Exploring Delaunay Triangulations
COMP 4202 Final Project
Justin Bacic
101233929

Delaunay Triangulation Introduction:

A fundamental issue of GIS is collecting and storing data to meet the requirements of the application. Since space and resources are finite there are a few methods to get approximations of location attributes without needing to collect and store data at every single point in the region of interest. One of these such methods are triangulations, or TINs, where data points are connected to form triangles and location attributes of points that are not stored can be interpolated from the adjacent vertices. Since we are doing interpolation we want to make sure that we are constructing our triangulation in such a way that these interpolations are more likely to be accurate. If we look at the following example (Figure 1), we can see two different triangulations of the same points each of which have an attribute, which we can take to mean elevation in this instance, but could be anything. We can see the vastly different interpolations they yield of our point of interest (marked in red). We can reason that it is likely that triangulation 2 is more accurate as the points we are using for interpolation are closer to the point of interest.

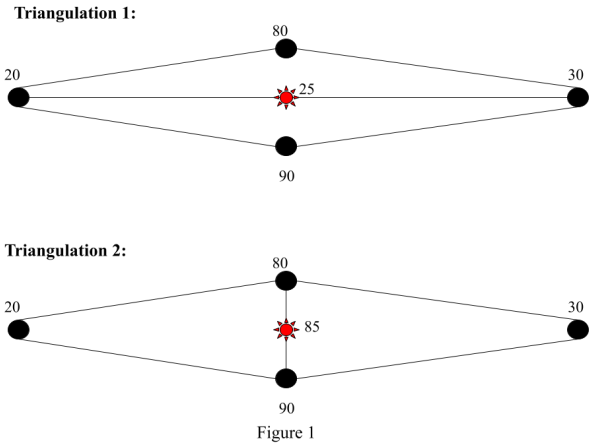


Figure 1

Now that we can see the importance of triangulations and more specifically having a good triangulation we now need to decide how we classify a good triangulation. As we can observe in Figure 1, when points closer together are triangulated it results in the triangles having more similar angles. On the other end when we have triangulations that connect far away points it results in triangles that have very large and very small angles. The Delaunay triangulation is a triangulation that maximizes the minimum angle of the triangulation and thus gives us a triangulation where each triangle has angles that are similar to each other. Since it satisfies this property Delaunay triangulations are used a lot in GIS for the aforementioned reasons of allowing us to make more accurate interpolations with our current set of data points.

Project Overview:

We have already seen why Delaunay triangulations are so important for producing accurate and workable data, so the rest of the project is focused on the actual construction of the triangulation versus the real world applications. The goal of my project is to explore the construction of a Delaunay triangulation, as seen in our lectures, in three ways: by implementing the incremental construction algorithm, creating a visualisation tool to see how the algorithm

works, and analysing the effect of randomization on the runtime. The expected outcomes of this project were a deeper understanding of how the underlying data structures change as the construction algorithm runs, and seeing in action how randomization benefits the algorithm runtime.

Brief Algorithm Overview (From the literature):

The Delaunay triangulation construction algorithm works incrementally where we insert a point into the current triangulation, retriangulate, make sure the Delaunay properties hold and repeat (3). When we insert a point into the triangulation we want it to fall within a preexisting triangle so that it can just be triangulated with the three points that define that triangle. To ensure that this is always the case the algorithm starts with constructing a super triangle that contains all of the points (3). For a triangulation to be Delaunay it requires that the circumcircle of every triangle is empty, and any triangulation that has this property is a Delaunay triangulation, this is very useful when we consider the construction as we can calculate this property for each newly created triangle (3). If we find that the circumcircle of our triangles are not empty we need to perform edge flips until all the circumcircles are empty. Edge flips are where we are given two adjacent triangles, we change the orientation of the shared edge to create two new triangles with the same points, this will be illustrated later on. After all points are inserted the resulting triangulation will be a Delaunay Triangulation. This gives us the starting point from which we can start to fill in the details of the algorithm during the implementation phase.

Implementation Details:

Now we can go into the discussion of my specific implementation of the Delaunay Triangulation in Python. This was based on the outline of the algorithm that we saw in class, this section will serve to highlight the challenges I encountered as well as the gaps in detail that I filled in from the lecture to get a working implementation. This section will be broken into parts, the underlying data structure, the insertion of points, and edge flips.

Data Structure:

The way that the Delaunay triangulation is stored is with a doubly-connected edge list (DCEL). DCELs are commonly used to store planar graphs and store records for the following three elements Vertices, Half-Edges and Faces. Vertices traditionally only store the coordinates of the point as well as a pointer to an arbitrary incident half-edge, but in my scheme I included a pointer to each of the outgoing edges from a vertex which will be useful later. Half-Edges are directed edges that store their origin vertex, the face they are incident to, a pointer to the next and previous edges along the perimeter of the face, and a pointer to its “twin” which is a half-edge that is oriented in the opposite direction. Finally, faces represent regions enclosed by half-edges and store a pointer to one of the half-edges on its outer boundary and a pointer to some half-edge of each hole, which in a triangulation there are no holes so we don’t need to worry about this attribute.

For the implementation, the DCEL was extended to fit the specialized nature of the triangulation. To do this I created a Triangulation class which had the vertices, half-edges, and faces of a DCEL but also had three other key data structures. The first was a list of the points which had not yet been inserted, it was important that the class was able to keep this information so that it would be able to efficiently handle point location. Additionally, the order of this list will be the order in which the points are inserted into the triangulation which will be important in the runtime analysis section. The second and third were maps from points to triangles, and from triangles to points, again these are necessary so that the point location can be done efficiently as will be discussed later. The triangulation also calculates and stores a super triangle that encloses all of the points as part of the initialization, which allows us to ensure that all of our points lie within a triangle that is already in the triangulation.

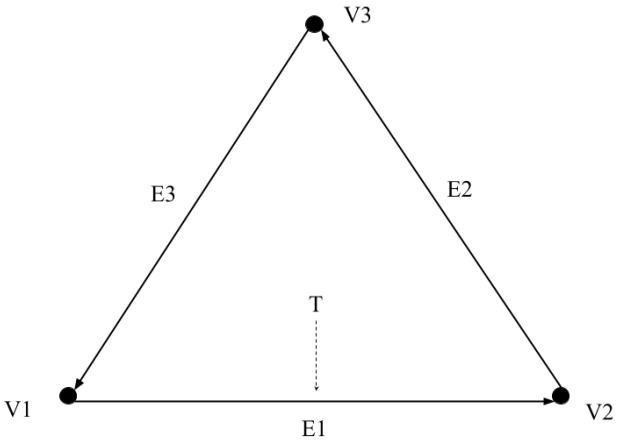
Point insertion:

When working on the insertion points there were some key problems that needed to be addressed and considered.

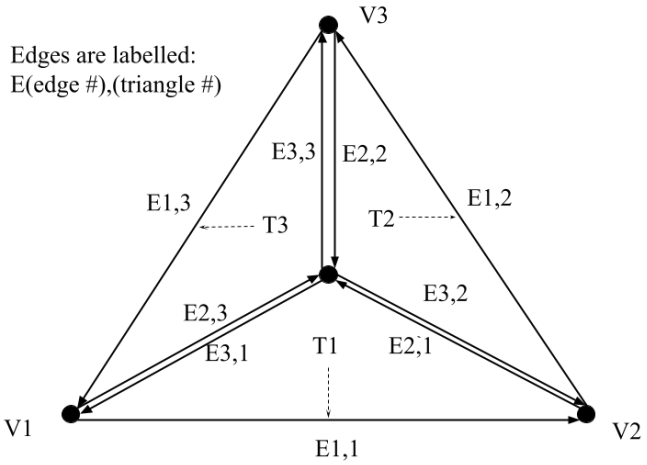
The first of these was finding which triangle a point is in so that we can determine which points it will triangulate with. If we were to naively go through and check each triangle to see if it lies in there or not this would be a linear amount of work for each insertion, which over the entire runtime would result in a $O(n^2)$ runtime from that alone. Instead we want something that would allow us to in constant time find the triangle a given point lies in. This is where we can use the point to triangle map which maps uninserted points to the triangle that they are in, using this we can find out which triangle a point lies in in O(1) amortized time since our map uses hashing (1). This is great but then we come to another issue of keeping these up to date after each insertion, if we were to check every point and recalculate which triangle it was in this would again result in a linear amount of operations which would again result in a $O(n^2)$ runtime. To address this issue I also implemented a map from triangles to points, which maps each triangle to the uninserted points that lie within it. What that allows us to do is that each time a new point is inserted we know that the triangle it is in will get divided into three triangles and thus each uninserted point will need to get remapped to one of the three new triangles. When a point is inserted and a triangle is divided into three triangles, that initial triangle needs to be removed from the map, and the points that map to it must be appended to one of the three new triangles. Since there are only three triangles that each point can be mapped to, each point is mapped to a new triangle in constant time. So the total cost of inserting a point in a triangle is linear in the number of points in that initial triangle instead of linear in the size of the graph. The combination of these two maps means that we are able to accurately find in constant time which triangle a point lies in and have this be always up to date.

The next issues have to do with the underlying data structure of the DCEL, these being the orientation of the new faces as well as making sure that half-edges are correctly storing their twin edges. With the DCEL I needed to make sure that adjacent faces would be oriented such that their shared edge would have the half-edges going in opposite directions. The first step to be

able to determine insertion would be to determine how I wanted the initial triangle to be oriented before a point was inserted into it. The final orientation I decided on was the following:



This orientation is relative to this face T, where its outer component pointer points to edge 1 and all the edges continue in a counter clockwise order. Having this orientation predefined is crucial for being able to be efficient in the rest of the construction as it means we are able to easily find the important edges and points we are dealing with. Now that we have this orientation defined for pre insertion we now need to consider after we insert a point in the middle and the triangle is divided:



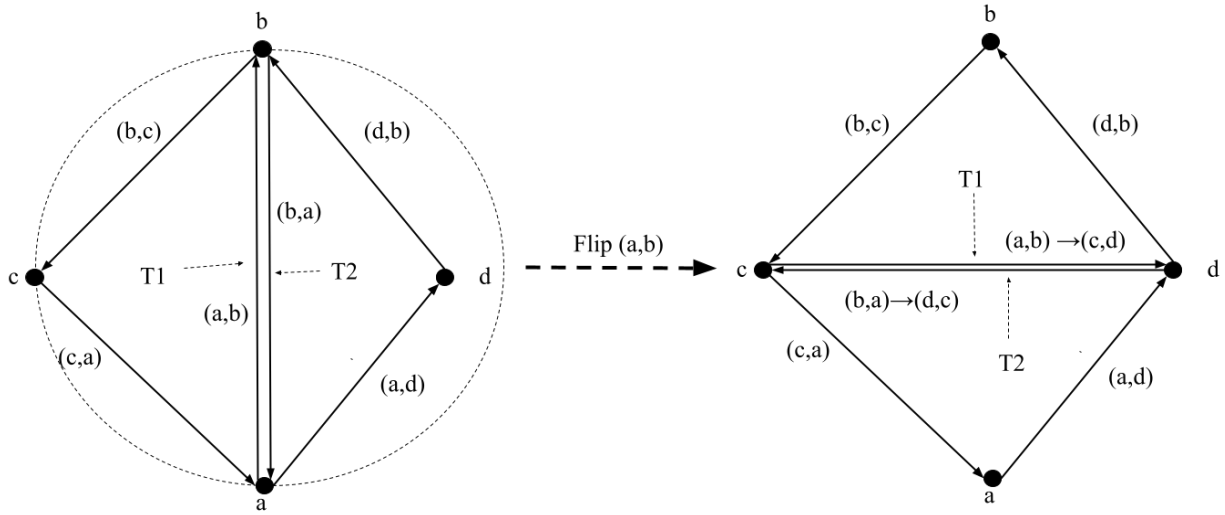
Inserting a point into the initial triangle, T creates three resulting triangles T1, T2, T3. The edges of the resulting triangle Tn are denoted E1,n, E2,n, E3,n such that the edge E1,n is the edge of Tn from the initial triangle T. Each resulting triangle Tn stores a pointer to E1,n. The shared edges of each of the faces have half-edges going in opposite directions while still maintaining that counter clockwise orientation of each of the faces outer edges. In the implementation once we have found our point and the triangle it is in we can create three new triangles with the edges oriented as seen in the figure. In order to determine E1,n for Tn, this is where storing the outgoing edges of each vertex became useful. We just need to find the preexisting edge between

two of the vertices in our new triangle and that will serve as our edge 1. Since our triangulation is planar we guarantee that the average degree of the vertices must be less than six by Euler's Formula which means that this will take constant amortized time for each search. There is still the matter of relating adjacent faces to each other which brings us to the next issue presented by the DCEL which is having the twin pointers of each half edge be correct. This is something that we can do easily using the layout in Figure 3, once we identify which edges in our newly made triangles are which in the diagram we can set the correct twins (ie. E2,1 to E3,2, E3,1 to E2,3 and E2,2 to E3,3). This same division of the triangles can be applied recursively for subsequent point insertions. Once this is completed our re-triangulation with our new vertex is now complete but there is still more that we need to consider.

Edge Flips:

The next major challenge that I faced was figuring out how to implement the edge flips. For each of the edges of each of the newly created triangles we needed to check if they need to be flipped or not. The idea was that each edge e has a twin f and f's incident face is made up of three vertices two of which are adjacent to e, so we just need to check if that third vertex on f's incident face is in the circumcircle of e's incident face. If this vertex wasn't inside the circumcircle then none of the other vertices in that direction would be. Then this process is repeated for all of the newly created half-edges which means that we are doing nine of these checks per insertion. To determine whether a point of interest is inside of the circumcircle of a triangle, also known as the incircle check, we can calculate this mathematically. If our incircle check indicates that the point is inside we would flip the edge we are currently considering. And recursively check if the incircle property holds for all of the edges of the newly created triangles.

After outlining how the edge flipping would work there were some key issues that needed to be addressed with the orientation and updating of the new triangles. To consider what needed to be updated in the new orientation I drew a generalization of the structure before and after an edge flip:



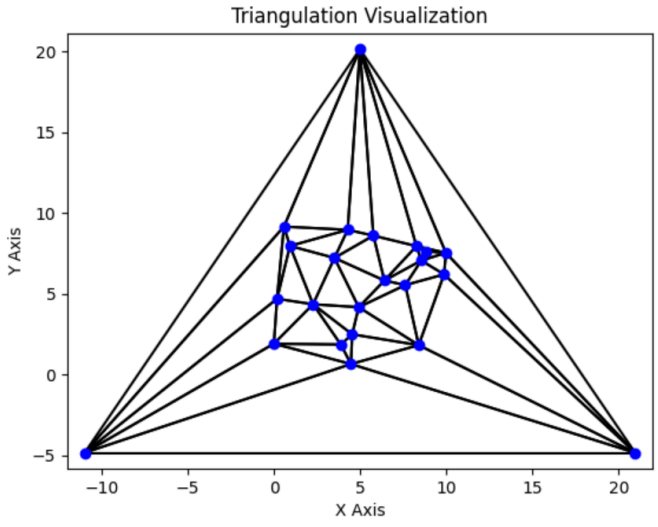
In this diagram we can see that when evaluating the incircle test with respect to the edge (a,b) we find that point d falls in the circumcircle so we need to flip this edge. As we can see this process is fairly involved as we need to update the origins of our flipped edges, the next and previous edges of all the edges in this quadrilateral, and we need to update the mappings between points and triangles that were affected by this edge flip. Once we flip an edge we have now created two new triangles who could possibly violate the Delaunay triangulation property as well. To handle this possibility, we check the edges of the two newly created triangles to see if any of them need to be flipped and this process occurs repeatedly until no edge flips are made. After all of these checks are done, we now have a triangulation that satisfies the Delaunay properties.

Putting all of these main solutions together we have a fully functional incremental Delaunay triangulation algorithm implementation.

Visualization Details

For the visualization I used matplotlib to plot the points and edges of the triangulation and tkinter to create the graphical user interface. The visualization program allows users to input points by clicking on the user interface and they can either submit them in the order that they were added to the screen or in a randomized order. Users can also have the application randomly generate a specified number of points. Once the points are submitted a visualization window for the triangulation appears and shows the inserted and uninserted data points, the lines of the triangulation and the circumcircles of the triangles when doing the incircle test. The purpose of the visualization is to not only show visually how a Delaunay Triangulation would look on a given point set, but also to showcase the details of how the retriangulation occurs with each point inserted and how Delaunay properties are maintained through the incircle checks. There are many interactive visualization tools for data structures such as heaps and binary trees that serve as learning tools for gaining a deeper understanding of how the data structures maintain their properties (2). In a similar way, the original intention for this part of the project was to serve as an interactive learning tool.

Originally, I wanted to create a separate class for the visualization that would have a triangulation attribute and would update its display after calling the next step in the incremental construction. This however had a couple of issues, the first being that this did not show edge flips, just the result after the flips had occurred, and with that it also did not show the incircle test for each of the inserted edges. Instead I had to create a modified triangulation class that had a visualisation attribute, this way every time the triangulation changed something



that could be updated visually it could just make a call to update the display window. A sample of the visualization of a triangulation on 20 points can be seen to the right.

Runtime Testing and Randomization

An element of the incremental Delaunay triangulation algorithm that I have only touched on briefly is the runtime. The algorithm has a worst case runtime of $\Theta(n^2)$ which results from the possibility of a chain reaction of edge flips. When a point is inserted it is possible that this insertion results in an edge flip that results in another edge flip and this can continue to affect a linear number of edges in the graph. This would make the runtime of each insertion O(n) and over the insertion of all of the n points this would make the worst case runtime of $\Theta(n^2)$. There is a theorem that the number of edge flips after each insertion is on average constant over all possible insertion orders and that the number of times that an uninserted point is reallocated to a new triangle is O(logn) expected (3). From this we can see that in a random insertion order we have an expected runtime of O(logn) for each of the n insertions and thus have a total expected runtime of O(nlogn).

In this section of the project I wanted to explore the runtimes of my implementation for different distributions of points and how these runtimes are affected by the randomization of the insertion order. First, we need to outline the metrics for what determines the complexity using what we have already discussed. The three main metrics are the raw runtime, the number of edge flips and the number of times that points are re-bucketed (ie. remapped in our implementation) during the course of the runtime. The runtime is useful for comparing two completely different distributions of n points and the edge flips and rebucketing is useful for comparing two different insertion orders of the same point set. The results that we expect to see according to the literature that we discussed is that the randomization will improve the runtime of poorly distributed point sets.

It's worth noting that these runtimes are subject to the local environment that the program is being run in and these are all running off of the regular Triangulation program which is completely separate from the visualization program. I had originally intended to visualize these test results into graphs but as you will see the differences in efficiency are so large that they do not lend themselves to an interesting visualization of data for some of the metrics.

Benchmark: Randomly Generated Point Sets

As a benchmark for what an average runtime should be for different distributions and numbers of points, I started with generating 100 random point sets of the different sizes and triangulating them. I recorded the average runtimes, average number of edge flips and the average number of rebucketing steps that occurred. All of which can be seen here:

# of Points	Avg Runtime (sec)	Avg Edge Flips	Avg Rebuckets
10	0.0004	11	25
100	0.0074	250	881
500	0.0594	1,412	7,842
1000	0.1646	2,919	16,943
2500	0.7178	7,370	55,239
5000	2.4751	14,630	123,973

Table 1: Measurements for randomly generated points

For testing the effectiveness of randomization I wanted to generate point sets with distributions and orders that I hypothesized would give bad runtimes. There is no mathematical basis that I had in mind when creating these cases, I just observed that geometrically uniform points tended to have poor runtimes when they were inserted in order. For all of these cases I tested with insertion of the points in the chronological order, then in randomized order. For all of these cases I ran them 100 times at each point set size and took the average runtime, edge flips and re-bucketing operations. Then for each of these cases I also ran the triangulation with randomization of the order, again with 100 runs per number of points and recorded the same metrics. The 100 runs of each case allows us to have some faith that the results are somewhat representative of the average results over an infinitely large number of runs.

Case 1: Spiral

For this case I generated points that were arranged in a spiral configuration, for the in order insertion, the results were as follows:

# of Points	Avg Runtime (secs)	Avg Edge Flips	Avg Rebuckets
10	0.0008	17	26
100	0.0131	294	1,531
500	0.3236	5,794	36,488
1000	1.7344	22,280	143,805
2500	20.17	135,750	897,184
5000	160.8997	538,817	3,577,621

Table 2: Measurements for spiral points in order

When we compare this to the benchmark or randomly distributed points we can see a massive increase in the runtime of over 6000% for the 5000 point case. From this we can see that this distribution and order is definitely worse, but as we mentioned before these variations in runtimes are largely based on two things which are the edge flips and the re-bucketing of points. When we look at these two columns for the 5000 points case, we can see where our huge runtime comes from as the spiral distribution and order results in over 3 million re-bucket operations and over 500 thousand of edge flips, whereas the benchmark case only has 123 thousand re-bucket operations and 14 thousand edge flips. As we can see there are massive disparities in the number

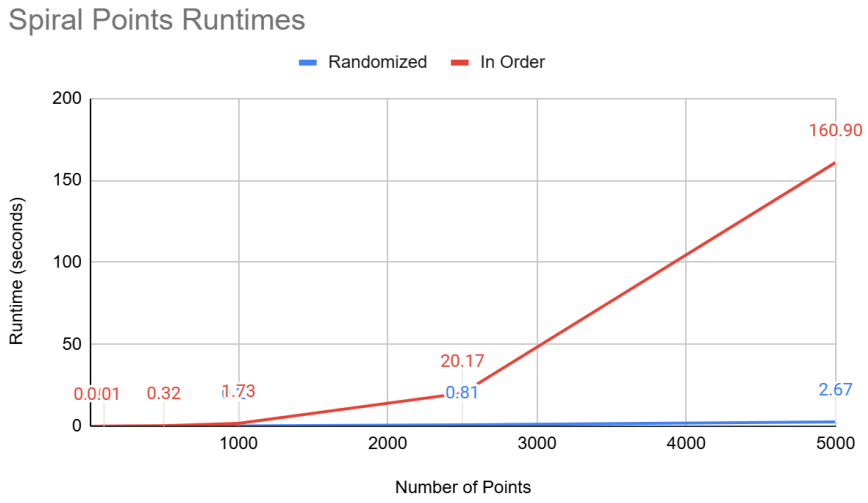
of operations between these two cases despite having the same amount of points. Now we will see if randomization of the order can help us triangulate these same points more efficiently. For the randomized order the results were as follows:

# of Points	Avg Runtime (secs)	Avg Edge Flips	Avg Rebuckets
10	0.0005	14	29
100	0.0105	253	986
500	0.0765	1,365	7,824
1000	0.1992	2,768	17,878
2500	0.8124	6,982	53,533
5000	2.6734	14,074	120,306

Table 3: Measurements for spiral points randomized

If we compare these results to the in order triangulation for the 5000 point case we can see that on average we are able to compute the triangulation in less than 2% of the time, with less than 5% of the edge flip and re-bucketing operations which is an insane improvement. If we take it a step further and compare the results for this randomized order to the benchmark of randomly generated points we can see that we are performing similarly to this case despite how poorly this point distribution performed when we did the insertion in order.

We can see with this graph how much the runtime is improved with randomization:



Case 2: Linear

Now for our next case we will be triangulating points along a diagonal line and inserting them from left to right to see how this distribution and order performs. Here are the results from the in order insertion.

# of Points	Avg Runtime (secs)	Avg Edge Flips	Avg Rebuckets
10	0.0005	4	45
100	0.0166	45	4,950
500	0.5485	228	124,750
1000	3.3363	457	499,500
2500	42.36	1,144	3,123,750
5000	319.29	2,288	12,497,500

Table 4: Measurements for linear points in order

From our runtimes we can see that it takes a massive amount of time to perform the triangulation for this order when comparing it to the benchmark case. For the 5000 points case it is taking over 300 seconds to triangulate whereas the benchmark took only 2 and a half, even comparing it to the spiral distribution in order, which already performed poorly, the runtime is almost doubled. If we compare the edge flips and re-bucketing operations of this case with the benchmark we can see something very interesting in that the number of edge flips here is actually less than in the random distribution. That however is not the only factor as play here as if we look at the number of re-bucketing operations we can see that for 5000 points we perform over 12 million re-bucketing operations which is 100 times more than we do for the random points and more than 3 times more than in the spiral distribution. Since we are inserting the points in a line, all the other points following the current one in the line will need to get re-bucketed. This means that on the first insertion we need to re-bucket 4999 points, then 4998 on the second and so on until all points are inserted. This means that in total we expect around $\Sigma^{5000}_{n=1} n \approx 12,500,000$ re-bucketing operations, which is consistent with the results we got. Once again we will see if the randomization can increase the performance for this point distribution. The results for the randomized order of insertion are as follows:

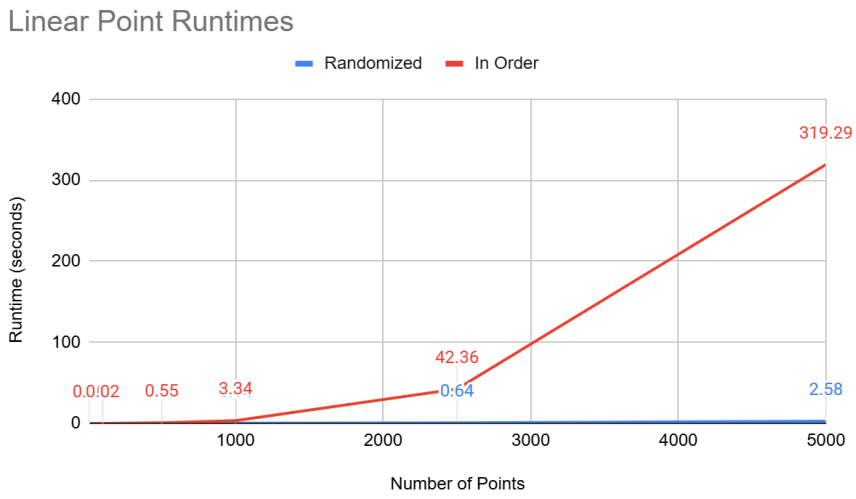
# of Points	Avg Runtime (secs)	Avg Edge Flips	Avg Rebuckets
10	0.0006	7	28
100	0.007	95	779
500	0.0525	494	5,875
1000	0.1377	992	13,543
2500	0.6431	2,492	39,383
5000	2.578	4,991	88,649

Table 5: Measurements for spiral points in order

As we can see here, and as expected, the randomization of the order does improve the runtime very significantly to the point that it is around the same as the benchmark. For 5000 points, the number of re-bucketing operations which was huge in our in-order insertion has not only dropped, but it is now on average below the benchmark in terms of operations. As we saw with the in order insertions the number of edge flips was very low relative to the runtime, with the randomization we still have a relatively low number of edge flips compared to the benchmark

case. We were able to keep the advantage of this point distribution, which was the low number of edge flips, while getting rid of the disadvantages, which was the high number of re-bucketing operations, all through simply randomizing the order of our insertion.

We can see with this graph how much the runtime is improved with randomization:



Case 3: Circle

The final point distribution that we will look at is points distributed around the outside of a circle. For the in order insertion of the points the results were:

# of Points	Avg Runtime (secs)	Avg Edge Flips	Avg Rebuckets
10	0.0006	11	36
100	0.0162	148	3,257
500	0.4548	749	81,051
1000	2.5535	1,508	324,104
2500	32.4755	3,756	2,024,020
5000	235.1983	7,446	8,095,542

Table 6: Measurements for circle points in order

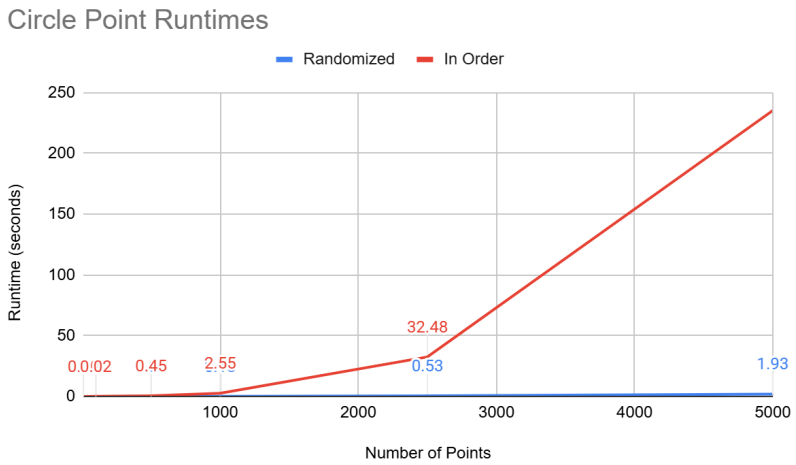
This case was similar to the linear case in that as the number of points increased we saw less edge flips than the benchmark, but we also saw a huge increase in the number of re-bucketing operations. The runtimes here were in between the spiral and linear cases and we can see that this can be attributed to having way more re-bucket operations than the spiral case but a significant amount less than the linear case. Again, the runtimes and number of operations show us that this is indeed a very bad distribution and order for the insertion of points. Once we randomized the order of the points the results were:

# of Points	Avg Runtime (secs)	Avg Edge Flips	Avg Rebuckets
10	0.0004	8	26
100	0.0073	103	703
500	0.0498	541	5,081
1000	0.1272	1,083	11,639
2500	0.5316	2,723	33,651
5000	1.9296	5,444	73,858

Table 7: Measurements for circle points randomized

For the randomization here we can see that this actually gives us the best runtimes we have seen thus far with it taking half a second less than the benchmark for 5000 points. We can see that the number of edge flips and re-bucketing operations here are close to half of that of the benchmark case which is what is causing our runtimes to be better than the other distributions. The randomization again allows us to improve on both the number of edge flips and the number of re-bucketing operations that we perform which results in a vastly increased runtime.

We can see with this graph how much the runtime is improved with randomization:



Conclusion

While these test cases are not exhaustive and there are many possible distributions and orders that could cause issues in the runtime efficiency of the triangulation we have seen here at least three examples where the randomization of the insertion order results in massive improvements in the runtime of the Delaunay triangulation algorithm.

Source Code

All of the source code for the triangulation, the run time testing, and the visualization, as well as a standalone version of the visualization application can be found here:
https://github.com/justinbacic/COMP-4202-Project

Triangulation.py: This file contains the triangulation implementation as used for the runtime testing

VisualizationVersionOfTriangulation.py: This file contains the triangulation implementation that includes the user interface

RuntimeTesting.py: This file contains the code for initializing and evaluating the test cases outlined in the runtime section

Delaunay Triangulation Visualization.exe: This is a standalone application that can be used to see in action the visualization application

References:

(1) Luna, Javier Canales. “A Guide to Python Hashmaps.” *DataCamp*, DataCamp, 3 Dec. 2024, www.datacamp.com/tutorial/guide-to-python-hashmaps.

(2) “Min Heap.” *Heap Visualization*, www.cs.usfca.edu/~galles/visualization/Heap.html.

(3) Sack, Jorg. “DT - Details” *COMP 4202,* Carleton University

(4) Sack, Jorg. “overlay - Copy” *COMP 4202,* Carleton University