A NEW APPROACH TO SOLVING 2-DIMENSIONAL PART NESTING AND LAYOUT PROBLEMS

By

Ananda Adhir Debnath

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Mechanical Engineering

1997

ABSTRACT

A NEW APPROACH TO SOLVING 2-DIMENSIONAL
PART NESTING AND LAYOUT PROBLEMS

By

Ananda Adhir Debnath

This thesis presents an approach to the 2-dimensional nesting and layout problem. Arbitrarily shaped geometric parts are placed in a near-to-optimal manner on an arbitrarily shaped stock. Features extracted from parts are tried in various configurations against features extracted from the stock. An optimal placement for a part is decided by computing a 'score' for that part depending on the area of the part, the area of the shadows the part casts and the contact length between the source and target features. The stock profile is then updated to exclude the recently nested part and the heuristic begins anew. Various scoring methods have been explored and recommendations made. The algorithm uses a simplistic approach and gets excellent results. Recommendations have also been made in extending this methodology for more complex shapes and problems.

This research is dedicated to the Case Center at the Michigan State University

ACKNOWLEDGEMENTS

I wish to express my gratitude to my advisor, Dr. Erik D. Goodman for his guidance and immense help throughout my graduate research and study. I thank Dr. Ronald Rosenberg for being on my committee. Warm thanks to Dr. William F. Punch III for serving on my committee and helping me with the programming aspects of this research. Thanks to my parents, sister and the rest of my family for their love and support. Special thanks to Timothy Hinds and Joyce Foley for their immense help and support. My thanks to B. S. Prabhu of the Indian Institute of Technology, Bombay, for helping me prepare for this research work and for furnishing me with a wealth of literature on the topic. Warm thanks to Shanthi for her love, understanding and help. Finally, thanks to all my friends and fellow researchers for their scientific and intuitive help.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 - INTRODUCTION

Various layout and cutting problems are of importance to manufacturing, as they involve the optimal use of valuable raw materials. In the parts nesting problem, the objective is to place a given set of polygonal shapes (markers, templates, or patterns) on a sheet of some material (sheet metal, cloth, cardboard, etc.), without allowing overlaps, in order to minimize the waste left over when the shapes are cut out. Expensive stock material can be wasted if the layout has been made inefficiently. Even though similarities exist in real-world cutting and packing problems, they often differ significantly with respect to specific goals, constraints and other criteria.

PROBLEM STATEMENT

The problem is thus stated as, "*The optimal arrangement of 2-dimensional polygonal parts onto a 2-dimensional polygonal stock piece, such that none of the parts overlap each other and all the parts that can be arranged, are contained within the boundaries of the stock piece.*"

Given a set of 2-dimensional parts, we need to *pack* the pieces into a configuration that minimizes the area of the pockets of empty space between the parts. Depending upon the application, we would need to decide the placement policy. We could place the parts such

that they would cluster towards one corner of the stock. Alternately, we could place the parts such that they would be close to a specified edge of the stock. The necessary constraints are that no part overlaps another and no part oversteps the boundaries of the stock.

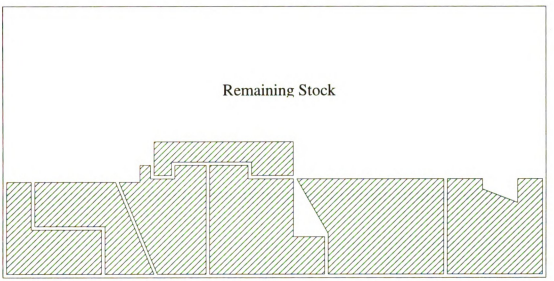

Figure 1. A typical nested solution

The *optimality* of the nested layout though is industry-specific. When the parts are cut out using flame or plasma cutting machines, the efficiency of a solution may be measured as a function of the ratio of the area of the closed polygon describing the outline of all the nested parts against the total area of all the parts.

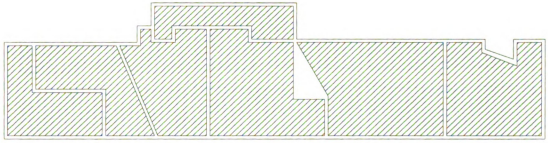

Figure 2. Optimality: Flame cut

Some other fabrication industries dealing with larger parts, usually employ initial large guillotine cuts. This plate is then sent to a cutting shop and the leftover material is scrapped. Hence, optimality is usually a function of how large a chunk needs to be cut off from an available metal plate. *This is not to be confused with the guillotinable nesting problem. A guillotinable pack is where all the parts can be cut out using guillotine cuts.*

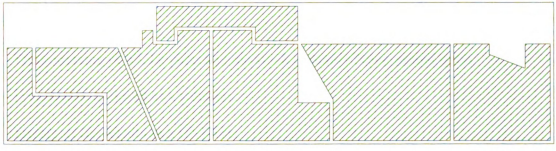

Figure 3. Optimality: Guillotine cut

In some other industries like the fabric industry and stamping industry, the optimality may be a function of how *many* parts can be nested within the given (usually rectangular) stock piece.

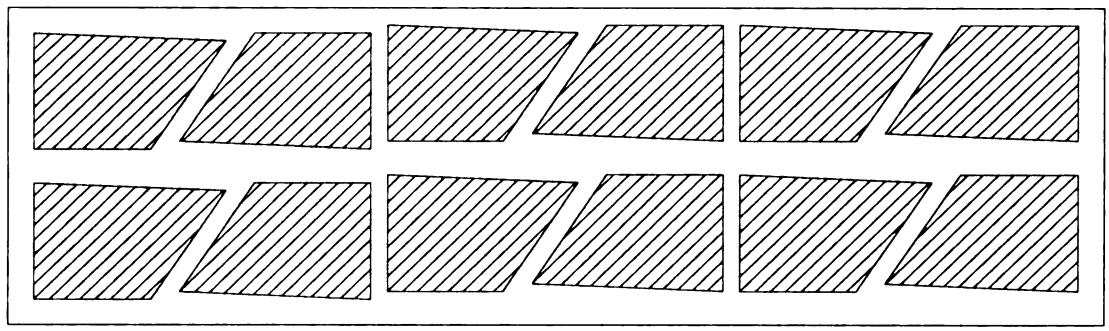

Figure 4. A typical stamping problem

BASIC OUTLINE OF THE APPROACH

The approach our heuristic takes is that of *an iterative nesting process based on matching up features from the parts to be nested against features on the stock.* The steps the heuristic takes is:

1. Reading in of the part and the stock data from a standard Digital Exchange Format (DXF) file.
2. Preprocessing of all the parts and the stock currently being nested upon.
3. Progressive building of a solution.
4. Saving of the results in a DXF file.

All the parts are initially pre-processed and feature data is extracted and stored into an array before beginning the nesting loop. A target feature is extracted from the stock by selecting the lowest vertex of the current stock profile. If there is more than one vertex with the same y co-ordinate, then the vertex with the smallest x co-ordinate is selected. *All* the corners on all the existing parts are then matched up in 2 configurations against this target feature and each match is evaluated by a scoring function. The best feature

match is retained and the corresponding part is placed in that configuration. This part is then removed from the list of parts that remain to be nested and the stock profile is updated. The process then repeats with the remaining parts. In case a target feature cannot nest any part in any valid configuration, the central vertex of that feature in included in a 'bad points' array and is excluded from being considered as a target feature in subsequent nesting loops.

CHAPTER 2 - LITERATURE REVIEW & BACKGROUND WORK

LITERATURE REVIEW

A significant amount of research has been devoted to the nesting problem over the past few decades. A number of researchers have approached this problem by nesting irregular geometry into simpler shapes and then packing these simpler shapes onto the available area. The most popular shape used for this is the rectangle. *Freeman and Shapira (1975)* tackled the problem of enclosing a given closed curve within the smallest possible rectangle. This approach proved satisfactory only when the pieces themselves were close in shape to rectangles and they had no constraints on their orientation. *Adamowicz and Albano (1976)* formulated a more effective approach by trying to nest more than one piece into a rectangle and by placing an acceptance threshold level on the area utilization. The logic was that if the placement of pieces within a rectangle met some pre-determined utilization criteria, then the problem was reduced to packing the resultant rectangles optimally into the stock with maximum utilization.

An alternative to using rectangles is to fit the parts into other regular shapes and then use a tiling algorithm to place the identical regular polygonal shapes into the plane. *Dori and Ben-Bassat (1984)* based their packing algorithm of this type on tessellation using hexagons. They limited themselves to convex polygons although they stated that similar

results could be achieved using non-convex polygons. *Karoupi and Loftus (1991)* extended this idea to include automatic nesting of curved and non-convex polygons. They nested non-convex polygons into an appropriate convex polygon, encased this non-convex polygon within a suitable hexagon and tiled the stock plane. Most of these methods usually cannot compete with direct methods in terms of solution quality. Direct methods like the straightforward single pass method, just take the pieces in a decided order and place them on the sheet according to some *placement policy*. Changing the ordering of parts may generate various solution nests and the best solution may be chosen. The Monte Carlo method is one such algorithm, in which the parts are randomly thrown into the stock sheet. *Böhme and Graham (1979)* tried this method and suggested that approximately 2000 such random trials are usually required to get satisfactory results. The best solution was then fine-tuned by fine random perturbations.

Some placement policies were tried out from successful rectangular nesting strategies. *Qu and Sanders (1987)* suggested approximating the shape of the part by a *union* of rectangular sections. The method was based on using a union of rectangles of different sizes to approximate the parts to be nested.

Other popular placement policies were the use of a directional placement policy (*e.g. lowest and leftmost vertex of the stock*). *Dowsland and Dowsland (1993)* used a random left-most placement policy and allowed for pieces to jump over pieces to fill gaps. This was to allow the smaller pieces to fill in the larger spaces left between placements of the bigger parts. *Amaral et al. (1990)* used a method that selected the next part dynamically

based on a sliding process. The pieces were ordered in decreasing order of area and two different placement policies were used for large and small pieces. The use of different placement policies was to facilitate the nesting of smaller pieces between the gaps formed by bigger pieces on nesting. *Albano and Sapuppo (1980)* also worked with the more complex problem in which the pieces may be placed in a number of orientations. Along with using a leftmost placement policy, they also allowed some backtracking. *Ismail and Hon (1992)* tackled the blank nesting problem by using a genetic algorithm. The results suggested that this type of approach was worthy of further consideration. *Batishchev (1996)* tackled the problem of nesting rectangular parts with a 2 level combination of a genetic algorithm and a wave theory scheme. *Fujita et al. (1993)* also formulated an approach using with a 2 level combination of a local minimization routine and a genetic algorithm. *Kröger (1995)* used a sequential and a parallel genetic approach to the guillotinable bin-packing problem. *G-C Han and S-J Na (1996)* used a two-stage method with a neural-net based heuristic for generating a good initial layout and a simulated annealing algorithm for fine tuning the solution.

EXISTING TECHNOLOGY

Nesting by hand

In most of the fabrication industry, nesting is carried out by hand. The quality of hand nesting is usually dependent upon the complexity of the geometry and skill of the person generating the nest. This approach usually works for relatively small problem sets, where either there are few parts to be nested and/or most of the parts are identical.

Traditional simple heuristics for nesting regular shapes

These heuristics give good solutions for regular identical part sets and are important benchmarking tools.

- The First Fit heuristic: Each part is placed in the first stock it fits. If no such stock exists, a new stock is used for placing the part.
- The First Fit Descending heuristic: The parts are sorted in the descending order by some criterion (*usually the area or the area of the convex hull*). The parts are then nested using the first-fit heuristic.
- The Best Fit heuristic: All the parts are placed on the stock with the smallest area available that can still accommodate the part. If no such stock exists, the part is placed on a new stock.
- The Best Fit Descending heuristic: The parts are sorted in the descending order by some criterion (*usually the area or the area of the convex hull*). The parts are then nested using the best-fit heuristic.

Structural Material Manager

E.J.E Industries, Inc manufactures this software. It is a Microsoft-DOS based software that can perform 1-dimensional packing and 2-dimensional nesting of rectangular parts. The nesting software is a subset of a program for material management. It uses a reasonably fast algorithm and allows for varying levels of optimization. It works across multiple stocks that may be user defined or standard sheets/plates picked out from a built-in database.

UltraNest

Komatsu Inc. manufactures this software as a part of CNC software for their plasma, flame and laser cutting machines. UltraNest can handle the nesting of regular and irregularly shaped parts across multiple regularly shaped stock sheets. UltraNest provides a convenient GUI for defining the part geometry and editing of a nested layout. It is also capable of reading part geometry from the current DXF and IGES formats. It runs on a version of UNIX called VENIX from a PC platform.

CATIA

This software is manufactured by Dassault and marketed by IBM. It is one of the most complex solid and surface modeling tools containing extensive manufacturing, simulation and stress analysis tools. CATIA offers a separate module for 2-dimensional nesting. The CATIA Nesting product advertises interactive capabilities for the automatic or manual nesting of parts on sheets.

The benefits of improved solutions to irregular nesting problems will be widespread. Although a lot of work has already been done in this area, it is still difficult to outperform an experienced operator with a purely automatic procedure. Much work is necessary before automated nesting methods will be able to replace human experts in many aspects of the material usage problem. This thesis furthers the existing work and opens up a whole new approach to solving this problem.

CHAPTER 3 - GEOMETRIC FRAMEWORK

For defining the problem, we created a set of basic geometric data structures. These data-structures mainly used object oriented concepts to model the problem

REPRESENTATION OF THE GEOMETRY

The basic geometric data-structures the heuristic uses are: *Point, Polygon - Part, Stock.* Each is defined to be a *class* of objects.

Point

A point object contains two data elements:

1. A double precision floating point number representing the x co-ordinate.
2. A double precision floating point number representing the y co-ordinate.

Polygon

A polygon object contains seven data elements:

1. An integer representing the number of vertices.
2. An array of point objects representing the vertices.
3. An array of double precision floating point numbers representing the internal angles at each vertex.
4. An array of double precision floating point numbers representing the length of each side. The convention used here is that the *n*th edge follows the *n*th vertex.

5. A character string of upto 31 characters representing a name for the polygon object.
6. A double precision floating point number representing the area of the polygon.
7. An integer representing the DXF color code for the polygon object.

Part

A part is extended from a polygon object. It inherits all the properties of the polygon object and adds more of its own. It also modifies some of the polygon-owned functions. A part object adds:

1. An array of boolean flags specifying whether each vertex is part of the convex hull.
2. A point object representing the lower-left corner of the rectangle containing this part. This point is dynamically maintained.
3. A point object representing the upper-right corner of the rectangle containing this part. This point is dynamically maintained.
4. An array of 4 points maintaining the vertices of the smallest enclosing rectangle of the part. These points are dynamically maintained.
5. Three boolean flags maintaining data about a part whether it may be subject to the three kinds of movement operations defined (*viz. translate, rotate and flip).*

Stock

A stock too is extended from a polygon object. It inherits all the properties of the polygon object and adds more of its own. It also modifies some of the polygon-owned methods. A stock object does not add any additional data members.

Feature

A feature contains three data members:

1. An array of 4 pointers that point to 4 vertices of a polygon object.
2. A pointer to the polygon object that contains the vertices defining this feature.
3. A double precision floating point number representing the interior angle of this feature.

GEOMETRIC OPERATIONS

These operations are divided into two sets. One set is 'owned' by the object and may be invoked by the object instance itself (called 'methods' in object-oriented jargon). The other set is a group of functions that perform operations on arguments passed to them.

Point methods

Point data structures may perform the following geometric operations:

1. Assignment '=' operation to another point where both x and y co-ordinate values are assigned.
2. Comparison '==' operation to another point that returns a boolean value of TRUE only when *both* x and y co-ordinate values are within a 10^{-6} units of the x and y values of point against which the comparison is being made.

Polygon methods

Polygon objects may perform the following geometric operations:

1. The method, `getAnyInternalPoint()` that returns a point object completely contained within this polygon object.
2. The method, `contains(point)` that returns an integer value representing whether the argument point object lies inside, on an edge, on a vertex or outside this polygon.

3. The method, `contains(polygon)` that returns an integer value representing whether the argument polygon object lies inside or outside this polygon.
4. The method, `doesAnyEdgeIntersect(polygon)` that returns a boolean value stating whether any edges of the argument polygon intersect with any of the edges of this polygon.

Part methods

In addition to the polygon methods, part objects may perform the following geometric operations:

1. The method, `set_bounds()` that sets the bounding rectangle of the part in the global co-ordinate system.
2. The method, `set_angles()` that sets the values of the internal angles of each vertex.
3. The method, `mark_convex_hull()` that marks the vertices which form a part of the convex hull of this part.
4. The method, `set_M_E_R()` that sets the minimum enclosing rectangle for this part.
5. The method, `rotate(angle, point)` that rotates this part by the specified angle (in radians) in a counter-clockwise manner around the specified point.
6. The method, `translate(point1, point2)` that translates this part by the vector formed by point1, point2.
7. The method, `flip(point1, point2)` that flips this part about the vector formed by point1, point2.

Stock methods

In addition to the polygon methods, stock objects may perform the following geometric operations:

1. The method, `leftShadow(part)` that computes the left shadow cast by the part onto the stock. It assumes that the part is completely contained with the bounds of this stock.

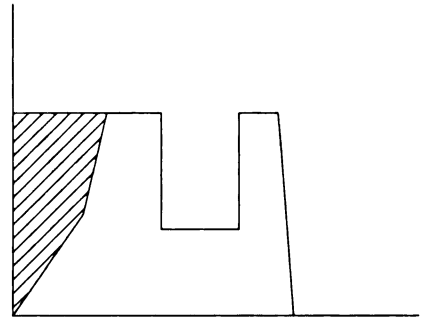

Figure 5. Left shadow

2. The method, `underShadow(part)` that computes the bottom shadow cast by the part onto the stock. It assumes that the part is completely contained with the bounds of this stock.

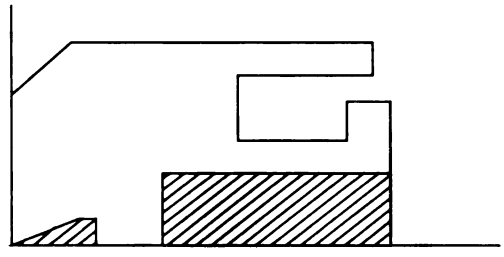

Figure 6. Bottom shadow

3. The method, `updateProfile(polygon)` that updates the profile of this stock based on the position of the part. It assumes that the part is completely contained with the bounds of this stock.

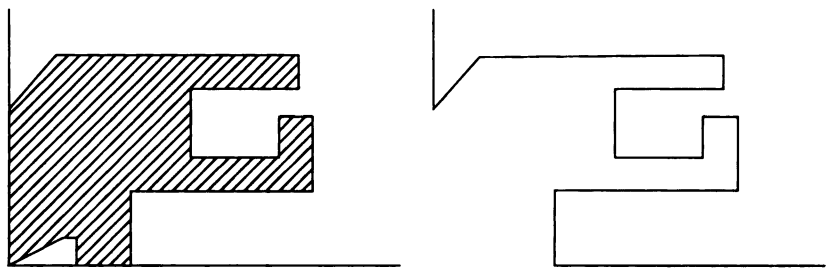

Figure 7. Update stock profile

OTHER OPERATIONS

The other main geometric functions used are:

1. The function, `contactLength(point1, point2, point3, point4)` that is used for calculating the length of contact between 2 line segments defined by (point1, point2) and (point3, point4). This function is extensively used in the scoring function.

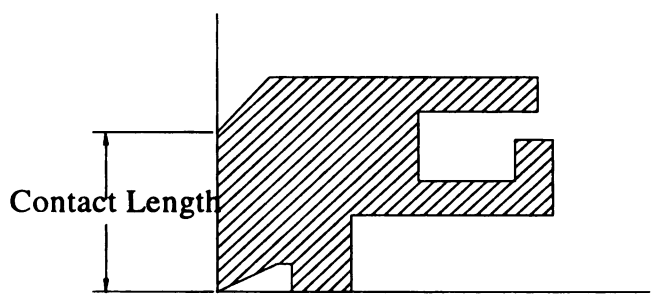

Figure 8. Contact length

2. The function, `get_angle(point1, point2)` that is used for calculating the angle made by the line segment defined by (point1, point2) and positive x-axis.
3. The function, `get_angle(point1, point2, point3)` that is used for calculating the angle swept by the line segment defined by (point2, point3) to align with line segment defined by (point2, point1) in the counter-clockwise direction.
4. The function, `get_distance(point1, point2)` that is used for calculating the distance between the specified points.
5. The function, `cornerFeatures(polygon, feature_array)` that is used for extracting all the corner features for the specified polygon and storing them in the feature array.

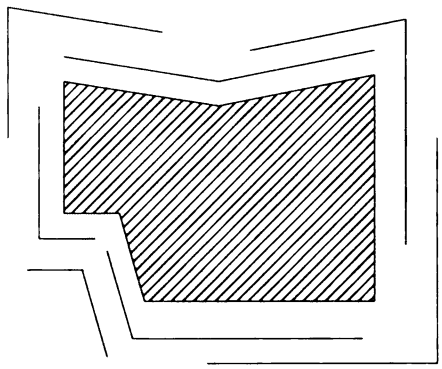

Figure 9. Corner features

6. The function, `selectStockFeatures(feature_array, badpoints, stock)` that is used for selecting the next stock feature against which the part features would be evaluated. The selected feature is added to the feature array.

7. The function, `calculateScore(part, stock, feature, feature)` is used for computing the score of a part-feature to stock-feature match. It uses the following criteria:

    - Area of the part.
    - Left Shadow area.
    - Bottom Shadow area.
    - Contact length of the features

The score is defined by the function:

$$a(\text{Area}) + b(\text{Left\_Shadow}) + c(\text{Bottom\_Shadow}) + d(\text{Contact\_Length})^2$$

where, $a, d \in \{1, 2, 3, 4\}$ and $b, c \in \{-1, -2, -3, -4\}$

CHAPTER 4 - THE NESTING HEURISTIC

The nesting heuristic is designed with the objective of *effectively matching complementary features on the parts and the stock.* Keeping to the scope of this thesis, we have defined a feature to be *an instance of two adjacent edges on a polygon.* The data contained by this type of a feature are:

1. The lengths of the 2 adjacent edges.
2. The internal angle between these edges.

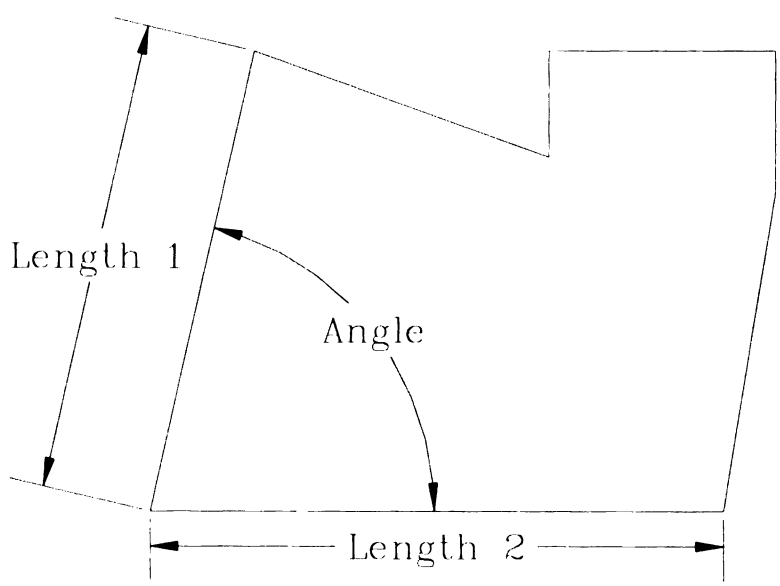

Figure 10. Feature information

The heuristic performs the following steps to create a nested layout:

1. It reads in information about all the parts and the stock from a DXF file. The heuristic maintains certain constraints and standards that the DXF file must comply with. They are:
    - Polygonal information is read only from POLYLINE entities.
    - All polygons are assumed closed so that they form a complete loop.
    - All polygons are defined in the counter-clockwise direction.
    - All curved edges within the polygons are approximated to straight edges. This is to reduce the computational complexity of the heuristic.
    - No polygon is self-intersecting.
    - The stock, onto which nesting is to be done, is defined by the first POLYLINE with the red color DXF tag.
    - Polygons do not have any internal voids that are disconnected from their defining perimeter loop.
2. It then creates instances of all parts and the stock from the DXF information.
3. It forms a *list* of features from all the part information and uses this, as its 'source' of features to nest with.
4. It selects the vertex on the stock with the lowest y co-ordinate. If more than one vertex has the same (low) y co-ordinate, the vertex with the smallest x co-ordinate is selected. This selection is based on the placement policy of *lowest and if necessary, leftmost*.

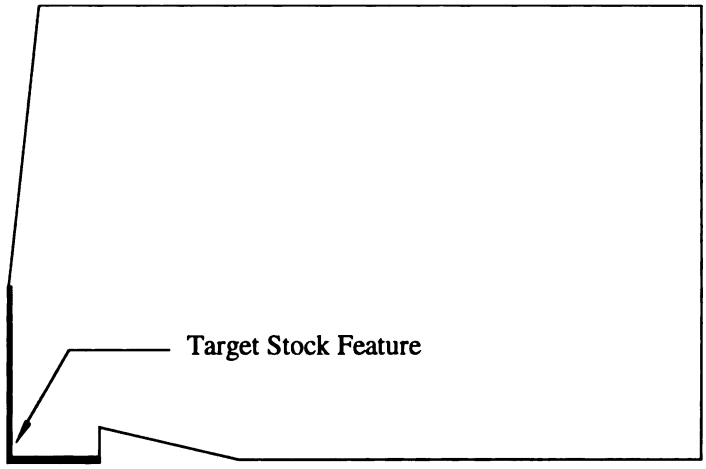

Figure 11. Target stock feature

5. It then forms a target feature on the stock with the 2 edges common to this vertex in a counter-clockwise manner.
6. It iterates through all the source part features and aligns each feature in two configurations against this target stock feature.

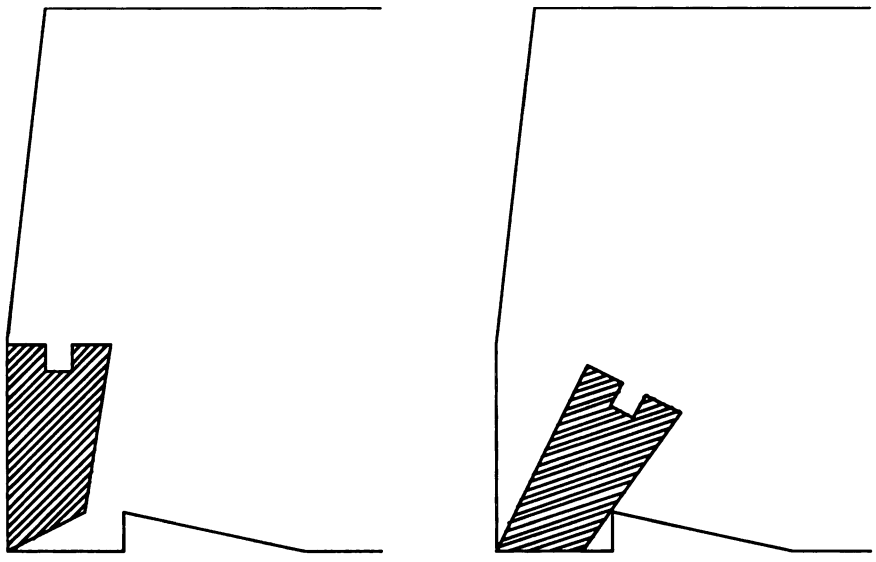

Figure 12. Two edge-to-edge configurations

7. If the part is not completely contained within the current stock, it slides the part along the current edge for its entire length till the part is contained. If no such valid configuration can be found, it assigns a 'bad-score' to this match.

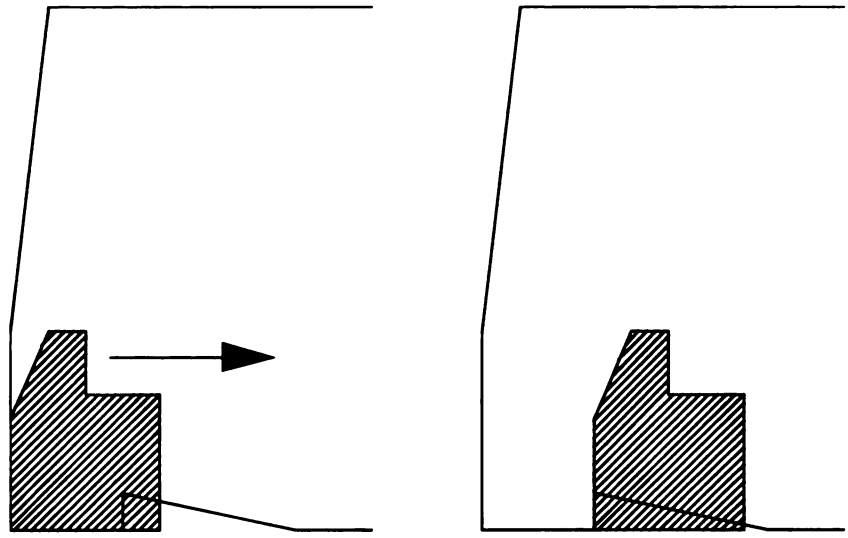

Figure 13. Slide the part along the edge

8. It assigns a score for each such configuration. The score is based on 4 parameters.
    - *Area of the part.* It is usually observed that most good solutions for nesting and packing problems exhibit a bias for packing bigger parts first and then using smaller parts to fill up the inter-polygonal gaps. We wanted to build this feature into the system as an option.
    - *Left Shadow area (see figure 5).* Since our placement policy was selected to fill up the stock from the left to the right, any *closed-off* areas of the stock would prove detrimental to a good solution.
    - Bottom Shadow area *(see figure 6).* Since out placement policy was also set to fill up the stock layer by layer along the increasing y-axis, any *closed-off* areas towards the bottom of the part would be detrimental.

- *Contact length of the feature (see figure 8).* To be able to effectively exploit a corner-feature, it was necessary for the heuristic to know *how much* contribution was being made to the nesting due to the feature itself. Given a set of parts with identical areas, left shadows and bottom shadows, a larger contact length would ensure better use of a feature. The value of this feature is squared so as to equalize the units in the scoring function.

The score is defined by the function:

$$a(\text{Area}) + b(\text{Left\_Shadow}) + c(\text{Bottom\_Shadow}) + d(\text{Contact\_Length})^2$$

where *a*, *b*, *c* & *d* are values of coefficients and usually $(b, c) < 0$ & $(a, d) > 0$.

9. The matched feature-set with the best score is retained and the part owning that feature, is placed in that configuration. All features from this part are then removed from the feature array and the stock profile is updated *(see figure 7).* In case all the scores are bad-scores, that vertex point is blacklisted and not given further consideration as a potential feature vertex, and a new stock feature is found.
10. Steps 4 through 9 are repeated till either no more parts are left or all stock vertices are exhausted and the parts cannot be fit into the stock anymore.

CHAPTER 5 - RESULTS

The Nesting Heuristic was tried with various problem sets with varying values for the weighing coefficients. The scoring used was based on the formula

$$a(\text{Area}) + b(\text{Left\_Shadow}) + c(\text{Bottom\_Shadow}) + d(\text{Contact\_Length})^2$$

where, $a, d \in \{1, 2, 3, 4\}$ and $b, c \in \{-1, -2, -3, -4\}$

The effectiveness of the solution was measured in two ways:

- (Total area of the parts)/(Area of the enclosing rectangle)
- (Total area of the parts)/(Area of the enclosing polygon)

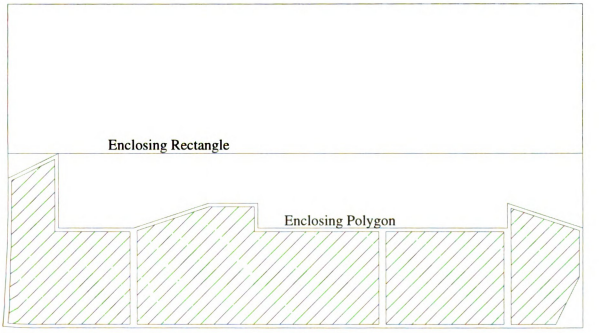

Figure 14. Effectiveness criteria

The speeds of execution is as measured on a program running within a DOS command window in Microsoft Windows 95 on an IBM PC compatible machine equipped with a Pentium processor (90 MHz) with 64 Mbytes of RAM. Various other processes were also running on the machine at the same time. An iteration was counted when step 9 from the nesting heuristic was completed, *i.e. a decision of either a part configuration or rejection of a target stock feature was made.*

INDEPENDENT PROBLEM SETS

The following problem sets were created for testing the heuristic against problems typically found in industry.

Problem 1

This problem is the nesting of typical right-angled geometric shapes within a rectangular stock shape. Shapes of this type are typically encountered in the small-scale fabrication industry.

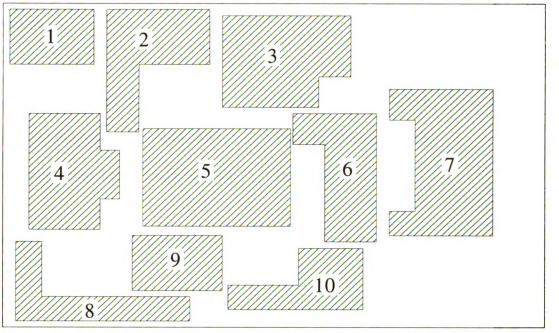

Figure 15. Problem 1

Part 1: Area = 7.3125 Part 2: Area = 12.4375 Part 3: Area = 17.1875

Part 4: Area = 14.5625 Part 5: Area = 23.0 Part 6: Area = 12.0625

Part 7: Area = 20.25 Part 8: Area = 9.0 Part 9: Area = 7.875

Part 10: Area = 9.0 Stock Area = 278.25

Total Area of Parts = 132.6875

Solution 1

`a = 1, b = -1, c = -1, d = 1`

Enclosing Rectangle Area = 178.5 Packing Ratio = 74.33%

Enclosing Polygon Area = 144.3492 Packing Ratio = 91.92%

Total computing time = 14.29 seconds Iterations = 21

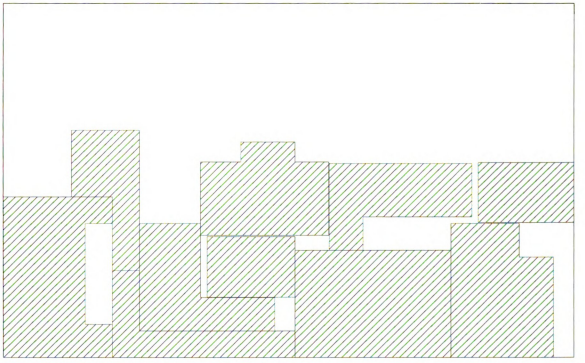

Figure 16. Problem 1 - Solution 1

Solution 2

`a = 1, b = -2, c = -2, d = 4`

Enclosing Rectangle Area = 223.65 Packing Ratio = 59.33%

Enclosing Polygon Area = 141.3385 Packing Ratio = 93.88%

Total computing time = 25.10 seconds Iterations = 24

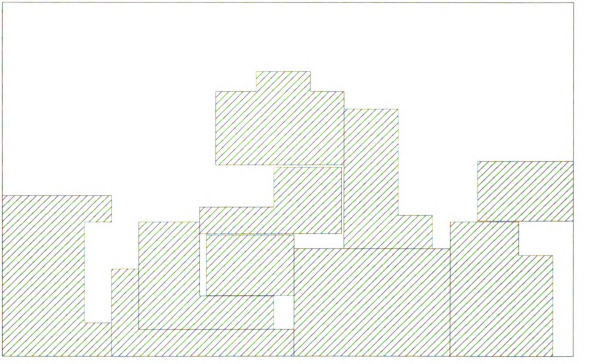

Figure 17. Problem 1 - Solution 2

Solution 3

`a = 2, b = -4, c = -3, d = 4`

Enclosing Rectangle Area = 200.235 Packing Ratio = 66.27%

Enclosing Polygon Area = 141.2913 Packing Ratio = 93.91%

Total computing time = 10.82 seconds Iterations = 14

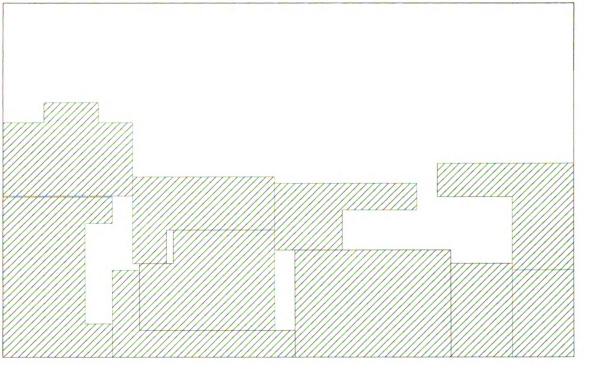

Figure 18. Problem 1 - Solution 3

Solution 4

`a = 4, b = -1, c = -2, d = 1`

Enclosing Rectangle Area = 168.0 Packing Ratio = 78.98%

Enclosing Polygon Area = 143.8963 Packing Ratio = 92.21%

Total computing time = 6.97 seconds Iterations = 11

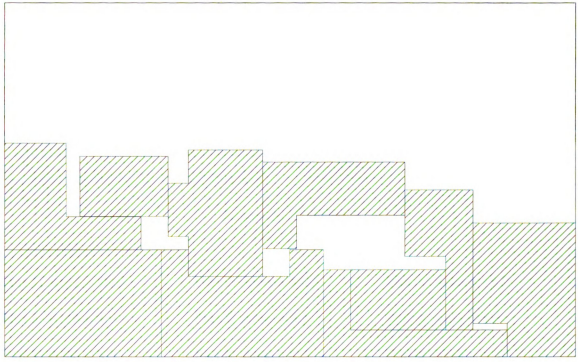

Figure 19. Problem 1 - Solution 4

Problem 2

This problem is the nesting of irregular geometric shapes within a rectangular stock. Some of the parts have a relatively high aspect ratio and the stock is oriented such that nesting is carried out iteratively along its longer edge.

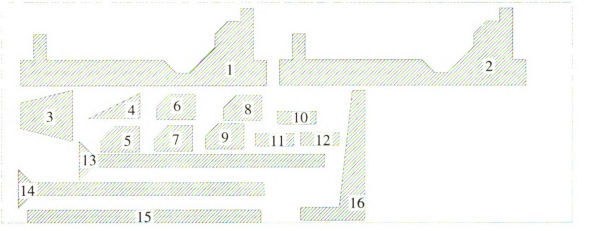

Figure 20. Problem 2

Part 1: Area = 48.0	Part 2: Area = 48.0	Part 3: Area = 12.0
Part 4: Area = 4.0	Part 5: Area = 5.5	Part 6: Area = 5.5
Part 7: Area = 5.5	Part 8: Area = 5.5	Part 9: Area = 5.5
Part 10: Area = 3.0	Part 11: Area = 3.0	Part 12: Area = 3.0
Part 13: Area = 20.0	Part 14: Area = 20.0	Part 15: Area = 18.0
Part 16: Area = 18.5	Stock Area = 748.0	

Total Area of Parts = 225.0

Solution 1

`a = 1, b = -3, c = -3, d = 2`

Enclosing Rectangle Area = 369.9598 Packing Ratio = 60.82%

Enclosing Polygon Area = 271.9970 Packing Ratio = 82.72%

Total computing time = 35.42 seconds Iterations = 18

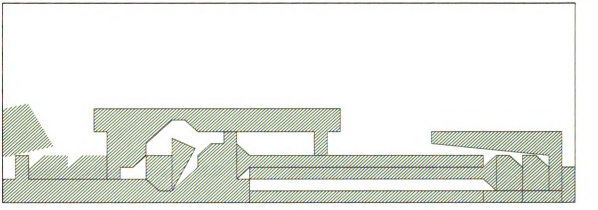

Figure 21.Problem 2 - Solution 1.

Solution 2

`a = 2, b = -3, c = -1, d = 1`

Enclosing Rectangle Area = 614.4253 Packing Ratio = 36.62%

Enclosing Polygon Area = 324.5441 Packing Ratio = 69.33%

Total computing time = 50.37 seconds Iterations = 45

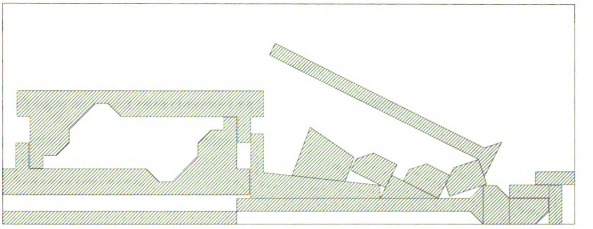

Figure 22.Problem 2 - Solution 2

Solution 3

`a = 3, b = -3, c = -1, d = 3`

Enclosing Rectangle Area = 748.0	Packing Ratio = 30.1%
Enclosing Polygon Area = 291.5242	Packing Ratio = 77.18%
Total computing time = 53.61 seconds	Iterations = 34

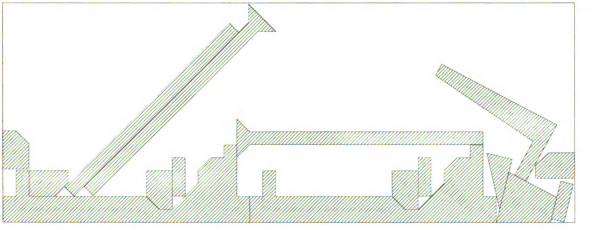

Figure 23. Problem 2 - Solution 3

Problem 3

This problem is the nesting of irregular geometric shapes within a rectangular stock. The parts have an aspect ratio close to 1 and the stock is oriented such that nesting is carried out iteratively along its shorter edge.

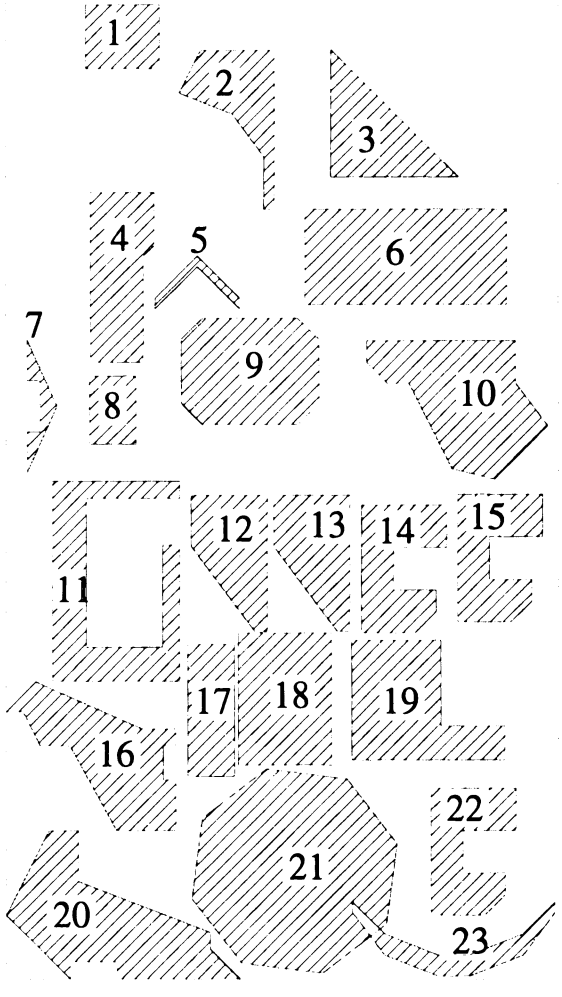

Figure 24. Problem 3

Part 1: Area = 2.6250	Part 2: Area = 3.75	Part 3: Area = 4.5
Part 4: Area = 5.3438	Part 5: Area = 0.5	Part 6: Area = 10.6875
Part 7: Area = 0.725	Part 8: Area = 1.76	Part 9: Area = 7.6250
Part 10: Area = 8.1563	Part 11: Area = 7.36	Part 12, 13: Area = 4.26
Part 14, 15: Area = 4.375	Part 16: Area = 6.68	Part 17: Area = 3.41
Part 18: Area = 6.82	Part 19: Area = 7.08	Part 20: Area = 8.9
Part 21: Area = 17.6946	Part 22: Area = 4.375	Part 23: Area = 2.1875

Stock Area = 331.5

Total Area of Parts =127.4496

Solution 1

`a = 1, b = -1, c = -1, d = 1`

Enclosing Rectangle Area = 200.3976 Packing Ratio = 63.6%

Enclosing Polygon Area = 161.8944 Packing Ratio = 78.72%

Total computing time = 486.31 seconds Iterations = 133

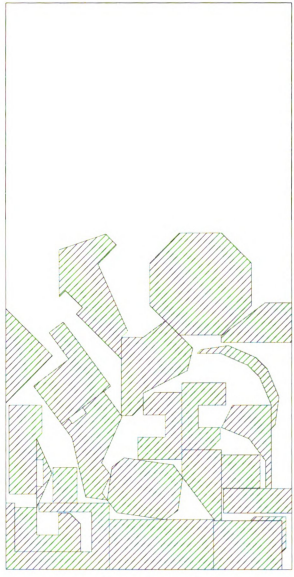

Figure 25. Problem 3 - Solution 1

Solution 2

`a = 2, b = -1, c = -2, d = 4`

Enclosing Rectangle Area = 194.8674 Packing Ratio = 65.4%

Enclosing Polygon Area = 156.8771 Packing Ratio = 81.24%

Total computing time = 1360.07 seconds Iterations = 131

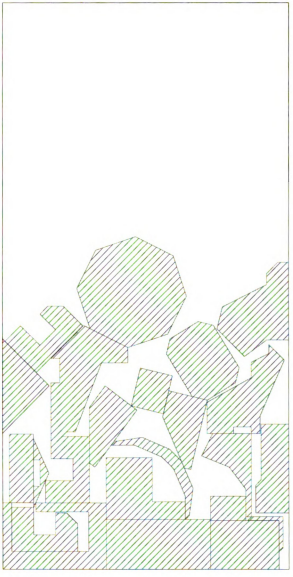

Figure 26. Problem 3 - Solution 2

Solution 3

`a = 3, b = -1, c = -4, d = 4`

Enclosing Rectangle Area = 201.6509 Packing Ratio = 63.2%

Enclosing Polygon Area = 163.7267 Packing Ratio = 77.84%

Total computing time = 2150.77 seconds Iterations = 128

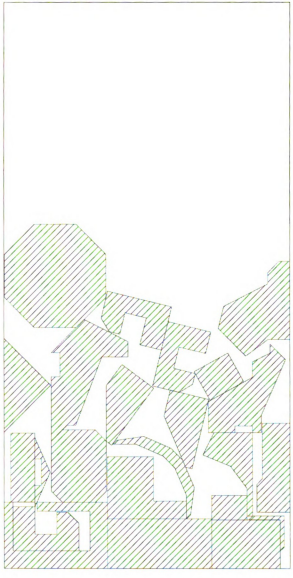

Figure 27. Problem 3 - Solution 3

Solution 4

`a = 4, b = -3, c = -1, d = 2`

Enclosing Rectangle Area = 177.8966 Packing Ratio = 71.64%

Enclosing Polygon Area = 160.0355 Packing Ratio = 79.63%

Total computing time = 1055.51 seconds Iterations = 114

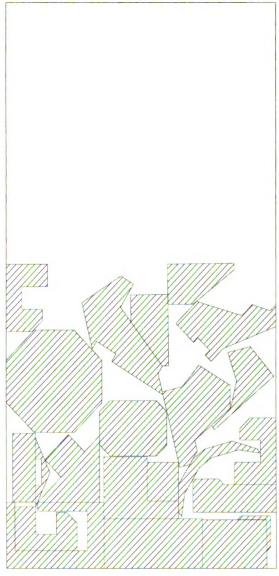

Figure 28. Problem 3 - Solution 4

Problem 4

This problem is the nesting of groups of repeating irregular geometric shapes within a rectangular stock. Nesting is carried out iteratively along the longer edge of the stock.

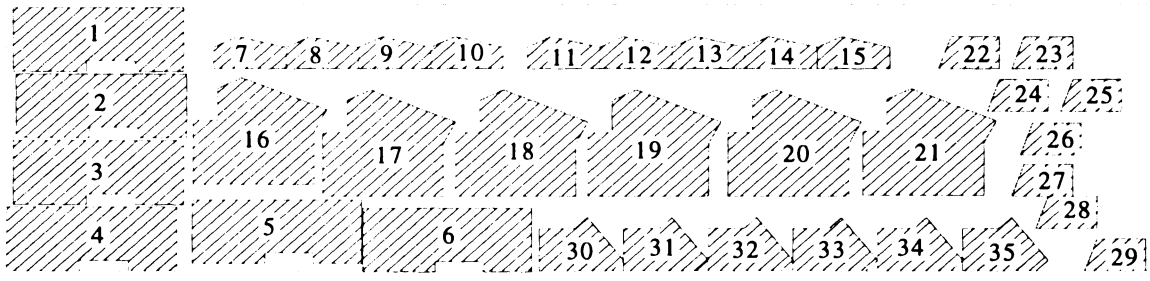

Figure 29. Problem 4

Part 1-6: Area = 5.0 Part 7-15: Area = 0.9375 Part 16-21: Area = 5.2812

Part 22-29: Area = 0.8438 Part 30-35: Area = 1.5625 Stock Area = 148.4375

Total Parts Area = 86.25

Solution 1

`a = 1, b = -1, c = -4, d = 4`

Enclosing Rectangle Area = 118.75 Packing Ratio = 72.63%

Enclosing Polygon Area = 93.88 Packing Ratio = 91.87%

Total computing time = 1574.88 seconds Iterations = 72

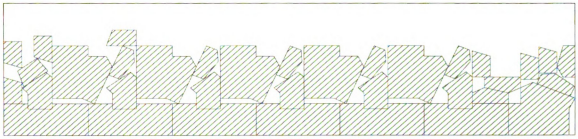

Figure 30. Problem 4 - Solution 1

Solution 2

`a = 2, b = -1, c = -4, d = 2`

Enclosing Rectangle Area = 123.5 Packing Ratio = 69.83%

Enclosing Polygon Area = 94.2702 Packing Ratio = 91.49%

Total computing time = 1652.60 seconds Iterations = 67

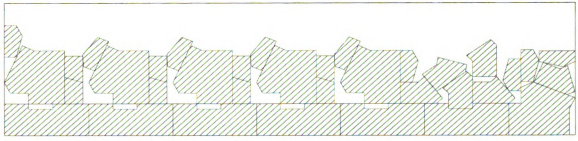

Figure 31. Problem 4 - Solution 2

Solution 3

`a = 3, b = -4, c = -4, d = 1`

Enclosing Rectangle Area = 125.9238 Packing Ratio = 68.49%

Enclosing Polygon Area = 94.9703 Packing Ratio = 90.81%

Total computing time = 1564.22 seconds Iterations = 90

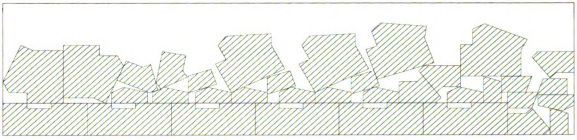

Figure 32. Problem 4 - Solution 3

Solution 4

`a = 4, b = -1, c = -2, d = 1`

Enclosing Rectangle Area = 123.3441 Packing Ratio = 69.92%

Enclosing Polygon Area = 97.3922 Packing Ratio = 88.56%

Total computing time = 1989.02 seconds Iterations = 69

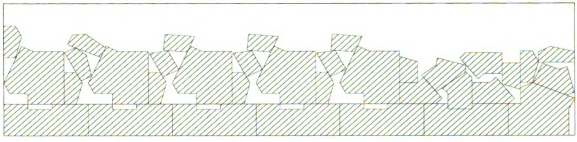

Figure 33. Problem 4 - Solution 4

PROBLEM SETS FROM PREVIOUS LITERATURE

The following problem sets were taken from previous literature mentioned earlier *(Chapter 2)*. To compare the solutions, we used the same methods for evaluation, *viz. the enclosing rectangle and the enclosing polygon.*

Comparison against a Genetic Algorithm based Nesting Method

This problem set was **approximated** from a paper by *Ismail and Hon* [13]. This method uses a genetic algorithm with specialized operators for achieving the 2-dimensional packing.

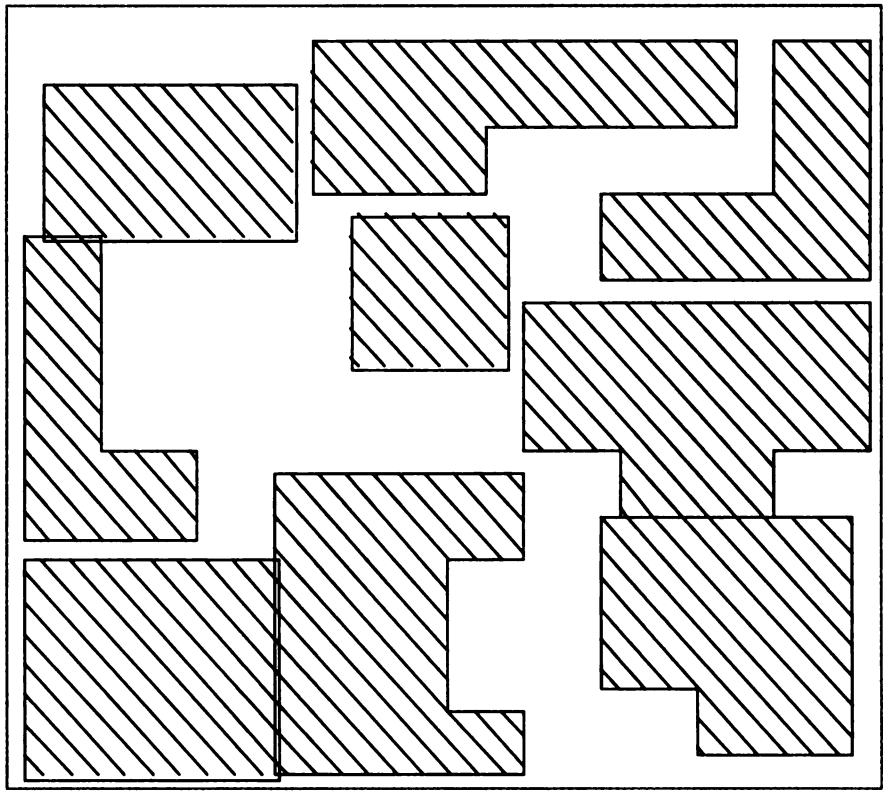

Figure 34. Solution from [13]

Total Area of parts = 61.9375.

Enclosing Rectangle Area = 97.75 Packing Ratio = 63.36%

Enclosing Polygon Area = 86.5 Packing Ratio = 71.6%

The following are some of the solutions that resulted from the experiment.

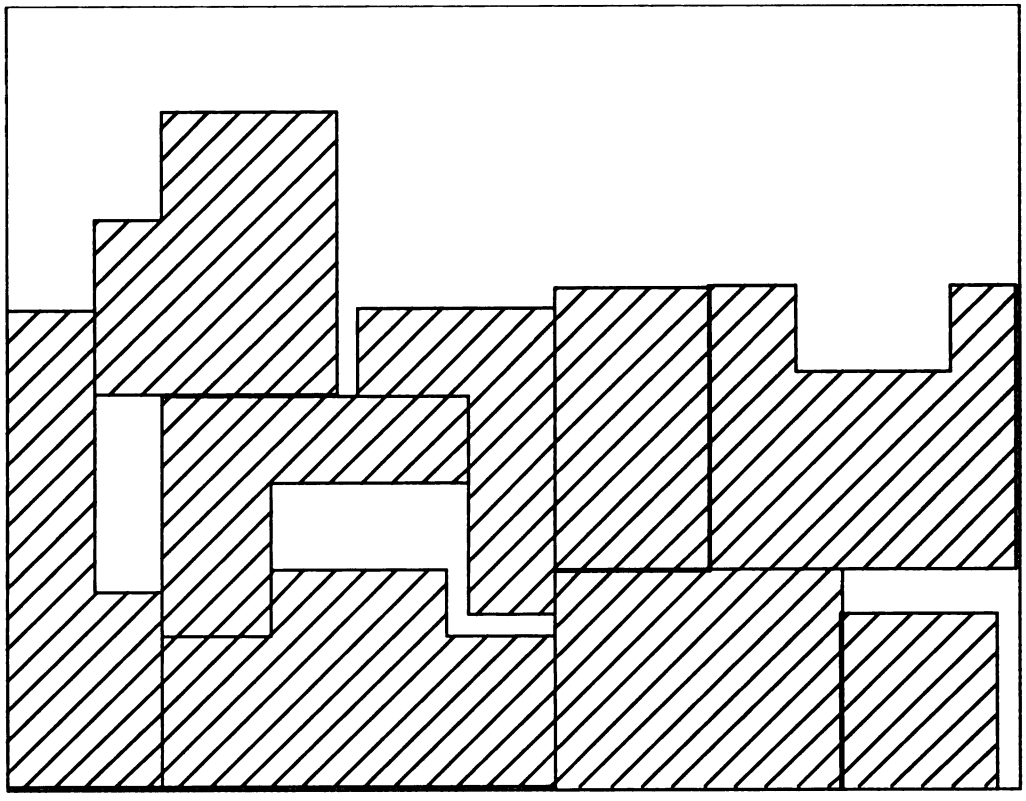

Figure 35. Solution from the heuristic with a=1, b=-1, c=-1, d=1

Enclosing Rectangle Area = 89.4125 Packing Ratio = 69.27%

Enclosing Polygon Area = 67.96625 Packing Ratio = 91.13%

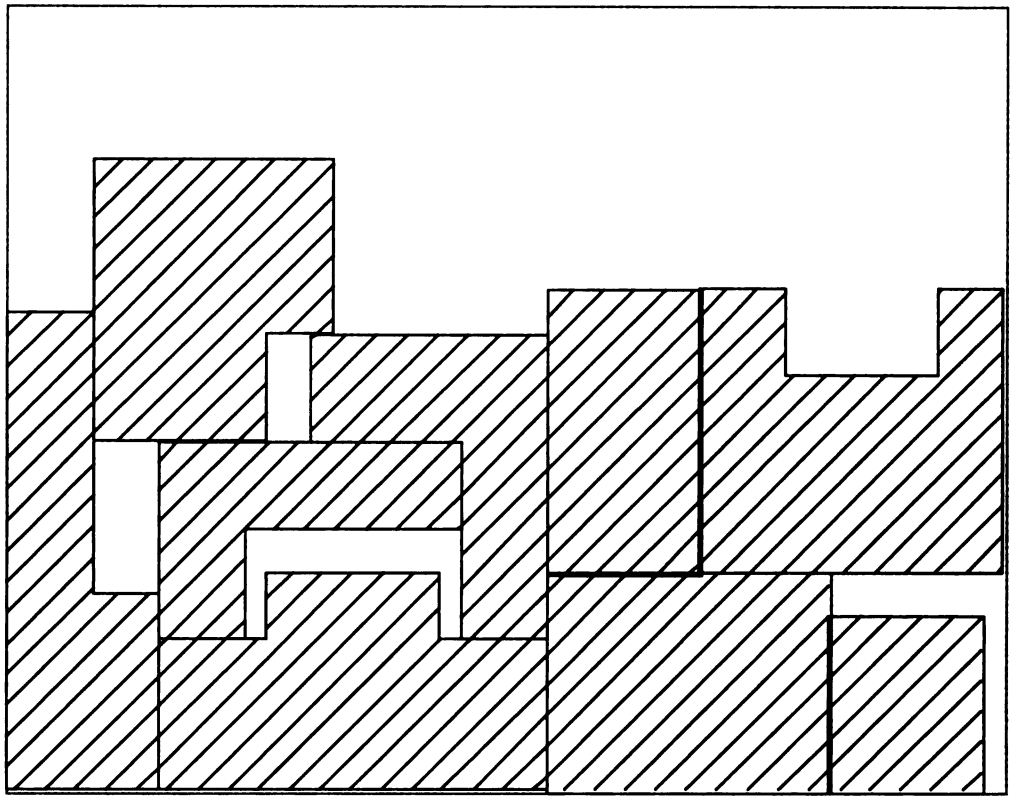

Figure 36. Solution from the heuristic with a=1, b=-1, c=-2, d=1

Enclosing Rectangle Area = 83.4325 Packing Ratio = 74.24%

Enclosing Polygon Area = 67.09 Packing Ratio = 92.32%

Since the solution from *Ismail and Hon* [13] has been approximated from digitized images made from photocopies of the paper, *a definitive comparison cannot be made* between the solutions. It does *appear* though that the Nesting Heuristic was able to outperform the solution from [13].

Comparison against a heuristic invented by Dağli and Tatoğlu

This problem set was **approximated** from a paper by *Dağli* and *Tatoğlu* [6]. This heuristic uses a sequential nesting approach that tries to minimize the area of a rectangle created when two or more parts are placed in different configurations around each other.

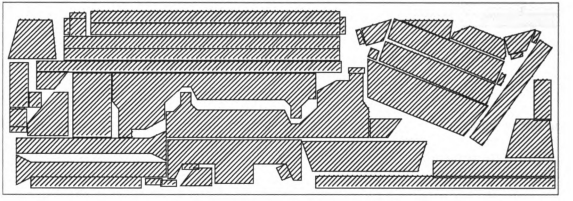

Figure 37. Solution from [6]

Total area of parts = 389.186

Enclosing Rectangle Area = 572.7459 Packing Ratio =67.95 %

Enclosing Polygon Area = 537.8946 Packing Ratio =72.35%

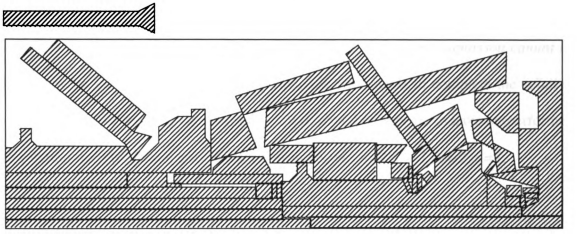

Figure 38. Solution from the heuristic with a=1, b=-1, c=-1, d=4

With the above values of weighing coefficients, the heuristic failed to complete the nest and one part was left out.

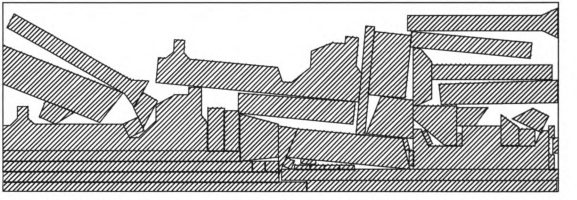

Figure 39. Solution from the heuristic with a=2, b=-1, c=-2, d=1

Enclosing Rectangle Area = 582.1265 Packing Ratio =66.86 %

Enclosing Polygon Area = 468.558 Packing Ratio =83.06%

As before, since the solution from *Dağli* and *Tatoğlu* [6] has been approximated from digitized images made from photocopies of the paper, *a definitive comparison cannot be made* between the solutions. The Nesting Heuristic was unable to complete the solution with the first set of weights, *viz. a=1, b=-1, c=-1, d=4*, but *appears* to have outperformed the solution from [6] with the second set of weights, *viz. a=2, b=-1, c=-2, d=1.*

Comparison against a heuristic invented by Lamousin et. al.

This problem set was **approximated** from a paper by *Lamousin et. al*[16].

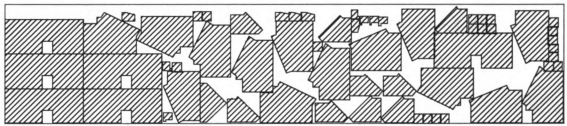

Figure 40. A solution from [16]

Total area of parts = 23.66

Enclosing Rectangle Area = 34.56 Packing Ratio =68.46%

Enclosing Polygon Area = 30.778 Packing Ratio =76.87%

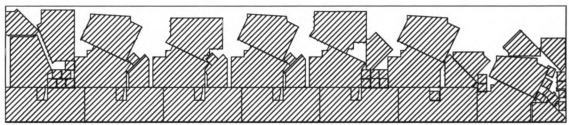

Figure 41. Solution from the heuristic with a=1, b=-1, c=-1, d=1

Enclosing Rectangle Area = 33.0752 Packing Ratio = 71.53%

Enclosing Polygon Area = 26.7443 Packing Ratio = 88.46%

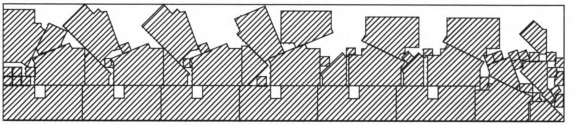

Figure 42. Solution from the heuristic with a=3, b=-1, c=-3, d=1

Enclosing Rectangle Area = 33.28 Packing Ratio = 71.09%

Enclosing Polygon Area = 27.6167 Packing Ratio = 85.67%

As before, since the solution from *Lamousin et. al*[16] has been approximated from digitized images made from photocopies of the paper, *a definitive comparison cannot be made*. It however *appears* that the Nesting Heuristic was able to outperform the solution from [16] with both sets of weights.

CHAPTER 6 - DISCUSSIONS AND CONCLUSIONS

The Nesting Heuristic was designed and implemented within the scope of this thesis to investigate a new approach for providing good solutions to 2-dimensional packing and nesting problems. The current implementation of the heuristic uses the most basic feature of polygons. The heuristic is very simplistic in its approach and runs as a *greedy* iterative process, *i.e. it selects the best current part configuration.* It does not provide for any level of backtracking. It simply tries a finite number of possible configurations and selects the best part configuration from the current set of parts under consideration for nesting. The heuristic has been tried with problem sets typically found in the industry. For most cases tried, the heuristic is able to find solutions with parts laid out with a minimum of *islands* between parts. Such islands on the stock would be the true wastes of the nested solution. There are still a few classes of problems for which the current implementation proves to be overly simplistic and is unable to find a solution.

1. The heuristic cannot handle a problem set where all parts are shaped like 'stars' and no feature can be extracted which would be completely contained within the stock on which the nesting was being performed.

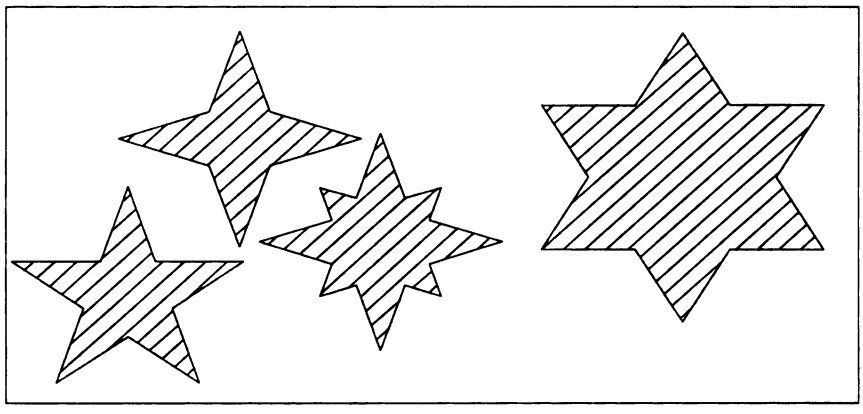

Figure 43. Limitation 1

Forming features from the convex hulls of these parts may still solve this problem, although the solution may not be very good.

2. The heuristic cannot handle unique configurations where no part features match any stock feature. A typical example is shown below:

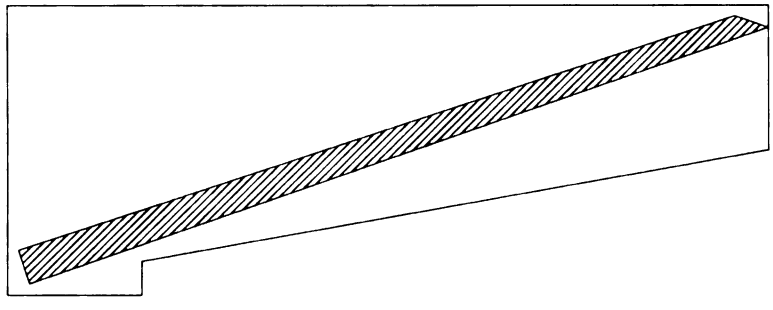

Figure 44. Limitation 2

3. Given a problem set with fixed parts, but varying the stock height, may lead to different solutions.

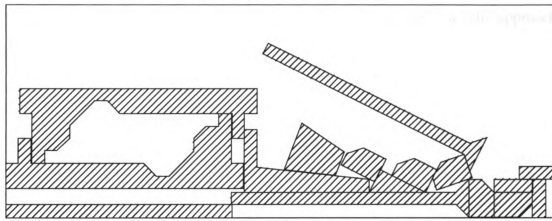

Figure 45. Problem 5: a=2, b=-2, c=-3, d=1

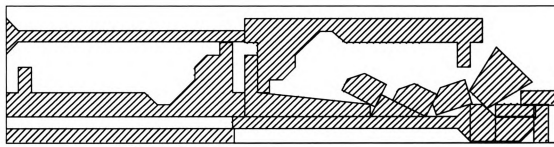

Figure 46. Problem 5: a=2, b=-2, c=-3, d=1

This happens because during the exploration of the search space, when the stock is shaped differently, different constraints may be violated and hence different areas of the search space would be explored.

On the whole, this heuristic has provided a basic method by which one can effectively use information provided by the part shapes for matching complementary features and providing an effective nested solution. The heuristic has been tried on the test cases with a limited variety of coefficients in its scoring function. Many solutions are repeated over

this search space. The thesis thus proves that feature matching is indeed a valid approach for solving this class of problems.

CHAPTER 7 - RECOMMENDATIONS

This thesis has endeavored to prove that feature matching may help in formulating an effective solution for the nesting problem. At this point, the heuristic is overly simplistic to provide good solutions for all classes of 2-dimensional packing and nesting problems.

We make the following recommendations for future research:

1. An instance of adjacent edges on a polygon is only the first and most basic type of *feature* information that can be extracted from a polygonal shape. More types of features need to be studied, classified and used. Edge features need not be made from adjacent edges alone. Non-adjacent edges may serve as features too. Edges from the convex hull of a non-convex polygonal shape may serve to make important features.
2. Curves need to be studied and classified as features. Information like curvature and arc length could be used effectively in this heuristic.
3. Methods need to be developed which would be able to *match up* two or more features together and decide how *complementary* they are. The heuristic would need to be able to compare different instances of matches between similar and dissimilar features.
4. The heuristic needs to be able to distinguish between *significant* and *insignificant* features. Often small features (usually small notches) which normally would not influence the final solution are evaluated as well. This causes a big computing

overhead with minimal contribution towards the solution. It would help in terms of computing time if the heuristic was able to distinguish between these types of features and optionally ignore them.

5. It is necessary to improve current geometric algorithms in terms of accuracy and speed of execution. Routines like *polygonal containment* are the major consumers of computing power and time in this heuristic. Additional efficient geometric algorithms would need to be developed as well for dealing with the other types of features.
6. The heuristic needs to be tried with a wide variety of coefficient combinations in the scoring function to be able to provide solutions that are good from the application point of view. Additional parameters could be incorporated into the scoring functions as well. For example, if the height of a nested solution is important, a factor penalizing the layout depending upon the highest vertex of the current part may be added. Genetic algorithms and evolutionary programming methods would be very useful in studying and tuning such factors in the heuristic possibly resulting in a major increase in its effectiveness.
7. Flipping of parts would vastly increase the solution space in which the heuristic looks for possible solutions.
8. It would help in particular classes of nesting and packing problems if one could place constraints on parts which could be translated, rotated and flipped. For example, the garment and upholstery industry requires parts to follow a certain orientation.
9. The heuristic does not distinguish between identical parts and performs redundant operations on identical parts that are yet to be nested. Eliminating such redundancies

would go a long way in improving the speed performance of the heuristic when dealing with problems which tend to have repeating parts.

10. By classifying and adding algorithms for handling 3-dimensional features the heuristic could be extended to be able to handle 3-dimensional packing and nesting problems as well.
11. The heuristic is capable of evaluating part features against *multiple* stock features. Methods of selecting multiple stock features should be studied and evaluated for better packing. Although the time of execution of the heuristic would increase linearly with the number of target stock features, it might result in better nesting outputs.
12. The heuristic performs overlap checks when it performs an alignment. In the current implementation, the heuristic slides the part in units of a 50^{th} of the length of the target feature side until it finds no overlap. It would be a great speed improvement if a procedure could be devised by means of which the part would be slid in discrete units depending upon the part and the stock geometry, thereby potentially greatly reducing the number of overlap computations performed.
13. The heuristic could place a *threshold* value as an acceptance criterion on the scoring function which would enable the heuristic to possibly improve its quality of nest by rejecting plausible, but extremely expensive part configurations in lieu of part configurations against other target stock features. This would work very well in conjunction with recommendation #11 above.

Bibliography

BIBLIOGRAPHY

[1] Adamowicz, M., and Albano, A. (1976), "Nesting two dimensional shapes in rectangular modules", *Computer Aided Design* 8/1, 27-33.

[2] Albano, A., and Sapuppo, G. (1980), "Optimal allocation of two-dimensional irregular shapes using heuristic search methods", *IEEE Transactions on Systems, Man, and Cybernetics* 10/5, 242-248.

[3] Amaral, C., Bernardo, J. and Jorge, J. (1990), "Marker making using automatic placement of irregular shapes for the garment industry", *Computers and Graphics* 14/1, 41-46.

[4] Batishchev, D. (1996), "Dense Arrangement of Objects Having Different Sizes at a Plan Using Genetic Algorithms", *Scientific Research Report - Nizhegorodsky State University,* Department of Computer Science and New Information Technologies.

[5] Böhme, D., and Graham, A. (1979), "Practical experiences with semi-automatic and partnesting methods", in: C. Kuo, K. J./ MacCallum and T. J. Williams (eds.), *Computer Applications in the Automation of Shipyard Operation and Ship Design III*, North-Holland, Amsterdam, 213-218.

[6] Dağli, Chan H., and Tatoğlu, M. Y. (1987), "An approach to two-dimensional cutting stock problems", *International Journal of Production Research* 25/2, 175-190.

[7] Dori, D., and Ben-Bassat, M. (1984), "Efficient nesting of congruent convex figures", *Communications of the ACM* 27/3, 228-235.

[8] Dowsland, K. A., and Dowsland, W. B. (1993), "Heuristic approaches to irregular cutting problems", Working Paper EBMS/1993/13, European Business Management School, UC Swansea, UK.

[9] Dowsland, K. A., and Dowsland, W. B. (1995), "Solution approaches to irregular nesting problems", *European Journal of Operational Research* 84, 506-521.

[10] Freeman, H., and Shapira, R. (1975), "Determining the minimum area encasing rectangle for an arbitrary closed curve", *Communications of the ACM* 81/7, 409-413.

[11] Fujita Kikuo, Shinsuke Akagi, Noriyasu Hirokawa (1993), "Hybrid approach for optimal nesting using a genetic algorithm and a local minimization algorithm", *Advances in Design Automation - ASME* 65/1, 477-484.

[12] Han, G-C, and Na, S-J (1996), "Two stage approach for nesting in two-dimensional cutting problems using neural-network and simulated annealing", *Proceedings of the Institution of Mechanical Engineers* 210, 509-519.

[13] Ismail, H. S., and Hon, K. K. B. (1995), "The nesting of two dimensional shapes using genetic algorithms", *Proceedings of the Institution of Mechanical Engineers* 209, 115-124.

[14] Karoupi, F., and Loftus, M. (1991), "Accommodating diverse shapes within hexagonal pavers", *International Journal of Production Research* 29/9, 1507-1519.

[15] Kröger, B. (1995), "Guillotinable bin packing: A genetic approach", *European Journal of Operational Research* 84, 645-661.

[16] Lamousin, H. J., Waggenspack, W. N. Jr., and Dobson, G. T., "Nesting of complex 2D parts within Irregular Boundaries", *Journal of Manufacturing Science & Engineering* 118, 615-622.

[17] Qu, W., and Sanders, J. L. (1987), "A nesting algorithm for irregular parts and factors affecting trim losses", *International Journal of Production Research* 25/3, 381-397.

[18] Glassner, Andrews A. (1990), "Graphics Gems", *Academic Press, Inc.*

[19] Arvo, James. (1991), "Graphics Gems II", *Academic Press, Inc.*

Appendices

Appendix A
Some geometric functions in detail

APPENDIX A – SOME GEOMETRIC FUNCTIONS IN DETAIL

POINTCLASS

`operator =`

Assigns both X and Y values of the source point to the target point.

`operator ==`

Returns true only if both X and Y values of the points being compared are within 10^{-6} of each other.

POLYGONCLASS

`contains (double x, double y)`

1. Loops through all the vertices of this polygon and returns an 'ON_VERTEX' flag if any vertex has a value of X and Y within 10^{-6} of the argument point.
2. Checks if the point lies on any edge of the polygon (by comparing distances between the point, and the current vertex and next vertex of this polygon). If the point lies on an edge, the function returns an 'ON_EDGE' flag.
3. Returns an 'INSIDE' flag if a ray from the point intersects the edges of the polygon an odd number of times. Otherwise it returns an 'OUTSIDE' flag.

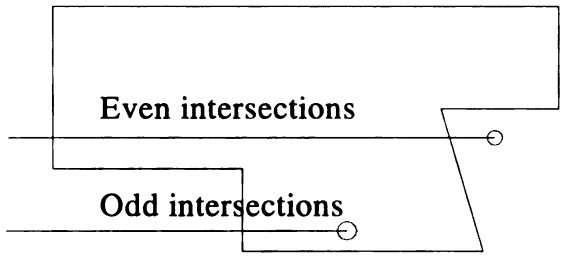

Figure 47. Point containment

`contains (PolygonClass p)`

1. Loops through all the vertices of polygon *p* and returns false if any vertex lies outside this polygon.
2. Loops through all the vertices of this polygon and returns false if any vertex of this polygon lies inside polygon *p*.
3. Loops through all the edges of this polygon and checks it for intersections with all the edges on the polygon *p*. If any intersection occurs on a point that is not a vertex of either polygon, the function returns a false.
4. Generates a point internal to polygon *p* and returns a false if this point does not lie inside this polygon.
5. It divides each edge of polygon *p* into 50 points and exits with a false value if any of these points along the edge of polygon *p* lies outside this polygon.

`getAnyInternalPoint()`

For each vertex of this polygon, it generates 8 points 45° apart around this point. The values of the x and y co-ordinates of each vertex are varied by $10^{-2}(\text{Area})^{0.5}$ to obtain these points. Each point is checked for containment within this polygon and the first point completely within this polygon is returned.

StockClass

leftShadow (PartClass p)

1. Selects the vertices of the part *p* with the highest and lowest y co-ordinates. If there is more than one such vertex, the vertex with the smallest x co-ordinate is selected.
2. Projects a ray from both these vertices to each edge of the stock. If the normal to the edge does not have a positive component in the direction of the x-axis, it is ignored. Else, the length of the intersecting segment of this ray is noted. The smallest segment forms part of the upper edge of the shadow polygon when the source of the ray is the upper selected vertex of *p*. Similarly the smallest such segment forms part of the bottom edge of the shadow polygon.
3. All vertices on the stock lying inside these two points of intersection in the counter-clockwise sense, the actual points of intersection and all the points on *p* lying within the two selected vertices in the clockwise sense form the shadow polygon. The area of this polygon is returned by this function.

bottomShadow(PartClass p)

This function uses the same logic as the above function. The only difference is that all rays and edge-normals are projected along the y-axis.

updateProfile(PolygonClass p)

This function uses a similar logic as the above two functions. It eliminates all stock vertices lying between the first and last points of contact of the polygon *p* against this stock. It then includes in the eliminated section, all the vertices of *p* that lie inside these two points of contact in the clock-wise sense.

Appendix B
Source code

APPENDIX B – SOURCE CODE

FILE : DXF.HPP

```
status DXFin (PartClass*, const int, StockClass*, const int,
              const char*);
status getPolylinesInFile(const char*, int&, int&);
status DXFout (const PartClass*, const int,
               const StockClass*, const int, const char*);
status DXFout (const PolygonClass*, const int,
               const char**, const PointArray&,
               const char*, const boolean);
status DXFout (const PartClass*, const int, const StockClass&,
               const PointArray&, const char*, const PointClass&,
               const char*);
status dxfOut (const char**, const PointArray&, const double = 0.3,
               FILE* = stdout);
status dxfOut (const char*, const PointClass&, const double = 0.3,
               FILE* = stdout);
```

FILE : GEOMETRY.HPP

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>

// CONSTANTS AND TYPE DECLARATIONS
const double PI    =  3.14159265358979323846;
const double DELTA =  1.0e-7;

enum boolean { FALSE = 0, TRUE = 1 };

const int red       = 1;
const int yellow    = 2;
const int green     = 3;
const int cyan      = 4;
const int blue      = 5;
const int magenta   = 6;
const int white     = 7;
```

```
const int gray      = 8;

// Return type for some Nesting functions
enum status
{
    NestCONSTRAINED = -1,
    NestFAILED      = 0,
    NestOK          = 1
};

// Return type for some other Nesting functions
enum position
{
    OUTSIDE     = -2,
    ON_VERTEX   = -1,
    ON_EDGE     = 0,
    INSIDE      = 1,
    TO_LEFT     = 2,
    TO_RIGHT    = 3,
    ABOVE       = 4,
    BELOW       = 5
};
// ------------------------------------------
// Forward declarations of classes
class PointClass;
class PointArray;
class PolygonClass;
class PartClass;
class StockClass;
class FeatureClass;
class FeatureArray;
// ------------------------------------------
/* BEGIN CLASS PointClass */

/*  Defines a 2-dimensional point with 2 private variables containing
    the X and Y locations of the point. These fields are private in
    order to encapsulate the vertex data within the Polygon in which
    they have been defined and to prevent individual polygon instances
    from accidentally/deliberately manipulating the vertex data of
    other polygonal instances.
*/

class PointClass
{
    private:
        // VARIABLES
        // The point location.
        double X, Y;

    public:

        // CONSTRUCTORS
        // Initialize this with 2 doubles with their defaults set to
        //  0.0, 0.0
        PointClass (const double = 0.0, const double = 0.0);
        // Initialize this with the geometric location of another
```

```
        //  point.
        PointClass (const PointClass&);

        // DESTRUCTOR
        ~PointClass ();

        // OPERATORS
        // Copy assignment operator
        PointClass& operator = (const PointClass&);
        // Equality operator: Checks if this point has equal values
        //  for BOTH the X & Y fields as compared to the argument
        //  point.
        boolean operator == (const PointClass&) const;


        // METHODS
        // Returns the X and Y values of this point resply.
        double getX (void) const;
        double getY (void) const;
        // Sets this point to contain the same values as the two
        //  arguments
        void set (const double, const double);
        // Provided just as a logical extension of the above function
        //  preferably use the overloaded assignment operator
        void set (const PointClass&);
        void dxfOut (FILE* = stdout, const double = 0.1) const;

        // DEBUG METHODS
        // Prints the X & Y values and the memory location of this
        //  point. if the boolean argument is != FALSE. Printing is
        //  done either  to the specified FILE pointer or to stdout
        //  (default).
        void print (boolean = FALSE, FILE* = stdout) const;

    friend class PointArray;

}; /* END CLASS PointClass*/
//---------------------------------------------------------------------
/*  BEGIN CLASS PointArray */
class PointArray
{
    private:
        PointClass *Point;
        unsigned int Length;

    public:
        PointArray (void);
        ~PointArray ();
        unsigned int getLength(void) const;
        PointClass& getPoint(unsigned int) const;
        status add (const PointClass&);
        status add (double, double);
        status remove (const PointClass&);
        status remove (double, double);
        status remove (unsigned int);
        void removeAll (void);
```

```
        boolean contains(const PointClass&) const;
        void dxfOut (FILE* = stdout, const double = 0.1) const;
        void print (boolean = FALSE, FILE* = stdout) const;
};
/*  END CLASS PointArray */
//---------------------------------------------------------------------
/*  BEGIN CLASS FeatureClass */
/*  At this time, this class represents the angular feature of two
    edges of a part. It may be used later to represent an (abstract?)
    generic feature which may be used as the base class for different
    types of features.
*/
class FeatureClass
{
    private:
        // A pointer to a constant PointClass... so that FeatureClass
        //  can't modify the contents of the PointClass instance its
        //  pointing at.
        // A major problem in the code right now is handling FLIPping
        //  the part. Since flipping inherently reverses the order
        //  of the vertices, all features pointing to vertices in that
        //  part need to be reinitialized.
        // One solution to this problem is by using linked lists
        //  instead of an array of points.
        const PointClass* PointPointer[4];
        double Angle;
        PolygonClass* Owner;


public:
        FeatureClass (void);
        void construct(const FeatureClass&);
        void construct (PolygonClass&, const PointClass&,
            const PointClass&, const PointClass&, const PointClass&,
            double);
        void construct (PolygonClass&, const PointClass&,
            const PointClass&, const PointClass&, const PointClass&);
        double getAngle(void) const;
        void print (boolean = FALSE, FILE* = stdout) const;
        PolygonClass* getOwner(void) const;
        PointClass getVertexValue(unsigned int) const;
        boolean operator == (const FeatureClass&) const;

    friend class FeatureArray;
    friend class StockClass;
    friend status evaluateMatches (const FeatureArray&, const
        FeatureArray&, FeatureClass&, FeatureClass&, unsigned int*,
        double*);
    friend status align(const int, const FeatureClass&,
        status&, status&, const int, const FeatureClass&);
    friend status selectStockFeatures(FeatureArray&,
        const PointArray&, StockClass&);
    friend double calculateScore(const PartClass&, const StockClass&,
        const FeatureClass&, const FeatureClass&);

};
```

```
/*  END CLASS FeatureClass */
//--------------------------------------------------------------------
/*  BEGIN CLASS FeatureArray */
class FeatureArray
{
    private:
        unsigned int Length;
        FeatureClass* Feature;

    public:
        FeatureArray (void);
        ~FeatureArray ();
        unsigned int getLength(void) const;
        FeatureClass& getFeature(unsigned int) const;
        status add (PolygonClass&, const PointClass&,
            const PointClass&, const PointClass&, const PointClass&,
            double);
        status add (PolygonClass&, const PointClass&,
            const PointClass&, const PointClass&, const PointClass&);
        status add (const FeatureClass&);
        status remove (const PolygonClass&);
        status remove (unsigned int);
        status remove (const FeatureClass&);
        void removeAll (void);
        void print (boolean = FALSE, FILE* = stdout) const;

    friend status evaluateMatches (const FeatureArray&, const
        FeatureArray&,
        FeatureClass&, FeatureClass&, unsigned int*, double*);
};
/*  END CLASS FeatureArray */
//--------------------------------------------------------------------
/*  BEGIN CLASS PolygonClass */
/*  It must be defined in the counter clockwise direction.
    It must be non-self intersecting.
    It must have only straight edges... something taken care of by
    the Part creator while reading in the data from the DXF FILE.
*/
class PolygonClass
{
    protected:
        // VARIABLES
        // Contains the number of vertices of this Polygon
        int         Number_Of_Vertices;
        // The Vertex list (dynamic array of doubles) and the values
        //  of their corresponding angles. The memory for the vertices
        //  and the angles are allocated in the PolygonClass
        //  construct()  method and deallocated in the PolygonClass
        //  destructor.
        PointClass* Vertex; // represents the actual location
        double*     Angle;  // represents the internal or external
        double*     Length; // represents the lengths of the sides
        // A generic 30 character name for ID checking.. last 2 for \n
        //  &  \0
        char        Name[32];
        // Area of the polygon.. computed once in the construct()
```

```
        //  method.
        double      Area;
        // Color
        int         Color;

        // CONSTRUCTORS
        // A void constructor array to be dynamically allocated with
        //  'new'
        PolygonClass (void);
        // Explicit constructor to initialize this polygon.
        void construct (const int, const PointClass*);

        // DESTRUCTOR
        ~PolygonClass ();

    public:
        // METHODS
        void setColor (int color);
        // Sets the Name field of this polygon.
        void set_name (const char*);
        // Sets the char* passed in the argument to contain the
        //  contents of the Name field.
        char *get_name (char*) const;

        inline double get_area () const
        {
            return Area;
        }
        // Returns a point which is inside this polygon.
        PointClass getAnyInternalPoint(void) const;
        // Checks if this polygon contains the specified point.
        position contains(const double, const double) const;
        position contains(const PointClass&) const;
        // Checks if this polygon contains the argument polygon.
        boolean contains(const PolygonClass&) const;
        // Returns the number of vertices this polygon has.
        int getNumber_Of_Vertices (void) const;
        boolean doesAnyEdgeIntersect(const PolygonClass&) const;
        void dxfOut (FILE* = stdout) const;

        // DEBUG METHODS
        // Prints out the list of vertices of this polygon to the
        //  specified FILE pointer, along with the memory address of
        //  each PointClass
        // instance if the boolean argument != FALSE.
        virtual void print (boolean = FALSE, FILE* = stdout) const;

    // Extracts all the 'corner features' of the specified polygon and
    // stores it in the specified FeatureArray
    friend status cornerFeatures(PolygonClass& , FeatureArray& );
    friend status DXFout (const PolygonClass*, const int, const
        char**, const PointArray&, const char*, const boolean = TRUE);
    friend double area(const PointClass*, const int, const char*);
    friend class StockClass;
}; /* END CLASS PolygonClass*/
//---------------------------------------------------------------------
```

```
/*  BEGIN CLASS PartClass */
/*  The class PartClass is derived from the virtual base class,
    PolygonClass defined above. This is to keep in mind that there
    Might be future specific parts derived from this class like
    rectangles, triangles and so on.
*/

class PartClass : public PolygonClass
{
    private:
        // VARIABLES
        // The number of PartClass instances.
        static int Number_Of_Parts;
        // Keeps track of the convex hull of this part.
        boolean   *Is_Convex_Hull;
        // The bounding rectangle in the Global Co-ordinate System.
        //  This would need to be dynamically recalculated everytime a
        //  movement operation is performed on the part. Since the
        //  stock  is stationary this method is provided only for the
        //  PartClass  class.
        PointClass Lower_Left_Point, Upper_Right_Point;
        // The Minimum Enclosing Rectangle (MER).. this too needs to
        //  be transformed everytime any kind of movement is performed
        //  on this part.
        PointClass M_E_R[4];
        // Constraint flags which determine if a part may be flipped,
        // rotated or translated.
        boolean canFlip,
              canRotate,
              canTranslate;

        // PRIVATE METHODS
        // Sets the enclosing rectangle during construction and during
        //  any kind of 'move' command on this part, in the Global
        //  Coordinate System (GCS).
        void set_bounds (void);
        // Sets the angles of the Part during construction only!!
        void set_angles (void);
        // Sets the convex hull flags after determining the convex
        //  hull  of this part.
        void mark_convex_hull (const PointClass*, const int,
            boolean**);
        // Sets the minimum enclosing rectangle for this part.
        void set_M_E_R (void);

    public:
        // CONSTRUCTORS
        // A void constructor taking no arguments for allowing a
        //  PartClass
        // array to be dynamically allocated with 'new'.
        PartClass (void);
        // Initializes this part with the number of vertices being
        //  formed and an array of vertices with at least that many
        //  PointClass instances representing the vertices.
        void construct (const int, const PointClass*);
```

```
        // DESTRUCTOR
        ~PartClass (void);

        // METHODS
        // The following 'move' functions also handle the modifying of
        //  the vertex locations for the M_E_R.
        // Rotates this part by the angle specified about the
        //  reference  point.
        status rotate (const double, const PointClass&);
        status rotate (const double, const double, const double);
        // Translates the part by the vector specified by the first
        //  point and the second.
        status translate (const PointClass&, const PointClass&);
        status translate (const double, const double,
            const double, const double);
        // Flips the part about the axis defined by the 2 points.
        status flip (const PointClass&, const PointClass&);
        status flip (const double, const double,
            const double, const double);
        // Returns a count of the Number_Of_Parts.
        int static how_many (void);

        // DEBUG METHODS
        // shows details about the entire part. If the argument is
        // non-zero, prints out the memory  locations of the
        // individual components too
        virtual void print (boolean = FALSE, FILE* = stdout) const;

    // FRIEND declarations.
    // The global function for writing out part data.
    friend status DXFout (const PartClass*, const int,
        const StockClass*, const int, const char*);
    friend status DXFout (const PartClass*, const int,
        const StockClass&, const PointArray&, const char*,
        const PointClass&, const char*);
    // Making StockClass a friend of the PartClass so that the Stock
    //  has complete access to the variables of Parts... especially
    //  the vertex data.
    friend class StockClass;
    friend int evaluate(const char*, FILE*, int, int, int, int);
};
/*  END CLASS PartClass */
//--------------------------------------------------------------------
/*  BEGIN CLASS StockClass */
/*  The "Stock" declaration and definition. Also to include the
    "voids" that open up while Nesting progresses
*/

class StockClass : public PolygonClass
{
    protected:
        // The number of StockClass instances.
        static int Number_Of_Stocks;

    public:
        // CONSTRUCTORS
```

```
        // A void constructor taking no arguments for allowing a
        //  StockClass array to be dynamically allocated with 'new'.
        StockClass (void);
        // Initializes this stock with the number of vertices being
        //  formed and an array of vertices with at least that many
        //  PointClass instances representing the vertices.
        void construct (const int, const PointClass*);

        // DESTRUCTOR
        ~StockClass (void);

        // METHODS
        // Returns a count of the Number_Of_Stocks.
        int static how_many (void);
        virtual void print (boolean = FALSE, FILE* = stdout) const;
        // Returns the area of the shadow cast by the argument part on
        //  the left wall of the stock.
        double leftShadow (const PartClass&) const;
        // Returns the area of the shadow cast by the argument part on
        //  the bottom wall of the stock.
        double underShadow (const PartClass&) const;
        status updateProfile (const PolygonClass&);

    // FRIEND declarations.
    // The global function for writing out stock data.
    friend status DXFout (const PartClass*, const int,
        const StockClass*, const int, const char*);
    friend status DXFout (const PartClass*, const int,
        const StockClass&, const PointArray&, const char*,
        const PointClass&, const char*);
    friend status selectStockFeatures(FeatureArray&,
        const PointArray&, StockClass&);
    friend status selectStockFeatures(FeatureArray&, const
        PointArray&, StockClass&, const double, const double);
};
/*  END CLASS PartClass */
//---------------------------------------------------------------------
/*  Public Utility Functions
*/
// Return the angle of the vector wrt +ve x-axis
double get_angle (const PointClass&, const PointClass&);
// Returns angle swept by vector(2,3) to align vector(2,1) in the CCW
//  sense
double get_angle (const PointClass&, const PointClass&, const
    PointClass&);
// Returns the distance between the two points
double get_distance (const PointClass&, const PointClass&);
double get_distance (const double, const double, const double, const
double);
// Rotates by the specified angle, the number of points, about the
//  specified point, from an input array of points to an output array
//  of  points. The output array may be specified to be the same input
//  array as the function uses a local copy of the input points.
void rotate (const double, const int, const PointClass&,
             const PointClass*, PointClass*);
// Gets the bounding rectangle of specified no. of points, in an array
```

```
// of input points, returning two points in the output point array
// and the area of the bounding box in the last argument passed
void get_bounds (const int, const PointClass*, PointClass*, double&);
// Set of Line segment intersection routines adapted from
// "Intersection of line segments", Mukesh Prasad, Graphics Gems - 2
// The first two functions return TRUE if there's an intersection.
// The next two functions return TRUE on an intersection and return
//  the point of intersection in the fifth reference PointClass
//  argument.
boolean intersects (double, double, double, double,
                    double, double, double, double);
boolean intersects (double, double, double, double, double, double,
                    double, double, double&, double&);
boolean intersects (const PointClass&, const PointClass&,
                    const PointClass&, const PointClass&);
boolean intersects (const PointClass&, const PointClass&,
                    const PointClass&, const PointClass&,
                    PointClass&);
// Returns TRUE if both doubles have the same sign; FALSE otherwise
boolean sameSign(double , double);
// Toggles a struct boolean reference wrt its current value
boolean toggle(boolean&);
// Returns the area of a polygon defined by the array of points
double area (const PointClass*, const int, const char*);

// Returns a double value of the length of contact of 2 line segments
// only if the lines are collinear
double contactLength (const PointClass&, const PointClass&,
                      const PointClass&, const PointClass&);
double contactLength(double, double, double, double,
                     double, double, double, double);

boolean filterRedundantVertices(int*, PointClass**);

inline double toDegrees(double radians)
{
    return radians/0.01745329251994;
}

inline boolean areParallel(const PointClass& A, const PointClass& B,
    const PointClass& C, const PointClass& D)
{
    double angleAB = ::get_angle(A, B);
    double angleCD = ::get_angle(C, D);

    if(fabs(angleAB - angleCD) < DELTA)
        return TRUE;

    angleAB += PI;
    if(angleAB > 2.0*PI)
        angleAB -= 2.0*PI;

    if(fabs(angleAB - angleCD) < DELTA)
        return TRUE;

    return FALSE;
```

```
}
```

FILE : NEST.HPP

```
/*  File containing all the prototypes for the actual nesting
    routines. The testing functions which make the actual part
    movements and new feature extractions etc.
*/
const double BADSCORE = -10000000000.0;

// Extracts all the 'corner features' of the specified polygon and
//  stores it in the specified FeatureArray... friend to PolygonClass
status cornerFeatures(PolygonClass&, FeatureArray&);

// Extracts all the 'convex hull features' of the specified polygon
//  and  stores it in the specified FeatureArray... friend to
//  PolygonClass
status convexHullFeatures(const PolygonClass&, FeatureArray&);

// Selects the current stock feature for matching
status selectStockFeatures(FeatureArray&, const PointArray&,
                           StockClass&);
status selectStockFeatures(FeatureArray&, const PointArray&,
                           StockClass&, const double, const double);

status align(const int, const FeatureClass&, status&, status&,
             const int, const FeatureClass&);
double calculateScore (const PartClass&, const StockClass&,
    const FeatureClass&, const FeatureClass&);
```

FILE : DXF.CPP

```
#include <strstrea.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <conio.h>
#include "geometry.hpp"
#include "dxf.hpp"

const MaxLineLength = 200; // a line isn't usually longer than this
// Keeps track of the current file name... a call to the
//  getPolylinesInFile() is necessary before the DXFin() function
```

```
//  as the static variable below is initialized in that function

// Routine to loop through the input file and get the number of
//  polylines present .... this function is always to be called
//  before a DXFIN
status getPolylinesInFile(const char* input_DXF_file_name,
                          int &number_of_stocks, int &number_of_parts)
{
    char file_name[MaxLineLength];
    strcpy(file_name, input_DXF_file_name);

    ifstream in_DXFile(file_name, ios::binary | ios::nocreate);
    if(!in_DXFile)
    {
        cerr << "\aCould not open input file \""
            << file_name << "\"" << endl;
        return NestFAILED;
    }

    char inbuf[MaxLineLength];

    // initialize
    number_of_stocks = 0;
    number_of_parts = 0;

    // Count POLYLINES
    // get the current line into 'buf' reading 'MaxLineLength'
    //  characters from the input file stream
    int line_count = 0;
    while (in_DXFile.getline(inbuf, MaxLineLength))
    {
        line_count++;
        char *found;
        // set found to first character of POLYLINE in the file
        if(found = strstr(inbuf, "POLYLINE"))
        {
            // check color tag for red (color # 1)
            while(in_DXFile.getline(inbuf, MaxLineLength))
            {
                // found dummy point tag before red color code
                //  .. hence polyline is a part
                if(found = strstr(inbuf, " 10"))
                {
                    number_of_parts++;
                    break;
                }

                if(found = strstr(inbuf, " 62")) // found color tag
                {
                    // next line contains code 1 for red
                    in_DXFile.getline(inbuf, MaxLineLength);
                    if
                        (
                        (found = strstr(inbuf, "1"))
                        ||
                        (found = strstr(inbuf, "     1"))
```

```
                        )
                        number_of_stocks++;

                    // any other color stands for a part
                    else
                        number_of_parts++;

                    break;
                }
            }
        }
    }
    in_DXFile.close();
    return NestOK;
}

status DXFin (PartClass *PartArray, const int number_of_parts,
              StockClass* StockArray, const int number_of_stocks,
              const char* fileName)
{
    char        file_name[MaxLineLength];
    int         *vertices_in_polyline;
    char        inbuf[MaxLineLength];
    int         line_count = 0;
    PointClass  *point_array;

    strcpy(file_name, fileName);

    // each element of this array will hold the value of the
    //  number of vertices each polyline has...
    //  vertices_in_polyline[0] = number of vertices the first
    //  polyline has etc etc.
    vertices_in_polyline =
        new int[number_of_parts + number_of_stocks];

    // Open file in a stream
    ifstream in_DXFile(file_name, ios::binary | ios::nocreate);
    if(!in_DXFile)
    {
        cerr << "\aCould not open input file \""
            << file_name << "\"" << endl;
        return NestFAILED;
    }


    // Count how many vertices each polyline has
    int polyline_index = 0;
    while (in_DXFile.getline(inbuf, MaxLineLength))
    {
        // get out of function if
        if (polyline_index >= (number_of_parts+number_of_stocks))
            break;

        char *found;
        if(found = strstr(inbuf, "POLYLINE"))
        {
```

```
        // initialize to 0; increment every time a VERTEX
        //  tag is found
        vertices_in_polyline[polyline_index] = 0;
        while (in_DXFile.getline(inbuf, MaxLineLength))
        {
            if(found = strstr(inbuf, "SEQEND"))
                break; // get out of inner loop

            if(found = strstr(inbuf, "VERTEX"))
            {
                // increment vertex count
                (vertices_in_polyline[polyline_index])++;

                while(in_DXFile.getline(inbuf, MaxLineLength))
                {
                    if(found = strstr(inbuf, " 42"))
                    {
                        // use bulge handler to increment
                        // # of points here
                        break;
                    }
                    // if no bulge..
                    if(found = strstr(inbuf, "  0"))
                        break;
                }
                continue;
            }
        }
        polyline_index++;
    }
}
in_DXFile.close();

// Create the parts & the stocks
in_DXFile.open(file_name, ios::binary | ios::nocreate);
line_count = 0; polyline_index = 0;
int stock_index = 0, part_index = 0;

while (in_DXFile.getline(inbuf, MaxLineLength))
{
    line_count++;
    char *found;
    if(found = strstr(inbuf, "POLYLINE"))
    {
        boolean skip_color_check = FALSE,
            is_this_a_stock = FALSE;
        // allocate memory to hold vertex list
        point_array =
            new
            PointClass[(vertices_in_polyline[polyline_index])];
        int vertex_index = 0;
        while (in_DXFile.getline(inbuf, MaxLineLength))
        {
            double x, y, bulge;
            line_count++;
            // ..exit this 'while' loop and
```

```
//  'if POLYLINE' condition if SEQEND found
if(skip_color_check == FALSE)
{
    if(found = strstr(inbuf, " 62"))
    {
        skip_color_check = TRUE;
        in_DXFile.getline(inbuf, MaxLineLength);
        if
            (
            (found = strstr(inbuf, "1"))
            ||
            (found = strstr(inbuf, "     1"))
            )
            is_this_a_stock = TRUE;
    }
}

if(found = strstr(inbuf, "SEQEND"))
    break;

// ..continue this 'while' loop and if VERTEX found
if(found = strstr(inbuf, "VERTEX"))
{
    skip_color_check = TRUE;
    do // find the x location
    {
        in_DXFile.getline(inbuf, MaxLineLength);
        line_count++;
        found = strstr(inbuf, " 10");
    } while(!found);
    in_DXFile.getline(inbuf, MaxLineLength);
    line_count++;
    x = (double) atof(inbuf);

    do // find the y location
    {
        in_DXFile.getline(inbuf, MaxLineLength);
        line_count++;
        found = strstr(inbuf, " 20");
    } while(!found);
    in_DXFile.getline(inbuf, MaxLineLength);
    line_count++;
    y = (double) atof(inbuf);

    do // find the OPTIONAL bulge factor
    {
        in_DXFile.getline(inbuf, MaxLineLength);
        line_count++;
        found = strstr(inbuf, " 42");
        if(found) // bulge found
        {
            in_DXFile.getline(inbuf, MaxLineLength);
            line_count++;
            bulge = (double) atof(inbuf);
        }
        // if no bulge, will find "  0" and stop loop
```

```
                    else
                    {
                        found = strstr(inbuf, "  0");
                        bulge = NULL;
                    }
                } while(!found);
                point_array[vertex_index].set(x,y);
                vertex_index++;
                continue;
            }
        }

        if(is_this_a_stock)
        {
            StockArray[stock_index].construct(
                (vertices_in_polyline[polyline_index]),
                point_array);
            char stockname[20] = "Stock ";
            char tmp[20];
            strcat(stockname, itoa(stock_index, tmp, 10));
            StockArray[stock_index].set_name(stockname);
            stock_index++;
        }

        else
        {
            PartArray[part_index].construct(
                (vertices_in_polyline[polyline_index]),
                point_array);
            char partname[20] = "Part ";
            char tmp[20];
            strcat(partname, itoa(part_index, tmp, 10));
            PartArray[part_index].set_name(partname);
            part_index++;
        }

        polyline_index++;
        delete [] point_array;
    }
}

in_DXFile.close();
delete [] vertices_in_polyline;
return NestOK;
}

status DXFout (const PartClass *PartArray, const int number_of_parts,
               const StockClass *StockArray, const int
               number_of_stocks, const char* out_file_name )
{
    // Check if the file exists and prompt for replacement
    ifstream check_exists_File(out_file_name, ios::binary |
        ios::nocreate);
    if(check_exists_File)
    {
        cout << "\aFile "<<out_file_name<<" exists! Overwrite? "<<
```

```
            flush;
        check_exists_File.close();
        char in = (char) getche(); cout << " " << flush;
        if(in != 'y')
        {
            if(in != 'Y')
            {
                cout << "\nNot writing DXF file "<<out_file_name <<
                    endl;
                return NestOK; // voluntary exit
            }
        }
    }

    ofstream out_DXFile(out_file_name);
    if(!out_DXFile)
    {
        cerr << "\aCould not open output file \""
            << out_file_name << "\"" << endl;
        return NestFAILED;
    }

    out_DXFile.precision(12);
    out_DXFile.setf(ios::showpoint);

    // Entities section
    out_DXFile << "  0\nSECTION\n  2\nENTITIES" << endl;

    // begin Parts
    for(int i = 0; i < number_of_parts; i++)
    {
        if(PartArray[i].Number_Of_Vertices <= 1)
            continue;
        // The Part data
        out_DXFile << "  0\nPOLYLINE" << endl  // Header
            <<"  8\nNest" << endl   // layer "Nest"
            <<"  62\n" << PartArray[i].Color << endl
            <<" 66\n     1" << endl  // vertices follow
            <<" 10\n0.0\n 20\n0.0\n 30\n0.0"<< endl // dummy points
            <<" 70\n     1" << endl;  // closed polyline
        for(int j = 0; j < PartArray[i].Number_Of_Vertices; j++)
        {
            out_DXFile << "  0\nVERTEX\n  8\nNest" << endl;
            // layer "Nest"
            out_DXFile << " 10" << endl;
            out_DXFile << PartArray[i].Vertex[j].getX() << endl;
            out_DXFile << " 20" << endl;
            out_DXFile << PartArray[i].Vertex[j].getY() << endl;
            out_DXFile << " 30" << endl;
            out_DXFile << "0.0" << endl; // told ya it's 2-D :)
        }
        out_DXFile << "  0\nSEQEND\n  8\nNest" << endl;
        // End of part data
}

    // begin Stocks
```

```
    for(i = 0; i < number_of_stocks; i++)
    {
        if(StockArray[i].Number_Of_Vertices <= 1)
            continue;
        out_DXFile << "  0\nPOLYLINE" << endl  // Header
            <<"  8\nNest" << endl   // layer "Nest"
            <<"  62\n" << StockArray[i].Color << endl  // color red
            <<" 66\n     1" << endl  // vertices follow
            <<" 10\n0.0\n 20\n0.0\n 30\n0.0"<< endl // dummy points
            <<" 70\n     1" << endl;  // closed polyline
        for(int j = 0; j < StockArray[i].Number_Of_Vertices; j++)
        {
            out_DXFile << "  0\nVERTEX\n  8\nNest" << endl;
            out_DXFile << " 10" << endl;
            out_DXFile << StockArray[i].Vertex[j].getX() << endl;
            out_DXFile << " 20" << endl;
            out_DXFile << StockArray[i].Vertex[j].getY() << endl;
            out_DXFile << " 30" << endl;
            out_DXFile << "0.0" << endl; // told ya it's 2-D :)
        }

        out_DXFile << "  0\nSEQEND\n  8\nNest" << endl;
    }

    // End of DXF file
    out_DXFile << "  0\nENDSEC\n  0\nEOF" << endl;
    out_DXFile.close();
    // cout << "DXF file \"" << out_file_name << "\" written" << endl;
    return NestOK;
}

status DXFout (const PolygonClass *PolygonArray,
               const int number,
               const char** strings,
               const PointArray& points,
               const char* out_file_name,
               const boolean checkFileExists)
{
    // Check if the file exists and prompt for replacement
    if(checkFileExists == TRUE)
    {
        ifstream check_exists_File(out_file_name, ios::binary |
            ios::nocreate);
        if(check_exists_File)
        {
            cout << "\aFile "<<out_file_name<<" exists! Overwrite? "<<
                flush;
            check_exists_File.close();
            char in = (char) getche(); cout << " " << flush;
            if(in != 'y')
            {
                if(in != 'Y')
                {
                    cout << "\nNot writing DXF file "<<out_file_name
                        << endl;
                    return NestOK; // voluntary exit
```

```
                }
            }
        }
    }

    FILE* outFile = fopen(out_file_name, "w");
    if(!outFile)
    {
        cerr << "\aCould not open output file \""
            << out_file_name << "\"" << endl;
        return NestFAILED;
    }

    fprintf(outFile, "  0\nSECTION\n  2\nENTITIES\n");

    for(int i = 0; i < number; i++)
        PolygonArray[i].dxfOut(outFile);

    dxfOut (strings, points, 0.3, outFile);

    // End of DXF file
    fprintf(outFile, "  0\nENDSEC\n  0\nEOF\n");
    fclose(outFile);
    return NestOK;
}

status DXFout (const PartClass* partArray,
               const int numberOfParts,
               const StockClass& stock,
               const PointArray& pointArray,
               const char* string,
               const PointClass& insertionPoint,
               const char* fileName)
{
    FILE* outFile = NULL;
    ifstream check_exists_File(fileName, ios::binary | ios::nocreate);
    if(check_exists_File)
    {
        cout << "\aFile "<<fileName<<" exists! Overwrite? "<<
            flush;
        check_exists_File.close();
        char in = (char) getche(); cout << " " << flush;
        if(in != 'y')
        {
            if(in != 'Y')
            {
                cout << "\nNot writing DXF file "<<fileName << endl;
                return NestOK; // voluntary exit
            }
        }
    }

    outFile = fopen(fileName, "w");
    if(outFile == NULL)
    {
        cout << "\aCould not open " << fileName << endl;
```

```
        return NestFAILED;
    }

    fprintf(outFile, "  0\nSECTION\n  2\nENTITIES\n");

    for(int i = 0; i < numberOfParts; i++)
        partArray[i].dxfOut(outFile);

    stock.dxfOut(outFile);

    pointArray.dxfOut(outFile);

    fprintf(outFile, "  0\nTEXT\n 10\n%f\n 20\n%f\n 40\n0.3\n",
        insertionPoint.getX(), insertionPoint.getY());
    fprintf(outFile, " 1\n%s\n 8\nNest\n", string);

    fprintf(outFile, "  0\nENDSEC\n  0\nEOF\n");

    fflush(outFile);
    fclose(outFile);
    return NestOK;
}

status dxfOut (const char** strings, const PointArray& points,
               const double letteringSize, FILE* fp)
{
    for(int i = 0; i < points.getLength(); i++)
    {
        fprintf(fp, "  0\nTEXT\n 10\n%f\n 20\n%f\n 40\n%d\n",
            points.getPoint(i).getX(), points.getPoint(i).getY(),
            letteringSize);
        fprintf(fp, " 1\n%s\n 8\nNest\n", strings[i]);
    }
    return NestOK;
}

status dxfOut (const char* string, const PointClass& point,
               const double letteringSize, FILE* fp)
{
    fprintf(fp, "  0\nTEXT\n 10\n%f\n 20\n%f\n 40\n%d\n",
        point.getX(), point.getY(), letteringSize);
    fprintf(fp, " 1\n%s\n 8\nNest\n", string);
    return NestOK;
}
```

FILE: GEOMETRY.CPP

```
/*  Standard Geometric Library for the Nesting Algorithm.*/
// PREPROCESSOR DIRECTIVES
#include <math.h>
#include <stdio.h>
#include <iostream.h>
```

```
#include <assert.h>
#include <string.h>
#include "geometry.hpp"
#include <conio.h>
#include <stdlib.h>
#include "dxf.hpp"

// initialize the static variables
int PartClass::Number_Of_Parts = 0;
int StockClass::Number_Of_Stocks = 0;

extern boolean      DEBUG;
int                 errorCount = 0;
extern    int       StockUpdateNumber;
extern  FILE*       errorFile;

PolygonClass *leftShadowPolygon = NULL;

// ---------------------------------------------
// BEGIN CLASS PointClass

// CONSTRUCTORS
// Deliberate constructor
PointClass::PointClass (const double a, const double b)
{
    X = a;  Y = b;
}
// Copy constructor
PointClass::PointClass (const PointClass &refPoint)
{
    X = refPoint.X;  Y = refPoint.Y;
}

// DESTRUCTOR
PointClass::~PointClass ()
{ // does nothing yet
}

// OPERATORS
PointClass& PointClass::operator = (const PointClass& refPoint)
{
    X = refPoint.X;
    Y = refPoint.Y;
    return *this;
}

boolean PointClass::operator == (const PointClass& refPoint) const
{
    // true only if BOTH values are the same
    if((fabs(X - refPoint.X)<DELTA) &&
       (fabs(Y - refPoint.Y)<DELTA))
        return TRUE;
    else return FALSE;
}

// METHODS
```

```
// return X value
double PointClass::getX (void) const
{
    return X;
}

// return Y value
double PointClass::getY (void) const
{
    return Y;
}

// set this Point to specified X & Y
void PointClass::set(const double newx, const double newy)
{
    X = newx; Y = newy;
}

// change this Point to values of refPoint...
void PointClass::set(const PointClass& refPoint)
{
    X = refPoint.X;  Y = refPoint.Y;
}

void PointClass::dxfOut (FILE* file, const double radius) const
{
    fprintf(file, "  0\nCIRCLE\n 10\n%f\n 20\n%f\n 40\n%f\n", X, Y,
        radius);
    fprintf(file, " 8\nNest\n");
    fflush(file);
}
// DEBUG METHODS

void PointClass::print (boolean address, FILE* fp) const
{
    fprintf(fp, " X= %8.4f  Y= %8.4f", X, Y);

    if(address)
        fprintf(fp, "  PointClass instance @ %u", this);
    fflush(fp);
}

// END CLASS PointClass
//---------------------------------------------------------------------
/*  BEGIN CLASS PointArray */
PointArray::PointArray (void)
{
    Length = 0;
    Point  = NULL;
}

PointArray::~PointArray ()
{
    delete [] Point;
}
```

```
unsigned int PointArray::getLength(void) const
{
    return Length;
}

PointClass& PointArray::getPoint(unsigned int index) const
{
    PointClass *point = new PointClass;
    if(index >= Length)
    {
        cout << "\aAlert: not that many points in array!\a" << endl;
        // potential memory leak
        return *point;
    }
    delete point;
    return Point[index];
}

// DISALLOW DUPLICATE POINTS
status PointArray::add (const PointClass& newPoint)
{
    unsigned int i;
    for(i = 0; i < Length; i++)
    {
        if(newPoint == Point[i])
        {
            cout << "\aTrying to add duplicate point " << flush;
            Point[i].print(); cout << endl;
            return NestFAILED;
        }
    }

    Length++;
    PointClass *newPoints = new PointClass[Length];
    for(i = 0; i < (Length-1); i++)
        newPoints[i] = Point[i];
    newPoints[Length-1] = newPoint;
    delete [] Point;
    Point = newPoints;
    return NestOK;
}

status PointArray::add (double x, double y)
{
    PointClass p(x, y);
    return this->add(p);
}

status PointArray::remove (const PointClass& p)
{
    unsigned int i, removeIndex;
    boolean proceed = FALSE;

    for(i = 0; i < Length; i++)
    {
        if(p == Point[i])
```

```
        {
            proceed = TRUE;
            removeIndex = i;
            break;
        }
    }

    if(proceed == FALSE)
    {
        cout << "Point does not exist in this array" << endl;
        return NestFAILED;
    }

    PointClass *newPoints = new PointClass[Length-1];

    for(i = 0; i < removeIndex; i++)
        newPoints[i] = Point[i];

    for(i = (removeIndex+1); i < Length; i++)
        newPoints[i-1] = Point[i];

    Length--;

    delete [] Point;
    Point = newPoints;

    return NestOK;

}

status PointArray::remove (double x, double y)
{
    PointClass p(x, y);
    return this->remove(p);
}

status PointArray::remove (unsigned int index)
{
    if(index > (Length-1))
        return NestFAILED;
    PointClass p = Point[index];

    return this->remove(p);
}

void PointArray::removeAll(void)
{
    if(Point != NULL)
        delete [] Point;
    Length = 0;
}

boolean PointArray::contains(const PointClass& point) const
{
    for(unsigned int i = 0; i < Length; i++)
    {
```

```
        if(point == Point[i])
            return TRUE;
    }

    return FALSE;
}

void PointArray::dxfOut (FILE* file, const double radius) const
{
    for(unsigned int i = 0; i < Length; i++)
        Point[i].dxfOut(file, radius);
}

void PointArray::print (boolean address, FILE* fp) const
{
    fprintf(fp, "Point Array of length %u", Length);
    if(address)
        fprintf(fp, "  @ %u", this);
    fprintf(fp, "\n====================================");
    fprintf(fp, "\n");
    for(unsigned int i = 0; i < Length; i++)
    {
        Point[i].print(address, fp);
        fprintf(fp, "\n");
    }
    fprintf(fp, "\n====================================\n");
    fflush(fp);
}

/* END CLASS PointArray */
//--------------------------------------------------------------------------
/*  BEGIN CLASS FeatureClass */

// CONSTRUCTORS
FeatureClass::FeatureClass (void)
{
    PointPointer[0] = NULL;
    PointPointer[1] = NULL;
    PointPointer[2] = NULL;
    PointPointer[3] = NULL;
    Angle = 0.0;
    Owner = NULL;
}

void FeatureClass::construct(const FeatureClass& f)
{
    if(this == &f)
        return;

    Owner = f.Owner;
    PointPointer[0] = f.PointPointer[0];
    PointPointer[1] = f.PointPointer[1];
    PointPointer[2] = f.PointPointer[2];
    PointPointer[3] = f.PointPointer[3];
    Angle = f.Angle;
}
```

```
void FeatureClass::construct(PolygonClass& owner,
                             const PointClass& one,
                             const PointClass& two,
                             const PointClass& three,
                             const PointClass& four,
                             double angle)
{
    Owner = &owner;
    PointPointer[0] = &one;
    PointPointer[1] = &two;
    PointPointer[2] = &three;
    PointPointer[3] = &four;
    Angle = angle;
}

void FeatureClass::construct(PolygonClass& owner,
                             const PointClass& one,
                             const PointClass& two,
                             const PointClass& three,
                             const PointClass& four)
{
    double angle = ::get_angle(two, one) - ::get_angle(three, four);

    while(angle > (2.0*PI))
        angle -= 2.0*PI;

    while(angle < 0.0)
        angle += 2.0*PI;

    this->construct(owner, one, two, three, four, angle);
}

double FeatureClass::getAngle(void) const
{
    return Angle;
}

PolygonClass* FeatureClass::getOwner(void) const
{
    return Owner;
}

PointClass FeatureClass::getVertexValue(unsigned int index) const
{
    PointClass p;

    if( (index != 0) && (index != 1) && (index != 2) && (index != 3) )
    {
        cout << "\aInvalid feature vertex[" << index <<
             "] requested!!" << endl;
        return p;
    }

    p = *(PointPointer[index]);
```

```
    return p;
}

boolean FeatureClass::operator == (const FeatureClass& refFeature)
const
{
    if( (*(this->PointPointer[0]) == *(refFeature.PointPointer[0])) &&
        (*(this->PointPointer[1]) == *(refFeature.PointPointer[1])) &&
        (*(this->PointPointer[2]) == *(refFeature.PointPointer[2])) &&
        (*(this->PointPointer[3]) == *(refFeature.PointPointer[3]))
        && (this->Owner == refFeature.Owner)
      )
        return TRUE;
    else
        return FALSE;
}

void FeatureClass::print (boolean address, FILE* fp) const
{
    char owners_name[32];
    Owner->get_name(owners_name);
    fprintf(fp, " Owner %s", owners_name);
    if(address)
        fprintf(fp, "  @ %u", this);
    fprintf(fp,", Angle %8.4f\370\n", toDegrees(Angle));
    PointPointer[0]->print(address, fp); fprintf(fp, "\n");
    PointPointer[1]->print(address, fp); fprintf(fp, "\n");
    PointPointer[2]->print(address, fp); fprintf(fp, "\n");
    PointPointer[3]->print(address, fp); fprintf(fp, "\n");

    fflush(fp);
}

/*  END CLASS FeatureClass */
//--------------------------------------------------------------------------
/* BEGIN CLASS FeatureArray */

// CONSTRUCTOR
FeatureArray::FeatureArray(void)
{
    Length = 0;
    Feature = NULL;
}

FeatureArray::~FeatureArray()
{
    delete [] Feature;
}

unsigned int FeatureArray::getLength(void) const
{
    return Length;
}

/* Function to retrieve a feature by the index count. */
FeatureClass& FeatureArray::getFeature(unsigned int index) const
```

```
{
    FeatureClass* feature = new FeatureClass;
    if(index >= Length)
    {
        cout << "\aAlert: not that many features!\a" << endl;
        // potential memory leak
        return *feature;
    }
    delete feature;
    return Feature[index];
}

status FeatureArray::add (PolygonClass& owner,
                          const PointClass& one,
                          const PointClass& two,
                          const PointClass& three,
                          const PointClass& four, double angle)
{
    if(angle <= 0.0)
    {
        cout << "\aCannot add any feature with an angle <= 0.0"
             << endl;
        return NestFAILED;
    }
    // local pointer
    FeatureClass* new_feature = NULL;
    unsigned int i = 0;
    unsigned int insert_at = 0;
    while((Feature != NULL) && (i < Length))
    {
        if(Feature[i].Angle > angle)
            break;
        i++;
    }
    insert_at = i;
    new_feature = new FeatureClass [Length+1];
    if(new_feature == NULL)
    {
        cout << "Could not add feature!\a" << endl;
        return NestFAILED;
    }
    // increment length
    Length++;
    // copy array till before the insertion point
    for(i = 0; i < insert_at; i++)
        new_feature[i] = Feature[i];

    // add the feature here
    new_feature[insert_at].construct(owner,one,two,three,four,angle);

    // copy array from after the insertion point
    for(i = (insert_at+1); i < Length; i++)
        new_feature[i] = Feature[i-1];
    if(Feature != NULL) delete [] Feature;
    Feature = new_feature;
    return NestOK;
```

```
}

status FeatureArray::add (PolygonClass& owner,
                          const PointClass& one,
                          const PointClass& two,
                          const PointClass& three,
                          const PointClass& four)
{
    double angle = ::get_angle(two, one) - ::get_angle(three, four);

    while(angle > (2.0*PI))
        angle -= 2.0*PI;

    while(angle < 0.0)
        angle += 2.0*PI;

    if(angle <= 0.0)
    {
        cout << "\aCannot add any feature with an angle <= 0.0 (" <<
            toDegrees(angle) << ")" << endl;
        one.print(); cout << "----" << flush;
        two.print(); cout << endl;
        three.print(); cout << "----" << flush;
        four.print(); cout << endl;
        return NestFAILED;
    }
    // new array
    FeatureClass* new_feature = NULL;
    unsigned int i = 0;
    unsigned int insert_at = 0;
    while(i < Length)
    {
        if(Feature[i].Angle > angle)
            break;
        i++;
    }
    insert_at = i;
    new_feature = new FeatureClass [Length+1];
    if(new_feature == NULL)
    {
        cout << "Could not add feature!\a" << endl;
        return NestFAILED;
    }
    // increment length
    Length++;
    // copy array till before the insertion point
    for(i = 0; i < insert_at; i++)
        new_feature[i] = Feature[i];

    // add the feature here
    new_feature[insert_at].construct(owner,one,two,three,four,angle);

    // copy array from after the insertion point
    for(i = (insert_at+1); i < Length; i++)
        new_feature[i] = Feature[i-1];
    delete [] Feature;
```

```
    Feature = new_feature;
    return NestOK;
}

status FeatureArray::add (const FeatureClass& feat)
{
    return this->add(*(feat.Owner), *(feat.PointPointer[0]),
        *(feat.PointPointer[1]), *(feat.PointPointer[2]),
        *(feat.PointPointer[3]));
}

// ADD CHECKING TO SEE IF index HAS A VALID VALUE
status FeatureArray::remove (unsigned int index)

{
    unsigned int i;
    PolygonClass* ptr = Feature[index].Owner;
    ptr->setColor(magenta);
    // Count how many features there are from the given polygon
    int member_features = 0;

    for(i = 0; i < Length; i++)
    {
        if(Feature[i].Owner == ptr)
            member_features++;
    }

    // Flag error on negative array length... the compiler tells me
    //  that this is a redundant check and always evaluates to FALSE..
    //  but I'd like to retain it all the same
    if((Length - member_features) < 0)
    {
        cout << "\aError deleting some features; new length of"
             << "array" <<" < 0" << endl;
        return NestFAILED;
    }

    // Delete array and set Length to 0 on 0 length.
    else if((Length - member_features) == 0)
    {
        Length = 0;
        delete [] Feature;
        Feature = NULL;
        return NestOK;
    }

    FeatureClass* new_feature = NULL;
    new_feature = new FeatureClass [Length - member_features];
    if(new_feature == NULL)
    {
        cout << "Could not allocate new feature array!\a" << endl;
        return NestFAILED;
    }

    unsigned int j = 0;
    for(i = 0; i < Length; i++)
```

```
    {
        // skip if polygon being removed
        if(Feature[i].Owner == ptr)
            continue;
        new_feature[j] = Feature[i];
        j++;
    }
    if(Feature != NULL) delete [] Feature;
    Feature = new_feature;
    Length -= member_features;
    return NestOK;
}

status FeatureArray::remove (const FeatureClass& f)
{
    return remove(*(f.getOwner()));
}

status FeatureArray::remove (const PolygonClass& polygon)
{
    unsigned int i;
    // Count how many features there are from the given polygon
    int member_features = 0;
    for(i = 0; i < Length; i++)
    {
        if(Feature[i].Owner == &polygon)
            member_features++;
    }

    if((Length - member_features) < 0)
    {
        cout << "\aError deleting some features; new length of"
             << "array"<< " < 0" << endl;
        return NestFAILED;
    }
    // Delete array and set Length to 0 on 0 length.
    else if((Length - member_features) == 0)
    {
        Length = 0;
        delete [] Feature;
        Feature = NULL;
        return NestOK;
    }
    FeatureClass* new_feature = NULL;
    new_feature = new FeatureClass [Length - member_features];
    if(new_feature == NULL)
    {
        cout << "Could not allocate new feature array!\a" << endl;
        return NestFAILED;
    }
    unsigned int j = 0;
    for(i = 0; i < Length; i++)
    {
        // skip if polygon being removed
        if(Feature[i].Owner == &polygon)
            continue;
```

```
        new_feature[j] = Feature[i];
        j++;
    }
    if(Feature != NULL) delete [] Feature;
    Feature = new_feature;
    Length -= member_features;

    return NestOK;
}

void FeatureArray::removeAll (void)
{
    Length = 0;
    if(Feature != NULL)
        delete [] Feature;
    Feature = NULL;
}

void FeatureArray::print (boolean address, FILE* fp) const
{
    fprintf(fp, "Feature Array of length %u", Length);
    if(address)
        fprintf(fp, "  @ %u", this);
    fprintf(fp, "\n====================================");
    fprintf(fp, "\n");
    for(unsigned int i = 0; i < Length; i++)
        Feature[i].print(address, fp);
    fprintf(fp, "\n====================================\n");
    fflush(fp);
}

/* END CLASS FeatureArray */
// ------------------------------------------
// BEGIN CLASS PolygonClass

// CONSTRUCTORS
PolygonClass::PolygonClass (void)
{
    Number_Of_Vertices = 0;
    Vertex             = NULL;
    Angle              = NULL;
    Length             = NULL;
    Area               = 0.0;
    Color              = green;
    set_name("\a*Uninitialized*\a");
}

void PolygonClass::construct (const int noOfVertices,
                      const PointClass* inputVertex)
{
    /*  1. Allocate memory to the Vertex list
        2. Allocate memory to the Angle list
        3. Initialize the Vertex list to values of the input list
        4. Calculate area and set the Area field
    */
```

```
// Cant have a polygon with fewer than 3 vertices
if(noOfVertices <= 2)
    return;

int validVertexCount = noOfVertices;
PointClass *inputVertexList = new PointClass[noOfVertices];
for(int i = 0; i < noOfVertices; i++)
    inputVertexList[i] = inputVertex[i];

filterRedundantVertices(&validVertexCount, &inputVertexList);
if(validVertexCount <= 2)
    return;

// 1 & 2
Number_Of_Vertices = validVertexCount; // noOfVertices;
Vertex             = new PointClass[validVertexCount];
Angle              = new double[validVertexCount];
Length             = new double[validVertexCount];

assert( Vertex != 0 );
assert( Angle  != 0 );
assert( Length != 0 );

// 3. Initialize the Vertex list to values of the input list
for(i = 0; i < validVertexCount; i++)
    (Vertex[i]) = (inputVertexList[i]);

for (i = 0; i < validVertexCount; i++)
{
    int j = i+1;
    if(j == validVertexCount) j = 0;
    Length[i] = get_distance(Vertex[i], Vertex[j]);
}

/* 4. Algorithm works for NON SELF OVERLAPPING polygons with
   STRAIGHT LINE EDGES in the COUNTER CLOCKWISE direction ONLY
   "Area of a Simple Polygon", Jon Rokne, Graphics Gems - 2. Also
   provides an easy means to determine whether the Polygon is ccw
   or cw. The area turns out to be negative if the sense of the
   polygon is cw, and hence the vertices may be reversed.
*/
Area = 0.0; // initialize
for (i = 0; i < validVertexCount-1; i++)
{
    Area += Vertex[i].getX()*Vertex[i+1].getY() -
        Vertex[i].getY()*Vertex[i+1].getX();
}

Area +=
    Vertex[validVertexCount-1].getX()*Vertex[0].getY() -
    Vertex[validVertexCount-1].getY()*Vertex[0].getX();

Area /= 2.0;

if(Area < -DELTA)
{
```

```
        // Write an error file!
        char dxf_file[30], errorno[10];
        strcpy(dxf_file, "CW");
        strcat(dxf_file, itoa(errorCount++, errorno, 10));
        strcat(dxf_file, ".dxf");
        DXFout (this, 1, dxf_file);
    }

    set_name("just a polygon");
    // change the "Not Initialized" name from default

    delete [] inputVertexList;
}


// DESTRUCTORS
PolygonClass::~PolygonClass ()
{
    /*  1. Free memory given to the Vertex list
        2. Free memory given to the Angle list
    */

    if( Vertex != NULL )
        delete [] Vertex;
    if( Angle != NULL)
        delete [] Angle;
    if( Length != NULL)
        delete [] Length;
}

// OPERATORS

// METHODS
void PolygonClass::setColor (int color)
{
    Color = color;
}
// sets the Name of the Polygon
void PolygonClass::set_name (const char* string)
{
    // character length restricted to 30
    strncpy(Name, string, 30);
    Name[31] = '\0';
}

// sets the "string" parameter to contain the name of the Polygon
char* PolygonClass::get_name (char* string) const
{
    strcpy(string ,Name);
    return string;
}

/* Function that returns TRUE if the specified point lies within the
   polygon; FALSE otherwise.
   Code modified from the Comp.graphics.algorithms FAQ §2.03 which
   refers to Gems 4, pp. 24-46 */
```

```
position PolygonClass::contains (const PointClass &testPoint) const
{
    return this->contains(testPoint.getX(), testPoint.getY());
}

/* returns 0 if outside; 1 if coincident with a vertex;
   2 if on an edge; 3 if inside */
position PolygonClass::contains (const double x, const double y) const
{
    boolean crossing = FALSE;
    int i, j;
    for(i = 0, j = Number_Of_Vertices-1; i < Number_Of_Vertices;
        j = i++)
    {
        double xi = Vertex[i].getX(), yi = Vertex[i].getY(),
            xj = Vertex[j].getX(), yj = Vertex[j].getY();

        if(( (fabs(x-xi)<DELTA) && (fabs(y-yi)<DELTA) ) ||
           ( (fabs(y-yj)<DELTA) && (fabs(x-xj)<DELTA) )
          )
            return ON_VERTEX; // coincides with a vertex

        double dist_AB = ::get_distance(xi,yi,x,y);
        double dist_BC = ::get_distance(x,y,xj,yj);
        double dist_AC = ::get_distance(xi,yi,xj,yj);
        if((fabs(dist_AB+dist_BC-dist_AC)) <= DELTA)
            return ON_EDGE;

        if((( (yi<=y) && (y<yj) )||( (yj<=y) && (y<yi) )
           )&&(x < (xj-xi)*(y-yi)/(yj-yi) + xi)
          )
        {
          toggle(crossing);
        }
    }

    if(crossing == TRUE) return INSIDE;
    else return OUTSIDE;
}

// Need to test each for speed.
boolean PolygonClass::contains (const PolygonClass& polygon) const
{
    int i;
    /* If any of the polygon.Vertex[i] 's are OUTSIDE this polygon
       instance, then 'polygon' lies outside. */
    for(i = 0; i < polygon.Number_Of_Vertices; i++)
    {
        if(this->contains(polygon.Vertex[i]) == OUTSIDE)
        {
            // polygon.Vertex[i].print();
            // cout << " lies outside #1 " << Name << endl;
            return FALSE;
        }
    }
```

```
/* If any of the this->Vertex[i] 's are within 'polygon', then
   'polygon' lies outside. */
for(i = 0; i < Number_Of_Vertices; i++)
{
    if(polygon.contains(Vertex[i])== INSIDE)
        return FALSE;
}

// this -> Edge(i,j)
for(i = 0; i < this->Number_Of_Vertices; i++)
{
    int j = (i+1) % this->Number_Of_Vertices;
    // Vs. poly -> Edge(k,l)
    for(int k = 0; k < polygon.Number_Of_Vertices; k++)
    {
        PointClass intersection_at;
        int l = (k+1) % polygon.Number_Of_Vertices;
        if(::intersects(this->Vertex[i], this->Vertex[j],
             polygon.Vertex[k], polygon.Vertex[l],
             intersection_at) == TRUE)
        {
            if( (intersection_at == Vertex[i])||
                (intersection_at == Vertex[j])||
                (intersection_at == polygon.Vertex[k])||
                (intersection_at == polygon.Vertex[l]))
                continue;
            else
                return FALSE;
        }
    }
}

PointClass int_pt = polygon.getAnyInternalPoint();
if(contains(int_pt) != INSIDE)
    return FALSE;

// this -> Edge(i,j)
for(int k = 0; k < polygon.Number_Of_Vertices; k++)
{
    int l = (k+1) % polygon.Number_Of_Vertices;
    double x_increment = polygon.Vertex[l].getX() -
        polygon.Vertex[k].getX();
    x_increment /= 50.0;
    double y_increment = polygon.Vertex[l].getY() -
        polygon.Vertex[k].getY();
    y_increment /= 50.0;

    for(int m = 0; m < 50; m++)
    {
        PointClass p;
        p.set(polygon.Vertex[k].getX() + (m*x_increment),
            polygon.Vertex[k].getY() + (m*y_increment) );
        if(this->contains(p) == OUTSIDE)
            return FALSE;
    }
}
```

```
    return TRUE;
}

PointClass PolygonClass::getAnyInternalPoint(void) const
{
    const double epsilon = (sqrt(Area))*0.01; // a small constant
    PointClass internal_point;
    PointArray PointsForDebug;

    for(int i = 0; i < Number_Of_Vertices; i++)
    {
        // increment X
        internal_point.set(Vertex[i].getX()+epsilon,
            Vertex[i].getY());
        PointsForDebug.add(internal_point);
        if(contains(internal_point) == INSIDE) return internal_point;
        // increment X & Y
        internal_point.set(Vertex[i].getX()+epsilon,
            Vertex[i].getY()+epsilon);
        PointsForDebug.add(internal_point);
        if(contains(internal_point) == INSIDE) return internal_point;
        // increment Y
        internal_point.set(Vertex[i].getX(),
            Vertex[i].getY()+epsilon);
        PointsForDebug.add(internal_point);
        if(contains(internal_point) == INSIDE) return internal_point;
        // decrement X, increment Y
        internal_point.set(Vertex[i].getX()-epsilon,
            Vertex[i].getY()+epsilon);
        PointsForDebug.add(internal_point);
        if(contains(internal_point) == INSIDE) return internal_point;
        // decrement X
        internal_point.set(Vertex[i].getX()-epsilon,
            Vertex[i].getY());
        PointsForDebug.add(internal_point);
        if(contains(internal_point) == INSIDE) return internal_point;
        // decrement X, decrement Y
        internal_point.set(Vertex[i].getX()-epsilon, Vertex[i].getY()-
            epsilon);
        PointsForDebug.add(internal_point);
        if(contains(internal_point) == INSIDE) return internal_point;
        // decrement Y
        internal_point.set(Vertex[i].getX(), Vertex[i].getY()-
            epsilon);
        PointsForDebug.add(internal_point);
        if(contains(internal_point) == INSIDE) return internal_point;
        // increment X, decrement Y
        internal_point.set(Vertex[i].getX()+epsilon,
            Vertex[i].getY()-epsilon);
        PointsForDebug.add(internal_point);
        if(contains(internal_point) == INSIDE) return internal_point;
    }
    char fileName[30];
    sprintf(fileName, "%d%s.dxf", errorCount++, this->Name);
    FILE* int_err = fopen(fileName, "w");
    fprintf(int_err, "  0\nSECTION\n  2\nENTITIES\n");
```

```
    this->dxfOut(int_err);
    PointsForDebug.dxfOut(int_err, epsilon/4.0);
    fprintf(int_err, "  0\nENDSEC\n  0\nEOF\n");
    fclose(int_err);
    cout << "\aWarning!! Internal Point not found for "<< Name <<
         endl;
    cout << "epsilon = " << epsilon << endl;
    return internal_point;
}

int PolygonClass::getNumber_Of_Vertices(void) const
{
    return Number_Of_Vertices;
}

/* This method checks the argument polygon poly and returns a TRUE if
   ANY of the edges of the argument polygon and the 'this' polygon
   intersect.
*/
boolean PolygonClass::doesAnyEdgeIntersect(const PolygonClass& poly)
const
{
    // this -> Edge (i,j)
    for(int i = 0; i < Number_Of_Vertices; i++)
    {
        int j = (i+1) % Number_Of_Vertices;
        // Vs. poly -> Edge(k,l)
        for(int k = 0; k < poly.Number_Of_Vertices; k++)
        {
            PointClass intersection_at;
            int l = (k+1) % poly.Number_Of_Vertices;
            if(::intersects(Vertex[i], Vertex[j],
                    poly.Vertex[k], poly.Vertex[l],
                    intersection_at) == TRUE)
            {
                if( (intersection_at == Vertex[i])||
                    (intersection_at == Vertex[j])||
                    (intersection_at == poly.Vertex[k])||
                    (intersection_at == poly.Vertex[l]))
                    cout << "Intersects at one end" << endl;
                return TRUE;
            }
        }
    }
    return FALSE;
}

void PolygonClass::dxfOut(FILE* file) const
{
    fprintf(file, "  0\nPOLYLINE\n  8\nNest\n  62\n%d\n",
        Color);
    fprintf(file, " 66\n     1\n 10\n0.0\n 20\n0.0\n 30\n0.0");
    fprintf(file, "\n 70\n     1\n");
    for(int j = 0; j < Number_Of_Vertices; j++)
    {
        fprintf(file, "  0\nVERTEX\n  8\nNest\n");
```

```
        fprintf(file, " 10\n%f\n", Vertex[j].getX());
        fprintf(file, " 20\n%f\n", Vertex[j].getY());
        fprintf(file, " 30\n");
        fprintf(file, "0.0\n");
    }
    fprintf(file, "  0\nSEQEND\n  8\nNest\n");
    fflush(file);
}

// DEBUG METHODS

// virtual function prints out the list of vertices
void PolygonClass::print(boolean address, FILE* fp) const
{
    fprintf(fp, "\nPolygonClass instance\n");
    fprintf(fp, "Area of Polygon = %8.4f\n", this->Area);
    if(address) fprintf(fp, "Polygon at %u\n", this);
    fflush(fp);
    fprintf(fp, "Vertex List for %d vertices\n", Number_Of_Vertices);
    for (int i = 0; i < Number_Of_Vertices; i++)
    {
        Vertex[i].print(address, fp); fprintf(fp, "\t");
        fprintf(fp, " Length of edge %8.4f",Length[i]);
        if(address) fprintf(fp, " @ %u",&(Length[i]));
        fprintf(fp, "\n"); fflush(fp);
    }
}

// END CLASS PolygonClass

// ---------------------------------------------
// BEGIN CLASS PartClass

// CONSTRUCTORS

PartClass::PartClass (void) : PolygonClass ()
{
    canFlip         = FALSE;
    canRotate       = FALSE;
    canTranslate    = FALSE;
    Is_Convex_Hull  = NULL;
    Color           = blue;
}

void PartClass::construct (const int noOfVertices,
                           const PointClass* inputVertexList)
{
    /*  1. Increment Part count
        2. Allocate Convex Hull flag memory
        3. Set Bounding Box
        4. Set the Angle (s) of the Part vertices
        5. Set the Convex Hull Flags corresponding to the vertices
            that lie on the Conv. Hull.
        6. Set the Minimum Enclosing Rectangle
        7. Set the 'movement' flags to default TRUE
    */
```

```
    // call the base class method
    PolygonClass::construct (noOfVertices, inputVertexList);
    // 1
    PartClass::Number_Of_Parts++;

    // 2
    Is_Convex_Hull = new boolean [noOfVertices];
    assert (Is_Convex_Hull != 0);

    // 3
    set_bounds();

    // 4
    set_angles();

    // 5
    // set an array of pointers to point at the Convex Hull Flags
    //  so that they may be modified by reference within recursive
    //  loops
    boolean **C_H_FlagPtr;
    C_H_FlagPtr = new boolean *[Number_Of_Vertices];

    for (int i = 0; i < Number_Of_Vertices; i++)
        C_H_FlagPtr[i] = &(Is_Convex_Hull[i]);

    mark_convex_hull(Vertex, Number_Of_Vertices, C_H_FlagPtr);

    assert ( C_H_FlagPtr != 0);
    delete [] C_H_FlagPtr;

    // 6
    set_M_E_R();

    // 7
    canFlip = TRUE;
    canRotate = TRUE;
    canTranslate = TRUE;

    return;
}

// DESTRUCTOR
PartClass::~PartClass(void)
{
    /*  1. Decrement the number of parts if the part has not been
           initialized... recognized by the fact that the
           Number_Of_Vertices would be 0
        2. Free Convex Hull Flag memory
    */
    if(Number_Of_Vertices != 0)
    {
        PartClass::Number_Of_Parts--;
        delete [] Is_Convex_Hull;
    }
}
```

```
// OPERATORS

// METHODS
// PRIVATE : sets the rectangular bounds
void PartClass::set_bounds(void)
{
    double ll_x = Vertex[0].getX();
    double ll_y = Vertex[0].getY();
    double ur_x = Vertex[0].getX();
    double ur_y = Vertex[0].getY();

    for (int i = 1; i < Number_Of_Vertices; i++)
    {
        if(ll_x > Vertex[i].getX())
            ll_x = Vertex[i].getX();
        if(ll_y > Vertex[i].getY())
            ll_y = Vertex[i].getY();
        if(ur_x < Vertex[i].getX())
            ur_x = Vertex[i].getX();
        if(ur_y < Vertex[i].getY())
            ur_y = Vertex[i].getY();
    }

    // directly set the LL and UR points for 'this' part
    Lower_Left_Point.set(ll_x, ll_y);
    Upper_Right_Point.set(ur_x, ur_y);
}

// set the angles of the Part during instantiation only!!
// CAUTION: Works for polygons defined in the CCW sense only
void PartClass::set_angles(void)
{
    int curr_index,
        prev_index;
    double *vector_angle,
           delta_x,
           delta_y;
    vector_angle = new double [Number_Of_Vertices];
    assert (vector_angle != 0);

    // begin with the last and zeroth vertices
    for(curr_index = 0, prev_index = (Number_Of_Vertices - 1);
        curr_index < Number_Of_Vertices; curr_index++)
    {
        delta_x = Vertex[curr_index].getX() -
            Vertex[prev_index].getX();
        delta_y = Vertex[curr_index].getY() -
            Vertex[prev_index].getY();

        // get the angles of the vectors
        if (delta_y == 0.0)
        {
            if (delta_x > 0) // case 0°
                vector_angle[curr_index] = 0.0;
            else if (delta_x < 0) // case 180°
```

```
                vector_angle[curr_index] = PI;
            else
            {
                // the consecutive points are the same
                printf("\aDuplicate Point!!\a\n");
                vector_angle[curr_index] = 0.0;
            }
        }
        else if (delta_x == 0.0)
        {
            if(delta_y > 0.0) // case 90°
                vector_angle[curr_index] = PI/2.0;
            else vector_angle[curr_index] = 1.5*PI; // case 270°
        }
        else
            vector_angle[curr_index] = atan (delta_y/delta_x);

        // case in 2nd or 3rd Quad.. atan wont be able to tell
        if (delta_x < 0)
            vector_angle[curr_index] += PI;
        prev_index = curr_index;
    }

    // loop through the vector angles to get the vertex angles
    for(curr_index = 0; curr_index < Number_Of_Vertices; curr_index++)
    {
        if(curr_index == (Number_Of_Vertices - 1)) // last vertex
            Angle[curr_index] = vector_angle[curr_index] + PI -
                vector_angle[0];

        else
            Angle[curr_index] = vector_angle[curr_index] + PI -
                vector_angle[curr_index+1];

        while (Angle[curr_index] >= (2.0*PI)) // change 380° to 20°
            Angle[curr_index] -= (2.0*PI);

        while (Angle[curr_index] < (-2.0*PI)) // change -380° to -20°
            Angle[curr_index] += (2.0*PI);

        // make -20° 340°
        if(Angle[curr_index] < 0.0) Angle[curr_index] += 2.0*PI;
    }
    assert ( vector_angle != 0);
    delete [] vector_angle;
    return;
}

void PartClass::mark_convex_hull (const PointClass* VertexList,
                        const int NumOfVertices,
                        boolean **CHFlagPtrs)
{
    double *angle, vector1, vector2;
    int curr_index, prev_index, num_of_CHs = 0, i, j;
    angle = new double [NumOfVertices];
```

```
prev_index = (NumOfVertices - 1);

for (curr_index = 0; curr_index < NumOfVertices; curr_index++)
{
    // get all the vector angles
    vector1 = get_angle(VertexList[prev_index],
        VertexList[curr_index]);
    if (curr_index == (NumOfVertices - 1))
        vector2 = get_angle(VertexList[curr_index],
        VertexList[0]);
    else
        vector2 =
        get_angle(VertexList[curr_index],VertexList[curr_index+1]);

    // then the vertex angles
    angle[curr_index] = vector1 + PI - vector2;

    while(angle[curr_index] >= 2.0*PI)
        angle[curr_index] -= 2.0*PI;

    while(angle[curr_index] < -2.0*PI)
        angle[curr_index] += 2.0*PI;

    if(angle[curr_index] < 0.0) angle[curr_index] += 2.0*PI;
    if (angle[curr_index] < PI)
    {
        *(CHFlagPtrs[curr_index]) = TRUE; // potential CH
        num_of_CHs++;
    }
    else
    {
        *(CHFlagPtrs[curr_index]) = FALSE;
    }
    prev_index = curr_index;
}

if (num_of_CHs == NumOfVertices)
{
    // EXIT CONDITION OF THE RECURSIVE FUNCTION
    assert (angle != 0);
    delete [] angle;
    return;
}

// else
PointClass *InternalVtxList;
boolean **InternalCHFlagPtrs;

InternalVtxList = new PointClass[num_of_CHs];
InternalCHFlagPtrs = new boolean *[num_of_CHs];
assert (InternalVtxList != 0);
assert (InternalCHFlagPtrs != 0);

for (i = 0, j = 0; i < NumOfVertices; i++)
{
    if (*(CHFlagPtrs[i]) == TRUE)
```

```
        {
            InternalVtxList[j] = VertexList[i];
            InternalCHFlagPtrs[j++] = CHFlagPtrs[i];
        }
    }

    mark_convex_hull(InternalVtxList, num_of_CHs, InternalCHFlagPtrs);

    assert (angle != 0);
    delete [] angle;
    assert (InternalVtxList != 0);
    delete [] InternalVtxList;
    assert (InternalCHFlagPtrs != 0);
    delete [] InternalCHFlagPtrs;

    return;
}

void PartClass::set_M_E_R (void)
{
    int curr_index, next_index, no_of_CH, min_area_index = 0;
    PointClass *CHPoints, *tempCHPoints;
    PointClass BoundPoints[2];
    double vector, area, last_area = 0.0;

    // count the number of vertices on the CH
    for (curr_index = 0, no_of_CH = 0; curr_index <
         Number_Of_Vertices; curr_index++)
        if(Is_Convex_Hull[curr_index] == TRUE) no_of_CH++;

    // Allocate memory
    CHPoints = new PointClass [no_of_CH];
    tempCHPoints = new PointClass [no_of_CH];

    // Set the points equal to the Convex Hull Vertices
    for(curr_index = 0, next_index = 0;
        curr_index < Number_Of_Vertices; curr_index++)
    {
        if(Is_Convex_Hull[curr_index] == TRUE)
            CHPoints[next_index++] = Vertex[curr_index];
    }

    // run the MER loop
    for (curr_index = 0; curr_index < no_of_CH; curr_index++)
    {
        if (curr_index == (no_of_CH-1)) next_index = 0;
        else next_index = (curr_index + 1);
        vector = get_angle(CHPoints[curr_index],
                 CHPoints[next_index]);
        ::rotate((-vector/0.01745329251994), no_of_CH,
            CHPoints[curr_index], CHPoints, tempCHPoints);

        ::get_bounds(no_of_CH, tempCHPoints, BoundPoints, area);
        if(curr_index == 0) last_area = area;
        if (area < last_area)
        {
```

```
            min_area_index = curr_index;
            last_area = area;
        }
    }

    if (min_area_index == (no_of_CH-1)) next_index = 0;
    else next_index = (min_area_index + 1);

    // get vector angle of edge giving min Bound area
    vector = get_angle(CHPoints[min_area_index],
             CHPoints[next_index]);

    // rotate CHPoints by -vector about the CHVertex coinciding
    // with the begin-point of this vector
    ::rotate((-vector/0.01745329251994), no_of_CH,
        CHPoints[min_area_index], CHPoints, tempCHPoints);

    // get the bounding box
    ::get_bounds(no_of_CH, tempCHPoints, BoundPoints, area);

    // build 2 more points of the box while its still aligned with
    //  the +x axis and describable by only 2 points..... i.e. when
    //  its rotated back to its actual position in the GCS, 4 points
    //  will be needed to describe the MER
    PointClass local_M_E_R[4];

    // lower left
    local_M_E_R[0].set(BoundPoints[0]);
    // lower right
    local_M_E_R[1].set(BoundPoints[1].getX(), BoundPoints[0].getY());
    // upper right
    local_M_E_R[2].set(BoundPoints[1]);
    // upper left
    local_M_E_R[3].set(BoundPoints[0].getX(), BoundPoints[1].getY());

    // 'unrotate' 4 points
    ::rotate((vector/0.01745329251994), 4,
        CHPoints[min_area_index], local_M_E_R, this->M_E_R);

    delete [] CHPoints;
    delete [] tempCHPoints;

    return;
}

// Rotates the Part by 'angle' degrees
status PartClass::rotate (const double angle,
                          const PointClass &referencePoint )
{
    double x = referencePoint.getX();
    double y = referencePoint.getY();
    return this->rotate(angle,x,y);
}

status PartClass::rotate (const double angle,
                          const double x, const double y)
```

```
{
    if(canRotate == FALSE)
    {
        printf("\aCan't rotate %s. constrained Part\a\n",Name);
        return NestCONSTRAINED;
    }

    // exit if angle is 0.. saves time
    if (!angle)
        return NestOK;

    // get angle in radians
    double radians = angle*0.01745329251994;
    double cosValue = cos(radians);
    double sinValue = sin(radians);
    double temp_x, temp_y;
    // work with a copy in case part is being rotated about one of
    // its vertices


    // set new vertex locations
    for (int i = 0; i < Number_Of_Vertices; i++)
    {
        temp_x = this->Vertex[i].getX()*cosValue -
            this->Vertex[i].getY()*sinValue -
            x*cosValue + y*sinValue + x;

        temp_y = this->Vertex[i].getX()*sinValue +
            this->Vertex[i].getY()*cosValue -
            x*sinValue - y*cosValue + y;

        this->Vertex[i].set(temp_x, temp_y);
    }

    // rotate the M_E_R
    for (i = 0; i < 4; i++)
    {
        temp_x = this->M_E_R[i].getX()*cosValue -
            this->M_E_R[i].getY()*sinValue -
            x*cosValue + y*sinValue + x;

        temp_y = this->M_E_R[i].getX()*sinValue +
            this->M_E_R[i].getY()*cosValue -
            x*sinValue - y*cosValue + y;
        this->M_E_R[i].set(temp_x, temp_y);
    }
    // reset rectangular bounds
    this->set_bounds();
    return NestOK;
}

// Translates the Part
status PartClass::translate (const PointClass &referencePoint,
                             const PointClass &toPoint)
{
    double x1, y1, x2, y2;
```

```
    x1 = referencePoint.getX();
    y1 = referencePoint.getY();
    x2 = toPoint.getX();
    y2 = toPoint.getY();
    return this->translate(x1, y1, x2, y2);
}

status PartClass::translate (const double x1, const double y1,
                             const double x2, const double y2)
{
    if(canTranslate == FALSE)
    {
        printf("\aCan't translate %s. constrained Part\a\n",Name);
        return NestCONSTRAINED;
    }

    // get deltas
    double delta_x = x2 - x1;
    double delta_y = y2 - y1;

    // save time if both points are the same
    if ( (delta_x == 0.0) && (delta_y == 0.0))
        return NestOK;
    double temp_x, temp_y;
    for (int i = 0; i < Number_Of_Vertices; i++)
    {
        temp_x = this->Vertex[i].getX();
        temp_y = this->Vertex[i].getY();
        temp_x += delta_x;
        temp_y += delta_y;
        this->Vertex[i].set(temp_x, temp_y);
    }
    for (i = 0; i < 4; i++)
    {
        temp_x = this->M_E_R[i].getX();
        temp_y = this->M_E_R[i].getY();
        temp_x += delta_x;
        temp_y += delta_y;
        this->M_E_R[i].set(temp_x, temp_y);
    }
    // reset rectangular bounds
    this->set_bounds();
    return NestOK;
}

// Flips the Part
status PartClass::flip (const PointClass &beginAxis,
                        const PointClass &endAxis)
{
    double x1, y1, x2, y2;
    x1 = beginAxis.getX();
    y1 = beginAxis.getY();
    x2 = endAxis.getX();
    y2 = endAxis.getY();
    return this->translate(x1, y1, x2, y2);
}
```

```
status PartClass::flip (const double x1, const double y1,
                        const double x2, const double y2)
{
    if (canFlip == FALSE)
    {
        printf("\aCan't flip %s. constrained Part\a\n",Name);
        return NestCONSTRAINED;
    }

    double temp_x, temp_y;

    // get deltas
    double delta_x = x1 - x2;
    double delta_y = y1 - y2;
    // save time if both points are the same
    if((delta_x == 0.0) && (delta_y == 0.0) )
    {
        printf("\aCan't flip about a single point!\n");
        return NestOK;
    }
    // duplicate the vertex list and M_E_R
    PointClass *exchangePoint;
    exchangePoint = new PointClass [Number_Of_Vertices + 4];
    assert ( exchangePoint != 0);

    // copy the initial vertex list
    for (int i = 0; i < Number_Of_Vertices; i++)
        (exchangePoint[i]) = (this->Vertex[i]);
    // and the M_E_R
    int j = 0;
    for (i = Number_Of_Vertices; i < (Number_Of_Vertices+4); i++)
        (exchangePoint[i]) = M_E_R[j++];

    // ... reverse ordering of angles
    for (i = 0; i < (int)(Number_Of_Vertices/2); i++)
    {
        double exchangeAngle;
        exchangeAngle = Angle[i];
        Angle[i]= Angle[(Number_Of_Vertices-1)- i];
        Angle[(Number_Of_Vertices-1)- i] = exchangeAngle;
    }

    // reverse lengths
    for (i = 0; i < (int)(Number_Of_Vertices/2); i++)
    {
        double exchangeLength;
        exchangeLength = Length[i];
        Length[i]= Length[(Number_Of_Vertices-1)- i];
        Length[(Number_Of_Vertices-1)- i] = exchangeLength;
    }

    // TO REVERSE the convex hull bit flags
    for (i = 0; i < (int)(Number_Of_Vertices/2); i++)
    {
        boolean temp;
        temp = Is_Convex_Hull[i];
```

```
    Is_Convex_Hull[i] = Is_Convex_Hull[(Number_Of_Vertices-1)-i];
    Is_Convex_Hull[(Number_Of_Vertices-1)-i] = temp;
}

if (delta_y == 0.0) // flip about line parallel to x-axis
{
    for (int i = 0; i < Number_Of_Vertices; i++)
    {
        temp_x =  exchangePoint[i].getX();
        temp_y = -exchangePoint[i].getY();
        temp_y += 2*y1;

        // reverse the order of the vertices to keep the Part
        // in the CCW direction.
        Vertex[Number_Of_Vertices-i-1].set(temp_x, temp_y);
    }

    int j = 3;
    for (i = Number_Of_Vertices; i < (Number_Of_Vertices+4); i++)
    {
        temp_x =  exchangePoint[i].getX();
        temp_y = -exchangePoint[i].getY();
        temp_y += 2*y1;

        // reverse the order of the vertices to keep the Part
        // in the CCW direction.
        M_E_R[j--].set(temp_x, temp_y);
    }
    delete [] exchangePoint;
    // reset rectangular bounds
    this->set_bounds();
    return NestOK;
}

if (delta_x == 0.0) // flip about line parallel to y-axis
{
    for (int i = 0; i < Number_Of_Vertices; i++)
    {
        temp_y =  exchangePoint[i].getY();
        temp_x = -exchangePoint[i].getX();
        temp_x += 2*x1;

        // reverse the order of the vertices to keep the Part
        // in the CCW direction.
        this->Vertex[Number_Of_Vertices-i-1].set(temp_x, temp_y);
    }

    int j = 3;
    for (i = Number_Of_Vertices; i < (Number_Of_Vertices+4); i++)
    {
        temp_y =  exchangePoint[i].getY();
        temp_x = -exchangePoint[i].getX();
        temp_x += 2*x1;

        // reverse the order of the vertices to keep the Part
        // in the CCW direction.
```

```
            M_E_R[j--].set(temp_x, temp_y);
        }
        delete [] exchangePoint;
        // reset rectangular bounds
        this->set_bounds();
        return NestOK;
    }

    double angle, slope, y_intercept, sin2Theta, cos2Theta;

    slope = delta_y/delta_x;
    angle = atan(slope);
    y_intercept = y1 - slope*x1;

    sin2Theta = sin(2.0*angle);
    cos2Theta = cos(2.0*angle);

    for (i = 0; i < Number_Of_Vertices; i++)
    {
        temp_x = exchangePoint[i].getX()*cos2Theta +
            sin2Theta*(exchangePoint[i].getY() - y_intercept);
        temp_y = exchangePoint[i].getX()*sin2Theta -
            exchangePoint[i].getY()*cos2Theta +
            y_intercept*(cos2Theta + 1.0);

        // reverse the order of the vertices to keep the Part
        // in the CCW direction.
        this->Vertex[Number_Of_Vertices-i-1].set(temp_x, temp_y);
    }

    j = 3;
    for (i = Number_Of_Vertices; i < (Number_Of_Vertices+4); i++)
    {
        temp_x = exchangePoint[i].getX()*cos2Theta +
            sin2Theta*(exchangePoint[i].getY() - y_intercept);
        temp_y = exchangePoint[i].getX()*sin2Theta -
            exchangePoint[i].getY()*cos2Theta +
            y_intercept*(cos2Theta + 1.0);

        // reverse the order of the vertices to keep the Part
        // in the CCW direction.
        M_E_R[j--].set(temp_x, temp_y);
    }

    delete [] exchangePoint;
    // reset rectangular bounds
    this->set_bounds();
    return NestOK;
}

// Gives count of Number_Of_Parts
int PartClass::how_many (void)
{
    return PartClass::Number_Of_Parts;
}
```

```
// DEBUG METHODS
void PartClass::print (boolean address, FILE* outfile) const
{
    fprintf(outfile, "\nPartClass instance \"%s\"", Name);
    if (address)
        fprintf(outfile, " at %u", this);
    fprintf(outfile, "\n");
    fprintf(outfile, "Flip[");
    if(canFlip) fprintf(outfile, "Y]  ");
    else fprintf(outfile, "N]  ");
    fprintf(outfile, "Rotate[");
    if(canRotate) fprintf(outfile, "Y]  ");
    else fprintf(outfile, "N]  ");
    fprintf(outfile, "Translate[");
    if(canTranslate) fprintf(outfile, "Y]  ");
    else fprintf(outfile, "N]  ");
    fprintf(outfile, "\n");
    fprintf(outfile, "Area of Part = %8.4f  ", Area);
    if (address) fprintf(outfile, "@ %u", &Area);
    fprintf(outfile, "\n");
    fprintf(outfile, "Bounding Box\n");
    fflush(outfile);
    Lower_Left_Point.print(address, outfile); fprintf(outfile, "\n");
    Upper_Right_Point.print(address, outfile); fprintf(outfile, "\n");
    fprintf(outfile, "Minimum Enclosing Rectangle\n");
    fflush(outfile);
    for(int i = 0; i < 4; i++)
    {
        M_E_R[i].print(address, outfile);
        fprintf(outfile, "\n");
    }
    fprintf(outfile, "Vertex List for %d vertices\n",
        Number_Of_Vertices);
    fflush(outfile);
    for (i = 0; i < Number_Of_Vertices; i++)
    {
        Vertex[i].print(address, outfile);
        fprintf(outfile, "  Angle %8.4f\370",
            Angle[i]/0.01745329251994);
        if (Is_Convex_Hull[i] == TRUE)
            fprintf(outfile, " CH ");
        else fprintf(outfile, " -- ");
        fprintf(outfile, " Length of edge %8.4f",Length[i]);
        if(address) fprintf(outfile, " @ %u",&(Length[i]));
        fprintf(outfile, "\n"); fflush(outfile);
    }
}

// END CLASS PartClass
// ------------------------------------------
// BEGIN CLASS StockClass

// CONSTRUCTORS
StockClass::StockClass (void) : PolygonClass ()
{
    Color = red;
```

```
}

void StockClass::construct (const int noOfVertices,
                        const PointClass* inputVertexList)
{
    PolygonClass::construct (noOfVertices, inputVertexList);
    StockClass::Number_Of_Stocks++;
}

// DESTRUCTOR
StockClass::~StockClass(void)
{
    /*  1. Decrement the number of stocks if the stock has not been
           initialized... recognized by the fact that the
           Number_Of_Vertices would be 0
    */
    if(Number_Of_Vertices != 0)
        StockClass::Number_Of_Stocks--;
}

// OPERATORS

// METHODS
int StockClass::how_many (void)
{
    return StockClass::Number_Of_Stocks;
}

void StockClass::print (boolean address, FILE* fp) const
{
    fprintf(fp, "\nStockClass instance\n");
    fprintf(fp, "Area of Stock = %8.4f\n", Area);
    if(address) fprintf(fp, "Address at %u\n", this);
    fflush(fp);
    fprintf(fp, "Vertex List for %d vertices\n", Number_Of_Vertices);
    for (int i = 0; i < Number_Of_Vertices; i++)
    {
        Vertex[i].print(address, fp); fprintf(fp, "\t");
        fprintf(fp, " Length of edge %8.4f",Length[i]);
        if(address) fprintf(fp, " @ %u",&(Length[i]));
        fprintf(fp, "\n"); fflush(fp);
    }
}

// Returns the area of the shadow cast by the part onto the current
// stock profile.
// NOTE: First check whether the part is COMPLETEY CONTAINED in the
// Stock profile.
double StockClass::leftShadow (const PartClass& part) const
{
    double      shadow_area = 0.0, // initialize to 0.0
                epsilon = 1.0; // for setting end_of_ray
    PointClass  first_intersection_on_stock,
                last_intersection_on_stock,
                end_of_ray;
    int         first_after_index,
```

```
// Stock: idx of 1st vtx after intersection
            last_after_index, // Stock
            part_top_index, // Part
            part_bottom_index; // Part
double      leftmost_X, // Stock
            previous_distance;
// distance of the previous intersection
char        out_file[40];

// find index of the topmost vertex (& leftmost if necessary)
//  on the part
part_top_index = 0;
for(int i = 1; i < part.Number_Of_Vertices; i++)
{
    // if next vertex is higher, set to next vtx
    if((part.Vertex[i].getY() -
        part.Vertex[part_top_index].getY()) > DELTA)
    {
        part_top_index = i;
        continue;
    }
    // if they have the same Y Coordinate, then select one more to
    //  the left
    if(fabs(part.Vertex[part_top_index].getY() -
        part.Vertex[i].getY()) < DELTA)
    {
        if((part.Vertex[part_top_index].getX() -
            part.Vertex[i].getX()) > DELTA)
            part_top_index = i;
    }
}


// find index of the bottommost vertex (& leftmost if necessary)
//  on the part
part_bottom_index = 0;
for(i = 1; i < part.Number_Of_Vertices; i++)
{
    // if next vertex is lower, set to next vtx
    if((part.Vertex[part_bottom_index].getY() -
        part.Vertex[i].getY()) > DELTA)
    {
        part_bottom_index = i;
        continue;
    }
    // if they have the same Y Coordinate...
    if(fabs(part.Vertex[part_bottom_index].getY()-
        part.Vertex[i].getY()) < DELTA)
    {
        if((part.Vertex[part_bottom_index].getX()-
            part.Vertex[i].getX()) > DELTA)
            part_bottom_index = i;
    }
}
```

```
leftmost_X = this->Vertex[0].getX();
for(i = 1; i < this->Number_Of_Vertices; i++)
    if(leftmost_X > this->Vertex[i].getX())
        leftmost_X = this->Vertex[i].getX();

end_of_ray.set(leftmost_X-epsilon,
    part.Vertex[part_top_index].getY());
previous_distance = 1000.0*sqrt(this->Area); // ARBIT!!!!!!!!!
for(i = 0; i < this->Number_Of_Vertices; i++)
{
    int j = (i+1) % this->Number_Of_Vertices;
    // 1. Check if this part.Vertex lies on the stock edge or
    //    vertex
    double ab = ::get_distance(this->Vertex[i],
        part.Vertex[part_top_index]);
    double bc = ::get_distance(part.Vertex[part_top_index], this-
        >Vertex[j]);
    double ac = ::get_distance(this->Vertex[i], this->Vertex[j]);
    if((part.Vertex[part_top_index] == this->Vertex[i]) ||
       (part.Vertex[part_top_index] == this->Vertex[j]) ||
       (fabs(ab + bc -ac) < DELTA)
      ) // PART TOP VERTEX LIES ON STOCK EDGE OR VERTEX
    {
        double angle = ::get_angle(this->Vertex[i], this-
            >Vertex[j]);
        // if the line is horizontal, continue
        if((fabs(angle) < DELTA) || (fabs(angle - PI) < DELTA))
            continue;
        angle += PI/2.0; // add 90 SINCE LINES ARE CCW!!
        // find if its x-component is left-to-right (OK) or
        //  right-to-left (not OK)
        PointClass a(0.0, 0.0);
        PointClass b(5.0*cos(angle), 5.0*sin(angle));
        if((b.getX() - a.getX()) > DELTA)
        {
            // Valid intersection at this point... direction of
            //  normal opposes that of the ray
            first_intersection_on_stock =
                part.Vertex[part_top_index];
            first_after_index = j;
            break;
        }
    }
    else if(intersects(part.Vertex[part_top_index], end_of_ray,
        this->Vertex[i], this->Vertex[j]) == TRUE)
    {
        double angle = ::get_angle(this->Vertex[i], this-
            >Vertex[j]);
        angle += PI/2.0; // add 90 SINCE LINES ARE CCW!!
        // find if its x-component is left-to-right (OK) or
        //  right-to-left (not OK)
        PointClass a(0.0, 0.0);
        PointClass b(5.0*cos(angle), 5.0*sin(angle));
        if((b.getX() - a.getX()) < -1.0*DELTA)
        {
            // Invalid intersection at this point... direction of
```

```
            // normal is same as that of the ray
            continue;
        }

        intersects(part.Vertex[part_top_index], end_of_ray,
            this->Vertex[i], this->Vertex[j], a);
        // PART TOP VERTEX DOES NOT LIE ON STOCK EDGE OR VERTEX
        if(::get_distance(part.Vertex[part_top_index], a) <
            previous_distance) // not using DELTA
        {
            previous_distance =
               ::get_distance(part.Vertex[part_top_index], a);
            first_intersection_on_stock = a;
            first_after_index = j;
        }
    }
}

end_of_ray.set(leftmost_X-
    epsilon,part.Vertex[part_bottom_index].getY());
previous_distance = 1000.0*sqrt(this->Area); // ARBIT!!!!!!!!!
for(i = 0; i < this->Number_Of_Vertices; i++)
{
    int j = (i+1) % this->Number_Of_Vertices;
    // 1. Check if this part.Vertex lies on the stock edge or
    //    vertex
    double ab = ::get_distance(this->Vertex[i],
        part.Vertex[part_bottom_index]);
    double bc = ::get_distance(part.Vertex[part_bottom_index],
        this->Vertex[j]);
    double ac = ::get_distance(this->Vertex[i], this->Vertex[j]);
    if((part.Vertex[part_bottom_index] == this->Vertex[i]) ||
       (part.Vertex[part_bottom_index] == this->Vertex[j]) ||
       (fabs(ab + bc -ac) < DELTA)
      ) // PART TOP VERTEX LIES ON STOCK EDGE OR VERTEX
    {
        double angle = ::get_angle(this->Vertex[i], this-
            >Vertex[j]);
        // if the line is horizontal, continue
        if((fabs(angle) < DELTA) || (fabs(angle - PI) < DELTA))
            continue;
        angle += PI/2.0; // add 90
        // find if its x-component is left-to-right (OK) or
        //  right to left (not OK)
        PointClass a(0.0, 0.0);
        PointClass b(5.0*cos(angle), 5.0*sin(angle));
        if((b.getX() - a.getX()) > DELTA)
        {
            // Valid intersection at this point... direction of
            //  normal opposes that of the ray
            last_intersection_on_stock =
                part.Vertex[part_bottom_index];
            last_after_index = j;
            break;
        }
    }
```

```
    else if(intersects(part.Vertex[part_bottom_index], end_of_ray,
        this->Vertex[i], this->Vertex[j]) == TRUE)
    {
        double angle = ::get_angle(this->Vertex[i], this-
            >Vertex[j]);
        angle += PI/2.0; // add 90 SINCE LINES ARE CCW!!
        // find if its x-component is left-to-right (OK) or
        //  right-to-left (not OK)
        PointClass a(0.0, 0.0);
        PointClass b(5.0*cos(angle), 5.0*sin(angle));
        if((b.getX() - a.getX()) < -1.0*DELTA)
        {
            // Invalid intersection at this point... direction of
            //  normal is same as that of the ray
            continue;
        }

        intersects(part.Vertex[part_bottom_index], end_of_ray,
            this->Vertex[i], this->Vertex[j], a);
        // PART TOP VERTEX DOES NOT LIE ON STOCK EDGE OR VERTEX
        if(::get_distance(part.Vertex[part_bottom_index], a) <
            previous_distance) // not using DELTA
        {
            previous_distance =
                ::get_distance(part.Vertex[part_bottom_index], a);
            last_intersection_on_stock = a;
            last_after_index = j;
        }
    }
}

// Count no. of vertices on part which are part of the shadow
//  polygon
int vertices_from_part = 0;
for(i = 0; i < part.Number_Of_Vertices; i++)
{
    vertices_from_part++;
    int j = (part_top_index+i) % part.Number_Of_Vertices;
    if(j == part_bottom_index) break;
}

int vertices_from_stock = 2; // at least
int advance_one_vtx = first_after_index;
int count = 0;

while(advance_one_vtx != last_after_index)
{
    count++;
    advance_one_vtx = (first_after_index+count) %
        Number_Of_Vertices;
    if(advance_one_vtx == first_after_index)
    {
        cout << "\a\aCould not determine how many vertices from"
             << " the" << "Stock!\a (left)" << endl;
        break;
    }
```

```
}
vertices_from_stock += count;

// cout << vertices_from_stock << " vertices from Stock" << endl;
PointClass *shadow_vertex =
    new PointClass [vertices_from_stock+vertices_from_part];
count = 0;
shadow_vertex[count++] = first_intersection_on_stock;
for(i = 0; i < (vertices_from_stock-2); i++)
{
    int stock_index = (first_after_index+i) % Number_Of_Vertices;
    shadow_vertex[count++] = Vertex[first_after_index+i];
}
shadow_vertex[count++] = last_intersection_on_stock;
for(i = 0; i < vertices_from_part; i++)
{
    int part_index = (part_bottom_index-i+part.Number_Of_Vertices)
        % part.Number_Of_Vertices;
    shadow_vertex[count++] = part.Vertex[part_index];
}
PolygonClass p;
p.construct((vertices_from_stock+vertices_from_part),
    shadow_vertex);

// FOR DEBUGGING PURPOSES!!!!!!!!!!!!!!
if(leftShadowPolygon != NULL)
    delete leftShadowPolygon;
leftShadowPolygon = new PolygonClass();
leftShadowPolygon->construct(
    (vertices_from_stock+vertices_from_part), shadow_vertex);
shadow_area = p.Area;

if(p.Area < -1.0*DELTA)
{
    PolygonClass temps[3];
    // This stock
    temps[0].construct(this->Number_Of_Vertices, this->Vertex);
    temps[0].Color = red;
    // Part
    temps[1].construct(part.Number_Of_Vertices, part.Vertex);
    temps[1].Color = cyan;
    // ERRANT Polygon
    temps[2].construct(p.Number_Of_Vertices, p.Vertex);
    temps[2].Color = green;

    strcpy(out_file, "LShad");
    char dummy[10];
    strcat(out_file, itoa(errorCount, dummy, 10));
    strcat(out_file, ".dxf");
    FILE* fp = fopen(out_file, "w");
    fprintf(fp, "  0\nSECTION\n  2\nENTITIES\n");
    temps[0].dxfOut(fp);
    temps[1].dxfOut(fp);
    temps[2].dxfOut(fp);
    ::dxfOut("1", first_intersection_on_stock, 0.1, fp);
    first_intersection_on_stock.dxfOut(fp);
```

```
        ::dxfOut ("2", last_intersection_on_stock, 0.1, fp);
        last_intersection_on_stock.dxfOut(fp);
        errorCount++;
        fprintf(fp, "  0\nENDSEC\n  0\nEOF\n");
        fclose(fp);
    }

    delete [] shadow_vertex;
    return shadow_area;
}

double StockClass::underShadow (const PartClass& part) const
{
    double      shadow_area = 0.0,
                epsilon = 1.0;
    PointClass  first_intersection_on_stock,
                last_intersection_on_stock,
                end_of_ray;
    int         first_after_index, // Stock
                last_after_index, // Stock
                part_left_index, // Part
                part_right_index; // Part
    double      bottommost_Y,
                previous_distance;
    char        out_file[40];

    // find the index of the leftmost (& bottom-most) part vertex
    part_left_index = 0;

    for(int i = 1; i < part.Number_Of_Vertices; i++)
    {
        // if the next vertex is more to the left, set to next vertex
        if((part.Vertex[part_left_index].getX()-part.Vertex[i].getX())
            > DELTA)
        {
            part_left_index = i;
            continue;
        }
        // if x co-ords are same, get lower of the two
        if(fabs(part.Vertex[part_left_index].getX() -
            part.Vertex[i].getX()) < DELTA)
        {
            if((part.Vertex[part_left_index].getY() -
                part.Vertex[i].getY()) > DELTA)
                part_left_index = i;
        }
    }

    // find the index of the rightmost (& bottom-most) vertex
    part_right_index = 0;
    for(i = 1; i < part.Number_Of_Vertices; i++)
    {
        // if the next vertex is more to the right
        if((part.Vertex[i].getX()-
            part.Vertex[part_right_index].getX()) > DELTA)
        {
```

```
            part_right_index = i;
            continue;
        }
        // if they have the same X co-ordinate
        if(fabs(part.Vertex[part_right_index].getX() -
            part.Vertex[i].getX()) < DELTA)
        {
            if((part.Vertex[part_right_index].getY() -
                part.Vertex[i].getY()) > DELTA)
                part_right_index = i;
        }
    }

    bottommost_Y = this->Vertex[0].getY();
    for(i = 1; i < this->Number_Of_Vertices; i++)
        if(bottommost_Y > this->Vertex[i].getY())
            bottommost_Y = this->Vertex[i].getY();

    // Shoot vertical rays from the vertices
    end_of_ray.set(part.Vertex[part_left_index].getX(),bottommost_Y-
        epsilon);
    previous_distance = 1000.0*sqrt(this->Area); // ARBIT!!!!!!!!!
    for(i = 0; i < this->Number_Of_Vertices; i++)
    {
        int j = (i+1) % this->Number_Of_Vertices;
        // 1. Check if this part.Vertex lies on the stock edge or
        //    vertex
        double ab = ::get_distance(this->Vertex[i],
            part.Vertex[part_left_index]);
        double bc = ::get_distance(part.Vertex[part_left_index], this-
            >Vertex[j]);
        double ac = ::get_distance(this->Vertex[i], this->Vertex[j]);
        if((part.Vertex[part_left_index] == this->Vertex[i]) ||
           (part.Vertex[part_left_index] == this->Vertex[j]) ||
           (fabs(ab + bc -ac) < DELTA)
          ) // PART TOP VERTEX LIES ON STOCK EDGE OR VERTEX
        {
            double angle = ::get_angle(this->Vertex[i], this-
                >Vertex[j]);
            // if the line is vertical, continue
            if((fabs(angle-(PI/2.0))<DELTA)||(fabs(angle-
                (3.0*PI/2.0))<DELTA))
                continue;
            angle += PI/2.0; // add 90 SINCE LINES ARE CCW!!
            // find if its Y-component is down-to-up (OK) or
            //  up-to-down (not OK)
            PointClass a(0.0, 0.0);
            PointClass b(5.0*cos(angle), 5.0*sin(angle));
            if((b.getY() - a.getY()) > DELTA)
            {
                // Valid intersection at this point... direction of
                //  normal opposes that of the ray
                first_intersection_on_stock =
                    part.Vertex[part_left_index];
                first_after_index = j;
                break;
```

```
        }
    }
    else if(intersects(part.Vertex[part_left_index], end_of_ray,
        this->Vertex[i], this->Vertex[j]) == TRUE)
    {
        double angle = ::get_angle(this->Vertex[i], this-
            >Vertex[j]);
        angle += PI/2.0; // add 90 SINCE LINES ARE CCW!!
        PointClass a(0.0, 0.0);
        PointClass b(5.0*cos(angle), 5.0*sin(angle));
        if((b.getY() - a.getY()) <= -1.0*DELTA)
        {
            // Invalid intersection at this point... direction of
            //  normal same as that of the ray
            continue;
        }

        intersects(part.Vertex[part_left_index], end_of_ray,
            this->Vertex[i], this->Vertex[j], a);
        // PART TOP VERTEX DOES NOT LIE ON STOCK EDGE OR VERTEX
        if(::get_distance(part.Vertex[part_left_index], a) <
            previous_distance) // not using DELTA
        {
            previous_distance =
                ::get_distance(part.Vertex[part_left_index], a);
            first_intersection_on_stock = a;
            first_after_index = j;
        }
    }
}

end_of_ray.set(part.Vertex[part_right_index].getX(),bottommost_Y-
    epsilon);
previous_distance = 1000.0*sqrt(this->Area); // ARBIT!!!!!!!!!
for(i = 0; i < this->Number_Of_Vertices; i++)
{
    int j = (i+1) % this->Number_Of_Vertices;
    // 1. Check if this part.Vertex lies on the stock edge or
    //    vertex
    double ab = ::get_distance(this->Vertex[i],
        part.Vertex[part_right_index]);
    double bc = ::get_distance(part.Vertex[part_right_index],
        this->Vertex[j]);
    double ac = ::get_distance(this->Vertex[i], this->Vertex[j]);
    if((part.Vertex[part_right_index] == this->Vertex[i]) ||
       (part.Vertex[part_right_index] == this->Vertex[j]) ||
       (fabs(ab + bc -ac) < DELTA)
      ) // PART TOP VERTEX LIES ON STOCK EDGE OR VERTEX
    {
        double angle = ::get_angle(this->Vertex[i], this-
            >Vertex[j]);
        // if the line is vertical, continue
        if((fabs(angle-(PI/2.0))<DELTA)||(fabs(angle-
            (3.0*PI/2.0))<DELTA))
            continue;
        angle += PI/2.0; // add 90
```

```
            PointClass a(0.0, 0.0);
            PointClass b(5.0*cos(angle), 5.0*sin(angle));
            if((b.getY() - a.getY()) > DELTA)
            {
                // Valid intersection at this point... direction of
                //  normal opposes that of the ray
                last_intersection_on_stock =
                    part.Vertex[part_right_index];
                last_after_index = j;
                break;
            }
        }
        else if(intersects(part.Vertex[part_right_index], end_of_ray,
            this->Vertex[i], this->Vertex[j]) == TRUE)
        {
            double angle = ::get_angle(this->Vertex[i], this-
                >Vertex[j]);
            angle += PI/2.0; // add 90 SINCE LINES ARE CCW!!
            PointClass a(0.0, 0.0);
            PointClass b(5.0*cos(angle), 5.0*sin(angle));
            if((b.getY() - a.getY()) <= -1.0*DELTA)
            {
                // Invalid intersection at this point... direction of
                //  normal same as that of the ray
                continue;
            }

            intersects(part.Vertex[part_right_index], end_of_ray,
                this->Vertex[i], this->Vertex[j], a);
            // PART TOP VERTEX DOES NOT LIE ON STOCK EDGE OR VERTEX
            if(::get_distance(part.Vertex[part_right_index], a) <
                previous_distance) // not using DELTA
            {
                previous_distance =
                    ::get_distance(part.Vertex[part_right_index], a);
                last_intersection_on_stock = a;
                last_after_index = j;
            }
        }
    }

    int vertices_from_part = 0;
    for(i = 0; i < part.Number_Of_Vertices; i++)
    {
        vertices_from_part++;
        int j = (part_left_index+i) % part.Number_Of_Vertices;
        if(j == part_right_index) break;
    }

    int vertices_from_stock = 2; // at least
    int advance_one_vtx = first_after_index;
    int count = 0;

    while(advance_one_vtx != last_after_index)
    {
        count++;
```

```
    advance_one_vtx = (first_after_index+count) %
        Number_Of_Vertices;
    if(advance_one_vtx == first_after_index)
    {
        cout << "\a\aCould not determine how many vertices from \
                the"  << " Stock!\a (bottom)" << endl;
        break;
    }
}
vertices_from_stock += count;

PointClass *shadow_vertex =
    new PointClass [vertices_from_stock+vertices_from_part];
count = 0;
shadow_vertex[count++] = first_intersection_on_stock;
for(i = 0; i < (vertices_from_stock-2); i++)
{
    int stock_index = (first_after_index+i) % Number_Of_Vertices;
    shadow_vertex[count++] = Vertex[first_after_index+i];
}
shadow_vertex[count++] = last_intersection_on_stock;
for(i = 0; i < vertices_from_part; i++)
{
    int part_index = (part_right_index-i+part.Number_Of_Vertices)
                      % part.Number_Of_Vertices;
    shadow_vertex[count++] = part.Vertex[part_index];
}
PolygonClass p;
p.construct((vertices_from_stock+vertices_from_part),
    shadow_vertex);
shadow_area = p.Area;

if(p.Area < -1.0*DELTA)
{
    PolygonClass temps[4];
    // This stock
    temps[0].construct(this->Number_Of_Vertices, this->Vertex);
    temps[0].Color = red;
    // Part
    temps[1].construct(part.Number_Of_Vertices, part.Vertex);
    temps[1].Color = cyan;
    // ERRANT Polygon
    temps[2].construct(p.Number_Of_Vertices, p.Vertex);
    temps[2].Color = green;
    // Left Shadow Polygon
    temps[3].construct(leftShadowPolygon->Number_Of_Vertices,
        leftShadowPolygon->Vertex);
    temps[3].Color = yellow;

    strcpy(out_file, "UShad");
    char dummy[10];
    strcat(out_file, itoa(errorCount, dummy, 10));
    strcat(out_file, ".dxf");

    FILE* fp = fopen(out_file, "w");
    fprintf(fp, "  0\nSECTION\n  2\nENTITIES\n");
```

```
        temps[0].dxfOut(fp);
        temps[1].dxfOut(fp);
        temps[2].dxfOut(fp);
        temps[3].dxfOut(fp);
        ::dxfOut("1", first_intersection_on_stock, 0.1, fp);
        first_intersection_on_stock.dxfOut(fp);
        ::dxfOut ("2", last_intersection_on_stock, 0.1, fp);
        last_intersection_on_stock.dxfOut(fp);
        errorCount++;
        fprintf(fp, "  0\nENDSEC\n  0\nEOF\n");
        fclose(fp);
        errorCount++;
    }

    delete [] shadow_vertex;

    return shadow_area;
}

// Routine ASSUMES THAT THE ARGUMENT POLYGON IS COMPLETELY CONTAINED
//  and is a part and at least one vertex of the polygon is in contact
//  with at least one stock edge
status StockClass::updateProfile (const PolygonClass& polygon)
{
    /* Edge naming convention is edge 0 is segment 0-1, edge 1 is
       segment 1-2 */
    if(contains(polygon) != TRUE)
        return NestFAILED;
    int StockTopmostIndex = -1,
        /* Index of uppermost (& rightmost) stock vtx since we do
           choose to nest progressively against the lowest and
           leftmost vertex... if the part is in contact with this
           vertex though, the algorithm will FAIL.
        */
        PolygonVertexInsideStock = -1,
        /* Index of a polygon vtx that lies inside... this vertex
           should be on the side of the stock which is to be retained
           after chopping off the part... eg. if a lower left vtx of
           the polygon is fully contained by the stock, then the
           algorithm will have problems!!
           So we find the RIGHTMOST VERTEX OF THE PART WHICH LIES
           WITHIN THE STOCK... HOPING THAT THIS IS THE RIGHT AREA OF
           THE STOCK WHICH WE WOULD LIKE TO RETAIN AFTER THE PART HAS
           BASICALLY BEEN SUBTRACTED FROM THE STOCK.
        */
        FirstIntersectionIndex = -1,
        LastIntersectionIndex  = -1,
        where1 = -1, // Where the intersection point was found
        where2 = -1, // Where the intersection point was found
        verticesFromStock = 0,
        // Keeps count of total points in updated
        verticesFromPolygon = 0,// stock profile
        i,
        indexFromTop,
        indexFromBottom,
        newNumber_Of_Vertices;
```

```
const int ON_A = 0,
    ON_B = 1,
    ON_AB = 2,
    maxPossibleVertices = Number_Of_Vertices +
        polygon.Number_Of_Vertices;
    // represents the maximum # of possible vertices of the new
    //  stock profile

PointClass* tempVtxArray = new PointClass[maxPossibleVertices+1];
PointClass OutOfBounds, *AnotherTempVtxArray;
char tempName[32];

// 1.  Find uppermost (& rightmost) vertex index of the stock
StockTopmostIndex = -1;
for(i = 0; i < Number_Of_Vertices; i++)
{
    if(Vertex[StockTopmostIndex].getY() < Vertex[i].getY())
        StockTopmostIndex = i;
    else if(Vertex[StockTopmostIndex].getY() == Vertex[i].getY())
    {
        if(Vertex[StockTopmostIndex].getX() < Vertex[i].getX())
            StockTopmostIndex = i;
    }
}
if(StockTopmostIndex == -1)
{
    cout << "\aInvalid StockTopmostIndex value!! ERROR!!\a" << \
         endl;
    return NestFAILED;
}

// Initialize all the vertices to a value none of the polygon or
//  stock can ever have [i.e. added (100, 100) to the highest
//  stock point here]
// Initialize the OutOfBounds Point first
OutOfBounds.set(Vertex[StockTopmostIndex].getX() + 100.0,
        Vertex[StockTopmostIndex].getY() + 100.0);
for(i = 0; i < maxPossibleVertices; i++)
    tempVtxArray[i] = OutOfBounds;

// Find the righmost vertex of the part completely inside the
// stock... see the top of this function for details.
PolygonVertexInsideStock = -1;
for(i = 0; i < polygon.Number_Of_Vertices; i ++)
{
    if(this->contains(polygon.Vertex[i]) == INSIDE)
    {
        if(PolygonVertexInsideStock == -1)
            PolygonVertexInsideStock = i;
        else
        {
            if(polygon.Vertex[PolygonVertexInsideStock].getX() <
                polygon.Vertex[i].getX())
                PolygonVertexInsideStock = i;
```

```
        }
    }
    else
        continue;
}
// If all vertices of the part lie on an edge of the stock, use
//  the highest vertex in the part as the starting point
if(PolygonVertexInsideStock == -1)
{
    cout << "\aStockClass::updateProfile : All vertices of the \
         part" << " lie on the stock edges... selecting topmost \
         vertex of polygon" << endl;
    PolygonVertexInsideStock = 0;
    for(i = 0; i < polygon.Number_Of_Vertices; i ++)
    {
        if(polygon.Vertex[PolygonVertexInsideStock].getY() <
            polygon.Vertex[i].getY())
            PolygonVertexInsideStock = i;
    }
}

// 2. Traverse along CCW from this point and find the first
//    polygon vtx that lies on the edge.
indexFromTop = 0;
// ...always points to the currently unfilled position in the
//  tempVtxArray
verticesFromStock = 0;

for(i = 0; i < this->Number_Of_Vertices; i++)
{
    // Start at the top most vertex of the Stock...
    int l = (StockTopmostIndex + i) % this->Number_Of_Vertices;
    // Traverse CCW...
    int m = (l + 1) % this->Number_Of_Vertices;
    boolean found = FALSE;
    // required to break out of the outer loop
    tempVtxArray[indexFromTop] = Vertex[l];
    indexFromTop++;
    for(int j = 0; j < polygon.Number_Of_Vertices; j++)
    {
        // Start at the selected polygon vertex...
        int n = (PolygonVertexInsideStock + j) %
            polygon.Number_Of_Vertices;
        // Part.Vtx[n] same as 1st Stock.Vtx[l] on this stock-
        //  edge-segment
        if(polygon.Vertex[n] == this->Vertex[l])
        {
            FirstIntersectionIndex = n;
            where1 = ON_A;
            found = TRUE;
            verticesFromStock++;
            break;
        }
        // If polygon.Vertex[n] lies on an edge, record the edge
        //  and the index of the vertex point
        double dist_AB = ::get_distance(this->Vertex[l],
```

```
            polygon.Vertex[n]);
        double dist_BC = ::get_distance(polygon.Vertex[n], this-
            >Vertex[m]);
        double dist_AC = ::get_distance(this->Vertex[l], this-
            >Vertex[m]);
        if((fabs(dist_AB + dist_BC - dist_AC)) <= DELTA)
        {
            FirstIntersectionIndex = n;
            where1 = ON_AB;
            found = TRUE;
            tempVtxArray[indexFromTop] = polygon.Vertex[n];
            indexFromTop++;
            break;
        }
        // If this->Vertex[l] lies on the part edge n--o, record
        //  the edge and the index of the vertex point
        int o = (n + 1) % polygon.Number_Of_Vertices;
        dist_AB = ::get_distance(polygon.Vertex[n], this-
            >Vertex[l]);
        dist_BC = ::get_distance(this->Vertex[l],
            polygon.Vertex[o]);
        dist_AC = ::get_distance(polygon.Vertex[n],
            polygon.Vertex[o]);
        if((fabs(dist_AB + dist_BC - dist_AC)) <= DELTA)
        {
            // Stock vtx lies on part edge
            FirstIntersectionIndex = o;
            found = TRUE;
            break;
        }
    }
    if(found == TRUE) break;
}

// 3.  Traverse along CW from this point and find the first
//     polygon vtx that lies on the edge.

indexFromBottom = maxPossibleVertices;
for(i = 0; i < this->Number_Of_Vertices; i ++)
{
    // Start at the top most vertex of the Stock... NOTICE THAT
    //  THE FIRST POINT WE PICK FROM THE STOCK IS BEING REPEATED
    //  HERE!!... THIS IS NECESSARY AS THE PART MAY HAVE AN
    //  INTERSECTION WITH THE FIRST CW EDGE MADE BY THIS STOCK
    //  VERTEX AND ITS NEXT CW VERTEX!
    int l = (StockTopmostIndex - i + this->Number_Of_Vertices) %
        this->Number_Of_Vertices;
    // Traverse CW...
    int m = (l - 1 + this->Number_Of_Vertices) % this-
        >Number_Of_Vertices;
    boolean found = FALSE;
    tempVtxArray[indexFromBottom] = Vertex[l];
    indexFromBottom--;
    verticesFromStock++;
    for(int j = 0; j < polygon.Number_Of_Vertices; j++)
    {
```

```
            // Start at the top most vertex of the part...traversing
            //  the part CW too.... NECESSARY!!... if there are 2 part
            //  vertices lying on the same stock edge, we want the one
            //  that comes before if u traverse the part CW!!
            int n = (PolygonVertexInsideStock - j +
                polygon.Number_Of_Vertices)%
                polygon.Number_Of_Vertices;
            // If Vertex[j] lies on an edge, record the edge and
            // the index of the vertex point
            if(polygon.Vertex[n] == this->Vertex[l])
            {
                // polygon vtx same as 1st stock vtx in this stock-
                //  edge-segment
                LastIntersectionIndex = n;
                where2 = ON_A;
                found = TRUE;
                break;
            }
            double dist_AB = ::get_distance(Vertex[l],
                polygon.Vertex[n]);
            double dist_BC = ::get_distance(polygon.Vertex[n],
                Vertex[m]);
            double dist_AC = ::get_distance(Vertex[l], Vertex[m]);
            if((fabs(dist_AB + dist_BC - dist_AC)) <= DELTA)
            {
                // polygon vtx lies on edge
                LastIntersectionIndex = n;
                where2 = ON_AB;
                found = TRUE;
                tempVtxArray[indexFromBottom] = polygon.Vertex[n];
                indexFromBottom--;
                verticesFromStock++;
                break;
            }
            int o = (n - 1 + polygon.Number_Of_Vertices) %
                polygon.Number_Of_Vertices;
            dist_AB = ::get_distance(polygon.Vertex[n], this-
                >Vertex[l]);
            dist_BC = ::get_distance(this->Vertex[l],
                polygon.Vertex[o]);
            dist_AC = ::get_distance(polygon.Vertex[n],
                polygon.Vertex[o]);
            if((fabs(dist_AB + dist_BC - dist_AC)) <= DELTA)
            {
                // polygon vtx lies on edge
                LastIntersectionIndex = o;
                found = TRUE;
                break;
            }
        }
        if(found == TRUE) break;
    }

    // Extract points from polygon
    // cout << "Adding points from the part now" << endl;
    for(i = 0; i < polygon.Number_Of_Vertices; i++)
```

```
    {
        int p = (FirstIntersectionIndex-1 - i +
            polygon.Number_Of_Vertices) % polygon.Number_Of_Vertices;
        if(p == LastIntersectionIndex) break;
        tempVtxArray[indexFromTop] = polygon.Vertex[p];
        indexFromTop++;
    }

    // set new count for Stock vertices
    newNumber_Of_Vertices = 0;
    for(i = 0; i < maxPossibleVertices; i++)
    {
        if(tempVtxArray[i] == OutOfBounds)
            continue;
        newNumber_Of_Vertices++;
    }

    // Create a final array to pass to Base Class re-constructor
    AnotherTempVtxArray = new PointClass[newNumber_Of_Vertices];
    int tempVtxArrayIndex = 0;
    for(i = 0; i < maxPossibleVertices; i++)
    {
        if(tempVtxArray[i] == OutOfBounds)
            continue;
        AnotherTempVtxArray[tempVtxArrayIndex++] = tempVtxArray[i];
    }

    // Update Stock
    delete [] Vertex;
    delete [] Angle;
    delete [] Length;

    strcpy(tempName, Name);
    PolygonClass::construct (newNumber_Of_Vertices,
        AnotherTempVtxArray);
    strcpy(Name, tempName);

    delete [] tempVtxArray;
    delete [] AnotherTempVtxArray;
    return NestOK;
}

// DEBUG METHODS
// END CLASS StockClass
// ---------------------------------------------
//  Standard Geometric Utilities

// Returns the angle made by the vector defined by 2 points
// with the +'ve x-axis in Radians
// There's no error checking to see if the points are identical
double get_angle(const PointClass& one, const PointClass& two)
{
    double delta_x, delta_y, angle;

    delta_x = two.getX() - one.getX();
    delta_y = two.getY() - one.getY();
```

```
    if (delta_y == 0.0)
    {
        if (delta_x > 0.0) // case 0°
            return 0.0;
        else if (delta_x < 0) // case 180°
            return PI;
        else
            return 0.0; // SAME POINT
    }
    else if (delta_x == 0.0)
    {
        if(delta_y > 0.0) // case 90°
            return (PI/2.0);
        else return (1.5*PI); // case 270°
    }
    else
        angle = atan (delta_y/delta_x);
    // in case angle in 2nd or 3rd Quad, wont show up in atan()
    if (delta_x < 0)
        angle += PI;

    // change angle -20 to 340
    while(angle < 0.0)
        angle += 2.0*PI;
    // just an error check
    if(angle > 2.0*PI)
        cout << "\a::get_angle() returning value larger than 360!" <<
            endl;

    return angle;
}

// Returns the angle between 3 points in Radians.. i.e. angle defined
// between vector(one,two) & vector(two,three)
// CAUTION: Use only in the CCW sense!!!!!   i.e. this function will
// return the INTERNAL angle of the vertex at point 'two' in the
// CCW direction..... OR the CCW angle swept by vector(2,3)
// when rotated to align vector(2,1)
double get_angle(const PointClass& one, const PointClass& two,
                 const PointClass& three)
{
    double angle_1, angle_2, angle;

    angle_1 = ::get_angle(two, one);
    angle_2 = ::get_angle(two, three);

    angle = angle_1 - angle_2;
    // change angle -20 to 340
    while(angle > (2.0*PI))
        angle -= 2.0*PI;

    while(angle < 0.0)
        angle += 2.0*PI;
    // just an error check
    if(angle > 2.0*PI)
```

```
        cout << "\a::get_angle() returning value larger than 360!" <<
            endl;

    return angle;
}

//  Returns the distance between two points. May have a slight error
//  as the function sqrt, works with doubles.
double get_distance(const PointClass& one, const PointClass& two)
{
    double delta_x, delta_y;

    delta_x = two.getX() - one.getX();
    delta_y = two.getY() - one.getY();

    return sqrt(delta_x*delta_x + delta_y*delta_y);
}

double get_distance(const double one, const double two,
    const double three, const double four)
{
    double delta_x, delta_y;

    delta_x = three - one;
    delta_y = four - two;

    return sqrt(delta_x*delta_x + delta_y*delta_y);
}

// angle in degrees; can use the same PointList for input as well as
// output too
void rotate (const double angle,
             const int NumberOfPointsInList,
             const PointClass &referencePoint,
             const PointClass *InPointList,
             PointClass *OutPointList)
{
    int i;
    PointClass *LocalInPoints,
        LocalRefPoint(referencePoint);

    LocalInPoints = new PointClass [NumberOfPointsInList];

    // Work with a duplicate point list in case u're modifying the
    // same set of input points
    for(i = 0; i < NumberOfPointsInList; i++)
            LocalInPoints[i] = InPointList[i];

    // if angle is 0.. save time
    if (fabs(angle) <= DELTA)
    {
        for(i = 0; i < NumberOfPointsInList; i++)
            OutPointList[i] = LocalInPoints[i];

        delete [] LocalInPoints;
        return;
```

```
    }

    // get angle in radians
    double radians = angle*0.01745329251994;
    double cosValue = cos(radians);
    double sinValue = sin(radians);
    double temp_x, temp_y;

    // set new vertex locations
    for (i = 0; i < NumberOfPointsInList; i++)
    {
        temp_x = LocalInPoints[i].getX()*cosValue -
            LocalInPoints[i].getY()*sinValue -
            LocalRefPoint.getX()*cosValue +
            LocalRefPoint.getY()*sinValue +
            LocalRefPoint.getX();

        temp_y = LocalInPoints[i].getX()*sinValue +
            LocalInPoints[i].getY()*cosValue -
            LocalRefPoint.getX()*sinValue -
            LocalRefPoint.getY()*cosValue +
            LocalRefPoint.getY();

        OutPointList[i].set(temp_x, temp_y);
    }

    delete [] LocalInPoints;
    return;
}


void get_bounds (const int NumberOfPointsInList,
                 const PointClass *InPointList,
                 PointClass *Out2Points, // 2 PointClass array
                 double &area)
{
    int i;
    PointClass *LocalInPoints;

    LocalInPoints = new PointClass [NumberOfPointsInList];

    // Work with a duplicate point list in case u're modifying the
    // same set of input points
    for(i = 0; i < NumberOfPointsInList; i++)
            LocalInPoints[i] = InPointList[i];

    double ll_x = LocalInPoints[0].getX();
    double ll_y = LocalInPoints[0].getY();
    double ur_x = LocalInPoints[0].getX();
    double ur_y = LocalInPoints[0].getY();

    for (i = 1; i < NumberOfPointsInList; i++)
    {
        if(ll_x > LocalInPoints[i].getX())
            ll_x = LocalInPoints[i].getX();
        if(ll_y > LocalInPoints[i].getY())
```

```
            ll_y = LocalInPoints[i].getY();
        if(ur_x < LocalInPoints[i].getX())
            ur_x = LocalInPoints[i].getX();
        if(ur_y < LocalInPoints[i].getY())
            ur_y = LocalInPoints[i].getY();
    }

    Out2Points[0].set(ll_x, ll_y);
    Out2Points[1].set(ur_x, ur_y);

    area = (ur_x - ll_x)*(ur_y - ll_y);

    return;
}

boolean intersects(const PointClass& One, const PointClass& Two,
                   const PointClass& Three, const PointClass& Four)
{
    return intersects(One.getX(),One.getY(),Two.getX(),Two.getY(),
        Three.getX(),Three.getY(),Four.getX(),Four.getY());
}

boolean intersects(const PointClass& One, const PointClass& Two,
                   const PointClass& Three, const PointClass& Four,
                   PointClass& Intersection)
{
    double xintersect, yintersect;
    boolean returnval =
        intersects(One.getX(),One.getY(),Two.getX(),Two.getY(),
        Three.getX(),Three.getY(),Four.getX(),Four.getY(),
        xintersect, yintersect);
    if(returnval == TRUE)
    {
        Intersection.set(xintersect, yintersect);
        return TRUE;
    }
    return FALSE;
}

boolean intersects(double x1, double y1, double x2, double y2,
                   double x3, double y3, double x4, double y4)
{
    // A1.x + B1.y + C1 = 0
    double A1, B1, C1;
    A1 = y2 - y1;
    B1 = x1 - x2;
    C1 = x2*y1 - x1*y2;

    double R3 = A1*x3 + B1*y3 + C1;
    double R4 = A1*x4 + B1*y4 + C1;

    if( (fabs(R3) > DELTA)&&
        (fabs(R4) > DELTA)&&
        (sameSign(R3, R4) == TRUE) )
        return FALSE;
```

```
    // A2.x + B2.y + C2 = 0
    double A2, B2, C2;

    A2 = y4 - y3;
    B2 = x3 - x4;
    C2 = x4*y3 - x3*y4;

    double R1 = A2*x1 + B2*y1 + C2;
    double R2 = A2*x2 + B2*y2 + C2;

    if( (fabs(R1) > DELTA)&&
        (fabs(R2) > DELTA)&&
        (sameSign(R1, R2) == TRUE) )
        return FALSE;

    double denominator = A1*B2 - A2*B1;
    if(fabs(denominator) <= DELTA)
        return FALSE; // COLLINEAR POINTS
    return TRUE;
}

boolean intersects(double x1, double y1, double x2, double y2,
                   double x3, double y3, double x4, double y4,
                    double& x_intersection, double& y_intersection)
{
    // A1.x + B1.y + C1 = 0
    double A1, B1, C1;
    A1 = y2 - y1;
    B1 = x1 - x2;
    C1 = x2*y1 - x1*y2;

    double R3 = A1*x3 + B1*y3 + C1;
    double R4 = A1*x4 + B1*y4 + C1;

    if( (fabs(R3) > DELTA)&&
        (fabs(R4) > DELTA)&&
        (sameSign(R3, R4)==TRUE) )
        return FALSE;

    // A2.x + B2.y + C2 = 0
    double A2, B2, C2;

    A2 = y4 - y3;
    B2 = x3 - x4;
    C2 = x4*y3 - x3*y4;

    double R1 = A2*x1 + B2*y1 + C2;
    double R2 = A2*x2 + B2*y2 + C2;

    if( (fabs(R1) > DELTA)&&
        (fabs(R2) > DELTA)&&
        (sameSign(R1, R2) == TRUE) )
        return FALSE;

    double denominator = A1*B2 - A2*B1;
    if(fabs(denominator) <= DELTA)
```

```
        return FALSE; // COLLINEAR POINTS

    x_intersection = (-C1*B2 + B1*C2)/denominator;
    y_intersection = (-C2*A1 + C1*A2)/denominator;

    return TRUE;
}

inline boolean sameSign(double a, double b)
{
    // 0.0 is considered a +ve value here
    if((a >= 0.0)&&(b >= 0.0))
        return TRUE;
    else if((a < 0.0)&&(b < 0.0))
        return TRUE;

    else if(fabs(a - b) <= DELTA)
        return TRUE;
    else
        return FALSE;
}

inline boolean toggle(boolean& bool)
{
    if(bool == TRUE)
        bool = FALSE;
    else if (bool == FALSE)
        bool = TRUE;
    return bool;
}


/* Algorithm works for NON SELF OVERLAPPING polygons with
   STRAIGHT LINE EDGES in the COUNTER CLOCKWISE direction ONLY
   "Area of a Simple Polygon", Jon Rokne, Graphics Gems - 2 */
double area (const PointClass* vtx_array, const int num_in_array,
             const char* entry)
{
    if(num_in_array < 3)
    {
        cout <<"\aCant calculate area for polygon with vertices <
            3!\a"<< endl;
        return(0.0);
    }

    double polygonarea = 0.0; // initialize
    for (int i = 0; i < num_in_array-1; i++)
    {
        polygonarea += (vtx_array[i]).getX()*(vtx_array[i+1]).getY() -
            (vtx_array[i]).getY()*(vtx_array[i+1]).getX();
    }

    polygonarea +=
        vtx_array[num_in_array-1].getX()*vtx_array[0].getY() -
        vtx_array[num_in_array-1].getY()*vtx_array[0].getX();
```

```
    if(fabs(polygonarea) <= DELTA)
        return(0.0);

    polygonarea /= 2.0;

    if(polygonarea < 0.0)
    {
        cout <<"\aarea(): Polygon has been defined clock-wise!\a.. "
             << " function called from " << entry << endl;
    }

    return polygonarea;
}

// Returns a double value of the length of contact of 2 line segments
// only if the lines are collinear
double contactLength (const PointClass& one, const PointClass& two,
                      const PointClass& three, const PointClass& four)
{
    return contactLength(one.getX(), one.getY(), two.getX(),
        two.getY(), three.getX(), three.getY(), four.getX(),
        four.getY());
}


double contactLength(double x1, double y1, double x2, double y2,
                     double x3, double y3, double x4, double y4)
{
    double length = 0.0;
    double slope1;
    double slope2;
    boolean parallel = FALSE;

    // check if lines are parallel
    // If both lines are vertical
    if((fabs(x2-x1) <= DELTA) && (fabs(x4-x3) <= DELTA))
        parallel = TRUE;
    else
    {
        // if either one is not vertical, then lines are not parallel
        if((fabs(x2-x1) <= DELTA) || (fabs(x4-x3) <= DELTA))
        {
            parallel = FALSE;
            return (0.0);
        }
        slope1 = (y2 - y1)/(x2 - x1);
        slope2 = (y4 - y3)/(x4 - x3);
        if(fabs(slope1 - slope2) <= DELTA)
            parallel = TRUE;
        else if(fabs(slope1*slope2 + 1.0) <= DELTA)
            parallel = TRUE;
    }
    // exit if not parallel
    if(parallel == FALSE)
        return 0.0;
```

```
    double length12 = get_distance(x1,y1, x2,y2);
    double length34 = get_distance(x3,y3, x4,y4);

    // get max dist between any two points
    double l12 = get_distance(x1,y1, x2,y2);
    double l13 = get_distance(x1,y1, x3,y3);
    double l14 = get_distance(x1,y1, x4,y4);
    double l23 = get_distance(x2,y2, x3,y3);
    double l24 = get_distance(x2,y2, x4,y4);
    double l34 = get_distance(x3,y3, x4,y4);
    // initialize
    double max_length = l12;
    if(max_length < l13) max_length = l13;
    if(max_length < l14) max_length = l14;
    if(max_length < l23) max_length = l23;
    if(max_length < l24) max_length = l24;
    if(max_length < l34) max_length = l34;

    // eliminate non-overlap conditions
    if(max_length >= (length12+length34))
        return 0.0;

    length = length12 + length34 - max_length;

    return length;
}

boolean filterRedundantVertices(int *numVertices, PointClass
        **vtxArray)
{
    // Cannot have a polygon with 2 or less vertices!!
    if(*numVertices <= 2)
    {
        *numVertices = 0;
        delete [] (*vtxArray);
        return TRUE;
    }

    PointClass *tempVertexList = new PointClass[*numVertices];
    boolean *boolList = new boolean[*numVertices];
    int falseVertexCount = 0, i, validVertexCount;
    // Duplicate the vertices
    for(i = 0; i < *numVertices; i++)
    {
        // FALSE means this vertex is unique
        boolList[i] = FALSE;
        tempVertexList[i] = (*vtxArray)[i];
    }
    // Mark repeating vertices
    for(i = 0; i < *numVertices; i++)
    {
        int indexBefore = (i + *numVertices - 1) % *numVertices;
        int indexAfter =  (i + *numVertices + 1) % *numVertices;
        if(tempVertexList[i] == tempVertexList[indexBefore])
        {
            boolList[i] = TRUE;
```

```
            falseVertexCount++;
        }
    }

    // Delete the repeating vertices and re initialize array
    if(falseVertexCount != 0)
    {
        delete [] (*vtxArray);
        (*vtxArray) = new PointClass[*numVertices - falseVertexCount];

        validVertexCount = 0;
        for(i = 0; i < *numVertices; i++)
        {
            if(boolList[i] == FALSE) // valid vertex
                (*vtxArray)[validVertexCount++] = tempVertexList[i];
        }

        *numVertices -= falseVertexCount;
        falseVertexCount = 0;

        delete [] boolList;
        boolList = new boolean[*numVertices];

        delete [] tempVertexList;
        tempVertexList = new PointClass[*numVertices];

        for(i = 0; i < *numVertices; i++)
        {
            // FALSE means this vertex is unique
            boolList[i] = FALSE;
            tempVertexList[i] = (*vtxArray)[i];
        }
    }

    for(i = 0; i < *numVertices; i++)
    {
        int indexBefore = (i + *numVertices - 1) % *numVertices;
        int indexAfter =  (i + *numVertices + 1) % *numVertices;
        double interiorAngle =
            ::get_angle(tempVertexList[indexBefore],
                  tempVertexList[i], tempVertexList[indexAfter]);
        // if angle is close to 0 or 180...
        if( (fabs(interiorAngle) < DELTA) ||
            (fabs(interiorAngle - PI) < DELTA) ||
            (fabs(interiorAngle - 2.0*PI) < DELTA)
          )
        {
            boolList[i] = TRUE;
            falseVertexCount++;
        }
    }

    if(falseVertexCount == 0)
        return TRUE;

    // else form the new vertex list for checking again
```

```
    delete [] (*vtxArray);
    *numVertices -= falseVertexCount;
    (*vtxArray) = new PointClass[*numVertices];
    validVertexCount = 0;
    for(i = 0; i < ((*numVertices) + falseVertexCount); i++)
    {
        if(boolList[i] == FALSE) // valid vertex
            (*vtxArray)[validVertexCount++] = tempVertexList[i];
    }

    filterRedundantVertices(numVertices, vtxArray);

    delete [] boolList;
    delete [] tempVertexList;

    return TRUE;
}
```

FILE: MAIN.CPP

```
/**
 *  C++ implementation of the Nesting Algorithm.
 *  author: Ananda A. Debnath
 *  Masters Thesis in Mechanical Engineering
 *  November 13, 1997
 */
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <conio.h>
#include "geometry.hpp"
#include "dxf.hpp"
#include "nest.hpp"
#include <time.h>
#include <dos.h>

// Globals
boolean         DEBUG = FALSE;
FILE*           errorFile = stdout;
double          AreaCoefficient = 1.0,
                LeftShadowAreaCoefficient = -1.0,
                BottomAreaCoefficient = -1.0,
                ContactLengthSquaredAreaCoefficient = 0.5;

int main (void)
{
    // An array of StockClass instances
    StockClass      *Stock;
    // An array of PartClass instances
```

```
PartClass       *Part;
// An array of features extracted from all the Parts
FeatureArray    PartFeatureArray;
// An array containing all the points against which nesting any
//  part shows a bad status
PointArray      badPoints;
// An array of the Stock Features selected as target features
FeatureArray    selectedStockFeatures;
boolean         doneNesting = FALSE;
int             StockUpdateNumber;
clock_t         begin,
                finish;
int             num_of_parts,
                num_of_stocks;
unsigned int    Alignment;
FeatureClass    bestPartFeature,
                bestStockFeature;
double          bestScore;
status          current_status,
                rotateStatus,
                translateStatus;
char            input_file_name[50]  = "Test1.dxf",
                output_file_name[50];
char            temp_string[20], dummy[30];
double          TotalPartArea,
                InitialStockArea,
                FinalStockArea;


cout << "Enter name of input file <" << input_file_name << "> " <<
   flush;
gets(temp_string);
if(temp_string[0] != '\0')
    strcpy(input_file_name, temp_string);

cout << "Enter name of process ouput file <CONSOLE> " << flush;
gets(temp_string);
if(temp_string[0] != '\0')
    errorFile = fopen(temp_string, "w");

cout << "Enter Area Coefficient                    <"<<
    AreaCoefficient<<"> " << flush;
gets(temp_string);
if(temp_string[0] != '\0')
    AreaCoefficient = atof(temp_string);

cout << "Enter Left_Shadow_Area Coefficient        <" <<
    LeftShadowAreaCoefficient << "> " << flush;
gets(temp_string);
if(temp_string[0] != '\0')
    LeftShadowAreaCoefficient = atof(temp_string);

cout << "Enter Bottom_Shadow_Area Coefficient      <" <<
    BottomAreaCoefficient << "> " << flush;
gets(temp_string);
if(temp_string[0] != '\0')
```

```
    BottomAreaCoefficient = atof(temp_string);

cout << "Enter Contact_Length_Squared Coefficient <" <<
    ContactLengthSquaredAreaCoefficient << "> " << flush;
gets(temp_string);
if(temp_string[0] != '\0')
    ContactLengthSquaredAreaCoefficient = atof(temp_string);

begin = clock();
// Get the number of polylines in the file
current_status =
    getPolylinesInFile(input_file_name, num_of_stocks,
        num_of_parts);
if(current_status != NestOK)
{
    cout << "\aError in getPolylinesInFile()\nAborting!" << endl;
    exit(1);
}

// allocate those many Parts
Part = new PartClass [num_of_parts];
if(!Part)
{
    cout << "\aError allocating run-time memory to \
          Parts\nAborting!" << endl;
    exit(1);
}

Stock = new StockClass [num_of_stocks];
if(!Stock)
{
    cout << "\aError allocating run-time memory to
       Stocks\nAborting!" << endl;
    exit(1);
}

// Create the parts from the DXF file
if(DXFin(Part, num_of_parts, Stock, num_of_stocks,
         input_file_name) != NestOK)
{
    cout << "Error in DXFin()\a" << endl;
    exit(1);
}
// Extract and generate all the 'corner-PartFeatureArray' for each
//  of the parts
for(int j = 0; j < num_of_parts; j++)
    cornerFeatures(Part[j], PartFeatureArray);
// To be called in the loop which actually performs the
//  nesting....
StockUpdateNumber = 0;
InitialStockArea = Stock[0].get_area();
int noOfDoLoops = 0;
clock_t start, end;

do
{
```

```
if(selectStockFeatures(selectedStockFeatures, badPoints,
    Stock[0]) != NestOK)
{
    fprintf(errorFile,"\nRan out of stock PartFeatureArray. \
          ");
    fprintf(errorFile,"\n  Normal termination of nesting \
        loop.");
    break;
}

start = clock();
if(evaluateMatches (PartFeatureArray, selectedStockFeatures,
    bestPartFeature, bestStockFeature, &Alignment, &bestScore)
    != NestOK)
{
    fprintf(errorFile,"\n  Evaluate matches returned non-
        NestOK!!");
    break;
}
end = clock();
fprintf(errorFile, " %.2f seconds",
    ((end-start)/(double)CLOCKS_PER_SEC) );

if(bestScore <= BADSCORE)
{
    for(int k = 0; k < selectedStockFeatures.getLength(); k++)
    {
        PointClass badPoint =
              selectedStockFeatures.getFeature(k).
                  getVertexValue(1);
        fprintf(errorFile,"\n  Adding bad point");
        badPoint.print(FALSE, errorFile);
        badPoints.add(badPoint);
    }
    doneNesting = FALSE;
    noOfDoLoops++;
    continue;
}
else
{
    char tempFileName[20];
    if(align(Alignment, bestPartFeature, rotateStatus,
            translateStatus, Alignment, bestStockFeature) !=
            NestOK)
    {
        fprintf(errorFile, "\n  Could not perform alignment of
            %s\n",
            bestPartFeature.getOwner()->get_name(dummy));
        fprintf(errorFile, "  Error in aligning of\n");
        bestPartFeature.print(FALSE, errorFile);
        fprintf(errorFile, "against\n");
        bestStockFeature.print(FALSE, errorFile);
        break;
    }

    if(Stock[0].updateProfile(*(bestPartFeature.getOwner()))
```

```
                                        != NestOK)
        {
            fprintf(errorFile, "\n  Error in stock profile update!
                Exiting\n");
            DXFout((PartClass*)bestPartFeature.getOwner(), 1,
                    Stock, 1, "updterr.dxf");
            bestPartFeature.getOwner()->print(TRUE, errorFile);
            break;
        }

        fprintf(errorFile, "\n Nested %s ...",
            bestPartFeature.getOwner()->get_name(dummy));
        PartFeatureArray.remove(bestPartFeature);
        StockUpdateNumber++;
    }
    // PartFeatureArray.print();
    if(PartFeatureArray.getLength() == 0)
    {
        fprintf(errorFile,
            "\nAll parts nested. Normal termination of nesting
               loop\n");
        doneNesting = TRUE;
        break;
    }
    noOfDoLoops++;
} while(doneNesting == FALSE);

PointClass origin;
char String[100];
char areaCoeff[30], lShadCoeff[30], bShadCoeff[30], clCoeff[30];
sprintf(areaCoeff, "%.2f*Area", AreaCoefficient);
if(LeftShadowAreaCoefficient >= 0.0)
    sprintf(lShadCoeff, "+ %.2f*LeftShadow",
        LeftShadowAreaCoefficient);

else
    sprintf(lShadCoeff,"- %.2f*LeftShadow",
        (-1.0*LeftShadowAreaCoefficient));
if(BottomAreaCoefficient >= 0.0)
    sprintf(bShadCoeff,"+ %.2f*BottomShadow",
        BottomAreaCoefficient);
else
    sprintf(bShadCoeff,"- %.2f*BottomShadow",(-
        1.0*BottomAreaCoefficient));
if(ContactLengthSquaredAreaCoefficient >= 0.0)
    sprintf(clCoeff,"+ %.2f*CL^2",
        ContactLengthSquaredAreaCoefficient);
else
    sprintf(clCoeff,"- %.2f*CL^2",
        (-1.0*ContactLengthSquaredAreaCoefficient));
sprintf(String, "%s %s %s %s",areaCoeff, lShadCoeff, bShadCoeff,
    clCoeff);
sprintf(output_file_name, "%d%d%d%d.dxf", a, b, c, d);
DXFout(Part, num_of_parts, Stock[0], badPoints,
    String, origin, output_file_name);
```

```
    fprintf(errorFile, "\nWrote Nest.dxf\n");
    finish = clock();
    fprintf(errorFile, "\nScore = %s", String);
    fprintf(errorFile, "\nTotal time elapsed            = %.2f
        seconds", ((finish-begin)/(double)CLOCKS_PER_SEC) );
    TotalPartArea = 0.0;
    for(j = 0; j < num_of_parts; j++)
        TotalPartArea += Part[j].get_area();
    fprintf(errorFile, "\nTotal part area               = %.2f",
        TotalPartArea);
    FinalStockArea = Stock[0].get_area();
    fprintf(errorFile, "\nTotal stock area used         = %.2f",
        (InitialStockArea-FinalStockArea));
    fprintf(errorFile, "\n(TotalPartArea/UsedStockArea) = %.2f",
        TotalPartArea/(InitialStockArea-FinalStockArea));

    // End of loop which actually performs the nesting....
    // Clean up
    delete [] Part; Part = NULL;
    delete [] Stock; Stock = NULL;
    cout << "16" << endl;
    return 0;
}
```

FILE: NEST.CPP

```
#include "geometry.hpp"
#include "nest.hpp"
#include <conio.h>
#include <string.h>
#include <stdlib.h>

extern int          errorCount;
extern boolean      DEBUG;
extern FILE*        errorFile;
extern double       AreaCoefficient,
                    LeftShadowAreaCoefficient,
                    BottomAreaCoefficient,
                    ContactLengthSquaredAreaCoefficient;

// Extracts all the 'corner features' of the specified polygon and
//  stores it in the specified FeatureArray
status cornerFeatures(PolygonClass& polygon, FeatureArray& array)
{
    for(int i = 0; i < polygon.Number_Of_Vertices; i++)
    {
        int j, k;
        j = (i+1) % polygon.Number_Of_Vertices;
        k = (i+2) % polygon.Number_Of_Vertices;
        array.add(polygon, polygon.Vertex[i], polygon.Vertex[j],
```

```
            polygon.Vertex[j], polygon.Vertex[k], polygon.Angle[j]);
    }
    return NestOK;
}

// Extracts all the 'convex hull features' of the specified polygon
//  and stores it in the specified FeatureArray... friend to
//  PolygonClass
status convexHullFeatures(const PolygonClass& polygon, FeatureArray&
    array)
{
    return NestOK;
}

// Selects a stock feature from the specified stock...
// SHOULD ALSO INCLUDE A METHOD BY WHICH WE CAN DEFINE OTHER FEATURES
// OF NON CONTINUOUS EDGES
status selectStockFeatures(FeatureArray& stockfeatures,
                           const PointArray& badPointArray,
                           StockClass& stock)
{
    // index of the vertex with the smallest y co-ordinate (and
    //  smallest x coord. in case there is more than 1 smallest y
    //  coord.)
    //  I. E. Select the bottom-most and left most point of the stock
    //  (in order of priority)
    int bottom_most_index;
    int prev_index,
        next_index,
        start_index;
    start_index = 0;

    while(1)
    {
        if(badPointArray.contains(stock.Vertex[start_index]) == TRUE)
        {
            start_index++;
            if(start_index == stock.Number_Of_Vertices)
            {
                cout << "\aCould not find start point in
                    selectStockFeature" << endl;
                badPointArray.print(FALSE, errorFile);
                return NestFAILED;
            }
            else
                continue;
        }
        else
            break;
    }

    bottom_most_index = start_index;
    for(int i = 1; i < stock.Number_Of_Vertices; i++)
    {
        int nextVertex = (start_index + i) % stock.Number_Of_Vertices;
        if(badPointArray.contains(stock.Vertex[nextVertex]) == TRUE)
```

```
            continue;
        if((stock.Vertex[bottom_most_index].getY() -
            stock.Vertex[nextVertex].getY()) > DELTA)
        {
            bottom_most_index = nextVertex;
            continue;
        }

        if(fabs(stock.Vertex[bottom_most_index].getY() -
            stock.Vertex[nextVertex].getY()) <= DELTA)
        {
            if(stock.Vertex[bottom_most_index].getX() >
                stock.Vertex[nextVertex].getX())
                bottom_most_index = nextVertex;
        }
    }

    prev_index = (bottom_most_index-1+(stock.Number_Of_Vertices))%
        (stock.Number_Of_Vertices);
    next_index = (bottom_most_index+1)%(stock.Number_Of_Vertices);

    stockfeatures.removeAll();
    if(stockfeatures.add(stock, stock.Vertex[prev_index],
            stock.Vertex[bottom_most_index],
            stock.Vertex[bottom_most_index],
            stock.Vertex[next_index]) != NestOK)
        return NestFAILED;
    return NestOK;
}


/* Evaluates the matches between features
    Parameters:
        partFeatureArray: Reference to a FeatureArray object which
                          contains the currently available part
                          features for matching against the specified
                          stock feature.
        stockfeature:     Reference to the stock feature to match
                          against
        best_index:       Pointer to contain the index of the best
                          match in the feature array
        best_alignment:   Pointer to contain the edge-to-edge case of
                          the best alignment
        best_score:       Pointer to contain the best score generated
                          by the alignments
*/
status evaluateMatches (const FeatureArray& partFeatureArray,
                        const FeatureArray& stockFeatureArray,
                        FeatureClass&       best_Part_Feature,
                        FeatureClass&       best_Stock_Feature,
                        unsigned int        *best_alignment,
                        double              *best_score)
{
    FeatureClass    currentStockFeature,
                    currentPartFeature;
    unsigned int    betterEdgeIndex,
```

```
                i,
                j;
status          action,
                currentPartRotated,
                currentPartTranslated;
double          currentScore;
                // bestScoreSoFar;

*best_score = BADSCORE;
if((stockFeatureArray.getLength() <= 0) ||
   (partFeatureArray.getLength() <= 0))
{
    cout << "\aCannot have feature array's of length 0 or less" <<
        endl;
    return NestFAILED;
}

for(i = 0; i < stockFeatureArray.getLength(); i++)
{
   currentStockFeature.construct(stockFeatureArray.getFeature(i));
    StockClass *OwnerStock =
     (StockClass*)currentStockFeature.Owner;
    for(j = 0; j < partFeatureArray.getLength(); j++)
    {
        currentPartFeature.construct(
            partFeatureArray.getFeature(j));
        PartClass *OwnerPart =
        (PartClass*)currentPartFeature.Owner;
        // for each part feature...
        action = align(0, currentPartFeature, currentPartRotated,
            currentPartTranslated, 0, currentStockFeature);
        if(action == NestOK)
        {
            currentScore = calculateScore(*OwnerPart, *OwnerStock,
                currentStockFeature, currentPartFeature);
            if((*best_score) < currentScore)
            {
                (*best_score) = currentScore;
                (*best_alignment) = 0;
                best_Part_Feature.construct(currentPartFeature);
                best_Stock_Feature.construct(currentStockFeature);
            }
        }
        action = align(1, currentPartFeature, currentPartRotated,
            currentPartTranslated, 1, currentStockFeature);
        if(action == NestOK)
        {
            currentScore = calculateScore(*OwnerPart, *OwnerStock,
                currentStockFeature, currentPartFeature);
            if((*best_score) < currentScore)
            {
                (*best_score) = currentScore;
                (*best_alignment) = 1;
                best_Part_Feature.construct(currentPartFeature);
                best_Stock_Feature.construct(currentStockFeature);
            }
```

```
            }
        }
    }

    return NestOK;
}

// This function aligns the specified edge of the specified part
//  (feature) against the specified edge of the specified stock
//  feature
// THE VALUE OF THE EDGE VARIABLES MAY BE ONLY 0 OR 1 AS THIS TYPE OF
//  FEATURE IS DEFINED ONLY BY TWO EDGES
// The function does the foll:
// 1. Rotates the part so that the part edge is parallel to the stock
//    edge.
// 2. Moves the part so that the edges touch each other in any of 4
//    configurations..... (see config4.dxf)
// 3. Increment the position of the part along the feature edge
//    until the part is completely inside the part
status align(const int part_feature_edge, const FeatureClass&
             part_feature, status& part_rotated_status, status&
             part_translated_status, const int stock_feature_edge,
const
             FeatureClass& stock_feature)
{
    if((part_feature_edge < 0) || (part_feature_edge > 1) ||
        (stock_feature_edge > 1) || (stock_feature_edge > 1))
    {
        cout << "\aCant have and edge value outside [0,1]!" << endl;
        return NestFAILED;
    }
    // Legal MatchCase's can be only [1, 2, 3, 4]
    int MatchCase = 5;

    if((part_feature_edge == 0) && (stock_feature_edge == 0))
        MatchCase = 0;
    else if((part_feature_edge == 1) && (stock_feature_edge == 1))
        MatchCase = 1;

    // The current vector angles of the edges in Radians.
    double current_angle,    // absolute angle of the part edge
        target_angle,        // absolute angle of the stock edge
        delta_angle,         // their difference
        x_increment = 0.0,
        y_increment = 0.0;
    boolean contained = TRUE;


    // Initialize... innocent till proven guilty ;)
    part_rotated_status = NestOK;
    part_translated_status = NestOK;

    PartClass* Owner_Part = (PartClass*)(part_feature.Owner);

    switch (MatchCase)
    {
```

```
case 0:
    // Part feature edge 0 is selected to match up against
    // Stock feature edge 0
    current_angle =
        ::get_angle(*(part_feature.PointPointer[1]),
            *(part_feature.PointPointer[0]));
    target_angle  =
        ::get_angle(*(stock_feature.PointPointer[1]),
            *(stock_feature.PointPointer[0]));
    delta_angle = target_angle - current_angle;
    // Convert to degrees
    delta_angle /= 0.01745329251994;
    // Rotate only if the angles are different
    if(current_angle != target_angle)
        part_rotated_status = Owner_Part->rotate(delta_angle,
            *(part_feature.PointPointer[1]));
    // As of *NOW* all stock features are *ADJACENT EDGES*,
    //  hence moving the part from
    //  part_feature.PointPointer[0]
    //  to stock_feature.PointPointer[1] will almost always
    //  cause the part to be outside the stock... so
    //  *currently*, am settling for
    //  part_feature.PointPointer[1] to
    //  stock_feature.PointPointer[1]

    part_translated_status =
        Owner_Part->translate(*(part_feature.PointPointer[1]),
        *(stock_feature.PointPointer[1]));
    // Check if the part is completely enclosed in the stock
    contained = stock_feature.Owner->contains(*Owner_Part);
    if(contained != TRUE)
    {
        x_increment = stock_feature.PointPointer[0]->getX()
            - part_feature.PointPointer[1]->getX();
        x_increment /= 50.0;
        y_increment = stock_feature.PointPointer[0]->getY()
            - part_feature.PointPointer[1]->getY();
        y_increment /= 50.0;
        int i = 0;
        while( i < 50)
        {
            part_translated_status =
                Owner_Part->translate(0.0,0.0, x_increment,
                    y_increment);
            i++;
            contained = stock_feature.Owner->
                contains(*Owner_Part);
            if(contained == TRUE)
                break;
        }
        if(contained != TRUE)
            return NestFAILED;
    }
    break;

case 1:
```

```
        // Part feature edge 1 is selected to match up against
        // Stock feature edge 1
        current_angle =
            ::get_angle(*(part_feature.PointPointer[2]),
                *(part_feature.PointPointer[3]));
        target_angle  =
            ::get_angle(*(stock_feature.PointPointer[2]),
                *(stock_feature.PointPointer[3]));
        delta_angle = target_angle - current_angle;
        // Convert to degrees
        delta_angle /= 0.01745329251994;
        if(current_angle != target_angle)
            part_rotated_status = Owner_Part->rotate(delta_angle,
                *(part_feature.PointPointer[2]));

        // As of *NOW* all stock features are *ADJACENT EDGES*,
        //  hence moving the part from
        //  part_feature.PointPointer[3]
        //  to stock_feature.PointPointer[2] will almost always
        //  cause the part to be outside the stock... so
        //  *currently* am settling for
        //  part_feature.PointPointer[1] to
        //  stock_feature.PointPointer[1]
        part_translated_status =
            Owner_Part->translate(*(part_feature.PointPointer[2]),
            *(stock_feature.PointPointer[2]));
        contained = stock_feature.Owner->contains(*Owner_Part);
        if(contained != TRUE)
        {
            x_increment = stock_feature.PointPointer[3]->getX()
                - part_feature.PointPointer[2]->getX();
            x_increment /= 50.0;
            y_increment = stock_feature.PointPointer[3]->getY()
                - part_feature.PointPointer[2]->getY();
            y_increment /= 50.0;
            int i = 0;
            while( i < 50)
            {
                part_translated_status =
                    Owner_Part->translate(0.0,0.0, x_increment,
                        y_increment);
                i++;
                contained = stock_feature.Owner->
                    contains(*Owner_Part);
                if(contained == TRUE)
                    break;
            }
            if(contained != TRUE)
                return NestFAILED;
        }
        break;

    default:
        cout << "\aIllegal align operation called for!!" << endl;
        break;
}
```

```
    return NestOK;
}

double calculateScore (const PartClass& part, const StockClass& stock,
    const FeatureClass& partFeature, const FeatureClass& stockFeature)
{
    double score;

    // Score is (Area - LeftShadow - UnderShadow + Contact_Length^2)
    score = AreaCoefficient*part.get_area();
    score += LeftShadowAreaCoefficient*stock.leftShadow(part);
    score += BottomAreaCoefficient*stock.underShadow(part);
    double contact_length =
        contactLength(*(partFeature.PointPointer[0]),
            *(partFeature.PointPointer[1]),
            *(stockFeature.PointPointer[0]),
            *(stockFeature.PointPointer[1]));
    contact_length +=
        contactLength(*(partFeature.PointPointer[2]),
            *(partFeature.PointPointer[3]),
            *(stockFeature.PointPointer[2]),
            *(stockFeature.PointPointer[3]));
    score += ContactLengthSquaredAreaCoefficient*
        contact_length*contact_length;
    return score;
}
```