.5). The intersection graph completely represents the information needed to find all possible triangulations of a point set or polygon. Since the intersection graph has vertices corresponding to edges of the visibility graph, it has $O(n^2)$ vertices for a set of n points or an n vertex polygon. Since it is possible for $O(n^2)$ of the edges in the visibility graph to each intersect $O(n^2)$ other edges, the total number of edges in the intersection graph is $O(n^4)$.



Point set has no three collinear points, but is not in general position.

Figure 2.5. A point set, the corresponding visibility and intersection graphs.

2.3.2. CIRCLE GRAPHS AND INDEPENDENT SETS

A **circle graph** is a graph that can be constructed as an intersection graph of a set of chords of a circle. A circle graph therefore has a vertex corresponding to each chord and edges connecting vertices corresponding to intersecting chords. We may assume without loss of generality that the chords do not share any endpoints (see [Gavril 73]).

An **independent set** is a set of vertices of a graph such that no two are connected by an edge in the graph. A **maximum** independent set is an independent set whose size is no less than that of any other independent set. A **maximal** independent set is an independent set that cannot be grown while maintaining its independence. There are $O(2^n)$ maximal or maximum independent sets of an n vertex graph.

Determining whether an arbitrary graph has an independent set whose size is at least a given integer is known to be NP-complete [Garey & Johnson 79]. Finding a maximum independent set in an arbitrary graph is therefore difficult. However, finding a maximum independent set in a circle graph can be done in polynomial time using the $O(n^3)$ algorithm of [Gavril 73] or the $O(n^2 \log n)$ algorithm of [Urrutia 80]. Therefore, the decision problem defined above is polynomial time solvable for circle graphs.

Counting maximum or maximal independent sets has not been studied as much. The problem is likely to be NP-complete for arbitrary graphs and polynomial time solvable for circle graphs. Algorithms for finding a maximum independent set in a circle graph, such as [Urrutia 80] can likely be adapted to count the number of such sets. Examples in which the number of maximum independent sets can be easily computed include the odd length cycle (n sets), the even length cycle (2 sets), the odd length chain (1 set), the even length chain ($\frac{n}{2}+1$ sets), and the complete graph (n sets).

2.3.3. TRIANGULATIONS

We can describe the triangulation of a point set or polygon by the set of vertices of the intersection graph corresponding to the edges which are included to form the triangulation. The notion that crossing edges cannot both be present in any triangulation can now be described by requiring that no two selected vertices of the intersection graph are connected by an edge. The notion that no edge can be added to a triangulated point set without causing an intersection can be described by requiring that as many vertices be selected as possible. These requirements are precisely those denoted by the maximum independent set of the intersection graph. There is a 1-to-1 correspondence between maximum independent sets and triangulations of the point set or polygon. The vertices in a maximal independent set correspond to the edges in the visibility graph which form the triangulation (see Figure 2.6). If the graph is an intersection graph of a point set or polygon, the maximal and maximum independent sets are equivalent, since any maximal independent set is a maximum independent set and vice versa.

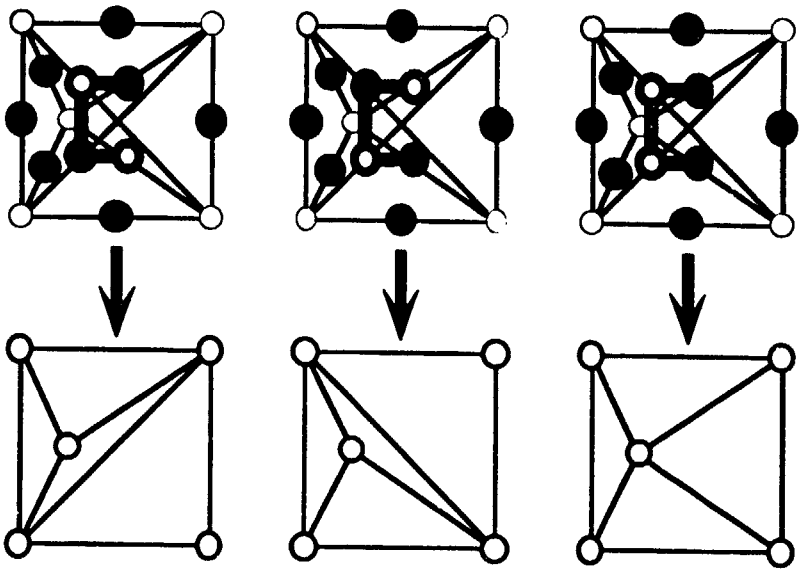


Figure 2.6. All the maximal independent sets and their corresponding triangulations.

For polygon intersection graphs, we know the size of all maximal independent sets is n - 2 if we include only the possible internal edges in the intersection graph, and not the polygon edges themselves, and is 2n - 2 if we include the n polygon edges as well. For point set intersection graphs, we know the size of all maximal independent sets is 3n - k - 3 if we include the convex hull edges, where n is the number of points in the set and k is the number of points in the convex hull. So, in both point sets and polygons, only O(n) of the $O(n^2)$ vertices of the intersection graph are included in any maximal independent set.

Theorem: A polygon intersection graph is a circle graph [Richter 92].

Proof:

Let the simple polygon be $P = (v_1, v_2, \ldots, v_n)$. We construct the circle and chords as follows. Start with a circle C, and place n points $(w_1, w_2, \ldots, w_n)$ on the perimeter of C at regular intervals, in clockwise order. These points correspond to the vertices of the polygon in clockwise order around the polygon. In other words, v_i corresponds to w_i. Construct chords of C for each possible triangulation edge of P. The intersection graph for

these chords is a circle graph. We can make the chords have distinct endpoints by using a circle slightly inset from C.

We now prove that this circle graph is the intersection graph of the polygon. Since the vertex set is the same for the graphs we are comparing, all we must prove is that the edge set is the same. We will do this by showing that edges which cross are mapped to chords which cross while edges that do not cross map to chords which do not cross. Let $e_1 = (v_a, v_b)$ and $e_2 = (v_c, v_d)$ be an arbitrary pair of possible triangulation edges of P.

If the edges e_1 and e_2 cannot both be in any triangulation of P then e_1 crosses e_2, so a,b,c,d are distinct indices and $e_1 \cap e_2 \neq \emptyset$. We therefore expect the chords $f_1 = (w_a, w_b)$ and $f_2 = (w_c, w_d)$ to intersect one another. We may assume without loss of generality that $a < b$ and $a < c < d$. The clockwise order of the four vertices of P, starting with a, must be either (a,c,b,d) or (a,d,b,c) for e_1 and e_2 to intersect one another. Therefore the clockwise order of the four vertices of the chords f_1 and f_2 must be one of these sequences. So $f_1 \cap f_2 \neq \emptyset$.

If the edges e_1 and e_2 are both in some triangulation of P then $e_1 \cap e_2 = \emptyset$ or $e_1 \cap e_2 \in \{v_a, v_b, v_c, v_d\}$. We therefore expect the chords $f_1 = (w_a, w_b)$ and $f_2 = (w_c, w_d)$ to not cross one another. If the four vertices of P are distinct, then the clockwise order of the four vertices of P, starting with a, must be (a,b,c,d), (a,b,d,c), (a,c,d,b), or (a,d,c,b). Therefore the clockwise order of the four vertices of the chords f_1 and f_2 must be one of these sequences. So $f_1 \cap f_2 = \emptyset$. If the four vertices of P are not distinct, then $e_1 \cap e_2 \in \{v_a, v_b, v_c, v_d\}$ implies that $f_1 \cap f_2 \in \{w_a, w_b, w_c, w_d\}$, so, in either case, f_1 and f_2 do not cross.

In summary, crossing edges of a polygon always have alternating endpoints, so their corresponding chords must intersect. Non-intersecting edges of a polygon never have alternating endpoints, so their corresponding chords cannot possibly intersect. □

The intersection graph for a point set may possess similar special properties of value, but we have not found any such properties. Specifically, the intersection graph for a point set need not be a circle graph. It is fairly clear that the capability to find the number of triangulations of a point set in polynomial time is a prerequisite for constructing a uniform random point set triangulation generator. To provide this capability, properties of the intersection graph would need to be used to compute the number of maximal independent sets in polynomial time.

The ability to compute the number of maximum independent sets (or maximal independent sets) in a circle graph in polynomial time therefore implies the ability to count triangulations of a simple polygon in polynomial time. However, unless this problem can be solved in a time complexity better than $O(n^2)$, this will not lead to an improved ability to generate uniformly random triangulations of polygons, as given below in Algorithm 10.7 (page 134). While our algorithm for counting triangulations takes $O(n^3)$ time, it computes the number of triangulations for all the sub-polygons needed to generate uniformly random simple polygons, thus saving a factor of $O(n)$. While algorithms which count maximal independent sets could also provide such useful intermediate results, it seems unlikely that an algorithm could be constructed that computes the number of maximal independent sets for the necessary sub-graphs in better than $O(n^3)$ time.

Chapter 3

3. RANDOM GEOMETRIC OBJECT GENERATORS

Before considering the generation of any particular geometric objects, we must formally define the notion of a random geometric object generator.

3.1. DEFINITION OF A RANDOM GEOMETRIC OBJECT GENERATOR

The primary goal of this thesis is to construct a generator capable of generating a wide selection of geometric objects. We will call such a generator a **random geometric object generator**. Although we cannot formally define all aspects of a random geometric object generator, we will clearly specify those parts that can be formally defined and those parts that cannot.

The inputs to a random geometric object generator are:

- the **type** of geometric object to generate
- the **size** of geometric objects to generate, *or* a distribution to use to select the size, *or* a geometric object on which to construct the results
- the **domain** in which to construct the geometric objects
- the desired **distribution** for the geometric objects produced

The output of a random geometric object generator is:

- a **stream** (sequence) of geometric objects

The properties of the resulting stream of geometric objects are described in Section 2.3 (page 31). For the remainder of this thesis, we will assume that the size is given directly rather than a distribution to use to select the size. If a distribution is given, we simply request a random number with the given distribution to determine the desired size. Some algorithms may not be capable of generating a geometric object of a given size. These algorithms may accept an approximation of

the size of the resulting object, or they may be completely incapable of predicting the size of the object generated.

3.2. EXAMPLES

In this section we give two examples of how to use a random geometric object generator.

3.2.1. GENERATION OF POINTS

We might generate a sequence of random points as follows:

- **type** of object is: point
- **size** of object is: not applicable
- **domain** is: axis parallel rectangle $R = \{(x,y) \mid x \in (R_L,R_R) \text{ and } y \in (R_B,R_T)\}$, where $R_L,R_R,R_B,R_T \in \Re$, $R_L \leq R_R$, $R_B \leq R_T$.
- desired **distribution** is: uniform

We would expect the generator to produce:

- a **stream** of uniformly distributed points in the rectangle R.

3.2.2. GENERATION OF TRIANGULATIONS OF A POINT SET

In this case, the inputs might be:

- **type** of object is: triangulation of a point set
- **geometric object** from which to construct the result is: the given point set
- **domain** is: the set of all triangulations of the given point set
- desired **distribution** is: uniform

and the output would then be:

- a **stream** of triangulations of the given point set, where each of the possible triangulations is equally likely to be produced.

3.3. PROPERTIES OF GEOMETRIC OBJECTS GENERATED AT RANDOM

Although it is often difficult to come up with a formal definition of randomness for geometric objects, there are several important properties that we expect geometric objects that have been generated at random to exhibit. For simplicity, we use the generation of simple polygons as an example throughout. Not all algorithms described in this thesis will have these properties.

3.3.1. ALL POSSIBLE OBJECTS ARE GENERATED

The objects should not share any properties other than those expected of all such objects. For example, random simple polygons should not have any special properties other than simplicity. Many algorithms exist to construct *nice* random simple polygons that are convex or star-shaped or monotone. Such polygons do not typically stress algorithms that accept arbitrary simple polygons. For example, triangulation of convex or monotone polygons is not likely to thoroughly test a triangulation algorithm for arbitrary simple polygons.

A more concrete way to look at this requirement is as follows: We expect the algorithm to be capable of generating any of the possibly infinite set of geometric objects that meet the requirements of the problem. For example, we may require the generator to be capable of producing all simple polygons that lie in some portion of the plane. In some cases, the set of all possible objects is infinite (such as all simple polygons in a portion of the plane), while in other cases the set is finite (such as all simple polygons on a given vertex set).

If the set of possible objects is finite, then we can formally state this property as $Pr_U(x) > 0$ for all possible objects $x \in \Omega$. On the other hand, if the set of possible objects is infinite, then it is typically the case that $Pr_U(x) = 0$ for all x, so we need another way to define this property.

Let G be a geometric object generator. Let Ω be the sample space we wish G to generate. In other words, we want G to be capable of generating any $x \in \Omega$.

Definition: The **result set** of a generator G, written as R(G), is the set of all objects that G can produce.

We use the term **complete** to refer to a generator that can produce all possible objects and never produces an impossible object:

Definition: A generator G is **complete** if $R(G) = \Omega$.

A generator G is **incomplete** if $R(G) \neq \Omega$.

A generator G is **over-complete** if $R(G) \supseteq \Omega$.

We will see that it is sometimes necessary to accept over-completeness in order to achieve some of the other properties listed in this section.

3.3.2. INVARIANCE UNDER EUCLIDEAN TRANSFORMATIONS

If we are considering geometric objects generated at random in some Euclidean space $\Re^n$, we would expect the probability to be unaffected by translations, rotations, and reflections (the Euclidean transformations). If E is an event consisting of a set of geometric objects, and E' is the event produced by bringing the geometric objects in E through a series of translations, rotations, and reflections, we would expect that Pr(E) = Pr(E'). We say that a probability distribution is **invariant under Euclidean transformations** [May & Smith 82] if this is the case. This idea is also referred to as planar isometry (by Euclid, for example).

Since we wish to work in a bounded domain, we must carefully word the formal definition:

Definition: A transformation is **Euclidean** if it is composed of translations, rotations, and reflections.

Definition: A probability distribution Pr is **invariant under Euclidean transformations** if, when the set of geometric objects G' produced by bringing the set of geometric objects G in an event E through a Euclidean transformation and there exists an event E' representing G', then Pr(E) = Pr(E').

An alternative, stronger definition of invariance is as follows: If a set of geometric objects G in an event E is brought through a Euclidean transformation, and the resulting set of geometric objects is G', and both G and G' lie entirely within the domain, then there exists an event E' representing G', and Pr(E) = Pr(E'). We will not use this alternative definition.

A common technique to ensure invariance under Euclidean transformations is to define the probability as a function of the Lebesgue measure (i.e., area in $\Re^2$, volume in $\Re^3$), since this measure is itself invariant under these transformations.

3.3.3. UNIFORM DISTRIBUTION

A primary goal for our random geometric object generator is that it has a uniform distribution. We will see that we cannot precisely define what a uniform distribution is for all types of geometric objects. In fact, it is one of the goals of this thesis to explore this notion for various types of geometric objects.

UNIFORMITY FOR GEOMETRIC OBJECTS

Armed with the tools needed to define probability distributions, we are now ready to consider how to define the uniform probability distribution for geometric objects. Asking a random geometric object generator for a geometric object can be thought of as an experiment in which the possible outcomes are geometric objects. Each of the possible outcomes is therefore an elementary event. The sample space may be finite (such as all triangulations of a given point set), or uncountably infinite (such as all points in the unit square). Since we do not consider integer domains, our sample spaces are always finite or uncountable.

If our sample space is finite, we can trivially define uniformity using the discrete uniform probability distribution, to be formally defined in Section 2.2.3 (page 21). A random geometric object generator has a **uniform distribution** if each of the finite number of possible outputs has the same probability of being generated. So, for example, a generator of random triangulations of a given point set is uniform if each of the finite number of triangulations is equally likely.

On the other hand, if our sample space is uncountable, defining uniformity is non-trivial. We clearly want to base such a definition on the continuous uniform probability distribution. We will see that a general definition of uniformity for all geometric objects is not possible.

UNIFORMITY FOR POINTS

As an example, we consider the construction of a uniform probability distribution for a very simple type of geometric object: points in the plane. Consider the definition of uniformity for points in the rectangle $D = \{(x,y) \mid x \in [D_L,D_R] \text{ and } y \in [D_B,D_T]\}$, where $D_L,D_R,D_B,D_T \in \Re$, $D_L \le D_R$, $D_B \le D_T$. The sample space consists of all points in D. We cannot satisfy all three probability axioms if we make all sets of points be events, since the size of the sample space is uncountable. We can use two numbers in $\Re^1$ to construct a point in $\Re^2$, and we can use two intervals in $\Re^1$ to construct an axis parallel rectangle in $\Re^2$, so the following definition of event is the logical extension of our event definition for numbers:

Definition: A point set in $\Re^2$ is an **event** if it can be obtained by a countable union of open or closed axis parallel rectangles.

This definition is not acceptable since the rotation of an event by an amount other than multiples of $\frac{\pi}{2}$ may not be representable as another event. However, let us continue working with this event space in order to bring out the difficulties in defining a uniform probability distribution for points.

Let the rectangle $R = \{(x,y) \mid x \in [R_L,R_R] \text{ and } y \in [R_B,R_T]\}$, where $R_L,R_R,R_B,R_T \in \Re$, $R_L \le R_R$, $R_B \le R_T$, and $R \subseteq D$. We can now define uniformity for points as follows:

Definition: Pr_u is **uniform** if $Pr_u(R) = \left(\frac{R_R - R_L}{D_R - D_L}\right)\left(\frac{R_T - R_B}{D_T - D_B}\right)$ for all (axis parallel) rectangles R satisfying $D_L \le R_L \le R_R \le D_R$ and $D_B \le R_B \le R_T \le D_T$.

We can simplify this definition using the area function:

Definition: Pr_u is **uniform** if $Pr_u(R) = \frac{I_L(R)}{I_L(D)}$ for all axis parallel rectangles $R \subseteq D$.

CONSEQUENCES

In defining our probability distribution, we divided by the area of the domain. If we were to consider a definition of uniformity for points in the unbounded plane, such a division would not be possible. In such a case, we must satisfy ourselves with defining a measure M rather than a probability distribution Pr, the difference being that $M(\Omega)$ may very well be infinite while $Pr(\Omega) = 1$ by definition. To make exposition easier, we will generally consider bounded domains.

One might ask whether this is the one and only definition of uniformity. Why do we define probability as a function of area rather than some other measure? Even our choice of event space could be questioned, since it does not support invariance under rotation. Although we can prove that this definition satisfies the probability axioms, we *cannot* prove that this is the one and only definition of uniformity for points. In fact, we will derive a probability space for points that provides invariance under Euclidean transformations.

Defining a particular probability distribution to be uniform may seem bold given that we recognize that the choice of a probability distribution is, in general, quite arbitrary. However, we can usually find, at least for fairly simple geometric objects, a probability distribution that is both useful and intuitively sensible (as we just did for points). We choose to give this probability distribution the name **uniform** so that we can easily refer to it, and we recognize that this probability distribution has no mathematical properties making it unique.

UNIFORMITY WITH RESPECT TO AN ALGORITHM

We have seen that defining uniformity for geometric object generators is non-trivial when the set of possible objects is uncountable. If we are producing these geometric objects as input to a particular algorithm, we may be able to avoid this problem in the following way.

If we assume that the objects are generated for a particular algorithm A, we could define uniformity with respect to the algorithm as follows. We could define some function f(p) that maps the infinite

domain of objects to some finite co-domain. We may then define uniformity with respect to the algorithm in this finite co-domain. Such a function f would give a high level description of the object. We would expect that two objects O_1 and O_2 whose descriptions are the same (i.e., $f(O_1) = f(O_2)$) would be processed in the same way by the algorithm.

More formally, we can define a partitioning of the domain into a finite set of sub-domains. This defines a natural equivalence relation on the domain with the implication that two equivalent objects are processed in the same way by the algorithm.

For example, the edge labeling of rectilinear polygons in [Sack 84] is a mapping from rectilinear polygons on n vertices (an uncountable set) to label sequences of length n (a finite set). If the algorithm makes decisions solely based on the label sequence, then it will behave identically for all polygons with the same label sequence. If our purpose is to test the algorithm, any polygon with a given label sequence will produce the same behavior.

If we have a particular algorithm in mind when we are producing the objects, and if we can devise such a function, then such a definition of uniformity is ideal. On the other hand, a different generator is needed for e: n algorithm, even if the algorithms require the same type of geometric object as input.

3.3.4. UNIFORM COVERAGE PROBABILITY

We will find **uniform coverage probability**, as defined below, to be useful in some applications.

Definition: A geometric object O **covers** x if $x \in O$.

Definition: The **coverage probability** $Pr_c(x)$ is the probability that a given value x in the domain is covered by the geometric object.

Definition: A geometric object generator has a **uniform coverage probability** if the coverage probability is fixed, independent of x (in other words, if $Pr_c(x) = p$ for some constant p). This probability (i.e., p) is called the **density**.

Examples of geometric objects in $\Re^1$ and $\Re^2$ where uniform coverage probability is used include arcs, intervals, simple arc or interval sets, and convex polygons. For example, the geometric object may be an interval or arc in $\Re^1$, and x might be a number. Another example is a simple polygon P in $\Re^2$ covering a point $p \in P$. A final example is a simple interval set $S = \{I_1, I_2, \ldots, I_k\}$ with density d in domain D. In this case, uniform coverage probability is satisfied if and only if $\forall x \in D, \sum_{i=1}^{k} Pr_c(x) = d$

It can be shown that $Pr_c(x) = \frac{E(I_L(O))}{I_L(D)}$, where O is the geometric object, D is the domain, I_L is the Lebesgue-measure (length in $\Re^1$, area in $\Re^2$), and E(f) is the expected value of f. For example, if O is an arc whose expected length is L_{exp}, and the domain is defined as $D = [D_1, D_2)$, then $Pr_c(x) = \frac{L_{exp}}{D_2 - D_1}$. In other words, the coverage probability is the expected length of the arc as a ratio of the total size of the domain. For example, if the probability of any point being covered by the arc is $\frac{1}{2}$ in D = [0,360), then the expected size of the arc is 180.

APPLICATIONS

Consider a simulation of customers queuing for bank tellers. There is a window at which customers wait for service. Tellers work shifts at the window. Only one teller works in the window at a time. Sometimes no teller is working at the window (i.e., it is closed), and customers must wait or go to another window. Each work shift is represented by an interval. The domain is the work day (time). We represent a set of work shifts for a given day as a simple interval set. If we wish to assign random work shifts for tellers such that the probability that the window is closed is independent of the time of day, then we require a uniform coverage probability.

Chapter 4

4. REQUIREMENTS

In this chapter we will state the capabilities we require in order to construct random geometric object generators.

4.1. MODEL OF COMPUTATION

In this section, we define the model of computation used for the algorithms described in this thesis. Most algorithms use the real RAM model, in which arithmetic operations can be performed on arbitrarily large infinite precision real numbers in constant time. For algorithms which work in a finite sample space, we use the RAM model, in which arithmetic operations can be performed on O(log n) bit integers in constant time, where n is the problem size. Some algorithms require a larger word size.

4.2. UNIFORM RANDOM NUMBER GENERATORS

The most obvious requirement is a supply of random numbers. Formally, we assume the existence of a generator of independently uniform random numbers. If pseudo-random generators are used, multiple independent generators should be provided, each having its own seed. We assume the ability to generate uniform random integers within any range, or to generate random real (typically floating point) numbers within any range, inclusive of or exclusive of either end of the range. Typically, this capability is provided by a pseudo-random number generator, which is a deterministic sequence of numbers that depends upon a seed, and which has a uniform distribution (see [Knuth 81]). While pseudo-random number generators produce sequences with properties that truly random sequences do not have, the algorithms we describe will not take advantage of any such properties, and should therefore work with any source of random numbers. We will assume a source of truly random numbers for the remainder of this thesis, ignoring any deficiencies introduced by pseudo-random number generators. If we need a non-uniform random number, we

can apply various algorithms to construct a generator from the required uniform generators (see [Knuth & Yao 76] or [Devroye 86]). For our complexity analysis we assume a random number generator takes O(1) time per number generated.

4.2.1. INCLUSIVE OR EXCLUSIVE

Whether the range specified when requesting a random integer is inclusive or exclusive is of little consequence, since we can easily adjust the endpoints of the range. For example, if a generator is inclusive of its start and exclusive of its end, and we desire a uniformly random integer between 1 and 100 (inclusive of both 1 and 100), we simple ask for a random number between 1 and 101.

However, for real numbers, it is not as simple to handle differences in the specification of the range. Although the probability of generating any single real number (including the endpoints of the range) is zero, it is still possible for the number to be generated. When working with floating point numbers, there are only a finite number of representable numbers in any given range, so the probability of generating a given number is non-zero. For floating point numbers, we might be able to adjust the range by the smallest representable amount. For example, if our generator is inclusive of its start and exclusive of its end, and we desire a uniformly random floating point number between 0 and 1, inclusive of both 0 and 1, we simply ask for a random number between 0 and the next largest representable number after 1. There are other representations of real numbers, and each requires its own solution to this problem.

4.2.2. GENERATING UNIFORM RANDOM FLOATING POINT NUMBERS

An interesting and rather esoteric aside involves the generation of random floating point numbers. Most practical real-world software uses some form of floating point representation for its real numbers. A floating point number is represented by a mantissa and an exponent (much like scientific notation). Such a representation is capable of making subtle distinctions between numbers near zero while being able to represent very large numbers. Consider the typical request for a uniform random floating point number between zero and one. If we look at all the

representable numbers in this range on a number line, there will be far more near zero than near one. It would seem that a good generator would be capable of producing any of the representable numbers in this range. However, since there are more representable numbers near zero, and since we require a uniform distribution, each of the representable numbers near zero should have a lower probability of being produced than numbers near one. Most real-world random number generators probably do not attempt to produce all possible representable numbers within the desired range, thereby not meeting our requirements of a random object generator.

4.3. POSSIBILITY OF NON-TERMINATION WITH LOW PROBABILITY

We will see that it is difficult to devise algorithms to construct geometric objects at random with a solid upper bound on their time complexity. It is often theoretically possible for these algorithms to never terminate. Most definitions of the term "algorithm" require guaranteed termination, so we are using the term loosely when we speak of algorithms that may never terminate. Typically, we can only specify a time bound which the algorithm will meet most of the time.

As a very simple example of an algorithm with this kind of bound, consider the problem of generating a set of *distinct* random integers of a given size. Let n be the number of distinct integers to be generated. Suppose the integers are to lie in the interval [a,b], where $a,b \in \Re$, $a < b$. Furthermore, let $d = b - a + 1$. Since no more than d distinct integers can be generated, we know that $n \leq d$. One way to generate a set of distinct integers is to repeatedly generate integers until enough distinct integers have been found. Suppose we request a random integer and get r_i. If we are dealing with a truly random integer generator, the probability that the next random integer $r_{i+1} = r_i$ is $\frac{1}{d}$. While most pseudo-random integer generators do not repeat any integer until a very large number of integers have been produced, we do not wish to take advantage of any such non-random properties of our generator. The probability that the same integer will be generated k times is $Pr(r_i = r_{i+1} = r_{i+2} = \ldots = r_{i+k-1}) = \frac{1}{d^{k-1}}$. This implies that there is no limit to the number of times an integer can be repeatedly produced by a random number generator. However, if n is

much less than d, then this algorithm will terminate in O(n) time with very high probability. We therefore say that this algorithm has an O(n) time complexity in this case.

Another example can be found in the problem of constructing a fair die roll from a fair coin, since there are 6 die rolls, and 6 is not a divisor of 2^m for any m (see [Knuth & Yao 76]). As well, whenever the rejection method [Devroye 86, page 40] is used, termination cannot be guaranteed. The rejection method will be discussed in Section 5.3 (page 44). Since many of the algorithms described below will use the rejection method, they have the potential to never terminate.

Chapter 5

5. GENERAL TECHNIQUES

5.1. RANDOM ORDERING OF A SET

We are often given a set of objects and asked to give them a random order. For example, randomized algorithms such as quicksort use random orderings to make the probability of the worst case scenario independent of the input. To generate a random order, we give the set an *arbitrary* order and then generate a random permutation of that ordering.

5.1.1. DEFINITION OF UNIFORMITY

Since the sample space is finite, we use the discrete uniform probability distribution:

Definition: All sets of permutations are **events**. A permutation generator is **uniform** if each of the n! permutations of n items has the same probability of being generated.

5.1.2. ALGORITHM 5.1

Input: A set $S = \{S_1, S_2, \ldots, S_n\}$ of n arbitrary objects.

Output: Destructive. S becomes a random permutation of S.

Properties: Capable of producing all permutations, uniform probability distribution.

Complexity: O(n).

There are two basic approaches to generating uniform random permutations. Either the items are permuted in place, or a new collection is constructed to hold the permutation. To permute a collection of objects in place (i.e., destructively), we use the technique described in [Knuth 81, page 139]:

FOR i := n DOWN TO 2 DO

Generate a uniform random integer $j \in [1,i]$.

IF $i \neq j$ THEN

Exchange items S_i and S_j.

END IF.

END FOR.

PROOF OF CORRECTNESS

Theorem: Algorithm 5.1 provides a uniform distribution.

Proof:

Algorithm 5.1 generates each of the n! permutations with the same probability. Consider an arbitrary permutation $X = \{x_1,x_2,\ldots,x_n\}$ (where x_i represents the index of the i^{th} element) of the n elements in the collection. For this permutation to be generated, The algorithm must select $j = x_1$ out of n total choices in the first iteration of the for loop, so the probability is $\frac{1}{n}$. The algorithm must also select $j = x_2$ out of n-1 total choices in the second iteration, giving a probability of $\frac{1}{n-1}$, and so on. For the algorithm to generate permutation X, all of these independent random choices must be made, so the net probability is the product of these terms: $\prod_{i=0}^{n-2} \frac{1}{n-i} = \frac{1}{n!}$. Therefore Algorithm 5.1 provides a uniform distribution.□

EXTENSIONS

A convenient way to generate a uniform random permutation without modifying the original collection is to simply copy the collection and then destructively permute the copy. The random permutation still takes O(n) time to be generated.

5.2. SETS OF DISTINCT GEOMETRIC OBJECTS

When asked to generate a set of n distinct geometric objects, we will use the following general technique. Repeatedly generate the geometric objects, placing them in the set. If an object already

in the set is generated, adding the object to the set is a null operation. Eventually, n distinct objects will be generated. If there are requirements on the members of the set, other than the requirement that members be distinct, then this technique cannot be used. When describing algorithms, we will simply say that we generate a set of distinct geometric objects of a particular type, implying that this algorithm is to be used, and that the algorithm may theoretically never terminate.

If we assume a brute force linear search is done to determine whether the object generated is already in the set, and we further assume an O(1) equality test for the objects in the set, then this algorithm will terminate with very high probability in $O(n^2 + nT)$ where T is the time taken to generate a single geometric object. The $O(n^2)$ is taken to do the n linear searches, and the O(nT) is taken to actually generate the O(n) geometric objects required to find n distinct geometric objects with high probability.

Geometric object sets that may be generated include sets of: numbers, angles, intervals, arcs, points, rectangles, and horizontal, vertical, axis parallel, or arbitrary line segments. Since generation of these sets is trivial using the technique described above, we will not consider these object sets further.

5.3. REJECTION METHOD

A powerful technique for constructing probability distributions is the rejection method [Devroye 86, page 40]. From an algorithmic point of view, the rejection technique is straightforward. If we are given an algorithm to generate some geometric objects at random, such as points in a square domain, and we wish to generate these same objects but with some additional restrictions (such as reducing the domain to a circle inside the square), then we simply request objects from the given generator until we get one that meets these additional requirements.

5.4. PARTITIONING THE DOMAIN

Suppose we are asked to generate geometric objects with a given uniform coverage probability $p = \frac{1}{n}$ for some positive integer n. The algorithm is as follows. We first partition the domain D

into n geometric objects $\{G_1,G_2,\ldots,G_n\}$. Next, we generate a uniform random integer $i \in [1,n]$. Finally, we return the geometric object G_i. Clearly, this algorithm provides a uniform coverage probability of $p = \frac{1}{n}$, regardless of how the partition is formed. For this technique to work, it must be possible to partition the domain into a finite number of pieces, each of which is a valid geometric object of the desired type. As well, we must prove in each application of this technique that any geometric object of the desired type in D can indeed be generated. To prove this, it is often necessary to place a lower bound such as 2 or 3 on n. This approach is used in most of the algorithms in this thesis that provide uniform coverage probability. Examples of geometric objects that can be generated using this technique include arcs, intervals, simple arc sets, simple interval sets, and convex polygons.

5.5. 1-TO-1 MAPPING

Suppose there exists a 1-to-1 mapping between the elementary events in sample space Ω and the elementary events in sample space Ω'. Let the 1-to-1 mapping be expressed as $f(s) = s'$ or $f^{-1}(s') = s$ where $s \in \Omega$ and $s' \in \Omega'$. If we have found a set of events $E \subseteq \Omega$ that is closed under countable union, countable intersection, and complement, and we have found a probability distribution $Pr(e \in E)$ satisfying the three probability axioms, then we can construct the set of events $E' \subseteq \Omega'$ and the probability distribution $Pr'(e' \in E')$ as follows.

We first define the 1-to-1 mapping between the event spaces: Let $F: E \rightarrow E'$ be defined by $F(e) = \{f(s) \mid s \in e\}$. The inverse of F can then be expressed as $F^{-1}(e') = \{f^{-1}(s') \mid s' \in e'\}$. We can now define our probability distribution over the mapped events as $Pr'(e' \in E') = Pr(F^{-1}(e'))$.

Claim: The set of events $E' = \{F(e) \mid e \in E\}$ is closed under countable union, countable intersection, and complement. $Pr'(e' \in E') = Pr(F^{-1}(e'))$ is a probability distribution.

Theorem: If Pr is the uniform probability distribution for Ω, then Pr' need not be the uniform probability distribution for Ω'.

Proof:

There may be multiple 1-to-1 mappings between Ω and Ω'. Therefore the uniform probability distribution for Ω could map to multiple distinct probability distributions for Ω'. These cannot all be uniform.□

Example:

Consider the following pair of 1-to-1 mappings between closed arcs and points:

1 $f([a,b]) = (a,b)$ and $f^{-1}((a,b)) = [a,b]$. Assume $a,b \in [0,2\pi]$, so points lie in the square $\{(x,y) \mid x,y \in [0,2\pi]\}$.

2 $g([r,\theta]) = (r\cos\theta, r\sin\theta)$ and $g^{-1}((x,y)) = \left[\sqrt{x^2+y^2}, \tan^{-1}\frac{y}{x}\right]$.

Assume $r \in \left(0,\frac{\pi}{2}\right]$ and $\theta \in \left[0,\frac{\pi}{2}\right]$, so points lie in the quarter disc $\left\{(x,y) \middle| x,y \geq 0 \text{ and } x^2+y^2 \leq \frac{\pi^2}{4}\right\}$.

If we are given a uniform probability distribution Pr for arcs, we could use either of these to construct probability distributions for points. While the first would give the expected behavior in which the probability that a point lies in any given region is proportional to the area of the region, the second would give the unexpected behavior in which the probability would be higher for regions near the center of the circle than for regions near the circumference. These cannot both be considered to be uniform.

Theorem: If Pr is the uniform probability distribution for a finite Ω, then Pr' is the uniform probability distribution for Ω'.

Proof:

Since Pr is uniform, we know that $Pr(x) = k$ for all $x \in \Omega$ and some k. There may be many 1-to-1 mappings between Ω and Ω'. For any such mapping, a given $y \in \Omega'$ will

have precisely one corresponding $x \in \Omega$. Therefore $Pr'(y) = Pr(x) = k$. Therefore, $Pr'(y) = k$ for all $y \in \Omega'$.□

So, while we can use 1-to-1 mappings to construct probability distributions for new domains, we cannot make any claim about these distributions being uniform, unless we are working in a finite domain.

5.5.1. KNOWN 1-TO-1 MAPPINGS

The following are some of the potentially useful 1-to-1 mappings:

- Closed arcs [a,b] to points (a,b). Mapping is defined by $f([a,b]) = (a,b)$ and $f^{-1}((a,b)) = [a,b]$. If $a,b \in D$, then points lie in the square $\{(x,y) \mid x,y \in D\}$.
- Closed intervals [a,b] to points (a,b). Mapping is defined by $f([a,b]) = (a,b)$ and $f^{-1}((a,b)) = [a,b]$. If $a,b \in D$, then points lie in the triangle $\{(x,y) \mid x,y \in D \text{ and } x \leq y\}$.
- Closed intervals [a,b] to sets of 2 numbers {a,b}. Mapping is defined by $f([a,b]) = \{a,b\}$ and $f^{-1}(\{a,b\}) = [a,b]$ where $a \leq b$.
- Line segments with distinct endpoints to sets of 2 distinct points. The endpoints of a line segment are a set of points since the segment is not directed. Any two points define a line segment. The points defining a line segment may be points in some portion of the plane, or they may be a subset of a given point set.
- Triangles with distinct vertices to sets of 3 distinct points. The vertices of a triangle are a set of three points. Any three points define a triangle (if we allow zero area triangles). The points defining a triangle may be points in some portion of the plane, or they may be a subset of a given point set.
- Binary trees on n - 2 nodes to triangulations of convex polygons with n vertices (this is a well known result -- see [Atkinson & Sack 92], for example).

Chapter 6

6. POINTS AND POINT SETS

In this chapter we consider the generation of points in various domains, as well as the generation of sets of points.

6.1. POINTS IN A RECTANGLE

In this section we consider the generation of points on the plane. This is a stepping stone in the generation of point sets and other geometric objects. The techniques developed here can easily be generalized to higher dimensional domains.

6.1.1. PROBLEM DEFINITION

We assume that the axis parallel rectangle defining the domain is $D = \{(x,y) \mid x \in [D_L,D_R] \text{ and } y \in [D_B,D_T]\}$, where $D_L,D_R,D_B,D_T \in \Re$, $D_L \le D_R$, $D_B \le D_T$. Each range can be inclusive or exclusive of its start and end, so, for example, the rectangle might exclude all sides: $D = \{(x,y) \mid x \in (D_L,D_R) \text{ and } y \in (D_B,D_T)\}$.

6.1.2. DEFINITION OF UNIFORMITY

This definition can be generalized to higher dimensional domains (i.e., points in $\Re^n$). The elementary events are points that lie in D. We wish to achieve invariance under Euclidean transformations. We therefore generalize the event space discussed in Section 3.3.3 (page 33):

Definition: A set of points is an **event** if it is Lebesgue-measurable, as defined in Section 2.1.5.

Definition: Pr_u is **uniform** if $Pr_u(E) = \frac{I_L(E)}{I_L(D)}$ for all events $E \subseteq D$.

Recall that the Lebesgue-measure of an axis parallel rectangle such as D is just its height multiplied by its width. When working in higher dimensional domains, we also use the Lebesgue measure in $\Re^n$, so the concept of area is replaced by volume in $\Re^3$.

Theorem: This definition of uniformity for points is a valid probability distribution.

Proof:

The set of events is closed under the following operations: union of a countable number of events, intersection of a countable number of events, and complement of an event:

Countable union, intersection, and complement: By Property 1, holds for all measures since measures are rings.

The probability axioms hold:

Axiom 1: $Pr_u(E) \geq 0$. By Axiom 2, this holds for all measures. Measures are non-negative.

Axiom 2: $Pr_u(\Omega) = 1$. The certain event is indeed an event since it is just the one rectangle equal to the domain, and rectangles are trivially Lebesgue-measurable. Holds, since $Pr_u(\Omega) = \frac{I_L(D)}{I_L(D)} = 1$.

Axiom 3: $Pr\left(\bigcup_{i=1}^{\infty} E_i\right) = \sum_{i=1}^{\infty} Pr(E_i)$ for any countable set of pairwise mutually exclusive events $\{E_1, E_2, \ldots\}$. By Property 4, we know that $I_L\left(\bigcup_{i=1}^{\infty} E_i\right) = \sum_{i=1}^{\infty} I_L(E_i)$. Therefore:

$$Pr\left(\bigcup_{i=1}^{\infty} E_i\right) = \frac{I_L\left(\bigcup_{i=1}^{\infty} E_i\right)}{I_L(D)} = \frac{\sum_{i=1}^{\infty} I_L(E_i)}{I_L(D)} = \sum_{i=1}^{\infty} \frac{I_L(E_i)}{I_L(D)} = \sum_{i=1}^{\infty} Pr(E_i). \square$$

Theorem: This definition of uniformity for points is invariant under Euclidean transformations.

Proof:

Since Lebesgue-measure is invariant under Euclidean transformations (Property 5), $I_L(E) = I_L(E')$, where E is any event and E' is the result of any Euclidean transformation on E. Clearly, $Pr_u(E) = \frac{I_L(E)}{I_L(D)} = \frac{I_L(E')}{I_L(D)} = Pr_u(E'). \square$

6.1.3. ALGORITHM 6.1

Input: A domain $D = \{(x,y) \mid x \in [D_L,D_R] \text{ and } y \in [D_B,D_T]\}$, where $D_L,D_R,D_B,D_T \in \Re$, $D_L \leq D_R$, $D_B \leq D_T$.

Output: A point P in D.

Properties: Complete, uniform probability distribution.

Complexity: O(1).

To generate a uniform random point $P = (P_x,P_y)$, we simply generate a uniform random x coordinate $P_x \in [D_L,D_R]$ and an independently uniform random y coordinate $P_y \in [D_B,D_T]$.

PROOF OF CORRECTNESS

We must show that the generator $G_{6.1}$ implementing Algorithm 6.1 is complete and provides a uniform probability distribution.

Theorem: $G_{6.1}$ is complete.

Proof:

Suppose we are given an arbitrary point $Q = (Q_x,Q_y) \subseteq D$. We now prove that our algorithm can generate this point. Since $Q_x \in [D_L,D_R]$ and $Q_y \in [D_B,D_T]$, it is possible for the random number generator to choose $P_x = Q_x$ and to choose $P_y = Q_y$. The point generated would then be $P = (P_x,P_y) = (Q_x,Q_y) = Q$. Therefore $G_{6.1}$ can generate any point in D.□

Theorem: $G_{6.1}$ provides a uniform distribution.

Proof:

We will only prove that axis parallel rectangle events are generated with the correct probability. Other events are not considered here. Let $R = \{(x,y) \mid x \in [R_L,R_R] \text{ and } y \in [R_B,R_T]\}$, where $R_L,R_R,R_B,R_T \in \Re$, $R_L \leq R_R$, $R_B \leq R_T$ be an arbitrary axis parallel

rectangle in D. Let $Pr_{6.1}$ be the probability distribution provided by Algorithm 6.1. We wish to show that $Pr_{6.1}(R) = Pr_u(R)$.

The Lebesgue-measure of the domain D is:

$$I_L(D) = (D_R - D_L)(D_T - D_B),$$

while the Lebesgue-measure of the region R is:

$$I_L(R) = (R_R - R_L)(R_T - R_B).$$

The probability the point generated lies in R is $Pr_{6.1}(R) = Pr(P_x \in [R_L, R_R)$ and $P_y \in [R_B, R_T))$. Since P_x and Py are independently generated, we can break this into the product of two probabilities:

$$Pr_{6.1}(R) = Pr(P \in R) = (Pr(P_x \in [R_L, R_R]))(Pr(P_y \in [R_B, R_T])).$$

Since P_x and P_y are chosen with a uniform distribution, we know that:

$$Pr(P_x \in [R_L, R_R]) = \frac{R_R - R_L}{D_R - D_L} \text{ and } Pr(P_y \in [R_B, R_T]) = \frac{R_T - R_B}{D_T - D_B}.$$

Therefore:

$$Pr_{6.1}(R) = \left(\frac{R_R - R_L}{D_R - D_L}\right)\left(\frac{R_T - R_B}{D_T - D_B}\right).$$

As expected:

$$Pr_{6.1}(R) = \frac{(R_R - R_L)(R_T - R_B)}{(D_R - D_L)(D_T - D_B)} = \frac{I_L(R)}{I_L(D)} = Pr_u(R). \square$$

EXTENSIONS

The problem is complicated by the question of whether the rectangle is inclusive or exclusive. If we assume the existence of uniform random number generators that are inclusive or exclusive of the start and/or end of the range of numbers they can generate, it should be possible to be inclusive or exclusive of any or all of the edges of the rectangle. For example, if we want to include all the edges of the rectangle, then we use inclusive random number generators for both the x and y coordinates.

We can easily extend this algorithm (and it's proof of correctness) to generate points in a higher dimension domain. For example, extending to three dimensions, we generate points uniformly in a rectangular parallelepiped (such as a cube) in $\Re^3$. The proof of correctness is then based on volume rather than area. In fact, we can easily generate points uniformly in $\Re^n$. Points in $\Re^n$ can also be thought of as n-tuples (sequences of n numbers).

6.2. POINTS IN A CIRCLE

The techniques described in this section can be generalized to spheres in $\Re^3$, or even to higher dimensional domains, and are detailed in [Devroye 86, page 233].

6.2.1. PROBLEM DEFINITION

The domain is the circle $D = \{(x,y) \mid (x - D_X)^2 + (y - D_Y)^2 \le D_R^2\}$, where $D_X, D_Y, D_R \in \Re$, $D_R > 0$ with center (D_X, D_Y) and radius D_R. We wish to generate a point whose Euclidean distance to (D_X, D_Y) is no more than D_R.

6.2.2. DEFINITION OF UNIFORMITY

As for points in a rectangle:

Definition: A set of points is an **event** if it is Lebesgue-measurable.

Definition: Pr_u is **uniform** if $Pr_u(E) = \frac{I_L(E)}{I_L(D)}$ for all events $E \subseteq D$.

6.2.3. ALGORITHM 6.2

Input: A domain $D = \{(x,y) \mid (x-D_X)^2 + (y-D_Y)^2 \le D_R^2\}$, where $D_X, D_Y, D_R \in \Re$, $D_R > 0$.

Output: A point P in D.

Properties: Complete.

Complexity: O(1).

The natural approach to generating points in a circle is to generate random polar coordinates. Specifically, we generate a uniform random angle $\theta \in [0,2\pi)$ and a uniform random radius

$r \in [0,D_R]$. We then return the point $(D_X + r\cos\theta, D_Y + r\sin\theta)$. Although this algorithm can generate any point in the circle, it does not provide a uniform probability distribution. To see why this is the case, compare the probability p_2 that the algorithm generates a point in the concentric circle with half the radius with the probability p_4 that the algorithm generates a point in the concentric circle with one quarter the radius. Clearly $p_2 = \frac{1}{2}$ and $p_4 = \frac{1}{4}$. We would therefore expect the areas of the two circle to be related in the same way. This is not the case, since area is proportional to the square of the radius. Therefore this algorithm does not provide a uniform probability distribution.

6.2.4. ALGORITHM 6.3

Input: A domain $D = \{(x,y) \mid (x-D_X)^2 + (y-D_Y)^2 \le D_R^2\}$, where $D_X, D_Y, D_R \in \Re$, $D_R > 0$.

Output: A point P in D.

Properties: Complete, uniform probability distribution.

Complexity: O(1), may not terminate.

This algorithm is based on the rejection method [Devroye 86, page 40]. Repeatedly generate uniform random points in the axis parallel square

$$S = \{(x,y) \mid x \in [D_X - D_R, D_X + D_R] \text{ and } y \in [D_Y - D_R, D_Y + D_R]\}$$

enclosing the circle. Reject points that lie outside the circle D. Accept the first point generated which lies in or on the circle. The acceptance condition for a point $P = (P_x, P_y)$ is $(P_x - D_X)^2 + (P_y - D_Y)^2 \le D_R^2$. The probability that the point generated is accepted is $\frac{\pi}{4}$. This can easily be shown by comparing the area of a unit radius circle with the area of the enclosing square. The expected number of points that are constructed to find one inside the circle is $\frac{4}{\pi}$. For higher dimensional domains, the expected number of points needed grows.

6.3. POINTS ON A CIRCLE

Some of the techniques described in this section can be generalized to spheres in $\Re^3$, or even to higher dimensional domains, and are detailed in [Knuth 81, pp. 130-131] and [Devroye 86, pp. 233-235].

6.3.1. PROBLEM DEFINITION

The domain is the circle $D = \{(x,y) \mid (x - D_X)^2 + (y - D_Y)^2 = D_R^2\}$, where $D_X, D_Y, D_R \in \Re$, $D_R > 0$ with center (D_X, D_Y) and radius D_R. We wish to generate a point whose Euclidean distance to (D_X, D_Y) is exactly D_R.

6.3.2. MAPPING POINTS ON A CIRCLE TO ANGLES

It is natural to define a 1-to-1 mapping between points on a circle and their angles. A point $(D_X + D_R \cos\theta, D_Y + D_R \sin\theta)$ is mapped to the angle θ. We know that the angle lies in $S = [0,2\pi)$.

6.3.3. DEFINITION OF UNIFORMITY

We have defined a 1-to-1 mapping between points on a circle and angles in $[0,2\pi)$. We use this mapping to define uniformity for points on the circle as follows:

Definition: A set of points on a circle is an **event** if the set of angles corresponding to the points is Lebesgue-measurable, as defined in Section 2.1.5.

Definition: Let E be any event and let F be the set of angles corresponding to E. Then Pr_u is **uniform** if $Pr_u(E) = \frac{I_L(F)}{I_L(S)}$ for all events $E \subseteq D$.

6.3.4. ALGORITHM 6.4

Input: A domain $D = \{(x,y) \mid (x-D_X)^2 + (y-D_Y)^2 = D_R^2\}$, where $D_X, D_Y, D_R \in \Re$, $D_R > 0$.

Output: A point P on D.

Properties: Complete, uniform probability distribution.

Complexity: O(1).

This algorithm is simple, but cannot be easily extended to higher dimensions. The computation is slow if the result is to be given in Cartesian coordinates, since it requires evaluation of trigonometric functions. Generate a random angle θ uniformly in $[0,2\pi)$. Return the point $P = (D_X + D_R \cos\theta, D_Y + D_R \sin\theta)$.

6.3.5. ALGORITHM 6.5

Input: A domain $D = \{(x,y) \mid (x-D_X)^2 + (y-D_Y)^2 = D_R^2\}$, where $D_X, D_Y, D_R \in \Re$, $D_R > 0$.

Output: A point P on D.

Properties: Complete, uniform probability distribution.

Complexity: O(1).

This algorithm can easily be extended to higher dimensions, but this algorithm is slow because it requires evaluation of square roots (as well as generation of normally distributed random numbers). Generate two independently normal (not uniform) random numbers N_x and N_y with mean 0 and variance 1. Let the point $N = (N_x, N_y)$. Compute $|N| = \sqrt{N_{x^2} + N_{y^2}}$. Return the point $P = \left(D_X + \frac{D_R N_x}{|N|}, D_Y + \frac{D_R N_y}{|N|}\right)$.

6.3.6. ALGORITHM 6.6

Input: A domain $D = \{(x,y) \mid (x-D_X)^2 + (y-D_Y)^2 = D_R^2\}$, where $D_X, D_Y, D_R \in \Re$, $D_R > 0$.

Output: A point P on D.

Properties: Complete, uniform probability distribution.

Complexity: O(1), may not terminate.

This algorithm is slow because it requires evaluation of square roots. Generate a point $Q = (Q_X, Q_Y)$ uniformly distributed *in* the unit circle $\{(x,y) \mid x^2 + y^2 \leq 1\}$ (using rejection from the enclosing square, for example). Compute $|Q| = \sqrt{Q_x^2 + Q_y^2}$. Return the point:

$$P = \left(D_X + \frac{D_R Q_x}{|Q|}, D_Y + \frac{D_R Q_y}{|Q|} \right)$$

6.3.7. ALGORITHM 6.7

Input: A domain $D = \{(x,y) \mid (x - D_X)^2 + (y - D_Y)^2 = D_R^2\}$.

Output: A point P on D.

Properties: Complete, uniform probability distribution.

Complexity: O(1), may not terminate.

This algorithm is based on the rejection method. It is easy to implement, and it executes fast because it avoids use of the square root. Repeatedly generate uniform random points in the axis parallel square $S = \{(x,y) \mid x \in [D_X - D_R, D_X + D_R]$ and $y \in [D_Y - D_R, D_Y + D_R]\}$ enclosing the circle. Reject points that lie outside the circle. Accept the first point Q generated which lies in or on the circle. The acceptance condition for a point $Q = (Q_x, Q_y)$ is $(Q_x - D_X)^2 + (Q_y - D_Y)^2 \leq D_R^2$. Do not return Q, but instead return the following function of Q: Return the point:

$$P = \left(\frac{Q_x^2 - Q_y^2}{Q_x^2 + Q_y^2}, \frac{2 \times Q_x \times Q_y}{Q_x^2 + Q_y^2} \right).$$

See [Devroye 86, page 234] for a proof of correctness.

6.4. POINTS IN A TRIANGLE

In this section we consider the generation of points within a triangular domain. This is a stepping stone in the generation of points within a polygonal domain.

6.4.1. PROBLEM DEFINITION

We assume that the triangle is defined by three non-collinear distinct points $D_1 = (D_{X1},D_{Y1})$, $D_2 = (D_{X2},D_{Y2})$, $D_3 = (D_{X3},D_{Y3})$, so $D = \{(x,y) \mid x = a\,D_{X1} + b\,D_{X2} + c\,D_{X3}$ and $y = a\,D_{Y1} + b\,D_{Y2} + c\,D_{Y3}\}$ where $a, b, c \geq 0$, and $a + b + c = 1$. Each of the three edges of the triangle can be considered to be inside or outside the triangle, and each of the three vertices can be considered inside or outside the triangle. Typically, we define the triangle to be inclusive of all three edges and all three vertices. However, for other purposes, it is sometimes useful to include only some edges or vertices and to exclude the others.

6.4.2. DEFINITION OF UNIFORMITY

As for points in a rectangle:

Definition: A set of points is an **event** if it is Lebesgue-measurable, as defined in Section 2.1.5.

Definition: Pr_u is **uniform** if $Pr_u(E) = \dfrac{l_L(E)}{l_L(D)}$ for all events $E \subseteq D$.

6.4.3. ALGORITHM 6.8

Input: A triangular domain D.

Output: A point P in D.

Properties: Complete, uniform probability distribution.

Complexity: O(1).

Several techniques exist for generating random points in a triangle with uniform distribution. The simplest technique to describe is due to M. D. Atkinson, and is as follows. Construct a

parallelogram from the triangle by reflecting the triangle about one of its edges, and then flipping the reflected triangle. We will select a random point within the parallelogram. If the point does not lie within the triangle, we will invert the above transformation to map the point to one that is within the triangle. To generate a uniformly distributed random point within a parallelogram, use two uniformly random numbers between 0 and 1 to scale each of two vectors corresponding to two adjacent edges of the parallelogram. Adding the resulting scaled vectors gives a relative position in the parallelogram. This technique can clearly generate any point in the parallelogram, and therefore any point in the triangle, but does the generator provide a uniform distribution? The difference between the parallelogram and the rectangle is that the vectors are not perpendicular. It seems that the points are still generated with uniform distribution within the parallelogram, and are therefore also uniform within the triangle.

6.4.4. ALGORITHM 6.9

Input: A triangular domain D with vertices D_1, D_2, and D_3.

Output: A point P in D.

Properties: Complete, uniform probability distribution.

Complexity: O(1), if domain is open or semi-open then algorithm may not terminate.

The following algorithm is due to [Devroye 86, page 569] and is a specialization of Algorithm 6.10 (page 60) for generating points uniformly in a convex polygon. Construct the vectors $\bar{v}_1$, $\bar{v}_2$, and $\bar{v}_3$ from the origin to each of the vertices of the triangle. Generate two uniformly distributed random numbers between 0 and 1. Let U be the smaller of the two numbers and let V be the larger. The resulting point is constructed as:

$$U\bar{v}_1 + (V - U)\bar{v}_2 + (1 - V)\bar{v}_3$$

It is not obvious that a uniform distribution is provided. However, this algorithm is clearly easy to implement. See [Devroye 86, page 569] for a proof of correctness.

Regardless of which of the above algorithms one chooses, we can handle different definitions for the triangle as follows. If we wish to exclude points on one or more edge of the triangle, we can simply test whether the point is wanted, and if it is not, try again. Since the probability that the point is rejected is zero, the expected number of iterations is one.

6.5. POINTS IN A CONVEX POLYGON

In this section we consider the generation of points in a convex polygonal domain. This is a stepping stone in the generation of points within an arbitrary polygonal domain. Since triangles are convex polygons, these techniques generalize those described above for generating points in a triangle. Techniques developed for convex polygons can be generalized to convex polyhedra in higher dimensional domains.

6.5.1. MOTIVATION

Some algorithms require as input a convex polygon and a constant number of points in the interior of the polygon. For example, a point inclusion algorithm would require a polygon and a query point. Such algorithms can be tested using a random convex polygon generator (will be considered later) and a random point generator (as described in this section).

6.5.2. PROBLEM DEFINITION

We assume that the polygon is defined by points $\{v_1, v_2, \ldots, v_n\}$, and that these points are in general position (no three points collinear). We define the polygon to be inclusive of all bounding edges and all vertices.

6.5.3. DEFINITION OF UNIFORMITY

As for points in a rectangle:

Definition: A set of points is an **event** if it is Lebesgue-measurable, as defined in Section 2.1.5.

Definition: Pr_u is **uniform** if $Pr_u(E) = \frac{I_L(E)}{I_L(D)}$ for all events $E \subseteq D$.

6.5.4. ALGORITHM 6.10

Input: A convex polygonal domain D of size n.

Output: A point P in D.

Properties: Complete, uniform probability distribution.

Complexity: O(n).

The basic approach is to construct the point as a linear combination of the vertices defining the convex polygon. This technique is described in [Devroye 86, page 568].

Generate the sequence $(S_1,S_2,\ldots,S_{d+1})$ of spacings between d independently uniform random numbers in [0,1]. This sequence has the property that $S_i \geq 0$ for all i, and that $\sum_{i=1}^{d+1} S_i = 1$. Spacings can be generated by constructing a set of uniform random numbers and sorting them in O(n log n) time, or we can generate uniform order statistics via exponential spacings [Devroye 86, page 214] in O(n) time. Given the spacings, we generate the resulting point P as:

$$P = \sum_{i=1}^{d+1} S_i v_i$$

See [Devroye 86] for proof of correctness.

6.6. POINTS IN A SIMPLE POLYGON

In this section we consider the generation of points in a polygonal domain.

6.6.1. MOTIVATION

Many algorithms require as input a polygon and a constant number of points in the interior of the polygon. For example, a shortest path algorithm would require a polygon, a source point, and destination point. A shortest path tree algorithm would require a polygon and a source point. Such algorithms can be tested using a random polygon generator (will be considered later) and a random point generator (as described in this section). The same approach can be used to test a point visibility algorithm. As well, an algorithm that determines whether a given query point lies inside or outside a given polygon can be tested in a similar manner.

6.6.2. PROBLEM DEFINITION

We are given a simple polygon in standard form. Each of the edges of the polygon can be considered to be inside or outside the polygon, and each of the vertices can be considered inside or outside the polygon. For the following discussion, we will assume the polygon to be defined to be inclusive of all edges and all vertices. In other words, we assume the polygon is a closed region.

6.6.3. DEFINITION OF UNIFORMITY

As for points in a rectangle:

Definition: A set of points is an **event** if it is Lebesgue-measurable, as defined in Section 2.1.5.

Definition: Pr_U is **uniform** if $Pr_u(E) = \frac{I_L(E)}{I_L(D)}$ for all events $E \subseteq D$.

6.6.4. ALGORITHM 6.11

Input: A simple polygonal domain D with n vertices.

Output: A point P in D.

Properties: Complete, uniform probability distribution.

Complexity: O(n), may not terminate.

Given the capability to generate uniformly random points within a triangle, it is easy to see that we can generate uniformly random points within a triangulated polygon. We must simply select a random triangle and then select a random point within the triangle. To achieve an overall uniform distribution, it is essential that the probability that a given triangle is selected be proportional to its area.

SELECTING A RANDOM TRIANGLE

Given that we can determine the areas of each of the triangles, we must select a random triangle using a weighted distribution. This can be done as follows: Give the triangles an arbitrary

ordering. Compute the total area of the polygon. The triangles now define a partitioning of the real numbers between 0 and the total area, with the size of any one partition being equal to the area of the corresponding triangle. We can therefore select a random triangle by generating a single uniform random number between 0 and the total area of the polygon, and determining which partition the number falls in.

COMPUTING THE AREA OF A TRIANGLE

All that is left is to determine the areas of the triangles, which can be done as follows: The area of a polygon defined by (x_1,y_1), (x_2,y_2), and (x_3,y_3) is:

$$\frac{1}{2}(x_1y_2 - x_2y_1 + x_1y_3 - x_3y_1 + x_2y_3 - x_3y_2)$$

AVOIDING OVERLAP OF THE TRIANGLES

This problem is complicated by the possibility that the random point may lie on an edge (or vertex) of the triangle. Since the triangles that form the triangulation of the polygon overlap at their edges, the diagonals are really being counted twice. In other words, since a point on a diagonal could be generated as a point in either adjacent triangle, the probability that our random point will lie on a diagonal is twice what it should be. To correct this shortcoming, we simply arbitrarily choose one of the two adjacent triangles and include the diagonal only in that triangle. When a random point is chosen in the other triangle, it is exclusive of that edge. We use a similar approach to avoid counting each vertex more than once. Any given vertex is arbitrarily assigned to one of the many possible triangles that contain it.

6.7. POINT SETS IN A RECTANGLE

In this section we consider the generation of sets of points. Such sets have many applications.

6.7.1. MOTIVATION

Point sets are used as inputs to many algorithms. One can take the convex hull of a point set (see for example, [Preparata & Shamos 85, page 89]), or one can find a Voronoi diagram for a point set

(see for example, [Preparata & Shamos 85, page 198]). To test these algorithms, we need a generator of random point sets.

6.7.2. PROBLEM DEFINITION

We assume that the point set is defined by one or more distinct points, all of which lie in a given axis parallel rectangle $D = \{(x,y) \mid x \in (D_L,D_R) \text{ and } y \in (D_B,D_T)\}$, where $D_L,D_R,D_B,D_T \in \Re$, $D_L \le D_R$, $D_B \le D_T$. We are given the size n of the set to generate.

6.7.3. ALGORITHM 6.12

Input: An axis parallel rectangular domain $D = \{(x,y) \mid x \in [D_L,D_R] \text{ and } y \in [D_B,D_T]\}$, where $D_L,D_R,D_B,D_T \in \Re$, $D_L \le D_R$, $D_B \le D_T$.

The number of points n.

Output: A point set of size n in D.

Properties: Complete.

Complexity: O(n), may not terminate.

To construct a random point set with a specified number of points, we simply collect a set of distinct random points. As we observed above in Algorithm 6.1, a random point that lies in D can be constructed by selecting a random x coordinate in $[D_L,D_R]$ and a random y coordinate in $[D_B,D_T]$.

Chapter 7

7. OBJECTS ON THE NUMBER LINE

Although objects in a one dimensional domain, such as numbers, angles, intervals and arcs are not normally thought of as geometric objects, they correspond to geometric objects such as points and line segments in the plane. As with most problems, it helps to understand the problem in a simpler domain before we consider the more general case. We therefore consider objects on the number line ($\Re^1$) briefly before going on to more complex geometric objects in the plane.

7.1. NUMBERS

Numbers on the number line correspond to points on the plane, and are therefore considered as a stepping stone.

There are two common representations of numbers. Integers can represent any whole number within some range. Floating point numbers can approximate real numbers in a wider range, with a fixed number of significant digits. We further require that all the numbers lie in some domain. For example, the domain might be the numbers between zero and one, inclusive of zero and one. In general, either end of the domain could be included or excluded.

For real numbers we use the continuous uniform probability distribution in the interval $\Omega = [\Omega_1, \Omega_2]$:

Definition: Pr_u is **uniform** if $Pr_u(A = [A_1, A_2]) = \frac{A_2 - A_1}{\Omega_2 - \Omega_1}$ for all $A \subseteq \Omega$.

As stated above in Section 4.2 (page 38), we require the ability to generate random numbers with any distribution we desire, so generation of numbers in O(1) time with a uniform distribution is a given.

7.2. ANGLES

Generation of random numbers in a circular domain is no different than their generation in a non-circular domain. We simply define our domain to be inclusive of one end and exclusive of the other, so that the position at which the circular domain "wraps around" is not over-represented. For example, to generate a uniform random angle in radians, we request a uniform random number in the semi-open interval $[0,2\pi)$.

7.3. ARCS

Generation of intervals on the number line is easier when the domain is considered to be circular rather than non-circular. We therefore consider this case as a stepping stone to intervals. In a circular domain, intervals are referred to as arcs, since they represent a portion of a circular domain.

7.3.1. PROBLEM DEFINITION

We will assume the arc to be inclusive of its start and exclusive of its end. We require that both ends lie in some circular domain D. We will assume that the domain is of the form $D = [D_1,D_2)$. For example, the domain might be [0,360) or $[0,2\pi)$. The expected length L_{exp} or the actual length L of the arc may be given.

For example, we may be asked to generate a random arc that covers half of the circle on average, measured in degrees. The domain is therefore D = [0,360). The expected length of the arc produced is $L_{exp} = 180$.

7.3.2. MAPPING ARCS TO POINTS IN THE PLANE

Since arcs and points are both described by ordered pairs of numbers, it is natural to define a 1-to-1 mapping between points in $\Re^2$ and arc as follows. An arc [x,y] is mapped to the point (x,y) in the plane. Since $x,y \in D$, we know that the point lies in the square defined by $\{(x,y) \mid x,y \in D\}$. Therefore, the domain D maps to a square S defined by $S = \{(x,y) \mid x,y \in D\}$. Since the mapping

is 1-to-1, we may use point and arc terminology interchangeably without explicitly describing the necessary conversions.

7.3.3. DEFINITION OF UNIFORMITY

This definition can be generalized to higher dimensional domains. The elementary events are arcs that lie in the given domain D. Arcs may be open, semi-open, or closed. We wish to achieve invariance under Euclidean transformations. We therefore base our event space on the event space for points in the plane, as discussed in Section 6.1.2:

Definition: A set of arcs is an **event** if the set of points corresponding to the intervals is Lebesgue-measurable, as defined in Section 2.1.5.

Definition: Let E be any event and let F be the set of points corresponding to E. Then Pr_u is **uniform** if $Pr_u(E) = \frac{I_L(F)}{I_L(S)}$ for all events $E \subseteq D$.

Theorem: This definition of uniformity for arcs is a valid probability distribution.

Proof:

The set of events is closed under the following operations: union of a countable number of events, intersection of a countable number of events, and complement of an event. Since the event space is exactly the same as for points (see Section 6.1.2, page 48), the proof is not repeated here.

The probability axioms hold:

Axiom 1: $Pr_u(E) \geq 0$. Holds, since the area function is non-negative, and since $I_L(S)$ is non-zero.

Axiom 2: The certain event Ω consists of the square S. $Pr_u(\Omega) = 1$ since $Pr_u(\Omega) = \frac{I_L(\Omega)}{I_L(S)} = \frac{I_L(S)}{I_L(S)} = 1$.

Axiom 3: $Pr\left(\bigcup_{i=1}^{\infty} E_i\right) = \sum_{i=1}^{\infty} Pr(E_i)$ for any countable set of pairwise mutually exclusive events $\{E_1, E_2, \ldots\}$. Let F_i be the set of points that corresponds to E_i for all $i \geq 1$. Since

the mapping from arcs to points is 1-to-1, we know that F_i and F_j are mutually exclusive if $i \neq j$. By Property 4, we know that $I_L\left(\bigcup_{i=1}^{\infty} F_i\right) = \sum_{i=1}^{\infty} I_L(F_i)$. Therefore:

$$\Pr\left(\bigcup_{i=1}^{\infty} E_i\right) = \frac{I_L\left(\bigcup_{i=1}^{\infty} F_i\right)}{I_L(S)} = \frac{\sum_{i=1}^{\infty} I_L(F_i)}{I_L(S)} = \sum_{i=1}^{\infty} \frac{I_L(F_i)}{I_L(S)} = \sum_{i=1}^{\infty} \Pr(E_i). \square$$

Theorem: This definition of uniformity for arcs is invariant under Euclidean transformations.

Proof:

Euclidean transformations in $\Re^1$ are simply translations and reflections. Let E be any event and let E' be the result of applying a Euclidean transformation to E. Further suppose that E' is an event. Let F and F' be the sets of points corresponding to E and E' respectively. F and F' are both Lebesgue-measurable by our definition of events. Given our mapping from arcs to points, it is clear that F' is the result of applying a Euclidean transformation on F. Specifically, the transformation consists of a translation along the line x = y and/or a reflection. Since Lebesgue-measure is invariant under Euclidean transformations (Property 5), $I_L(F) = I_L(F')$. So, $\Pr_u(E) = \frac{I_L(F)}{I_L(D)} = \frac{I_L(F')}{I_L(D)} = \Pr_u(E'). \square$

7.3.4. ALGORITHM 7.1

Input: A domain $D = [D_1, D_2)$.

Output: A closed arc in D.

Properties: Complete, uniform probability distribution, invariant under Euclidean transformations.

Complexity: O(1).

To generate a random arc A, we simply generate two independent uniform random numbers $x, y \in D$, and return $A = [x, y]$.

The numbers x and y map to a point in the square $\{(x,y) \mid x,y \in D\}$. We are simply generating a uniformly random point in the square, exactly as done in Algorithm 6.1.

PROOF OF CORRECTNESS

Since our definition of uniformity is just that of a point, and since we generate an arc by generating a uniform random point, correctness is implied by the correctness of Algorithm 6.1.

7.3.5. ALGORITHM 7.2

Input: A circular domain $D = [D_1,D_2)$.

Output: An arc in D.

Properties: Complete, uniform coverage probability = $\frac{1}{2}$, fixed expected arc size = $\frac{D_2 - D_1}{2}$.

Complexity: O(1), may not terminate.

First, we generate two distinct uniform random angles $a,b \in D$. To do this, we use the set generation technique (i.e., produce random angles until two distinct angles are found). Next, we flip a fair coin to decide whether to return $A = [a,b)$ or $A = [b,a)$.

For any pair of angles $x,y \in D$, the probability that this algorithm would generate $[x,y)$ is the same as the probability that it would generate $[y,x)$. In other words, $Pr(A = [x,y)) = Pr(A = [y,x))$ for all $x,y \in D$ such that $x \neq y$. The coverage probability for any angle is therefore $\frac{1}{2}$, since $[x,y)$ and $[y,x)$ form a partition of D (see Section 5.4, page 44). So, the expected arc size is half the domain size, or $\frac{D_2 - D_1}{2}$. This algorithm can clearly generate any arc in D.

7.3.6. ALGORITHM 7.3

Input: A circular domain $D = [D_1, D_2)$, and the arc length L.

Output: An arc in D.

Properties: Complete, uniform coverage probability = $\frac{L}{D_2 - D_1}$, fixed arc size = L.

Complexity: O(1).

We are given L, the length of the arc to be produced. We first generate a random number m uniformly within the domain. This number will represent the midpoint of the arc. Since the length of the arc is given, we have a complete description of the arc in the pair (m,L). Clearly the probability of hitting the arc is $\frac{L}{D_2 - D_1}$. Since m is chosen with uniform distribution, the arc coverage probability is fixed.

7.3.7. ALGORITHM 7.4

Input: A circular domain $D = [D_1, D_2)$, and the expected arc length L_{exp}.

Output: An arc in D.

Properties: Complete, uniform coverage probability = $\frac{L_{exp}}{D_2 - D_1}$.

Complexity: O(1).

As above, we first generate the midpoint m uniformly within the domain. To compute the length of the arc, we generate another random number $0 < L < D_2 - D_1$, such that the expected value of L is L_{exp}. Any distribution that provides these limits and this expected value will do. Once L is known, the pair (m,L) define the arc produced, as above.

7.4. INTERVALS

Intervals on the number line correspond to line segments in the plane, and are therefore considered as a stepping stone. The domain is non-circular. In this section we will develop algorithms for constructing intervals with a uniform probability distribution or with a uniform coverage

probability. We will also see why it is not possible to construct an algorithm that can produce any possible interval of a given fixed length in a given domain while providing a uniform interval coverage probability.

7.4.1. PROBLEM DEFINITION

We require that the interval I lie in some domain D, so $I \subseteq D$ (note that we do not require the endpoints of I to lie in D for open intervals). The domain may be any interval $D = [D_1,D_2]$, $[D_1,D_2)$, $(D_1,D_2]$, or (D_1,D_2). The expected length of the interval L_{exp} or the actual length of the interval L may be given.

7.4.2. MAPPING INTERVALS TO POINTS IN THE PLANE

Since intervals and points are both described by ordered pairs of numbers, it is natural to define a 1-to-1 mapping between points in $\Re^2$ and intervals as follows (see Figure 7.1).

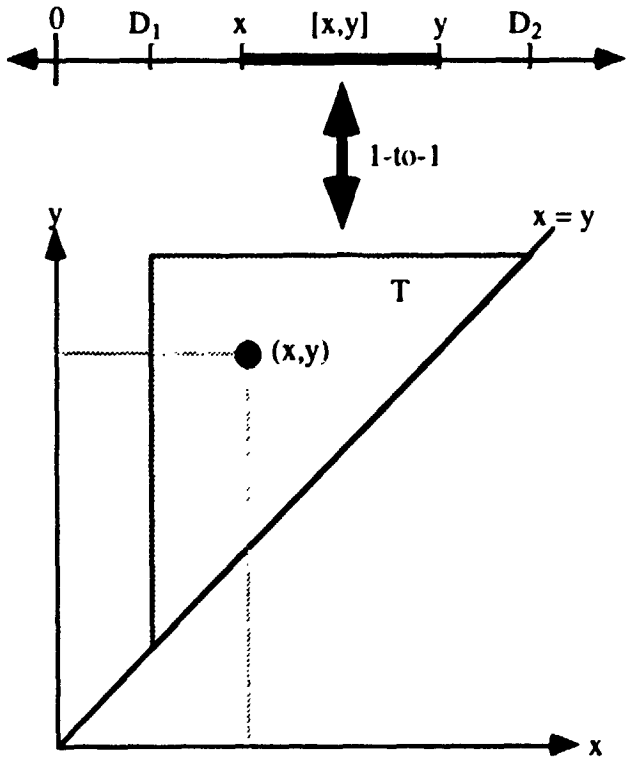


Figure 7.1. The 1-to-1 mapping between intervals and points in the triangle T.

We will find that working with points rather than intervals is a powerful technique. An interval [x,y] is mapped to the point (x,y) in the plane. Since $x \leq y$, we know that the point lies above the line $x = y$ (assuming a non-negative domain). Since $x,y \in D$, we know that the point lies in the square defined by $\{(x,y) \mid x,y \in D\}$. Therefore, the domain D maps to a triangle T defined by $T = \{(x,y) \mid x,y \in D \text{ and } x \leq y\}$. Since the mapping is 1-to-1, we may use point and interval terminology interchangeably without explicitly describing the necessary conversions.

7.4.3. DEFINITION OF UNIFORMITY

This definition can be generalized to higher dimensional domains, such as rectilinear rectangles in $\Re^2$. The elementary events are intervals that lie in the given domain D. Intervals may be open, semi-open, or closed. We wish to achieve invariance under Euclidean transformations. We therefore base our event space on the event space for points in the plane, as discussed in Section 6.1.2:

Definition: A set of intervals is an **event** if the set of points corresponding to the intervals is Lebesgue-measurable, as defined in Section 2.1.5.

Definition: Let E be any event and let F be the set of points corresponding to E. Then Pr_u is **uniform** if $Pr_u(E) = \frac{I_L(F)}{I_L(T)}$ for all events $E \subseteq D$.

Theorem: This definition of uniformity for intervals is a valid probability distribution.

Proof:

The set of events is closed under the following operations: union of a countable number of events, intersection of a countable number of events, and complement of an event. Since the event space is exactly the same as for points (see Section 6.1.2, page 48), the proof is not repeated here.

The probability axioms hold:

Axiom 1: $Pr_u(E) \geq 0$. Holds, since the area function is non-negative, and since $l_L(T)$ is non-zero.

Axiom 2: The certain event Ω consists of the triangle T. $Pr_u(\Omega) = 1$ since $Pr_u(\Omega) = \frac{I_L(\Omega)}{I_L(T)} = \frac{I_L(T)}{I_L(T)} = 1$.

Axiom 3: $Pr\left(\bigcup_{i=1}^{\infty} E_i\right) = \sum_{i=1}^{\infty} Pr(E_i)$ for any countable set of pairwise mutually exclusive events $\{E_1, E_2, \ldots\}$. Let F_i be the set of points that corresponds to E_i for all $i \geq 1$. Since the mapping from intervals to points is 1-to-1, we know that F_i and F_j are mutually exclusive if $i \neq j$. By Property 4, we know that $I_L\left(\bigcup_{i=1}^{\infty} F_i\right) = \sum_{i=1}^{\infty} I_L(F_i)$. Therefore:

$$Pr\left(\bigcup_{i=1}^{\infty} E_i\right) = \frac{I_L\left(\bigcup_{i=1}^{\infty} F_i\right)}{I_L(T)} = \frac{\sum_{i=1}^{\infty} I_L(F_i)}{I_L(T)} = \sum_{i=1}^{\infty} \frac{I_L(F_i)}{I_L(T)} = \sum_{i=1}^{\infty} Pr(E_i). \square$$

Theorem: This definition of uniformity for intervals is invariant under Euclidean transformations.

Proof:

Euclidean transformations in $\Re^1$ are simply translations and reflections. Let E be any event and let E' be the result of applying a Euclidean transformation to E. Further suppose that E' is an event. Let F and F' be the sets of points corresponding to E and E' respectively. F and F' are both Lebesgue-measurable by our definition of events. Given our mapping from intervals to points, it is clear that F' is the result of applying a Euclidean transformation on F. Specifically, the transformation consists of a translation along the line $x = y$ and/or a reflection. Since Lebesgue-measure is invariant under Euclidean transformations (Property 5), $I_L(F) = I_L(F')$. So,

$$Pr_u(E) = \frac{I_L(F)}{I_L(D)} = \frac{I_L(F')}{I_L(D)} = Pr_u(E'). \square$$

7.4.4. ALGORITHM 7.5

Input: A domain $D = [D_1,D_2]$.

Output: A closed interval in D.

Properties: Complete, uniform probability distribution, invariant under Euclidean transformations.

Complexity: O(1), may not terminate if rejection method is used.

To generate a random interval I, we simply generate two independent uniform random numbers $x,y \in D$. We return $I = [x,y]$ if $x \leq y$ and we return $I = [y,x]$ if $x > y$. Note that we could equally well use the rejection method (see Section 5.3), choosing to generate a new x,y pair if $x > y$.

The numbers x and y map to a point in the square $\{(x,y) \mid x,y \in D\}$. This point may lie in the upper or lower triangle. If the point lies in the upper triangle (i.e., if $x \leq y$), we can return it directly, so we generate the interval $I = [x,y]$. If the point lies in the lower triangle (i.e., if $x > y$), the point does not correspond to a valid interval. However, if we reflect the point about the $x = y$ diagonal, we get the point (y,x) which does correspond to a valid interval, so we generate $I = [y,x]$.

Since the numbers x and y are not required to be distinct, it is possible that $x = y$. In this case, the interval $I = [x,y]$ maps to a point on the $x = y$ diagonal. Such intervals are only represented by the one point [x,x] while all other intervals are represented by the two points [x,y] and [y,x].

PROOF OF CORRECTNESS

We must show that the generator $G_{7.5}$ implementing Algorithm 7.5 is complete and provides a uniform distribution.

Theorem: $G_{7.5}$ is complete.

Proof:

The sample space Ω is the set of all closed intervals in D. Suppose we are given an arbitrary interval $J = [J_x,J_y] \subseteq D$. We now prove that our algorithm can generate this

interval. Since $J_x \in D$ and $J_y \in D$, it is possible for the random number generator to choose $I_x=J_x$ and to choose $I_y=J_y$. The interval generated would then be $I = [I_x,I_y] = [J_x,J_y] = J$. Therefore Algorithm 7.5 can generate any interval in Ω.□

Theorem: $G_{7.5}$ generates intervals with a uniform distribution.

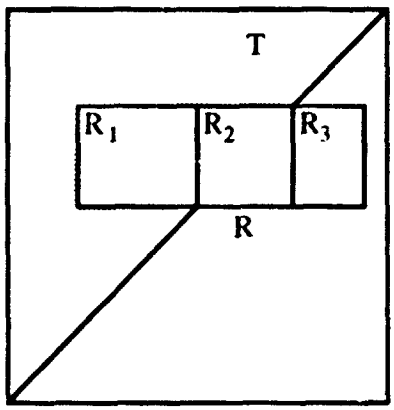
Proof:

We consider only events which correspond to axis parallel rectangles intersected with T. Let $R = \{(x,y) \mid x \in [R_L,R_R] \text{ and } y \in [R_B,R_T]\}$, where $R_L,R_R,R_B,R_T \in \Re$, $R_L \leq R_R$, $R_B \leq R_T$ be an arbitrary axis parallel rectangle in the square $\{(x,y) \mid x,y \in D\}$. Let E be the event corresponding to the set of points in $R \cap T$. We may assume one of the following properties of R is satisfied without loss of generality:

Property 1: $R_L = R_B$ and $R_R = R_T$

or

Property 2: $[R_L,R_R] \cap [R_B,R_T] = \emptyset$.



Rectangle R is partitioned into 3 rectangles R_1, R_2, and R_3.

R_2 satisfies property 1 while R_1 and R_3 satisfy property 2.

Figure 7.2. Partitioning a rectangle into rectangles satisfying property 1 or 2.

If the rectangle R does not satisfy either of these properties, we can partition the rectangle into 3 or fewer rectangles, each of which satisfies one of these properties (see Figure 7.2). Axiom 3 implies that the probability of the union of the rectangles forming this partition is the sum of the probabilities of the rectangles.

Suppose R satisfies property 1: R is therefore a square whose lower-left and upper-right corners lie on the line $x = y$. An interval corresponds to a point in this square if both of its endpoints lie in the same interval. Our algorithm will produce such an interval with probability $\frac{(R_R - R_L)^2}{(D_2 - D_1)^2}$ since x and y are chosen independently with uniform distribution, and since the order in which the endpoints are generated has no effect. The point corresponding to the interval constructed will lie in R if and only if $x,y \in [R_L,R_R]$. Since R satisfies property 1, we know that $I_L(R \cap T) = \frac{(R_R - R_L)^2}{2}$. The Lebesgue-measure of the domain triangle T is: $I_L(T) = \frac{(D_2 - D_1)^2}{2}$. Therefore our algorithm will produce an interval corresponding to a point in $R \cap T$ with probability $\frac{I_L(R \cap T)}{I_L(T)} = Pr_u(E)$.

Suppose R satisfies property 2: R is therefore a rectangle that does not intersect the line $x = y$. There are two possible sub-cases: either $R_R < R_B$ or $R_T < R_L$. We will consider each of these cases in turn.

Further suppose that $R_R < R_B$: Clearly, $R \subseteq T$. An interval I corresponds to a point in this square if one of $I_x \in [R_L,R_R]$ and $I_y \in [R_B,R_T]$. Our algorithm will produce such an interval if either:

$$x \in [R_L,R_R] \text{ and } y \in [R_B,R_T]$$

or:

$$x \in [R_B,R_T] \text{ and } y \in [R_L,R_R].$$

Since $[R_L,R_R] \cap [R_B,R_T] = \varnothing$, it is not possible for both of these conditions to be satisfied by any pair of numbers x,y. Therefore we can add the probabilities for these two

possibilities. Since x and y are chosen with a uniform distribution, our algorithm will produce an interval corresponding to a point in R with probability:

$$\left(\frac{(R_R - R_L)}{(D_2 - D_1)}\right)\left(\frac{(R_T - R_B)}{(D_2 - D_1)}\right) + \left(\frac{(R_T - R_B)}{(D_2 - D_1)}\right)\left(\frac{(R_R - R_L)}{(D_2 - D_1)}\right)$$
$$= \frac{2I_L(R)}{(D_2 - D_1)^2} = \frac{2I_L(R)}{2I_L(T)} = \frac{I_L(R)}{I_L(T)} = \frac{I_L(R \cap T)}{I_L(T)} = Pr_u(E).$$

Further suppose that $R_T < R_L$: Clearly, $R \cap T = \emptyset$. Therefore points in R do not correspond to intervals, since $I = [I_x, I_y]$ is an interval if and only if $I_x \leq I_y$. Therefore E is the null event. Our generator can only produce intervals because it sorts the numbers it uses to construct the interval. Therefore our algorithm will produce an interval corresponding to a point in R with probability $0 = \frac{I_L(R \cap T)}{I_L(T)} = Pr_u(E)$.

We have shown that our generator will produce an interval corresponding to a point in R with probability $Pr_u(E)$ in all cases. Therefore $G_{7.5}$ provides a uniform distribution.□

Clearly the probability of being covered by an interval is very low for values near D_1 or D_2, while the midpoint of the domain $D_{mid} = \frac{D_1 + D_2}{2}$ has the highest probability of being covered by an interval. In other words $Pr_c(D_1) < Pr_c(D_{mid})$. Therefore, while this algorithm does provide a uniform distribution, it does not provide a uniform coverage probability.

7.4.5. ALGORITHM 7.6

Input: A domain $D = [D_1, D_2)$, the fixed interval coverage probability $p = \frac{1}{N}$ for some integer $N \geq 3$.

Output: An interval in D of the form [a,b) other than the interval D.

Properties: Complete, uniform coverage probability = p.

Complexity: O(N), where $N = \frac{1}{p}$, may not terminate.

This algorithm was inspired by an alternative approach jointly obtained by Devroye, Sack, and Epstein [Devroye et al. 91]. In the case of arcs, given an arc, we can form a partition of the domain using the given arc and one other arc. We could therefore use a fair coin to select an arc

from the partition, achieving a uniform coverage probability of $\frac{1}{2}$. In the case of intervals, such a partition of the domain must, in general, include the given interval as well as two other intervals. We must therefore accept a coverage probability of $\frac{1}{3}$ or less if we want a simple algorithm.

Generate N - 1 distinct random numbers $r_1, r_2, \ldots, r_{N-1}$ such that $D_1 < r_1 < r_2 < \ldots < r_{N-1} < D_2$ (in sorted order). The distribution used to generate these numbers is not required to be, but probably should be uniform. The most straightforward way to generate this sequence is to generate N - 1 distinct uniform random numbers and then sort them. Another approach, which requires no sorting, is to use the algorithm of [Devroye 86, page 215] for generating uniform order statistics via uniform or exponential spacings.

Let $r_0 = D_1$ and $r_N = D_2$. Generate a single uniform random integer i such that $1 \le i \le N$. Return the interval $[r_{i-1}, r_i)$.

PROOF OF CORRECTNESS

We must show that the generator $G_{7.6}$ implementing Algorithm 7.6 is complete and provides a uniform coverage probability, and that Algorithm 7.6 has O(N) time complexity.

Theorem: $G_{7.6}$ provides a uniform coverage probability.

Proof:

Let x be an arbitrarily chosen position within the domain. We wish to compute the probability $Pr_c(x)$ of covering x by an interval generated by $G_{7.6}$. We assume the algorithm has already generated $r_1, r_2, \ldots, r_{N-1}$. The intervals $[r_{i-1}, r_i)$ for i = 1 to N form a partition of D (see Section 5.4, page 44).

We can find the unique integer j such that $x \in [r_{j-1}, r_j)$. The probability that the algorithm will generate $[r_{j-1}, r_j)$ is precisely the probability that the algorithm selects an i such that i = j. Since j is one of N possible choices, and since i is chosen with a uniform distribution, $Pr_c(x) = \frac{1}{N}$ for all x in the domain.

In either case, we have the same fixed value for $Pr_c(x)$ independent of x.□

Theorem: $G_{7.6}$ is complete.

Proof:

The sample space Ω is the set of all intervals of the form [s,e) in D other than D itself. Let $[s,e) \subseteq D$ represent an arbitrary interval from s to e such that $s \neq D_1$ or $e \neq D_2$. We will show that such an interval may be produced by the algorithm. When selecting the random numbers r_i, for $1 \leq i \leq N-1$, it is possible that $r_{j-1} = s$ and $r_j = e$, for some $0 \leq j \leq N$. It is possible that the algorithm selects i = j, thereby producing the interval [s,e).□

Theorem: Algorithm 7.6 has time complexity O(N).

Proof:

Algorithm 7.6 generates O(N) distinct uniform random numbers. This takes O(N) time most of the time. These numbers can be generated in sorted order, so an O(N log N) sorting step is avoided. Selecting a random index and constructing the interval takes O(1) time, so the total time complexity is O(N).□

EXTENSIONS

If we generate the integer i before generating the list of random numbers $r_1,r_2,\ldots,r_{N-1}$, we need not generate the entire list. We can improve the algorithm by generating only r_{i-1} and r_i. This can be done using the algorithm from [Ramberg & Tadikamalla 78] for generating subsets of uniform order statistics. With this algorithm, we can generate r_{i-1} and r_i without computing any other values in the list. We first compute r_i using a beta distribution and then compute r_{i-1} via an exponential spacing. This algorithm still requires the generation of N uniform random numbers to construct the beta distribution.

By randomly choosing one of two values N_0 and $N_1 = N_0 + 1$ for N, we can get any coverage probability p between $\frac{1}{N_0}$ and $\frac{1}{N_1}$. Let $p = \frac{1}{N_p}$, where N_p is a real number between N_0 and N_1.

We generate a single uniform random number q, and if $q > N_p - N_0$ then we let $N = N_0$. Otherwise we let $N = N_1$. This extension allows the algorithm to accept any real $p \leq \frac{1}{3}$. As well, we can randomly choose to sometimes return the interval D rather than compute the interval as above. This allows p to be greater than $\frac{1}{3}$. Given both of these extensions, the algorithm supports any coverage probability $p \in (0,1)$.

7.4.6. AN INTERESTING NON-EXISTENCE PROOF

In this section, we will show that it is not possible to construct an algorithm that can produce any possible interval of a given fixed length in a given domain while providing a uniform interval coverage probability. Devroye, Sack, and Epstein jointly developed this proof [Devroye et al. 91].

AN ALGORITHM THAT CANNOT GENERATE ALL INTERVALS OF THE GIVEN LENGTH

Before giving this proof, we will briefly describe an algorithm meeting all the above requirements other than that the algorithm be complete. We do this to show that this requirement is necessary for the proof to hold. Define D to be the size of the domain ($D = D_2 - D_1$). Let the length of the intervals produced be $L = \frac{D}{n}$, where n is a positive integer. If we partition the domain into n equal length intervals i_1 to i_n, each of these intervals will have length L. We define our domain as well as our intervals to be inclusive of their start and exclusive of their end so that, for any given position x within the domain, precisely one of these intervals will cover x. If we therefore select a random index j between 1 and n with a uniform distribution, and make our generator return the interval i_j, we will provide a uniform interval coverage probability, since the probability of covering any x is $\frac{1}{n}$. Unless n = 1, there are an infinite number of possible intervals of length L, while this algorithm is only capable of producing n distinct intervals.

AN EXAMPLE

Before giving the formal proof, we give a brief example. Let the domain be the numbers between zero and one, inclusive of both. Let the desired interval length be $\frac{1}{2}$. Any interval of that length in that domain must cover $x = \frac{1}{2}$. Therefore the interval coverage probability at $\frac{1}{2}$ must be 1. Since

the distribution must be uniform, this implies all positions in the domain must be covered by all intervals produced. The only way that this can be achieved is if the only interval produced is the full interval [0,1], but the full interval does not have the desired length of $\frac{1}{2}$. We have a contradiction. Therefore it is not possible to provide a uniform interval coverage probability while producing all possible intervals of a given fixed length.

THE PROOF

Theorem: There does not exist a random interval generator capable of producing all possible intervals in a given domain D that have a given length L while providing a uniform coverage probability.

Proof: (by contradiction)

We assume the existence of a random interval generator G capable of producing all possible intervals in a given domain D that have a given length L while providing a uniform coverage probability. Define D_1 and D_2 such that $D = [D_1, D_2]$. Define f(x,y) as the probability that G will generate the interval from x to y. The function f(x,y) is defined for all $x,y \in D$ such that $x \leq y$. Define

$$F(t) = \int_{D_1}^{t} \int_{t}^{D_2} f(x,y)\,dy\,dx.$$

F(t) is therefore the probability that G will generate an interval with start position $s \leq t$ and end position $e \geq t$. In other words, F(t) is the probability that G will generate an interval that covers t. We can therefore characterize uniform coverage probability as F(t) being constant, independent of t. In other words, $F(t) = p$ for all $t \in D$, for some fixed positive number p. Intervals (s,e) with fixed length L can be characterized by $e - s = L$. Since we require all the possible intervals in D of length L to be capable of being produced by G, $f(x,y) > 0$ for all $x,y \in D$ satisfying $y - x = L$. Since we require that no intervals of length other than L be produced by G, $f(x,y) = 0$ for all x,y satisfying $y - x \neq L$.

Let $t_1 = (D_2 - D_1) - L$. Let t_2 be chosen arbitrarily such that $t_1 < t_2 < D_2$. $F(t_1) = A + B$ where A and B are defined as:

$$A = \int_{D_1}^{t_1}\int_{t_1}^{t_2} f(x,y)dydx, \text{ and } B = \int_{D_1}^{t_1}\int_{t_2}^{D_2} f(x,y)dydx.$$

$F(t_2) = B + C$ where B is defined above and:

$$C = \int_{t_1}^{t_2}\int_{t_2}^{D_2} f(x,y)dydx.$$

Since we require F(t) to be constant independent of t, we know $F(t_1) = F(t_2)$. So $A + B = B + C$, or equivalently, $A = C$. Since no pair (x,y) exists that satisfies $y - x = L$ and $t_1 \le x \le t_2$ and $t_2 \le y \le D_2$, we know that $C = 0$. Therefore $A = 0$ as well. Since at least one pair (x,y) exists that satisfies $y - x = L$ and $D_1 \le x \le t_1$ and $t_1 \le y \le t_2$, we know that $A > 0$. We have a contradiction. Therefore no such generator G exists.□

7.5. SIMPLE ARC SETS

Before considering the problem of generating simple interval sets, we consider the corresponding problem in a circular domain, which is the generation of a set of non-intersecting arcs.

7.5.1. PROBLEM DEFINITION

Arcs may not even intersect at their endpoints. Arc sets (simple or otherwise) may not contain intervals of zero length.

7.5.2. ALGORITHM 7.7

Input: A circular domain $D = [D_1,D_2)$, size n of arc set.

Output: A simple arc set of size n in D whose arcs are of the form [a,b).

Properties: Complete, uniform coverage probability = $\frac{1}{2}$.

Complexity: O(n log n), may not terminate, improved to O(n) incremental.

We start by generating the set $R = \{r_1,r_2,r_3,\ldots,r_{2n}\}$ of endpoints of the arcs in the set. Each endpoint is a distinct number in D. The distribution used to generate the endpoints is not

important, so we use the uniform distribution. We therefore generate 2n distinct numbers in D and sort them to get $R = \{r_1, r_2, r_3, \ldots, r_{2n}\}$, where $r_i \in D$ for all $i \in [1,2n]$, and $r_i < r_j$ for all $i < j$.

There are only two simple arc sets with any given set of endpoints R. Specifically, the possible arc sets are $S_1 = \{[r_1,r_2),[r_3,r_4),\ldots,[r_{2n-1},r_{2n})\}$ and $S_2 = \{[r_2,r_3),[r_4,r_5),\ldots,[r_{2n-2},r_{2n-1}),[r_{2n},r_1)\}$. S_1 and S_2 form a partition of D (see Section 5.4, page 44). We therefore flip a fair coin to decide which of these two sets to return.

PROOF OF CORRECTNESS

We must show that the generator $G_{7.7}$ implementing Algorithm 7.7 is complete and provides a uniform coverage probability, and that Algorithm 7.7 has O(n log n) time complexity.

Theorem: Algorithm 7.7 takes O(n log n) time.

Proof:

O(n) time is taken to generate the 2n distinct numbers. O(n log n) time is taken to sort the 2n endpoints. Flipping a fair coin takes O(1) time, so the algorithm is dominated by sorting, and has O(n log n) time complexity.□

Theorem: $G_{7.7}$ is complete.

Proof:

The sample space Ω is the set of all simple arc sets in D of size n in which arcs are of the form [a,b). Any simple arc set in Ω can be expressed as either $S_1 = \{[x_1,x_2),[x_3,x_4),\ldots,[x_{2n-1},x_{2n})\}$ or $S_2 = \{[x_2,x_3),[x_4,x_5),\ldots,[x_{2n-2},x_{2n-1}),[x_{2n},x_1)\}$, where $x_i < x_j$ for $i < j$ and $x_i \in D$ for all $i \in [1,2n]$. We will show that the algorithm can generate S_1 and S_2. The algorithm could generate the set $R = \{x_1,x_2,x_3,\ldots,x_{2n}\}$. The algorithm could then flip the coin such that it chooses to return either S_1 or S_2.□

Theorem: $G_{7.7}$ provides a uniform coverage probability.

Proof:

Since we define our arcs to be inclusive of one endpoint and exclusive of the other, S_1 and S_2 form a partition of D (see Section 5.4, page 44). Let x be an arbitrarily chosen position within the domain. We wish to compute the probability $Pr_c(x)$ of covering x by an arc set generated using the this algorithm. Regardless of the value of x, it will be covered for precisely one of the two fair coin flip outcomes. Therefore $Pr_c(x) = \frac{1}{2}$ for all x in the domain.□

EXTENSIONS

Suppose we wanted to generate the arc set incrementally, in order through the domain. The above approach requires sorting, so the arcs are not really known until all 2n random numbers are generated. Since the numbers are just independent uniformly random numbers that have been sorted, we are really generating **uniform order statistics**, which can be done via exponential spacings [Devroye 86, page 214] in O(n) time. To explain the technique, we will consider the problem of generating n uniform random numbers $\{r_1, r_2, \ldots r_n\}$ in [0,1] in sorted order. We generate the numbers in reverse order:

LET $r_{n+1} := 1$.

FOR j := n DOWN TO 1 DO

LET r := a uniform random number in [0,1].

LET $r_j := \left(r^{1/j}\right)\left(r_{j+1}\right)$.

END FOR.

This algorithm takes O(1) time to generate each number, for a total time complexity of O(n) in the worst case. Since the coin flip to determine which way to connect the endpoints is independent of the endpoints themselves, we can flip this coin first, allowing the intervals to be produced incrementally in O(1) time, as the random numbers are produced in sorted order in O(1) time.

7.6. SIMPLE INTERVAL SETS

Before considering the problem of generating simple line segment sets on the plane, we consider the corresponding problem on the number line, which is the generation of a set of non-intersecting intervals. The domain is non-circular.

7.6.1. PROBLEM DEFINITION

Intervals may not even intersect at their endpoints. Interval sets (simple or otherwise) may not contain intervals of zero length. The size of the interval set is given. See [Kendall & Moran 63, pp. 36-37] for a treatment of simple interval sets in which all intervals are constrained to have the same constant length. The following treatment assumes intervals may have arbitrary random lengths subject to the constraint that the intervals lie within the domain and do not overlap.

7.6.2. ALGORITHM 7.8

Input: A domain $D = [D_1,D_2)$, *approximate* size n of interval set.

Output: A simple interval set in D of size n or n + 1 with intervals of the form [a,b).

Properties: Over-complete for intervals of size m if the algorithm is executed for both n = m and n = m - 1, uniform coverage probability = $\frac{1}{2}$.

Complexity: O(n log n), may not terminate, improved to O(n) incremental.

We can adapt our simple arc set generator Algorithm 7.7 to generate interval sets by defining a mapping between arc sets and interval sets. The only difficulty is in mapping an arc that *wraps around* the ends of the domain. We map such an arc to two intervals. Specifically, we map the arc [a,b) where a > b to the intervals $[D_1,b)$ and $[a,D_2)$. If we were to always generate 2n distinct numbers as we did for arc sets, almost all sets containing an interval starting at D_1 would also include an interval ending at D_2. To avoid this, we generate 2n - 1 numbers half of the time, by flipping a fair coin (call it coin A).

If coin A is heads, we generate 2n independently uniform distinct numbers $R = \{r_1,r_2,r_3,\ldots,r_{2n}\}$, where $r_i \in (D_1,D_2)$ for all $i \in [1,2n]$, and $r_i < r_j$ for all $i < j$, and then flip a fair coin B to decide whether to include neither or both endpoints of the domain. We return either $S_1 = \{[r_1,r_2),[r_3,r_4),\ldots,[r_{2n-1},r_{2n})\}$ or $S_2 = \{(D_1,r_1),[r_2,r_3),[r_4,r_5),\ldots,[r_{2n-2},r_{2n-1}),[r_{2n},D_2)\}$.

If coin A is tails, we generate 2n - 1 independently uniform distinct numbers $R = \{r_1,r_2,r_3,\ldots, r_{2n-1}\}$, where $r_i \in (D_1,D_2)$ for all $i \in [1,2n - 1]$, and $r_i < r_j$ for all $i < j$, and then flip a fair coin B to decide which endpoint of the domain to include. We return either $S_3 = \{[r_1,r_2),[r_3,r_4),\ldots, [r_{2n-1},D_2)\}$ or $S_4 = \{(D_1,r_1),[r_2,r_3),[r_4,r_5),\ldots,[r_{2n-2},r_{2n-1})\}$.

PROOF OF CORRECTNESS

We must show that the generator $G_{7.8}$ implementing Algorithm 7.8 is complete and provides a uniform coverage probability, and that Algorithm 7.8 has O(n log n) time complexity.

Theorem: $G_{7.8}$ provides a uniform coverage probability.

Proof:

Since we define our arcs to be inclusive of one endpoint and exclusive of the other, S_1 and S_2 form a partition of D (see Section 5.4, page 44). By the same argument, S_3 and S_4 form a partition of D. Any given point x in the domain will be covered by precisely one of S_1 and S_2, and will be covered by precisely one of S_3 and S_4. Since coin B is a fair coin, $Pr_C(x) = \frac{1}{2}$ for all $x \in D$.□

Theorem: $G_{7.8}$ is over-complete for intervals of size m if it is executed for both n = m and n = m - 1.

Proof:

Let S be an arbitrary interval set of size m. The four cases are as follows:

Case 1: If S contains an interval starting at D_1 and an interval ending at D_2, then S can be generated if the first coin flip selects an even number of random numbers and the second coin flip selects inclusion of D_1 and D_2.

Case 2: If S contains no interval start at D_1 or ending at D_2, then S can be generated if the first coin flip selects an even number of random numbers and the second coin flip selects exclusion of D_1 and D_2.

Case 3: If S contains an interval starting at D_1 but does not contain an interval ending at D_2, then S can be generated if the first coin flip selects an odd number of random numbers and the second flip selects inclusion of D_1.

Case 4: Correspondingly, if S contains an interval ending at D_2 but does not contain an interval starting at D_1, then S can be generated if the first coin flip selects an odd number of random numbers and the second flip selects inclusion of D_2.

In all four of these cases, the remaining endpoint set of S, excluding D_1 and D_2, forms a sorted list of distinct numbers within the domain, exclusive of D_1 and D_2. This list may be produced for either n = m or n = m - 1, so any simple interval set may be produced by $G_{7.8}$.□

Theorem: Algorithm 7.8 takes O(n log n) time.

Proof:

O(n) time is taken to generate the approximately 2n distinct numbers. O(n log n) time is taken to sort these numbers. Flipping two fair coins takes O(1) time, so the algorithm is dominated by sorting, and has O(n log n) time complexity.□

EXTENSIONS

As for arc sets, we can generate the intervals incrementally in O(1) time. We flip both coins first, then generate the appropriate number of numbers in sorted order as uniform order statistics.

Chapter 8

8. LINE SEGMENT SETS

This and all remaining chapters will consider generation of geometric objects in the plane ($\mathbf{R}^2$). In this chapter we consider the generation of sets of line segments.

8.1. SIMPLE LINE SEGMENT SETS IN A RECTANGLE

The difficulty in constructing a simple line segment set is in ensuring simplicity, which is defined as follows:

Definition: A line segment set is **simple** if no two line segments in the set intersect, where line segments intersect if they share any point, including their endpoints.

8.1.1. ISOLATION OF POINTS BY LINE SEGMENT SETS

In the algorithms given below, we will consider the number of points in a point set that may be wasted when forming line segments whose endpoints are members of this set. Some points may become isolated, being unusable for forming line segments.

Definition: A point p in a point set P is **isolated** by a line segment set L if no line segment exists whose endpoints are p and $q \in P$ that does not intersect any segment in L.

We will see that we can bound the number of points that will be wasted, so we can build our point set large enough to guarantee that we can construct a line segment set of a given size before exhausting our supply of points. Consider the following related problem: Given a point set, determine the number of non-intersecting line segments needed to make the points pair-wise invisible.

Theorem: Any set of m points can be isolated with m - 1 non-intersecting line segments.

Proof: (constructive)

If there is only one point in the set, then no line segments are needed, and so our result trivially holds. If there are two points in the set, then we can isolate one from the other using a line segment of arbitrarily small length by placing it on the perpendicular bisector. Again, our result holds trivially.

If there are more than two points in the point set, we can isolate any one point on the convex hull from all other points in the point set using only one line segment by placing the line segment very close to the point. To find such a point, we construct the convex hull of the point set [Preparata & Shamos 85] and select any hull vertex. All that is left is to isolate the remaining points from one another. We have reduced the size of the problem by one and we have used only one line segment to do so. Therefore, by repeated application of the above procedure, we will eventually have only two points left (which will require only one line segment to isolate), and we will have completed the isolation of m points using only m - 1 line segments.□

Theorem: Some set of m points requires m - 1 non-intersecting line segments to be isolated.

Proof: (constructive)

Let the set of m points be collinear. In this case, one line segment is needed between each adjacent pair of points. Therefore m - 1 line segments are needed in total.□

Conjecture: Any set of m non-intersecting line segments can isolate at most m + 1 points.

A corollary would be that, if we start with 2n + 1 points, we will construct n line segments without exhausting our supply of non-isolated points.

8.1.2. AVOIDING COLLINEARITIES

We would like to avoid wasting points due to their being on the segment formed between two other points. The obvious way to do this is to require the point set to be in standard form (i.e. no three points collinear). Constructing a random point set with this property is straightforward. As points are added to the set, they are tested for collinearity with all existing pairs of points in the set. If any collinearity is found, the point is not added. If we do not place this requirement on our point set, then there is no bound on the number of points that can be wasted. It is theoretically possible for all the points in the random set to be collinear. If the pair chosen for the initial line segment are the extremal ones, all the remaining points will be wasted, and only one line segment will be produced.

There is another problem with avoiding all collinearities in the point set: we do not want the resulting line segment set to have any special properties. If we have no collinearities in the point set, the resulting line segment set will never have any endpoints anywhere on the line through any segment. For this reason, it is better to allow collinearities in the point set and handle the problems later when constructing the line segment set. Since we cannot put a bound on the number of points wasted, we will have to live with the possibility that we will run out of points. However, the probability of points being collinear is zero for a set of independently uniformly distributed points, so the expected number of points wasted is zero.

8.1.3. ALGORITHM 8.1

Input: A rectangle D, a size n.

Output: A simple line segment set of size n in D.

Properties: Complete.

Complexity: $O(n^2 \log n)$, may not terminate.

The most straightforward technique for building a random simple line segment set is to iteratively add line segments to the set, ensuring that each new line segment does not intersect any of those

already in the set. To do this, we select a random point in the rectangle for one endpoint of the new segment. This point should be chosen so as to not lie on any of the segments in the set. We are now left with the task of selecting a random point for the other endpoint of the new segment such that the segment does not intersect any of the other segments. To do this, we compute the visibility polygon from the new point. Any point in the interior of this polygon could be used as the other endpoint. We therefore select a point randomly from within this polygon using Algorithm 6.11.

PROOF OF CORRECTNESS

Theorem: Algorithm 8.1 takes $O(n^2 \log n)$ time.

Proof:

Selecting a random source point takes O(n) time, since the probability that a uniform random point lies on a segment is zero. To compute the visibility polygon in a line segment set, we use the O(n log n) algorithm of [Lee & Chen 85]. Selecting a random destination point takes O(n) time given the visibility polygon, since the visibility polygon has O(n) vertices. We add O(n) line segments, so the total cost is $O(n^2 \log n)$.□

8.1.4. ALGORITHM 8.2

Input: A rectangle D, a size n.

Output: A simple line segment set of size n in D.

Properties: Complete.

Complexity: $O(n^2 \log n)$, may not terminate.

We generate all the endpoints of the line segments in advance, with a uniform distribution over the rectangle. Any set of an even number of points can form at least one simple line segment set. We construct line segments one at a time, using the points. To start, we select two of the points and construct the first line segment between them. Some of the remaining points may lie on this line

segment. We therefore remove such points from consideration. It is possible that some of the points are now isolated, meaning that there is no possible line segment which has the point as an endpoint and which does not intersect the existing line segments.

For the remaining line segments, we construct them one at a time as follows: We select a candidate endpoint from the remaining points, and test all other points as possible line segments. We compute the visibility polygon from the point. All other points that lie inside the visibility polygon are candidates for the other endpoint of the line segment. If no points lie in the visibility polygon, the point is isolated, so we remove it from consideration and try another random point. Otherwise, we select one of these candidate segments randomly and add it to the set of line segments. Note that some points from the original point set may not be used. This implies that we must make the point set larger than necessary so that we do not run out of points before building the desired number of line segments.

PROOF OF CORRECTNESS

Theorem: Algorithm 8.2 takes $O(n^2 \log n)$ time.

Proof:

To generate the initial random point set, we simply generate a set of O(n) points in O(n) time using Algorithm 6.12. To compute the visibility polygon in a line segment set, we use the O(n log n) algorithm of [Lee & Chen 85]. In O(n) time we then determine which points lie in the visibility polygon, and choose one to form a new line segment. The total time is therefore $O(n^2 \log n)$, since n line segments must be added, and each requires a visibility query.□

8.1.5. ALGORITHM 8.3

Input: A rectangle D, a size n.

Output: A simple line segment set of size n in D.

Properties: Complete.

Complexity: $O(n^3)$, may not terminate.

If we start with a random triangulation of a random point set, we can select a subset of the diagonals of the triangulation to form a simple line segment set. This idea is due to [Rau-Chaplin 91]. We select a random edge of the triangulation, which could be a diagonal or an edge of the convex hull of the point set. This edge is chosen as a member of the resulting line segment set S. Any other incident edges are therefore not allowed to be in S. These line segments are therefore removed from consideration, and we repeat the process, randomly selecting one of the remaining edges of the triangulation. We stop when we have the desired number of line segments in our set.

The difficulty with this algorithm lies with the fact that some vertices may be isolated. If we want n line segments, we need to start with more than 2n points, since some of the points may not be able to form line segments. A point cannot form a line segment if all its incident edges were removed. For example, consider a triangle of the triangulation with two external edges and one diagonal. If the diagonal is chosen as a member of our line segment set, clearly neither of the two incident edges on the only remaining vertex of the triangle can be used, so the vertex is isolated and therefore wasted. We saw above that we can bound the number of vertices wasted due to isolation, and can therefore specify how many points we need in our original point set in order to guarantee that we won't run out of edges before obtaining the desired number of line segments.

The only remaining difficulty with this approach is that some point sets produce degenerate triangulations. For example, a point set in which all the points lie on a line has no triangulation. We simply restart the algorithm if this occurs. Clearly, the probability of generating a degenerate point set is zero, so the expected number of rejected point sets is zero.

This algorithm is of little interest unless a more efficient point set triangulation generator is found.

PROOF OF CORRECTNESS

Theorem: Algorithm 8.3 takes $O(n^3)$ time.

Proof:

To generate a random point set in O(n) we use Algorithm 6.12. To generate a random triangulation of the point set in $O(n^3)$ time, we use Algorithm 10.3 or 10.4. In the remaining steps of the algorithm, each edge of the triangulation is processed only once. A given edge is either removed from the set because it is adjacent to an edge placed in S, or it is chosen at random to be added to S. Since there are O(n) edges in the triangulation, O(n) time is taken to construct the line segment set once the triangulated point set is known. The total time complexity is therefore dominated by the triangulation step.□

Chapter 9

9. POLYGONS

In this chapter we consider the generation of various types of polygons. We are either given a domain in which the vertices must lie or a set of points that are the vertices of all polygons produced.

9.1. STAR-SHAPED POLYGONS ON A POINT SET

Polygons generated are in standard form (i.e. all vertices distinct and no three consecutive vertices collinear).

Given a vertex set, it is straightforward to construct a random star-shaped polygon. Simply select a center point which is guaranteed to be inside any star-shaped polygon formed from the point set. In other words, select a random point within the convex hull of the point set as the center point. Sort the points by their angle with respect to the center point. If 2 points have the same angle, select a random order for these points. If more than 2 points have the same angle, all but the extreme points are not used. This sorted order defines a star-shaped polygon. Note that the center point is not included as one of the points in the polygon.

To avoid collinearities, it is necessary to remove such vertices after constructing the polygon. This eliminates the ability to specify exactly how many vertices the resulting polygon will have. In fact, if all the points in the given point set lie on a line, then it is clearly impossible to construct a polygon from these points.

9.2. STAR-SHAPED POLYGONS IN A RECTANGLE OR CIRCLE

Construct a random point set in the domain, then apply the above algorithm to generate a star-shaped polygon with this point set as its vertex set. In this case, the point set may have three or more points that lie on a line, making the resulting polygon have fewer vertices than the point set

generated. In fact, if all the points in the random point set lie on a line, the algorithm will not be able to construct a polygon from the point set. However, the probability of this occurring is zero.

9.3. MONOTONE POLYGONS ON A POINT SET

9.3.1. PROBLEM DEFINITION

We are given a point set V and asked to generate a monotone polygon with a given direction of monotonicity θ. If no specific direction of monotonicity is desired, this direction can be chosen uniformly at random from $\theta \in [0,2\pi)$. Polygons generated are in standard form (i.e. all vertices distinct and no three consecutive vertices collinear).

9.3.2. DEFINITION OF UNIFORMITY

Since the sample space is finite, we use the discrete uniform probability distribution:

Definition: All sets of monotone polygons whose vertices are precisely the set V are **events**. A monotone polygon generator is **uniform** if each of the monotone polygons whose vertices are precisely the set V has the same probability of being generated.

9.3.3. ALGORITHM 9.1

Input: A point set V of size n.

Output: A monotone polygon whose vertices are the set V.

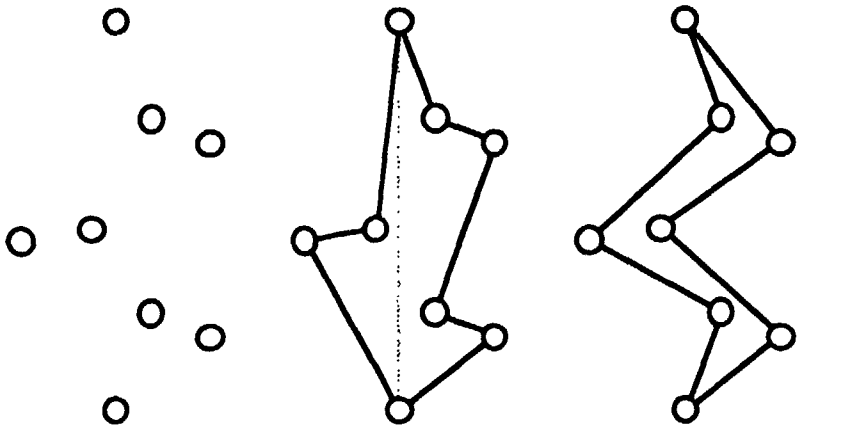
Properties: *Incomplete.*

Complexity: O(n log n).

This is the standard algorithm used to generate a simple polygon from a set of points. We will see that it is incomplete. We therefore develop a complete algorithm in Section 9.3.5 (page 97).

We first find the extreme points in V with respect to θ. Divide the remaining points (excluding the extremal points) into two groups by drawing a line between the extremal points. Each group is

then sorted with respect to the direction of monotonicity to form a chain, and the chains are connected to the extremal points to form the monotone polygon. The problem with this technique is that the polygons generated have a special property (other than being monotone). As an example, the y-monotone polygon on the right side of Figure 9.1 would not be generated.



Point set y-monotone polygon 1 y-monotone polygon 2

Polygon 1 would be generated by this algorithm.

Polygon 2 could not be generated by this algorithm.

Figure 9.1. A monotone polygon that cannot be generated by Algorithm 9.1.

Specifically, polygons generated with this algorithm have the following property: The point sets defining the left and right chains are linearly separable by the line parallel to the axis of monotonicity and through the extreme points.

9.3.4. ALGORITHM 9.2

Input: A point set V of size n, and a direction of monotonicity θ.

Output: A monotone polygon whose vertices are the set V.

Properties: Complete, uniform probability distribution.

Complexity: Preprocessing: $O(n^n)$ time and space, query: O(1) time.

This is a brute force algorithm. We construct all possible (not necessarily simple) polygons for the point-set V, select only those which are simple and θ-monotone, and finally, uniformly randomly choose one of these polygons. Testing simplicity and θ-monotonicity can be done in polynomial time [Preparata & Shamos 85].

This is very slow because there are $\frac{(n-1)!}{2}$ (not necessarily simple) polygons through n points. This quantity is derived as follows: There are n! orderings of the n points in V. Since we are interested in circular orderings, we do not care about which point we start with (a factor of n) or which direction the points are enumerated in (i.e., clockwise or counterclockwise, a factor of 2). The result is that there are $\frac{(n-1)!}{2}$ polygons.

9.3.5. ALGORITHM 9.3

Input: A point set V and a direction of monotonicity θ.

Output: A monotone polygon whose vertices are the set V.

Properties: Complete.

Complexity: $O(n^2)$.

We construct random monotone polygons by adding points to form larger and larger monotone polygons. We first find the extreme points in V with respect to θ. We select an additional random point from V to form the initial triangle. Since a triangle is always convex, it is always monotone in all directions. We then add the remaining points in V to the polygon, one at a time, while

maintaining the monotonicity and simplicity of the polygon. The order in which the remaining points are added is not important. We therefore use a uniform random order.

Each time a point is added, we consider breaking each of the chains to insert the new point. If adding the point to a chain would cause the polygon to become non-simple, then we must add the point to the other chain. If the point can be added to either chain, a fair coin is flipped to determine which chain to break.

If the point being added lies outside the polygon, then the point must be added to the chain on the appropriate side. This case can be tested for in O(n) time. If the point to be added lies inside the polygon, then we have one of the cases from Figure 9.2. To determine whether the point can be added to a left chain, check whether any of the points of the right chain lie on the left side of either of the edges that would be created if the point were added. If such a point is found, then the resulting polygon would be non-simple, so the point can not be added to the left chain. Testing whether the point can be added to the right chain is symmetrical. These tests take O(n) time.

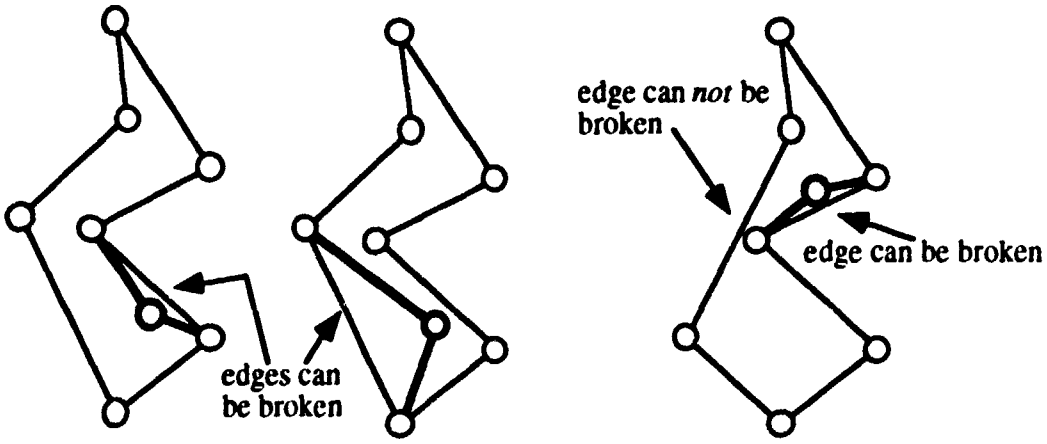


Figure 9.2. Adding a point inside a y-monotone polygon.

Each point can be added in O(n) time, so the total time to generate an n vertex θ-monotone polygon is therefore $O(n^2)$.

PROOF OF CORRECTNESS

We must show that the generator $G_{9.3}$ implementing Algorithm 9.3 is complete.

Theorem: $G_{9.3}$ is complete.

Proof:

Let Q be an arbitrary θ-monotone polygon on the point set V. We will show that the algorithm can construct the polygon P = Q. Let $Q_1 = Q$. There exists a sequence of θ-monotone simple polygons $Q_1,Q_2,Q_3,\ldots,Q_n$ in which Q_i is constructed from Q_{i-1} by removing a non-extremal vertex v_{i-1}, where the final polygon Q_n is a triangle consisting of the extremal vertices and one other point in V.

We constructively prove the existence of such a sequence as follows: The first part of the sequence is generated by repeatedly removing reflex vertices (in any order) until none remain. The extreme vertices are never reflex vertices and are therefore never removed. Removing reflex vertices always maintains simplicity and θ-monotonicity. When no reflex vertices remain, the polygon is convex. The last part of the sequence is generated by repeatedly removing non-extremal vertices (in any order) until only a triangle consisting of the extremal vertices and one other vertex remains. Removing vertices from a convex polygon always maintains convexity, and therefore also maintains simplicity and θ-monotonicity. The final polygon is a triangle satisfying the required properties. We have therefore constructed the sequence $Q_1,Q_2,Q_3,\ldots,Q_n$ as required.

This algorithm can generate the polygon Q by starting with the triangle Q_n and adding vertices $v_{n-1},v_{n-2},\ldots,v_1$ in that order to produce the intermediate polygons $Q_{n-1},Q_{n-2},\ldots,Q_1$, the final polygon being $P = Q_1 = Q$. Therefore, $G_{9.3}$ can generate any θ-monotone polygon on the point set V.□

9.4. MONOTONE POLYGONS IN A RECTANGLE OR CIRCLE

As with star-shaped polygons, we start with a random point set in the domain. We then select a random angle $\theta \in [0,2\pi)$ for the direction of monotonicity. We then apply the above algorithm to generate a monotone polygon with the given direction of monotonicity and the given vertex set.

9.5. CONVEX POLYGONS ON A POINT SET

If the point set is not convex, it is not possible to built a convex polygon with the point set as its vertex set. If the point set is convex, only one convex polygon can be produced. In fact, only one simple polygon can be produced. Therefore, this problem is uninteresting unless we allow the resulting convex polygon to use only a subset of the given point set. We could then select a random subset of the given point set with at least three points in it, and construct the convex hull of this subset.

9.6. CONVEX POLYGONS IN A RECTANGLE

9.6.1. PROBLEM DEFINITION

Polygons generated are in standard form (i.e. all vertices distinct and no three consecutive vertices collinear).

9.6.2. ALGORITHM 9.4

Input: A rectangle D, an upper bound on the polygon size n.

Output: A convex polygon in D with at most n vertices.

Properties: Complete.

Complexity: O(n log n), may not terminate.

One approach to constructing convex polygons is to take the convex hull of a random point set. Since the point set will not, in general, be convex, the size of the resulting convex polygon is not known, and depends upon the distribution of the points. This technique is capable of generating any convex polygon. It takes O(n log n) time to generate a convex polygon with at most n

vertices, since O(n log n) time is taken to find the convex hull of n points [Preparata & Shamos 85]. See [Efron 65] for results on expected values of various properties of convex polygons generated in this way. We generate the point set using Algorithm 6.12.

9.6.3. ALGORITHM 9.5

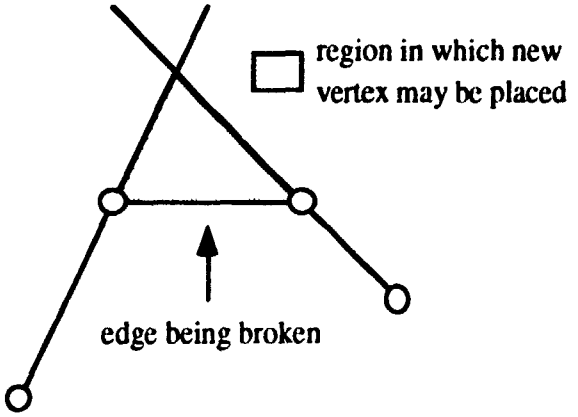
Input: A rectangle D, a polygon size n.

Output: A convex polygon in D with n vertices.

Properties: Complete.

Complexity: O(n).

In this algorithm, we repeatedly add vertices to a convex polygon to form a larger convex polygon, starting with a random triangle. To add a vertex, we randomly select an edge to break and then determine the region in which the new vertex must lie for the polygon to remain convex. We then select a random point within this region as the position of the new vertex (see Figure 9.3). This is done using Algorithm 6.10 (page 60) for generating a random point in a convex polygon.

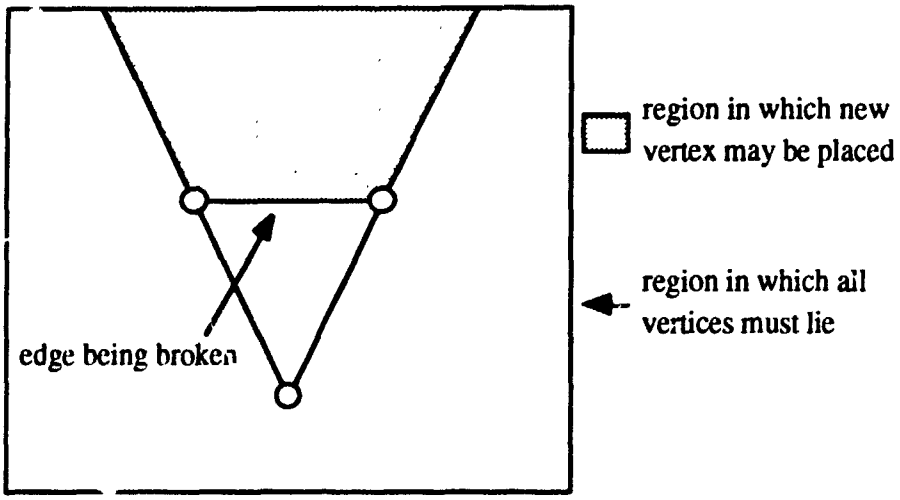


Polygon must remain convex and retain all vertices.

Figure 9.3. Adding a vertex to a convex polygon -- case 1.

Note that the region in which the new vertex may be placed could be infinite if we did not require the polygon to lie in rectangle D. Since we require the points to lie within some rectangular domain

D, our vertex placement regions are always finite (see Figure 9.4). In general, the more edges the polygon has, the more constrained the position of the new vertex becomes.



Polygon must remain convex and retain all vertices.

The region would be infinite if we did not intersect it with the domain.

Figure 9.4. Adding a vertex to a convex polygon -- case 2.

To add a vertex to the polygon, a random edge is selected in O(1) time. Determining the region in which the new vertex may be placed involves examination of only the two neighboring edges. The resulting polygon has at most 6 edges, so computing the polygon and generating a random point in the polygon takes only O(1) time. The total time to add a vertex to the polygon is therefore O(1), so the time to generate an n vertex convex polygon is $O(n^2)$.

9.6.4. ALGORITHM 9.6

Input: A rectangle D and an integer m.

Output: A convex polygon in D with at most m + 4 sides.

Properties: Over-complete for polygons of size m, uniform coverage probability.

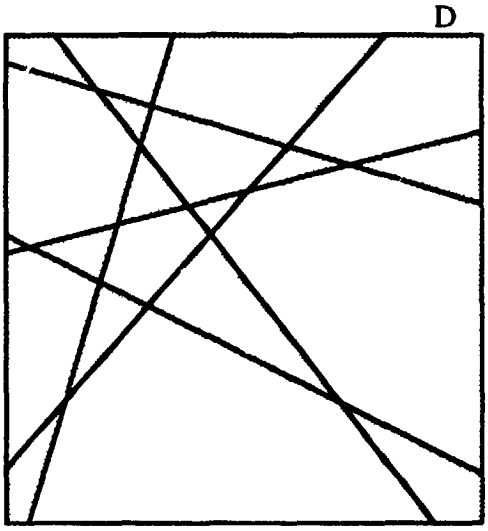
Complexity: $O(m^2)$ time and space, may not terminate.

The following is an adaptation of an algorithm by [May & Smith 82]. We assume the domain is a rectangle $D = \{(x,y) \mid x \in [D_L,D_R] \text{ and } y \in [D_B,D_T]\}$, where $D_L,D_R,D_B,D_T \in \Re$, $D_L \leq D_R$, $D_B \leq D_T$,

but the method presented here generalizes to generate convex polyhedra in n dimensions in a convex polyhedral domain.

The only input to the algorithm other than the domain D is a positive integer m. The size of the resulting polygon is between 3 and m + k, where k is the number of sides in the domain polygon D (i.e., 4 in our case). Surprisingly, although the coverage probability is constant for all points in D, the probability is not specified and is not known even once the polygon is generated.

First, we generate m random lines $L = \{L_1, L_2, \ldots, L_m\}$ in any fashion such that any line intersecting D might be created, and such that all lines in L intersect D. For example, to generate a given line L_i, we could generate two distinct points A_i and B_i independently uniformly distributed on the perimeter of D, and connect the points to form the line L_i.



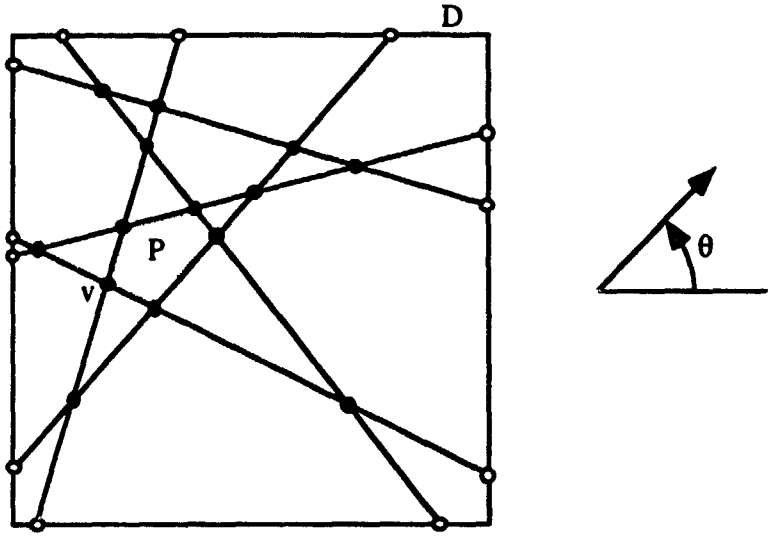
m = 6 lines, n = 21 polygons

Figure 9.5. The domain partitioned by m lines into n polygons.

Next, we construct a partitioning $\{P_1, P_2, \ldots, P_n\}$ of the domain D with the lines $L = \{L_1, L_2, \ldots, L_m\}$ (see Figure 9.5). To ensure that the polygons form a partition, we assign the points in each line L_i to the polygons on one of the sides of the line. This implies that we assign each point of intersection between every pair of lines in L to one of the adjacent triangles. The partitioning consists solely of polygons. Finally, we select one of the polygons uniformly at

random. The difficulty lies in selecting a polygon from the partition without explicitly constructing the entire partitioning.

In O(m) time, we can construct the set B of all intersections of lines in L with the boundary of D. In $O(m^2)$ time, we can find the set V of all vertices of all polygons in the partition $\{P_1,P_2,\ldots,P_n\}$ by intersecting all pairs of lines, and selecting those intersection points that lie inside D. This set will require $O(m^2)$ storage. Let $V' = V \cup B$. To select a polygon, we first find an angle $\theta \in [0,2\pi)$ which is not parallel or perpendicular to any line in L. Next, we select a vertex v uniformly at random from V'. Finally, we find the polygon P in the partition that is in direction θ from v (see Figure 9.6). There will be at most one such polygon, as we will prove in Lemma 1. If there is no polygon in that direction from that vertex (i.e. $V \in B$, and the direction θ is outside D), we reject v. We keep the same angle θ, but select another vertex v from V'.



Black dots indicate vertices in V. White dots indicate vertices in B. $v \in V \cup B$.

Figure 9.6. The polygon P in direction θ from v.

The probability that a vertex is rejected converges to zero as the number of lines m grows since the number of intersections with D is 2m, while the number of intersections among the lines is $O(m^2)$. Since rejection occurs only for some of the 2m points at which lines intersect D, and never for intersections of lines, rejection probability converges to zero.

PROOF OF CORRECTNESS

We must show that the generator $G_{9.6}$ implementing Algorithm 9.6 is complete and provides a uniform coverage probability.

Lemma 1: For any angle θ which is not parallel or perpendicular to any line in L, there is at most one polygon $P_i \in \{P_1,P_2,\ldots,P_n\}$ in direction θ, from any vertex $v \in V'$ (see Figure 9.6).

Proof:

The only directions which yield more than one polygon from a vertex $v \in V' = V \cup B$ are directions parallel or perpendicular to the lines $L = \{L_1,L_2,\ldots,L_m\}$. If direction θ points outside D from v, then there will be no polygon. If direction θ points inside D from v, then there will be exactly one polygon since the ray from v in direction θ cannot be collinear with any lines in L.□

Lemma 2: $G_{9.6}$ selects a polygon $P_i \in \{P_1,P_2,\ldots,P_n\}$ uniformly at random.

Proof:

Implied by Lemma 1. For any angle θ not parallel or perpendicular to any line in L, there is a 1-to-1 mapping between a subset of $V \cup B$ and the polygons $\{P_1,P_2,\ldots,P_n\}$.□

Theorem: $G_{9.6}$ is over-complete for polygons of size m.

Proof:

Let Q be an arbitrary polygon in D with m sides. We will prove that this algorithm can produce Q. Let $M = \{M_1,M_2,\ldots,M_m\}$ be the lines through the m sides of the polygon Q. These lines must clearly intersect D, so they must also intersect C. Our line generation algorithm can generate any such line M_i as follows. Let A'_i and B'_i be any two distinct points where M_i intersects the boundary of D. The algorithm could generate $L_i = M_i$ by

selecting $A_i = A'_i$ and $B_i = B'_i$. The algorithm could generate a line set $L \supseteq M$. So, we may assume $Q \in \{P_1,P_2,\ldots,P_n\}$.

We will now show that Q can be selected from this set of polygons. For all but a finite number of angles α, there is one vertex v_α of Q that is furthest in direction $\alpha + \pi$, where $v_\alpha \in V' = V \cup B$. Therefore, if the algorithm selects $\theta = \alpha$ to be any angle other than those perpendicular to the edges of Q, there will be one vertex v_θ, which, if selected, will cause the algorithm to return Q. Since $v_\theta \in V'$, this vertex may be selected by the algorithm, causing the algorithm to return P = Q. Therefore $G_{9.6}$ can produce Q.□

Theorem: $G_{9.6}$ provides a uniform coverage probability.

Proof:

Let p be an arbitrary point in D. We will show that $Pr_c(p) = x$ for some $x \in \Re$. The point p lies in precisely one polygon P_i of the partition $\{P_1,P_2,\ldots,P_n\}$. Since there are n polygons in the partition, and since our technique for polygon selection ensures that each is selected with the same probability (Lemma 2), the probability that P_i will be selected is $\frac{1}{n}$ (see Section 5.4, page 44). So, the probability that p will be covered is also $Pr_c(p) = \frac{1}{n}$. Since this probability is independent of the location of the point p, we have uniform coverage probability.□

9.7. SIMPLE POLYGONS ON A POINT SET

9.7.1. PROBLEM DEFINITION

Polygons generated are in standard form (i.e. all vertices distinct and no three consecutive vertices collinear). Informally, a simple polygon is one whose edges do not cross. The difficulty in constructing random polygons is in ensuring that the result is simple. We must ensure that no two edges of the resulting polygon cross. The polygons produced have the entire given point set V as their vertex set.

9.7.2. DEFINITION OF UNIFORMITY

Since the sample space is finite, we use the discrete uniform probability distribution:

Definition: All sets of simple polygons whose vertices are precisely the set V are **events.** A simple polygon generator is **uniform** if each of the simple polygons whose vertices are precisely the set V has the same probability of being generated.

9.7.3. ALGORITHM 9.7

Input: A point set V.

Output: A simple polygon whose vertices are the point set V.

Properties: Complete, uniform probability distribution.

Complexity: Preprocessing: O(n!) time and space, query: O(1).

We construct the set S of simple polygons having the vertex set V. We give the set S an arbitrary ordering, and index the elements from 1 to $|S|$. Finally, at query time, we select a uniform random integer i such that $1 \leq i \leq |S|$, and return S_i. The construction of the set S constitutes a preprocessing step which need not be repeated for each random polygon produced. Clearly the upper bound on the size of S is an exponential function of the size of V.

A brute-force approach to construct the set S is to generate all permutations P of the vertex set V which start with a particular arbitrary vertex V_0. Each permutation in P corresponds to a polygon that may or may not be simple. Two permutations in P may correspond to the same polygon, one specifying vertices in clockwise order and the other in counter-clockwise order. Let Q be the result of selecting those permutations from P which correspond to simple polygons. Each simple polygon will be represented twice in Q: once in clockwise order and once in counter-clockwise order. Let R be those permutations in Q which correspond to polygons specifying their vertices in clockwise order. Let S be the polygons corresponding to the permutations in Q.

EXTENSIONS

We now consider ways to construct S more efficiently. When generating permutations starting with V_0, if the prefix of the permutation corresponds to a non-simple chain, we can skip construction of all permutations with that prefix. Since, by induction, we already know that all but the last vertex of the prefix form a simple chain, we need only test for intersection with the final edge of the chain. When adding the last vertex, we must ensure that both edges connecting to this vertex do not intersect any of the other edges of the polygon. This technique does not improve the worst-case complexity of constructing S.

Another way to construct S is as follows: Construct the intersection graph of the point set. The intersection graph has vertices corresponding to possible edges, and edges corresponding to possible edges which cross one another (i.e., both cannot be present in the polygon). With this graph, we can quickly find all possible edges crossed by a given possible edge. Now, as we build chains, we can keep track of all edges crossed by the chain in a suffix sharing list, so that we need only consider edges that are incident on the final vertex of the chain and are not in the list of crossed edges. This approach will not improve the worst case complexity of constructing S. For example, in a convex point set, it will waste a lot of time building chains which include non-hull edges, only to find later that there is no way to extend the chain or that there is no way to link it back to the start.

9.8. SIMPLE POLYGONS IN A RECTANGLE

A great many algorithms in computational geometry take as their input a simple polygon. For example, one can triangulate a simple polygon using one of several methods, one can find the convex hull of a simple polygon, one can perform visibility queries from a point or line segment inside the polygon, etc. When testing these algorithms, it is quite useful to construct random simple polygons.

9.8.1. PROBLEM DEFINITION

Polygons generated are in standard form (i.e. all vertices distinct and no three consecutive vertices collinear). Informally, a simple polygon is one whose edges do not cross. The difficulty in constructing random polygons is in ensuring that the result is simple. We must ensure that no two edges of the resulting polygon cross.

We will describe two constructive algorithms to add a vertex to an existing simple polygon. In the first algorithm, we first select a random point which will be the new vertex, and then select an edge to break. In the second algorithm, we first select a random edge to break, and then select a random point such that the new edges do not cross any existing edges.

9.8.2. ALGORITHM 9.8

Input: A rectangle D, a size n.

Output: A simple polygon in D of size n.

Properties: Complete.

Complexity: $O(n^3)$, may not terminate.

Given a point to add to a polygon, we can clearly determine which of the existing edges may be broken to insert the new vertex. An edge may be broken if the two new edges that would be formed do not cross any of the existing edges (i.e., if both endpoints of e are visible from v). We can then make a random choice from among the possible edges, and insert the vertex. The problem with this approach is that there may be no edges that can be broken (see Figure 10.11, page 141). In such a case, we simply try another random point. Eventually, a point will be found that is visible from both endpoints of some edge.

Given a point, determining which edges can be broken takes $O(n^2)$ time to determine which of the O(n) vertices are visible. While it is possible that the algorithm never terminates, it will only attempt to add O(n) points with high probability. The total time complexity is therefore $O(n^3)$.

9.8.3. ALGORITHM 9.9

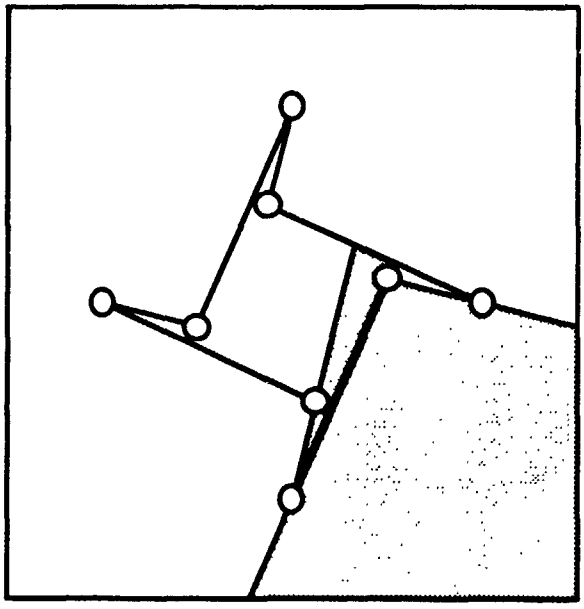
Input: A rectangle D, a size n.

Output: A simple polygon in D of size n.

Properties: Complete.

Complexity: $O(n^2)$, may not terminate.

Given an edge to break, we can determine the region in which the inserted point must lie. We then simply select a random point within the region, and place the new vertex at that point, breaking the previously chosen edge. There will always be some place at which the new vertex can be placed such that the new edges do not cross any existing edges. Determining the region in which the new vertex may be placed amounts to a visibility query. A point is acceptable if both endpoints of the edge to be broken are visible from the point. The point may lie inside or outside the polygon. We further constrain the point to be inside some rectangular region D of the plane (see Figure 9.7).



Edge being split is bold. Gray area is region in which vertex may be placed.

Figure 9.7. Splitting an edge of a polygon.

To compute the region visible from both endpoints of an edge, we simply perform two point visibility queries (one for each endpoint), and intersect the resulting polygons. Each point visibility query is really the union of an internal and an external visibility query. These visibility queries can be performed in O(n) time, so the time to construct a polygon with n vertices is $O(n^2)$. This algorithm can clearly generate any polygon in D. A final concern is the problem of selecting a random point from within a polygonal region. This problem is considered in detail in Section 6.6, page 60.

9.8.4. ALGORITHM 9.10

Input: A rectangle D, an upper bound n on N.

Output: A simple polygon in D.

Properties: Complete, uniform coverage probability = $\frac{1}{N}$.

Complexity: $O(n^3)$.

First, we partition D into N simple polygons. Then, we select one uniformly at random and return it. The coverage probability $Pr_c(x)$ for any point $x \in D$ is clearly $\frac{1}{N}$ since we have a partition of D (see Section 5.4, page 44). The problem we must solve is the construction of a random partitioning of D into simple polygons such that any given simple polygon can be generated. We proceed as follows: First, we generate a random point set. We augment the point set with a set of random points on the perimeter of D. We further augment the point set with the corners of D. Next, we generate a random triangulation T of the point set. Since the point set includes the corners of D, T forms a partitioning of D composed solely of triangles. We then remove sets of adjacent internal edges at random so that T remains a planar subdivision. When removing an edge, if either endpoint currently has degree 2, we also remove the other edge incident on that endpoint. The result is a random planar subdivision that partitions D into a random number of simple polygons.

Let n be the upper bound on the number of polygons in the partition. To construct the partition, we take O(n) time to generate the point set, O(n) time to augment the point set, and $O(n^3)$ time to triangulate the point set. We then remove O(n) edges at random, which can also be done in $O(n^3)$ time. We finally select a random polygon from the partition in O(1) time. Therefore the total time complexity is $O(n^3)$, where n is the upper bound on the number of polygons in the partition.

Chapter 10

10. TRIANGULATIONS

In this chapter we consider the generation of triangulations of various types of polygons and point sets.

10.1. TRIANGULATIONS OF A CONVEX POLYGON

10.1.1. PROBLEM DEFINITION

We are given a convex polygon P in standard form and asked to generate random triangulations of this polygon. Let the n vertices of P be $\{v_0,v_1,\ldots,v_{n-1}\}$ in clockwise order.

10.1.2. DEFINITION OF UNIFORMITY

Since the sample space is finite, we use the discrete uniform probability distribution:

Definition: All sets of triangulations are **events**. A triangulation generator is **uniform** if each of the triangulations of the polygon has the same probability of being generated.

10.1.3. ALGORITHM 10.1

Input: A convex polygon P with n vertices.

Output: A triangulation of P.

Properties: Capable of producing all triangulations of P, uniform probability distribution.

Complexity: $O(n^2)$, assuming capability to do arithmetic operations on O(n) bit integers in O(1) time.

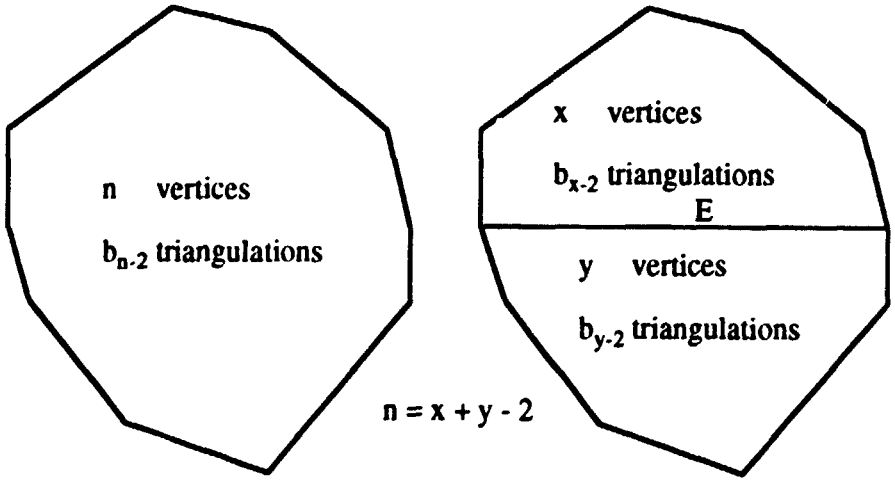
Although this algorithm is not optimal, it is interesting in that it is based on the geometric properties of convex polygons, and that it can be generalized to work for arbitrary simple polygons. See Algorithm 10.2 (page 122) for the purely combinatorial optimal solution.

In the case of convex polygons, we know precisely how many triangulations exist, and we can easily enumerate them. It is therefore straightforward to select a random triangulation with a uniform distribution. However, in order to efficiently construct a random triangulation, we must avoid constructing all possible triangulations.

We start with a fundamental property of triangulations of convex polygons: A convex polygon P with n vertices has b_{n-2} triangulations, where b_k is defined as:

Definition: The k^{th} **Catalan number** is $b_k = \frac{1}{k+1}\binom{2k}{k} = \frac{(2k)!}{(k+1)(k!)^2}$.

Since any edge E breaks the polygon P into two smaller convex polygons, we can easily compute the number of triangulations of P that include or exclude E (see Figure 10.1). One might expect this to lead directly to an algorithm to generate uniformly random triangulations of P. However, difficulties arise if we choose not to include E, since this implies that at least one edge that crosses E must be included in the triangulation.



$(b_{x-2})(b_{y-2})$ triangulations include the edge E.

$(b_{n-2} - b_{x-2})(b_{y-2})$ triangulations exclude the edge E.

Figure 10.1. Counting triangulations that include and exclude an edge.

The algorithm will be recursive, so we will consider adding various edges which break the polygon into parts, each part being either a triangle or a smaller convex polygon. If n = 3, then P

is a triangle, so the only possible triangulation has no internal edges. This is the way our recursion will terminate.

We label some of the possible edges as follows (see Figure 10.2): Let e_0 be the edge between v_0 and v_{n-2}. Let e_i be the edge between v_i and v_{n-1} for i = 1 to n-3, and let e'_i be the edge between v_i and v_0 for i = 2 to n-3.

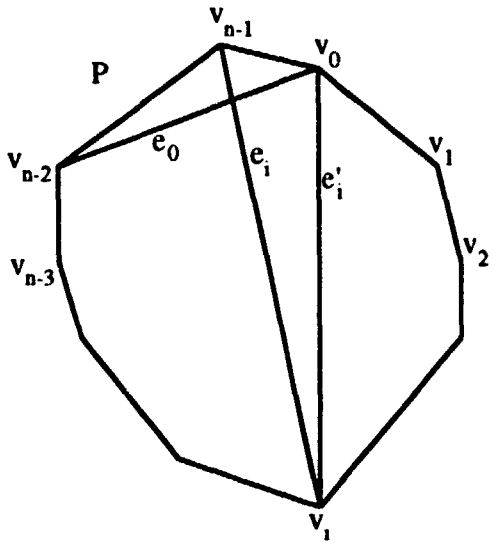


Figure 10.2. Vertex and edge labels for the convex polygon P.

We now consider the addition of the edge e_0 to a triangulation. If the triangulation is to include e_0 then it cannot include any edge e_i for i = 1 to n-3. Conversely, if the triangulation does not include e_0 then it must include some edge e_i for i = 1 to n-3. If we choose not to include e_0 we must now carefully consider the selection of such an edge e_i.

Let T be the set of all triangulations of P. Let T_0 be the subset of T in which the edge e_0 is present. Let T_i be the subset of T in which e_i is present and no edge e_j is present where $j \in \{1,i)$. Lemma 1 will prove that $\{T_0,T_1,\ldots,T_{n-3}\}$ forms a partition of T. Clearly then, $|T| = b_{n-2} = \sum_{i=0}^{n-3} |T_i|$. Lemma 2 will prove that

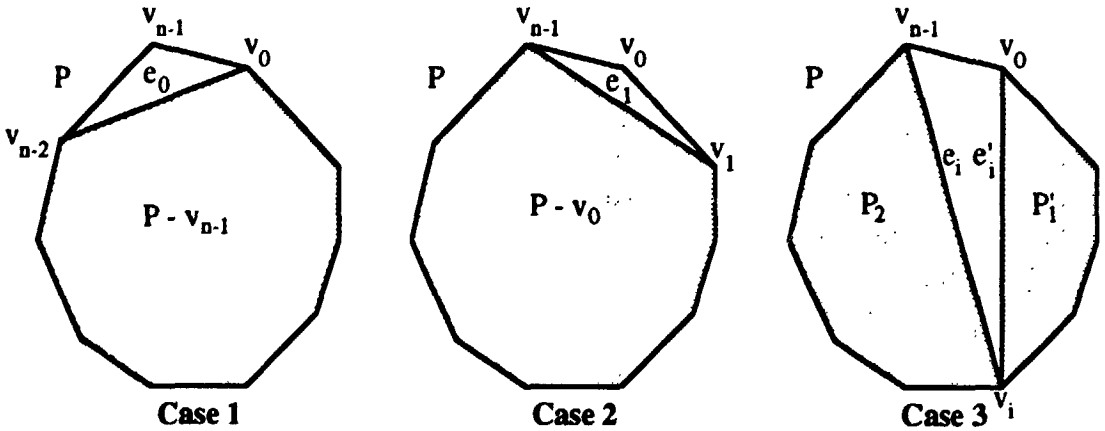
$|T_0| = |T_1| = b_{n-3}$ and that $|T_i| = (b_{i-1})(b_{n-i-2})$ for $i \geq 2$.

To generate a uniformly random triangulation, we select a partition T_i, and then recursively select a random triangulation uniformly within that partition. To select a partition, we compute the sizes of the partitions, and then select one such that the probability of selecting any given T_i is $\frac{|T_i|}{|T|}$. Once T_i has been selected, proceed as follows (see Figure 10.3):

Case 1: If i = 0 then add edge e_0 to result, and recursively triangulate the convex polygon P - v_{n-1} (i.e., P with vertex v_{n-1} removed).

Case 2: If i = 1 then add edge e_1 to result, and recursively triangulate the convex polygon P - v_0 (i.e., P with vertex v_0 removed).

Case 3: If i ≥ 2 then add edges e_i and e'_i to result. Recursively triangulate the convex polygon $P'_1 = (v_0, v_1, \ldots, v_i)$ and the convex polygon $P_2 = (v_i, v_{i+1}, \ldots, v_{n-1})$.



In Cases 1 and 2, we remove a single vertex and triangulate the shaded polygon.
In Case 3, we add two edges and triangulate the two shaded polygons.

Figure 10.3. Three cases for triangulating a convex polygon.

PROOF OF CORRECTNESS

We must show that the generator $G_{10.1}$ implementing Algorithm 10.1 is complete and provides a uniform probability distribution, and that Algorithm 10.1 has $O(n^2)$ time complexity.

Theorem: $G_{10.1}$ is complete.

Proof:

We use induction on n -- the size of P. Our base case is n = 3, in which there is only one triangulation, and our algorithm finds it. We now assume n > 3. We assume that the algorithm can produce any triangulation of any convex polygon with fewer than n vertices. Let t be an arbitrary triangulation of P. We will show that $G_{10.1}$ can construct t.

Case 1: If t includes edge e_0 then, by induction, all possible triangulations of P - v_{n-1} will be capable of being constructed. The triangulation t will be one of these triangulations augmented with the edge e_0, so it is possible that t is constructed when the algorithm selects i = 0.

Case 2: If t excludes edge e_0 but includes edge e_1, then, by induction, all possible triangulations of P - v_1 will be capable of being constructed. As in Case 1, this implies that it is possible that t is constructed when the algorithm selects i = 1.

Case 3: If t excludes edges e_0 and e_1, then t must include some edge e_a such that no edge e_b exists for any $b \in [1,a)$. If the algorithm selects i = a, it will include edges e_a and e'_a. Since no edge e_b exists for any $b \in [1,a)$, we know that t must also include the edge e'_a. The triangulation t will be the combination of some triangulation of the polygon with vertices $\{v_0,v_1,\ldots,v_i\}$ and some triangulation of the polygon with vertices $\{v_i,v_{i+1},\ldots,v_{n-1}\}$ augmented by edges e_a and e'_a. It is therefore possible that t is constructed when the algorithm selects i = a.□

Lemma 1: $\{T_0,T_1,\ldots,T_{n-3}\}$ forms a partition of T.

Proof:

We must show that $T = \bigcup_{i=0}^{n-3} T_i$ and that $T_i \cap T_j = \varnothing$ for all $i \neq j$.

Part 1: $T \subseteq \bigcup_{i=0}^{n-3} T_i$. To prove this, we show that any triangulation t of P is a member of T_i for some $i \in [0,n-3]$. If triangulation t includes the edge e_0 then $t \in T_0$. If t does not include the edge e_0, it must include at least one internal edge incident on v_{n-1}. Let e_j be the edge with smallest index incident on v_{n-1}. Therefore, $t \in T_j$.

Part 2: $T \supseteq \bigcup_{i=0}^{n-3} T_i$. This trivially holds, since each set T_i is defined to be a set of triangulations of P, and therefore a subset of T, which is constrained to meet certain additional requirements.

Part 3: $T_i \cap T_j = \varnothing$ for all $i \neq j$. First, consider the case when $i = 0$. T_0 is the set of triangulations that include the edge e_0, while all other sets T_i for $i > 0$ include only triangulations that exclude e_0. Therefore $T_0 \cap T_j = \varnothing$ for all $j \neq 0$. Second, consider the case when $i,j > 0$. By contradiction: Suppose a triangulation t is a member of both T_i and T_j. The triangulation t excludes edge e_0, so it must include an edge e_a such that no edge e_b exists for any $b \in [1,a)$. By the definition of the sets T_i and T_j, $i = j = b$. We have reached a contradiction, so $T_i \cap T_j = \varnothing$ for all positive $i \neq j$.□

Lemma 2: $|T_0| = |T_1| = b_{n-3}$ and $|T_i| = (b_{i-1})(b_{n-i-2})$ for $i \geq 2$.

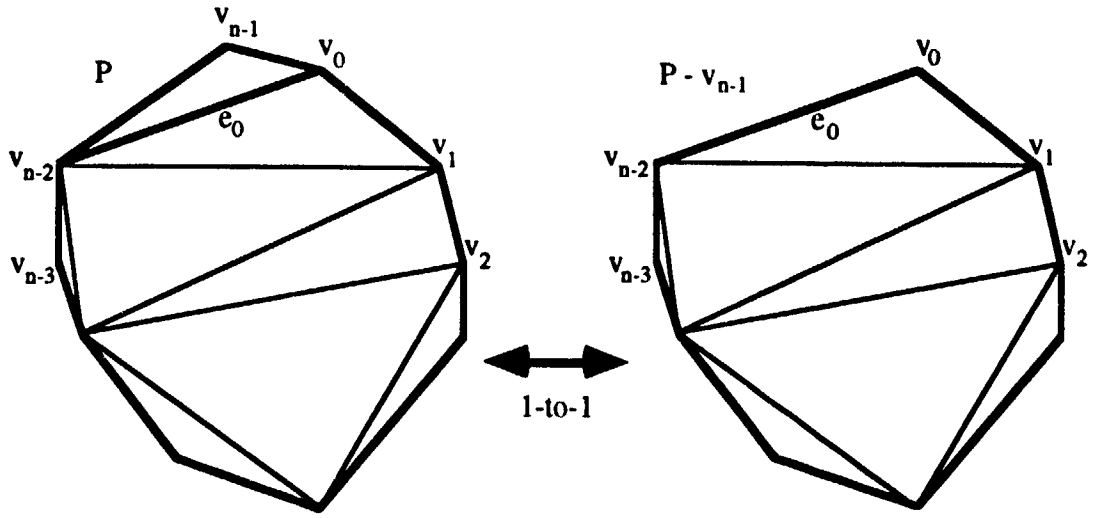
Proof:

Case 1: $|T_0|$ is the number of triangulations of P that include edge e_0. There is a 1-to-1 mapping between such triangulations of P and triangulations of P - v_{n-1} (see Figure 10.4). Since there are b_{n-3} triangulations of P - v_{n-1}, we know that $|T_0| = b_{n-3}$.

Case 2: $|T_1|$ is the number of triangulations of P that include edge e_1. As in Case 1, there is a 1-to-1 mapping between such triangulations of P and triangulations of P - v_1. Since there are b_{n-3} triangulations of P - v_1, we know that $|T_1| = b_{n-3}$.

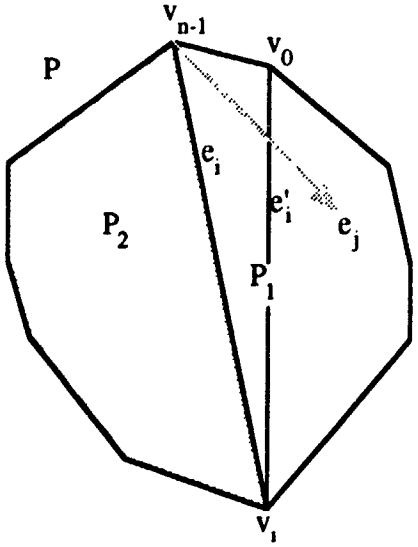
Case 3: $|T_i|$ for $i \geq 2$ is the number of triangulations of P that include edge e_i and do not include any edge e_j for $j \in [1,i)$. Such triangulations must include the edge e'_i (see Figure 10.5). These two edges form a triangle in the center of P, breaking P into two smaller

convex polygons P'_1 and P_2. $|T_i|$ is therefore the product of the number of triangulations of each of these polygons. These polygons have $i + 1$ and $n - i$ vertices respectively, so $|T_i| = (b_{i-1})(b_{n-i-2})$.□



A triangulation of P that includes edge e_0 corresponding triangulation of $P - v_{n-1}$

Figure 10.4. 1-to-1 mapping of triangulations of convex polygons.



e_i breaks P into two convex polygons P_1 and P_2. No edge e_j exists for any $j \in [1,i)$.

P_1 cannot have any triangulation edges incident on v_0. So, e'_i must be present.

Figure 10.5. Rightmost edge e_i existence implies existence of e'_i.

Theorem: $G_{10.1}$ provides a uniform probability distribution.

Proof:

We must prove that each triangulation of P is generated with the same probability. Specifically, this probability is $\frac{1}{b_{n-2}}$ since there are b_{n-2} triangulations of P.

We use induction on n -- the size of P. Our base case is n = 3, in which there is only one triangulation, and our algorithm finds it. In this case, the probability should be $\frac{1}{b_{n-2}} = \frac{1}{b_1} = 1$, as expected. We now assume n > 3. We assume that the algorithm provides a uniform probability distribution for any convex polygon with fewer than n vertices. Let t be an arbitrary triangulation of P. We will show that the algorithm will construct t with probability $\frac{1}{b_{n-2}}$.

Case 1: If t includes edge e_0: The algorithm will construct t if and only if it selects i = 0 *and* it selects the triangulation of P - v_{n-1} that, when augmented with the edge e_0 gives t. The algorithm selects i = 0 with probability $\frac{|T_0|}{|T|} = \frac{b_{n-3}}{b_{n-2}}$ by Lemma 1 and 2. By induction, all possible triangulations of P - v_{n-1} will be constructed with equal probability. Specifically, any one triangulation of P - v_{n-1} will be constructed with probability $\frac{1}{b_{n-3}}$ by Lemma 1 and 2. These are independent events, so we multiply their probabilities to get the probability that t is generated, $\left(\frac{b_{n-3}}{b_{n-2}}\right)\left(\frac{1}{b_{n-3}}\right) = \frac{1}{b_{n-2}}$, as expected.

Case 2: If t excludes edge e_0 but includes edge e_1: The algorithm will construct t if and only if it selects i = 1 *and* it selects the triangulation of P - v_0 that, when augmented with the edge e_1 gives t. As in Case 1, this implies that t is generated with probability $\left(\frac{b_{n-3}}{b_{n-2}}\right)\left(\frac{1}{b_{n-3}}\right) = \frac{1}{b_{n-2}}$, as expected.

Case 3: If t excludes edges e_0 and e_1, then t must include some edge e_a such that no edge e_b exists for any $b \in [1,a)$. The algorithm will construct t if and only if it selects i = a *and* it selects the appropriate triangulations of P'_1 and P_2, which, when combined and augmented with edges e_a and e'_a form t. The algorithm selects i = a with probability

$\frac{|T_a|}{|T|} = \frac{(b_{i-1})(b_{n-i-2})}{b_{n-2}}$. By induction, all possible triangulations of P'_1 will be constructed with equal probability. Specifically, any one triangulation will be constructed with probability $\frac{1}{b_{i-1}}$. Similarly, all possible triangulations of P_2 will be constructed with probability $\frac{1}{b_{n-i-2}}$. These are independent events, so we multiply their probabilities to get the probability that t is generated, $\left(\frac{(b_{i-1})(b_{n-i-2})}{b_{n-2}}\right)\left(\frac{1}{b_{i-1}}\right)\left(\frac{1}{b_{n-i-2}}\right) = \frac{1}{b_{n-2}}$, as expected.□

Theorem: Algorithm 10.1 has time complexity $O(n^2)$.

Proof:

Evaluating a single Catalan number takes O(n) time. However, evaluating all Catalan numbers up to b_{n-2} also takes O(n) time, since $b_k = b_{k-1}\left(\frac{4k-2}{k+1}\right)$. So, using O(n) words of storage ($O(n^2)$ bits), we can compute and store all Catalan numbers needed. For a triangle, the algorithm takes constant time, so that is O(1) given the table of Catalan numbers. Computing a single $|T_i|$ requires at most two Catalan number evaluations, so it takes O(1) time. Determining which of the partitions the resulting triangulation lies in takes O(n) time to evaluate the O(n) expressions $\frac{|T_i|}{|T|}$ for $i \in [0,n-3]$. Since this information allows the algorithm to add at least one edge to the triangulation, the total number of recursive executions of the triangulation is bounded by n. Therefore, the total time complexity is $O(n^2)$.□

10.1.4. ALGORITHM 10.2

Input: A convex polygon P with n vertices.

Output: A triangulation of P.

Properties: Complete, uniform probability distribution.

Complexity: O(n), uses only integers with O(log n) bits, optimal.

This optimal algorithm is described in [Atkinson & Sack 92], and is summarized briefly here. There is a 1-to-1 mapping between triangulations of a convex polygon with n vertices and binary trees with n - 2 nodes. Since uniform random binary trees with n - 2 nodes can be generated in O(n) time, and since we can construct the triangulation corresponding to a binary tree in O(n) time, we can generate uniform random triangulations in O(n) time.

10.2. TRIANGULATIONS OF A POINT SET

In this section we consider the generation of triangulations of a given point set. Such a generator can be used as a stepping stone for the generation of random triangulated point sets.

10.2.1. PROBLEM DEFINITION

We are given a point set S in standard form and asked to generate a random triangulation of this point set. Let the n points of S be $\{v_0, v_1, \ldots, v_{n-1}\}$. There are only a finite number of triangulations of any given point set. Some algorithms require points to be in general position while others do not.

Recall that any triangulation of n points has exactly 3n - k - 3 edges, where k is the number of points on the convex hull of the point set. In the resulting triangulation, k of the edges will form the convex hull of the point set, and the remaining 3n - 2k - 3 edges will be internal edges.

10.2.2. DEFINITION OF UNIFORMITY

Since the sample space is finite, we use the discrete uniform probability distribution:

Definition: All sets of triangulations are **events**. A point set triangulation generator is **uniform** if each of the triangulations of the point set has the same probability of being generated.

10.2.3. COUNTING THE NUMBER OF POSSIBLE TRIANGULATIONS

A polynomial time algorithm for counting the number of triangulations of a point set is likely to be needed for any polynomial time algorithm to be constructed to generate triangulations of a point set with a uniform probability distribution. We do not know of any such counting algorithm.

10.2.4. ALGORITHM 10.3

Input: A point set S of size n.

Output: A triangulation of S.

Properties: Complete.

Complexity: $O(n^3)$.

We will add edges to get a triangulation, starting with no edges. We will also construct a set of triangles, again starting with no triangles. We know that the convex hull edges are in any triangulation, so, pick a random edge of the convex hull and grow the triangulation in from that edge. To grow from a given edge in a given direction (i.e., a given side of the edge), proceed as follows: First, if requested to grow out from a convex hull edge, do nothing. Otherwise, add the edge to the triangulation, and then find all triangles formed by the edge and points in S satisfying the following properties:

- No other points in S lie in the triangle.
- The triangle lies on the appropriate side of the edge.
- The point defining the triangle does not lie in any of the triangles already created. The point can however be a vertex of a triangle already created.

Select one of these triangles at random and add it to the triangle set for the triangulation. Recursively grow out from each of the other two edges of the triangle.

PROOF OF CORRECTNESS

We must show that the generator $G_{10.3}$ implementing Algorithm 10.3 is complete, and that Algorithm 10.3 has $O(n^3)$ time complexity.

Theorem: $G_{10.3}$ is complete.

Proof:

Let t be an arbitrary triangulation of S. We will show that $G_{10.3}$ can construct t. The algorithm could start at any convex hull edge h. It could then form its first triangle as the triangle in t which contains the edge h. Now consider the general case. When the algorithm is asked to grow from an edge e in a given direction, it is possible that the algorithm selects the triangle which lies in the given direction from e in t. Therefore, $G_{10.3}$ could produce t.□

Theorem: Algorithm 10.3 has time complexity $O(n^3)$.

Proof:

The algorithm finds the convex hull of S, which takes O(n log n) time. Each grow operation adds at least one edge to the triangulation, so O(n) grow operations will be performed. Testing a given triangle to determine whether it satisfies the three properties list above takes O(n) time. The first property can be tested in O(n) time since point inclusion takes only O(1) time for a triangle. The second property can be tested in O(1) time this amounts to a single turn test to determine on which side of the line defined by the edge the point lies. The third property takes O(n) time since there are O(n) triangles and point inclusion takes O(1) time. Since we must test O(n) triangles, the total time to find all triangles satisfying all three properties is $O(n^2)$. Therefore the grow operation takes $O(n^2)$, so the total time is $O(n^3)$.□

10.2.5. ALGORITHM 10.4

Input: A point set.

Output: A triangulation of the point set.

Properties: Complete.

Complexity: $O(n^3)$.

The triangulation of a point set has some special properties that triangulated polygons do not have. For example, the edges of the convex hull of the point set are always included in the triangulation. Most reasonably large point sets have many triangulations. Any pair of points may have an edge between them if no other points in the set lie on the line segment joining them.

To generate a random triangulation of a point set, we can simply construct a list of all $O(n^2)$ possible edges using the above rule. These possible edges can then be assigned a random order. The triangulation is constructed by adding edges from the list. If an edge would cross one or more edges which have already been added to the triangulation, then the edge is not added. Once all the possible edges have been considered, the result is a valid triangulation of the point set. The edges of the convex hull may appear anywhere in the ordering, but they will always be added to the triangulation because they do not intersect any of the other possible edges. For this reason, the edges of the convex hull can be removed from the set of possible edges before a random ordering is chosen. In fact, any edge which does not intersect any other possible edge may be excluded from the random ordering since it will always be present in the triangulation.

PROOF OF CORRECTNESS

We must show that the generator $G_{10.4}$ implementing Algorithm 10.4 is complete, and that Algorithm 10.4 has $O(n^3)$ time complexity.

Theorem: Algorithm 10.4 takes $O(n^3)$ time.

Proof:

To determine whether an edge is possible takes $O(n)$ time since all other points in the point set must be checked for collinearity. There are $\binom{n}{2} = O(n^2)$ edges to consider, so the total time to construct a list of possible edges is $O(n^3)$. If the point set is in general position, then all edges are possible, so the set can be constructed in $O(n^2)$ time. We now construct a random ordering of this set of $O(n^2)$ possible edges in $O(n^2)$ time. For each edge in this ordering, we must check whether it intersects any of the $O(n)$ edges already added. The total time to construct the triangulation is therefore $O(n^3)$.□

Theorem: $G_{10.4}$ is complete.

Proof:

Let t be an arbitrary triangulation of the point set. We will show that $G_{10.4}$ can generate t. Let p be any ordering of the possible edges in which the first edges are the edges of t. The algorithm could select the ordering p since it can choose any ordering of the possible edges, and the edges of t are clearly possible edges. If the algorithm selected the ordering p, it would add all the edges of t without finding any intersections. Furthermore, all of the remaining edges in the ordering will not be added since they will intersect at least one edge of t. Therefore, $G_{10.4}$ produces t if it selects the ordering p.□

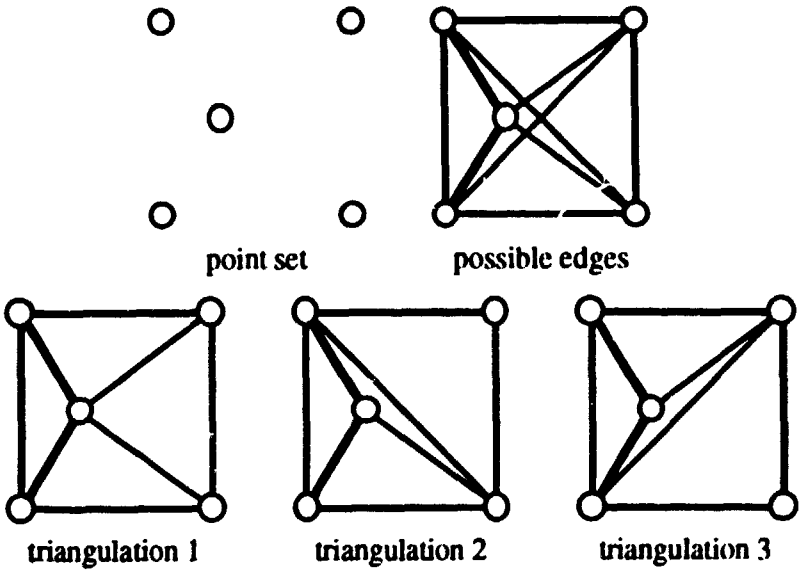
PROOF OF NON-UNIFORMITY

Theorem: $G_{10.4}$ does not provide a uniform probability distribution.

Proof: (by counter-example)

There is a fixed number of possible triangulations of any given point set. Each one of these triangulations should have the same probability of being generated. The point set of

Figure 10.6 has three possible triangulations. For $G_{10.4}$ to provide a uniform probability distribution, it must generate these with equal probability.



The edges of the convex hull, as well as two other edges, are present in all triangulations.

Figure 10.6. A point set and its possible triangulations.

There are four possible edges other than the edges which are always present. The algorithm must chose one of the 4! = 24 orderings of four items. For the algorithm to provide a uniform probability distribution, exactly 8 of these orderings must yield each of the three possible triangulations. On examination of the 24 possible orderings, we find that 6 produce triangulation 1 while 9 produce triangulations 2 and 3. In other words, our algorithm would under-represent triangulation 1, producing it less often than the other triangulations. This counter-example shows that $G_{10.4}$ does not provide a uniform probability distribution.□

10.3. TRIANGULATIONS OF A SIMPLE POLYGON

In this section we consider generating triangulations of a given polygon. Such a generator can be used as a stepping stone in the generation of triangulated simple polygons.

10.3.1. PROBLEM DEFINITION

We are given a simple polygon in standard form and asked to generate random triangulations of this polygon. There are a finite number of triangulations of any given polygon.

10.3.2. DEFINITION OF UNIFORMITY

Since the sample space is finite, we use the discrete uniform probability distribution:

Definition: All sets of triangulations are **events.** A polygon triangulation generator is **uniform** if each of the triangulations of the polygon has the same probability of being generated.

10.3.3. COMPUTING THE NUMBER OF POSSIBLE TRIANGULATIONS

ALGORITHM 10.5

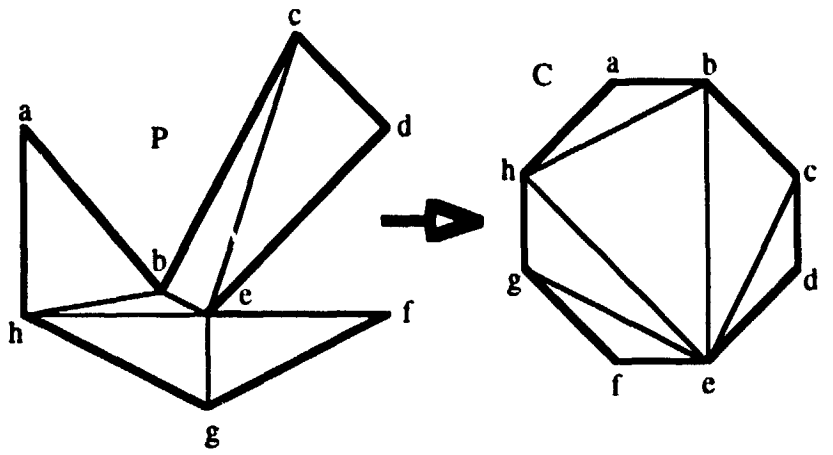
Input: A simple polygon $P = (v_1, v_2, \ldots, v_n)$.

Output: $N(P)$ = the number of possible triangulations of P.

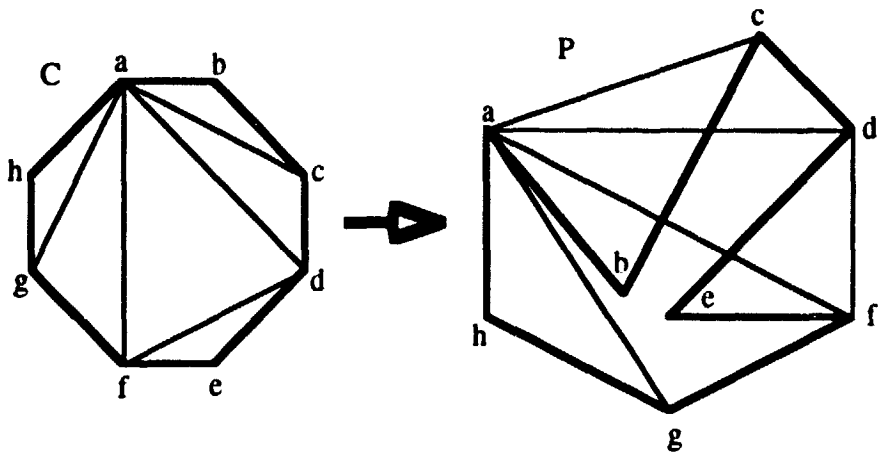
Complexity: $O(n^3)$.

In the case of an arbitrary simple polygon as compared to a convex polygon, it is more difficult to determine how many triangulations exist, and it is therefore more difficult to produce a uniform distribution in polynomial time. We now consider the problem of counting triangulations. This algorithm was developed by Sack and Epstein.

We construct a convex polygon C from P by placing the vertices of P regularly around a circle, in order around P. C is therefore a regular n-gon. Let $C = (w_1, w_2, \ldots, w_n)$ such that v_i corresponds to w_i. Any triangulation of P corresponds to a triangulation of C (see Figure 10.7). However, a triangulation of C may not correspond to a triangulation of P, since it may use an edge (w_i, w_j) such that v_i and v_j are not mutually visible in P. More precisely, the triangulations of P are precisely those triangulations of C which use only those edges (w_i, w_j) for which v_i and v_j are mutually visible in P.



Any triangulation of P corresponds to a triangulation of C.



A triangulation of C may not correspond to a triangulation of P.

Figure 10.7. Relationship between triangulations of P and triangulations of C.

We have therefore reduced the problem of counting triangulations of a simple polygon to counting triangulations of a convex polygon which use only edges from a given set E. We now use dynamic programming to solve this problem.

Let T(i,i+k) be the number of triangulations of the polygon $C(i,i+k) = (w_i,w_{i+1},\ldots,w_{i+k})$ which use only edges from the set E. The result is therefore N(P) = T(1,n). We will compute values of T for increasing values of k, storing the results in an $n \times n$ array T:

```
FOR i := 1 TO n DO
    FOR j := i TO n DO
        LET T[i,j] := 0.
    END FOR.
END FOR.
FOR I := 1 TO n - 1 DO
    LET T[i,i + 1] := 1.
END FOR.
FOR k := 2 TO n - 1 DO
    FOR i := 1 TO n - k DO
        IF (w_i,w_{i+k}) ∈ E THEN
            FOR j := i + 1 TO i + k - 1 DO
                LET T[i,i + k] := T[i,i + k] + (T[i,j])(T[j,i + k]).
            END FOR.
        END IF.
    END FOR.
END IF.
RETURN T[1,n].
```

PROOF OF CORRECTNESS

Theorem: This algorithm correctly computes T[i,i + k] = T(i,i + k).

Proof:

If $(w_i,w_{i+k}) \in E$ then the algorithm computes $T[i,i+k] = \sum_{j=i+1}^{i+k-1}(T[i,j])(T[j,i+k])$. We use induction on k. If k = 1 then we have "flat" polygons with only two vertices. Such polygons clearly have only one triangulation, so T(i,i + 1) = 1, as given by the second initialization phase of the algorithm. If k > 1, then we assume that T[i,i + k'] = T(i,i + k') for all k' < k. We partition the triangulations of C(i,i+k) into k - 1 sets according to which

vertex in the set $\{w_{i+1},w_{i+2},\ldots,w_{i+k-1}\}$ is the third vertex of the triangle with vertices w_i and w_{i+k}. This is clearly a partition since any triangulation of C(i,i+k) lies in exactly one of these sets. Consider a set in this partition which includes those triangulations containing the triangle with vertices $\{w_i,w_j,w_{i+k}\}$ for some j. Every triangulation in this set can be constructed by triangulating the two polygons $(w_i,w_{i+1},\ldots,w_j)$ and $(w_j,w_{j+1},\ldots,w_{i+k})$, so the number of triangulations in the set is $\sum_{j=i+1}^{i+k-1}(T(i,j))(T(j,i+k))$ (see Figure 10.8). This recurrence relation was also independently formulated by Bruce Richter [Richter 92]. By induction, this is precisely what the algorithm computes.

If $(w_i,w_{i+k}) \notin E$ then there can be no triangulation of the sub-polygon C(i,i + k), so the algorithm correctly computes T[i,i + k] = T(i,i + k) = 0.□

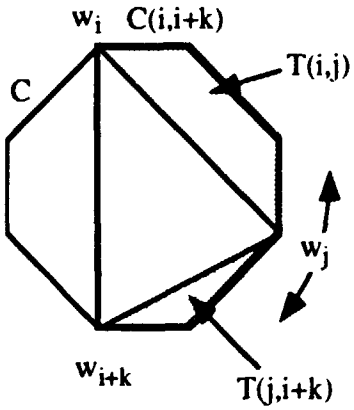


Figure 10.8. Counting triangulations of C(i,i+k) that include the triangle (w_i,w_j,w_{i+k}).

Theorem: This algorithm has time complexity $O(n^3)$.

Proof:

Each of the three nested loops iterates O(n) times, and the actual change to T[i,i + k] takes O(1) time, so the total time is $O(n^3)$.□

EXTENSIONS

In the process of computing N(P), the algorithm computes the number of triangulations of many sub-polygons. Specifically, the algorithm computes the number of triangulations of the simple polygons $P(i,i+k) = (v_i,v_{i+1},\ldots,v_{i+k})$ for k = 2 to n - 1 and for i = 1 to n - k. If we extend the set of such polygons to include all polygons produced by adding an edge between mutually visible vertices of P, we will not increase the time complexity of the algorithm. The only change required in the algorithm is to let i loop to n rather than to n - k, and to use modular arithmetic on the vertex indices. The result is that the algorithm provides N(P(i,j)) for all i, j. We will see that this capability will be useful in generating uniformly random triangulations of simple polygons.

COROLLARY

The following result is easily seen given the above algorithm:

Corollary: A convex n-gon has more triangulations than a non-convex simple n-gon.

Proof:

The fact that any triangulation of P corresponds to a triangulation of C implies that the number of triangulations of P is no more than the number of triangulations of C. In other words, $N(P) \leq N(C)$. Since P is an arbitrary simple polygon with n vertices, we know that convex n-gons have the most triangulations of all the n vertex simple polygons. In fact, since any non-convex polygon C' will have at least one pair of vertices v_i,v_j which are not mutually visible, the set of edges available for triangulating C' will always be a strict subset of the set of edges available for triangulating C. There exists at least one triangulation of C that uses an edge not available for triangulations of C'. Therefore, C has strictly more triangulations than C'.□

10.3.4. ALGORITHM 10.6

Input: A simple polygon P with n vertices.

Output: A triangulation of P.

Properties: Complete.

Complexity: $O(n^2)$.

This is a recursive algorithm. If n = 3, we are triangulating a triangle, so no internal edges are needed. If $n \geq 3$, we compute the visibility polygon in P from a randomly selected vertex v, without triangulation, using an O(n) algorithm such as [ElGindy & Avis 81]. Next, we construct a list L of vertices of P visible from v other than the neighbors of v, which are always visible. If $|L| = 0$, then v must be an ear in all triangulations, so we cut off the ear and triangulate the remainder (P - v) recursively. If $|L| \geq 1$, we select a vertex $w \in L$ at random, add an edge from v to w, and recursively triangulate both polygons produced. O(n) work is done and one edge is added to the triangulation, so the total time complexity is $O(n^2)$.

While it might seem that this algorithm could be improved to achieve an O(n log n) time complexity, this is not the case. To achieve such a complexity, we must ensure that the edge added splits the polygon into two polygons of about the same size, so that the depth of recursion is O(log n). The worst case scenario is when a polygon with n vertices is split into two polygons with m and 2m + 1 vertices respectively. When selecting a vertex from the list L, we can therefore require the edge to split the polygon into polygons with this worst case size relationship without affecting the completeness of the algorithm. However, since there may be no such edges incident on the chosen vertex v, we may have to try more vertices to find such an edge. In the worst case, the polygon has a unique triangulation, and only three of the n vertices of the polygon admit such an edge. The number of trial vertices would therefore be O(n), leading to an $O(n^2 \log n)$ worst case time complexity.

10.3.5. ALGORITHM 10.7

Input: A simple polygon P with n vertices.

Output: A triangulation of P.

Properties: Complete, uniform probability distribution.

Complexity: $O(n^4)$, assuming capability to do arithmetic operations on O(n) bit integers in O(1) time.

This is a generalization of Algorithm 10.1 (page 113) for triangulating convex polygons. We use the ability to compute N(P), the number of triangulations of P in polynomial time. We will use Algorithm 10.5, to compute the number of triangulations of P and of all polygons produced by adding a triangulation edge to P in $O(n^3)$ time.

Given the ability to count triangulations of a polygon in polynomial time, we can easily count the number of triangulations that include a given edge E by multiplying the number of triangulations of each of the smaller polygons produced by adding the edge E. One might expect this to lead directly to an algorithm to generate uniformly random triangulations of P. However, difficulties arise if we choose not to include E, since this implies that at least one edge that crosses E must be included in the triangulation. We deal with this problem in exactly the same way we dealt with it for convex polygons.

The algorithm will be recursive, so we will consider adding various edges which break the polygon into parts, each part being either a triangle or a smaller polygon. If n = 3, then P is a triangle, so the only possible triangulation has no internal edges. This is the way our recursion will terminate.

Find a non-reflex vertex v whose neighbors u and w are mutually visible (u, v, and w are in clockwise order around P). At least two such vertices must exist by the well known Two Ear Theorem. Let e_0 be the ear edge connecting these neighbors. In other words, we could triangulate P by cutting off v, and using e_0 as a new polygon edge. Let $\{v_1,v_2,\ldots,v_k\}$ be the vertices visible

from v, excluding the neighbors of v, in clockwise order around the polygon (see Figure 10.9). Let e_i be the edge between v and v_i. Let e'_i be the edge between w and v_i, if that edge is a possible triangulation edge. Let P_i and Q_i be the polygons produced by adding edge e_i to P, where P_i includes w and Q_i includes u (see Figure 10.10). Further let P'_i be P_i - v.

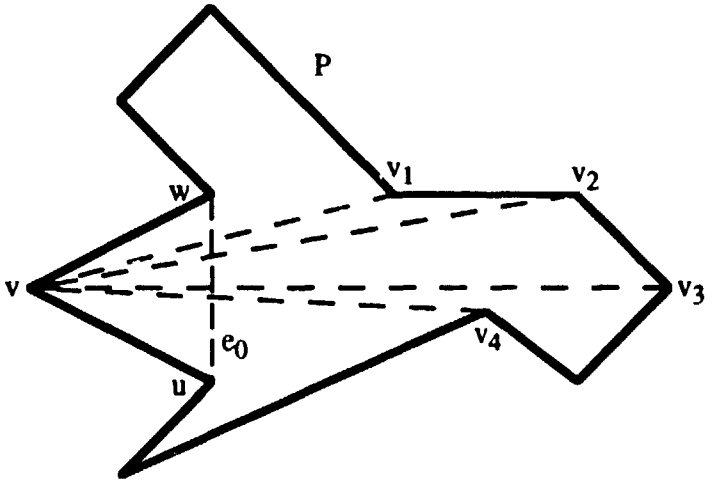


Figure 10.9. Vertex labelling from an ear.

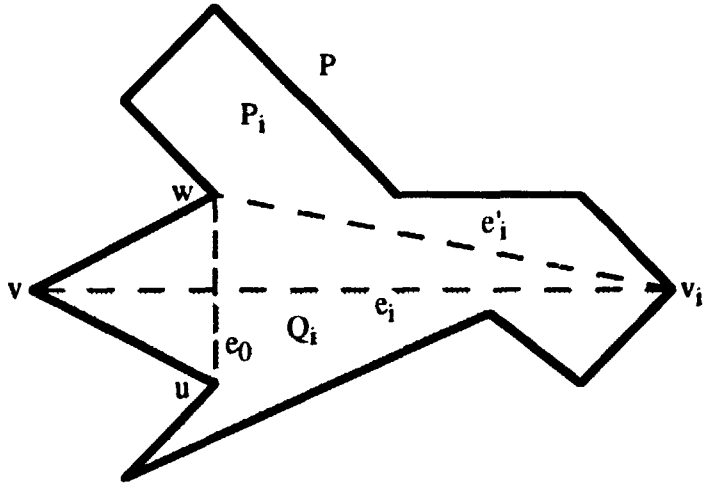


Figure 10.10. Edge labelling from an ear.

We now consider the addition of the ear edge e_0 to a triangulation. If the triangulation is to include e_0 then it cannot include any edge incident on v. Conversely, if the triangulation does not include e_0 then it must include some edge incident on v. If we choose not to include e_0 we must now carefully consider the selection of such an edge e_i for i = 1 to k.

Let T be the set of all triangulations of P. Let T_0 be the subset of T in which the edge e_0 is present. Let T_i be the subset of T in which e_i is present and no edge e_j is present where $j \in [1,i)$. A given T_i may be empty, since it may be that some edge e_i cannot be the extreme edge incident on v. This will be the case if the edge e'_i is not a possible triangulation edge. We will prove that $\{T_0,T_1,\ldots,T_k\}$ forms a partition of T. Clearly then, $|T| = \sum_{i=0}^{k} |T_i|$. Clearly $|T_0| = N(P - v)$ and $|T_i| = (N(P'_i))(N(Q_i))$ for $i \geq 2$.

To generate a uniformly random triangulation, we select a partition T_i, and then recursively select a random triangulation uniformly within that partition. To select a partition, we compute the sizes of the partitions, and then select one such that the probability of selecting any given T_i is $\frac{|T_i|}{|T|}$. Our algorithm for counting the number of triangulations of a simple polygon will provide $|T|$ as well as $|T_i|$ for $i = 0$ to k. Once T_i has been selected, proceed as follows:

Case 1: If $i = 0$ then add edge e_0 to result, and recursively triangulate the convex polygon P - v (i.e., P with vertex v removed).

Case 2: If $i = 1$ and v_1 and w are neighbors then add edge e_1 to result, and recursively triangulate the convex polygon P - w (i.e., P with vertex w removed).

Case 3: Otherwise add edges e_i and e'_i to result. Recursively triangulate the polygons $P'_i = (w,\ldots,v_i)$ and $Q_i = (v, v_i,\ldots,u)$.

PROOF OF CORRECTNESS

We must show that the generator $G_{10.7}$ implementing Algorithm 10.7 is complete and provides a uniform probability distribution, and that Algorithm 10.7 has $O(n^4)$ time complexity.

Theorem: $G_{10.7}$ is complete.

Proof:

We use induction on n -- the size of P. Our base case is $n = 3$, in which there is only one triangulation, and $G_{10.7}$ finds it. We now assume $n > 3$. We assume that $G_{10.7}$ can

produce any triangulation of any polygon with fewer than n vertices. Let t be an arbitrary triangulation of P. We will show that $G_{10.7}$ can construct t. Let v be any ear vertex of t, and define our vertex and edge labels correspondingly.

Case 1: If t includes edge e_0 then, by induction, all possible triangulations of P - v will be capable of being constructed. The triangulation t will be one of these triangulations augmented with the edge e_0, so it is possible that t is constructed when $G_{10.7}$ selects i = 0.

Case 2: If t excludes edge e_0 but includes edge e_1, and v_1 and w are neighbors, then, by induction, all possible triangulations of P - v_1 will be capable of being constructed. As in Case 1, this implies that it is possible that t is constructed when $G_{10.7}$ selects i = 1.

Case 3: Otherwise, t must include some edge e_a such that no edge e_b exists for any $b \in [1,a)$. If the algorithm selects i = a, it will include edges e_a and e'_a. Since no edge e_b exists for any $b \in [1,a)$, we know that t must also include the edge e'_a. The triangulation t will be the combination of some triangulation of the polygon with vertices $(w,\ldots,v_i)$ and some triangulation of the polygon with vertices $(v, v_i,\ldots,u)$ augmented by edges e_a and e'_a. It is therefore possible that t is constructed when $G_{10.7}$ selects i = a.□

Theorem: $\{T_0,T_1,\ldots,T_k\}$ forms a partition of T.

Proof:

We must show that $T = \bigcup_{i=0}^{k} T_i$ and that $T_i \cap T_j = \varnothing$ for all $i \neq j$.

Part 1: $T \subseteq \bigcup_{i=0}^{k} T_i$. To prove this, we show that any triangulation t of P is a member of T_i for some $i \in [0,k]$. If triangulation t includes the edge e_0 then $t \in T_0$. If t does not include the edge e_0, it must include at least one internal edge incident on v. Let e_j be the edge with smallest index incident on v. Therefore, $t \in T_j$.

Part 2: $T \supseteq \bigcup_{i=0}^{k} T_i$. This trivially holds, since each set T_i is defined to be a set of triangulations of P, and therefore a subset of T, which is constrained to meet certain additional requirements.

Part 3: $T_i \cap T_j = \varnothing$ for all $i \neq j$. First, consider the case when $i = 0$. T_0 is the set of triangulations that include the edge e_0, while all other sets T_i for $i > 0$ include only triangulations that exclude e_0. Therefore $T_0 \cap T_j = \varnothing$ for all $j \neq 0$. Second, consider the case when $i,j > 0$. By contradiction: Suppose a triangulation t is a member of both T_i and T_j. The triangulation t excludes edge e_0, so it must include an edge e_a such that no edge e_b exists for any $b \in [1,a)$. By the definition of the sets T_i and T_j, $i = j = b$. We have reached a contradiction, so $T_i \cap T_j = \varnothing$ for all positive $i \neq j$.□

Theorem: $G_{10.7}$ provides a uniform probability distribution.

Proof:

We must prove that each triangulation of P is generated with the same probability. Specifically, this probability is $\frac{1}{N(P)}$.

We use induction on n -- the size of P. Our base case is n = 3, in which there is only one triangulation, and our algorithm finds it. In this case, the probability should be $\frac{1}{N(P)} = \frac{1}{1} = 1$, as expected. We now assume n > 3. We assume that the algorithm provides a uniform probability distribution for any polygon with fewer than n vertices. Let t be an arbitrary triangulation of P. We will show that $G_{10.7}$ will construct t with probability $\frac{1}{N(P)}$.

Case 1: If t includes edge e_0: The algorithm will construct t if and only if it selects $i = 0$ *and* it selects the triangulation of P - v that, when augmented with the edge e_0 gives t. The algorithm selects $i = 0$ with probability $\frac{|T_0|}{|T|} = \frac{N(P-v)}{N(P)}$. By induction, all possible triangulations of P - v will be constructed with equal probability. Specifically, any one triangulation of P - v will be constructed with probability $\frac{1}{N(P-v)}$. These are independent events, so we multiply their probabilities to get the probability that t is generated, $\left(\frac{N(P-v)}{N(P)}\right)\left(\frac{1}{N(P-v)}\right) = \frac{1}{N(P)}$, as expected.

Case 2: If t excludes edge e_0 but includes edge e_1, and v_1 is adjacent to w: The algorithm will construct t if and only if it selects i = 1 *and* it selects the triangulation of P - w that, when augmented with the edge e_1 gives t. As in Case 1, this implies that t is generated with probability $\left(\frac{N(P-v)}{N(P)}\right)\left(\frac{1}{N(P-v)}\right)=\frac{1}{N(P)}$, as expected.

Case 3: Otherwise, t must include some edge e_a such that no edge e_b exists for any $b \in [1,a)$. The algorithm will construct t if and only if it selects i = a *and* it selects the appropriate triangulations of P'_i and Q_i, which, when combined and augmented with edges e_a and e'_a form t. The algorithm selects i = a with probability

$$\frac{|T_a|}{|T|}=\frac{(N(P'_i))(N(Q_i))}{N(P)}.$$

By induction, all possible triangulations of P'_i will be constructed with equal probability. Specifically, any one triangulation will be constructed with probability $\frac{1}{N(P'_1)}$. Similarly, all possible triangulations of Q_i will be constructed with probability $\frac{1}{N(Q_i)}$. These are independent events, so we multiply their probabilities to get the probability that t is generated,

$$\left(\frac{(N(P'_i))(N(Q_i))}{N(P)}\right)\left(\frac{1}{N(P'_i)}\right)\left(\frac{1}{N(Q_i)}\right)=\frac{1}{N(P)}, \text{ as expected.}\square$$

Theorem: Algorithm 10.7 has time complexity $O(n^4)$.

Proof:

For a triangle, the algorithm takes constant time, so that is O(1). Computing a single $|T_i|$ takes $O(n^3)$. However, we can determine which of the partitions the resulting triangulation lies in using only one execution of the $O(n^3)$ Algorithm 10.5 for counting triangulations, so the time to evaluate the O(n) expressions $\frac{|T_i|}{|T|}$ for $i \in [0,k]$ is $O(n^3)$. Since this information allows the algorithm to add at least one edge to the triangulation, the total number of recursive executions of the triangulation is bounded by n. Therefore, the total time complexity is $O(n^4)$.$\square$

10.4. TRIANGULATED SIMPLE POLYGONS ON A POINT SET

10.4.1. PROBLEM DEFINITION

The polygons produced have the entire given point set V as their vertex set.

10.4.2. DEFINITION OF UNIFORMITY

Since the sample space is finite, we use the discrete uniform probability distribution:

Definition: All sets of triangulations of simple polygons whose vertices are precisely the set V are **events**. A triangulated simple polygon generator is **uniform** if each of the triangulations of each of the simple polygons whose vertices are precisely the set V has the same probability of being generated.

10.4.3. ALGORITHM 10.8

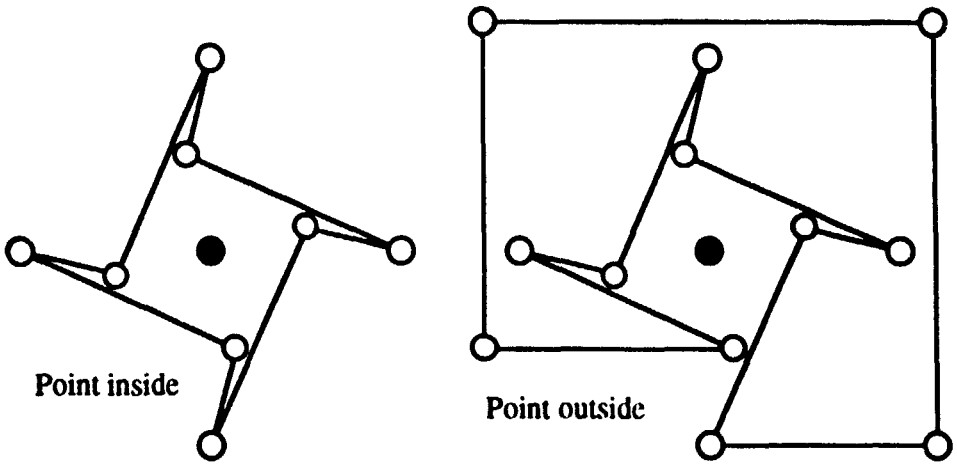
Input: A point set V of size n.

Output: A triangulated simple polygon whose vertices are the point set V.

Properties: Complete.

Complexity: $O(n^4)$, may not terminate.

Given a point v to add to a polygon P, we can clearly determine which of the existing edges may be broken to insert the new vertex. An edge e may be broken if and only if the two new edges that would be formed do not cross any of the existing edges (i.e., if both endpoints of e are visible from v). We can then make a random choice from among the possible edges, and insert the vertex. The problem with this approach is that there may be no edges that can be broken. This problem may even occur when the point being added lies outside the polygon (see Figure 10.11). This problem may cause the algorithm to "get stuck", being unable to add all of the points in V to the polygon. In this case, we try again, restarting the algorithm. If we are given a point set V, we can construct the polygon P iteratively, such that none of the remaining points in V lie inside P. When adding a vertex $v \in V$ to P, there may be no edge that can be broken to add v.



Black points are not visible from both endpoints of any edge.

No existing edges can be broken to insert black points.

Figure 10.11. Two cases in which a point cannot be added to a polygon.

We will remove points from the set V while we add these vertices to the polygon P. Initialize P to be a random triangle consisting of three points in V such that no other points in V lie inside the triangle. Remove these three points from V. We now repeatedly add points in V to P, removing them from V. We add a point to P as follows:

Iterate through all vertices v remaining in V. For each such v, iterate through all edges e of P. If v is visible from both endpoints of e *and* no points remaining in V are inside the triangle formed by v and e, then v may be added to P by breaking edge e. Therefore, add the pair (v,e) to a set S of possible ways to grow P. When all such pairs have been considered, and the set S is complete, choose a pair at random and perform the grow operation. Specifically, remove the vertex v from V, and add it to P by inserting it between the endpoints of e. If the set S of possible ways to grow P is empty, then we are "stuck", so restart the algorithm. The algorithm will rarely have to restart.

The result is really a triangulated polygon since P was created by adding triangles one at a time. Each triangle is defined by the vertex v and the endpoints of e from the chosen (v,e) pair.

PROOF OF CORRECTNESS

We must show that the generator $G_{10.8}$ implementing Algorithm 10.8 is complete, and that Algorithm 10.8 has $O(n^4)$ time complexity.

Theorem: $G_{10.8}$ is complete.

Proof:

Let Q be an arbitrary triangulated polygon whose vertices are the point set V. We will show that the algorithm can generate Q. Select any triangle t_1 of Q as a starting point. The triangle t_1 clearly contains no other points in V, so the algorithm could initialize P to t_1. Now consider a neighboring triangle t_2 to t_1 in Q. The algorithm could clearly select a (v,e) pair which would cause it to add t_2 to P. Such a (v,e) pair would be selected as a possible way to grow P, since v is clearly visible from both endpoints of e, and since no points in V are inside the triangle t_2. In fact, the triangles of Q could be added in any order such that triangles added are always adjacent to other previously added triangles. Therefore, there are many ways in which $G_{10.8}$ could construct Q.□

Theorem: Algorithm 10.8 has time complexity $O(n^4)$ if we assume O(1) algorithm restarts.

Proof:

To select a random initial triangle, the algorithm iterates through $O(n^3)$ triangles and do containment tests for each of the O(n) points in V, for a total time of $O(n^4)$. The polygon is then grown in O(n) steps. For each step, the algorithm iterates through $O(n^2)$ pairs of vertices in V and edges of P. For each such pair, O(n) vertices are tested for triangle containment and O(n) edges are tested to determine visibility of the endpoints of the edge, so the total time per step is $O(n^3)$. Therefore, the time to construct the triangulated polygon P is $O(n^4)$.□

10.5. TRIANGULATED SIMPLE POLYGONS IN A RECTANGLE

In this section we consider generation of triangulated simple polygons. This is a central geometric object which has many applications.

10.5.1. MOTIVATION

Consider the problem of testing an algorithm which requires a triangulated polygon as input. For example, the visibility polygon can be constructed for any given source point within the triangulated polygon. To test such an algorithm we need random triangulated polygons. The obvious way to produce these is to construct random polygons and then triangulate them using some algorithm. The problem with this approach is that only one of the many possible triangulations for any given polygon will be produced. A better solution is to construct the triangulated polygon directly.

10.5.2. PROBLEM DEFINITION

Polygons generated are in standard form (i.e. all vertices distinct and no three consecutive vertices collinear). We are given only a rectangular domain and asked to generate a triangulated simple polygon that lies entirely in the domain. We are not given a polygon to triangulate.

10.5.3. ALGORITHM 10.9

Input: A rectangle D.

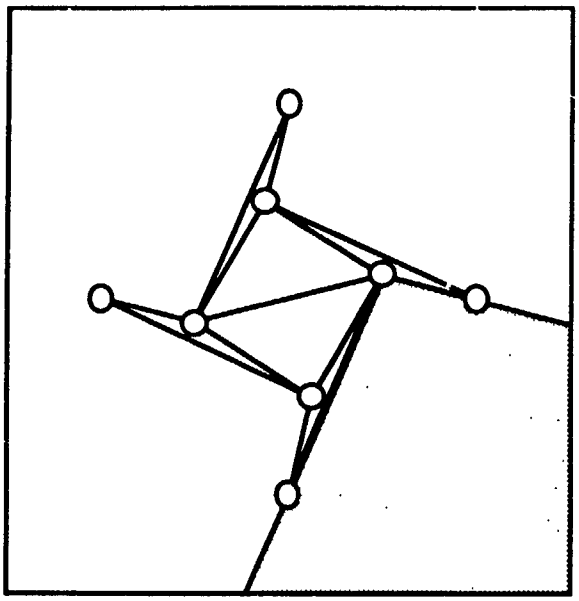
Output: A triangulated simple polygon in D.

Properties: Complete.

Complexity: $O(n^2)$, may not terminate.

The approach is based on the technique used in Algorithm 9.9 for constructing a random simple polygon by selecting a random edge to break and then selecting a random point that is visible from the endpoints of the edge. When computing the visibility polygons from the endpoints of the edge, we are performing two visibility queries for each endpoint, an internal visibility query and an

external visibility query. In this algorithm, we require the new point to be outside the polygon, so we need only perform the external visibility query (see Figure 10.12). We induce a triangulation of the simple polygon produced. Rather than replacing the broken edge, we simply make the broken edge be an internal edge of the triangulation.



Edge being split is bold. Gray area is region in which vertex may be placed.

Figure 10.12. Splitting an edge of a triangulated polygon.

Chapter 11

11. CONCLUSIONS

We have explored and developed a large set of algorithms to generate geometric objects at random. Although some of these generators require exponential time to provide a uniform distribution, a non-uniform distribution can typically be generated in a polynomial time complexity. For many applications, these polynomial time algorithms are practical in that they are fairly easy to implement while providing acceptable time complexity. For some types of geometric objects, we have developed algorithms that provide both a uniform probability distribution and a polynomial time complexity.

The following sections list the significant contributions to our theoretical knowledge of geometrical probability, as well as stating some of the open problems this research has led to.

11.1. CONCEPTS EXPLORED

We defined uniform coverage probability for geometric objects in general and constructed algorithms to generate intervals, arcs, simple interval sets, simple arc sets, convex polygons, and simple polygons with a uniform coverage probability. For example, we developed algorithms to generate simple interval sets or simple arc sets with uniform coverage probability in linear time. This is an important and practical probability distribution for many applications. In addition, we have shown that it is not possible to construct an algorithm that can produce any possible interval of a given fixed length in a given domain while providing a uniform coverage probability.

11.2. SUMMARY OF ALGORITHMS DEVELOPED

We have developed several algorithms that provide a uniform coverage probability in a bounded domain, as listed in Table 1.

Table 1. Summary of algorithms providing uniform coverage probability.

Geometric Object	Time Complexity	Coverage Probability	Algorithm
Arc	O(1)	$\frac{1}{2}$	Algorithm 7.2
Arc	O(1)	$\frac{L}{D_2 - D_1}$	Algorithm 7.3
Arc	O(1)	$\frac{L_{exp}}{D_2 - D_1}$	Algorithm 7.4
Interval	O(N)	$\frac{1}{N}$	Algorithm 7.6
Simple arc set of size n	O(n)	$\frac{1}{2}$	Algorithm 7.7
Simple interval set of size n	O(n)	$\frac{1}{2}$	Algorithm 7.8
Convex O(n)-gon	$O(n^2)$	unknown	Algorithm 9.6
Simple O(n)-gon	$O(n^3)$	unknown	Algorithm 9.10

Many algorithms take a triangulated simple polygon as input. To test such algorithms, a source of random triangulations is needed. Algorithm 10.7 for generating uniform random triangulations of a simple polygon in $O(n^4)$ time provides such a source, and is therefore a particularly important part of this thesis. If a uniform probability distribution is not required, Algorithm 10.6 takes only $O(n^2)$ time. See Table 2 for a summary of algorithms that provide a uniform probability distribution.

Table 3 provides a summary of algorithms that can generate all possible outputs. These algorithms typically execute in polynomial time, and are practical for many applications where no particular probability distribution is required.

Table 2. Summary of algorithms providing uniform probability distribution.

Geometric Object	Time Complexity	Algorithm
Arc	O(1)	Algorithm 7.1
Interval	O(1)	Algorithm 7.5
Point in rectangle	O(1)	Algorithm 6.1
Point in circle	O(1)	Algorithm 6.3
Point on circle	O(1)	Algorithms 6.4 - 6.7
Point in triangle	O(1)	Algorithm 6.8 - 6.9
Point in convex n-gon	O(n)	Algorithm 6.10
Point in simple n-gon	O(n)	Algorithm 6.11
Monotone n-gon on point set	$O(n^n)$	Algorithm 9.2
Convex n-gon on point set	O(n!)	Algorithm 9.7
Triangulation of convex n-gon	$O(n^2)$	Algorithm 10.1
Triangulation of convex n-gon	O(n)	Algorithm 10.2
Triangulation of simple n-gon	$O(n^4)$	Algorithm 10.7

Table 3. Summary of algorithms providing completeness.

Geometric Object Generated	Time Complexity	Algorithm
Point in circle	O(1)	Algorithm 6.2
Point set of size n	O(n)	Algorithm 6.12
Simple line segment set of size n	$O(n^2 \log n)$	Algorithm 8.1 - 8.2
Simple line segment set of size n	$O(n^3)$	Algorithm 8.3
Monotone n-gon on point set	$O(n^2)$	Algorithm 9.3
Convex O(n)-gon	O(n log n)	Algorithm 9.4
Convex n-gon	O(n)	Algorithm 9.5
Simple n-gon	$O(n^3)$	Algorithm 9.8
Simple n-gon	$O(n^2)$	Algorithm 9.9
Triangulation of point set of size n	$O(n^3)$	Algorithm 10.3 - 10.4
Triangulation of simple n-gon	$O(n^2)$	Algorithm 10.6
Triangulated simple n-gon on point set	$O(n^4)$	Algorithm 10.8
Triangulated simple n-gon	$O(n^2)$	Algorithm 10.9

11.3. OPEN PROBLEMS

In this section we list some open problems that this research has led to.

Can uniform random triangulations of a simple polygon be generated more efficiently than our $O(n^4)$ algorithm? As discussed in Section 2.3.3, an $O(n^2)$ time algorithm to construct maximal independent sets of circle graphs could lead to such an algorithm if it provided the intermediate results we need.

Can uniform random triangulations of a point set be generated in polynomial time? We have not found a property of point set intersection graphs, such as the circle graph property for polygon intersection graphs, which would allow the number of maximal independent sets to be computed in polynomial time. Does such a property exist? Given the ability to count point set triangulations in polynomial time, how can a uniform random triangulation generator be constructed?

REFERENCES

[Atkinson & Sack 92] M. D. Atkinson and Jörg-Rüdiger Sack, *Generating binary trees at random*, Information Processing Letters, Volume 41, pp. 21-23, North-Holland, 1992.

[Billingsley 86] Patrick Billingsley, *Probability and Measure*, John Wiley & Sons, Second edition, 1986.

[Bollobás 85] Béla Bollobás, *Random Graphs*, Academic Press, 1985.

[Bondy & Murty 76] John Adrian Bondy and U. S. R. Murty, *Graph Theory with Applications*, North-Holland, 1976.

[Chung 74] Kai Lai Chung, *Elementary Probability Theory with Stochastic Processes*, Springer-Verlag, 1974.

[Cormen et al. 90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill, 1990.

[Culberson & Rawlins 85] J. Culberson and G. Rawlins, *Turtlegons: generating simple polygons from sequences of angles*, Proceedings of the First ACM Symposium on Computational Geometry, pp. 305-310, 1985.

[Devroye 82] Luc Devroye, *On the computer generation of random convex hulls*, Computers and Mathematics with Applications, Volume 8, pp. 1-13, 1982.

[Devroye 86] Luc Devroye (McGill University), *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.

[Devroye 89] Luc Devroye, *private communication*, McGill University, November 1989.

[Devroye et al. 91] Luc Devroye, Peter Epstein, and Jörg-Rüdiger Sack, *Uniform Coverage Probability for Intervals*, manuscript, Carleton University, November 1991.

[Doe & Edwards 84] T. Doe and S. Edwards, *Course Project: Random generation of polygons*, 95.508, Carleton University, 1984.

[Drake 67] Alvin W. Drake, *Fundamentals of Applied Probability Theory*, McGraw-Hill, 1967.

[Edelsbrunner 87] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, W. Brauer, G. Rozenberg and A. Salomaa (Ed.), EATCS Monographs on Theoretical Computer Science 10, Springer-Verlag, 1987.

[Efron 65] Bradley Efron, *The convex hull of a random set of points*, Biometrika, Volume 52, pp. 331-343, 1965.

[ElGindy & Avis 81] H. ElGindy and D. Avis, *A linear algorithm for computing the visibility polygon from a point*, Journal of Algorithms, Volume 2, Number 2, pp. 186-197, 1981.

[Feller 68] William Feller, *An Introduction to Probability Theory and Its Applications*, John Wiley & Sons, Volumes 1 & 2, Third edition, 1968.

[Garey & Johnson 79] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.

[Gavril 73] F. Gavril, *Algorithms for a Maximum Clique and a Maximum Independent Set of a Circle Graph*, Networks, Volume 3, pp. 261-273, 1973.

[Halmos 74] Paul Richard Halmos, *Measure Theory*, Graduate Texts in Mathematics, Springer-Verlag, 1974.

[Kendall & Moran 63] M. G. Kendall and P. A. P. Moran, *Geometrical Probability*, Griffin's Statistical Monographs #10, Charles Griffin & Co., London, 1963.

[Knuth & Yao 76] Donald Ervin Knuth and Andrew C. Yao, *The Complexity of Non-Uniform Random Number Generation*, Algorithms and Complexity: New Directions and Recent Results, edited by J. F. Traub, Academic Press, pp. 357-428, 1976.

[Knuth 81] Donald Ervin Knuth (Stanford University), *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms, Second Edition, Addison-Wesley, 1981.

[Lee & Chen 85] D. T. Lee and I. M. Chen, *Display of visible edges of a set of convex polygons*, Computational Geometry, edited by G. T. Toussaint, North Holland, Amsterdam, 1985.

[May & Smith 82] Jerrold H. May and Robert L. Smith, *Random polytopes: their definition, generation, and aggregate properties*, Mathematical Programming, Volume 24, pp. 39-54, North-Holland, 1982.

[O'Rourke et al. 87] Joseph O'Rourke, Heather Booth, and Richard Washington, *Connect-the-dots: A new heuristic*, Computer Vision, Graphics, and Image Processing, Volume 39, Number 2, pp. 258-266, Academic Press, 1987.

[O'Rourke & Virmani 91] Joseph O'Rourke and M. Virmani, *Generating random polygons*, Smith Technical Report 11, August 1991.

[Preparata & Shamos 85] Franco P. Preparata and Michael Ian Shamos, *Computational Geometry: An Introduction*, D. Gries (Ed.), Texts and Monographs in Computer Science, Springer-Verlag, New York, 1985.

[Ramberg & Tadikamalla 78] J. S. Ramberg and P. R. Tadikamalla, *On the generation of subsets of order statistics*, Journal of Statistical Computation and Simulation, Richard G. Krutchkoff (Ed.), Volume 6, pp. 239-241, 1978.

[Rau-Chaplin 91] Andrew Rau-Chaplin, *private communication*, Carleton University, 1991.

[Reingold et al. 77] E. M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice-Hall, 1977.

[Reinhardt & Soeder 74] Fritz Reinhardt and Heinrich Soeder, *dtv-Atlas zur Mathematik Tafeln und Texte*, Volumes 1 & 2, in German, Deutsher Taschenbuch Verlag, 1974.

[Richter 92] Bruce Richter, *private communication*, Carleton University, March 1992.

[Sack 84] Jörg-Rüdiger Sack, *Rectilinear Computational Geometry*, Ph.D. Thesis, McGill University; Technical Report SCS-TR-54, Carleton University School of Computer Science, June 1984.

[Trivedi 82] Kishor Shridharbhai Trivedi (Duke University), *Probability and Statistics with Reliability*, Queuing, and Computer Science Applications, Prentice-Hall, New Jersey, 1982.

[Urrutia 80] Jorge Urrutia Galicia, *Intersection Graphs of Some Families of Plane Curves*, Ph.D. Thesis, University of Waterloo, 1980.

END

24 05 93

FIN