CS 478 Final Report

Project Name:

Implementing Three Voronoi Diagram Computation Algorithms and Comparing Their Performance

Students:

Kaan Kurçer (21902348), Osman Serhat Yılmaz (21902903)

Project Description:

The program will generate a set of random points in two dimensions using various distributions as input and will calculate as well as visualize the 2D Voronoi Diagram as a graphical output. The program will allow the users to specify parameters such as number of points. It will also include features such as zooming in/out and translation while displaying the 2D Voronoi Diagram.

The following Voronoi Diagram computation algorithms will be used:

- Randomized Incremental Algorithm
- Fortune's Algorithm
- The Flipping Algorithm

The program will be tested with a number of arbitrary point sets. Performance results will be noted and performance of the algorithms will be compared for different numbers of points.

For this project we decided to use Python because of our familiarity with working with Python, abundance of libraries related to our topic which we can use and well described documentation which serve our purpose.

Table of Contents

1. Voronoi Diagram.. 3
2. Algorithms.. 4
2.1. Randomized Incremental Algorithm.. 4
2.2. Fortune's Algorithm.. 6
2.3. The Flipping Algorithm... 7
3. Implementation... 9
3.1. Algorithm Implementation Details..................................... 10
3.1.1. Randomized Incremental Algorithm.......................... 10
3.1.2. Flipping Algorithm.. 11
4. Results.. 13

1. Voronoi Diagram

A Voronoi diagram is a mathematical concept used to partition a given space into regions based on the proximity to a set of points in the space. It's a way of dividing up a space based on which point in a set of points is closest to any given point in that space.

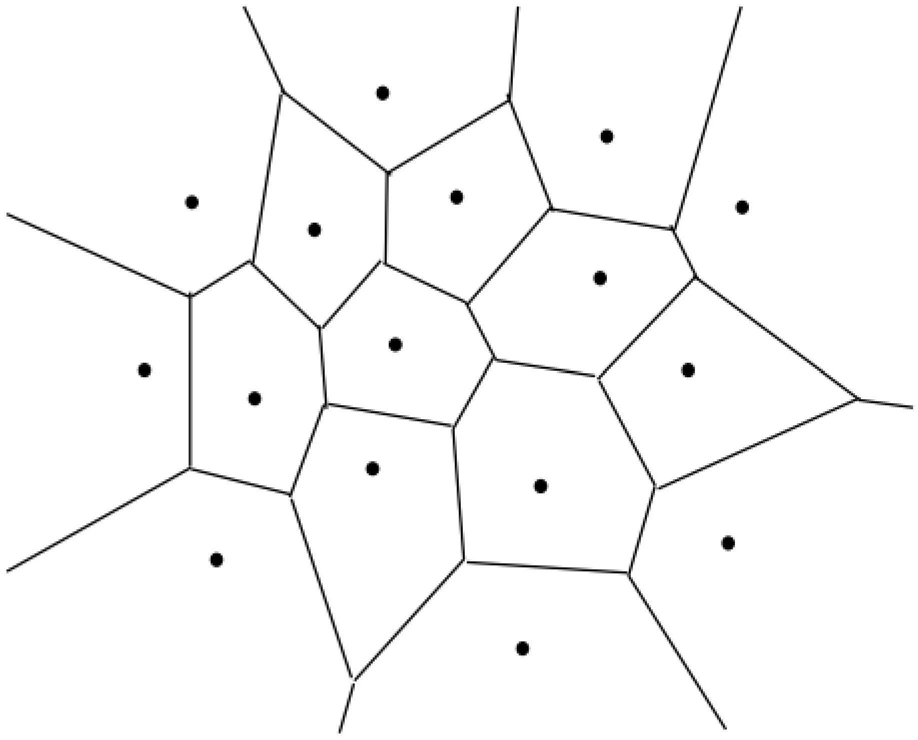


Figure 1: An example of a Voronoi Diagram

To construct a Voronoi diagram, a set of points are first placed in the space. Then the Voronoi diagram defines a cell for every point in the space that corresponds to the generator that is closest to that point, while the cell being the set of all points that are closer to that generator than any other generator. The Voronoi diagram forms a pattern of polygonal shapes, each cell having a unique shape that is defined by the generators. These shapes can be represented as a diagram, where the edges of the polygons represent the boundaries between the cells.

2. Algorithms

Here are the three algorithms that we will implement for this project, after that we will test and compare the performances of these three algorithms. We will demonstrate our results in the project final report.

2.1. Randomized Incremental Algorithm

Initially, Randomized Incremental Algorithm will be used to construct a Delaunay Triangulation from the given points. We add sites one by one into the Delaunay Triangulation and update the Delaunay Triangulation after each insertion. This update consists of discovering all Delaunay faces whose circumspheres contain the new site. These faces are deleted and the empty region is partitioned into new faces, each of which has the new site as a vertex.

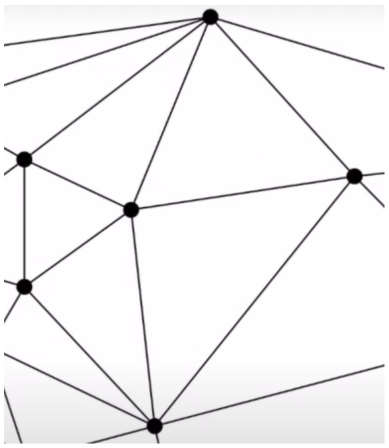
The algorithm works as follows;

1. A triangle which contains all of the points is generated.
2. A random point which hasn't been selected already is chosen.
3. A line is drawn to each of the vertices of the triangle which contains the point.
4. For each edge of the triangle, a circumcircle which contains the points that make up that edge and the chosen point is drawn.
    4.1. If the circumcircle contains another point which has been introduced into the algorithm, draw a line from the three points to that point. Then do the same check for each of the newly generated edges.
5. Do steps 2 - 4 until all points are chosen.
6. Remove the initial triangle and all the edges that are connected to its vertices.

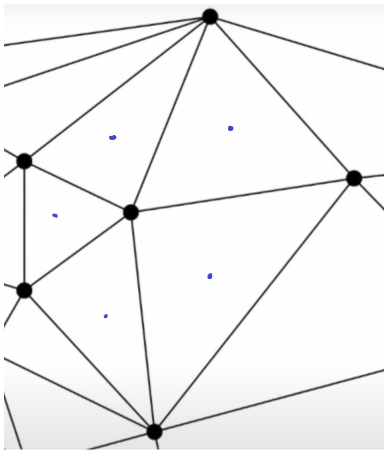
After obtaining the Delaunay Triangulation with the use of Randomized Incremental Algorithm, we use the fact that a Delaunay Triangulation is the dual graph of a Voronoi Diagram. This means that a face in a Delaunay Triangulation represents a vertex in the Voronoi Diagram and vice versa. We'll use this fact to construct an algorithm as such:

1. Generate a Delaunay Triangulation from the given points using Randomized Incremental Algorithm.
2. Determine the center point of each circumcircle of a triangle that forms a Delaunay Face.
3. Draw a line from one center point to each of its neighboring center points for every single center point.
4. Remove the edges that were generated by the Randomized Incremental Algorithm.

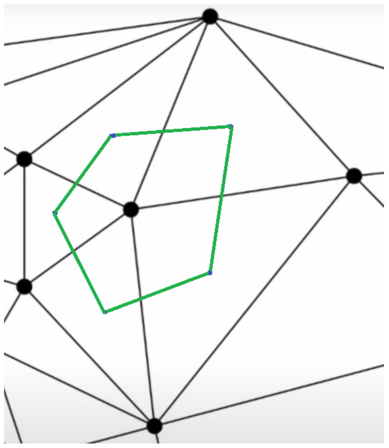
The end result of the algorithm will be the Voronoi Diagram created from the given points.
A visual representation of the algorithm can be seen below;



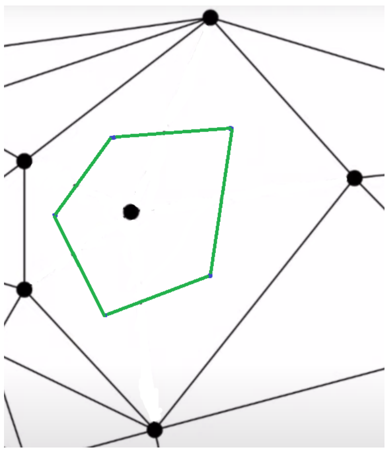
Step 1:
A Delaunay Triangulation was formed using the Randomized Incremental Algorithm.



Step 2:
The center points of the circumcircles of the triangles which form the Delaunay faces are determined.



Step 3:
The determined points are then connected to each of their neighboring center points via a line.



Step 4:
The edges generated from the Delaunay Triangulation are removed, and the Voronoi Diagram is obtained.

2.2. Fortune's Algorithm

Fortune's Algorithm is a sweep line algorithm, it processes the input data linearly using a line that sweeps across the input points. The algorithm uses a priority queue to store and process the events that occur as the sweep line moves across the input points. These events can be classified into two types: point events, vertex events.

Point events occur when the sweep line encounters a new input point, and these points are used to define the Voronoi cells. When a point event occurs, a new parabolic arc is created in the beach line, which is a data structure that represents the current state of the diagram as the sweep line moves across the input points.

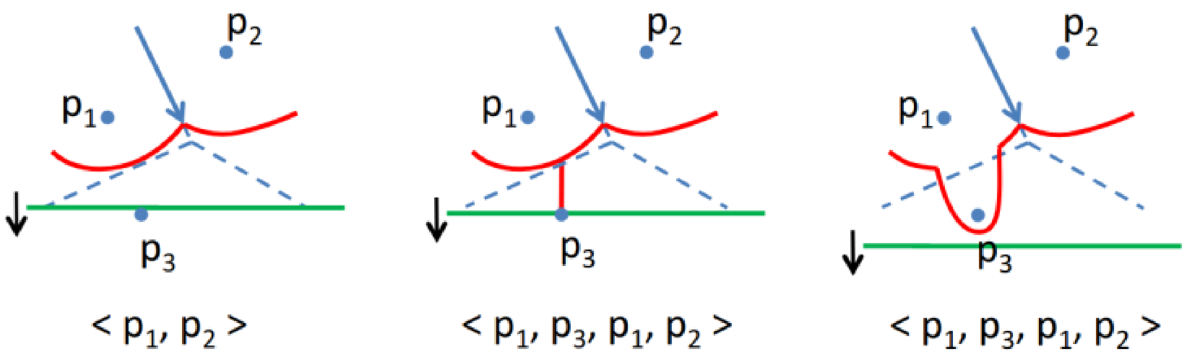


Figure 2: Point Events

Vertex events occur when the sweep line reaches the bottom of a parabolic arc in the beach line. These events represent the creation of a new Voronoi vertex, which is the intersection point of three Voronoi edges. The algorithm uses a vertex event queue to store these events, and processes them as they occur.

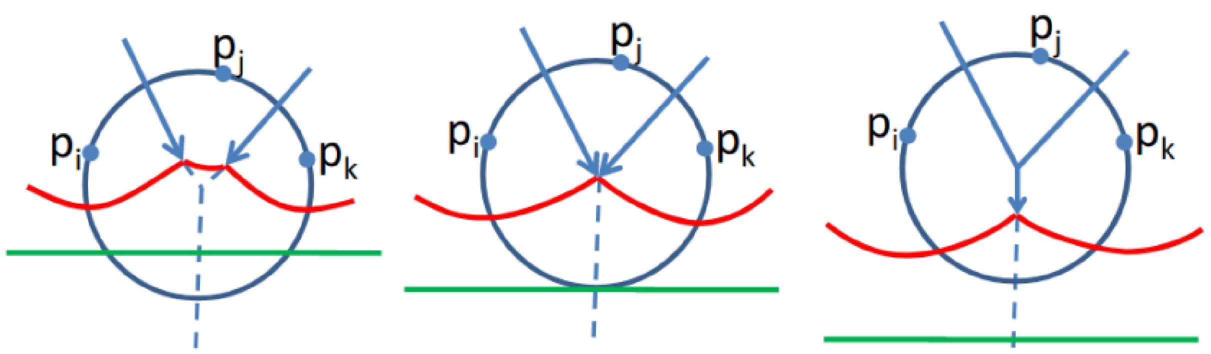


Figure 3: Vertex Events

By processing these events in the appropriate order, Fortune's algorithm is able to construct the Voronoi diagram.

2.3. The Flipping Algorithm

The flipping algorithm is used to improve the quality of a triangulation. The goal of the flipping algorithm is to improve the aspect ratio of the triangles in the triangulation. The flipping algorithm requires an initial arbitrary triangulation; for this purpose, we will use Lexicographic Triangulation.

Lexicographic triangulation is a type of triangulation of a set of points in the plane that is constructed by first ordering the points lexicographically. It creates a "skeleton" of a triangulation by connecting adjacent points in the lexicographic order, and then to fill in the remaining triangles to complete the triangulation. To do this, the algorithm proceeds as follows:

1. Sort the input points lexicographically.
2. Connect each adjacent pair of points in the sorted order with a straight line segment.
3. For each set of three consecutive points in the sorted order, check whether the triangle connecting those points is already in the triangulation. If it is not, add it to the triangulation.
4. For each set of four consecutive points in the sorted order, check whether the two triangles formed by connecting the first three points and the last three points are already in the triangulation. If they are not, add them to the triangulation.
5. Continue this process for each set of five consecutive points, and so on, until all possible triangles have been added to the triangulation.

The resulting triangulation will create a lexicographic triangulation, then we will improve this triangulation using the flipping algorithm.

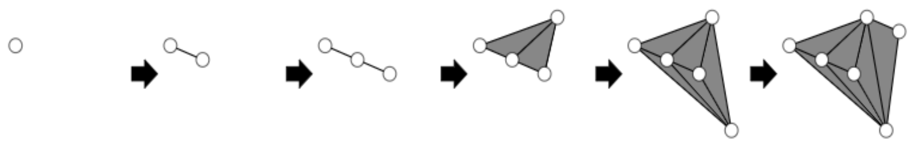


Figure 4: Lexicographic Triangulation

The flipping algorithm works by identifying pairs of triangles in the triangulation that share an edge and if the circumcircles of this pair triangle intersect it means that the two triangles are not neighbors. So the shared edge should be flipped to create two new triangles. To perform the flip, the common edge is removed, and the new vertices are connected to form two new triangles. The new triangles are then checked to see if they satisfy the Delaunay condition, which is that no point is inside the circumcircle of any triangle, and if not, the process is repeated until the Delaunay condition is satisfied.

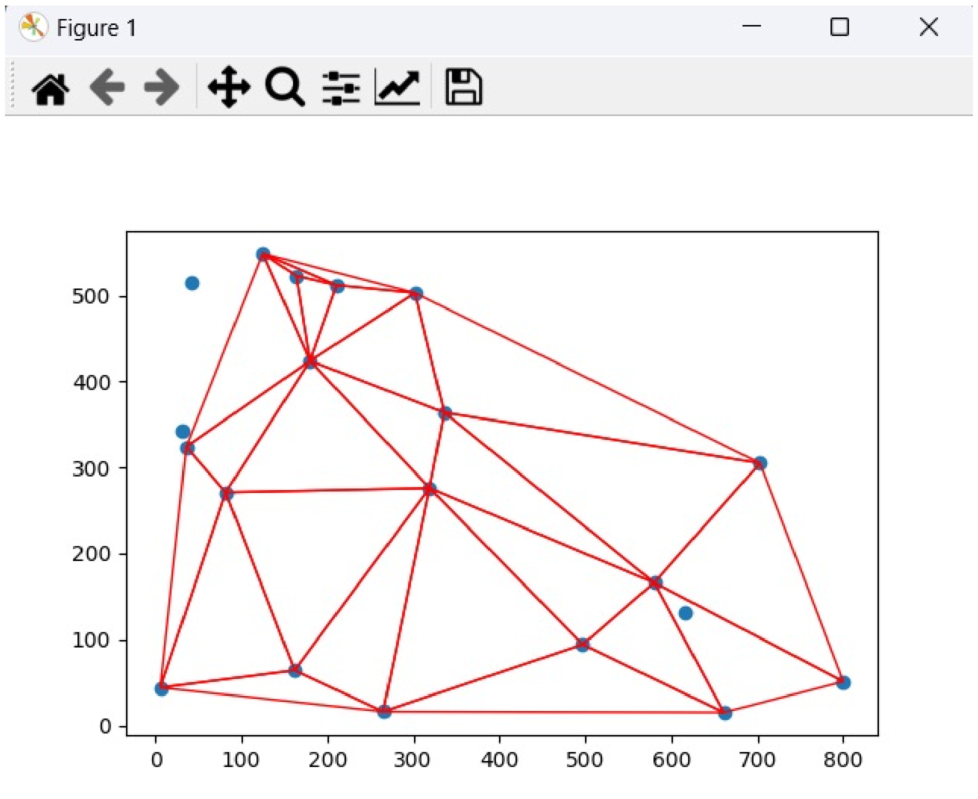
The flipping algorithm can be used to iteratively improve the quality of a Delaunay triangulation by identifying and flipping edges that violate the Delaunay condition. This can lead to triangulations with fewer long, skinny triangles and more regular shapes. After the Flipping Algorithm is completed, we will convert this Delaunay triangulation to a Voronoi diagram, by implementing the same steps we explained in the Randomized Incremental Algorithm subsection.

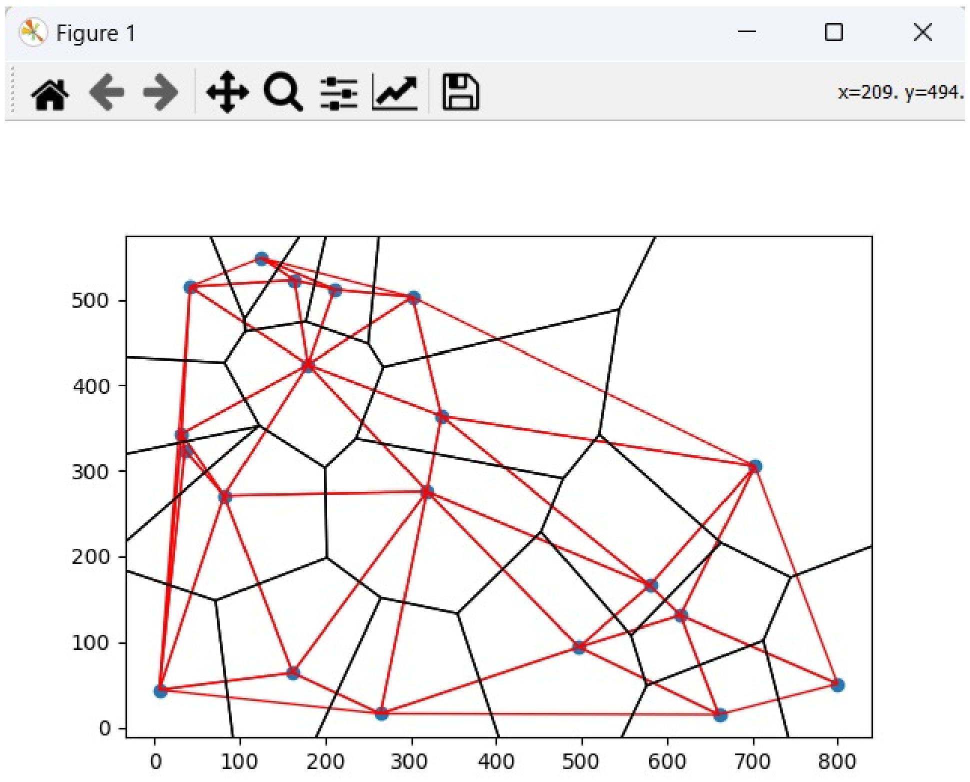
3. Implementation

During the implementation of the project Python was used as the programming language. The following libraries were used;

- NumPy for mathematical functions
- Matplotlib for plotting the diagrams

The application is run through a single executable where the user can choose which algorithm they want to try through the user interface. The program allows the user to specify what kind of distribution they want as well as how many points they want for point generation. Then, the program displays step-by-step building steps of the Voronoi Diagram. The user can also toggle the Delaunay Triangulation and the Voronoi Diagram if they want to see them both or see only one.





3.1. Algorithm Implementation Details

In our project, we implemented 2 of the 3 algorithms, these algorithms are randomized incremental and flipping.

3.1.1. Randomized Incremental Algorithm

Below are the used functions and classes for the implementation of this algorithm:

- The *DelaunayRI* class initializes with a center and radius, which define the bounding triangle of the triangulation. The coordinates of the bounding triangle are stored, and initial triangles and their circumcenters are calculated.

- The *circumcenter* method computes the circumcenter and radius of a given triangle using the geometric properties of the triangle.

- The *inCircleFast* method checks if a point lies within the circumcircle of a given triangle.

- The *addPoint* method adds a new point to the triangulation. It first identifies the "bad" triangles whose circumcircles contain the new point. It then finds the boundary of the affected region by traversing neighboring triangles. The bad triangles are removed, and new triangles are created based on the boundary edges and the new point.

- The *exportDT* method exports the coordinates and triangles of the Delaunay triangulation. It filters out the coordinates and triangles related to the bounding triangle.

- The *exportVoronoiRegions* method exports the Voronoi regions and Voronoi vertices. It collects the vertices of each region by traversing the neighboring triangles.

- The *plot_triangles_RI* and *plot_voronoi_RI* functions are helper functions to visualize the triangulation and Voronoi diagram using matplotlib.

- The *get_triangles_RI* and *get_regions_RI* functions convert the coordinate indices to actual coordinate points for visualization purposes.

- The *calculate_voronoi_RI* function takes a list of points, iteratively adds them to the triangulation, and updates the visualization at each step. Finally, it computes and plots the Voronoi diagram.

3.1.2. Flipping Algorithm

Below are the used functions for the implementation of this algorithm:

- *findCircumCenter(P, Q, R)*: This function calculates the circumcenter of a given triangle formed by three points (P, Q, R) in 2D space. It uses helper functions to calculate the line equations and find the intersection point.

- *is_illegal(pr, edge, pk)*: This function checks if a given point pr violates the Delaunay condition with respect to an edge and an adjacent triangle. It calls the *findCircumCenter* function and uses the math.dist function to calculate the Euclidean distance between points.

- *find_adjacent_triangle(pr, pi, pj, T)*: This function finds the third vertex (pl) of an adjacent triangle given two vertices (pi, pj) and the current set of triangles T.

- *flip_edge(edge1, edge2, T)*: This function flips the given edge (edge1) with another edge (edge2) in the triangulation. It removes the triangle formed by edge1 from T and creates two new triangles with edge2.

- *find_initial_triangle(P)*: This function calculates the initial triangle that bounds the set of input points P. It determines the minimum and maximum coordinates of the points and creates an equilateral triangle centered at their midpoint.
- *find_triangle_containing_point(pr, T)*: This function finds the triangle in the current triangulation T that contains a given point pr. It checks if pr is inside the triangle using the is_inside_triangle function.

- *split_triangle(pr, pi, pj, pk, T)*: This function splits the triangle formed by three vertices (pi, pj, pk) in the triangulation T into three new triangles by adding edges from pr to each vertex.

- *find_third_vertex(pr, pi, pj, T)*: This function finds the third vertex of a triangle in the triangulation T given two vertices (pi, pj) and the point pr. It looks for a triangle in T that contains pi and pj but not pr and returns the adjacent vertex.

- *split_triangles_with_edge(pr, pi, pj, pk, pl, T)*: This function splits two triangles in the triangulation T by adding edges from pr to each pair of adjacent vertices (pi, pj, pk, pl).

- *remove_initial_triangle(p_minus, T)*: This function removes the initial bounding triangle (p_minus) from the triangulation T.

- *generate_points(distribution, num_points)*: This function generates a set of points based on a specified distribution ('random', 'uniform', or 'gaussian') and the number of points.

- *LEGALIZEEDGE(pr, edge, T)*: This function ensures the legality of an edge in the triangulation with respect to a given point pr. It recursively flips any illegal edges until the Delaunay condition is satisfied.

- *DELAUNAYTRIANGULATION(P)*: This function performs the Delaunay triangulation on a set of input points P. It iteratively adds each point to the triangulation, splits triangles, and legalizes edges as necessary.

- *plot_triangles(fig, ax, T)*: This function visualizes the triangulation by plotting the triangles using matplotlib.

- *plot_voronoi(points, edges, T)*: This function plots the Voronoi diagram based on the Delaunay triangulation. It takes the set of points, the Voronoi edges, and the triangulation T as inputs.

4. Results

In order to test the algorithms' efficiency, their run-times were analyzed. Points were generated using three different distributions: random, uniform and gaussian. Different amounts of points were tested on each algorithm. All of the times that are given in the charts below are in seconds.

Flipping Algorithm	**Random**	**Uniform**	**Gaussian**
64 Points	0.0865438	0.0977953	0.0957385
256 Points	1.4456029	1.505821	1.5920878
512 Points	6.6161054	6.2149324	6.5254246
1024 Points	24.8294821	25.4814265	34.2482546
2048 Points	125.3442515	128.7920128	112.4517881

Randomized Incremental Algorithm	**Random**	**Uniform**	**Gaussian**
64 Points	0.0547292	0.0512497	0.0303546
256 Points	0.6181948	0.6153923	0.3640604
512 Points	2.1364411	2.0782666	1.3257558
1024 Points	8.3478266	8.5434931	5.6070762
2048 Points	33.0469071	19.9199718	17.7604167

These results indicate that;

- In general, Random Incremental Algorithm performs better than Flipping Algorithm regardless of the input size or the distribution of points.
- Comparing the point distributions in Random Incremental Algorithm, it was found that Gaussian Distribution is the most efficient, whereas Random Distribution is the least efficient.
- Comparing the point distributions in The Flipping Algorithm, it was found that Random Distribution is the most efficient, whereas Gaussian Distribution is the least efficient.