CS478 Final Report: Implementation of the Linear Time Algorithm for Finding the Kernel of a Star-Shaped Polygon

Batuhan Tosyalı, 21702055

., .

Contents

1 INTRODUCTION

- **What is a polygon?** The definition of a polygon is, a geometric figure which has more than 4 sides[1], making the triangle is the smallest polygon in the polygon family.
- **What is a star-shaped polygon?** The definition of a star shaped polygon shaped if there exists a point z not external to P such that for all points p of P the line segment zp lies entirely within P[2].
- **What is kernel area?** The definition of Kernel is is the locus of points z where z is a point that every vp line segment is inside the polygon P where p is any of the points in polygon P[2].
- **What is a common supporting line?** The definition of a common supporting line is a supporting line which is supporting two disjoint polygons[3]. But I will be using this terms as a line segment, which are from two supported vertices.

Purpose of the Project: The purpose of this project is to demonstrate the steps of constructing a kernel for a star shaped polygon in O(n) (linear) time, which is a direct implementation of the algorithm stated in [3].

1.1 BACKGROUND LITERATURE:

As mentioned in D.T. Lee's paper [4], Shamos and Hoey, defined kernel unorthodox, in order to find a lower bound for the problem[5] , which turns out to be O(nlogn). They achieved this, via defining kernel area as; The kernel K(P) of a simple polygon P is the locus of the points internal to P

which can be joined to every vertex of P by a segment totally contained in P[2]. Which transforms the problem into the problem of intersecting half planes. Although it is true that the lower bound for intersection of half planes are O(nlogn), D.T Lee discusses that there is a difference in these problems. The difference being that in case of determining kernel, half planes are already ordered in counterclockwise order, therefore reducing the total time required proportional to O(n).

1.2 PROJECT SPECIFICATION:

The project is getting implemented in Java language, where I will be using JavaFX library for the UI and visualising the algorithm. The version of the Java is 10. There is a 200 to 100 2D plane where the user can click in to any of the integer x,y tuple therefore determining there as a vertex of polygon p. There is not any upper bounds for the points that will be used as the vertices of the polygon, but, since my display window is static, with high numbers of vertices, the chance that polygon has no kernel increases, which effects my experimental results.

1.3 TYPES OF PRIMITIVE OPERATIONS

(1) **Left test:** Left test is a test on 3 points in order to understand if the points are ordered in terms of counterclockwise or clockwise order. The result of the left test gives the twice the signed area of the triangle that can be generated from those 3 points. If the result of this operation is positive, these points are indeed in counterclockwise order, if negative they are in clockwise order. I will simply refer this constant time operation as left() in algorithm where it returns true or false depending on the result.

(2) **Slope test:** This is a test in order to tell if the selected line segment's slope is between two other line segments slope. It is needed to find the successor supporting lines between the polygon and kernel. I will simply refer this constant time operation as isSlopeBetween(), where it returns true or false. This approach is needed in part 1.1 and 2.1 of the algorithm.

(3) **Line containing point:** It is basically a test, where if the line segments passes over the given point. But since in order to see if the point is contained in the line segment, it is needed to compare the slope of the the point looked for and the regular slope of the line. But since, the plane in this system is sensitive (every point is a floating point), there are some errors in calculating the kernel area.

1.4 DATA STRUCTURES AND OBJECTS

1.4.1 Data Structures. I shall specify the data structures required for the algorithm.

- **Doubly Circular Linked List** Also as the paper suggests, it is sufficient to use a doubly circular linked List in order to hold the vertices and edges of both the Polygon and the kernel of the Polygon, where edges are stored between vertices and vertices stored between edges. To give an example, Polygon p which has let's assume 3 vertices and 3 edges are stored at the following way, in this list. "v0,e0,v1,e1,v2,e2,v0" where v denotes vertex and e denotes edges. This circular linked list is surely will be circular for the polygon, but it can be non-circular while kernel polygon K while we are in the midst of constructing the kernel. For now I haven't seen any data structures other than Doubly Circular Linked List, if I need any I will be adding on Final report.

1.4.2 Objects. Here I shall define objects in the project and algorithm which might be useful

- **Edge:** I shall define edge as an object with properties starting Coordinate ending Coordinate, slope, and direction. Indeed, this object is similar to line in JavaFx but the direction is not defined in Line of JavaFx, and since direction of the edges are playing a crucial role in this

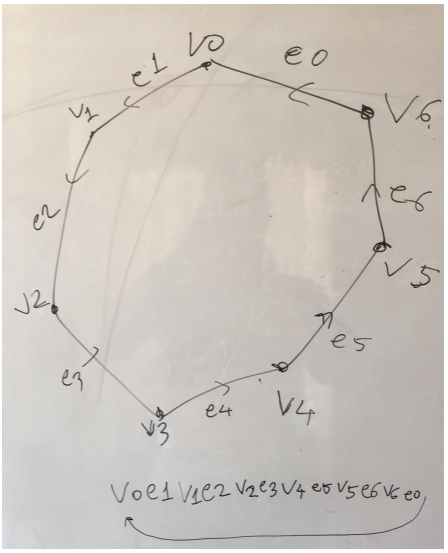
algorithm I shall define 2 functions as isleft() and isight() according to directions. And lastly a function to check if the specified point given is inside the edge.

- **Ray:** The ray I will implement is actually a' half line"[4] in the paper of D.T Lee. It will have all of the elements same as the edge above, only difference is that the ray does not have an endpoint.It's constructor can also take an edge in order to construct it as a ray.

2 ALGORITHM

2.1 Preliminaries

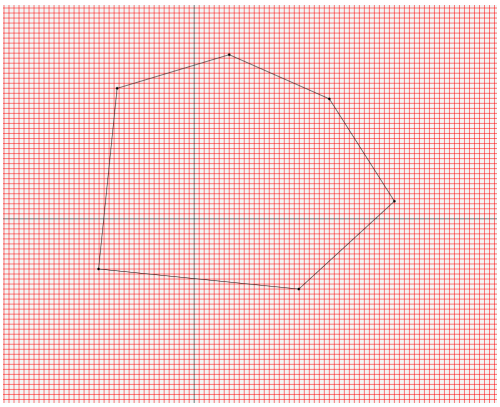
It shall be shown how the doubly circular linked list shall be implemented and shown here, in order to hold the kernel and polygon values. As can be seen in the figure below, it holds the values in counterclockwise order for both polygon and kernel area.



As can be seen here vertices are stored in a circular list for this polygon. A simple Array list is used in project, where traversing through the indexes is meant to e counterclockwise, and traversing backwards is considered as traversing clockwise.

Also, another thing to do before starting the algorithm is that,checking every vertex of the polygon, and consider the first reflex vertex. Because,The kernel of the polygons which have only convex vertices constructs kernel as the polygon itself[2], therefore if the function returns false, we can simply return kernel as the whole Polygon.

For example, since the following picture has a convex polygon, the function of mine does not do anything to find the kernel.



Also, this is the reason why the algorithm suggests starting from a reflex vertex[2].Before starting to build the algorithm vertex F and vertex L shall be defined. They are dynamic supporting lines for the algorithm. They shall be replaced again in every iteration of the building stage, where clockwise angle from f, to l, in the plane wedge containing K (kernel polygon), is no greater than pi[2]. Unfortunately, the program won't mark the locations of F and L, so there is no depiction of these line for showing.

2.2 Constructing

2.2.1 Constructing K1. The construction of K1 is as follows, After finding the first reflex vertex in the polygon, send rays from adjacent edges to infinity, and denote F and L on the infinity(as the first following picture shows).

D. T. LEE AND F. P. PREPARATA

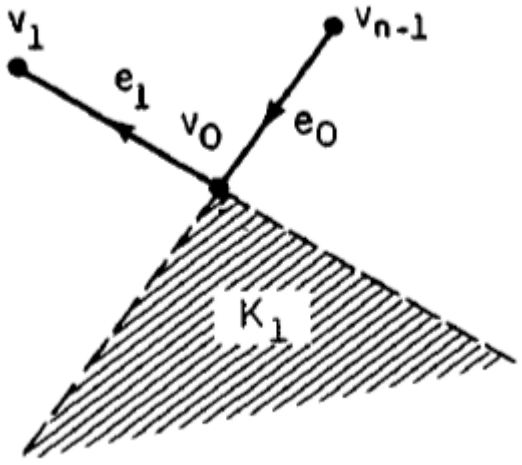
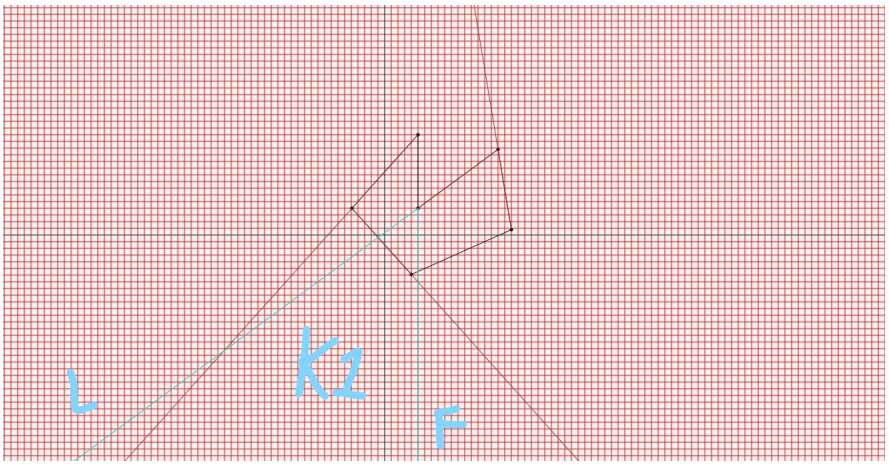


FIG 2 Illustration of polygon K_1



As seen above the first (and only) reflex vertex, sends rays to the infinity (blue rays), and denotes the kernel area K1 between the rays. Here, F and L are at infinity, although due to programming concerns, the maximum and minimum coordinates are around (100000, -100000). So that these points reside there for this occasion. Also, as seen in the picture above, the degree between F and L are less than 180, so it satisfies the conditions given for supporting vertices.

2.2.2 Constructing Ki+1 from Ki. From now on, there are 4 possibilities for constructing the Ki+1 from Ki. These being:

- The reflex is vertex and point Fi is in the right of the ray created from vertex
- The reflex is vertex but point Fi is not in the right of the ray
- The reflex is convex and point Li is in the right of the ray created from vertex
- The reflex is convex but point Li is on the right of the ray.

1. The reflex is vertex and point Fi is in the right of the ray created from vertex (1.1)

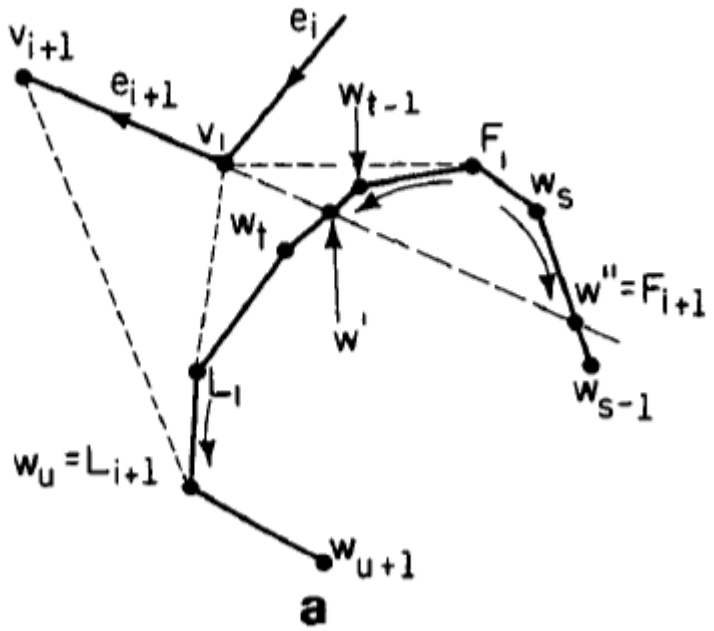


FIG 3 General step

In this step, the half line ending at point vi and directed like ei+1 should be checked through the elements of kernel counterclockwise if they intersect, meaning array list must be traversed onwards in implementation. There are three possibilities for the intersections.

- **No intersection:** This means there is no kernel.
- **One intersection:** Usual case, need to look for second intersection, and act accordingly. In order to find the second intersection the list will traversed **clockwise**.
- **Two intersection:** When traversing counterclockwise remove the parts between the intersection, that is the Ki+1.

After these step the points F and L must be updated for the next vertex of the polygon. Determination is as follows:

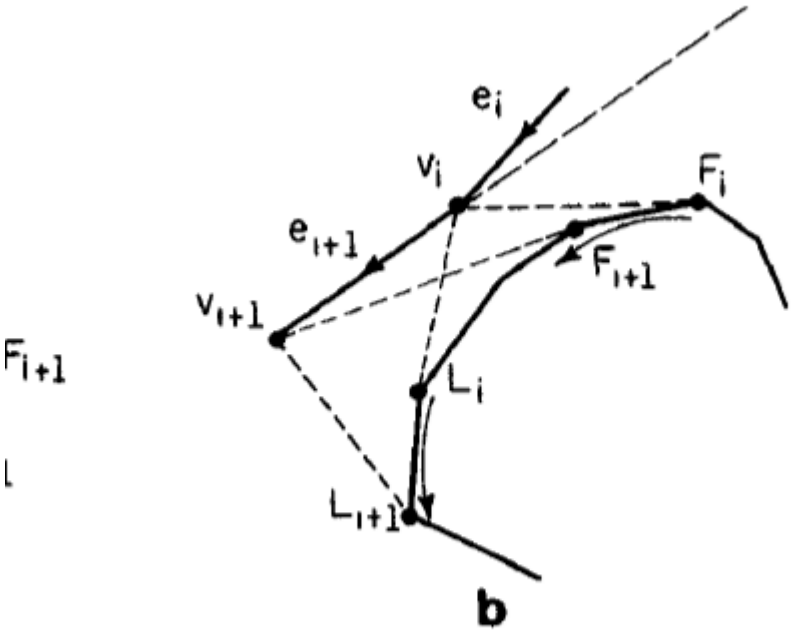
- **F:** I will give the code for F and then explain

```
if (w2 == null) {
    Ray ray2 = new Ray(polygonListCounterClockwise.get(i + 1), polygonListCounterClockwise.get(i),  terminate_initiate: 5);
    Line l2 = ray2.getRay();
    Circle c;
    Vertex temp1 = new Vertex(c = new Circle(l2.getEndX(), l2.getEndY(),  radius: 2.5, Color.AZURE));
    f = temp1;
    indexf = v1Index;
} else {
    f = w2;
    indexf = v2Index;
}
```

w2 refers to the second intersection. And as can be seen here, if second intersection is null we determine a ray ending at the i^{th} vertex of the polygon, and it's slope is same as the line segment between Vi and Vi+1. if this ray has one intersection with the kernel, we denote F to that intersection, otherwise second intersection is F.

- **L:** L is much more complex. Simply it needs to go through the kernel **clockwise**, and determine the position of the rays for i^{th} and $i+1^{th}$ vertex for each polygon vertex.

2. The reflex is vertex but point Fi is not in the right of the ray created from that vertex (1.2)



eral step when v_i is reflex

This step cannot change the kernel polygon since they won't be intersecting. Therefore only requirement is to update F and L.

- **F:** Scan **counterclockwise** (go through the list) and determine the position of the rays for i^{th} and $i+1^{th}$ vertex for each polygon vertex.
- **L:** Go through the kernel **clockwise**, and determine the position of the rays for i^{th} and $i+1^{th}$ vertex for each polygon vertex.

3. The reflex is convex and point Li is in the right of the ray created from that vertex (2.1)

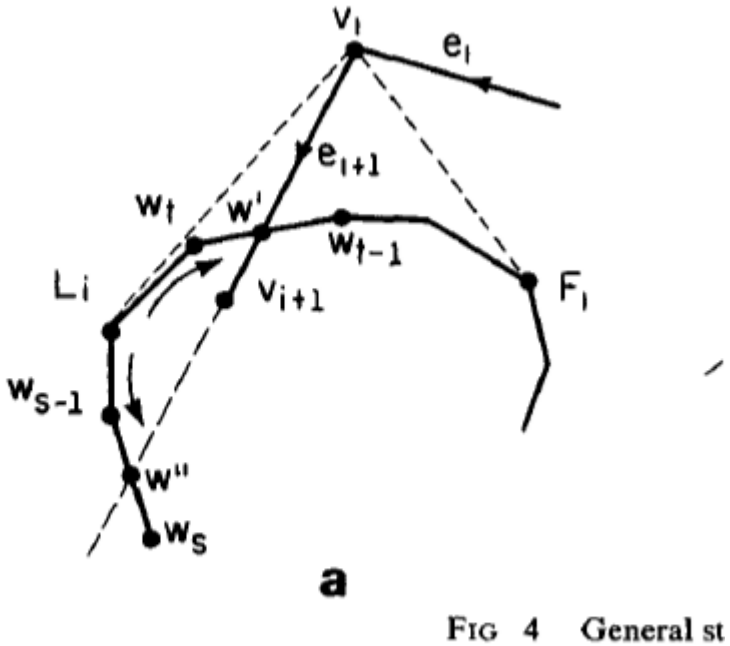


FIG 4 General st

This step is almost same as the step 1.2 . The number of intersection shall be determined.

- **No intersection:** This means there is no kernel.
- **One intersection:** Usual case, need to look for second intersection, and act accordingly. In order to find the second intersection the list will traversed textbfcounterclockwise.
- **Two intersection:** When traversing **clockwise** remove the parts between the intersection, that is the Ki+1.

The only difference here that, clockwise counters becomes counterclockwise in 1.1 and vice versa. But determination of the point F and L are different. First, it shall be determined one or two intersection have happened.

- **Two intersection:** If the **vi+1** is between first intersection **(w1)** and vertex i **(vi)** Scan **counterclockwise** (go through the list) and determine the position of the rays for i^{th} and $i+1^{th}$ vertex for each polygon vertex, and assign F. Otherwise, F is **w1**. For the case of L, the same thing happens with a small difference : If the **vi+1** is between first intersection **(w1)** and vertex i **(vi)** L is **w2**. Otherwise, go through the kernel **clockwise**, and determine the position of the rays for i^{th} and $i+1^{th}$ vertex for each polygon vertex,starting from **w2**.

- **One intersection:** If the **vi+1** is between first intersection **(w1)** and vertex i **(vi)**, scan **counterclockwise** (go through the list) and determine the position of the rays for i^{th} and $i+1^{th}$ vertex for each polygon vertex and denote F. otherwise F is **w1**. And L is set to infinity at the end of ray starting from **vi** and directed like edge between **vi** and **vi+1**.

4. The reflex is convex and point Li is in the right of the ray created from that vertex (2.2)

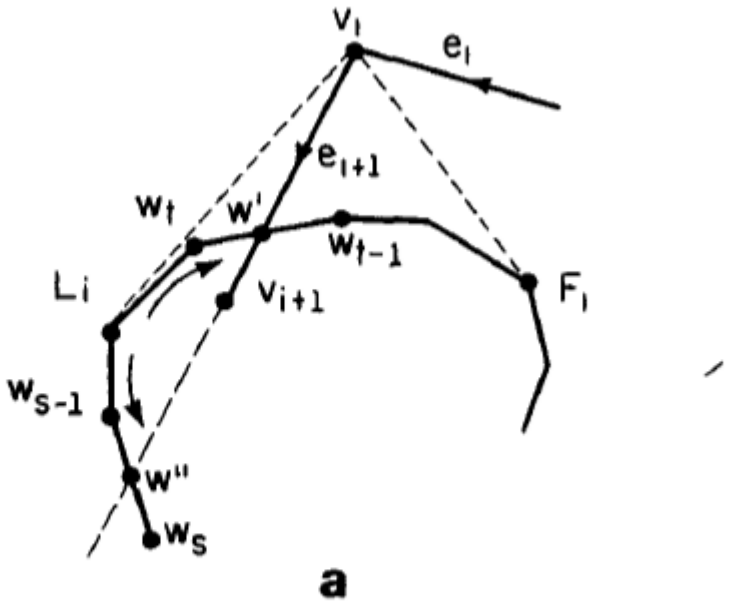


FIG 4 General st

Again, the rays from edges are not intersecting the kernel, therefore only F and L must be updated.

- **F:** Scan **counterclockwise** (go through the list) and determine the position of the rays for i^{th} and $i+1^{th}$ vertex for each polygon vertex.
- **L:** For L, first need to check if the kernel is a bounded polygon or not. If bounded go through the kernel **clockwise**, and determine the position of the rays for i^{th} and $i+1^{th}$ vertex for each polygon vertex and select L accordingly. otherwise L does not change.

3 IMPLEMENTATION

3.1 Constructing the Polygon:

I constructed the polygon with a trivial approach, found the smallest and biggest vertices in terms of x coordinate. And draw a line between them, sort the upper part and lower part of the line in terms of x coordinate, then, connect them.

```
Pseudocode:
    Vertex highest, lowest
    for i = 0 to polygon.size()
        highest = getHighestVertex()
        lowest = getLowestVertex()
    endfor
    List upperVertices, lowerVertices
    for i = 0 to polygon.size()
        if polygon.get(i).y < lowest.Y && polygon.get(i).y  <highest.Y
            lowerVertices.add(polygon.get(i))
        endif
        else
            upperVertices.add(polygon.get(i))
    endfor
    sort(upperVertices) // in terms of x
    sort(lowerVertices)// in terms of x
    // do two for loops to connect these to highest and lowest vertex.
```

3.2 Determining K1

Just go through the v0 of the polygon, and traverse through the vertices, do a left test for each of them, if it gives a negative result, it means this vertex is reflex, start from there. If you can't find any reflex vertex terminate the algorithm. Send two rays, to the infinity, one is starting from that vertex one is starting from that vertex. Put the F at the end of rightmost ray, and put the L at the other ray, with this, the rule about 180 degrees won't be broken.

```
Pseudocode:
Vertex StartVertex= null;
for i = 0 to polygon.size()
    if lefttest(!polygon.get(i-1) , polygon.get(i)
,polygon.get(i+1) )
        StartVertex = polygon.get(i)
        break
    endif
endfor
if  StartVertex = null
    endProgram()
endif
// now start looking for F and L
Ray ray1 = createRay(polygon.get(i) , polygo,get(i-1) )
Ray ray2 = createRay(polygon.get(i) , polygo,get(i+1) )
if ( ray1.isRight(ray2) )
    F = ray1.endPoint
    L = ray2.endPoint
endif
else
    F = ray2.endPoint
    L = ray1.endPoint
```

3.3 Traversing through the polygon

Since the algorithm is really complex and long, just the significant parts will be separately written.

(1) **Traversing counterclockwise and clockwise:** Traversing counterclockwise can be implemented as traversing onwards through the array list as mentioned before, since the list has already been created by going counterclockwise. Which means that going clockwise can be though as traversing backwards in the list.
(2) **Checking if F or L is right or left of the rays:** As mentioned before this can be checked with a simple leftest.
(3) **Checking if Lines Intersect:** If two line segments or lines colliding, that means from one line's perspective, two points, which are staying at the different half planes have different left test. Or, if those are two lines rather than line segments, checking the slope will be enough.
(4) **Finding intersections:** If the result above is positive, the intersection may be needed to determine **w1** and **w2** .The intersection can be found via manipulating the formula. Assume the equations of lines given as follows:

y = ax + b
y = cx + d
Now,slopes and point b and d can be found with giving 2 pairs for (x,y) in each line. After finding a , c b and d ; x can be determined as
x = (b - d) / (a-c)
After finding x y can be found simply with the line equation.

(5) **Line Contains a Point:** Pick 2 points from the line, calculate their slopes, calculate the slope again using the point looked for, if the slopes calculated are equal, that means it is inside the line. With these functions algorithm could easily be implemented, in each of the steps (1.1, 1.2, 1.3, 1.4) only the order and the parameters of the functions will change as seen in 2.2.2.

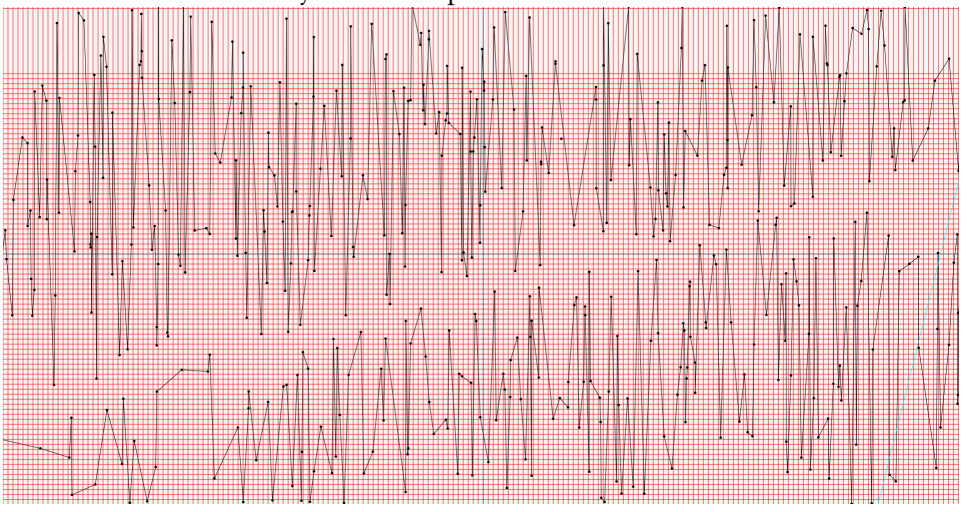
4 Testing

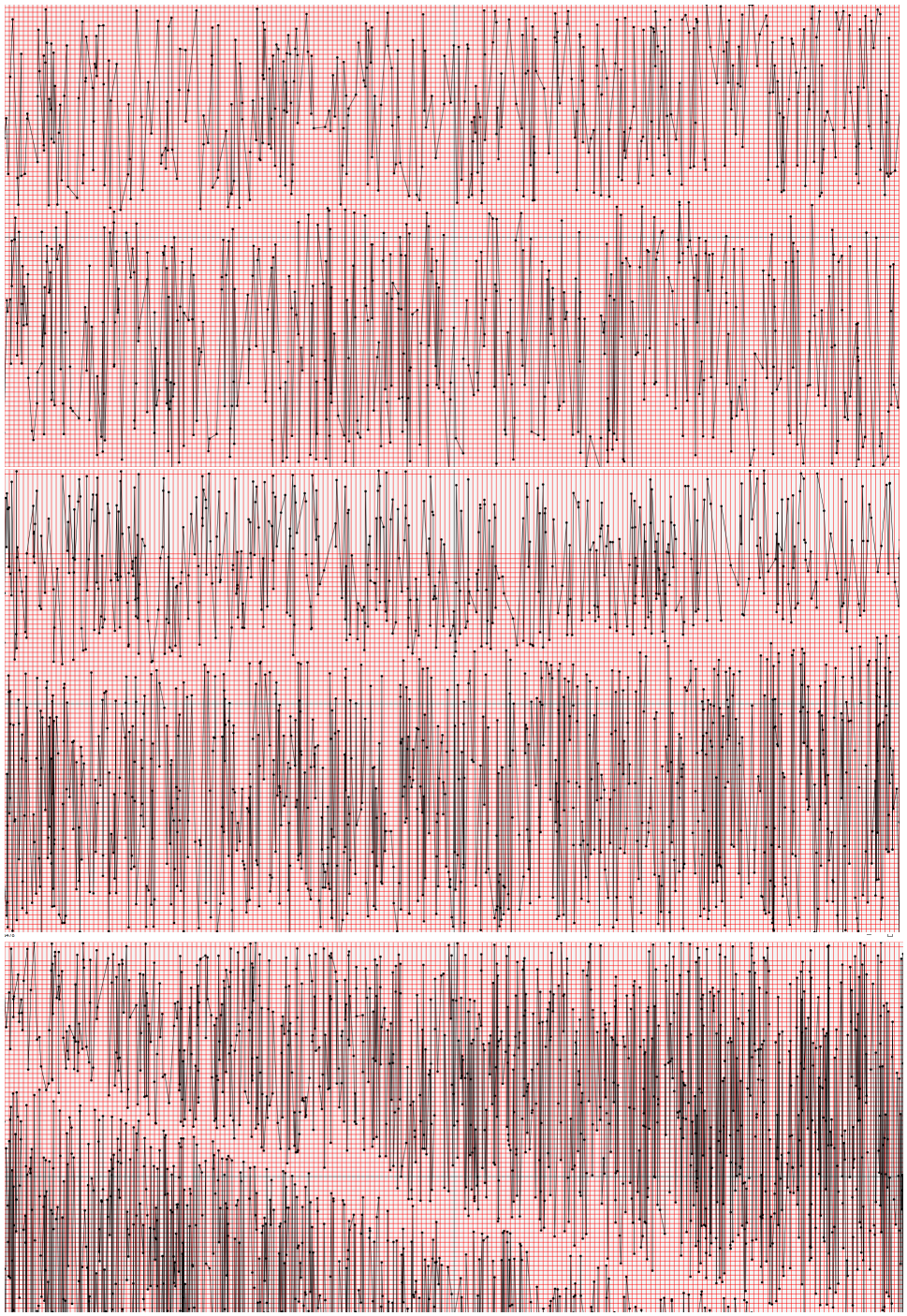
Before giving the final results,the following detail must be mentioned.

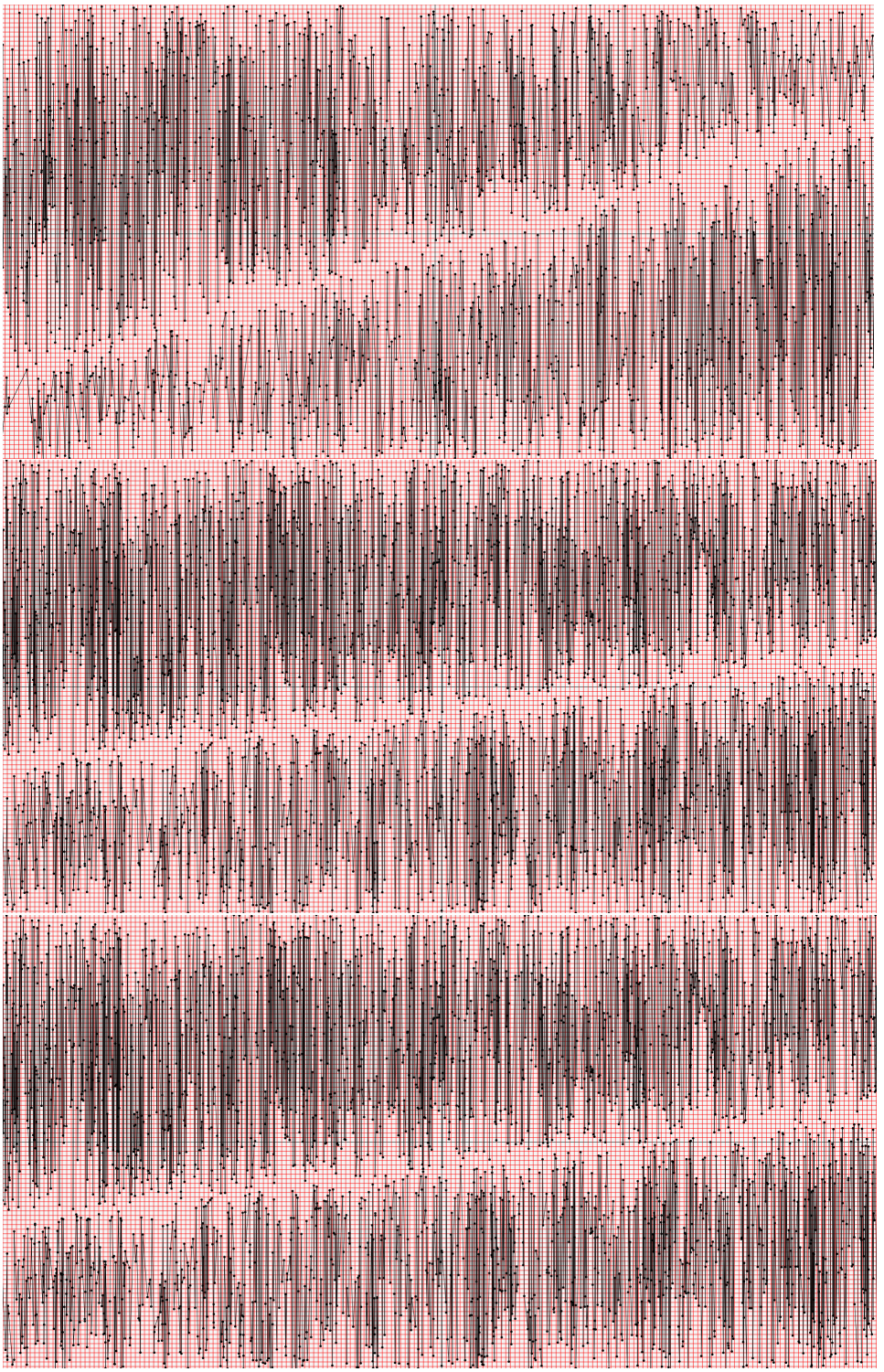
I The vertices created at random, with random coordinates on Javafx between the screen size which is 2000 for x and 1000 for y. Meaning anything out of the scope could not be visualised, therefore the program compresses high amount of vertices in a small place, resulting in these polygons does not have a kernel. in order to conduct the tests program lets itself loop through all of the vertices of the polygon and act accordingly even though there is no kernel.I believe that with this solution the run time can be approximated.

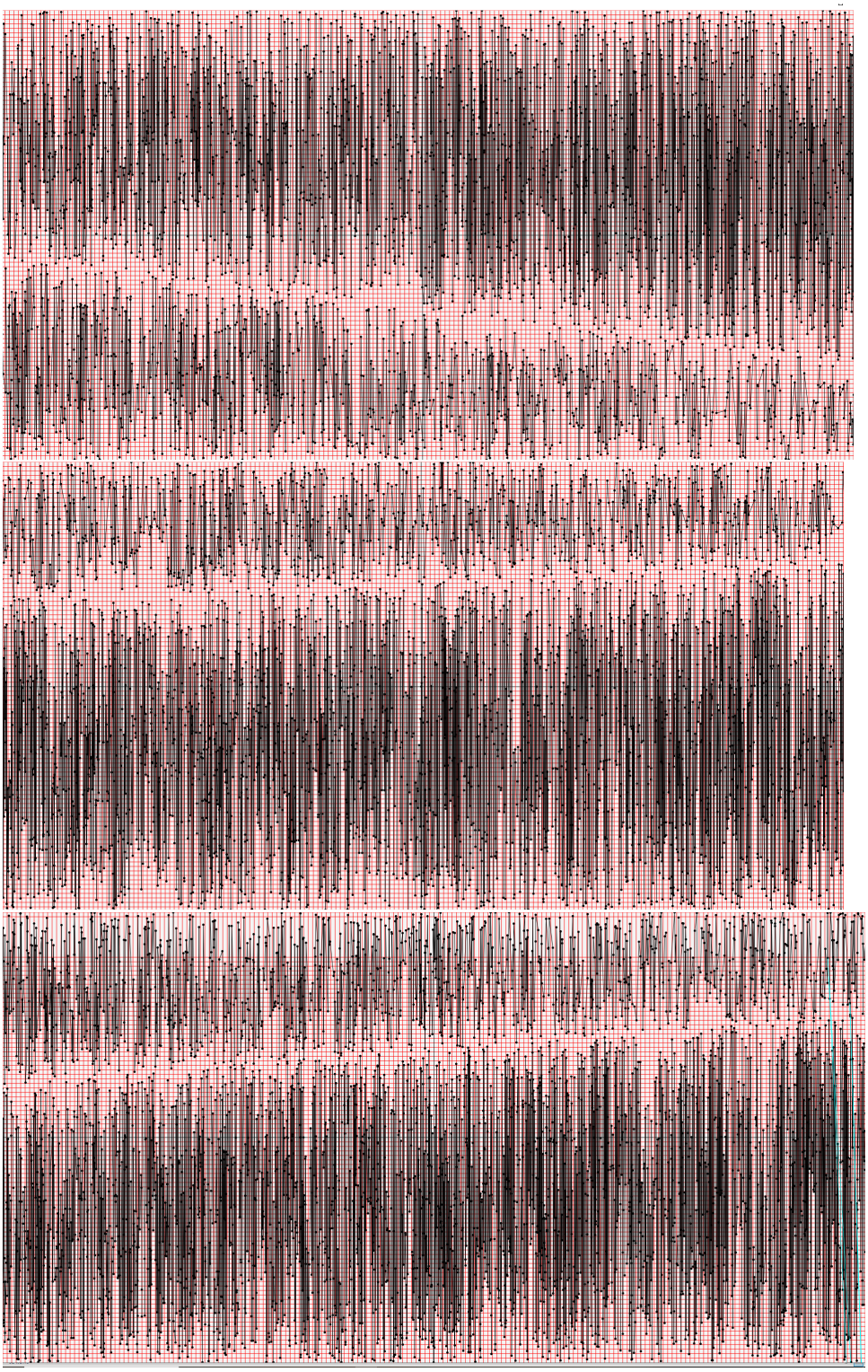
II The algorithm I was working on when I was making a presentation was flawed, thus new form inspects much larger data group, making a better graph.

Now, before the result graph, here are the polygons which times has been based on: Starting from 500 vertex increases by 500 in each picture.

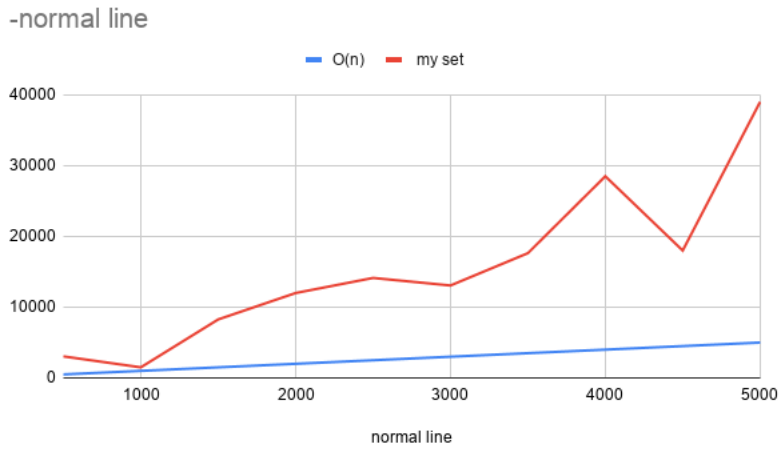








Here is the graph of it :



The time consumed by the set of those polygon in term of milliseconds are denoted in red, and a straight 45 degree line has been denoted with blue. As seen here although there seems to be a big constant for O(n) complexity of this algorithm, thus showing us the results are bounded with O(n).

References

[1] F. P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. SpringerVerlag, 1985.
[2] John Craig. *-A New Universal Etymological, Technological, and Pronouncing Dictionary of the English Language*. Routledge, London, 1858.
[3] Lee, D. T., and F. P. Preparata. *An Optimal Algorithm For Finding The Kernel Of A Polygon*. Journal Of The ACM (JACM), vol 26, no. 3, 1979, pp. 415-421. *Association For Computing Machinery (ACM)*
[4] Michel M. Deza, Elena Deza. *Encyclopedia of Distances*. 2018.
[5] SHAMOS, M I, AND HOEY, D. *Geometric intersection problems 17th Annual Syrup on Foundanons of Computer Science*. Houston, Tex, Oct 1976, pp 208-215 (1EEE).