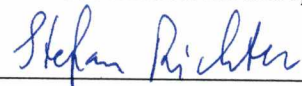


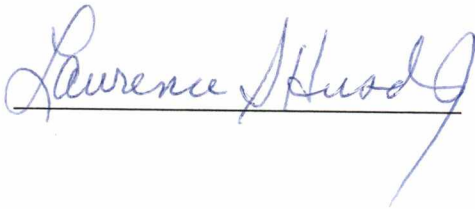
To the Graduate Council:

I am submitting herewith a thesis written by Dennis Max Depp entitled "Converting Circle-Packing Algorithms to work on the MasPar MP-2 Parallel Computer." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Mathematics.


Kenneth Stephenson, Major Professor

We have read this thesis
and recommend its acceptance:





Accepted for the Council:


Associate Vice Chancellor and
Dean of The Graduate School

Converting Circle-Packing Algorithms to work on the MasPar-2

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Dennis M. Depp
December 1996

Dedication

This thesis is dedicated to my wife

Terry Pitman Depp

who provided me with motivation and support.

Abstract

Ken Stephenson has created a computer program, CirclePack^o, that utilizes a sequential iterative approach to approximate a packing label for a given cp-complex. In this paper the author explains the migration of this algorithm to a parallel computer, the MasPar MP-2. The author explores three different algorithms, a binary search, a monotone binary search and a Newton's method algorithm. Two variables within the algorithms, factor and iterations, are adjusted to attempt to find the most efficient combination of variable for a particular algorithm. To best utilize the MasPar, the author utilized the physical connections between each node to impose a regular hexagonal complex on the Main Processor Array of the MasPar. Data was gathered on the times each algorithm took to create an approximate packing. These times were compared to identify the best conditions to reach an approximate packing.

Table of Contents

Chapter	Page
1. Introduction.....	1
2. The MasPar MP-2 Parallel Processor	7
3. Project Description	11
4. Hyperbolic Binary Algorithm	14
5. Hyperbolic Monotone Algorithm.....	20
6. Hyperbolic Newton's Method	27
7. Comparison of Hyperbolic Algorithms.....	33
8. Euclidean Binary Algorithm	38
9. Euclidean Monotone Algorithm.....	43
10. Euclidean Newton's Method Algorithm	48
11. Comparison of Euclidean Circle-Packing Algorithms	51
12. Connecting CirclePack[®] with the MasPar	54
13. Summary and Conclusions.....	55
Bibliography	57
Appendices.....	60
Vita.....	146

List of Figures

Figure		Page
Figure 1	A Sample Representation of a Complex.....	1
Figure 2	Sample Star and Flower.....	2
Figure 3	Two Different Labels on the Same Complex.....	3
Figure 4	Ordered Triple of a Complex.....	4
Figure 5	Regular Degree 8 Packing in Euclidean Geometry.....	8
Figure 6	Lazy Hex.....	9
Figure 7	Cross with Andreev Radii Packed with Hyperbolic Binary Algorithm (factor)	15
Figure 8	Cross with Andreev Radii Packed with Hyperbolic Binary Algorithm (iterations).....	16
Figure 9	Small Hex, Lh, Packed with Hyperbolic Binary Algorithm (factor).....	17
Figure 10	Cross with Randomized Radii Packed with Hyperbolic Binary Algorithm (factor)	19
Figure 11	Cross Andreev Packing Packed with Hyperbolic Monotone Algorithm (factor)	21
Figure 12	Cross Andreev Packing Packed with Hyperbolic Monotone Algorithm (iterations).....	22
Figure 13	Small Hex, Lh, Packed with the Hyperbolic Monotone Algorithm (factor).....	23
Figure 14	Small Hex, Lh, Packed with the Hyperbolic Monotone Algorithm (iterations)	24
Figure 15	Cross Andreev Packing Packed with Hyperbolic Monotone Algorithm (factor)	25
Figure 16	Cross with Andreev Radii Packed with Hyperbolic Newton's Method Algorithm (factor)	28
Figure 17	Cross with Andreev Radii Packed with Hyperbolic Newton's Method Algorithm (iterations) ..	28
Figure 18	Small hex , Lh, Packed with Hyperbolic Newton's Method Algorithm (factor).....	29
Figure 19	Small hex , Lh, Packed with Hyperbolic Newton's Method Algorithm (iterations)	30
Figure 20	El with Randomized Radii, Packed with Hyperbolic Newton's Method Algorithm (factor).....	31
Figure 21	Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Andreev Cross Packing	34
Figure 22	Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Randomized Radii Cross Packing.....	34
Figure 23	Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Small Hex Packing Lh.....	36
Figure 24	Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Small Hex Packing Lh2.....	36
Figure 25	Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Small Hex Packing Lh3.....	37
Figure 26	Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Small Hex Packing Lh4.....	37
Figure 27	Small Hex, Lh, Packed with Euclidean Binary Algorithm (factor)	39
Figure 28	Small Hex, Lh2, Packed with Euclidean Binary Algorithm (factor)	40
Figure 29	Cross with Randomized Radii Packed with Euclidean Binary Algorithm (factor).....	41
Figure 30	Small Hex Lh Packed with Euclidean Monotone Algorithm (factor)	43
Figure 31	Small Hex, Lh, Packed with Euclidean Monotone Algorithm (iterations).....	44
Figure 32	Cross with Randomized Radii Packed with Euclidean Monotone Algorithm (factor).....	45
Figure 33	Comparison of Euclidean Binary and Monotone Algorithms using Cross with Randomized Radii (factor).....	46
Figure 34	Small Hex, Lh2, Packed with Hyperbolic Newton's Method Algorithm (factor)	48
Figure 35	Cross with Randomized Radii Packed with Euclidean Newton's Method Algorithm (factor)...	49
Figure 36	Comparison of Euclidean Circle-Packing Algorithms with Small Hex, Lh.....	51
Figure 37	Comparison of Euclidean Circle-Packing Algorithms with the Cross Packing with Randomized Radii.....	52

Figure 38 El with Andreev Radii Packed with Hyperbolic Binary Algorithm (factor).....	61
Figure 39 Hemisphere with Andreev Radii Packed with Hyperbolic Binary Algorithm (factor).....	61
Figure 40 Oval with Andreev Radii Packed with Hyperbolic Binary Algorithm (factor).....	62
Figure 41 El with Andreev Radii Packed with Hyperbolic Binary Algorithm (iterations).....	62
Figure 42 Hemisphere with Andreev Radii Packed with Hyperbolic Binary Algorithm (iterations).....	63
Figure 43 Oval with Andreev Radii Packed with Hyperbolic Binary Algorithm (iterations).....	63
Figure 44 Small Hex with Boundary Condition Lh2 Packed with Hyperbolic Binary Algorithm (factor).....	64
Figure 45 Small Hex with Boundary Condition Lh3 Packed with Hyperbolic Binary Algorithm (factor).....	64
Figure 46 Small Hex with Boundary Condition Lh4 Packed with Hyperbolic Binary Algorithm (factor).....	65
Figure 47 El with Randomized Radii Packed with Hyperbolic Binary Algorithm (factor).....	65
Figure 48 Hemisphere with Randomized Radii Packed with Hyperbolic Binary Algorithm (factor).....	66
Figure 49 Oval with Randomized Radii Packed with Hyperbolic Binary Algorithm (factor).....	66
Figure 50 El with Andreev Radii Packed with Hyperbolic Monotone Algorithm (factor).....	67
Figure 51 El with Andreev Radii Packed with Hyperbolic Monotone Algorithm (iterations).....	67
Figure 52 Hemisphere with Andreev Radii Packed with Hyperbolic Monotone Algorithm (factor).....	68
Figure 53 Hemisphere with Andreev Radii Packed with Hyperbolic Monotone Algorithm (iterations)....	68
Figure 54 Oval with Andreev Radii Packed with Hyperbolic Monotone Algorithm (factor).....	69
Figure 55 Oval with Andreev Radii Packed with Hyperbolic Monotone Algorithm (iterations).....	69
Figure 56 Small Hex, Lh2, Packed with the Hyperbolic Monotone Algorithm (factor).....	70
Figure 57 Small Hex, Lh2, Packed with the Hyperbolic Monotone Algorithm (iterations).....	70
Figure 58 Small Hex, Lh, Packed with the Hyperbolic Monotone Algorithm (factor).....	71
Figure 59 Small Hex, Lh3, Packed with the Hyperbolic Monotone Algorithm (iterations).....	71
Figure 60 Small Hex, Lh4, Packed with the Hyperbolic Monotone Algorithm (factor).....	72
Figure 61 Small Hex, Lh4, Packed with the Hyperbolic Monotone Algorithm (iterations).....	72
Figure 62 El with Randomized Radii Packed with Hyperbolic Monotone Algorithm (factor).....	73
Figure 63 El with Randomized Radii Packed with Hyperbolic Monotone Algorithm (iterations).....	73
Figure 64 Hemisphere with Randomized Radii Packed with Hyperbolic Monotone Algorithm (factor) ...	74
Figure 65 Hemisphere with Randomized Radii Packed with Hyperbolic Monotone Algorithm (iterations)	74
Figure 66 Oval with Randomized Radii Packed with Hyperbolic Monotone Algorithm (factor).....	75
Figure 67 Oval with Randomized Radii Packed with Hyperbolic Monotone Algorithm (iterations).....	75
Figure 68 El with Andreev Radii Packed with Hyperbolic Newton's Method (factor).....	76
Figure 69 Hemisphere with Andreev Radii Packed with Hyperbolic Newton's Method (factor).....	76
Figure 70 Oval with Andreev Radii Packed with Hyperbolic Newton's Method (factor).....	77
Figure 71 El with Andreev Radii Packed with Hyperbolic Newton's Method (iterations).....	77
Figure 72 Hemisphere with Andreev Radii Packed with Hyperbolic Newton's Method (iterations).....	78
Figure 73 Oval with Andreev Radii Packed with Hyperbolic Newton's Method (iterations).....	78
Figure 74 Small Hex, Lh2, Packed with Hyperbolic Newton's Method (factor).....	79
Figure 75 Small Hex, Lh3, Packed with Hyperbolic Newton's Method (factor).....	79
Figure 76 Small Hex, Lh4, Packed with Hyperbolic Newton's Method (factor).....	80
Figure 77 Small Hex, Lh2, Packed with Hyperbolic Newton's Method (iterations).....	80
Figure 78 Small Hex, Lh3, Packed with Hyperbolic Newton's Method (iterations).....	81
Figure 79 Small Hex, Lh4, Packed with Hyperbolic Newton's Method (iterations).....	81
Figure 80 Cross with Randomized Radii Packed with Hyperbolic Newton's Method (factor).....	82
Figure 81 Hemisphere with Randomized Radii Packed with Hyperbolic Newton's Method (factor).....	82
Figure 82 Oval with Randomized Radii Packed with Hyperbolic Newton's Method (factor).....	83
Figure 83 Cross with Randomized Radii Packed with Hyperbolic Newton's Method (iterations).....	83
Figure 84 Hemisphere with Randomized Radii Packed with Hyperbolic Newton's Method (iterations)...	84
Figure 85 Oval with Randomized Radii Packed with Hyperbolic Newton's Method (iterations).....	84
Figure 86 Comparison of the Hyperbolic Algorithms with the Andreev El Packing.....	85
Figure 87 Comparison of the Hyperbolic Algorithms with the Randomized El Packing.....	85
Figure 88 Comparison of the Hyperbolic Algorithms with the Andreev Hemisphere Packing.....	86
Figure 89 Comparison of the Hyperbolic Algorithms with the Randomized Hemisphere Packing.....	86
Figure 90 Comparison of the Hyperbolic Algorithms with the Andreev Oval Packing.....	87

Figure 91 Comparison of the Hyperbolic Algorithms with the Randomized Oval Packing	87
Figure 92 Small Hex, Lh3, Packed with Euclidean Binary Algorithm (factor)	88
Figure 93 Small Hex, Lh4, Packed with Euclidean Binary Algorithm (factor)	88
Figure 94 El with Randomized Radii, Packed with Euclidean Binary Algorithm (factor).....	89
Figure 95 Hemisphere with Randomized Radii, Packed with Euclidean Binary Algorithm (factor).....	89
Figure 96 Oval with Randomized Radii, Packed with Euclidean Binary Algorithm (factor)	90
Figure 97 Small Hex, Lh2, Packed with Euclidean Monotone Algorithm (factor)	90
Figure 98 Small Hex, Lh3, Packed with Euclidean Monotone Algorithm (factor)	91
Figure 99 Small Hex, Lh4, Packed with Euclidean Monotone Algorithm (factor)	91
Figure 100 Small Hex, Lh2, Packed with Euclidean Monotone Algorithm (iterations).....	92
Figure 101 Small Hex, Lh3, Packed with Euclidean Monotone Algorithm (iterations).....	92
Figure 102 Small Hex, Lh4, Packed with Euclidean Monotone Algorithm (iterations).....	93
Figure 103 El with Randomized Radii Packed with Euclidean Monotone Algorithm (factor).....	93
Figure 104 Hemisphere with Randomized Radii Packed with Euclidean Monotone Algorithm (factor)...	94
Figure 105 Oval with Randomized Radii Packed with Euclidean Monotone Algorithm (factor).....	94
Figure 106 Small hex, Lh2, Packed with Euclidean Newton's Method Algorithm (factor).....	95
Figure 107 Small hex, Lh3, Packed with Euclidean Newton's Method Algorithm (factor).....	95
Figure 108 Small hex, Lh4, Packed with Euclidean Newton's Method Algorithm (factor).....	96
Figure 109 El with Randomized Radii Packed with Euclidean Newton's Method Algorithm (factor)	96
Figure 110 Hemisphere with Randomized Radii Packed with Euclidean Newton's Method Algorithm (factor)	97
Figure 111 Oval with Randomized Radii Packed with Euclidean Newton's Method Algorithm (factor)...	97
Figure 112 Comparison of Euclidean Binary and Newton's Method Algorithms, Small Hex Lh2.....	98
Figure 113 Comparison of Euclidean Binary and Newton's Method Algorithms, Small Hex Lh3.....	98
Figure 114 Comparison of Euclidean Binary and Newton's Method Algorithms, Small Hex Lh4.....	99
Figure 115 Comparison of Euclidean Binary and Newton's Method Algorithms, El.....	99
Figure 116 Comparison of Euclidean Binary and Newton's Method Algorithms, Hemisphere.....	100
Figure 117 Comparison of Euclidean Binary and Newton's Method Algorithms, Oval.....	100
Figure 118 Cross	101
Figure 119 El	101
Figure 120 Hemisphere	101
Figure 121 Oval	101
Figure 122 Small hex, Lh.....	102
Figure 123 Small Hex, Lh2	102
Figure 124 Small Hex, Lh3	102
Figure 125 Small Hex, Lh4	102
Figure 126 Flowgraph for Euclidean and Hyperbolic Algorithms.....	103
Figure 127 Flowgraph for Euclidean and Hyperbolic Monotone Algorithm (continued on next page)...	104
Figure 128 Flowgraph of Newton's Method Algorithms.....	106

1. Introduction

In 1985 during an invited talk at Purdue University on the occasion of the proof of the Bieberbach conjecture, William Thurston conjectured that maps between certain collections of mutually tangent circles can be used to approximate classical conformal mappings. This talk is sometimes referred to as the birth of circle packing. Thurston's conjecture was proven by Rodin and Sullivan in 1987. In addition to its visual appeal, circle packing has an experimental approach. Ken Stephenson capitalized on circle packing's experimental and visual appeal by creating CirclePack[©], an X-windows program which can be used to create, manipulate and view circle packings. CirclePack[©] utilizes Thurston's iterative approach to create an approximate circle packing for a given complex. Throughout this thesis the author will be working in both Euclidean and hyperbolic geometries.

A circle packing is a collection of circles with a specific pattern of tangencies that maintains a certain combinatoric pattern. This combinatoric pattern is encoded in an abstract simplicial complex K . (Figure 1) We call K a *cp-complex* if it meets the following conditions:

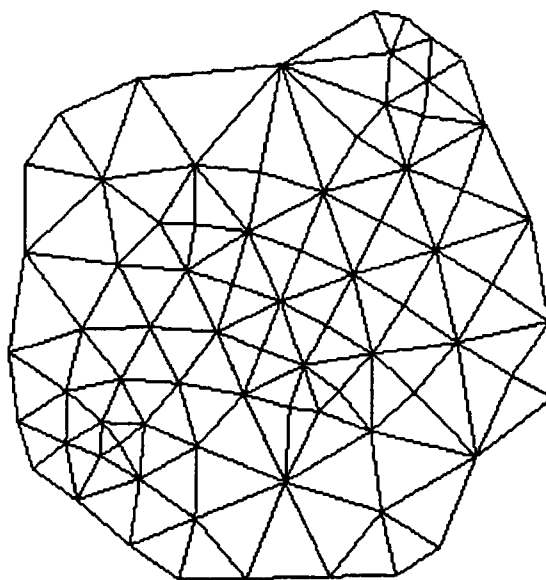


Figure 1 A Sample Representation of a Complex

1. K is simplicially equivalent to a triangulation of an orientable surface.
2. Every boundary vertex of K has an interior neighbor.
3. The set of interior vertices is edge connected, i.e. the complex is in one piece.

For each vertex v_j in K we can assign a value $r_j \in (0, \infty]$. If R is the set of all r_j , we call R a label for K . R can also be thought of as a radius function for K in a geometry (hyperbolic or Euclidean) because R associates a radius r_j to each vertex $v_j \in K$. $K(R)$ is a *labeled complex* within hyperbolic (or Euclidean) geometry.

Within a cp-complex, we are interested in some special sets of vertices. The set of three vertices where each vertex is the neighbor of the other two is called a *face*. The collection of faces at a given vertex v is called the *star* of v (Figure 2). If we look at a star within a circle packing, we have a flower (Figure 2.). Each of the neighboring tangent circles to the circle at v is called a *petal* of the flower. The number of vertexes in the star of v is called the *degree of v* , $\deg(v)$. The reader

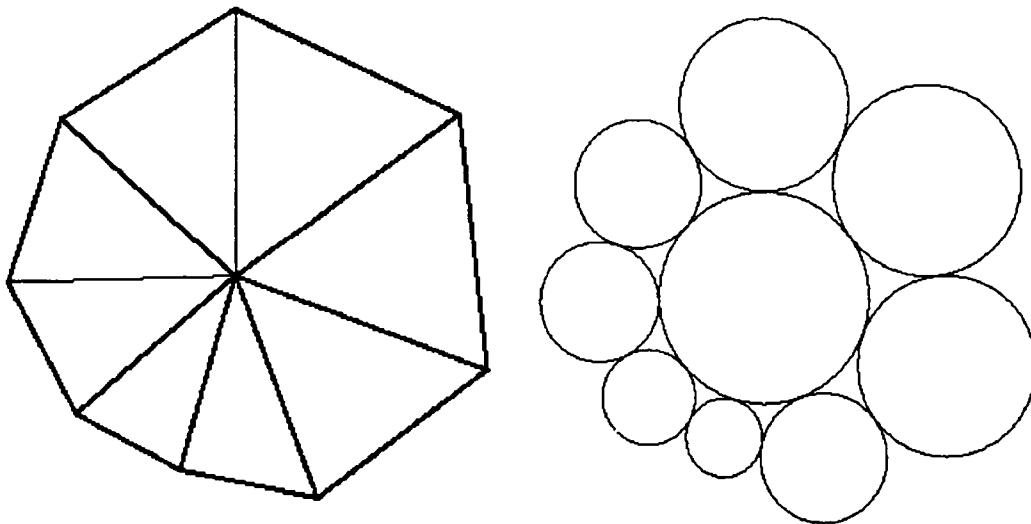


Figure 2 Sample Star and Flower

should note the $\deg(v)$ can vary for each $v \in K$. The *degree of a complex K*, $\deg(K) = \text{Max}(\deg(v) \forall v \in K)$. If for any v_i and $v_j \in \text{interior of } K$ then $\deg(v_i) = \deg(v_j)$ then we say K is of regular degree and $\deg(K) = \deg(v)$.

The combination of cp-complex K and label R is a circle packing P if when K is laid on a surface with circles of radius R it produces a collection of mutually tangent circles. Each complex K can have many labels associated with it. Some of these will be a packing label and $K(R)$ will be a circle packing, but many more of these will not. Figure 3 shows two different labels for the same complex. One is a packing label and results in a circle packing, while the other is only a label and does not result in a circle packing. This may create an additional circle packing, but often it will not. There is a wide variety of circle packings and cp-complexes available to study. Within this paper, we will look only at finite packings of constant degree 6.

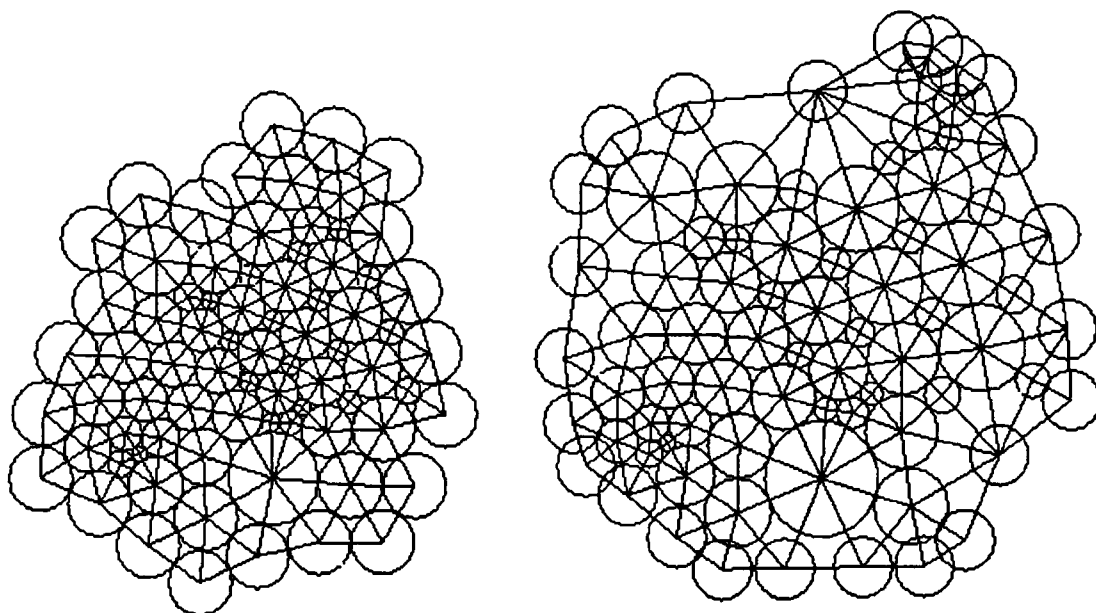


Figure 3 Two Different Labels on the Same Complex

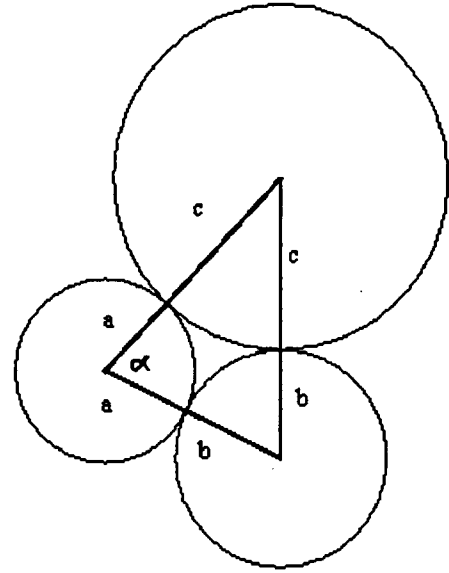
Given a cp-complex K and a Radius function R ,
if $K(R)$ is a circle packing P , then R is called a
packing label of K . If we look at a face within P
(Figure 4) we would like to find the angle α .
Using the law of cosines in Euclidean geometry
we have

$$\cos \alpha = \frac{(a+b)^2 + (a+c)^2 - (b+c)^2}{2(a+b)(a+c)}$$

or

$$\alpha = \cos^{-1} \left(\frac{(a+b)^2 + (a+c)^2 - (b+c)^2}{2(a+b)(a+c)} \right)$$

Figure 4 Ordered Triple of a Complex



for hyperbolic geometry we have

$$\alpha = \cos^{-1} \left(\frac{\cosh(a+b) \cosh(a+b) - \cosh(b+c)}{\sinh(a+b) \sinh(a+c)} \right)$$

We can define the **angle sum** at $v \in K$ to be the sum of the angles at v in all the faces of its star.

We denote this as $\Theta_R(v)$. Given a vertex v with neighbors $\{v_1, \dots, v_j\}$, then

$$\Theta_R(v) = \sum_{i=1}^j \cos^{-1} \left(\frac{(r+r_i) + (r+r_{i+1}) - (r_i+r_{i+1})}{2(r+r_i)(r+r_{i+1})} \right) \text{ for Euclidean geometry and}$$

$$\Theta_R(v) = \sum_{i=1}^j \cos^{-1} \left(\frac{\cosh(r+r_i) \cosh(r+r_{i+1}) - \cosh(r_i+r_{i+1})}{\sinh(r+r_i) \sinh(r+r_{i+1})} \right) \text{ for the hyperbolic case.}$$

In order to ensure we can achieve a circle packing P , we need necessary and sufficient conditions on R so the $K(R)$ corresponds to a circle packing P . To achieve this, we will use the following:

Theorem:

Suppose K is a simply connected cp-complex, a function R for K is the radius function for some circle packing P of K iff $\Theta_R(v) = 2\pi n(v)$ for every interior vertex $v \in K$ where $n(v)$ is a non-negative integer. P is uniquely determined up to rigid motions.

Throughout this paper, we will restrict our use of this theorem to $n(v)=1$. Circle Packings where $\Theta_R(v) = 2\pi$ are called *locally univalent*. These same experiments could be performed on non-locally univalent packings and the results compared with those presented in this paper.

In hyperbolic geometry we will look at a circle packing that fills as much of the unit disc as possible. This maximal packing of the unit disc for a locally univalent cp-complex K is called the *Andreev Packing* for K . The Andreev Packing is the packing $K(R_*)$, where $R_* = \infty$ at every vertex $v \in \partial K$.

We will utilize the following theorem (Dirichlet's Problem) with our work on the MasPar.

Theorem, Dirichlet's Problem:

Let K be a finite simply connected cp -complex. Let $g: \partial K \rightarrow (0, \infty]$ be defined on the boundary vertices. Then there exists a unique Radius function R for K $\ni K(R)$ is a locally univalent packing and $R|_{\partial K} = g$.

This thesis will utilize Thurston's iterative process to find an approximation for a packing label for a given complex. The necessary and sufficient conditions from the first theorem will be utilized to determine how close the approximation is to a packing label. The project will take a cp -complex and a label with a specified boundary condition and attempt to find the most efficient algorithm to achieve an approximate packing label.

2. The MasPar MP-2 Parallel Processor

This thesis looks at developing a circle packing algorithm to work on the MP-2, a massively parallel single instruction multiple execution (SIME) machine and modifying the algorithm to increase its efficiency. The MP-2 has a collection of 4096 processors in a physical 64x64 processor array. Each processor is physically connected to each of its eight nearest neighbors, N, NE, E, SE, S, SW, W and NW. The boundaries of the physical array are wrapped north to south and east to west to form a logical torus. The MP-2 has a front-end machine on which work can be done to prepare data for the Main Processor Array, MPA. Efficiency of the algorithms will be compared based on the amount of time an algorithm takes to create an approximate packing label.

The first hurdle to overcome with creating estimates of circle packings on the MP-2 is to get the complex and label data onto the MPA. To best utilize the power of the MP-2, we need to understand the types of communication that exist between two nodes of the MP-2. The fastest communication between two nodes occurs between two nodes that are connected in a primary direction, i.e. N, NE, E, etc. Communication can happen between random nodes on the MPA, but this communication is an order of magnitude slower than communication in a primary direction. The MPA has imbedded in its structure the complex for a regular packing of degree 8. While this could be utilized it is difficult to observe what happens around the boundary with a regular degree 8 packing (Figure 5). The author instead preferred to work with hexagonal packing since they display much better on a representation of a plane such as a computer monitor. This was accomplished on the MP-2 processor array by using a "lazy hex." The lazy hex was created by selecting a node and ignoring the connection between it and its NW and SE neighbor. If this

selection was viewed on the physical processor array, it would look like the hexagon was leaning on its right side, thus the name lazy hex (Figure 6). In actuality each of the neighbors is the same distance on the processor array, one hop. While the MP-2 allows utilization of more data sets than there are processors, in order to simplify the algorithm, potential packings were limited to non-overlapping complexes. This placed a bound on the size of the packings to around 3000.

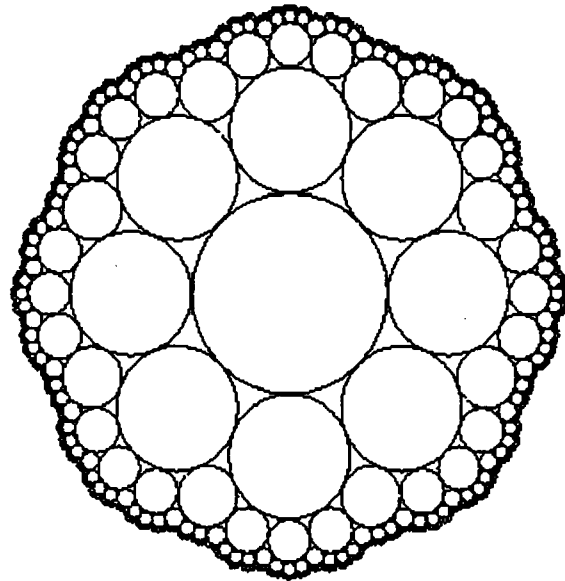


Figure 5 Regular Degree 8 Packing in Euclidean Geometry

The utilization of the lazy hex created the cp-complex for a regular cp-complex of degree 6 within the physical structure of the MPA. With this structure, we only need to load the label at each of the data points to have a labeled complex on the MPA. We next needed to create an algorithm to map the data from each vertex of a complex onto a physical processor of MPA. To do this we utilized a 64x64 array to represent the processor array. Information received from CirclePack^o utilizes the vertex number with a listing of the neighbors of that vertex. There is no correlation between a vertex number and any of its neighbors, (i.e. given two vertexes x and $x+1$, there is no guarantee if they are or are not neighbors). To get this information the complex data is run

through the **complex_count** routine from CirclePack[©] which identifies boundary and interior circles and identifies the faces and ordered triples of the complex.

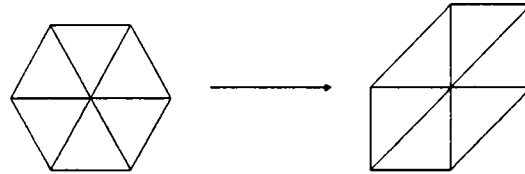


Figure 6 Lazy Hex

A modified version of **facedraworder** routine, also from CirclePack[©], is then used to draw the faces onto the

representation of the processor array. **Facedraworder** finds an ordered triple that contains the alpha vertex and plots the alpha vertex in the center, location (32,32) of the array. The second node of ordered vertexes containing the alpha vertex is plotted at location (31,32). Then the third vertex is plotted at (31,33). **Facedraworder** next finds an ordered triple where two of the vertexes have already been plotted and plots the third vertex of this ordered triple. This continues until each node is plotted using the face information calculated in **complex_count**. While it is realized the alpha vertex does not have to be in the center of the complex, its exact location only affects the size of the packing that can be utilized and has no impact on the algorithm. Once the node information is represented on the 64x64 array, it is ready to be sent to the main processor array. Note: Up to this point, all the calculations have been done on the front-end processor.

To pack a complex on the MP-2, the original algorithms that Ken Stephenson used in his CirclePack[©] program were utilized. These algorithms utilize an iterative process that is attributed to Thurston. In this iterative approach, the combinatorics remain fixed while the radii are changed. The angle sum is used to check on how closely we are approximating the packing label. The calculations of angle sum within the computer algorithms are called curvature, **curv**. Because we

are only approximating the packing label, the curvature describes how much our surface must be curved in order to hold the label. The MasPar is particularly suited to the types of iterative computations such as those used in the circle-packing algorithms. There is a concern that the parallel algorithm uses “old” data in the sense that each radii is being changed concurrently. Despite these concerns, the parallel algorithms are able to converge to a packing.

The MasPar is very efficient in calculating the error of a packing. The error of a vertex., **err**, is $|\Theta_R(v) - \text{curv}(v)|$. We want the error at each vertex to be less than **toler** which we set as our tolerance for error. The error of a packing, **maxerror**, is $\max |\Theta_R(v) - \text{curv}(v)| \forall v \in K$. In the non-parallel case, each node must be visited sequentially to calculate the error. As a result, the larger the number of nodes, the more calculations required to check the error condition. In the MasPar environment, one calculation provides the error for all the nodes on the processor array, **err** = $|\text{pe_aim} - \text{curv}|$ where **pe_aim** is the target angle sum and **curv** is the calculated curvature for the packing label. The MasPar environment also provides a maximum function to determine the largest value for error, **maxerr** = **reduceMaxf(err)**. The number of calculations and thus the time involved in calculating the error is independent on the number of nodes in the cp-complex.

The tradeoff with the MasPar is the increased amount of time needed to create the connection to CirclePack[©] and prepare the packing data for use on the MasPar. Also the MasPar is not suited for generalized packings and currently only works with regular hex packings.

3. Project Description

During this project, three algorithms were tested. They were 1. binary search, 2. monotone binary search and 3. Newton's method. Flow graphs of each of these algorithms can be found in Appendix B. At each iteration, the binary search finds the weighted average of two estimates of the radius, provided the radius is not "too large" or "too small." The monotone binary search only changes the radii that are too long by shortening them. This creates a super-packing. Once it creates this super-packing, it will begin to lengthen radii until a packing is reached. The same weighted average from the binary search is used in the monotone binary search algorithm. Since all the adjustments to the radii are made in the same direction, there is little chance of oscillation not allowing the cp-complex to pack. The Newton's method utilizes Newton's method to find the next estimate.

For a given complex, the maximum error (tolerance) changes to certain variables were watched to see how they affected the performance of the packing algorithm. The performance of the algorithm was calculated by comparing its total run time to produce a packing. Changes were made to the **factor** and **iterations**. The **factor** $\in (0,1)$ determines how much each radii can be raised or lowered on each pass. The **iterations** is a positive integer that is the number of times the radii are adjusted before a check against the tolerance is made. Tolerance is used to determine our stop condition. There is also an upper bound on the number of passes. If this number of passes is reached without reaching the tolerance condition, the program halts without reaching a packing. Changes were made in the **factor** and the **iterations** for a given complex and error. The **factor** is used to ensure the program does not make drastic changes in a given radius. If a change is too drastic, it could

create a higher error by increasing the error of the neighboring vertices. If the change is too minor, i.e. a small value of **factor**, it will take the algorithm longer to converge to a packing. Some time is lost when checking the tolerance to determine if the packing meets the tolerance condition. The **iterations** variable was implemented to attempt to minimize this time loss. By increasing the number of **iterations**, we reduce the number of checks against the error condition. We want to find optimum for the **iterations** and the **factor** for a general 6 degree packing.

For the optimization experiments, the author selected a series of packings consisting of a small hex, a cross, an oval, an el, and a hemisphere. These packings were created as Euclidean packings (see Appendix B) and converted to hyperbolic packings through CirclePack^o. Each of these packings, both Euclidean and hyperbolic, was given a set of boundary radii. For the small hex, four different sets of boundary radii were used. These were named lh, lh2, lh3, and lh4. Packed versions of these packings can be seen in Appendix B. The author utilized Andreev radii for each of the cross, oval, el, and hemisphere as one set of boundary conditions. Also, each of these packings was run through a randomizer. The randomizer is a program the author wrote to change the radii in a given packing. The randomizer generates two random numbers for each radius. The first random number determines the type of change, increase or decrease, while the second random number determines the amount of change each radii would receive. These packings were run through the randomizer for three iterations, i.e. the results from the first iteration was submitted to the randomizer a second time and this result was submitted to the randomizer a third time. This created a new set of boundary conditions, while also creating new values for internal radii as well. This result of the third pass through the randomizer was used in some of the experiments described in this paper.

To facilitate gathering of timing data on the various packings, the front-end program, mptime2.c, was modified to read packing data from a file instead of acquiring the data from CirclePack^o. Similarly, the data was output to a file as well. The efficiency of the algorithms was judged using a timing mechanism with the programs. After the data was read into the main processor array on the MasPar, the program read the system time and stored the information in a variable. After the program performed its computations and the halt condition was met, the program checked the system time again and calculated the difference. This information was then sent to the standard output for the program and the data was returned to the front end processor. To gather the data needed, the packings were run in a Unix script file which ran the packings through one of the three algorithms and gathered the time data. Microsoft Excel was used to process this data into graphs to assist with the analysis of the data. A limit of 5000 passes was placed on each algorithm. If the algorithm did not create an approximate packing label within 5000 passes, a packing time of 0 was returned.

4. Hyperbolic Binary Algorithm

The experimentation was done in steps. The first step took the cross, hemisphere, el and oval Euclidean packings, converted them to hyperbolic packings, and created Andreev packings for each. Each of these packings was packed with a common tolerance of .0001. The number of iterations was increased from one iteration to forty-one iterations with a step increase of five while the factor was increased from 0.1 to 0.9. Analysis of this data was accomplished by creating a series of graphs. Two series of graphs were compiled for each packing. The first series of graphs utilized the factor as the independent variable and the time to reach a packing was the dependent variable. A graph was created for each of the iteration settings. The second series of graphs utilized the iterations as the independent variable and the time to reach a packing was the dependent variable. A graph was created for each value of factor. We want to look at the first series of graphs for the cross packing which can be seen in Figure 7. The reader should notice that this graph is optimized for a factor greater than 0.9. This is to be expected because with an Andreev packing, all of the boundary radii must be increased to reach the optimum. Similarly, each of the interior circles will typically be increasing as well. The closer the factor is to 1, the more the radii are allowed to change with each iteration. Because the Andreev packings tend to expand each radius, we would expect a larger factor to produce a faster packing. We can observe a similar behavior in graphs of the el, hemisphere and oval packing which can be seen in Figure 39 through Figure 41 in Appendix A.

The previously mentioned graphs give us some insight into how the number of iterations affect the efficiency of our packing algorithm. The reader should notice that as the number of iterations is

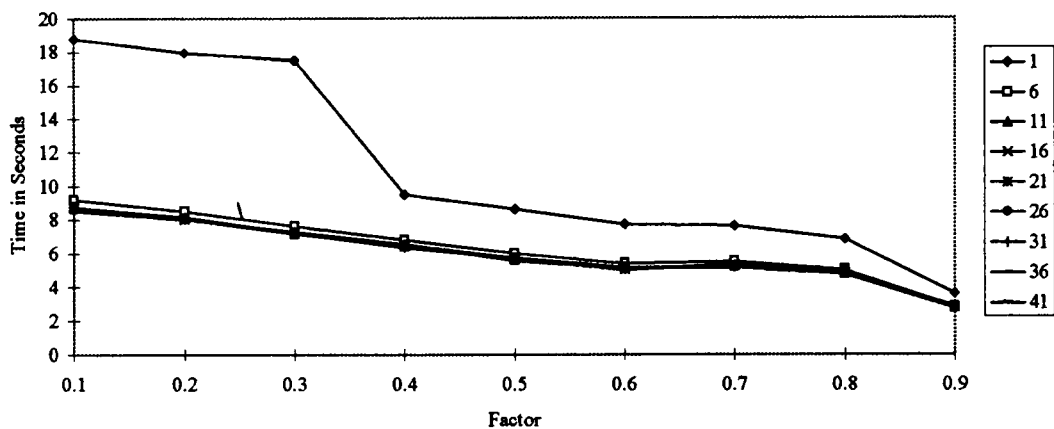


Figure 7 Cross with Andreev Radii Packed with Hyperbolic Binary Algorithm (factor)

increased, there is little effect on the algorithm after 6 iterations. We can look at a graph of the cross with the iterations as the independent variable. This graph can be seen in Figure 8. We can now see that an increase in the number of iterations has little effect on the efficiency of the hyperbolic binary algorithm with the cross packing. Graphs of the el, hemisphere, and oval packing with the number of iterations as the independent variable can be found in Appendix A, Figure 41 through Figure 43. The reader should notice that the el, hemisphere, and oval Andreev packings behave in a similar manner to the cross Andreev packing.

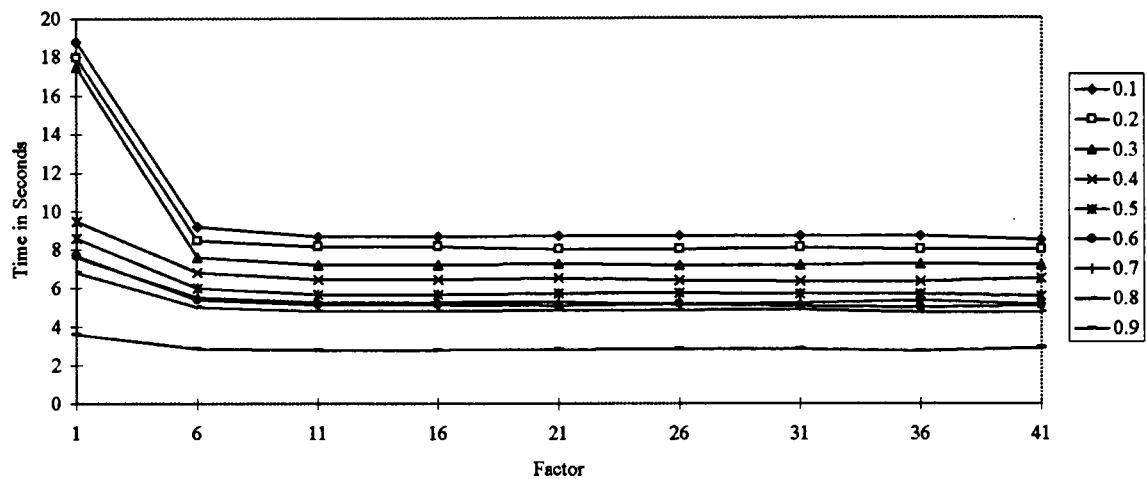


Figure 8 Cross with Andreev Radii Packed with Hyperbolic Binary Algorithm (iterations)

The results obtained for the Andreev packings are interesting, but Andreev packings are very special. Can these same results be generalized to other hyperbolic packings? To look at this, we will first look at a small hex packing with four sets of boundary radii. These packings will be called, lh, lh2, lh3 and lh4. The graph of the data gathered for the lh packing with the factor as the independent variable can be found in Figure 9. The reader should notice that the number of iterations still has little effect after a small number of iterations. This can be confirmed by examining the graphs of the small hex packing with boundary radii conditions lh2, lh3, and lh4 which can be found in Appendix A, Figure 45 through Figure 47.

For each of these graphs the number of iterations seems to have little impact after 6 iterations. A similar pattern is not as obvious for the factor. Two of the packings, lh3 and lh4, seem to have

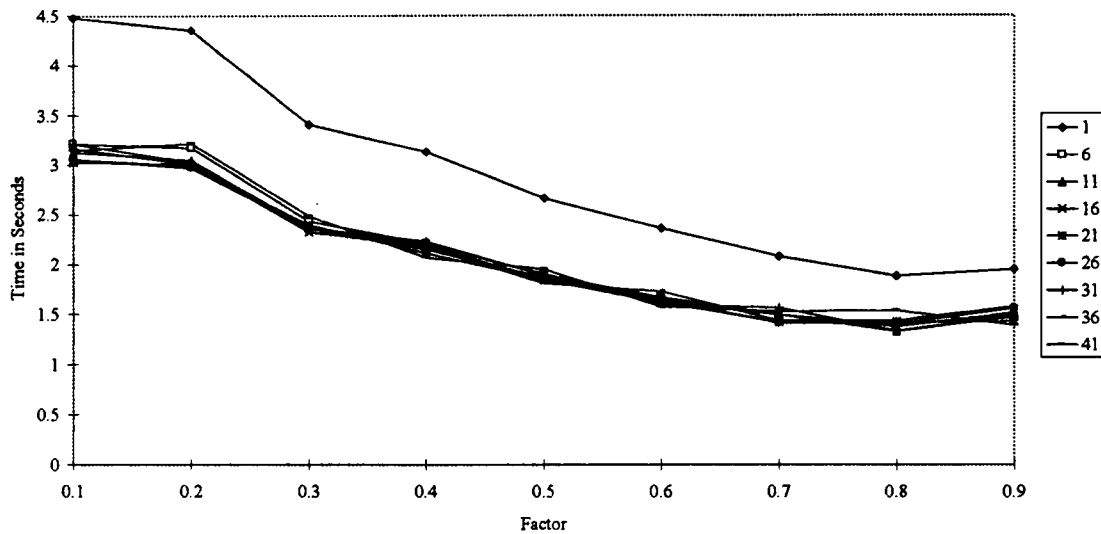


Figure 9 Small Hex, Lh, Packed with Hyperbolic Binary Algorithm (factor)

definite peaks in packing performance at factors of .9 and .8 respectively. Packings lh and lh2 do not have as distinct best values for factor. While the best value for factor is not as distinct with the lh and lh2 boundary radii conditions, both of these packings slowly reach a best factor around 0.8 and 0.9 respectively. This leads the author to conclude that perhaps the best value for the factor is closer to one.

To continue this analysis, the author looked at the final set of packings which are the randomized packings. These graphs were packed with the hyperbolic binary algorithm. The graph of the cross with randomized radii can be found in Figure 10. The reader should notice that as with the previous packings we looked at, the number of iterations has little effect for iterations greater than 6. For the iterations higher than 6, the graphs seem to converge to a common graph. Graphs

of the el, hemisphere, and oval are included in Appendix A, Figure 48 through Figure 50 so the reader can confirm these observations hold for these packings as well.

To conclude this section, the author provides the following observations of the hyperbolic binary algorithm:

1. The number of **iterations** has little effect on the efficiency of the hyperbolic binary algorithm after a small number of **iterations** is reached.
2. For Andreev packings, a high value of **factor**, 0.9 in our experiments, gives the most efficient packing time.
3. Even for more generalized packings, higher values of **factor**, > 0.7 , give the best packing times.

Because in the hyperbolic binary algorithm we are changing all of the radii in a complex at the same time, we could reach a condition where the radii begin to oscillate between values that are not within the range of the desired **tolerance**. While the author did not experience this oscillation in his experimentation, it is still a potential problem which needs to be addressed. In an effort to address this, the author modified the hyperbolic binary algorithm to only adjust the radii in one direction at a time. This algorithm is called the hyperbolic monotone algorithm and will be analyzed in the next section.

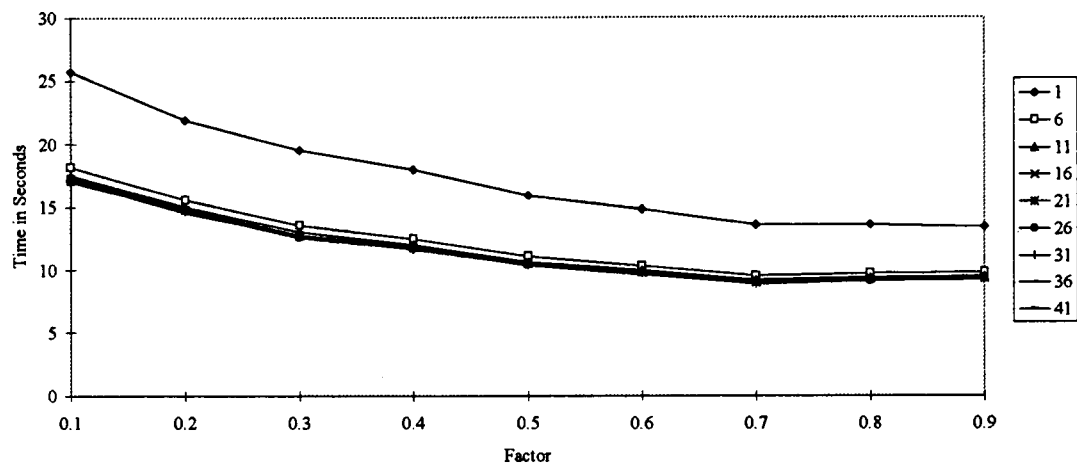


Figure 10 Cross with Randomized Radii Packed with Hyperbolic Binary Algorithm (factor)

5. Hyperbolic Monotone Algorithm

The hyperbolic monotone algorithm uses the same general algorithm as the hyperbolic binary algorithm. The difference between the two algorithms is the hyperbolic binary algorithm only makes changes to the radii in one direction at a time. Because the radii are being adjusted in only one direction at a time, there is less chance of reaching an oscillatory state. In the hyperbolic binary/Newton's Method algorithm, oscillation could possibly result from a change increasing one set of radii that could be offset by a decrease in another set of radii. This oscillation could occur because all the changes are being made concurrently. Because the hyperbolic monotone algorithm only makes changes in one direction, no oscillation is possible. The Andreev packings using the hyperbolic binary algorithm from the last section are similar to the hyperbolic monotone algorithm in that the Andreev packings only increase the radii. Will the packings with the hyperbolic monotone algorithm be as predictable as the Andreev packings from the last section? We will begin by looking at the Andreev packings.

The author ran his analysis of the hyperbolic monotone algorithm in the same general manner as with the hyperbolic binary algorithm. The author first examined the Andreev packing of the cross, el, hemisphere and oval. Next he looked at the small hex packings with four sets of boundary conditions and finally he looked at the randomized packings of the cross, el, hemisphere and oval.

The author begins his analysis by looking at the data obtained from packing the cross with Andreev radii using the hyperbolic monotone algorithm. Figure 11 portrays this data in a graph with the

factor as the independent variable and packing time as the dependent variable. Two things can be readily noticed from this chart. First, this graph depicts a dependence on the number of iterations. The reader should recall that the cross with Andreev radii packed with the hyperbolic binary algorithm is not affected by changes in the number of iterations after a small number of iterations (Figure 12.). However, a graph of the cross with Andreev radii packed with the hyperbolic monotone algorithm, shows much more of a dependence on the number of iterations. As the number of iterations increases, the time to reach a packing also increases.

The second item that is readily noticed is that the hyperbolic monotone algorithm is much less stable than the hyperbolic binary algorithm. This instability is noticed by peaks and jumps observed in Figure 11. It should also be noted that for one iteration the two algorithms have very close packing times. If you look at the data behind the graphs, we can see that at some points, the hyperbolic monotone algorithm performs better than the hyperbolic binary algorithm; however, this is the exception rather than the rule. In general, for the Andreev cross packing, the hyperbolic binary algorithm is more efficient than the hyperbolic monotone algorithm. Another observation

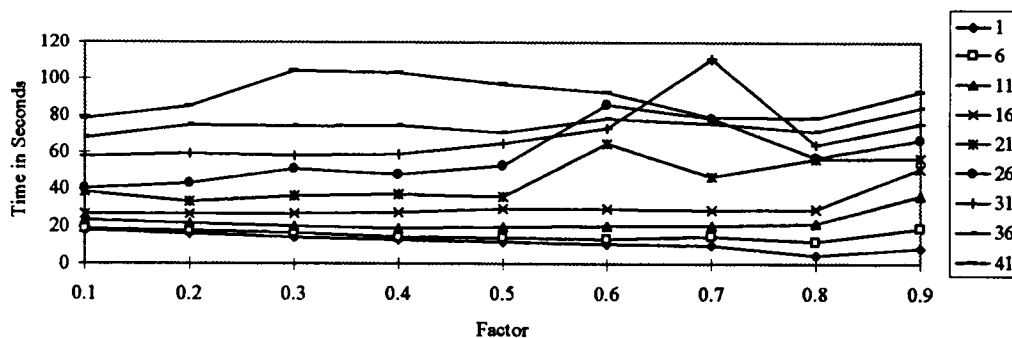


Figure 11 Cross Andreev Packing Packed with Hyperbolic Monotone Algorithm (factor)

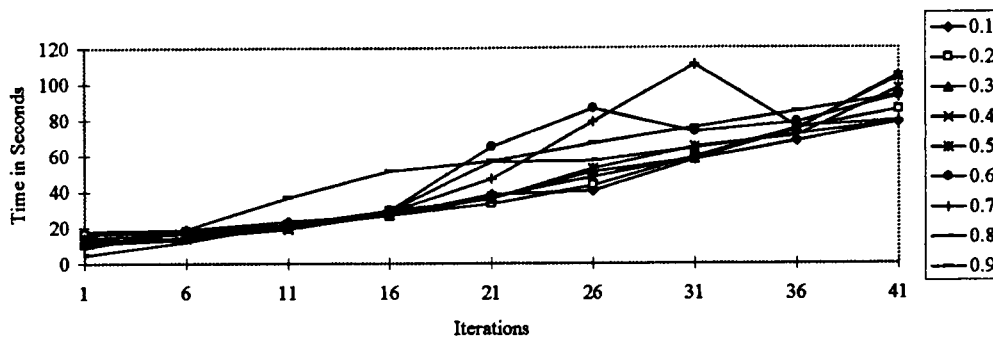


Figure 12 Cross Andreev Packing Packed with Hyperbolic Monotone Algorithm (iterations)

is the exception rather than the rule. In general, for the Andreev cross packing, the hyperbolic binary algorithm is more efficient than the hyperbolic monotone algorithm. Another observation we can make from Figure 11 is that for a given **factor**, as the number of **iterations** increases, the packing time also increases. This can be better observed if we look at a graph of the data with the **iterations** as the independent variable as in Figure 12.

From this graph, we can better see the trend that as the **iterations** increase, the packing time increases also. Also from this graph, the reader should again notice that the graph of the hyperbolic monotone algorithm seems more random than the graphs of the hyperbolic binary algorithms. Where the hyperbolic binary algorithms created smooth curves, the hyperbolic monotone algorithm has spikes that show the algorithm is not as predictable in the Andreev packing case as the hyperbolic binary algorithm was. The graphs for the el, hemisphere and oval with both the **factor** as the independent variable and the **iterations** as the independent variable can be found in Appendix A, Figure 51 through Figure 56. By looking at these additional graphs, the

reader can confirm the observations the author made with the cross Andreev packing hold for these packings as well.

We will now look at the small hex packings to observe their behavior when packed with the hyperbolic monotone algorithm. We will first look at the small hex packing with boundary condition lh. A graph of this packing with the **factor** as the dependent variable can be found in Figure 13. We observe again that this graph has a similar dependency on the number of **iterations** that we observed with the Andreev packings. By looking at a graph of the lh packing with the number of **iterations** as the independent variable, we can observe the trend of the packing time increases as the number of **iterations** is increased. This graph can be seen in Figure 14.

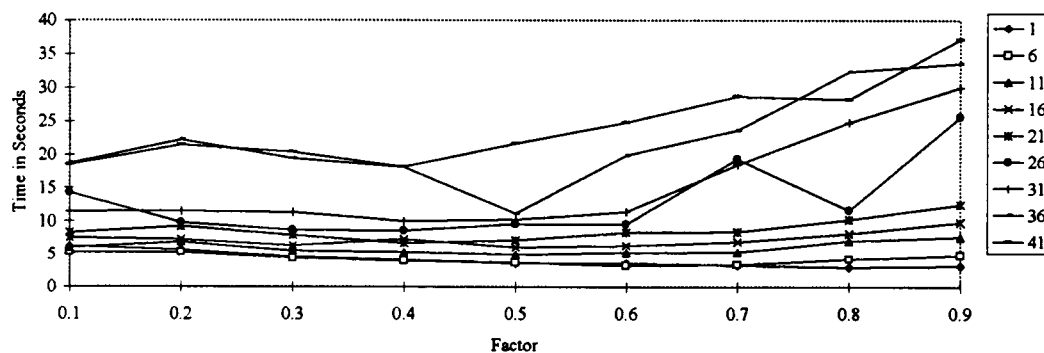


Figure 13 Small Hex, Lh, Packed with the Hyperbolic Monotone Algorithm (factor)

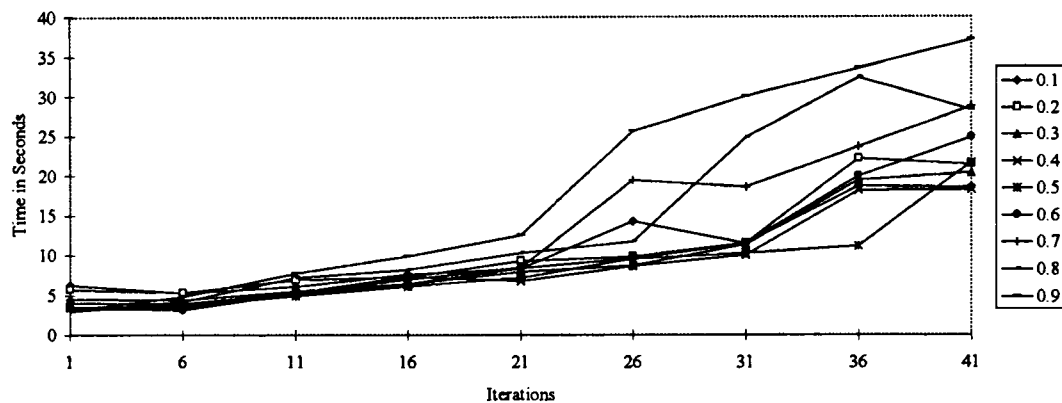


Figure 14 Small Hex, Lh, Packed with the Hyperbolic Monotone Algorithm (iterations)

Graphs of the packing for the small hex packing with boundary conditions lh2, lh3, and lh4 can be found in Appendix A, Figure 56 through Figure 61. As the reader can confirm, each of these has some similar characteristics. Each of these packings has a dependency on the number of iterations. As the number of iterations increases, the packing becomes very inefficient for large values of iterations. If we compare the hyperbolic monotone algorithm to the hyperbolic binary algorithm, we notice that for small values of iterations, the values are similar although they are not as close as with the Andreev packings. Upon further comparison of the hyperbolic monotone and hyperbolic binary algorithms, the reader should notice that for small values of iterations the best choice for factor in the hyperbolic monotone case is 0.8 or 0.9 for any of the four packings. This is the same best value that we saw for these packings with the hyperbolic binary algorithm.

The final collection of packings we will look at with this algorithm are the packings with randomized radii conditions. A graph of the cross with randomized radii with the **factor** as the independent variable can be found in Figure 15. We notice some of the same observations that we made with the Andreev packings and the small hex packings, namely the number of **iterations** has more of an effect on the hyperbolic monotone algorithm than it did with the hyperbolic binary algorithm. With the Andreev packings, the hyperbolic monotone algorithm packing times were close to the hyperbolic binary algorithm packing times for one **iteration** and small values for **factor**. With the randomized cross packing, this is not the case. The packing time of the hyperbolic monotone algorithm for randomized cross is almost twice that of the hyperbolic binary algorithm. This can be observed in the other random radii packings which can be seen in Appendix A, Figure 62 through Figure 67. This is expected since in the Andreev case the boundary radii are all expanding to infinity. This creates a monotone expansion on the interior radii as well. With the

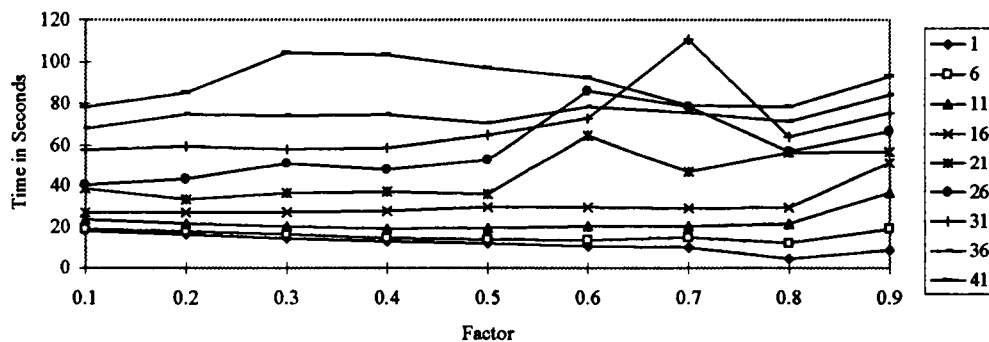


Figure 15 Cross Andreev Packing Packed with Hyperbolic Monotone Algorithm (factor)

randomized cross, the boundary radii are a mix. Some of the boundary radii were increased while other were decreased. This created an inefficiency within the hyperbolic monotone algorithm that lead to the dramatic increase in the packing time.

To conclude this section, the author provides the following observations of the hyperbolic binary algorithm:

1. The number of **iterations** has a direct effect on the packing. An increase in the number of **iterations** will increase the packing time.
2. Small values for the number of **iterations** provide the best packing times.
3. The packing times are best for **factor** > 0.8 .
4. Compared to the hyperbolic binary algorithm, the hyperbolic monotone algorithm is much less effective.

6. Hyperbolic Newton's Method

The next algorithm the author chose to look at is the hyperbolic Newton's method algorithm. The code for the hyperbolic Newton's method algorithm can be found in Appendix D. This algorithm utilizes Newton's method to find the best fit for the radii with a modification to attempt to ensure convergence. As we will see, this is not always successful. The analysis for the hyperbolic Newton's method algorithm was accomplished in the same manner as the other algorithms. The author first looked at the Andreev packings of the cross, el, hemisphere and oval. Next we look at the small hex packing with four different boundary conditions, lh, lh2, lh3, and lh4. Finally we will look at packings of the cross, el, hemisphere and oval with randomized radii.

A graph of the Andreev cross packing with the **factor** as the independent variable and the time in seconds as the dependent variable can be found in Figure 16. Two items are quickly noticeable from these graphs. First there are no values for **factor** below .5. When this algorithm was run with a **factor** below .5, the algorithm did not produce a packing in 5000 passes through the algorithm. The second observation is the graphs for each of the different values for **iterations** varies very little. As was the case in the hyperbolic binary algorithm, packing time is not dependent on the number of **iterations**. Figure 17 depicts a graph of the **factor** for values above 0.5 with the number of **iterations** as the independent variable. The reader should make particular note of the scale used for the packing time. The time to reach a packing has only minimal

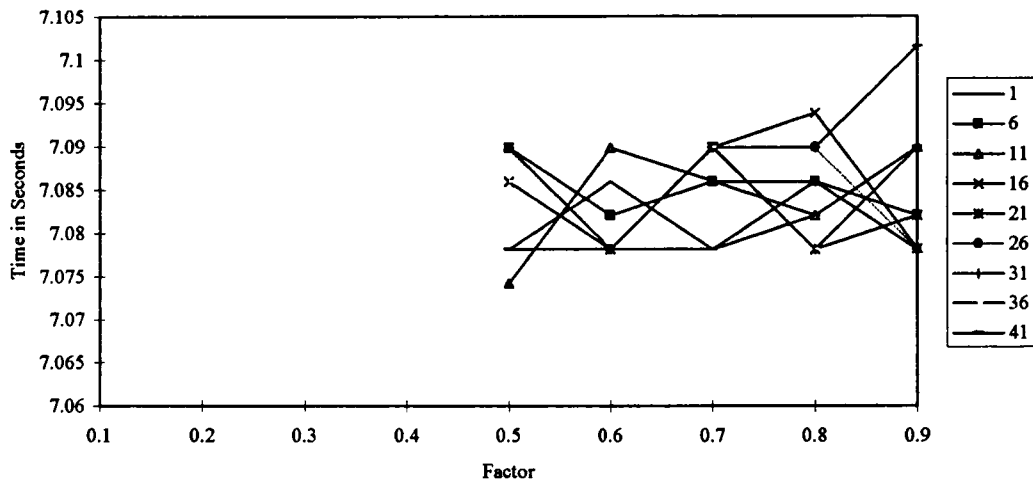


Figure 16 Cross with Andreev Radii Packed with Hyperbolic Newton's Method Algorithm (factor)

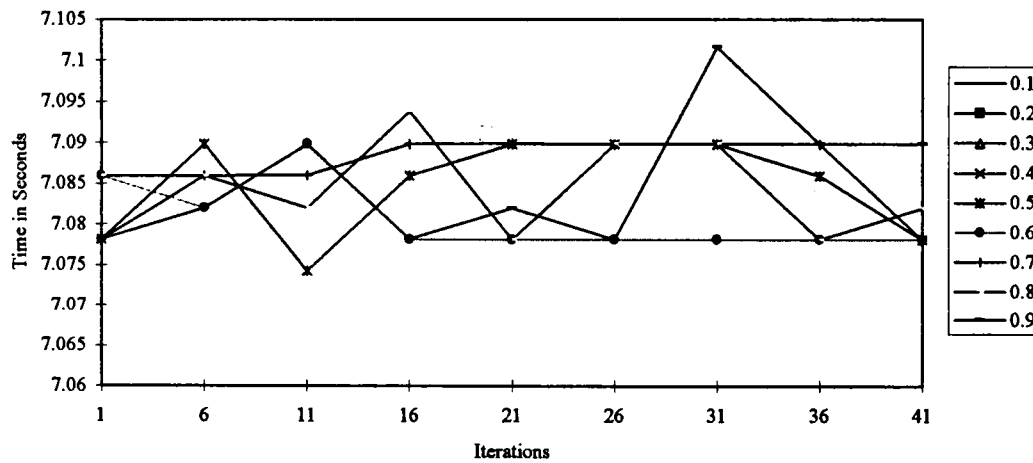


Figure 17 Cross with Andreev Radii Packed with Hyperbolic Newton's Method Algorithm (iterations)

differences for changes in the **factor** and **iterations**. From these graphs, we observe that to obtain the best packing time for an Andreev packing the number of **iterations** is immaterial and the **factor** need only be greater than 0.5. These observations can be confirmed by looking at the packings of the el, hemisphere and oval which can be found in Appendix A, **Figure 68** through **Figure 70**.

To continue the analysis, we look at a small hex packing, lh. **Figure 18** depicts a graph of the **iterations** with the **factor** as the independent variable. This graph demonstrates the same behavior observed with the Andreev packings. For this complex, the number of **iterations** has no effect on the efficiency of the algorithm and the **factor** need only be greater than 0.5. **Figure 19** shows

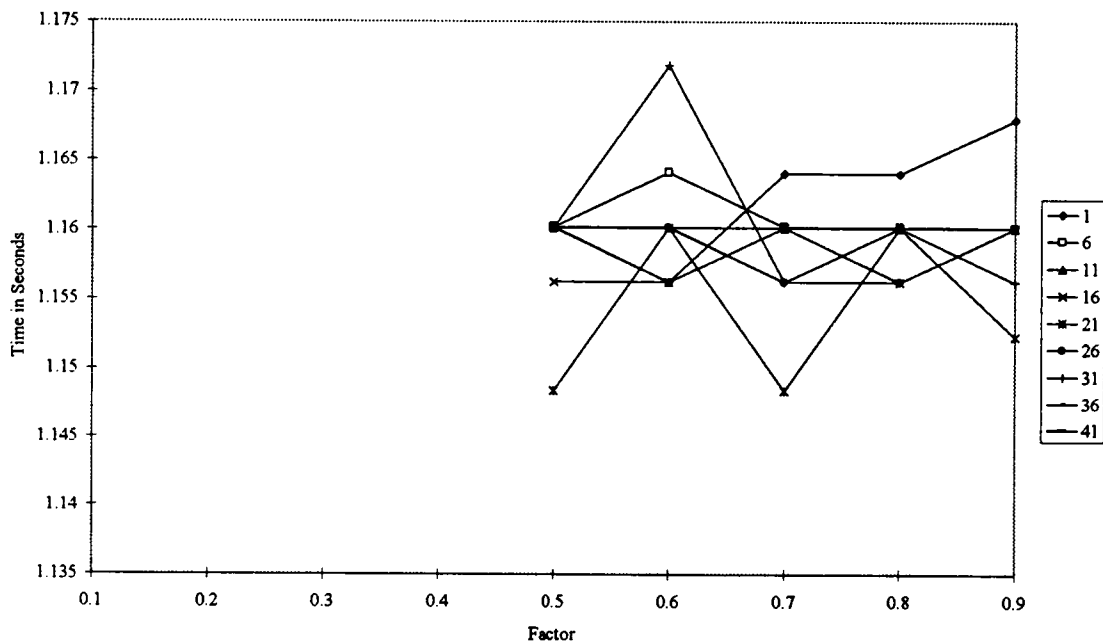


Figure 18 Small hex , Lh, Packed with Hyperbolic Newton's Method Algorithm (factor)

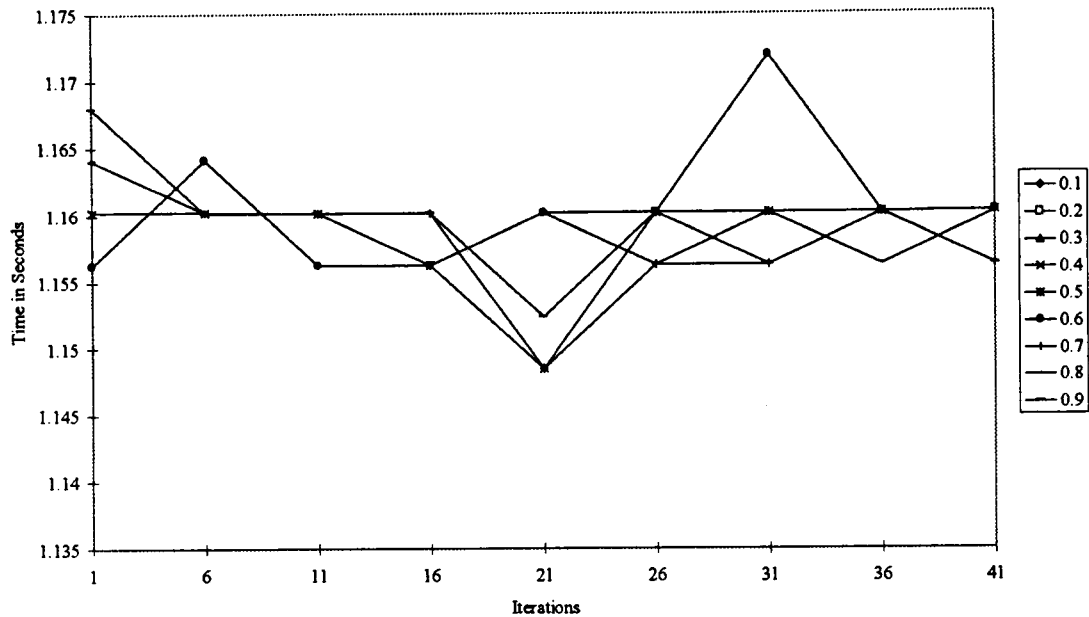


Figure 19 Small hex , Lh, Packed with Hyperbolic Newton's Method Algorithm (iterations)

graphs of the values for **factor** > 0.5 with the **iterations** as the independent variable. The reader can see there are only minor variations in the packing time for different values of **iterations**. These observations can be confirmed by looking at the remaining small hex packings. These graphs can be seen in **Figure 74** through **Figure 79** in Appendix A.

To conclude the analysis of the Hyperbolic Newton's method algorithm, the author will now explore the random packings. A graph of the el packing with the **factor** as the independent variable can be found in Figure 20. The reader will notice from this graph that there is a jump in the packing time as the **factor** becomes .6 and larger. This jump is in addition to the observations we noticed with the Andreev packings and the small hex packings. In the hyperbolic case, we are

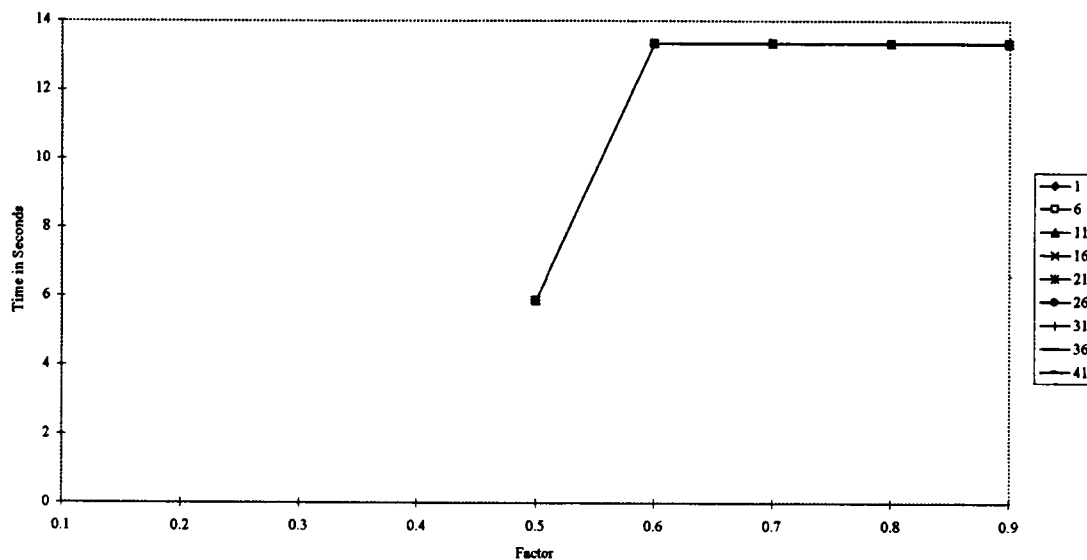


Figure 20 El with Randomized Radii, Packed with Hyperbolic Newton's Method Algorithm (factor)

using the s-radius to simplify the computations in the hyperbolic algorithms. The s-radius= e^{-r} where r is the hyperbolic radius has an upper limit of 1. The hyperbolic Newton's method algorithm checks the s-radius to ensure that it does not go over 1. Having the s-radius greater than 1 creates an error condition in the program that represents a divergent condition. Placing the bound on the s-radius allows the packing to converge, but at a slower rate. This explains the increase in the packings for a factor of .6 and higher. This same result can be seen in the hemisphere and oval packings which can be seen in **Figure 81** and **Figure 82** in Appendix A. The cross packing, Appendix A **Figure 80**, does not display this behavior. It behaves in the same manner as the small hex packings and the Andreev packings.

To conclude this section, the author makes the following observations for the hyperbolic Newton's method algorithm:

1. The algorithm diverges for values of factor < 0.5 .
2. The packing time seems to be dependent only on the algorithms convergence. Changes in the factor and iterations have only minimal impact on the packing time.

7. Comparison of Hyperbolic Algorithms

We will conclude our discussion of the hyperbolic packings by attempting to determine which algorithm produces the packing label in the fastest time. We will limit our comparison to the hyperbolic binary algorithm and the hyperbolic Newton's method algorithm. We have eliminated the hyperbolic monotone algorithm as our previous study has shown this algorithm is much less efficient than the hyperbolic binary algorithm. To assist with the analysis of these two algorithms, we will look at a graph with the number of iterations as the independent variable and the time as the dependent variable. We will graph the value for factor which creates the fastest convergence for the hyperbolic binary algorithm and the hyperbolic Newton's method algorithm. (Note: in the case of the hyperbolic Newton's method algorithm we will always use a factor of .5.) We will first compare the packings for the cross with Andreev radii, Figure 21, and the cross with randomized radii, Figure 22. Notice that with the Andreev packing, the hyperbolic binary algorithm creates the fastest packing. But with the random packings, the Newton's method algorithm creates the fastest packings. Will this hold for the other packings? We can look at graphs of the el, hemisphere, and oval for both the Andreev packings and the random radii packings. These can be found in Appendix A, Figure 86 through Figure 91. Notice that in each of the Andreev packings, the hyperbolic binary algorithm is more efficient than the hyperbolic Newton's method algorithm. Also notice that on the random radii packings, the hyperbolic Newton's method packing is more efficient than the hyperbolic binary packing except in the case of the oval packing.

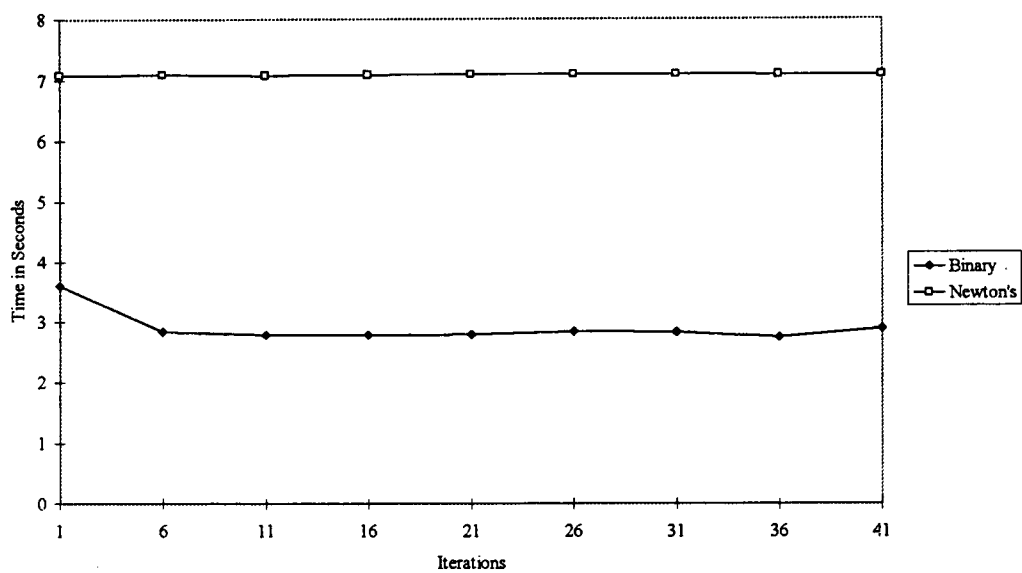


Figure 21 Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Andreev Cross Packing

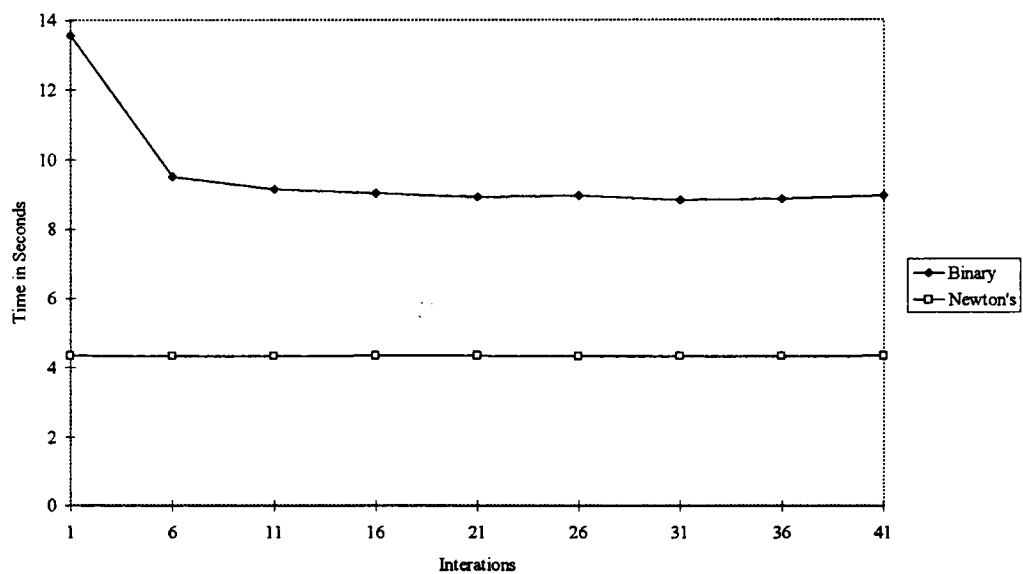


Figure 22 Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Randomized Radii Cross Packing

Let's continue this exploration by looking at the small hex packings. Graphs of the hyperbolic binary and Newton's method packing times graphed with the number of iterations as the independent variable can be found in Figure 23 through Figure 26. We notice that the lh and the lh2 packings are packed more efficiently with the hyperbolic Newton's method algorithm while the lh3 and the lh4 packings are packed more efficiently with the hyperbolic binary algorithm. Lh3 and lh4 behave more like the Andreev packings. We might assume these two packings are similar to the Andreev packings in that the boundary circles are generally larger than the interior circles. We can check this by looking at the pictures of the packing in Appendix B, Figure 122 through Figure 125. By comparing the four small hex packings, it is difficult to come to a conclusion why there is a difference between the packings. Furthermore, if we reduced all the boundary radii, which algorithm would produce the best packing time? The answer to this question lies within the two algorithms. Newton's algorithm places an upper bound on any radii change of $(1 + \text{old_radius}) * \text{factor}$ and a lower bound of $\text{old_radius} * \text{factor}$, while the binary algorithm places an upper bound of $1 - \text{factor} + \text{old_radius} * \text{factor}$ and a lower bound of $\text{old_radius} * \text{factor}$. This difference in the upper bound for the new radii accounts for why the binary algorithm out performs the Newton's method algorithm for Andreev packings and other packings where the general trend on a radii is increasing. It may also account for the trend of the binary algorithm to favor factors closer to one.

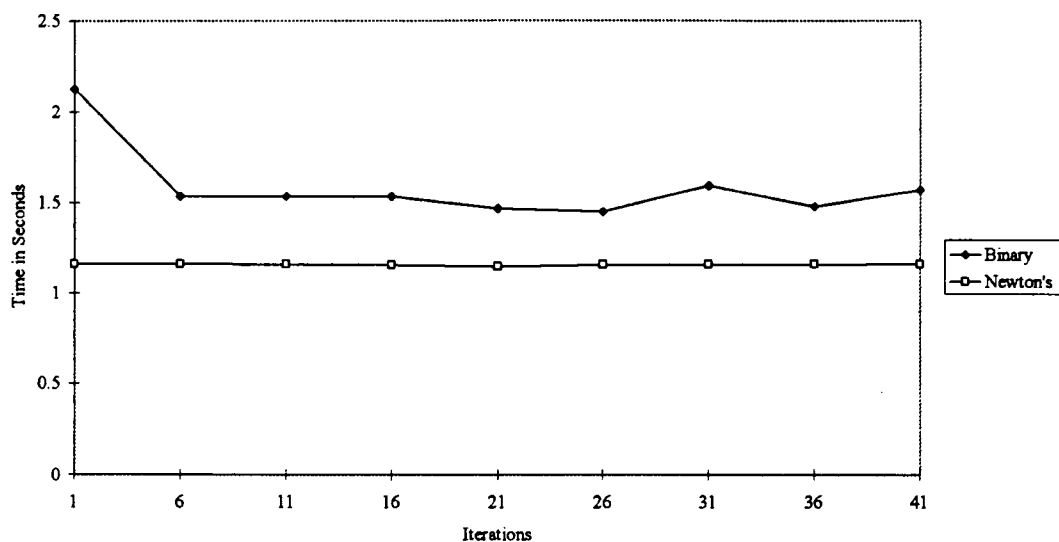


Figure 23 Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Small Hex Packing Lh

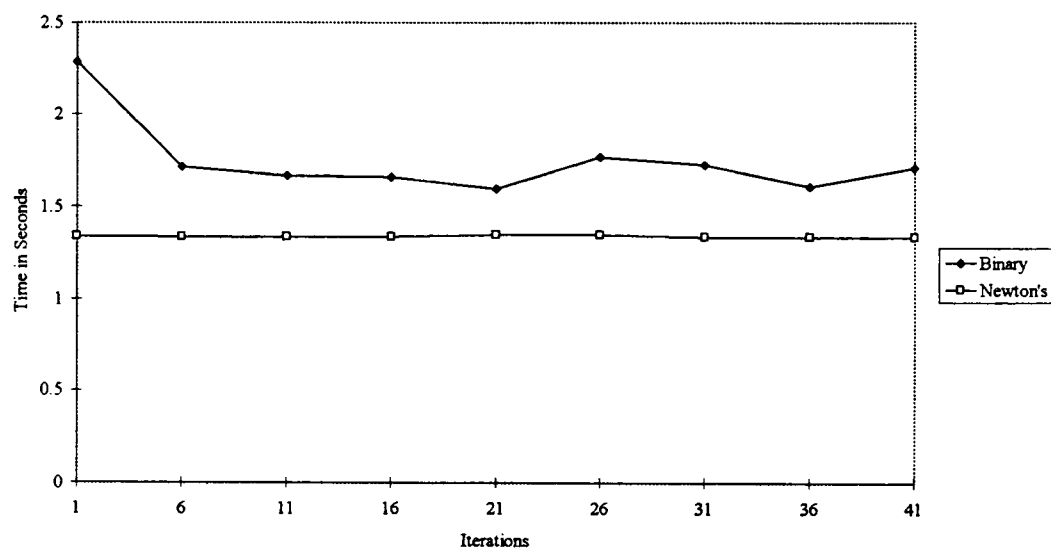


Figure 24 Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Small Hex Packing Lh2

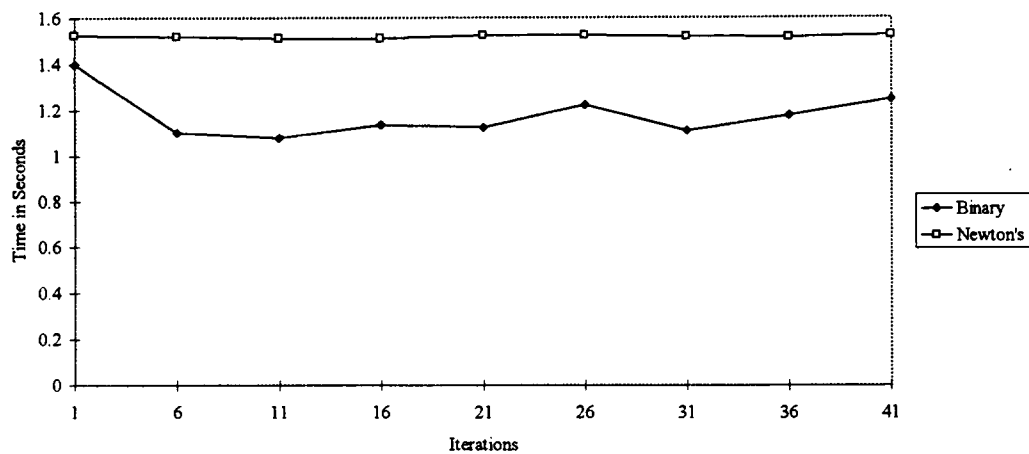


Figure 25 Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Small Hex Packing Lh3

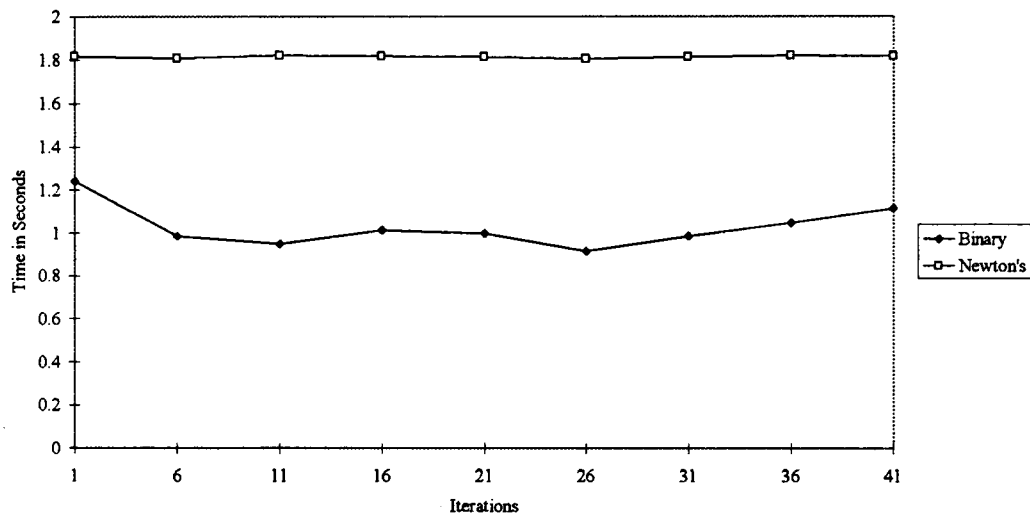


Figure 26 Comparison of Hyperbolic Algorithms, Binary and Newton's Method for the Small Hex Packing Lh4

8. Euclidean Binary Algorithm

We will follow the same pattern in analyzing the Euclidean packings as we did the hyperbolic packings except we will not be using Andreev radii as the Andreev radii are unique to the hyperbolic setting. We will begin by looking at our four small hex packings and then study the four packings with randomized radii. We will be using the same set of boundary radii for the small hex packings that we used in the hyperbolic case. Using the CirclePack[©] program developed by Ken Stephenson, the author converted the packings from hyperbolic geometry to Euclidean geometry.

We will first look at the Euclidean binary algorithm with the small hex packings, lh. The first graph we will look at will have the **factor** as the independent variable, time as the dependent variable and will graph the **iterations** for 1 through 41 **iterations** by a step of 5. This graph can be found in Figure 27. The reader should notice that the behavior of the **iterations** is similar to the behavior seen with the hyperbolic algorithms, namely that the speed of the packing algorithm drops from 1 to 6 **iterations**, but has only minor influence. The graphs of the **iterations** for 6 through 41 **iterations** are almost identical. After a small number of **iterations**, the **iterations** seem to have no significant impact on the speed of the packings. The reader should also notice that the best value for **factor** is 0.4 for the lh packing.

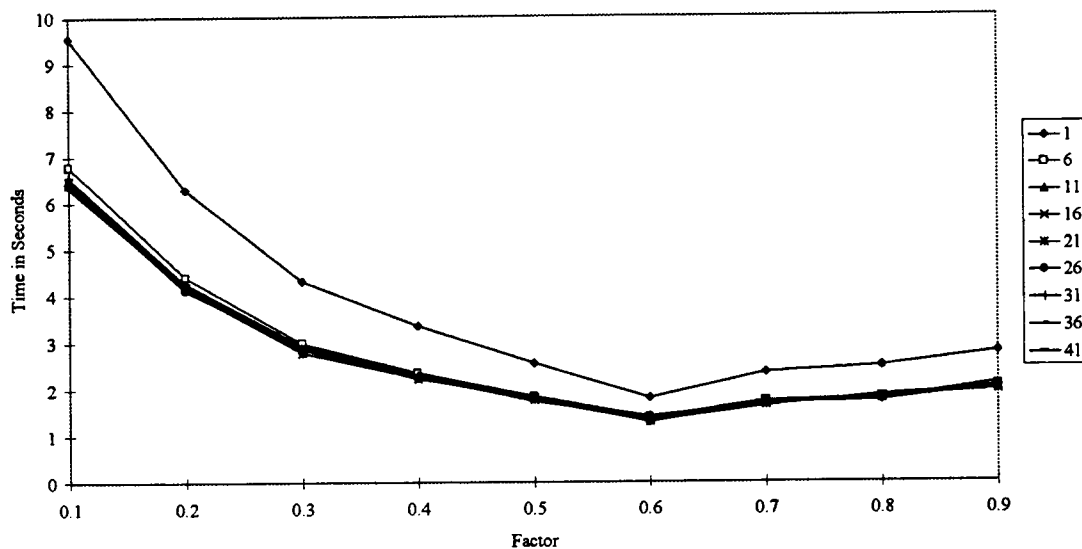


Figure 27 Small Hex, Lh, Packed with Euclidean Binary Algorithm (factor)

To continue the analysis, we want to look at another small hex packing. A graph of lh2 with the **factor** as the independent variable can be found in Figure 28. We notice the same behavior of the number of **iterations** having minimal impact for higher values of **iterations** that we noticed with the lh packing. The lh2 packing has a best value for **factor** of 0.6. Graphs for the small hex packings lh3 and lh4 can be found in Appendix A, Figure 92 and Figure 93. By looking at these graphs, we can notice that the best value for the small hex packings lies between 0.4 and 0.6 for all of the small hex packings. In the hyperbolic case, the best value for the **factor** was .8 and .9 for this same set of packings. The reader should also notice that while the best value for these packings lies between 0.4 and 0.6, changes in the packing time for values of **factor** greater than the best value are not as dramatic as the changes for values of **factor** less than the best value.

To continue our analysis of the Euclidean Newton's method algorithm, we will look at one of the randomized packings, cross. The randomized packings utilized in the analysis of the Euclidean algorithms are the same packings utilized with the hyperbolic algorithms. These packings were converted from hyperbolic to Euclidean using CirclePack^o. A graph of the randomized cross packing with the factor as the independent variable can be found in Figure 29. Notice that this graph portrays the same decrease in packing time from 1 iteration to 6 that we saw with the small hex packings and from 6 iterations and beyond there are only minor differences in the packing time due to the number of iterations. Also note that in this packing, the packing time decreases

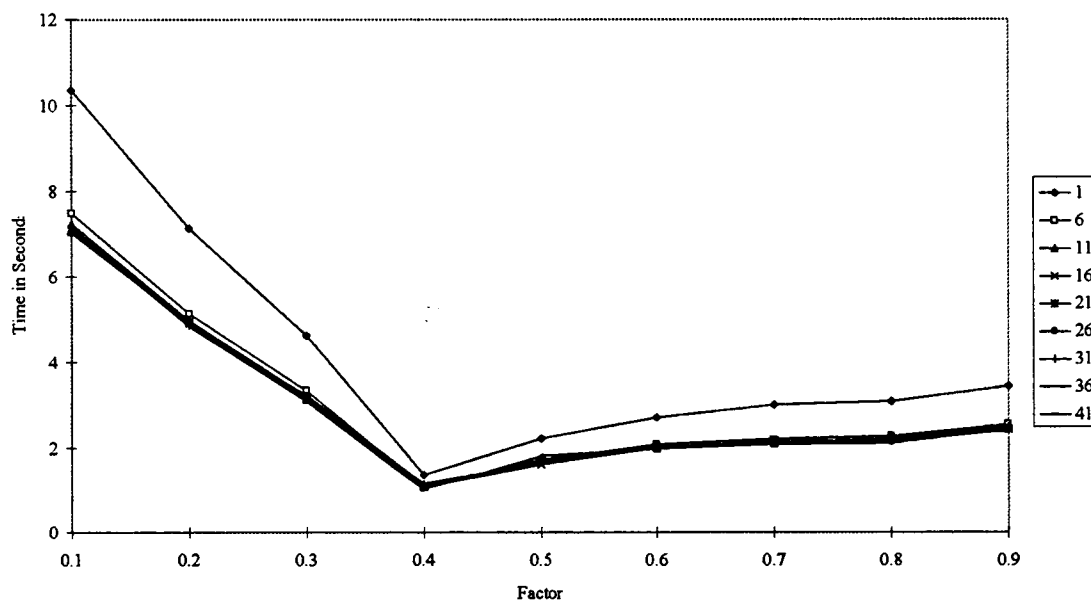


Figure 28 Small Hex, Lh2, Packed with Euclidean Binary Algorithm (factor)

rather quickly from a **factor** of .1 to .3 and then begins to decrease slowly to a low at .9. Graphs of the el, hemisphere and oval packings can be found in Appendix A, Figure 95 through Figure 97. The reader can confirm that each of these packings behaves similarly to the cross packing. One difference noticed between the small hex packings and the randomized radii packings is the difference in the best value for **factor**. For the small hex packings, the best value for **factor** fell between 0.4 and 0.6. However, for the randomized radii packings, the best value for **factor** falls at 0.9.

To conclude this section, the author makes the following observations for the Euclidean binary algorithm:

1. The best value for **factor** is above 0.3 and varies depending on the packing.

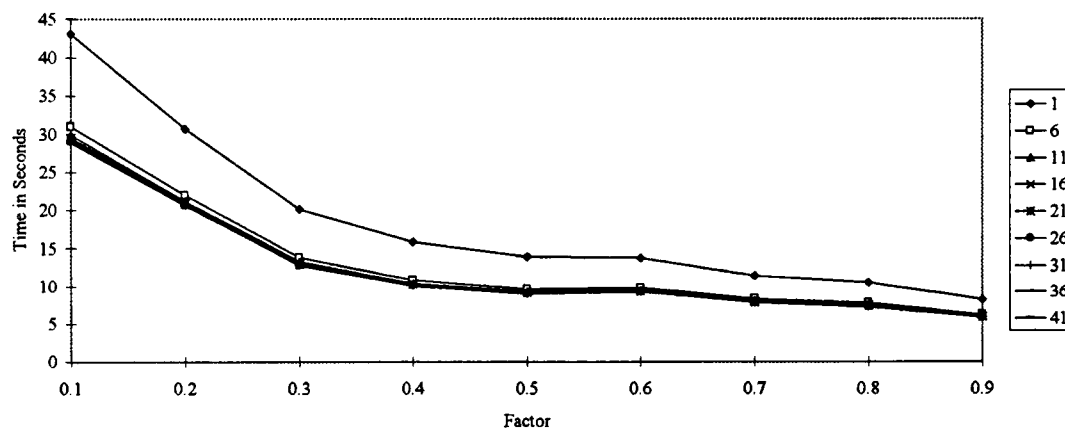


Figure 29 Cross with Randomized Radii Packed with Euclidean Binary Algorithm (factor)

2. The packing time is not dependent on the number of **iterations** after a small number of **iterations**.

9. Euclidean Monotone Algorithm

We want to continue our analysis by looking at the timing of packings created with the Euclidean monotonous algorithm. We will follow the same general outline as we did with the Euclidean binary algorithm. To start, we will look at one of the four small hex packings, lh. The graph of lh with the **factor** as the independent variable can be seen in Figure 30. The reader should notice that as with the hyperbolic monotone algorithm, the Euclidean monotone algorithm also has a dependence on the number of iterations that we did not observe in the Euclidean binary algorithm. Also notice that for a small number of iterations and small value of **factor**, the packing times for the Euclidean monotone algorithm are close to the values for the Euclidean binary algorithm. We also notice that the graphs in Figure 30 are not as “smooth” as they are with the Euclidean binary algorithm, particularly as the number of iterations gets higher. We can look at this same data with

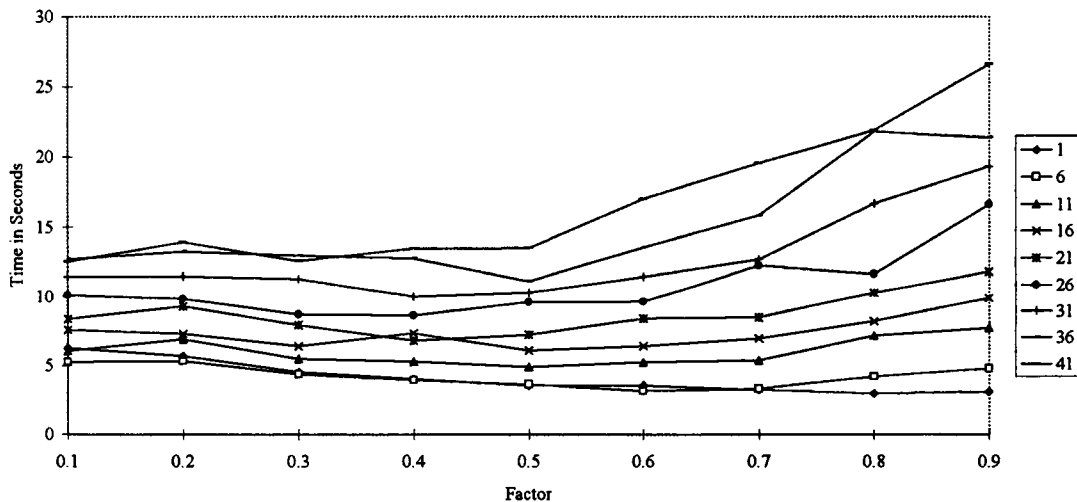


Figure 30 Small Hex Lh Packed with Euclidean Monotone Algorithm (factor)

the number of iterations as the independent variable. This graph can be found in Figure 31. With this graph, we observe for a given factor the packing time increases as the number of iterations increases. Now let's compare the time for packing with this algorithm to the time for the Euclidean binary algorithm. For one iteration, the time to pack with each of these algorithms is similar for small values of factor. However, as the Euclidean binary algorithm becomes more efficient as we increase the factor, the Euclidean monotone algorithm has only minor changes after a factor of .4. As we increase the number of iterations, the Euclidean monotone binary algorithm quickly becomes less efficient while the number of iterations has little effect with the Euclidean binary algorithm. Graphs of the remaining small hex packings are included in Appendix A, Figure 98 through Figure 102 for the reader to confirm these observations.

We will now look at the random packings. We will first look at the graph of the cross packing

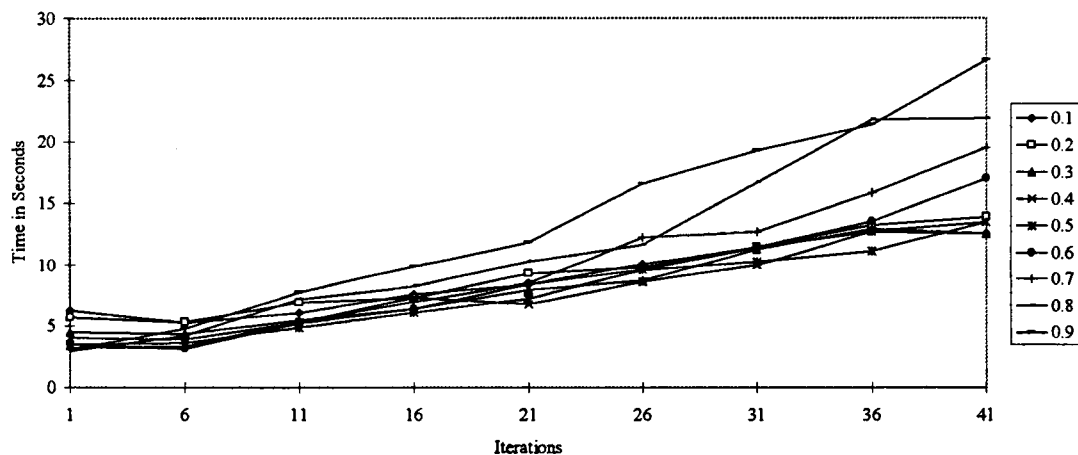


Figure 31 Small Hex, Lh, Packed with Euclidean Monotone Algorithm (iterations)

with randomized radii. Figure 32 shows the graph of the number of iterations with the factor as the independent variable and the time as the dependent variable. The reader should notice that this graph is dependent on the number of iterations. The Euclidean monotone binary algorithm shows a correspondence between the number of iterations and the length of the packing time. When comparing the packing times of this graph to that of the Euclidean binary algorithm, the reader should notice that the packing times of the cross packing with Euclidean monotone algorithm closely follows the packing times of the Euclidean binary algorithm for small values of factor. The reader can confirm these observations with graphs of the el, hemisphere, and oval can be found in Appendix A, Figure 103 through Figure 105.

To further analyze the comparison of the Euclidean binary algorithm with the Euclidean monotone algorithm, let's look at the graph of the best packing time for the number of iterations. For the

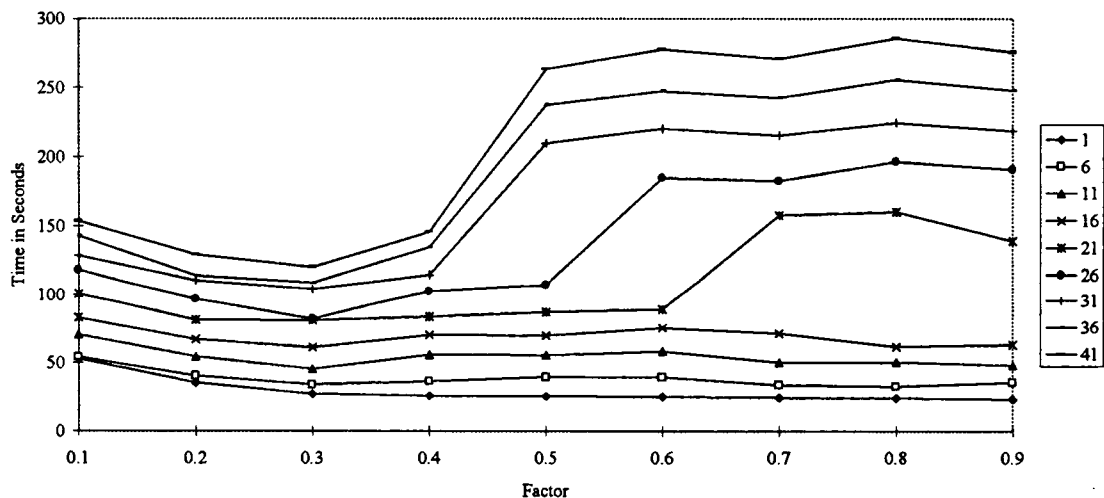


Figure 32 Cross with Randomized Radii Packed with Euclidean Monotone Algorithm (factor)

Euclidean monotone algorithm, this is the graph of one iteration and for the Euclidean binary algorithm, the author will use the graph of 11 iterations. Figure 33 combines these data points into a graph for easy comparison. The reader should notice that the packing times for the monotone graph are close to double or more the packing times of the binary graph. This is to be expected as one pass with the monotone algorithm requires twice as many computations as one pass with the binary algorithm.

To conclude this section, the author makes the following observations for the Euclidean monotone algorithm:

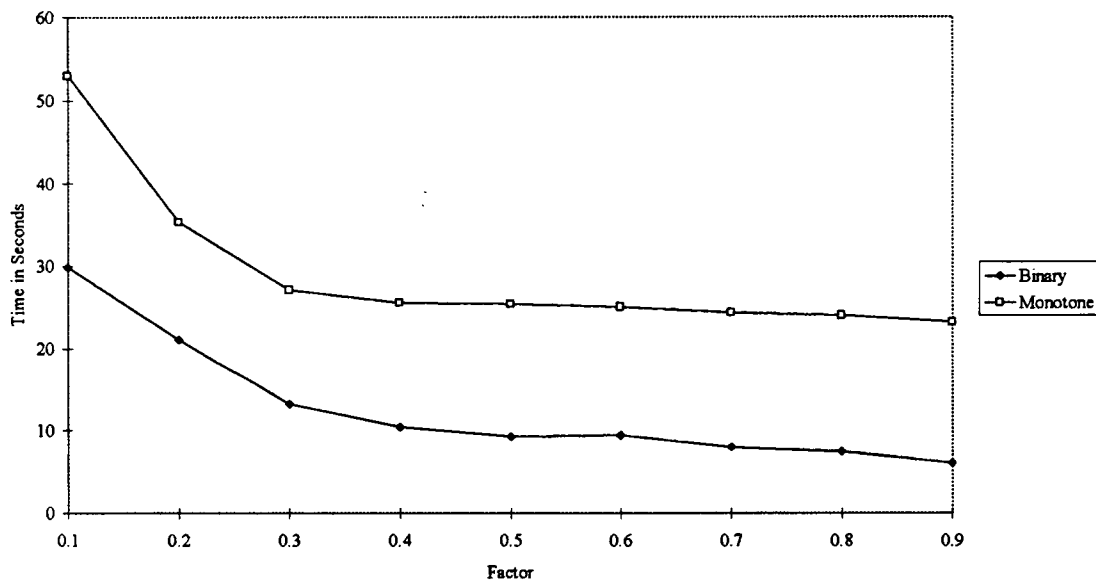


Figure 33 Comparison of Euclidean Binary and Monotone Algorithms using Cross with Randomized Radii (factor)

1. The packing time is dependent on the number of **iterations**.
2. For a given value of **factor**, as the number of **iterations** increases the packing time also increases.
3. For a given value of **factor**, the best packing time for the Euclidean monotone algorithm is at least almost double the packing time for the Euclidean binary algorithm.

10. Euclidean Newton's Method Algorithm

The final algorithm the author studied is the Euclidean Newton's method algorithm. The Euclidean Newton's method algorithm utilizes Newton's method for determining the next estimate for each radii. An upper and lower bound are placed on the radii to avoid adjusting the radii too drastically. To analyze this algorithm, the author first looked at the small hex packings. A graph of lh2 with the factor as the independent variable is included as Figure 34. The reader should quickly notice that packing types for a factor of 0.1 are drastically higher than the packing times for other values of factor and are plotted off the scale of the graph. The author scaled the y-axis to better view the data points for factor >0.1. The data point for factor=0.1 is 97.33 for one

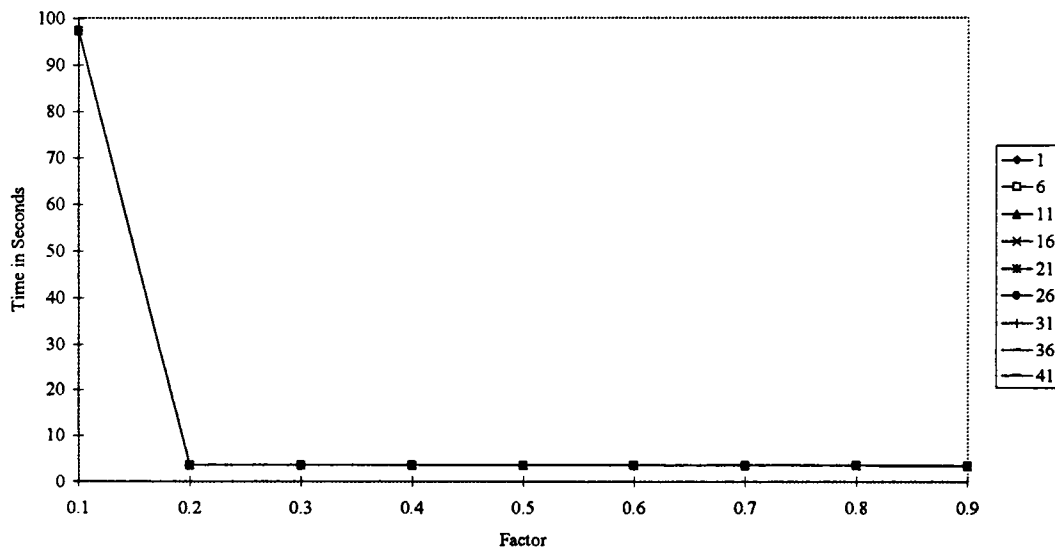


Figure 34 Small Hex, Lh2, Packed with Hyperbolic Newton's Method Algorithm (factor)

iteration and has similar values for other values of iteration. This data point is an order of magnitude greater than the data points for $\text{factor} \geq 0.2$. The other small hex graphs show the same behavior and can be seen in Appendix A, Figure 107 through Figure 109.

To continue this analysis, the author will next look at the random radii packings. Figure 35 is a graph using data from the packing of the cross with random radii. This graph has **factor** as the independent variable and graphs the data for each of the different iterations. The reader should notice that values for $\text{factor}=0.1$ are of the same order of magnitude as the rest of the values for **factor**. Also, the graphs have no dependence on the number of iterations. Figure 35 displays a distinct best value for **factor** at 0.6. Graphs of el, hemisphere, and oval packings with randomized

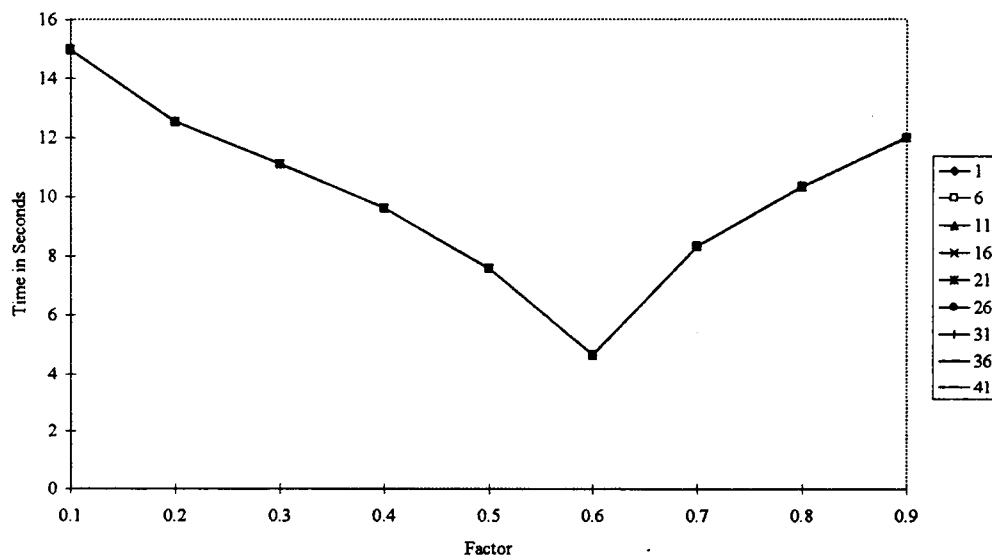


Figure 35 Cross with Randomized Radii Packed with Euclidean Newton's Method Algorithm (factor)

radii can be found in Appendix A, **Figure 110** through **Figure 112**. These additional graphs demonstrate a best value for **factor** to be between 0.4 and 0.6.

To conclude this section, the author makes the following observations for the Euclidean Newton's method algorithm:

1. The best value for **factor** lies between 0.4 and 0.6.
2. The number of **iterations** has little effect on the packing time.

11. Comparison of Euclidean Circle-Packing

Algorithms

The author concludes his discussion of the Euclidean circle-packing algorithms with a comparison of the Euclidean binary, monotone, and Newton's method algorithms. To compare these algorithms, the author took the iteration that contained the best packing time for each packing and algorithm combination. For each packing, a graph was created of the packing times for each algorithm with the factor as the independent variable. Each of these graphs was plotted on the same chart for comparison. Figure 36 contains the graph of the small hex packing lh. The author reduced the scale for the y-axis to allow enough detail in the graph to make observations. This caused the data point at factor=0.1 for the Newton's method algorithm to be plotted off the chart.

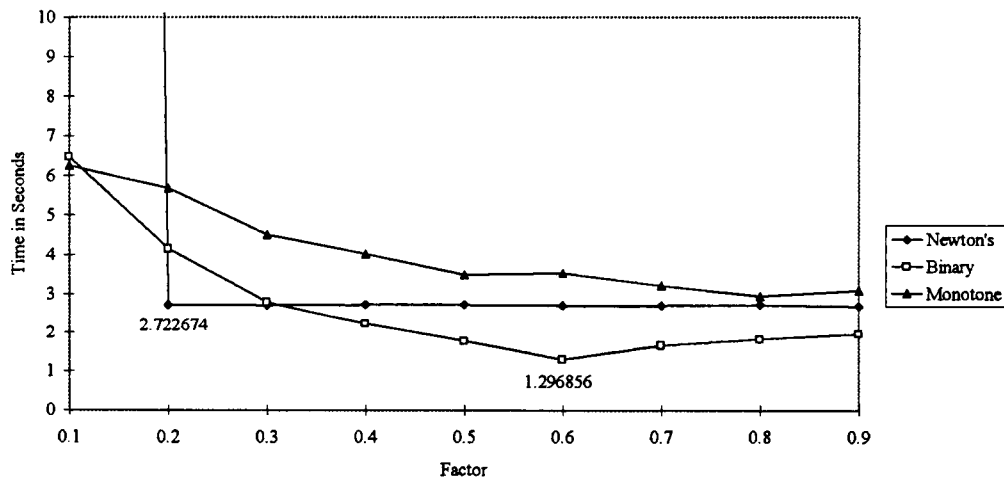


Figure 36 Comparison of Euclidean Circle-Packing Algorithms with Small Hex, Lh

The reader should notice that with the small hex, lh packing, the binary algorithm gives the best packing time. The data points of the best times for the binary algorithm and the Newton's method algorithm are labeled for easy comparison. The graph of the Euclidean monotone algorithm was included for completeness. The graphs of the remaining small hex packings are included in Appendix A, Figure 113 through Figure 115. The reader should confirm that the binary algorithm is best for all of the small hex packings as long as $\text{factor} > 0.4$.

Figure 37 contains graphs of the data-points for each of the three Euclidean circle-packing algorithms for the cross with randomized radii. This packing behaves the opposite of the small hex packings. For the cross packing, the Newton's method offers the best performance. The binary algorithm was best with the small hex packings for a value of $\text{factor} > 0.4$. Newton's method is

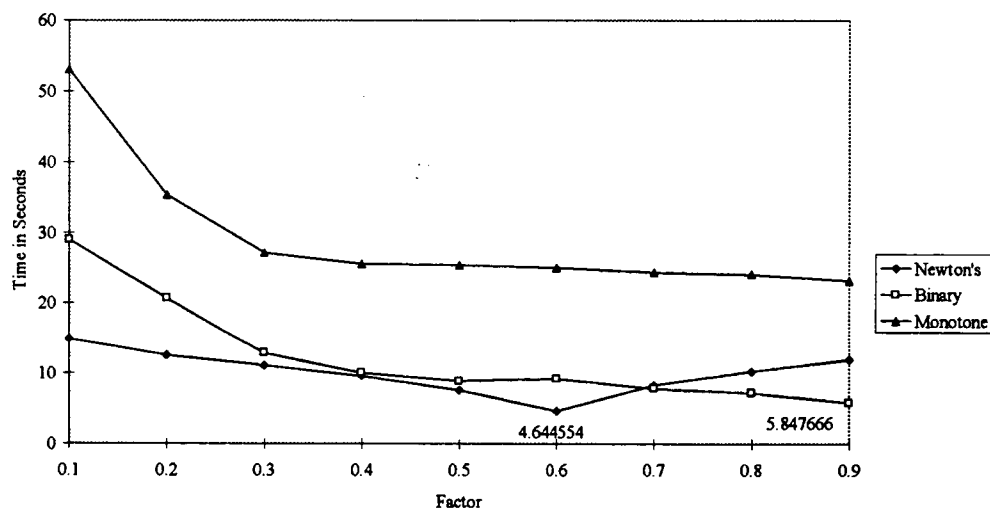


Figure 37 Comparison of Euclidean Circle-Packing Algorithms with the Cross Packing with Randomized Radii

best with the randomized cross packing for a value of **factor** < 0.6 . Graphs of the el, hemisphere, and oval are included in Appendix A for the reader to verify this holds for each of the randomized radii packings. The reader should also note that the graph of the monotone algorithm closely matches the graph of the binary algorithm. While the graphs of the monotone algorithm somewhat followed the graph of the binary algorithm for the small hex packing, it is more obvious with the random radii packings. The reader can confirm these observations with the other random radii packings which can be found in Appendix A, **Figure 113** through **Figure 118**.

12. Connecting CirclePack[©] with the MasPar

The front-end of the MasPar is a DEC Alpha running DEC ULTRIX version 4.2. CirclePack[©] has not been compiled to run on the DEC ULTRIX, so we needed to be able to pass information from CirclePack[©] running on a Sun SparcStation to the MasPar running on DEC ULTRIX. To accomplish this, the author utilized Parallel Virtual Machine, PVM. PVM was created as a joint venture between the University of Tennessee, Oak Ridge National Laboratory and the Massachusetts Institute of Technology. CirclePack[©] is started on a SUN processor as normal. CirclePack[©] has built-in functionality to call an external program. We use this functionality to start an external subroutine from CirclePack[©] which accepts the cp-complex data from CirclePack[©] and generates a 64x64 array that simulates the MasPar processor array. The routine then creates a connection to the MasPar front-end and sends the array to the MasPar front-end. The MasPar front-end acts only as a conduit for the information. As soon as the information is received, it is loaded onto the MasPar processor array. Once the calculations are completed on the processor array, the information is sent back to the MasPar front-end where it is passed back to the Sun and resubmitted to CirclePack[©]. While this utilizes only a small amount of the capabilities of PVM, it was an effective solution. This same PVM solution can be used to connect CirclePack[©] to any number of different machines or it can be used to split the packing into smaller chunks and have each chunk sent to a different machine. For example, each of the Sun machines in the Hydra lab are set up to run PVM. They could be utilized to work on a massive cp-complex.

13. Summary and Conclusions

The author has looked at three different algorithms in two different geometries. There are some similarities in how the algorithms acted in the two different geometries. In the binary algorithm, we saw no dependence on the number of iterations after a small number of iterations. While the best value for **factor** varied between the different packings and between the two geometries, lower values for **factor**, ≤ 0.3 , performed poorly. The hyperbolic binary algorithm performed better with higher values for **factor**, but the author believes that to be due to the upper bound placed on the radii. With the monotone algorithms, we noticed a direct correlation between the packing time and the number of iterations. Small values for iterations performed best and the packing time quickly worsened as the number of iterations increased.

The two Newton's method algorithms did not have as much in common as the binary and monotone algorithms. In the hyperbolic case, we saw with Newton's method factors below 0.5 did not reach an approximate packing label. In the Euclidean case, we saw the values for **factor** having an impact on the packing time.

For the two fastest algorithms, the binary and Newton's method algorithms, the best value for the iterations is easily determined. For the **factor** the best value is more elusive. Even though a change in **factor** of 0.1 can have an impact on the packing time of up to 25% or more, the author feels that the best place to look for improvements in the overall packing time between CirclePack^o and the MP-2 is to concentrate on the algorithm used to prepare the data for use on the MasPar.

While this was not included in the timings, the amount of time required to perform this task seems significant.

The algorithms used on the MasPar are good first attempts at parallel circle packing. There are many enhancements that could be made. The author chose not to consider branch points during this analysis, but regular hex packings with branch points should be able to be processed using these algorithms. The current algorithms assume the boundary radii are fixed. Within CirclePack© either the boundary radii are fixed or the boundary angle sum is fixed. This would be a simple modification to the current algorithm. The current limitation of the packing size could be eliminated. The easiest portion of this task would be to create a function that would wrap with the main processor array on the MasPar and pick up the correct data set. The more difficult task would be to get the data in a form to be sent to the MasPar processor array. The algorithms used could easily be modified to work with a regular octagonal packing or the ball bearing packing. The utilization of PVM has possibilities for generating a parallel packing algorithm which would work with a more generalized packing than the hex packing.

Bibliography

Bibliography

1. D. Aharonov, *The Hexagonal Packing Lemma and the Rodin Sullivan Conjecture*, **343** (1), May 1994, 157-167.
2. I. Barany, Z. Furedi, and J. Pach, *Discrete Convex Functions and Proof of the Six Circle Conjecture of Fejes Toth*, *Can. J. Math.*, vol. XXXVI (3) (1984), 569-576.
3. A. F. Beardon and K. Stephenson, *The Schwarz-Pick Lemma for Circle Packings*, *Illinois Journal of Mathematics*, **35** (4) Winter 1991, 577-605.
4. A. F. Beardon and K. Stephenson, *The Uniformization Theorem for Circle Packings*, *Indiana University Mathematics Journal*, **39** (4) (1990), 1383-1425.
5. A. F. Beardon and K. Stephenson, *Circle Packings in Different Geometries*, *Tohoku Math. J.*, **43** (1991), 27-36.
6. R. Brooks, *The Spectral Geometry of the Apollonian Packing*, *Communications on Pure and Applied Mathematics*, Vol. XXXVIII (1985), 357-366.
7. R. Brooks, *On the Deformation Theory of Classical Schottky Groups*, *Duke Mathematical Journal*, **52** (4), December 1985, 1009-1024.
8. I. Carter and B. Rodin, *An Inverse Problem for Circle Packing and Conformal Mapping*, *Transactions of the American Mathematical Society*, **334**, (2) December 1992, 861-875.
9. T. Dubejko and K. Stephenson, *The Branched Schwarz Lemma: A Classical Result via Circle Packing*, *Michigan Math. J.* **42** (1995), 211-233.
10. T. Dubejko and K. Stephenson, *Circle Packing: Experiments in Discrete Analytic Function Theory*, preprint.
11. Z. He, *An Estimate for Hexagonal Circle Packings*, *J. Differential Geometry*, **33** (1991), 395-412.
12. Z. He, *Solving Beltrami Equations by Circle Packing*, *Transactions of the American Mathematical Society*, **322** (2), December 1990, 657-670.
13. D. Minda and B. Rodin, *Circle Packing and Riemann Surfaces*, *Journal D'Analyse Mathematique*, **57** (1991), 221-249.
14. B. Mohar, *A Polynomial Time Circle Packing Algorithm*, *Discrete Mathematics*, **117** (1993), 257-263.

Appendices

Appendix A Additional Figures

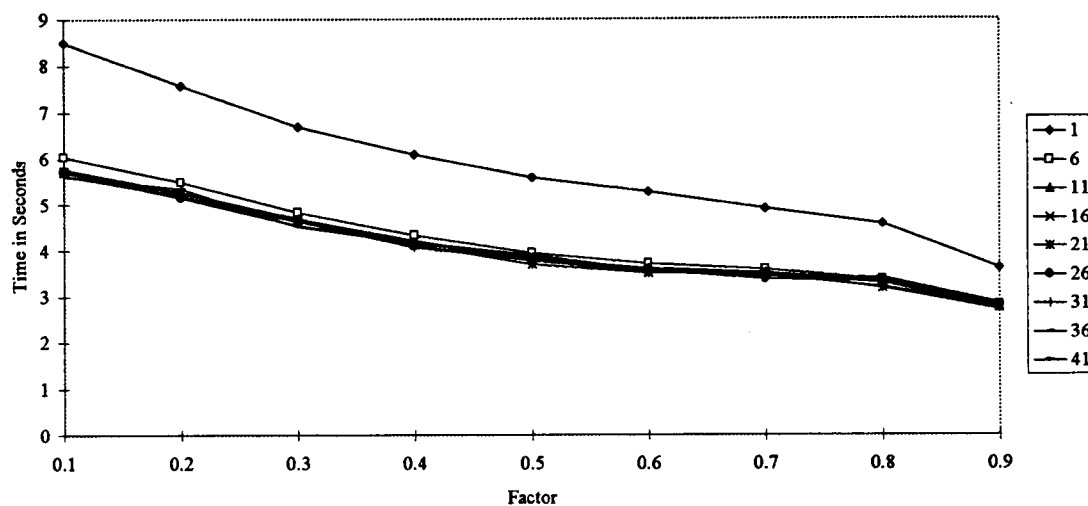


Figure 38 El with Andreev Radii Packed with Hyperbolic Binary Algorithm (factor)

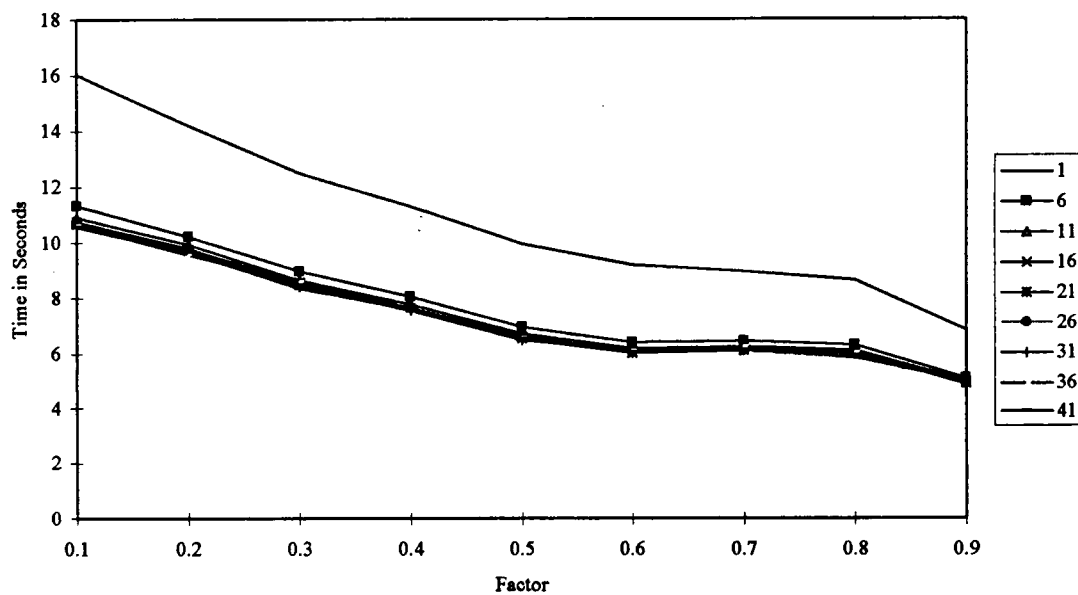


Figure 39 Hemisphere with Andreev Radii Packed with Hyperbolic Binary Algorithm (factor)

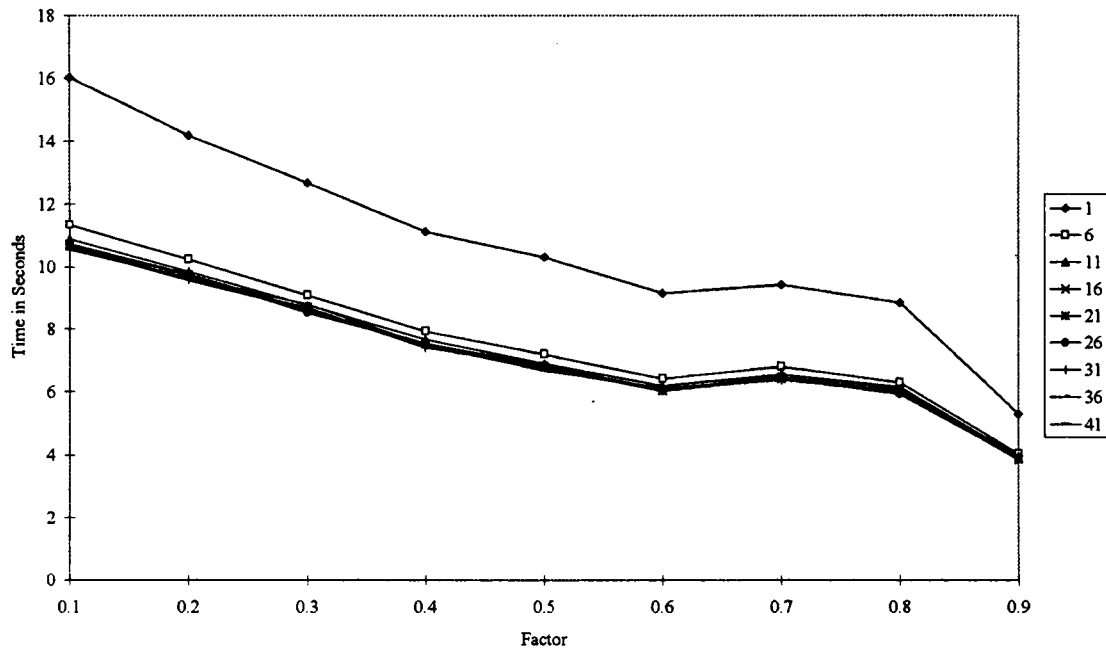


Figure 40 Oval with Andreev Radii Packed with Hyperbolic Binary Algorithm (factor)

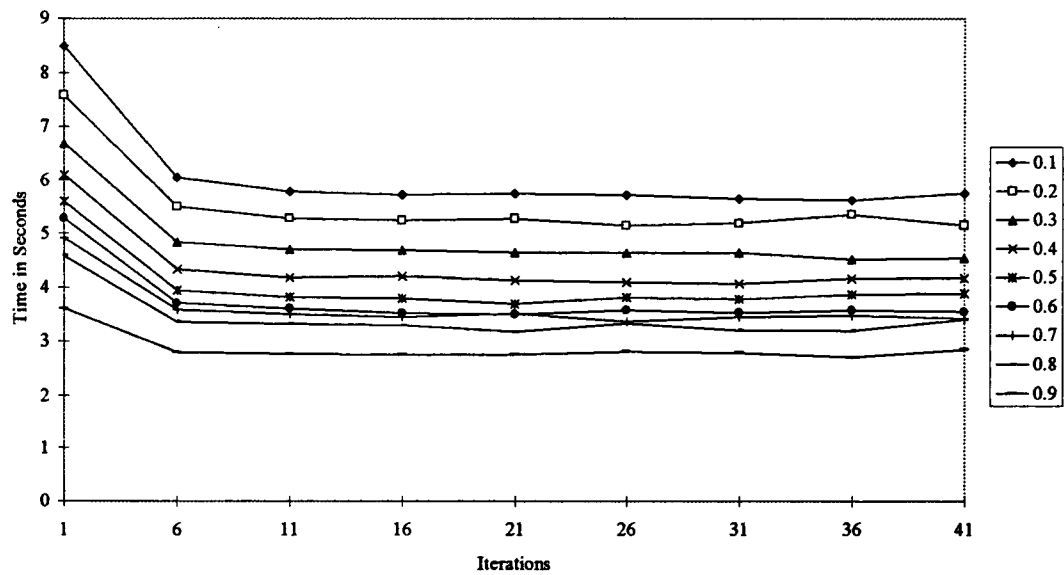


Figure 41 El with Andreev Radii Packed with Hyperbolic Binary Algorithm (iterations)

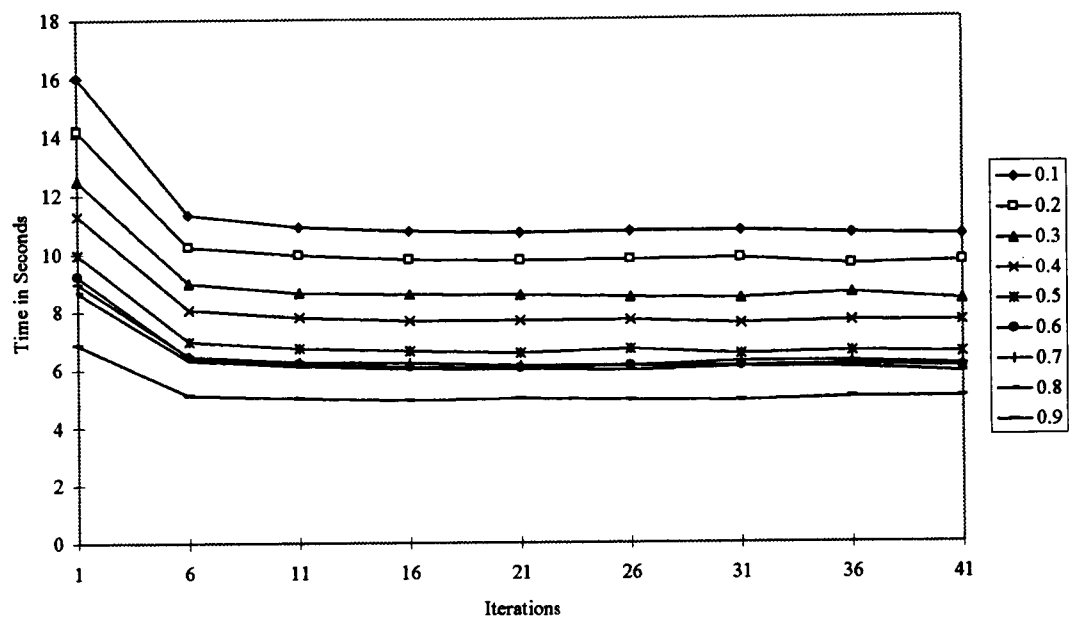


Figure 42 Hemisphere with Andreev Radii Packed with Hyperbolic Binary Algorithm (iterations)

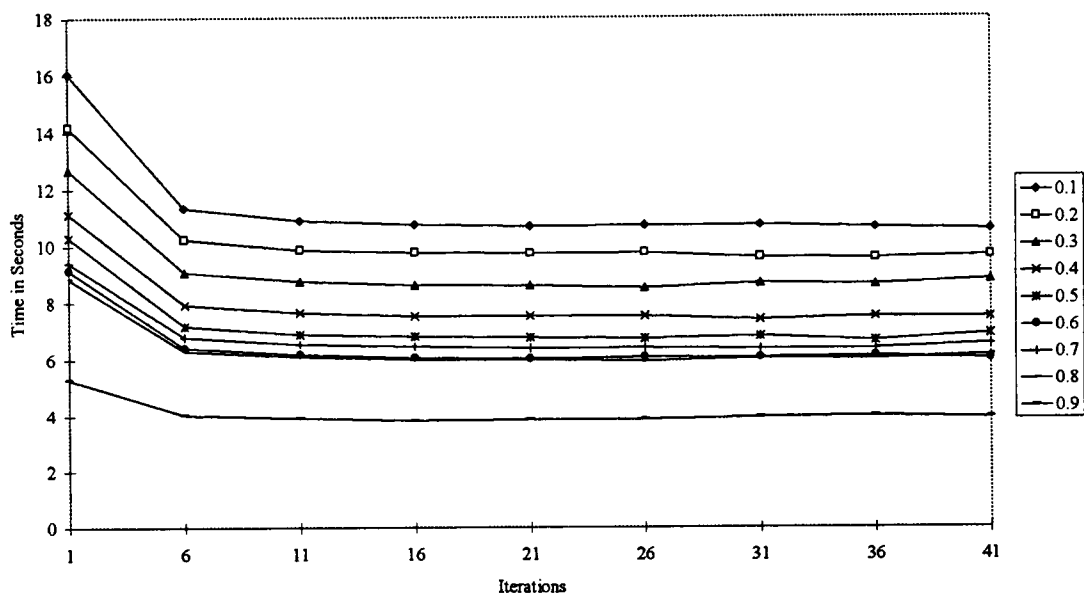


Figure 43 Oval with Andreev Radii Packed with Hyperbolic Binary Algorithm (iterations)

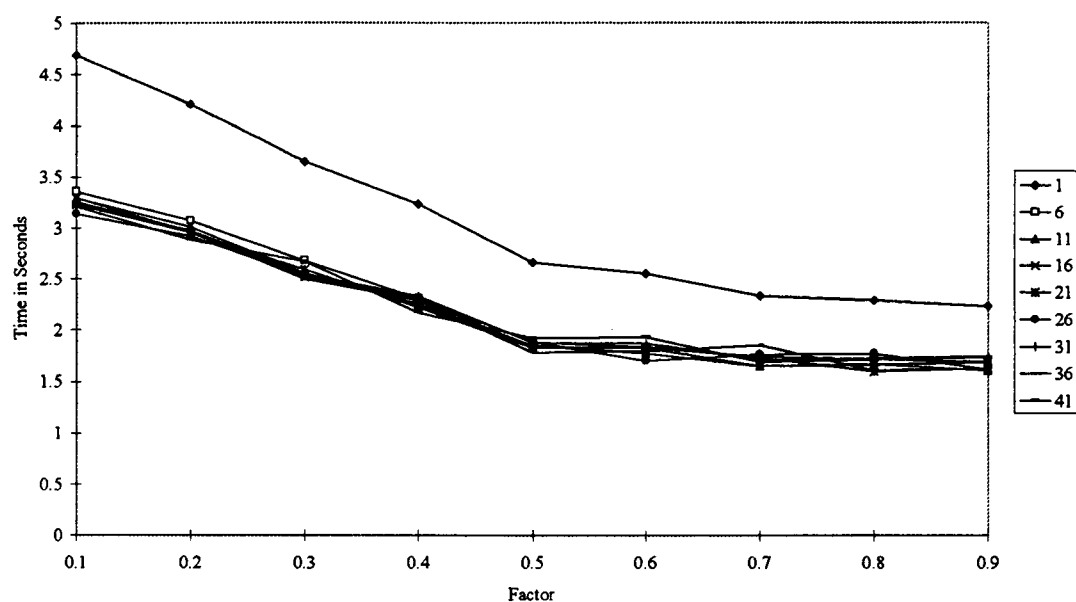


Figure 44 Small Hex with Boundary Condition Lh2 Packed with Hyperbolic Binary Algorithm (factor)

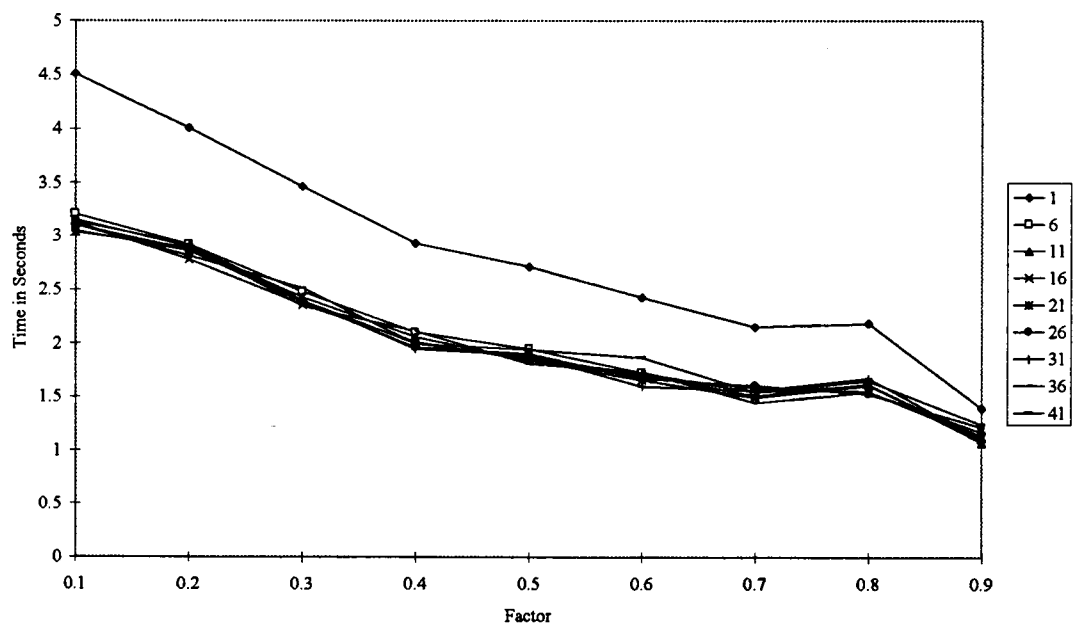


Figure 45 Small Hex with Boundary Condition Lh3 Packed with Hyperbolic Binary Algorithm (factor)

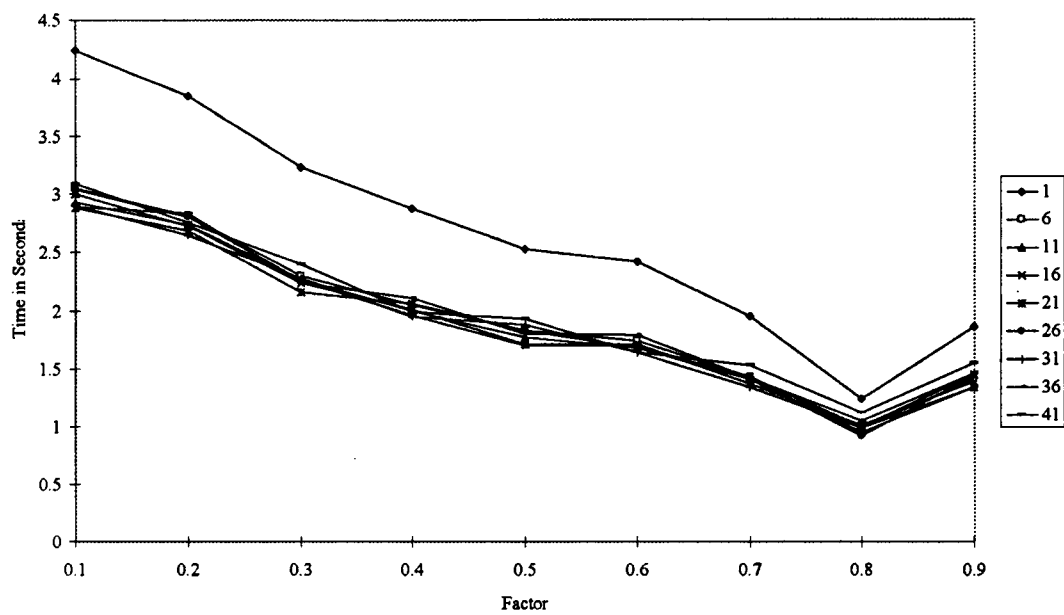


Figure 46 Small Hex with Boundary Condition Lh4 Packed with Hyperbolic Binary Algorithm (factor)

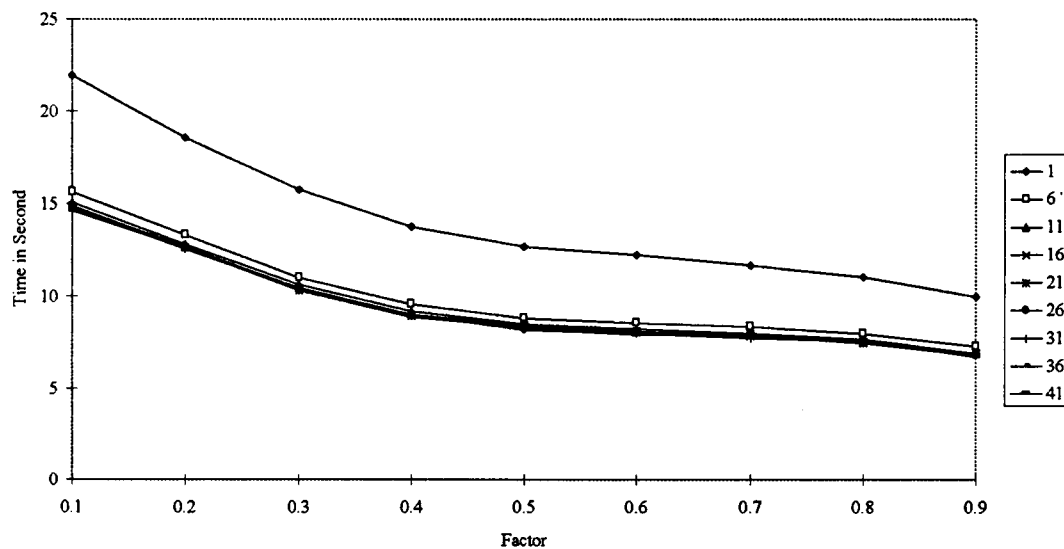


Figure 47 El with Randomized Radii Packed with Hyperbolic Binary Algorithm (factor)

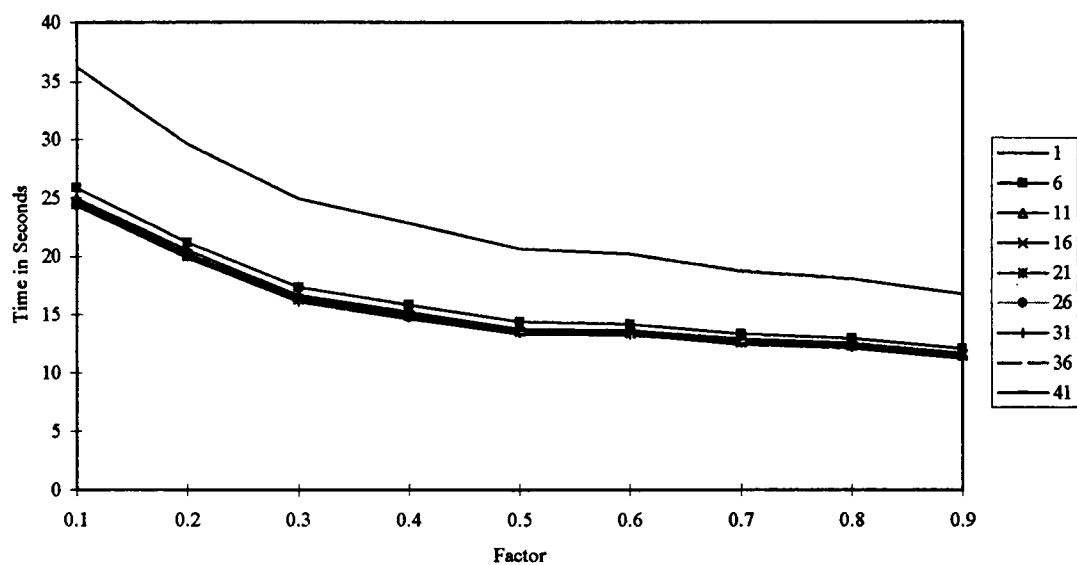


Figure 48 Hemisphere with Randomized Radii Packed with Hyperbolic Binary Algorithm (factor)

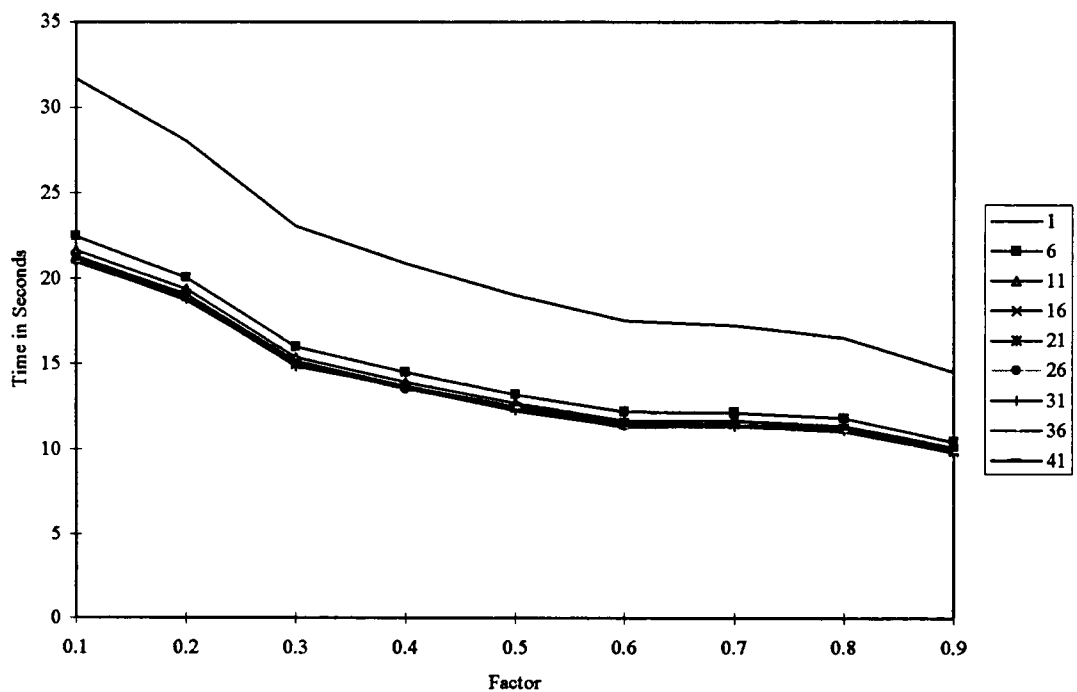


Figure 49 Oval with Randomized Radii Packed with Hyperbolic Binary Algorithm (factor)

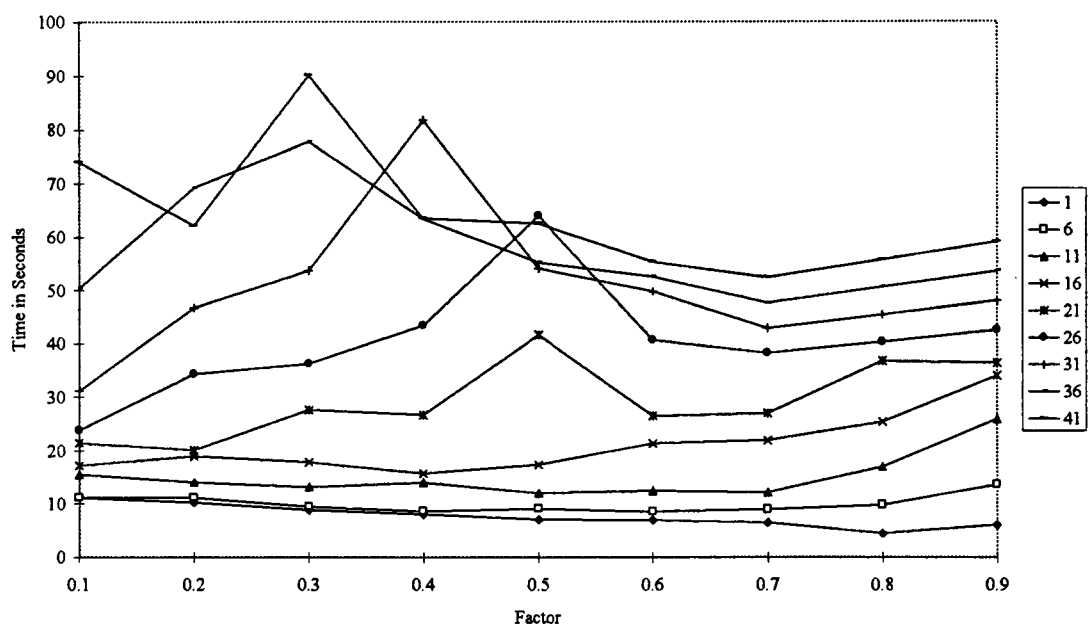


Figure 50 El with Andreev Radii Packed with Hyperbolic Monotone Algorithm (factor)

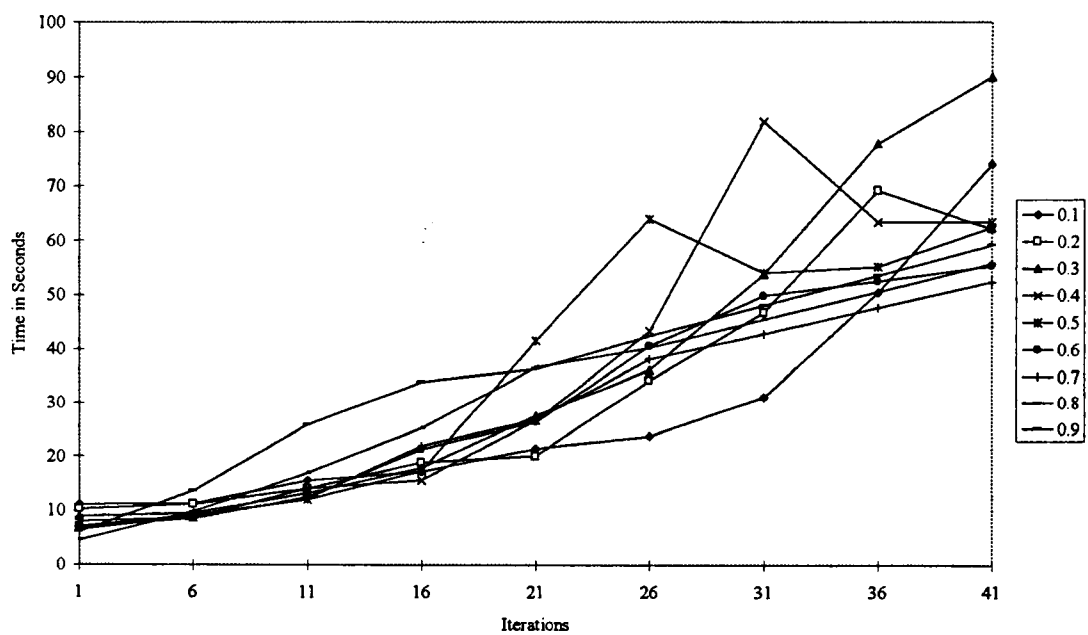


Figure 51 El with Andreev Radii Packed with Hyperbolic Monotone Algorithm (iterations)

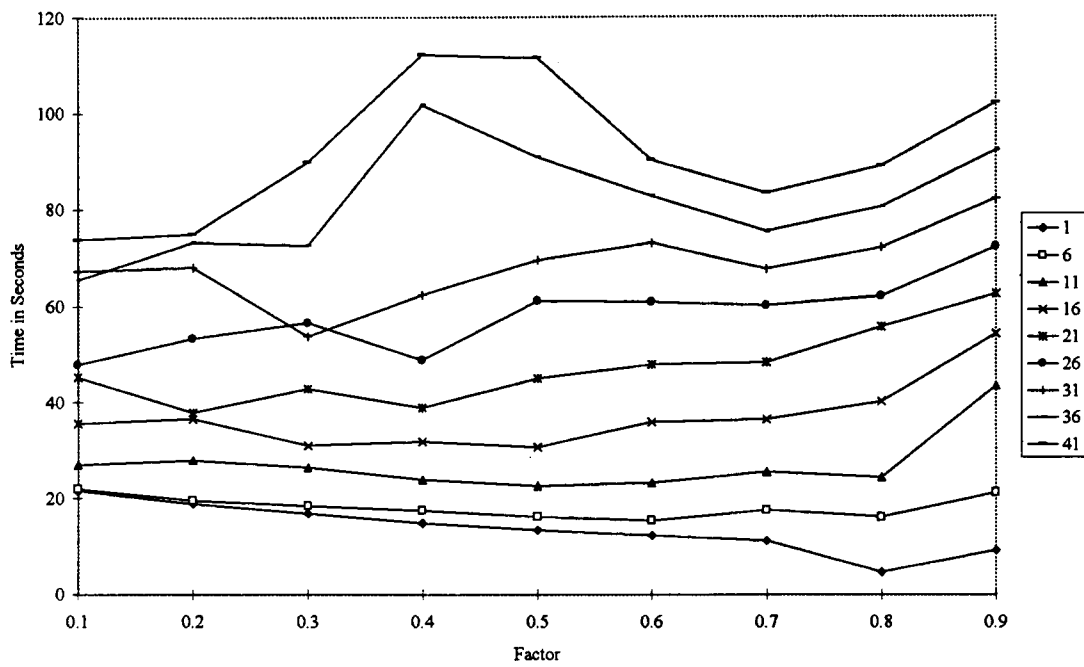


Figure 52 Hemisphere with Andreev Radii Packed with Hyperbolic Monotone Algorithm (factor)

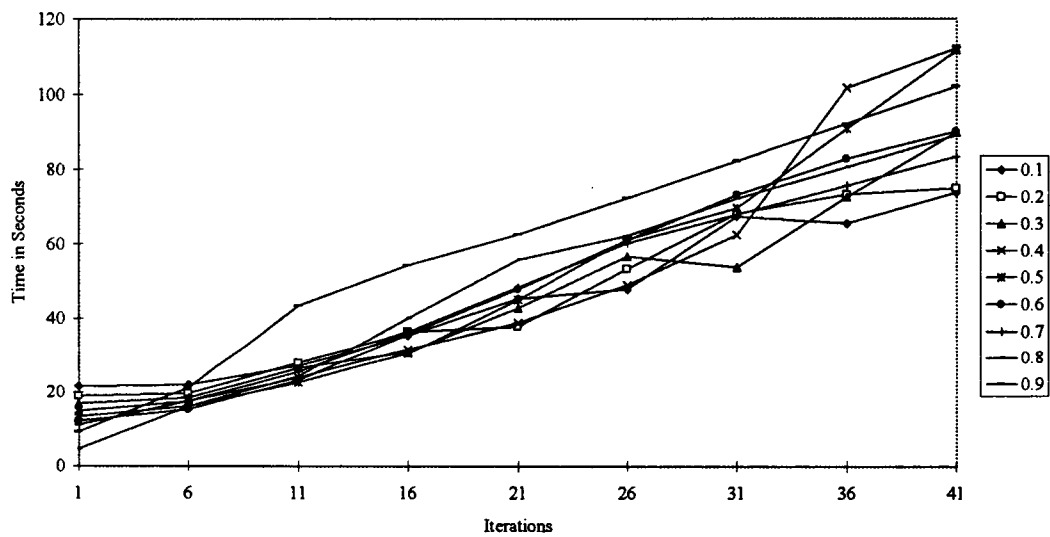


Figure 53 Hemisphere with Andreev Radii Packed with Hyperbolic Monotone Algorithm (iterations)

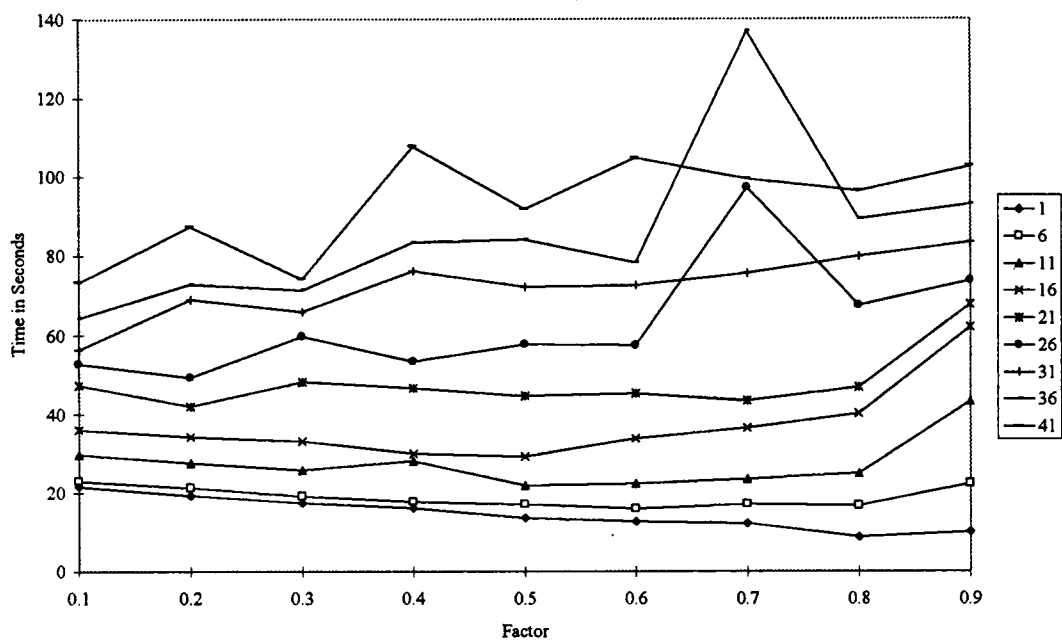


Figure 54 Oval with Andreev Radii Packed with Hyperbolic Monotone Algorithm (factor)

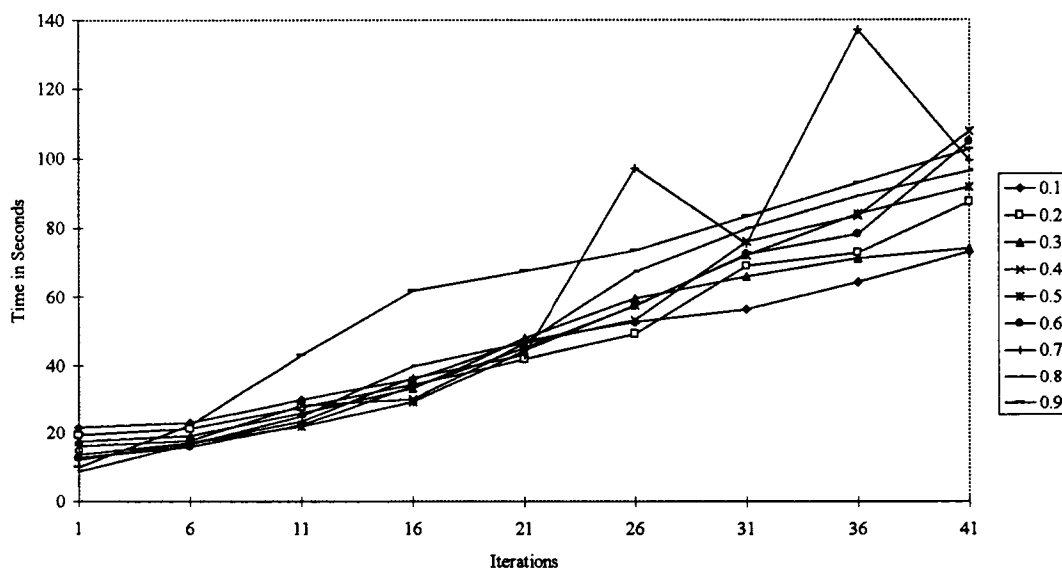


Figure 55 Oval with Andreev Radii Packed with Hyperbolic Monotone Algorithm (iterations)

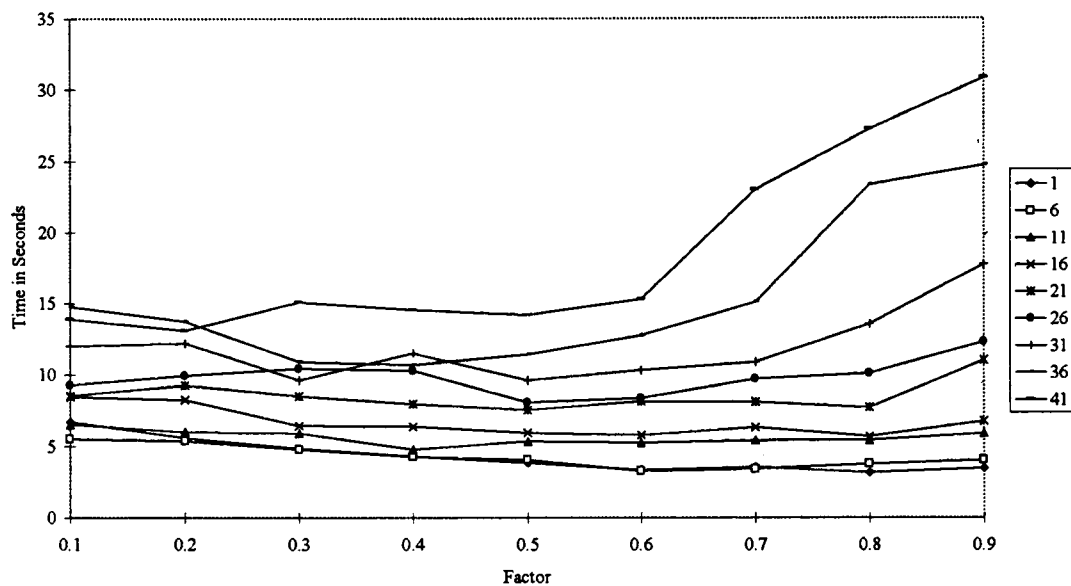


Figure 56 Small Hex, Lh2, Packed with the Hyperbolic Monotone Algorithm (factor)

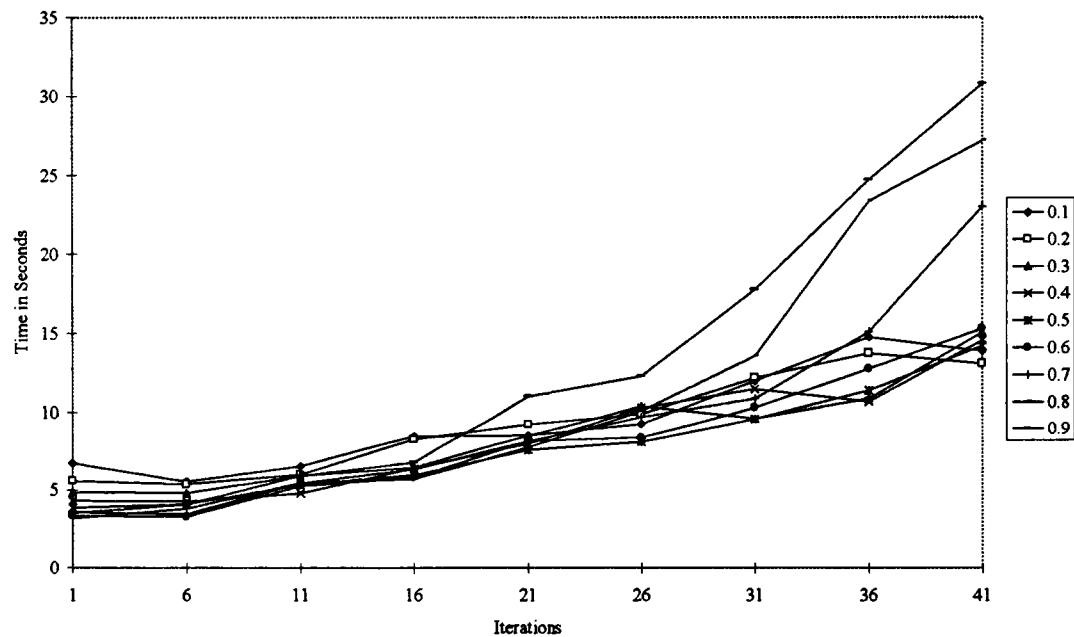


Figure 57 Small Hex, Lh2, Packed with the Hyperbolic Monotone Algorithm (iterations)

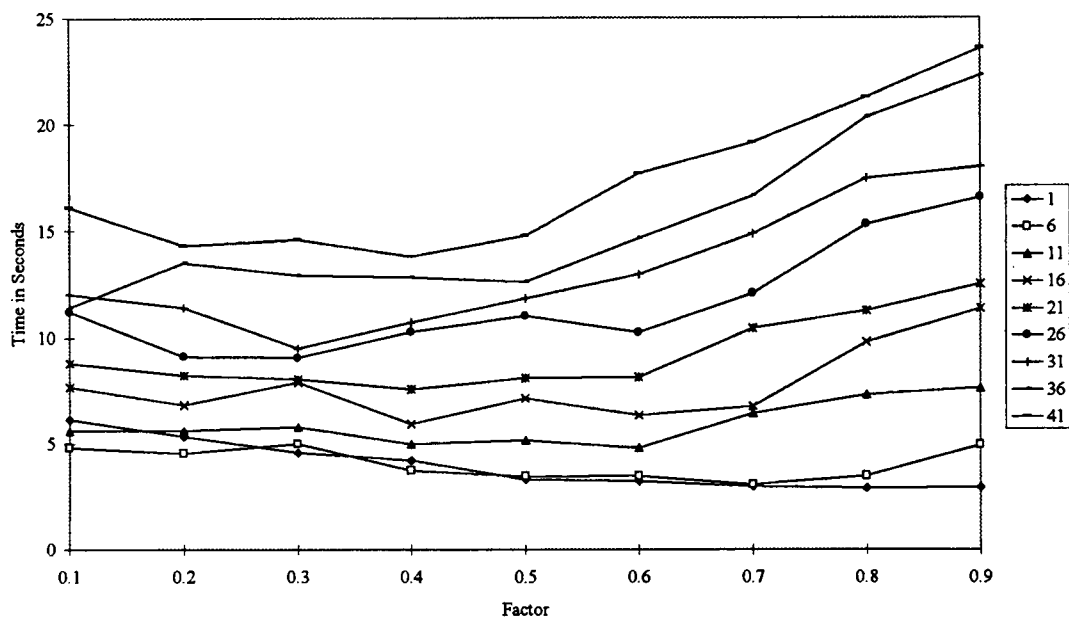


Figure 58 Small Hex, Lh, Packed with the Hyperbolic Monotone Algorithm (factor)

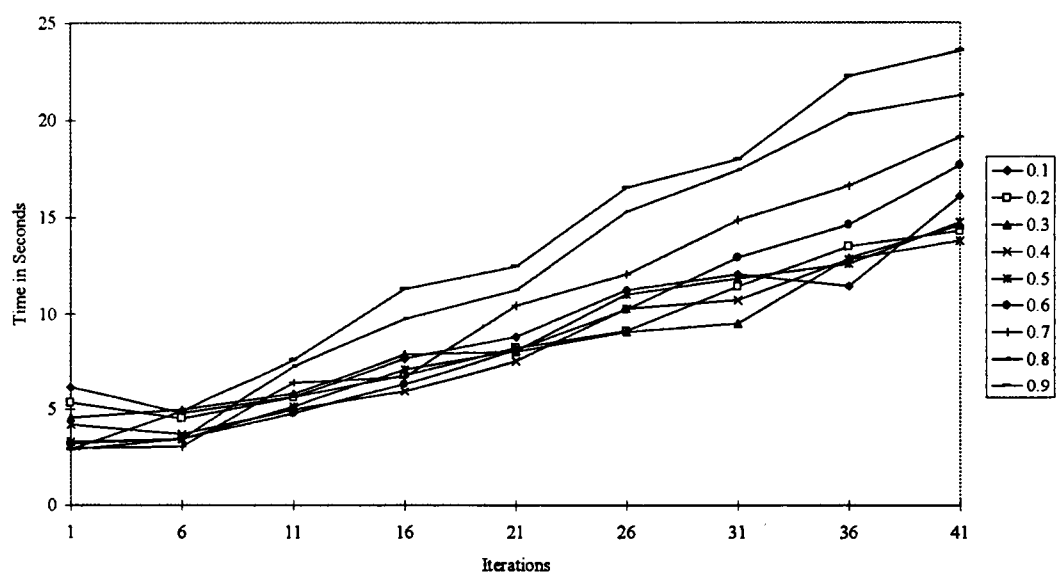


Figure 59 Small Hex, Lh3, Packed with the Hyperbolic Monotone Algorithm (iterations)

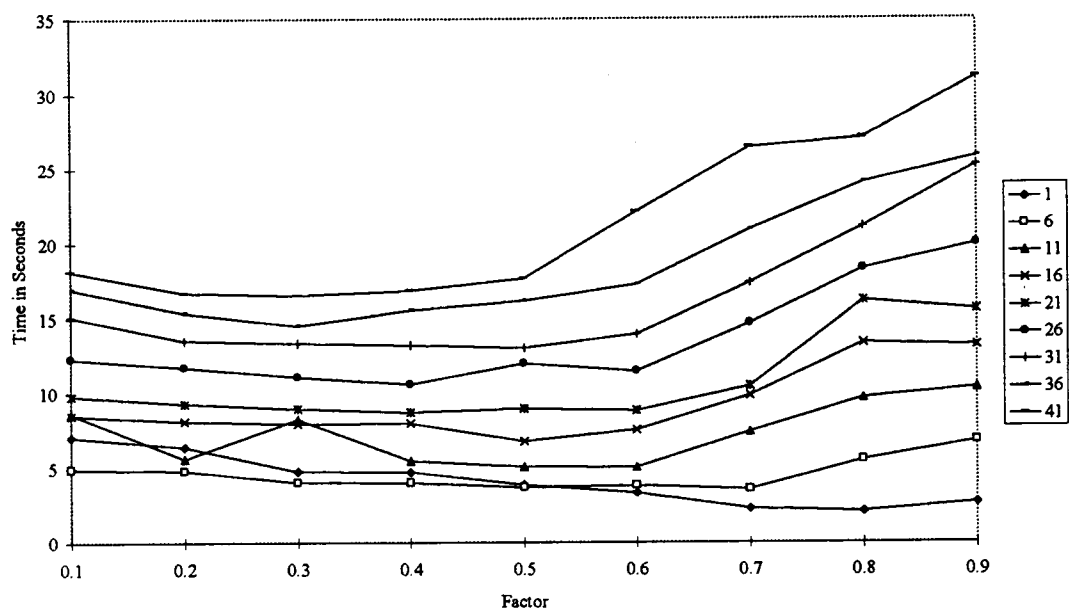


Figure 60 Small Hex, Lh4, Packed with the Hyperbolic Monotone Algorithm (factor)

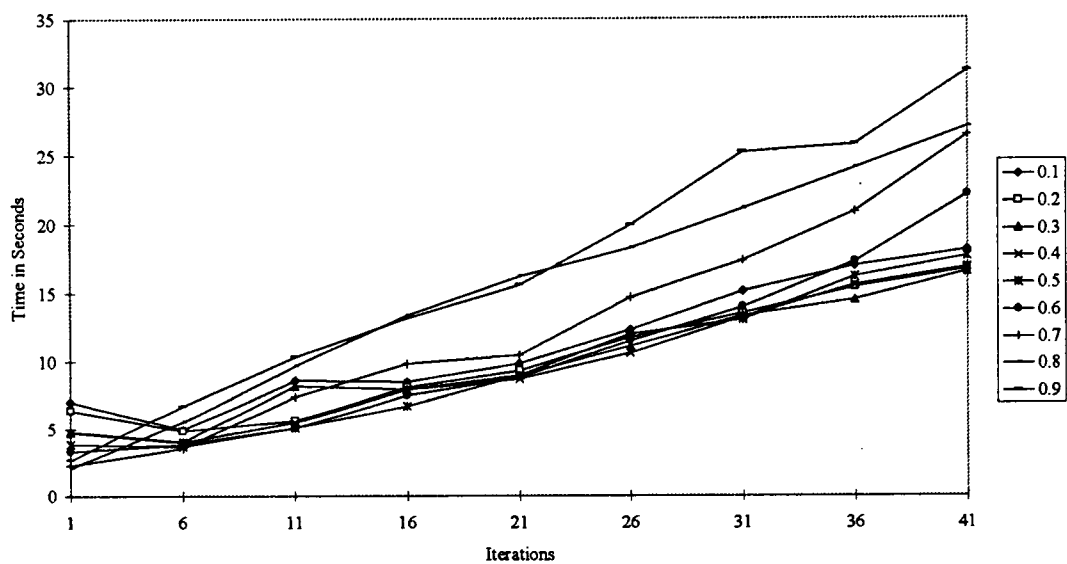


Figure 61 Small Hex, Lh4, Packed with the Hyperbolic Monotone Algorithm (iterations)

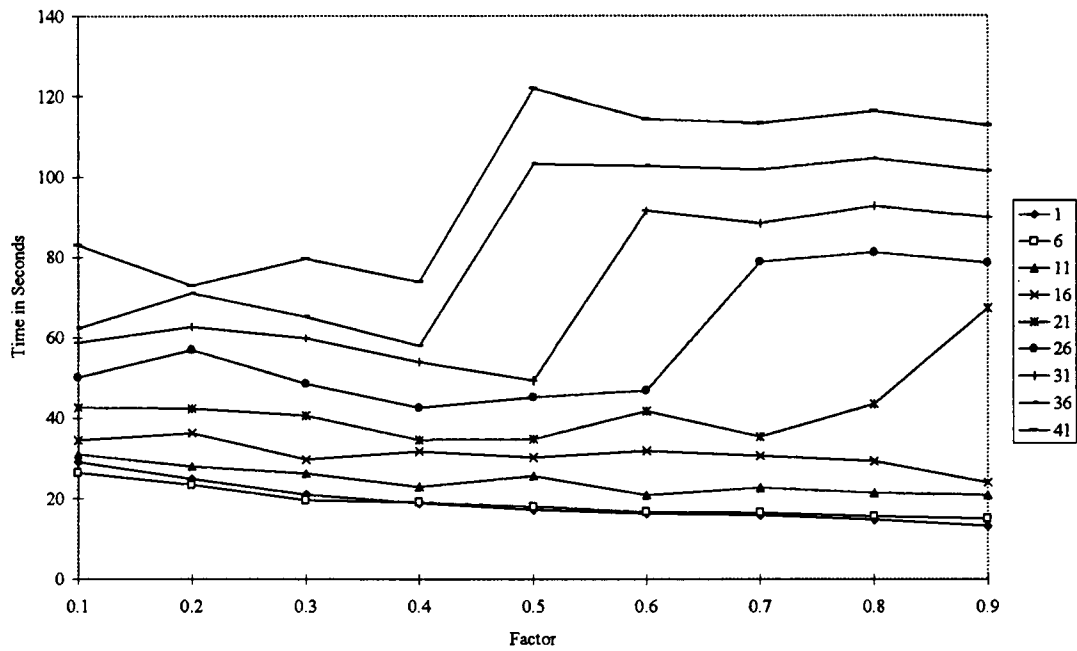


Figure 62 El with Randomized Radii Packed with Hyperbolic Monotone Algorithm (factor)

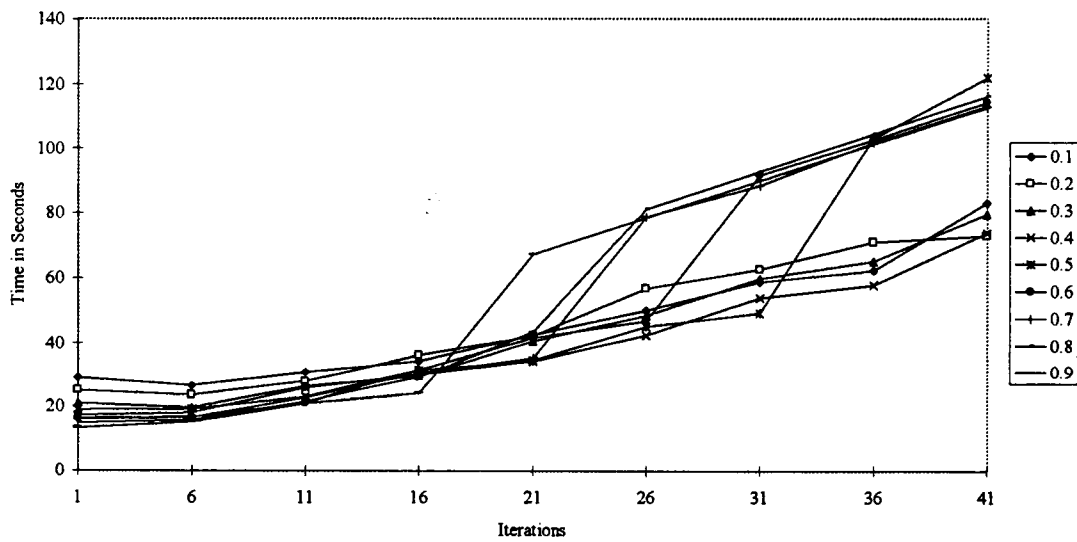


Figure 63 El with Randomized Radii Packed with Hyperbolic Monotone Algorithm (iterations)

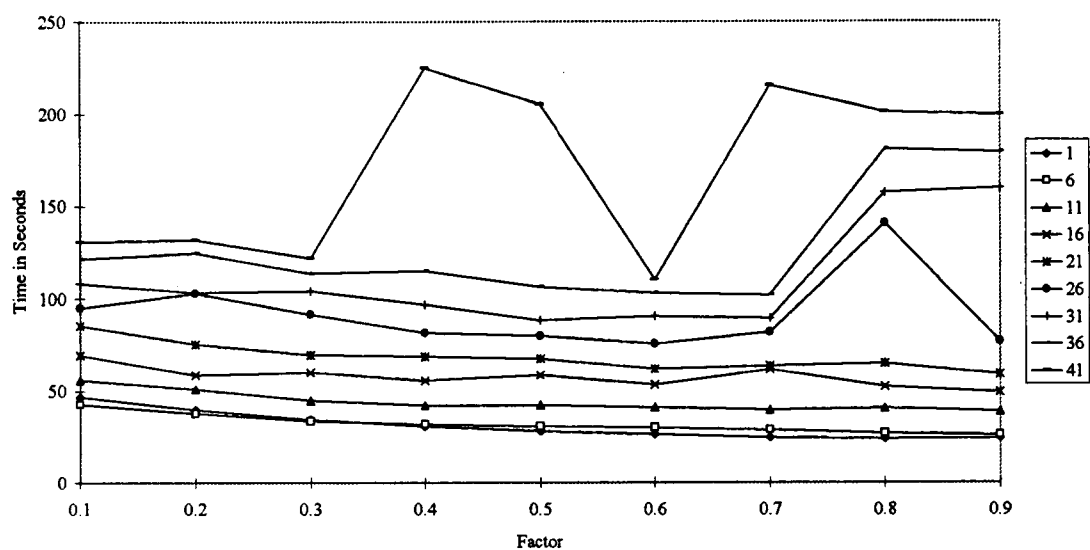


Figure 64 Hemisphere with Randomized Radii Packed with Hyperbolic Monotone Algorithm (factor)

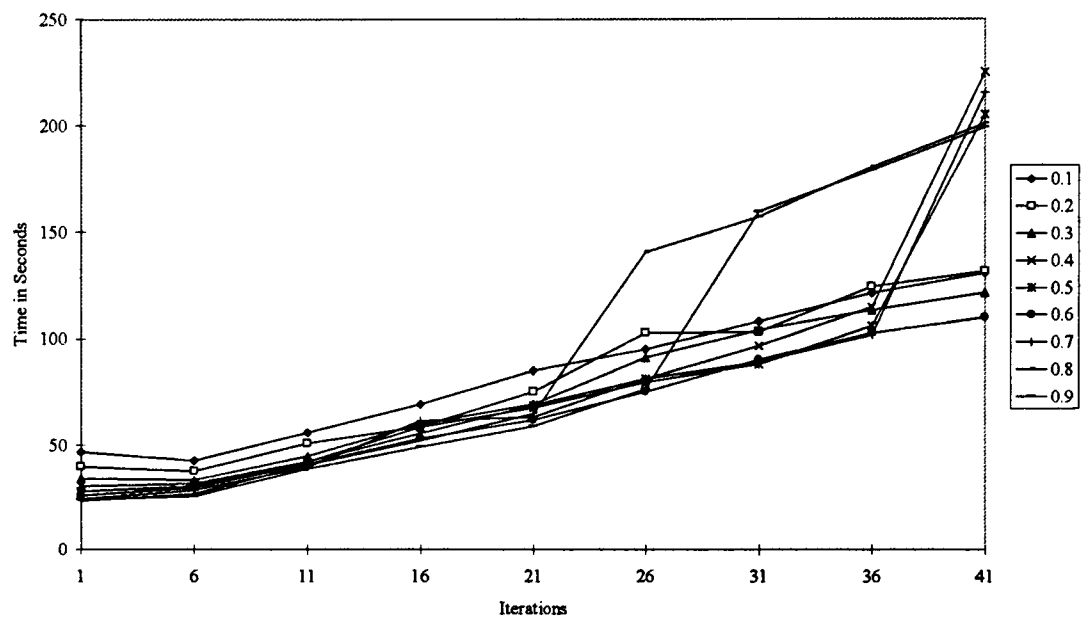


Figure 65 Hemisphere with Randomized Radii Packed with Hyperbolic Monotone Algorithm (iterations)

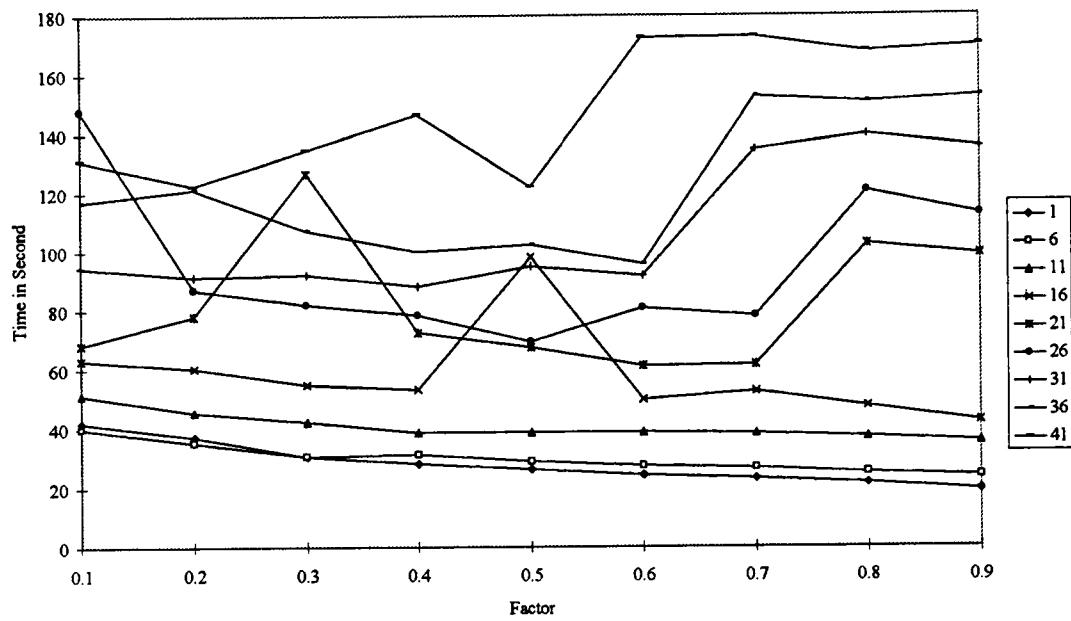


Figure 66 Oval with Randomized Radii Packed with Hyperbolic Monotone Algorithm (factor)

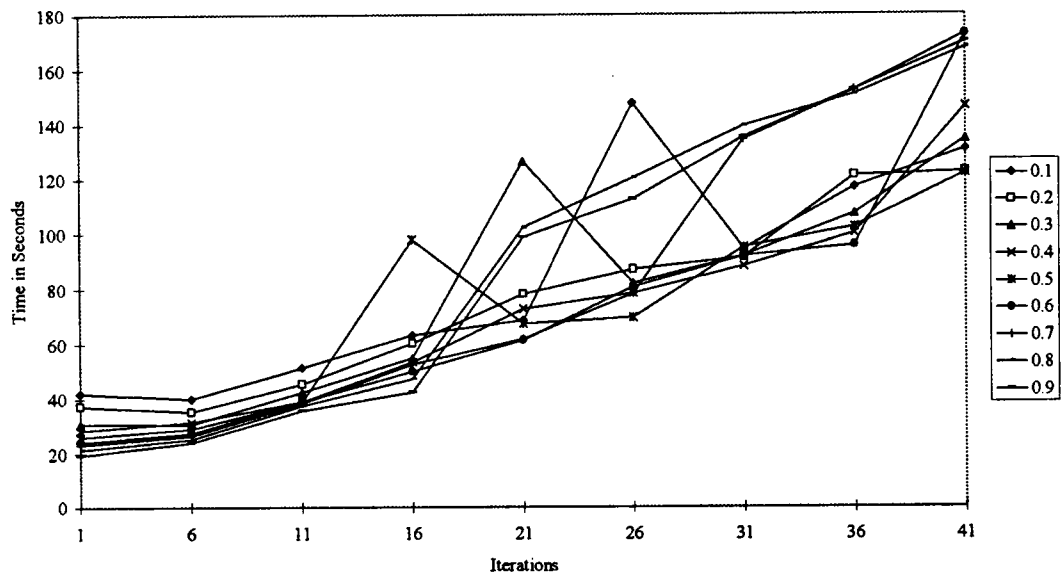


Figure 67 Oval with Randomized Radii Packed with Hyperbolic Monotone Algorithm (iterations)

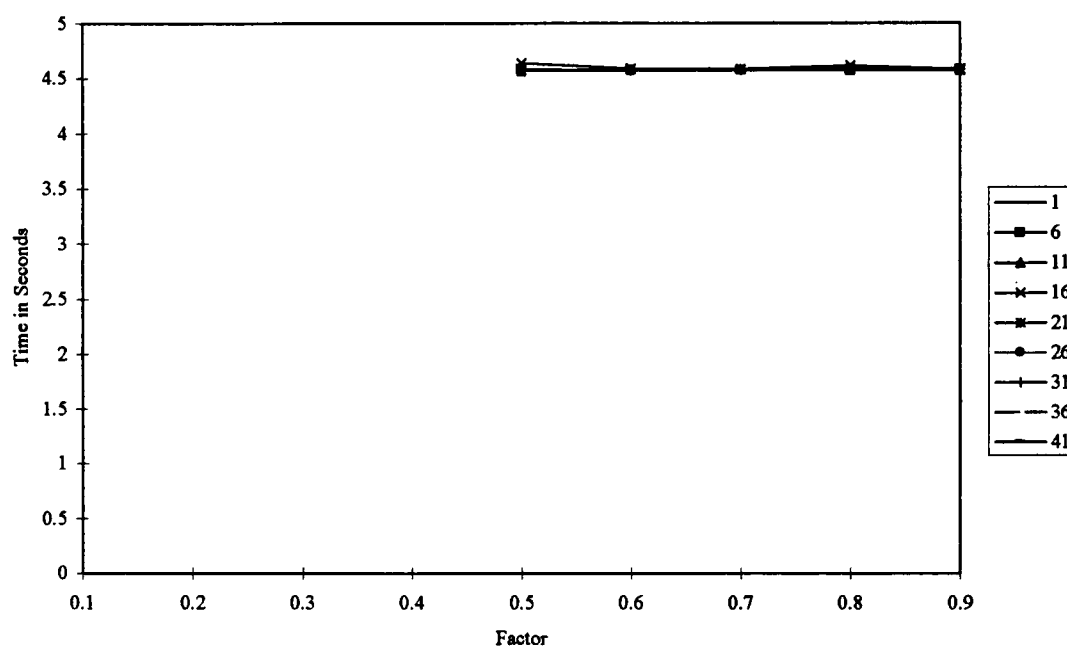


Figure 68 El with Andreev Radii Packed with Hyperbolic Newton's Method (factor)

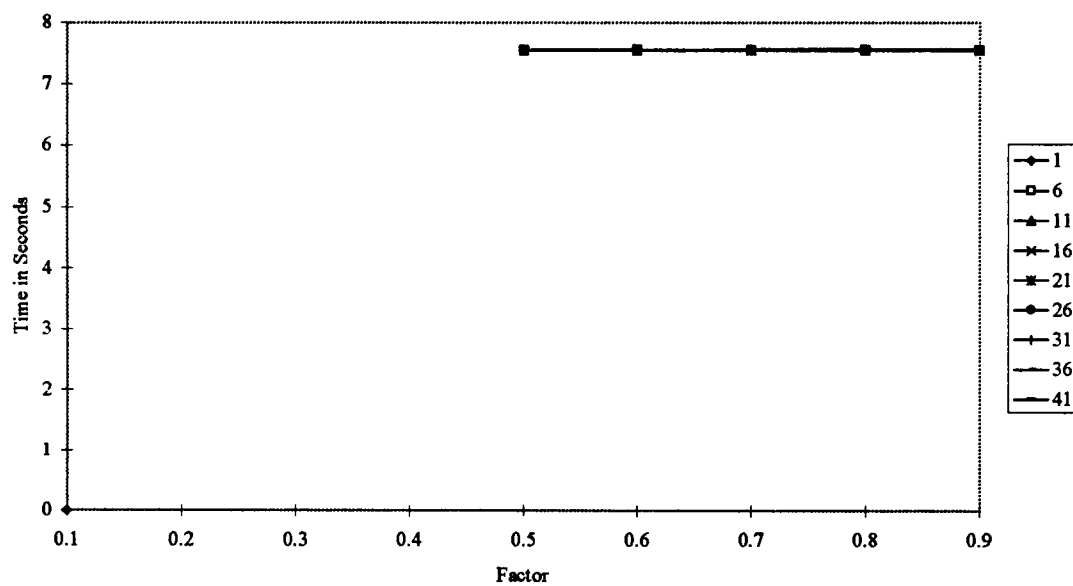


Figure 69 Hemisphere with Andreev Radii Packed with Hyperbolic Newton's Method (factor)

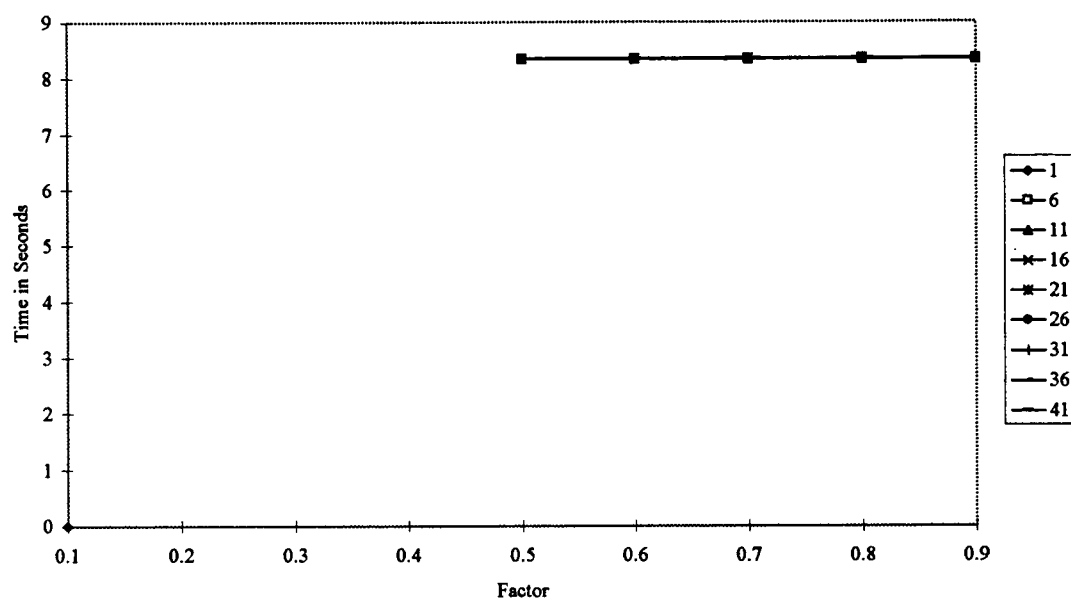


Figure 70 Oval with Andreev Radii Packed with Hyperbolic Newton's Method (factor)

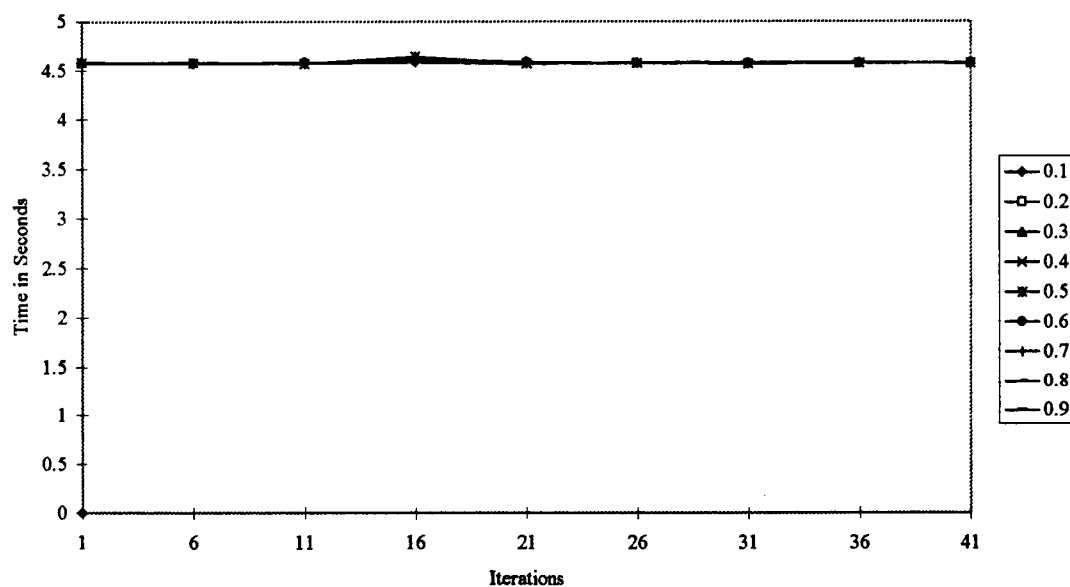


Figure 71 El with Andreev Radii Packed with Hyperbolic Newton's Method (iterations)

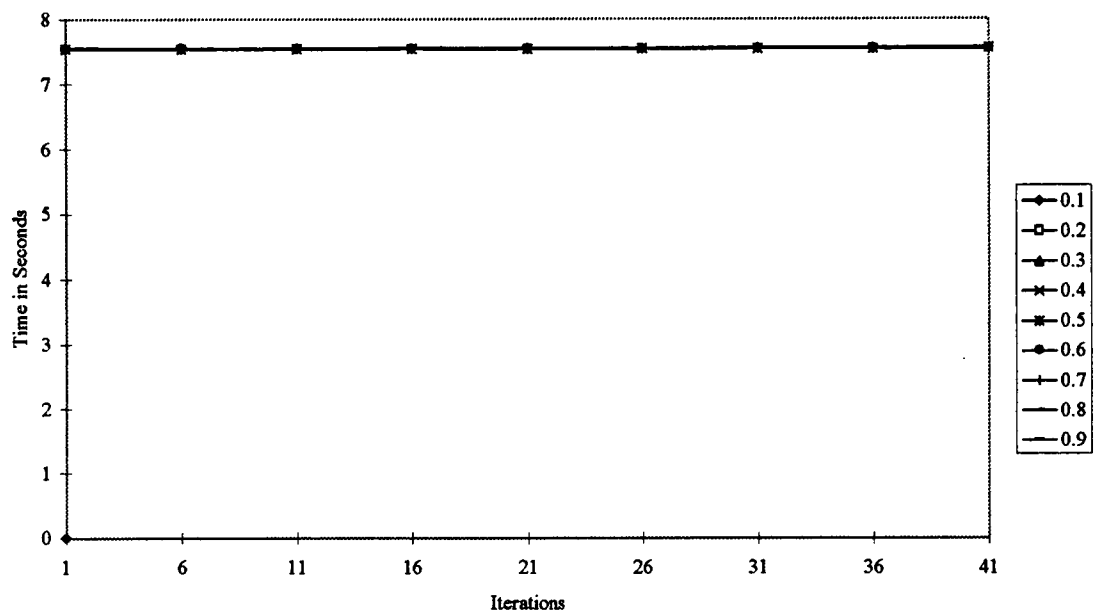


Figure 72 Hemisphere with Andreev Radii Packed with Hyperbolic Newton's Method (iterations)

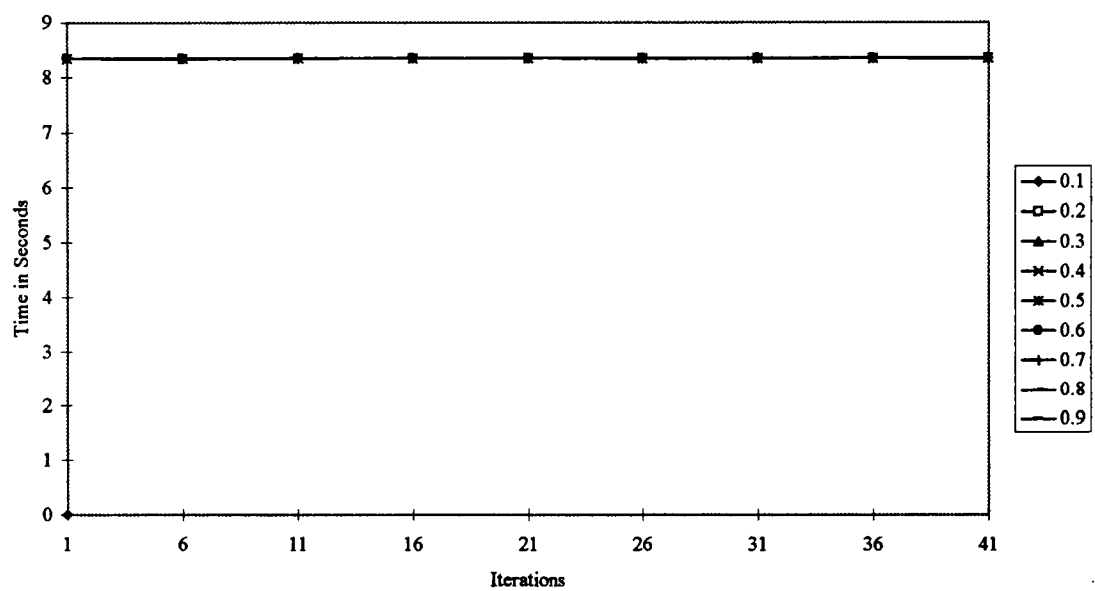


Figure 73 Oval with Andreev Radii Packed with Hyperbolic Newton's Method (iterations)

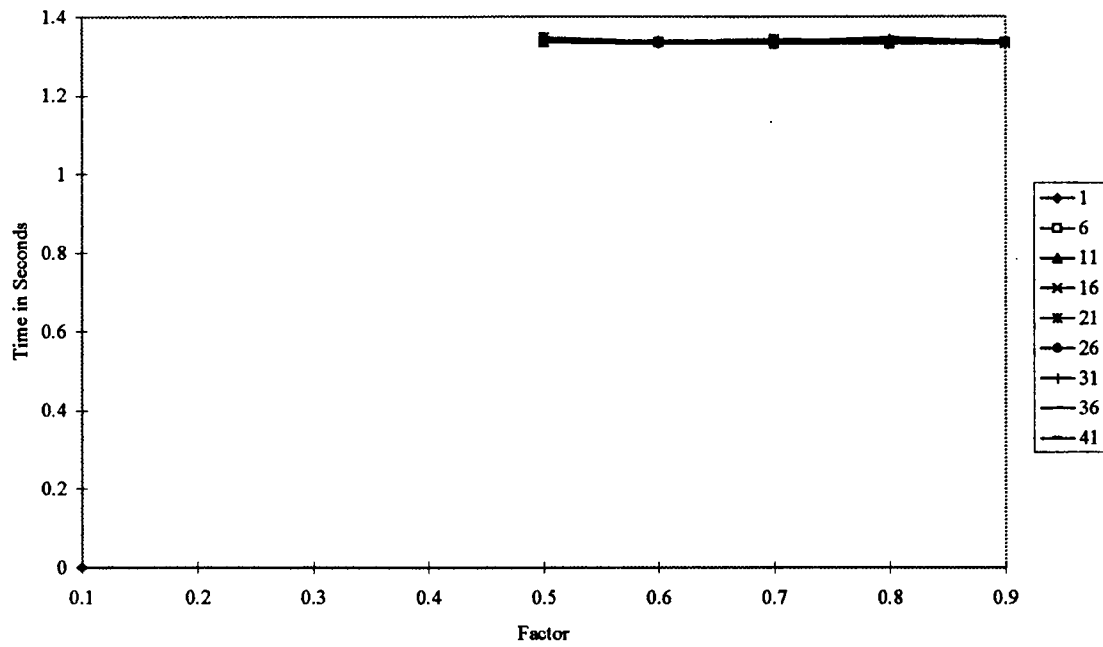


Figure 74 Small Hex, Lh2, Packed with Hyperbolic Newton's Method (factor)

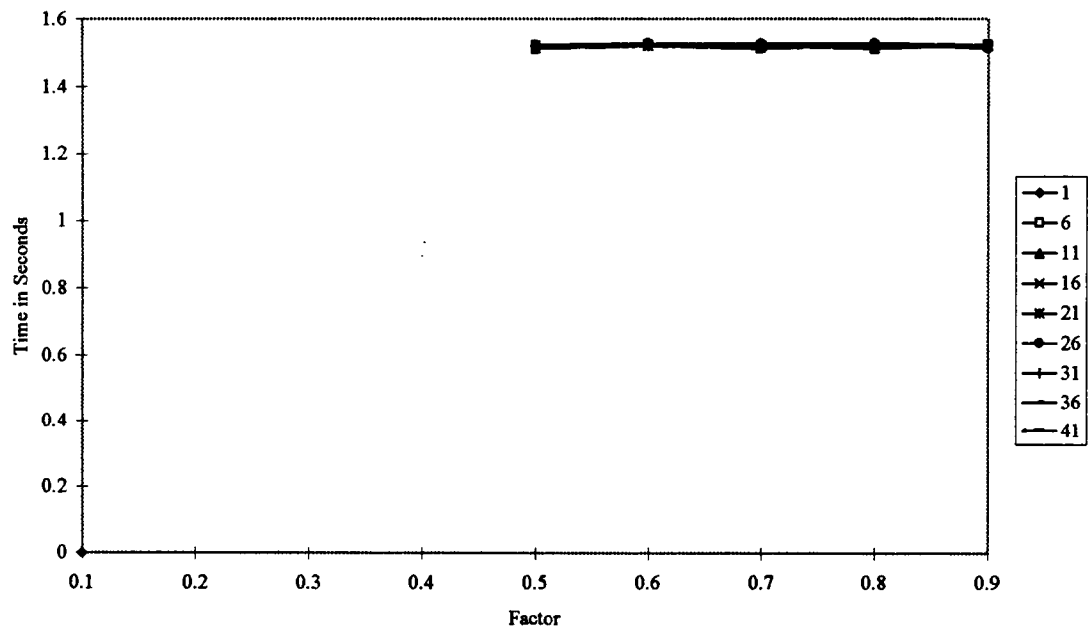


Figure 75 Small Hex, Lh3, Packed with Hyperbolic Newton's Method (factor)

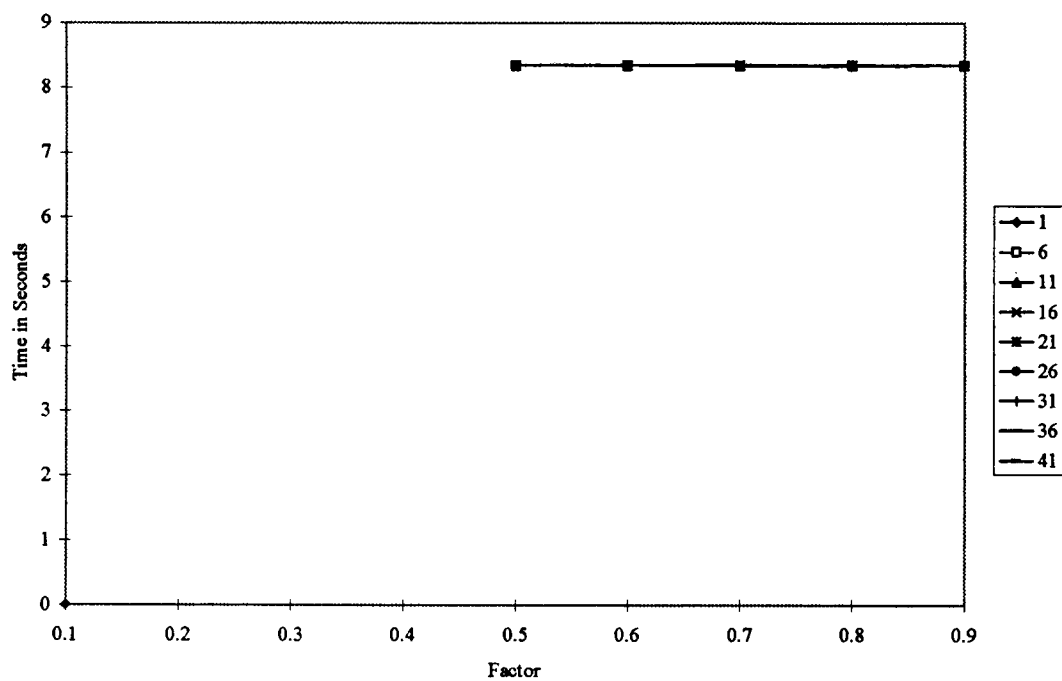


Figure 76 Small Hex, Lh4, Packed with Hyperbolic Newton's Method (factor)

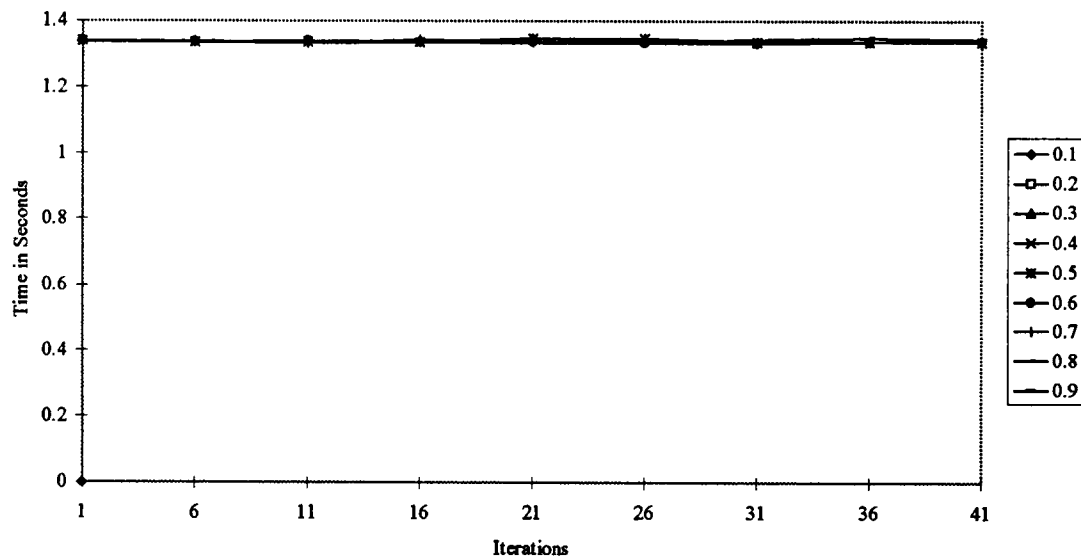


Figure 77 Small Hex, Lh2, Packed with Hyperbolic Newton's Method (iterations)

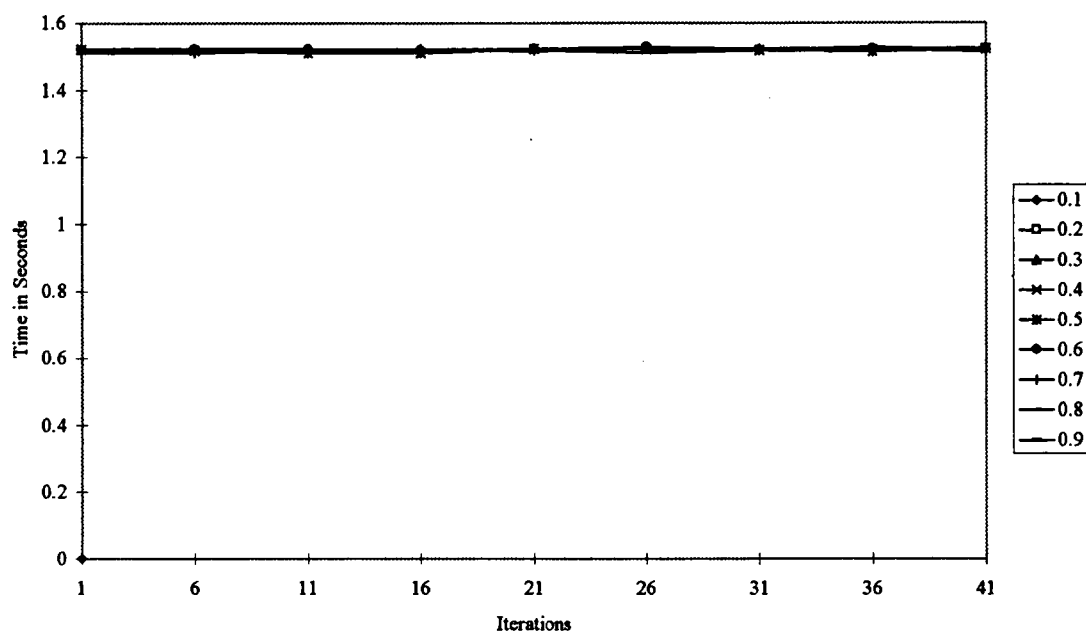


Figure 78 Small Hex, Lh3, Packed with Hyperbolic Newton's Method (iterations)

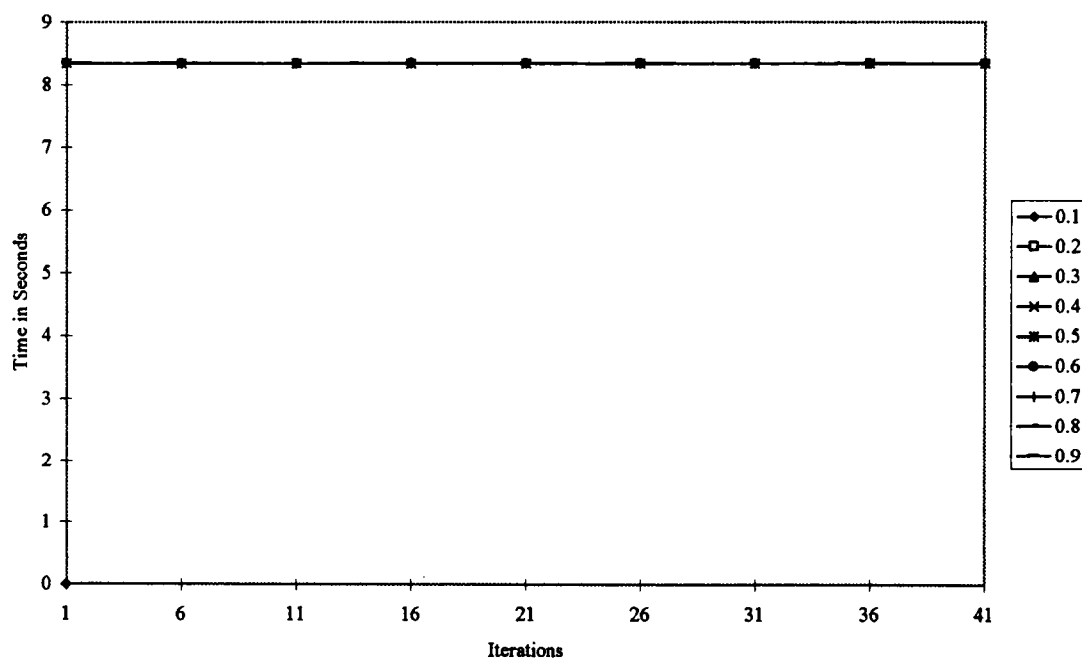


Figure 79 Small Hex, Lh4, Packed with Hyperbolic Newton's Method (iterations)

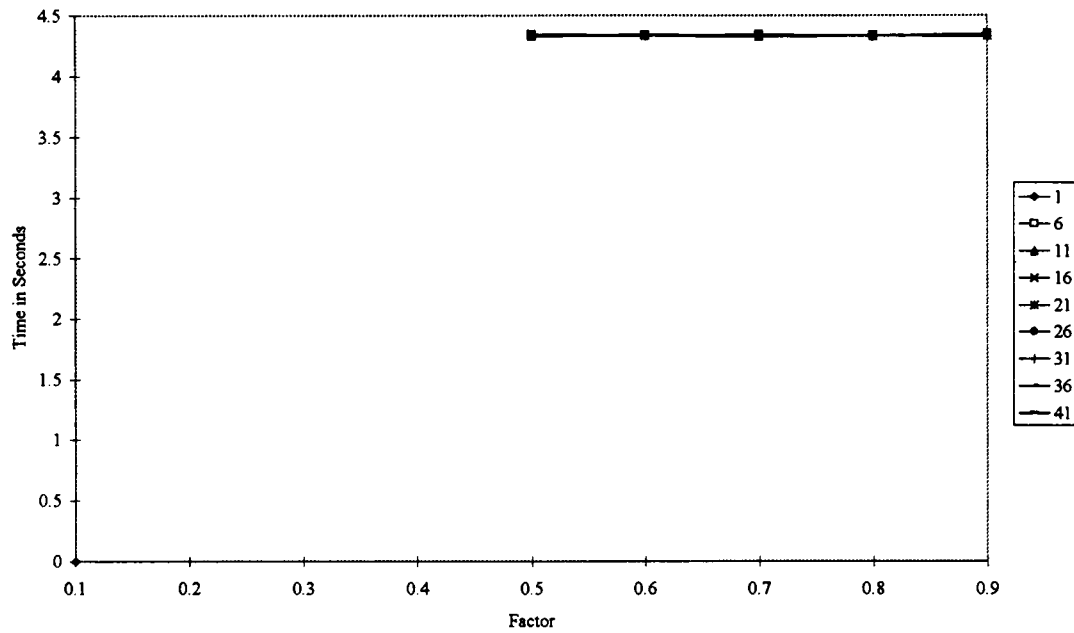


Figure 80 Cross with Randomized Radii Packed with Hyperbolic Newton's Method (factor)

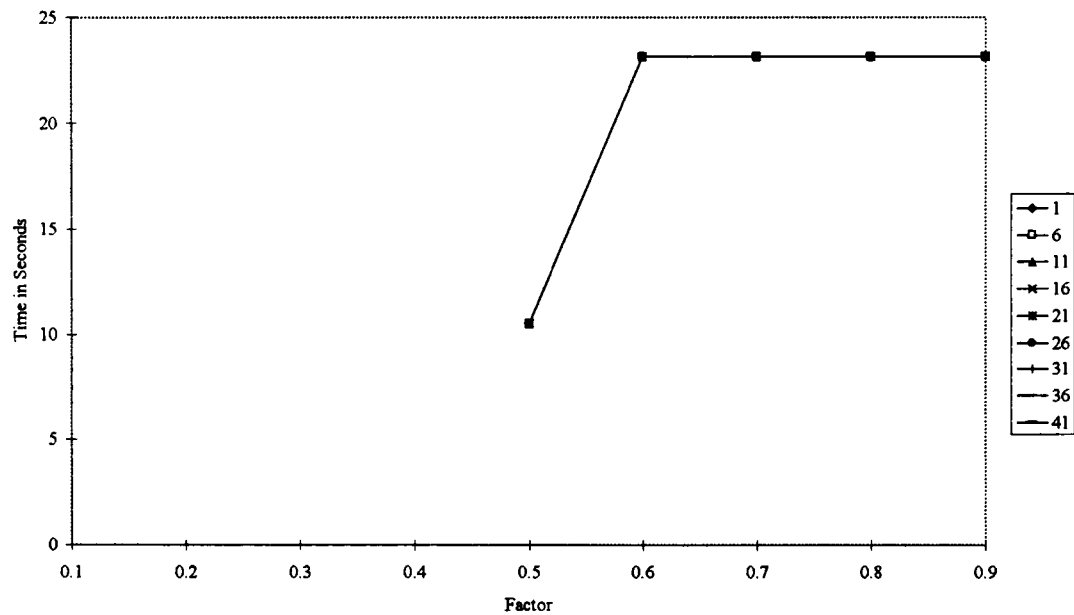


Figure 81 Hemisphere with Randomized Radii Packed with Hyperbolic Newton's Method (factor)

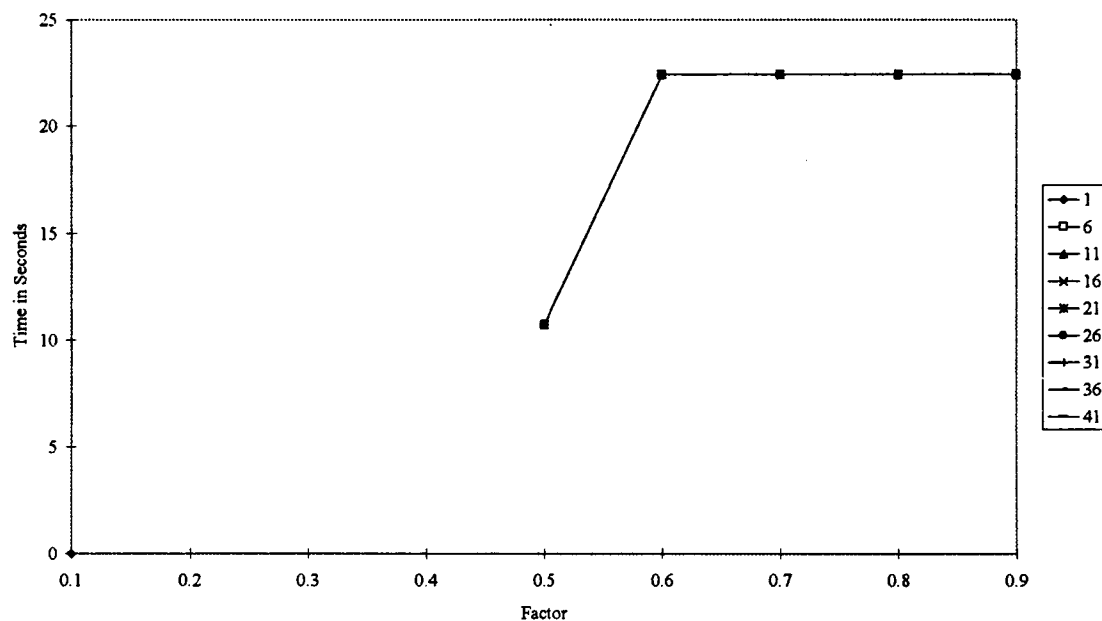


Figure 82 Oval with Randomized Radii Packed with Hyperbolic Newton's Method (factor)

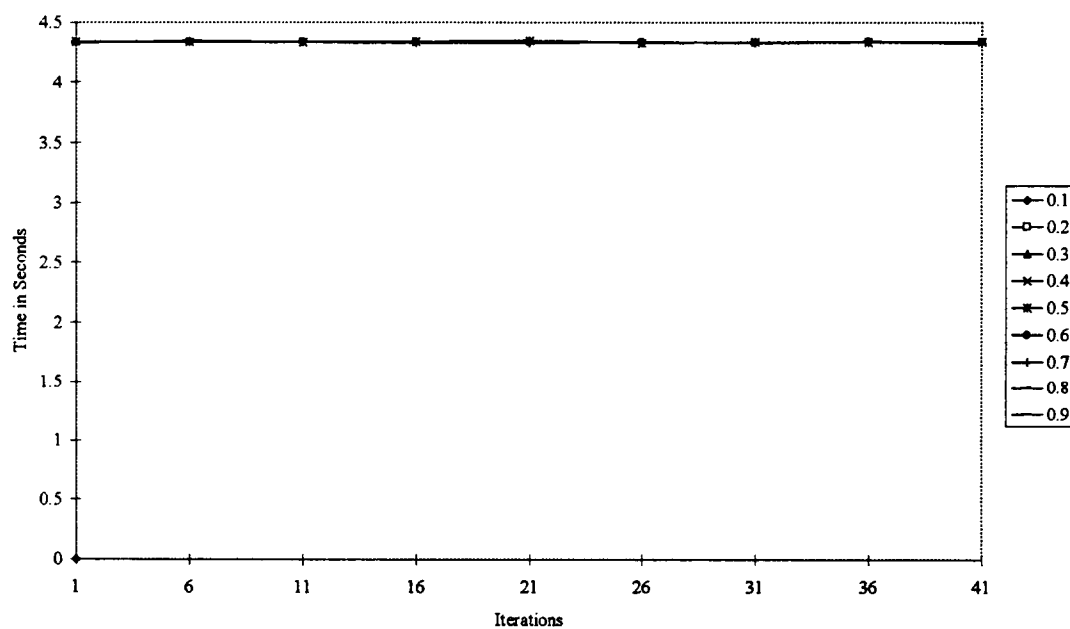


Figure 83 Cross with Randomized Radii Packed with Hyperbolic Newton's Method (iterations)

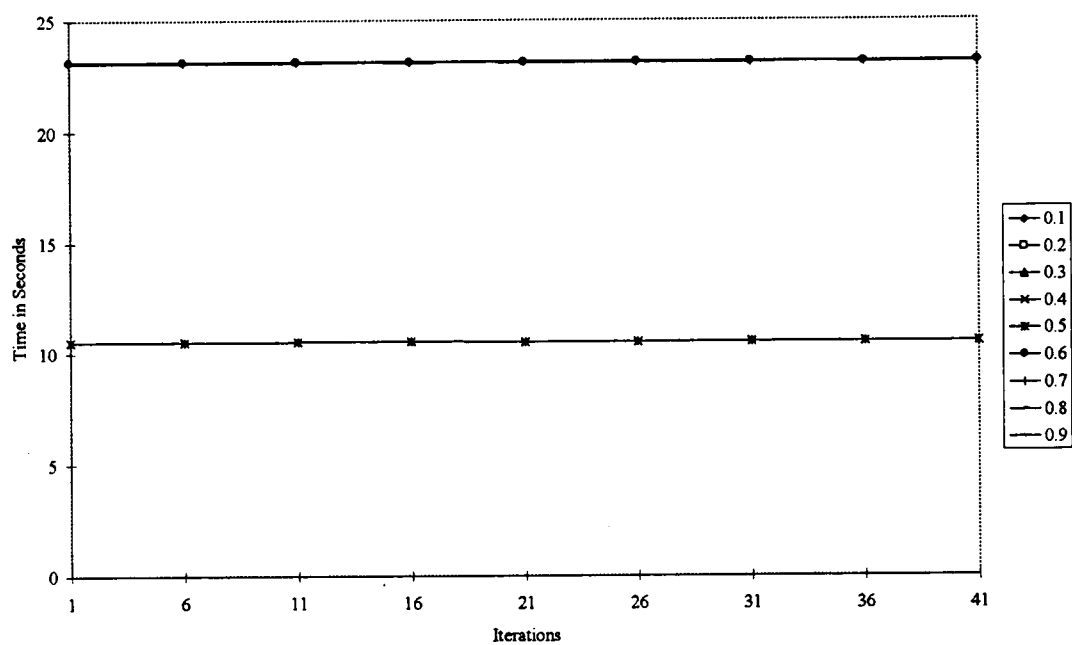


Figure 84 Hemisphere with Randomized Radii Packed with Hyperbolic Newton's Method (iterations)

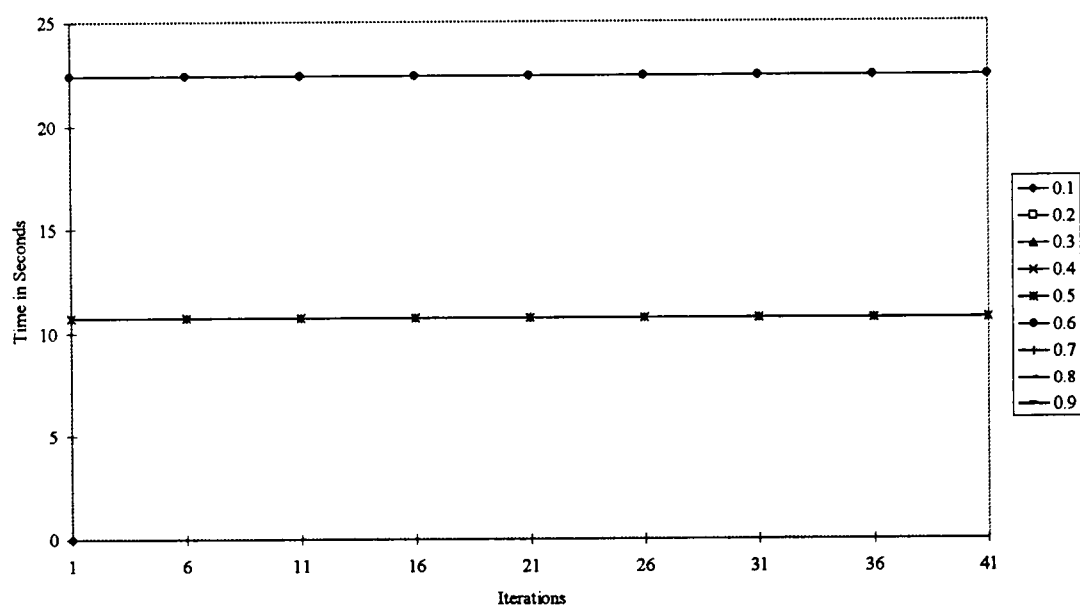


Figure 85 Oval with Randomized Radii Packed with Hyperbolic Newton's Method (iterations)

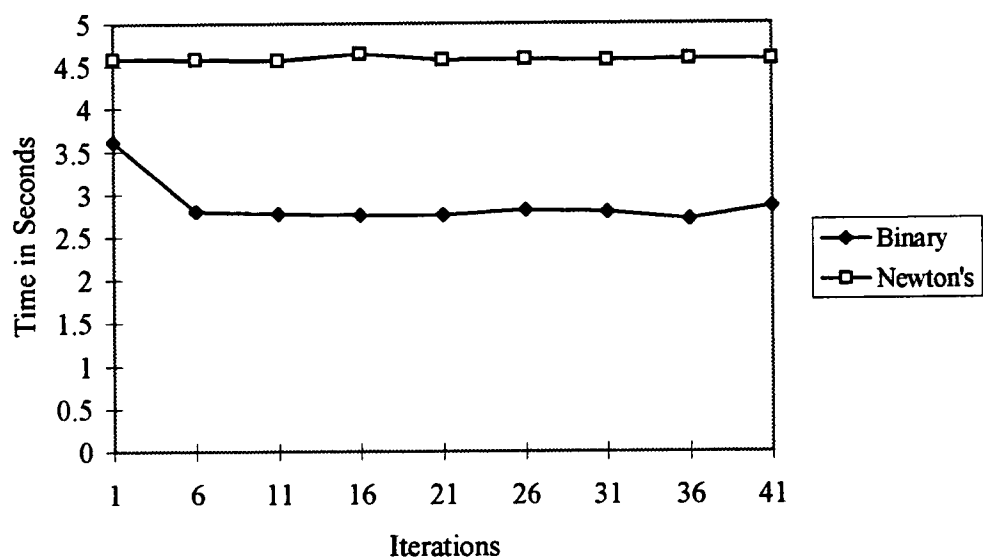


Figure 86 Comparison of the Hyperbolic Algorithms with the Andreiev El Packing

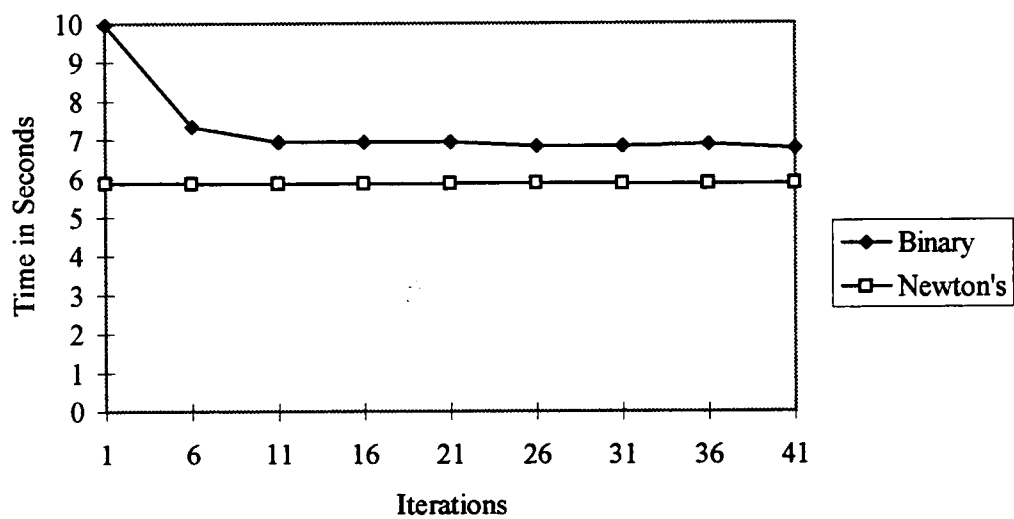


Figure 87 Comparison of the Hyperbolic Algorithms with the Randomized El Packing

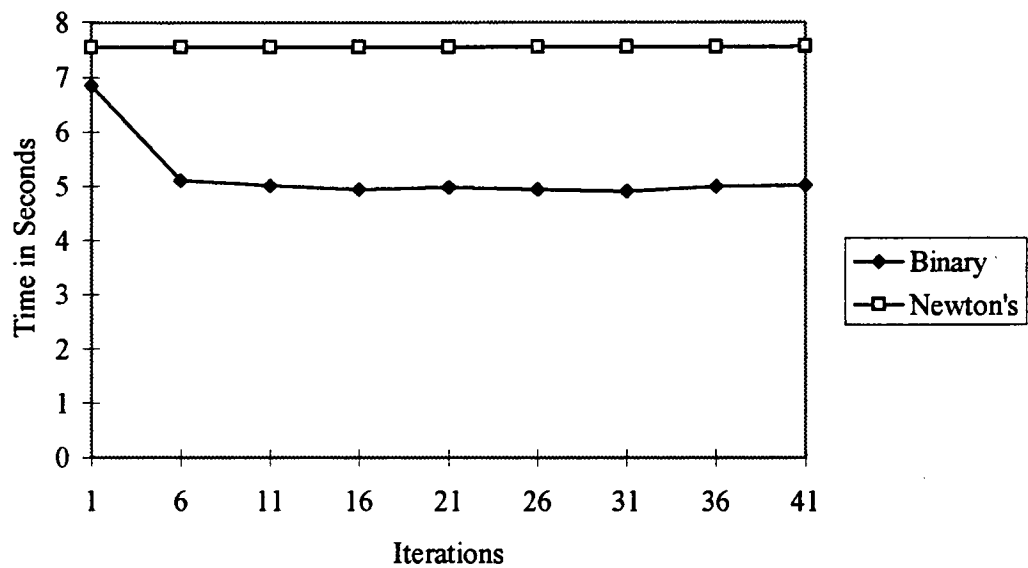


Figure 88 Comparison of the Hyperbolic Algorithms with the Andreev Hemisphere Packing

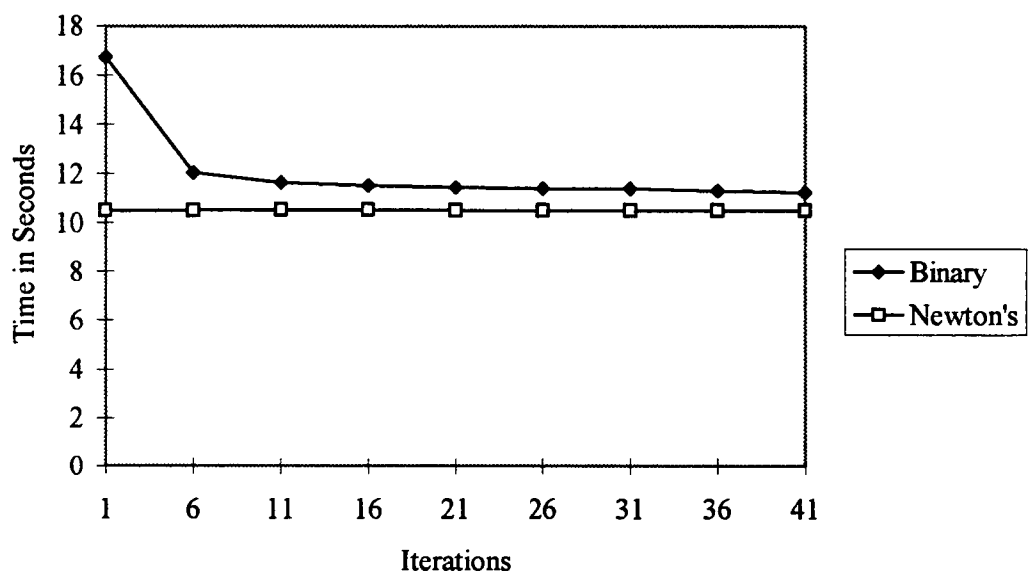


Figure 89 Comparison of the Hyperbolic Algorithms with the Randomized Hemisphere Packing

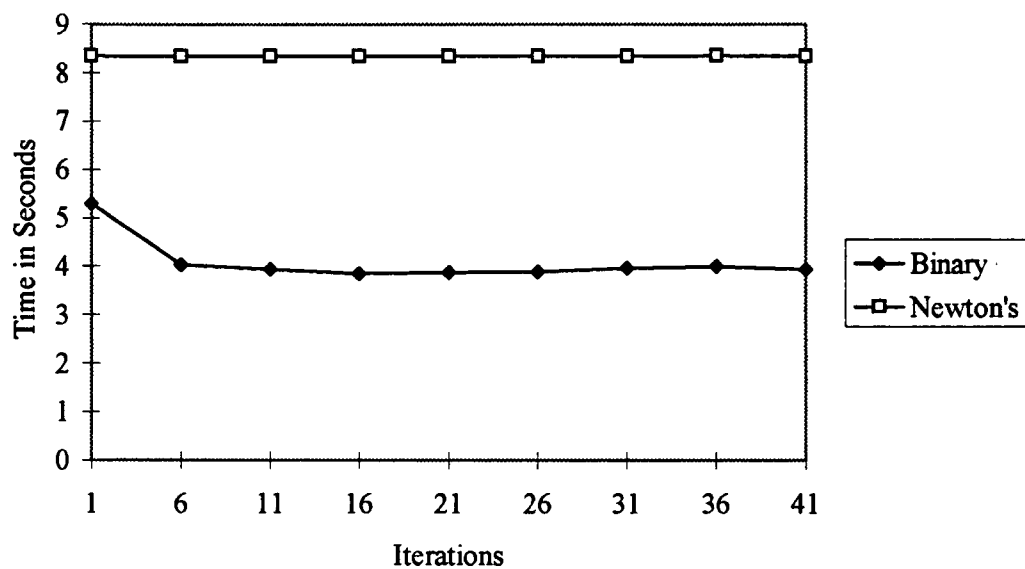


Figure 90 Comparison of the Hyperbolic Algorithms with the Andreev Oval Packing

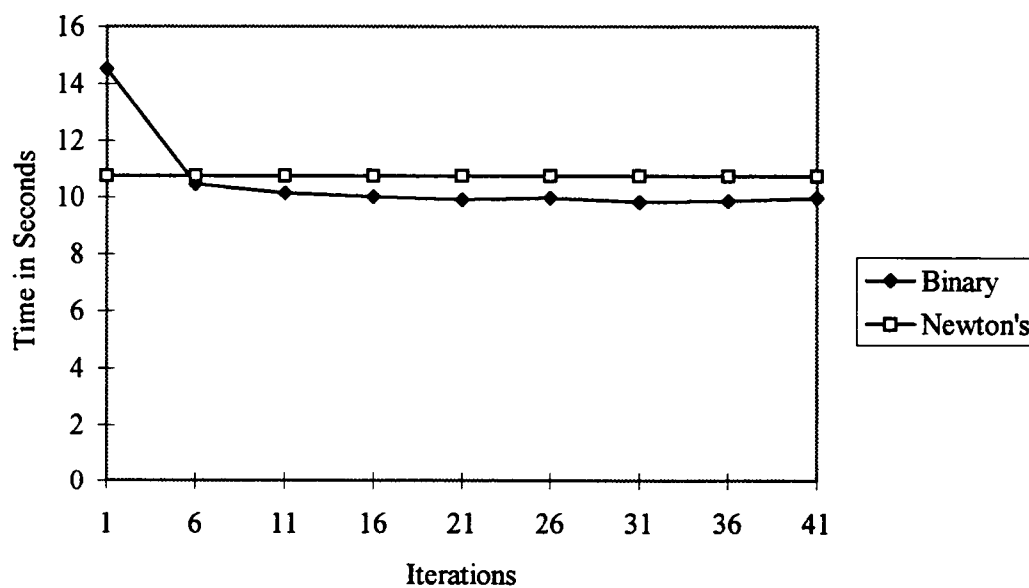


Figure 91 Comparison of the Hyperbolic Algorithms with the Randomized Oval Packing

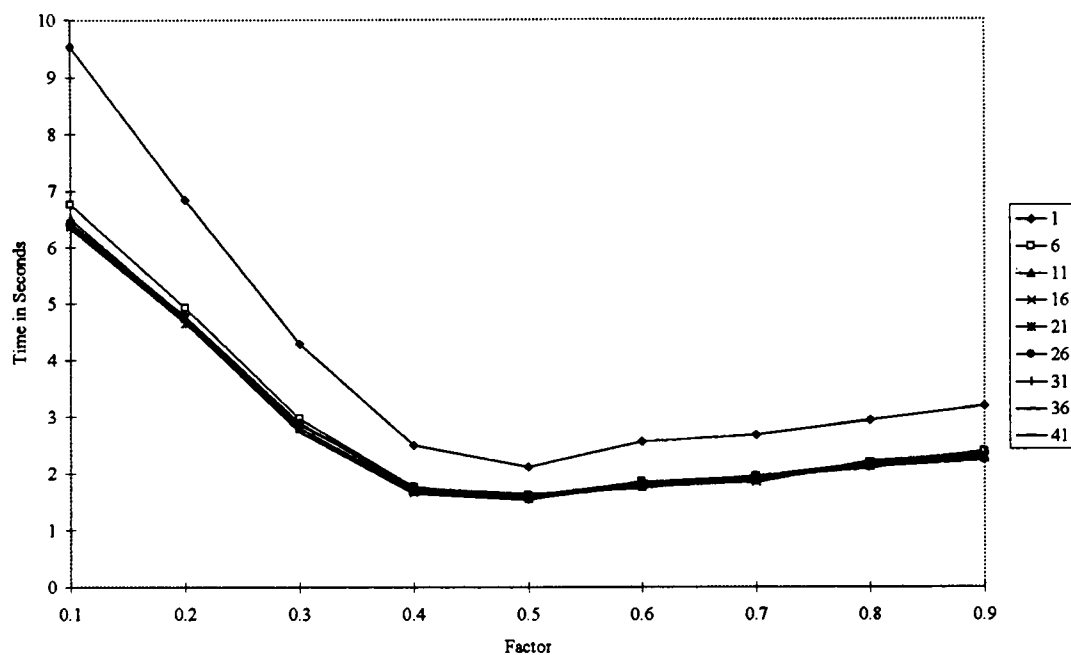


Figure 92 Small Hex, Lh3, Packed with Euclidean Binary Algorithm (factor)

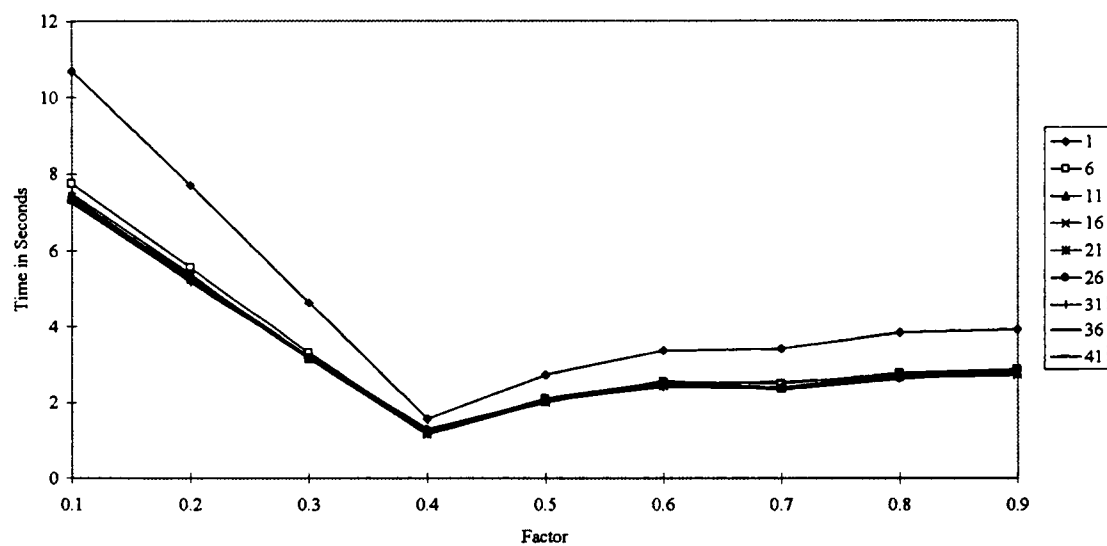


Figure 93 Small Hex, Lh4, Packed with Euclidean Binary Algorithm (factor)

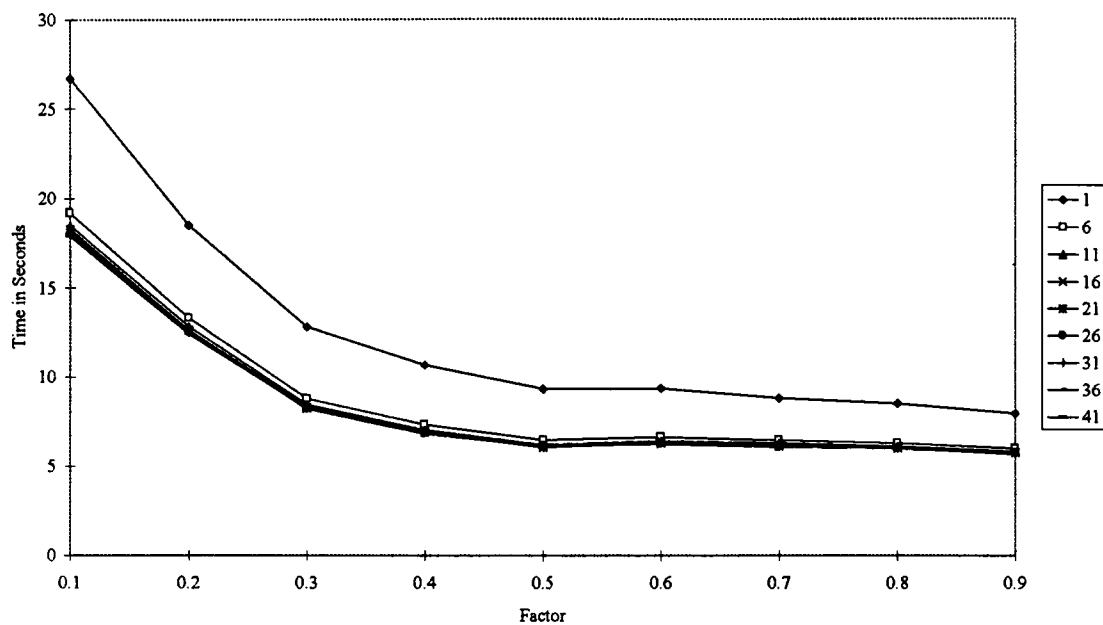


Figure 94 El with Randomized Radii, Packed with Euclidean Binary Algorithm (factor)

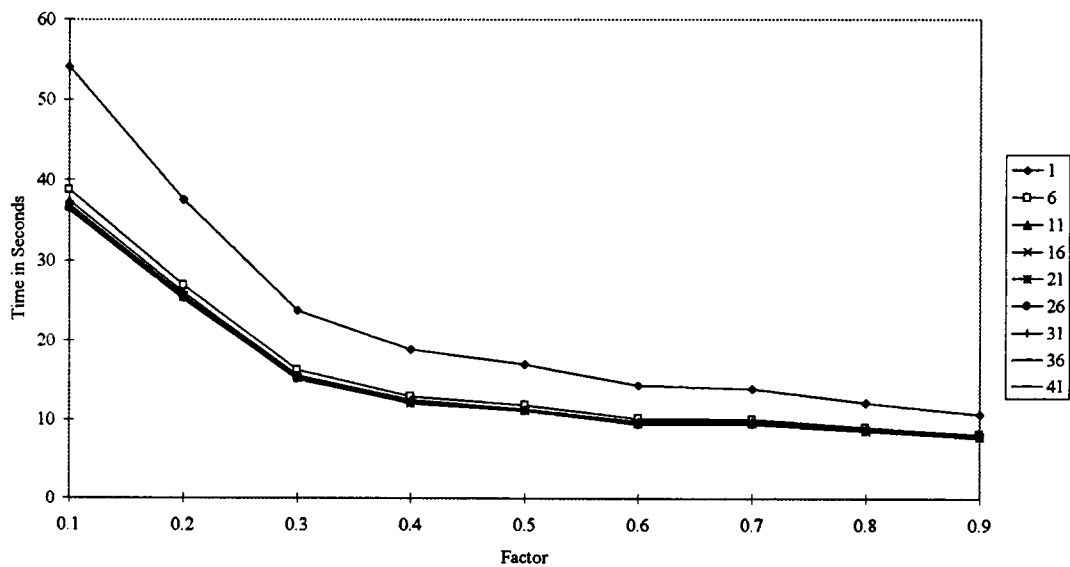


Figure 95 Hemisphere with Randomized Radii, Packed with Euclidean Binary Algorithm (factor)

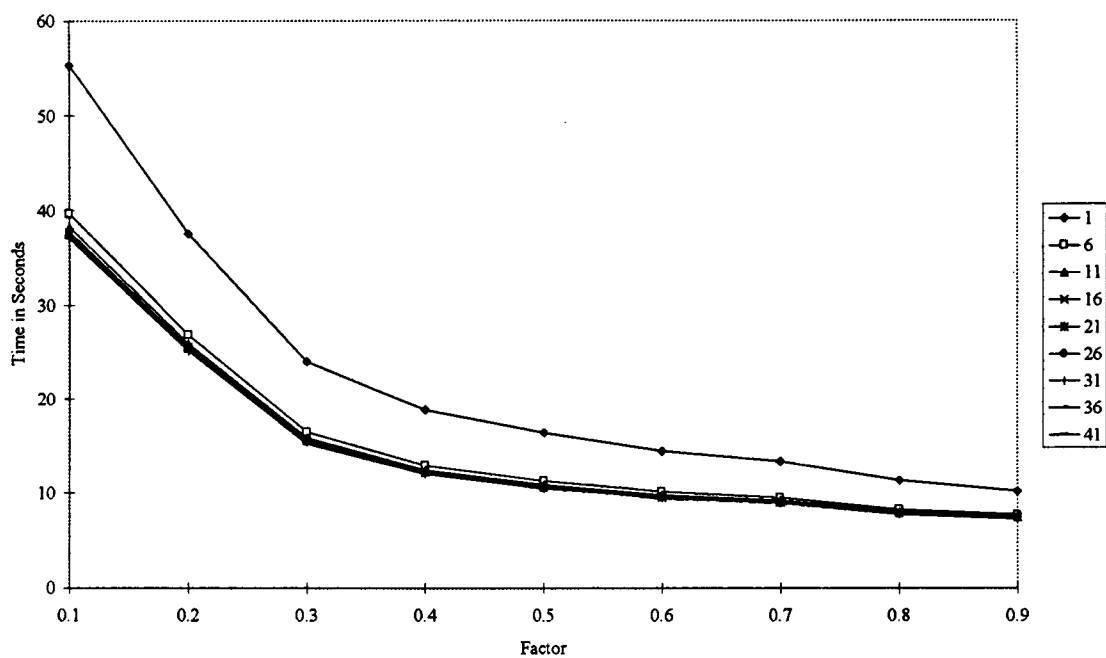


Figure 96 Oval with Randomized Radii, Packed with Euclidean Binary Algorithm (factor)

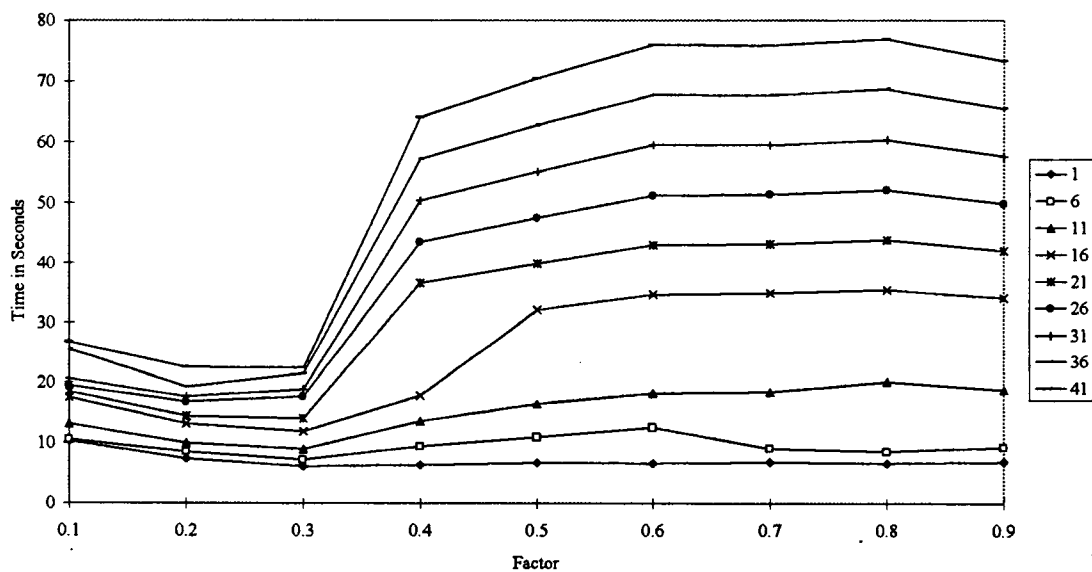


Figure 97 Small Hex, Lh2, Packed with Euclidean Monotone Algorithm (factor)

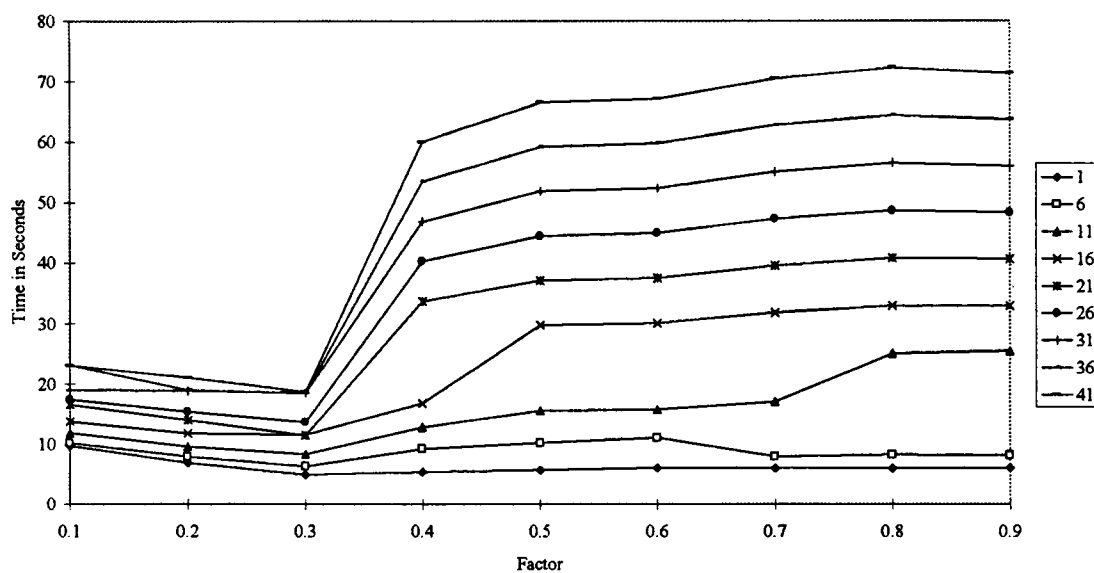


Figure 98 Small Hex, Lh3, Packed with Euclidean Monotone Algorithm (factor)

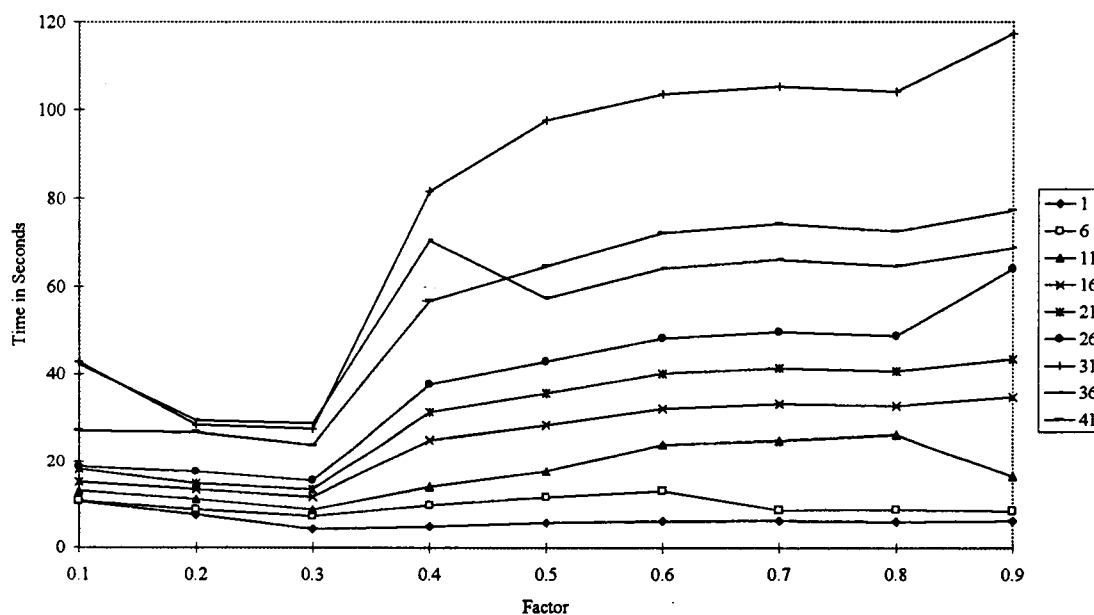


Figure 99 Small Hex, Lh4, Packed with Euclidean Monotone Algorithm (factor)

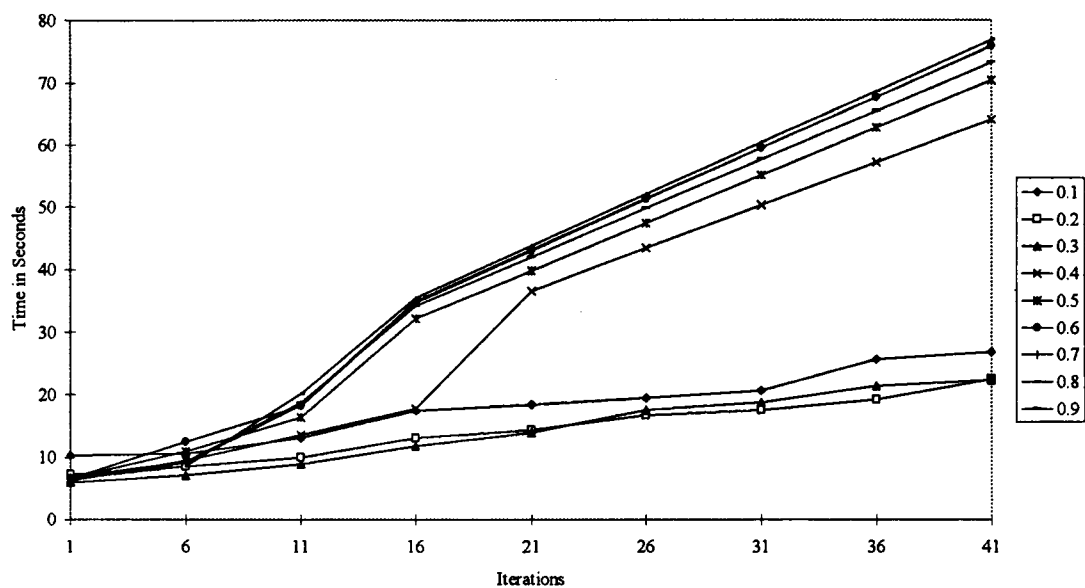


Figure 100 Small Hex, Lh2, Packed with Euclidean Monotone Algorithm (iterations)

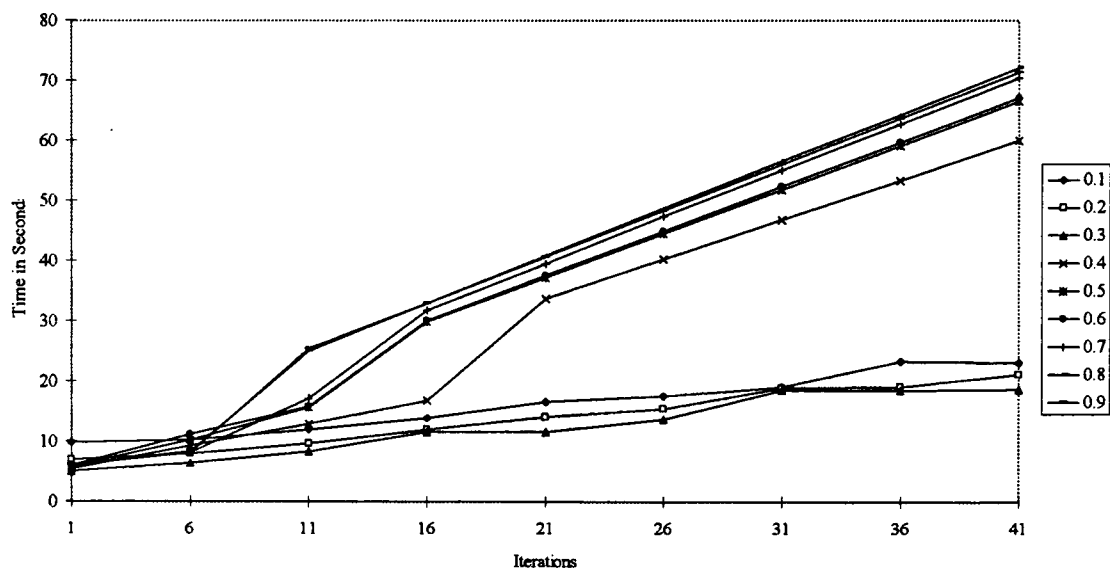


Figure 101 Small Hex, Lh3, Packed with Euclidean Monotone Algorithm (iterations)

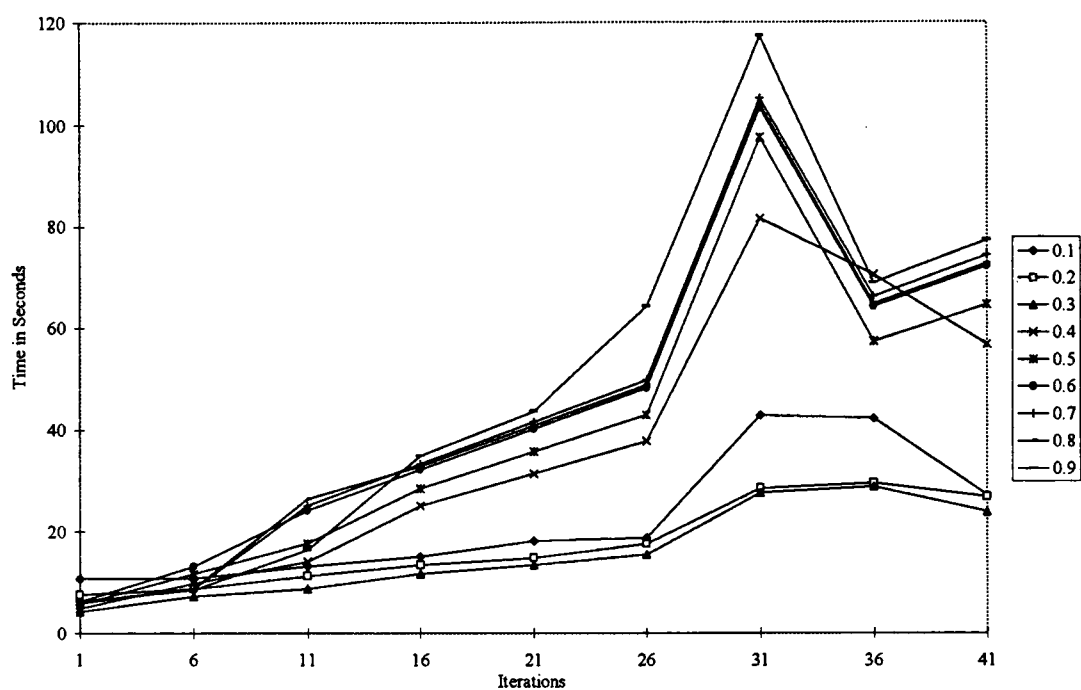


Figure 102 Small Hex, Lh4, Packed with Euclidean Monotone Algorithm (iterations)

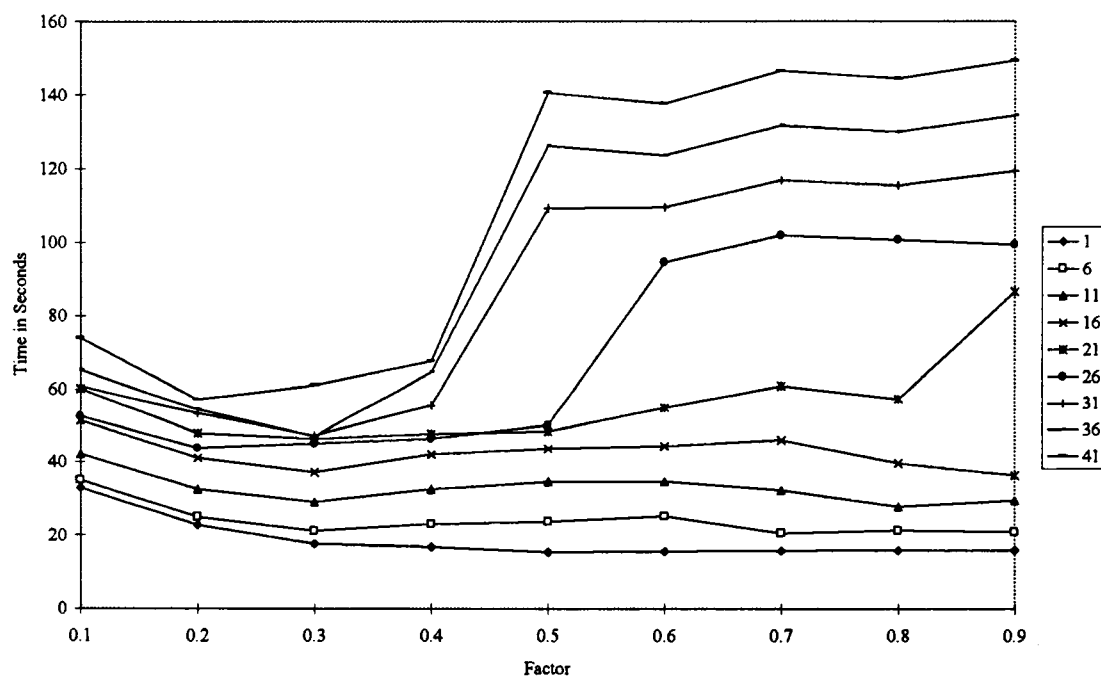


Figure 103 El with Randomized Radii Packed with Euclidean Monotone Algorithm (factor)

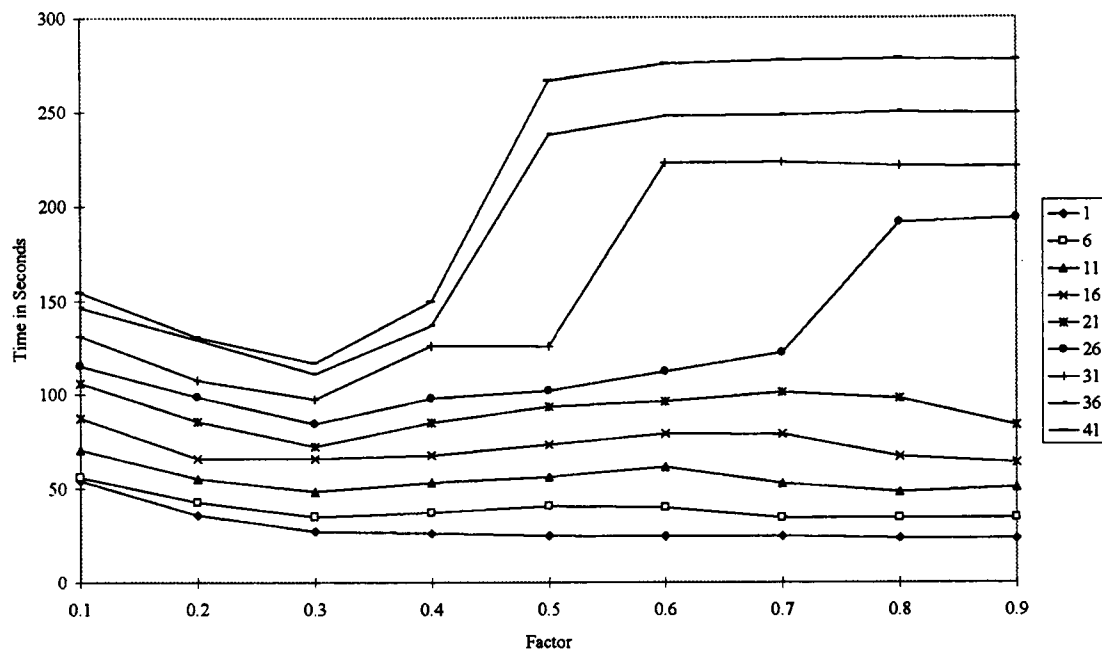


Figure 104 Hemisphere with Randomized Radii Packed with Euclidean Monotone Algorithm (factor)

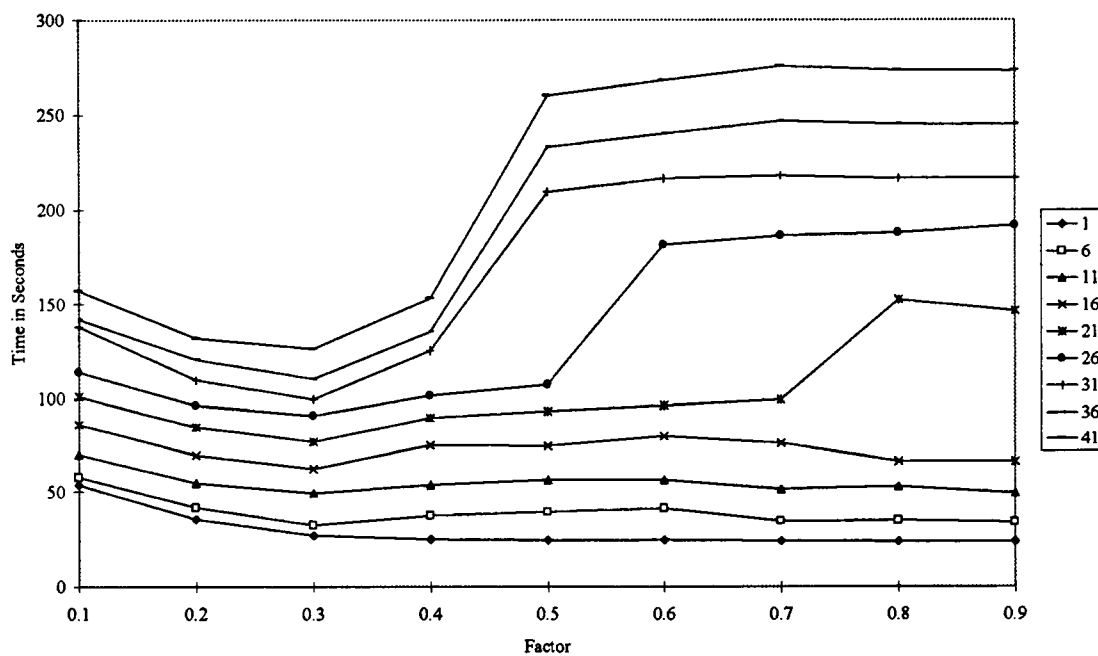


Figure 105 Oval with Randomized Radii Packed with Euclidean Monotone Algorithm (factor)

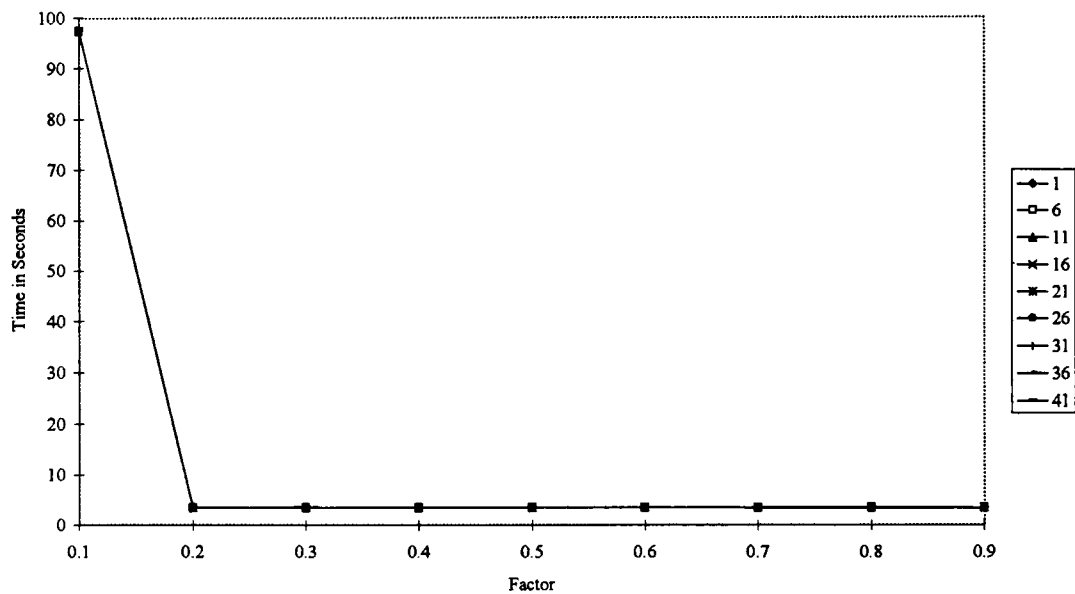


Figure 106 Small hex, Lh2, Packed with Euclidean Newton's Method Algorithm (factor)

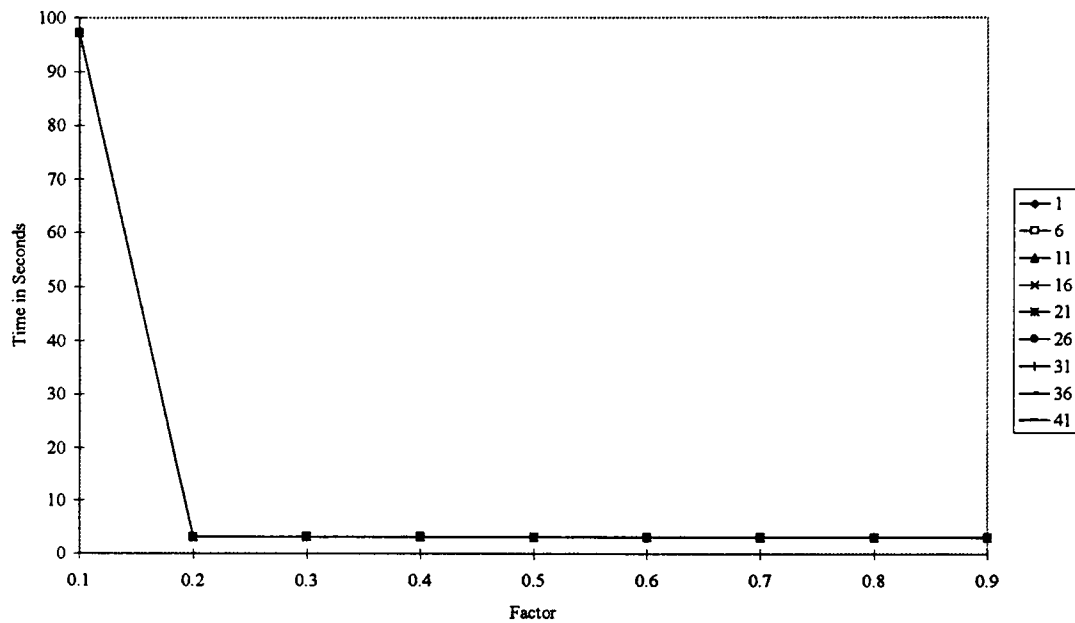


Figure 107 Small hex, Lh3, Packed with Euclidean Newton's Method Algorithm (factor)

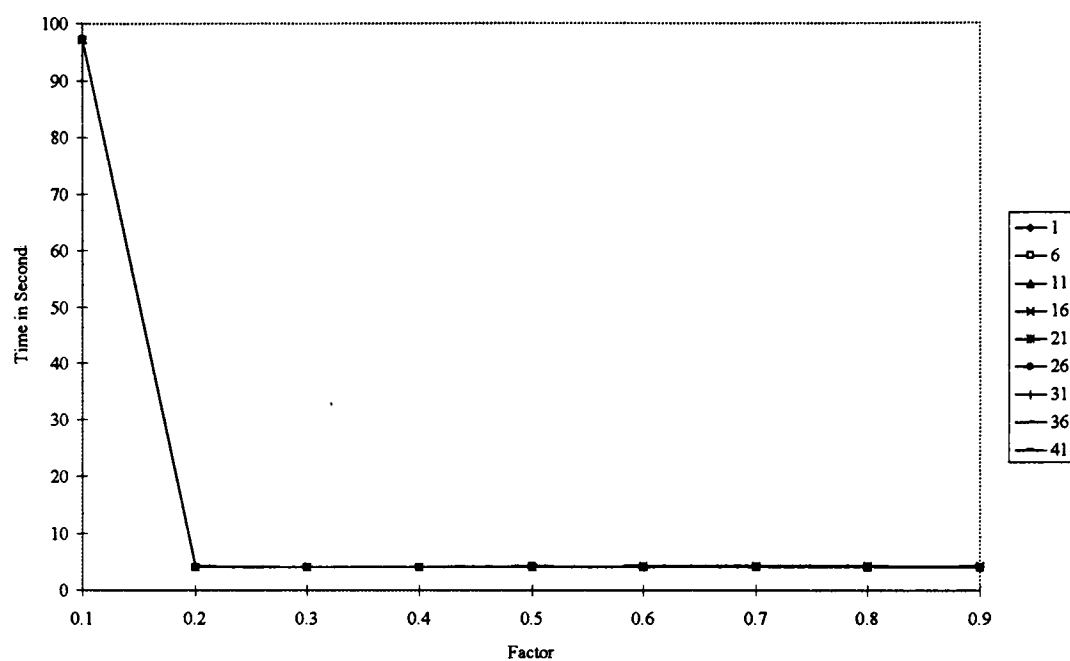


Figure 108 Small hex, Lh4, Packed with Euclidean Newton's Method Algorithm (factor)

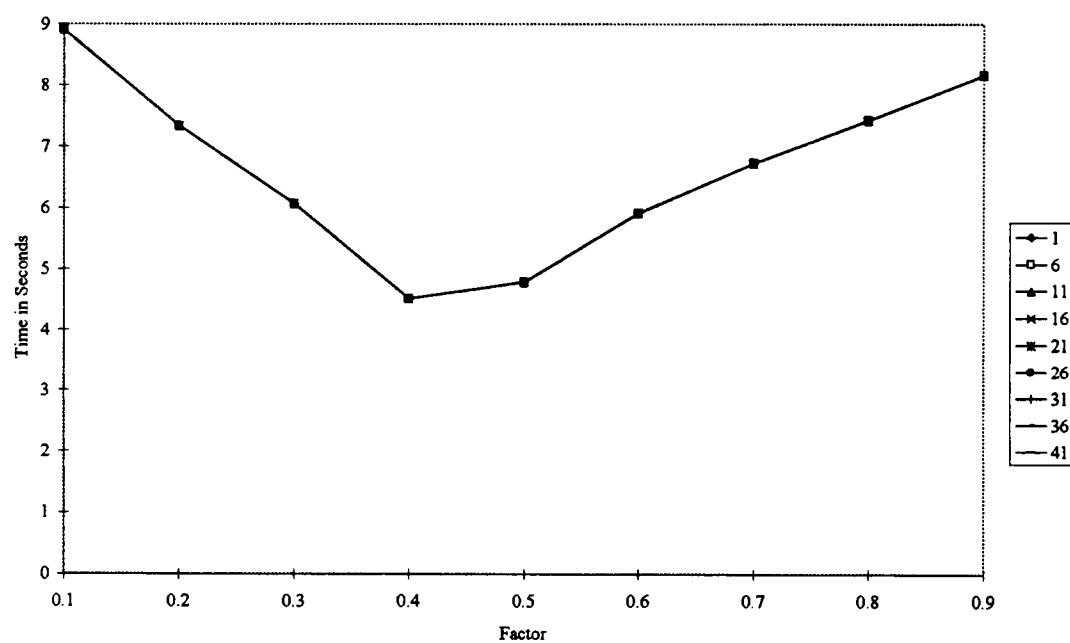


Figure 109 El with Randomized Radii Packed with Euclidean Newton's Method Algorithm (factor)

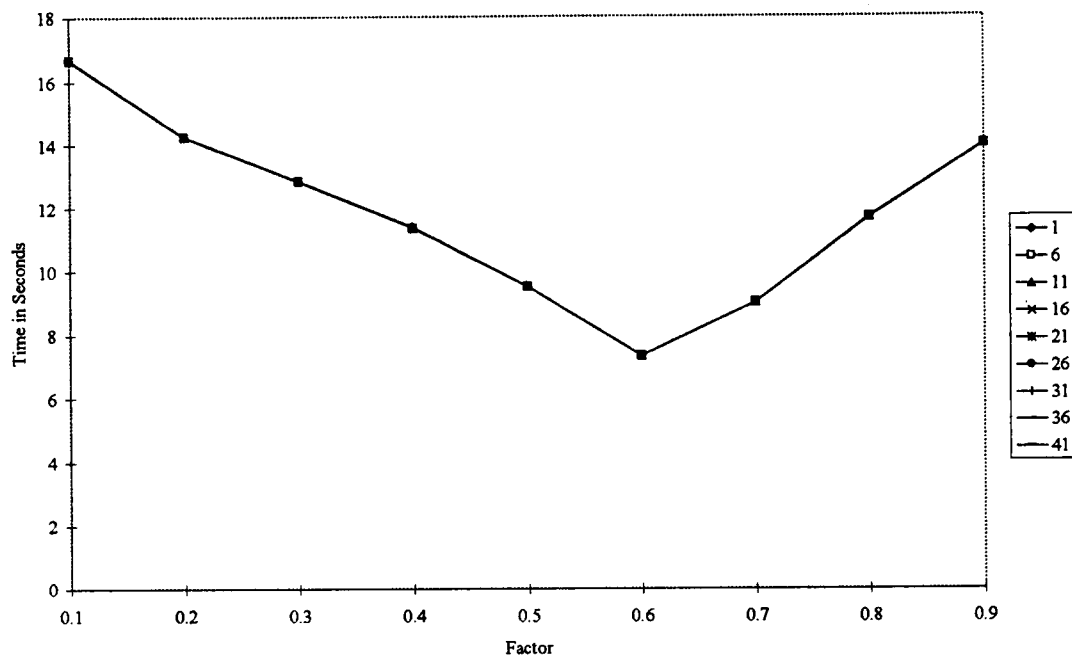


Figure 110 Hemisphere with Randomized Radii Packed with Euclidean Newton's Method Algorithm (factor)

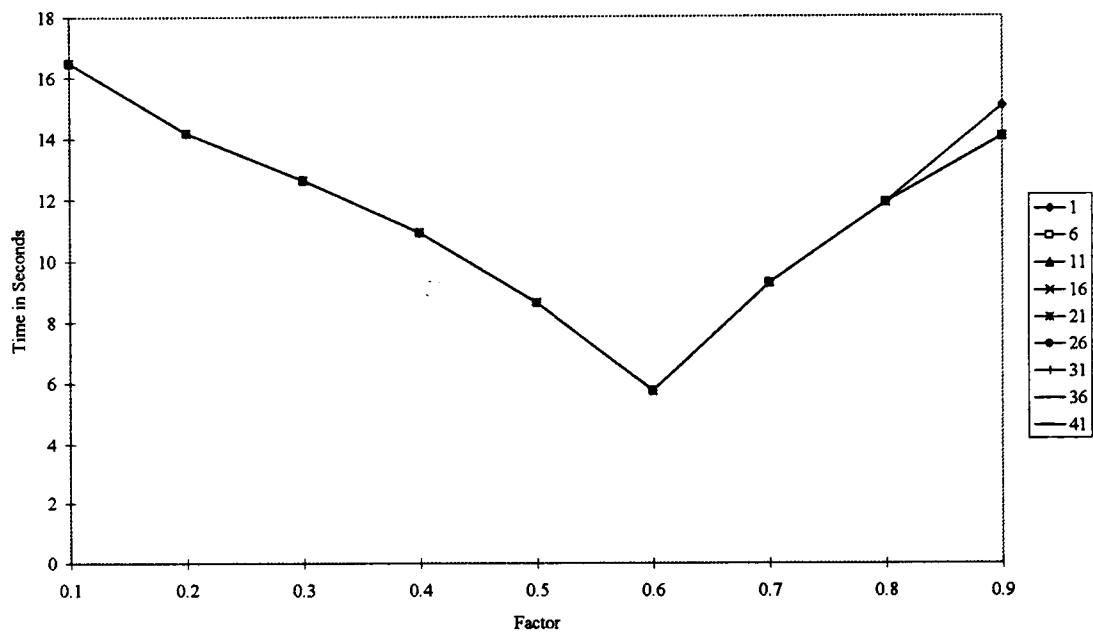


Figure 111 Oval with Randomized Radii Packed with Euclidean Newton's Method Algorithm (factor)

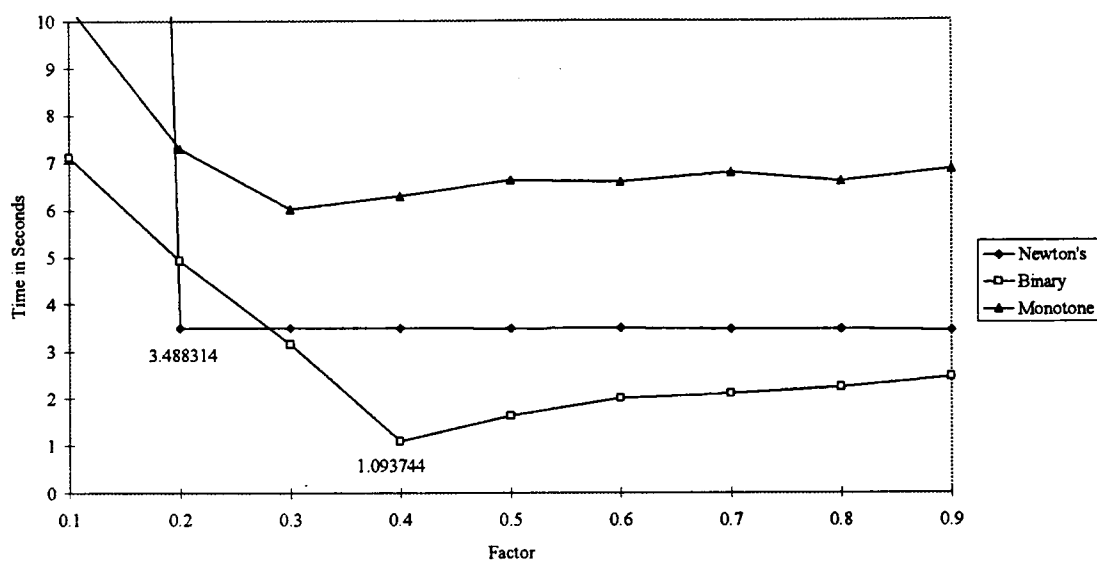


Figure 112 Comparison of Euclidean Binary and Newton's Method Algorithms, Small Hex Lh2

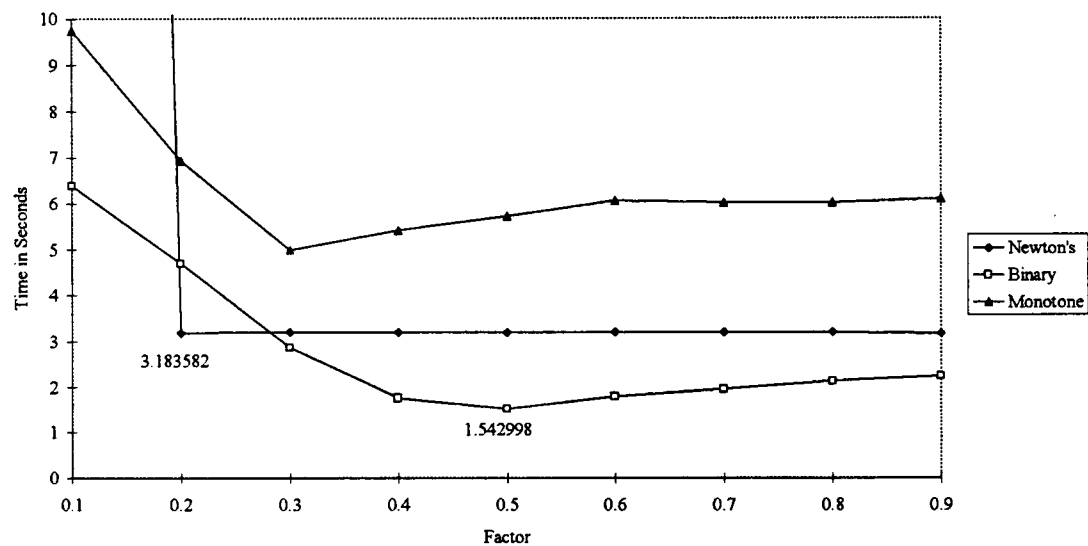


Figure 113 Comparison of Euclidean Binary and Newton's Method Algorithms, Small Hex Lh3

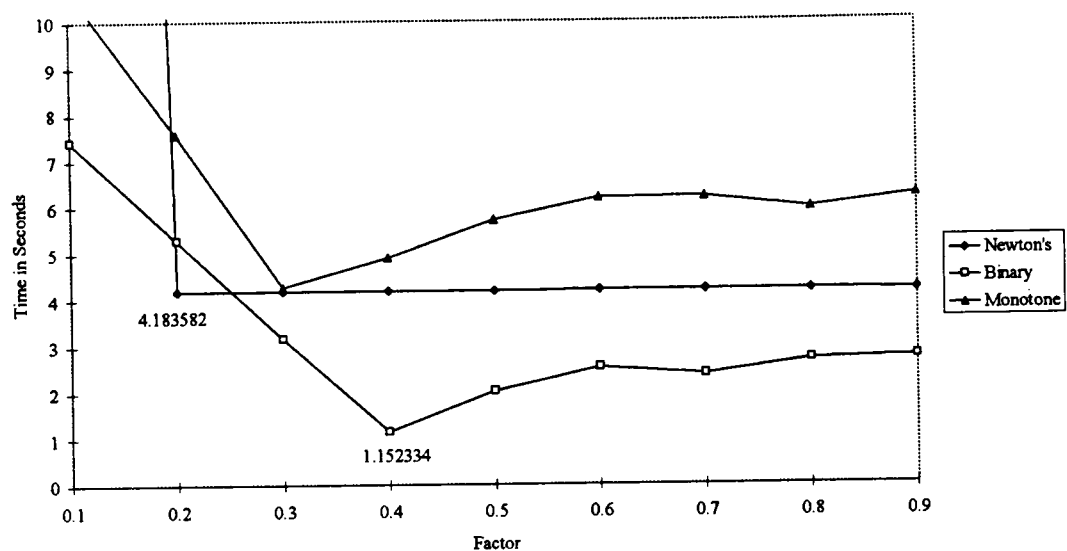


Figure 114 Comparison of Euclidean Binary and Newton's Method Algorithms, Small Hex Lh4

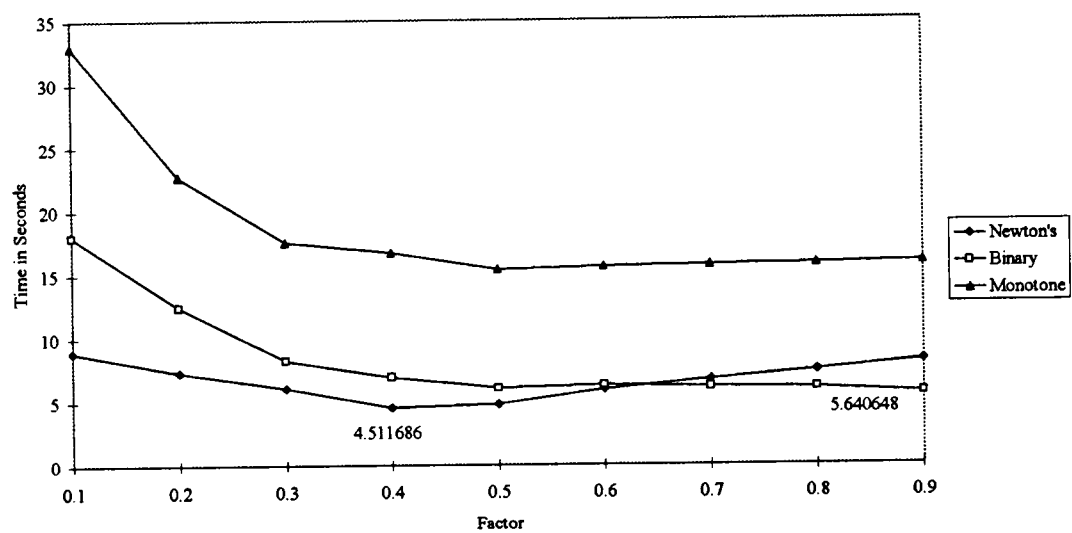


Figure 115 Comparison of Euclidean Binary and Newton's Method Algorithms, El

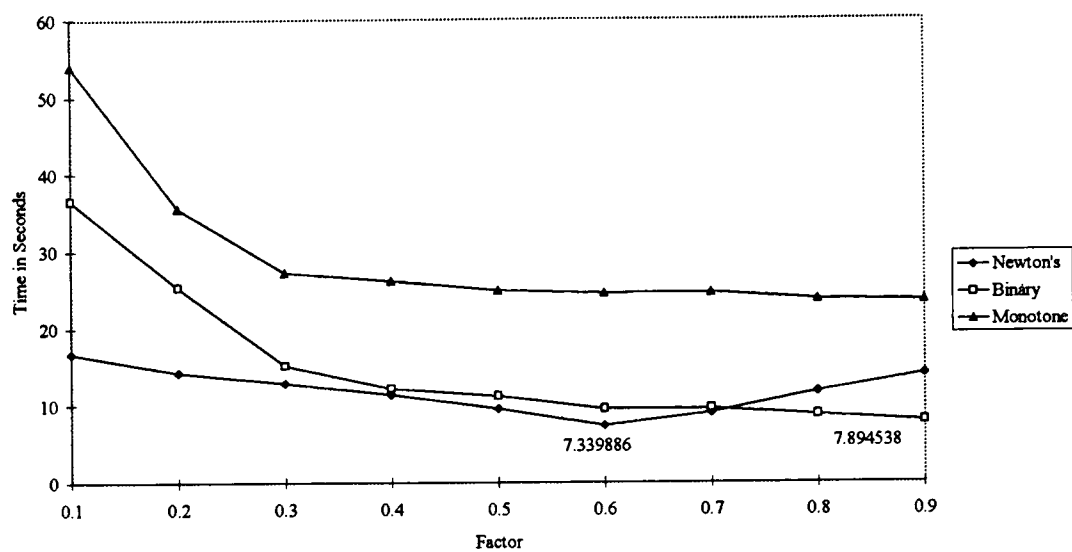


Figure 116 Comparison of Euclidean Binary and Newton's Method Algorithms, Hemisphere

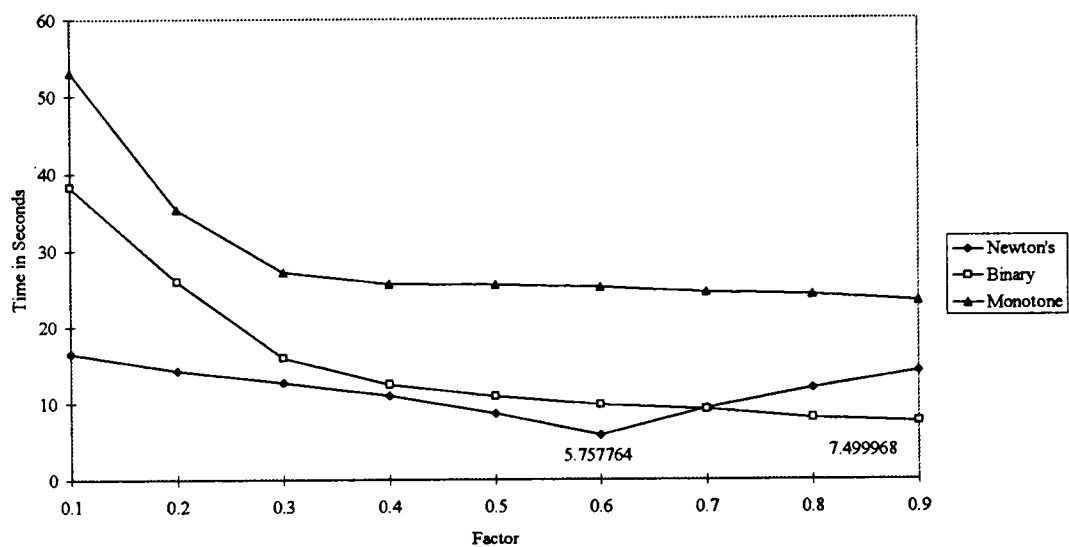


Figure 117 Comparison of Euclidean Binary and Newton's Method Algorithms, Oval

Appendix B Packings

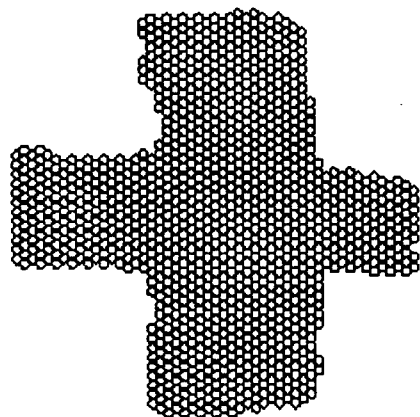


Figure 118 Cross

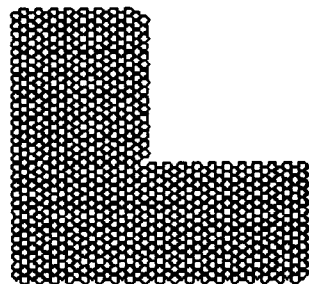


Figure 119 El

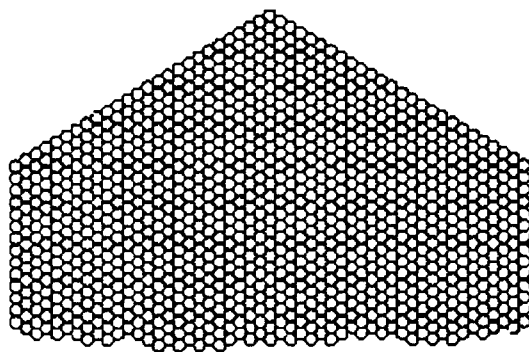


Figure 120 Hemisphere

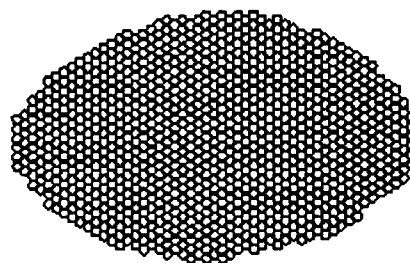


Figure 121 Oval

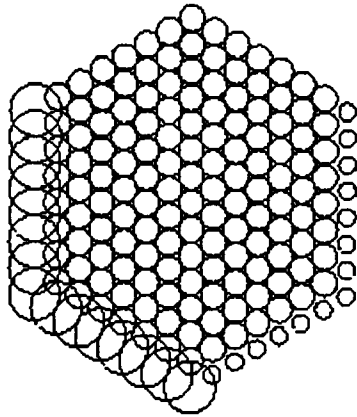


Figure 122 Small hex, Lh

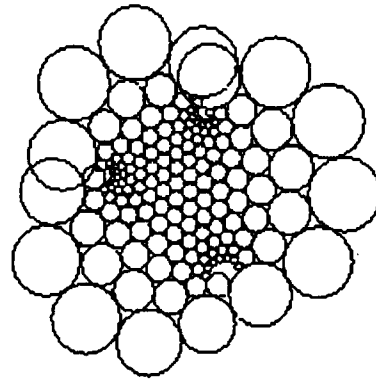


Figure 123 Small Hex, Lh2

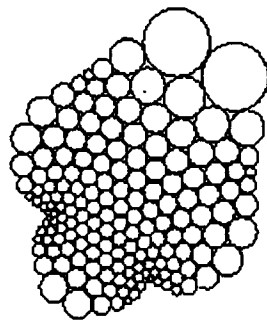


Figure 124 Small Hex, Lh3

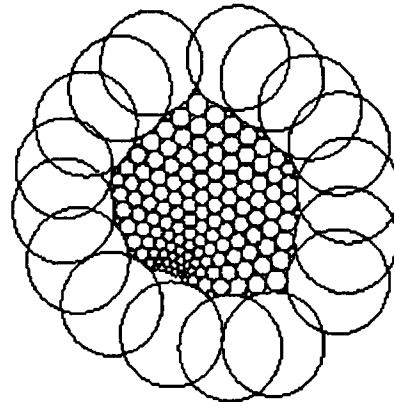


Figure 125 Small Hex, Lh4

Appendix C Algorithm Flowgraphs

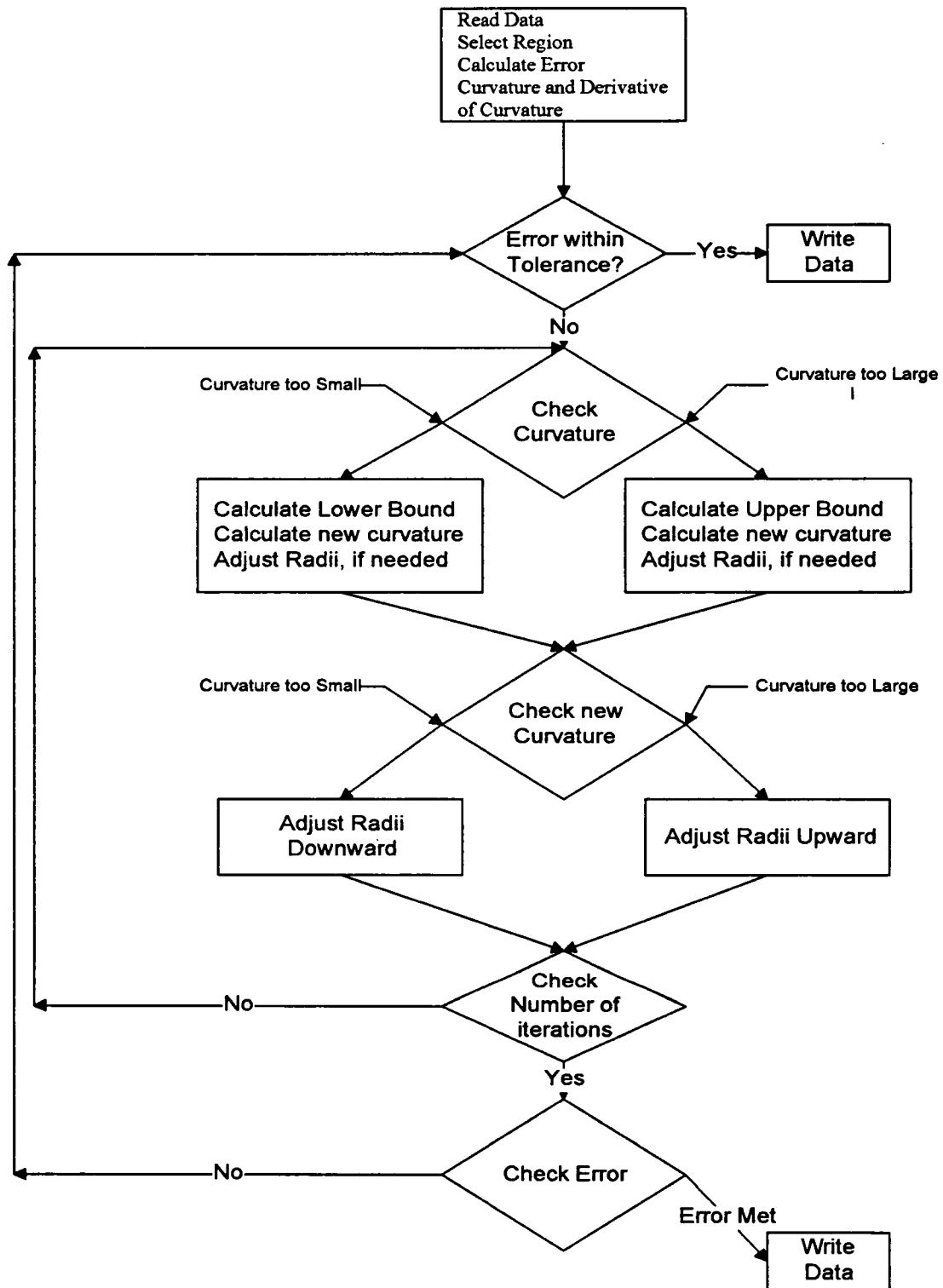


Figure 126 Flowgraph for Euclidean and Hyperbolic Algorithms

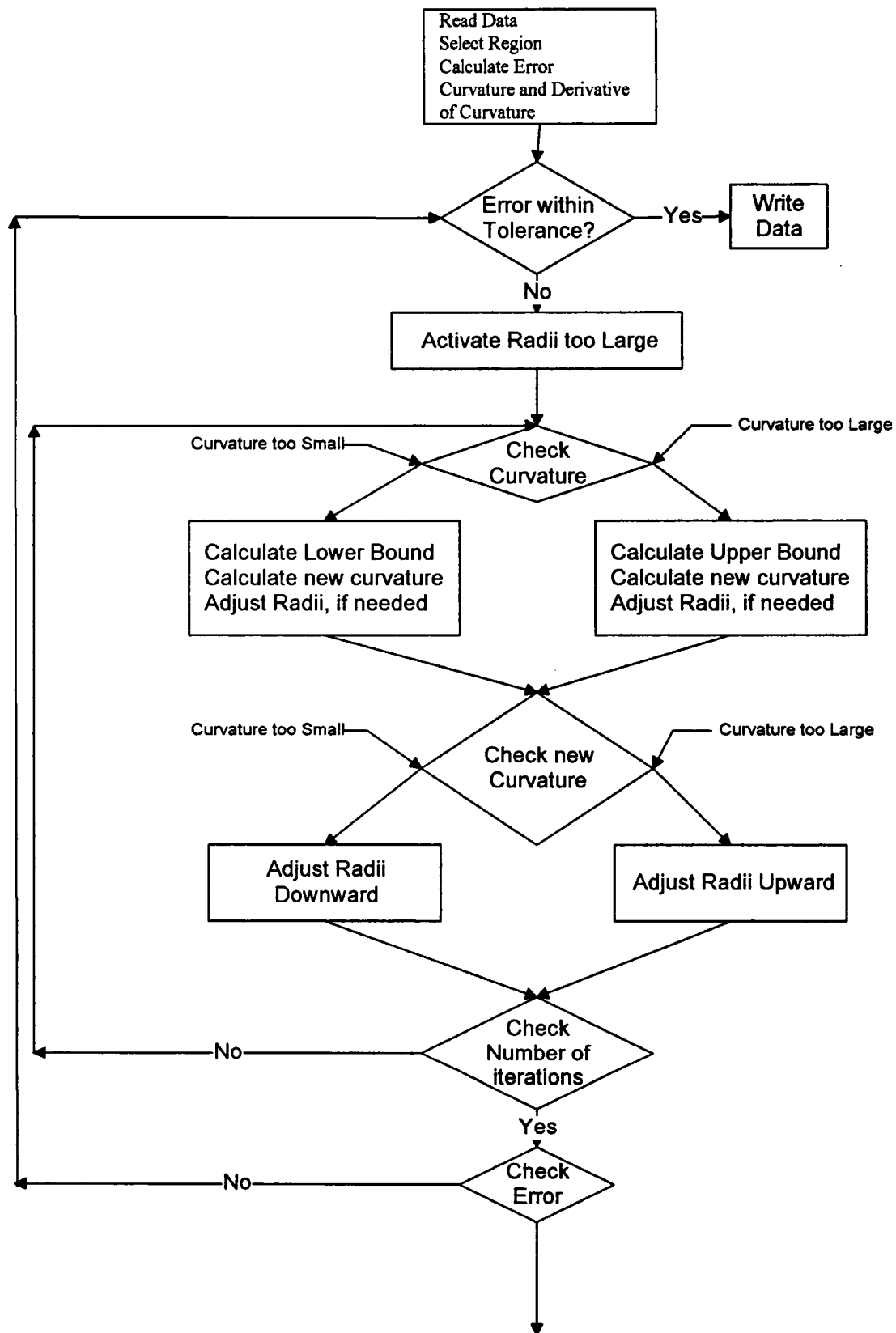


Figure 127 Flowgraph for Euclidean and Hyperbolic Monotone Algorithm (continued on next page)

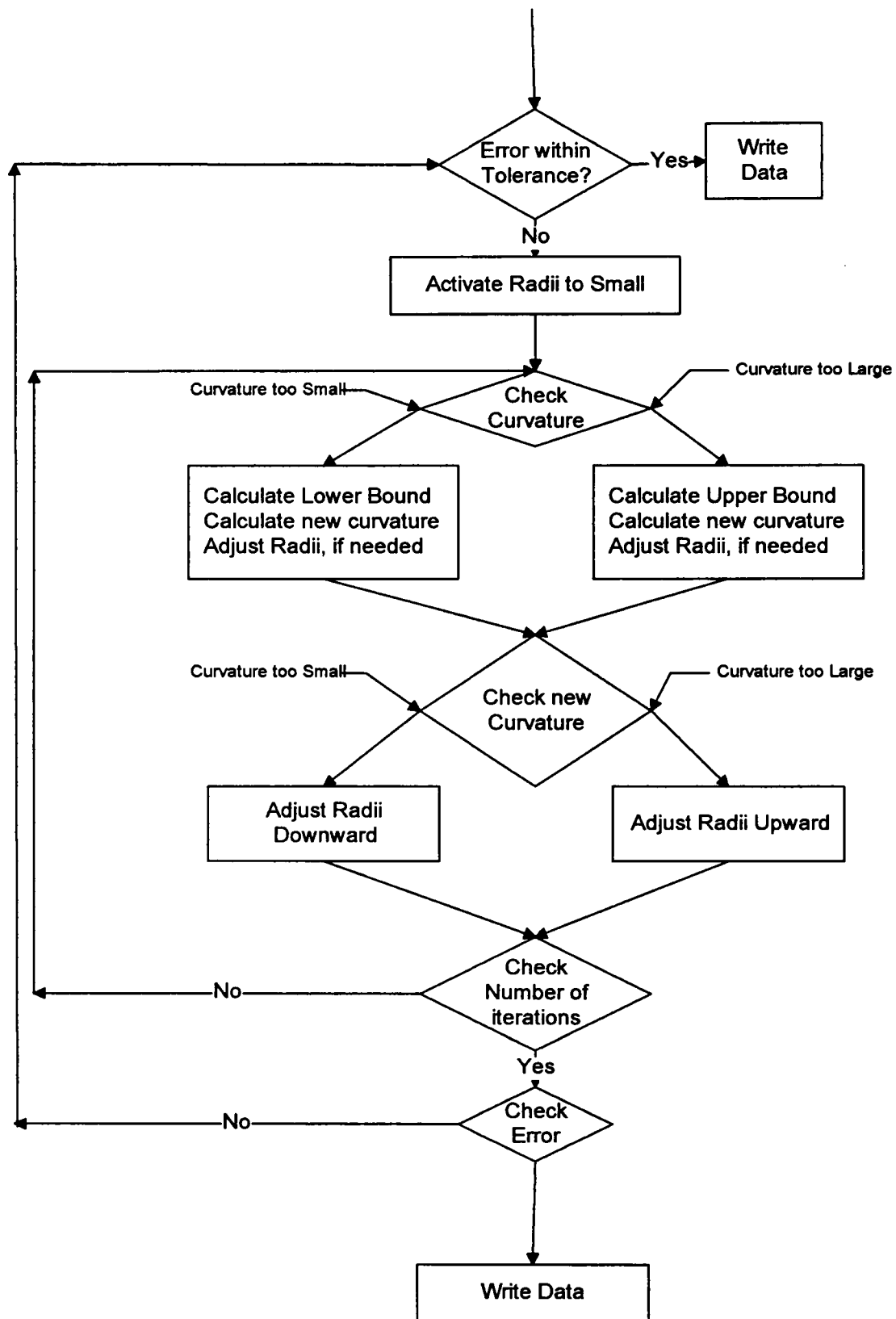


Figure 127 (Continued)

Flowgraph for Euclidean and Hyperbolic Monotone Algorithm

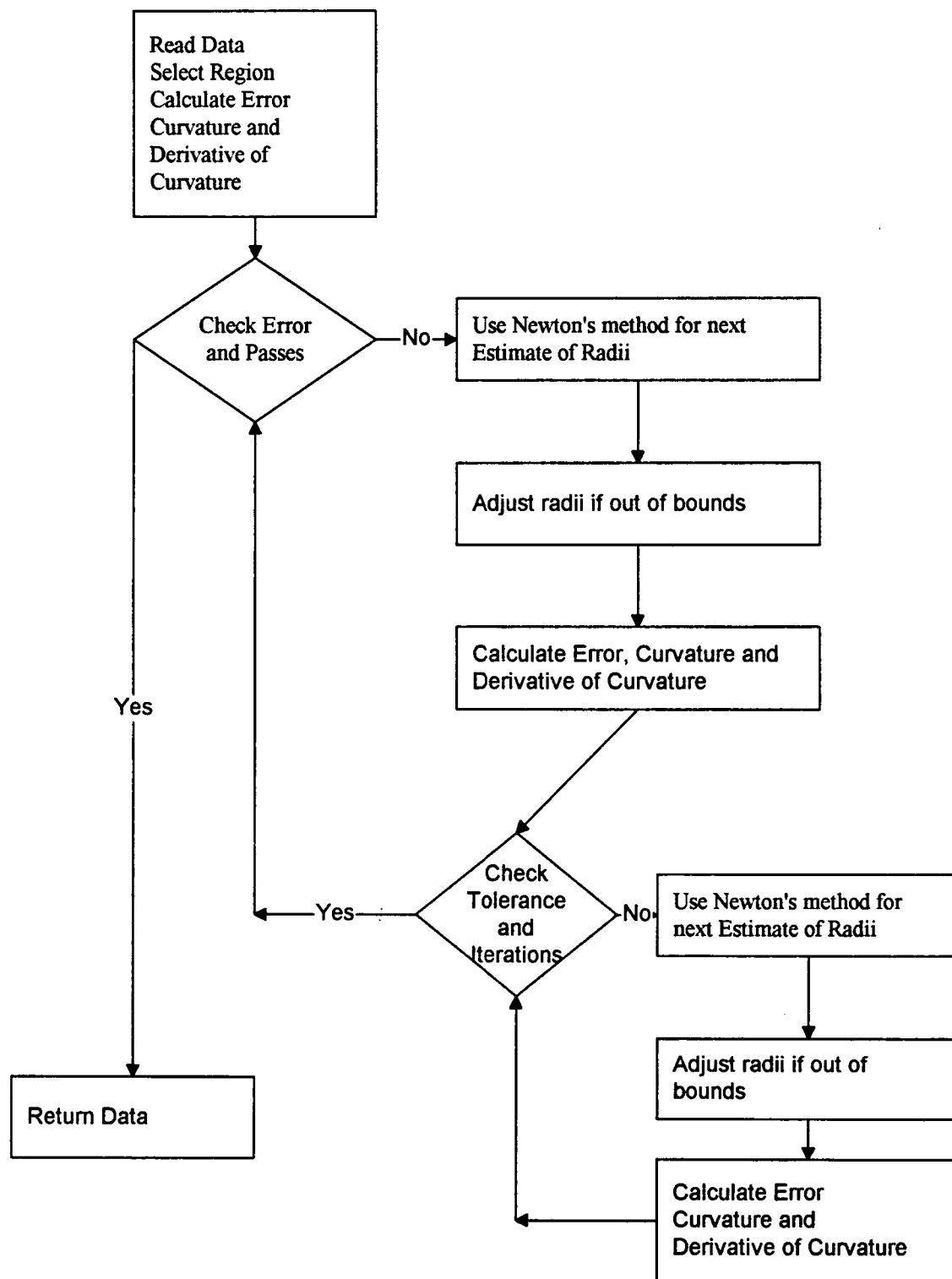


Figure 128 Flowgraph of Newton's Method Algorithms

Appendix D Source Code

Euclidean Binary Algorithm Source Code Ebin.m

```
/*      Last modified on 12/7/94  Fixed error in the Curv calculations

      Last modified on 10/9/94.
      This version of ebin.m only computes radii if the error is less than
      tolerr/6.
*/

#include <mpl.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <mp_time.h>

#define NUM_PACKS 1          /* increase this if you want to work
                             with more than one pack at a time */

#define MAX_GRAPH 2000
#define NUM_PETALS 20
#define MAX_COMPONENTS 20
#define float double

visible void
mpl_sub(fe_rad, fe_aim, fe_toler, fe_factor, fe_inter)
void * fe_rad;
void * fe_aim;
void * fe_toler;
void * fe_factor;
void * fe_inter;
{
plural double a, b, c;

    struct timeval *sta, *en;
    struct timezone *tz;
    int count=0,n,f, passes, interact, pe_inter, dummy;
    plural float err,bestcurv,lower,upper,upcurv,lowcurv,pe_rad,pe_aim;
    plural float curv;
    float factor,maxerr,toler,pe_toler,pe_factor, okerr,chgflag;
    plural act;
    long unsigned int t;
    blockIn(fe_rad, &pe_rad, 0, 0, nxproc, nyproc, sizeof(pe_rad));
    blockIn(fe_aim, &pe_aim, 0, 0, nxproc, nyproc, sizeof(pe_aim));
    copyIn(fe_toler, &pe_toler, sizeof(pe_toler));
    copyIn(fe_factor, &pe_factor, sizeof(pe_factor));
    copyIn(fe_inter, &pe_inter, sizeof(pe_inter));

    gettimeofday(sta, tz);
```

```

passes=50000;
toler=pe_toler;
okerr=toler;
factor=pe_factor;
if (pe_rad>0 && pe_aim >-.5)
{
    curv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        -xnetS[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        +xnetS[1].pe_rad*xnetE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        -xnetW[1].pe_rad*xnetN[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        +xnetW[1].pe_rad*xnetN[1].pe_rad));
}

err=curv-pe_aim;
if (curv<pe_aim) err=pe_aim-curv;
if (pe_rad>0 && pe_aim > -.5)
{
    maxerr=reduceMaxf(err);
    while (maxerr>toler && count<passes)
    {
        err=curv-pe_aim;
        if (curv<pe_aim) err=pe_aim-curv;

        interact=0;
        while (interact<pe_inter)
        {
            interact++;
            bestcurv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
                -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
                (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
                +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
                p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
                -xnetNE[1].pe_rad*xnetE[1].pe_rad)/

```

```

        (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        -xnetS[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        +xnetS[1].pe_rad*xnetE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        -xnetW[1].pe_rad*xnetN[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        +xnetW[1].pe_rad*xnetN[1].pe_rad));

```

```

if (bestcurv>(pe_aim+okerr))

```

```

{
    upper=pe_rad/factor;
    upcurv=
        p_acos((upper*(upper+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
        (upper*(upper+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((upper*(upper+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
        (upper*(upper+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((upper*(upper+xnetS[1].pe_rad+xnetE[1].pe_rad)
        -xnetS[1].pe_rad*xnetE[1].pe_rad)/
        (upper*(upper+xnetS[1].pe_rad+xnetE[1].pe_rad)
        +xnetS[1].pe_rad*xnetE[1].pe_rad))+
        p_acos((upper*(upper+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
        (upper*(upper+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((upper*(upper+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
        (upper*(upper+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((upper*(upper+xnetW[1].pe_rad+xnetN[1].pe_rad)
        -xnetW[1].pe_rad*xnetN[1].pe_rad)/
        (upper*(upper+xnetW[1].pe_rad+xnetN[1].pe_rad)
        +xnetW[1].pe_rad*xnetN[1].pe_rad));

```

```

    if (upcurv>pe_aim) pe_rad=upper;

```

```

}

```

```

else if (bestcurv<(pe_aim-okerr))
{
    lower=pe_rad*factor;
    lowcurv=
        p_acos((lower*(lower+xnetN[1].pe_rad+xnetNE[1].pe_rad)
            -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
            (lower*(lower+xnetN[1].pe_rad+xnetNE[1].pe_rad)
            +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((lower*(lower+xnetNE[1].pe_rad+xnetE[1].pe_rad)
            -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
            (lower*(lower+xnetNE[1].pe_rad+xnetE[1].pe_rad)
            +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((lower*(lower+xnetS[1].pe_rad+xnetE[1].pe_rad)
            -xnetS[1].pe_rad*xnetE[1].pe_rad)/
            (lower*(lower+xnetS[1].pe_rad+xnetE[1].pe_rad)
            +xnetS[1].pe_rad*xnetE[1].pe_rad))+
        p_acos((lower*(lower+xnetS[1].pe_rad+xnetSW[1].pe_rad)
            -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
            (lower*(lower+xnetS[1].pe_rad+xnetSW[1].pe_rad)
            +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((lower*(lower+xnetW[1].pe_rad+xnetSW[1].pe_rad)
            -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
            (lower*(lower+xnetW[1].pe_rad+xnetSW[1].pe_rad)
            +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((lower*(lower+xnetW[1].pe_rad+xnetN[1].pe_rad)
            -xnetW[1].pe_rad*xnetN[1].pe_rad)/
            (lower*(lower+xnetW[1].pe_rad+xnetN[1].pe_rad)
            +xnetW[1].pe_rad*xnetN[1].pe_rad));

    if (lowcurv<pe_aim) pe_rad=lower;
}

if (bestcurv>(pe_aim+okerr))
{
    lower=pe_rad;
    lowcurv=bestcurv;
    pe_rad += (pe_aim-bestcurv)*(upper-pe_rad)/(upcurv-bestcurv);
}
else if (bestcurv<(pe_aim-okerr))
{
    upper=pe_rad;
    upcurv=bestcurv;
    pe_rad -= (bestcurv-pe_aim)*(lower-pe_rad)/(lowcurv-bestcurv);
}
}

/* e_pe_radcalc */
count++;
curv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
    -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
    (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
    +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
    p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
    -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
    (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
    +xnetNE[1].pe_rad*xnetE[1].pe_rad));

```

```

(pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
+xnetE[1].pe_rad*xnetNE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
-xnetS[1].pe_rad*xnetE[1].pe_rad)/
(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
+xnetS[1].pe_rad*xnetE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
-xnetS[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
+xnetS[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
-xnetW[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
+xnetW[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
-xnetW[1].pe_rad*xnetN[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
+xnetW[1].pe_rad*xnetN[1].pe_rad));

err= curv- pe_aim;
if (curv < pe_aim) err = pe_aim - curv;
maxerr=reduceMaxd(err);
} /* end of while */
} /* end of if */
/*      printf("count %d\n\n",count);*/
gettimeofday(en,tz);
printf("sec %f\n",en->tv_sec-sta->tv_sec+(en->tv_usec-sta->tv_usec)*.000001);
printf("count %d\n\n",count);
blockOut(&pe_rad, fe_rad, 0, 0, nxproc, nyproc, sizeof(pe_rad));

/* e_rifle */
}

```

Euclidean Monotone Algorithm Source Code Emono.m

```
/*      Last modified on 12/7/94 Fixed error in the Curv calculations
      Last modified on 11/16/94.
      This version of MPL_repack_mono.m computes all radii whose
      Curvature is larger than the aim. It then computes the
      changes the radii whose curvature is less than the aim. This
      ensures a monotone condition whenever we are changing the
      radii.
*/

#include <mpl.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <mp_time.h>

#define NUM_PACKS 1          /* increase this if you want to work
                             with more than one pack at a time */

#define MAX_GRAPH 2000
#define NUM_PETALS 20
#define MAX_COMPONENTS 20
#define float double

visible void
mpl_sub(fe_rad, fe_aim, fe_toler, fe_factor, fe_inter)
void * fe_rad;
void * fe_aim;
void * fe_toler;
void * fe_factor;
void * fe_inter;
{
plural double a, b, c;

    struct timeval *sta, *en;
    struct timezone *tz;
    int count2=0, count=0, n, passes, interact, pe_inter, dummy;
    plural float err, bestcurv, lower, upper, upcurv, lowcurv, pe_rad, pe_aim;
    plural float curv;
    float factor=0.5, maxerr, toler, pe_toler, pe_factor, okerr, chgflag;
    plural act;
    long unsigned int t;

    blockIn(fe_rad, &pe_rad, 0, 0, nxproc, nyproc, sizeof(pe_rad));
    blockIn(fe_aim, &pe_aim, 0, 0, nxproc, nyproc, sizeof(pe_aim));
    copyIn(fe_toler, &pe_toler, sizeof(pe_toler));
    copyIn(fe_factor, &pe_factor, sizeof(pe_factor));
    copyIn(fe_inter, &pe_inter, sizeof(pe_inter));

    gettimeofday(sta, tz);
```

```

passes=50000;
toler=pe_toler;
okerr=toler;
factor=pe_factor;
act=0;
if (pe_rad>0 && pe_aim >-.5)
{
    curv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        -xnetS[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        +xnetS[1].pe_rad*xnetE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        -xnetW[1].pe_rad*xnetN[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        +xnetW[1].pe_rad*xnetN[1].pe_rad));

}

err=curv-pe_aim;
maxerr=reduceMaxf(err);
if (pe_rad>0 && pe_aim > -.5)
{
    while(maxerr > toler && count2<passes)
    {
        while (maxerr>toler && count<passes)
        {
            curv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
                -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
                (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
                +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
                p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
                -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
                (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
                +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
                p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
                -xnetS[1].pe_rad*xnetE[1].pe_rad)/
                (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
                +xnetS[1].pe_rad*xnetE[1].pe_rad))+
                p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
                -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
                (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
                +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
                p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
                -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
                (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
                +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
                p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
                -xnetW[1].pe_rad*xnetN[1].pe_rad)/
                (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
                +xnetW[1].pe_rad*xnetN[1].pe_rad));

```

```

+xnetS[1].pe_rad*xnetE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
-xnetS[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
+xnetS[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
-xnetW[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
+xnetW[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
-xnetW[1].pe_rad*xnetN[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
+xnetW[1].pe_rad*xnetN[1].pe_rad));

err=curv-pe_aim;
act=0;
if (err > toler) act=1;

if (act)
{
    interact=0;
    while (interact<pe_inter)
    {
        interact++;
        bestcurv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        -xnetS[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        +xnetS[1].pe_rad*xnetE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        -xnetW[1].pe_rad*xnetN[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        +xnetW[1].pe_rad*xnetN[1].pe_rad));

        if (bestcurv>(pe_aim+okerr))
        {
            upper=pe_rad/factor;
            upcurv=

```

```

        p_acos((upper*(upper+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
        (upper*(upper+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((upper*(upper+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
        (upper*(upper+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((upper*(upper+xnetS[1].pe_rad+xnetE[1].pe_rad)
        -xnetS[1].pe_rad*xnetE[1].pe_rad)/
        (upper*(upper+xnetS[1].pe_rad+xnetE[1].pe_rad)
        +xnetS[1].pe_rad*xnetE[1].pe_rad))+
        p_acos((upper*(upper+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
        (upper*(upper+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((upper*(upper+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
        (upper*(upper+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((upper*(upper+xnetW[1].pe_rad+xnetN[1].pe_rad)
        -xnetW[1].pe_rad*xnetN[1].pe_rad)/
        (upper*(upper+xnetW[1].pe_rad+xnetN[1].pe_rad)
        +xnetW[1].pe_rad*xnetN[1].pe_rad));

    if (upcurv>pe_aim) pe_rad=upper;

    lower=pe_rad;
    lowcurv=bestcurv;
    pe_rad += (pe_aim-bestcurv)*(upper-pe_rad)/(upcurv-bestcurv);
}
}
/* e_pe_radcalc */
count++;
err= curv-pe_aim;
act=1;
if (err > toler) act=1;
maxerr=reduceMaxd(err);
} /* end of while */
act=1;
maxerr=1;
while (maxerr>toler && count<passes)
{
    curv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
    -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
    (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
    +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
    p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
    -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
    (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)

```

```

+xnetE[1].pe_rad*xnetNE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
-xnetS[1].pe_rad*xnetE[1].pe_rad)/
(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
+xnetS[1].pe_rad*xnetE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
-xnetS[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
+xnetS[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
-xnetW[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
+xnetW[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
-xnetW[1].pe_rad*xnetN[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
+xnetW[1].pe_rad*xnetN[1].pe_rad));

err=pe_aim-curr;
act=0;
if (err > toler) act=1;

if (act)
{
    interact=0;
    while (interact<pe_inter)
    {
        interact++;
        bestcurv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        -xnetS[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        +xnetS[1].pe_rad*xnetE[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        -xnetW[1].pe_rad*xnetN[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        +xnetW[1].pe_rad*xnetN[1].pe_rad));
    }
}

```

```

if (bestcurv<(pe_aim-okerr))
{
    lower=pe_rad*factor;
    lowcurv=
        p_acos((lower*(lower+xnetN[1].pe_rad+xnetNE[1].pe_rad)
            -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
            (lower*(lower+xnetN[1].pe_rad+xnetNE[1].pe_rad)
            +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((lower*(lower+xnetNE[1].pe_rad+xnetE[1].pe_rad)
            -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
            (lower*(lower+xnetNE[1].pe_rad+xnetE[1].pe_rad)
            +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
        p_acos((lower*(lower+xnetS[1].pe_rad+xnetE[1].pe_rad)
            -xnetS[1].pe_rad*xnetE[1].pe_rad)/
            (lower*(lower+xnetS[1].pe_rad+xnetE[1].pe_rad)
            +xnetS[1].pe_rad*xnetE[1].pe_rad))+
        p_acos((lower*(lower+xnetS[1].pe_rad+xnetSW[1].pe_rad)
            -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
            (lower*(lower+xnetS[1].pe_rad+xnetSW[1].pe_rad)
            +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((lower*(lower+xnetW[1].pe_rad+xnetSW[1].pe_rad)
            -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
            (lower*(lower+xnetW[1].pe_rad+xnetSW[1].pe_rad)
            +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
        p_acos((lower*(lower+xnetW[1].pe_rad+xnetN[1].pe_rad)
            -xnetW[1].pe_rad*xnetN[1].pe_rad)/
            (lower*(lower+xnetW[1].pe_rad+xnetN[1].pe_rad)
            +xnetW[1].pe_rad*xnetN[1].pe_rad));

    if (lowcurv<pe_aim) pe_rad=lower;
    upper=pe_rad;
    upcurv=bestcurv;
    pe_rad += (pe_aim-bestcurv)*(lower-pe_rad)/(lowcurv-bestcurv);
}
}

/* e_pe_radcalc */
count++;
err= pe_aim-curv;
act=1;
if (err > toler) act=1;
maxerr=reduceMaxd(err);
} /* end of while */
curv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
    -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
    (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
    +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
    p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
    -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
    (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
    +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
    p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
    -xnetS[1].pe_rad*xnetE[1].pe_rad)/

```

```

(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
+xnetS[1].pe_rad*xnetE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
-xnetS[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
+xnetS[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
-xnetW[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
+xnetW[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
-xnetW[1].pe_rad*xnetN[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
+xnetW[1].pe_rad*xnetN[1].pe_rad));

err= pe_aim-curv;
if (curv>pe_aim)
{
    err=curv-pe_aim;
}
maxerr=reduceMaxd(err);
} /* end of if */
} /* end of if */
printf("count %d\n\n",count);
gettimeofday(en,tz);
printf("sec %f\n",en->tv_sec-sta->tv_sec+(en->tv_usec-sta->tv_usec)*.000001);
printf("count %d\n\n",count);
blockOut(&pe_rad, fe_rad, 0, 0, nxproc, nyproc, sizeof(pe_rad));

/* e_rifle */
}

```

Euclidean Newton's Method Source Code Ebinewt.m

```
/*      Last modified on 12/7/94  Fixed error in the Curv calculations

      Last modified on 10/9/94.
      This version of MPI_E_Repack.m only computes radii if the error is less than
      tolerr/6.

*/

#include <mpl.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <mp_time.h>

#define NUM_PACKS 1          /* increase this if you want to work
                             with more than one pack at a time */

#define MAX_GRAPH 2000
#define NUM_PETALS 20
#define MAX_COMPONENTS 20
#define float double

visible void
mpl_sub(fe_rad, fe_aim, fe_toler, fe_factor, fe_inter)
void * fe_rad;
void * fe_aim;
void * fe_toler;
void * fe_factor;
void * fe_inter;
{
plural double a, b, c;

    struct timeval *sta, *en;
    struct timezone *tz;
    int count=0,n,f, passes, interact, pe_inter, dummy;
    plural float err,bestcurv,lower,upper,upcurv,lowcurv,pe_rad,pe_aim;
    plural float curv,bestrad,curvderiv,tryrad,trycurv;
    float factor,maxerr,toler,pe_toler,pe_factor, okerr,chgflag;
    plural act;
    long unsigned int t;
    extern plural float e_cosderv_mpl();

    blockIn(fe_rad, &pe_rad, 0, 0, nxproc, nyproc, sizeof(pe_rad));
    blockIn(fe_aim, &pe_aim, 0, 0, nxproc, nyproc, sizeof(pe_aim));
    copyIn(fe_toler, &pe_toler, sizeof(pe_toler));
    copyIn(fe_factor, &pe_factor, sizeof(pe_factor));
    copyIn(fe_inter, &pe_inter, sizeof(pe_inter));

    gettimeofday(sta, tz);
    passes=5000;
    toler=pe_toler;
```

```

okerr=toler;
factor=pe_factor;
if (pe_rad>0 && pe_aim >-.5)
{
bestcurv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
-xnetN[1].pe_rad*xnetNE[1].pe_rad)/
(pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
+xnetN[1].pe_rad*xnetNE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
-xnetNE[1].pe_rad*xnetE[1].pe_rad)/
(pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
+xnetE[1].pe_rad*xnetNE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
-xnetS[1].pe_rad*xnetE[1].pe_rad)/
(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
+xnetS[1].pe_rad*xnetE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
-xnetS[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
+xnetS[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
-xnetW[1].pe_rad*xnetSW[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
+xnetW[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
-xnetW[1].pe_rad*xnetN[1].pe_rad)/
(pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
+xnetW[1].pe_rad*xnetN[1].pe_rad));

curvderiv=e_cosderv_mpl(pe_rad,xnetN[1].pe_rad,xnetNE[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetE[1].pe_rad,xnetNE[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetE[1].pe_rad,xnetS[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetS[1].pe_rad,xnetSW[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetW[1].pe_rad,xnetSW[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetW[1].pe_rad,xnetN[1].pe_rad);
bestrad=pe_rad;
}

err=curv-pe_aim;
if (curv<pe_aim) err=pe_aim-curv;
if (pe_rad>0 && pe_aim > -.5)
{
maxerr=reduceMaxf(err);
while (maxerr>toler && count<passes)
{
count++;
tryrad=bestrad-(bestcurv-pe_aim)/curvderiv;
if (tryrad<bestrad*factor) tryrad=bestrad*factor;
if (tryrad>(1+bestrad)*factor) tryrad=(1+bestrad)*factor;
trycurv=p_acos((tryrad*(tryrad+xnetN[1].tryrad+xnetNE[1].tryrad)
-xnetN[1].tryrad*xnetNE[1].tryrad)/
(tryrad*(tryrad+xnetN[1].tryrad+xnetNE[1].tryrad)
+xnetN[1].tryrad*xnetNE[1].tryrad))+

```

```

p_acos((tryrad*(tryrad+xnetNE[1].tryrad+xnetE[1].tryrad)
-xnetNE[1].tryrad*xnetE[1].tryrad)/
(tryrad*(tryrad+xnetNE[1].tryrad+xnetE[1].tryrad)
+xnetE[1].tryrad*xnetNE[1].tryrad))+
p_acos((tryrad*(tryrad+xnetS[1].tryrad+xnetE[1].tryrad)
-xnetS[1].tryrad*xnetE[1].tryrad)/
(tryrad*(tryrad+xnetS[1].tryrad+xnetE[1].tryrad)
+xnetS[1].tryrad*xnetE[1].tryrad))+
p_acos((tryrad*(tryrad+xnetS[1].tryrad+xnetSW[1].tryrad)
-xnetS[1].tryrad*xnetSW[1].tryrad)/
(tryrad*(tryrad+xnetS[1].tryrad+xnetSW[1].tryrad)
+xnetS[1].tryrad*xnetSW[1].tryrad))+
p_acos((tryrad*(tryrad+xnetW[1].tryrad+xnetSW[1].tryrad)
-xnetW[1].tryrad*xnetSW[1].tryrad)/
(tryrad*(tryrad+xnetW[1].tryrad+xnetSW[1].tryrad)
+xnetW[1].tryrad*xnetSW[1].tryrad))+
p_acos((tryrad*(tryrad+xnetW[1].tryrad+xnetN[1].tryrad)
-xnetW[1].tryrad*xnetN[1].tryrad)/
(tryrad*(tryrad+xnetW[1].tryrad+xnetN[1].tryrad)
+xnetW[1].tryrad*xnetN[1].tryrad));

curvderiv=e_cosderv_mpl(tryrad,xnetN[1].tryrad,xnetNE[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetE[1].tryrad,xnetNE[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetE[1].tryrad,xnetS[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetS[1].tryrad,xnetSW[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetW[1].tryrad,xnetSW[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetW[1].tryrad,xnetN[1].tryrad);

err=trycurv-pe_aim;
if (err<0) err=(-err);
interact=0;
maxerr=reduceMaxf(err);

/*
while ( ((trycurv-pe_aim)>toler || (trycurv-pe_aim)<(-toler))
&& (interact<pe_inter))
{
    interact++;
    tryrad=(tryrad+bestrad)*factor;
    bestcurv=p_acos((tryrad*(pe_rad+xnetN[1].tryrad+xnetNE[1].tryrad)
-xnetN[1].tryrad*xnetNE[1].tryrad)/
(tryrad*(tryrad+xnetN[1].tryrad+xnetNE[1].tryrad)
+xnetN[1].tryrad*xnetNE[1].tryrad))+
p_acos((tryrad*(tryrad+xnetNE[1].tryrad+xnetE[1].tryrad)
-xnetNE[1].tryrad*xnetE[1].tryrad)/
(tryrad*(tryrad+xnetNE[1].tryrad+xnetE[1].tryrad)
+xnetE[1].tryrad*xnetNE[1].tryrad))+
p_acos((tryrad*(tryrad+xnetS[1].tryrad+xnetE[1].tryrad)
-xnetS[1].tryrad*xnetE[1].tryrad)/
(tryrad*(tryrad+xnetS[1].tryrad+xnetE[1].tryrad)
+xnetS[1].tryrad*xnetE[1].tryrad))+
p_acos((tryrad*(tryrad+xnetS[1].tryrad+xnetSW[1].tryrad)
-xnetS[1].tryrad*xnetSW[1].tryrad)/
(tryrad*(tryrad+xnetS[1].tryrad+xnetSW[1].tryrad)

```

```

        +xnetS[1].tryrad*xnetSW[1].tryrad))+
p_acos((tryrad*(tryrad+xnetW[1].tryrad+xnetSW[1].tryrad)
        -xnetW[1].tryrad*xnetSW[1].tryrad)/
        (tryrad*(tryrad+xnetW[1].tryrad+xnetSW[1].tryrad)
        +xnetW[1].tryrad*xnetSW[1].tryrad))+
p_acos((tryrad*(tryrad+xnetW[1].tryrad+xnetN[1].tryrad)
        -xnetW[1].tryrad*xnetN[1].tryrad)/
        (tryrad*(tryrad+xnetW[1].tryrad+xnetN[1].tryrad)
        +xnetW[1].tryrad*xnetN[1].tryrad));

curvderiv=e_cosderv_mpl(tryrad,xnetN[1].tryrad,xnetNE[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetE[1].tryrad,xnetNE[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetE[1].tryrad,xnetS[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetS[1].tryrad,xnetSW[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetW[1].tryrad,xnetSW[1].tryrad)
+ e_cosderv_mpl(tryrad,xnetW[1].tryrad,xnetN[1].tryrad);
}

*/

interact++;
bestrad=tryrad;
pe_rad=bestrad;

bestcurv=p_acos((pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        -xnetN[1].pe_rad*xnetNE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetN[1].pe_rad+xnetNE[1].pe_rad)
        +xnetN[1].pe_rad*xnetNE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        -xnetNE[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetNE[1].pe_rad+xnetE[1].pe_rad)
        +xnetE[1].pe_rad*xnetNE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        -xnetS[1].pe_rad*xnetE[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetE[1].pe_rad)
        +xnetS[1].pe_rad*xnetE[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        -xnetS[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetS[1].pe_rad+xnetSW[1].pe_rad)
        +xnetS[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        -xnetW[1].pe_rad*xnetSW[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetSW[1].pe_rad)
        +xnetW[1].pe_rad*xnetSW[1].pe_rad))+
p_acos((pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        -xnetW[1].pe_rad*xnetN[1].pe_rad)/
        (pe_rad*(pe_rad+xnetW[1].pe_rad+xnetN[1].pe_rad)
        +xnetW[1].pe_rad*xnetN[1].pe_rad));

curvderiv=e_cosderv_mpl(pe_rad,xnetN[1].pe_rad,xnetNE[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetE[1].pe_rad,xnetNE[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetE[1].pe_rad,xnetS[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetS[1].pe_rad,xnetSW[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetW[1].pe_rad,xnetSW[1].pe_rad)
+ e_cosderv_mpl(pe_rad,xnetW[1].pe_rad,xnetN[1].pe_rad);

```

```

        err=bestcurv-pe_aim;
        if (err<0) err=(-err);
        pe_rad=bestrad;

        maxerr=reduceMaxf(err);
    } /* end of while */
} /* end of if */
if(n>passes-1) printf("sec 0 maxerr= %f\n",maxerr);
gettimeofday(en,tz);
printf("sec %f\n",en->tv_sec-sta->tv_sec+(en->tv_usec-sta->tv_usec)*.000001);
printf(" passes %d",passes);
printf("count %d\n\n",count);
blockOut(&pe_rad, fe_rad, 0, 0, nxproc, nyproc, sizeof(pe_rad));

}

plural float
e_cosderv_mpl(r1,r2,r3)
plural float r1,r2,r3;
{
    plural float ans,sqroot,a;
    ans=1;
/*    if (r1<=0) ans=0;
    if (ans>0)
    {
*/
        a=(r1*r1+r1*r2+r1*r3-r2*r3)/(r1*r1+r1*r2+r1*r3+r2*r3);
        sqroot=p_sqrt(1-(a*a));
        if (sqroot<.0000001) sqroot=.00000001;
        ans=(1/sqroot)*((2*r1+r2+r3)*(2*r2*r3))/
            ((r1+r2)*(r1+r3)*(r1+r2)*(r1+r3));

        return (-ans);
    }
}

```

PVM MasPar Front End Source Code

- ```
/* mppackfe.c Version 1.0, 11/15/95
```
- \* August 18 95 acts as a data transporter from a PVM front end to the maspar. Currently the color functions have been removed, but will be added with the next version.
  - \* 30 June 95 returns value for the colors of each radius. Program given a new name, mpcolor.c.
  - \* Modified on 9 Feb 95 to include additional paramaters. Currently 5 are required, read pack, write pack, factor, tolerance, iterations  
Factor is the amount to divide tolerance by for calculations.  
Tolerance is the tolerace for computations.  
Iterations is the number of times the radius is changed before a new curvature is recalculated.
  - \* Modified on 4 Feb 95 to include timing mechanisms using time.h
  - \* This is a shell program for processes to be forked from CirclePack.
  - \* It manages communication and read/write of packing data from the parent program. If you want manipulate data in some certain way, you can modify this shell, store it under some "program\_name.c", and call it from CirclePack. Here is the compiler command:

```
mpl_cc -o color mpcolor.c htesta_color.m -g -lm
```

```
gcc program_name.c -o program_name -g -lm
```

- \* This program is called by mpcp.

```
=====
===== */

/* *****
*
*
===== includes and defines =====
*
*
***** */
/*#include "cp_remote.h"*/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>
#include "pvm3.h"

#define NUM_PACKS 1 /* increase this if you want to work
 with more than one pack at a time */

#define MAX_SIZE 1000000
#define MAX_GRAPH 4000
```

```

#define TOLER .000001
#define OKERR .00000001
#define NUM_PETALS 20
#define MAX_COMPONENTS 20
#define float double

#define DATAMSG 10
#define REPLYMSG 11
#define NN 4096

/* *****
*
*
===== global variable/type declares =====
*
* Note: You will need to know how data is stored in
* order to write your routines. These are the main data
* structures.
***** */
FILE *output;
/*struct K_data packK[NUM_PACKS][MAX_SIZE+1];
struct R_data packR[NUM_PACKS][MAX_SIZE+1];
struct p_data *packdata;
*/
extern struct Pathlist *pathlist; /* storage for paths */
extern int pathlength;

char buf[1024], msgbuf[1024];

struct
{
 int v;
 float rad;
 float curv;
 float aim;
} mp2[64][64];

struct
{
 int row;
 int col;
} mp2list[MAX_GRAPH+1];
float a_rad[64][64], a_aim[64][64], a_col[64][64], a_best[64][64], a_orig[64][64];

static char buf[1024], msgbuf[1024]; /* utility buffers for strings. */
float toler=.000001;
float okerr=.000001;
extern mpl_sub(); /* the subroutine doing the job on each PE */

/* *****
*
*

```

```

===== main program =====
*
*
***** */

main()

{
 int mytid;
 int tids[32];
 int nproc, numt;

 int i,j,p,q,l,m,inter;
 float toler,factor;
 time_t start, end;
 long unsigned int t;
j=8;
p=8;

output = fopen("mp.log","w");

/* ----- manipulate data phase ----- */

/* ???
??
??
??
 This is section where you manipulate or call routines
 to manipulate the data.
??
??
??
?? */
 fputs("Enrolling in PVM \n", output);
 /* enroll in pvm */
 mytid = pvm_mytid();
 nproc = 1;
 gethostname(buf + strlen(buf), 64);
 fprintf(output,"task mytid = %d on %s\n",mytid,buf);

 fprintf(output,"at Front End\n");

 start=time(NULL);
 fprintf(output,"Sending data \n");
 /* Broadcast initial data to slave tasks */

 /* recieve the data via PVM from another machine */
 pvm_rcv(-1, DATAMSG);
 pvm_upkdouble(a_rad, NN, 1);

```

```

pvm_upkdouble(a_aim, NN, 1);
pvm_upkdouble(&toler, 1, 1);
pvm_upkdouble(&factor, 1, 1);
pvm_upkint(&inter, 1, 1);
callRequest(mpl_sub,32, a_rad, a_aim, &toler, &factor, &inter, a_col);
/* send the results back via pvm */
pvm_initsend(PvmDataDefault);
pvm_pkdouble(a_rad, NN, 1);
pvm_send(pvm_parent(), REPLYMSG);

```

```

end=time(NULL);
fprintf(output,"time %f\n\n",diffime(end,start));

```

```

} /* end of main */

```

```

msg(msg) /* sends error message to parent. */
char *msg;
{
 int tl;

 if ((tl=strlen(msg)-1)<0) return;
 if (msg[tl]!='\n') printf("msg: %s\n",msg);
 else printf("msg: %s",msg);
 fflush(stdout);
}

```

## PVM SUN to MasPar Source Code

```

/* mppack.c Version 1.0, 11/15/95
 * 30 June 95 returns value for the colors of each radius. Program
 * given a new name, mpcolor.c.

 * Modified on 9 Feb 95 to include additional paramaters. Currently
 * 5 are required, read pack, write pack, factor, tolerance, iterations
 * Factor is the amount to divide tolerance by for calculations.
 * Tolerance is the tolerace for computations.
 * Iterations is the number of times the radius is changed
 * before a new curvature is recalculated.

 * Modified on 4 Feb 95 to include timing mechanisms using time.h

 * This is a shell program for processes to be forked from CirclePack.
 * It manages communication and read/write of packing data from the
 * parent program. If you want manipulate data in some certain way,
 * you can modify this shell, store it under some "program_name.c",
 * and call it from CirclePack. Here is the compiler command:

```

```

mpl_cc -o color mpcolor.c htesta_color.m -g -lm

```

```

gcc program_name.c -o program_name -g -lm

```

To use this program from within CirclePack type

```

exec mpcp p q factor tolerance iterations

```

where p and q are pack numbers, factor and tolerance are (0,1) and iterations is an integer.

```

=====
===== */

/* *****
 *
 *
===== includes and defines =====
 *
 *
***** */
#include "cp_remote.h"
/*
#include <stdio.h>
#include <string.h>
#include <math.h>
*/
#include <time.h>
#include "pvm3.h"

#define NUM_PACKS 1 /* increase this if you want to work

```

```

 #define TOLER .000001 /* threshold for repack
computations */
#define OKERR .00000001 /* Roundoff for zero */

#define MAX_SIZE 100000 /* largest packings allowed, effects
 memory usage. */

#define NAME_MAX 255
/*#define MAX_GRAPH 4000*/
#define NUM_PETALS 20
#define MAX_COMPONENTS 20
#define float double

#define DATAMSG 10
#define REPLYMSG 11
#define NN 4096

struct
{
 int v;
 float rad;
 float curv;
 float aim;
} mp2[64][64];

struct K_data packK[NUM_PACKS][MAX_SIZE+1];
struct R_data packR[NUM_PACKS][MAX_SIZE+1];
struct p_data *packdata;

extern struct Pathlist *pathlist; /* storage for paths */
extern int pathlength;

char prog_name[1024];
char buf[1024],msgbuf[1024]; /* utility buffers for strings. */
float toler=.000001,okerr=.0000001;

struct
{
 int row;
 int col;
} mp2list[MAX_GRAPH+1];
float a_rad[64][64], a_aim[64][64], a_col[64][64];

/*static char buf[1024],msgbuf[1024]; /* utility buffers for strings. */

/* *****
*
*
===== main program =====
*
*

```

```

***** */

main(argc,argv)
 int argc;
 char *argv[];
{
 int mytid;
 int tids[32];
 int nproc, numt;

 double a_rad[64][64], a_aim[64][64], a_col[64][64];

 int i,j,p,q,l,m,inter;
 float toler,factor;
 time_t start, end;
 long unsigned int t;

 /* ----- initialization ----- */
 sprintf(msgbuf,"starting cp_pvm");
 msg(msgbuf);
 if ((packdata=(struct p_data *)
 malloc(NUM_PACKS*sizeof(struct p_data)))==NULL)
 {
 sprintf(msgbuf,"Routine exited: insufficient memory.");
 emsg(msgbuf);
 exit(0);
 }
 for (i=0;i<NUM_PACKS;i++)
 {
 packdata[i].packK_ptr=packK[i];
 packdata[i].packR_ptr=packR[i];
 packdata[i].overlap_ptr=NULL;
 packdata[i].Andr_ptr=NULL;
 }

 /* ----- interpret arguments ----- */
 /* if (argc < 2)*/ /* not enough arguments */
 /* {
 sprintf(msgbuf,"External routine exited; call malformed.");
 emsg(msgbuf);
 exit(0);
 }
 */
 msg("setting up varialbes");
 inter=6;
 factor=.5;
 toler=.0001;
 q=1;
 p=0;
 msg("variables setup");
 if (argc>5){
 p=atoi(argv[1]);
 /* pack no. of input data from parent */

```

```

q=atoi(argv[2]); /* pack no. when data returns to parent */
factor=atof(argv[3]);
toler=atof(argv[4]);
inter=atoi(argv[5]);
}
if (inter < 1) inter=1;
if (factor > .99) factor=.5;
if (factor < 0) factor=.5;
if (toler <=0) toler=.00001;

/* ----- locking phase ----- */

/* set up any necessary "lock" of packs in parent. E.g. if
you are going to modify pack q, you send a signal
so that the parent can't do anything to q until you're
done with it. */
/* msg("setting lock\n");
if (!lock_set(q))
{
 sprintf(msgbuf,"remote_shell: lock failed.");
 emsg(msgbuf);
}
*/
/* ----- read data phase ----- */
msg("reading pack\n");
if (!pack_read(&packdata[0].p,2)
 || !readpack(&packdata[0].p,MAX_SIZE+1))
/* 3 indicates amount of data */
{
 sprintf(msgbuf,"remote_shell: read failed.");
 emsg(msgbuf);
}
if (packdata[0].nodecount<=0)
 msg("no nodes\n");
/*sprintf(msgbuf, "hes=%d\n", packdata[0].hes);
msg(msgbuf);
*/
pathlength=0; /* storage needed if program need paths */
pathlist=NULL;
/* if (!path_read())
 {
 sprintf(msgbuf,"remote_shell: no path was received.");
 emsg(msgbuf);
 }
*/
/* ----- manipulate data phase ----- */

/* ???
??
??
??

```

```

This is section where you manipulate or call routines
to manipulate the data.
??
??
??
?? */
for (i=0;i<64;i++)
 for (j=0;j<64;j++)
 a_rad[i][j]=0;

complex_count(&packdata[0]);
facedraworder(&packdata[0],"temp",&a_rad[0][0],&a_aim[0][0]);

msg("Enrolling in PVM \n");
/* enroll in pvm */
mytid = pvm_mytid();
nproc = 1;
/* start up slave task */
if (packdata[0].hes<0) sprintf(prog_name,"hmppack");

else if (packdata[0].hes>0) return;
else sprintf(prog_name,"emppack");
msg(prog_name);

numt=pvm_spawn(prog_name, (char**)0, PvmTaskHost,"pulsar.cs.utk.edu",nproc,tids);

sprintf(msgbuf, "hes=%d\n", packdata[0].hes);
msg(msgbuf);

if(numt < nproc){
 msg("Trouble spawning slaves. Aborting. Error codes are:\n");
 for(i=numt ; i<nproc ; i++) {
 msg("TID %d %d\n",i,tids[i]);
 }
 for(i=0 ; i<numt ; i++){
 pvm_kill(tids[i]);
 }
 pvm_exit();
 exit();
}
if (packdata[0].nodecount<=0)
 msg("no nodes\n");

start=time(NULL);
msg("Sending data \n");
/* Broadcast initial data to slave tasks */
pvm_initsend(PvmDataDefault);
pvm_pkdouble(a_rad, NN, 1);
pvm_pkdouble(a_aim, NN, 1);
pvm_pkdouble(&toler, 1, 1);
pvm_pkdouble(&factor, 1, 1);

```

```

pvm_pkint(&inter, 1, 1);

pvm_send(tids[0], DATAMSG);

printf("Recieving results \n");
/* Wait for results from slaves */
pvm_recv(tids[0], REPLYMSG);
pvm_upkdouble(a_rad, NN, 1);

end=time(NULL);
/*printf("time %f\n\n",difftime(end,start));*/
printf("data received\n");
for (i=1; i<packdata[0].nodecount;i++)
{
 packR[0][i].rad=a_rad[mp2list[i].row][mp2list[i].col];

 packK[0][i].color=a_col[mp2list[i].row][mp2list[i].col];
}

/* ----- write data phase ----- */
if (packdata[0].nodecount<=0)
 msg("no nodes\n");
if (!pack_write(&packdata[0],q,2))
 /* 2 indicates amount of data */
 {
 sprintf(msgbuf,"remote_shell: write failed.");
 emsg(msgbuf);
 }
/* if (!path_write())
 {
 sprintf(msgbuf,"remote_shell: write of path failed.");
 emsg(msgbuf);
 }
 */

/* test messages */

sprintf(msgbuf,"Test message.");
msg(msgbuf);

/* sprintf(msgbuf,"act1");
 cmsg(msgbuf);*/
sprintf(msgbuf,"act %d",q);
cmsg(msgbuf);

sprintf(msgbuf,"disp -w -c");
cmsg(msgbuf);
cmsg("fix\n");
sprintf(msgbuf,"disp -w -c");
cmsg(msgbuf);

/* ----- unlock phase ----- */

```

```

if (!unlock(q))
{
 sprintf(msgbuf,"remote_shell: unlock failed.");
 emsg(msgbuf);
} else msg("unlock succeeded\n");

/* ----- signal readiness to exit ----- */

if (done_signal())
{
 msg("remote_shell: exited successfully.\n");
}
else msg("remote_shell: bailed out.\n");
/* ----- alternately, watch for message from CirclePack --- */

if (scanf("%s",buf) && strcmp(buf,"die")==0)
{
 msg("remote_shell: died, as directed.\n");
}
msg("end\n");
exit(7);

} /* end of main */

/* *****
*
*
===== input subroutines =====
*
* Note: all communication with the parent is via stdin/stdout
***** */

/* *****
*
*
===== output subroutines =====
*
*
***** */

int
writepackold2(p_in,q_out,act) /* write pack p data to pack q_out in parent.
 Return 1 if okay. "act" tells how much data: > 99 means only partial.
*/
/*struct p_data *p;*/
int p_in,q_out,act;

```

```

{
int i,count,n,j,flag=0,colorflag;
float dum;
char *name;
struct p_data *p;
struct K_data *pK_ptr;
struct R_data *pR_ptr;
extern float radius();
p= &packdata[p_in];
pK_ptr=p->packK_ptr;
pR_ptr=p->packR_ptr;
count=p->nodecount;
if (act < 100) /* usual pack data.
 act: 0==> combinatorics, 1==> minimal, 2==> standard,
 3==> complete */
{
fprintf(stdout,"NODECOUNT: %d\n\n",count);
if (act>0)
{
fprintf(stdout,"ALPHA/BETA/GAMMA: %d %d %d\n\n",
 p->alpha,p->beta,p->gamma);
if (p->hes==0) sprintf(buf,"euclidean\n");
else sprintf(buf,"hyperbolic\n");
fprintf(stdout,"GEOMETRY: %s\n",buf);
}
fflush(stdout);
fprintf(stdout,"FLOWERS: \n");
for (n=1;n<=count;n++)
{
fprintf(stdout,"\n%d %d ",n,pK_ptr[n].num);
for (i=0;i<=pK_ptr[n].num;i++)
fprintf(stdout,"%d ",pK_ptr[n].flower[i]);
}
fprintf(stdout,"\n\n");
fflush(stdout);
if (act>0)
{
if (p->overlap_ptr!=NULL)
{
fprintf(stdout,"OVERLAPS:\n");
for (i=1;i<=count;i++)
{
fprintf(stdout,"\n%d ",i);
for (j=0;j<=pK_ptr[i].num;j++)
{
fprintf(stdout," %.6e ",
 (p->overlap_ptr+i(NUM_PETALS+1)+j));
if ((j % 5)==4) fprintf(stdout,"\n");
}
}
fprintf(stdout,"\n\n");
fflush(stdout);
}
}
}

```

```

for (i=1;i<=count;i++)
{
 if (pK_ptr[i].bdry_flag
 && pR_ptr[i].aim>=0.0) flag++;
 else if (!pK_ptr[i].bdry_flag
 && ((pR_ptr[i].aim-2.0*M_PI)>toler
 || (2.0*M_PI-pR_ptr[i].aim)>toler)) flag++;
}
if (flag)
{
 fprintf(stdout,"AIMS: \n");
 for (i=1;i<=count;i++)
 {
 fprintf(stdout," %.6e ",pR_ptr[i].aim);
 if ((i % 5)==0) fprintf(stdout,"\n");
 }
 fprintf(stdout,"\n\n");
 fflush(stdout);
}
}
if (act>1)
{
 fprintf(stdout,"RADII: \n");
 for (i=1;i<=count;i++)
 {
 fprintf(stdout," %.10e ",radius(p,i));
 if ((i % 5)==0) fprintf(stdout,"\n");
 }
 fprintf(stdout,"\n\n");
 fflush(stdout);
 fprintf(stdout,"CENTERS: \n");
 for (i=1;i<=count;i++)
 {
 fprintf(stdout," %.6e %.6e ",pR_ptr[i].center.re,
 pR_ptr[i].center.im);
 if ((i % 2)==0) fprintf(stdout,"\n");
 }
 fprintf(stdout,"\n\n");
 fflush(stdout);
 fprintf(stdout,"CIRCLE-COLORS: %d \n",count);
 for (i=1;i<=count;i++)
 {
 fprintf(stdout," %d %d ",i,pK_ptr[i].color);
 if ((i % 5)==0) fprintf(stdout,"\n");
 }
 fflush(stdout, "\n\n");
}
}
if (act>2)
{
 fprintf(stdout,"ANGLESUMS: \n");
 for (i=1;i<=count;i++)
 {

```

```

 fprintf(stdout, " %.6e ", pR_ptr[i].curv);
 if ((i % 5) == 0) fprintf(stdout, "\n");
 }
 fprintf(stdout, "\n\n");
 fflush(stdout);
 if (p->Andr_ptr != NULL)
 {
 fprintf(stdout, "ANDREEV_RADII: \n");
 for (i=1; i<=count; i++)
 {
 dum=((p->Andr_ptr)+i)->rad;
 if (dum>0) fprintf(stdout, " %.6e ", dum); /*-log(dum));*/
 else fprintf(stdout, " %.6e ", dum);
 if ((i % 5) == 0) fprintf(stdout, "\n");
 }
 fprintf(stdout, "\n\n");
 }
}
}
fprintf(stdout, "END\n");
fflush(stdout);
return 1;
}
else
/* only partial data
act: 100==> geometry/a/b/c, 101==> radii,
102==> centers, 103==> angle sums,
104==> aims, 105==> overlaps, 106==> Andr radii */
{
 fprintf(stdout, "CHECKCOUNT: %d\n\n", count);
 if (act==100)
 {
 fprintf(stdout, "ALPHA/BETA/GAMMA: %d %d %d\n\n",
 p->alpha, p->beta, p->gamma);
 if (p->hes==0) sprintf(buf, "euclidean\n");
 else sprintf(buf, "hyperbolic\n");
 fprintf(stdout, "GEOMETRY: %s\n", buf);
 }
 else if (act==101)
 {
 fprintf(stdout, "RADII: \n");
 for (i=1; i<=count; i++)
 {
 fprintf(stdout, " %.6e ", radius(p, i));
 if ((i % 5) == 0) fprintf(stdout, "\n");
 }
 fprintf(stdout, "\n\n");
 fflush(stdout);
 }
 else if (act==102)
 {
 fprintf(stdout, "CENTERS: \n");
 for (i=1; i<=count; i++)
 {

```

```

 fprintf(stdout, " %.6e %.6e ", pR_ptr[i].center.re,
 pR_ptr[i].center.im);
 if ((i % 2) == 0) fprintf(stdout, "\n");
 }
 fprintf(stdout, "\n\n");
 fflush(stdout);
}
else if (act == 103)
{
 fprintf(stdout, "ANGLESUMS: \n");
 for (i = 1; i <= count; i++)
 {
 fprintf(stdout, " %.6e ", pR_ptr[i].curv);
 if ((i % 5) == 0) fprintf(stdout, "\n");
 }
 fprintf(stdout, "\n\n");
 fflush(stdout);
}
else if (act == 104)
{
 fprintf(stdout, "AIMS: \n");
 for (i = 1; i <= count; i++)
 {
 fprintf(stdout, " %.6e ", pR_ptr[i].aim);
 if ((i % 5) == 0) fprintf(stdout, "\n");
 }
 fprintf(stdout, "\n\n");
 fflush(stdout);
}
else if (act == 105 && p->overlap_ptr != NULL)
{
 fprintf(stdout, "OVERLAPS: \n");
 for (i = 1; i <= count; i++)
 {
 fprintf(stdout, "\n%d ", i);
 for (j = 0; j <= pK_ptr[i].num; j++)
 {
 fprintf(stdout, " %.6e ",
 *(p->overlap_ptr + i * (NUM_PETALS + 1) + j));
 if ((j % 5) == 4) fprintf(stdout, "\n");
 }
 }
 fprintf(stdout, "\n\n");
 fflush(stdout);
}
else if (act == 106 && p->Andr_ptr != NULL)
{
 fprintf(stdout, "ANDREEV_RADII: \n");
 for (i = 1; i <= count; i++)
 {
 dum = ((p->Andr_ptr) + i) ->rad;
 if (dum > 0) fprintf(stdout, " %.6e ", dum); /*-log(dum)*/;
 else fprintf(stdout, " %.6e ", dum);
 }
}

```



```

{
 for (k=0; k<3 && !hit; k++)
 if (pface[j].vert[k]==a)
 {
 hit=j;
 index=k;
 }
}
if (!hit) return 0;
p->first_face=hit;
pface[hit].index_flag=index;
pface[hit].plot_flag=1;
for (j=0; j<3; j++) pK_ptr[pface[hit].vert[j]].plot_flag=1;
lastface=hit;
mp2[32][32]=pface[hit].vert[0];
f_a_rad[32][32]=pR_ptr[pface[hit].vert[0]].rad;
f_a_aim[32][32]=pR_ptr[pface[hit].vert[0]].aim;
mp2list[pface[hit].vert[0]].row=32;
mp2list[pface[hit].vert[0]].col=32;
f_a_rad[31][32]=pR_ptr[pface[hit].vert[1]].rad;
f_a_aim[31][32]=pR_ptr[pface[hit].vert[1]].aim;
mp2[31][32]=pface[hit].vert[1];
mp2list[pface[hit].vert[1]].row=31;
mp2list[pface[hit].vert[1]].col=32;
f_a_rad[31][33]=pR_ptr[pface[hit].vert[2]].rad;
f_a_aim[31][33]=pR_ptr[pface[hit].vert[2]].aim;
mp2[31][33]=pface[hit].vert[2];
mp2list[pface[hit].vert[2]].row=31;
mp2list[pface[hit].vert[2]].col=33;
count=3;
k=1;
loopcheck=1;
while (count<p->nodecount && loopcheck<50000)
{
 loopcheck++;
 hit=0;
 /* while (trace!=NULL && (j=trace->v)>0 && j<=F && (pface[trace->v].plot_flag ||
face_okay(pK_ptr,pface[j])!=-2))
 trace=trace->next; */
 if (j<F && (index=face_okay(pK_ptr,&pface[j]))>=0) hit=j;
 else
 {
 if (k++ > F) k=1;
 while (pface[k].plot_flag && k <= F) k++;
 if (!pface[k].plot_flag)
 {
 index=face_okay(pK_ptr,&pface[k]);
 if (index==(-2)) pface[k].plot_flag=1;
 else if (index>=0) hit=k;
 }
 }
}
if (hit)
{

```

```

pface[hit].plot_flag=1;
pface[hit].index_flag=index;
pface[lastface].next_face=hit;
v=pface[hit].vert[(index+2) % 3];
pK_ptr[v].plot_flag=1;
/* pK_ptr[v].face=hit; */
lastface=hit;
trow[0]=mp2list[pface[hit].vert[index]].row;
trow[1]=mp2list[pface[hit].vert[(index+1) % 3]].row;

tcol[0]=mp2list[pface[hit].vert[index]].col;
tcol[1]=mp2list[pface[hit].vert[(index+1) % 3]].col;
count++;
tvert = pface[hit].vert[(index+2) % 3];
if (trow[0]==trow[1])
{
 mp2list[tvert].col=tcol[0];
 if (tcol[0]>tcol[1]) mp2list[tvert].row=trow[0]-1;
 else mp2list[tvert].row= trow[0]+1;
}
else if (tcol[0]==tcol[1])
{
 mp2list[tvert].row=trow[1];
 if (trow[0]>trow[1]) mp2list[tvert].col= tcol[0]+1;
 else mp2list[tvert].col= tcol[0]-1;
}
else
{
 mp2list[tvert].col=tcol[1];
 mp2list[tvert].row=trow[0];
}
mp2[mp2list[tvert].row][mp2list[tvert].col]=tvert;
f_a_rad[mp2list[tvert].row][mp2list[tvert].col]
 =pR_ptr[tvert].rad;
f_a_aim[mp2list[tvert].row][mp2list[tvert].col]
 =pR_ptr[tvert].aim;
k=1;
}
if (loopcheck >= 50000) return 0;
else pface[lastface].next_face=p->first_face;
}
return 1;

```

```

} /* facedraworder */

```

```

int
face_okay(pK_ptr,face)
/* return -1 if face notready to draw, -2 if vertices already drawn, else return the index of the circle to
draw first*/
{
 struct K_data *pK_ptr;
 f_data *face;
}

```

```

if (pK_ptr[face->vert[0]].plot_flag && pK_ptr[face->vert[1]].plot_flag)
{
 if (pK_ptr[face->vert[2]].plot_flag) return -2;
 else return 0;
}
if (pK_ptr[face->vert[1]].plot_flag && pK_ptr[face->vert[2]].plot_flag)
{
 if (pK_ptr[face->vert[0]].plot_flag) return -2;
 else return 1;
}
if (pK_ptr[face->vert[2]].plot_flag && pK_ptr[face->vert[0]].plot_flag)
{
 if (pK_ptr[face->vert[1]].plot_flag) return -2;
 else return 2;
}
return -1;
} /* face_okay*/

```

int

complex\_count(p) /\* Identify and count the no. of bdry and interior  
nodes, bdry and interior components, Euler characteristic, genus, etc,  
for pack p. Stores pointers to bdry and int components in packdata (at

most

MAX\_COMPONENTS of each). Sets default color codes for

```

 circles/faces.*/
 struct p_data *p;
{

 int i,node,count,bcount,new_vert,num_edges,m,n;
 int comp_count,ptr,k,j,fj,icount,more_flag,next_bdry;
 struct K_data *pK_ptr;

 pK_ptr=p->packK_ptr;

 /* some initialization and counting */
 node=p->nodecount;
 for (i=1;i<=MAX_COMPONENTS;i++)
 p->bdry_starts[i]=p->int_starts[i]=0;
 bcount=0;
 comp_count=0;
 for (i=1;i<=node;i++) /* initialize plot_flag, color, bdry_flag */
 {
 pK_ptr[i].plot_flag=0;
 pK_ptr[i].color=201;
 if (pK_ptr[i].flower[0]!=pK_ptr[i].flower[pK_ptr[i].num])
 {
 bcount++;
 pK_ptr[i].bdry_flag=1;
 }
 else pK_ptr[i].bdry_flag=0;
 }
 p->intnode=node-bcount;

```

```

/* identify bdry components, give pointers; note: index starts with 1 */
count=0;
new_vert=0;
do
{
 new_vert++;
 if (pK_ptr[new_vert].bdry_flag && !pK_ptr[new_vert].plot_flag)
 {
 comp_count++;
 if (comp_count>MAX_COMPONENTS)
 {
 sprintf(msgbuf,
 "This complex has too many bdry components.");
 emsg();
 return 0;
 }
 p->bdry_starts[comp_count]=new_vert;
 next_bdry=new_vert;
 do
 {
 pK_ptr[next_bdry].plot_flag=1;
 next_bdry=pK_ptr[next_bdry].flower[0];
 count++;
 if (count>30000)
 {
 sprintf(msgbuf,"Problem tracing a boundary path; combinatoric error.");
 emsg();
 return 0;
 }
 }
 while (next_bdry!=new_vert);
 }
}
while (count<bcount);
p->num_bdry_comp=comp_count;
/* identify int components and pointers to them */
icount=node-bcount; /* number of interiors */
if (icount>0)
{
 comp_count=0;
 do
 {
 ptr=0;
 do {ptr++;}
 while (pK_ptr[ptr].plot_flag);
 /* find smallest index not plotted (hence int node) */
 comp_count++;
 if (comp_count>MAX_COMPONENTS)
 {
 sprintf(msgbuf,
 "This complex has too many interior components.");
 emsg();
 return 0;
 }
 }
}

```

```

 }
 icount--;
 p->int_starts[comp_count]=ptr;
 pK_ptr[ptr].plot_flag=ptr;
 do
 {
 more_flag=0; /* will tell me whether I am still finding anything */
 for (k=ptr;k<=node;k++)
 {
 if (pK_ptr[k].plot_flag==ptr)
 {
 for (j=0;j<=pK_ptr[k].num;j++)
 {
 fj=pK_ptr[k].flower[j];
 if (!pK_ptr[fj].plot_flag)
 {
 pK_ptr[fj].plot_flag=ptr;
 icount--;
 more_flag=1;
 }
 }
 }
 }
 } /* end of inside do */
 while (more_flag && icount>0);
} /* end of outside do */
while (icount>0);
p->num_int_comp=comp_count;
} /* end of 'if icount' */
/* find Euler characteristic and genus */
count=0;
for (k=1;k<=node;k++) count += pK_ptr[k].num;
p->facecount=count/3;
num_edges=(count + bcount)/2;
p->euler=node-num_edges+p->facecount;
p->genus=(2-p->euler-p->num_bdry_comp)/2;
/* check if geom is appropriate */
/* find ordered triples of vertices for faces, store in packdata */
count=1;
i=1;
while (count<=p->facecount && i<=node)
{
 for (j=0;j<pK_ptr[i].num;j++)
 {
 m=pK_ptr[i].flower[j];
 n=pK_ptr[i].flower[j+1];

 if (m>i && n>i)
 {
 p->faces[count].vert[0]=i;
 p->faces[count].vert[1]=m;
 p->faces[count].vert[2]=n;
 p->faces[count].color=201;

```

```
 count++;
 }
 i++;
}
return 1;
} /* complex_count */
```

## VITA

Dennis Depp was born in DuBois, Pennsylvania on November 13, 1961. He attended the public school system of Jefferson County, Pennsylvania where he graduated from Punxsutawney High School in June 1979. He entered Indiana University of Pennsylvania, Indiana, Pennsylvania during August of 1979 and graduated in December 1984 when he received the Bachelor of Science degree with a double major in Applied Mathematics and Computer Science. He completed education courses leading to teacher certification at Indiana University of Pennsylvania in August 1988. After teaching for several years, he entered The University of Tennessee, Knoxville, Tennessee in the fall of 1991, officially receiving the Master's of Science degree in December 1996. He is presently employed as a Computer Analyst within the Information Management Services Division at Lockheed Martin Energy Systems in Oak Ridge, Tennessee.