

Creating Triangle Meshes for the Finite Element Method

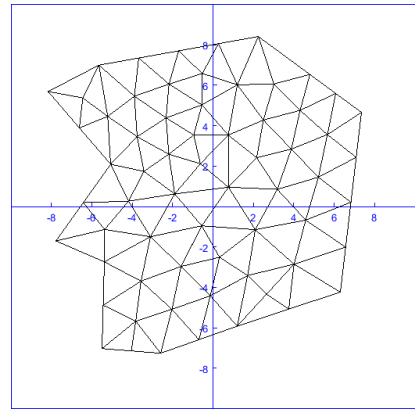
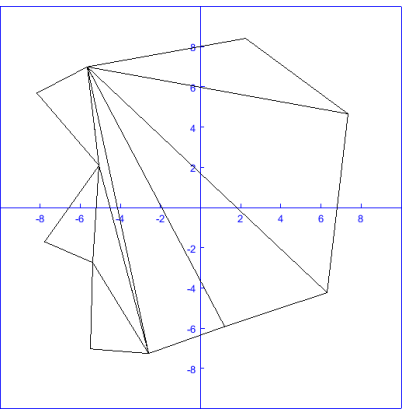
Shea Yonker

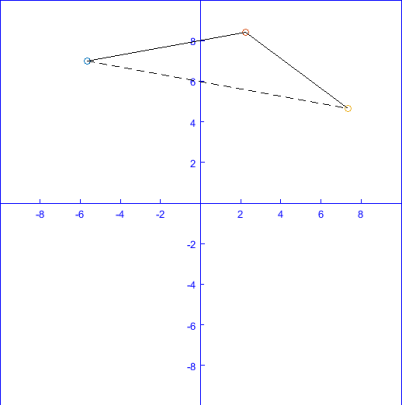
Host: Chris Deotte

Thursday, May 18th, 2017

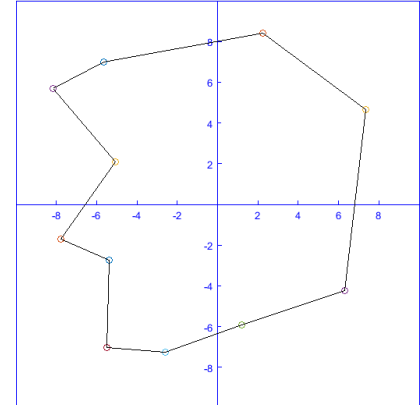
11:00 AM

AP&M 2402

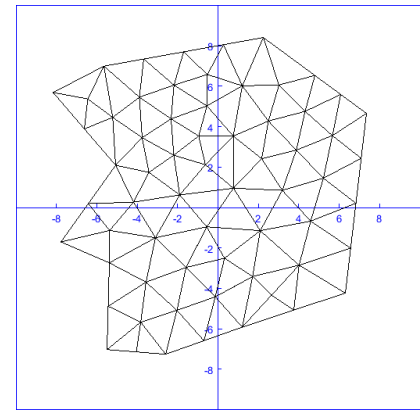
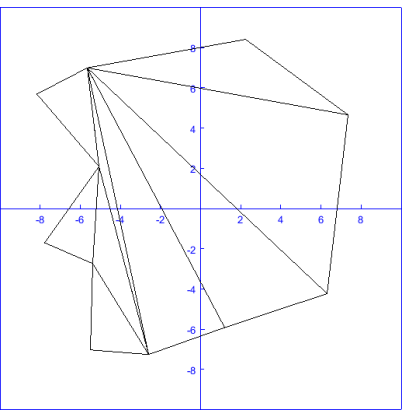




Abstract

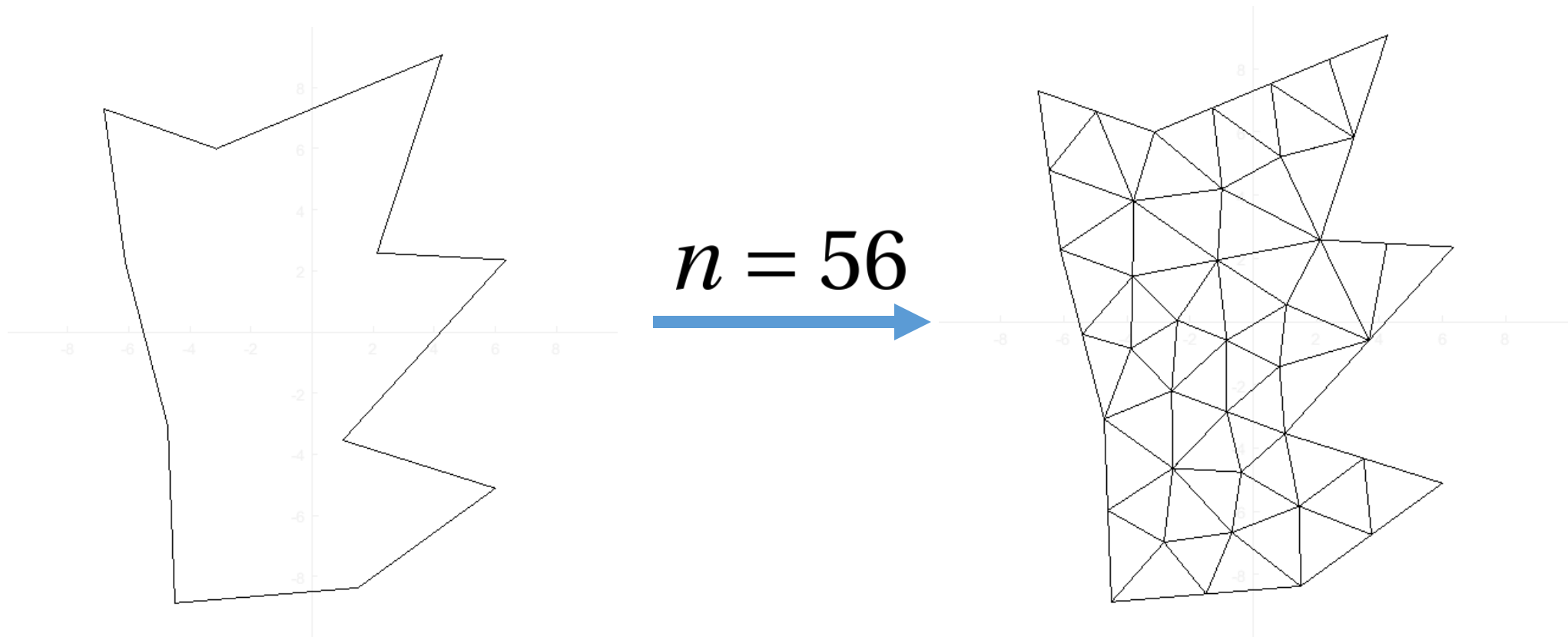


When utilizing the finite element method in two dimensions, one requires a suitable mesh of the domain they wish to solve on. In this talk we will go over the term suitable, an in depth approach to arrive at this goal, and strategies for programming implementations. This talk will additionally demonstrate the workings behind the culmination of this research: a program which allows users to create 2D triangle meshes for any domain they desire.



Our Goal!

- We wish to construct a program that will allow a user to create a mesh of a self chosen number triangles out of any polygon skeleton of their choosing.



Three Steps

Fan

- From our initial skeleton we parse the shape into triangles

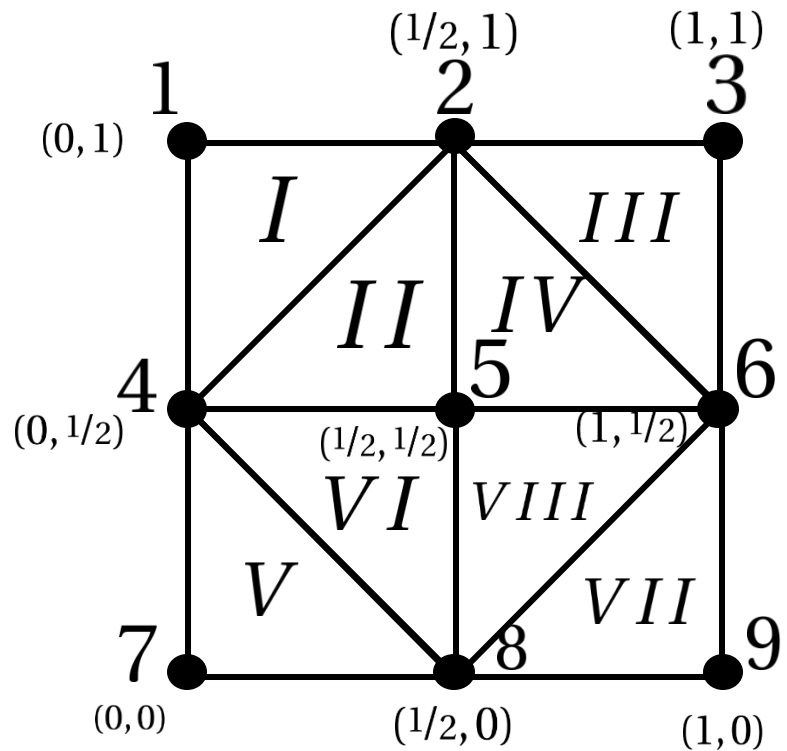
Add

- We create additional triangles until our user specified number is reached

Improve

- With the desired number of triangles how can we make changes that will improve the overall mesh?

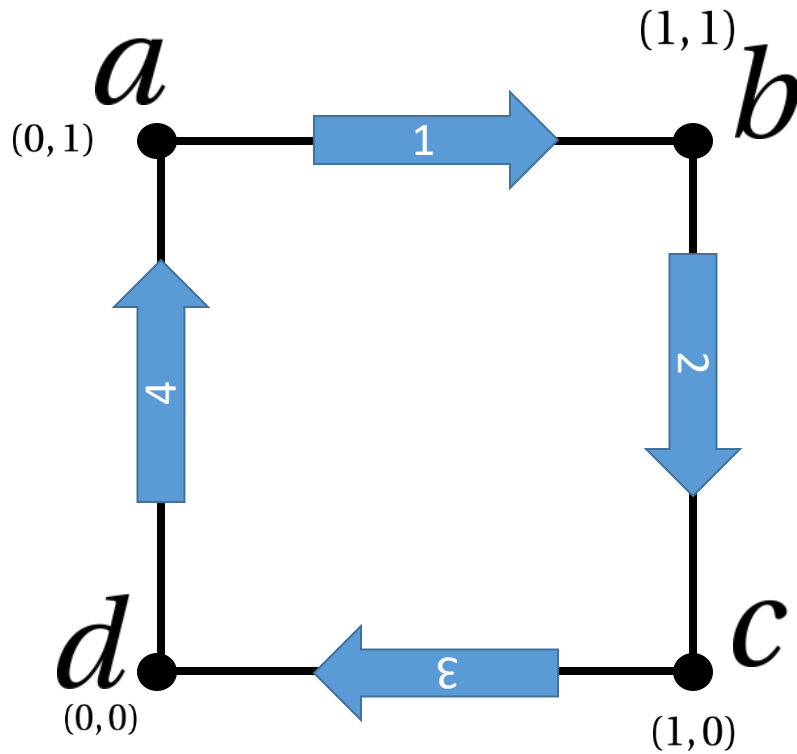
Programming Objects



$$V = \begin{bmatrix} 0 & 1 \\ 1/2 & 1 \\ 1 & 1 \\ 0 & 1/2 \\ 1/2 & 1/2 \\ 1 & 1/2 \\ 0 & 0 \\ 1/2 & 0 \\ 1 & 0 \end{bmatrix}$$

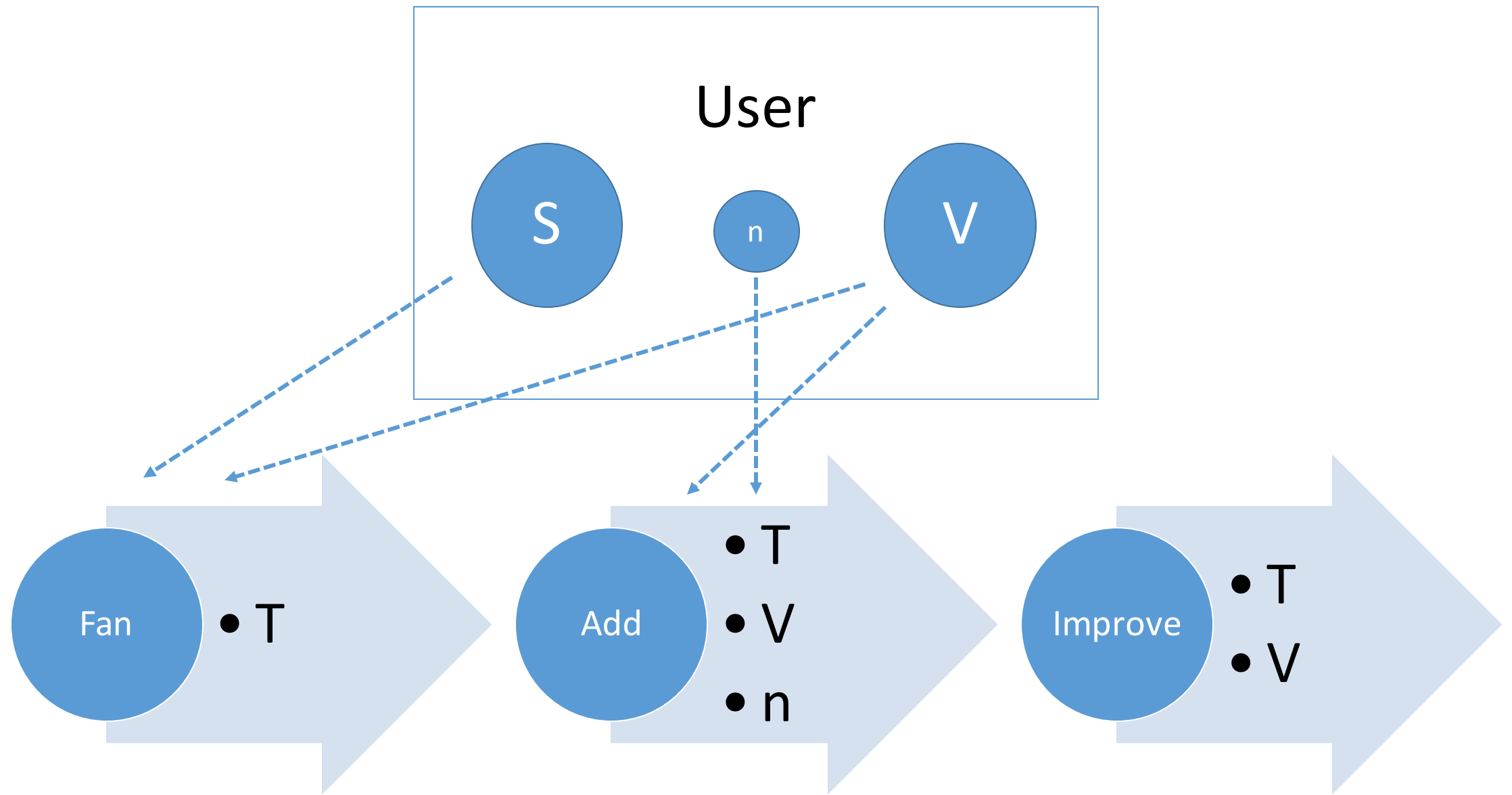
$$T = \begin{bmatrix} I & 1 & 2 & 4 \\ II & 2 & 4 & 5 \\ III & 2 & 3 & 6 \\ IV & 2 & 5 & 6 \\ V & 4 & 7 & 8 \\ VI & 4 & 5 & 8 \\ VII & 6 & 8 & 9 \\ VIII & 5 & 6 & 8 \end{bmatrix}$$

Programming Objects

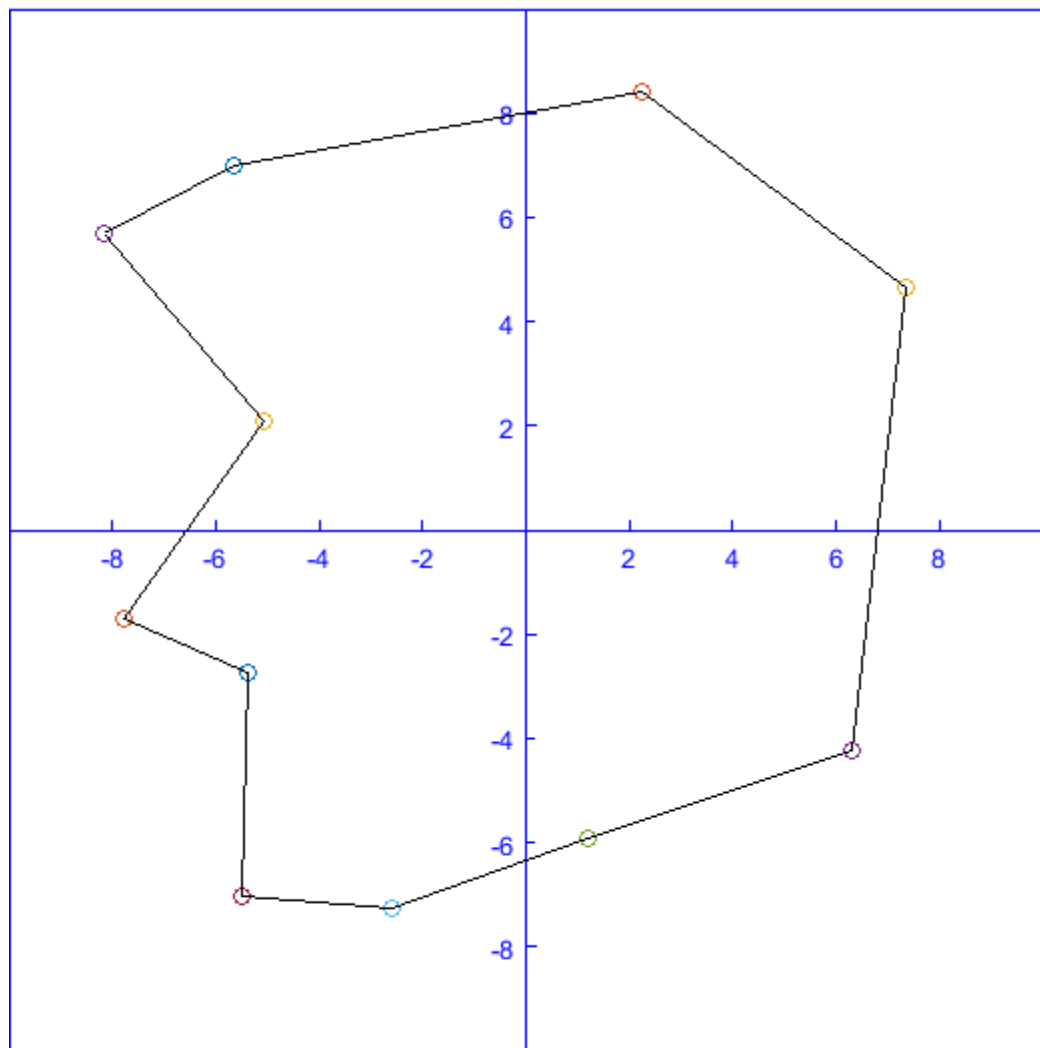


$$V = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{matrix} d \\ b \\ c \\ a \end{matrix}$$

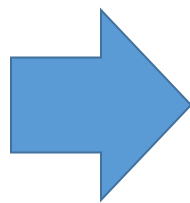
$$S = \begin{matrix} & a & b & c & d \\ \begin{bmatrix} 4 & 2 & 3 & 1 \end{bmatrix} \end{matrix}$$



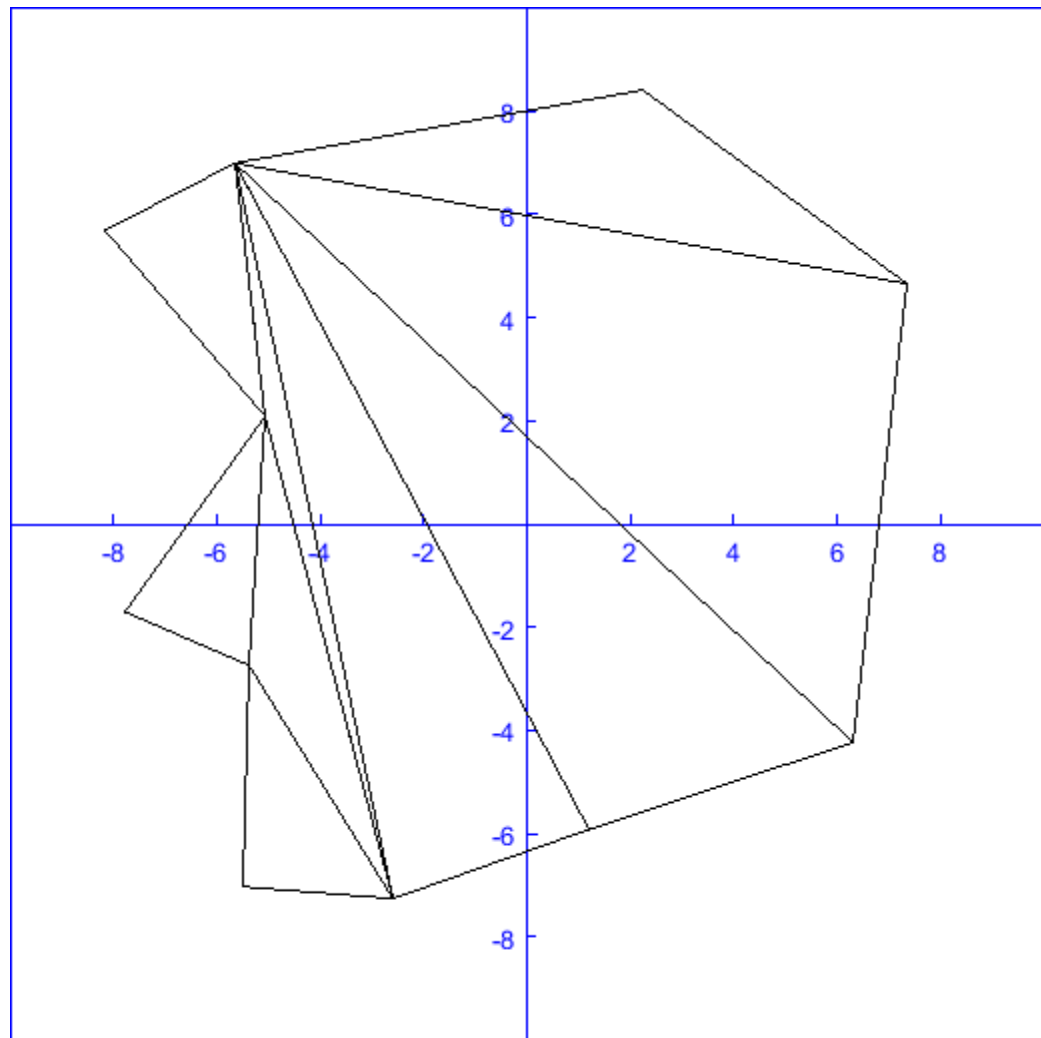
(V, S)



Fan



(T)

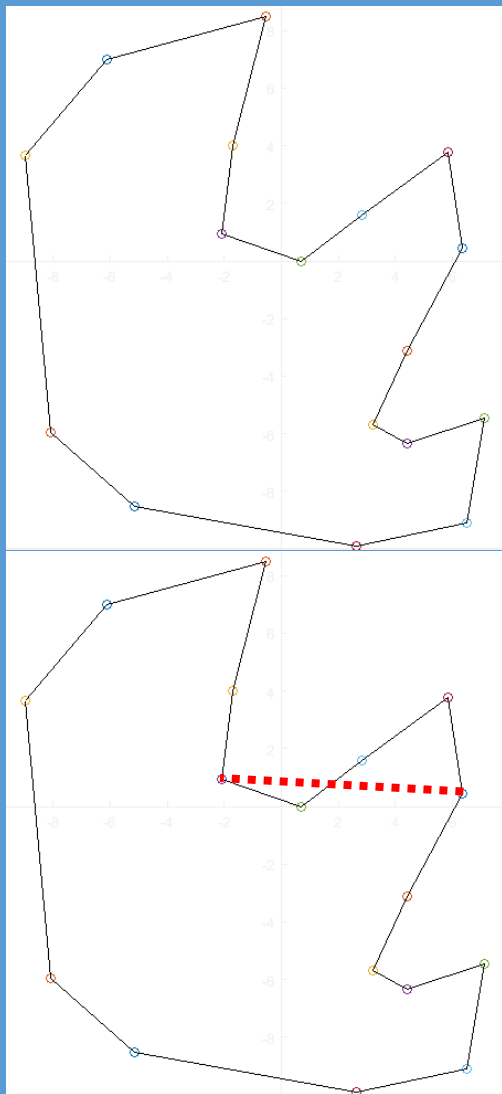


Fan Process Overview

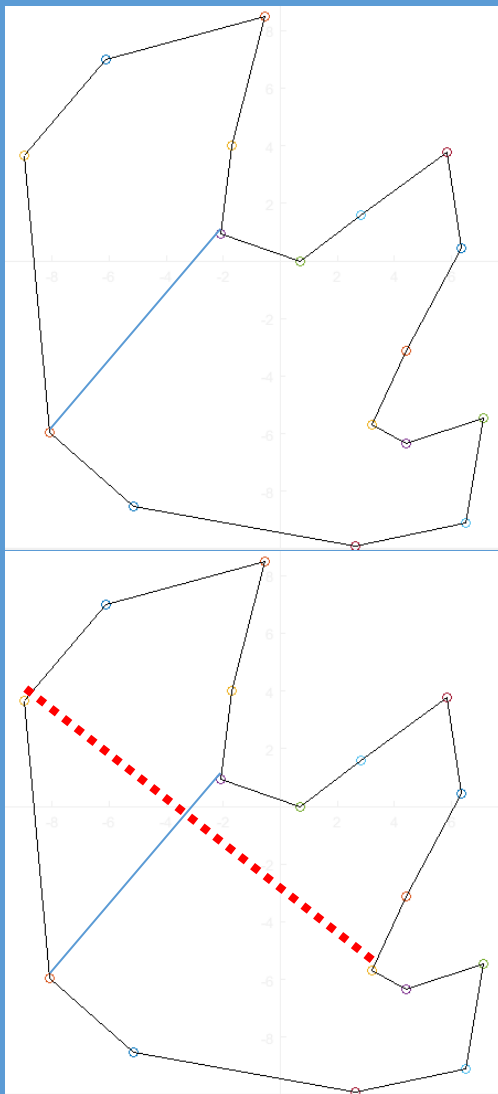
- For each vertex i
 - For each non adjacent vertex j
 - See if a newline can be made joining point i and j
 - If so save it
- Construct our T matrix

New Line Violations

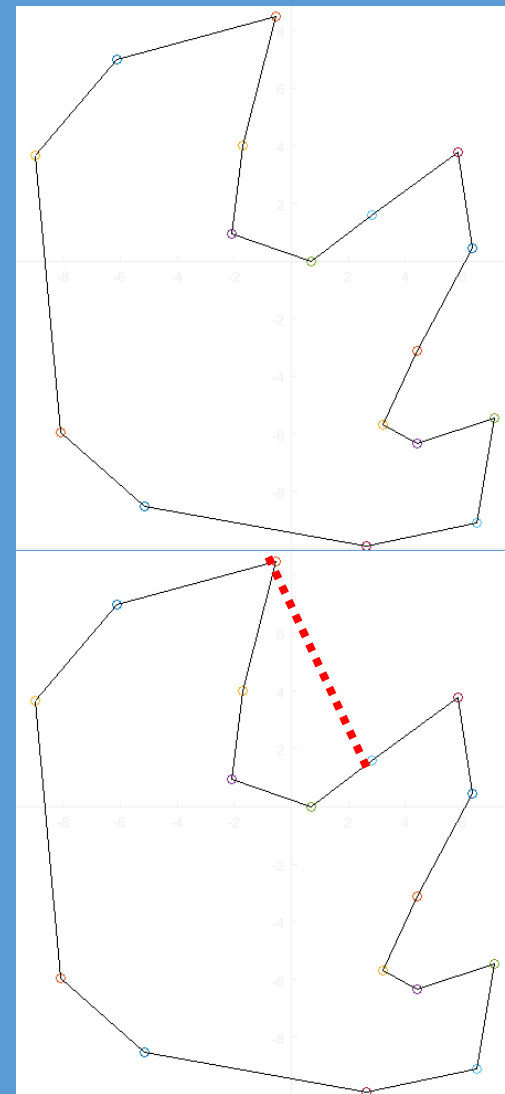
Original Skeleton



Other New Lines



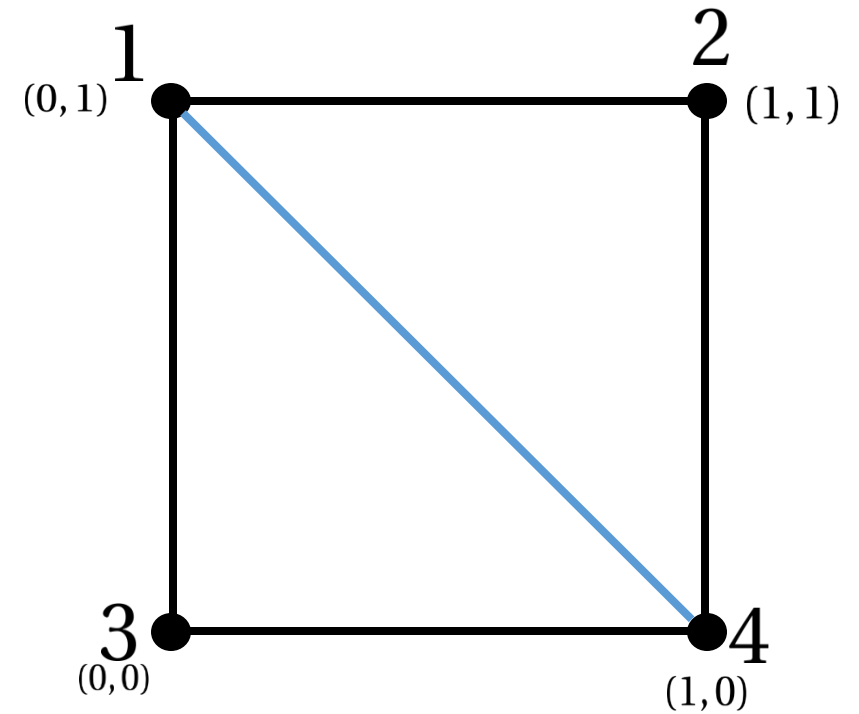
Outside the Shape



New Line Object

- Throughout this process we will wish to save all the eligible new lines that will not violate our original shape
- This will be done with the $nLine$ vector

$$nLine = [1 \ 4]$$



- Each pair represents the vertex numbers we wish to join

Line Crossing Violations

Original Edges

```
for a=1:numV
    if (a==numV)
        b=1;
    else
        b=a+1;
    end

    intersection(a,b,i,j);
```

Where intersection(A,B,C,D) returns true if the lines AB and CD intersect and false if not.

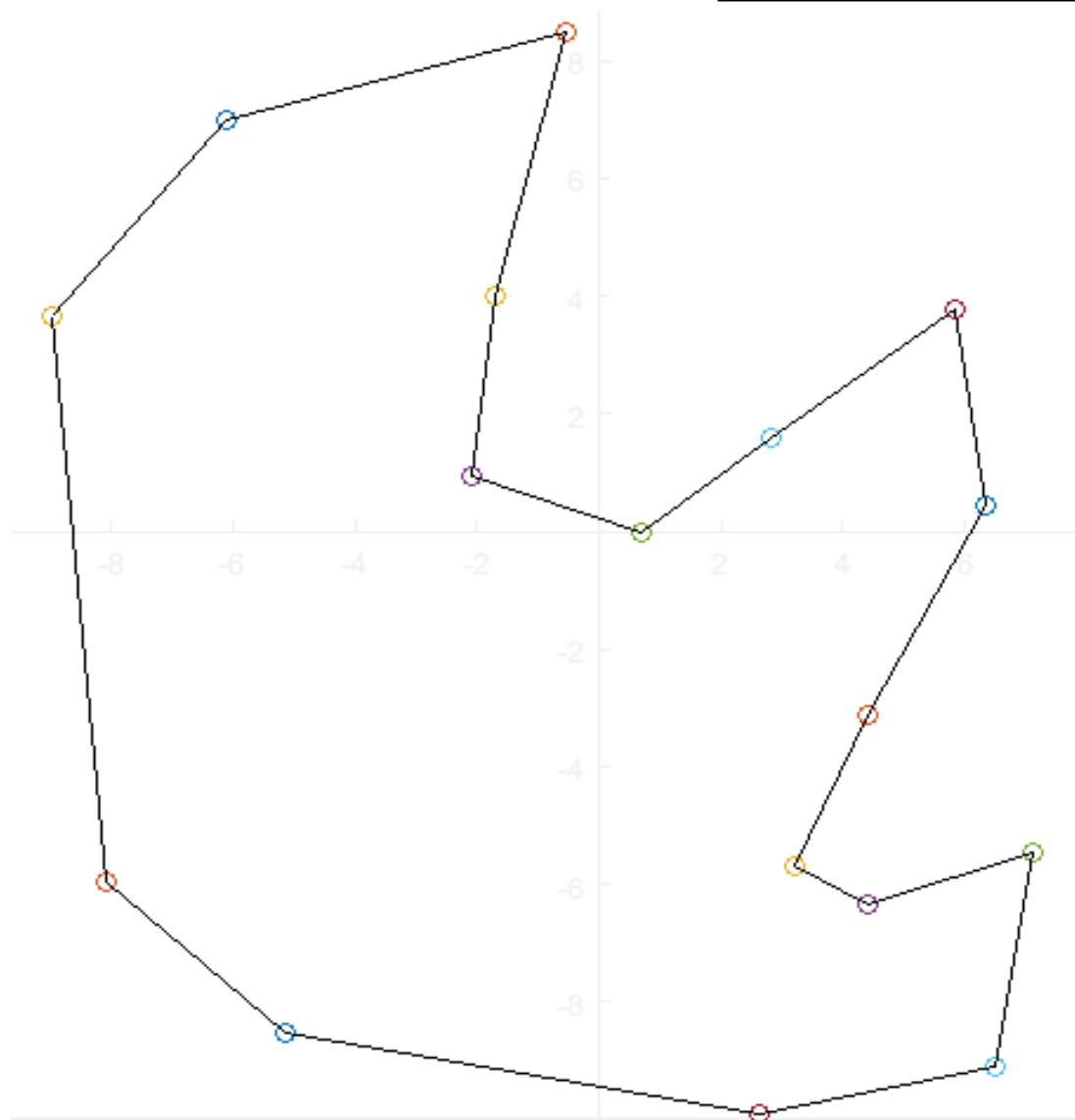
New Lines

```
a = 1;
while a < (size(nLine))-1
    b = a+1;
    q = nLine(1,a);
    r = nLine(1,b);

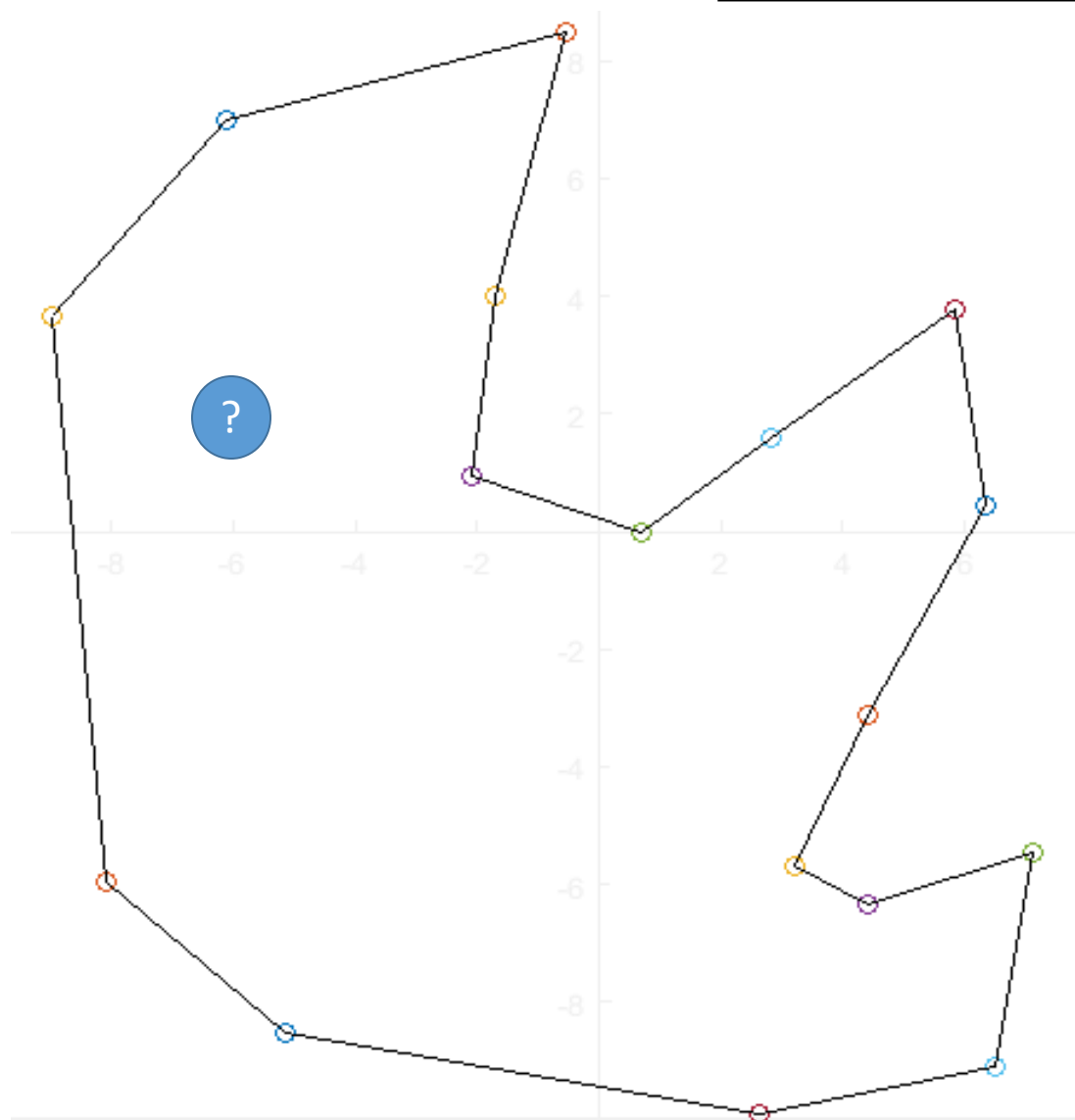
    if ((q==i) && (r==j)
        || (q==j) && (r==i))
        break;
    end

    intersection(q,r,i,j);
```

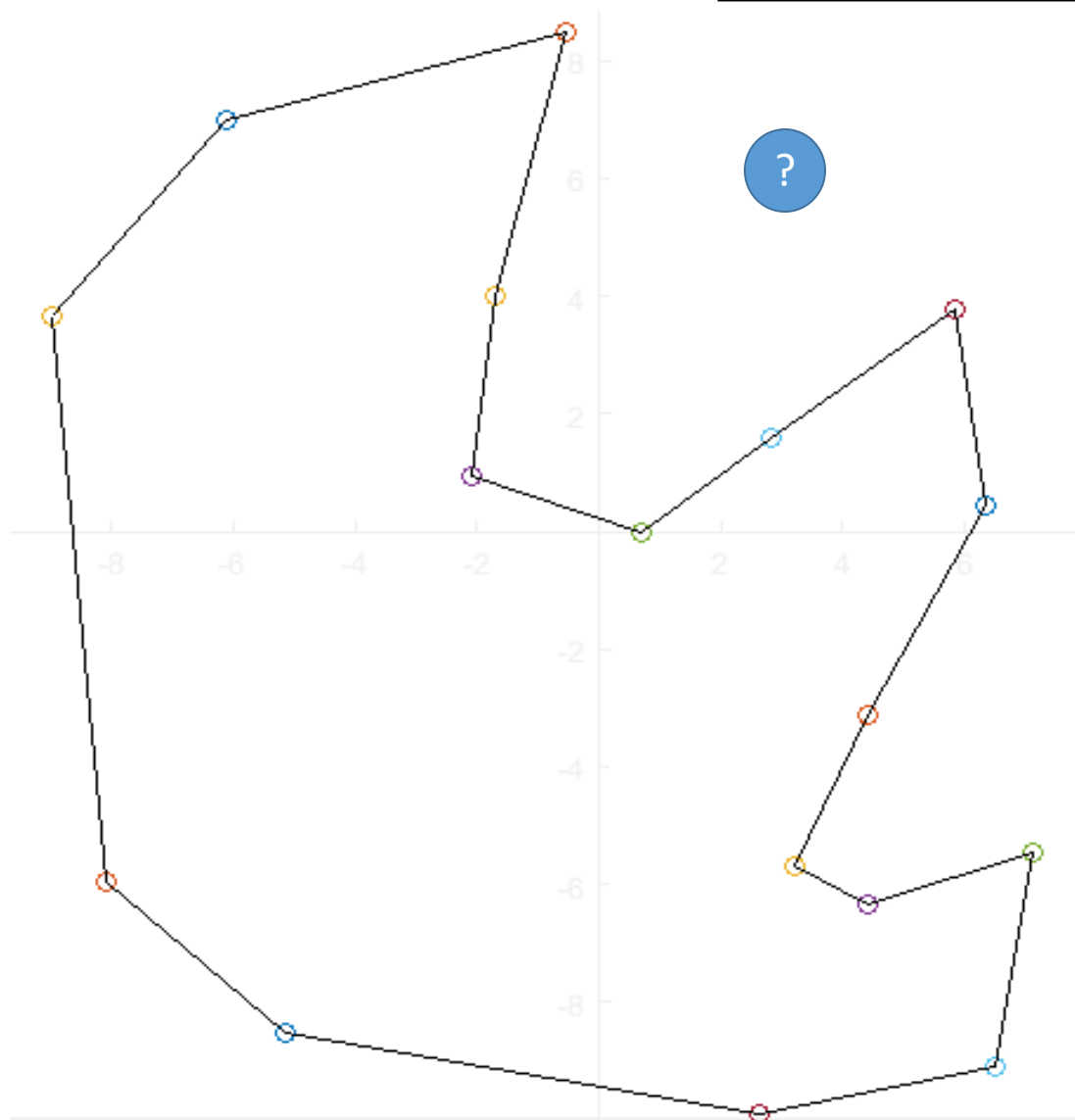
Inside or Outside?



Inside or Outside?

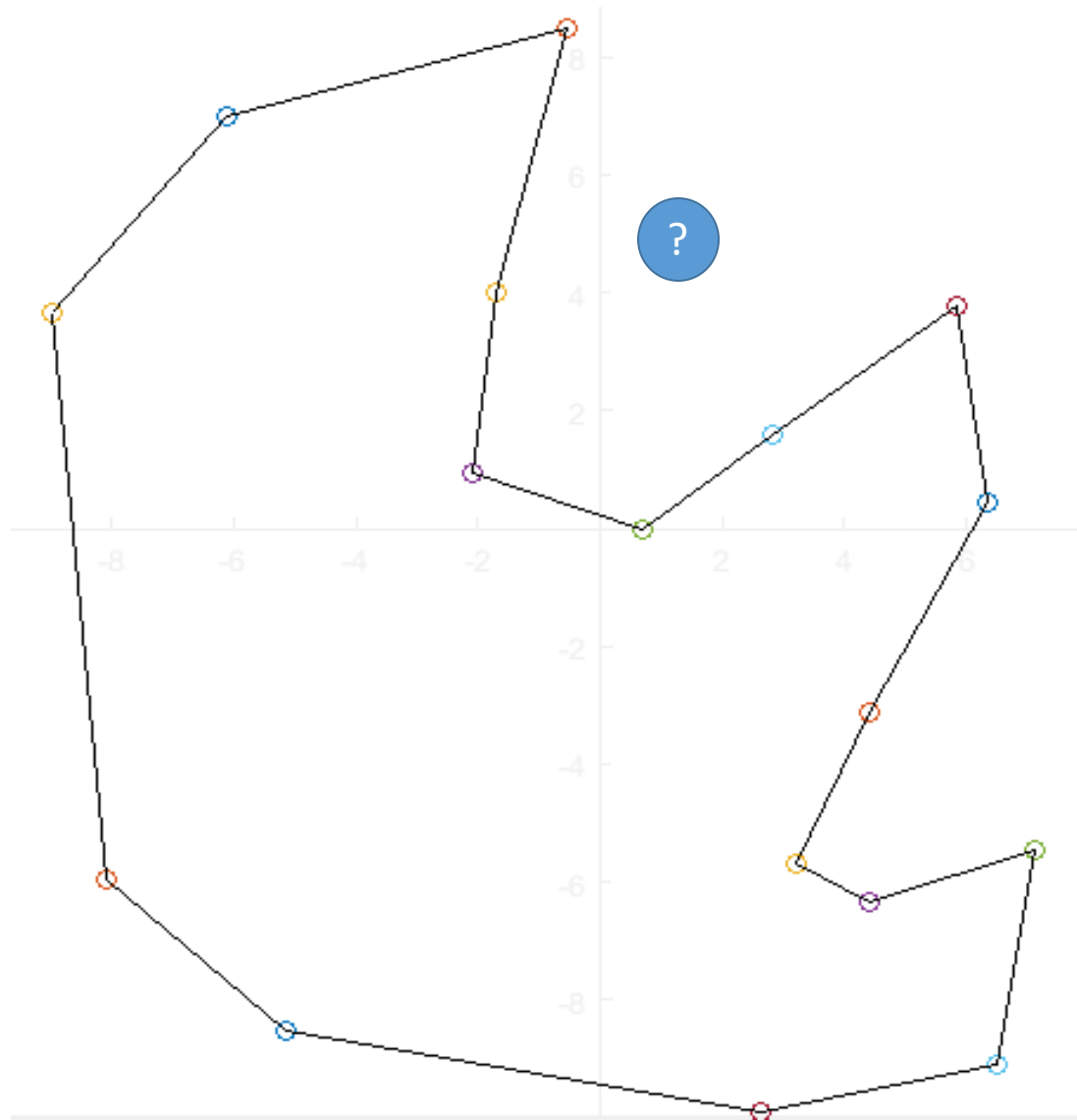


Inside or Outside?



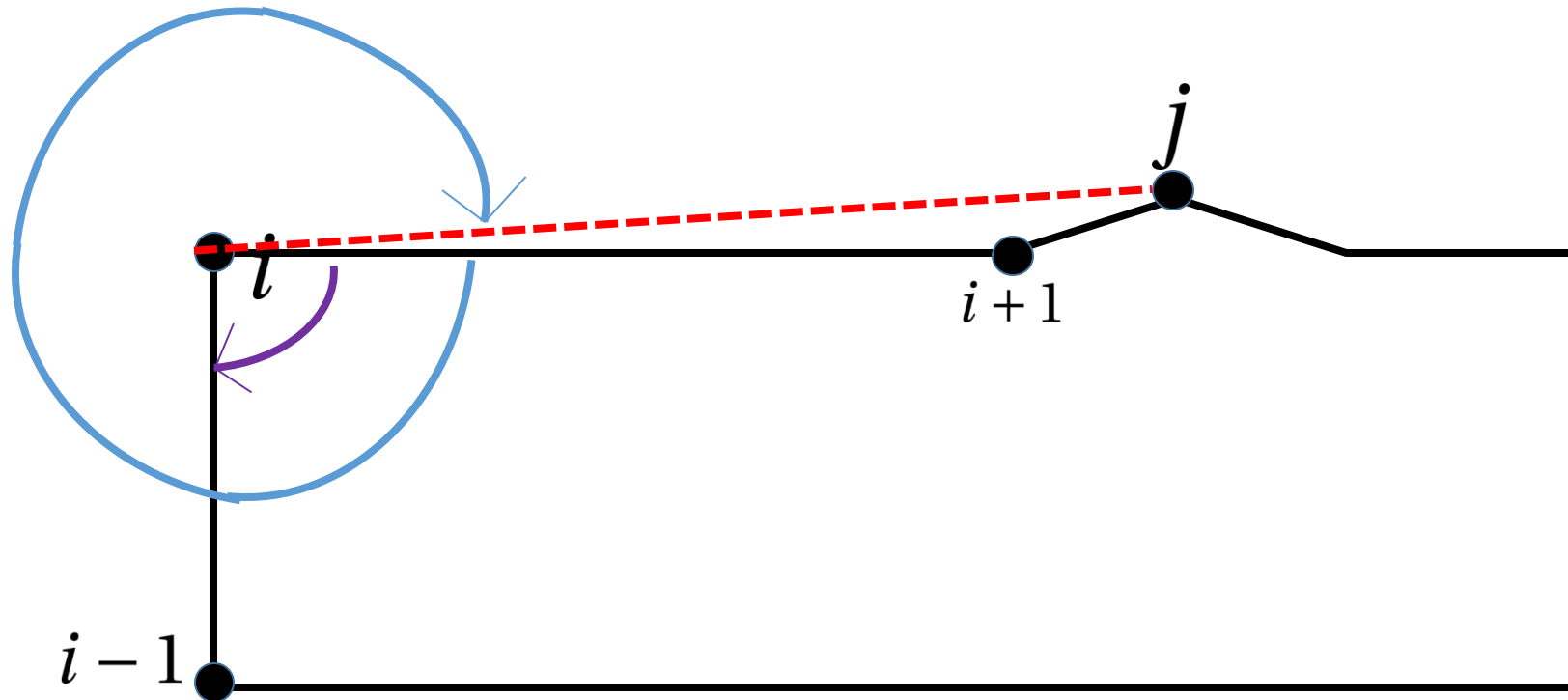
Your Turn Computer

?



Teaching the Computer

To do this we will check to see if the clockwise angle between line $(i,i+1)$ and line (i,j) is less than that between line $(i,i+1)$ and line $(i,i-1)$.



Clockwise Angle Between

```
function[Angle] = angleBetween(startV,middleV,endV)
```

```
    v1=startV-middleV;
```





```
    v2=endV-middleV;
```

```
    ang = atan2((v1(1)*v2(2)-v2(1)*v1(2)),(v1(1)*v2(1)+v1(2)*v2(2)));
```

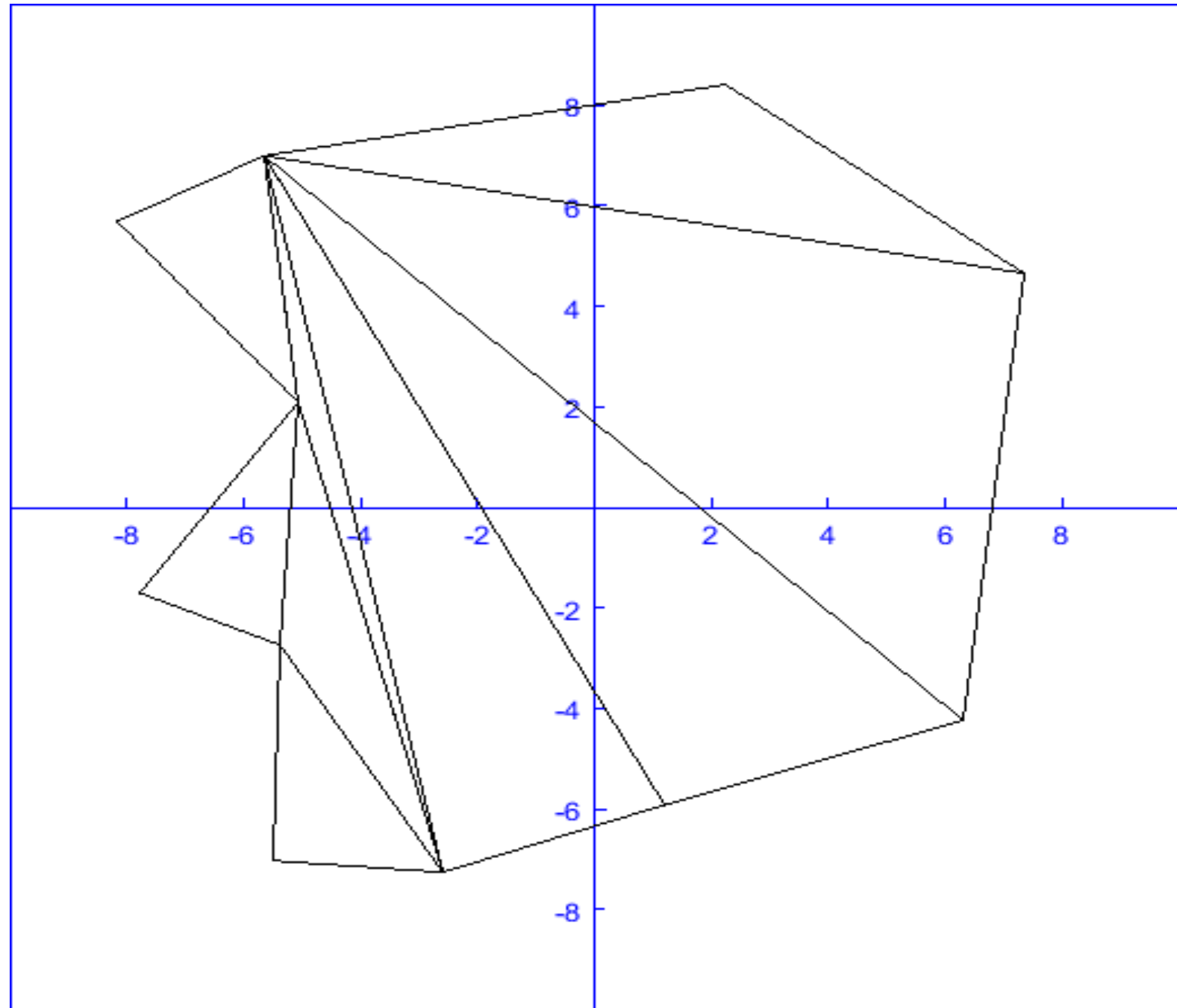
```
    Angle = mod(-180/pi * ang, 360);
```

```
end
```

Fan Process Overview

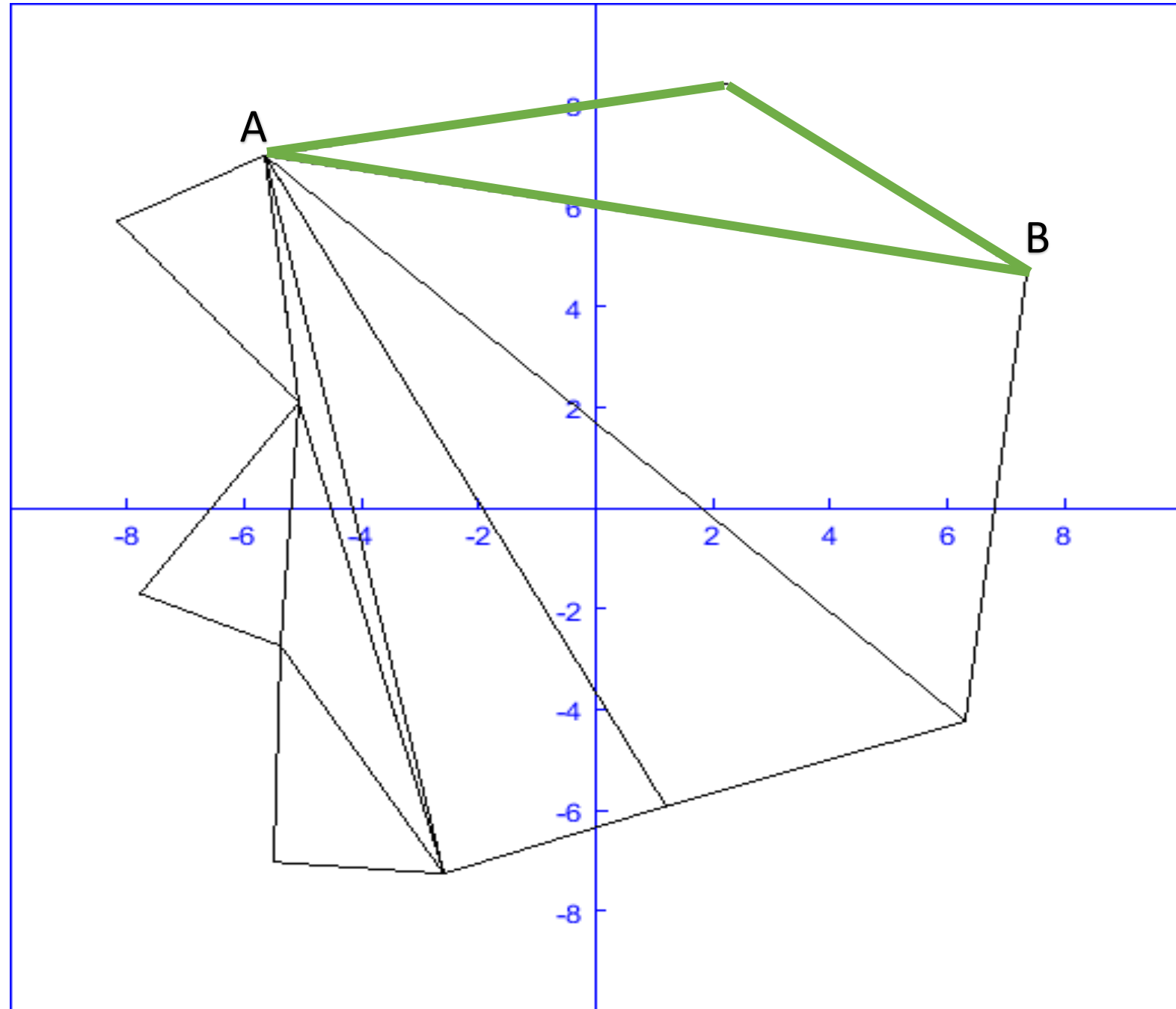
- For each vertex i 
 - For each non adjacent vertex j 
 - See if a newline can be made joining point i and j 
 - If so save it 
- Construct our T matrix

Three Types of New Triangles



1 New Line

We can accomplish this by stepping through our new line vector and check if any of the pairs (A,B) are only two away from one another on the skeleton



```
for nlIndex=1:2:z
```

```
    A = nLine(nlIndex);
```

```
    B = nLine(nlIndex+1);
```

```
    if (S(A) == (S(B)-2))
```

```
        %finds the middle vertex as i
```

```
        for i=1:numV
```

```
            if (S(i) == S(B)-1)
```

```
                break;
```

```
            end
```

```
        end
```

```
        %create new triangle A,B,i
```

```
        nT = nT+1;
```

```
        T(nT,1) = A;
```

```
        T(nT,2) = B;
```

```
        T(nT,3) = i;
```

```
    elseif ((S(A)==2) && (S(B)==numV))
```

```
        %finds the middle vertex as i
```

```
        for i=1:numV
```

```
            if (S(i) == 1)
```

```
                break;
```

```
            end
```

```
        end
```

```
        %create new triangle A,B,i
```

```
        nT = nT+1;
```

```
        T(nT,1) = A;
```

```
        T(nT,2) = B;
```

```
        T(nT,3) = i;
```

```

elseif ((S(A)==1) && (S(B)==numV-1))

    %finds the middle vertex as i
    for i=1:numV
        if (S(i) == numV)
            break;
        end
    end

    %create new triangle A,B,i
    nT = nT+1;

    T(nT,1) = A;
    T(nT,2) = B;
    T(nT,3) = i;

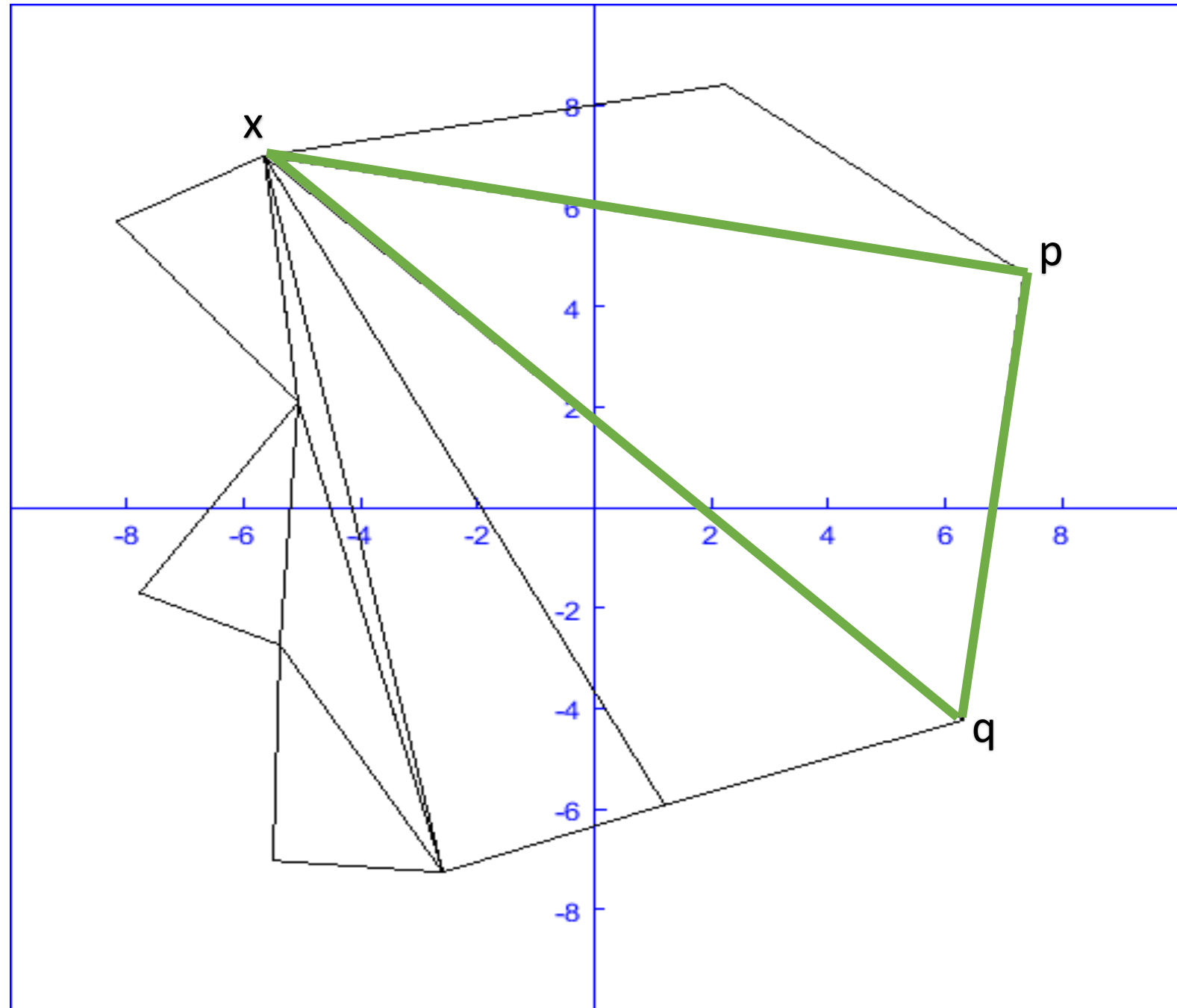
end

end

```

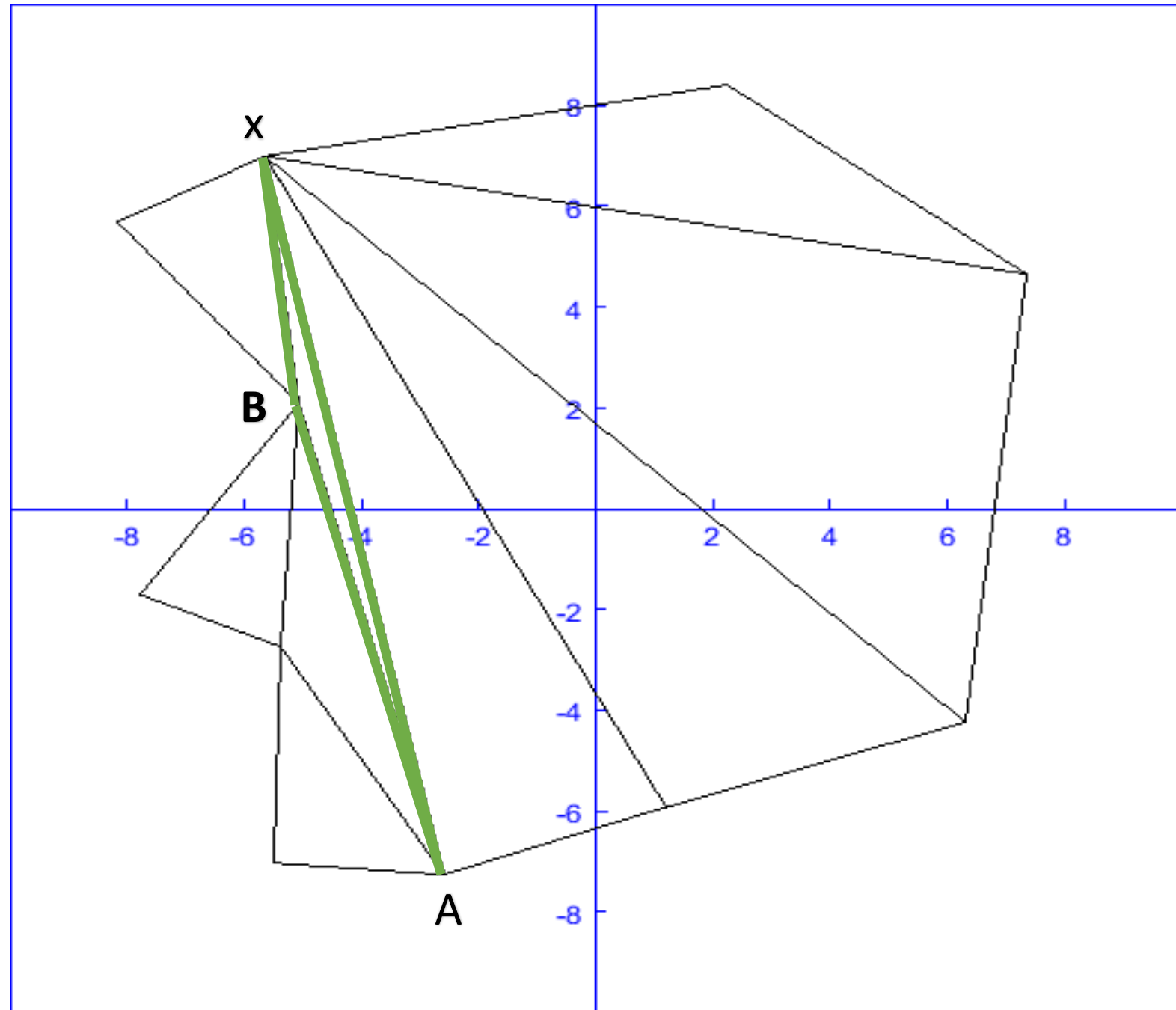
2 New Lines

We will go through each edge (p,q) and look for any (p,x) (q,x) pairings in the new lines



3 New Lines

For each newline (A,B) we will look through all the other newlines for (A,x) and (B,x) for $x \neq A,B$



```
for Aindex=1:2:z
```

```
    A=nLine (Aindex);
```

```
    B=nLine (Aindex+1);
```

```
    for AAindex=Aindex+2:2:z
```

```
        if (nLine (AAindex)==A)
```

```
            BB=nLine (AAindex+1);
```

```
        else
```

```
            break;
```

```
        end
```

```
    for AAAindex=AAindex+2:2:z
```

```
        AAA=nLine (AAAindex);
```

```
        BBB=nLine (AAAindex+1);
```

```
        if ( (AAA==B)  &&  (BBB==BB) )
```

```
            %save triangle
```

```
            nT=nT+1;
```

```
            T (nT,1) =A;
```

```
            T (nT,2) =B;
```

```
            T (nT,3) =BB;
```

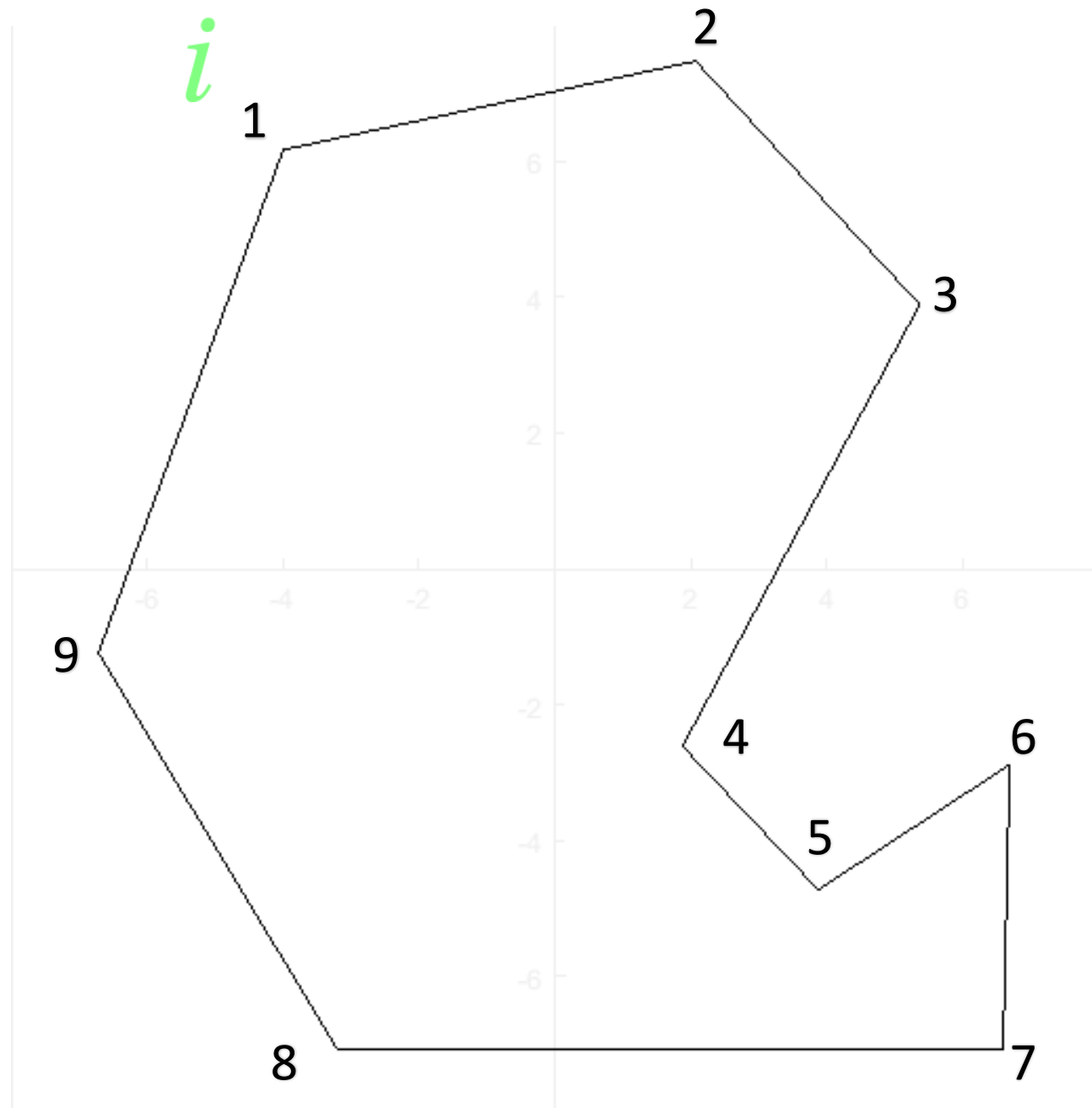
```
        end
```

```
    end
```

```
end
```

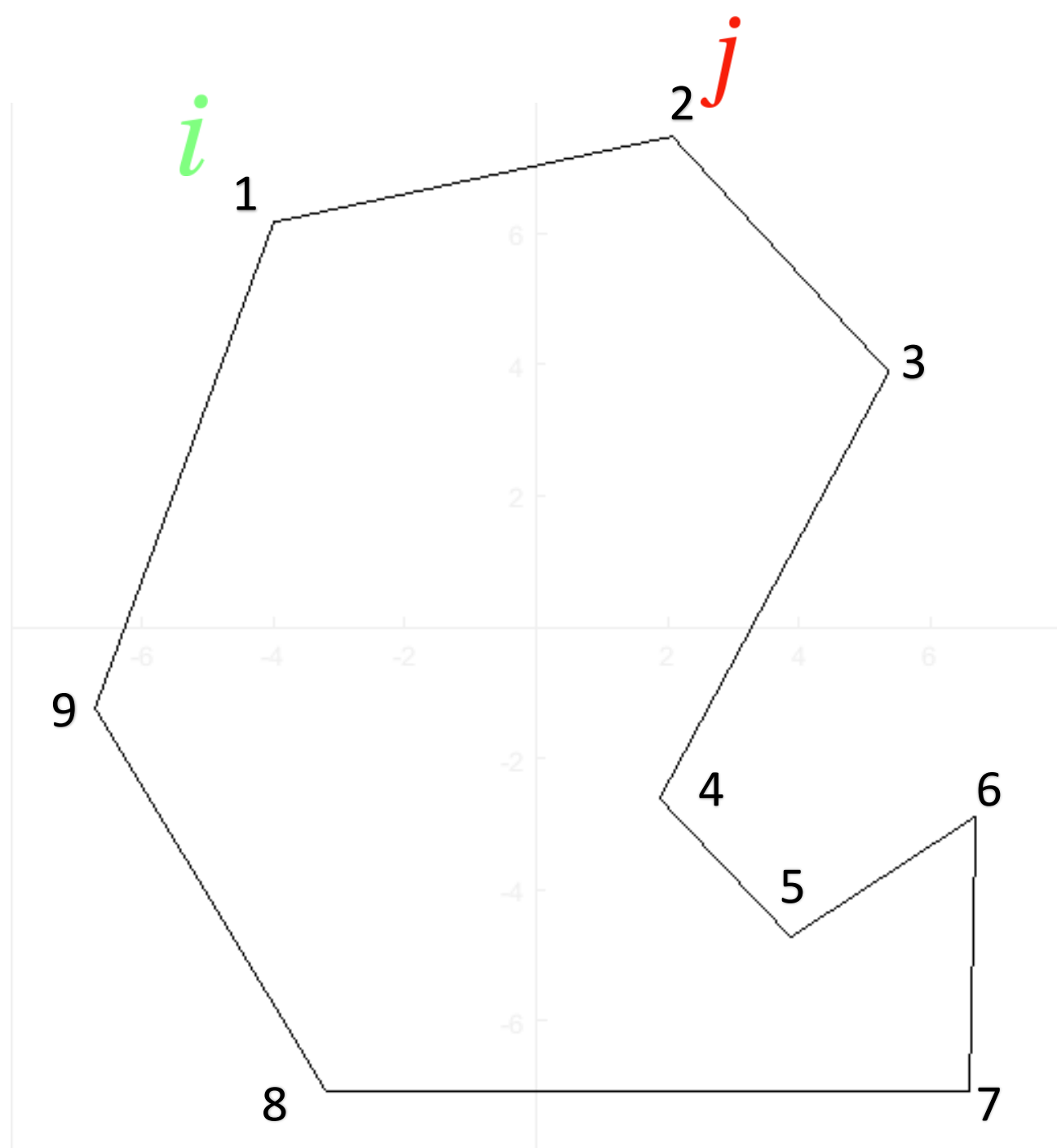
```
end
```

Lets Fan!



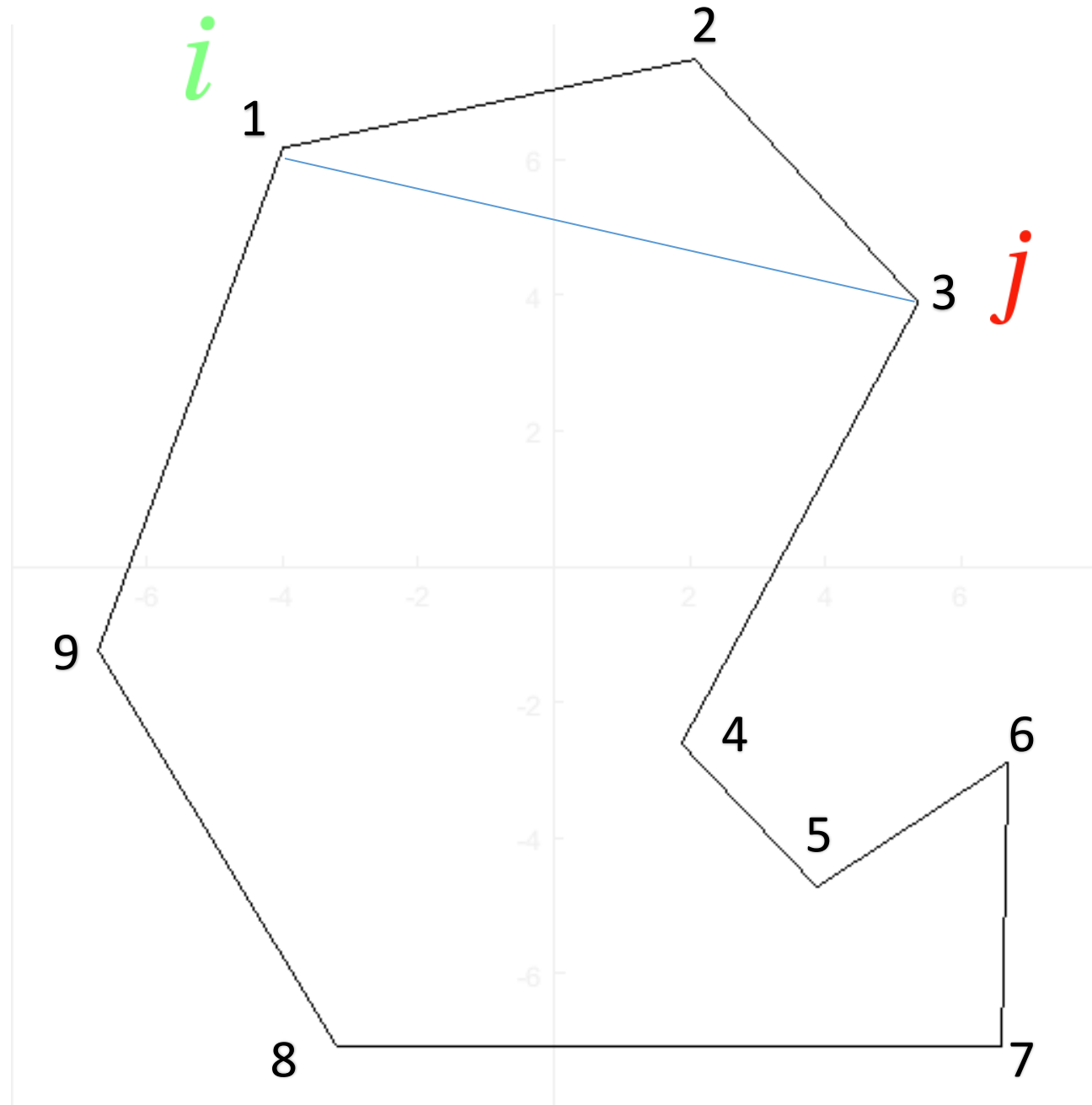
Making New Lines

$nLine = []$



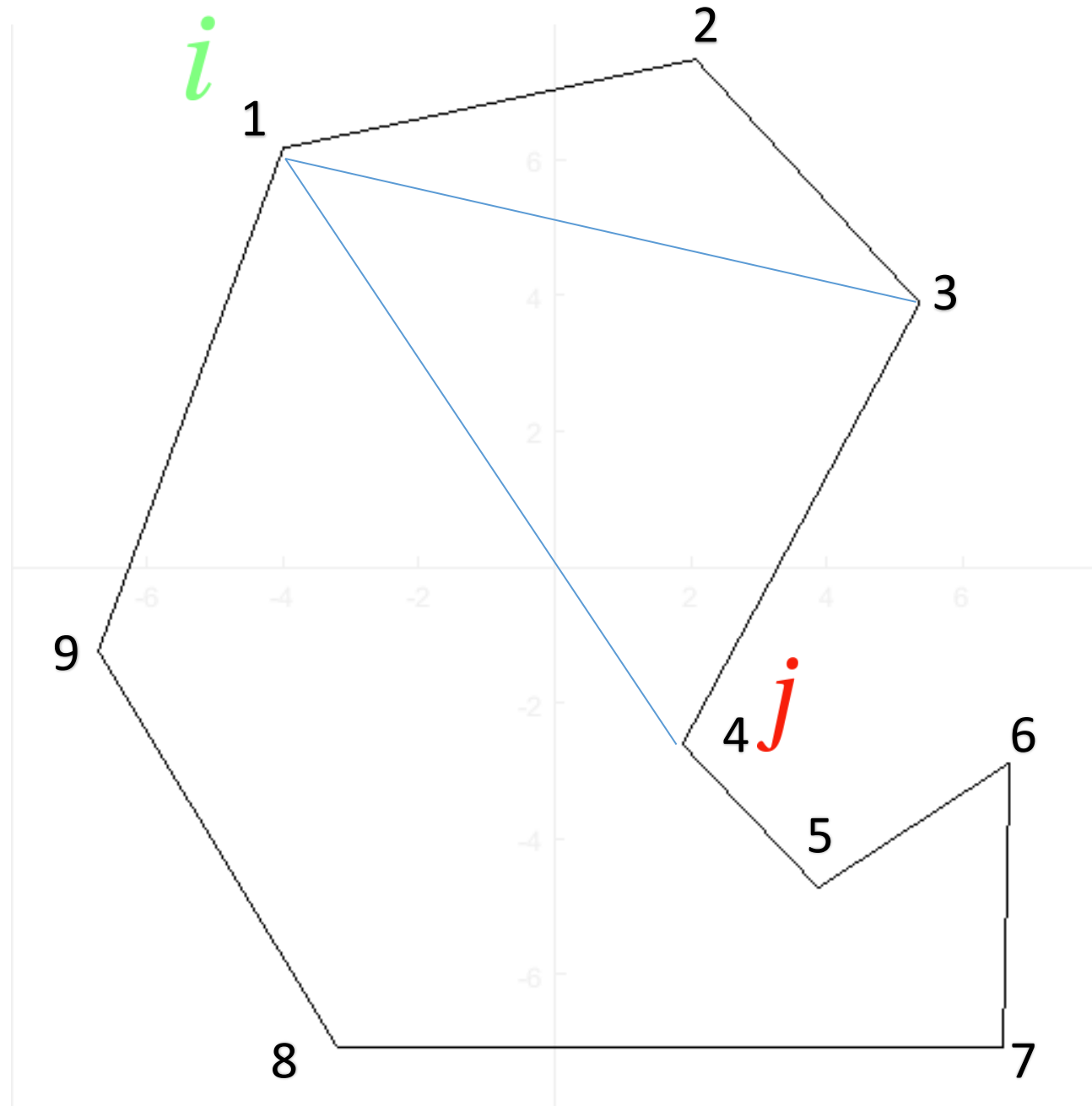
Making New Lines

$nLine = [1\ 3]$



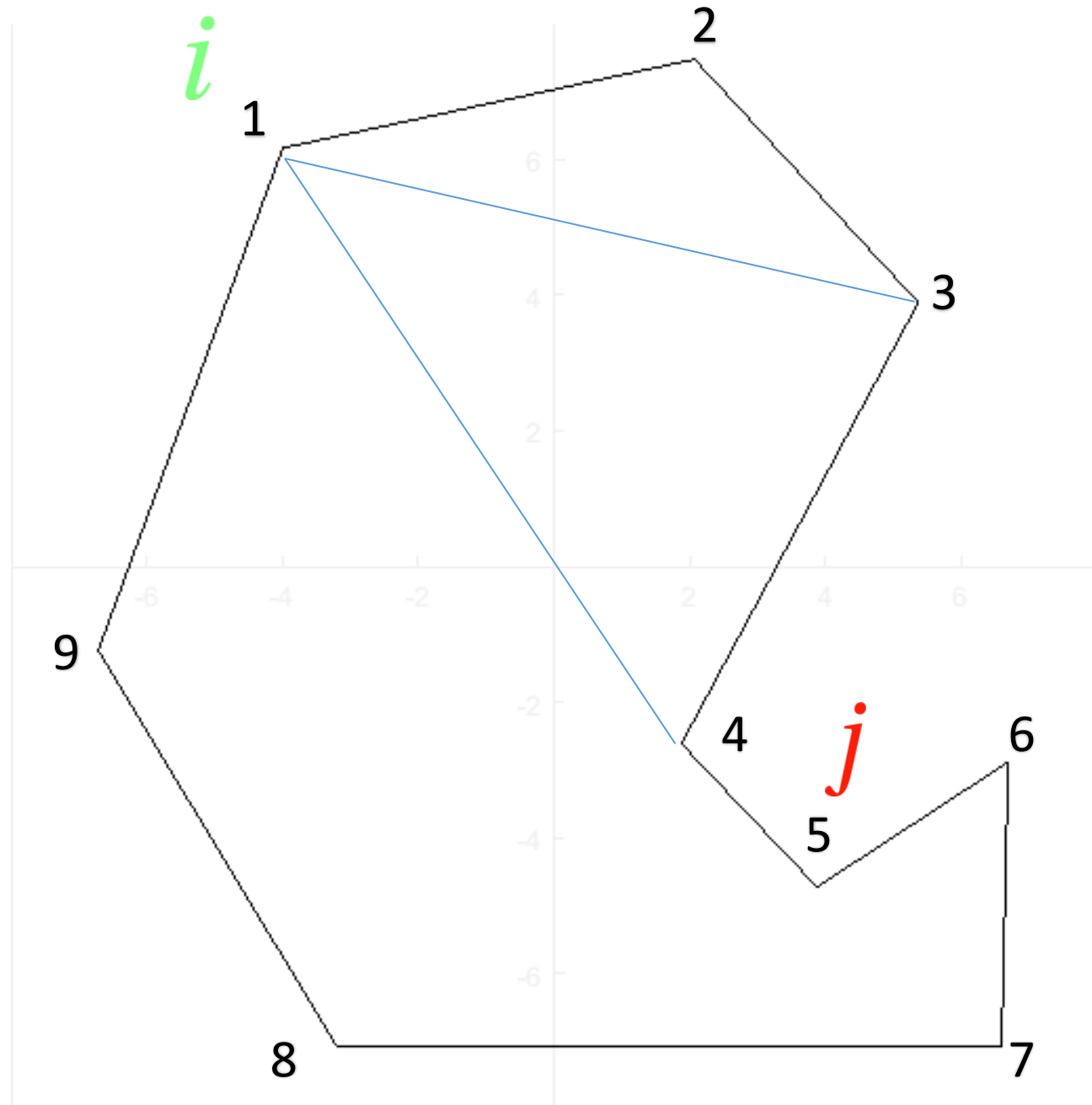
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \end{bmatrix}$$



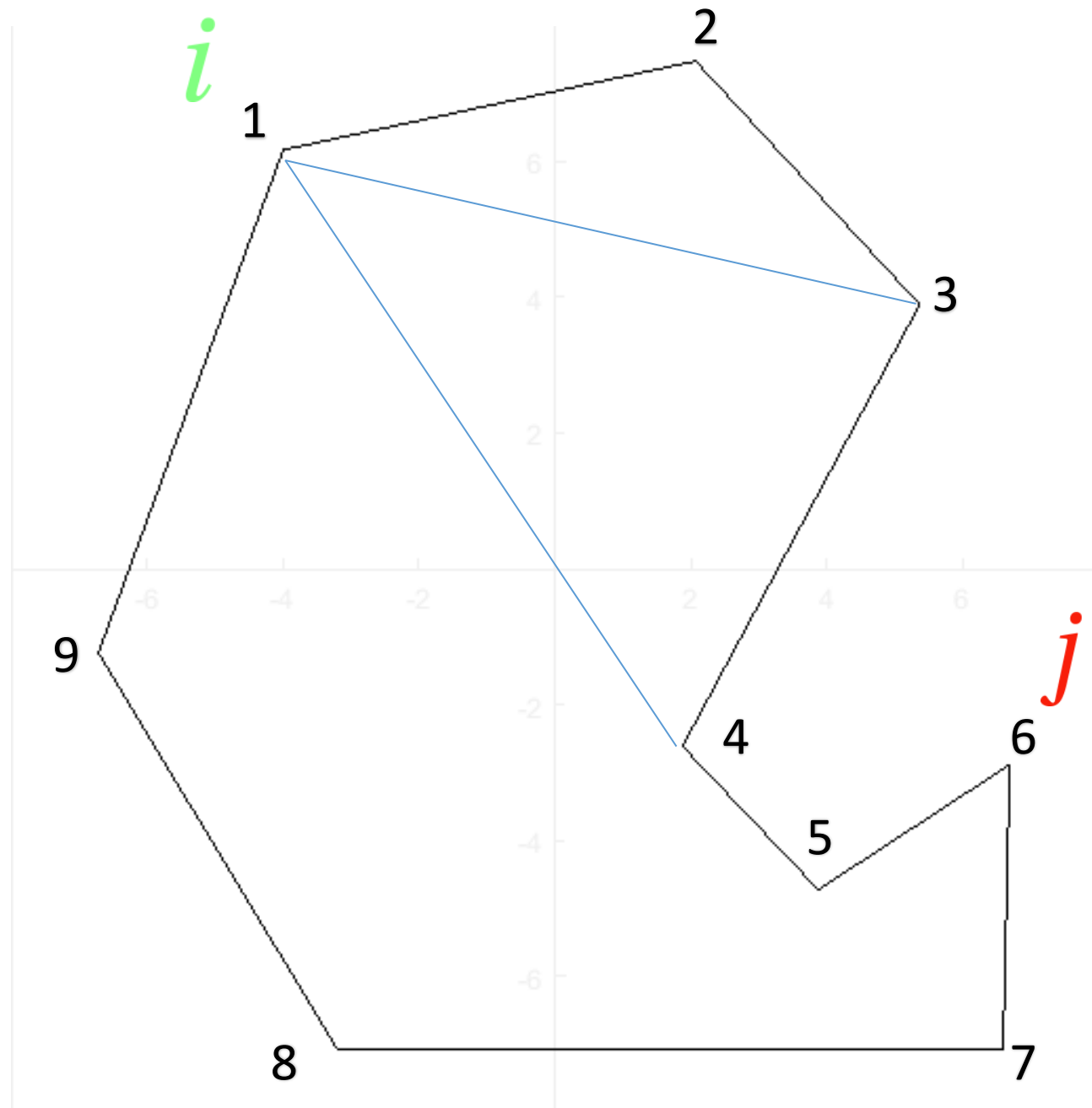
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \end{bmatrix}$$



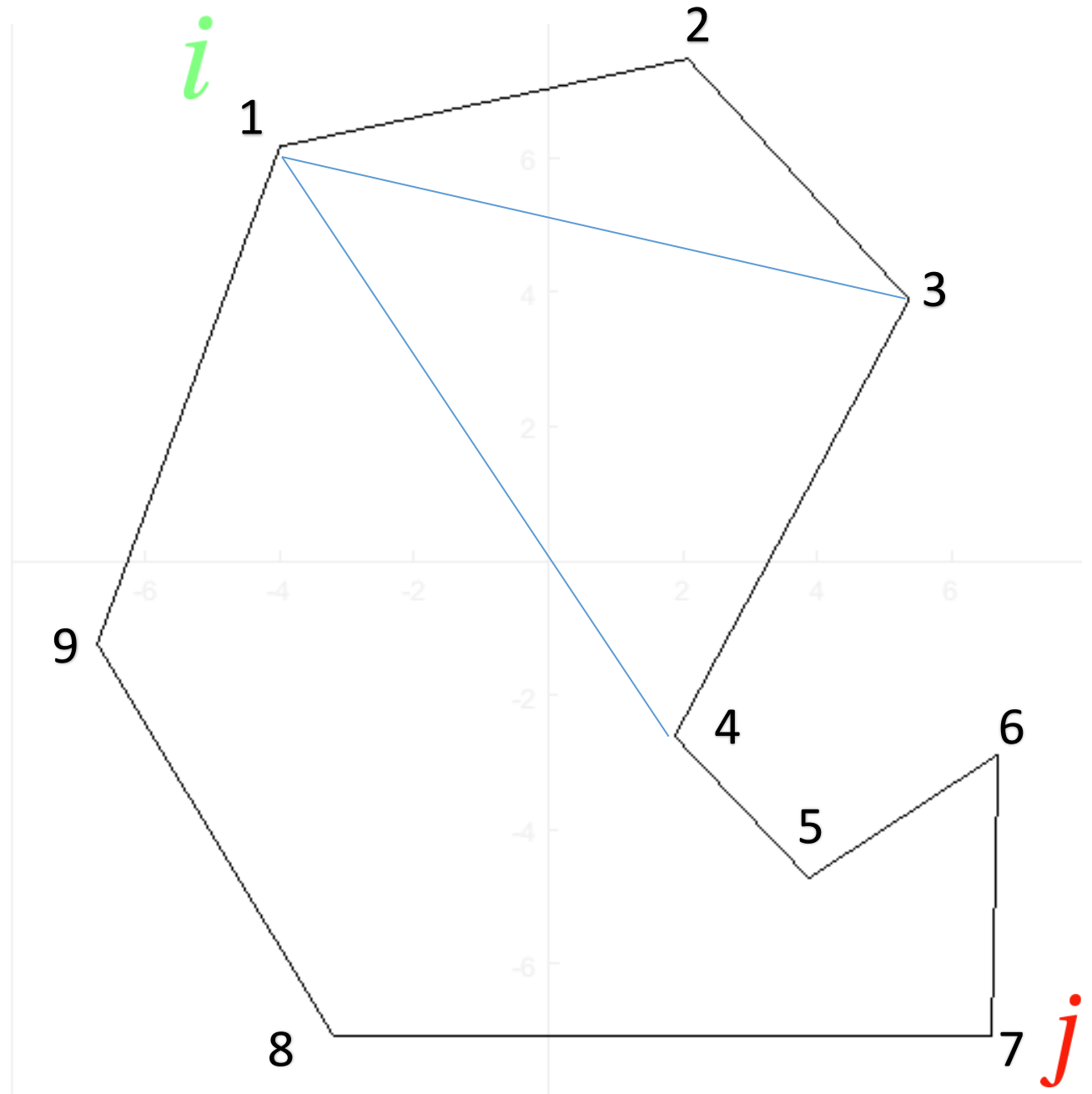
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \end{bmatrix}$$



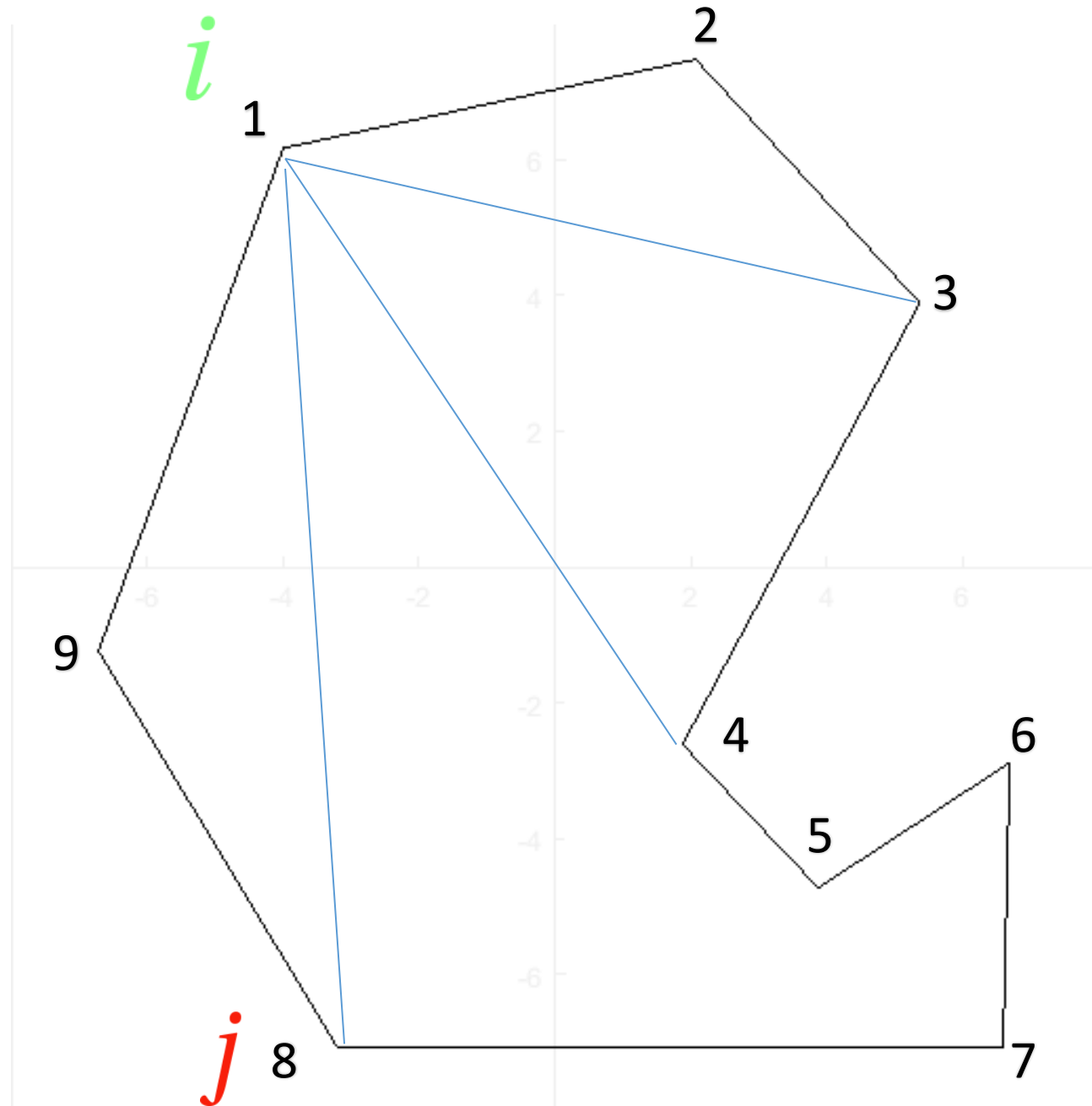
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \end{bmatrix}$$



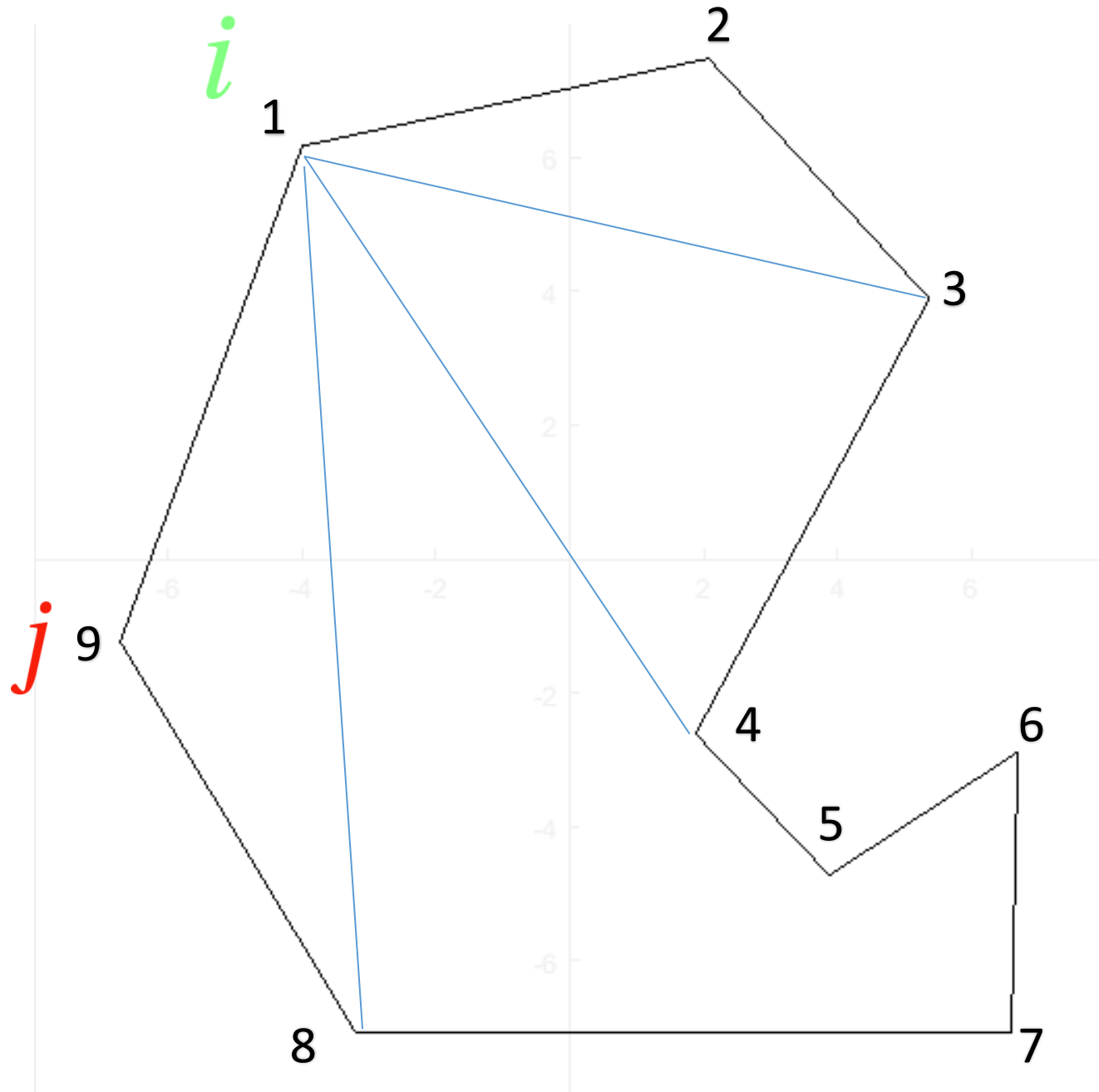
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \end{bmatrix}$$



Making New Lines

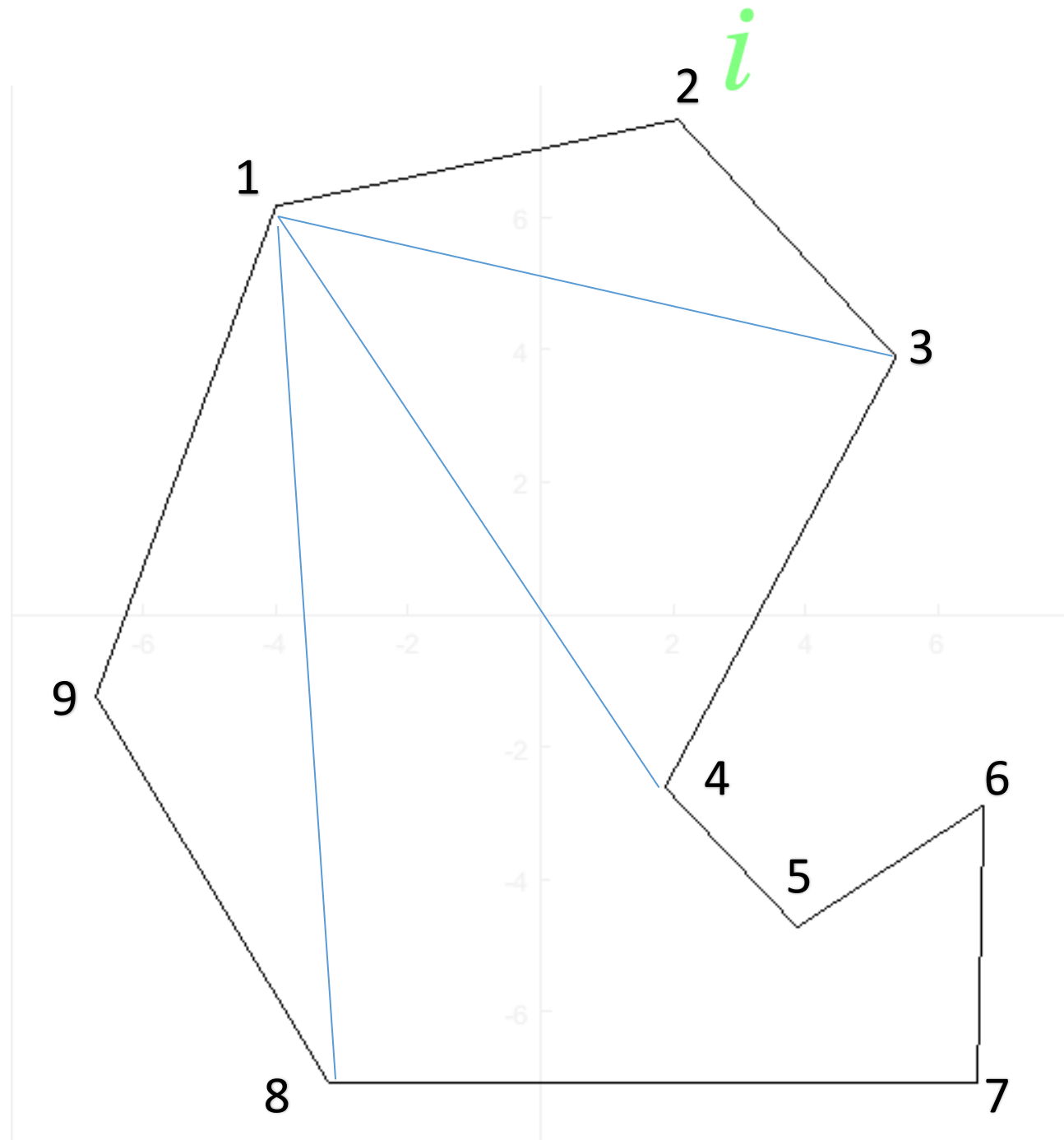
$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \end{bmatrix}$$



Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \end{bmatrix}$$

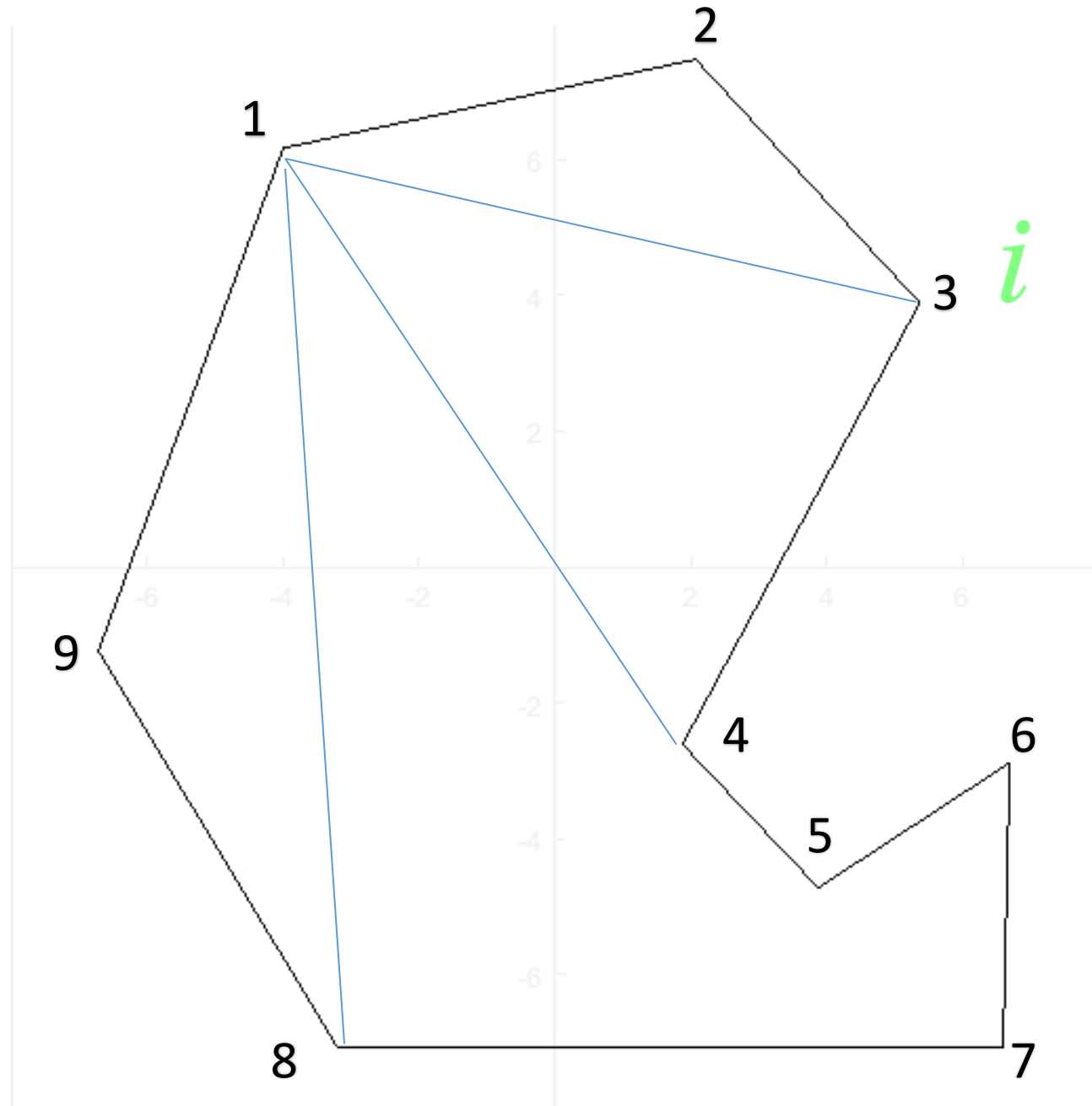
No *j* works



Making New Lines

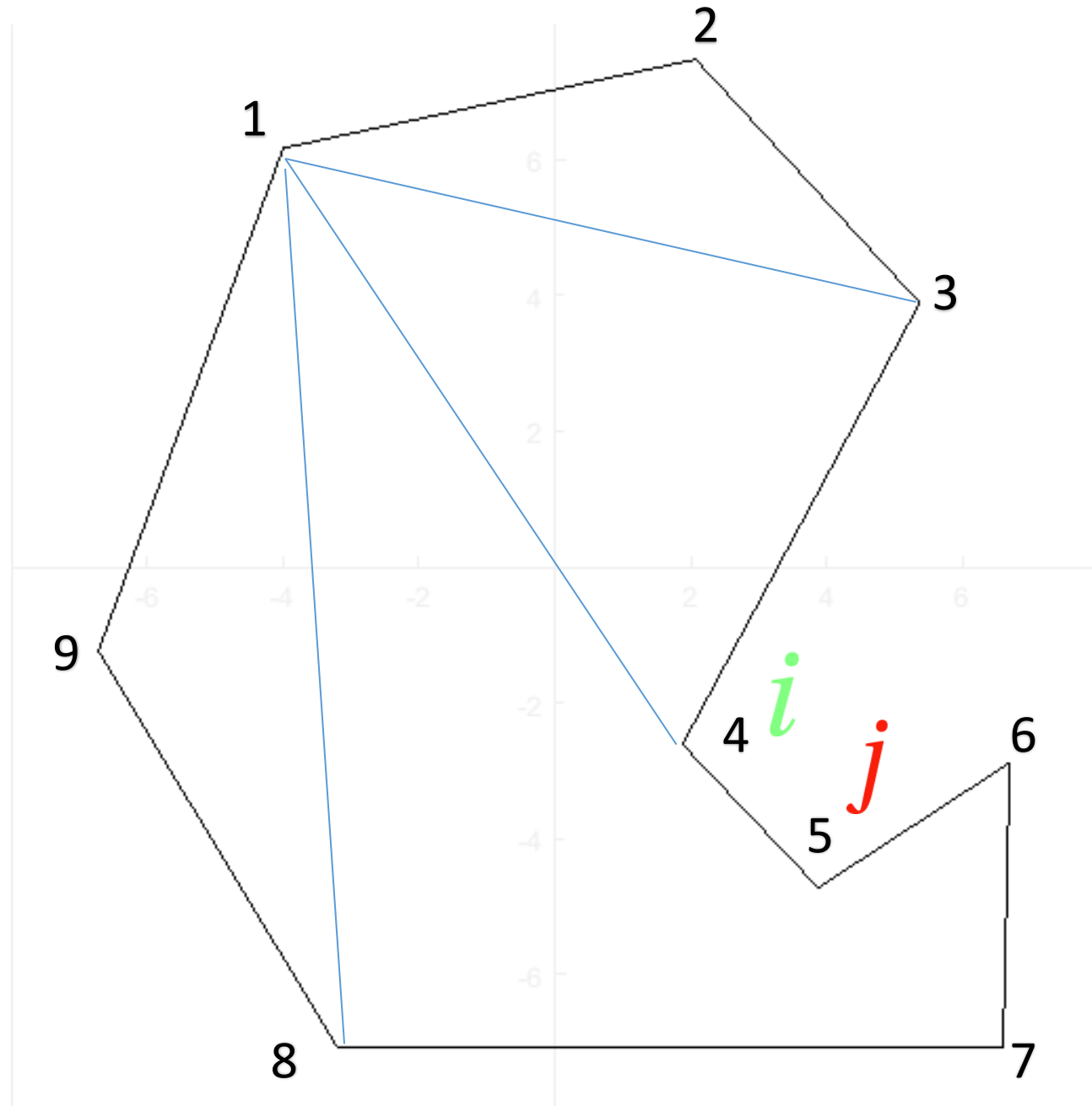
$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \end{bmatrix}$$

No *j* works



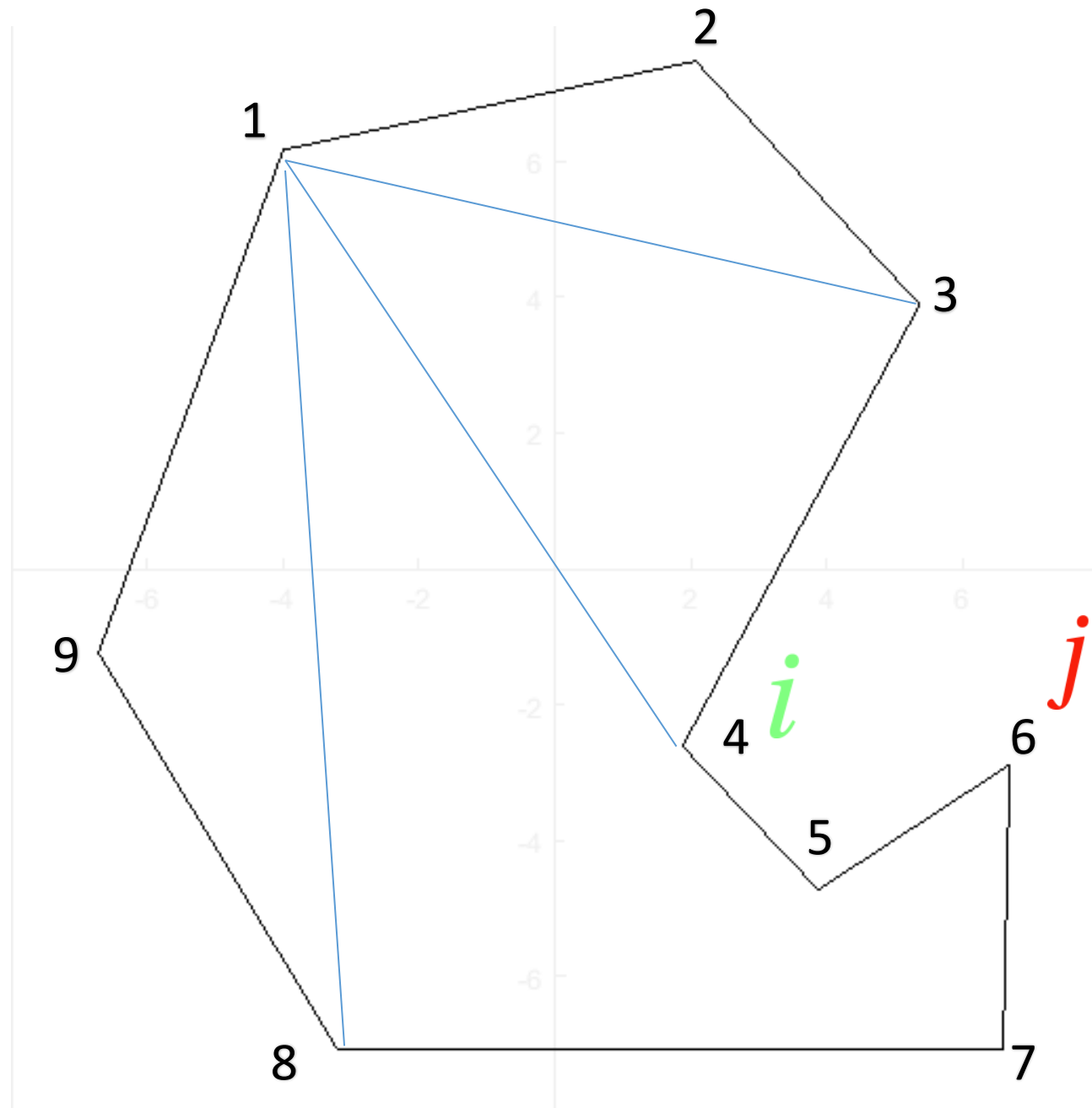
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \end{bmatrix}$$



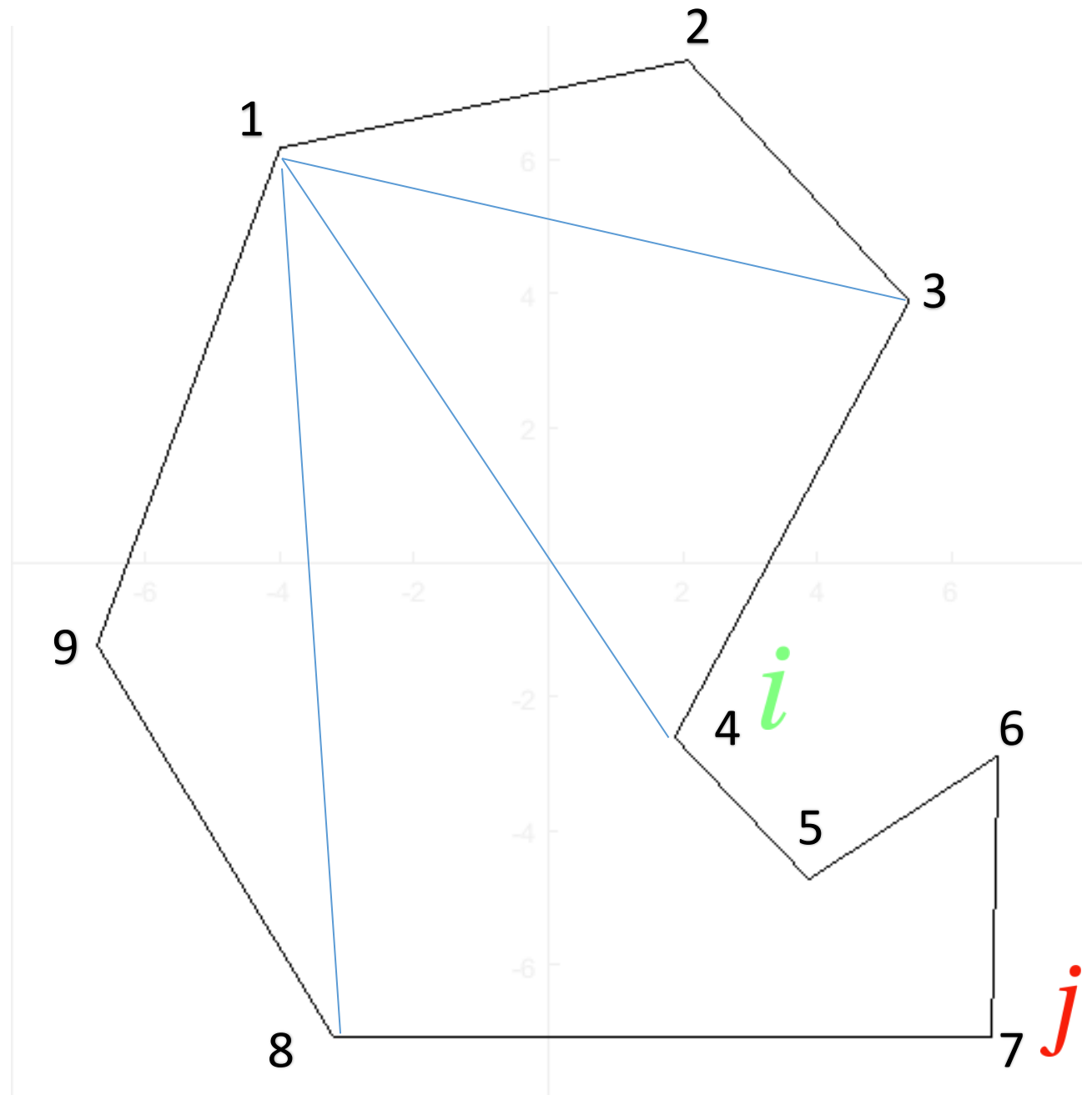
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \end{bmatrix}$$



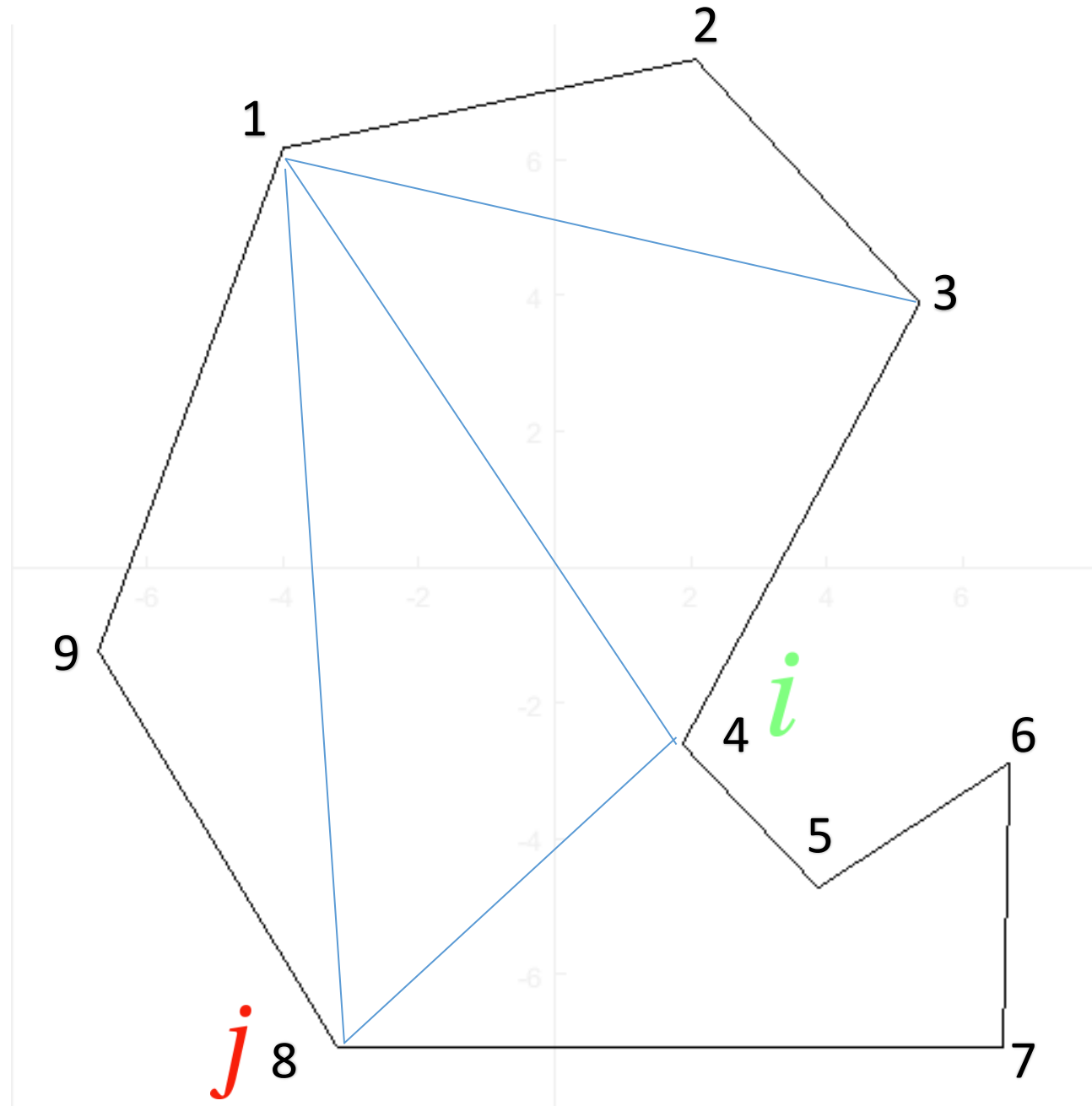
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \end{bmatrix}$$



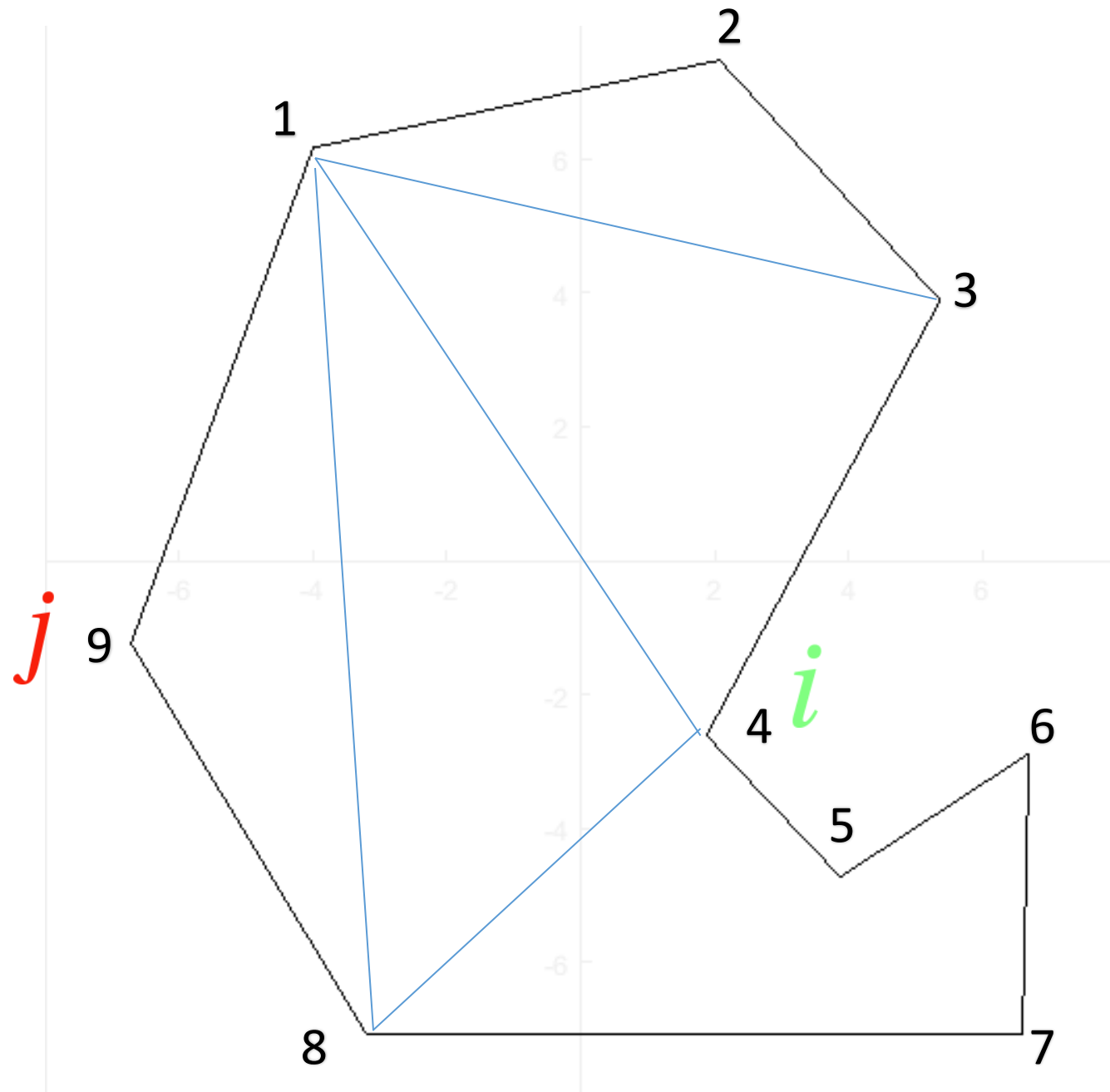
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \end{bmatrix}$$



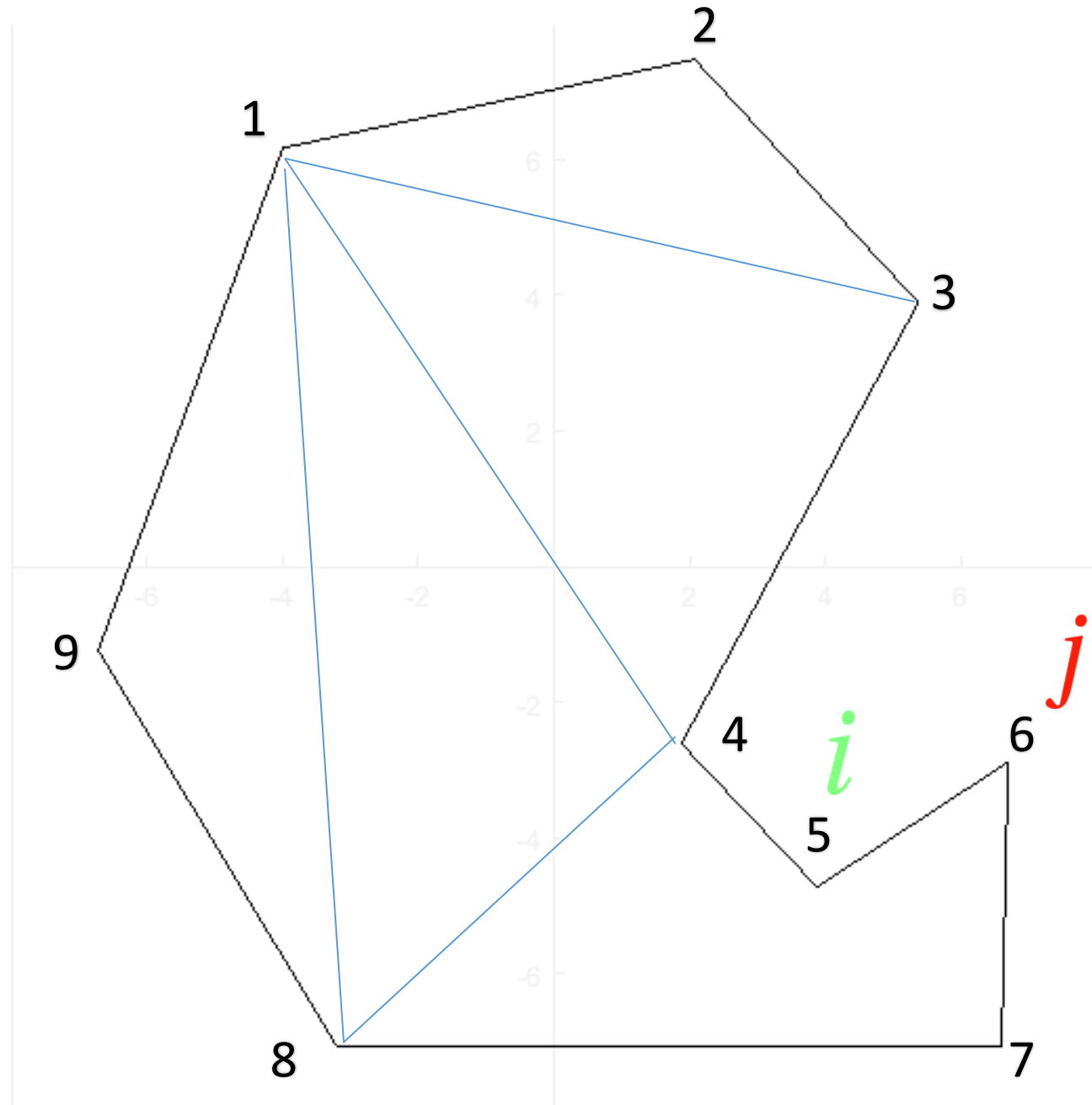
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \end{bmatrix}$$



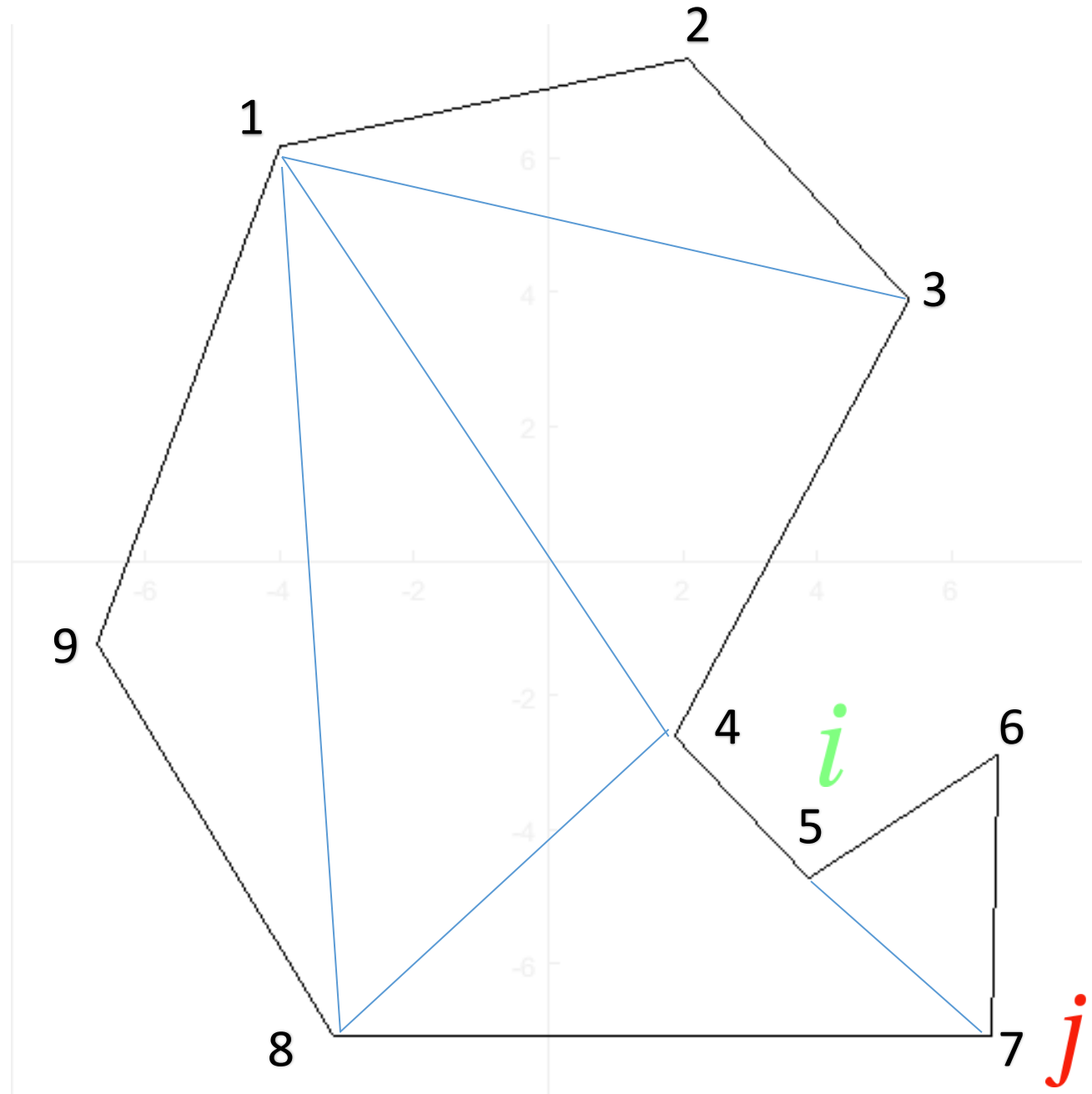
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \end{bmatrix}$$



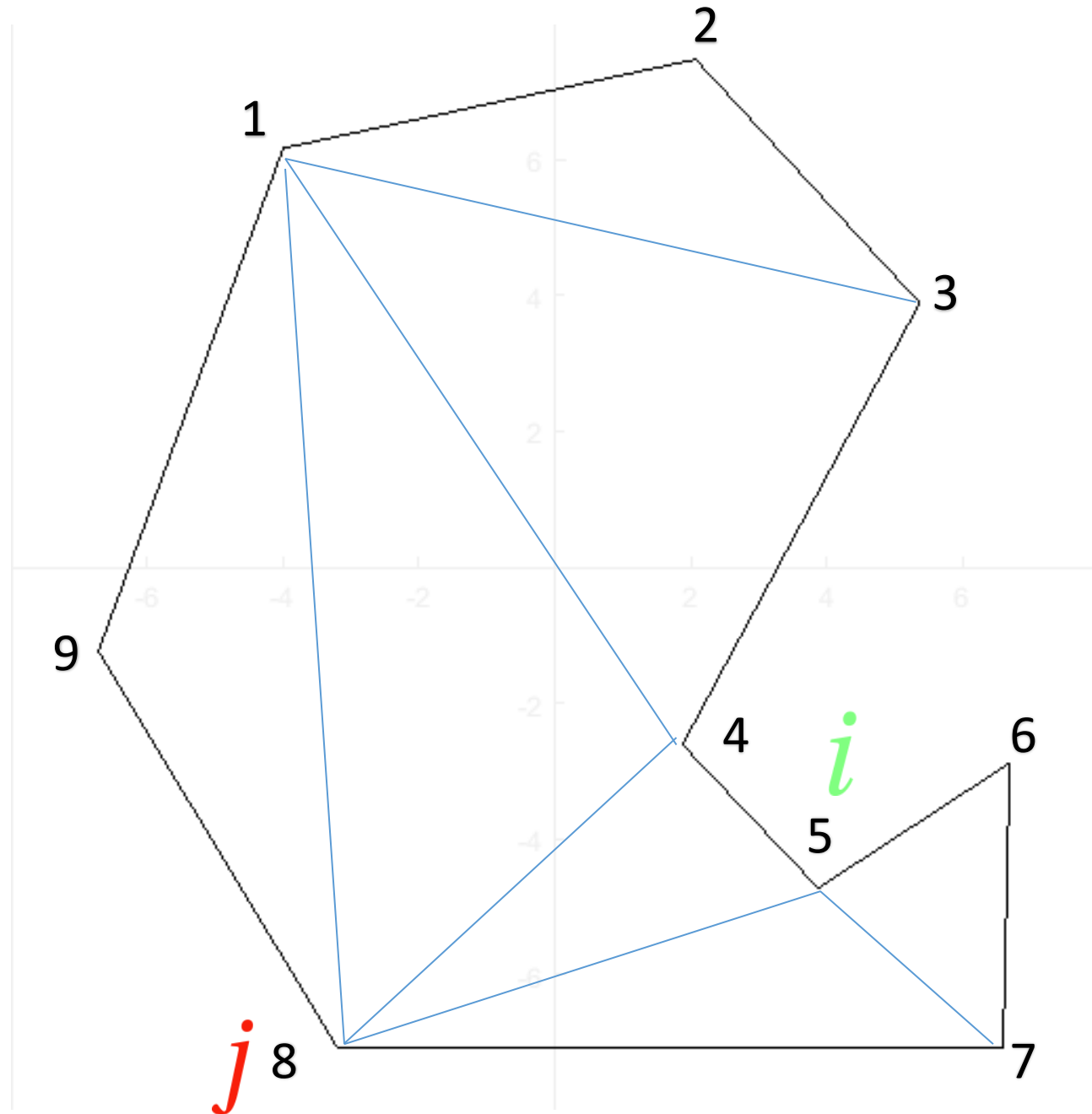
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \\ 5 & 7 \end{bmatrix}$$



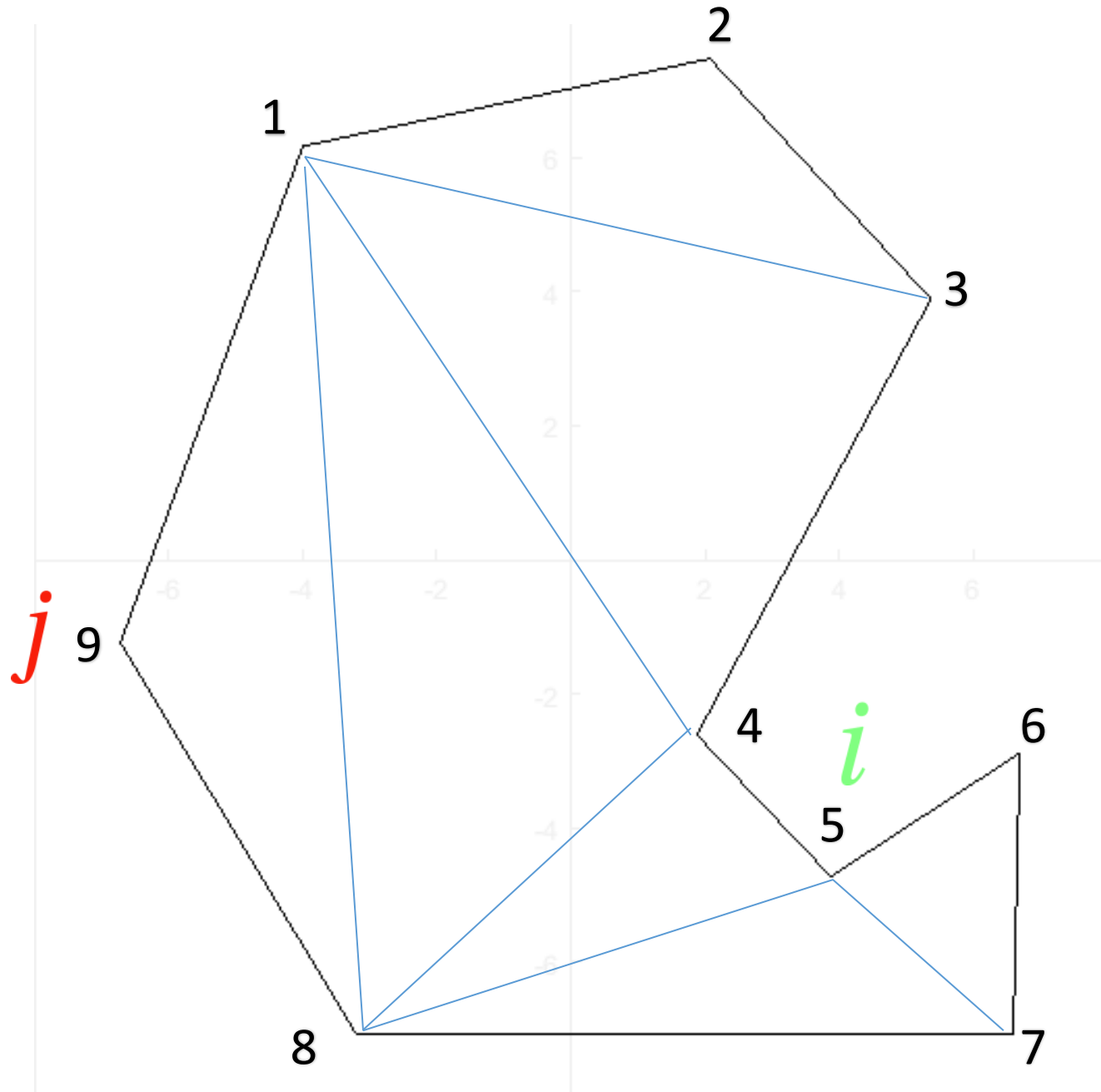
Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \\ 5 & 7 \\ 5 & 8 \end{bmatrix}$$



Making New Lines

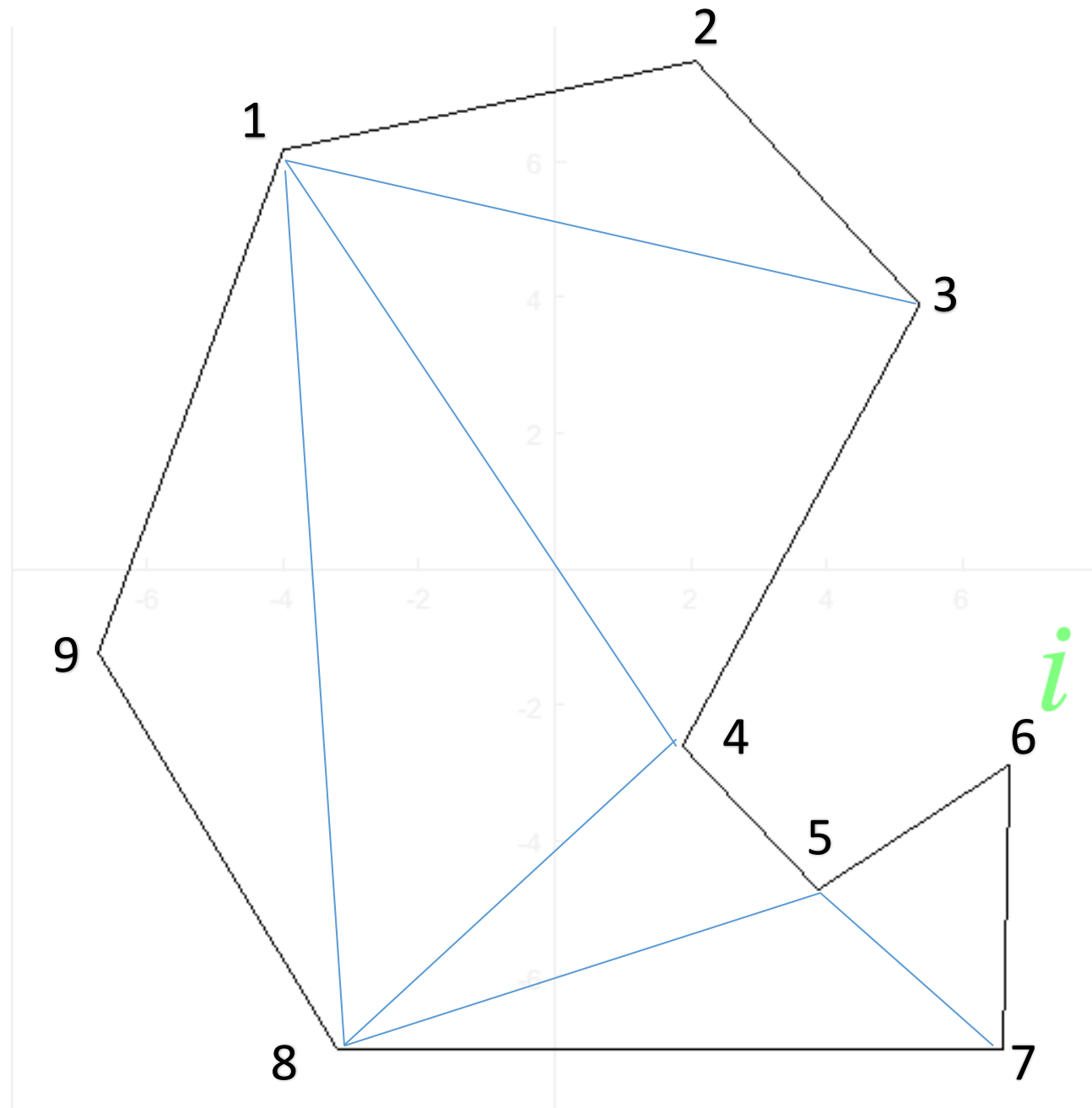
$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \\ 5 & 7 \\ 5 & 8 \end{bmatrix}$$



Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \\ 5 & 7 \\ 5 & 8 \end{bmatrix}$$

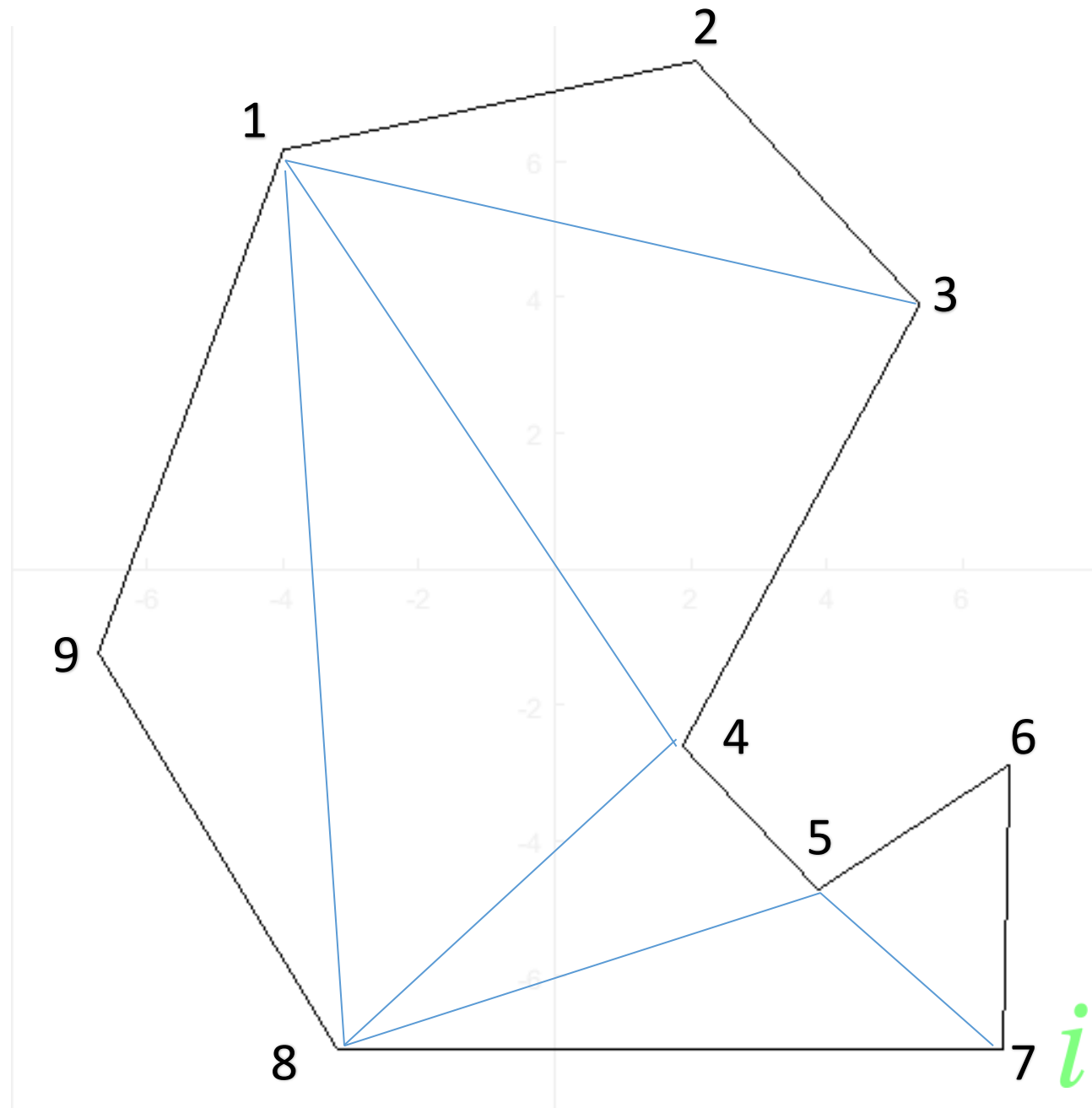
No *j* works



Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \\ 5 & 7 \\ 5 & 8 \end{bmatrix}$$

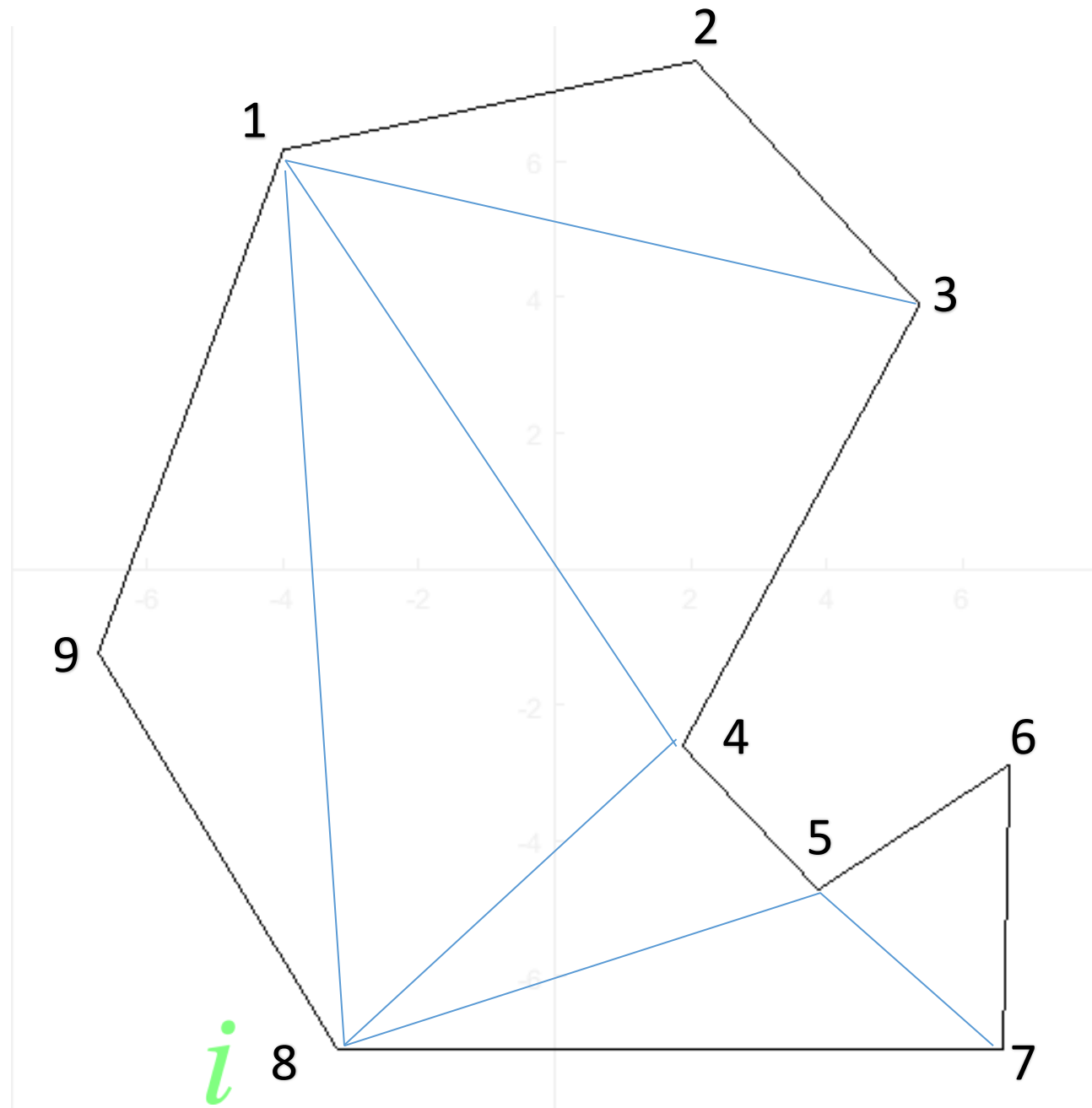
No *j* works



Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \\ 5 & 7 \\ 5 & 8 \end{bmatrix}$$

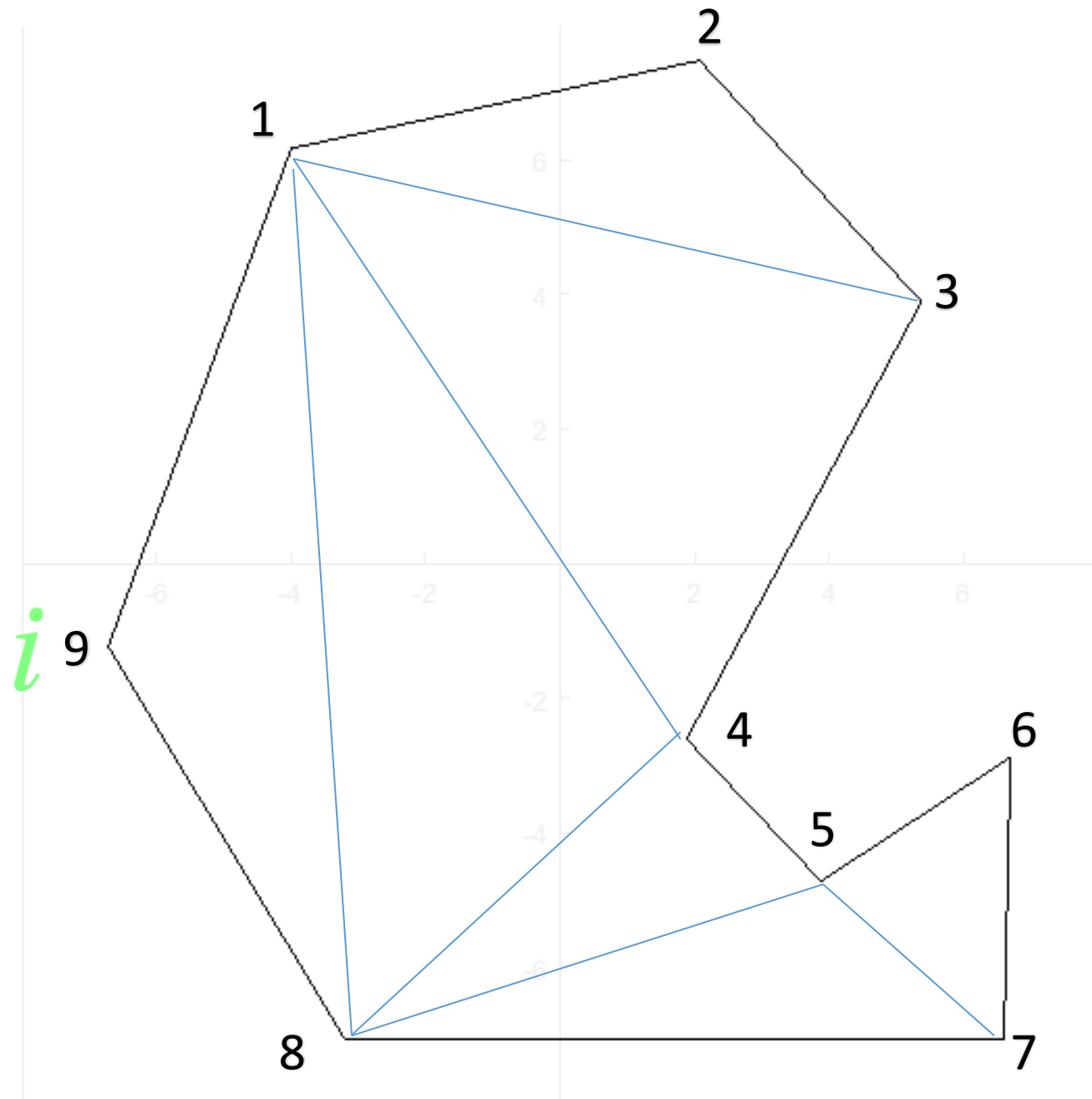
No *j* works

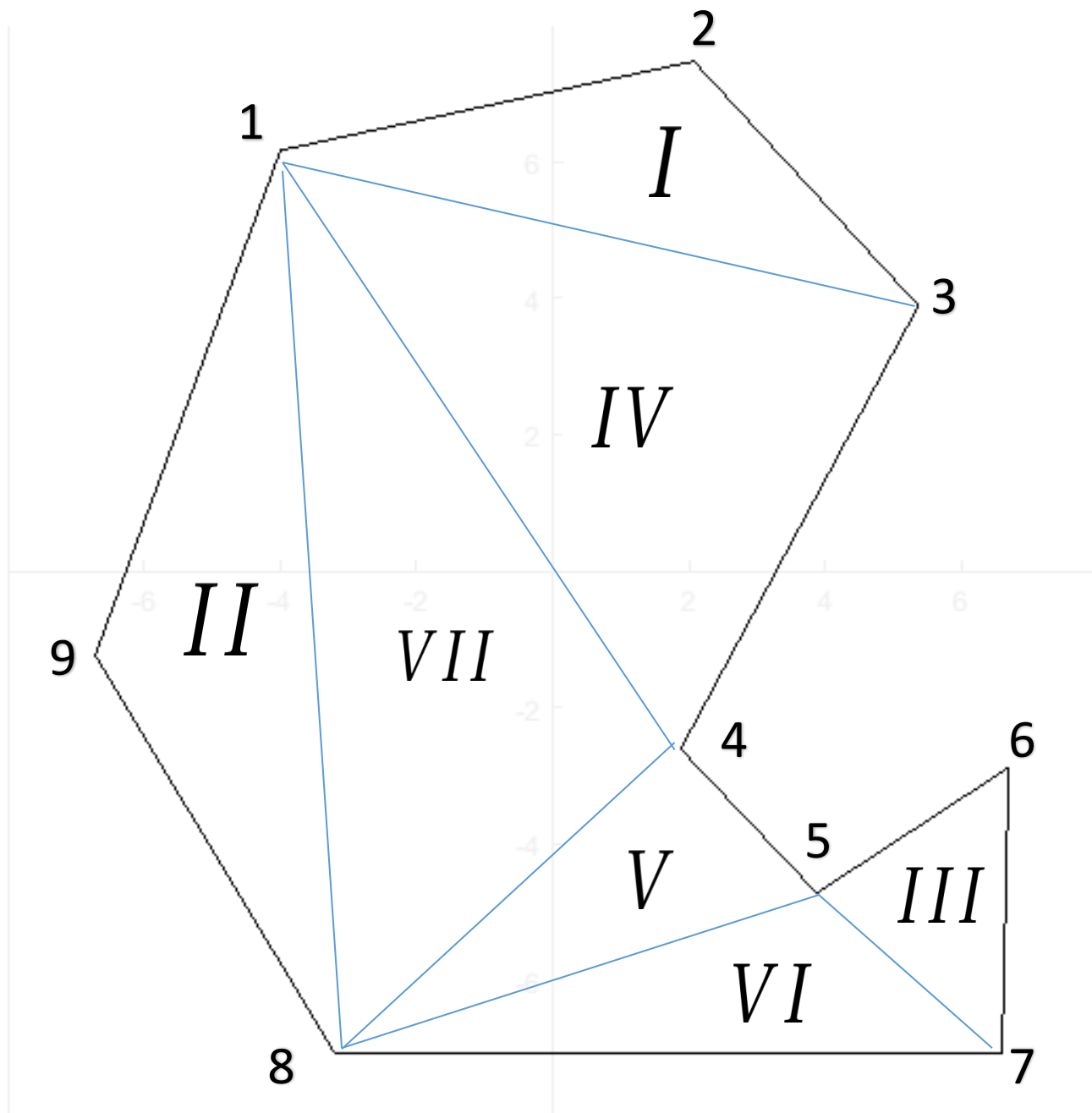


Making New Lines

$$nLine = \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 1 & 8 \\ 4 & 8 \\ 5 & 7 \\ 5 & 8 \end{bmatrix}$$

No *j* works

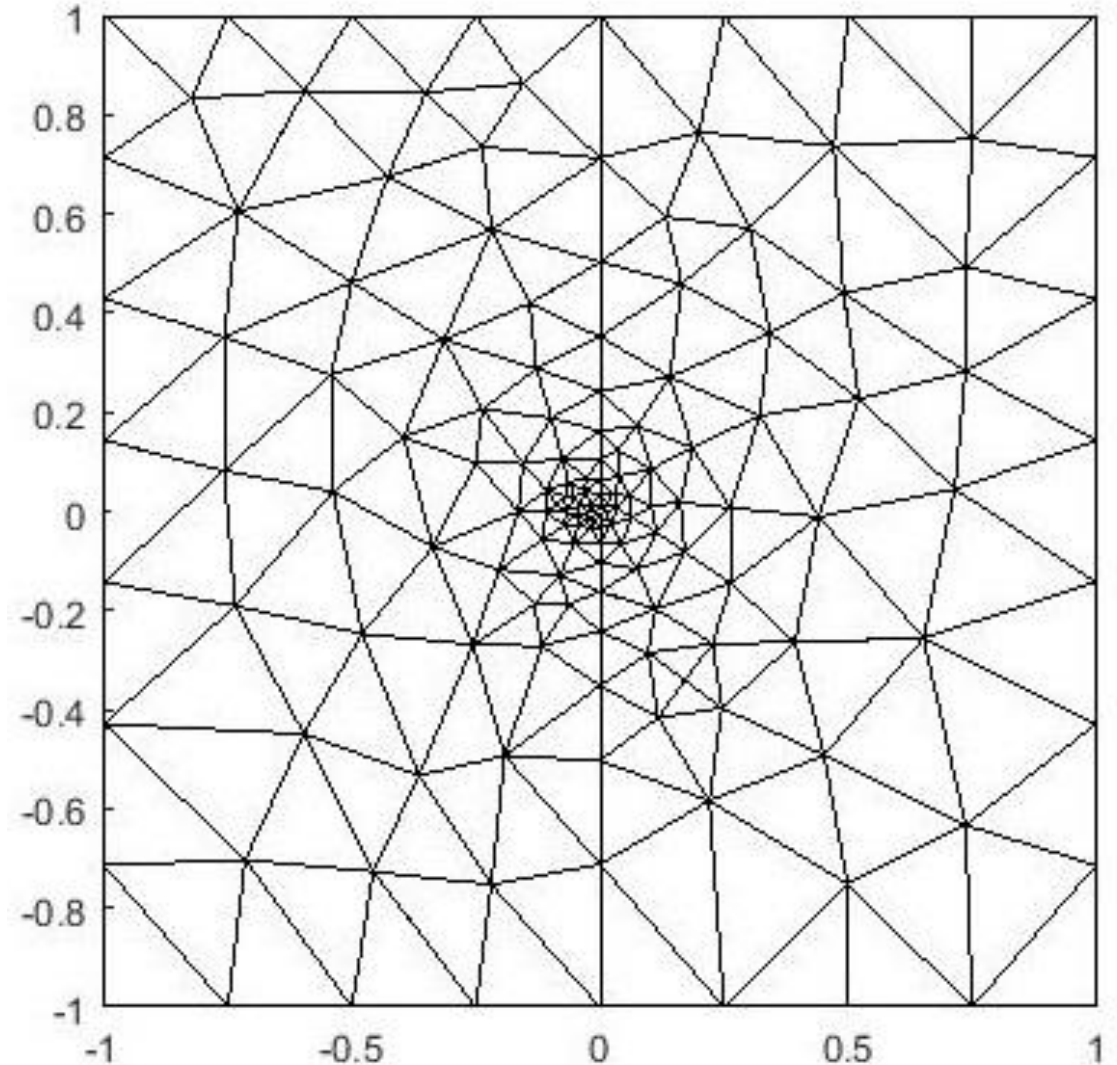




$$T = \begin{bmatrix} 1 & 3 & 2 \\ 1 & 8 & 9 \\ 5 & 7 & 6 \\ 3 & 4 & 1 \\ 4 & 5 & 8 \\ 7 & 8 & 5 \\ 1 & 4 & 8 \end{bmatrix} \begin{matrix} I \\ II \\ III \\ IV \\ V \\ VI \\ VII \end{matrix}$$

Adding Triangles

- There are many ways to add triangles to our mesh, however many are undesired
- For instance we could bisect the top left triangle over and over again until we reach the number requested
- Or we could proceed as featured on the right, (much better than above but still not ideal)

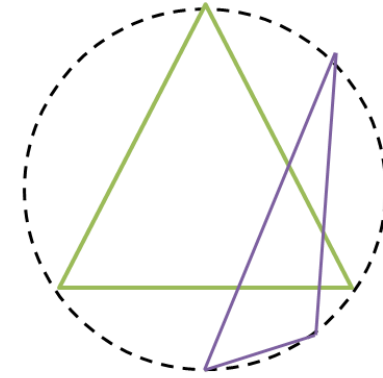


Quality

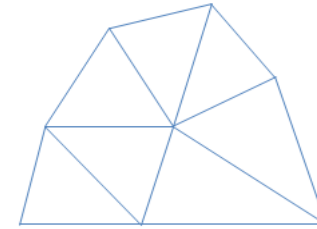
- When utilizing the finite element method in two dimensions, with infinite computing resources one would always choose to represent their domain with an infinite amount of points
- However in reality we solve on a domain using a finite amount of points, and these points will interpolate the area of the domain which does not have a point

Quality

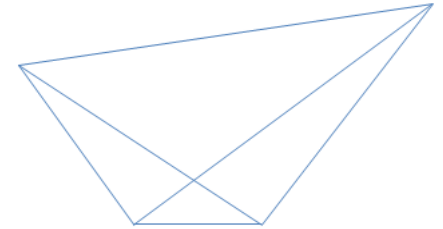
- Therefore when utilizing a triangular mesh to cover such a region we want as little skewness as possible
- We also want smooth changes in cell size
- And finally the ratio of the longest to shortest side in a cell should be as close to 1 as possible



Skewness



Smooth Change in cell size



Large jump in cell size



Aspect ratio = 1



High aspect ratio triangle



Aspect ratio = 1



High aspect ratio quad

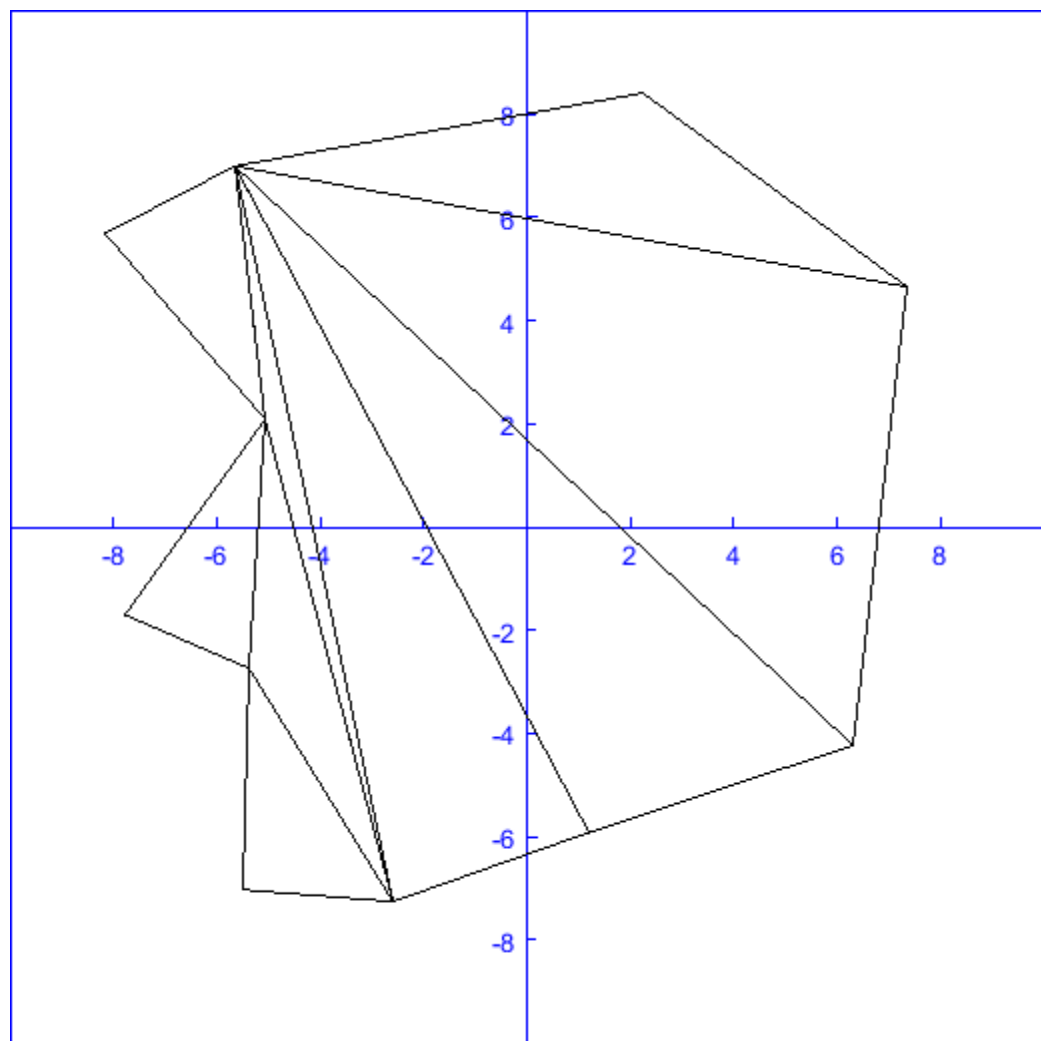
Goal: Equilateral Triangles

- Therefore the quality of each cell will be measured in its likeness to an equilateral triangle
- This can be done with the ratio:
$$\frac{\text{Area}}{\text{Sum of the Sides Squared}}$$
- Because out of all triangles an equilateral will maximize this
- So if we set the quality of each triangle to be this we now have an ordering that we can use to prioritize bisecting
- But it turns out we can do a little better than this

Avoiding Telescoping

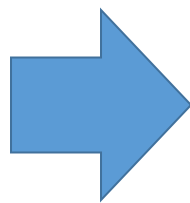
- If we strictly rank and bisect the triangles based on this measure we could end up bisecting one worst quality triangle over and over, getting smaller and smaller in one portion of the polygon
- To avoid this we factor in the size of the triangles to prioritize them higher based on a larger area as well
- The result of the two approaches combined gives us a ranking of triangles to bisect based on: $\frac{1}{\text{Area}} \times \frac{\text{Area}}{\text{Sum of the Sides Squared}}$
- Or: $\frac{1}{\text{Sum of the Sides Squared}}$

(T, V)

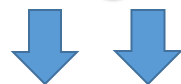


Add

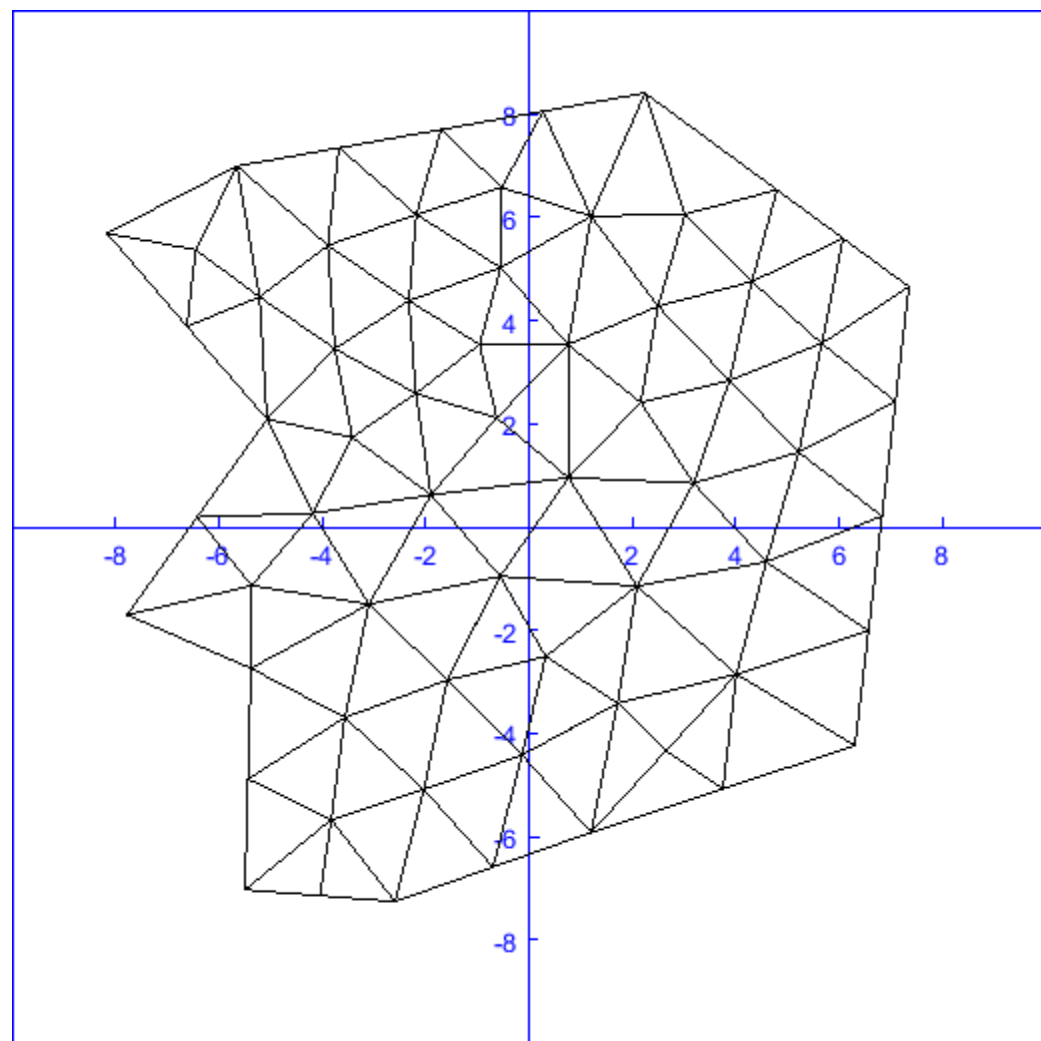
$n = 100$



Larger



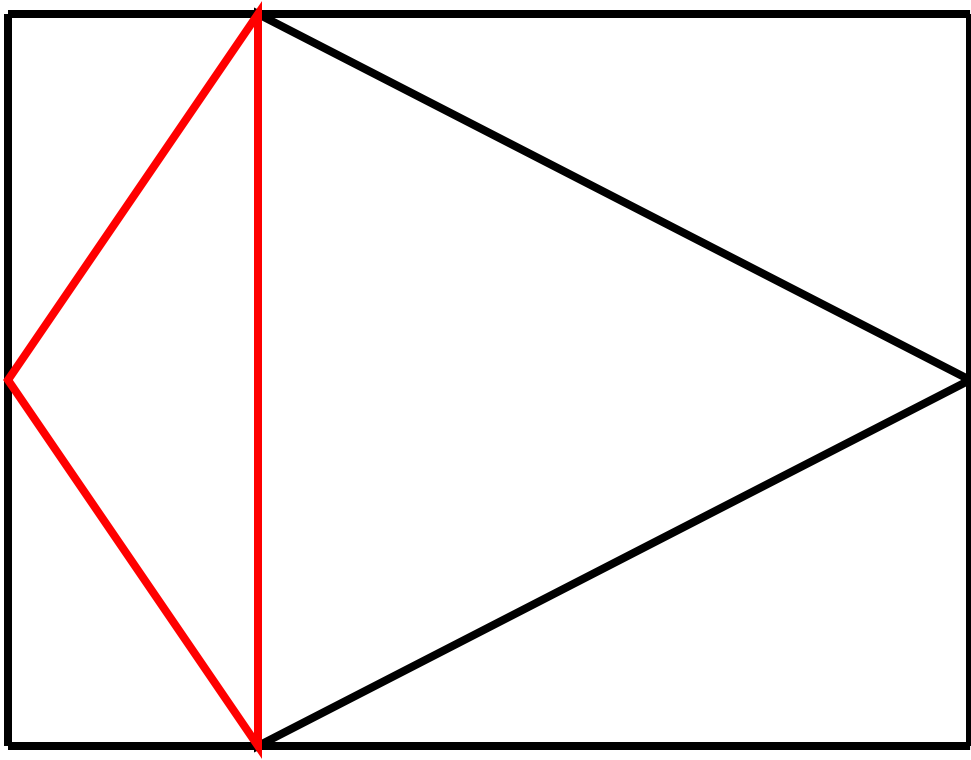
(T, V)



Adding Process Overview

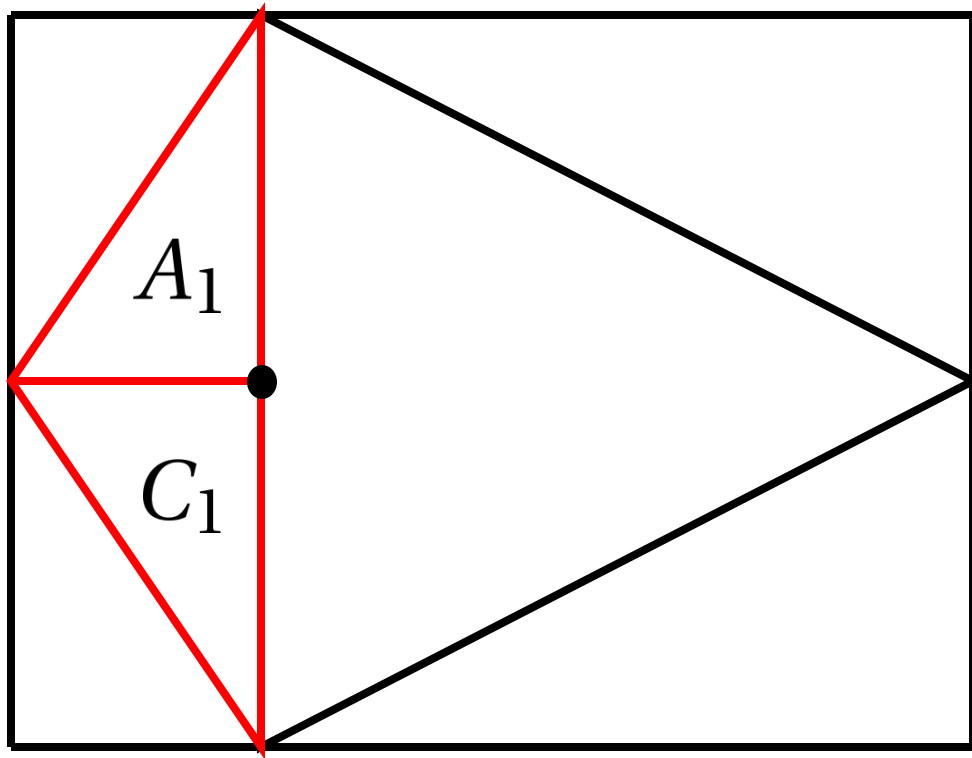
- While $\text{numT} < n$
 - Sort triangles from worst to best quality
 - Start with worst quality and bisect it
 - Check neighbors for eligible flips and bisections
- Remove the triangles that were changed, add new ones, and add new vertices

Bisecting and Flipping



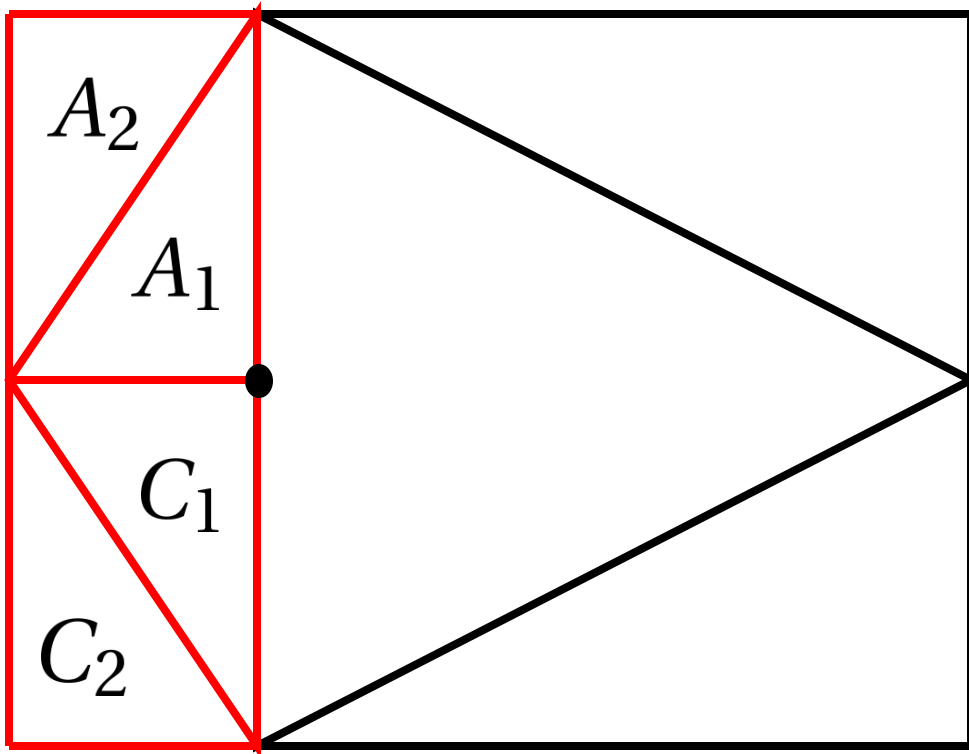
- First we take the triangle with the worst quality (designated in red) and bisect it

Bisecting and Flipping

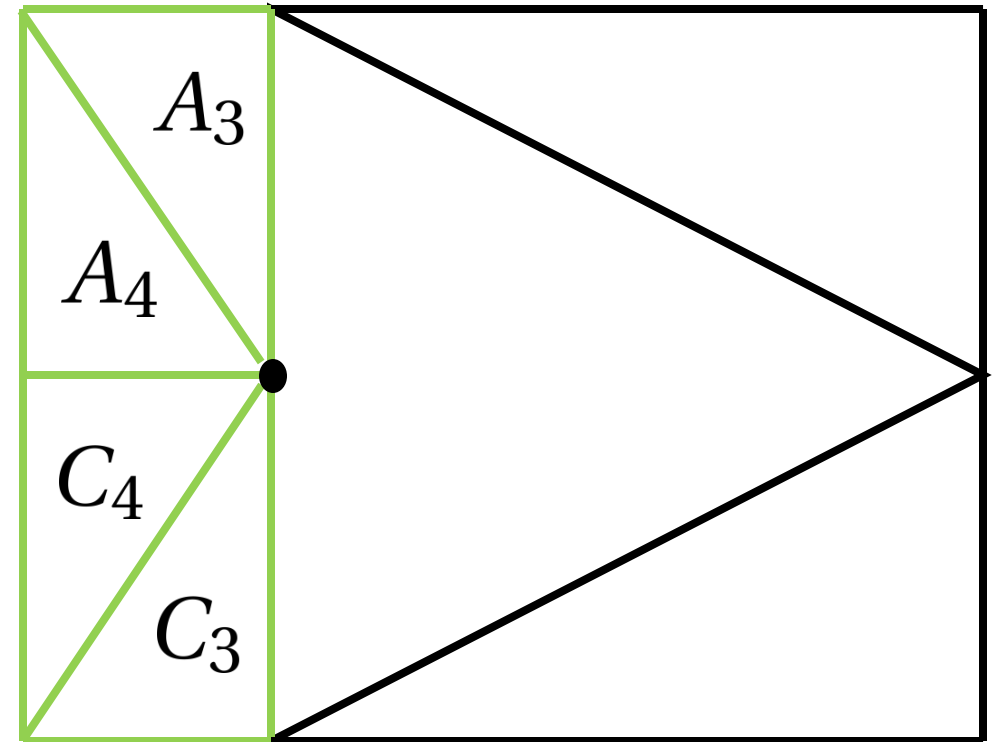


- So now we have two new triangles A_1 and C_1 .
- But is this the best we can do?

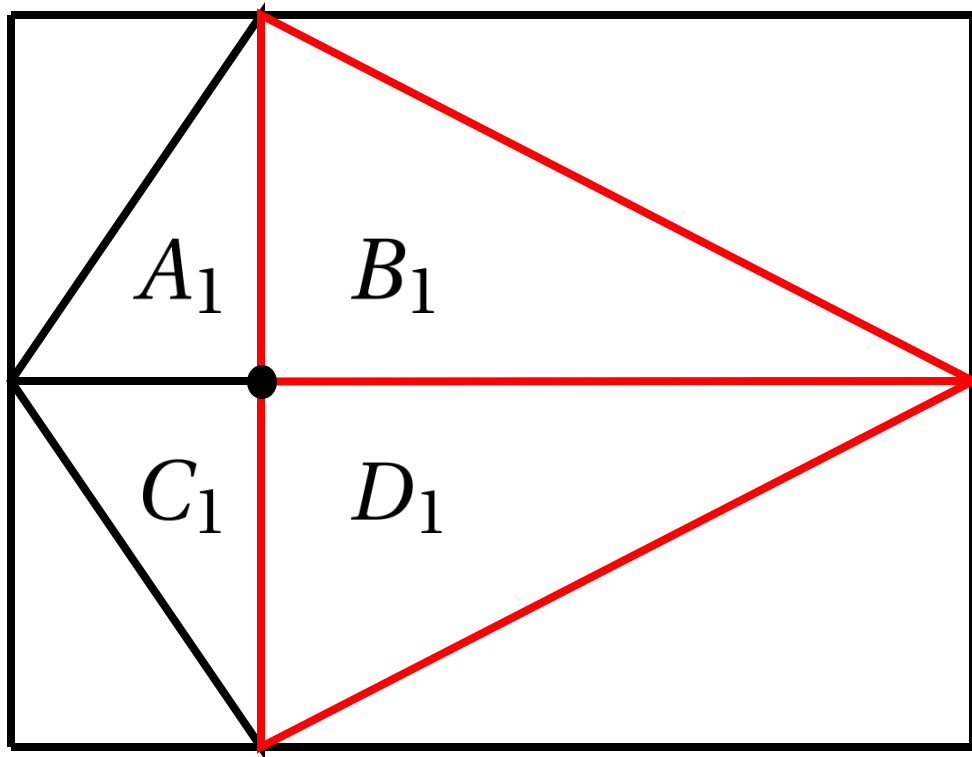
- Using our quality function we will compare the total quality of $A_1 + A_2$ with $A_3 + A_4$ and the total quality of $C_1 + C_2$ with $C_3 + C_4$.
- And then select the best out of these to include



vs

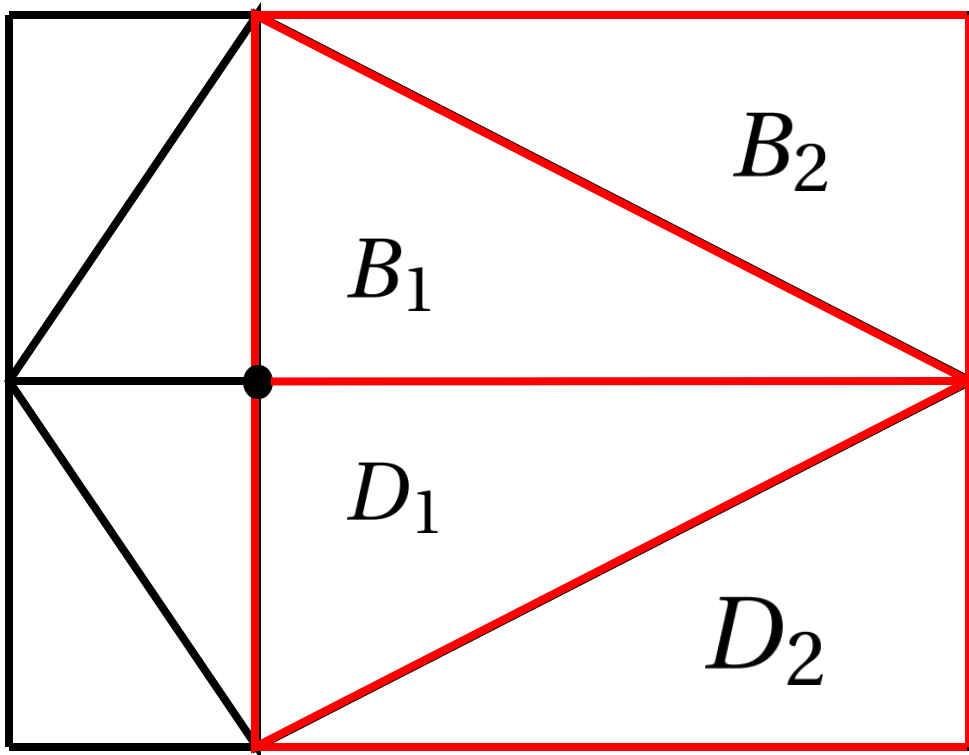


Bisecting Adjacent Triangle

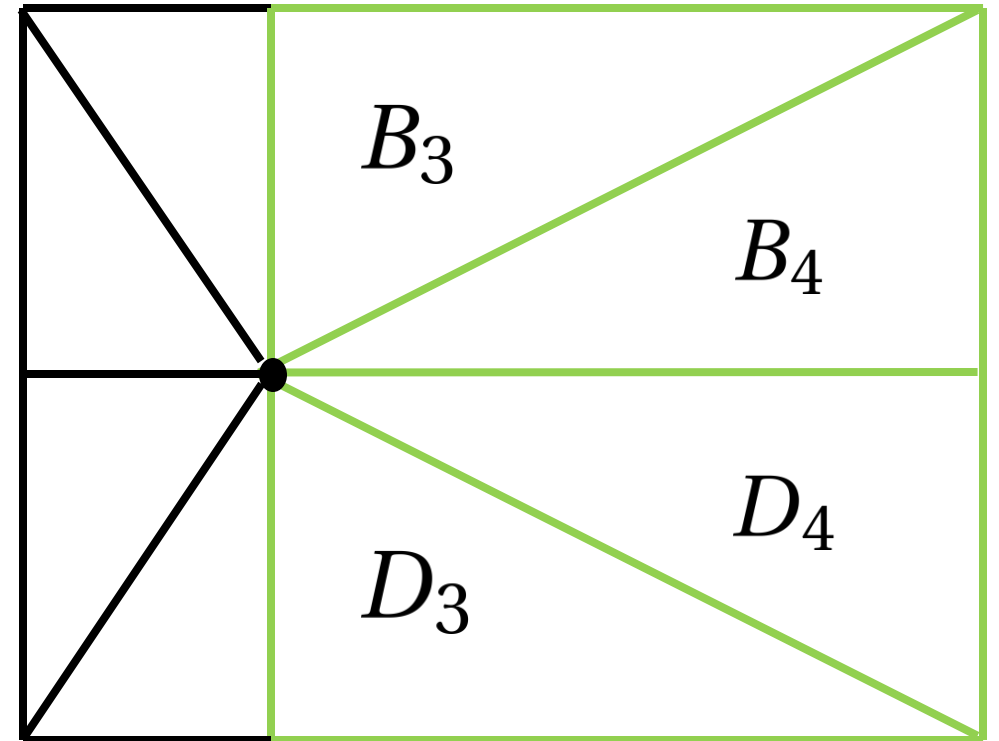


- If there is a neighboring triangle to the one with the worst quality we will consider bisecting it as well
- Giving us the new triangles B_1 and D_1 .

- Similarly our quality function will compare the total quality of $B_1 + B_2$ with $B_3 + B_4$ and the total quality of $D_1 + D_2$ with $D_3 + D_4$.
- And then select the best out of these to include



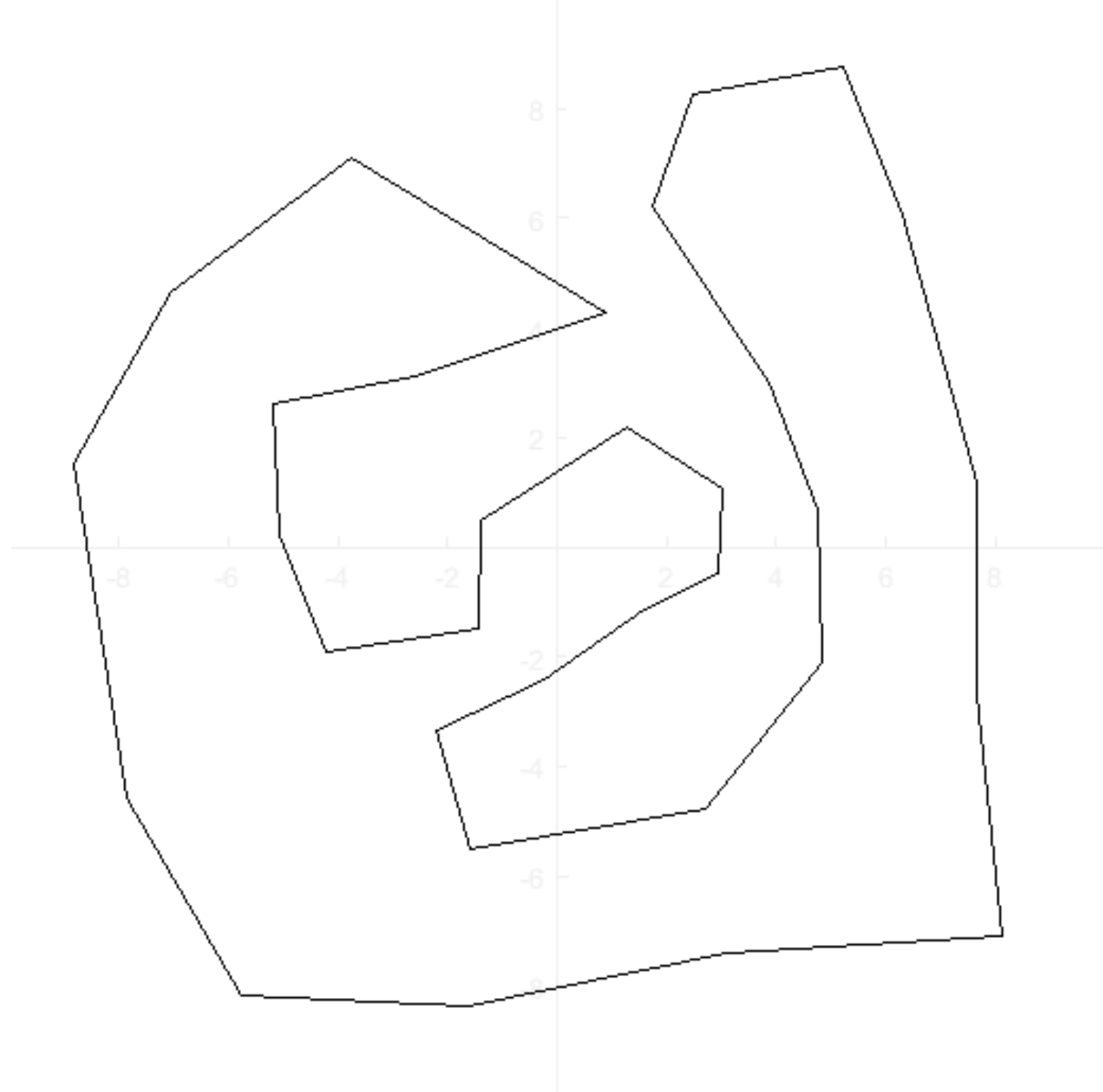
VS



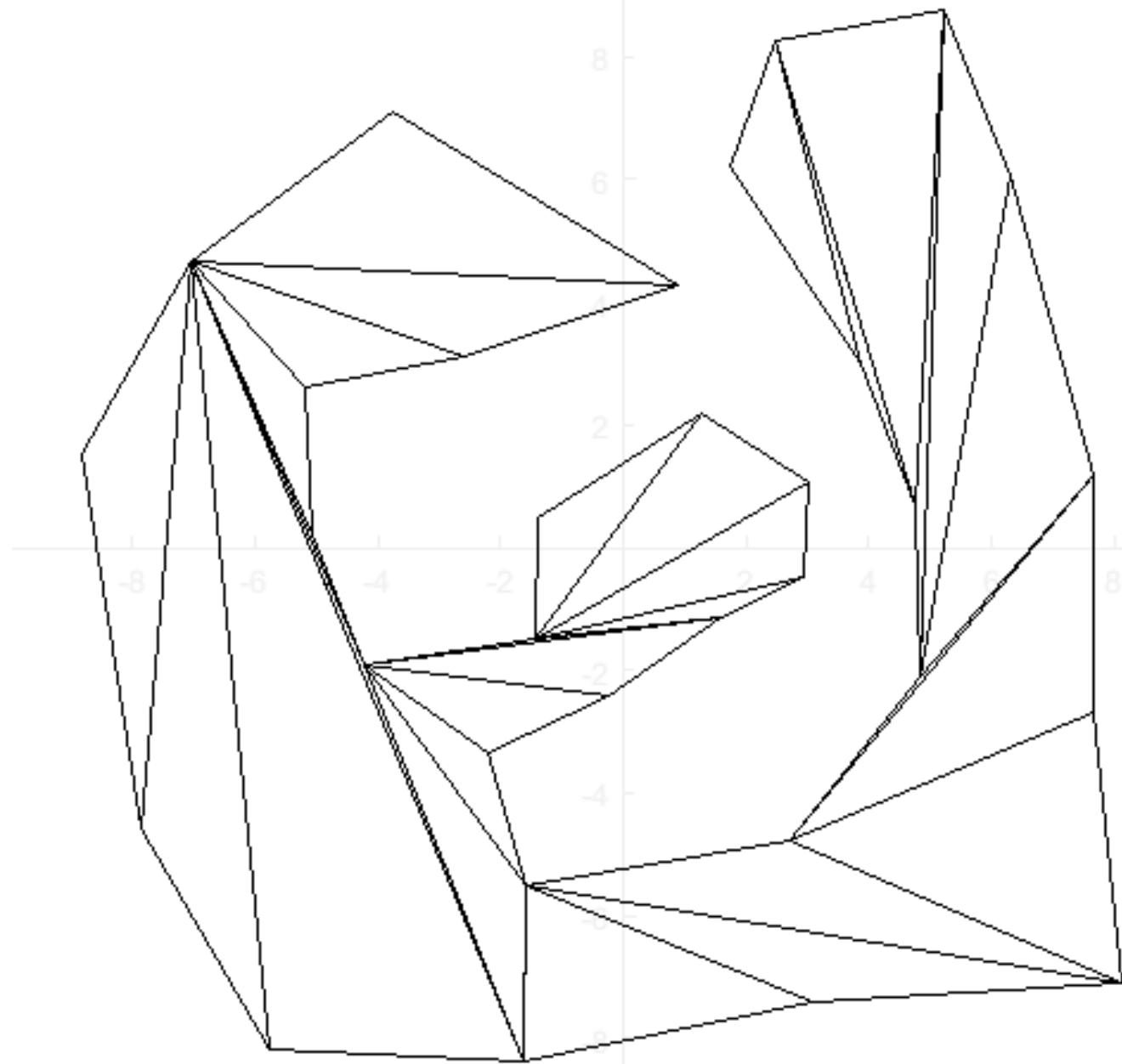
Bisecting and Flipping

- Finally with the all the triangles chosen with the highest quality we save our new triangles and new vertex
- This process continues until we have at least as many triangles as the user requested

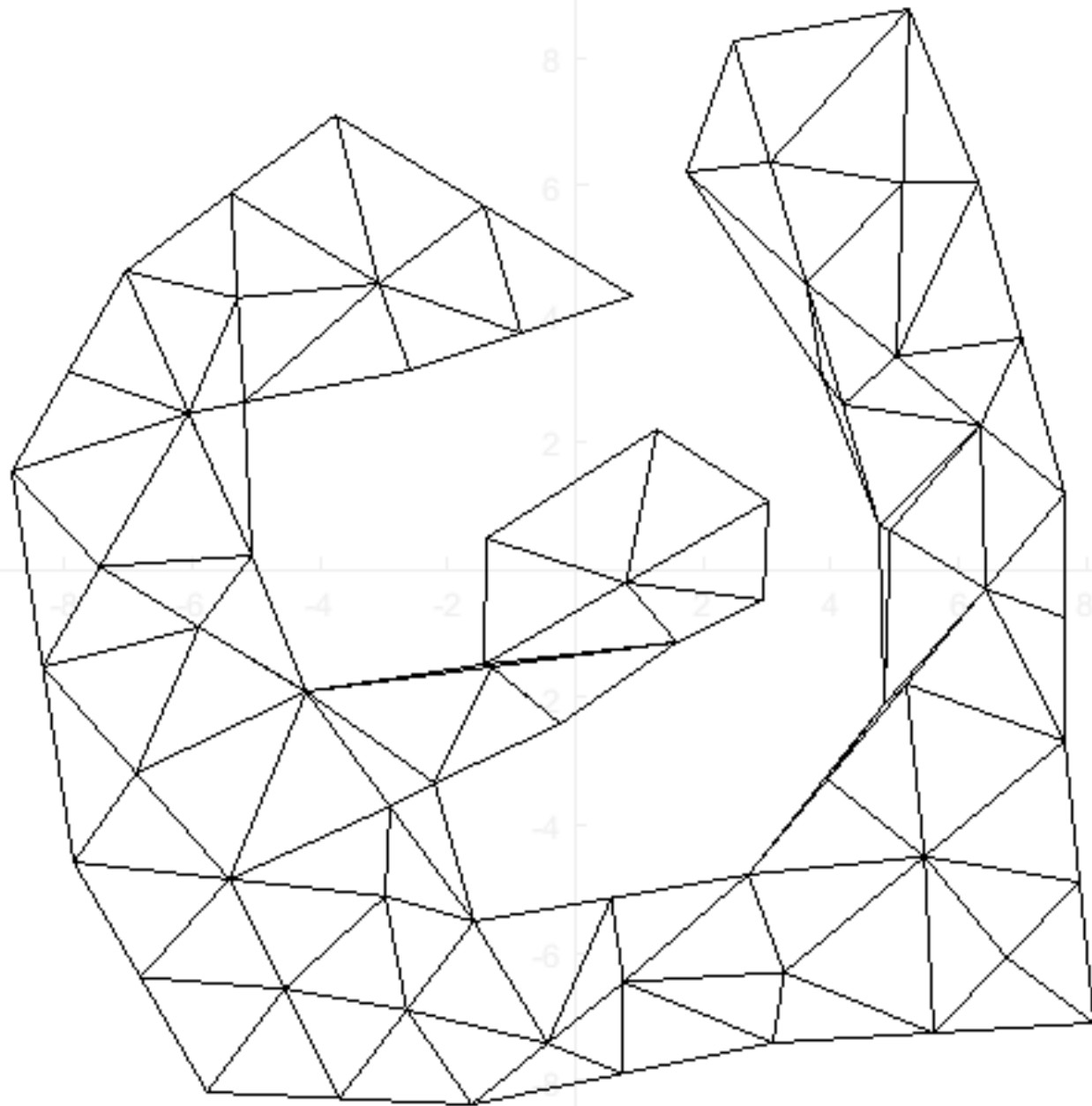
Original



Fan

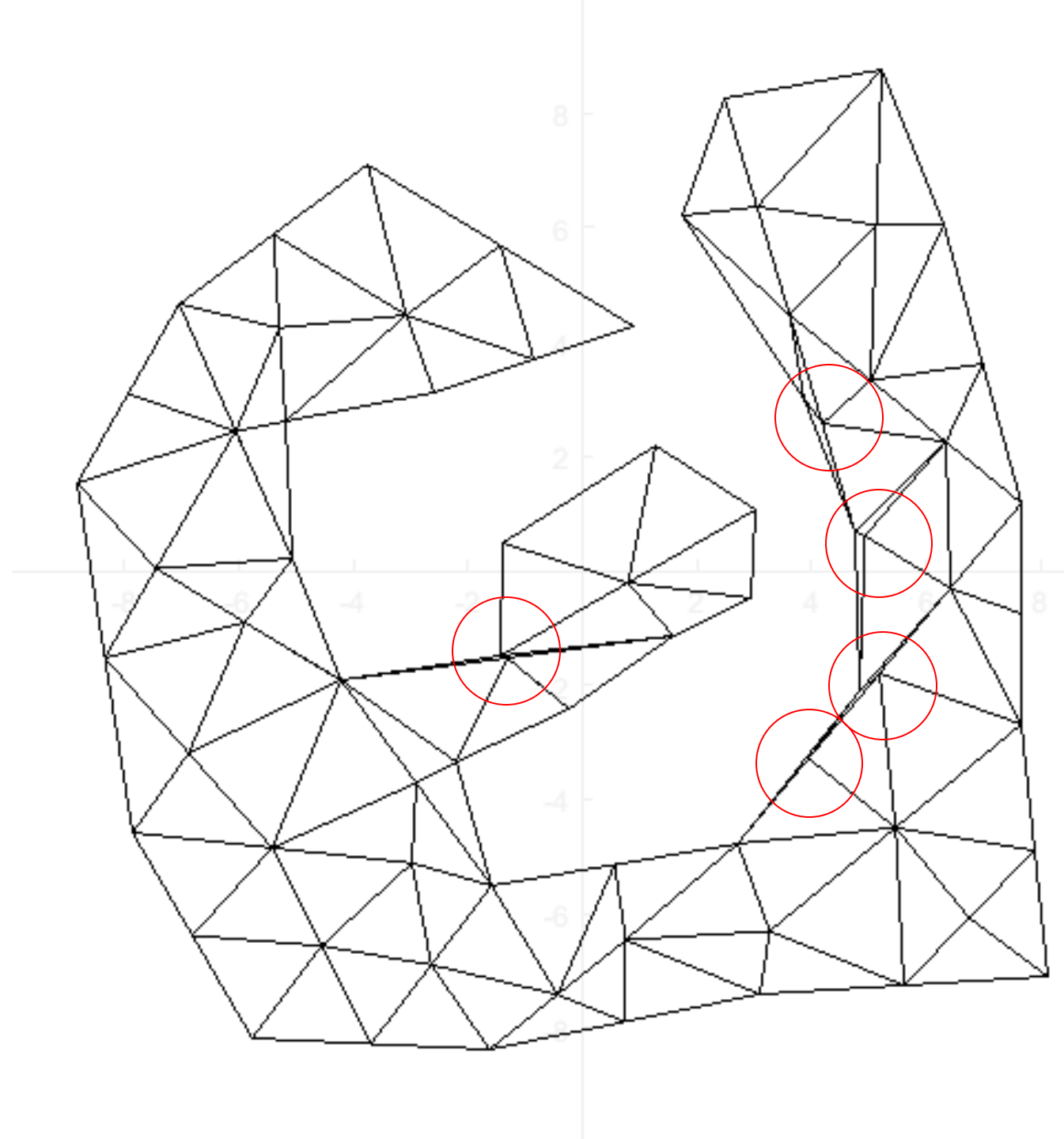


Add Triangles
 $n = 100$

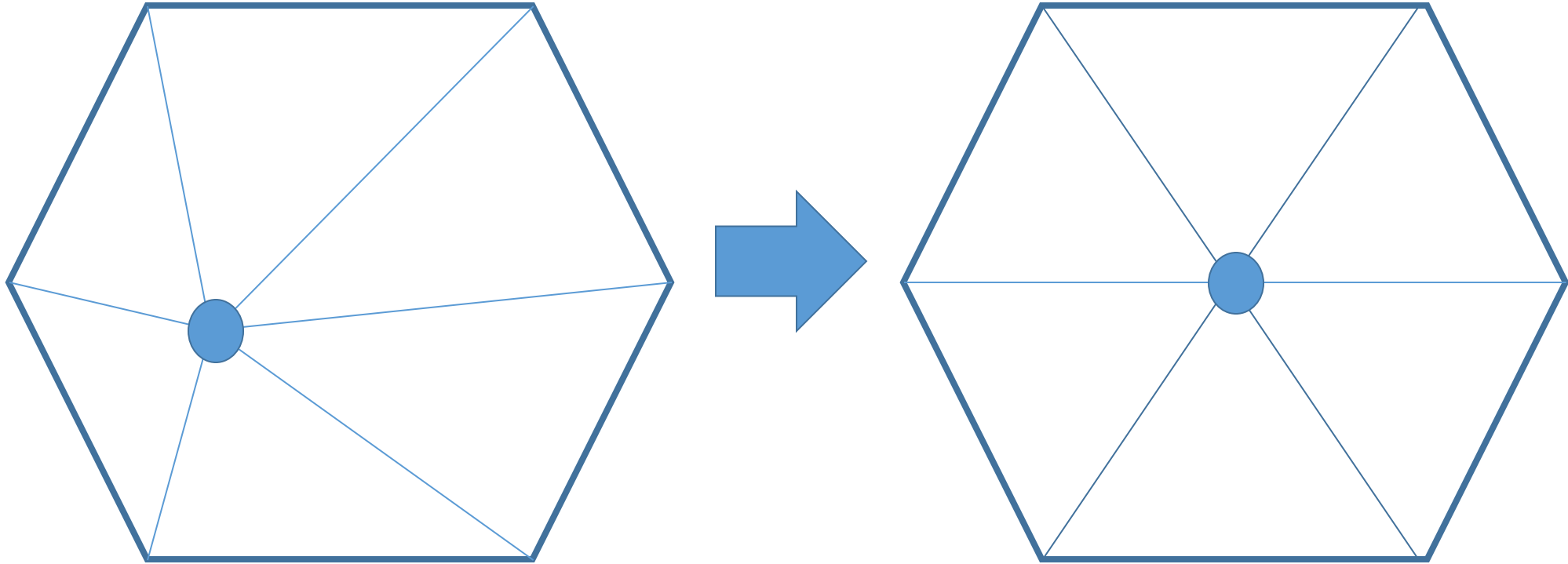


Vertex Shifting

- We still need a way to improve the mesh as we increase the number of triangles
- By shifting interior vertices in a way that improves the overall quality of all the triangles they are connected to we can accomplish this

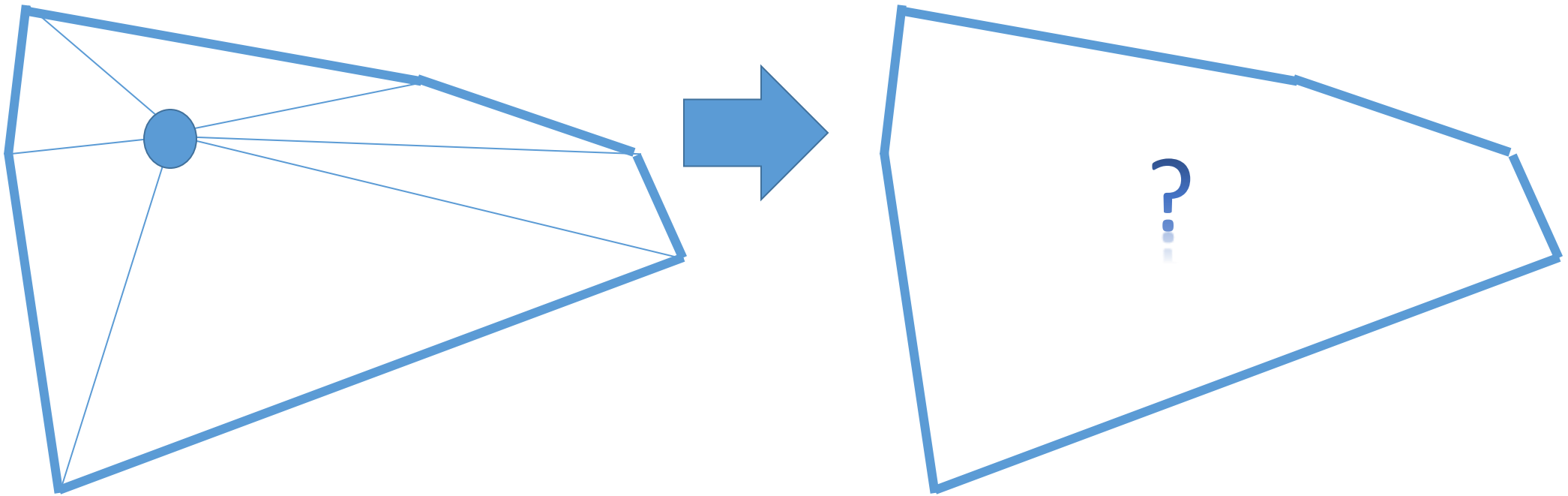


Vertex Shifting



- For this example it is rather obvious where the best point would be, however rarely will any of our interior polygons have all sides of the same length

Vertex Shifting



- For instance where exactly would the perfect point be for this shape?

Vertex Shifting

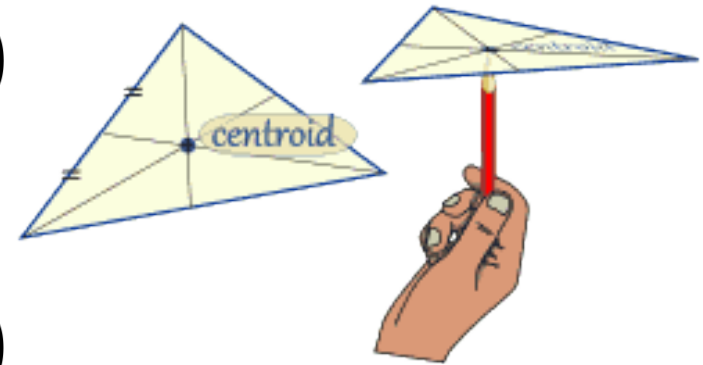
- It turns out finding this exact point is an NP Complete problem
- So for our purposes we will try to come up with a point that improves the triangles even though it may not be the optimal one
- I found that the point which visually resembles that which we would guess as being the perfect point typically turns out to be the Centroid

Centroid of a Polygon

- Same as center of mass as each point has the same weight
- So the Centroid = (C_x, C_y) with

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

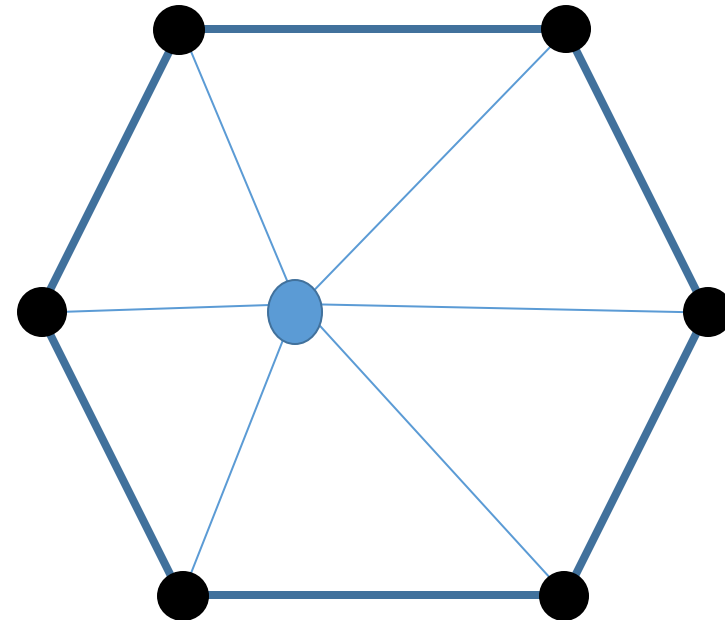
$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$



- Where $A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$ gives the signed area.
- Note: In this formula the vertices are in clockwise order around the perimeter

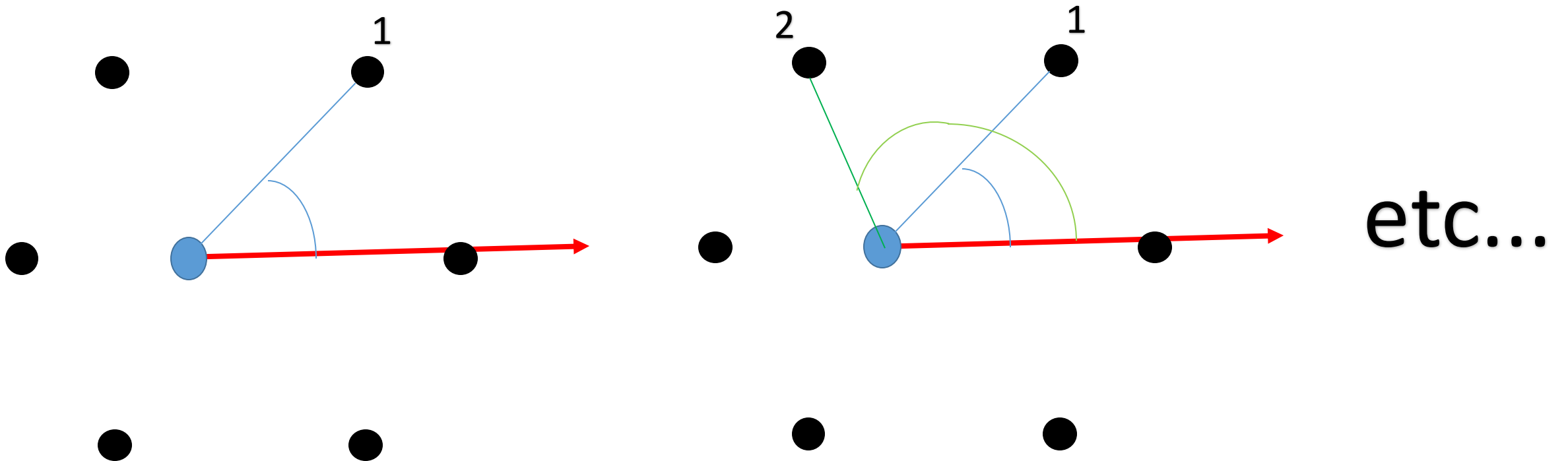
Clockwise to a Computer

- By looping through our V and T matrix we can get each interior vertex and all the triangles it is a part of. This will give us all the vertices that make up the polygon around it however they will not be in order
- Again we arrive at something simple for us but that requires inventive thinking to teach to a computer



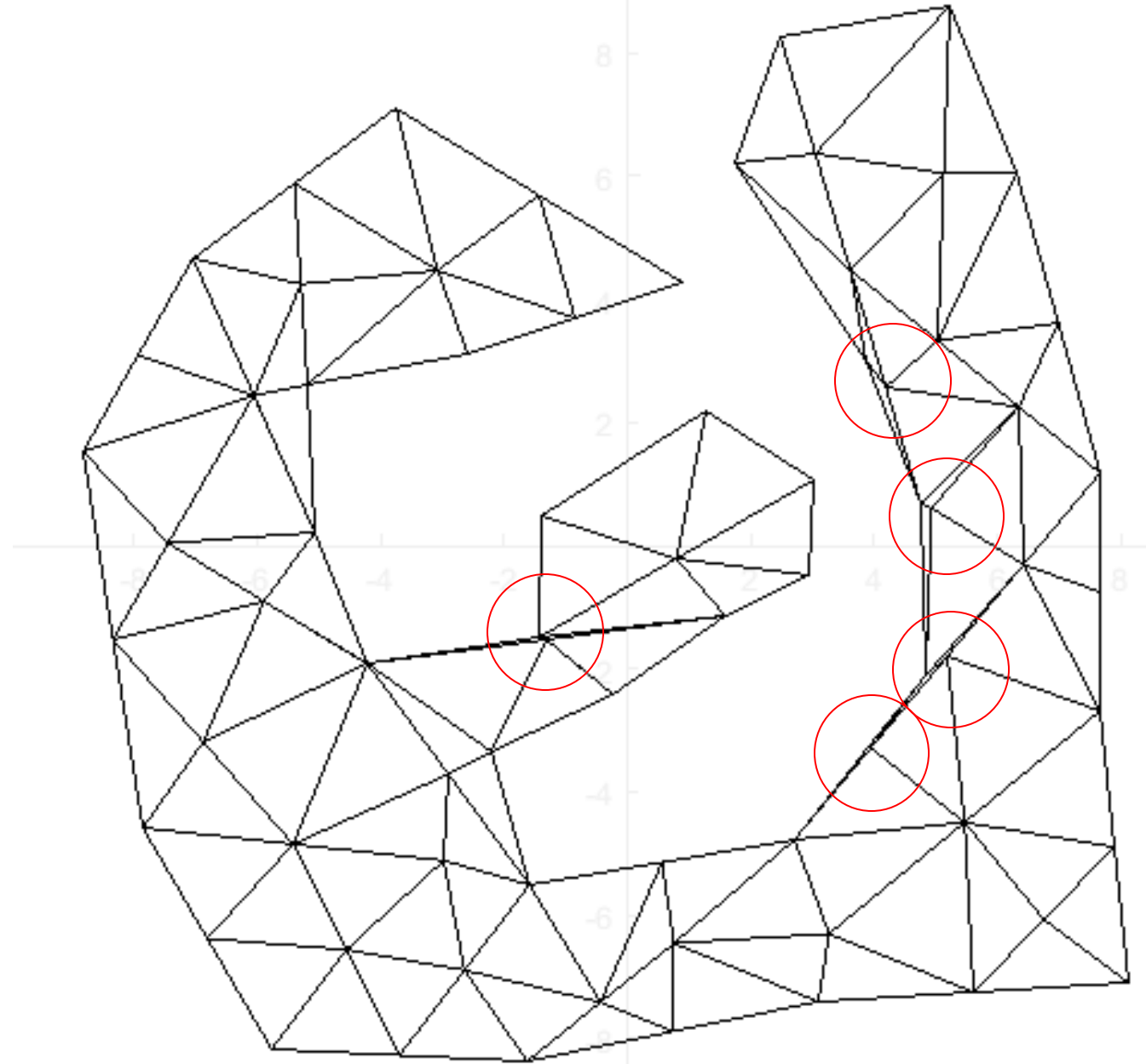
Clockwise to a Computer

- We can organize these by extending a horizontal vector out to the right of our inner vertex
- Then we can use our angle between method from before to order the outer vertices based on the size of their clockwise angles



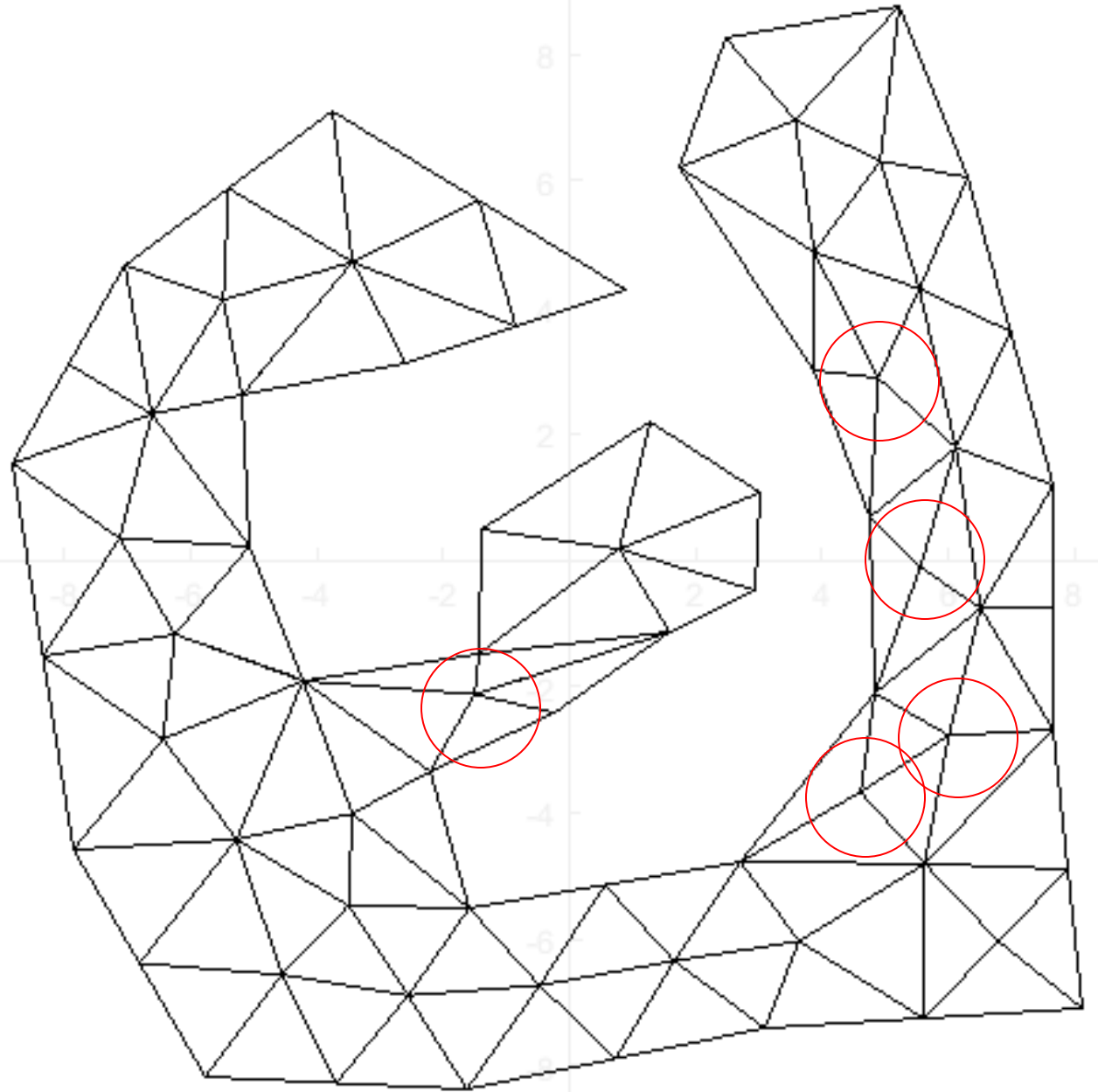
Vertex Shifting

Before



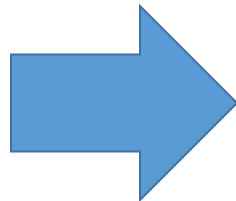
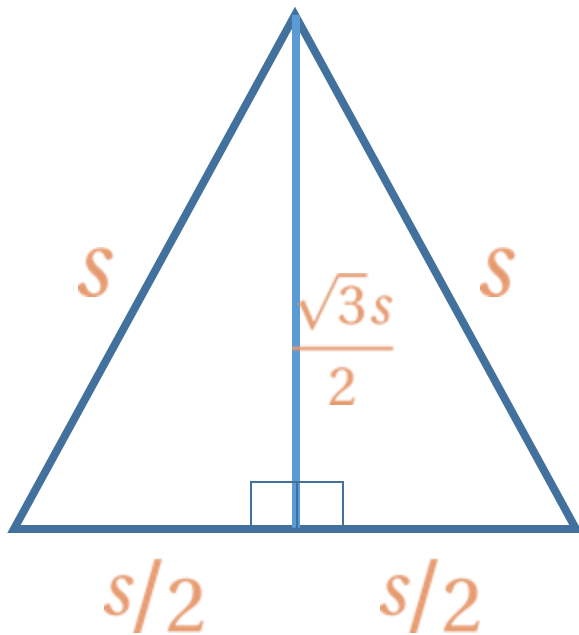
Vertex Shifting

After



Measure our Improvement

- To get a quantitative measure for how well this has improved our mesh we return to how we defined $\text{Quality} = \text{Area} / \text{Sum Sides Squared}$ for each triangle
- For an ideal equilateral triangle this comes out to be



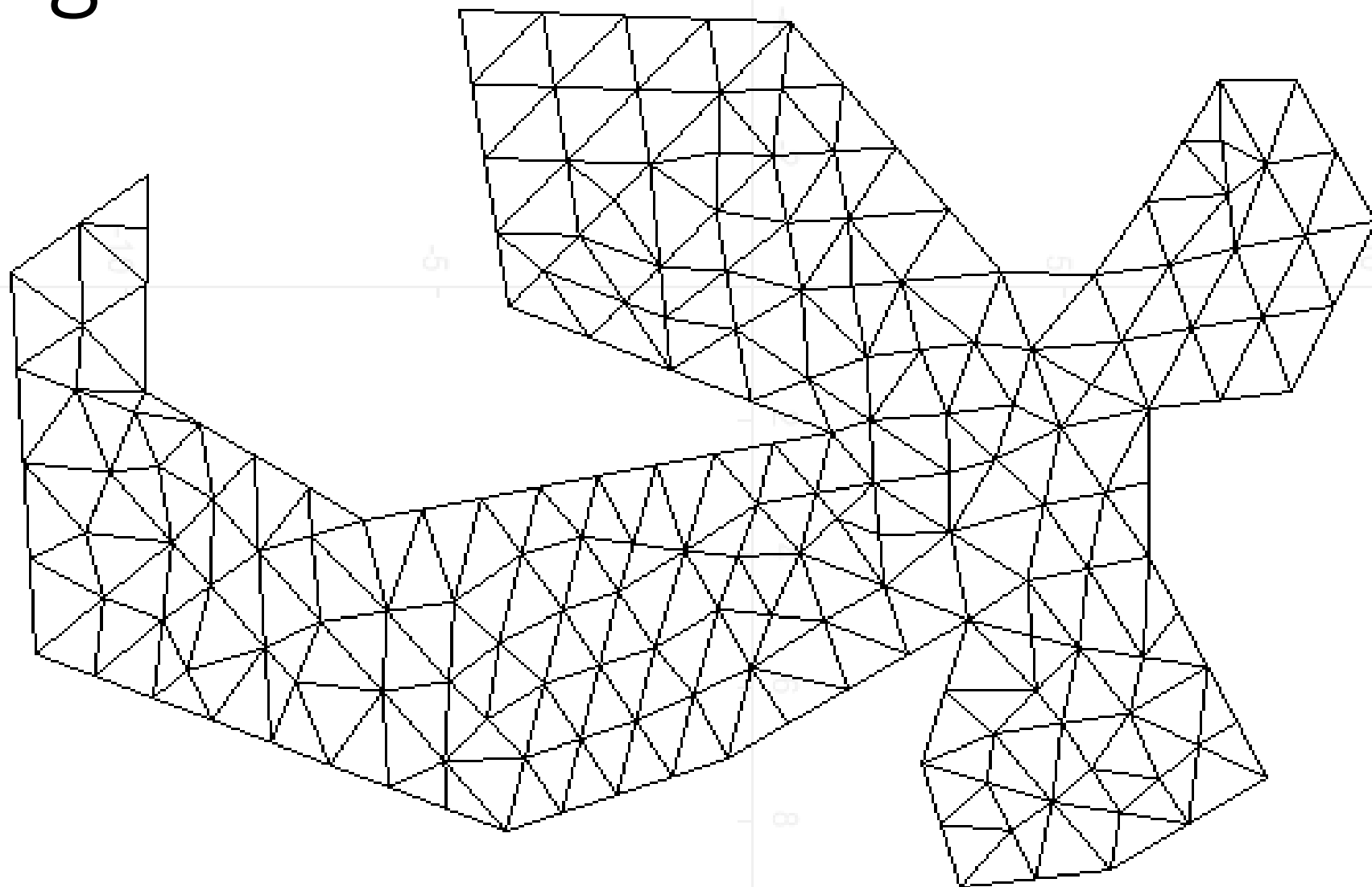
$$Q = \frac{\frac{\sqrt{3}}{4}s^2}{3s^2} = \frac{\sqrt{3}}{12}$$

Measure our Improvement

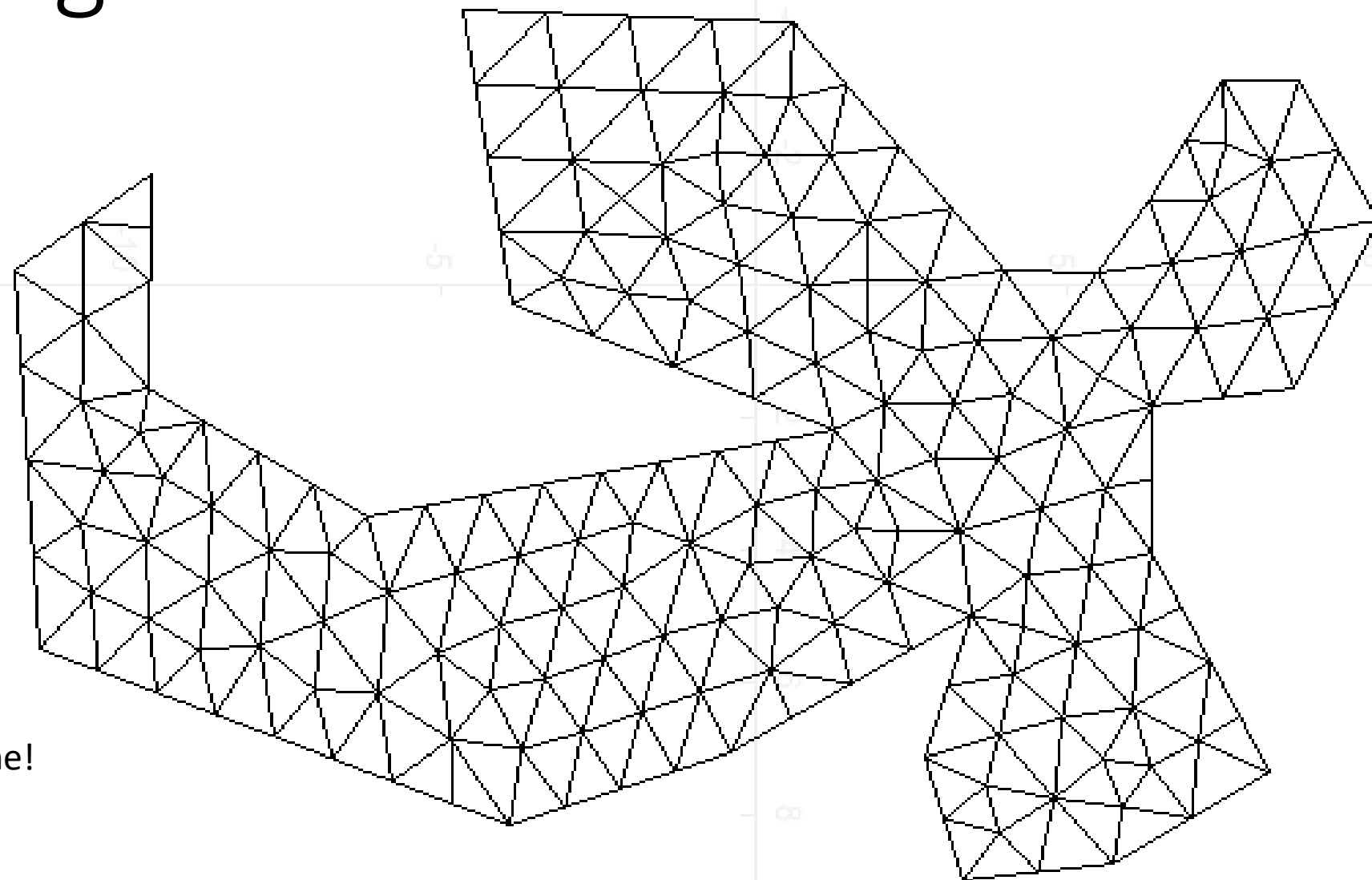
- So to represent our meshes average quality on a scale for 0 to 1 we set Overall Quality to be

$$\textit{Overall Quality} = \textit{AverageQ} * \frac{12}{\sqrt{3}}$$

Average Triangle
Quality
.90049



Average Triangle
Quality
.94010



After just one time!

Combining Methods

- As you can see this form of vertex shifting is quite powerful
- We are only doing it currently as a separate step after we have our desired number of triangles to demonstrate its effectiveness
- In practice one would have this run around 3 times for each sixth of the way to our total number goal
- In this way it would become a part of our adding program and the final result from fan and add would be ideal meshes

Future Goals/Ideas

- Speed:
 - I would love to program the methods up in C instead of Matlab, and spend some more time optimizing my algorithms
 - First thing that comes to mind is the quality sort of the triangles. Although in the beginning it is effective to target the single triangle with the absolute worst quality, as the number of triangles increase we do not by any means have to always operate on the extreme worst one
- 3D:
 - Once the speed is optimized an extension to three dimensions would not be that different!