Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Monika Wiech

Nr albumu: 1092318

Triangulacja Delaunaya w geometrii obliczeniowej

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Informatyki Stosowanej

Kraków 2020

Oświadczenie autora pracy

Świadoma odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Bardzo serdecznie dziękuję Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za pomoc merytoryczną, analizę przedstawionych rozwiązań oraz zaangażowanie i cierpliwość przy dopracowywaniu kolejnych wersji tej pracy.

Streszczenie

W pracy przedstawiono implementację w języku Python pięciu algorytmów wyznaczających triangulację Delaunaya w geometrii obliczeniowej. Triangulacja Delaunaya jest to taka triangulacja zbioru punktów P na płaszczyźnie, że żaden punkt ze zbioru P nie znajduje się wewnątrz okręgu opisanego na trójkącie należącym do triangulacji. Triangulacja Delaunaya jest używana m.in. do modelowania terenu, automatycznego generowania meshu.

Przygotowane algorytmy triangulacji to algorytm naiwny, metoda odwróceń Lawsona, algorytm inkrementacyjny Bowyera-Watsona, algorytm rekurencyjny Lee-Schachtera oparty na metodzie dziel i zwyciężaj, oraz algorytm quickhull. Algorytmy zostały przetestowane i mogą być uruchamiane dla podanego zbioru punktów przy pomocy programu wiersza poleceń lub programu z interfejsem graficznym. Algorytmy triangulacji zostały szczegółowo opisane (pseudokody), łącznie z pomocniczymi strukturami danych.

Słowa kluczowe: triangulacja Delaunaya, diagram Voronoi, otoczka wypukła, odwracanie krawędzi, metoda dziel i zwyciężaj

English title: Delaunay triangulation in computational geometry

Abstract

Python implementation of five algorithms finding a Delaunay triangulation in computational geometry is presented. A Delaunay triangulation of a given set P of points in the plane is a triangulation such that no point in P is inside the circumcircle of any triangle from the triangulation. Delaunay triangulations are used for modelling terrain, automatic mesh generation, and other applications.

The following triangulation algorithms are prepared: a naive algorithm, the Lawson flipping algorithm, the incremental Bowyer-Watson algorithm, a divide and conquer algorithm by Lee and Schachter, and the quickhull algorithm. The algorithms were tested and they can be run for a given point set by means of a command line program or a program with graphical user interface. Triangulation algorithms are described in detail together with auxiliary data structures.

Keywords: Delaunay triangulation, Voronoi diagram, convex hull, edge flipping, divide and conquer algorithm

Spis treści

Spis rysunków

List of Algorithms

Listings

1. Wprowadzenie

Tematem niniejszej pracy jest triangulacja Delaunaya [1], która pełni ważną rolę w geometrii obliczeniowej. *Triangulacją* nazywamy podział płaskiej figury geometrycznej na trójkąty w taki sposób, że część wspólna dowolnych dwóch trójkątów jest ich wspólnym bokiem, wspólnym wierzchołkiem lub zbiorem pustym.

Istnieje wiele rodzajów triangulacji, w zależności m.in. od rodzaju obiektu poddanego triangulacji.

— *Triangulacja wielokąta* polega na podziale wielokąta na trójkąty, przy czym wierzchołkami trójkątów mogą być tylko wierzchołki wielokąta. Można dokonać triangulacji każdego wielokąta.
— *Triangulacja zbioru punktów* P polega na triangulacji *otoczki wypukłej* $\mathrm{CH}(P)$ tego zbioru punktów [2], przy czym wierzchołkami trójkątów są wszystkie punkty ze zbioru P. Do tej kategorii należy triangulacja Delaunaya i triangulacja o najmniejszej wadze (minimalna suma długości krawędzi).
— W metodzie elementów skończonych triangulacje są używane jako siatki do obliczeń (ang. *mesh*). Obok punktów podanych na wejściu, wierzchołkami trójkątów mogą być dodatkowe *punkty Steinera*, aby powstałe trójkąty spełniały pewne wymagane kryteria.

Książki poświęcone częściowo lub w całości geometrii obliczeniowej, z których zaczerpnięto informacje na temat triangulacji Delaunaya: [3], [4], [5], [6].

1.1. Cele pracy

Celem pracy jest analiza i implementacja w języku Python [7] wybranych algorytmów triangulacji Delaunaya. Język Python ma czytelną składnię, typy dynamiczne i bogatą bibliotekę standardową, przez co idealnie nadaje się do szybkiego tworzenia prototypów aplikacji, prezentacji rozwiązań bezpośrednio za pomocą kodu źródłowego, czy też jako język do skryptów zapewniających wymianę danych pomiędzy nie całkiem kompatybilnymi systemami.

Praca ma w zamyśle rozwijać pakiet `planegeometry`, którego zalążek powstał w pracach Marcina Permusa [8] i Wojciech Chrobaka [9]. Wykorzystano podstawowe obiekty geometryczne, takie jak punkty, odcinki, trójkąty, poza tym użyto kilku funkcji pomocniczych.

1.2. Organizacja pracy

Organizacja niniejszej pracy jest następująca. Rozdział 1 zawiera wprowadzenie i cele pracy. Rozdział 2 zawiera opis domeny zagadnienia. Znajduje się w nim opis podstawowych pojęć związanych z triangulacją oraz ich wyjaśnienie. Rozdział 3 zawiera opis wybranych algorytmów w postaci pseudokodów oraz szczegółowych objaśnień poszczególnych kroków. Rozdział 4 przedstawia decyzje implementacyjne oraz opisuje algorytmy pomocnicze, niezbędne do realizacji triangulacji. W tym rozdziale jest też opis programów demonstrujących działanie poszczególnych algorytmów triangulacji. Rozdział 5 poświęcony jest przeprowadzonym testom złożoności algorytmów, wraz z analizą wyników. Rozdział 6 podsumowuje pracę i przedstawia możliwe kierunki rozwoju.

2. Domena zagadnienia

Rozdział zawiera podstawowe definicje z geometrii obliczeniowej, które są niezbędne do opisu triangulacji Delaunaya.

2.1. Geometria obliczeniowa

Geometria obliczeniowa jest dziedziną algorytmiki zajmującą się badaniem algorytmów i struktur danych dla obiektów geometrycznych, z naciskiem na dokładne algorytmy, które są asymptotycznie szybkie. Rozwiązania geometrii obliczeniowej są używane w dziedzinach takich jak: robotyka, grafika komputerowa, CAD/CAM i systemy informacji geograficznej [3].

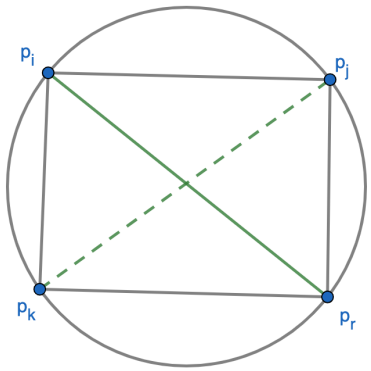
2.2. Triangulacja

Definicja: *Triangulacja Delaunaya (lub Delone)* DT(P) zbioru punktów P w R^n jest to podział R^n na triangulację T złożoną ze skończonej ilości n-sympleksów, w której:
— sympleksy nie nakładają ani nie przecinają się (tzn. dowolne 2 sympleksy dzielą ścianę, wierzchołek lub nie mają części wspólnej),
— przestrzeń zakreślona przez n-hipersferę opisaną na jednym z sympleksów nie zawiera wierzchołków żadnego innego sympleksu poza powierzchnią n-hipersfery.
Triangulację wymyślił rosyjski matematyk Borys Delone w roku 1934 [1]. W niniejszej pracy analizowana jest triangulacja punktów na płaszczyźnie, gdzie sympleksami są trójkąty.

Przykład zastosowania: Triangulacja znajduje swoje zastosowania w geodezji i topologii [3, s. 221]. Za jej pomocą możemy odwzorować mapę terenu ze wzniesieniami. Załóżmy, że mamy do dyspozycji punkty o określonej wysokości, które są punktami do triangulacji. Następnie wykonujemy triangulację tego zbioru punktów, nie uzwględniając ich wysokości. Dysponując triangulacją 2D i wysokościami punktów, można utworzyć triangulację 3D poprzez podniesienie wierzchołków trójkątów na odpowiednią wysokość. Dzięki użyciu triangulacji Delaunaya, unikamy trójkątów z ostrymi kątami i długimi bokami. To sprawia, że powstaly model 3D uśrednia wysokości punktów mapy bez pomiarów w oparciu o względnie mało oddalone punkty o znanych wysokościach, czyli daje dokładniejszą aproksymację i w efekcie wiarygodniejszy model.

Własności triangulacji Delaunaya [1]:

— Suma wszystkich sympleksów triangulacji (trójkątów) jest otoczką wypukłą danego zbioru punktów.
— Dla n punktów na płaszczyźnie, gdzie h punktów należy do otoczki wypukłej, dowolna triangulacja ma $2n-2-h$ trójkątów (bez ściany zewnętrznej grafu) i $3n-3-h$ krawędzi [korzystamy z właściwości grafów planarnych maksymalnych, z których usunięto $h-3$ krawędzi].
— Na płaszczyźnie każdy wierzchołek triangulacji Delaunaya jest wspólny dla średnio sześciu trójkątów [każdy z $2n-2-h$ trójkątów ma trzy wierzchołki, co daje sumę trójkątów sąsiadujących z wierzchołkami równą $6n-6-3h$].
— Triangulacja Delaunaya na płaszczyźnie maksymalizuje najmniejszy kąt w zbiorze kątów ze wszystkich trójkątów triangulacji. Długości krawędzi trójkątów nie muszą być minimalizowane.
— Czwórka punktów współokręgowych wyklucza jednoznaczność triangulacji Delaunaya (rys. 2.1).
— Rozważmy dwa trójkąty w triangulacji Delaunaya które mają wspólną krawędź, a kąty przeciwległe w tych trójkątach to α i β. Suma miar kątów przeciwległych $(\alpha+\beta)$ jest nie większa niż 180°. Ta własność jest związana z techniką *przekręcania krawędzi* (ang. *edge flipping*) używaną do znajdowania prawidłowej triangulacji Delaunaya.
— Triangulacja Delaunaya odpowiada grafowi dualnemu diagramu Voronoi [10] (rys. 2.3).
— Graf najbliższego sąsiedztwa (krawędzie łączą wierzchołki o najbliższym sąsiedztwie) jest podgrafem triangulacji Delaunay'a.



Rysunek 2.1: Przypadek szczególny niejednoznacznej triangulacji Delaunaya.

2.3. Diagram Voronoi

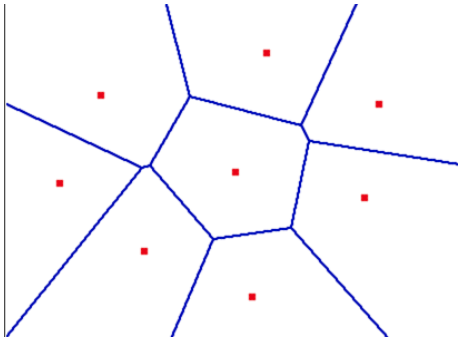
Definicja [3, s. 179]: Oznaczmy odległość euklidesową między punktami p i q przez $\mathrm{dist}(p,q)$. Na płaszczyźnie mamy:

$$\mathrm{dist}(p,q)=\sqrt{(p_x-q_x)^2+(p_y-q_y)^2}. \tag{2.1}$$

Niech $P=\{p_1,p_2,...p_n\}$ będzie zbiorem n różnych punktów na płaszczyźnie. Punkty te są centrami. Definiujemy diagram Voronoi dla P jako podział

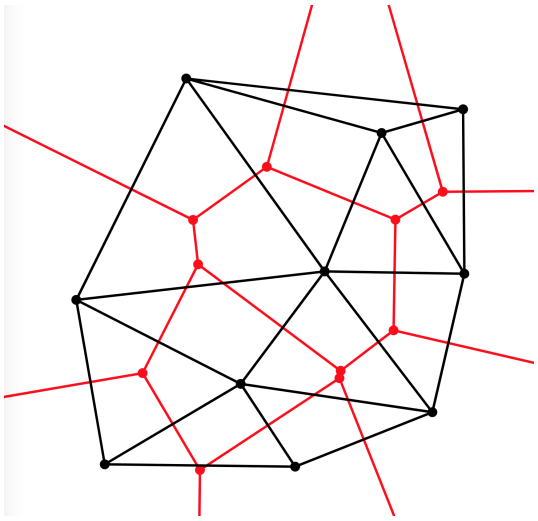
płaszczyzny na n komórek, jedna dla każdego centrum, o tej własności, że punkt q leży w komórce odpowiadającej centrum p_i wtedy i tylko wtedy, gdy $\text{dist}(q, p_i) < \text{dist}(q, p_j)$ dla każdego $p_j \in P$ takiego, że $j \neq i$ (rys. 2.2).

Podstawowy diagram Voronoi opisuje obszary najbliższe zestawowi określonych punktów. Można je postrzegać jako strefy kontroli, które można wykorzystać na przykład aby znaleźć najbliższy supermarket, aptekę czy szpital. Istnieją liczne zastosowania diagramu Voronoi w takich dziedzinach jak biologia, inżynieria biomedyczna, medycyna, planowanie przestrzeni, logistyka.



Rysunek 2.2: Diagram Voronoi, czerwone punkty są centrami komórek Voronoi.

Istotną własnością triangulacji Delaunaya jest to, że tworzy graf dualny z diagramem Voronoi (rys. 2.3). Czarne krawędzie to krawędzie triangulacja, a czerwone to komórki Voronoi. Oznacza to, że dysponując triangulacją Delaunaya możemy obliczyć diagram Voronoi.



Rysunek 2.3: Graf dualny, czerwone krawędzie wyznaczają komórki Voronoi, a czarne triangulację Delaunaya.

3. Wybrane algorytmy

W tej części znajduje się opis poszczególnych kroków algorytmów służących do przeprowadzenia triangulacji Delaunaya.

3.1. Algorytm naiwny

Algorytm naiwny jest to algorytm typu 'brute force', opisujący triangulację zbioru punktów P wg definicji, bez jakichkolwiek optymalizacji. Metoda polega na tworzeniu kolejno trójkątów ze zbioru punktów (sprawdza się wszystkie trójki różnych punktów). Dla każdego nowego trójkąta, przed dodaniem go do trójkątów triangulacji T upewniamy się, że w kole na nim opisanym nie znajduje się żaden punkt ze zbioru P. Złożoność czasowa algorytmu to $O(n^4)$. Poniżej pseudokod algorytmu:

Algorithm 1 Algorym naiwny

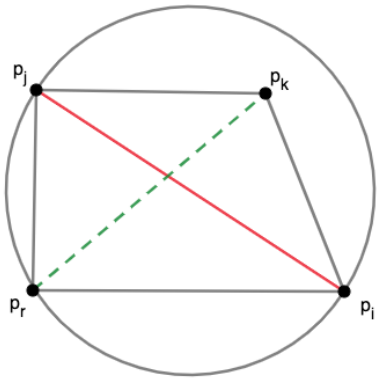
```
1:  procedure DelaunayTriangulationNaive(P)
2:      n ← |P|        ▷ P ← {p_1, p_2, ..., p_n} zbiór zrandomizowanych punktów
3:      T ← {}                                          ▷ pusty zbiór trójkątów
4:      for i ← 1 to n − 2 do
5:          for j ← i + 1 to n − 1 do
6:              for k ← j + 1 to n do
7:                  if points (p_i, p_j, p_k) are collinear then
8:                      continue
9:                  delaunayCondition ← true
10:                 for r ← 1 to n do
11:                     if p_r = p_i or p_r = p_j or p_r = p_k then
12:                         continue
13:                     if p_r in circumcircle(p_i, p_j, p_k) then
14:                         delaunayCondition ← false
15:                         break
16:                 if delaunayCondition then
17:                     T ← T ∪ {triangle(p_i, p_j, p_k)}
18:     return T
```

Algorytm działa prawidłowo dla punktów w pozycji ogólnej. Jeżeli pojawią się cztery punkty leżące na jednym okręgu, przez co triangulacja stanie się niejednoznaczna, to algorytm wytworzy trójkąty nadmiarowe z różnych triangulacji.

3.2. Metoda odwróceń Lawsona

Jest to metoda osiągnięcia triangulacji Delaunay'a przez legalizację krawędzi. Algorytm jest dokładnie opisany w książce de Berga [3, str. 231]. Algorytm buduje triangulację iteracyjnie, z dwuetapowych sekwencji:
1. dodanie trójkątów z wierzchołkami w kolejnym punkcie ze zbioru,
2. legalizacja powstałej triangulacji jako triangulacji Delaunay'a.

Legalizacja jest oparta na odwróceniach Lawsona. Jeśli dodane trójkąty nie spełniają założeń triangulacji Delaunay'a, grupujemy je parami ze wspólną krawędzią i odwracamy wspólną krawędź, tj. wspólna krawędź p_jp_i zostanie zastąpiona krawędzią p_rp_k (rys. 3.1).



Rysunek 3.1: Legalizacja krawędzi.

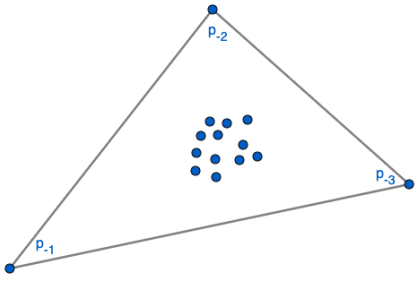
Triangulację nazywamy legalną, jeśli nie zawiera krawędzi nielegalnych. Trójkąty dzielące nielegalną krawędź, zawarte w okręgu opisującym jeden z nich, tworzą zawsze czworokąt wypukły. Odwrócenie Lawsona nie zmienia otoczki, którą jest ten czworokąt, legalizując wewnętrzną krawędź.

Algorytm ma złożoność w przypadku ogólnym $O(n \log n)$, a w najgorszym $O(n^2)$. Poniżej pseudokod algorytmu odwróceń [2].

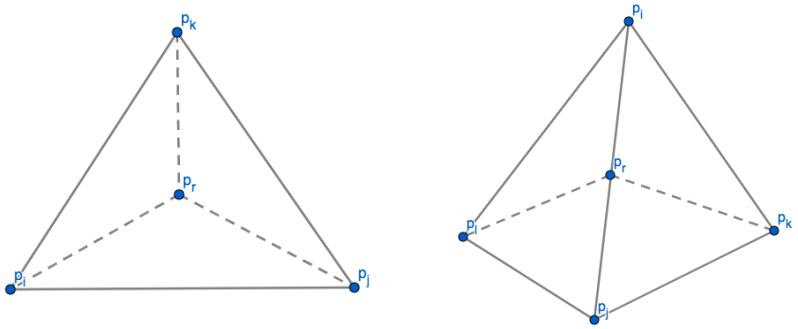
Algorithm 2 Metoda odwróceń Lawsona

procedure DelaunayTriangulationLawson(P)
  $n \leftarrow |P|$ ▷ $P \leftarrow \{p_1, p_2, ..., p_n\}$ zbiór zrandomizowanych punktów
  Niech $\{p_{-1}, p_{-2}, p_{-3}\}$ będzie zbiorem trzech punktów takim, że P jest zawarty w triangle(p_{-1}, p_{-2}, p_{-3}) (rys. 3.2).
  $T \leftarrow \{\text{triangle}(p_{-1}, p_{-2}, p_{-3})\}$ ▷ zbiór trójkątów
  for $r \leftarrow 1$ **to** n **do**
    Zlokalizuj triangle(p_i, p_j, p_k) w T zawierający p_r.
    if p_r leży wewnątrz triangle(p_i, p_j, p_k) **then**
      Dodaj krawędzie $p_r p_i, p_r p_j, p_r p_k$ dzieląc w ten sposób triangle(p_i, p_j, p_k) na trzy trójkąty, następnie legalizuj nowo dodane krawędzie [rys. 3.3 (a)]
      LegalizeEdge$(p_r, p_i p_j, T)$ [alg. 3]
      LegalizeEdge$(p_r, p_j p_k, T)$
      LegalizeEdge$(p_r, p_k p_i, T)$
    else ▷ p_r leży na pewnej krawędzi trójkąta
      Dodaj cztery trójkąty do triangulacji, następnie legalizuj nowo dodane krawędzie [rys. 3.3 (b)].
      LegalizeEdge$(p_r, p_i p_l, T)$
      LegalizeEdge$(p_r, p_l p_j, T)$
      LegalizeEdge$(p_r, p_j p_k, T)$
  Usuń z T punkty $\{p_{-1}, p_{-2}, p_{-3}\}$ wraz z incydentnymi krawędziami.
  return T

Aby znaleźć trójkąt wewnątrz którego jest zawarty punkt p, można skorzystać z grafu rejestrującego historię triangulacji. Graf ten znacznie zmniejsza ilość przeszukiwań.



Rysunek 3.2: Supertrójkąt.



(a) Punkt p_r wewnątrz trójkąta. (b) Punkt p_r na krawędzi.

Rysunek 3.3: Lokalizacja punktu.

Algorithm 3 Legalizacja krawędzi

1: **procedure** LEGALIZEEDGE(p_r, p_ip_j, T)
2: Wstawianym punktem jest p_r, a p_ip_j jest krawędzią T, która może wymagać przekręcenia.
3: **if** p_ip_j jest nielegalna **then**
4: Niech triangle(p_i, p_j, p_k) sąsiaduje z triangle(p_r, p_i, p_j) wzdłuż p_ip_j (rys. 3.1).
5: Przekręć p_ip_j, czyli zastąp p_ip_j przez p_rp_k.
6: LEGALIZEEDGE(p_r, p_ip_k, T)
7: LEGALIZEEDGE(p_r, p_kp_j, T)

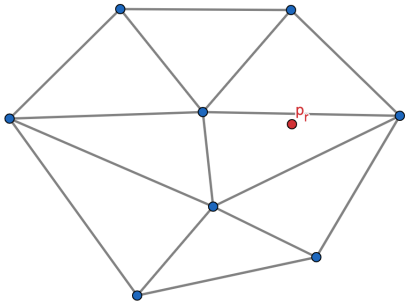
3.3. Algorytm Bowyera-Watsona

Algorytm Bowyera-Watsona jest to algorytm iteracyjny, który pokazuje alternatywne podejście legalizacji, zastępujące przekręcanie krawędzi [11]. Polega ono na retriangulacji otoczenia dodawanego punktu, czyli usuwaniu niepasujących trójkątów i tworzeniu nowych trójkątów przez dodawanie krawędzi z jednym końcem w dodanym punkcie.

Jeżeli potrafimy szybko znajdować trójkąty do usunięcia (niepasujące), to algorytm wymaga $O(n \log n)$ operacji do triangulacji n punktów, choć istnieją przypadki zdegenerowane z czasem $O(n^2)$.

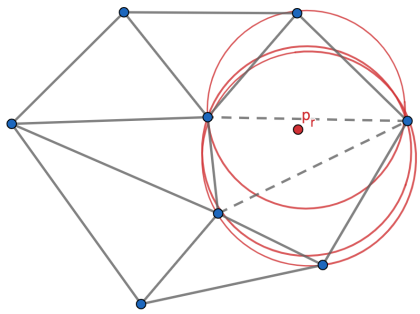
Kroki algorytmu

1. Tworzenie supertrójkąta (rys. 3.2).
2. Dodanie punktu p_r ze zbioru P (rys. 3.4).



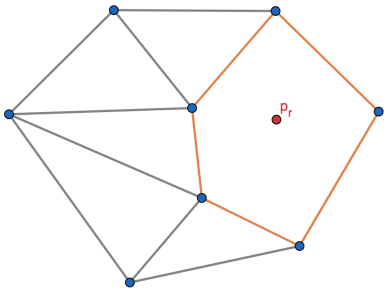
Rysunek 3.4: Nowy punkt w triangulacji.

3. Lokalizowanie trójkątów Z, dla których punkt p_r zawiera się w przestrzeni zakreślonej opisanym okręgiem (rys. 3.5).



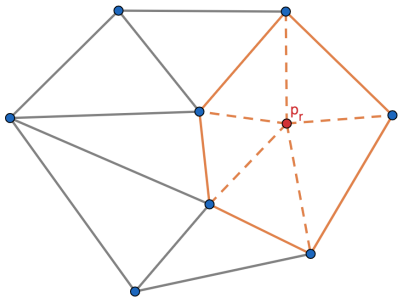
Rysunek 3.5: Trójkąty nielegalne.

4. Utworzenie wieloboku W wokół punktu p_r przez usunięcie trójkątów Z (rys. 3.6).



Rysunek 3.6: Wielobok W powstały po usunięciu nielegalnych trójkątów.

5. Retriangulacja przez połączenie każdego boku wieloboku W z punktem p_r (rys. 3.7).



Rysunek 3.7: Triangulacja po dodaniu punktu p_r.

Pseudokod algorytmu Bowyera-Watsona wygląda następująco.

Algorithm 4 Metoda Bowyera-Watsona

procedure DelaunayTriangulationBW(P)
  Niech $\{p_{-1}, p_{-2}, p_{-3}\}$ będzie zbiorem trzech punktów takim, że P jest zawarty w triangle(p_{-1}, p_{-2}, p_{-3}). ▷ P to zbiór punktów
  $T \leftarrow \{\text{triangle}(p_{-1}, p_{-2}, p_{-3})\}$ ▷ T to zbiór trójkątów
  for p in P **do**
    $TR \leftarrow \{\}$ ▷ pusty zbiór trójkątów do usunięcia
    for t in T **do**
      if p wewnątrz okręgu opisanego na t **then**
        Dodaj t do TR.
    for t in TR **do**
      for krawędź in t **do**
        if krawędź nie jest wspólna z żadną z krawędzi TR **then**
          Dodaj krawędź do wielokąta.
    for t in TR **do**
      Usuń t z T.
    for krawędź in wielokąt **do**
      Dodaj do T trójkąt tworzony z końców krawędzi i punktu p.
  Usuń z T punkty $\{p_{-1}, p_{-2}, p_{-3}\}$ wraz z incydentnymi krawędziami.
  return T

3.4. Algorytm dziel i zwyciężaj

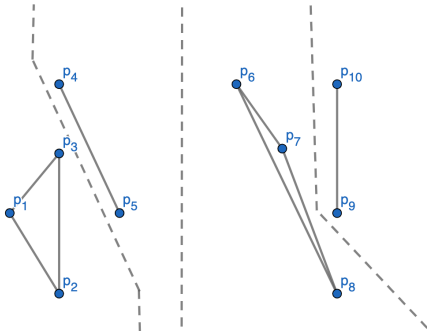
Algorytm dziel i zwyciężaj dla triangulacji w dwóch wymiarach został opracowany przez Lee i Schachtera, ulepszony przez Guibasa i Stolfiego, a później przez Dwyera. Triangulacja powstaje przez łączenie triangulacji podzbiorów. Operację scalania opartą na dodawanych od dołu liniach bazowych można wykonać w czasie $O(n)$. Algorytm wymaga sortowana punktów triangulacji po współrzędnej X, więc całkowity czas działania wynosi $O(n \log n)$.

Kroki algorytmu

1. Sortowanie punktów w P po współrzędnych X (rys. 3.8)

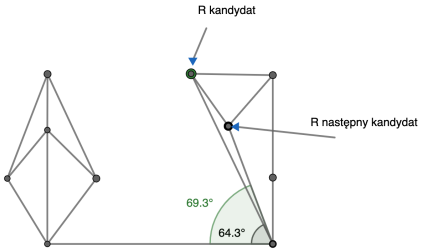
Rysunek 3.8: Zbiór posortowanych punktów P.

2. Rekurencyjne dzielenie P na pary podzbiorów L i R.
3. Obliczanie triangulacji Delaunaya dla dostatecznie małych podzbiorów L i R (2-3 punktowych) (rys. 3.9).

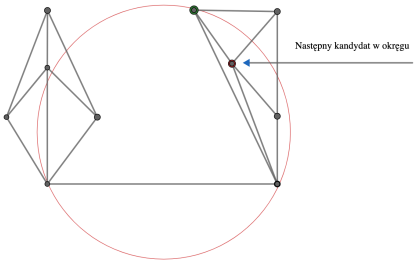


Rysunek 3.9: Triangulacje minimalnych podzbiorów P.

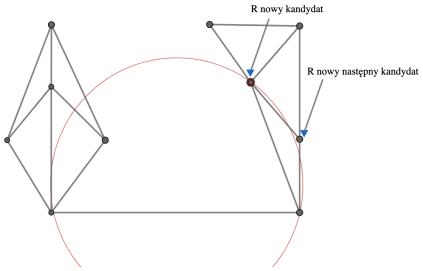
4. Rekurencyjne łączenie powstałych par triangulacji L i R wzdłuż linii podziału.
   a) Połączenie linią bazową najniżej położonych punktów podzbiorów L i R.
   b) Wybranie kandydata z podzbioru R. Najlepszym kandydatem jest ten o najmniejszym kącie pomiędzy krawędzią bazową uwzględniając orientację prawostronną, przy czym kąt nie może być większy niż 180°. Musimy sprawdzić czy w okręgu opisanym na trójkącie znajduje się następny kandydat, jeśli tak jest krawędź zostaje usunięta a kandydat zastąpiony (rys. 3.10)
   c) To samo dla lewego podzbioru (rys. 3.11).



(a) Wyznaczenie najlepszego kandydata.

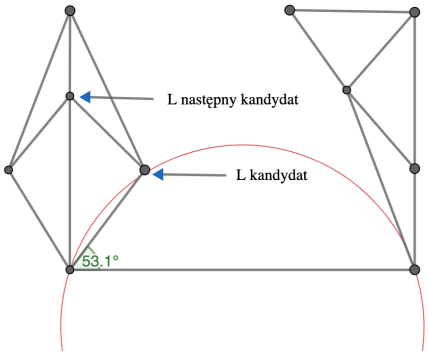


(b) Sprawdzenie poprawności.



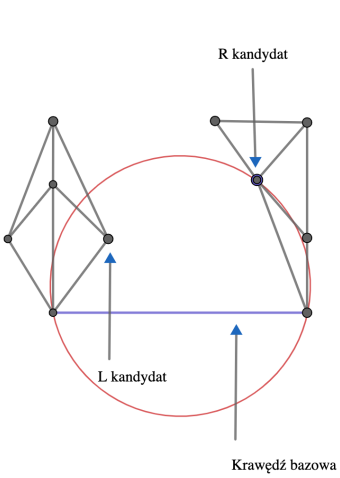
(c) Zmiana najlepszego kandydata.

Rysunek 3.10: Wybór kandydata ze zbioru R.

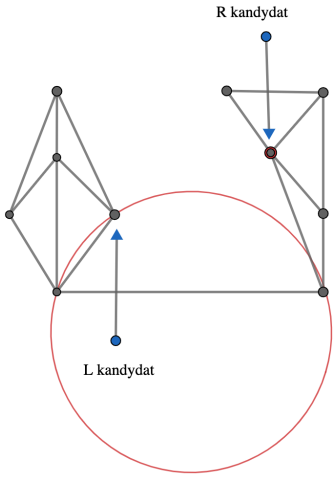


Rysunek 3.11: Wyznaczenie najlepszego kandydata z pozdbioru L.

d) Następnie rozstrzygamy, który z kandydatów (L czy R) utworzy krawędź z linią bazową. Wybieramy tego, dla którego okrąg opisany na powstałym z nim trójkącie nie zawiera punktu kandydata z przeciwnego zbioru. Na rys. 3.12 poniżej widzimy, że kandydat L spełnia to kryterium.



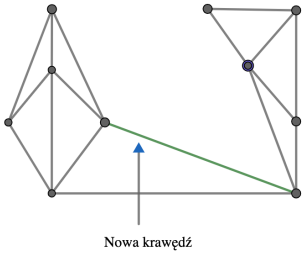
(a) Sprawdzenie kandydata R.



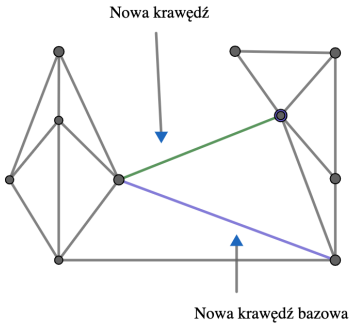
(b) Sprawdzenie kandydata L.

Rysunek 3.12: Wybór jednego z kandydatów L, R.

e) Proces wygląda tak samo dla kolejnych krawędzi, a każdą nowo dodaną traktujemy jak nową krawędź bazową (rys. 3.13).



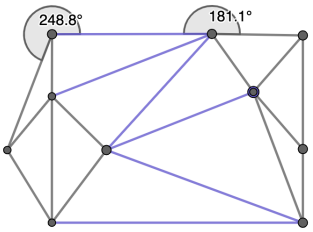
(a) Dodanie krawędzi.



(b) Uzupełnienie kolejnej krawędzi.

Rysunek 3.13: Dodanie kolejnych krawędzi.

f) Triangulacja się kończy, gdy nie jesteśmy w stanie wyznaczyć ani kandydata R ani L, czyli wtedy gdy minimalny kąt między bieżącą linią bazową a potencjalnym kandydatem jest większy bądź równy 180° dla obu kandydatów, jak na rys. 3.14.



Rysunek 3.14: Zakończenie triangulacjii.

Algorithm 5 Metoda Lee-Schachtera

```
1:  procedure LeeSchachter(P)
2:      n ← |P|                          ▷ P ← [p_1, p_2, ..., p_n] lista punktów
3:      Posortuj punkty z P względem współrzędnej X.
4:      Podziel P na dwie podlisty L ← P[0 : n/2] i R ← P[n/2 : n].
5:      if |L| > 3 then
6:          LeeSchachter(L)
7:      else
8:          Oblicz triangulację L bezpośrednio.
9:      if |R| > 3 then
10:         LeeSchachter(R)
11:     else
12:         Oblicz triangulację R bezpośrednio.
13:     Połącz krawędziami dwa podzbiory L i R.
```

3.5. Algorytm quickhull

Algorytm quickhull jest algorytmem służącym przede wszystkim do wyznaczenia otoczki wypukłej zbioru punktów. Opis idei algorytmu został zaczerpnięty z książki Galliera i Quaintance [16, str. 332-336].

Istnieje bliski związek pomiędzy otoczką wypukłą a triangulacją Delaunaya. Posiadając zbiór P punktów w przestrzeni Euklidesowej E^m o wymiarze m możemy podnieść te punkty na paraboloidę umiejscowioną w przestrzeni E^{m+1} (hiperpowerzchnia). Triangulacją Delaunaya będzie rzut skierowanych w dół ścianek otoczki wypukłej zestawu podniesionych punktów (rys. 3.16). To niezwykłe połączenie zostało po raz pierwszy odkryte przez Edelsbrunnera i Seidela.

Dla przykładu rozważymy zbiór punktów w przestrzeni E^2 oraz w pozycji ogólnej. Rozwiniemy punkty P w równanie paroboloidy przez dodanie wymiaru Z, który opisuje równanie:

$$z = x^2 + y^2. \tag{3.1}$$

Punkt $p = (x, y)$ na płaszczyźnie jest podniesiony do punktu $l(p) = (X, Y, Z)$ w E^3, gdzie $X \leftarrow x$, $Y \leftarrow y$, $Z \leftarrow x^2 + y^2$. Pierwszą kluczową obserwacją

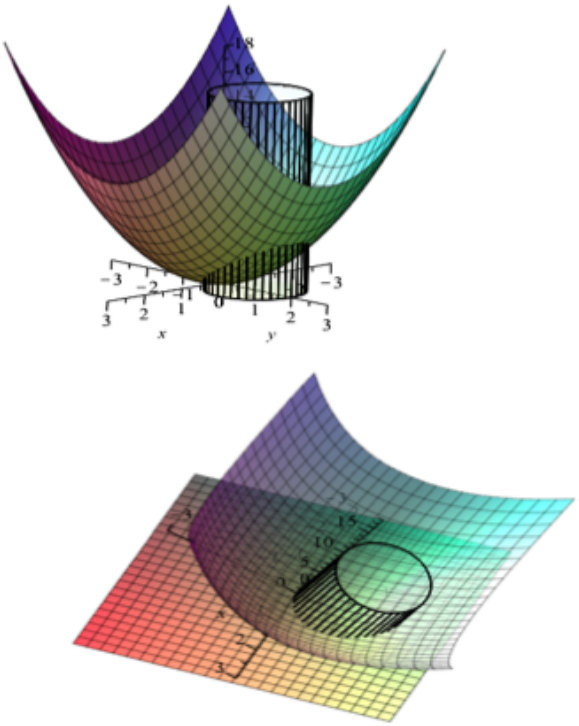
jest to, że okrąg na płaszczyźnie zostaje podniesiony do elipsy. Zdefiniujmy okrąg C przez równanie

$$x^2 + y^2 + ax + by + c = 0. \tag{3.2}$$

Ponieważ $Z \leftarrow x^2 + y^2$, po przejściu do E^3 dostaniemy równanie liniowe

$$aX + bY + Z + c = 0. \tag{3.3}$$

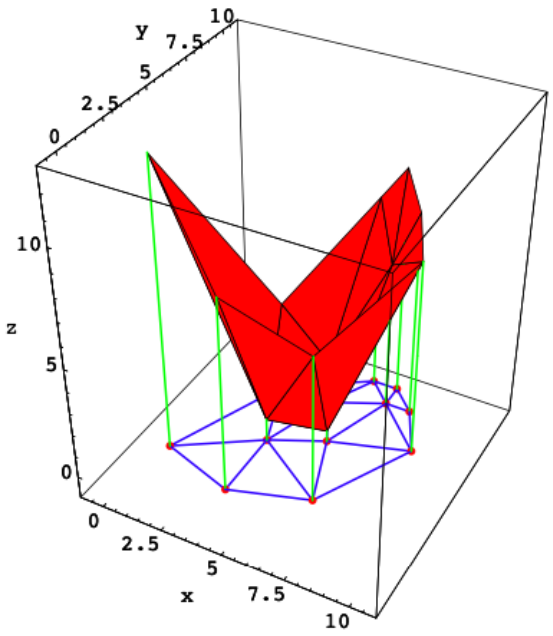
Jest to równanie płaszczyzny. Zatem przecięcie cylindra obrotu składającego się z linii równoległych do osi z i przechodzących przez punkt okręgu C z paraboloidą $z = x^2 + y^2$ jest elipsą, jak pokazano na rys. 3.15.



(a) Przecięcie paraboloidy $x^2 + y^2 = z$ z walcem $x^2 + (y - 1)^2 = 0$. Przecięcie to elipsa na płaszczyźnie $z = 2y - 1$.

Rysunek 3.15: Okrąg podniesiony do paraboloidy.

Możemy obliczyć otoczkę wypukłą zbioru podniesionych punktów. Skoncentrujmy się na skierowanych w dół ścianach otoczki wypukłej. Ściana dolna jest ścianą taką, że współrzędna Z wektora normalnego płaszczyzny, podtrzymującej tę ścianę, skierowanego do wnętrza otoczki wypukłej zestawu podniesionych punktów, jest dodatnia.



Rysunek 3.16: Triangulacja Delaunaya i jej podniesienie do paraboloidy.

Fakt, że triangulację Delaunaya można uzyskać poprzez rzutowanie dolnej otoczki wypukłej, można wykorzystać do znalezienia wydajnych algorytmów obliczania triangulacji Delaunaya. Zależność ta dotyczy również wyższych wymiarów. Algorytm quickhull jest szybki [jego złożoność to $O(n \log n)$] oraz uniwersalny - można go rozwijać do dowolnej liczby wymiarów. Jego implementację możemy znaleźć między innymi w popularnych bibliotekach geometrii obliczeniowej (CGAL) oraz w popularnych programach matematycznych (MatLab).

Pseudokod algorytmu pokazuje obliczanie triangulacji w 2D.

Algorithm 6 Algorytm quickhull dla triangulacji Delaunaya

procedure QUICKHULL-DELAUNAY(P)
  $n \leftarrow |P|$ ▷ $P \leftarrow [p_1, p_2, ..., p_n]$ lista punktów
  $T \leftarrow \{\}$ ▷ pusty zbiór trójkątów
  $S \leftarrow \{\}$ ▷ pusty zbiór podniesionych punktów
  for p in P **do**
    Podnieś punkt $p(x, y)$ do punktu $p'(x, y, z)$, gdzie $z = x^2 + y^2$.
    Dodaj podniesiony punkt p' do zbioru S.
  $H \leftarrow$ QHULL(S) ▷ Oblicz otoczkę wypukłą dla S
  for t' in H **do**
    if t' jest trójkątem skierowanym w dół względem środka paraboloidy **then**
      Mapuj trójkąt t' na trójąt t leżący w płaszczyźnie.
      Dodaj trójkąt t do zbioru T.

4. Implementacja

Algorytmy zostały zaimplementowane w języku Python w wersji 3.6. Zaimportowane biblioteki to SciPy do algorytmu quickhull, NumPy i Gnuplot do generowania rysunków i wykresów oraz PyQt5 do implementacji interfejsu aplikacji.

4.1. Struktury danych

W tej sekcji opiszę podstawowe obiekty geometryczne potrzebne do implementacji triangulacji, oraz użyte struktury danych.

Figury geometryczne: Klasy `Point`, `Circle`, `Edge`, `Triangle` reprezentują kolejno punkt, okrąg, odcinek (używany jako krawędź trójkąta) oraz trójkąt na płaszczyźnie. Okrąg opisany na trójkącie jest niezbędny do legalizacji trójkątów. Trójkąt przechowuje informację o swoich wierzchołkach, jak i krawędziach.

Kolekcje: Pomocnicze struktury danych.

— `triangle_pairs` - słownik mapujący krawędź dzieloną przez parę trójkątów na listę z tymi trójkątami; użyty w algorytmie "Legal Triangulation".
— `triangles_collection` - lista trójkątów triangulacji; użyta w algorytmach "Bowyer Watson" i "Legal".
— `points_graph` - graf łączący każdy punkt triangulacji z sąsiadującymi z nim punktami; użyty w algorytmie "Divide and Conquer".
— `triangles_history` - drzewo zawierające kolejno dodawane trójkąty triangulacji; użyte w algorytmie "Legal".

4.2. Algorytmy pomocnicze

W tej sekcji zawarte są kluczowe algorytmy, które są używane w algorytmach służących do triangulacji.

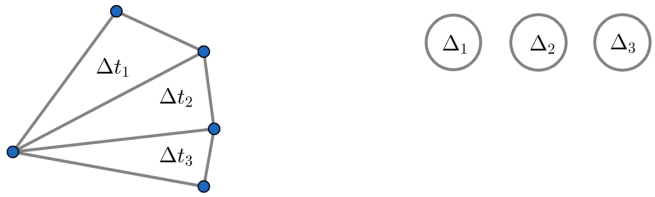
4.2.1. Lokalizacja trójkąta zawierającego dodany punkt

Opis struktury: Aby przyspieszyć lokalizację trójkąta, możemy użyć dodatkowej struktury - `triangles_history` [3, str. 235]. Struktura ta będzie przechowywała historię triangulacji w postaci drzewa kolejno powstałych trójkątów (poniżej znajdują się rysunki wyjaśniające budowę drzewa). Zawężamy ilość przeszukiwanych trójkątów przez przeszukiwanie tylko w rejonach, w

których jest szukany punkt. Zastosowanie drzewa historii znacząco wpływa na wydajność algorytmu w porównaniu z wersją z przeszukiwaniem liniowym, co można zaobserwować w porównawczych testach złożoności [sekcja 5.2].

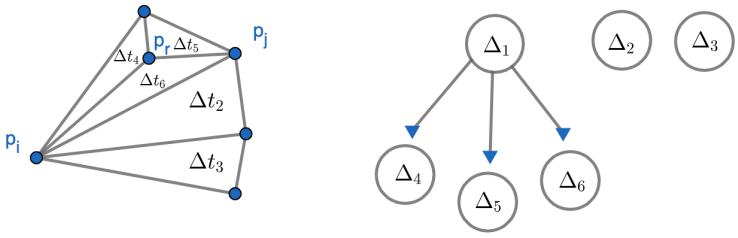
Tworzenie drzewa historii trójkątów:

1. Załóżmy, że dysponujemy częściową triangulacją i mamy częściowo wypełnione drzewo historii [rys. 4.1]:



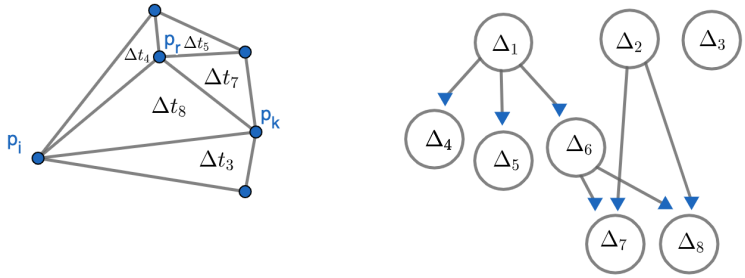
Rysunek 4.1: Początek drzewa historii.

2. Następnie dokładamy nowy punkt p_r. Prowadzi to do podziału t_1 na trzy nowe trójkąty, w tej chwili również dodajemy nowe połączenia w grafie dla trójkąta Δ_1 [rys. 4.2]:



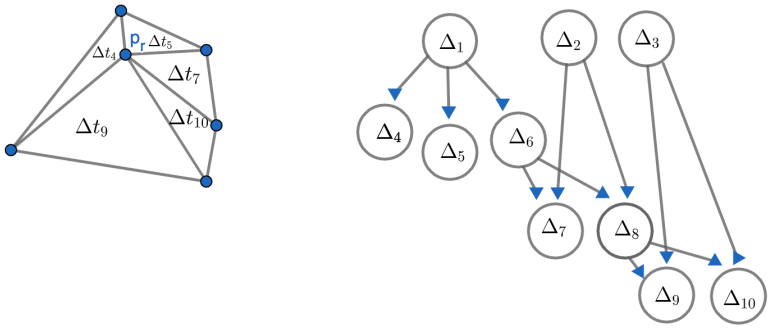
Rysunek 4.2: Podział trójkąta.

3. Następnie podczas legalizacji okazało się, że jedna z krawędzi nie jest legalna i została przekręcona. Powoduje to dodanie nowych trójkątów do drzewa historii jako liści oraz połączenia liści z resztą drzewa [rys. 4.3]:



Rysunek 4.3: Legalizacja krawędzi.

4. Kolejne legalizacje skutkują kolejnymi przekręceniami krawędzi; powoduje to dodanie nowych trójkątów i połączeń do drzewa historii [rys. 4.4]:



Rysunek 4.4: Graf z historią zmian.

Liście drzewa to trójkąty triangulacji, a korzeń i gałęzie pośrednie to historia. Historia pozwala nam na przeszukiwanie triangulacji rejonami, czyli zawężania obszaru szukania aż dotrzemy do liścia, czyli trójkątu, wewnątrz którego jest nowo dodawany punkt triangulacji.

4.2.2. Sprawdzenie, czy punkt zawiera się wewnątrz trójkąta

Opis algortmu: Występują dwa podejścia do określenia czy punkt znajduje się we wnętrzu trójkąta:

1. Metoda porównania pól powierzchni: Podczas dodawania nowego punktu p_r do trójkąta $p_a p_b p_c$ sprawdzamy, czy pole powierzchni trójkąta $p_a p_b p_c$ jest równe sumie pól powierzchni kolejno trzech trójkątów: $p_a p_b p_r$, $p_a p_c p_r$ i $p_b p_c p_r$. Jeśli tak, punkt p_r jest wewnątrz trójkąta.
2. Metoda orientowania punktu względem krawędzi trójkąta. Dla każdej krawędzi trójkąta sprawdzamy, czy punkt leży względem niej po tej same stronie, co przeciwległy do tej krawędzi wierzchołek. Jeśli tak, punkt ten jest w jego wnętrzu.

Implementacja algorytmu jest oparta na podejściu 2:

Listing 4.1: Test `point in triangle`.

```
class Triangle:
# inne metody ...
    def __contains__(self, other):
        """Test if a point is in a triangle."""
        if isinstance(other, Point):
            a12 = orientation(self.pt1, self.pt2, self.pt3)
            b12 = orientation(self.pt1, self.pt2, other)
            a23 = orientation(self.pt2, self.pt3, self.pt1)
            b23 = orientation(self.pt2, self.pt3, other)
            a31 = orientation(self.pt3, self.pt1, self.pt2)
            b31 = orientation(self.pt3, self.pt1, other)
            return (a12 * b12 >= 0) and (a23 * b23 >= 0) and (a31 * b31 >= 0)
        else:
            raise ValueError("not a point")
```

4.2.3. Inicjalizacja supertrójkąta

W algorytmach wyznaczających triangulację wygodnie jest rozpocząć z dużym trójkątem zawierającym wszystkie punkty. Ten supertrójkąt wyznaczają trzy dodatkowe punkty, które będą usuwane po zakończeniu triangulacji.

Opis algortmu: Znajdujemy maksymalną wartość bezwzględną wszystkich współrzędnych punktów (zmienna `big`). Tej wartości, przemnożonej przez współczynnik m, użyjemy do wyznaczania współrzędnych supertrójkąta [alg. 4.2]. W książce de Berga [3, s. 236] autorzy ustalają $m = 3$, ale w szczególnej sytuacji pewne punkty mogą znaleźć się na boku supertrójkąta, co nie jest pożądane. Z tego powodu ustaliliśmy wartość $m = 4$.

Listing 4.2: Tworzenie supertrójkąta.

```
big = max(max(abs(point.x), abs(point.y)) for point in point_list)
m = 4  # najlepiej typu int
p1 = Point(m * big, 0)
p2 = Point(0, m * big)
p3 = Point(-m * big, -m * big)
# Triangle(p1, p2, p3) to supertrojkat.
```

4.2.4. Sprawdzenie warunku Delaunaya

Algorytmy triangulacji wymagają częstego sprawdzenia warunku Delaunaya - testu, czy dany punkt p_4 leży wewnątrz okręgu opisanego na trójkącie o wierzchołkach p_1, p_2, p_3. Jeden ze sposobów sprawdzenia tego warunku wymaga obliczenia wyznacznika układu równań wyznaczającego współokręgo-

wość p_1, p_2, p_3 oraz p_4, gdzie p_1, p_2, p_3, p_4 są ułożone w kolejności przeciwnej do ruchu wskazówek zegara [12]:

$$\text{in\_circumcircle}(p_1, p_2, p_3, p_4) \equiv \det \begin{pmatrix} x_1 & y_1 & x_1^2 + y_1^2 & 1 \\ x_2 & y_2 & x_2^2 + y_2^2 & 1 \\ x_3 & y_3 & x_3^2 + y_3^2 & 1 \\ x_4 & y_4 & x_4^2 + y_4^2 & 1 \end{pmatrix} > 0. \tag{4.1}$$

Listing 4.3: Metoda `Triangle.in_circumcircle()`.

```
class Triangle:
# inne metody ...
    def in_circumcircle(self, point):
        """ Check if point is inside triangle circumcircle."""
        # Preparing parameters for calculating det for 3x3 matrix
        a = self.pt1 - point
        b = self.pt2 - point
        c = self.pt3 - point
        det = (a*a) * b.cross(c) - (b*b) * a.cross(c) + (c*c) * a.cross(b)
        if orientation(self.pt1, self.pt2, self.pt3) > 0:
            return det > 0
        else:
            return det < 0
```

Warto przytoczyć inne wzory na parametry okręgu opisanego na trójkącie p_1, p_2, p_3. Przyjmijmy następujące oznaczenia

$$\begin{aligned} s_1 &= x_1^2 + y_1^2, \quad \Delta x_{13} = x_1 - x_3, \quad \Delta y_{12} = y_1 - y_2, \\ s_2 &= x_2^2 + y_2^2, \quad \Delta x_{32} = x_3 - x_2, \quad \Delta y_{23} = y_2 - y_3, \\ s_3 &= x_3^2 + y_3^2, \quad \Delta x_{21} = x_2 - x_1, \quad \Delta y_{31} = y_3 - y_1, \end{aligned} \tag{4.2}$$

$$f = 2(x_1 \Delta y_{23} + x_2 \Delta y_{31} + x_3 \Delta y_{12}). \tag{4.3}$$

Współrzędne środka okręgu opisanego na trójkącie wyrażają się wzorami

$$x_0 = \frac{s_1 \Delta y_{23} + s_2 \Delta y_{31} + s_3 \Delta y_{12}}{f}, \tag{4.4}$$

$$y_0 = \frac{s_1 \Delta x_{32} + s_2 \Delta x_{13} + s_3 \Delta x_{21}}{f}. \tag{4.5}$$

Promień okręgu opisanego na trójkącie wynosi

$$R = \frac{\sqrt{(\Delta x_{21}^2 + \Delta y_{12}^2)(\Delta x_{13}^2 + \Delta y_{31}^2)(\Delta x_{32}^2 + \Delta y_{23}^2)}}{|f|}. \tag{4.6}$$

4.2.5. Wyznaczenie kąta pomiędzy trzema punktami

W algorytmie dziel i zwyciężaj tworzone są podzbiory triangulacji, które następnie są rekurencyjnie łączone odpowiednio dobranymi krawędziami. Dobór krawędzi oparty jest na kolejności kątów między kandydatami,

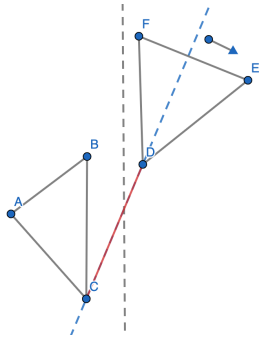
a uprzednio utworzonymi krawędziami. Potrzebny jest sposób wyznaczania kątów o wysokiej dokładności i pełnym zakresie zwracanej wartości kąta. Warunki te spełnia następująca metoda, oparta na funkcjach trygonometrycznych z biblioteki `math`:

Listing 4.4: Funkcja `get_angle()`.

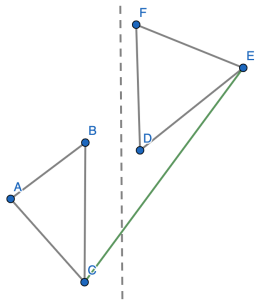
```
def get_angle(a, b, c):
    """The function calculates an angle using the atan() function,
    and mathematical order - counterclockwise.
    Source:
    https://python-forum.io/Thread-finding-angle-between-three-points-on-a-2d-graph
    """
    angle = math.degrees(math.atan2(c.y - b.y, c.x - b.x)
                         - math.atan2(a.y - b.y, a.x - b.x))
    return angle
```

4.2.6. Wyznaczenie krawędzi bazowej

Krawędź bazowa jest pierwszą krawędzią łączącą dwa podzbiory triangulacji w algorytmie dziel i zwyciężaj. Implementacja algorytmu łączenia opartego na linii bazowej łączącej bez przecięć najniższe punkty podzbiorów [13] okazała się niewystarczająca do utworzenia otoczki wypukłej w niektórych przypadkach (rys. 4.5).



(a) Początkowi kandydaci.



(b) Punkt E jest właściwym kandydatem.

Rysunek 4.5: Lokalizacja krawędzi bazowej.

W poszukiwaniu ulepszenia metody wyznaczania krawędzi bazowej, początkowy algorytm obliczał kąty między punktami potencjalnej linii bazowej, a ich sąsiadami. Jeżeli kąt między sąsiadem a linią bazową przekraczał 180°, oznaczało to, że sąsiad ten jest lepszym kandydatem do utworzenia linii bazowej niż początkowo rozpatrywany najniższy punkt z danego podzbioru.

Zachodzi prostą zależność, która dalej upraszcza implementację: dla poprawnie wyznaczonej krawędzi bazowej, żaden punkt nie może leżeć poniżej niej, co oznacza że wszystkie sąsiadujące punkty mają taką samą oriantację względem linii bazowej. Jeśli znajdziemy punkt o przeciwnej orientacji, to staje się lepszym kandydatem do krawędzi bazowej, niż obecny w niej jego sąsiad. Obliczenie orientacji trzech punktów jest mniej kosztowne obliczeniowo, niż obliczenie kąta pomiędzy trzema punktami.

Rozwiązanie to wymaga dobrego obsłużenia przypadku granicznego, w którym rozpatrywany sąsiad jest współliniowy z potencjalną linią bazową - orientacja trzech punktów jest równa zero. W takiej sytuacji porównujemy współrzedne punktów tak, aby wybrać do linii bazowej punkty położone najbliżej siebie z obu podzbiorów. Kod algorytmu jest przedstawiony na listingu 4.5.

Listing 4.5: Funkcja `find_baseline()`.

```
def find_baseline(l, r):
    """Searches for baseline for two subsets.
     Baseline is a line that connects two subsets
     in the way that lower convex hull is created.
    """
    # Determine which site has lower candidate to search from it.
    l_candidate = find_lowest(l, "L")
    r_candidate = find_lowest(r, "R")
    while True:
        one_side = True
        for l_neighbor in l:
            if orientation(l_candidate, r_candidate, l_neighbor) == -1:
                one_side = False
                l_candidate = l_neighbor
            # corner case: all points in line
            if orientation(l_candidate, r_candidate, l_neighbor) == 0:
                if l_candidate.y < r_candidate.y:
                    if l_neighbor.y > l_candidate.y:
                        one_side = False
                        l_candidate = l_neighbor
        for r_neighbor in r:
            if orientation(l_candidate, r_candidate, r_neighbor) == -1:
                one_side = False
                r_candidate = r_neighbor
            # corner case: all points in line
            if orientation(l_candidate, r_candidate, r_neighbor) == 0:
                if r_candidate.y < l_candidate.y:
                     if r_neighbor.y > r_candidate.y:
                        one_side = False
                        r_candidate = r_neighbor
        if one_side:
            baseline = Edge(l_candidate, r_candidate)
            return baseline
```

4.2.7. Wyznaczenie dolnej ściany otoczki w 3D względem środka paraboloidy

Algorytm Q Hull tworzy triangulację z krawędzi dolnej ściany otoczki punktów podniesionych do trzeciego wymiaru. Potrzebny jest sposób odróżnienia ściany dolnej od górnej. W tym celu wyznaczamy wektor normalny do płaszczyzny ściany otoczki, skierowany w punkt p_4 znajdujący się wewnątrz otoczki, czyli w grupie punktów triangulacji. Wektor normalny p_1p_4 do płaszczyzny zawierającej trójkąt $p_1p_2p_3$ jest liczony z wyznacznika wektorów p_1p_2 i p_1p_3. Współrzędna pionowa otrzymanego wektora normalnego określa orientację ściany otoczki. Listing 4.6 przedstawia kod algorytmu.

Listing 4.6: Funkcja `is_lower_face()`.

```
# http://kitchingroup.cheme.cmu.edu/blog/2015/01/18/Equation-of-a-plane-through-three-points/
# function is returning true if point p4 is above plane (a,b,c)
def is_lower_face(a, b, c, p4):
    p1 = np.array([a[0], a[1], a[2]])
    p2 = np.array([b[0], b[1], b[2]])
    p3 = np.array([c[0], c[1], c[2]])

    # These two vectors define the plane
    points = [p1, p2, p3]
    v1 = points[2] - points[0]
    v2 = points[1] - points[0]

    # the cross product is a vector normal to the plane
    cp = np.cross(v1, v2)
    a, b, c = cp

    # This evaluates a * x3 + b * y3 + c * z3 which equals d
    d = np.dot(cp, p3)

    z = (d - a * p4[0] - b * p4[1]) / c
    return p4[2] >= z
```

4.3. Algorytmy triangulacji

Implementacje triangulacji Delaunaya wykorzystują dwie główne techniki programowania:

— podejście iteracyjne: `BWDelaunayTriangulation` (algorytm Bowyera-Watson), `LegalDelaunayTriangulation` (metoda odwróceń Lawsona), `NaiveDelaunayTriangulation` (algorytm naiwny), `QHull` (algorytm Quickhull dla triangulacji Delaunaya),

— podejście dziel i zwyciężaj: `DivideConquer` (algorytm zoptymalizowany przez Stolfi i Guibas),

— podejście hybrydowe: `SHull` (algorytm w szybki sposób wyznacza triangulację poprzez: 'radially propagating sweep-hull', którą następnie trzeba zlegalizować. Jest to szybki algorytm, ponieważ triangulacja przed legalizacją jest wyznaczana bardzo wydajnie, a ilość trójkątów potrzebujących legalizacji jest stosunkowo mała).

Algorytmy oparte na tych trzech podejściach mają złożoność rzędu $O(n \log n)$, przy czym podejście dziel i zwyciężaj jest bardziej wydajne, niż iteracyjne. W pracy zawarte są opisy oraz implemantacje wyżej wymienionych technik.

4.3.1. Algorytm naiwny

Implementacja algorytmu na wejściu potrzebuje listy punktów. Po obliczeniach algorytm zwraca listę trójkątów triangulacji. Implementacja jest zawarta w module `NaiveDelaunayTriangulation`.

4.3.2. Algorytm z legalizacją krawędzi

Implementacja algorytmu na wejściu porzebuje listy punktów oraz wartości maksymalnej współrzędniej zbioru. Po obliczeniach algorytm zwraca listę trójkatów triangulacji. Przygotowano trzy różne implementacje algorytmu:
— wersja z kolekcją trójkątów (moduł `LegalDelaunayTriangulation`),
— wersja z kolekcją krawędzi `triangle pairs` (moduł `LegalDelaunayTriangulation2`),
— wersja z kolekcją krawędzi `triangle pairs` oraz trójkątów `triangles history` (moduł `LegalDelaunayTriangulation3`),

4.3.3. Algorytm Bowyera-Watsona

Implementacja algorytmu na wejściu porzebuje listy punktów oraz wartości maksymalnej współrzędniej zbioru. Po obliczeniach algorytm zwraca listę trójkatów triangulacji. Implementacja jest zawarta w module `BWDelaunayTriangulation`.

4.3.4. Algorytm dziel i zwyciężaj

Implementacja algorytmu na wejściu porzebuje listy punktów. Po obliczeniach algorytm zwraca listę trójkatów triangulacji. Implementacja jest zawarta w module `DivideConquer2`.

4.3.5. Algorytm quickhull dla triangulacji Delaunaya

Implementacja algorytmu na wejściu porzebuje listy punktów. Po obliczeniach algorytm zwraca listę trójkatów triangulacji. Koniecznym krokiem algorytmu jest wygenerowanie otoczki wypukłej zbioru punktów w 3D. Implementacja nie zawiera tego kroku tylko używa gotowego rozwiązania z biblioteki `scipy.spatial`. Implementacja jest zawarta w module `QHullDT`.

4.4. Program z interfejsem tekstowym

Program znajduje się w folderze `mw_triangulation_cmd` i umożliwia wybranie i uruchomienie dowolnego algorytmu triangulacji. Po wykonaniu obliczeń, generuje plik z danymi utworzonych trójkątów oraz rysunek triangulacji.

Program jest napisany w języku Python 3.6. Przed uruchomieniem należy zainstalować bibliotekę SciPy; do tego celu używamy w terminalu polecenia 'pip3 install scipy'. Dodatkowo wymagana jest instalacja programu Gnuplot.

Program może pracować w jednym z dwóch trybów: generacji albo wczytania danych punktów triangulacji. Parametry podajemy przy uruchomieniu jako cztery argumenty:
— Wielkość maksymalnej współrzędniej x lub y: liczba całkowita nie mniejsza niż 3.
— Tryb pracy: `random` do generacji danych wejściowych albo `file` do wczytania ich z pliku.
— Dane triangulacji: nazwa przygotowanego wcześniej w folderze `Input` pliku z danymi (tryb `random`) lub ilość punktów do wygenerowania losowo (tryb `random`). Program wymaga unikalnych punktów w danych wejściowych.

— Nazwa algorytmu, który ma zostać uruchomiony na danych wejściowych: `naive`, `legal`, `legal2`, `legal3`, `bw` (Bowyer-Watson), `dc` (Divide and Conquer), `qhull` albo `all` (uruchomione zostaną kolejno wszystkie algorytmy).

Przkładowe wywołania programu z folderu `mw_triangulation_cmd`:

```
python3 __run__.py 200 random 50 all
python3 __run__.py 50 file data.txt naive
```

Lub za pomocą skryptu:

```
./run_cmd.sh 200 random 50 all
./run_cmd.sh 50 file data.txt naive
```

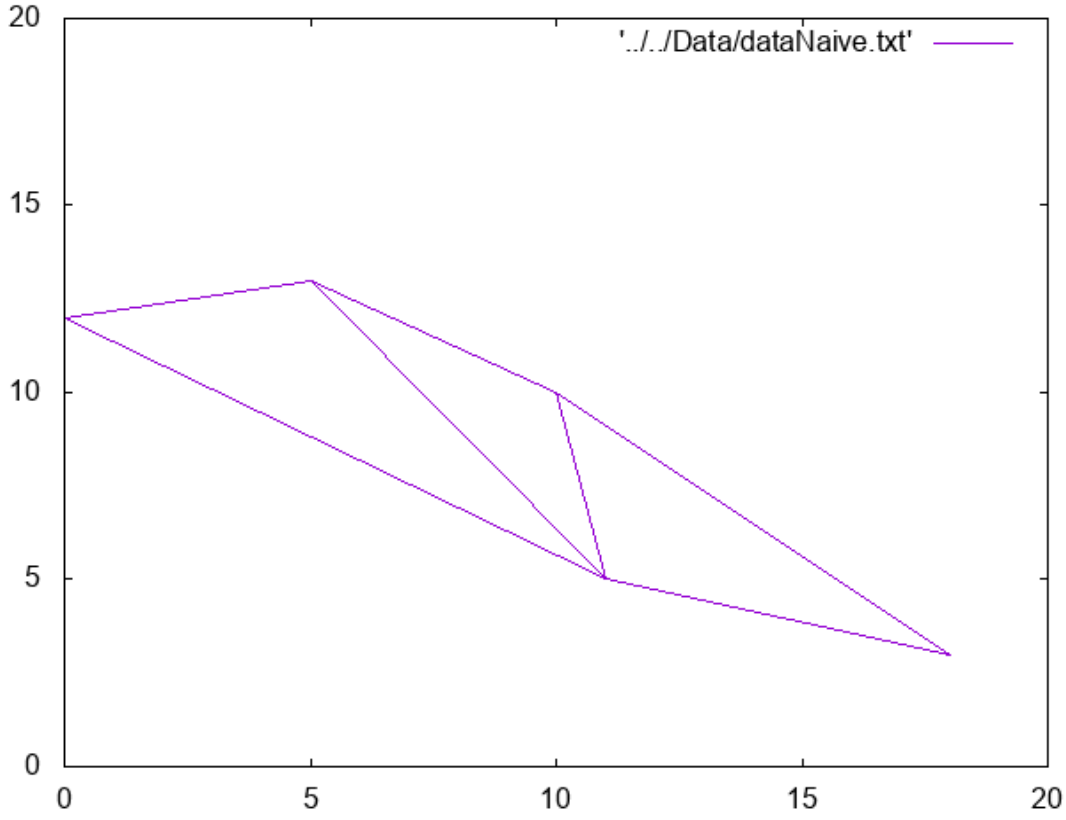
Plik z danymi wejściowymi powinien zawierać współrzędne punktów triangulacji, oddzielone spacjami, podane w nowych liniach dla każdego punktu, tj:

...
2 3
11 7
...

Każdy z algorytmów zwraca dokument .txt, w którym zawarte są trójkąty triangulacji w formacie (pierwszy punkt jest powtórzony na końcu):

...
18 3
10 10
11 5
18 3
...

Oprócz danych trójkątów, tworzony jest też rysunek triangulacji w formacie .png.



(a) Program: generowanie triangulacji.

4.5. Program z interfejsem graficznym

Program `mw_triangulation_gui` powstał, aby pokazać nie tylko efekt końcowy, ale też kroki niektórych algorytmów. Możliwe jest tylko generowanie punktów losowych, nie więcej niż 100 na raz.

Program jest zaimplementowany w języku python 3.6 z użyciem biblioteki graficznej PyQt5. Bibliotekę można pobrać za pomocą:

```
pip3 install pyqt5
```

Dodatkowo potrzebna jest też biblioteka SciPy:

```
pip3 install scipy
```

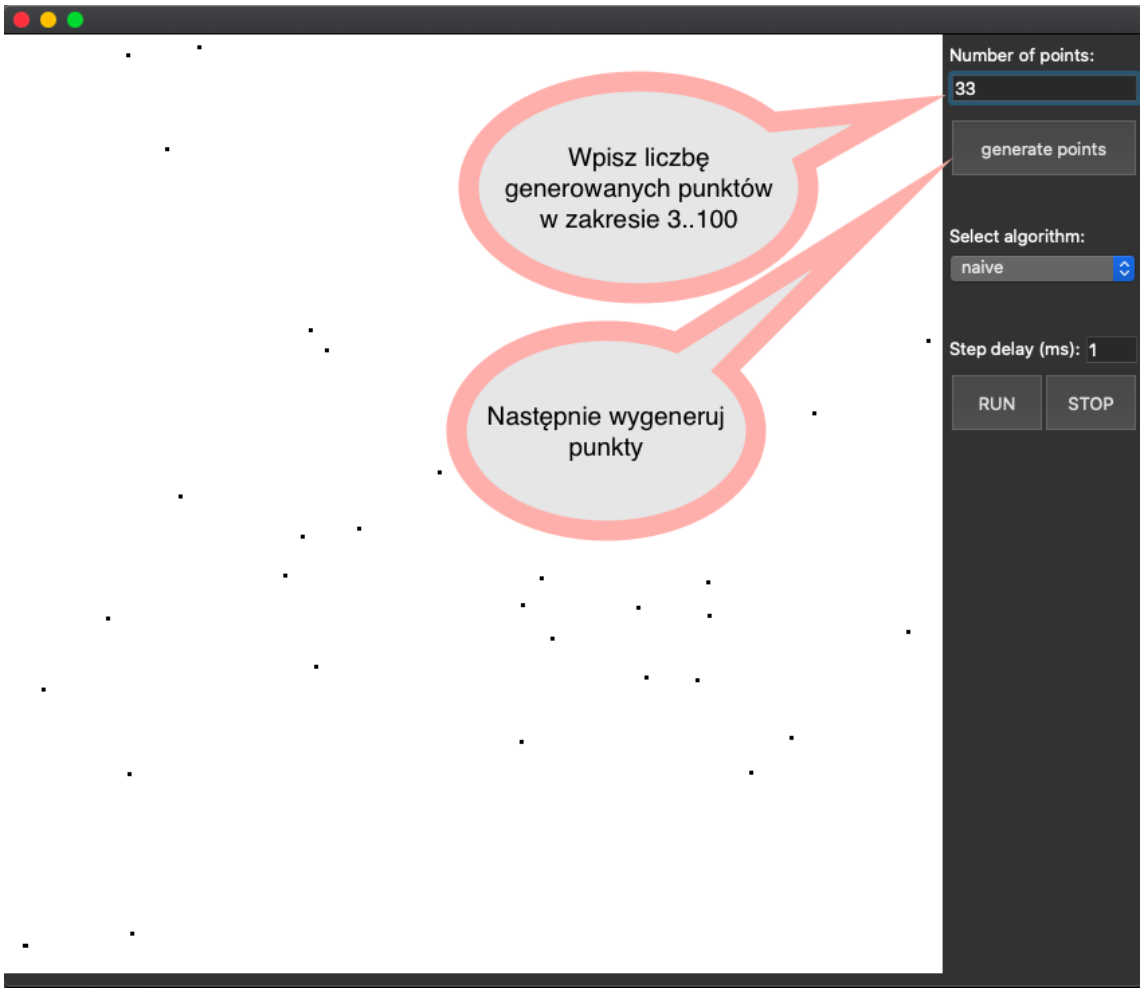
Uruchomienie programu z folderu `mw_triangulation_gui`:

```
python3 __run__.py
```

albo za pomocą skryptu powłoki Bash

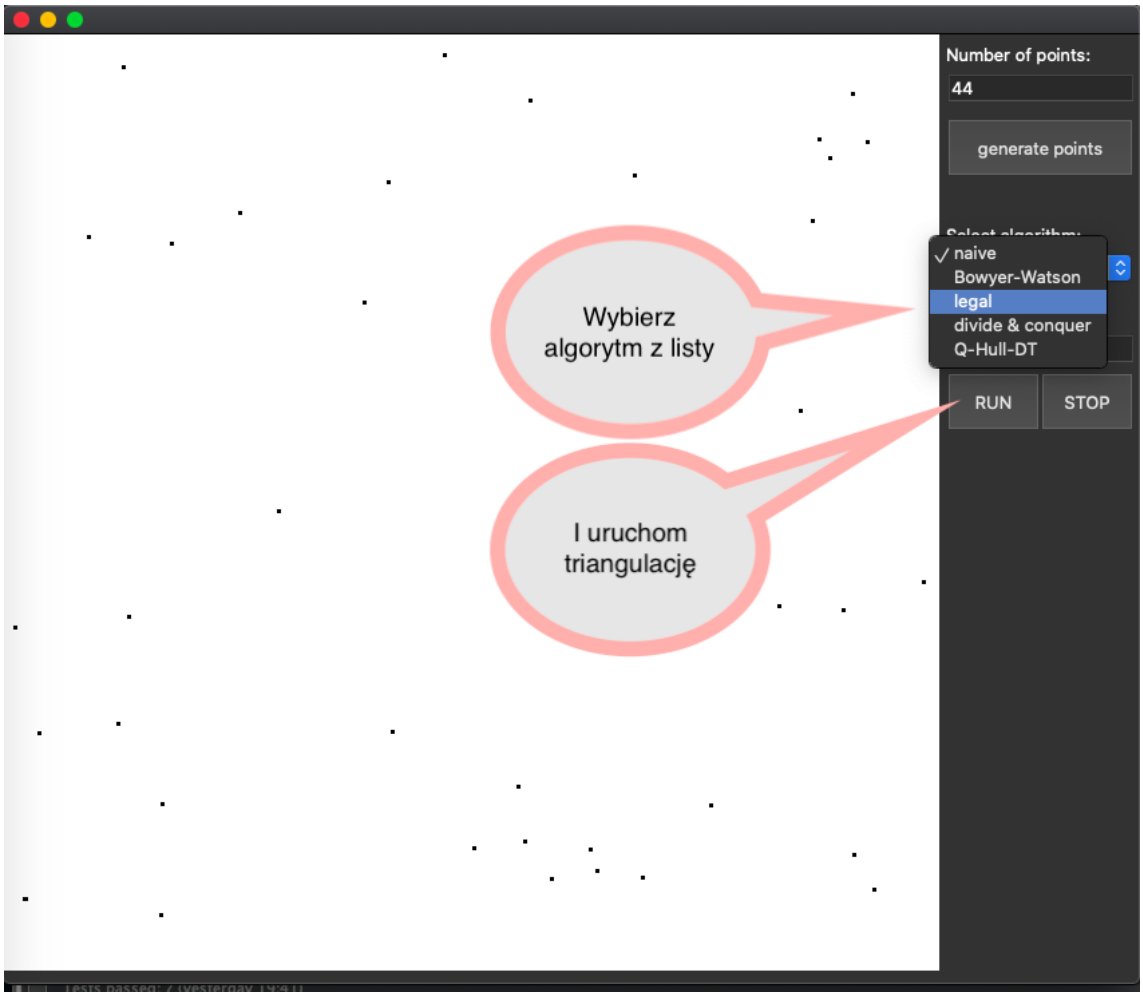
```
./run_gui.sh
```

Przed wyborem i uruchomieniam algorytmu należy wygenerować listę punktów, podając ich ilość i kilkając "generate points".



(b) Interfejs: generowanie punktów.

Nastepnie należy wybrać z listy algorytm, który wygeneruje triangulację i nacisnąć "RUN".

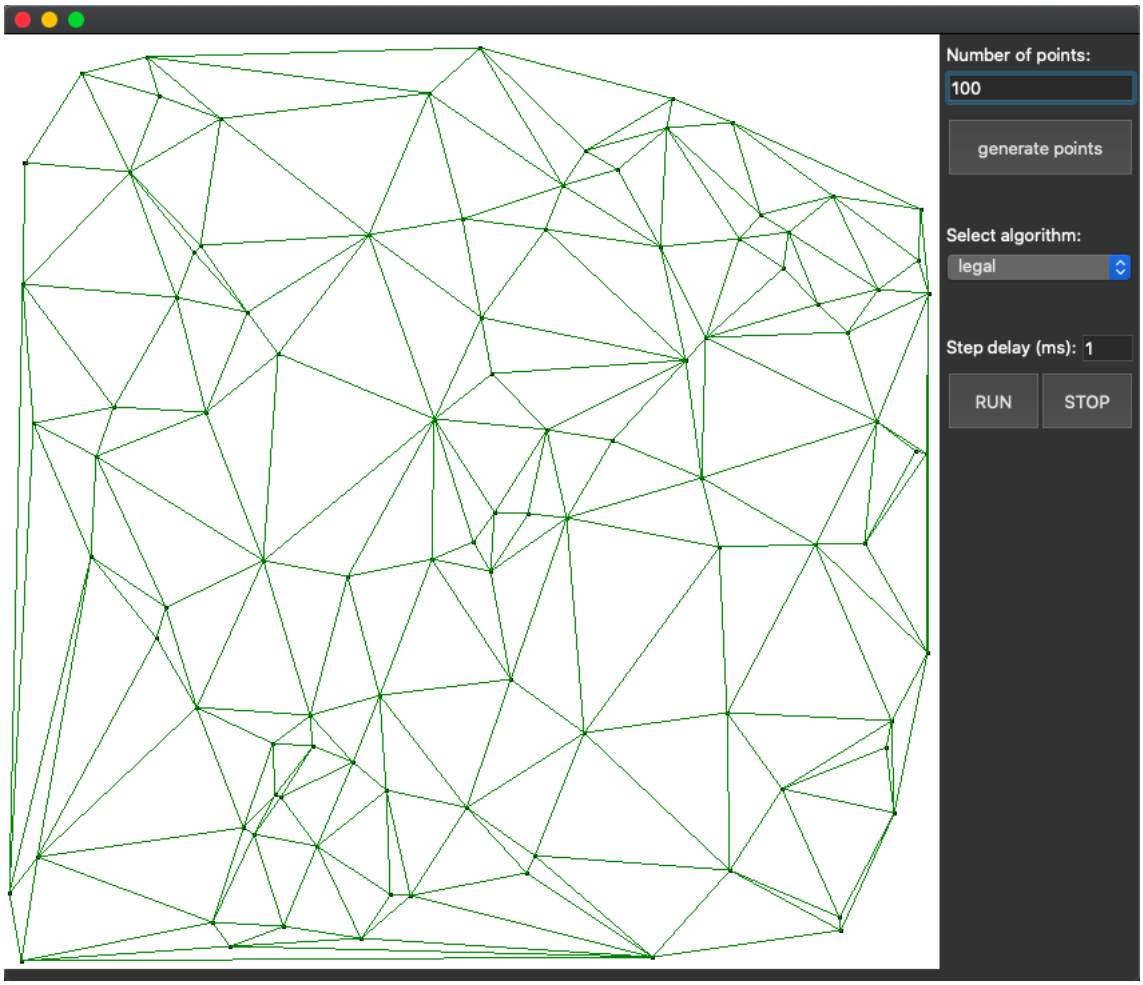


(c) Interfejs: wprowadzenie danych triangulacji.

Dodatkowe funkcje:

— Delay: umożliwia podanie opóźnienia kroków algorytmu (w milisekundach) dla dokładniejszej analizy przebiegu triangulacji.
— Stop: umożliwia zatrzymanie algorytmu.

Gotowa triangulacja jest widoczna w oknie pixmapy.



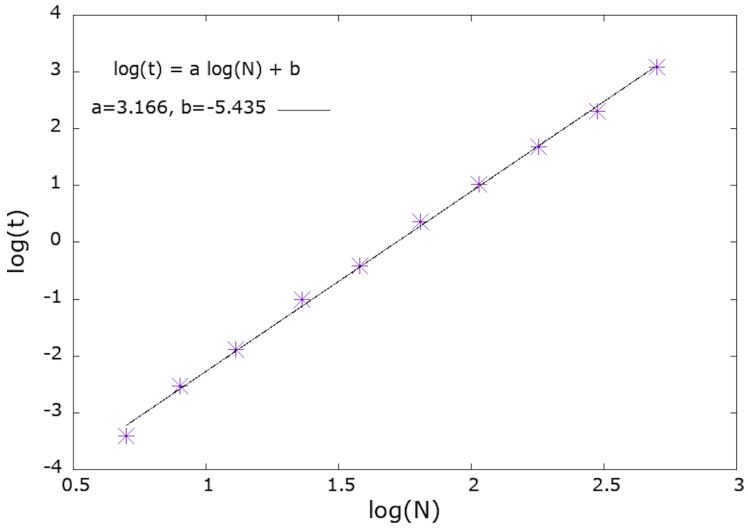
(d) Interfejs: gotowa triangulacja.

5. Testowanie

W tym rozdziale opracowane są wyniki testów złożoności algorytmów. Wynikiem testu wydajności jest wykres zależności czasu triangulacji od liczności zbioru punktów triangulacji. Każdy algorytm testowany był na zestawie dziesięciu zbiorów punktów losowo wybranych i unikalnych, od 5-punktowego do zbioru o maksymalnej założonej liczności (liczności kolejnych zbiorów w progresji logarytmicznej). Maksymalna liczność zbiorów punktów była dobierana do wydajności algorytmu, od max. 500 dla algorytmu naiwnego, do 1 000 000 quickhull. Testy były przeprowadzone na komputerze z procesorem Intel Core i5 2.7GHz.

5.1. Naiwny

Teoretyczna złożoność algorytmu naiwnego wynosi $O(n^4)$. Współczynnik $a = 3.166$, co oznacza, że złożoność jest niższa. Wynika to z optymalizacji przechodzenia przez zbiór punktów; każda kolejna pętla iteruje po podzbiorze mniejszym, niż poprzednia. W praktyce, przetworzenie zbioru 500 punktów trwa około 17 minut, co wyklucza zastosowanie go przy przetwarzaniu licznych zbiorów danych.



Rysunek 5.1: Wyniki pomiarów złożoności algorytmu naiwnego.

5.2. Legal

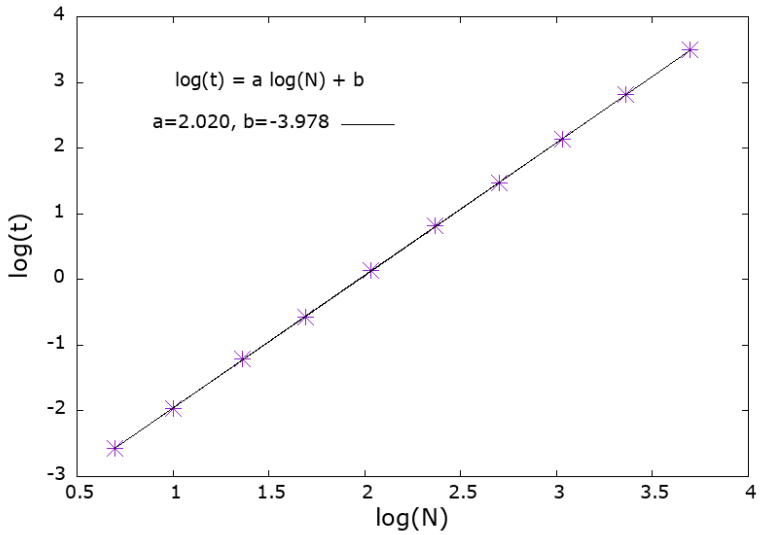
Powstały trzy wersje algorytmu z legalizacją krawędzi. Różnice dotyczą:

— Przechowywania trójkątów triangulacji,
— Lokalizacji trójkąta, w którym jest zawarty dodany punkt.

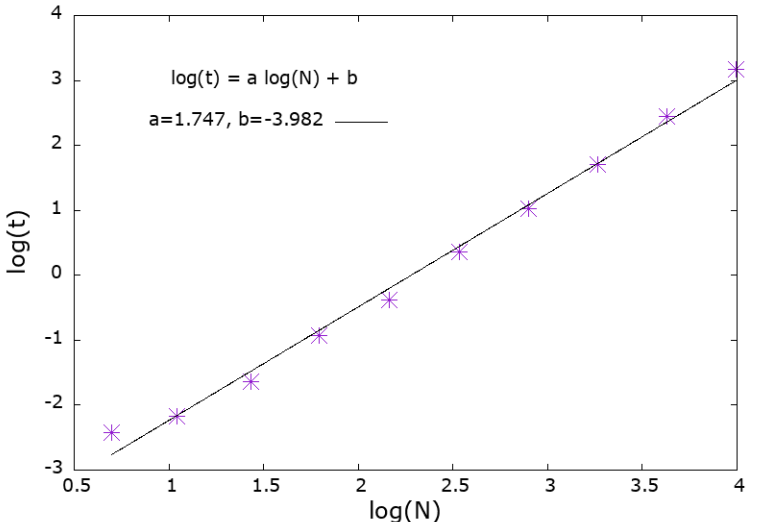
Pierwsza wersja (rys. 5.2) zawiera listę trójkątów: triangles collection. Powoduje to konieczność przeszukania wszystkich trójkątów dla znalezienia sąsiada w momencie legalizacji krawędzi dzielonej przez dwa trójkąty. W efekcie złożoność jest równa $O(n^2)$ - na wykresie współczynnik $a = 2.020$.

Kolejna wersja (rys. 5.3) zawiera kolekcję krawędzi, które przechowują danę do sąsiadujących trójkątów. Jest to optymalna struktura dla tego algorytmu, która usprawnia legalizację krawędzi. Dzięki niej złożoność jest stabilnie niższa niż $O(n^2)$, co widać na wykresie poniżej: $a = 1.747$.

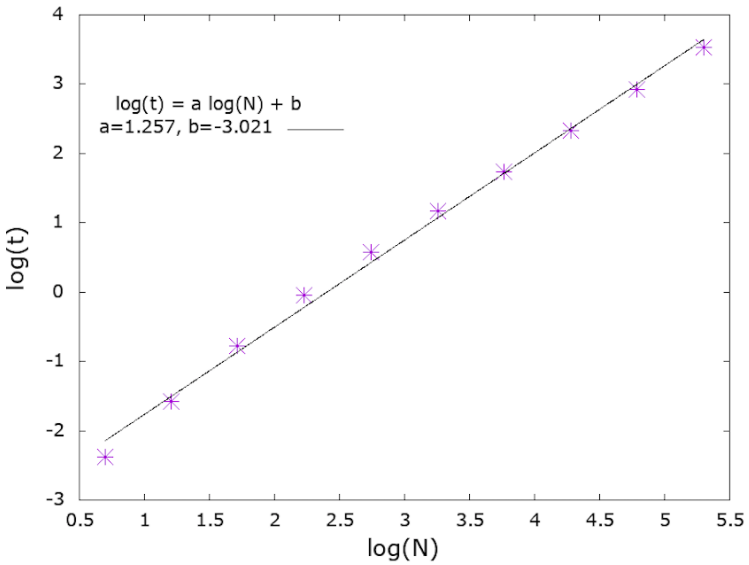
Ostatnia optymalizacja (rys. 5.4) w znaczny sposób przyspieszyła przeszukianie trójkątów w celu znalezienia tego, w którym znajduje się dodany punkt. Dzięki niej, złożoność jest blista tej oczekiwanej, czyli $O(n \log n)$; współczynnik $a = 1.315$.



Rysunek 5.2: Wyniki pomiarów złożoności algorytmu bez drzewa historii i z kolekcją trójkątów.



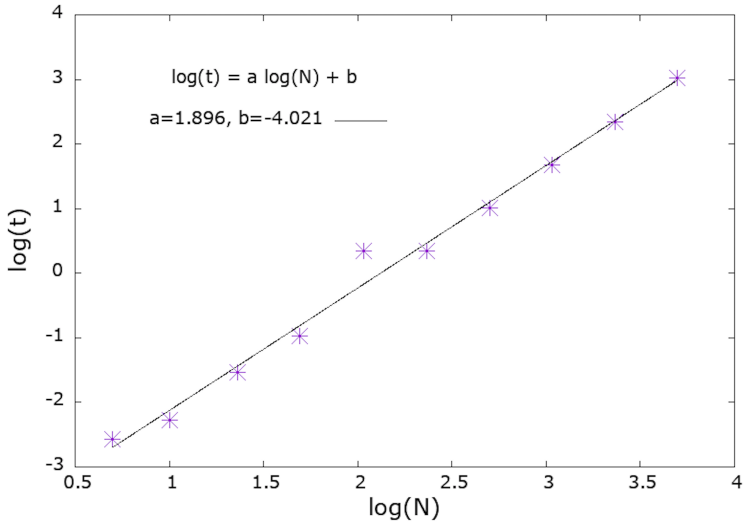
Rysunek 5.3: Wyniki pomiarów złożoności algorytmu z użyciem struktury danych `triangle_pairs` i bez drzewa historii.



Rysunek 5.4: Wyniki pomiarów złożoności algorytmu z użuciem struktury danych `triangle_pairs` i z drzewem historii.

5.3. Bowyer-Watson z triangles collection

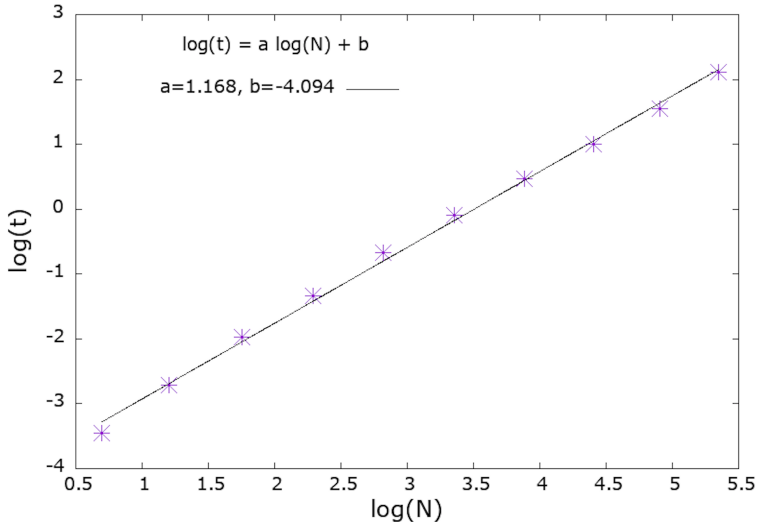
Algorytm Bowyera Watsona jest zaimplementowany z użyciem kolekcji trójkątów, tak samo jak w przypadku pierwszej wersji algorytmu z legalizacją krawędzi. Po testach złożoności wypadł on lepiej, niż wersja pierwsza algorytmu legal; współczynnik $a = 1.896$, czyli złożoność jest bliska $O(n^2)$.



Rysunek 5.5: Wyniki pomiarów złożoności algorytmu Bowyera-Watsona.

5.4. Divide and Conquer

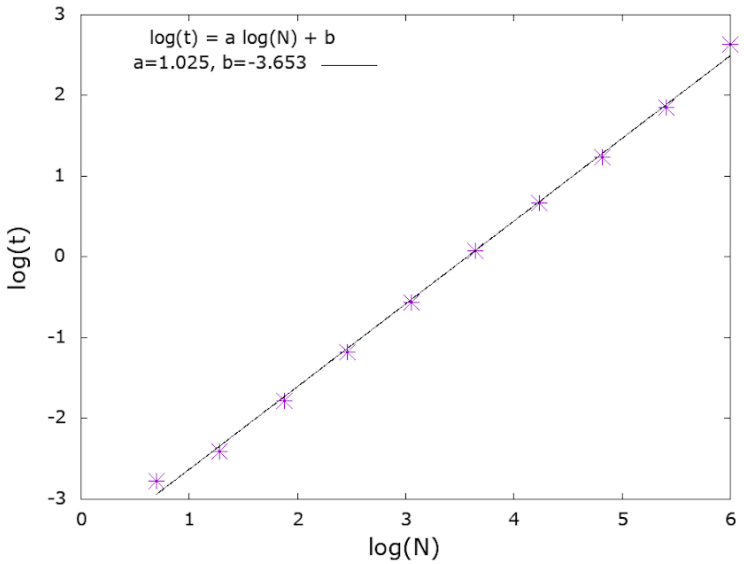
Algorytm jest zdecydowanie najszybszy z dotąd analizowanych. Współczynnik $a = 1.168$ wskazuje na złożoność $O(n \log n)$ (rys 5.6). Algorytm względnie szybko przechodzi przez liczne zbiory setek tysięcy punktów. Wersja implementacji korzysta z rekurencji, co skutkuje przepełnieniem stosu przy zbiorze liczącym powyżej pół miliona punktów (przy standardowej konfuguracji stosu w języku Python 3). Rozwiązaniem jest przepisanie programu na wersję bez rekurencji.



Rysunek 5.6: Wyniki pomiarów złożoności algorytmu dziel i zwyciężaj.

5.5. Quickhull dla triangulacji Delaunaya

Algorytm został testowany na zbiorze losowych zbiorów punktów bez powtórzeń. W sumie zostało utworzonych dziesięć zbiorów liczności od 10 do 1000000 losowych punktów. Współczynnik $a = 1.025$ wskazuje na złożoność nieznacznie większą od złożoności liniowej, $O(n \log n)$ (rys. 5.7). Algorytm szybko przechodzi przez liczne zbiory punktów.



Rysunek 5.7: Wyniki pomiarów złożoności algorytmu quickhull dla triangulacji Delaunaya.

6. Podsumowanie

W ramach pracy przygotowano implementację w języku Python pięciu algorytmów wyznaczających triangulację Delaunaya dla podanego zbioru punktów. Algorytm naiwny korzysta wprost z definicji triangulacji Delaunaya, ale z powodu słabej wydajności i punktów wejściowych w postaci ogólnej na ogół nie nadaje się do zastosowań. Algorytmy inkrementacyjne, takie jak metoda odwróceń Lawsona i algorytm Bowyera-Watsona są już bardziej wydajne. Następny algorytm Lee-Schachtera pokazuje z sukcesem zastosowanie techniki dziel i zwyciężaj do problemu triangulacji. Ostatni algorytm quickhull wykorzystuje nieoczekiwany związek pomiędzy triangulacją a otoczką wypukłą w większej liczbie wymiarów.

Na potrzeby triangulacji zaimplementowano specjalne struktury danych, takie jak kolekcja trójkątów, czy drzewo historii trójkątów. Powstały również implementacje przydatnych funkcji pomocniczych: do sprawdzania warunku Delaunaya, do inicjalizacji supertrójkąta, do wyznaczania kąta pomiędzy trzema punktami.

W celu lepszego zobrazowania triangulacji stworzono dwa mechanizmy uruchamiania algorytmów triangulacji. Interfejs wiersza poleceń pozwala wygenerować rysunki triangulacji. Interfejs graficzny pozwala wygodnie śledzić kolejne etapy algorytmów triangulacji.

Realizacja triangulacji Delaunaya nie jest w założeniach skomplikowana, natomiast istnieje szeroka gama podejść i struktur, z których możemy skorzystać przy implementacji. Warto zwrócić szczególą uwagę na poprawność wykonanej triangulacji oraz na jej szybkość, zwłaszcza, gdy chcemy przetworzyć ogromną ilość punktów. Ważnym zagadnieniem jest odporność implementacji na patologiczne ułożenie punktów, a to nie dla każdego algorytmu da się osiągnąć.

Każdy z algorytmów generuje triangulację. Można jej użyć np. do mapowania terenu. Na podstawie triangulacji można wygenerować diagram Voronoi i wykorzystać do planowania i logistyki. Niektóre z implementacji pozwalają na rozwinięcie do trzeciego wymiaru i zastosowanie np. w grafice komputerowej.

Bibliografia

[1] Wikipedia, Delaunay triangulation, 2020,
https://en.wikipedia.org/wiki/Delaunay_triangulation.

[2] Wikipedia, Convex hull, 2020,
https://en.wikipedia.org/wiki/Convex_hull.

[3] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Geometria obliczeniowa. Algorytmy i zastosowania*, WNT, Warszawa 2007.

[4] Franco P. Preparata, Michael Ian Shamos, *Geometria obliczeniowa. Wprowadzenie*, Helion, Gliwice 2003.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[6] Joseph O'Rourke, *Computational Geometry in C. Second Edition*, Series: Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1998.

[7] Python Programming Language - Official Website, 2020,
https://www.python.org/.

[8] Marcin Permus, *Algorytmy geometryczne w języku Python*, Uniwersytet Jagielloński, Kraków 2018.

[9] Wojciech Chrobak, *Technika zamiatania płaszczyzny*, Uniwersytet Jagielloński, Kraków 2019.

[10] Wikipedia, Voronoi diagram, 2020,
https://en.wikipedia.org/wiki/Voronoi_diagram.

[11] Wikipedia, Bowyer-Watson algorithm, 2020,
https://en.wikipedia.org/wiki/Bowyer%E2%80%93Watson_algorithm.

[12] Dave Mount, *CMSC 754: Lecture 13 Delaunay Triangulations: Incremental Construction*, Department of Computer Science, University of Maryland, 2020.
https://www.cs.umd.edu/class/spring2020/cmsc754/Lects/lect13-delaun-alg.pdf

[13] Divide and conquer Delaunay triangulation, Samuel Peterson, 2020.
http://www.geom.uiuc.edu/~samuelp/del_project.html#algorithms

[14] S-hull: a fast sweep-hull routine for Delaunay triangulation, David Sinclair, 2020.
http://s-hull.org.

[15] The Geometry Center Home Page: Qhull, 2020.
http://www.qhull.org

[16] Jean Gallier and Jocelyn Quaintance, *Aspects of Convex Geometry Polyhedra, Linear Programming, Shellings, Voronoi Diagrams, Delaunay Triangulations*, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, USA, 2017.
https://www.cis.upenn.edu/~jean/combtopol.pdf

[17] C. Bradford Barber, David P. Dobkin, Hannu Huhdanpaa, *The Quickhull Algorithm for Convex Hulls*, ACM Transactions on Mathematical Software 22, 469-483 (1992).

https://www.cise.ufl.edu/~ungor/courses/fall06/papers/QuickHull.pdf