THE EXPECTED DISTANCE BETWEEN TWO RANDOM POINTS IN A POLYGON

by

Arthur C. Hsu

Submitted to the Department of Civil Engineering
in partial fulfillment of the requirements for the degrees of

MASTER OF SCIENCE IN TRANSPORTATION

and

BACHELOR OF SCIENCE IN CIVIL ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1990

Signature of Author .....
Department of Civil Engineering
January 30, 1990

Certified by .....
Amedeo R. Odoni
Professor of Aeronautics and Astronautics and Civil Engineering
Codirector, Operations Research Center
Thesis Supervisor

Accepted by..... .....
Ole S. Madsen
Chairman, Departmental Committee on Graduate Students

THE EXPECTED DISTANCE BETWEEN TWO RANDOM POINTS IN A POLYGON

by

Arthur C. Hsu

Submitted to the Department of Civil Engineering
on January 30, 1990 in partial fulfillment
of the requirements for the degrees of
Master of Science in Transportation and
Bachelor of Science in Civil Engineering

Abstract

This thesis is a study of the problem of finding the expected value of the distance between two uniformly, identically, and independently distributed random points in a polygonal region. The measures of distance considered are the Euclidean (or straight-line) distance metric and the Manhattan (or rectangular) distance metric.

Separate numerical methods are developed for finding the expected distance in a convex polygon using the Euclidean distance metric and for finding the expected distance in any polygon using the Manhattan distance metric. Both methods employ a "divide-and-conquer" strategy which divides the polygon into regions that are mathematically tractable so that analytic expressions can be derived in closed form.

The methods have been implemented in computer programs, and numerical examples are provided.

Thesis Supervisor: Amedeo R. Odoni
Title: Professor of Aeronautics and Astronautics and Civil Engineering
Codirector, Operations Research Center

Acknowledgments

I would like to express my deepest appreciation to Professor Amedeo R. Odoni for stimulating my interest in operations research and for his guidance, encouragement, and patience in supervising this thesis.

My thanks go to my family and friends who share my joy and relief that this thesis is finally complete. Most of all, I am indebted to my parents, Pei Tung and Lai San.

Contents

Chapter 1

Introduction

1.1 Motivation

This thesis is a study of the problem of finding the expected value of the distance between two uniformly, identically, and independently distributed random points in a polygonal region. The problem of finding expected distances is a common one in distribution management, transportation systems analysis, and urban operations research. The expected distance may be used to estimate average trip lengths or the average service time for urban response systems. The result that we are seeking can be applied in any situation where there is a point source and point sink for goods or services such that the two points are uniformly and independently distributed in a region.

In the next section, we will define the problem more precisely.

1.2 Defining the problem

Before we can discuss the problem of finding the "expected distance" between two "random points" in a "polygon", it is necessary to precisely define some of these terms. The purpose of this section is to formally define these terms and, in so doing, establish a system of notation to be used throughout this thesis.

Polygon

Our definition of a polygon is taken from a geometry text: it is "a simple closed curve which is the union of line segments. Each of the line segments is a side of the polygon and each of the endpoints of the line segments is a vertex of the polygon." [1] In the context of this thesis, where the phrase "in a polygon" would be more correctly phrased as "within a polygonal region". The distinction is that a polygon is simply the boundary of a region, whereas a polygonal region is the union of the polygon and its interior.

We will denote the polygonal region by the symbol Q and we will designate n to be the number of sides in the polygon. A polygon is defined by the set of vertex points $\{v_1, v_2, \ldots, v_n\}$. The set of vertex points is ordered so that the sequence represents the arrangement of points on the perimeter as we follow the perimeter in the counter-clockwise direction. (A polygon with its points arranged in this order is often referred to as a "right-oriented" polygon.)

The "sides" of the polygon are the line segments that connect adjacent vertex points. The side S_i represents the line segment connecting the vertex points v_i and v_{i+1}.

$$S_i = \{t \cdot v_i + (1-t) \cdot v_{i+1} : 0 \leq t \leq 1\}$$

(Note that the sequence of vertex points and the sequence of sides is circular (modulo n) in the sense that the vertices adjacent to v_n are v_{n-1} and v_1. Similarly, the vertices adjacent to v_1 are v_n and v_2.)

Random point

The term "random point" refers to a "point that is randomly and uniformly distributed within a polygon such that any two random points are by definition also statistically independent". It is perhaps less unwieldy to define "two random points" as meaning "two uniformly, identically and independently distributed random points".

We will use the probability density function $f_Q(q)$ to denote the distribution of a random point q in a polygon Q. In the polar coordinate system, for a point located at (r, θ) that is

[1] Merlyn J. Behr and Dale G. Jungst, *Fundamentals of Elementary Mathematics: Geometry*, (New York: Academic Press, 1972), p. 121.

uniformly distributed,

$$\mathrm{f}_Q(r,\theta) = \frac{r}{\mathrm{Area}(Q)}$$

where $\mathrm{Area}(Q)$ is the area of the polygon Q. In the Cartesian coordinate system, a uniform distribution for a point located at (x, y) is expressed by:

$$\mathrm{f}_Q(x,y) = \frac{1}{\mathrm{Area}(Q)}$$

Expected distance

The term "expected distance" means the "expected value of the distance". If $\mathrm{d}(q_1, q_2)$ is the measure of distance between points q_1 and q_2, and $f_Q(q)$ is the probability density function that distributes the points in a region Q, then the expected distance $\bar{\mathrm{D}}(Q)$ between the two points is:

$$\bar{\mathrm{D}}(Q) = \iint_Q \iint_Q \mathrm{f}_Q(q_1)\mathrm{f}_Q(q_2)\mathrm{d}(q_1,q_2)dq_2dq_1$$

The measure of the distance requires further elaboration. Distances are measured according to a "distance metric" and in this thesis, we will consider the Manhattan distance metric and the Euclidean distance metric. The Manhattan distance, also known as the rectangular distance or the right-angle distance, is one that is measured along the x and y axes. We can express the Manhattan distance $\mathrm{d}_M(x_1, y_1, x_2, y_2)$ in the Cartesian coordinate system between two points (x_1, y_1) and (x_2, y_2) as:

$$\mathrm{d}_M(x_1,y_1,x_2,y_2) = |x_1 - x_2| + |y_1 - y_2|$$

The Euclidean distance is also known as the straight-line distance. In the Cartesian coordinate system, the Euclidean distance $\mathrm{d}_E(x_1, y_1, x_2, y_2)$ is expressed as:

$$\mathrm{d}_E(x_1,y_1,x_2,y_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The distance expressions above represent "direct distances" in the sense that they represent the length of the path between two points if there is a direct, unobstructed path between the two points. However, if barriers to paths are considered, the length of the path

may be somewhat greater. In fact, in this thesis, we will assume that a path may not cross the perimeter of the polygonal region. In other words, the path between two points in a polygonal region must itself be interior to the polygonal region.

In the Euclidean metric, the path between every pair of points in a polygon is "direct" if and only if the polygon is convex. We define a convex polygon to be a polygon where every vertex in the polygon is a convex vertex. A convex vertex is a vertex where the angle on the interior side of the polygon formed between the sides adjacent to the vertex is less than or equal to 180°. A concave polygon is a polygon that is not convex.

Unlike the Euclidean metric, there may be more than one "direct" path between a pair of points in the Manhattan metric where the length of the direct path is expressed by $d_M(q_1, q_2)$ earlier. In fact, the path between any pair of points using the Manhattan metric may be "direct" in certain concave polygons. However, in the Manhattan metric, the path between every pair of points in a polygon is "direct" if and only if the polygon is monotone with respect to the x and y axes. We define a polygon to be "monotone" with respect to an axis if the perimeter of the polygon forms two envelopes that are each monotone with respect to that axis. The perimeter of the polygon forms an envelope that is monotone to an axis if the projections of the vertices on the perimeter onto the axis is ordered the same as in the perimeter. [2] The implications of a direct path and the property of monotonicity will be discussed in more detail in Chapter 2.

For a pair of points where there can be no direct path without crossing the side of the polygon, the shortest path will pass by the intermediate points I_k where each of the intermediate points are concave vertices of the polygon. The length of an indirect path in the Cartesian coordinate system for either of the two distance metrics is:

$$\begin{aligned} \mathrm{d}(x_1, y_1, x_2, y_2) = \; & \mathrm{d}(x_1, y_1, I_{X,1}, I_{Y,1}) + \mathrm{d}(I_{X,m}, I_{Y,m}, x_2, y_2) + \\ & \sum_{k=1}^{m-1} \mathrm{d}(I_{X,k}, I_{Y,k}, I_{X,k+1}, I_{Y,k+1}) \end{aligned}$$

Because of the fact that we are considering the sides of the polygon to be barriers, the term "expected distance" is more precisely expressed as the "expected value of the length of the shortest interior path".

[2] Joseph O'Rourke, *Art Gallery Theorems and Algorithms*, (New York: Oxford University Press, 1987), p. 14.

1.3 Outline of the Contents

In this chapter, we have provided some provided some motivation for finding the expected distance between two random points, and have given a precise definition of the problem. In Chapter 2, we will study the expected distance between two random points in a polygon using the Manhattan distance metric. In Chapter 3, we will focus on the Euclidean distance metric for convex polygons only. We summarized our results and offer suggestions for further investigation in the concluding chapter.

The numerical methods that we present in Chapters 2 and 3 have been implemented in computer programs. Appendix A contains the listings for the programs as well as some numerical examples. Appendix B discusses the problem of finding the expected distance using the Euclidean distance metric in concave polygons.

Chapter 2

The Expected Manhattan Distance in a Polygon

In this chapter, we will present a method for finding the expected distance between two random points in a polygon using the Manhattan distance metric.

2.1 A general approach

From the general expected distance formula from Chapter 1, we state that the expected Manhattan distance $\bar{\mathrm{D}}_M(Q)$ between two randomly distributed points in a polygon Q is:

$$\bar{\mathrm{D}}_M(Q) = \iint_Q \iint_Q \mathrm{f}_Q(q_1)\mathrm{f}_Q(q_2)\mathrm{d}_M(q_1, q_2)dq_2dq_1 \tag{2.1}$$

where $\mathrm{d}_M(q_1, q_2)$ is the distance of the shortest path in the Manhattan metric between the random points q_1 and q_2 in the polygon Q, and $\mathrm{f}_Q(q)$ is the density function for distributing a random point in the polygon. For the Manhattan metric, the most appropriate coordinate system for distributing the random points is the Cartesian coordinate system.

$$q_1 \equiv (x_1, y_1), \qquad q_2 \equiv (x_2, y_2)$$

The density function $\mathrm{f}_Q(q)$ is in this case:

$$\mathrm{f}_Q(q_1) = \frac{1}{\mathrm{Area}(Q)}, \qquad \mathrm{f}_Q(q_2) = \frac{1}{\mathrm{Area}(Q)}$$

Let us assume for the moment that the polygon Q is monotone with respect to both the x and y axes. (Refer to section 1.2 for the definition of a "monotone" region.) Then the path between any pair of random points in the polygon is "direct" and the distance between the two random points is:

$$d_M(q_1, q_2) = d_M(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$$

The expression for the expected Manhattan distance from Equation 2.1 becomes:

$$\bar{D}_M(Q) = \frac{\iint_Q \iint_Q |x_1 - x_2| + |y_1 - y_2| \, dy_2 dx_2 dy_1 dx_1}{\text{Area}^2(Q)} \tag{2.2}$$

We can take advantage of the fact that the x component of the distance is independent of the y component:

$$\bar{D}_M(Q) = \frac{\iint_Q \iint_Q |x_1 - x_2| \, dy_2 dx_2 dy_1 dx_1 + \iint_Q \iint_Q |y_1 - y_2| \, dy_2 dx_2 dy_1 dx_1}{\text{Area}^2(Q)}$$

which is equivalent to:

$$\bar{D}_M(Q) = \bar{D}_X(Q) + \bar{D}_Y(Q) \tag{2.3}$$

where:

$$\bar{D}_X(Q) = \frac{1}{\text{Area}^2(Q)} \iint_Q \iint_Q |x_1 - x_2| \, dy_2 dx_2 dy_1 dx_1$$

$$\bar{D}_Y(Q) = \frac{1}{\text{Area}^2(Q)} \iint_Q \iint_Q |y_1 - y_2| \, dy_2 dx_2 dy_1 dx_1$$

The purpose of separating the expression into two subexpressions in Equation 2.3 is that each of the subexpressions is much easier to evaluate than the whole expression. Furthermore, the two subexpressions are so similar that we can express $\bar{D}_Y(Q)$ in terms of $\bar{D}_X(Q)$:

$$\bar{D}_Y(Q) = \bar{D}_X(Q^T)$$

where Q^T may be defined as the polygon Q that has been rotated $\pm 90°$ about any arbitrary point, or as the reflection of Q about the line $y = x + c$ for any arbitrary constant c. For computational simplicity, we will define Q^T as the "transpose" or reflection of Q about the $y = x$ line. In other words, for every point (x, y), we replace the x value by the y value, and the y value by the x value.

$$(x, y)^T \equiv (y, x)$$

We illustrate the effect of transposing a polygon in Figure 2-1. Mathematically, we can

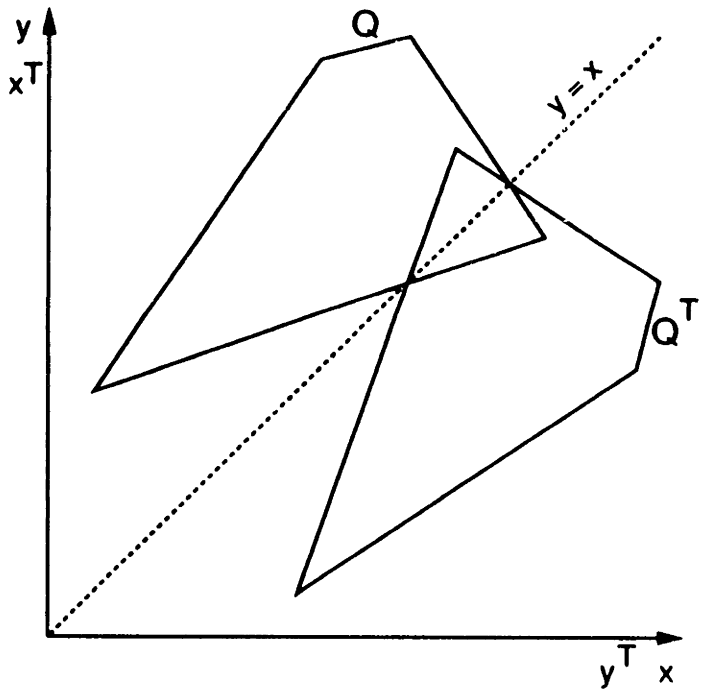

Figure 2-1: Q^T is the polygon Q transposed about the $y = x$ line

prove that $\bar{\mathrm{D}}_Y(Q) = \bar{\mathrm{D}}_X(Q^T)$ as follows:

$$\bar{\mathrm{D}}_Y(Q) = \frac{1}{\mathrm{Area}^2(Q)} \iint_Q \iint_Q |y_1 - y_2| \, dy_2 dx_2 dy_1 dx_1$$

$$\bar{\mathrm{D}}_Y(Q) = \frac{1}{\mathrm{Area}^2(Q)} \iint_Q \iint_Q |y_1 - y_2| \, dx_2 dy_2 dx_1 dy_1$$

To transpose each point in the polygon about the $y = x$ line, we substitute x by y^T, and y by x^T:

$$\bar{D}_Y(Q) = \frac{1}{\text{Area}^2(Q)} \iint_{Q^T} \iint_{Q^T} \left| x_1^T - x_2^T \right| dy_2^T dx_2^T dy_1^T dx_1^T$$

Since the area of a polygon Area(Q) is the same as the area of the transposed polygon Area(Q^T),

$$\bar{D}_Y(Q) = \frac{1}{\text{Area}^2(Q^T)} \iint_{Q^T} \iint_{Q^T} \left| x_1^T - x_2^T \right| dy_2^T dx_2^T dy_1^T dx_1^T$$

and:

$$\bar{D}_Y(Q) = \bar{D}_X(Q^T)$$

In deriving the result in Equation 2.3, we showed that the distance formula for $d_M(q_1, q_2)$ is such that the x component and y component are independent and could be separated mathematically, but under the assumption that the polygon is monotone with respect to both the x and y axes. We now turn to the case where the two random points are in a polygon that is not monotone, so that a direct path between two random points might not be possible due to barriers caused by the sides in a concave polygon.

However, we will show that Equation 2.3 is general even for polygons that are not monotone. We showed in section 1.2 that where the path between two random points was "indirect", the length of the shortest path is expressed by:

$$\begin{aligned} d_M(x_1, y_1, x_2, y_2) = \; & |x_1 - I_{X,1}| + |y_1 - I_{Y,1}| + |x_2 - I_{X,m}| + |y_2 - I_{Y,m}| + \\ & \sum_{k=1}^{m-1} \left(|I_{X,k} - I_{X,k+1}| + |I_{Y,k} - I_{Y,k+1}| \right) \end{aligned}$$

where $I_{X,i}$ and $I_{Y,i}$ are the x coordinate and y coordinate respectively of the m intermediate vertex points in the path from (x_1, y_1) to (x_2, y_2). (Figure 2-2 shows a path between two points that is indirect. With two intermediate vertex points in the path, there are three pairs of x component and y component segments in the path.) Once again, we can separate the length of the path into an x component and a y component which are independent of one another:

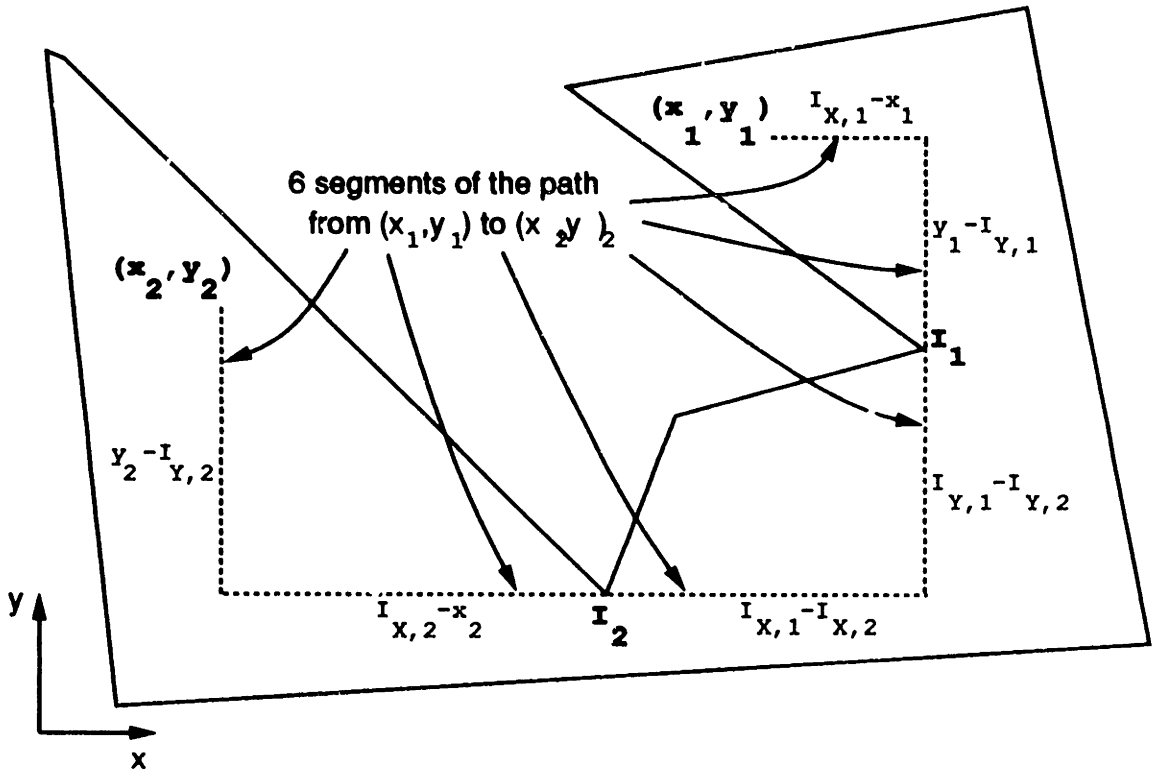

Figure 2-2: Path from (x_1, y_1) to (x_2, y_2) must go around intermediate vertex points I_1 and I_2

$$
\begin{aligned}
d_M(x_1, y_1, x_2, y_2) &= d_X(x_1, y_1, x_2, y_2) + d_Y(x_1, y_1, x_2, y_2) \\
d_X(x_1, y_1, x_2, y_2) &= |x_1 - I_{X,1}| + |x_2 - I_{X,m}| + \sum_{k=1}^{m-1} |I_{X,k} - I_{X,k+1}| \\
d_Y(x_1, y_1, x_2, y_2) &= |y_1 - I_{Y,1}| + |y_2 - I_{Y,m}| + \sum_{k=1}^{m-1} |I_{Y,k} - I_{Y,k+1}|
\end{aligned}
\tag{2.4}
$$

Because the same conditions exist that were necessary for our earlier proof of Equation 2.3, we can state that the y component of the expected Manhattan distance of a general polygon can also be calculated by finding the x component of the expected distance of the transpose of the polygon.

In short, we have shown that we can evaluate the expected Manhattan distance in any polygon by finding its x and y components independently. Furthermore, any method that we develop to evaluate one component can be used to evaluate the other component. Our next step is to develop such a method, and we arbitrarily choose to evaluate the x component.

2.2 The x component of the expected Manhattan distance in a polygon that is monotone with respect to the x axis

We will attempt to evaluate the x component $\bar{D}_X(Q)$ of the expected Manhattan distance in a polygon. In order that we may begin with Equation 2.3, let us consider polygons that are monotone with respect to the x axis. To reduce verbiage, we will call these polygons "x-monotone" polygons.

$$\bar{D}_X(Q) = \frac{1}{\text{Area}^2(Q)} \iint_Q \iint_Q |x_1 - x_2| \, dy_2 dx_2 dy_1 dx_1$$

For a polygon bounded in the x dimension between x_l and x_r and enveloped in the y dimension by the functions $bottom(x)$ and $top(x)$, we have:

$$\bar{D}_X(Q) = \frac{1}{\text{Area}^2(Q)} \int_{x_l}^{x_r} \int_{bottom(x_1)}^{top(x_1)} \int_{x_l}^{x_r} \int_{bottom(x_2)}^{top(x_2)} |x_1 - x_2| \, dy_2 dx_2 dy_1 dx_1 \qquad (2.5)$$

Even for the simple polygon shown in Figure 2-3, it is not easy to express the envelope

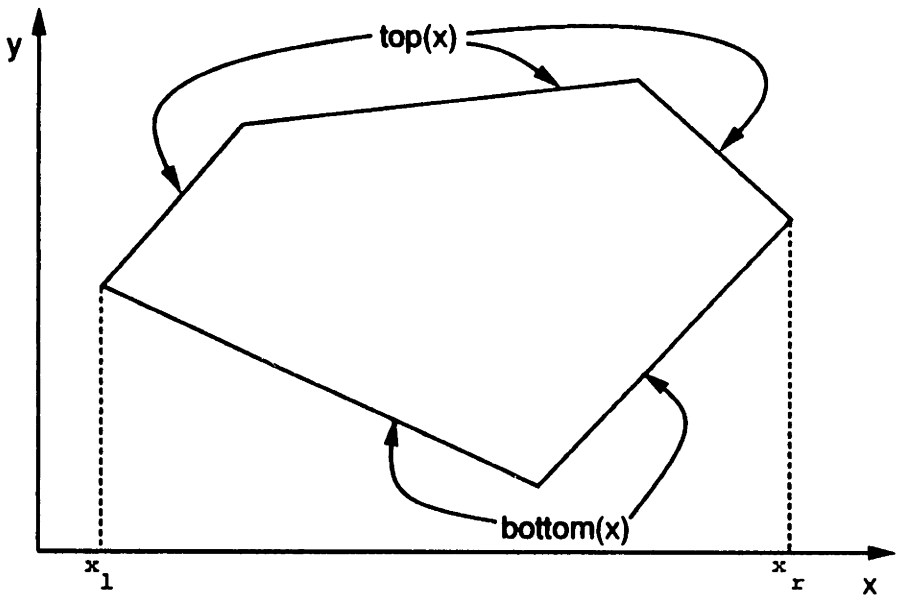

Figure 2-3: An x-monotone polygon bounded by x_l, x_r, $bottom(x)$ and $top(x)$

functions $bottom(x)$ and $top(x)$. Rather than evaluating $\bar{D}_X(Q)$ in one expression, we will attempt to divide the polygon into convenient pieces that can be evaluated individually.

2.2.1 Dividing the x-monotone polygon

We observe in Figure 2-3 that the envelope functions are linear functions between vertex points on the envelope. The fact that we can easily express linear functions suggests that we should divide the polygon at the vertex points. Figure 2-4 shows lines at $x = v_{X,i}$ for all vertex points $i = 1, \ldots, n$ in an n-side polygon, where $v_{X,i}$ is the x-coordinate of vertex point i. We use these lines to partition the polygon into "zones". We claim that the leftmost

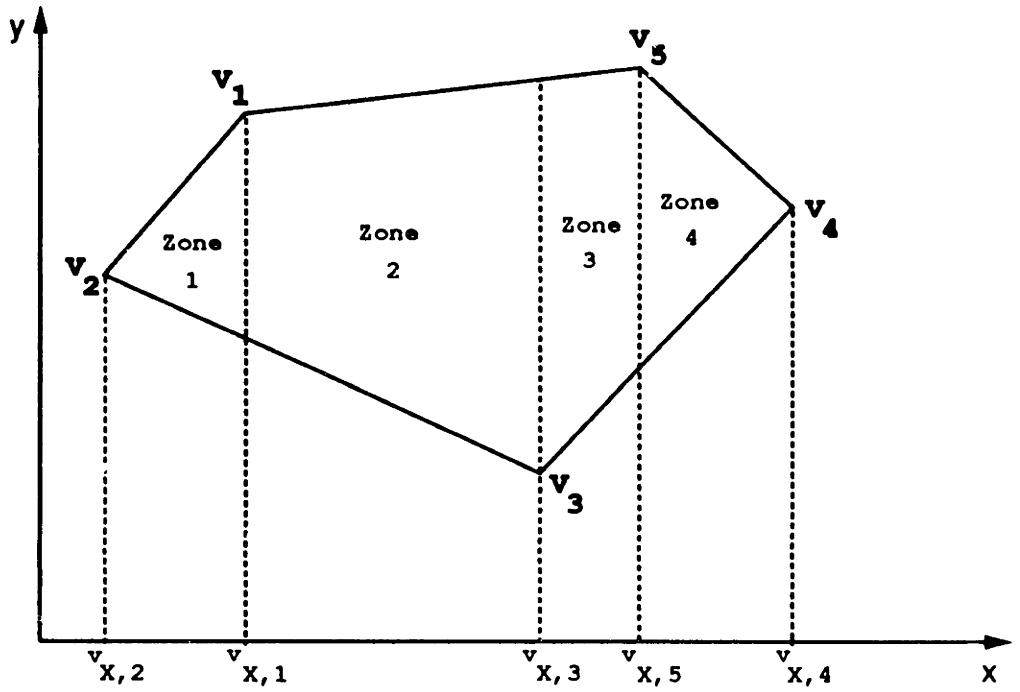

Figure 2-4: An x-monotone polygon partitioned by lines on $x = v_{X,i}$ into "zones"

line (*i.e.* the one with the smallest value of x) is at x_l, and the rightmost line is at x_r. In other words, the entire polygon is bounded in the x dimension between the minimum x value of all the vertex points and the maximum x value of all the vertex points.

$$x_l = \min_i v_{X,i}, \text{ for } i = 1, \ldots, n$$

$$x_r = \max_i v_{X,i}. \text{ for } i = 1, \ldots, n$$

This is due to the fact that the boundary of a polygon consists of straight line segments, and so there can be no part of a polygon that extends in the x dimension beyond the leftmost

(or rightmost) vertex. (This argument is identical to the contention in linear programming that when maximizing one linear objective function (in this case, the magnitude of x), the maximum value must occur at some corner point if the region is bounded by linear contraints, which is true of any polygon.) Thus, the polygon is enclosed between the two lines at $x = v_{X,i}$ having the lowest and highest values of x.

Given that the polygon is enclosed between the two extreme lines at $x = v_{X,i}$, the remaining $n - 2$ lines divide the polygon into $n - 1$ zones. Even when two of the lines are coincident (which occurs when two vertex points happen to have the same x value), we say that there is a zone between them (albeit a zone having an area 0). Each zone is bounded on the left and right by two of the lines at $x = v_{X,i}$. Each zone is bounded on the bottom and the top by the envelope functions of the polygon. Note that the top and bottom of each zone are line segments because the envelope functions of the polygon are necessarily linear between each pair of adjacent lines at $x = v_{X,i}$. Thus, each zone is trapezoidal with the parallel sides formed by two of the lines at $x = v_{X,i}$, and the other two lines formed by the bottom and top envelope of the polygon. (The leftmost and rightmost zones are triangular, but these are still trapezoids with the property that one of the parallel sides has length 0.)

In order that we can express the polygon in terms of zones, let us define P_i such that $\{P_1, P_2, \ldots, P_n\}$ is the sequence of vertex points ranked by their x value. Then, if we let $p_{X,i}$ be the x value of the vertex point P_i,

$$p_{X,1} \leq p_{X,2} \leq \cdots \leq p_{X,n}$$

Let us also define $p_{B,i}$ and $p_{T,i}$ to be the y value of the bottom and top envelopes respectively of the polygon at $x = p_{X,i}$. Because the envelopes are linear in the x domain of the zone, the bottom envelope function for a zone i is:

$$bottom(x) = p_{B,i} + \frac{x - p_{X,i}}{p_{X,i+1} - p_{X,i}} (p_{B,i+1} - p_{B,i})$$

and the top envelope function for zone i is similarly:

$$top(x) = p_{T,i} + \frac{x - p_{X,i}}{p_{X,i+1} - p_{X,i}} (p_{T,i+1} - p_{T,i})$$

Figure 2-5 illustrates a zone i, which lies between $x = p_{X,i}$ and $x = p_{X,i+1}$ that is defined

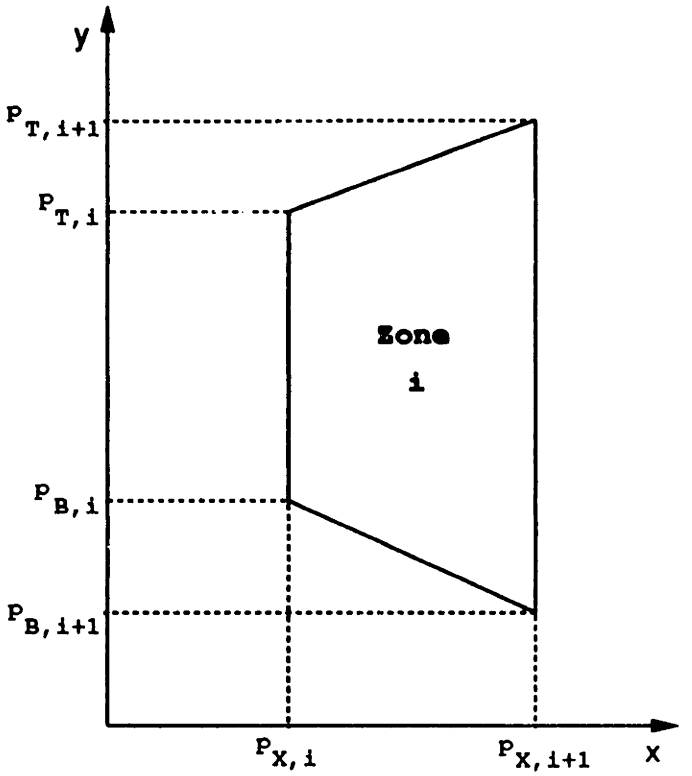

Figure 2-5: The corners of zone i defined by $p_{X,i}$, $p_{B,i}$, $p_{T,i}$, $p_{X,i+1}$, $p_{B,i+1}$ and $p_{T,i+1}$

by $p_{X,i}$, $p_{B,i}$, $p_{T,i}$, $p_{X,i+1}$, $p_{B,i+1}$ and $p_{T,i+1}$. Thus, one possible way to generate all zones would be to find the quantities $p_{X,i}$, $p_{B,i}$, and $p_{T,i}$ for all i ($i \in 1 \ldots n$).

We will briefly describe a method for finding $p_{X,i}$, $p_{B,i}$, and $p_{T,i}$.

1. To find $p_{X,i}$, we rank all vertex points by their x value so that we have a sequence of vertex points $\{P_1, P_2, \ldots, P_n\}$. The value of $p_{X,i}$ is by definition the x value of the point P_i.

2. The bottom and top envelope values for vertices P_1 and P_n are simply the y values of the respective vertices.

$$p_{B,1} = p_{T,1} = p_{Y,1}$$

$$p_{B,n} = p_{T,n} = p_{Y,n}$$

Finding the envelope values $p_{B,i}$ and $p_{T,i}$ for the other vertices depends on which envelope the vertex P_i lies on.

3. If P_i lies on the bottom envelope, then $p_{B,i}$ is the y value of the P_i.

$$p_{B,i} = p_{Y,i}$$

The top envelope value $p_{T,i}$ can be found by interpolating the y value between the nearest vertex on the top envelope to the left of $p_{X,i}$ and the nearest vertex on the top envelope to the right of $p_{X,i}$. In Figure 2-6, the nearest vertex on the top envelope to the left of $p_{X,i}$ is called P_l and the vertex on the top envelope to the right is called P_r. (Note that depending on the polygon, P_l is not necessarily P_{i-1}, and P_r is not necessarily P_{i+1}.) The interpolation formula is:

$$p_{T,i} = p_{Y,l} + \frac{p_{X,i} - p_{X,l}}{p_{X,r} - p_{X,l}} (p_{Y,r} - p_{Y,l})$$

4. On the other hand, if the vertex point P_i is on the top envelope, then:

$$p_{T,i} = p_{Y,i}$$

and the bottom envelope value $p_{B,i}$ can be found similarly by interpolating between the y values of two vertex points on the bottom envelope immediately to the left and right of $p_{X,i}$.

We can re-state Equation 2.5 so that the limits of the integrals that define the polygon are in terms of our definitions for the $n-1$ zones:

$$\bar{\mathrm{D}}_X(Q) = \frac{\displaystyle\sum_{i=1}^{n-1} \int_{p_{X,i}}^{p_{X,i+1}} \int_{p_{B,i}+\frac{x_1 - p_{X,i}}{p_{X,i+1} - p_{X,i}}(p_{B,i+1} - p_{B,i})}^{p_{T,i}+\frac{x_1 - p_{X,i}}{p_{X,i+1} - p_{X,i}}(p_{T,i+1} - p_{T,i})} \sum_{j=1}^{n-1} \int_{p_{X,j}}^{p_{X,j+1}} \int_{p_{B,j}+\frac{x_2 - p_{X,j}}{p_{X,j+1} - p_{X,j}}(p_{B,j+1} - p_{B,j})}^{p_{T,j}+\frac{x_2 - p_{X,j}}{p_{X,j+1} - p_{X,j}}(p_{T,j+1} - p_{T,j})} |x_1 - x_2| \, dy_2 dx_2 dy_1 dx_1}{\mathrm{Area}^2(Q)}$$

Moving the summations to the left of the expression,

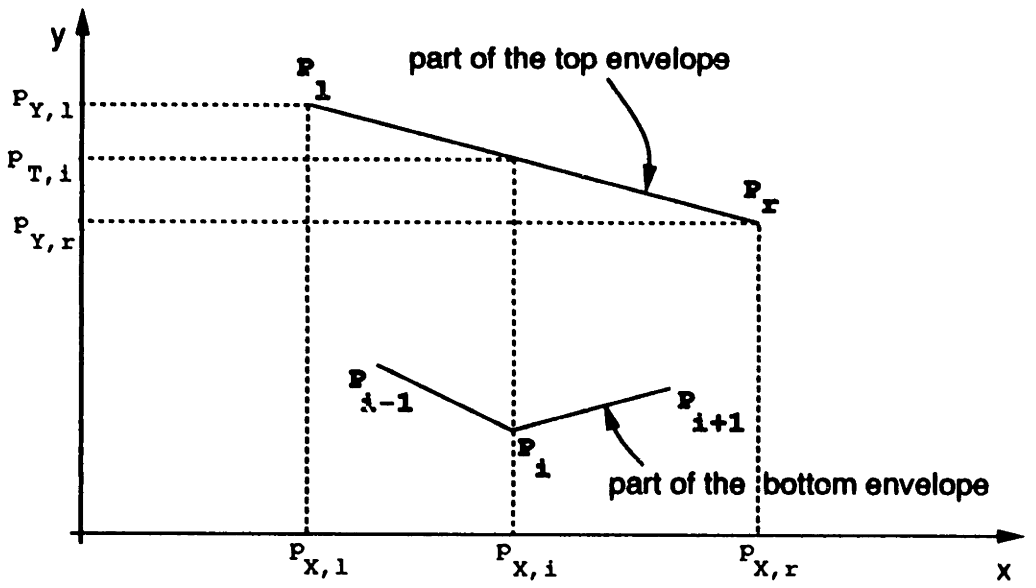

Figure 2-6: P_l and P_r are the nearest vertices on the top envelope to the left and right of $p_{X,i}$

$$\bar{D}_X(Q) = \frac{\sum_{i=1}^{n-1}\sum_{j=1}^{n-1} \int_{p_{X,i}}^{p_{X,i+1}} \int_{p_{B,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})}^{p_{T,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})} \int_{p_{X,j}}^{p_{X,j+1}} \int_{p_{B,j}+\frac{x_2-p_{X,j}}{p_{X,j+1}-p_{X,j}}(p_{B,j+1}-p_{B,j})}^{p_{T,j}+\frac{x_2-p_{X,j}}{p_{X,j+1}-p_{X,j}}(p_{T,j+1}-p_{T,j})} |x_1 - x_2|\, dy_2 dx_2 dy_1 dx_1}{\text{Area}^2(Q)}$$

We can distinguish between two scenarios within the above expression – where i and j are the same (hence the points (x_1, y_1) and (x_2, y_2) are in the same zone), and where i and j are different (hence the points (x_1, y_1) and (x_2, y_2) are in different zones).

$$\bar{D}_X(Q) = \frac{1}{\text{Area}^2(Q)} \left(\sum_{i=1}^{n-1} Z_{\bowtie}(i) + \sum_{i=1}^{n-1}\sum_{j=1, j\neq i}^{n-1} Z_{\diamond\!\!\diamond}(i,j) \right) \tag{2.6}$$

where the "intrazonal distance" factor $Z_{\bowtie}(i)$ is expressed by:

$$Z_{\bowtie}(i) = \int_{p_{X,i}}^{p_{X,i+1}} \int_{p_{B,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})}^{p_{T,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})} \int_{p_{X,i}}^{p_{X,i+1}} \int_{p_{B,i}+\frac{x_2-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{B,i+1}-p_{B,i})}^{p_{T,i}+\frac{x_2-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})} |x_1 - x_2|\, dy_2 dx_2 dy_1 dx_1$$

and the "interzonal distance" factor $Z_{\diamond\!\!\diamond}(i,j)$ is expressed by:

$$Z_{\bowtie}(i,j) = \int_{p_{X,i}}^{p_{X,i+1}} \int_{p_{B,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{B,i+1})}^{p_{T,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})} \int_{p_{X,j}}^{p_{X,j+1}} \int_{p_{B,j}+\frac{x_2-p_{X,j}}{p_{X,j+1}-p_{X,j}}(p_{T,j+1}-p_{B,j})}^{p_{T,j}+\frac{x_2-p_{X,j}}{p_{X,j+1}-p_{X,j}}(p_{T,j+1}-p_{T,j})} |x_1 - x_2| \, dy_2 dx_2 dy_1 dx_1$$

We will devote the next subsection to evaluating the expressions for the two factors.

2.2.2 The intrazonal and interzonal distance factor expressions

We begin by evaluating the intrazonal distance factor $Z_{\bowtie}(i)$. For a particular zone i, let us define the quantity l_i for the length of the left edge, r_i for the length of the right edge, and w_i for the width. Figure 2-7 illustrates the quantities, and we can express them in terms of

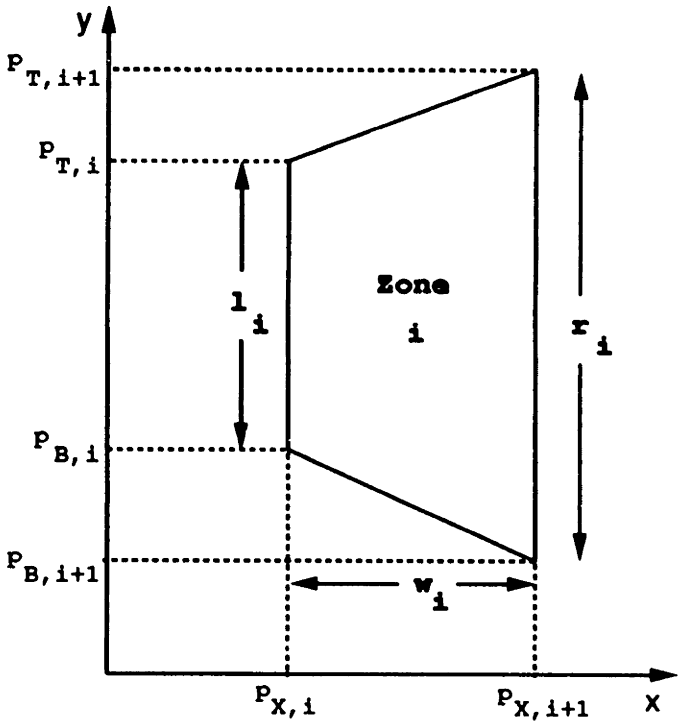

Figure 2-7: A zone showing l_i, r_i, and w_i

previous definitions. (For convenience, we also define the quantity c_i.)

$$l_i = p_{T,i} - p_{B,i} \tag{2.7}$$

$$r_i = l_{i+1} = p_{T,i+1} - p_{B,i+1}$$

$$w_i = p_{X,i+1} - p_{X,i}$$

$$c_i = \frac{r_i - l_i}{w_i}$$

Starting with the expression for $Z_{\bowtie}(i)$ from Equation 2.6, we can remove the absolute value function in the integrand $|x_1 - x_2|$ by dividing the integration over x_2 into two subranges: one where x_2 is less than x_1 (and thus is in the subrange $[p_{X,i}; x_1]$), and another where x_2 is greater than x_1 (in the subrange $[x_1; p_{X,i+1}]$).

$$Z_{\bowtie}(i) = \int_{p_{X,i}}^{p_{X,i}+w_i} \int_{p_{B,i}+\frac{x_1-p_{X,i}}{w_i}(p_{T,i+1}-p_{T,i})}^{p_{T,i}+\frac{x_1-p_{X,i}}{w_i}(p_{T,i+1}-p_{T,i})} \left(\begin{array}{l} \int_{p_{X,i}}^{x_1} \int_{p_{B,i}+\frac{x_2-p_{X,i}}{w_i}(p_{B,i+1}-p_{B,i})}^{p_{T,i}+\frac{x_2-p_{X,i}}{w_i}(p_{T,i+1}-p_{T,i})} (x_1 - x_2)\, dy_2 dx_2 \\ \int_{x_1}^{p_{X,i}+w_i} \int_{p_{B,i}+\frac{x_2-p_{X,i}}{w_i}(p_{B,i+1}-p_{B,i})}^{p_{T,i}+\frac{x_2-p_{X,i}}{w_i}(p_{T,i+1}-p_{T,i})} (x_2 - x_1)\, dy_2 dx_2 \end{array} + \right) dy_1 dx_1$$

By substituting $\hat{x_2} + p_{X,i}$ for x_2, and substituting $\hat{x_1} + p_{X,i}$ for x_1, we have:

$$Z_{\bowtie}(i) = \int_0^{w_i} \int_{p_{B,i}+(p_{B,i+1}-p_{B,i})\frac{1}{w_i}\hat{x_1}}^{p_{T,i}+(p_{T,i+1}-p_{T,i})\frac{1}{w_i}\hat{x_1}} \left(\begin{array}{l} \int_0^{\hat{x_1}} \int_{p_{B,i}+(p_{B,i+1}-p_{B,i})\frac{1}{w_i}\hat{x_2}}^{p_{T,i}+(p_{T,i+1}-p_{T,i})\frac{1}{w_i}\hat{x_2}} (\hat{x_1} - \hat{x_2})\, dy_2 d\hat{x_2} \\ \int_{\hat{x_1}}^{w_i} \int_{p_{B,i}+(p_{B,i+1}-p_{B,i})\frac{1}{w_i}\hat{x_2}}^{p_{T,i}+(p_{T,i+1}-p_{T,i})\frac{1}{w_i}\hat{x_2}} (\hat{x_2} - \hat{x_1})\, dy_2 d\hat{x_2} \end{array} + \right) dy_1 d\hat{x_1}$$

Integrating over y_2:

$$Z_{\bowtie}(i) = \int_0^{w_i} \int_{p_{B,i}+(p_{B,i+1}-p_{B,i})\frac{1}{w_i}\hat{x_1}}^{p_{T,i}+(p_{T,i+1}-p_{T,i})\frac{1}{w_i}\hat{x_1}} \left(\begin{array}{l} \int_0^{\hat{x_1}} \left[p_{T,i} + (p_{T,i+1} - p_{T,i})\frac{1}{w_i}\hat{x_2} - p_{B,i} - (p_{B,i+1} - p_{B,i})\frac{1}{w_i}\hat{x_2} \right] (\hat{x_1} - \hat{x_2})\, d\hat{x_2} \\ \int_{\hat{x_1}}^{w_i} \left[p_{T,i} + (p_{T,i+1} - p_{T,i})\frac{1}{w_i}\hat{x_2} - p_{B,i} + (p_{B,i+1} - p_{B,i})\frac{1}{w_i}\hat{x_2} \right] (\hat{x_2} - \hat{x_1})\, d\hat{x_2} \end{array} + \right) dy_1 d\hat{x_1}$$

Substituting the definitions in Equation 2.7:

$$Z_{\bowtie}(i) = \int_0^{w_i} \int_{p_{B,i}+\left(p_{B,i+1}-p_{B,i}\right)\frac{1}{w_i}\hat{x_1}}^{p_{T,i}+\left(p_{T,i+1}-p_{T,i}\right)\frac{1}{w_i}\hat{x_1}} \left(\begin{array}{l} \int_0^{\hat{x_1}} \left[l_i + c_i\hat{x_2}\right]\left(\hat{x_1} - \hat{x_2}\right) d\hat{x_2} \\ \int_{\hat{x_1}}^{w_i} \left[l_i + c_i\hat{x_2}\right]\left(\hat{x_2} - \hat{x_1}\right) d\hat{x_2} \end{array} + \right) dy_1 d\hat{x_1}$$

Integrating over $\hat{x_2}$:

$$Z_{\bowtie}(i) = \int_0^{w_i} \int_{p_{B,i}+\left(p_{B,i+1}-p_{B,i}\right)\frac{1}{w_i}\hat{x_1}}^{p_{T,i}+\left(p_{T,i+1}-p_{T,i}\right)\frac{1}{w_i}\hat{x_1}} \left(l_i\hat{x_1}^2 + \tfrac{1}{3}c_i\hat{x_1}^3 + \tfrac{1}{3}c_i w_i^3 + \tfrac{1}{2}l_i w_i^2 - \tfrac{1}{2}c_i w_i^2\hat{x_1}^2 - l_i w_i\hat{x_1}\right) dy_1 d\hat{x_1}$$

Integrating over y_1, and using our definitions for l_i, r_i, and c_i:

$$Z_{\bowtie}(i) = \int_0^{w_i} \left[l_i + c_i\hat{x_1}\right]\left(l_i\hat{x_1}^2 + \frac{1}{3}c_i\hat{x_1}^3 + \frac{1}{3}c_i w_i^3 + \frac{1}{2}l_i w_i^2 - \frac{1}{2}c_i w_i^2\hat{x_1}^2 - l_i w_i\hat{x_1}\right) d\hat{x_1}$$

After some additional manipulation, we finally obtain:

$$Z_{\bowtie}(i) = \frac{\left(l_i^2 + 3l_i r_i + r_i^2\right) w_i^3}{15} \tag{2.8}$$

Note that our expression for the intrazonal distance factor $Z_{\bowtie}(i)$ is not dependent on the x and y location of the zone, but rather is dependent only upon the length of the left side l_i of the zone, the length of the right side r_i, and the width w_i.

It should be noted that the x component of the expected distance between two random points in a zone is not the intrazonal distance factor $Z_{\bowtie}(i)$, but rather $Z_{\bowtie}(i)$ divided by the square of the area of the zone.

We will now evaluate the expression for the interzonal distance factor $Z_{\diamond\!\!\diamond}(i,j)$ where one random point (x_1, y_1) is distributed in zone i, and the other random point (x_2, y_2) is distributed in zone j where zone i and zone j are not the same. According to our definition of a zone, the x domain for a zone i never overlaps with the x domain of a zone j. In other words, zone i will either be to the left of zone j, in which case:

$$p_{X,i} \leq p_{X,i+1} \leq p_{X,j} \leq p_{X,j+1}$$

or else, zone i will be to the right of zone j, in which case:

$$p_{X,j} \leq p_{X,j+1} \leq p_{X,i} \leq p_{X,i+1}$$

Let us first consider the case where zone i is to the left of zone j. Then, the x component of the distance d_X between the first random point (x_1, y_1) in zone i and the second point (x_2, y_2) in zone j can be expressed without the absolute value sign:

$$d_X(x_1, y_1, x_2, y_2) = x_2 - x_1$$

The expression for $Z_{\Leftrightarrow}(i,j)$ from Equation 2.6 is in this case:

$$Z_{\Leftrightarrow}(i,j) = \int_{p_{X,i}}^{p_{X,i+1}} \int_{p_{B,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})}^{p_{T,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})} \int_{p_{X,j}}^{p_{X,j+1}} \int_{p_{B,j}+\frac{x_2-p_{X,j}}{p_{X,j+1}-p_{X,j}}(p_{T,j+1}-p_{T,j})}^{p_{T,j}+\frac{x_2-p_{X,j}}{p_{X,j+1}-p_{X,j}}(p_{T,j+1}-p_{T,j})} (x_2 - x_1)\, dy_2 dx_2 dy_1 dx_1 \tag{2.9}$$

We can solve Equation 2.9 much like we evaluated the intrazonal distance factor and substitute the definitions from Equation 2.7 wherever possible. Alternatively, we can arrive at the same result by separating Equation 2.9 into two terms:

$$Z_{\Leftrightarrow}(i,j) = \iint_{Zone\ i} \iint_{Zone\ j} x_2 dy_2 dx_2 dy_1 dx_1 - \iint_{Zone\ i} \iint_{Zone\ j} x_1 dy_2 dx_2 dy_1 dx_1$$

$$Z_{\Leftrightarrow}(i,j) = \iint_{Zone\ j} \iint_{Zone\ i} x_2 dy_1 dx_1 dy_2 dx_2 - \iint_{Zone\ i} \iint_{Zone\ j} x_1 dy_2 dx_2 dy_1 dx_1$$

Because the integrand of the first term is independent of x_1 and y_1, and the integrand in the second term is independent of x_2 and y_2, then:

$$Z_{\Leftrightarrow}(i,j) = \text{Area}(i) \iint_{Zone\ j} x_2 dy_2 dx_2 - \text{Area}(j) \iint_{Zone\ i} x_1 dy_1 dx_1$$

We know that the mathematical definition of the center of mass in the x dimension for a region R is:

$$C_X(R) = \frac{\iint_R x dy dx}{\text{Area}(R)}$$

where $C_X(R)$ is the center of mass in the x dimension and $\text{Area}(R)$ is the area of region R. Since zones i and j are instances of regions:

$$Z_{\triangleleft\triangleright}(i,j) = \text{Area}(i)\text{Area}(j)\text{C}_X(j) - \text{Area}(j)\text{Area}(i)\text{C}_X(i)$$

$$Z_{\triangleleft\triangleright}(i,j) = \text{Area}(i)\text{Area}(j)\left(\text{C}_X(j) - \text{C}_X(i)\right) \tag{2.10}$$

with:

$$\text{Area}(i) = \frac{(l_i + r_i)w_i}{2}$$

and, with some simple analysis, we find:

$$\text{C}_X(i) = p_{X,i} + \frac{(l_i + 2r_i)w_i}{3(l_i + r_i)} \tag{2.11}$$

The previous results presupposed that zone i was to the left of zone j. If we assume the other case where zone j is to the left of zone i, we have Equation 2.9 except that the integrand is different:

$$Z_{\triangleleft\triangleright}(i,j) = \int_{p_{X,i}}^{p_{X,i+1}} \int_{p_{B,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{B,i+1}-p_{B,i})}^{p_{T,i}+\frac{x_1-p_{X,i}}{p_{X,i+1}-p_{X,i}}(p_{T,i+1}-p_{T,i})} \int_{p_{X,j}}^{p_{X,j+1}} \int_{p_{B,j}+\frac{x_2-p_{X,j}}{p_{X,j+1}-p_{X,j}}(p_{B,j+1}-p_{B,j})}^{p_{T,j}+\frac{x_2-p_{X,j}}{p_{X,j+1}-p_{X,j}}(p_{T,j+1}-p_{T,j})} (x_1 - x_2)\, dy_2 dx_2 dy_1 dx_1$$

Solving this equation results in:

$$Z_{\triangleleft\triangleright}(i,j) = \text{Area}(i)\text{Area}(j)(\text{C}_X(i) - \text{C}_X(j))$$

We can generalize the expressions for the $Z_{\triangleleft\triangleright}(i,j)$ above and in Equation 2.10 to state that the interzonal distance factor is in all cases:

$$Z_{\triangleleft\triangleright}(i,j) = \text{Area}(i)\text{Area}(j)\left|\text{C}_X(i) - \text{C}_X(j)\right| \tag{2.12}$$

From the above equation we can clearly see that $Z_{\triangleleft\triangleright}(j,i)$ is equal to $Z_{\triangleleft\triangleright}(i,j)$, so we can re-write Equation 2.6 as:

$$\bar{\text{D}}_X(Q) = \frac{1}{\text{Area}^2(Q)}\left(\sum_{i=1}^{n-1} Z_{\bowtie}(i) + 2\sum_{i=1}^{n-2}\sum_{j=i+1}^{n-1} Z_{\triangleleft\triangleright}(i,j)\right) \tag{2.13}$$

To summarize the ideas discussed in this section, we proposed to divide a polygon that is monotone with respect to the x axis into subregions called "zones", and then derived expressions for the x component of the expected Manhattan distance within one zone and between different zones. In the next section, we will develop a method that works with general polygons.

2.3 The x component of the expected Manhattan distance in a general polygon

In this section, we are once again concerned with finding the x component of the expected Manhattan distance in a polygon. However, we will not be restricted to polygons that are monotone with respect to the x axis. The results from the previous section cannot be directly applied to a general polygon because without monotone envelope functions, the length of a path between two points in a general polygon is not always expressed by the formula for the x component of the Manhattan distance:

$$d_X(x_1, y_1, x_2, y_2) = |x_1 - x_2| \tag{2.14}$$

Nevertheless, let us consider a "divide-and-conquer" strategy where we partition the polygon into special regions that meet the criterion that the distance between two points within such a region can be expressed by (2.14). Once this is accomplished, we can calculate the expected distance between random points that are distributed within the same region by utilizing the expressions that we developed in the previous section. Then, if we also develop an expression for the expected distance between points in different regions, we will have an expression for the expected distance in the polygon.

The first step in our proposed strategy is to partition the polygon into regions such that the distance between two points in a region can be expressed by the formula in (2.14). There are, in fact, an infinite number of ways to partition a polygon that meet this criterion. We will develop a method to partition a polygon into regions that have two characteristics: *(i)* the regions are x-monotone, and *(ii)* the regions are bounded in the x dimension at the x value of a cusp that is on the bottom or top envelope.

These two characteristics need further elaboration. We define a region to be "x-monotone" if it is bounded on the bottom and top by envelopes that are x-monotone and that are on the perimeter of the polygon. The notion of the x-monotone polygon described in the previous section is an example of an x-monotone region. To better understand this characteristic, it is helpful to study the polygon in Figure 2-8. We will refer to each region that is partitioned according to our requirements as a "channel". Notice how each channel maintains the property that the bottom and top envelopes of each channel are on the

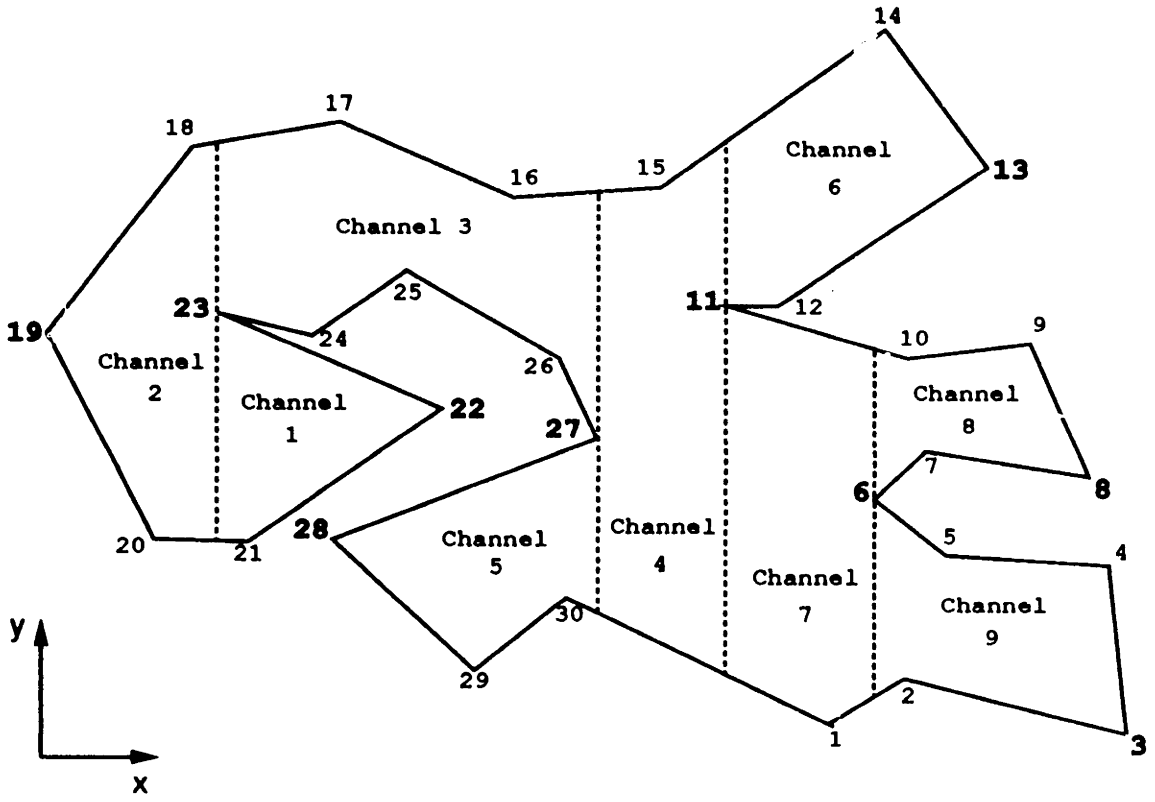

Figure 2-8: A non-monotone polygon divided into 9 x-monotone "channels"

perimeter of the polygon and that both envelopes are x-monotone.

The description of the second characteristic refers to a "cusp", which we define to be a vertex point on the polygon such that the adjacent vertex points either both have a greater x value or both have a lesser x value than the x value of the cusp. In the Figure 2-8, those vertex points labelled in boldface are cusps. For example, the vertex point 19 is a cusp because the adjacent vertex points 18 and 20 both have an x value which is greater than that of point 19. It is worth noting that cusps are the only locations on the perimeter of the polygon where the property of x-monotonicity does not hold true. From the diagram, we can observe the second characteristic whereby the x bounds of each channel is located at a cusp.

We have stated our general "divide-and-conquer" strategy and have described the characteristics of the regions called "channels" into which a polygon may be divided. In the next subsection, we will present a method for dividing a polygon into channels. In the

subsequent subsection, we will develop an expression for the expected distance in the x direction between two random points that are distributed within one channel and that are distributed in different channels.

2.3.1 Dividing the polygon into channels

In devising a method to divide the polygon into channels, the fact that each channel is bounded by vertex points that are cusps suggests that we should first identify all cusps. The procedure for accomplishing this task follows directly from the definition of a cusp given earlier whereby a cusp can be determined by comparing the x coordinate of a vertex point with the x coordinates of the two adjacent vertex points. In fact, we will categorize every vertex point as belonging to one of the following five classes:

1. "Intermediate"–any vertex point that is not a cusp.

2. "Open"–a cusp at a convex vertex point where the adjacent vertex points both have a greater x value than that of the cusp. The vertex points that are in the open class in Figure 2-8 are points 19 and 28. Pictorially, a vertex point in the open class "opens" a new channel, forming the left limit.

3. "Divide"–a cusp at a concave vertex point where the adjacent vertex points both have a greater x value than that of the cusp. The vertex points in the figure that are in the divide class are points 23, 11 and 6. We see in the diagram that a vertex point in the divide class "divides" one channel on the left into two channels on the right.

4. "Close"–a cusp at a convex vertex point where the adjacent vertex points both have a lesser x value than that of the cusp. Vertex points that are in the "close" class in the figure are points 22, 13, 8 and 3. Pictorially, a vertex point in this class "closes" a channel, forming the right limit.

5. "Join"–a cusp at a concave vertex point where the adjacent vertex points both have a lesser x value than that of the cusp. The only vertex point that is in this class in the figure is point 27.) In the diagram, we see that this vertex point joins two channels on the left into one channel on the right.

We observe that every x-monotone polygon (including all convex polygons) has exactly two cusps – one in the "open" class, and one in the "close" class. Every polygon has one or more vertex points in the "intermediate" class. Only polygons that are not x-monotone have "join" or "divide" vertex points. For every "divide" vertex, there is also an extra "close" vertex, and for every "join" vertex, there is an extra "open" vertex. The number of channels is one less than the number of cusps.

We should note that there is a slight complication when two or more adjacent vertex points have the same x coordinate. In this case, the group of adjacent vertex points sharing the same x coordinate are treated as one unit, and its x coordinate is compared with the x coordinates of the vertex points that are adjacent to the group. For instance, in a hypothetical polygon where vertex points 3, 4, and 5 have the same x coordinate, then this x coordinate is compared with the x coordinates of vertex 2 and of vertex 6. When the class for the group of vertex points is found, only one of the vertex points is assigned to that class. For instance, if the group of vertex points 3, 4 and 5 have the same x coordinate and is found to be of the "divide" class, then one arbitrarily chosen vertex point (let us choose point 3) is assigned to the "divide" class, and the other vertex points in the group (in this case, 4 and 5) are assigned to the "intermediate" class.

There is more than one method for finding the channels in a polygon once the vertex points are categorized into the five classes. The method that we will adopt starts at the vertex points in the "open" class, and each such vertex point is necessarily at the left edge of a channel. We then "process" the channels one by one. By the term "process", we mean that we will apply a procedure to divide the channel into zones (as we had in the previous section) by following the bottom and top envelope of the channel until we encounter a cusp which defines the right edge of the channel.

To implement this method, we will need to keep track of the channels in a list which we will call the "channel queue". The channel queue represents the known channels that we have yet to "process". Initially, we place all vertex points in the "open" class into the channel queue in any arbitrary order. Our method is structured around an iteration loop that continues until the channel queue is empty, or, in other words, until all channels have been processed. In each iteration loop, we take the first channel in the channel queue which

we call the "current channel" and we "process" it.

The steps to "process" a channel are similar to the steps to find the zones in an x-monotone polygon. In fact, because we will eventually need the zones to compute the expected distance, we will generate and save the zone definitions for later use. To traverse the channel one zone at a time, we must find the "next point" which we define as the next closest vertex point to the right in the x dimension. To do this, we follow the bottom and top envelopes knowing that in a polygon defined with vertex point indicies arranged in counter-clockwise order, the point indicies *increase* as we move to the right on the bottom envelope and *decrease* as we move to the right on the top envelope. For example, in Figure 2-8, if we are starting at vertex 19, the next vertex point on the bottom envelope is vertex 20, and the next vertex point on the top envelope is 18. Because vertex 20 has a smaller x value (hence is the closer vertex point in terms of the x coordinate), it is the "next point" and defines the right edge of the first zone. As long as the class for the "next point" is "intermediate", then we repeat the procedure to find the right edge of the next zone until we encounter a vertex point that is a cusp.

Actually, we must refine our definition of the "next point" so that we consider not only the next vertex point on each of the two envelopes, but also any vertex point of the "divide" class that is between the two envelopes. The "next point" is the closest of all of these. For example, in Channel 2 in Figure 2-8, the sequence of "next points" found is: 20, then 18, and finally 23, which is a "divide" point between the top and bottom envelope of the channel that is closer than either 21 or 17.

The cusp marks the right edge of the current channel, and we can remove the current channel from the channel queue. The class of the cusp also indicates if there are any channels connected to the right of the current channel. The possible classes for a cusp encountered by traversing a channel from left to right are: "close", "divide", and "join". If the class of the cusp is "close", then there is no channel to the right.

However, if the class is "divide", there are two new channels to the right, which we place into the channel queue. In Figure 2-8, vertex point 23 pictorially "divides" one channel (Channel 2) into two channels (Channel 1 and Channel 3). Of the two new channels, the lower channel is bounded at the bottom by same bottom envelope as the current channel.

The top envelope of the lower channel begins at the "divide" point. Conversely, the upper channel has the same top envelope as the top envelope for the current channel, and its bottom envelope starts at the "divide point".

If the class of the cusp is "join", then there is one channel to the right. For example, in Figure 2-8, if we have processed Channel 3 which has a right edge at the "join" point 27, then we know that there is a new channel to the right. If the "join" point is on the bottom envelope of the current channel (as is the case for Channel 3), then the top envelope of the new channel is the same as the top envelope for the current channel. However, we will not know the bottom envelope until the other channel (in this case, Channel 5) that meets at the "join" point is processed. For each pair of channels that are "joined" at a vertex point belonging to the "join" class, one new channel is placed into the channel queue.

The method iteratively processes all channels in the channel queue until channel queue is empty, at which point the polygon is completely divided into channels. This concludes our discussion on the first phase of the "divide-and-conquer" strategy. Before proceeding to the final phase of the strategy in the next subsection, let us summarize our method for dividing the polygon into channels:

1. For every vertex point in the polygon, categorize the point as belonging to one of the following classes: "open", "close", "divide", "join", or "intermediate".

2. Create a "channel queue" for channels yet to be processed. For every vertex point belonging to the "open" class, place a channel into the queue whose left edge is formed by the vertex point. In the subsequent steps, process the first channel in the queue as the "current channel".

3. To process the "current channel", begin at the left edge of the channel and find the zones in the channel one at a time. The left edge of the first zone in the channel is the left edge of the channel. To find the right edge of the first zone, find the next vertex points to the right that are on the bottom and top envelopes. Choose the closer one of the two (the one with the smaller x value). We will call this point the "next point" unless there happens to be some other point belonging to the "divide" class that is both *between* the bottom and top envelopes and that is even closer. In that case, this

other point is the "next point". In either case, the "next point" defines the right edge of the first zone in the channel.

4. If the "next point" belongs to the class "close", "divide", or "join", go to step 6 (since we are at right edge of the current channel).

5. Otherwise, proceed to find all other zones in the channel by the same method as step 4 – look for the next point to the right on the bottom and top envelopes of the channel and choose the closer of the two. Unless there is a closer point that is between the envelopes (belonging to the "divide" class), this point is the "next point" defining the right edge of the zone. The left edge of the zone is at the same place as the right edge of the previous zone. Go back to step 4.

6. At this point, we have processed the "current channel" and have found all of its zones from the left to the right. Remove the current channel from the channel queue. The most recent "next point" not only defines the right edge of the channel, but also its class determines if there are more channels to the right:

   - If the class of the "next point" is "close", then there are no channels immediately to the right.
   - If the class of the "next point" is "divide", then put two new channels into the "channel queue". The left edge of both of the new channels is at the "divide" point. One of the new channels will have the same bottom envelope as the bottom envelope of the current channel. The top envelope of this new channel with begin at the "divide" point. The other new channel will have the same top envelope as the current channel's top envelope, and the bottom envelope of this new channel will begin at the "divide" point.
   - Otherwise, the "next point" belongs to the "join" class. In this scenario, two channels join together from the left of the "next point" to form one channel to the right. If the current channel is the first channel of the two channels to reach the "next point", then save this fact for later reference. On the other hand, if the "next point" has already been reached by another channel, then put a new

channel into the "channel queue". The left edge of the new channel is at the "next point", and the bottom and top envelopes are continued from the two channels that are joined together.

7. As long as there are some channels in the "channel queue" that have not been processed, go back to step 3 and process the next channel in the queue as the "current channel". Otherwise, the entire polygon has been divided into channels and zones.

2.3.2 The expected distance between points in different channels

Having found a method for dividing a polygon into channels, we now turn to the problem of finding an expression for the x component of the expected distance between random points that are distributed in different channels. Given a polygon Q that is divided into H channels with one random point (x_1, y_1) distributed in channel a and the other random point (x_2, y_2) distributed in channel b:

$$\bar{\mathrm{D}}_X(Q) = \frac{\sum_{a=1}^{H}\sum_{b=1}^{H}\int\limits_{Channel\ a}\int\limits_{Channel\ b}\mathrm{d}_M(x_1,y_1,x_2,y_2)dy_2dx_2dy_1dx_1}{\mathrm{Area}^2(Q)} \tag{2.15}$$

In the same way that we earlier found the expected distance between random points distributed in zones, we can express Equation 2.15 in terms of an "intrachannel distance" factor $C_{\bowtie}(a)$ and an "interchannel distance" factor $C_{\diamond\!\!\diamond}(a,b)$.

$$\bar{\mathrm{D}}_X(Q) = \frac{1}{\mathrm{Area}^2(Q)}\left(\sum_{a=1}^{H} C_{\bowtie}(a) + \sum_{a=1}^{H}\sum_{b=1,b\neq a}^{H} C_{\diamond\!\!\diamond}(a,b)\right) \tag{2.16}$$

The intrachannel distance factor is expressed by:

$$C_{\bowtie}(a) = \int_{C_{L,a}}^{C_{R,a}}\int_{bottom_a(x_1)}^{top_a(x_1)}\int_{C_{L,a}}^{C_{R,a}}\int_{bottom_a(x_2)}^{top_a(x_2)}|x_1 - x_2|\,dy_2dx_2dy_1dx_1$$

where the x domain of the channel a is from $C_{L,a}$ to $C_{R,a}$ and the channel is enveloped by the $bottom_a(x)$ and $top_a(x)$ functions. Given the fact that a channel is by definition an x-monotone region, it is not surprising that the above expression and Equation 2.5 are very similar. In fact, because our method for dividing the polygon into channels also divided each channel into zones, we can use the same procedure to evaluate the intrachannel distance factor. With a channel a divided into $C_{Z,a}$ zones, we can adapt Equation 2.13, which represents the expected distance factor between points distributed in zones, to:

$$C_{\bowtie}(a) = \sum_{i=1}^{C_{Z,a}} Z_{\bowtie}(i) + 2 \sum_{i=1}^{C_{Z,a}-1} \sum_{j=2}^{C_{Z,a}} Z_{\Leftrightarrow}(i,j) \tag{2.17}$$

where $Z_{\bowtie}(i)$ and $Z_{\Leftrightarrow}(i,j)$ were evaluated in the previous section (see Equations 2.8 and 2.12).

Evaluating the "interchannel distance" factor $C_{\Leftrightarrow}(a,b)$ is very similar to evaluating the interzonal distance factor. In fact, two points are necessarily in different zones if the points are in different channels hence the distance between them is technically "interzonal". The distance factor is simply the summation of individual distance factors $Z_{\Leftrightarrow}(a,i,b,j)$ between zone i of channel a to zone j of channel b.

$$C_{\Leftrightarrow}(a,b) = \sum_{i=1}^{C_{Z,a}} \sum_{j=1}^{C_{Z,b}} Z_{\Leftrightarrow}(a,i,b,j)$$

However, the expected distance between points in zones of different channels is not merely the difference in the x coordinates between the centers of mass $C_X(i)$ for the two zones as was the case earlier for x-monotone regions (see Equation 2.12). Depending on the configuration of the channels, the x component of the expected distance between points in zones of different channels may have to "go around" sides of the polygon that are barriers. Figure 2-9 shows a zone i in channel a and a zone j in a different channel b where the x component of the path between points in the two zones is "indirect", and involves four segments. In Equation 2.4, we wrote an expression for the x component of the distance of an indirect path between two points with m intermediate vertex points. We will use that distance expression to represent the path between a point in zone i of a channel a to a point in zone j of some other channel b:

$$Z_{\Leftrightarrow}(a,i,b,j) = \int\int_{Zone\ a,i} \int\int_{Zone\ b,j} |x_1 - I_{X,1}| + |x_2 - I_{X,m}| + \sum_{k=1}^{m-1} |I_{X,k} - I_{X,k+1}|\, dy_2 dx_2 dy_1 dx_1$$

Separating the three terms in the integrand:

$$\begin{aligned} Z_{\Leftrightarrow}(a,i,b,j) = & \int\int_{Zone\ a,i} \int\int_{Zone\ b,j} |x_1 - I_{X,1}|\, dy_2 dx_2 dy_1 dx_1 + \\ & \int\int_{Zone\ a,i} \int\int_{Zone\ b,j} |x_2 - I_{X,m}|\, dy_2 dx_2 dy_1 dx_1 + \\ & \int\int_{Zone\ a,i} \int\int_{Zone\ b,j} \textstyle\sum_{k=1}^{n-1} |I_{X,k} - I_{X,k+1}|\, dy_2 dx_2 dy_1 dx_1 \end{aligned}$$

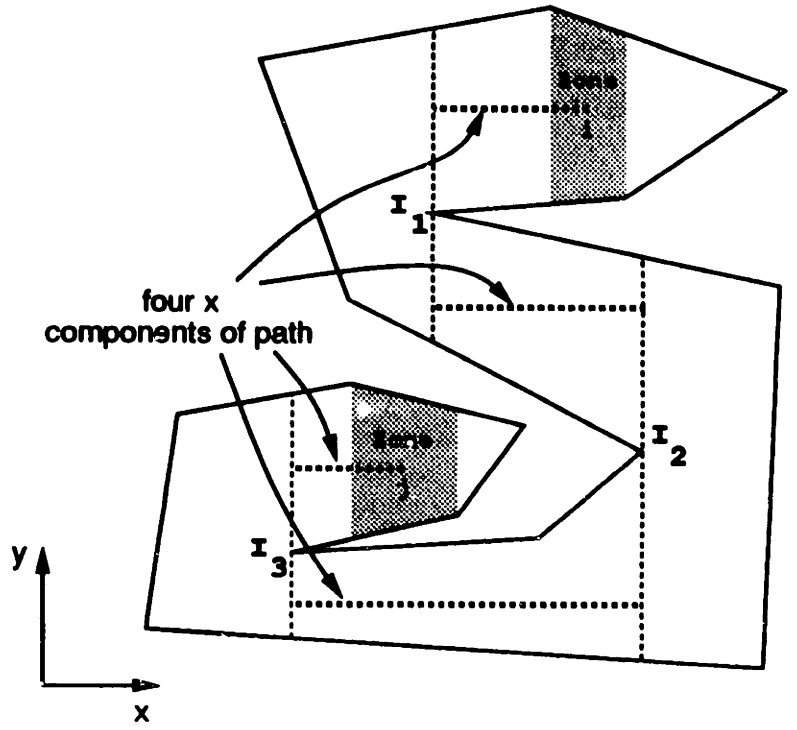

Figure 2-9: Indirect path between points in zones of different channels

Employing the same analysis that produced Equation 2.12, we have:

$$\begin{aligned} Z_{\ll\gg}(a,i,b,j) = \; & Area(j)\iint\limits_{Zone\ a,i} |x_1 - I_{X,1}|\, dy_1 dx_1 + \\ & Area(i)\iint\limits_{Zone\ b,j} |x_2 - I_{X,m}|\, dy_2 dx_2 + \\ & Area(i)Area(j)\textstyle\sum_{k=1}^{m-1} |I_{X,k} - I_{X,k+1}| \end{aligned}$$

$$\begin{aligned} Z_{\ll\gg}(a,i,b,j) = \; & Area(j)\,|C_X(i) - I_{X,1}| + Area(i)\,|C_X(j) - I_{X,m}| + \\ & Area(i)Area(j)\mathrm{d}_X(a,b) \end{aligned} \tag{2.18}$$

where $\mathrm{d}_X(a,b)$ is the length of the shortest path connecting channel a and channel b the intermediate vertices I_1 and I_m along the path between a point in channel a to a point in channel b is:

$$\mathrm{d}_X(a,b) = \sum_{k=1}^{m-1} |I_{X,k} - I_{X,k+1}|$$

Our one remaining task is to find the path between two different channels. There are many techniques that may be used to accomplish task. One way is to simply keep track of connections between channels as we divide the polygon into channels. Another way is the build an adjacency matrix after the channels have been divided. (This technique is implemented in the computer program in Appendix A.2.) Yet another way is to build a tree data structure which represents the connections between adjacent channels, and to use a tree traversal algorithm to produce the path between different channels. Because these techniques are documented elsewhere, we will not dwell on the procedure of finding the path between different channels.

One final note is that in sections 2.2 and 2.3, we found the x component of the expected Manhattan distance only. As expressed in Equation 2.3, we must also repeat the procedure for the y component and add the two components in order to find the correct expected distance in the Manhattan metric.

2.3.3 Analysis of the method

The order of complexity of the method remains to be analyzed. At the heart of the method to divide the polygon into channels is the procedure to define the individual zones within the channels. In n-sided polygons without cusps belonging to the "divide" class, each zone requires computation of order $\mathcal{O}(1)$ and there are $n-1$ zones so that the complexity of the method is $\mathcal{O}(n)$. However, if there are m cusps of the "divide" class, each zone requires m tests to check if such cusps bound the zone, and the method performs at order $\mathcal{O}(m \cdot n)$ which, in the worse case, is of order $\mathcal{O}(n^2)$.

Apart from the method to divide the polygon into zones and channels, there is the actual calculation of the expected distance factors themselves. The calculation for monotone polygons in section 2.2 is expressed in Equation 2.13. This equation contains a pair of nested summations each of order $\mathcal{O}(n)$, thus the order of complexity is $\mathcal{O}(n^2)$. For general polygons, there is the added computation to search for the shortest path between different channels. However, if shortest path computations are performed separately (not nested in the loop that calculates the expected distance factor) with results stored in a matrix (as is done in Appendix A.2), the total order of complexity for general polygons remains at $\mathcal{O}(n^2)$.

2.4 Conclusions

In this chapter, we studied the problem of finding the expected distance between two random points in a polygon. We first showed that we could find the expected Manhattan distance by summing two separate quantities–the expected distance between two random points where the distance was measured in the x direction, and the expected distance between two random points in the y direction. Because we were able to show that these two quantities could be found independently and equivalently, we focused our attention on the problem of finding the expected distance in one dimension at a time.

To solve this problem, we developed a method to find the expected distance in an x-monotone polygon, where the polygon was enveloped at the top and the bottom by a perimeter that was monotone with respect to the x-axis. We utilized the fact that the envelope functions for a polygon are piecewise linear, and divided the polygon into subregions called "zones" where the top and bottom envelopes are linear. Expressions were derived for the expected distance between two random points in the same zone, and for the expected distance between two random points in different zones. These expressions were incorporated into the expression for the expected distance in the x dimension for an x-monotone polygon.

In general polygons, the property of x-monotone envelopes does not hold true. The approach that we used to find the expected distance was to partition the polygon into regions which we call "channels" that are x-monotone. The expected distance between random points in the same channel can be computed in the identical way as for random points in x-monotone polygon. The technique to find the expected distance between random points in different channels was very similar to finding the expected distance between random points in different zones, only that the path between the two different channels had to be considered. Before find the path between channels, it is first necessary to define the channels themselves, and perhaps the central part of the approach is the method to divide the polygon into channels. This we accomplished by observing that the perimeter of a polygon is x-monotone except at vertex points that are cusps. By finding each cusp and determining its class, we were able to find each of the channels.

In summary, we have studied the problem of finding the expected Manhattan distance

in a poi gon, and have presented a numerical method for expressing a solution in closed form.

Chapter 3

The Expected Euclidean Distance in a Convex Polygon

In this chapter, we will present a method for finding the expected distance between two random points in a convex polygon using the Euclidean distance metric. We will proceed by first deriving a mathematical expression for finding the expected Euclidean distance in general convex planar regions. Then, we will present an algorithm that divides a convex polygon in such a way that we can apply the expression in closed form.

3.1 Expected Euclidean Distance Expression

3.1.1 Direct Methods

We will begin by stating the general expected distance expression using the same notation as in Chapter 1.

$$\bar{\mathrm{D}}_E(Q) = \iint_Q \iint_Q \mathrm{f}_Q(q_1)\mathrm{f}_Q(q_2)\mathrm{d}_E(q_1, q_2)dq_2dq_1 \tag{3.1}$$

where $\mathrm{d}_E(q_1, q_2)$ is the distance in the Euclidean metric between the random points q_1 and q_2 in the region Q, and $\mathrm{f}_Q(q)$ is the density function for distributing a random point in the region.

As discussed in Chapter 1, it is reasonable to limit our consideration to polygons as special cases of general planar regions, because a polygon can approximate any simply-

connected planar region. Let us first try to solve the expression in Cartesian coordinates, since polygons are most easily defined in this coordinate system.

$$q_1 \equiv (x_1, y_1), \qquad q_2 \equiv (x_2, y_2)$$

In this chapter, we will study the Euclidean distance metric. So that we can use the straight-line distance formula without having to consider the barriers possible in concave regions, let us consider convex polygons only.

$$\mathrm{d}_E(q_1, q_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Once again, we are interested in the expected distance between two random points that are uniformly, independently and identically distributed.

$$\mathrm{f}_Q(q_1) = \mathrm{f}_Q(q_2) = \frac{1}{\mathrm{Area}(Q)}$$

Therefore, for this specific subset of expected distance problems, the expression (3.1) becomes:

$$\bar{\mathrm{D}}_E(Q) = \frac{\iint_Q \iint_Q \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} dy_2 dx_2 dy_1 dx_1}{\mathrm{Area}^2(Q)} \tag{3.2}$$

Let us consider the simplest of all planar regions defined in Cartesian coordinates – the unit square.

$$\bar{\mathrm{D}}_E(square) = \int_0^1 \int_0^1 \int_0^1 \int_0^1 \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} dy_2 dx_2 dy_1 dx_1 \tag{3.3}$$

It so happens that the four integrals in this expression cannot be evaluated in closed form. Even for this simple polygon, we can only perform two of the four integrations before intractable terms such an inverse hyperbolic sine term multiplied with a square-root expression prevent further evaluation. We conclude that we cannot derive an expression that distributes q_1 and q_2 in Cartesian coordinates.

We now attempt to set up the expected Euclidean distance expression that distributes the two points q_1 and q_2 in polar coordinates.

$$q_1 \equiv (r_1, \theta_1), \qquad q_2 \equiv (r_2, \theta_2)$$

$$d_E(q_1, q_2) = \sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\theta_2 - \theta_1)}$$

$$f_Q(q) = \frac{r}{\text{Area}(Q)}$$

For the simplest planar region defined in polar coordinates – a circle of radius a – this expression can be solved analytically. [1]

$$\bar{D}_E(circle_a) = \frac{\int_0^a \int_0^{2\pi} \int_0^a \int_0^{2\pi} \sqrt{r_1^2 + r_2^2 - 2r_1 r_2 \cos(\theta_2 - \theta_1)} r_1 r_2 d\theta_2 dr_2 d\theta_1 dr_1}{(\pi a^2)^2} \tag{3.4}$$

$$\bar{D}_E(circle_a) = \frac{8a}{5(\Gamma(\frac{5}{2}))^2}$$

$$\bar{D}_E(circle_a) = \frac{128a}{45\pi}$$

Therefore, the expression for a circle can be evaluated. However, the class of planar regions that can easily be defined in polar coordinates -- circles, cardioids and conic sections – is rather small and cannot be used easily to approximate arbitrary regions.

3.1.2 An alternative approach

Even though Cartesian coordinates are ideal for defining polygons, it is extremely difficult to produce closed form results for Euclidean distances primarily due to difficulty in integrating the distance formula in that coordinate system. Conversely, with polar coordinates, we have some degree of analytic tractability but polygons (or other general planar regions) cannot easily be defined. These observations would suggest an alternative approach, whereby one point is distributed in Cartesian coordinates, and the other in polar coordinates.

We will begin by stating the general expected distance expression for any convex region Q written in terms of conditional expectation.

$$\bar{D}_E(Q) = \iint_Q f_Q(q_1) \iint_Q f_{Q_2|q_1}(q_2|q_1) d_E(q_1, q_2|q_1) dq_2 dq_1 \tag{3.5}$$

We can choose to distribute q_1 and q_2 in any two-dimensional coordinate system. Let us distribute q_2 in polar coordinates where the origin is at q_1.

[1] Samuel Eilon, C.D.T. Watson-Gandy and Nicos Christofides, *Distribution Management: Mathematical modelling and practical analysis* (New York: Hafner Publishing Company, 1971), p. 154.

$$q_2 \equiv (r_2, \theta_2)$$

Then,

$$\bar{D}_E(Q) = \iint_Q f_Q(q_1) \int_0^{2\pi} \int_0^{r(q_1,\theta_2)} f_{R,\Theta|q_1}(r_2, \theta_2|q_1) d_E(q_1, r_2, \theta_2|q_1) dr_2 d\theta_2 dq_1 \qquad (3.6)$$

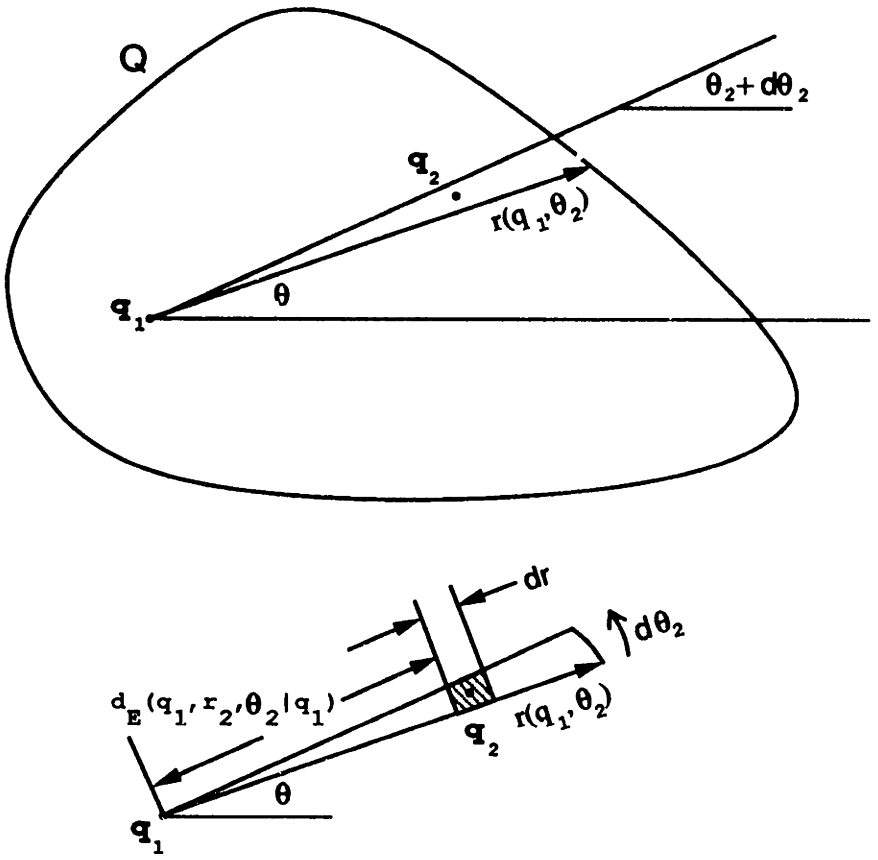

Figure 3-1: A convex region defined by $r(q_1, \theta_2)$ at a given q_1

As shown in Figure 3-1, the function $r(q_1, \theta_2)$ is the distance from q_1 to the boundary of the region Q for a given value of θ_2. The function $d_E(q_1, r_2, \theta_2|q_1)$ is the Euclidean distance from q_1 to q_2.

For uniform distribution of q_2 and independent distribution between q_1 and q_2 in the polar coordinate system,

$$f_{R,\Theta|q_1}(r_2, \theta_2|q_1) = \frac{r_2}{\iint_Q r_2 dr_2 d\theta_2} = \frac{r_2}{\int_0^{2\pi} \int_0^{r(q_1,\theta_2|q_1)} r_2 dr_2 d\theta_2}$$

Because we are assuming Q to be a convex region such that the distance between any two points in the region is the straight-line distance, and q_1 is by definition at the origin of the polar coordinate system, the distance between q_1 and q_2 is simply:

$$\mathrm{d}_E(q_1, r_2, \theta_2 | q_1) = r_2$$

Integrating (3.6) with respect to r_2, we in effect absorb the distribution of q_2:

$$\bar{\mathrm{D}}_E(Q) = \frac{\iint_Q \mathrm{f}_Q(q_1) \int_0^{2\pi} \frac{1}{3}\mathrm{r}^3(q_1, \theta_2) d\theta_2 dq_1}{\int_0^{2\pi} \frac{1}{2}\mathrm{r}^2(q_1, \theta_2) d\theta_2 dq_1} \tag{3.7}$$

Changing the order of integration:

$$\bar{\mathrm{D}}_E(Q) = \frac{\frac{2}{3}\int_0^{2\pi} \iint_Q \mathrm{f}_Q(q_1)\mathrm{r}^3(q_1, \theta_2) dq_1 d\theta_2}{\int_0^{2\pi} \mathrm{r}^2(q_1, \theta_2) dq_1 d\theta_2} \tag{3.8}$$

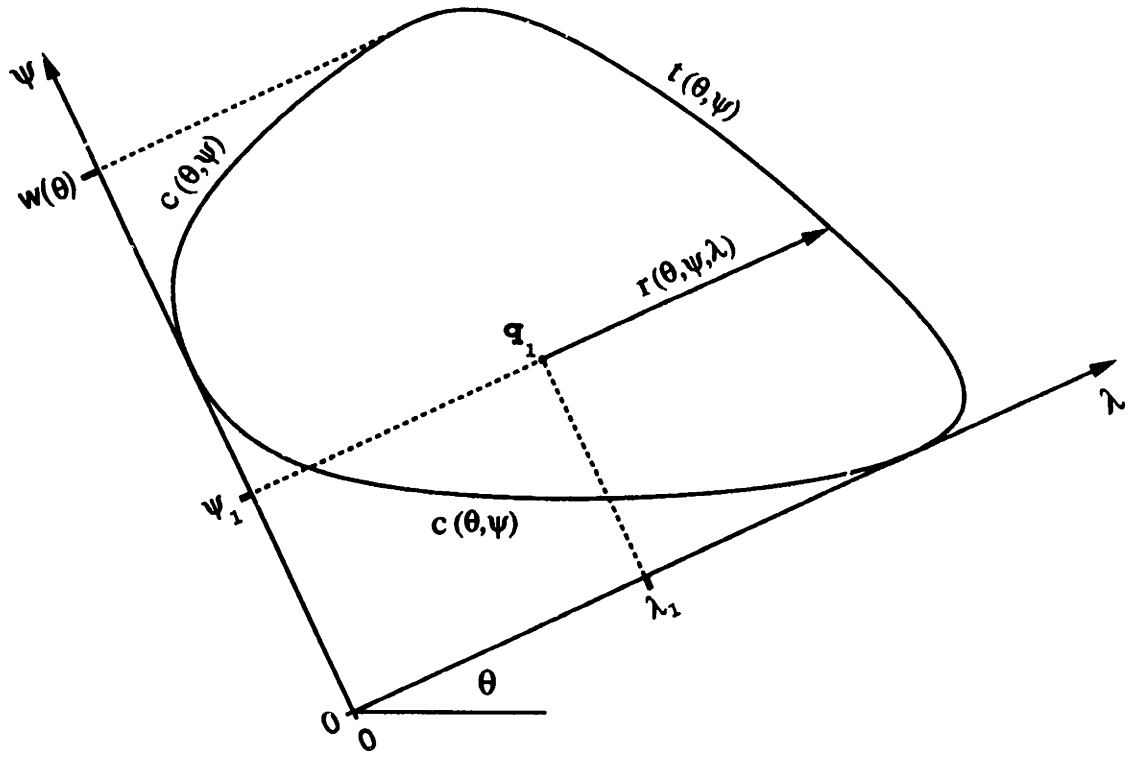

Figure 3-2: A convex region shown in the (θ,ψ,λ) coordinate system

Now let us choose to distribute q_1 in a quasi-Cartesian coordinate system (ψ, λ) where the λ-axis is in the direction of θ and ψ is perpendicular to λ. We define ψ such that the extremity of the region Q with the lowest value of ψ is at $\psi = 0$, and the extremity of Q with the highest value of ψ is at $\psi = \mathrm{w}(\theta)$. Therefore, the value of $\mathrm{w}(\theta)$ represents the "width" of the region Q in the ψ direction. Figure 3-2 shows an example of a region Q with the ψ and λ axes drawn for a given value of θ.

The functions $\mathrm{c}(\theta, \psi)$ and $\mathrm{t}(\theta, \psi)$ represent the λ values of the lower and upper boundary of the region Q at a given ψ value for a given θ angle.

$$\bar{\mathrm{D}}_E(Q) = \frac{\frac{2}{3}\int_0^{2\pi}\int_0^{\mathrm{w}(\theta)}\int_{\mathrm{c}(\theta,\psi)}^{\mathrm{t}(\theta,\psi)} \mathrm{f}_Q(\theta,\psi,\lambda)\mathrm{r}^3(\theta,\psi,\lambda)d\lambda d\psi d\theta}{\int_0^{2\pi}\mathrm{r}^2(\theta,\psi,\lambda)d\theta} \tag{3.9}$$

In our quasi-Cartesian coordinate system,

$$\mathrm{f}_Q(\theta,\psi,\lambda) = \frac{1}{\int_0^{\mathrm{w}(\theta)}\int_{\mathrm{c}(\theta,\psi)}^{\mathrm{t}(\theta,\psi)} d\lambda d\psi}$$

$$\mathrm{r}(\theta,\psi,\lambda) = \mathrm{t}(\theta,\psi) - \lambda$$

Thus,

$$\bar{\mathrm{D}}_E(Q) = \frac{\frac{2}{3}\int_0^{2\pi}\int_0^{\mathrm{w}(\theta)}\int_{\mathrm{c}(\theta,\psi)}^{\mathrm{t}(\theta,\psi)}(\mathrm{t}(\theta,\psi)-\lambda)^3 d\lambda d\psi d\theta}{\int_0^{2\pi}\int_0^{\mathrm{w}(\theta)}\int_{\mathrm{c}(\theta,\psi)}^{\mathrm{t}(\theta,\psi)}(\mathrm{t}(\theta,\psi)-\lambda)^2 d\lambda d\psi d\theta} \tag{3.10}$$

Let us shift λ by $\mathrm{c}(\theta, \psi)$ so that $\lambda = 0$ does not correspond to the locus of points on the ψ-axis, but rather the locus of points on $\mathrm{c}(\theta, \psi)$.

$$\bar{\mathrm{D}}_E(Q) = \frac{\frac{2}{3}\int_0^{2\pi}\int_0^{\mathrm{w}(\theta)}\int_0^{\mathrm{t}(\theta,\psi)-\mathrm{c}(\theta,\psi)}(\mathrm{t}(\theta,\psi)-(\lambda+\mathrm{c}(\theta,\psi)))^3 d\lambda d\psi d\theta}{\int_0^{2\pi}\int_0^{\mathrm{w}(\theta)}\int_0^{\mathrm{t}(\theta,\psi)-\mathrm{c}(\theta,\psi)}(\mathrm{t}(\theta,\psi)-(\lambda+\mathrm{c}(\theta,\psi)))^2 d\lambda d\psi d\theta} \tag{3.11}$$

Let us define a new function,

$$\mathrm{l}(\theta,\psi) = \mathrm{t}(\theta,\psi) - \mathrm{c}(\theta,\psi)$$

From Figure 3-3, we observe that $\mathrm{l}(\theta, \psi)$ has the physical interpretation of being the length or extent of the polygon in the θ direction at a given value of ψ.

$$\bar{\mathrm{D}}_E(Q) = \frac{\frac{2}{3}\int_0^{2\pi}\int_0^{\mathrm{w}(\theta)}\int_0^{\mathrm{l}(\theta,\psi)}(\mathrm{l}(\theta,\psi)-\lambda)^3 d\lambda d\psi d\theta}{\int_0^{2\pi}\int_0^{\mathrm{w}(\theta)}\int_0^{\mathrm{l}(\theta,\psi)}(\mathrm{l}(\theta,\psi)-\lambda)^2 d\lambda d\psi d\theta}$$

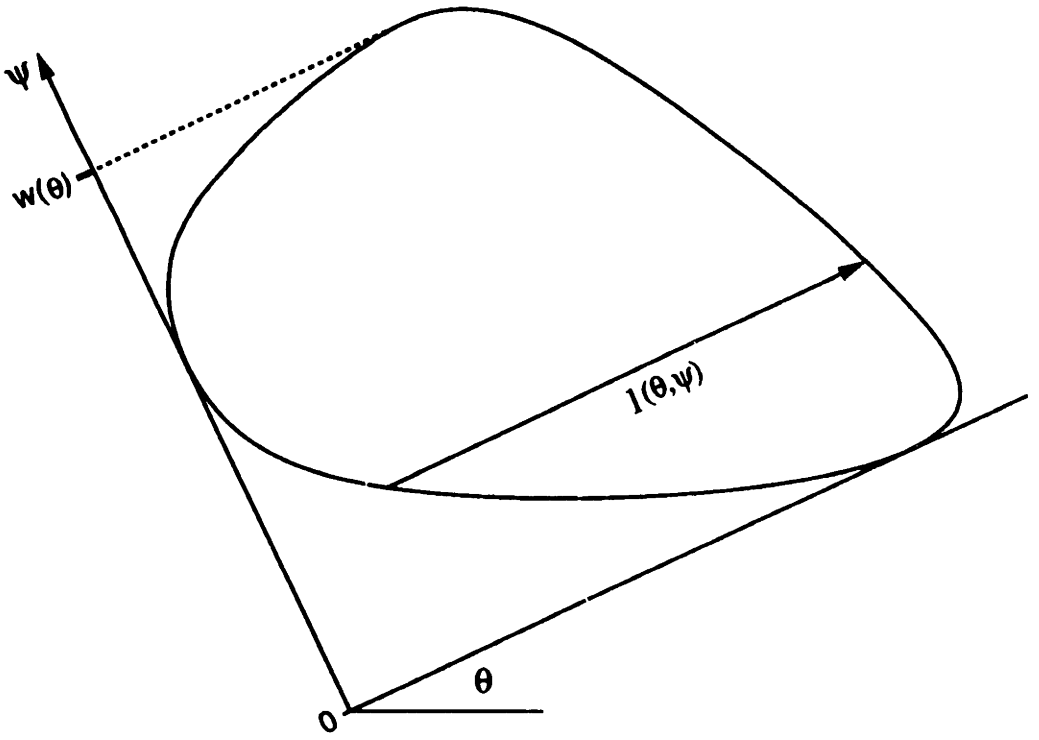

Figure 3-3: The length function $l(\theta, \psi)$

$$\bar{D}_E(Q) = \frac{\frac{2}{3}\int_0^{2\pi}\int_0^{w(\theta)}\left[l^3(\theta,\psi)\psi - \frac{3}{2}l^2(\theta,\psi)\psi^2 + \frac{3}{3}l(\theta,\psi)\psi^3 - \frac{1}{4}\psi^4\right]_0^{l(\theta,\psi)} d\psi d\theta}{\int_0^{2\pi}\int_0^{w(\theta)}\left[l^2(\theta,\psi)\psi - \frac{2}{2}l(\theta,\psi)\psi^2 + \frac{1}{3}\psi^3\right]_0^{l(\theta,\psi)} d\psi d\theta}$$

$$\bar{D}_E(Q) = \frac{\frac{2}{3}\frac{1}{4}\int_0^{2\pi}\int_0^{w(\theta)} l^4(\theta,\psi) d\psi d\theta}{\frac{1}{3}\int_0^{2\pi}\int_0^{w(\theta)} l^3(\theta,\psi) d\psi d\theta}$$

$$\bar{D}_E(Q) = \frac{\frac{1}{2}\int_0^{2\pi}\int_0^{w(\theta)} l^4(\theta,\psi) d\psi d\theta}{\int_0^{2\pi}\int_0^{w(\theta)} l^3(\theta,\psi) d\psi d\theta} \tag{3.12}$$

We observe in Figure 3-4 that $w(\theta)$ and $l(\theta, \psi)$ are periodic in θ with period π. Mathematically stated,

$$w(\theta + \pi) = w(\theta)$$

$$l(\theta + \pi, \psi) = l(\theta, w(\theta) - \psi)$$

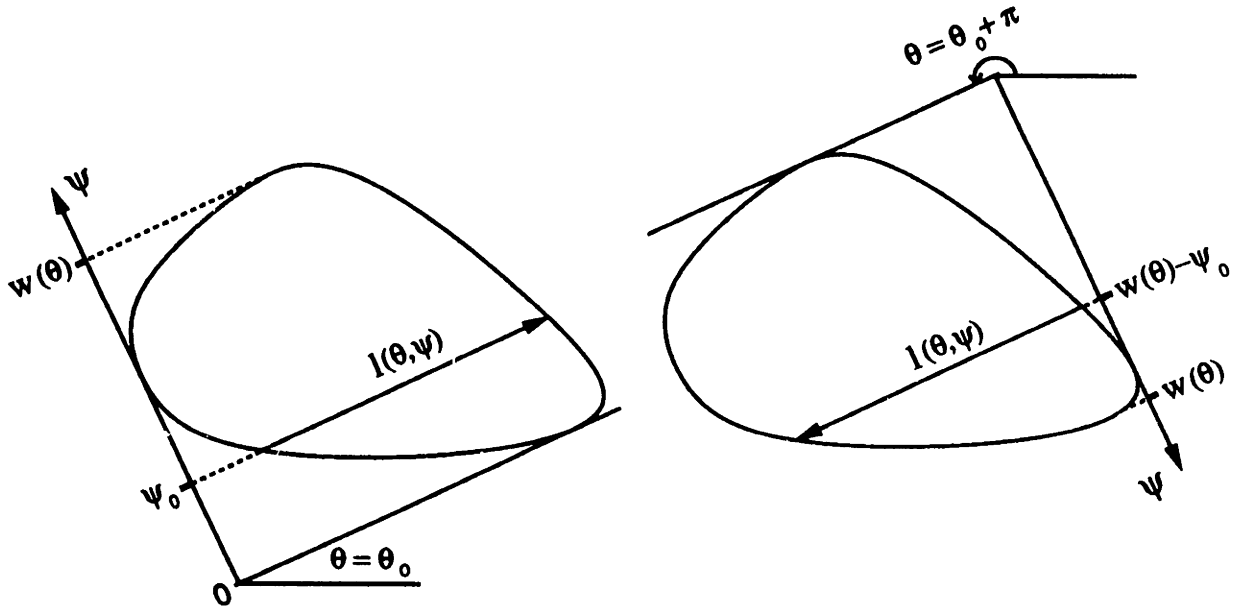

Figure 3-4: Defining a polygon in the (θ,ψ,λ) coordinate system is periodic in θ

Thus, $\int_0^{w(\theta)} l^4(\theta,\psi)d\psi$ and $\int_0^{w(\theta)} l^3(\theta,\psi)d\psi$ are also periodic in θ with period π. The expression (3.12) can be generalized to:

$$\bar{D}_E(Q) = \frac{\frac{1}{2}\int_{\theta_0}^{\theta_0+\pi}\int_0^{w(\theta)} l^4(\theta,\psi)d\psi d\theta}{\int_{\theta_0}^{\theta_0+\pi}\int_0^{w(\theta)} l^3(\theta,\psi)d\psi d\theta} \tag{3.13}$$

where θ_0 is any arbitrary value. If $\theta_0 = 0$ then we obtain:

$$\bar{D}_E(Q) = \frac{\frac{1}{2}\int_0^{\pi}\int_0^{w(\theta)} l^4(\theta,\psi)d\psi d\theta}{\int_0^{\pi}\int_0^{w(\theta)} l^3(\theta,\psi)d\psi d\theta} \tag{3.14}$$

3.1.3 Applying the expression to a circle and a square

We will test the expression on a circle with radius a. From previously derived results, we anticipate the expected distance to be $\frac{128a}{45\pi}$. Figure 3-5 illustrates the width function $w(\theta)$ and length function $l(\theta,\psi)$ for a circle of radius a. We state these functions mathematically as:

$$w(\theta) = 2a$$

$$l(\theta,\psi) = 2\sqrt{a^2-(\psi-a)^2}$$

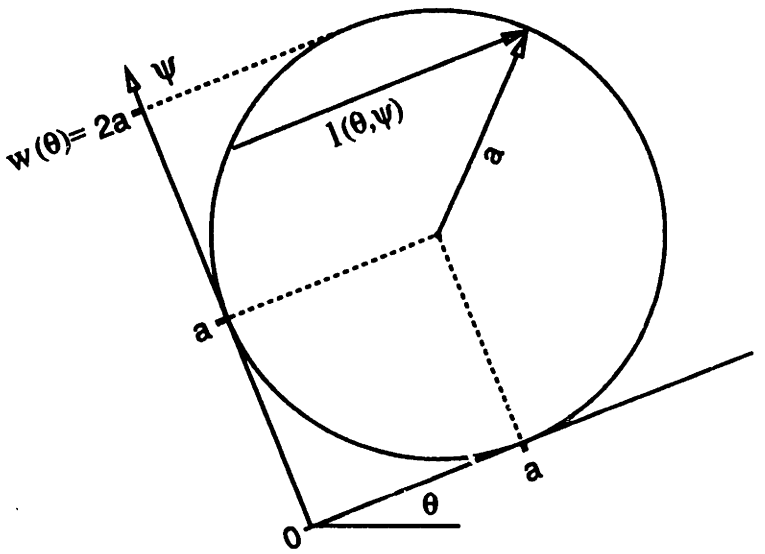

Figure 3-5: A circle of radius a defined in terms of $w(\theta)$ and $l(\theta, \psi)$

which, with Equation 3.14, produces:

$$\bar{D}_E(circle_a) = \frac{\frac{1}{2}\int_0^{2\pi}\int_0^{2a}(2\sqrt{a^2-(\psi-a)^2})^4 d\psi d\theta}{\int_0^{2\pi}\int_0^{2a}(2\sqrt{a^2-(\psi-a)^2})^3 d\psi d\theta} \tag{3.15}$$

Because none of the terms in the integrands contain θ, we can drop the θ integrals from the numerator and the denominator. Also, because the semi-circles on either side of $\psi = a$ are identical, we only need to consider one of them. Figure 3-6 shows only the upper

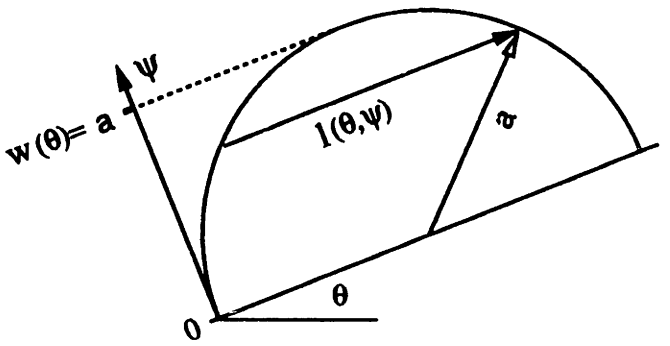

Figure 3-6: Half of the circle

semi-circle. Mathematically, we shift ψ by a, resulting in:

$$w(\theta) = a$$

$$l(\theta, \psi) = 2\sqrt{a^2 - \psi^2}$$

and so,

$$\bar{D}_E(circle_a) = \frac{\frac{1}{2}2^4 \int_0^a (a^2 - \psi^2)^2 d\psi}{2^3 \int_0^a (a^2 - \psi^2)\sqrt{a^2 - \psi^2} d\psi}$$

$$\bar{D}_E(circle_a) = \frac{\int_0^a a^4 - 2a^2\psi^2 + \psi^4 d\psi}{a^2 \int_0^a \sqrt{a^2 - \psi^2} d\psi - \int_0^a \psi^2 \sqrt{a^2 - \psi^2} d\psi}$$

$$\bar{D}_E(circle_a) = \frac{\left[a^4\psi - \frac{2}{3}a^2\psi^3 + \frac{1}{5}\psi^5\right]_0^a}{a^2\left[\frac{\psi}{2}\sqrt{a^2 - \psi^2} + \frac{a^2}{2}\sin^{-1}\frac{\psi}{a}\right]_0^a - \left[\frac{a^4}{8}\sin^{-1}\frac{\psi}{a} - \frac{1}{8}\psi\sqrt{a^2 - \psi^2}(a^2 - \psi^2)\right]_0^a}$$

$$\bar{D}_E(circle_a) = \frac{\frac{8}{15}a^5}{\frac{a^4}{a}\frac{\pi}{2} - \frac{a^4}{8}\frac{\pi}{2}} = \frac{\frac{8}{15}a^5}{\frac{3}{16}a^4\pi}$$

$$\bar{D}_E(circle_a) = \frac{128a}{45\pi}$$

Therefore, the expression produces a result which agrees with the previously derived result for a circle.

We have not previously derived the expected Euclidean distance for a unit square, but from an analytic expression that already exists, we know that the result is approximately

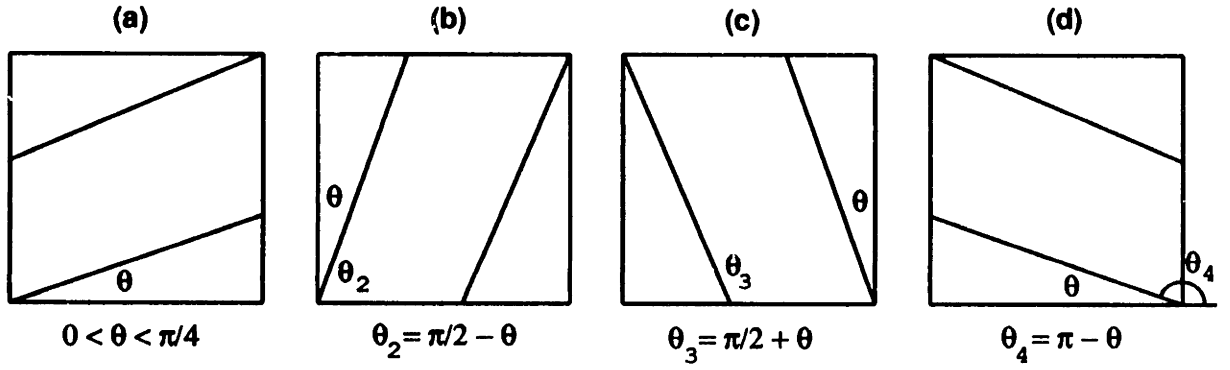

Figure 3-7: Four subranges of θ produce equivalent geometries for a square

0.521405. [2]

Due to the natural symmetry of a square, we can reduce the limits of integration of θ from being in the range from 0 to π to being in the range from 0 to $\frac{\pi}{4}$. This is because integrating θ_2 from $\frac{\pi}{4}$ to $\frac{\pi}{2}$ is equivalent to integrating θ from 0 to $\frac{\pi}{4}$ with $\theta = \frac{\pi}{2} - \theta_2$ (see Figure 3-7 (b)). Similarly, the range of θ_3 from $\frac{\pi}{2}$ to $\frac{3\pi}{4}$ can be translated to $\theta = \theta_3 - \frac{\pi}{2}$ (see Figure 3-7(c)) and the fourth subrange with θ_4 from $\frac{3\pi}{4}$ to π can be translated to $\theta = \pi - \theta_4$ (see Figure 3-7(d)).

For these reasons, we can reduce the limits of integration of θ such that the expression for the expected distance within a square is,

$$\bar{D}_E(square) = \frac{\frac{1}{2}\int_0^{\frac{\pi}{4}} \int_0^{w(\theta)} l^4(\theta,\psi) d\psi d\theta}{\int_0^{\frac{\pi}{4}} \int_0^{w(\theta)} l^3(\theta,\psi) d\psi d\theta} \tag{3.16}$$

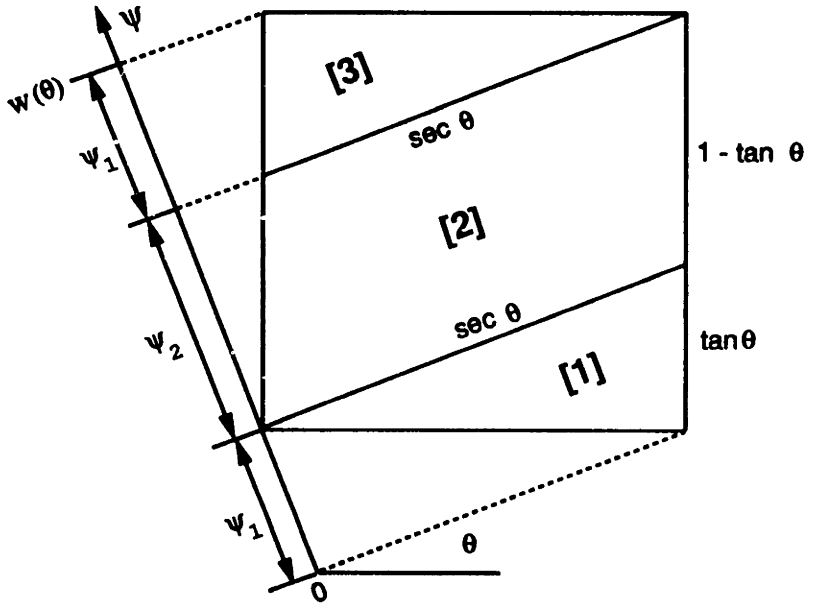

Figure 3-8: Square divided into 3 subregions

We can divide the square into three subregions labelled [1], [2], and [3] in Figure 3-8. Note that subregions [1] and [3] are symmetrical.

$$\psi_1 = \cos\theta \tan\theta = \sin\theta$$

[2]Private communication with Dr. Amedeo R. Odoni.

$$\psi_2 = \cos\theta(1 - \tan\theta) = \cos\theta - \sin\theta$$

$$l_1(\theta, \psi) = \frac{\sec\theta}{\psi_1}\psi = \frac{\sec\theta}{\sin\theta}\psi$$

$$l_2(\theta, \psi) = \sec\theta$$

The expression thus becomes:

$$\bar{D}_E(square) = \frac{\frac{1}{2}\int_0^{\frac{\pi}{4}}\left(2\int_0^{\psi_1} l_1^4(\theta,\psi)d\psi + \int_0^{\psi_2} l_2^4(\theta,\psi)d\psi\right)d\theta}{\int_0^{\frac{\pi}{4}}\left(2\int_0^{\psi_1} l_1^3(\theta,\psi)d\psi + \int_0^{\psi_2} l_2^3(\theta,\psi)d\psi\right)d\theta} \tag{3.17}$$

$$\bar{D}_E(square) = \frac{\frac{1}{2}\int_0^{\frac{\pi}{4}} \sec^4\theta\left(\frac{2}{\sin^4\theta}\int_0^{\sin\theta}\psi^4 d\psi + \int_0^{\cos\theta-\sin\theta} d\psi\right)d\theta}{\int_0^{\frac{\pi}{4}} \sec^3\theta\left(\frac{2}{\sin^3\theta}\int_0^{\sin\theta}\psi^3 d\psi + \int_0^{\cos\theta-\sin\theta} d\psi\right)d\theta}$$

$$\bar{D}_E(square) = \frac{\frac{1}{2}\int_0^{\frac{\pi}{4}} \sec^4\theta\left(\frac{2\sin^5\theta}{5\sin^4\theta} + \cos\theta - \sin\theta\right)d\theta}{\int_0^{\frac{\pi}{4}} \sec^3\theta\left(\frac{2\sin^4\theta}{4\sin^3\theta} + \cos\theta - \sin\theta\right)d\theta}$$

$$\bar{D}_E(square) = \frac{\frac{1}{2}\int_0^{\frac{\pi}{4}} \sec^3\theta - \frac{3}{5}\sin\theta\sec^4\theta d\theta}{\int_0^{\frac{\pi}{4}} \sec^2\theta - \frac{1}{2}\sin\theta\ \sec^3\theta d\vartheta}$$

Since:

$$\int \sec^3\theta d\theta = \frac{\sec\theta\tan\theta}{2} + \frac{1}{2}\ln\left|\frac{1+\sin\theta}{\cos\theta}\right|$$

$$\int \sin\theta\sec^4\theta d\theta = \frac{1}{3}\sec^3\theta$$

$$\int \sec^2\theta d\theta = \tan\theta$$

$$\int \sin\theta\sec^3\theta d\theta = \frac{1}{2}\sec^2\theta$$

then,

$$\bar{D}_E(square) = \frac{\frac{1}{2}\left[\frac{\sec\theta\tan\theta}{2} + \frac{1}{2}\ln\left|\frac{1+\sin\theta}{\cos\theta}\right| - \frac{1}{5}\sec^3\theta\right]_0^{\frac{\pi}{4}}}{\left[\tan\theta - \frac{1}{4}\sec^2\theta\right]_0^{\frac{\pi}{4}}}$$

$$\bar{D}_E(square) = \frac{\frac{1}{2}\left(\frac{1}{2}\sqrt{2} + \frac{1}{2}\ln\left|\frac{1+\frac{1}{\sqrt{2}}}{\frac{1}{\sqrt{2}}}\right| - \frac{2}{5}\sqrt{2} + \frac{1}{5}\right)}{1 - 2\cdot\frac{1}{4} + \frac{1}{4}}$$

$$\bar{D}_E(square) = \frac{\frac{1}{2}(\frac{1}{10}\sqrt{2} + \frac{1}{2}\ln|\sqrt{2}+1| + \frac{1}{5})}{\frac{3}{4}}$$

$$\bar{D}_E(square) = \frac{\sqrt{2}}{15} + \frac{1}{3}\ln|\sqrt{2}+1| + \frac{2}{15}$$

$$\bar{D}_E(square) = 0.52140543\ldots$$

With our expression verified for the unit square and for a circle, we will continue in the next section to develop a method for computing the expected Euclidean distance in any convex polygon.

3.2 Expected Euclidean Distance in a Convex Polygon

In the previous section, we derived an expression (3.14) for finding the expected Euclidean distance in a convex planar region. However, in order to obtain an analytic result, two conditions must be met – that we can express the region in terms of the width function $w(\theta)$ and the length function $l(\theta, \psi)$, and that when these functions are inserted into the expression, the integral can be evaluated.

We have successfully applied the expression to find the expected distance in a circle and a square. For such special regions, we can directly write expressions for $w(\theta)$ and $l(\theta, \psi)$. However, it is not immediately clear how these functions can be expressed for arbitrary regions, or even for arbitrary convex polygons, which have several simplifying characteristics.

In this section, we will turn to the problem of adapting the expression to find the expected Euclidean distance in convex polygons. In the previous chapter, we encountered this very problem, and focused on a method that divided a polygon into convenient pieces, then applied a mathematical expression which we developed for each piece, and then aggregated results. We will use the same "divide-and-conquer" strategy as the basis for a method to compute the expected Euclidean distance in a convex polygon.

3.2.1 Dividing the Convex Polygon

Our technique for dividing the polygon contains many of the same ideas as in the method that we presented in section 2.2.1 where we divided the convex polygon into trapezoidal

regions called "zones". The techniques are not identical, however, because the angle of a path between two random points in the Euclidean metric is continuous between $[0, \pi)$, whereas the orientation of the path between random points in the Manhattan metric is restricted to the direction of the x and y axes only.

Let us first introduce some terminology to facilitate the discussion of our method. We define a "scan line" for a vertex point to be the line that is in the θ direction that crosses the vertex point. An n-sided polygon has n vertices, thus there are n scan lines.

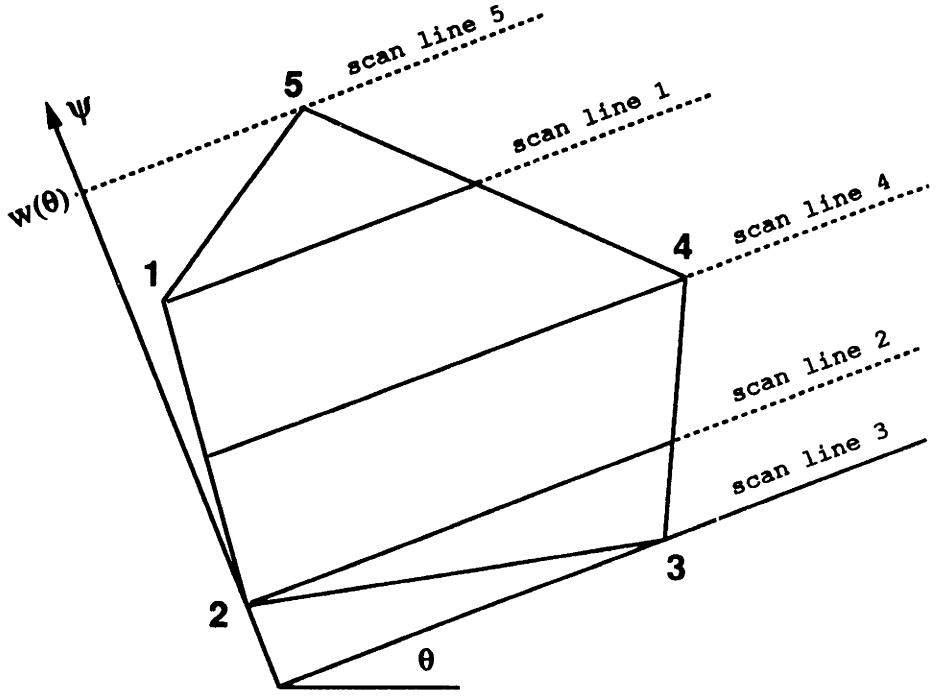

Figure 3-9: An n-sided polygon shown with the n scan lines

Let us refer to the scan line with the lowest ψ value as the "bottom scan line", and let the vertex point on which the bottom scan line falls be called the "bottom point". (In Figure 3-9, vertex 3 is the bottom point.) Similarly, let the scan line with highest ψ value be the "top scan line" and its corresponding vertex point be the "top point". (In the same figure, vertex 5 is the top point.) Now, according to our previous definitions, the bottom scan line occurs at $\psi = 0$, and the top scan line occurs at $\psi = \mathrm{w}(\theta)$. We claim that the entire polygon is enclosed between the bottom scan line and the top scan line because the

sides of a polygon are straight line segments, and so there can be no part of a polygon that extends beyond the highest (or lowest) corner value of ψ.

Given that the top and bottom scan lines enclose the entire polygon, the remaining $n-2$ scan lines divide the polygon into $n-1$ regions. We will call these regions "facets". (When two scan lines coincide, we say that there still exists a facet between them, even though the facet has no thickness.) Each of these facets has the desirable property that two of its sides are scan lines, therefore are both parallel and in the θ direction. Also, the other sides are necessarily two straight line segments. This is because a polygon is bounded by straight line segments connecting vertex points, and because there can be no vertex points between adjacent scan lines (since every vertex point is on a scan line). In short, there are $n-1$ facets each of which is trapezoidal. (Note that the two facets on the ends are triangular, but they may be considered to be trapezoids where one of the parallel sides has length 0.)

Having divided a polygon into facets, we shall define the "configuration" of the facets

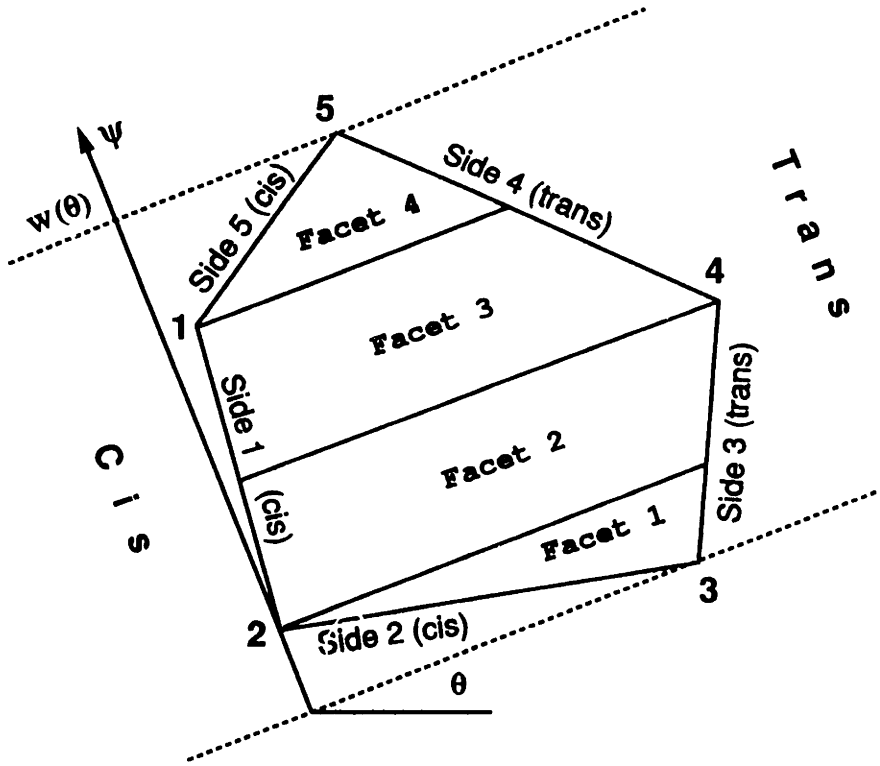

Figure 3-10: Configuration of facets

for a given value of θ to be the arrangement of facets. Each facet is defined by four lines: a

lower scan line, an upper scan line (with the higher value of ψ), a "cis" side, and a "trans" side. The cis side is on the near side of the polygon relative to the direction of θ. The trans side is on the far side of the polygon relative to θ (thus having a larger value of \). In Figure 3-10, the sides which are "cis" are sides 2, 1, and 5; sides 3 and 4 are "trans". Vertex points 3, 2, and 1 are cis while vertex points 4 and 5 are trans. (The bottom point and top point can be classified as either cis or trans, but arbitrarily we will call the bottom point cis and the top point trans.)

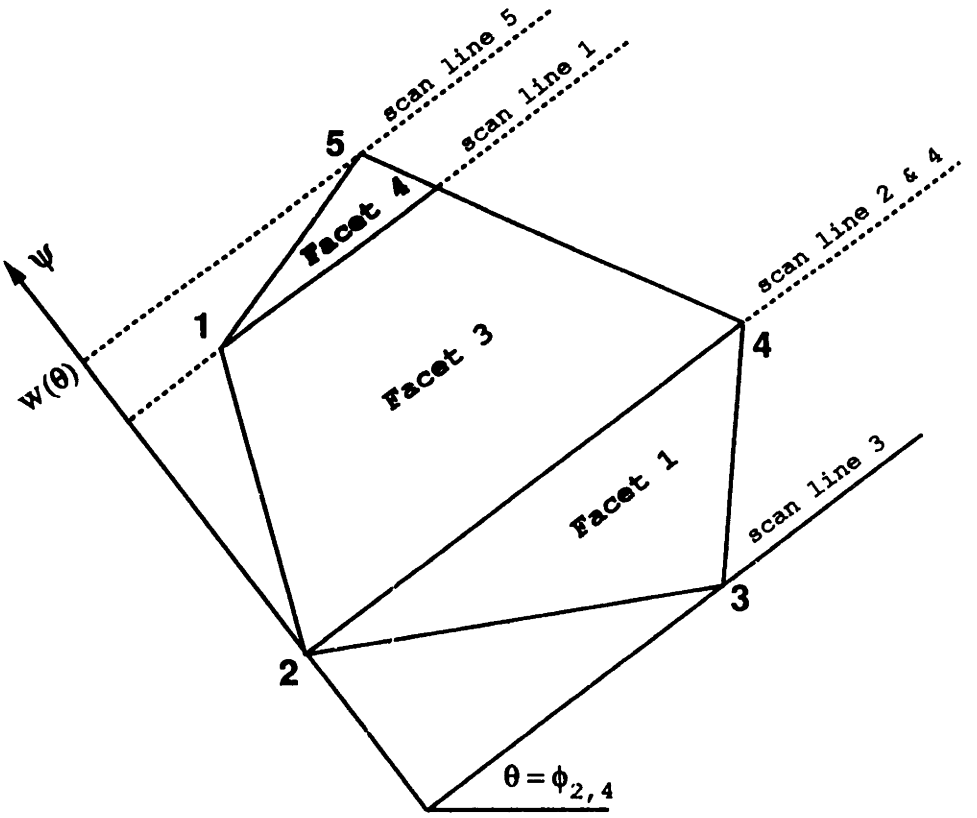

Figure 3-11: Conjunction between vertex points 2 and 4

The configuration of the four facets in Figure 3-10 is as follows:

	Lower Scan Line	Upper Scan Line	Cis Side	Trans Side
Facet 1	3	2	2	3
Facet 2	2	4	1	3
Facet 3	4	1	1	4
Facet 4	1	5	5	4

As θ is increased, the size and shape of each facet will change, but the configuration remains the same until two scan lines coincide. When this happens, a facet disappears (has no thickness). The situation when two scan lines coincide is called a "conjunction". Figure 3-11 shows the polygon at a conjunction. (Note that facet 2 has disappeared.) The value of θ is called the "conjunction angle" $\phi_{a,b}$ between the vertex points a and b that correspond to the coincidental scan lines.

When θ increases beyond the conjunction angle, a second configuration is formed (shown in Figure 3-12) as follows:

	Lower Scan Line	Upper Scan Line	Cis Side	Trans Side
Facet 1	3	4	2	3
Facet 2	4	2	2	4
Facet 3	2	1	1	4
Facet 4	1	5	5	4

As θ is further increased, the configuration will remain until scan lines from another pair of vertex points coincide–in other words, when θ reaches another "conjunction angle". We once again observe that between any pair of successive conjunction angles, the configuration (*i.e.* the arrangement of facets each defined by a lower and upper scan line and a cis and trans side) does not change. We claim that this observation reflects a situation that is true in general.

Up to now, we have given an intuitive description and introduced the related terminology on how a convex polygon might be represented by a "configuration" and how it might be divided into "facets". Eventually, we will present formally an algorithm that generates all

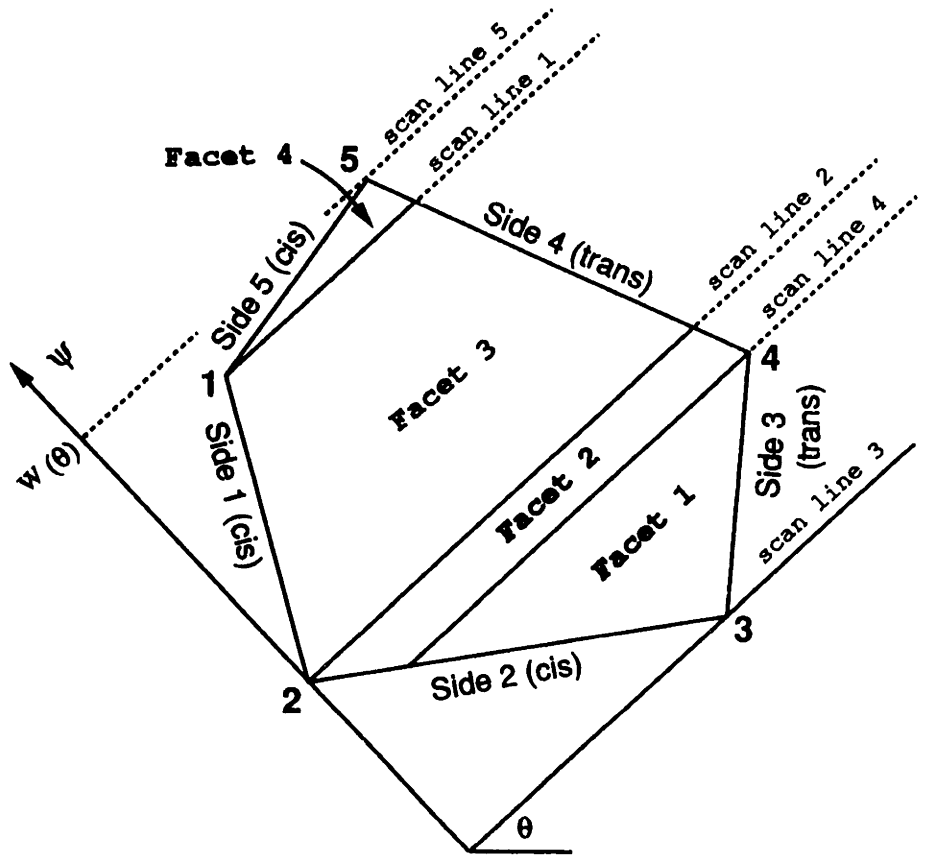

Figure 3-12: A second configuration of facets

of the configurations and facets and will solve the expected distance problem. For now, let us show that a decomposition of a polygon in terms of configurations and facets can be derived from our previous results.

Firstly, let us consider all conjunction angles $\phi_{a,b}$ between vertex points a and b such that the angle $\phi_{a,b}$ is defined to be in the range $0 \leq \phi_{a,b} < \pi$. Knowing that one conjunction angle occurs for each pair of vertex points, the number of conjunction angles C is thus:

$$C = \frac{n(n-1)}{2}$$

Let us arrange the conjunction angles in ascending order and refer to them as ϕ_i where:

$$0 \leq \phi_1 \leq \phi_2 \leq \ldots \leq \phi_C < \pi$$

We also define ϕ_0 so that:

$$\phi_0 = \phi_C - \pi$$

Due to our observation that a unique configuration exists between any pair of successive conjunction angles, C represents the number of configurations in the range from ϕ_0 through ϕ_C. We will refer to configuration i as the configuration that exists between the conjunction angles ϕ_{i-1} and ϕ_i.

From Equation 3.13, we can choose any value for the lower limit of integration (as long as the upper limit of integration has the value of π greater than the lower limit). It is convenient for us to choose a range of integration over θ that begins with a lower limit that is a particular conjunction angle ϕ_0.

$$\bar{\mathrm{D}}_E(Q) = \frac{\frac{1}{2}\int_{\phi_0}^{\phi_0+\pi}\int_0^{\mathrm{w}(\theta)} \mathrm{I}^4(\theta,\psi)d\psi d\theta}{\int_{\phi_0}^{\phi_0+\pi}\int_0^{\mathrm{w}(\theta)} \mathrm{I}^3(\theta,\psi)d\psi d\theta} \tag{3.18}$$

Dividing the range over θ into the C subranges representing each configuration,

$$\bar{\mathrm{D}}_E(polygon) = \frac{\frac{1}{2}\sum_{i=1}^{C}\int_{\phi_{i-1}}^{\phi_i}\int_0^{\mathrm{w}(\theta)} \mathrm{I}^4(\theta,\psi)d\psi d\theta}{\sum_{i=1}^{C}\int_{\phi_{i-1}}^{\phi_i}\int_0^{\mathrm{w}(\theta)} \mathrm{I}^3(\theta,\psi)d\phi d\theta} \tag{3.19}$$

In a similar fashion to the way that we divided the range of integration over θ into C subranges, we can divide the range of integration over ψ into subranges. We define $\mathrm{p}_{i,j}(\theta)$ as the ψ values of the scan lines in configuration i that are ordered so that:

$$\mathrm{p}_{i,1}(\theta) \leq \mathrm{p}_{i,2}(\theta) \leq \ldots \leq \mathrm{p}_{i,n}(\theta)$$

We recall that each facet occurs between a pair of adjacent scan lines, and so each facet j in configuration i is bounded by a lower scan line at $\psi = \mathrm{p}_{i,j}(\theta)$ and an upper scan line at $\psi = \mathrm{p}_{i,j+1}(\theta)$ as depicted in Figure 3-13. From the definition of ψ and $\mathrm{w}(\theta)$, it is true that:

$$\mathrm{p}_{i,1}(\theta) = 0$$

$$\mathrm{p}_{i,n}(\theta) = \mathrm{w}(\theta)$$

We can now divide the integration over ψ into F subranges where F is the number of facets:

$$F = n - 1$$

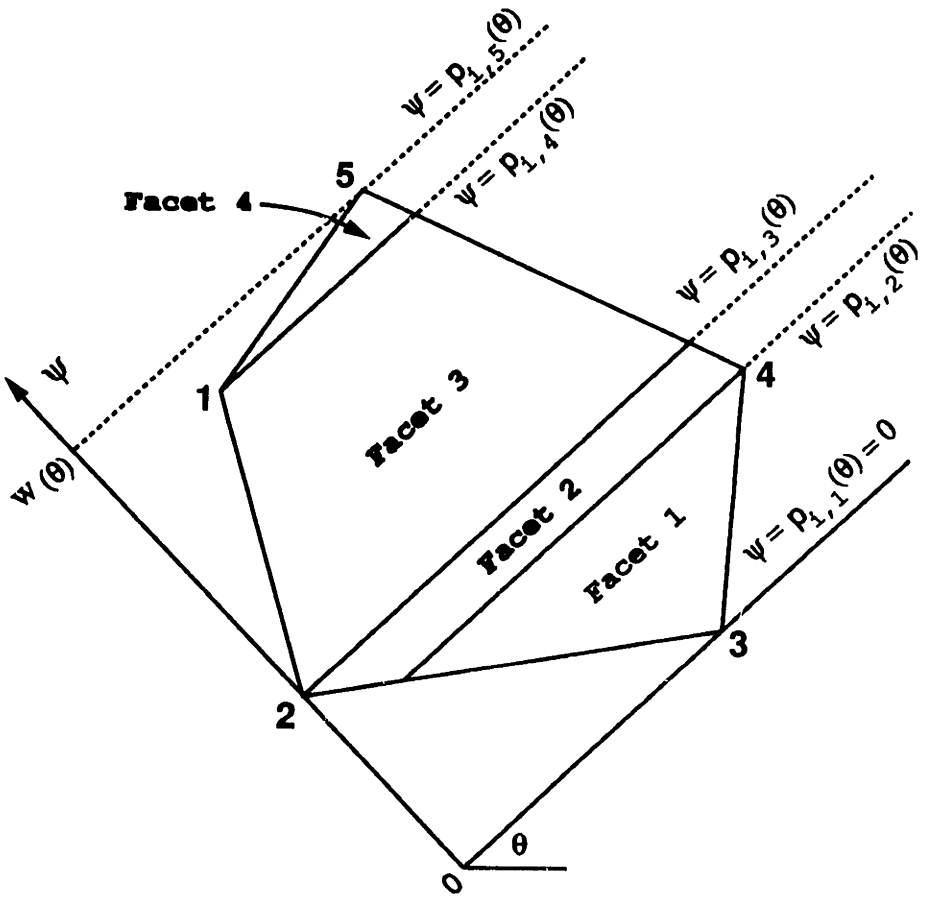

Figure 3-13: Scan lines defined with $p_{i,j}(\theta)$

We express Equation 3.19 as:

$$\bar{D}_E(polygon) = \frac{\frac{1}{2}\sum_{i=1}^{C}\int_{\phi_{i-1}}^{\phi_i}\sum_{j=1}^{F}\int_{p_{i,j}(\theta)}^{p_{i,j+1}(\theta)} I^4(\theta,\psi)d\psi d\theta}{\sum_{i=1}^{C}\int_{\phi_{i-1}}^{\phi_i}\sum_{j=1}^{F}\int_{p_{i,j}(\theta)}^{p_{i,j+1}(\theta)} I^3(\theta,\psi)d\psi d\theta}$$

Moving the summations to the left of the expressions,

$$\bar{D}_E(polygon) = \frac{\frac{1}{2}\sum_{i=1}^{C}\sum_{j=1}^{F}\int_{\phi_{i-1}}^{\phi_i}\int_{p_{i,j}(\theta)}^{p_{i,j+1}(\theta)} I^4(\theta,\psi)d\psi d\theta}{\sum_{i=1}^{C}\sum_{j=1}^{F}\int_{\phi_{i-1}}^{\phi_i}\int_{p_{i,j}(\theta)}^{p_{i,j+1}(\theta)} I^3(\theta,\psi)d\psi d\theta} \tag{3.20}$$

We can restate the above equation as:

$$\bar{D}_E(polygon) = \frac{\frac{1}{2}\sum_{i=1}^{C}\sum_{j=1}^{F} G(i,j)}{\sum_{i=1}^{C}\sum_{j=1}^{F} H(i,j)} \tag{3.21}$$

with:

$$G(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_{p_{i,j}(\theta)}^{p_{i,j+1}(\theta)} I^4(\theta,\psi) d\psi d\theta$$

$$H(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_{p_{i,j}(\theta)}^{p_{i,j+1}(\theta)} I^3(\theta,\psi) d\psi d\theta$$

The algorithm that finds the conjunction angles ϕ_i and the arrangements of the facets in the configurations is called the "Configurator Method" and is presented in the following subsection. In subsection 3.2.3, the "Facet Term Expressions" for G and H will be solved in closed-form.

3.2.2 Configurator Method

The Configurator Method essentially is the algorithm that performs the summations of the "Facet Term Expressions" in Equation 3.21. The method sets up the summations by finding the conjunction angles and the associated configuration of facets.

The first task is to arrange the conjunction angles. We compute the conjunction angle between points a and b as $\phi_{a,b} = \tan^{-1}\left(\frac{y_a - y_b}{x_a - x_b}\right)$ for every possible pair of vertex points a and b. Having computed the C conjunction angles $\phi_{1,2}, \phi_{1,3}, \ldots, \phi_{2,3}, \phi_{2,4}, \ldots, \phi_{n-1,n}$, we re-arrange them in increasing order such that $0 \leq \phi_1 \leq \phi_2 \leq \ldots \leq \phi_C \leq \pi$.

The procedure to set up the arrangement of facets for a configuration is more involved. The C conjunction angles that we have arranged define the θ bounds for exactly C configurations, of which $C-1$ are bounded within successive pairings of conjunction angles: $\phi_1 \leq \theta \leq \phi_2, \phi_2 \leq \theta \leq \phi_3, \ldots, \phi_{C-1} \leq \theta \leq \phi_C$. The remaining configuration, which we will refer to as the "first configuration", occurs in the remaining θ subrange $\phi_0 \leq \theta \leq \phi_1$ where $\phi_0 = \phi_C - \pi$. The principal interest that we have in a configuration is the arrangement of facets. We will find the arrangement of facets by first finding the sequence of n vertex points whose scan lines define the $n-1$ facets.

For a configuration i, let us define a "sequence" $\{P_{i,1}, P_{i,2}, \ldots, P_{i,n}\}$ which is the ordering of the n vertex points such that the points are ranked by ψ value. Referring back to Figure 3-13, the first point $P_{i,1}$ in the sequence would be point 3, the second point $P_{i,2}$ in the sequence would be point 4, *etc.*.

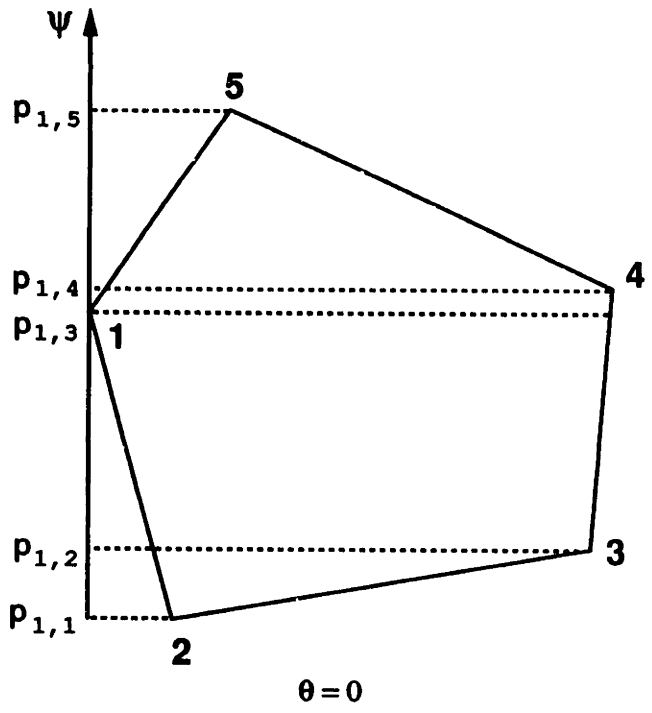

Figure 3-14: A sample polygon shown when $\theta = 0$

Let us begin by finding the sequence for the first configuration. By definition, the first configuration occurs for θ in the subrange $\phi_0 \leq \theta \leq \phi_1$. We know that $\phi_0 < 0$ since $\phi_0 = \phi_C - \pi$ and that $\phi_C < \pi$. We also know that $\phi_1 \geq 0$ thus the situation when $\theta = 0$ occurs in the first configuration. The polygon in Figure 3-14 is shown when $\theta = 0$.

By comparing Figure 3-14 to Figure 3-15, which shows how the same polygon might be defined in Cartesian coordinates, we note that ranking the vertex points by ψ value when $\theta = 0$ is equivalent to ranking the vertex points of the polygon by their y value. Thus, we can generate the sequence of points for the first configuration by ranking the vertex points by their y value. (When points share the same y coordinate, we rank them in increasing x value.)

Once we have determined the sequence of vertex points for the first configuration, we can easily set up the arrangement of facets. The lower scan line of facet 1 passes through the first point in the sequence at $\psi = \mathrm{p}_{1,1}(\theta)$; the upper scan line of facet 1 passes through the second point in the sequence at $\psi = \mathrm{p}_{1,2}(\theta)$; the lower scan line of facet 2 is the same

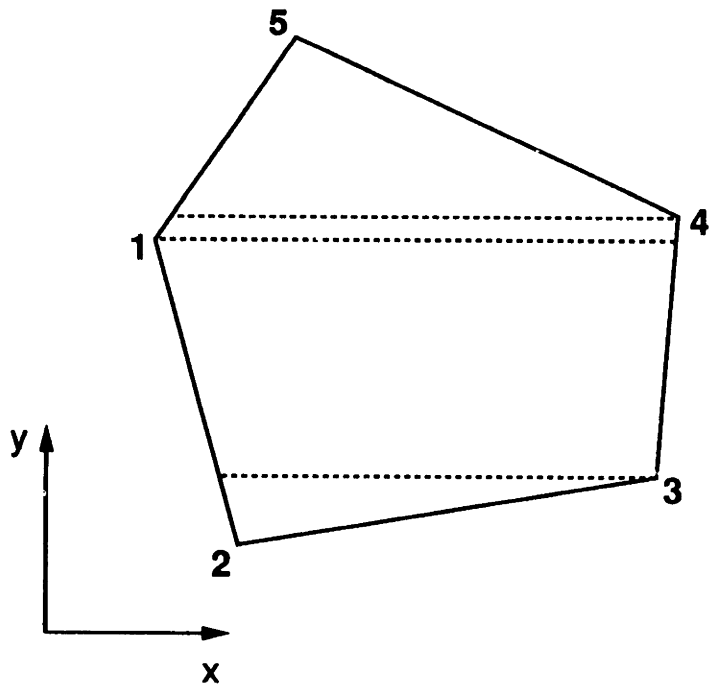

Figure 3-15: Same polygon defined in (x, y) coordinates

as the upper scan line of facet 1; the upper scan line of facet 2 passes through the third point in the sequence at $\psi = p_{1,3}(\theta)$; and so on. For the polygon in Figure 3-16, we show the complete sequence for the first configuration in Table 3-1. By convention, the bottom

Facet j	Vertex Point	Pivot Side	Crosses Side
1	3	Cis	3
2	2	Cis	3
3	4	Trans	1
4	1	Cis	4
5	5	Trans	5

Table 3-1: Sequence of Vertex Points for the first configuration

point (first point in the sequence) is on the cis side, and the top point (last point in the sequence) is on the trans side.

Up to this point, we have stated that a "sequence" of n points is sufficient for defining the

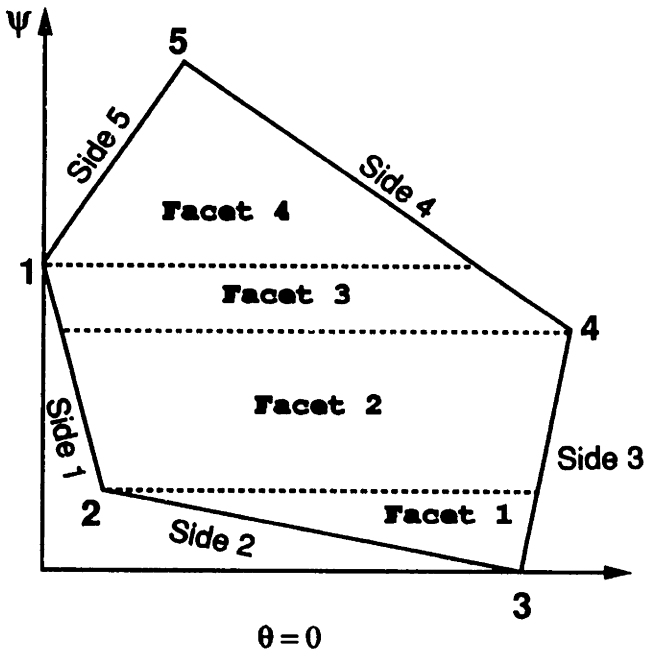

Figure 3-16: Sample polygon in first configuration

$n-1$ facets in a configuration, and we have shown how the sequence for the first configuration can be found. We must now find the sequences for the remaining $C-1$ configurations. One way to accomplish this is to repeat the process as in the first configuration but instead of ranking each vertex points by its y-coordinate, we rank them by the quantity $y\cos\theta_i - x\sin\theta_i$ where θ_i is an angle that is strictly within the θ bounds of the configuration.

However, it is computationally simpler to incrementally find the sequence of vertex points in a configuration from the sequence in the previous configuration. For example, with the same polygon from Figure 3-16, we can use the sequence for the first configuration (where θ is in the range $\phi_0 \leq \theta \leq \phi_1$) given in Table 3-1 to find the sequence in the second configuration in Table 3-2. (where θ is in the range $\phi_1 \leq \theta \leq \phi_2$). The second configuration is shown in Figure 3-17. By comparing the sequences in the two tables, we make the following four observations. There are exactly two points that change their position in the sequence (vertex points 2 and 4 in the example). Those two points are the same two points whose conjunction angle separated the two configurations. The two points are adjacent in the sequences. One of the points is on the cis side and the other is on the

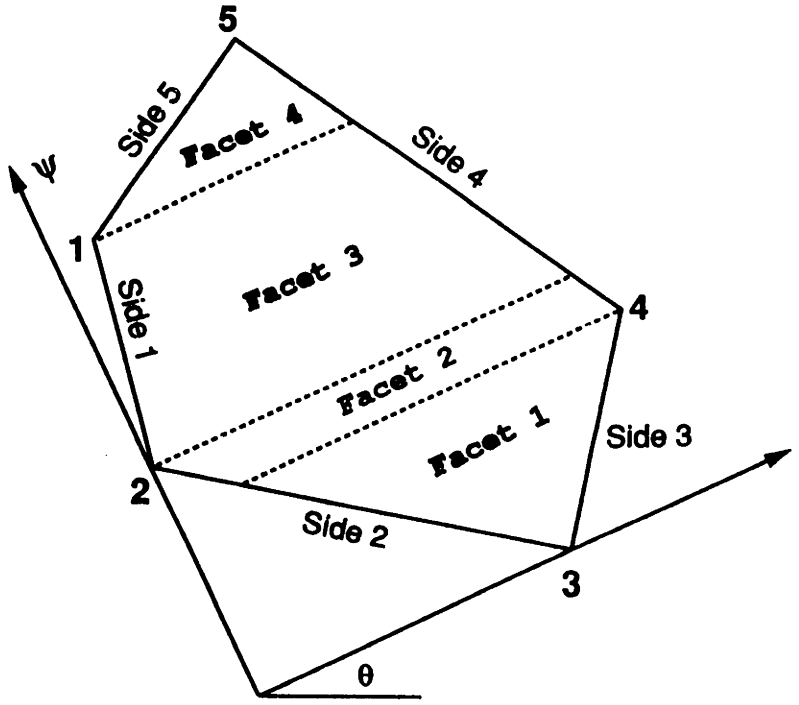

Figure 3-17: Sample polygon in second configuration

trans side. The values in the "crosses side" category increased by 1 for the two points. Without lengthy explanation, these observations are not surprising given our definition of a configuration, and of a conjunction angle, and the fact that the indicies for the sides in the polygon are arranged counterclockwise. In fact, we can in general find the new sequence given: *(i)* the sequence of the current configuration, and *(ii)* the information that the conjunction separating the next configuration and the current configuration occurs between

Facet j	Vertex Point	Pivot Side	Crosses Side
1	3	Cis	3
2	4	Trans	2
3	2	Cis	4
4	1	Cis	4
5	5	Trans	5

Table 3-2: Sequence of Vertex Points for the second configuration

points a and b as follows:

1. Swap the two points in the sequence together with the corresponding pivot and "crosses side" information.
2. Increment the value of the "crosses side" for the two points. (Note: incrementing a "crosses side" value of n results in a value of 1.)
3. If either of the two points is a bottom point or a a top point, then ensure that the bottom point's pivot type is cis and the top point's pivot type is trans.

This concludes our treatment of the Configurator Method. To summarize the method:

1. Find the conjunction angles between all pairs of points.
2. Arrange them so that:

   $$0 \leq \phi_1 \leq \phi_2 \leq \ldots \leq \phi_C \leq \pi$$

   Define ϕ_0 such that:

   $$\phi_0 = \phi_C - \pi$$

3. Generate the sequence of points for the first configuration by the vertex points in increasing y-coordinate order.
4. For each facet in the configuration's arrangement of facets apply the formulas for the "Facet Term Expressions" (to be derived in the next subsection). The resulting values are summed into the numerator and denominator terms as in Equation 3.21.
5. Generate the next configuration from the current configuration and go back to step 4.
6. When all C configurations have been considered, then the expected Euclidean distance for the polygon can be found by dividing the summation of numerator terms by the summation of denominator terms (and factoring in a constant).

The remaining step is to find the "Facet Term Expressions" themselves, which we will discuss next.

3.2.3 Facet Term Expressions

Earlier in the chapter, we derived Equation 3.21 which consists of a summation of numerator terms divided by a summation of denominator terms. Each instance of a numerator and denominator term corresponds to a trapezoidal subregion of the polygon called a "facet". In the previous subsection, we presented an algorithm for systematically generating all possible facets. In this subsection, we will evaluate in closed form the expressions, which we call the "Facet Term Expressions" for the numerator term and denominator term.

We previously defined a facet j in configuration i by the lower scan line (which crosses the vertex point $P_{i,j}$ at $\psi = \mathrm{p}_{i,j}(\theta)$), the upper scan line (which crosses the vertex point $P_{i,j+1}$ at $\psi = \mathrm{p}_{i,j+1}(\theta)$), the cis side, the trans side, and the θ subrange for the configuration $\phi_{i-1} \leq \theta \leq \phi_i$. There are two classes of facets each requiring separate analysis. A "same-based" facet is one where the vertex points $P_{i,j}$ and $P_{i,j+1}$ are on the same side of the polygon–in other words, where the vertex points are either both on the cis side or both on the trans side. (The example in Figure 3-18(a) happens to have both vertex points on the

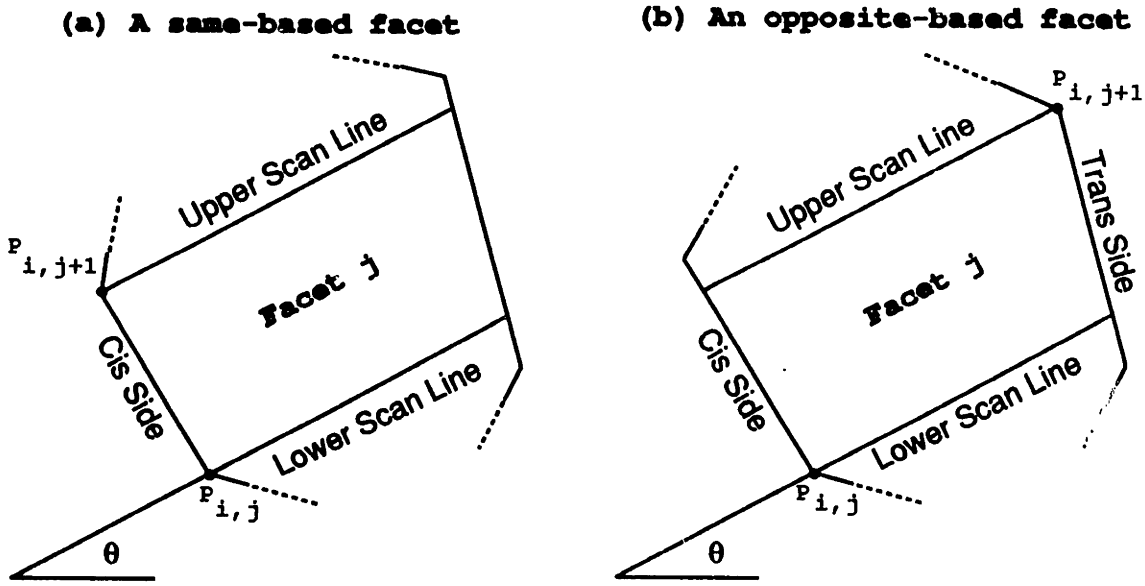

Figure 3-18: (a) A same-based facet and (b) an opposite-based facet

cis side.) The other class of facets is "opposite-based". In this case, the vertex points are on opposite sides. For example, in Figure 3-18(b), the vertex point $P_{i,j}$ for the lower scan

line is on the cis side while the vertex point $P_{i,j+1}$ for the upper scan line is on the trans side.

It is convenient for us to define a facet using a notation that is specific to the particular facet. In the same way that we had defined the ψ dimension to be perpendicular to the θ direction such that the minimum value of ψ for the polygon is $\psi = 0$, we define $\psi_{i,j}$ perpendicular to the θ direction (*i.e.* in the same direction as ψ) such that the minimum value of $\psi_{i,j}$ for the facet j in configuration i is $\psi_{i,j} = 0$. In other words, $\psi_{i,j}$ is the same as ψ except that it is offset by $\mathrm{p}_{i,j}(\theta)$. We can also define a "facet width" function $\mathrm{w}_{i,j}(\theta)$ which represents the thickness of the facet in the $\psi_{i,j}$ dimension at a given value of θ. (Just as $\mathrm{w}(\theta)$ is the extent of the polygon in the ψ dimension, $\mathrm{w}_{i,j}(\theta)$ is the extent of the facet in the $\psi_{i,j}$ dimension.) Finally, we define a "facet length" function $\mathrm{l}_{i,j}(\theta, \psi_{i,j})$ which represents the extent of the facet in the λ direction at a given θ value and at a given value of $\psi_{i,j}$. Our definitions for the $\psi_{i,j}$, $\mathrm{w}_{i,j}(\theta)$, and $\mathrm{l}_{i,j}(\theta, \psi_{i,j})$ are:

$$\psi_{i,j} = \psi - \mathrm{p}_{i,j}(\theta) \tag{3.22}$$

$$\mathrm{w}_{i,j}(\theta) = \mathrm{p}_{i,j+1}(\theta) - \mathrm{p}_{i,j}(\theta)$$

$$\mathrm{l}_{i,j}(\theta, \psi_{i,j}) = \mathrm{l}(\theta, \psi_{i,j} + \mathrm{p}_{i,j}(\theta))$$

We can restate $G(i,j)$ and $H(i,j)$ in Equation 3.21 as:

$$\mathrm{G}(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^{\mathrm{w}_{i,j}(\theta)} \mathrm{l}_{i,j}^4(\theta, \psi_{i,j}) d\psi_{i,j} d\theta \tag{3.23}$$

$$\mathrm{H}(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^{\mathrm{w}_{i,j}(\theta)} \mathrm{l}_{i,j}^3(\theta, \psi_{i,j}) d\psi_{i,j} d\theta$$

Let us begin by considering a "same-based" facet which we will refer to as facet j in configuration i. As shown in the Figure 3-19, $P_{i,j}$ and $P_{i,j+1}$ are the vertex points at the lower and upper scan lines which occur at $\psi_{i,j} = 0$ and $\psi_{i,j} = \mathrm{w}_{i,j}(\theta)$ respectively. Let $\phi_{i,j,j+1}$ be the angle formed between points $P_{i,j}$ and $P_{i,j+1}$ and the $\theta = 0$ direction and let $d_{i,j,j+1}$ be the distance between $P_{i,j}$ and $P_{i,j+1}$.

Let us define the "opposite side" $S_{i \triangleright\triangleleft j}$ as the side of the polygon on the opposite side of facet j to the vertex point $P_{i,j}$. Let us also define the "altitude" from a point $P_{i,j}$ to its

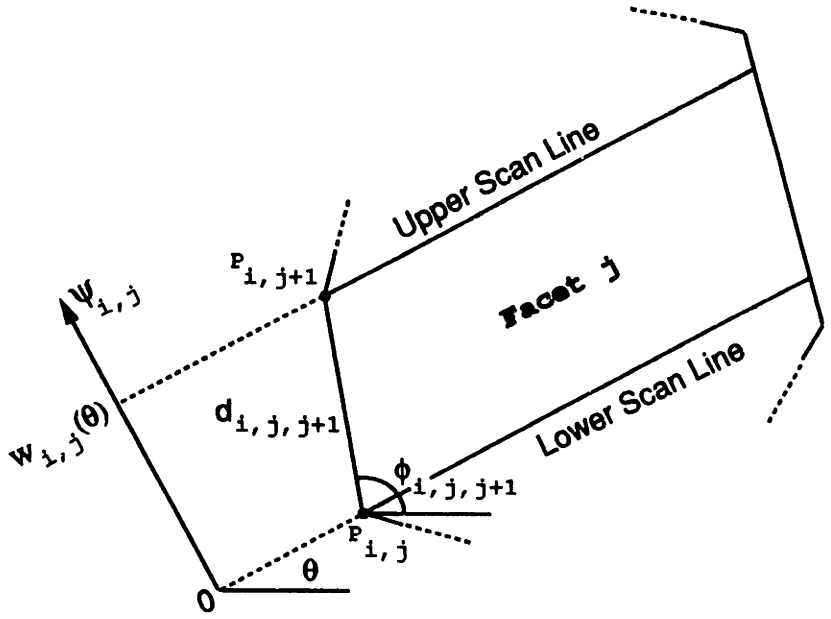

Figure 3-19: Same-based facet showing $d_{i,j,j+1}$ and $\phi_{i,j,j+1}$

"opposite side" $S_{i \triangleright\triangleleft j}$ to be the length of the line segment connecting $P_{i,j}$ to $S_{i \triangleright\triangleleft j}$ such that the line segment is perpendicular to the line containing $S_{i \triangleright\triangleleft j}$. We let $a_{i,j}$ be the altitude from the lower vertex point $P_{i,j}$ to its "opposite side", and we let $a_{i,j+1}$ be the altitude from the upper vertex point $P_{i,j+1}$ to its "opposite side". (Because the facet is "same-based", then the "opposite sides" with respect to $P_{i,j}$ and $P_{i,j+1}$ are one and the same.) Let the angle formed between this "opposite side" and the $\theta = 0$ direction be called $\phi_{i \triangleright\triangleleft j}$. We illustrate $a_{i,j}$, $a_{i,j+1}$, $S_{i \triangleright\triangleleft j}$ and $\phi_{i \triangleright\triangleleft j}$ in Figure 3-20.

For the purposes of finding $l_{i,j}(\theta, \psi_{i,j})$, we must first find the lengths of the lower and upper scan lines for the facet. We can determine the length of the lower scan line (which at $\psi_{i,j} = 0$) by studying Figure 3-21. We are given θ, $\phi_{i \triangleright\triangleleft j}$, and $a_{i,j}$. By inspection, we see that $\angle x = \phi_{i \triangleright\triangleleft j} - \frac{\pi}{2}$, the angle $\angle x$ is equal to the angle $\angle y$, and the angle $\angle z = \theta - \angle y$. By substitution,

$$\angle z = \theta - \phi_{i \triangleright\triangleleft j} + \frac{\pi}{2}$$

The length of the lower scan line is:

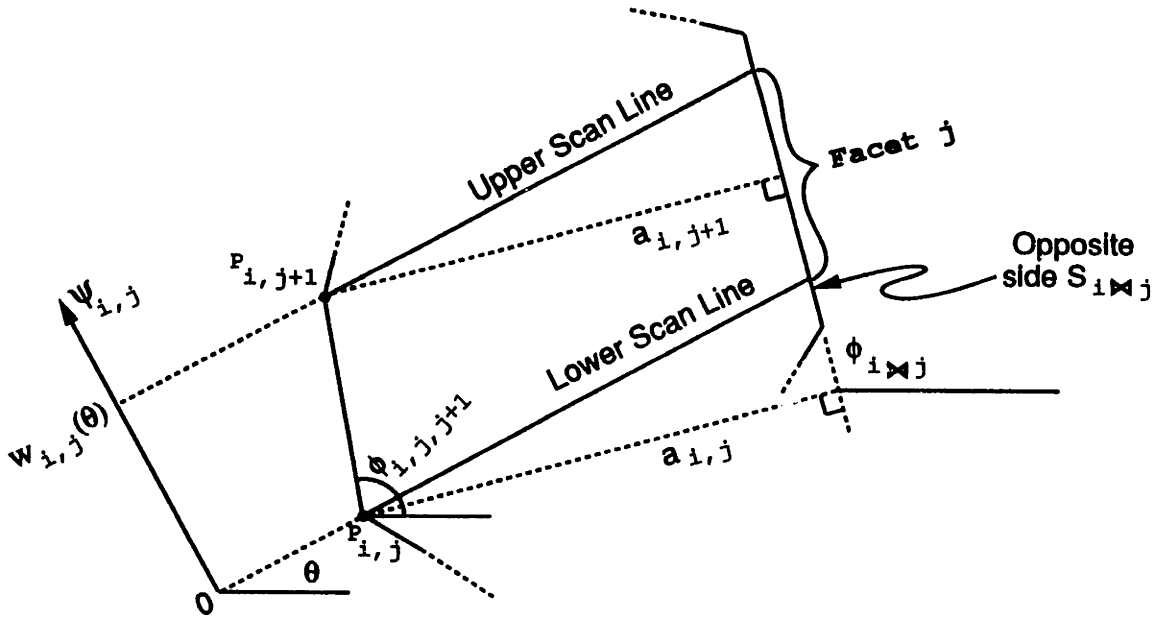

Figure 3-20: Facet showing $a_{i,j}$, $a_{i,j+1}$, $S_{i \triangleright\triangleleft j}$, and $\phi_{i \triangleright\triangleleft j}$

$$l_{i,j}(\theta, 0) = \frac{a_{i,j}}{|\cos \angle z|}$$

$$l_{i,j}(\theta, 0) = \frac{a_{i,j}}{|\cos(\theta - \phi_{i \triangleright\triangleleft j} + \frac{\pi}{2})|}$$

$$l_{i,j}(\theta, 0) = \frac{a_{i,j}}{|\sin(\theta - \phi_{i \triangleright\triangleleft j})|} \tag{3.24}$$

By similar reasoning, the length of the upper scan line (which is at $\psi_{i,j} = w_{i,j}(\theta)$) is:

$$l_{i,j}(\theta, w_{i,j}(\theta)) = \frac{a_{i,j+1}}{|\sin(\theta - \phi_{i \triangleright\triangleleft j})|} \tag{3.25}$$

Because the length of the polygon $l_{i,j}(\theta, \psi_{i,j})$ is clearly a linear function of $\psi_{i,j}$ within a facet, then the length of the polygon within the domain of the facet (*i.e.* $0 \leq \psi_{i,j} \leq w_{i,j}(\theta)$) can be expressed in terms of the length of the lower scan line (from Equation 3.24), the length of the upper scan line (from Equation 3.25), the value of $\psi_{i,j}$, and the facet "width" $w_{i,j}(\theta)$:

$$l_{i,j}(\theta, \psi_{i,j}) = \frac{a_{i,j}}{|\sin(\theta - \phi_{i \triangleright\triangleleft j})|} + \left(\frac{a_{i,j+1}}{|\sin(\theta - \phi_{i \triangleright\triangleleft j})|} - \frac{a_{i,j}}{|\sin(\theta - \phi_{i \triangleright\triangleleft j})|} \right) \frac{\psi_{i,j}}{w_{i,j}(\theta)}$$

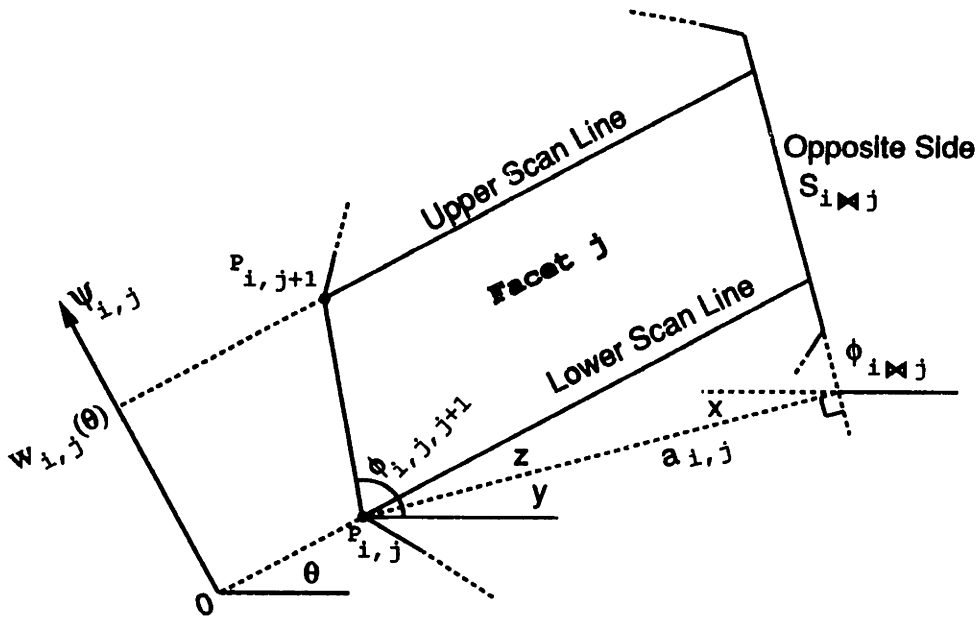

Figure 3-21: Angles for finding the length of the lower scan line

$$l_{i,j}(\theta, \psi_{i,j}) = \frac{a_{i,j}}{|\sin(\theta - \phi_{i \bowtie j})|} + \frac{a_{i,j+1} - a_{i,j}}{|\sin(\theta - \phi_{i \bowtie j})|} \frac{\psi_{i,j}}{w_{i,j}(\theta)} \tag{3.26}$$

We are now ready to solve for the numerator and denominator terms in Equation 3.23 for same-based facets. Let us begin with the numerator term G:

$$G(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^{w_{i,j}(\theta)} l_{i,j}^4(\theta, \psi_{i,j}) d\psi_{i,j} d\theta$$

Substituting from Equation 3.26:

$$G(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^{w_{i,j}(\theta)} \left(\frac{a_{i,j}}{|\sin(\theta - \phi_{i \bowtie j})|} + \frac{a_{i,j+1} - a_{i,j}}{|\sin(\theta - \phi_{i \bowtie j})|} \frac{\psi_{i,j}}{w_{i,j}(\theta)} \right)^4 d\psi_{i,j} d\theta \tag{3.27}$$

For algebraic simplicity, let us restate this equation with the following substitutions:

$$A = \frac{a_{i,j}}{|\sin(\theta - \phi_{i \bowtie j})|} \tag{3.28}$$

$$B = \frac{a_{i,j+1} - a_{i,j}}{|\sin(\theta - \phi_{i \bowtie j})| \; w_{i,j}(\theta)}$$

$$C = w_{i,j}(\theta)$$

Equation 3.27 can now be expressed as:

$$G(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^C (A + B\psi_{i,j})^4 d\psi_{i,j} d\theta$$

Solving:

$$G(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^C \left(A^4 + 4A^3 B\psi_{i,j} + 6A^2 B^2 \psi_{i,j}^2 + 4AB^3 \psi_{i,j}^3 + \frac{1}{5} B^4 \psi_{i,j}^4 \right) d\psi_{i,j} d\theta$$

$$G(i,j) = \int_{\phi_{i-1}}^{\phi_i} \left[A^4 \psi_{i,j} + 2A^3 B \psi_{i,j}^2 + 2A^2 B^2 \psi_{i,j}^3 + AB^3 \psi_{i,j}^4 + \frac{1}{5} B^4 \psi_{i,j}^5 \right]_0^C d\theta$$

$$G(i,j) = \int_{\phi_{i-1}}^{\phi_i} C \left(A^4 + 2A^3 (BC) + 2A^2 (BC)^2 + A(BC)^3 + \frac{1}{5} (BC)^4 \right) d\theta \qquad (3.29)$$

Referring back to Figure 3-19, we can express the term C which is the "width" $w(\theta)$ of the facet in the $\psi_{i,j}$ direction in terms of the length $d_{i,j,j+1}$ of the side of the facet connecting points $P_{i,j}$ and $P_{i,j+1}$:

$$C = w_{i,j}(\theta) = d_{i,j,j+1} \left| \sin(\theta - \phi_{i,j,j+1}) \right| \qquad (3.30)$$

With this expression for C, we expand the term (BC) from Equation 3.28:

$$(BC) = \frac{a_{i,j+1} - a_{i,j}}{d_{i,j,j+1} \left| \sin(\theta - \phi_{i,j,j+1}) \right| \left| \sin(\theta - \phi_{i \triangleright\!\triangleleft j}) \right|} d_{i,j,j+1} \left| \sin(\theta - \phi_{i,j,j+1}) \right|$$

$$(BC) = \frac{a_{i,j+1} - a_{i,j}}{\left| \sin(\theta - \phi_{i \triangleright\!\triangleleft j}) \right|}$$

We can now substitute the expressions for A, (BC), and C back into the expression for G in Equation 3.29:

$$G(i,j) = \int_{\phi_{i-1}}^{\phi_i} \frac{d_{i,j,j+1} \left| \sin(\theta - \phi_{i,j,j+1}) \right|}{\sin^4(\theta - \phi_{i \triangleright\!\triangleleft j})} \begin{pmatrix} a_{i,j}^4 + 2a_{i,j}^3 (a_{i,j+1} - a_{i,j}) + \\ 2a_{i,j}^2 (a_{i,j+1} - a_{i,j})^2 + a_{i,j} (a_{i,j+1} - a_{i,j})^3 \\ + \frac{1}{5} (a_{i,j+1} - a_{i,j})^4 \end{pmatrix} d\theta$$

After some algebraic manipulation, we obtain:

$$G(i,j) = \frac{d_{i,j,j+1}}{5} \left(a_{i,j}^4 + a_{i,j}^3 a_{i,j+1} + a_{i,j}^2 a_{i,j+1}^2 + a_{i,j} a_{i,j+1}^3 + a_{i,j+1}^4 \right) \int_{\phi_{i-1}}^{\phi_i} \frac{\left| \sin(\theta - \phi_{i,j,j+1}) \right|}{\sin^4(\theta - \phi_{i \triangleright\!\triangleleft j})} d\theta \qquad (3.31)$$

The absolute value signs may be dropped if we define $\hat{\phi}_{i,j,j+1}$ which is the same as $\phi_{i,j,j+1}$ "adjusted" so that $\sin(\theta - \hat{\phi}_{i,j,j+1})$ is greater than or equal to zero. Given that θ and $\phi_{i,j,j+1}$ are always in the range $(0;\pi]$, then:

$$\hat{\phi}_{i,j,j+1} = \begin{cases} \phi_{i,j,j+1} & \text{for } \theta \geq \phi_{i,j,j+1}, \\ \phi_{i,j,j+1} - \pi & \text{otherwise} \end{cases} \tag{3.32}$$

We can further simplify Equation 3.31 if the sine term in the numerator of the integral is a simple function of θ instead of the sum of θ and a constant. This can be accomplished by translating θ by $\hat{\phi}_{i,j,j+1}$:

$$\varphi_{i,1} = \phi_{i-1} - \hat{\phi}_{i,j,j+1} \tag{3.33}$$

$$\varphi_{i,2} = \phi_i - \hat{\phi}_{i,j,j+1}$$

$$\hat{\varphi} = \hat{\phi}_{i,j,j+1} - \phi_{i \bowtie j}$$

Equation 3.31 can now be stated as:

$$G(i,j) = \frac{d_{i,j,j+1}}{5}\left(a_{i,j}^4 + a_{i,j}^3 a_{i,j+1} + a_{i,j}^2 a_{i,j+1}^2 + a_{i,j} a_{i,j+1}^3 + a_{i,j+1}^4\right) \int_{\varphi_{i,1}}^{\varphi_{i,2}} \frac{\sin\theta}{\sin^4(\theta - \hat{\varphi})} d\theta \tag{3.34}$$

The key to deriving a closed-form solution for G lies in the ability to evaluate the integral above. Fortunately, this is possible: [3]

$$\int \frac{\sin\theta}{\sin^4(\theta - \hat{\varphi})} = \tag{3.35}$$

$$\frac{\begin{pmatrix} 3\cos(6\theta + 7\hat{\varphi}) + 3\cos(6\theta + 5\hat{\varphi}) - 18\cos(4\theta + 5\hat{\varphi}) - \\ 18\cos(4\theta + 3\hat{\varphi}) + 45\cos(2\theta + 3\hat{\varphi}) + 45\cos(2\theta + \hat{\varphi}) - 60\cos\hat{\varphi} \end{pmatrix} \ln\frac{1+\cos(\theta+\hat{\varphi})}{1-\cos(\theta+\hat{\varphi})} + \\ -12\cos(5\theta + 6\hat{\varphi}) - 12\cos(5\theta + 4\hat{\varphi}) + 68\cos(3\theta + 4\hat{\varphi}) \\ +4\cos(3\theta + 2\hat{\varphi}) - 120\cos(\theta + 2\hat{\varphi}) + 72\cos\theta}{6\cos(4\theta + 2\hat{\varphi}) - \cos(6\theta + 6\hat{\varphi}) - 15\cos(2\theta + 2\hat{\varphi}) + 10}$$

[3] This result and the result in (3.38) were obtained using the MACSYMA mathematical symbolic manipulation system.

We can solve for the denominator term H from Equation 3.23 in a similar fashion:

$$\mathrm{H}(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^{w_{i,j}(\theta)} \mathrm{l}_{i,j}^3(\theta, \psi_{i,j}) d\psi_{i,j} d\theta$$

Substituting from Equations 3.28:

$$\mathrm{H}(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^{C} (A + B\psi_{i,j})^3 d\psi_{i,j} d\theta$$

Solving:

$$\mathrm{H}(i,j) = \int_{\phi_{i-1}}^{\phi_i} \int_0^{C} A^3 + 3A^2 B\psi_{i,j} + 3AB^2\psi_{i,j}^2 + B^3\psi_{i,j}^3 d\psi_{i,j} d\theta$$

$$\mathrm{H}(i,j) = \int_{\phi_{i-1}}^{\phi_i} \left[A^3\psi_{i,j} + \frac{3}{2}A^2 B\psi_{i,j}^2 + AB^2\psi_{i,j}^3 + \frac{1}{4}B^3\psi_{i,j}^4\right]_0^C d\theta$$

$$\mathrm{H}(i,j) = \int_{\phi_{i-1}}^{\phi_i} C\left(A^3 + \frac{3}{2}A^2(BC) + A(BC)^2 + \frac{1}{4}(BC)^3\right) d\theta$$

Substituting the expressions for A, (BC), and C back into the equation for H:

$$\mathrm{H}(i,j) = \int_{\phi_{i-1}}^{\phi_i} \frac{d_{i,j,j+1}\left|\sin(\theta - \phi_{i,j,j+1})\right|}{\left|\sin^3(\theta - \phi_{i\triangleright\triangleleft j})\right|} \left(a_{i,j}^3 + \tfrac{3}{2}a_{i,j}^2(a_{i,j+1} - a_{i,j}) + a_{i,j}(a_{i,j+1} - a_{i,j})^2 + \tfrac{1}{4}(a_{i,j+1} - a_{i,j})^3\right) d\theta$$

$$\mathrm{H}(i,j) = \frac{d_{i,j,j+1}}{4}\left(a_{i,j}^3 + a_{i,j}^2 a_{i,j+1} + a_{i,j}a_{i,j+1}^2 + a_{i,j+1}^3\right) \int_{\phi_{i-1}}^{\phi_i} \frac{\left|\sin(\theta - \phi_{i,j,j+1})\right|}{\left|\sin^3(\theta - \phi_{i\triangleright\triangleleft j})\right|} d\theta$$

The absolute value signs may be dropped if we define $\hat{\phi}_{i\triangleright\triangleleft j}$ which is the same as $\phi_{i\triangleright\triangleleft j}$ adjusted such that $\sin(\theta - \hat{\phi}_{i\triangleright\triangleleft j})$ is greater than or equal to zero. Given that θ and $\phi_{i\triangleright\triangleleft j}$ are always in the range $(0; \pi]$, then:

$$\hat{\phi}_{i\triangleright\triangleleft j} = \begin{cases} \phi_{i\triangleright\triangleleft j} & \text{for } \theta \geq \phi_{i\triangleright\triangleleft j}, \\ \phi_{i\triangleright\triangleleft j} - \pi & \text{otherwise} \end{cases} \tag{3.36}$$

As we did earlier for G, we will eliminate the constant in the numerator of the integral by translating θ with the expressions defined in Equation 3.33:

$$\mathrm{H}(i,j) = \frac{d_{i,j,j+1}}{4}\left(a_{i,j}^3 + a_{i,j}^2 a_{i,j+1} + a_{i,j}a_{i,j+1}^2 + a_{i,j+1}^3\right) \int_{\varphi_{i,1}}^{\varphi_{i,2}} \frac{\sin\theta}{\sin^3(\theta - \hat{\varphi})} d\theta \tag{3.37}$$

where the solution to the integral is:

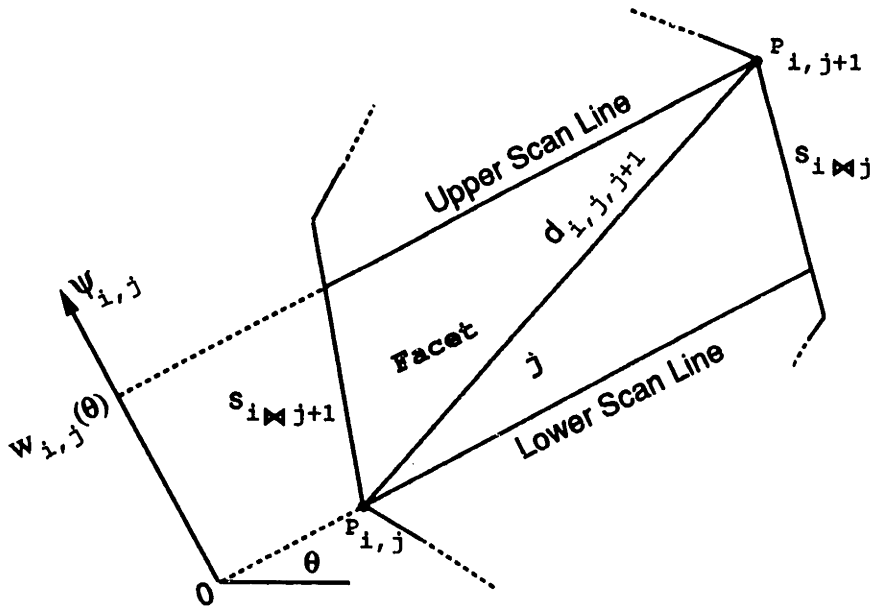

Figure 3-22: Opposite-based facet showing $S_{i\triangleright\triangleleft j}$ and $S_{i\triangleright\triangleleft j+1}$

$$\int \frac{\sin\theta}{\sin^3(\theta-\hat{\varphi})} = \frac{\sin(4\theta+5\hat{\varphi})+\sin(4\theta+3\hat{\varphi})-4\sin(2\theta+3\hat{\varphi})+4\sin\hat{\varphi}}{2\cos(4\theta+4\hat{\varphi})-8\cos(2\theta+2\hat{\varphi})+6} \tag{3.38}$$

We now have closed form solutions for the expected distance expression in same-based facets. There remains the case in which the facet is opposite-based. As shown in Figure 3-22, we define $S_{i\triangleright\triangleleft j}$ and $S_{i\triangleright\triangleleft j+1}$ to be the sides of the polygon that are on the opposite side of the facet from points $P_{i,j}$ and $P_{i,j+1}$ respectively. Figure 3-23 shows the altitudes $a_{i,j}$ and $a_{i,j+1}$ from these points to their respective opposite sides and shows the respective angles $\phi_{i\triangleright\triangleleft j}$ and $\phi_{i\triangleright\triangleleft j+1}$ formed between these sides and the $\theta = 0$ direction. We can state the lengths of the scan lines, similar to Equations 3.24 and 3.25:

$$\mathrm{l}_{i,j}(\theta,0) = \frac{a_{i,j}}{|\sin(\theta-\phi_{i\triangleright\triangleleft j})|}$$

$$\mathrm{l}_{i,j}(\theta,\mathrm{w}_{i,j}(\theta)) = \frac{a_{i,j+1}}{|\sin(\theta-\phi_{i\triangleright\triangleleft j+1})|}$$

The expression for the facet length function $\mathrm{l}_{i,j}(\theta,\psi_{i,j})$ is then:

$$\mathrm{l}_{i,j}(\theta,\psi_{i,j}) = \frac{a_{i,j}}{|\sin(\theta-\phi_{i\triangleright\triangleleft j})|} + \left(\frac{a_{i,j+1}}{|\sin(\theta-\phi_{i\triangleright\triangleleft j+1})|} - \frac{a_{i,j}}{|\sin(\theta-\phi_{i\triangleright\triangleleft j})|}\right)\frac{\psi_{i,j}}{\mathrm{w}_{i,j}(\theta)}$$

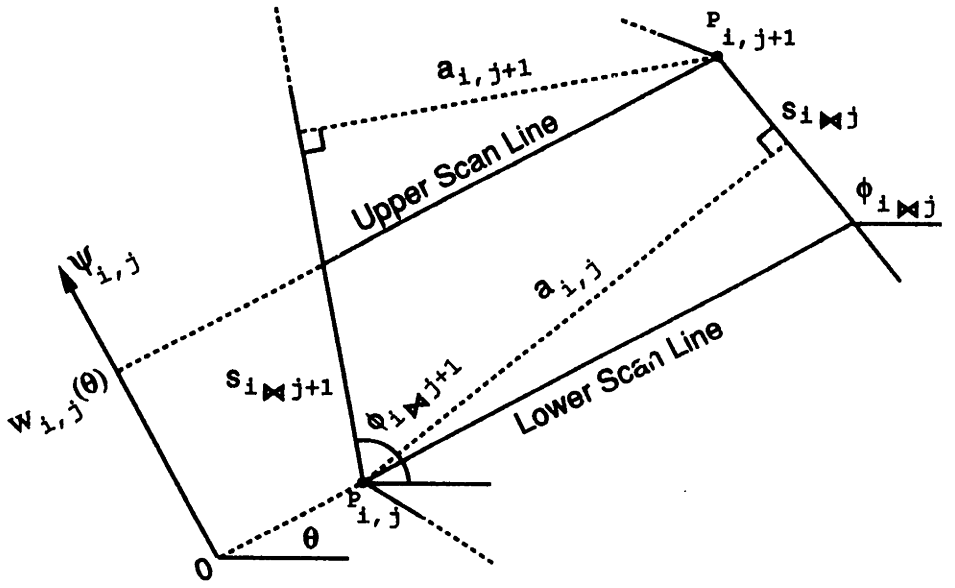

Figure 3-23: Opposite-based facet showing altitudes and angles of the sides

In fact, compared with our derivation for same-based facets, the existence of the additional factor $\sin(\theta - \phi_{i \bowtie j+1})$ complicates the expression for $l_{i,j}(\theta, \psi_{i,j})$ such that it cannot be simplified and evaluated in closed-form in Equation 3.23.

However, we observe that we can convert the G and H expressions for any opposite-based facet into G and H expressions for two same-based facets. Let us assume for the moment that the sides $S_{i \bowtie j}$ and $S_{i \bowtie j+1}$ are not parallel. Then we can find the point P_X which is the intersection of the lines of those sides as shown in Figure 3-24. Note in Figure 3-25(a) how the large triangle A forms a same-based facet with $P_{i,j}$ and P_X as the scan points and $S_{i \bowtie j+1}$ and $S_{i \bowtie j}$ as the sides. In Figure 3-25(b), the triangle B forms a same-based facet with $P_{i,j+1}$ and P_X as the scan points and $S_{i \bowtie j+1}$ and $S_{i \bowtie j}$ as the sides. The fact that both facets are same-based is due to the fact that P_X is by definition on both $S_{i \bowtie j}$ and $S_{i \bowtie j+1}$.

Essentially, we claim that the G and H expressions that we are seeking for opposite-based facet j can be found by taking the difference between the G and H expressions for the "fictitious" facets A and B. We can show this mathematically by considering the limits of the integrals for either of the expressions for G and H in Equation 3.23. If $w_X(\theta)$, which is the value of ψ for the point P_X at a given θ angle, is such that $w_X(\theta) \geq w_{i,j}(\theta)$, then:

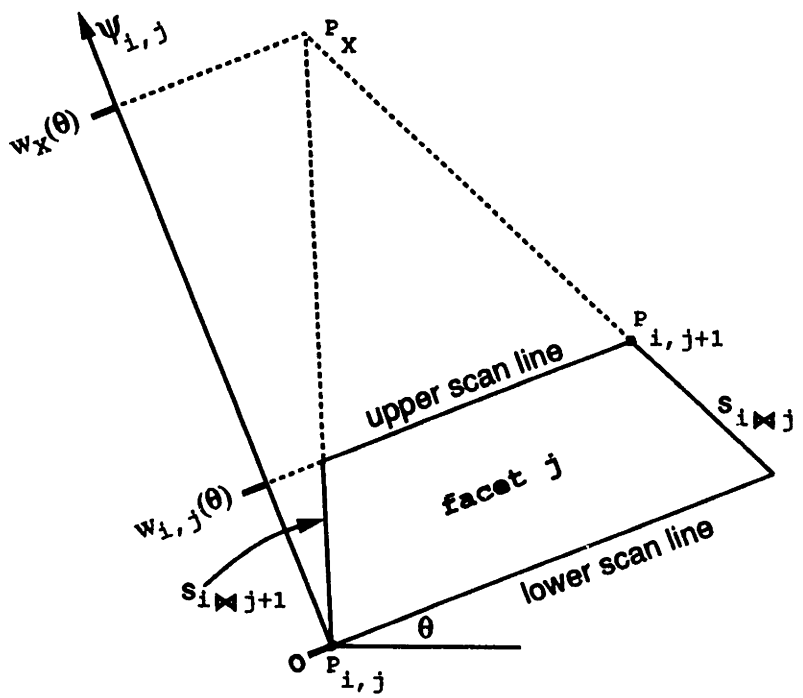

Figure 3-24: The intersection point P_X

$$\int_{\phi_{i-1}}^{\phi_i}\int_{0}^{w_{i,j}(\theta)} \cdots \, d\psi_{i,j}d\theta = \int_{\phi_{i-1}}^{\phi_i}\int_{0}^{w_X(\theta)} \cdots \, d\psi_{i,j}d\theta - \int_{\phi_{i-1}}^{\phi_i}\int_{w_{i,j}(\theta)}^{w_X(\theta)} \cdots \, d\psi_{i,j}d\theta \quad (3.39)$$

If not, then $w_X(\theta) \leq 0$. (It is not possible for $0 < w_X(\theta) < w_{i,j}(\theta)$). In this case:

$$\int_{\phi_{i-1}}^{\phi_i}\int_{0}^{w_{i,j}(\theta)} \cdots \, d\psi_{i,j}d\theta = \int_{\phi_{i-1}}^{\phi_i}\int_{w_X(\theta)}^{w_{i,j}(\theta)} \cdots \, d\psi_{i,j}d\theta - \int_{\phi_{i-1}}^{\phi_i}\int_{w_X(\theta)}^{0} \cdots \, d\psi_{i,j}d\theta$$

$$\int_{\phi_{i-1}}^{\phi_i}\int_{0}^{w_{i,j}(\theta)} \cdots \, d\psi_{i,j}d\theta = \int_{\phi_{i-1}}^{\phi_i}\int_{0}^{w_X(\theta)} \cdots \, d\psi_{i,j}d\theta - \int_{\phi_{i-1}}^{\phi_i}\int_{w_{i,j}(\theta)}^{w_X(\theta)} \cdots \, d\psi_{i,j}d\theta$$

Therefore, equation 3.39 holds true no matter where the point P_X is located. We conclude that an opposite-based facet with non-parallel sides can be evaluated by finding the intersection point P_X between the sides and creating two fictitious same-based facets for which we can find G and H. The values of G and H for the opposite-based facet are thus equal to the difference between the values of G and H respectively for the two fictitious facets.

It should be noted that the computation for these values is adversely affected when the sides are nearly (but not quite) parallel. In this situation, the location of the intersection

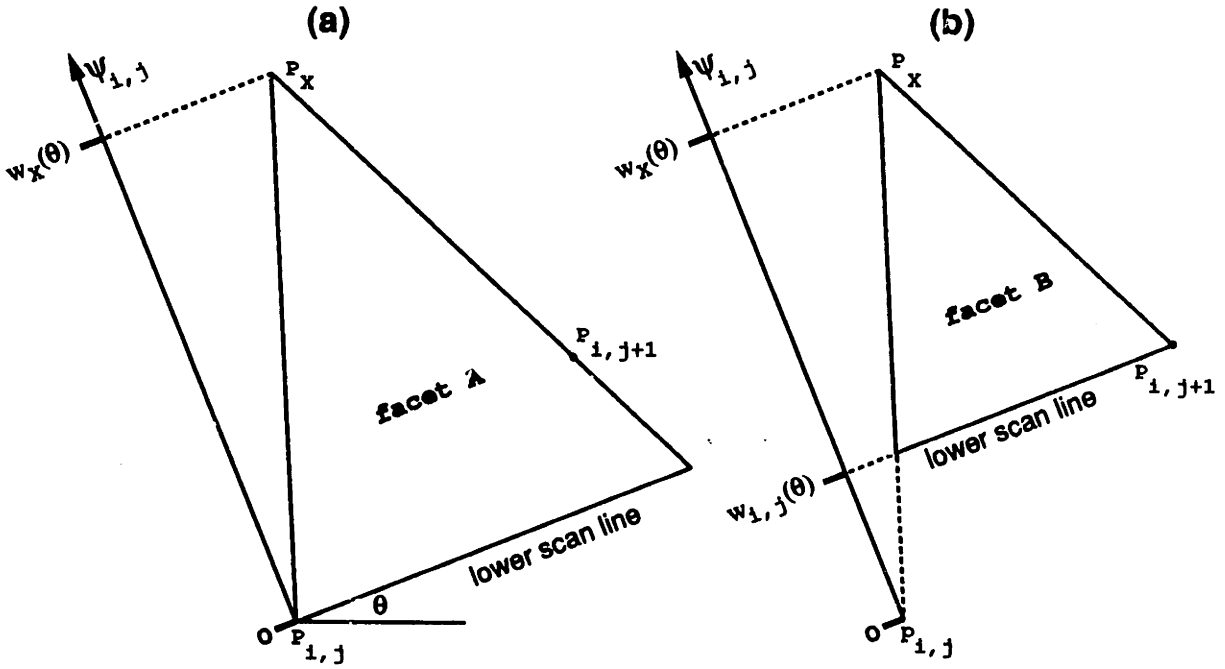

Figure 3-25: Fictitious facets formed with P_X

point P_X will be very far away from the rest of the polygon resulting in very large values for the G and H expressions for facet A and facet B. Taking the difference between very large numbers will result in roundoff errors.

If, however, the facet is opposite-based, and the opposite sides $S_{i \triangleright\triangleleft j}$ and $S_{i \triangleright\triangleleft j+1}$ *are* parallel, then the previously described procedure for handling opposite-based facets will not work. This is because the intersection point P_X cannot be found since parallel lines by definition do not intersect. However, we can treat this case as if the facet were same-based. In Figure 3-26, we observe that the facet width function $\mathrm{w}_{i,j}(\theta)$ is the same as for same-based facets (see Equation 3.30).

$$\mathrm{w}_{i,j}(\theta) = d_{i,j,j+1} \, |\sin(\theta - \phi_{i,j,j+1})|$$

Because the opposite sides are parallel, the facet length function $\mathrm{l}_{i,j}(\theta, \psi_{i,j})$ is the same for all values of $\psi_{i,j}$ in the facet. Since we have already obtained the length of the bottom scan line in Equation 3.24, we will use that for the facet length function.

$$\mathrm{l}_{i,j}(\theta, \psi_{i,j}) = \frac{a_{i,j}}{|\sin(\theta - \phi_{i \triangleright\triangleleft j})|}$$

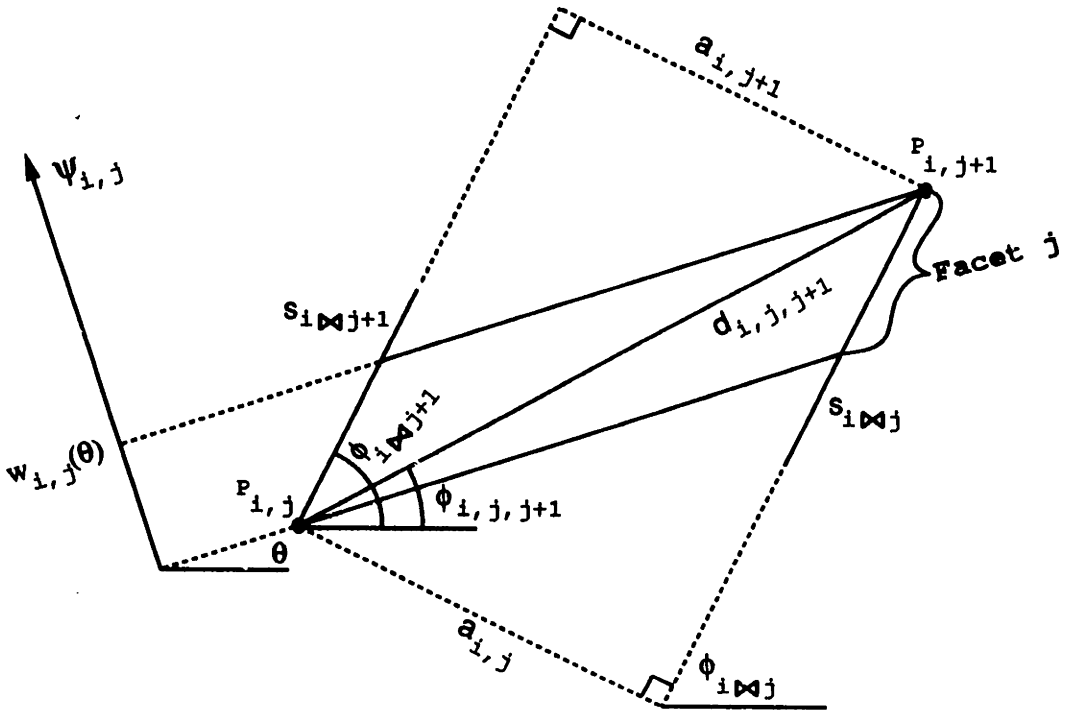

Figure 3-26: Opposite-based facet with parallel "opposite sides"

Thus, the case where the facet is opposite-based but the opposite sides are parallel reduces to the scenario where the facet is same-based.

In summary, we have shown that the definition of any facet can be manipulated so that the Facet Term Expressions G and H can be evaluated. The key steps in preparation for evaluating the Facet Term Expressions are:

1. Identify whether or not the facet is same-based or opposite-based. If the facet is opposite-based, but the opposite sides $S_{i\bowtie j}$ and $S_{i\bowtie j+1}$ are parallel, then treat the facet as if it were same-based.

2. If it is opposite-based but the sides $S_{i\bowtie j}$ and $S_{i\bowtie j+1}$ are not parallel, then find the intersection point P_X of the two sides, and compute the Facet Term Expressions as the difference between the Facet Term Expressions for the facet bounded by the two sides from $P_{i,j}$ to P_X and the Facet Term Expressions for the facet bounded by the two sides from $P_{i,j+1}$ to P_X. The Facet Term Expressions for the two facets are calculated according to step 3.

3. Determine the following quantities: the altitudes $a_{i,j}$ and $a_{i,j+1}$ from the facet's vertex points to their respective opposite sides, the distance $d_{i,j,j+1}$ between the two vertex points, the angle $\phi_{i,j,j+1}$ between the points $P_{i,j}$, $P_{i,j+1}$ and the $\theta = 0$ direction, and the angle $\phi_{i \triangleright\triangleleft j}$ between the opposite side of the facet to $P_{i,j}$ and the $\theta = 0$ direction. Then find the "adjusted" ϕ values $\hat{\phi}_{i,j,j+1}$ from Equation 3.32 and $\hat{\phi}_{i \triangleright\triangleleft j}$ from Equation 3.36. Then evaluate G and H using Equations 3.33, 3.34, 3.35, 3.37 and 3.38.

This concludes the discussion of our method for finding the expected Euclidean distance in a convex polygon.

3.2.4 Analysis of the method

We will now discuss the complexity of the algorithm with respect to the number of sides of the polygon n. The value of C is the number of unique configurations which is $\frac{n(n-1)}{2}$. Sorting the C conjunction angles has a complexity of order $\mathcal{O}(C \log C)$ which translates to $\mathcal{O}\left(n^2 \log n^2\right)$. This operation, however, is not as complex as the number of terms that are summed for each of the $n-1$ facets in C configurations. Because each facet requires computations of order $\mathcal{O}(1)$ (the "Facet Term Expressions"), the total complexity of the algorithm is $\mathcal{O}\left(n^3\right)$.

3.3 Conclusions

In this chapter, we studied the problem of finding the expected distance between two random points in a planar region using the Euclidean distance metric. Even for the simplest of shapes defined entirely through the Cartesian coordinate system, direct methods may be unable to produce results in closed form. However, we presented an alternative approach where one random point is distributed in polar coordinates and the other is distributed in a quasi-Cartesian coordinate system and derived an expression for the expected Euclidean distance. This expression was successfully adapted to a circle and to a square producing results that matched previously known ones.

We proceeded by describing a method for dividing a convex polygon into trapezoidal regions called facets and showed how a comprehensive enumeration of all configurations of

facets is equivalent to distributing two random points in a polygon according to our derived expression for the expected Euclidean distance. The derived expression was then evaluated in closed form for any definition of a facet generated by our method. We have implemented a computer program that follows our method for finding the expected Euclidean distance in any convex polygon. The listing for the program is given in Appendix A.3.1, and some numerical examples are given in Appendix A.3.2.

Regarding the generality of our results, the expression for the expected Euclidean distance can be applied to convex planar regions other than polygons and the circle. First, it must be possible to express the width function $w(\theta)$ and the length function $l(\theta, \psi)$ that define the region. In all but the simplest of regions, this will involve some decomposition procedure (much like the Configurator Method) to divide the integration over the region into various θ and ψ subranges. Furthermore, the resulting integrand(s) must be integrable in order to obtain closed form results.

An interesting extension to our method would be to be able to find the expected Euclidean distance in concave polygons. We will briefly discuss this topic in the Appendix B.

Chapter 4

Conclusions

We have studied the problem of finding the expected distance between two random points in a polygon. More precisely, we have investigated the problem of finding analytic expressions for the expected value of the distance between two identically, uniformly, and independently distributed random points in a polygonal region.

The measure of distance between two points depends upon the distance metric. In Chapter 2, we studied the problem of finding expected distances in the Manhattan or rectangular distance metric, and we derived an analytic expression for any polygon using that distance metric. In Chapter 3, we considered the same problem except with the Euclidean or straight-line distance metric, and we derived an analytic expression for convex polygons only.

In Appendix A, we present implementations of the methods in computer programs. We provide some numerical examples of the methods using these programs.

Even though the methods for evaluating the expected distance were different for the two distance metrics, the same "divide-and-conquer" strategy was employed. The polygon was first divided into simpler regions so that the expected distance expressions could be evaluated.

As suggestions for further work, there remains the problem of finding the expected Euclidean distance between random points in a concave polygon. In Appendix B, we discuss this problem briefly. Also in the appendix is an implementation of a Monte Carlo simulation for finding the expected distance between two random points. The Monte Carlo analysis is

only preliminary, and a complete analysis is left as a possible topic of study.

There are other areas for further investigation that are closely related. While this thesis focused entirely on the expected value (the first moment of expectation) of the distance, the higher moments of expectation such as the variance are also of importance. Besides the Manhattan and Euclidean distance metrics, there are other measures of distance that may be considered. We have studied the expected distance problem in polygons, but there are other types of planar regions that are worth studying. For instance, polygonal regions are, by definition, simply-connected, but regions that contain voids may more accurately reflect real situations for some applications. The assumption that the random points are uniformly distributed may be relaxed so that subregions of the polygon may have different probability densities. (In an urban operations research application, the different probability densities may be useful for modelling different demand densities within an urban region.) Solving these problems will likely make use of the "divide-and-conquer" strategy and some of the expressions and partitioning methods presented in this thesis.

Appendix A

Computer Program Implementations

In this appendix, we present computer programs that implement the numerical methods described in this thesis. The appendix is divided into four sections containing: (1) basic geometric routines that are fundamental to the computational manipulation of polygons, (2) the implementation of the method for finding the expected Manhattan distance from Chapter 2, (3) the implementation for the Euclidean metric from Chapter 4, and (4) a Monte Carlo simulation for the purpose of verifying our numerical results.

The computer programs are written in Turbo Pascal Version 4.0. Turbo Pascal is an implementation of the Pascal computer language which runs under the PC-DOS/MS-DOS operating system. Our programs deliberately avoid taking advantage of some of the language extensions that are offerred by Turbo Pascal. With a few minor exceptions that are noted in the source code, the programs conform to ANSI Pascal standards.

For the purposes of evaluating the speed of program execution, the elapsed time are included with the examples, which have all been performed on an AT&T PC 6300. The PC 6300 is an IBM PC-XT clone that uses an Intel 8086 processor running at 8 MHz. The particular machine used did not have an 8087-2 numerical coprocessor (which probably would reduce the computation time by over an order of magnitude).

A.1 Basic Geometric Routines

The basic geometric routines are simple computational geometry utility routines, as well as system-specific routines. The computational geometry routines include basic polygon indexing routines, polygon validity test, and if a line does not cross the perimeter of the polygon. There also routines of a system-specific nature that have been isolated in this module. In particular, the routines include input/output with the user, CPU usage measurement, floating-point equality test, and random number generation.

A.1.1 Pascal Source Code

```
{-------------------------------------------------------------------------
The source code shown below is a group of data definitions and routine
definitions that are used by other programs (particularly, the MANHAT (for
computing the expected Manhattan distance between two random points in a
polygon), EUCLID (for computing the expected Euclidean distance between two
random points in a convex polygon), and MCARLO (a Monte Carlo simulation to
estimate the expected Manhattan and Euclidean distances between two random
points in a polygon.)  Note that the source code below cannot be compiled
stand-alone.

Written in Turbo Pascal, Version 4.0.
Runs under PC-DOS/MS-DOS Version 2.0 or later.
Runs on IBM PC or compatible.
Written by Arthur Hsu, 1984 (revised January 1990).
-------------------------------------------------------------------------}

USES DOS;                       { Turbo Pascal unit for access to
                                  "GetTime" routine }
CONST
  max_points = 20;              { maximum number of points in polygon }
TYPE
  point_range = 1..max_points;
  point_array = ARRAY[point_range] OF REAL;
  point_set = SET OF point_range;
  string_type = STRING[31];     { string type in ANSI Pascal, but in Standard
                                  Pascal, use "PACKED ARRAY[1..31] OF CHAR" }

VAR
  n_points : point_range;       { number of points in polygon }
  x, y : point_array;           { coordinates of points in polygon }
  operation_count : INTEGER;    { for computational statistics }

  polygon_area : REAL;
  initial_time, elapsed_time : LONGINT;  { LONGINT is a 32-bit integer }
```

```
  output_file : TEXT;

{-------------------------------------------------------------------------
This function takes the index of a point and returns the index of the next
point in the polygon in the counter-clockwise direction. }

FUNCTION plus_1 (index : point_range) : point_range;

BEGIN
  IF index < n_points THEN
    plus_1 := index + 1
  ELSE
    plus_1 := 1
END;  { plus_1 }

{-------------------------------------------------------------------------
This function takes the index of a point and returns the index of the next
point in the polygon in the clockwise direction. }

FUNCTION minus_1 (index : point_range) : point_range;

BEGIN
  IF index > 1 THEN
    minus_1 := index - 1
  ELSE
    minus_1 := n_points
END;  { minus_1 }

{-------------------------------------------------------------------------
This function returns the magnitude of the cross-product of the vector from
(x1;y1) to (x2;y2) and the vector from (x2;y2) to (x3;y3). }

FUNCTION crossproduct (x1, y1, x2, y2, x3, y3 : REAL) : REAL;

BEGIN
  crossproduct := (x1 - x2)*(y2 - y3) - (y1 - y2)*(x2 - x3)
END;

{-------------------------------------------------------------------------
IFF the line from (x1;y1) to (x2;y2) does not cross or touch any of the
line segments in the polygon except for the line segments specified in the
set of excluded sides.  Note: 'excluded' contains indicies that represent

the line segment from the point number (given by the index) the next point
in the polygon in the counter-clockwise direction. }

FUNCTION uncrossed (x1, y1, x2, y2 : REAL; excluded : point_set) : BOOLEAN;

VAR
  valid : BOOLEAN;  { if the line has not been crossed by any side so far }
```

```
i, j : point_range;

{-------------------------------------------------------------------------
IFF the line segment from (x1;y1) to (x2;y2) is separate from the line
segment from (x3;y3) to (x4;y4) -- that is, if the line segments neither
cross nor touch. }

FUNCTION separate (x1, y1, x2, y2, x3, y3, x4, y4 : REAL) : BOOLEAN;

VAR
  cross_123, cross_124, _1_2_on_same_side_of_3_4 : REAL;

  {-----------------------------------------------------------------------
  This routine works in principle by comparing the length of the line
  from (x_end1, y_end1) to (x_end2, y_end2) to the sum of the lines from
  the testpoint to the ends of the line segment.  If they are equal, then
  the test point must lie on the line (therefore, 'in_between' returns
  TRUE).  For reasons of computational efficiency, it is unnecessary to
  evaluate the Euclidean distance formula (involving a SQRT operation)
  since the Manhattan distance formula is sufficient for comparisons if
  it is known that the three points are collinear. }

  FUNCTION in_between (x_test, y_test, x_end1, y_end1, x_end2, y_end2 :
      REAL) : BOOLEAN;

  BEGIN
    in_between := ABS (x_end1 - x_end2) + ABS (y_end1 - y_end2) =
        ABS (x_end1 - x_test) + ABS (y_end1 - y_test) +
        ABS (x_end2 - x_test) + ABS (y_end2 - y_test)
  END;  { in_between }

{-------------------------------------------------------------------------}

BEGIN  { separate }
  cross_123 := crossproduct (x1, y1, x2, y2, x3, y3);
  cross_124 := crossproduct (x1, y1, x2, y2, x4, y4);

  IF cross_123 * cross_124 > 0 THEN  { points 3 and 4 are on same side }
    separate := TRUE                 {   of line from points 1 to 2    }

  ELSE BEGIN
    _1_2_on_same_side_of_3_4 := crossproduct (x3, y3, x4, y4, x1, y1) *
        crossproduct (x3, y3, x4, y4, x2, y2);

    IF _1_2_on_same_side_of_3_4 > 0 THEN
      separate := TRUE               { points 1 and 2 on same side }

    ELSE IF (cross_123 <> 0) OR (cross_124 <> 0) OR
        (_1_2_on_same_side_of_3_4 <> 0) THEN
      separate := FALSE              { lines completely intersect }
```

```
{ For the remaining tests below, the points are necessarily collinear.
  Test where on the line that the line segment from points 1 to 2 falls
  relative to line segment from points 3 to 4. }
      ELSE IF in_between (x3, y3, x1, y1, x2, y2) THEN
        separate := FALSE
      ELSE IF in_between (x4, y4, x1, y1, x2, y2) THEN
        separate := FALSE
      ELSE IF in_between (x1, y1, x3, y3, x4, y4) THEN
        separate := FALSE
      ELSE
        separate := TRUE
    END
  END;  { separate }

{-------------------------------------------------------------------------}

BEGIN  { uncrossed }
  valid := TRUE;
  i := n_points;
  j := 1;

  WHILE valid AND (j <= n_points) DO BEGIN
{   Ensure that line is separate from side of polygon unless excluded }
    IF NOT (i IN excluded) THEN
      valid := separate (x1, y1, x2, y2, x[i], y[i], x[j], y[j]);
    i := j;
    j := j + 1
  END;

  uncrossed := valid
END;  { uncrossed }

{-------------------------------------------------------------------------
If the polygon is such that the points are clockwise order, the points may
be re-arranged by swapping point 1 with point "n_point", point 2 with point
"n_point - 1", etc..  This routine is used by "get_points" and in the
MANHAT program. }

PROCEDURE reverse_order;

VAR
  half_way, p : point_range;
  temp_x, temp_y : REAL;

BEGIN
  half_way := n_points DIV 2;
  FOR p := 1 TO half_way DO BEGIN  { swap coordinates between "p" and }
    temp_x := x[p];                {   "npoints + 1 - p"               }
    temp_y := y[p];
```

```
    x[p] := x[n_points + 1 - p];
    y[p] := y[n_points + 1 - p];
    x[n_points + 1 - p] := temp_x;
    y[n_points + 1 - p] := temp_y
  END
END;  { reverse_order }

{------------------------------------------------------------------------
This routine gets the definition of the polygon from the user.  It also
verifies that the polygon is valid (with sides that do not intersect),
ensures that the polygon is convex if that is required by "must_be_convex"
parameter, ensures that the points defining the polygon are arranged in
counter-clockwise order, and outputs the polygon to an output file. }

PROCEDURE get_points (must_be_convex : BOOLEAN; filename : string_type);

VAR
  p : point_range;

  {------------------------------------------------------------------------
  This function tests if the polygon with 'n_points' sides specified in the
  arrays 'x' and 'y' is convex.  The method used is to look at the angle at
  every vertex of the polygon formed by the two sides adjacent to the
  vertex and to determine if the angle there is convex.  (The test for
  convexity does not assume that the points in the polygon are arranged in
  counter-clockwise order, so the sign of using the crossproduct must not
  always be positive, but rather, must always be the same.) }

  FUNCTION convex_polygon_test : BOOLEAN;

  VAR
    p : point_range;
    convex_counter_clockwise, convex_clockwise : BOOLEAN;
    xprod : REAL;

  BEGIN
    p := 1;
    convex_counter_clockwise := TRUE;
    convex_clockwise := TRUE;

    WHILE (convex_counter_clockwise OR convex_clockwise) AND
        (p <= n_points) DO BEGIN
      xprod := crossproduct (x[minus_1 (p)], y[minus_1 (p)], x[p], y[p],
          x[plus_1 (p)], y[plus_1 (p)]);
      IF xprod > 0 THEN
        convex_clockwise := FALSE;
      IF xprod < 0 THEN
        convex_counter_clockwise := FALSE;
      p := p + 1
    END;
```

```
    convex_polygon_test := convex_counter_clockwise OR convex_clockwise
  END;  { convex_polygon_test }

  {-------------------------------------------------------------------------
  This routine checks if the polygon is valid -- i.e. has sides that do not
  cross or even touch (except for adjacent sides). }

  FUNCTION valid_polygon_test : BOOLEAN;

  VAR
    valid : BOOLEAN;
    p : point_range;
    exclude : point_set;  { set of sides of the polygon to exclude }

  BEGIN
  { special test for triangles -- points cannot be collinear }
    IF n_points = 3 THEN
      valid := crossproduct (x[1], y[1], x[2], y[2], x[3], y[3]) <> 0

    ELSE BEGIN
{ To test side 1 (from point 1 to point 2), note how three sides (the side
  from points "n_points" to 1, the side from points 1 to 2, and the side
  from points 2 to 3) must be excluded from the call to "uncrossed". }
      exclude := [1, 2];
      valid := uncrossed (x[1], y[1], x[2], y[2], exclude + [n_points]);

      p := 2;              { now test the other sides (except final two) }
      WHILE valid AND (p <= n_points - 2) DO BEGIN
        exclude := exclude + [p+1];  { add to (via union operator) set }
        valid := uncrossed (x[p], y[p], x[p+1], y[p+1], exclude);
        p := p + 1
      END
    END;

    IF NOT valid THEN
      WRITELN ('    Invalid polygon.  Please re-input:')

{ If polygon must be convex, then invoke convex polygon test }
    ELSE IF must_be_convex AND NOT convex_polygon_test THEN BEGIN
      WRITELN ('    Polygon must be convex.  Please re-input:');
      valid := FALSE
    END;

    valid_polygon_test := valid
  END;  { valid_polygon_test }

{-------------------------------------------------------------------------}

BEGIN  { get_points }
```

```
{ First get number of points }
  REPEAT
    WRITE ('Enter number of points = ');
    READLN (n_points);
    IF (n_points < 3) OR (n_points > max_points) THEN
      WRITELN ('     (must be >= 3 and <= ', max_points, ')')
  UNTIL (n_points >= 3) AND (n_points <= max_points);

{ Read in the x and y coordinate pair for each of the 'n_points' points }
  REPEAT
    FOR p := 1 TO n_points DO BEGIN
      WRITE ('     X[', p:2,'], Y[', p:2, ']      = ');
      READLN (x[p], y[p])
    END;
  UNTIL valid_polygon_test;

{ Now find the direction in which the points in the polygon are arranged by
  the sign of the area of the polygon using the crossproduct method }
  polygon_area := 0;
  FOR p := 2 TO n_points - 1 DO
    polygon_area := polygon_area +
        crossproduct (x[1], y[1], x[p], y[p], x[p+1], y[p+1]);

  IF polygon_area < 0 THEN  { if in clockwise order, then reverse order }
    reverse_order;

  polygon_area := 0.5*ABS (polygon_area);
  WRITELN ('Polygon area = ',polygon_area);
  WRITELN;

{ Open output file for saving results }
  ASSIGN (output_file, filename);
  REWRITE (output_file);

  WRITELN (output_file, 'POLYGON AREA  = ',polygon_area:7:6);
  FOR p := 1 TO n_points DO
    WRITELN (output_file, '  X[',p:2,'] Y[',p:2,'] = ',x[p],' ',y[p]);

{ Other initializations }
  operation_count := 0
END;  { get_points }

{-------------------------------------------------------------------------
It is not possible to use the "=" operator to test for equality between two
real numbers.  Instead, two real numbers that are very close -- within some
"epsilon"-- are considered, for all intents and purposes, equal.  An
appropriate value for "epsilon" depends on the precision of the
floating-point number representation. }
```

```
FUNCTION real_equal (a, b : REAL) : BOOLEAN;

CONST
  epsilon = 1E-10;

VAR
  difference, abs_max, abs_b, relative_error : REAL;

BEGIN
  IF a = b THEN                          { the obvious test }
    real_equal := TRUE

  ELSE BEGIN
    difference := ABS (a - b);

    IF difference < epsilon THEN         { close enough }
      real_equal := TRUE

    ELSE BEGIN
      abs_max := ABS (a);
      abs_b := ABS (b);
      IF abs_b > abs_max THEN
        abs_max := abs_b;

      relative_error := difference/abs_max;
      IF relative_error < epsilon THEN  { compare relative error }
        real_equal := TRUE
      ELSE
        real_equal := FALSE
    END
  END
END;  { real_equal }

{-------------------------------------------------------------------------
This system-specific routine is intended to store the current CPU usage for
for the user process, so that it can be compared after the computations in
the program have been run.  On this system, which is single-user, we can
use the current time. }

PROCEDURE start_timer;

VAR
  h, m, s, cs : WORD;

BEGIN
  GetTime (h, m, s, cs);

  initial_time := m + 60*h;
  initial_time := cs + 100*(s + 60*initial_time)
END;  { start_timer }
```

```
{-------------------------------------------------------------------------
This is the complement of "start_timer" in that it takes the current CPU
usage for the user process, and subtracts it from the CPU usage stored by
"start_timer" to compute the elapsed CPU time.  Note that our
implementation here assumes that "start_timer" and "stop_timer" are both
called during the same day. }

PROCEDURE stop_timer;


VAR
  h, m, s, cs : WORD;

BEGIN
  GetTime (h, m, s, cs);
  elapsed_time := m + 60*h;
  elapsed_time := cs + 100*(s + 60*elapsed_time);
  elapsed_time := elapsed_time - initial_time;

  WRITELN (output_file)
END;  { stop_timer }

{-------------------------------------------------------------------------
This routine outputs the result in "distance" with the label in
"result_name" to the user console, and to "output file" which was opened by
"get_points". }

PROCEDURE print_results (result_name : string_type; distance : REAL);

BEGIN
  WRITELN (result_name, distance);
  WRITELN (output_file, result_name, distance)
END;  { print_results }

{-------------------------------------------------------------------------
This routine is the final action of the program, showing the elapsed time
and closing "output_file". }

PROCEDURE terminate (show_operation_count : BOOLEAN);

BEGIN
  WRITELN ('Elapsed time     = ',elapsed_time/100:2:1,' seconds');
  WRITELN (output_file, 'Elapsed time = ',elapsed_time/100:2:1,' seconds');

  IF show_operation_count THEN BEGIN
    WRITELN ('Operation count = ',operation_count);
    WRITELN (output_file, 'Operation count = ',operation_count)
  END;
```

```
  CLOSE (output_file)
END;  { terminate }

{-------------------------------------------------------------------------
This routine reads the parameters "n_iterations" (number of samples) and
"seed" (for the random number generator) that are used by the Monte Carlo
simulation. }

PROCEDURE get_Monte_Carlo_parameters (VAR n_iterations, seed : INTEGER);

BEGIN
  WRITE ('Number of iterations in simulation = ');
  READLN (n_iterations);
  WHILE n_iterations < 1 DO BEGIN
    WRITELN ('     (must be > 0)');
    WRITE ('Number of iterations in simulation = ');
    READLN (n_iterations)
  END;
  operation_count := n_iterations;

  WRITE ('Random number seed                  = ');
  READLN (seed);
  IF seed < 0 THEN seed := -seed;

{ Turbo Pascal only: initialization of random number generator }
  RandSeed := seed;

  WRITELN
END;  { get_Monte_Carlo_parameters }

{-------------------------------------------------------------------------
This procedure returns two random numbers that are uniformly distributed
between 0 and 1 (for the Monte Carlo simulation).  The implementation here
is specific to Turbo Pascal and does not actually modify 'seed'. }

PROCEDURE new_random (VAR seed : INTEGER;  VAR r1, r2 : REAL);

BEGIN
  r1 := Random;  { Turbo Pascal function to get pseudo-random number }
  r2 := Random
END;  { new_random }
```

A.2 Numerical method for the Manhattan metric

A.2.1 Pascal Source Code

```
{-------------------------------------------------------------------------
This program implements the method in Chapter 2 to find the expected
Manhattan distance between two uniformly, identically, and independently
distributed points in a polygon.

Written in Turbo Pascal, Version 4.0.
Runs under PC-DOS/MS-DOS Version 2.0 or later.
Runs on IBM PC or compatible.
Written by Arthur Hsu, 1984 (revised January 1990).
-------------------------------------------------------------------------}

PROGRAM manhattan;

{$I BASICS.PAS}              { Turbo Pascal command to include external file
                               "BASICS.PAS" containing basic polygon routines
                               -- see listing in Appendix A.1.2 }
TYPE
  class_types = (open, close, join, divide, intermediate);
  channel_range = 0..max_points;
  connexion_range = -max_points..max_points;

VAR
  i, d : point_array;
  class : ARRAY[point_range] OF class_types;  { class for each vertex }
  n_channels, n_zones : channel_range;        { number of channels, zones }
  channel : ARRAY[point_range] OF             { parameters per channel }
    RECORD
      connexion : ARRAY[point_range] OF connexion_range;
      low_i, high_i : REAL
    END;
  zone : ARRAY[point_range] OF                { parameters for each zone }
    RECORD
      area, centre, intrazonal_distance : REAL;
      in_channel : point_range
    END;
  average_x, average_y : REAL;                { avg distance in each axis }

{-------------------------------------------------------------------------
This procedure finds the "class" for each point 'p', and sets the array
'class' accordingly.  The default class is 'intermediate', and four special
classes are 'open', 'close', 'join', and 'divide'. }

PROCEDURE determine_classes;

VAR
```

```
  p, high_p, initial_p : point_range;
  continue : BOOLEAN;

  {-------------------------------------------------------------------------
  This routine identifies the class for point 'p' given that the points 'p'
  through 'high_p' are all contiguous points including 'p' that share the
  same 'i' value. }

  PROCEDURE identify_class (p, high_p : point_range);

  VAR
    previous, next : point_range;
    convex_turn : BOOLEAN;

  BEGIN
{ Based on definition for 'p' and 'high_p', 'i[previous]' <> 'i[p]', and }
    previous := minus_1 (p);  {   'i[next]' <> 'i[p]' }
    next := plus_1 (high_p);

{ Separate the classes }
    IF (i[p] - i[previous])*(i[p] - i[next]) <= 0 THEN
      class[p] := intermediate

    ELSE BEGIN                  { previous and next are on same side of p }
      convex_turn := crossproduct (i[previous], d[previous], i[p], d[p],
          i[high_p], d[high_p]) + crossproduct (i[previous], d[previous],
          i[high_p], d[high_p], i[next], d[next]) > 0;

      IF i[p] < i[next] THEN  { p is to the left of previous and next }
        IF convex_turn THEN   { p is a convex vertex }
          class[p] := open
        ELSE                  { p is a concave vertex }
          class[p] := divide
      ELSE                    { p is to the right of previous and next }
        IF convex_turn THEN   { p is a convex vertex }
          class[p] := close
        ELSE                  { p is a concave vertex }
          class[p] := join
    END  { ELSE }
  END;  { identify_class }

{-------------------------------------------------------------------------}

BEGIN  { determine_classes }
{ get initial 'p', the point in the group of points adjacent to and sharing
  the same 'i' value as point 1 that has the lowest relative index }
  initial_p := 1;
  WHILE i[initial_p] = i[minus_1 (initial_p)] DO
    initial_p := minus_1 (initial_p);
  p := initial_p;
```

```
  REPEAT
{ find largest group of points contiguous with 'p' with same 'i' value }
    high_p := p;
    WHILE i[high_p] = i[plus_1 (high_p)] DO
      high_p := plus_1 (high_p);

    identify_class (p, high_p);
{ having assigned class to 'p', set all others in group to 'intermediate' }
    WHILE p <> high_p DO BEGIN
      p := plus_1 (p);
      class[p] := intermediate
    END;

    p := plus_1 (high_p)
  UNTIL p = initial_p
END;  { determine_classes }

{-------------------------------------------------------------------------
Having identified the class for all of the points, this routine divides the
polygon into zones and channels }

PROCEDURE divide_polygon_into_zones_and_channels;

TYPE
  edge_parameters =
    RECORD
      high_index, low_index : point_range;
      i_value, high_d, low_d : REAL
    END;

VAR
  p, p1 : point_range;
  c : channel_range;
  left_edge : ARRAY[point_range] OF edge_parameters;
  join_channel : ARRAY[point_range] OF channel_range;  { this indicates if
      the point is of the 'join' class where there is channel that has
      already been found whose right edge is at that point }

  {-------------------------------------------------------------------------
  This routine starts at the left edge of the channel and sweeps to the
  right until the right edge of the channel, in so doing, dividing the
  channel into zones.  The left edge is specified by the left edge
  parameters and 'low_i'.  The right edge is encountered when a 'close',
  'join', or 'divide' (internal) is encountered. }

  PROCEDURE process_channel_into_zones (c : point_range);

  TYPE
    index_types = (high, low, divider);
```

```
VAR
  old_left : edge_parameters;
  next_low, next_high, index : point_range;
  index_type : index_types;
  alpha, beta, lambda : REAL;

  {-------------------------------------------------------------------------
  For the current left edge, this routine finds the index of the point in
  the channel with next highest 'i' value. }

  PROCEDURE find_next_index (VAR next_high, next_low, next_index :
      point_range;  VAR next_index_type : index_types);

  VAR
    divide_point : point_range;
    next_i : REAL;

  BEGIN
    WITH left_edge[c] DO BEGIN
      next_high := minus_1 (high_index);  { since points in polygon are }
      next_low := plus_1 (low_index);     {   in counterclockwise order }

      IF i[next_high] < i[next_low] THEN BEGIN  { take lower one }
        next_index_type := high;
        next_index := next_high
      END
      ELSE BEGIN
        next_index_type := low;
        next_index := next_low
      END;
      next_i := i[next_index];

{ But must check for divide cases in between the low and high envelopes }
      FOR divide_point := 1 TO n_points DO
        IF class[divide_point] = divide THEN
          IF (i[divide_point] >= i_value) AND (i[divide_point] < next_i)
              THEN                          { if in possible 'i' range }
            IF (crossproduct (i[next_high], d[next_high], i[high_index],
                d[high_index], i[divide_point], d[divide_point]) > 0) AND
                (crossproduct (i[low_index], d[low_index], i[next_low],
                d[next_low], i[divide_point], d[divide_point]) > 0)
                THEN BEGIN                  { if in middle of channel }
              next_index_type := divider;
              next_index := divide_point;
              next_i := i[next_index]
            END
    END
  END;  { find_next_index }
```

```
{-------------------------------------------------------------------
This routine sets up the necessary connexion(s) to the channel(s) at
the right end of the channel: there will be two new channels formed for
each 'divide' class; one new channel for each pair of 'join'-class
channels; and no new channels for a 'close' class. }

PROCEDURE channel_interface;

VAR
  upper, lower : point_range;

BEGIN
  IF class[index] = divide THEN BEGIN
    n_channels := n_channels + 1;       { set up upper channel }
    channel[c] . connexion[n_channels] := n_channels;
    left_edge[n_channels] := left_edge[c];
    WITH left_edge[n_channels] DO BEGIN
      low_index := index;
      low_d := d[index]
    END;


    n_channels := n_channels + 1;       { set up lower channel }
    channel[c] . connexion[n_channels] := n_channels;
    left_edge[n_channels] := left_edge[c];
    WITH left_edge[n_channels] DO BEGIN
      high_index := index;
      high_d := d[index]
    END;
  END

  ELSE IF class[index] = join THEN
    IF join_channel[index] = 0 THEN     { first of a pair of channels }
      join_channel[index] := c
    ELSE BEGIN                          { second of the pair }
{ Distinguish between the upper and lower of the pair of join channels }
      IF left_edge[c] . high_d >
          left_edge[join_channel[index]] . high_d THEN BEGIN
        upper := c;
        lower := join_channel[index]
      END
      ELSE BEGIN
        upper := join_channel[index];
        lower := c
      END;

      n_channels := n_channels + 1;
{ Set up connexions }
      channel[upper] . connexion[n_channels] := n_channels;
      channel[lower] . connexion[n_channels] := n_channels;
```

```
{ Set up new channel }
          WITH left_edge[n_channels] DO BEGIN
            high_index := left_edge[upper] . high_index;
            low_index := left_edge[lower] . low_index;
            i_value := i[index];
            high_d := left_edge[upper] . high_d;
            low_d := left_edge[lower] . low_d
          END
        END  { ELSE }
    END;  {  channel_interface }

  {-------------------------------------------------------------------------}

  BEGIN  { process_channel_into_zones }
    channel[c] . low_i := left_edge[c] . i_value;

{ For each zone, set up parameters }
    REPEAT
      old_left := left_edge[c];
      find_next_index (next_high, next_low, index, index_type);

{ Update new left_edge[c] }
      WITH left_edge[c] DO BEGIN
        i_value := i[index];

{ Find high_d }
        IF index_type = high THEN BEGIN  { zone ends due to high point }
          high_index := next_high;
          high_d := d[next_high]                    { high_d at vertex }
        END
        ELSE IF i[next_high] > old_left . i_value THEN
          high_d := high_d + (d[next_high] - high_d)*
              (i_value - old_left . i_value)/       { interpolate high_d }
              (i[next_high] - old_left . i_value)
        ELSE
          high_d := d[next_high];

{ Similarly, find low_d }
        IF index_type = low THEN BEGIN
          low_index := next_low;
          low_d := d[next_low]
        END
        ELSE IF i[next_low] > old_left . i_value THEN
          low_d := low_d + (i_value - old_left . i_value)*  { interpolate }
              (d[next_low] - low_d)/(i[next_low] - old_left . i_value)
        ELSE
          low_d := d[next_low]
      END;  { WITH left_edge[c] }
```

```
{ Add new zone }
      n_zones := n_zones + 1;
      WITH zone[n_zones] DO BEGIN
        alpha := old_left . high_d - old_left . low_d;        { left edge }
        beta := left_edge[c] . high_d - left_edge[c] . low_d;     { right }
        lambda := left_edge[c] . i_value - old_left . i_value;    { width }
        area := 0.5*lambda*(alpha + beta);  { area of trapezoid }

        IF area = 0 THEN                    { in class area is zero }
          n_zones := n_zones - 1            { remove zone }

        ELSE BEGIN
          in_channel := c;
                                            { from Equation 2.12 }
          centre := old_left . i_value +
              (alpha + 2*beta)*lambda/(3*(alpha + beta));
                                            { adapted from Equation 2.8 }
          intrazonal_distance := (SQR(alpha) + 3*alpha*beta + SQR (beta))*
              lambda*SQR(lambda)/(15*SQR(area))
        END
      END
    UNTIL class[index] <> intermediate;     { until end of channel }

{ Set channel parameters }
    channel[c] . high_i := left_edge[c] . i_value;
    channel_interface                       { interface to other channels }
  END;  { process_channel_into_zones }

{-------------------------------------------------------------------------}

BEGIN  { divide_polygon_into_zones_and_channels }

{ Initializations }
  n_channels := 0;
  n_zones := 0;
  FOR p := 1 TO max_points DO
    FOR p1 := 1 TO max_points DO
      channel[p] . connexion[p1] := 0;  { channels are unconnected }
  FOR p := 1 TO n_points DO
    join_channel[p] := 0;               { channels not joined to others }

{ Start new channels at all 'open' classes, and set left edge parameters }
  FOR p := 1 TO n_points DO
    IF class[p] = open THEN BEGIN
      n_channels := n_channels + 1;
      WITH left_edge[n_channels] DO BEGIN
        high_index := p;
        low_index := p;
        i_value := i[p];
        high_d := d[p];
```

```
        low_d := d[p]
      END
    END;

{ Loop through all channels.  Note that as a side-effect of
  'process_channels_into_zones', n_channels may be incremented }
  c := 0;
  WHILE c < n_channels DO BEGIN
    c := c + 1;
    process_channel_into_zones (c)
  END
END;  { divide_polygon_into_zones_and_channels }

{------------------------------------------------------------------------
This routine, given the zone and channel parameters, actually computes the
expected Manhattan distance in the independent axis. }

FUNCTION compute_distance_in_independent_axis : REAL;

VAR
  distance : REAL;
  interchannel : ARRAY[point_range, point_range] OF
    RECORD
      edge1, edge2, common_path : REAL
    END;

  {------------------------------------------------------------------------
  In order to reduce the computation in the function interzonal_distance,
  this routine not only sets up the rest of the inter-channel connexions,
  but computes the interchannel distances as well in setting up the array
  'interchannel'. }

  PROCEDURE complete_connexions;

  VAR
    c1, c2, destination, int_channel : point_range;
    done : BOOLEAN;
    int_position, next_int_position : REAL;
      { int stands for intermediary channel }

  BEGIN

{ First reflect the upper right "triangle" of matrix into the lower left }
    FOR c1 := 1 TO n_channels - 1 DO
      FOR c2 := c1 + 1 TO n_channels DO
        IF channel[c1] . connexion[c2] <> 0 THEN
          channel[c2] . connexion[c1] := -c1;

{ Then fill the matrix }
    REPEAT
```

```
      done := TRUE;
      FOR c1 := 1 TO n_channels DO
        WITH channel[c1] DO
          FOR c2 := 1 TO n_channels DO
            IF ABS (connexion[c2]) = c2 THEN
              FOR destination := 1 TO n_channels DO
                IF (connexion[destination] = 0) AND
                    (channel[c2] . connexion[destination] <> 0) THEN BEGIN
                  connexion[destination] := connexion[c2];
                  done := FALSE
                END
    UNTIL done;

{ Now set up interchannel -- only upper right triangle need be done }
    FOR c1 := 1 TO n_channels - 1 DO
      FOR c2 := c1 + 1 TO n_channels DO
        WITH interchannel[c1, c2] DO BEGIN
          WITH channel[c1] DO
            IF connexion[c2] > 0 THEN  { 'c2' to right of 'c1' }
              edge1 := high_i
            ELSE
              edge1 := low_i;          { 'c2' to left of 'c1' }
          WITH channel[c2] DO
            IF connexion[c1] > 0 THEN
              edge2 := high_i
            ELSE
              edge2 := low_i;

{ Now find the common_path length by traversing all intermediary channels }
          common_path := 0;
          int_channel := c1;
          int_position := edge1;
          REPEAT
            WITH channel[int_channel] DO BEGIN
              IF connexion[c2] > 0 THEN
                next_int_position := high_i
              ELSE
                next_int_position := low_i;
              common_path := common_path +
                  ABS (next_int_position - int_position);
              int_position := next_int_position;
              int_channel := ABS (channel[int_channel] . connexion[c2])
            END
          UNTIL int_channel = c2
        END  { WITH interchannel[c1, c2] }
  END;  { complete_connexions }

  {-------------------------------------- ----------------------------
  This function returns the expected distance between zone1 and zone2 (in
  the direction of the independent axis }
```

```
  FUNCTION interzonal_distance (zone1, zone2 : point_range) : REAL;

  VAR
    channel1, channel2 : point_range;

  BEGIN
    channel1 := zone[zone1] . in_channel;
    channel2 := zone[zone2] . in_channel;

    IF channel1 = channel2 THEN
      interzonal_distance := ABS (zone[zone1] . centre -
          zone[zone2] . centre)
    ELSE
      WITH interchannel[channel1, channel2] DO
        interzonal_distance := ABS (edge1 - zone[zone1] . centre) +
            common_path + ABS (zone[zone2] . centre - edge2)
  END;  { interzonal_distance }

  {-------------------------------------------------------------------------
  With all parameters set up, this routine actually completes the
  calculation for the expected distance (according to Equation 2.13). }

  PROCEDURE sum_up_distance_components;

  VAR

    z1, z2 : point_range;
    total_area : REAL;

  BEGIN
    distance := 0;
    total_area := 0;

    FOR z1 := 1 TO n_zones DO
      WITH zone[z1] DO BEGIN
        total_area := total_area + area;
        distance := distance + SQR(area)*intrazonal_distance;

        FOR z2 := z1 + 1 TO n_zones DO
          distance := distance +
              2 * area * zone[z2] . area * interzonal_distance (z1, z2)
      END;  { WITH zone[z1] }

    distance := distance/SQR (total_area)
  END;  { sum_up_distance_components }

{-------------------------------------------------------------------------}

BEGIN  { compute_distance_in_independent_axis }
```

```
  complete_connexions;
  sum_up_distance_components;
  compute_distance_in_independent_axis := distance
END;  { compute_distance_in_independent_axis }

{-------------------------------------------------------------------------}

FUNCTION apply_method_for_independent_axis
    (independent, dependent : point_array) : REAL;

BEGIN
  i := independent;
  d := dependent;

  determine_classes;
  divide_polygon_into_zones_and_channels;
  apply_method_for_independent_axis := compute_distance_in_independent_axis
END;  { apply_method_for_independent_axis }

{-------------------------------------------------------------------------}

BEGIN  { MAIN - manhattan }
  get_points (FALSE, 'MANHAT.OUT');
  start_timer;

  average_x := apply_method_for_independent_axis (x, y);
  reverse_order;
  average_y := apply_method_for_independent_axis (y, x);


  stop_timer;
  print_results ('Expected Manhattan Method = ',
      average_x + average_y);                  { according to Equation 2.3 }
  terminate (FALSE)
END.  { MAIN - manhattan }
```

A.2.2 Numerical Examples

To demonstrate the program for the Manhattan metric, we have provided some numerical examples. The computation time required varied from 0.1 seconds for triangles and quadrilaterals, to 0.6 seconds for a 20-sided regular polygon, to 1.0 seconds for an irregular, concave 20-sided polygon.

Example A-1. Unit Square

Our first example is of a unit square. We will also use this example to demonstrate how

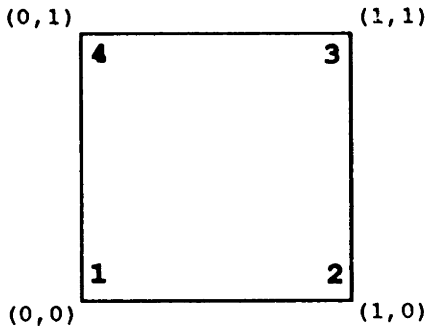

Figure A-1: A unit square

to use the program. To start the program on an MS-DOS machine, the user would type "MANHAT". The user would first enter the number of points in the polygon – in this case, 4 – and then the coordinates for each point defining the polygon. Then the program would calculate the expected Manhattan distance between two random points in the defined polygon. For this example, the user would see on the console:

```
Enter number of points = 4
     X[ 1], Y[ 1]      = 0 0
     X[ 2], Y[ 2]      = 1 0
     X[ 3], Y[ 3]      = 1 1
     X[ 4], Y[ 4]      = 0 1
Polygon area =  1.0000000000E+00

Expected Manhattan Method =  6.6666666667E-01
Elapsed time    = 0.1 seconds
```

There is also an output file created called "MANHAT.OUT" that contains the following text:

```
POLYGON AREA   = 1.000000
  X[ 1] Y[ 1] =  0.0000000000E+00  0.0000000000E+00
  X[ 2] Y[ 2] =  1.0000000000E+00  0.0000000000E+00
  X[ 3] Y[ 3] =  1.0000000000E+00  1.0000000000E+00
  X[ 4] Y[ 4] =  0.0000000000E+00  1.0000000000E+00

Expected Manhattan Method =  6.6666666667E-01
Elapsed time = 0.1 seconds
```

The calculated result of 0.666667 matches the previously known value.

We performed further experiments on the square. If the square was translated elsewhere in the coordinate system – for example, a unit square where the lower-left corner is at (50,100) – the result is still the same. If the lengths of the sides of the square were scaled, the resulting expected distance is scaled proportionately. (For example, for a square with sides of length 100, the expected distance is 66.666667.) If we used more than 4 points to define the same square such as in Figure A-2, the result was still the same.

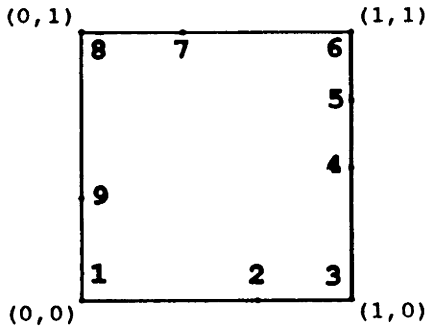

Figure A-2: A nonagon shaped like a square

Example A-2. Rotated Unit Square

Using the Manhattan metric, the expected distance of a polygon is affected if the polygon is rotated. Figure A-3 shows a unit square that is rotated, and some numerical results at different rotation angles are given in Table A-1.

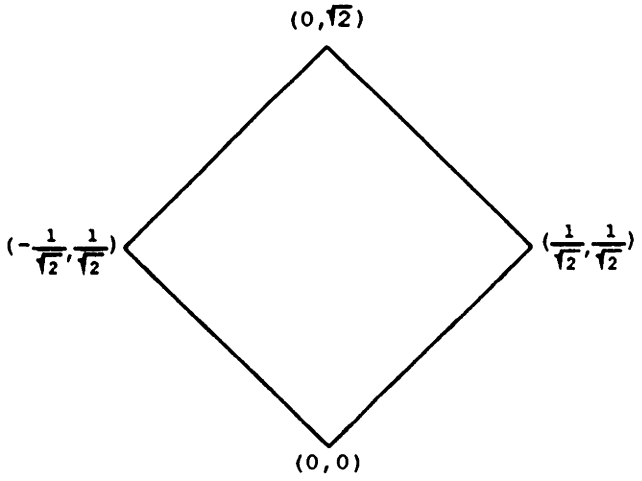

Figure A-3: A unit square rotated by 45°

Rotation angle	Expected Manhattan distance
0°	0.666667
15	0.665829
30	0.662464
45	0.659966

Table A-1: Expected Manhattan distance within a unit square that is rotated

Example A-3. Other regular polygons

We can experiment with other regular polygons of unit area. Because the orientation (rotation) of the polygon affects the expected distance, our examples in Table A-2 all are of regular polygons that have at least one side that is parallel to the x-axis.

It is known that the expected Manhattan distance between two random points in a circle of unit area is $\frac{512}{45\pi^2\sqrt{\pi}} \simeq 0.650403$. [1] It is not surprising that the expected Manhattan distance of a regular polygon approaches this value as the number of sides increases since the shape of a regular polygon approaches a circle as the number of sides is increased.

[1] Samuel Eilon, C.D.T. Watson-Gandy and Nicos Christofides, *Distribution Management: Mathematical modelling and practical analysis* (New York: Hafner Publishing Company, 1971), p. 163.

Number of sides	Expected Manhattan distance
3	0.705543
4	0.666667
5	0.655374
6	0.652637
10	0.650676
15	0.650456
20	0.650420

Table A-2: Expected Manhattan distances for various regular polygons

Example A-4. Linear figure

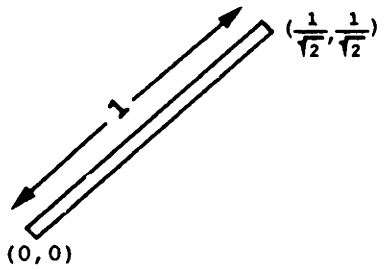

Figure A-4: A very thin rectangle that have been rotated by 45°

Regular polygons are fairly "round" in the sense that they have a shape that is fairly similar to a circle. Consequently, in Example A-2, we found that rotating the figure had a relatively small effect. However, in this example, we will consider a very thin rectangle that approaches a line segment in shape. The expected Manhattan distance is 0.333667 for a rectangle of length 1 and width 0.001. This could have been calculated knowing that the expected Manhattan distance in a rectangle aligned parallel to the x and y axes is $\frac{\text{length}+\text{width}}{3}$.

For an infinitesimally thin rectangle that is parallel to the x-axis, the distance between two points is the difference between their x coordinates which is the same as the straight-line distance. However, if we rotate the rectangle by 45° as in Figure A-4, the expected Manhattan distance is 0.471405 which is approximately $\sqrt{2}$ times greater than that of the "unrotated" rectangle. Even though both rectangles have the identical shape, the difference

is to be expected because the Manhattan distance between the two points is a factor of $(\sin\theta + \cos\theta)$ greater than the straight-line distance, where θ is the angle of rotation. In this case, with $\theta = 45°$, the factor is $\sin 45° + \cos 45° = \sqrt{2}$.

Example A-5. Complex concave polygon

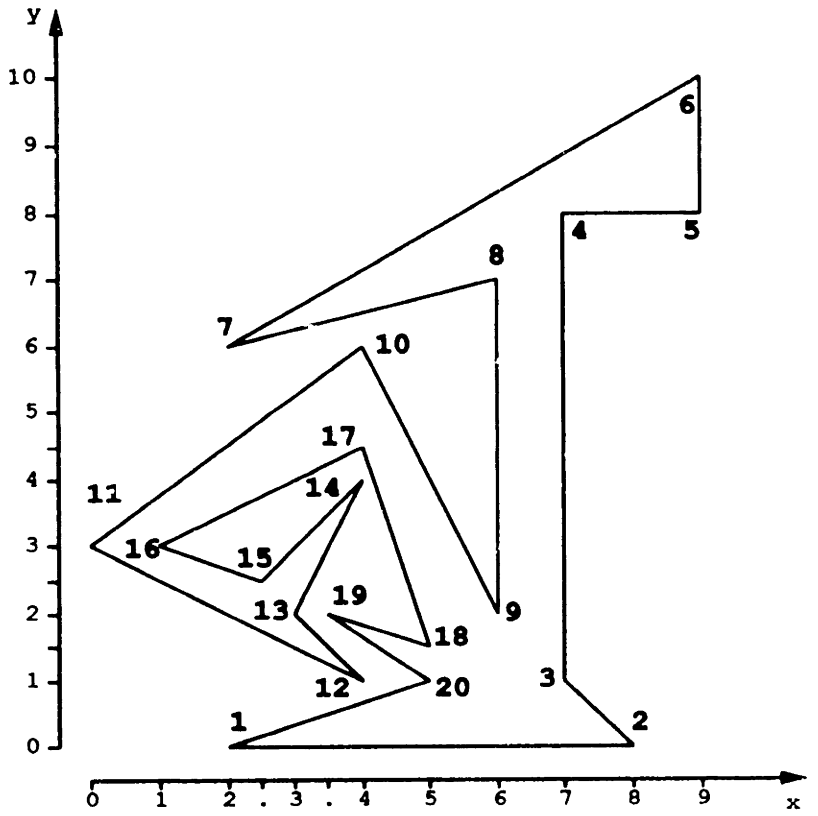

Figure A-5: A complex concave polygon

This example tests a case of a polygon (shown in Figure A-5) with vertices of all of the types of classes (open, divide, join, intermediate, close) that were described in Chapter 2. The expected Manhattan distance is 7.584389.

Example A-6. Another concave polygon

The previous example of a concave polygon produced a result that cannot easily be verified. In this example, we will test the concave polygon shown in Figure A-6 that has been contrived so that we can predict the expected distance. The polygon consists of two

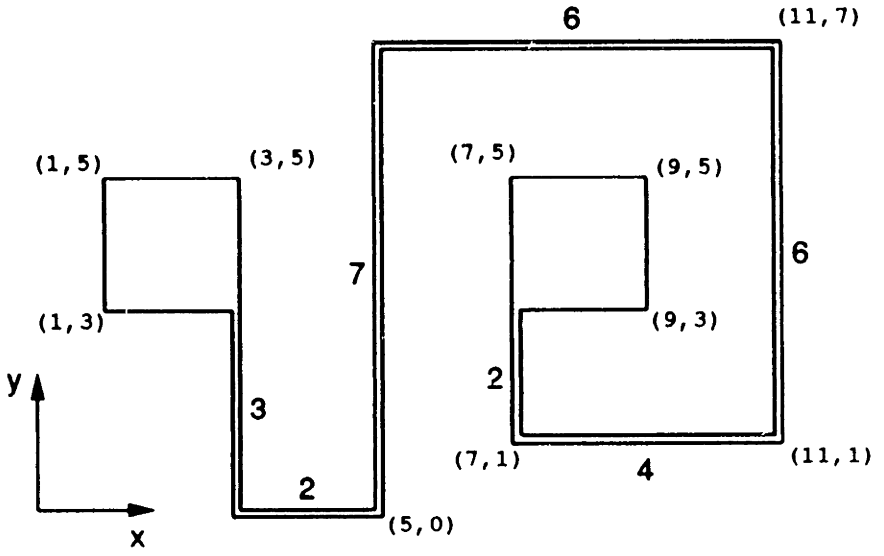

Figure A-6: A concave polygon with two lobes connected by a narrow pathway

"lobes" of equal size connected by a very narrow pathway. If the pathway has an area that is infinitesimally small, then we would anticipate a 50% probability that the expected distance would be for two random points in the same lobe, while the remaining 50% the two random points would be in opposite lobes. Because each lobe is a square with sides of length 2, the "intralobal" expected distance is $\frac{2+2}{3} \simeq 1.333333$. The length of the narrow path is 30, and the expected distance to "connect" a point in a lobe to the end of the narrow pathway is 2 (since the expected Manhattan distance between a random point in a square and any one of the corners is the length of a side of the square). Because the expected "connecting" distance is required at each lobe, the expected distance between two points in opposite lobes is $(2+30+2) = 34$. Thus, we would anticipate that the expected Manhattan distance to be the average of 1.333333 and 34, which is 17.666667. In fact, with the pathway of thickness 0.000001, our program produces an expected distance of 17.666657.

A.3 Numerical method for the Euclidean metric

A.3.1 Pascal Source Code

```
{-------------------------------------------------------------------------
This program implements the method in Chapter 3 to find the expected
Euclidean distance between two uniformly, identically, and independently
distributed points in a convex polygon.

Written in Turbo Pascal, Version 4.0.
Runs under PC-DOS/MS-DOS Version 2.0 or later.
Runs on IBM PC or compatible.
Written by Arthur Hsu, 1984 (revised January 1990).
-------------------------------------------------------------------------}

PROGRAM euclidean;

{$I BASICS.PAS}           { Turbo Pascal command to include external file
                            "BASICS.PAS" containing basic polygon routines
                            -- see listing in Appendix A.1.2 }

  CONST
    max_conjunctions = 190;  { <-- max_points*(max_points - 1) }

  TYPE
    pivot_type = (cis, trans);
    sequence_element =       { each element (scan line) in the sequence }
      RECORD
        point, side : point_range;
        pivot : pivot_type
      END;
    conjunction_range = 0..max_conjunctions;
    conjunction_element =    { defines each conjunction }
      RECORD
        angle : REAL;
        cis_point, trans_point : point_range
      END;

  VAR
    sequence : ARRAY[point_range] OF     { contains entire sequence of }
        sequence_element;                {   scan lines                }
    n_conjunctions : conjunction_range;  { number of conjunctions }
    conjunction :                        { all conjunction angles }
        ARRAY[conjunction_range] OF conjunction_element;
    phi :                                { all angles between points }
        ARRAY[point_range, point_range] OF REAL;
    distance_result : REAL;              { holds expected distance result }

{-------------------------------------------------------------------------
```

```
This procedure takes the indicies of two elements (scan lines) in the
sequence and swaps them. }

PROCEDURE swap_sequence (index1, index2 : point_range);

VAR
  temp : sequence_element;

BEGIN
  temp := sequence[index1];
  sequence[index1] := sequence[index2];
  sequence[index2] := temp
END;  { swap_sequence }

{-------------------------------------------------------------------------
Step 3 in the Configurator Method summary (section 3.2.2).
This procedure generates the sequence of points for the first
configuration. }

PROCEDURE generate_first_configuration;

VAR
  p, pair_range, current_cis_side, current_trans_side : point_range;
  continue_sort : BOOLEAN;
  x_bottom, y_bottom, x_top, y_top : REAL;

BEGIN

{ Set first sequence by sorting the points in increasing y-coordinate }
  FOR p := 1 TO n_points DO     { initialize bubble sort }
    sequence[p] . point := p;
  pair_range := n_points - 1;
  continue_sort := TRUE;

  WHILE continue_sort DO BEGIN  { bubble sort loop }
    p := 1;
    continue_sort := FALSE;     { assume all is sorted }

    WHILE p <= pair_range DO BEGIN
                                { primary comparison test in sort }
      IF y[sequence[p] . point] > y[sequence[p+1] . point] THEN BEGIN
        swap_sequence (p, p+1);
        continue_sort := TRUE
      END
                                { secondary test on x-coordinate }
      ELSE IF (y[sequence[p] . point] = y[sequence[p+1] . point]) AND
          (x[sequence[p] . point] > x[sequence[p+1] . point]) THEN BEGIN
        swap_sequence (p, p+1);
        continue_sort := TRUE   { all is not sorted }
      END;
```

```
      p := p + 1
    END;

    pair_range := pair_range - 1  { no need to check last value again }
  END;

{ Now that the 'sequence' is sorted so that 'point' is arranged in
  increasing y-coordinate, set the other sequence fields 'pivot' (pivot
  type is either cis or trans) and 'side' (the side crossed by the scan
  line on the point) }
  WITH sequence[1] DO BEGIN          { the bottom point }
    x_bottom := x[point];
    y_bottom := y[point];
    current_cis_side := point;
    current_trans_side := point
  END;

  WITH sequence[n_points] DO BEGIN  { the top point }
    x_top := x[point];
    y_top := y[point];
    pivot := trans;                 { by convention, top point is trans }
    side := point
  END;

{ Considering each point in 'sequence', set 'pivot' and 'side' }
  FOR p := 1 TO n_points - 1 DO
    WITH sequence[p] DO
      IF crossproduct (x_bottom, y_bottom, x_top, y_top, x[point],
          y[point]) >= 0 THEN BEGIN  { point is on cis side }
        pivot := cis;
        side := current_trans_side;
        current_cis_side := point
      END
      ELSE BEGIN                      { point is on trans side }
        pivot := trans;
        side := minus_1 (current_cis_side);
        current_trans_side := point
      END
END;  { generate_first configuration }

{-------------------------------------------------------------------------
Steps 1 and 2 in the Configurator Method (section 3.2.2).
This routine sets up the conjunctions -- one for each pair of points. }

PROCEDURE set_up_conjunctions;

VAR
  p1, p2 : point_range;
  continue_sort : BOOLEAN;
```

```
  pair_range, c : conjunction_range;
  temp : conjunction_element;

  {-------------------------------------------------------------------------
  For each pair of points 'p1' and 'p2', the conjunction angle is
  determined, and also if which is the cis-point (the one nearest to the
  scanning direction). }

  PROCEDURE set_conjunction_parameters (c : conjunction_range);

  BEGIN
    WITH conjunction[c] DO BEGIN
      IF y[p1] = y[p2] THEN BEGIN  { horizontal angle }
        angle := 0;

        IF x[p1] < x[p2] THEN      { cis has the lesser x value }
          cis_point := p1
        ELSE
          cis_point := p2
      END

      ELSE BEGIN
        IF x[p1] = x[p2] THEN      { vertical angle }
          angle := 0.5*pi

        ELSE BEGIN                 { all other angles }
          angle := ARCTAN ((y[p1] - y[p2])/(x[p1] - x[p2]));
          IF angle < 0 THEN        { adjust so all angles in [0;pi) range }
            angle := angle + pi
        END;

        IF y[p1] < y[p2] THEN      { cis has the lesser y value }
          cis_point := p1
        ELSE
          cis_point := p2
      END;

      IF cis_point = p1 THEN       { trans point is whatever is not cis }
        trans_point := p2
      ELSE
        trans_point := p1;

      phi[p1, p2] := angle;
      phi[p2, p1] := angle
    END  { WITH conjunction[c] }
  END;  { set_conjunction_parameters }

{-------------------------------------------------------------------------}

BEGIN  { set_up_conjunctions }
```

```
  n_conjunctions := 0;          { number of unique conjunction angles }

  FOR p1 := 1 TO n_points - 1 DO
    FOR p2 := p1 + 1 TO n_points DO BEGIN
      n_conjunctions := n_conjunctions + 1;
      set_conjunction_parameters (n_conjunctions)
    END;

{ Now sort the conjunctions by angle value using bubble sort }
  continue_sort := TRUE;
  pair_range := n_conjunctions - 1;
  WHILE continue_sort DO BEGIN
    continue_sort := FALSE;

    c := 1;
    WHILE c <= pair_range DO BEGIN
      IF conjunction[c] . angle > conjunction[c+1] . angle THEN BEGIN
        temp := conjunction[c+1];
        conjunction[c+1] := conjunction[c];
        conjunction[c] := temp;
        continue_sort := TRUE
      END;
      c := c + 1
    END;

    pair_range := pair_range - 1
  END
END;  { set_up_conjunctions }

{-------------------------------------------------------------------------
This procedure takes a facet as defined between two sequences (scan lines)
in 'sequence1' and 'sequence2' and for the angles between 'theta1' and
'theta2', and returns the factors to be summed into the numerator and
denominator terms of Equation 3.21 as "G" and "H".
This procedure can be internal to 'perform_angle_scan_method'. }

PROCEDURE facet_angle_scan (sequence1, sequence2 : sequence_element;
    theta1, theta2 : REAL;  VAR numerator, denominator : REAL);

VAR
  numerator_term, numerator_term1, numerator_term2, denominator_term,
      denominator_term1, denominator_term2, x_j, y_j : REAL;
  parallel, same_based : BOOLEAN;
  temp : sequence_element;

  {-------------------------------------------------------------------------
  This routine finds the point (x_j;y_j) where the extensions of sides
  'side1' and 'side2' intersect.  In the text of the thesis, the "joiner
  point" is referred to as the intersection point Px. }
```

```
  PROCEDURE get_joiner_point (side1, side2 : point_range;
      VAR x_j, y_j : REAL;  VAR parallel : BOOLEAN);

  VAR
    x1, y1, x2, y2, x3, y3, x4, y4 : REAL;
    y_denominator : REAL;

  BEGIN
    x1 := x[side1];  x2 := x[plus_1 (side1)];
    y1 := y[side1];  y2 := y[plus_1 (side1)];
    x3 := x[side2];  x4 := x[plus_1 (side2)];
    y3 := y[side2];  y4 := y[plus_1 (side2)];

{ Theoretically, the next statement have should read:
    IF crossproduct (x1, y1, x2, y2, x3, y3) =
        crossproduct (x1, y1, x2, y2, x4, y4) THEN
  but because two real numbers cannot be tested safely for equality using
  the equal sign, we have implemented a special 'real_equal' function that
  tests if two real numbers are close enough to be considered equal }

    IF real_equal (crossproduct (x1, y1, x2, y2, x3, y3),
        crossproduct (x1, y1, x2, y2, x4, y4)) THEN
      parallel := TRUE

    ELSE BEGIN
      parallel := FALSE;

      IF real_equal (y1, y2) THEN BEGIN  { horizontal line }
        y_denominator := y3-y4;
        IF real_equal (y_denominator, 0) THEN
          parallel := TRUE
        ELSE BEGIN
          y_j := y1;
          x_j := x3 + (y_j-y3)*(x3-x4)/(y3-y4)
        END
      END

      ELSE BEGIN
        y_denominator := (y1-y2)*(x3-x4) - (x1-x2)*(y3-y4);
        IF real_equal (y_denominator, 0) THEN
          parallel := TRUE

        ELSE BEGIN                          { compute intersection }
          y_j := ((x1-x3)*(y1-y2)*(y3-y4) - y1*(x1-x2)*(y3-y4) +
              y3*(x3-x4)*(y1-y2)) / y_denominator;
          x_j := x1 + (y_j-y1)*(x1-x2)/(y1-y2)
        END
      END
    END  { ELSE }
  END;  { get_joiner_point }
```

```
  {-------------------------------------------------------------------------
  This routine embodies the heart of the method by finding the actual
  terms in the expected distance expression. }

  PROCEDURE scan_procedure (base_sequence, end_sequence : sequence_element;
      opposite_based : BOOLEAN; x_j, y_j, theta1, theta2 : REAL;
      VAR numerator_term, denominator_term : REAL);

  VAR
    a1, a2, d_1_2, phi_1_2, phi_side, phi_hat, theta1_hat, theta2_hat,
        average_theta, numerator_theta1, denominator_theta1,
        numerator_theta2, denominator_theta2 : REAL;

    {-----------------------------------------------------------------------
    This function returns the altitude from (xp;yp) to the line on (x1;y1)
    and (x2;y2). }

    FUNCTION altitude (xp, yp, x1, y1, x2, y2 : REAL) : REAL;

    BEGIN
      altitude := ABS (crossproduct (xp, yp, x1, y1, x2, y2)/
          SQRT (SQR (x1 - x2) + SQR (y1 - y2)))
    END;  { altitude }

    {-----------------------------------------------------------------------
    This routine evaluates the integrals that form part of the numerator
    expressions "G" and the denominator expression "H", as taken from
    Equations 3.35 and 3.38.  't' and 'p' represented the adjusted angles
    theta and phi_hat. }

    PROCEDURE evaluate_integrals (t, p : REAL;
        VAR numerator_expression, denominator_expression : REAL);

    BEGIN

numerator_expression :=
   ( ( 3*COS(6*t+7*p) + 3*COS(6*t+5*p) - 18*COS(4*t+5*p) - 18*COS(4*t+3*p)
         + 45*COS(2*t+3*p) + 45*COS(2*t+p) - 60*COS(p) )
       * LN( (1 + COS(t+p)) / (1 - COS(t+p)))
     - 12*COS(5*t+6*p) - 12*COS(5*t+4*p) + 68*COS(3*t+4*p) + 4*COS(3*t+2*p)
     - 120*COS(t+2*p) + 72*COS(t) )
 / (6*COS(4*t+4*p) - COS(6*t+6*p) - 15*COS(2*t+2*p) + 10 );

denominator_expression :=
    ( SIN(4*t+5*p) + SIN(4*t+3*p) - 4*SIN(2*t+3*p) + 4*SIN(p) )
  / ( 2*COS(4*t+4*p) - 8*COS(2*t+2*p) + 6)

    END;  { evaluate_integrals }
```

```
  {-------------------------------------------------------------------}

  BEGIN  { scan_procedure }
    operation_count := operation_count + 1;

    WITH base_sequence DO BEGIN
      a1 := altitude (x[point], y[point], x[side], y[side],
          x[plus_1 (side)], y[plus_1 (side)]);
      phi_side := phi[side, plus_1 (side)]
    END;

    IF opposite_based THEN BEGIN
      a2 := 0;
      WITH base_sequence DO
        d_1_2 := SQRT (SQR (x[point] - x_j) + SQR (y[point] - y_j));
      WITH end_sequence DO
        phi_1_2 := phi[side, plus_1 (side)]
    END

    ELSE
      WITH end_sequence DO BEGIN
        a2 := altitude (x[point], y[point], x[side], y[side],
            x[plus_1 (side)], y[plus_1 (side)]);
        d_1_2 := SQRT (SQR (x[base_sequence . point] - x[point]) +
            SQR (y[base_sequence . point] - y[point]));
        phi_1_2 := phi[base_sequence . point, point]
      END;

{ adjustments }
    average_theta := 0.5*(theta1 + theta2);
    IF average_theta < phi_1_2 THEN
      phi_1_2 := phi_1_2 - pi;
    IF average_theta < phi_side THEN
      phi_side := phi_side - pi;
{ set parameters for 'evaluate_integrals' }
    theta1_hat := theta1 - phi_1_2;
    theta2_hat := theta2 - phi_1_2;
    phi_hat := phi_1_2 - phi_side;
{ now evaluate them }
    evaluate_integrals (theta1_hat, phi_hat, numerator_theta1,
        denominator_theta1);
    evaluate_integrals (theta2_hat, phi_hat, numerator_theta2,
        denominator_theta2);
    numerator_term := d_1_2 / 60
        * (SQR(SQR(a1)) + a1*a2*SQR(a1) + SQR(a1*a2) + a1*a2*SQR(a2) +
           SQR(SQR(a2)))
        * ABS (numerator_theta1 - numerator_theta2);
    denominator_term := d_1_2
        * (a1*SQR(a1) + a2*SQR(a1) + a1*SQR(a2) + a2*SQR(a2))
        * ABS (denominator_theta1 - denominator_theta2)
```

```
  END;  { scan_procedure }

{-------------------------------------------------------------------------}

BEGIN  { facet_angle_scan }
  IF sequence1 . pivot = sequence2 . pivot THEN
    same_based := TRUE

  ELSE IF sequence1 . side = plus_1 (sequence2 . side) THEN  { bottom }
      BEGIN
    same_based := TRUE;  { bottom point is both cis and trans }

    temp := sequence1;   { swap the two sequences }
    sequence1 := sequence2;
    sequence2 := temp
  END
  ELSE IF sequence2 . side = plus_1 (sequence1 . side) THEN  { top }
    same_based := TRUE   { top point is both cis and trans }

  ELSE BEGIN             { opposite-based unless sides are parallel }
    get_joiner_point (sequence1 . side, sequence2 . side, x_j, y_j,
        parallel);
    same_based := parallel
  END;

  IF same_based THEN
    scan_procedure (sequence1, sequence2, FALSE, 0, 0, theta1, theta2,
        numerator_term, denominator_term)

  ELSE BEGIN  { opposite-based }
    scan_procedure (sequence1, sequence2, TRUE, x_j, y_j, theta1, theta2,
        numerator_term1, denominator_term1);
    scan_procedure (sequence2, sequence1, TRUE, x_j, y_j, theta1, theta2,
        numerator_term2, denominator_term2);
    numerator_term := ABS (numerator_term1 - numerator_term2);
    denominator_term := ABS (denominator_term1 - denominator_term2)
  END;

  numerator := numerator + numerator_term;
  denominator := denominator + denominator_term
END;  { facet_angle_scan }

{-------------------------------------------------------------------------
With first configuration set up and conjunction angles sorted, this routine
does the main part of the method by doing steps 4, 5, and 6 of the
Configurator method. }

PROCEDURE perform_angle_scan_method (VAR distance : REAL);

VAR
```

```
  n_facets, f, cis_location, trans_location : point_range;
  n_configurations, c : conjunction_range;
  numerator, denominator, theta1, theta2 : REAL;

BEGIN
  n_configurations := n_conjunctions;
  n_facets := n_points - 1;
  numerator := 0;
  denominator := 0;
  theta2 := conjunction[n_conjunctions] . angle - pi;

  FOR c := 1 TO n_configurations DO BEGIN
{ Step 4 in Configurator method }
    theta1 := theta2;
    theta2 := conjunction[c] . angle;


    IF theta2 > theta1 THEN   { do it unless conjunctions are parallel }
      FOR f := 1 TO n_facets DO
        facet_angle_scan (sequence[f], sequence[f+1], theta1, theta2,
            numerator, denominator);

{ Step 5 in Configurator method - set up next configuration }
    cis_location := 1;
    WHILE sequence[cis_location] . point <> conjunction[c] . cis_point DO
      cis_location := cis_location + 1;
    trans_location := 1;
    WHILE sequence[trans_location] . point <> conjunction[c] . trans_point
        DO
      trans_location := trans_location + 1;

    swap_sequence (cis_location, trans_location);  { swap sub-step }

    WITH sequence[cis_location] DO                 { increment sub-step }
      side := plus_1 (side);
    WITH sequence[trans_location] DO
      side := plus_1 (side);
                                                   { ensure top/bottom pivot }
    IF (cis_location = 1) OR (trans_location = 1) THEN
      WITH sequence[1] DO BEGIN
        pivot := cis;
        side := plus_1 (side)
      END;
    IF (cis_location = n_points) OR (trans_location = n_points) THEN
      WITH sequence[n_points] DO BEGIN
        pivot := trans;
        side := plus_1 (side)
      END
  END;  { FOR c }
```

```
{ Step 6 in Configurator method - distance calculation }
  distance := numerator/denominator
END;  { perform_angle_scan_method }

{------------------------------------------------------------------------}

BEGIN  { MAIN - euclidean }
  get_points (TRUE, 'EUCLID.OUT');
  start_timer;

  generate_first_configuration;
  set_up_conjunctions;
  perform_angle_scan_method (distance_result);

  stop_timer;
  print_results ('Expected Euclidean distance = ', distance_result);
  terminate (TRUE)
END.  { MAIN - euclidean }
```

A.3.2 Numerical Examples

Example A-7. Unit square

This example is the same as Example A-1 in section A.2.3 except for the distance metric. The user interface is the same, and the output is very similar to the method for the Manhattan metric.

The result that we obtained, 0.521405, matches what was previously calculated. Unlike the Manhattan metric, rotating the square did not affect the expected distance.

Example A-8. Other regular polygons

The program for the Euclidean metric was tested with other regular polygons. As in the

Number of sides	Expected Euclidean distance	Computation time (seconds)	Calls to "scan_procedure"
3	0.554363	1.7	6
4	0.521405	3.6	15
5	0.514733	15.2	50
6	0.512614	14.3	49
7	0.511762	51.5	167
8	0.511364	41.7	153
10	0.511041	132.3	437
15	0.510867	580.8	1868
16	0.510858	504.0	1737
20	0.510839	1022.7	3416

Table A-3: Expected Euclidean distances for various regular polygons

Manhattan metric examples, we see in Table A-3 that as the number of sides increases, the expected distance approaches that of a circle of unit area, which is $\frac{128}{45\pi\sqrt{\pi}} \simeq 0.510826$. [2] However, the computational effort seems to be substantial–a 20-sided polygon requires 17 minutes of computation (on a PC clone). The computation time is closely linked with the number of calls to the routine "scan_procedure", and we can observe that the number of calls does indeed follow the $\mathcal{O}(n^3)$ complexity of the algorithm. Each call to "scan_procedure"

[2] Samuel Eilon, C.D.T. Watson-Gandy and Nicos Christofides, *Distribution Management: Mathematical modelling and practical analysis* (New York: Hafner Publishing Company, 1971), p. 151.

involves numerous trigonometric and floating-point operations which the test machine is relatively slow to calculate.

There appears to be an anomaly in the test results regarding the computational effort as a function of the number of sides in the regular polygon–it takes more time to compute the expected distance for a heptagon than a hexagon, and more time for a 15-sided polygon than a 16-sided polygon. This is explained by the fact that polygons with an even-number of sides will have many conjunction angles that are eliminated due to parallel sides and parallel conjunctions. Furthermore, a facet defined between parallel sides is considered by the procedure "facet_angle_scan" to be same-based, thus eliminating the need to compute the intersection point P_X, and requiring only one call to "scan_procedure".

Example A-9. Different rectangle shapes

We will consider a rectangle with a unit length, and experiment by varying the width y, as shown in Figure A-7. We see in Table A-4 that as we reduce y, the expected Euclidean

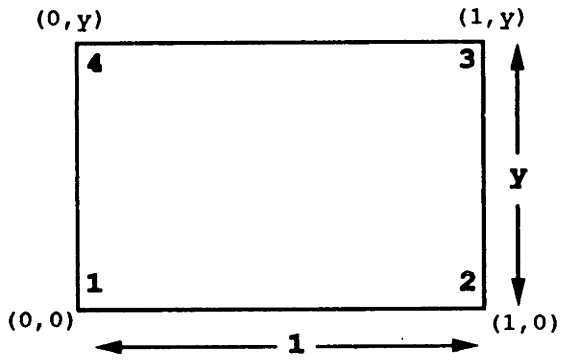

Figure A-7: Rectangle in Example A-9

distance rapidly approaches the limit of 0.333333. Unfortunately, we could not reduce y further than 0.1 without causing round-off errors to become apparent. (We know that the inaccuracies for small values of y in this example are due to round-off as opposed to some methodological error because we have been able to compare results with the same program implemented using floating-point numbers of greater precision. In the same tests, the results from 1.0 to 0.1 were identical (to 8 or more significant figures), but below 0.1, our

y	Expected Euclidean distance
1.0	0.521405
0.5	0.402386
0.2	0.349759
0.1	0.338531

Table A-4: Expected Euclidean distances for different rectangles

PC implementation produced some values that diverged from the anticipated pattern, while the implementation with more precise numbers produced accurate results for considerably smaller values of y.)

A.3.3 Conclusions

Our experiments with the program for finding the expected Euclidean distance show that a machine with powerful floating-point capabilities is highly desirable. In particular, the hardware that we used in our experiments was slow and did not have enough precision. Our previous implementation of this method (using VAX Pascal running on VAX/VMS) with 32-bit floating point (real) numbers provided extremely inaccurate results and we resorted to double precision numbers (64-bits). Our examples here were performed using Turbo Pascal's 48-bit floating-point numbers (with a 39-bit mantissa) which provided fairly accurate results as long as there were no angles between lines that were almost parallel (producing acute angles). However, this problematic situation was encountered often enough to suggest porting the program over to a more suitable machine if detailed experiments are required.

As for expected Euclidean distances in convex polygons, it seems that this quantity is bounded at the upper end by 0.510826 times the maximum dimension of the polygon and at the lower end by 0.333333 times the maximum dimension, where we define the "maximum dimension" of a convex polygon to be the longest possible length between two points in the polygon. The value for the upper bound is taken from the circle, which is the "most convex" shape, while the value for the lower bound is taken from a linear figure as in Example A-9, which is the "least convex" shape that is still convex.

A.4 Monte Carlo simulation

For the purposes of generating approximate values for the expected Manhattan and Euclidean distance between two random points, we have implemented a Monte Carlo simulation. The Monte Carlo analysis is admittedly not complete, but serves as a basis for verifying the results from the analytic methods.

In each run of the Monte Carlo program, expected distances are computed for both the Euclidean and Manhattan distance metrics.

A.4.1 Pascal Source Code

```
{-------------------------------------------------------------------------
This program implements a Monte Carlo simulation for finding the expected
Manhattan and Euclidean distances between two uniformly, identically, and
independently distributed points in a polygon.

Written in Turbo Pascal, Version 4.0.
Runs under PC-DOS/MS-DOS Version 2.0 or later.
Runs on IBM PC or compatible.
Written by Arthur Hsu, 1984 (revised January 1990).
-------------------------------------------------------------------------}

PROGRAM monte_carlo;

{$I BASICS.PAS}           { Turbo Pascal command to include external file
                            "BASICS.PAS" containing basic polygon routines
                            -- see listing in Appendix A.1.2 }

CONST
  max_real : REAL = 1.7E37;

TYPE
  distances =
    RECORD
      Euclidean, Manhattan : REAL
    END;

VAR
  n_iterations,                       { number of iterations (samples) }
  initial_seed : INTEGER;             { for random number generator }
  polygon_is_convex : BOOLEAN;
  concave_vertex :                    { iff vertex is concave }
    ARRAY[point_range] OF BOOLEAN;
  best_path :                         { in concave polygons, the length of }
    ARRAY[point_range, point_range] {   the best path between every pair }
```

```
    OF distances;                    {   of concave vertices              }
  triangle : ARRAY[point_range] OF  { triangles that make up the polygon }
    RECORD
      base_x, base_y, alpha_x, alpha_y, beta_x, beta_y, range : REAL
    END;
  vertex : ARRAY[point_range] OF    { the vertices of each triangle }
    RECORD
      v1, v2, v3 : point_range
    END;

  expected_distance, standard_deviation : distances;

{-------------------------------------------------------------------------
This utility routine adds distance components (both Euclidean and
Manhattan).}

PROCEDURE add_distances (VAR distance1 : distances; distance2 : distances);

BEGIN
  WITH distance1 DO BEGIN
    Euclidean := Euclidean + distance2 . Euclidean;
    Manhattan := Manhattan + distance2 . Manhattan
  END
END;  { add_distances }

{-------------------------------------------------------------------------
This function returns true iff 'point1' and 'point2' cannot be connected by
a line segment that is within the polygon by checking if the angle between
'point1' and 'point2' falls between the angle from 'point1' and the two
adjacent points.  It is assumed that the points defining the polygon are
arranged in counter-clockwise order. }

FUNCTION backdoor (point1, point2 : point_range) : BOOLEAN;

VAR
  to_previous, to_following, to_point2 : REAL;

  {-----------------------------------------------------------------------
  For reasons of efficiency and simplicity, this function will return an
  approximate pseudo-angle that has the necessary property that if an angle
  is greater than a second angle, then the pseudo-angle of the first angle
  is greater than the pseudo-angle of the second angle.  (The pseudo-angle
  is a strictly monotonically increasing function of angle.) }

  FUNCTION pseudo_angle (delta_x, delta_y : REAL) : REAL;

  VAR
    angle : REAL;

  BEGIN
```

```
{ Find quantity within quadrant }
    IF delta_x*delta_y <> 0 THEN
      angle := 90*ABS(delta_y)/(ABS(delta_x) + ABS(delta_y))
    ELSE IF delta_y <> 0 THEN
      angle := 90
    ELSE
      angle := 0;

{ Adjust depending on quadrant }
    IF (delta_x <= 0) AND (delta_y > 0) THEN
      pseudo_angle := 180 - angle
    ELSE IF (delta_x < 0) AND (delta_y <= 0) THEN
      pseudo_angle := angle + 180
    ELSE IF (delta_x >= 0) AND (delta_y < 0) THEN
      pseudo_angle := 360 - angle
    ELSE
      pseudo_angle := angle
  END;  { pseudo_angle }

{------------------------------------------------------------------------}

BEGIN  { backdoor }
  to_previous := pseudo_angle (x[minus_1 (point1)] - x[point1],
      y[minus_1 (point1)] - y[point1]);
  to_following := pseudo_angle (x[plus_1 (point1)] - x[point1],
      y[plus_1 (point1)] - y[point1]);
  to_point2 := pseudo_angle (x[point2] - x[point1], y[point2] - y[point1]);

{ Adjust angles so that it can be compared to 'to_previous' }
  IF to_following > to_previous THEN
    to_following := to_following - 360;
  IF to_point2 > to_previous THEN
    to_point2 := to_point2 - 360;

  backdoor := to_following > to_point2
END;  { backdoor }

{------------------------------------------------------------------------
This procedure sets up the boolean array 'concave_vertex', and then finds
the shortest path between all pairs of concave vertices.  This is done by
first finding the pairs of concave vertices that directly connect (can be
connected by a line segment that does not cross the perimeter of the
polygon), and then for the others, by calling the recursive routine
'find_path' that finds the shortest path among vertices that do not
directly connect by minimizing the set of all possible connecting paths. }

PROCEDURE set_up_best_path_matrix;

TYPE
  connexion_types = (direct, indirect, not_done);
```

```
  node_array = ARRAY[point_range] OF point_range;
  node_stack_type = ARRAY[0..max_points] OF point_range;

VAR
  p1, p2, n_nodes, gap, seq, first, last : point_range;
  connexion : ARRAY[point_range, point_range] OF connexion_types;
  sequence : node_array;

  {-------------------------------------------------------------------------
  If 'point1' and 'point2' are directly connected (as determined by
  'uncrossed') then set up 'connexion' and 'best_path'. }

  PROCEDURE direct_path (point1, point2 : point_range);

  BEGIN
    connexion[point1, point2] := direct;

    IF point1 = point2 THEN
      WITH best_path[point1, point2] DO BEGIN
        Euclidean := 0;
        Manhattan := 0
      END

    ELSE BEGIN
      connexion[point2, point1] := direct;
      WITH best_path[point1, point2] DO BEGIN
        Euclidean := SQRT (SQR(x[point1] - x[point2]) +
            SQR(y[point1] - y[point2]));
        Manhattan := ABS (x[point1] - x[point2]) +
            ABS (y[point1] - y[point2])
      END;
      best_path[point2, point1] := best_path[point1, point2]
    END
  END;  { direct_path }

  {-------------------------------------------------------------------------
  This procedure passes back the sequence of nodes in the shortest path
  between the given origin and destination (concave) nodes and also the
  number of nodes. }

  PROCEDURE find_path (origin, destination : point_range;
      VAR sequence : node_array;  VAR n_nodes : point_range);

  VAR
    distance : REAL;
    reverse_sequence : node_stack_type;
    s : point_range;

    {-----------------------------------------------------------------------
    This recursive routine is the key to the search between the shortest
```

```
    path between two nodes.  By enumerating all the possible next nodes
    to "visit" that have not yet been visited, the routine searches down
    subsequent subpaths until a node is found for which an optimal path
    is known to the destination (if any).  The routine that chooses the
    shortest path by finding the one with the minimum Euclidean distance. }

    PROCEDURE path_finder (origin, destination : point_range;
        visited : point_set;
        VAR distance : REAL;  VAR node_stack : node_stack_type);

    VAR
      p : point_range;
      s : 0..max_points;
      children, new_visited : point_set;
      new_distance : REAL;
      new_node_stack : node_stack_type;

    BEGIN
      IF connexion[origin, destination] <> not_done THEN BEGIN
        distance := best_path[origin, destination] . Euclidean;
        node_stack[0] := 2;  { number of nodes in stack }
        node_stack[1] := destination;
        node_stack[2] := origin
      END

      ELSE BEGIN
        children := [];
        FOR p := 1 TO n_points DO
          IF concave_vertex[p] AND (connexion[origin, p] = direct) AND NOT
              (p IN visited) THEN
            children := children + [p];
        new_visited := visited + children;

        distance := max_real;
        FOR p := 1 TO n_points DO
          IF p IN children THEN BEGIN
            path_finder (p, destination, new_visited, new_distance,
                new_node_stack);
            new_distance := new_distance +
                best_path[origin, p] . Euclidean;

            IF new_distance < distance THEN BEGIN
              distance := new_distance;
{ Copy 'new_node_stack' into 'node_stack' }
              FOR s := 0 TO new_node_stack[0] DO
                node_stack[s] := new_node_stack[s]
            END
          END;  { IF p }

        IF distance < max_real THEN BEGIN  { push origin onto stack }
```

```
          node_stack[0] := node_stack[0] + 1;
          node_stack[node_stack[0]] := origin
        END
      END  { ELSE }
    END;  { path_finder }

  {-------------------------------------------------------------------------}

  BEGIN  { find_path }
    path_finder (origin, destination, [origin], distance,
        reverse_sequence);

    n_nodes := reverse_sequence[0];
    FOR s := 1 TO n_nodes DO
      sequence[s] := reverse_sequence[n_nodes + 1 - s]
  END;  { find_path }

{-----------------------------------------------------------------------------}

BEGIN  { set_up_best_path_matrix }
  FOR p1 := 1 TO n_points DO  { set up 'concave_vertex' array }
    concave_vertex[p1] := crossproduct (x[minus_1 (p1)], y[minus_1 (p1)],
        x[p1], y[p1], x[plus_1 (p1)], y[plus_1 (p1)]) < 0;

  polygon_is_convex := TRUE;  { assumption until concave vertex is seen }
  FOR p1 := 1 TO n_points DO  { set up direct connect subset of paths }
    IF concave_vertex[p1] THEN BEGIN
      polygon_is_convex := FALSE;
      direct_path (p1, p1);

      FOR p2 := p1 + 1 TO n_points DO
        IF concave_vertex[p2] THEN
          IF uncrossed (x[p1], y[p1], x[p2], y[p2], [minus_1 (p1), p1,
              minus_1 (p2), p2]) AND NOT backdoor (p1, p2) THEN
            direct_path (p1, p2)
          ELSE BEGIN
            connexion[p1, p2] := not_done;
            connexion[p2, p1] := not_done
          END
    END;

{ Now fill 'best_path' matrix when it is not yet connected }
  FOR p1 := 1 TO n_points - 1 DO
    IF concave_vertex[p1] THEN
      FOR p2 := p1 + 1 TO n_points DO
        IF concave_vertex[p2] THEN
          IF connexion[p1, p2] = not_done THEN BEGIN
{ Find shortest path from 'p1' and 'p2' including all points passed }
            find_path (p1, p2, sequence, n_nodes);
```

```
{ Also fill 'best_path' for subpaths 's1' to 's2' within the sequence }
            FOR gap := 2 TO n_nodes - 1 DO
              FOR seq := 1 TO n_nodes - gap DO BEGIN
                first := sequence[seq];
                last := sequence[seq + gap];
                connexion[first, last] := indirect;
                connexion[last, first] := indirect;
                best_path[first, last] := best_path[first,
                    sequence[seq + 1]];
                add_distances (best_path[first, last],
                    best_path[sequence[seq + 1], last]);
                best_path[last, first] := best_path[first, last]
              END
          END  { IF connexion }
END;  { set_up_best_path_matrix }

{-------------------------------------------------------------------------
This procedure divides the n-sided polygon into n-2 triangles. }

PROCEDURE set_up_triangles;

VAR
  p, lead, middle, lag, t : point_range;
  hull : ARRAY[point_range] OF BOOLEAN;

  {-----------------------------------------------------------------------
  Iff the angle between the line from 'p1' to 'p2' and the line from 'p2'
  and 'p3' is convex (assuming the points in the polygon are defined in
  counter-clockwise order). }

  FUNCTION convex (p1, p2, p3 : point_range) : BOOLEAN;

  BEGIN
    convex := crossproduct (x[p1], y[p1], x[p2], y[p2], x[p3], y[p3]) >= 0
  END;  { convex }

{-------------------------------------------------------------------------}

BEGIN  { set_up_triangles }
  FOR p := 1 TO n_points DO  { initially, all points are on hull }
    hull[p] := TRUE;

  lag := 1;                  { arbitrary starting group of points }
  middle := 2;
  lead := 3;

  FOR t := 1 TO n_points - 2 DO BEGIN  { for each triangle }
                          { find a "middle" that can be removed from hull }
    WHILE NOT convex (lag, middle, lead) OR NOT uncrossed (x[lag], y[lag],
        x[lead], y[lead], [minus_1 (lag), lag, minus_1 (lead), lead]) OR
```

```
      backdoor (lag, lead) DO BEGIN
    lag := middle;          { shift around hull }
    middle := lead;
    REPEAT                  { get new lead }
      lead := plus_1 (lead)
    UNTIL hull[lead]
  END;

  hull[middle] := FALSE;    { remove "middle" from hull }
  WITH vertex[t] DO BEGIN   { save triangle points }
    v1 := lag;
    v2 := middle;
    v3 := lead
  END;

  middle := lead;        { with point removed from hull, find next group }
  REPEAT
    lead := plus_1 (lead)
  UNTIL hull[lead]
 END  { FOR t }
END;  { set_up_triangles }

{-------------------------------------------------------------------------
This procedure takes the triangles stored in 'vertex' and computes the
triangle parameters to be put into 'triangle'. }

PROCEDURE parameterize_triangles;

VAR
  t : point_range;
  total_area : REAL;
  area : ARRAY[point_range] OF REAL;

BEGIN
                                    { find relative areas of triangles }
  total_area := 0;
  FOR t := 1 TO n_points - 2 DO BEGIN
    WITH vertex[t] DO
      area[t] :=
          ABS (crossproduct (x[v1], y[v1], x[v2], y[v2], x[v3], y[v3]));
    total_area := total_area + area[t];
    triangle[t] . range := total_area
  END;  { FOR t }
                                    { now set up triangle parameters }
  FOR t := 1 to n_points - 2 DO
    WITH triangle[t] DO BEGIN
      WITH vertex[t] DO BEGIN
        base_x := x[v1];
        base_y := y[v1];
        alpha_x := x[v2] - x[v1];
```

```
        alpha_y := y[v2] - y[v1];
        beta_x := x[v3] - x[v2];
        beta_y := y[v3] - y[v2]
      END;

      range := range / total_area
    END
END;  { parameterize_triangles }

{-------------------------------------------------------------------------
This routine actually performs the Monte Carlo method -- first determines
the triangle in which the random point is located (this takes one random
number), and then the location of the point within the triangle using two
other random numbers.  Each pair of random points thus takes 6 random
numbers, and the distances for each metric are computed between the random
points, summed up and averaged for the expected distances. }

PROCEDURE simulate;

VAR
  path, path_square, total_distance, total_square_distance : distances;
  seed, iteration : INTEGER;
  triangle1, triangle2 : point_range;
  r_a, r_b, x1, y1, x2, y2 : REAL;

  {-------------------------------------------------------------------------
  This function takes a uniformly distributed random number 'r' and returns
  the triangle corresponding the range in which 'r' falls.  Recall that the
  ranges were determined in 'parameterize_triangles' and are proportional
  to the area of the triangles.  Note that there are more efficient means
  of finding the triangles -- e.g. binary search, ordering so the larger
  triangles are tested first, etc. --  but the following method is simplest
  to implement. }

  FUNCTION find_triangle (r : REAL) : point_range;

  VAR
    t : point_range;

  BEGIN
    t := 1;
    WHILE triangle[t] . range <= r DO
      t := t + 1;
    find_triangle := t
  END;  { find_triangle }

  {-------------------------------------------------------------------------
  If it is known that a straight-line path between the randomly chosen
  points (x1;y1) and (x2;y2) cannot be made without travelling outside the
  polygon, then this routine finds the shortest path within the polygon.
```

```
This is done by generating the sets of concave vertices ('direct_set1'
and 'direct_set2') that can be directly connected to each of the two
points.  Given the best path between any pair of concave vertices
(pre-calculated in the variable 'best_path'), the shortest path is
determined by finding the minimum path length of the paths from 'point1'
to every point in 'direct_set1', to every point in 'direct_set1', to
'point2'.  Note that the Manhattan path follows easily, since the
shortest Euclidean path is also a shortest Manhattan path. }

PROCEDURE get_shortest_path (VAR shortest_path : distances);

VAR
  p1, p2, min1, min2 : point_range;
  min_path, test_path : REAL;
  direct_set1, direct_set2 : point_set;

  {-------------------------------------------------------------------------
  This procedure generates the set of concave vertices that can be
  directly connected to (x_test;y_test) assuming that (x_test;y_test) is
  within the polygon. }

  PROCEDURE generate_direct_set (x_test, y_test : REAL;
      VAR direct_set : point_set);

  VAR
    p : point_range;

  BEGIN
    direct_set := [];
    FOR p := 1 TO n_points DO
      IF concave_vertex[p] THEN
        IF uncrossed (x_test, y_test, x[p], y[p], [minus_1 (p), p]) THEN
          direct_set := direct_set + [p]
  END;  { generate_direct_set }

{-------------------------------------------------------------------------}

BEGIN  { get_shortest_path }
  generate_direct_set (x1, y1, direct_set1);
  generate_direct_set (x2, y2, direct_set2);

  min_path := max_real;
  FOR p1 := 1 TO n_points DO
    IF p1 IN direct_set1 THEN
      FOR p2 := 1 TO n_points DO
        IF p2 IN direct_set2 THEN BEGIN
{ Test all combinations of points between the two direct-sets }
          test_path := SQRT (SQR(x1 - x[p1]) + SQR(y1 - y[p1])) +
              best_path[p1, p2] . Euclidean +
              SQRT (SQR(x[p2] - x2) + SQR(y[p2] - y2));
```

```
            IF test_path < min_path THEN BEGIN
              min_path := test_path;
              min1 := p1;
              min2 := p2
            END
          END;  { IF p2 }

    shortest_path . Euclidean := min_path;
    shortest_path . Manhattan := ABS(x1 - x[min1]) + ABS(y1 - y[min1]) +
        best_path[min1, min2] . Manhattan +
        ABS(x[min2] - x2) + ABS(y[min2] - y2)
  END;  { get_shortest_path }

{-------------------------------------------------------------------------}

BEGIN  { simulate }
  seed := initial_seed;
  total_distance . Euclidean := 0;
  total_distance . Manhattan := 0;
  total_square_distance := total_distance;

  FOR iteration := 1 TO n_iterations DO BEGIN
    new_random (seed, r_a, r_b);
    triangle1 := find_triangle (r_a);  { first pair of random numbers to  }
    triangle2 := find_triangle (r_b);  {   find triangles of the 2 points }

    new_random (seed, r_a, r_b);       { use 2nd pair of random numbers }
    IF r_a < r_b THEN BEGIN            {   for location of first point  }
      r_a := 1 - r_a;
      r_b := 1 - r_b
    END;
    WITH triangle[triangle1] DO BEGIN
      x1 := base_x + r_a*alpha_x + r_b*beta_x;
      y1 := base_y + r_a*alpha_y + r_b*beta_y
    END;

    new_random (seed, r_a, r_b);       { use 3rd pair of random numbers }
    IF r_a < r_b THEN BEGIN            {   for location of second point }
      r_a := 1 - r_a;
      r_b := 1 - r_b
    END;
    WITH triangle[triangle2] DO BEGIN
      x2 := base_x + r_a*alpha_x + r_b*beta_x;
      y2 := base_y + r_a*alpha_y + r_b*beta_y
    END;

    IF polygon_is_convex THEN BEGIN   { if convex, then path is direct }
      path . Euclidean := SQRT (SQR(x1 - x2) + SQR(y1 - y2));
      path . Manhattan := ABS(x1 - x2) + ABS(y1 - y2)
    END
```

```
                                        { else, first try a direct path }
    ELSE IF uncrossed (x1, y1, x2, y2, []) THEN BEGIN
      path . Euclidean := SQRT (SQR(x1 - x2) + SQR(y1 - y2));
      path . Manhattan := ABS(x1 - x2) + ABS(y1 - y2)
    END
    ELSE                                { else find the shortest path }
      get_shortest_path (path);
                                        { accumulate statistics }
    add_distances (total_distance, path);
    WITH path_square DO BEGIN           { second order statistics }
      Euclidean := SQR(path . Euclidean);
      Manhattan := SQR(path . Manhattan)
    END;
    add_distances (total_square_distance, path_square)
  END;  { FOR iteration }
                                        { final massaging }
  WITH expected_distance DO BEGIN
    Euclidean := total_distance . Euclidean/n_iterations;
    Manhattan := total_distance . Manhattan/n_iterations
  END;
  WITH standard_deviation DO BEGIN
    Euclidean := SQRT (total_square_distance . Euclidean/n_iterations -
        SQR(expected_distance . Euclidean));
    Manhattan := SQRT (total_square_distance . Manhattan/n_iterations -
        SQR(expected_distance . Manhattan))
  END
END;  { simulate }

{------------------------------------------------------------------------}

BEGIN  { MAIN - monte_carlo_simulation }
  get_points (FALSE, 'MCARLO.OUT');
  get_Monte_Carlo_parameters (n_iterations, initial_seed);
  start_timer;

  set_up_best_path_matrix;   { startup overhead }
  set_up_triangles;
  parameterize_triangles;

  simulate;                  { the simulation itself }

  stop_timer;
  WITH expected_distance DO BEGIN
    print_results ('Average Euclidean distance = ', Euclidean);
    print_results ('Average Manhattan distance = ', Manhattan)
  END;
  WITH standard_deviation DO BEGIN
    print_results ('Euclidean standard deviation = ', Euclidean);
    print_results ('Manhattan standard deviation = ', Manhattan)
  END;
```

```
  terminate (TRUE)
END.  { MAIN - monte_carlo_simulation }
```

A.4.2 Numerical Examples

We used Monte Carlo test examples to verify results from our numerical methods. Of course, Monte Carlo simulations produce results that are approximate, and cannot be used to either prove or disprove the validity of an exact method. In all examples, the same random number seed was used (17173).

Example A-10. Unit Square

From our previous tests on a unit square in Examples A-1 and A-7, we know that the expected distance for the Manhattan metric is 0.666667, and for the Euclidean metric, it is 0.521405. To produce Table A-5, we ran the simulation with the same random number seed but with a different number of iterations. In each iteration, one pair of random points is generated.

Number of Iterations	Expected Manhattan distance	Expected Euclidean distance	Computation time (seconds)
10	0.5015	0.3870	0.1
100	0.7567	0.5836	0.9
1000	0.6651	0.5184	8.4
10000	0.6664	0.5199	83.6

Table A-5: Expected distances and computation time at different numbers of iterations

Notice as the number of iterations increases, the expected distance approaches its known exact value. In a Monte Carlo simulation, there is no guarantee that this tendency will always occur, but theoretically, the variance of the error should decrease as the number of iterations increases. Also note that the computation time is virtually directly proportional to the number of iterations.

Example A-11. Other regular polygons

Various tests were run on other regular polygons. The results, while reasonably close to exact results as shown in Examples A-3 and A-8, could not be used to show how the expected distances approach that of a circle with the same area. The computation time required for

1000 iterations is shown below in Table A-6. Even though our version of the Monte Carlo

Number of sides	Computation time (seconds)
3	10.1
4	10.7
10	11.6
20	13.2

Table A-6: Effect of number of sides in a regular polygon on the computation time of a Monte Carlo simulation

simulation is $\mathcal{O}(n)$ where n is the number of sides, it is difficult to confirm this statement from the above table with values of n that are relatively small.

Note that the computation time required to perform 1000 iterations for a square in this example (10.7 seconds) does not match the apparently identical case in Example A-10 (8.4 seconds). The reason for this is that the unit square in Example A-10 was oriented differently. (Specifically, in Example A-10, the two sub-triangles of the square had alpha and beta parameters of 1. This number can be multiplied with less computational effort than for other real numbers.)

Example A-12. Concave polygons

The two previous examples were of convex polygons where the Monte Carlo simulation produced results that were consistent with our exact numerical methods. A main strength of the Monte Carlo method is that it can be used to find results for problems where analytic methods do not exist. For instance, using our Monte Carlo program, we can find the expected Euclidean distance in a concave polygon, or find second order (variance) statistics.

In our test with the polygon shown in Figure A-5, the simulation with 1000 iterations produced an expected Manhattan distance of 7.43 and an expected Euclidean distance of 6.01. In fact, the exact expected Manhattan distance is 7.584389 (from Example A-5). The computation time was 657.8 seconds–significantly more time is required to find the shortest path in a concave polygon.

Another example with the polygon shown in Figure A-6, the simulation with 1000 iterations produced an expected Manhattan distance of 17.86 (compared with 17.666657

from Example A-6), and an expected Euclidean distance of 17.22. The computation time in this case is 273.8 seconds.

Appendix B

Expected Euclidean Distance in Concave Polygons

In Chapter 3, we described a numerical method for finding the expected Euclidean distance between two random points in a convex polygon. Let us explore the added complexity of a method that handles concave polygons by considering the polygon in Figure B-1 that

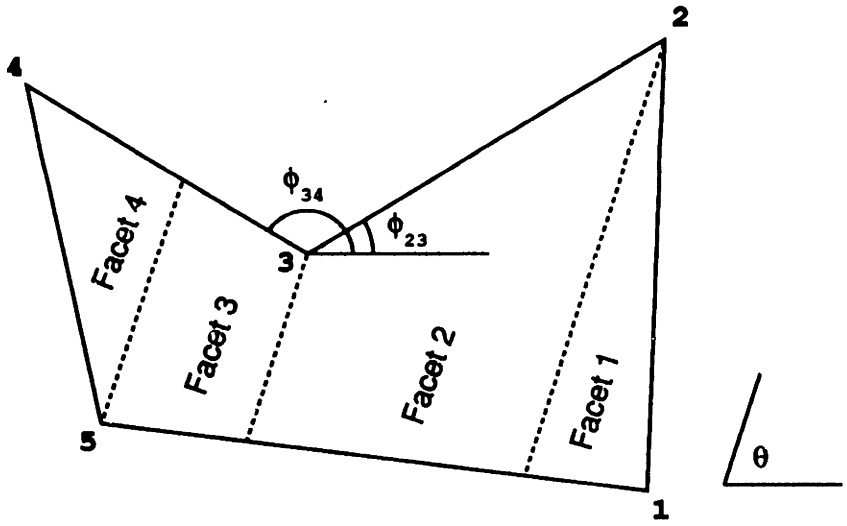

Figure B-1: Sample concave polygon with $\phi_{2,3} < \theta < \phi_{3,4}$

contains a concave angle at vertex 3. For this polygon, we can use the same Facet Term Expressions as we have done in section 3.2.3, but only for values of θ in the range $\phi_{2,3} \leq \theta \leq$

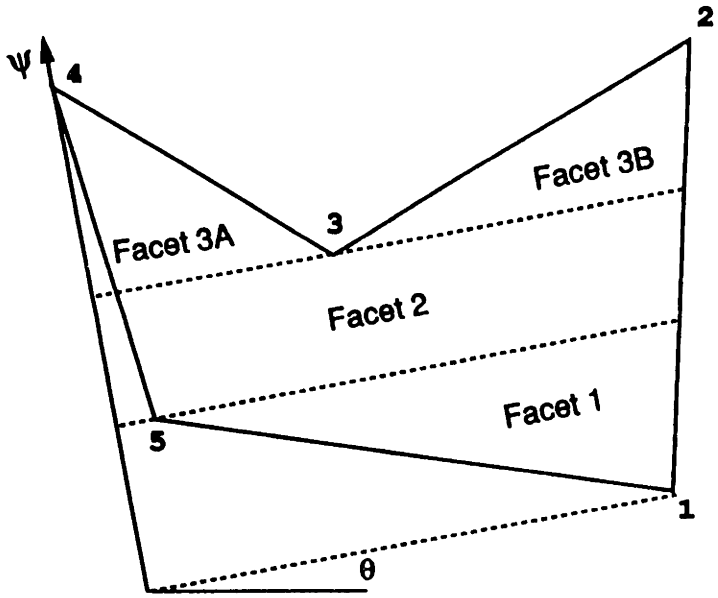

Figure B-2: Concave polygon where ψ values for different facets overlap

$\phi_{3,4}$. However, when θ is outside of this range, additional complications arise. For instance, at the value of θ shown in Figure B-2, the ψ range of facets 3A and 3B overlap. The length function $l(\theta, \psi)$ for the polygon cannot be defined as we had done for convex polygons since $l(\theta, \psi)$ is no longer a function of θ and ψ alone. The main complication, however, is in expressing the distance from a point q_A in facet 3A to a point q_B with the same value of ψ in facet 3B shown in Figure B-3. Because we assume that the sides of a polygon form a barrier to travel, a direct path from q_A and q_B is not possible, and the shortest path must "go around" vertex 3. Handling this condition presents a difficult problem.

However, utilizing some of the insights that we have gathered from both Chapters 2 and 3, we can suggest a possible approach. Conceptually, we can divide a concave polygon into "channels" for a particular range of θ. We define a channel such that the straight-line path between two points with the same value of ψ at a given θ does not cross any side of the polygon if and only if the two points are in the same channel. (For example, the points q_1 and q_2 in Figure B-4 are in the same channel; q_3 is not.) We can now divide our examination of expected Euclidean distances in concave regions into two parts: the "intrachannel distances", and the "interchannel distances". Note that the arrangement of channels depends upon θ. For the same polygon as in Figure B-4 but with a different value

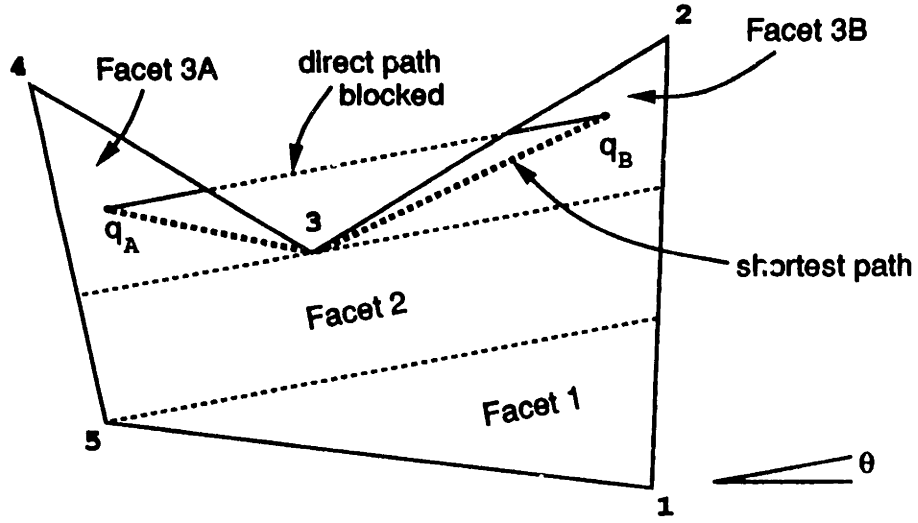

Figure B-3: Concave polygon where sides form barrier to travel

of θ, Figure B-5 shows only one channel (with all nine facets in the same channel).

With this definition of a "channel", let us begin by finding the intrachannel distances. As it turns out, almost all of the steps that we need have already been developed earlier for the Configurator Method in subsection 3.2.2:

1. Find the conjunction angles between all pairs of points. Discard any conjunction angle between a pair of points that cannot be connected by a straight line without crossing any side of the polygon. (This condition may be tested using the "uncrossed" routine developed in appendix A.1.) The number of conjunction angles, which is the same as the number of configurations C', is now such that:

$$C' < \frac{n(n-1)}{2}$$

2. Arrange the conjunction angles under consideration so that:

$$0 \leq \phi_1 \leq \phi_2 \leq \ldots \leq \phi_{C'} \leq \pi$$

Define ϕ_0 such that:

$$\phi_0 = \phi_{C'} - \pi$$

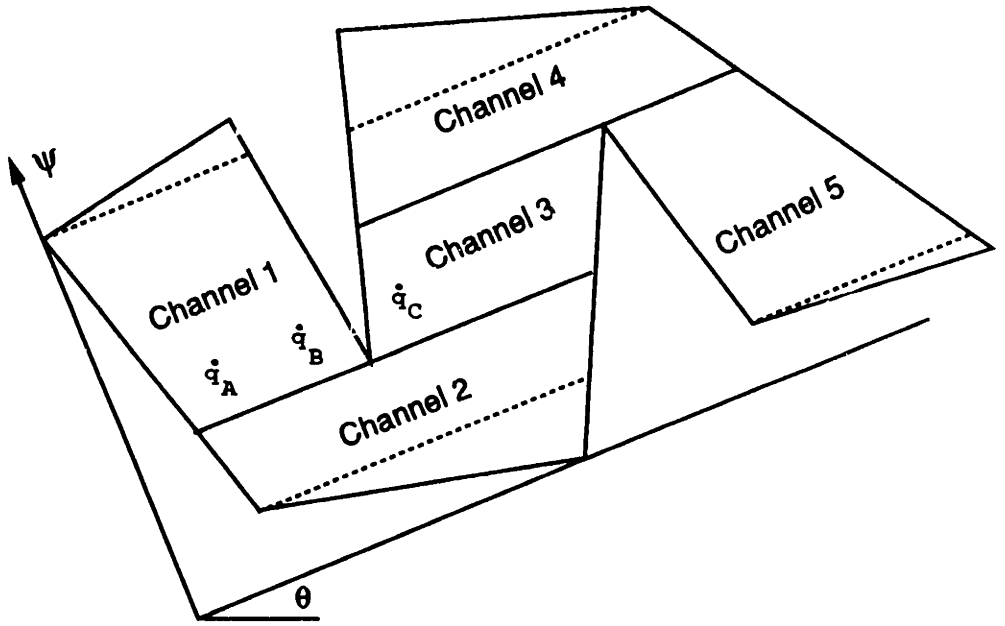

Figure B-4: Concave polygon shown with multiple channels

3. For each of the "configurations" that occurs between successive sorted conjunction angles, divide the polygon into channels and zones. We can use the same method presented in Chapter 2 with the only difference being that we replace the independent axis x by the ψ-axis, and the dependent axis y by the λ-axis. We note that.

$$\psi(x, y, \theta) = y \cos\theta - x \sin\theta$$

$$\lambda(x, y, \theta) = x \cos\theta + y \sin\theta$$

4. The resulting zones are equivalent to the facets defined in Chapter 3, and we can apply the Facet Term Expressions G and H derived in subsection 3.2.3 and tally the results into the numerator and denominator terms of the expected distance equation (3.21).

5. Generate the next configuration from the current configuration and go back to step 3. (The steps for generating the next configuration is somewhat more complicated than the method for convex polygons described in subsection 3.2.2.)

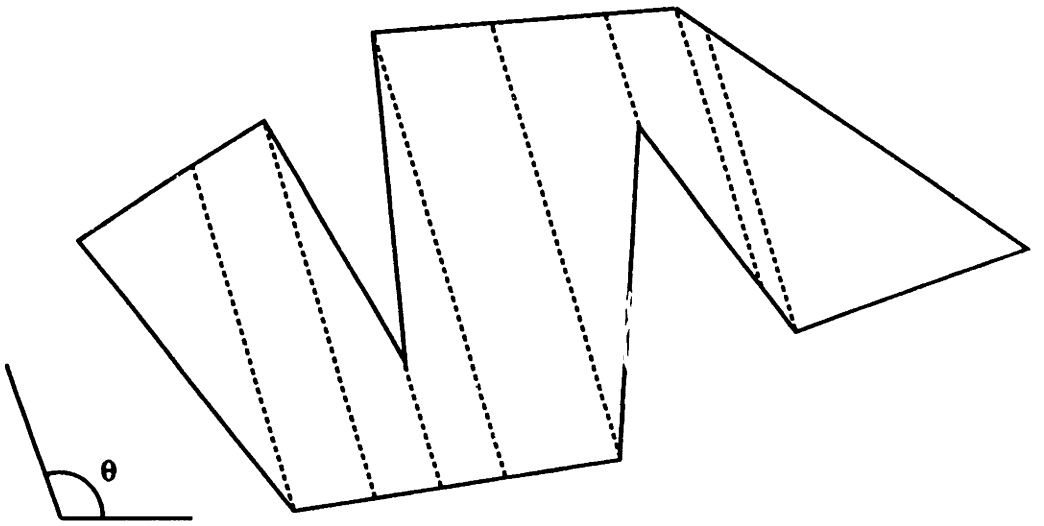

Figure B-5: Same polygon with one channel at different value of θ

The above procedure will only provide part of the solution, since the numerator terms and denominator terms only consider the intrachannel distances. We will now outline a possible approach for finding the interchannel distances that is conceptually equivalent to the strategy for finding the expected Euclidean distance in convex polygons that we developed in Chapter 3. Starting with Equation 3.5, we can separate the distribution of the two points q_1 and q_2 into separate cases: when the points are intrachannel–denoted by a prime symbol ($'$), and when the points are interchannel–denoted by a double-prime symbol ($''$).

$$\bar{\mathrm{D}}_E(polygon) = \frac{\begin{array}{l}\iint_Q \mathrm{f}_Q(q_1) \iint_Q \mathrm{f}'_{Q_2|q_1}(q_2|q_1)\mathrm{d}'_E(q_1,q_2|q_1)dq_2dq_1 + \\ \iint_Q \mathrm{f}_Q(q_1) \iint_Q \mathrm{f}''_{Q_2|q_1}(q_2|q_1)\mathrm{d}''_E(q_1,q_2|q_1)dq_2dq_1\end{array}}{\begin{array}{l}\iint_Q \mathrm{f}_Q(q_1) \iint_Q \mathrm{f}'_{Q_2|q_1}(q_2|q_1)dq_2dq_1 + \\ \iint_Q \mathrm{f}_Q(q_1) \iint_Q \mathrm{f}''_{Q_2|q_1}(q_2|q_1)dq_2dq_1\end{array}} \tag{B.1}$$

where the intrachannel probability density function $\mathrm{f}'_{Q_2|q_1}(q_2|q_1)$ is only defined when q_1 and q_2 are in the same channel:

$$f'_{Q_2|q_1}(q_2|q_1) = \begin{cases} f_Q(q_2) & \text{for } q_1 \text{ and } q_2 \text{ in same channel,} \\ 0 & \text{otherwise} \end{cases}$$

and where the interchannel probability density function $f''_{Q_2|q_1}(q_2|q_1)$ is only defined when q_1 and q_2 are in different channels:

$$f''_{Q_2|q_1}(q_2|q_1) = \begin{cases} f_Q(q_2) & \text{for } q_1 \text{ and } q_2 \text{ in different channels,} \\ 0 & \text{otherwise} \end{cases}$$

We can rewrite (B.1) to incorporate the intrachannel distance terms from Chapter 3:

$$\bar{D}_E(polygon) = \frac{\frac{1}{2}\sum_{i=1}^{C'}\sum_{j=1}^{D'_i}\sum_{k=1}^{F'_{i,j}} G'(i,j,k) + \iint_Q f_Q(q_1) \iint_Q f''_{Q_2|q_1}(q_2|q_1) d''_E(q_1,q_2|q_1) dq_2 dq_1}{\sum_{i=1}^{C'}\sum_{j=1}^{D'_i}\sum_{k=1}^{F'_{i,j}} H'(i,j,k) + \iint_Q f_Q(q_1) \iint_Q f''_{Q_2|q_1}(q_2|q_1) dq_2 dq_1} \quad \text{(B.2)}$$

where D'_i is the number of channels in configuration i, $F'_{i,j}$ is the number of facets or zones in channel j of configuration i, and G' and H' are the same Facet Term Expressions derived earlier except that the constants that were cancelled from the numerator and denominator during our earlier analysis are factored back into the expressions. We can rewrite Equation B.2 in terms of the interchannel distance expressions, which we call I and J, that are respectively summed into numerator and denominator.

$$\bar{D}_E(polygon) = \frac{\frac{1}{2}\sum_{i=1}^{C'}\sum_{j=1}^{D'_i}\sum_{k=1}^{F'_{i,j}} G'(i,j,k) + \sum_x I(x)}{\sum_{i=1}^{C'}\sum_{j=1}^{D'_i}\sum_{k=1}^{F'_{i,j}} H'(i,j,k) + \sum_x J(x)} \quad \text{(B.3)}$$

The summation over x implies a decomposition method that enumerates the possible scenarios in which two random points in a polygon are distributed in different channels. The function I and J are similar in function to the Facet Term Expressions (Equation 3.21) in that they represent the terms that express the expected distance for each such scenario.

Without formally deriving the expressions for I and J, we can gain some insight into the problem of finding the distance between points in different channels by studying the polygon in Figure B-6. According to our definition of a channel, for the configuration in $\phi_A \leq \theta \leq \phi_B$, the polygon contains three unique channels which are labelled in the figure as A, B, and C. The computation for the expected distance will involve four sets of the

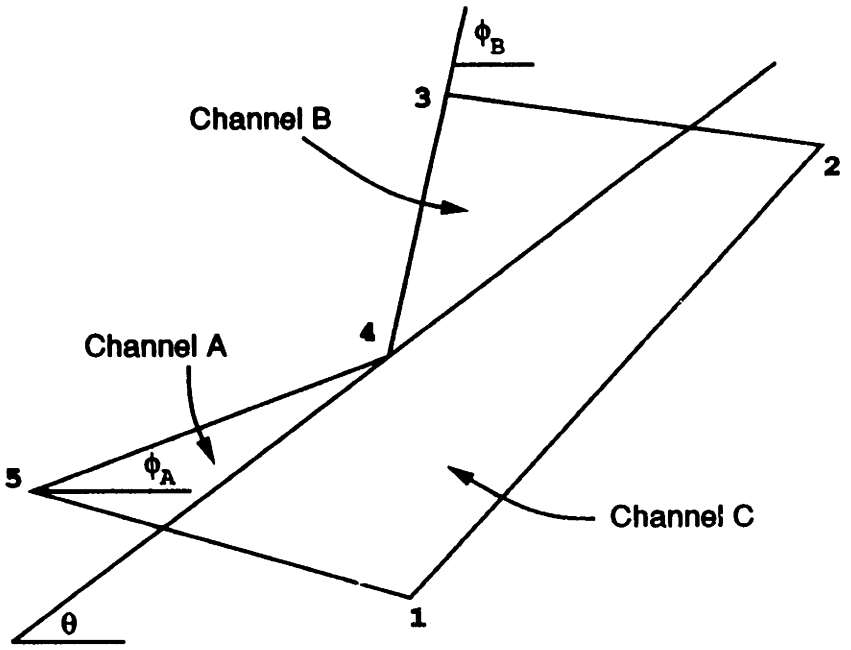

Figure B-6: Concave polygon with multiple channels

intrachannel Facet Term Expressions G' and H'–one for channel A, one for channel B, and two for channel C (since there are two facets in channel C). We have introduced another set of expressions I and J that represent the interchannel distances between points in channel A and channel B. The path from point q_A in channel A to a point q_B in channel B will have to "go around" the vertex point 4.

Unlike the Facet Term Expressions which are associated with a trapezoidal subregion of the polygon, I and J will consider a wedge-shaped (triangular) subregion where the apex of the "wedge" is at a vertex point around which the path between two points in different channels must pass. In Figure B-6, the vertex point 4 serves as the apex for both the wedge that is channel A and the wedge that is channel B. (In general, a channel may contain more than one wedge depending upon the polygon, much as a channel in Chapter 2 may contain more than one zone. Also, in general, the apex for wedges in different channels need not be the same.)

We propose to distribute one of the random points q_A in a thin "pie slice" of wedge A between the angles θ and $\theta + d\theta$. We distribute q_B in the "pie slice" between the angles ω

and $\omega + d\omega$ where ω is in the range $\theta \leq \omega \leq \phi_B$ as shown in Figure B-7. The probability

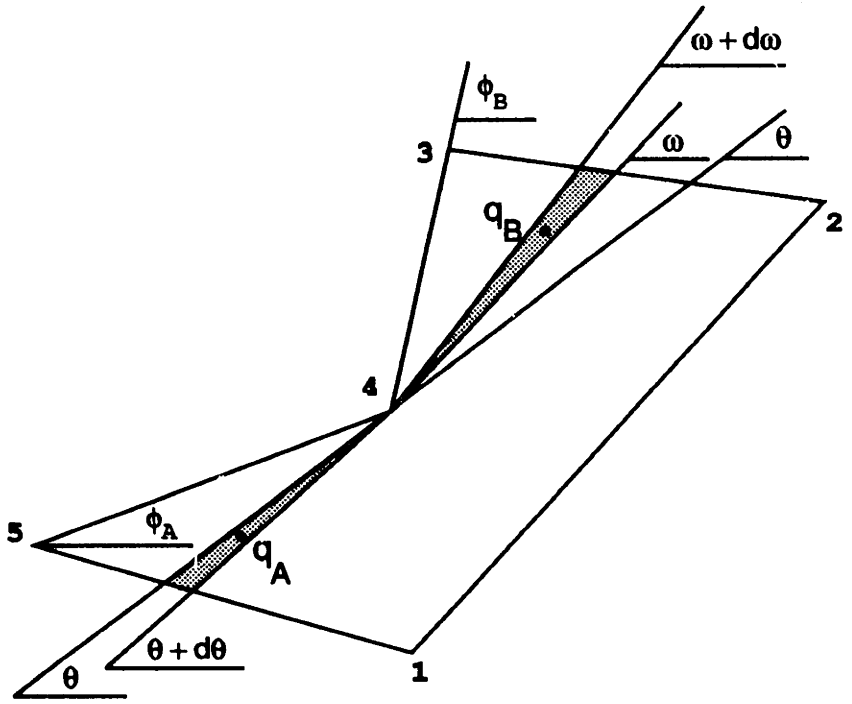

Figure B-7: Distribution of q_A and q_B

that a point q_A lies in the slice is related to the area of the slice, given our assumption of uniform distribution:

$$f_A(\theta) = \frac{1}{2} l_A(\theta) \cdot l_A(\theta)$$

where $f_A(\theta)$ is the unnormalized probability density function and $l_A(\theta)$ is the length of the "scan line" in the θ direction bordering channel A from vertex 4 to the far side of the polygon. We state without proof that the expected distance $d_{A,X}(\theta)$ between a random point q_A in the triangular slice and the apex is:

$$d_{A,X}(\theta) = \frac{2}{3} l_A(\theta)$$

Similarly, $f_B(\omega)d\omega$ is the unnormalized probability density function that a point q_B lies in the "slice" between the angles ω and $\omega + d\omega$.

$$f_B(\omega) = \frac{1}{2} l_B(\omega) \cdot l_B(\omega)$$

where $l_B(\omega)$ is the length of the "scan line" in the ω direction bordering channel B from the apex to the far side of the polygon. The expected distance $d_{B,X}(\omega)$ between a random point q_B in the slice and the apex is:

$$d_{B,X}(\omega) = \frac{2}{3} l_B(\omega)$$

The distance between q_A and q_B is simply the sum of the distance between the apex and q_A and the distance between the apex and q_B:

$$d_{A,B}(\theta, \omega) = d_{A,X}(\theta) + d_{B,X}(\omega)$$

In general, the distance expression between points in different channels that do not share a common apex will have a fixed component $d_f(A, B)$ representing the distance of the path between the apex points. Using the notation in Chapter 1, we can express the length of this component as:

$$d_f(A, B) = \sum_{k=1}^{m-1} \sqrt{(I_{X,k} - I_{X,k+1})^2 + (I_{Y,k} - I_{Y,k+1})^2}$$

where m is the number of intermediate apex points (which are all concave vertices) which are located at $(I_{X,k}; I_{Y,k})$.

Returning the simple case in Figure B-7, $m = 1$ so $d_f(A, B) = 0$ and we can express I and J as:

$$I_{A,B} = \int_{\phi_A}^{\phi_B} \int_{\theta}^{\phi_B} f_A(\theta) f_B(\omega)(d_{A,X}(\theta) + d_{B,X}(\omega)) d\omega d\theta \tag{B.4}$$

$$J_{A,B} = \int_{\phi_L}^{\phi_B} \int_{\theta}^{\phi_B} f_A(\theta) f_B(\omega) d\omega d\theta$$

The expressions for I and J remain to be evaluated.

This concludes our preliminary discussion on an approach for finding the expected Euclidean distance in concave polygons. A completely generalized method to handle concave polygons would be an important and interesting extension to the methods for finding the expected Euclidean distances covered in this thesis.

Bibliography

Behr, Merlyn J., and Jungst, Dale G.. *Fundamentals of Elementary Mathematics: Geometry.* New York: Academic Press, 1972.

Eilon, Samuel, Watson-Gandy, C.D.T., and Christofides, Nicos. *Distribution Management: Mathematical modelling and practical analysis.* New York: Hafner Publishing Company, 1971.

O'Rourke, Joseph. *Art Gallery Theorems and Algorithms.* New York: Oxford University Press, 1987.