Experiments on Spanning Ratios of Minimum Weighted Triangulations of Convex Polygons

Gautam Singh

Computational Geometry, Graph Theory
Indian Institute of Science, Education and Research, Kolkata

Spanning Ratios of well known graphs have been studied thoroughly and are well understood. However spanning ratios of Minimum Weighted Triangulations have not been explored. Here we will experiment with the spanning ratios of the **Minimum Weighted Triangulations** of Convex Polygons. We have written a program to compute the triangulation of an arbitrary convex n-gon and it's resulting spanning ratio. The code can be found at the following repository: spanning-factor.

Contents

1. Introduction

Triangulation is a very common mathematical problem and triangulating a geometric object is one of the basic problems of Computational Geometry and is applied in various areas like terrain visualisation[DG02], mesh generation, numeric simulations and optimisations. It involves decomposing a geometrical object into simpler elements known as *triangulating elements*. These elements can be triangles in $\mathbb{R}^2$, tetrahedrons in $\mathbb{R}^3$, and *simplices* (the simplest possible polytopes) in higher dimensions.

Although this problem is `NP-HARD` for point-set inputs, it can be calculated exactly in polynomial time for polygon inputs. A simple polygon,i.e, a polygon that does not intersect itself, can be optimally triangulated (by optimizing some cost function for each of the triangulating elements) in $\mathcal{O}(n^3)$ time using dynamic programming. For our purposes we will only be considering points that lie on a convex boundary.

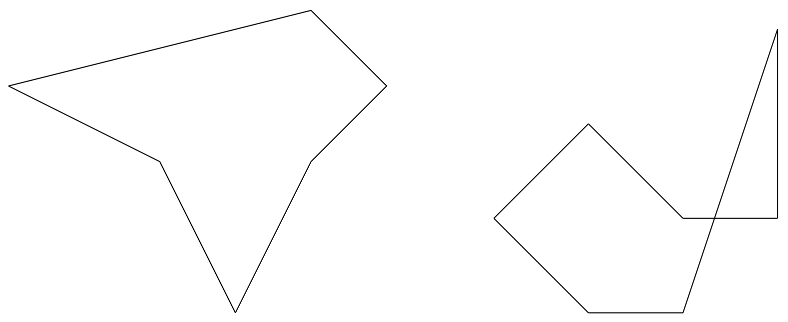
In the following sections we will try to deal with the following problem.

Problem. Given a convex polygon $\mathcal{P}$ find it's minimum weight triangulation $\mathcal{T}$ and compute the spanning ratio of the minimum weight triangulation.

For our problem we used a very well established dynamic programming algorithm to generate triangulations with minimum weights. We will give an overview of the algorithm in Section 2. The generated triangulation can be treated as a connected and undirected Euclidean graph. For finding the *spanning ratio* of the graph we first solved the `AllPairsShortestPath` problem to find the shortest path between all vertices of the graph. Computing the spanning ratio is relatively straight forward after solving the AllPairs problem.

1.1. Terminology

In this section we define some simple notation and definitions for clarity. Henceforth, all mentioned convex polygons would be *simple*, i.e, they do not self-intersect.



(a) Simple Polygon (b) Non-Simple Polygon

Figure 1: Types of Polygons

Notation (Simple Polygon)**.** We can represent a polygon $\mathcal{P}$ as a list (order of elements matters) of vertices listed in clockwise or counter-clockwise order as

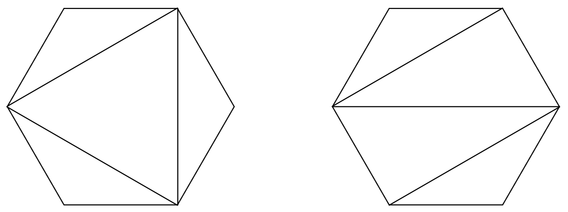
$$\mathcal{P} = (v_1, v_2, \dots, v_n).$$

The set of the edges of a polygon $\mathcal{P}$ can be defined as

$$E := \{(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n), (v_n, v_0)\}.$$

2. Minimum Weighted Triangulation

Problem. Given a convex polygon $\mathcal{P}$ find a set of pairwise non-intersecting triangles whose combined perimeter is minimal and their union gives $\mathcal{P}$.



(a) Optimal Triangulation (b) Non-Optimal Triangulation

Figure 2: Different Triangulations of Polygons

As we can see in the problem we must minimise the perimeter of the generated triangles. Let's define a *perimeter* function $P : \mathcal{P} \times \mathcal{P} \times P \to \mathbb{R}$ such that

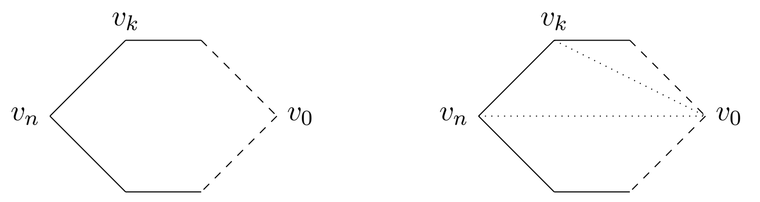
$$P(v_i, v_j, v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|.$$

For the optimal triangulation of the polygon we must minimise the function P over the generated triangulations of the polygon. Let's take a vertex v_k from the polygon $\mathcal{P}$. Now we have subdivided polygon $\mathcal{P}$ into the triangle (v_0, v_k, v_n) and two subpolygons to the left and the right of the triangle.

We must check all possible subdivisions to minimise our perimeter function. We can do this recursively, for which a dynamic programming approach can be used. Let $T(v_i, v_j)$ be the cost of triangulation of the polygon $(v_i, v_{i+1}, \dots, v_j)$. We can observe the following recursive function:

$$T(v_i, v_j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} T(v_i, v_k) + T(v_{k+1}, v_j) + P(v_i, v_k, v_j) & \text{if } i < j. \end{cases}$$

Thus $T(v_1, v_n)$ is the cost of optimal triangulation of the polygon that we need to compute. Here we can initialise two $n \times n$ array `memo` and `k_store` which are two



(a) Polygon we need to triangulate (b) Subdivision of the Polygon

Figure 3: Running an iteration of the algorithm

memoisation arrays where we can store the values of $T(v_i, v_j)$ at `memo[i][j]` and store the value of k at `k_store[i][j]` at each k. The pseudocode for the algorithm is given at Algorithm 1.

Array `memo[0][n]` only stores the value of $T(v_0, v_n)$ which is the minimum cost of triangulating the polygon. However we can also get the explicit triangulation of the polygon with Algorithm 2. Let n be the size of the polygon array `P`.This takes $\mathcal{O}(n^2)$ memory due to the memoisation arrays and the time complexity is of the order $\mathcal{O}(n^3)$ because we do n things on an array of the order of n^2 [Jim98].

Algorithm 2 is the iterative form of a recursive function, which is why a stack is used in it's implementation. When we call the `Triangulation` function to calculate the triangulation of the polygon $P = (v_0, v_1, \ldots, v_n)$ it traverses the array `k_store` and assigns triples of the index of the `P` (the polygon array) to the array `triangle`. We can get the coordinates of each vertex of the triangulating elements by accessing the `P` at each element of `triangle`.

Example To see the Algorithm in action we can run a sample polygon through it. We can see the polygon in the figure below:

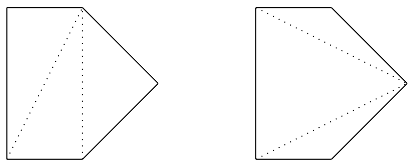


Figure 4: Minimum Weight Triangulation

This will generate a Minimum Weight Tree (Figure 5) from which we can compute the optimal triangulation.

3. The Spanning Ratio of an Euclidean Graph

Definition 3.1 (Euclidean Graphs)**.** Let $G(V, E)$ be a graph with $V \subseteq \mathbb{R}^n$ and E as the set of it's edges. The length of any edge $(p, q) \in E$ for $p, q \in V$ is defined as the Euclidean distance between p and q denoted as $|pq|$. Such graphs are called *Euclidean Graphs.*

Algorithm 1 MinCostDP

```
1: procedure MinCostDP(P ← Array of polygon coordinates)
2:     n ← Length of $\mathcal{P}$
3:     memo_t[n][n] ← $-1$
4:     k_store[n][n] ← $-1$
5:     for gap = 0 to n - 1 do
6:         $i \leftarrow 0$
7:         for j = gap to $n - 1$ do
8:             if j ≥ i + 2 then
9:                 memo_t[i][j] ← 0
10:            else
11:                memo_t[i][j] ← $\infty$
12:                for k = i + 1 to j do
13:                    p ← Perimeter of the triangle (P[i], P[j], P[k])
14:                    cost ← memo_t[i][k] + memo_t[k][j] + p
15:                    if memo_t[i][j] > cost then
16:                        memo_t[i][j] ← cost
17:                        k_store[i][j] ← k
18:            $i \leftarrow i + 1$
19:    return memo_t[0][n - 1], k_store
```

Algorithm 2 Triangulation

```
1: procedure Triangulation(k_store ← Memoisation array of MinCostDP)
2:     triangle ← Positions of the triangulating elements
3:     S ← A stack of pairs
4:     S ← {0, n - 1}
5:     while S is not empty do
6:         {i, j} ← s.top
7:         S.pop
8:         if abs(i - j) > 1 then
9:             k ← k_store[i][j]
10:            triangle ← $\{i, j, k\}$
11:            S ← $i, k$
12:            S ← $k, j$
13:    return triangle
```

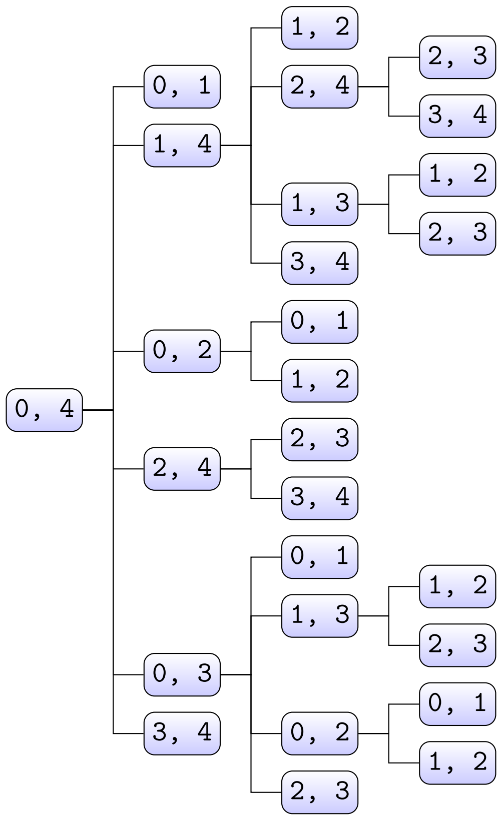


Figure 5: Minimum Weight Tree

Henceforth, all graphs considered will be undirected and connected Euclidean graphs.

Definition 3.2. Let V be a set of n points in $\mathbb{R}^n$ and $G(V,E)$ be a graph with V as it's vertices and E as the set of it's edges. We define, $g : V \times V \to \mathbb{R}$ as the *shortest path function* and

$$g(p,q) = \begin{cases} 0 & \text{if } p = q \\ \text{shortest path from } p \text{ to } q & \text{otherwise.} \end{cases}$$

Definition 3.3 (t-spanners)**.** Let $t > 1$ be a real number. Let $G(V,E)$ be a graph. We say that G is a t-spanner for V, if for each pair of points $p, q \in V$ we have $g(p,q) \leq t \cdot |pq|$. Thus, if G is t-spanner for the set of points in V then for points $p, q \in V$ there exists a path in G at most t times the Euclidean distance between p and q.

Definition 3.4 (Stretch Factor)**.** Let $G(V,E)$ be a graph. The minimum value of t for which G is t-spanner is called the *stretch factor* or *spanning ratio* of G. We will denote the stretch factor as t_s.

Note. We can observe that

$$t_s = \max\left\{ \frac{g(p,q)}{|pq|} \mid p, q \in V \right\}$$

3.1. Computing the Spanning Ratio

Problem (All Pairs Shortest Path)**.** Let $G(V, E)$ be a graph. Find the shortest path from the vertex p to the vertex q in the graph for all possible pairs $p, q \in V$.

As we can clearly see, to compute the stretch factor of a given graph we must first solve the *All Pairs Shortest Path* (APSP) problem. There are two algorithms in particular that we can use to solve the APSP problem, namely

- *Floyd-Warshall Algorithm*
- *Johnson's Algorithm*

In our implementation we have used *Floyd-Warshall* algorithm. We shall see a brief overview of how we have used it to compute the spanning ratio.

3.1.1. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is based on the dynamic programming paradigm. Unlike Dijkstra's algorithm which finds the shortest path from a particular node p (say) to another node q (say) in the graph G, it is supposed to find the shortest path from pairs of nodes p, q in the graph G. Let $G(V, E)$ be a graph where $V = \{v_1, v_2, \dots, v_n\}$ be the vertices of the graph. Let $w : V \times V \to \mathbb{R}$ be the *weight function* and

$$w(p,q) = \begin{cases} |pq| & \text{if } (p,q) \in E \\ \infty & \text{if } (p,q) \notin E. \end{cases}$$

Let $s : V \times V \times \mathbb{N}_n \to \mathbb{R}$ be the *shortest intermediate path* function such that $s(v_i, v_j, k)$ be the weight of the shortest path from v_i to v_j through the vertices $\{v_1, v_2, \dots, v_k\}$. Thus we can observe that $s(v_i, v_j, n)$ is the shortest path from v_i to v_j that we can find through the graph. We will recursively compute the value of $s(v_i, v_j, n)$. Let's first note that

$$s(v_i, v_j, k) \leq s(v_i, v_j, k-1).$$

Suppose $k > 0$. If $s(v_i, v_j, k)$ is strictly less than $s(v_i, v_j, k-1)$ then there exists a path from v_i to v_j through the vertices $\{v_1, v_2, \dots, v_k\}$ that is shorter than any path that does not use the vertex v_k. Since the path passes through v_k there exists a subpath of weight $s(v_i, v_k, k-1)$ and another subpath of weight $s(v_k, v_j, k-1)$. If $s(v_i, v_j, k) = s(v_i, v_j, k)$ it means that the shortest path from v_i to v_j does not pass through the vertex v_k.

This implies the following recursive function:

$$s(v_i, v_j, k) = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\{s(v_i, v_j, k-1), s(v_i, v_k, k-1) + s(v_k, v_j, k-1)\} & \text{if } k > 0. \end{cases}$$

Our goal is to compute $s(v_i, v_j, n)$ for all pairs $v_j, v_k \in V$. The pseudocode for the algorithm is given as:

Algorithm 3 Floyd-Warshall

```
1: procedure FloydWarshall(V ← Set of vertices, E ← Set of edges, w ← Weight function)
2:     n ← Size of V
3:     ShortestDistance ← n × n Array initiated with $\infty$
4:     for Each edge $(v_i, v_j)$ in E do
5:         ShortestDistance[i][j] ← w$(v_i, v_j)$
6:     for Each vertex $(v_i)$ in V do
7:         ShortestDistance[i][i] ← 0
8:     for i = 0 to $n - 1$ do
9:         for j = 0 to $n - 1$ do
10:            for k = 0 to $n - 1$ do
11:                cost ← ShortestDistance[j][i] + ShortestDistance[i][k]
12:                ShortestDistance[j][k] = min(ShortestDistance[j][k], cost)
13:    return ShortestDistance
```

Let n be the size of the vertex arrays. This takes $\mathcal{O}(n^2)$ memory due to the `ShortestDistance` array. We compute n^2 values of `ShortestDistance[j][k]` for each pair `(j, k)` for each intermediate vertex `i`. Thus this takes $\mathcal{O}(n^3)$ time.

Example We will illustrate the algorithm for a sample graph as follows:

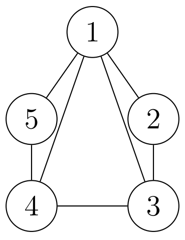


Figure 6: Euclidean Graph of a Corresponding Pentagon

Now we can see as to how the distance matrix changes during each iteration in. This is the Euclidean Graph of a triangulated regular pentagon with unit sides. The length of the diagonals is 1.6.

$$\begin{bmatrix} 0 & 1 & 1.6 & 1.6 & 1 \\ 1 & 0 & 1 & \infty & \infty \\ 1.6 & 1 & 0 & 1 & \infty \\ 1.6 & \infty & 1 & 0 & 1 \\ 1 & \infty & \infty & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1.6 & 1.6 & 1 \\ 1 & 0 & 1 & 2.6 & 2 \\ 1.6 & 1 & 0 & 1 & 2.6 \\ 1.6 & 2.6 & 1 & 0 & 1 \\ 1 & 2 & 2.6 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1.6 & 1.6 & 1 \\ 1 & 0 & 1 & 2.6 & 2 \\ 1.6 & 1 & 0 & 1 & 2.6 \\ 1.6 & 2.6 & 1 & 0 & 1 \\ 1 & 2 & 2.6 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1.6 & 1.6 & 1 \\ 1 & 0 & 1 & 2.6 & 2 \\ 1.6 & 1 & 0 & 1 & 2.6 \\ 1.6 & 2.6 & 1 & 0 & 1 \\ 1 & 2 & 2.6 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1.6 & 1.6 & 1 \\ 1 & 0 & 1 & 2.6 & 2 \\ 1.6 & 1 & 0 & 1 & 2.6 \\ 1.6 & 2.6 & 1 & 0 & 1 \\ 1 & 2 & 2.6 & 1 & 0 \end{bmatrix}$$

Final Calculation The Floyd Warshall algorithm gives us a $n \times n$ matrix A such that a_{ij} is the shortest path between the vertices v_i and v_j. Thus we observe that

$$t_s = \max\left\{ \frac{a_{ij}}{|v_i v_j|} \mid A = [a_{ij}] \right\}.$$

We can compute t_s for a given graph using the following algorithm:

Algorithm 4 SpanningFactor

```
1: procedure SpanningFactor(ShortestDistance ← Output of FloydWarshall,
   n ← Size of vertices of the graph)
2:     SpanningFactor ← $\infty$
3:     for i = 0 to $n-1$ do
4:         for j = 0 to $n-1$ do
5:             if i = j then
6:                 continue
7:             dist ← Euclidean distance between the vertices $(v_i, v_j)$
8:             SpanningFactor ← min(SpanningFactor, SpanningFactor/dist)
9:     return SpanningFactor
```

4. Experimenting on Spanning Ratios

We have solved the problem of calculating the triangulation of a simple convex polygon and finding the corresponding spanning ratio exactly. We now want to see how spanning ratio of different types of convex polygons. To test these we need to generate several sets of convex polygons.

Problem. Generate a list of coordinates in clockwise or anticlockwise order which form the vertices of a convex polygon.

To solve this problem we first need the help of the following theorem.

Theorem 4.1. *Any non self-intersecting polygon that can be inscribed in a circle, i.e, all vertices of the polygon touch the circle is convex.*

The proof of the theorem can be found at [Wu20]. Suppose we need to generate an n-gon, i.e, an n sided convex polygon. The idea is to first generate n random numbers from 0 to 2π. Then sort the random numbers. We now have a list of sorted numbers from 0 to 2π. These angles now represent a set of points on a circle which have been sorted in clockwise or counter-clockwise order. These numbers can be converted to cartesian coordinates via a simple trigonometric transform.

Let $L = (\theta_1, \theta_2, \dots, \theta_n)$ be the list of sorted angles. Then we can have

$$x_k = r \cos \theta_k \tag{1}$$
$$y_k = r \sin \theta_k \tag{2}$$

for some $r > 0$ and $\theta_k \in L$. Thus we can have list of coordinates of a polygon which by Theorem 4.1 is guaranteed to be convex. The pseudocode for the following algorithm is given as:

```
1: procedure RandomConvex(n ← Number of vertices in the convex polygon, r ← Radius of circle)
2:     Angles ← Empty array of angles
3:     Polygon ← Empty array of points
4:     for i = 0 to $n - 1$ do
5:         Angles ← RandomFloat(0, 2 * pi)
6:     sort(Angles)
7:     for i = 0 to $n - 1$ do
8:         theta ← Angles[i]
9:         x ← r * cos(theta)
10:        y ← r * sin(theta)
11:        Polygon ← {x, y}
12:    return Polygon
```

Example To illustrate the algorithm we will look at a case of a randomly generated quadrilateral. This can be seen in Figure 7.

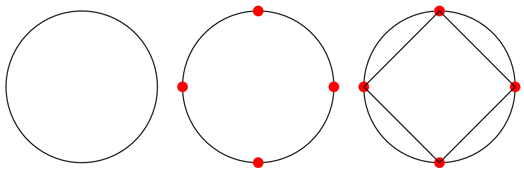


Figure 7: Random Polygon Generation

4.1. Program and Analysis

The `spanning-factor` module is written in `C++17` and compiled with `g++` version: `13.2.1 20230801`. The `create_convex` module was written in `Python3.12.2`. In the `draw_convex` and the `draw_graph` module `matplotlib` version: `3.9.0` was used to visualise the convex polygon and it's corresponding triangulation.

We will now see an overview of each module:

1. `spanning-factor`: We have implemented structs for points, helper functions to calculate Euclidean distances between two points and the perimeter of a triangle. We have also implemented `MinCostDP`, `Floyd-Warshall` and `SpanningFactor` algorithms in the functions `minimum_cost_polygon_dp` and `spanning_ratio` respectively.

2. `create_convex`: Create the required convex polygon. Implementation of the algorithm `RandomConvex`.

3. `draw_convex`: Visualise the generated convex polygon.

4. `draw_graph`: Visualise the triangulated polygon.

We have calculated the spanning ratio of convex polygons generated by the algorithm `RandomConvex` with vertices $n = 4, 5, \dots, 14$ and $r = 2, 6, 8$.

4.2. Results

Regular Polygons One of the first things that come to mind is to explore the spanning ratio of regular polygons. For this experiment, we generate the coordinates of regular polygons from $n = 4$ to $n = 1000$. Here n is the number of sides of our polygon. We discovered that that spanning ratios have no relation to the size of the original circle.

We have plotted the values spanning ratio against n in Figure 8.

Random Polygons We also experimented with different values of r, i.e, the radius of the original circle and n. For a given value of r and n we generated several iterations of polygons and calculated their values. We have plotted these values in Figure 9, Figure 10, Figure 11 and Figure 12.

Normally Distributed Polygons We we also curious about whether if we generate polygons according to a random distribution how the variation in spanning ratios will change. In our experiment we generated 1000 sets of coordinates of polygons with $n = 100$. We then calculated their spanning ratios and plotted them in Figure 13.

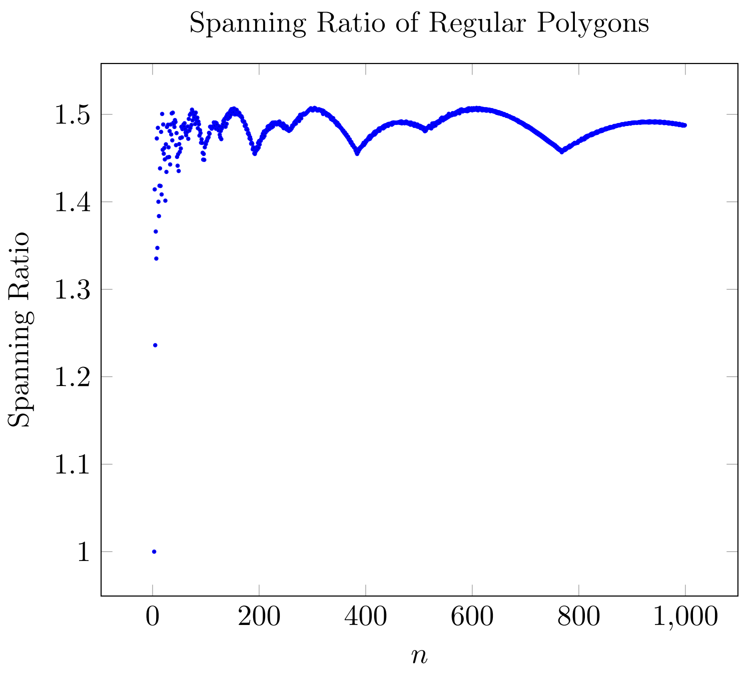


Figure 8: Plot of Regular Stretch Factor with n

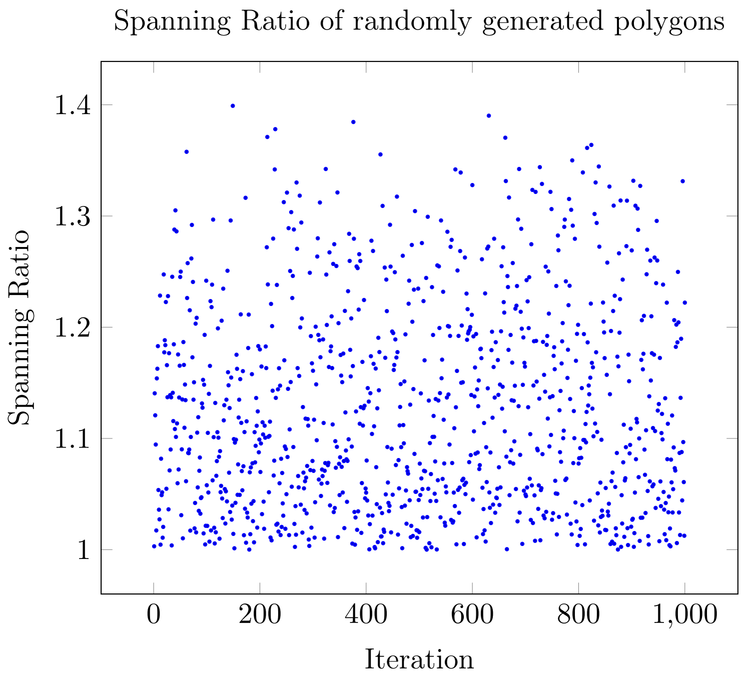


Figure 9: Spanning Ratio of Polygons with $r = 1$ and $n = 4$

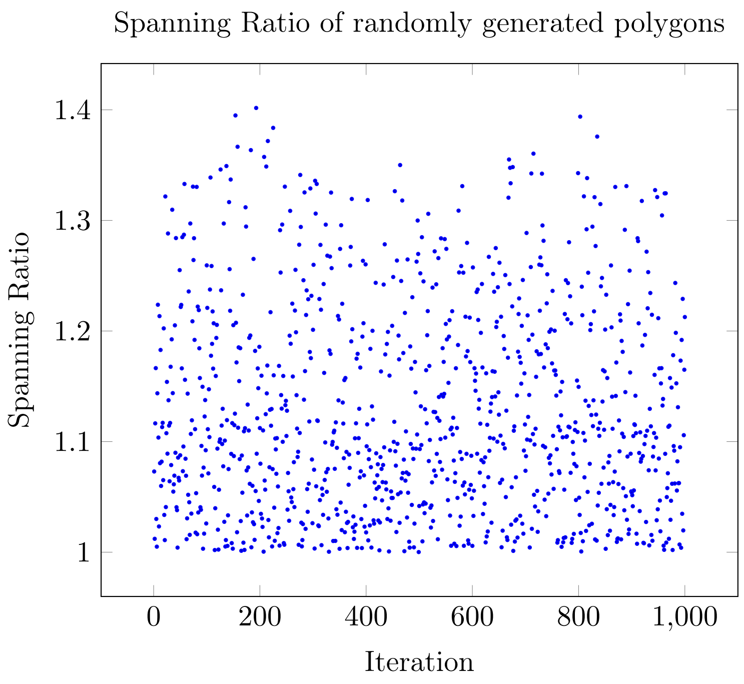


Figure 10: Spanning Ratio of Polygons with $r = 2$ and $n = 4$

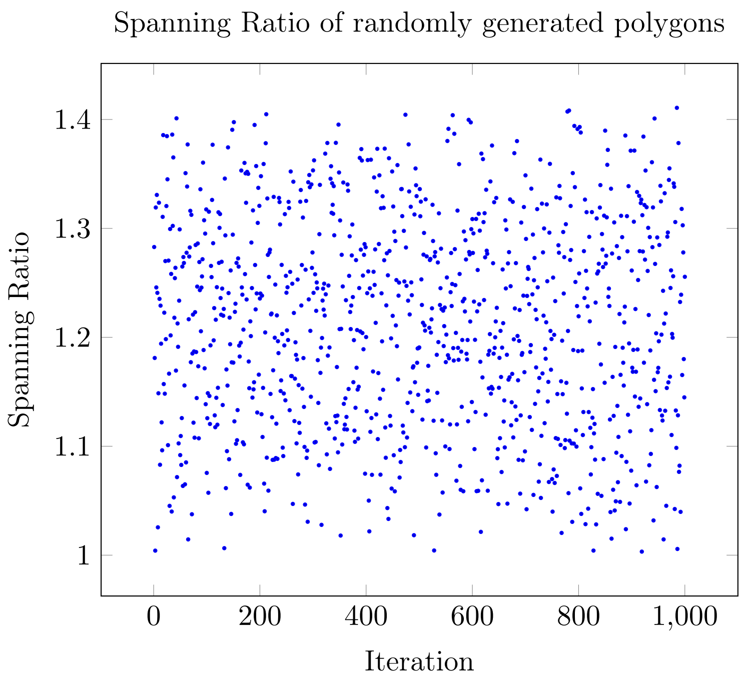


Figure 11: Spanning Ratio of Polygons with $r = 1$ and $n = 5$

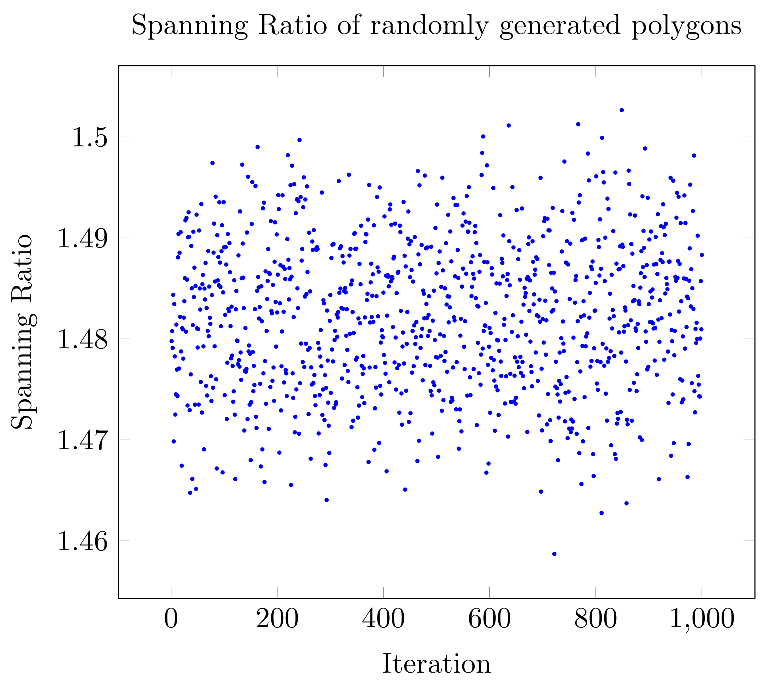


Figure 12: Spanning Ratio of Polygons with $r = 1$ and $n = 100$

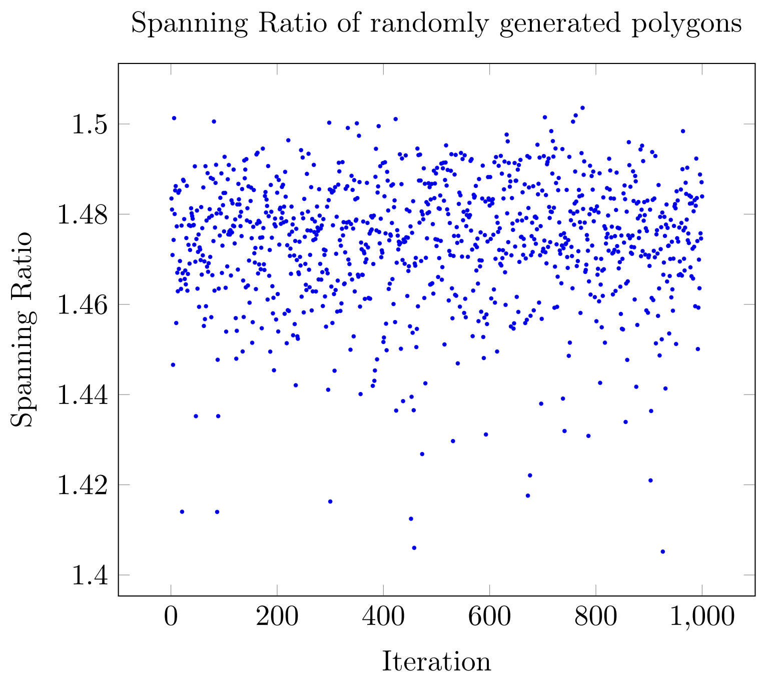


Figure 13: Spanning Ratio of Normally Distributed Polygons with $r = 1$ and $n = 100$

A. Convex Polygon Coordinates

The coordinates of each polygon and it's minimal triangulation can be found in the GitHub repository in the directory `experiment`.

References

[Jim98] Daniel Ángel Jiménez. *Analysis of Algorithms*. `https://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture12.html`. Accessed: 2024-06. 1998.

[DG02] Maciej Dakowicz and Christopher Gold. "Extracting Meaningful Slopes from Terrain Contours". In: *Computational Science — ICCS 2002*. Ed. by Peter M. A. Sloot et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 144–153. ISBN: 978-3-540-47789-1.

[Wu20] H. Wu. *A characterization of regular polygons*. `https://math.berkeley.edu/~wu/RNLE_polygon.pdf`. Accessed: 2024-06. 2020.