Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Gabriela Mazur

Nr albumu: 1134766

Wielokąty monotoniczne w geometrii obliczeniowej

Praca licencjacka na kierunku Informatyka

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

Kraków 2021

Oświadczenie autora pracy

Świadom 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

Chciałabym bardzo podziękować Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za ogromną pomoc, wyrozumiałość oraz poświęcony czas, bez których napisanie poniższej pracy nie byłoby możliwe.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych algorytmów związanych z wielokątami monotonicznymi. Wielokąt jest monotoniczny względem prostej L, jeżeli każda prosta prostopadła do L przecina wielokąt co najwyżej dwa razy. Każdy wielokąt monotoniczny jest prosty, a każdy wielokąt wypukły jest monotoniczny względem dowolnego kierunku.

W pracy przedstawiono, działający w czasie liniowym, algorytm rozpoznawania wielokątów y-monotonicznych i x-monotonicznych, przy czym wyznaczane są dwa łańcuchy wierzchołków zgodne z kierunkiem monotoniczności. Te łańcuchy wierzchołków pozwalają na sprawdzanie należenia punktu do wielokąta w czasie logarytmicznym. Najistotniejszym elementem pracy była triangulacja wielokątów w czasie liniowym. Powstały dwie implementacje triangulacji wachlarzowej wielokąta wypukłego, gdzie dopuszczono istnienie kolejnych współliniowych wierzchołków. Przedstawiony został również algorytm triangulacji wielokąta y-monotonicznego.

W implementacji wykorzystano bibliotekę planegeometry, która udostępnia implementacje podstawowych obiektów geometrycznych, takich jak punkt, trójkąt i wielokąt. W klasie Polygon poprawiono metodę porównującą wielokąty, tak aby wynik nie zależał od wyboru pierwszego wierzchołka i orientacji wierzchołków. Algorytmy zostały przetestowane pod względem poprawności i rzeczywistej złożoności obliczeniowej.

Słowa kluczowe: wielokąty, wielokąty monotoniczne, wielokąty wypukłe, triangulacja wielokąta, triangulacja wachlarzowa, należenie punktu do wielokąta

English title: Monotone polygons in computational geometry

Abstract

Python implementation of selected algorithms for monotone polygons is presented. A polygone is monotone with respect to a straight line L, if every line orthogonal to L intersects this polygon at most twice. Every monotone polygon is simple and every convex polygon is monotone with respect to any straight line.

Linear time recognition algorithms for y-monotone and x-monotone polygons are presented, where boundaries are decomposed into two monotone polygonal chains. These chains allow us to answer point in polygon queries in logarithmic time. The most important element of the work is polygon triangulation. Two implementations of a fan triangulation of convex polygons are created, where neighboring collinear vertices are possible. The problem of an y-monotone polygon triangulation is also presented.

We used the planegometry library, because implementations of some basic geometric objects were available (points, triangles, polygons). The polygon comparison was improved and the result is correct regardless of the direction and order of points on the list. All algorithms were tested with respect to correctness and real computational complexity.

Keywords: polygons, monotone polygons, convex polygons, polygon triangulation, fan triangulation, point in polygon query

Spis treści

Spis rysunków

Listings

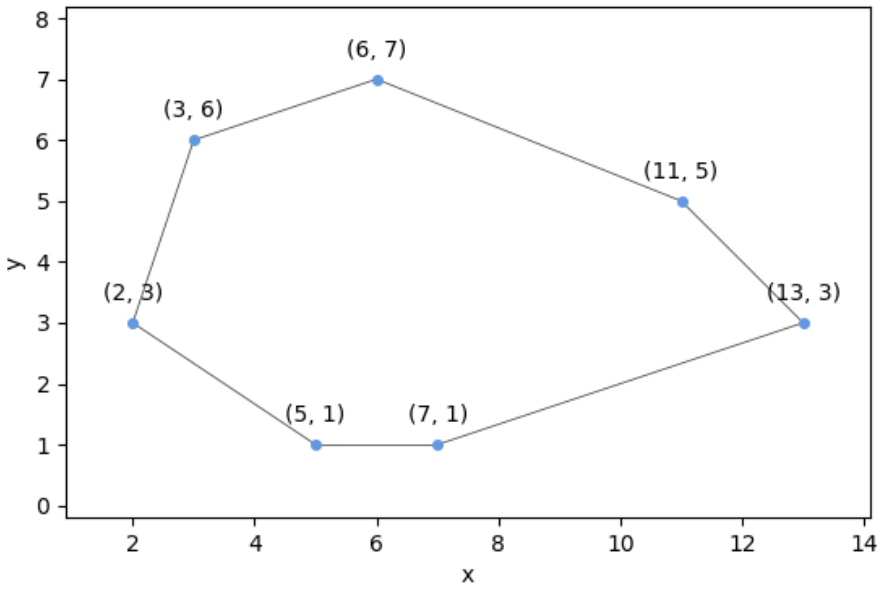
1. Wstęp

Tematem niniejszej pracy są wielokąty monotoniczne na płaszczyźnie [1]. Jest to ciekawa rodzina wielokątów, będąca ogniwem pośrednim między wielokątami wypukłymi, a ogólnymi wielokątami prostymi. Ścisłe definicje tych rodzin wielokątów zostaną podane w rozdziale 2, natomiast tutaj podamy przykłady wielokątów z kilku rodzin na rysunkach 1.1, 1.2, 1.3, 1.4. Wielokąt na rysunku 1.2 jest monotoniczny względem prostej pionowej (y-monotoniczny), ponieważ każda prosta pozioma przecina go co najwyżej w dwóch punktach.

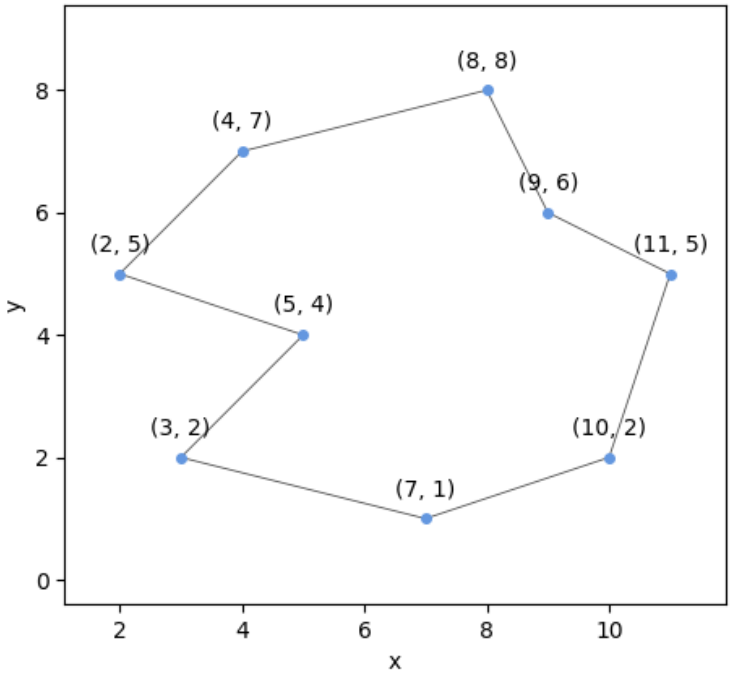
Wielokąty monotoniczne mają duże znaczenie w geometrii obliczeniowej ze względu na istnienie szybkich algorytmów dla pewnych problemów geometrycznych, takich jak triangulacja wielokąta, testowanie czy punkt należy do wielokąta, itp. W pewnych sytuacjach skomplikowane wielokąty lepiej jest podzielić na prostsze obiekty, które nie wymagają złożonych struktur danych. Ogólny wielokąt można podzielić na wielokąty wypukłe (np. trójkąty, trapezy), albo na mniejszą liczbę wielokątów monotonicznych.

Naszym celem jest zebranie dostępnych informacji o wielokątach monotonicznych, oraz implementacja wybranych algorytmów z nimi związanych. Do implementacji algorytmów został wybrany język Python [2] ze względu na czytelną składnię i bogatą bibliotekę standardową. Podstawowe obiekty geometryczne (punkt, odcinek, wielokąt, trójkąt) zostały wcześniej zaimplementowane w pakiecie *planegeometry* rozwijanym na Wydziale FAIS [3]. Niniejsza praca będzie wzbogaceniem pakietu o nowe funkcjonalności. W pracy korzystano z kilku podręczników poświęconych geometrii obliczeniowej, m.in. [4], [5], a także z innych materiałów.

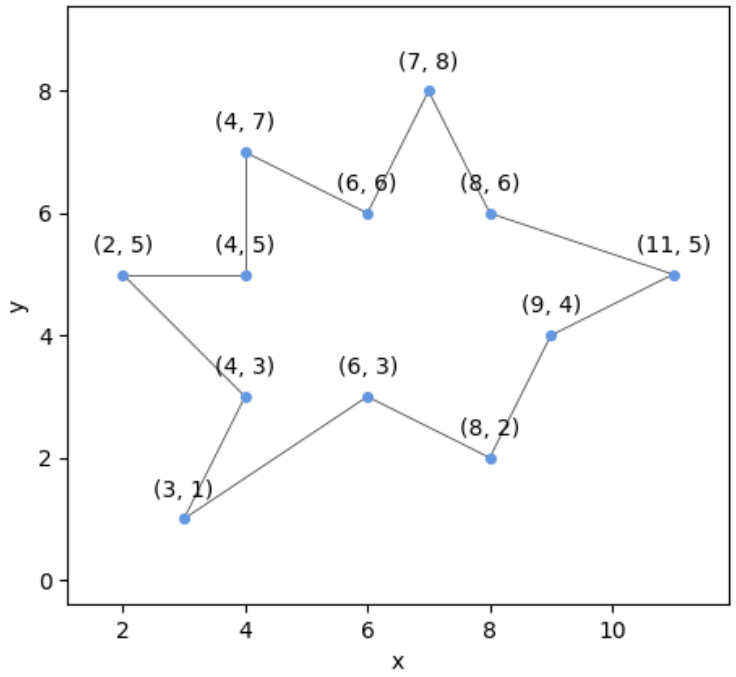
Praca została zorganizowana w następujący sposób. Rozdział 2 zawiera definicje potrzebnych obiektów i pojęć z geometrii obliczeniowej. Rozdział 3 skupia się na interfejsie użytkownika i sposobie korzystania z kodu. Rozdział 4 prezentuje algorytmy związane z wielokątami. Rozdział 5 podsumowuje pracę. Dodatek A zawiera testy przygotowanych algorytmów.



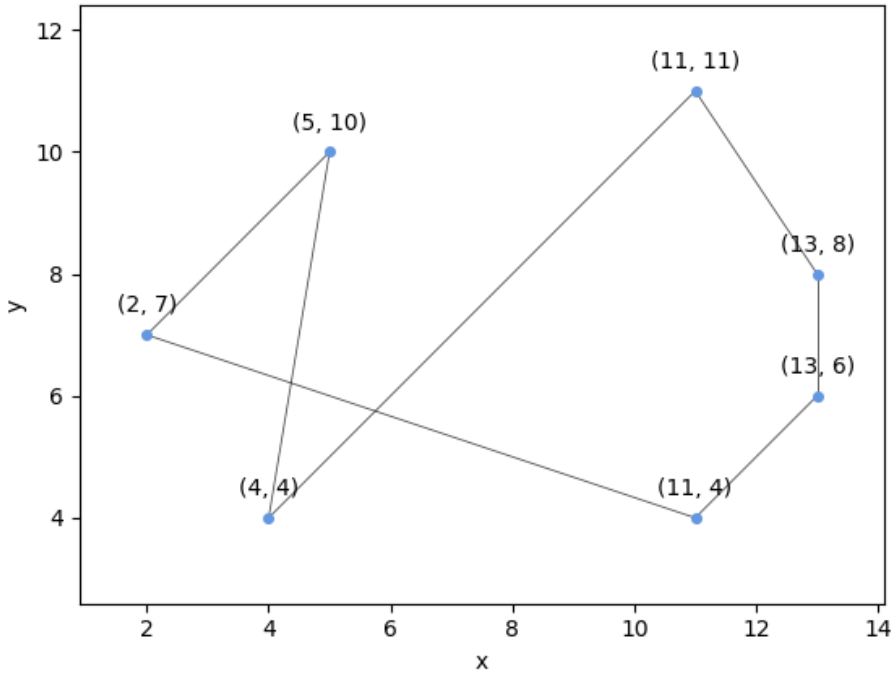
Rysunek 1.1. Wielokąt wypukły.



Rysunek 1.2. Wielokąt y-monotoniczny, ale nie wypukły.



Rysunek 1.3. Wielokąt prosty, ale nie monotoniczny.



Rysunek 1.4. Wielokąt złożony.

2. Geometria obliczeniowa

Rozdział zawiera podstawowe definicje i twierdzenia z geometrii obliczeniowej wykorzystywane w pracy.

2.1. Wielokąty

Definicja: *Wielokąt* (ang. *polygon*) jest to figura płaska ograniczona przez skończoną liczbę odcinków (boków, krawędzi) [6]. Koniec jednego odcinka jest początkiem następnego odcinka. Końce odcinków stykają się w wierzchołkach wielokąta.

Definicja: *Wielokąt prosty* (ang. *simple polygon*) jest to wielokąt, który nie ma przecinających się boków i nie ma dziur [7]. W wierzchołkach wielokąta spotykają się dokładnie dwie krawędzie. Jeżeli wielokąt ma n wierzchołków, to ma również n krawędzi.

Definicja: *Wielokąt wypukły* (ang. *convex polygon*) jest to wielokąt prosty, którego dwa dowolne punkty można połączyć odcinkiem zawartym w całości w wielokącie [8]. Część wspólna wielokątów wypukłych jest wielokątem wypukłym. Wielokąt wypukły nie może mieć kątów wewnętrznych rozwartych, co można sprawdzić w czasie $O(n)$. Jeżeli wszystkie kąty wewnętrzne są mniejsze od 180 stopni, to wielokąt jest ściśle wypukły. Jeżeli wielokąt prosty nie jest wypukły, to nazywamy go wklęsłym (ang. *concave*).

2.2. Wielokąty monotoniczne

Definicja: *Wielokąt monotoniczny* (ang. *monotone polygon*) jest to wielokąt P na płaszczyźnie, dla którego można wskazać prostą L taką, że każda prosta prostopadła do niej przecina wielokąt w najwyżej dwóch punktach (silna monotoniczność). Definicja może być rozszerzona na wielokąty posiadające krawędzie prostopadłe do L (słaba monotoniczność) [1].

W roku 1981 Preparata i Supowit opisali algorytm rozpoznawania, czy wielokąt prosty jest monotoniczny, przy czym algorytm znajdował wszystkie kierunki monotoniczności [9]. Algorytm opiera się na analizie ciągu kątów tworzonych przez kolejne skierowane boki wielokąta z pewnym wybranym kierunkiem na płaszczyźnie. Złożoność czsowa algorytmu wynosi $O(n)$.

2.3. Triangulacja wielokątów

Definicja: *Triangulacja wielokąta* (ang. *polygon triangulation*) jest to podział wielokąta prostego na trójkąty, przy czym wierzchołkami trójkątów są wierzchołki wielokąta, a dwa trójkąty mogą mieć wspólny tylko wierzchołek lub bok [10]. Każdy wielokąt prosty posiada triangulację. Jeżeli wielokąt ma n wierzchołków, to w wyniku triangulacji powstaną $n - 2$ trójkąty.

Definicja: *Triangulacja wachlarzowa* (ang. *fan triangulation*) polega na wyborze wierzchołka wielokąta i rysowaniu przekątnych do wszystkich innych niesąsiednich wierzchołków [11]. Nie każdy wielokąt można podzielić na trójkąty w ten sposób. Jest to możliwe przykladowo dla wielokątów wypukłych, wielokątów wklęsłych z jednym wklęsłym wierzchołkiem (z tego wierzchołka należy rysować cięciwy).

Triangulacja redukuje złożone kształty do kolekcji prostych trójkątów, które są łatwiejsze do przetwarzania. Stąd triangulacja jest pierwszym krokiem w wielu złożonych algorytmach.

Przykład: Strażnicy w galerii. Problem galerii sztuki przedstawiony został pierwszy raz przez Victora Klee. Przedstawia on wielokąt prosty jako galerie sztuki, w którym chcemy rozmieścić jak najmniejszą liczbę strażników, tak aby każda z części galerii była strzeżona. Wierzchołki wielokąta odpowiadają w tym przypadku strażnikom galerii.

3. Implementacja

Praca powstała dzięki wykorzystaniu biblioteki planegeometry rozwijanej na Wydziale FAIS Uniwersytetu Jagiellońskiego. W tym rozdziale przedstawimy struktury danych wykorzystane w pracy, oraz przykładowe użycia algorytmów.

3.1. Struktury danych

Tworząc algorytmy zawarte w tej pracy, korzystaliśmy z kilku struktur danych dostępnych w pakiecie planegeometry. Były one niezbędne i bardzo ułatwiły przechowywanie informacji potrzebnych w implementacji. Poniżej zostaną krótko opisane.

3.1.1. Punkt

Punkt jest to instancja klasy Point. Pozwala na przechowywanie dwóch wartości współrzędnych określających położenie w układzie kartezjańskim.

3.1.2. Trójkąt

Trójkąt jest to instancja klasy Triangle, przechowuje trzy obiekty klasy Point, które przedstawiają trzy wierzchołki trójkąta (nie mogą być one współliniowe).

3.1.3. Kolekcja trójkątów

Instancja klasy TriangleCollection zawiera pewną liczbę trójkątów (obiektów klasy Triangle). W pracy służyła nam do przechowywania trójkątów powstałych w wyniku triangulacji wielokąta.

3.1.4. Wielokąt

Wielokąt jest to instancja klasy Polygon. Wewnętrznie zawiera listę punktów (obiekty klasy Point), których musi być minimum trzy. W klasie ulepszono metodę porównującą wielokąty. Listy punktów reprezentujące wielokąty mogą być ułożone w różnych orientacjach i mogą zaczynać się od dowolnego wierzchołka.

Listing 3.1. Porównywanie wielokątów z modułu polygons.

```
>>> from polygons import Polygon
>>> from points import Point
# Stworzenie pierwszego wielokata.
>>> p1 = Polygon(Point(2, 3), Point(1, 4), Point(3, 8), Point(8, 9),
```

```
Point(8, 4), Point(5, 2), Point(3, 2))
# Stworzenie drugiego wielokata.
>>> p2 = Polygon(Point(8, 4), Point(8, 9), Point(3, 8), Point(1, 4),
Point(2, 3), Point(3, 2), Point(5, 2))
# Sprawdzenie, czy wielokaty sa takie same.
>>> p1 == p2
True
```

3.2. Przykładowe użycie algorytmów

Korzystając z sesji interaktywnych pokażemy, jak należy korzystać z przygotowanych implementacji algorytmów.

3.2.1. Rozpoznawanie wielokąta y-monotonicznego

Algorytm pozwala nam na sprawdzenie, czy wielokąt jest y-monotoniczny, a jeśli tak, to czy jest to słaba czy silna monotoniczność. Jednocześnie podczas działania algorytmu powstają dwie listy punktów, dzielące wielokąt na prawy i lewy łańcuch. Wielokąt testowany w tej sesji interaktywnej przedstawiony został na rysunku 4.1.

Listing 3.2. Korzystanie z klasy YMonotone z modułu ymonotone.

```
>>> from ymonotone import YMonotone
>>> from points import Point
# Stworzenie listy punktow wielokata.
>>> points = [Point(9, 4), Point(7, 6), Point(11, 7), Point(9, 8), Point(7, 11),
Point(5, 9), Point(3, 8), Point(4, 7), Point(2, 5), Point(6, 3), Point(8, 1), Point(10, 2)]
>>> ym = YMonotone(points)
# Uruchomienie algorytmu.
>>> ym.run()
# Sprawdzenie czy wielokat jest y-monotoniczny.
>>> ym.is_monotone
True
# Sprawdzenie czy jest to slaba monotonicznosc.
>>> ym.poor_monotonicity
False
# Lewy lancuch punktow.
>>> ym.left_chain
[Point(7, 11), Point(5, 9), Point(3, 8), Point(4, 7), Point(2, 5), Point(6, 3), Point(8, 1)]
# Prawy lancuch punktow.
>>> ym.right_chain
[Point(7, 11), Point(9, 8), Point(11, 7), Point(7, 6), Point(9, 4), Point(10, 2), Point(8, 1)]
```

3.2.2. Należenie punktu do wielokąta monotonicznego

Metoda wykorzystuje dwa łańcuchy (lewy oraz prawy) punktów wielokąta i sprawdza, czy dany punkt leży między nimi. Wielokąt testowany w tej sesji interaktywnej przedstawiony został na rysunku 4.4.

Listing 3.3. Użycie metody __contains__ z modułu ymonotone.

```
>>> from ymonotone import YMonotone
>>> from points import Point
# Stworzenie listy punktow wielokata.
>>> points = [Point(4, 5), Point(3, 6), Point(5, 8), Point(7, 7), Point(6, 6),
Point(7, 5), Point(8, 3), Point(6, 1), Point(2, 3), Point(2, 4)]
# Rozpoznanie wielokata y−monotonicznego.
>>> ym = YMonotone(points)
>>> ym.run()
# Sprawdzenie, czy punkt lezy wewnatrz wielokata.
>>> Point(6, 3) in ym
True
>>> Point(2, 7) in ym
False
```

3.2.3. Triangulacja wachlarzowa wielokąta wypukłego - pomijanie punktów współliniowych

Algorytm wykonuje triangulację wachlarzową wielokąta wypukłego. Sprawdza, czy wśród wierzchołków wielokąta znajdują się punkty współliniowe, jeśli tak, omija je. Trójkąty powstałe w wyniku triangulacji przechowywuje w TriangleCollection. Wielokąt, na którym została przeprowadzona triangulacja wachlarzowa z pominięciem punktów współliniowych, znajduje się na rysunku 4.5.

Listing 3.4. Klasa FanTriangulation2 z modułu fan_triangulation2.

```
>>> from fan_triangulation2 import FanTriangulation2
>>> from points import Point
# Stworzenie listy punktow wielokata.
>>> points = [Point(9, 2), Point(10, 3), Point(10, 5), Point(9, 6),
Point(8, 7), Point(7, 8), Point(5, 8), Point(3, 8),
Point(2, 4), Point(3, 3), Point(4, 2), Point(8, 1)]
>>> ft = FanTriangulation2(points)
# Uruchomienie algorytmu.
>>> ft.run()
# Kolekcja trojkatow powstalych w wyniku algorytmu.
>>> print(ft.tc)
{Triangle(10, 3, 2, 4, 4, 2), Triangle(10, 3, 7, 8, 3, 8), Triangle(10, 3, 3, 8, 2, 4),
Triangle(10, 3, 4, 2, 8, 1), Triangle(10, 3, 10, 5, 7, 8)}
```

3.2.4. Triangulacja wachlarzowa wielokąta wypukłego - uwzględnienie punktów współliniowych

Algorytm wykonuje triangulację wachlarzową wielokąta wypukłego. Jeśli na bokach wielokąta znajdują się punkty współliniowe, są one brane pod uwagę w triangulacji. Trójkąty powstałe w wyniku triangulacji przechowywuje w TriangleCollection. Wielokąt, na którym została przeprowadzona triangulacja wachlarzowa, znajduje się na rysunku 4.7.

Listing 3.5. Klasa FanTriangulation1 z modułu fan_triangulation1.

```
>>> from fan_triangulation1 import FanTriangulation1
>>> from points import Point
# Stworzenie listy punktow wielokata.
>>> points = [Point(15, 2), Point(12, 4), Point(9, 6), Point(6, 8),
Point(3, 10), Point(3, 2), Point(10, 2), Point(12, 2)]
>>> ft = FanTriangulation1(points)
# Uruchomienie algorytmu.
>>> ft.run()
# Kolekcja trojkatow powstalych w wyniku algorytmu.
>>> print(ft.tc)
{Triangle(6, 8, 3, 10, 3, 2), Triangle(12, 4, 10, 2, 12, 2), Triangle(12, 4, 9, 6, 3, 2),
Triangle(9, 6, 6, 8, 3, 2), Triangle(12, 4, 3, 10, 3, 2), Triangle(12, 4, 12, 2, 15, 2),
Triangle(12, 4, 3, 2, 10, 2)}
```

3.2.5. Triangulacja wielokąta y-monotonicznego

Algorytm wykonuje triangulację wielokąta y-monotonicznego. Trójkąty powstałe w wyniku triangulacji przechowywuje w TriangleCollection. Wielokąt, na którym została przeprowadzona triangulacja, jest przedstawiony na rysunku 4.8.

Listing 3.6. Klasa MonotoneTriangulation z modułu triangulation1.

```
>>> from triangulation1 import MonotoneTriangulation
>>> from points import Point
# Stworzenie listy punktow wielokata.
>>> points = [Point(0, 0), Point(7, 1), Point(7, 4), Point(5, 6),
Point(4, 7), Point(3, 9), Point(7, 11), Point(6, 14), Point(3, 17),
Point(4, 14), Point(3, 12), Point(1, 10), Point(0, 8)]
>>> mt = MonotoneTriangulation(points)
# Uruchomienie algorytmu.
>>> mt.run()
# Kolekcja trojkatow powstalych w wyniku algorytmu.
>>> print(mt.tc)
{Triangle(4, 7, 0, 8, 3, 9), Triangle(7, 1, 7, 4, 0, 8), Triangle(7, 4, 5, 6, 0, 8),
Triangle(3, 9, 1, 10, 7, 11), Triangle(3, 12, 6, 14, 4, 14), Triangle(7, 11, 3, 12, 6, 14),
Triangle(5, 6, 4, 7, 0, 8), Triangle(1, 10, 7, 11, 3, 12), Triangle(6, 14, 4, 14, 3, 17),
Triangle(0, 8, 3, 9, 1, 10), Triangle(0, 0, 7, 1, 0, 8)}
```

4. Algorytmy

W tym rozdziale skupimy się na przedstawieniu kilku istotnych algorytmów w geometrii obliczeniowej. Pomagają nam one poznawać wielokąty i wykonywać na nich użyteczne operacje.

4.1. Ulepszenia pakietu planegeometry

Podczas pracy nad algorytmami działającymi na wielokątach powstało kilka ulepszeń kodu biblioteki planegeometry.

— Do struktury danych TriangleCollection dodano zabezpieczenie przed dwukrotnym dodaniem tego samego trójkąta do kolekcji. Ponadto strukturę oparto na zbiorach, a nie na listach, przez co wyszukiwanie i usuwanie danego trójkąta zajmuje stały czas $O(1)$.

— Wielokąty są reprezentowane przez klasę Polygon, która wewnętrznie przechowuje listę wierzchołków wielokąta. Pojawia się tu dowolność wyboru pierwszego wierzchołka wielokąta oraz kierunku obiegania wielokąta. Ta dowolność nie była pierwotnie uwzględniona w metodzie porównywania wielokątów (polygon1 == polygon2). Porównywanie wielokątów zostało poprawione.
Nowy algorytm porównywania wielokątów wstępnie sprawdza długości obu list L i M z punktami, a następnie znajduje położenie i punktu L[0] na liście M. W celu wykrycia orientacji drugiego wielokąta następuje porównanie punktu L[1] z punktami M[i−1] oraz M[i+1]. Dalej porównywane są następne punkty listy L z punktami listy M, zgodnie z orientacją. W przypadku wykrycia niezgodności na jednym z etapów algorytmu, od razu zwracana jest wartość False. W przeciwnym wypadku, zwracana jest wartość True.

Listing 4.1. Moduł polygons, porównywanie wielokątów.

```
def __eq__(self, other):
    """Comparison of polygons (polygon1 == polygon2)."""
    n = len(self.point_list)
    if len(other.point_list) != n:
        return False
    first_i = None

    for i in range(n):
        if self.point_list[0] == other.point_list[i]:
            first_i = i
            break
    if first_i is None:
        return False
```

```
if self.point_list[1] == other.point_list[(first_i+n-1) % n]:
    for i in range(2, n):
        if self.point_list[i] != other.point_list[(first_i+n-i) % n]:
            return False
elif self.point_list[1] == other.point_list[(first_i+1) % n]:
    for i in range(2, n):
        if self.point_list[i] != other.point_list[(first_i+i) % n]:
            return False
else:
    return False
return True
```

4.2. Rozpoznawanie wielokąta wypukłego

W celu sprawdzenia, czy wielokąt jest wypukły, wykorzystujemy metodę polygon.is_convex() z biblioteki planegeometry. Na wstępie metoda ta sprawdza, czy wielokąt jest prosty. Jeżeli tak, algorytm poprzez orientację sprawdza każdy z wierzchołków wielokąta. Gdy żaden z wierzchołków nie jest wklęsły, metoda zwraca True. Oznacza to, że wielokąt jest wypukły. Złożoność obliczeniowa algorytmu wynosi $O(n)$.

4.3. Rozpoznawanie wielokąta monotonicznego

Z definicji podanej na początku pracy wiemy, że wielokąt można nazwać monotonicznym, jeżeli istnieje kierunek monotoniczności. Takim kierunkiem monotoniczności nazywamy prostą L, a każda prosta do niej prostopadła przecina wielokąt w maksymalnie dwóch punktach. Taki przypadek jest nazywany silną monotonicznością. Może się również zdarzyć tak, że wielokąt posiada krawędź prostopadłą do prostej L. Wtedy wielokąt również nazywamy monotonicznym, ale jest to słaba monotoniczność.

Ze względu na to, że znalezienie kierunku monotoniczności dla dowolnego wielokąta jest trudnym zadaniem, w mojej pracy skupiłam się na monotoniczności względem osi X i Y. Tym sposobem, powstały dwa algorytmy sprawdzające, czy wielokąt jest y-monotoniczny lub x-monotoniczny lub nie jest monotoniczny względem żadnej z tych osi. Algorytm dla y-monotoniczności jest kluczowy w pracy, ponieważ zostanie on wykorzystany w kolejnych algorytmach omówionych w tym dokumencie. Poniżej opiszę szczegółowo ten problem.

4.3.1. Wielokąt y-monotoniczny

Dane wejściowe: Lista punktów wielokąta (obiekty klasy Point).

Dane wyjściowe: Dwie wartości boolean (pierwsza mówi nam, czy wielokąt jest monotoniczny, a druga, czy monotoniczność jest silna), oraz listy odpowiadające prawemu i lewemu łańcuchowi punktów wielokąta.

Problem: Sprawdzenie, czy wielokąt jest monotoniczny względem osi Y.

Opis algorytmu: Pierwszy z algorytmów skupia się na osi Y, jest ona dla nas w tym przypadku kierunkiem monotoniczności. Zależy nam na sprawdzeniu kolejnych wierzchołków wielokąta. Zaczynamy więc od znalezienia dwóch punktów - jednego położonego najwyżej (największa wartość y) oraz drugiego położonego najniżej (najniższa wartość y). Oczywiście może pojawić się sytuacja, w której więcej niż jeden punkt będzie miał określoną wartość y. W takim przypadku algorytm wybiera pierwszy w kolejności punkt spełniający podany warunek.

Po znalezieniu dwóch najbardziej wysuniętych punktów tworzymy dwie listy punktów, odpowiadające lewemu oraz prawemu łańcuchowi wierzchołków wielokąta. Początkiem obu łańcuchów jest punkt położony najwyżej. Następnie sprawdzamy oba łancuchy. Wartości y w każdym z łańcuchów nie powinny rosnąć wraz z poruszaniem się po nim. Jeżeli w jakimś przypadku wartość y punktu $p[i+1]$ będzie większa od wartości y punktu $p[i]$, mamy do czynienia z wielokątem, który nie jest monotoniczny względem osi Y.

Podczas sprawdzania listy punktów, algorytm porównując kolejne wartości y, znajduje również takie $p[i]$ i $p[i+1]$, których y są takie same. Oczywiście tylko jeżeli wielokąt takie posiada. W takiej sytuacji, zwracana po wykonaniu algorytmu wartość poor_monotonicity wynosi True.

Rozpatrzymy teraz trzy możliwe przypadki dla podanego problemu.

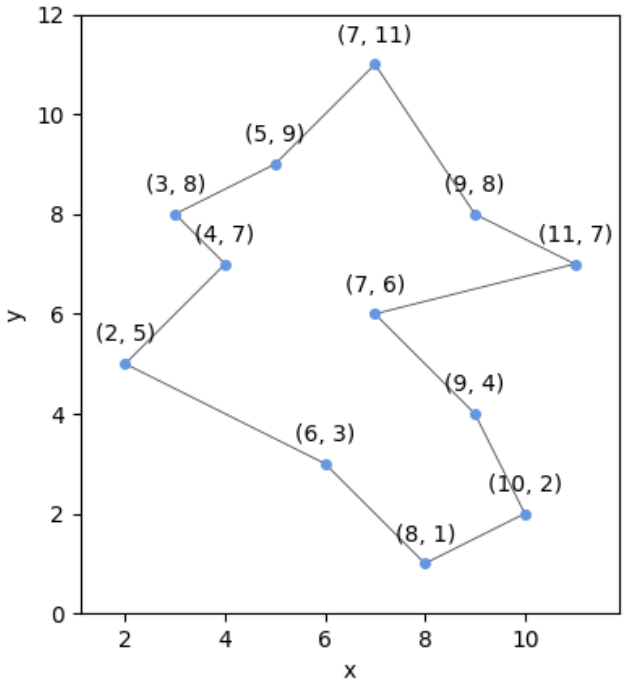
Pierwszy z nich to wielokąt silnie monotoniczny względem osi Y (rys. 4.1). Jak widzimy, nie posiada on dwóch kolejnych punktów takich, że wartości y dla $p[i]$ i $p[i+1]$ są sobie równe, więc jest to silna monotoniczność.

Kolejny wielokąt jest również monotoniczny, lecz jest to słaba monotoniczność. Jak widać na rysunku 4.2, posiada on dwie pary punktów, których wartości y są takie same. Gdybyśmy poprowadzili linie prostopadłe do osi y, dwie z tych linii pokryłyby dwie krawędzie tego wielokąta.

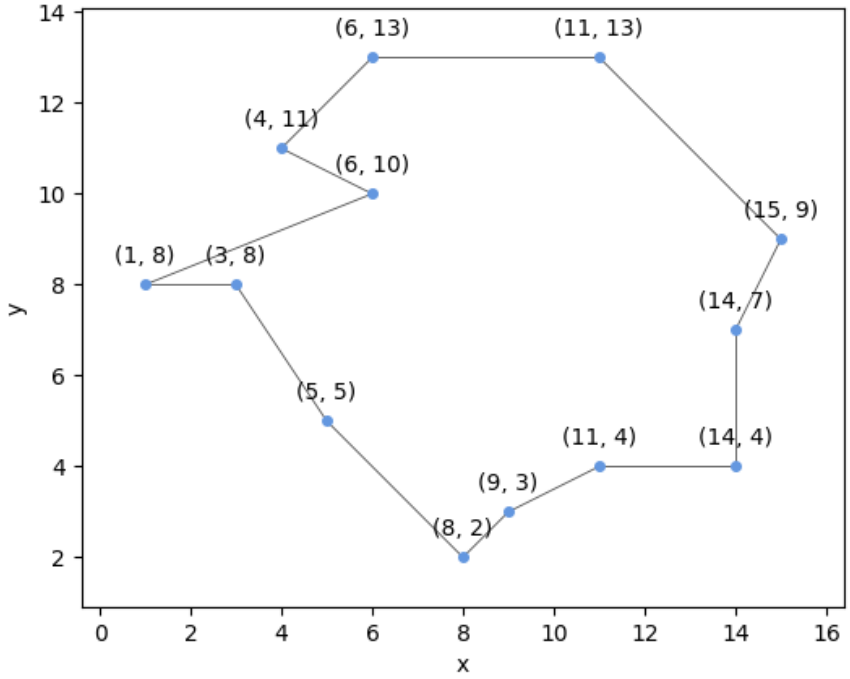
Ostatni jest wielokąt, który nie jest monotoniczny (rys. 4.3). Prowadząc linie prostopadłe do osi y, w kilku miejscach przecięłyby one wielokąt na czterech krawędziach.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n)$.

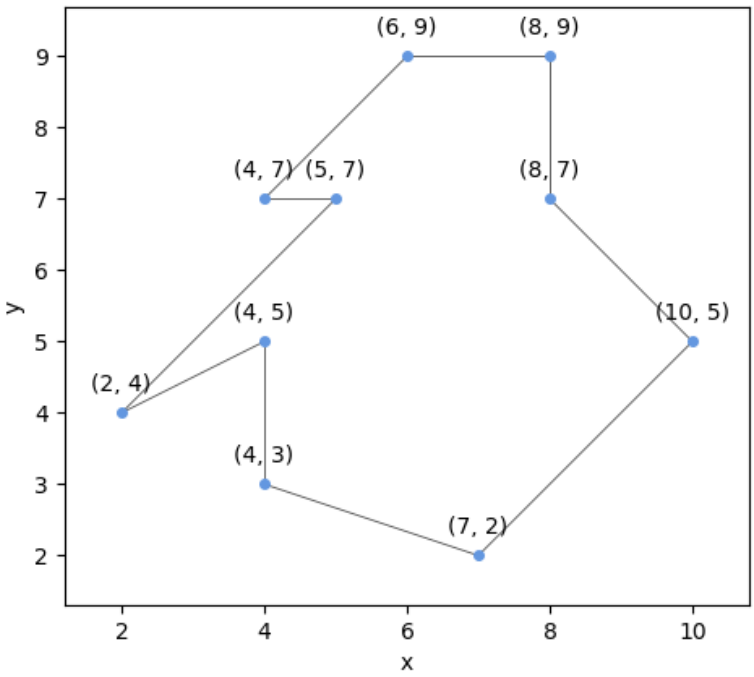
Uwagi: Opisany wyżej algorytm sprawdza monotoniczność jedynie względem jednego kierunku monotoniczności. Więc może istnieć inny kierunek monotoniczości, dla którego wielokąt określony przez nas jako niemonotoniczny jest monotoniczny. Dlatego idealnym rozwiązaniem jest znalezienie takiego kierunku monotoniczności, dla którego wielokąt jest monotoniczny.



Rysunek 4.1. Przykład wielokąta silnie monotonicznego względem osi Y.



Rysunek 4.2. Przykład wielokąta słabo monotonicznego względem osi Y.



Rysunek 4.3. Przykład wielokąta niemonotonicznego względem osi Y.

Listing 4.2. Moduł ymonotone.

```
#!/usr/bin/python

try:
    integer_types = (int, long)
except NameError: # Python 3
    integer_types = (int,)
    xrange = range

import bisect
from geomtools import orientation
from points import Point

class YMonotone:

    def __init__(self, point_list):
        if len(point_list) < 3:
            raise ValueError("minimum 3 points")
        self.point_list = point_list
        self.n = len(point_list)
        self.poor_monotonicity = False
        self.is_monotone = True
        self.left_chain = []
        self.right_chain = []
        self.sorted_left = []
        self.sorted_right = []
        self.keys_left = []
```

```
        self.keys_right = []

    def run(self):
        self.find_chains()
        self.check_monotonicity()
        self.sorted_left = self.left_chain.copy()
        self.sorted_right = self.right_chain.copy()
        self.sorted_left.reverse()
        self.sorted_right.reverse()
        self.keys_left = [p.y for p in self.sorted_left]
        self.keys_right = [p.y for p in self.sorted_right]

    def find_chains(self):
        i_max = max(xrange(self.n), key=lambda i:
        (self.point_list[i].y, self.point_list[i].x))
        i_min = min(xrange(self.n), key=lambda i:
        (self.point_list[i].y, self.point_list[i].x))

        for i in xrange(self.n):
            j = (i_max + i) % self.n
            self.left_chain.append(self.point_list[j])
            if j == i_min:
                break
        for i in xrange(self.n):
            j = (i_min + i) % self.n
            self.right_chain.append(self.point_list[j])
            if j == i_max:
                break
        self.right_chain.reverse()

        orient = orientation(self.right_chain[1], self.right_chain[0],
                             self.left_chain[1])
        if orient < 0:
            self.left_chain, self.right_chain = self.right_chain, self.left_chain

    def check_monotonicity(self):
        for i, point in enumerate(self.left_chain):
            if i == 0:
                continue
            if point.y == self.left_chain[i - 1].y:
                self.poor_monotonicity = True
            if point.y > self.left_chain[i - 1].y:
                self.is_monotone = False
        for i, point in enumerate(self.right_chain):
            if i == 0:
                continue
            if point.y == self.right_chain[i - 1].y:
                self.poor_monotonicity = True
            elif point.y > self.right_chain[i - 1].y:
                self.is_monotone = False
```

Podczas pracy został stworzony również algorytm rozpoznawania wielokąta x-monotonicznego. Ze względu na podobieństwo oraz identyczny schemat, jak w przypadku algorytmu sprawdzania y-monotoniczności, jego dokładny opis został pominięty.

4.4. Należenie punktu do wielokąta monotonicznego

Poniższy algorytm pozwoli nam sprawdzić, czy punkt należy do wielokąta y-monotonicznego. Podczas sprawdzania monotoniczności w poprzednim rozdziale, wyznaczyliśmy dwa łańcuchy punktów - lewy oraz prawy. Wykorzystamy je teraz do sprawdzenia położenia punktu względem wielokąta. Pozwoli to wykonać test w czasie $O(\log n)$.

Dane wejściowe: Dwa łańcuchy punktów wielokąta (prawy i lewy) oraz punkt, który chcemy sprawdzić.

Dane wyjściowe: Wartość True lub False mówiąca nam, czy punkt należy do wielokąta.

Problem: Sprawdzenie, czy dany punkt należy do wielokąta y-monotonicznego.

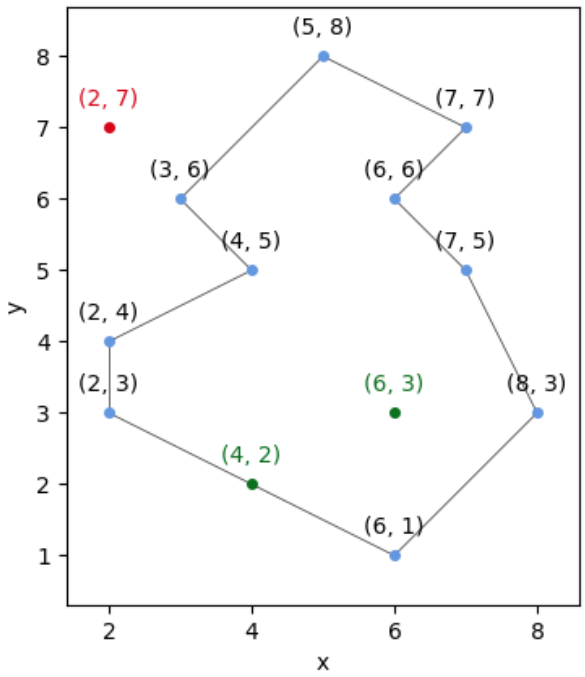
Opis algorytmu: Pierwszym krokiem algorytmu jest sprawdzenie, czy podany punkt leży ponad najwyżej położonym wierzchołkiem wielokąta, ewentualnie poniżej tego położonego na dole. Wystarczy zrobić porównanie z pierwszym i ostatnim punktem dowolnego łańcucha. W ten sposób szybko wykluczymy punkt leżący poza wielokątem.

Następnie wykorzystamy moduł bisect. Posłuży nam on do szybkiego sprawdzenia (w czasie logarytmicznym), na wysokości jakiej współrzędnej y leży testowany punkt. Moduł ten wykorzystuje posortowaną tablicę i sprawdza, w którym miejscu powinien być wstawiony element, aby zachować uporządkowaną postać tablicy. W obu łańcuchach znajdujemy pary sąsiednich punktów, których współrzędna y ogranicza współrzędną y danego punktu.

Podany punkt będzie należał do wielokąta, jeżeli znajduje się na prawo od lewego łańcucha i na lewo od prawego łańcucha. Można to sprawdzić badając orientację trójki punktów, dwóch punktów należących do łańcucha i podanego punktu.

Przykładowe położenie punktów wewnątrz wielokąta y-monotonicznego i poza nim, zostało przedstawione na rysunku 4.4.

Złożoność: Złożoność czasowa algorytmu wynosi $O(\log n)$. Kluczowe jest przygotowanie posortowanych list współrzędnych y punktów z obu łańcuchów na etapie rozpoznawania wielokąta monotonicznego, tak aby zapytanie o położenie punktu mogło korzystać jedynie z bisekcji.



Rysunek 4.4. Punkty leżące wewnątrz i poza wielokątem y-monotonicznym.

Listing 4.3. Moduł ymonotone_contains.

```
def __contains__(self, point):
    """Test if a point is in a polygon in O(log n) time."""
    if not isinstance(point, Point):
        raise ValueError("This is not a point.")

    if point.y > self.left_chain[0].y or point.y < self.left_chain[-1].y:
        return False

    li = bisect.bisect(self.keys_left, point.y)
    a = self.sorted_left[li]
    b = self.sorted_left[li - 1]
    if orientation(a, b, point) < 0:
            return False

    li = bisect.bisect(self.keys_right, point.y)
    a = self.sorted_right[li]
    b = self.sorted_right[li - 1]
    if orientation(a, b, point) > 0:
            return False

    return True
```

Algorytm sprawdzenia czy punkt należy do wielokąta został zaimplementowany również w wersji dla wielokąta x-monotonicznego. Jego dokładny opis został jednak pominięty w pracy ze względu na duże podobieństwo do powyższego rozwiązania.

4.5. Triangulacja wachlarzowa wielokąta wypukłego

Triangulacja wachlarzowa jest jedną z prostszych metod triangulacji wielokąta. Jednak może być ona stosowana jedynie w przypadku wielokątów wypukłych. Nasz algorytm pozwoli na sprawdzenie, czy wielokąt jest wypukły. Ta opcja jest możliwa do pominięcia poprzez przekazanie w argumencie funkcji wartości False. Następnie wykonujemy triangulację wachlarzową wielokąta.

Najistotniejsze w tym algorytmie jest rozpatrzenie wielokątów, które posiadają punkty współliniowe. Przygotowano dwie różne implementacje triangulacji. W pierwszej triangulacji punkty współliniowe zostaną pominięte, będzie to czysta triangulacja wachlarzowa. Natomiast w drugiej implementacji wszystkie punkty będą one uwzględnione w triangulacji i każdy punkt wielokąta będzie wierzchołkiem jakiegoś trójkąta w triangulacji.

4.5.1. Pomijanie punktów współliniowych

Dane wejściowe: Lista punktów wielokąta (obiekty klasy Point).

Dane wyjściowe: Zbiór trójkątów (obiekty klasy Triangle) przechowywanych wewnątrz instancji klasy TriangleCollection.

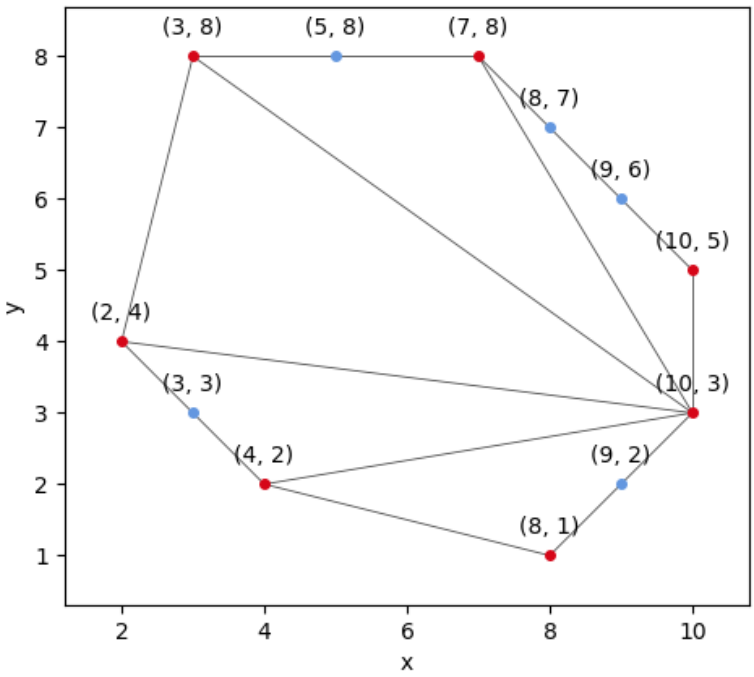
Problem: Wykonanie czystej triangulacji wachlarzowej wielokąta wypukłego przy wykluczeniu punktów leżących przy kątach wewnętrznych równych 180 stopni.

Opis algorytmu: W pierwszej implementacji algorytmu skupiamy się na sytuacji, w której wielokąt zawiera kąty wewnętrzne mniejsze oraz równe 180 stopni. W tym przypadku kąty pominiemy punkty leżące przy przy kątach wewnętrznych równych 180 stopni. Skupimy się wyłącznie na kątach wewnętrznych mniejszych niż 180 stopni.

Na początku, po wywołaniu metody run(), przeprowadzona jest wstępna walidacja danych wejściowych. Sprawdzamy, czy lista wprowadzonych punktów zawiera co najmniej trzy obiekty klasy Point. Opcjonalnie możemy sprawdzić również, czy wielokąt jest wypukły (warunek konieczny do przeprowadzenia triangulacji wachlarzowej). W obu przypadkach niepowodzenia zostanie rzucony wyjątek ValueError.

Kolejnym krokiem jest usunięcie z listy punktów współliniowych. Gdy lista jest już wyczyszczona, przechodzimy do wykonywania triangulacji. Tworzymy obiekt klasy TriangleCollection, w którym będą przechowywane powstałe w wyniku działania algorytmu trójkąty. Lista $p[]$ zawiera kolejne wierzchołki wielokąta. Pierwszy wierzchołek na liście $p[0]$ będzie bazą triangulacji wachlarzowej. Każdy powstały trójkąt przechowujemy w obiekcie klasy Triangle. W przypadku każdego trójkąta, pierwszym wierzchołkiem jest nasz punkt $p[0]$, czyli baza triangulacji.

Każdy trójkąt budujemy z trzech punktów: $p[0]$, $p[i+1]$, $p[i+2]$. Przyjmując n jako liczbę punktów na liście, poszukujemy $n-2$ trójkątów. Wynikiem



Rysunek 4.5. Triangulacja wachlarzowa z pominięciem punktów współliniowych.

naszego algorytmu jest kolekcja TriangleCollection, zawierająca powstałe trójkąty.

Powyższy algorytm przetestujemy dla wielokąta foremnego, zbudowanego z dwunastu punktów, przy czym pięć z nich jest punktami współliniowymi. Na rysunku 4.5 są one zaznaczone kolorem niebieskim. Pozostałe osiem punktów (zaznaczonych kolorem czerwonym) zostało użyte do triangulacji wachlarzowej i są one wierzchołkami trójkątów.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n)$.

Uwagi: Przedstawiony w tym rozdziale algorytm triangulacji wachlarzowej bierze pod uwagę jedynie najprostszy sposób podziału wielokąta. Pomija on punkty współliniowe, które czasem są równie istotne. Dlatego w kolejnej implementacji takie punkty również zostaną wykorzystane w algorytmie. Każdy punkt z listy będzie wierzchołkiem jakiegoś trójkąta.

Listing 4.4. Moduł fan_triangulation2.

```
#!/usr/bin/python

from polygons import Polygon
from trianglecollections import TriangleCollection
from triangles import Triangle
from geomtools import orientation
```

```
class FanTriangulation2:

    def __init__(self, points_list):
        if len(points_list) < 3:
            raise ValueError("Minimum 3 points.")
        self.points_list = points_list
        self.collinear_points = []
        self.points = []
        self.tc = TriangleCollection()

    def run(self, test_is_convex=True):
        """Checking if the polygon is convex."""
        if test_is_convex:
            polygon = Polygon(*self.points_list)
            if not polygon.is_convex():
                raise ValueError("The polygon is not convex.")
        return self.triangulation()

    def triangulation(self):
        """A fan triangulation of a convex polygon."""
        for i in range(len(self.points_list)):
            a = self.points_list[i % len(self.points_list)]
            b = self.points_list[(i + 1) % len(self.points_list)]
            c = self.points_list[(i + 2) % len(self.points_list)]
            if orientation(a, b, c) == 0:
                self.collinear_points.append(b)
            else:
                self.points.append(b)

        a = self.points[0]
        for number in range(len(self.points) - 2):
            b = self.points[number + 1]
            c = self.points[number + 2]
            triangle = Triangle(a, b, c)
            self.tc.insert(triangle)
```

4.5.2. Uwzględnienie punktów współliniowych

W tym rozdziale rozpatrzymy triangulację wielokąta, która uwzględnia punkty współliniowe. Każdy wierzchołek wielokąta będzie należał do co najmniej jednego z powstałych trójkątów. Przykładem figury będzie prostokąt, w którym dodano dodatkowe punkty na poszczególnych bokach. Innym przypadkiem będzie trójkąt z dodatkowymi punktami na bokach.

Dane wejściowe: Lista punktów wielokąta (obiekty klasy Point).

Dane wyjściowe: Zbiór trójkątów (obiekty klasy Triangle) przechowywanych wewnątrz instancji klasy TriangleCollection.

Problem: Wykonanie uogólnionej triangulacji wachlarzowej wielokąta wypukłego z uwzględnieniem punktów leżących przy kątach wewnętrznych równych 180 stopni.

Opis algorytmu: Pierwszym krokiem algorytmu jest sprawdzenie, czy wielokąt jest wypukły. Wielokąt posiadający chociaż jeden kąt wklęsły nie może być podzielony na trójkąty za pomocą triangulacji wachlarzowej.

Na wstępie sprawdzamy, czy wielokąt posiada jedynie trzy wierzchołki. Jeśli tak, to triangulacja kończy się na tym etapie i wynikiem jej, jest trójkąt o danych trzech wierzchołkach.

Trudnym przypadkiem takiej triangulacji jest wielokąt będący trójkątem, który posiada dodatkowe punkty współliniowe na jednym boku. W takiej sytuacji musimy znaleźć odpowiedni wierzchołek, z którego możemy rozpocząć algorytm triangulacji. Sprawdzamy więc kolejno po trzy punkty z listy i szukamy $p[i]$ takiego, że kąt $p[i]$ wynosi 180°, a kąt punktu $p[i-1]$ jest mniejszy niż 180°. Po znalezieniu takiego wierzchołka musimy przesunąć naszą listę punktów tak, aby pierwszy z nich był wyznaczonym wcześniej wierzchołkiem wielokąta $p[0]$. Jeżeli nie ma w wielokącie wierzchołka z kątem równym 180°, wtedy nie robimy żadnych zmian na tym etapie. Teraz możemy przejść do głównej części algorytmu.

Na początku rozpatrujemy punkty znajdujące się po prawej stronie od znalezionego wcześniej wierzchołka. Przyjmijmy, że rozmiar listy z punktami to points_size. Tworzymy trójkąty, gdzie wierzchołkiem a za każdym razem będzie $p[0]$. Natomiast punkty b oraz c to odpowiednio p[points_size − 1] i p[points_size − 2].

Robimy tak do momentu, aż orientacja punktów c, p[points_size − 3] oraz $p[0]$ będzie równa 0, czyli będą to punkty współliniowe. Wtedy przechodzimy do tworzenia trójkątów takich, że punktem a będzie za każdym razem zapamiętany z ostatniej iteracji punkt c - $p[j]$. Natomiast punkty b i c to kolejne punkty $p[j-1]$ oraz $p[j-2]$. W taki sposób otrzymujemy kolekcje trójkątów powstałych w wyniku triangulacji wachlarzowej.

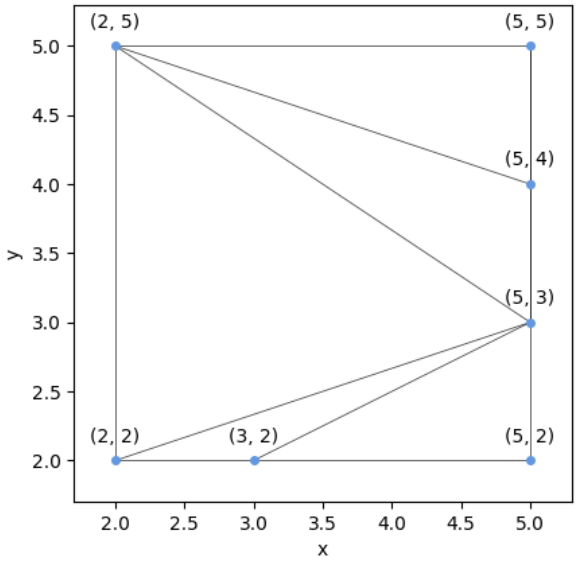
Na rysunku 4.6 przedstawiona została triangulacja nietypowego prostokąta, posiadającego dodatkowe punkty współliniowe na poszczególnych bokach. Natomiast na rysunku 4.7 triangulacja nietypowego trójkąta, również z dodatkowymi punktami współliniowymi na bokach.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n)$.

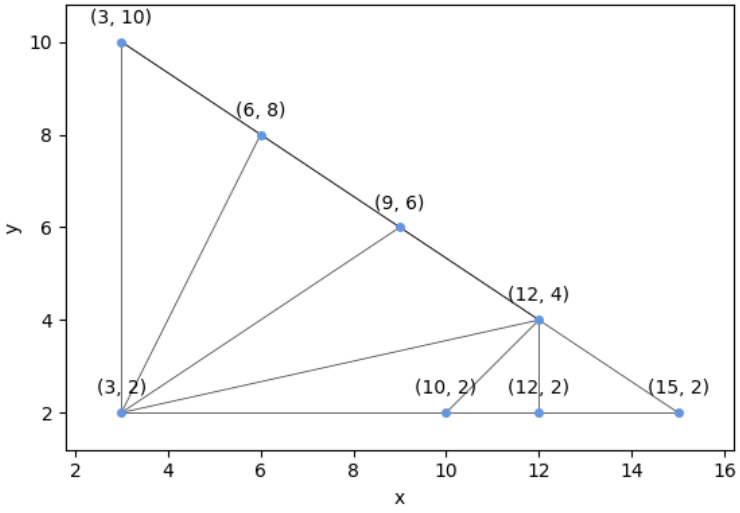
Listing 4.5. Moduł fan_triangulation1.

```
#!/usr/bin/python

from geomtools import orientation
from polygons import Polygon
from trianglecollections import TriangleCollection
from triangles import Triangle

class FanTriangulation1:

    def __init__(self, points_list):
        self.points = points_list
        self.tc = TriangleCollection()
        self.size = len(self.points)
        if len(points_list) < 3:
            raise ValueError("Minimum 3 points.")
```

```
def run(self, test_is_convex=True):
    if test_is_convex:
        polygon = Polygon(*self.points)
        if not polygon.is_convex():
            raise ValueError("The polygon is not convex.")
    return self.triangulation()

def triangulation(self):
    if self.size == 3:
        t = Triangle(*self.points)
        self.tc.insert(t)
        return

    idx = 0
    for i in range(self.size):
        if orientation(self.points[i],
                       self.points[(i + 1) % self.size],
                       self.points[(i + 2) % self.size]) == 0:
            idx = (i + 1) % self.size
            break

    p_list = list(self.points)
    while p_list[0] != self.points[idx]:
        p_list.append(p_list.pop(0))

    a = p_list[0]
    j = 0
    for i in range(self.size, 0, -1):
        if orientation(p_list[(i - 2) % self.size],
                       p_list[(i - 3) % self.size], a) != 0:
            b = p_list[(i - 1) % self.size]
            c = p_list[(i - 2) % self.size]
            j = i
            triangle = Triangle(a, b, c)
            self.tc.insert(triangle)
        else:
            break

    a = p_list[(j - 2) % self.size]
    for i in range(j - 3, 0, -1):
        b = p_list[i % self.size]
        c = p_list[(i - 1) % self.size]
        triangle = Triangle(a, b, c)
        self.tc.insert(triangle)
```



Rysunek 4.6. Triangulacja wachlarzowa prostokąta z dodatkowymi punktami współliniowymi na bokach.



Rysunek 4.7. Triangulacja wachlarzowa trójkąta z dodatkowymi punktami współliniowymi na bokach.

4.6. Triangulacja wielokąta monotonicznego

W tym rozdziale omówimy dokładnie algorytm triangulacji wielokąta y-monotonicznego. Kierunkiem monotoniczności w tym przypadku będzie oś Y. Wynikiem działania programu jest kolekcja trójkątów.

Dane wejściowe: Lista punktów wielokąta (obiekty klasy Point).

Dane wyjściowe: Zbiór trójkątów (obiekty klasy Triangle) przechowywanych wewnątrz instancji klasy TriangleCollection.

Problem: Triangulacja wielokąta monotonicznego (algorytm zachłanny).

Opis algorytmu: W algorytmie będziemy posługiwać się stosem, który posłuży nam do przechowywania punktów, na których aktualnie będziemy pracować.

Danymi wejściowymi jest lista punktów (obiekty klasy Point). Wywołujemy metodę run, która zawiera cały kod potrzebny do otrzymania rozwiązania. Najpierw szukamy dwóch najbardziej wysuniętych punktów wielokąta - jednego na górze (największa wartość y), drugiego na dole (najmniejsza wartość y).

Kolejnym krokiem jest podział listy punktów na dwa łańcuchy, lewy oraz prawy. Potrzebne są one do poruszania się po wielokącie w kolejnych etapach algorytmu. Od tego momentu wszystkie wierzchołki przechowujemy na jednej liście $p[]$, wraz z informacją do którego łańcucha należą. Lista została posortowana od największej do najmniejszej wartości y. W przypadku punktów leżących na tej samej współrzędnej y, są one segregowane według rosnącego x. Numerem 0 oznaczamy łańcuch lewy, a numerem 1 łańcuch prawy.

Następnie wrzucamy na stos dwa pierwsze punkty z listy: $p[0]$ i $p[1]$, oraz rozpoczynamy pętle po pozostałych indeksach listy (od drugiego). Do dalszych kroków algorytmu potrzebujemy dwóch punktów. Pierwszy z nich to punkt o aktualnym indeksie, natomiast drugim punktem jest ostatni element stosu.

Dalszy etap algorytmu opiera się na porównywaniu łańcuchów kolejnych par punktów. W jednym z przypadków punkty leżą na tych samych łańcuchach, a w drugim na różnych.

Teraz rozpatrzymy pierwszy z warunków: punkty leżą na różnych łańcuchach. Jeśli rozmiar stosu jest większy niż jeden, zdejmujemy jego ostatni element, który jest dla nas point2. Punkt pierwszy, czyli point1, to punkt o aktualnym indeksie iteracji. Ostatnim, trzecim punktem point3 jest końcowy element stosu. Z wyznaczonych trzech punktów powstaje jeden z trójkątów triangulacji.

Wierzchołek wielokąta z największą wartością y, czyli point2, został wykorzystany do zbudowania trójkąta i w takiej sytuacji nie może być wykorzystany kolejny raz do budowy kolejnego trójkąta. Zdejmujemy więc ten punkt ze stosu. Jeśli po tym kroku nasz stos jest pusty, wrzucamy na niego dwa punkty, tworzące diagonale (bok powstałego trójkąta).

Drugim warunkiem jest należenie punktów do tego samego łańcucha. Zdejmujemy ostatni punkt ze stosu i przechowujemy go w point2. Następnie ponownie potrzebujemy ostatniego elementu ze stosu, będzie to point3. W kolejnym kroku sprawdzamy orientację na podstawie łańcucha wyznaczonego na początku point1. Jeśli dany punkt leży na lewym łańcuchu, orientacja wynosi -1, gdy na prawym to $+1$. Teraz obliczamy orientację trzech wybranych wcześniej punktów: point1, point2 i point3, oraz porównujemy ją z orientacją dla point1.

Jeśli wyznaczone orientacje są różne, to z punktów: point1, point2 i point3 tworzymy trójkąt, a point2 zdejmujemy ze stosu, ponieważ zostaje on odcięty w wyniku stworzenia trójkąta. Nową diagonale tworzą punkty: point1 i point3. Jeśli w danym momencie algorytmu stos nie jest pusty, to do point3 przypisujemy wartość ostatniego punktu ze stosu. Następnie na stos wrzucamy point2 oraz punkt z listy o aktualnym indeksie pętli.

Ostatnim etapem algorytmu jest rozpatrywanie ostatniego punktu wielokąta, który jest dolnym zakończeniem obu łańcuchów punktów. W tej sytuacji point1 jest ostatni punkt z listy, a point2 zdjęty ze stosu końcowy element. Gdy wyznaczymy już te dwa punkty, sprawdzamy, czy na stosie nadal są jakieś elementy. Jeśli tak, przypisujemy do point3 ostatni zdjęty element ze stosu. Z tych wyznaczonych trzech punktów, powstaje ostatni trójkąt w triangulacji.

Na rysunku 4.8 przedstawiony został wielokąt y-monotoniczny, do triangulacji którego wszystkie kroki powyższego algorytmu zostały wykorzystane.

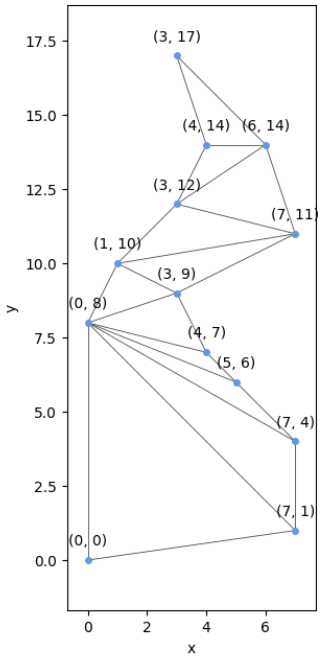
Złożoność: Złożoność czasowa algorytmu wynosi $O(n)$.

Listing 4.6. Moduł triangulation.

```
#!/usr/bin/python

try:
    integer_types = (int, long)
except NameError: # Python 3
    integer_types = (int,)
    xrange = range

from points import Point
from triangles import Triangle
from trianglecollections import TriangleCollection
from geomtools import orientation

class Chain:
    LEFT = 0
    RIGHT = 1

class MonotoneTriangulation:
    """Triangulation of a strictly y-monotone polygon in O(n) time."""

    def __init__(self, point_list):
        self.point_list = point_list
        self.n = len(self.point_list)
        self.tc = TriangleCollection()
        self.ulist = []
        self.stack = []
```

```
        self.i_max = None
        self.i_min = None

    def run(self):
        self.merge_chains()
        self.stack.append(self.ulist[0])
        self.stack.append(self.ulist[1])
        for i in xrange(2, self.n - 1):
            point1, chain1 = self.ulist[i]
            point2, chain2 = self.stack[-1]

            if chain1 != chain2:
                while len(self.stack) > 1:
                    point2, chain2 = self.stack.pop()
                    point3, chain3 = self.stack[-1]
                    triangle = Triangle(point1, point2, point3)
                    self.tc.insert(triangle)
                point2, chain2 = self.stack.pop()
                assert len(self.stack) == 0
                self.stack.append(self.ulist[i - 1])
                self.stack.append(self.ulist[i])

            else:
                point2, chain2 = self.stack.pop()
                point3, chain3 = self.stack[-1]
                if chain1 == Chain.LEFT:
                    orient = -1
                else:
                    orient = +1
                while orientation(point1, point2, point3) == orient:
                    triangle = Triangle(point1, point2, point3)
                    self.tc.insert(triangle)
                    point2, chain2 = self.stack.pop()
                    if len(self.stack) == 0:
                        break
                    else:
                        point3, chain3 = self.stack[-1]
                self.stack.append((point2, chain2))
                self.stack.append(self.ulist[i])

        assert len(self.stack) >= 2
        point1, chain1 = self.ulist[self.n - 1]
        point2, chain2 = self.stack.pop()
        while len(self.stack) > 0:
            point3, chain3 = self.stack.pop()
            triangle = Triangle(point1, point2, point3)
            self.tc.insert(triangle)
            point2 = point3

    def merge_chains(self):
        self.i_max = max(xrange(self.n), key=lambda i: (self.point_list[i].y))
        self.i_min = min(xrange(self.n), key=lambda i: (self.point_list[i].y))
        left_chain = []
        right_chain = []
        for i in xrange(self.n):
            j = (self.i_max + i) % self.n
            if j == self.i_min:
```

```
            break
        left_chain.append(self.point_list[j])
    for i in xrange(self.n):
        j = (self.i_min + i) % self.n
        if j == self.i_max:
            break
        right_chain.append(self.point_list[j])
    right_chain.reverse()
    self.ulist.extend((point, Chain.LEFT) for point in left_chain)
    self.ulist.extend((point, Chain.RIGHT) for point in right_chain)
    self.ulist.sort(key=lambda item: (item[0].y, -item[0].x), reverse=True)
```



Rysunek 4.8. Triangulacja wielokąta y-monotonicznego.

5. Podsumowanie

Celem powyższej pracy było przedstawienie wybranych algorytmów związanych z wielokątami monotonicznymi, ze względu na ich ciekawe właściwości i różne praktyczne zastosowania. Szczególnie istotne jest istnienie sybkich algorytmów rozwiązujących różne problemy z geometrii obliczeniowej.

W pracy przygotowano implementacje rozpoznające wielokąt y-monotoniczny i x-monotoniczny w czasie $O(n)$. Rozpoznawanie wiąże się z wyznaczeniem dwóch łańcuchów wierzchołków zgodnych z kierunkiem monotoniczności. Łańcuchy te pozwalają na sprawdzanie należenie punktu do wielokąta monotonicznego w czasie $O(\log n)$. Przy lokalizacji położenia badanego punktu względem łańcucha wykorzystuję się metodę bisekcji.

W kolejnych częściach pracy przedstawione zostały trzy sposoby triangulacji wielokąta. Dla każdej z poniższych triangulacji, czas działania jest rzędu $O(n)$.

— Pierwsza z nich to triangulacja wachlarzowa wielokąta wypukłego, która pomija punkty współliniowe leżące na bokach danej figury.
— Druga triangulacja to triangulacja wachlarzowa wielokąta wypukłego, uwzględniająca punkty współliniowe.
— Ostatnia implementacja pokazuje triangulację wielokąta y-monotonicznego.

Powstały w pracy kod w języku Python, został przetestowany pod względem poprawności zaimplementowanych algorytmów (moduł unittest) oraz czasu ich wykonywania (moduł timeit). Kod działa poprawnie w Pythonie 2.7 i Pythonie 3.

A. Testy algorytmów

W tym rodziale przedstawimy wyniki testów wydajnościowych, które zostały wykonane w celu potwierdzenia złożoności obliczeniowej omówionych wcześniej algorytmów. W opisach wyników testów n oznacza liczbę wierzchołków wielokąta.

A.1. Sprawdzanie monotoniczności wielokąta.

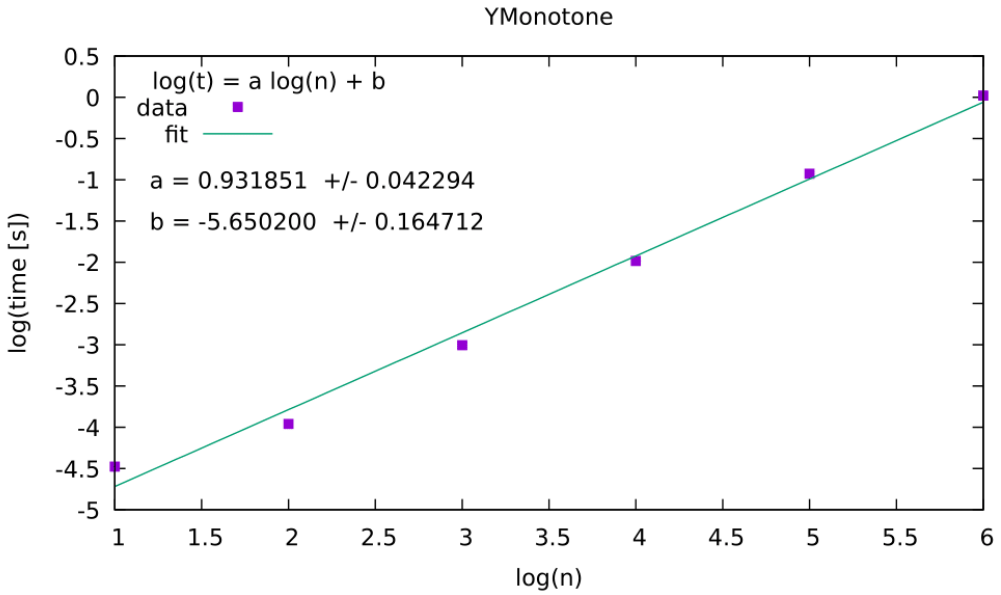
Testy wydajnościowe przeprowadzone z użyciem algorytmów sprawdzających, czy wielokąt jest y-monotoniczny lub x-monotoniczny, potwierdzają złożoność $O(n)$. Widzimy to na wykresach A.1 oraz A.2.

A.2. Triangulacja wachlarzowa wielokąta wypukłego.

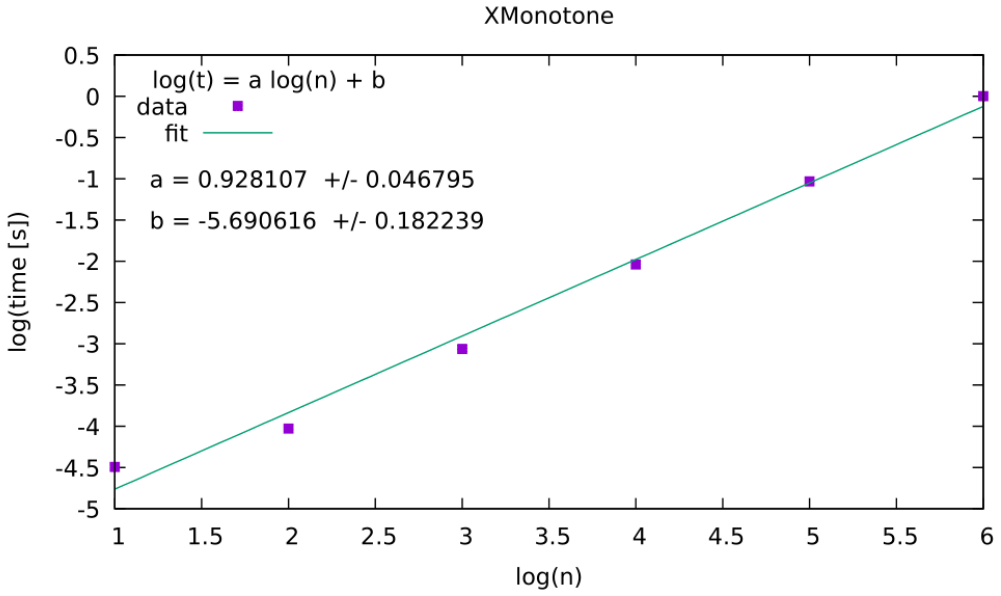
Testy wydajnościowe przeprowadzone z użyciem algorytmów wykonujących triangulację wachlarzową wielokąta wypukłego, potwierdzają złożoność $O(n)$. Widzimy to na wykresach A.3 oraz A.4.

A.3. Triangulacja wielokąta y-monotonicznego.

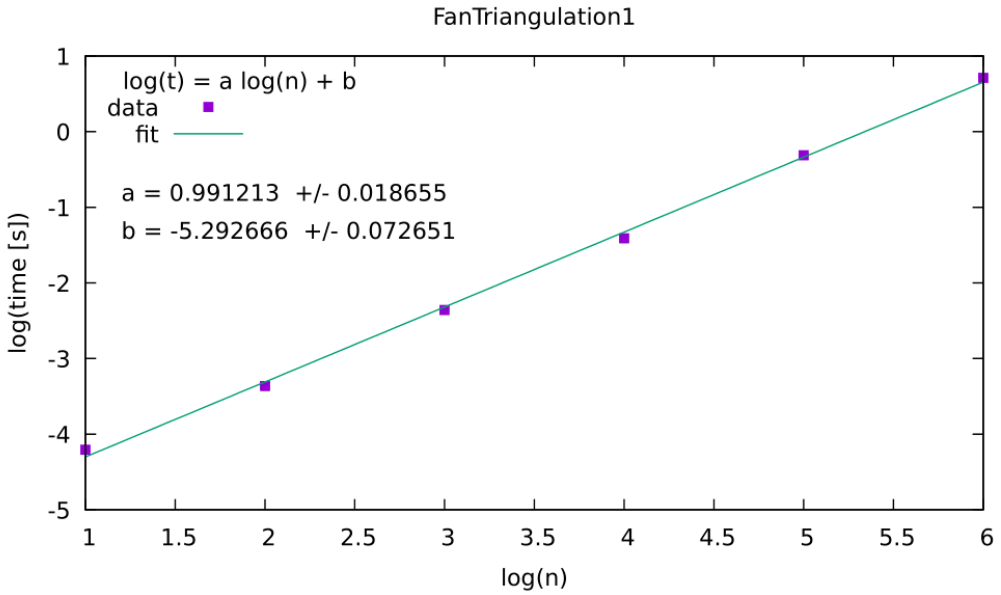
Testy wydajnościowe przeprowadzone z użyciem algorytmu wykonującego triangulację wielokąta y-monotonicznego, potwierdzają złożoność $O(n)$. Widzimy to na wykresie A.5.



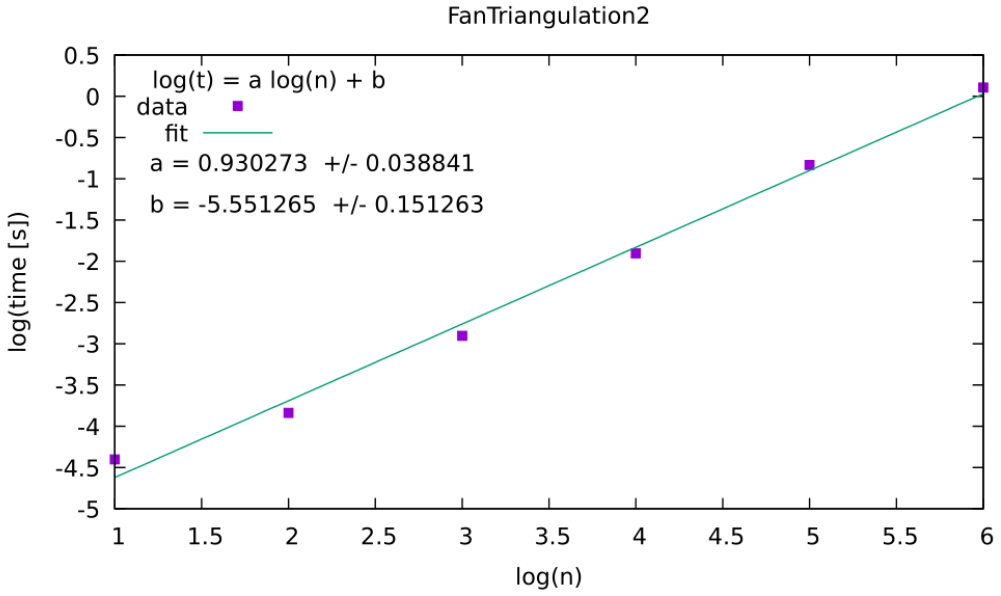
Rysunek A.1. Wykres wydajności algorytmu sprawdzania czy wielokąt jest y-monotoniczny. Współczynnik $a = 0.932(42)$ potwierdza złożoność implementacji $O(n)$.



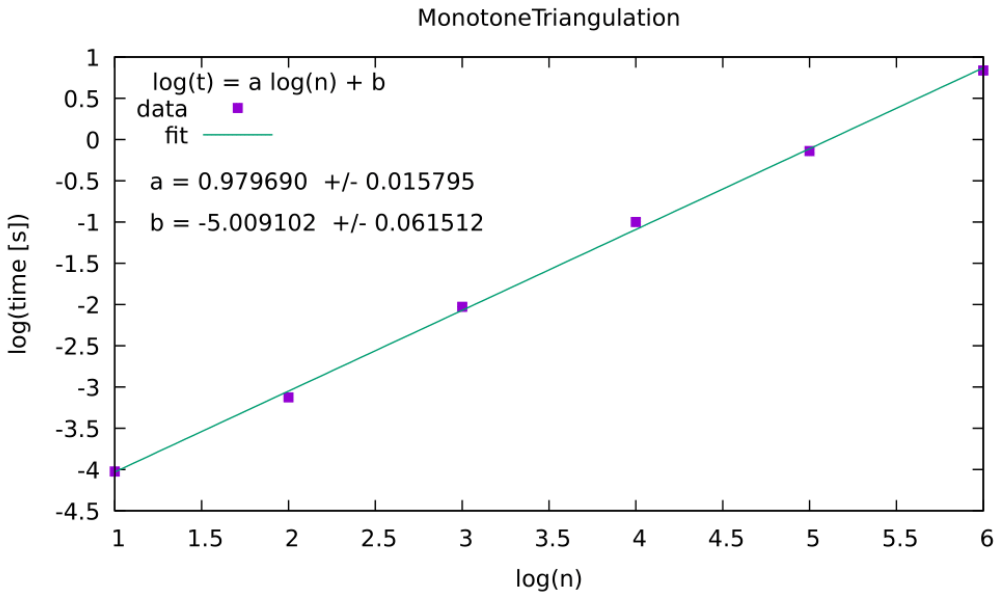
Rysunek A.2. Wykres wydajności algorytmu sprawdzania czy wielokąt jest x-monotoniczny. Współczynnik $a = 0.928(47)$ potwierdza złożoność implementacji $O(n)$.



Rysunek A.3. Wykres wydajności algorytmu triangulacji wachlarzowej z uwzględnieniem punktów współliniowych. Współczynnik $a = 0.991(19)$ potwierdza złożoność implementacji $O(n)$.



Rysunek A.4. Wykres wydajności algorytmu triangulacji wachlarzowej z pominięciem punktów współliniowych. Współczynnik $a = 0.930(39)$ potwierdza złożoność implementacji $O(n)$.



Rysunek A.5. Wykres wydajności algorytmu triangulacji wielokąta y-monotonicznego. Współczynnik $a = 0.980(16)$ potwierdza złożoność implementacji $O(n)$.

Bibliografia

[1] Wikipedia, Monotone polygon, 2021,
`https://en.wikipedia.org/wiki/Monotone_polygon`.

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

[3] Andrzej Kapanowski, *planegeometry*, GitHub repository, 2021,
`https://github.com/ufkapano/planegeometry`.

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

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

[6] Wikipedia, Polygon, 2021,
`https://en.wikipedia.org/wiki/Polygon`.

[7] Wikipedia, Simple polygon, 2021,
`https://en.wikipedia.org/wiki/Simple_polygon`.

[8] Wikipedia, Convex polygon, 2021,
`https://en.wikipedia.org/wiki/Convex_polygon`.

[9] Franco P. Preparata, Kenneth J. Supowit, *Testing a simple polygon for monotonicity*, Information Processing Letters 12 (4), 161-164 (1981).

[10] Wikipedia, Polygon triangulation, 2021,
`https://en.wikipedia.org/wiki/Polygon_triangulation`.

[11] Wikipedia, Fan triangulation, 2021,
`https://en.wikipedia.org/wiki/Fan_triangulation`.

[12] Python documentation, Bisect module, 2021,
`https://docs.python.org/3/library/bisect`.

[13] Waldemar Izdebski, Wyznaczenie położenia punktu względem odcinka, 2007/2008,
`http://www.izdebski.edu.pl/WykladySIT/WykladSIT_09.pdf`.

[14] Subhash Suri, Polygon triangulation, 2002,
`https://sites.cs.ucsb.edu/~suri/cs235/Triangulation.pdf`.