POLYGON OVERLAY IN PROLOG

by

Peter Y.F. Wu

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
Major Subject: Computer System Engineering

Approved by the
Examining Committee:

[illegible]
Wm. Randolph Franklin, Thesis Advisor

Frank DiCesare
Frank DiCesare, Member

Robert B. Kelley
Robert B. Kelley, Member

[illegible]
Mukkai S. Krishnamoorthy, Member

Rensselaer Polytechnic Institute
Troy, New York

May, 1987
(For Graduation August 1987)

CONTENTS

LIST OF FIGURES

FOREWORD

"路漫漫其修遠兮
吾將上下而求索"
——屈原《離騷》

It has been a long process of growth for me as a graduate student in RPI. Apart from skills and knowledge, it has been a process of learning self-discipline, to take initiatives on my own, a process from dependence to independence. This thesis reports my work during the last two of those years. However, I do think that the training in me is much more important than the results in the thesis. Yet most certainly, its completion has brought much relief, and much to be thankful for.

I thank God for the opportunity and the ability to learn and grow.

I thank my advisor, Prof. Wm. Randolph Franklin, for his guidance, support and patience in the long course of my graduate work. He has brought me into the intriguing world of geometry and equipped me with modern computing tools. I thank also my committee members, Prof. F. DiCesare, Prof. R.B. Kelley and Prof. M.S. Krishnamoorthy, for reviewing my work. I thank Prof. G. Nagy for his encouraging comments and remarks in the past year, and I thank Prof. H. Freeman for my time as a system programmer in the Image Processing Laboratory under his directorate. His supervision has contributed to the awakening of my growth in those earlier years.

The work reported herein has been supported by the National Science Foundation under PYI grant ECS-83-51942, and the Data Systems Division of IBM Corporation. Their support is gratefully acknowledged.

I thank my fellow brothers at the *PARC* for their forbearance, in putting me up, and putting up with me. It has been a home away from home. I thank many brothers and sisters in the *ACCF* for their concern, their prayers and their emotional support.

I thank my parents for teaching me to be hard-working, by their own examples, and for their sustaining encouragement. To my father who did not stay to see this finished, this is dedicated to you.

PWu

To my beloved father
who taught me the virtue
of diligence and perseverance,
and is now rested in heaven.

ABSTRACT

A polygon overlay system is developed in Prolog. The complex process of polygon overlay is decomposed into a number of simple stages, resulting in much simplified data structures. The system in Prolog adopts a relational approach to data structuring. Geometric entities and their relationships are represented as Prolog "facts," and Prolog "rules" encoding geometry algorithms perform data processing. An adaptive grid sorts out potentially intersecting edge segments to within those that occupy some common grid cells. Geometric intersections are calculated using multiple precision rational arithmetic. Numerical accuracy is therefore preserved, and topological consistency guaranteed. Special cases of touching and partially overlapping chains are properly identified and handled. Stability in the computation process of polygon overlay is achieved.

Using Prolog is a venture motivated by the quest for better programming tools in computational geometry. The practicability and suitability of Prolog for geometry problems are investigated. Besides the general high level programming environment, and the relational approach to data structuring, specifically three paradigms of programming in Prolog are formulated. These are set-based operation, pattern matching geometric and topological properties, and using unification to form equivalence classes.

Chapter 1 INTRODUCTION

Polygon overlay is a process of superimposing two or more maps into one, so that the output map conveys the selected information of the input maps together to illustrate the spatial correlation between them. Automation of the polygon overlay process on the computer encompasses a number of geometric and topological computation problems. The problem originates from the field of computer cartography. We overlay two maps with different attributes from the same area to perform categorization of spatial information. A map, in this context, refers to a data structure representation of spatial relationships, and polygon overlay applies to maps using vector representations of spatial data.

This thesis presents a system developed in Prolog to perform polygon overlay. We decompose the process of polygon overlay into a number of stages, each of which performs certain local operations. The strategy simplifies data structuring and system design. Furthermore, our system achieves stability using multiple precision rational arithmetic to calculate geometric intersections. The calculation preserves numerical accuracy and thus guarantees topological consistency. Special cases of geometric intersection such as tangent and coincidence are properly handled.

One fundamental problem in dealing with geometry on the computer is the primitive nature of conventional programming languages. Conceptually simple ideas often turn out to be unexpectedly difficult to implement. The descriptive nature of Prolog provides a much more intuitive programming environment, and hence fosters more readable programs. More specifically, the system demonstrates several advantages offered by Prolog. These include data structuring, set-based operations, pattern matching geometric and topological properties, and using unification to transmit an equivalence relationship.

In what follows in chapter 1, section 1.1 presents a brief history of the polygon overlay problem arising in the field of cartography. In section 1.2, we review the literature disseminating the issues involved in the relevant areas of study. These areas include computational geometry, the special issues related to polygon and polyline intersection, the operational problems with nearly coincidental data and slivers, and with coordinates and computer arithmetic. We will also review some significant polygon overlay packages and systems. The use of Prolog introduces a new approach towards programming and implementation of geometry algorithms. Section 1.2.6 briefly describes the history and development of Prolog, leading to the applications in geometry topology problems. Section 1.3 identifies the scope of the research effort in the work of this thesis and summarizes the research contributions. Section 1.4, the last section in chapter 1, presents an overview of the rest of the thesis.

1.1 History of the Polygon Overlay Problem

Overlay is the process of superimposing two or more maps, so that the output map contains the information of the selected data items from different input maps. Before the early 60's, overlay was a process done manually. The approach was to make transparencies of the original maps, and to trace out the new map from the transparencies on a light table. Features of interest were traced out manually, polygon by polygon. Figure 1.1 illustrates the idea of polygon overlay with transparencies. As it is obvious, the process is very expensive and time consuming for complex situations (Sinton 72).

Computerized data processing did not come to aid this manual process until the 70's. The concept of polygon overlay may be intuitive, but automation of the process seems to be unexpectedly difficult. Fundamental to the difficulties involved are the problems of data structuring and algorithmic design. Studies of these problems were scanty in the field of geography and cartography in the early 70's. As geographic information systems began to develop in the late 70's, it became clear that the polygon overlay problem...

> ''...(is) the most complex problem of geographic data structuring...'' (Chrisman 76, p.6)
>
> ''...contains a number of challenging computational and statistical problems...'' (Goodchild 77, p.1)
>
> ''...has been one of the most interesting and challenging problems in computer cartography...'' (White 77, p.8)

Polygon overlay encompasses a host of many subproblems, mostly geometric and topological. The implementation on a computer also involves a number of operational problems. The following identifies certain categories of these subproblems:

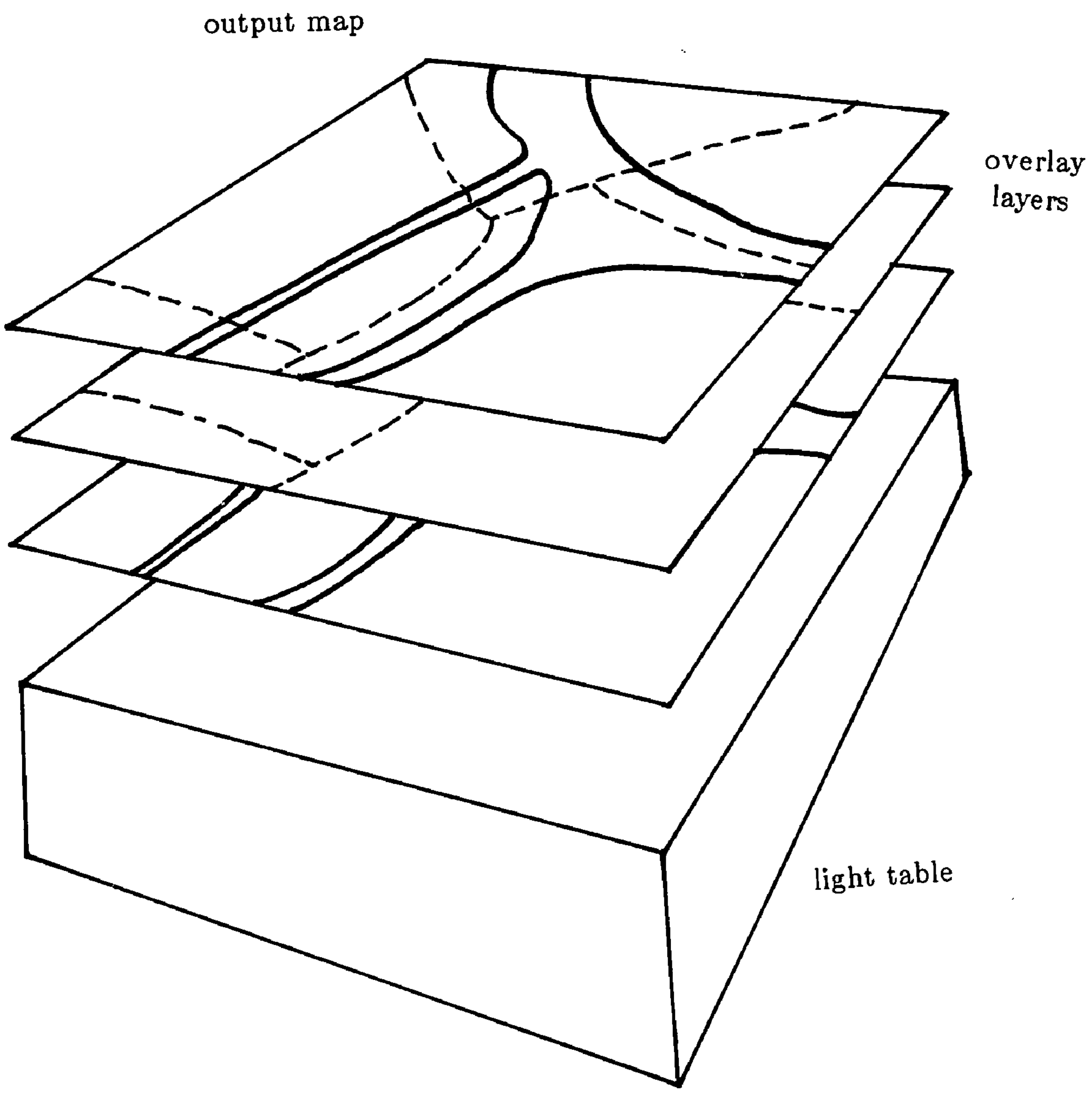


Figure 1.1 The manaul process of polygon overlay using transparencies.

- Boolean Combination of Sets
- Geometric Intersections
- Set Membership Classification
- Cartographic Line Generalization
- Geometry Data Structuring
- Slivers and Error Estimates
- Arithmetic and Representation of Coordinates

Research work which has directly and indirectly studied the polygon overlay problem by analyzing certain subproblems extends to areas such as computer graphics (hidden line/surface removal: Potmesil 80), geometric modelling (set membership classification: Tilove 80), computational geometry (algorithm design and analysis: Shamos 79), graph theory (mathematics of maps: White 79), computer arithmetic and algebra (discretization errors in computer cartography: Franklin 84). In the following section, we will review the literature in various areas contributing to problems involved with polygon overlay.

1.2 Review of the Literature

Fundamental to the development of a solution for polygon overlay is the research effort in computational geometry. Studies on polyline intersection were particularly important to the polygon overlay problem. Somewhat hidden but not less significant are certain operational problems in performing polygon overlay on the computer. We will discuss the problem of coincidental input data and that of numerical inaccuracy. In this section, we will review the literature in these areas. More, we will also report on packages and systems for polygon overlay. In closing, we will bring up the issue of using Prolog, which presents a radically different approach toward programming, for geometric/topological problems.

1.2.1 Computational Geometry

Algorithms to solve the basic problems of polygon intersection and the point in polygon inclusion were respectively reported in (Eastman 72; Franklin 72) and (Ferguson 73). During the 70's, many more algorithms for geometric intersection problems were developed for graphics rendering. Sutherland, Sproull, and Schumacker presented a study characterizing ten different hidden surface algorithms (Sutherland 74). Formal treatment in the analysis of geometry algorithms developed in the mid 70's. Shamos and Hoey reported algorithms to detect geometric intersections, with complexity analysis showing the problem to be asymptotically as difficult as sorting by comparison, and that the algorithms reported were optimal (Shamos 75,76). Bentley and Ottman extended the algorithms to count and report such intersections (Bentley 79). In the 80's, much work continued to flourish along the line of algorithmic design and analysis, setting the theme of theoretical computational geometry. These results were later gathered in a survey paper by Lee and Preparata (Lee 84), and much more comprehensively in the book *Computational Geometry* by Preparata and Shamos (Preparata 85). In his thesis, Guevara presented a formal treatment of the polygon overlay problem based on complexity analysis (Guevara 83). More recently, Guibas and Seidel presented their theoretical work on an algorithm and its analysis for a restricted version of polygon overlay - convex planar subdivisions using reciprocal search (Guibas 86).

Much of the work in computational geometry is concerned with the design and analysis of algorithms. The approach to efficient algorithms is often borrowed from the general techniques in the discipline, such as divide-and-conquer, recursion, and balancing (Aho 74). A different technique, uniquely and naturally suited to geometry problems, is the plane sweep (Nievergelt 82). The plane sweep approach, based on the total order defined in real numbers, often reduces the asymptotic complexity of an algorithm to that of sorting by comparison. While these results indeed reveal much about the nature of solving geometry problems on the computer, they often fall short of being directly applicable from an engineering standpoint. We note that the complexity analysis of an algorithm is almost always done for the worst case only. There is a need for more analysis of the average cases (Forrest 86). On the lack of average case analysis, Preparata and Shamos offered the two-fold reason:

> "first, substantial mathematical difficulties arise even when the underlying distribution is conveniently selected; second, there is frequently scarce consensus on a claim that the selected distribution is a realistic model of the situation being studied." (Preparata 85, p.9)

This is indeed the case for polygon overlay. Very few results on average case analysis of the polygon overlay problem were available. McAlpine and Cook used maps of regular hexagons in random orientation, displacement, and size to approximate the relationship between the number of output polygons and the number of input polygons. They derived the following formula (McAlpine 71):

$$N = (\sum_k \sqrt{M_k})^2$$

where N is the expected number of output polygons and M_k is the number of input polygons in the k-*th* map. Lam applied the formula and reported that the estimates tended to be 5% to 25% higher (Lam 77). Goodchild studied the statistical aspects and claimed that the number of output polygons created in an overlay process depends not on the number of input polygons, but on the complexity of each, defined by the vertices (Goodchild 77). Based on fractal mathematics (Mandelbrot 77), there were studies on characterizing monotonic divisions of a polyline (Dutton 81; Shelberg 82). Clearly, conclusive analysis on the average case behavior of the polygon overlay problem is difficult.

1.2.2 Polyline Intersection

Asymptotic complexity analysis has ascertained that the determining factor in the cost of polygon overlay is in polyline intersection (Guevara 83, p.111). In this section, we will focus on the development of algorithms and data structures for polyline intersection.

Obviously, the complexity of a polyline depends on the number of edge segments it contains. To avoid searching into the polylines to simply determine that they do not intersect, a common technique is to use an enclosing rectangle for each polyline, so that if the enclosing rectangles do not intersect, we can conclude that the polylines do not intersect. Enclosing rectangles with sides parallel to the X and Y axes are easy to form and easy to check for intersection. But of course there is the problem that when the enclosing rectangles overlap, the two polylines do not necessarily intersect. Freeman and Shapira developed a method to form the minimal area enclosing rectangle (Freeman 75), which would have less cases of overlapping rectangles and non-intersecting polylines. Little and Peucker introduced the enclosing band which allowed for recursive subdivision alternately on two intersecting polylines to determine the intersection points (Little 79). Burton, on the other hand, covered the polyline with a tree hierarchy of rectangles in a fixed orientation (Burton 77). Ballard incorporated both of these ideas into the "strip tree" — a binary tree in which each node represents a subdivision of the polyline enclosed in a strip (or band in Little and Peucker's terminology). The tree hierarchy facilitates binary search into the polyline structure and therefore divide-and-conquer techniques can apply (Ballard 81). Guevara proposed to decompose polylines into components monotonic in angle, forming sections between adjacent points of inflection (Guevara 83). The work however was not further disseminated.

Perhaps in view of the large data volume involved in map data processing, Denis White introduced the concept of local processing (White 77). The concept originated from the field of digital picture processing (Rosenfeld 69), making explicit the possible benefit of parallel processing. But in White's approach, a single processor instead scans through the input stream already arranged in pre-sorted order, operating only on a "local" section to ensure low temporary storage at any time. However, the significance of local processing in terms of computational geometry is the plane sweep approach in use. In fact, local processing in geographic information systems has predated complexity studies of plane sweep in computational geometry (Nievergelt 82). More recent theoretical work based on plane sweep is reported in (Mairson 87) on overlay of two sets of arcs of single-valued curves - a restricted version of polyline intersection.

Targeted toward a better average case performance, Franklin introduced the adaptive grid method for geometric operations (Franklin 83b). We used the grid to sort out potentially intersecting edge segments to within those which occupy some common grid cell. In chapter 3, we will discuss the adaptive grid method.

1.2.3 Slivers and Nearly Coincidental Data

How many output polygons can an overlay process produce? An overlay of two polygons, consisting of N_1 and N_2 edges respectively, can produce up to $2+N_1N_2$ output polygons. The maximal case occurs when every edge in one polygon crosses every other edge of the other polygon. Refer to figure 1.2 for an example. Input maps showing statistical independence will produce a much more moderate number of output polygons in the overlay process. Serious problems arise, however, when data in the input maps has a tendency to coincide. This happens often since prominent lines features such as rivers or major roads in one input map also appear in another input map as property lines or census tract

"How many polygons can overlay produce?"

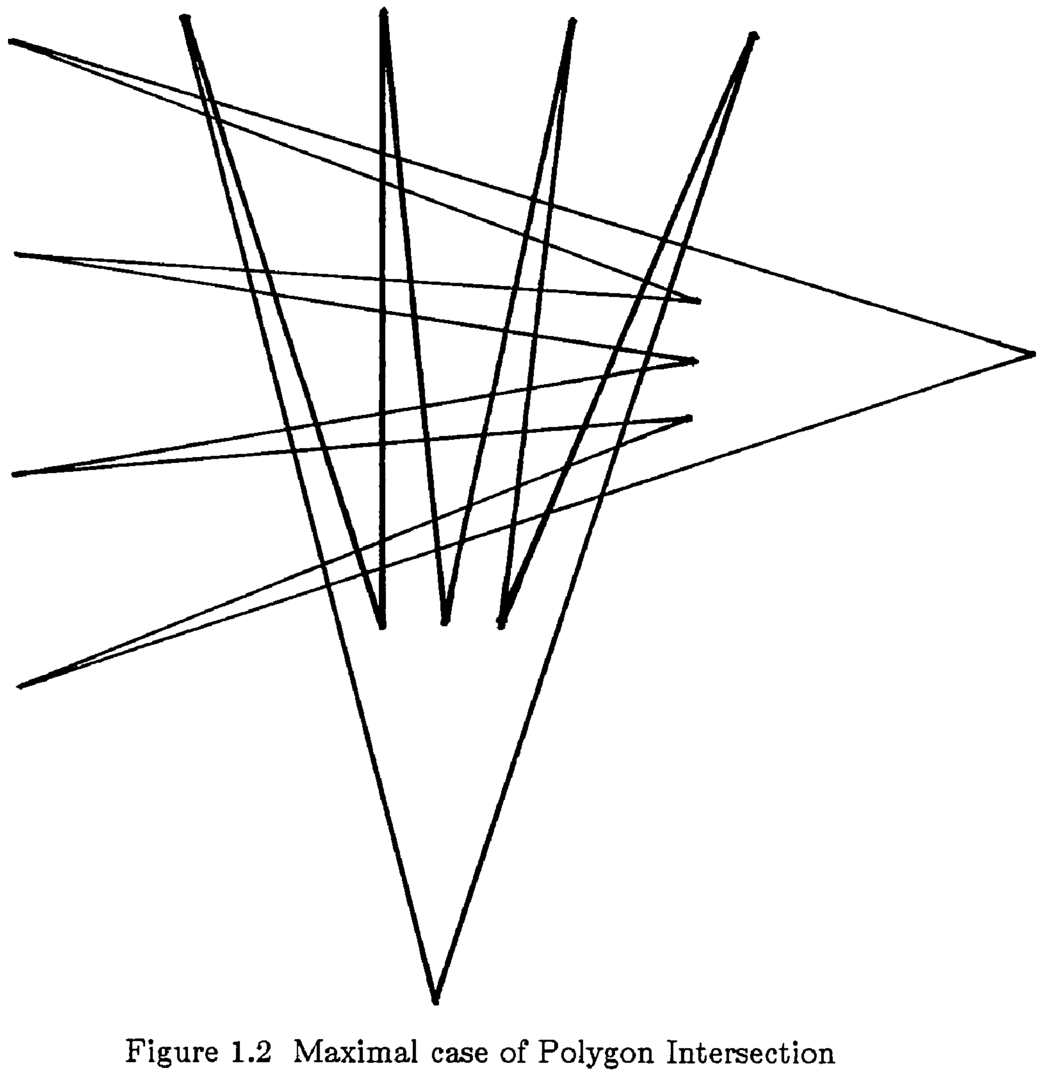


Figure 1.2 Maximal case of Polygon Intersection

boundaries. But the two versions of the same line feature do not coincide exactly, and the overlay process will produce many spurious polygons, slivers, along such line features. Goodchild established a measure of the expected number of slivers, $E(S)$, given by

$$E(S) = \frac{2\,N_1 N_2}{N_1 + N_2} - 3$$

where N_1 and N_2 are respectively the number of edge segments of the two chains for the same line feature (Goodchild 77).

Slivers should be removed as a human cartographer would put two nearly coincidental chains into one single chain in the overlay. Often slivers can be recognized by their small area. Two nearly coincidental chains may be recognized in the intersection process by the overlap of their narrow enclosing bands, using the polyline intersection algorithm due to Little and Peucker (Little 79). There was not much study reported on slivers in the past six years. The system presented in this thesis does not remove slivers. However, we will describe an outline of a strategy for sliver removal in section 8.2 for further considerations.

1.2.4 Coordinates and Arithmetic

Traditionally cartographers do not pay much attention to the accuracy of coordinates. They do not have to since coordinates are not represented in numerical values but graphically on the map. In an automated system, coordinates play an essential role in representing the geometry. However, calculated from measurements, coordinates are inherently subject to error, despite the precision level of the computer word used for storage. Marvin White called for attention to the proper use of coordinates and the identifiers of point features (White 83). Polygon overlay imposes another problem on the use of coordinates. Calculation of geometric intersections is subject to accuracy problems in floating point arithmetic on the computer, and overlay systems are therefore prone to unstability when dealing with tangent conditions. Studies on the peculiarities of floating point arithmetic were reported in (Malcolm 72; Gentleman 74). Franklin examined alternative models for arithmetic, including finite and multiple precision rational numbers (Franklin 84).

Rational arithmetic has been in use in many symbolic mathematics computation systems, most notably MACSYMA (Macsyma 83), which is in Lisp. Several Lisp versions, such as Franz Lisp (Foderaro 83) and Common Lisp (Steele 84), are built in with multiple precision integer (known as big number) arithmetic. The Unix system also provides a library of big integer arithmetic in C (Sun 86a) and tools are available for calculations using rational arithmetic (Sun 86b). We developed two packages in Prolog for big integers and rational numbers (Wu 86), which we used in our polygon overlay system for calculating geometric intersections. Chapter 4 will describe further details.

1.2.5 Polygon Overlay Packages and Systems

The Fortran subroutine package ANOTB (Franklin 72) is an early stride to polygon overlay. ANOTB takes two polygons and returns their Boolean combinations. OVER (Goodchild 74) is a stand-alone Fortran program which represents a more complex approach to polygon overlay. OVER first determines all the intersection points between the chains of the two input maps, and then constructs the output polygons of the overlay map. One of the oldest and once most advanced integrated system is the Canadian Geographic Information System - CGIS (Tomlinson 76). CGIS incorporates a combined grid/raster approach to polygon overlay, circumventing some of the geometric range searching and intersection problems. The Polygon Information Overlay System - PIOS (DeBerry 79) takes two sets of polygon records, and operates on two polygons at one time. One polygon is divided up into sixteen strips and PIOS uses a point-in-polygon test to determine the intersecting edges. The Map Overlay and Statistics System - MOSS (Reed 82) is similar to PIOS in that it operates on pairwise comparison of polygons. MOSS also performs pairwise comparison of edge segments to determine the intersection points. Then MOSS traces out the chains split by intersections to form the output polygons.

WHIRLPOOL (White 77), one of the programs in the ODYSSEY system (Dutton 77), represents by far a much more advanced approach to perform polygon overlay. One major contribution of ODYSSEY is the concept of local processing. Based on this approach WHIRLPOOL in effect exploits spatial coherence in performing a plane sweep in the problem space to report chain intersections. Guevara analyzed the performance of WHIRLPOOL and obtained the following worst case time complexity, which is optimal:

$$O(\ (N_1 + N_2) + (1 + k)\ log(\ N_1\ N_2\ )\ )$$

N_1 and N_2 are respectively the number of edge segments in each input map, and k is the number of intersections (Guevara 83, p.110).

1.2.6 Prolog and Geometry Projects

Since its inception in the 70's, Prolog has been used in a variety of applications. Among the best known are those for natural language processing, symbolic mathematical computation, compiler construction, and Prolog is becoming widely accepted as a major programming language for artificial intelligence applications. In the following, we will briefly review the history of Prolog, from inception to its proliferation. We will also review the work on using Prolog for geometry applications. An elaborate introduction to Prolog is in (Warren 79). A very concise descriptive note is in (Colmerauer 85). We refer the readers to (Clocksin 81) and (Sterling 86) as two comprehensive text books on Prolog programming.

During the 60's many were interested in programming computers to make logical deductions and prove theorems automatically. A significant mark of progress was Robinson's discovery of the resolution principle (Robinson 65). The resolution principle is an inference rule particularly suited to computing whereby one logical proposition can be shown to follow from others. The saga went on to the early 70's when Colmerauer attempted to apply logical deduction on the computer to do automatic translation between French and English. Colmerauer developed a prototype "programming language" (Colmerauer 73) to solve simple problems of logical deduction in children's stories. The prototype illustrated the important feature of treating functions as logical relations in a program.

The theoretical foundation of Prolog is based on the subset of predicate logic proposed by Kowalski. In a paper titled *Predicate Logic as a Programming Language* presented in the 1974 IFIP conference, Kowalski reported his studies on clausal forms of logic, showing that any problem can be expressed in logic as well as a particular subset of clausal forms (Kowalski 74). These were the Horn clauses (so named after the logician Alfred Horn). A set of Horn clauses can be shown to be logically consistent using the resolution principle. This theory which was implemented in the form first by Colmerauer constitutes the basis of Prolog.

The original version of Prolog was developed on a micro-computer. It was powerful but very slow (Roussel 75). David Warren implemented Prolog on the DEC System-10 (Warren 79), and this version, later known as the Edinburgh version, became instrumental in the spreading of Prolog. Prolog started its trip in Europe, and when the Japanese Fifth Generation Computer Project adopted Prolog as the major computer language for knowledge engineering in the intelligent computers of the future (Fuchi 81), Prolog began to proliferate.

Using Prolog for geometry problems is a venture in an attempt to answer the quest for more suitable tools to implement geometry and graphics systems (Forrest 86). Swinson reported studies in using Prolog for architectural design (Swinson 82,83). Gonzalez compared Prolog to Pascal and concluded in favor of Prolog for CAD applications (Gonzalez 84). Nichols investigated using Prolog for interactive computer graphics; a subset of Graphical Kernel System (GKS) was implemented in Prolog (Nichols 85). Using Prolog in two different approaches for geometric modelling was also reported: construction of objects defined by constraints (Bruderlin 85), and solid modelling in octree (Guerrieri 86). A more recent paper (Franklin 86) included a study of using Prolog in a range of geometry projects including 2D convex hull, polygon intersection, graph traversal, and photo reconnaissance inference. The paper also described an earlier version of the work involved in this thesis, on cartographic map overlay.

1.3 The Scope of Research Effort

In this section we will identify the scope of research effort in the work of this thesis. This includes using Prolog for geometry problems, and aspects of the polygon overlay problem - specifically the problem of topological inconsistency due to numerical inaccuracy, and the polygon containment problem. This section concludes with a summary of the research contributions.

1.3.1 Use of Prolog

Prolog represents a radically different approach toward programming. The approach has been commonly known as ''logic programming.'' Unlike conventional programming languages, a Prolog program does not present a prescribed set of instructions to the computer to solve a particular problem, but it describes the objects and their relationships involved in the problem to be solved. Prolog therefore provides a higher level, more intuitive programming environment, even though in practice one often has to be somewhat imperative. A fundamental difficulty in dealing with geometry on the computer is the primitive nature of conventional programming languages, and conceptually simple ideas often turn out to be unexpectedly difficult to implement. How much will Prolog help in filling this gap between concept and implementation? More specifically, we considered the issues of relational data structuring and paradigms of logic programming, both being features offered in Prolog. In our approach, we model as Prolog facts the geometric entities and their relationships, for the input/output maps and many interim data structures. The Prolog rules formulate geometric data processing. Through the experience, we identified features of Prolog which we deemed advantageous, and extracted the programming paradigms.

1.3.2 Polygon Overlay

Polygon overlay is an intricate problem and it encompasses a host of many subproblems. In this thesis, we focus in particular on the problem of numerical inaccuracy in calculating geometric intersections. The problem is important since it leads to topological inconsistency in the polygon overlay process. Rational arithmetic offers total numerical accuracy for geometric intersections. We exploited the flexibility in Prolog to implement rational arithmetic, to illustrate its practicability. Since accuracy is preserved, basic intersection algorithms can be extended to properly identify and handle all special cases.

The polygon overlay algorithm implemented is due to Franklin. The algorithm decomposes the complex process of polygon overlay into a number of simple stages, resulting in simplified data structures (Franklin 83a). To the knowledge of the author, the algorithm has not been implemented in any system disseminated in the public literature. I implemented the algorithm. Further, resolving the polygon containment problem, I extended the algorithm to handle also polygons with holes and maps with separate components.

1.3.3 Research Contributions

Based on the preceding outline of the scope of research effort in this thesis, we summarize the contributions in the following:

(1) Using polygon overlay as a vehicle, we investigated the practicability and suitability of Prolog for geometry applications. Other than the general high level nature and the relational approach to data structuring in Prolog, we identified these features, namely, set-based operations, pattern matching geometric and topological properties, and unification to form equivalence classes, as useful paradigms of programming in Prolog, being advantages as a programming tool for geometry problems.

(2) The system presented in this thesis illustrated the use of rational arithmetic for geometric intersection problems. Rational arithmetic preserves total accuracy, and topological consistency is therefore guaranteed.

(3) With numerical accuracy preserved, we were able to extend the line segment intersection algorithm to cover all special cases, properly identifying and handling coincidental and tangent conditions. This combined with (2) above provided for stability in the polygon overlay process.

(4) We resolved the polygon containment cases and thus extended the polygon overlay algorithm due to Franklin to handle input maps with separate components (and polygons with holes).

1.4 An Overview of The Thesis

In chapter 1, we have reviewed the related literature and summarized the research contributions.

Chapter 2 will describe the polygon overlay system developed in Prolog. The system decomposes the process of polygon overlay into a number of stages. We begin with definitions and data structures, and present an overview of the system. Each stage of the system is then further subdivided for more detailed description.

Chapter 3, 4 and 5 discuss three major features in the system: the adaptive grid, rational arithmetic, and special cases handling. The adaptive grid is a strategy to speed up computing chain intersections. In chapter 3, we describe the strategy as an approach toward distribution sort to avoid pairwise comparison between individual edge segments. An analysis of its performance based on random edge segments is presented. Chapter 4 is on rational arithmetic: we discuss the problem of numerical inaccuracy in geometric intersection and the mathematics of rational arithmetic as a solution to the problem. We describe the implementation in Prolog, and discuss the cost in terms of CPU resources in using rational arithmetic. In chapter 5, we will develop an algorithm for edge segment intersections, complete with special cases handling. Exact numerical results due to rational arithmetic enable us to properly identify these special cases. With complete special cases handling, stability in the polygon overlay process is achieved.

Chapter 6 is concerned with the use of Prolog: motivated by the quest for better programming tools, we venture into the experiment of using Prolog for our polygon overlay system. In chapter 6, we describe our experience with Prolog and illustrate some of its features with examples from the polygon overlay system. Specifically we have formulated several paradigms of programming which appear to be useful in general.

Chapter 7 documents the implementation; the Prolog source listing of the entire polygon overlay system is appended to the thesis. We describe the various steps in the programs. In chapter 7, we also demonstrate the results from two sets of test runs on our system. One set is on testing the stability; the other is on general performance.

Chapter 8 closes with a summary of the thesis, and in conclusion, reiterates the research contributions. More, further considerations are discussed. These include sliver removal, map data verification, and map generalization.

Chapter 2 A POLYGON OVERLAY SYSTEM IN PROLOG

Chapter 2 presents our polygon overlay system developed in Prolog. In chapter 1, we have briefly described the polygon overlay problem. In chapter 2, we will begin with the fundamental geometric entities first and then give a definition for the polygon overlay problem in section 2.1. Section 2.2 will explain the input and output file structures. To introduce the system, section 2.3 presents with an overview first, and the rest of the chapter will describe further details in each of the stages in turn. Chapter 3, 4 and 5 will follow up to discuss the major features of the system.

2.1 Definitions

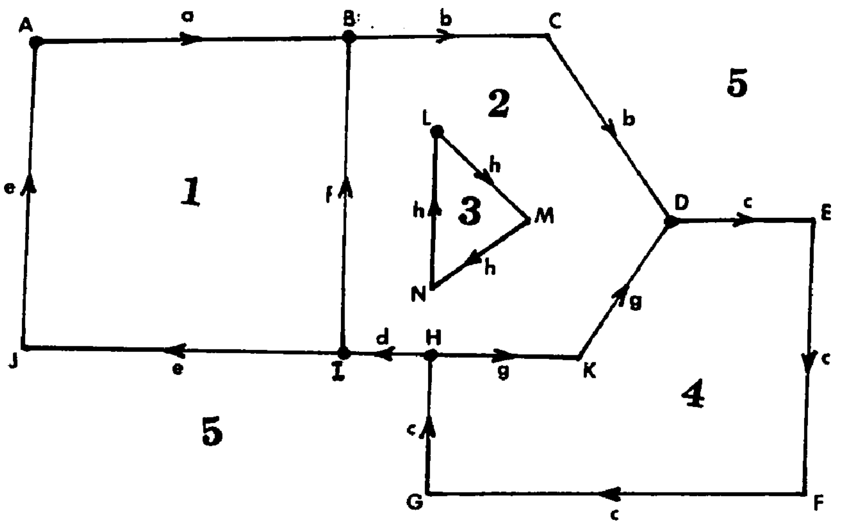
A map refers to a 2D spatial data structure, consisting of three generic classes of geometric entities, namely, node, chain, and polygon. A node is a point in the 2D plane which is a topological junction. A chain is a directed polyline structure - a sequence of contiguous and non-intersecting line segments in the plane beginning at a node and ending at a node. The two nodes may or may not be the same node. A chain does not intersect with any other chains in the same map, nor with itself. Two chains may touch each other only at a node where they begin or end. Hence, the network of nodes and chains in a map partitions the 2D plane into regions, called polygons. Each polygon is a connected subset of the 2D plane. A polygon is bounded by one or more sequences of alternating nodes and chains; each sequence forms a complete cycle, that is a simple closed curve. The cyclic order determines whether the interior of the polygon is to the inside or outside of the boundary. Figure 2.1 shows a map illustrating the ideas aforementioned.

Given two maps A and B, the polygon overlay process produces an output map C by superimposing A and B. Since chains of map A may intersect those of map B, new nodes are formed at the intersection points. The output map C contains all the nodes of A and B, and the new nodes generated. Map C contains all the chains of A and B; those involved in intersection are split up at the new nodes, that is, the intersection points. The network of chains and nodes in map C partitions the plane into polygons, each of which is a resultant polygon from the intersection between a polygon in A and another one in B. An overlay relationship associates each polygon in C with these two polygons in A and B. The polygon overlay process constructs output map C, and establishes the overlay relationship. Figure 2.2 illustrates two input maps, the overlay output map, and the overlay relationship.

In the context of our definition, we have assumed consistency in the input map data, and that the maps are properly registered in scale and orientation for the overlay process.

2.2 Input/Output and File Structures

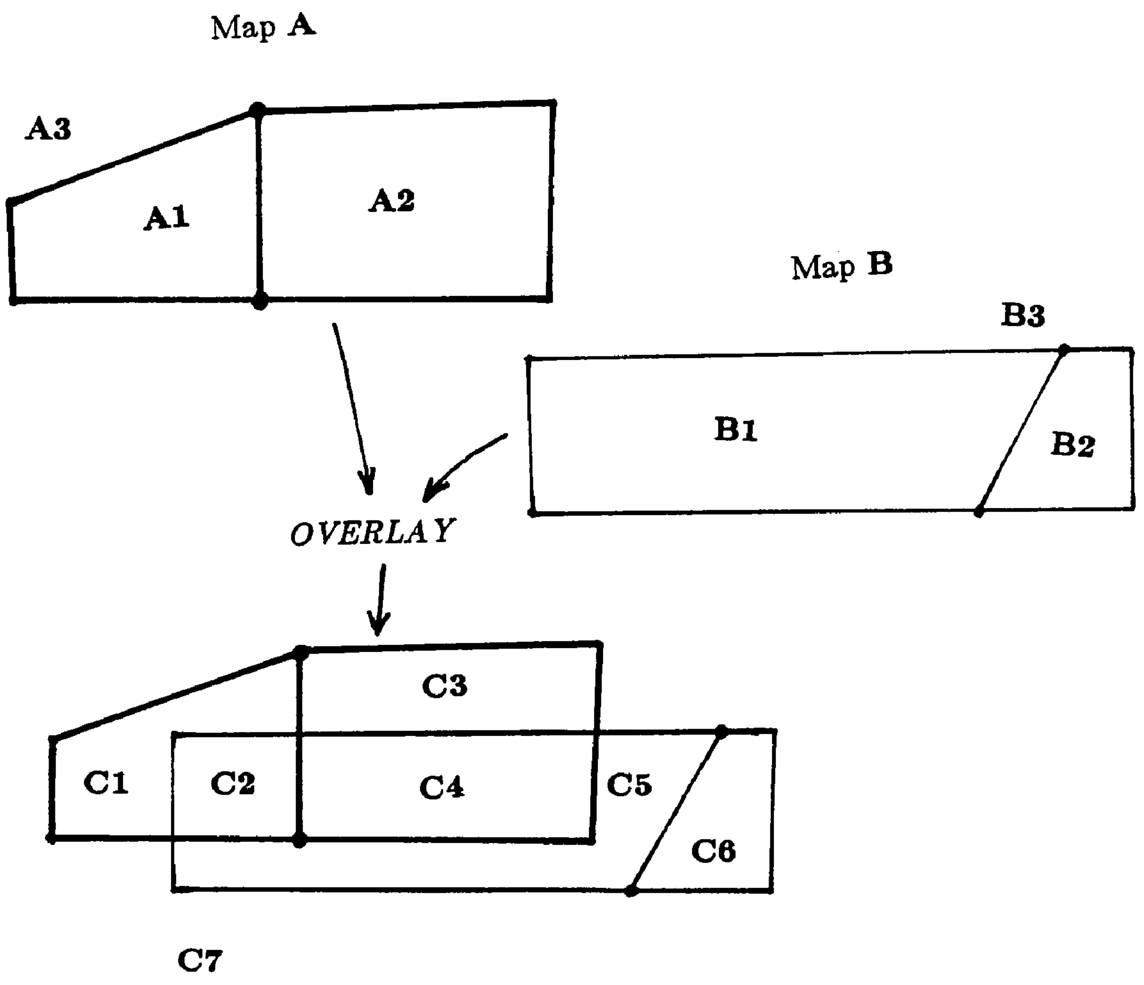
A map as defined above consists of nodes, chains, and polygons. It however suffices to represent a map with the set of chains complete with the following set of attributes:



nodes: **A,B,D,H,I,L.**

chains:		polygons:	
a [A,B]			
b [B,C,D]			**#1**[*ht*(a),*ht*(e),*th*(f)]
c [D,E,F,G,H]			**#2**[*ht*(b),*ht*(f),*ht*(d),*th*(g)]
d [H,I]			**#2**[*th*(h)]
e [I,J,A]			**#3**[*ht*(h)]
f [I,B]			**#4**[*ht*(g),*ht*(c)]
g [H,K,D]			**#5**[*th*(a),*th*(b),*th*(c),*th*(d),*th*(e)]
h [L,M,N,L]			

Figure 2.1 Nodes, Chains, and Polygons in a Map



Overlay Relationship: *over*(C1,A1,B3).
over(C2,A1,B1).
over(C3,A2,B3).
over(C4,A2,B1).
over(C5,A3,B1).
over(C6,A3,B2).
over(C7,A3,B3).

Figure 2.2 Polygon Overlay and the overlay relationship.

- MAP: map id;
- $C\#$: chain id;
- N_1,N_2: beginning and ending node id's;
- [$[x,y]$,...]: polyline in a list of (x,y) coordinates from N_1 to N_2;
- P_1,P_2: polygon id's to the left and right of the chain.

The nodes and the polygons can be derived from the chains as defined above. An input map is therefore set up as a collection of variable length records, represented as **chain** facts in Prolog:

chain(MAP,$C\#$,N_1,N_2,[$[x,y]$,...],P_1,P_2).

The output map from polygon overlay consists of a similar set of chains, with a new map id. Although redundant, the process also produces for output a set of **node** facts and a set of **polygon** facts:

node($N\#$,$[x,y]$,[I_1,I_2,...]).

$N\#$ is the node id; $[x,y]$ the coordinates of the node $N\#$. [I_1,I_2,...] is the list of chain incidences in positive cyclic order about the node. Each chain incidence I_i can be $h(C_i)$ for head (ie, starting node), or $t(C_i)$ for tail (ie, ending node).

polygon($P\#$,[B_1,B_2,...]).

$P\#$ is the id of a polygon. Each **polygon** fact identifies a polygon boundary: $P\#$ is therefore not unique for each fact, but identifies a boundary of the polygon. [B_1,B_2,...] is the list of boundary chains in proper cyclic order around, identifying a polygon boundary: positive cyclic order indicates that the polygon $P\#$ is to the inside, and negative, to the outside. Each chain boundary B_i can be $th(C_i)$ or $ht(C_i)$, for tail-to-head and head-to-tail respectively, indicating the direction of the chain in the polygon boundary.

The polygon overlay process also resolves the overlay relationship between the polygons of the output map and those of the input maps. The set of **over** facts establishes the association:

over$(P\#,P_1,P_2)$.

$P\#$ is the unique identifier for each output polygon; P_1 and P_2 are the id's of the source polygons in each of the input maps. $P\#$ is generated in the overlay process by the intersection of P_1 and P_2.

To sum this up, the polygon overlay system requires an input file of chains for each input map. For the output map, four files are generated: chains, nodes, polygons, and the overlay relationship with the input polygons. During the process of polygon overlay, the system also generates different interim data files for use in various subsequent steps. We will explain these interim data files as we proceed on to describe the steps involved in the process.

2.3 Description of The System

Our system divides the polygon overlay process into four major stages. Each is further subdivided into a number of steps. The following presents an overview and a brief description of the four different stages in the process:

Stage 1. Chain Intersection.

Determine intersecting chains and split them at the intersection points, generating new nodes.

Stage 2. Polygon Boundary Formation.

Form the polygon boundaries by linking up the network chains.

Stage 3. Polygon Overlay Identification.

Identify for each polygon boundary the source polygons of the input maps and establish the overlay relationship.

Stage 4. Boundary Containment Resolution.

Resolve the containment relationships between polygon boundaries to identify the polygons with multiple boundaries.

We are going to describe each stage in further details in this chapter. There are a few major components and features, which we will describe in the chapters to follow.

2.3.1 Chain Intersection

Given two sets of chains, each forming a map, we want to determine the intersecting chains and split them at the intersection points. In search of intersections, we observe that when two chains intersect, the intersection occurs between two edge segments which occupy the same locality. Our approach breaks down each chain into its edge segments and casts a grid over the edge segments. The intention is to isolate cases of potential intersections to within those edge segments that occupy the same grid cell. Figure 2.3 illustrates the idea. We can then determine the intersecting edge segments by pairwise comparison. The strategy is called the adaptive grid method (Franklin 83b) since the appropriate grid cell size is determined by statistical measures from the two input maps. The process involves the following steps:

Step 1.1 Determine the appropriate grid cell size for the adaptive grid.

Step 1.2 Cast the grid over the edge segments; determine for each edge segment the grid cells it occupies and collect the edge segments into sets for each grid cell they occupy.

Step 1.3 Determine intersections in each set of edge segments by pairwise comparison; split the edge segments and generate a new node for each intersection.

Step 1.4 Connect the edge segments back to chains, already split at the intersection points.

In step 1.1, we examine the input chains and break them up into the constituent edge segments, each identified by the edge id *E#*. This forms the **edge** file:

$$\textbf{edge}(E\#, [V_1, V_2]).$$

In step 1.2, we distribute these edges to the grid cells each edge occupies. This step generates the **grid** file consisting of one entry for each grid cell,

When two chains intersect,
they do so between two edge segments
which occupy the same locality.

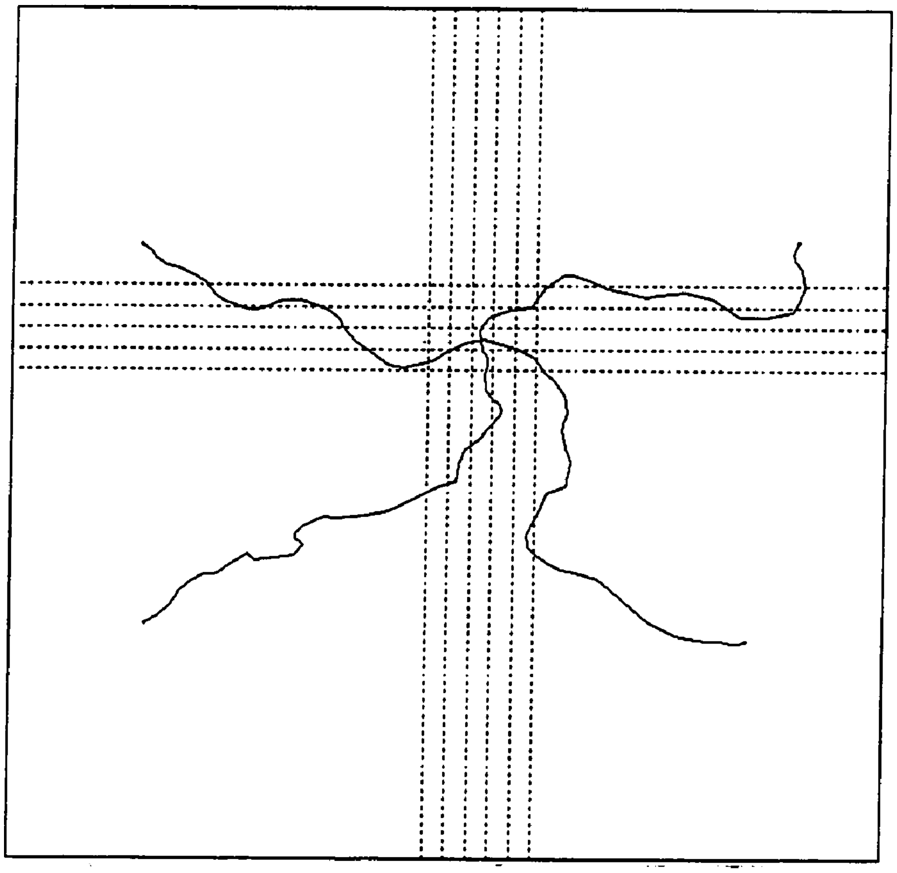


Figure 2.3 The adaptive grid isolates potentially intersecting edge segments.

identified by $G\#$, and the set of edge segments for each entry:

grid($G\#,[E_1,E_2,...]$).

In step 1.3, we go through the **grid** file sequentially, and determine intersections by pairwise comparison between the edge segments in each grid cell, making references to the **edge** file. Intersecting edges are split at the intersection points. In step 1.4, we connect the edge segments back to chains. These are the network chains for the output map:

chain($C\#,N_1,N_2,[\ [x,y],...]$).

They are called network chains because we do not yet have the information for the adjacent polygons.

Chapter 3 will discuss further our strategy in using the adaptive grid for chain intersection. Calculation of geometric intersection is done using multiple precision rational arithmetic, which we have implemented in Prolog. Chapter 4 will describe the rational arithmetic package. We have limited the handling of special cases of chain intersection to only the low level operations, which in our case is in intersecting edge segments. Chapter 5 will develop an algorithm for edge segment intersection complete with special cases handling.

2.3.2 Polygon Boundary Formation

The chains and nodes partition the 2D plane into polygons. We link up the network chains from the previous stage around the polygon corners at each node. Each completed cycle of chain linkages forms the boundary of a polygon. This stage involves the following steps:

Step 2.1 Identify for each chain the nodes where it begins and terminates, and calculate the incident angles.

Step 2.2 Sort the chains by the incident angles around each node into proper cyclic order. Each adjacent pair of chains identify a corner of an output polygon at the node. Form linkage of the adjacent chains in the specified direction.

Step 2.3 Match the linkages and connect them until each list of directed chains begins and ends with the same entry; each list identifies the boundary chains of a polygon in its proper cyclic order.

Step 2.1 generates the chain-to-node **incidence** file. Each incidence entry has the following format:

i(*Node#*,*Incidence*,*ANGLE*).

Node# is the node identifier. *Incidence* can be *h*(*C#*) or *t*(*C#*), to indicate the head or tail of chain *C#*. *ANGLE* is the incident angle the adjacent vector of the chain makes with the positive X-axis at the node. Step 2.2 then collects the **incidence** entries at each node and sorts them out by the *ANGLE*. Then we have the **nodes** of the output map:

node(*Node#*,[x,y],[*list of chain incidences...*]).

Around each node, adjacent chain incidences identify a polygon corner.

We generate a **linkage** file, with one entry for each polygon corner:

linkage(B_1,B_2,[B_1,B_2, ...]).

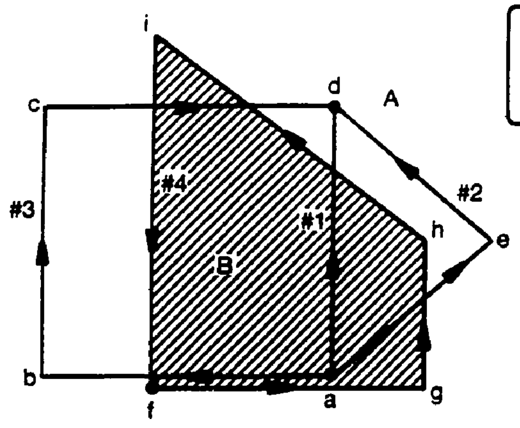
We call it **linkage** since we intend to link them up to form the polygon boundaries. B_1 and B_2 are chains marked with a specified direction: *th*(*C#*) or *ht*(*C#*) to indicate the chain *C#* from tail to head or from head to tail. The list [B_1,B_2] will be **link**ed with other lists. B_1 and B_2 identify the first and last elements of the list.

In step 2.3, we connect two linkages if they match together. Two linkages match if the last entry in one list is the same as the first entry in the other list. The concatenated list becomes a new, and longer, linkage. For example, **linkage**(B_1,B_2,[B_1,B_2]) and **linkage**(B_2,B_3,[B_2,B_3]) are connected to form **linkage**(B_1,B_3,[B_1,B_2,B_3]). When no more linkages match, we should have each linkage beginning and ending with the same chain entry. Then each of the list [B_1,B_2,B_3,... ,B_1] identifies a polygon boundary. We generate new polygon names for each boundary list for the **polygon** file:

polygon(*P#*,[B_1,B_2,B_3,...]).

We should note that dangling chains and bridges do not present any difficulty to the process. They end up to be in the same boundary list in both directions. Figure 2.4 demonstrates the process of linking up directed chains to form polygon boundaries.

Once we have the polygons, we can also re-format the network chains (from the previous stage) into complete chains with the information

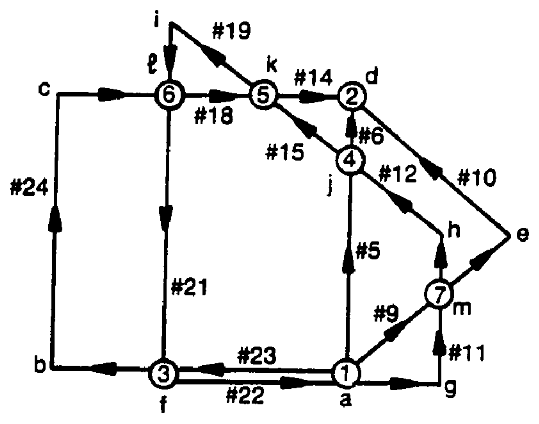


Map A
chain #1: [a, d]
chain #2: [a, e, d]
chain #3: [a, b, c, d]

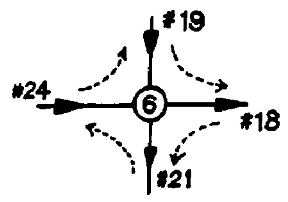
Overlay Example

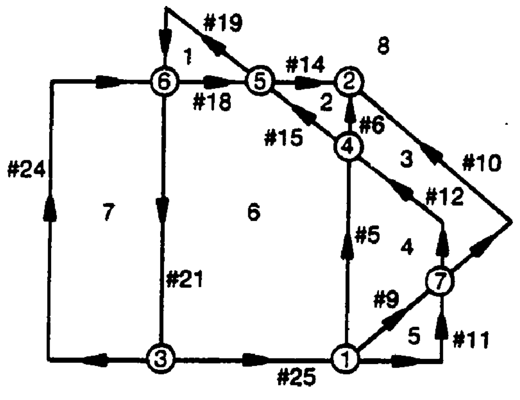
Map B
chain #4: [f, g, h, i, f]

Edges split by intersection



Sort Chains at each node by incident angles





Match linkages to form polygon boundaries

```
polygon(1, [ht(18), ht(19)])
polygon(2, [ht(6), th(14), th(15)])
polygon(3, [th(6), th(12), ht(10)])
polygon(4, [th(5), ht(9), ht(12)])
polygon(5, [ht(11), th(9)])
polygon(6, [ht(5), ht(15), th(18), ht(21), ht(25)])
polygon(7, [th(21), th(24)])
polygon(8, [ht(24), th(19), ht(14), th(10), th(11), th(25)])
```

Figure 2.4 Linking chains to form polygon boundaries

for the adjacent polygons. These are then the complete chains for the output map.

2.3.3 Polygon Overlay Identification

Each output polygon is the intersection of two input polygons, one in each input map. However, the polygon boundaries may or may not be involved in intersection. If the list of boundary chains is involved in any intersection, it consists of chains from the two different input polygons. We can therefore determine the input polygons by examining the chains in the boundary list. If no chains in the boundary list is involved in any intersection, all the chains comes from one polygon in one of the input maps, the polygon being completely contained in another polygon of the other input map. We can only determine one input polygon from the chains in the boundary list; the unknown one is the containment polygon. Observe, however, that all of its neighbors must also be inside the same containment polygon of the other input map. Figure 2.5 illustrates this. We can then recursively search its neighbors for the containment polygon.

The recursive search fails only when the entire connected group of polygon boundaries are not involved in any intersection with the other input map. In this case, we identify the possible containment polygons from the other input map, and determine the containment polygon by testing each one with the point-in-polygon test.

The following outlines the steps to determine the overlay

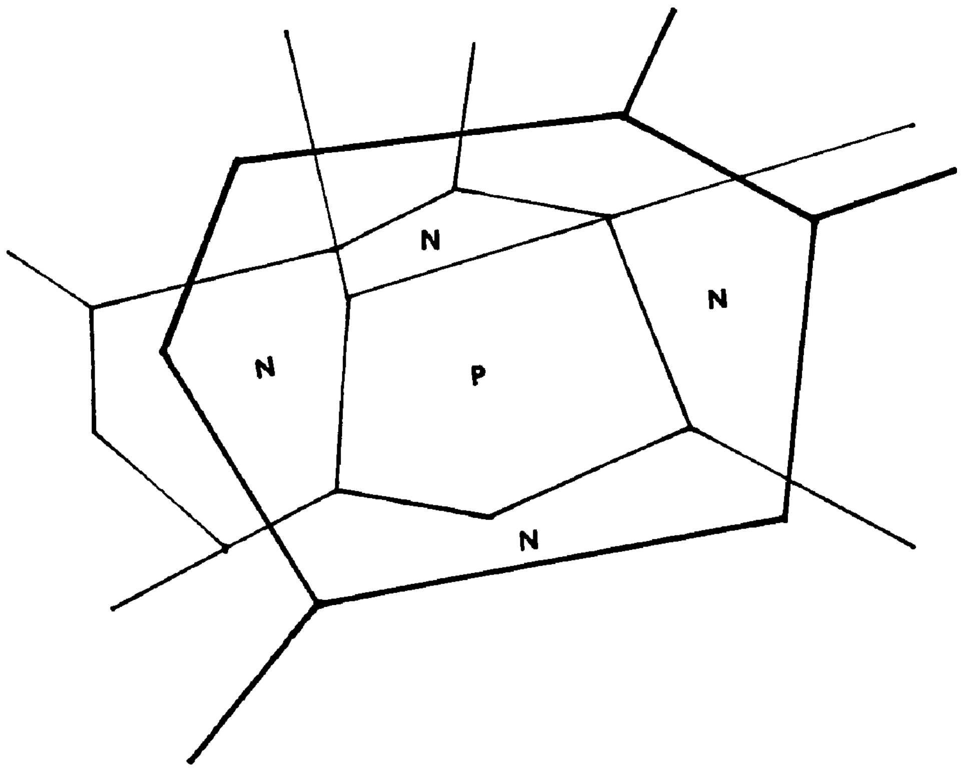


Figure 2.5 All neighbors are inside the same containment polygon.

relationships between the input and output polygons:

Step 3.1 For each output polygon boundary *P#*, if involved with intersection, determine the input polygons P_1 and P_2; if not involved with any intersection, recursively search the neighbors to determine the containment polygon.

Step 3.2 If the search fails, identify the connected group of neighboring polygons together.

Step 3.3 For each group, identify the polygon boundary which refers to the outside polygon in the input map. One of the polygons in the other input map which intersect with the outside polygon is the containment polygon we are looking for. Take a point from the group for point-in-polygon test to determine the containment polygon.

Step 3.1 generates the file for overlay relationship entries. For each output polygon *P#*, an entry has the format:

over(*P#*,P_1,P_2).

P_1 and P_2 are the two input polygons from the two input maps, such that *P#* is from $P_1 \cap P_2$. If any search for the containment polygon fails within the group of connected neighbors, the **over** file cannot be completed in step 3.1, and will have to be appended later on. Step 3.2 generates two files, one for each input map:

over1(*Group#*,*P#*,P_1). and **over2**(*Group#*,*P#*,P_2).

Each entry identifies an output polygon *P#* known to be in one input map (as P_1 or P_2), but its containment polygon in the other input map is still unknown. Each *Group#* identifies a group of connected neighbors together. Step 3.3 resolves the unknown containment polygon by point-in-polygon test for each connected group. The results are then appended to the **over** file for overlay relationships.

2.3.4 Boundary Containment Resolution

Two polygons may intersect to form more than one polygon. Further, we have defined a polygon to be a connected subset of the 2D plane, each polygon may have more than one boundary (some boundaries being holes). Figure 2.6 illustrate one case of two polygons intersecting to form two polygons each with one hole. While we have determined the input polygons for each output polygon boundary, we must also determine which of the boundaries refer to the same connected piece of output polygon.

We examine the polygon boundaries which are formed from the same pair of input polygons. We then divide them into two groups by their cyclic order: positive polygon boundaries, and the holes which are the negative ones. Each positive boundary identifies one connected polygon. The negative ones are each a hole contained in one of the positive polygons. The exception is the outside polygon which is the overlay of the outside polygons in the two input maps; it contains only holes and no positive polygons. The following describes the steps to resolve the

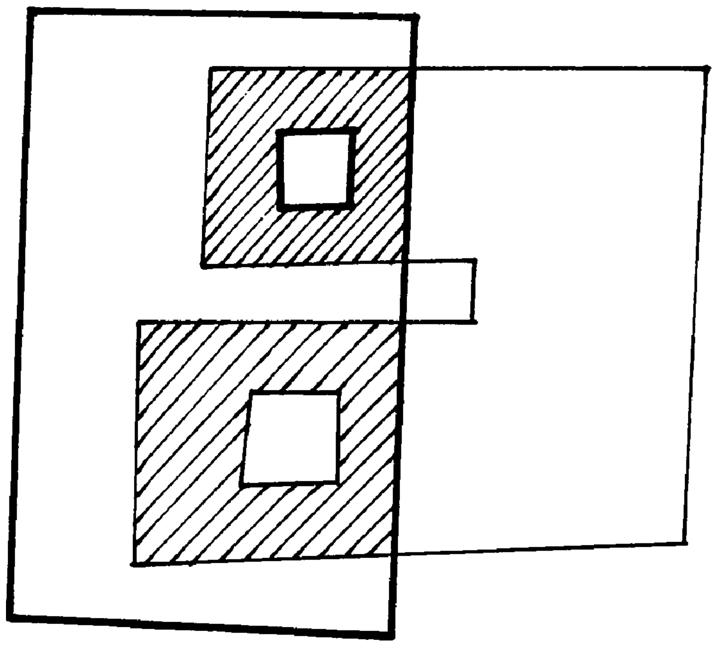


Figure 2.6 Intersection of two polygons producing two polygons with two holes.

containment relationships and identify the holes for each polygon:

Step 4.1 Gather the polygon boundaries which are formed from the overlay of the same pair of input polygons.

Step 4.2 Divide into two groups according to the boundary cyclic order: positive ones are polygons and negative ones are holes.

Step 4.3 Determine the containment polygon for each hole *P#* using the point-in-polygon test. Update the polygon id for *P#* to that of its containment polygon.

After these steps, we can update the output files of chains, polygons, and overlay relationships. If a polygon is found to be a hole in another polygon, it should take the polygon id of its containment polygon. All its boundary chains will have to be updated and the entry in the overlay relationships can be deleted, since the entry for the containment polygon is already there.

2.4 Process Decomposition

A theme in our design philosophy we would like to note here is process decomposition. Process decomposition simply means the dividing up of a complex process into less complicated steps. But it is a powerful concept in both software and hardware architecture. In our system, we have divided up the polygon overlay process into quite a number of steps, organized into four major stages as we have presented here in this chapter.

The benefit that is obvious here is in data structuring. Each step performs a certain operation to its input files and generates its output files. There is no complicated nesting of process or searching of data structures. This makes our system unique among the other systems we surveyed in chapter 1. For example, we will not have to keep a set of open chains as in WHIRLPOOL (White 77) but can still maintain local processing. Nor do we have to deal with strips of a polygon as in PIOS (DeBerry 79).

Another benefit of process decomposition is modularity. While we have a number of different modules, each performing an operation for one step in the process, these modules can be organized in other ways for other purposes. There is possibility of re-use for every module. The polygon overlay system can be modified to perform map data verification. A module for sliver elimination can be added immediately after we form all the output polygons and before examining their overlay relationships. Chapter 8 will discuss these in further considerations.

Process decomposition also makes suitable candidates for pipe-lining in a multi-processor environment. Although we have not taken advantage of this in our project, we refer to it here since it is also one of the benefits of the same design philosophy.

Chapter 3 THE ADAPTIVE GRID

Given two sets of chains, each forming a map, we want to determine the intersecting chains and split them at the intersection points. This is the polygon overlay version of the polyline intersection problem. This problem is also the most costly step in the entire process of polygon overlay. We implemented the adaptive grid approach to tackle the problem. In this chapter, we will classify the polyline intersection algorithms mentioned in the literature survey, and present the adaptive grid method as a different approach to the polyline intersection problem. We will also present an average case timing analysis for random edge segments. The question concerning the performance of the adaptive grid for polygon overlay remains open. We do not have an answer in this thesis, but will discuss the validity of the assumptions made in the analysis. We have quite a range of freedom in choosing an appropriate grid cell size, and we will discuss that. The benefit along with the local processing concept in the adaptive grid method is also presented.

3.1 Polyline Intersection

We surveyed a number of different algorithms for polyline intersection in section 1.2.2. In the course of the development, we see that the polyline intersection problem, although much more complicated, in many ways resembles a sorting problem. Intuitively, the problem involves both pairwise comparisons between polylines as well as those between the constituent edge segments. Algorithms can be classified into two groups: the first group deals with intersection between two polylines. Methods testing the enclosing rectangles eliminates some of the cases of the need to examine the edge segments in the polylines. The recursive algorithm using an enclosing band (Little 79) exploits the contiguous nature of the constituent edge segments to minimize pairwise examination, resulting in a linear worst case time complexity. The Binary-Search-Polyline-Representation (Burton 77) forms a hierarchical structure to perform binary search into the polyline to locate an intersecting edge segment, achieving the $log(N_1+N_2)$ worst case time complexity. The strip-tree (Ballard 81) integrates the two ideas together, and simplifies the data structure. These algorithms work with two polylines at a time and do not tackle of problem of pairwise comparisons between polylines. The second group of algorithms deal with the entire set of polylines and attempt to exploit spatial coherence in the total ordering of the real space: an approach commonly called the "plane-sweep" (Nievergelt 82) in computational geometry. This is the local processing concept in (White 77) which pre-sorts the input polylines and then scans over the polylines in the pre-sorted order, operating in a "local" section of the object space at one time. It is then necessary only to examine those polylines in the same local section at the same time for potential intersections. This approach achieves the optimal worst case time complexity.

3.2 The Adaptive Grid

In the adaptive grid approach (Franklin 83b), we deal with the entire set of polylines all at one time. But we also deal with individual constituent edge segments disregard of the number of polylines. Instead of treating the polyline intersection problem like sorting by comparison, the adaptive grid method sets up a way to perform sorting by distribution.

The adaptive grid is a regular, rectangular grid imposed onto the object space. It is adaptive because the grid cell size is determined by statistical measurements from the input data set. Once the grid is set up, each grid cell serves as a bucket in sorting by distribution. Since the grid is regular, it is a constant time operation to locate the grid containing any point in the object space. For an edge segment, we can determine and report the grid cells it occupies in time linear to the number of grid cells occupied (Foley 82). The total time required to distribute all the input edge segments to the buckets for the grid cells occupied is therefore linear to the total number of (grid-cell, edge-segment) pairs. We have to be more careful with the data structure for the adaptive grid so that empty grid cells will not take up any storage space. In Prolog, each non-empty grid cell is represented as a fact with a unique functor name for each cell. For example, grid cell #344 is the Prolog fact

$$g344([E_1,E_2,...]).$$

where $[E_1,E_2,...]$ is the set of edge segments in the bucket. Each bucket is treated as a set and access to the buckets is handled by linear hashing.

When we have finished the pre-processing step of distributing the edge segments to the grid cells, we need to examine only the ones which share at least one common grid cell for potential intersection. We do so by pairwise comparison between the edge segments in each non-empty grid cell. In the set-based data structure in Prolog, empty grid cells do not take up any storage, nor do they need any CPU time to check that they are empty.

The adaptive grid has avoided pairwise polyline comparison in considering the constituent edge segments individually. Furthermore, it minimizes comparing edge segments one to another by classifying the object space into grid cells. Our motivation is the observation that when two polylines intersect, they do so between two edge segments in the same locality. Hence the adaptive grid method attempts to focus the attention directly on the intersection points by sorting out potentially intersecting edge segments to within those that occupy at least one common grid cell.

3.3 Performance Assessment

In this section, we attempt to analyze the performance of the adaptive grid method. However, due to the same difficulties which we also mentioned in section 1.2.1 - the lack of common consensus on a realistic model for map data, and the substantial mathematical complications of the possible data models - the analysis presented here is based on random edge segments, identically and independently distributed over the object space. Other assumptions will be stated as we proceed.

We consider the problem of computing all intersections between two sets S and T of edge segments. Let N_s and N_t be the number of edge segments in S and T, respectively. We cast the adaptive grid over the scene to sort out groups of potentially intersecting edge segments which share at least one commonly occupied grid cell. The grid is a regular grid so that we can determine the cells occupied by each edge segment, in time linear to the number of (cell, edge) pairs. Let U_s and U_t be the expected number of grid cells each edge of S and T occupies, and W_s and W_t the expected number of edges of S and T in each grid cell. For each of the sets S and T, we count the total number of (cell, edge) pairs. We have

$$\begin{aligned} G \times W_s &= N_s \times U_s \\ G \times W_t &= N_t \times U_t \end{aligned}$$

and jointly,

$$G \times (W_s + W_t) = (N_s \times U_s) + (N_t \times U_t)$$

Then, consider the expected time T for the adaptive grid method to determine all intersections.

$$T = T_1 + T_2$$

where T_1 is the time for casting the grid over all edge segments, and T_2 the time to compute all intersections given the (cell, edge) association. We adopt the following convention, first popularized by Knuth (Knuth 76) to characterize the asymptotic behavior of the timing functions:

$$T = \Theta(\, f(N)\,)$$

denotes that there exist positive constants C_1 and C_2 such that

$$C_1\, f(N) \leq T \leq C_2\, f(N)$$

for all $N > N_0$, that is, in our case, N_s and N_t sufficiently large. T_1 is the time required to form all the (cell, edge) pairs. Hence,

$$T_1 = \Theta(N_s\ U_s + N_t\ U_t)$$

while T_2 is the time required to compare all pairs of edges in $S \times T$ for each cell. Here we assume that the edge segments of each set are independently and identically distributed. We have

$$T_2 = \Theta(G\ W_s\ W_t) = \Theta(\frac{N_s\ U_s\ N_t\ U_t}{G})$$

If we assume that edge segment lengths are relatively uniform, then U_s and U_t can be treated as constants. T_1 is linear to the sum of number of edge segments in S and T:

$$T_1 = \Theta(N_s + N_t)$$

and

$$T_2 = \Theta(\frac{N_s\ N_t}{G})$$

If G remains constant independent of the growth of N_s and N_t, then $T_2 = \Theta(N_s\ N_t)$. We have T_2 given as above because when N_s and N_t grow with G remaining constant, the number of intersections grows linearly to the product $N_s \times N_t$: in any algorithm to determine intersections, we must visit each intersection point at least once. But if there is a reasonable limit to the density of edge segments in an input map, as N (N_s or N_t) increases, W (W_s or W_t), the expected number of edges in each grid cell, will be bounded above asymptotically. The asymptotic growth of G will then be linear to N_s, and to N_t. We have the total timing measure given by:

$$T = \Theta(N_s + N_t) + \Theta(\frac{N_s N_t}{N_s + N_t})$$

The expected time performance is linear to the sum of the size of input data set and that of the output data set. Empirical data in (Franklin 83b) testing up to 50,000 random edge segments verified this result.

This assessment, however, is based on random edge segments identically and independently distributed over the object space. We have also assumed that the edge segments are of relatively uniform lengths, and that there is an upper bound to the density of edge segments. Contrary to the identical and independent distribution assumption, real map data consists of edge segments linked up at their respective end-points into polylines which do not intersect one another in the same map. Our measure for T_2 to be $\Theta(G\ W_s W_t)$ will no longer hold since the product of the expected values is not the same as the average of the products in each grid cell. Accounting for the correlation in map data calls for a more sophisticated mathematical model than what we now have. The question concerning the expected performance of the adaptive grid method for polygon overlay remains open.

The assumption that edge segment lengths have a relatively small standard deviation may be more reasonable for certain classes of polyline data, such as contours, drainage patterns, soil and vegetation boundaries. Artificial caricatures such as property lines, reference grids, or structures in urban planning would show a much larger deviation from uniform length edge segments. Polygon overlay cases often involve the two different classes of these maps. An analysis for the performance may take that into account. Furthermore, although maps can be extremely complex and dense in information, it should be reasonable to assume that there exists a limit to the density of such information in each processing step. An upper bound to the density of edge segments in the map would be a useful restriction in assessing the performance for polygon overlay.

3.4 Grid Cell Size

To set up the adaptive grid, we need to determine an appropriate size for the grid cells. Since the grid is intended to isolate cases of intersecting edge segments, it may seem desirable to use smaller grid cells so that two edge segments would not occupy a common grid cell unless they intersect each other. On the other hand, smaller grid cells result in a grid with more cells, and will take more pre-processing time in distributing the edge segments into the grid. In this section, we will address the issue of determining an appropriate grid cell size.

Let T_1 be the time needed in the pre-processing step, and T_2 the time to examine all pairwise combinations of edge segments in each of the grid cells. Let S be the area of a grid cell, which is inversely proportional to the number of cells in the grid for a given input data set. Then we have,

$$T_1 = \Theta\left(\frac{1}{S} \right) \quad \text{and} \quad T_2 = \Theta(S^2)$$

Since T_1 is monotonically decreasing with S, and T_2 monotonically increasing, the total time $T = T_1 + T_2$ attains minimum at S^* for which $T_1 = T_2$. Figure 3.1 illustrates this.

Optimal grid cell size S^*:

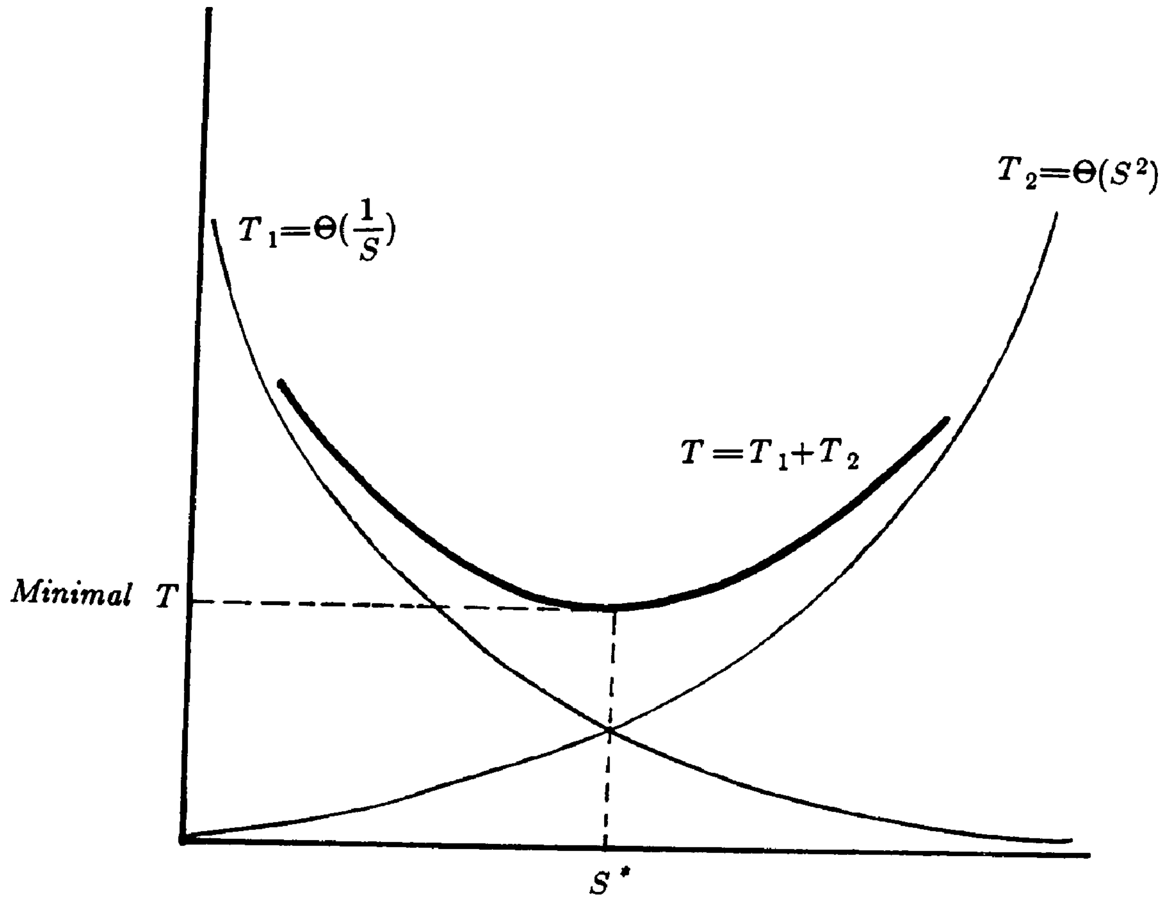


Figure 3.1 Minimum of $T = T_1 + T_2$ occurs at $T_1 = T_2$.

Experimentally, we assume relatively uniform lengths for the edge segments and a low upper bound for the density. We compute the grid cell size separately along the X and Y directions using the projections onto the axes. We choose our grid cell size to be slightly larger than the average length. Let L be the extent of the coordinates along an axis, and e the average length of the projections on the axis. The number of grid cells along that direction is given by G, where

$$G = \left\lfloor \frac{L}{e} \right\rfloor$$

and grid cell size

$$g = \frac{L}{G}.$$

Based on empirical results in (Franklin 83b), the total performance is not sensitive for grid cell sizes close to this chosen value, by as much as a factor of 2. This can be explained intuitively by that the function $T = T(S)$ is relatively flat in the vicinity of the minimum point.

3.5 Local Processing

Local processing as introduced in (White 77) is the concept to deal with large data volume in a computer with little direct access memory. The computer processes a local section at a time so that we do not need to store the entire map in the memory.

The adaptive grid method supports the local processing concept: each grid cell is treated as a local unit in the process. Hence we do not keep the entire grid in memory in this approach. Instead, in the pre-processing stage, we generate a (cell, edge) pair for each grid cell occupied by an edge segment. When finished with the pre-processing stage, we can perform an external sort on the entire file of (cell, edge) pairs, sorting by the cell number. Then we can go through the the file sequentially, examining each non-empty grid cell for intersecting edges. In this approach also, empty grid cells do not use any memory space, and no extra CPU time to check that they are empty. However, the external sorting step does incur additional cost. Here we should note that since we are sorting by the cell numbers, we can do this by distribution sort (Knuth 72) which takes time linear to the total number of (cell, edge) pairs.

The adaptive grid approach is not primarily aimed at saving storage. We may refer to this as an additional advantage in the method.

Chapter 4 RATIONAL ARITHMETIC

Our polygon overlay system uses rational arithmetic to calculate geometric intersections. We are motivated by the problem of numerical inaccuracy in floating point arithmetic. However, here we are not primarily concerned with accuracy but rather with the topological consistency which can be affected by numerical inaccuracy. Section 4.1 will discuss this problem. Rational arithmetic is exact and enables us to circumvent the problem of numerical inaccuracy. Section 4.2 presents the mathematics of rational arithmetic for geometric intersection in polygon overlay. Section 4.3 describes the implementation in our overlay system in Prolog. Section 4.4 addresses the plausible problems involved in conversion between regular floating point numbers and exact rational numbers. Section 4.5 concludes with an assessment on the cost of CPU time in using rational arithmetic.

4.1 An Operational Problem In Polygon Overlay

A problem in implementation of geometry algorithms is numerical inaccuracy. It is, however, not so much a problem of accuracy but that of topological consistency which can be affected by numerical inaccuracy. Coordinates, derived from measurements, are inherently subject to error despite the precision level used in the numerical representation. More important is the problem of geometric computation in which results are subject to discretization errors in floating point arithmetic. Inaccurate results may lead to topological inconsistency; figure 4.1 illustrates the case of an overlay example with a triangle and a square. The vertex P of the

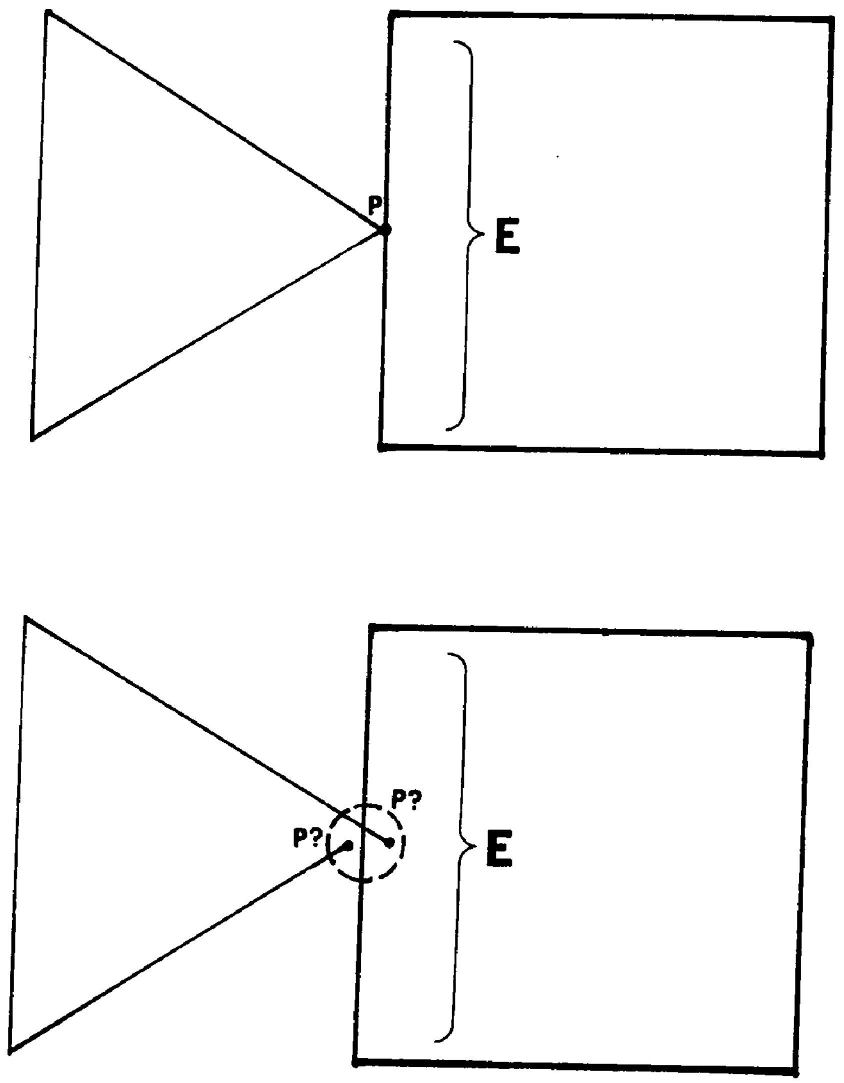


Figure 4.1 An Operational Problem:
topological inconsistency due to numerical inaccuracy

triangle falls on the edge E of the square, but due to numerical inaccuracy, only one of the two adjacent edges of the triangle intersects with E. The problem then is that the output map is topologically inconsistent. Overlay systems are therefore prone to unstability when dealing with situations such as nearly coincidental points, almost touching chains, and other tangent conditions.

Our system uses rational arithmetic which preserves total accuracy. We are then able to circumvent the problems of inaccuracy arising from discretization errors in calculating intersections, and we can guarantee topological consistency in the overlay process.

4.2 Rational Arithmetic For Geometric Intersection

The arithmetic of calculating geometric intersections in polygon overlay is based on that of calculating edge segment intersections. Given the end-points of an edge segment, (x_1,y_1) and (x_2,y_2), the equation of the extended straight line is given by $Ax + By + C = 0$ where

$$\begin{aligned} A &= y_2 - y_1 \\ B &= x_1 - x_2 \\ C &= y_1 x_2 - x_1 y_2 \end{aligned}$$

and all the coefficients are rational. Given two straight lines represented by

$$\begin{cases} A_1 x + B_1 y + C_1 = 0 \\ A_2 x + B_2 y + C_2 = 0 \,. \end{cases}$$

The intersection point (x,y) is given by the solution to the above system of two equations. Given the input coordinates in rational numbers, the intersection point will always have rational coordinates since it is the solution to a linear system with rational coefficients. Hence, we have closure of numerical representation in using rational arithmetic for geometric intersections.

4.3 Rational Arithmetic In Prolog

This section describes our implementation of rational arithmetic in Prolog. The package consists of two parts, one being built on top of the other. The first package, BIG, implements multiple precision integers, and the second, XQ, which builds upon BIG, implements exact rational numbers. Arithmetic using these rational numbers is exact since the operations are based on integer arithmetic with virtually no overflow limit. The two following sections describe the packages, BIG and XQ, respectively, and present the complexity measures of the arithmetic operations based on data precision. The third section remarks on the importance on modular installation in an experimental system. In the fourth section, we present a test run of the BIG and XQ packages, calculating π to an arbitrarily close approximation by a rational number using an infinite series.

4.3.1 BIG - Multiple Precision Integers

A BIG number is an integer with no overflow limit. We represent a BIG number with a list of integers in Prolog. The absolute value of each term in the list is limited to less than a certain maximum term size, say M. The BIG number represented by the list [a_1, a_2, ..., a_n] is given by

$$a_1 + a_2 M + a_3 M^2 + \cdots + a_n M^{n-1}$$

Since Prolog does not impose a limit to the number of terms in a list, the BIG number has virtually no overflow limit. A legitimate BIG number should have leading zeroes trimmed. Zero is therefore represented as [ ], the empty list. A negative BIG number has the most significant term negative and all other terms in the list non-negative. Given this standard, we have both uniqueness and completeness in our representation. Figure 4.2 shows some examples, for $M = 100$.

The maximum term size, M, is chosen such that

$$M^2 \leq L$$

where L is the regular integer overflow limit. This is to prevent integer overflow in any one term when evaluating an arithmetic expression. In the following we will describe the arithmetic operations implemented: addition, subtraction, multiplication, division and remainder (modulo arithmetic). Arithmetic comparison is included. Given that the BIG integer operands N_1 and N_2 having n_1 and n_2 terms each, respectively, we also present the worst case time complexity measures. We will use Knuth's notation:

0	represented as	[ ]
1	represented as	[1]
100	represented as	[0,1]
12345	represented as	[45,23,1]
-10023	represented as	[-23,0,-1]
-1	represented as	[-1]

Figure 4.2 Examples of BIG integers ($M = 100$)

$$T = O(\ f(N)\ )$$

denotes that there exist positive constants C and N_0 such that $T \leq Cf(N)$ for $N > N_0$ (Knuth 76).

Addition/Subtraction. This can be done by taking the corresponding terms together for the operation, so that

$$\sum_{i=1}^{n_1} a_i M^{i-1} \pm \sum_{j=1}^{n_2} b_j M^{j-1} = \sum_{k=1}^{max(n_1,n_2)} c_k M^{k-1}$$

$$\begin{aligned} &\textit{where} \quad c_k = a_k \pm b_k \\ &\textit{and} \quad a_k = 0 \ \text{ for } \ k > n_1; \quad b_k = 0 \ \text{ for } \ k > n_2. \end{aligned}$$

The result is then a BIG integer with $max(n_1,n_2)$ terms to be validated to a legitimate representation. Since each term is visited once, the time complexity is $O(max(n_1,n_2))$.

Multiplication. This is done by the common "shift-and-add." Each term in one operand is multiplied to every term of the other operand, shifting according to the significance rank of each term and adding the results together.

$$\begin{aligned} (\sum_{i=1}^{n_1} a_i M^{i-1}) \times (\sum_{j=1}^{n_2} b_j M^{j-1}) &= \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} a_i b_j M^{i+j-2} \\ &= \sum_{k=1}^{n_1+n_2-1} c_k M^{k-1} \end{aligned}$$

$$\textit{where} \quad c_k = \sum_{r=1}^{k} a_r \, b_{k-r+1}$$

Since we have to consider every pair of terms from the two operands, the complexity measure is $O(n_1\ n_2)$.

Division. Division gives both quotient and remainder. There are two stages in our version of BIG integer division: long division and trial division. Let n_1 be the number of terms in the dividend and n_2 the number of terms in the divisor. First we apply long division: the more significant portion of n_2 terms from the dividend is considered for trial division by the divisor. This division yields a one term quotient. The next term from the dividend is then prepended (least significant term is at the beginning of the list) to the remainder for another trial division to get the next term of the quotient. The process is repeated $n_1 - n_2$ times until all the remaining $n_1 - n_2$ terms of the dividend are considered. Figure 4.3 illustrates the long division process. We apply one trial division each time to obtain each term in the quotient, the most significant one first. In trial division, we recursively test subtracting the divisor from the dividend, doubling the divisor in each step. With the recursion expanded, we can express the quotient Q in the following form,

$$Q = \sum_{i=1}^{\lceil \log_2(M) \rceil} q_i\, 2^{i-1}$$

where $q_i = 1$ *or* 0, depending on whether or not the testing subtraction is successful for the corresponding step in the recursion. M is the maximum term size, and $\lceil \log_2(M) \rceil$ determines the maximum depth of recursion necessary in the worst case. Trial division is therefore of complexity $O(n_2)$, and coupled with the long division, the complexity measure of our division algorithm is $O(n_2(n_1 - n_2))$.

$$[\ 12,\ 34,\ 56\ ] \div [\ 78,\ 9\ ] = [\ 76,\ 5\ ] \ldots [\ 84\ ]$$

$$\begin{array}{r|lll}
\multicolumn{1}{r}{} & & [\ 76, & 5\ ] \\ \cline{2-4}
[\ 78,\quad 9\ ] & [\ 12, & 34, & 56\ ] \\
 & & [\ 90, & 48\ ] \\ \cline{2-4}
 & [\ 12, & 44, & 7\ ] \\
 & [\ 28, & 43, & 7\ ] \\ \cline{2-4}
 & [\ 84\ ] & &
\end{array}$$

Figure 4.3 Long division of BIG integers

Arithmetic Comparison. To compare two BIG numbers, we first compare the sign, and then the number of terms in the representation. If both are equal, we then compare each corresponding term in the list, starting with the most significant term first. Worst case occurs when two operands are equal, or they differ only at the least significant term; the complexity is $O(n_1+n_2)$.

4.3.2 XQ - Exact Rational Numbers

XQ numbers are built on BIG integers. An XQ number is a fraction in which both the numerator and the denominator are BIG integers. We represent an XQ number by a division expression with BIG integer operands. For uniqueness of representation, a legitimate XQ number has the numerator and the denominator reduced so that their greatest common divisor, $gcd = 1$. Negative XQ numbers have negative numerators, and the denominators are always positive. To provide compatibility with BIG integers, we omit the denominator when it is unity. In other words, a BIG integer is a legal XQ number with denominator equal to [1]; figure 4.4 illustrates some examples of XQ numbers. An XQ arithmetic expression can then have BIG integers mixed together. Given the operands as $\frac{N_1}{D_1}$ and $\frac{N_2}{D_2}$, we will describe the arithmetic operations, gcd computation, and arithmetic comparisons in the following. We will also give the worst case time complexity, given that the numerators N_1 and N_2 each has n_1 and n_2 terms, respectively, and that the denominators D_1 and D_2, d_1 and d_2 terms.

0	represented as	[ ]
33	represented as	[33]
11/310	represented as	[11] / [10,3]
48/100	represented as	[12] / [25]
-321/170	represented as	[-21,-3] / [70,1]
-1	represented as	[-1]

Figure 4.4 Examples of XQ - multiple precision rational numbers

Addition/Subtraction. We take the product of the denominators to form the common denominator. Hence we have

$$\frac{N_1}{D_1} \pm \frac{N_2}{D_2} = \frac{N_1\,D_2 \pm N_2\,D_1}{D_1\,D_2}.$$

For the asymptotic complexity, we need to consider only the BIG integer multiplications, and it is given by $O(d_1 d_2 + n_1 d_2 + n_2 d_1)$.

Multiplication/Division. Multiplication and division are simpler than addition and subtraction for XQ numbers. They are given by

$$\frac{N_1}{D_1} \times \frac{N_2}{D_2} = \frac{N_1 N_2}{D_1 D_2}, \quad \text{and} \quad \frac{N_1}{D_1} \div \frac{N_2}{D_2} = \frac{N_1 D_2}{N_2 D_1}.$$

Division needs some extra work when N_2 is negative. Considering the BIG integer multiplications, the complexity measures are given by $O(n_1 n_2 + d_1 d_2)$ and $O(n_1 d_2 + n_2 d_1)$, for multiplication and division, respectively.

gcd Computation. To reduce the result from an arithmetic expression, we need to divide both numerator and denominator by their *gcd*. We compute *gcd* by repeated mutual division: given numerator N and denominator D, we calculate $N_1 = N \; mod \; D$. N_1 is non-zero if D does not divide N. Then, we calculate $D_1 = D \; mod \; N_1$, and so on until we get 0 or 1. If we get 0, the last divisor is the *gcd*. If we get 1, N and D are mutually prime. The asymptotic complexity is dominated by that of the first integer division, which is $O(d(n-d))$ if $n>d$, or $O(n(d-n))$ if $d>n$, where n and d are the number of terms in the numerator and denominator, respectively.

Arithmetic Comparison. To compare two XQ numbers $\frac{N_1}{D_1}$ and $\frac{N_2}{D_2}$, we only need to compare the two BIG integers N_1D_2 and N_2D_1. Hence, the complexity is given by $O(n_1d_2+n_2d_1)$.

4.3.3 Modularity In An Experimental System

Rational arithmetic as installed by BIG and XQ in Prolog demonstrates high modularity. Modularity here means that we are able to isolate in implementation the algorithm and the arithmetic domain. This is important in an experimental system, such as our polygon overlay system. We investigated the use of rational arithmetic for chain intersections. Our system provides a vehicle for further experimentation. Since Prolog allows operator overloading, the syntax for arithmetic expression remains unchanged. This provides for the modular installation of other arithmetic domains in both existing and new programs. On the other hand, the BIG and XQ packages can also be installed into other systems for experimentation. We note this here since this is an important design feature of our system.

4.3.4 A Test for BIG and XQ: Computing π

To test BIG and XQ, and in part also to demonstrate the modularity, we present here a short program to calculate π to an arbitrarily close approximation by a rational number. We use the following infinite series.

$$\pi = 2 \times \frac{2}{1} \times \frac{2}{3} \times \frac{4}{3} \times \frac{4}{5} \times \frac{6}{5} \times \frac{6}{7} \times \cdots$$

This series converges very slowly, but fast convergence is not the point of the test run. BIG and XQ are installed using "*are*" for evaluation of BIG integers and "*isx*" for XQ, the exact rational numbers. The following is the Prolog code to compute π; each backtracking step computes one more term in the series.

```
pi([ ],[2]).          %preset pi=2

step :-
    pi(R,P),
    R1 are R+ [1],
    R2 are R1 mod [2],
    P1 isx P*((R1+R2)/(R1-R2+ [1])),
    retract(pi(R,P)),
    assert(pi(R1,P1)),!.

go :- repeat, step, fail.
```

The result after 100 steps of iteration is

$$\frac{50216813883093446110686315385661331328818843555712276103168}{16063834434771661191161473607166722989851247353354683757549}.$$

The source listings of BIG and XQ are appended to this thesis.

4.4 Input/Output Conversion Problems

Since rational arithmetic preserves total accuracy in numerical values, we are able to guarantee topological consistency in the polygon overlay process. This is based on having all the coordinates in rational numbers. In practice, input maps with coordinates in floating point numbers must first go through a conversion process. We convert the coordinates into rational numbers before passing the maps on to the overlay process. After the overlay process, we then convert the output map back to one with floating point coordinates. It may seem plausible that the conversion process can introduce inconsistency into the topology since it also suffers from numerical inaccuracy. In this section we will address the problems of converting coordinates from floating point to rational, and vice versa.

In input conversion, since rational numbers can be arbitrarily close to any numerical value, and floating point numbers have finite precision, we *can* represent the given coordinates in rational numbers without loss in accuracy. In practice, we often impose certain limits to the number of significant digits in the numerical input values, and the conversion process may then result in inconsistent topology. We however note that the problem is then not due to the conversion imposed by the use of rational arithmetic, but by the change in resolution beyond the minimum required to maintain the map topology.

In output conversion, unfortunately, with floating point numbers we cannot preserve totally the accuracy inherently in rational numbers. However, it is possible to determine the minimal resolution in the coordinates necessary to maintain the topology. This then determines the precision level needed in the floating point numbers for output conversion. We should note that the information density of the output map is the total of those of the input maps. Precision level of the input data may not be sufficient for the output data. In that case, the map topology is unstable because of insufficient precision in the source data; our overlay system maintains the topology and the problem is identified in output conversion. In section 8.3.1 under Further Considerations, we discuss a strategy to remove sliver polygons in the output map. In the process, we may eliminate nearly coincidental features and help to alleviate the problem of unstable topology in output conversion. Furthermore, we refer to (Saalfeld 87) on the recent report in the determination of minimal resolution in the coordinates to maintain map topology. Work in that direction is not within the scope of this thesis.

4.5 On The Cost Of Rational Arithmetic

Rational arithmetic provides total accuracy. But how much CPU time does it cost to achieve the accuracy? The section here will try to answer this question.

There are two aspects to the cost of CPU time in arithmetic: precision level, and data volume. In section 4.3, we described our implementation of rational arithmetic, and presented the complexity measures for each operation based on the number of significant terms in the operands. This is a measure of data precision level. For the polygon overlay system, we will then have to trace out the route of arithmetic operations to measure the cost. Given the end-points of two edge segments: (x_{11},y_{11}), (x_{12},y_{12}) for one edge segment and (x_{21},y_{21}),

(x_{22}, y_{22}) for the other. The equations of the two straight lines through the edge segments are $A_1x + B_1y + C_1 = 0$ and $A_2x + B_2y + C_2 = 0$, respectively, where

$$\begin{cases} A_1 = y_{12} - y_{11} \\ B_1 = x_{11} - x_{12} \\ C_1 = y_{11}x_{12} - x_{11}y_{12} \end{cases} \quad \textit{and} \quad \begin{cases} A_2 = y_{22} - y_{21} \\ B_2 = x_{21} - x_{22} \\ C_2 = y_{21}x_{22} - x_{21}y_{22} \end{cases}$$

The intersection point (x,y) is then given by

$$x = \frac{B_1C_2 - B_2C_1}{A_1B_2 - A_2B_1} \quad \textit{and} \quad y = \frac{A_2C_1 - A_1C_2}{A_1B_2 - A_2B_1}$$

Assume that the input numerical values (i.e., x_{11}, y_{11}, x_{12}, y_{12}, ... and so on) have L significant terms. Then A_1, B_1, A_2, B_2 will each have L terms and C_1, C_2 will have $2L$ terms, because of the multiplication in the expression for C_1 and C_2. For the asymptotic complexity measure, the multiplication dominates, giving $O(L^2)$. In the expression for the intersection point (x,y), note that x and y each is a fraction of numerator having $3L$ terms and denominator $2L$ terms, and the complexity measure for the evaluation of the expressions involved is $O(L^2)$. Finally the *gcd* computation for two BIG integers having $3L$ terms and $2L$ terms has complexity $O(L^2)$. Consider now the whole process of calculating the intersection point: the highest order in complexity measure is $O(L^2)$. The asymptotic growth of CPU cost with the requirement in precision level is no faster than quadratic order.

Given the operands and their precision level fixed, evaluation of an arithmetic expression is a constant time operation. Then in relation to data volume, rational arithmetic does not affect the asymptotic behavior of the algorithm. The CPU time required to process any data volume will only change by a multiplicative factor, with the use of rational arithmetic. Hence, rational arithmetic will not impose a need for CPU time more than linear to the size of the data volume.

Chapter 5 SPECIAL CASES AND STABILITY

Rational arithmetic gives exact results in calculation. We can therefore properly identify the special cases of intersection such as touching and partially overlapping edge segments. We check absolute equality instead of setting a tolerance to identify cases of an end-point lying exactly on another point or another edge segment, as well as cases of intersection between two collinear edge segments.

Algorithm designers for computational geometry often put aside special cases because of the rarity of their occurrence. It is also argued on the basis that the probability of occurrence for the special cases such as two exactly coincidental points, or two collinear lines vanishes in statistics. While the approach renders a simplified picture for easier understanding of a problem, it also leaves the algorithm specification incomplete for implementation. Moreover, special cases do occur partly due to the inherent limited resolution in the computer to represent numerical values, and also partly due to the reason that we are not dealing with random data. For polygon overlay, data dealing with artificial caricatures such as in urban planning and parcel boundaries, very often carries numerous cases of data coincidence by design.

Section 5.1 will describe the special cases and how these cases are reduced to cases of edge segment intersections. In section 5.2, we develop an algorithm for edge segment intersection complete with special cases handling. In section 5.3, we discuss the stability of the overlay system thus achieved, and the cost we have to pay.

5.1 Special Cases In Polygon Overlay

Special cases are ''special'' because they call for special handling when processing on the computer. Special cases occur not just because of the data, but also because of the data structure. This means that certain cases need special handling because of our view we impose on these cases by the data structure. A map is a spatial data structure consisting of nodes, chains, and polygons. In polygon overlay, we deal with polygon intersections. However, we will show that we can limit our special cases handling to only those cases of edge segment intersection. In other words, when the special cases in edge segment intersection are properly handled, cases of coincidental nodes or chains do not need special handling. In this section, we shall classify all the cases of intersection between two edge segments, and in the next section, develop an algorithm to properly identify and resolve them.

5.1.1 Coincidental Nodes And Vertices

Since a node or a vertex must be an end-point of an edge segment, coincidental nodes or vertices are simply cases of edge segments touching at the end-points. There is therefore no need to sort out coincidental nodes and vertices; they are identified when processing edge segment intersections. Furthermore, the edge segment intersection procedure takes advantage of the adaptive grid strategy. The end-point in common becomes a new node in the output map since it is an intersection point between two touching edge segments.

5.1.2 Partially And Totally Overlapping Chains

In the chain intersection stage of our polygon overlay system, we use the adaptive grid method and we deal with the constituent edge segments individually disregard of the chains. The approach is based on the observation that when two chains intersect, they do so in between two constituent edge segments in the same locality. The basis is the same in dealing with partially or totally overlapping chains: overlapping chains must comprise edge segments which overlap. The adaptive grid sorts them out in the same way. When overlapping edge segments are properly handled, overlapping chains do not impose a special case for special handling.

5.1.3 Cases Between Two Edge Segments

Figure 5.1 shows different cases of intersection between two edge segments. Since each edge segment is associated with a specified direction (along the chain), we distinguish as different cases when the intersecting segments are in a different orientation. There are four groups of these cases: when two intersecting segments cross over each other, we have the well understood popular case of edge segment intersection; the rest are *special* cases. When two edge segments are touching each other, the intersection may involve an end-point and an interior point, or two coincidental end-points. Partial overlap cases involve two intersection points, between which the two intersecting edge segments overlap. Exact overlap cases may involve two segments in the same or opposite direction, but they are not treated as ''intersecting'' since the pair is to be replaced by

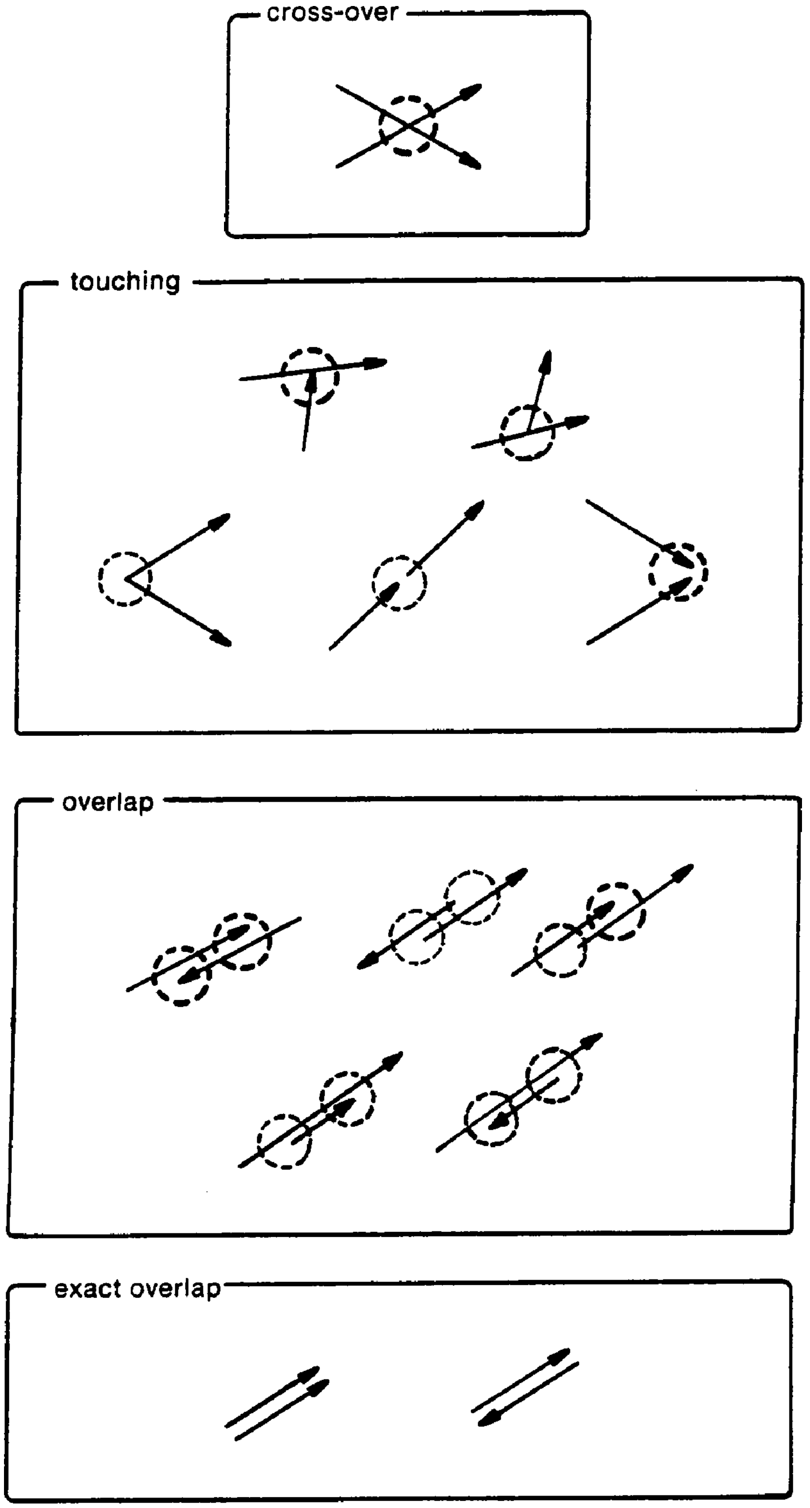


Figure 5.1 Cases of Edge Segment Intersection

a single edge segment.

5.2 A Complete Algorithm For Edge Segment Intersection

In this section, we will develop an algorithm for edge segment intersection, complete with special cases handling. We will re-organize the groups of intersection cases presented in the last section in order to describe a more pragmatic approach to distinguish these cases. We will begin with a discussion on the basis for the algorithm.

A straight line on the 2D plane is characterized by the equation $Ax + By + C = 0$. The straight line divides the plane into three sets, given by

$$\left\{ (x,y) \mid Ax + By + C = 0 \right\},$$

$$\left\{ (x,y) \mid Ax + By + C > 0 \right\}, \text{ and } \left\{ (x,y) \mid Ax + By + C < 0 \right\}.$$

Given the coordinates (x_0, y_0) of a point on the plane, let

$$d = Ax_0 + By_0 + C.$$

The sign of d characterizes the topological relationship between the point and the line, namely, that which side of the line the point lies on, or that it lies on the line. This is going to play an important role in our algorithm. Now we re-organize the different cases of edge segment intersection into the following:

Case 1. Intersecting edge segments are collinear. If we disregard the total overlap cases, two partially overlapping edge segments have two intersection points: an intersection point is an end-point of one segment lying on the other segment. Here we also include the case when two collinear edge segments touching each other at an end-point, which is the only intersection point.

Case 2. One segment lies entirely on one side of the other segment. This case can be further divided into two: the edge segments meeting at the end-points, and the end-point of one touching the interior of the other. In either case, there is exactly one intersection point.

Case 3. Intersecting edge segments cross over each other.

Our algorithm approaches the intersection problem by sorting out these cases. We check these cases out in reverse order so that most non-intersecting cases and the cross-over intersection cases will be identified first without checking into other special cases.

Let the two edge segments be E_1 and E_2, and their extended lines be L_1 and L_2. The end-points of E_1 are P_{11} and P_{12}; those of E_2 are P_{21} and P_{22}. The following outlines our algorithm, checking the special cases in the order presented:

- If P_{11} and P_{12} lie entirely on one side of L_2, or if P_{21} and P_{22} lie entirely the same side of L_1, E_1 and E_2 do not intersect.

- If P_{11} and P_{12} lie on opposite sides of L_2, and P_{21} and P_{22} also lie on the opposite sides of L_1, E_1 and E_2 intersect, crossing over each other.

- If L_1 and L_2 are not collinear, E_1 and E_2 intersect at an end-point of one of the edge segments, and the point touches the other edge segment.

- If L_1 and L_2 are collinear, E_1 and E_2 may or may not overlap. If they do, we consider two intersection points: an end-point of one edge segment lying on the other edge segment is an intersection point. The portion between the two intersection points is the overlap portion. The overlap portion shrinks to a single point when E_1 and E_2 just touch each other at an end-point.

Now we formulate further programming details to identify these cases. Refer back to our characterization of the topology between a point $P = (x_o, y_o)$ and a line $L: Ax+By+C = 0$. We have $d = d(P,L)$ given by

$$d = Ax_o + By_o + C.$$

The sign of d indicates which side of L P lies on; P lies on L if $d=0$. For the four end-points, we let

$$\begin{cases} d_{11} = d(P_{11}, L_2) \\ d_{12} = d(P_{12}, L_2), \end{cases} \quad and \quad \begin{cases} d_{21} = d(P_{21}, L_1) \\ d_{22} = d(P_{22}, L_1). \end{cases}$$

And we let

$$d_1 = d_{11} \times d_{12}, \quad \textit{and} \quad d_2 = d_{21} \times d_{22}.$$

The sign of d_1 indicates whether P_{11} and P_{12} are on the same or opposite sides of L_2, and similarly d_2 for P_{12} and P_{22}. Our algorithm consists of the following series of testing to identify all the cases; these are tabulated below, in the prescribed order:

If $(d_1>0$ or $d_2>0)$,	E_1 and E_2 do not intersect.
If $(d_1<0$ and $d_2<0)$,	E_1 and E_2 cross over each other.
If $(d_{11}=0$ and $d_2<0)$,	P_{11} touches E_2.
If $(d_{12}=0$ and $d_2<0)$,	P_{12} touches E_2.
If $(d_1<0$ and $d_{21}=0)$,	P_{21} touches E_1.
If $(d_1<0$ and $d_{22}=0)$,	P_{22} touches E_1.
If $(d_{11}\neq 0$ and $d_{21}=0)$,	P_{12} touches P_{21}.
If $(d_{11}\neq 0$ and $d_{22}=0)$,	P_{12} touches P_{22}.
If $(d_{12}\neq 0$ and $d_{21}=0)$,	P_{11} touches P_{21}.
If $(d_{12}\neq 0$ and $d_{22}=0)$,	P_{11} touches P_{22}.

These account for all the cases when E_1 and E_2 are not collinear. If a case passes through these tests, we can at this point ascertain that $d_{11}=d_{12}=d_{21}=d_{22}=0$. In order words, E_1 and E_2 are collinear. E_1 and E_2 may or may not overlap; we examine the end-points P_{11}, P_{12}, P_{21}, and P_{22} to determine whether or not each lies on the other edge segment. If none is reported, E_1 and E_2 do not overlap. If two are reported, E_1 and E_2 overlap between these two points. These two points may coincide; in that case E_1 and E_2 touch each other at their end-points. If all four points are reported, E_1 and E_2 overlap exactly over each other. The pair should be replaced by a single edge segment.

5.3 Stability And What It Costs

We guarantee stability in the polygon overlay process based on two aspects achieved in our system in Prolog. On the one hand, rational arithmetic provides for total accuracy in numerical computation. Unstable map topology does not suffer from problems of discretization errors in the process. Consistency is therefore preserved. On the other hand, we have also taken advantage of exact numerical results to identify all special cases of intersection for proper handling. Neither tangent conditions nor nearly coincidental features can cause failure in the overlay process. Thus we have achieved stability. But how much does it cost, in terms of our resources, to achieve stability in these aspects? In retrospective, we will try to answer this question here.

In section 4.5, we have considered the cost of rational arithmetic. The CPU time for rational arithmetic does not grow asymptotically more than linear to the size of the data volume. In this chapter, although the edge segment intersection problem may appear to be simple, we have gone at length to deal with all the special cases involved. In fact, a substantial portion of the code is there for the handling of these special cases. Fortunately, this does not mean that special cases handling will impose a substantial cost in CPU time. Here we must note that the special cases do occur, but they do not occur frequently. Worst case performance analysis would be deceptive, and should not take into account handling for the special cases. While it is difficult to assess the average case performance, we offer the following argument: identifying special cases does not necessarily cost extra CPU time, since it can be done in the process of calculating output data. In the case of intersecting edge segments, testing for intersection at the same time produces the information necessary to identify touching and collinear cases. In the way we have prioritized checking for ''regular'' cases first, special cases handling will not affect the average case performance unless the ''special'' cases dominate in processing.

Chapter 6 USING PROLOG FOR GEOMETRY

Prolog represents a radically different approach toward programming. Using Prolog for our polygon overlay system is a venture motivated by the quest for better programming tools to deal with geometry on the computer. In this section, we will describe our experience in using Prolog, focused on a few issues illustrated in the polygon overlay problem. We have formulated certain paradigms of programming which appears to be useful in general. Along with the discussion, we will also bring up certain pragmatic issues involved with programming in Prolog.

6.1 A Logic Programming Example

Unlike conventional programming languages, Prolog is a declarative language. This means, at least in theory, that a Prolog program is not a prescribed set of instructions to solve a particular problem, but is instead a description of the objects and their relationships involved in the problem which provides sufficient information to solve the problem. Such an approach has been commonly known as "logic programming." Consider the Prolog program to append one list to another.

```
append([ ],L,L).
append([H |L1],L2,[H |L3]) :- append(L1,L2,L3).
```

Let us call the arguments of **append** by name: we shall call the list to be appended *list#*1, and we append *list#*2 to *list#*1, to form the result which we shall call *list#*3. The first rule above says that if *list#*1 is empty, what we get for *list#*3 is just the same as *list#*2. The second rule gives a recursive definition of what it means to append to a list, if it is not empty:

- The first element of *list#3* is the first element of *list#1*.
- The tail of *list#3* consists of the tail of *list#1*, with *list#2* appended to it.

Such is the description in Prolog to append a list, and it provides sufficient information to perform a number of functions, appending to a list included. We show the following examples:

append([a,b],[c,d,e],*LIST*).

This gives *LIST*=[a,b,c,d,e] for a solution, an example of appending to a list. But we can also use **append** to generate successive partitions of a given list.

append(*L1*,*L2*,[a,b,c,d,e]).

L1 and *L2* will give the following 6 possible solutions:

L1	=	[ ],	*L2*	=	[a,b,c,d,e].
L1	=	[a],	*L2*	=	[b,c,d,e].
L1	=	[a,b],	*L2*	=	[c,d,e].
L1	=	[a,b,c],	*L2*	=	[d,e].
L1	=	[a,b,c,d],	*L2*	=	[e].
L1	=	[a,b,c,d,e],	*L2*	=	[ ].

We can also obtain the first and last elements of a list, such as getting the beginning and ending nodes in the polyline LIST of a chain,

append([*N1* |_],[*N2*],LIST).

And examine in succession the vertices of a polyline LIST,

append(_,[V | _],$LIST$).

In spite of the power of logic programming demonstrated by the predicate **append** here, we must note that in practice we still have to be somewhat imperative. An example is the definition of **append**. The two rules for append must be maintained in that order so that linear search for resolution would work properly in Prolog.

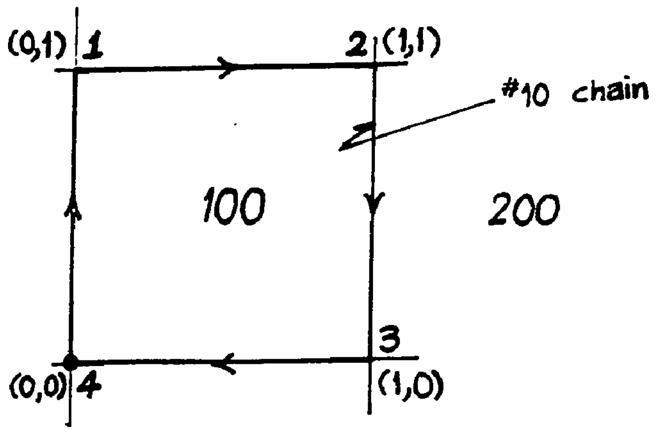
6.2 Data Structuring

Logic programs can be considered an extension to the relational data model. The basic operations of relational algebra can be easily expressed in logic programming. Using a relational model, we are able to simplify much the design of data structures necessary in our polygon overlay system. We represent geometric entities and their relationships as Prolog facts, such as in

v($V\#$,[x,y]).	% for a vertex
e($C\#$,$E\#$,[V_1, V_2]).	% for an edge segment
c($C\#$,N_1,N_2,P_1,P_2).	% for a chain, (or polyline)
p($P\#$,[B_1,B_2,...]).	% for a polygon,
	% where B_i = th(C_i) or ht(C_i)
	% ...of directed chain C_i.

Figure 6.1 shows the data structure for a unit square. The same is also demonstrated in interim data structures generated in the process of polygon overlay, for example

i($N\#$,$E\#$,Angle).	% for edge-to-node incident angle



vertices	*v*(1,[0,1]).
	v(2,[1,1]).
	v(3,[1,0]).
node	*n*(4,[0,0],[h(10),t(10)]).
chain	*c*(10,4,4,200,100).
edges	*e*(10,1,4,1).
	e(10,2,1,2).
	e(10,3,2,3).
	e(10,4,3,4).
polygons	*p*(100,[*ht*(10)]).
	p(200,[*th*(10)]).

Figure 6.1 Data Structures for the Unit Square

A pragmatic issue is involved with efficiency in retrieving information from a large set of facts. This may be different in different implementations of Prolog. But most Prolog systems use hash coding on the functor names to access the set of facts under the same functor name and then search the set in linear search. We exploit the hashing of functor names to evade unnecessary linear search when the key index is fully instantiated. Our scheme is to generate a functor name using the key. For example, to retrieve vertex with identifier =23, we use the following functor name,

```
v23([x,y]).                        %vertex id =23
```

On the other hand, to facilitate for iterative search through all vertices, we maintain the set of identifiers in a list for linear search.

```
v(id#).                            %list of vertex identifiers
```

Although many Prolog versions have different built-in facilities for database applications, our scheme, based on relatively standardized specification, would achieve better transportability in practice.

6.3 Set-Based Operations

Our polygon overlay system illustrates decomposing a complicated process into simple steps; often each step applies a certain operation to an entire set of geometric entities. For example, in the adaptive grid method, each grid cell is a bucket to keep a set of edge segments. We consider the entire set of edge segments and distribute them to each grid cell it occupies.

Clearly this is easy in Prolog. However, we must also be careful when simultaneously iterating and updating the same database. More specifically, **assert** and **retract** may affect an iterating sequence differently in different Prolog implementations.

We opt to avoid this in many cases. In the case of polygon boundary formation, we connect LINKAGE facts until the set is exhausted. The operation involves **retract** and **assert** on the facts while iterating through. We use a **repeated_retract** predicate to retrieve each fact until no more facts exist:

```
repeated_retract( LINKAGE ) :-
    repeat,
    ( not( call( LINKAGE ) ),!,fail;
      retract( LINKAGE ) ).
```

Obviously, one can use the variable LINKAGE for any other set of facts.

6.4 Pattern Matching Geometric/Topological Properties

Another paradigm uses pattern matching to propagate properties. For example, when linking chains to form the polygon boundaries, we first form the corners of each polygon. The corners can be considered fragments of the polygon boundaries. Whenever two fragments exist such that they can be connected, we retract both fragments and assert the new connected fragment. When no such fragments exist, we have all the polygon boundaries.

Given the following corner fragments of a polygon, as illustrated in figure 6.2, each fragment being a **linkage** fact.

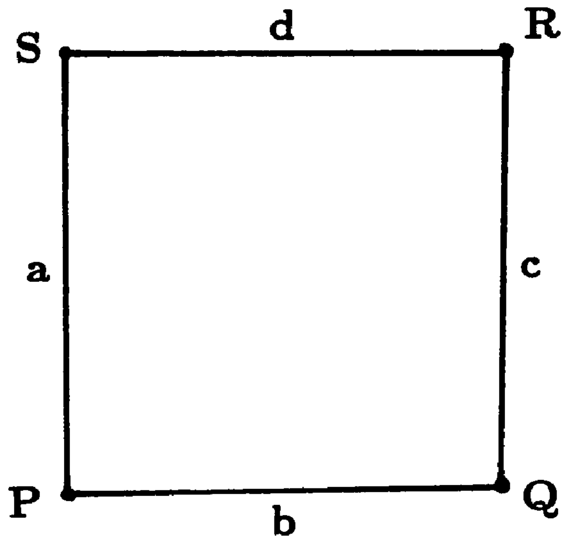
```
linkage(a,b,[a,b]).            % corner of a-P-b
linkage(b,c,[b,c]).            % corner of b-Q-c
linkage(c,d,[c,d]).            % corner of c-R-d
linkage(d,a,[d,a]).            % corner of d-S-a
```

To form and "recognize" the polygon, we define **connect** for pattern matching linkage properties, and we use **repeated_retract** from section 6.3 to perform set-based operation on the set of fragments.

```
connect(A,A,[A |L]) :-
    !,assert(new_polygon(L)).     % complete cycle is polygon.

connect(A,B,L1) :-
    retract(linkage(B,C,[B |L2])),
    append(L1,L2,L3),
    assert(linkage(A,C,L3)).      % fragments matched.

match_linkages :-
    repeated_retract(linkage(A,B,L)),
    connect(A,B,L),
    fail.                         % match until linkages exhausted.

match_linkages.
```

The predicate **match_linkages** performs a set operation on the set of **linkage**'s; **connect** matches fragments of polygon corners until no more fragments can be found. Given the set of **linkage** facts as illustrated in figure 6.2, we will have one polygon:

```
new_polygon([a,b,c,d]).
```



linkage(**a**,**b**,[a,b]).

linkage(**b**,**c**,[b,c]).

linkage(**c**,**d**,[c,d]).

linkage(**d**,**a**,[d,a]).

Figure 6.2 Matching fragments of a polygon.

Note that the cyclic order in the polygon boundary has been omitted for clarity, and ease of presentation. In linking chains for the overlay output map, chains have their specified directions.

6.5 Unification and Graph Connectivity

Given a set of objects, if there exists an equivalence relation, we may use unification of variables to classify objects under each equivalence class. Graph connectivity is an equivalence relation between the nodes, and so is polygon boundary adjacency between the polygons. When resolving containment relationships between polygon boundaries to determine the overlay containment polygon, we need to determine only for one polygon in each group of adjacent polygons. While we are searching recursively through the neighbors of each polygon, we assign free variable to each neighbor for the containment polygon, and have them unified in each group of adjacent polygons. Once the containment polygon is resolved, the uninstantiated variables in the entire group are resolved.

As an example, consider the following list of nodes in the graph illustrated in figure 6.3; polygon adjacency is shown in dotted lines as the dual graph. We associate a free variable as an attribute to each node.

```
[ [a,_],[b,_],[c,_],[d,_],[e,_],[f,_] ]
```

Process the set of edges and for each edge, unify the free variables associated with the end-points. The list becomes:

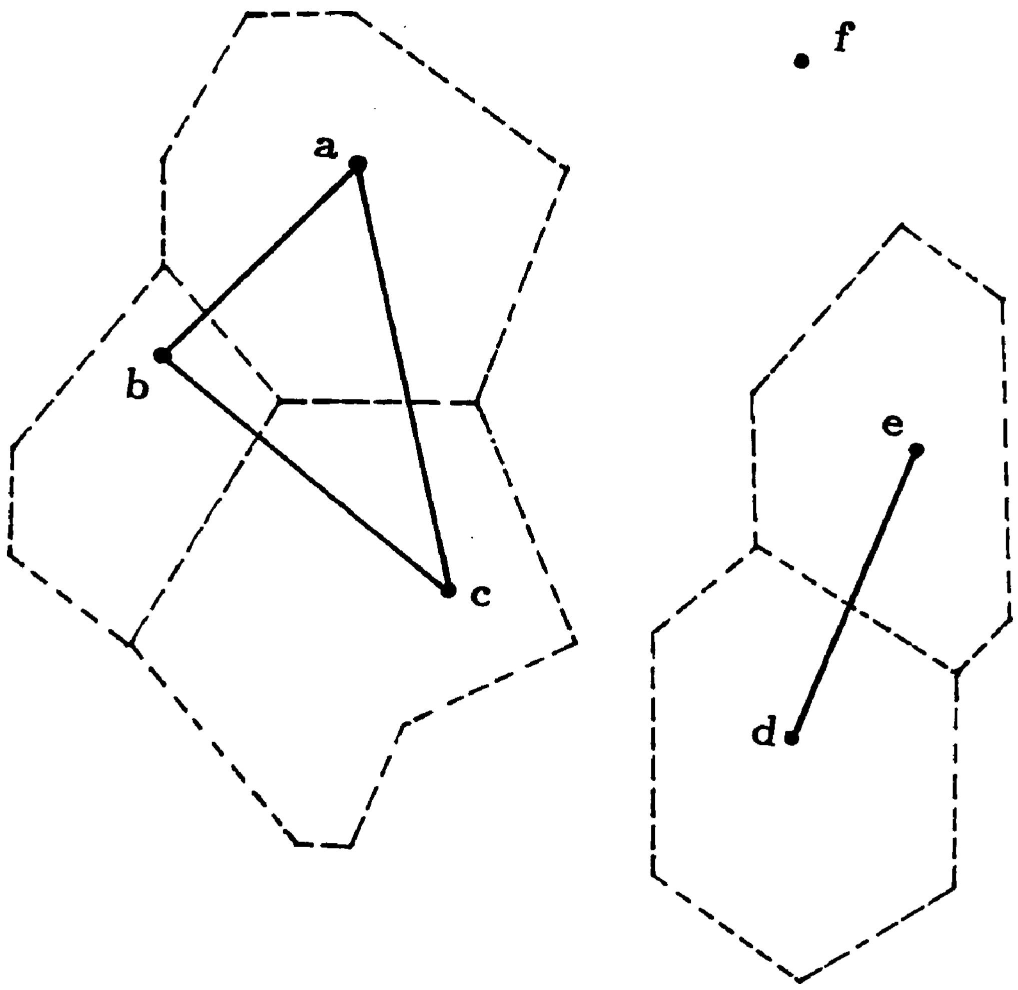


Figure 6.3 Graph connectivity / Polygon adjacency

[[**a**,_1],[**b**,_1],[**c**,_1],[**d**,_2],[**e**,_2],[**f**,_3]]

Binding a name to each uninstantiated variable will then identify the connected components of the graph, such as in the following.

[[**a**,*one*],[**b**,*one*],[**c**,*one*],[**d**,*two*],[**e**,*two*],[**f**,*three*]]

We then have each class of connected nodes identified and properly named.

6.6 Quick Prototype versus Production System

A final remark on using Prolog is, on the one hand, demonstrated in the development of a quick prototype system. The general high level nature of Prolog fosters an intuitive programming environment for experimentation. Our polygon overlay system in Prolog exemplifies such an experimental development effort: we have illustrated the advantages offered in Prolog in this section. On the other hand, however, the performance of our system in Prolog is far from that expected of a production system. Will Prolog be suitable for a production system? In other words, it is asking what can one expect of the performance of a system in Prolog. We do not have the answer in this thesis, but we will discuss briefly some of the prospects in the following.

Much of the performance factor in a Prolog system is tied to the linear search strategy in accessing the database. Partially instantiated keys make it difficult to design general search strategies, but present very challenging problems for search optimization. We refer to (Freeston 86; Vielle 86) for some of the recent work on organizing database search strategies in pre-processing for Prolog compilation.

Another possibility to improve Prolog performance depends on parallel computation. Based on logic and automatic resolution, Prolog appears to be inherently appropriate for parallel computation. In (Dwork 84), it is however proved that the general unification problem is log-space complete, and it leads to the belief that unification, the primitive operation in Prolog, is inherently sequential (Mizell 86). But in practice, many applications in Prolog, such as in polygon overlay, do not depend on the generalized unification, but rather commonly use the unification mechanism in Prolog for term matching (or *one-sided* unification) when searching a database with partially instantiated keys as we have mentioned above. To our delight, searching multi-indexed database can parallelize well to improve performance.

Furthermore, many are also considering architectures more suitable for logic programming such as in Prolog. Hopefully, this will bring answers to the demand for better performance in Prolog systems.

Chapter 7 IMPLEMENTATION AND RESULTS

The chapter describes the organization of the programs for the polygon overlay system, and presents the results of the test runs we have made. Section 7.1 dwells on the details of software organization: programs and data files in the overlay process. Readers who are not interested in these details may skip the section since the concepts are simply re-stated to explain the organization and the strategies involved. The rest of the chapter presents results of our overlay system on two types of test runs: section 7.2 describes tests on its stability, and section 7.3 on its performance.

7.1 Implementation

This section is written for those who are concerned with the details of implementation. Also for them we append to this thesis the source listing of our polygon overlay system and the rational arithmetic package. Our system is in C-Prolog (Pereira 86) version 1.5, a Prolog interpreter written in C. Our system runs on a Sun 2 micro-computer with Unix 4.2 bsd. We have altogether 12 programs organized into four groups, for the four stages of the overlay system. We have revised the system four times now in the course of this development, and we for now have named the system OVER4. OVER4 consists of these four groups of programs:

• XSECT:	to calculate chain intersections.
• LINK:	to form polygon boundaries.
• OVER:	to determine overlay relationships.
• CONTR:	to resolve boundary containment.

We will discuss each group in the following sections.

7.1.1 XSECT: Chain Intersection.

XSECT takes the input maps and calculates all the chain intersections. Each intersection splits the intersecting chains at the intersection point, where a new node is generated. XSECT produces three output files: the new set of network chains, the new nodes, and the old adjacent polygons for each chain. XSECT implements the adaptive grid method for edge segment intersections: the whole process is divided into five steps. We will discuss each step in the following:

dform. We decompose the chains from the input maps into three data sets: boundaries, vertices, and edges. The input chains are complete chains of the following format:

chain($C\#, MAP, N_1, N_2, [\ [x,y],\ldots], P_1, P_2$).

$C\#$ is the chain id. MAP identifies the input map. N_1 and N_2 are the beginning and ending nodes. The list $[\ [x,y],\ldots]$ comprises the coordinates of the polyline from N_1 to N_2. P_1 and P_2 are the polygons respectively on the left and right of the chain. *dform* constructs the following data sets:

b($B\#, P_1, P_2$).
v($V\#, TYPE, [x,y]$).
e($E\#, [V_1, V_2]$).

The id names $B\#$, $V\#$, and $E\#$ are 7 digit integers, with the MAP id, $C\#$, integrated into different fields of the integer. (Note: For this reason, OVER4 may not work in some modified versions of C-Prolog which do not support full 27 bit integers.) The data set **v** includes both vertices and nodes; *TYPE* can be *n* or *v* to indicate the type: node or vertex.

gstat. *gstat* sets the parameters for the adaptive grid. In *gstat*, we take the input map data from *dform* to calculate several statistical measures, and with them we calculate the parameters for the adaptive grid. We examine the coordinates to determine the full extent of the object space. We divide the size of the object space by the average edge segment length to get the number of grid cells in each dimension. The parameters for the adaptive grid consist of:

left(*X-Origin*).	%X origin
bottom(*Y-Origin*).	%Y origin
x_grid(*XN-grids*).	%Number of cells along X-axis
y_grid(*YN-grids*).	%Number of cells along Y-axis
x_gsize(*X-Cell-Size*).	%Grid cell X dimension
y_gsize(*Y-Cell-Size*).	%Grid cell Y dimension

xgrid. In *xgrid*, we cast the grid over the set of edge segments. We go through sequentially the set of edges from *dform*, and maintain a set of edge segments for each non-empty grid cell. *gstat* also needs to refer to the set of vertices from *dform* for the coordinates. Output from *gstat* is the set of (grid, set-of-edges) entries:

ge(*G#*,[E_1,E_2,...]).

We remove the entries for those grid cells that contain only edges from one map, since they do not need to be examined for edge segment intersection.

xsect. *xsect* performs pairwise comparison between the edges if the pair occupies some common grid cell. We take the set of (grid, set-of-edges) entries from *xgrid* and examine every entry. We calculate edge segment intersections for each pair of edges which are from different input maps but occupy the same grid cell. *xsect* updates the set of edges as well as the set of vertices, both from *dform.* When two edge segments intersect, a new node is generated and added to the set of vertices,

$$\mathbf{v}(N\#, n, [x, y]).$$

and the new node is marked on the intersecting edge segments as

$$\mathbf{e}(E\#, [V_1, N\#, V_2]).$$

xsect also keeps track of all the equations for the edge segments. An equation is calculated once when the edge segment is first examined, and is kept for subsequent use. When *xsect* examines a pair of edge segments, it also keeps a record $\mathbf{ok}(E_1, E_2)$ so that if the pair occupies more than one common grid cell, they will not have to be examined again. *xsect* takes care of all special cases of edge segment intersection. The output from *xsect* are the two updated sets of vertices and edges.

xconn. *xconn* connects the edge segments from xsect back into chains, having them split at the new nodes. We take the two sets of edges and vertices from *xsect,* and make references to the set of boundaries from *dform.* From *xconn,* we can get the set of network chains and the set of nodes for the output map:

> **c**(*C#*, N_1, N_2, [[x,y],...]).
> **n**(*N#*, [x,y]).

7.1.2 LINK: Polygon Boundary Formation

The network of chains and nodes partitions the 2D plane into polygons. LINK takes the network chains and the nodes of the output map to form the polygon boundaries, generating new names for each polygon. Once we have the polygons, we can update the network chains with complete information of the adjacent polygons. LINK comprises four programs:

calci. We calculate for each chain the incident angles at both the beginning and ending nodes. Taking for input the network chains from *xconn*, *calci* computes the angle each incident vector makes with the positive X-axis. Each node is therefore associated with either the head or tail of each incident chain (*h* (*C#*) or *t* (*C#*)) and the incident angle. *calci* generates the **incidence** file:

> **i**(*N#*, *Chain-Incidence*, *ANGLE*).

The *ANGLE* is represented by an ordered pair [Q,A] where Q =1,2,3,4 to identify the quadrant the angle falls in, and A is a rational number of the ratio such that

$$A = \frac{sin(ANGLE)}{cos(ANGLE)} \text{ for } Q=1,3 \text{ and } A =- \frac{cos(ANGLE)}{sin(ANGLE)} \text{ for } Q=2,4$$

This is done to preserve accuracy using rational arithmetic. The representation conforms to the counter-clockwise positive cyclic order.

sorti. With the trig-rational representation for *ANGLE* 's, we can sort them into proper cyclic order. Taking the incidence file from *calci* for input, *sorti* sorts the incident chains around each node by the incident angles. The set of nodes from *xconn* is then updated with the list of chain incidences at each node. The output nodes have the format:

node($N\#,[x,y],[I_1,I_2,...]$).

where I_i is either t ($C\#$) or h($C\#$) to indicate tail or head of the chain. More, *sorti* generates **linkage** file with one entry for each polygon corner:

link($[B_1,B_2],B_2$).

where B_i is either th($C\#$) or ht ($C\#$) to indicate the direction of the chain (tail-to-head or head-to-tail). The last element in the list is duplicated for easy access.

cpoly. *cpoly* takes the **linkage** file from *sorti* and matches up the records to form the boundary chains of each polygon. Each completed list of boundary chains identifies a polygon boundary; *cpoly* generates new polygon names and the output **polygon** file:

p($P\#,[B_1,B_2,...]$).

<u>*bound.*</u> When we have the polygons, we can update the network chains from *xconn* with the information of the adjacent polygons for each chain. *bound* takes the polygon file from *cpoly* and generates for each entry of the list of boundary chains

left(*C#*,*P#*).	% for *th* (*C#*).
right(*C#*,*P#*).	% for *ht* (*C#*).

Then *cpoly* updates each chain from *xconn* as

chain($C\#, N_1, N_2, [\ [x,y], ...], P_1, P_2$).

7.1.3 OVER: Overlay Identification

OVER takes the topology of the new output map and determines for each output polygon *P#* the input polygons P_1 and P_2 in the two input maps such that *P#* is from $P_1 \cap P_2$. The output from OVER is the file of overlay relationships, of the following format:

over($P\#, P_1, P_2$).

with one entry for each output polygon. OVER comprises two programs. They are described below:

<u>*xover.*</u> *xover* takes the polygon file from LINK and the boundary file from *dform* for input. *xover* goes through the list boundary chains for each polygon to find the input polygons. If both P_1 and P_2 can be found, *xover* enters the information into the file for overlay relationships as **over**($P\#, P_1, P_2$). If only one of the two can be found, *xover* keeps a record

of either **overl**($P\#,P_1$) or **over2**($P\#,P_2$). The next step in *xover* examines each entry in **overl** and **over2.** *xover* searches the neighbors of $P\#$ to resolve the overlay relationship. If resolved, an entry for $P\#$ is appended to the overlay relationships. If not, the connected group of polygons is identified as

$$\textbf{overl}(Group\#,P\#,P_1). \quad \text{or} \quad \textbf{over2}(Group\#,P\#,P_2).$$

pcont. For each connected group of polygons, *pcont* performs the point-in-polygon test to determine the containment polygon. *pcont* then resolves all the **overl** and **over2** entries and the file of overlay relationships is appended.

7.1.4 CONTR: Containment Resolution

Our definition for a polygon is that it is a connected subset of the 2D plane. Therefore, a polygon may have holes. In OVER, we have treated each polygon boundary, holes included, as a separate polygon. CONTR now determines which hole is inside which polygon and updates the information in the output files of the new map. CONTR has only one module:

jconn. In *jconn*, we join the connected pieces of polygons. We take the file of **over**($P\#,P_1,P_2$) and search for $P\#$'s with the same P_1 and P_2. Each then identifies a polygon or a hole. *jconn* determines the cyclic order in the boundary list to separate the holes and the polygons. Then each hole is tested by point-in-polygon against each of the polygons. If a hole is inside a polygon, the two $P\#$'s refer to the same polygon. The information in the set of chains and polygons will be updated, and one of the two entries of **over**($P\#,P_1,P_2$) will be deleted.

The output map from OVER4 comprises four files:

node($N\#,[x,y],[h\ (C\#)\ or\ t(C\#),\ ...]$).
chain($C\#,N_1,N_2,[\ [x,y],...],P_1,P_2$).
polygon($P\#,[th\ (C\#)\ or\ ht(C\#),\ ...]$).
over($P\#,P_1,P_2$).

These files describe, respectively, the nodes, the chains, the polygons, and the overlay relationships.

7.2 Test Runs On Stability

There are some commonly known tests for polygon overlay systems: the overlay of a map on itself, and on a reduced version of the map itself. We have done both of these here and an additional test of a map over itself rotated by a small angle.

We use a United States map of the state boundaries. The map has 164 chains, 913 edges, 861 nodes and vertices, and 50 polygons. Figure 7.1 shows the map. We performed an overlay of the map on itself. The overlay system generated a 113×73 adaptive grid, examined 5387 pairs of edges and 873 actually intersected. The US map survived the test and passed undistorted. The measurements concerning the adaptive grid and total CPU time are tabulated together with the other tests in Figure 7.6.

The second test involved expanding the US map to higher complexity. We took the US map and for each edge segment long enough, we shifted the mid-point to one side in the direction perpendicular to the edge by a distance $\frac{1}{10}$ of the edge length. An edge was long enough if the perturbation would not result in any internal inconsistency. We repeated the process two times to generate a US map with approximately four times the number of edge segments. Figure 7.2 shows the US map with more than 2 times the complexity: 164 chains, 1916 edges, 1864 nodes and vertices. We put this map and the US map through the overlay system. Figure 7.3 shows the output map, which now has 1738 chains, 3612 edges, 2948 nodes and vertices.

In the third test, we first rotated the US map by 1° about St. Louis (approximately the center). Figure 7.4 shows the rotated map with reference to the US map in dotted lines. Then we performed an overlay with the two maps. Figure 7.5 shows the output map, which has 1288 chains, 2773 edges, 2619 nodes and vertices.

These results should verify the stability of our polygon overlay system. While the measurements involved with the adaptive grid and CPU time usage are interesting, we do not intend to make further inference here, but we have included the data for future reference.

7.3 Performance Results

In this section we will demonstrate our polygon overlay system with some "normal" test runs: for the categorization of spatial data. We will still use the same US map which we also used in section 7.2. We will also tabulate the distribution of resources, namely CPU time and storage space, in the various steps of the process.

Figure 7.7 shows the US map with an overlay of January isotherms. The overlay layer of January isotherms contains 24 chains, 633 edges, and 627 nodes and vertices. Figure 7.8 has the timing and storage space usage for each step tabulated.

Figure 7.9 show the US map with an overlay of July isotherms. The overlay layer of July isotherms contains 28 chains, 884 edges, and 880 nodes and vertices. Figure 7.10 has the timing and storage space usage for each step tabulated.

Looking at the results, we can be sure that our experimental system will not measure up to the speed necessary for a production system. However, our results also show that it is due to the Prolog interpreter and the use of rational arithmetic. We do note that this is not a necessary trade-off for the stability we have achieved. The reasons are: (1) we do not have to tie ourselves to Prolog, even though a conventional language would make the programming task more cumbersome; (2) our implementation of rational arithmetic is not aimed at fast execution. A good package for rational arithmetic can perform much better. We have however made clear that the asymptotic growth of CPU time for rational arithmetic is only linear to the size of the data volume.

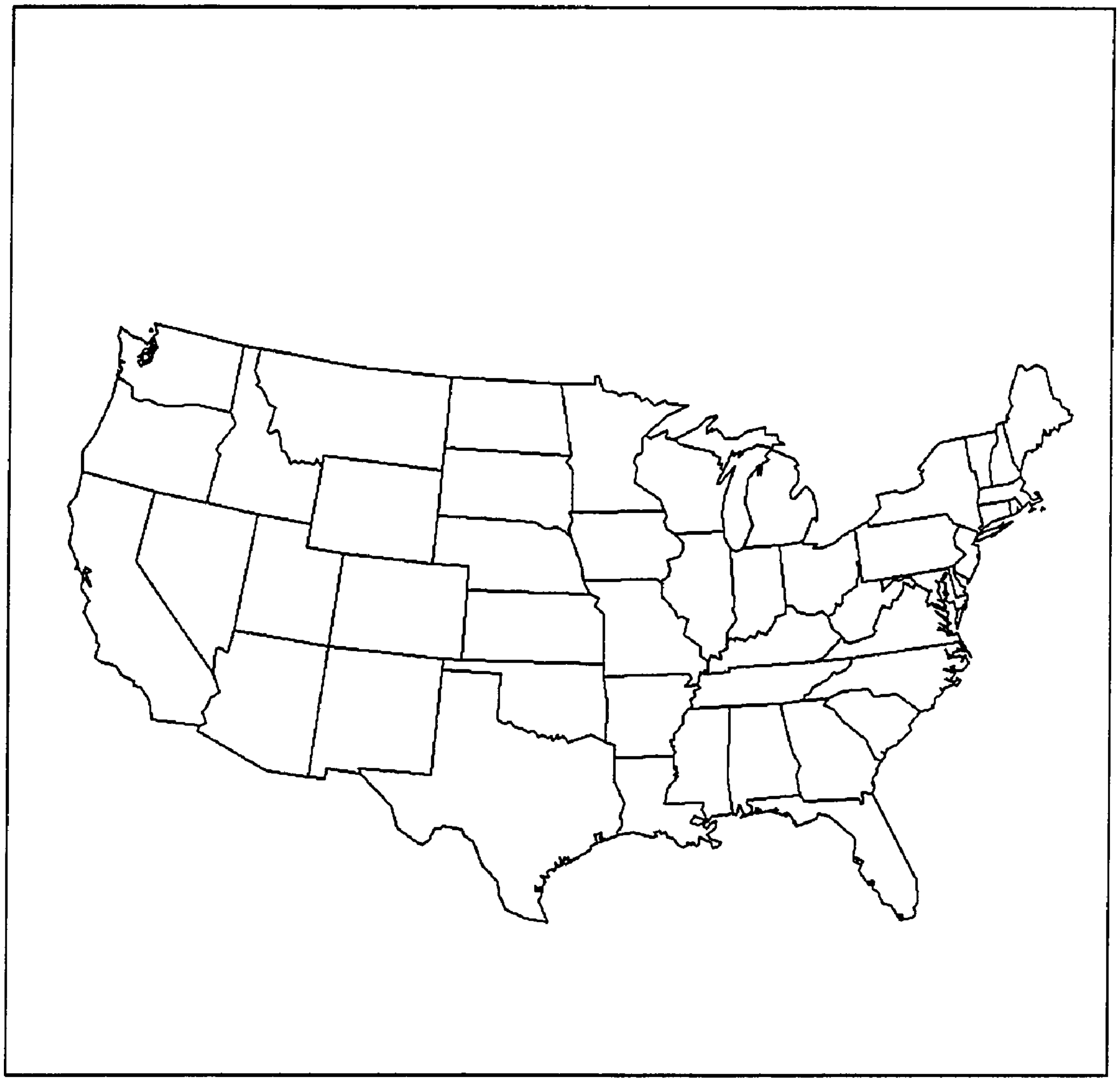


Figure 7.1 The US Map of state boundaries

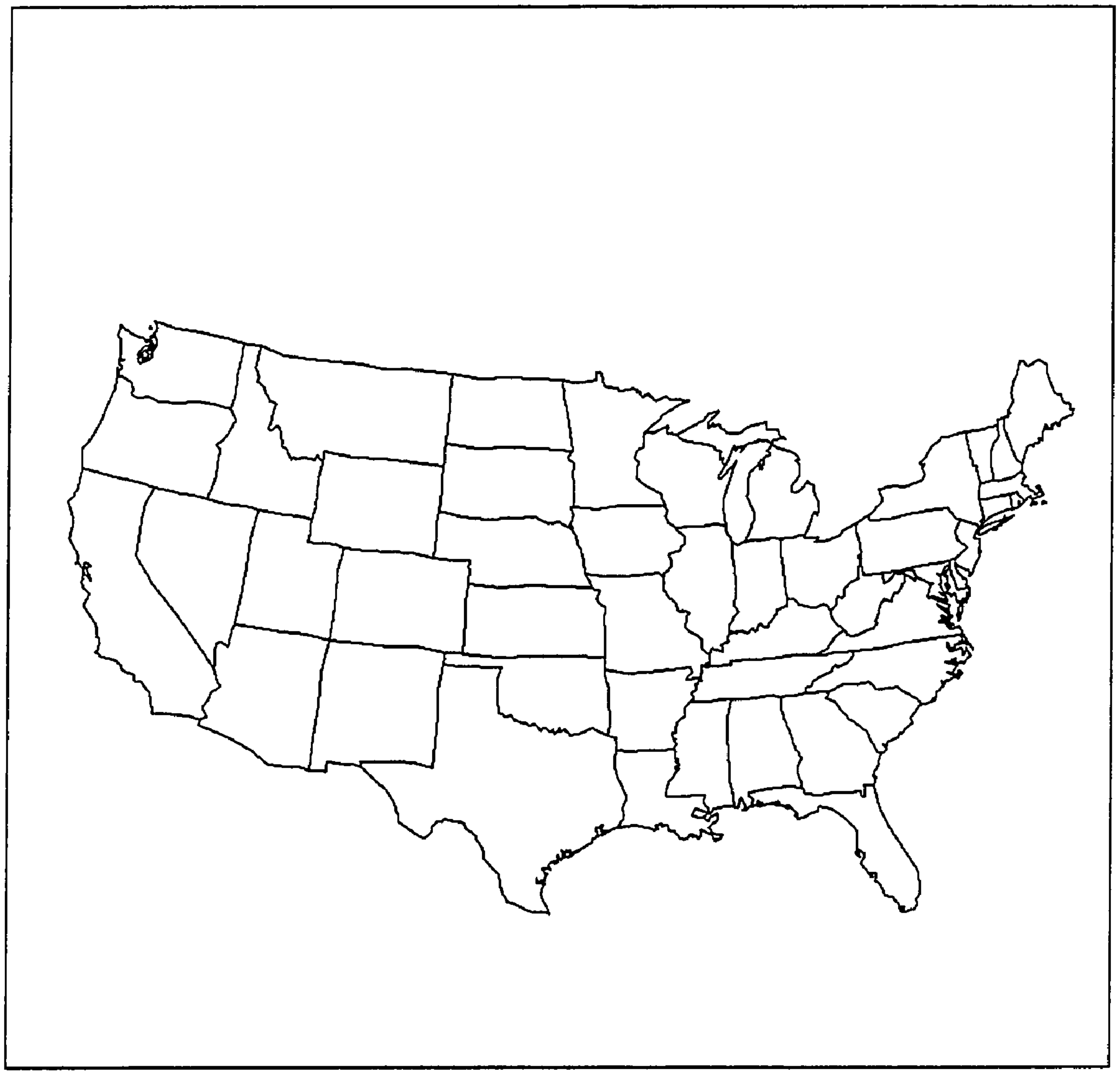


Figure 7.2 The US map with >2 times complexity

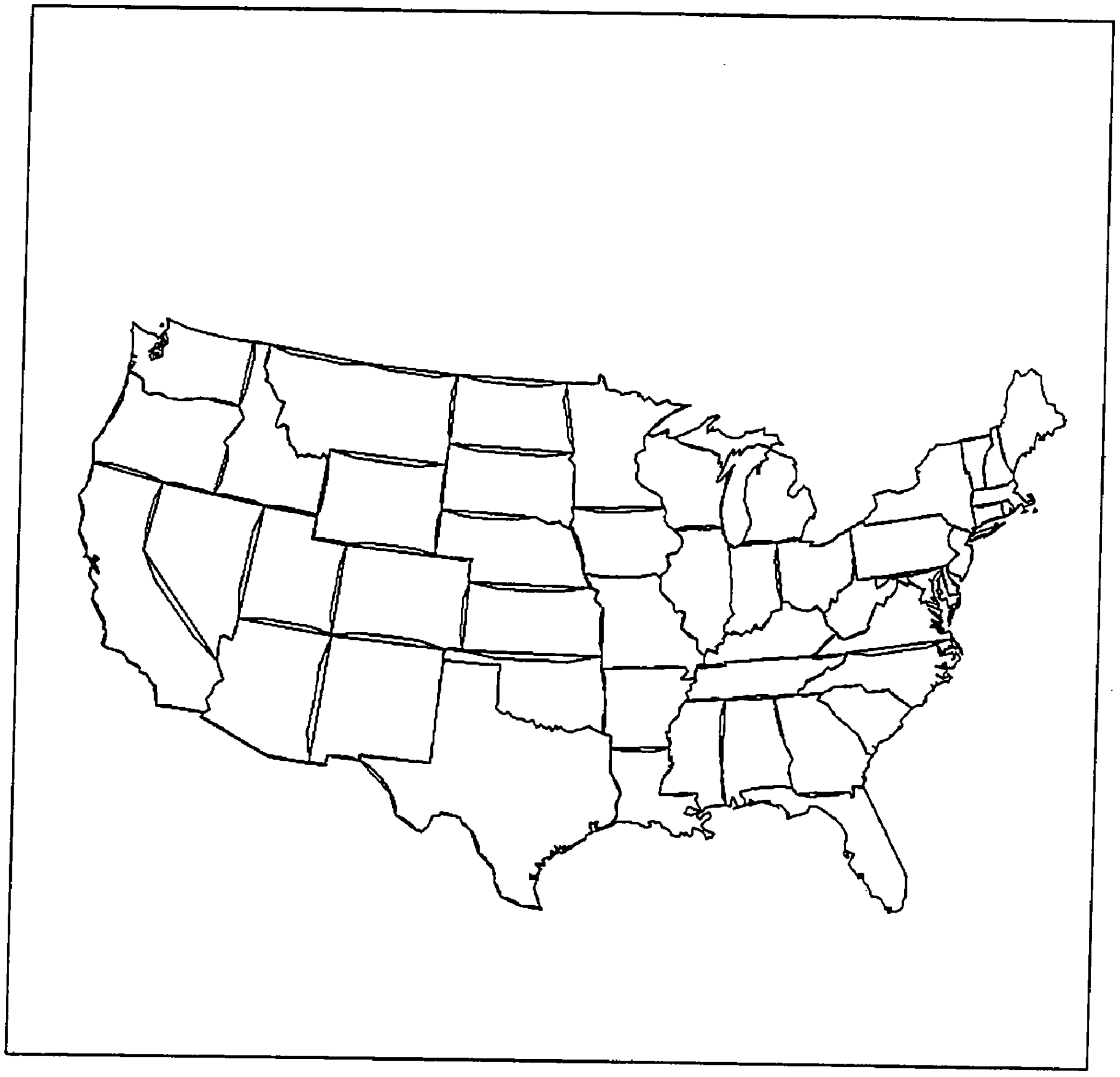


Figure 7.3 The US map overlay on itself with >2 times complexity

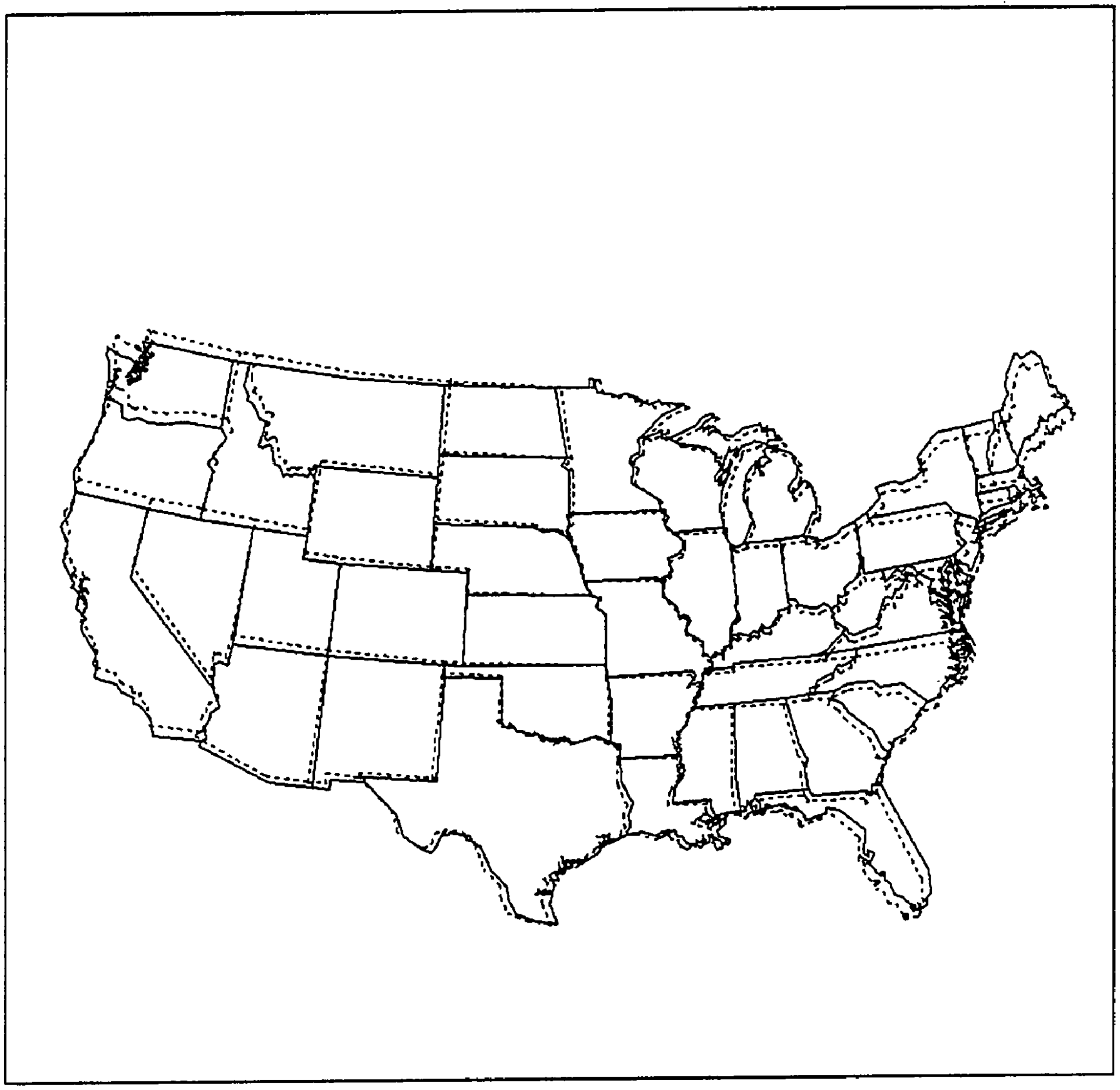


Figure 7.4 The US map rotated 1° about St. Louis

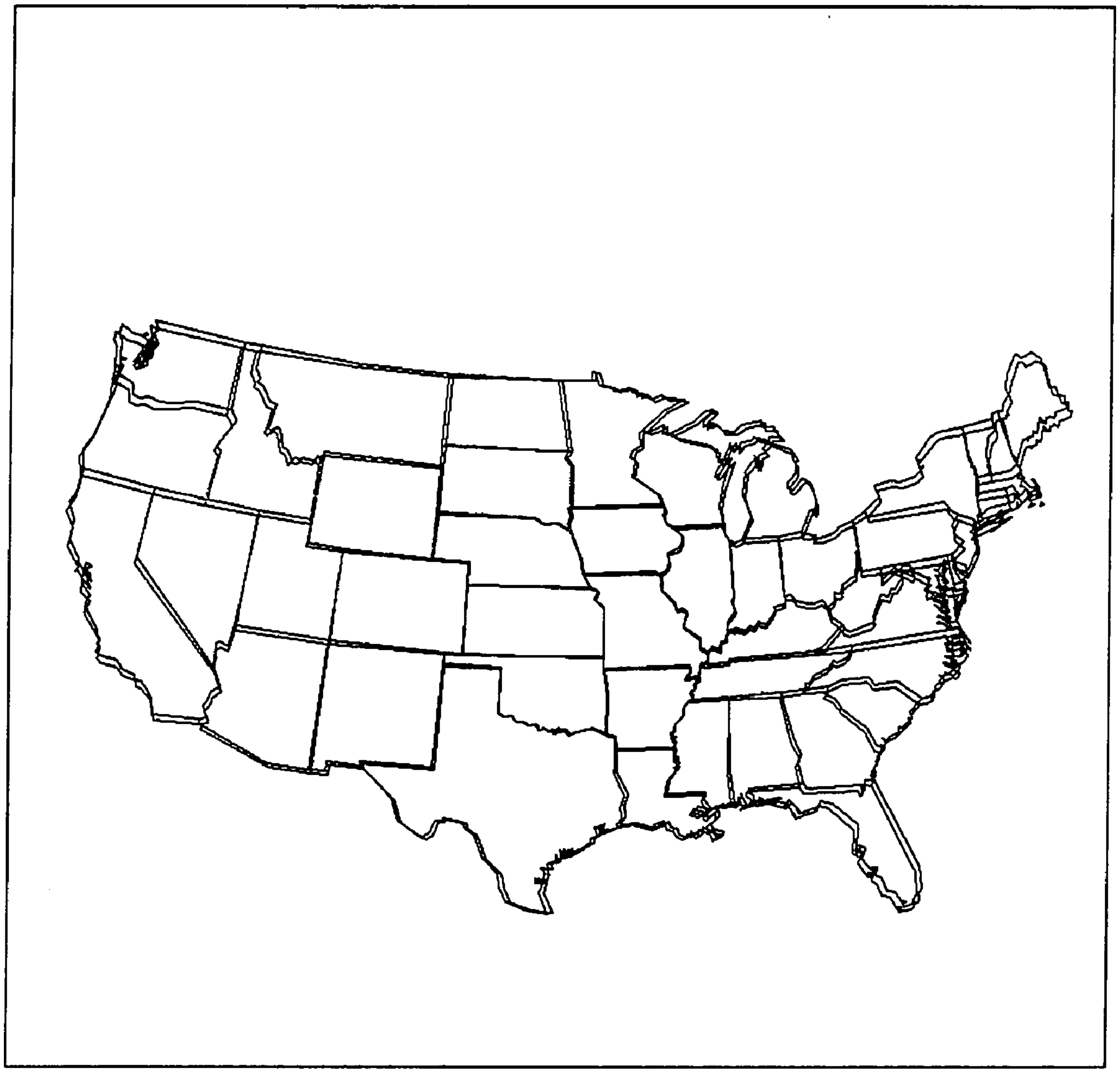


Figure 7.5 The US map overlay on itself rotated by 1 °

	# of edges	adaptive grid	# of pairs examined	# of inter-sections	total CPU (seconds)
USA	913	133 ×73	5387	873	54637
>2 times USA	1916	175 ×113	6219	910	91668
rotated USA	913	113 ×74	4632	483	44716

Figure 7.6 Stability Tests: timing and measurements

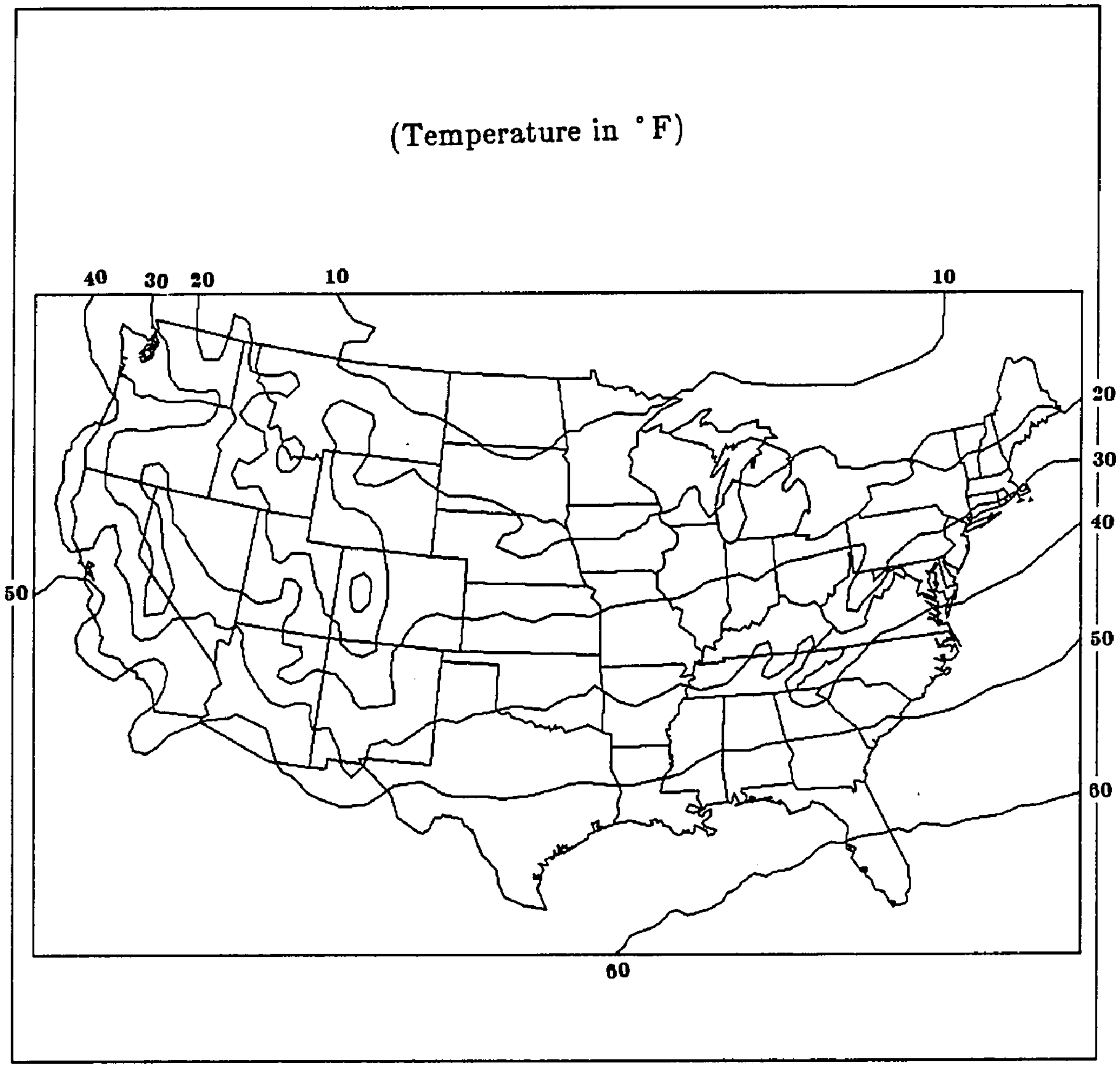


Figure 7.7 The US map with January isotherms

Overlay Performance: USA with *January* isotherms							
	CPU (seconds)			STORAGE (K-words)			
	total	XQ	gcd	atom	heap	global stack	local stack
dform	63	-	-	20	282	30	13
gstat	739	3	1	25	328	49	109
xgrid	6518	5693	25	170	690	23	6
xsect	4641	3822	224	24	375	16	3
xconn	566	-	-	19	393	31	19
calci	1686	1620	731	23	292	7	3
sorti	87	-	-	20	238	1	20
cpoly	193	-	-	19	108	1	2
bound	139	-	-	19	415	4	2
xover	20	-	-	19	415	4	2
pcont	228	-	-	24	188	78	27
jconn	160	-	-	24	279	204	15

Figure 7.8 Overlay Performance: US map with January isotherms

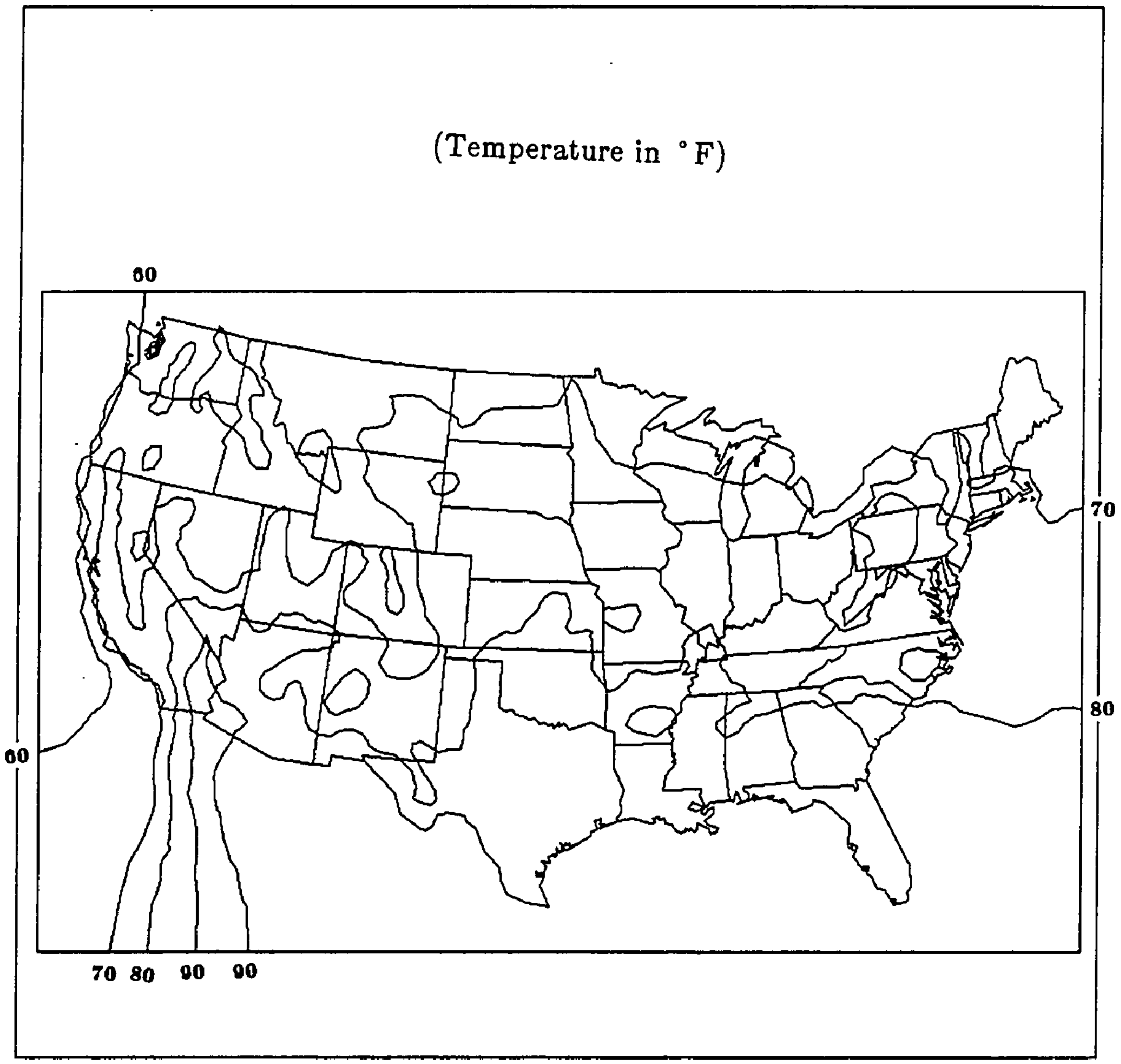


Figure 7.9 The US map with July isotherms

Overlay Performance: USA with *July* isotherms							
	CPU (seconds)			STORAGE (K-words)			
	total	XQ	gcd	atom	heap	global stack	local stack
dform	76	-	-	20	312	59	26
gstat	941	3	1	25	371	58	129
xgrid	7448	6405	29	204	818	36	12
xsect	4873	3934	271	24	409	16	3
xconn	762	-	-	19	434	58	35
calci	1740	1671	766	23	309	7	3
sorti	91	-	-	20	243	1	2
cpoly	182	-	-	19	109	1	2
bound	149	-	-	19	447	3	2
xover	23	-	-	20	280	3	2
pcont	389	-	-	24	204	143	52
jconn	246	-	-	24	297	207	15

Figure 7.10 Overlay Performance: US map with July isotherms

Chapter 8 CONCLUSION

This chapter presents a summary of the thesis, and reiterates the research contributions. Then we will discuss some further considerations, most of which concern the more realistic issues involved in automated cartography and map data processing, as well as further applications of Prolog in these areas.

8.1 Summary

Polygon overlay is the process of superimposing two maps into one, so that the output map conveys the selected information of the input maps together to illustrate the spatial correlation between them. Before the early 60's, the process was done manually. The cartographer made transparencies of the original maps and traced out the new map on the light table. Computerized data processing did not come to aid this manual and tedious process until the 70's. However, attempts to automate the process revealed that polygon overlay encompasses a host of many subproblems. These are mostly related to geometry and topology, but also include the computer sciences such as algorithm design, complexity analysis, arithmetic and numerical representations. We reviewed the research work in computational geometry and specifically polyline intersection, which contributed fundamentally to the development of algorithms for polygon overlay. We also reviewed a few systems and software packages that performed polygon overlay.

Prolog represents a radically new approach toward programming, an approach known as logic programming. Instead of presenting a prescribed set of instructions, a Prolog program describes the objects and their relationships involved in the problem to be solved. Using Prolog for geometry problems is a venture motivated by the quest for better programming tools in computational geometry. The polygon overlay problem becomes for us a vehicle to investigate the practicability and suitability of using Prolog for geometry applications.

We developed a polygon overlay system in Prolog, implementing an algorithm due to Franklin (Franklin 83a) and extending it to handle also maps with separate components. The algorithm decomposes the complex process of polygon overlay into simple stages, resulting in much simplified data structures. Each stage is further subdivided into a number of steps. The following are the four major stages:

Stage 1. Chain Intersection.
Determine intersecting chains and split them at the intersection points.

Stage 2. Polygon Boundary Formation.
Link up the chains to form the polygon boundaries.

Stage 3. Polygon Overlay Identification.
For each polygon boundary, identify the two input polygons to establish the overlay relationship.

Stage 4. Boundary Containment Resolution.
Resolve the containment relationships between boundaries to compound polygons (with multiple boundaries).

The system adopts a relational approach to data structuring. Geometric entities are defined as Prolog facts, and the Prolog rules encoding geometry algorithms perform data processing.

An adaptive grid sorts out potentially intersecting edge segments to within those that occupy some common grid cells. We determine the chain intersections by pairwise comparisons of the edges in each grid cell. Geometric intersections are calculated using rational arithmetic. This preserves numerical accuracy and thus topological consistency. It also allows proper handling of special cases of touching and overlapping chains. Hence, we can guarantee stability in processing geometric intersections.

In using Prolog, we have demonstrated several advantages as a programming tool for geometry applications. Besides the general high level nature of logic programming and a built-in relational data base in Prolog, we formulated a few paradigms of programming revealed to be useful in our application.

In conclusion, we have achieved in the experimental polygon overlay system the following two-fold purpose:

On the one hand, we presented rational arithmetic as a practicable solution to the problems stemming from discretization errors, in polygon overlay. We achieved stability in the process of computation on two aspects: (1) rational arithmetic preserves numerical accuracy in calculating geometric intersections, and (2) a line segment intersection algorithm complete with all special cases can properly identify and handle tangential situations.

On the other hand, we showed that Prolog is a viable programming tool for geometric and topological problems. Prolog offers a general high level environment and a relational approach toward data structuring. Specifically, we formulated three paradigms of logic programming which appear to be useful: set-based operation, pattern matching geometric/topological properties, and using unification to form equivalence classes.

8.2 Further Considerations

We have defined our polygon overlay problem to be strictly geometric and topological. In this section, we will present some realistic issues involved in automated cartography and map data processing. These may be referred to as extensions to the polygon overlay system or further application areas.

8.2.1 Sliver Removal

We have used rational arithmetic to guarantee stability in numerical computation and circumvent possible problems of discretization errors leading to topological inconsistency (Franklin 84). However, a realistic problem with map data is nearly coincidental input data: two input maps may have data values of the same prominent feature (such as road or river) only approximately equal. As a result, the overlay process generates an output map with sliver polygons around the feature. These slivers should be removed as a human cartographer would remove them. We can outline certain rules to automatically recognize these sliver polygons: a sliver can often be characterized by its size and shape, and the small number of edge segments around the boundary which involves two chains, one from each map (Goodchild 77). Figure 8.1 illustrates three kinds of slivers, along the trail of two nearly coincidental input chains: a rounded sliver, an elongated strip, and a bent strip. We can respectively identify these slivers by small area, small minimum diameter (diameter is defined as the distance between two parallel lines enclosing the polygon), and small ratio of its area to that of its convex hull. To remove a sliver, we can coalesce it with one of its neighboring polygons which is not a sliver. This is to avoid coalescing slivers into a non-sliver polygon. A polygon is coalesced to its neighbor by removing the common boundary chain, and updating the new polygon with combined attributes.

8.2.2 Map Data Verification

We have freely assumed data consistency in our input maps. Unfortunately, real maps are often plagued with errors and inconsistencies. An interesting idea is to exploit overlay processing on one map to perform data consistency verification. If an overlay system can identify an error in given map data, or even better the source of such an error, it may also possibly edit the data intelligently in the process. Some of these data

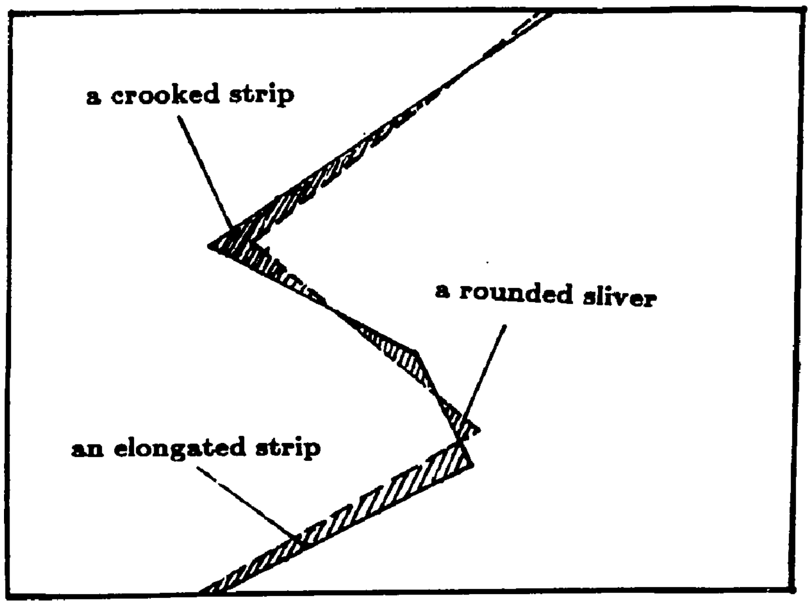


Figure 8.1 Three kinds of sliver polygons

inconsistencies may be chains crossing over each other (which need to be separated), or same chain duplicated, or polygon identifier mismatch around its boundary chains. Figure 8.2 illustrates the case of polygon identifier mismatch due to incorrect orientation of one chain.

8.2.3 Map Generalization

Probably the most challenging problem in automated cartography is that of map generalization - the problem of generating a map of reduced scale from a given map. A map communicates spatial relationships. Map generalization involves simplifying features, deleting insignificant features, and converting features from one type to another, to effectively communicate the spatial relationships in the reduced scale. Figure 8.3 depicts an example. An automated map generalization system will deal with not only geometric/topological computation, but also common sense and expert reasoning of the cartographer. While numerous applications have been done in Prolog to demonstrate automated reasoning, we have shown that it is also a viable tool for geometric/topological computation. We therefore suppose that Prolog would be effective for automated map generalization.

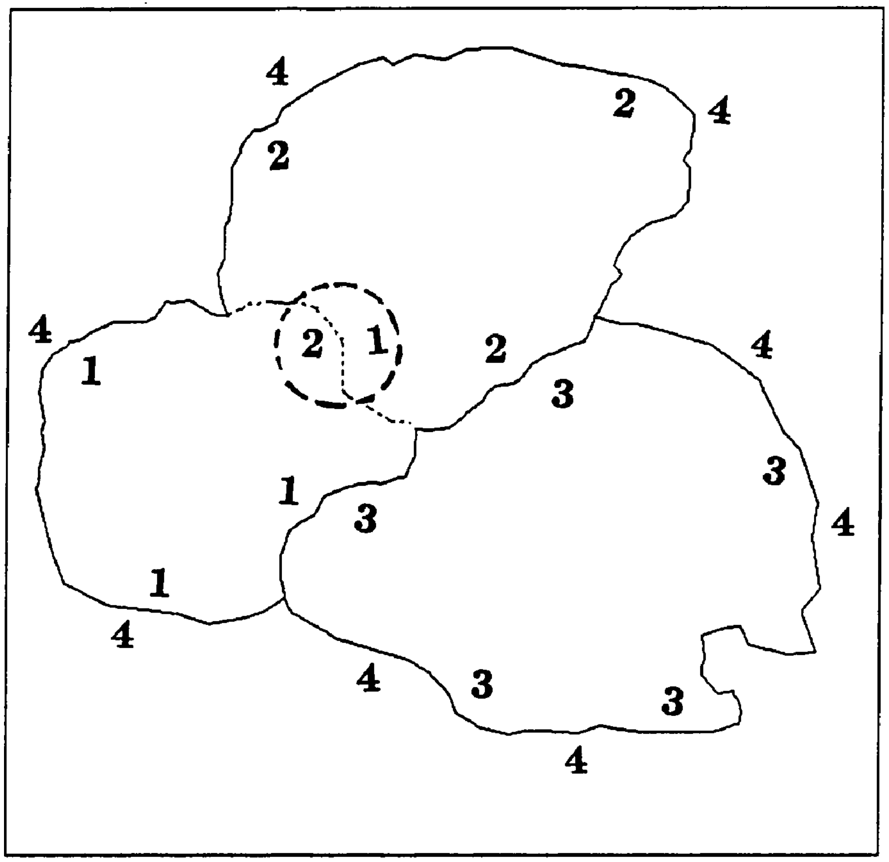


Figure 8.2 Polygon id mismatch due to incorrect orientation of a boundary chain

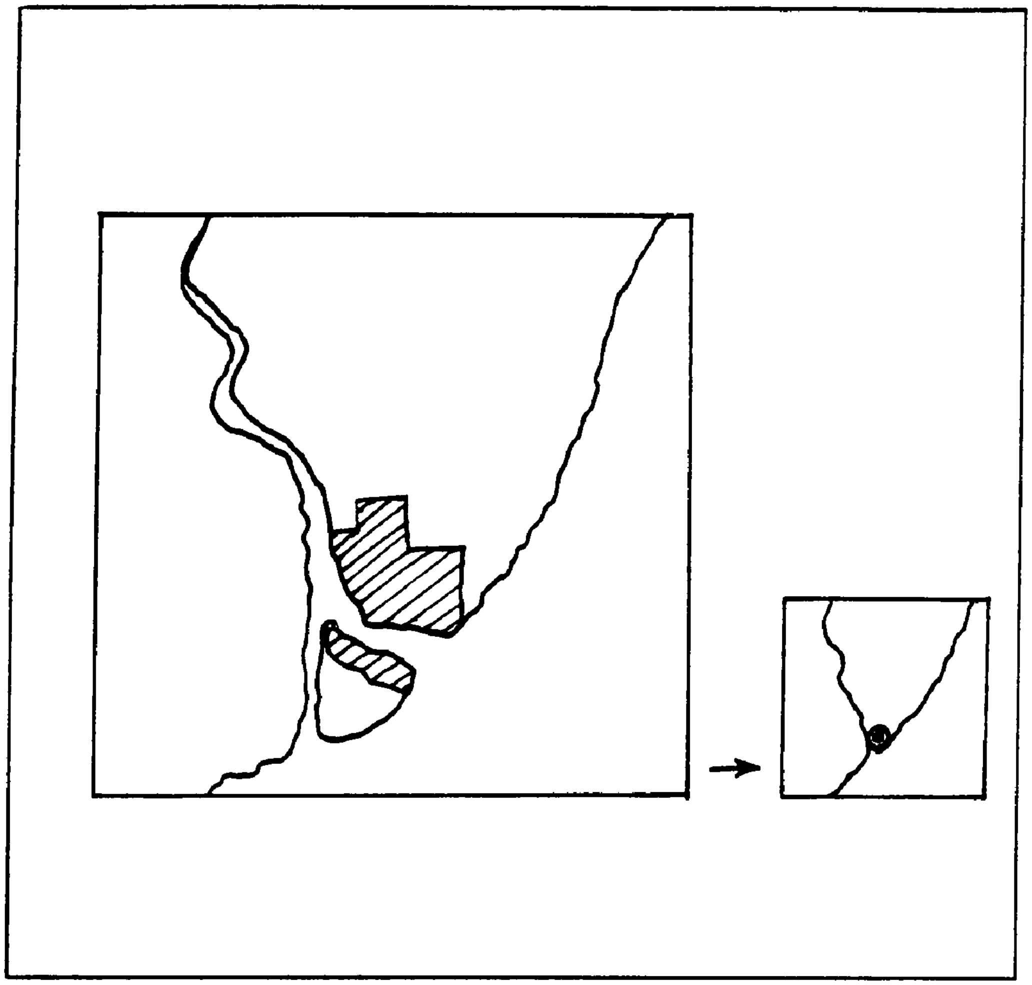


Figure 8.3 A map generalization example

BIBLIOGRAPHY

Aho, Alfred V., J.E. Hopcroft, and J.D. Ullman. (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts.

Ballard, Dana. (1981) "Strip Trees," *Communications of ACM*, Vol.24, No.5, pp.310-321.

Bentley, Jon L. and T.A. Ottmann. (1979) "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Transactions on Computer*, C-28, 9, pp.643-647.

Bruderlin, B. (1985) "Using Prolog for Constructing Geometric Objects Defined by Constraints," *Proceedings*, EUROCAL, Linz, Austria.

Burton, Warren. (1977) "Representation of Many-sided Polygons and Polygonal Lines for Rapid Processing," *Communications of ACM*, Vol.20, No.3, pp.166-171.

Chrisman, Nicholas R. (1976) "Local versus Global: the Scope of Memory Required For Geographic Information Processing," *Internal Report 76-14*, Laboratory for Computer Graphics and Spatial Analysis, Harvard University, Cambridge, Massachusetts.

Clocksin, W.F. and C.S. Mellish. (1981) *Programming in Prolog*, Springer-Verlag.

Colmerauer, A., H. Kanoui, R. Pasero and P. Roussel. (1973) *Un Systeme de Communication Homme-Machine en Francais*, Groupe d'Intelligence Artificielle, Faculte des Sciences de Luminy, Marseilles, France.

Colmerauer, Alain. (1985) "Prolog In 10 Figures," *Communications of ACM*, Vol.28, No.12, pp.1296-1310.

DeBerry, T. (1979) *Polygon Information Overlay System User's Manual*, Environmental Systems Research Institute, Redlands, California.

Dutton, Geoffrey. (1977) "Navigating ODYSSEY," *Proceedings*, Advanced Study Symposium on Topological Data Structures for Geographic Information Systems, Vol.3, Laboratory for Computer Graphics and Spatial Analysis, Harvard University, Cambridge, Massachusetts.

Dutton, Geoffrey. (1981) "Fractal Enhancement of Cartographic Line Detail," *The American Cartographer*, Vol.8, No.1, pp.23-40.

Eastman, C.M. and C.I. Yessio. (1972) "An Efficient Algorithm for Finding the Union, Intersection, and Differences of Spatial Domains," *Technical Report*, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Ferguson, H.R. (1973) "Point in Polygon Algorithms," *Technical Report*, Urban Data Center, University of Washington, Seattle, Washington.

Foderaro, John K., K.L. Sklower and K. Layer. (1983) *The Franz Lisp Manual*, University of California, Berkeley, California.

Foley, John D. and A. vam Dam. (1982) *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, Massachusetts.

Forrest, A. Robin. (1986) "Computational Geometry and Software Engineering," *invited talk*, Second ACM Annunal Symposium on Computational Geometry, Yorktown Heights, New York.

Franklin, W. Randolph (1972) "ANOTB: Routine to Overlay Two Polygons," *Collected Algorithms*, Douglas (eds), Laboratory for Computer Graphics and Spatial Analysis, Harvard University, Cambridge, Massachusetts.

Franklin, W. Randolph (1983a) "A Simplified Map Overlay Algorithm," *Proceedings*, Harvard Computer Graphics Conference, Vol.I, Cambridge, Massachusetts.

Franklin, W. Randolph. (1983b) "Adaptive Grids for Geometric Operations," *Proceedings*, Sixth International Symposium on Automated Cartography, (AUTO-CARTO 6), Vol.II, pp.230-239, Ottawa, Ontario.

Franklin, W. Randolph (1984) "Cartographic Errors Symptomatic of Underlying Algebra Problems," *Proceedings*, First International Symposium on Spatial Data Handling, Vol.II, pp.190-208, Zurich, Switzerland.

Franklin, W. Randolph, Peter Y.F. Wu, S. Samaddar and M. Nichols. (1986) "Prolog and Geometry Projects," *IEEE Computer Graphics and Applications*, Vol.6, No.11, pp.46-55.

Freeman, Herbert and R. Shapira. (1975) "Determining the Minimum Area Encasing Rectangle for An Arbitrary Closed Curve," *Communications of ACM*, Vol.18, No.7, pp.409-413.

Freeston, M. (1986) "Data Structures for Knowledge Base: Multi-dimensional File Organizations," *Technical Report* TR-KB-13, European Computer-Industry Research Center, Munich, West Germany.

Fuchi, K. (1981) "Aiming for Knowledge Information Processing System," International Conference on Fifth Generation Computer Systems, Tokyo, Japan.

Gentleman, W.M. and S.B. Marovich. (1974) "More on Algorithms That Reveal Properties of Floating Point Arithmetic Units," *Communications of ACM*, Vol.17, No.5, pp.276-277.

Gonzalez, J.C., M.H. Williams and I.E. Aitchison. (1984) "Evaluation of the Effectiveness of PROLOG for a CAD Application," *IEEE Computer Graphics and Applications*, Vol.4, No.3, pp.67-75.

Goodchild, Michael F. (1974) *PLUS: Technical Users' Guide*, Department of Geography, University of Western Ontario, London, Ontario.

Goodchild, Michael F. (1977) "Statistical Aspects of the Polygon Overlay Problem," *Proceedings*, Advanced Study Symposium on Topological Data Structures for Geographic Information Systems, Vo.1, Laboratory for Computer Graphics and Spatial Analysis, Harvard University, Cambridge, Massachusetts.

Guerrieri, Ernesto and Vinod Grover. (1986) "Octree Solid Modeling with Prolog," *Reprint*, First International Conference on Applications of Artificial Intelligence to Engineering Problems, Southampton, United Kingdoms.

Guevara, J. Amando (1983) "A Framework for the Analysis of Geographic Information System Procedures: The Polygon Overlay Problem, Computational Complexity and Polyline Intersection," *Ph.D. Thesis*, State University of New York, Buffalo, New York.

Guibas, Leonidas J. and Raimund Seidel. (1986) "Computing Convolutions by Reciprocal Search," *Proceedings*, Second ACM Annunal Symposium on Computational Geometry, Yorktown Heights, New York.

Dwork, Cynthia, P.C. Kanellakis, and J.C. Mitchell. (1984) "On The Sequential Nature Of Unification," *Journal of Logic Programming*, No.1, pp.35-50.

Knuth, Daniel E. (1972) "Sorting and Searching," *The Art of Computer Programming*, Vol.3, Addison-Wesley, pp.170-180.

Knuth, Daniel E. (1976) "Big Omicron and Big Omega and Big Theta," *SIGACT News*, 8, 2, pp.18-24.

Kowalski, Robert A. (1974) "Predicate Logic as a Programming Language," *Proceedings*, International Federation of Information Processing Conference, Stockholm, Sweden.

Lam, Nina S. (1977) "Polygon Overlay: An Examination Of An Algorithm and Related Problems," *Master Thesis*, Department of Geography, University of Western Ontario, London, Ontario.

Lee, D.T. and F.P. Preparata. (1984) "Computational Geometry - A Survey," *IEEE Transactions on Computers*, C-33, 12, pp.1072-1101.

Little, J.J. and T.K. Peucker. (1979) "A Recursive Procedure for Finding the Intersection of Two Digital Curves," *Computer Graphics and Image Processing*, Vol.10, No.2, pp.159-171.

Macsyma Group. (1983) *MACSYMA Reference Manual*, Version 10, Vol.II, MIT Press, Cambridge, Massachusetts.

Mairson, Harry G. and J. Stolfi. (1987) "Reporting and Counting Intersections Between Two Sets of Line Segments," *unpublished manuscript*, personal communication.

Malcolm, M.A. (1972) "Algorithms to Reveal Properties of Floating Point Arithmetic," *Communications of ACM*, Vol.15, No.11, pp.949-951.

Mandelbrot, B.B. (1977) *Fractals - Form, Chance And Dimension*, W. Freeman, San Francisco.

McAlpine, J.R. and B.G. Cook. (1971) "Data Reliability from Map Overlay," Division of Land Research, Canberra, Australia.

Mizell, David. (1986) "Prolog and Parallelism: The Inherently Sequential Nature Of Unification," *Naval Research Review*, No.1, 1986, pp.22-24.

Nichols, Magaret. (1985) "A Prolog Implementation Of The Graphics Kernel System," *Master Thesis*, Department of Electrical, Computer, and System Engineering, Rensselaer Polytechnic Institute, Troy, New York.

Nievergelt, J. and F.P. Preparata. (1982) "Plane-Sweep Algorithms for Intersecting Geometric Figures," *Communications of ACM*, Vol.25, No.10, pp.739-747.

Pereira, Fernando, ed., et al. (1986) *C-Prolog User's Manual*, Version 1.5, Department of Architecture, University of Edinburgh, Edinburgh, United Kingdoms.

Potmesil, Michael and H. Freeman. (1980) "Implementation of Two Hidden Line Algorithms," *Computer and Graphics*, Vol.5, No.1, pp.31-40.

Preparata, Franco P. and Michael I. Shamos. (1986) *Computational Geometry: an introduction*, Springer-Verlag.

Reed, C.W. (1982) *Map Overlay and Statistical System User's Manual*, Western Energy and Land Use Team, U.S. Fish and Wildlife Services, Fort Collins, Colorado.

Robinson, J.A. (1965) "A Machine Oriented Logic Based On The Resolution Principle," *Journal of ACM*, Vol.12, No.1, pp.23-41.

Rosenfeld, A. (1969) *Picture Processing by Computer*, Academic Press, New York.

Roussel, P. (1975) "Prolog, Manual de reference et d'Utilisation, " Groupe d'Intelligence Artificielle, Faculte des Sciences de Luminy, Marseilles, France.

Saalfeld, Alan. (1987) "Stability of Map Topology and Robustness of Map Geometry," *Proceedings*, Eighth International Symposium on Computer-Assisted Cartography, (AUTO-CARTO 8), pp.78-86, Baltimore, Maryland.

Shamos, Michael I. and D. Hoey. (1975) "Closest-Point Problems," *Proceedings*, 16th IEEE Symposium on Foundations of Computer Science, pp.151-162.

Shamos, Michael I. and D. Hoey. (1976) "Geometric Intersection Problems," *Proceedings*, 17th IEEE Symposium on Foundations of Computer Science, pp.208-215. Shamos, Michael I. (1979) "Computational Geometry," *Ph.D. Thesis*, Yale University, New Haven, Connecticut.

Shelberg, M., H. Moellering and N.S. Lam. (1982) "Measuring The Fractal Dimensions of Empirical Cartographic Curves," *Proceedings*, Fifth International Symposium on Automated Cartography, (AUTO-CARTO 5), pp.481-490.

Sinton, David, et al. (1972) "Spatial Data Analysis Techniques," *Geographical Data Handling*, Tomlinson (ed), Vol.2, IGU Commission on Geographical Data Sensing and Processing, Ottawa, Ontario.

Steele, Guy L. Jr. (1984) *Common Lisp*, Digital Press, Burlington, Massachusetts.

Sterling, Leon and Ehud Shapiro. (1986) *The Art of Prolog: Advanced Programming Techniques*, MIT Press, Cambridge, Massachusetts.

Sun Microsystems. (1986a) *Commands Reference Manual*, Revision G of 17, Part No: 800-1295-02, pp.18-19 and 82-83, Sun Microsystems, Inc., Mountain View, California.

Sun Microsystems. (1986b) *UNIX Interface Reference Manual*, Revision A of 17, Part No: 800-1341-02, pp.270-271, Sun Microsystems, Inc., Mountain View, California.

Sutherland, I.E., R.F. Sproull and R.A. Schumacker. (1974) "A Characterization of Ten Hidden-Surface Algortihms," *ACM Computing Surveys*, Vol.6, No.1, pp.1-55.

Swinson, Peter S.G. (1982) "Logic Programming: A Computing Tool for the Architect Of The Future," *Computer Aided Design*, Vol.14, No.2, pp.97-104.

Swinson, Peter S.G. (1983) "Prolog: A Prelude to a New Generation of CAAD," *Computer Aided Design*, Vol.15, No.6, pp.335-343.

Tilove, Robert B. (1980) "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *Transactions on Computers*, C-29, 10, pp.874-883.

Tomlinson, R.F., H.W. Calkins and D.F. Marble. (1976) *Computer Handling of Geographical Data*, The Unesco Press.

Vielle, Laurent. (1986) "Recursion in Deductive Databases: DedGin, a Recursive Query Evaluator," *Technical Report* TR-KB-14, European Computer-Industry Research Center, Munich, West Germany.

Warren, David. (1979) "Prolog on the DECSystem-10," *Expert Systems In The Micro Electronics Age*, Michie (ed), Edinburgh University Press, Edinburgh, United Kingdoms.

White, Denis. (1977) "A New Method Of Polygon Overlay," *Proceedings*, Advanced Study Symposium on Topological Data Structures for Geographic Information Systems, Vol.1, Harvard University, Cambridge, Massachusetts.

White, Marvin. (1979) "A Survey of The Mathematics of Maps," *Proceedings*, Fourth International Symposium on Automated Cartography, (AUTO-CARTO 4), Vol.I, pp.82-96, Reston, Virginia.

White, Marvin. (1983) "Tribulations of Automated Cartography and How Mathematics Helps," *Proceedings,* Sixth International Symposium on Automated Cartography, (AUTO-CARTO 6), Vol.I, pp.408-418, Ottawa, Ontario.

Wu, Peter Y.F. (1986) "Two Arithmetic Packages In Prolog: Infinite Precision Fixed Point and Exact Rational Numbers," *Technical Report* TR-082, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York.

Appendix 1: **Rational Arithmetic Package**

I append to the thesis here the Prolog source code which installs rational arithmetic. The reader is referred to (Wu 86) which is a technical report of the work on the package. Chapter 4, section 4.3 also explains many of the underlying ideas. The version here is the latest version as of the writing of this thesis. The software package should be very transportable, and is free for sharing and further distribution.

The appendix includes the BIG package - multiple precision integers, the XQ package - exact rational numbers, and the test program to compute π.

```
% big, prolog/num
% Big Number Package for infinite precision integers
% Peter YF Wu - 14Jan87 %

% X are <big_number_expression>.
% Define infix operator "are" :-
% ...to evaluate an arithmetic expression of "big numbers".

:-op(700,xfx,are).

% Operator "are" uses an asserted fact to return the result in
% order fail through big(_,_); this will release stack space
% acquired for the evaluation of the BIG number expression.

_ are Y :- big(Y,Z),
           asserta(are_evaluated(Z)),
           fail.

X are _ :- retract(are_evaluated(Z)),!,
           X=Z,!.

% A Big Number is an infinite precision integer:
% A Big Number is represented as a list of integers, each of
% which is within the range 0 to 10000 - the big_term_size.
% The list begins with the least significant term and ends
% with the most significant (non-zero) term.  A legitimate
% big number should have all preceding zeroes trimmed. Hence...
%
%                 Zero is represented as [];
%                  One is represented as [1];
%          Seventy Two is represented as [72];
%                10001 is represented as [1,1];
%               218077 is represented as [8077,21] and so on.
%
% Negative big numbers are represented with all negative terms
% except those which are zeroes.  Hence...
%
%            Minus One is represented as [-1];
%   Minus One Thousand is represented as [-1000];
%               -10001 is represented as [-1,-1];
%              -200041 is represented as [-41,-20] etc...

% Define big_term_size(BIG) as the limit of the absolute value
% of each term in a big number list... (in this case: 10000)

big_term_size(10000).

% float_big(BIG,Floating_Point).
% Convert BIG number into regular floating point value.

float_big([],0).
float_big([A|B],F):-
   float_big(B,E),
   big_term_size(D),
   F is A+D*E,
   !.

% Define infix operators "<<",">>","=<<",">>=" for BIG
% numbers comparison. Operands are first evaluated.
```

```
:-op(700,xfx,'<<').
:-op(700,xfx,'>>').

X << Y :- big(X,A),big(Y,B),
          compare_big(A,B,S),
          asserta(compare_evaluated(S)),
          fail.
X << Y :- retract(compare_evaluated(S)),!,
          S='<'.

X >> Y :- big(X,A),big(Y,B),
          compare_big(A,B,S),
          asserta(compare_evaluated(S)),
          fail.
X >> Y :- retract(compare_evaluated(S)),!,
          S='>'.

:-op(700,xfx,'=<<').
:-op(700,xfx,'>>=').

X =<< Y :- big(X,A),big(Y,B),
           compare_big(A,B,S),
           asserta(compare_evaluated(S)),
           fail.
X =<< Y :- retract(compare_evaluated(S)),!,
           (S='=';S='<').

X >>= Y :- big(X,A),big(Y,B),
           compare_big(A,B,S),
           asserta(compare_evaluated(S)),
           fail.
X >>= Y :- retract(compare_evaluated(S)),!,
           (S='=';S='>').

% compare_big(A,B,S).
% Compare two big numbers A and B.  Result is returned in "S",
% where S can be "<","=",">".  The signed lengths of A and B are
% compared first.  If the signed lengths are equal, then compare
% the big numbers term by term (most significant first).

compare_big(A,B,S):-
   length_big(A,A1),
   length_big(B,B1),
   compare_big_scalars(A1,B1,S1),!,
   S=S1.
compare_big(A,B,S):-
   compare_big_by_terms(A,B,S1),!,
   S=S1.
compare_big(_,_,'=').

% length_big(B,Signed_Length).
% Signed_Length is the number of terms, positive or negative, in
% big number B, which must be a well-formed big number.

length_big(B,L):-
   sign_of_big_number(B,S),
   length(B,L0),
   L is S*L0.

sign_of_big_number([0|B],S):-!,sign_of_big_number(B,S).
sign_of_big_number([B1|_],1):-B1>0,!.
sign_of_big_number([_|_],S):-S is -1,!.
sign_of_big_number([],0).

% compare_big_scalars(A1,B1,S).
% S returns "<" or ">" according to integers A1 and B1;
% fails if A1 and B1 are equal.

compare_big_scalars(A1,B1,'<'):-A1<B1,!.
compare_big_scalars(A1,B1,'>'):-A1>B1,!.

% compare_big_by_terms(A,B,S).
% Big numbers A and B have equal signed length. Compare each term
% in A and B; most significant term first at end of list.  Terms are
% compared using "compare_big_scalars/3"; fails if A and B equal.

compare_big_by_terms([_|A],[_|B],S):-
   compare_big_by_terms(A,B,S),
   !.
compare_big_by_terms([A1|_],[B1|_],S):-
   compare_big_scalars(A1,B1,S).

% big(+X,Y).
% plus sign on big number...

big(+X,Y):-
   !,
   big(X,Y).

% big(X+Y,Z).
% addition of big numbers...

big(X+Y,Z):-
   !,
   big(X,A),
   big(Y,B),
   plus_big(A,B,C),
   validate_big(C,Z).

plus_big([],Z,Z):-!.
plus_big(Z,[],Z):-!.
plus_big([X1|X],[Y1|Y],[Z1|Z]):-
   Z1 is X1+Y1,
   plus_big(X,Y,Z),
   !.

% big(-X,Y).
% minus big number...

big(-X,Y):-
   !,
   big(X,A),
   minus_big(A,Y).

minus_big([],[]).
minus_big([X1|X],[Y1|Y]):-
   Y1 is -X1,
   minus_big(X,Y),
   !.
```

```
% big(X-Y,Z).
% subtraction of big numbers...

big(X-Y,Z):-
   !,
   big(X,A),
   big(Y,B),
   minus_big(B,C),
   plus_big(A,C,D),
   validate_big(D,Z).

% big(abs(X),Y).
% absolute value of big numbers...

big(abs(X),Y):-
   !,
   big(X,A),
   sign_of_big_number(A,S),
   scale_big(S,A,Y).

% big(X*Y,Z).
% multiplication of big numbers...

big(X*Y,Z):-
   !,
   big(X,A),
   big(Y,B),
   times_big(A,B,Z).

times_big([],_,[]).
times_big([X1|X],Y,Z):-
   scale_big(X1,Y,A),
   times_big(X,Y,B),
   plus_big(A,[0|B],C),
   validate_big(C,Z),
   !.

scale_big(_,[],[]).
scale_big(A1,[X1|X],[Y1|Y]):-
   Y1 is A1*X1,
   scale_big(A1,X,Y),
   !.

% big(X//Y,Z).
% integer division of big numbers...
% (same as X/Y).

big(X//Y,Z):-!,big(X/Y,Z).

% big(X/Y,Z).
% division of big numbers...

big(X/Y,Z):-
   !,
   big(X,A),
   big(Y,B),
   decompose_big(A,S1,C),
   decompose_big(B,S2,D),
   divide_big(C,D,Q,_),
   ((S1=S2,Q=Z);minus_big(Q,Z)),!.
```

```
% big(X mod Y,Z).
% modulo arithmetic of big numbers...

big(X mod Y,Z) :-
    !,
    big(X,A),
    big(Y,B),
    decompose_big(A,S,C),
    decompose_big(B,_,D),
    divide_big(C,D,_,R),
    ((S='+',R=Z);minus_big(R,Z)),!.

% decompose_big(X,sign,value).
% Decompose big number into sign and magnitude.

decompose_big(X,'+',X) :- (X=[];sign_of_big_number(X,1)),!.
decompose_big(X,'-',Y) :-minus_big(X,Y).

% compose_big(sign,value,X).
% Compose big number from sign and magnitude.

compose_big('+',X,X).
compose_big('-',X,Y) :-minus_big(X,Y).

% divide_big(X,Y,Q,R).
% big number division...
% quotient Q = X/Y; remainder R = X mod Y.

divide_big(_,[],_,_) :-
    nl,write('! Arithmetic error: Division by Zero.'),
    nl,!,fail.

divide_big([],_,[],[]) :-!.
divide_big(X,[1],X,[]) :-!.
divide_big(X,[Y1],Q,R) :-
    divide_by_1_term(X,Y1,Q,R1),
    form_big(R1,[],R),
    !.
divide_big(X,Y,[],X) :-
    compare_big(X,Y,'<'),
    !.
divide_big([X1|X],Y,Q,R) :-
    divide_big(X,Y,A,B),
    form_big(X1,B,C),
    divide_to_1_term(C,Y,Q1,R),
    form_big(Q1,A,Q),
    !.

% divide_by_1_term(X,Y1,Q,R1).
% Division with 1-term divisor: Y1
% quotient Q=X/Y1; 1-term remainder R1 = X mod Y1.

divide_by_1_term([],_,[],0).
divide_by_1_term([X1|X],Y1,Q,R1) :-
    divide_by_1_term(X,Y1,A,B1),
    big_term_size(BIG),!,
    Z1 is X1+B1*BIG,
    Q1 is Z1//Y1,
    R1 is Z1 mod Y1,
```

```
    form_big(Q1,A,Q),
    !.

% divide_to_1_term(X,Y,Q1,R).
% Division to yield 1-term quotient: Q1
% 1-term quotient Q1=X/Y; remainder R = X mod Y.

divide_to_1_term(X,Y,0,X) :-
    compare_big(X,Y,'<'),
    !.
divide_to_1_term(X,Y,Q1,R) :-
    scale_big(2,Y,A),
    organize_big('+',A,B,0),
    divide_to_1_term(X,B,P2,C),
    divide_to_0_or_1(C,Y,P1,R),
    Q1 is P1+2*P2,
    !.

% divide_to_0_or_1(X,Y,Q1,R).
% Division X/Y to give a quotient of either 0 or 1.
% called from divide_to_1_term (0=<X<2Y known already)

divide_to_0_or_1(X,Y,1,R) :-
    minus_big(Y,A),
    plus_big(X,A,B),
    sign_big(B,'+'),!,
    organize_big('+',B,C,0),
    trim_big(C,R),
    !.
divide_to_0_or_1(X,Y,0,X).

% big(X^Y,Z).
% big number exponentiation...

big(X^Y,Z) :-
    !,
    big(X,A),
    big(Y,B),
    decompose_big(B,S,C),
    expt_big(A,S,C,Z),
    !.

expt_big([],_,[],_) :-
    nl,write('! Arithmetic error: Zero raised to power Zero.'),
    nl,!,fail.

expt_big(_,_,[],[1]) :-!.
expt_big(_,'-',[_|_],[]).
expt_big(X,'+',P,Y) :-
    decompose_big(X,S,V),
    up_big(V,P,V,[1],Z,_),
    expt_sgn(S,P,T),
    compose_big(T,Z,Y).

expt_sgn('-',[P1|P],'-') :-1 is P1 mod 2 .
expt_sgn(_,_,'+').

up_big(X,P,Y,Q,Y,R) :-
    minus_big(Q,A),
    plus_big(P,A,B),
   sign_big(B,'+'),
   organize_big('+',B,C,0),
   trim_big(C,R),
   compare_big(Q,R,'>'),
   !.
up_big(X,P,Y,Q,Z,R):-
   plus_big(Q,Q,A),
   organize_big('+',A,B,0),
   times_big(Y,Y,C),
   up_big(X,P,C,A,D,E),
   down_big(Y,Q,D,E,Z,R),
   !.

down_big(X,P,Y,Q,Y,Q):-
   compare_big(P,Q,'>'),
   !.
down_big(X,P,Y,Q,Z,R):-
   minus_big(P,A),
   plus_big(Q,A,B),
   organize_big('+',B,C,0),
   trim_big(C,R),
   times_big(X,Y,Z),
   !.

% big(X,Y).
% relates big number "Y" with...
% ill-formed big number "X", including integer "X".

big(0,[]):-!.

big(X,Y):-
   integer(X),!,
   sign_big([X],S),
   organize_big(S,[X],Y,0).

big(X,Y):-validate_big(X,Y),!.

big(X,_):-nl,
   write('! Arithmetic error: illegal term "'),
   write(X),write('" in expression.'),nl,
   !,fail.

% trim_big(X,Y):-trim leading zero's...
% Big Number "X" may have leading Zero's at end of list;
% "Y" has all leading Zero terms removed.

trim_big([],[]).
trim_big([H|T],T1):-trim_big(T,T0),form_big(H,T0,T1).

% form_big(X1,X,Y).
% big number Y = [X1|X] where X1 is integer and X big number...
% takes care of special case when X1=0 and X=[].

form_big(0,[],[]):-!.
form_big(H,T,[H|T]).

% reduce_big(X,V1).
% Reduce big number X to integer value V1...
% X can have at most two terms (otherwise fails!).
```

```
reduce_big([],0).
reduce_big([X1],X1).
reduce_big([X1,X2],V1):-
   big_term_size(BIG),!,
   V1 is X1+X2*BIG.

% sign_big(X,S):-get sign (of big number)...
% Big Number "X" may be illegitimate, with non-uniform terms, etc.
% ...must find absolute value sign before organize_big.
% S returns "+" or "-" (0 is positive).

sign_big([],'+'):-!.
sign_big(X,S):-sign_of_big(X,'+',S),!.

sign_of_big([0],S,S):-!.
sign_of_big([X1],_,'-'):-X1<0,!.
sign_of_big([X1],_,'+'):-X1>0,!.
sign_of_big([X1,X2|X],S0,S):-
   big_term_size(BIG),!,
   X0 is X1 mod BIG,
   X3 is X1//BIG+X2,
   sign_of_big([X0],S0,S1),
   sign_of_big([X3|X],S1,S),
   !.

% validate_big(X,Y):-validate a big number
% ...to organize into a legitimate big number.

validate_big(X,Y):-
   sign_big(X,S),
   organize_big(S,X,A,0),
   trim_big(A,Y),
   !.

% organize_big(S,X,Y,C):-organize big number...
% S "+"/"-" is the intended sign of big number X.
% X may have illegitimate terms (wrong sign or term value size)...
% C is the integer carry over.
% Y is returned as re-organized big number.

organize_big(_,[],[],0):-!.
organize_big(S,[],Y,C):-organize_big(S,[C],Y,0),!.
organize_big(S,[X1|X],[Y1|Y],C1):-
   X0 is X1+C1,
   carry_big(S,X0,Y1,C),
   organize_big(S,X,Y,C),
   !.

% carry_big(S,A,B,C):-compute carry out.
% S "+"/"-" is the intended sign of the big number with carry out;
% A is an integer value to compute a valid term B and carry out C...

carry_big(_,0,0,0):-!.
carry_big(S,A0,B1,C1):-
   big_term_size(BIG),!,
   B0 is A0 mod BIG,
   C0 is A0//BIG,
   carry_out(S,B0,C0,B1,C1).

carry_out('+',B0,C0,B1,C1):-
```

```
   B0<0,!,
   big_term_size(BIG),!,
   B1 is B0+BIG,
   C1 is C0-1 .
carry_out('-',B0,C0,B1,C1):-
   B0>0,!,
   big_term_size(BIG),!,
   B1 is B0-BIG,
   C1 is C0+1 .
carry_out(_,B1,C1,B1,C1).

% pb(X).
% print big number "X".

pb([A]):-
   write(A),
   !.
pb([A|B]):-
   pb(B),
   (A<0,A1 is -A;A=A1),!,
   big_term_size(BIG),
   pb(A1,BIG),
   !.
pb([]):-write(0).

pb(0,1):-!.
pb(A,D):-
   D1 is D/10,
   X is A//D1,
   write(X),
   A1 is A-X*D1,
   pb(A1,D1).
```

```
% xq. prolog/num
% Exact Rational Number Arithmetic Package
% This package is supported by the Big Number Arithmetic Package
% which implements infinite precision integer arithmetic using lists.
% Peter YF Wu - 14Jan87 %

% X isx <big_number_expression>.
% Define "isx" as an infix operator in place of "is"...
% ...results in the evaluation of big number arithmetic
%    up to single division: i.e., exact rational number.

:- op(700,xfx,isx).

% Operator "isx" uses an asserted fact to return the result in
% order fail through xq(_,_); this will release stack space
% acquired for the evaluation of the XQ arithmetic expression.

_ isx Y :- xq(Y,Q),
           compress_xq(Q,Z),
           asserta(xq_evaluated(Z)),
           fail.

X isx _ :- retract(xq_evaluated(Z)),!,
           X=Z,!.

% float_xq(XQ,Floating_Point).
% Convert XQ number into regular floating point value.

float_xq(X,F):-
   decompose_xq(X,P/Q),
   float_big(P,N),float_big(Q,D),
   F is N/D,
   !.

% Exact Rational Number is represented as a big number
% expression evaluated up to the last division...

% Addition (and plus sign)...

xq(+U,W)  :- !,xq(U,W).

xq(U+V,W)  :- !,xq(U,X),xq(V,Y),
              decompose_xq(X,N1/D1),decompose_xq(Y,N2/D2),
              N are N1*D2+N2*D1, D are D1*D2,
              compose_xq(N/D,W),!.

% Subtraction (and minus sign)...

xq(-U,W)  :- !,xq(U,V),
             decompose_xq(V,N/D),
             N1 are -N,
             compose_xq(N1/D,W),!.

xq(U-V,W)  :- !,xq(U,X),xq(V,Y),
              decompose_xq(X,N1/D1),decompose_xq(Y,N2/D2),
              N are N1*D2-N2*D1, D are D1*D2,
              compose_xq(N/D,W),!.

% Absolute Value and Integral Part...
```

```
xq(abs(U),W)  :- !,xq(U,V),
                 decompose_xq(V,N/D),
                 N1 are abs(N),
                 compose_xq(N1/D,W),!.

xq(fix(U),W)  :- !,xq(U,V),W are V,!.

% Multiplication...

xq(U*V,W)  :- !,xq(U,X),xq(V,Y),
              decompose_xq(X,N1/D1),decompose_xq(Y,N2/D2),
              N are N1*N2, D are D1*D2,
              compose_xq(N/D,W),!.

% Division...
% evaluate up to the last division.

xq(U/V,W)  :- U=[_|_],V=[_|_],!,
              big(V,Y),decompose_big(Y,S,D),
              compose_big(S,[1],S1),big(S1*U,N),
              (not(D=[]);err_xq_zero_den),!,
              compose_xq(N/D,W),!.

xq(U/V,W)  :- !,xq(U,X),xq(V,Y),
              decompose_xq(X,N1/D1),decompose_xq(Y,N2/D2),
              decompose_big(N2,S2,M2),compose_big(S2,[1],S),
              N are S*(N1*D2), D are D1*M2,
              (not(D=[]);err_xq_zero_den),!,
              compose_xq(N/D,W),!.

err_xq_zero_den :- nl,
   write('! XQ arithmetic error: Zero Denominator.'),nl,
   !,fail.

% Integer Division...
% not supported in XQ arithmetic.

xq(U // V,W) :- nl,
   write('! XQ arithmetic error: Integer Divide "//" is illegal.'),nl,
   !,fail.

% Modulo...
% not supported in XQ arithmetic.

xq(U mod V,W) :- nl,
   write('! XQ arithmetic error: Modulo "mod" is illegal.'),nl,
   !,fail.

% Exponentiation...
% only to integral power represented as big number.

xq(U^V,W)  :- !,xq(U,X),xq(V,Y),
              decompose_xq(X,N1/D1),
              (decompose_big(Y,S,E);err_xq_int_power),!,
              N are N1^E, D are D1^E,
              form_xq_exponential(S,N,D,W),!.

err_xq_int_power :- nl,
   write('! XQ arithmetic error: Non-Integral exponent.'),nl,
   !,fail.
```

```
form_xq_exponential('+',N,D,W) :- compose_xq(N/D,W).
form_xq_exponential('-',N,D,W) :-
   decompose_big(N,S,D1),
   compose_big(S,[1],S1),
   N1 are S1*D,
   compose_xq(N1/D1,W).

% Upward compatibility: BIG number is an XQ number.

xq(U,V) :- big(U,V).

% compress_xq(X,Y).
% Compress the fractional XQ expression "X" by fractorization into
% the GCD of numerator and denominator; "Y" returns compressed.

compress_xq(N/D,Y) :-
   !,
   decompose_big(N,_,M),
   gcd_big(M,D,G),
   P are N/G,
   Q are D/G,
   compose_xq(P/Q,Y),
   !.
compress_xq(X,X).

% Decomposition and Composition.
% into/from numerator and denominator...

decompose_xq(X/Y,X/Y):-!.
decompose_xq(X,X/[1]).

compose_xq(X/[1],X):-!.
compose_xq(X,X).

% gcd_big(A,B,G).
% Compute greatest common divisor G of big numbers A and B.

gcd_big(A,[],A):-!.
gcd_big(A,B,G):-
   C are A mod B,
   gcd_big(B,C,G).

% Re-define infix operators "<<",">>","=<<",">>=" for XQ
% numerical comparison. Operands are first evaluated.

:-abolish('<<',2).
:-abolish('>>',2).

X << Y :- xq(X,A),xq(Y,B),
          compare_xq(A,B,S),
          asserta(compare_evaluated(S)),
          fail.
X << Y :- retract(compare_evaluated(S)),!,
          S='<'.

X >> Y :- xq(X,A),xq(Y,B),
          compare_xq(A,B,S),
          asserta(compare_evaluated(S)),
          fail.
```

```
X >> Y :- retract(compare_evaluated(S)),!,
          S='>'.

:-abolish('=<<',2).
:-abolish('>>=',2).

X =<< Y :- xq(X,A),xq(Y,B),
           compare_xq(A,B,S),
           asserta(compare_evaluated(S)),
           fail.
X =<< Y :- retract(compare_evaluated(S)),!,
           (S='=';S='<').

X >>= Y :- xq(X,A),xq(Y,B),
           compare_xq(A,B,S),
           asserta(compare_evaluated(S)),
           fail.
X >>= Y :- retract(compare_evaluated(S)),!,
           (S='=';S='>').

% compare_xq(X,Y,S).
% Compare two XQ numbers X and Y.  Result is returned in S which
% can be "<","=",">". Compare_xq is supported by compare_big.

compare_xq(X,Y,S) :-
   decompose_xq(X,P1/Q1),
   decompose_xq(Y,P2/Q2),
   R1 are P1*Q2, R2 are P2*Q1,
   compare_big(R1,R2,S).

% pxq(Q).
% Print exact rational number Q...

pxq(N/D):-!,pb(N),write('/'),pb(D).
pxq(Q):-pb(Q).
```

```
% pi, prolog/num
% Compute pi up to any desired precision level...
% this is supported by the exact rational numbers package
% as well as the big number arithmetic package (XQ and BIG).
% Peter Wu - 1/25/85

:-consult(big).      % arithmetic package: BIG
:-consult(xq).       % arithmetic package: XQ

% Pi is computed using the following infinite series:
%
%               2   2   4   4   6   6   8   8
%    Pi = 2 * - * - * - * - * - * - * - * - * ...
%               1   3   3   5   5   7   7   9
%
% or in product series...
%
%    Pi = 2 * PRODUCT{(r + r mod 2) / (r + 1 - r mod 2)}
%
% where r=1,2,3,...
%
% This program asserts and updates a fact:  pi(Index,Pi).
% where "Index" is the number of terms computed along the
% infinite series, and "Pi" is the value of pi thus far.
% "Index" is expressed as a big number, and "Pi" as exact
% rational number - a fraction with both numerator and
% denominator as big numbers...

pi([],[2]).    % preset value: Pi = 2

step :-
   pi(R,P),
   R1 are R+[1],
   R2 are R1 mod [2],
   P1 isx P*((R1+R2)/(R1-R2+[1])),
   retract(pi(R,P)),
   asserta(pi(R1,P1)),
   pb(R1),write(': '),pxq(P1),nl,!.

go :- repeat,step,fail.
```

Appendix 2: **OVER4 Source Listing**

In this appendix, we include the Prolog source code of all 12 programs of OVER4, the polygon overlay system in Prolog. I have documented the function of these programs in Chapter 7, section 7.1. These programs are quite readable, and I have put in sufficient comment in the listing. I hope this can be an example of a higher level approach in programming toward geometry problems.

OVER4 consists of 12 programs in 4 groups: XSECT, LINK, OVER and CONTR. XSECT consists of *dform*, *gstat*, *xgrid*, *xsect*, *xconn*; LINK consists of *calci*, *sorti*, *cpoly*, *bound*; OVER consists of *xover* and *pcont*, and CONTR consists of only *jconn*.

```
% dform, prolog/over4
% Input data format: from chains to b,v,e...
% Peter Wu - 3/10/87

% *dform* organizes the input chains data into three sets:
% b, v, and e, for boundaries, vertices, and edges.  Input
% chain and output data sets have the following formats:
%
%   chain(C#,Map,N1,N2,[[X,Y],...],P1,P2).
%
%   b(B#,P1,P2).          Boundary "B" and left/right polygons P1,P2;
%   v(V#,Type,[X,Y]).     Vertex "V" at (X,Y), Type=n:node,v:vertex;
%   e(E#,[V1,V2]).        Edge "E" from V1 to V2.
%
% The id#'s are arbitrary assigned, but generated accordingly...
%
% boundary "B#"        [1]: Map id {1,2,...}
%                    [2-4]: Chain #
%                    [5-7]: (not used)
%
% node/vertex "V#"     [1]: Map id {1,2,...}
%                      [2]: Type (0:node,1:vertex).
%                    [3-7]: vertex # (arbitrary)
%
% edge segment "E#"    [1]: Map id {1,2,...}
%                    [2-4]: Chain #
%                    [5-7]: Edge # within chain.
%
% Note: vertex may turn into node in the intersection process.
% For each input map, we also keep these two parameters:
%
%   root(Map,id)       Map identifier;    eg: root(usa,1000000).
%   vroot(Map,count)   Vertices counter;  eg: vroot(usa,1100000).

dform:-
   T is cputime,
   assert(dform_timer(T)),
   fail.

dform:-
   retract(chain(C,MAP,N1,N2,L,P1,P2)),
   separate_n_and_v(L,V1,V2,VL),
   dcl_node(MAP,N1,V1,VN1),
   dcl_node(MAP,N2,V2,VN2),
   dcl_v_and_e(C,MAP,VN1,VN2,VL,1),
   dcl_boundary(C,MAP,P1,P2),
   fail.

dform:-
   retract(dform_timer(T0)),
   T is cputime-T0,
   write('CPU used: '),write(T),write('s (dform).'),
   nl.

% separate_n_and_v(L,V1,V2,VL).
% Separate polyline list L into nodes (V1,V2) and list of
% the inerior vertices VL...

separate_n_and_v(L,V1,V2,VL):-append([V1|VL],[V2],L),!.
```

```
append([],L,L).
append([H|L1],L2,[H|L3]):-append(L1,L2,L3).

% dcl_node(Map,Node,[X,Y],N#).
% Declare Node in Map at (X,Y), as N#...

dcl_node(MAP,N,V,VN):-
   root(MAP,MROOT),
   VN is MROOT+N,
   ( v(VN,n,V); asserta(v(VN,n,V))),!.

% dcl_v_and_e(C#,Map,N1,N2,VL,E#).
% Declare the vertices and the edges along the chain C#...

dcl_v_and_e(C,MAP,VN1,VN2,[],E):-
   dcl_edge(C,MAP,[VN1,VN2],E),
   !.

dcl_v_and_e(C,MAP,VN1,VN2,[V|VL],E):-
   dcl_vertex(MAP,V,VN),
   E1 is E+1,
   dcl_v_and_e(C,MAP,VN,VN2,VL,E1),
   dcl_edge(C,MAP,[VN1,VN],E),
   !.

% dcl_edge(C#,Map,[V1,V2],E#).
% Declare Edge from V1 to V2, as E#...

dcl_edge(C,MAP,EDGE,E):-
   root(MAP,MROOT),
   ID is MROOT+C*1000+E,
   asserta(e(ID,EDGE)),
   !.

% dcl_vertex(Map,[X,Y],V#).
% Declare Vertex in Map at (X,Y), as V#...

dcl_vertex(MAP,V,VN):-
   retract(vroot(MAP,VROOT)),
   VN is VROOT+1,
   asserta(vroot(MAP,VN)),
   asserta(v(VN,v,V)),
   !.

% dcl_boundary(C#,Map,P1,P2).
% Declare chain C# as boundary B# between polygons P1,P2.

dcl_boundary(C,MAP,P1,P2):-
   root(MAP,MROOT),
   B is MROOT+C*1000,
   asserta(b(B,P1,P2)),
   !.
```

```
% gstat, prolog/over4
% Compute statistics and grid parameters.
% Peter Wu - 3/11/87

% 'gstat' has three parts: 'vstat', 'cstat', and 'gstat'.
% Each computes statistical measures from the input maps,
% and sets up the parameters for the adaptive grid.  The
% input data sets include...
%
% b(B#,P1,P2).   % chain B#: left, right polygons P1,P2
% e(E#,V1,V2).   % edge E#: from vertex V1 to vertex V2
% v(V#,T,[x,y]). % vertex V#: at (x,y) T= n or v...

% 'vstat' computes statistical measures concerning the nodes
% and vertices.  'vstat' needs "v" for input; counts total
% number of nodes and vertices...
%
%  no_of_nodes(NNS).       % no. of nodes
%  no_of_vertices(NTS).    % no. of vertices (nodes included)
%
% 'vstat' also determines the extents of the coordinates.
% Calculation is done using XQ arithmetic to preserve accuracy.
% The extents of the coordinates are kept as...
%
%   left(XORG). right(XLIM). bottom(YORG). top(YLIM).
%

vstat:-T is cputime,
       assert(vstat_timer(T)),
       fail.

vstat:-bagof(VN,V^v(VN,n,V),NS),
       length(NS,NNS),
       write('Total # of nodes: '),write(NNS),nl,
       assert(no_of_nodes(NNS)),
       fail.

vstat:-bagof(VN,V^v(VN,v,V),VS),
       length(VS,NVS),no_of_nodes(NNS),
       NTS is NNS+NVS,
       write('Total # of nodes and vertices: '),write(NTS),nl,
       assert(no_of_vertices(NTS)),
       fail.

vstat:-init_extents,fail.

vstat:-v(_,_,[X,Y]),
       (float_xq(X,XF),extend_left_right(X,XF);
        float_xq(Y,YF),extend_top_bottom(Y,YF)),
        fail.

vstat:-retract(top(YLIM,Y)),retract(right(XLIM,X)),
       write('Top-Right  (X,Y) co-ordinates: '),write([X,Y]),nl,
       assert(top(YLIM)),assert(right(XLIM)),
       fail.

vstat:-retract(bottom(YORG,Y)),retract(left(XORG,X)),
       write('Bottom-Left  (X,Y) co-ordinates: '),write([X,Y]),nl,
       assert(bottom(YORG)),assert(left(XORG)),
       fail.
```

```
vstat:-retract(vstat_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (vstat).'),nl.

% init_extents.
% Initialize the statistical parameteres: left/right, bottom/top.

init_extents:-v(_,_,[X,Y]),
              float_xq(X,XF),
              assert(left(X,XF)),
              assert(right(X,XF)),
              float_xq(Y,YF),
              assert(bottom(Y,YF)),
              assert(top(Y,YF)),!.

% extend_left_right.
% Locate X-Coordinate to update left/right values.

extend_left_right(X,XF):-
   left(XMIN,XMF),
   XF < XMF,!,
   retract(left(XMIN,XMF)),
   asserta(left(X,XF)),!.

extend_left_right(X,XF):-
   right(XMAX,XMF),
   XF > XMF,!,
   retract(right(XMAX,XMF)),
   asserta(right(X,XF)),!.

% extend_top_bottom.
% Locate Y-Coordinate to update top/bottom values.

extend_top_bottom(Y,YF):-
   top(YMAX,YMF),
   YF > YMF,!,
   retract(top(YMAX,YMF)),
   asserta(top(Y,YF)),!.

extend_top_bottom(Y,YF):-
   bottom(YMIN,YMF),
   YF < YMF,!,
   retract(bottom(YMIN,YMF)),
   asserta(bottom(Y,YF)),!.

% 'cstat' computes statistical measures concerning the edges
% and chains.  'cstat' needs all of three "b","e","v" data sets
% and also the root parameters below for input...
%
%  root(Map,MRoot).    % MRoot index for each input Map
%
% 'cstat' counts for each input map the number of edges and
% chains, as well as the total...
%
%  no_of_edges(Map,Edges).    no_of_edges(Total_#_edges).
%  no_of_chains(Map,Chains).  no_of_chains(Total_#_chains).
%
% 'cstat' calculates the total and average edge segment lengths
% in X,Y directions.  Calculations are done in regular floating
% point arithmetic % since absolute accuracy is not required...
%
%  x_total(XT).  y_total(YT).      % total lengths
%  x_length(XL). y_length(YL).     % average length per edge
%

cstat:-T0 is cputime,
       assert(cstat_timer(T0)),
       fail.

% Count number of chains and edges for each input map...

cstat:-root(MAP,MROOT),
       bagof(E,EDGE^(e(E,EDGE),map_root(E,MROOT)),ES),
       bagof(C,P1^P2^(b(C,P1,P2),map_root(C,MROOT)),CS),
       length(ES,NE),
       length(CS,NC),
       write('Map "'),write(MAP),write('": '),
       write(NC),write(' chains, '),write(NE),write(' edges.'),nl,
       asserta(no_of_edges(MAP,NE)),
       asserta(no_of_chains(MAP,NC)),
       fail.

% Identify the id for a specific input map.

map_root(ID,MROOT) :- MROOT is ID-(ID mod 1000000).

% Sum up total number of chains and edges in all maps...

cstat:-bagof(NE,MAP^no_of_edges(MAP,NE),SE),
       sum_list(SE,NSE),
       asserta(no_of_edges(NSE)),
       fail.

cstat:-bagof(NC,MAP^no_of_chains(MAP,NC),SC),
       sum_list(SC,NSC),
       write('Total: '),
       write(NSC),write(' chains, '),
       asserta(no_of_chains(NSC)),
       no_of_edges(NSE),
       write(NSE),write(' edges.'),nl,
       fail.

% sum_list(List,Total).
% Sum up a list of numbers.

sum_list([],0).
sum_list([N|L],SUM):-
   sum_list(L,SUBTOTAL),
   SUM is N+SUBTOTAL.

% Calculate the total edge segment length...

cstat:-init_lengths,
       e(_,[V1,V2]),
       v1(V1,[X1,Y1]),v1(V2,[X2,Y2]),
       ( sum_x_lengths(X1,X2); sum_y_lengths(Y1,Y2)),
       fail.

init_lengths:-
```

```
   assert(x_total(0)),
   assert(y_total(0)),!.

v1(VNAME,V):-v(VNAME,_,V),!.

sum_x_lengths(X1,X2):-
   retract(x_total(X0)),
   diff_xq(X1,X2,XD),
   XT is X0 + XD,
   asserta(x_total(XT)),!.

sum_y_lengths(Y1,Y2):-
   retract(y_total(Y0)),
   diff_xq(Y1,Y2,YD),
   YT is Y0 + YD,
   asserta(y_total(YT)),!.

% diff_xq.
% Difference of two XQ values returned in floating point.

diff_xq(Q1,Q2,DIFF):-
   float_xq(Q1,F1),
   float_xq(Q2,F2),
   (F1<F2,DIFF is F2-F1; DIFF is F1-F2),
   !.

% Calculate average edge segment length...

cstat:-x_total(XT),
       no_of_edges(N),
       XL is XT/N,
       assert(x_length(XL)),
       fail.

cstat:-y_total(YT),
       no_of_edges(N),
       YL is YT/N,
       assert(y_length(YL)),
       fail.

cstat:-x_length(XL),
       y_length(YL),
       write('Average edge (X,Y) length: '),
       write([XL,YL]),nl,
       fail.

cstat:-retract(cstat_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (cstat).'),
       nl.

% *gstat* computes the cell size for the adaptive grid, which
% is casted over all the edge elements to isolate potentially
% intersecting edges into groups of grid cells for subsequent
% intersection tests.  *gstat* takes the results from *vstat*
% and *cstat* to compute the (X,Y) grid cell sizes...
% The input parameters are:
%
%   left(XORG).
%   bottom(YORG).   % origin coordinates...
```

```
%
%   x_length(XL).
%   y_length(YL).   % average edge lengths...
%
% 'gstat' sets up the grid with number of grids and grid cell
% sizes; the output parameters are:
%
%   x_grid(XN).
%   y_grid(YN).    % number of grids...
%
%   x_gsize(XG).
%   y_gsize(YG).   % grid cell sizes...
%

gstat:-T0 is cputime,
       assert(gstat_timer(T0)),
       fail.

% compute X,Y grids:
% number of grids, and grid cell (x,y) sizes.

gstat:-left(XORG),
       right(XLIM),
       x_length(XL),
       XS isx XLIM-XORG,
       float_xq(XS,XSIDE),
       XN is floor(XSIDE/XL),
       XG isx XS/XN,
       assert(x_grid(XN)),
       assert(x_gsize(XG)),
       fail.

gstat:-top(YLIM),
       bottom(YORG),
       y_length(YL),
       YS isx YLIM-YORG,
       float_xq(YS,YSIDE),
       YN is floor(YSIDE/YL),
       YG isx YS/YN,
       assert(y_grid(YN)),
       assert(y_gsize(YG)),
       fail.

% Report the results...

gstat:-x_grid(XN),
       y_grid(YN),
       write('# of (X,Y) Grid cells: '),
       write([XN,YN]),nl,
       fail.

gstat:-x_gsize(XG),
       y_gsize(YG),
       float_xq(XG,FXG),
       float_xq(YG,FYG),
       write('Grid cell (X,Y) size: '),
       write([FXG,FYG]),nl,
       fail.

gstat:-retract(gstat_timer(T0)),
```

```
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (gstat).'),
       nl.
```

```
% xgrid, prolog/over4
% Cast grid over edge segments.
% Peter Wu - 3/10/87

% *xgrid* takes the edge segments and determines the grid cells
% each edge segment occupies.  Each grid entry collects the
% edges covering the grid cell.  *xgrid* takes as input data:
%
%   e(E#, [V1,V2]).        ...edges
%   v(V#,Type, [X,Y]).     ...vertices
%
% For the grid, *xgrid* needs the following parameters:
%
%   x_grid(XN).
%   y_grid(YN).        % number of grids...
%
%   x_gsize(XG).
%   y_gsize(YG).       % grid cell sizes...
%
%   left(XORG).
%   bottom(YORG).      % origin coordinates... (from "parm")
%
% *xgrid* generates a grid entry for each grid collection of
% edges (no entry for empty grid cells).  The grid entry is
% a fact under a functor name "gxxxxx" where xxxxx is the
% grid entry id#:
%
%   g344([E1,E2,...]).     for grid entry# 344.
%
% Hence, *xgrid* may need a large atom space.
% *xgrid* gathers the grid cell collections together as:
%
%   ge(G, [E,..]).
%
% and removes entries which do not involve edges of both input
% maps, since there will be no edge intersections.

xgrid:-T0 is cputime,
       assert(xgrid_timer(T0)),
       fail.

xgrid:-left(XORG),
       bottom(YORG),
       x_grid(XN),
       y_grid(YN),
       x_gsize(XG),
       y_gsize(YG),
       e(E, [V1,V2]),    % process each edge segment...
       v1(V1, [X1,Y1]),
       v1(V2, [X2,Y2]),
       NX1 isx fix((X1-XORG)/XG),
       NX2 isx fix((X2-XORG)/XG),
       NY1 isx fix((Y1-YORG)/YG),
       NY2 isx fix((Y2-YORG)/YG),
       sort_index(NX1,NX2,XN,IX1,IX2),
       sort_index(NY1,NY2,YN,IY1,IY2),
       xxgrid(E,IX1,IX2,IY1,IY2),
       fail.

xgrid:-abolish(v,3), % make room for others.
```

```
       abolish(e,2),
       fail.

xgrid:-[GX]="g",
       g(G), % process each grid entry.
       name(G,NAME),
       name(GNAME, [GX|NAME]),
       GRID =.. [GNAME,ES],
       call(GRID),
       not(same_map(ES,_)),
       asserta(ge(G,ES)),
       fail.

xgrid:-retract(xgrid_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (xgrid).'),
       nl.

% v1(VName,Coordinates).
% Look up for vertex coordinates...

v1(VNAME,COORD)  :- v(VNAME,_,COORD),!.

% sort_index(XQ1,XQ2,Limit,Low,High).
% Convert XQ indexes into floating point
% and determine low and high...

sort_index(P1,P2,N,I1,I2):-
   float_index(P1,N,N1),
   float_index(P2,N,N2),
   sort2(N1,N2,I1,I2),
   !.

% float_index(XQ,Limit,Index).
% Convert XQ index into floating point.
% (Index=0,1,2,...,Limit; taking care of top/right edges)

float_index([],_,0).
float_index([N],N,I):-I is N-1,!.
float_index([I],_,I).

% sort2(N1,N2,Low,High).
% Sort 2 numbers into low and high.

sort2(N1,N2,N1,N2):-N1<N2,!.
sort2(N1,N2,N2,N1).

% xxgrid(E,X1,X2,Y1,Y2).
% Mark edge "E" in the grid cells
% with indexes from X1 to X2 and Y1 to Y2.

xxgrid(E,IX,IX,IY1,IY2):-xygrid(E,IX,IY1,IY2),!.

xxgrid(E,IX,IX2,IY1,IY2):-
   IX1 is IX+1,
   xygrid(E,IX,IY1,IY2),
   xxgrid(E,IX1,IX2,IY1,IY2),
   !.

xygrid(E,IX,IY,IY):-eg_entry(E,IX,IY),!.
xygrid(E,IX,IY,IY2):-
   IY1 is IY+1,
   eg_entry(E,IX,IY),
   xygrid(E,IX,IY1,IY2),
   !.

% eg_entry(E,X,Y).
% Mark edge "E" in grid cell (X,Y).

eg_entry(E,X,Y):-
   y_grid(YN),
   G is X*YN+Y,
   ge_entry(G,E),
   !.

% ge_entry(G,E).
% Enter edge "E" into grid cell G.

ge_entry(G,E):-
   [GX]="g",
   name(G,NAME),
   name(GNAME,[GX|NAME]),
   G0 =.. [GNAME,ES],
   (retract(G0);
    ES=[],asserta(g(G))),
   G1 =.. [GNAME,[E|ES]],
   asserta(G1),
   !.

% same_map(Edge_List,Map).
% Verify that a list of edges is from the same map.

same_map([],_).
same_map([E|ES],MAP):-
   MAP is E//1000000,
   same_map(ES,MAP),
   !.
```

```
% xsect, prolog/over4
% Intersect edge segments in each grid cell.
% Peter Wu - 4/20/87

% `xsect' determines intersection points in the overlay map.
% We collected the potentially intersecting edges within each
% grid cell.  Examine the edges in each grid cell by pairwise
% comparison.  If two edges intersect, update the edges as
% split at the point of intersection and declares the point
% a new node.  `xsect' needs the following input data sets:
%
%   e(E#,[V1,V2]).         % set of edges;
%   v(V#,Type,[X,Y]).      % set of vertices and nodes;
%   ge(G#,[E1,E2,...]).    % set of edges in each grid cell;
%
% In the process, `xsect' generates two interim data sets:
%
%   ok(E1,E2).             % pair E1,E2 has been examined;
%   equation(E#,[A,B,C]).  % equation: Ax + By + C = 0.
%
% `xsect' outputs the updated set of edges and vertices:
%
%   e(E#,[V,...]).         % edges, including split ones;
%   v(V#,Type,[X,Y]).      % vertices, with new nodes;
%   xnode(V#,N#).          % node id V# replaced by N#.

xsect:-T0 is cputime,
       assert(xsect_timer(T0)),
       fail.

xsect:-retract(ge(G,ES)),  % process each grid
       xsect_es(ES),       % intersect the set of edges
       fail.

xsect:-assert(ok(0)),      % initialize ok counter
       ok(E1,E2),          % count how many pairs...
       ok_count,
       fail.

xsect:-retract(ok(NPC)),  % pairs compared...
       write('Edge Segments examined: '),
       write(NPC),write(' pairs.'),nl,
       fail.

xsect:-xroot(N),           % # of new intersection points
       write('# of Intersections: '),write(N),nl,
       fail.

xsect:-retract(xsect_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (xsect).'),
       nl.

% ok_count.
% Count number of pairs compared...

ok_count:-retract(ok(N0)),
          N1 is N0+1,
          assert(ok(N1)),
          !.
```

```
% xsect_es(Set_of_Edges).
% Determine intersections in a set of edges.
% (compare each pair of edges)

xsect_es([]).
xsect_es([E|ES]):-xsect_e_vs_list(E,ES),fail.
xsect_es([_|ES]):-xsect_es(ES),!.

xsect_e_vs_list(_,[]).
xsect_e_vs_list(E1,[E2|_]):-xsect_2_edges(E1,E2),fail.
xsect_e_vs_list(E1,[_|ES]):-xsect_e_vs_list(E1,ES),!.

% xsect_2_edges(E1,E2).
% Compute intersection of Two Edges.

xsect_2_edges(E1,E2):-
  E1//1000000 =:= E2//1000000,  % Both edges same map.
  !.
xsect_2_edges(E1,E2):-
  (ok(E1,E2); ok(E2,E1)),       % Done in another grid.
  !.
xsect_2_edges(E1,E2):-
  asserta(ok(E1,E2)),           % Mark the pair as examined.
  write('... pair: '),write([E1,E2]),nl,
  fail.

xsect_2_edges(E1,E2):-
  e(E1,VL1),                    % determine intersection...
  append([V11|_],[V12],VL1),
  cal_equation(E1,V11,V12,EQT1),
  e(E2,VL2),
  append([V21|_],[V22],VL2),
  point_vs_line(V21,EQT1,D21),
  point_vs_line(V22,EQT1,D22),
  D2 is D21*D22,
  ( D2>0,      % No intersection!
    !;
    D2<0,      % V21,V22 on opposite sides...
    cal_equation(E2,V21,V22,EQT2),
    point_vs_line(V11,EQT2,D11),
    point_vs_line(V12,EQT2,D12),
    D1 is D11*D12,
    ( D1>0,   % No intersection!
      !;
      D1<0,   % V11,V12 also on opposite sides...
      x_2_lines(EQT1,EQT2,T),
      x_split_edge(T,E1,VL1,D11,EQT2),
      x_split_edge(T,E2,VL2,D21,EQT1),
      !;
      D11=0,  % V11 touching E2...
      x_1_point(V11),
      x_split_edge(V11,E2,VL2,D21,EQT1),
      !;
      D12=0,  % V12 touching E2...
      x_1_point(V12),
      x_split_edge(V12,E2,VL2,D21,EQT1),
      !);
    D22=\=0,  % V21 along E1 (may touch)...
    x_sweep_edge(V21,E1,VL1),
    !;
    D21=\=0,  % V22 along E1 (may touch)...
    x_sweep_edge(V22,E1,VL1),
    !;
    % (D21,D22 both 0): E1,E2 colinear.
    x_colinear_edge(VL2,E1,VL1),
    x_colinear_edge(VL1,E2,VL2),
    !).

% GEOMETRY & SUPPORT UTILITIES %

% cal_equation(Edge,V1,V2,Equation).
% Calculate equation of straight line: Ax+By+C=0
% Keep the result for later use...

cal_equation(E,_,_,EQT):-equation(E,EQT),!.
cal_equation(E,V1,V2,[A,B,C]):-
   v(V1,_,[X1,Y1]),
   v(V2,_,[X2,Y2]),
   A isx Y2-Y1,
   B isx X1-X2,
   C isx X2*Y1-X1*Y2,
   asserta(equation(E,[A,B,C])),
   !.

% x_2_lines(Equation1,Equation2,New_Node).
% Determine intersection point between two lines.
% (a new node is therefore born)

x_2_lines([A1,B1,C1],[A2,B2,C2],T):-
   D isx A1*B2-A2*B1,
   X isx (B1*C2-B2*C1)/D,
   Y isx (A2*C1-A1*C2)/D,
   new_node(T),
   asserta(v(T,n,[X,Y])),
   !.

% x_split_edge(Node,Edge,Vlist,D1,Xsect_Equation).
% Edge intersects with Xsect_Equation at Node.  Search for the
% right place in Vlist to insert Node to mark Edge as split at
% the intersection point.  D1 is the value of substituting the
% 1st point in Vlist into Equation...

x_split_edge(T,E,VL,D,EQT):-
   x_split_insert(T,VL,D,EQT,NVL),  % insert new node "T"...
   x_update_edge(E,VL,NVL),         % update edge.
   !.

% x_split_insert(Node,Vlist,D1,Xsect_Equation,New_Vlist).
% Insert Node into Vlist to get New_Vlist for the split edge...

x_split_insert(T,[V1,V2],_,_,[V1,T,V2]):-!.
x_split_insert(T,[V1,T|VL],_,_,[V1,T|VL]):-!.
x_split_insert(T,[V1,V2|VL],D1,EQT,[V1,T,V2|VL]):-
   point_vs_line(V2,EQT,D2),
   D1*D2<0,
   !.
x_split_insert(T,[V1|VL],D1,EQT,[V1|NVL]):-
   x_split_insert(T,VL,D1,EQT,NVL),
   !.
```

```
% point_vs_line.
% Substitute the point into the equation.
% D=Ax+By+C: returns {-1,0,1}...

point_vs_line(V,[A,B,C],D):-
   v(V,_,[X,Y]),
   XQ isx A*X+B*Y+C,
   float_xq(XQ,S),
   ( S>0,D=1;
     S<0,D is -1;
     D=S ),
   !.

% SUPPORT FOR SPECIAL CASES OF TOUCHING AND OVERLAP EDGES %

% x_sweep_edge(Vertex,Edge,Vlist).
% Vertex lies on the extended line of Edge.  Search thru Vlist for
% the place to insert Vertex.  If Vertex is outside of Vlist, there
% is no intersection.  Beware of exact overlap...

x_sweep_edge(T,_,[T|_]):-!.
x_sweep_edge(T,E,[V1|VL]):-
   v(T,_,VT),
   point_vs_point(V1,VT,R1),
   x_sweep_insert(T,[V1|VL],R1,VT,NVL),
   x_update_edge(E,[V1|VL],NVL),
   !.

% x_sweep_insert(Vertex,Vlist,R1,[x,y],New_Vlist).
% Vertex at [x,y] lies on the extended line of Vlist.  If [x,y] is
% is on the Vlist edge segment, insert Vertex to form New_Vlist;
% otherwise there is no intersection.  R1 is the reference value
% of the 1st point on Vlist to [x,y]...  (Beware: R1 may be 0.)

x_sweep_insert(T,[V1|VL],0,_,[V1|VL]):-
   x_2_points(T,V1),      % point vs point intersection...
   !.
x_sweep_insert(_,[V],_,_,[V]):-!.
x_sweep_insert(T,[V1,T|VL],_,_,[V1,T|VL]):-!.
x_sweep_insert(T,[V1,V2|VL],R1,VT,[V1|NVL]):-
   point_vs_point(V2,VT,R2),
   R1*R2>=0,
   x_sweep_insert(T,[V2|VL],R2,VT,NVL),
   !.
x_sweep_insert(T,[V1|VL],_,_,[V1,T|VL]):-x_1_point(T).

% x_colinear_edge(Vs,Edge,Vlist).
% Set of vertices "Vs" is colinear to Edge of Vlist: insert each of
% the vertices of Vs into Vlist...

x_colinear_edge(VS,EDGE,VL):-
   x_colinear_insert(VS,VL,NVL),
   x_update_edge(EDGE,VL,NVL),
   !.

% x_colinear_insert(Vs,Vlist,New_Vlist).
% Use "x_sweep_insert" for set of vertices in Vs: insert each node
% into Vlist if intersecting...
```

```
x_colinear_insert([],VL,VL).
x_colinear_insert([T|VS],[T|VL],TVL):-!,
   x_colinear_insert(VS,[T|VL],TVL),
   !.
x_colinear_insert([T|VS],[V1|VL],TVL):-
   v(T,_,VT),
   point_vs_point(V1,VT,R1),
   x_sweep_insert(T,[V1|VL],R1,VT,NVL),
   x_colinear_insert(VS,NVL,TVL),
   !.

% x_update_edge(Edge,Old_Vlist,New_Vlist).
% Update Edge with a new list of points.

x_update_edge(_,VL,VL):-!.
x_update_edge(E,VL1,VL2):-x_update_edge_action(E,VL1,VL2),fail.
x_update_edge(_,_,_).      % reclaim heap space with fail...

x_update_edge_action(E,VL1,VL2):-
   retract(e(E,VL1)),
   asserta(e(E,VL2)),
   !.

% x_1_point(V#).
% Intersection at a point: end-point at edge.

x_1_point(VN):-
   xnode(VN,_),      % already done.
   !.
x_1_point(VN):-
   new_node(T),
   asserta(xnode(VN,T)),
   mark_as_node(VN),
   !.

% x_2_points(V1#,V2#).
% Two points intersection: edge touching edge at end-points.

x_2_points(VN1,VN2):-
   xnode(VN1,_),
   xnode(VN2,_),      % already done.
   !.
x_2_points(VN1,VN2):-
   new_node(T),
   asserta(xnode(VN1,T)),
   asserta(xnode(VN2,T)),
   mark_as_node(VN1),
   mark_as_node(VN2),
   !.

% mark_as_node(V#).
% Vertex point becomes a node...

mark_as_node(VN):-
   (VN//100000) mod 10 =:= 0,   % node.
   !.
mark_as_node(VN):-
   v(VN,TYPE,COORD),
   (TYPE=n,
    !;
```

```
    TYPE=v,
    retract(v(VN,v,COORD)),
    asserta(v(VN,n,COORD)),
    !).

% point_vs_point.
% Determine which side in 1D projection.

point_vs_point(P,Q,R):-
   v(P,_,V),
   pt_vs_pt(V,Q,R),
   !.

pt_vs_pt(P,P,0):-!.
pt_vs_pt([X,Y1],[X,Y2],R):-compare_2nums(Y1,Y2,R),!.
pt_vs_pt([X1,_],[X2,_],R):-compare_2nums(X1,X2,R),!.

compare_2nums(Z1,Z2,1):-Z1<<Z2,!.
compare_2nums(_,_,R):-R is -1 .

% new_node(N#).
% New node id generated...

xroot(0).   % keeps count of new nodes...

new_node(N):-
   retract(xroot(N0)),
   N is N0+1,
   asserta(xroot(N)),
   !.

append([],L,L).
append([H|L1],L2,[H|L3]):-append(L1,L2,L3).
```

```
% xconn, prolog/over4
% Connect edges into chains after intersection.
% Peter Wu - 4/20/87

% 'xconn' takes the edges after the intersection process to
% connect them into polyline structures, forming the chains
% of the overlay map.  Chains are split at the new nodes of
% the intersection points.  'xconn' needs 4 input data sets:
%
%   b(C#,_,_).          % chain id#'s
%   e(E#,[V...]).       % set of edges
%   v(V#,Type,[x,y]).   % set of vertices
%   xnode(V#,N#).       % node intersection (special cases).
%
% 'xconn' uses the last 3 digits to re-connect the edges of
% each chain: first the list of V#'s, then polyline list of
% [x,y] coordinates.  The chain is split at the new nodes.
% The output chain has new C# retaining the original map id
% and old C#, to keep track of other attributes.  'xconn'
% generates as output two data sets:
%
%   c(C#,N1,N2,[[x,y],...]).   set of chains
%   v(N#,n,[x,y]).             set of nodes

xconn:-T0 is cputime,
       assert(xconn_timer(T0)),
       fail.

xconn:-b(C,_,_),        % original Chain's...
       conn_vs(C,VS),
       asserta(cvs(C,VS)),
       fail.

xconn:-abolish(b,3),    % claim some space...
       abolish(e,2),
       fail.

xconn:-cvs(C,VS),       % split chains...
       xs_cvs(C,VS),
       fail.

xconn:-xnode(XN,N),     % replaced node...
       rm_xnode(XN,N),
       fail.

xconn:-retract(v(_,v,_)),fail.

xconn:-retract(xconn_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (xconn).'),
       nl.

% conn_vs(C#,VSet).
% Connect edges of chain C# to the list of V#'s.

conn_vs(C,VS):-
   E is C+1,
   e(E,VS1),
   (conn_vs(E,[_|VS2]),    % get the rest...
    append(VS1,VS2,VS),
    !;
    VS=VS1,                % last one.
    !).

% xs_cvs(C#,VSet).
% Construct the polyline list of [x,y] coordinates and split
% into chains...

xs_cvs(C,[VN1|VS]):-
   v(VN1,n,V1),
   t_xnode(VN1,N1),             % node replaced ?
   xs_cvs2(C,VS,L,N2),          % get the rest of polyline...
   asserta(c(C,N1,N2,[V1|L])),
   !.

% xs_cvs2(C#,VSet,Polyline,N2).
% Continuation to gather the rest of polyline...

xs_cvs2(_,[VN2],[V2],N2):-
   v(VN2,n,V2),                 % last node.
   t_xnode(VN2,N2),             % node replaced ?
   !.

xs_cvs2(C,[VN|VS],L,N):-
   v(VN,TYPE,V),
   (TYPE=v,                     % vertex...
    xs_cvs2(C,VS,L2,N),         % more in polyline.
    L=[V|L2],
    !;
    TYPE=n,                     % node...
    L=[V],                      % last node for chain
    t_xnode(VN,N),              % get replacement N#
    C1 is C+1,                  % starting next chain...
    xs_cvs2(C1,VS,L2,N2),
    asserta(c(C1,N,N2,[V|L2])),
    !).

% t_xnode(VName,N#).
% Test replacement: get N# if replaced.

t_xnode(XN,N):-xnode(XN,N),!.
t_xnode(N,N).

% rm_xnode(VName,N#).
% Remove replacement, set up new node N#.

rm_xnode(XN,N):-
   retract(v(XN,n,V)),
   ( v(N,n,V); asserta(v(N,n,V)) ),!.

append([],L,L).
append([H|L1],L2,[H|L3]):-append(L1,L2,L3).
```

```
% calci, prolog/over4
% Calculate chain-to-node incidence angles.
% Peter Wu - 3/12/87

% *calci* calculates for each chain the incident angles at
% the beginning and ending nodes.  We can then sort out the
% topological relationships around the neighborhood of each
% node.  *calci* takes for input the set of chains in the
% following format:
%
%   c(C#,N1,N2,[[x,y],...]).     % network chains...
%
% *calci* uses rational arithmetic; angle is expressed
% as an ordered pair [Q,A] where Q=1,2,3,4 identifies
% the quadrant for the angle, and A is the value of
% sin(A)/cos(A) for Q=1,3 or -cos(A)/sin(A) for Q=2,4.
% We preserved cyclic order.
%
% *calci* generates as output the incidence records:
%
%   i(N#,Incidence,Angle).
%
% where N# is the node name; Incidence is "h(C#)" or
% "t(C#)" for the head or tail of chain C#; Angle is
% explained above.

calci:-T0 is cputime,
       assert(calci_timer(T0)),
       fail.

calci:-c(C,N1,N2,L),
       inc_vectors(L,VECT1,VECT2),
       inc_angle(VECT1,ANGL1),
       inc_angle(VECT2,ANGL2),
       asserta(i(N1,t(C),ANGL1)),
       asserta(i(N2,h(C),ANGL2)),
       fail.

calci:-retract(calci_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (calci).'),
       nl.

% inc_vectors(Polyline,1st_Incident_Vector,2nd_Incident_Vector).
% Get the beginning and ending incident vectors from the polyline list.

inc_vectors(L,U1,U2):-
   inc_vect_1(L,U1),
   inc_vect_2(L,U2).

inc_vect_1([P1,P2|_],U1):-subtract_vects(P2,P1,U1).

inc_vect_2([P1,P2],U2):-!,subtract_vects(P1,P2,U2).

inc_vect_2([_|L],U2):-inc_vect_2(L,U2).

subtract_vects([X1,Y1],[X2,Y2],[X,Y]):-
   X isx X1-X2,
   Y isx Y1-Y2.
```

```
% inc_angle(Vector,Angle).
% Compute incident angle of vector against +ve X-axis.

inc_angle([X,Y],ANGLE):-
    float_xq(X,XF),
    float_xq(Y,YF),
    inc_angle2([XF,YF],ANGLE),
    !.

% inc_angle2(Vector,Angle).
% Vector coordinates in floating point for speed.

inc_angle2([0,0],_):-
   error('Incident Angle for Zero Vector.').

inc_angle2([U,0],[Q,0]):-
   ( U>0,Q=1;
     Q=3 ),!.

inc_angle2([0,V],[Q,0]):-
   ( V>0,Q=2;
     Q=4 ),!.

inc_angle2([U,V],[Q,A]):-
   U>0,!,
   ( V>0,Q=1,A is V/U;
     Q=4,A is -U/V ),!.

inc_angle2([U,V],[Q,A]):-
   ( V>0,Q=2,A is -U/V;
     Q=3,A is V/U),!.
```

```
% sorti, prolog/over4
% Sort incident chains around each node.
% Peter Wu - 3/25/87

% *sorti* considers each node: sorting the incident
% chains around the node into positive cyclic order
% (counter-clockwise). Hence each pair of adjacent
% chains identifies one polygon corner.  *sorti*
% then generates these pairs in linkage records for
% subsequent linking up of chains to form polygons.
% The input data sets for *sorti* include:
%
%   v(N#,n,[x,y]).          set of nodes
%   i(N#,Incidence,Angle).  chain-to-node incidence
%
% where N# is the node name; Incidence is "h(C#)" or
% "t(C#)" for the head or tail of chain C#; Angle is
% the incident angle made with the positive x-axis.
% Linkage records from *sorti* have the format:
%
%   link([B1,B2],B2).  where Bi is th(C#) or ht(C#).
%
% While "th(C#)" or "ht(C#)" indicate the chain in the
% direction tail-to-head or head-to-tail respectively.
% Furthermore, *sorti* generates the output nodes with
% the list of incident chains in proper cyclic order:
%
%   node(N#,[x,y],[I,...]).  where I = t(C#) or h(C#);
%
% And "t(C#)" and "h(C#)" indicate tail/head of chain
% C#.  *sorti* identifies special case of exact overlap
% chains by equal incident angles.  *sorti* marks the
% chains to be replaced by a new chain id#...
%
%   xb(C#,C,X).    % C by C#; X=replaced/reversed...
%   xc(C1,C2,C#).  % C1,C2 replaced (C# same dir as C1).
%
% Both "xb" and "xc" are output for subsequent processes.
% "xb" is also needed in *sorti* ...

sorti:-T0 is cputime,
       assert(sorti_timer(T0)),
       fail.

sorti:-v(N,n,V),
       sort_inc(N,IS),
       gen_link(IS),
       asserta(node(N,V,IS)),
       fail.

sorti:-retract(sorti_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (sorti).'),
       nl.

% sort_inc(Node,List_of_Incidences).
% Gather the incident chains at Node and sort in cyclic order.

sort_inc(N,IS):-
   setof(x(ANGLE,I),i(N,I,ANGLE),S),
```

```
   form_ilist(S,IS),
   !.

% form_ilist(Set,Incidence_List).
% Remove Angle field and check for overlap chains.

form_ilist([x(A,I1),x(A,I2)|S],IS):-
   I1 =.. [D1,C1],
   I2 =.. [D2,C2],                       % dir and chain id#
   x_2_chains(D1,C1,D2,C2,D3,C3),!,      % mark as overlap...
   I3 =.. [D3,C3],
   form_ilist([x(A,I3)|S],IS).

form_ilist([x(_,I1)|S],[I1|IS]):-form_ilist(S,IS).

form_ilist([],[]).

% gen_link(Incidence_List).
% Generate linkage records to mark each polygon corner.

gen_link([I|L]):-gen_link2([I|L],I),!.

gen_link2([I1],I2):-
   incoming_link(I2,B2),
   outgoing_link(I1,B1),
   asserta(link([B2,B1],B1)).

gen_link2([I1,I2|L],I0):-
   incoming_link(I2,B2),
   outgoing_link(I1,B1),
   asserta(link([B2,B1],B1)),
   gen_link2([I2|L],I0).

% incoming/outgoing_link(Incident_Chain).
% Mark direction from chain incidence.

incoming_link(t(C),ht(C)).
incoming_link(h(C),th(C)).

outgoing_link(t(C),th(C)).
outgoing_link(h(C),ht(C)).

% x_2_chains(D1,C1,D2,C2,D3,C3).
% Overlap Chains: replace with new chain and incidence.

x_2_chains(D1,C1,D2,C2,D3,C3):-
   ( xb(C3,C1,X),            % already replaced ?
     x_dir(D1,X,D3);
     xb(C3,C2,X),
     x_dir(D1,X,D3) ),
   !.

x_2_chains(D1,C1,D2,C2,D3,C3):-
  D1=D3,
  new_chain_id(C3),
  asserta(xb(C3,C1,rep)),
  ( D1=D2,
    asserta(xb(C3,C2,rep));
    asserta(xb(C3,C2,rev)) ),
  asserta(xc(C1,C2,C3)),   % C1,C2 replaced by C3.
```

```
   !.

% x_dir(Dir,Replaced/Reversed,New_Direction).
% Compose new direction of replacing chains...

x_dir(h,rep,h).
x_dir(t,rep,t).
x_dir(h,rev,t).
x_dir(t,rev,h).

% new_chain_id(C#).
% Get new chain id# for the overlay map...

new_chain_id(C):-
   retract(xcroot(C0)),
   C is C0+1,
   assert(xcroot(C)),
   !.

xcroot(0).
```

```
% cpoly, prolog/over4
% Link up chain incidences to form polygons boundaries.
% Peter Wu - 4/20/87

% *cpoly* works on a set of "link facts," which are
% given in the following format:
%
%   link([B1,B2],B2).  where Bi is th(C#) or ht(C#).
%
% While "th(C#)" or "ht(C#)" indicate the chain in the
% direction tail-to-head or head-to-tail respectively.
% Each "link fact" identifies the corner of a polygon.
% *cpoly* links them up to form "longer" linkages:
%
%   link([B1,B2,....,Bn],Bn).
%
% If a link fact completes a cycle (ie, B1=Bn), *cpoly*
% declares it the boundary chains of a polygon.  Input
% for *cpoly* is the set of "link facts;" output is the
% set of polygons thus generated:
%
%   polygon(P#,[B1,B2,...]).
%
% *cpoly* uses a special "repeated_retract" to perform
% the set-based operation, until the set is exhausted.

cpoly:-T0 is cputime,
       assert(cpoly_timer(T0)),
       fail.

cpoly:-repeated_retract(link(L1,B2)),
       connect1(L1,B2),
       fail.

cpoly:-retract(cpoly_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (cpoly).'),
       nl.

% repeated_retract(link(List,Last)).
% Repeated retract always starts at top of list; it fails
% when no link facts can be found...

repeated_retract(LINK):-
   repeat,
   ( not(LINK),!,fail;
     retract(LINK) ).

% connect1(List,Last).
% Check List for completed polygon, otherwise find the
% link for connection...

connect1([B|L],B):-            % first=last
   new_polygon_id(P),          % it is a polygon...
   asserta(polygon(P,L)),
   !.

connect1(L1,B2):-
   retract(link([B2|L2],B3)),  % find linkage
   append(L1,L2,L3),           % connect the two...
   asserta(link(L3,B3)),
   !.

connect1(L,B):-
   nl,
   write('| erroneous linkage topology: core dumped.'),nl,
   save(core),
   write('| Processing: '),write(link(L,B)),nl,
   abort.

% new_polygon_id(P#).
% Get new polygon id# for the overlay map...

new_polygon_id(P):-
   retract(proot(P0)),
   P is P0+1,
   assert(proot(P)),
   !.

proot(0).

% utility.. %

append([],L,L).
append([H|L1],L2,[H|L3]):-append(L1,L2,L3).
```

```
% bound. prolog/over4
% Determine bounding polygons for each chain.
% Peter Wu - 3/24/87

% 'bound' determines for each chain the polygons on both
% sides.  Input to 'bound' consists two data sets:
%
%   c(C,N1,N2,[[x,y],...]).     set of chains
%   polygon(P,[B1,B2,...]).     set of polygons
%
% where Bi is either th(C) or ht(C) to indicate direction
% of the boundary chains around the polygon.  'bound'
% generates a temporary set of left/right records:
%
%   left(C,P).     polygon P is to the left of chain C
%   right(C,P).    polygon P is to the right of chain C
%
% 'bound' incorporates this information into the chains
% to generate the output chains in the following format:
%
%   chain(C,N1,N2,[[x,y],...],P1,P2).
%
% where P1,P2 are the polygons to the left and right of
% chain C, respectively.
%
% 'bound' also removes special cases of exact overlap
% chains identified in 'sorti', recorded as
%
%   xc(C1,C2,C#).
%
% for overlap chains C1 and C2 replaced by C#...

bound:-T0 is cputime,
       assert(bound_timer(T0)),
       fail.

bound:-xc(C1,C2,NC),
       replace_overlap_chains(C1,C2,NC),
       fail.

bound:-polygon(P,L),
       mark_boundary_chains(L,P),
       fail.

bound:-c(C,N1,N2,L),
       find_bounding_polygons(C,P1,P2),
       asserta(chain(C,N1,N2,L,P1,P2)),
       fail.

bound:-retract(bound_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (bound).'),
       nl.

% replace_overlap_chains(C1,C2,New_Chain_id).
% Remove overlap chains C1 and C2, replace with New Chain.

replace_overlap_chains(C1,C2,NC):-
   retract(c(C1,N11,N12,L1)),
   retract(c(C2,N21,N22,L2)),
```

```
   asserta(c(NC,N11,N12,L1)),
   !.

% mark_boundary_chains(List_of_Chains,Polygon_id).
% Mark the polygon to the left/right of each boundary chain.

mark_boundary_chains([th(C)|_],P):-
   asserta(left(C,P)),
   fail.

mark_boundary_chains([ht(C)|_],P):-
   asserta(right(C,P)),
   fail.

mark_boundary_chains([_|L],P):-
   mark_boundary_chains(L,P),
   !.

mark_boundary_chains([],_).

% find_bounding_polygons(C#,Left_P#,Right_P#).
% Determine the Polygon_id#'s to the left/right of chain C#.

find_bounding_polygons(C,P1,P2):-
   retract(left(C,P1)),
   retract(right(C,P2)),
   !.

% utilities %

append([],L,L).
append([H|L1],L2,[H|L3]):-append(L1,L2,L3).
```

```
% xover, prolog/over4
% Identify overlay polygons.
% Peter Wu - 3/25/87

% *xover* attempts to determine for each output polygon the
% two input polygons which overlay (intersect) to form it.
% If the boundary chains are involved in intersections, the
% output polygon has boundary chains split from those of
% the input polygons.  The original input polygons can then
% be determined from the list of boundary chains.  However,
% if an output polygon is not involved with any intersection,
% all its boundary chains come from the same map, and the
% same input polygon.  It is contained in a polygon in the
% other input map.  Here we search its neighbors, which must
% be contained in the same polygon.  The process fails only
% when the output polygon is not connected to another output
% polygon involved in intersection.
%
% *xover* needs for input the set of polygons, the original
% map id's, and boundary records of the input maps. *xover*
% also needs the set of chains to search the neighbors:
%
%   polygon(P,[B1,B2,...]).               polygons
%   root(Map,Id).                         input map id's
%   b(B#,Left_P,Right_P).                 input map boundaries
%   chain(C,N1,N2,[[x,y],...],P1,P2).     chains
%
% For chains replaced due to exact overlap...
%
%   xb(C#,C,Dir).    % Dir=replaced/reversed: chain C by C#
%
% *xover* records the results of polygon overlay as:
%
%   over(P#,P1,P2).
%
% for output polygon P# with overlay polygons P1 in map#1 and
% P2 in map#2.  During the process, *xover* generates
%
%   o1(P#,P1).
%   o2(P#,P2).
%
% whenever P1 and P2 for P# can be found from the boundary
% chains.  "o1" and "o2" are combined to form over(P,P1,P2).
% For the ones left, *xover* searches their neighbors for the
% orginial input polygons to form "over".  But if the search
% fails, we have a group of connected polygons not involved
% with any intersection.  These are then grouped together:
%
%   o1(G#,P#,P1).
%   o2(G#,P#,P2).
%
% identified to be the same group by G#.  These are cases of
% polygon containment for subsequent processing.

xover:-T0 is cputime,
       assert(xover_timer(T0)),
       fail.

% make "o1" and "o2" for each polygon...
```

```
xover:-setof(M,MAP^root(MAP,M),[M1,M2]),
       polygon(P,L),
       xover_polygon(o1,M1,P,L),
       xover_polygon(o2,M2,P,L),
       fail.

% combine "o1" and "o2" to form "over"...

xover:-retract(o1(P,P1)),
       retract_o2_or_assert_o1(P,P1,P2),
       asserta(over(P,P1,P2)),
       fail.

% search neighbors for singletons of "o1"/"o2"...

xover:-retract(o1(P,P1)),
       ogroup(G),
       asserta(o1(G,P,P1)),
       search_thru_neighbors(1,G,P,P2),
       proc_o_group(1,G,P2),
       fail.

xover:-retract(o2(P,P2)),
       ogroup(G),
       asserta(o2(G,P,P2)),
       search_thru_neighbors(2,G,P,P1),
       proc_o_group(2,G,P1),
       fail.

xover:-retract(xover_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (xover).'),
       nl.

% xover_polygon(Over,Map,P,Boundary_List).
% Determine one original input polygon from the boundary chains of P,
% and assert Over fact...

xover_polygon(O,M0,P,L):-
   member(B,L),                    % take one boundary.
   B =.. [D,C],                    % direction and chain.
   M is C - C mod 1000000,         % from which map?
   ( M=M0,                         % if this one...
     B0 is C - C mod 1000,         % get original chain id#...
     (D=th,b(B0,P0,_);             % and original polygon.
      D=ht,b(B0,_,P0)),
     OVER =.. [O,P,P0],            % form "o1" or "o2" fact
     asserta(OVER),                % assert it!
     !;
     M=0,!,                        % overlap boundary?
     xover_2_ps(P,D,C)).           % Yes. Get both polygons...

xover_polygon(_,_,_,_).            % for containment case...

% xover_2_ps(P#,Dir,C#).
% C# is a common chain replaced: get both over polygons for
% P# by checking into "xb" table...

xover_2_ps(P,D,C):-
   setof([C0,X0],xb(C,C0,X0),[[C1,X1],[C2,X2]]),
     B1 is C1 - C1 mod 1000,  % and original chain id#s.
     B2 is C2 - C2 mod 1000,
     xover_p_orientation(D,B1,X1,P1),
     xover_p_orientation(D,B2,X2,P2),
     asserta(over(P,P1,P2)),
     !,  % both polygons done.
     (retract(o1(P,P1));retract(o2(P,P2))),
     fail.

xover_p_orientation(th,B,rep,P):-b(B,P,_),!.
xover_p_orientation(th,B,rev,P):-b(B,_,P),!.
xover_p_orientation(ht,B,rep,P):-b(B,_,P),!.
xover_p_orientation(ht,B,rev,P):-b(B,P,_),!.

% retract_o2_or_assert_o1(P#,P1,P2).
% Taken "o1", retract the matching "o2"; but if there is no
% matching "o2", put the "o1" back..., and fail.

retract_o2_or_assert_o1(P,_,P2):-retract(o2(P,P2)),!.
retract_o2_or_assert_o1(P,P1,_):-asserta(o1(P,P1)),!,fail.

% ogroup(G#).
% Generate group id# for each connected group of polygons.

ogroup(G):-
   retract(oroot(G0)),
   G is G0+1,
   assert(oroot(G)),!.

oroot(0).

% search_thru_neighbors(Index,Group#,P,Contained_P).
% Search the neighbors of P for CP, the polygon from the other input
% map containing P.  Index=1 or 2 to indicate that P is an original
% polygon from map#1 or map#2.  Group# is unique to the group of
% neighbors connected together.

search_thru_neighbors(X,G,P,CP):-
   polygon(P,BL),!,
   member(B,BL),
   x_boundary_to_neighbor(B,NP),
   (X=1,
    not(o1(G,NP,_)),                  % already looked at?
    search_polygon_for_p2(G,NP,CP);   % depth first search...
    X=2,
    not(o2(G,NP,_)),
    search_polygon_for_p1(G,NP,CP)),
   !.

member(B,[B|_]).
member(B,[_|BL]):-member(B,BL).

x_boundary_to_neighbor(th(C),NP):-chain(C,_,_,_,_,NP),!.
x_boundary_to_neighbor(ht(C),NP):-chain(C,_,_,_,NP,_),!.

% search_polygon_for_px(Group#,P,Px).
% To determine overlay Px for polygon P: if P is resolved,
% the entire group G is resolved; otherwise, keep searching
% the neighbors.  If it still fails, the connected neighbors
% are grouped under G#.
```

```
search_polygon_for_p1(G,P,P1):-
   retract(o2(P,P2)),
   asserta(o2(G,P,P2)),!,
   search_thru_neighbors(2,G,P,P1).

search_polygon_for_p1(_,P,P1):-over(P,P1,_),!.

search_polygon_for_p2(G,P,P2):-
   retract(o1(P,P1)),
   asserta(o1(G,P,P1)),!,
   search_thru_neighbors(1,G,P,P2).

search_polygon_for_p2(_,P,P2):-over(P,_,P2),!.

% proc_o_group(X,Group#,Px).
% The overlay polygon for group G is determined to be Px.
% Resolve singleton (X=1/2:o1/o2) records of the group.

proc_o_group(1,G,P2):-
   retract(o1(G,P,P1)),
   asserta(over(P,P1,P2)),
   fail.

proc_o_group(2,G,P1):-
   retract(o2(G,P,P2)),
   asserta(over(P,P1,P2)),
   fail.

proc_o_group(_,_,_).
```

```
% pcont, prolog/over4
% Resolve polygon containment cases.
% Peter Wu - 4/7/87

% *pcont* resolves polygon containment cases in the process of
% polygon overlay.  Having the entire tolopology, *pcont* takes
% the groups of polygons which are not involved in any inter-
% section determines for each connected group their containment
% polygon.  *pcont* takes adavantage of the known topology to
% cut down the number of point-in-polygon tests necessary.
% *pcont* takes for input the polygon overlay information in
%
%   over(P#,P1,P2).   ... output polygon P = P1.intersect.P2
%
% and the sets of polygon containment cases as
%
%   o1(Group#,P#,P1).    o2(Group#,P#,P2).
%
% where polygon P# is P1 in input map.1 or P2 in input map.2,
% but the containment polygon in the other map is not known.
% Entries of "o1" or "o2" are grouped together by the same G#
% to identify connected groups.  *pcont* also needs the input
% map data and the original root id's...
%
%   root(MROOT,Map.id).
%   chain(C#,Map.id,N1,N2,[[x,y],...],P1,P2).
%
% *pcont* first determines the outside polygons for each input
% map.  Then determines the containment for each group testing
% point-in-polygon for polygons which intersect with the outside
% polygon.  Resolving "o1" and "o2", *pcont* adds new entries
% to "over(P#,P1,P2)".

pcont:-T0 is cputime,
       assert(pcont_timer(T0)),
       fail.

% determine the outside polygon in each input map...

pcont:-setof([ROOT,M],root(M,ROOT),[[_,M1],[_,M2]]),
       assert(mroot(M1,M2)),
       (det_outside_p1(M1,P1),assert(outside_p(M1,P1));
        det_outside_p2(M2,P2),assert(outside_p(M2,P2))),
       fail.

% resolve "o1" and then "o2" ...

pcont:-mroot(M1,M2),
       (pcont1(M1,M2); pcont2(M2,M1)),
       fail.

pcont:-retract(pcont_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (pcont).'),
       nl.

% det_outside_pX(MapX,Outside_Polygon).
% Determine outside polygon in input map X: there is one polygon
% in each group of connected polygons which is also involved in
% intersection in the overlay process...
```

```
det_outside_p1(M1,P1):-o1(_,_,P1),over(_,P1,_),!.
det_outside_p1(M1,P1):-o1(_,_,P1),p_cycle(M1,P1,S1),S1<0,!.

det_outside_p2(M2,P2):-o2(_,_,P2),over(_,_,P2),!.
det_outside_p2(M2,P2):-o2(_,_,P2),p_cycle(M2,P2,S2),S2<0,!.

% p_cycle(Map,Polygon,Cyclic_Order).
% Calculate total cyclic order for given polygon: outside polygon
% has a negative cyclic order. (needed if no intersection at all)

p_cycle(_,_,_):-
   assert(cycle_sum(0)),
   fail.

p_cycle(M,P,_):-
   (chain(_,M,_,_,L,P,_);
    chain(_,M,_,_,L0,_,P),reverse(L0,L)),
   float_p1(L,FL),cycle(FL,S),
   update_cycle_sum(S),
   fail.

p_cycle(_,_,S):-
   retract(cycle_sum(S)),
   !.

% reverse(List1,List2).
% Reverse List1 to make List2.

reverse(L0,L):-reverse2(L0,[],L),!.

reverse2([],L,L).
reverse2([H|L0],L1,L):-reverse2(L0,[H|L1],L).

% cycle(Polyline,Cycle_Sum).
% Sum of cyclic order measure: area subtended at origin.

cycle([_],0).
cycle([[X1,Y1],[X2,Y2]|L],S):-
   cycle([[X2,Y2]|L],CS),
   S is X1*Y2-X2*Y1+CS,
   !.

% update_cycle_sum(Cyclic_Order_Measure).
% Update accumulated measre of total cyclic order.

update_cycle_sum(S):-
   retract(cycle_sum(S0)),
   S1 is S0+S,
   assert(cycle_sum(S1)),
   !.

% pcontX(Map1,Map2).
% Process cases of polygon containment for map#1-polygons-in-map#2.

pcont1(M1,M2):-
   setof(P2,over_p2s(P2),P2S),    % possible containment polygons
   group1_pt(M1,G,PT),            % for each Group and its Point...
   det_cpoly(PT,M2,P2S,P2),       % determine containment: P2.
   proc_o_group(1,G,P2).          % group G resolved.
```

```
pcont2(M2,M1):-
    setof(P1,over_p1s(P1),P1S),  % possible containment polygons
    group2_pt(M2,G,PT),           % for each Group and its Point...
    det_cpoly(PT,M1,P1S,P1),      % determine containment: P1.
    proc_o_group(2,G,P1).         % group G resolved.

% over_pXs(Polygon_X).
% Select possible containment polygons in Map X: those which inter-
% sect the outside polygon, as well as the unknown ones...

over_p1s(P1):-
    mroot(M1,M2),!,
    (outside_p(M2,O2),over(_,P1,O2); o1(_,_,P1)),
    not(outside_p(M1,P1)).

over_p2s(P2):-
    mroot(M1,M2),!,
    (outside_p(M1,O1),over(_,O1,P2); o2(_,_,P2)),
    not(outside_p(M2,P2)).

% groupX.pt(Map,Group#,Point).
% Iterate through groups of connected polygons of Map, and return
% a point within the group for subsequent containment tests...

group1_pt(M1,G,PT):-
    repeat,
    (not(o1(_,_,_)),!,fail;
     get_group1_pt(M1,G,PT)).

get_group1_pt(M1,G,PT):-
    o1(G,_,P1),
    not(outside_p(M1,P1)),
    (chain(_,M1,_,_,[PT|_],P1,_);
     chain(_,M1,_,_,[PT|_],_,P1)),
    !.

group2_pt(M2,G,PT):-
    repeat,
    (not(o2(_,_,_)),!,fail;
     get_group2_pt(M2,G,PT)).

get_group2_pt(M2,G,PT):-
    o2(G,_,P2),
    not(outside_p(M2,P2)),
    (chain(_,M2,_,_,[PT|_],P2,_);
     chain(_,M2,_,_,[PT|_],_,P2)),
    !.

% det_cpoly(Point,Map,Possible_cPs,containment_P).
% Determine containment polygon in Map for the given Point...

det_cpoly(PT,M,PS,CP):-
    member(CP,PS),
    point_in_polygon(PT,M,CP),
    !.
det_cpoly(_,M,_,CP):-
    outside_p(M,CP),
    !.
```

```
member(P,[P|_]).
member(P,[_|PS]):-member(P,PS).

% proc_o_group(X,Group#,Pc).
% The containment polygon for group G is determined to be Pc.
% Resolve the singleton (X=1/2:o1/o2) records of the group G.

proc_o_group(1,G,P2):-
    retract(o1(G,P,P1)),
    asserta(over(P,P1,P2)),
    fail.

proc_o_group(2,G,P1):-
    retract(o2(G,P,P2)),
    asserta(over(P,P1,P2)),
    fail.

proc_o_group(_,_,_).

% point_in_polygon(Point,Map,Polygon).
% Determine whether Point in inside Polygon in specified Map.

point_in_polygon(_,_,_):-
    assert(pip_xings(1)),
    fail.

point_in_polygon(PT,M,P):-
    (chain(_,M,_,_,L,P,_); chain(_,M,_,_,L,_,P)),
    float_v2(PT,PTF),
    float_pl(L,FL),
    pip(PTF,FL,S),
    update_pip_xings(S),
    fail.

point_in_polygon(_,_,_):-
    retract(pip_xings(S)),
    S<0,
    !.

% update_pip_xings(Sign).
% Count odd/even number of crossings by Sign=-1/+1.

update_pip_xings(S):-
    retract(pip_xings(S0)),
    S1 is S0*S,
    assert(pip_xings(S1)),
    !.

% pip(Point,Polyline,S).
% Jordan Curve Theorem for point-in-polygon check: extend ray
% from point in positive Y-direction to count the number of
% crossings the polyline makes.  S=+1/-1:even/odd, S=0:on.

pip(P,[V1|VL],S):-
    comp2vs(P,V1,R,D),
    pip2(P,R,D,[V1|VL],S),
    !.

pip2(P,0,D1,[_,V2|VL],S):-
    comp2vs(P,V2,R2,D2),
   (R2=0,!,
     (D1*D2>0,
      pip2(P,0,D2,[V2|VL],S),
      !;
      S=0,
      !);
    D1>0,R2<0,!,
    pip2(P,R2,D2,[V2|VL],S2),
    S is -1*S2,
    !;
    pip2(P,R2,D2,[V2|VL],S),
    !).

pip2(P,R1,D1,[V1,V2|VL],S):-
    comp2vs(P,V2,R2,D2),
   (R1*R2<0,
    y_intercept(D1,D2,I),
     (I>0,!,
      pip2(P,R2,D2,[V2|VL],S2),
      S is -1*S2,
      !;
      I=0,S=0,
      !);
    R2=0,D2>0,R1<0,!,
    pip2(P,0,1,[V2|VL],S2),
    S is -1*S2,
    !;
    pip2(P,R2,D2,[V2|VL],S),
    !).

pip2(P,0,0,[P],0):-!.
pip2(_,_,_,[_],1).

% comp2vs(Vect1,Vect2,R,D).
% Compare 2 vectors:  R=-1/+1:left/right, D=V1-V0;
% same x-coordinate:  R=0, D=-1/+1:below/above, D=0:equal.

comp2vs([X,Y],[X,Y],0,0):-!.

comp2vs([X,Y0],[X,Y1],0,D):-
    (Y0<Y1,D=1; D is -1),
    !.

comp2vs([X0,Y0],[X1,Y1],R,[DX,DY]):-
    DX is X1-X0,
    DY is Y1-Y0,
    (DX>0,R=1; R is -1),
    !.

y_intercept([X1,Y1],[X2,Y2],I):-
    Z1 is X1*Y2,
    Z2 is X2*Y1,
    comp2ns(Z1,Z2,I1),
    comp2ns(X1,X2,I2),
    I is I1*I2,
    !.

comp2ns(N,N,0):-!.
comp2ns(N1,N2,R):-
    (N1<N2,R=1;
```

```
     R is -1),
     !.

% float_pl(Polyline,Polyline_in_Floating_pt).
% Convert List of (x,y) from XQ to regular floating pt.

float_pl([],[]).
float_pl([P|L],[FP|FL]):-
    float_v2(P,FP),
    float_pl(L,FL).

float_v2([X,Y],[XF,YF]):-
    float_xq(X,XF),
    float_xq(Y,YF).
```

```
% jconn, prolog/over4
% Connect polygons into regions.
% Peter Wu - 4/7/87

% 'jconn' joins connected output polygons into regions. 'jconn'
% resolves containment relationships in the output polygons from
% the overlay process.  'jconn' examines each group of polygons
% from the same intersection of two input polygons; then sorts
% out the ones which are positive cyclic from the negative ones.
% Each positive cyclic polygon is one region; a negative cyclic
% polygon is a hole in a positive polygon if it is contained in
% it.  The negative cyclic polygons which are not contained in
% any of the positive polygons are themselves connected to form
% the outside region in the output map.
%
% 'jconn' takes for input the polygon overlay information and
% the set of chain information for polygon containment testing:
%
%   over(P#,P1,P2). ... output polygon P = P1.intersect.P2
%
%   chain(C#,N1,N2,[[x,y],...],Left_P,Right_P).
%
% 'jconn' joins connected polygons into one region by unifying
% the polygon id's involved.  'jconn' therefore generates the
% updated sets of "chain", "polygon" and "over"...

jconn:-T0 is cputime,
       assert(jconn_timer(T0)),
       fail.

% gather groups of polygons from the same polygon intersection...

jconn:-bagof(P,over(P,P1,P2),PS),    % groups of same intersection
       length(PS,L),L>1,             % OK if there is only one
       sort_cycles(PS,PPS,NPS),      % divide in +/- cyclic orders
       sort_connected(PPS,NPS),      % sort out connected ones.
       fail.

jconn:-retract(jconn_timer(T0)),
       T is cputime-T0,
       write('CPU used: '),write(T),write('s (jconn).'),
       nl.

% sort_cycles(Polygons,Positive_Ones,Negative_Ones).
% Divide set of polygons into two groups: positive cyclic order
% and negative cyclic order.

sort_cycles([P|PS],[P|PPS],NPS):-
   p_cycle(P,S),
   S>0,!,
   sort_cycles(PS,PPS,NPS),
   !.
sort_cycles([N|PS],PPS,[N|NPS]):-
   sort_cycles(PS,PPS,NPS),
   !.
sort_cycles([],[],[]).

% p_cycle(Polygon,Cyclic_Order).
% Calculate total cyclic order for given polygon.
```

```
p_cycle(_,_):-
   assert(cycle_sum(0)),
   fail.

p_cycle(P,_):-
   (chain(_,_,_,L,P,_);
    chain(_,_,_,L0,_,P),reverse(L0,L)),
   float_pl(L,FL),cycle(FL,S),
   update_cycle_sum(S),
   fail.

p_cycle(_,S):-
   retract(cycle_sum(S)),
   !.

% reverse(List1,List2).
% Reverse List1 to make List2.

reverse(L0,L):-reverse2(L0,[],L),!.

reverse2([],L,L).
reverse2([H|L0],L1,L):-reverse2(L0,[H|L1],L).

% cycle(Polyline,Cycle_Sum).
% Sum of cyclic order measure: area subtended at origin.

cycle([_],0).
cycle([[X1,Y1],[X2,Y2]|L],S):-
   cycle([[X2,Y2]|L],CS),
   S is X1*Y2-X2*Y1+CS,
   !.

% update_cycle_sum(Cyclic_Order_Measure).
% Update accumulated measre of total cyclic order.

update_cycle_sum(S):-
   retract(cycle_sum(S0)),
   S1 is S0+S,
   assert(cycle_sum(S1)),
   !.

% sort_connected(Positive_ones,Negative_ones).
% Sort out the polygons with connected interior.

sort_connected([P|PPS],NPS):-
   sort_contained(P,NPS,NPS2),   % take the holes inside P
   sort_connected(PPS,NPS2),     % ..and process the rest.
   !.
sort_connected([],[N|NPS]):-
   n_connect(N,NPS),             % exterior of all holes..
   !.
sort_connected([],_).

% sort_contained(P,Negative_ones,Ones_not_holes_in_P).
% Select the negative polygons which are contained with the
% polygon "P" - these are holes in "P"; the rest are returned
% for further processing.

sort_contained(P,[N|NPS],NPS2):-
   take_one_pt(N,PT),             % take a point in N
   point_in_polygon(PT,P),!,      % is it contained in P ?
   p_connect(P,N),                % Yes: N is a hole in P.
   sort_contained(P,NPS,NPS2),    % check the rest...
   !.
sort_contained(P, [N|NPS], [N|NPS2]) :-
   sort_contained(P,NPS,NPS2),    % N is not a hole in "P"..
   !.
sort_contained(_, [], []).

% take_one_pt(Polygon,Point).
% Take a point from the polygon (to check containment).

take_one_pt(P,PT) :-
   (chain(_,_,_,[PT|_],P,_); chain(_,_,_,[PT|_],_,P)),
   !.

% point_in_polygon(Point,Polygon).
% Determine whether Point is in inside the Polygon.

point_in_polygon(_,_) :-
   assert(pip_xings(1)),
   fail.

point_in_polygon(PT,P) :-
   (chain(_,_,_,L,P,_); chain(_,_,_,L,_,P)),
   float_v2(PT,PTF),
   float_pl(L,FL),
   pip(PTF,FL,S),
   update_pip_xings(S),
   fail.

point_in_polygon(_,_) :-
   retract(pip_xings(S)),
   S<0,
   !.

% update_pip_xings(Sign).
% Count odd/even number of crossings by Sign=-1/+1.

update_pip_xings(S) :-
   retract(pip_xings(S0)),
   S1 is S0*S,
   assert(pip_xings(S1)),
   !.

% pip(Point,Polyline,S).
% Jordan Curve Theorem for point-in-polygon check: extend ray
% from point in positive Y-direction to count the number of
% crossings the polyline makes.  S=+1/-1:even/odd, S=0:on.

pip(P, [V1|VL],S) :-
   comp2vs(P,V1,R,D),
   pip2(P,R,D,[V1|VL],S),
   !.

pip2(P,0,D1,[_,V2|VL],S) :-
   comp2vs(P,V2,R2,D2),
   (R2=0,!,
     (D1*D2>0,
      pip2(P,0,D2,[V2|VL],S),
      !;
      S=0,
      !);
   D1>0,R2<0,!,
   pip2(P,R2,D2,[V2|VL],S2),
   S is -1*S2,
   !;
   pip2(P,R2,D2,[V2|VL],S),
   !).

pip2(P,R1,D1,[V1,V2|VL],S) :-
   comp2vs(P,V2,R2,D2),
   (R1*R2<0,
    y_intercept(D1,D2,I),
     (I>0,!,
      pip2(P,R2,D2,[V2|VL],S2),
      S is -1*S2,
      !;
      I=0,S=0,
      !);
   R2=0,D2>0,R1<0,!,
   pip2(P,0,1,[V2|VL],S2),
   S is -1*S2,
   !;
   pip2(P,R2,D2,[V2|VL],S),
   !).

pip2(P,0,0,[P],0) :-!.
pip2(_,_,_,[_],1).

% comp2vs(Vect1,Vect2,R,D).
% Compare 2 vectors:  R=-1/+1:left/right, D=V1-V0;
% same x-coordinate:  R=0, D=-1/+1:below/above, D=0:equal.

comp2vs([X,Y],[X,Y],0,0) :-!.

comp2vs([X,Y0],[X,Y1],0,D) :-
   (Y0<Y1,D=1; D is -1),
   !.

comp2vs([X0,Y0],[X1,Y1],R,[DX,DY]) :-
   DX is X1-X0,
   DY is Y1-Y0,
   (DX>0,R=1; R is -1),
   !.

y_intercept([X1,Y1],[X2,Y2],I) :-
   Z1 is X1*Y2,
   Z2 is X2*Y1,
   comp2ns(Z1,Z2,I1),
   comp2ns(X1,X2,I2),
   I is I1*I2,
   !.

comp2ns(N,N,0) :-!.
comp2ns(N1,N2,R) :-
   (N1<N2,R=1;
    R is -1),
   !.
```

```
% n_connect(Outside_Region,Holes).
% Connect all the holes together to form the outside region.

n_connect(N,[Q|QS]) :-
   p_connect(N,Q),
   n_connect(N,QS),
   !.
n_connect(N,[]).

% p_connect(Polygon,Hole).
% The exterior of a Hole is connected to the inside of Polygon.

p_connect(P,Q) :-
   retract(chain(C,N1,N2,L,Q,P2)),
   asserta(chain(C,N1,N2,L,P,P2)),
   fail.

p_connect(P,Q) :-
   retract(chain(C,N1,N2,L,P1,Q)),
   asserta(chain(C,N1,N2,L,P1,P)),
   fail.

p_connect(P,Q) :-
   retract(polygon(Q,BS)),
   asserta(polygon(P,BS)),
   fail.

p_connect(_,Q) :-
   retract(over(Q,_,_)),
   !.

% float_pl(Polyline,Polyline_in_Floating_pt).
% Convert List of (x,y) from XQ to regular floating pt.

float_pl([],[]).
float_pl([P|L],[FP|FL]) :-
   float_v2(P,FP),
   float_pl(L,FL).

float_v2([X,Y],[XF,YF]) :-
   float_xq(X,XF),
   float_xq(Y,YF).
```