

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

A

COMPUTING THE DOMAIN OF A DIGITAL CURVE

by

JONATHAN ROBINSON

**A dissertation submitted to the Graduate Faculty in Computer
Science in partial fulfillment of the requirements for the degree of
Doctor of Philosophy, The City University of New York**

2001

UMI Number: 3024828

Copyright 2001 by
Robinson, Jonathan

All rights reserved.

UMI[®]

UMI Microform 3024828

Copyright 2001 by Bell & Howell Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© Copyright 2001

Jonathan Robinson

All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

September 22, 2001

Date

John Golden

Chair of Examining Committee

Sept 20, 01

Date

[Signature]

Executive Officer

Dr. Terrance Boulton

Dr. Ted Brown

Dr. Robert Goldberg

Dr. Jacob Shapiro

Dr. Jerry Waxman

Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

COMPUTING THE DOMAIN OF A DIGITAL CURVE

by

JONATHAN ROBINSON

Adviser: Dr Robert Goldberg

Abstract: In this thesis we offer a methodology for computing the domain, the set of all estimates that would digitize to a digital curve. For line segments, this is accomplished using convex hulls and an extension of the convex hull: the *infinite hull*. By incorporating the α -hull, introduced by Edelsbrunner (1983), these results can be extended to compute the domain of digital circular arcs where the radius is known. We claim that similar results can be found when computing the domain of a standard elliptical digital curve by using the α, β -infinite hull that we introduce. These algorithms exhibit linear time during experimentation.

We then propose an algorithm that computes a digital circular arc's (circular) domain, under all radii. This algorithm is further applied to find the standard elliptical domain of standard elliptical digital curves under all radii. Both algorithms exhibit an $O(n)$ run time where n is the number of pixels in the digital curve. The approach used by the algorithm is then extended to compute the domain for gray scale images.

To Rivka

TABLE OF CONTENTS

General Contents

Chapter 1	Digital Curve Analysis: A Survey	1
Abstract		2
1.1.	Introduction	3
1.2.	Digital Curve Analysis	4
1.2.1	Digital Curve Properties	5
1.2.2	Digital Curve Approximation	6
1.2.3	Digital Curve Decomposition into Linear Components	7
1.2.4	Characterization and Domain Computation	8
1.2.5	Digital Circular Arcs	9
1.3.	Digitization, Images and Digital Curves	11
1.3.1	Reconstruction of a Digital Curve: Digital Line Segments	14
1.3.1.1	The Chord and Compact Chord Properties	15
1.3.1.2	Linear Decomposition of Digital Curves and Characterization	17
1.3.1.3	The Domain and Estimates of a Digital Line	21
1.3.2	Reconstruction of a Digital Curve: Digital Circular Arcs	26

Chapter 2	The Passing Area and Domain of a Digital	33
	Line Segment	
Abstract		34
2.	Introduction	36
2.1	The Duality of the Passing Area and the	40
	Domain	
2.2	The Digital Curve's Constraints on the	45
	Passing Area	
2.3.	Infinite Hull Algorithm	54
2.3.1	Classes From Java 2 and the Methods Used	55
2.3.2	Classes, Methods and Variables Headings	56
2.3.3.	The Complete Algorithm	59
2.3.4	Additional Cases	71
2.4	Extensions	72
2.4.1	Applications Of The Algorithm	72
2.4.1.1	Disconnected Line Segment Components	72
2.4.1.2	Inherent Parallelism of the Algorithm	74
2.4.2.	Further Research and Subsequent Chapters	76
Chapter 3	Computing The Domain of a Digital	78
	Circular Arc	
Abstract		79
3.1.	Introduction	80

3.2.	Background	81
3.2.1	Definitions	81
3.2.2	The Passing Area of a Digital Circular Curve	88
3.3.	Constraints on the Passing Area	90
3.3.1	Digital Curves as Separators	91
3.3.2.	Circular Separability	95
3.3.3.	Characterizing the Minimal Passing Area	99
3.4.	Computing the Passing Area of a Digital Circular Arc	100
3.4.1.	Boundary Edges That Contribute To The α -Hull and α_c -Hull	100
3.4.2.	The Algorithm For Determining the α_c -hull	109
3.4.2.1	The α -Hull Computation	110
3.4.2.2	Handling Elements of the Boundary that are Edges	116
3.4.2.3	Degenerate Cases with Edge Elements	121
3.4.2.4	The α -Separation Hull Constraint	122
3.5.	α_c -Hull Algorithm	128
3.5.1.	Classes from Java 2 and the Methods Used	129
3.5.2.	Classes, Methods and Variables Headings	129
3.5.3	The Complete Algorithm	134
3.5.3.1	Introduction	134

3.5.3.2	The Code	136
3.5.3.3	The Demonstration	142
3.6.	Extensions and Further Directions	150
3.6.1	Determining the Passing Area For All Radii	150
3.6.2	Disconnected Components and 8- Connected Curves	153
3.6.3	Inherent Parallelism	156
3.6.4	Further Research Directions	158
Chapter 4	Domain of a Digital Circular Arc Under All Radii	160
Abstract		161
4.1	Introduction	162
4.2	The Effects of α on the Passing Area	166
4.2.1	Trivial and Critical Changes	167
4.2.2	Critical Elements, Critical Ranges and Critical Sets	170
4.2.3	Computing the Critical Sets	177
4.2.3.1.	Conditional Critical Range	178
4.2.3.2	Computing the Conditional Critical Range	182
4.2.3.2.1	Case A: Three Vertices	186
4.2.3.2.2	Case B: Two Vertices and One Edge	187
4.2.3.2.3	Case C: One Vertex and Two Edges	191

4.2.3.2.4	Case D: Three Edges	197
4.3	Algorithm for Domain Computation of a Digital Circular Arc	202
4.3.1	Algorithmic Details	204
4.3.1.1	Phase 1 – Creating the Candidate List	204
4.3.1.2	Discussion of Phase 1	208
4.3.1.3	Phase 2 – Computing the Minimum Preimage Arc, $r_{\min}$	209
4.3.1.4	Phase 3 – Updating the Critical Range (Iteration)	209
4.3.1.4	Discussion of Phase 3	211
4.3.2	Iterations of the Algorithm	211
4.3.3	Algorithm Run Time	217
4.4	The Shape of the Domain	220
4.4.1	Digitization Models	224
4.4.2	Effects on the Domain	226
Chapter 5	Gray Scale Image Constraints on the Passing Area	230
Abstract		231
5	Gray Scale Images	232
5.1	Constraint Imposed on the Passing Area by the Pixel's Gray Level	234

5.2	Objects Bounded By Line Segments	236
5.2.1	Linear Pixel Configurations	236
5.2.2	Convex Pixel Configurations	238
5.2.3	Concave Pixel Configurations	242
5.3	Objects Bounded By Circular Arcs	244
5.3.1	Linear Pixel Configurations	244
5.3.2	Concave and Convex Pixel Configurations	258
5.4	Gray Scale Error	264
5.5	Summary	274
Chapter 6	The Passing Area and Domain of a Digital Standard Elliptical Arc (Part I): Fixed Semi- Axes Lengths	276
Abstract		277
6.	Digital Elliptical Curves	278
6.1	Definitions of Geometric Constructs	278
6.2	Implementation Considerations	285
6.3	Disconnected and Parallel Curves	294
6.4	Summary	296
Chapter 7	The Passing Area and Domain of a Digital Elliptical Arc (Part II): Fixed Rotation and	297

	Fixed Eccentricity for All Semi-Axes Lengths	
Abstract		298
7.	Introduction	299
7.1	The Standard Elliptical Passing Area for All Radii	299
7.2	The Shape of the Elliptical Domain	321
7.3	General Ellipses	331
Chapter 8	The Passing Area Solver Software (PASS) Application	335
Abstract		336
8.	Passing Area Solver Software (PASS) Package	337
8.1	The User Interface	338
8.2	The Draw Panel	339
8.2.1	Shapes	340
8.2.1.1	Lines	341
8.2.1.2	Circles and Ellipses	343
8.2.2	Draw Style	346
8.3	The α Panel	347
8.3.1	Plotting a Digital Curve and the Inner and Outer Boundaries	348

8.3.2	Computing the Convex. α and $\alpha\beta$ -hulls of the Joined Boundary	350
8.3.3	Computing the Hulls and Critical Sets as the Radius or Semi-Axes Change and the Domain	355
8.4	Summary	360
Appendix	Java Source Code for PASS	362
Area.java		363
Chain.java		371
Circle.java		379
Domain.java		381
DrawContext		388
DrawScrollPane		390
Ellipse.java		409
EllipseArc.java		424
Image.java		426
Lines.java		460
Bibliography		464

LIST OF ILLUSTRATIONS, CHARTS, DIAGRAMS

Section 1

Figure 1.1	4- and 8-connected digital curve.	5
Figure 1.2.	Correspondence of lines in the (x,y) plane and (m,b) parameter space.	9
Figure 1.3.	Digitization under OBQ and GIQ.	12
Figure 1.4.	The 8 Freemanchain values	13
Figure 1.5.	The multi-level pattern found in a digital line's chaincode	16
Figure 1.6.	Illustrations of the chord properties.	17
Figure 1.7.	The (n,p,q,s) characterization of a digital line.	20
Figure 1.8.	The upper and lower convex hulls, C_u and C_l .	24
Figure 1.9.	The circular arc domain of a digital curve under all radii.	29
Figure 1.10.	The domain of fixed radius of a digital curve and the changes in the relevant point set.	32

Section 2

Figure 2.1.	A digital curve.	36
Figure 2.2.	The convex hull of the upper and lower boundaries superimposed on the grid.	37
Figure 2.3.	An estimate of the preimage.	38

Figure 2.4.	Unrealizable and realizable estimates of a digital line segment.	39
Figures 2.5.-2.7	Correspondence of the passing area and domain.	42-3
Figure 2.8.	Line segments and line segment extensions define the passing area of the digital line segment.	44
Figure 2.9.	Correspondence of the passing area and domain	45
Figure 2.10.	A non-minimal passing area obtained by the convex hull algorithm.	46
Figure 2.11.	The truncated pixel boundary.	47
Figure 2.12.	Convex hull constraints on a line passing between two point sets.	48-9
Figure 2.13.	$\text{hull}_{\text{sep}}(S_1, S_2)$.	49
Figure 2.14.	The infinite hull of S_1 and S_2 .	50
Figure 2.15.	The infinite hull minimization of the passing area.	51
Figure 2.16.	The truncated boundary where upper and lower hulls are connected by adding line segment extensions.	53
Figure 2.17-2.21.	Illustration of significant events during the algorithm's iteration.	57-61
Figure 2.22.	The Java 2 code for the infinite hull algorithm.	62-4
Figure 2.23-2.36.	Iteration of the infinite convex hull algorithm.	65-9
Figure 2.37.	The domain in (m,b) .	70
Figure 2.38.	The passing area of a single pixel.	71
Figure 2.39.	The initial passing area of a disconnected curve.	73

Figure 2.40.	The infinite hull of the disconnected line segment	73
Figure 2.41.	The domain of the disconnected line segment shown in figure 2.39	74
Figure 2.42.	Inherent parallelism.	75

Section 3

Figure 3.1.	The upper and outer pixel boundaries of a digital curve	82
Figure 3.2.	The joined boundary of the digital curve..	83
Figure 3.3.	The passing area of a digital curve.	84
Figure 3.4.	The domain of the digital curve presented in figure 3.3.	84
Figure 3.5.	α -hulls for different values of α .	86
Figure 3.6.	A digital circular curve.	87
Figure 3.7.	The passing area of the digital curve where the radius equals 8.	87
Figure 3.8.	Correspondence of passing area and domain.	89
Figure 3.9.	$\text{hull}_{\text{sep}}(S_1, S_2)$.	92
Figure 3.10.	The infinite hull of S_1 and S_2 .	93
Figure 3.11.	$\alpha\text{-hull}_{\text{sep}}(S_1, S_2)$ where $\alpha > 0$ and $\alpha\text{-hull}_{\text{sep}}(S_2, S_1)$ where $\alpha < 0$.	94
Figure 3.12.	The α_{∞} -hull of S_1 and S_2 .	95
Figure 3.13.	Circular Separability.	96
Figure 3.14-15.	α -hull non-intersection constraints.	96-7

Figure 3.16.	The passing area of a digital circular curve.	99
Figure 3.17.	The hull of a set of line segments.	101
Figure 3.18	The α -hull increases with α .	101
Figure 3.19.	Determining the α -hull where $\alpha > 0$ and where $\alpha < 0$.	102
Figure 3.20.	Edges that may contribute to the α -hull.	104
Figure 3.21.	The α -hull of the edge pairs from the perspectives of the concave and convex.	105
Figure 3.22-23	Computing the α -hull of the outer boundary.	106-7
Figure 3.24.	Only edges that are at the digital stationary points contribute to the α_{∞} -hull.	109
Figure 3.25.	The α -hull of the inner boundary.	111
Figure 3.26.	Two ordered points, p_i and p_j , and a curvature α , define a single circular edge.	112
Figure 3.27-29	Algorithm's Iteration	114-6
Figure 3.30b.	The eight possibilities for the ordering of three consecutive elements along the digital curve's pixel.	117
Figure 3.31.	Edges and vertices that contribute to the passing area.	118
Figure 3.32.	Three unique orderings of the first two elements exist modulo symmetry.	119
Figure 3.33.	Two elements and curvature, α , define a unique circle.	119
Figure 3.34-5.	Determining the portion of an edge that is along the passing area boundary.	120-1

Figure 3.36.	The α -hull digital curve passing area.	123
Figure 3.37-8.	Iteration when two consecutive elements are on different boundaries.	125-7
Figure 3.39.	Resetting an edge's endpoint.	135
Figure 3.40.	The Java 2 source code for the α -hull algorithm.	136-41
Figure 3.41.	The elements along the passing area.	142
Figure 3.42-53.	α -infinite hull algorithm iteration	143-9
Figure 3.54.	The complete passing area and domain.	149
Figure 3.55.	Similar passing areas for different α values.	151
Figure 3.56.	Changes in the passing area as α changes.	152
Figure 3.57.	Connecting disconnected components.	154
Figure 3.58.	Single pixel components.	155
Figure 3.59.	8-connected digital curves.	155
Figure 3.60.	Removing unnecessary vertices.	157

Section 4

Figure 4.1.	The inner and outer pixel boundaries of a digital circular arc.	163
Figure 4.2.	The joined boundary of the digital curve.	164
Figure 4.3.	The α -hull and the domain of the digital curve..	165
Figure 4.5.	Trivial changes in the passing area as the radius changes.	168

Figure 4.6.	Critical changes as the radius changes.	169
Figure 4.7.	The radial value where critical changes occur.	170
Figure 4.8.	Critical ranges of passing area elements	172
Figure 4.9.	Critical elements along the passing area perimeter.	173
Figure 4.10.	The 10 critical sets of a digital curve and their critical ranges.	174
Figure 4.11-20.	The passing areas representing each of the 10 critical sets.	175-7
Figure 4.21.	Conditional critical ranges for elements on the outer boundary.	180
Figure 4.22.	Conditional critical ranges for elements on the inner boundary.	181
Figure 4.23.	The criteria for applying the appropriate boundary rule to determine the conditional critical ranges	184
Figure 4.24.	The eight permutations of the identities of the three elements e^- , e and e^+ used in determining $r_{\text{circle}}(e^-, e, e^+)$.	184
Figure 4.25.	The unique circles that intersect two points and is tangent to a line in order counter clockwise.	188
Figure 4.26.	The unique circles that intersect two points and an edge endpoint.	191
Figure 4.27.	The circle that intersects a point and two edges at most once each.	197
Figure 4.28a.	The circle that is tangent to three edges.	198

Figure 4.28b.	The circle that intersects three edges once each.	201
Figure 4.29.	The pixels in I^2 and the pixel area represented by each pixel.	203
Figure 4.30.	The non-intersection constraint of the passing area.	205
Figure 4.31.	Candidate critical elements.	207
Figure 4.32.	The minimum radial value for which a passing area exists.	213
Figure 4.33-39	The eight critical sets of the digital curve.	213-6
Figure 4.40.	Table summarizing the critical sets and their conditional ranges.	217
Figure 4.41.	The number of critical sets plotted against the curve's radius.	219
Figure 4.42.	The number of critical sets plotted against the digital curve's size.	219
Figure 4.43.	The intersection of discs and disc complements centered at points on the inside and outside of the digital circular arc is the digital curve's domain.	221
Figure 4.44	The intersection of cones and cone complements centered at points on the inside and outside of the digital curve yield the complete circular arc domain of the digital curve.	222
Figure 4.45.	The closest point Voronoi diagram and the furthest point Voronoi.	223

Figure 4.46.	An object bounded by a curve is digitized under three schemes: OBQ, GIQ and luminescence.	225
Figure 4.47.	Edges that are at the digital stationary points may contribute to the α -infinite hull.	227
Figure 4.48.	The medial curve that is the boundary of the region where $d(x,p) \leq d(x,q)$.	228
Figure 4.49.	The domain of the digital curve with the superimposed parabola	229

Section 5

Figure 5.1.	A star digitized as a binary image with minimal and the gray scale image of a circular disc.	233
Figure 5.2.	A close-up view of a gray scale digital boundary.	234
Figure 5.3.	The inner and outer boundaries of a digital curve.	234
Figure 5.4.	The pixel median of the middle pixel is shown for three connected pixels that are linear, convex and concave.	235
Figure 5.5.	The linear boundary of any region with area A will intersect the Δ -point where $\Delta = A$.	237
Figure 5.6.	The Δ -point.	238
Figure 5.7.	$\Delta_{\min}$ and $\Delta_{\max}$ of a convex pixel configuration.	241
Figure 5.8.	A graph of $\Delta_{\min}$ and $\Delta_{\max}$ as the area, A , changes.	241
Figure 5.9.	$\Delta_{\min}$ and $\Delta_{\max}$ of a concave pixel configuration.	242

Figure 5.10.	A graph of $\Delta_{\min}$ and $\Delta_{\max}$ as the area, A , changes.	243
Figure 5.11.	The geometry of the pixel and intersecting circular arc.	245
Figure 5.12.	Δ is at a minimum at $\alpha = \frac{\pi}{2}$.	253
Figure 5.13.	Determining $\Delta_{\max}$.	254
Figure 5.14.	An upper bound on the value of $\Delta_{\max}$.	255
Figure 5.15.	A lower bound on $\Delta_{\max}$.	256
Figure 5.16.	An upper bound on the Δ -point when A is less than 0.5.	257
Figure 5.17.	An upper bound on the Δ point when A is greater than 0.5.	258
Figure 5.18.	The values of $\Delta_{\min}$ and $\Delta_{\max}$ of a convex pixel configuration.	259
Figure 5.19.	The values of $\Delta_{\min}$ and $\Delta_{\max}$ of a concave pixel configuration.	259
Figure 5.20.	A graph of Δ with respect to the radius r .	260
Figure 5.21.	An upper limit on the Δ -point of a convex pixel configuration.	261
Figure 5.22.	An upper limit may be placed on the Δ -point of a concave pixel configuration.	263
Figure 5.23.	The upper and lower bounds for $\Delta_{\min}$ and $\Delta_{\max}$.	264
Figure 5.24.	Error considerations for A .	265
Figure 5.25.	A line segment superimposed on the grid and the digital curve that is its digitization.	266

Figure 5.26.	The gray scale constraints superimposed on the grid.	267
Figure 5.27.	The passing area as determined by including the gray scale constraints.	267
Figure 5.28.	The size of passing area of the digital line segment decreases as the number of gray levels increases.	268
Figure 5.29.	The average size of the passing area decreases as the number of gray scale levels increases.	269
Figure 5.30.	The size of passing area of the digital circular curve decreases as the number of gray levels increases.	270
Figure 5.31.	The average size of a circular digital curve's passing area decreases as the number of gray scale levels increases.	270
Figure 5.32.	The region of each pixel the is included within the passing area decreases as the digital curve's length increases.	271
Figure 5.33.	The passing area size decreases as the gray scale levels increases and curve length increases.	272
Figure 5.34.	A gray scale image and its passing area.	273
Figure 5.35.	The gray scale constraints and the passing area.	273

Section 6

Figure 6.1.	A standard ellipse with horizontal radius, r_x , and vertical radius, r_y .	279

Figure 6.2.	The $\alpha\beta$ -hull of a point set where $\alpha, \beta > 0$.	280
Figure 6.3.	The $\alpha\beta$ -hull of a point set where $\alpha, \beta < 0$.	280
Figure 6.4.	The $\alpha\beta$ -separation hull of two point sets, where $\alpha, \beta > 0$.	281
Figure 6.5.	The $\alpha\beta$ -separation hull of two point sets, where $\alpha, \beta < 0$.	282
Figure 6.6.	The $\alpha\beta$ -infinite hull of two point sets, P1 and P2.	283
Figure 6.7.	The inner and outer boundaries of a digital elliptical curve.	284
Figure 6.8.	The passing area of the digital elliptical curve.	285
Figure 6.9.	Two vertices and the semi-define 2 ellipses.	287
Figure 6.10.	One vertex p_i and one edge and the semi-axes define 2 ellipses.	288
Figure 6.11.	One vertex and one edge and the semi-axes define 2 ellipses.	289
Figure 6.12.	Two perpendicular edges have four tangent ellipses.	290
Figure 6.13.	The ellipse determined by a point and a line.	292
Figure 6.14-15.	The ellipse that intersects a point and an edge at exactly one point.	292-3
Figure 6.16.	Comparing the $\alpha\beta$ -infinite hull of a connected and disconnected digital elliptical curve.	294
Figure 6.17.	The $\alpha\beta$ -infinite hull of a disconnected digital curve with a single pixel component.	295

Section 7

Figure 7.1.	A standard ellipse with horizontal radius, r_x , and vertical radius, r_y .	300
Figure 7.2.	The passing area of the digital elliptical curve.	300
Figure 7.3.	The ellipse with known semi-axes ratio that intersects three points is unique.	302
Figure 7.4.	The four possible identities of e^- , e and e^+ .	303
Figure 7.5.	Two circles exist that intersect two points and are tangent to a line.	307
Figure 7.6.	The ellipse intersecting an endpoint of the edge without including any portion of the edge.	310
Figure 7.7.	Four ellipses that intersect the vertex and an endpoint of the two edges.	316
Figure 7.8.	Six ellipses that are tangent to two edges and intersects the third edge's endpoints.	319
Figure 7.9.	The domain as the intersection of eccentric cones and eccentric cone complements.	321
Figure 7.10.	Skewed Euclidean distance 1.	323
Figure 7.11.	Skewed Euclidean distance 2.	323
Figure 7.12.	The skewed bisector.	326
Figure 7.13.	Determining the medial curve of a point and line using	327

skewed Euclidean distance.

Figure 7.14.	Parabolas as the locus of ellipse centers.	329
Figure 7.15.	The domain is bounded by skewed bisectors and parabolas.	330
Figure 7.16.	r-space is partitioned into regions such that each region has at most one associated critical set	331
Figure 7.17.	The projection of a digital elliptical curve's domain onto the (r_x, r_y) plane.	333
Figure 7.18.	Critical regions that partition the semi-axis (r_x, r_y) space.	334

Section 8

Figure 8.1.	A screen view of PASS.	337
Figure 8.2.	The draw command panel and the α -Hull command panel.	339
Figure 8.3.	Displaying lines, circles and ellipses.	340
Figure 8.4.	Information recorded in the status bar.	341
Figure 8.5.	The status bar recording the mathematical description of the last curve drawn.	342
Figure 8.6.	Perpendicular and skewed bisectors.	343
Figure 8.7.	Three points, in order counter clockwise, specify a circular arc.	344
Figure 8.8.	Drawing an ellipse	346

Figure 8.9.	Color palette.	347
Figure 8.10.	Disconnected digital curves.	348
Figure 8.11.	Constructing digital curves based on the curve's parameters.	349
Figure 8.12.	The inner and outer boundaries of the digital curve.	350
Figure 8.13.	The convex hull constraints of the upper and lower boundaries.	351
Figure 8.14.	The perimeter of the infinite convex hull of the digital line segment.	351
Figure 8.15.	The α -hull a digital circular arc.	352
Figure 8.16.	The α -infinite hull of a digital circular arc.	353
Figure 8.17.	The $\alpha\beta$ -hull of a digital elliptical arc.	354
Figure 8.18.	The $\alpha\beta$ -infinite hull of a digital elliptical arc.	354
Figure 8.19.	The critical ranges in and critical sets of a digital curve.	356
Figure 8.20.	The domain of the digital curve is shown.	356
Figure 8.21.	The critical sets of the digital curve interpreted as an ellipse.	357
Figure 8.22.	The domain of the digital curve interpreted as an ellipse.	358
Figure 8.23.	The critical sets of the digital curve interpreted as an ellipse with variable <i>ratio</i> .	359
Figure 8.24	The domain of the digital curve interpreted as an ellipse with variable <i>ratio</i> .	360

Chapter One:

Digital Curve Analysis: A Survey

Abstract: The digitization process yields a digital image from the world scene. The digital image has significantly less information than the scene including coarser pixel granularity and a smaller color spectrum. In order to compensate for this information deficiency, a number of geometric domains must be better understood. In this chapter, we present a survey of the areas that have most significantly contributed to resolving this problem: properties of digital curves, preimage estimates, primitive interpretations of maximal length and domain computation for both digital line segments and digital circular arcs.

1.1. Introduction

Understanding vision has been an elusive endeavor since antiquity. The complexity of the physical optics of the eye, the interpretive function of the brain and the interaction of the two through the optic nerve are a major cause for the difficulty in unraveling the enigma known as vision. The revolutionary advent of high-powered computers coupled together with cameras has broadened the horizon of the vision researcher from understanding visual systems to building them. Since biological vision is significantly more complex than its digital counterpart, problems that seem intractable in the biological domain are more solvable in the less complex digital arena.

In the real world a visual scene is assumed to have infinite granularity. That is, the magnification of the scene can, for most purposes, increase the detail ad infinitum. In addition, world scenes are perceived in color and, with the appropriate sensors, in frequencies that can be off the color scale. In the simplified digital model we study, those scenes are *represented* by images of finite granularity where each pixel within the image is binary – either black or white. The goal of the papers cited here is to understand, qualify and characterize the shapes of the objects found in the image so that the original scene can be interpreted.

In order to recognize objects found in a digital image, an important component of the complex process of image understanding is necessary, that of digital curve recognition. A two or three dimensional image may contain objects whose boundaries are easily found. Those boundaries define the shape of the objects they bound. Understanding those boundaries is useful in understanding the images they define.

1.2. Digital Curve Analysis

Research in the field of digital curve analysis can be divided into five areas all of which aim to understand the effects of the digitization process on a continuous curve. The five areas of study are:

- 1) digital curve properties (such as the chord property),
- 2) approximation of the preimage (the image that was digitized),
- 3) decomposition of a curve into maximal linear segments,
- 4) characterization and domain computation of the digital curve (e.g. succinctly describing the digital line segment and determining the set of real line segments that digitize to the digital line segment),
- 5) the above four areas with respect to digital lines, digital circular arcs and curves of greater complexity.

In order to simplify matters and define a problem that is somewhat tractable, research has dealt with understanding the shapes of the digital curves assuming the curves are mathematically simple. Thus, much of the study has focused on digital line segments and circular arcs (and possibly ellipses). Before we proceed to discuss the papers that deal with the topics mentioned the areas just outlined are briefly defined.

1.2.1 Digital Curve Properties

The first area, digital curve properties, is the study of the somewhat abstract notion of the properties that digital curves must obey. One property that the curves studied obey is called connectivity (see figure 1.1). A digital arc is connected set of pixels such that each pixel on the curve has two neighbors on that curve except for the first and last pixels that have only one neighbor. The curve may either be 4 or 8-connected. A 4-connected curve will contain a path from the first to last pixel with a single pixel move either vertically or horizontally. An 8-connected curve may contain diagonal moves as well, as in figure 1.1.

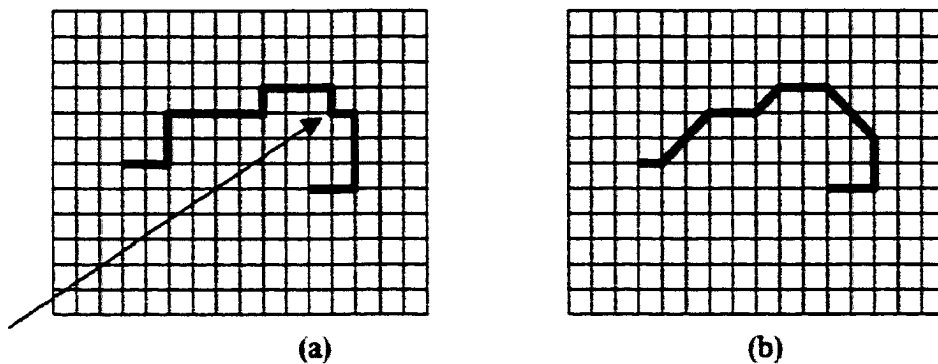


Figure 1.1 (a) A 4-connected and (b) 8-connected digital curve. Note that (a) is not an 8-connected digital arc since the indicated pixel has three 8-neighbors.

More pertinent to this research are the properties that digital line segments and digital circular arcs must obey. A digital line segment is a digital curve that is the result of the digitization of a line segment. Similarly, a digital circular arc is the digitization of a

circular arc from the image. (The precise process that takes place during digitization, also known as the digitization scheme will be discussed at length later (sections 1.3 and 4.4.1)) For example, Freeman (1970) stated that a digital line segment has a repetitive nature (that reflects the line's slope) and thus successive occurrences of both vertical and horizontal movements on the grid are "as uniformly spaced as possible. Rosenfeld (1974), Wu (1980) and Smeulders (1984) made this rather general property more concrete by showing that this repetitive nature is found at every level within the digital line. Simply stated, although the slope of line segments connecting points in a digital line varies over the entire digital line, the digital line's slope, nevertheless, is relatively stable. Thus, researchers in this subfield of digital curve analysis try to understand the regularities found in digital curves and the rules that those curves must obey.

1.2.2 Digital Curve Approximation

The second major area of investigation is approximating the real curve as it existed before digitization. This curve is known as the predigitization "image", or preimage. Estimates exist for many properties that a preimage has. For example, the slope, y-intercept (Koplowitz, 1987; Dorst and Smeulders, 1986), and length of line segments can be estimated (Dorst and Smeulders, 1987). For digital circular arcs, a system may approximate the centers, radii (Worring, 1993 and 1995) and arc length. For nonlinear digital arcs, estimates are found for the arc length (Freeman, 1970; Dorst and Smeulders, 1987) and tangent to each point in the digital curve (Koplowitz, 1987).

1.2.3 Digital Curve Decomposition into Linear Components

The third area of investigation is the decomposition of a curve into maximal linear segments, approximating the nonlinear curve with connected linear segments. This is similar to the notion from calculus that a curve can equivalently be perceived as connected line segments of infinitesimal length. (Though in the digital domain, the approximation of a digital nonlinear curve is by connected digital lines of *maximal* length. Using digital lines of minimal length would not yield accurate and thus useful results since the discretization of the grid does not retain detailed information regarding the curvature of the preimage when using those minimal length lines. The range of both slope and y-intercept for such small lines would be too wide to yield accurate estimates of the original curve).

Research in this area is premised on the fact that any digital curve is locally linear. That is, there *exists* a line segment that may be the preimage of a subsection of the curve. It also presumes that estimates that are realizable are computable. A *realizable* estimate is an estimate that would digitize to the digital curve as opposed to estimates which approximate the preimage such as estimates based on minimal mean square of the pixels in the curve. The researchers have found that finding maximal linear segments within a digital curve is useful in estimating arc length (Dorst and Smeulders, 1987), curvature (Koplowitz, 1987) and maximum error with curves whose preimage are connected line segments (Yuan and Suen, 1995).

1.2.4 Characterization and Domain Computation

The fourth subject of study, and the most recent, is the *domain* computation and characterization of digital curves. A digital curve whose preimage is a given primitive shape (line segment, circular or elliptical arc, etc.) can be the digitization of a fixed set of those primitives. This set of primitives that digitize to the digital curve is known as the digital curve's domain with respect to that primitive. The parameter space for which the domain is defined changes from one primitive to the next. For example, the domain for digital lines spans two dimensions: the slope and the y-intercept, as shown in figure 1.2. With digital circles the domain can be defined in a three-dimensional parameter space where two of the dimensions are for describing the circle center (that lies in the plane of the circle) and the third dimension for the circle radius. Clearly, the more complex the curve the more dimensions found in the domain.

A characterization of a digital curve provides a useful description of some of the properties of that digital curve. Many characterizations exist for any given curve; some of which are more useful for specific applications. For example, the number of pixels a digital curve contains can characterize the curve. This minimal characterization, for example, can be useful in estimating the curve's length (i.e. the arc length can be estimated to be the pixel count). For greater accuracy a digital curve can be characterized by two integers: the number of horizontal and diagonal neighbors in the digital curve. This has been found to yield more accurate estimates of the arc length. Other characterizations exist, but, the most accurate characterization, called a *faithful* characterization (Dorst and Smeulders, 1987), is a complete description of the digital

curve that can be used to reconstruct the entire digital curve. Thus, a bijection exists between each of the following three sets: digital curves, faithful characterizations and domains.

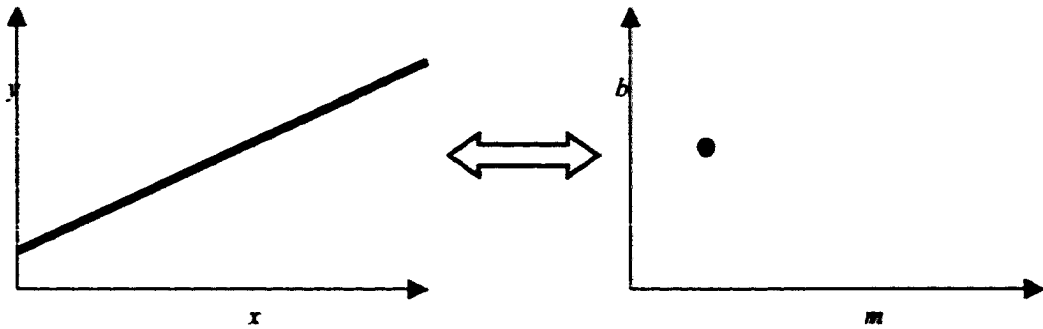


Figure 1.2. Every line in the (x,y) plane corresponds to a point in the (m,b) parameter space. Similarly, though not shown, a line in the (m,b) parameter space corresponds to a point in the (x,y) domain (through which all lines must intersect.) An area in the primitive's parameter space represents a digital curve's domain.

1.2.5 Digital Circular Arcs

The fifth area of study applies the first four areas in curve analysis, which were focused on digital line segments, to digital circular arcs. All of the fields of research outlined above have been studied with respect to digital line segments and to a lesser degree to digital circles. For example, researchers have been far more successful in defining properties of digital lines than digital circles. Recently, Worring and Smeulders

(1995) have successfully computed the domain for digital circles and have found superior estimates. However, regarding characterization, Smeulders (1987) conjectures that he believes that no faithful characterization exists for circles.

These five areas of study are rather dependent on each other. As noted by Ronse (1989) in his exhaustive bibliography on digital and computational convexity, many of the papers in one field of digital curve analysis are interdependent on other fields within this general subject. For example, the properties of a digital curve are used to characterize them. See Dorst and Smeulders (1984) who used the repetitive nature of digital lines first introduced by Freeman (1970) and Rosenfeld (1974) to characterize any digital line segment as a 4-tuple. Similarly, preimage approximations can be used to decompose a nonlinear digital curve into digitally linear components that approximate the shape of the digital curve (Sharaiha and Christofides, 1993). Also, a digital curve's domain can be used to find an estimate that is both realizable (i.e. would actually digitize to the digital curve as opposed to only being similar to the preimage) and optimal over the entire solution space of possible preimages (Dorst and Smeulders, 1986; Worring and Smeulders, 1995). Furthermore, concepts from computational geometry such as visibility (Sharaiha and Garat, 1993), convex hulls (Koplowitz, 1987), Voronoi diagrams and alpha hulls (Edelsbrunner, 1983; Fisk, 1986; Worring, 1993) are used to define the shape of digital curves, their properties and to find preimage estimates.

A more detailed discussion of topics outlined in this section will now follow. The next section discusses the earlier papers and track the development of digital curve analysis, including both theory and application, as found in the literature.

1.3. Digitization, Images and Digital Curves

An image is a mapping of the world scene in R^2 onto the digital image in I^2 . The color that the pixels in the image are set to is determined by the digitization function. Two common digitization schemes exist (see figure 1.3): OBQ (Object Boundary Quantization) and GIQ (Grid Intersection Quantization). OBQ will turn on any pixel that is in the object's boundary in the scene whereas GIQ will turn on the pixels nearest the object's boundary crossing over either a vertical or horizontal grid line (Dorst and Smeulders, 1984; Freeman, 1970). As noted by Rosenfeld (1974) OBQ can be used to digitize objects in the scene and GIQ digitizes curves. A curve cannot be digitized via OBQ since a curve does not have an "inside" to test whether a pixel's grid point is in it. GIQ, though, can digitize objects by digitizing their boundaries and filling them in. For the purpose of digital curve analysis an object in the image can be digitized using OBQ and the object's boundary detected using a boundary-searching algorithm or the boundary can be generated directly with GIQ. It has been shown that results for one digitization technique are applicable to other digitization schemes with modest changes (Dorst and Smeulders, 1987).

(Interestingly, these digitization schemes that are assumed by most researchers do not seem to reflect the real digitization process. During imaging pixels are set "on" based on the luminescence of the region in the scene that corresponds to the pixel in the image. Nevertheless, as noted above results from GIQ can be adapted to other schemes including, presumably, the real digitization process.)

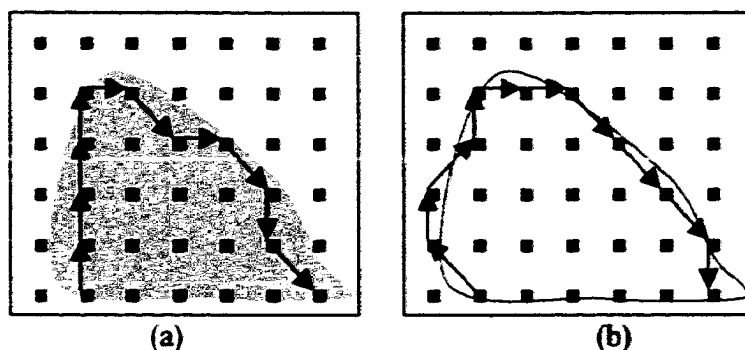


Figure 1.3. (a) Digitization under OBQ (b) Digitization under GIO.

The digital curve exists as a set of (x,y) coordinates where each coordinate corresponds to a pixel in the image. (The main algorithms presented in this research will deal mainly with binary images). This set of pixels is the digital curve. The curve's connectivity is not directly available from the coding of the curve since the pixel set is unordered. Freeman (1970) described a technique for encoding connected digital curves by the connectivity inherent in the curve's pixels – a system now known as Freeman chaincoding or, simply, chaincoding. He suggested that a connected digital curve could be described by its first pixel and by the curve's connectivity from that pixel along the curve until the last pixel is reached. Thus from the second pixel onward the pixel's location is described by its relative location from the previous point on the curve. He noted that on the standard square grid every pixel has only eight immediate neighbors: 2 horizontal, 2 vertical and 4 diagonal (see Figure 1.4). We can thus describe a curve by its first pixel and a sequence of those 8 directions. Thus the path taken from its first point to its last point by following each point to its neighbor defines a curve.

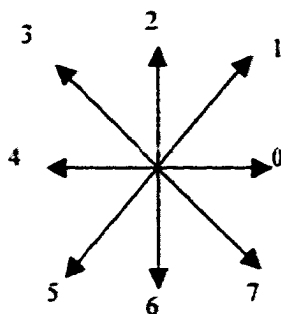


Figure 1.4. The 8 Freemanchain values correspond to the 8 immediate neighbors each pixel has on the grid.

A digital curve is 8-connected if a path exists from each point to every other point as a sequence of the 8 freeman codes. A curve is 4-connected if that path is a sequence of the even freeman codes, i.e. only the horizontal or vertical directions to the exclusion of diagonal directions. Thus, a 4-connected curve contains only even codes. Digital curves define the boundaries (perimeters) and therefore shape of a solid two dimensional object. Simple algorithms exist to find the digital curve that bounds a digital image. These algorithms can be set to find the 4 or 8-connected boundaries in either the clockwise or counter clockwise directions. (Rosenfeld and Kak, 1982)

An important distinction exists between a digital object and a digital curve. Whereas an object in the image is a two-dimensional object and has an interior, a digital curve that represents an object's boundary does not have an interior. Furthermore, in the case of a digital line it is only one-dimensional (and in more complex curves is of fractal dimension.) Also, a digital curve has a minimal digital thickness of one pixel whereas a real curve has minimal real thickness 0. This led some to refer to digital curves as "thin solid objects" (Rosenfeld, 1974).

1.3.1 Reconstruction of a Digital Curve: Digital Line Segments

The actual real curve that digitized to the digital curve in an image is known as the digital curve's preimage. A major problem that researchers in digital curve analysis have studied is the determination of the preimage from the digital curve. Given an object boundary from a digitized image, an infinite number of world scene curves can possibly digitize to the given digital curve comprising part of the boundary. For example, the digitization of a line segment, a circular or elliptical arc with low curvature may yield identical digital curves. To complicate matters even further, the number of real line segments that digitize to any digital line segment is infinite. Thus, reversing the digitization process is impossible without more information.

To make the reconstruction problem tractable, the assumed primitive type of the preimage must be specified. The results, whether as a specific estimate or a range in the parameter space of valid preimages, will then be valid for that primitive type. The primitive types studied are line segments and circular/elliptical arcs. Other more complex curves such as nonlinear (that is, non-circular) curves can be interpreted as connected lines and circular/elliptical arcs since these arcs can be locally approximated by line segments and circular/elliptical arcs. Since locally complex curves can be categorized as either linear or circular, the simple curve approximation follows directly from the observation.

1.3.1.1 The Chord and Compact Chord Properties

Freeman (1970) observed some regular properties of digital line segments. Digital line segments are digital curves that resulted from the digitization of a line segment. He suggested that a digital line's chain codes satisfy the following three conditions:

- 1) They consist of at most two codes which differ by one modulo eight.
- 2) One of the two values of the chain code, called the principal direction, occurs singly.
- 3) The principal direction is as uniformly spaced as possible.

Consider the following example of a chain code in figure 1.5. The chain includes only 0's and 1's satisfying the first criteria. Note also that "1" occurs singly. The run of 0's is either 3 or 4 as recorded in level 1. Again "3" and "4" are the only codes and they differ by 1 and "4" occurs singly. The pattern repeats itself again in level 2.

Chaincode:

0000100010001000010001000010001000010001000010001000010001000100

Pattern 1: 4 3 3 4 3 4 3 4 3 3 4 3 4 3 3

Pattern 2: 2 1 1 2 1 2

Pattern 3: 2 1

Figure 1.5. The multi-level pattern found in a digital line's chaincode.

Rosenfeld (1974) proved these properties. He elaborated on the rather general third property and showed that the length of the "run" of codes in the nonprincipal direction can have only two lengths, which are consecutive integers. He continued to show that a certain element of recursion is found within a digital line. Thus, runs of runs, i.e. runs that are composed of both a principal and nonprincipal direction, similarly occur so that one occurs singly and the other can have two lengths that are consecutive integers. Similarly, such patterns exist for higher levels of runs. Rosenfeld thus succeeded in showing that a certain regularity exists within a digital line segment on a low level and continues to be present on a larger scale throughout the entire digital line. These regular properties were proven as a result of the chord property which states that every point on a digital line must lie less than distance 1 from any line drawn between two points of the digital line (see figure 1.6a on next page.)

The chord property has relevance to us because of the regular properties of digital line segments that logically follow from it. A tighter condition that digital line segments

must satisfy, the compact chord property (figure 1.6b, next page), was proven by Sharaiha and Garat (1993). This property states that a line segment connecting any two points in the curve must lie less than 4-distance 1 from the line segments that connect successive pixels in the curve.

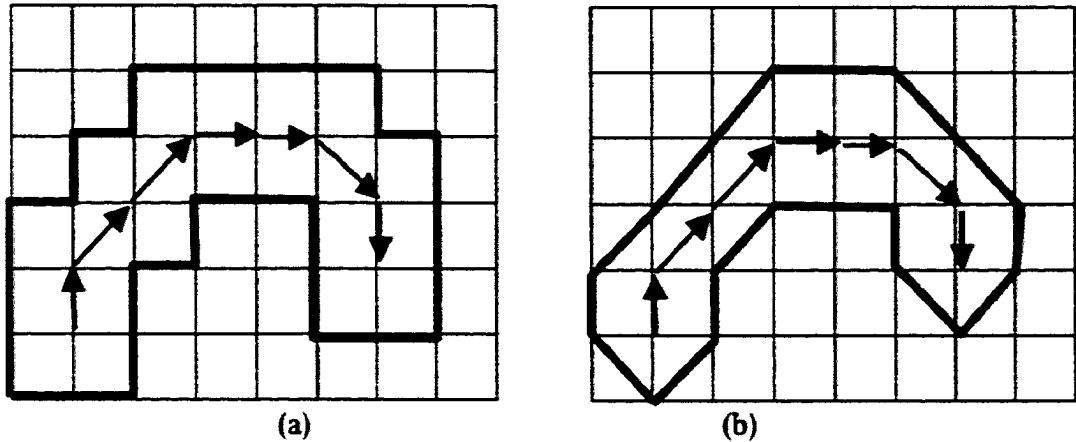


Figure 1.6. Illustrations of the chord properties. (a) To satisfy the basic chord property, all line segments connecting any two pixels must lie within city distance of one from some pixel. (b) To satisfy the compact chord property, the constraint is even tighter.

1.3.1.2. Linear Decomposition of Digital Curves and Characterization

Sharaiha and Christofides (1993) use the compact chord property of the previous section to construct an optimal algorithm for the straight segment approximation of digital arcs. These researchers computed in $O(n^2)$ time a minimum decomposition of any digital arc into piecewise linear segments. A minimum decomposition of a digital curve is a linear interpretation of that curve constructed of a minimal number of line segments.

First determine the linearity of components within the digital arc to yield a graph that connects each pixel to other pixels of the digital arc that can be represented as a single digital line. Then find the minimum path within the graph from the first to last pixel of the digital arc.

They further show that there may exist multiple sets of linear decompositions that are minimal and indicate how to detect the sets that satisfy certain criteria. For example, one can select the set that has digital line lengths with minimal standard deviation, maximal length of individual segments and so on. Under certain circumstances, such as noisy data, it may be preferable to relax the compact chord property thus interpreting marginally nonlinear curves as linear.

This result extends the results of Dorst and Smeulders (1991) who computed a decomposition of a digital curve into piecewise linear segments in $O(n)$. (This algorithm finds one decomposition – not necessarily the minimal decomposition.) They also computed in $O(n)$ time the characterization of each digital line segment. This is an extension of earlier results where Dorst and Smeulders (1984) proved that a digital line can be characterized by a 4-tuple of integers. The later work showed that this 4-tuple characterization represents the domain of the digital line (i.e. the set of all line segments that digitize to the digital line segment) and thus can be called a faithful characterization.

Elsewhere, Dorst and Smeulders (1987) detail the different types of characterizations of digital lines. The simplest characterization is the chaincode count (equal to the pixel_count – 1). Thus, all digital lines of n pixels would be characterized identically. Clearly, such a characterization retains some of the information found in the

digital curve while totally neglecting to record details such as slope and y-intercept. They identify the following characterizations of digital lines (figure 1.7):

- a) (n) where n is the number of chain codes in the digital line.
- b) (n_e, n_o) where n_e is the number of even chain codes and n_o is the number of odd chain codes in the digital line.
- c) (n_e, n_o, n_c) where n_e and n_o are as defined above and n_c is the number of occurrences of consecutive unequal chain code (i.e. "knight moves").
- d) (n, p, q, s) where n is the length of the digital string (in pixels), q is its smallest periodicity (i.e. the size of the smallest repetitive sequence), p / q is the simplest (i.e. irreducible) slope that is consistent with the string and s is the "phase shift, i.e. the number of pixels within the basic repetitive pattern where the digital string begins.

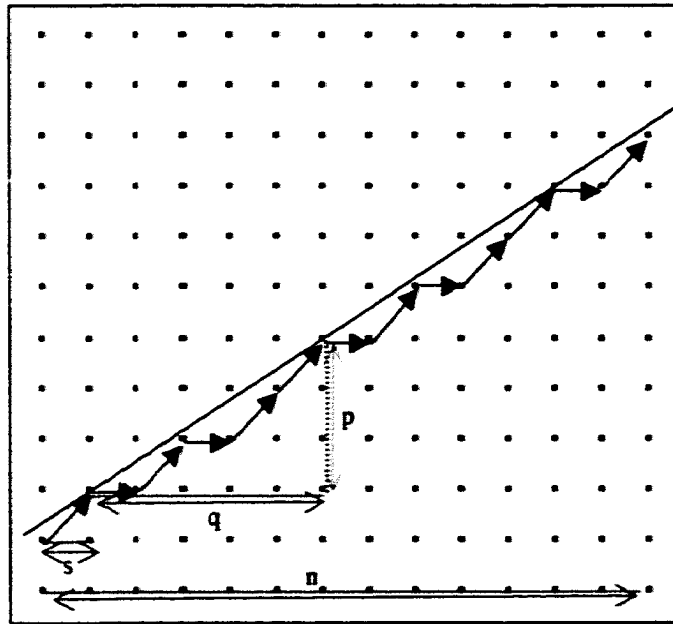


Figure 1.7. The (n,p,q,s) characterization of a digital line.

Each characterization can be used to find an optimal estimate of a property of the digital line (e.g., line length). With finer granularity (i.e. more pixels in the curve per inch) greater accuracy can be achieved. However, for some characterizations, accuracy of the estimate does not improve regardless of the sampling density. For more descriptive characterizations, a strong correlation between the sampling density and the accuracy of the estimate exists such that with infinite granularity the estimate is exact.

The best characterization for digital line segments is (n,p,q,s) . By using this quadruple, the characterization of the estimates can be the most accurate. Using the parameters (n, p, q, s) , Dorst and Smeulders (1984) derive mathematical expressions that define the domain, the set of all real line segments that digitize to the digital curve. The domain of a digital curve is the “best” inverse of the digitization function $f_{\text{deg}}: \mathbb{R}^2 \rightarrow \mathbb{I}^2$ that is used for the conversion of the scene to the digital image. In other words, during

digitization a smooth curve undergoes a transformation from being a continuous curve with infinite granularity to become a discrete digital curve with finite and thus coarser granularity (depending on the camera's resolution). Clearly, in the condensation of scene data a loss of information takes place where the fine detail found in the smooth real curve is irretrievably lost. Thus, the coarse digital curve represents not one real curve but an infinite number of real curves – all the real curves that digitize to the digital curve. Therefore, the optimal result for “undigitizing” a digital curve is to find the domain, the entire set of real curves whose digitizations are the digital curve in question.

1.3.1.3. The Domain and Estimates of a Digital Line

The domain for a digital line with respect to line segments is the set of line segments that digitize to the digital line. This domain can be defined on the (m, b) plane, i.e. m and b are the slope and y-intercept of the line satisfying the equation $y = mx + b$. Each digital line segment is the digitization of lines defined by a region in the (m, b) plane.

With no loss of generality, we assume the digital line begins at the origin. We thus limit the range of b values to between 0 and 1. That is, each digital line segment can be defined by its first pixel and the chain code for the remaining pixels. We set the start pixel to pixel (1,1) and thus the b value must be between 0 and 1. Also, we study chain codes consisting of 0's and 1's only. m is thus limited to the neighborhood of the range between 0 and 1. Consider the domain for chain codes of lengths one through seven. Dorst and Smeulders (1984) showed that this area of the plane can be divided into regions so that

each region is representative of a different chaincode. The domain is subdivided into smaller areas as the chaincode length increases. This indicates that the domain can be computed incrementally. Based on this, the 4-tuple characterization of a digital line and thus its domain are computable in $O(n)$.

Considering the regions thus formed by the domains for chain codes of lengths one through seven, a symmetry seems to exist along the center of the region implying that only half the region is necessary to find the domain of all remaining chain codes. It would thus appear that the b range can be limited to $(0, .5)$ and the m range to $(0, 1)$. Such symmetries within chain codes should appear with circles as well as all more complex curves since the grid is symmetric. What is fascinating about the symmetries found in the domain is that it does not seem to be based on horizontal or vertical symmetry. In other words, one would expect that a "0022" chaincode has the 'inverse' chaincode as "6644" but we also find that "0000" is symmetric to "1111" even though "0000" is horizontal whereas "1111" is diagonal.

In a later paper, Dorst and Smeulders (1986) used their results for finding the BLUE, or best linear unbiased estimates. Using an accurate characterization of the digital curve BLUEs are optimal estimates with respect to a given parameter where the estimate is Best (i.e., has minimal mean square error), Linear (i.e., being an average over "observations") and is Unbiased (i.e. is consistently accurate over all data). The paradigm developed first finds a faithful characterization (that is a bijection with the domain) of a digital curve and utilizes that result to find an estimate. This is particularly appealing in that it takes all the possible preimage line segments into consideration when determining

an estimate. Significantly, the BLUE estimated for slope and y-intercept is realizable, i.e. would digitize to the digital curve.

Unrealizable estimates have the practical advantage over estimates computed from the domain in that, historically, domain algorithms work only with “clean” data (i.e. without noise) while many unrealizable estimates are based on probabilistic techniques and find the “best” fit based on all (including erroneous) data points. Regarding these approaches, see for example Worring (1995) and Yeun and Feng (1996).

The results of Dorst and Smeulders (1984) brought attention to the fact the any estimate should consider all the possible preimages of a digital curve. The estimate can then be set to the preimage of greatest probability as defined by some probabilistic function. Previously, research in finding a linear estimate for a digital line did not take the entire domain into consideration. This earlier research was divided into two camps: finding estimates that may be unrealizable (i.e. that approximate the preimage although they may not actually digitize to the digital curve) and finding realizable estimates. As Koplowitz (1987) points out, for instance, an estimate found by least squares fit may not digitize to the digital curve – although it will probably be similar to the preimage.

Some researchers employed mathematical approaches at finding estimates, as did Dorst and Smeulders (1984), while others used geometric methods. For example, O'Rourke (1981) formulated the domain computation problem as an intersection of $2k$ half spaces in the (m, b) plane. Specifically, each pixel in the digital line constrains the real curves that are preimage candidates by requiring that the preimage pass within distance $\frac{1}{2}$ of the pixel's center point. This constraint is the intersection of two half-

planes in the (m,b) plane. The intersection of $2n$ half planes for n pixels is computable in $O(n)$ time.

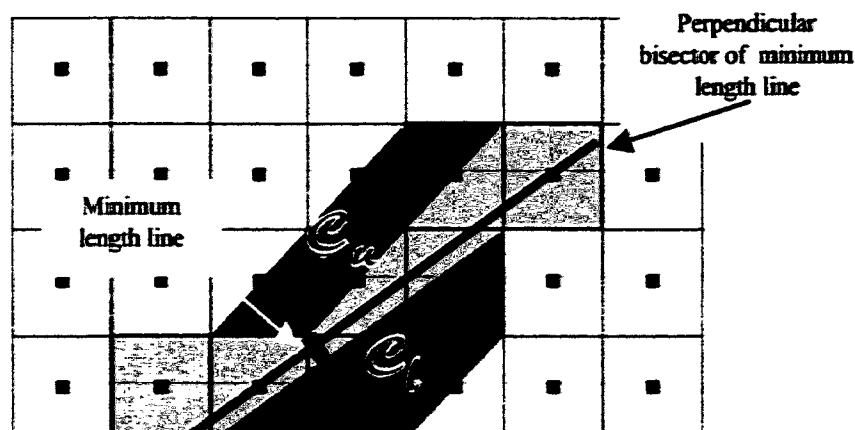


Figure 1.8. The upper and lower convex hulls, C_u and C_l , respectively, that any preimage must not intersect. The perpendicular bisector of the minimum length line yields an accurate estimate of the preimage.

Koplowitz (1987) uses a geometric approach to find an estimate of the preimage line segment (See figure 1.8.). He demonstrates that convex hulls can be drawn above and below the digital curve thus creating a passing area through which any preimage line segment must pass. Any preimage line must not intersect either convex hull and must intersect the smallest line connecting those hulls, l_m . He suggests as an estimate the perpendicular bisector of l_m . Since every preimage must intersect l_m in order to minimize the maximum displacement along l_m he chose as an estimate the bisector of l_m and to minimize the maximum angular error the bisector must be perpendicular. This technique is also computable in $O(n)$.

In addition to finding estimates, Koplowitz (1987) uses his technique to find maximal digital line segments of digitized straight lines as was done later by Dorst and Smeulders (1991) and Sharaiha and Christofides (1993) (see above). As long as the upper and lower convex hulls are nonintersecting, the digital arc is a digital line; otherwise, it is not. By assuming that digital circles are locally linear he uses this approach for linear smoothing and the determination of the tangent to a point on the digital circle. The algorithm used is intuitive in that it finds a passing area through which all preimage lines must traverse and uses that passing area to find an estimate.

Yuan and Suen (1995), like Koplowitz, use a passing area algorithm and developed an $O(n)$ time domain computing algorithm. (Interestingly, they make no mention of Koplowitz or of convex hulls although their technique appears to be conceptually identical.) They proceed to discuss the error factor present when decomposing digital curves into piecewise linear segments. In the case where the digital arc is circular some error will obviously occur when approximating a circle with line segments. This error, though, will decrease with increased radius size since then the curve's curvature more closely reflects that of a straight line which has zero curvature. They show that even when reconstructing connected digital line segments two line segments can be reconstructed as one (maximal line). Specifically, the error increases with decreasing angle between the two preimage line segments and increases with decreasing line segment length. For example, two digital line segments where one has length 10 and the other is infinite that have a 15 degree angle between them, a distance along the digital curve of 33.1 pixels may be interpreted as a single digital line segment.

The number of pixels in the curve that may be interpreted as a single line increases with a decrease in angle between the two line segments and a decrease in line length.

1.3.2 Reconstruction of a Digital Curve: Digital Circular Arcs

Earlier papers in the field focused largely on digital lines and attempts were made to extend those results to more complex curves. Much of this effort was placed in finding estimates, characterizations, domains and shape analysis of circular digital arcs. Some researchers believe no solution exists to some of these problems. For example, Dorst and Smeulders (1987) state that they do not believe a faithful characterization of digital arcs exists. This pessimism was similarly expressed by Sauer (1993) who wrote “it is not possible to use well-known properties of digital straight lines for the recognition of digital circles.” However, Goldberg and Robinson (1998) show a very clear relationship exists between lines and circles and more complex curves. Subsequently, researchers have successfully discovered many significant results with respect to circular arcs.

Much of the research has been focused on finding accurate estimates for digital circles and circular arcs. A digital circle's parameter is defined in 3-space (x, y, r) where x and y are the circle's center on the grid and r is the circle's radius. Although not found in the recent literature, a digital circular arc may have two additional parameters: start and end angle. Clearly, this adds a level of complexity that one may wish to delay until after the (x, y, r) parameters are computed.

Two approaches are found for parameter estimation: curve fitting and Hough transform (Yuen and Feng, 1996). Curve fitting techniques include least squares and

recursive methods that are dependent on the initial position. With Hough transforms peaks in the parameter space (which are interpreted as the voted values of points on the digital arc) are interpreted as circle centers. The accuracy of this method improves with increasing the size of the quantization bins in the parameter space – which in turn increases computational complexity.

In this context, Yuen and Feng (1996) suggest a novel method for unbiased and consistent estimating of a digital arc's parameter. Their algorithm is better than recursive methods since it is independent of an initial position. They essentially take the average over all data points of the digital arc and find the average center over those points. The computational complexity of this method is $O(n)$.

Another focus of attention is the domain computation determining the set of realizable preimages. In (x,y,r) parameter space The domain of digital circles is the intersection of cones at digital points inside the circle and cone complements of digital points outside the circle, as in Worring (1993). This intersection yields the domain since the center of a circle that digitizes to a point on the digital circle must lie within distance r , the circle radius, of the digital point (or of a point on the plane that under the digitization scheme would digitize to that digital point on the grid). Similarly, the circle must not digitize to points not in the digital circle; hence the intersection of cones' complements of points not in the digital circle requirement. compute whether the domain is empty; if it is, the digital arc is not circular. Damaschke (1995) presents a different $O(n)$ algorithm to determine if the domain is empty as well as finding elements within the domain.

Bhattacharya (1988) computed the minimum and maximum radii found in the domain. Fisk (1986) used circular separability and Voronoi diagrams to find the set of all circle centers over all radii. This is the 2D projection of the 3D-cone intersection mentioned earlier. Circular separability is the study of how two point sets can be separated by a circle. In our case this would be the circular separability of the upper and lower boundaries of the digital curve. If the curve is separable it is a digital circular arc. This is the computational geometry analogue of the passing area discussed earlier for digital lines. The generalized Voronoi diagram of two point sets, S_P and S_Q , can be used to find the circular separability of those two sets. With digital circular arcs S_P is the set of digital points in the digital curve and S_Q is the set of neighboring points in the curve's background. This analysis assumes that the digital curve is a boundary of a digital object where background pixels can be found using a boundary-searching algorithm as detailed in (Worring, 1995), citing (Rosenfeld, 1982). In case that the curve is not a digital boundary similar simple techniques exist to replace S_Q with either the right or left neighboring background pixels. The generalized Voronoi diagram is the projection onto the (x,y) plane of all circle centers – without distinguishing between circle centers for different radii.

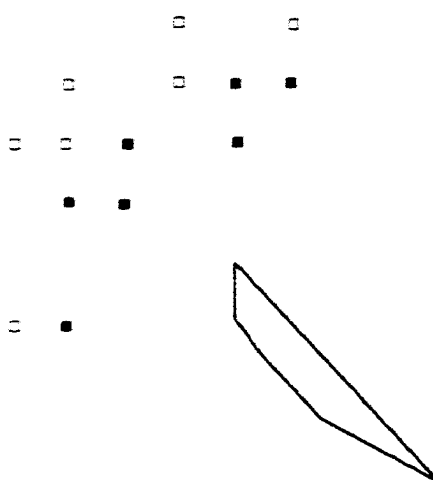


Figure 1.9. The circular arc domain of a digital curve under all radii is the generalized Voronoi polygon of the inner and outer points of that curve. The inner and outer points are shown as filled and empty squares, respectively.

Chattopadhyay (1994) computes the domain of a complete digital circle for a fixed radius. This is equivalent to computing the cross-section of the cone intersection mentioned earlier at a specific radius. The algorithm first divides the complete digital circle into monotonic quarter circles and computes their domains. The quarter circle domains may, under some circumstances, be combined to form the complete circle's domain. See Sauer (1993) who also computes the domain for complete circles using the minimum circle-covering algorithm from Meggido (1983) but as noted by Damaschke (1995), techniques for complete circle computations are not easily extended to (partial) circular arcs.

Worring (1993, 1995) did the most recent and most sweeping analysis of digital circles. He developed an algorithm that computes the domain of a digital circular arc. He

showed that circular interpretations of a digital curve are constrained by different points of S_P and S_Q at varying radii. The ordered set of points is called the r -sequence. Each point corresponds to an edge in the domain. Specifically, when the radius is smaller than the minimum radius the r -sequence is null. At the minimal radius the r -sequence will contain 3 members and this set will add or lose one member at a time until the radius exceeds the largest radius circle interpretation of the digital arc. At that point the r -sequence is again null.

Thus, at specific radius values the r -sequence changes. Knowledge of the r -sequences and the radii at which they change is equivalent to the digital arc domain under all radii. He finds these changes in the r -sequence by first dividing the generalized Voronoi diagram into disjunct s -regions. Each point in S_P and S_Q has a related s -Region. The s -Region of point, s , in S_P (S_Q) is the intersection of the generalized Voronoi diagram and furthest (closest) point Voronoi polygon $V(S_P, s)$ ($V(s, S_Q)$). By following the vertices and edges in the s -Regions the algorithm he has computes the complete domain of a digital circular arc. The efficiency of this algorithm is $O(n^2)$.

Another interesting result Worring found is the limitations inherent in reconstructing a digital circle. He showed that a digital curve may be a digital line segment or a digital circular arc. As the radius increases or the length of the digital curve decreases a digital circle may also be a digital line segment. Worring quantified these findings with the MRR (Maximum Realizable Radius). The $MRR(n)$ for a digital arc is the maximum realizable radius of a circle for a digital circle of length n . In other words, a large radius may have an equivalent digital representation as a digital line. For a smaller radius a digital arc of length n may have no linear interpretation. The maximum realizable

radius for a digital arc of length n , thus is the maximum preimage radius that when digitized with length n would have no linear interpretation. Clearly, as the length on n increases the $MRR(n)$ also increases.

Consider figure 1.10 (next page). Note that the domain presented is a subset of the generalized Voronoi diagram of figure 1.9, which is the union of all such areas. The changes in the relevant point set, as the radius increases, is shown in (b). As the radius increases from 16.8 to 31.6 the new relevant points are introduced and eliminated. Relevant points determine the domain for the radii for which they are valid.

This chapter presented an overview of the issues involved in digital curve analysis and the main problems that researchers investigated in properly understanding the digital domain. The literature was reviewed from a historical perspective, leading to more recent results that are specifically relevant to this thesis. The subsequent chapters present algorithms that exhibit linear time requirements for computing the digital domain of various digital curves. The next chapter concentrates on that of digital line segments. Clearly, such a treatment has been well studied. However, the main contribution of the next chapter is that it sets up a nomenclature and an approach for analyzing the computation of domains of all types of digital curves and as such can be considered as presenting the “definitions and notations” for the rest of the thesis.

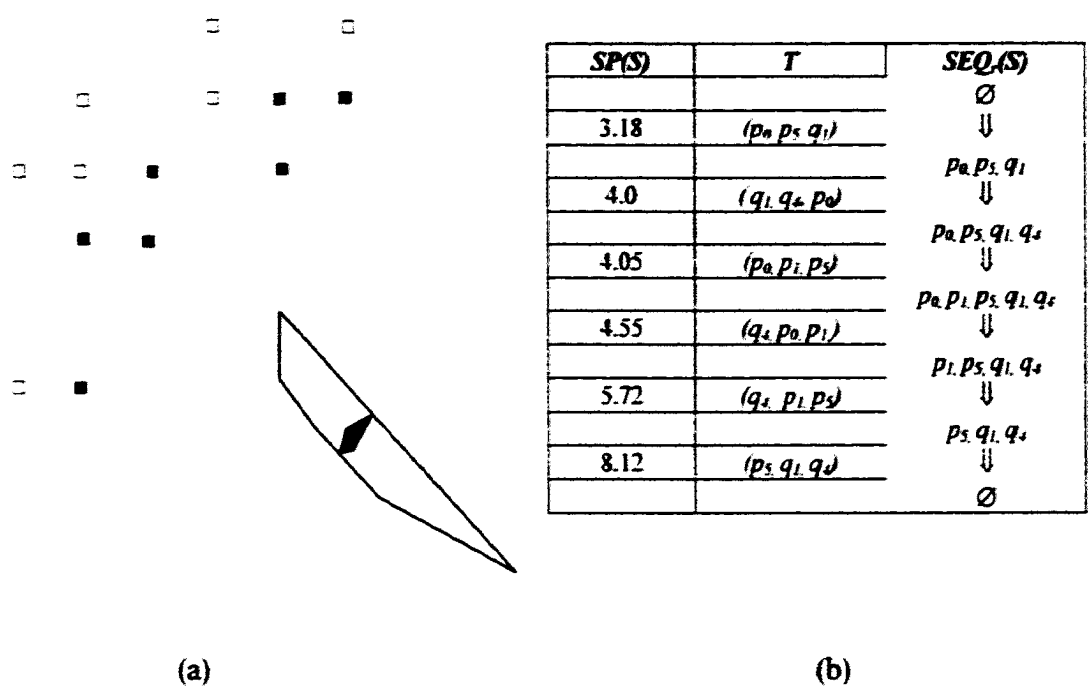


Figure 1.10. The domain of fixed radius of a digital curve
(a) and the changes in the relevant point set (b).

Chapter Two:

The Passing Area and Domain of a Digital Line Segment

Abstract

Describing the geometry comprising the boundary of an object is an essential task for computer vision systems. A fundamental part of this task is the determination of maximal line segments that digitize to the object boundary. This chapter concerns itself with an efficient manner in recovering the set of all possible lines (and segments) that could digitize to a part of an object boundary.

Koplowitz (1987) demonstrated that convex hulls could be used to determine an estimate of the underlying line segment that has been digitized to a digital line segment in $O(n)$. However, his method did not yield enough information to obtain the minimal passing area of the digital line segment, the sub-pixel area of the grid where real lines must be located. Using the parameters (n, p, q, s) , Dorst and Smeulders (1984) derive mathematical expressions that define the domain, the set of all real line segments that digitize to the digital curve, characterize digital lines (1987) and based on this, decompose a digital curve into piecewise linear segments in $O(n)$. However, closed-form approaches are highly susceptible to noise and are not easily extendible to scenes that involve partial occlusion of the object boundary.

We define an extension of the convex hull, the infinite hull, that is used to determine the digital line segment's minimal passing area in $O(n)$. This passing area can, in turn, be used in constant time to compute the digital line segment's domain, the set of all possible underlying lines segments in (m, b) parameter space that digitize to the digital line defined. There are three direct advantages of the infinite hull method. Firstly, the domain of a disconnected digital line may also be computed from the passing area in

constant time. Second, the approach may be extended to more complex curves for which an infinite hull may be defined. Third, this yields a highly parallelizable algorithm.

2. Introduction

The digitization process converts visual data from the world scene into a digital image. Each object in the image is bounded by digital curves that define the object's shape. This chapter considers object boundaries that are (partially) comprised of line segments and addresses the following problem: Given a connected digital line segment represented by a list of connected pixels (figure 2.1), find the set of all line segments that digitize to that digital curve. This set is known as the *domain* (Dorst and Smeulders, 1984).

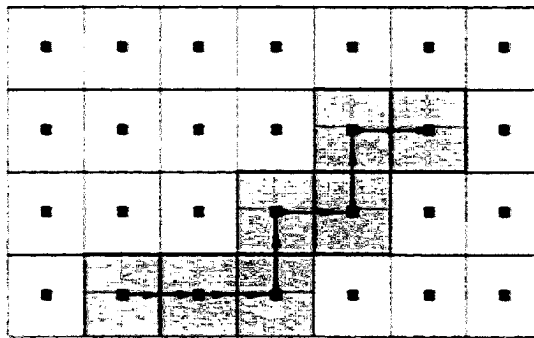


Figure 2.1. A digital curve consisting of the pixels shaded in gray on the grid. 4-connectivity is considered and the first pixel is in the bottom left side.

Each line segment that is a member of the domain may be the original line segment that exists in the world scene. The actual line segment from the world scene is referred to as the *preimage line segment*, or *preimage*. The digitization process causes the inevitable loss of detail and thereby preventing the recovery of the preimage from the

digital curve. The best possible reversal of digitization is the recovery of the domain; this led some to consider the domain as the pseudo-inverse of the digital curve (Dorst and Smeulders, 1987). Koplowitz (1987) introduced a geometric approach to finding an estimate of the preimage of a digital line segment. He demonstrated that the area on the grid through which any preimage line segment must pass (termed the *passing area*) is bounded above and below the digital line segment by convex hulls (figure 2.2).

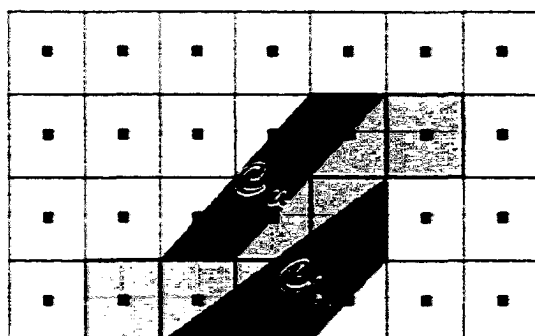


Figure 2.2. The convex hull of the upper and lower boundaries superimposed on the grid. The preimage of a digital line segment cannot intersect the upper (C_u) or lower (C_l) convex hulls.

Consider figure 2.3. Any preimage line cannot intersect either the upper, C_u , or lower convex hull, C_l . However, it must intersect the smallest line segment connecting the two hulls, I_m . Koplowitz suggested as an estimate, the perpendicular bisector of I_m , labeled I_c , since it minimizes the maximum translational displacement along I_m . Similarly, it minimizes the maximum angular dispersion from I_m . Although the general solution for computing the convex hull of a point set is $O(n \log(n))$, the connectivity of the pixels and, consequently, the ordering of vertices along the upper and convex hulls, facilitates an $O(n)$ solution.

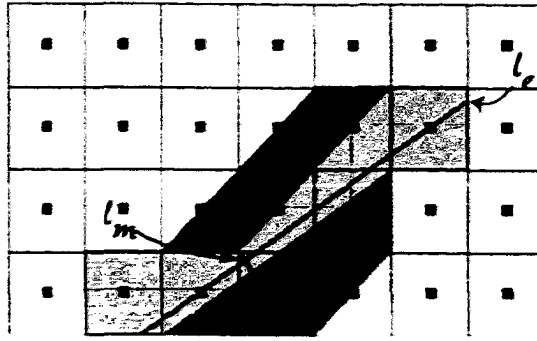


Figure 2.3. l_m is the minimum length line between the upper and lower convex hulls (shaded). l_e , the perpendicular bisector of l_m , yields an estimate of the preimage that minimizes the maximum displacement error along l_m as well as minimizing the maximum angular error from l_m .

A major advantage of Koplowitz's technique is that the computed estimate will digitize to the actual digital line segment found in the image data. This feature is not necessarily present in other estimation techniques such as the least squares method. For example, consider a horizontal digital line segment with an arbitrarily long run in the horizontal direction followed by a single pixel in the vertical direction (figure 2.4, next page). The least squares method will estimate this digital curve as a line segment (figure 2.4a) in the interest of minimizing the distance between the estimate and the vast majority of the pixels in the horizontal digital line segment. Clearly, though, a line segment will not digitize to a slightly non-horizontal digital line segment (figure 2.4b). We refer to real-world estimates of a digital curve that, upon digitization, yield that digital curve as *realizable* estimates.

The convex hull method proposed by Koplowitz for finding an estimate yields only a single realizable estimate. It will be shown that an extension of the convex hull method can be used to determine the entire set of realizable estimates: the *domain*. (This information would enable more accurate estimates, from a statistical point of view, to the preimage. See Dorst and Smeulders, 1986). In order to accomplish this, the correspondence between the passing area and the domain of all realizable estimates represented in (m,b) parameter space is first demonstrated. Then, the minimum passing area of the digital line segment can be used to construct the domain (section 2.2). In section 2.2 the infinite hull is defined. It is shown that the minimal passing area can be determined by first computing the infinite hull of the upper and lower digital boundaries. In section 2.3, the algorithm to accomplish this is demonstrated. Based on the methodology presented, extensions of the methods to other related problem areas (such as discontinuous digital lines and parallelization) are discussed (section 2.4).

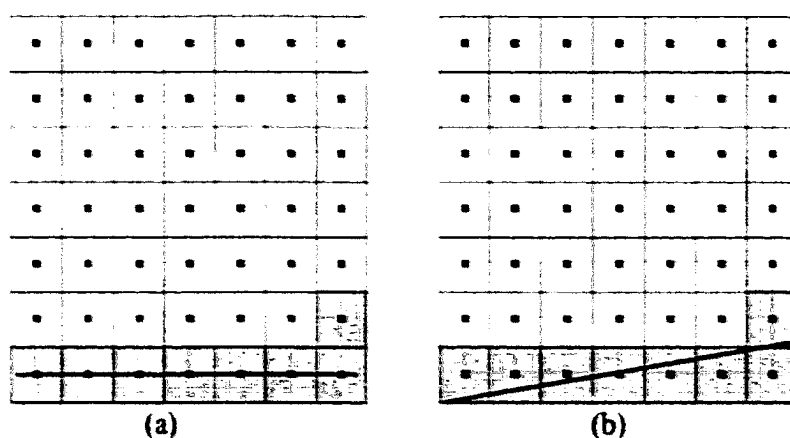


Figure 2.4. Unrealizable (a) and realizable (b) estimates of a digital line segment. The unrealizable estimate does not intersect every pixel of the

digital curve; however, the realizable estimate would digitize to the given digital curve.

2.1 The Duality of the Passing Area and the Domain

The passing area of a digital line segment is the area on the grid in which every realizable estimate of that digital line segment must be found. The passing area thus represents the union of all line segments that digitize to the digital line segment. Each such line segment can be represented in (m,b) parameter space as a point. That point in (m,b) space represents the (infinite) line that traverses the passing area of which that line segment is a portion. The union of all such points is the digital line segment's domain. We thus have a bijection between lines traversing the passing area and points in the domain. Consequently, the domain is directly computable from the passing area.

We now discuss the correspondence that exists between line segments in the passing area and points in the domain. Three possibilities exist for the way a line l_e passes through the passing area. Each of those possibilities results in a different correspondence for that line segment with the domain. Those correspondences and their rationale are now discussed. The three cases are as follows:

Case A: The line, l_e , intersects a perimeter edge of the passing area. (figure 2.5, next page).

A line segment incident to an edge on the passing area boundary will yield the (m,b) value of that edge. Since the m (slope) or b (y-intercept) values of edges on the passing area boundary are extremum, this line segment directly corresponds to a vertex on the domain boundary.

Case B: The line, l_e , intersects a perimeter vertex, v , without being incident to the edges $(e_1 \text{ or } e_2)$ (figure 2.6, next page).

l_e must have a slope (m) and y-intercept (b) within the range of m and b of its two incident edges, e_1 and e_2 ; otherwise the line segment would intersect either or both of those edges. We will now show that the relationships amongst the m values and amongst the b values are linear.

Let the two incident edges $\{e_1, e_2\}$ of vertex, v , and the line segment, l_e , have (m,b) values of (m_{e1}, b_{e1}) , (m_{e2}, b_{e2}) and (m_l, b_l) , respectively. Let $m_l = m_{e1} + k(m_{e2} - m_{e1})$ where $0 \leq k \leq 1$, since m_l must be in the range between m_{e1} and m_{e2} . We solve for b_l : $b_l = b_{e1} + k(b_{e2} - b_{e1})$. The relationship is clearly linear in parameter k . Let $p_1 = (m_{e1}, b_{e1})$ and $p_2 = (m_{e2}, b_{e2})$, both of which are vertices on the domain boundary as explained in case (a). The linear edge $\overline{p_1 p_2}$ on the domain boundary, therefore, represents the set of line segments that pass through the passing area and intersect vertex, v . The domain, in (m, b) space, thus includes all points along the line

segments that are bounded by points that represent edges on the passing area boundary.

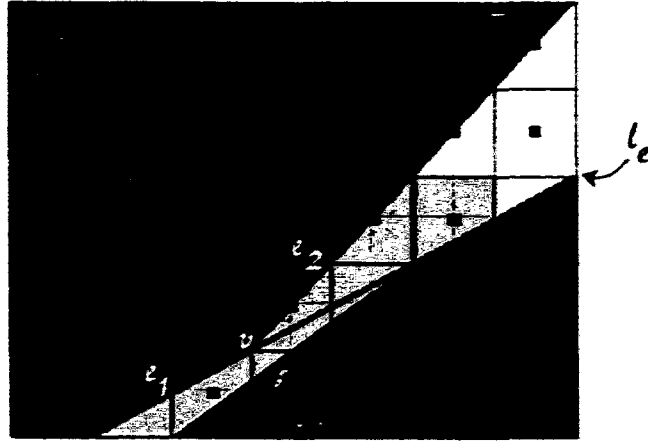


Figure 2.5. Line, l_e , is incident to edge e_1 . In the domain, the parameter space point corresponding to l_e is the (m, b) value of e_1 .

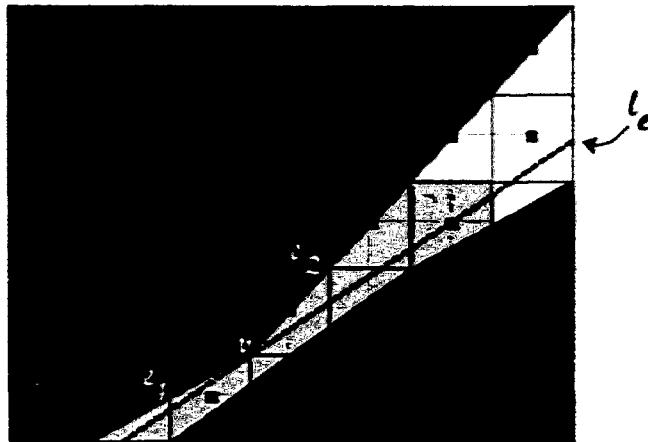


Figure 2.6 Line, l_e , is incident to vertex, v . In the domain, the parameter space point corresponding to l_e will lie on the line segment defined by points p_1 and p_2 , where $p_1 = (m_{e1}, b_{e1})$ and $p_2 = (m_{e2}, b_{e2})$.

Case C: The line segment intersects neither a perimeter edge nor a perimeter vertex. This case corresponds to a point within the domain area (figure 2.7).

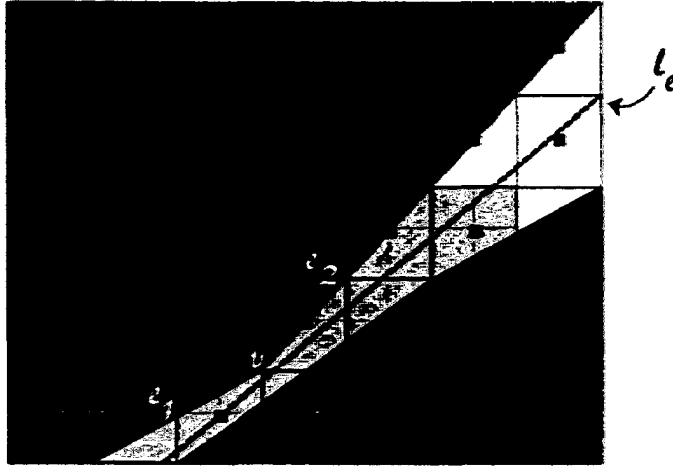


Figure 2.7 Line, l_e , is not incident to edges or vertices. It corresponds to a point in the interior of the domain boundary.

Every point in the domain area not on the domain's boundary, represents the line segments not considered in cases (1) and (2), and are thus the line segments that are incident to neither passing area vertices nor edges, but rather lie in their ranges. This directly corresponds to the lines found in the interior of the passing area.

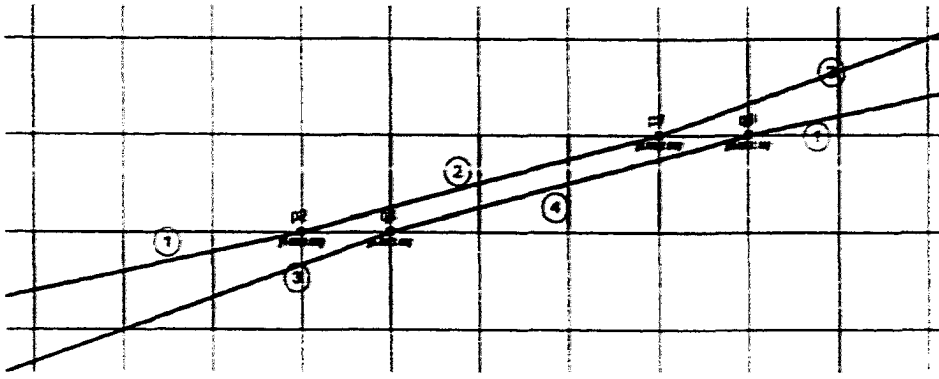


Figure 2.8. Line segments (labeled 2 and 4) and line segment extensions (labeled 1 and 3) define the passing area of the digital line segment, shaded in gray.

In summary, this bijection yields an interesting relationship between the boundaries of the passing area and the domain. Consider the digital line segment in figure 2.8. The line segment extensions labeled 1 fall on the same line (and similarly with those labeled 3) and are obtained by removing the line segment portions $\overline{p_2q_3}$ from the line connecting the portions labeled 1 (and $\overline{q_1p_2}$ from the line connecting the portions labeled 3). The remaining portions of these lines are termed *extensions*. Then, the passing area of the digital curve presented is bounded by the line segments (labeled 2 and 4) and line segment extensions (labeled 1 and 3). Based on the above demonstrated correspondence, the line segments and vertices on the passing area boundary correspond exactly to vertices and line segments on the domain's boundary, respectively (figure 2.9).

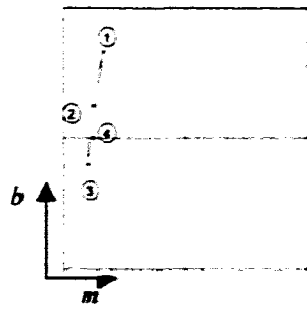


Figure 2.9. The vertices of the domain boundary correspond to the line segments and extensions of the passing area. The numbers labeled are those from the passing area boundary in figure 2.8.

2.2 The Digital Curve's Constraints on the Passing Area

The Koplowitz algorithm that computes the upper and lower convex hull results in a reduced passing area without resulting in the minimal passing area: for example, compare figure 2.10 with figure 2.8. Whereas his algorithm involved the constraints on the passing area that are imposed independently by the upper and lower hulls in the digital curve, the minimal passing area must also consider the constraints imposed by both hulls collectively (and independently). This section introduces the infinite hull as a geometric structure that can be used to compute the minimal passing area.

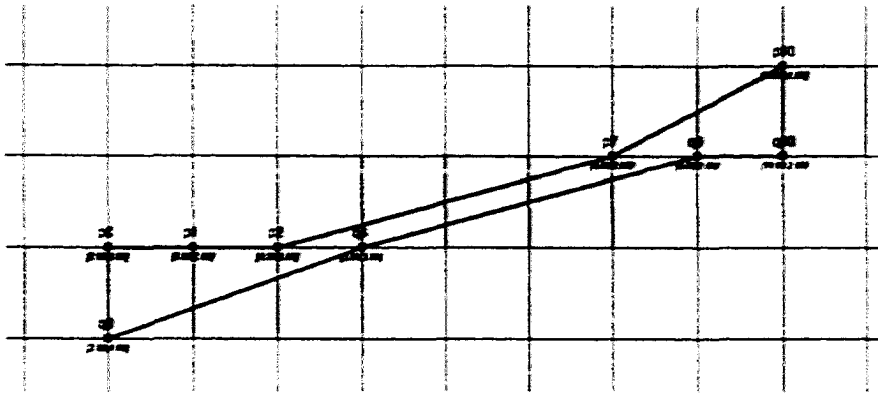


Figure 2.10. A non-minimal passing area obtained by the convex hull algorithm.

The digital curve places limitations on the line segment that can digitize to that digital curve. The basic constraint, as imposed by the individual pixels in the grid, is that the line segment must intersect each pixel in the curve and must not intersect any pixel not in the digital curve. We can thus construct a non-minimal passing area, as a first approximation, the *truncated pixel boundary*, illustrated in figure 2.11. The pixel boundaries of the first and last pixel of the digital curve are not included since the preimage line segment is not confined within those pixels' boundaries; the preimage must simply intersect the pixel somewhere, as constrained by their neighboring pixels' boundaries. Clearly, the line segment must be found within the upper and lower boundaries.

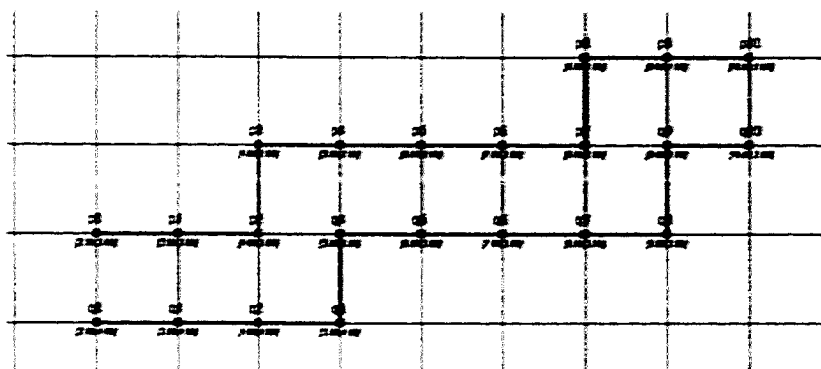


Figure 2.11. The truncated pixel boundary. This boundary provides the first set of constraints used in the determination of the passing area.

Removing from the truncated pixel boundary the boundaries' upper and lower convex hulls can further minimize the passing area, as Koplowitz proposed (figure 2.10). This passing area reduction is a result of the constraints that the upper and lower boundaries place on the passing area. Since the line segment must lie below the vertices of the upper boundary, it must lie under the convex hull of those vertices (see figures 2.12a-b). Similarly, the line must lie above the lower boundary's convex hull. This constraint results in the strict convex hull reduction of the passing area (figure 2.12c).

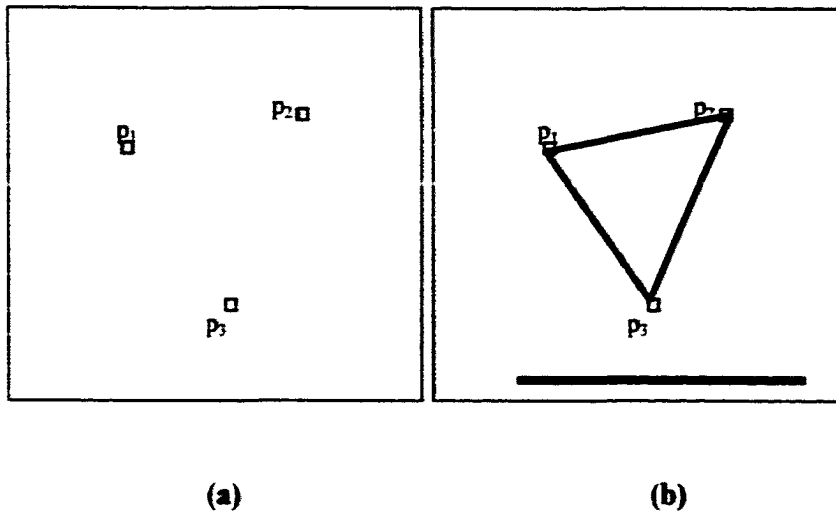


Figure 2.12a-b. A line segment that is below a set of points

(a) must lie beneath the convex hull of those points (b).

The Koplowitz algorithm considered the constraints provided by the upper (lower) convex hulls of the boundary to reduce the sub-pixel passing area of the digital curve. However, the algorithm did not consider the added constraints on the passing area as imposed by simultaneously considering the vertices from both the upper and lower convex hulls. This consideration is now addressed.

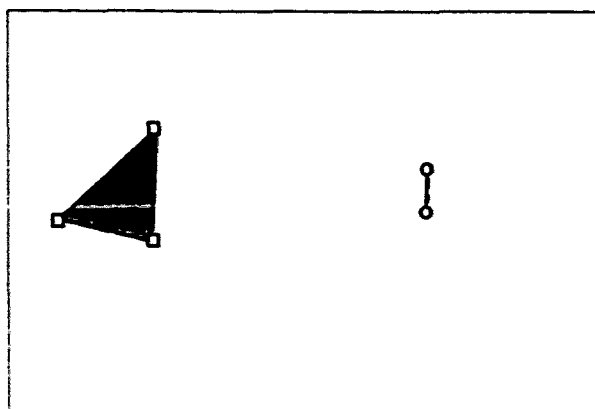


Figure 2.12c. If the line is below a set of points and above another point set, the line must be found between the convex hulls of the point sets.

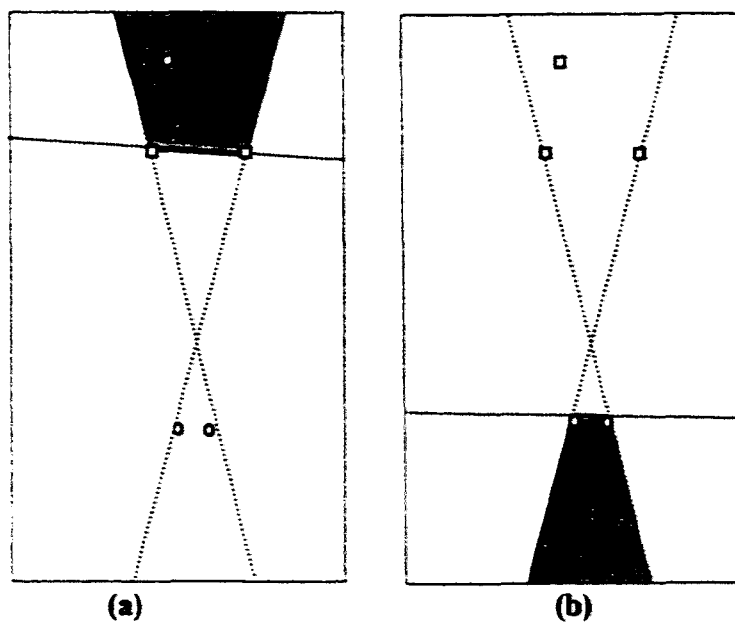


Figure 2.13. Shaded in gray (a) is $\text{hull}_{\text{sep}}(S_1, S_2)$ where S_1 include the three square points on the left and S_2 include the two round points on the right of each image. In (b), $\text{hull}_{\text{sep}}(S_2, S_1)$ is shown. A line that is found between the two point sets must not intersect either hull. The half spaces

boundaries that determine the separation hull are shown as dotted lines.

Definition: The *separation hull* denoted by $hull_{sep}(S_1, S_2)$ of two point sets S_1 and S_2 is the minimum intersection of half-spaces where each half space includes every point in S_1 and excludes every point in S_2 (figure 2.13).

Definition: The *infinite hull* denoted by $hull_{infinite}(S_1, S_2)$ of two point sets S_1 and S_2 , is $hull_{sep}(S_1, S_2) \cup hull_{sep}(S_2, S_1)$ (figure 2.14).

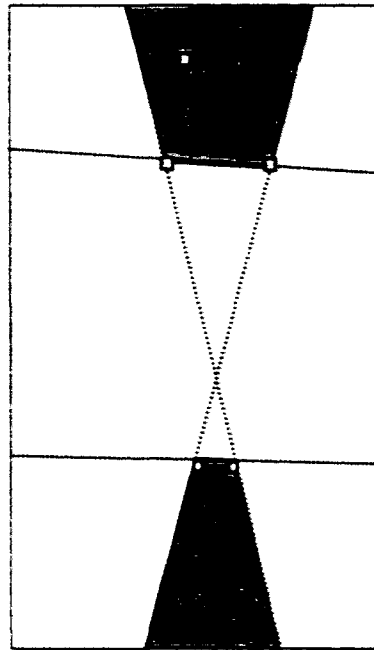


Figure 2.14. The infinite hull of S_1 and S_2 is shaded in gray. It is the union of the two separation hulls shown in figure 2.13. The half spaces boundaries that determine the separation hull are shown as dotted lines.

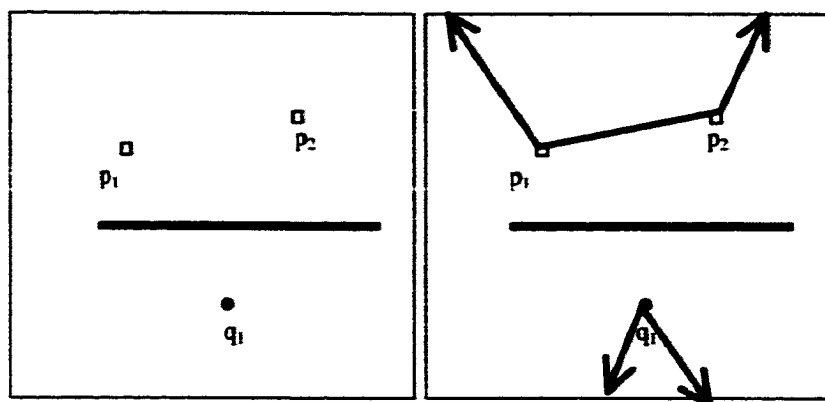


Figure 2.15. A line segment that is below one point set $\{p_1, p_2\}$ and above another point set $\{q_1\}$ must lie between the upper and lower components of the infinite hull of those two point sets. This additional constraint results in the infinite hull minimization of the passing area that is proposed.

The method that we consider in this paper to determine the passing area of a digital line segment is to compute the infinite hull of S_1 and S_2 , where S_1 is the upper boundary and S_2 is the lower boundary. We refer to this simply as the infinite hull. The minimal passing area is the intersection of the digital curve's area with the infinite hull complement. See figure 2.15 above and compare that with figure 2.8.

In general, the computation of the infinite hull may involve $O(n \log(n))$ computations required to sort the vertices radially from an arbitrary point. However, in our case, the two sets for which we compute the infinite hull are the upper and lower boundary. The connectivity of the boundaries creates an implicit sorting of the vertices, thus, allowing us to determine the infinite hull in $O(n)$ time. This is analogous to the

discussion from section 2.1 where it was pointed out that the general solution for the convex hull of a point set in $O(n \log(n))$ but where the pixel connectivity is known, it is solvable in $O(n)$ time.

Basically, to compute the convex hull of the upper (or lower) boundary each vertex is tested whether it is within the convex hull of its previous and successive vertices along that boundary. If it is found to be within the hull the vertex is deleted. There is no need to test for convexity with respect to other vertices. The ability to determine the convex hull of the entire boundary by testing each vertex along the boundary against its neighbors only, results in the linear time determination of the convex hull.

A similar approach is used to find the infinite hull. The method used to compute the infinite hull is to "connect" the upper and lower boundaries. This upper and lower hull connection is achieved by connecting a line segment extension from head and tail of the upper hull to tail and head of the lower hull, respectively. See figure 2.16 below. The line segment extension defined by two points p_0 and q_{10} (and similarly for p_{10} and q_0) is the infinite line defined by those points where the closed line segment $\overline{p_0q_{10}}$ (or $\overline{p_{10}q_0}$) is removed. The vertices of the upper and lower boundaries are then merged into one list and are referred to in a unified manner as the *joined boundary*.

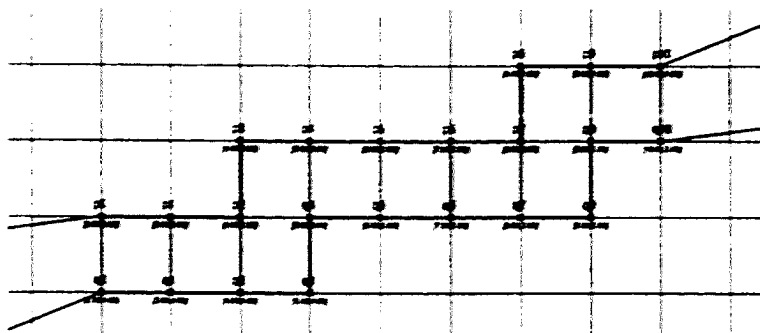


Figure 2.16. The truncated boundary where upper and lower hulls are connected by adding line segment extensions. The minimal passing area can be determined from this information. Note that the extensions through p_{10} and q_0 fall on the same line and similarly with those through p_0 and q_{10} .

Now, each vertex may be tested for inclusion within the infinite hull by considering only its immediate neighbors. The infinite hull of the joined boundary can be obtained with slight modifications from the standard techniques of computing convex hulls. The next section presents a Java implementation of this algorithm that computes the infinite hull of the joined boundary and provides a brief demonstration of the algorithm.

2.3. Infinite Hull Algorithm

This section describes the data structures and the algorithm used to compute the infinite hull of the joined boundary. The algorithm receives a linked list of the vertices of the joined boundary in sequential order as input (upper hull followed by lower hull), and prunes it to a linked list that only contains the vertices on the infinite hull boundary. The algorithm iterates through each vertex on the joined boundary and removes it from the list if that vertex is in the joined boundary's infinite hull, thus maintaining a linear runtime complexity.

For simplicity of discussion this section assumes that the digital curve considered contains three or more pixels. In fact, the algorithm holds for two pixels and in the case of one pixel will appropriately return an empty set representing the infinite plane as the passing area. These two situations are presented in figures at the end of the section (2.3.4).

The implementation is in Java 2, which provides a set of classes rich of features that are used in the data structures and algorithm. The `PassingArea` class is embedded as a subclass of the Java class `LinkedList`. The joined boundary and the infinite hull algorithm, are encapsulated in the `PassingArea` class. The inheritance from the Java `LinkedList` class allows for efficient and simple manipulation of each vertex in the joined boundary. Two classes are defined for the algorithm: `PassingAreaPoint` and `PassingArea` whose purposes are to store and manipulate the vertices and linked list of vertices, respectively. The classes from the Java 2 API (section 2.3.1) as well as the classes and

their member methods and variables that we introduce, are defined below (section 2.3.2) followed by a discussion of the full implementation (section 2.3.3).

2.3.1. Classes From Java 2 and the Methods Used

Class `Point2D` from the `java.awt.geom` package encapsulate the vertices of the joined boundary. The individual vertices are placed into an indexed circularly doubly linked list with header node, encapsulated in the `LinkedList` class from the `java.util` package. The member methods of `LinkedList` that are used are `size()` which returns the number of elements in `LinkedList`, and `clear()` which removes all the `LinkedList` elements.

A vertex from that linked list is tested for inclusion in the infinite hull by testing whether it is to the left or right of the line segment defined by its immediate neighbors, as will be discussed in 2.3.3. That line is stored as a `Line2D` object from the `java.awt` package.

The constructor for the `Line2D` object is: `Line2D.Double(Point2D vertex1, Point2D vertex2)` which sets the `Line2D` object to the line segment defined by the two vertices, `vertex1` and `vertex2`. The member methods of class `Line2D` used is `relativeCCW(Point2D p)` which returns 1 and -1 when `Point2D p` is to the left and right of `Line2D`, respectively.

2.3.2. Classes, Methods and Variables Headings

This section presents the classes and their member variables and methods that are used in the infinite hull computation implementation found in section 2.3.3. Class `PassingArea` encapsulates the joined boundary. As mentioned earlier, the boundary of the digital curve is composed of line segments that meet at vertices along the boundary. Since the vertices totally define the shape of the boundary, the `PassingArea` object is initially set to the ordered sequence of vertices along the joined boundary.

Method `computeInfiniteHull`, a member method of class `PassingArea`, removes from the `PassingArea` object vertices that are not on the infinite hull boundary, with the result that only the vertices on the infinite hull boundary remain. The determination whether a vertex shall be deleted is computed by the `deletePoint` method. The vertices are stored as `PassingAreaPoint` objects.

class `PassingAreaPoint` extends `Point2D`

A `PassingAreaPoint` is deleted if it is found to be within the infinite hull of its immediate neighbors. This will depend if the point considered is on the upper or lower hull. See figure 2.17. For the upper hull, a `PassingAreaPoint` is deleted if its successive `PassingAreaPoint` is to the right of the line defined by its previous `PassingAreaPoint` and itself. For the lower hull the `PassingAreaPoint` is deleted when the next `PassingAreaPoint` is to the left of that line. Therefore, the `deletePoint` method calls the `isLeftHull` method to determine if the vertex is on the upper or lower boundary.

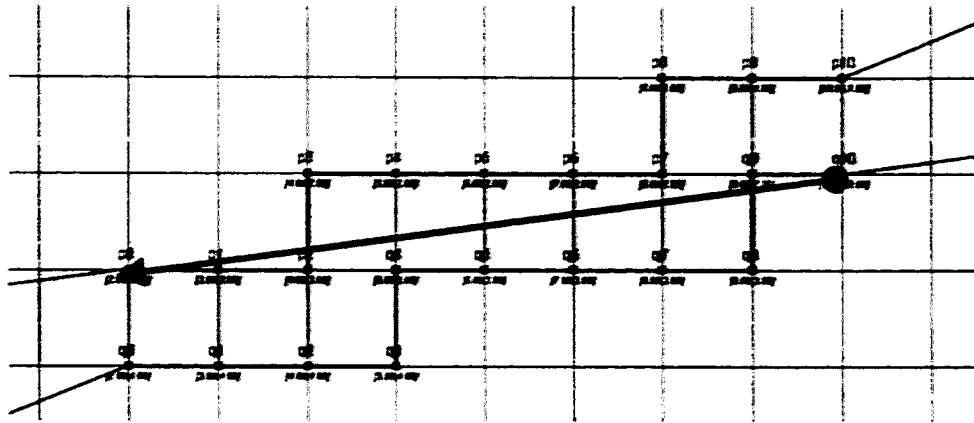


Figure 2.18. p_0 is being considered for deletion. Its immediate neighbors are q_{10} and p_1 , which are on different components of the boundary.

The following is a summary of the methods and variables used to determine the infinite hull.

class PassingArea extends LinkedList

The data structure used to contain the vertices of the joined boundary

boolean isLeftHull(int index)

returns true if the vertex at index is on the upper boundary. An index is a reference to a vertex on the upper hull if it is less than or equal to leftTail.

boolean deletePoint(int index1, int index2, int index3)

returns true if the vertex at index2 is in the convex hull and should be deleted. The determination whether a vertex should be deleted is completely determined by its previous and next vertices in the linked list (referenced by index1 and index3.)

void computeInfiniteHull()

computes the infinite hull of the joined boundary. Iterates through the vertices, calling method deletePoint on each vertex, deleting vertices as necessary.

boolean differentHulls(int index1, int index2)

returns true if the vertex at index1 is on the upper hull and the vertex at index2 is on the lower, or vice versa. An index is determined to be from the upper or lower hull by calling method isLeftHull.

size()

returns the number of elements in the LinkedList

void delete(PassingAreaPoint p)

deletes vertex p from the joined boundary.

int leftTail

the index of the last element in the upper boundary

2.3.3 The Complete Algorithm

The two methods that are the “workhorses” of the infinite hull algorithm are `computeInfiniteHull` and `deletePoint` (figure 2.22. below). Method `computeInfiniteHull` iterates through every `PassingAreaPoint` index on the joined boundary (which includes both the upper and lower boundaries) and calls method `deletePoint` for each `PassingAreaPoint` index. A `PassingAreaPoint` is removed from the joined boundary if `deletePoint` returns true. This occurs when a given vertex is found to be within the infinite hull and hence can be deleted.

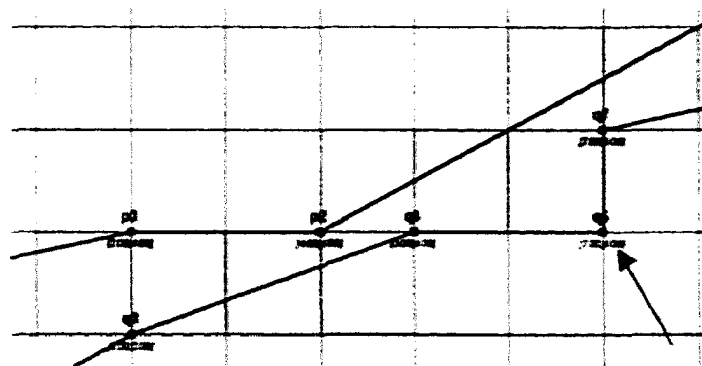


Figure 2.19. q_6 is tested for removal. It is in the convex hull of q_4 and q_7 and thus `deletePoint` will return true.

The `deletePoint` method takes three `PassingAreaPoint` indices as parameters. `PassingAreaPoint` is the class used for the boundary vertices. The three corresponding `PassingAreaPoints` are vertices p_1 , p_2 and p_3 . `deletePoint` determines if `PassingAreaPoint` p_2 , which is in `PassingArea`, should be deleted. This determination is based on the relative location of p_2 with respect to p_1 and p_3 (left of or right of line segment, as discussed in section 4.2. If p_2 is within the infinite hull (the new constraint introduced by this paper) of p_1 and p_3 , p_2 should be deleted.

If a `PassingAreaPoint` is not removed, iteration proceeds with the next `PassingAreaPoint` on the boundary. If the `PassingAreaPoint` is removed, iteration continues with the previous `PassingAreaPoint` because the previous `PassingAreaPoint` has potentially re-entered the infinite hull due to the removal of its successive `PassingAreaPoint` (figures 2.19 above and figures 2.20-21 below.)

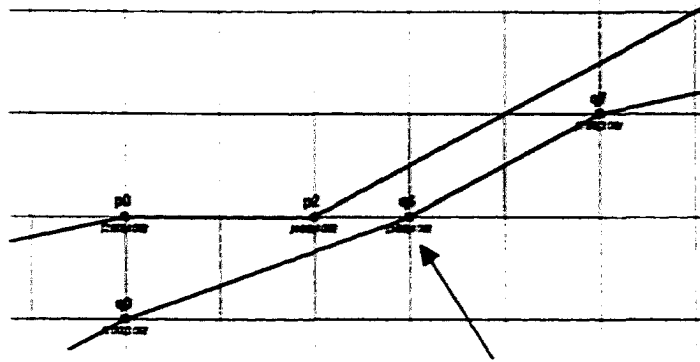


Figure 2.20. After the removal of q_6 , the algorithm backtracks to q_4 . The removal of q_6 causes q_4 to be in the convex hull of q_0 and q_7 . Therefore, `deletePoint` will return true for q_4 .

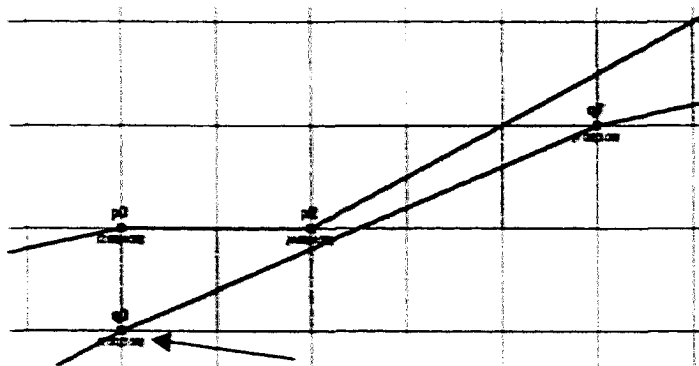


Figure 2.21. After the removal of q_4 , the algorithm backtracks to q_0 , which will be tested with p_2 and q_7 .

The algorithm finds the infinite hull in linear time. Each `PassingAreaPoint` is inspected for infinite once during forward iteration. Upon the deletion of a `PassingAreaPoint` the previous `PassingAreaPoint` must be inspected again. The upper bound on such backward re-inspections after deletion is n , the number of `PassingAreaPoints`. Hence, the upper limit on vertex inspections, both forward and backward, is $2n$; yielding an overall efficiency of $O(n)$.

The following is the code that implements the infinite hull algorithm. The complete implementation is found in the appendix.

```

1.  boolean deletePoint(int index1, int index2, int index3) {
2.
3.      PassingAreaPoint p1 = (PassingAreaPoint) get( index1 );
4.          p2 = (PassingAreaPoint) get( index2 );
5.          p3 = (PassingAreaPoint) get( index3 );
6.
7.      Line2D line = new Line2D.Double( p1, p2 );
8.
9.      // * p2 is deleted if p3 is on line(p1,p2), or p3 is to the right
10.     of that line and p3 is in the upper hull, or p3 is to the left of
11.     that line and p3 is in the lower hull */
12.     return (line.ptLineDist( p3 ) == 0) ||
13.         ((line.relativeCCW( p3 ) == 1) ^
14.         isLeftHull(index3) ^ differentHull(index1, index2));
15. }
16.

```



```

17.
18. void computeInfiniteHull()
19. {
20.     int i = 0, index1, index2, index3;
21.     Point2D p1, p2, p3;
22.     while ( ( i < size() ) && ( leftTail >= 0 ) &&
23.           ( size() - leftTail - 1 >= 1 ) &&
24.           ( size() > 2 ) )
25.         // iterate while there are at least 1 element in each hull and 3
26.         // elements in both hulls
27.         {
28.             index1 = i % size();
29.             index2 = (i + 1) % size();
30.             index3 = (i + 2) % size();

31.
32.             if ( deletePoint(index1, index2, index3) )
33.                 {
34.                     delete( index2 );
35.                     i = (i == 0 ? 0 : i - 1); // decrement
36.                     if (index2 == 0) i = (i == 0 ? 0 : i - 1);

37.                     /* deletion may ripple backwards and the index is decremented. It
38.                     is unnecessary to backtrack before the 1st element since those
39.                     elements will be reached during forward iteration */
40.                 }
41.             else i++; // increment forward
42.         }
43.         if (leftTail < 0 || size() <= 2 || size() - leftTail <= 1) clear();
44.
45.         /* if either hull has no elements or the both hulls together have

```

```

46. less than 3 elements the upper and lower hulls intersect and the
47. passing area is therefore null. Therefore the PassingArea is cleared */
48. ;

```

Figure 2.22. The Java 2 code for the infinite hull algorithm.

The following figures illustrate the passing area's joined boundary before the algorithm commences (figure 2.23) and during iterations (figures 2.24-36). The algorithm iterates through all points in the truncated pixel boundary of the digital curve (figure 2.23). When the algorithm concludes the double hull appears as in figure 2.36. The first point considered for removal is p_1 (figure 2.24), and then p_2 , and so on. After p_{10} , q_0 is considered until q_{10} is reached. Then p_0 is considered, at which point the iteration concludes. In the following figures the individual iterations are shown.

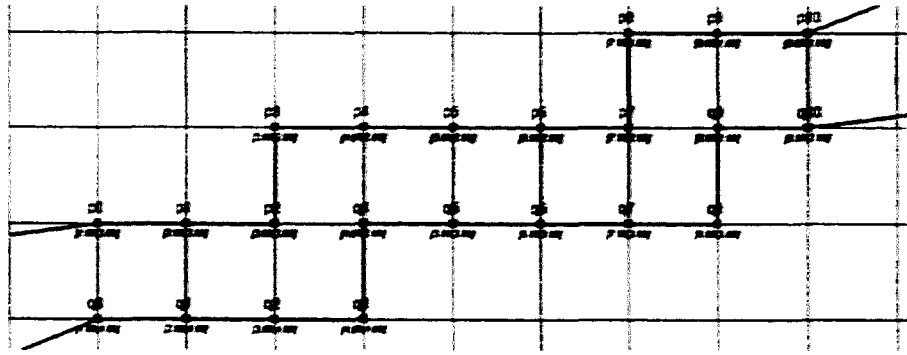


Figure 2.23. The truncated pixel boundary with the upper and lower hulls connected via line segment extensions. p_0 is connected to q_{10} and p_{10} is connected to q_0 via the line segment extensions of line segment p_0, q_{10} and q_0, p_{10} , respectively.

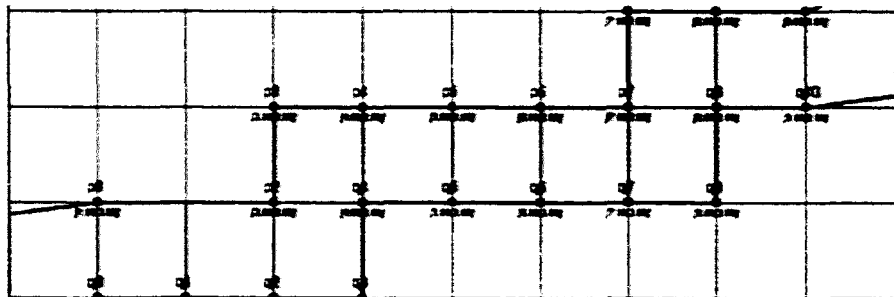


Figure 2.24. p_1 was deleted. After every deletion inspection backtracks to the previous vertex. However, p_1 is the first vertex to be inspected.

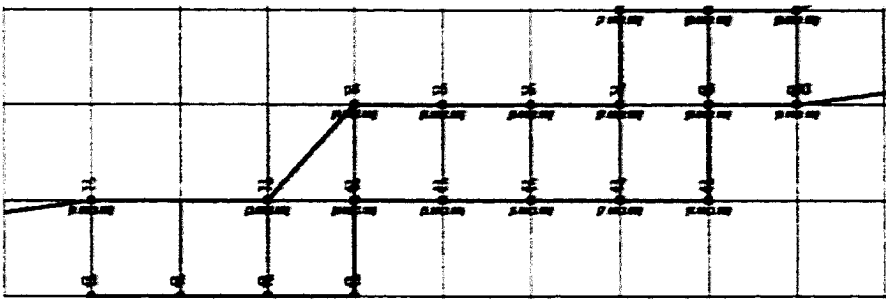


Figure 2.25. p_3 is deleted. Backtracking to p_2 follows the deletion. Since p_2 is not deleted, iteration continues with p_4 .

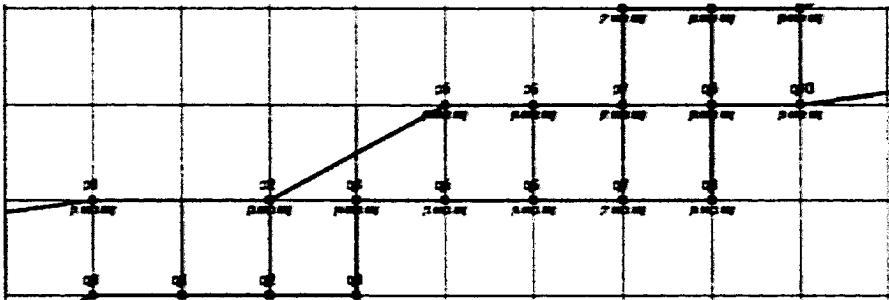


Figure 2.26. p_4 is deleted. Backtracking to p_2 follows the deletion. Since p_2 is not deleted, iteration continues with p_5 .

Further mention of backtracking will be omitted unless the backtracking results with a deletion.

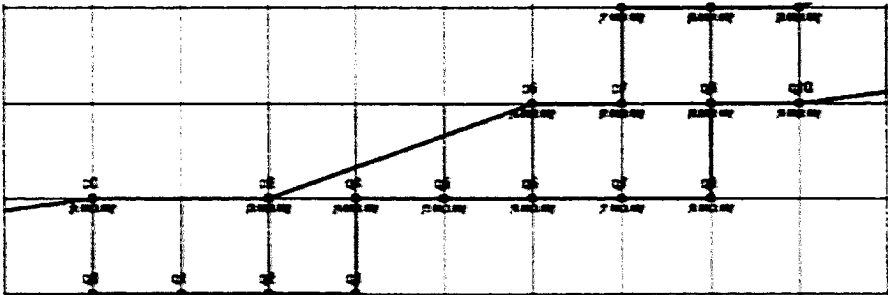


Figure 2.27. p_5 is deleted. p_6 is now considered.

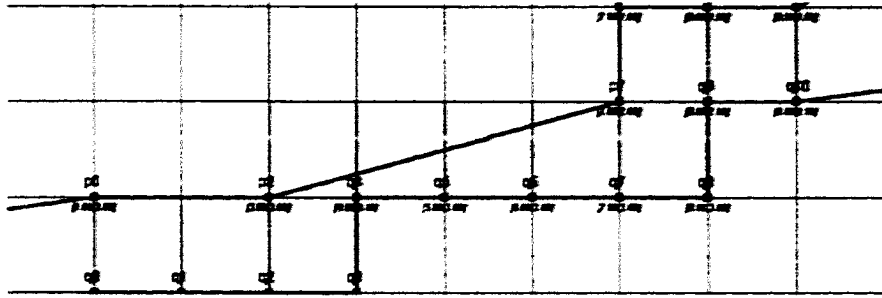


Figure 2.28. p_6 is deleted. Iteration proceeds to p_7 , which is not deleted. Iteration then continues with p_8 .

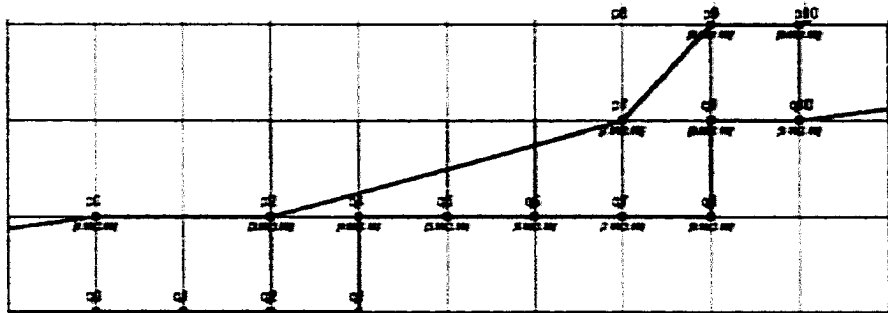


Figure 2.29. p_8 is deleted.

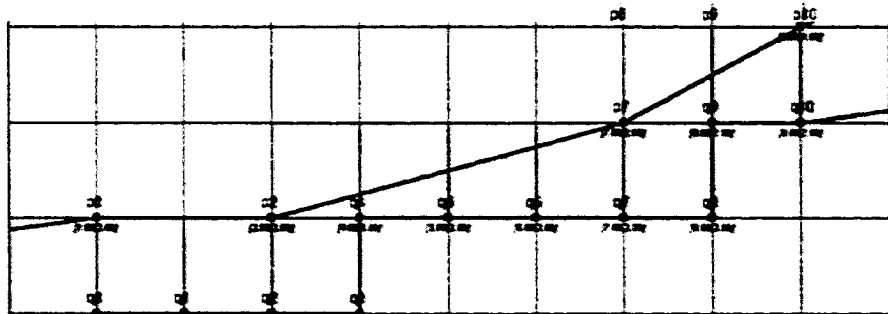


Figure 2.30. p_9 is deleted.

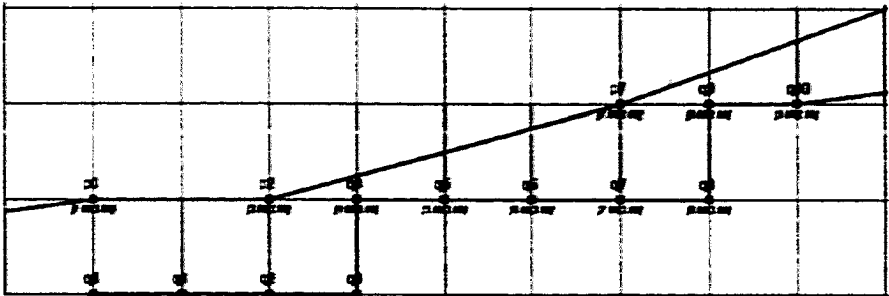


Figure 2.31. p_{10} is deleted. p_{10} is deleted because it lies on the extension of $\overline{p_0q_0}$..

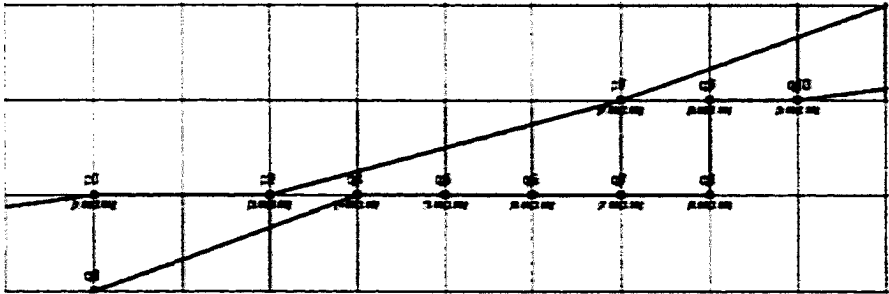


Figure 2.32. q_1, q_2, q_3 are deleted. Backtracking to q_0 .

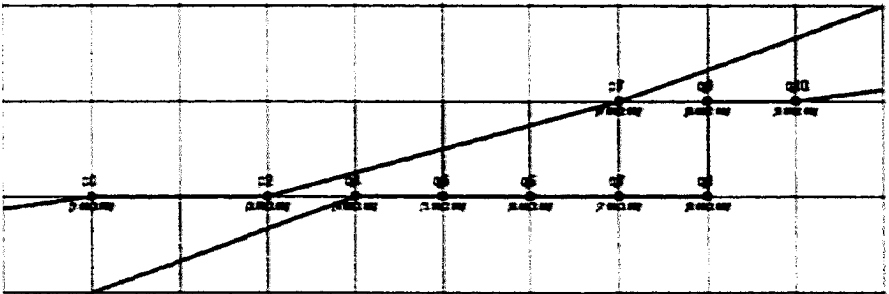


Figure 2.33. q_0 is deleted because it lies on the line extension of $\overline{p_0q_0}$ and is therefore in the infinite hull.

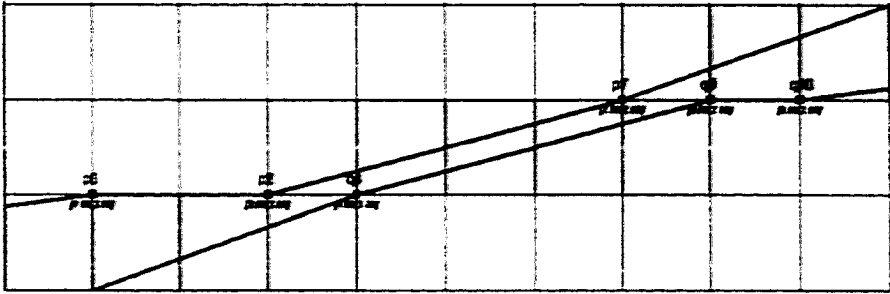


Figure 2.34. q_4 is not deleted and then q_5, q_6, q_7, q_8 are deleted.

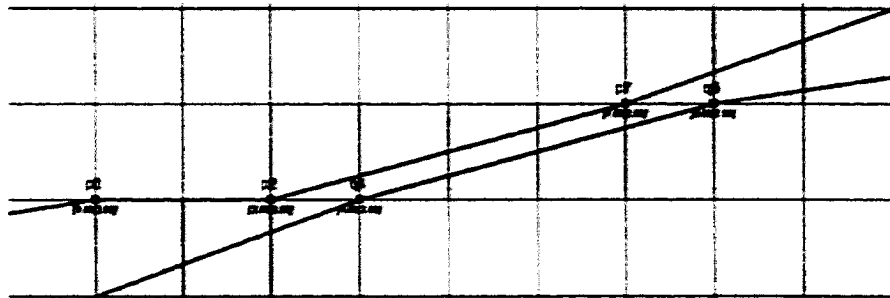


Figure 2.35. q_9 is not deleted but q_{10} is deleted.

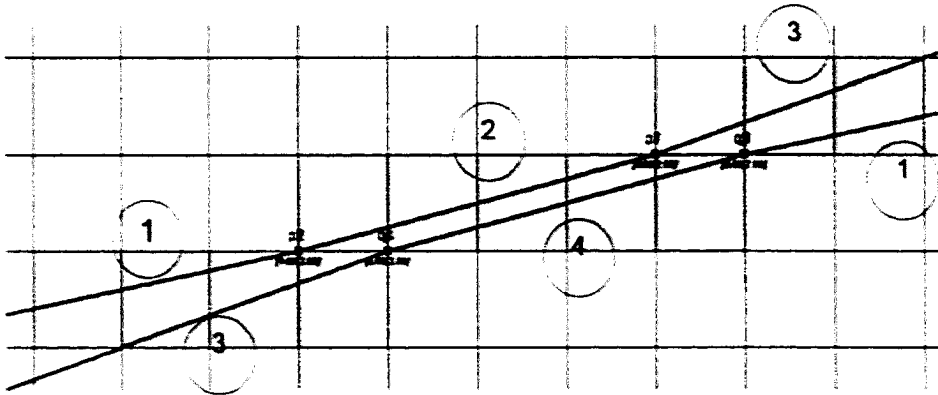


Figure 2.36. Finally, p_0 is deleted because it lies in the infinite hull as determined by the line segment extension of $\overline{q_9 p_2}$.

The algorithm has now terminated. The passing area lies between the upper and lower boundaries. The correspondence between the passing area (figure 2.36) and domain (figure 2.37) is shown with the circled numbers.

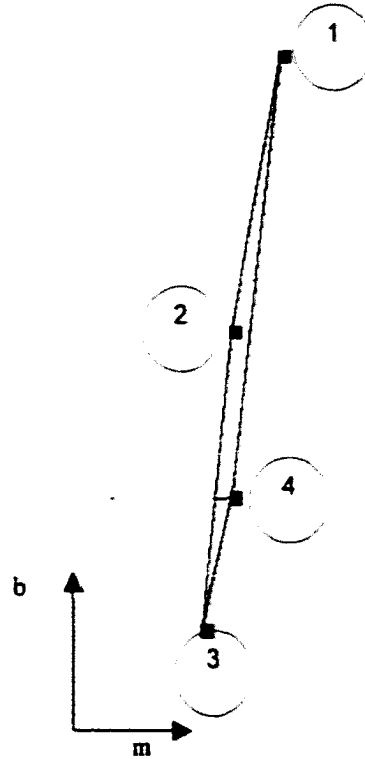


Figure 2.37. The domain in (m, b) space is shown. Each of the four vertices in the domain corresponds to one of the four line segments or line segment extensions in the passing area (figure 4.6). The (m, b) value of the line segment or line segment extension is the vertex location shown.

2.3.4. Additional Cases

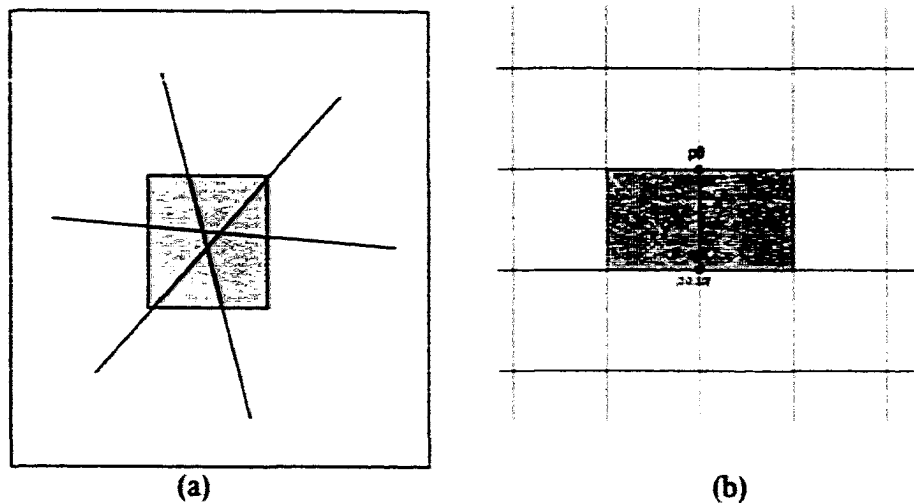


Figure 2.38. The passing area of a single pixel digital curve is the entire plane (a) since the union of all lines that intersect the pixel is the plane (a). A two pixel digital curve has a two vertex passing area that is the digital curve's joined boundary (b).

As noted above (2.3) the algorithm described assumes a digital curve containing at least 3 pixels. Where the digital curve contains a single pixel the passing area is the entire plane as shown in figure 2.38a. Therefore, no passing area boundary exists. Where the curve contains two pixels the passing area is constrained by the two vertices that specify the joined boundary as shown in figure 2.38b.

2.4. Extensions

This section investigates extensions (section 2.4.1) and further directions (section 2.4.2) of the algorithm and underlying methodology, introduced in this chapter.

2.4.1 Applications Of The Algorithm

This section explores two direct application of the algorithm in the subsequent two subsections. 2.4.1.1 deals with disconnected line segment components and 2.4.1.2 deals with the inherent parallelism of this algorithm.

2.4.1.1. Disconnected Line Segment Components

The algorithm discussed in the previous section may be used to compute the passing area and domain of a digital line segment where some pixel information is lost. This results in a disconnected digital line segment where the connectivity of the individual components is known. Where each 4-connected component contains at least 2 pixels the infinite hulls of the upper and lower boundaries of all of the components (figure 2.39) shall be computed using the algorithm of section 2.2 without modification. Figure 2.40 illustrates the resultant infinite hull and figure 2.41 shows the corresponding domain.

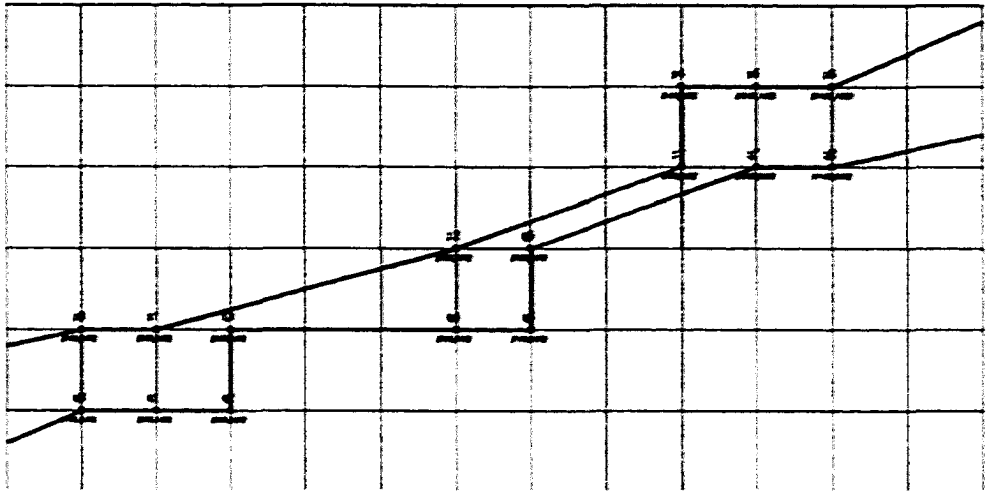


Figure 2.39. The vertices of the disconnected curve are connected to form an upper and lower boundary of the entire curve. The passing area is then computed using the double hull algorithm described above.

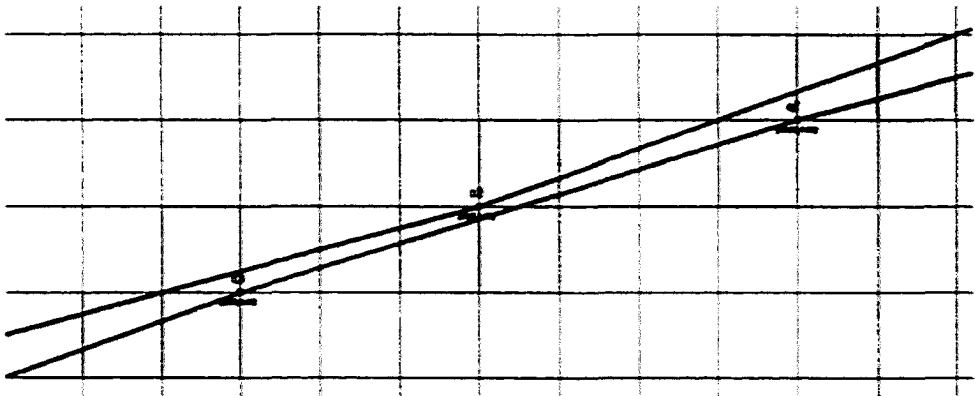


Figure 2.40. The infinite hull of the disconnected line segment

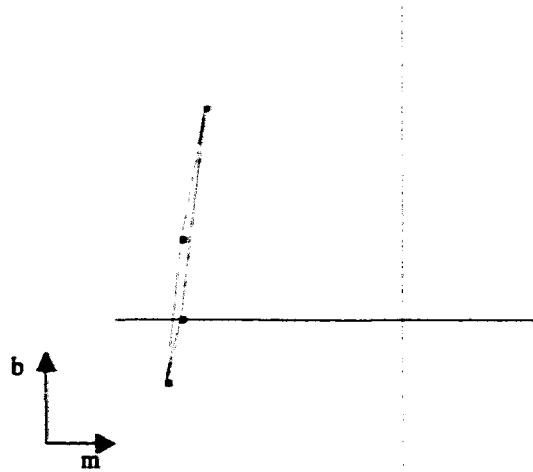


Figure 2.41. The domain of the disconnected line segment shown in figure 2.39

2.4.1.2 . Inherent Parallelism of the Algorithm

The computation of the convex hulls of each disconnected component can be accomplished separately using the methods discussed in section 2.2, and thus can be executed in parallel. Each component can then be connected together head to tail identically to the connections between components shown in figure 2.39. For example, in figure 2.42, point p_1 of component 1 is linked by an edge to point p_0 of component 2 and p_0 of component 2 is linked to p_0 of component 3 and p_3 of component 3 is linked to q_0 of component 1 and so on.

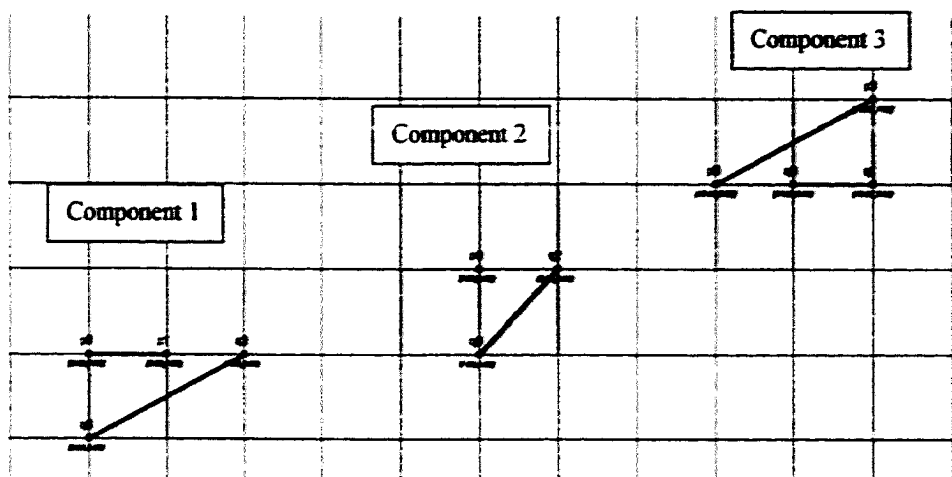


Figure 2.42. The infinite hull of each component may be computed separately. Subsequently, each component may be connected to form the complete upper and lower boundaries from which the infinite hull of the entire digital curve is computed as in figure 2.40. Only a vertex that has an incident edge that connects that vertex to another component need be inspected in this final stage.

Assuming m components of equal size with a total of n vertices, the convex hull of each component can be computed in $2n/m$ iterations. Solving in parallel, with m processors, each processor may compute the infinite hull of one component. As shown in (Dorst and Smeulders, 1984), the domain contains a maximum of 4 vertices. The passing area, then, must also have four vertices, as discussed in section 2.2. Then, each processor may compute the infinite hull of two components, with an upper bound of 16 iterations. The number of processors used, for this step is $m/2$ because each processor is now

processing 2 components. Each successive step uses half the processors of the previous step until a single processor computes the infinite hull. The efficiency of such a parallel algorithm is $2n/m + 16 \log_2(m)$.

As noted above (2.3.4) the algorithm will determine the passing area for a digital curve with one, two or more pixels. Therefore, when solving in parallel, each component can contain any number of pixels. However, since with fewer than three pixels the algorithm does not perform hull constraints the component size should be at least 3 pixels to perform optimal efficiency.

2.4.2 Further Research and Subsequent Chapters

The definition of the computational geometry structure introduced in this chapter, the infinite hull, is described in terms of general point sets. Thus, the algorithm should be directly extendible to find the three dimensional domain of digital planar surface. That digital surface can be decomposed into upper and lower surface boundaries from which the infinite hull is computed. Each vertex is inspected for inclusion in the infinite hull against all its neighbors, which were just two vertices in the two-dimensional case. As with 2D digital lines, we may compute the infinite hull of disconnected digital surfaces and each component can be solved separately in parallel. However, the iteration of the boundary vertices through the upper and lower surfaces must be defined differently. With two dimensional digital line segments, the sequence of iteration and backtracking is straightforward since each vertex is connected to exactly one previous and one successive vertex. With three (or more) dimensions, each vertex is connected to at least four vertices

and, therefore, a different approach must be used. This is a topic that is currently being investigated. In addition, it is not as clear if the linear runtime complexity will still hold.

The foundation of the method discussed in this paper is that convex hulls of a point set can be used to determine a digital curve's passing area. It can be conjectured that the algorithm would yield the same results for any curve for which a hull can be computed. Edelsbruner (1983) extended the concept of convex hulls with the α -hull. The α -hull of a point set, S , is defined as the minimum intersection of discs with curvature α that include all points in S . Using such a framework we shall extend the results discussed in this chapter to circular (chapter 3), as well as elliptical digital curves (chapter 4). We shall see that these extensions, too, are computable in linear time.

Chapter Three:

Computing The Domain of a Digital Circular Arc

Abstract

Let $C_d = \langle P_0, \dots, P_m \rangle$ be a digital curve in discrete space Z^n that was generated by some digitization function of an original curve in world space $C_r \in R^n$. An important question in computational geometry is the recovery of the original curve in world space subject to the constraints of the digitization function. In computer vision and object recognition tasks, the dimensionality of the world space is $n = 2$ or 3 . This chapter considers circular arcs of a known curvature α and answers the following question in linear time: from a sequence of 4-connected pixels, determine with subpixel accuracy the possible area in R^n that the original curve could have passed through. In order to compute this passing area, a geometric structure termed the α_∞ -hull is introduced.

From this information, the set of all possible circle centers (termed the *domain*) of radius $r = 1/\alpha$ is constructed in constant time. The advantages that the computational geometric approach used in this chapter has over other methods in the literature is that it is directly parallelizable and can readily handle a sequence of disconnected digital arcs. Based on this, the algorithm and its complexity will be shown to likewise hold for 8-connected digital arcs. In addition, the approach presented here appears to be extendible to other curves for which hulls may be determined.

3.1. Introduction

The digitization process converts visual data from the world scene into a digital image. Each region in the image is bounded by digital curves that define that region. This chapter considers regional boundaries that are (partially) comprised of circular arcs and addresses the following problem: Given a connected digital circular arc represented by a list of n connected pixels, find the set of all circular curves that digitize to that digital curve. This set is known as the *domain* (Dorst, Smeulders, 1984). The domain will be described by the set of all of the circle centers of a given radius r known *a priori*.

Each circular arc that is a member of the domain can be the actual curve that existed in the world scene. The original curve from the world scene is referred to as the *preimage curve*, or *preimage*. The digitization process causes the inevitable loss of detail and therefore prevents the recovery of the preimage from the digital curve. This led some to consider the domain as the pseudo-inverse of the digital curve (Dorst and Smeulders, 1987).

Section 3.2 characterizes the domain in terms of sub-pixel boundaries that all the circles of the domain must pass through (termed the *passing area*). A bijection is shown between the vertices of the domain perimeter (of circle centers) and the circular arcs of the passing area perimeter. From the passing area information, the domain of the digital circular arcs is computable in constant time. Section 3.3 computes the passing area in $O(n)$ time by using the α_x -hull, an extension of the α -hull of Edelsbrunner (1983), which incorporates constraints from all boundaries of the passing area simultaneously.

In section 3.4, the shape of the passing area is explored and implementation considerations are considered. Based on this, in section 3.5, the algorithm that computes

the passing area is presented followed by a demonstration of the algorithm. In the final section 3.6, applications of this methodology are incorporated, including the situation of disconnected digital arcs and the inherent parallelism of the algorithms presented. For further directions, a conjecture is proposed for an algorithm that computes the domain of a digital curve whose radius is unknown.

3.2. Background

This section presents the definitions and concepts (section 3.2.1) found in the literature from which new structures (section 3.3) and subsequent algorithms (in sections 3.4 and 3.5) are introduced. Because of the importance of the duality between the passing area and the domain of the digital curve, section 3.2.2 elaborates on these constructs and their relationship to each other.

3.2.1 Definitions

For consistency with the various citations found in the literature dealing with digital curves representing line segments, the terminology used throughout this chapter is first presented. The digital curve is assumed to be a sequence of 4-connected pixels (unless explicitly stated as 8-connected.) Because of the simplicity of presentation of these concepts when assuming linear curves, the definitions are often depicted in a corresponding figure with digital line segments. However, the definitions themselves will broadly assume digital curves.

Definition. The pixel *boundary* of a digital curve consists of the upper and lower boundaries (figure 3.1).

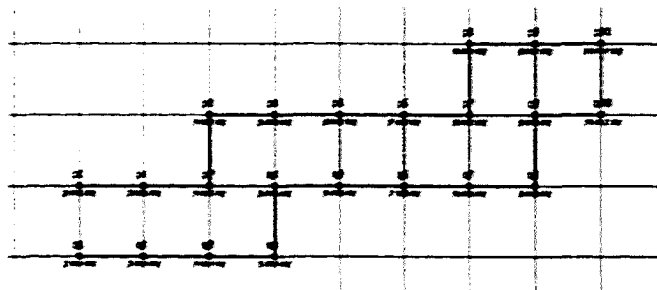


Figure 3.1. The upper and outer pixel boundaries of a digital curve. Note that the first and last pixel's boundaries are not included in the upper or lower boundaries.

Definition. The *joined boundary* of a digital curve is a (linked) list of the pixels comprising the upper and lower boundaries of the curve so that the head of each boundary is connected to the tail of the other (figure 3.2). The joined boundary is thus a circularly linked list.

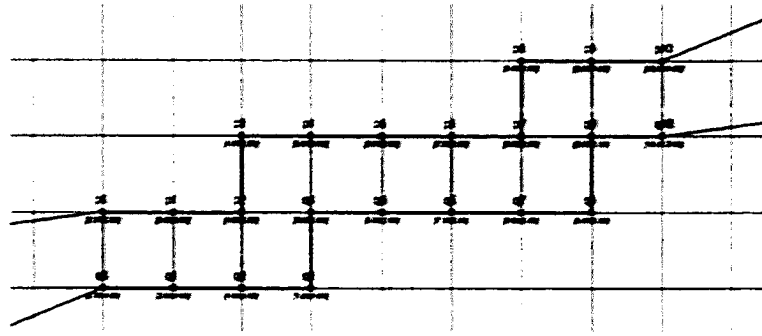


Figure 3.2. The joined boundary of the digital curve. The connectivity of the heads and tails of the upper boundaries is emphasized by the extra line segment extensions.

Definition. The *extension* of a digital curve segment is the continuation of that curve without the digital curve itself.

In figure 3.2 above, the edge connecting (in the joined boundary structure) the head of the inner boundary, p_0 , to the tail of the outer boundary, q_{10} , is the *extension* of the line segment defined by the points p_0 and q_{10} . Similarly, the edge connecting the head of the outer boundary, q_0 , to the tail of the inner boundary, p_{10} , is the extension of the line segment defined by the points p_{10} and q_0 .

Definition. (Yuan and Suen, 1995). The *passing area* of a digital curve is the maximal sub-pixel grid area that a world space preimage curve can be located (see figure 3.3).

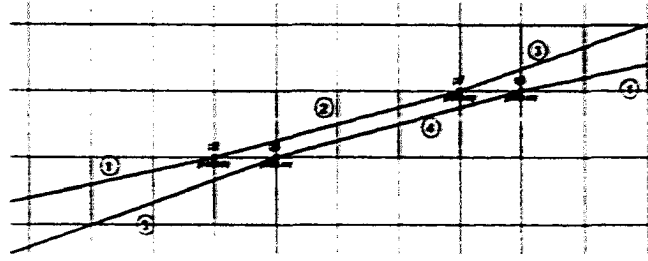


Figure 3.3. The line segments (labeled 2 and 4) and line segment extensions (labeled 1 and 3) define the passing area of the entire digital curve (shown in gray).

Definition. (Dorst and Smeulders, 1984). The *domain* of a digital curve is a region of parameter values from which any digital curve in the passing area can be generated (figure 3.4).

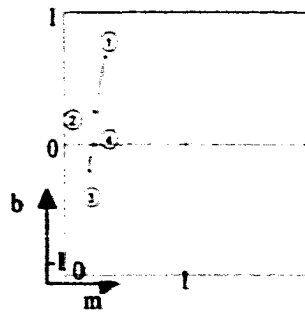


Figure 3.4. The domain of the digital curve presented in figure 3.3. The circled numbers by the domain boundary vertices directly correspond to the numbered line segments and line segment extensions of figure 3.3.

The Hough space of parameter values is a common method of describing the domain of a digital curve (as in figure 3.4 above that the (slope m , intercept b) values are the parameters for the line segments that digitize to the digital curve in figure 3.3). It should be noted that some in the literature (Yuan and Suen, 1995) parameterize the preimage curve with the passing area. While such a definition is simpler to describe geometrically, it is not useful in a practical sense since given the passing area boundaries it is not clear how to describe a *specific curve* that can digitize to the set of pixels. By using a representative system of parametric values, any specific curve that digitizes to the digital curve can be uniquely and readily described. A bijection exists between vertices and edges of the passing area boundary (figure 3.3) and edges and vertices, respectively, on the domain boundary (figure 3.4). In the case of circular arcs, these “edges” are circular arcs of curvature α (discussed in the next section).

Definition. (Edelsbrunner, 1983). Let S be a point set (see figure 3.5). The α -*hull* of S is the minimum intersection of circular discs of curvature $\alpha > 0$ (figure 3.5a) that include all points in S . If $\alpha < 0$ (figure 3.5b), then the α -hull of S is the minimum intersection of the complements of the circular discs (of curvature α) that do not include any points in S . If $\alpha = 0$ (figure 3.5c), then the α -hull is the convex hull of S .

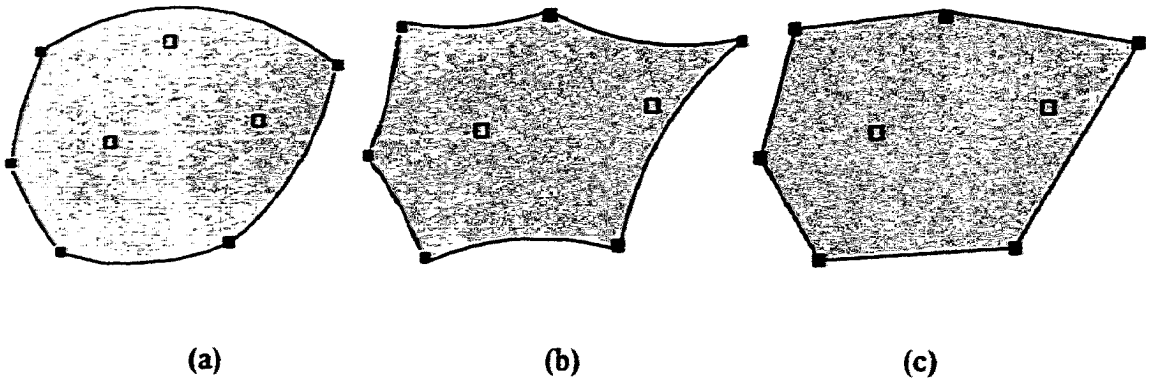


Figure 3.5. α -hulls for different values of α (a) $\alpha > 0$, (b) $\alpha < 0$, (c) $\alpha = 0$. The point set S includes the points represented by square pixels. Interior points of S are depicted hollow and exterior/boundary points are filled in.

In the next section, we extend the α -hull to the α_x -hull using the notion of the α -separation. Using the α_x -hull of the joined boundary, the passing area of the curve may be computed in $O(n)$ time. See figures figure 3.6 and 3.7 for input and output of this computation.

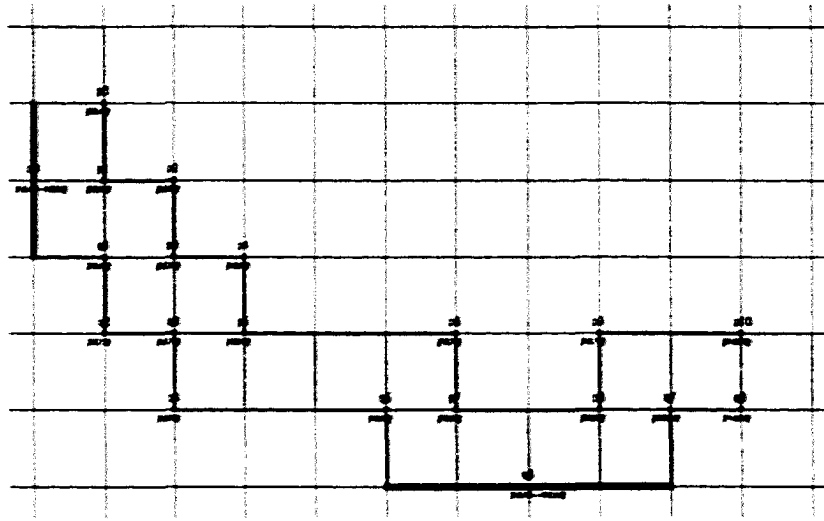


Figure 3.6. A digital circular curve. The joined boundary includes the upper and lower boundaries. The upper boundary includes vertices p_0 through p_{10} and all incident edges and the lower boundary includes q_0 to q_8 and all incident edges.

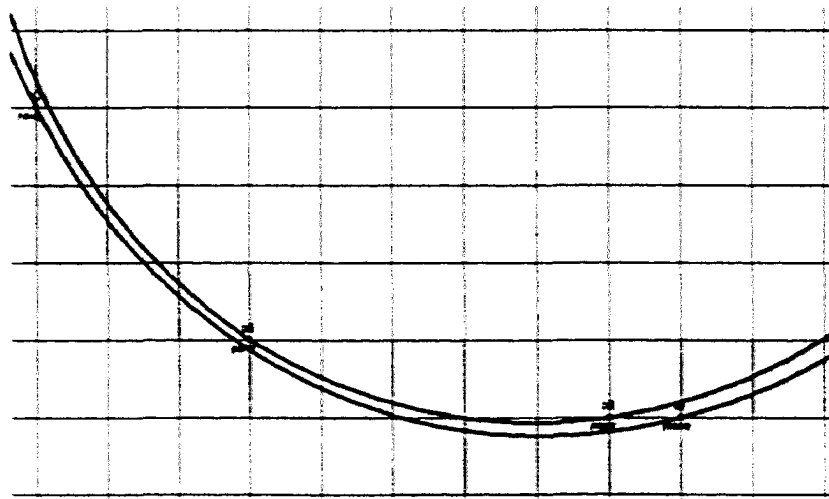


Figure 3.7. The passing area of the digital curve where the radius equals 8 (curvature $\alpha = 0.125$)

3.2.2 The Passing Area of a Digital Circular Curve

The passing area of a digital circular curve (henceforth, in this chapter, the digital curve) is the area on the grid in which every preimage of that digital curve must be found. The passing area is thus the union of all circular curves (henceforth, curves) that digitize to the digital curve. Each such world curve can be represented in (x,y) parameter space as a point that is the center of a circular arc with fixed radius r that digitizes to the digital curve. The union of all such points is the digital curve's domain. We thus have a bijection between complete circles that traverse the passing area and points in the domain.

For example, see figure 3.8 where a preimage circular arc (dotted) traverses the passing area and is incident to q_3 . The center of that arc is on the domain perimeter along the domain boundary edge that corresponds to vertex q_3 (as marked in the figure by a circled x). That edge (not necessarily linear) is comprised of all centers of a circular arc incident to q_3 ; that is, any circular arc centered at a point along that domain perimeter arc will intersect vertex q_3 and is a valid preimage.

In general, each circular edge and vertex on the passing area boundary corresponds to a vertex and circular edge on the domain's boundary, respectively. See again figure 3.8, for example, where there are five vertices on the passing area boundary such that each of those vertices is the circle center of one of the five circular edges on the domain boundary (since the circle centers of all circular arcs of radius r which are incident to a given vertex must be r away from that vertex, thus comprising a circular arc itself). Also, each circular edge of the passing area perimeter is centered at one of the five vertices of the domain boundary (since the domain boundary vertex is adjacent to two

domain boundary arcs which represent all circular arcs that are respectively incident to two passing area boundary vertices, each at a distance of r from those passing area vertices, a domain boundary vertex generates a circular arc that is distance r from two passing area boundary vertices which is thus connected by a circular arc in the passing area). Consequently, the domain is directly computable from the passing area, as well as the passing area from the domain, in $O(n)$ time where n is the number of vertices in the joined boundary.

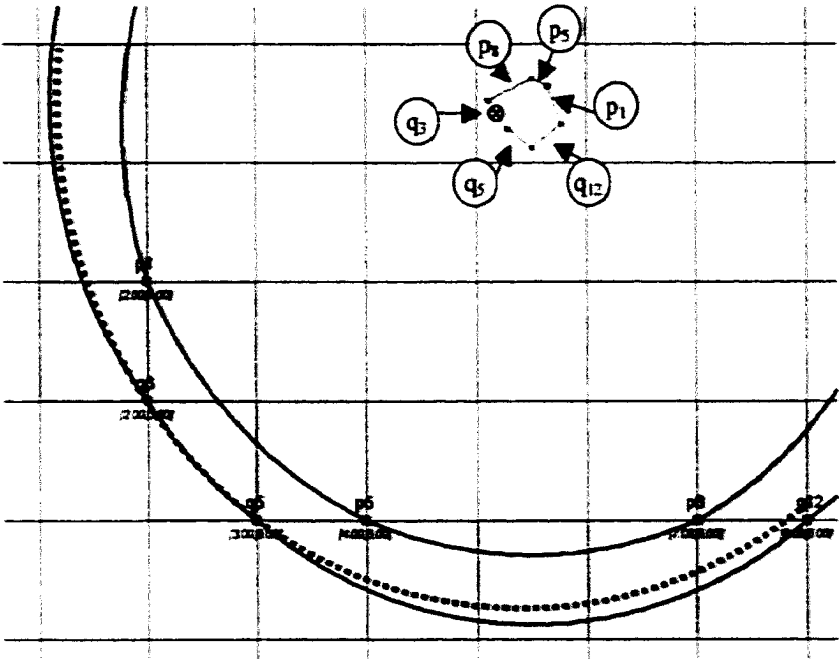


Figure 3.8. Each vertex and edge in the passing area corresponds to an edge and vertex in the domain, respectively. The corresponding vertices and edges are numbered. The dotted circular arc intersecting q_3 is centered along the corresponding domain circular edge as marked.

3.3. Constraints on the Passing Area

The digital curve places obvious limits on the world curve that can possibly digitize to that digital curve. The basic constraint, as imposed by the individual pixels in the passing area, is that the curve must intersect each pixel in the curve and must not intersect any pixel not in the digital curve. We can thus construct a non-minimal passing area, bounded above and below by the inner and outer boundaries, as shown in the previous section in figures 3.1 for line segments and 3.6 for circular arcs. Clearly, the world curve must be found within these upper and lower boundaries.

When dealing with digital line segments, the two boundaries were defined simply as upper and lower boundaries (section 2). In that case, both of the boundaries can be handled in an equivalent manner by computing their hulls (next section). With digital circular arcs, however, the two boundaries must be treated differently since the circle center to be considered is either on one side or the other of the circular arc. To distinguish between the boundaries in the case of digital circular arcs, these boundaries will be referred to as the inner and outer boundaries where the inner boundary is closer to the circle center than the outer boundary. Thus, for example in figure 3.8 above, the points p_1 , p_5 and p_8 are on the inner boundary and q_3 , q_5 and q_{12} are on the outer boundary. (Likewise notice that their corresponding vertices and edges on the domain of the digital curve are similarly grouped.)

3.3.1 Digital Curves as Separators

The passing area of a digital curve is found between the extremal boundaries of the digital curve's pixel space (defined in section 3.2.1). Thus, the digital curve can be thought of as separating (dividing) the point sets of the digital curve boundary such that it includes the inner boundary points and excludes the outer boundary points. A precise treatment of this consideration will allow for the determination of the minimal passing area of a digital curve (section 3.3.2).

Definition: The *separation hull* of two point sets S_1 and S_2 , $\text{hull}_{\text{sep}}(S_1, S_2)$, is the minimum intersection of half-spaces such that each half space includes every point in S_1 and excludes every point in S_2 .

Consider figures 3.9a-b. Let S_1 be the set of (squared) points on top and S_2 be the set of (rounded) points on the bottom. A digital curve that is found between the two point sets must not intersect either hull. The half spaces boundaries that determine the separation hulls $\text{hull}_{\text{sep}}(S_1, S_2)$ and $\text{hull}_{\text{sep}}(S_2, S_1)$ are shown as dotted lines.

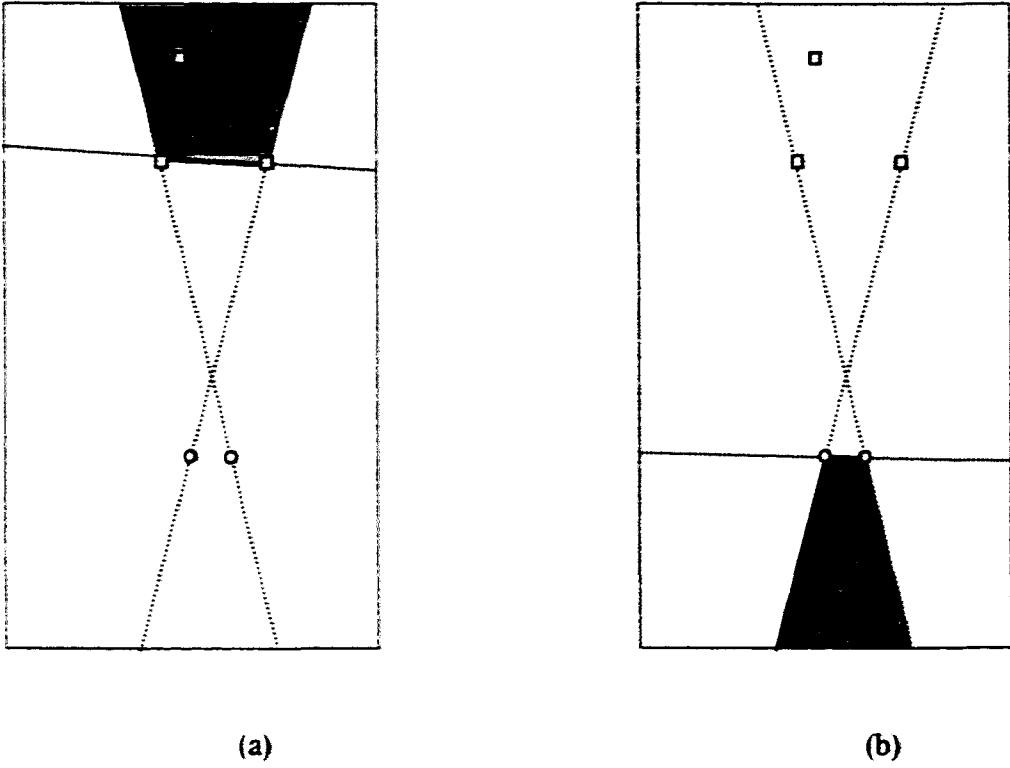


Figure 3.9. Shaded in gray (a) is $\text{hull}_{\infty}(S_1, S_2)$ where S_1 includes the three square points on the top and S_2 includes the two round points on the bottom. In (b), $\text{hull}_{\infty}(S_2, S_1)$ is shown.

Definition: The *infinite hull* of two point sets, S_1 and S_2 , $\text{hull}_{\infty}(S_1, S_2)$, is the area of $\text{hull}_{\infty}(S_1, S_2) \cup \text{hull}_{\infty}(S_2, S_1)$. See figure 3.10.

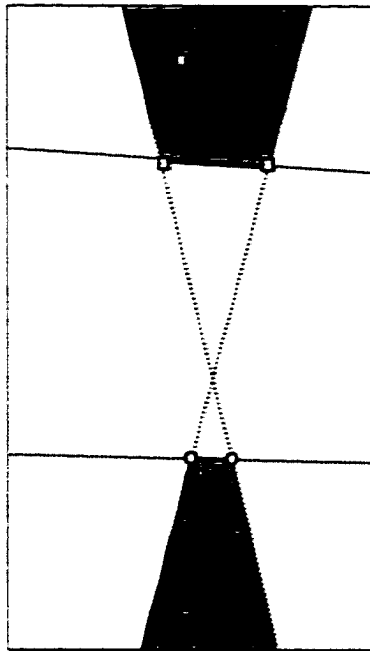


Figure 3.10. The infinite hull of S_1 and S_2 is shaded in gray. It is the union of the two separation hulls shown in figure 3.9. The half space boundaries that determine the separation hull are shown as dotted lines.

Whereas the above definitions are sufficient to compute the passing area for a digital line, the notion of the half-space will have to be extended to incorporate the concept of the α -hull in order to compute the passing area for digital circular arcs. Consider figures 3.11a and 3.11b on the next page. Again point set S_1 is comprised of the squared points on top and point set S_2 comprises of the rounded points on the bottom of the subsequent figures. Shaded in gray (figure 3.11a) is $\alpha\text{-hull}_{\text{sep}}(S_1, S_2)$ ($\alpha > 0$) where S_1 include the three square points on the top and S_2 include the two round points on the bottom. In figure 3.11b, $\alpha\text{-hull}_{\text{sep}}(S_2, S_1)$ is shown where $\alpha < 0$. A circle with radius $r = 1/\alpha$ that is found between the two point sets must not intersect either hull. The circular disc boundaries that determine the α -separation hull are dotted. It is the union of these

two structures, termed the α -*hull* (figure 3.12), that will provide the constraints to characterize the minimal passing area for digital circular arcs. The next section will more formally treat these definitions and elaborate on these structures.

Definition: The α -separation hull, $\alpha\text{-hull}_{\text{sep}}(S_1, S_2)$, where $\alpha > 0$ ($\alpha < 0$), of two point sets S_1 and S_2 is the minimum intersection of (the complements of) circular discs with radius $1/\alpha$ that include every point in S_1 and exclude every point in S_2 . (If $\alpha = 0$, then consider the half-planes as above.)

Definition: The α -infinite hull of two point sets, S_1 and S_2 , denoted $\alpha\text{-hull}(S_1, S_2)$, is $\alpha\text{-hull}_{\text{sep}}(S_1, S_2) \cup (-\alpha)\text{-hull}_{\text{sep}}(S_2, S_1)$.

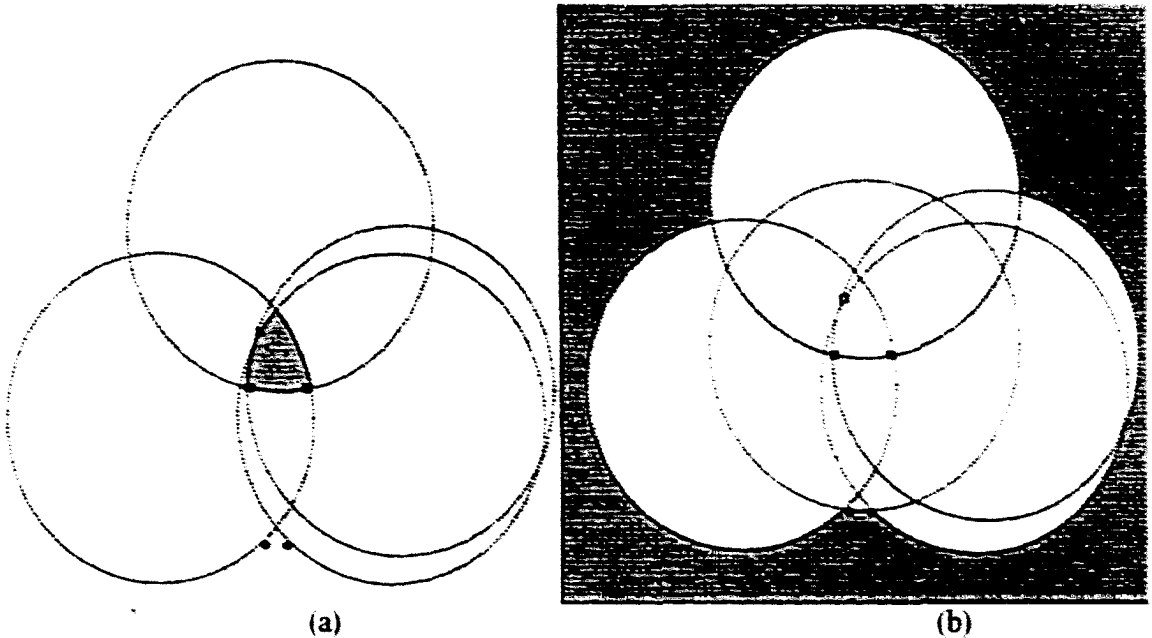


Figure 3.11. (a) $\alpha\text{-hull}_{\text{sep}}(S_1, S_2)$ where $\alpha > 0$ and (b) $\alpha\text{-hull}_{\text{sep}}(S_2, S_1)$ where $\alpha < 0$. S_1 is the set of square points on top and S_2 is the set of rounded points on bottom.

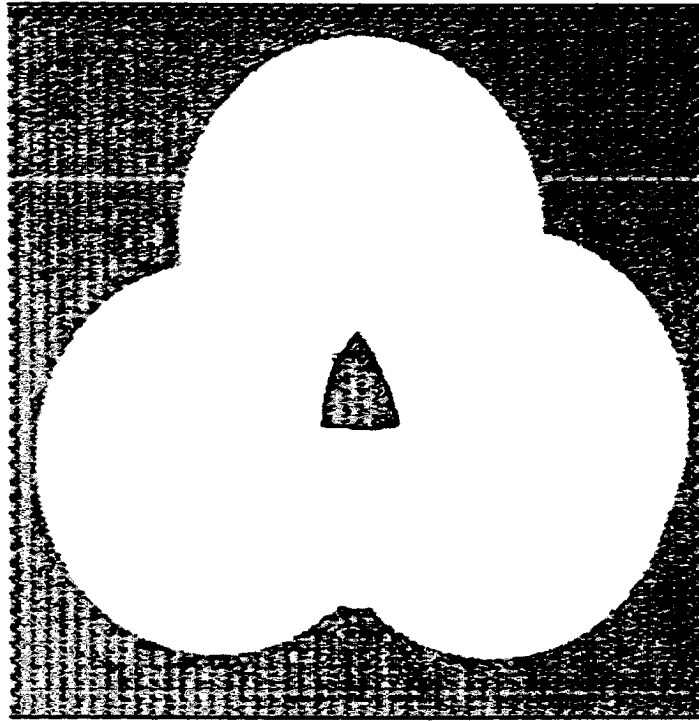


Figure 3.12. The α_x -hull of S_1 and S_2 is shaded in gray. It is the union of the two separation hulls shown above in figure 3.11.

3.3.2. Circular Separability

Figure 3.13 below presents the situation to be considered with digital arcs. Let $S_p = \{p_1, p_2, p_3\}$ be a point set that is encompassed by some circular arc C_r with radius r and at the same time excludes the point set $S_Q = \{q_1, q_2, q_3\}$. Then, the following two constraints and subsequent theorem characterizes any circular arc that passes between (separates) S_p and S_Q .

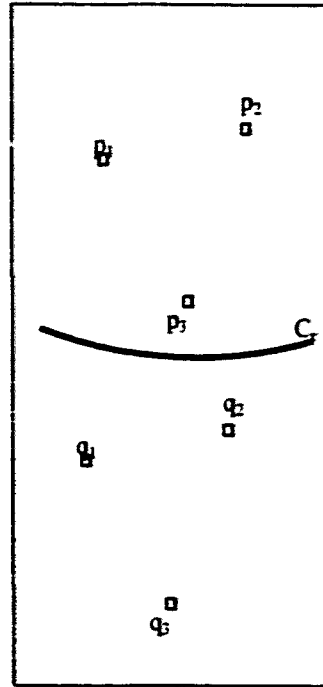


Figure 3.13. Circular arc C_r with radius r whose circular extension encompasses (includes) the point set $S_P = \{p_1, p_2, p_3\}$ and excludes the point set $S_Q = \{q_1, q_2, q_3\}$.

Constraint 1: C_r must not intersect the interior of the α -hull of S_P where $\alpha = 1/r$.

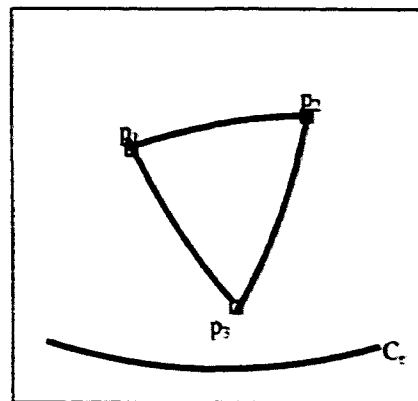


Figure 3.14. A circular arc C_r with radius r that includes the point set $S_P = \{p_1, p_2, p_3\}$ must not intersect the interior of the α -hull of that set where $\alpha = 1/r$.

Proof of Constraint 1. Assume that there exists a circular disc, d , with radius, r , that includes all points in S_P and intersects the interior of $\alpha\text{-hull}(S_P)$. Then $\alpha\text{-hull}(S_P) \cap d \subset \alpha\text{-hull}(S_P)$. But $\alpha\text{-hull}(S_P)$ is a minimal region, resulting in a contradiction. $\square$

Constraint 2: C_r must not intersect the α -hull of S_Q where $\alpha = -1/r$.

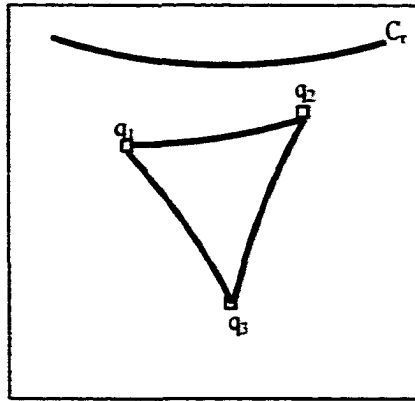


Figure 3.15. A circular arc C_r with radius r that excludes the point set $S_Q = \{q_1, q_2, q_3\}$ must not intersect the α -hull of that set where $\alpha = -1/r$.

Proof of Constraint 2. Assume that there exists a circular disc complement, d' , where the disc has radius, r , that includes all points in S_Q and intersects the interior of $\alpha\text{-hull}(S_Q)$. Then, $\alpha\text{-hull}(S_Q) \cap d' \subset \alpha\text{-hull}(S_Q)$. But $\alpha\text{-hull}(S_Q)$ is a minimal region, resulting in a contradiction. $\square$

These two constraints are summarized in the following theorem.

Theorem 3.1 (Circular Separability). Let C_r be a circular arc with radius r whose extension encompasses a set of points S_P and excludes a set of points S_Q . Then, C_r must not intersect the interior of the α -hull of S_P where $\alpha = 1/r$ must not intersect the α -hull of S_Q where $\alpha = -1/r$.

Note: Finding the passing area of a digital line is a special case of a circular arc with infinite radius. No distinction is made between the value of α from the upper and lower hulls since the values for both are 0 (a line has curvature 0).

As noted by Fisk (1986), any circular arc C_r with radius r (curvature α) that traverses the passing area must include every point in the inner boundary and exclude every point in the outer boundary. Moreover, corollary 3.2 presents a tighter constraint on the sub-pixel passing area of a digital circular arc.

Corollary 3.2. A circle C_r with radius r that includes every point in a set S_P and excludes all points from a set S_Q cannot intersect the α_x -hull(S_P, S_Q) where $\alpha=1/r$.

Proof. From theorem 3.1 it follows that by setting S_P and S_Q to the inner and outer pixel boundaries, respectively, the passing area must not intersect α_x -hull(S_P, S_Q). This will be proven by contradiction.

Suppose that circle C_r intersected α_x -hull (S_P, S_Q). Then, that circle must have intersected either α -hull_{sep}(S_P, S_Q) or α -hull_{sep}(S_Q, S_P). But, then, that α -separation hull is not a minimal region, contrary to its definition. □

3.3.3. Characterizing the Minimal Passing Area

The α_c -hull complement, shown in white in figure 3.12 above (at the end of section 3.3.1), is the circle's passing area throughout the entire grid. However, in computing the passing area of a circular arc of a digital curve, the region of interest is only the area that the arc passes through within the digital curve's pixel boundaries. This passing area is the area between the inner and outer boundaries of the digital arc as in figure 3.16 below and is identical to the intersection of the complement of the α_c -hull with the digital curve area.

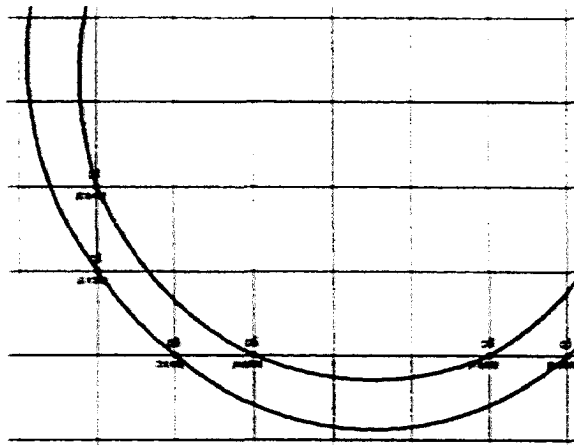


Figure 3.16. The passing area (the region between the circular curves) is the intersection of the pixel area (shown in gray) and the complement of the α_c -hull of the inner and outer boundaries.

In the next section many considerations in computing the α -hull as well as the details of the algorithm are considered. For example, it is shown that in addition to testing

vertices along the boundary it may also be necessary to consider some boundary edges along the outer boundary. Section 3.5 presents a Java implementation of this algorithm that computes the α_x -hull of the joined boundary in linear time and provides a brief demonstration of the algorithm.

3.4. Computing the Passing Area of a Digital Circular Arc

In the previous section, it was shown that a digital curve's passing area can be determined by computing the intersection of the digital curve area with the complement of the α_x -hull of the inner and outer boundaries. The inner and outer boundaries are composed of a path of linear edges that connect at vertices. It would thus seem necessary to consider each vertex and each linear edge of the boundary to determine the α_x -hull. In fact, in section 3.4.1, it will be shown that it is sufficient to consider the vertices of the inner and outer boundaries and a maximum of four linear edges from the outer boundary. In section 3.4.2 the complete methodology for determining the α_x -hull will be demonstrated.

3.4.1 Boundary Edges That Contribute To The α -Hull and α_x -Hull

For a set of line segments L comprised of points from a set of points P , $\text{hull}_{\text{convex}}(L) = \text{hull}_{\text{convex}}(P)$ where $\text{hull}_{\text{convex}}(S)$ is the convex hull of points incident to S . This follows from the definition of a convex region R , that is, $p_i, p_j \in R$ implies that $\overline{p_i p_j} \in R$. Similarly for arcs, when the curvature α of an arc is nonnegative, the α -hull

region is convex and thus, $\alpha\text{-hull}(L) = \alpha\text{-hull}(P)$. However, when α is negative, the α -hull is concave and then $\alpha\text{-hull}(L) \neq \alpha\text{-hull}(P)$.

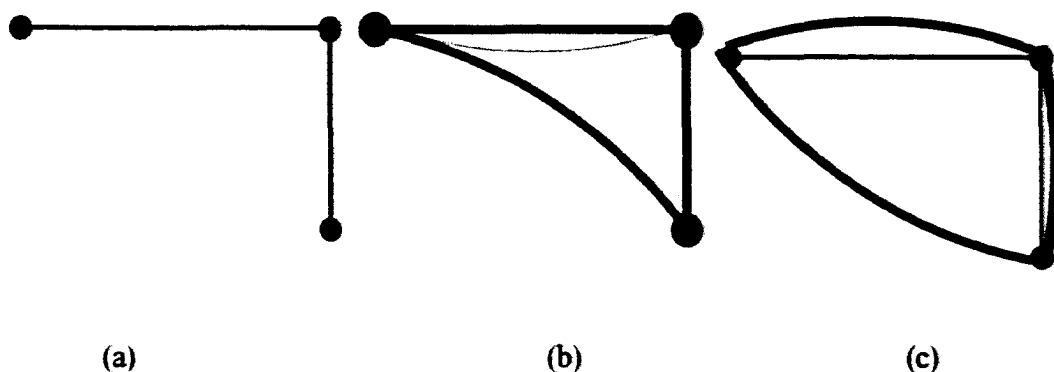


Figure 3.17. The hull of a set of line segments. The α -hull of the edges from (a) is shown in heavy lines and the α -hull of the edge end points is shown with the dotted lines. The endpoints of the edges alone is insufficient to determine the α -hull for $\alpha < 0$ (b). When $\alpha > 0$ the endpoints alone are sufficient to determine the α -hull (c).

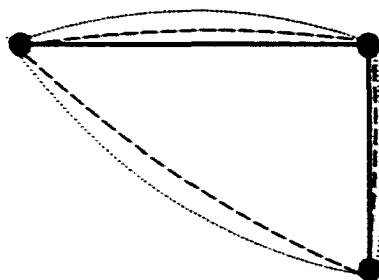


Figure 3.18 The α -hull increases with α . The α_1 -hull of a set of curves with curvature α_2 , is equivalent to the α_1 -hull of the endpoints of those curves.

Consider figure 3.18. The notion of convexity can be generalized from that of linear convexity to that of α -convexity for $\alpha \geq 0$. A region is α -convex if for any two points, $p_i, p_j \in R$ we have $p_i, p_j \rightarrow c \in R$ where c is a circular arc, with endpoints p_i, p_j , has curvature α , and angle less than π . It is clear that, when $\alpha_1 \geq \alpha_2$, any region R that is α_1 -convex is also α_2 -convex. Thus, where C_{α_2} is a set of arcs with curvature α_2 and P is the set of endpoints of those arcs, if $\alpha_1 \geq \alpha_2$, $\alpha_1\text{-hull}(P) = \alpha_1\text{-hull}(C_{\alpha_2})$, as illustrated in figure 3.18. Therefore, when computing the α -hull of a set of line segments, L , if $\alpha > 0$ then $\alpha\text{-hull}(P) = \alpha\text{-hull}(L)$. However, where $\alpha < 0$, in general, $\alpha\text{-hull}(P) \neq \alpha\text{-hull}(L)$.

We now apply the previous observations to the discussion of digital circular arcs. The value of α when computing the α_x -hull of the inner boundary is positive (since the actual circle encompasses the points on this boundary and the region is convex). Thus in this case, only the *vertices* along the inner boundary need be considered to determine the α_x -hull, as illustrated in figure 3.19a.

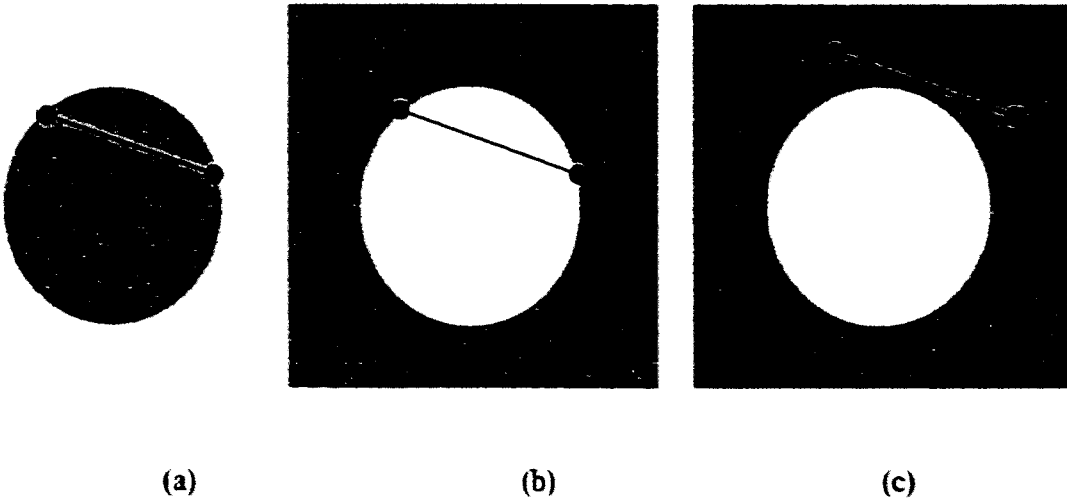


Figure 3.19. A circular disc is convex. Therefore for any edge whose endpoints are within the disc, the edge, too, must lie within the disc (a). However, a disc complement is

concave. Therefore, although the endpoints may lie within the disc complement, the edge may (c) or may not (b) lie in that region.

However, the value of α when computing the α_x -hull of the outer boundary is always negative. It is therefore possible for the endpoints of a linear edge to be within a disc complement while the edge itself would be outside the region (figure 3.19b). It may also be the case that the vertices *and* the linear edge are within the region (figure 3.19c). Therefore, it may be necessary to consider the edges in addition to the vertices that comprise their endpoints in determining the α_x -hull of the outer boundary. (When $\alpha=0$, the α_x -hull of the upper and lower boundary degenerates to the convex hull. Therefore, in this situation only the vertices need to be considered.)

The circumstances under which points along the edge will be found in the α -hull of the edge is now studied. The outer boundary is composed of a connected path of linear edges that intersect perpendicularly (as in figure 3.20). Denote two arbitrary edges, $e_1 = \overline{p_1 p_2}$ and $e_2 = \overline{p_2 p_3}$, as an *edge pair* (e_1, e_2) if both edges are incident to p_2 having external points p_1 and p_3 and internal point, p_2 (as illustrated in figure 3.20). The edge pair has a concave and convex side (assuming outside the boundary of some region).

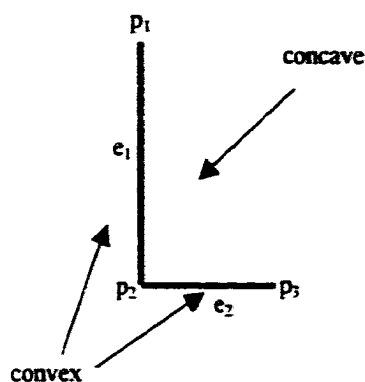


Figure 3.20. An edge pair has a concave and convex region. Depending on the value of α , the edges may contribute to the α -hull.

For each edge pair, either the concave or the convex side will face the digital curve. On the concave side of the edge pair in figure 3.20, when α is greater than or equal to the curvature of the maximal tangent circle to either edge, the endpoints alone will determine the α -hull. Denote by $\alpha_{\min}(p_1, p_2)$ the minimal curvature for which the hull of edge pair (e_1, e_2) is completely determined by its external points p_1 and p_2 . As shown in figure 3.21a, the center of that circle is the intersection of the normal n to the edge of greater length at its external (end) point and the perpendicular bisector b of the edge connecting the external (end) points.

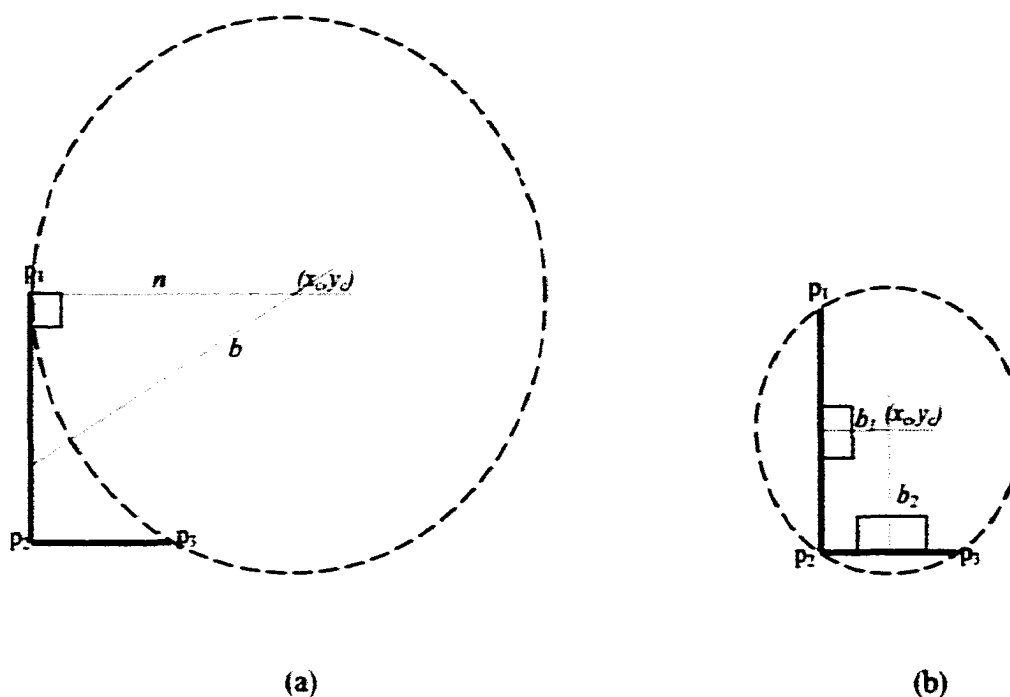


Figure 3.21. The α -hull of the edge pair $\overline{p_1p_2}$ and $\overline{p_2p_3}$ from the perspectives of the concave (a) and convex sides (b). In the case shown, the α -hull is completely determined by the edges' endpoints. For (a), n is the normal to $\overline{p_1p_2}$ at point p_1 , and b is the perpendicular bisector of $\overline{p_1p_3}$. For (b), b_1 and b_2 are the perpendicular bisectors of $\overline{p_1p_2}$ and $\overline{p_2p_3}$, respectively. (x_c, y_c) is the circle center of the maximal tangent circle to either edge.

On the convex side of the edge pair in figure 3.20, when α is greater than 0, the endpoints alone determine the α -hull. However, as shown in figure 3.21b, the minimum value for α must be at least the curvature of the minimum circle that circumscribes the

edge pair. If α is less than that value, the α -hull is undefined. The center of that circle is at the intersection of the perpendicular bisectors of $\overline{p_1p_2}$ and $\overline{p_2p_3}$, labeled b_1 and b_2 , respectively.

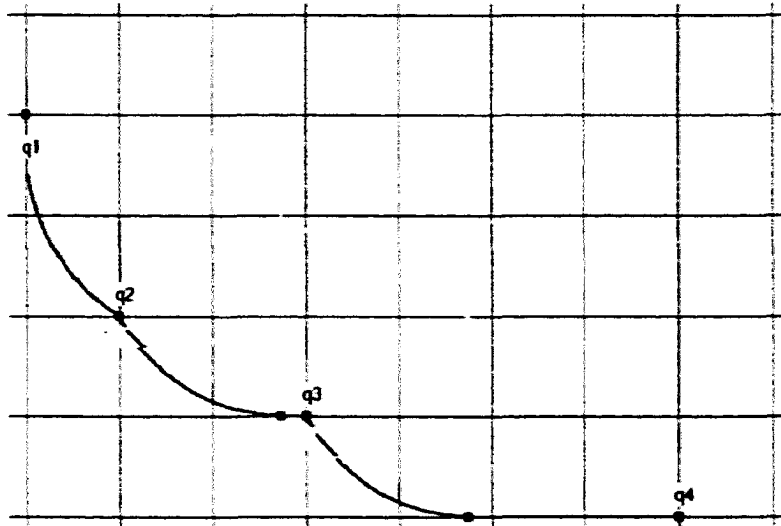


Figure 3.22. α is set to -0.5 when computing the α -hull of the outer boundary. The $\alpha_{\min}$ values are as follows: $\alpha_{\min}(q_1, q_2) = -0.4$, $\alpha_{\min}(q_2, q_3) = -0.4$ and $\alpha_{\min}(q_3, q_4) = -0.12$. Since all $\alpha_{\min}$ values are greater than α , an edge from each edge pair is on the α -hull, as shown. Note that the α -hull of the inner boundary (where $\alpha = 0.5$) is undefined and not shown.

Thus, when determining the α -hull of the outer boundary, only those edges whose $\alpha_{\min}$ value is greater than α need to be considered. See figure 3.22 where $\alpha_{\min}(q_1, q_2)$ and $\alpha_{\min}(q_2, q_3)$ equal -0.4 and $\alpha_{\min}(q_3, q_4)$ equals -0.12 . Therefore, when computing the α -

hull of the outer boundary with α set to -0.5 , since all $\alpha_{\min}$ values are greater than α , an edge from each edge pair is on the α -hull, as shown in figure 3.22. However, when α is -0.2 , only the final edge is on the outer hull and contribute to the α -hull, as shown in figure 3.23 (next page). The next theorem indicates that although when computing the α -hull of the outer boundary it may be necessary to consider edges from edge pairs whose $\alpha_{\min}$ value is greater than α , when computing the α_x -hull it is not necessary except for edges that may contain stationary points.

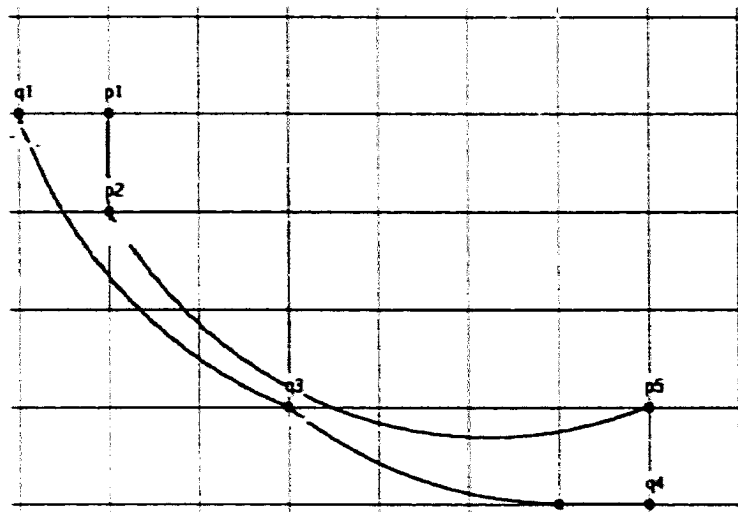


Figure 3.23. In this example α is set to -0.2 when computing the α -hull of the outer boundary. Only $\alpha_{\min}(q_3, q_4) = -0.12$ is greater than α . Thus, an edge from that edge pair is on the α -hull. Note that the α -hull of the inner boundary (where $\alpha = 0.2$) is defined and is shown.

Theorem 3.3. The passing area of any digital circle may have a maximum of four linear edges along its outer boundary. These edges will lie along the digital stationary pixels located on the maximal and minimal vertical and horizontal edges of the outer boundary.

Proof. In section 3.3 it was demonstrated that the intersection of the α_∞ -hull complement and the digital curve's pixel area is the passing area. A point along the passing area perimeter represents a point along a circular arc that traverses the passing area. If that point lies along a linear edge, then, the circle intersecting that point must traverse the pixel area of each pixel in the digital curve. But, a circle that is tangent to a line segment changes direction vertically or horizontally at that tangent point and can not intersect pixels along the passing area perimeter that are in the same direction the arc had been following. Thus, all pixels after the tangent point must also change direction. The tangent point, then, must be either along the minimal or maximal vertical or horizontal edges of the outer boundary, which generates at most four edges. □

See figure 3.24 on the next page for two examples where edges along the outer boundary contribute to the α_∞ -hull perimeter.

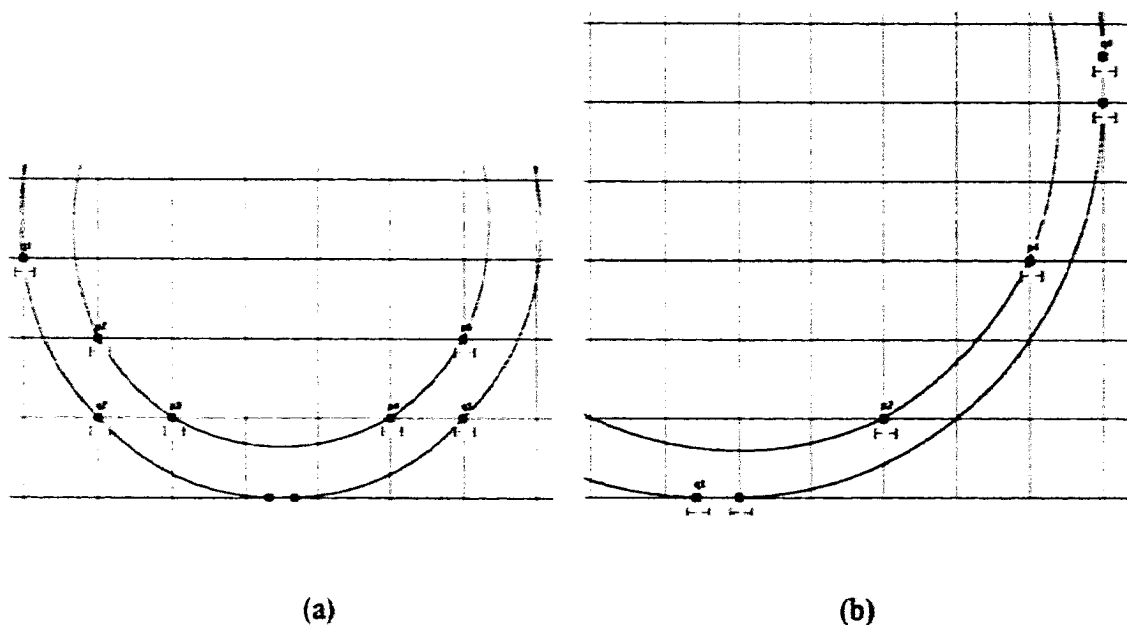


Figure 3.24. Only edges that are at the digital stationary points may contribute to the α_x -hull. In (a) this occurs on the bottom edge and in (b) occurs on the bottom and rightmost edge.

3.4.2. The Algorithm For Determining the α_x -hull

In the previous section, it was shown that along the inner boundary only vertices contribute to the α_x -hull while along the outer boundary both vertices and edges contribute. A vertex or edge along the inner or outer boundary is referred to as an *element*.

The passing area may be defined as a list of vertices and edges along its perimeter that are connected via circular arcs of curvature α , as shown in figure 3.24 above. The constraints that the α_x -hull places on the passing area may be divided to two major constraints, as discussed in section 3: the α -hull and α -separation hull constraints. The α -

hull constraint states that each element in the passing area lies within the α -hull of its resident boundary (either inner or outer boundary). The α -separation hull constraint additionally requires that each element of the inner hull not lie within the discs whose complements' intersection composes the α -hull of the outer boundary, and conversely.

In section 3.4.2.1, the computation of the passing area is more fully described as being constrained by the α -hull. The situation of edge elements on the boundary requires further computation and is dealt with in section 3.4.2.2. Section 3.4.2.3 handles a few special cases and the entire discussion is then followed by that of the α_∞ -hull computation (section 3.4.2.4).

3.4.2.1 The α -Hull Computation

When considering the constraints that the α -hull imposes on the passing area, only the portion of the α -hull that intersects the digital curve area is relevant. We therefore limit the discussion to computing that portion of the α -hull, as shown in figure 3.25.

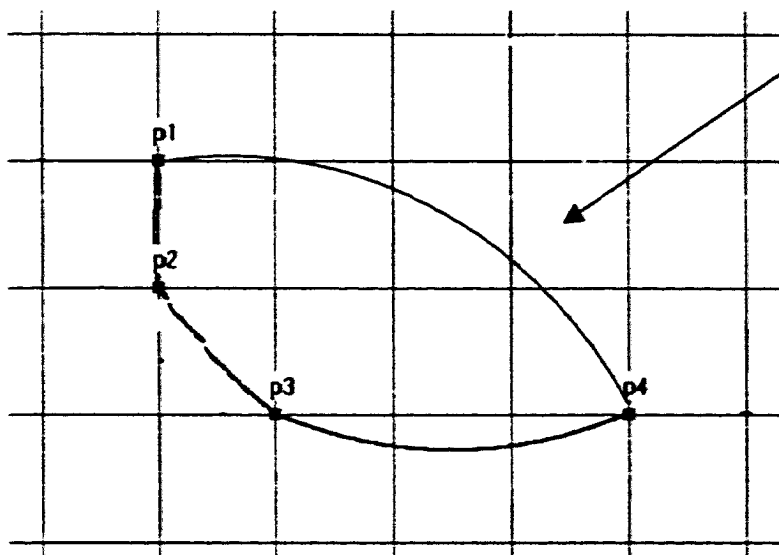


Figure 3.25. The α -hull of the inner boundary includes all arcs including the arc identified with the arrow. However, the arrowed arc does not constrain the passing area since only the arcs in the digital curve's pixel area constrain the passing area.

Let $P = (p_1, p_2, \dots, p_n)$ be the ordered point set where each of the p_i ($1 \leq i \leq n$) is a point along the inner or outer boundary. Note that two points, p_i, p_j ($1 \leq i, j \leq n, i \neq j$) and a curvature, α , define a single circle, denoted by $circle(p_i, p_j, \alpha)$ as shown in figure 3.26 since the center is $r = 1/\alpha$ away from the points p_i and p_j . (We follow the convention that when $\alpha > 0$ ($\alpha < 0$) the concave region is to the right (left) of the vector $\overrightarrow{p_i p_j}$.)

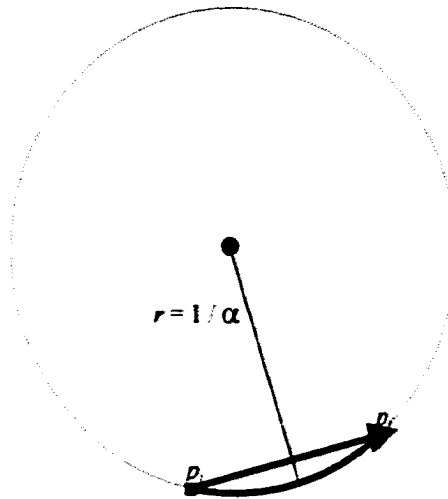


Figure 3.26. Two ordered points, p_i and p_j , and a curvature α , define a single circular edge. The edge's *circular extension* is shown as a dotted circle with radius $1/\alpha$. When $\alpha > 0$, the circular minor arc is to the right of vector $\overrightarrow{p_i p_j}$, as shown.

Initially, all points on the digital boundary are candidates for being on the α -hull perimeter. Let R be this ordered candidate list. The α -hull constraint states (section 3.3.1) that elements can be tested for removal from this list, based on the circular extension of the edge joining two successor elements along a boundary (figure 3.26). If this constraint were literally implemented, it would be necessary to traverse every element on the candidate list and then based on that element determine which of the other elements can not remain on the passing area perimeter. This would imply an $O(n^2)$ algorithm to compute the α -hull.

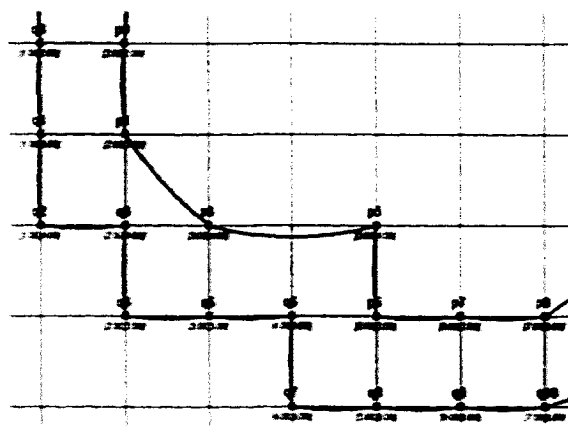
However, the pixel boundary places an implicit radial ordering to the elements thus allowing for a constant checking per element as follows:

When $\alpha > 0$, for any three consecutive points in R , p_i , p_j and p_k , which are all part of the same boundary (whether inner or outer), if p_k is in $circle(p_i, p_j, \alpha)$, p_j is on the α -hull perimeter; otherwise it is not. When $\alpha < 0$, if p_k is not in $circle(p_i, p_j, \alpha)$, p_j is on the α -hull perimeter; otherwise it is not.

We thus have the following $O(n)$ algorithm:

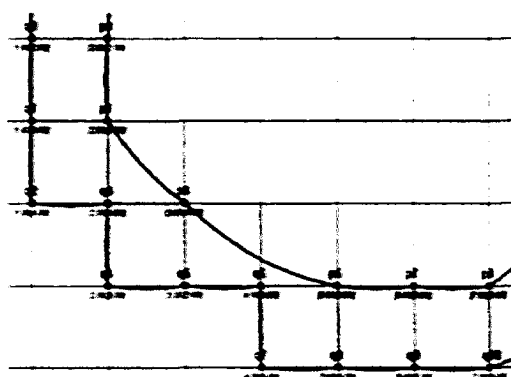
Initialize the candidate list, R , to P where P is the element set of either the inner or outer boundary. Traverse R in order. For each element, p_j , in R , determine if p_j is on the α -hull perimeter based on its immediate neighbors, as just described. If p_j is found *not* to be on the α -hull perimeter, remove p_j from R . Since each element's inclusion in the α -hull perimeter is determined by its immediate neighbors, the removal of p_j causes its forward and backwards neighbors' constraints to change. The forward neighbor's constraint will be determined as the algorithm proceeds to iterate. The backward neighbor, though, must be tested again for inclusion in the α -hull. Therefore, when p_j is removed from R , the iteration *backtracks* to the previous element in R (which may not be p_{j-1} if it was previously removed) and proceeds with traversing the rest of R . For example, in figure 3.27, after determining and deleting p_5 (a) the algorithm backtracks (b) and re-inspects the previous element p_3 . p_3 is then also deleted causing backtracking to p_2 which is not deleted (c). The algorithm has an $O(n)$ run time. Although backtracking occurs

after deleting an element from R , the maximum number of such backtracks is n , the maximum number of elements that may be deleted from R , prior to R becoming empty.

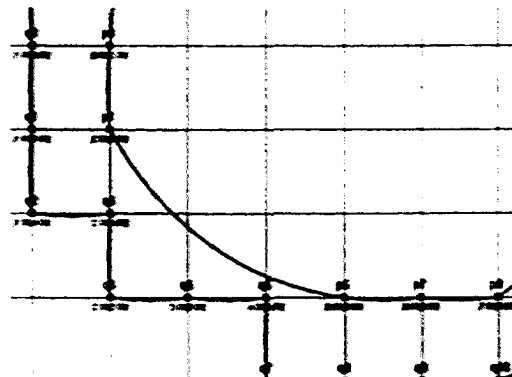


(2)

Figure 3.27a. The algorithm inspects p_5 and deletes it.



(b)



(c)

Figure 3.27b–c. After deleting p_5 (b) iteration backtracks to p_3 , which is also deleted (c).

The above rule for determining whether element p_j is on the perimeter is based on the inclusion or exclusion of the successive element p_k in R within $\text{circle}(p_i, p_j, \alpha)$. In the case that p_k is a vertex, this determination is straightforward since a vertex, as a single point, is either totally inside or outside $\text{circle}(p_i, p_j, \alpha)$. However, when p_k is an edge it is possible that only part of the edge is included in the circle, as shown in figure 3.28. The simple inclusion or exclusion rule that applies to vertices in determining whether to delete q_3 must be reformulated for edges. Also, when p_j is an edge, and the algorithm keeps p_j in R , it is possible that only a portion of p_j is on the α -hull perimeter (figure 3.29). The algorithm must eliminate the portion of edge p_j that is not on the α -hull perimeter. The next section discusses precisely how to handle these situations and other special cases that arise with edge elements on the boundary.

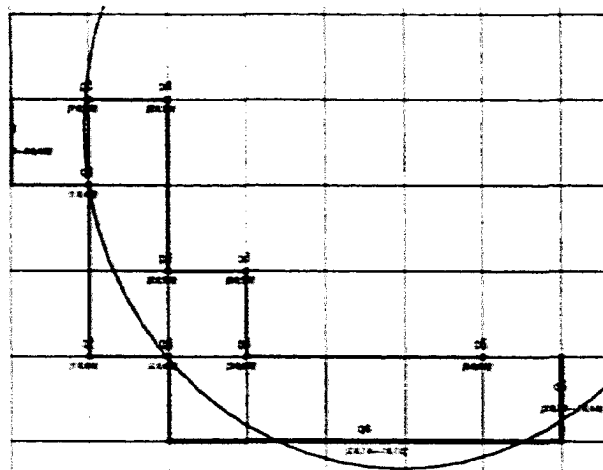


Figure 3.28. $\text{circle}(q_1, q_3, 1/3)$ intersects edge q_4 . The simple (vertex) rule does not apply to edges since the edge is both inside and outside $\text{circle}(q_1, q_3, 1/3)$.

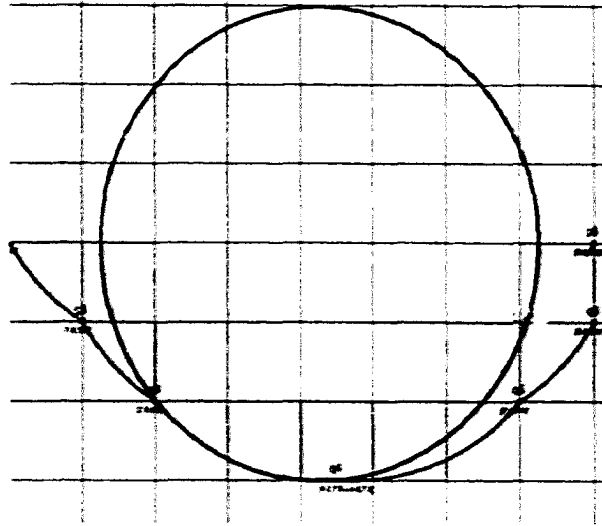


Figure 3.29. Although edge q_4 is not removed, only a portion of that edge remains on the α -hull. The circle shown is the circle defined by q_3 and q_4 which determines an endpoint of q_4 .

3.4.2.2 Handling Elements of the Boundary that are Edges

The α -hull perimeter is determined and can be described by the boundary elements that include vertices and linear edges. Since an element's inclusion in the α -hull is determined by its immediate neighbors, eight distinct possibilities exist for whether an element and its neighbors are vertices or edges, as listed in figure 3.30.

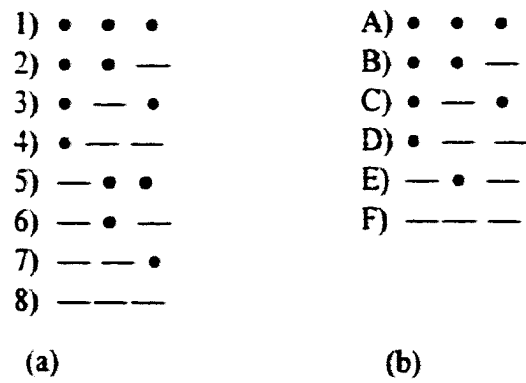


Figure 3.30a-b. The eight possibilities for the ordering of three consecutive elements along the digital curve's pixel boundary (a). Removing symmetries, six cases (A-F) must be considered in (b).

We indicate a vertex and edge with a dot (•) and dash (—), respectively. Of these eight, two symmetries exist: 2&5 and 4&7; thus, only six cases need to be considered. On the hull of the inner boundary, the α -hull is determined solely by vertices and thus only case A) is relevant. On the hull of the outer boundary though, all six possibilities exist. See figure 3.31 for examples.

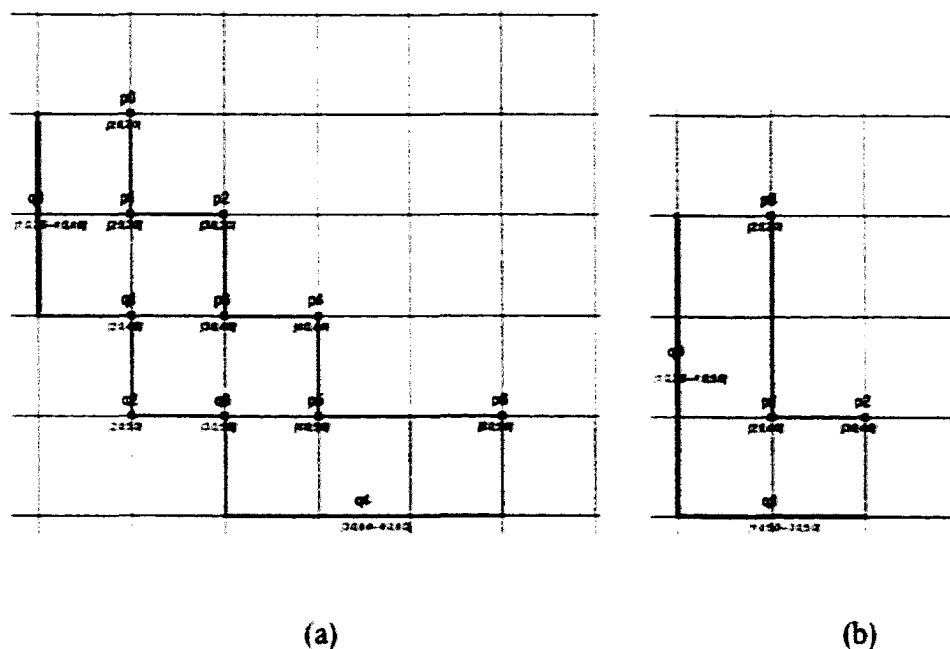


Figure 3.31a-b. Two examples of how the element ordering consists of edges and vertices. In (a) q_0 and q_4 are edges and all other elements are vertices. In (b) q_0 and q_1 are edges.

The six cases that exist on the outer hull are now listed and details are provided as to how they are handled. Call three consecutive elements in the candidate list as p_i , p_j and p_k . As we have seen in section 3.4.2.1, on the outer boundary if p_k is (not) in $circle(p_i, p_j, \alpha)$ then p_j will not be (will be) on the α -hull perimeter. When determining $circle(p_i, p_j, \alpha)$ we must consider the three possibilities that exist for the identities of p_i and p_j : (A) they are both vertices, (B) one is a vertex and the other is an edge and (C) they are both edges, as illustrated in figure 3.32.

- A. • •
- B. • —
- C. — —

Figure 3.32. Three unique orderings of the first two elements exist (modulo symmetry).

In case (A), the center is set on the perpendicular bisector of the two vertices at distance of $1/\alpha$ from p_i (figure 3.33a). Two such points exist, one to the right and one to the left of vector $\overrightarrow{p_i p_j}$. Since a counter-clockwise movement through the boundary is assumed, the minor arc from p_i to p_j is $< \pi$ and $\alpha < 0$; thus, the left point is selected, as illustrated in figure 3.33a.

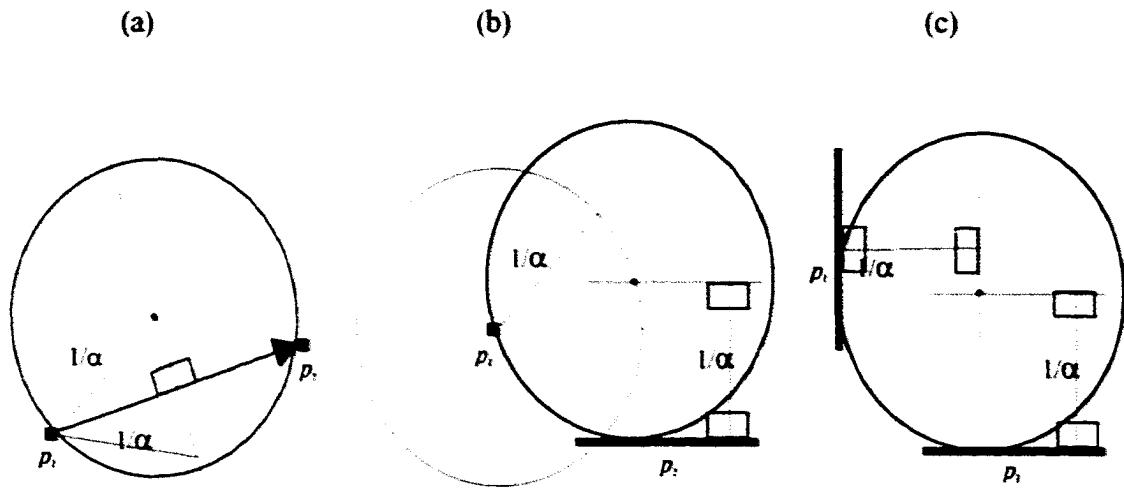


Figure 3.33. Two elements and curvature, α , define a unique circle. Three cases exist for the identities of those two elements: they may both be vertices (a), one vertex and one edge (b) or both edges (c).

In case (B) the circle center is located at the intersection of the circle with radius $1/\alpha$ centered at vertex, p_i , and the edge, p_j , translated to the left $|1/\alpha|$ units (figure 3.33b). In case (C), the center is located at the intersection of the edges, p_i and p_j translated $|1/\alpha|$ units to the left (figure 3.33c). For simplicity of notation, p_i and p_j in figures 3.33b-c refer possibly to edges also.

We now consider how the algorithm proceeds in determining whether p_j shall be removed from R . In summary, if no part of p_k is in $circle(p_i, p_j, \alpha)$, p_j is on the α -hull perimeter, otherwise it is deleted from the candidate list. When it is determined that p_j remains on the perimeter where p_j is a vertex, p_j is simply not removed (section 3.4.2.1). Likewise, when p_k is an edge, if any portion of that edge is found in $circle(p_i, p_j, \alpha)$, p_j is simply removed. However, when p_j is an edge, only a portion of the original edge will remain on the α -hull perimeter for the digital curve. Thus, the *portion* of the edge that is on the α -hull perimeter must be determined. That portion will be bounded by the intersection of the edge p_j with $circle(p_i, p_j, \alpha)$ and the intersection of p_j with $circle(p_j, p_k, \alpha)$, as illustrated in figure 3.34.

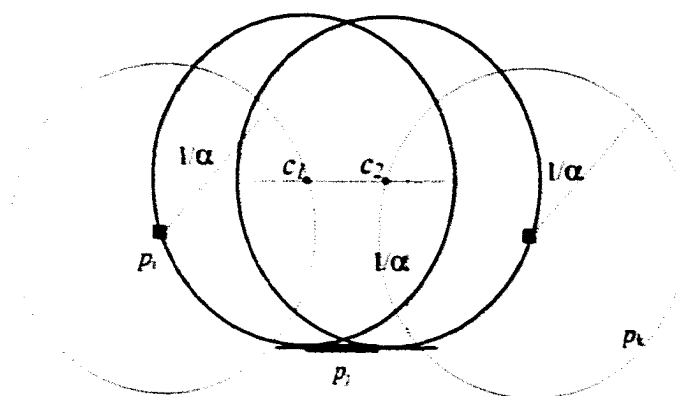


Figure 3.34. The endpoints of edge, p_j , that are on the α -hull of the outer boundary are determined by the two circles (centered at c_1 and c_2) defined by the neighboring elements

and p_j . Circle centers c_1 and c_2 are at a distance of $1/\alpha$ from edge element p_j .

3.4.2.3. Degenerate Cases with Edge Elements

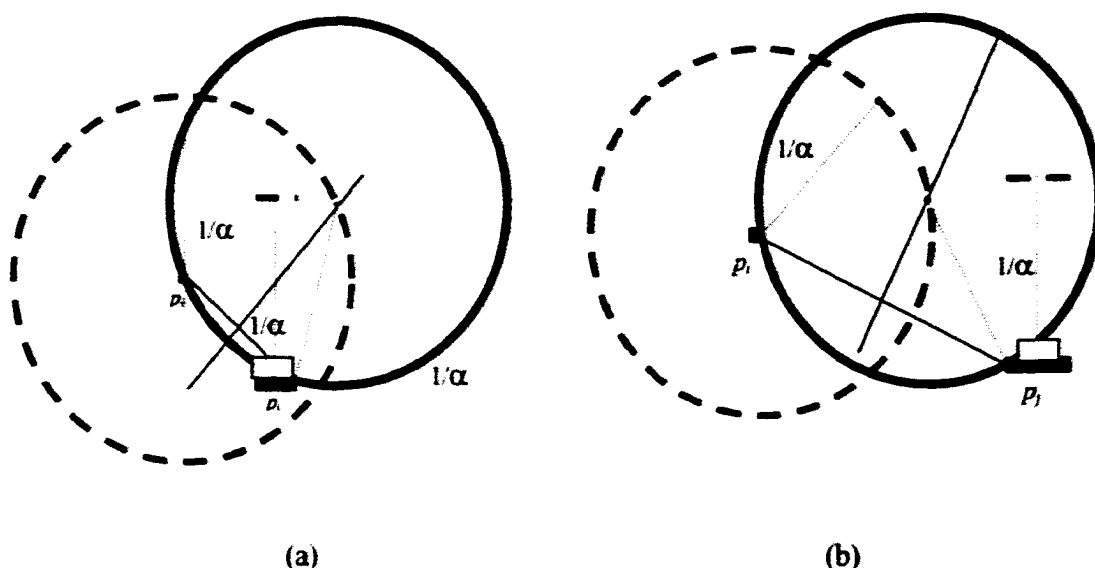


Figure 3.35. When one element is a vertex, p_i , and one element, p_j , is an edge, it is possible that no circle with curvature α will intersect p_i and be tangent to p_j . The heavy solid circle is the circle defined by p_i and p_j . The circle may be too large (a) or too small (b). Notice that in both cases the circle centered at p_i and the edge translated by $1/\alpha$ from p_j (both of these dotted) do not intersect.

Another consideration is that when p_i or p_j are edges it is possible that α is sufficiently small or large that a tangent circle to edge p_i or p_j does not exist. In case α is too large (figure 3.35a), the circle is determined by p_i and the endpoint of edge p_j furthest

from p_i . Where α is too small (figure 3.35b), the closest of the two endpoints is used. Whether α is too large or too small may be determined by testing if the translated edge (at a distance $1/\alpha$ from p_j) is inside the circle centered at p_i , or outside, respectively, as illustrated in figure 3.35. Thus in figure 3.32, case B may degenerate to case A and, similarly, case C may degenerate to cases B or A.

3.4.2.4 The α -Separation Hull Constraint

The α -separation hull, $\alpha\text{-hull}_{sep}(S_1, S_2)$, essentially constrains the α -hull so that the discs (or disc complements) whose intersection compose the α -hull of S_1 must not include any point in S_2 . Since the α -separation hull as defined is minimal, the circular extension of the arcs that are on the hull's perimeter must be incident to two points from either S_1 or S_2 , or both, as illustrated in figure 3.36.

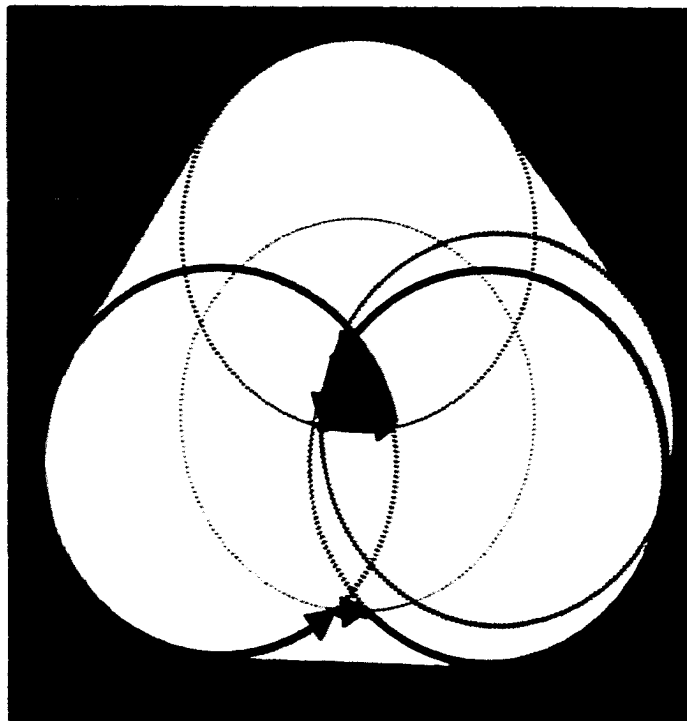


Figure 3.36. The α -hull is depicted in white and its complement in gray. The perimeter of the α -hull complement consists of a connected path of elements (arrows indicating direction) connected by circular edges with curvature $1/\alpha$. In this example, the endpoints of the circular edges are vertices from the inner and outer boundaries.

As defined earlier (section 3.4.1), the α -hull of S_1 and S_2 is the union of α -hull_{sep}(S_1 , S_2) and α -hull_{sep}(S_2 , S_1). When finding the passing area, the only relevant portion of the α -hull complement is its intersection with the pixel area. Therefore, only the vertices and edges of the inner and outer boundary are of significance; not the entire

α_x -hull. Based on this discussion, it is shown that the passing area may be defined as a path of boundary elements, consisting of vertices and edges, connected by circular edges.

The α -separation constraint (section 3.3.1), as it constrains elements along the boundary by its neighbors, is identical to the α -hull constraint with one difference. In this case, the neighbors of an element on the candidate list are considered from either the inner or outer boundary and these will determine whether that element is on the hull's perimeter. This requires a new ordering for element connectivity. This ordering of all elements from both boundaries is defined as follows: Let $S_1 = \{p_1, \dots, p_{n_1}\}$ be the ordered sequence of the n_1 inner boundary elements listed in a consecutive counter-clockwise fashion. Let $S_2 = \{q_1, \dots, q_{n_2}\}$ be the ordered sequence of the n_2 outer boundary elements listed in a consecutive counter-clockwise fashion. Then, $S = S_1 \cup S_2 = \{s_1, \dots, s_t\} = \{p_1, \dots, p_{n_1}, q_1, \dots, q_{n_2}\}$ where the $\cup$ is ordered and contains $t = n_1 + n_2$ elements initially on the candidate list for the passing area perimeter. For this connectivity, the ordering of elements in S is circular so that the first and last elements in S are also neighbors.

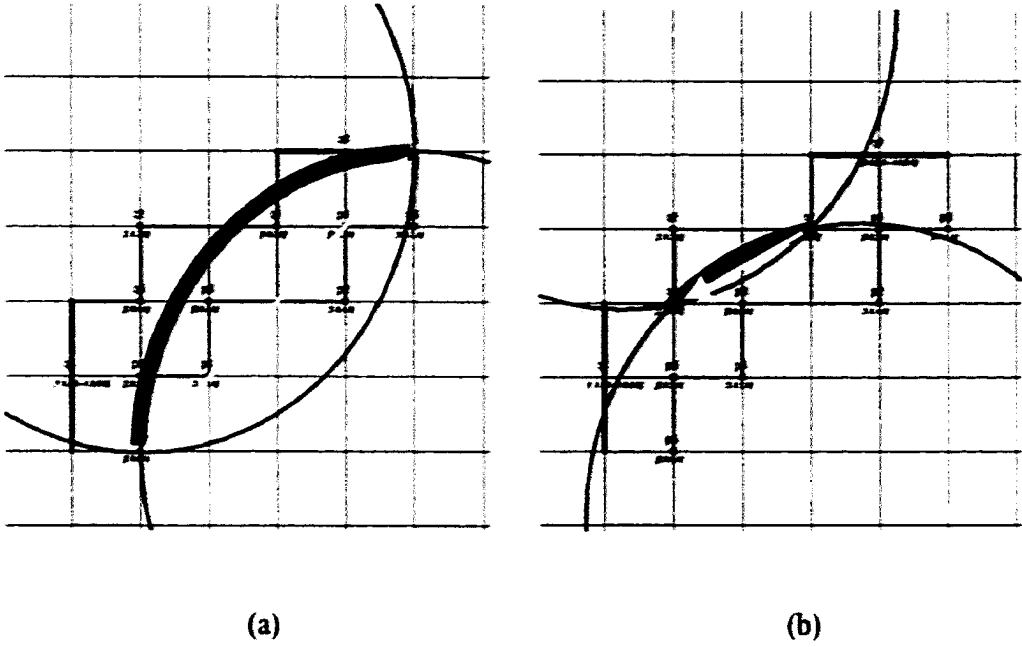


Figure 3.37. When the two elements that determine a circle are on different boundaries (a), the circle center to the right of vector $\overrightarrow{r_i r_j}$, shown as a white arrow, is selected (left arc). When both elements are on the same boundary the left circle center (right arc) is selected (b). The candidate circular arc is darkened.

Algebraically, two elements of S , r_a and r_b , are neighbors if and only if $|a \bmod t - b \bmod t| = 1$. Informally, this new ordering essentially connects the elements of S_l to those of S_2 so that the first element of S_l and S_2 are connected to the last element of S_2 and S_l , respectively. This connectivity will be used to determine the α_∞ -hull in the same fashion as connectivity along the inner and outer boundaries was used in the previous sections to determine the α -hulls of those boundaries.

In section 3.4.2.1 it was shown how to determine the $circle(p_i, p_j, \alpha)$ where both p_i and p_j are elements on either the inner or outer boundary. Now, when determining $circle(r_i, r_j, \alpha)$ it is possible that r_i and r_j are on different boundaries. As a result of this, there are two arcs whose circular extensions can be defined by one point in S_1 and one point in S_2 (as in figure 3.37). The construction of this type of circle is now demonstrated.

There are two possible circles with curvature α that intersect or are tangent to two elements (figure 3.37). When r_i and r_j are located on different boundaries, the circle chosen is to the or right of vector $\overrightarrow{r_i r_j}$ (the circle center to the left of vector $\overrightarrow{r_i r_j}$ was selected). This center defines the major arc (i.e. greater than π) connecting r_{i-1} and r_i . In this research, it is assumed that the digital curve is less than π and thus all arcs connecting elements within either the inner or outer boundaries will be minor arcs and arcs connecting an element from one boundary to another boundary will be major, as in figure 3.37. The algorithm can determine the α_∞ -hull of a digital curve greater than π by setting $circle(r_i, r_j, \alpha)$ to the minor arc when r_i and r_j are on different boundaries.

The rule for removing an element from the candidate list, R , is updated from the previous sections as follows: For three consecutive elements in R , r_i , r_j and r_k , r_j is removed from R based on $circle(r_i, r_j, \alpha)$ and the boundary that element r_k is located on. If r_k is on the outer boundary, r_j is removed if r_k is in $circle(r_i, r_j, \alpha)$. If r_k is on the inner boundary, r_j is removed if r_k is not in $circle(r_i, r_j, \alpha)$. When r_k is an edge it is treated identically as was discussed regarding the α -hull (section 3.4.2.2-4). See figure 3.38 for an illustration of the α_∞ -rule just detailed. The rationale for this rule is that an element on the outer boundary may not be in a circle including inner boundary points by the

definition of the α -separation hull. Similarly, an element on the inner boundary may not be outside a candidate circle that does not include elements from the outer hull.

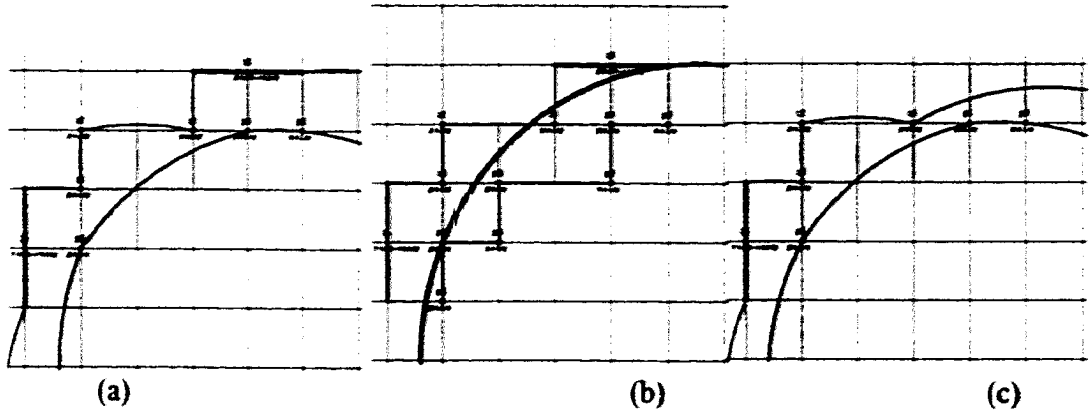


Figure 3.38. Edge q_0 is removed (compare (a) and (c)) because element q_1 , found on the outer boundary, is inside $\text{circle}(p_s, q_0, 0.2)$, shown darkened in (b). If q_1 were on the inner hull, q_0 would be removed only if q_1 was outside $\text{circle}(p_s, q_0, 0.2)$.

Assume that the candidate list R now has n initial elements from both boundaries. The α_x -hull algorithm has an $O(n)$ run time, similar to the α -hull. Each element is tested once during forward iteration with a total of n inspections. As mentioned in section 3.4.2.1, since each element is constrained by its immediate neighbors, deletion of an element from the candidate list may cause its previous or successive neighbors to also leave the candidate list. Therefore, after deletion of an element from the candidate list, the previous and next element must be tested for inclusion in the α -hull perimeter. The maximum bound on backtracking after deletion is also n , the numbers of elements

initially in R that may be deleted. In general, after deleting an element, the algorithm forward iterates through all the remaining elements in the candidate. However, when the deleted element is the last element in the candidate list, its next neighbor must be explicitly tested since forward iteration will not inspect it again. The upper bound on such explicit next neighbor inspections after deletion of the last element is n , the maximum number of possible deletions. Therefore the upper bound on element inspections for the algorithm is $3n$.

The next section includes the Java implementation of the algorithm followed by a graphical display of the iterations.

3.5. α_∞ -Hull Algorithm

This section describes the data structures and the algorithm used to compute the α_∞ -hull of the joined boundary. The algorithm receives a linked list of the elements of the joined boundary in sequential order as input (inner hull followed by outer hull), and prunes it to a linked list that only contains the elements on the α_∞ -hull boundary. The algorithm iterates through each element on the joined boundary and removes it from the list if that element is in the joined boundary's α_∞ -hull, thus maintaining a linear runtime complexity.

The implementation is in Java 2, which provides a set of classes rich of features that are used in the data structures and algorithm. The joined boundary and the α_∞ -hull algorithm are encapsulated in the `PassingArea` class. The `PassingArea` class is embedded as a subclass of the Java class `LinkedList`. The inheritance from the Java

`LinkedList` class allows for efficient and simple manipulation of each element in the joined boundary.

Three classes are defined to manipulate elements: `PassingAreaElement`, `PassingAreaPoint` and `PassingAreaLine`. `PassingArea` is defined to store and manipulate the linked list of elements. The classes from the Java 2 API (section 3.5.1) as well as the classes and their member methods and variables that are introduced, are defined below (section 3.5.2) followed by a discussion of the full implementation (section 3.5.3).

3.5.1. Classes from Java 2 and the Methods Used

Classes `Point2D` and `Line2D` from the `java.awt.geom` package encapsulate the elements of the joined boundary that include vertices and linear edges, respectively. The individual elements are placed into an indexed circularly doubly linked list with header node, encapsulated in the `LinkedList` class from the `java.util` package.

3.5.2. Classes, Methods and Variables Headings

This section presents the classes and their member variables and methods that are used in the α_x -hull computation implementation found in section 3.5.3. Class `PassingArea` encapsulates the joined boundary. As mentioned earlier, the boundary of the digital curve is composed of line segments that meet at vertices along the boundary. Since the vertices and linear edges at digital stationary points totally define the shape of

the boundary, the `PassingArea` object is initially set to the ordered sequence of vertices and edges along the joined boundary.

Method `computeAlphaInfiniteHull`, a member method of class `PassingArea` removes from the `PassingArea` object elements (i.e. vertices and edges) that are not on the α_∞ -hull boundary, with the result that only the elements on the infinite hull boundary remain. The determination whether an element shall be deleted is computed by the `elementDelete` method. The vertices and edges are stored as `PassingAreaPoint` and `PassingAreaLine` objects both of which are implementations of interface `PassingAreaElement`.

A `PassingAreaElement` is deleted if it is found to be within the α_∞ -hull of its immediate neighbors. For three consecutive elements in the `PassingArea`, `e1`, `e2` and `e3`, a circle `c` with radius `r` is determined that intersects `e1` and `e2`. `e2` is deleted if either `e3` is on the outer hull and `e3` is in `c`, or, `e3` is on the inner hull and is not inside `c`. (See section 3.4.2.2 for a full discussion.) Since two points define two circles of a given radius `r`, the circle chosen is determined by vector $\overrightarrow{e_1e_2}$ and whether `e1` and `e2` are on the same boundaries. Therefore a call to `isDifferentHulls` is made to apply the correct condition for determining the circle, `c`.

A `PassingAreaElement` from that linked list is tested for inclusion in the α_∞ -hull by testing whether the successive neighbor is within or without a circle defined by itself and the previous neighbor. That circle is stored as an `Ellipse` object that is a subclass of `Ellipse2D` from the `java.awt.geom` package.

The following is a summary of the methods and variables used.

```
class PassingArea extends LinkedList
```

The data structure used to contain the elements of the joined boundary

```
boolean isLeftHull(int index)
```

returns true if the vertex at index is on the inner boundary. An index is a reference to a element on the inner hull if it is less than or equal to leftTail.

```
boolean elementDelete (int index1, int index2, int  
index3, double r)
```

returns true if the element at index2 is in the α_x -hull and should be deleted. r is the radius and α is set to $1 / r$. The determination whether an element should be deleted is completely determined by that element and its previous and next element in the linked list, referenced by index1 and index3.

```
void computeAlphaInfiniteHull(double r)
```

computes the α_x -hull of the joined boundary where $\alpha = 1/r$. Iterates through the elements, calling method elementDelete on each element, deleting elements as necessary.

```
boolean differentHulls(int index1, int index2)
```

returns true if the element at index1 is on the inner hull and the element at index2 is on the outer, or vice versa. An index is determined to be from the inner or outer hull by calling method `isLeftHull`.

`size()`

returns the number of elements in the `LinkedList`.

`void delete(int index)`

deletes the element at index from the candidate list.

`int leftTail`

the index of the last element in the inner boundary, used in method `isLeftTail` to determine if an element is on the inner hull.

`void setLabel(String s)`

sets the label of an element to string `s`.

`interface PassingAreaElement`

The superclass of all elements along the passing area perimeter. Defines the polymorphic methods available to subclasses `PassingAreaPoint` and `PassingAreaLine`.

`class PassingAreaPoint extends Point2D`

`implements PassingAreaElement`

The data structure used to contain the vertices.

```
class PassingAreaLine extends Line2D
    implements PassingAreaElement
```

The data structure used to contain the linear edges.

```
class Ellipse extends Ellipse2D
```

The data structure used to contain the circle defined by two element and α .

```
int intersects(Line2D line, Point2D p[])
```

Sets $p[0]$ and $p[1]$ to the intersection points, if they exist, of this Ellipse and line. If only one intersection exists $p[0]$ is set to that intersection and $p[1]$ to null. If no intersection exists $p[0]$ and $p[1]$ are both set to null. The method returns the number of intersections that exist.

```
boolean contains(Point2D p)
```

Returns true if this Ellipse contains Point2D p and returns false otherwise.

```
void setEllipse(PassingAreaElement e1, PassingAreaElement
e2, double radius, int point[])
```

3.5.3 The Complete Algorithm

In the following sections the algorithm's structure is introduced (3.5.3.1) followed by the specification of the code (3.5.3.2). Finally, the algorithm is graphically displayed as iteration proceeds (3.5.3.3).

3.5.3.1 Introduction

The two methods that are the “workhorses” of the α_∞ -hull algorithm are `computeAlphaInfiniteHull` and `elementDelete` (figure 3.39 below). Method `computeAlphaInfiniteHull` iterates through every `PassingAreaElement` index on the candidate list (which includes both the inner and outer boundaries) and calls method `elementDelete` for each `PassingAreaElement` index. A `PassingAreaElement` is removed from the joined boundary if `elementDelete` returns true. This occurs when a given element is found to be within the α_∞ -hull and therefore the element is deleted.

The `elementDelete` method takes three `PassingAreaElement` indices as parameters, and the value of the circle radius, $r = 1 / \alpha$, as the fourth parameter. `PassingAreaElement` is the interface (i.e. superclass) used for the boundary elements. The three corresponding `PassingAreaElements` are `e1`, `e2` and `e3`. `elementDelete` determines if `PassingAreaElement` `e2`, which is in `PassingArea`, should be deleted. This determination is based on the relative location

of e_2 with respect to e_1 and e_3 as discussed in section 3.4.2. If e_2 is within the α_∞ -hull (the new constraint introduced by this paper) of e_1 and e_3 , then e_2 should be deleted.

During the determination whether an element should be deleted, method `elementDelete` must compute the circle as defined by e_1 and e_2 . The circle is set with a call to method `setCircle`, which sets the circle to intersect object e_1 and e_2 with radius r . This circle also determines the start and endpoints of e_1 and e_2 if they are edges, as discussed in 3.4.2.3-4. Therefore, `elementDelete` sets the edges to their appropriate portions, in addition to determining if e_2 should be deleted (as for example in figure 3.39 where q_4 is tested for deletion and the start point of that edge is reset.)

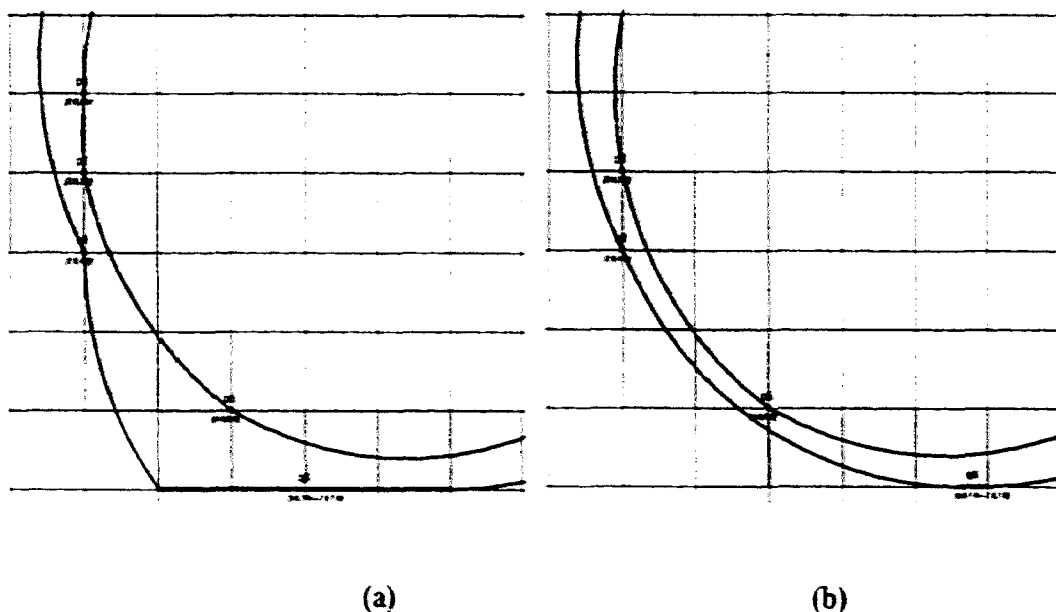


Figure 3.39. Method `elementDelete` tests element q_4 for removal. (a) The portion of edge q_4 that is on the α_∞ -perimeter is determined and, (b) the start and end points of q_4 are then reset.

If a `PassingAreaElement` is not removed, iteration proceeds with the next `PassingAreaPoint` on the boundary. If the `PassingAreaElement` is removed, iteration continues with the previous `PassingAreaElement` because the previous `PassingAreaElement` has potentially re-entered the α_x -convex hull due to the removal of its successive `PassingAreaElement`. If the deleted element is the last element in the joined boundary, the successive element is also tested (as discussed in section 3.4.2).

The algorithm finds the α_x -hull in linear time. Each `PassingAreaElement` is inspected for infinite convexity once during forward iteration totaling n inspections. Upon the deletion of a `PassingAreaElement`, the previous `PassingAreaElement` must be inspected again. The upper bound on such backward re-inspections after deletion is n , the number of `PassingAreaElements`. Upon the deletion of the last element forward inspection must also follow also with an upper bound of n additional inspections. Hence, the upper limit on element inspections, including both forward and backward reinspections, is $3n$; yielding an overall runtime efficiency of $O(n)$.

3.5.3.2. The Code

The following is the implementation of the `elementDelete` and `computeInfiniteHull` methods.

```
boolean elementDelete(int index1, int index2, int index3, double r) {
```

```

    PassingAreaElement e1 = (PassingAreaElement) get( index1 ),
                e2 = (PassingAreaElement) get( index2 ),
                e3 = (PassingAreaElement) get( index3 );

    boolean plp2 = (index1 == leftTail) || (index1 == size()-1);

    boolean isPoint1 = (e1 instanceof PassingAreaPoint),
            isPoint2 = (e2 instanceof PassingAreaPoint),
            isPoint3 = (e3 instanceof PassingAreaPoint);

    Ellipse circle = new Ellipse();
    Point2D intPoint[] = new Point2D[2];
    intPoint[0] = new Point2D.Double();
    intPoint[1] = new Point2D.Double();

    // sets circle to the circle that intersects e2 and e1, minor arc
    clockwise

    // with radius, r. If e1 and e2 are on different hulls the major
    arc is

    // selected indicated to method setEllipse with -r. The points
    along e1

    // and e2 that circle is tangent to are returned in intPoint[1]
    // and intPoint[2], respectively. If either e2 or e1 is not an
    edge, the

    // corresponding intPoint is set to 0,0.

    circle.setEllipse(e2, e1, ! plp2 ? r : -r, intPoint);

    PassingAreaLine line;

    // Only a portion of edge e1 is on a-hull perimeter. Must reset
    endpoints

```

```

if (! intPoint[1].equals( new Point2D.Double() ) ) {
    line = (PassingAreaLine) e1;
    line.setLine(line.getP1(),intPoint[1]);
    if ( line.getP1().distance( line.getP2() ) <= Lines.TOLERANCE) {

        // edge is a vertex. Must remove edge and replace it with vertex
        this.delete( index1 );

        PassingAreaPoint newPt = new PassingAreaPoint(
                                intPoint[1].getX(), intPoint[1].getY()
                                );

        newPt.setLabel( e1.getLabel() );
        this.add(index1, newPt );
    }
}

// Only a portion of edge e2 is on a-hull perimeter. Must reset
endpoints

if (! intPoint[0].equals( new Point2D.Double() ) ) {
    line = (PassingAreaLine) e2;
    line.setLine(intPoint[0], line.getP2());

    if ( line.getP1().distance( line.getP2() ) <= Lines.TOLERANCE) {

        // edge is a vertex. Must remove edge and replace it with vertex
        this.delete( index2 );

        PassingAreaPoint newPt = new PassingAreaPoint(
                                intPoint[0].getX(), intPoint[0].getY()
                                );

        newPt.setLabel( e2.getLabel() );
        this.add(index2, newPt );
    }
}

```

```

    }
}

    // determine if e2 should be deleted.
if (isPoint3)
    return circle.contains( (Point2D) e3 ) ^ isLeftHull(index3);
else {    // determine if entire edge (not) in circle.
    // if only portion is then e2 is deleted
    Point2D intersectionPts[] = new Point2D[2];

    // intersect is true if a portion of edge intersects ellipse.
    boolean intersect = circle.intersects( (Line2D) e3 , intersectionPts) >
0,

    // inside is true if one or both edge endpoints are inside
ellipse.

        inside = circle.contains( ((Line2D) e3 ).getP1() ) ||
            circle.contains( ((Line2D) e3 ).getP2() );
    // e2 is removed if edge intersects circle regardless of the
boundary

    // e3 is on. e2 is also removed if the entire edge is outside
ellipse

    // (inside = false) and e3 is on the inner boundary
(isLeftHull(index3) =
    // true) or the edge is inside ellipse (inside = true) and e3 is
on the

    // outer boundary isLeftHull(index3) = false).

    return intersect || ( inside ^ isLeftHull(index3) );
}
} // elementDelete

```

```

void computeAlphaInfiniteHull (double r)
{
    int i = 0, index1, index2, index3;

    // iterate while there are at least 1 element in each hull and 3
    // elements in both hulls

    // variables cycle2 and forward ensure that the algorithm continues so
    that
    // the first element is inspected twice. If the last element is deleted
    this
    // ensures the constraints of the successive element is tested, as
    discussed
    // in section 4.2.2.
    boolean cycle2 = false;    // is true after iteration completes forward
                                // iteration once
    int forward = 0;           // the number of times forward iteration
    proceeded

                                // after cycle2 is true

    // iteration proceeds as long as there are elements whose
    // constraints were either not yet checked or whose constraints may have
    // changed since they were last checked and there is at least one element
    // in both the inner and outer boundary
    // and a total of 3 elements in the joined boundary.

    while ( (forward < 2) && (leftTail >= 0) &&
            (size() - leftTail - 1 >= 1) &&
            ( size() > 2 ))
    {
        index1 = (i + size()) % size();
        index2 = (i + 1) % size();

```

```

index3 = (i + 2) % size();

if ( elementDelete(index1, index2, index3,r) )
{
    delete( index2 );

    if (i > 0) i--;

    if (index2 == 0) i = (i == 0 ? 0 : i - 1);

    forward = 0;

/* deletion may ripple backwards and the index is decremented. It
   is unnecessary to backtrack before the 1st element since those
   elements will be reached during forward iteration */

}

else {
    i++;

    cycle2 = cycle2 || i >= size();

    if (cycle2) forward++;
}

if (! infiniteHull)
{
    if ( (i == leftTail-1) || (i == leftTail) ) i = leftTail+1;

    if ( (i == size()-2) || (i == size() - 1) ) i = size();
}

}

} //computeAlphaInfiniteHull

```

Figure 3.40. The Java 2 source code for the α_x -hull algorithm.

3.5.3.3 The Demonstration

The following figures display the boundary elements as the algorithm iterates. The algorithm iterates through all elements along the boundaries of the digital curve (figure 3.41a). When the algorithm concludes, the α_x -hull appears as in figure 3.41b with its domain.

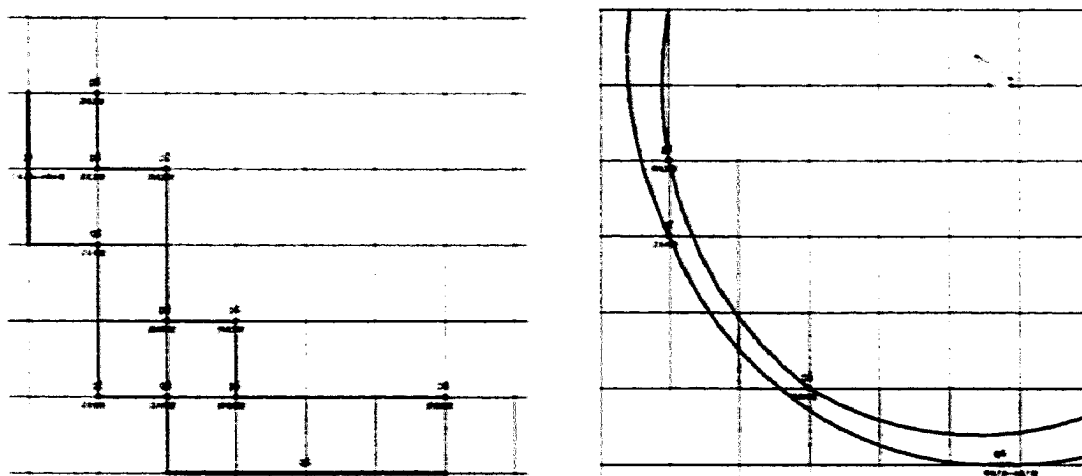


Figure 3.41. The elements along the inner and outer boundary are labeled in (a). Edges appear only on the outer boundary. After the algorithm concludes only vertices and edges that constrain the α_x -hull are found on the perimeter in (b). In the upper right corner of (b) the domain is shown.

The first point considered for removal is p_1 (figure 3.42), and then p_2 , and so forth. Although p_0 is the first element on the inner boundary, p_1 is tested first since `elementDelete` takes three parameters. They are initially p_0 , p_1 and p_2 where p_1 is

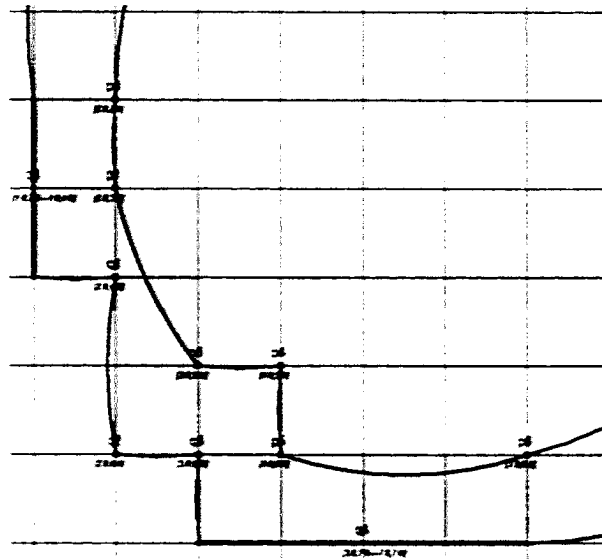


Figure 3.43. p_2 is deleted. After every deletion, inspection backtracks to the previous vertex. Thus we backtrack to p_1 following the deletion of p_2 . Since p_1 is not deleted, iteration continues with p_3 .

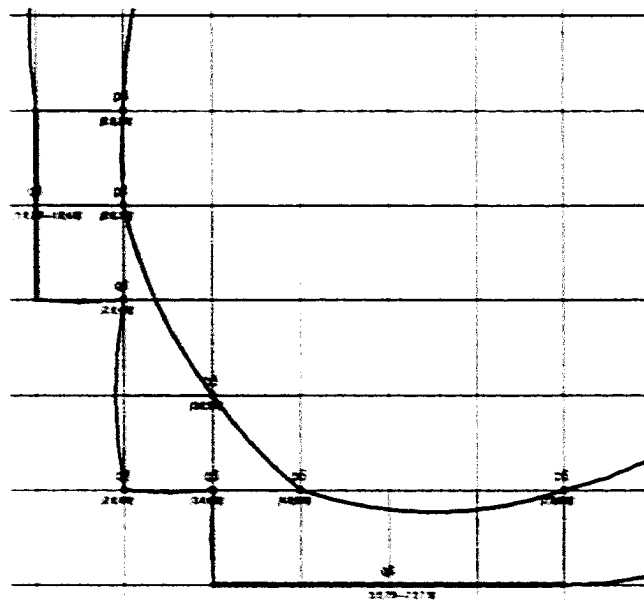


Figure 3.44. p_3 is not deleted. p_4 is deleted causing a backtrack to p_3 .

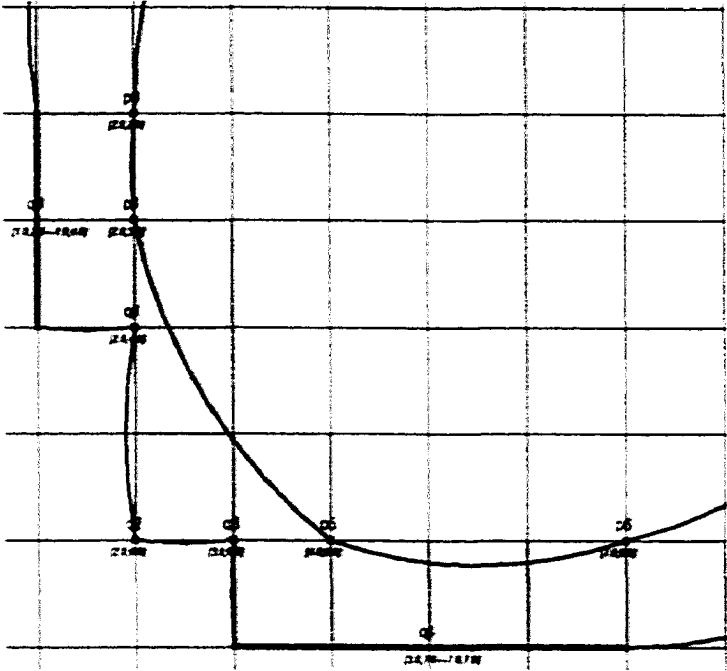


Figure 3.45. p_3 is deleted backtracking to p_1 which is not deleted. Iteration continues with p_5 .

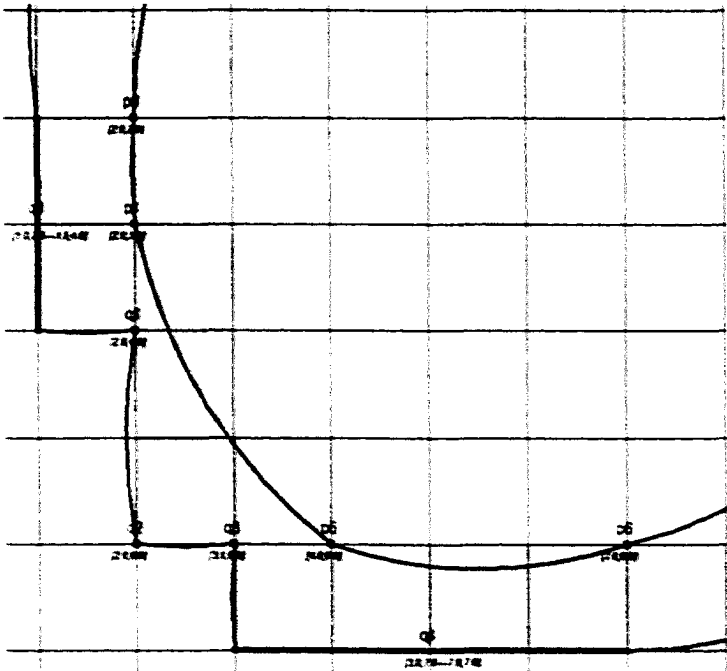


Figure 3.46. p_3 is not deleted. The algorithm continues with p_6 .

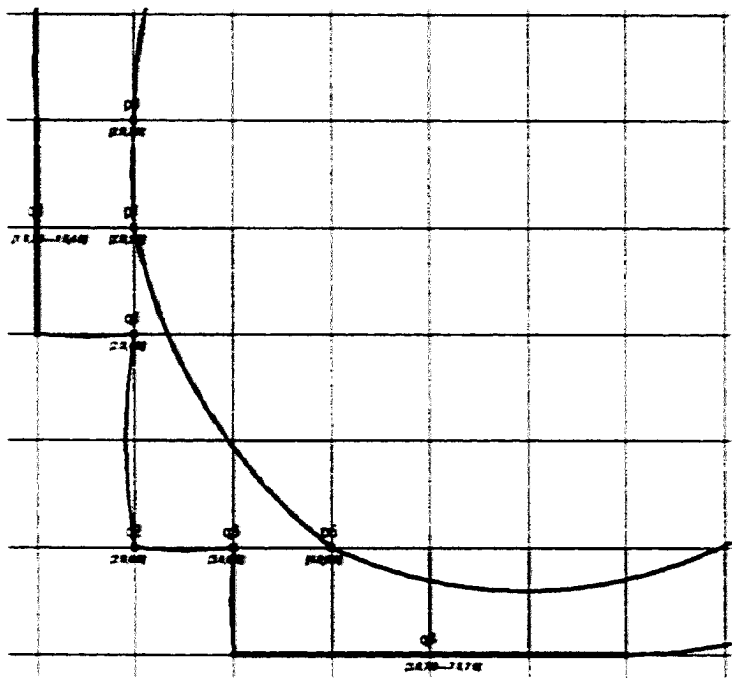


Figure 3.47. p_6 is deleted, backtracking to p_5 which is not deleted. Iteration continues with q_0 , an edge.

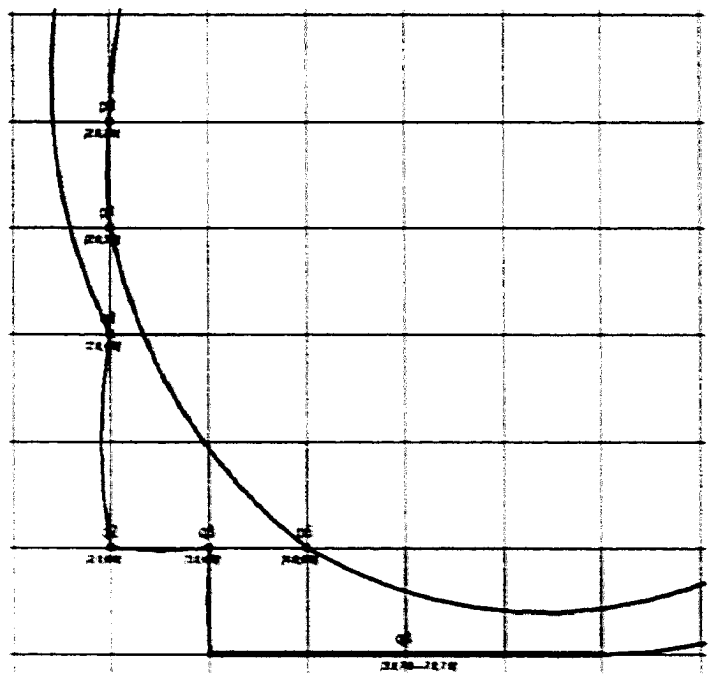


Figure 3.48. Edge q_0 is deleted. After backtracking to p_5 , iteration continues with q_1 which is not deleted.

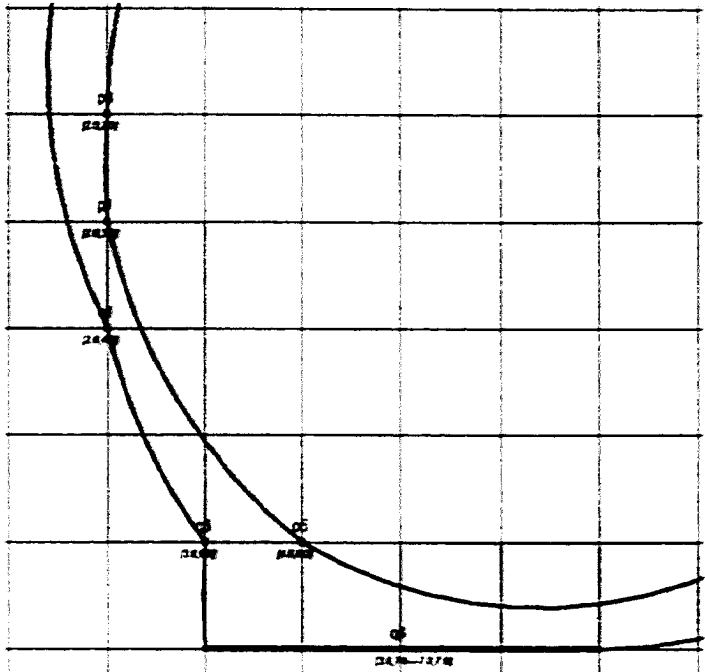


Figure 3.49. q_2 is deleted. After backtracking to q_1 , iteration continues with q_3 .

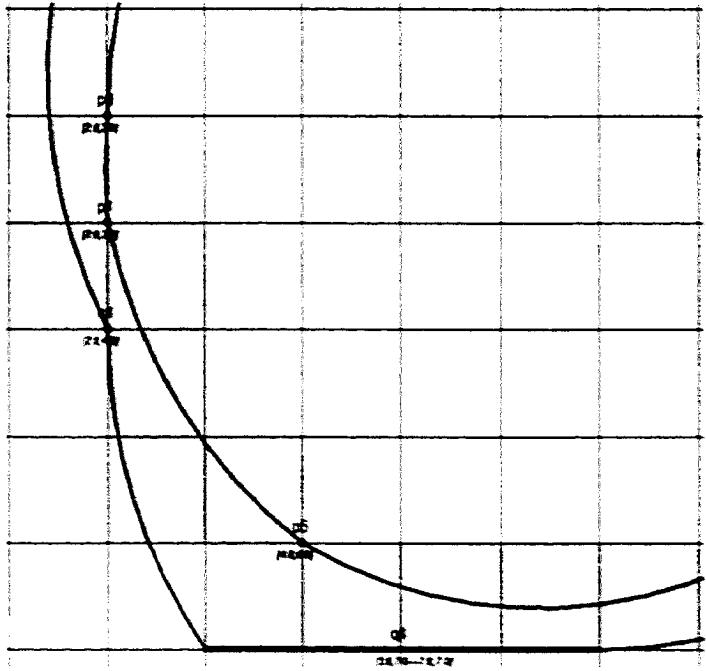


Figure 3.50. q_3 is deleted. The start point of edge q_4 must be reset to the tangent of the circle defined by q_1 and q_4 .

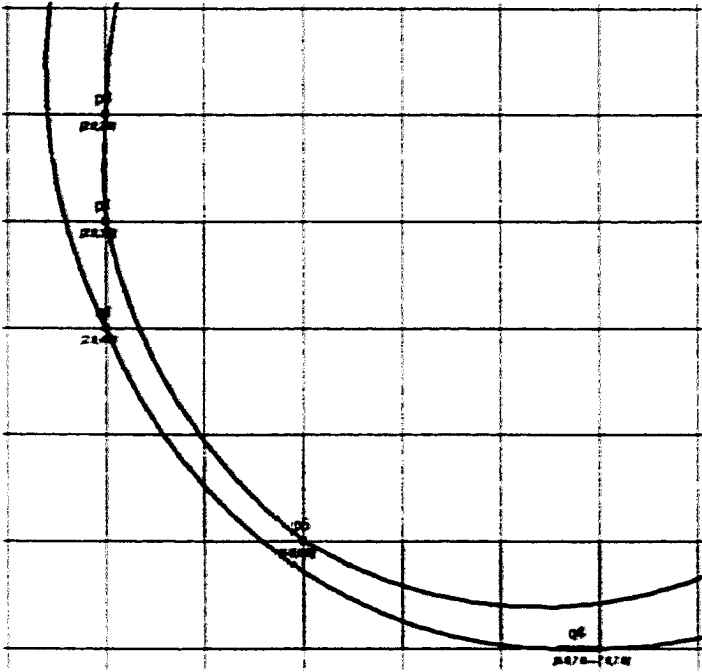


Figure 3.51. The start point of q_4 is reset and edge q_4 is not deleted. Its start point is simply reset. Iteration proceeds with p_0 .

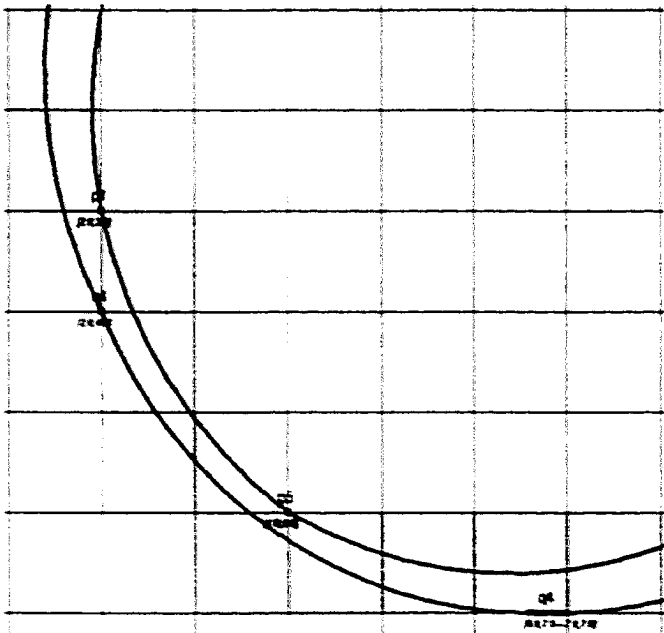


Figure 3.52. p_0 is deleted causing backtracking to q_4 .

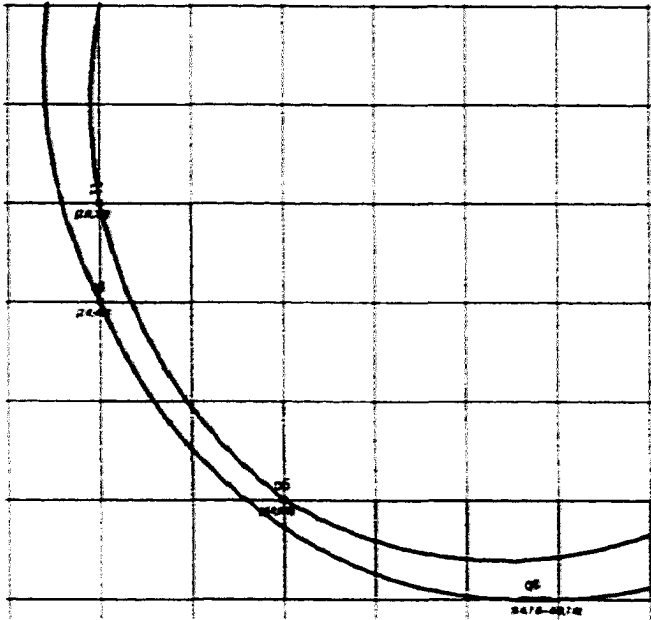


Figure 3.53. Although q_4 is not deleted, its end point must be reset to the tangent of the circle defined by q_4 and p_0 . Iteration continues with p_1 , which is not deleted.

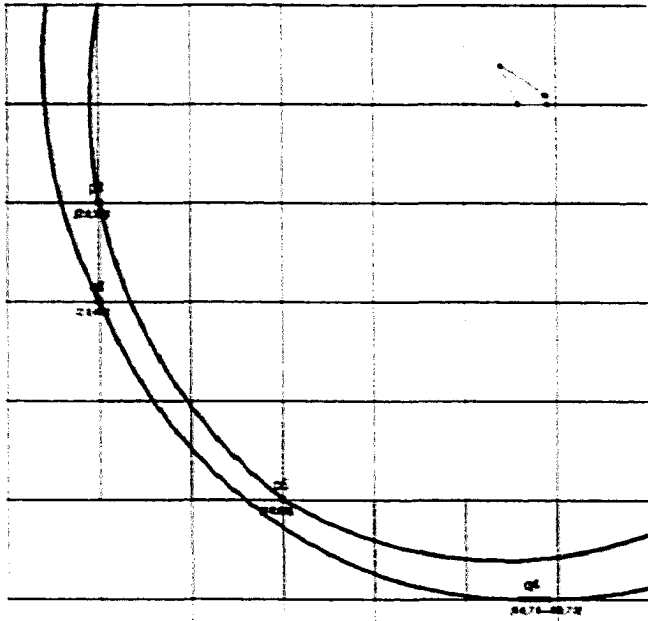


Figure 3.54. The complete passing area is constrained by only four elements: p_1 , p_5 , q_1 and q_4 . The domain is shown in the upper right corner.

3.6. Extensions and Further Directions

This section investigates further extensions (sections 3.6.1, 3.6.2 and 3.6.3) and directions (section 3.6.4) of the algorithm and underlying methodology, introduced in this research.

3.6.1 Determining the Passing Area For All Radii

This paper has assumed that, *a priori*, the digital curve was the digitization of a circle of a given curvature, α . Previous sections demonstrated that the passing area for that digital curve could be expressed as a path of line segments and vertices connected with circular arcs of radius $1/\alpha$. If the radius changes, the passing area also changes. One change that occurs is the size of the radius of the circular arcs connecting the elements on the passing area perimeter. For example, in figure 3.55 elements $\{p_1, p_3, p_4, q_0, q_1, q_3\}$ are on the passing area perimeter in both (a) and (b).

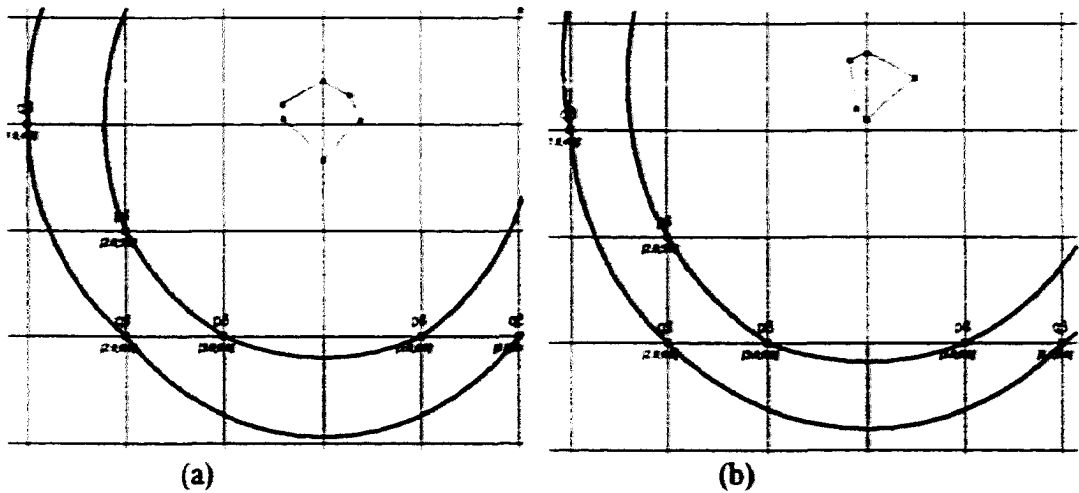


Figure 3.55. The elements along the passing area of the digital curve in (a) are identical to those in (b). The difference between them is the value of the radius of the circular arcs joining the elements. In (a) and (b) the radii are respectively 2.6 and 2.8.

Edelsbrunner (1987) observed that, when the (strict) α -hull of a point set, S , is taken, only certain points in S are found along the α -hull perimeter. As the value of α changes these perimeter points may also change. In chapter 5 it will be shown that, as in a similar fashion studied here, the elements (which include points and line segments) along the passing area (α_x -hull (S_P , S_Q) perimeter) change with α in a predictable manner. This is illustrated in figure 3.56 where the elements on the passing area have changed from figure 3.55.

Worring (1993), applying these results to the points along the passing area, referred to those points as relevant points and presented an $O(n^2)$ algorithm that

determines those points for all passing areas. O'Rourke (1986) has shown that using linear programming it is possible to determine in linear time the minimum radius circular arc that will digitize to a given digital circle. Based on the research presented in this chapter, the α -values where passing area perimeter changes occur are computable from an initial value in $O(n)$ for each such α value. Additionally, it appears from experimental simulations that the number of α values where the critical elements change is bounded by 25, with an average of 10.6.

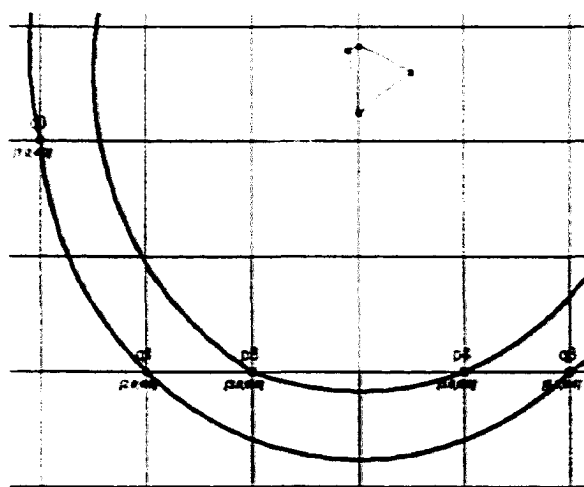


Figure 3.56. When the radius is 3.0, element p_2 is no longer on the passing area. The values at which a change in radius induces a change in the elements on the passing area is necessary to determine the passing area for all curvatures.

The algorithm for computing the domain for all radii may then be divided into two components: (1) determining the initial minimum α value and (2) determining iteratively

each subsequent α value. Step 1 is achievable in $O(n)$ using the O'Rourke method. Also, each iteration of step 2 is achievable in $O(n)$ as will be discussed in chapter 5. Since the number of iterations is bound by a constant, the entire algorithm has a linear run time.

3.6.2 Disconnected Components and 8-Connected Curves

The algorithm may be used to compute the passing area and domain of disconnected digital curves as well. See figure 3.57a where the elements along the inner and outer boundaries of all the digital curve's (disconnected) components are labeled. The algorithm iterates through all the elements shown, in order counterclockwise, using the algorithm of section 3.4.2. Figure 3.37b illustrates the resultant α_x -hull and domain. Note that the passing area and domain are both significantly larger than those of figure 3.56 since with fewer pixels there are fewer constraints.

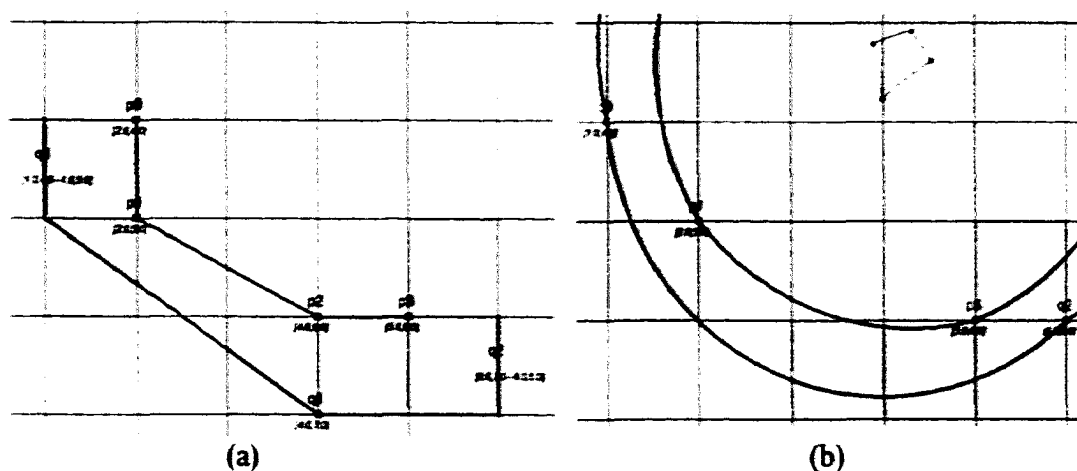


Figure 3.57. The vertices of the disconnected curve are connected to form an inner and outer boundary of the entire curve (a). The passing area (b) is then computed using the α_x -hull algorithm described above. $\alpha = 1/3$.

When each component of a disconnected curve contains two or more pixels the inner and outer boundaries of each component are the same as those of a connected curve. However, when components contain a single pixel the inner boundary is the pixel corner closest to the interior. The outer boundary is, similarly, the pixel corner furthest from the interior, shown in figure 3.58.

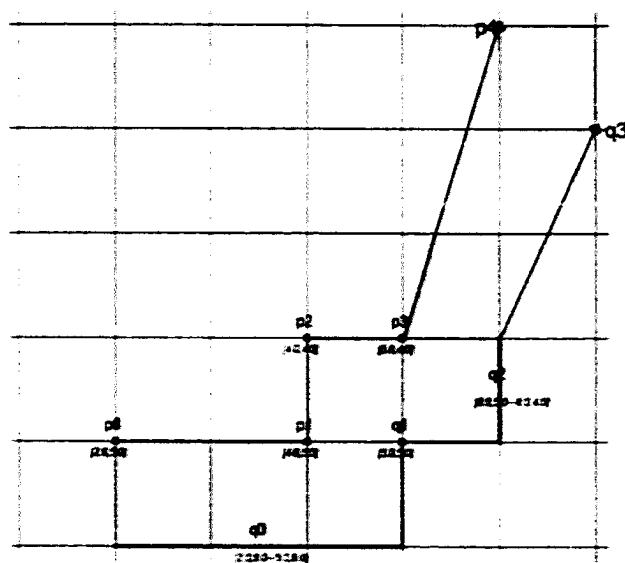


Figure 3.58. The pixel corners closest and furthest from the circle interior mark the inner and outer boundaries of a single pixel component.

The passing area and domain of an 8-connected digital curve may be determined in a similar manner as a 4-connected disconnected curve. The entire 8-connected curve may be interpreted as a disconnected 4-connected curve, as shown in figure 3.59a. The passing area may then be determined as discussed above, as displayed in figure 3.59b.

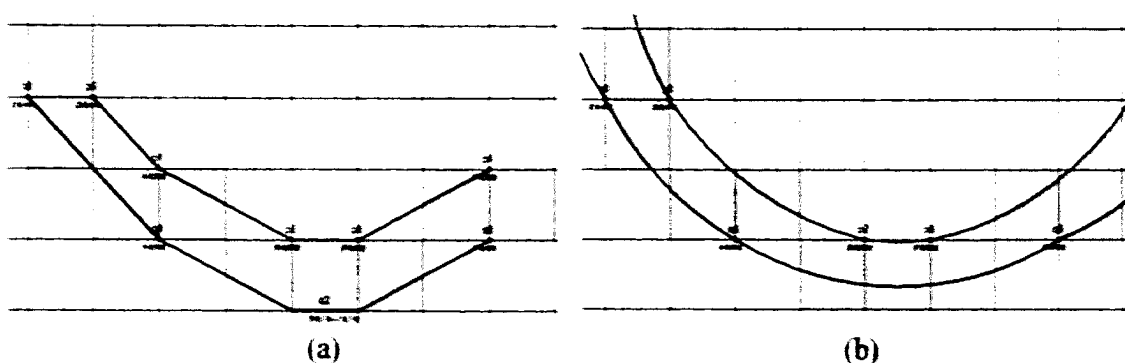


Figure 3.59. The vertices of the 8-connected digital curve are connected to form an inner and outer boundary (b). The passing area of this figure is demonstrated where $\alpha = 1/5$.

3.6.3 Inherent Parallelism

The computation of the convex hulls of each disconnected component (section 3.6.2) can be accomplished independently using the methods discussed in section 3.4, and thus can be executed in parallel. Each component can then be connected together head to tail identically to the connections between components as was shown in figure 3.57 of section 3.6.2.

Assuming m components of equal size with a total of n vertices, the α_∞ -hull of each component can be computed in $3n/m$ iterations. Solving in parallel, with m processors, each processor may compute the infinite hull of one component. Let the maximum number of elements in the passing area be MAX . Then, each processor may compute the infinite hull of two components, with an upper bound of $6 MAX$ iterations. The number of processors used, for this step is $m/2$ because each processor is now processing 2 components. Each successive step uses half the processors of the previous step until a single processor computes the infinite hull. The efficiency of such a parallel algorithm is $3n/m + 6 MAX \log_2(m)$.

The maximum number of elements on the passing area, defined here as MAX , appears to have an upper bound (from empirical evidence). After running numerous simulations the maximum number of elements was found to be 14. What needs clarification is whether in fact this is the upper bound and if it is what the mathematical proof may be. The domain can be computed in parallel in constant time from the information of the passing area.

Further efficiency can be obtained by realizing that only about half the inner boundary vertices need to be considered during the α -hull computation. Moving counter-clockwise along the inner boundary some edges follow a left direction and some follow a right direction. Only the vertices that are the starting point of a left-moving edge contribute to the α -hull since they are in a locally convex region (shown as darkened vertices in figure 3.60). Determining which vertices are locally convex though has the same run time complexity as determining whether they are on the α -hull perimeter, and therefore no savings is achieved by first eliminating them.

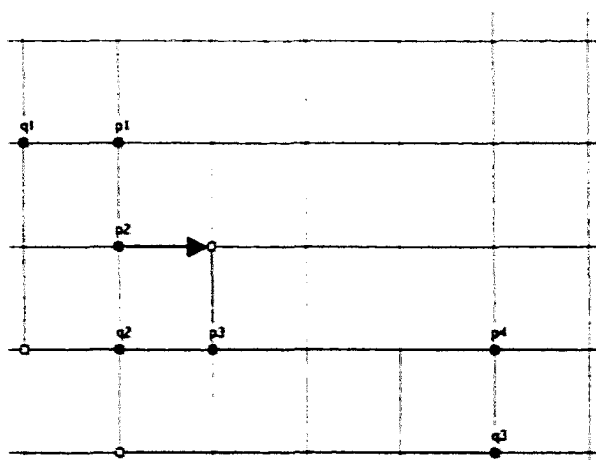


Figure 3.60. Moving along the inner hull counter clockwise (from p_1 to p_2), only vertices that follow a left direction (e.g. p_2) contribute to the α -hull. Therefore, only the labeled, darkened vertices contribute to the α -hull.

3.6.4 Further Research Directions

The definition of the infinite hull, the computational geometry structure introduced in this research paper, is described in terms of general point sets. Thus, the algorithm should be directly extendible to find the three (and n -) dimensional domain of a digital hyper-planar and spherical surface. That digital surface can be decomposed into upper and lower surface boundaries from which the infinite hull is computed. Each vertex is inspected for inclusion in the infinite hull against all its neighbors, which were just two vertices in the two-dimensional case. As with 2D digital lines, the infinite hull may be computed of disconnected digital surfaces and each component can be solved separately in parallel. However, the iteration of the boundary vertices through the upper and lower surfaces must be defined differently. With two-dimensional digital arcs the sequence of iteration and backtracking is straightforward since each element is connected to exactly one previous and one successive element. With three (or more) dimensions, each element is connected to at least four elements and, therefore, a different approach must be used. This is a topic that is currently being investigated. In addition, it is not as clear if the linear runtime complexity will still hold.

The foundation of the method discussed in this research is that extensions of convex hulls of a point set can be used to determine a digital curve's passing area. It can be conjectured that the algorithm would yield the same results for any curve for which a hull can be computed. Hulls may be defined for certain conics as well as any convex shape. Using such a framework it seems plausible to extend the results discussed in this

paper to conical, as well as more complex, digital curves. If this directly follows, then these extensions, too, could possibly be computable in linear time.

Finally, how the techniques developed in this research paper can be applied to finding the passing area of a digital curve found in gray scale images may be further investigated (chapter 6). The level of gray for each pixel can be used to set a range within the pixel that the passing area must include.

Chapter Four

Domain of a Digital Circular Arc

Under All Radii

Abstract

An extension of the convex hull, the infinite convex hull, was defined (chapter 2) that was used to determine a digital line segment's minimal passing area. This passing area was then used to compute a digital line segment's domain, the set of all possible lines segments that digitize to the digital line, defined in (m,b) parameter space. Similarly, the α -hull that was defined by Edelsbrunner (1983) was extended to the α -infinite hull (chapter 3) and was used to compute the passing area of any digital circular arc in $O(n)$. Subsequently, the domain may be determined in constant time.

The methodology used for determining the passing area where the radius of the underlying arc is known can be extended to determine the passing area where the radius is unknown. In this chapter, we introduce an algorithm that determines the radius value where the elements along the passing area perimeter change. The radius values where such changes occur can directly be used to find the domain of the digital curve for all radial values. Based on experimental simulations, the computation has an $O(n)$ run time complexity.

4.1. Introduction

The digitization process converts visual data from the world scene into a digital image. Each object in the image is bounded by digital curves that define the object's shape. This chapter considers object boundaries that are (partially) comprised of circular arcs and addresses the following problem: Given a connected digital circular arc represented by a list of connected pixels, find the set of all possible circular curves that digitize to that digital curve. This set is known as the *domain* (Dorst, Smeulders, 1984).

Each circular arc that is a member of the domain may be the original curve that exists in the world scene. The actual curve from the world scene is referred to as the *preimage curve*, or *preimage*. The digitization process causes the inevitable loss of detail and recovering the preimage from the digital curve is therefore impossible. The best possible reversal of digitization is the recovery of the domain; this led some to consider the domain as the pseudo-inverse of the digital curve (Dorst and Smeulders, 1987).

The pixel boundary of a digital circular curve consists of the inner and outer boundaries (figure 4.1). These boundaries may be connected in a linked list so that the head of each boundary is connected to the tail of the other, known as the *joined boundary* (figure 4.2). The intersection of the pixel area (shown in gray in figure 4.1) and the complement of the infinite α -hull (denoted α_{∞} -hull) of the joined boundary is the passing area of that digital circular arc. The perimeter of the passing area consists of vertices and line segments that are connected via circular arcs. The vertices and line segments that define the passing area are referred to as *elements*. A bijection exists between elements and arcs of the passing area boundary and edges and vertices on the domain boundary,

respectively (figure 4.3) An algorithm was demonstrated that computes the passing area in $O(n)$ where n is the number of pixels in the digital curve. (Please see chapter 3 for a full treatment of these topics.)

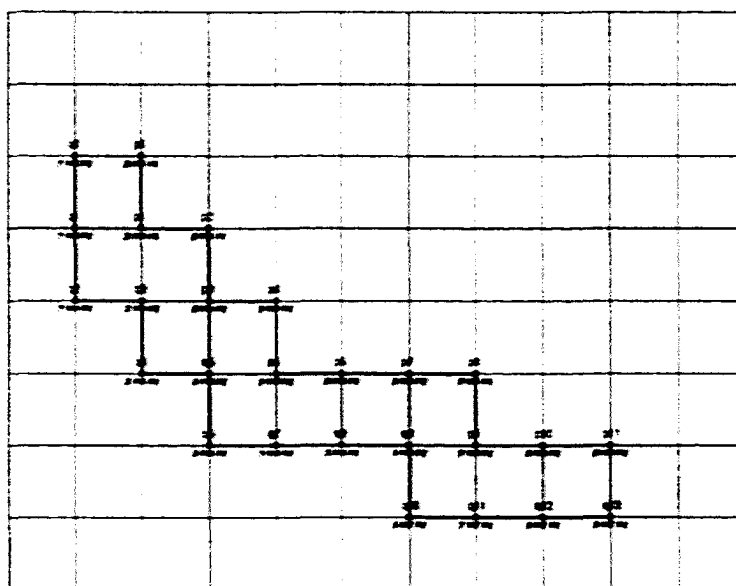


Figure 4.1. The inner ($p_0 - p_{11}$) and outer ($q_0 - q_{13}$) pixel boundaries of a digital circular arc. Note that the first and last pixel's boundaries are not included in the inner or outer boundaries.

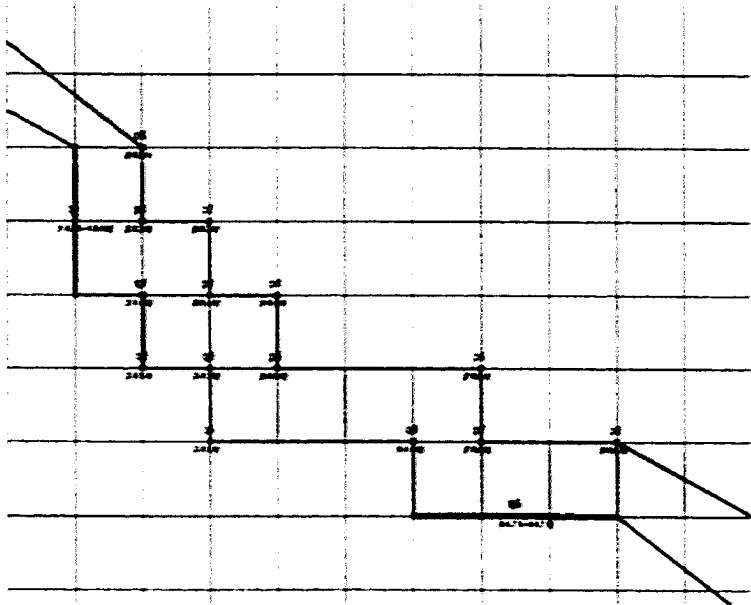


Figure 4.2. The joined boundary of the digital curve. The connectivity of the inner and outer boundaries is illustrated by the line segment extension from p_8 to q_6 and from q_6 to p_0 . The elements consisting of vertices and edges that constrain the passing area are correspondingly labeled.

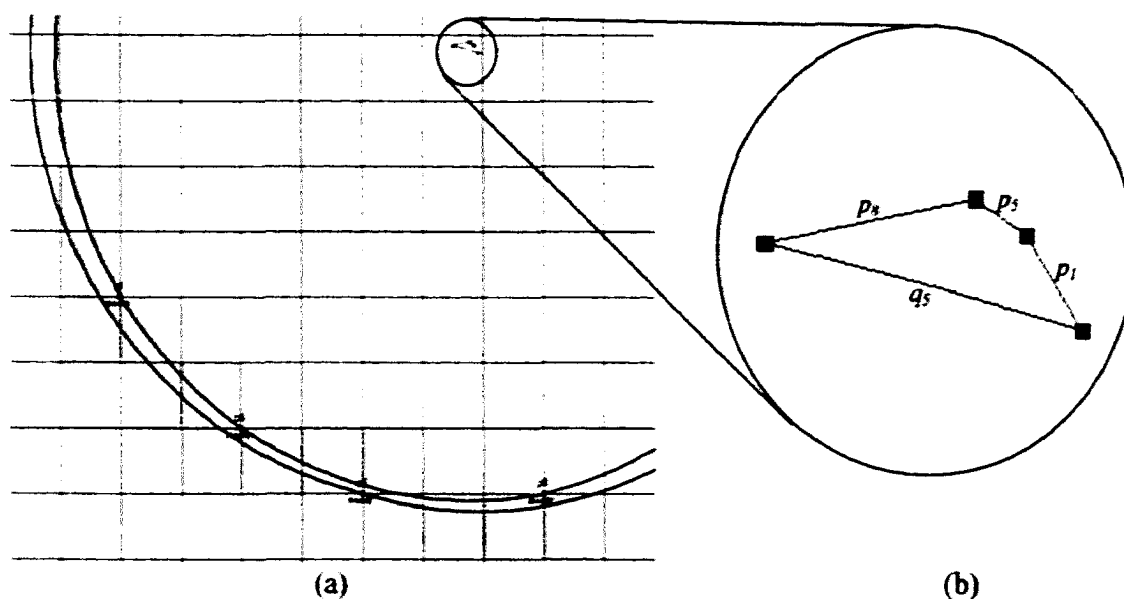


Figure 4.3. The α_x -hull where $\alpha=1/7$ (a) and the domain (b) of the digital curve. Each element in the passing area perimeter corresponds to an arc along the domain perimeter. The element labels from the passing area are used to identify the corresponding arcs in the domain.

In section 4.2 it is shown that, the passing area changes with α in a predictable manner. Specifically, the elements along the passing area perimeter change at specific values of α . We show that by determining the value of α where elements along the passing area perimeter change we may determine the domain of a digital circular arc for all radii. In section 4.3 the algorithm that determines the passing area is presented. In section 4.4 the methodology used for digital circular arcs is extended to handle digital standard elliptical curves where the relative sizes of horizontal and vertical axes are known.

In section 4.5 the shape of the domain under all radii is considered. We shall see that line segments and parabolas bound the domain. This result differs from that of other researchers who bounded the domain with line segments alone. The results found here follow from the assumption that edges along the outer boundary may constrain the passing area, which occurs when the luminescence digitization scheme is followed. Others assumed a simplified mathematical model such as OBQ (object boundary quantization) and therefore found a different domain boundary. We shall also see that the domain of digital elliptical curves is also bounded by line segments and parabolas.

4.2. The Effect of α on the Passing Area

Edelsbrunner (1987) showed that the points comprising the perimeter of the α -hull of a point set S are only from a subset of S . As α changes the perimeter point set also changes. The set of all perimeter point sets, where each perimeter point set is only valid for specific α values, was defined as the *shape spectrum* of S .

Worring (1995) presented an $O(n^2)$ algorithm that determines the points along the inner and outer boundaries of a digital curve that constrain the domain. As discussed in section 2.4, the passing area of a digital curve where the radius is known may be determined in $O(n)$. From empirical evidence (4.3.3) it is evident that the number of passing areas with distinct constraining point sets is bounded by a constant. Therefore, by considering the passing area in terms of the α_∞ -hull, the shape spectrum of the points along the inner and outer boundaries can be determined in $O(n)$. This information yields a

constant time determination of the domain. The effects that the specific α value has on the passing area are presented in this section.

The passing area of a digital circular arc can be determined by computing the α_x -hull of the inner and outer boundaries of the pixel area. For a given digital curve c , the actual passing area will change with the value of α , which is set to $1/r$ where r is the radius of the preimage circular arc that digitized to c . As α changes, the passing area will also change, as shown in figure 4.5 (next page). Therefore, the passing area of a digital arc may be determined by computing the α_x -hull only if the size of the radius of the preimage arc is known, *a priori*.

The discussion now focuses on how the passing area may be computed where r is not known. The smallest and largest values for which a passing area exists are denoted r_{min} and r_{max} , respectively. Since the α_x -hull of the digital curve's boundary is different for each r , $r_{min} \leq r \leq r_{max}$, for any two different values of r the passing area will be different. Thus, explicitly computing the passing area for each r in the valid range is unfeasible, since the passing areas of an infinite number of radius values must be computed. In the next section the methodology for the *implicit* computation of the passing areas under all radii is discussed.

4.2.1 Trivial and Critical Changes

Changes to the passing area due to the change in α may be categorized as either *trivial* or *critical* changes. A *trivial change* in the passing area is when, with a change in r , the elements along the passing area perimeter remain the same and it is only the arc

radius that connects the elements that changes. Since the arc radius that connects elements along the passing area equals r , trivial changes occur with every change in r . See figure 4.5a, b where the passing area and domain for radial values 2.6 and 2.9 are shown. Although the shapes of the passing area and domain for the two differ, the elements along the passing area perimeter remain the same.

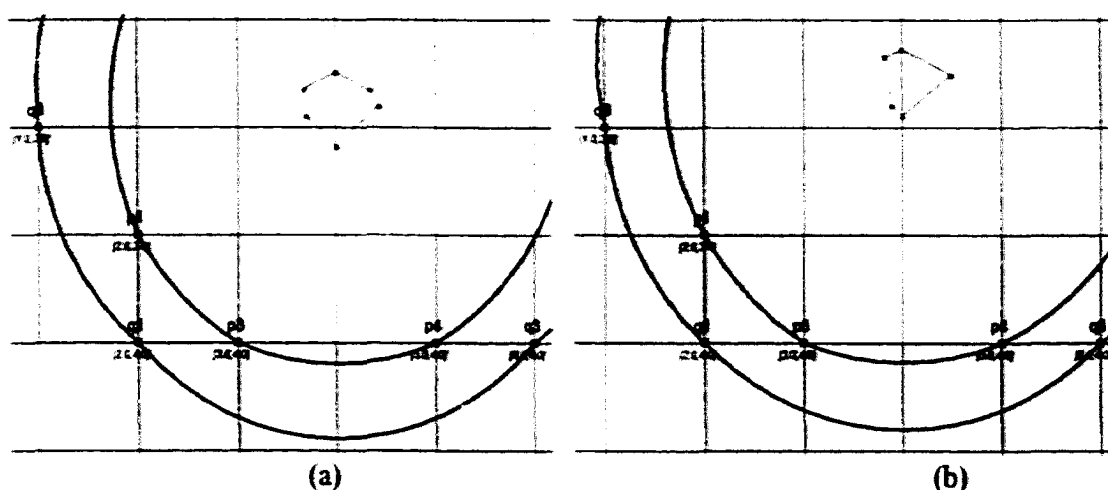


Figure 4.5. The radial values shown are 2.6 (a) and 2.9 (b). The change in the passing area is *trivial* because the elements along the passing area perimeter did *not* change; only the circular arcs connecting the elements changed radial values from 2.6 to 2.9.

A **critical change** occurs, when, in addition to the arc radius changing, the elements along the passing area perimeter also change. In figure 4.6a, r increases from 3.0 to 3.1 and element q_1 , which is present on the passing area perimeter in (a), is not found in (b). Critical changes can occur when an element(s) leaves the passing area (as in

figure 4.6) or when an element(s) enters the passing area. Likewise, consider figure 4.7 where element p_3 and q_3 are not present in the passing area when $r = 2.2$ but are present when $r = 2.3$. In distinction from trivial changes that occur with every change in r , critical changes only occur at specific radial values. The next section studies qualitatively for which radial values those changes occur and presents a framework to determine quantitatively the actual radius values for which critical changes occur.

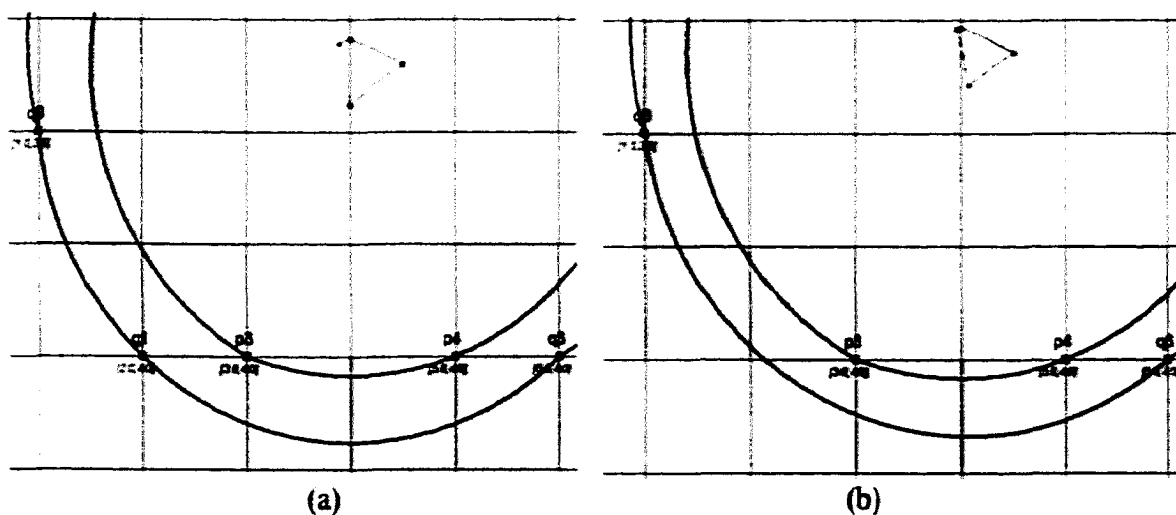


Figure 4.6. The radial values shown are 3.0 (a) and 3.1 (b).

The change in the passing area is *critical* because the elements along the passing area perimeter *did* change; q_1 that is present in (a) is not present in (b).

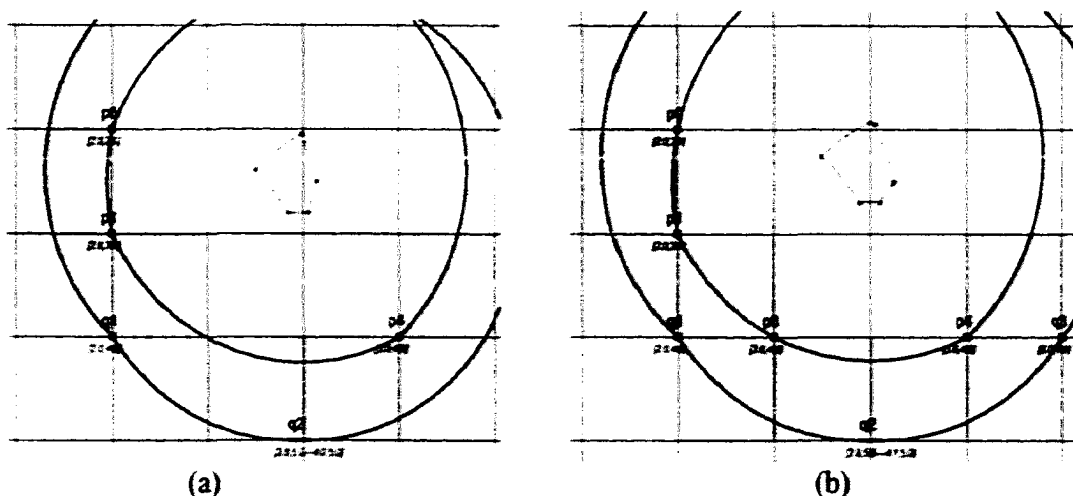


Figure 4.7. The radial values shown are $r = 2.2$ (a) and $r = 2.3$ (b). When $r = 2.2$ (left), p_3 and q_3 are not in the passing area. However, when $r = 2.3$ (right), p_3 and q_3 are present. The precise radial value where p_3 and q_3 enters the passing area is 2.24.

4.2.2 Critical Points, Sets and Ranges

At every critical change at least one (critical) element will either enter or leave the passing area. After determining the radial values where critical changes occur as well as the elements that enter or leave the passing area for those radial values, the collective passing area under all radii is thus determined. For circular arcs, the parameter space from which all digital curves are generated is comprised of the circular arc radius value r . Therefore, element e is a critical element if, and only if, e is along the passing area perimeter for some value of r .

This concept is formalized with the following definitions.

Definition. A **critical element** e is an element on the passing area perimeter for some parameter space value.

Definition. A critical element's **critical range**, denoted $\langle e \rangle$, is the range of the parameter space where a critical element actively constrains the passing area.

For example, a critical element with respect to circular arcs will constrain the passing area only within a certain range of radius values. (Worring (1995) referred to points that constrain the domain as relevant points.) In order to avoid ambiguity at the endpoints of this range, the convention used here is that the minimum range value, denoted $\langle e \rangle_{min}$, is the closed interval and the maximum range value, denoted $\langle e \rangle_{max}$, is the open interval such that $\langle e \rangle = [\langle e \rangle_{min}, \langle e \rangle_{max})$.

Definition. For any value in the parameter space where a passing area exists for the digital curve, only certain critical elements constrain the passing area. The **critical set** for a given parameter value is the set of critical elements whose critical ranges include that value.

For example, in figure 4.7a-b, the critical set when $r = 2.2$ is $\{p_0, p_1, p_4, q_1, q_2\}$ and when $r = 2.3$ is $\{p_0, p_1, p_3, p_4, q_1, q_2, q_3\}$. Each element in the critical set is present on a passing area only within the element's critical range. The critical range of element q_2 , for example, is $[2.0, 2.5)$ since it is found on the passing area only when r is within that

range. See figure 4.7 where q_2 is on the passing area ($r = 2.2$ and $r = 2.3$ in figures a and b, respectively). However, when $r = 1.9$ or $r = 2.7$, q_2 is not on the passing area as shown in figure 4.8.

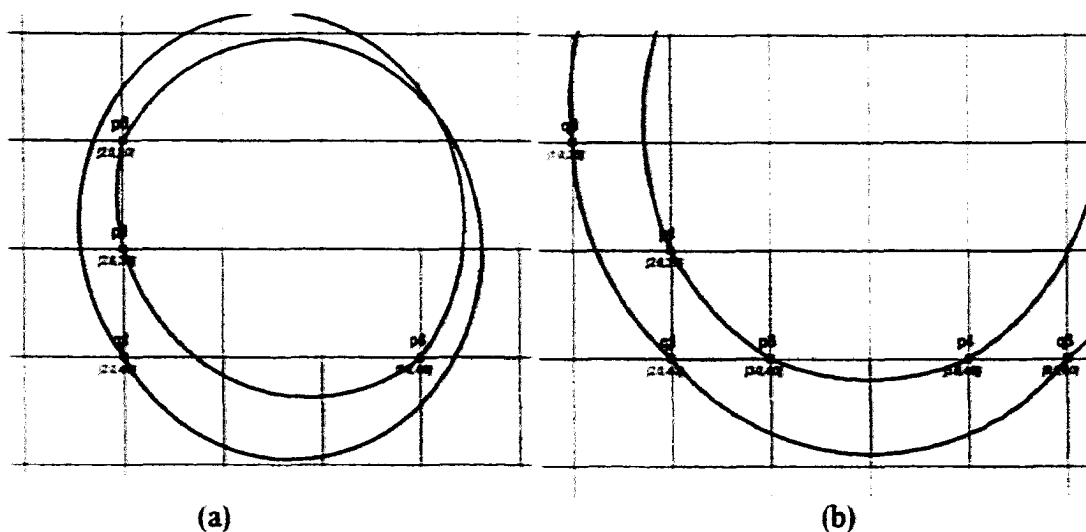


Figure 4.8. When $r = 1.9$ (a) or when $r = 2.7$ q_2 is not on the passing area since the critical range of q_2 is $[2.0, 2.5)$.

Every labeled element along the passing area in figure 4.9 is a critical element except for p_2 since, at some value of r , each element, except p_2 , will be on the passing area.

The table in figure 4.10 identifies the critical sets and their critical ranges for the digital curve found in figure 4.9. The passing area corresponding to each critical set is illustrated in figures 4.11-4.20.

	CS	$\langle CS \rangle_{min}$	$\langle CS \rangle_{max}$	Figure	Value of r in figure
1)	p_0, p_4, q_1	1.80	1.90	5.11	1.85
2)	p_0, p_1, p_4, q_1	1.90	2.00	5.12	1.95
3)	p_0, p_1, p_4, q_1, q_2	2.00	2.24	5.13	2.1
4)	$p_0, p_1, p_3, p_4, q_1, q_2$	2.24	2.25	5.14	2.245
5)	$p_0, p_1, p_3, p_4, q_1, q_2, q_3$	2.25	2.30	5.15	2.27
6)	$p_1, p_3, p_4, q_1, q_2, q_3$	2.30	2.50	5.16	2.4
7)	$p_1, p_3, p_4, q_0, q_1, q_3$	2.50	2.92	5.17	2.7
8)	p_3, p_4, p_0, q_1, q_3	2.92	3.01	5.18	3.0
9)	p_3, p_4, q_0, q_3	3.01	3.16	5.19	3.1
10)	p_3, q_0, q_3	3.16	3.81	5.20	3.5

Figure 4.10. The 10 critical sets and their critical ranges.

Each critical set is illustrated with a figure (4.11 - 4.20)

where r is within the critical range.

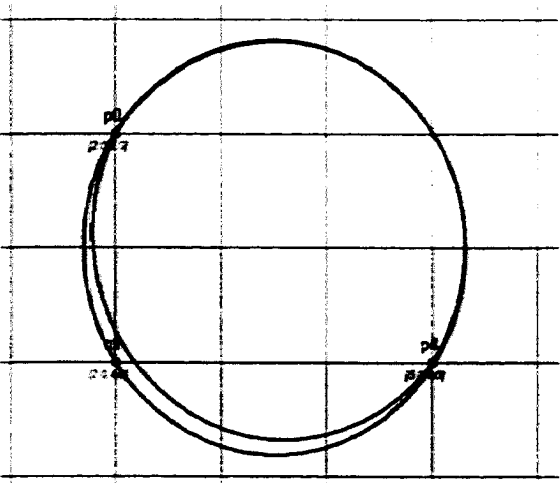


Figure 4.11. $r = 1.85$

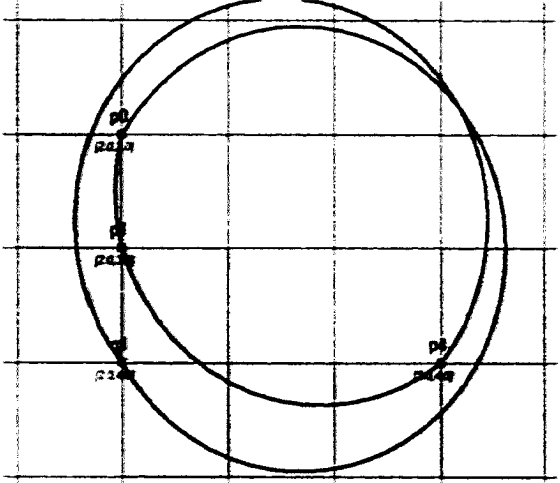


Figure 4.12. $r = 1.95$

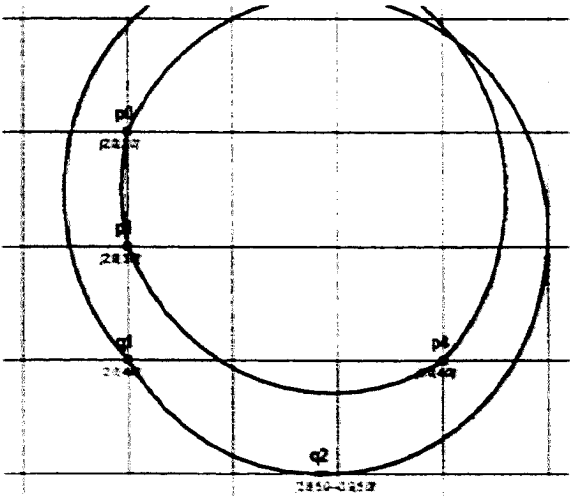


Figure 4.13. $r = 2.1$

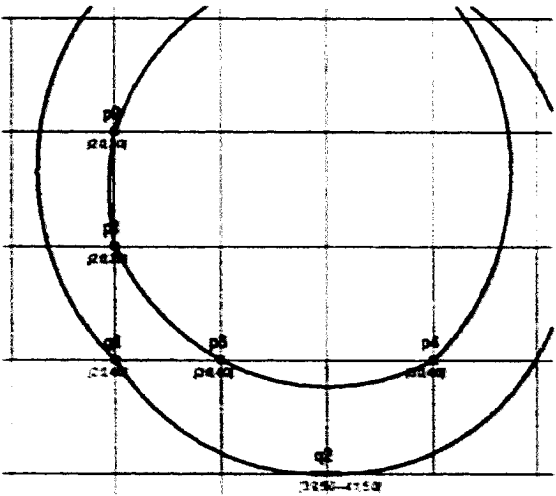


Figure 4.14. $r = 2.245$

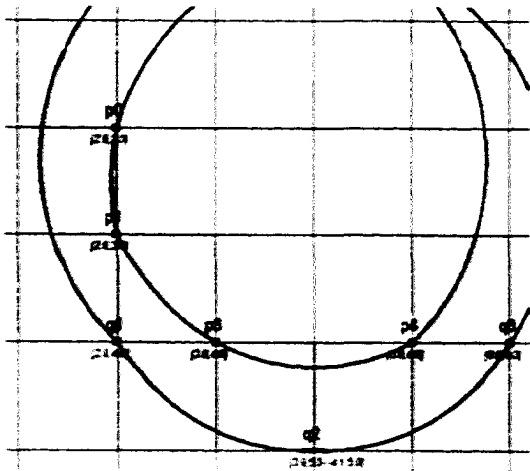


Figure 4.15. $r = 2.27$

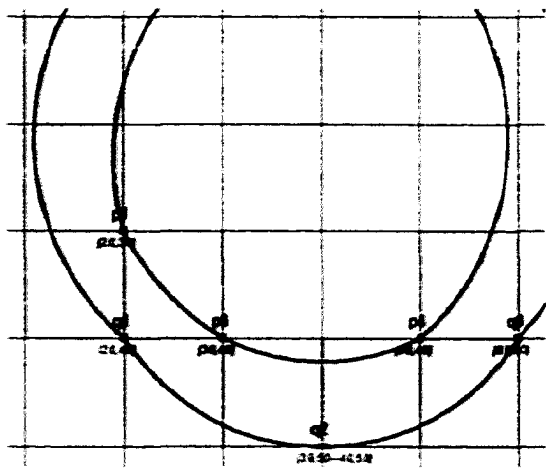


Figure 4.16. $r = 2.4$

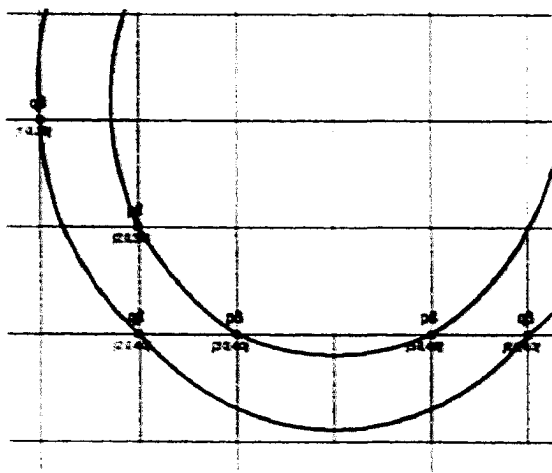


Figure 4.17. $r = 2.7$

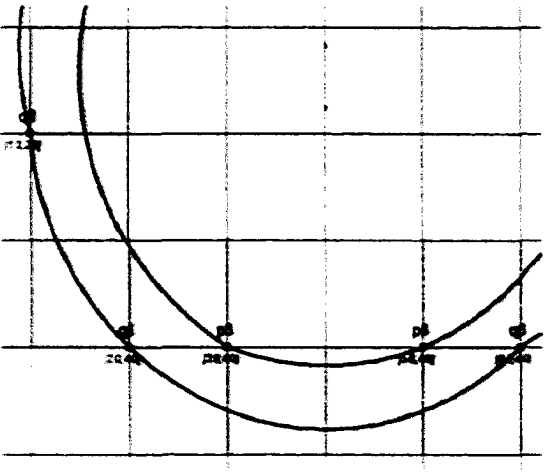
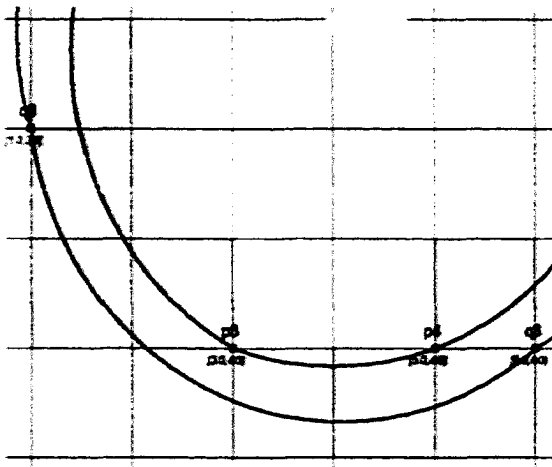
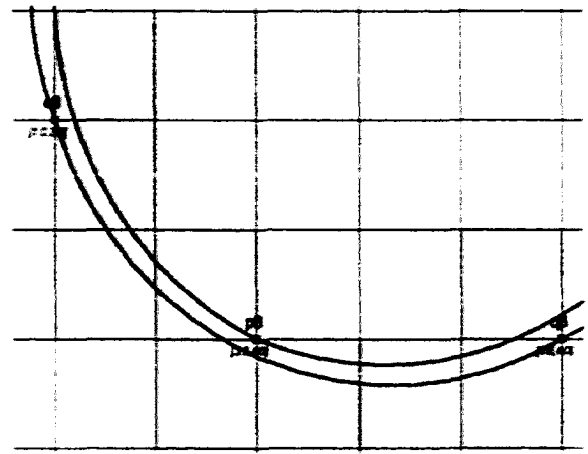


Figure 4.18. $r = 3.0$

Figure 4.19. $r = 3.1$ Figure 4.20. $r = 3.5$

4.2.3 Computing the Critical Sets

The critical sets and their ranges contain all the information necessary to construct the passing area for all radii. The elements of a critical set CS are the only elements found along the passing area perimeter where $r \in \langle CS \rangle$. The circular arc joining the elements have radius r . As r increases to another value in $\langle CS \rangle$, the elements will remain the same and only the joining arc radius will change, defined in section 4.1 as a trivial change. When r exceeds $\langle CS \rangle$, a critical change occurs and the elements along the passing area perimeter are modified.

In figure 4.6 above, element q_1 exited the critical set at some radial value between 3.0 and 3.1. Notice that when $r = 3.1$ the arc that subtends q_0 and q_3 does not include q_1 . As r decreases, that arc gets closer to q_1 until, when $r \approx 3.011$, that arc intersects q_1 . When r is less than ≈ 3.011 , q_1 is on the passing area, as in figure 4.6a. The critical change will occur at the radius value of the circle intersecting q_0 , q_1 and q_3 . By determining the radius values at which individual elements enter and leave the critical set,

the critical ranges of the critical sets that they are members of can be determined since $\langle CS \rangle = \bigcap_{e \in CS} \langle e \rangle$. These observations are now formalized by defining the conditional critical range of an element.

4.2.3.1 Conditional Critical Range

Definition: For critical element e and critical set CS , where e may or may not be a member of CS the *predecessor element* of e is the closest preceding element along the joined boundary (see section 3.2.1) that is also a critical element in CS . This will be denoted by e^- .

Definition: For critical element e and critical set CS , where e may or may not be a member of CS the *successor element* of e is the closest successive element along the joined boundary that is also a critical element in CS . This will be denoted by e^+ .

For example, in figure 4.18, where $CS = \{p_3, p_4, p_0, q_1, q_3\}$ it is clear by inspection that q_1^- and q_1^+ are q_0 and q_3 , respectively. Similarly, q_3^- and q_3^+ are q_1 and p_3 , respectively, since the inner and outer hulls are connected. However, since q_2 is not on the passing area in figure 4.18, the values of q_2^- and q_2^+ are based on the previous and successive neighbors of q_2 if it was placed on the passing area. Placing q_2 on the passing area, it would appear between q_1 and q_3 . Thus, q_2^- and q_2^+ are q_1 and q_3 , respectively.

Definition: The *conditional critical range* of e with respect to e^- and e^+ , denoted $\langle e | e^-, e^+ \rangle$ is the range of parameter space values for which $e \in CS$ assuming e^- and $e^+ \in CS$. This is the constraint that e^- and e^+ place on e in determining when e will be on the passing area.

Again, in figure 4.18 above, q_0 and q_3 are the previous and successive elements to q_1 along the passing area. The constraints that q_0 and q_3 impose on q_1 in determining the critical range of q_1 is $\langle q_1 | q_0, q_3 \rangle = [0, \sqrt{9.065}) \approx [0, 3.01)$. As mentioned above the maximum range value $\langle q_1 | q_0, q_3 \rangle_{\max} = \sqrt{9.065}$, the maximum radius of a circle intersecting q_0, q_1 and q_3 . The minimum range, $\langle q_1 | q_0, q_3 \rangle_{\min}$ is 0 because q_1 will still be on the passing area at the smallest radial value. Since q_1 is on the outer boundary, q_1 will be along the α -hull of the outer boundary even where r is very small. See figure 4.21 where the α -hull of the outer boundary is shown for $r = 1$ ($\alpha = 1/r = 1$).

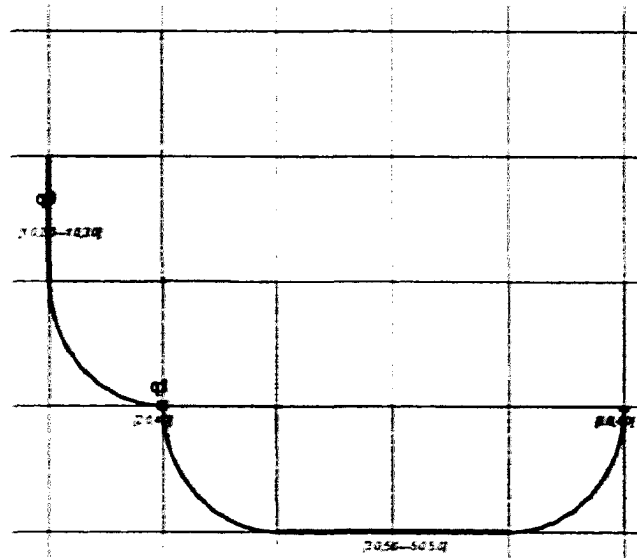


Figure 4.21. The α -hull of the outer boundary will include q_1 even at trivially small r values. In essence, q_0 and q_3 do not constrain q_1 at its minimum critical range bound.

Consider figure 4.14 where p_3 enters the passing area when $r = \sqrt{5}$. The value of $\langle p_3 \mid p_1, p_4 \rangle$ is $[\sqrt{5}, \infty) \approx [2.237, \infty)$. When r is less than $\sqrt{5}$, p_3 will not be on the passing area since p_3 will be located within the arc subtending p_1 and p_4 . The value of $\langle p_3 \mid p_1, p_4 \rangle_{\max}$ is ∞ since the α -hull of p_1, p_3 and p_4 will include p_3 on the α -hull perimeter even when $\alpha = 1/\infty = 0$, as shown in figure 4.22.

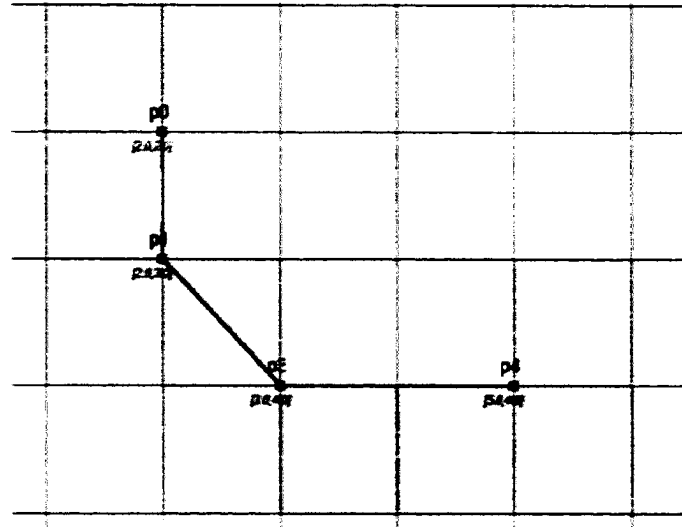


Figure 4.22. The α -hull of the inner boundary will include p_3 even when $r = \infty$. Thus, p_1 and p_4 do not constrain p_3 at its maximum range bound.

The application of the conditional critical range concept is to determine all of the critical sets. Let the critical set with minimum critical range be CS_1 and successive critical sets be denoted CS_2 , CS_3 and so forth. Assuming CS_1 is found, the conditional critical range may be used to determine at which radial values critical elements enter and exit CS_1 . To find the radius value where a critical change occurs, it is necessary to determine the r value where elements in CS_1 exit CS_1 , denoted $r_{\text{exit}}(CS_1)$, and the r value where elements not in CS_1 enter the critical set, $r_{\text{enter}}(CS_1)$. The new critical set, CS_2 , will contain the elements from CS_1 in addition to the elements that entered the critical set and excluding the elements that left CS_1 .

To determine when an element present in the critical set CS_1 will exit we must find $r_{\text{exit}}(CS_1) = \min_{e \in CS_1} \langle e | e^-, e^+ \rangle_{\text{max}}$. The element(s) $e \in CS_1$ such that $\langle e | e^-, e^+ \rangle =$

$r_{exit}(CS_I)$ will be the first element in CS_I to exit. To determine when an element, $e \notin CS_I$, will enter CS_I we must determine $r_{enter}(CS_I) = \max_{e \in CS_I} \langle e | e^-, e^+ \rangle_{\min}$. At this value, element e where $\langle e | e^-, e^+ \rangle = r_{enter}$ will enter the passing area.

Thus, the r value where CS_I will undergo a critical change is the minimum of r_{exit} and r_{enter} . At this radial value either an element in CS_I will exit or an element not in CS_I will enter causing a critical change in CS_I . The new critical set, CS_2 will include the elements of CS_I in addition to any elements that entered and excluding any elements that exited.

4.2.3.2 Computing the Conditional Critical Range

Earlier in figure 4.14 the conditional critical range of p_3 is $\langle p_3 | p_1, p_4 \rangle \approx [2.24, \infty)$ since p_3 enters the α -hull at $r = 2.24$ and remains on the α -hull *ad infinitum*. Thus, where e, e^- and e^+ are on the inner boundary $\langle e | e^-, e^+ \rangle = [r_{circle}(e^-, e, e^+), \infty)$ where $r_{circle}(e^-, e, e^+)$ is the radius of the circle intersecting elements e^-, e and e^+ . This is referred to as the **strict inner boundary rule**.

This rule is derived from the observation that as α increases the number of points along the alpha hull perimeter decreases. Thus, when $\alpha = \infty$ (figure 4.22) p_0, p_1, p_3 and p_4 are on the α -hull of the inner boundary. As α increases points leave (figures 4.17-4.20) the α -hull. When an element e is found inside some circle defined by two perimeter elements and α, e will not be found on the α -hull. Clearly, the first α value when this occurs is when the element is precisely on the circle defined by two perimeter elements

and α . Assuming only that elements e^- and e^+ are on the α -hull this radius value is clearly equals $r_{\text{circle}}(e^-, e, e^+)$.

In distinction, in figure 4.6 the conditional critical range of q_1 is $\langle q_1 | q_0, q_4 \rangle \approx [0, 3.01)$ since q_1 is on the α -hull of the outer boundary when $r = 0$ and remains on the α -hull until $r \approx 3.01$. When e, e^- and e^+ are on the outer boundary, $\langle e | e^-, e^+ \rangle = [0, r_{\text{circle}}(e^-, e, e^+))$. This situation is referred to as the **strict outer boundary rule**.

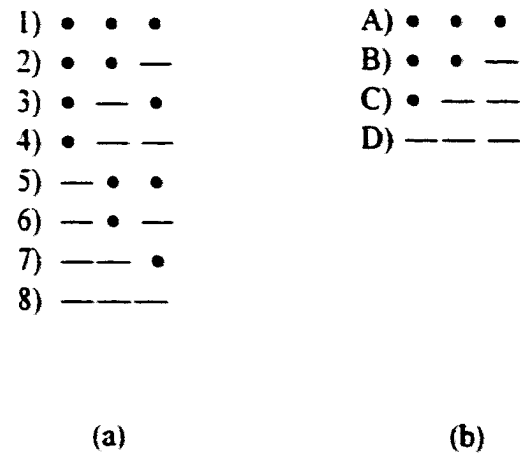
The value of the conditional critical range when elements e^-, e and e^+ are on different boundaries is now presented. See figure 4.17 where element p_1 is on the passing area. p_1 will leave the passing area when $r = r_{\text{circle}}(q_3, p_1, p_3) \approx 2.92$, as illustrated in figure 4.18. The conditional critical range of p_1 with respect to q_3 and p_3 , its previous and successive neighbors along the critical set of figure 4.17, is $\langle p_1 | q_3, p_3 \rangle = [0, r_{\text{circle}}(p_1, p_3, q_3)) \approx [0, 2.92]$. Thus, when e is along the inner boundary and either e^- or e^+ are on the outer boundary $\langle e | e^-, e^+ \rangle = [0, r_{\text{circle}}(e^-, e, e^+))$. This is the **loose inner boundary rule**.

Conversely, observe that q_3 is not present along the passing area boundary in figure 4.14 ($r = 2.245$) but is present in figure 4.15 ($r = 2.27$). The value of $\langle q_3 | q_2, p_0 \rangle \approx [2.25, \infty]$. In general, then, when e is on the outer boundary and either e^- or e^+ are on the inner boundary $\langle e | e^-, e^+ \rangle = [r_{\text{circle}}(e^-, e, e^+), \infty)$. This is the **loose outer boundary rule**. The table in figure 4.23 summarizes the criteria for determining the values of $\langle e | e^-, e^+ \rangle$ in all four situations.

Boundary of e	Are e^- and e^+ on the same boundary as e ?	$\langle e e^-, e^+ \rangle$	Boundary Rule
inner	Y	$[r_{\text{circle}}(e^-, e, e^+), \infty)$	strict inner
inner	N	$[0, r_{\text{circle}}(e^-, e, e^+))$	loose inner
outer	Y	$[0, r_{\text{circle}}(e^-, e, e^+))$	strict outer
outer	N	$[r_{\text{circle}}(e^-, e, e^+), \infty)$	loose outer

Figure 4.23. The criteria for applying the appropriate boundary rule to determine the conditional critical ranges

Earlier, in this section, $r_{\text{circle}}(e^-, e, e^+)$ was defined as the radius that intersects elements e^- , e and e^+ . Since an element may be either a vertex or edge (i.e. line segment) along the joined boundary, there are eight possibilities for the identities of the three elements. Eliminating redundant combinations only four remain, as shown in figure 4.24.



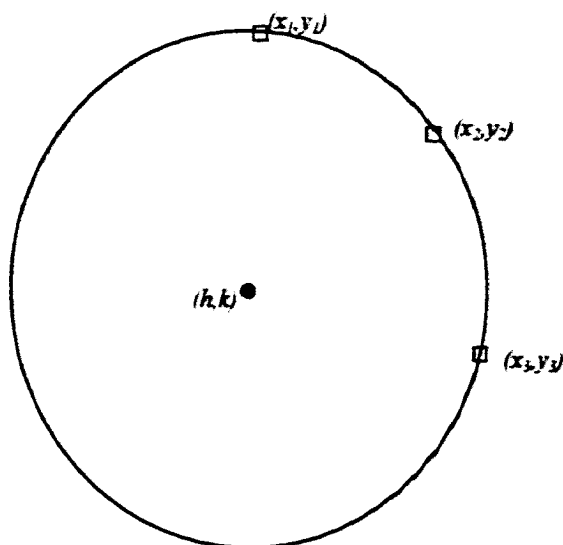
In all four cases the circle c computed by $r_{\text{circle}}(e^-, e, e^+)$ must intersect the three elements e^- , e and e^+ exactly once. Where an element is an edge there is an additional requirement. Since edges constrain the passing area only along the outer boundary, the circle may not include the line segment as in figure 4.16 where the passing area is tangent to edge q_2 . Therefore, the circle must either be tangent to the line segment or intersect the edge at one of its endpoints without including any portion of the edge. Finally, circle c must intersect elements e^- , e and e^+ in order counterclockwise, just as the arcs that bound the passing area perimeter do, as we shall see in figure 4.25. The subsequent subsections compute $r_{\text{circle}}(e^-, e, e^+)$ for each of the four cases in figure 4.24b.

Notational Comment: The points where c intersects e^- , e and e^+ are denoted by (x_1, y_1) , (x_2, y_2) and (x_3, y_3) , respectively. The center of c is at (h, k) and its radius is r .

The simultaneous equations presented were solved using the Mathematica software package (version 4.0). Therefore, the solutions are accompanied with a sprinkling of Mathematica notation. Specifically, each set of equations is solved, simplified and, when necessary, redundant solutions eliminated using the `Solve`, `FullSimplify` and `Union` commands, respectively. Solutions are presented as comma separated lists where the arrow symbol, “ $\rightarrow$ ”, is interpreted as “equal to”. Each solution is enclosed in curly braces “{ }”.) For example, to solve for the intersection point (x, y) of two lines, $y = m_1 x + b_1$ and $y = m_2 x + b_2$, `Solve` is called as follows: `Solve[{y == m1 x + b1 and y == m2 x + b2}, {x, y}]`.

To simplify the results and eliminate any possible redundant solutions FullSimplify and Union are called as follows: Union[FullSimplify [Solve[{y == m1 x + b1 and y == m2 x + b2},{x,y}}]]]

4.2.3.2.1 Case A: Three Vertices: (• • •)



A circle centered at (h, k) with radius r intersecting a point (x, y) must satisfy the equation $\frac{(x-h)^2}{r^2} + \frac{(y-k)^2}{r^2} = 1$. Solving for the circle center (h, k) and radius, r , that intersect three points (x_1, y_1) , (x_2, y_2) and (x_3, y_3) we have:

$$\text{FullSimplify}\left[\text{Solve}\left[\left\{\frac{(x_1-h)^2}{r^2} + \frac{(y_1-k)^2}{r^2} = 1, \frac{(x_2-h)^2}{r^2} + \frac{(y_2-k)^2}{r^2} = 1, \frac{(x_3-h)^2}{r^2} + \frac{(y_3-k)^2}{r^2} = 1\right\}, \{h, k, r\}\right]\right]$$

$$\left\{\left\{r \rightarrow -\frac{1}{2} \sqrt{\frac{((x_1-x_2)^2 + (y_1-y_2)^2)((x_1-x_3)^2 + (y_1-y_3)^2)((x_2-x_3)^2 + (y_2-y_3)^2)}{(x_3(-y_1+y_2) + x_2(y_1-y_3) + x_1(-y_2+y_3))^2}},\right.\right.$$

$$h \rightarrow \frac{x_3^2(y_1-y_2) + (x_1^2 + (y_1-y_2)(y_1-y_3))(y_2-y_3) + x_2^2(-y_1+y_3)}{2(x_3(y_1-y_2) + x_1(y_2-y_3) + x_2(-y_1+y_3))},$$

$$k \rightarrow \frac{-x_2^2 x_3 + x_1^2(-x_2+x_3) + x_3(y_1^2-y_2^2) + x_1(x_2^2-x_3^2+y_2^2-y_3^2) + x_2(x_3^2-y_1^2+y_3^2)}{2(x_3(y_1-y_2) + x_1(y_2-y_3) + x_2(-y_1+y_3))}\left.\right\},$$

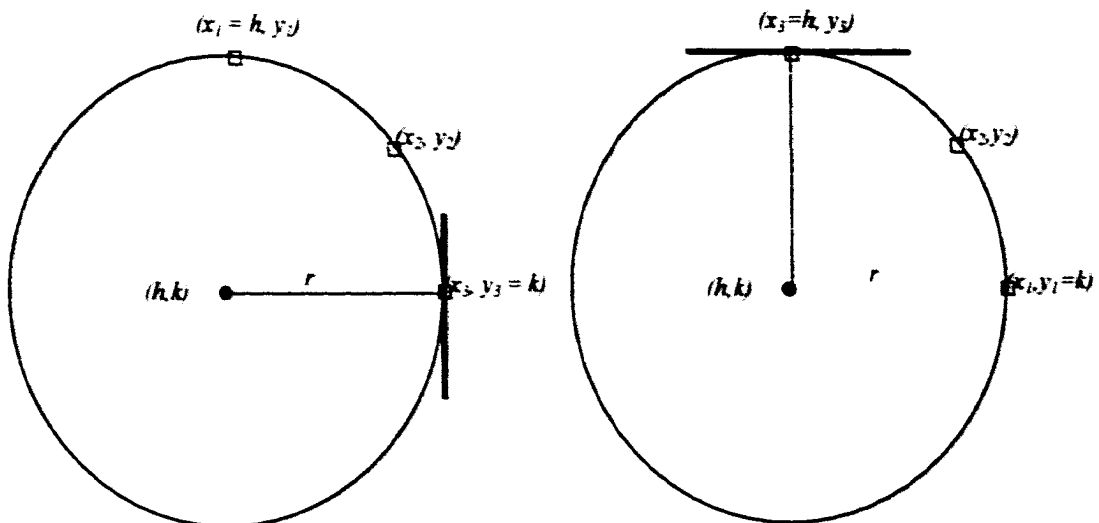
$$\left\{r \rightarrow \frac{1}{2} \sqrt{\frac{((x_1-x_2)^2 + (y_1-y_2)^2)((x_1-x_3)^2 + (y_1-y_3)^2)((x_2-x_3)^2 + (y_2-y_3)^2)}{(x_3(-y_1+y_2) + x_2(y_1-y_3) + x_1(-y_2+y_3))^2}},\right.$$

$$h \rightarrow \frac{x_3^2(y_1-y_2) + (x_1^2 + (y_1-y_2)(y_1-y_3))(y_2-y_3) + x_2^2(-y_1+y_3)}{2(x_3(y_1-y_2) + x_1(y_2-y_3) + x_2(-y_1+y_3))},$$

$$k \rightarrow \frac{-x_2^2 x_3 + x_1^2(-x_2+x_3) + x_3(y_1^2-y_2^2) + x_1(x_2^2-x_3^2+y_2^2-y_3^2) + x_2(x_3^2-y_1^2+y_3^2)}{2(x_3(y_1-y_2) + x_1(y_2-y_3) + x_2(-y_1+y_3))}\left.\right\}$$

The only positive value of r is in the second solution and thus the second solution is valid.

4.2.3.2.2 Case B: Two Vertices and One Edge: (• • —)



The circle c must intersect all three elements in exactly one point and not include any portion of the edge. The equations presented below solve for the tangent point

(x_3, y_3) along the linear extension of the edge. The edge may either be vertical (case 1) or horizontal (case 2). When the edge is vertical, y_3 is known ($= k$) and we solve for x_3 . Similarly, when the edge is horizontal, x_3 is fixed ($= h$) and we solve for y_3 .

Since the circle is tangent to the edge at (x_3, y_3) when the edge is vertical (horizontal) the distance from the tangent point to the center (h, k) is r along the horizontal (vertical) axis. Thus, with a vertical edge $(h + r - x_3)(h - r - x_3) = 0$ and for a horizontal edge, $(k + r - y_3)(k - r - y_3) = 0$. We solve for (h, k) , r and (x_3, y_3) .

The solutions for the simultaneous equations for both cases are expressed below. Four solutions exist, of which only two have a positive radius. Although two solutions with positive r exist, as in figure 4.25, only one circle exists that intersects the three elements, e^- , e and e^+ in order counterclockwise, as indicated by the doubled circle.

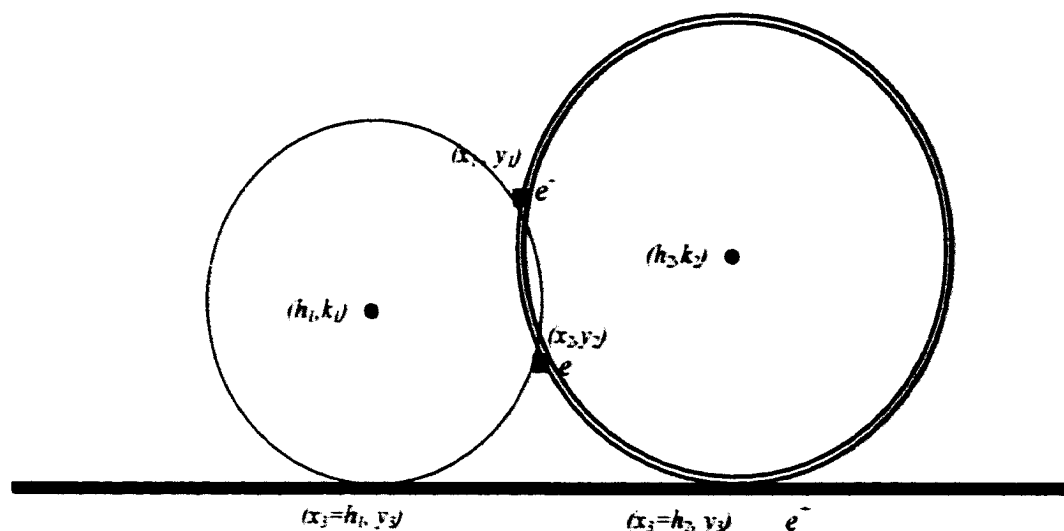


Figure 4.25. Two circles exist that intersect two points and are tangent to a line. Only one will intersect e^- , e and e^+ in order counter clockwise, as shown with the double lined circle.

Case 1: Vertical Edge

Union[

FullSimplify[

$$\text{Solve}\left[\left\{\frac{(x_1 - h)^2}{r^2} + \frac{(y_1 - k)^2}{r^2} = 1, \frac{(x_2 - h)^2}{r^2} + \frac{(y_2 - k)^2}{r^2} = 1, \frac{(x_3 - h)^2}{r^2} + \frac{(y_3 - k)^2}{r^2} = 1, \right.\right. \\ \left.\left.(h + r - x_3) + (h - r - x_3) = 0\right\}, \{h, k, r, y_3\}\right]\right]$$

$$\left\{\left\{y_3 \rightarrow \frac{-x_2 y_1 - \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + x_3 (y_1 - y_2) + x_1 y_2}{x_1 - x_2}, \right.\right. \\ z \rightarrow \frac{(x_1 + x_2 - 2 x_3) ((x_1 - x_2)^2 + (y_1 - y_2)^2) - 2 \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} (y_1 - y_2)}{2 (x_1 - x_2)^2}, \\ k \rightarrow \frac{-x_2 y_1 - \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - x_3 (y_1 - y_2) - x_1 y_2}{x_1 - x_2}, \\ h \rightarrow \frac{1}{2 (x_1 - x_2)^2} \left\{x_1^3 - x_1^2 x_2 + x_2^3 - x_2 (y_1 - y_2)^2 - x_1 (x_2 - y_1 - y_2) (x_2 - y_1 + y_2) - \right. \\ \left. 2 (y_1 - y_2) \left[-x_3 y_1 + \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + x_3 y_2\right]\right\}, \\ \left\{y_3 \rightarrow \frac{-x_2 y_1 - \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + x_3 (y_1 - y_2) + x_1 y_2}{x_1 - x_2}, \right. \\ z \rightarrow \frac{(x_1 + x_2 - 2 x_3) ((x_1 - x_2)^2 + (y_1 - y_2)^2) + 2 \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} (y_1 - y_2)}{2 (x_1 - x_2)^2}, \\ k \rightarrow \frac{-x_2 y_1 - \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - x_3 (y_1 - y_2) - x_1 y_2}{x_1 - x_2}, \\ h \rightarrow \frac{1}{2 (x_1 - x_2)^2} \left\{x_1^3 - x_1^2 x_2 + x_2^3 - x_2 (y_1 - y_2)^2 - x_1 (x_2 - y_1 - y_2) (x_2 - y_1 + y_2) - \right. \\ \left. 2 (y_1 - y_2) \left[-x_3 y_1 + \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + x_3 y_2\right]\right\}, \\ \left\{y_3 \rightarrow \frac{-x_2 y_1 + \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + x_3 (y_1 - y_2) + x_1 y_2}{x_1 - x_2}, \right. \\ z \rightarrow \frac{(x_1 + x_2 - 2 x_3) ((x_1 - x_2)^2 + (y_1 - y_2)^2) - 2 \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} (y_1 - y_2)}{2 (x_1 - x_2)^2}, \\ k \rightarrow \frac{-x_2 y_1 + \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - x_3 (y_1 - y_2) - x_1 y_2}{x_1 - x_2}, \\ h \rightarrow \frac{1}{2 (x_1 - x_2)^2} \left\{x_1^3 - x_1^2 x_2 + x_2^3 - x_2 (y_1 - y_2)^2 - x_1 (x_2 - y_1 - y_2) (x_2 - y_1 + y_2) - \right. \\ \left. 2 (y_1 - y_2) \left[x_3 y_1 - \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - x_3 y_2\right]\right\}, \\ \left\{y_3 \rightarrow \frac{-x_2 y_1 + \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + x_3 (y_1 - y_2) + x_1 y_2}{x_1 - x_2}, \right. \\ z \rightarrow \frac{-(x_1 + x_2 - 2 x_3) ((x_1 - x_2)^2 + (y_1 - y_2)^2) + 2 \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} (y_1 - y_2)}{2 (x_1 - x_2)^2}, \\ k \rightarrow \frac{-x_2 y_1 + \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + x_3 (y_1 - y_2) - x_1 y_2}{x_1 - x_2}, \\ h \rightarrow \frac{1}{2 (x_1 - x_2)^2} \left\{x_1^3 - x_1^2 x_2 + x_2^3 - x_2 (y_1 - y_2)^2 - x_1 (x_2 - y_1 - y_2) (x_2 - y_1 + y_2) - \right. \\ \left. 2 (y_1 - y_2) \left[x_3 y_1 - \sqrt{(x_1 - x_3)(x_2 - x_3)} \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} - x_3 y_2\right]\right\}\right\}$$

Case 2 : Horizontal Edge

Union[

FullSimplify[

$$\text{Solve}\left[\left\{\frac{(x1-h)^2}{r^2} + \frac{(y1-k)^2}{r^2} = 1, \frac{(x2-h)^2}{r^2} + \frac{(y2-k)^2}{r^2} = 1, \frac{(x3-h)^2}{r^2} + \frac{(y3-k)^2}{r^2} = 1,\right.\right.$$

$$\left.\left.(k+r-y3) + (k-r+y3) = 0\right\}, \{h, k, r, x3\}\right]$$

$$\begin{aligned} (x3) &= \frac{x2(y1-y3) - \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - x1(-y2-y3)}{y1-y2} \\ c &= \frac{((x1-x2)^2 - (y1-y2)^2)(y1-y2-2y3) + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}}{2(y1-y2)^2} \\ n &= \frac{x2(y1-y3) - \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - x1(-y2-y3)}{y1-y2} \\ k &= \frac{1}{2(y1-y2)^2} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - \\ &\quad y2^3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3) \\ (x3) &= \frac{x2(y1-y3) - \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - x1(-y2-y3)}{y1-y2} \\ c &= \frac{((x1-x2)^2 - (y1-y2)^2)(y1-y2-2y3) + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}}{2(y1-y2)^2} \\ n &= \frac{x2(y1-y3) - \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - x1(-y2-y3)}{y1-y2} \\ k &= \frac{1}{2(y1-y2)^2} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - \\ &\quad y2^3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3) \\ (x3) &= \frac{1}{(y1-y2)^2} ((y1-y2)(x2(y1-y3) - x1(y2-y3) + \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}) - \\ &\quad \sqrt{2} \sqrt{(y1-y2)^2 y3} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - y2^3 - \\ &\quad 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)})) \\ c &= \frac{-2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - ((x1-x2)^2 - (y1-y2)^2)(y1-y2-2y3)}{2(y1-y2)^2} \\ n &= \frac{x2(y1-y3) - x1(y2-y3) - \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}}{y1-y2} \\ k &= \frac{1}{2(y1-y2)^2} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - \\ &\quad y2^3 + 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}) \\ (x3) &= \frac{1}{(y1-y2)^2} ((y1-y2)(x2(y1-y3) - x1(y2-y3) + \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}) - \\ &\quad \sqrt{2} \sqrt{(y1-y2)^2 y3} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - y2^3 - \\ &\quad 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)})) \\ c &= \frac{-2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - ((x1-x2)^2 - (y1-y2)^2)(y1-y2-2y3)}{2(y1-y2)^2} \\ n &= \frac{x2(y1-y3) - x1(y2-y3) - \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}}{y1-y2} \\ k &= \frac{1}{2(y1-y2)^2} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - \\ &\quad y2^3 + 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}) \\ (x3) &= \frac{1}{(y1-y2)^2} ((y1-y2)(x2(y1-y3) - x1(y2-y3) + \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}) - \\ &\quad \sqrt{2} \sqrt{(y1-y2)^2 y3} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - y2^3 - \\ &\quad 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)})) \\ c &= \frac{2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - ((x1-x2)^2 - (y1-y2)^2)(y1-y2-2y3)}{2(y1-y2)^2} \\ n &= \frac{x2(y1-y3) - x1(y2-y3) - \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}}{y1-y2} \\ k &= \frac{1}{2(y1-y2)^2} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - \\ &\quad y2^3 + 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}) \\ (x3) &= \frac{1}{(y1-y2)^2} ((y1-y2)(x2(y1-y3) - x1(y2-y3) + \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}) - \\ &\quad \sqrt{2} \sqrt{(y1-y2)^2 y3} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - y2^3 - \\ &\quad 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)})) \\ c &= \frac{2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} - ((x1-x2)^2 - (y1-y2)^2)(y1-y2-2y3)}{2(y1-y2)^2} \\ n &= \frac{x2(y1-y3) - x1(y2-y3) - \sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}}{y1-y2} \\ k &= \frac{1}{2(y1-y2)^2} (x1^2 y1 - 2 x1 x2 y1 - x2^2 y1 + y1^3 - x1^2 y2 - 2 x1 x2 y2 - x2^2 y2 - y1^2 y2 - y1 y2^2 - \\ &\quad y2^3 + 2 x1^2 y3 - 4 x1 x2 y3 - 2 x2^2 y3 + 2(x1-x2)\sqrt{(x1-x2)^2 - (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)}) \end{aligned}$$

If a circle exists that is tangent to the edge, its radius is the value of $r_{\text{circle}}(e^-, e, e^+)$. Otherwise, the circles, c_1 and c_2 , defined by the two vertices and each edge endpoint, are constructed as in case A. The circle that does not contain any internal portion of the edge is selected. See figure 4.26 where the two circles intersecting e^-, e and either endpoint of e^+ is shown. Only the circle with the double line does not include the edge and is therefore selected.

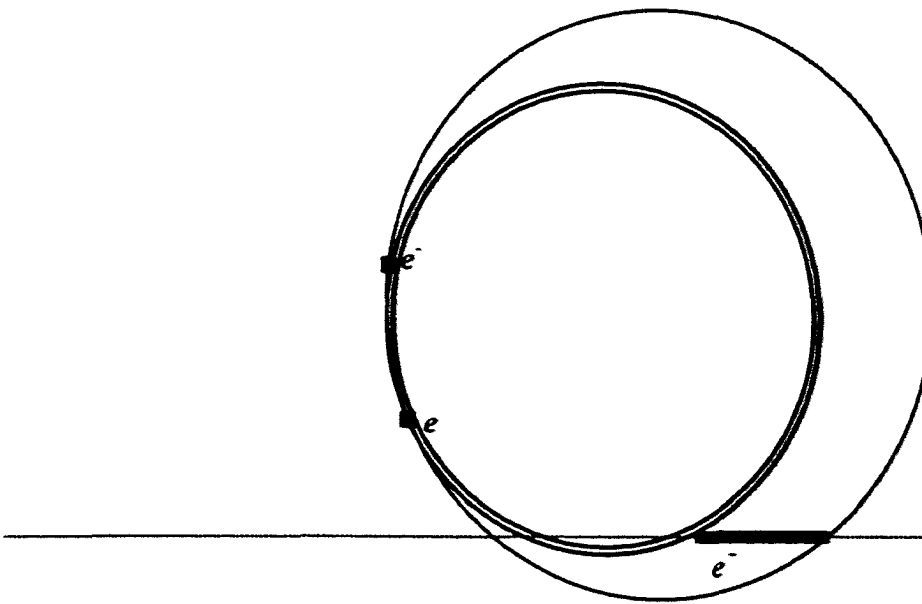


Figure 4.26. When the circle tangent to the edge does not exist, the circle intersecting an endpoint of the edge without including any portion of the edge is selected. This circle is shown as a doubled circle.

4.2.3.2.3 Case C: One Vertex and Two Edges: ($\bullet$ — —)

We refer to the vertex as v and the edges as e_2 and e_3 . The following three orientation of the edges are considered:

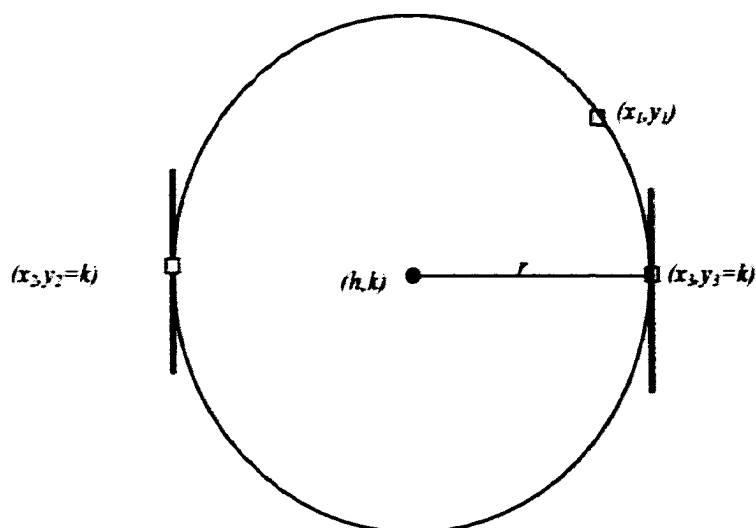
- 1) e_2 and e_3 are both vertical
- 2) e_2 and e_3 are both horizontal
- 3) e_2 is vertical and e_3 is horizontal.

As with case B, we solve for the circle that is tangent to the edges' linear extensions. If that circle is tangent to both edges, $r_{\text{circle}}(e^-, e, e^+)$ is the radius of that circle. If that circle is not tangent to one of the edges, one of the edge's endpoint is selected in lieu of the edge, as with case B. The details are discussed below.

Since the center of a circle that intersects a point and is tangent to two lines is equidistant to the point and lines, the circle center must lie on the medial curve of the two lines, the bisector of the lines and the medial curve of v and edge e_2 , a parabola with focus v and directrix e_2 . A line and parabola intersect in at most two points. There is, thus, a maximum of two solutions. However, only one solution will exist that will intersect e^-, e and e^+ in order counterclockwise.

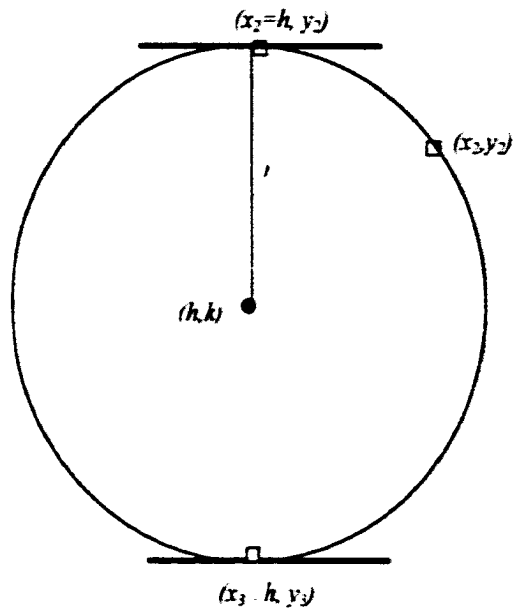
Solving for the tangent points along e_2 and e_3 and the radius, r , we solve the following set of simultaneous equations:

Case 1: e_2 and e_3 are both vertical.



$$\begin{aligned}
 &\text{Union[} \\
 &\quad \text{FullSimplify[} \\
 &\quad \quad \text{Solve[} \left\{ \frac{(x_1 - h)^2}{r^2} + \frac{(y_1 - k)^2}{r^2} = 1, \frac{(x_2 - h)^2}{r^2} + \frac{(y_2 - k)^2}{r^2} = 1, \frac{(x_3 - h)^2}{r^2} + \frac{(y_3 - k)^2}{r^2} = 1, \right. \\
 &\quad \quad \left. (h + r - x_2) + (h - r - x_2) = 0, (h + r - x_3) + (h - r - x_3) = 0 \right\}, \{h, k, r, y_2, y_3\} \}]] \\
 &\quad \left\{ \left\{ y_2 \rightarrow -\sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, y_3 \rightarrow -\sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, \right. \right. \\
 &\quad \quad \left. \tau \rightarrow \frac{x_2 - x_3}{2}, k \rightarrow -\sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, h \rightarrow \frac{x_2 + x_3}{2} \right\}, \left\{ y_2 \rightarrow -\sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, \right. \\
 &\quad \quad \left. y_3 \rightarrow -\sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, \tau \rightarrow \frac{1}{2}(-x_2 - x_3), k \rightarrow -\sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, h \rightarrow \frac{x_2 + x_3}{2} \right\}, \\
 &\quad \left\{ y_2 \rightarrow \sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, y_3 \rightarrow \sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, \tau \rightarrow \frac{x_2 - x_3}{2}, \right. \\
 &\quad \quad \left. k \rightarrow \sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, h \rightarrow \frac{x_2 + x_3}{2} \right\}, \left\{ y_2 \rightarrow \sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, \right. \\
 &\quad \quad \left. y_3 \rightarrow \sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, \tau \rightarrow \frac{1}{2}(-x_2 + x_3), k \rightarrow \sqrt{-(x_1 - x_2)(x_1 - x_3)} + y_1, h \rightarrow \frac{x_2 + x_3}{2} \right\} \}
 \end{aligned}$$

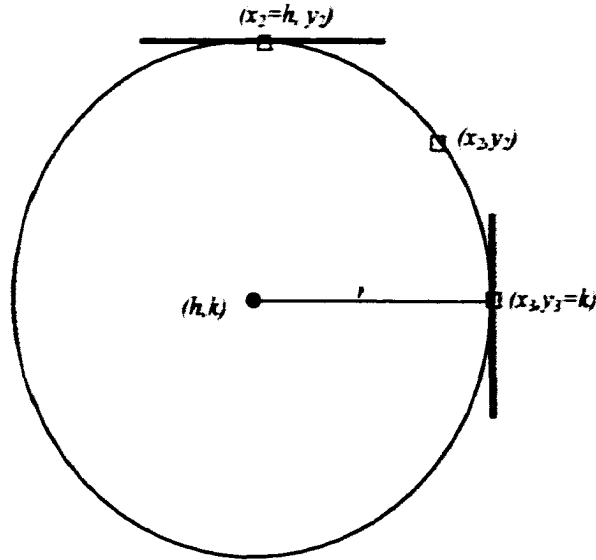
Case 2: e_2 and e_3 are both horizontal.



```

Union[
  FullSimplify[
    Solve[ {  $\frac{(x1-h)^2}{r^2} + \frac{(y1-k)^2}{r^2} = 1$ ,  $\frac{(x2-h)^2}{r^2} + \frac{(y2-k)^2}{r^2} = 1$ ,  $\frac{(x3-h)^2}{r^2} + \frac{(y3-k)^2}{r^2} = 1$ ,
       $(k+r-y2) + (k-r-y2) = 0$ ,  $(k+r-y3) + (k-r-y3) = 0$  }, {h, k, r, x2, x3} ] ] ]
  { {  $x2 \rightarrow x1 - \sqrt{-(y1-y2)(y1-y3)}$ ,  $x3 \rightarrow x1 - \sqrt{-(y1-y2)(y1-y3)}$ ,
     $r \rightarrow \frac{y2-y3}{2}$ ,  $h \rightarrow x1 - \sqrt{-(y1-y2)(y1-y3)}$ ,  $k \rightarrow \frac{y2+y3}{2}$  }, {  $x2 \rightarrow x1 - \sqrt{-(y1-y2)(y1-y3)}$ ,
     $x3 \rightarrow x1 - \sqrt{-(y1-y2)(y1-y3)}$ ,  $r \rightarrow \frac{1}{2}(-y2+y3)$ ,  $h \rightarrow x1 - \sqrt{-(y1-y2)(y1-y3)}$ ,  $k \rightarrow \frac{y2+y3}{2}$  },
    {  $x2 \rightarrow x1 + \sqrt{-(y1-y2)(y1-y3)}$ ,  $x3 \rightarrow x1 + \sqrt{-(y1-y2)(y1-y3)}$ ,  $r \rightarrow \frac{y2-y3}{2}$ ,
     $h \rightarrow x1 + \sqrt{-(y1-y2)(y1-y3)}$ ,  $k \rightarrow \frac{y2+y3}{2}$  }, {  $x2 \rightarrow x1 + \sqrt{-(y1-y2)(y1-y3)}$ ,
     $x3 \rightarrow x1 + \sqrt{-(y1-y2)(y1-y3)}$ ,  $r \rightarrow \frac{1}{2}(-y2+y3)$ ,  $h \rightarrow x1 + \sqrt{-(y1-y2)(y1-y3)}$ ,  $k \rightarrow \frac{y2+y3}{2}$  } } ] ]
```

Case 3: e_2 is vertical and e_3 is horizontal.



As mentioned earlier, only two real solutions exist. Of these two only one will intersect the three elements in order counterclockwise. If (x_2, y_2) and (x_3, y_3) are on edges e_2 and e_3 , respectively, $r_{\text{circle}}(e^-, e, e^+) = r$. If either tangent point is not on its respective edge we may then solve for r_{circle} by finding the circles c_1 and c_2 which are the circles defined by v , e_2 and an endpoint of e_3 , respectively, and circles c_3 and c_4 which are defined by v , e_3 and an endpoint of e_2 , respectively. These circles are computed as set forth in case B. Of the four circles' radii the radius of the circle that intersects e_2 and e_3 exactly once and does not include any part of either e_2 or e_3 , is selected.

Union[

FullSimplify[

$$\text{Solve}\left[\left\{\frac{(x1-h)^2}{r^2} + \frac{(y1-k)^2}{r^2} = 1, \frac{(x2-h)^2}{r^2} + \frac{(y2-k)^2}{r^2} = 1, \frac{(x3-h)^2}{r^2} + \frac{(y3-k)^2}{r^2} = 1,\right.\right.$$

$$\left.\left.(h+r-x3) + (h-r-x3) = 0, (k+r-y2) + (k-r-y2) = 0\right\}, \{h, k, r, y3, x2\}\right]$$

$$\begin{aligned} & \left\{ \left\{ y3 \rightarrow x1 - x3 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2}, x2 \rightarrow x1 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, \right. \right. \\ & r \rightarrow x1 - x3 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, h \rightarrow x1 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, \\ & k \rightarrow x1 - x3 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} \}, \{ y3 \rightarrow x1 - x3 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2}, \\ & x2 \rightarrow x1 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, r \rightarrow -x1 + x3 - y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} + y2, \\ & h \rightarrow x1 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, k \rightarrow x1 - x3 + y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} \}, \\ & \{ y3 \rightarrow x1 - x3 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2}, x2 \rightarrow x1 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, \\ & r \rightarrow x1 - x3 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, h \rightarrow x1 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, \\ & k \rightarrow x1 - x3 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} \}, \{ y3 \rightarrow x1 - x3 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2}, \\ & x2 \rightarrow x1 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, r \rightarrow -x1 + x3 - y1 - \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} + y2, \\ & h \rightarrow x1 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} - y2, k \rightarrow x1 - x3 + y1 + \sqrt{2} \sqrt{x1 - x3} \sqrt{y1 - y2} \}, \\ & \{ y3 \rightarrow -x1 + x3 + y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)}, x2 \rightarrow x1 - y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, \\ & r \rightarrow -x1 + x3 + y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} - y2, h \rightarrow x1 - y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, \\ & k \rightarrow -x1 + x3 + y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} \}, \{ y3 \rightarrow -x1 + x3 + y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)}, \\ & x2 \rightarrow x1 - y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, r \rightarrow x1 - x3 - y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, \\ & h \rightarrow x1 - y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, k \rightarrow -x1 + x3 + y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} \}, \\ & \{ y3 \rightarrow -x1 + x3 + y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)}, x2 \rightarrow x1 - y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, \\ & r \rightarrow -x1 + x3 + y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} - y2, h \rightarrow x1 - y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, \\ & k \rightarrow -x1 + x3 + y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} \}, \{ y3 \rightarrow -x1 + x3 + y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)}, \\ & x2 \rightarrow x1 - y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, r \rightarrow x1 - x3 - y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, \\ & h \rightarrow x1 - y1 - \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} + y2, k \rightarrow -x1 + x3 + y1 + \sqrt{2} \sqrt{-(x1 - x3) (y1 - y2)} \} \} \end{aligned}$$

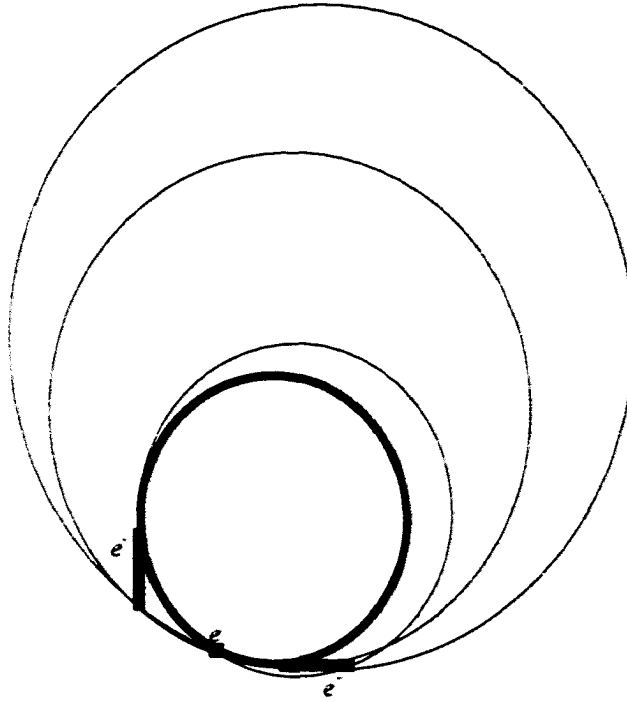


Figure 4.27. When no circle exists that is tangent to both edges, four circles are determined that intersect the vertex and an endpoint of the two edges. The circle that does not include any portion of either edge (darkened) is selected.

4.2.3.2.4 Case D: Three Edges: (— — —)

The three edges, e_1 , e_2 and e_3 may have one of two combinations of their orientations: either two edges are vertical and one is horizontal or two edge's are horizontal and one is vertical. With no loss in generality these cases may be expressed as:

- 1) e_1 and e_2 are vertical and e_3 is horizontal
- 2) e_1 and e_2 are horizontal and e_3 is vertical.

A single solution exists since the circle center must lie on the intersection of the edge bisectors, which are lines, as in figure 4.28. Two nonparallel lines intersect at exactly one point. The simultaneous equations and solutions are as follows.

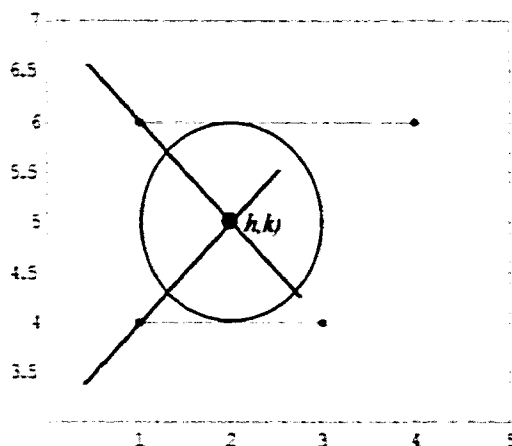
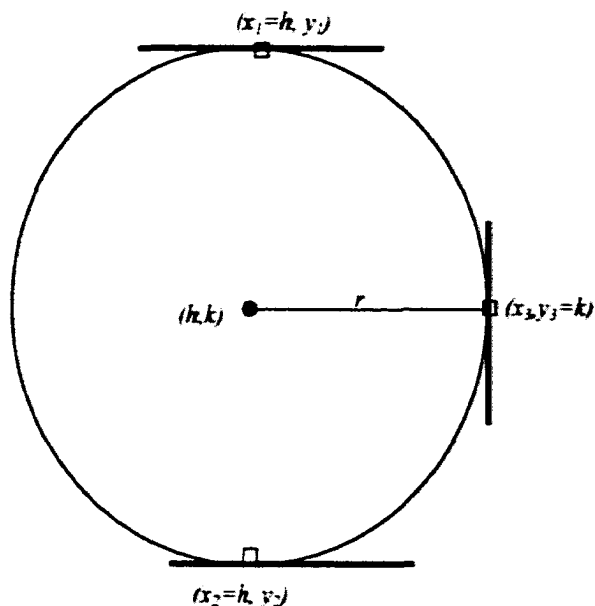


Figure 4.28a. The center of the circle that is tangent to the edges must lie on the intersection of the angle bisectors of the edge extensions.

Case 1: e_2 and e_3 are both horizontal.



```

Union[
  FullSimplify[
    Solve[ {  $\frac{(x_1 - h)^2}{r^2} + \frac{(y_1 - k)^2}{r^2} = 1$ ,  $\frac{(x_2 - h)^2}{r^2} + \frac{(y_2 - k)^2}{r^2} = 1$ ,  $\frac{(x_3 - h)^2}{r^2} + \frac{(y_3 - k)^2}{r^2} = 1$ ,
       $(k + r - y_1) + (k - r - y_1) = 0$ ,  $(k + r - y_2) + (k - r - y_2) = 0$ ,  $(h + r - x_3) + (h - r - x_3) = 0$  },
      {h, k, r, x1, x2, y3} ] ] ]

```

$$\left\{ \left\{ x_1 \rightarrow \frac{1}{2} (6 - y_1 - y_2), x_2 \rightarrow \frac{1}{2} (6 - y_1 - y_2), y_3 \rightarrow \frac{1}{2} (y_1 - 2\sqrt{-3 - x_3}\sqrt{3 - x_3 - y_1 - y_2} + y_2), \right. \right.$$

$$r \rightarrow \frac{y_1 - y_2}{2}, h \rightarrow \frac{1}{2} (6 - y_1 - y_2), k \rightarrow \frac{y_1 + y_2}{2} \Big\}, \left\{ x_1 \rightarrow \frac{1}{2} (6 - y_1 - y_2), x_2 \rightarrow \frac{1}{2} (6 - y_1 - y_2), \right.$$

$$y_3 \rightarrow \frac{1}{2} (y_1 + 2\sqrt{-3 - x_3}\sqrt{3 - x_3 - y_1 - y_2} + y_2), r \rightarrow \frac{y_1 - y_2}{2}, h \rightarrow \frac{1}{2} (6 - y_1 - y_2), k \rightarrow \frac{y_1 + y_2}{2} \Big\},$$

$$\left\{ x_1 \rightarrow \frac{1}{2} (2x_3 - y_1 - y_2), x_2 \rightarrow \frac{1}{2} (2x_3 - y_1 - y_2), y_3 \rightarrow \frac{y_1 - y_2}{2}, r \rightarrow \frac{1}{2} (-y_1 - y_2), h \rightarrow \frac{1}{2} (2x_3 - y_1 - y_2), \right.$$

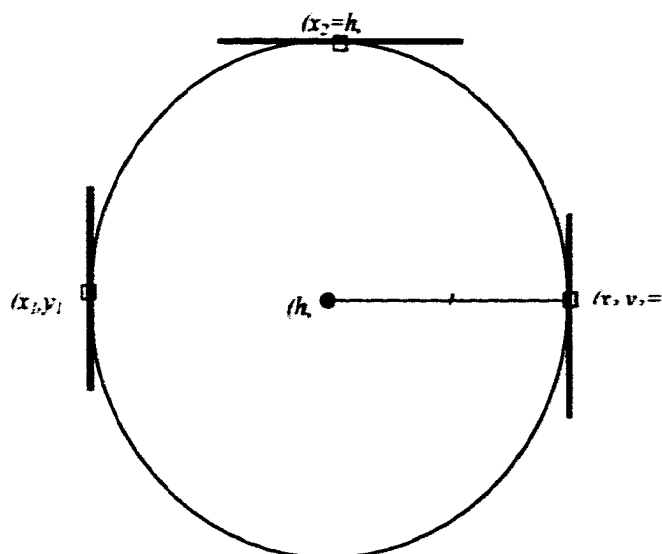
$$k \rightarrow \frac{y_1 + y_2}{2} \Big\}, \left\{ x_1 \rightarrow \frac{1}{2} (6 - y_1 - y_2), x_2 \rightarrow \frac{1}{2} (6 - y_1 - y_2), y_3 \rightarrow \frac{1}{2} (y_1 - 2\sqrt{-(-3 - x_3)}\sqrt{-3 - x_3 - y_1 - y_2} + y_2), \right.$$

$$r \rightarrow \frac{1}{2} (-y_1 - y_2), h \rightarrow \frac{1}{2} (6 - y_1 - y_2), k \rightarrow \frac{y_1 + y_2}{2} \Big\}, \left\{ x_1 \rightarrow \frac{1}{2} (6 - y_1 - y_2), x_2 \rightarrow \frac{1}{2} (6 - y_1 - y_2), \right.$$

$$y_3 \rightarrow \sqrt{-(-3 - x_3)}\sqrt{-3 - x_3 - y_1 - y_2} + \frac{y_1 + y_2}{2}, r \rightarrow \frac{1}{2} (-y_1 - y_2), h \rightarrow \frac{1}{2} (6 - y_1 - y_2), k \rightarrow \frac{y_1 + y_2}{2} \Big\},$$

$$\left\{ x_1 \rightarrow \frac{1}{2} (2x_3 - y_1 - y_2), x_2 \rightarrow \frac{1}{2} (2x_3 - y_1 - y_2), y_3 \rightarrow \frac{y_1 - y_2}{2}, r \rightarrow \frac{y_1 - y_2}{2}, h \rightarrow \frac{1}{2} (2x_3 - y_1 - y_2), k \rightarrow \frac{y_1 + y_2}{2} \right\}$$

Case 2: e_2 and e_3 are both vertical.



```

Union[
  FullSimplify[
    Solve[ {  $\frac{(x_1 - h)^2}{r^2} + \frac{(y_1 - k)^2}{r^2} = 1$ ,  $\frac{(x_2 - h)^2}{r^2} + \frac{(y_2 - k)^2}{r^2} = 1$ ,  $\frac{(x_3 - h)^2}{r^2} + \frac{(y_3 - k)^2}{r^2} = 1$ ,
       $(h + r - x_1) + (h - r - 1) = 0$ ,  $(h + r - x_2) + (h - r - x_2) = 0$ ,  $(k + r - y_3) + (k - r - y_3) = 0$  },
      {h, k, r, y_1, y_2, x_3} ] ] ]

```

$$\left\{ \left\{ y_1 \rightarrow \frac{x_1 - x_2}{2} - y_3, y_2 \rightarrow \frac{x_1 - x_2}{2} - y_3, x_3 \rightarrow \frac{x_1 - x_2}{2}, r \rightarrow \frac{x_1 - x_2}{2}, k \rightarrow \frac{x_1 - x_2}{2} - y_3, h \rightarrow \frac{x_1 - x_2}{2} \right\}, \right.$$

$$\left\{ y_1 \rightarrow \frac{1}{2} - \frac{x_2}{2} - \sqrt{-1 + x_1} \sqrt{-x_1 - x_2} - y_3, y_2 \rightarrow \frac{1}{2} - \frac{x_2}{2} - y_3, x_3 \rightarrow \frac{1 + x_2}{2}, r \rightarrow \frac{1}{2} (-1 - x_2), \right.$$

$$\left. k \rightarrow \frac{1}{2} - \frac{x_2}{2} - y_3, h \rightarrow \frac{1 + x_2}{2} \right\}, \left\{ y_1 \rightarrow \frac{1}{2} - \frac{x_2}{2} + \sqrt{-1 + x_1} \sqrt{-x_1 - x_2} - y_3, y_2 \rightarrow \frac{1}{2} - \frac{x_2}{2} - y_3, \right.$$

$$\left. x_3 \rightarrow \frac{1 + x_2}{2}, r \rightarrow \frac{1}{2} (-1 + x_2), k \rightarrow \frac{1}{2} - \frac{x_2}{2} - y_3, h \rightarrow \frac{1 + x_2}{2} \right\}, \left\{ y_1 \rightarrow -\frac{1}{2} - \frac{x_2}{2} + \sqrt{-1 + x_1} \sqrt{-x_1 - x_2} - y_3, \right.$$

$$\left. y_2 \rightarrow \frac{1}{2} (-1 - x_2) - y_3, x_3 \rightarrow \frac{1 + x_2}{2}, r \rightarrow \frac{1}{2} (-1 - x_2), k \rightarrow \frac{1}{2} (-1 - x_2) - y_3, h \rightarrow \frac{1 + x_2}{2} \right\},$$

$$\left\{ y_1 \rightarrow \frac{1}{2} (-1 - x_2) - y_3, y_2 \rightarrow \frac{1}{2} (-1 - x_2) - y_3, x_3 \rightarrow \frac{x_1 - x_2}{2}, r \rightarrow \frac{x_1 - x_2}{2}, k \rightarrow \frac{1}{2} (-1 - x_2) - y_3, h \rightarrow \frac{x_1 - x_2}{2} \right\},$$

$$\left\{ y_1 \rightarrow \frac{1}{2} (-1 - x_2 - 2\sqrt{-1 + x_1} \sqrt{-x_1 - x_2} + 2y_3), y_2 \rightarrow \frac{1}{2} (-1 - x_2) - y_3, \right.$$

$$\left. x_3 \rightarrow \frac{1 + x_2}{2}, r \rightarrow \frac{1}{2} (-1 - x_2), k \rightarrow \frac{1}{2} (-1 - x_2) - y_3, h \rightarrow \frac{1 + x_2}{2} \right\}$$

As with case C (section 4.2.3.2.3), the tangent points of the circle and edge's linear extension may not be located on the edge. In that case six circles are determined:

c_1 and c_2 , the circles defined by an endpoint of e_1 and edges e_2 and e_3 (as indicated in the figure).

c_3 and c_4 , the circles defined by an endpoint of e_2 and edges e_5 and e_6 , and c_1 and c_2 , defined by an endpoint of e_3 and edges e_1 and e_2 .

These circles are computed as set forth in case C. Of these six circles, the circle that intersects e_1 , e_2 and e_3 exactly once and does not include any point of edges e_1 , e_2 and e_3 is selected.

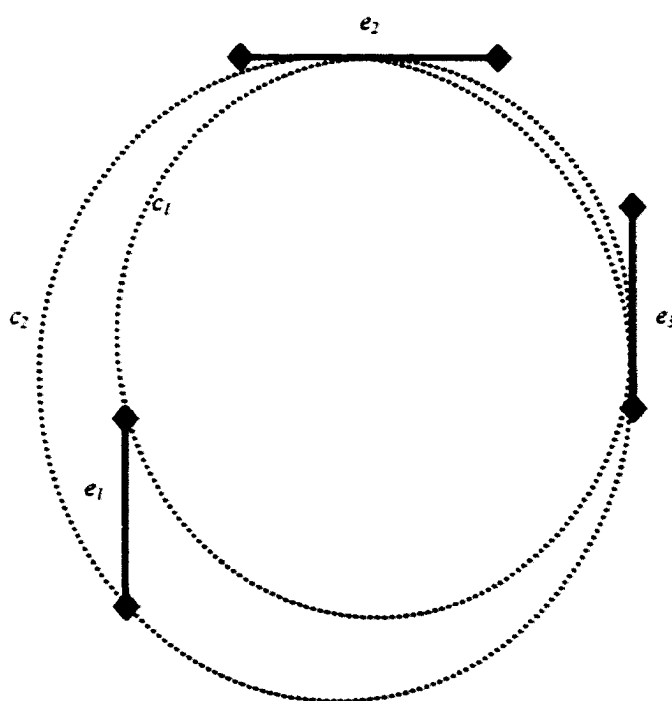


Figure 4.28b . In the event the circle as determined by the edge's linear extensions is not tangent to the all edges six circles are constructed. Each such circle is tangent to two edges and intersects the third edge's endpoints. Two such circles, c_1 and c_2 are shown here.

In this section the methodology for determining the constraints each critical element places on its neighbors was defined. Based on analytical geometry, four major cases and eight subcases were shown to determine the elements' critical ranges. These observations and calculation form the basis of the algorithm that is presented in the next section.

4.3 Algorithm for the Domain Computation of a Digital Circular Arc

In order to define the shape of an object in an image, boundary detection algorithms are employed to find the object's 4-connected digital boundary. That boundary can then be passed to the following algorithm (of this section) to determine the boundary's passing area. Given a 4-connected digital circular arc encoded as a Freeman chain (section 1.3) the algorithm presented in this section will compute the passing area of that digital arc considering all possible radii. If given an 8-connected digital curve the necessary adaptations to the α -infinite hull algorithm (discussed in section 3.6.2) would be implemented.

There are essentially three phases to this algorithm, presented in sub-sections within section 4.3.1. This is followed by a graphical demonstration of the algorithm step-by-step in section 4.3.2. The first phase (section 4.3.1.1) determines the digital curve's default (initial) passing area and initializes the data structure storing the passing area information. The second phase (section 4.3.1.2) computes the radius of the minimum circular arc that will digitize to the digital curve. The third and final phase (section 4.3.1.3) iteratively computes the critical sets for all valid radial ranges.

The convention that is used in this algorithm is that the pixels in the digital curve are integer values on the (x,y) grid. On the $\mathbb{R}^2$ (x,y) grid, these integer pixel values represent the square area bounded by (x,y) and $[x+1, y+1]$, called the pixel area. (Thus, any point (i, j) on the real grid maps to $(\lfloor i \rfloor, \lfloor j \rfloor)$ on the digital curve's integer grid.) Others sometimes refer to the integer pixels as being located on the real grid with a $\frac{1}{2}$ pixel phase shift in both the x and y direction from the integer values we assign them here. (See figure 4.29)

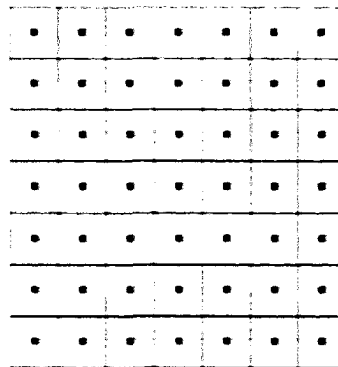


Figure 4.29. The pixels in $\mathbb{R}^2$, shown as dots, represent the pixel area bounded by a square.

For simplicity of description, the algorithm assumes that the concavity of the digital curve is known. In cases where the digital curve admits a linear interpretation, both boundaries may be interpreted as either concave or convex. In that case the algorithm is run twice to compute the passing area for both interpretations of which side of the digital curve boundary, the convexity lies. Where the concavity is unknown, the concavity is easily determined by finding the maximum linear segment, as was computed by Smeulders (1991) and then adding the pixel that causes the curve to become nonlinear.

The direction of that pixel relative to the rest of the digital curve will determine the concavity of the whole curve.

4.3.1 Algorithm (3 Phases)

As mentioned above, the algorithm is divided into three phases. In phase 1 (next section), the inner and outer boundaries of the digital curve are determined from the 4-connected point set, the possible critical elements are isolated (termed the “candidate list”) and are placed in a data structure. Then, in phase 2 (section 4.3.1.2), the minimum radius circular arc that digitizes to the digital curve is determined. Starting from this minimum radius, the algorithm iterates, in phase 3 (section 4.3.1.3), through each critical set determining its critical range. In the following subsections, the individual phases of the algorithm are followed by subsections that discuss implementation details.

4.3.1.1 Phase 1 – Creating the Candidate List

Note: Forward movement along the digital curve as well as the candidate list is counter clockwise.

1. Determine the truncated pixel area boundary of the digital curve. This is the pixel area boundary (i.e. the real area that the digital curve represents) of the digital curve where the boundaries of the first and last pixels of the digital curve are removed. The truncated pixel boundary is composed of two disjoint boundaries:

the inner and outer boundary, both of which are connected paths of line segments. (See figure 4.30) The inner boundary is closer to the circle center of the underlying circular arc than the outer boundary.

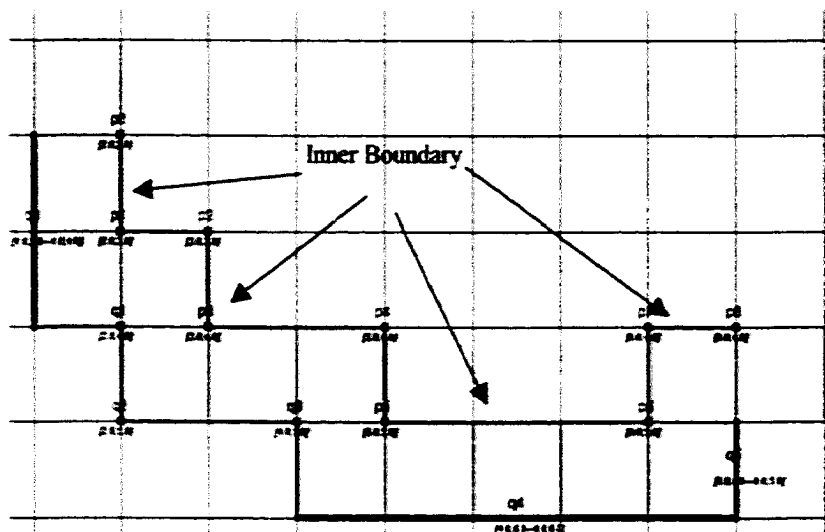


Figure 4.30. The preimage of the digital curve must pass from one end of the passing area to the other end without intersecting either inner or outer boundary. The outer boundary is the other side of each pixel than the inner boundary.

Any curve that digitizes to the digital curve must pass wholly between the inner and outer boundaries. The area between the inner and outer boundaries is the “initial passing area” and is the upper bound on the passing area of the underlying curve.

2. For the inner border find the corner points. For any three consecutive elements e_1 , e_2 and e_3 along the inner boundary, e_2 is a corner point if e_3 is to the left of $\overline{e_1e_2}$. The corner points are entered into the inner elements set. This set contains the constraints that the passing area's inner boundary places on the real curves that may digitize to the digital curve. Non-corner points do not constrain the passing area and are not included in the set of constraining points. For example, in figure 4.31 points p_2 , p_4 and p_7 are not corner points and thus do cannot constrain the passing area in any further manner.

3. For the outer border, find the corner points and enter them into the outer elements set. For any three consecutive elements e_1 , e_2 and e_3 along the outer boundary, e_2 is a corner point if e_3 is to the right of $\overline{e_1e_2}$. The corner points are entered into the outer elements set. In addition to points, edges can be considered outer border elements. Add edges (i.e. line segments) of the outer boundary to the outer elements set only if they are vertical or horizontal maxima. For example, in figure 4.31, edge q_6 may contain a horizontal stationary point and may constrain the passing area. Contained within these edges can be stationary points of the preimage curve which further constrain the passing area as discussed in section 3.4.

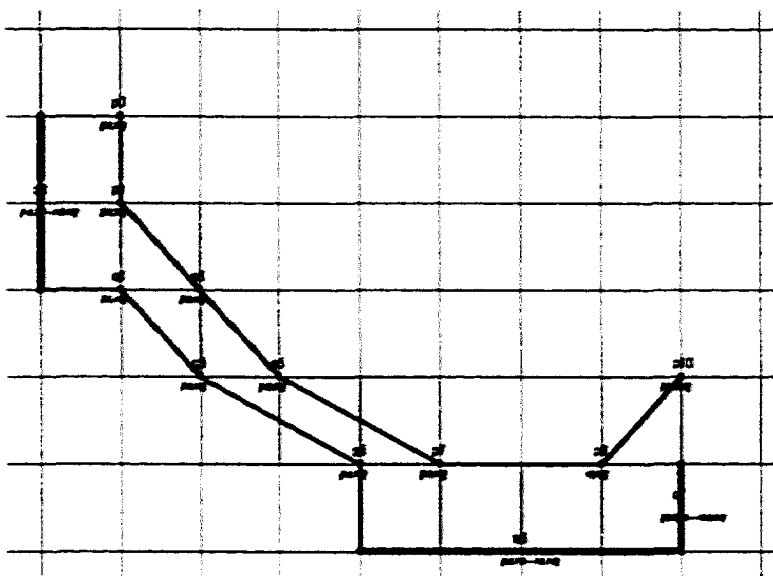


Figure 4.31. The elements that may not constrain the passing area have been removed (steps 1-3). The remaining elements are labeled and are then connected showing the connectivity of all elements in the joined boundary as described in steps 4 and 5.

Only the elements in the inner and outer elements sets as shown in figure 4.31 are included in the candidate list. The vertices and elements not included in the candidate list will not constrain the passing area in a further manner.

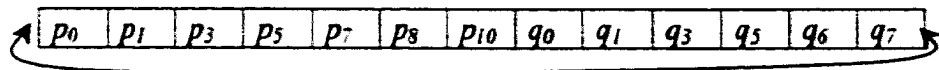
4. In a doubly linked list with header node, connect the inner elements set based on counter clockwise movement through the digital curve. Do the same for the outer

p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}

q_0	q_1	q_3	q_5	p_6	q_7

elements set. For the digital curve in figure 4.31 the inner and outer boundary lists includes the following elements, in order, respectively:

5. Connect the last element of the inner (outer) set to the first element in the outer (inner) set. We now have a circularly doubly linked list of all the elements, which is called the candidate list. The elements of the candidate list are elements in the candidate set. For the elements in figure 4.31 the candidate list is as follows:



4.3.1.2 Discussion of Phase 1

Each element in the candidate list may impose some constraint on the passing area. Some of these elements may impose constraints at certain radii and not at others. Also, some elements won't impose constraints at any radius. The conditional critical range (section 4.2.3.1) of each element determines the radius range where that element constrains the passing area. The connectivity of the elements within the candidate list helps define each element's neighbors; based on the strict inner and outer boundary rules (section 4.2.3.2), the conditional critical range can be determined. Since elements in the inner and outer sets may impose constraints on elements of the other (based on the loose inner and outer boundary rules, section 4.2.3.2), each element in the inner and outer set is connected via the doubly linked list implementation of the candidate list.

4.3.1.3 Phase 2 – Computing the Minimum Preimage Arc, r_{min}

Phase 1 (previous section) performed the initial phase of the algorithm. At its completion all possible critical elements are obtained. In this section, the algorithm computes r_{min} , the radius of the minimum radius for which a passing area exists, and determines the first critical set for which the radius equals r_{min} . The first critical set, labeled in section 2 as CS_1 , is used as the basis to determine the next critical set, CS_2 , and similarly for CS_2 determining CS_3 and so on until all critical sets along with their critical ranges are identified.

6. The minimal radius circular interpretation, r_{min} , is computed in $O(n)$ as was computed by O'Rourke (1986). This algorithm uses linear programming to determine minimum circular separability of two point sets.
7. Compute the α -infinite hull of the digital curve where $\alpha = 1 / r_{min}$. Let the *critical list* be a doubly linked list containing all elements in the critical set (i.e. along the passing area) of the α -infinite hull. We will refer to the elements in the current critical list as CS. The implementation details of the critical list are presented in the discussion section of phase 3.

4.3.1.4 Phase 3 – Updating the Critical Range (Iteration)

8. Traverse the candidate linked list and compute for each critical element its conditional critical range. As defined earlier, the conditional critical range of element e (denoted by $\langle e \mid e^-, e^+ \rangle$) is the range in parameter space (e.g. radius, r , for circular arcs) where e will be on the passing area perimeter predicated on the fact that the previous and successive elements, e^- and e^+ , respectively, are on the passing area. See section 4.2.3 for a complete discussion.
9. $r_{current}$ is initialized to r_{min} .
10. Find the minimum radius value where an element present in the critical set will exit the critical set. This was defined in section 4.2.3 as $r_{exit}(CS) = \min_{e \in CS} \langle e \mid e^-, e^+ \rangle_{max}$. Also find the radius value where an element not in the critical list will enter the critical list defined as $r_{enter}(CS) = \max_{e \notin CS} \langle e \mid e^-, e^+ \rangle_{min} \cdot r_{critical}$, the radius where a critical change takes place, is set to $\min(r_{exit}, r_{enter})$.

As the radius increases elements may enter and/or exit the critical set causing the members of the current critical set to change. The radial value at which this takes place is the minimum of $r_{enter}(CS)$ and $r_{exit}(CS)$, the minimum radial value where an element enters and exits the critical set CS .

11. Output all elements in CS along with its critical range, $\langle CS \rangle$, which is $[r_{current}, r_{critical})$.

12. If $r_{critical} = r_{exit}$, remove from the critical list all element(s) e such that $\langle e | e^-, e^+ \rangle = r_{exit}$. If $r_{critical} = r_{enter}$, add all element(s) e where $\langle e | e^-, e^+ \rangle = r_{enter}$.
 $r_{current}$ is set to $r_{critical}$.
13. If CS is not empty goto step 8 (beginning of this section)

4.3.1.5 Discussion of Phase 3

Two lists have been defined: the candidate list and the critical list. Every element in the critical list must also be an element in the candidate list. In order to eliminate redundant data structures and to simplify the implementation, all elements in the candidate list are equipped with two pointer records (each containing previous and next node pointer fields). One record is used to define the connectivity of the candidate list and the other defines the connectivity of the critical list.

The candidate list contains elements that constrain the passing area where the radius is equal to or greater than $r_{current}$. Elements that have constrained the passing area in an earlier iteration of phase 3 but presently, and in future iterations will not, are eliminated from the candidate list. Thus, in step 12 when elements are removed from the critical list they may be removed from the candidate list, as well.

4.3.2 The Algorithm Step By Step

The following figures illustrate the algorithm from section 4.3.1 as it iterates and determines the 8 critical sets for the digital curve. These iterations occur in phase 3 of the algorithm. Figures 4.32–4.39 (with a summary table in figure 4.40) show the critical elements as computed at step 11 of the algorithm. Those pixel endpoints that are shown in black are critical elements from the critical list; those that are shown in white are general members of the candidate list.

Critical Set #1

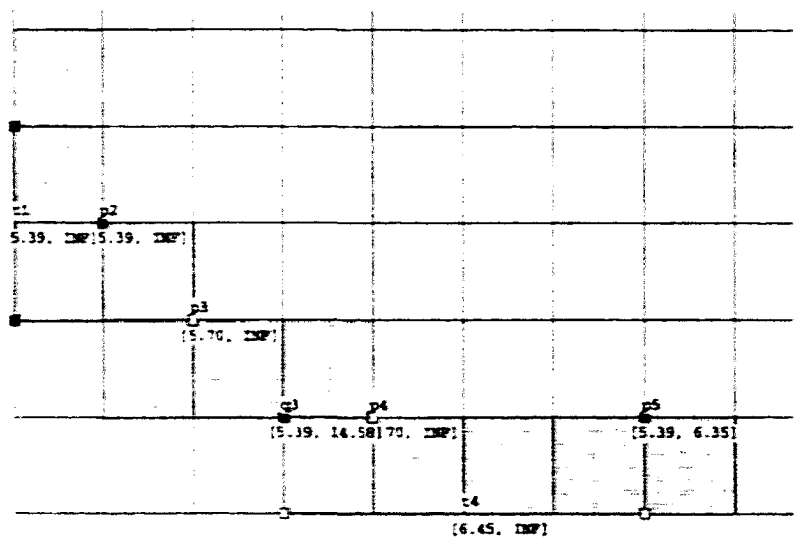


Figure 4.32. $r_{max} = 4.47$. The critical set when $4.47 \leq r < 5.39$ is $\{p_2, p_5, q_3\}$. Note that element t_1 will enter the critical set at 5.39.

Critical Set #2

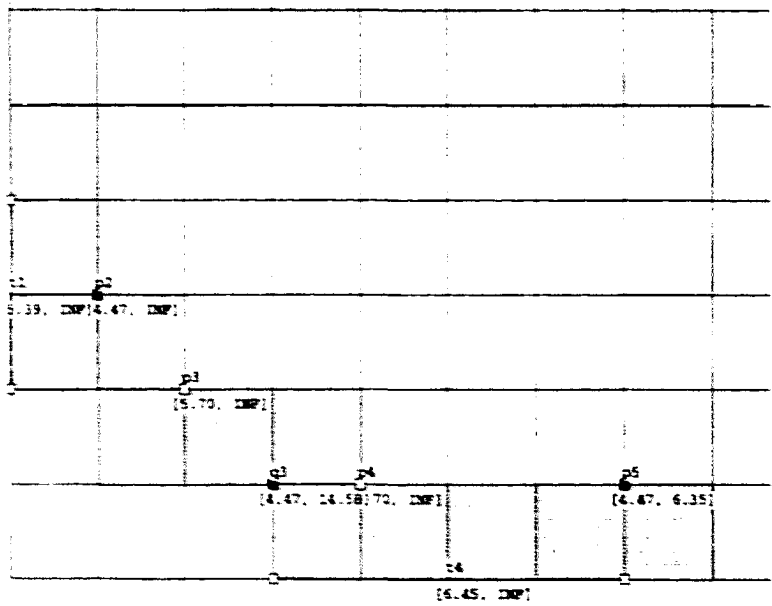


Figure 4.33 The critical set when $5.39 \leq r < 5.70$ is $\{p_2, p_5, t_1, q_3\}$. Element p_3 and p_4 will enter the critical set at 5.70.

Critical Set #5

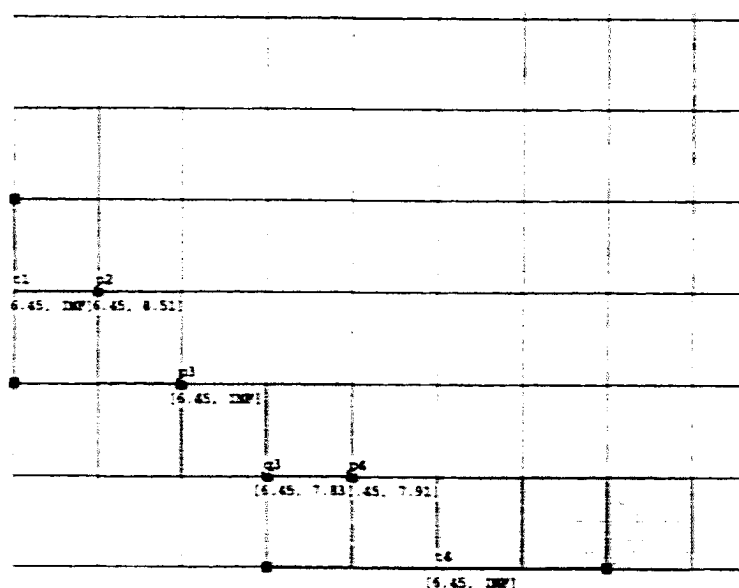


Figure 4.36 The critical set when $6.45 \leq r < 7.83$ is $\{p_2, p_3, p_4, t_1, q_3, t_4\}$. Element q_3 will exit the critical set at 7.83.

Critical Set #6

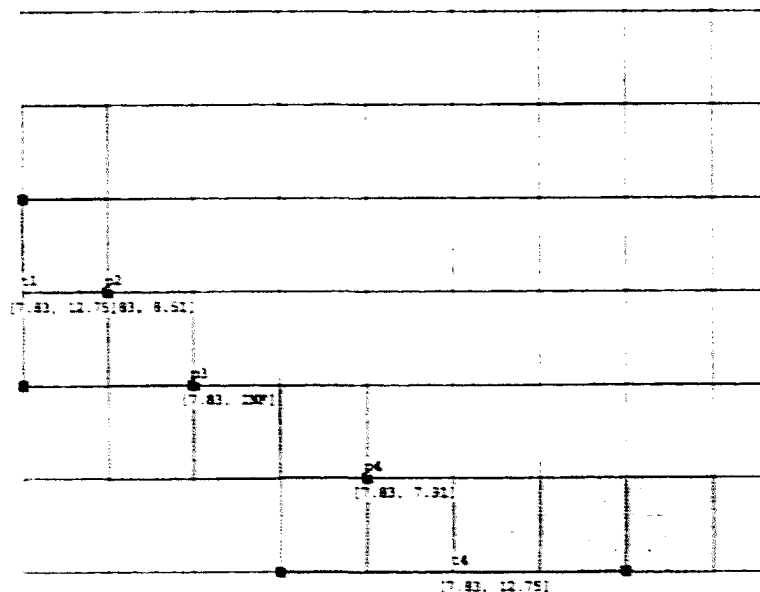


Figure 4.37. The critical set when $7.83 \leq r < 7.91$ is $\{p_2, p_3, p_4, t_1, t_4\}$. Element p_4 will exit the critical set at 7.91.

Critical Set #7

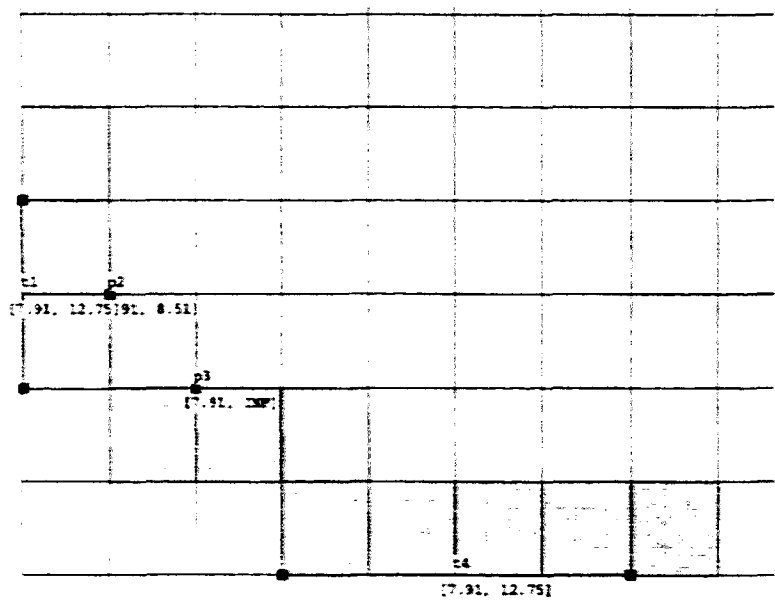


Figure 4.38. The critical set when $7.91 \leq r < 8.51$ is $\{p_2, p_3, t_1, t_4\}$. Element p_2 will exit the critical set at 8.51.

Critical Set #8

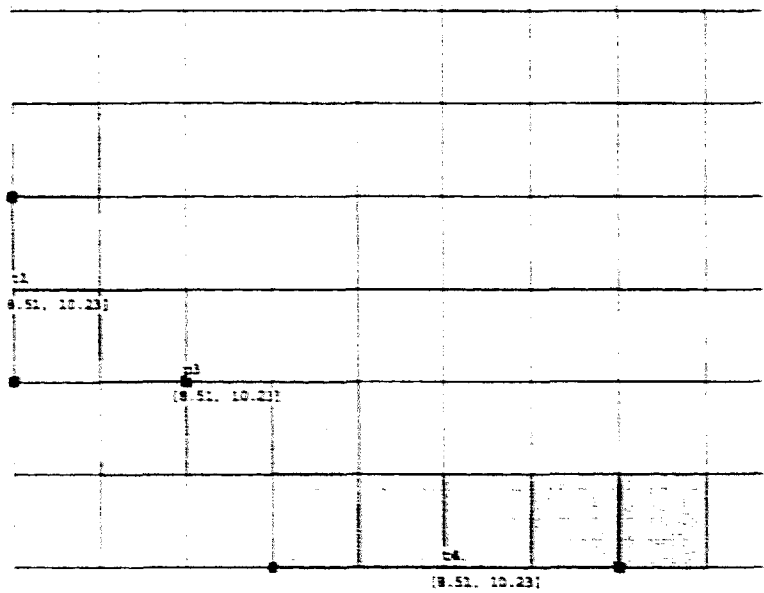


Figure 4.39. The critical set when $8.51 \leq r < 10.23$ is $\{p_3, t_1, t_4\}$. Elements p_3, t_1, t_4 will exit the critical set at 10.53. The minimum number of elements in a critical set is 3.

Radius Range		Critical Set
4.47	5.39	p_2, p_5, p_3
5.39	5.70	p_2, p_5, t_1, q_3
5.70	6.35	$p_2, p_3, p_4, p_5, t_1, q_3$
6.35	6.45	p_2, p_3, p_4, t_1, q_3
6.45	7.83	$p_2, p_3, p_4, t_1, q_3, t_4$
7.83	7.91	p_2, p_3, p_4, t_1, t_4
7.91	8.51	p_2, p_3, t_1, t_4
8.51	10.23	p_3, t_1, t_4

Figure 4.40. Table summarizing the critical sets and their conditional ranges.

Having overlooked the set of figures 4.32–4.39 of this section, summarized here in the table (figure 4.39), it is interesting to note that p_1 and q_2 are not critical elements for any critical set. In addition, the size of the critical sets seem tractable and bounded by a finite number. As to what that number is exactly and to what extent, if any, that depends on the input data of pixels, the research presented in the next section will shed some light on these relevant issues.

4.3.3. Algorithm Run Time

The run time efficiency of the algorithm described in section 4.3.1 and depicted in the previous section is now addressed. Phase 1 (sections 4.3.1.1 and 4.3.1.2) involves determining the inner and outer boundary of the digital curve and finding the corner points. This is achieved by walking along the digital curve one pixel at a time and determining the upper and lower regions of each pixel’s pixel area. Each pixel is visited

once. The corner points are determined by the relative position of three consecutive vertices along a boundary, which is also computed once per vertex. Since vertices have a linear relationship with the numbers of pixels, the corner point determination is also in $O(n)$. Finally, adding the candidate elements to the linked list is achievable in $O(n)$. Thus phase 1, has an $O(n)$ run time.

Phase 2 (section 4.3.1.3) involves the computation of the minimal radius circular arc that digitizes to the digital curve. O'Rourke (1986) has shown that this is computable in $O(n)$ by using linear programming.

Phase 3 (sections 4.3.1.4 and 4.3.1.5) involves iterating through all critical sets. Each iteration requires $O(n)$ computations to determine the conditional critical ranges of each element in the candidate list. The number of iterations, though, appears to be bounded by a fixed constant. Simulations were run on random digital circular arcs generated with a radius up to 2000 pixels and up to 6400 pixels in the digital curve. The maximum number of critical sets was 25 and the average critical set count was approximately 10.7. The number of critical sets of a digital curve does not have a correlation with the curve's radius, perimeter or angular span as the correlation function was less than 0.02 for all data set pairs. The correlation function of a vector pair, X and Y ,

$$\text{is } \rho_{X,Y} = \frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y}, \text{ where } -1 \leq \rho_{X,Y} \leq 1 \text{ and } \text{Cov}(X,Y) = \frac{1}{n} \sum_{i=1}^n (x_i - \mu_x)(y_i - \mu_y).$$

When $\rho_{X,Y} \approx 0$ there is no correlation between the two vectors. A scatter plot of the number critical sets as the radius and curve perimeter vary is shown in figures 4.41 and 4.42, respectively.

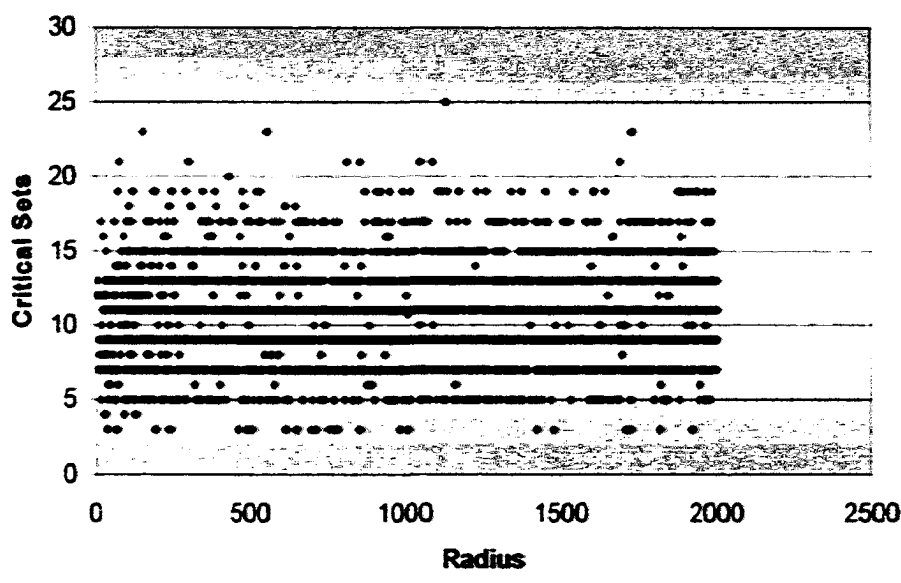


Figure 4.41. The number of critical sets plotted against the curve's radius. The critical set number seems independent of the radius of the digital curve

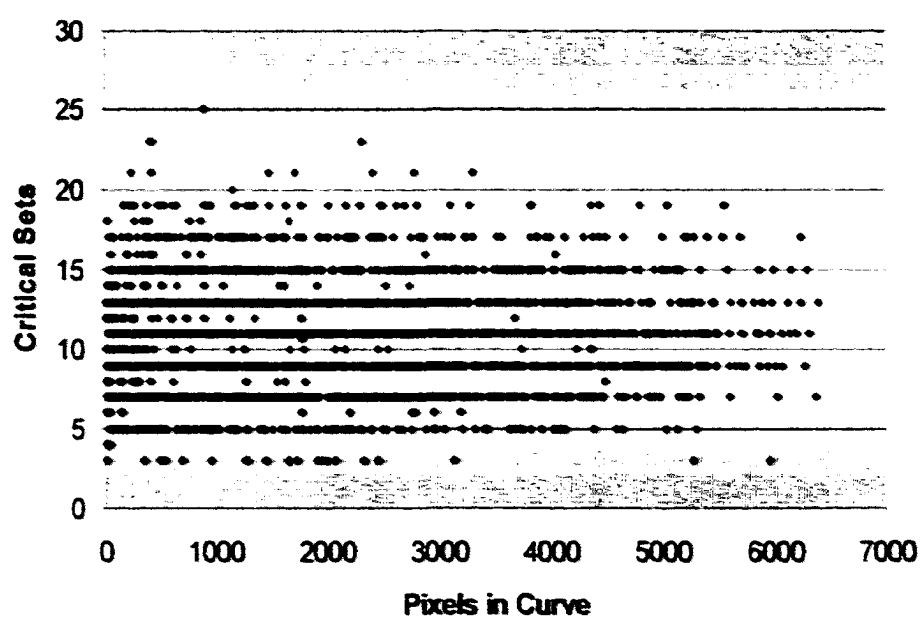


Figure 4.42. The number of critical sets plotted against the digital curve's size. The critical set number seems independent of the size of the digital curve.

4.4 The Shape of the Domain

The domain for digital circular arcs is parameterized by circle center (x,y) and radius, r . The center of a circle that digitizes to a point on the digital curve must lie within distance r of the digital point (or of a point on the plane that under the digitization scheme would digitize to that digital point on the grid). Similarly, a circle in the domain must not digitize to points not in the digital curve. Thus, where r is fixed, the domain is the intersection of discs with radius r centered at digital points inside the digital curve and disc's complements centered at digital points outside the curve. See figure 4.43.

Where r is unknown the domain is the intersection of cones centered at digital points inside the digital curve and cone's complements centered at digital points outside the curve (Edelsbrunner, 1987, Worring 1995). See figure 4.44. If at height h the cone's radius is r the intersection of the cones, cones' complements and plane at height h will yield the domain for radius r .

The shape of the domain where r is fixed, the intersection of discs and disc complements with radius r , is clearly bounded by circular arcs of with radius. The shape of the domain where r is unknown is similarly the cone and cones' complements intersection. We no investigate the shape of the projection of the cone and cone intersection onto the plane. This projection represents the circle centers whose circles digitize to the digital curve. We first review some concepts from computational geometry.

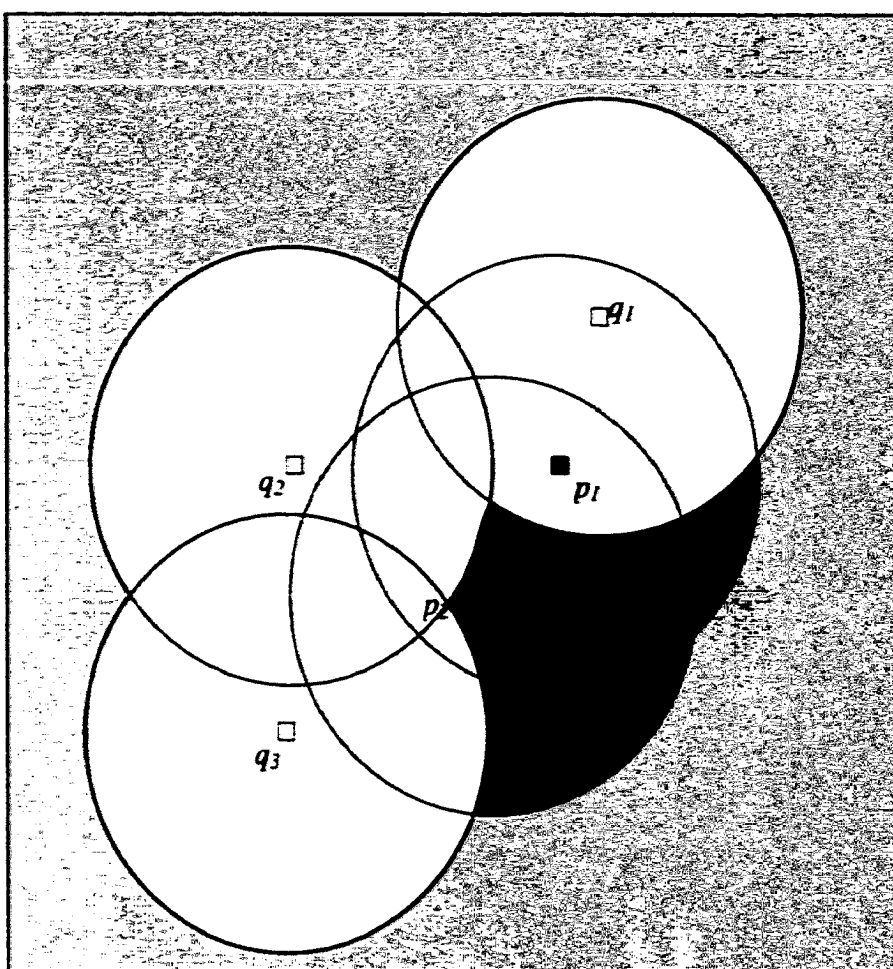


Figure 4.43. The intersection of discs and disc complements centered at points on the inside (p_i) and outside (q_i) of the digital circular arc is the digital curve's domain.

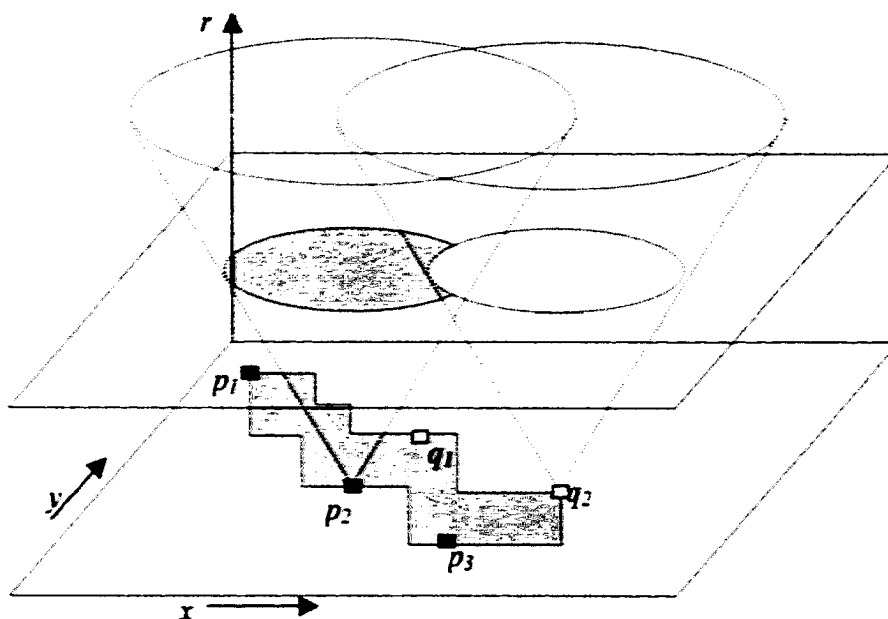


Figure 4.44 The intersection of cones and cone complements centered at points on the inside (p_i) and outside (q_i) of the digital curve yield the complete circular arc domain of the digital curve.

For S , a set of points in $\mathbb{R}^2$, where $s \in S$ and $d(x, y)$ is the (Euclidean) distance between points x and y we have (Worring, 1995):

Closest Point Voronoi Polygon: $V(s, S) = \{x \mid \forall s' \in S : d(x, s) \leq d(x, s')\}$

(See figure 4.44a)

Furthest Point Voronoi Polygon: $V(S, s) = \{x \mid \forall s' \in S : d(x, s') \leq d(x, s)\}$

(See figure 4.44b)

Generalized Voronoi Polygon $V(S_P, S_Q) = \{x \mid \forall p \in S_P, \forall q \in S_Q : d(x, p) \leq d(x, q)\}$

(See figure 4.44c)

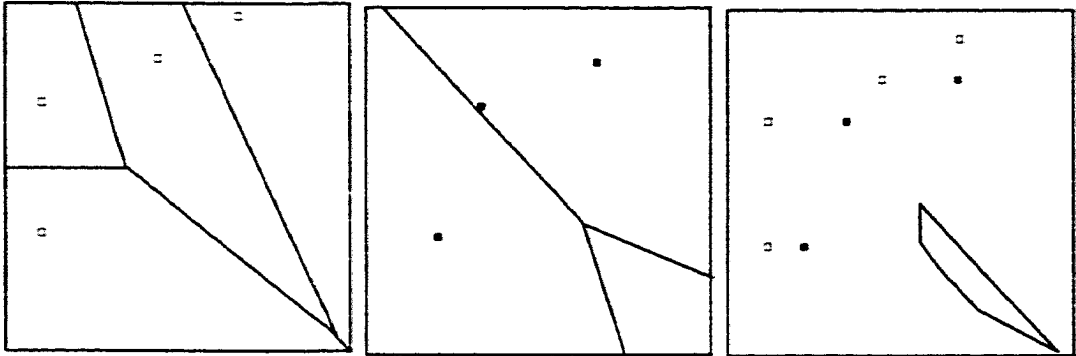


Figure 4.45. The closest point Voronoi diagram of S_Q (a) and the furthest point Voronoi of S_P (b) are shown. The circular arc domain of a digital curve under all radii is the generalized Voronoi polygon (c) of the inner and outer points of that curve (from Worring, 1995).

Fisk (1986) showed that the projection of the 3 dimensional cone intersection onto the plane is the generalized Voronoi diagram of two point sets, S_P and S_Q . S_P is the set of digital points in the digital curve and S_Q is the set of neighboring points in the curve's background. The generalized Voronoi diagram is the projection onto the (x,y)-plane of all circle centers – with no distinction made between circle centers of different radii.(see figure 4.45c).

The generalized Voronoi diagram is the domain because the passing area problem may be reformulated (as discussed in section 3.3) in terms of the circular separability

problem: find the circles, if any, that separate two point sets, S_P and S_Q . Circle center (x,y) and radius (r) parameterize the set of all such circles. Since we are considering the projection onto the (x,y) - plane, the circles are parameterized by circle center, alone.

Let us consider the separability problem of two points $p \in S_P$ and $q \in S_Q$. The center, c , of any circle that intersects the two points, p and q must lie along the perpendicular bisector of those points, $p \perp q$. This is because that circle must be equidistant to both points. Therefore, the half-space that includes point, p , that is bounded by the $p \perp q$ contains all circle centers because for every circle center in that region $d(c,p) \leq d(c,q)$. The intersection of all half spaces $d(c,p) \leq d(c,q) \mid p \in S_P, q \in S_Q$ will result in the center of circles that includes all point in S_P and exclude all points in S_Q . This intersection will result in a domain that is bounded by line segments that are along some $p \perp q$ where $p \in S_P$ and $q \in S_Q$.

We will now show that the domain is dependent on the digitization scheme used to convert the real world curve to a digital curve.

4.4.1 Digitization Models

An image is a mapping of the world scene in R^2 onto the digital image in I^2 . The pixels in the image are set to a color determined by the digitization model. Researchers use two common digitization models: OBQ (Object Boundary Quantization) and GIQ (Grid Intersection Quantization). OBQ (figure 4.45a) will turn on any pixel that is within the object's boundary in the scene whereas GIQ (figure 4.45b) will turn on the pixels

nearest the object's boundary crossing over either a vertical or horizontal grid line (Freeman, 1970; Dorst and Smeulders, 1984).

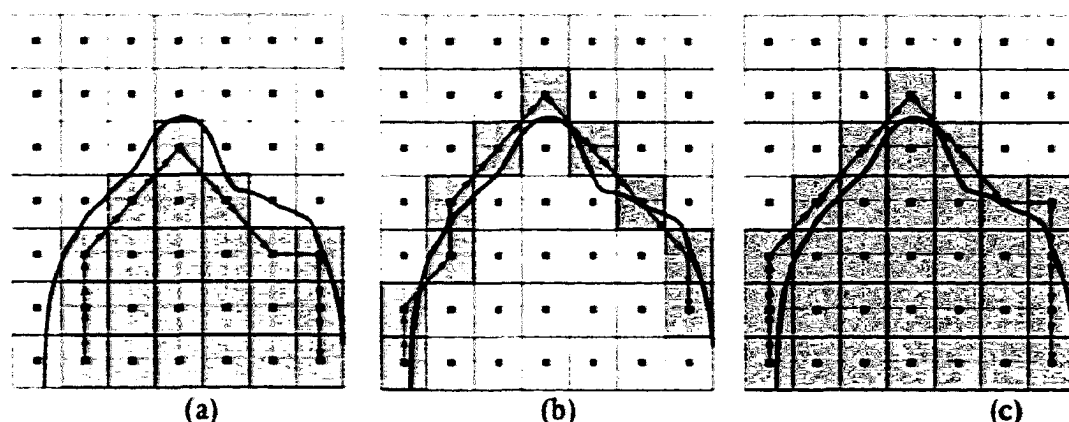


Figure 4.46. An object bounded by a curve is digitized under three schemes: (a) OBQ, (b) GIQ and (c) luminescence. The arrowed lines show the 8-connected boundaries.

As noted by Rosenfeld (1974), OBQ is more useful for digitizing objects in the scene and GIQ for digitizing curves. A curve cannot be digitized via OBQ since a curve does not have an “inside” to test whether a pixel’s grid point is in it. GIQ, though, can digitize objects by digitizing their boundaries and filling them in. For the purpose of digital curve analysis a digital curve is needed. This curve can be generated directly with GIQ or by first digitizing using OBQ and then detecting the object’s boundary using a boundary-searching algorithm.

Another digitization scheme is the luminescence model (figure 4.45c) where the digitization function is similar to the actual digitization process that takes place when an image is scanned or a digital picture is taken. During imaging, pixels are set “on” based

on the luminescence (i.e. the light recorded by the scanning device) of the region in the scene that corresponds to the pixel in the image. If the luminescence exceeds some threshold (possibly adjustable by the scanner operator) the pixel is set "on". In future discussions we assume that a pixel is set "on" when the minimal non-zero luminescence value is attained. For example, when using a 256 gray scale scanner where value 0 represents no luminescence and 255 maximal luminescence, a pixel is set on when its luminescence is one or greater. (Note that figure 4.45c assumes a monochrome digitization function.)

4.4.2 Effects on the Domain

As noted in the introduction to this subsection, Fisk (1986) and Worring (1995) observed that the generalized Voronoi diagram of S_P and S_Q , the set of points inside and outside the digital curve respectively, is the digital curve's domain. The generalized Voronoi diagram of the point sets is bounded by line segments that lie along the perpendicular bisector of p and q where $p \in S_P$, $q \in S_Q$. In reconstructing the preimage of a digital curve digitized under OBQ, the circular separability problem can be stated to find the circle that separates S_P and S_Q , which only include points and not line segments. Under OBQ line segments do not constrain the passing area since the only criteria for determining whether a pixel is on the curve is whether the pixel's center point is located within the the object bounded by the curve, as shown in figure 4.45a. Under OBQ although the boundary may intersect other pixel areas, those pixels will not be on the digital curve unless their center points are within the object's boundary.

Under the luminescence digitization model, however, any pixel whose pixel area is intersected by the object's boundary will be on the digital curve, as in figure 4.46. Therefore, the object boundary must intersect the pixel area of every pixel on the digital curve and *not* intersect the pixel area of any pixel not on the digital curve. This non-intersection constraint results in the requirement that the boundary curve must not intersect either the inner or outer boundaries. Earlier, in section 3.4.1, it was shown that the inner boundary is equivalently constrained by the endpoints of line segments along it. The outer boundary, though, is constrained by line segments endpoints, as well as line segments at digital stationary points, as shown in figure 4.46.

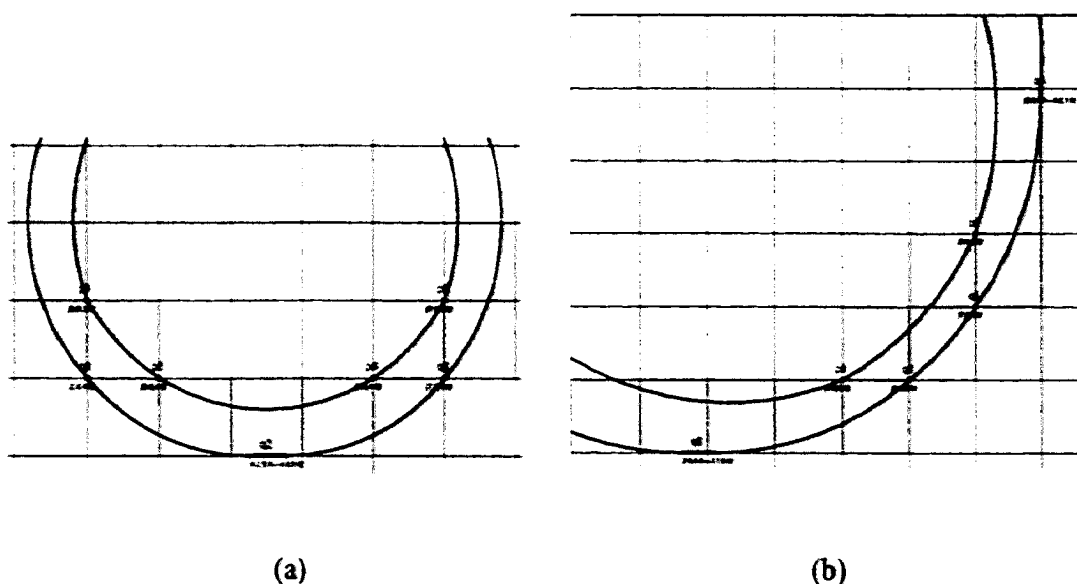


Figure 4.47. Only edges that are at the digital stationary points may contribute to the α -infinite hull. In (a) this occurs on the bottom edge and in (b) occurs on the bottom and rightmost edge.

Thus, under the luminescence model, the domain is the generalized Voronoi diagram of S_p and S_q , the sets of constraining elements (i.e. points and line segments) of the inner and outer boundaries, respectively. When p and q are both points the region where $d(x,p) \leq d(x,q)$ is bounded by $p \perp q$ (figure 4.47a). Similarly, when p and q are both edges that region is bounded by the angular bisector of the angle between the linear extension of p and q (figure 4.47b). However, when p is a point and q is an edge the region where $d(x,p) \leq d(x,q)$ is bounded by a parabola with focus p and directrix q (figure 4.47c). Therefore, the domain which is the intersection of those regions is bounded by line segments and parabolic curves, as shown in figure 4.48.

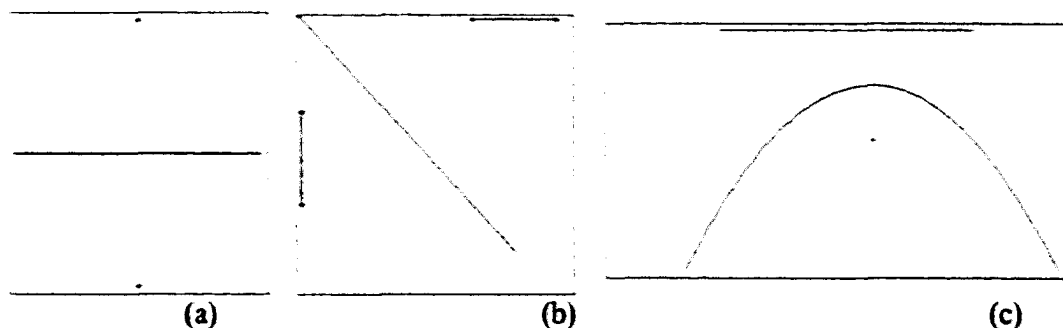


Figure 4.48. The boundary of the region where $d(x,p) \leq d(x,q)$ is the medial curve of element p and q . When p and q are both points the medial curve is the perpendicular bisector of p and q (a). When they are both line segments (b) it is the angular bisector of the angle made by the edges extensions. When one is a point and the other a line segment (c) the region boundary is a parabola with the point as focus and the linear extension of the edge as directrix.

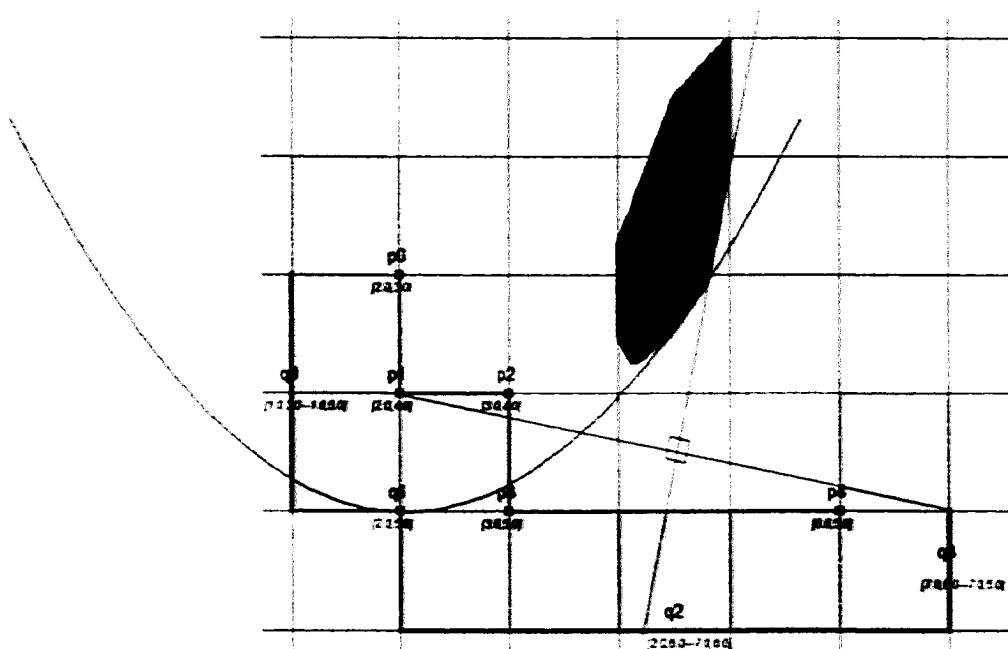


Figure 4.49. The domain of the digital curve with the parabola with focus p_1 and directrix q_2 superimposed over the parabolic portion of the domain. The perpendicular bisector of p_2 and an endpoint of q_2 defines another domain boundary, as shown.

Chapter Five

Gray Scale Image Constraints on the Passing Area

Abstract

An extension of the convex hull, the infinite convex hull, was defined (chapter 2) to determine a digital line segment's minimal passing area. This passing area was then used to compute a digital line segment's domain, the set of all possible lines segments that digitize to the digital line, defined in (m,b) parameter space. Similarly, the α -hull that was defined by Edelsbrunner (1983) was extended to the α -infinite hull (chapter 3) and used to compute the passing area of any digital circular arc in a run time of $O(n)$. From this information, the domain may be determined in constant time.

The constraint that binary digital curves impose on the passing area was shown to be specified by the digital curve's pixel boundary and α -infinite hull. In this chapter, the additional constraints that gray scale information imposes on a digital curve are shown. Incorporating these additional constraints produces a passing area that is significantly smaller and, thus, more accurately specifies the domain. Furthermore, the run time complexity remains $O(n)$.

5 Gray Scale Images

In earlier sections (2-5) it has been shown that the domain of the digital boundary of an image may be determined by computing the infinite hull of the digital boundary's inner and outer boundaries. Since gray scale images contain significantly more information than their binary counterparts, this additional information may be used to tighten the inner and outer boundary constraints. This section presents a methodology to take advantage of the gray scale information to determine a more localized passing area and domain of the preimage curve.

As mentioned earlier (section 4.4) the digitization model used in this paper is luminescence digitization. Luminescence digitization in a binary image turns a pixel "on" if and only if the intersection of the pixel area and object in the world scene is greater than some threshold. For example, in figure 5.1 a star is superimposed on the grid. The background is assumed to be white and the foreground object is black. With minimal threshold each pixel that has some portion of the star within its pixel area is turned on (shown in gray). In a gray scale image each pixel is set to a gray value corresponding to the region of intersection of the pixel area and the world scene object. Thus, pixels that are totally enclosed within the object are darkest, pixels partially enclosed are lighter and pixels outside of the object are lightest. See figure 5.1b where a gray scale circle is shown with the outline of the disc that was used in the image's digitization.

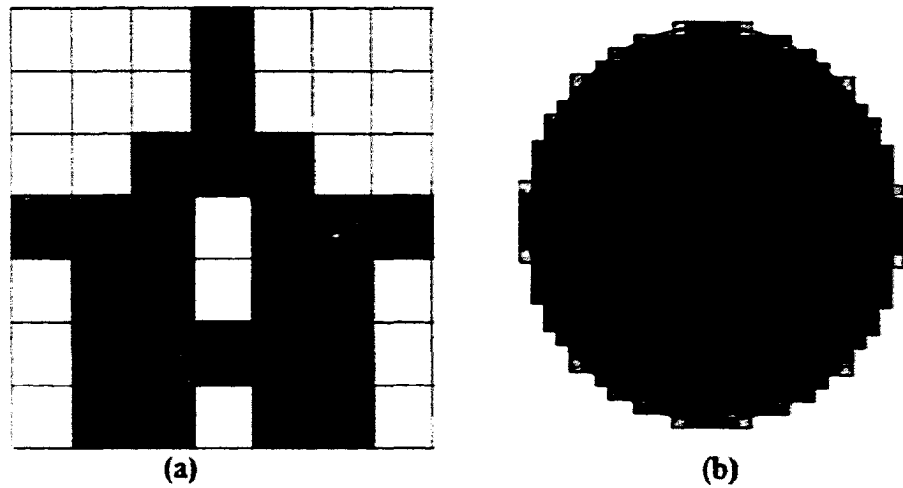


Figure 5.1. A star digitized as a binary image with minimal threshold (a). Any pixel whose pixel area intersects the star is turned on. The gray scale image of a circular disc is shown (b) with the circular boundary superimposed over the grid. The percentage of gray of each pixel is the fraction of its pixel area that intersects the circular disc.

The gray level of each pixel embeds the information of the percentage of the pixel area that the object occupies. Thus, the digital representation of the object's boundary will be the pixels with a gray value greater than solid white and less than solid black. As an example, pixels found on the upper boundary of the circle from figure 5.1b are shown embedded in figure 5.2. Using this digital boundary we will now present a technique that recovers the passing area more precisely than with a simple binary digital boundary. We first present the constraints as they apply to gray scale curves in general and then apply those results to curves that are digital line segments and circular arcs.



Figure 5.2. A portion of the digital boundary from figure 5.1b. The circular arc intersects every pixel in the digital boundary.

5.1 Constraint Imposed on the Passing Area by the Pixel's Gray Level

As discussed in section 2, the passing area of the digital boundary of a digital binary image must lie between the inner and outer pixel boundaries. With a gray level image this constraint must also hold since the object boundary must intersect every pixel in the digital boundary and not intersect pixels not on the digital boundary, as shown in figure 5.3. An additional constraint may be derived since the curve must traverse each pixel such that the proportion of the object area found within each pixel must correlate the gray level of that pixel. These concepts are now formalized with the following definitions.



Figure 5.3. The object boundary must be found between the inner and outer boundaries shown. This is the initial passing area constraint for binary images.

Definition: The *pixel median*, or *median*, of three 4-connected pixels, in order, p_1 , p_2 and p_3 , is the ray along the medial line of the common boundary of pixels p_1 and p_2 and the common boundary of p_2 and p_3 . The ray begins at the midpoint of the common boundary of p_2 and its neighboring interior pixel. (See figure 5.4)

The pixel median has a different form for each of the three pixel configurations: linear, concave and convex as shown in figure 5.4 (next page). These three pixel configurations represent the three instances that three 4-connected pixels may be connected. Clearly, any curve traversing the middle pixel, p_2 , and not intersecting either its upper or lower boundary must intersect the pixel median at some point. It will be shown that the precise intersection point or range of intersection points along the median may be determined as a function of the pixel's gray level.

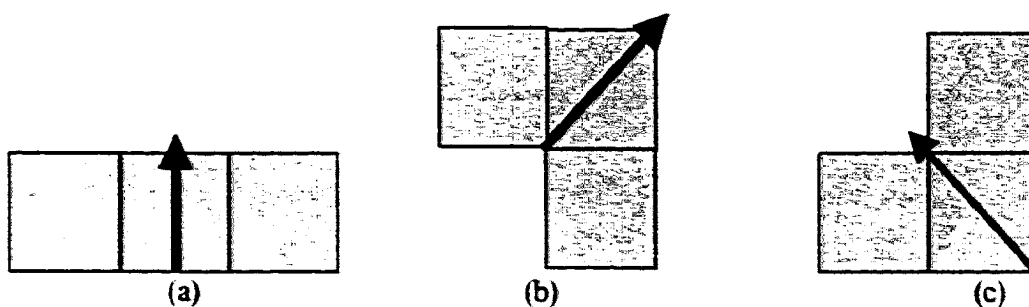


Figure 5.4. The pixel median of the middle pixel is shown for three connected pixels that are linear (a), convex (b) and concave (c). The interior of the object is below the curve portion shown and is in the opposite direction of the median arrow.

In the following section these observations are applied to find the passing area of a gray scale curve that represents line segments and circular arcs. For both digital line segments and digital circular arcs three successive pixels may be locally linear, concave or convex. Each of these three instances is studied separately for both digital line segments (section 5.2) and circular arcs (section 5.3).

5.2 Objects Bounded By Line Segments

In this section the constraints imposed on the upper and lower boundaries of a digital gray-scale line segment are explored for linear, concave and convex pixel configurations.

5.2.1 Linear Pixel Configurations

Consider a linear sequence of pixels as in figure 5.4a with center pixel that has area A ($0 \leq A \leq 1$) that is covered by an object with a linear boundary, as in figure 5.5. The boundary line must intersect the median at point p as shown. Any linear object boundary intersecting this point will cause the pixel to be covered by area A of the object. This follows from the symmetry of the triangles shown in that figure. This point is referred to as the Δ -point and for a linear pixel configuration $\Delta = A$. This is formalized with the following definition:

Definition. A point along the median that the preimage curve must intersect is the *pixel's Δ -point*. This point is parameterized by Δ , the distance of the point from the median's start point.

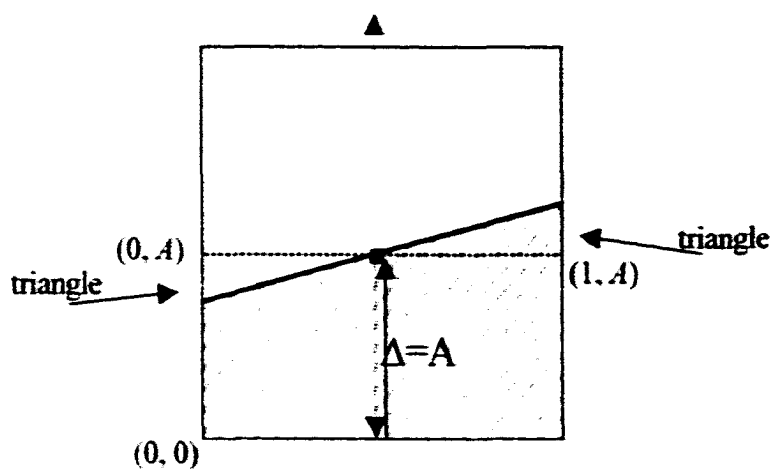


Figure 5.5. The two triangles shown are reflections of each other having equal area. Thus, in general, the linear boundary of any region with area A will intersect the Δ -point where $\Delta = A$.

5.2.2 Convex Pixel Configurations

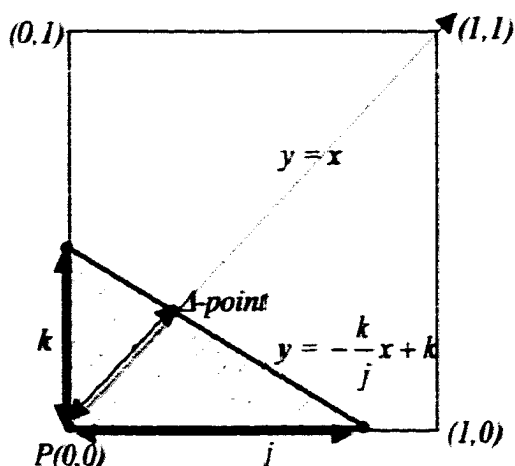


Figure 5.6. Δ is the distance from P to Δ -point. P is at the origin and the Δ -point is the intersection of the median (with equation $y = x$) and the line with equation $(y = -\frac{k}{j}x + k)$. (The pixel shown is a rotation of the pixel shown in figure 5.4 and was chosen for the equational simplicity.)

When the pixel sequence is concave or convex a single Δ -point does not exist. Rather, a *portion* of the median must be intersected by the preimage curve. See for example figure 5.6 where the Δ -point for a specific slope and area is shown. As the slope changes the Δ -point will also change. This range of Δ -points is referred to as the Δ -segment and is defined as follows:

Definition. A line segment along the median that the preimage curve must intersect is the *pixel's Δ -segment*. This segment is parameterized by $\Delta_{\min}$ and $\Delta_{\max}$, the minimum and maximum distances of the segment from the median's start point, respectively.

Depending on the slope of the object boundary the value of Δ will range from a minimum value, $\Delta_{\min}$, to a maximum value, $\Delta_{\max}$. Referring to figure 5.6 k and j are the y -intercept and x -intercept of the preimage curve and $0 \leq k, j \leq 1$. We now solve for Δ in terms of A and k . By solving for the second derivative of $\Delta = f(A, k)$ where A is the intersection area and k is the y -intercept the values of $\Delta_{\min}$ and $\Delta_{\max}$ are determined.. See figure 5.6 where the following statements hold:

$$1) \quad A = \frac{kj}{2} \quad (j \text{ and } k \text{ are the boundary line's } x\text{-intercept and } y\text{-intercept, respectively.})$$

$$2) \quad \text{Median line equation is } y = x.$$

$$3) \quad \text{Boundary line equation is } y = -\frac{k}{j}x + k$$

$$4) \quad \text{Solving for } x \text{ and } y \text{ from equation (2) and (3) where the } \Delta\text{-point} =$$

$$(x, y) \text{ and } \Delta = \sqrt{x^2 + y^2} \text{ we have } \Delta\text{-point} = \left(\frac{jk}{j+k}, \frac{jk}{j+k} \right) \text{ and } \Delta = \frac{\sqrt{2}jk}{j+k}.$$

$$\text{Solving for } \Delta \text{ in terms of } A \text{ and } k \text{ (from 1 and 4) we have } \Delta = \frac{2\sqrt{2}Ak}{2A+k^2} \text{ and thus}$$

$$\text{stationary points exist where } \frac{d\Delta}{dk} = \frac{2\sqrt{2}A(2A-k^2)}{(2A+k^2)^2} = 0. \text{ Solving for } k \text{ we obtain } k =$$

$$\sqrt{2A}. \text{ This is a maximum since the } 2^{\text{nd}} \text{ derivative } \frac{d^2\Delta}{d^2k} = -\frac{4\sqrt{2}Ak(-6A+k^2)}{(2A+k^2)^3} \text{ when}$$

$$\text{evaluated where } k = \sqrt{2A} \text{ the derivative equals } -\frac{1}{2\sqrt{A}} \text{ which is negative. Thus, where}$$

$$k = \sqrt{2A} \text{ the maximum } \Delta\text{-value } \Delta_{\max} = \sqrt{A}. \text{ Since only one stationary point exists for } k \geq$$

$$0, \text{ the value of } \Delta \text{ for all other values of } k \text{ must decrease with } k. \text{ Therefore, the furthest } k$$

value from $\sqrt{2A}$ will yield $\Delta_{\min}$. This value will be the maximum or minimum value of k whose region has area A . Clearly, this is where the line intersects a corner point of the pixel as shown in figure 5.7a. $\Delta_{\min}$ and $\Delta_{\max}$ thus equal $\frac{2\sqrt{2}A}{1+2A}$ and $\sqrt{A}$, respectively.

The following is the Mathematica representation of the solutions to the equations discussed above:

```
FullSimplify[Solve[{delta == Sqrt[x^2 + y^2], y == -k/j x + k, x == y}, {delta}, {x, y}],
{0 <= j <= 1 And 0 <= k <= 1}]
```

$\{\{\text{delta} \rightarrow -\frac{\sqrt{2}jk}{j+k}\}, \{\text{delta} \rightarrow \frac{\sqrt{2}jk}{j-k}\}\}$ (since delta must be positive the first solution

is rejected.

```
FullSimplify[Solve[{delta == Sqrt[2]jk/(j+k), A == (kj)/2}, {delta}, {j}]]
```

$\{\{\text{delta} \rightarrow \frac{2\sqrt{2}Ak}{2A+k^2}\}\}$

```
FullSimplify[Solve[{delta == 2Sqrt[2]Ak/(2A+k^2), k == Sqrt[2A]}, {delta}, {k}]]
```

$\{\{\text{delta} \rightarrow \sqrt{A}\}\}$ ($\Delta_{\max}$)

```
FullSimplify[Solve[{delta == 2Sqrt[2]Ak/(2A+k^2), k == 1}, {delta}, {j}]]
```

$\{\{\text{delta} \rightarrow \frac{2\sqrt{2}A}{1+2A}\}\}$ ($\Delta_{\min}$)

In figure 5.8 a plot of Δ as a function of the area, A , is shown. Refer to figure 5.6 where the maximum values for k and j are 1. Thus, the area A is at most $\frac{1}{2}$.

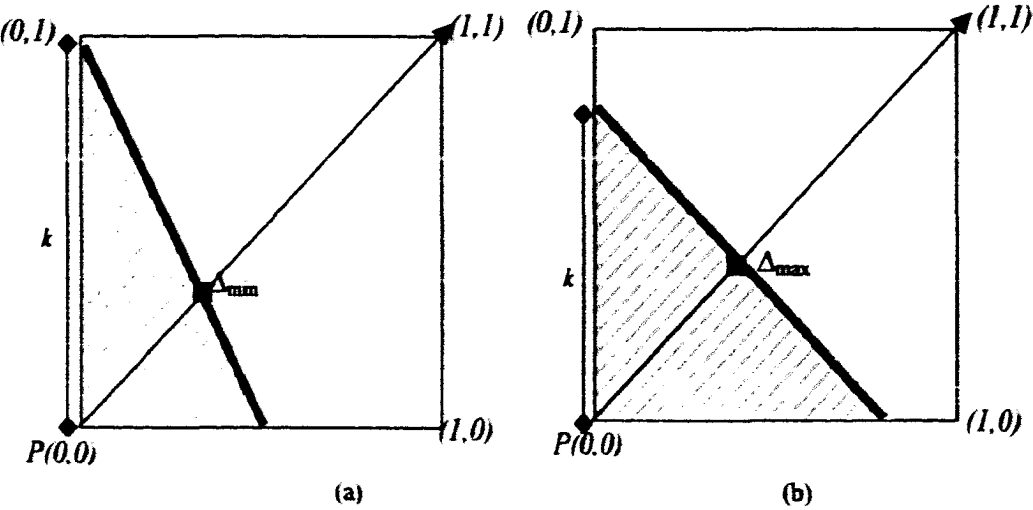


Figure 5.7. The values of $\Delta_{\min}$ (a) and $\Delta_{\max}$ (b) for a given area, A , are determined by the minimum and maximum Δ values of intersecting linear regions.

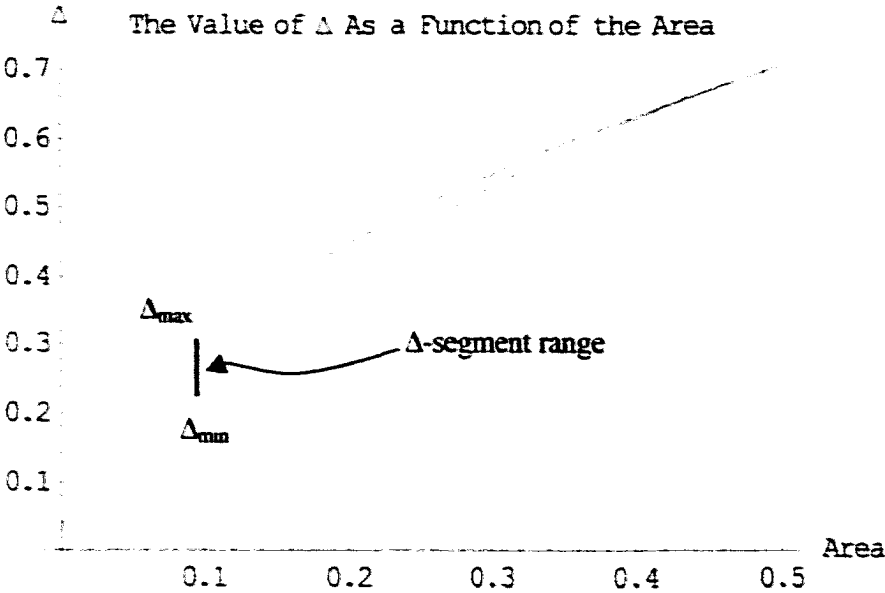


Figure 5.8. The values of $\Delta_{\min}$ and $\Delta_{\max}$ as the area, A , changes. The linear boundary intersecting a pixel with gray level 0.1 must intersect the median at a Δ value between $\Delta_{\min}$ and $\Delta_{\max}$, the Δ -segment range shown.

5.2.3 Concave Pixel Configurations

The values of $\Delta_{\min}$ and $\Delta_{\max}$ for the concave configuration of pixels are similar to the convex configuration. The unshaded region shown in figure 5.9 is equal the shaded region of figure 5.7. Specifically, the value of A in the concave configuration is equal $1 - A$ from the convex configuration. Similarly, Δ in this case equals $\sqrt{2} - \Delta$ from the convex case. With trivial substitution we have $\Delta_{\min}$ and $\Delta_{\max}$ equaling $\sqrt{2} - \sqrt{1 - A}$ and $\sqrt{2} - \frac{2\sqrt{2}(1 - A)}{1 + 2(1 - A)}$ (which is simplified to $\frac{\sqrt{2}}{3 - 2A}$). Refer to figure 5.9 where k and j range between 0 and 1 and thus area A ranges between 0.5 and 1. In figure 5.10 a plot of Δ as a function of the area, A , is shown.

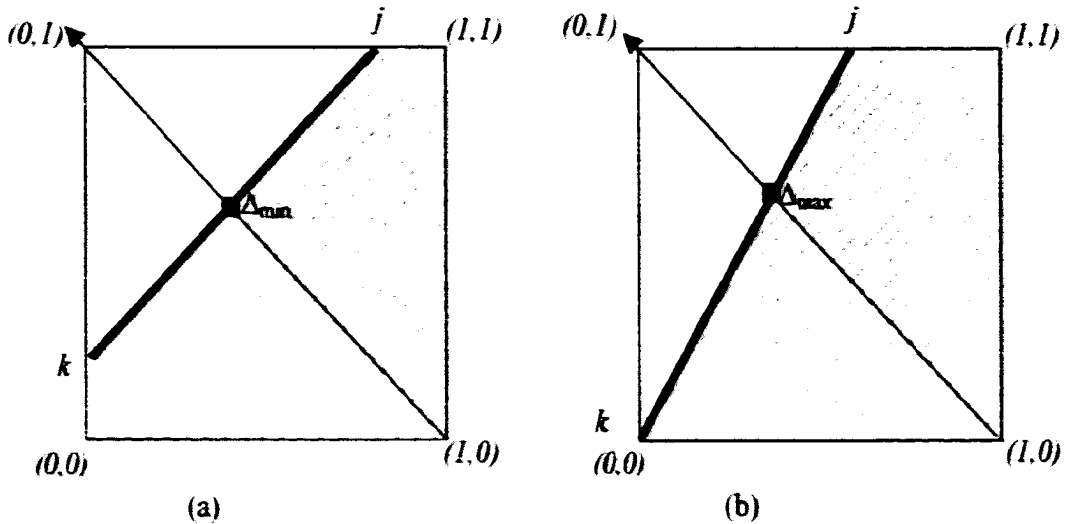


Figure 5.9. The values of $\Delta_{\min}$ (a) and $\Delta_{\max}$ (b) for a given area, A , are determined by the minimum and maximum Δ values of intersecting linear regions. The minimum value for A and Δ are $\frac{1}{2}$ and $\frac{\sqrt{2}}{2}$, respectively.

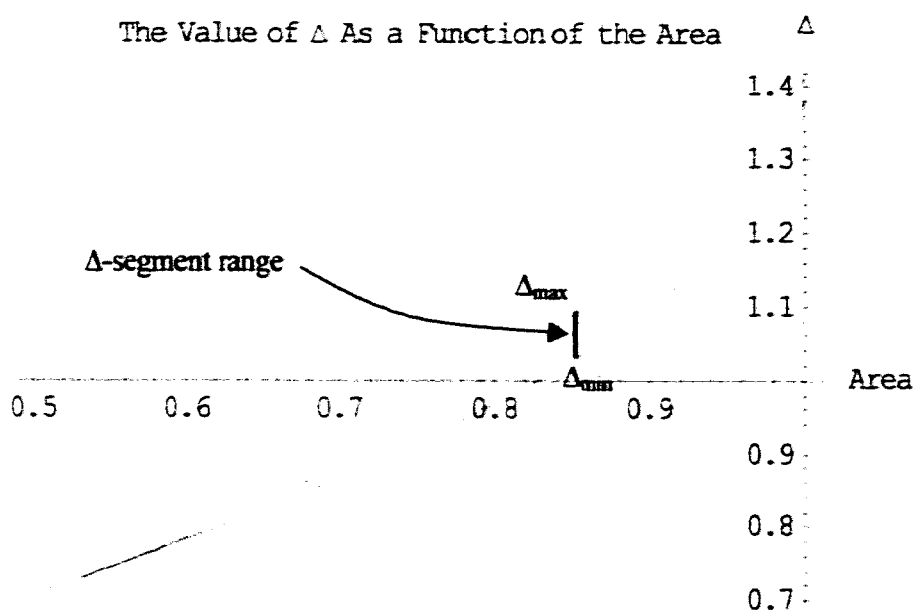


Figure 5.10. The values of $\Delta_{\min}$ and $\Delta_{\max}$ as the area, A , changes. The linear boundary intersecting a pixel with gray level 0.85 must intersect the median at a Δ value between $\Delta_{\min}$ and $\Delta_{\max}$, the Δ -segment range shown. Note that this plot is the rotated plot of the concave configuration from figure 5.8.

The values for $\Delta_{\min}$ and $\Delta_{\max}$ are summarized in the following table:

Pixel Configuration	$\Delta_{\min}$	$\Delta_{\max}$
Linear	$\frac{A}{2}$	$\frac{A}{2}$
Convex	$\frac{2\sqrt{2A}}{1+2A}$	$\sqrt{A}$
Concave	$\sqrt{2} - \sqrt{1-A}$	$\sqrt{2} - \frac{2\sqrt{2}(1-A)}{1+2(1-A)}$

5.3 Objects Bounded By Circular Arcs

As with objects bounded with linear boundaries the gray scale value of pixels along a circular arc digital boundary places constraints on the region of the pixel that the preimage boundary can pass possibly through. This constraint is expressed as a line segment along the pixel median, the Δ -segment, that must be intersected by the preimage object boundary. The Δ values that bound the Δ -segment, $\Delta_{\min}$ and $\Delta_{\max}$, for each of the three possible pixel configurations (linear, convex and concave) is now presented.

5.3.1 Linear Pixel Configurations

To determine the values of $\Delta_{\min}$ and $\Delta_{\max}$ we find the 1st and 2nd derivatives of the Δ function, $\Delta = f(\alpha) = f(A, r, \alpha)$ where A is the area of intersection between the pixel and the object, r is the radius of the circle and α is the angle between the median and the line defined by the two points that are the circle's intersection with the pixel circle (shown in

figure 5.11). For conventional purposes α refers to the upper angle to the right of the median.

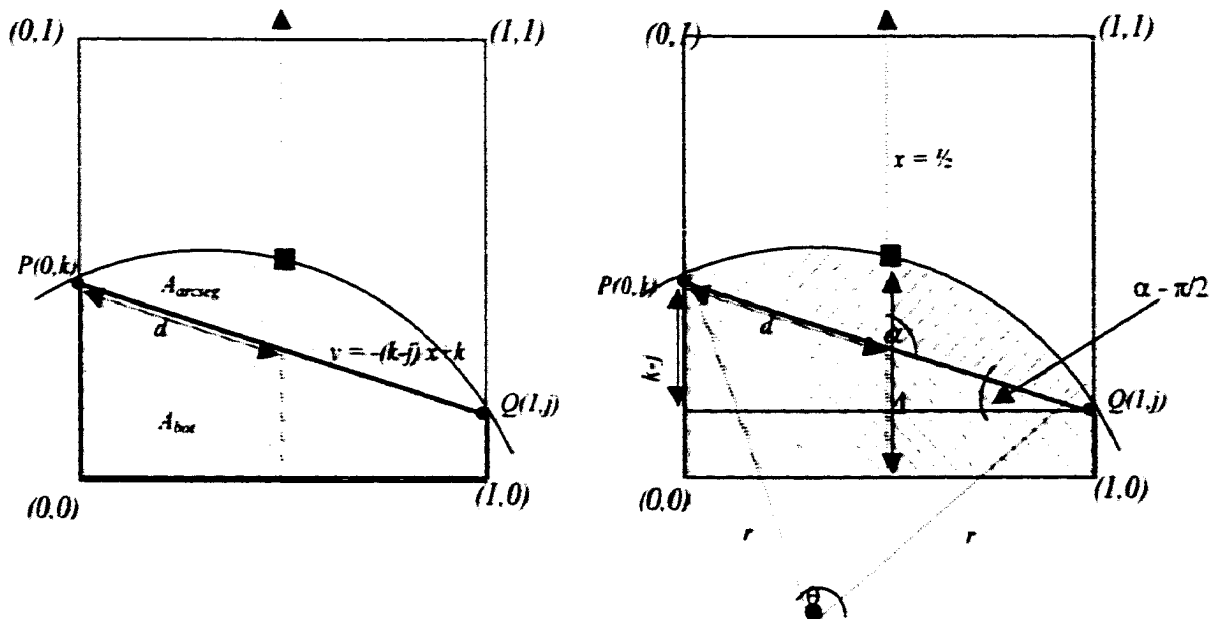


Figure 5.11. The intersection area of the pixel and circle includes two components: (a) the arc segment bound by points P and Q (A_{arcseg}) and the polygonal region below line segment $\overline{PQ}$ (A_{box}). The arc segment area is computed as a function of θ and r while the polygonal area is a function of α and k (b).

To solve for Δ in terms of A , r and α the following relationships are noted:

- 1) Δ = the distance along the median where the circle intersects the median.
- 2) $k = \tan(\alpha - \frac{\pi}{2}) + j$. j and k are the y-coordinates of points Q and P , respectively. α is the angle between the median and $\overline{PQ}$.

$$3) 2d = \sqrt{1 + (k-j)^2} . \quad d \text{ is half the length of } \overline{PQ} .$$

$$4) \sin(\theta/2) = d/r . \quad \theta \text{ is the size of the arc found inside the pixel (between } P \text{ and } Q).$$

$$5) A_{arcseg} = \frac{r^2(\theta - \sin(\theta))}{2} . \quad A_{arcseg} \text{ is the area of the arc segment joining } P \text{ and } Q$$

(i.e. the area between arc PQ and $\overline{PQ}$). This is the difference between the area of the θ sector and the area of the triangle formed by the sector radii the θ sector's chord.

$$6) A_{box} = \frac{k+j}{2} . \quad A_{box} \text{ is the area of the region of the pixel intersecting the circle not found in the arc segment.}$$

$$7) A = A_{arcseg} + A_{box} . \quad A \text{ is the total intersection area of the pixel and circle.}$$

Solving for j , d and θ in terms of α , k and r we have:

$$8) j = k + \cot(\alpha) \text{ (from 2)}$$

$$9) d = \frac{\csc(\alpha)}{2} \text{ (from 2 and 3)}$$

$$10) \quad \theta = 2 \text{ArcSin}\left(\frac{\csc(\alpha)}{2r}\right) \text{ (from 4 and 9)}$$

Solving for A , A_{arcseg} and A_{box} in terms of α , k and r we have:

$$11) \quad A_{arcseg} = -\frac{1}{2} r^2 (-2 \text{ArcCsc}[2r \sin[\alpha]] + \sin[2 \text{ArcCsc}[2r \sin[\alpha]]]) \text{ (from 5 and 10)}$$

$$12) \quad A_{box} = k + \frac{\cot[\alpha]}{2} \text{ (from 6 and 8)}$$

$$13) \quad A = k + r^2 \operatorname{ArcSin}\left[\frac{\operatorname{Csc}[\alpha]}{2r}\right] + \frac{\operatorname{Cot}[\alpha]}{2} - \frac{1}{2} r^2 \operatorname{Sin}\left[2 \operatorname{ArcSin}\left[\frac{\operatorname{Csc}[\alpha]}{2r}\right]\right] \quad (\text{from 11 and 12})$$

Solving for k in terms of A , r and α we have:

$$14) \quad k = \frac{1}{2} (\sin(2 \operatorname{csc}^{-1}(2r \sin(\alpha))) - 2 \operatorname{csc}^{-1}(2r \sin(\alpha))) r^2 + A - \frac{\operatorname{cot}(\alpha)}{2}$$

(from 13)

j may be expressed in terms of A , r and α as well (eq. 9):

$$15) \quad j = \frac{1}{2} ((\sin(2 \operatorname{csc}^{-1}(2r \sin(\alpha))) - 2 \operatorname{csc}^{-1}(2r \sin(\alpha))) r^2 + 2A + \operatorname{cot}(\alpha)) \quad (\text{from 8 and 13})$$

Solving for the center (c_x, c_y) of the circle that intersects $P(0,k)$ and $Q(1,j)$ expressed by

$$16) \quad \frac{(0 - c_x)^2}{r^2} + \frac{((k) - c_y)^2}{r^2} = 1, \quad \frac{(1 - c_x)^2}{r^2} + \frac{((j) - c_y)^2}{r^2} = 1 \quad \text{we have:}$$

$$17) \quad c_x = \frac{1}{2} \left(1 + \operatorname{Cos}[\alpha] \sqrt{-1 + 4r^2 - \operatorname{Cot}[\alpha]^2} \right),$$

$$c_y = A - \frac{1}{2} \sqrt{-1 + 4r^2 - \operatorname{Cot}[\alpha]^2} \operatorname{Sin}[\alpha] + \frac{1}{2} r^2 (-2 \operatorname{ArcCsc}[2r \operatorname{Sin}[\alpha]] + \operatorname{Sin}[2 \operatorname{ArcCsc}[2r \operatorname{Sin}[\alpha]]])$$

Solving for Δ which is the y-coordinate of the intersection of circle centered at (c_x, c_y)

with the median ($x = 1/2$) expressed by $x = \frac{1}{2}, \frac{(x - c_x)^2}{r^2} + \frac{(y - c_y)^2}{r^2} = 1$ we have:

$$18) \quad \Delta =$$

$$\frac{1}{4} \left(4A + \sqrt{2} \sqrt{1 + 3r^2 + (1 - 4r^2) \operatorname{Cos}[2\alpha] + r^2 \operatorname{Cos}[4\alpha]} \operatorname{Csc}[\alpha] - 2 \sqrt{-1 + 4r^2 - \operatorname{Cot}[\alpha]^2} \operatorname{Sin}[\alpha] + 2r^2 (-2 \operatorname{ArcCsc}[2r \operatorname{Sin}[\alpha]] + \operatorname{Sin}[2 \operatorname{ArcCsc}[2r \operatorname{Sin}[\alpha]]]) \right)$$

The following is the Mathematica code which expresses the solutions to the equations described above:

$$\text{Simplify}\left[\text{Solve}\left[\left\{k = \text{Tan}\left[\alpha - \frac{\pi}{2}\right] + j, 2d = \sqrt{1 - (k - j)^2}, \text{Sin}\left[\frac{\theta}{2}\right] = \frac{d}{z}\right\}, (j, d, \theta)\right], d > 0 \ \&\& \ z > 0 \ \&\& \ k \geq j \ \&\& \ j \geq 0 \ \&\& \ \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4}\right]$$

$$\left\{\left\{d \rightarrow -\frac{\text{Csc}[\alpha]}{z}, j \rightarrow k + \text{Csc}[\alpha], \theta \rightarrow -2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{z z}\right]\right\}, \left\{d \rightarrow \frac{\text{Csc}[\alpha]}{z}, j \rightarrow k + \text{Csc}[\alpha], \theta \rightarrow 2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{z z}\right]\right\}\right\}$$

The second solution is valid since $d > 0$

$$\text{Simplify}\left[\text{Solve}\left[\left\{\text{Sin}\left[\frac{\theta}{2}\right] = \frac{d}{z}, d = \frac{\text{Csc}[\alpha]}{2}, (\theta, d)\right\}, d > 0 \ \&\& \ z > 0 \ \&\& \ k \geq j \ \&\& \ j \geq 0 \ \&\& \ \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4}\right]$$

$$\left\{\left\{d \rightarrow \frac{\text{Csc}[\alpha]}{z}, \theta \rightarrow 2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{z z}\right]\right\}\right\}$$

$$\text{FullSimplify}\left[\text{Solve}\left[\left\{A_{\text{resum}} = \frac{z^2}{2} ((\theta) - \text{Sin}[\theta]), \theta = 2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{2 z}\right]\right\}, (A_{\text{resum}}, \theta), (j)\right], d > 0 \ \&\& \ z > 0 \ \&\& \ k \geq j \ \&\& \ j \geq 0 \ \&\& \ \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4}\right]$$

$$\left\{\left\{A_{\text{resum}} \rightarrow -\frac{1}{2} z^2 (-2 \text{ArcCsc}[2 z \text{Sin}[\alpha]] + \text{Sin}[2 \text{ArcCsc}[2 z \text{Sin}[\alpha]]]), \theta \rightarrow 2 \text{ArcCsc}[2 z \text{Sin}[\alpha]]\right\}\right\}$$

$$A_{\text{resum}} = -\frac{1}{2} z^2 (-2 \text{ArcCsc}[2 z \text{Sin}[\alpha]] + \text{Sin}[2 \text{ArcCsc}[2 z \text{Sin}[\alpha]]])$$

$$\text{Simplify}\left[\text{Solve}\left[\left\{j = k + \text{Cot}[\alpha], A_{\text{res}} = \frac{k + j}{2}\right\}, (A_{\text{res}}, j)\right], \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4}\right]$$

$$\left\{\left\{A_{\text{res}} \rightarrow k + \frac{\text{Cot}[\alpha]}{z}\right\}\right\}$$

$$A_{\text{res}} = k + \frac{\text{Cot}[\alpha]}{z}$$

$$\text{Simplify}\left[\text{Solve}\left[\left\{A_{\text{resum}} = \frac{z^2}{2} \left(\left(2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{2 z}\right]\right) - \text{Sin}\left[2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{2 z}\right]\right)\right\}, A_{\text{res}} = k + \frac{\text{Cot}[\alpha]}{2}, A = A_{\text{res}} + A_{\text{resum}}\right\}, (j, A)\right], \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4}\right]$$

$$\text{FullSimplify}\left[\text{Solve}\left[A = k + r^2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{2r}\right] + \frac{\text{Cot}[\alpha]}{2} - \frac{1}{2} r^2 \text{Sin}\left[2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{2r}\right]\right], k\right], r > 0 \text{ \& } k \geq 0 \text{ \& } A \geq 0 \text{ \& } \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4}\right]$$

$$\text{FullSimplify}\left[\text{Solve}\left[\left\{A = k + r^2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{2r}\right] + \frac{\text{Cot}[\alpha]}{2} - \frac{1}{2} r^2 \text{Sin}\left[2 \text{ArcSin}\left[\frac{\text{Csc}[\alpha]}{2r}\right]\right], j = k + \text{Cot}[\alpha]\right\}, j, k\right], r > 0 \text{ \& } j \geq 0 \text{ \& } \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4}\right]$$

$$\left\{\left\{j \rightarrow \frac{1}{2} \left\{2A - 2r^2 \text{ArcCsc}[2r \text{Sin}[\alpha]] + \text{Cot}[\alpha] + r \text{Csc}[\alpha] \sqrt{1 - \frac{\text{Csc}[\alpha]^2}{4r^2}}\right\}\right\}\right\}$$

$$k = A - \frac{\text{Cot}[\alpha]}{2} + \frac{1}{2} r^2 (-2 \text{ArcCsc}[2r \text{Sin}[\alpha]] + \text{Sin}[2 \text{ArcCsc}[2r \text{Sin}[\alpha]]])$$

$$j = \frac{1}{2} (2A + \text{Cot}[\alpha] + r^2 (-2 \text{ArcCsc}[2r \text{Sin}[\alpha]] + \text{Sin}[2 \text{ArcCsc}[2r \text{Sin}[\alpha]]]))$$

j and k are now in terms of A , r and α . We now solve for the circle center (cx, cy) in terms of A , r and α . Then we solve for the intersection of that circle and the median located at $x = 1/2$.

$$\text{FullSimplify}\left[\text{Solve}\left[\left\{\frac{(0 - cx)^2}{r^2} + \frac{((k) - cy)^2}{r^2} = 1, \frac{(1 - cx)^2}{r^2} + \frac{((j) - cy)^2}{r^2} = 1\right\}, (cx, cy)\right], r > 0 \text{ \& } A \geq 0 \text{ \& } \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4}\right]$$

$$\left\{\left\{cx \rightarrow \frac{1}{2} \left(1 - \text{Csc}[\alpha] \sqrt{-1 + 4r^2 - \text{Csc}[\alpha]^2}\right), cy \rightarrow \frac{1}{2} \left(2A + \sqrt{-1 + 4r^2 - \text{Csc}[\alpha]^2} \text{Sin}[\alpha] + r^2 (-2 \text{ArcCsc}[2r \text{Sin}[\alpha]] + \text{Sin}[2 \text{ArcCsc}[2r \text{Sin}[\alpha]]])\right)\right\}, \left\{cx \rightarrow \frac{1}{2} \left(1 + \text{Csc}[\alpha] \sqrt{-1 + 4r^2 - \text{Csc}[\alpha]^2}\right), cy \rightarrow 2A - \frac{1}{2} \sqrt{-1 + 4r^2 - \text{Csc}[\alpha]^2} \text{Sin}[\alpha] + \frac{1}{2} r^2 (-2 \text{ArcCsc}[2r \text{Sin}[\alpha]] + \text{Sin}[2 \text{ArcCsc}[2r \text{Sin}[\alpha]]])\right\}\right\}$$

We select the second solution as in that solution the value of c_y is clearly less than that of the first. (An positive expression is subtracted in the 2nd and not added, as is the case in the 1st sltn.)

2nd solution has negative c_y — correct. choose general

form :

$$\left\{ \begin{aligned} c_x &\rightarrow \frac{1}{2} \left(1 + \cos[\alpha] \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \right), \\ c_y &\rightarrow A - \frac{1}{2} \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \sin[\alpha] - \\ &\quad \frac{1}{2} x^2 (-2 \operatorname{ArcCsc}[2x \sin[\alpha]] + \sin[2 \operatorname{ArcCsc}[2x \sin[\alpha]]]) \end{aligned} \right\}$$

$$\begin{aligned} c_y &= A - \frac{1}{2} \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \sin[\alpha] - \\ &\quad \frac{1}{2} x^2 (-2 \operatorname{ArcCsc}[2x \sin[\alpha]] + \sin[2 \operatorname{ArcCsc}[2x \sin[\alpha]]]) \end{aligned}$$

$$\begin{aligned} A &= \frac{1}{2} \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \sin[\alpha] + \\ &\quad \frac{1}{2} x^2 (-2 \operatorname{ArcCsc}[2x \sin[\alpha]] + \sin[2 \operatorname{ArcCsc}[2x \sin[\alpha]]]) \end{aligned}$$

$$c_x = \frac{1}{2} \left(1 + \cos[\alpha] \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \right)$$

$$\frac{1}{2} \left(1 + \cos[\alpha] \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \right)$$

Solve for the circle and median intersection

$$\text{FullSimplify}\left[\text{Solve}\left[\left\{x = 1/2, (x - c_x)^2 + (y - c_y)^2 = x^2, (x, y)\right\}, \right.\right. \\ \left.\left. x > 0 \ \&\& \ A \geq 0 \ \&\& \ \frac{\pi}{2} \leq \alpha \leq \frac{3\pi}{4} \right]\right]$$

$$\left\{ \left\{ x \rightarrow \frac{1}{2}, \right. \right. \\ \left. y \rightarrow \frac{1}{4} \left(4A - \sqrt{2} \sqrt{1 + 3x^2 + (1 - 4x^2) \cos[2\alpha]} + x^2 \cos[4\alpha] \cos[\alpha] - \right. \right. \\ \left. 2 \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \sin[\alpha] + \right. \\ \left. 2x^2 (-2 \operatorname{ArcCsc}[2x \sin[\alpha]] + \sin[2 \operatorname{ArcCsc}[2x \sin[\alpha]]]) \right\}, \\ \left\{ x \rightarrow \frac{1}{2}, y \rightarrow \frac{1}{4} \left(4A + \sqrt{2} \sqrt{1 + 3x^2 + (1 - 4x^2) \cos[2\alpha]} + x^2 \cos[4\alpha] \cos[\alpha] \right. \right. \\ \left. \left. \cos[\alpha] - 2 \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \sin[\alpha] + \right. \right. \\ \left. \left. 2x^2 (-2 \operatorname{ArcCsc}[2x \sin[\alpha]] + \sin[2 \operatorname{ArcCsc}[2x \sin[\alpha]]]) \right\} \right\}$$

Determine correct delta value

2nd solution has y value greater than A and is thus correct

$$\begin{aligned} c_y &= \\ &\frac{1}{4} \left(4A + \sqrt{2} \sqrt{1 + 3x^2 + (1 - 4x^2) \cos[2\alpha]} + x^2 \cos[4\alpha] \cos[\alpha] - \right. \\ &\quad \left. 2 \sqrt{-1 + 4x^2 - \cos[\alpha]^2} \sin[\alpha] - \right. \\ &\quad \left. 2x^2 (-2 \operatorname{ArcCsc}[2x \sin[\alpha]] + \sin[2 \operatorname{ArcCsc}[2x \sin[\alpha]]]) \right) \end{aligned}$$

$$\text{Simplify}[\text{DF}_A = D[\mathcal{L}_A, \alpha]]$$

$$\frac{1}{4} \left[\frac{2 \sqrt{2} \cos[\alpha] (1 - 4x^2 + 4x^2 \cos[2\alpha])}{\sqrt{1 + 2x^2 + (1 - 4x^2) \cos[2\alpha] + x^2 \cos[4\alpha]}} - \frac{2 \cos[\alpha] \sqrt{-1 + 4x^2 - \cos[\alpha]^2}}{\sqrt{2} \sqrt{1 + 2x^2 + (1 - 4x^2) \cos[2\alpha] + x^2 \cos[4\alpha]}} \cos[\alpha] \csc[\alpha] - \frac{2 \cos[\alpha] \csc[\alpha]}{\sqrt{-1 + 4x^2 - \cos[\alpha]^2}} + \frac{2 \cos[\alpha] \csc[\alpha]^2}{x \sqrt{4 - \frac{\cos[\alpha]^2}{x^2}}} \right]$$

Checking

$$\alpha = \pi/2$$

$$\frac{\pi}{2}$$

$$\text{DF}_A$$

$$0$$

$$\text{2nd derivative } \text{DDF}_A = \text{Simplify}[D[\text{DF}_A, \alpha]]$$

$$\frac{1}{4} \left[\frac{4 \cos[\alpha]^2 (\csc[\alpha]^2)^{3/2}}{x \left(4 - \frac{\cos[\alpha]^2}{x^2}\right)^{3/2}} - \frac{\cos[\alpha]^2 (\csc[\alpha]^2)^{3/2}}{x^2 \left(4 - \frac{\cos[\alpha]^2}{x^2}\right)^{3/2}} - \frac{4x \cos[\alpha]^2 \sqrt{\csc[\alpha]^2}}{\sqrt{4 - \frac{\cos[\alpha]^2}{x^2}}} - \frac{4x (\csc[\alpha]^2)^{3/2}}{\sqrt{4 - \frac{\cos[\alpha]^2}{x^2}}} - \frac{4 \cos[\alpha]^2 (\csc[\alpha]^2)^{3/2}}{x \sqrt{4 - \frac{\cos[\alpha]^2}{x^2}}} - \frac{(\csc[\alpha]^2)^{3/2}}{x \sqrt{4 - \frac{\cos[\alpha]^2}{x^2}}} + x \cos[\alpha]^2 \sqrt{\csc[\alpha]^2} \sqrt{4 - \frac{\csc[\alpha]^2}{x^2}} + x (\csc[\alpha]^2)^{3/2} \sqrt{4 - \frac{\csc[\alpha]^2}{x^2}} + \frac{4(1 - x^2 + x^2 \cos[2\alpha])^2 \cos[\alpha]^2 \csc[\alpha]^2}{(4x^2 \csc[\alpha]^2 - \csc[\alpha]^4)^{3/2}} + \frac{4 \cos[\alpha]^2 \csc[\alpha]^2}{\sqrt{4x^2 \csc[\alpha]^2 - \csc[\alpha]^4}} + \frac{4 \csc[\alpha]^2 (-2x^2 + \csc[\alpha]^2)}{\sqrt{4x^2 \csc[\alpha]^2 - \csc[\alpha]^4}} - 4 \cos[2\alpha] \sqrt{4x^2 \csc[\alpha]^2 - \csc[\alpha]^4} + \frac{2(4x^2 \cos[2\alpha] + (2 + \cos[2\alpha]) \csc[\alpha]^2)}{\sqrt{4x^2 - 4x^2 \cos[\alpha]^2 + \cos[\alpha]^2}} - \frac{2(\cos[\alpha] \csc[\alpha]^2 - 4x^2 \cos[\alpha] \sin[\alpha])^2}{(4x^2 - 4x^2 \cos[\alpha]^2 + \cos[\alpha]^2)^{3/2}} - \frac{4 \cos[\alpha] \csc[\alpha]^2 (-2x^2 + \csc[\alpha]^2) \sin[2\alpha]}{\sqrt{4x^2 \csc[\alpha]^2 - \csc[\alpha]^4}} \right]$$

We refer the value of Δ as a function of A , r and α as the Δ function f_Δ .

The first and second derivatives of f_Δ with respect to α are:

$$\frac{1}{4} \left[\frac{4r \cot(\alpha) \sqrt{\csc(\alpha)^2}}{\sqrt{4 - \frac{\csc(\alpha)^2}{r^2}}} + \frac{\cot(\alpha) (\csc(\alpha)^2)^{3/2}}{r \sqrt{4 - \frac{\csc(\alpha)^2}{r^2}}} - r \cot(\alpha) \sqrt{\csc(\alpha)^2} \sqrt{4 - \frac{\csc(\alpha)^2}{r^2}} + \frac{4 \cot(\alpha) (2r^2 - \csc(\alpha)^2)}{\sqrt{4r^2 \csc(\alpha)^2 - \csc(\alpha)^4}} - \frac{2 \sqrt{4r^2 \csc(\alpha)^2 - \csc(\alpha)^4} \sin(2\alpha) + \frac{-2 \cot(\alpha) \csc(\alpha)^2 + 4r^2 \sin(2\alpha)}{\sqrt{4r^2 - 4r^2 \csc(\alpha)^2 + \cot(\alpha)^2}} \right]$$

and

$$\frac{1}{4} \left[\frac{4 \cot(\alpha)^2 (\csc(\alpha)^2)^{3/2}}{r (4 - \frac{\csc(\alpha)^2}{r^2})^{3/2}} - \frac{\cot(\alpha)^2 (\csc(\alpha)^2)^{5/2}}{r^2 (4 - \frac{\csc(\alpha)^2}{r^2})^{3/2}} - \frac{4r \cot(\alpha)^2 \sqrt{\csc(\alpha)^2}}{\sqrt{4 - \frac{\csc(\alpha)^2}{r^2}}} - \frac{4r (\csc(\alpha)^2)^{3/2}}{\sqrt{4 - \frac{\csc(\alpha)^2}{r^2}}} - \frac{4 \cot(\alpha)^2 (\csc(\alpha)^2)^{3/2}}{r \sqrt{4 - \frac{\csc(\alpha)^2}{r^2}}} - \frac{(\csc(\alpha)^2)^{5/2}}{r \sqrt{4 - \frac{\csc(\alpha)^2}{r^2}}} + r \cot(\alpha)^2 \sqrt{\csc(\alpha)^2} \sqrt{4 - \frac{\csc(\alpha)^2}{r^2}} + r (\csc(\alpha)^2)^{3/2} \sqrt{4 - \frac{\csc(\alpha)^2}{r^2}} + \frac{3(1 - r^2 + r^2 \cos(2\alpha))^2 \cot(\alpha)^2 \csc(\alpha)^4}{16r^2 \csc(\alpha)^2 - \csc(\alpha)^4)^{3/2}} + \frac{3 \cot(\alpha)^2 \csc(\alpha)^2}{\sqrt{4r^2 \csc(\alpha)^2 - \csc(\alpha)^4}} + \frac{4 \csc(\alpha)^2 (1 - 2r^2 + \csc(\alpha)^2)}{\sqrt{4r^2 \csc(\alpha)^2 - \csc(\alpha)^4}} - 4 \cos(2\alpha) \sqrt{4r^2 \csc(\alpha)^2 - \csc(\alpha)^4} + \frac{2(4r^2 \cos(2\alpha) + (2 + \cos(2\alpha)) \csc(\alpha)^2)}{\sqrt{4r^2 - 4r^2 \csc(\alpha)^2 + \cot(\alpha)^2}} - \frac{2(\cot(\alpha) \csc(\alpha)^2 - 4r^2 \cot(\alpha) \sin(2\alpha))^2}{(4r^2 - 4r^2 \csc(\alpha)^2 + \cot(\alpha)^2)^{3/2}} - \frac{4 \cot(\alpha) \csc(\alpha)^2 (1 - 2r^2 + \csc(\alpha)^2) \sin(2\alpha)}{\sqrt{4r^2 \csc(\alpha)^2 - \csc(\alpha)^4}} \right]$$

, respectively.

The first derivative $\frac{df_\Delta}{d\alpha}$ equals 0 when $\alpha = \frac{\pi}{2}$, denoting a stationary point. See figure

5.12 where a plot of f_Δ is shown where $1 \leq r \leq 1000$ and $\frac{\pi}{4} \leq \alpha \leq \frac{3\pi}{4}$). This α value

represents a minimum, since the second derivative at this point is $\frac{1}{\frac{3r^2}{\sqrt{4r^2-1}} + 4r}$ which is

positive. Thus, for a linear pixel configuration $\Delta_{\min}$ is f_Δ evaluated at $\frac{\pi}{2}$ which equals

$A + \frac{1}{4} \sqrt{4 - \frac{1}{r^2}} r + \sqrt{r^2} - \frac{1}{2} \sqrt{-1 + 4r^2} - r^2 \text{ArcCsc}[2r]$. Since $r > 0$ we may simplify

the result so that $\Delta_{\min} = A + r - \frac{1}{4} \sqrt{4r^2 - 1} - \csc^{-1}(2r)r^2$.

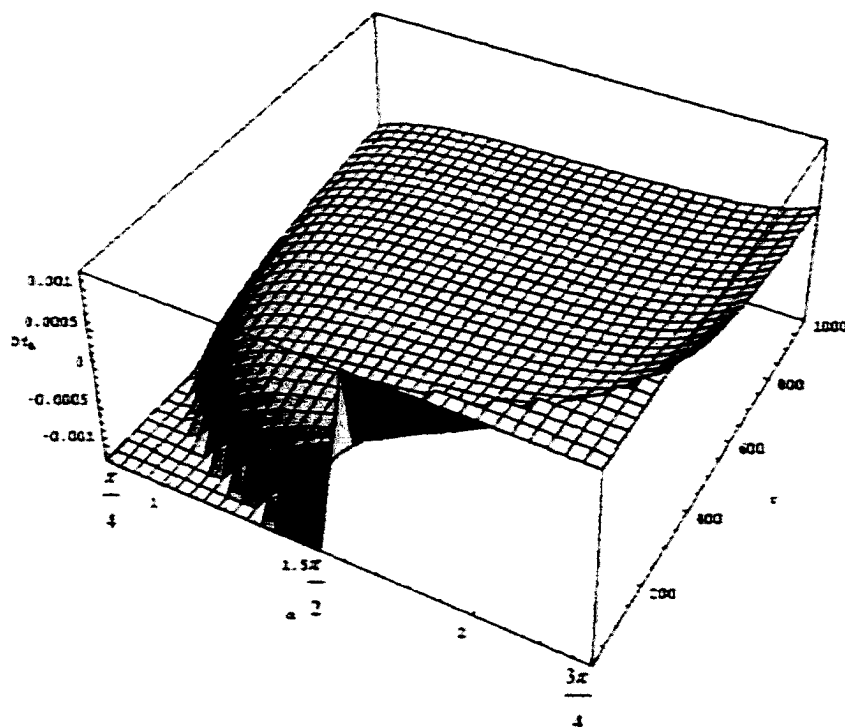


Figure 5.12. When $\alpha = \frac{\pi}{2}$ the first derivative of Δ with respect to α is 0

for all radii. The second derivative is positive indicating that when $\alpha = \frac{\pi}{2}$,

Δ is at a minimum.

Clearly, as α increases or decreases from $\pi/2$ the value of Δ will increase. The maximum will be achieved at the value of α furthest from $\pi/2$. By inspection it is clear that this occurs when the circle intersects a corner of the pixel and the pixel configuration ceases being linear. This occurs when either $j = 0$ or $k = 1$, as shown in figure 5.13b,c. The

values of Δ in both these cases are identical by symmetry. We may solve for A where $j =$

0 resulting in $A = \frac{1}{2} \left(\left(\sin \left(2 \csc^{-1} \left(\frac{2r}{\sqrt{k^2+1}} \right) \right) - 2 \csc^{-1} \left(\frac{2r}{\sqrt{k^2+1}} \right) \right) r^2 + k \right)$ (from 13

and substitution in previous equations). Solving for the value of $\Delta_{\max}$ for this case is very difficult because there does not appear to be a closed form solution to express k in terms of A and r alone in equation. Thus, $\Delta_{\max}$ cannot be expressed in terms of A and r . Instead, we present an upper bound on $\Delta_{\max}$ in terms of A and r . A few definitions are first necessary.

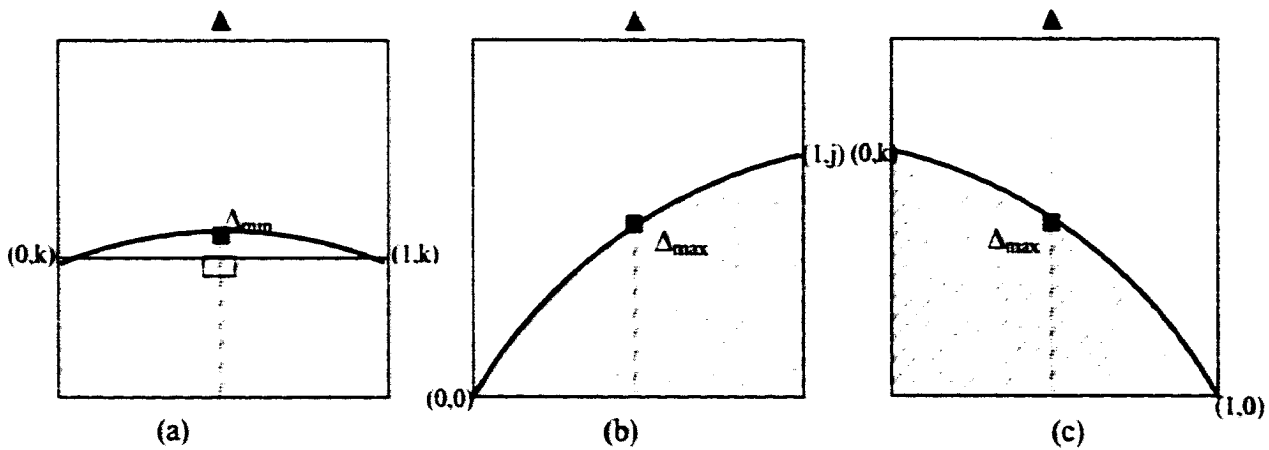


Figure 5.13. Although a closed form equation exists for computing $\Delta_{\max}$,

where $\alpha = \pi / 2$, it does not appear that a closed form solution for $\Delta_{\max}$ (b and c) exists.

Definition: The *bulge height* of a circular arc bound by points P and Q is the length of the maximal line segment perpendicular to $\overline{PQ}$ with one endpoint along the arc and the other along $\overline{PQ}$. [Vossler, 2000]

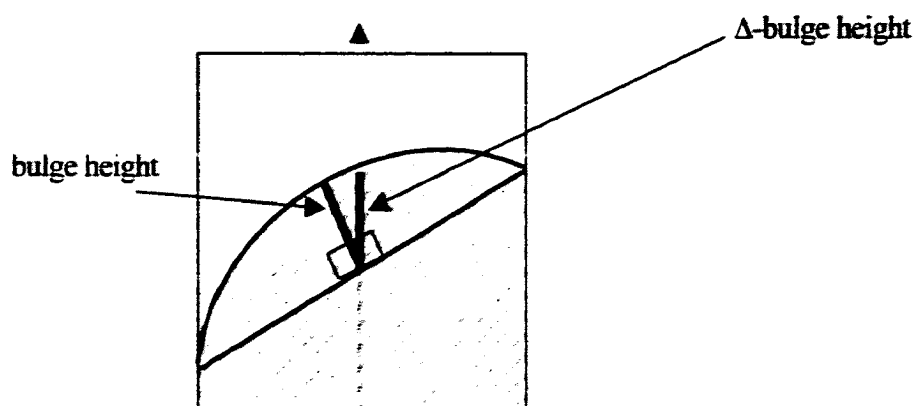


Figure 5.14. We may place an upper bound on the value of $\Delta_{\max}$ by considering the maximum Δ -bulge height of the arc defined by P and Q .

Definition: The Δ -*bulge height* of a circular arc bound by points P and Q that are on a pixel boundary is the distance between the intersection of the line segment $\overline{PQ}$ with the pixel's median and the intersection of arc PQ with the median.

$\Delta_{\max}$ cannot be less than the value of Δ where $r = \infty$ since Δ decreases as r increases. This value, denoted $\Delta_{\max}^{\infty}$ is A since $j = 2A$ and $\Delta = j / 2$ as shown in figure 5.15a. (For figure 5.15b $k = 2A - 1$ and $\Delta = (1+k) / 2$. At the other extreme, $\Delta_{\max}$ cannot exceed the sum of $\Delta_{\max}^{\infty}$ and the maximum Δ -bulge, $\Delta\text{-bulge}_{\max}$. As shown earlier, $\Delta\text{-bulge}_{\max}$ is achieved when α is furthest from $\pi / 2$. This occurs in one of two cases depending on the value of A :

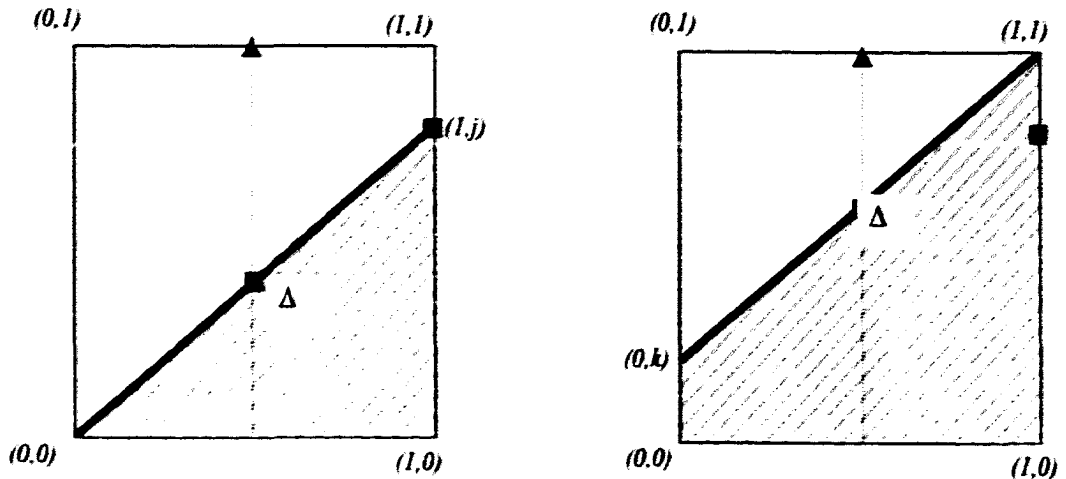


Figure 5.15. The lower bound on $\Delta_{\max}$ is the value of $\Delta_{\max}$ when $r = \infty$.

In this case $\Delta = A$.

Case A: when $0 \leq A < 0.5$ the arc segment endpoints are at $(0, 0)$ and $(1, j)$ (see figure 5.16). Solving for the circle center (c_x, c_y) that intersect points $(0,0)$ and $(0, j = 2A)$ we

have $c_x = \frac{1}{2} + \frac{A\sqrt{-(1+4A^2)(1+4A^2-4r^2)}}{1+4A^2}$ and $c_y = A - \frac{\sqrt{-(1+4A^2)(1+4A^2-4r^2)}}{2+8A^2}$ (equation

16). Solving for Δ (equation 18) we have

$$\Delta = \frac{2A + 8A^3 - \sqrt{-(1+4A^2)(1+4A^2-4r^2)} + 2\sqrt{(1+4A^2)(A^2+4A^4+r^2)}}{2+8A^2}.$$

Case B: when $0.5 \leq A < 1$ the arc segment endpoints are at $(1,1)$ and $(0, k)$ (see figure 5.17). Solving for the circle center (c_x, c_y) that intersect points $(0, k = 2A-1)$ and $(1,1)$ we

have $c_x = \frac{1}{2} - \frac{(-1+A)\sqrt{-(5+4(-2+A)A)(5+4(-2+A)A-4r^2)}}{5+4(-2+A)A}$ and $c_y =$

$A - \frac{\sqrt{-(5+4(-2+A)A)(5+4(-2+A)A-4r^2)}}{2(5+4(-2+A)A)}$ (equation 16). Solving for Δ (equation 18) we

have

$$\Delta = \frac{1}{2(5+4(-2+A)A)} \left(2A(5+4(-2+A)A) - \sqrt{(-5-4(-2+A)A)(5+4(-2+A)A-4r^2)} + 2\sqrt{(5+4(-2+A)A)((-1+A)^2(5+4(-2+A)A)+r^2)} \right)$$

These intersection points are the upper bounds for the Δ -points and bound $\Delta_{\max}$ for their respective cases.

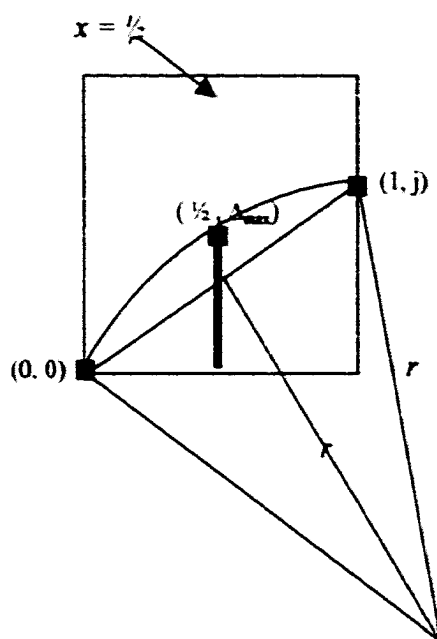


Figure 5.16. An upper bound on the Δ -point when A is less than 0.5 is at the shown arc segment's intersection with the median

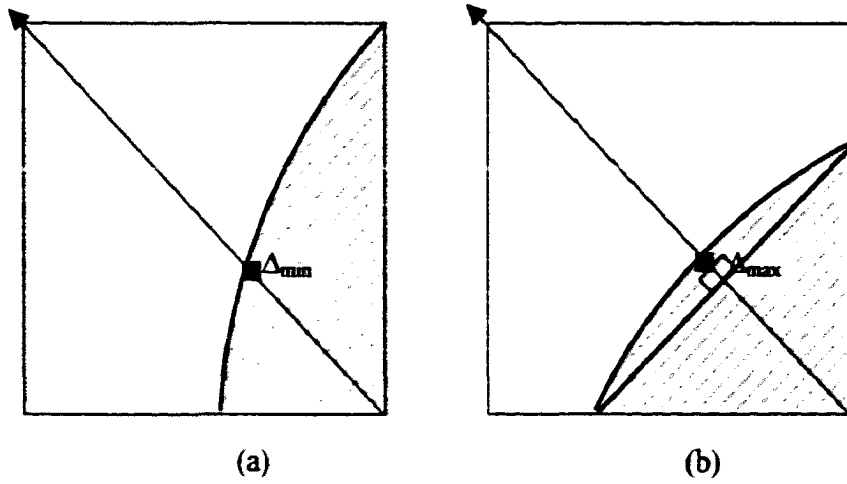


Figure 5.18. The values of Δ_{min} and Δ_{max} of a convex pixel configuration is where the curve intersects a corner of the pixel (a) and the curve is at right angles with the median (i.e. $\alpha = \pi / 2$), respectively.

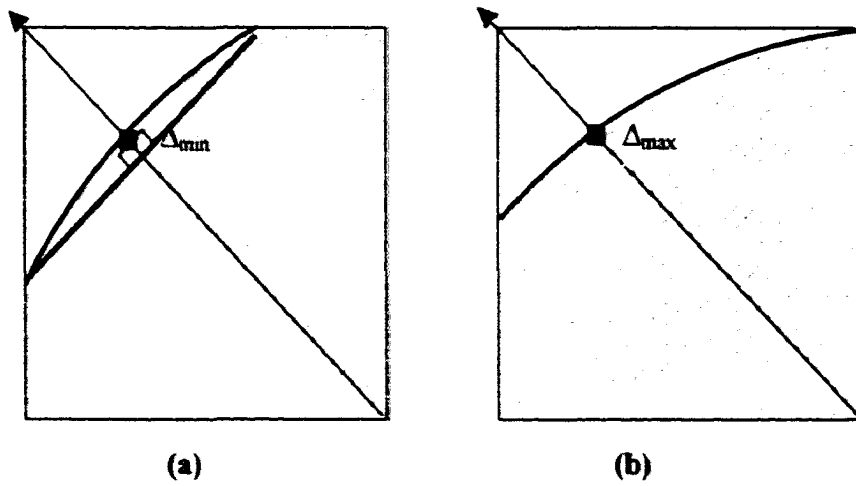


Figure 5.19. The values of Δ_{min} and Δ_{max} of a concave pixel configuration is (a) where the curve is at right angles with the median and (b) the curve intersects a corner of the pixel, respectively.

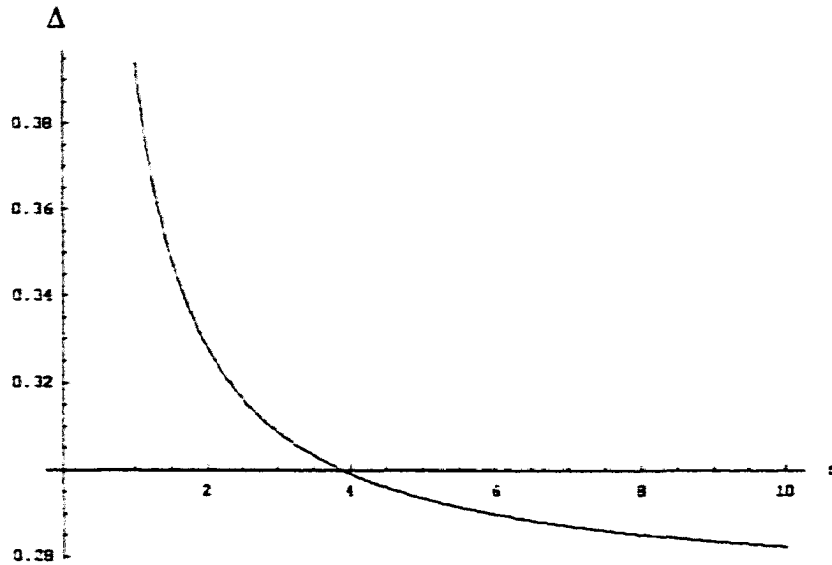


Figure 5.20. The value of Δ decreases with increased radius r . Therefore $\Delta_{\min}$ for any radius value cannot be less than $\Delta_{\min}$ where $r = \infty$. The lower bound on $\Delta_{\max}$ is the value of $\Delta_{\max}$ when $r = \infty$.

As shown in figure 5.20, r is inversely related to Δ such that as r increases Δ decreases. Thus $\Delta_{\min}$ for convex and concave pixel configurations have the lower bound of Δ where $r = \infty$, denoted $\Delta_{\min}^*$. This is the same Δ value as when the curve represents a digital line (section 5.2.2-3). For a convex configuration $\Delta_{\min}$ has an upper bound of $\frac{2\sqrt{2}A}{1+2A}$. For a concave pixel configuration $\Delta_{\min}$ has an upper bound of $\sqrt{2} - \sqrt{1-A}$.

$\Delta_{\max}$ is bound above by $\Delta_{\max}^* + \Delta\text{-bulge}_{\max}$ which is determined as follows. For the convex case we follow the following steps (figure 5.21):

- 1) Solve for the center of the circle (c_x, c_y) that intersects the pixel at $(0, k)$

and $(k, 0)$ with equation $\frac{(0-c_x)^2}{r^2} + \frac{(k-c_y)^2}{r^2} = 1$, $\frac{(k-c_x)^2}{r^2} + \frac{(0-c_y)^2}{r^2} = 1$. The

maximum value for $k = \sqrt{2A}$ and, thus, k is substituted with $\sqrt{2A}$. The solution

$$\text{is } c_x = A - \frac{\sqrt{r^2 - 2A^2}}{\sqrt{2}} \text{ and } c_y = A - \frac{\sqrt{r^2 - 2A^2}}{\sqrt{2}}.$$

2) The intersection of this circle and the median (with equation $x = y$) is at

$$(\Delta_x, \Delta_y) \text{ where } \Delta_x = \Delta_y, \Delta_x \text{ evaluates to } A + \frac{r - \sqrt{-2A^2 + r^2}}{\sqrt{2}}.$$

3) Solving for $\Delta_{\max}$ where $\Delta_{\max} = \sqrt{(\Delta_x^2 + \Delta_y^2)}$ yields

$$\sqrt{2} \sqrt{-(\sqrt{2}A + r)(-r + \sqrt{-2A^2 + r^2})} \text{ as shown in figure 5.21.}$$

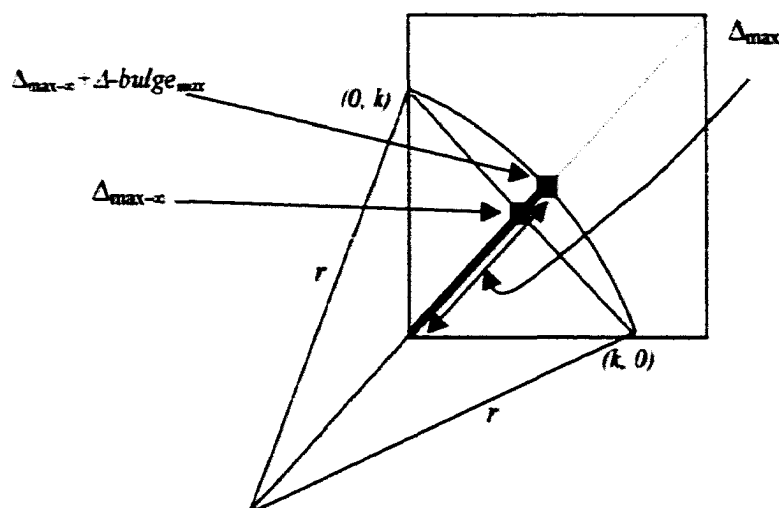


Figure 5.21. An upper limit may be placed on the Δ -point of a convex pixel configuration as the sum of $\Delta_{\max}^*$ and $\Delta\text{-bulge}_{\max}$. $\Delta_{\max}^*$ is the upper bound on Δ when $r = \infty$ and $\Delta\text{-bulge}_{\max}$ is the maximum bulge height of the arc along the median.

For the concave case we follow the following steps:

1) Solve for the center of the circle (c_x , c_y) that intersects the pixel at (0,1)

and (1, k) with equation $\frac{(0-c_x)^2}{r^2} + \frac{(1-c_y)^2}{r^2} = 1$, $\frac{(1-c_x)^2}{r^2} + \frac{(k-c_y)^2}{r^2} = 1$. The

maximum value for $k = 2A - 1$ and, thus, k is substituted with $2A - 1$. The

solution is $c_x = \frac{1}{2} + \frac{(-1+A)\sqrt{-(5+4(-2+A)A)(5+4(-2+A)A-4r^2)}}{5+4(-2+A)A}$,

$$c_y = A - \frac{\sqrt{-(5+4(-2+A)A)(5+4(-2+A)A-4r^2)}}{2(5+4(-2+A)A)}.$$

2) The intersection of this circle and the median (with equation $x = y$) is at

(Δ_x , Δ_y) where $\Delta_x = \Delta_y$. Δ_x evaluates

$$\text{to } \frac{1}{4(5+4(-2+A)A)} \left(5+2A-12A^2+8A^3+(-3+2A)\sqrt{(-5-4(-2+A)A)(5+4(-2+A)A-4r^2)} + \right. \\ \left. \sqrt{2} \sqrt{(5+4(-2+A)A)(2(3-2A)^2r^2+(1-2A)^2\sqrt{(-5-4(-2+A)A)(5+4(-2+A)A-4r^2)})} \right).$$

3) Solving for $\Delta_{\max}$ where $\Delta_{\max} = \sqrt{(\Delta_x^2 + \Delta_y^2)}$ yields

$$\frac{1}{2\sqrt{2}(5+4(-2+A)A)} \left(\sqrt{5+2A-12A^2+8A^3+(-3+2A)\sqrt{(-5-4(-2+A)A)(5+4(-2+A)A-4r^2)} +} \right. \\ \left. \sqrt{2} \sqrt{(5+4(-2+A)A)(2(3-2A)^2r^2+(1-2A)^2\sqrt{(-5-4(-2+A)A)(5+4(-2+A)A-4r^2)})} \right)^2 \quad \text{as}$$

shown in figure 5.22.

Pixel Configuration	Lowest Bound of $\Delta_{\min}$	Upper Bound of $\Delta_{\max}$
Linear	$A + r - \frac{1}{4} \sqrt{4r^2 - 1} - \csc^{-1}(2r)r^2$	$A < 1/2:$ $\frac{8A^3 + 2A + 2\sqrt{(4A^2 + 1)(4A^4 + A^2 + r^2)} - \sqrt{-(4A^2 + 1)(4A^2 - 4r^2 - 1)}}{8A^2 + 2}$ $A > 1/2:$ $\frac{1}{2(4(A-2)A+5)} (2A(4(A-2)A+5) +$ $2\sqrt{(4(A-2)A+5)(4(A-2)A+5)(A-1)^2 + r^2}) -$ $\sqrt{-(4(A-2)A+5)(-4r^2 + 4(A-2)A+5)})$
Convex	$\frac{2\sqrt{2A}}{1+2A}$	$\sqrt{2} \sqrt{-(\sqrt{2}A+r)(-r+\sqrt{-2A^2+r^2})}$
Concave	$\sqrt{2} - \sqrt{1-A}$	$\frac{1}{4(4(A-2)A+5)} \left(\sqrt{(8A^3 - 12A^2 + 2A + (2A-3))} \right.$ $\sqrt{(-4(A-2)A-5)(-4r^2 + 4(A-2)A+5)} +$ $\sqrt{2} \sqrt{(4(A-2)A+5)}$ $\left. \left(\sqrt{(-4(A-2)A-5)(-4r^2 + 4(A-2)A+5)} \right. \right.$ $\left. \left. (1-2A)^2 + 2(3-2A)^2 r^2 \right) + 5 \right)^{1/2}$

Figure 5.23. The upper and lower bounds for $\Delta_{\min}$ and $\Delta_{\max}$, respectively, for a circular digital curve. Although a closed form solution does not exist for $\Delta_{\min}$ and $\Delta_{\max}$ for all cases, lower and upper bounds may be placed on both $\Delta_{\min}$ and $\Delta_{\max}$.

5.4 Gray Scale Error

The values of Δ , $\Delta_{\min}$ and $\Delta_{\max}$ that were determined in sections 5.1-5.3 are valid only where A is precisely known. In gray level images, however, the exact gray level is

approximated by an integral value in an integral range. For example, with 256 gray levels a pixel must have a gray level, g , such that $0 \leq gray \leq 255$. When the scene is digitized a pixel whose pixel area intersecting the object has an area A , $gray$ will equal $A * 255$ rounded to the nearest integer. Therefore, even with a linear configuration of digital line segments, where $\Delta = A$, since A is not precisely known, an exact value for Δ cannot be determined. Rather, a range of Δ values exists, bounded by Δ_{min} and Δ_{max} , as with the concave and convex configurations. Similarly, the ranges of the Δ -segment for the concave and convex configurations presented above has assumed A is known precisely. Since A is not known accurately, the Δ -segment must be adjusted to reflect the entire range of possible values of A .

With n shades of gray and pixel gray level, $gray$, such that $0 \leq gray < n$ ($gray, n \in I$), A may be bound so that $\frac{gray - 0.5}{n} \leq A < \frac{gray + 0.5}{n}$. Clearly, the maximum possible error, $\epsilon = |A - gray / n|$ is $\frac{1}{2n}$. The values of Δ_{min} and Δ_{max} of an object bounded by line segments as a function of A where $A = gray / n$ for the linear, concave and convex configurations are expressed in table 5.24.

	Δ_{min}	Δ_{max}
Linear	$A - \epsilon$	$A + \epsilon$
Convex	$\frac{2\sqrt{2(A-\epsilon)}}{1+2(A-\epsilon)}$	$\sqrt{A+\epsilon}$
Concave	$\sqrt{2} - \sqrt{1-(A-\epsilon)}$	$\sqrt{2} - \frac{2\sqrt{2(1-(A+\epsilon))}}{1+2(1-(A+\epsilon))}$

Figure 5.24. The precise value of A , the pixel intersection area, for a gray scale pixel is not known. The accuracy of A is within error, ϵ . The

values of $\Delta_{\min}$ and $\Delta_{\max}$ for all three pixel configurations of a line segment boundary as a function of the area, A , and ϵ are shown.

The implementation for determining the passing area of a digital curve as constrained by the gray level of each individual pixel and that pixel's connectivity is now presented. First the initial passing area of the digital curve is computed as consisting of the upper and lower pixel boundaries. This expresses the simple binary constraint on the passing area. See figure 5.25 where a line segment is projected onto the grid that contains the digital curve that is the digitization of an object bounded by that line segment.

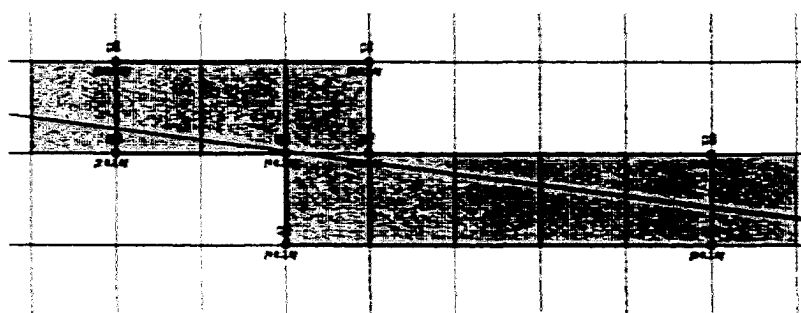


Figure 5.25. A line segment superimposed on the grid and the digital curve that is its digitization. The upper and lower pixel boundaries constrain the binary image passing area.

Next, the gray scale constraints area added as determined by $\Delta_{\min}$ and $\Delta_{\max}$ as set forth in figure 5.23. $\Delta_{\min}$ and $\Delta_{\max}$ are points along the pixel median that the preimage curve must pass between. Thus, in addition to the requirement that the preimage curve pass between the upper and lower boundaries to satisfy the binary constraint it must also intersect the median between the $\Delta_{\min}$ and $\Delta_{\max}$ points along the median of each pixel. See

figure 5.26 where the constraints expressed by $\Delta_{\min}$ and $\Delta_{\max}$ are depicted as points along the median that are connected to the upper or lower boundaries. As discussed in section 5.2 the median and values are determined by the pixel orientation of the pixel that contains the median, the gray level shading of the pixel and the shape of the preimage curve. The infinite convex hull of the upper and lower boundaries together with the added constraints results in determination of the passing area as shown in figure 5.27.

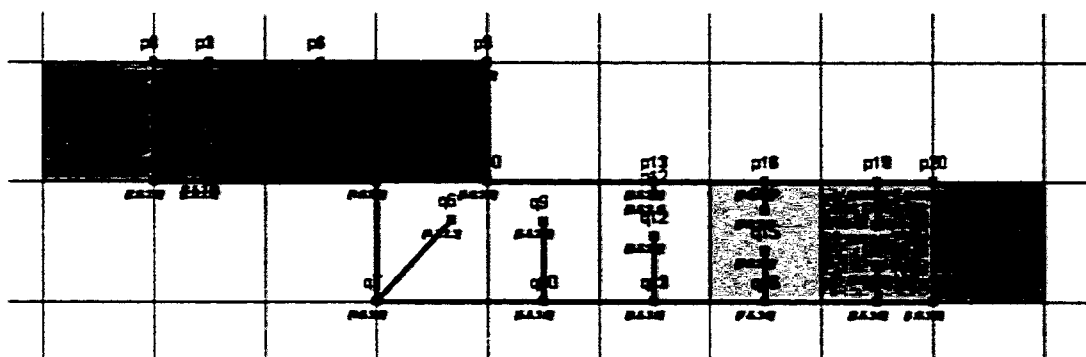


Figure 5.26. The gray scale constraints on the passing area are in addition to the upper and lower pixel boundaries

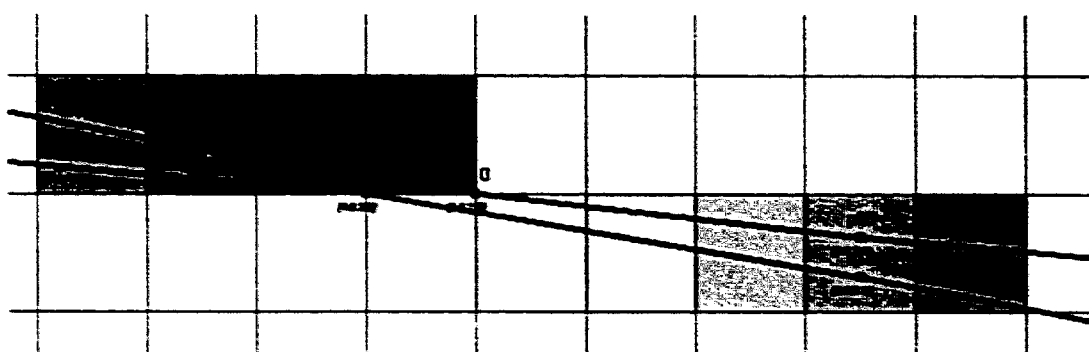


Figure 5.27. The passing area as determined by including the gray scale constraints. The size of the passing area is 1.62 pixels. Before imposing the gray scale constraints (i.e. binary constraints, only) the passing area was 3.62 pixels.

An inverse linear relationship exists between the size of the passing area and the levels of gray such that as the gray level doubles the passing area is cut in half, as shown in figure 5.28. This follows from the fact that for a linear pixel configuration $\Delta_{\min}$ and $\Delta_{\max}$ are $A - \epsilon$ and $A + \epsilon$, respectively, and $\epsilon = \frac{1}{2n}$; the distance between $\Delta_{\min}$ and $\Delta_{\max}$ is

$$2\epsilon = \frac{1}{n}.$$

Gray Scale	Area
<i>binary</i>	3.62
4	1.62
8	0.82
16	0.43
32	0.23
64	0.11
128	0.06
∞	0.0

Figure 5.28. The size of passing area of the digital line segment decreases as the number of gray levels increases.

Monte Carlo simulations were run that computed 200,000 random digital line segments whose length ranged from 6 to 140 pixels and whose gray scale levels ranged from 2 to 877. The average size of the pixel area decreased similarly to the example shown in figure 5.28. Figure 5.29 illustrates the effect that increasing gray scale levels has on the passing area size.

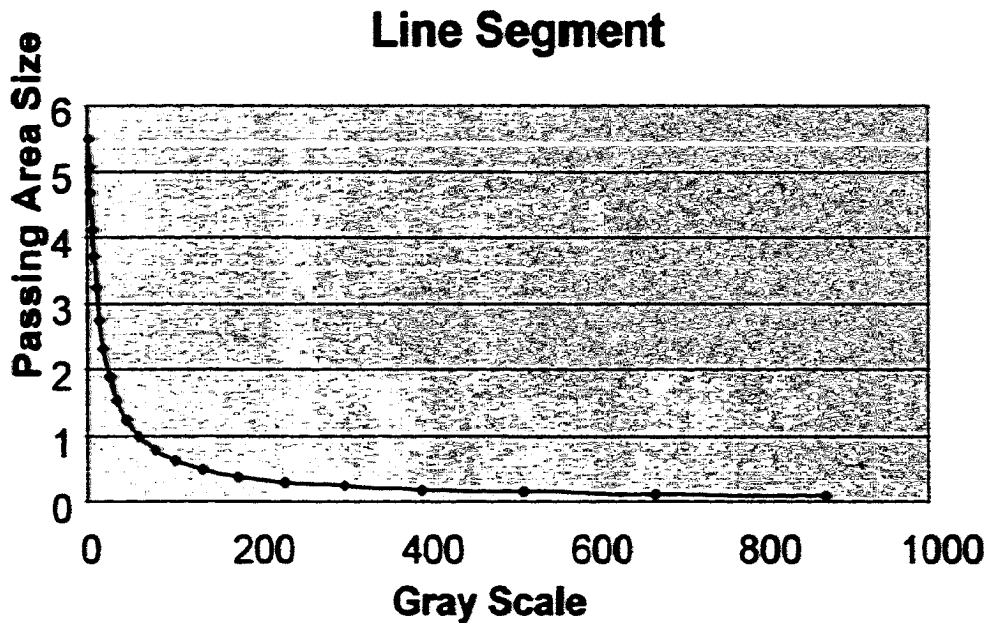


Figure 5.29. The average size of the passing area decreases as the number of gray scale levels increases.

With digital circular arcs as the levels of gray increases the size of the passing area decreases. However, as the gray levels increase, at some point the passing area size fails to decrease, as shown in figure 5.30. This appears to be a result of the non-minimal and non-maximal nature of the bounds placed on $\Delta_{\min}$ and $\Delta_{\max}$ for a circular digital curve in section 5.3.

Simulations involving 150,000 random digital circular curves ($5 \leq \text{length} \leq 1400$, $\text{binary} \leq \text{gray scale levels} \leq 1800$) similarly showed that the passing area size significantly decreases as the gray scale levels increase as shown in figure 5.31.

Gray Scale	Area
<i>binary</i>	2.53
4	1.72
8	0.88
16	0.55
32	0.35
64	0.25
128	0.20
∞	0.13

Figure 5.30. The size of the passing area of a digital curve as the number of gray levels increases. Note that as the levels of gray increases the size of the passing area converges to 0.13, as opposed to 0 that was the case with digital line segments.

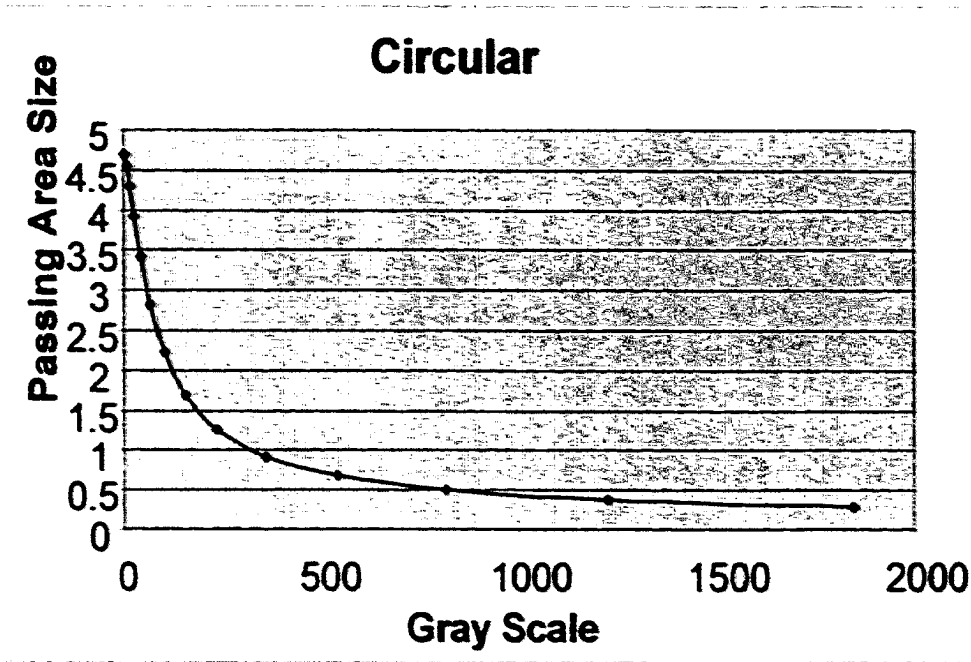


Figure 5.31. The average size of a circular digital curve’s passing area decreases as the number of gray scale levels increases.

In figure 5.32 the effects of the digital curve's length on the passing area size is illustrated. In this figure the passing area *size per pixel* is shown in place of the passing area size for the *entire* digital curve (used in previous figures). This was necessary since as a digital curve's length increases the passing area size for the *entire* curve will also increase since there are more pixels that the preimage must pass through. However, the size of the passing area through each pixel will decrease as a result of the additional constraints imposed by the added pixels. Finally, in figure 5.33 a 3D plot of how the gray scale levels and digital curve length together influence the passing area size is shown.

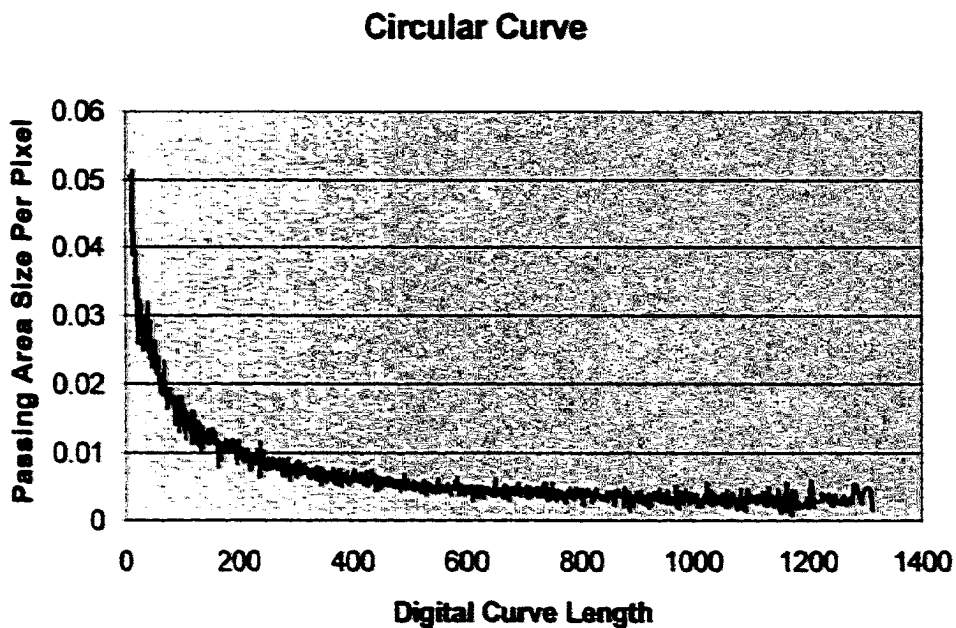


Figure 5.32. The region of each pixel that is included within the passing area decreases as the digital curve's length increases.

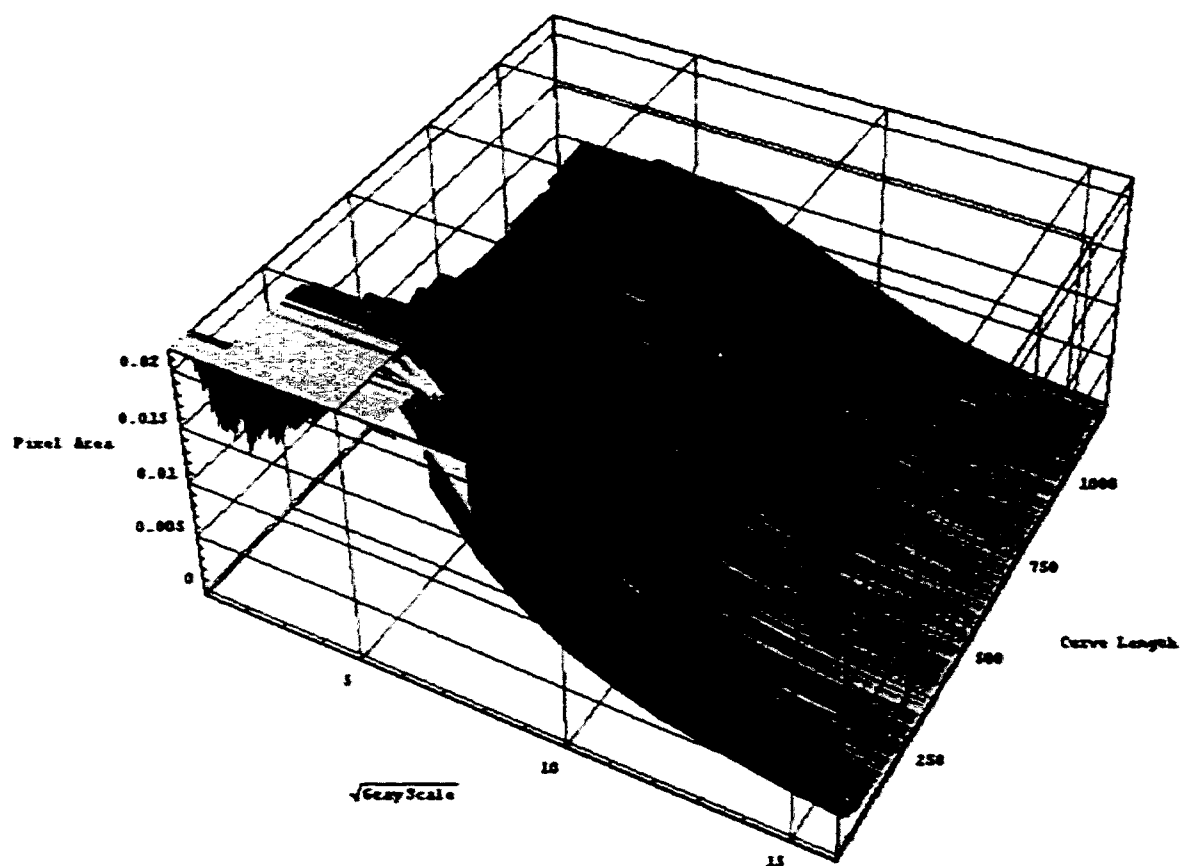


Figure 5.33. The passing area size decreases as the gray scale levels increase and curve length increases.

In figure 5.34 a digital circular arc is shown with the superimposed circular arc preimage. In figure 5.35 the gray scale constraints are added and the passing area determined for those additional constraints.

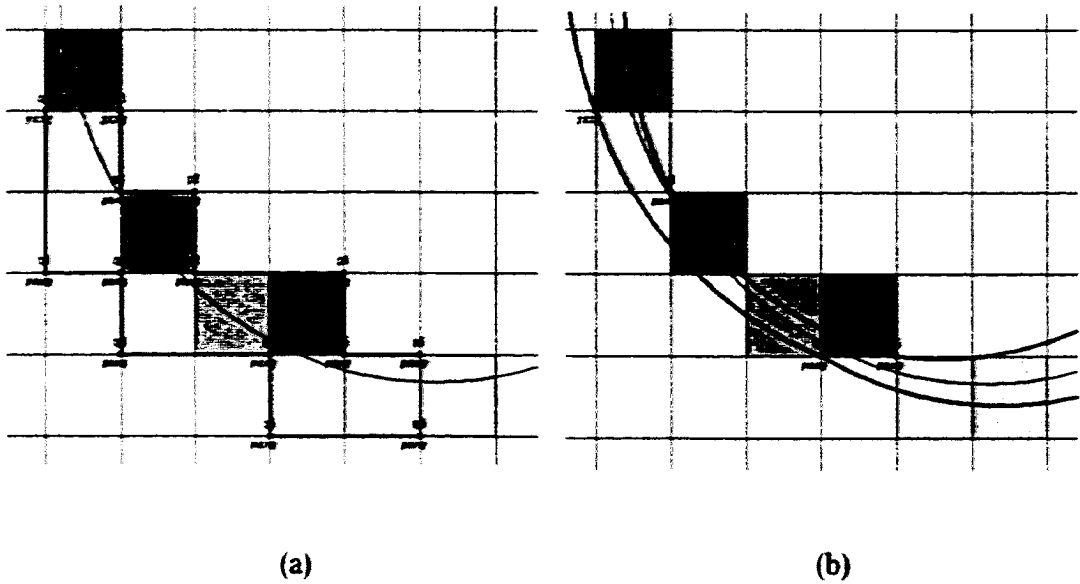


Figure 5.34. The constraint on the passing area as imposed by the binary constraints is that it must pass between the upper and lower boundaries (a). The minimal passing area is the infinite convex hull of the pixel boundary (b). The passing area's size is 2.53.

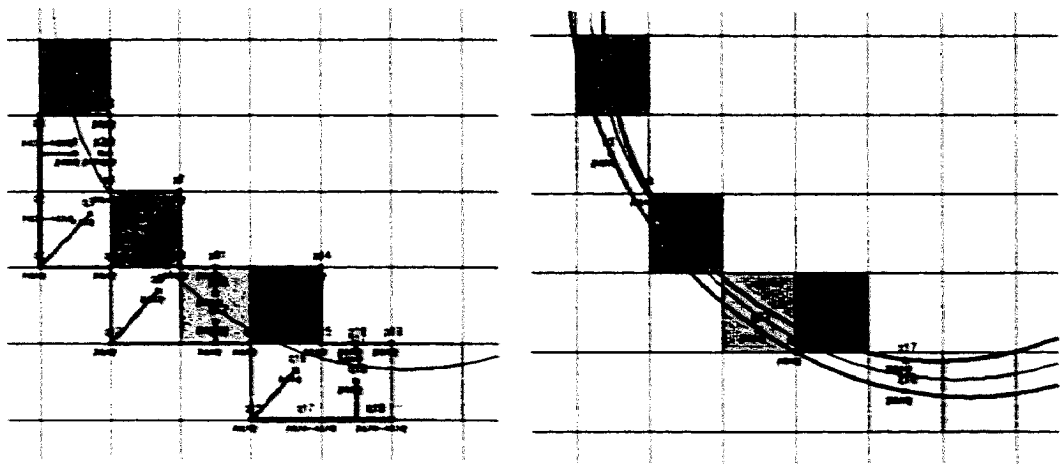


Figure 5.35. The gray scale constraints (a) further limit the size of the passing area of the digital circular arc (b). The passing area size is 1.72.

5.5 Summary

The passing area of digital binary curves is determined by first finding the pixel boundary and secondly its α -infinite hull. The area of the initial pixel boundary contains the entire region of the digital curve since binary curves indicate only whether the preimage intersects a pixel with no information regarding the localized position of that intersection. With binary images that detail must be inferred from the α -infinite hull of the pixel boundary.

Gray scale curves, however, do relate additional information regarding the position of the preimage curve within each pixel. Specifically, the gray level (under luminescence digitization) is set to the gray level percentage of the intersection of the pixel area and preimage object. This additional gray scale data can be used to further limit the range of the initial curve boundary. Subsequently, the α -infinite hull may be computed to further constrain the passing area.

With line segment bounded objects the gray level constraints may be computed completely with no need to determine lower bounds for the passing area. Thus, when incorporating gray level data in determining the passing area of a line segment the resultant passing area is in fact minimal. For objects bound by circular arcs, though, the theoretical lower limit on the passing area has been evasive and therefore non-minimal bounds are used. Still, significant increases in subpixel accuracy of the preimage are attained such that the passing area decreases in reverse proportion to the number of gray levels (up to some limit).

Finally, the effect of the number of gray levels in the image on the size of the passing area was studied. For line segments as the gray level increases the passing area size decreases in direct proportion. In the case of objects bound by circular arcs after an initial decrease in passing area size as gray levels increase the size converges to some value.

In the following section the techniques and algorithms used for determining the passing area and domain for circular arcs are extended to standard elliptical arcs.

Chapter Six:

The Passing Area and Domain of a Digital Standard Elliptical Arc (Part I):

Fixed Semi-Axes Lengths

Abstract

Given a digital curve in discrete space, an important question in computer vision and object recognition tasks is the recovery of the original curve in world space subject to the constraints of the digitization function. This chapter deals with a standard elliptical arc with known semi-axes, r_x and r_y , and computes and answers the following question in linear time: from a sequence of 4-connected pixels, determine the passing area of the curve with subpixel accuracy. The passing area is determined by computing the $\alpha\beta$ -infinite hull of the digital curve's inner and outer boundaries, where $\alpha = 1 / r_x$ and $\beta = 1 / r_y$.

From this information, the set of all possible ellipse centers (termed the *domain*) is constructed in constant time. The advantages that the computational geometric approach used in this chapter has over other methods in the literature is threefold:

- 1) it is directly parallelizable
- 2) it can readily handle a sequence of disconnected digital arcs
- 3) it appears to be extendible to more complex curves for which hulls are computable, such as general conics and three dimensional surfaces.

6. Digital Elliptical Curves

In chapter 3 the α -infinite hull was used to determine the passing area of a digital circular arc. Those results are now extended to compute the $\alpha\beta$ -infinite hull, which can be used to determine the passing area of a digital elliptical curve.

We first summarize the α -infinite hull results of chapter 3. The passing area of a digital circular arc with radius r can be determined in linear time by computing the α -infinite hull of the joined pixel boundary of the digital curve. The α -infinite hull is an extension of the α -hull (introduced by Edelsbrunner) such that the perimeter of the passing area is bounded by arcs with curvature α or $-\alpha$ where $\alpha = 1/r$. The resultant α -infinite hull is such that the constraints that each individual pixel independently places on the passing area as well as all constraints that are collectively placed by the pixel sequence are applied so that the α -infinite hull complement is the minimal passing area.

Since the α -hull is the intersection of circular discs with curvature α (or $-\alpha$) to extend the achieved results to handle more complex curves such as ellipses new definitions are required. In the next section these definitions are presented.

6.1 Definitions of Geometric Constructs

Definition: $ellipse_{\alpha\beta}$ is the standard filled ellipse with horizontal and vertical curvatures α and β , respectively. The horizontal and vertical semi-axes are r_x and r_y such

that $r_x = 1/\alpha$ and $r_y = 1/\beta$. When either $\alpha < 0$ or $\beta < 0$, $\text{ellipse}_{\alpha, \beta}$ refers to the ellipse complement, that is, $\mathbb{R}^2 - \text{ellipse}_{|\alpha|, |\beta|}$.

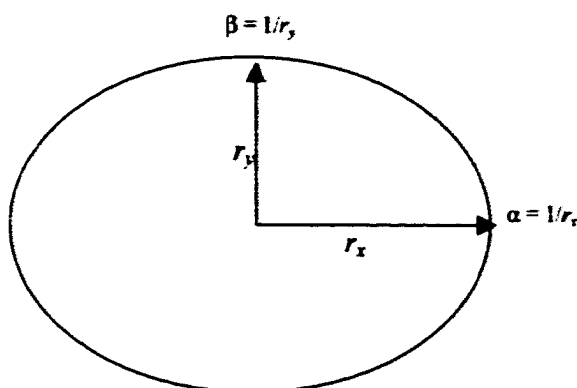


Figure 6.1. A standard ellipse with horizontal radius, r_x , and vertical radius, r_y . The curvatures, α and β , are the inverses of those radii, respectively.

Definition. The $\alpha\beta$ -hull of a point set P , $\alpha\beta\text{-hull}(P)$, is the minimum intersection of $\text{ellipse}_{\alpha\beta}$ that includes all points in P . See figure 6.2. When $\alpha < 0$ or $\beta < 0$, $\alpha\beta\text{-hull}(P)$ is the minimum intersection of ellipse complements that includes all points in P , as illustrated in figure 6.3.

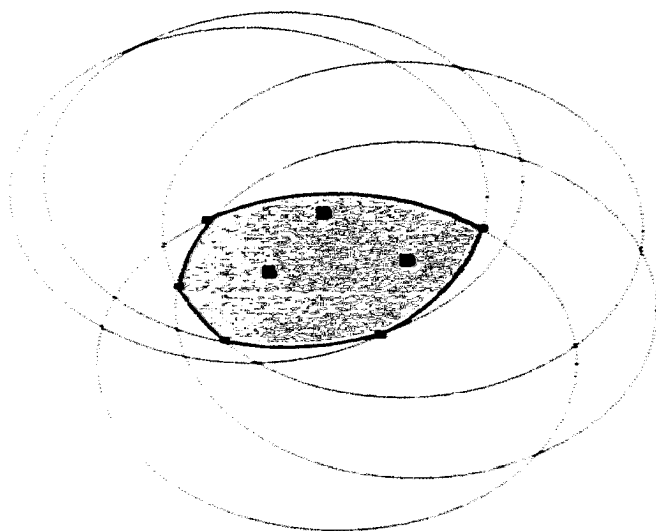


Figure 6.2. The $\alpha\beta$ -hull of a point set where $\alpha, \beta > 0$. The ellipse is the same as that shown in figure 6.1.

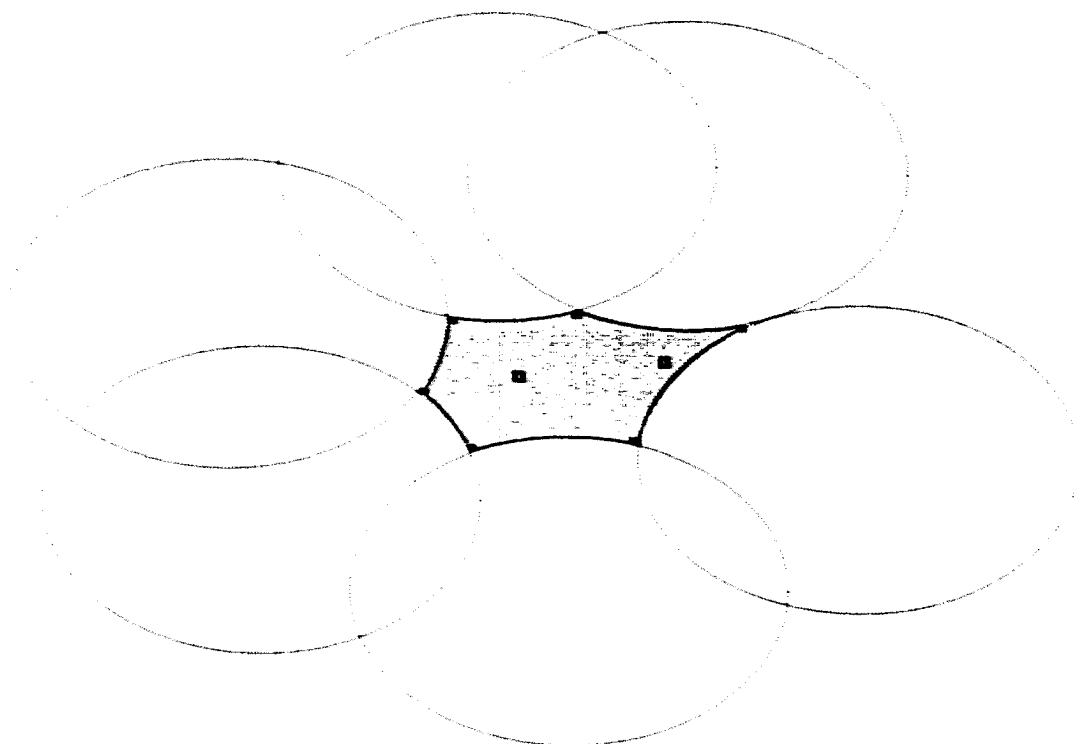


Figure 6.3. The $\alpha\bar{}$ -hull of the point set of figure 6.2 where $\alpha, \bar{} < 0$.

Definition: The $\alpha\beta$ -separation hull of two point sets P_1 and P_2 , $\alpha\beta\text{-hull}_{sep}(P_1, P_2)$ is the minimum intersection of ellipse $_{\alpha}$ that include all points in P_1 and exclude all points in P_2 . See figure 6.4. Where $\alpha < 0$ or $\beta < 0$ $\alpha\beta\text{-hull}_{sep}(P_1, P_2)$ is the minimum intersection of ellipse complements that include all points in P_1 and exclude all points in P_2 , as shown in figure 6.5.

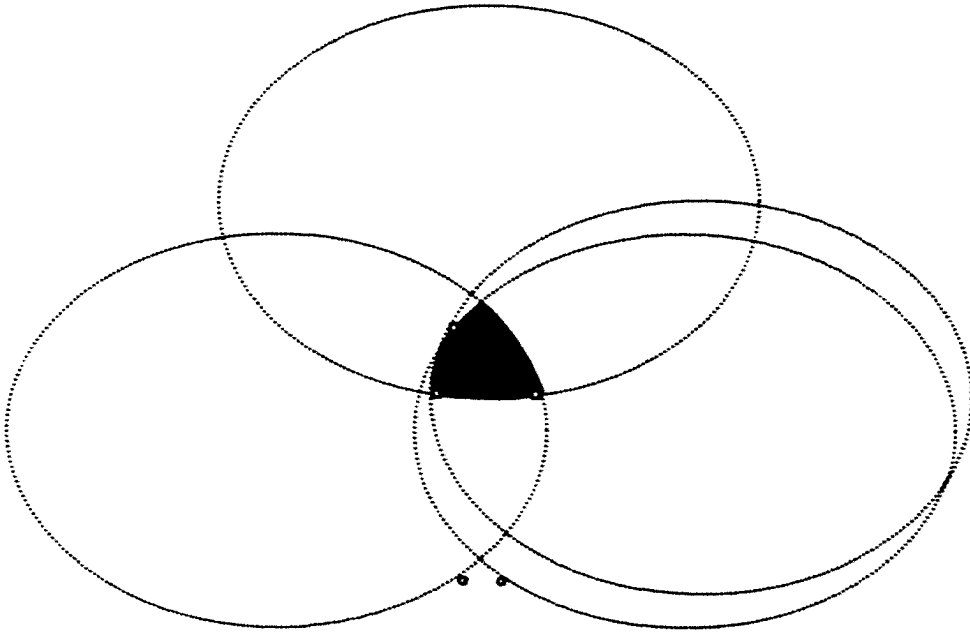


Figure 6.4. The $\alpha\beta$ -separation hull of two point sets, P_1 and P_2 where $\alpha, \beta > 0$. P_1 includes the three (square) points on top and P_2 includes the two (circular) points on the bottom.

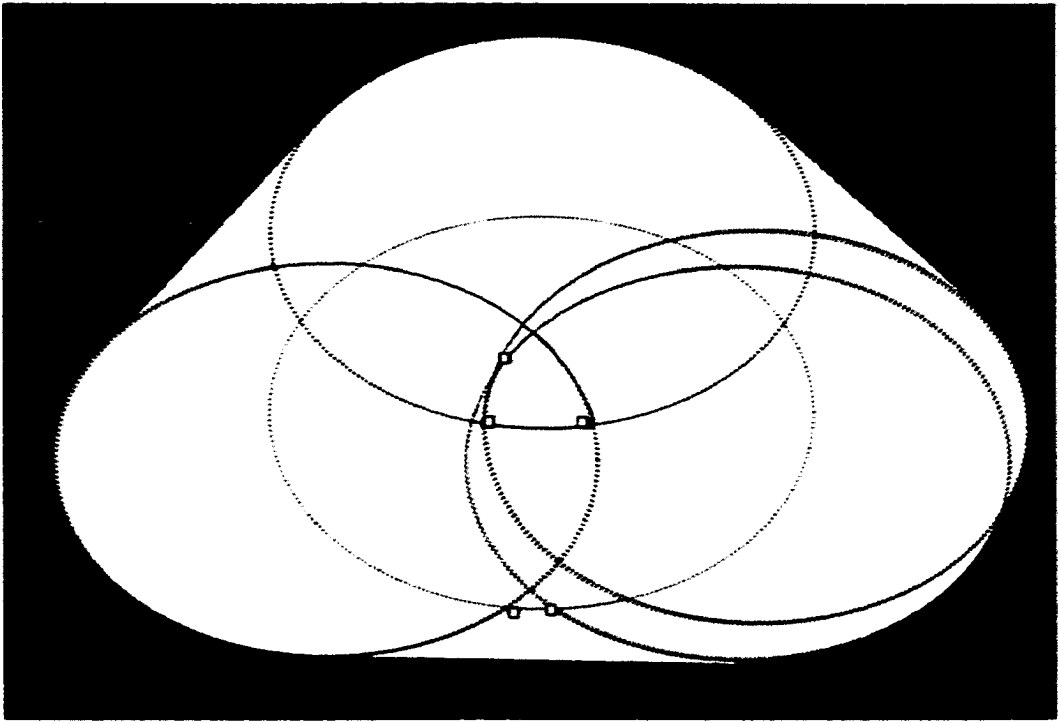


Figure 6.5. The $\alpha\beta$ -separation hull of two point sets, P_1 and P_2 where $\alpha\beta < 0$.

Definition: The *$\alpha\beta$ -infinite hull* of two point sets, P_1 and P_2 , $\alpha\beta$ -*infinite hull* (P_1, P_2) is $\alpha\beta$ - $\text{hull}_{\text{sep}}(P_1, P_2) \cup \alpha'\beta'\text{hull}_{\text{sep}}(P_2, P_1)$ where $\alpha' = -\alpha$ and $\beta' = -\beta$. See figure 6.6.

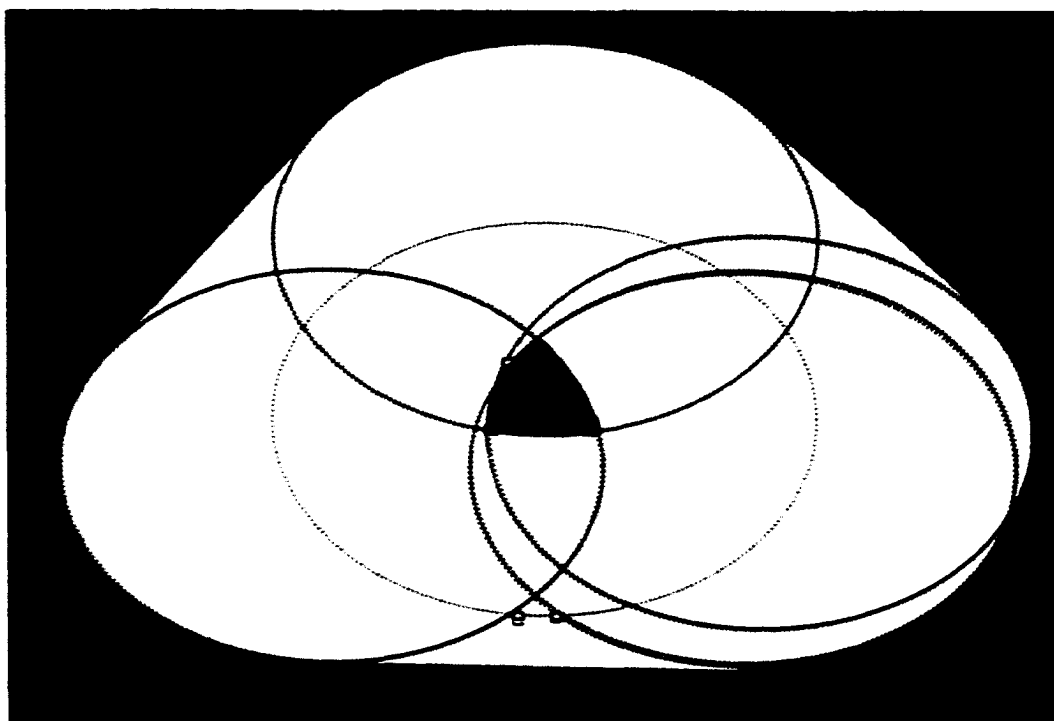


Figure 6.6. The $\alpha\beta$ -infinite hull of two point sets, P_1 and P_2 .

In figure 6.7 the inner and outer boundary elements are labeled. As with circles, only the vertices along the inner boundary constrain the passing area. On the outer boundary, however, line segment tangents at digital stationary points can also constrain the passing area. The proof that vertices and elements alone constrain the passing area is identical to the circular arc proof discussed in section 3.4.

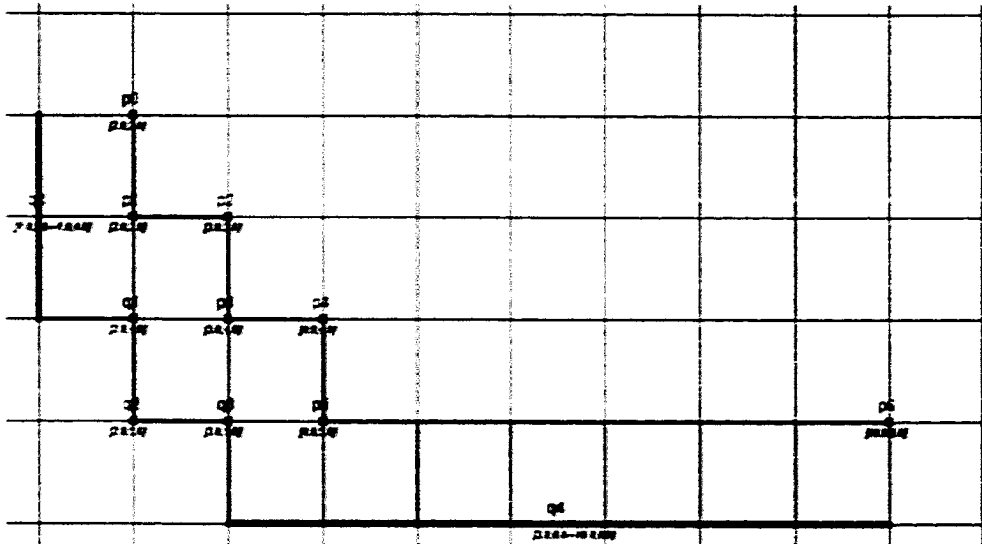


Figure 6.7. The inner and outer boundaries of a digital elliptical curve. The elements that constrain the passing area are labeled.

The passing area of a digital ellipse may be determined by computing the $\alpha\beta$ -infinite hull of the inner and outer boundaries. The domain may then be determined from the passing area, as discussed in chapter 3. The domain of an elliptical curve is parameterized by ellipse centers where each point in the domain corresponds to the center of an ellipse, a portion of which will digitize to the digital curve.

Similar to the domain of a circular digital arc (chapter 3), each vertex and arc in the digital elliptical arc's passing area corresponds to an elliptical arc and vertex along the domain perimeter. Figure 6.8 illustrates the elliptical passing area and domain where $\alpha=1/7$ and $\beta=1/4$.

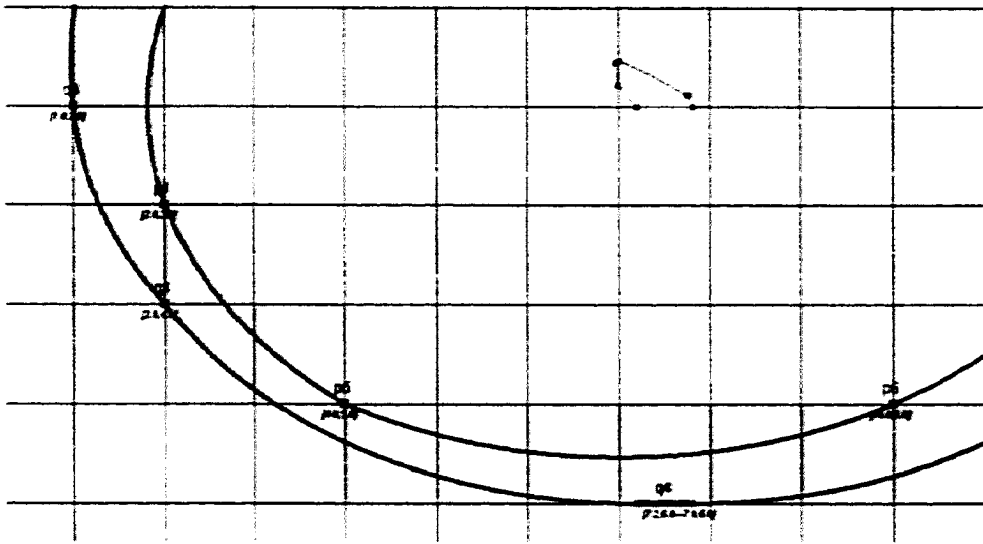


Figure 6.8. The passing area of the digital elliptical curve. The passing area is found by determining the $\alpha\beta$ -hull (P_1, P_2) ($\alpha = 0.143$, $\beta = 0.25$) where P_1 includes the elements along the inner boundary and P_2 includes the elements along the outer boundary. The domain is shown above the passing area.

6.2 Implementation Considerations

The algorithm that computes the passing area iterates through each element in the joined boundary (the collective list of inner and outer boundary elements) and determines if that element is on the passing area perimeter as determined by its immediate neighbors. In the circular arc algorithm described earlier (section 4) for three consecutive elements in the candidate list, p_i , p_j and p_k , element p_j is determined to be on the passing area based on whether p_k is inside or outside the circle intersecting p_i and p_j exactly once, without

including any other portion of p_i or p_j . Now, when dealing with ellipses, p_j is determined to be on passing area if p_k is in the ellipse that intersects p_i and p_j exactly once, without including any other portion of p_i or p_j . As with circles there are three distinct cases to consider: (A) p_i and p_j are vertices, (B) one of p_i and p_j is a vertex and the other an edge and (C) both p_i and p_j are edges. We now discuss how to handle each of these three situations.

In each case, a set of simultaneous sets of equations is solved. Mathematica (version 4.0) is used to find the solutions to these equations and the Mathematica notation for these solutions is presented. In the following discussion the following variables are defined as follows:

(h,k) the ellipse center

rx the ellipse horizontal semiaxis, r_x .

ry the ellipse vertical semiaxis, r_y .

$(x1,y1)$ a point that the ellipse intersects (p_1)

$(x2,y2)$ a point that the ellipse intersects (p_2).

We solve for (h,k) given rx , ry , $(x1,y1)$ and $(x2,y2)$.

Although for some cases multiple solutions exist, since the ellipse center as defined by two elements is real and is set to either be to the right or left of vector $\overline{p_i p_j}$ (determined whether p_i and p_j are on different boundaries), the solution is unique, as with circles.

Case A: Two vertices.

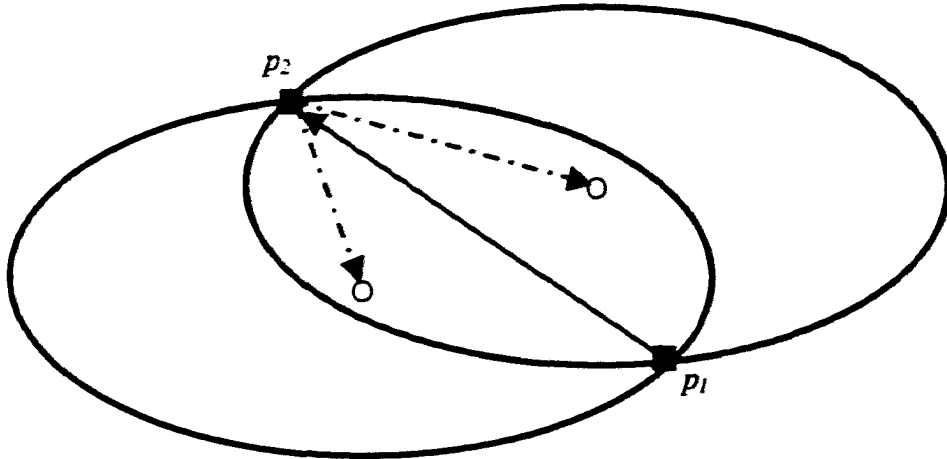


Figure 6.9. Two vertices (p_1 and p_2) and the semi-axes (r_x and r_y) define 2 ellipses: one whose center is the left of vector $\overline{p_1 p_2}$ and one to its right.

$$\text{FullSimplify[Solve[{\frac{(x1-k)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-k)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1}, \{k, h\}]]$$

$$\left\{ \left\{ h \rightarrow \frac{1}{2} \left(x1 + x2 + \frac{rx (-y1 + y2) \sqrt{(ry^2 (x1 - x2)^2 + rx^2 (y1 - y2)^2)} (-ry^2 (x1 - x2)^2 + rx^2 (2 ry + y1 - y2) (2 ry - y1 + y2))}{ry (ry^2 (x1 - x2)^2 + rx^2 (y1 - y2)^2)} \right), \right. \right.$$

$$k \rightarrow \frac{1}{2} \left(y1 + y2 + \frac{ry (x1 - x2) \sqrt{(ry^2 (x1 - x2)^2 + rx^2 (y1 - y2)^2)} (-ry^2 (x1 - x2)^2 + rx^2 (2 ry + y1 - y2) (2 ry - y1 + y2))}{rx (ry^2 (x1 - x2)^2 + rx^2 (y1 - y2)^2)} \right),$$

$$\left\{ h \rightarrow \frac{1}{2} \left(x1 + x2 + \frac{rx (y1 - y2) \sqrt{(ry^2 (x1 - x2)^2 + rx^2 (y1 - y2)^2)} (-ry^2 (x1 - x2)^2 + rx^2 (2 ry + y1 - y2) (2 ry - y1 + y2))}{ry (ry^2 (x1 - x2)^2 + rx^2 (y1 - y2)^2)} \right), \right.$$

$$\left. k \rightarrow \frac{1}{2} \left(y1 + y2 + \frac{ry (-x1 + x2) \sqrt{(ry^2 (x1 - x2)^2 + rx^2 (y1 - y2)^2)} (-ry^2 (x1 - x2)^2 + rx^2 (2 ry + y1 - y2) (2 ry - y1 + y2))}{rx (ry^2 (x1 - x2)^2 + rx^2 (y1 - y2)^2)} \right) \right\}$$

Case B: One vertex and one edge.

Without loss of generality, assume that the vertex is p_1 and the edge's linear extension's tangent point to the ellipse is $p_2 = (x_2, y_2)$. The edge may either be vertical (case 1) or horizontal (case 2).

We consider both cases:

Case B1: Edge is vertical.

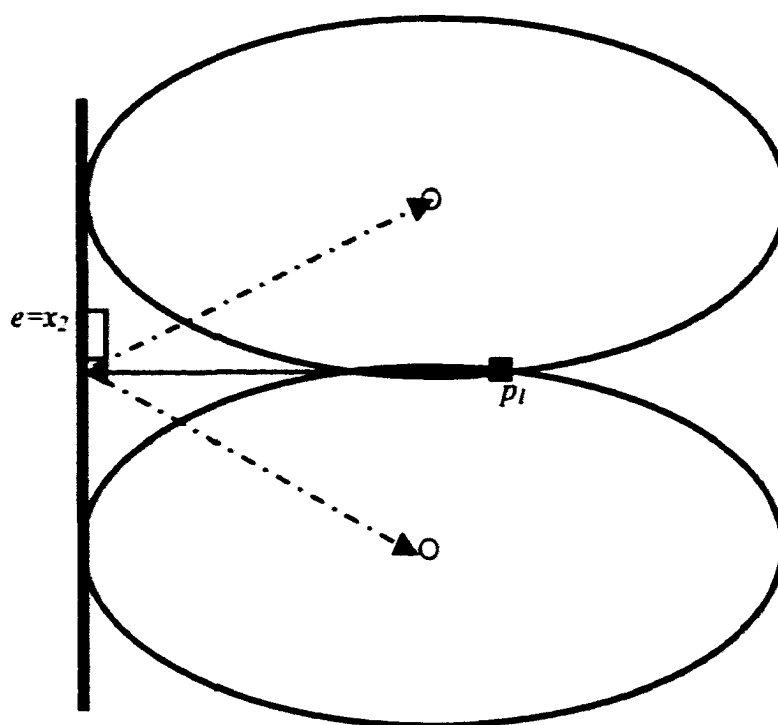


Figure 6.10. One vertex $p_1 = (x_1, y_1)$ and one edge $e (= x_2)$ and the semi-axes (r_x and r_y) define 2 ellipses: one whose center is above of the vector from $\overline{p_1 n}$ and one below it. n is the point along e such that $\overline{p_1 n}$ is perpendicular to e .

We solve for h and k .

$$\text{FullSimplify[Solve[{ $\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1$, $\frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1$, $(h+rx-x2)(h-rx-x2)=0$ }, {h, k, {y2}}]]$$

$$\left\{ \left\{ k \rightarrow \frac{ry\sqrt{(x1-x2)(2rx-x1-x2)}}{rx}, h \rightarrow rx-x2, y2 \right\}, \left\{ k \rightarrow -\frac{ry\sqrt{(x1-x2)(2rx-x1-x2)}}{rx}, h \rightarrow rx-x2, y2 \right\}, \right.$$

$$\left. \left\{ k \rightarrow \frac{\sqrt{-rx^2 ry^2 (x1-x2)(2rx-x1-x2)}}{rx^2}, h \rightarrow rx-x2, y2 \right\}, \left\{ k \rightarrow -\frac{\sqrt{-rx^2 ry^2 (x1-x2)(2rx-x1-x2)}}{rx^2}, h \rightarrow rx-x2, y2 \right\} \right\}$$

p_1 may be located to the right of e (as shown in figure 6.10) or to its right. For each case two solutions exist. Therefore 4 solutions are specified above. That is, in any situation only two of the above solutions will evaluate to values of h and k that are real (nonimaginary) numbers.

Case B2: Edge is horizontal.

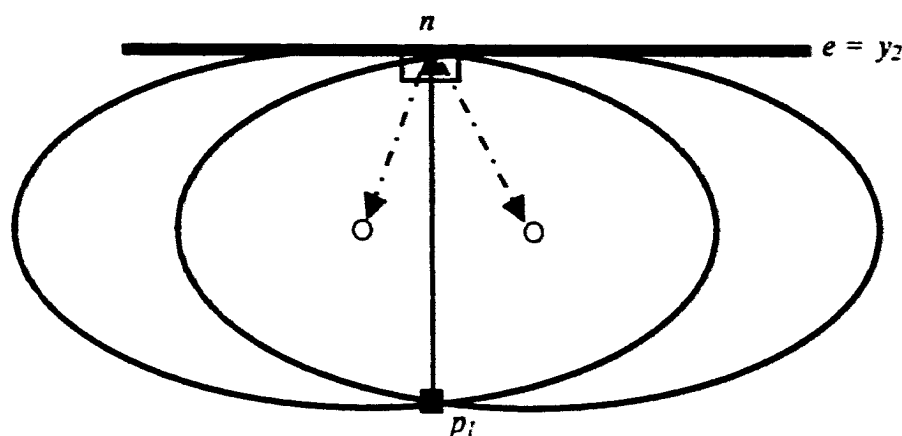


Figure 6.11. One vertex $p_1 = (x_1, y_1)$ and one edge $e (= y_2)$ and the semi-axes (rx and ry) define 2 ellipses: one whose center is to the left of the vector from $\overline{p_1 n}$ and one to its right. n is the point along e such that $\overline{p_1 n}$ is perpendicular to e .

We solve for h and k .

$$\text{FullSimplify[Solve[{\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, (k-ry-y2)(k-ry+y2) = 0}, \{h, k\}, \{x2\}]]$$

$$\left\{ \left\{ h \rightarrow x1 - \frac{rx \sqrt{(y1-y2)(2ry-y1-y2)}}{ry}, k \rightarrow ry - y2 \right\}, \left\{ h \rightarrow x1 + \frac{rx \sqrt{(y1-y2)(2ry-y1-y2)}}{ry}, k \rightarrow ry + y2 \right\}, \right.$$

$$\left. \left\{ h \rightarrow x1 - \frac{\sqrt{-rx^2 ry^2 (y1-y2)(2ry+y1-y2)}}{ry^2}, k \rightarrow -ry + y2 \right\}, \left\{ h \rightarrow x1 + \frac{\sqrt{-rx^2 ry^2 (y1-y2)(2ry+y1-y2)}}{ry^2}, k \rightarrow -ry + y2 \right\} \right\}$$

Similar to what it is mentioned in case B1, p_1 may be located below e (as shown in figure 6.11) or above it. For each case two solutions exist. Therefore 4 solutions are specified above. That is, in any situation only two of the above solutions will evaluate to values of h and k that are real (nonimaginary) numbers.

Case C: Two Edges.

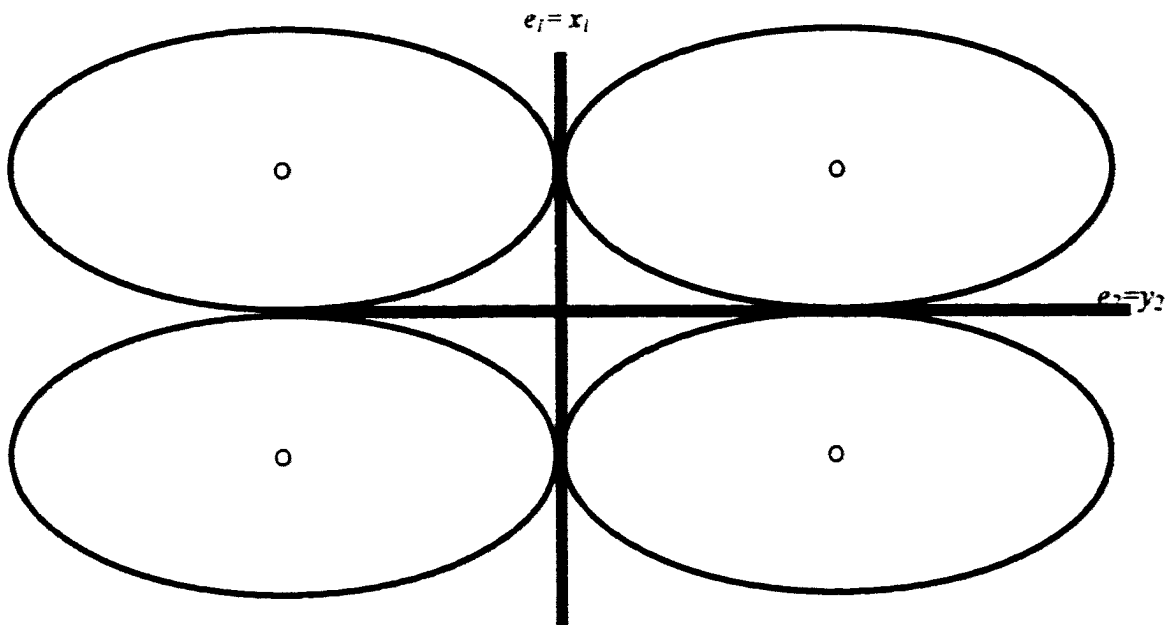


Figure 6.12. Two edges $e1 = x1$ and $e2 = y2$ have four tangent ellipses. The ellipse selected during the $\alpha\beta$ -infinite hull computation is the ellipse that is to the left of the edges (moving counterclockwise).

With no loss of generality e_1 and e_2 are the vertical and horizontal edges, respectively. We solve for h and k .

```
FullSimplify[
  Solve[{ $\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1$ ,  $\frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1$ ,  $(h-rx-x1)(h-rx-x1) = 0$ ,
     $(k-ry-y2)(k-ry-y2) = 0$ }, {h, k}, {y1, x2}]]
{{h  $\rightarrow$  -rx+x1, k  $\rightarrow$  -ry+y2}, {h  $\rightarrow$  -rx+x1, k  $\rightarrow$  ry+y2}, {h  $\rightarrow$  rx+x1, k  $\rightarrow$  -ry+y2}, {h  $\rightarrow$  rx+x1, k  $\rightarrow$  ry+y2}}
```

The two perpendicular edges have four tangent ellipses. The ellipse selected during the $\alpha\beta$ -infinite hull computation is the ellipse that is to the left of the edges (moving counterclockwise).

As mentioned above, the solutions to cases B and C will find the ellipse that is tangent to the linear extensions of the edges. That ellipse is not necessarily tangent to the edge segment from which the infinite linear extensions are constructed. (The analog to this situation with circular arcs was discussed in section 3.2.1.) Therefore, in case B, for example, if the tangent point is found not to be on the actual edge element, one of the edge's endpoints is used to determine the ellipse. In that event, case B will degenerate into case A. Each endpoint determines a single ellipse. The endpoint selected whose ellipse does not include any portion of the edge is selected.

For example consider figure 6.13 where the ellipse defined by the point and the line (dashed) are shown. When the tangent point is along the edge, the ellipse intersecting both the point and tangent point is selected, as determined by the above solutions. Where the edge does not include the tangent point, as in figures 6.14 and 6.15, the ellipse that intersects the edge exactly once that does not include the edge is chosen. The endpoint closest to the tangent point is always selected.

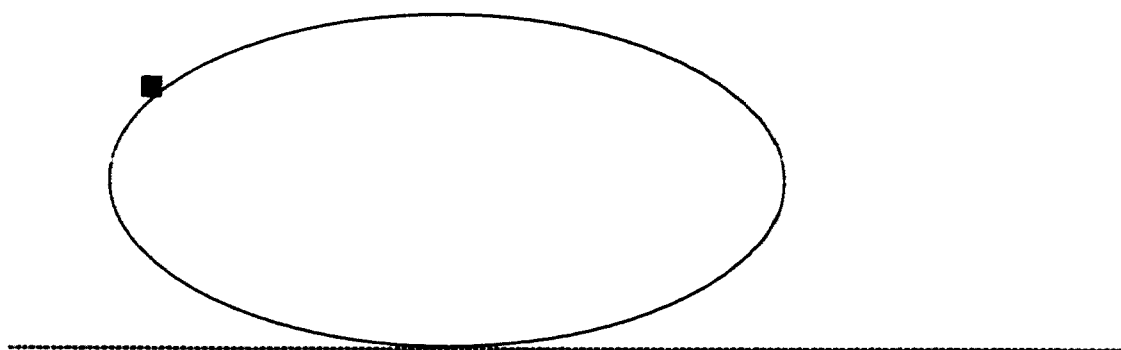


Figure 6.13. The ellipse determined by a point and a line.

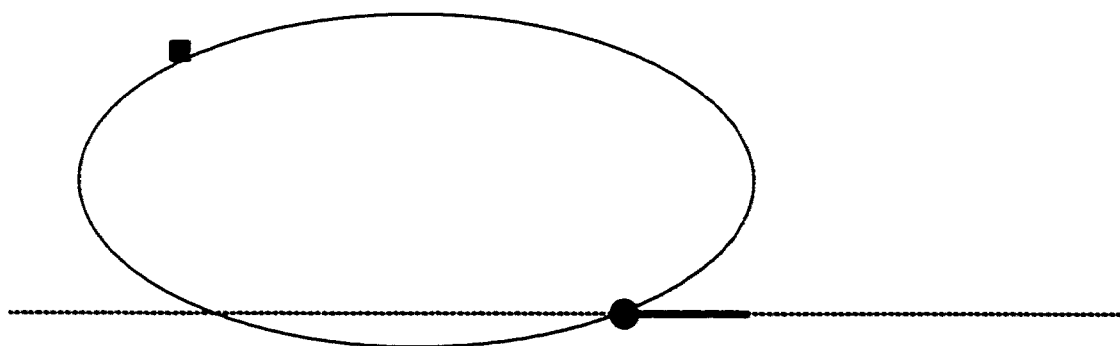


Figure 6.14. The ellipse is not tangent to any point along the (solid and darkened) edge. The ellipse that intersects the edge at one point and does not include the edge intersects the endpoint shown.

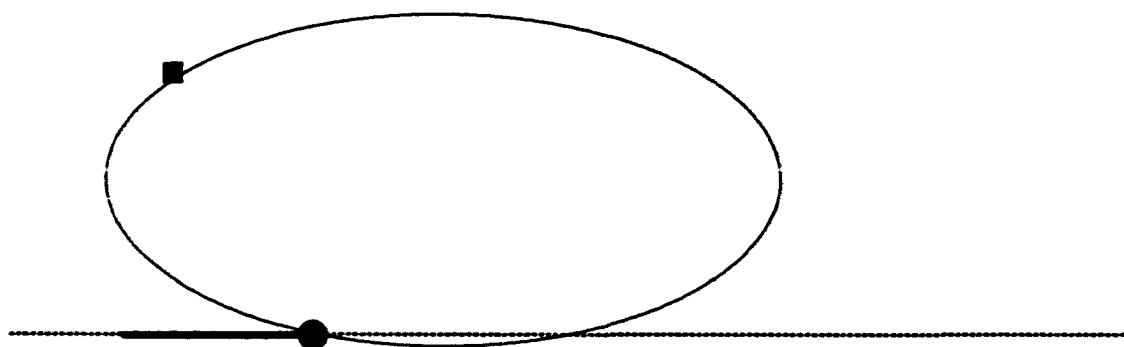


Figure 6.15. The ellipse is not tangent to any point along the (solid and darkened) edge. The ellipse that intersects the edge at one point and does not include the edge intersects the endpoint shown.

The elliptical curve passing area algorithm is almost identical to the circular curve passing area algorithm. The two algorithms diverge in one important way: whereas the circle algorithm constructs the circle as determined by the three consecutive elements in the joined boundary the ellipse algorithm constructs instead an ellipse. In all other respects the algorithms are the same. Thus, as with the circle algorithm, the upper bound on element inspections is $3n$ and, thus, the algorithm retains $n O(n)$ run time, as discussed in detail in section 3.

6.3 Disconnected Components and Parallelism

The algorithm may be used to compute the passing area and domain of disconnected digital curves as well. See figures 6.16a,b where a connected and disconnected digital curve are shown. To determine the passing area of a disconnected curve the algorithm simply iterates through the elements along the inner and outer

boundaries of all the digital curve's (disconnected) components eliminating non-constraining elements. Note that the passing area and domain of the disconnected curve (b) are significantly larger than those of the connected curve (a) and includes it since with fewer pixels there are fewer constraints.

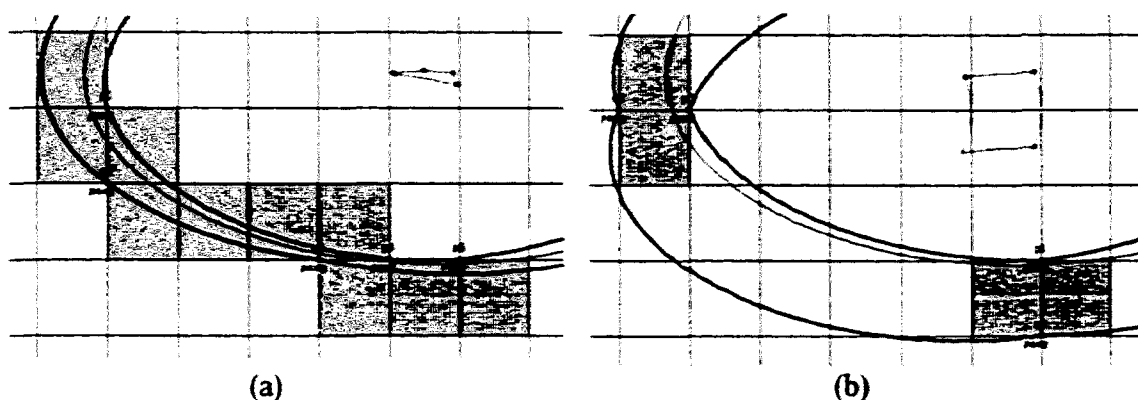


Figure 6.16. The $\alpha\beta$ -infinite hull of a connected digital elliptical arc (a) and the same curve whose components are disconnected (b). The disconnected curve, with fewer constraints, has a larger passing area and domain. The ellipse's dimensions are $r_x = 5$ and $r_y = 2.5$.

When each component of a disconnected curve contains two or more pixels the inner and outer boundaries of each component are the same as those of a connected curve. However, when components contain a single pixel the inner boundary is the pixel corner closest to the interior. The outer boundary is, similarly, the pixel corner furthest from the interior, shown in figure 6.17.

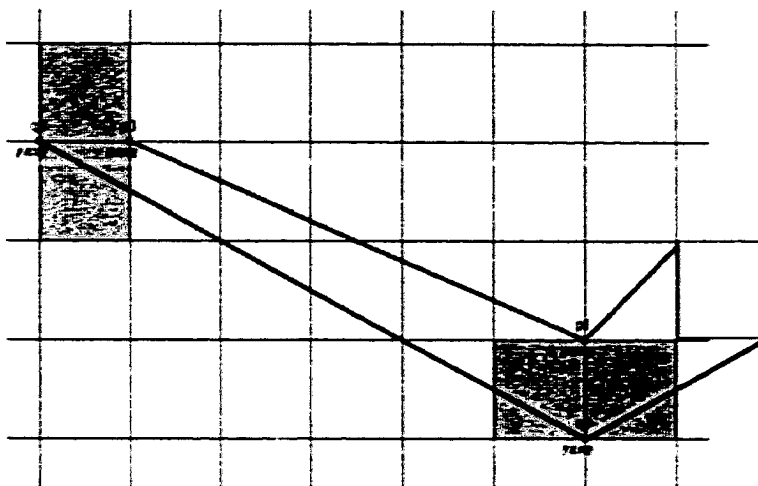


Figure 6.17. The pixel corners closest and furthest from the ellipse interior mark the inner and outer boundaries of a single pixel component.

The computation of the convex hulls of each disconnected component can be accomplished independently using the methods discussed in section 6.2, and thus can be executed in parallel. Each component can then be connected together head to tail identically to the connections between components as was shown in figure 3.17.

Assuming m components of equal size with a total of n vertices, the α_x -hull of each component can be computed in $3n/m$ iterations. Solving in parallel, with m processors, each processor may compute the infinite hull of one component. Let the maximum number of elements in the passing area be MAX . Then, each processor may compute the infinite hull of two components, with an upper bound of $6 MAX$ iterations. The number of processors used, for this step is $m/2$ because each processor is now processing 2 components. Each successive step uses half the processors of the previous

step until a single processor computes the infinite hull. The efficiency of such a parallel algorithm is $3n/m + 6 MAX \log_2(m)$.

The maximum number of elements on the passing area, defined here as *MAX*, appears to have an upper bound (from empirical evidence). After running numerous simulations the maximum number of elements was found to be 14. What needs clarification is whether in fact this is the upper bound and if it is what the mathematical proof may be. The domain can be computed in parallel in constant time from the information of the passing area.

6.4 Summary

In this section the system for determining the passing area and domain of a standard elliptical digital curve was found to be computable by determining the $\alpha\beta$ -infinite hull of the digital curve's joined boundary. The algorithm's complexity is $O(n)$ thus maintaining the same linear run time achieved for digital line segments and circular arcs. This suggests, possibly, that the algorithm may be extended to handle more complex conics with the same efficiency.

In the next section we present an algorithm to determine the domain of a digital standard elliptical curve with fixed eccentricity but unknown semi-axes. This follows the approach developed earlier (section 4) where the domain of a digital circular arc with unknown radius is determined. Based on empirical data the complexity of this algorithm, as well, is linear.

Chapter Seven:**The Passing Area and Domain of a Digital Elliptical Arc (Part II):****Fixed Rotation Eccentricity for All Semi-Axes Lengths**

Abstract

The $\alpha\beta$ -infinite hull (of chapter 6) is a geometric construct that connects points (of the passing area) by elliptical arcs where each arc is a portion of the standard ellipse with horizontal and vertical semi-axes equaling $1/\alpha$ and $1/\beta$, respectively. By constructing the $\alpha\beta$ -infinite hull computing the passing area of any digital elliptical arc in a runtime of $O(n)$ can be accomplished. The methodology used for determining the passing area of an ellipse where the semi-axes and rotation of the underlying arc are known can be extended to determining the passing area in the case of a digital elliptical arc with known eccentricity where the lengths of the semi-axes are unknown. In this chapter, an algorithm is presented that determines the semi-axes values where the elements (vertices and possibly tangent segments) along the passing area perimeter change. The semi-axes values where such changes occur can directly be used to find the domain of the digital curve for all semi-axes values for ellipses of the same eccentricity. Based on experimental simulations, the computation has an $O(n)$ run time complexity.

7. Introduction

In section 4 the algorithm for determining the domain of a digital circular arc under all radii was presented. The direct relationship between the domain of the digital arc and its passing area in the digital plane was demonstrated there as well. In this section the necessary adaptations to those definitions, computations and algorithms to determine the passing area of a standard elliptical curve with fixed rotation is discussed. The circle algorithm was founded on determining the critical changes of the digital curve's passing area in a predictable manner. Similar critical changes take place to the passing area of elliptical curves.

In section 7.1 we present the method for determining when critical changes take place in a digital elliptical curve's passing area. Based on these results the adaptation to the circle algorithm (section 4.3) will solve for the passing area over all radii of an elliptical digital curve where the eccentricity and rotation of the major semi-axis from the x-axis are known. Although the examples given are with zero rotation (i.e., a standard ellipse) the algorithm may be applied to find the passing area for any ellipse with a fixed rotation. In section 7.2 the shape of the digital elliptical curve's domain is presented.

7.1 The Standard Elliptical Passing Area for All Radii

Earlier (in section 6) the $\alpha\beta$ -infinite hull was defined which was used to determine the passing area of a digital elliptical curve. The values of α and β are the horizontal and vertical curvatures, respectively. With horizontal and vertical semi-axes, r_x

and r_y , respectively, $\alpha = 1 / r_x$ and $\beta = 1 / r_y$, as depicted in figure 7.1. See figure 7.2 for the passing area of a digital elliptical curve where $r_x = 7$ and $r_y = 4$.

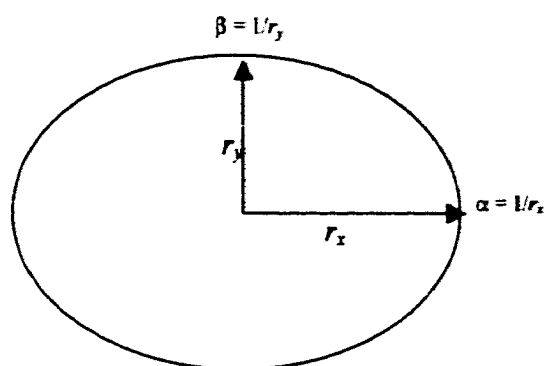


Figure 7.1. A standard ellipse with horizontal radius, r_x , and vertical radius, r_y . The curvatures, α and β , are the reciprocals of those radii, respectively.

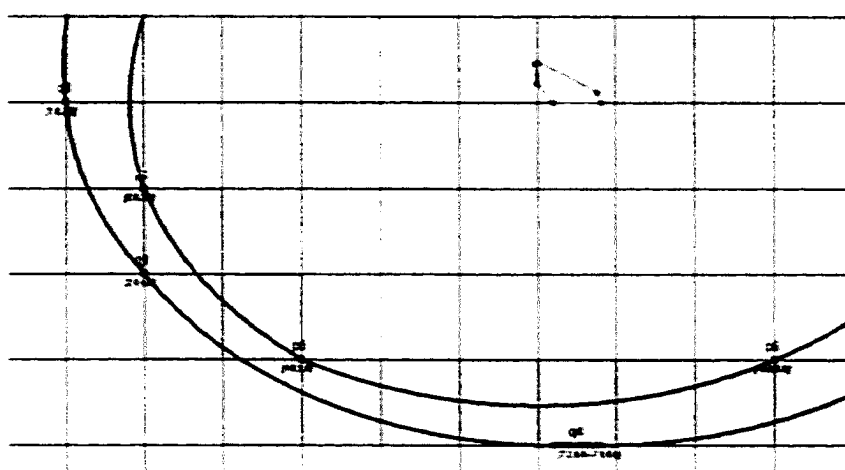


Figure 7.2. The passing area of the digital elliptical curve. The passing area is found by determining the $\alpha\beta$ -hull $_{\alpha}(P, Q)$ ($\alpha = 1/7$, $\beta = 1/4$) where P includes the elements along the inner boundary and Q includes the elements along the outer boundary.

With a fixed eccentricity, ecc , the values of r_y may be defined in terms of r_x . The value of ecc is $\frac{\sqrt{r_x^2 - r_y^2}}{r_x} < 1$ [Vossler, 2000]. Solving for r_y in terms of r_x we have $r_y = r_x \sqrt{1 - ecc^2}$ since $r_x, r_y > 0$. The notation used here is to fix the ratio of semi-axes such that $ratio = \frac{r_x}{r_y}$. $ratio$ can then be expressed in terms of the eccentricity: $ratio = \frac{1}{\sqrt{1 - ecc^2}}$ and r_y may be expressed in terms of $ratio$ and r_x : $r_y = \frac{r_x}{ratio}$. With $ratio$ fixed, r_y grows with r_x . We therefore refer to an increasing horizontal radius, r_x , without need for the explicit mention of the corresponding increase of r_y .

As the radii of a proposed preimage ellipse changes, the elements along its passing area change. The radius value when these critical changes occur is dependent on the immediate neighbors of each element along the joined boundary. Specifically, the conditional critical range of an element, e , denoted $\langle e | e^-, e^+ \rangle_{ratio}$ is the radial range when that element will be found in the passing area. $\langle e | e^-, e^+ \rangle_{ratio}$ will be dependent on the value of the ellipse defined by e and its previous and successive neighbors, e^- and e^+ , respectively. The radius of this ellipse is denoted $r_{ellipse}(ratio, e^-, e, e^+)$. The value of the conditional critical range will either be $(0, r_{ellipse}(ratio, e^-, e, e^+)]$ or $(r_{ellipse}(ratio, e^-, e, e^+), \infty]$, dependent on the locations of e^-, e and e^+ , as established in section 3.

The value of $r_{ellipse}(ratio, e^-, e, e^+)$ is set to the horizontal semi-axis value of the largest ellipse with semi-axes $ratio$ such that the ellipse intersects e^-, e and e^+ each at most once counterclockwise. For example, where $ratio = 1$ (representing a circle) and e^-, e and e^+ are points the value of $r_{ellipse}(ratio, e^-, e, e^+)$ is the radius of the circle centered at the intersection of the perpendicular bisector of e^- and e with the perpendicular bisector

of e and e^- , as shown in figure 7.3a. Where $\text{ratio} \neq 1$ $r_{\text{ellipse}}(\text{ratio}, e^-, e, e^-)$ is set to the horizontal semi-axis of the ellipse that intersects the elements once (figure 7.3b). We shall later see by using analytical geometry that only one ellipse with a fixed ratio will intersect all 3 elements in order counterclockwise.

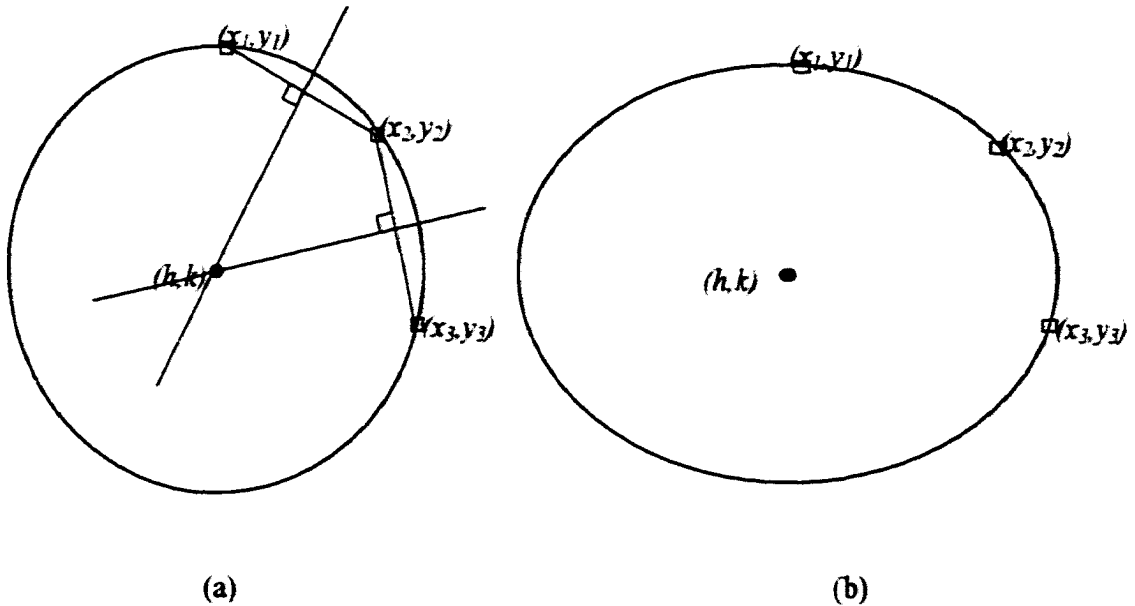


Figure 7.3. The ellipse with known semi-axes ratio that intersects three points is unique. The values of *ratio* in (a) and (b) are 1 and 1.6, respectively.

Since an element may either be a vertex or an edge four combinations (labeled A-D) for the elements' identities exist as indicated in figure 7.4. Consequently the value of $r_{\text{ellipse}}(\text{ratio}, e^-, e, e^-)$ for these four cases are now considered.

- A) • • •
 B) • • —
 C) • — —
 D) — — —

Figure 7.4. The computation of $r_{\text{ellipse}}(\text{ratio}, e^-, e, e^+)$ is dependent on the whether e^- , e and e^+ are points or edges.

Four combinations exist. “•” and “—” denote points and edges, respectively.

The following solutions express the value of r_{ellipse} as determined by the three elements. As in the earlier discussions, the intersection points along the three elements e^- , e and e^+ are (x_1, y_1) , (x_2, y_2) and (x_3, y_3) , respectively. The ellipse horizontal and vertical semi-axes are r_x and r_y , respectively, and the ellipse center is (h,k) .

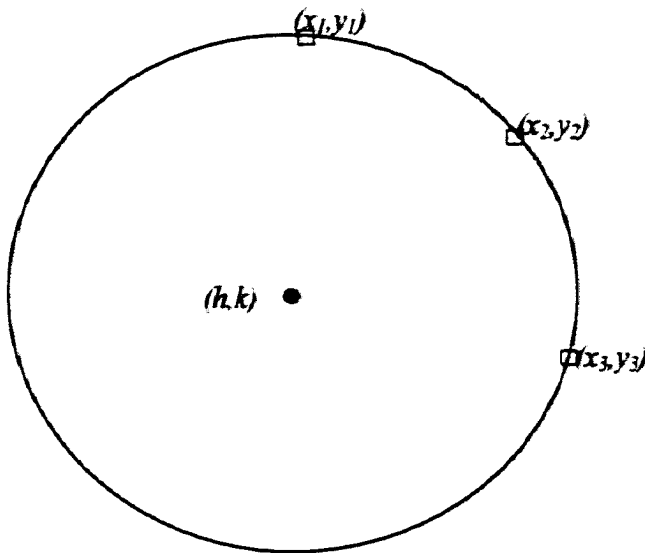
The four cases enumerated below use the formula for r_{ellipse} explicitly as part of the set of equations. This set of equations are solved for the semi-axes, r_x and r_y . When an element is an edge, the equations are solved for the tangent point of the ellipse along the given edge.

The values of all variables must be real and positive and the ellipses must intersect e^- , e and e^+ at most once, in order, counter clockwise. Therefore, a single correct geometric solution exists for each set of mathematical solution presented below. As discussed with circles (section 4.2.3.2.2), for cases B and C, if the ellipse is not tangent to an edge, the ellipses determined by the edge's endpoints are found and the ellipse that does not include the edge is selected. The mathematical solutions for all cases are found below.

Since the geometric constraints involve ellipses they can be translated to a system of quadratic equations. Mathematica (version 4.0) was employed to solve these simultaneous equations. Therefore, the solutions are accompanied with a sprinkling of Mathematica notation. Specifically, each set of equations is solved, simplified and, when necessary, redundant solutions eliminated using the `Solve`, `FullSimplify` and `Union` commands, respectively. Solutions are presented as comma separated lists where the arrow symbol, " $\rightarrow$ ", is interpreted as "equal to". Each solution is enclosed in curly braces "{ }". For example, to solve for the intersection point (x,y) of two lines, $y = m_1 x + b_1$ and $y = m_2 x + b_2$, where m_1 , m_2 are the slopes and b_1 , b_2 are the y-intercepts, `Solve` is called as follows: `Solve[{y == m1 x + b1 and y == m2 x + b2},{x,y}]`.

To simplify the results and eliminate any possible redundant solutions `FullSimplify` and `Union` are called as follows: `Union[ FullSimplify [Solve[{y == m1 x + b1 and y == m2 x + b2},{x,y}] ] ]`

Case A: Three Vertices: (• • •)



A standard ellipse centered at (h,k) with semi-axes rx and ry intersecting a point

(x,y) must satisfy the equation $\frac{(x-h)^2}{rx^2} + \frac{(y-k)^2}{ry^2} = 1$. Solving for the ellipse center (h, k)

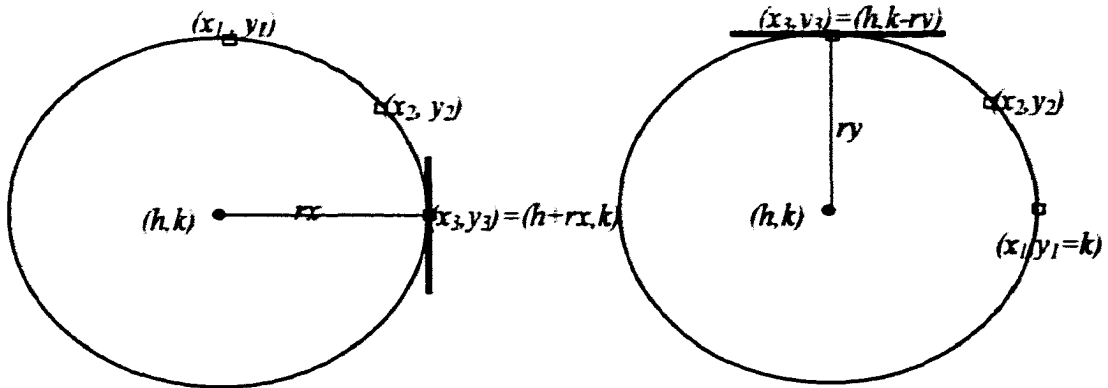
and radius, r , that intersect three points $(x1, y1)$, $(x2, y2)$ and $(x3, y3)$ we have:

Union[
FullSimplify[
Solve[{ $\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1, \text{ratio} = \frac{rx}{ry}$ },
{ rx, ry }, { h, k }]]]

$\left\{ \left\{ rx \rightarrow -\left(\sqrt{\left((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2 \right) \left((x1-x3)^2 + \text{ratio}^2 (y1-y3)^2 \right) \left((x2-x3)^2 + \text{ratio}^2 (y2-y3)^2 \right)} \right) / \right.$
 $\left. \left(2 \text{ratio} \sqrt{(x3(-y1+y2) + x2(y1-y3) + x1(-y2+y3))^2} \right), \right.$
 $ry \rightarrow -\left(\sqrt{\left((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2 \right) \left((x1-x3)^2 + \text{ratio}^2 (y1-y3)^2 \right) \left((x2-x3)^2 + \text{ratio}^2 (y2-y3)^2 \right)} \right) /$
 $\left. \left(2 \text{ratio}^2 \sqrt{(x3(-y1+y2) + x2(y1-y3) + x1(-y2+y3))^2} \right) \right\},$
 $\left\{ rx \rightarrow \left(\sqrt{\left((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2 \right) \left((x1-x3)^2 + \text{ratio}^2 (y1-y3)^2 \right) \left((x2-x3)^2 + \text{ratio}^2 (y2-y3)^2 \right)} \right) / \right.$
 $\left. \left(2 \text{ratio} \sqrt{(x3(-y1+y2) + x2(y1-y3) + x1(-y2+y3))^2} \right), \right.$
 $ry \rightarrow \left(\sqrt{\left((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2 \right) \left((x1-x3)^2 + \text{ratio}^2 (y1-y3)^2 \right) \left((x2-x3)^2 + \text{ratio}^2 (y2-y3)^2 \right)} \right) /$
 $\left. \left(2 \text{ratio}^2 \sqrt{(x3(-y1+y2) + x2(y1-y3) + x1(-y2+y3))^2} \right) \right\} \}$

The only positive value of rx and ry are in the second solution and thus only the second solution is valid.

Case B: Two Vertices and One Edge: (• • —)



The ellipse e must intersect all three elements in exactly one point and not include any portion of the edge. The equations presented below solve for the tangent point (x_3, y_3) along the linear extension of the edge. The edge may either be vertical (case 1) or horizontal (case 2). When the edge is vertical, y_3 is fixed at k and we solve for x_3 . Similarly, when the edge is horizontal, x_3 is fixed ($= h$) and we solve for y_3 .

Since the ellipse is tangent to the edge at (x_3, y_3) when the edge is vertical (horizontal) the distance from the tangent point to the center (h, k) is rx (ry) along the horizontal (vertical) axis. Thus, with a vertical edge that is to the left (right) the vertices $h - rx = x_3$ ($h + rx = x_3$). For a horizontal edge that is above the vertices $k - ry = y_3$ ($k + ry = y_3$). We solve for rx and ry .

The solutions for the simultaneous equations for both cases are expressed below. Of the four solutions only the second and third have a positive and therefore valid horizontal semi-axis rx . Although two solutions with positive rx exist only one ellipse exists that intersects the three elements, e^+ , e and e^- in order counterclockwise, as shown in figure 7.5.

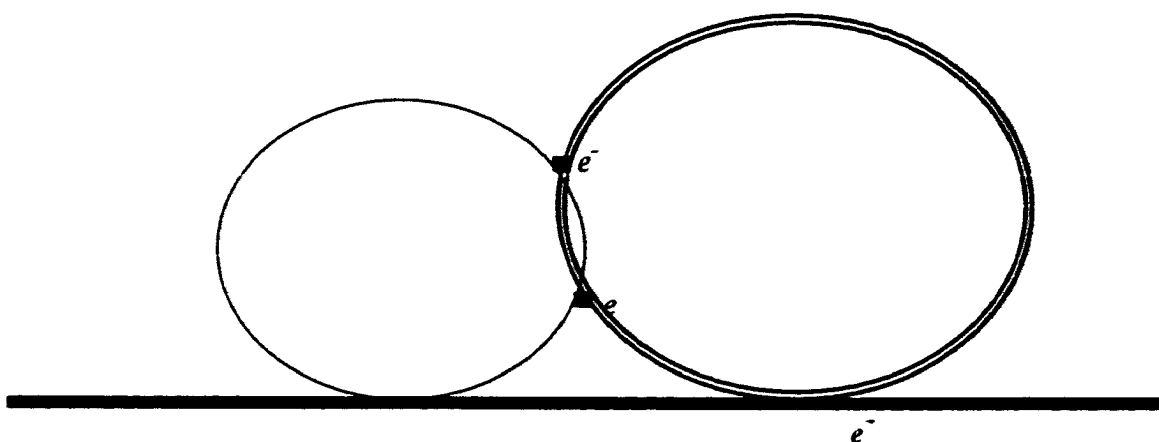


Figure 7.5. Two circles exist that intersect two points and are tangent to a line. Only one will intersect e^- , e and e^- in order counter clockwise, as shown with the double lined circle.

Case 1: Vertical Edge

Case a: The edge is to the left of the vertices

$$\begin{aligned}
 &\text{Union[} \\
 &\quad \text{FullSimplify[} \\
 &\quad \text{Solve[} \left\{ \frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1, \right. \\
 &\quad \left. \text{ratio} = \frac{rx}{ry}, h - rx == x3 \right\}, \{rx, ry\}, \{h, k, y3\} \} \\
 &\quad \left\{ \left\{ y \rightarrow \frac{1}{2 \text{ratio} (x1-x2)^2} \left((x1+x2-2x3) ((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) - \right. \right. \right. \\
 &\quad \left. \left. 2 \text{ratio} \sqrt{(x1-x3)(x2-x3)} \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} (y1-y2) \right), \right. \\
 &\quad \left. rx \rightarrow \frac{1}{2 (x1-x2)^2} \left((x1+x2-2x3) ((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) - \right. \right. \\
 &\quad \left. \left. 2 \text{ratio} \sqrt{(x1-x3)(x2-x3)} \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} (y1-y2) \right) \right\}, \\
 &\quad \left\{ \left\{ y \rightarrow \frac{1}{2 \text{ratio} (x1-x2)^2} \left((x1+x2-2x3) ((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) + \right. \right. \right. \\
 &\quad \left. \left. 2 \text{ratio} \sqrt{(x1-x3)(x2-x3)} \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} (y1-y2) \right), \right. \\
 &\quad \left. rx \rightarrow \frac{1}{2 (x1-x2)^2} \left((x1+x2-2x3) ((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) + \right. \right. \\
 &\quad \left. \left. 2 \text{ratio} \sqrt{(x1-x3)(x2-x3)} \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} (y1-y2) \right) \right\} \} \\
 &\quad \left. \right] \\
 &\quad \left. \right]
 \end{aligned}$$

Case b: The edge is to the right of the vertices

$$\begin{aligned}
 & \text{Union[} \\
 & \quad \text{FullSimplify[} \\
 & \quad \quad \text{Solve[} \left\{ \frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1, \right. \\
 & \quad \quad \left. \text{ratio} = \frac{rx}{ry}, h + rx == x3 \right\}, \{rx, ry\}, \{h, k, y3\}]]] \\
 & \{ \{ ry \rightarrow \frac{1}{2 \text{ratio} (x1-x2)^2} \left(-(x1+x2-2x3) ((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) + \right. \\
 & \quad \left. 2 \text{ratio} \sqrt{(x1-x3)(x2-x3)} \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} (y1-y2) \right), \\
 & \quad rx \rightarrow \frac{1}{2 (x1-x2)^2} \left(-(x1+x2-2x3) ((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) + \right. \\
 & \quad \left. 2 \text{ratio} \sqrt{(x1-x3)(x2-x3)} \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} (y1-y2) \right) \}, \\
 & \{ ry \rightarrow -\frac{1}{2 \text{ratio} (x1-x2)^2} \left((x1+x2-2x3) ((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) + \right. \\
 & \quad \left. 2 \text{ratio} \sqrt{(x1-x3)(x2-x3)} \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} (y1-y2) \right), \\
 & \quad rx \rightarrow -\frac{1}{2 (x1-x2)^2} \left((x1+x2-2x3) ((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) + \right. \\
 & \quad \left. 2 \text{ratio} \sqrt{(x1-x3)(x2-x3)} \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} (y1-y2) \right) \} \}
 \end{aligned}$$

Case 2 : Horizontal Edge

Case a: The edge is above the vertices

$$\begin{aligned}
 & \text{Union[} \\
 & \quad \text{FullSimplify[} \\
 & \quad \quad \text{Solve[} \left\{ \frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1, \right. \\
 & \quad \quad \left. \text{ratio} = \frac{rx}{ry}, k - ry == y3 \right\}, \{rx, ry\}, \{h, k, x3\}]]] \\
 & \{ \{ rx \rightarrow \frac{1}{2 \text{ratio} (y1-y2)^2} \left(((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) (y1+y2-2y3) - \right. \\
 & \quad \left. 2 (x1-x2) \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} \right), \\
 & \quad ry \rightarrow \frac{1}{2 \text{ratio}^2 (y1-y2)^2} \left(((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) (y1+y2-2y3) - \right. \\
 & \quad \left. 2 (x1-x2) \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} \right) \}, \\
 & \{ rx \rightarrow \frac{1}{2 \text{ratio} (y1-y2)^2} \left(((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) (y1+y2-2y3) + \right. \\
 & \quad \left. 2 (x1-x2) \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} \right), \\
 & \quad ry \rightarrow \frac{1}{2 \text{ratio}^2 (y1-y2)^2} \left(((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) (y1+y2-2y3) + \right. \\
 & \quad \left. 2 (x1-x2) \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} \right) \} \}
 \end{aligned}$$

Case b: The edge is below the vertices

$$\begin{aligned}
 & \text{Union}[\\
 & \quad \text{FullSimplify}[\\
 & \quad \quad \text{Solve}\left[\left\{\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1, \right. \right. \\
 & \quad \quad \left. \left. \text{ratio} = \frac{rx}{ry}, k \cdot ry == y3\right\}, \{rx, ry\}, \{h, k, x3\}\right] \\
 & \quad \left\{ \left\{ rx \rightarrow \frac{1}{2 \text{ratio} (y1-y2)^2} \left(-((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) (y1+y2-2y3) + \right. \right. \right. \\
 & \quad \quad \left. \left. 2 (x1-x2) \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} \right), \right. \\
 & \quad \left. ry \rightarrow \frac{1}{2 \text{ratio}^2 (y1-y2)^2} \left(-((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) (y1+y2-2y3) + \right. \right. \\
 & \quad \quad \left. \left. 2 (x1-x2) \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} \right) \right\}, \\
 & \quad \left\{ rx \rightarrow -\frac{1}{2 \text{ratio} (y1-y2)^2} \left(((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) (y1+y2-2y3) + \right. \right. \\
 & \quad \quad \left. \left. 2 (x1-x2) \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} \right), \right. \\
 & \quad \left. ry \rightarrow -\frac{1}{2 \text{ratio}^2 (y1-y2)^2} \left(((x1-x2)^2 + \text{ratio}^2 (y1-y2)^2) (y1+y2-2y3) + \right. \right. \\
 & \quad \quad \left. \left. 2 (x1-x2) \sqrt{(x1-x2)^2 + \text{ratio}^2 (y1-y2)^2} \sqrt{(y1-y3)(y2-y3)} \right) \right\} \} \\
 & \quad \left. \right\}
 \end{aligned}$$

For all four subcases if an ellipse exists that is tangent to the edge, $r_{\text{ellipse}}(\text{ratio}, e^-, e, e^+)$ is set to its horizontal semi-axis, rx . Otherwise, the ellipses, ellipse_1 and ellipse_2 , defined by the two vertices and each edge endpoint, are constructed as in case A. The ellipse that does not contain any internal portion of the edge is then selected. See figure 7.6 where the two ellipses intersecting e^-, e and either endpoint of e^+ is shown. Only the ellipse with the double line does not include the edge and is therefore selected.

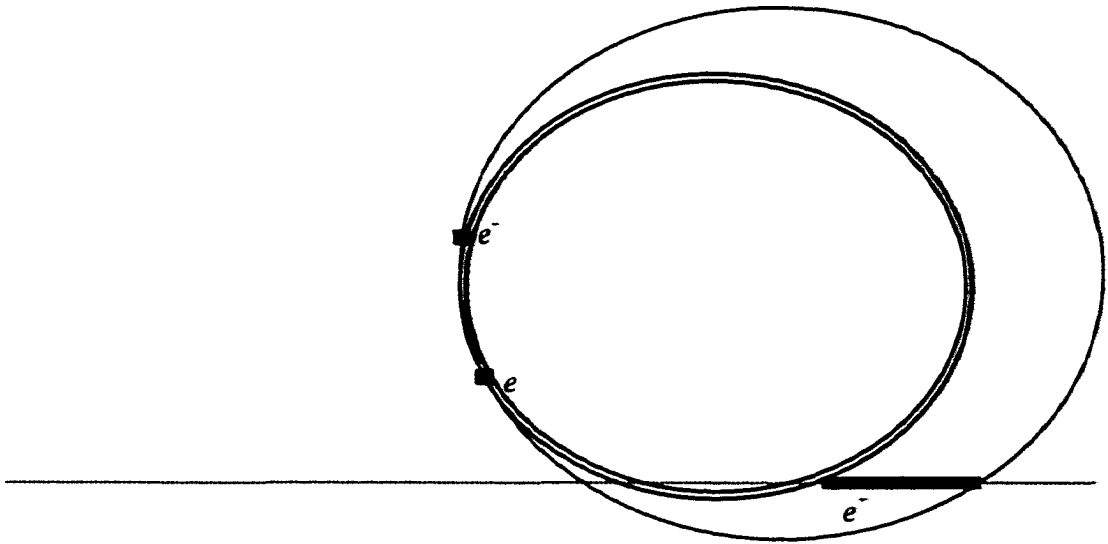


Figure 7.6. When the ellipse tangent to the edge does not exist, the ellipse intersecting an endpoint of the edge without including any portion of the edge is selected. This ellipse chosen is shown as a doubled ellipse.

Case C: One Vertex and Two Edges: (• — —)

We refer to the vertex as v and the edges as $edge_2$ and $edge_3$. The following three orientation of the edges are considered:

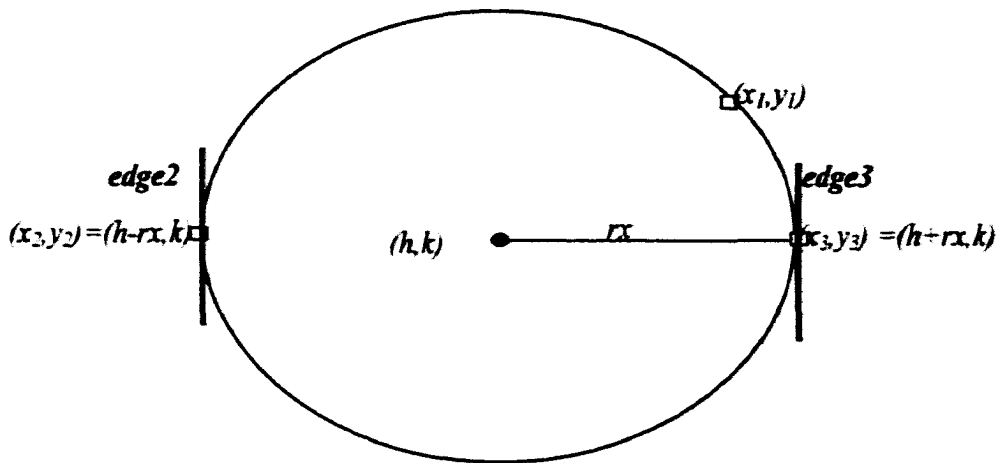
- 1) $edge_2$ and $edge_3$ are both vertical
- 2) $edge_2$ and $edge_3$ are both horizontal
- 3) $edge_2$ is vertical and $edge_3$ is horizontal.

As with case B, we solve for the ellipse that is tangent to the edges' linear extensions. If that ellipse is tangent to both edges, $r_{\text{ellipse}}(e^-, e, e^+)$ is the horizontal semi-

axis of that ellipse. If that ellipse is not tangent to one of the edges, one of the edge's endpoint is selected in lieu of the edge, as with case B. The details are discussed below.

At most two ellipses exist that intersect a vertex and are tangent to two edges. There is a maximum of two solutions for the ellipse semi-axes, r_x and r_y . Of these two solutions only one solution will exist that will intersect e^- , e and e^+ in order counterclockwise.

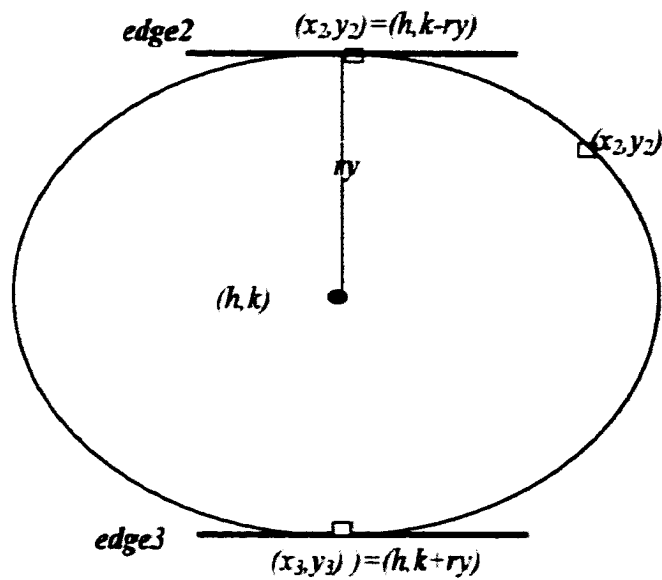
Solving for the tangent points along $edge_2$ and $edge_3$ and the radius, r_x , we solve the following set of simultaneous equations:



Case 1: $edge_2$ and $edge_3$ are both vertical.

$$\begin{aligned} & \text{Union} [\\ & \quad \text{FullSimplify} [\\ & \quad \quad \text{Solve} [\left\{ \frac{(x1-h)^2}{r_x^2} + \frac{(y1-k)^2}{r_y^2} = 1, \frac{(x2-h)^2}{r_x^2} + \frac{(y2-k)^2}{r_y^2} = 1, \frac{(x3-h)^2}{r_x^2} + \frac{(y3-k)^2}{r_y^2} = 1, \right. \\ & \quad \quad \left. \text{ratio} = \frac{r_x}{r_y}, h - r_x == x2, h + r_x == x3 \right\}, \{r_x, r_y\}, \{h, k, y2, y3\}]]] \\ & \left\{ \left\{ r_y \rightarrow \frac{-x2 + x3}{2 \text{ ratio}}, r_x \rightarrow \frac{1}{2} (-x2 + x3) \right\} \right\} \end{aligned}$$

Case 2: $edge_2$ and $edge_3$ are both horizontal.



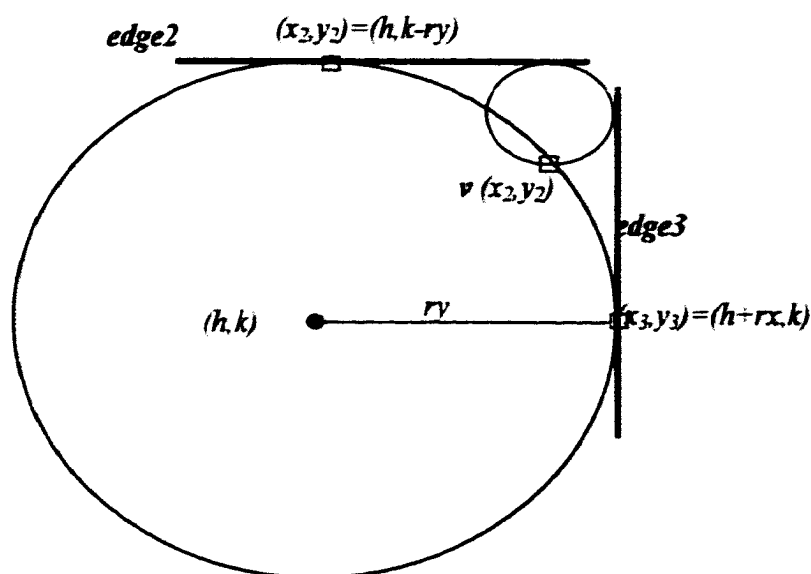
Union[

FullSimplify[

$$\text{Solve}\left[\left\{\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1, \right.\right. \\ \left.\left.\text{ratio} = \frac{rx}{ry}, k - ry == y2, k + ry == y3\right\}, \{rx, ry\}, \{h, k, x2, x3\}\right]$$

$$\left\{\left\{rx \rightarrow -\frac{1}{2} \text{ratio} (y2 - y3), ry \rightarrow \frac{1}{2} (-y2 + y3)\right\}\right\}$$

Case 3: $edge_2$ is vertical and $edge_3$ is horizontal.



In the above figure v lies below $edge_2$ and to the left of $edge_3$. Vertex v may be in one of four possible orientations with respect to $edge_2$ and $edge_3$. v may lie above or below $edge_2$ and either to the right or left of $edge_3$. Each of these subcases is considered below (cases a – d). The solutions to each of the simultaneous equations yields two solutions representing the two ellipses that intersect the three elements once. The solution selected is the ellipse that intersects the elements in order counterclockwise.

Case a: v is above and left of $edge_2$ and $edge_3$, respectively

```

union[
  FullSimplify[
    solve[ {  $\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1,$ 
      ratio =  $\frac{rx}{ry}, k - ry == y2, h - rx == x3$  }, {rx, ry}, {h, k, x2, y3} ] ] ]

```

$$\left\{ \left\{ ry \rightarrow \frac{x1 - x3 + ratio \cdot y1 - \sqrt{2} \sqrt{ratio} \sqrt{x1 - x3} \sqrt{y1 - y2} - ratio \cdot y2}{ratio}, \right. \right. \\ rx \rightarrow x1 - x3 + ratio \cdot y1 - \sqrt{2} \sqrt{ratio} \sqrt{x1 - x3} \sqrt{y1 - y2} - ratio \cdot y2 \left. \right\}, \\ \left\{ ry \rightarrow \frac{x1 - x3 + ratio \cdot y1 + \sqrt{2} \sqrt{ratio} \sqrt{x1 - x3} \sqrt{y1 - y2} - ratio \cdot y2}{ratio}, \right. \\ \left. rx \rightarrow x1 - x3 + ratio \cdot y1 + \sqrt{2} \sqrt{ratio} \sqrt{x1 - x3} \sqrt{y1 - y2} - ratio \cdot y2 \right\} \left. \right\}$$

Case b: v is above and right of edge2 and edge3, respectively

$$\text{Union} \left[\right. \\ \text{FullSimplify} \left[\right. \\ \text{Solve} \left[\left\{ \frac{(x1 - h)^2}{rx^2} + \frac{(y1 - k)^2}{ry^2} = 1, \frac{(x2 - h)^2}{rx^2} + \frac{(y2 - k)^2}{ry^2} = 1, \frac{(x3 - h)^2}{rx^2} + \frac{(y3 - k)^2}{ry^2} = 1, \right. \right. \\ \left. \left. ratio = \frac{rx}{ry}, k - ry == y2, h + rx == x3 \right\}, \{rx, ry\}, \{h, k, x2, y3\} \right] \left. \right] \\ \left\{ \left\{ ry \rightarrow \frac{-x1 + x3 + ratio \cdot y1 - ratio \cdot y2 - \sqrt{2} \sqrt{ratio} (x1 - x3) (-y1 + y2)}{ratio}, \right. \right. \\ rx \rightarrow -x1 + x3 + ratio \cdot y1 - ratio \cdot y2 - \sqrt{2} \sqrt{ratio} (x1 - x3) (-y1 + y2) \left. \right\}, \\ \left\{ ry \rightarrow \frac{-x1 + x3 + ratio \cdot y1 - ratio \cdot y2 + \sqrt{2} \sqrt{ratio} (x1 - x3) (-y1 + y2)}{ratio}, \right. \\ \left. rx \rightarrow -x1 + x3 + ratio \cdot y1 - ratio \cdot y2 + \sqrt{2} \sqrt{ratio} (x1 - x3) (-y1 + y2) \right\} \left. \right\}$$

Case c: v is below and left of edge2 and edge3, respectively

$$\text{Union} \left[\right. \\ \text{FullSimplify} \left[\right. \\ \text{Solve} \left[\left\{ \frac{(x1 - h)^2}{rx^2} + \frac{(y1 - k)^2}{ry^2} = 1, \frac{(x2 - h)^2}{rx^2} + \frac{(y2 - k)^2}{ry^2} = 1, \frac{(x3 - h)^2}{rx^2} + \frac{(y3 - k)^2}{ry^2} = 1, \right. \right. \\ \left. \left. ratio = \frac{rx}{ry}, k + ry == y2, h - rx == x3 \right\}, \{rx, ry\}, \{h, k, x2, y3\} \right] \left. \right] \\ \left\{ \left\{ ry \rightarrow \frac{x1 - x3 - ratio \cdot y1 + ratio \cdot y2 - \sqrt{2} \sqrt{ratio} (x1 - x3) (-y1 + y2)}{ratio}, \right. \right. \\ rx \rightarrow x1 - x3 - ratio \cdot y1 + ratio \cdot y2 - \sqrt{2} \sqrt{ratio} (x1 - x3) (-y1 + y2) \left. \right\}, \\ \left\{ ry \rightarrow \frac{x1 - x3 - ratio \cdot y1 + ratio \cdot y2 + \sqrt{2} \sqrt{ratio} (x1 - x3) (-y1 + y2)}{ratio}, \right. \\ \left. rx \rightarrow x1 - x3 - ratio \cdot y1 + ratio \cdot y2 + \sqrt{2} \sqrt{ratio} (x1 - x3) (-y1 + y2) \right\} \left. \right\}$$

Case d: v is below and right of $edge_2$ and $edge_3$, respectively

```

Union[
  FullSimplify[
    Solve[ {  $\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1,$ 
      ratio =  $\frac{rx}{ry}, k + ry == y2, h + rx == x3$  }, {rx, ry}, {h, k, x2, y3} ] ] ]
  { {ry -> -  $\frac{x1 - x3 + ratio y1 + \sqrt{2} \sqrt{ratio} \sqrt{x1 - x3} \sqrt{y1 - y2} - ratio y2}{ratio},$ 
    rx -> -x1 + x3 - ratio y1 -  $\sqrt{2} \sqrt{ratio} \sqrt{x1 - x3} \sqrt{y1 - y2} + ratio y2$  },
    {ry ->  $\frac{-x1 + x3 - ratio y1 + \sqrt{2} \sqrt{ratio} \sqrt{x1 - x3} \sqrt{y1 - y2} + ratio y2}{ratio},$ 
    rx -> -x1 + x3 - ratio y1 +  $\sqrt{2} \sqrt{ratio} \sqrt{x1 - x3} \sqrt{y1 - y2} + ratio y2$  } }

```

For subcases 3a-d, as with case B, it is possible that the ellipse determined by the above solutions is tangent to the edge's linear extension without actually being tangent to the edge. Thus, if $(x2, y2)$ and $(x3, y3)$ are on edges e_2 and e_3 , respectively, $r_{\text{ellipse}}(\text{ratio}, e^-, e, e^+) = rx$. However, if either tangent point is not on its respective edge we may then solve for r_{ellipse} by finding the ellipses $ellipse_1$ and $ellipse_2$ which are the ellipses defined by v , $edge_2$ and an endpoint of $edge_3$, respectively, and ellipses $ellipse_3$ and $ellipse_4$ which are defined by v , $edge_3$ and an endpoint of $edge_2$, respectively. These ellipses are computed as set forth in case B. Of the four ellipses' horizontal semi-axis rx the rx value of the ellipse that intersects $edge_2$ and $edge_3$ exactly once and does not include any part of either $edge_2$ or $edge_3$, is selected (figure 7.7).

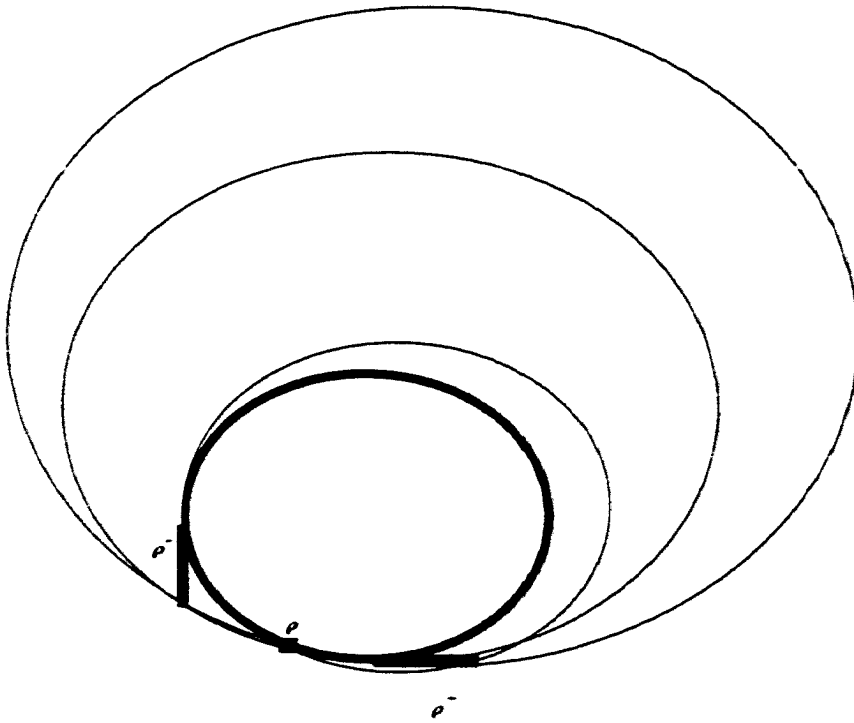


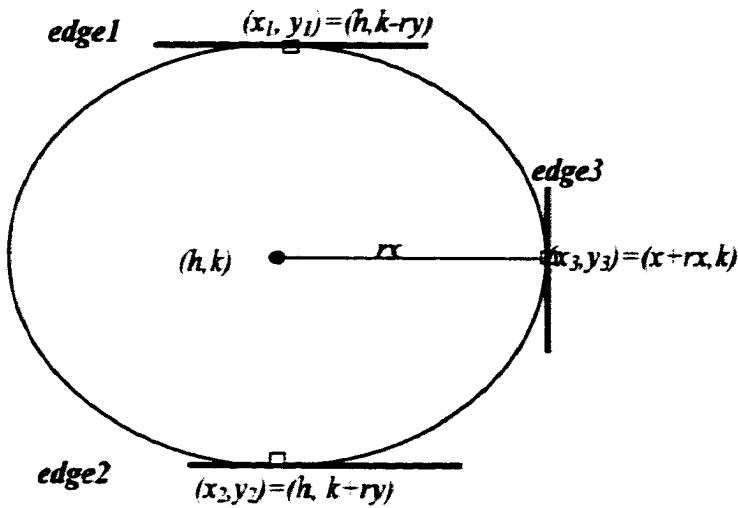
Figure 7.7. When no ellipse exists that is tangent to both edges, four ellipses are determined that intersect the vertex and an endpoint of the two edges. The ellipse that does not include any portion of either edge (darkened) is selected.

Case D: Three Edges: (— — —)

The three edges, $edge_1$, $edge_2$ and $edge_3$ may have one of two combinations of their orientations: either two edges are vertical and one is horizontal or two edge's are horizontal and one is vertical. With no loss in generality these cases may be expressed as:

- 1) $edge_1$ and $edge_2$ are vertical and $edge_3$ is horizontal
- 2) $edge_1$ and $edge_2$ are horizontal and $edge_3$ is vertical.

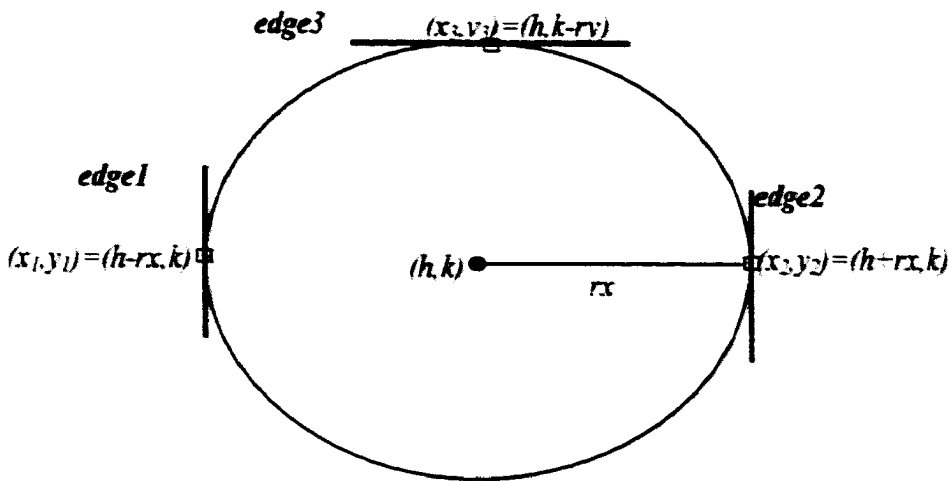
A single solution exists for both cases. The simultaneous equations and solutions are as follows.



Case 1: e_1 and e_2 are both horizontal.

$$\begin{aligned} & \text{Union}[\\ & \quad \text{FullSimplify}[\\ & \quad \text{solve}\left[\left\{\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1, \right. \right. \\ & \quad \left. \left. \text{ratio} = \frac{rx}{ry}, h - rx == x1, h + rx == x2, (k - ry - y3)(k + ry - y3) = 0\right\}, \{rx, ry\}, \{h, k, y1, y2, x3\}\right] \\ & \quad \left.\left\{\left\{ry \rightarrow \frac{-x1 + x2}{2 \text{ratio}}, rx \rightarrow \frac{1}{2}(-x1 + x2)\right\}\right\} \right] \end{aligned}$$

Case 2: e_1 and e_2 are both vertical.



```

Union[
  FullSimplify[
    Solve[ {  $\frac{(x1-h)^2}{rx^2} + \frac{(y1-k)^2}{ry^2} = 1, \frac{(x2-h)^2}{rx^2} + \frac{(y2-k)^2}{ry^2} = 1, \frac{(x3-h)^2}{rx^2} + \frac{(y3-k)^2}{ry^2} = 1,$ 
      ratio =  $\frac{rx}{ry}, k - ry == y1, k \cdot ry == y2, (h - rx - x3) (h \cdot rx - x3) = 0 \}, \{rx, ry\}, \{h, k, x1, x2, y3\} ] ] ]
  { {  $rx \rightarrow -\frac{1}{2} \text{ratio} (y1 - y2), ry \rightarrow \frac{1}{2} (-y1 + y2)$  } }$ 
```

As with the previous case C, the tangent points of the ellipse found on the edge's linear extension does not have to be located on the edge. If such a case occurs six ellipses are determined:

ellipse₁ and *ellipse₂*, the ellipses defined by an endpoint of *e₁* and edges *edge₂* and *edge₃* (as indicated in the figure),

ellipse₃ and *ellipse₄*, the ellipses defined by an endpoint of *edge₂* and edges *edge₁* and *edge₃*, and

ellipse₅ and *ellipse₆*, defined by an endpoint of *edge₃* and edges *edge₁* and *edge₂*.

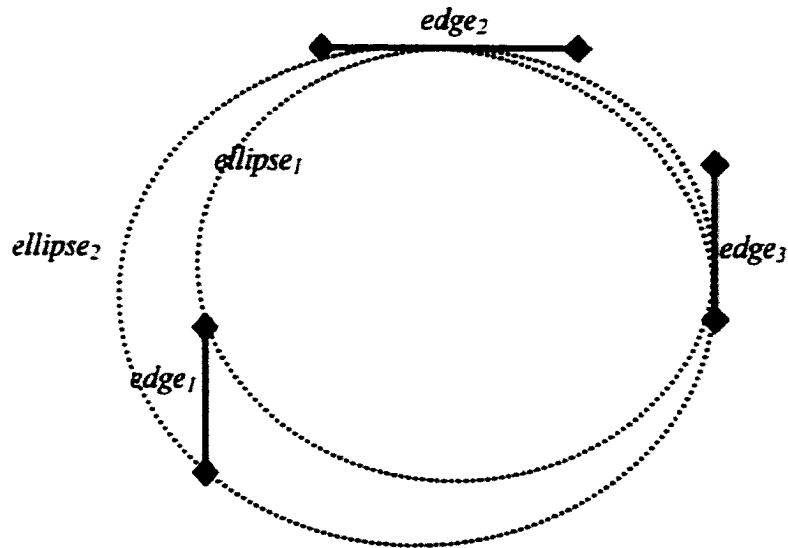


Figure 7.8. In the event the circle as determined by the edge's linear extensions is not tangent to the all edges six ellipses are constructed. Each such ellipse is tangent to two edges and intersects the third edge's endpoints. Two such ellipses, *ellipse₁* and *ellipse₂*, are shown here.

These ellipses are computed as set forth in case C. Of these six ellipses, the ellipse that intersects e_1 , e_2 and e_3 exactly once and does not include any point of edges e_1 , e_2 and e_3 is selected. See figure 7.8.

In this section the methodology for determining the conditional critical ranges of elements along the passing area was defined. Based on analytical geometry, four major cases (0,1,2 or 3 elements are edges) and eight subcases (due to vertical and horizontal relative positions of elements) were presented and in each case the conditional critical ranges of the elements were determined. As with circles (section 4) these conditional critical ranges determine the r_x semi-axis value (and hence r_y) at which the passing area experiences critical changes. This information immediately yields the digital curve's critical sets and domain. These observations and calculation form the basis of the algorithm that determines the domain of an elliptical curve where r_x may change.

All the phases of the algorithm discussed in section 4.3 for circular digital curves are identical for the elliptical digital curve except for the determination of the critical ranges as set forth in this section. Thus, the *only* distinction in the elliptical curve algorithm from the circular case is that the values for the critical ranges for each element are based on $r_{\text{ellipse}}(\text{ratio}, e^-, e, e^+)$ instead of $r_{\text{circle}}(e^-, e, e^+)$. In fact, in the implementation of the algorithm class Circle, the encapsulating class of r_{circle} , is derived from class Ellipse with $\text{ratio} = 1$. In the next section the shape of the elliptical curve's domain is discussed.

7.2. The Shape of the Elliptical Domain

The domain of an elliptical digital curve with fixed eccentricity is the intersection of eccentric cones centered at points $p \in S_P$ and eccentric cone complements centered at $q \in S_Q$ where S_P and S_Q are points along the inner and outer boundaries of the digital curve, respectively. This three dimensional domain is parameterized by ellipse center, (x, y) and r_x , the horizontal semi axis (figure 7.9). The shape of the projection of the domain onto the two dimensional x, y -plane is now discussed.

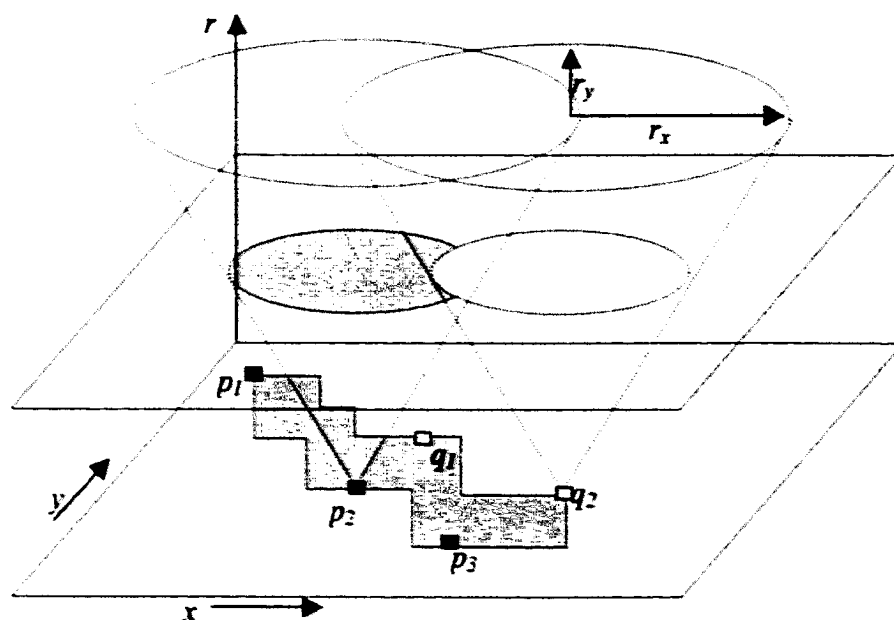


Figure 7.9. The intersection of eccentric cones and eccentric cone complements centered at points on the inside (p_i) and outside (q_i) of the digital curve yield the complete circular arc domain of the digital curve.

The two dimensional domain represents the set of ellipse centers such that for each point c in the domain there exists an ellipse, with c as the ellipse's center, of which a portion digitizes to the digital elliptical curve. Each ellipse whose center is in the domain will separate sets S_P and S_Q . Thus, with an elliptical digital curve the task of identifying the passing area of an elliptical digital curve may be reformulated as the elliptical separability problem of the two sets, S_P and S_Q .

The following are defined (which parallel definitions of circles, chapter 4):

Definition: Given a standard ellipse e with horizontal and vertical semi-axes ratio, *ratio*, the *skewed Euclidean distance*, $d_{se}(p_1, p_2)$, between two points, $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ equals $\sqrt{(x_1 - x_2)^2 + ratio^2(y_1 - y_2)^2}$.

Clearly, when *ratio* = 1, representing a circle, the skewed Euclidean distance between two points reduces to the Euclidian distance, $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

The skewed Euclidean distance between an ellipse center c and two points, p_i and p_j , may be related to each other in the following manner (as shown in figures 7.10-7.11):

$$d_{se}(c, p_i) = d_{se}(c, p_j) :$$

Some ellipse centered at c will intersect p_i and p_j (figure 7.10)

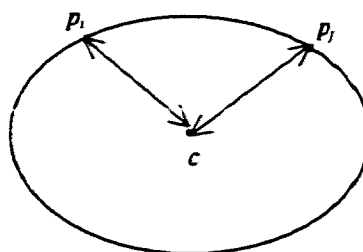


Figure 7.10. The relative skewed Euclidean distances of two points p_i and p_j to the ellipse center c determines whether an ellipse with the given semi-axes ratio exists that will intersect both.

$$d_{se}(c, p_i) < d_{se}(c, p_j) :$$

Some ellipse centered at c will include p_i and exclude p_j (figure 7.11)

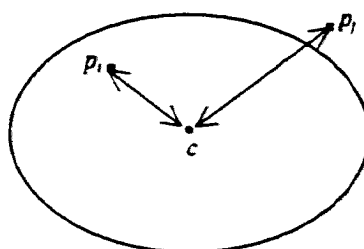


Figure 7.11. The relative skewed Euclidean distances of two points p_i and p_j to the ellipse center, c , determines whether an ellipse with the given semi-axes ratio exists that will include one of the points (p_i and p_j).

In the literature [Worring, 1995], Voronoi polygons are used to compute the domain centers for pre-image circles that can digitize to a specific digital curve. For a set

of points S in R^2 , Voronoi polygons lie along the perpendicular bisectors of points in S . Bisectors are determined by Euclidean distance. In the following discussion the Voronoi polygons are generalized to the skewed Voronoi polygons using skewed Euclidean distance as the metric for determining the bisector. When the ellipse's semi-axes ratio is unity, the skewed Voronoi polygons reduce to the simple Voronoi polygon.

For P, Q , two point sets in R^2 , where $p \in P, q \in Q$ and $t \in R^2$ we have:

Definition: Closest Point Skewed Voronoi Polygon:

$$V_s(t, Q) = \{z \in R^2 \mid \forall q \in Q : d_s(z, t) \leq d_s(z, q)\}$$

Definition: Furthest Point Skewed Voronoi Polygon:

$$V_f(P, t) = \{z \in R^2 \mid \forall p \in P : d_s(z, p) \leq d_s(z, t)\}$$

Definition: Generalized Skewed Voronoi Polygon

$$V_s(P, Q) = \{z \in R^2 \mid \forall p \in P, \forall q \in Q : d_s(z, p) \leq d_s(z, q)\}$$

Simple generalized Voronoi polygons represent the region on the discrete grid where *some* circle will separate S_P and S_Q . This result is guaranteed since the generalized Voronoi polygon includes all the grid points that are closer to all points in S_P than to *any* point in S_Q . A circle centered within that region will clearly separate S_P and S_Q with a radial value between the maximum distance of a point in S_P and the minimum distance of a point in S_Q .

Similarly, the generalized skewed Voronoi polygons only contain the points that are *closer* to S_p than to S_q where *closeness* is defined by the skewed Euclidean distance metric. Therefore, each point in the region can be the center of some ellipse with the given semi-axes ratio that separates S_p and S_q . Thus, if S_p and S_q are the inner and outer boundaries of a digital elliptical curve, $V_s(S_p, S_q)$ will contain the domain of the elliptical curve centers.

The shape of the generalized skewed Voronoi polygon is now addressed. Based on its definition, the region is clearly bound by the locus of points where $d_{se}(x, p) = d_{se}(x, q)$, $p \in S_p$, $q \in S_q$. In computing the domain of a digital curve, the set S_p contains the elements on the inner boundary and the set S_q contains the elements on the outer boundary. As established in chapter 6, since p is on the inner boundary it can only be a vertex. However, q , which is found on the outer boundary, may either be a vertex or a line segment. There are thus two more cases to consider depending on whether q is a vertex or edge.

Consider the case where q is a vertex. Let p and q be vertices on the upper and lower pixel boundaries. The region where $d_{se}(x, p) \leq d_{se}(x, q)$ is bounded by the bisector

$$\text{of } \overline{pq} \text{ with slope } m = \frac{x_1 - x_2}{ratio^2(y_2 - y_1)} \text{ and } y\text{-intercept of } \frac{1}{2} \left(y_1 + \frac{(x_1 - x_2)(x_1 + x_2)}{ratio^2(y_1 - y_2)} + y_2 \right).$$

This result is found when solving for the line that intersects the ellipse centers of two different ellipses with same eccentricity that intersect two points, as shown in figure 7.12.

This skewed (i.e. non-perpendicular) bisector is the perpendicular bisector with y coordinate scaled by $ratio^{-2}$, as is evidenced by the value of the slope m .

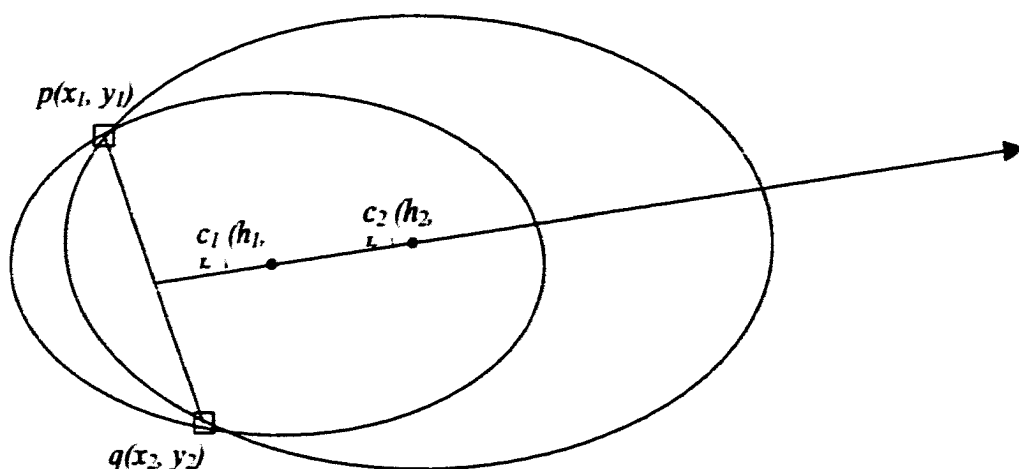


Figure 7.12. The center of standard ellipses with semi-axes ratio *ratio* that intersect two points (x_1, y_1) and (x_2, y_2) must intersect the skewed bisector of (x_1, y_1) and (x_2, y_2) .

An ellipse centered at (h_1, k_1) with horizontal and vertical semi-axes rx_1 and ry_1 , respectively, that intersects two points (x_1, y_1) and (x_2, y_2) must satisfy the equations

$$\frac{(x_1 - h_1)^2}{rx_1^2} + \frac{(y_1 - k_1)^2}{ry_1^2} = 1 \quad \text{and} \quad \frac{(x_2 - h_1)^2}{rx_1^2} + \frac{(y_2 - k_1)^2}{ry_1^2} = 1. \quad \text{Similarly another ellipse}$$

centered at (h_2, k_2) with horizontal and vertical semi-axes rx_2 and ry_2 , respectively, that

also intersects (x_1, y_1) and (x_2, y_2) must solve $\frac{(x_1 - h_2)^2}{rx_2^2} + \frac{(y_1 - k_2)^2}{ry_2^2} = 1$ and

$$\frac{(x_2 - h_2)^2}{rx_2^2} + \frac{(y_2 - k_2)^2}{ry_2^2} = 1. \quad \text{Since the two ellipses have the same semi-axes ratio their}$$

semi-axes are related by some constant a such that $rx_1 = a rx_2$ and $ry_1 = a ry_2$. The line that intersects both ellipse centers (h_1, k_1) and (h_2, k_2) must satisfy $k_1 = m h_1 + b$ and $k_2 = m h_2 + b$ where m and b are the slope and y-intercept, respectively.

$$\begin{aligned}
 & \text{Union[} \\
 & \quad \text{FullSimplify[} \\
 & \quad \quad \text{Solve}\left[\left\{\frac{(x1-h1)^2}{rx1^2} + \frac{(y1-k1)^2}{ry1^2} = 1, \frac{(x2-h1)^2}{rx1^2} + \frac{(y2-k1)^2}{ry1^2} = 1, \frac{(x1-h2)^2}{rx2^2} + \frac{(y1-k2)^2}{ry2^2} = 1, \right. \right. \\
 & \quad \quad \left. \left. \frac{(x2-h2)^2}{rx2^2} + \frac{(y2-k2)^2}{ry2^2} = 1, rx1 = a \, rx2, ry1 = a \, ry2, k1 = m \, h1 + b, k2 = m \, h2 + b\right\}, \right. \\
 & \quad \quad \left. (m, b, h1, k1, h2, k2)\right] \left. \right] \\
 & \quad \left\{b \rightarrow \frac{1}{2} \left\{y1 + \frac{ry2^2 (x1 - x2) (x1 + x2)}{rx2^2 (y1 - y2)} + y2\right\}, m \rightarrow \frac{ry2^2 (x1 - x2)}{rx2^2 (-y1 + y2)}\right\}
 \end{aligned}$$

When p is a vertex and q an edge the region is bounded by the parabola defined by p as the focus and q as the directrix with y coordinate scaled by $ratio^{-2}$. This is found by solving for the ellipse centers that intersect a point $P(0,n)$ and is tangent to a line $\overline{AB}$ at point $L(x_o,m)$, as shown in figure 7.13.

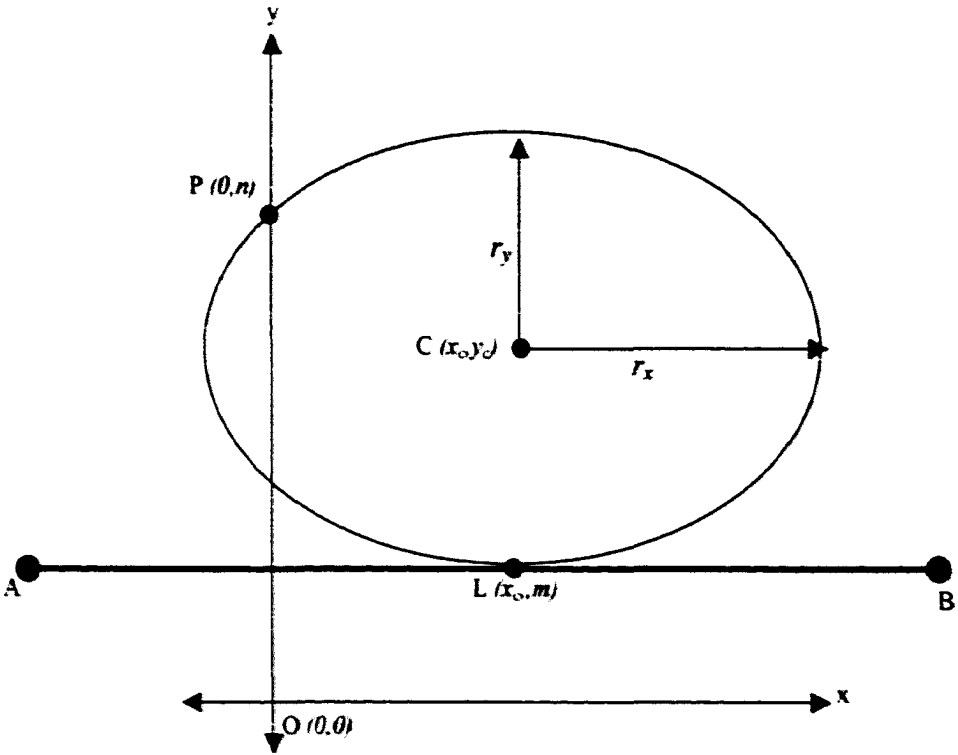


Figure 7.13. An ellipse centered at (x_o, y_c) that intersects point $P(0,n)$ and is tangent to line $\overline{AB}$ at point $L(x_o,m)$.

Solving for the y_c in terms of x_c , m , n and $ratio$ we have

$$y_c = \frac{1}{2} \left(-\frac{x_c^2}{(m-n)ratio^2} + m + n \right) \text{ as shown by the following Mathematica results.}$$

$$\begin{aligned} & \text{FullSimplify[} \\ & \quad \text{Union[Solve[{\frac{(0-x_c)^2}{r_x^2} + \frac{(n-y_c)^2}{r_y^2} = 1, \frac{(x_c-x_c)^2}{r_x^2} + \frac{(m-y_c)^2}{r_y^2} = 1, ratio = \frac{r_x}{r_y}},} \\ & \quad \{y_c\}, \{r_x, r_y\}]]] \\ & \{ \{y_c \rightarrow \frac{1}{2} \left(m + n - \frac{x_c^2}{(m-n)ratio^2} \right) \} \} \end{aligned}$$

It is interesting to note that this curve is a parabola skewed by $1 / ratio^2$. Where $ratio = 1$ this is the simple unskewed parabola. The locus of circle centers for ellipses of with semi-axes ratios of 0.5, 1 and 2 are shown in figure 7.14.

The simple parabola defined by a point p (i.e. focus) and $\overline{AB}$ (i.e. directrix) are shown in figure 7.14. Figure 7.15 shows the domain as the intersection of skewed bisectors and parabolas.

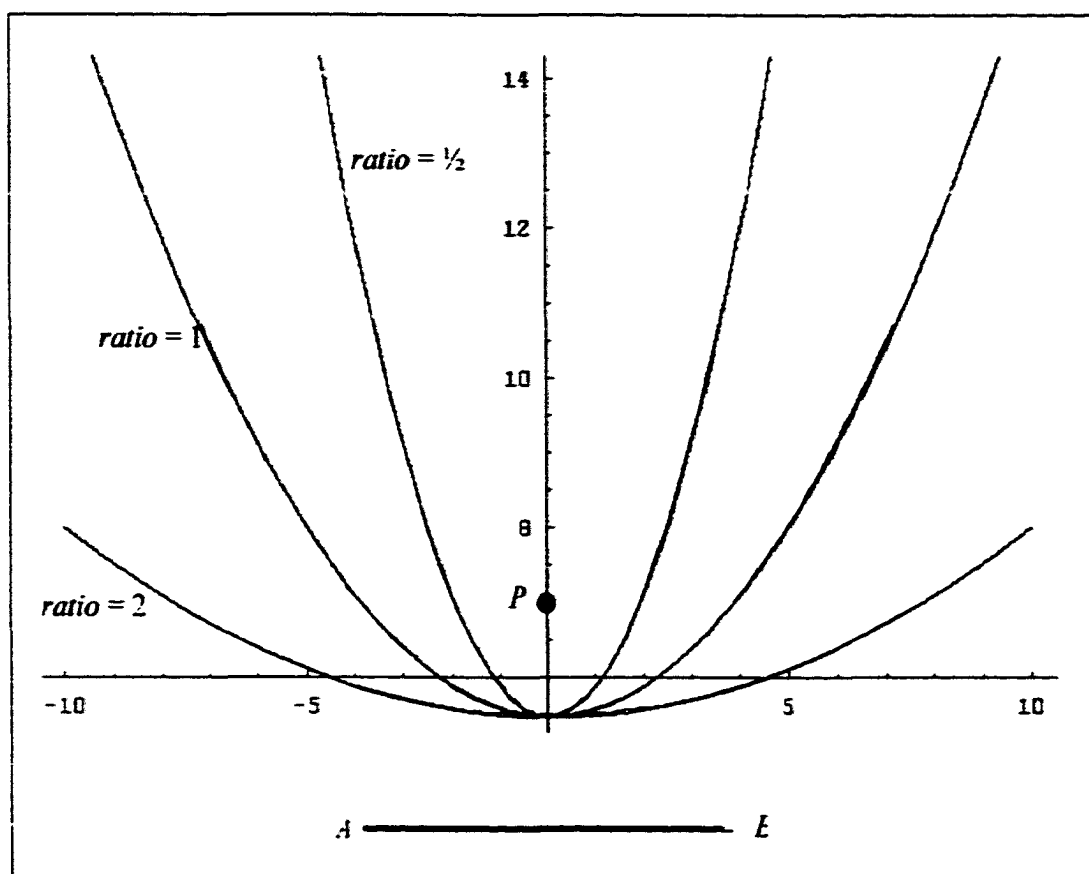


Figure 7.14. The parabolas shown represent the locus of ellipse centers where the ellipse intersects point P and is tangent to line $\overline{AB}$. The parabolas represent the locus of ellipse centers where the semi-axes ratios $\frac{1}{2}$, 1 and 2.

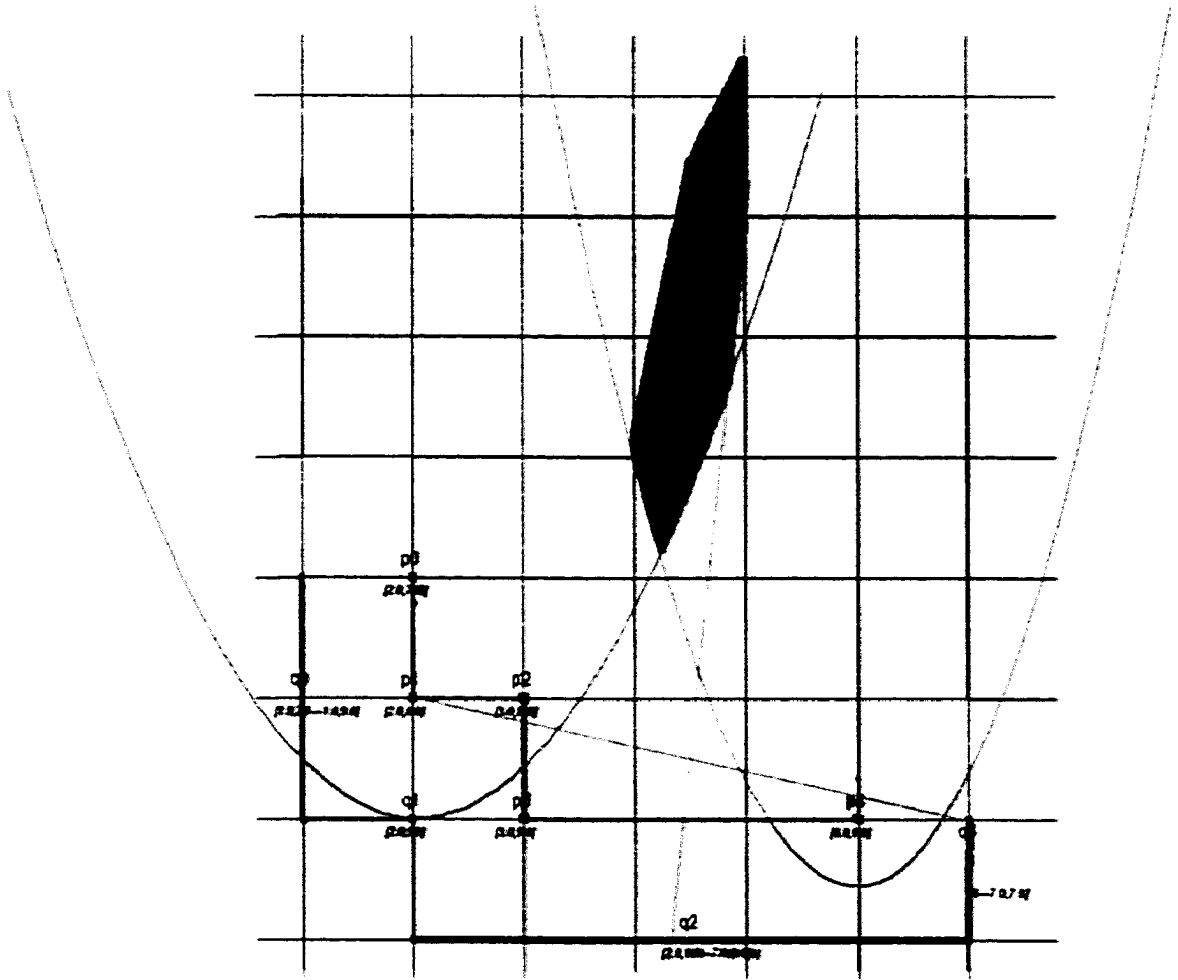


Figure 7.15. The domain where $ratio = 0.75$ is bounded by skewed bisectors and parabolas. Three additional skewed bisectors bound the domain and are not shown.

7.3 General Ellipses

Earlier we determined the domain of a digital circular curve under all radii (section 4) and the domain of a digital standard elliptical curve with fixed eccentricity (section 7.1). We now discuss the issues that are encountered in determining the domain of a standard ellipse for all eccentricities.

With a circular (standard elliptical) digital curve domain the domain is in 3-space where x and y represent the circle (ellipse) center and z is the radius (horizontal semi-axis). As discussed in section 4.2, as the radius increases the critical points along the passing area perimeter change yielding new critical sets. We may this partition the 1 dimensional radius-space such that each region in radius -space contains either no critical set (corresponding to a non circular digital curve) or a unique critical set. This is shown in figure 7.16 which is adapted from figure 4.10.

	CS	$\langle CS \rangle_{min}$	$\langle CS \rangle_{max}$
1)	p_0, p_4, q_1	1.80	1.90
2)	p_0, p_1, p_4, q_1	1.90	2.00
3)	p_0, p_1, p_4, q_1, q_2	2.00	2.24
4)	$p_0, p_1, p_3, p_4, q_1, q_2$	2.24	2.25
5)	$p_0, p_1, p_3, p_4, q_1, q_2, q_3$	2.25	2.30
6)	$p_1, p_3, p_4, q_1, q_2, q_3$	2.30	2.50
7)	$p_1, p_3, p_4, q_0, q_1, q_3$	2.50	2.92
8)	p_3, p_4, p_0, q_1, q_3	2.92	3.01
9)	p_3, p_4, q_0, q_3	3.01	3.16
10)	p_3, q_0, q_3	3.16	3.81

Figure 7.16. r -space is partitioned into regions such that each region has at most one associated critical set

The algorithm presented begins with the minimum radius that supports a circular interpretation to the digital curve and iterates determining each successive critical set, as discussed in section 4.3. This algorithm was adapted to standard elliptical digital curves as discussed in section 7.1.

The algorithm is dependent on its ability to determine the radial values where critical changes occur. With a circular or standard elliptical digital curve those critical changes occur at specific radius values such that all values of r that are less than the value where a critical change occurs are associated with the current critical set. For example, in figure 7.16 critical set #2 is associated with r value 1.9. When the next critical change is determined to be at $r = 2.0$ we may correctly conclude that critical set #2 is valid in the range 1.9 to 2.0.

However, when determining the domain of a standard elliptical curve with unknown eccentricity the domain is in 4-space where x and y are the ellipse center and z and z' are r_x and r_y (the horizontal and vertical semi-axes), respectively. Thus, whereas earlier a partition of 1-space defined the domain, now the 2-space plane corresponding to r_x and r_y must be partitioned. More significantly, the regions in the plane associated with each critical set are not linear as shown in figure 7.17. In the figure the projection onto the z and z' plane are shown. The shape of the domain's projection is clearly nonlinear.

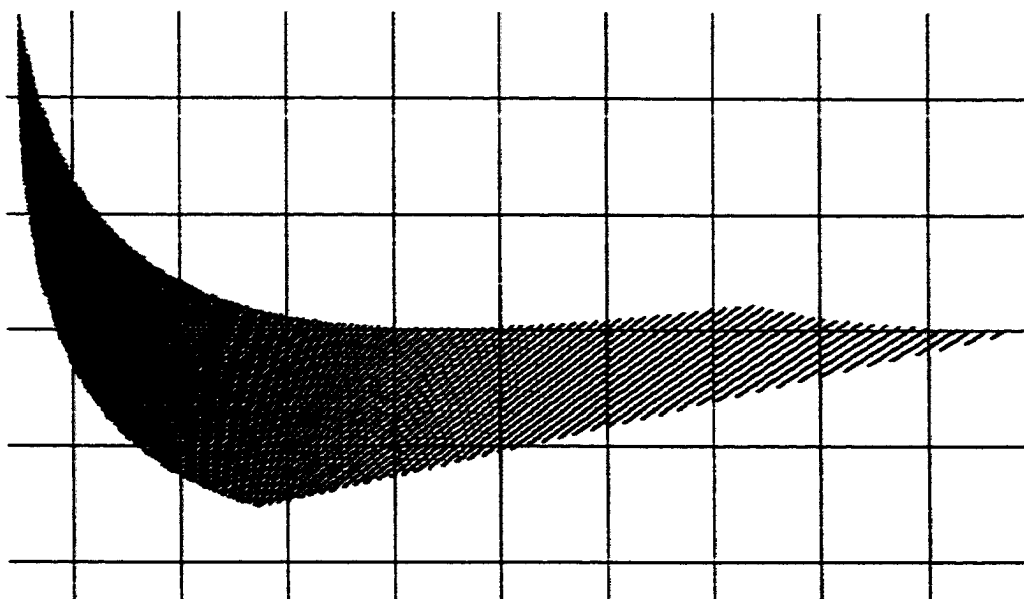


Figure 7.17. The projection of a digital elliptical curve's domain onto the (r_x, r_y) plane.

Thus, to adapt the algorithm discussed earlier to handle ellipses with unknown eccentricity two computations are required:

- 1) Determining the (r_x, r_y) values where critical changes occur
- 2) Determining the shape of the curve that separates every two critical regions in (r_x, r_y) space.

The first computation is complex since it includes computing multiple critical changes for each current critical set value. Since the critical sets partition the plane each region's boundary point may bound multiple critical sets. See figure 7.18 where a boundary point of some hypothetical critical set is shown in black. That point may lie on the boundary (shown in white) of many critical regions. It is unclear how to generalize

the conditional critical range computation discussed earlier to handle this additional complexity introduced with unknown eccentricity. The second component, determining the shape of the curve that separates two critical regions is an area that, too, must be further investigated.

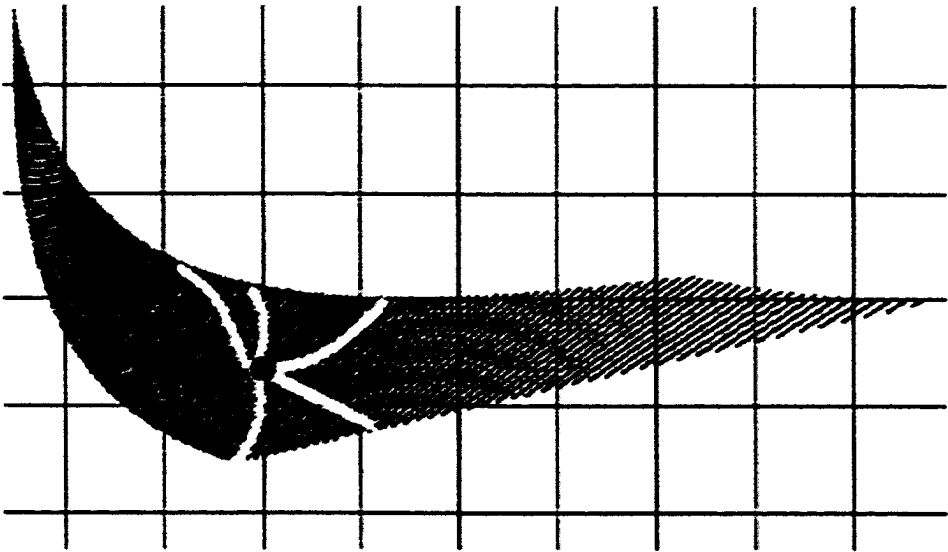


Figure 7.18. Critical regions that partition the semi-axis (r_x, r_y) space.

Chapter Eight:

The Passing Area Solver Software (PASS) Application

Abstract

The Passing Area Solver Software (PASS) application was developed to facilitate the computation of the α and $\alpha\beta$ -infinite hulls and their respective domains. PASS was developed as a research tool for digital curve analysis which yielded many of the results found in this dissertation and also provided many of the figures found herein.

Programmed in Java 2, PASS is a menu-driven graphical environment that allows the application user to draw shapes such as lines, circular arcs and ellipses as well as general digital curves. The digital curves are shown as pixel sequences. After drawing a digital curve the curve's upper and lower boundaries, convex infinite hull and α and $\alpha\beta$ -infinite hulls may be computed. The domain for each computation as well as the critical sets are computed and displayed.

8. Passing Area Solver Software (PASS) Package

In this section the Passing Area Solver Software (PASS) package, developed as a research aid and visualization tool for computing and understanding the passing area and domain of digital curves, is presented. This tool was developed using the Java 2 programming language which contains an abundant number of packages that are rich in mathematical and graphical objects. In addition, Java programs are platform independent and run on all major operating systems. A typical screen view of PASS is found in figure 8.1.

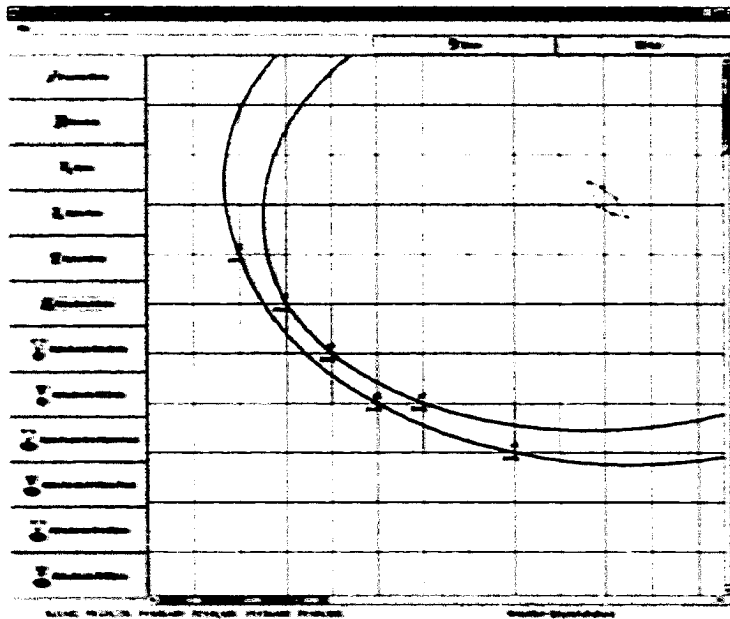


Figure 8.1. A screen view of PASS. A digital curve (here an ellipse) may be plotted by selecting the pixels with the mouse. The $\alpha\beta$ -infinite hull may then be computed by pressing the $\alpha\beta$ -infinite button and entering the values of the semi-axes, r_x and r_y .

Although a number of geometric software applications exist such as Geometer's SketchPad [Jackiw, 1997] and TI Cabri Geometry [Wilgus, 1998], these programs are geared towards simple geometric visualizations and are not easily extendable to handle more complex geometric computations. Mathematical applications such as Mathematica are very useful in computing mathematical solution to complex problems and graphically displaying complex shapes but lack a built-in ability for menus and windowing. Therefore, PASS was developed completely in Java 2 which supports all necessary features.

8.1 The User Interface

The user interface is mainly via the mouse and command buttons with occasional pop-up window for numeric data input when necessary. A user may use PASS as a drawing utility, to enter a digital curve, to compute the α and $\alpha\beta$ -hulls of a digital curve, determine a digital curve's critical sets and members or display the digital curve's domain. Built in is the ability to magnify and scroll the image presented so that details may be viewed with greater resolution. In the following discussion the features of PASS are discussed with an eye toward their application to concepts described earlier in this paper.

PASS runs in two modes: the draw mode and the α mode. The specific mode is selected by pressing either the Draw button, , or the α -Hull button, , found on the mode panel on the top of the window. In each mode a different command

panel appears on left of the window, as in figure 8.1. The features for each mode are found along its corresponding command panel: the draw panel and the α panel, shown in figure 8.2. The next two sections discuss the features found in both of these panels.

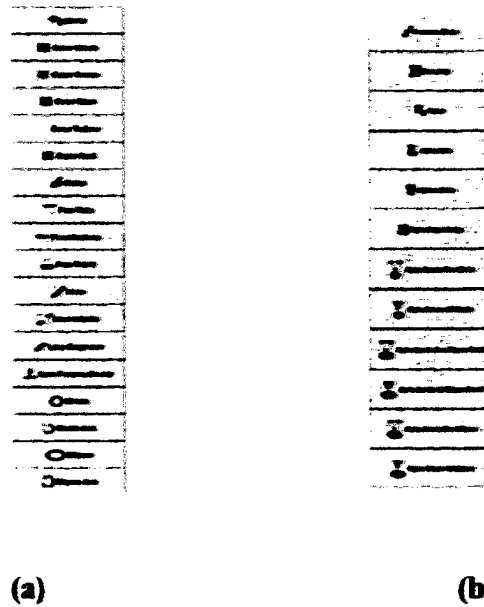


Figure 8.2. The draw command panel (a) and the α -Hull command panel (b) facilitate a simple and straightforward interface between the user and PASS.

8.2 The Draw Panel

The draw panel (figure 5.2a) contains all of the buttons that facilitate drawing in PASS. The commands are generally divided into two major categories: shape and style. The shapes that may be drawn are line segments, lines, line segment extensions, perpendicular bisectors, skewed bisectors, circles, circular arcs, ellipses and elliptical arcs; these are shapes that have significance in the context of computing or displaying the

passing area or domain of a digital curve. Styles include pen thickness and color. See figure 8.3 for an example of the shapes and styles available.

Another feature available is unlimited undo. Any shape drawn may be removed from the display area by clicking on the Undo button . This is implemented by a stack data structure the most recent shape from the stack.

In the next section we discuss the individual shapes available and the techniques used to display them.

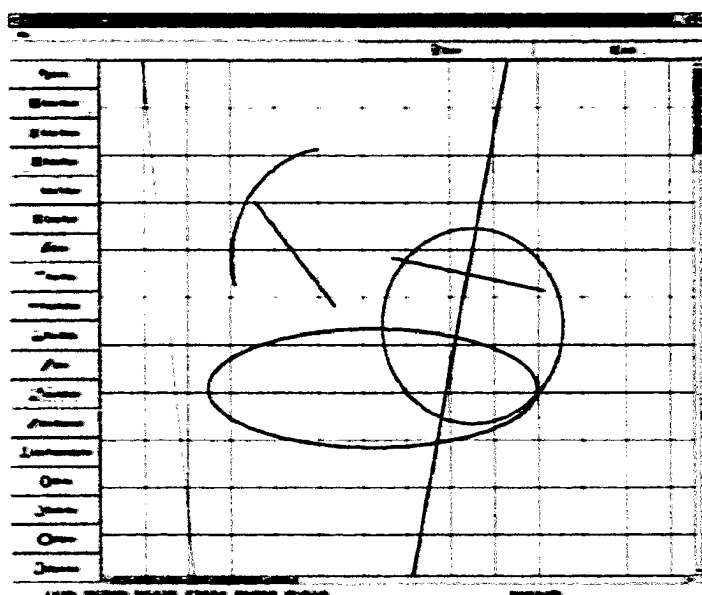


Figure 8.3. Lines, circles and ellipses are displayed by clicking on grid points and the appropriate buttons.

8.2.1 Shapes

The categories of shapes that may be drawn using PASS are lines, circles and ellipses. Each shape requires clicking with the mouse on at least one grid point. The

status bar indicates the present location of the mouse pointer as well as the points (numbered p_0 to p_4 in figure 8.4), which have been selected, as shown in figure 8.4. The individual shapes and the methods in which they are drawn is now discussed.

5.46,3.10 p_0 — p_1 — p_2 — p_3 — p_4 —

Figure 8.4. The status bar, found on the bottom left of the main window tracks the current location (x,y) of the cursor shown here as 5.46,3.10 as well as the five points that the user may select, numbered p_0 to p_4 (here shown unselected as indicated by the “—” placeholder).

8.2.1.1 Lines

Drawing lines, line segments and line segment extensions (i.e., the linear extension of a line segment) are useful in determining the convex hull and infinite convex hull of the upper and lower boundaries of a digital line segment. Simple (infinite) lines may be drawn by clicking two points on the grid followed by the Line button

. Line segments and line segment extensions are similarly drawn by selecting two points and the Line Segment  and Line Extensions  button, respectively.

On the right side of the status bar the slope and y-intercept values of the last line drawn is displayed, as shown in figure 8.5.

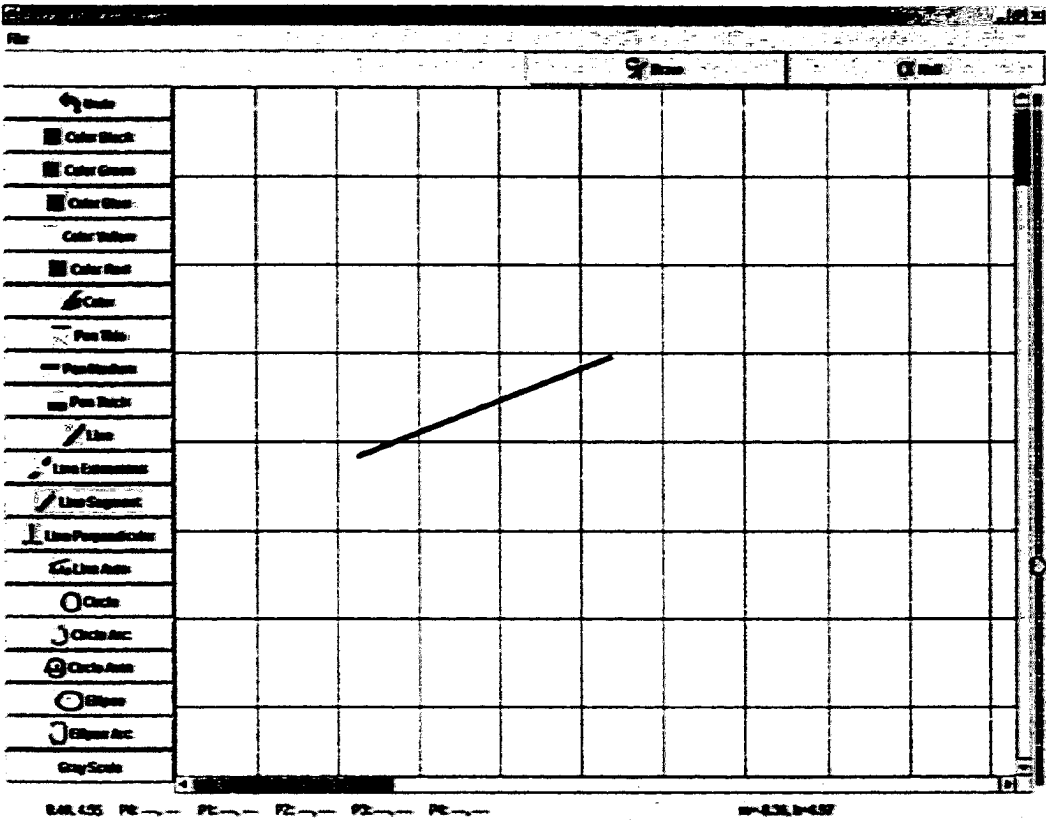
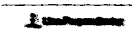


Figure 8.5. The mathematical description of the last curve draw is specified in the status bar. Above the line segment shown has slope and y-intercept of 0.36 and 4.97, respectively.

The perpendicular bisectors of line segments whose endpoints are critical points along the curve’s inner and outer boundary bound the domain of a digital circular arc. Similarly, skewed bisectors, parameterized by the ratio of the semi-axes, $ratio = r_x / r_y$, bound the boundary of digital elliptical arcs. The value of *ratio* for the unskewed perpendicular bisector is 1. After selecting two grid points, pressing the Line Perpendicular button  and entering the *ratio* value in the pop-up window,

the perpendicular or skewed bisector is displayed. As discussed in section 7.5 the domain of an elliptical digital curve where $ratio = r_x / r_y$ is bound by a skewed bisector of two critical points with equal *ratio* value.

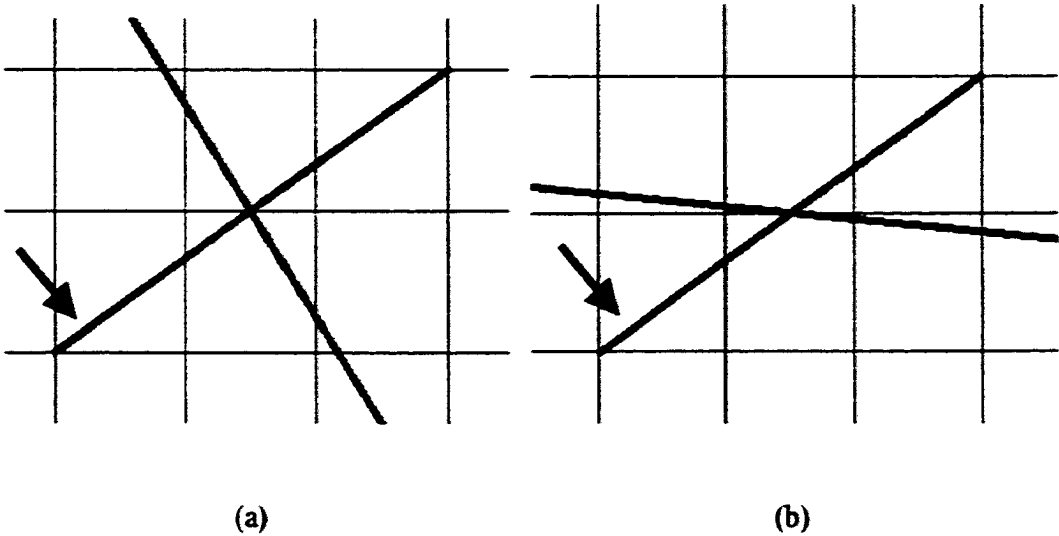
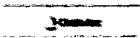


Figure 8.6. The perpendicular bisector of the indicated line segment (a) is unskewed (ratio $x:y = 1$). With a skew ratio of 4 (b) the bisector intersects the line segment's midpoint but is skewed by a factor of $1/16$ ($=1/ratio^2 = 1/4^2$) in the x axis direction.

8.2.1.2 Circles and Ellipses

Circles and circular arcs may be displayed in a number of ways. Each is achieved by selecting the appropriate grid points, pressing the appropriate button  or  and entering the information as prompted. A circle is displayed in three ways:

- a) selecting the center point and entering the circle's radius
- b) selecting two points on the circle, ordered counterclockwise, and entering the radius
- c) selecting three points along the circle.

A circular arc may be displayed by selecting three points, ordered counterclockwise, found on the circular arc. The first and last points indicate the start and end of the arc, respectively.

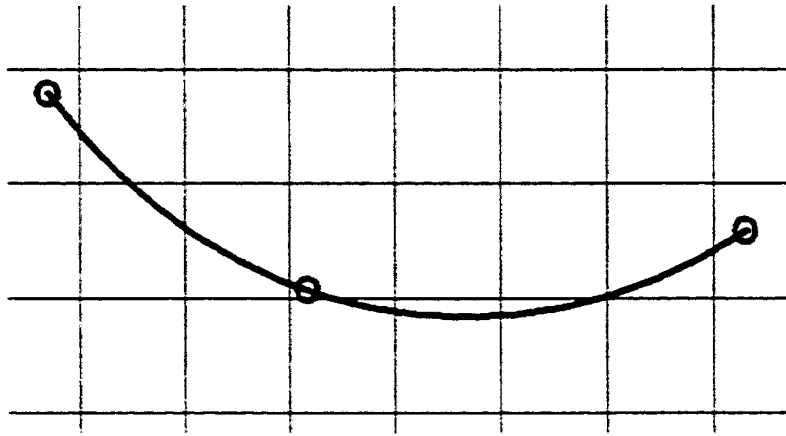
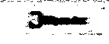


Figure 8.7. Three points, in order counter clockwise, specify a circular arc. The first and last points selected are the arc's endpoints.

When a circle or circular arc is displayed, the circle center or circular arc center and radius are displayed in the right side of the status bar. This feature is particularly useful where the radius and center are not known such as when a circle or circular arc are drawn by selecting three points along the circle or circular arc.

Ellipses and elliptical arcs are displayed in a similar fashion by first pressing command buttons  and  respectively and then entering the ellipse's (lengths of) semi-axes ratio (r_x / r_y). Analytically, an ellipse may be drawn by:

- a) selecting the ellipse center point and entering (length of) the horizontal semi-axis, r_x , and *ratio* of the lengths of the two semi-axes.
- b) selecting two points on the ellipse, ordered counterclockwise, and the horizontal semi-axis, r_x , and the *ratio*.
- c) selecting three points on the ellipse and entering the *ratio* of the lengths of the two semi-axes.

An elliptical arc is displayed by selecting three points on the arc, ordered counterclockwise, and entering *ratio*. The arc's endpoints are set to the first and last points selected. The ellipse center and elliptical arc center and semi axes appear on the right side of the status bar. See figure 8.8.

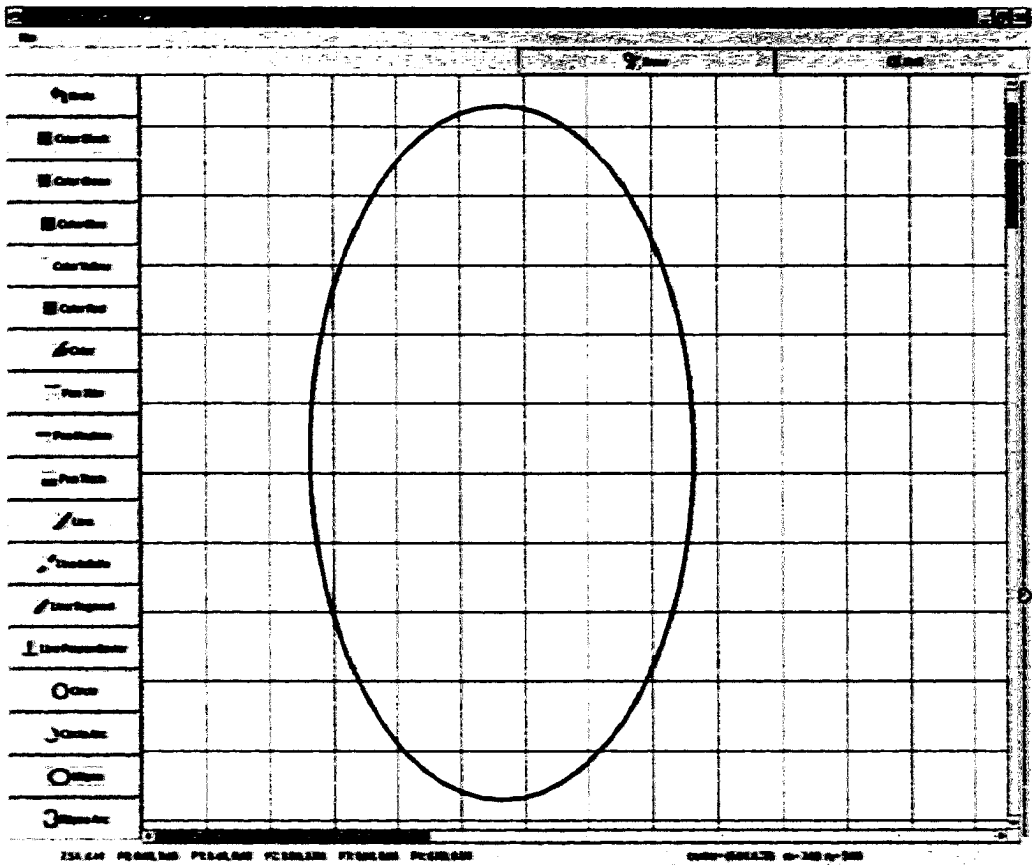
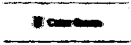
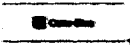
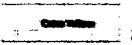
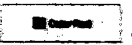
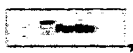
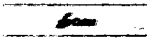


Figure 8.8. An ellipse where $r_x = 3$ and $ratio = 0.6$. The ellipse drawn has $r_x = 3$ and $r_y = 5$. The center and values of the semi-axes are shown on the status bar (left).

8.2.2 Draw Style

The draw style determines the color and thickness of the shapes displayed. The default draw style is set so that the color is black and the thickness is “thin”. By pressing a color button for the colors black, green, blue, yellow or red using buttons , , ,  and  or a pen thickness style (thin, medium

or thick) via buttons ,  and  the current draw style changes such that all subsequent drawings are displayed in the new style until the style is changed. Any color available on the local OS (platform) may be chosen via the Select Color button , which opens a window for color selection as shown in figure 8.9.

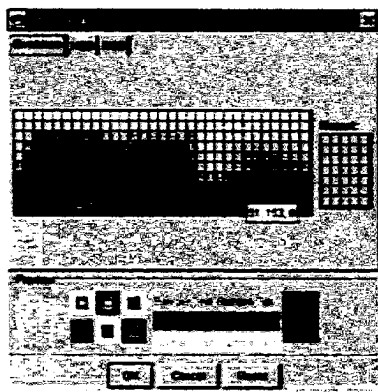


Figure 8.9. Color palette. A shape may be drawn in any color supported by the platform by selecting a color value from the color palette.

8.3 The α Panel

With the α panel the user may graphically construct a digital curve and compute the α -hulls and $\alpha\beta$ -hulls of that digital curve. Features include the ability to manually select the individual pixels of the digital curve as well as automating the pixel selection process by specifying the curve's parameters (section 8.2.1). After constructing a digital curve the α or $\alpha\beta$ -infinite hulls may be determined in a number of formats (section 8.2.2) such as computing a single $\alpha\beta$ -infinite hull or solving many $\alpha\beta$ -infinite hulls iteratively.

8.3.1 Plotting a Digital Curve and the Inner and Outer Boundaries

A 4-connected digital curve is plotted by selecting the Freeman Chain button, , and clicking on pixels found on the grid that make up the digital curve. The pixels must be selected in order counterclockwise. Although technically a 4-connected digital curve must be a single curve where each pixel is connected to two other pixels with city distance of one, this requirement is relaxed to allow the entry of disconnected curves, as shown in figure 8.10.

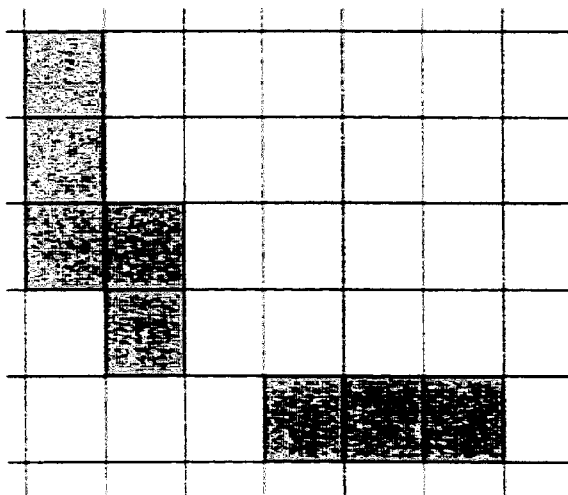
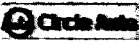


Figure 8.10. Disconnected digital curves may be constructed and all operations on disconnected digital curves are supported. An $\alpha\beta$ -hull of the disconnected digital curve is shown.

Another option for plotting a digital line segment or circular arc is by selecting the Line Auto  and Circle Auto  buttons, respectively. The user is then prompted to specify the curves parameters: length and slope for the line segment and

radius and angular span for the circular arc. The complete line and circle preimage are displayed as well as shown in figure 8.11.

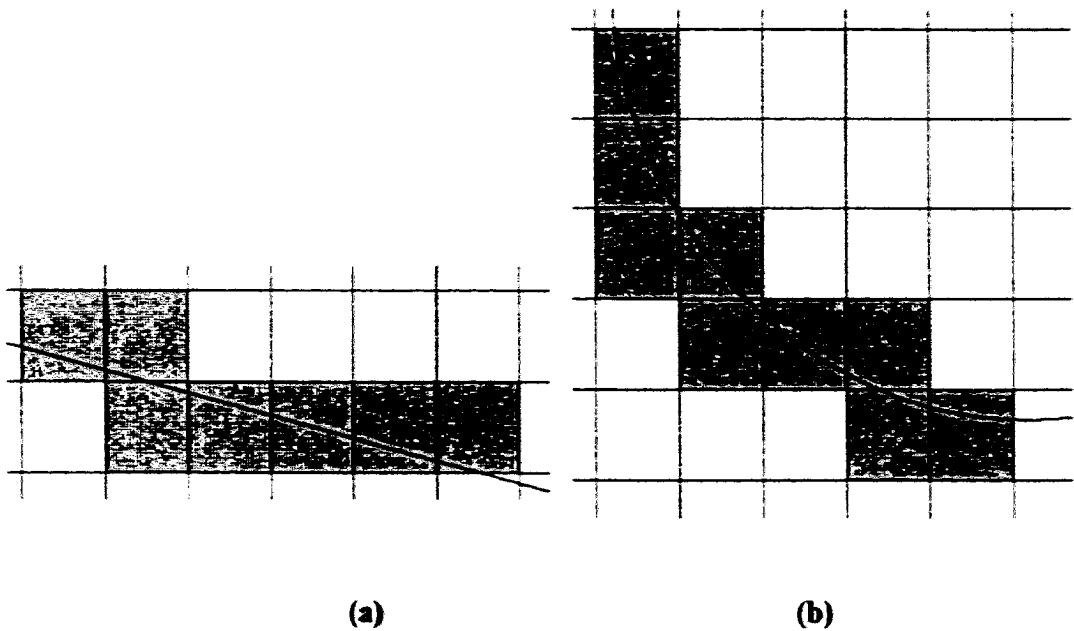


Figure 8.11. Digital curves may be constructed by specifying the curve's parameters. For example, a digital line segment may be displayed by specifying the segment's length and slope (a) and a digital circular arc is displayed by specifying the radius and angular span.

The inner and outer boundaries of a digital curve may be computed and displayed by selecting the Boundary button, . The boundary may be displayed as shown in figure 8.12. All elements that are candidates for inclusion in the critical set are identified with their grid locations and are labeled as in the figure.

Infinite button, , and enter 0 for the radius value. The results of these operations are shown in figure 8.13 and 8.14.

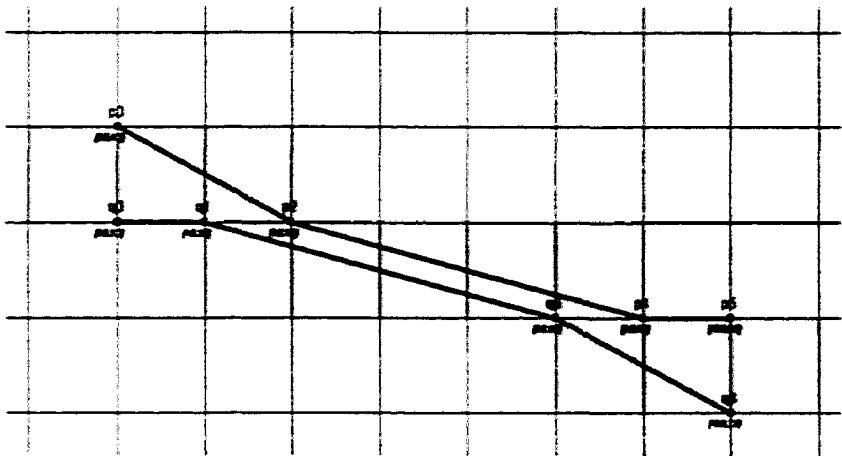


Figure 8.13. The portion of the convex hull of the upper and lower boundaries that constrains the passing area is shown.

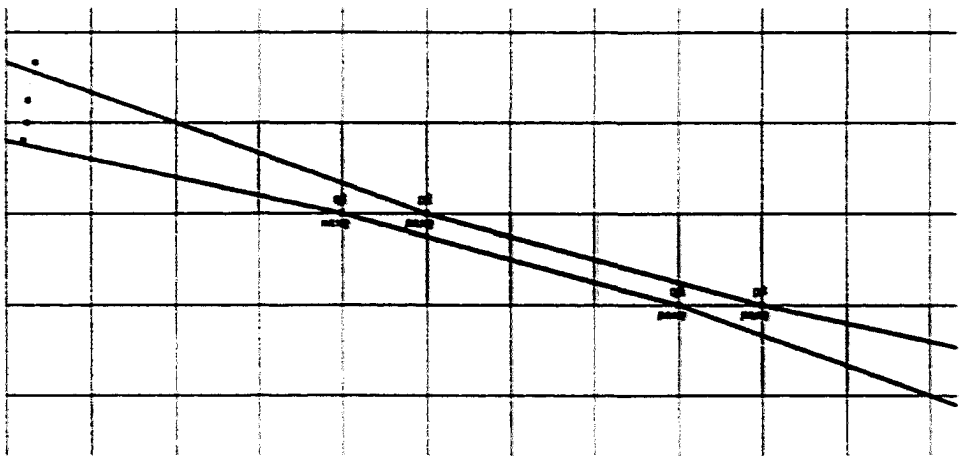
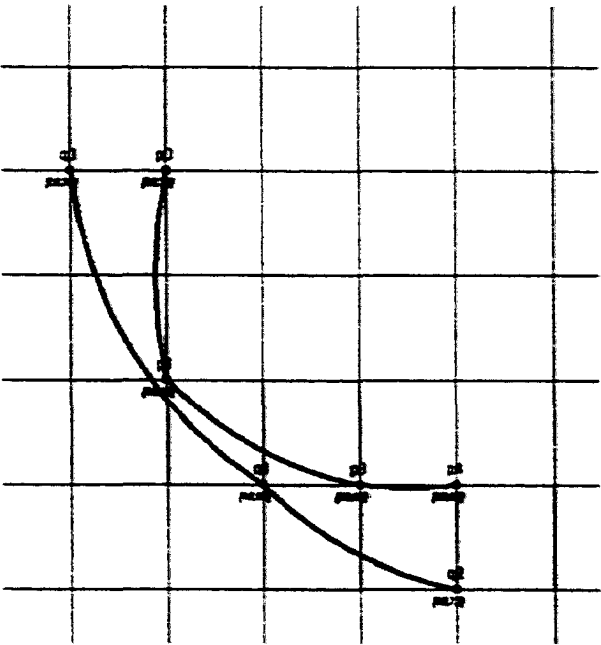


Figure 8.14. The perimeter of the infinite convex hull of the digital line segment. In the upper left the domain in (m, b) space is shown.

The α -hull and α -infinite hulls of digital circular arcs are computed in a similar manner. Instead of entering 0 for the radius the value of $r_x (= 1 / \alpha)$ is entered as the radial value. The domain is superimposed on the grid. See figures 8.15 and 8.16, where the α -hull and α -infinite hull (where $r = 3$) is shown. After computing the α -infinite hull the passing area's size and critical elements along the passing area perimeter are displayed as such (referring to figure 8.16):

Area=8.92 Critical Set={p1 p4 q0 q1}



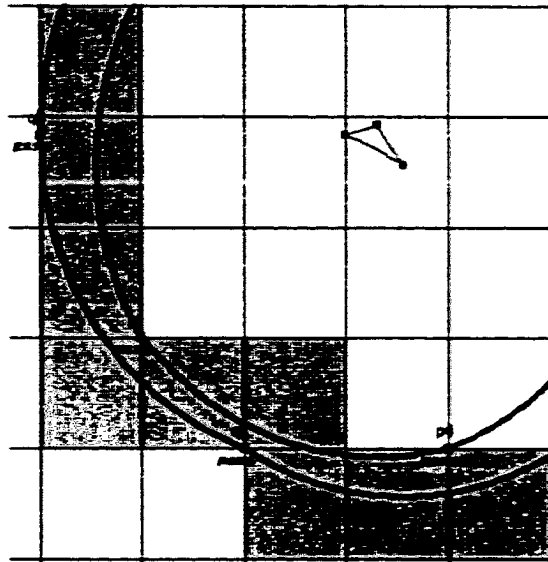


Figure 8.16. The α -infinite hull of a digital circular arc, where $r = 3$. The passing area is clearly narrower than that shown in figure 8.15. The (triangular) domain of the digital arc is shown as well.

The $\alpha\beta$ -hull and $\alpha\beta$ -infinite hull of the joined boundary (for ellipses and elliptical arcs) are computed by pressing the  and  buttons respectively. The user is then prompted for the values of the horizontal semi-axis, r_x , followed by the vertical semi-axis, r_y . The $\alpha\beta$ -hull (or $\alpha\beta$ -infinite hull) is then displayed with its domain, as in figures 8.17 and 8.18.

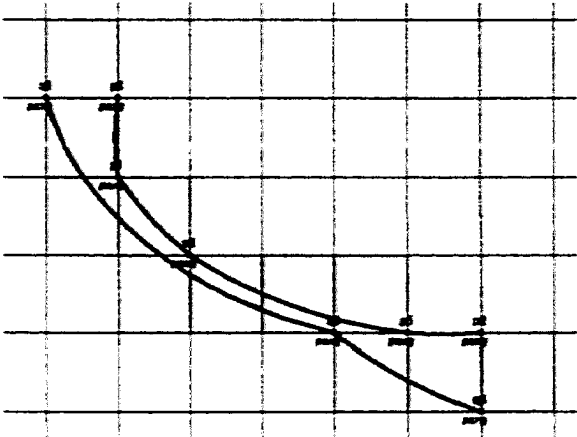


Figure 8.17. The $\alpha\beta$ -hull of a digital elliptical arc, where $r_x = 6$ and $r_y = 4$.



Figure 8.18. The $\alpha\beta$ -infinite hull of a digital elliptical arc, where $r_x = 6$ and $r_y = 4$. The domain of the digital arc is also shown.

When the α -hull or $\alpha\beta$ -hull is computed the status bar will specify the size of the passing area and identify the critical elements found in the critical set for the radial value entered. For example, when the critical set includes p_1, p_3, p_5, q_0 and q_3 the status bar will include this information as. **Area-153 Critical Set={p1 p3 p5 q0 q3}**

8.3.3 **Computing the Hulls and Critical Sets as the Radius or Semi-Axes Change and the Domain**

As the radius of a circular or elliptical arc changes, so does the α -hull of the digital curve representation (chapter 5). Critical changes occur only where the elements in the critical set (i.e. those elements along the passing area boundary) change. PASS includes a utility, which displays the changes in the critical set as the radius increases. The radius value where these changes take place along with the critical elements is presented to the application user.

After plotting the digital curve, the user presses the Alpha Iterate button, . The user is prompted to for a start radius and an end radius which is the iteration range. PASS will then proceed to iterate through 200 radial values in the specified range and displays the critical sets of the digital curve within that radial range. At the conclusion of iteration the set of all critical ranges in (rx, ry) format and its critical elements is presented in a pop-up window, shown in figure 8.19. In addition the projection of the domain in that range onto the (x,y) -plane is displayed, as in figure 8.20.

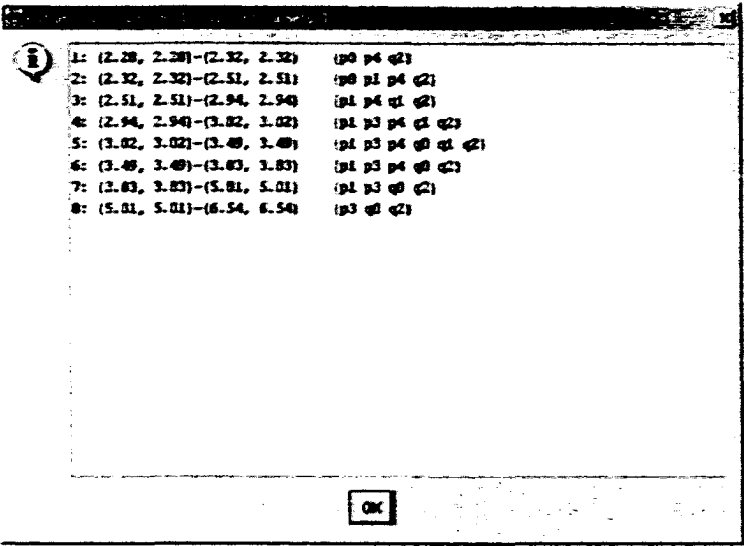


Figure 8.19. The critical ranges in (rx, ry) format and the critical sets of a digital curve (from figure 8.20) within the specified radial range contain 8 critical sets. The radial range and number of critical sets are found in the title bar.

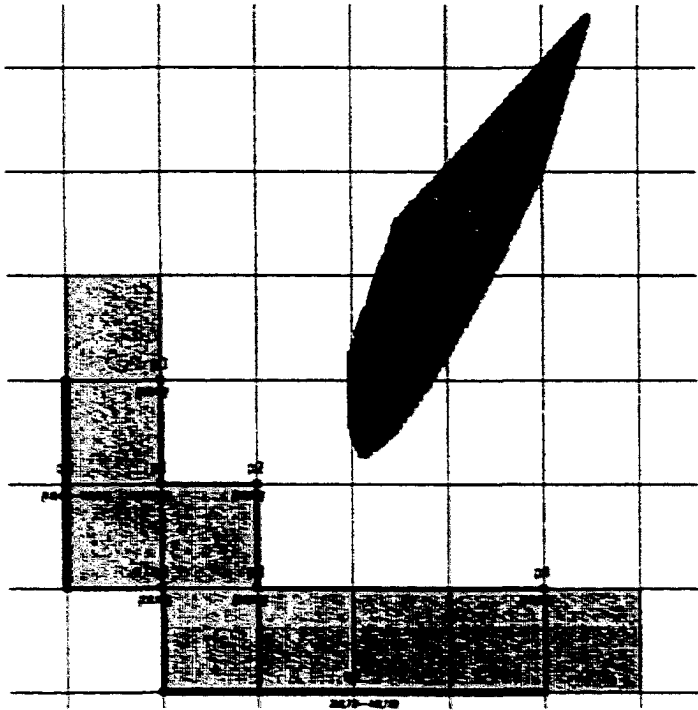


Figure 8.20. The domain of the digital curve is shown. Selecting the Alpha Iterate operation produces this figure.

Similarly, pressing the Alpha Beta Fixed Iterate button, , will output the critical sets of an ellipse with fixed semi-axes ratio. The user is first prompted for the start and end horizontal semi-axis (r_x) value followed by the ellipse's *constant* semi-axes ratio (i.e. r_x / r_y). Iteration then proceeds determining the critical sets. Thus, for example, if the start and stop values for r_x are 3 and 50, respectively, and the *ratio* is 2, the critical sets and domain for that range will be displayed, as shown in figures 8.21 and 8.22. Note that the critical sets as well as the shape of the domain are different than that of a circle shown earlier.

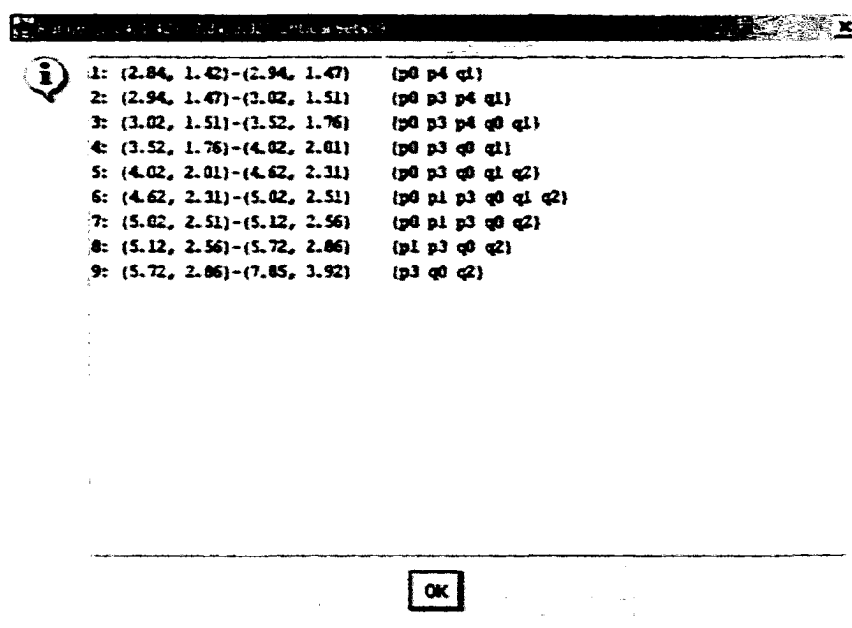


Figure 8.21. The critical sets of the digital curve interpreted as an ellipse with $ratio = 2$. The radial range and number of critical sets are found in the title bar.

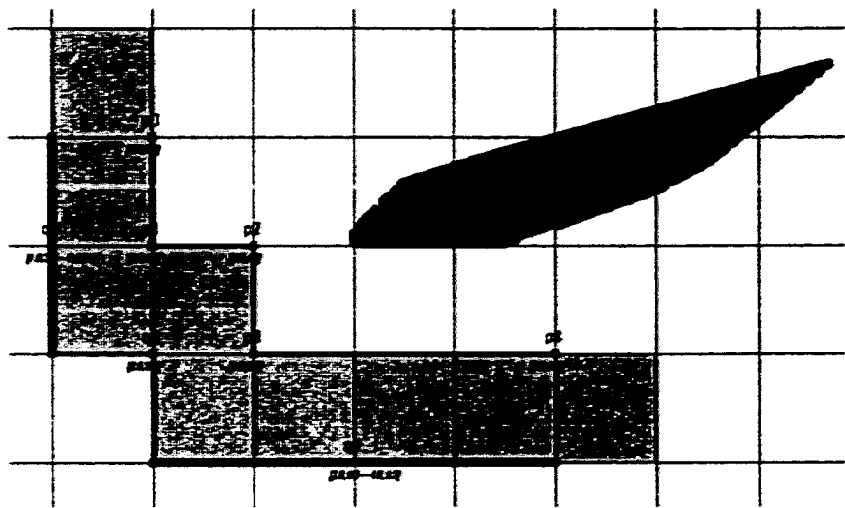


Figure 8.22. The domain of the digital curve interpreted as an ellipse with *ratio* = 2 is shown. Selecting the Alpha Beta Fixed Iterate operation produces this figure.

Finally, the application supports the ability to track the critical sets for an ellipse where the semi-axes ratio changes. This is accomplished with the Alpha Beta General Iterate button, . After pressing the button the user is prompted for the start semi axes and end semi axes and the number of iterations to compute the $\alpha\beta$ -infinite hull. PASS will then iterate through a nested loop where r_x ranges between the start and end values entered and r_y ranges between its start and end values. The number of iterations entered is the number of iterations per loop. Thus, the total number of iterations is the iteration number squared. The critical sets as well as the domain in the range entered are displayed, as shown in figures 8.23 and 8.24.

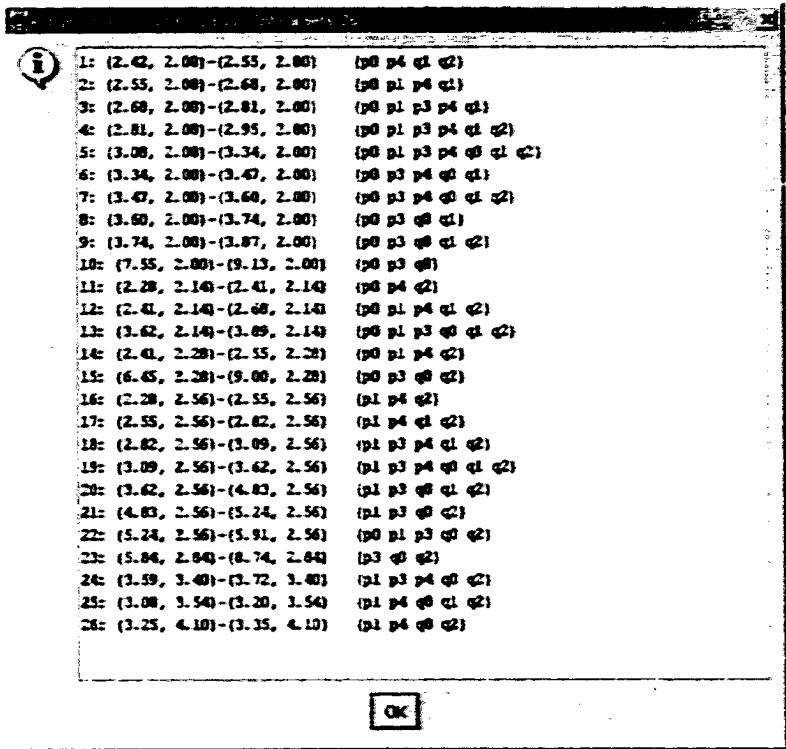


Figure 8.23. The critical sets of the digital curve interpreted as an ellipse where r_x and r_y both range between 2 and 9. The radial range and number of critical sets are found in the title bar.

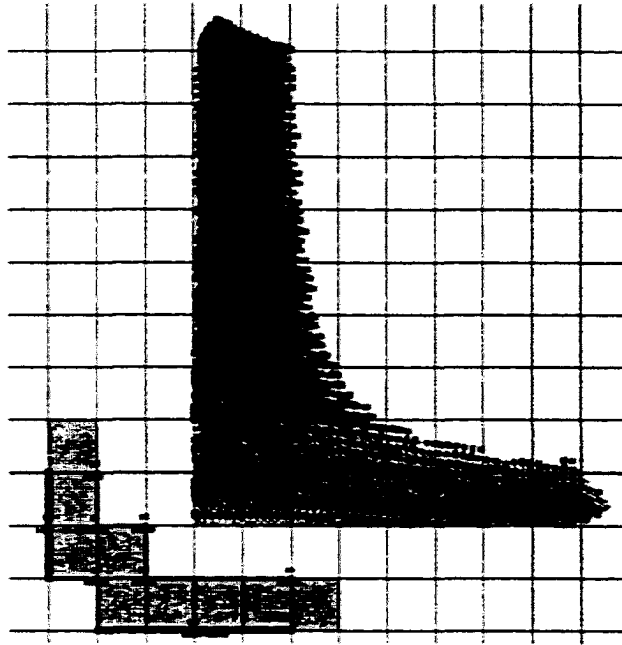


Figure 8.24 The domain of the digital curve is shown where r_x and r_y both range from 2 to 9. This domain includes the circular and elliptical domains from the previous figures in addition to all other ellipses in the specified range.

8.4 Summary

The Passing Area Solver Software (PASS) package was developed as a research aid and visualization tool for computing and understanding the passing area and domain of digital curves. This application was pivotal in verifying results discussed in this dissertation as well as geometrically displaying them. Furthermore, it has the capability to be further extended to provide the platform necessary to investigate more complex questions such as what the maximum number of critical sets for an elliptical curve whose eccentricity is unknown. In addition, the algorithms can similarly be extended to

determine convex hulls for complex shapes such as general conics and three-dimensional surfaces.

APPENDIX

Java Source Code for PASS

Area.java

```

package domain;

import java.awt.geom.*;
import java.lang.*;

/**
 * Title:
 * Description:
 * Copyright: Copyright (c) 2001
 * Company: Ph.D. Application
 * Author: Jonathan Robinson
 * Version:
 *
 * computes the area of different shapes including an infinite hull
 * passing area
 */
public class Area {
    /**
     * Represents the direction of a half space bounded by a vector defined
     * by a line.
     * @see #RectangleLine
     */
    public static int RIGHT = 1;
    /**
     * Represents the direction of a half space bounded by a vector defined
     * by a line.
     * @see #RectangleLine
     */
    public static int LEFT = 2;
    /**
     * returns the area of the triangle with points p1, p2 and p3.
     */
    public static double triangle(Point2D p1, Point2D p2, Point2D p3) {
        //Archimedes / Heron of Alexandria
        double a = p1.distance(p2);
        double b = p1.distance(p3);
        double c = p2.distance(p3);
        double s = ( a + b + c ) / 2;
        return Math.sqrt(s * ( s - a ) * ( s - b ) * ( s - c ));
    }
    /**
     * returns the area of the triangle with points p1, p2 and p3. Returns
     * the
     * positive area if p3 is to the right of vector p1p2 and negative
     * otherwise.
     */
    public static double triangleOrientated(Point2D p1, Point2D p2,
        Point2D p3) {
        double area = triangle(p1, p2, p3);
        if (Double.isNaN(area))
            area = 0;
        return (new Line2D.Double(p1, p2).relativeCCW( p3 ) == - 1) ? area :
            -area;
    }
}

```

```

    }

    /**
     * returns the area of the minor segment defined by points p1, p2 and
     * radius r. This method assumes that
     * the crescent is defined by a circular arc
     */
    public static double segment(Point2D p1, Point2D p2, double r) {

        if (r == PointAndArc.LINE) return 0.0;           // crescent is empty

        double b = p1.distance( p2 );
        double theta = 2 * Math.asin( b / ( 2 * r ) );
        double area = (r * r) / 2 * ( theta - Math.sin( theta ) );
        // if (Double.isNaN(area))
        //     area = 0;
        return area;
    }

    /**
     * returns the area of the passing area where list1 and 2 are the upper
     * and inner hulls respectively.
     * This method assumes the passing is is constrained by circular --
     * not elliptical arcs. The
     * method can be extended to handle elliptical passing areas by
     * overloading method crescent
     *
     * This method may also be used to compute the area of any convex
     * region as well
     * as a concave region where list1 and list2 are relatively convex.
     * Relative convexity of a region defined by two lists is where every
     * line segment
     * from any point in list1 to any point in list2 is within the region.
     */
    public static double passingArea(PointAndArc list1[], PointAndArc
list2[]) {
        double total = 0,
            tempArea;

        /* first adds/ subtracts triangle area and adds/subtract crescent
        area from
        /* upper and lower hulls*/
        for (int i = 0; i < list1.length - 1; i++) {
            total += triangleOrientated( list1[i], list1[i+ 1], list2[0] );
            total -= segment( list1[i], list1[i+ 1], list1[i].getRadiusX() );
        }

        int list1Last = list1.length - 1;
        for (int i = 0; i < list2.length - 1; i++) {
            total += triangleOrientated( list2[i], list1[list1Last],
list2[i+1] );
            total -= segment( list2[i], list2[i+ 1], list2[i].getRadiusX() );
        }

        return total;
    } // end of passingArea

```

```

    /**
     * Returns true if the points are on the same vertical or horizontal
     * line within the specified tolerance
     */
    public static boolean onSameLine(Point2D p1, Point2D p2, double
tolerance){
        return Math.abs( p1.getX() - p2.getX() ) < tolerance ||
Math.abs( p1.getY() - p2.getY() ) < tolerance;
    }

    /**
     * Returns true if the points are on the same vertical or horizontal
     * line within
     * {see #Lines.TOLERANCE
     */
    public static boolean onSameLine(Point2D p1, Point2D p2){
        return onSameLine(p1, p2, Lines.TOLERANCE);
    }

    /**
     * Sets array corner and lines to the corner points and perimeter
     * lines
     * of rect. corner and lines MUST have at least four elements.
     */
    private static void getCornerPointsAndPerimeterLines(Rectangle2D
rect, Point2D corner[], Line2D lines[]) {
        corner[0] = new Point2D.Double( rect.getX(), rect.getY());
        corner[1] = new Point2D.Double( rect.getX() + rect.getWidth(),
rect.getY() );
        corner[2] = new Point2D.Double( rect.getX() , rect.getY() +
rect.getHeight() );
        corner[3] = new Point2D.Double( rect.getX() + rect.getWidth(),
rect.getY() + rect.getHeight() );

        lines[0] = new Line2D.Double( corner[0], corner[1] );
        lines[1] = new Line2D.Double( corner[0], corner[2] );
        lines[2] = new Line2D.Double( corner[2], corner[3] );
        lines[3] = new Line2D.Double( corner[1], corner[3] );
    }

    /**
     * Returns the area of the intersection of the rectangle with the
     * circle. The current implementation
     * can be extended to handle ellipses as well, assuming the segment
     * method is rectone.
     * Assumes circle's diameter is greater than rect's height and width;
     * with a the two, thus,
     * intersecting at most in four places.
     */
    public static double RectangleCircle(Rectangle2D rect, Ellipse
circle) {
        // determine number of corner points in circle and if circle
        intersects rectange perimeter 4 times.
        double area = 0;

        Point2D corner[] = new Point2D[4];

```

```

Line2D lines[] = new Line2D[4];
getCornerPointsAndPerimeterLines(rect, corner, lines);

int inside = 0, outside = 0;
Point2D cornerInside[] = new Point2D[4],
      cornerOutside[] = new Point2D[4];

for (int i = 0; i < corner.length; i++) {
    if ( circle.contains( corner[i] ) )
        cornerInside[inside++] = corner[i];
    else
        cornerOutside[outside++] = corner[i];
}

// find intersection points of circle and rectangle 2 MUST exist
Point2D intersection[] = new Point2D[4],
      tempIntersection[] = new Point2D[2];
int intersectionIndex = 0,
    sameLineIndex = -1;
for (int i = 0; i < lines.length; i++) {
    switch (circle.intersects( lines[i], tempIntersection ) ) {
        case 0:
            break;
        case 1:
            intersection[ intersectionIndex++ ] = tempIntersection[ 0];
            break;
        case 2:
            sameLineIndex = intersectionIndex;
            intersection[ intersectionIndex++ ] = tempIntersection[ 0];
            intersection[ intersectionIndex++ ] = tempIntersection[ 1];
            break;
    }
}

switch ( inside ) {
    case 0:
        area = segment(intersection[ 0], intersection[1],
circle.radiusX);
        break;
    case 1:
        area = triangle(cornerInside[ 0], intersection[0],
intersection[1] ) +
            segment(intersection[ 0], intersection[1], circle.radiusX
);
        break;
    case 2:
        double base = cornerInside[ 0].distance( cornerInside[1] );
        Point2D topIntersection[] = new Point2D[2],
            sideIntersection[] = new Point2D[2]; // parallel to
cornerInside

        if (intersectionIndex == 2) { // case A
            for (int crnr = 0; crnr < 2; crnr++)
                for (int ip = 0; ip < intersectionIndex; ip++)
                    if ( onSameLine( cornerInside[crnr], intersection[ip] ) )
                        sideIntersection[ crnr ] = intersection[ip];

```



```

        area = (cornerInside[0].distance( sideIntersection[0] ) +
               cornerInside[1].distance( sideIntersection[1] ) ) *
        base / 2.0; // base * (height1 + height2 )/2
        break;
    }
    else { // case 3
        double tri1, tri2, seg1, seg2;
        for (int crnr = 0; crnr < 2; crnr++)
            for (int ip = 0; ip < intersectionIndex; ip++) {
                if (ip == sameLineIndex || ip == sameLineIndex + 1)
                    continue;
                if ( onSameLine( cornerOutside[crnr], intersection[ip] ) )
                    sideIntersection[ crnr ] = intersection[ip];
            }

            if ( sideIntersection[0].distance(
intersection[sameLineIndex]) <
                sideIntersection[0].distance( intersection[sameLineIndex
+ 1] ) ) {
                tri1 = triangle( sideIntersection[0],
intersection[sameLineIndex], cornerOutside[0]);
                seg1 = segment( sideIntersection[0],
intersection[sameLineIndex], circle.radiusX );
                tri2 = triangle( sideIntersection[1],
intersection[sameLineIndex+1], cornerOutside[1]);
                seg2 = segment( sideIntersection[1],
intersection[sameLineIndex+1], circle.radiusX );
            }
            else {
                tri1 = triangle( sideIntersection[0],
intersection[sameLineIndex+1], cornerOutside[0]);
                seg1 = segment( sideIntersection[0],
intersection[sameLineIndex+1], circle.radiusX );
                tri2 = triangle( sideIntersection[1],
intersection[sameLineIndex], cornerOutside[1]);
                seg2 = segment( sideIntersection[1],
intersection[sameLineIndex], circle.radiusX );
            }
            area = 1 - tri1 - tri2 + seg1 + seg2;
            break;
        }
        case 3:
            area = 1 - triangle( intersection[0], intersection[1],
cornerOutside[0] )
                + segment( intersection[0], intersection[1],
circle.radiusX );
            break;
        case 4:
            area = 1; // circle with infinite radius ??
            break;
    }
    return area ;
}
/*
**

```

```

    * Returns the area of the intersection of the rectangle with the
    half space to the right/left of the line.
    * The current implementation
    * can be extended to handle ellipses as well, assuming the segment
    method is re-done.
    */
    public static double RectangleLine(Rectangle2D rect, Line2D line, int
direction) {
        double area = 0;

        // first determine number of points inside half space
        Point2D corner[] = new Point2D[4];
        Line2D lines[] = new Line2D[4];
        getCornerPointsAndPerimeterLines(rect, corner, lines);

        int inside = 0, outside = 0;
        Point2D cornerInside[] = new Point2D[4],
            cornerOutside[] = new Point2D[4];

        for (int i = 0; i < corner.length; i++) {
            if ( line.relativeCCW( corner[i] ) == - 1 ) // right of vector
"line"
                cornerInside[inside++] = corner[i];
            else
                cornerOutside[outside++] = corner[i];
        }
        Point2D intersection[] = new Point2D[2];
        int intersectionIndex = 0;

        // if ( inside > 0 && inside < 4 ) {
        // find intersection points of line and rectangle
        for (int i = 0; i < lines.length; i++) {
            if ( (intersection[ intersectionIndex ] = Lines.linesIntersect(
lines[i], line) ) != null)
                // if lines overlap null is returned and will not count as
intersection
                intersectionIndex++;
            if ( intersectionIndex > 1 ) break; // both intersection were
found
        }
        //

        switch ( inside ) {
            case 0:
                area = 0;
                break;
            case 1:
                area = triangle(cornerInside[ 0], intersection[0],
intersection[1] );
                break;
            case 2:
                double base = cornerInside[ 0 ].distance( cornerInside[ 1 ] ),
                    height1,
                    height2;

                if ( onSameLine( cornerInside[ 0 ], intersection[0] ) ) {
                    height1 = cornerInside[ 0 ].distance( intersection[0] );

```

```

        height2 = cornerInside[ 1 ].distance( intersection[ 1 ] );
    }
    else {
        height1 = cornerInside[ 0 ].distance( intersection[ 1 ] );
        height2 = cornerInside[ 1 ].distance( intersection[ 0 ] );
    }
    area = ( height1 + height2 ) / 2.0;
    break;
case 3:
    area = 1 - triangle( intersection[ 0 ], intersection[ 1 ],
cornerOutside[ 0 ] );
    break;
case 4:
    area = 1;
    break;
}
// if (Double.isNaN(area))
//     area = 0;
return direction == RIGHT ? area : 1 - area;
}

/*
1
    using onSameLine to determine which point goes with which corner
    (corner1, corner2, ip1, ip2)
    return (distance (corner1, ip1) - distance (corner2, ip2)) / 2 -
segment (ip1, ip2);

2

    return ( triangle (ip1, ip2, corner1) - segment (ip1, ip2) );

3

    cornerMinusSegment (ip1, ip2, corner1)
    1 - triangle (ip1, ip2, corner1) - segment (ip1, ip2)

4

    1 - cornerMinusSegment (ip1, ip2, corner1)
    cornerMinusSegment (ip1, ip2, corner1)

5

    return (segment (ip1, ip2, )

6

7

*/
}

```



Area.java

Chain.java

```

package domain;
import java.awt.geom.*;
import java.awt.*;
import javax.swing.*;
import java.util.*;

/**
 *
 */
public class Chain {

    private static boolean neighbor(Rectangle2D r1, Rectangle2D r2) {
        return (Math.abs(r1.getX()-r2.getX()) <= 1) &&
            (Math.abs(r1.getY()-r2.getY()) <= 1);
    }

    public static void makeFreemanChainList(FreemanChainList freemanList,
        Line2D line) {
        makeFreemanChainList( freemanList, line, 0.0);
    }

    public static void makeFreemanChainList(FreemanChainList freemanList,
        Line2D line, double threshold) {
        final int UPRIGHT = 1,
            UPLEFT = 2,
            DOWNRIGHT = 3,
            DOWNLEFT = 4;
        String chain = "";

        boolean addPix = false;
        int mode = DOWNRIGHT;

        FreemanChain freeman = new FreemanChain();

        Rectangle2D
            start = new Rectangle2D.Double((int) line.getP1().getX(), (int)
line.getP1().getY(), 1, 1),
            end = new Rectangle2D.Double((int) line.getP2().getX(), (int)
line.getP2().getY(), 1, 1),
            current = (Rectangle2D) start.clone();
        int i = 0;
        do { //mode == DOWNRIGHT      mode != UPRIGHT. ; // BOTTOM HALF OF
CIRCLE
            Rectangle2D
                // uses screen coordinate system
                // upper left is 0,0
                up = new Rectangle2D.Double( current.getX(),current.getY() -
1, 1, 1),
                down = new Rectangle2D.Double( current.getX(),current.getY()
+ 1, 1, 1),
                left = new Rectangle2D.Double( current.getX() -
1,current.getY(), 1, 1),
                right = new Rectangle2D.Double(current.getX() +
1,current.getY(), 1, 1);

            switch (mode) {

```

```

case UPRIGHT:
    if (addPix = line.intersects(right) ) {
        chain += "0";
        current.setRect(right);
    }
    else if (addPix = line.intersects(up) ) {
        chain += "2";
        current.setRect(up);
    }
    else if (line.intersects(left) )
        mode = UPLEFT;
    else if (line.intersects(down) )
        mode = DOWNRIGHT;
    break;
case UPLEFT:
    if (addPix = line.intersects(up) ) {
        chain += "2";
        current.setRect(up);
    }
    else if (addPix = line.intersects(left) ) {
        chain += "4";
        current.setRect(left);
    }
    else if (line.intersects(down) )
        mode = DOWNLEFT;
    else if (line.intersects(right) )
        mode = UPRIGHT;
    break;
case DOWNRIGHT:
    if (addPix = line.intersects(down) ) {
        chain += "6";
        current.setRect(down);
    }
    else if (addPix = line.intersects(right) ) {
        chain += "0";
        current.setRect(right);
    }
    else if (line.intersects(up) )
        mode = UPRIGHT;
    else if (line.intersects(left) )
        mode = DOWNLEFT;
    break;
case DOWNLEFT:
    if (addPix = line.intersects(left) ) {
        chain += "4";
        current.setRect(left);
    }
    else if (addPix = line.intersects(down) ) {
        chain += "6";
        current.setRect(down);
    }
    else if (line.intersects(right) )
        mode = DOWNRIGHT;
    else if (line.intersects(up) )
        mode = UPLEFT;
    break;
} // switch

```

```

        if (addPix) {
            freeman.add(new Pixel((int) current.getX(), (int)
current.getY(), Area.RectangleLine( current, line, Area.LEFT) ));
        }
        addPix = false;
    } while (! neighbor(end,current) ); // while

    // recompute freeman for other thresholds
    FreemanChain digitalCurve = new FreemanChain();

    if (threshold == 0)
        digitalCurve = freeman;
    else {
        Iterator iter = freeman.iterator();
        if (!iter.hasNext()) return;
        Pixel p = (Pixel) iter.next(),
            relative = p,
            lastRelative;
        double area;

        Rectangle2D rect;

        while ( (area = Area.RectangleLine(rect = relative.rectangle(),
line, Area.LEFT) ) <= threshold )
            relative = relative.moveDirection(2); // set first pixel
            relative.greylevel = area; // sets area correctly

        digitalCurve.add( relative );
        // remaining pixels
        while ( iter.hasNext() ) {
            p = (Pixel) iter.next();
            lastRelative = relative;
            relative = relative.moveDirection(p.code);
            if ( ! p.equals(lastRelative) ) // p is not in digital curve
                if ( (area = Area.RectangleLine( relative.rectangle(),
line, Area.LEFT) ) >= threshold ) {
                    relative.greylevel = area;
                    digitalCurve.add( relative );
                }
            if ( p.greylevel >= threshold )
                digitalCurve.add( relative = new Pixel(p) );
        }
        //while
        //threshold > 0
    }

    //add digitalCurve to freemanlist
    ListIterator iter = digitalCurve.listIterator();
    while (iter.hasNext())
        freemanList.add((Pixel) iter.next());
}

public static void makeFreemanChainList(FreemanChainList freemanList,
double cx, double cy, double radius, double startAngle, double
sweepAngle) {
    makeFreemanChainList(freemanList, cx, cy, radius, radius,
startAngle, sweepAngle, 0);
}

```

```

    public static void makeFreemanChainList(FreemanChainList
freemanList, double cx, double cy, double radiusX, double radiusY,
double startAngle, double sweepAngle) {
    makeFreemanChainList(freemanList, cx, cy, radiusX, radiusY,
startAngle, sweepAngle, 0);
}

    public static void makeFreemanChainList(FreemanChainList freemanList,
double cx, double cy, double radiusX, double radiusY, double
startAngle, double sweepAngle, double threshold) {
//      if (0 < Math.PI * radius * (sweepAngle / 360) < 10) return;
//this eliminates some unidentified infinite loop
    FreemanChain freeman = new FreemanChain();

    final int UPRIGHT = 1,
        UPLEFT = 2,
        DOWNRIGHT = 3,
        DOWNLEFT = 4;
    String chain = "";
    Arc2D arc;

    boolean addPix = false;
    int mode = DOWNRIGHT;

    arc = new Arc2D.Double(cx - radiusX, cy - radiusY, radiusX * 2,
radiusY * 2, startAngle+180, sweepAngle, Arc2D.OPEN);
    Ellipse circle = new Ellipse( new Point2D.Double( cx, cy ),
radiusX, radiusX/radiusY );

    Rectangle2D
start = new Rectangle2D.Double((int)
arc.getStartPoint().getX(), (int) arc.getStartPoint().getY() , 1, 1 ),
end = new Rectangle2D.Double((int) arc.getEndPoint().getX(),
(int) arc.getEndPoint().getY() , 1, 1 ),
current = (Rectangle2D) start.clone();
    int i = 0;
    do { //mode == DOWNRIGHT      mode != UPRIGHT) { // BOTTOM HALF OF
CIRCLE
        Rectangle2D // uses screen coordinate system
upper left is 0,0
        up = new Rectangle2D.Double( current.getX(),current.getY() -
1,1, 1),
        down = new Rectangle2D.Double( current.getX(),current.getY()
+ 1, 1, 1),
        left = new Rectangle2D.Double( current.getX() -
1,current.getY() , 1, 1),
        right = new Rectangle2D.Double(current.getX() +
1,current.getY() , 1, 1);

        switch (mode) {
            case UPRIGHT:
                if (addPix = arc.intersects(right) ) {
                    chain += "0";
                    current.setRect(right);
                }

```



```

else if (addPix = arc.intersects(up) ) {
    chain += "2";
    current.setRect(up);
}
else if (arc.intersects(left) )
    mode = UPLEFT;
else if (arc.intersects(down) )
    mode = DOWNRIGHT;
break;
case UPLEFT:
    if (addPix = arc.intersects(up) ) {
        chain += "2";
        current.setRect(up);
    }
    else if (addPix = arc.intersects(left) ) {
        chain += "4";
        current.setRect(left);
    }
    else if (arc.intersects(down) )
        mode = DOWNLEFT;
    else if (arc.intersects(right) )
        mode = UPRIGHT;
    break;
case DOWNRIGHT:
    if (addPix = arc.intersects(down) ) {
        chain += "6";
        current.setRect(down);
    }
    else if (addPix = arc.intersects(right) ) {
        chain += "0";
        current.setRect(right);
    }
    else if (arc.intersects(up) )
        mode = UPRIGHT;
    else if (arc.intersects(left) )
        mode = DOWNLEFT;
    break;
case DOWNLEFT:
    if (addPix = arc.intersects(left) ) {
        chain += "4";
        current.setRect(left);
    }
    else if (addPix = arc.intersects(down) ) {
        chain += "6";
        current.setRect(down);
    }
    else if (arc.intersects(right) )
        mode = DOWNRIGHT;
    else if (arc.intersects(up) )
        mode = UPLEFT;
    break;
} // switch
if (addPix) {
    double area = 0;
    try {
        area = Area.RectangleCircle( current, circle);
    }
}

```

```

        catch (ArrayIndexOutOfBoundsException e) { // catches
unidentified error from Area.RectangleCircle
            area = 0;
        }
        freeman.add(new Pixel((int) current.getX(), (int)
current.getY(), area));
    }
    addPix = false;
} while (! neighbor(end,current) ); // while

// recompute freeman for other thresholds
FreemanChain digitalCurve = new FreemanChain();

if (threshold == 0)
    digitalCurve = freeman;
/*
else {
    Iterator iter = freeman.iterator();
    if (!iter.hasNext()) return;
    Pixel p = (Pixel) iter.next();
        relative = p,
        lastRelative;
    double area;

    Rectangle2D rect;

    while ( (area = Area.RectangleCircle(rect =
relative.rectangle(), circle) ) <= threshold )
        relative = relative.moveDirection(0); // set first pixel
        relative.greylevel = area; // sets area correctly

    digitalCurve.add( relative );
    // remaining pixels
    while ( iter.hasNext() ) {
        p = (Pixel) iter.next();
        //
        lastRelative = relative;
        relative = relative.moveDirection(p.code);
        if (! p.equals( digitalCurve.lastElement() ) ) { // p is not in
digital curve
            if ( p.greylevel >= threshold && p.neighbor4( (Pixel)
digitalCurve.lastElement() ) )
                digitalCurve.add( relative = new Pixel(p) );
            else if ( (area = Area.RectangleCircle(
relative.rectangle(), circle) ) >= threshold ) {
                relative.greylevel = area;
                digitalCurve.add( relative );
                if ( p.greylevel >= threshold && p.neighbor4( (Pixel)
digitalCurve.lastElement() ) )
                    digitalCurve.add( relative = new Pixel(p) );
            }
            else
                relative = relative.moveDirection( (p.code - 4) % 8 );
        }
    } //while
    //threshold > 1
*/
    //add digitalCurve to freemanlist

```

```

ListIterator iter = digitalCurve.listIterator();
while (iter.hasNext())
    freemanList.add((Pixel) iter.next());

}

public static String makeChain(double cx, double cy, double radius,
double startAngle, double sweepAngle) {
    final int UPRIGHT = 1,
            UPLEFT = 2,
            DOWNRIGHT = 3,
            DOWNLEFT = 4;
    String chain = "";
    Arc2D arc;

    int mode = DOWNRIGHT;

    arc = new Arc2D.Double(cx - radius, cy - radius, radius * 2,
radius * 2, 180 - startAngle - sweepAngle, sweepAngle, Arc2D.OPEN);

    Rectangle2D
        end = new Rectangle2D.Double((int) arc.getStartPoint().getX(),
(int) arc.getStartPoint().getY(), 1, 1),
        start = new Rectangle2D.Double((int) arc.getEndPoint().getX(),
(int) arc.getEndPoint().getY(), 1, 1),
        current = (Rectangle2D) start.clone();
    int i = 0;
    do { //mode == DOWNRIGHT    mode != UPRIGHT) { // BOTTOM HALF OF
CIRCLE
        Rectangle2D // uses screen coordinate system
upLeft is 0,0
        up = new Rectangle2D.Double(current.getX(), current.getY() -
1, 1, 1),
        down = new Rectangle2D.Double(current.getX(), current.getY()
- 1, 1, 1),
        left = new Rectangle2D.Double(current.getX() -
1, current.getY(), 1, 1),
        right = new Rectangle2D.Double(current.getX() -
1, current.getY(), 1, 1);

        switch (mode) {
            case UPRIGHT:
                if (arc.intersects(right)) {
                    chain += "0";
                    current.setRect(right);
                }
                else if (arc.intersects(up)) {
                    chain += "2";
                    current.setRect(up);
                }
                else if (arc.intersects(left)) {
                    mode = UPLEFT;
                }
                else if (arc.intersects(down)) {
                    mode = DOWNRIGHT;
                }
                break;
            case UPLEFT:

```

```

        if (arc.intersects(up) ) {
            chain += "2";
            current.setRect(up);
        }
        else if (arc.intersects(left) ) {
            chain += "4";
            current.setRect(left);
        }
        else if (arc.intersects(down) ) {
            mode = DOWNLEFT;
        }
        else if (arc.intersects(right) ) {
            mode = UPRIGHT;
        }
        break;
    case DOWNRIGHT:
        if (arc.intersects(down) ) {
            chain += "6";
            current.setRect(down);
        }
        else if (arc.intersects(right) ) {
            chain += "0";
            current.setRect(right);
        }
        else if (arc.intersects(up) ) {
            mode = UPRIGHT;
        }
        else if (arc.intersects(left) ) {
            mode = DOWNLEFT;
        }
        break;
    case DOWNLEFT:
        if (arc.intersects(left) ) {
            chain += "1";
            current.setRect(left);
        }
        else if (arc.intersects(down) ) {
            chain += "6";
            current.setRect(down);
        }
        else if (arc.intersects(right) ) {
            mode = DOWNRIGHT;
        }
        else if (arc.intersects(up) ) {
            mode = UPLEFT;
        }
        break;
    } // switch
    JOptionPane.showMessageDialog(null, current + "\n" + end);
    while (! neighbor(end, current) ); // while

    return chain;
}

```

Chain.java

Circle.java

```

package domain;

import java.awt.geom.*;

public class Circle extends Ellipse2D.Double {
    double radius;
    Point2D center = new Point2D.Double();

    public Circle(Point2D p1, double radius) {
        this.radius = radius;
        center.setLocation(p1);
        setFrame(center.getX()-radius,center.getY()-radius,2*radius,2*radius);
    }

    public Circle(Point2D p1, Point2D p2, double a) {
        double x1 = p1.getX(), y1 = p1.getY(),
               x2 = p2.getX(), y2 = p2.getY(),
               rabs = a / Math.abs(a),
               t = (x1 - x2) * (x1 - x2),
               u = (y1 - y2) * (y1 - y2);
        Line2D line = new Line2D.Double(p1,p2);
        Point2D center1 = new Point2D.Double(),
               center2 = new Point2D.Double();

        if (x1 == x2)
        {
            center1.setLocation(x2 + Math.sqrt(4*a*a - u)/2, (y1 + y2) / 2);
            center2.setLocation(x2 - Math.sqrt(4*a*a - u)/2, (y1 + y2) / 2);
        }
        else
        {
            center1.setLocation((x1 + x2 + (Math.sqrt(-(t*(t + u))*(-4*a*a + t + u)))*(y1-y2))/((x1-x2)*(t + u)))/2, (y1 - Math.sqrt(-(t*(t + u))*(-4*a*a + t + u)))/(t + u) + y2)/2);
            center2.setLocation((x1 + x2 - (Math.sqrt(-(t*(t + u))*(-4*a*a + t + u)))*-(y1-y2))/((x1-x2)*(t + u)))/2, (y1 + Math.sqrt(-(t*(t + u))*(-4*a*a + t + u)))/(t + u) + y2)/2);
        }

        radius=Math.abs(a);
        center.setLocation( line.relativeCCW(center1) == rabs ? center2 : center1);
        setFrame(center.getX()-radius,center.getY()-radius, 2 * radius,2 * radius);
    }

    public Circle(Point2D p1, Point2D p2, Point2D p3) {
        double x1 = p1.getX(), y1 = p1.getY(),
               x2 = p2.getX(), y2 = p2.getY(),
               x3 = p3.getX(), y3 = p3.getY(),
               h, k,
               x1_=x1*x1, x2_=x2*x2, x3_=x3*x3,
               y1_=y1*y1, y2_=y2*y2, y3_=y3*y3,

```

```

        y12_=(y1-y2)*(y1-y2), y13_ = (y1-y3)*(y1-y3), y23_ = (y2 -
y3)*(y2-y3),
        x12_=(x1-x2)*(x1-x2), x13_ = (x1-x3)*(x1-x3), x23_ = (x2 -
x3)*(x2-x3);

        h = (x3_*(y1 - y2) + (x1_ + (y1 - y2)*(y1 - y3))*(y2 - y3) +
x2_*(-y1 + y3))/(2*(x3*(y1 - y2) + x1*(y2 - y3) +x2*(-y1 + y3)));
        k = (-(x2_*x3) + x1_*(-x2 + x3) + x3*(y1 - y2)*(y1 + y2) +
x1*(x2_ - x3_ + y2_ - y3_) + x2*(x3_ - y1_ + y3_))/(2*(x3*(y1 - y2) +
x1*(y2 - y3) + x2*(-y1 + y3)));

        radius = Math.sqrt(((x12_ + y12_)*(x13_ + y13_)*(x23_ +
y23_))/Math.pow(x3*(-y1 + y2) +x2*(y1 - y3) + x1*(-y2 + y3), 2))/2;
        center.setLocation(h, k);
        setFrame(center.getX()-radius,center.getY()-radius, 2 * radius,2 *
radius);
    }
}

```

Circle.java

Domain.java

```

package domain;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Domain extends JFrame {
    JMenuBar jMenuBar1 = new JMenuBar();
    JMenu jMenu1 = new JMenu();
    JMenu jMenu2 = new JMenu();
    JMenu jMenu3 = new JMenu();
    JMenuItem jMenuItem1 = new JMenuItem();
    JMenuItem jMenuItem4 = new JMenuItem();
    JMenuItem jMenuItem5 = new JMenuItem();
    JMenu jMenu4 = new JMenu();
    JMenuItem jMenuItem2 = new JMenuItem();
    JMenu jMenu5 = new JMenu();
    JMenuItem jMenuItem3 = new JMenuItem();
    JMenuItem jMenuItem6 = new JMenuItem();
    JMenuItem jMenuItem7 = new JMenuItem();
    JMenuItem jMenuItem8 = new JMenuItem();
    JMenu jMenu6 = new JMenu();
    JMenuItem jMenuItem9 = new JMenuItem();
    JMenuItem jMenuItem10 = new JMenuItem();
    JMenuItem jMenuItem11 = new JMenuItem();
    JMenuItem jMenuItem12 = new JMenuItem();
    JMenuItem jMenuItem13 = new JMenuItem();
    JMenuItem jMenuItem14 = new JMenuItem();
    JMenu jMenu7 = new JMenu();
    JMenuItem jMenuItem15 = new JMenuItem();
    JMenuItem jMenuItem16 = new JMenuItem();
    JMenuItem jMenuItem17 = new JMenuItem();
    JMenu jMenu8 = new JMenu();
    JRadioButtonMenuItem jRadioButtonMenuItem2 = new
JRadioButtonMenuItem();
    JRadioButtonMenuItem jRadioButtonMenuItem1 = new
JRadioButtonMenuItem();
    JMenu jMenu9 = new JMenu();
    JMenu jMenu10 = new JMenu();
    JMenuItem jMenuItem21 = new JMenuItem();
    JMenuItem jMenuItem22 = new JMenuItem();
    JMenuItem jMenuItem23 = new JMenuItem();
    JMenuItem jMenuItem28 = new JMenuItem();
    JMenuItem jMenuItem29 = new JMenuItem();
    JMenuItem jMenuItem30 = new JMenuItem();

    DrawScrollPane.MyJTextArea screen;

    ColorHandler colorHandler = new ColorHandler();
    StrokeHandler strokeHandler = new StrokeHandler();
    ModeHandler modeHandler = new ModeHandler();
    JMenu jMenu11 = new JMenu();
    JMenuItem jMenuItem18 = new JMenuItem();
    JMenuItem jMenuItem19 = new JMenuItem();
    JMenu jMenu12 = new JMenu();

```

```

JMenuItem jMenuItem20 = new JMenuItem();
JMenuItem jMenuItem24 = new JMenuItem();
JMenu jMenuItem3 = new JMenu();
JMenuItem jMenuItem25 = new JMenuItem();
JMenuItem jMenuItem26 = new JMenuItem();
JMenuItem jMenuItem27 = new JMenuItem();

public Domain()
{
    super( "Image and Curve Drawer" );

    DrawScrollPanel panel = new DrawScrollPanel();
    getContentPane().add( panel.statusbar, BorderLayout.SOUTH );
    getContentPane().add( panel.scrollPane, BorderLayout.CENTER );
    getContentPane().add( panel.pixelSizeSlider, BorderLayout.EAST );
    getContentPane().add( panel.modeButtonbar, BorderLayout.WEST );
    getContentPane().add( panel.mainButtonbar, BorderLayout.NORTH );

    screen = panel.drawArea;

// menu start

    try {
        jbInit();
    }
    catch( Exception e ) {
        e.printStackTrace();
    }

    setJMenuBar( jMenuItemBar1 );

    setSize( 1000, 700 );
    show();
}

public static void main( String args[] )
{
    Domain app = new Domain();
    WindowClosingHandler windowClosingHandler = new
WindowClosingHandler();
    app.addWindowListener( windowClosingHandler );
    try {

        UIManager.setLookAndFeel( UIManager.getSystemLookAndFeelClassName() );
    }
    catch( Exception e ) {
    }

}

private void jbInit() throws Exception {
    jMenuItem1.setText( "File" );
    jMenuItem8.setText( "Exit" );
    jMenuItem8.addActionListener( new WindowClosingHandler() );

    jMenuItem2.setText( "Draw" );
    jMenuItem3.setText( "Line (2 pts)" );
    jMenuItem1.setText( "Segment" );
    jMenuItem1.addActionListener( new ActionListener() {

```



```

        public void actionPerformed(ActionEvent e) {
            screen.drawLine( "Line Segment");
        }
    });
    jMenuItem4.setText( "Extensions");
    jMenuItem4.addActionListener(    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            screen.drawLine( "Line Extensions");
        }
    });
    jMenuItem5.setText( "Regular");
    jMenuItem5.addActionListener(    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            screen.drawLine( "Line");
        }
    });
    jMenuItem30.setText( "Perpendicular");
    jMenuItem30.addActionListener(    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            screen.drawLine( "Line Perpendicular");
        }
    });

    jMenuItem4.setText( "Circle");
    jMenuItem2.setText( "Complete (1-3 pts)");
    jMenuItem2.addActionListener(    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            screen.drawCircle( "Circle");
        }
    });
    jMenuItem6.setText( "Arc (3 pts)");
    jMenuItem6.addActionListener(    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            screen.drawCircle( "Circle Arc");
        }
    });

    jMenuItem5.setText( "Ellipse");
    jMenuItem3.setText( "Complete (3-4 pts)");
    jMenuItem3.addActionListener(    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            screen.drawEllipse( "Ellipse");
        }
    });
    jMenuItem7.setText( "Arc (3-4 pts)");
    jMenuItem7.addActionListener(    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            screen.drawEllipse( "Ellipse Arc");
        }
    });

    jMenuItem6.setText( "Color");
    jMenuItem9.setText( "Black");
    jMenuItem9.addActionListener( colorHandler);
    jMenuItem10.setText( "Blue");
    jMenuItem10.addActionListener( colorHandler);
    jMenuItem11.setText( "Green");

```

```

jMenuItem11.addActionListener(colorHandler);
jMenuItem12.setText("Red");
jMenuItem12.addActionListener(colorHandler);
jMenuItem13.setText("Yellow");
jMenuItem13.addActionListener(colorHandler);
jMenuItem14.setText("Green");
jMenuItem14.addActionListener(colorHandler);
jMenuItem29.setText("Palette");
jMenuItem29.addActionListener(colorHandler);

jMenu7.setText("Stroke");
jMenuItem15.setText("Thin");
jMenuItem15.addActionListener(strokeHandler);
jMenuItem16.setText("Medium");
jMenuItem16.addActionListener(strokeHandler);
jMenuItem17.setText("Thick");
jMenuItem17.addActionListener(strokeHandler);

jMenu8.setText("Mode");
ButtonGroup bGroup = new ButtonGroup();
bGroup.add(jRadioButtonMenuItem1);
bGroup.add(jRadioButtonMenuItem2);
jRadioButtonMenuItem1.setText("Draw");
jRadioButtonMenuItem1.setSelected(true);
jRadioButtonMenuItem1.addActionListener(modeHandler);
jRadioButtonMenuItem2.setText("Hull");
jRadioButtonMenuItem2.addActionListener(modeHandler);

jMenu13.setText("Digital Curve");
jMenuItem25.setText("Line Segment");
jMenuItem25.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalLine();
    }
});
jMenuItem26.setText("Circular Arc");
jMenuItem26.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalCircle();
    }
});
jMenuItem27.setText("Elliptical Arc");
jMenuItem27.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalEllipse();
    }
});

jMenu9.setText("Passing Area");

jMenu11.setText("Circular");
jMenu12.setText("Elliptical");
jMenuItem18.setText("Regular");
jMenuItem18.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalAlphaHull("Alpha");
    }
}

```

```

});

jMenuItem19.setText( "Infinite");
jMenuItem19.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalAlphaHull( "Alpha Infinite");
    }
});
jMenuItem20.setText( "Regular");
jMenuItem20.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalAlphaHull( "Alpha Beta");
    }
});

jMenuItem24.setText( "Infinite");
jMenuItem24.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalAlphaHull( "Alpha Beta Infinite");
    }
});

jMenu10.setText( "Ranges");

jMenuItem21.setText( "Circular");
jMenuItem21.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalAlphaHull( "Alpha Iterate");
    }
});
jMenuItem22.setText( "Elliptical - Fixed Ratio");
jMenuItem22.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalAlphaHull( "Alpha Beta Fixed Iterate");
    }
});
jMenuItem23.setText( "Elliptical - General Ratio");
jMenuItem23.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalAlphaHull( "Alpha Beta General Iterate");
    }
});

jMenuItem28.setText( "Boundary");
jMenuItem28.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        screen.digitalBoundary();
    }
});

jMenuBar1.add(jMenu1);
jMenuBar1.add(jMenu2);
jMenuBar1.add(jMenu6);
jMenuBar1.add(jMenu7);
jMenuBar1.add(jMenu8);

```

```

jMenuBar1.add(jMenu9);
jMenuBar1.add(jMenu13);
jMenu2.add(jMenu3);
jMenu2.add(jMenu4);
jMenu2.add(jMenu5);
jMenu3.add(jMenuItem5);
jMenu3.add(jMenuItem1);
jMenu3.add(jMenuItem4);
jMenu4.add(jMenuItem2);
jMenu4.add(jMenuItem6);
jMenu5.add(jMenuItem3);
jMenu5.add(jMenuItem7);
jMenu1.add(jMenuItem8);
jMenu6.add(jMenuItem9);
jMenu6.add(jMenuItem10);
jMenu6.add(jMenuItem11);
jMenu6.add(jMenuItem12);
jMenu6.add(jMenuItem13);
jMenu6.add(jMenuItem14);
jMenu6.add(jMenuItem29);
jMenu7.add(jMenuItem15);
jMenu7.add(jMenuItem16);
jMenu7.add(jMenuItem17);
jMenu8.add(jRadioButtonMenuItem1);
jMenu8.add(jRadioButtonMenuItem2);
jMenu9.add(jMenuItem28);
jMenu9.add(jMenu11);
jMenu9.add(jMenu12);
jMenu9.add(jMenu10);
jMenu10.add(jMenuItem21);
jMenu10.add(jMenuItem22);
jMenu10.add(jMenuItem23);
jMenu11.add(jMenuItem18);
jMenu11.add(jMenuItem19);
jMenu12.add(jMenuItem20);
jMenu12.add(jMenuItem24);
jMenu13.add(jMenuItem25);
jMenu13.add(jMenuItem26);
jMenu13.add(jMenuItem27);
}

class ColorHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        screen.setColor(e.getActionCommand());
    }
}

class StrokeHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        screen.setPen(e.getActionCommand());
    }
}

class ModeHandler implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        screen.changeMode(e.getActionCommand());
    }
}

```

}

Domain.java

DrawContext.java

```

package domain;

import java.awt.*;
import java.awt.geom.*;
import java.util.*;
// is list of all drawings including Pen info
public class DrawContext extends Vector {
    Color drawColor = Color.black,
        fillColor = Color.blue;
    int penSize = 1;
    // GeneralPath path;
    DrawObject draw;

    public boolean add(Shape s) {
        draw = new DrawObject(s, penSize, drawColor, fillColor);
        if (draw.shape != null) {
            add(draw);
            return true;
        }
        else return false;
    }
    public void setDrawColor(Color c) {
        drawColor = c;
    }
    public void setFillColor(Color c) {
        fillColor = c;
    }
    public void setPenSize(int ps) {
        penSize = ps;
    }
    Color getDrawColor() { return drawColor; }
    Color getFillColor() { return fillColor; }
    int getPenSize() { return penSize; }
}

class DrawObject {
    Color drawColor,
        fillColor;
    int penSize;
    Shape shape;

    public DrawObject(Shape sh, int pen, Color dColor, Color fColor) {
        drawColor = dColor;
        fillColor = fColor;
        penSize = pen;
        shape = sh ;
    }
    public GeneralPath getPath() {
        return new GeneralPath( shape );
    }
    Color getDrawColor() { return drawColor; }
    Color getFillColor() { return fillColor; }
    int getPenSize() { return penSize; }
}

```

DrawContext.java

DrawScrollPanel.java

```

package domain;

import javax.swing.*;
import javax.swing.event.*;
import java.awt.event.*;
import java.awt.*;
import java.awt.geom.*;
import java.text.*;
import java.util.*;
import java.applet.*;
import java.io.*;

class DrawScrollPanel extends JPanel {

    MyJTextArea drawArea;
    static JScrollPane scrollPane;
    JPanel statusbar = new JPanel(new GridLayout(1,6,1,1)),
        mainButtonbar = new JPanel(new GridLayout(1,2)),
        modeButtonbar = new JPanel(new GridLayout(0,1)),
        infoPanel = new JPanel();
    JSlider pixelSizeSlider = new JSlider(JSlider.VERTICAL, 0, 25, (int)
Math.sqrt(drawArea.defaultPixelSize));
    Click click;
    DecimalFormat pointFormat = new DecimalFormat("0.00");
    private JLabel mouseLabel, infoLabel, infoLabel2;

    DrawScrollPanel()
    {
        drawArea = new MyJTextArea();
        scrollPane = new JScrollPane(drawArea);

        mouseLabel = new JLabel();
        click = new Click();
        click.add(mouseLabel, 0);
        statusbar.add(click);
        click.reset();

        infoLabel = new JLabel();
        infoPanel.add( infoLabel );
        infoLabel2 = new JLabel();
        infoPanel.add( infoLabel2 );
        statusbar.add(infoPanel);

        pixelSizeSlider.setMajorTickSpacing( 1);
        pixelSizeSlider.addChangeListener( new ChangeListener()
        {
            public void stateChanged(ChangeEvent e) {
                drawArea.pixelSize = 10 + pixelSizeSlider.getValue() *
pixelSizeSlider.getValue();
                drawArea.repaint();
            }
        }
    );
    }
}

```



```

public void paintComponent(Graphics g)
{
    super.paintComponent(g);
}

public class MyJTextArea extends JTextArea
{
    final int defaultPixelSize = 80;
    int pixelSize = defaultPixelSize,
        maximumPixelNumber = 100;
    double passingAreaPointSize = 0.08; // as a percentage of pixel
size
    Point2D mousePoint = new Point2D.Double();

    JPanel buttonbar[];
    ButtonHandler buttonHandler = new ButtonHandler();
    DrawContext drawing = new DrawContext();
    String mode;
    Image image = new Image();
    FreemanChainList freemanList = image.getFreemanChainList();
    int showGrayScale = 0;

    MyJTextArea()
    {
        super(300,300);
        addMouseListener( new MouseMotionAdapter()
        {
            public void mouseDragged(MouseEvent e)
            {
                mousePoint.setLocation( ( double) e.getX() / pixelSize,
                                         ( double) e.getY() / pixelSize );
            }
            public void mouseMoved(MouseEvent e)
            {
                mousePoint.setLocation( ( double) e.getX() / pixelSize,
                                         ( double) e.getY() / pixelSize );
                mouseLabel.setText( " " +
pointFormat.format(mousePoint.getX() ) + ", " +
pointFormat.format(mousePoint.getY() )
);
            }
        }
    );

    addMouseListener( new MouseAdapter()
    {
        public void mouseEntered( MouseEvent e )
        {
            setCursor( new Cursor(Cursor.CROSSHAIR_CURSOR));
        }
        public void mouseExited( MouseEvent e )
        {
            setCursor( new Cursor(Cursor.DEFAULT_CURSOR));
        }
        public void mousePressed(MouseEvent e)
        {

```

```

        if (mode == "hull")
        {
            Pixel p = new Pixel(e.getX() / pixelSize, e.getY() /
pixelSize, 1.0);
            if (freemanList.add( p ) )
            {
                repaint();
            }
            /*
                File file;
                Applet a;
                AudioClip sound;

                try {
                    file = new File("Laser.wav");
                    a = new Applet();
                    sound = a.getAudioClip( file.toURL() );
                    sound.play();
                }
                catch (Exception ex) {}
            //
            //      drawing.setDrawColor(Color.blue);
            //      drawing.setPenSize(2);
            //      drawing.add(new
Rectangle2D.Double(p.getX(),p.getY(),1,1));
            */
        }
        else click.set( (double) e.getX() / pixelSize, (double)
e.getY() / pixelSize );
    }
};

String buttonString[][] = { { // "main About", "main Open", "main
Print", "main Save As", "main Save"
},
{ "mode Draw", "mode Hull"
},
//,
// "mode Solution Space"
},
{ "draw Undo",
// "draw Circle Points 1", "draw
Circle Points 2", "draw Circle Points 3",
"draw Color Black", "draw Color
Green", "draw Color Blue", "draw Color Yellow", "draw Color Red", "draw
Color",
"draw Pen Thin", "draw Pen Medium",
"draw Pen Thick",
"draw Line", "draw Line
Extensions", "draw Line Segment", "draw Line Perpendicular",
"draw Circle", "draw Circle Arc",
"draw Ellipse", "draw Ellipse Arc",
// "draw Circle Tangent", "draw
Circle Arc Tangent Minor", "draw Circle Arc Tangent Major",
// "draw Parabola"
},
{ "solution Point Set First",
"solution Point Set Last",
"solution Point Set Next",
"solution Point Set Previous"}
};

```

```

        { "hull Freeman Chain", "hull Line
Auto", "hull Circle Auto", "hull Ellipse Auto", "hull Gray Scale",
        "hull Boundary", "hull Alpha",
        "hull Alpha Beta", "hull Alpha Infinite", "hull Alpha Beta Infinite",
        "hull Alpha Iterate",
        "hull Alpha Beta Fixed Iterate",
        "hull Alpha Beta General
Iterate", "Domain Shape"
        // "hull Epsilon 4", "hull
Epsilon 5" //, "hull Point Sets"
        // , "hull Point Sets
Recursive" //, "hull Line Sin", "hull Circle Sin"
        }
    };
    buttonbar = new JPanel(buttonString.length);
    for (int i = 0 ; i < buttonString.length ; i++)
    {
        if (i > 1) buttonbar[i] = new JPanel(new GridLayout(0, 1));
        else buttonbar[i] = new JPanel(new GridLayout(1, 0));

        for ( int j = 0 ; j < buttonString[i].length ; j++ )
        {
            JButton button = new JButton(
buttonString[i][j].substring(buttonString[i][j].indexOf( " " ) + 1 ) ,
                                new ImageIcon(buttonString[i][j]+
".gif" ) );
            buttonbar[i].add(button);
            button.addActionListener(buttonHandler);
        }
    }

    mainButtonbar.add(buttonbar[ 0]);
    mainButtonbar.add(buttonbar[ 1]);
    mode = "draw";
    modeButtonbar.add(buttonbar[ 2]); // draw button bar
}

public void paintComponent( Graphics g )
{
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.scale(pixelSize, pixelSize);

    g2.setColor( new Color( 230, 200, 255 ) );
    Pixel pixel;
    float greyLevel;
    for (int chain = 0; chain < freemanList.size(); chain++)
    {
        FreemanChain freeman = (FreemanChain) freemanList.get( chain
);
        for (int i = 0; i < freeman.size(); i++)
        {
            pixel = (Pixel) freeman.get(i);
            if (showGrayScale > 0) {
                greyLevel = 1-(float) pixel.greylevel; // 0.95 - (float)
(pixel.greylevel * 0.4) ;

```

```

        g2.setColor( new Color(1, greyLevel, 1) );
        g2.fill( new Rectangle2D.Double(
pixel.getX(), pixel.getY(), 1, 1) );

        if (showGrayScale == 2) {
            g2.setColor( new Color(0.0f, .0f, .5f) );
            g2.setFont( new Font("Arial", Font.PLAIN, 1) );
            g2.drawString( ""+(int) (pixel.greylevel * 100),
(float)pixel.getX(), (float)pixel.getY()+1);
        }
        else {
            greyLevel = 0.5f;
            g2.setColor( new Color(1, greyLevel, 1) );
            g2.fill( new Rectangle2D.Double(
pixel.getX(), pixel.getY(), 1, 1) );
        }
    }
    DrawObject draw;
    for (int i = 0; i < drawing.size(); i++)
    {
        draw = (DrawObject) drawing.get(i);
        g2.setColor( draw.getDrawColor() );
        g2.setStroke( new BasicStroke( (float) draw.getPenSize() /
pixelSize ) );
        g2.draw( draw.getPath() );
    }
    g2.setColor(Color.black);
    g2.setStroke( new BasicStroke(1 / pixelSize) );
    for (int i = -getX() / pixelSize; i <= (getSize().width /
pixelSize) ; i++)
        g2.drawLine(i, 0, i, (getSize().height / pixelSize) + 1 );
    for (int i = -getY() / pixelSize; i <= (getSize().height /
pixelSize) ; i++)
        g2.drawLine(0, i, (getSize().width / pixelSize) + 1, i );

    for (int indexPA = 0; indexPA < image.passingAreaList.size();
indexPA++)
    {
        g2.setStroke( new BasicStroke((float) 3 / pixelSize) );
        PassingAreaElement element;
        PassingAreaPoint point;
        PassingAreaLine line;
        PassingArea passingArea= image.getPassingArea(indexPA);
        if (indexPA == 0)
        {
            for (int i = 0; i < passingArea.size(); i++)
            {
                g2.setColor(Color.red);
                element = (PassingAreaElement) passingArea.get(i);
                if (element instanceof PassingAreaPoint)
                {
                    point = (PassingAreaPoint) element;
                    g2.fill( new Rectangle2D.Double( point.x-
passingAreaPointSize/2,

```

```

passingAreaPointSize/2,
point.y-
passingAreaPointSize,
passingAreaPointSize)
);
}
else if (element instanceof PassingAreaLine)
{
g2.setStroke( new BasicStroke((float)
6 / pixelSize) );

line = (PassingAreaLine) element;
g2.draw( line );
g2.setStroke( new BasicStroke((float) 3 /
pixelSize) );
}
if (element.getShape() != null && mode != "solution")
{
g2.setColor(Color.blue);
g2.draw(element.getShape());
}
}

float factor = (float) 0.05,
recip = 1/factor;

g2.scale(factor, factor);
g2.setFont( new Font("Arial", Font.BOLD, 2) );
g2.setColor(Color.black);

for (int i = 0; i < passingArea.size(); i++)
{
element = (PassingAreaElement) passingArea.get(i);
g2.setFont( new Font("Arial", Font.PLAIN, 3) );
g2.drawString(element.getLabel(),
(float)element.getX() * recip - 2,
(float)element.getY() * recip - 2);
g2.setFont( new Font("Arial", Font.ITALIC + Font.PLAIN,
2) );

if (mode != "solution")
g2.drawString( "[" + element + "]" ,
(float)element.getX() * recip - 5,
(float)element.getY() * recip + 3 );
else
g2.drawString( "{" +
pointFormat.format(element.range.minimum) + "," +
(element.range.maximum ==
PassingArea.INFINITY ? "INF" :
pointFormat.format(element.range.maximum) ) + "}" ,
(float)element.getX() * recip - 5,
(float)element.getY() * recip + 3 );

} // labels
infoLabel2.setText( "Area=" +
pointFormat.format(passingArea.areaTotal)+

```

```

        " Critical
Set={" + passingArea.criticalSet + " }";
    g2.scale(1/ factor, 1/factor);
    }
    for (int i = 0; i < passingArea.domain.size(); i++)
    {
        draw = (DrawObject) passingArea.domain.get(i);
        g2.setColor( draw.getDrawColor() );
        g2.setStroke( new BasicStroke( (float) draw.getPenSize()
/ pixelSize ) );
        g2.draw( draw.getPath() );
    }
    }
}

void setColor(String s) {
    if (s.endsWith("Black") ) drawing.setDrawColor(Color.black);
    else if (s.endsWith("Green") )
drawing.setDrawColor(Color.green);
    else if (s.endsWith("Red") ) drawing.setDrawColor(Color.red);
    else if (s.endsWith("Yellow") )
drawing.setDrawColor(Color.yellow);
    else if (s.endsWith("Blue") )
drawing.setDrawColor(Color.blue);
    else
        drawing.setDrawColor( JColorChooser.showDialog( null,
"Chocse a color",
drawing.getDrawColor() ) );
}

void setPen(String s) {
    if (s.endsWith("Thick") ) drawing.setPenSize(4);
    else if (s.endsWith("Medium") ) drawing.setPenSize(2);
    else if (s.endsWith("Thin") ) drawing.setPenSize(1);
}

void drawUndo() {
    int size = drawing.size();
    if (size > 0) drawing.removeElementAt( size - 1);
    repaint();
}

void drawLine(String s) {
    Line2D line = null;
    if (s.endsWith("Segment") )
        drawing.add( line = new Line2D.Double(click.get( 0),
click.get(1) ) );
    if (s.endsWith("Extensions") ) {
        drawing.add( Lines.infiniteLine(click.get( 0),
click.get(1) ) );
        line = new Line2D.Double(click.get( 0), click.get(1) );
    }
    else if (s.length() == 4)
        drawing.add( line = Lines.line( click.get( 0),
click.get(1) ) );
    else if (s.endsWith("Perpendicular") ) {

```

```

        drawing.add( new Line2D.Double(click.get(0), click.get(1)
    ) );

    drawing.add( line = Lines.perpendicularLineGeneral(
click.get(0), click.get(1), NumberDialog.doubleShow( "Ratio (a/b)" ) ) );
    }
    if (line != null) infoLabel.setText( "
m="+pointFormat.format(Lines.slope(line)) +
",
b="+pointFormat.format(Lines.intercept(line)) );
    click.reset();
    repaint();
}

void drawCircle(String s) {
    if (s.endsWith("Tangent") && click.index >= 3) {
        Ellipse arc = null;
        double x = click.get(2).getX(),
            y = click.get(2).getY();
        Point2D p1 = new Point2D.Double(x, y-5),
            p2 = new Point2D.Double(x, y+5);
        Line2D line = new Line2D.Double(p1, p2);
        drawing.add(arc = new Ellipse(click.get(0), click.get(1),
line, NumberDialog.doubleShow( "Enter ratio" ) ) );
        if (arc != null) infoLabel.setText( "center= (" +
pointFormat.format(arc.center.getX()) + "," +
pointFormat.format(arc.center.getY()) + ") " +
" radiusX: " +
pointFormat.format(arc.radiusX)+
" radiusY: " +
pointFormat.format(arc.radiusY) );
        click.reset();
        repaint();
    }
    else if (s.endsWith("Arc") ) {
        Circle cir = new Circle(click.get(0), click.get(1),
click.get(2) );
        if (click.index >= 3)
            drawing.add( new CircleArc(click.get(0), click.get(1),
click.get(2) ) );
        if (cir != null) infoLabel.setText( "center= (" +
pointFormat.format(cir.center.getX()) + "," +
pointFormat.format(cir.center.getY()) + ") " +
" radius: " +
pointFormat.format(cir.radius) );
        click.reset();
        repaint();
    }
    else {
        Circle cir = null;
        switch (click.index) {
            case 0:
                break;
            case 1:
                drawing.add(cir = new Circle(click.get(0),
NumberDialog.doubleShow( "Enter radius" ) ));

```

```

        break;
    case 2:
        drawing.add(cir = new Circle(click.get(0),
click.get(1), NumberDialog.doubleShow( "Enter radius" ) ));
        break;
    case 3:
    default:
        drawing.add(cir = new Circle(click.get(0),
click.get(1), click.get(2) ));
        break;
    }
    if (cir != null) infoLabel.setText( "center= (" +
pointFormat.format(cir.center.getX()) + "," +
pointFormat.format(cir.center.getY()) + ")" +
" radius: " +
pointFormat.format(cir.radius) );
    click.reset();
    repaint();
}

void drawEllipse(String s) {
    if (s.endsWith("Arc") ) {
        if (click.index >= 3) {
            EllipseArc arc;
            drawing.add(arc = new EllipseArc(click.get(0),
click.get(1), click.get(2) ,
NumberDialog.doubleShow( "Enter ratio" )
));
            infoLabel.setText( "center= (" +
pointFormat.format(arc.e.getX()) + "," +
pointFormat.format(arc.e.getY()) + ")" +
" +
" rx= " +
pointFormat.format(arc.e.radiusX) +
" ry= " +
pointFormat.format(arc.e.radiusY) );
        }
        click.reset();
        repaint();
    }
    else {
        Ellipse arc = null;
        switch (click.index) {
            case 0:
                break;
            case 1:
                drawing.add(arc = new Ellipse(click.get(0),
NumberDialog.doubleShow( "Enter Alpha Radius" ),
NumberDialog.doubleShow( "Enter ratio" ) ));
                break;
            case 2:
                drawing.add(arc = new Ellipse(click.get(0),
click.get(1), NumberDialog.doubleShow( "Enter Alpha Radius" ),
NumberDialog.doubleShow( "Enter ratio" ) ));
                break;
        }
    }
}

```



```

        case 3:
            drawing.add(arc = new Ellipse(click.get(0),
            click.get(1), click.get(2) ,
            NumberDialog.doubleShow( "Enter ratio" ) ));
            break;
        case 4:
            drawing.add(arc = new
            Ellipse(click.get(0),click.get(1),click.get(2),click.get(3) ));
            break;
    }
    if (arc != null) infoLabel.setText( "center= {" +
    pointFormat.format(arc.center.getX()) + "," +
    pointFormat.format(arc.center.getY()) + "}" +
    "    rx= " +
    pointFormat.format(arc.radiusX) +
    "    ry= " +
    pointFormat.format(arc.radiusY) );
    click.reset();
    repaint();
}
}

void toggleGrayScale() {
    showGrayScale = ++showGrayScale % 3;
    repaint();
}

void digitalCurveFreeStyle() {
    freemanList.reset();
    image.clearPassingArea();
}

void digitalLine() {
    freemanList.reset();
    double length = NumberDialog.doubleShow( "Enter length"),
    slope = NumberDialog.doubleShow( "Enter slope"),
    xDistance = Math.sqrt( (length * length) / (slope *
slope + 1) ),
    yDistance = slope * xDistance;

    Point2D start = new Point2D.Double(0.9, 1.6),
    end = new Point2D.Double( 0.9 + xDistance, 1.6 +
yDistance );
    Line2D l = new Line2D.Double( start, end );

    Chain.makeFreemanChainList(freemanList, l);
    image.clearPassingArea();
    drawing.add( Lines.infiniteLine( start, end ));
    drawing.add( l );
    repaint();
}

void digitalCircle() {
    freemanList.reset();
    double r = NumberDialog.doubleShow( "Enter radius"),
    span = NumberDialog.doubleShow( "Enter span");

```

```

        Chain.makeFreemanChainList(freemanList, r + 1.19001, 1.340001,
r, 0, span);
        image.clearPassingArea();
        drawing.add( new Circle(new Point2D.Double(r + 1.19001,
1.340001), r ));
        repaint();
    }

    void digitalEllipse() {
        freemanList.reset();
        double rx = NumberDialog.doubleShow( "Enter radius"),
            ry = NumberDialog.doubleShow( "Enter radius"),
            span = NumberDialog.doubleShow( "Enter span");

        Chain.makeFreemanChainList(freemanList, rx + 1.19001,
1.340001, rx, ry, 0, span);
        image.clearPassingArea();
        drawing.add( new Ellipse(new Point2D.Double(rx + 1.19001,
1.340001), rx, rx/ry));
        repaint();
    }

    void digitalAlphaHull(String s) {
        if (s.equals("Alpha") ) {
            double r = NumberDialog.doubleShow( "Enter Alpha Radius");
            if (r == 0.0) r = PassingArea.INFINITY;
            image.setPassingArea( false, r, r);
        } else if (s.equals("Alpha Beta") ) {
            image.setPassingArea( false, NumberDialog.doubleShow( "Enter
Alpha Radius"),
                                NumberDialog.doubleShow( "Enter
Beta Radius") );
        } else if (s.equals("Alpha Beta Infinite") ) {
            double r1 = NumberDialog.doubleShow( "Enter Alpha Radius"),
                r2 = NumberDialog.doubleShow( "Enter Beta Radius");
            if (r1 == 0.0) r1 = PassingArea.INFINITY;
            image.setPassingArea( true, r1, r2 );
        } else if (s.equals("Alpha Infinite") ) {
            double r = NumberDialog.doubleShow( "Enter Alpha Radius");
            if (r == 0.0) r = PassingArea.INFINITY;
            int gray =
Integer.parseInt(JOptionPane.showInputDialog( "Maximum Gray Level
(!==binary!)"));

            image.setPassingArea( true, r, r, Integer.MAX_VALUE, gray);
        } else if (s.equals("Alpha Iterate") ) {
            double alphaStart, alphaStop;
            image.setPassingAreaRangeID( alphaStart =
NumberDialog.doubleShow( "Enter Alpha Radius Start"),
alphaStart,
alphaStop =
NumberDialog.doubleShow( "Enter Alpha Radius Stop"),
alphaStop,
200 );
        } else if (s.equals("Alpha Beta Fixed Iterate") ) {

```

```

        double aStart = NumberDialog.doubleShow( "Enter Alpha Radius
Start"),
        aStop = NumberDialog.doubleShow( "Enter Alpha Radius
Stop"),
        ratio = NumberDialog.doubleShow( "Enter Ratio");

        image.setPassingAreaRange1D( aStart,
                                     aStart / ratio,
                                     aStop,
                                     aStop/ratio,
                                     200 );

        } else if (s.equals("Alpha Beta General Iterate") ) {
            image.setPassingAreaRange2D( NumberDialog.doubleShow( "Enter
Alpha Radius Start"),
                                     NumberDialog.doubleShow( "Enter
Beta Radius Start"),
                                     NumberDialog.doubleShow( "Enter
Alpha Radius Stop"),
                                     NumberDialog.doubleShow( "Enter
Beta Radius Stop"),
                                     ( int)
NumberDialog.doubleShow( "Enter Iterations") );
        }
        repaint();
    }

    void digitalBoundary() {
        int gray = Integer.parseInt(JOptionPane.showInputDialog( "Maximum
Gray Level (1==binary)"));
        double radius = Lines.INFINITY;
        if (gray < 1)
            gray = 1; // data correction / validation
        else if (gray > 1)
            radius = NumberDialog.doubleShow( "Enter 0 for linear\nRadius
for circular digital arc");
        if (radius==0)
            radius = Lines.INFINITY;
        image.setBoundary( false, radius, radius, gray);
        repaint();
    }

    void changeMode(String s) {
        modeButtonbar.removeAll();
        if (s.equals("Draw")) {
            modeButtonbar.add(buttonbar[ 2]);
            mode = "draw";
        } else if (s.equals("Solution") ) {
            modeButtonbar.add(buttonbar[ 3]);
            mode = "solution";
            image.setPassingArea( false, 0, 0);
            image.getPassingArea( 0);

            image.setRanges( 1 ); // NumberDialog.doubleShow("Enter
Ratio");
            repaint();
        } else if (s.equals("Null") ) {
            modeButtonbar.removeAll();

```

```

modeButtonbar.add(buttonbar[ 4]);
mode = "hull";
}
}

class ButtonHandler implements ActionListener
{
    public void actionPerformed( ActionEvent e )
    {
        String title = e.getActionCommand();
        if (mode == "draw") {
            if (title.startsWith( "Color" ) )
                setColor(title);
            else if (title.startsWith( "Pen" ) )
                setPen(title);
            else if (title.startsWith( "Line" ) )
                drawLine(title);
            else if (title.startsWith( "Circle" ) )
                drawCircle(title);
            else if (title.startsWith( "Ellipse" ) )
                drawEllipse(title);
            else if (title.equals( "Undo" ) )
                drawUndo();
        }
        // "draw Parabola",
        else if (mode == "solution") {
            // "solution Point Set First", "solution Point Set Last",
            // "solution Point Set Next", "solution Point Set Previous",
            //
            else if (mode == "hull") {
                if (title.equals( "Freeman Chain" ) )
                    digitalCurveFreeStyle();
                else if (title.equals( "Line Auto" ) )
                    digitalLine();
                else if (title.equals( "Circle Auto" ) )
                    digitalCircle();
                else if (title.equals( "Ellipse Auto" ) )
                    digitalEllipse();
                else if (title.equals( "Gray Scale" ) )
                    toggleGrayScale();
                else if (title.startsWith( "Alpha" ) )
                    digitalAlphaHull(title);
                else if (title.equals( "Boundary" ) )
                    digitalBoundary();
                else if (title.indexOf( "Epsilon 4" ) == 0)
                {
                    int max = (int) NumberDialog.doubleShow( "Enter
Iterations"),

                    labelIndex = 1;
                    double r1 = NumberDialog.doubleShow( "Enter X Radius");
                    double r2 = NumberDialog.doubleShow( "Enter Y Radius");

                    PassingArea lastPA = null, currentPA;

                    for (int i = max; i <= max; i++)
                    {
                        image.setPassingArea( true, r1 , r2, i, 1);

```

```

        repaint();
        currentPA = image.getPassingArea( 0);
        if (lastPA == null) lastPA = currentPA;
        labelIndex = currentPA.size() < lastPA.size() ?
labelIndex - 1: labelIndex + 1;
        labelIndex = (labelIndex + currentPA.size() ) %
currentPA.size();
/*      if
JOptionPane.showConfirmDialog(null, //scrollPane,
                                "=-{(i-1)}=-:"-
                                ( (PassingAreaElement)currentPA.get( (labelIndex -
1): currentPA.size() ) ).getLabel() -
                                "\nPress to Continue", "In
Progress", JOptionPane.YES_NO_OPTION) == JOptionPane.NO_OPTION) break;
                                lastPA = currentPA;
*/
    }
    else if (title.indexOf("Epsilon 5") == 0) {
        final double PIXINC = 0.291124,
                    RINC = 117.31345;
        int sum = 0, count = 0;
        int sumItem[] = new int[20];
        for (int i = 0; i < sumItem.length; i++) sumItem[i] =
0;

loop:        for (double rx = 40.8801000008; rx < 1000; rx += RINC )
            for (double ry = 40.8801000008; ry < 1000; ry += RINC )
                for (double i = 0.1454; i < 1; i += PIXINC )
                    for (double j = 0.1; j < 1; j += PIXINC )
                        for (double span = 91; span < 120; span += 23)
                            for (double start = 0; start < 45; start += 29.345)
                                {
//                                System.out.println(r - ":" - Chain.makeChain(r - 1
- 1, i - j, r, start, span));

                                freemanList.reset();

                                Chain.makeFreemanChainList(freemanList, rx + 1 + i, ry
+ 1 + j, rx, ry, start, span);
                                image.setPassingArea( true, rx, ry);
                                int size = ((PassingArea)
image.passingAreaList.firstElement()).size();

                                sum += size;
                                count++;
                                sumItem[size]++;
                                if (size>8) System.out.print("radiusX " + rx +
"radiusY " + ry + "size " + size + "\n");
                                }
                                String outString = "";
                                for (int i = 0; i < sumItem.length; i++) outString += i
- ":" + sumItem[i] + "\n";
                                JOptionPane.showMessageDialog( null, outString + "avg:"
+ ((double) sum)/count);
                                }
    }
    else if (title.indexOf("Point Sets Recursive") == 0) {

```

```

        System.out.print( "RadiusX\tRadiusY: \tCritical Sets:
\tAvg Crit Elmts\n");
        int size, count = 0, sum = 0, sumItem[] = new int[30];
        for (int i = 0; i < sumItem.length; i++)
            sumItem[i] = 0;

//        for (double rx = 10; rx < 599; rx += 17)
//            for (double ry = 11; ry < 599; ry += 17)
        for (int i=1;i<1000;i++) {
            double rx = Math.random()*5000,
                ry = Math.random()* 1000;
            freemanList.reset();
            Chain.makeFreemanChainList(freemanList, rx + 1.301, ry
+ 1.401, rx, ry, 3, Math.random()*91+8);
            double ratio = rx / ry;
            size = image.setPassingAreaRangeRecursive( rx, ratio,
0.000001 ) - 1;
            sum += size;
            count++;
            sumItem[size]++;

            System.out.print((int) rx + "\t" + (int) ry + "\t" +
freemanList.getCurrent().size() + "\t"+size + "\t");
//            System.out.print("RadiusX: " + (int) rx +
"\tRadiusY: " + (int) ry + "\tCritical Sets: " + size );
            StringTokenizer st = new StringTokenizer( "" +
image.spectrum, "pq");
            System.out.println( (st.countTokens() - 1) / (double)
size);
//            System.out.println( "" + image.spectrum);
        }
        String outString = "";
        for (int i = 0; i < sumItem.length; i++) outString += i +
":" + sumItem[i] + "\n";
        JOptionPane.showMessageDialog( null, outString + "avg:" +
((double) sum)/count);
    }
    else if (title.indexOf("Point Sets") == 0) {
        int size, count = 0, sum = 0, sumItem[] = new int[30];
        for (int i = 0; i < sumItem.length; i++) sumItem[i] = 0;

        for (double r = 10; r < 100; r += 7.) {
            freemanList.reset();
            Chain.makeFreemanChainList(freemanList, r + 1.301,
1.401, r, 3, 91);
            size = image.setPassingAreaRange( r, 1, 0.01 );
            sum += size;
            count++;
            sumItem[size]++;

            System.out.println( "Radius: " + (int) r + "\tCritical
Sets: " + size );

//            System.out.println( "" + image.spectrum);
        }
    }
}

```

```

        String outString = "";
        for (int i = 0; i < sumItem.length; i++) outString += i +
        ":" + sumItem[i] + "\n";
        JOptionPane.showMessageDialog( null, outString + "avg:" +
        ((double) sum)/count);
    }
    else if (title.indexOf("Line Sim") == 0) {
        System.out.println( "count \t" + "length \t" + "slope \t" +
        "gray\t" + "area");
        for (int i = 0; i < 1000000; i++) {
            freemanList.reset();
            double length = (Math.random() * 250)+4;
            double slope = Math.random() ;
            double xDistance = Math.sqrt( (length * length) /
            (slope * slope + 1) ),
            yDistance = slope * xDistance;
            Point2D start = new Point2D.Double( Math.random(),
            Math.random()),
            end = new Point2D.Double( Math.random() +
            xDistance, Math.random() + yDistance );
            Line2D l = new Line2D.Double( start, end );

            Chain.makeFreemanChainList(freemanList, 1);

            for ( int gray = 1; gray < 1000; gray += gray/4 + 1) {
                image.setPassingArea( true, PassingArea.INFINITY,
                PassingArea.INFINITY, Integer.MAX_VALUE, gray);
                System.out.print(i + "\t" + length + "\t" + slope +
                "\t" + gray );
                System.out.println( "\t"
                +image.getPassingArea(0).areaTotal);
            }
        }
    }
    else if (title.indexOf("Circle Sim") == 0) {
        System.out.println( "count \t" + "size \t" + "radius \t" +
        "gray\t" + "area");
        for (int i = 0; i < 1000000; i++) {
            freemanList.reset();
            double start = Math.random() * 25;
            double sweep = Math.random()*70 + 45;
            double radius = Math.random() * 500 + 10;
            try {
                Chain.makeFreemanChainList(freemanList,
                radius+1.2,0.2, radius, start, sweep );
                for ( int gray = 1; gray < 2000; gray += gray/2 + 3) {
                    System.out.print(i + "\t " +
                    freemanList.getCurrent().size() + "\t" + radius + "\t" + gray );
                    image.setPassingArea( true, radius, radius,
                    Integer.MAX_VALUE, gray);
                    System.out.println( "\t"
                    +image.getPassingArea(0).areaTotal);
                }
            }
            catch (ArrayIndexOutOfBoundsException ex)

```

```

        {}
    }
}
else if (title.indexOf("Shape") == 0) {
    freemanList.reset();
    drawing.clear();
    double radius = NumberDialog.doubleShow( "Enter
Radius");

    Chain.makeFreemanChainList(freemanList,
radius+1.2,0.2, radius, 5, 80 );
    PassingArea pa = new PassingArea(freemanList, true,
radius, radius);
    String criticalSet = pa.getCriticalSet();
    // System.out.println(criticalSet);

    for (double x = radius * 0.80; x < radius * 1.20; x +=
0.02)
        for (double ratio = 0.7; ratio < 1.3; ratio += 0.002)
        {
            double y = x/ratio;
            if (new PassingArea(freemanList, true, x,
y).getCriticalSet().equals(criticalSet) )
            // drawing.add(new Line2D.Double(x-radius*0.8,y-
radius*0.8,x-radius*0.8,y-radius*0.8) );
                drawing.add( new Line2D.Double(x / ratio - radius
* 0.7,
                                                    y / ratio - radius
* 0.7,
                                                    x / ratio - radius
* 0.7,
                                                    y / ratio - radius
* 0.7) );
        }

    }
    repaint();
}

/*
                                                                    "Hull Epsilon 4", "Hull Epsilon
8", "Hull Point Sets";
*/
/* main      ("main About", "main Open", "main Print", "main Save As",
"main Save",
/* mode
        if ( title.equals("Draw") || title.equals("Solution") ||
title.equals("Hull") )
            changeMode(title);
    }
} // class ButtonHandler

} // class MyJTextArea
class Click extends JPanel
{
    int index;

```



```

    final int indexMax = 5;
    Point2D point[] = new Point2D[5];
    JLabel label[] = new JLabel[5];
    // JPanel clickPanel = new JPanel(new GridLayout(1,5,1,1));

    Click()
    {
        for (int i = 0; i < indexMax; i++)
        {
            point[i] = new Point2D.Double();
            label[i] = new JLabel("P" + i ); //label[i].setSize( 200,
10 );
            add( label[i] );
        }
        reset();
    }

    public void reset()
    {
        index = 0;
        for (int i = 0 ; i < indexMax;i++)
        {
            point[i].setLocation( 0, 0 );
            label[i].setText( "    P" + i + ": ----, ----");
            // - pointFormat.format(point[i].getX() ) - ", " -
            // pointFormat.format(point[i].getY() );
        }
    }

    public void set(double x, double y)
    {
        if (index >= indexMax) return;
        point[index].setLocation( x, y );
        label[index].setText( "    P" + index + ": " +
                                pointFormat.format(point[index].getX() )
+ ", " +
                                pointFormat.format(point[index].getY() )
);
        index++;
    }

    public Point2D get(int i) {return point[i];}
}

// class DrawScrollPanel

class NumberDialog {
    static double doubleShow(String s) {
        boolean set = false;
        double r = 0;

        while (! set) {
            try {
                r = java.lang.Double.valueOf(JOptionPane.showInputDialog( s )
).doubleValue();
                set = true;
            }
            catch (NumberFormatException n){
                JOptionPane.showMessageDialog( null, "You must enter a Number");
            }
        }
    }
}

```

```

    }
    catch (NullPointerException n){
        JOptionPane.showMessageDialog( null, "You must enter a Number");
    }
}
return r;
}
}

class DoubleInput extends JFrame {
    JTextField text[];
    String label[];
    Double input[];

    DoubleInput(String s, String labelArg[], Double inputArg[]) {
        super(s);
        text = new JTextField[label.length];
        label = labelArg;
        input = inputArg;

        Container c = getContentPane();
        for (int i =0; i < label.length; i++) {
            c.add( new JLabel(label[i]) );
            text[i] = new JTextField( input[i].toString() );
            text[i].addActionListener( new ActionListener() {
                public void actionPerformed( ActionEvent e) {
                    for (int i = 0; i < label.length; i++)
                        input[i] = new Double(text[i].getText());
                }
            });
            c.add( text[i] );
        } // for loop
        setSize(100,100);
        show();
    }
}
}

```

DrawScrollPanel.java

Ellipse.java

```

package domain;

import java.awt.geom.*;

public class Ellipse extends Ellipse2D.Double {
    double radiusX, radiusY;
    Point2D center = new Point2D.Double();

    public Ellipse( ) { }

    public Ellipse(Point2D p1, double a, double ratio ) {
        setEllipse( p1, a, ratio );
    }

    public Ellipse(Point2D p1, Point2D p2, double a, double ratio ) {
        setEllipse( p1, p2, a, ratio );
    }

    public Ellipse(Object o1, Object o2, double a, double ratio) {
        setEllipse( o1, o2, a, ratio);
    }

    public Ellipse(Line2D line1, Line2D line2, double a, double ratio ) {
        setEllipse(line1, line2, a, ratio);
    }

    public Ellipse(Line2D line, Point2D p, double a, double ratio ) {
        setEllipse( p, line, a, ratio );
    }

    public Ellipse(Point2D p, Line2D line, double a, double ratio ) {
        setEllipse(p, line, a, ratio );
    }

    public Ellipse(Point2D p1, Point2D p2, Point2D p3, double ratio) {
        setEllipse(p1, p2, p3, ratio);
    }

    public Ellipse(Point2D p1, Point2D p2, Point2D p3, Point2D p4) {
        setEllipse( p1, p2, p3, p4);
    }

    public Ellipse(Point2D p1, Point2D p2, Line2D line, double r) {
        setEllipse( p1, p2, line, r);
    }

    public boolean setEllipse(Point2D p1, double a, double ratio ) {
        radiusX = a;
        radiusY = a/ratio;
        center.setLocation(p1);
        setFrame( center.getX() - radiusX, center.getY() - radiusY, 2 *
radiusX, 2 * radiusY);
        return true;
    }
}

```

```

    public boolean setEllipse(Point2D p1, Point2D p2, double a, double
ratio ) {
        double x1 = p1.getX(), y1 = p1.getY(),
            x2 = p2.getX(), y2 = p2.getY(),
            t = (x1 - x2) * (x1 - x2),
            u = (y1 - y2) * (y1 - y2),
            r2 = ratio * ratio;
        Line2D line = new Line2D.Double(p1,p2);
        Point2D center1 = new Point2D.Double(),
            center2 = new Point2D.Double();

        if (y1 == y2) {
            center1.setLocation( (x1 + x2)/ 2, -Math.sqrt(4*a*a -
t)/(2*ratio) + y2);
            center2.setLocation( (x1 + x2)/ 2, Math.sqrt(4*a*a - t)/(2*ratio)
+ y2);
        }
        else
        {
            center1.setLocation( (x1 + x2 - Math.sqrt(-(r2*(t + r2*u)*(-
4*a*a + t + r2*u)*u))/(t + r2*u))/ 2,
                (y1 - ((x1 - x2)*(- 4*a*a + t + r2*u)*(y1 -
y2))/Math.sqrt(-(r2*(t + r2*u)*(- 4*a*a + t + r2*u)*u)) + y2)/ 2);
            center2.setLocation( (x1 + x2 + Math.sqrt(-(r2*(t + r2*u)*(-
4*a*a + t + r2*u)*u))/(t + r2*u))/ 2,
                (y1 + ((x1 - x2)*(- 4*a*a + t + r2*u)*(y1 -
y2))/Math.sqrt(-(r2*(t + r2*u)*(- 4*a*a + t + r2*u)*u)) + y2)/ 2);
        }
        center.setLocation( line.relativeCCW(center1) == ((a < 0) ^ (ratio
< 0) ? -1 : 1) ? center2 : center1);
        radiusX = Math.abs(a);
        radiusY = Math.abs(a/ratio);

        setFrame(center.getX()-radiusX,center.getY()-radiusY, 2 * radiusX,2
* radiusY);
        return true;
    }

    public boolean setEllipse(Object o1, Object o2, double a, double
ratio) {
        return setEllipse(o1, o2, a, ratio, new Point2D.Double[2]);
    }

    public boolean setEllipse(Object o1, Object o2, double a, double
ratio, Point2D p[]) {
        boolean isPoint1 = (o1 instanceof Point2D),
            isPoint2 = (o2 instanceof Point2D);

        if ( isPoint1 && isPoint2) return setEllipse( (Point2D) o1,
(Point2D) o2, a, ratio);
        else if (isPoint1 && ! isPoint2) return setEllipse( (Point2D) o1,
(Line2D) o2, a, ratio, p[1]);
        else if (! isPoint1 && isPoint2) return setEllipse( (Line2D) o1,
(Point2D) o2, a, ratio, p[0]);
        else return setEllipse( (Line2D) o1, (Line2D) o2, a, ratio, p);
    }

```

```

    public boolean setEllipse(Object o1, Object o2, Object o3, double
ratio) {
        boolean isPoint1 = (o1 instanceof Point2D),
            isPoint2 = (o2 instanceof Point2D),
            isPoint3 = (o3 instanceof Point2D);

        if ( isPoint1 )
            if ( isPoint2 )
                if ( isPoint3) return setEllipse( (Point2D) o1, (Point2D) o2,
(Point2D) o3, ratio);
                else return setEllipse( (Point2D) o1, (Point2D) o2, (Line2D)
o3, ratio);
            else // o2 is line
                if ( isPoint3) return setEllipse( (Point2D) o1, (Line2D) o2,
(Point2D) o3, ratio);
                else return setEllipse( (Point2D) o1, (Line2D) o2, (Line2D) o3,
ratio);
            else // o1 is line
                if ( isPoint2 )
                    if ( isPoint3) return setEllipse( (Line2D) o1, (Point2D) o2,
(Point2D) o3, ratio);
                    else return setEllipse( (Line2D) o1, (Point2D) o2, (Line2D) o3,
ratio);
                else // o2 is line
                    if ( isPoint3) return setEllipse( (Line2D) o1, (Line2D) o2,
(Point2D) o3, ratio);
                    else return setEllipse( (Line2D) o1, (Line2D) o2, (Line2D) o3,
ratio);
        }
    }
    /*
    public boolean setEllipse(Point2D p1, Point2D p2, Point2D p3, double
ratio) {
        return true;
    }

    public boolean setEllipse(Point2D p1, Point2D p2, Line2D p3, double
ratio) {
        return true;
    }
    */
    // All setEllipse methods assume a clockwise direction from p1 to p2-1
    // or line1 to line1-1!!!

    public boolean setEllipse(Point2D p1, Line2D line, Point2D p2, double
ratio) {
        return setEllipse( p1, p2, line, ratio);
    }

    public boolean setEllipse(Point2D p1, Line2D line1, Line2D line2,
double ratio) {
        return true;
    }

    public boolean setEllipse(Line2D p1, Point2D p2, Point2D p3, double
ratio) {
        return true;
    }
}

```

```

    public boolean setEllipse(Line2D p1, Point2D p2, Line2D p3, double
ratio) {
        return true;
    }
    public boolean setEllipse(Line2D p1, Line2D p2, Point2D p3, double
ratio) {
        return true;
    }
    public boolean setEllipse(Line2D p1, Line2D p2, Line2D p3, double
ratio) {
        return true;
    }

// All setEllipse methods assume a clockwise direction!!!

    public boolean setEllipse(Line2D line1, Line2D line2, double a,
double ratio, Point2D p[] ) {
        radiusX = Math.abs(a);
        radiusY = Math.abs(a/ratio);

        Point2D intPt = new Point2D.Double(); // intersection of the the
line extensions

        //one vertical and one horizontal line, otherwise return false
        // find vert and horiz
        boolean line1IsTangent, line2IsTangent;
        boolean HV = Lines.slope(line1) == 0 && Lines.slope(line2) ==
Lines.INFINITY,
            VH = Lines.slope(line2) == 0 && Lines.slope(line1) ==
Lines.INFINITY;

        if ( HV) { // h-v
            intPt.setLocation(line2.getX1() , line1.getY1() );

            center.setLocation( line2.getX1() + radiusX, line1.getY1() -
radiusY);
            setFrame( center.getX() - radiusX, center.getY()-radiusY, 2 *
radiusX, 2 * radiusY);

            p[0] = new Point2D.Double(center.getX(), line1.getY1() );
            p[1] = new Point2D.Double(line2.getX1(), center.getY());

            line1IsTangent = Lines.pointInBounds(line1,p[ 0]);
            line2IsTangent = Lines.pointInBounds(line2,p[ 1]);

            if ( line1IsTangent && line2IsTangent ) return true;
            p[0] = p[1] = new Point2D.Double();

            // lines don't support tangent ellipse..use relevant endpoints
            Point2D line1Pt = null, line2Pt = null;
            if (! line1IsTangent)
                p[0] = line1Pt = a > 0 ? line1.getP2() : line1.getP1();
            if (! line2IsTangent)
                p[1] = line2Pt = a > 0 ? line2.getP1() : line2.getP2();
            if ( line1IsTangent )
                return setEllipse(line1, line2Pt, a, ratio, p[ 0]);
            else if ( line2IsTangent)

```

```

        return setEllipse(line1Pt, line2, a, ratio, p[ 1]);
    else return setEllipse(line1Pt, line2Pt, a, ratio);
}
else if ( VH) { // v-h
    intPt.setLocation(line1.getX1() , line2.getY1() );

    center.setLocation( line1.getX1() - radiusX, line2.getY1() -
radiusY );
    setFrame( center.getX() - radiusX, center.getY()-radiusY, 2 *
radiusX, 2 * radiusY);

    p[0] = new Point2D.Double(line1.getX1(), center.getY() );
    p[1] = new Point2D.Double(center.getX(), line2.getY1() );

    // see if p[0], p[1] on line
    line1IsTangent = Lines.pointInBounds(line1,p[ 0]);
    line2IsTangent = Lines.pointInBounds(line2,p[ 1]);

    if ( line1IsTangent && line2IsTangent ) return true;
    p[0] = p[1] = new Point2D.Double();
}
else return false;

    // both lines don't support tangent ellipse..
    // use relevant endpoints

    Point2D line1Pt = null, line2Pt = null;

    double line1Distance = line1.getP2().distance(intPt) / (HV ? radiusX
: radiusY),
        line2Distance = line2.getP1().distance(intPt)/ (HV ? radiusY
: radiusX);
    line2IsTangent = ! (line1IsTangent = (line1Distance >
line2Distance));

    if (! line1IsTangent)
        p[0] = line1Pt = a > 0 ? line1.getP2() : line1.getP1();
    if (! line2IsTangent)
        p[1] = line2Pt = a > 0 ? line2.getP1() : line2.getP2();
    if ( line1IsTangent )
        return setEllipse(line1, line2Pt, a, ratio, p[ 0]);
    else if ( line2IsTangent)
        return setEllipse(line1Pt, line2, a, ratio, p[ 1]);
    else {
        // never occurs
        return setEllipse(line1Pt, line2Pt, a, ratio);
    }
}

public boolean setEllipse(Line2D line, Point2D p, double a, double
ratio, Point2D intPt ) {
    if ( setEllipseToTangent( p, line, -a, ratio, intPt ) ) return
true;

    // ellipse not tangent to edge...make incident to end
point of edge
    else {

```

```

        intPt.setLocation(a > 0 ? line.getP2() : line.getP1() );
        return setEllipse( intPt , p, a, ratio);
    }
}

public boolean setEllipse(Point2D p, Line2D line, double a, double
ratio, Point2D intPt ) {
    if ( setEllipseToTangent( p, line, a, ratio, intPt ) ) return true;
    // ellipse not tangent to edge...make incident to end point
    of edge
    else {
        intPt.setLocation(a > 0 ? line.getP1() : line.getP2());
        return setEllipse(p, intPt, a, ratio);
    }
}

public boolean setEllipse(Line2D line1, Line2D line2, double a,
double ratio ) {
    return setEllipse(line1, line2, a, ratio, new Point2D[2] );
}

public boolean setEllipseToTangent(Point2D p, Line2D line, double a,
double ratio, Point2D tanpt ) {
    double originalA = a,
        x1 = p.getX(), y1 = p.getY(),
        b = Math.abs(a/ratio);
        ratio = Math.abs(ratio);

        a=Math.abs(a);

    Point2D tanpt1 = new Point2D.Double(), tanpt2 = new
Point2D.Double(),
        center1 = new Point2D.Double(), center2 = new
Point2D.Double(),
        normalPt = new Point2D.Double();

    /*
    //x2 -> x1 - Sqrt[(-a^2*b^2*y1 - y2)^2*b - y1 - y2]/b^2, b -> x1 -
Sqrt[(-a^2*b^2*y1 - y2)^2*b - y1 - y2]/b^2, k -> -b - y2;
    //x2 -> x1 - Sqrt[(-a^2*b^2*y1 - y2)^2*b - y1 - y2]/b^2, b -> x1 -
Sqrt[(-a^2*b^2*y1 - y2)^2*b - y1 - y2]/b^2, k -> -b - y2;
    //x2 -> x1 - (a*Sqrt[y1 - y2]^2*b - y1 - y2)/b, b -> x1 -
(a*Sqrt[y1 - y2]^2*b - y1 - y2)/b, k -> b - y2;
    //x2 -> x1 - (a*Sqrt[y1 - y2]^2*b - y1 - y2)/b, b -> x1 -
(a*Sqrt[y1 - y2]^2*b - y1 - y2)/b, k -> b - y2;
    */

    //
    if (Lines.slope(line)==0.0) { //horizontal
        double y2 = line.getP1().getY(),
            valX = (a*Math.sqrt((y1 - y2)*( 2*b - y1 + y2)))/b,
            valY = b + y2;
        if (java.lang.Double.isNaN(valX)) {
            valX = Math.sqrt(-(a*a*b*b*(y1 - y2)*( 2*b + y1 - y2)))/(b*b);
            valY = -b + y2;
        }
    }
}

```



```

        tanpt1.setLocation( x1 - valX, line.getP1().getY());
        center1.setLocation(x1 - valX, valY);
        tanpt2.setLocation( x1 + valX, line.getP1().getY() );
        center2.setLocation(x1 + valX, valY);

        normalPt.setLocation(p.getX(),line.getP1().getY());
    }
    else if (Lines.slope(line)==Lines.INFINITY) { //vertical
/*
        {y2 -> -(Sqrt[-a^2*b^2*(x1 - x2)*(2*a - x1 - x2)]/a^2) - y1, k -> -
        (Sqrt[-a^2*b^2*(x1 - x2)*(2*a - x1 - x2)]/a^2) - y1, h -> -a - x2},
        {y2 -> Sqrt[-a^2*b^2*(x1 - x2)*(2*a - x1 - x2)]/a^2 + y1, k ->
        Sqrt[-a^2*b^2*(x1 - x2)*(2*a - x1 - x2)]/a^2 - y1, h -> -a - x2},
        {y2 -> -(b*Sqrt[(x1 - x2)*(2*a - x1 - x2)]/a) - y1, k -> -
        (b*Sqrt[(x1 - x2)*(2*a - x1 - x2)]/a) - y1, h -> a - x2},
        {y2 -> (b*Sqrt[(x1 - x2)*(2*a - x1 - x2)]/a - y1, k -> (b*Sqrt[(x1 -
        x2)*(2*a - x1 - x2)]/a - y1, h -> a - x2};
*/
        double x2 = line.getP1().getX(),
            valX = a + x2,
            valY = (b*Math.sqrt((x1 - x2)*( 2*a - x1 + x2)))/a;
        if (java.lang.Double.isNaN(valY)) {
            valX = -a + x2;
            valY = Math.sqrt(-(a*a*b*b*(x1 - x2)*( 2*a + x1 - x2)))/(a*a);
        }
        tanpt1.setLocation(line.getP1().getX(), y1 - valY );
        center1.setLocation(valX, y1 - valY );
        tanpt2.setLocation(line.getP1().getX(), y1 + valY );
        center2.setLocation(valX, y1 + valY);

        normalPt.setLocation(line.getP1().getX(), p.getY());
    }

    Line2D lineDirection = new Line2D.Double(p, normalPt);
    if ( lineDirection.relativeCCW(center1) == ((originalA < 0) ^
(ratio < 0) ? -1 : 1) ) {
        center = center2;
        tanpt.setLocation( tanpt2 );
    }
    else if ( lineDirection.relativeCCW(center2) == ((originalA < 0) ^
(ratio < 0) ? -1 : 1) )
    {
        center = center1;
        tanpt.setLocation( tanpt1 );
    }
    else {
        System.out.println( "setEllipseToTangent Error: a=" + a + ",
ratio="+ratio);
        return false;
    }

    boolean inBounds = Lines.pointInBounds(line,tanpt);

    if ( ! inBounds) return false;

    radiusX = a;

```

```

radiusY = b;

setFrame(center.getX()-radiusX,center.getY()-radiusY, 2 * radiusX,2
* radiusY);
return true;
}

public boolean setEllipse(Point2D p1, Point2D p2, Point2D p3, double
ratio) {
    double x1 = p1.getX(), y1 = p1.getY(),
           x2 = p2.getX(), y2 = p2.getY(),
           x3 = p3.getX(), y3 = p3.getY(),
           h, k, a, b,
           x1_=x1*x1, x2_=x2*x2, x3_=x3*x3,
           y1_=y1*y1, y2_=y2*y2, y3_=y3*y3,
           ratio_ = ratio*ratio,
           y12_=(y1-y2)*(y1-y2), y13_ = (y1-y3)*(y1-y3), y23_ = (y2 -
y3)*(y2-y3),
           x12_=(x1-x2)*(x1-x2), x13_ = (x1-x3)*(x1-x3), x23_ = (x2 -
x3)*(x2-x3);

    h = (x3_*(y1 - y2) + (x1_ + ratio_*(y1 - y2)*(y1 - y3))*(y2 - y3) -
x2_*(-y1 + y3))/(2*(x3_*(y1 - y2) + x1_*(y2 - y3) + x2_*(-y1 + y3)));
    k = (-(x2_*x3_) + x1_*(-x2 + x3) + ratio_*x3_*(y1 - y2)*(y1 + y2) +
x1_*(x2_ - x3_ + ratio_*(y2 - y3)*(y2 + y3)) + x2_*(x3_ + ratio_*(-y1_ +
y3_)))/(2*ratio_*(x3_*(y1 - y2) + x1_*(y2 - y3) + x2_*(-y1 + y3)));
    a = -Math.sqrt((x12_ + ratio_*y12_)*(x13_ + ratio_*y13_)*(x23_ +
ratio_*y23_))/(2*ratio_*Math.sqrt(Math.pow(x2*y1 - x3*y1 - x1*y2 + x3*y2
+ x1*y3 - x2*y3, 2)));
    b = -Math.sqrt((x12_ + ratio_*y12_)*(x13_ + ratio_*y13_)*(x23_ +
ratio_*y23_))/(2*ratio_*Math.sqrt(Math.pow(x2*y1 - x3*y1 - x1*y2 +
x3*y2 + x1*y3 - x2*y3, 2)));

    radiusX =Math.abs(a);
    radiusY =Math.abs(b);

    center.setLocation(h,k);
    setFrame(center.getX()-radiusX,center.getY()-radiusY, 2 * radiusX,2
* radiusY);
    return true;
}

public int intersects(Line2D line, Point2D[] p) {
    double a = radiusX,
           b = radiusY,
           h = center.getX(),
           k = center.getY();

    if (line.getP1().getX() == line.getP2().getX()) { //vertical
        double x = line.getP2().getX(),
               temp1 = a*a*k,
               sqr = a*a - h*h + 2*h*x - x*x;
        if (sqr < 0) return 0;
        double temp2 = a*b*Math.sqrt(sqr);

```

```

        p[0] = new Point2D.Double(line.getP1().getX(), (temp1-
temp2)/(a*a) );
        p[1] = new Point2D.Double(line.getP1().getX(),
(temp1+temp2)/(a*a) );
        // y -> (a^2*k - a*b*Sqrt[a^2 - b^2 - 2*b*x - x^2])/a^2,
        // -y -> (a^2*k + a*b*Sqrt[a^2 - b^2 - 2*b*x - x^2])/a^2;
    }
    else if (line.getP1().getY() == line.getP2().getY() ) {
//horizontal
        double y = line.getP2().getY(),
            temp1 = b*b*h,
            sqr = b*b - k*k + 2*k*y - y*y;
        if (sqr < 0) return 0;
        double temp2 = a*b*Math.sqrt(sqr);
        p[0] = new Point2D.Double((temp1-temp2)/(b*b),
line.getP1().getY() );
        p[1] = new Point2D.Double((temp1+temp2)/(b*b),
line.getP1().getY() );
        // -x -> (b^2*h - a*b*Sqrt[b^2 - k^2 - 2*k*y - y^2])/b^2,
        // -x -> (b^2*h + a*b*Sqrt[b^2 - k^2 - 2*k*y - y^2])/b^2;
        // y is constant    [x -> -(a*Sqrt[1 - y^2/b^2]), [x -> a*Sqrt[1
- y^2/b^2]];
    }
    else { // UNTESTED general solution
        double slope = Lines.slope(line.getP1(),line.getP2()),
            intercept = Lines.intercept(line.getP1(),line.getP2()),
            a2 = a*a, b2 = b*b, i2=intercept * intercept,
            s2=slope*slope,h2=h*h,k2=k*k,
            sqr = b2 - i2 + 2*intercept*k - k2 - 2*h*intercept*slope +
2*h*k*slope + a2*s2 - h2*s2;
        if (sqr < 0) return 0;
        double ytemp1 = intercept + (b2*h*slope)/(b2 + a2*s2) -
(a2*intercept*s2)/(b2 + a2*s2) + (a2*k*s2)/(b2 + a2*s2),
            ytemp2 = (a*b*slope*Math.sqrt(sqr))/(b2 + a2*s2),
            xtemp1 = b2*h - a2*intercept*slope + a2*k*slope,
            xtemp2 = a*b*Math.sqrt(sqr);
        p[0] = new Point2D.Double( (xtemp1-xtemp2)/(b2 + a2*s2), ytemp1-
ytemp2);
        p[1] = new Point2D.Double( (xtemp1+xtemp2)/(b2 + a2*s2),
ytemp1+ytemp2);
    }
    //
    y -> intercept + (b^2*h*slope)/(b^2 - a^2*slope^2) -
(a^2*intercept*slope^2)/(b^2 - a^2*slope^2) - (a^2*k*slope^2)/(b^2 -
a^2*slope^2) - (a*b*slope*Sqrt[b^2 - intercept^2 - 2*intercept*k - k^2
- 2*h*intercept*slope - 2*h*k*slope - a^2*slope^2 - b^2*slope^2])/b^2
- a^2*slope^2,
    x -> (b^2*h - a^2*intercept*slope - a^2*k*slope - a*b*Sqrt[b^2 -
intercept^2 - 2*intercept*k - k^2 - 2*h*intercept*slope - 2*h*k*slope -
a^2*slope^2 - b^2*slope^2])/b^2 - a^2*slope^2,
    y -> intercept + (b^2*h*slope)/(b^2 - a^2*slope^2) -
(a^2*intercept*slope^2)/(b^2 - a^2*slope^2) - (a^2*k*slope^2)/(b^2 -
a^2*slope^2) - (a*b*slope*Sqrt[b^2 - intercept^2 - 2*intercept*k - k^2
- 2*h*intercept*slope - 2*h*k*slope - a^2*slope^2 - b^2*slope^2])/b^2
- a^2*slope^2,
    x -> (b^2*h - a^2*intercept*slope - a^2*k*slope - a*b*Sqrt[b^2 -
intercept^2 - 2*intercept*k - k^2 - 2*h*intercept*slope - 2*h*k*slope -
a^2*slope^2 - b^2*slope^2])/b^2 - a^2*slope^2;

```

```

    }
    int solutions = 2;
    boolean p1In = Lines.pointInBounds(line,p[ 0]),
           p2In = Lines.pointInBounds(line,p[ 1]);
    if (p1In && p2In) {
        if ( p[0].equals(p[1]) ) {
            p[1] = null;
            solutions = 1;
        }
        else solutions = 2;
    }
    else if (p1In) {
        solutions = 1;
        p[1] = null;
    }
    else if (p2In) {
        solutions = 1;
        p[0] = p[1];
        p[1] = null;
    }
    else {
        p[0] = p[1] = null;
        solutions = 0;
    }
    return solutions;
}

public void setEllipse(Point2D p1, Point2D p2, Point2D p3, Point2D
p4) {
    double x1 = p1.getX(), y1 = p1.getY(),
           x2 = p2.getX(), y2 = p2.getY(),
           x3 = p3.getX(), y3 = p3.getY(),
           x4 = p4.getX(), y4 = p4.getY(),
           x1_ = x1 * x1, y1_ = y1 * y1,
           x2_ = x2 * x2, y2_ = y2 * y2,
           x3_ = x3 * x3, y3_ = y3 * y3,
           x4_ = x4 * x4, y4_ = y4 * y4,

           f = x1 * x1 - x2 * x2,
           g = y1 * y1 - y2 * y2,
           j = x1 - x2,
           l = y1 - y2,
           m = x1 * x1 - x3 * x3,
           n = y1 * y1 - y3 * y3,
           p = x1 - x3,
           q = y1 - y3,
           r = x1 * x1 - x4 * x4,
           s = y1 * y1 - y4 * y4,
           t = x1 - x4,
           u = y1 - y4,

           h_den = 2.0 * ((j * n - p * g) * (q * s - u * n)
- (p * s - t * n) * (l * n - q * g)),
           k_den = 2.0 * ((l * m - q * f) * (p * r - t * m)
- (q * r - u * m) * (j * m - p * f)),

```

```

        e_den = ((l * n - q * g) * (p * s - t * n) - (q
* s - u * n) * (j * n - p * g)),
        h, k, eq8, a_b2, a, b, in_a, in_b, eq9, temp;
        if ( EllipseArc.approximate(h_den, 0.0) ||
EllipseArc.approximate(k_den, 0.0) || EllipseArc.approximate(e_den, 0.0)
) return;
        h = ( (f * n - m * g) * (q * s - u * n) - (m * s
- r * n) * (l * n - q * g)) / h_den;
        k = ( (g * m - n * f) * (p * r - t * m) - (n * r
- s * m) * (j * m - p * f)) / k_den;

        eq8 = ((f * n - m * g) * (p * s - t * n) - (m *
s - r * n) * (j * n - p * g)) / e_den;
        // eq9 = (y1 * y1 - y2 * y2 - 2.0 * x * (y1 - y2) )
/ (2.0 * n * (x1 - x2) - (x1 * x1 - x2 * x2));

        a_b2 = eq8 / (2.0 * k);
        // temp = (eq8 * eq9) / (2.0 * k);
        in_a = h * h + k*k*a_b2 + x1 * x1 + a_b2 * y1 *
y1 - 2.0 * h * x1 - 2 * k * a_b2 * y1;
        in_b = 1.0 / a_b2;
        if ((in_a <= 0.0) || (in_b <= 0.0)) return;
        a = Math.sqrt(in_a);
        b = Math.sqrt(in_b) * a;

        center.setLocation( h, k);
        radiusX = a;
        radiusY = b;
        setFrame(center.getX()-radiusX,center.getY()-radiusY,  2 * radiusX,2
* radiusY);

        // start = stop = 0.0;
    }

/* FROM MATHEMATICA..Solution for the STANDARD CONIC THAT INTERSECTS 4
POINTS:

Solve[{A x1^2 + B y1^2 + C x1 + F y1 == 1,
      A x2^2 + B y2^2 + C x2 + F y2 == 1,
      A x3^2 + B y3^2 + C x3 + F y3 == 1,
      A x4^2 + B y4^2 + C x4 + F y4 == 1}, {A, B, C, F}]
FullSimplify[%]

From In[24]:=
{A -> (x4*(y1 - y2)*(y1 - y3)*(y2 - y3) - x3*(y1 - y2)*(y1 - y4)*(y2 -
y4) - (x2*(y1 - y3)*(y1 - y4) - x1*(y2 - y3)*(y2 - y4))*(y3 - y4))/
(y3*(x4*x2*x4*y1*(y1 - y3) - x1*(x1 - x4)*y2*(y2 - y3) -
x2^2*y1*(y1 - y3)) - x1*(x1 - x2)*x2*y3*y4 - x1*x2*(-x1 - x2)*y4^2) -
x3^2*x4*y1*(y1 - y2)*y2 - y4*(x1*y2*(y2 - y4) - x2*y1*(y1 -
y4)) - x3*(x4^3*y1*y2*(y1 - y2) - y4*(x2^2*y1*(y1 - y4) - x1^2*y2*(-
y2 - y4))),
B -> 1-(x1*(x1 - x4)*x4*(y2 - y3)) - x3*(x4^2*(y1 - y2) - x1^2*(y2 -
y4) - x3^2*(-(x4*y1) - x1*y2 - x4*y2 - x1*y4) -
x2^2*x4*y1 - x1*y3 - x4*y3 - x1*y4 - x3*(-y1 - y4)) - x1*(x4^2*(-
y1 - y3) - x3^2*(y1 - y4) - x1^2*(-y3 - y4))),
y3*(x4*x2*x4*y1*(y1 - y3) - x1*(x1 - x4)*y2*(y2 - y3) -
x2^2*y1*(y1 - y3)) - x1*(x1 - x2)*x2*y3*y4 - x1*x2*(-x1 - x2)*y4^2) -

```

```

x3^2*(x4*y1*(y1 - y2)*y2 - y4*(x1*y2*(y2 - y4) + x2*y1*(-y1 -
y4)) - x3*(x4^2*y1*y2*(-y1 - y2) - y4*(x2^2*y1*(y1 - y4) + x1^2*y2*(-
y2 - y4))) ,
D -> (x4^2*(y1 - y2)*(y1 - y3)*(y2 - y3) - x3^2*(y1 - y2)*(y1 -
y4)*(y2 - y4) - (x2^2*(y1 - y3)*(y1 - y4) - x1^2*(y2 - y3)*(y2 -
y4))*y3 - y4^2) /
(x4*(x3*x4*y1*(y1 - y2)*y2 - x3^2*y1*y2*(-y1 - y2) -
y3*(x2^2*y1*(y1 - y3) - x1*(x1 - x4)*y2*(y2 - y3) - x2*x4*y1*(-y1 +
y3))) -
(x2*x3*(-x2 - x3)*y1^2 - x1*(x1 - x3)*x3*y2^2 - x1*x2*(-x1 +
x2)*y3^2)*y4 - (x3*(x2*(x2 - x3)*y1 - x1*(-x1 - x3)*y2) - x1*(x1 -
x2)*x2*y3)*y4^2) ,
F -> (x1*(x1 - x4)*x4*y2 - y3)*(y3 - y3) - x2^2*(x3*y1^2 - x4*y1^2 -
x1*y3^2 - x4*y3^2 - (x1 - x3)*y4^2) - x3^2*(x4*(y1 - y2)*(y1 - y2) -
x1*(y2 - y4)*(y2 - y4)) -
x2*(x4^2*(y1 - y3)*(y1 - y3) - x1^2*(y3 - y4)*(y3 - y4) - x3^2*(-
y1^2 - y4^2)) - x3*(x4^2*(-y1^2 - y2^2) - x1^2*(-y2^2 - y4^2))) /
(y3*(x4*(x2*x4*y1*(y1 - y3) - x1*(x1 - x4)*y2*(y2 - y3) -
x2^2*y1*(-y1 - y3)) - x1*(x1 - x2)*x2*y3*y4 - x1*x2*(-x1 - x2)*y4^2) -
x3^2*(x4*y1*(y1 - y2)*y2 + y4*(x1*y2*(y2 - y4) + x2*y1*(-y1 -
y4)) - x3*(x4^2*y1*y2*(-y1 - y2) - y4*(x2^2*y1*(y1 - y4) - x1^2*y2*(-
y2 - y4))))))
*/

```

```

public boolean setEllipse(Point2D p1, Point2D p2, Line2D line, double
r) {
    Ellipse ellipse[] = new Ellipse[2], solution;
    Point2D tanPt[] = new Point2D.Double[2];

    getEllipse( p1, p2, line, r, ellipse, tanPt);

    if (ellipse[0] == null) // no tangent ellipse
        return setEllipse(p1, p2, r > 0 ? line.getP2() : line.getP1(), r
);

    if (ellipse[1] != null) { //two ellipses to choose from
        Line2D lineDirection = new Line2D.Double(p1, p2);
        if ( lineDirection.relativeCCW(ellipse[0].center) == ((r < 0) ? -
1 : 1) ) {
            solution = ellipse[1];
        }
        else {
            solution = ellipse[0];
        }
    }
    else solution = ellipse[0];
    radiusX = Math.abs(solution.radiusX);
    radiusY = Math.abs(solution.radiusY);
    setFrame( solution.center.getX() - radiusX,
              solution.center.getY() - radiusY, 2 * radiusX, 2 *
radiusY);
    return true;
}

```

```

public boolean getEllipse(Point2D p1, Point2D p2, Line2D line, double
r,
                        Ellipse ellipse[], Point2D tanPt[] ) {

    ellipse[0] = ellipse[1] = null;
    tanPt[0] = tanPt[1] = null;

    double x1 = p1.getX(), y1 = p1.getY(),
           x2 = p2.getX(), y2 = p2.getY(),
           x3 = line.getX1(),
           y3 = line.getY1(),
           h, k, a=0, b,
           x1_ =x1*x1, x2_ =x2*x2, x3_ =x3*x3,
           y1_ =y1*y1, y2_ =y2*y2, y3_ =y3*y3,
           r_ = r*r,
           y12_ =(y1-y2)*(y1-y2), y13_ = (y1-y3)*(y1-y3), y23_ = (y2 -
y3)*(y2-y3),
           x12_ =(x1-x2)*(x1-x2), x13_ = (x1-x3)*(x1-x3), x23_ = (x2 -
x3)*(x2-x3);

    Ellipse ellipse1=null, ellipse2=null;
    Point2D tanPt1 = null, tanPt2 = null;

    if (Lines.slope(line)==Lines.INFINITY) { // vertical
        if (x12_ == 0) {
            b = Math.sqrt((x13_ + (r_*y12_)/4)*(x23_ + (r_*y12_)/4)*(x12_ +
r_*y12_))/(r_*Math.sqrt((x1 + x2 - 2*x3)*(x1 + x2 - 2*x3)*y12_));
            a = Math.sqrt((x13_ + (r_*y12_)/4)*(x23_ + (r_*y12_)/4)*(x12_ +
r_*y12_))/(r_*Math.sqrt((x1 + x2 - 2*x3)*(x1 + x2 - 2*x3)*y12_));
            h = (2*(x1 + x2 - 2*x3)*(x1 + x3) + r_*y12_)/(4*(x1 + x2 -
2*x3));
            k = (y1 + y2)/2;
            y3 = (y1 + y2)/2;
            ellipse1 = new Ellipse(new Point2D.Double(h,k),a,r);
            tanPt1 = new Point2D.Double(x3,y3);
        }
        else {
            a = ((x1 + x2 - 2*x3)*(x12_ + r_*y12_) + 2*Math.sqrt(r_*(x1 -
x3)*(x2 - x3)*(x12_ + r_*y12_))*(y1 - y2))/(2*x12_);
            b = ((x1 + x2 - 2*x3)*(x12_ + r_*y12_) + 2*Math.sqrt(r_*(x1 -
x3)*(x2 - x3)*(x12_ + r_*y12_))*(y1 - y2))/(2*r*x12_);
            h = ((x1*x1_ - x1_*x2 + (x2*x2_) + r_*x2*y12_ - x1*(x2 + r*(y1 -
y2))*(x2 + r*(-y1 + y2)) + 2*(y1 - y2)*(Math.sqrt(r_*(x1 - x3)*(x2 -
x3)*(x12_ + r_*y12_) + r_*x3*(-y1 + y2)))/(2*x12_);
            k = (-Math.sqrt(r_*(x1 - x3)*(x2 - x3)*(x12_ + r_*y12_)) + r_*(-
(x2*y1) - x3*y1 + x1*y2 - x3*y2))/(r_*(x1 - x2));
            y3 = (-Math.sqrt(r_*(x1 - x3)*(x2 - x3)*(x12_ + r_*y12_)) + r_*(-
(x2*y1) - x3*y1 + x1*y2 - x3*y2))/(r_*(x1 - x2));
            ellipse1 = new Ellipse(new Point2D.Double(h,k),a,r);
            tanPt1 = new Point2D.Double(x3,y3);

            a = ((x1 + x2 - 2*x3)*(x12_ + r_*y12_) - 2*Math.sqrt(r_*(x1 -
x3)*(x2 - x3)*(x12_ + r_*y12_))*(y1 - y2))/(2*x12_);
            b = ((x1 + x2 - 2*x3)*(x12_ + r_*y12_) - 2*Math.sqrt(r_*(x1 -
x3)*(x2 - x3)*(x12_ + r_*y12_))*(y1 - y2))/(2*r*x12_);

```

```

        h = ((x1*x1_) - x1_*x2 + (x2*x2_) - 2*(Math.sqrt(r_*(x1 - x3)*(x2
- x3)*(x12_ + r_*y12_)) + r_*x3*(y1 - y2))*(y1 - y2) + r_*x2*y12_ -
x1*(x2 + r*(y1 - y2))*(x2 + r*(-y1 + y2)))/(2*x12_);
        k = (Math.sqrt(r_*(x1 - x3)*(x2 - x3)*(x12_ + r_*y12_)) + r_*(-
(x2*y1) + x3*y1 + x1*y2 - x3*y2))/(r_*(x1 - x2));
        y3 = (Math.sqrt(r_*(x1 - x3)*(x2 - x3)*(x12_ + r_*y12_)) + r_*(-
(x2*y1) + x3*y1 + x1*y2 - x3*y2))/(r_*(x1 - x2));
        ellipse2 = new Ellipse(new Point2D.Double(h,k),a,r);
        tanPt2 = new Point2D.Double(x3,y3);
    }
}
else if (Lines.slope(line) == 0){ //line is horizontal
    if (y12_ == 0) {
        a = Math.sqrt((x12_ + r_*y12_)*(x12_/4 + r_*y13_)*(x12_/4 +
r_*y23_))/(r*Math.sqrt(x12_*(y1 + y2 - 2*y3)*(y1 + y2 - 2*y3)));
        b = Math.sqrt((x12_ + r_*y12_)*(x12_/4 + r_*y13_)*(x12_/4 +
r_*y23_))/(r*Math.sqrt(x12_*(y1 + y2 - 2*y3)*(y1 + y2 - 2*y3)));
        h = (x1 + x2)/2;
        k = (x12_ + 2*r_*(y1 + y2 - 2*y3)*(y2 + y3))/(4*r_*(y1 + y2 -
2*y3));
        x3 = (x1 + x2)/2;
        ellipse1 = new Ellipse(new Point2D.Double(h,k),a,r);
        tanPt1 = new Point2D.Double(x3,y3);
    }
    else {
        a = ((x12_ + r_*y12_)*(y1 + y2 - 2*y3) + 2*(x1 -
x2)*Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 - y3)*(y2 -
y3)))/(2*r_*y12_);
        b = ((x12_ + r_*y12_)*(y1 + y2 - 2*y3) + 2*(x1 -
x2)*Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 - y3)*(y2 -
y3)))/(2*r_*y12_);
        h = (x2*(y1 - y3) - Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 -
y3)*(y2 - y3)) + x1*(-y2 + y3))/(y1 - y2);
        k = (x1_*y1 - 2*x1*x2*y1 + x2_*y1 + r_*y1*y1_ + x1_*y2 -
2*x1*x2*y2 + x2_*y2 - r_*y1_*y2 - r_*y1*y2_ + r_*y2*y2_ -
2*(x1 - x2)*Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 -
y3)*(y2 - y3)) - 2*x1_*y3 + 4*x1*x2*y3 - 2*x2_*y3)/(2*r_*y12_);
        x3 = (x2*(y1 - y3) - Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 -
y3)*(y2 - y3)) + x1*(-y2 + y3))/(y1 - y2);
        ellipse1 = new Ellipse(new Point2D.Double(h,k),a,r);
        tanPt1 = new Point2D.Double(x3,y3);

        a = ((x12_ + r_*y12_)*(y1 + y2 - 2*y3) - 2*(x1 -
x2)*Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 - y3)*(y2 -
y3)))/(2*r_*y12_);
        b = ((x12_ + r_*y12_)*(y1 + y2 - 2*y3) - 2*(x1 -
x2)*Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 - y3)*(y2 -
y3)))/(2*r_*y12_);
        h = (x2*(y1 - y3) + Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 -
y3)*(y2 - y3)) + x1*(-y2 + y3))/(y1 - y2);
        k = (x1_*y1 - 2*x1*x2*y1 + x2_*y1 + r_*y1*y1_ + x1_*y2 -
2*x1*x2*y2 + x2_*y2 - r_*y1_*y2 - r_*y1*y2_ + r_*y2*y2_ -
2*(x1 - x2)*Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 - y3)*(y2
- y3)) - 2*x1_*y3 + 4*x1*x2*y3 - 2*x2_*y3)/(2*r_*y12_);
        x3 = (x2*(y1 - y3) + Math.sqrt(x12_ + r_*y12_)*Math.sqrt((y1 -
y3)*(y2 - y3)) + x1*(-y2 + y3))/(y1 - y2);
    }
}

```



```

        ellipse2 = new Ellipse(new Point2D.Double(h,k),a,r);
        tanPt2 = new Point2D.Double(x3,y3);
    }
}
else {
    System.err.println("Tangent Line in method Ellipse is neither
vertical nor horizontal");
    System.exit(1);
}

boolean inBounds1 = Lines.pointInBounds(line,tanPt1),
        inBounds2 = Lines.pointInBounds(line,tanPt2);

if (inBounds1 && ! inBounds2) {
    ellipse[0] = ellipse1;
    tanPt[0] = tanPt1;
}
else if (inBounds2 && ! inBounds1) {
    ellipse[0] = ellipse2;
    tanPt[0] = tanPt2;
}
else if (inBounds1 && inBounds2) { // must find correct
    ellipse[0] = ellipse1;
    ellipse[1] = ellipse2;
    tanPt[0] = tanPt1;
    tanPt[1] = tanPt2;
}
else return false;

return true;
}
}

```

Eclipse.java

EllipseArc.java

```

package domain;

import java.awt.geom.*;

public class EllipseArc extends Arc2D.Double {
    public static int NEITHER_PT = 0,
                     START_PT = 1,
                     END_PT = 2,
                     BOTH_PT = 3;

    Ellipse e;
    public EllipseArc(Point2D p1, Point2D p2, Point2D p3, double ratio) {
        e = new Ellipse(p1, p2, p3, ratio);
        setFrame(e.getFrame());
        setAngles( unSquash(p1), unSquash(p3) );
        setArcType(OPEN);
        setAngleExtent(- 360+extent);
        setAngleStart(start);
    }
    public EllipseArc(Point2D p1, Point2D p2, double a, double ratio) {
        e = new Ellipse(p1, p2, a, ratio);
        setFrame(e.getFrame());
        setAngles(unSquash(p1),unSquash(p2) );
        setArcType(OPEN);
        setAngleExtent(- 360+extent);
        setAngleStart(start);
    }

    public Point2D unSquash(Point2D p) {
        return new Point2D.Double( (p.getX() - getCenterX()) / getWidth() +
                                   getCenterX() ,
                                   (p.getY() - getCenterY()) / getHeight()
                                   +getCenterY() );
    }

    public int intersects(Line2D line, Point2D[] p, int
intersectEndPoints) {
        int solutions = e.intersects(line,p);
        double angleP1, angleP2;
        boolean p1In = false, p2In = false;

        if ( solutions > 0) {
            angleP1 = Lines.vectorAngle(e.center, p[ 0]);
            p1In = containsAngle( angleP1 , Lines.TOLERANCE) ;
            if (p1In &&
                ( ( ( intersectEndPoints == NEITHER_PT || intersectEndPoints
== END_PT) && approximate(p[ 0],getStartPoint() ) ) ||
                ( ( intersectEndPoints == NEITHER_PT || intersectEndPoints
== START_PT) && approximate(p[ 0],getEndPoint() ) ) ) )
                p1In = false;
            if (solutions == 2) {
                angleP2 = Lines.vectorAngle(e.center, p[ 1]);
                p2In = containsAngle( angleP2, Lines.TOLERANCE );
                if (p2In &&

```

```

        ( ( ( intersectEndPoints == NEITHER_PT ||
intersectEndPoints == END_PT) && approximate(p[ 1],getStartPoint() ) )
||
        ( ( intersectEndPoints == NEITHER_PT ||
intersectEndPoints == START_PT) && approximate(p[ 1],getEndPoint() ) ) )
)

        p2In = false;
    }
    if (p1In && p2In) return 2;
    else if (p1In) {
        p[1] = null;
        return 1;
    }
    else if (p2In) {
        p[0] = p[1];
        p[1] = null;
        return 1;
    }
    p[0] = p[1] = null;
    return 0;
}

public boolean containsAngle( double angle, double tolerance) {
    return containsAngle(angle + tolerance) ||
        containsAngle(angle - tolerance);
}

}

public static boolean approximate(double a, double b, double
tolerance) {
    return Math.abs(a-b) < tolerance;
}
public static boolean approximate(double a, double b) {
    return approximate(a,b,Lines.TOLERANCE);
}
public static boolean approximate(Point2D a, Point2D b, double
tolerance) {
    return Math.abs(a.getX()-b.getX()) < tolerance &&
Math.abs(a.getY()-b.getY()) < tolerance;
}
public static boolean approximate(Point2D a, Point2D b) {
    return approximate(a,b,Lines.TOLERANCE);
}
}
}

```

EclipseArc.java

Image.java

```

package domain;

import java.awt.*;
import java.awt.geom.*;
import java.util.*;
import javax.swing.*;
import java.text.*;
import java.lang.*;

public class Image {
    FreemanChainList freemanList = new FreemanChainList();
    Vector passingAreaList = new Vector();
    Spectrum spectrum = new Spectrum();

    DecimalFormat decimalFormat = new DecimalFormat("0.00");

    public Image() { }

    public FreemanChainList getFreemanChainList() { return freemanList; }

    public FreemanChain getFreemanChain() { return
freemanList.getCurrent(); }

    public void setPassingArea(boolean infiniteLine, double alpha, double
beta)
    {
        setPassingArea( infiniteLine, alpha, beta, Integer.MAX_VALUE, 1);
    }

    public void clearPassingArea()
    {
        passingAreaList.clear();
    }

    public void setBoundary(boolean infiniteLine, double alpha, double
beta, int grayLevelMax)
    {
        PassingArea passingArea = PassingArea.boundary(freemanList, alpha,
beta, grayLevelMax);
        passingAreaList.clear();

        passingAreaList.add(passingArea);
    }

    public void setPassingArea(boolean infiniteLine, double alpha, double
beta, int iter, int grayLevelMax)
    {
        PassingArea passingArea = new PassingArea(freemanList,
infiniteLine, alpha, beta, iter, grayLevelMax);
        passingAreaList.clear();

        passingAreaList.add(passingArea);
    }
}

```

```

    public void setPassingAreaRange2D(double alphaStart, double
betaStart,
                                double alphaStop, double betaStop,
                                int increments)
    {
        spectrum.clear();
        setPassingArea(false, 0, 0); // pixel boundary
        double betaCurrent,
            betaIncrement = Math.max(0.01, (betaStop-betaStart) /
increments );
        String display = "";
        for (betaCurrent = betaStart; betaCurrent < betaStop; betaCurrent
+= betaIncrement)
            display += setPassingAreaRange(alphaStart, betaCurrent,
alphaStop, betaCurrent,
increments, false);
        JTextArea text = new JTextArea(display, spectrum.size() + 2, 80);
        text.setEditable(false);
        JOptionPane.showMessageDialog(null, new
JScrollPane(text, JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED, JScrollPane.H
ORIZONTAL_SCROLLBAR_AS_NEEDED),
            "Range: " + cartesian(alphaStart, betaStart) + "-" +
cartesian(alphaStop, betaStop) + " Critical Sets: " + spectrum.size(),
JOptionPane.INFORMATION_MESSAGE);
    }

    public void setPassingAreaRangeID(double alphaStart, double
betaStart,
                                double alphaStop, double betaStop,
                                int increments)
    {
        spectrum.clear();
        setPassingArea(false, 0, 0); // pixel boundary
        setPassingAreaRange(alphaStart, betaStart,
alphaStop, betaStop,
increments, true);
    }

    public static String cartesian(double n1, double n2) {
        DecimalFormat decimalFormat = new DecimalFormat("0.00");
        return "(" + decimalFormat.format(n1) + ", " +
decimalFormat.format(n2) + ")";
    }

    public String setPassingAreaRange(double alphaStart, double
betaStart,
                                double alphaStop, double betaStop,
                                int increments, boolean show)
    {
        double alphaCurrent, betaCurrent,
            alphaIncrement = (alphaStop-alphaStart) / increments,
            betaIncrement = (betaStop-betaStart) / increments;

        alphaCurrent = alphaStart;

```

```

        betaCurrent = betaStart;
        // find first
        for (alphaCurrent = alphaStart; alphaCurrent < alphaStop;
            alphaCurrent += alphaIncrement)
        {
            PassingArea passingArea = new PassingArea(freemanList, true,
                alphaCurrent, betaCurrent);
            if (passingArea.size() > 2) {
                alphaStart = alphaCurrent;
                betaStart = betaCurrent;
                break;
            }
            betaCurrent += betaIncrement;
        }
        alphaCurrent = alphaStop;
        betaCurrent = betaStop;
        // find last
        for (alphaCurrent = alphaStop; alphaCurrent > alphaStart;
            alphaCurrent -= alphaIncrement)
        {
            PassingArea passingArea = new PassingArea(freemanList, true,
                alphaCurrent, betaCurrent);
            if (passingArea.size() > 2) {
                alphaStop = alphaCurrent;
                betaStop = betaCurrent;
                break;
            }
            betaCurrent -= betaIncrement;
        }

        alphaIncrement = (alphaStop-alphaStart) / increments;
        betaIncrement = (betaStop-betaStart) / increments;

//      alphaCurrent = alphaStart;
//      betaCurrent = betaStart;

        double alphaPrev, betaPrev;
        boolean first = true,
            newPassingArea = false;
        PassingArea passingArea = null,
            lastPassingArea = null;
        String display = "";

//
        int totalIterations = show ? increments : increments * increments /
        4,
            iterations = 0,
            iterationBreak = Math.max(totalIterations / 100, 1);
//
        for (alphaCurrent = alphaPrev = alphaStart, betaCurrent = betaPrev
            = betaStart;
            alphaCurrent < alphaStop;
            alphaCurrent += alphaIncrement, betaCurrent += betaIncrement)
        {
            int lastSize = alphaCurrent == alphaStart ? 0 :
                passingArea.size();

```

```

        passingArea = new PassingArea(freemanList, true, alphaCurrent,
betaCurrent);
        if (passingArea.size() > 2) {
            // if ( iterations-- & iterationBreak == 0)
            passingAreaList.add(passingArea);
            // if (! spectrum.contains(passingArea.getCriticalSet())) {
            if (first || !
passingArea.getCriticalSet().equals(lastPassingArea.getCriticalSet() )
) {
                if (! first & newPassingArea)
                    display += spectrum.size() + ": " +
                        cartesian(alphaPrev, betaPrev) +
                            "-" +
                                cartesian(alphaCurrent,betaCurrent) +
                                    "\t{" + lastPassingArea.getCriticalSet()
+ "}" + "\n";
                else
                    first = false;
                    newPassingArea = !
spectrum.contains(passingArea.getCriticalSet());
                    spectrum.add(passingArea.getCriticalSet());
                    alphaPrev = alphaCurrent;
                    betaPrev = betaCurrent;
                }
                lastPassingArea = passingArea;
            }
        }
        if ( ! first & newPassingArea )
            display += spectrum.size() + ": " +
                cartesian(alphaPrev, betaPrev) + "-" +
                    cartesian(alphaCurrent,betaCurrent) +
                        "\t{" + passingArea.getCriticalSet() + "}" + "\n";
        if (show) {
            JTextArea text = new JTextArea(display, spectrum.size() + 2, 60);
            text.setEditable( false);
            JOptionPane.showMessageDialog( null, new
JScrollPane(text,JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,JScrollPane.H
ORIZONTAL_SCROLLBAR_AS_NEEDED),
                "Range: " + cartesian(alphaStart, betaStart) + "-" +
                    cartesian(alphaStop, betaStop) + " Critical Sets: " + spectrum.size(),
JOptionPane.INFORMATION_MESSAGE);
        }
        return display;
    }

    public int setPassingAreaRange( double radius, double ratio, double
increment)
    {
        DecimalFormat decimalFormat = new DecimalFormat("0.00");

        double alphaCurrent, betaCurrent,
            alphaIncrement, betaIncrement;

        PassingArea passingArea;
        spectrum = new Spectrum();

        alphaCurrent = radius;

```

```

    betaCurrent = radius / ratio;

    alphaIncrement = increment;
    betaIncrement = increment / ratio;

    do {
        passingArea = new PassingArea(freemanList, true, alphaCurrent,
betaCurrent);
        if (passingArea.size() > 2)
            spectrum.add(passingArea.getCriticalSet());
        alphaCurrent += alphaIncrement;
        betaCurrent += betaIncrement;
    }
    while (passingArea.size() > 2);

    alphaCurrent = radius;
    betaCurrent = radius / ratio;

    alphaIncrement = -increment;
    betaIncrement = -increment / ratio;

    do {
        passingArea = new PassingArea(freemanList, true, alphaCurrent,
betaCurrent);
        if (passingArea.size() > 2)
            spectrum.add(passingArea.getCriticalSet());
        alphaCurrent += alphaIncrement;
        betaCurrent += betaIncrement;
    }
    while (passingArea.size() > 2);

    // JOptionPane.showMessageDialog(null,"size: " + spectrum.size()
    -"in"-spectrum;
    // JOptionPane.showMessageDialog(null, "Critical Sets: " + counter
    -"in" + spectrum, "spectrum", JOptionPane.INFORMATION_MESSAGE);
    return spectrum.size();
}

public int setPassingAreaRangeRecursive( double radius, double ratio,
double increment)
{
    double alphaCurrent, betaCurrent,
        alphaIncrement, betaIncrement,
        radiusLo, radiusHi;
    int counter = 0;

    PassingArea passingArea;
    spectrum = new Spectrum();

    // find radiusLo and radiusHi
    double incrementBig = 0.05;
    //radiusLo

    alphaCurrent = radius;
    betaCurrent = radius / ratio;

    alphaIncrement = -incrementBig;
    betaIncrement = -incrementBig / ratio;

```



```

do {
    alphaCurrent += (alphaIncrement *= 2);
    betaCurrent += (betaIncrement *= 2);
    passingArea = new PassingArea(freemanList, true, alphaCurrent,
betaCurrent );
}
while (passingArea.size() > 2);
radiusLo = alphaCurrent;

//radiusHi
alphaCurrent = radius;
betaCurrent = radius / ratio;

alphaIncrement = incrementBig;
betaIncrement = incrementBig / ratio;
do {
    alphaCurrent += (alphaIncrement *= 2);
    betaCurrent += (betaIncrement *= 2);
    passingArea = new PassingArea(freemanList, true, alphaCurrent,
betaCurrent );
}
while (passingArea.size() > 2);
radiusHi = alphaCurrent;
// JOptionPane.showMessageDialog(null,"radius" - radius - "\n radiusLo:
" - radiusLo - "\nradiusHi: " - radiusHi);

spectrum.add(new PassingArea(freemanList, true, radiusLo, radiusLo
/ ratio ).getCriticalSet() );
spectrum.add(new PassingArea(freemanList, true, radius, radius /
ratio ).getCriticalSet() );
setCriticalSetsInterval(radiusLo, radius, ratio, increment);

spectrum.add(new PassingArea(freemanList, true, radiusHi, radiusHi
/ ratio ).getCriticalSet() );
setCriticalSetsInterval(radius, radiusHi, ratio, increment);
return spectrum.size();
}

public void setCriticalSetsInterval(double lo, double hi, double
ratio, double increment) {
    double range, mid;
    PassingArea passingArea;
    String loCS, midCS, hiCS;
    boolean loEq, hiEq;
    while ( (range = (hi - lo)) > increment ) {
        mid = (lo + hi) / 2;
        loCS = (passingArea = new PassingArea(freemanList, true, lo, lo
/ ratio )).getCriticalSet();
        midCS = (passingArea = new PassingArea(freemanList, true, mid,
mid / ratio )).getCriticalSet();
        hiCS = (passingArea = new PassingArea(freemanList, true, hi, hi
/ ratio )).getCriticalSet();

        if ( (loEq = loCS.equals(midCS)) | (hiEq = hiCS.equals(midCS)) )
        {
            if (loEq) lo = mid;
            if (hiEq) hi = mid;

```

```

    }
    else if ( spectrum.add(midCS) ) {
        setCriticalSetsInterval(lo, mid, ratio, increment);
        lo = mid;
        setCriticalSetsInterval(mid, hi, ratio, increment);
        hi = mid;
    }
    else System.out.println("setCriticalSetsInterval method : this
possibility is theoretically impossible");
}

}

public void setRanges(double r) {
    this.getPassingArea(0).setRangeAll(r);
}

PassingArea getPassingArea( int index)
{
    return (PassingArea) passingAreaList.elementAt(index);
}
}

class Pixel extends Point {
    double greylevel; // as a percentage of pixel 0 < greylevel <= 1
    int code = -1; //code from previous pixel
    Pixel() {
        greylevel = 1.0;
    }

    Pixel(Pixel p) {
        super(p.x,p.y);
        greylevel = p.greylevel;
    }

    Pixel(int x, int y, double g) {
        super(x,y);
        greylevel = g;
    }

    public boolean neighbor4(Pixel p) {
        return Math.abs(p.x - x) == 1 ^ Math.abs(p.y - y) == 1;
    }

    public boolean neighbor8(Pixel p) {
        return Math.abs(p.x - x) == 1 || Math.abs(p.y - y) == 1;
    }

    public Rectangle2D rectangle() {
        Rectangle2D rect= new Rectangle2D.Double(x, y, 1, 1);
        return rect;
    }

    public Pixel moveDirection (int d) {
        int x = 0, y = 0;

```

```

switch (d) {
    case 0:
        x = 1;
        y = 0;
        break;
    case 1:
        x = 1;
        y = -1;
        break;
    case 2:
        x = 0;
        y = -1;
        break;
    case 3:
        x = -1;
        y = -1;
        break;
    case 4:
        x = -1;
        y = 0;
        break;
    case 5:
        x = -1;
        y = 1;
        break;
    case 6:
        x = 0;
        y = 1;
        break;
    case 7:
        x = 1;
        y = 1;
        break;
}
return new Pixel(this.x + x, this.y + y, -1);
}

void setGreyLevel(int g) { greylevel = g; }

int getGreyLevel() { return greylevel; }

boolean setFreemanCode(Pixel prev, int connected) { // 4 connected
    code = -1;
    if ( ((Point) this).distance(prev) == 1)
    {
        if (this.getX() > prev.getX()) code = 0;
        if (this.getX() < prev.getX()) code = 4;
        if (this.getY() > prev.getY()) code = 6;
        if (this.getY() < prev.getY()) code = 2;
    }
    else if ( (connected == FreemanChain.EIGHT_CONNECTED) && (
((Point) this).distance(prev) < 2) ) // 8 connected
    {
        if (this.getX() > prev.getX())
            if (this.getY() < prev.getY()) code = 1;
        else code = 2;
    }
}

```

```

        else if (this.getY() < prev.getY()) code = 3;
        else code = 5;
    }
    if (code == -1) return false;
    else return true;
} // setFreemanCode
}

class FreemanChainList extends Vector {
    FreemanChain freemanCurrent;

    FreemanChainList() {
        super.add( freemanCurrent = new FreemanChain() );
    }

    FreemanChain getCurrent() {
        return freemanCurrent;
    }

    void reset() {
        clear();
        super.add( freemanCurrent = new FreemanChain() );
    }

    // boolean add(FreemanChain f);
    // super.add(f);
    // freemanCurrent = f;
    // return true;

    //
    //

    boolean add(Pixel p)
    {
        if ( freemanCurrent.add( p ) ) return true;
        // not first pixel or 4-connected
        if (freemanCurrent.size() < 2) return false; // each chain must
        // have at least 2 pixels. Later this will be amended to allow even
        // single pixels

        super.add( freemanCurrent = new FreemanChain() );
        freemanCurrent.add( p );
        return true;
    }
}

class FreemanChain extends Vector {
    final static int FOUR_CONNECTED = 1;
    final static int EIGHT_CONNECTED = 2;
    FreemanChain() {
        // super();
    }

    void reset() {
        clear();
    }
}

```

```

    boolean add(Pixel p)
    {
        if (this.size() == 0)
        {
            super.add(p);
            return true;
        }
        else if ( p.setFreemanCode( (Pixel) lastElement(), FOUR_CONNECTED
    ) )
        {
            super.add(p);
            return true;
        }
        return false;
    }
}

interface PassingAreaElement {
    DecimalFormat pointFormat = new DecimalFormat("0.0");
    Range range = new Range();

    abstract boolean setShape(Shape s);
    abstract void setLabel(String s);
    abstract void setRange(PassingAreaElement p1, PassingAreaElement p2);

    abstract Shape getShape();
    abstract String getLabel();
    // abstract String getRange();
    abstract double getX();
    abstract double getY();
}

class PassingAreaLine extends Line2D.Double
                        implements PassingAreaElement {
    String label;
    Shape s;

    // Range range = new Range();

    PassingAreaLine() {
    }

    PassingAreaLine(Point2D p1, Point2D p2) {
        setLine(p1,p2);
    }

    public boolean setShape(Shape s)
    {
        this.s = s; // check to make sure valid shape
        return true;
    }
    public void setLabel(String s) {label = s;}

    public void setRange(PassingAreaElement p1, PassingAreaElement p2){

```

```

    }

    public Shape getShape() {return s;}
    public String getLabel() {return label;}
    public double getX() {
        return (getP1().getX() + getP2().getX()) / 2;
    }
    public double getY() {
        return (getP1().getY() + getP2().getY()) / 2;
    }
    public String toString() {
        return pointFormat.format(getP1().getX()) + "," +
        pointFormat.format(getP1().getY()) + "----" +
        pointFormat.format(getP2().getX()) + "," +
        pointFormat.format(getP2().getY());
    }
}

class PassingAreaPoint extends Point2D.Double
    implements PassingAreaElement {
    String label;
    Shape s;
    // Range range = new Range();

    PassingAreaPoint() {
    }

    PassingAreaPoint(double x, double y) {
        setLocation(x,y);
    }

    public boolean setShape(Shape s)
    {
        this.s = s; // check to make sure valid shape
        return true;
    }
    public void setLabel(String s) {label = s;}

    public void setRange(PassingAreaElement p1, PassingAreaElement p2){

    }

    public Shape getShape() {return s;}
    public String getLabel() {return label;}

    PassingAreaPoint diagonalPoint( PassingAreaPoint p1, PassingAreaPoint
p2)
    { // returns object's diagonal wrt p1,p2. Diagram
        //

```

```

                                The diag. of pt 1 wrt 2,3 is 4.
                                2
1,4    3.                                2                                4,5
                                */
    PassingAreaPoint diagpt = null;

    if ( (x == p1.x) && (y == p2.y) ) diagpt = new
PassingAreaPoint(p2.x, p1.y);
    else if ( (x == p2.x) && (y == p1.y) ) diagpt = new
PassingAreaPoint(p1.x, p2.y);
    return diagpt;
}

    public String toString() {
        return pointFormat.format(getX()) + "," +
pointFormat.format(getY());
    }

    public PassingAreaPoint midPoint(PassingAreaPoint p) {
        return new PassingAreaPoint( (x + p.x) / 2.0, (y + p.y) / 2.0);
    }
}

class PassingArea extends LinkedList {
    final static double INFINITY = Double.MAX_VALUE;
    final static double TINY = Double.MIN_VALUE;

    final static int MINIMUM = 1;
    final static int MAXIMUM = 2;
    final static int CONCAVE = 1;
    final static int CONVEX = 2;
    final static int LINEAR = 3;

    double alphaRadius, betaRadius,
        ratio; // used for critical ranges

    double areaTotal; // gets set in setArea()
    Line2D startLine, endLine; //initialized in setAlphaBetaHull to
compute areaTotal after null minimization

    private int leftTail = -1;
    DrawContext domain = new DrawContext();
    String criticalSet = "";

    PassingArea() {
    }

    PassingArea(FreemanChainList freemanList, boolean infiniteHull,
double a, double b) {
        this(freemanList, infiniteHull, a, b, Integer.MAX_VALUE, 1);
    }

    PassingArea(FreemanChainList freemanList, boolean infiniteHull,
double a, double b, int iter, int grayLevelMax) {
        initialize(freemanList, infiniteHull, a, b, iter, grayLevelMax);
    }
}

```

```

    }

    void initialize(FreemanChainList freemanList, boolean infiniteHull,
double a, double b, int iter, int grayLevelMax) {
    alphaRadius = a;
    betaRadius = b;
    setOutline(freemanList, grayLevelMax);
    removeJoints();
    if (a != Lines.INFINITY ) setElements(); // when finding convex
hull only vertices are relevant
    setLabels();
    // remove comment to display only candidate elements and not all
boundary points
    // if (a == 0) cornerPoints[];

    if (a > 0)
    {
        // compute hull
        setAlphaBetaHull( infiniteHull, a, b, iter );
    // read these methods
        if (iter == Integer.MAX_VALUE)
            setDomain();
            setCriticalSet();
        }

        else setLines(infiniteHull); // boundary lines w/o infinite lines

        areaTotal = a > 0 ? getArea(infiniteHull) : freemanList.size() -
1;
    }

    public static PassingArea boundary(FreemanChainList freemanList,
double a, double b, int grayLevelMax) {
        PassingArea pa = new PassingArea();
        pa.alphaRadius = a;
        pa.betaRadius = b;
        pa.setOutline(freemanList, grayLevelMax);
        pa.removeJoints();
        if (a != Lines.INFINITY ) pa.setElements(); // when finding
convex hull only vertices are relevant
        pa.setLabels();

        pa.alphaRadius = pa.betaRadius = Lines.INFINITY; // causes
setLines to draw lines not arcs
        pa.setLines( false);

        pa.alphaRadius = a; // reset radii to previous val's
        pa.betaRadius = b;

        pa.areaTotal = a > 0 ? pa.getArea( false) : freemanList.size() -
1;
        return pa;
    }

    void setOutline(FreemanChainList freemanList, int grayLevelMax)
    {
        for (int i = 0; i < freemanList.size(); i++)

```



```

        setOutline( (FreemanChain) freemanList.get(i), grayLevelMax);
    }

/**
 * sets this to the inner and outer pixel boundary of freeman adding
 * the constraint
 * imposed by the grey level values of each pixel in the image where
 * grayLevelMax
 * is the number of levels of gray in the image.
 * When grayLevelMax equals 1 the image is the binary image and this is
 * set to the inner and outer pixel boundary of freeman
 * 2 is a binary image with rounding
 */
    void setOutline(FreemanChain freeman) {
        setOutline(freeman, 1); //binary image
    }

/**
 * sets this to the inner and outer pixel boundary of freeman.
 */
    void setOutline(FreemanChain freeman, int grayLevelMax)
    {
        // find where circle of radius a,b intersects the pixel box
        minimally and maximally

        if (freeman.size() == 1) {
            PassingAreaPoint left = (PassingAreaPoint) get(leftTail),
                right = (PassingAreaPoint) get(size() - 1);

            // find where circle of radius a,b intersects the pixel box
            minimally and maximally

        }

        int index = 1;           // chaincode index
        PassingAreaPoint
            lpnw = new PassingAreaPoint(),
            rpnw = new PassingAreaPoint(),
            lplast = new PassingAreaPoint(),
            rplast = new PassingAreaPoint(),
            pextra;
        Pixel
            pixelnow = new Pixel(),
            pixel;
        PassingAreaPoint cp1, cp2;

        lpnw.setLocation(-1, -1);    // initialization, used in Stage
        // specific value not important;
        rpnw.setLocation(-1, -1);

        pixelnow.setLocation( (Point) freeman.firstElement() );
        // Pixel (0,0) is 1st pixel right & above origin and is bounded
        // by the real pts (0,0) (0,1) (1,0) (1,1)
        while (index < freeman.size()) { // Stage 1: compute new right
            and left points
            lplast.setLocation( (Point2D) lpnw );
            rplast.setLocation( (Point2D) rpnw );

```

```

switch (( pixel = ( Pixel )freeman.elementAt(index++)).code)
{
    case 0:
        lpnw.setLocation( pixelnow.x + 1, pixelnow.y );
        rpnw.setLocation( pixelnow.x + 1, pixelnow.y + 1);
        pixelnow.setLocation( pixelnow.x + 1, pixelnow.y );
        break;
    case 2:
        lpnw.setLocation( pixelnow.x , pixelnow.y );
        rpnw.setLocation( pixelnow.x + 1, pixelnow.y );
        pixelnow.setLocation( pixelnow.x, pixelnow.y - 1);
        break;
    case 4:
        lpnw.setLocation( pixelnow.x, pixelnow.y + 1 );
        rpnw.setLocation( pixelnow.x, pixelnow.y );
        pixelnow.setLocation( pixelnow.x - 1, pixelnow.y );
        break;
    case 6:
        lpnw.setLocation( pixelnow.x + 1, pixelnow.y + 1);
        rpnw.setLocation( pixelnow.x, pixelnow.y + 1);
        pixelnow.setLocation( pixelnow.x, pixelnow.y + 1);
        break;
}
double gray = (( Pixel )freeman.elementAt(index-
2)).greylevel;

    // Stage 1 : adding point to list
    if ( ( lpnw.equals(lplast) ) && (! rpnw.equals( rplast ) ) )
{ // don't add lp but add extra rp and rp
    pxtra = lpnw.diagonalPoint(rpnw,rplast); // add pxtra
    cp1 = new PassingAreaPoint( pxtra.x, pxtra.y ); // add rp
    cp2 = new PassingAreaPoint( rpnw.x, rpnw.y );
    // cp1->SetBetweenPixels(0);
    // cp2->SetBetweenPixels(1);

    if ( grayLevelMax > 1) {
        addLeft( grayPoint( gray, pxtra, lpnw, CONCAVE,
MINIMUM ,grayLevelMax) );
        addLeft( (PassingAreaPoint) lpnw.clone() );
    }

    addRight(cp1);

    if ( grayLevelMax > 1) {
        addRight( grayPoint( gray, pxtra, lpnw, CONCAVE,
MAXIMUM ,grayLevelMax) );
        addRight(cp1);
    }
    addRight(cp2);
}
    else if ((! lpnw.equals( lplast ) ) && (rpnw.equals(rplast)
) ) { // don't add rp but add extra lp and lp
    pxtra = rpnw.diagonalPoint(lpnw,lplast); // add pxtra
    cp1 = new PassingAreaPoint( pxtra.x, pxtra.y ); // add rp
    cp2 = new PassingAreaPoint( lpnw.x, lpnw.y );
    //cp1->SetBetweenPixels(0);
    //cp2->SetBetweenPixels(1);
}

```

```

        if ( grayLevelMax > 1 ) {
            addRight( grayPoint( gray, rpnw, pxtra, CONVEX,
MAXIMUM, grayLevelMax) );
            addRight( (PassingAreaPoint) rpnw.clone() );
        }

        addLeft(cp1);

        if ( grayLevelMax > 1 ) {
            addLeft( grayPoint( gray, rpnw, pxtra, CONVEX, MINIMUM,
grayLevelMax) );
            addLeft(cp1);
        }

        addLeft(cp2);
    }
    else if ((index == 1) ||
        ((! lpnw.equals(lplast) ) && (!
rpnw.equals(rplast) )) ) { // add both lp and rp
        cp1 = new PassingAreaPoint(lpnw.x, lpnw.y );
        cp2 = new PassingAreaPoint(rpnw.x, rpnw.y );
        // cp1->SetBetweenPixels(1);
        // cp2->SetBetweenPixels(1);

        if (lplast.x >= 0 && grayLevelMax > 1) {
            addLeft( lplast.midPoint( lpnw ) );
            addLeft( grayPoint( gray, rplast.midPoint( rpnw ),
lplast.midPoint( lpnw ), LINEAR, MINIMUM ,grayLevelMax) );
            addLeft( lplast.midPoint( lpnw ) );
        }

        addLeft(cp1);

        if (lplast.x >= 0 && grayLevelMax > 1) {
            addRight( rplast.midPoint( rpnw ) );
            addRight( grayPoint( gray, rplast.midPoint( rpnw ),
lplast.midPoint( lpnw ), LINEAR, MAXIMUM,grayLevelMax) );
            addRight( rplast.midPoint( rpnw ) );
        }

        addRight(cp2);
    }
} // while

// setting startline and endline (used to compute end of
passing area perimeter

PassingAreaPoint start1 = (PassingAreaPoint) get( 0),
start2 = (PassingAreaPoint) get(leftTail+ 1),
end1 = (PassingAreaPoint) get(leftTail),
end2 = (PassingAreaPoint) get(size() - 1);

startLine = new Line2D.Double(start1, start2);
endLine = new Line2D.Double(end1, end2);

```

```

//      JOptionPane.showMessageDialog(null,"" + startLine.getP1() +
"\n"- startLine.getP2() + "\n"- endLine.getP1() + "\n"- endLine.getP2()
+ "\n" );
}

    public PassingAreaPoint grayPoint( double grayLevel,
PassingAreaPoint rightPt,
PassingAreaPoint leftPt,
int pixelConfiguration, int range, int
grayLevelMax)
    {
        if (alphaRadius == Lines.INFINITY)
            return grayPointDigitalLine(grayLevel, rightPt, leftPt,
pixelConfiguration, range, grayLevelMax);
        else
            return grayPointDigitalCircle(grayLevel, rightPt, leftPt,
pixelConfiguration, range, grayLevelMax);
    }

    /**
     * returns the gray point for a pixel along a digital line segment.
     */
    public PassingAreaPoint grayPointDigitalLine( double grayLevel,
PassingAreaPoint rightPt,
PassingAreaPoint leftPt,
int pixelConfiguration, int range, int
grayLevelMax)
    {
        PassingAreaPoint gray = null;

        double epsilon = grayLevelMax == 1 ? 1 : 1.0 / ((grayLevelMax - 1)
* 2),
            area;

        switch (pixelConfiguration) {
            case LINEAR:
                area = range == MINIMUM ? grayLevel - epsilon : grayLevel +
epsilon;
                area = rangeMinMax(area, 0, 1);

                gray = pointOnLine( new Line2D.Double(leftPt, rightPt), area);
                break;
            case CONCAVE:
                if ( range == MINIMUM ) {
                    area = rangeMinMax(grayLevel - epsilon, 0, 0.5);

                    double min = Math.sqrt(2 * Math.pow((2 * area) / (1 + 2 *
area), 2 ));
                    min = rangeMinMax(min, 0, Math.sqrt(2)/2 );
                    gray = pointOnLine( new Line2D.Double(leftPt, rightPt),
min);
                }
                else { // MAXIMUM:
                    area = rangeMinMax(grayLevel + epsilon, 0, 0.5 );
                    gray = pointOnLine( new Line2D.Double(leftPt, rightPt),
Math.sqrt(area) );
                }
            }
    }

```

```

        break;
    case CONVEX:
        if ( range == MINIMUM ) {
            area = rangeMinMax(grayLevel - epsilon, 0.5, 1.0);
            gray = pointOnLine( new Line2D.Double(leftPt, rightPt),
Math.sqrt(2) - Math.sqrt( 1 - area ) );
        }
        else { // MAXIMUM:
            area = rangeMinMax(grayLevel + epsilon, 0.5, 1.0);

            double max = Math.sqrt(2) - Math.sqrt(2 * Math.pow((2 *
(1-area)) / (1 + 2 * (1-area) ), 2 ));
            max = rangeMinMax(max, Math.sqrt( 2)/2, Math.sqrt(2));

            gray = pointOnLine( new Line2D.Double(leftPt, rightPt),
max );
        }
        break;
    }
    return gray;
}

public PassingAreaPoint grayPointDigitalCircle( double grayLevel,
PassingAreaPoint rightPt,
PassingAreaPoint leftPt,
int pixelConfiguration, int range, int
grayLevelMax)
{
    PassingAreaPoint gray = null;

    double r = alphaRadius,
        epsilon = grayLevelMax == 1 ? 1 : 1.0 / ((grayLevelMax - 1)
* 2),
        area;

    switch (pixelConfiguration) {
        case LINEAR:

            if ( range == MINIMUM ) {
                area = rangeMinMax(grayLevel - epsilon, 0, 1);
                double min = area + r - Math.sqrt(4 * r * r - 1) / 4.0 -
                    r * r * Math.asin( 1.0 / ( 2 * r ) );
                gray = pointOnLine( new Line2D.Double(leftPt, rightPt), min);
                break;
            }
            else { // MAXIMUM
                area = rangeMinMax(grayLevel + epsilon, 0, 1);
                double max;

                if (area > 0.5) {
                    double k = 2 * area - 1;
                    max = (1 / (2 * (2 + (k - 2) * k)))
                        * (2 + (k - 1) * k * k - Math.sqrt(( -2 - (k - 2)
) * k) * (2 + (k - 2) * k - 4 * r * r))
                        + Math.sqrt(( 2 + (k - 2) * k) * ((k - 1)
* (k - 1) * (2 + (k - 2) * k) + 4 * r * r)));
                }
            }
        }
    }
}

```

```

        else {
            double
                k = area * 2,
                k2 = k * k,
                k3 = k2 * k,
                k4 = k2 * k2,
                r2 = r * r;
            max = ( k + k3 - Math.sqrt( - ( 1 + k2 ) * ( 1 + k2 - 4 *
r2) )
                                + Math.sqrt( ( 1 + k2) * (k2 + k4 + 4 * r2)
) )
                                / ( 2 * ( 1 + k2 ) );

/* alternate method..not as good
            double max = area - 0.5 * (1 - Math.sqrt(-1 - 2 * r * r) +
Math.sqrt(1 - 2 * r * r));
*/
        }
        max = rangeMinMax(max, 0, 1);
        gray = pointOnLine( new Line2D.Double(leftPt, rightPt), max);
        break;
    }
    case CONCAVE:
        if ( range == MINIMUM ) {
            area = rangeMinMax(grayLevel - epsilon, 0, 1);

            double min = Math.sqrt(2 * Math.pow((2 * area) / (1 + 2 *
area),2 ));
            min = rangeMinMax(min, 0, Math.sqrt(2)/2 );
            gray = pointOnLine( new Line2D.Double(leftPt, rightPt),
min);
        }
        else { // MAXIMUM:
            area = rangeMinMax(grayLevel + epsilon, 0, 1);
            double k = Math.sqrt(2 * area),
                max = (k + Math.sqrt( 2) * r - Math.sqrt( - ( k * k )
+ 2 * r * r) ) / 2.0;
            max = Math.sqrt( 2 * max * max);
            max = rangeMinMax(max, 0, Math.sqrt(2));
            gray = pointOnLine( new Line2D.Double(leftPt, rightPt), max
);
        }
        break;

    case CONVEX:
        if ( range == MINIMUM ) {
            area = rangeMinMax(grayLevel - epsilon, 0.5, 1.0);
            gray = pointOnLine( new Line2D.Double(leftPt, rightPt),
Math.sqrt(2) - Math.sqrt( 1 - area) );
        }
        else { // MAXIMUM:
            area = rangeMinMax(grayLevel + epsilon, 0, 1);

            double k = 2 * area - 1,
                k2 = k * k,

```

```

        k3 = k2 * k,
        r2 = r * r,
        km2 = k - 2,
        max = ( 1 / ( 4 * ( 2 + km2 * k ) ) )
            * ( 4 - 2 * k + k3 + km2 * Math.sqrt( ( - 2 - km2
* k ) * ( 2 + km2 * k - 4 * r2 ) )
            + Math.sqrt( 2 )
            * Math.sqrt( ( 2 + km2 * k ) * ( 2 * km2 * km2 *
r2 + k2 * Math.sqrt( ( - 2 - km2 * k ) * ( 2 + km2 * k - 4 * r2 ) ) ) ) );

        max = Math.sqrt( 2 * max * max );
        max = rangeMinMax(max, Math.sqrt( 2 ) / 2, Math.sqrt( 2 ) );

        gray = pointOnLine( new Line2D.Double(leftPt, rightPt),
max );

//            gray = (PassingAreaPoint) rightPt.clone();
    }
    break;
}
return gray;
}

public double rangeMinMax(double x, double min, double max) {
    return Math.min(Math.max(x,min),max);
}

public PassingAreaPoint pointOnLine( Line2D line, double delta) {
    double hyp = line.getP1().distance( line.getP2()),
        xDist = line.getX2() - line.getX1(),
        yDist = line.getY2() - line.getY1();

    delta = Math.max( Math.min( hyp, delta), 0 );

    double pX = delta * xDist / hyp,
        pY = delta * yDist / hyp;

    return new PassingAreaPoint(line.getX1() + pX, line.getY1() + pY);
}

void setLabels() {
    int index = 0;
    while ( index < size() )
    {
        ( (PassingAreaElement) get(index) ).setLabel(
            isLeftHull(index) ? "p" + index : "q" + (index - 1 - leftTail)
        );
        index++;
    }
}

void setLines(boolean infiniteLines) {
//deprecated    setEdges();

    int index;

```

```

    PassingAreaElement o1, o2;
    Point2D p1, p2;

    for (index = 1; index <= size(); index++) {
        o1 = (PassingAreaElement) get(index - 1);
        o2 = (PassingAreaElement) get(index % size());

        p1 = o1 instanceof PassingAreaPoint ? (PassingAreaPoint) o1 :
            ((PassingAreaLine) o1).getP2();
        p2 = o2 instanceof PassingAreaPoint ? (PassingAreaPoint) o2 :
            ((PassingAreaLine) o2).getP1();
        if ( ( index - 1 == leftTail ) || (index == size()) )
        {
            if (infiniteLines)
                if ( (alphaRadius == INFINITY) || (betaRadius == INFINITY) ||
                    (alphaRadius == 0) || (betaRadius == 0) )
                    o1.setShape( Lines.infiniteLine( p1, p2 ) );
            // else if (alphaRadius == betaRadius) p1.setShape( new
            CircleArc( p2, p1, -alphaRadius );
            else o1.setShape( new EllipseArc( p2, p1, -alphaRadius,
                alphaRadius/betaRadius ) );
        }
        else
        {
            if ( (alphaRadius == INFINITY) || (betaRadius == INFINITY) ||
                (alphaRadius == 0) || (betaRadius == 0) )
                o1.setShape( new Line2D.Double( p1, p2 ) );
            // else if (alphaRadius == betaRadius) p1.setShape( new
            CircleArc( p2, p1, alphaRadius );
            else o1.setShape( new EllipseArc( p2, p1, alphaRadius,
                alphaRadius/betaRadius ) );
        }
    }
}
/**
 * Computes and sets the area for this passing area
 */
public double getArea(boolean infiniteLines) {
    /**
     * Add all points
     * to upperList and lowerList. Each line is set as two points.
     */

    if (size() < 3) return 0.0;

    Vector upperList = new Vector(),
        lowerList = new Vector();

    PassingAreaElement p;
    Vector list;
    for (int index = 0; index <= size() - 1; index++) {
        p = (PassingAreaElement) get(index);
        list = index <= leftTail ? upperList : lowerList;
        if (p instanceof PassingAreaPoint)
            list.add(
                new PointAndArc( ((PassingAreaPoint) p).getX(),
                    ((PassingAreaPoint) p).getY(), alphaRadius ) );
    }
}

```



```

        else { // line
            list.add( new PointAndArc( ((PassingAreaLine) p).getP1(),
PointAndArc.LINE ));
            list.add( new PointAndArc( ((PassingAreaLine) p).getP2(),
alphaRadius ));
        }
    }

    // first must determine points that intersect start and endline.
    if (infiniteLines) { // if not infinite then points are strat and
end point are set correctly
        Point2D upperStart, upperEnd, lowerStart, lowerEnd;

        Point2D upperFirstElement = (Point2D) upperList.get( 0),
            upperLastElement = (Point2D) upperList.lastElement(),
            lowerFirstElement = (Point2D) lowerList.get( 0),
            lowerLastElement = (Point2D) lowerList.lastElement();

        if ( alphaRadius == INFINITY || betaRadius == INFINITY ||
alphaRadius == 0 || betaRadius == 0 ) {
            Line2D line = Lines.line( lowerLastElement, upperFirstElement);
            upperStart = Lines.linesIntersect( line, startLine );
            lowerEnd = Lines.linesIntersect( line, endLine );
        }
        else {
            Point2D intersectionPoints[] = new Point2D[2];
            EllipseArc e = new EllipseArc( upperFirstElement,
lowerLastElement, -alphaRadius, alphaRadius/betaRadius);
            e.intersects( startLine, intersectionPoints,
EllipseArc.BOTH_PT);
            upperStart = intersectionPoints[ 0];
            e.intersects( endLine, intersectionPoints, EllipseArc.BOTH_PT);
            lowerEnd = intersectionPoints[ 0];
        }

        if ( alphaRadius == INFINITY || betaRadius == INFINITY ||
alphaRadius == 0 || betaRadius == 0 ) {
            Line2D line = Lines.line( upperLastElement, lowerFirstElement);
            lowerStart = Lines.linesIntersect( line, startLine );
            upperEnd = Lines.linesIntersect( line, endLine );
        }
        else {
            Point2D intersectionPoints[] = new Point2D[2];
            EllipseArc e = new EllipseArc( lowerFirstElement,
upperLastElement, -alphaRadius, alphaRadius/betaRadius);
            e.intersects( startLine, intersectionPoints,
EllipseArc.BOTH_PT);
            lowerStart = intersectionPoints[ 0];
            e.intersects( endLine, intersectionPoints, EllipseArc.BOTH_PT);
            upperEnd = intersectionPoints[ 0];
        }

        if (upperStart != null && upperEnd != null && lowerStart != null
&& lowerEnd != null) {
            // add startline and endline intersection points, if necessary
            if (upperStart.distance(upperFirstElement) > Lines.TOLERANCE)
                upperList.add( 0, new PointAndArc(upperStart, alphaRadius));

```

```

        if (upperEnd.distance(upperLastElement) > Lines.TOLERANCE)
            upperList.add( new PointAndArc(upperEnd, alphaRadius));
        if (lowerStart.distance(lowerFirstElement) > Lines.TOLERANCE)
            lowerList.add( 0, new PointAndArc(lowerStart, alphaRadius));
        if (lowerEnd.distance(lowerLastElement) > Lines.TOLERANCE)
            lowerList.add( new PointAndArc(lowerEnd, alphaRadius));
    }
}

PointAndArc list1[] = new PointAndArc[upperList.size()],
list2[] = new PointAndArc[lowerList.size()];

upperList.toArray(list1);
lowerList.toArray(list2);
areaTotal = Area.passingArea( list1, list2);

// JOptionPane.showMessageDialog(null,"Area = " + areaTotal +
"size" + list1.length + "size " + list2.length);
return areaTotal;
}

void setDomain() {
    if (size() < 3) return; // null domain

    int index;
    double pointSize = 0.01;
    if (this.alphaRadius != INFINITY)
    {
        EllipseArc p1, p2;
        for (index = size() - 1; index >= 0 ; index--)
        {
            boolean point2null = false;
            p1 = (EllipseArc)((PassingAreaElement) get(index)).getShape();
            if (p1 == null) continue;
            p2 = (EllipseArc)((PassingAreaElement) get( (index + 1) %
size() )).getShape();
            if (p2 == null)
            {
                p2 = (EllipseArc)((PassingAreaElement) get( (index + 2) %
size() )).getShape();
                point2null = true;
            }
            if (p1.e.center.equals(p2.e.center)) continue;
            domain.setPenSize( 4);
            domain.setDrawColor(Color.blue);
            domain.add( new Rectangle2D.Double( p1.e.center.getX() -
pointSize / 2,
                                                p1.e.center.getY() -
pointSize / 2,
                                                pointSize, pointSize ) );

            domain.setPenSize( 1);
            domain.setDrawColor(Color.red);
            domain.add(
                isLeftHull(index+(point2null ? 2 : 1) )
                ? new EllipseArc( p2.e.center, p1.e.center,
alphaRadius, alphaRadius/betaRadius)

```

```

        : new EllipseArc( p1.e.center, p2.e.center,
alphaRadius, alphaRadius/betaRadius));
    // domain for outer hull have concavity reversed
    }
}
else // line domain in [m,b] space
{
    PassingAreaPoint p1, p2;
    Point2D current = new Point2D.Double(),
        last = new Point2D.Double(),
        first = new Point2D.Double();
    for (index = size() - 1; index >= 0 ; index--)
    {
        p1 = (PassingAreaPoint) get(index);
        p2 = (PassingAreaPoint) get( (index + 1) % size() );

        domain.setPenSize( 4);
        domain.setDrawColor(Color.blue);
        current.setLocation(Lines.slope(p1,p2) - pointSize / 2,
            Lines.intercept(p1,p2) - pointSize / 2);
        domain.add( new Rectangle2D.Double( current.getX(),
current.getY(),
                                pointSize, pointSize ) );

        domain.setPenSize( 1);
        if (index < size() - 1)
        {
            domain.setDrawColor(Color.green);
            domain.add( new Line2D.Double( last.getX(), last.getY(),
current.getX(),
current.getY() ) );
        }
        else first.setLocation( current );
        last.setLocation( current );
    }
    domain.setDrawColor(Color.green);
    domain.add( new Line2D.Double( last.getX(), last.getY(),
first.getX(), first.getY() ) );
}
}

/* old 2-10-00
void setDomain() {
    if (size() < 3) return; // null domain

    int index;
    double pointSize = 0.01;
    if (this.alphaRadius != INFINITY)
    {
        EllipseArc p1, p2;
        for (index = size() - 1; index >= 0 ; index--)
        {
            p1 = (EllipseArc)((PassingAreaPoint) get(index)).getShape();
            p2 = (EllipseArc)((PassingAreaPoint) get( (index + 1) % size()
)).getShape();
            if (p1.e.center.equals(p2.e.center)) continue;
            domain.setPenSize(4);

```

```

        domain.setDrawColor(Color.blue);
        domain.add( new Rectangle2D.Double( p1.e.center.getX() -
pointSize / 2,
                                                    p1.e.center.getY() -
pointSize / 2,
                                                    pointSize, pointSize ); );

        domain.setPenSize(1);
        domain.setDrawColor(Color.green);
        domain.add(
            isLeftHull(index-1) ? new EllipseArc( p2.e.center,
p1.e.center, alphaRadius, alphaRadius/betaRadius) :
            new EllipseArc( p1.e.center,
p2.e.center, alphaRadius, alphaRadius/betaRadius);
        // domain for outer hull have concavity reversed
    }
    else // line domain in [m,c] space
    {
        PassingAreaPoint p1, p2;
        Point2D current = new Point2D.Double();
        last = new Point2D.Double();
        first = new Point2D.Double();
        for (index = size() - 1; index >= 0 ; index--)
        {
            p1 = (PassingAreaPoint) get(index);
            p2 = (PassingAreaPoint) get( (index - 1) % size());

            domain.setPenSize(4);
            domain.setDrawColor(Color.blue);
            current.setLocation( Lines.slope(p1,p2) * pointSize / 2,
                Lines.intercept(p1,p2) * pointSize / 2);
            domain.add( new Rectangle2D.Double( current.getX(),
current.getY(),
                                                    pointSize, pointSize ); );

            domain.setPenSize(1);
            if (index < size() - 1)
            {
                domain.setDrawColor(Color.green);
                domain.add( new Line2D.Double( last.getX(), last.getY(),
current.getX(),
current.getY() ); );
            }
            else first.setLocation( current );
            last.setLocation( current );

            domain.setDrawColor(Color.green);
            domain.add( new Line2D.Double( last.getX(), last.getY(),
                first.getX(), first.getY() ); );
        }
    }

    public void setRangeAll(double ratio)
    {
        cornerPoints();
        this.ratio = ratio;
    }

```

```

    for (int i = 0; i < size(); i++)
        setRange(i, ratio);
}

public void setRange(int index, double ratio)
{
    int i = index,
        i_prev = (index - 1 - size()) % size(),
        i_next = (index + 1) % size();

    PassingAreaElement prev = (PassingAreaElement) get(i_prev),
        p = (PassingAreaElement) get(index),
        next = (PassingAreaElement) get(i_next);

    if (! (prev instanceof PassingAreaPoint &&
        p instanceof PassingAreaPoint &&
        next instanceof PassingAreaPoint)) return;

    double x1, y1, x2, y2, x3, y3;

    x1 = prev instanceof PassingAreaPoint ? prev.getX() :
    prev.getP1().getX();
    y1 = prev instanceof PassingAreaPoint ? prev.getY() :
    prev.getP1().getY();

    x2 = p.getX(), y2 = p.getY(),
    x3 = next.getX(), y3 = next.getY();

    double r = ratio,
        r_ = r * r,
        x1_ = x1 * x1, y1_ = y1 * y1,
        x2_ = x2 * x2, y2_ = y2 * y2,
        x3_ = x3 * x3, y3_ = y3 * y3,

    // h = x3*(y1 - y2) - (x1_ - r_*y1 - y2)*(y1 - y3)/(y2 - y3) -
    x2_*(-y1 - y3)/(2*(x3*(y1 - y2) - x1_*y2 - y3) - x2*(-y1 - y3)),
    // k = (- x2_*x3) - x1_*(-x2 - x3) - r_*x3*(y1 - y2)*(y1 - y2) -
    x1*(x2_ - x3_ - r_*y2 - y3)*y2 - y3)/( - x2*(x3_ - r_*(-y1_ - y3_))/
    (2*r_*(x3*(y1 - y2) - x1_*y2 - y3) - x2*(-y1 - y3)),
    a = Math.sqrt((x1 - x2)*(x1 - x2) - r_*(y1 - y2)*(y1 - y2))*((x1 -
    x3)*(x1 - x3) - r_*(y1 - y3)*(y1 - y3))/(x2 - x3)*(x2 - x3) - r_*y2 -
    y3*(y2 - y3))/(2*r*Math.abs(x3*(-y1 - y2) - x2*(y1 - y3) - x1*(-y2 -
    y3))),
    b = Math.sqrt((x1 - x3)*(x1 - x3) - r_*(y1 - y2)*(y1 - y2))*((x1 -
    x3)*(x1 - x3) - r_*(y1 - y3)*(y1 - y3))/(x2 - x3)*(x2 - x3) - r_*y2 -
    y3*(y2 - y3))/(2*r*Math.abs(x3*(-y1 - y2) - x2*(y1 - y3) - x1*(-y2 -
    y3))),

    Line2D line = new Line2D.Double( (Point2D) prev, (Point2D) next );
    boolean neg = line.relativeCCW( (Point2D)p ) == -1 ? isLeftHalf(1)
    ;

    if (isLeftHalf(1) &&
    ;

```

```

    if ( i == 0 || i == leftTail )
    {
        ((PassingAreaPoint) p).range.minimum = 0;
        ((PassingAreaPoint) p).range.maximum = a;
    }
    else
    {
        if (neg) a = -a;
        ((PassingAreaPoint) p).range.minimum = a;
        ((PassingAreaPoint) p).range.maximum = INFINITY;
    }
}
else
{
    if ( i == leftTail - 1 || i == size() - 1 )
    {
        ((PassingAreaPoint) p).range.minimum = a;
        ((PassingAreaPoint) p).range.maximum = INFINITY;
    }
    else
    {
        if (neg) a = -a;
        ((PassingAreaPoint) p).range.minimum = 0;
        ((PassingAreaPoint) p).range.maximum = a;
    }
}
}

boolean isLeftHull(int i) {
    return (i % size()) <= leftTail;
}

void addRight(PassingAreaElement point)
{
    add( point );    // add to end of Vector
}

void addLeft(PassingAreaElement point) {
    add( ++leftTail, point );
}

public void delete( int index ) {
    if (index <= leftTail) leftTail--;
    remove(index);
}

public void addAtIndex( int index, PassingAreaElement o ) {
    if (index <= leftTail) leftTail++;
    add(index, o);
}

/** leaves points changes minima and maxima of outer hull to edges */
void setElements() {
    if (size() < 2) return;

```

```

    PassingAreaPoint
    p1, p2, p3;
    double minX, maxX, minY, maxY;
    minX = minY = Double.MAX_VALUE;
    maxX = maxY = -Double.MAX_VALUE;
    int index = 0;

    // finding passingAreaLines (minima and maxima)
    while (index < size() - 1) { // a pt to be considered for
        // elimination must be between others
        p2 = (PassingAreaPoint) get(index);

        minX = Math.min(p2.getX(),minX);
        maxX = Math.max(p2.getX(),maxX);
        minY = Math.min(p2.getY(),minY);
        maxY = Math.max(p2.getY(),maxY);
        index++;
    }

    PassingAreaLine edge = null;
    index = leftTail + 1;

    while (index < size() - 1) { // replacing PassingAreaPoints
        //max and min with PassingAreaLines
        p1 = (PassingAreaPoint) get(index);
        p2 = (PassingAreaPoint) get(index + 1);
        if ( ( p1.getX() == minX && p2.getX() == minX) ||
            (p1.getX() == maxX && p2.getX() == maxX) ||
            (p1.getY() == minY && p2.getY() == minY) ||
            (p1.getY() == maxY && p2.getY() == maxY) ) {
            // && ( p1.distance(p2) > 1.0) ;
            {
                edge = new PassingAreaLine( p1, p2 );
                delete(index);
                if (index == size() - 1)
                    delete(index);
                addAtIndex( index, edge );
            }
            else if (edge != null && EllipseArc.approximate( edge.getP2(), p1
        ) ) {
            delete(index);
            edge = null;
        }
        index++;
    } // while

} // setElements

void cornerPoints() {
    if (size() < 2) return;

    Object o1,o2,o3;
    Point2D p1, p2, p3;
    int index = 1;

    while (index < size() - 1) {
        o1 = get(index - 1);

```

```

    o2 = get(index);
    o3 = get(index + 1);

    p1 = o1 instanceof PassingAreaPoint ? (Point2D) o1 : ((Line2D)
o1).getP1();
    p2 = o2 instanceof PassingAreaPoint ? (Point2D) o2 : ((Line2D)
o2).getP1();
    p3 = o3 instanceof PassingAreaPoint ? (Point2D) o3 : ((Line2D)
o3).getP1();
    Line2D l = new Line2D.Double( p1, p3);
    if ( l.ptLineDist(p2) < TINY ||
        ( ( l.relativeCCW( p2 ) != 1 ) ^
          isLeftHull(index) ) )
        delete(index);
    else index++;
    if (index == leftTail) index += 2;
}
// setLabels();
}

void removeJoints() {
    if (size() < 2) return;

    PassingAreaPoint
    p1, p2, p3;
    int index = 1;

    while (index < size() - 1) { // a pt to be considered for
elimination must be between others
        p1 = (PassingAreaPoint) get(index - 1);
        p2 = (PassingAreaPoint) get(index);
        p3 = (PassingAreaPoint) get(index + 1);
        Line2D l = new Line2D.Double( p1, p3);
        double d = l.ptLineDist(p2);
        if ( d < TINY ) {
            delete(index);
        }
        else index++;
        if (index == leftTail) index += 2;
    }
}

// returns true if p2 should be deleted
boolean elementDelete(int index1, int index2, int index3, double a,
double b) {
    PassingAreaElement o1 = (PassingAreaElement) get( index1 ),
    o2 = (PassingAreaElement) get( index2 ),
    o3;

    boolean isLine = a == INFINITY;
    plp2 = (index1 == leftTail) || (index1 == size()- 1);

    if (isLine) { // edges were not inserted...only vertices are in
list
        Point2D p1 =(Point2D) o1,
        p2 =(Point2D) o2,
        p3 =(Point2D) get( index3 );

```



```

Line2D line = new Line2D.Double( p1, p2 );
return ( line.ptLineDist( p3 ) < TINY) ||
        ( ( line.relativeCCW( p3 ) == 1 ) ^
          isLeftHull(index3) ^
          plp2 ) ;
}
boolean isPoint1 = (o1 instanceof PassingAreaPoint),
        isPoint2 = (o2 instanceof PassingAreaPoint),
        isPoint3;

Ellipse ellipse = new Ellipse();
Point2D intPoint[] = new Point2D[2];
intPoint[0] = new Point2D.Double();
intPoint[1] = new Point2D.Double();

ellipse.setEllipse(o2, o1, ! plp2 ? a : -a, a/b, intPoint); // [0]
= intPoint of o2

PassingAreaLine line;

if (! intPoint[1].equals( new Point2D.Double() ) ) { // o1 has
change and is PassingAreaLine
    line = (PassingAreaLine) o1;
//    if (! plp2)
        line.setLine(line.getP1(), intPoint[ 1]);
//    else line.setLine(intPoint[1], line.getP2());
    if ( line.getP1().distance( line.getP2() ) <= Lines.TOLERANCE)
{ // edge is a vertex
    remove( index1 );
    PassingAreaPoint newPt = new PassingAreaPoint(
intPoint[1].getX(), intPoint[1].getY() );
    newPt.setLabel( o1.getLabel() );
    add(index1, newPt );
}
}

if (! intPoint[0].equals( new Point2D.Double() ) ) { // o2 has
change and is PassingAreaLine
    line = (PassingAreaLine) o2;
//    if (! plp2)
        line.setLine(intPoint[ 0], line.getP2());
//    else line.setLine(line.getP1(), intPoint[0]);

    if ( line.getP1().distance( line.getP2() ) <= Lines.TOLERANCE)
{ // edge is a vertex
    remove( index2 );
    PassingAreaPoint newPt = new PassingAreaPoint(
intPoint[0].getX(), intPoint[0].getY() );
    newPt.setLabel( o2.getLabel() );
    add(index2, newPt );
}
}

if (index3 < 0) return false; // no 3rd element called this
method to reset edges only
o3 = (PassingAreaElement) get( index3 );
isPoint3 = (o3 instanceof PassingAreaPoint);

```

```

    if (isPoint3)
        return ( ellipse.contains( (Point2D) o3 ) ||
                Double.isNaN( ellipse.x ) ||
                Double.isNaN( ellipse.y ) ) ^
                isLeftHull(index3);
    else { // determine if entire edge (not) in ellipse. if only
portion is reset edge start and end
        Point2D intersectionPts[] = new Point2D[2];
        boolean intersect = ellipse.intersects( (Line2D) o3 ,
intersectionPts) > 0,
        temp1 = ellipse.contains( ((Line2D) o3 ).getP1() ),
        temp2 = ellipse.contains( ((Line2D) o3 ).getP2() ),
        same = temp1 == temp2,
        inside = temp1 && same,
        outside = !temp1 && same;

        return intersect ||
            ( inside ||
              Double.isNaN( ellipse.x ) ||
              Double.isNaN( ellipse.y ) ) ^
              isLeftHull(index3) );
    }
} // elementDelete

void setAlphaBetaHull(boolean infiniteHull, double a, double b) {
    setAlphaBetaHull(infiniteHull, a, b, Integer.MAX_VALUE);
}

void setAlphaBetaHull(boolean infiniteHull, double a, double b, int
maximumIteration)
{
    int i = 0, index1, index2, index3,
        iteration = 0;

    PassingAreaElement o1 = (PassingAreaElement) get( 0),
        o2 = (PassingAreaElement) get(leftTail+ 1),
        o3 = (PassingAreaElement) get(leftTail),
        o4 = (PassingAreaElement) get(size() - 1);

    Point2D p1 = o1 instanceof PassingAreaPoint ? (Point2D) o1 :
((PassingAreaLine) o1).getP1(),
        p2 = o2 instanceof PassingAreaPoint ? (Point2D) o2 :
((PassingAreaLine) o2).getP1(),
        p3 = o3 instanceof PassingAreaPoint ? (Point2D) o3 :
((PassingAreaLine) o3).getP2(),
        p4 = o4 instanceof PassingAreaPoint ? (Point2D) o4 :
((PassingAreaLine) o4).getP2();

    startLine = new Line2D.Double( p1, p2 ); // used here and in
computeArea
    endLine = new Line2D.Double( p3, p4 );

    boolean cycle2 = false;
    int forward = 0;

    while ( (forward < 2) && (leftTail >= 0) &&

```

```

        (size() - leftTail - 1 >= 1) &&
        ( size() > 2 ) &&
        iteration++ < maximumIteration )
    {
        index1 = (i + size()) % size();
        index2 = (i + 1) % size();
        index3 = (i + 2) % size();

        if ( elementDelete(index1, index2, index3,a,b) )
        {
            delete( index2 );
            i = (i == 0 ? 0 : i - 1);
            if (index2 == 0) i = size() - 2;
            forward = 0;
        }
        else {
            i++;
            cycle2 = cycle2 || i >= size();
            if (cycle2) forward++;
        }
        if (! infiniteHull)
        {
            if (i == size()-2) elementDelete(i, i+1, i-1,a,b);
            if ( (i == leftTail-1) || (i == leftTail) ) i = leftTail+ 1;
            if ( (i == size()-2) || (i == size() - 1) ) i = size();
        }
    }

    if (leftTail < 0 || size() <= 2 || size() - leftTail - 1 < 1) {
        clear();
        return;
    }

    setLines(infiniteHull);
    if (a == INFINITY || ! infiniteHull) return;

    // if (true) return;

    Point2D p[] = new Point2D[2];

    EllipseArc arc1 = (EllipseArc) ((PassingAreaElement)
get(leftTail)).getShape(),
        arc2 = (EllipseArc) ((PassingAreaElement) get(size()-
1)).getShape();

    double h1 = arc1.e.center.getX(),h2 = arc2.e.center.getX(),
        k1 = arc1.e.center.getY(),k2 = arc2.e.center.getY(),
    sqr = (b*b*(h1 - h2)*(h1 - h2) + a*a*(k1 - k2)*(k1 - k2))*
        {-(b*b*(h1 - h2)*(h1 - h2)) +
        a*a*( 2*b + k1 - k2)*(2*b - k1 + k2)};
    if (sqr > 0) {
        double x1 = (h1 + h2 + (a*(-k1 + k2)*Math.sqrt(sqr))/
(b*(b*b*(h1 - h2)*(h1 - h2) + a*a*(k1 - k2)*(k1-k2))))/ 2,
            y1 = (k1 - k2 + (b*(h1 - h2)*Math.sqrt(sqr))/
(a*(b*b*(h1 - h2)*(h1 - h2) + a*a*(k1 - k2)*(k1-k2))))/ 2,
            x2 = (h1 + h2 - (a*(k1 - k2)*Math.sqrt(sqr))/ (b*(b*b*(h1
- h2)*(h1 - h2) + a*a*(k1 - k2)*(k1-k2))))/ 2,

```

```

        y2 = (k1 + k2 + (b*(-h1 + h2)*Math.sqrt(sqr)) /
(a*(b*b*(h1 - h2)*(h1 - h2) + a*a*(k1 - k2)*(k1-k2)))) / 2;
        Point2D sltn1 = new Point2D.Double(x1, y1),
        sltn2 = new Point2D.Double(x2, y2);

        if (EllipseArc.approximate( startLine.getP1(), sltn1) ||
            EllipseArc.approximate( startLine.getP2(), sltn1) ||
            EllipseArc.approximate( endLine.getP1(), sltn1) ||
            EllipseArc.approximate( endLine.getP2(), sltn1) ||
            EllipseArc.approximate( startLine.getP1(), sltn2) ||
            EllipseArc.approximate( startLine.getP2(), sltn2) ||
            EllipseArc.approximate( endLine.getP1(), sltn2) ||
            EllipseArc.approximate( endLine.getP2(), sltn2) )
            clear();
        else return;
    }
    clear();
    return;
}

public String getCriticalSet() { return this.criticalSet;}

void setCriticalSet() {
    for (int i = 0; i < size(); i++)
    {
        criticalSet += ((PassingAreaElement) this.get(i)).getLabel() + (i
< size()-1 ? " ":"");
    }
}

}

class Range {
    double minimum, maximum;
}

class Spectrum extends HashSet {
    Spectrum() {
        super();
    }
    Spectrum(Vector passingAreaList) {
        super();
        if (passingAreaList.size() < 1) return;

        ListIterator iter = passingAreaList.listIterator();
        add( ((PassingArea) iter).getCriticalSet() );
        while (iter.hasNext())
        {
            add( ((PassingArea) iter).getCriticalSet() );
            iter.next();
        }
    }
}

```



Lines.java

```

package domain;
import java.awt.geom.*;

public class Lines {
    /**
     * equals Double.MAX_VALUE
     */
    public final static double INFINITY = Double.MAX_VALUE;
    /**
     * equals 0.001
     */
    public final static double TOLERANCE = 0.001;
    /**
     * returns true if p is within distance TOLERANCE of line
     */
    public static boolean pointInBounds(Line2D line, Point2D p) {
        return ! (line.ptSegDist(p) > TOLERANCE);
    }
    /**
     * Returns the slope. If the slope is infinite, the double constant
     * INFINITY is returned.
     */
    public static double slope(Point2D p1, Point2D p2)
    {
        double x_diff = p1.getX() - p2.getX(),
            y_diff = p1.getY() - p2.getY();

        return x_diff != 0.0 ? y_diff / x_diff : INFINITY;
    }
    public static double slope(Line2D l)
    {
        return slope(l.getP1(), l.getP2() );
    }

    public static double intercept(Point2D p1, Point2D p2)
    {
        double num = -( p2.getX()*p1.getY() - p1.getX()*p2.getY()),
            den = p1.getX() - p2.getX();
        return den == 0.0 ? INFINITY : num/den;
    }
    /**
     * Returns the y-intercept unless the slope is infinite in which case
     * the x-intercept is returned.
     */
    public static double interceptRedone(Point2D p1, Point2D p2)
    {
        double num = -( p2.getX()*p1.getY() - p1.getX()*p2.getY()),
            den = p1.getX() - p2.getX();
        return den == 0.0 ? INFINITY : num/den;
    }

    public static double interceptRedone(Line2D l)
    {
        return intercept(l.getP1(), l.getP2() );
    }
}

```

```

    public static double intercept(Line2D l)
    {
        return interceptRedone(l.getP1(), l.getP2());
    }

    public static double vectorAngle(Point2D startPt, Point2D
endPt) {
        return Math.toDegrees(Math.atan2(endPt.getY() -
startPt.getY(),
                                         endPt.getX() -
startPt.getX() ) );
    }

    public static Line2D line(Point2D p1, Point2D p2)
    {
        double m;
        if ( (m = slope(p1,p2)) == INFINITY) return new
Line2D.Double(p1.getX(), 0, p1.getX(), 100);
        else
        {
            double b = p1.getY() - m * p1.getX(),
            x_diff = p1.getX() - p2.getX(),
            y_diff = p1.getY() - p2.getY(),
            max = Math.max(Math.abs(x_diff),Math.abs(y_diff));
            x_diff = x_diff / max * 100;
            y_diff = y_diff / max * 100;

            return new Line2D.Double(p1.getX() - x_diff, p1.getY() -
y_diff,
                                     p2.getX() + x_diff, p2.getY() +
y_diff);
        }
        // get y intercept is non infinite and intercept at x=5000
    }
    public static GeneralPath infiniteLine(Point2D p1, Point2D p2)
    {
        GeneralPath path = new GeneralPath();
        Line2D line1, line2;
        double m = slope(p1,p2);
        if ( m == INFINITY) {
            line1 = new Line2D.Double(p1.getX(), Math.max(p1.getY(),
p2.getY()), p1.getX(), 100);
            line2 = new Line2D.Double(p1.getX(), Math.min(p1.getY(),
p2.getY()), p1.getX(), 0);
        }
        else
        {
            double b = p1.getY() - m * p1.getX(),
            x_diff = p2.getX() - p1.getX(),
            y_diff = p2.getY() - p1.getY(),
            max = Math.max(Math.abs(x_diff),Math.abs(y_diff));
            x_diff = x_diff / max * 100;
            y_diff = y_diff / max * 100;

            line1 = new Line2D.Double( p1.getX() - x_diff, p1.getY() -
y_diff,

```

```

        p1.getX(), p1.getY() );
        line2 = new Line2D.Double( p2.getX(), p2.getY(),
                                   p2.getX() + x_diff, p2.getY() +
y_diff);
    }
    path.append(line1, false);
    path.append(line2, false);
    return path;
    // get y intercept if non infinite and intercept at x=5000
}

public static Line2D perpendicularLine(Point2D p1, Point2D p2)
{
    Line2D line;
    double m = slope(p1,p2);
    Point2D mid = new Point2D.Double( (p1.getX() + p2.getX()) /
2, ( p1.getY() + p2.getY() ) / 2 );
    if ( m == INFINITY)
        line = new Line2D.Double(0, mid.getY(), 100, mid.getY() );
    else if (m == 0.0)
        line = new Line2D.Double(mid.getX(), 0, mid.getX(), 100);
    else
    {
        m = -1/m;
        Point2D point2 = new Point2D.Double(mid.getX() + 1,
mid.getY() + m);
        line = line(mid, point2);
    }
    return line;
    // get y intercept if non infinite and intercept at x=5000
}

public static boolean approx(double a, double b, double approx)
{
    return Math.abs(a-b) < approx;
}

public static Point2D linesIntersect(Line2D line1, Line2D
line2) {
    double slope1 = Lines.slope(line1),
        slope2 = Lines.slope(line2);
    if ( approx(slope1, slope2, TOLERANCE) && !(slope1 ==
INFINITY ^ slope2 == INFINITY) )
        return null; // parallel lines -> no intersection
    else {
        double intercept1 = Lines.interceptRedone(line1),
            intercept2 = Lines.interceptRedone(line2);
        Point2D point;
        if ( slope1 == INFINITY ) {
            double ipX = line1.getP1().getX();
            point = new Point2D.Double( ipX, slope2 * ipX +
intercept2 );
        }
        else if ( slope2 == INFINITY ) {
            double ipX = line2.getP1().getX();
            point = new Point2D.Double( ipX, slope1 * ipX +
intercept1 );
        }
    }
}

```



```

    }
    else point = new Point2D.Double( (intercept1 - intercept2)
/ (-slope1 + slope2),
                                   (intercept2 * slope1 -
intercept1 * slope2) / (slope1 - slope2) );
    // determine if point is on both line1 and line2
    double d1 = line1.ptSegDist( point ),
           d2 = line2.ptSegDist( point );
    if ( (d1 <= TOLERANCE || Double.isNaN(d1) ) && (d2 <=
TOLERANCE || Double.isNaN(d2) ) )
        return point;
    else return null;
}
}

public static Line2D perpendicularLineGeneral(Point2D p1,
Point2D p2, double ratio)
{
    Line2D line;
    double a=ratio * 10,
           b=10,
           num = (b*b*(p1.getX() - p2.getX() ) ),
           den = (a*a*(-p1.getY() + p2.getY() ) ),
           m;
    Point2D mid = new Point2D.Double( (p1.getX() + p2.getX()) /
2, ( p1.getY() + p2.getY() ) / 2 );
    if ( num == 0.0)
        line = new Line2D.Double(0, mid.getY(), 100, mid.getY() );
    else if (den == 0.0)
        line = new Line2D.Double(mid.getX(), 0, mid.getX(), 100);
    else
    {
        m = num/den;
        Point2D point2 = new Point2D.Double(mid.getX() + 1,
mid.getY() + m);
        line = line(mid, point2);
    }
    return line;
    // get y intercept if non infinite and intercept at x=5000
}
}

```

Lines.java

Bibliography

- [1] C. Barber, D. Dobkin, and H. Huhdanpaa, "The Quickhull algorithm for convex hulls", *ACM Trans. on Mathematical Software*, Vol. 22, pp. 469-483, 1997.
- [2] Barrow, H.G., Tenenbaum, J.M., Retrospective On "Interpreting Line Drawings As Three-Dimensional Surfaces", *AI(59)*, 1993, pp. 71-80.
- [3] M. de Berg and J. Matousek and O. Schwarzkopf, "Piecewise Linear Paths Among Convex Obstacles", *Discrete Comput. Geom.*, Vol. 14, pp. 9-29, 1995.
- [4] Carlos Berenstein, Laveen N. Kanal, David Lavine and Eric Olson, "A Geometric Approach to Subpixel Registration Accuracy", *Computer Vision, Graphics, and Image Processing*, vol. 40, pp. 334-360, 1987.
- [5] B.K. Bhattacharya, "Circular Separability of Planar Point Sets", *Computational Morphology*, G.T. Toussaint, ed., vol. , pp. 25-39, 1988.
- [6] S. Chattopadhyay, P. P. Das and D. Ghosh Dastidar, "Reconstruction of a Digital Circle", *Pattern Recognition*, vol. 27 (12), pp. 1663-1676, 1994.
- [7] Peter Damaschke, "An Optimal Parallel Algorithm for Digital Curve Segmentation", *Theoretical Computer Science*, vol. 178, pp. 225-236, 1997.

- [8] Peter Damaschke, "The Linear Time Recognition Of Digital Arcs", *Pattern Recognition Letters*, vol. 16, pp. 543-548, 1995.
- [9] Leo Dorst and Arnold W. M. Smeulders, "Best Linear Unbiased Estimators for Properties of Digitized Straight Lines", *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. PAMI-8, pp. 276-282, 1986.
- [10] Leo Dorst and Arnold W. M. Smeulders, "Discrete Representation of Straight Lines", *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. PAMI-6, pp. 450-463, 1984.
- [11] Leo Dorst and Arnold Smeulders, "Length Estimators for Digitized Contours", *Computer Vision, Graphics, and Image Processing*, vol. 40, pp. 311-333, 1987.
- [12] Herbert Edelsbrunner, David G. Kirkpatrick and Raimund Seidel, "On the Shape of a Set of Points in the Plane", *IEEE Transactions on Information Theory*, vol. IT-29 (4), pp. 551-559, 1983.
- [13] S. Fisk, "Separating Point Set by Circles and the Recognition of Digital Disks", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, pp. 554-557, 1986.

- [14] H. Freeman, "Boundary Encoding and Processing", *Picture Processing and Psychopictorics*, B.S. Lipkin and A. Rosenfeld, editors, vol. , pp. 241-266, 1970.
- [15] H. Freeman, "Boundary Encoding and Processing", *Picture Processing and Psychopictorics*, B.S. Lipkin and A. Rosenfeld, editors, vol. , pp. 241-266, 1970.
- [16] Robert Goldberg and Jonathan Robinson, "A Platform Independent Optical Character And Curve Recognition System", *Proceedings of the SPIE, Machine Vision Systems for Inspection*, B. G. Batchelor, J. W. Miller, S. S. Solomon, Ed., vol. 3521, pp. 342-350, 1998.
- [17] Tatsuya Hayashi, Koji Nakano and Stephan Olariu , "An $O((\log \log n)^2)$ Time Algorithm to Compute the Convex Hull of Sorted Points on Reconfigurable Meshes", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 12, 1998.
- [18] Jackiw, R. Nicholas. "Drawing Worlds: Scripted Exploration Environments in The Geometer's Sketchpad." *Geometry Turned On: Dynamic Software in Learning, Teaching, and Research*, ed. James R. King and Doris Schattschneider (The Mathematical Association of America, 1997): 179–184.
- [19] J. Koplowitz and A.P. Sundar Raj, "A Robust Filtering Algorithm for Subpixel Reconstruction of Chain Coded Line Drawings", *IEEE Transactions of Pattern Analysis and Machine Intelligence*, vol. PAMI-9 (3), pp. 451-457, 1987.

- [20] N. Megiddo, "Linear Time Algorithm for Linear Programming in R^3 and Related Problems", *SIAM J. Computing*, vol. 12, pp. 759-776, 1983.
- [21] Joseph O'Rourke, "An On-Line Algorithm for Fitting Straight Lines Between Data Ranges", *Communications of the ACM*, vol. 24 (9), pp. 574-578, 1981.
- [22] J. O'Rourke, S. R. Kosaraju and N. Megiddo, "Computing Circular Separability", *Discrete and Computational Geometry*, vol. 1, pp. 105-113, 1986.
- [23] Christian Ronse, "A Bibliography on Digital and Computational Complexity", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11 (2), pp. 181-190, 1989.
- [24] Rosenfeld and A.C. Kak, "Digital Picture Processing", , vol. , pp. , 1982.
- [25] Azriel Rosenfeld, "Digital Straight Line Segments", *IEEE Transactions on Computers*, vol. C-23, pp. 1264-1269, 1974.
- [26] Peter Sauer, "On the Recognition of Digital Circles in Linear Time", *Computational Geometry: Theory and Applications*, vol. , pp. 287-302, 1993.

- [27] Wendy Wilgus, "Exploring the Basics of Geometry With Cabri" Texas Instruments, 1998.

- [28] Yazid M. Sharaiha and Nicos Christofides, "An Optimal Algorithm for Traight Segent Approximation of Digital Arc", *CVGIP:Graphical Modeling and Image Processing*, vol. 55 (5), pp. 397-407, 1993.

- [29] Yazid M. Sharaiha and Phillipe Garat, "A Compact Chord Property for Digital Arcs", *Pattern Recognition*, vol. 26 (5), pp. 799-803, 1993.

- [30] A.W.M. Smeulders and L. Dorst, "Decomposition of Discrete Curves into Piecewise Straight Segments in Linear Time", *Comtemporary Mathematics*, vol. 119, pp. 169-178, 1991.

- [31] Donald L. Vossler, *Exploring Analytic Geometry with Mathematica*, Academic Press, 2000.

- [32] Marcel Worring and Arnold W.M. Smeulders, "Digitized Circular Arcs: Characterization and Parameter Estimation", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17 (6), pp. 587-598, 1995.

- [33] Marcel Worring, "Shape Analysis of Digital Curves", *Ph.D. Thesis, University of Amsterdam*, vol. , pp. , 1993.

- [34] L.D. Wu, "On the Freeman's Conjecture about the Chaincode of a Line", *Proceedings Fifth Conference on Pattern Recognition*, vol. , pp. 32-34, 1980.
- [35] P.C. Yeun, G.C. Feng, "A Novel Method for Parameter Estimation of Digital Arc", *Pattern Recognition Letters*, vol. 17, pp. 929-938, 1996.
- [36] Jianxing Yuan and Ching Y. Suen, "An Optimal $O(n)$ Algorithm for Identifying Line Segments from a Sequence of Chain Codes", *Pattern Recognition*, vol. 28, pp. 635-646, 1995.