Final Project

Marching Squares algorithm:

This algorithm can be briefly described in 4 steps:

1. Make a grid and initialize each grid sub square as a cell
2. Traverse the cells, compute the function at the corner points and make a binary index
3. Compare the binary index to the standard 16 cases
4. Based on the case, compute the segment using interpolation

Max Length of each cell side:
The max length of each cell side would less than the max segment length because the diagonal of the cell has to be the maximum. Therefore the max cell side length has been chosen as h*sqrt(2)/4.

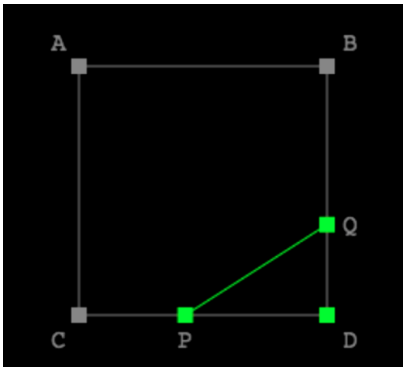
Making the grid and cells:

In order to make the grid, the algorithm has been inspired from the way numpy.meshgrid works. First compute the Xs and Ys using numpy.linspace(). After this, compute the centers of each cell using these Xs and Ys. Next, we take the centers of the squares and compute the corners by incrementing and decrementing the center's x and y coordinates by maximum cell side length.

The cells are defined by cell data structure. This data structure stores the corners of the sub-squares, center of the cells and the binary index of the cell. The binary index is computed immediately when a cell is computed.

Computing the boundary:

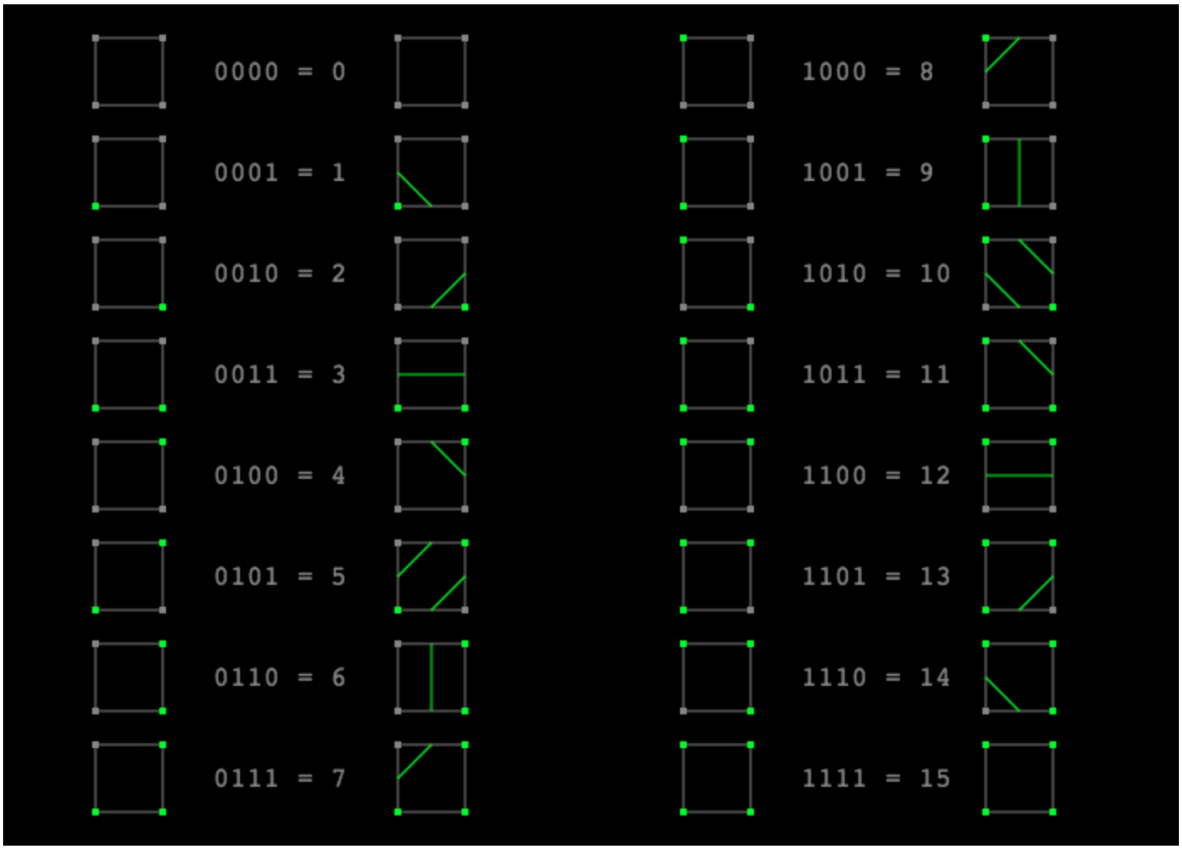
After all the cells are made, we iterate over the cells. For each cell we compare its binary index against the 16 standard binary cases shown below. Next, we need to use linear interpolation to compute the actual polygon segment. One example will be discussed in this discussion. Let's take the following example:



We know that, Qx = Bx and Py = Cy. Even though the segment isn't completely linear, because of our small cell size, liner interpolation will give us a good approximation. This allows us to compute the Qy with the following formula:

$$Q_y = B_y + (D_y - B_y)\left(\frac{1 - f(B_x, B_y)}{f(D_x, D_y) - f(B_x, B_y)}\right)$$

Px can be computed in a very similar fashion using the above formula. Dy will be replaced with Cx and all Y's will be replaced by X terms.



The configuration number between 0-15 is computed by assigning a value of 0 to each of the corners where $f(x, y) < 1$, and a value of 1 where $f(x, y) \geq 1$, then interpreting these bits as a binary number, ordered (southwest, southeast, northeast, northwest).

Note: All the computations at every step of the algorithm is rounded to 3 digits for high accuracy.

Ordering the segments:

In order to order the segments, the following steps are used:

1. Pick a segment from the start or the end of the segments array.
2. Choose one of its point as the last point and the other point as reference point.
3. Add the last point to the ordered points array
4. Because every line segment has to be connected to another line segment, they will share common point.
5. Search for the line segment that contains our reference point.
6. Now put the reference point in the ordered points and make the other point In the newly found segment as reference and search again
7. Repeat this process until the reference point because the point that we added first to the ordered points array.
8. This way, the algorithm returns a counter clockwise oriented list of points.

Complexity of Marching Squares:

The actual algorithm where the segments are computed is just O(N) because we just iterate over each cell. But, in order to order the computed segments and points, the algorithm designed has to look through all the segments for each reference point. This means that the worst-case scenario of the comparison is O(N^2). This makes the entire algorithm O(N^2) unfortunately. This can be avoided by iterating over cells in a different way.

Polygon Evaluation:

In order to evaluate the polygon as a simple polygon, two rules must be satisfied:

1. Number of line segments = Number of Points
2. Number of line intersections = 0

To check the first rule, we just compare the lengths of the segments array and the ordered points array. To check the next rule, the following steps are used:
Let, x be x coordinate y be y coordinate and P-N be a point
x1, y1 = P1[0], P1[1]
x2, y2 = P2[0], P2[1]
x3, y3 = P3[0], P3[1]
x4, y4 = P4[0], P4[1]

1. If the domain of X's don't overlap, then there can be no intersection. That is, if max(x1, x2) < min(x3, x4) or max(x3, x4) < min(x1, x2)
2. Same case with domain of Y's. That's is, max(y1, y2) < min(y3, y4) or max(y3, y4) < min(y1, y2).
3. Now if they do overlap, we check if the lines are parallel to each other or the same line! If any is true there is no overlap.
4. Finally, if all the above tests don't work, we set both the lines equal to each other and compute for x value of intersection. If the value of x computed is outside the domain of X's then there is no overlap

Polygon Triangulation:

The following steps define the algorithm:

1. The first step is to store the polygon as a doubly linked list so that you can quickly remove ear tips. Construction of this list is an O(n) process.

2. The second step is to iterate over the vertices and find the ears. For each vertex V and corresponding triangle vertices, test all other vertices to see if any are inside the triangle. If exactly none are inside, it is an ear. This test is done by considering only reflex angles (theta > 180).

3. The vertices of the polygon are stored in a cyclical list, the convex vertices are stored in a linear list, the reflex vertices are stored in a linear list, and the ear tips are stored in a cyclical list.

4. Once the initial lists for reflex vertices and ears are constructed, the ears are removed one at a time.

5. If Vi is an ear that is removed, then the edge configuration at the adjacent vertices Vi−1 and Vi+1 can change. If an adjacent vertex is convex, it remains convex.

6. If an adjacent vertex is an ear, it does not necessarily remain an ear after Vi is removed. If the adjacent vertex is reflex, it is possible that it becomes convex and, possibly, an ear. Thus, after the removal of Vi , if an adjacent vertex is convex you must test if it is an ear by iterating over the reflex vertices and testing for containment in the triangle of that vertex.

7. There are O(n) ears. Each update of an adjacent vertex involves a test for ear, a process that is O(n) per update. Thus, the total removal process is O(N^2).

Point membership:

Ray-Casting algorithm for point membership:

The algorithm is based on the observation that if a point moves along a ray from infinity to the probe point and if it crosses the boundary of a polygon, possibly several times, then it alternately goes from the outside to inside, then from the inside to the outside, etc. As a result, after every two "border crossings" the moving point goes outside. The algorithm is so simple that the actual code and pseudo code are very much the same!

Pseudo code:

```
Let a, b be edges and p be a point
IF a.y > b.y,
   THEN a, b = b, a
IF (p.y > b.y OR p.y < a.y) OR (p.x > MAX OF(a.x, b.x)):
   Then No intersection
IF p.x < min(a.x, b.x):
   There is intersection
ELSE,
   IF abs(a.x - b.x) is considerable,
      m_red = (b.y - a.y) / float(b.x - a.x)
   ELSE,
      m_red = _huge
   IF abs(a.x - p.x) is considerable,
      m_blue = (p.y - a.y) / float(p.x - a.x)
   ELSE
      m_blue = _huge
   intersect is true and  = m_blue >= m_red
```

We compute these intersections for all the edges. If the number of intersection is even, then the point is outside or else inside. This makes the algorithm O(N).

Shortest distance between the point and the polygon:

The algorithm for this is short and simple:

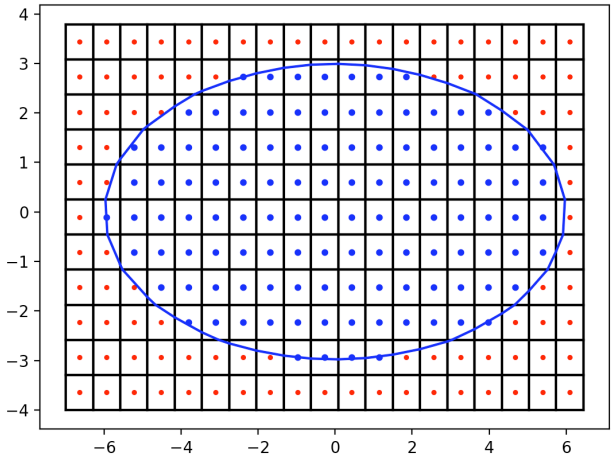
1. Check if the point projects onto the line segment.
2. If yes, we calculate the perpendicular distance from the point to the line using the cross product and norm.

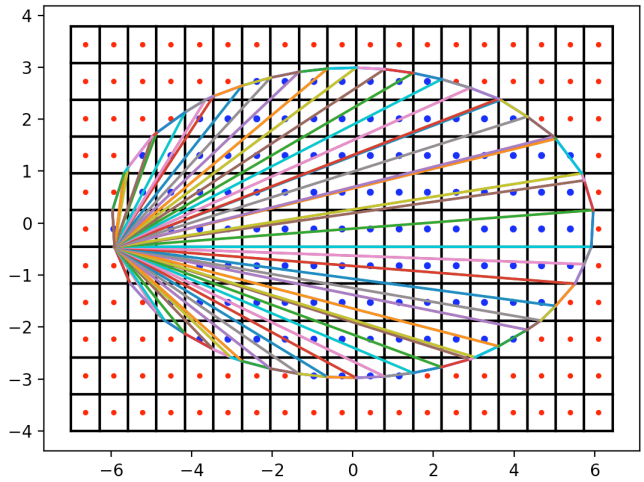
```
Unit Line = Line[1] – Line[0]                                    //line = [x,y]
Distance = norm(cross(line[1] - line[0], line[0] - point)) / norm(unit_line)
```

3. If no, we calculate the distance to both endpoints and take the shortest distance!

This distance will be calculated for all the edge sof the polygon. Making this a O(N) algorithm.

Testing:
The following code has been tested using functions and bounds obtained from wolfram alpha. One such case is shown here (Example from description).





The constructed boundary is a simple polygon.
(0.00.0) : in = True, distance =2.9764321421771345
(100.00.0) : in = False, distance =94.0303270439915