Point-in-Polygon Detection

by
Jared Petker

A senior thesis submitted to the faculty of

The University of California, Merced

In partial fulfillment of the requirements for the degree of

Bachelor of Science

School of Natural Sciences

University of California, Merced

April 2010

Abstract

Point-in-Polygon Detection

Jared Petker

School of Natural Sciences

Bachelor of Science

The Point-in-Polygon problem involves determining whether a point in a two-dimensional plane resides inside, outside, or on the boundary of a given polygon. This topic is a very relevant and well-studied topic in several fields of research – such as computer graphics and computer vision. Here I develop and implement an algorithm with roots in the "grid-method". This algorithm consists of a pre-processing, sorting method, as well as a method for querying points for polygon inclusion. This search method will be shown to have run-times on the order of O(m*log(n)), with *m* being the number of query points and *n* being the number of edges representing the polygon. The algorithm is coded in Matlab and the results of this algorithm's performance versus Matlab's built-in "inpolygon" method will be presented.

Contents

Table of Contents

Section 1

Introduction

As humans, we have certain luxuries that a computer alone does not contain, with one of these luxuries being the ability to see. With the want to create a computer system which is able to mimic the human's visual system, the area in computer science called computer vision was born. This is a heavily studied area in computer science encompassing areas of research exploring topics such as object recognition and path planning. However, computers fail to compare to humans even at low level types of observations, such as determining if a point is inside a polygon. Many algorithms have been devised for a computer to perform point-in-polygon detection, many of them having run-times along the order of O(m*n) (with *m* being the number of query points for the detection and *n* being the number of edges which define the polygon; from here on these two values will be represented by *m* and *n* respectively). For simple implementations, and those which may not rely on speed, any one of these types of algorithms would be respectable. Though in cases where speed may be of importance, more intricate algorithms may be required.

An example which motivates the need for our new point-in-polygon detection algorithm can be explained as follows. Imagine the classical modeling of an atom, such as hydrogen, in a 2-D plane with a polygon representing its outermost boundary. This region can be defined rigorously in dynamical systems theory using stable and unstable manifolds, which are generalizations of the familiar WHAT

WAS HERE. Inside of this boundary we have an electron which is free to move around within the atom until ionization occurs when it escapes from the defined boundary. To completely consider every initial condition this electron could take on, we may want to do batch calculations of electron trajectories by placing many electrons within this boundary with different initial positions and conditions. Also, to model this situation accurately we may want to use millions of different initial conditions as well as thousands of edges for defining the polygonal bounds of the atom. At each time step we may want to compute how many electron trajectories are still left inside of the atom. Here, by simply using one of the early noted algorithms, these computations could take an exuberant amount of time; however, we have devised an algorithm which will pre-process the initial polygon by using our own sorting method and a searching method which can be used at each time step. This searching method will be shown to have a run-time on the order of O(m*log(n)), a huge improvement over the previously mentioned "simple" algorithms. This algorithm is programmed in Matlab's programming environment and will be tested against Matlab's built-in point-in-polygon detection algorithm "inpolygon" which is believed to have a run-time of O(m*n).

Section 2

The Algorithm

As explained, this algorithm consists of two steps – a preprocessing and searching step. Both steps are governed by their own intricacies and algorithms but work towards the goal of point-in-polygon detection. First the pre-processing will be outlined and explained, followed by the searching method.

i. Pre-Processing

Our algorithm's pre-processing has its roots in what is referred to as the "gridding method" for pre-processing a polygon (REF). In this method a bounding box is placed around the polygon and a grid is created inside of this bounding box as seen in figure 1. Data is coached for each grid cell containing information such as if its corners lie inside or outside of the polygon as well as which edges cross the cell - which is imperative information used for point-in-polygon queries. When a point for in-polygon detection is queried the grid cell which contains this point is first found, (this query point being the open circle in figure 1). The next step is to draw a line from the point to one of the corners of the grid cell, shown by the arrow pointing from the open circle to the corner of the cell which contains it. The number of times this line crosses edges of the polygon will determine whether the point lies inside

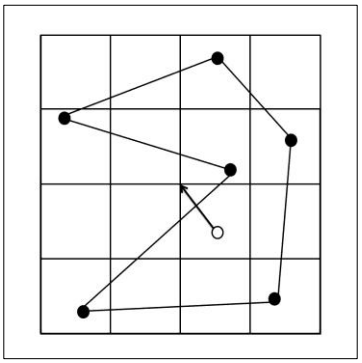

Figure 1: A basic representation of the gridding algorithm.

or outside of the polygon. The question for this algorithm comes with how large (or small) to make each grid cell. If the grid cells are too large then many edges may need to be checked for every point query, although too small of grid cells will create a huge amount of trivial cells, which also may not be desired.

Our algorithm improves on the gridding method in many ways but begins the same – by surrounding the polygon with a bounding box. However, unlike the gridding method, our algorithm does not place a simple grid inside of this bounding box. Instead, a series of binary cuts are done onto the bounding box through recursion, creating 2 smaller cells to deal with at each binary cut. Cuts are made onto the current cell perpendicular to the edge of the box that is longest. These cuts are made onto the polygon until one of three situations occurs:

1. When there are no edges of the polygon inside of or crossing the current cell, the cell is then considered a terminal cell. Essentially, the cell in this case is empty, and any other partitions conducted on this cell would produce a higher amount of trivial cells
2. When there is at most one vertex inside of the current cell with at most two edges crossing the boundaries of the cell, no more partitions are done onto this cell. Partitioning further would not eliminate this case from existing.
3. When there are no vertexes of the polygon inside of the current cell with at most one edge of the polygon crossing the cell, partitioning is also considered complete by following the same reasoning presented by case 2.

When one of these three conditions is met, no more cuts are done on that current cell and the algorithm continues to run and go through recursion. At each step of recursion while the algorithm is running, a binary tree structure is being created containing pertinent data for the pre-processing as well as the following search. The data which is stored inside each node of the tree is listed here:

- The cell's right and left child.
- The position and direction of the cut that splits that cell into two (if the cell is indeed cut).

- Whether the entire cell is inside or outside the polygon, or neither.
- Information about the edges which lie inside of or pass through the cell (if it is a terminal node).

This data is then used by the search function for determining if a query point is found inside or outside of the polygon. Below is a simple non-convex polygon which shows how these binary cuts by the pre-processing are made from start to finish.

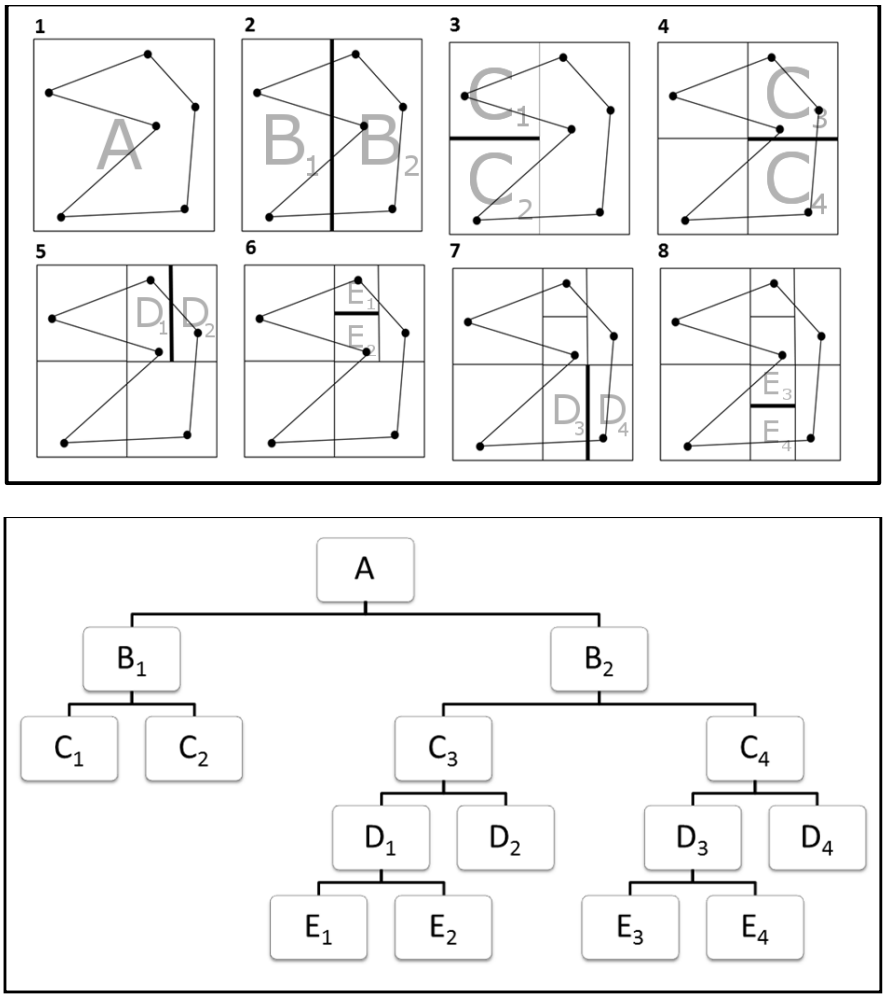

Figure 2: An example polygon is shown here (on top) with the pre-processing being conducted on it. At every step the bold line represents the new binary cut being done on the current cell in question and labels representing the node in the tree that that cell represents. The bottom figure represents the tree created by this polygon.

The above is of course a simple example of the implementation of our algorithm. However, a polygon which has so few edges would not benefit as much from this algorithm as a polygon with thousands of edges. In Figure 3 a ten thousand sided circle can be seen with the pre-processing already

conducted on it. What is not visible in figure 3 (and will be seen in figure 4) is that the boundary of the circle is actually slightly jagged so as to introduce some fine structure to the polygon. This helps in displaying how the terminal grid cells arrange themselves along the edges of the polygon.

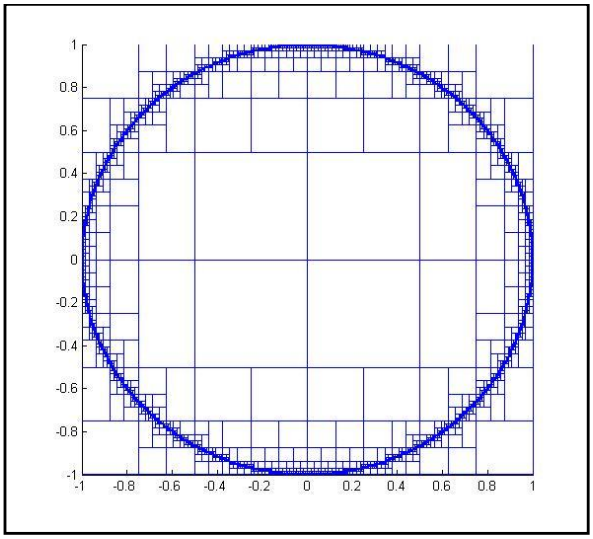

Figure 3: A polygon with 10,000 edges is shown here with the gridding created by the pre-processing shown overlaid.

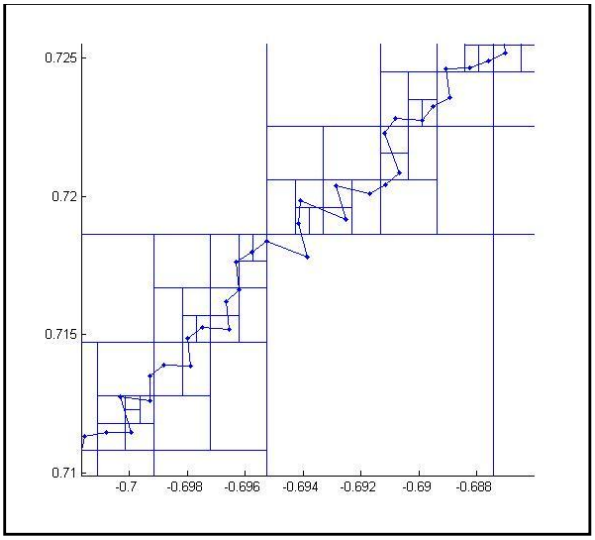

Figure 4: The fine structure around the edges of the polygon is seen here

Some of the advantages of our algorithm over the previously described "gridding method" can be seen. It is clear here that unlike the "gridding method" our algorithm ends up creating smaller cells which are for the most part localized along the boundaries of the polygon. What is gained by this approach is that for a completely random distribution of query points, the query point will be found inside a cell that is already defined as entirely inside or entirely outside of the polygon, making searches extremely fast. The fine structure of the cell formations along the boundary of the polygon can be seen above in figure 4 where several of this algorithm's terminating conditions can be seen.

ii. Search

The searching method of our algorithm utilizes the binary tree structure which is returned by the pre-processing. This binary tree only needs to be computed once for the polygon and can be used from then on without alteration so long as the polygon goes unaltered as well. This makes the algorithm very useful for cases where the polygon remains unchanged while being queried for large sets of point-in-polygon detections. The entire journey of a single query point through the main loop of the function occurs as follows:

a. Binary Search.

A binary search is conducted on the binary tree structure to determine which cell the query point is found in. The search begins at the top node of the tree, which contains the main bounding box surrounding the entire polygon as shown in figure 5. If the point in question is found inside of this bounding cell then we look to that node's left and right child's respective node. In each child's node we check to see which cell contains query point and move to this node in the tree where this tree traversal continues. Once the search reaches a terminal node which contains no children, the terminal cell containing the query point has been found and the next step of the search can be conducted. Figure 5 shows the traversal through the tree as well as a visual walkthrough of the polygon search.

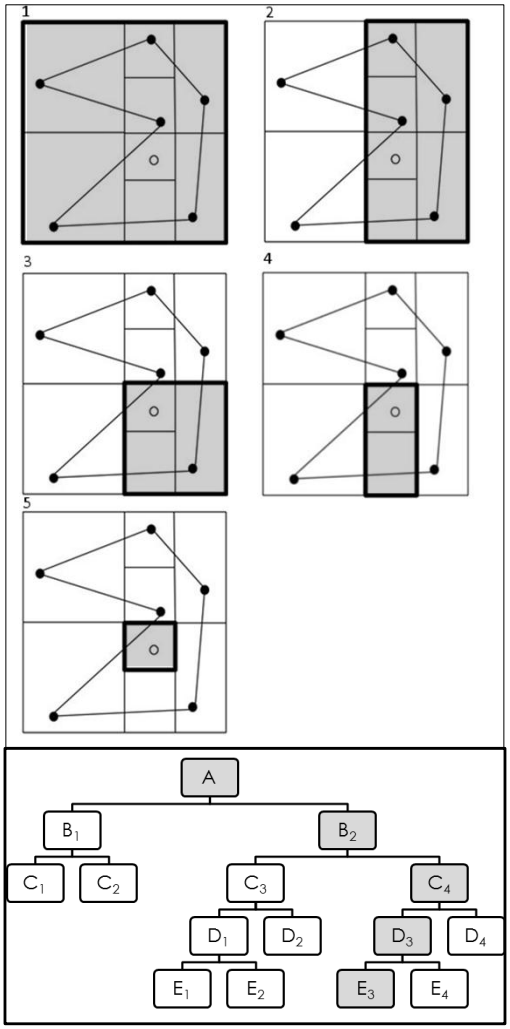

Figure 5: (Top) A walkthrough of a query point (open circle) being found by querying the tree (bottom) on where to traverse to. Gray indicates the nodes where traversal is occurring.

b. Terminal Node's Position

There are several things that can be done now to determine the query point's location, the first of which is what most commonly occurs. In the pre-processing we defined if each terminal cell was entirely inside the polygon entirely outside the polygon or neither. If the cell is defined as entirely inside or entirely outside the polygon, then we know immediately whether the point is inside or outside of the polygon. In the case where the cell's position is not entirely inside or entirely outside of the polygon, there is a final check to conduct in order to determine where the point lies.

c. Cross Product Checks

If this step is reached in locating the query point's position then some of the structure from the polygon itself must be used to finish the process. One piece of data which is required is the orientation of the polygon. The orientation of the polygon is solely dependent on how the points defining the polygon were ordered. When traversing the points defining the polygon, the points are listed in the clockwise or counter-clockwise direction, thus defining the polygon's orientation. Each orientation is also tracked by considering a vector **n** which is pointing either into the polygon's plane or out of it (where a clockwise orientation may be tagged by a vector pointing into the plane, and a counter-clockwise orientation would result in a vector pointing out of the plane). Now we arrive at one of three cases dealing with how the edges of the polygon may be passing through the terminal cell.

Case 1: 1 Edge Crossing

This first case is also the most trivial of the three cases. Here we simply have one edge which is crossing the terminal node's cell as seen in figure 6.

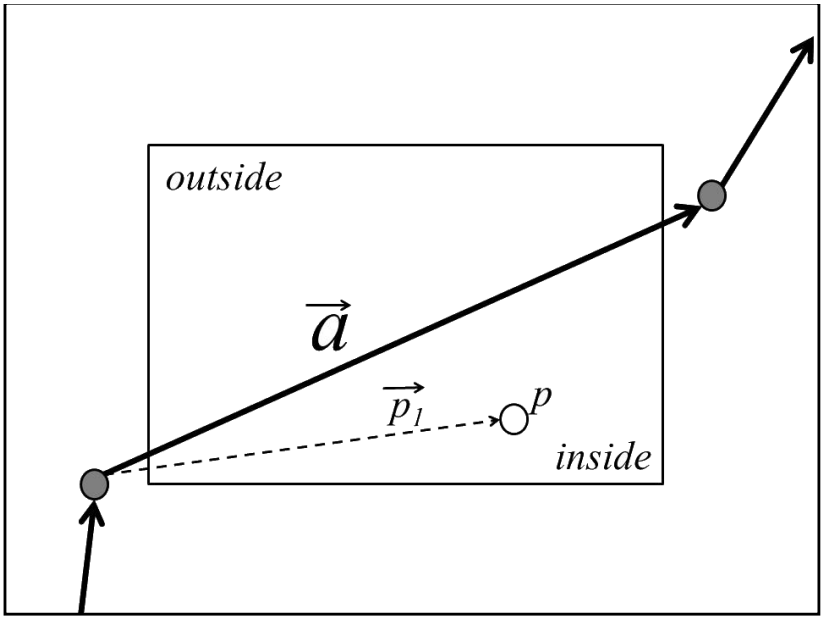

Figure 6: The simplest situation of edges crossing the terminal node's cell is seen here where only one cross product must be computed to determine the query point's position (shown as the open circle above).

In this example as with the two following, the solid arrowed lines represent the edges as well as the directional orientation of the polygon; in this case we assume the polygon as listed in the clockwise direction and tag the orientation with a vector $\mathbf{n}$ pointing into the plane. Knowing this, to determine the point's location relative to the polygon we take the cross product between $\mathbf{a}$, the edge of the polygon crossing the cell, and $\mathbf{p_1}$, the vector from the base of $\mathbf{a}$ to the query point p (which is defined by the open circle). The cross product of these two vectors will result in a vector pointing into the plane of the polygon, which is parallel with the polygon's orientation vector $\mathbf{n}$. This means that the query point would be determined to be inside of the polygon. If the cross product returned a vector pointing out of

the plane, anti-parallel with the orientation's vector, the point would then be determined as outside. These can be defined mathematically as:

Rules for Case 1:

- If **a** × **p1** is parallel to **n** ⇔ *p* is inside of the polygon.
- If **a** × **p1** is anti-parallel to **n** ⇔ *p* is outside of the polygon.

Case 2: 2 Edges Crossing

There can be two different situations which arise from two edges crossing the terminal node's cell. The first is shown in figure 7 below.

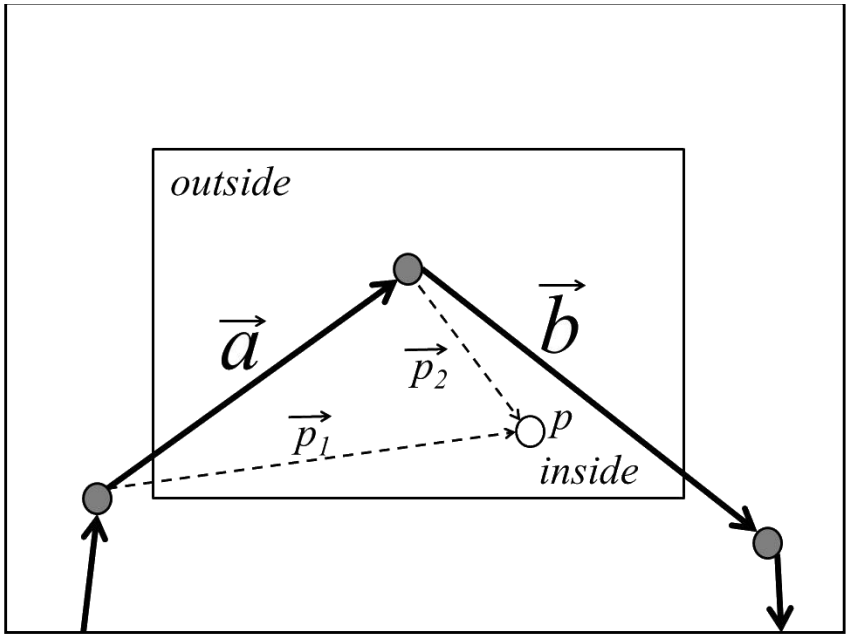

Figure 7: Here the situation with two edges crossing a cell can be seen. By implementing the use of cross products described, we can determine that the point *p* is indeed inside of the polygon.

In this situation we must employ a slightly different method than we did before. With the case of two edges crossing the terminal node's cell we need to first compute the cross product between the two edges crossing the cell. In Figure 7, these two edges are represented by the vectors **a** and **b**. The cross product **a** x **b** will yield a vector pointing into the plane, parallel to the polygon's orientation vector **n**. Knowing this we must now calculate two more cross products between vectors **a** and $\mathbf{p}_1$, and **b** and

$\mathbf{p}_2$. For the point *p* to be considered inside of the polygon both cross products $\mathbf{a} \times \mathbf{p}_1$ and $\mathbf{b} \times \mathbf{p}_2$ must be parallel with **n**, which in this case they are. If either of these two cross products were to yield a vector pointing out of the plane, anti-parallel to the orientation vector and the cross product of vectors a and b, the point would be outside of the polygon. Mathematically this transfers to:

Rules for case 2:

Assuming **a** x **b** is parallel to **n,**

- If $\mathbf{a} \times \mathbf{p}_1$ and $\mathbf{b} \times \mathbf{p}_2$ is parallel to **n** ⇔ *p* is inside of the polygon.
- If $\mathbf{a} \times \mathbf{p}_1$ or $\mathbf{b} \times \mathbf{p}_2$ is anti-parallel to **n** ⇔ *p* is outside of the polygon.

The next situation is very similar to previous, however the cross product between **a** and **b** evaluate to a vector pointing out of the polygon's plane. This can be seen below in figure 8.

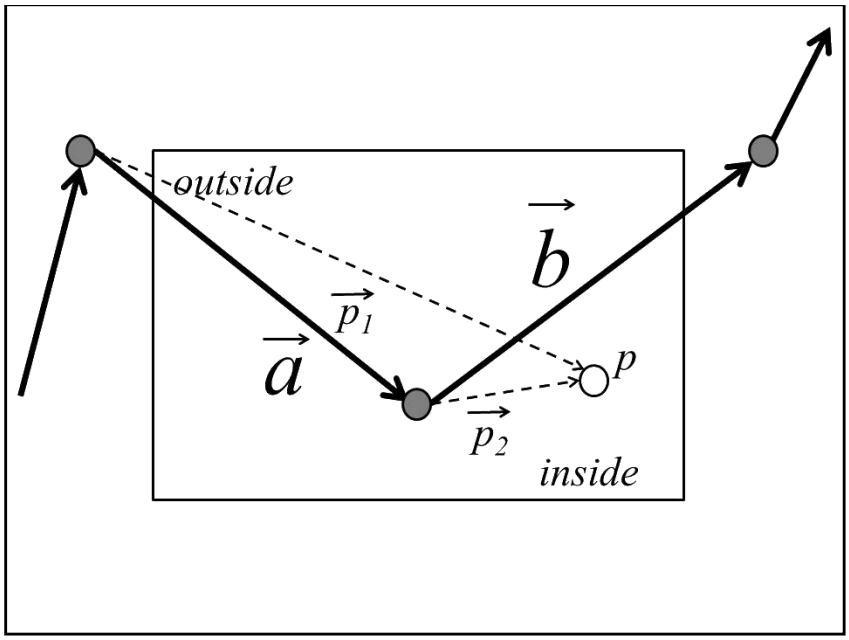

Figure 8: As in figure 7 we have two edges crossing the polygon, however the way the edges are oriented with respect to the bounding box is a little different here. However, we can still use our cross product procedure to determine that the point *p* is inside of the polygon.

With this case, there is a different, but similar, set of rules for determining the point's position. Since the cross product between **a** and **b** evaluate to a vector pointing out of the plane, the cross products between **a** and $\mathbf{p}_1$, and **b** and $\mathbf{p}_2$ must both point out of the plane for the point to be

considered outside of the polygon, otherwise it is inside. Again, we can formulate this mathematically as:

Rules for case 3:

Assuming $\mathbf{a} \times \mathbf{b}$ is anti-parallel to $\mathbf{n}$,

- If $\mathbf{a} \times \mathbf{p_1}$ or $\mathbf{b} \times \mathbf{p_2}$ is parallel to $\mathbf{n}$ $\Leftrightarrow$ *p* is inside the polygon.
- If $\mathbf{a} \times \mathbf{p_1}$ and $\mathbf{b} \times \mathbf{p_2}$ is anti-parallel to $\mathbf{n}$ $\Leftrightarrow$ *p* is outside the polygon.

d. Search Conclusions:

This simple technique proves to be very fast because of the binary tree structure used. No matter where the query point is located, the search for any specific point will be at worst on the order of O(log(m)) , assuming the use of a polygon which does not demonstrate any type of fine scale structure. In this case, the running time could vary widely depending on the formation of the polygon. For polygons defined by a high amount of edges, this method will usually end with a query point being found in a terminal node's cell that is already defined as inside or outside of the polygon, however in the case of this not occurring, a simple calculation must be done to determine where the point is in respect to the polygon as explained.

Section 3

Data

The following is data which has been collected from comparing our algorithm to Matlab's built in function "inpolygon". We believe Matlab uses a simple and commonly used technique for point-in-polygon detection, and is conducted as follows. Assuming the polygon is defined by *n* points in an array *P*, this algorithm computes the summation of angles between the query point and every pair of points defining each edge of the polygon (i.e. the angle between the query point and P[n] and P[n+1]). If this summation computes to 2π (or near 2π within some tolerance), then the point is inside the polygon. If the summation computes to zero (or near zero) then the point is outside of the polygon. Below is some data collected comparing our function "treeInPoly" to Matlab's "inpolygon".

i. Runtime vs. Number of Edges:

The next two figures show the runtime vs. number of edges defined by the polygon for a fixed 10,000 point query. The polygon used is a smooth circle where the amount of edges vary per test. At lower number of edges, it can be seen that our algorithm is actually slower than when the polygon contains a higher number of edges – this is most likely due to cross products needing to be calculated more frequently whereas at higher edge counts more of the terminal cells are already defined as inside or outside of the polygon, making for less computations in hardware.

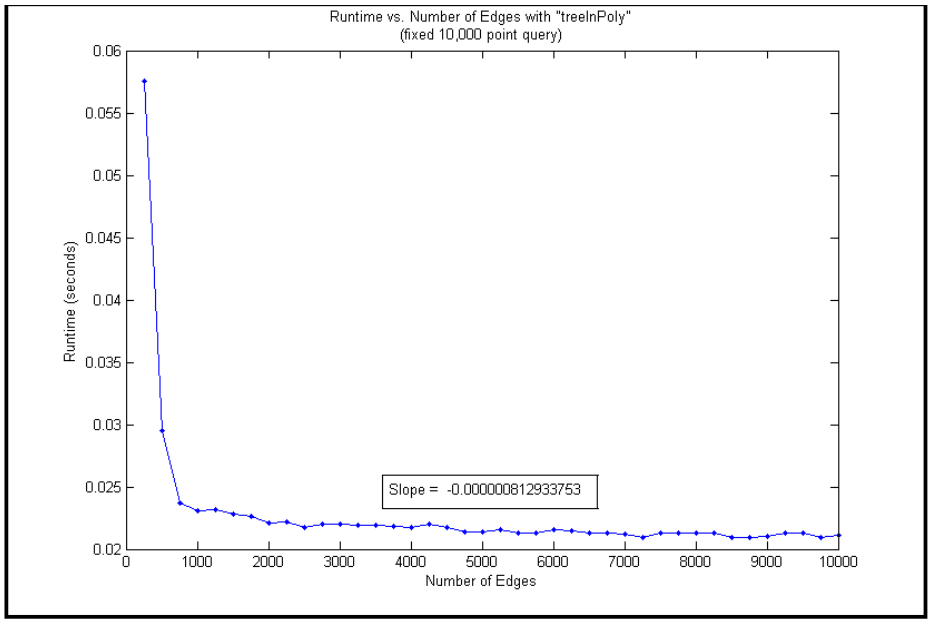

Figure 9: The quickness of our algorithm as the number of edges rises can be seen here.

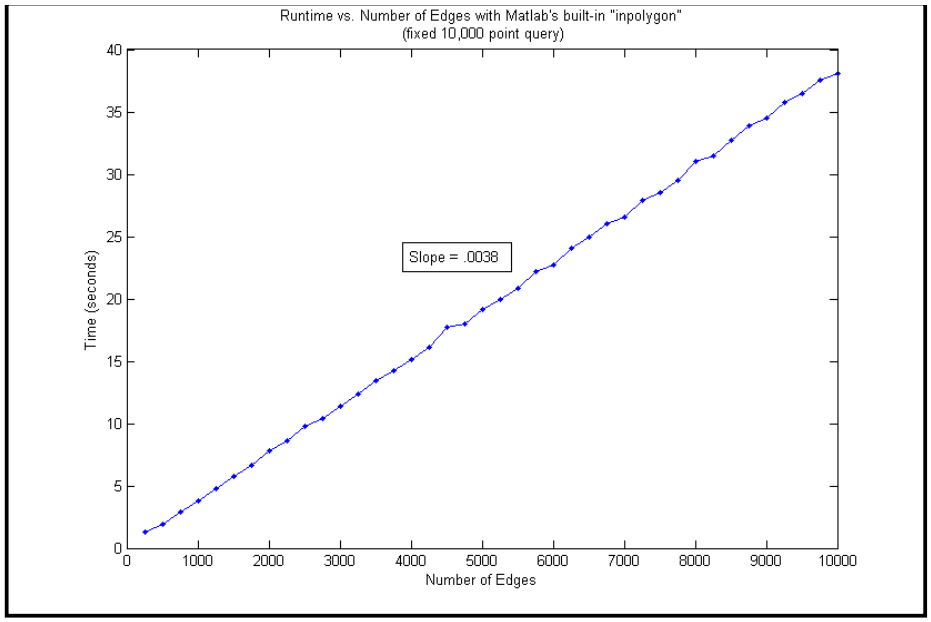

Figure 10: Matlab's "inpolygon" linear performance is very well demonstrated here.

ii. Runtime vs. Number of Query Points:

Both algorithms scale linearly when varying the number of query points and fixing the number of edges defining the polygon. Shown below are figures which reflect this linear scaling.

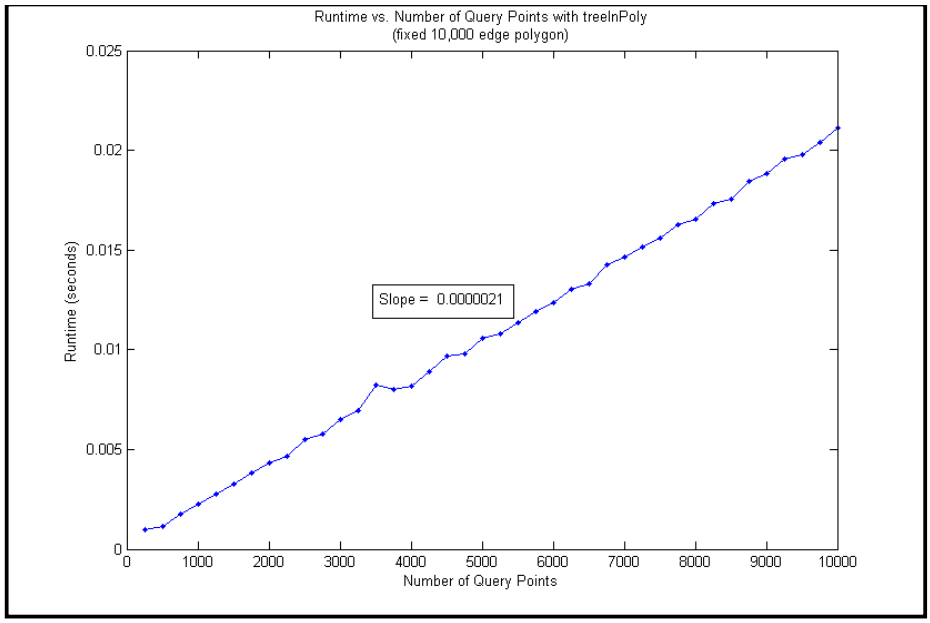

Figure 11: The run time vs. # of query points can be seen here using our algorithm and a fixed 10,000 edges. The algorithm should scale linearly in the number of query points, which it does as shown.

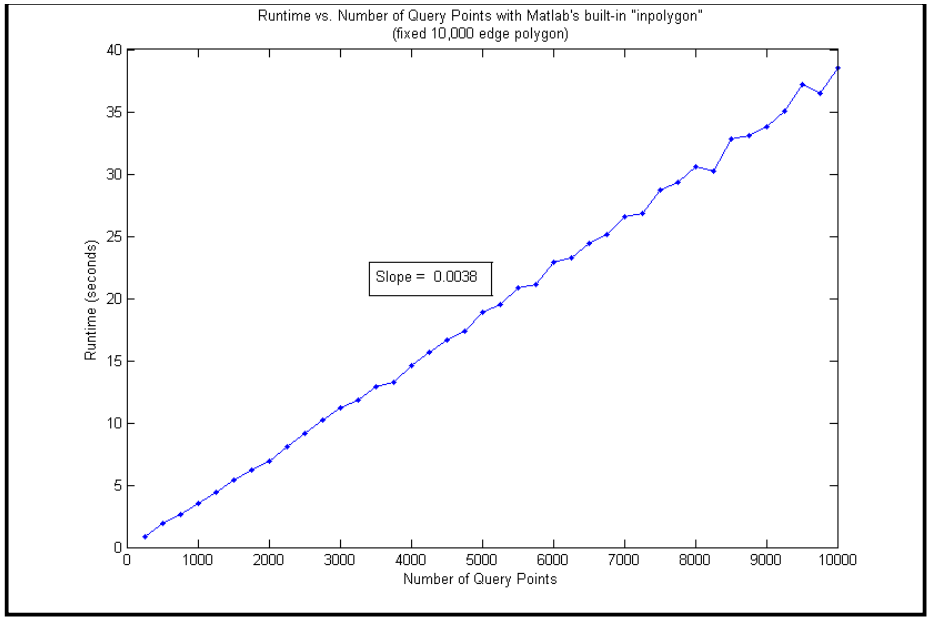

Figure 12: Matlab's "inpolygon" linear scaling can still be seen here.

iii. Pre-Processing Performance:

The performance of the pre-processing can vary widely. It's performance is highly dependent on the type of polygon being processed. Simpler polygons which do not exhibit fine scale structure will likely be processed much faster than those which do, such as the smooth circle used for the previous testing. When the polygon tends to be of a "nice" form, the runtime of the pre-processing tends to be linear as seen in figure 13. Figure 13 shows the runtime vs. the number of edges for the pre-processing of a smooth circle with 10,000 edges used for the previous testing.

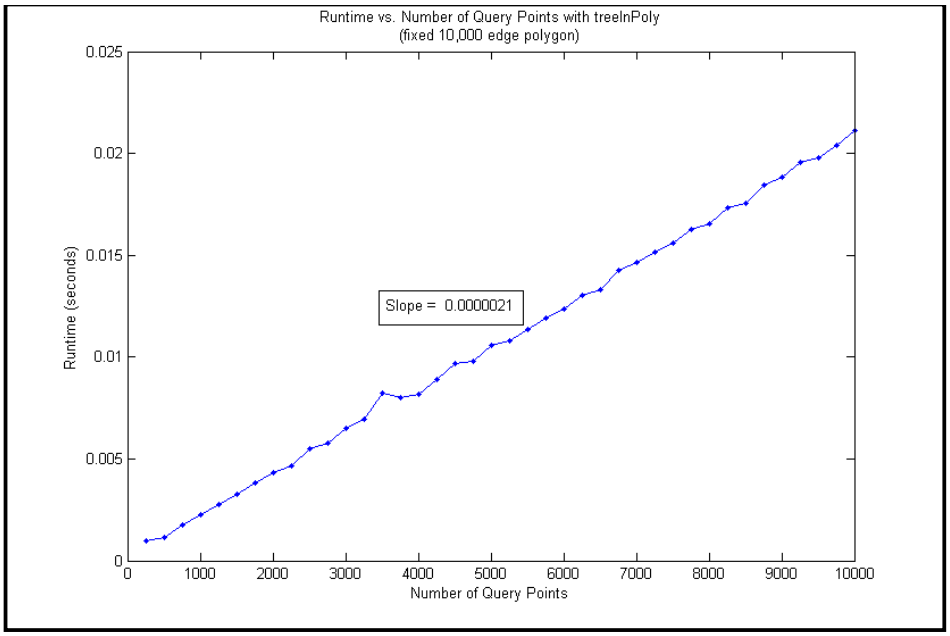

Figure 13: The runtime of our algorithm's pre-processing is shown as the number of edges in the polygon rises. The run time seems to be on the order of polynomial.

However, this linearity is only due to the simplicity of the polygon. For a more interesting look into the runtime of this algorithm, we consider a circle with jagged edges which exhibit some fine scale structure. Figure 3 and figure 4 display what this polygon looks like as a whole, as well as along the edges where it exhibits some jaggedness. Figure 14 shows the results of the runtime vs. the number of edges of the polygon when making this change.

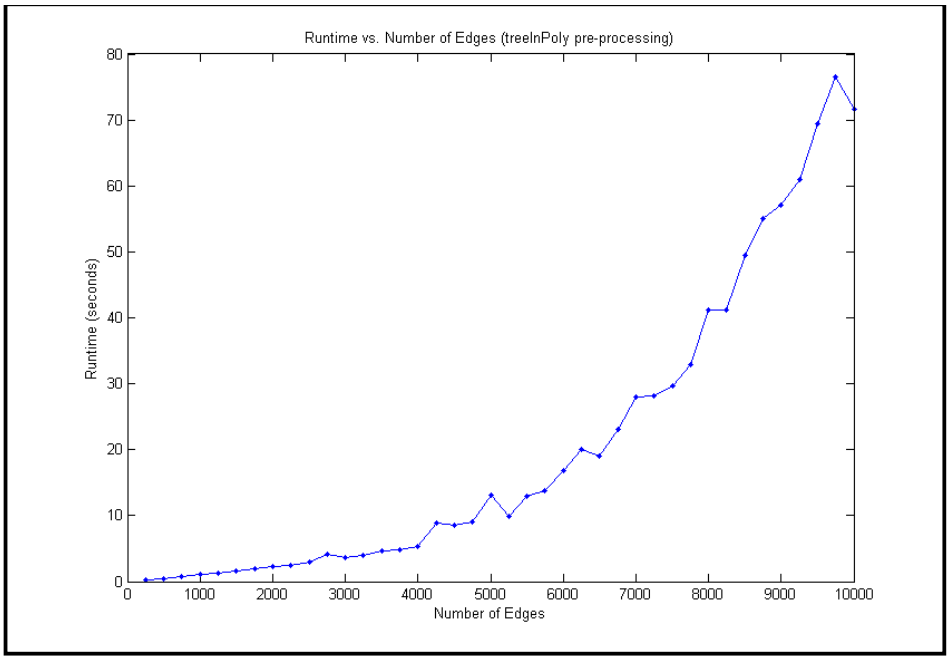

Figure 14: The exponential nature of the pre-processing can be seen here, as the running time grows exponentially with increasing the number of edges in the polygon

To determine what order the exponential is defining the pre-processing, we can do a log-log plot of this data. This can be seen in Figure 15 below.

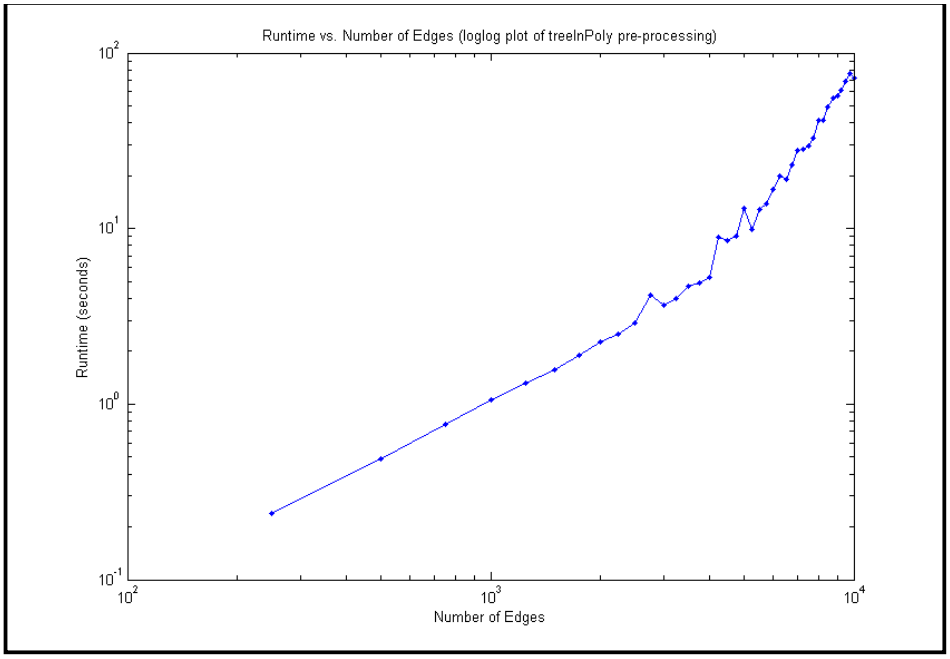

Figure 15: Here we want to zoom in on where the slope gets larger, which occurs at higher edges counts. This is where the fine scale structure will affect the pre-processing the most.

It can be seen that as the edge count grows, the slope of the line actually changes. This is because at lower edge counts the pre-processing algorithm is not affected as much by the fine scale structure, as it isn't too prevalent. However, as the edge count grows, so does the affect of the jaggedness on the pre-processing. Because of this, we extrapolate the slope from the sloped region to the right of the plot to determine the degree of the polynomial defining the pre-processing. We can then determine that the pre-processing runs along the order of $O(n^{2.6})$ for this.

Section 4

Conclusion

Our algorithm presents a new twist on already defined ways of computing point-in-polygon locations. As stated, it builds upon the gridding algorithm however optimizes it in several respects. Instead of placing a uniform sized grid on the given polygon, we instead perform binary cuts onto this polygon, creating a tree structure which can be queried for point locations relative to the polygon. This design may create a bit of overhead with respect to the pre-processing when considering more complicated polygons with fine scale structure. However, for more smooth polygons of simpler structure, this algorithm outperforms Matlab's built-in function "inpolygon" by a large amount. Again, this is due to the fact that the searching portion of our algorithm has a runtime on the order of O(m*log(n) where as "inpolyon" has a runtime on the order of O(n*m). The data produced has reinforced these notions, showing that "inpolygon" scaled linearly no matter which parameter, edges of the polygon or query points, is scaled with respect to runtime. Our algorithm, however, was shown to be linear in increasing the number of query points and for a nice structured polygon, and was shown to have a near zero slope when increasing the edge count while locking in the number of query points. Though for some cases it may not be the best to use our algorithm, as its pre-processing could become cumbersome when using a polygon exhibiting fine scale or complicated structure, there are many places where its use can highly speed up productivity. In the case counting the number of escape trajectories

of hydrogen from a given bounding polygon, as stated before, this algorithm works extremely well. Our algorithm, treeInPoly, has proven that it is far superior to Matlab's built-in function "inpolyon", and is very competitive, if not better to use depending on the situation than other point-in-polygon detection algorithms.

Bibliography

[1] Goodman, J. E. , and O'Rourke, J. , eds. *Handbook of Discrete and Computational Geometry,* Boca Raton, FL , April 2004 , Second Edition, Print.

[2] Haines, Eric, "Point in Polygon Strategies," *Graphics Gems .* Ed. Heckbert, Paul. Academic Press, 1994. Print.

[3] Schneider, Philip J. , and Eberly, David H. , *Geometric Tools for Computer Graphics*, Elsevier Science, USA, 2003, Print.

Source Code

```
function tree=getTree(data)
% gT   Takes a 2-dimensional convex polygon defined by an M x 2
%           array of data points and Builds/Returns a Matlab structure
%           resembling a tree to be used by the fastInPoly function for
%           point-in-polygon determination.
%
% Syntax:
%           tree=getTree(data)
%
% Input:
%           data =  An M x 2 matrix/array consisting of the data points
%                   that define the 2-dimensional convex polygon.
%                   NOTE: data can also also be a filename containing
%                   the polygonal data points.
% Output:
%           tree =  A matlab structure which basically represents a
%                   binary tree containing data which is used by the
%                   fastInPolygon function for point-in-polygon
%                   determination.
%
% History:
%   Jared Petker    5/17/2010 (updated)
%   E-mail: JPetker@ucmerced.edu

global index; global Tree;
index=0;
if ischar(data) %data is a filename string

    try
        data=csvread(data);
    catch E
        throw(E)
    end

elseif isnumeric(data) %data is an array

else
    error('input must be a filename string or numeric array');
end

 %figure;
 %hold on;
[tree]=init(data); %initialization
refineNode(tree,1,0);   %serves as the recursive function for creating the
tree
Tree(index+1:end,:)=[]; %trim back the array if it is too large
defTermPos(); %define empty terminal nodes as in/out
tree=Tree; %redefine the global Tree array for output

clear global; % clear those globals!!!
```

```
end

function refineNode(node,i,dir)
%% refineNode
global index; global base; global Tree;

index=index+1;
if (index~=1)
    if (dir) %1=left
        Tree(i,3)=index;
    else
        Tree(i,4)=index;
    end
    base(index).Lines=node.Lines;
end
Tree(index,15)=i;

base(index).bBox=node.bBox;

bBox=node.bBox;
[pointsIn linesC] = pointAdder(); %adds points to the node structure

%Creation of the left/right nodes if needed
if  pointsIn>1 || linesC==1
    Tree(index,6:13)=reshape(bBox',[1 8]);
    %if lengthX > lengthY, or lengthX=lengthY, cut in x
    if bBox(2,1) - bBox(1,1) > bBox(1,2) - bBox(4,2) || bBox(2,1) - bBox(1,1)
== bBox(1,2) - bBox(4,2)

        leftNode.bBox=[bBox(1,1) bBox(1,2); ...
            bBox(1,1)+(bBox(2,1)-bBox(1,1))/2 bBox(2,2); ...
            bBox(1,1)+(bBox(2,1)-bBox(1,1))/2 bBox(3,2); ...
            bBox(4,1) bBox(4,2)];


        rightNode.bBox=[bBox(1,1)+(bBox(2,1)-bBox(1,1))/2 bBox(1,2); ...
            bBox(2,1) bBox(1,2); ...
            bBox(3,1) bBox(3,2); ...
            bBox(1,1)+(bBox(2,1)-bBox(1,1))/2 bBox(3,2)];

        leftNode.points=base(index).points;
        rightNode.points=leftNode.points;
        leftNode.Lines=base(index).Lines;
        rightNode.Lines=leftNode.Lines;
        Tree(index,1:2)=[leftNode.bBox(2,1),1];
        ind=index;
        refineNode(leftNode,ind,1);
        refineNode(rightNode,ind,0);

        %if lengthY > lengthX, cut in y
    else

        leftNode.bBox=[node.bBox(1,1) node.bBox(1,2);node.bBox(2,1)
node.bBox(2,2);node.bBox(2,1) node.bBox(2,2)-(node.bBox(2,2)-
```

```
node.bBox(4,2))/2 ;node.bBox(1,1) node.bBox(2,2)-(node.bBox(2,2)-
node.bBox(3,2))/2];
        rightNode.bBox=[node.bBox(1,1) node.bBox(2,2)-(node.bBox(2,2)-
node.bBox(3,2))/2;node.bBox(2,1) node.bBox(2,2)-(node.bBox(2,2)-
node.bBox(3,2))/2;node.bBox(2,1) node.bBox(4,2);node.bBox(1,1)
node.bBox(4,2)];

        leftNode.points=base(index).points;
        rightNode.points=leftNode.points;
        leftNode.Lines=base(index).Lines;
        rightNode.Lines=leftNode.Lines;
        Tree(index,1:2)=[leftNode.bBox(3,2),0];
        ind=index;
        refineNode(leftNode,ind,0);     %   FUNCTION RECURSIVELY CALLED TO
CREATE INFORMATION FOR THE LEFT NODE
        refineNode(rightNode,ind,1);    %   FUNCTION RECURSIVELY CALLED TO
CREATE INFORMATION FOR THE RIGHT NODE

    end
   % Plot(node);
elseif pointsIn==1

    pointsOfInterestOnePoint(); %appneds POI points if one point is through
box

    Tree(index,1:2)=inf;
    Tree(index,14)=1;
     %Plot(node);
else
    %    inpolygon
    pointsOfInterestOneLine();
    Tree(index,1:2)=inf;
    Tree(index,14)=1;
   % Plot(node);
end

end

function node=init(polyPoints)
%% init
%Initializes and forms the full "data" structure as well as the root node
%for the tree
global base; global Tree; global I;

%pre-allocate the array for the tree, the Tree will usually consist of ~4
%times the amount of points that define the polygon, if this amount
%happens to be too much, the size is clipped afterwards.
Tree=zeros(size(polyPoints,1)*4,15);

s = struct('points',cell(1),'Lines',cell(1),'bBox',cell(1));
base = repmat(s,size(polyPoints,1)*4,1);

base(1).points=polyPoints; %points in each node of the tree

node.points=polyPoints;
xMin=min(polyPoints(:,1));
```

```
xMax=max(polyPoints(:,1));
yMin=min(polyPoints(:,2));
yMax=max(polyPoints(:,2));

node.bBox=[xMin yMax;xMax yMax;xMax yMin;xMin yMin];
base(1).bBox=node.bBox;
Tree(1,6:13)=reshape(node.bBox',[1 8]);

if polyArea(polyPoints)>0
    Tree(1,5)=-1;
else
    Tree(1,5)=1;
end

for i = 1:(size(base(1).points,1)-1)
    base(1).Lines(i,1:4)=[base(1).points(i,1) base(1).points(i,2)
base(1).points(i+1,1) base(1).points(i+1,2)];
end
base(1).Lines(end+1,1:4)=[base(1).points(end,1) base(1).points(end,2)
base(1).points(1,1) base(1).points(1,2)];

%%sorting
[X I]=sort(base(1).points(:,1));
Y=base(1).points(I,2);
base(1).sortedPoints=[X Y];


end

function [pointsIn linesC] = pointAdder()
%% pointAdder
% Loops through checking what points/lines are crossing the bounding cell
% and adding them to their respective arrays
global base; global index;

linesC=0;
pointsIn=0;

ind=1;
nLines=base(index).Lines;
bBox=base(index).bBox;
tempPoints=zeros(size(nLines,1),2);
Lines=zeros(size(nLines,1),size(nLines,2));
% Lines=zeros(size(nLines,1),size(nLines,);
for rowNum=1:size(nLines,1)
    firstPoint=nLines(rowNum,1:2);
    lastPoint=nLines(rowNum,3:4);

    %% FIRST POINT
    if firstPoint(1) > bBox(1,1) && firstPoint(1) < bBox(3,1) &&
firstPoint(2) >bBox(3,2)  && firstPoint(2) < bBox(1,2)
        firstPointIN=1;
        pointsIn=pointsIn+1;
        tempPoints(pointsIn,1)=nLines(rowNum,1); %#ok<AGROW>
        tempPoints(pointsIn,2)=nLines(rowNum,2); %#ok<AGROW>
        tempPoints(pointsIn,3)=0; %#ok<AGROW>
```

```
    elseif firstPoint(1) < bBox(1,1) || firstPoint(1) > bBox(3,1) ||
firstPoint(2) < bBox(3,2)  || firstPoint(2) > bBox(1,2)
        firstPointIN=0;
    else
        pointsIn=pointsIn+1;
        tempPoints(pointsIn,1)=nLines(rowNum,1); %#ok<AGROW>
        tempPoints(pointsIn,2)=nLines(rowNum,2); %#ok<AGROW>
        tempPoints(pointsIn,3)=1; %#ok<AGROW>
        firstPointIN=1;
    end

    %% SECOND POINT

    if lastPoint(1) >= bBox(1,1) && lastPoint(1) <= bBox(3,1) && lastPoint(2)
>= bBox(3,2)  && lastPoint(2) <= bBox(1,2)
        lastPointIN=1;
    elseif lastPoint(1) < bBox(1,1) || lastPoint(1) > bBox(3,1) ||
lastPoint(2) < bBox(3,2)  || lastPoint(2)  > bBox(1,2)
        lastPointIN=0;
    end

    %%LINES

    if firstPointIN~=0 || lastPointIN~=0
        Lines(ind,:)=nLines(rowNum,:); %#ok<AGROW>
        ind=ind+1;
    else
        t=0;
                                 if lastPoint(1)<firstPoint(1)
                                    if lastPoint(1)>bBox(2,1) ||
firstPoint(1)<bBox(1,1)
                                        t=1;
                                    end

                                 elseif firstPoint(1)>bBox(2,1) ||
lastPoint(1)<bBox(1,1)

                                        t=1;

                                 end

                                 if lastPoint(2)<firstPoint(2) && t==0
                                    if lastPoint(2)>bBox(2,2) ||
firstPoint(2)<bBox(3,2)
                                        t=1;
                                    end

                                 elseif firstPoint(2)>bBox(2,2) ||
lastPoint(2)<bBox(3,2)

                                        t=1;

                               end
        if t==0 && ifLineCrosses(firstPoint,lastPoint,bBox)==1
        Lines(ind,:)=nLines(rowNum,:); %#ok<AGROW>
```

```
        ind=ind+1;
        end
    end

end

tempPoints(pointsIn+1:end,:)=[];


Lines(ind:end,:)=[];

base(index).points=tempPoints;
base(index).Lines=Lines;


switch pointsIn
    case 0
        if size(base(index).Lines,1)>1
            linesC=1;
        end

    case 1
        if size(base(index).Lines,1)>2
            linesC=1;
        end
end



end

function success = pointsOfInterestOneLine()
%% pointsOfInterestOneLine
%
global index; global base; global Tree;
success=0;
if size(base(index).Lines,1)
    Tree(index,6:7)=base(index).Lines(1:2);
    Tree(index,8:9)=base(index).Lines(3:4);
    Tree(index,10:11)=inf;
    Tree(index,5)=-1;
    success=~success;
end

end

function pointsOfInterestOnePoint()
%% pointsOfInterestOnePoint
global base; global index; global Tree;

Tree(index,5)= -1;
points=base(1).points;
ind=find(points(:,1)==base(index).points(1) & points(:,2) ==
base(index).points(2));

if ind~=1 && ind ~=size(points,1)
```

```
    Tree(index,6:11)=[points(ind-1,1:2) points(ind,1:2) points(ind+1,1:2)];
elseif ind==1
    Tree(index,6:11)=[points(end,1:2) points(ind,1:2) points(ind+1,1:2)];
else
    Tree(index,6:11)=[points(ind-1,1:2) points(ind,1:2) points(1,1:2)];
end



end

function Area = polyArea(Data)
%% Area
%Area = polyArea(Data) => returns the area of the polygon while iterating
%through the points.  Positive Area means the polygon's direction is CCW,
%negative means CW

Area=0;
Data(end+1,:)=Data(1,:);

for rowNum=1:size(Data,1)-1

    x1=Data(rowNum,:);
    x2=Data(rowNum+1,:);

    Area = Area + (x2(1) - x1(1)) * (x2(2) + x1(2)) / 2;
end



end

function Plot(node)
%% plot
x=node.bBox(:,1);
y=node.bBox(:,2);
x(end+1)=node.bBox(1,1);
y(end+1)=node.bBox(1,2);
plot(x,y);

end

function num = ifLineCrosses(thisP,lastP,bBox)
%% ifLineCrosses

num=0;
for bPos=1:3

    cX=bBox(bPos,1);
    cY=bBox(bPos,2);
    dX=bBox(bPos+1,1);
    dY=bBox(bPos+1,2);
    denominator = (thisP(1) - lastP(1)) * (dY - cY) - (thisP(2) - lastP(2)) *
(dX - cX);

    numeratorA = (lastP(2) - cY) * (dX - cX) - (lastP(1) - cX) * (dY - cY);
    A = numeratorA / denominator;
```

```
    numeratorB = (lastP(2) - cY) * (thisP(1) - lastP(1)) - (lastP(1) - cX) *
(thisP(2) - lastP(2));
    B = numeratorB / denominator;

    if (denominator == 0)
        %lines llastP(2) on top of each other, do nothing
    elseif (A < 0 || A > 1 || B < 0 || B > 1)
        %lines don't cross, do nothing
    else

        num=1;

        break;

    end
end

end

function defTermPos()
global Tree; global base;

mask=find(Tree(:,5)~=-1 & Tree(:,3)==0);
if ~isempty(mask)
%mask(1)=[];
end

for i=1:size(mask)

    maskedNode=Tree(mask(i),:);

    if  Tree(maskedNode(15),3)==mask(i) %Tree(i) is a left Node, go to the
parents right node

        parent=Tree(maskedNode(15),:);
        curNode=Tree(parent(4),:); %parents left node
        if parent(2) %use cut and the min/mlastP(1)y
            anchors=[parent(1) parent(9);parent(1)  parent(11)]; %anchors,
must make sure one of these two points is at each traversed node
        else
            anchors=[parent(6) parent(1);parent(8) parent(1)];
        end

        %look at curNode's left and right to see which one contains 1 (or
        %more) of the anchors

        while ~curNode(14)

            if ~isempty(find(base(curNode(3)).bBox(:,1)==anchors(1,1) &
base(curNode(3)).bBox(:,2)==anchors(1,2),1))
                curNode=Tree(curNode(3),:); %left
                curAnchor=(anchors(1,:));
            elseif isempty(find(base(curNode(4)).bBox(:,1)==anchors(1,1) &
base(curNode(4)).bBox(:,2)==anchors(1,2),1))
```

```
                curNode=Tree(curNode(4),:); %left
                curAnchor=(anchors(1,:));
            elseif isempty(find(base(curNode(3)).bBox(:,1)==anchors(2,1) &
base(curNode(3)).bBox(:,2)==anchors(2,2),1))
                curNode=Tree(curNode(3),:); %right
                curAnchor=(anchors(2,:));
            elseif isempty(find(base(curNode(4)).bBox(:,1)==anchors(2,1) &
base(curNode(4)).bBox(:,2)==anchors(2,2),1))
                curNode=Tree(curNode(4),:); %right
                curAnchor=(anchors(2,:));
            end
            if isinf(curNode(1)) && curNode(5)~=-1

                p=Tree(curNode(15),:);
                if Tree(p(3),:)==curNode
                    temp=Tree(p(4),:);
                else
                    temp=Tree(p(3),:);
                end
                tempAnchor=anchors;
                if p(2) %use cut and the min/mlastP(1)y
                    anchors=[p(1) temp(9);p(1) temp(11)]; %anchors, must make
sure one of these two points is at each traversed node
                else
                    anchors=[temp(6) p(1);temp(8) p(1)];
                end
                if anchors(1,:)~=curAnchor
                    if anchors(2,:)~=curAnchor
                        anchors=tempAnchor;
                    end
                end
                curNode=temp;
            end
        end

        %compute if curNode is in or out
        Tree(mask(i),5) = isNodeIn(curNode,curAnchor(1,:));

    else %Tree(i) is a right Node, go to the parents left node
        parent=Tree(maskedNode(15),:);
        curNode=Tree(parent(3),:); %parents left node
        if parent(2) %use cut and the min/mlastP(1)y
            anchors=[parent(1) parent(9);parent(1)  parent(11)]; %anchors,
must make sure one of these two points is at each traversed node
        else
            anchors=[parent(6) parent(1);parent(8) parent(1)];
        end

        %look at curNode's left and right to see which one contains 1 (or
        %more) of the anchors
        while ~curNode(14)
             if ~isempty(find(base(curNode(3)).bBox(:,1)==anchors(1,1) &
base(curNode(3)).bBox(:,2)==anchors(1,2),1))
                curNode=Tree(curNode(3),:); %left
                curAnchor=(anchors(1,:));
            elseif isempty(find(base(curNode(4)).bBox(:,1)==anchors(1,1) &
base(curNode(4)).bBox(:,2)==anchors(1,2),1))
```

```
                curNode=Tree(curNode(4),:); %left
                curAnchor=(anchors(1,:));
            elseif isempty(find(base(curNode(3)).bBox(:,1)==anchors(2,1) &
base(curNode(3)).bBox(:,2)==anchors(2,2),1))
                curNode=Tree(curNode(3),:); %right
                curAnchor=(anchors(2,:));
            elseif isempty(find(base(curNode(4)).bBox(:,1)==anchors(2,1) &
base(curNode(4)).bBox(:,2)==anchors(2,2),1))
                curNode=Tree(curNode(4),:); %right
                curAnchor=(anchors(2,:));
            end
            if isinf(curNode(1)) && curNode(5)~=-1


                p=Tree(curNode(15),:);
                if Tree(p(3),:)==curNode
                    temp=Tree(p(4),:);
                else
                    temp=Tree(p(3),:);
                end

                tempAnchor=anchors;
                if p(2) %use cut and the min/mlastP(1)y
                    anchors=[p(1) temp(9);p(1) temp(11)]; %anchors, must make
sure one of these two points is at each traversed node
                else
                    anchors=[temp(6) p(1);temp(8) p(1)];
                end

                if anchors(1,:)~=curAnchor
                    if anchors(2,:)~=curAnchor
                        anchors=tempAnchor;
                    end
                end
               curNode=temp;
            end
        end
       %compute if curNode is in or out
        Tree(mask(i),5) = isNodeIn(curNode,curAnchor(1,:));
    end
end
end
function IN=isNodeIn(node,point)
global Tree
IN=logical(false);
in=Tree(1,5);
out=-in;
if isinf(node(10))
    pointsOfInterest=reshape(node(6:9),[2 2])';
    vector1=[pointsOfInterest(2,1)-pointsOfInterest(1,1)
pointsOfInterest(2,2)-pointsOfInterest(1,2) 0];
    vector2=[point(1)-pointsOfInterest(1,1) point(2)-pointsOfInterest(1,2)
0];
    signCross=sign([vector1(2).*vector2(3)-vector1(3).*vector2(2)...
        vector1(3).*vector2(1)-vector1(1).*vector2(3)...
        vector1(1).*vector2(2)-vector1(2).*vector2(1)]);
    if signCross(3)~=out
```

```
        IN=~IN;
    end
else
    pointsOfInterest=reshape(node(6:11),[2 3])';
    vector1=[pointsOfInterest(2,1)-pointsOfInterest(1,1)
pointsOfInterest(2,2)-pointsOfInterest(1,2) 0];
    vector2=[pointsOfInterest(3,1)-pointsOfInterest(2,1)
pointsOfInterest(3,2)-pointsOfInterest(2,2) 0];
    vector3=[point(1,1)-pointsOfInterest(1,1) point(1,2)-
pointsOfInterest(1,2) 0];
    vector4=[point(1,1)-pointsOfInterest(2,1) point(1,2)-
pointsOfInterest(2,2) 0];

    signCross1= [vector1(2).*vector2(3)-vector1(3).*vector2(2)...
        vector1(3).*vector2(1)-vector1(1).*vector2(3)...
        vector1(1).*vector2(2)-vector1(2).*vector2(1)];
    signCross2=[vector1(2).*vector3(3)-vector1(3).*vector3(2)...
        vector1(3).*vector3(1)-vector1(1).*vector3(3)...
        vector1(1).*vector3(2)-vector1(2).*vector3(1)];
    signCross3=[vector2(2).*vector4(3)-vector2(3).*vector4(2)...
        vector2(3).*vector4(1)-vector2(1).*vector4(3)...
        vector2(1).*vector4(2)-vector2(2).*vector4(1)];
    s=sign([signCross1(3) signCross2(3) signCross3(3)]);

    if s(1)==in

        if s(2)==in && s(3)==in
            IN=~IN;
        elseif s(2)==0 || s(3)==0
            IN=~IN;

        end

    elseif s(1)==out
        if s(2)==out && s(3)==out
            %intN=0;
        elseif s(2)==0 || s(3)==0
            IN=~IN;
        else
            IN=~IN;
        end
    else
        if s(2)==out
            %intN=0;
        elseif s(2)==0 || s(3)==0
            IN=~IN;
        else
            IN=~IN;
        end
    end
end
end
```

```
function IN = in_fastInPoly(points,node)
% in_fastInPoly   Takes a 2-dimensional convex polygon defined by an M x 2
%           array of data points and Builds/Returns a Matlab structure
%           resembling a tree to be used by the fastInPoly function for
%           point-in-polygon determination.
%
% Syntax:
%           IN = in_fastInPoly(points,node)
%
% Input:
%           points =  The query points.
%
%           node = The tree returned by the getTree method.
% Output:
%           IN = A vector representing if each respective query point is
%           inside or outside of the polygon.
%
% History:
%   Jared Petker    5/17/2010 (updated)
%   E-mail: JPetker@ucmerced.edu

IN=false(size(points,1),1); %#ok
tempSize=size(points,1);

in=node(1,5);
out=-in;
bBox=reshape(node(1,6:13),[2 4])';

for i=1:tempSize;

    point=points(i,:);
    n=1;

    if point(1) > bBox(1,1) && point(1) < bBox(3,1) && point(2) >bBox(3,2)
&& point(2) < bBox(1,2)
    % main loop for basically doing a binary search through the bounding
boxes.
        while ~node(n,14)

            n=node(n,(point(2-node(n,2))>=node(n,1))+3);

        end



        if node(n,5)>=0
            IN(i)=node(n,5);

        else


            if isinf(node(n,10))
                pointsOfInterest=reshape(node(n,6:9),[2 2])';
                vector1=[pointsOfInterest(2,1)-pointsOfInterest(1,1)
pointsOfInterest(2,2)-pointsOfInterest(1,2) 0];
```

```
                vector2=[point(1)-pointsOfInterest(1,1) point(2)-
pointsOfInterest(1,2) 0];
                signCross=sign([vector1(2).*vector2(3)-
vector1(3).*vector2(2)...
                    vector1(3).*vector2(1)-vector1(1).*vector2(3)...
                    vector1(1).*vector2(2)-vector1(2).*vector2(1)]);
                if signCross(3)~=out
                    IN(i)=~IN(i);
                end


            else
                pointsOfInterest=reshape(node(n,6:11),[2 3])';
                vector1=[pointsOfInterest(2,1)-pointsOfInterest(1,1)
pointsOfInterest(2,2)-pointsOfInterest(1,2) 0];
                vector2=[pointsOfInterest(3,1)-pointsOfInterest(2,1)
pointsOfInterest(3,2)-pointsOfInterest(2,2) 0];
                vector3=[point(1,1)-pointsOfInterest(1,1) point(1,2)-
pointsOfInterest(1,2) 0];
                vector4=[point(1,1)-pointsOfInterest(2,1) point(1,2)-
pointsOfInterest(2,2) 0];


                signCross1= [vector1(2).*vector2(3)-vector1(3).*vector2(2)...
                    vector1(3).*vector2(1)-vector1(1).*vector2(3)...
                    vector1(1).*vector2(2)-vector1(2).*vector2(1)];
                signCross2=[vector1(2).*vector3(3)-vector1(3).*vector3(2)...
                    vector1(3).*vector3(1)-vector1(1).*vector3(3)...
                    vector1(1).*vector3(2)-vector1(2).*vector3(1)];
                signCross3=[vector2(2).*vector4(3)-vector2(3).*vector4(2)...
                    vector2(3).*vector4(1)-vector2(1).*vector4(3)...
                    vector2(1).*vector4(2)-vector2(2).*vector4(1)];
                s=sign([signCross1(3) signCross2(3) signCross3(3)]);

                if s(1)==in

                    if s(2)==in && s(3)==in
                        IN(i)=~IN(i);
                    elseif s(2)==0 || s(3)==0
                        IN(i)=~IN(i);

                    end

                elseif s(1)==out
                    if s(2)==out && s(3)==out
                        %intN=0;
                    elseif s(2)==0 || s(3)==0
                        IN(i)=~IN(i);
                    else
                        IN(i)=~IN(i);
                    end
                else
                    if s(2)==out
                        %intN=0;
                    elseif s(2)==0 || s(3)==0
                        IN(i)=~IN(i);
```

```
                    else
                        IN(i)=~IN(i);
                    end
                end


            end


        end


    end

end
end
```