Triangulacja Delaunay'a

Projekt nr 6.

Wojciech Palczewski
Paweł Małkowski

8.12.2025

Spis treści

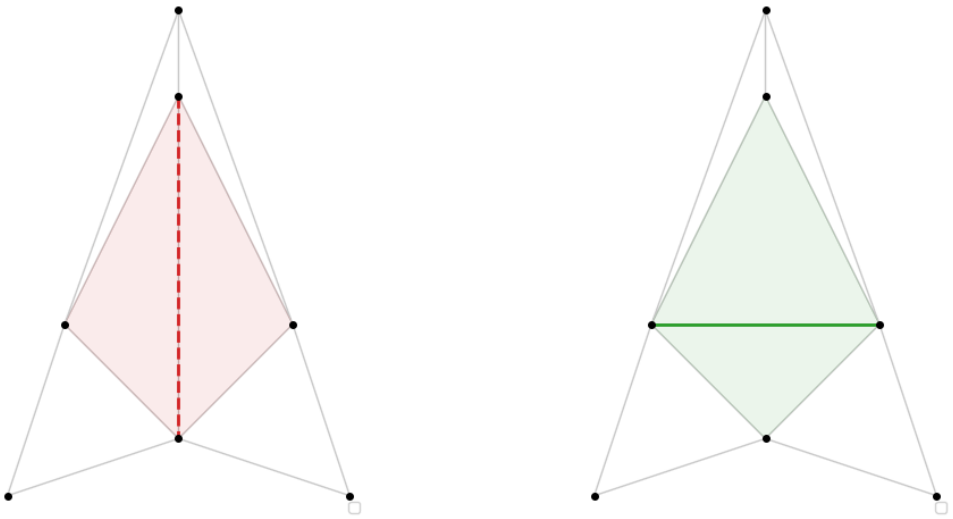
1. Wstęp teoretyczny

1.1. Zarys problemu

Rozważamy zbiór punktów na płaszczyźnie $\mathbb{R}^2$, każdemu z tych punktów przypiszemy wysokość za pomocą funkcji $f : A \subset \mathbb{R}^2 \to \mathbb{R}$. Skończony zbiór punktów A wraz z f nazwiemy *Terenem*. Wprowadzimy jeszcze dodatkowe założenie, każda prosta pionowa przecina teren w jednym punkcie. *Teren* może być na przykład prostym modelem topologii pewnego obszaru np. gór.

Już na tym etapie możemy sformować pytanie, Jak oszacować wysokość punktów, które nie zostały podane, a znajdują się na płaszczyźnie. Lub inaczej jak zrobić trójwymiarową siatkę naszego *terenu*.

Można by uznać, że dowolny algorytm triangulacji byłby wystarczający, jednak to co musi realizować nasz algorytm to odnalezienie triangulacji optymalnej kątowo, która maksymalizuje miarę najmniejszych kątów. Unikamy trójkątów „długich i ostrych".



Rysunek 1: Po lewej stronie dowolna triangulacja po prawiej triangulacje Delaunay'a

1.2. Triangulacja Delaunay'a

Mając zbiór n punktów P, możemy stworzyć diagram Voronoi zbioru P oznaczony jako $\mathrm{Vor}(P)$. Każdemu z punktów (centrów) $p \in P$ odpowiada zbiór punktów płaszczyzny, dla którego p jest najbliższym centrum, obszar ten nazywamy komórką Voronoi $V(p)$.

Tworzymy odcinek $\overline{pq}$, jeśli komórki $V(p)$ oraz $V(q)$ są sąsiadujące, otrzymany graf jest grafem Delaunay'a.

Graf ten jest podstawą do otrzymania triangulacji Delaunay'a, wystarczy dokonać triangulacji wielokątów znajdujących się w tym grafie.

Wprowadźmy teraz kilka pojęć potrzebnych do sformułowania twierdzenia pokazującego, że właśnie triangulacja Delaunay'a rozwiązuje problem postawiony w paragrafie 1.1.

Definicja 1.

Wektorem triangulacji T nazywamy wektor wszystkich powstałych kątów ułożonych w porządku rosnącym $A(\mathrm{T}) = (\alpha_1, \alpha_2, ..., \alpha_{3m})$ gdzie m to liczba trójkątów.

Definicja 2.

Krawędź nielegalna triangulacji to krawędź, po której przekręceniu wektor kątów triangulacji rośnie (w porządku leksykograficznym).

Twierdzenie 1.

Niech P będzie zbiorem punktów na płaszczyźnie. Triangulacja T zbioru P jest legalna (nie zawiera krawędzi nielegalnych) wtedy i tylko wtedy, gdy triangulacja T jest triangulacją Delaunay'a zbioru P.

1.3. Algorytm Triangulacji Delaunay'a

W naszej implementacji wykorzystaliśmy iteracyjny algorytm triangulacji Delaunay'a (algorytm Bowyera–Watsona) w dwóch wariantach. Pierwszy wariant stanowi implementację podstawową, natomiast drugi wykorzystuje informację o topologicznym sąsiedztwie trójkątów należących do triangulacji. Wydajność obu podejść oraz szczegółowy opis algorytmów przedstawiono w dalszej części dokumentu.

2. Algorytmy wyznaczania triangulacji Delaunay'a

2.1. Idea Bowyera i Watsona

Oba nasze algorytmy realizują idea Bowyera-Watsona do odnalezienia triangulacji Delaunay'a. Oto kroki, realizowane przez naszą implementację:

- Stworzenie „supertrójkąta" zawierającego wewnątrz wszystkie punkty chmury poddawanej triangulacji. Dostajemy dzięki temu początkową triangulacje Delaunay'a T_0.
- Dodanie punktu p_{i+1} do istniejącej triangulacji T_i w celu wyznaczenia triangulacji T_{i+1}.
- Odnalezienie wszystkich trójkątów, których koła opisane zawierają nowo dodany punkt. Zbiór znalezionych trójkątów zostanie usunięty z triangulacji.
- Powstała po usunięciu trójkątów pusta przestrzeń (w dokumentacji nazywana „wnęką") stanowi spójny wielokąt. Algorytm łączy nowo dodany punkt p_{i+1} ze wszystkimi wierzchołkami brzegu tej wnęki, tworząc wachlarz nowych, poprawnych trójkątów wypełniających powstały obszar.

Fundamentalną różnicą w dwóch wersjach algorytmów jest sposób wyszukiwania, do których okręgów opisanych należy dodany punkt.

2.2. Podstawowy algorytm

W tym podejściu algorytm identyfikuje trójkąty do usunięcia (tj. te, które łamią warunek Delaunay'a) poprzez przegląd wszystkich elementów aktualnej triangulacji. Dla każdego istniejącego trójkąta sprawdzany jest warunek geometryczny: czy nowo dodawany punkt znajduje się we wnętrzu koła opisanego na tym trójkącie.

2.3. Przeszukiwanie sąsiedztwa topologicznego

Algorytm wyszukuje trójkąt zawierający punkt poprzez nawigację z wykorzystaniem sąsiedztwa topologicznego. Wyszukiwanie rozpoczyna się od ostatnio znalezionego trójkąta i przechodzi przez krawędzie do kolejnych sąsiadów w kierunku nowego punktu, aż do momentu trafienia na właściwy element.

3. Porównanie algorytmów

3.1. Informacje wstępne

Środowisko testowe

- Interpreter: Python 3.14.2
- Kluczowe biblioteki: `numpy` (wersja 2.4.0), `matplotlib` (wersja 3.10.8)

Specyfikacja sprzętowa

- System operacyjny: Windows 11 Pro
- Procesor: AMD Ryzen 5 3600 3.6GHz
- Pamięć RAM: 16 GB DDR4 3200MHz

Metodologia pomiarów

Czas wykonania mierzony był za pomocą funkcji `time.perf_counter()`, która zapewnia najwyższą dostępną rozdzielczość zegara systemowego. Każdy wynik przedstawiony na wykresach stanowi wartość dla pojedynczego uruchomienia na losowym zestawie danych.

Generowanie danych

Zbiory punktów testowych generowane były losowo z rozkładem jednostajnym (funkcja `np.random.uniform`). Zakres współrzędnych skalowano wraz ze wzrostem liczby punktów.

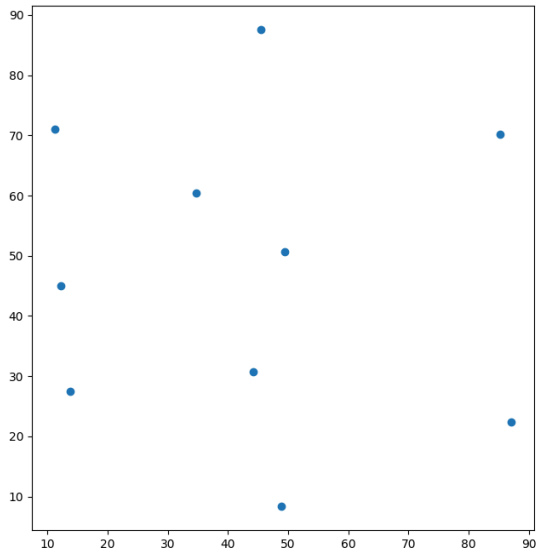
Przyjęte zaokrąglenie dla zera

Algorytmy geometryczne są wrażliwe na błędy numeryczne. W celu eliminacji problemów wynikających z arytmetyki zmiennoprzecinkowej, zastosowano tolerancję zera. Dwie wartości a i b uznawane są za równe, jeśli $|a - b| < \varepsilon$. Przyjęta wartość: $\varepsilon = 10^{-24}$.

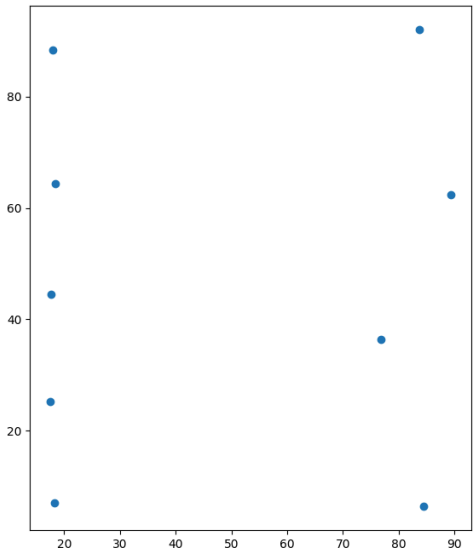
3.2. Testowe Zbiory punków

Do porównania wydajności oraz wyników algorytmów wykorzystaliśmy następujące zbiory punktów:

- 11 losowo generowane zbiory n punktów na płaszczyźnie $\mathbb{R}^2$
- Zbiór I i Zbiór II o małej liczbie punktów wprowadzone przy użyciu narzędzia graficznego, w celu wizualizacji deterministyczności algorytmów triangulacji Delauney'a.



Rysunek 2: Zbiór I

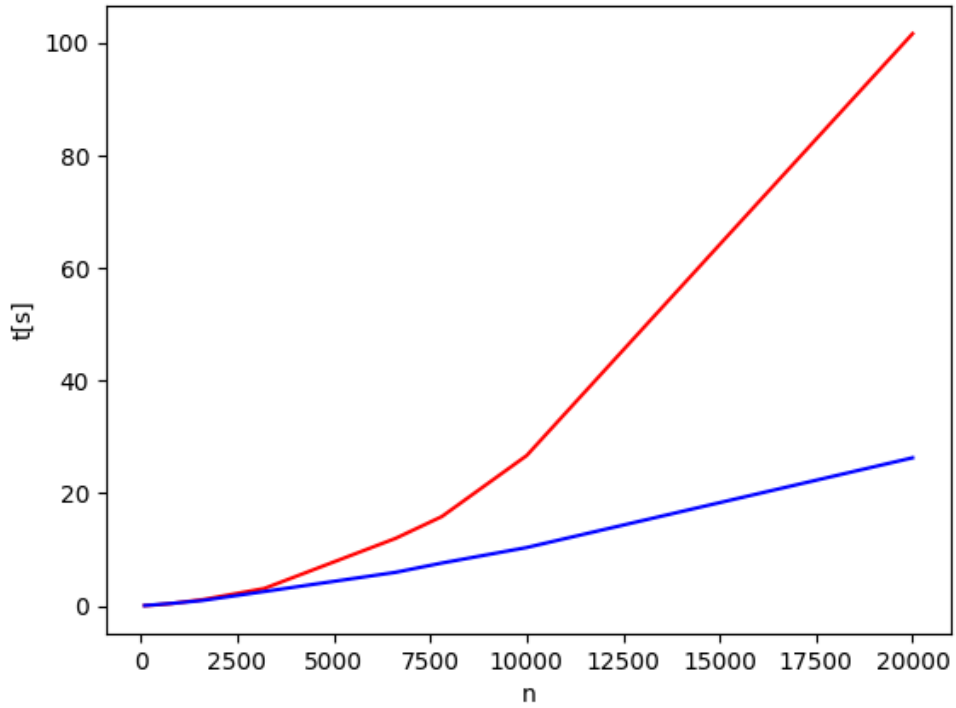


Rysunek 3: Zbiór II

3.3. Porównanie wydajności

Liczba punktów [n]	Czas działania algorytmu podstawowego [s]	Czas działania algorytmu wykorzystującego sąsiedztwo topologiczne [s]
100	0.03373	0.03784
200	0.07369	0.08394
300	0.11613	0.1324
400	0.15564	0.193
800	0.38443	0.40695
1600	1.07883	0.93624
3200	3.06416	2.52176
6600	11.96054	5.93036
7800	15.83734	7.58221
10000	26.6862	10.34222
20000	101.65853	26.28842

Tabela 1: Porównanie czasu działania algorytmów

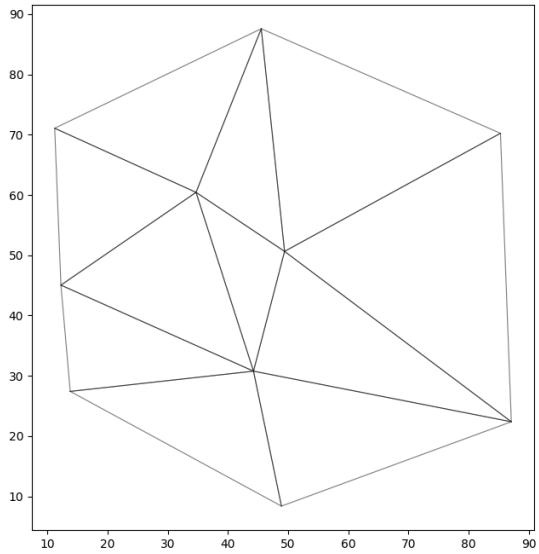


Rysunek 4: Porównanie wydajności dla dwóch metod wyszukiwania trójkąta. Niebieska linia: Podstawowy algorytm, czerwona: Przeszukiwanie sąsiedztwa topologicznego. Liczba losowo generowanych punktów znajduje się na osi n

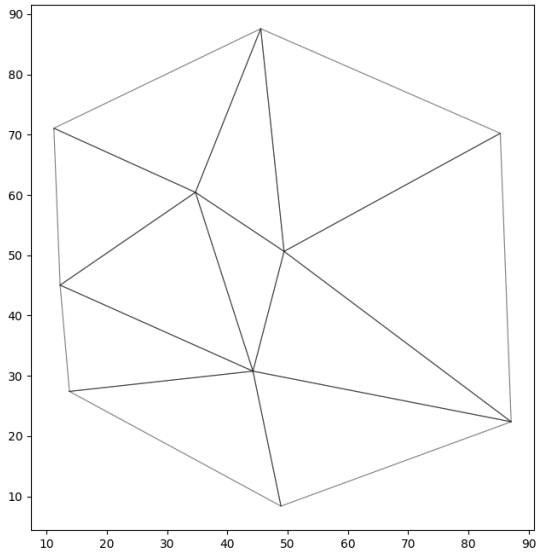
Dla niewielkich zbiorów danych (poniżej 1000 punktów) różnice w czasie wykonywania obu implementacji są minimalne, a algorytm podstawowy bywa paradoksalnie nieco szybsza. Wynika to z faktu, że dla małej liczby punków narzut związany z budowaniem i aktualizacją relacji sąsiedztwa w strukturze trójkątów przewyższa zyski z optymalizacji wyszukiwania. Dopiero wraz ze wzrostem liczby punktów, nieliniowa różnica w złożoności obliczeniowej – $O(N^2)$ dla metody podstawowej względem estymowanego asymptotycznego czasu $O\left(N\sqrt{N}\right)$ dla sąsiedztwa topologicznego – staje się wyraźnie widoczna, czyniąc tę drugą metodę znacznie wydajniejszą.

3.4. Wizualizacja wyników triangulacji dla zbiorów testowych

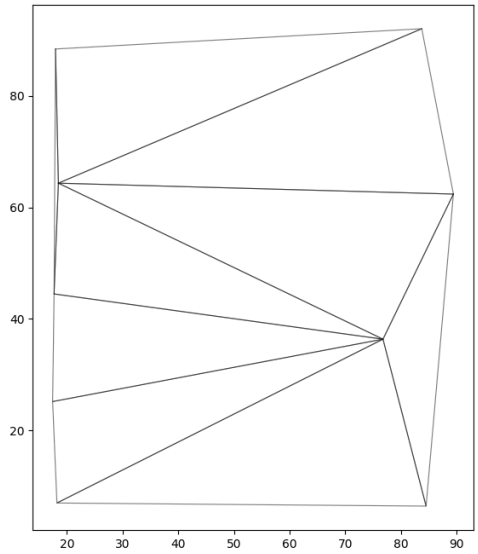
Porównanie wyników potwierdziło, że obie implementacje generują identyczną siatkę trójkątów. Ewentualne różnice mogłyby pojawić się tylko w przypadku niejednoznaczności triangulacji, co ma miejsce wyłącznie dla punktów współokręgowych.



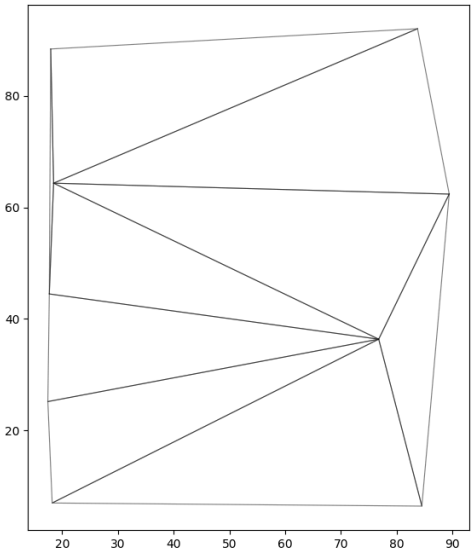
Rysunek 5: Zbiór I Algorytm podstawowy



Rysunek 7: Zbiór I Algorytm sąsiedztwa topologicznego



Rysunek 6: Zbiór II Algorytm podstawowy



Rysunek 8: Zbiór II Algorytm sąsiedztwa topologicznego

3.5. Wnioski

Po przeprowadzeniu testów implementacji oraz dokonaniu analizy otrzymanych wyników badań, możemy sformułować następujące wnioski, które obejmują:

1. **Wpływ topologii na wydajność obliczeniową**

   Wykorzystanie relacji sąsiedztwa między trójkątami znacząco usprawnia proces lokalizacji punktu dodawanego do triangulacji. W przeciwieństwie do metody podstawowej, która dla każdego wstawianego punktu przeszukuje cały zbiór trójkątów (złożoność liniowa $O(M)$ względem liczby trójkątów), metoda wykorzystująca sąsiedztwo topologiczne ogranicza przeszukiwanie lokalnie. Dzięki temu czas generowania triangulacji dla dużych zbiorów danych ($N > 1000$) jest o rząd wielkości krótszy.

2. **Poprawność i jednoznaczność wyników**

   Porównanie wygenerowanych triangulacji potwierdziło, że metoda lokalizacji punktu startowego nie wpływa na ostateczny kształt triangulacji. Zarówno metoda podstawowa, jak i algorytm oparty na sąsiedztwie topologicznym produkują identyczne zbiory trójkątów (zakładając brak przypadków zdegenerowanych). Potwierdza to, że algorytm Bowyera-Watsona jest deterministyczny niezależnie od sposobu znalezienia trójkąta zawierającego nowy punkt, pod warunkiem zachowania poprawności numerycznej.

3. **Skalowalność rozwiązań**

   Wraz ze wzrostem liczby punktów (N), różnica w czasie wykonania między badanymi metodami pogłębia się nieliniowo. **Metoda podstawowa** staje się nieefektywna dla bardzo dużych zbiorów danych ($N > 1000$) ze względu na kwadratową złożoność całego procesu triangulacji ($O(N^2)$), podczas gdy metoda oparta na **sąsiedztwie topologicznym** zachowuje znacznie lepszą skalowalność, dzięki złożoności obliczeniowej zbliżonej do $O\left(N\sqrt{N}\right)$.

4. Dokumentacja

4.1. Wymagania techniczne

Program został zaimplementowany w języku Python i wymaga do poprawnego działania następujących komponentów:

- **Python 3.10+** – zalecana wersja zapewniająca stabilność bibliotek graficznych.
- **NumPy (wersja 2.4.0+)** – wykorzystywany do generowania chmur punktów oraz poprawy wydajności obliczeń.
- **Matplotlib (wersja 3.10.8)** – służy jako silnik renderujący triangulację oraz podstawa interfejsu graficznego.
- **Pillow (PIL)** – niezbędny do generowania animacji w formacie GIF.
- **Tkinter** – biblioteka standardowa wykorzystana do ręcznego zadawania punktów przez użytkownika.

4.2. Opis programu

Program zaimplementowany w środowisku Jupyter Notebook (`user.ipynb`) oraz biblioteka `delaunay.py` stanowią zestaw narzędzi do generowania i wizualizacji triangulacji Delaunay'a chmury punktów w przestrzeni dwuwymiarowej R^2. Oprogramowanie zawiera dwa sposoby definiowania zbiorów punktów:

4.3. Interfejs graficzny `tniap`

Program oferuje interaktywne środowisko oparte na bibliotece `Tkinter`, które integruje wykresy `Matplotlib` bezpośrednio w oknie systemowym. Pozwala to na dynamiczne wprowadzanie danych i natychmiastowy podgląd wyników. Punkty zostają następnie zapisane do pliku `.JSON` w lokalizacji podanej przez użytkownika.

Budowa okna głównego

Główny panel składa się z obszaru rysowania (Canvas) oraz zestawu przycisków funkcyjnych. Integracja została zrealizowana za pomocą klasy `FigureCanvasTkAgg`.

Funkcjonalność i obsługa zdarzeń

Aplikacja opiera się na trzech kluczowych mechanizmach:

- **Przechwytywanie punktów (`on_click`)**: Każde kliknięcie lewym przyciskiem myszy w obszarze roboczym wykresu powoduje pobranie współrzędnych (x, y) i dodanie ich do globalnej listy `points`. Każdy punkt jest natychmiastowo wizualizowany jako niebieska kropka.

```
def on_click(event):
    if event.xdata is not None and event.ydata is not None:
        new_point = Point(event.xdata, event.ydata)
        points.append(new_point)
        # ... aktualizacja wykresu ...
```

- **Eksport danych (`save_to_json`)**: Funkcja otwiera systemowe okno dialogowe zapisu pliku. Cała lista zebranych punktów zostaje zserializowana oraz zapisana do pliku w formacie JSON, co pozwala na trwałe przechowywanie wygenerowanych układów. Zadany zbiór punktów zapisywany jest do nowego pliku o nazwie i lokalizacji podanej przez użytkownika.

```
def save_to_json():
    data = [{"x": p.x, "y": p.y} for p in points]
```

```
    with open(filename, 'w') as f:
        json.dump(data, f)
```

- **Zarządzanie stanem (`clear_canvas`)**: Przycisk „Wyczyść” pozwala na resetowanie listy punktów i odświeżenie widoku wykresu, co umożliwia szybkie rozpoczęcie nowej sesji projektowej.

```
def clear_canvas():
    points.clear()
    counter[0] = 0
    ax.clear()
    ax.set_xlim(0, 100)
    ax.set_ylim(0, 100)
    canvas.draw()
```

4.4. Import danych z formatu JSON

W celu umożliwienia zapisu danych oraz przeprowadzania testów na wcześniej zdefiniowanych zbiorach znajdujących się w pliku generowanym przez narzędzie `tniap`, zaimplementowana została funkcja `json_parser`. Pozwala ona na wczytanie współrzędnych punktów z pliku tekstowego o rozszerzeniu `.json` i ich automatyczną konwersję na listę obiektów klasy `Point`. Do funkcji należy przekazać ścieżkę do pliku `.json` w postaci tekstowego typu danych (`string`).

```
def json_parser(jsonFile):
    jsonFile = open(jsonFile, 'r')
    data = json.load(jsonFile)
    jsonFile.close()
    points = []
    for x,y in data:
         points.append(Point(x,y))
    return points
```

Przykładowy plik `.json` dla trzech punktów:

```
[[21.548387096774196, 80.03806538289297], [35.483870967741936, 24.339453649798465],
[79.87096774193549, 44.77160770264217]]
```

4.5. Funkcja generująca losowe punkty

Funkcja `generate_uniform` służy do automatycznego tworzenia testowych chmur punktów, co pozwala na weryfikację wydajności algorytmów dla dużych zbiorów danych. Generuje ona n instancji klasy `Point`, których współrzędne są losowane zgodnie z rozkładem jednostajnym w zadanym obszarze prostokątnym $[0, x] \times [0, y]$.

```
def generate_uniform(n,x,y):
    P = [Point(np.random.uniform(0,x),np.random.uniform(0,y)) for _ in range(n)]
    return P
```

Do generowania liczb losowych została wykorzystana funkcja `random.uniform()` biblioteki `numpy`.

4.6. Funkcje wizualizacji

Dzięki zastosowaniu parametru `vis=True` w funkcjach triangulacji, można wymusić na programie zapisywanie danych do zmiennej globalnej `HISTORY`.

- **Rejestracja stanów (`record_state`)**: Podczas kluczowych etapów, takich jak usuwanie „wnęki” czy retriangulacja, wywoływana jest funkcja tworząca zapis aktualnego stanu.

```
def record_state(triangulation, point=None, color='blue'):
    snap = {
        'tris': [Triangle(t.p1, t.p2, t.p3) for t in triangulation],
        'point': point,
        'color': color
    }
    HISTORY.append(snap)
```

- **Generowanie animacji**
  Po zakończeniu obliczeń, użytkownik może wywołać funkcję `render_gif`, która na podstawie zebranych danych w globalnej tablicy `HISTORY` generuje klatki animacji, wizualizując kroki algorytmu. Do funkcji należy przekazać chmurę punktów (tablice obiektów klasy `Point`) poddaną triangulacji, nazwę pliku do którego ma zostać zapisany `.gif` z wizualizacją. Zmiana parametru `duration` pozwala na modyfikację prędkości animacji.

```
def render_gif(original_points, filename, duration=300):
  #... reszta implementacji...#
```

- **Funkcja `plot`** Po wygenerowaniu triangulacji T (np. poprzez wywołanie `T = walkingSearch(P)`) możemy przekazać tę triangulacje do funkcji `plot()` wyświetlając tym samym wygenerowaną siatkę.

4.7. Opis struktur danych

W celu umożliwienia przebiegu triangulacji zaimplementowane zostały trzy struktury geometryczne: Punkt, Krawędź oraz Trójkąt. Ten zestaw klas pozwala na wprowadzenie sąsiedztwa topologicznego dla usprawnienia algorytmu Bowyera-Watsona.

- **Punkt (`Point`)**: Podstawowa jednostka geometryczna przechowująca współrzędne kartezjańskie $p_i = (x, y)$. Klasa ta implementuje metody do obliczania odległości euklidesowej oraz przeciąża operatory porównania (`__eq__`) i haszowania (`__hash__`). Porównywanie punktów uwzględnia tolerancję błędu numerycznego (stała `EPS`), a haszowanie stosuje zaokrąglanie współrzędnych, co zapobiega problemom wynikającym z niedokładności arytmetyki zmiennoprzecinkowej.

- **Krawędź (`Edge`)**: Struktura reprezentująca odcinek łączący dwa obiekty klasy `Point`. Kluczową cechą tej klasy jest traktowanie krawędzi (p_i, p_j) jako tożsama z krawędzią (p_j, p_i). Jest to osiągnięte poprzez odpowiednią konstrukcję metody haszującej (sortowanie punktów przed haszowaniem), co jest niezbędne do poprawnego wykrywania krawędzi granicznych podczas usuwania trójkątów z siatki.

- **Trójkąt (`Triangle`)**: Najbardziej złożona struktura, przechowująca trzy wierzchołki (p_i, p_j, p_k) oraz trzy odpowiadające im krawędzie. Konstruktor klasy automatycznie wymusza orientację wierzchołków przeciwną do ruchu wskazówek zegara (CCW – *Counter-Clockwise*) przy użyciu wyznacznika, co upraszcza późniejsze testy geometryczne.
  - **Topologia**: Klasa przechowuje listę `neighbours` – referencje do trzech topologicznie sąsiednich obiektów `Triangle`. Umożliwia to szybką nawigację po grafie triangulacji, redukując złożoność poszukiwania trójkąta, w którym znajduje się nowy punkt.
  - **Metody**: Zawiera test przynależności punktu do wnętrza (`is_inside`) oraz test okręgu opisanego (`is_in_circumcircle`), który jest warunkiem koniecznym triangulacji Delaunay'a.

Cała struktura triangulacji przechowywana jest wewnątrz kolekcji typu `set` (zbiór), co zapewnia unikalność trójkątów i umożliwia usuwanie oraz dodawanie elementów w czasie stałym $O(1)$.

4.8. Funkcje i metody

- **Klasa `Point`**
  Podstawowa struktura przechowująca współrzędne oraz obsługująca błędy precyzji.

```
def distance(self, other):
    return ((self.x - other.x)**2 + (self.y - other.y)**2)

def __eq__(self, other):
    return self.distance(other) <= EPS
```

  Metoda `__eq__` wykorzystuje stałą `EPS` (ustaloną na 10^{-24}), aby uniknąć problemów z porównywaniem wartości zmiennoprzecinkowych.

- **Klasa `Edge`**
  Służy do reprezentacji odcinka łączącego dwa obiekty klasy `Point`. W kontekście triangulacji Delaunay'a, krawędzie pełnią rolę łączników budujących strukturę trójkątów, a ich poprawna implementacja jest krytyczna dla procesu identyfikacji brzegu „wnęki".

  Zapewnienie braku skierowania krawędzi
  Podstawowym założeniem tej klasy jest to, że krawędź jest obiektem nieskierowanym. Oznacza to, że odcinek łączący punkt A z punktem B jest identyczny z odcinkiem łączącym B z A. Właściwość ta jest zaimplementowana poprzez odpowiednią konstrukcję operatora porównania `__eq__`.

```
def __eq__(self, other):
    if not isinstance(other, Edge):
        return False
    return (self.p1 == other.p1 and self.p2 == other.p2 or
            self.p1 == other.p2 and self.p2 == other.p1)
```

  Rola w wykrywaniu wnęki
  Podczas dodawania nowego punktu, algorytm znajduje wszystkie trójkąty naruszające warunek Delaunay'a i zbiera ich krawędzie. Krawędzie, które występują w tym zbiorze dokładnie raz, tworzą brzeg wnęki (wielokąta), który należy połączyć z nowym punktem. Dzięki nieskierowalności krawędzi, program może łatwo wykryć, które odcinki są współdzielone przez dwa usuwane trójkąty (wtedy występują dwa razy i są ignorowane), a które stanowią zewnętrzną granicę.

  Haszowanie niezależne od kolejności
  Aby krawędzie mogły być efektywnie zliczane (np. przy użyciu słowników lub zbiorów), klasa implementuje metodę `__hash__`. Wykorzystuje ona wartości hash punktów końcowych, sortując je (funkcje `min` i `max`) przed ostatecznym haszowaniem.

```
def __hash__(self):
    s = hash(self.p1)
    b = hash(self.p2)
    return hash((min(s, b), max(s, b)))
```

  Reprezentacja tekstowa
  Metoda `__repr__` pozwala wyświetlanie współrzędne obu punktów tworzących krawędź w konsoli.

```
def __repr__(self):
    return f"Edge({self.p1}, {self.p2})"
```

- **Klasa `Triangle`**
  Struktura danych w programie, reprezentująca pojedynczy trójkąt w siatce. Klasa ta łączy w sobie informacje o położeniu wierzchołków oraz relacji sąsiedztwa z innymi trójkątami. Implementuje kluczowe testy geometryczne niezbędne dla triangulacji Delaunay'a.

Konstruktor i wymuszanie orientacji
Głównym zadaniem konstruktora jest zapewnienie, że wierzchołki trójkąta są zawsze przechowywane w orientacji przeciwnej do ruchu wskazówek zegara. Jest to krytyczne dla poprawności predykatów geometrycznych używanych w dalszych częściach programu. Jeśli podane punkty tworzą orientację zgodną z ruchem wskazówek zegara, program zamienia dwa ostatnie punkty miejscami.

```
def __init__(self, p1, p2, p3):
    if orient(p1,p2,p3) < 0:
        self.p1 = p1
        self.p2 = p3
        self.p3 = p2
    else:
        self.p1 = p1
        self.p2 = p2
        self.p3 = p3

    self.edges = [Edge(self.p1, self.p2), Edge(self.p2, self.p3), Edge(self.p3, self.p1)]
    self.neighbours = [None, None, None]
    self.is_bad = False
```

Metoda `is_inside(p)`
Służy do weryfikacji, czy zadany punkt p znajduje się wewnątrz obszaru ograniczonego przez trójkąt. Dzięki stałej orientacji CCW, wystarczy sprawdzić, czy punkt p leży po lewej stronie każdej z trzech krawędzi skierowanych trójkąta

```
def is_inside(self, p):
    return (    orient(self.p1, self.p2, p) >= 0
            and orient(self.p2, self.p3, p) >= 0
            and orient(self.p3, self.p1, p) >= 0)
```

Metoda `is_in_circumcircle(p)`
Implementuje matematyczny warunek Delaunay'a. Sprawdza, czy punkt p leży wewnątrz okręgu opisanego na trójkącie. Test ten nie wymaga jawnego wyznaczania środka okręgu; zamiast tego obliczany jest wyznacznik macierzy 3x3 ze współrzędnymi przesuniętymi w lewo względem punktu p. Wynik większy od zera (z uwzględnieniem precyzji `EPS`) oznacza naruszenie warunku Delaunay'a.

```
def is_in_circumcircle(self, p):
    ax_, ay_ = self.p1.x - p.x, self.p1.y - p.y
    bx_, by_ = self.p2.x - p.x, self.p2.y - p.y
    cx_, cy_ = self.p3.x - p.x, self.p3.y - p.y

    det_a = ax_ * ax_ + ay_ * ay_
    det_b = bx_ * bx_ + by_ * by_
    det_c = cx_ * cx_ + cy_ * cy_

    det = (ax_ * (by_ * det_c - det_b * cy_) -
           ay_ * (bx_ * det_c - det_b * cx_) +
           det_a * (bx_ * cy_ - by_ * cx_))

    return det > EPS
```

Nadpisana metoda porównania i haszowania
W celu umożliwienia efektywnego porównywania oraz haszowania trójkątów, klasa Trójkąt (`Triangle`) nadpisuje metody `__eq__` oraz `__hash__`. Dwa trójkąty są uznawane za równe, jeśli posiadają ten sam zbiór wierzchołków, niezależnie od kolejności ich zdefiniowania w obiekcie.

```
def __eq__(self, other):
    if not isinstance(other, Triangle):
```

```
        return False
    vertices_self = {self.p1, self.p2, self.p3}
    vertices_other = {other.p1, other.p2, other.p3}
    return vertices_self == vertices_other

def __hash__(self):
    return hash(self.p1) ^ hash(self.p2) ^ hash(self.p3)
```

4.9. Algorytm Bowyera-Watsona

Algorytm Bowyera-Watsona to mechanizm, który buduje triangulację Delaunay'a poprzez sukcesywne wstawianie punktów do istniejącej struktury. Każde wstawienie musi gwarantować zachowanie warunku pustego koła opisanego.

Inicjalizacja: Funkcja `SuperTriangle(P)`

Proces rozpoczyna się od zdefiniowania supertrójkąta, który obejmuje wszystkie punkty zbioru wejściowego P. Dzięki temu każdy nowo dodawany punkt zawsze znajduje się wewnątrz istniejącej siatki, co eliminuje konieczność obsługi przypadków brzegowych na zewnątrz otoczki wypukłej podczas trwania algorytmu.

- **Logika wyznaczania supertrójkąta**: Funkcja wyznacza ekstremalne wartości współrzędnych chmury punktów, a następnie tworzy odpowiednio duży trójką zawierający wszystkie te punkty.

```
def SuperTriangle(P):
  maxX = -float('inf')
  minX = float('inf')
  maxY = -float('inf')
  minY = float('inf')
  for p in P:
    x = p.x
    y = p.y
    if x > maxX:
      maxX = x
    if x < minX:
      minX = x
    if y > maxY:
      maxY = y
    if y < minY:
      minY = y

  dx = maxX - minX
  dy = maxY - minY
  delta = max(dx, dy)
  if delta == 0: delta = 1

  p1 = Point((minX + maxX) / 2, maxY + 20 * delta)
  p2 = Point(minX - 20 * delta, minY - delta)
  p3 = Point(maxX + 20 * delta, minY - delta)

  return Triangle(p1, p2, p3)
```

Procedura wstawiania: `add_point_to_triangulation`

Główna logika Retriangulacji Delaunay'a opiera się na trzech krokach realizowanych dla każdego punktu p:

1. **Identyfikacja „wnęki"**: Przy użyciu przeszukiwania wgłąb (DFS), algorytm znajduje wszystkie trójkąty, których okręgi opisane zawierają punkt p. Zbiór ten tworzy „wnękę". Kluczową cechą tego obszaru jest bezpo-

średnia widoczność całego brzegu z punktu p. Oznacza to, że każda nowa krawędź poprowadzona z punktu p do wierzchołka „wnęki” w całości zawiera się w jej wnętrzu, co gwarantuje poprawność nowej siatki.

```
def dfs(t):
    visited_dfs.add(t)
    if t.is_in_circumcircle(p):
        removed.add(t)
        for n in t.neighbours:
            if n and n not in visited_dfs: dfs(n)
```

2. **Wyznaczenie brzegu wnęki**: Program analizuje krawędzie wszystkich usuniętych trójkątów. Krawędzie, które występują tylko raz w całym zbiorze usuniętych elementów, stanowią zewnętrzną granicę „wnęki”.
3. **Rekonstrukcja**: Punkt p zostaje połączony z każdą krawędzią brzegową, tworząc nowe trójkąty. Na koniec aktualizowane są relacje sąsiedztwa (`neighbours`), aby siatka zachowała spójność topologiczną.

Metody wyszukiwania

Wydajność algorytmu Bowyera-Watsona zależy w dużej mierze od tego, jak szybko algorytm jest w stanie znaleźć trójkąt, w którym znajduje się punkt p.

- **Metoda podstawowa (`naiveSearch`)**
  Jest to podstawowa technika lokalizacji punktu, polegająca na sekwencyjnym sprawdzaniu każdego istniejącego elementu siatki.

  Inicjalizacja i przygotowanie struktur:

  ```
  triangulation = set()
  st = SuperTriangle(points)
  triangulation.add(st)
  ```

  Proces rozpoczyna się od utworzenia „supertrójkąta” (`st`), który gwarantuje, że każdy punkt z chmury wejściowej znajdzie się wewnątrz początkowego obszaru roboczego. Triangulacja jest inicjalizowana jako zbiór (`set`), co pozwala na efektywne dodawanie i usuwanie elementów w kolejnych krokach.

  Mechanizm wyszukiwania liniowego
  Algorytm dla każdego punktu p wykonuje pełny przegląd zbioru trójkątów należących do aktualnej triangulacji. Lokalizacja odbywa się poprzez bezpośrednie badanie relacji geometrycznej między punktem a każdym trójkątem z osobna, bez uwzględniania ich wzajemnych powiązań.

  ```
  found = None
  for t in triangulation:
      if t.is_inside(p):
          found = t
          break
  ```

  Podczas iteracji wywoływana jest metoda `is_inside(p)`. Ponieważ wierzchołki trójkątów są zorientowane CCW, test ten sprawdza, czy punkt leży po lewej stronie (lub na linii) każdej z trzech skierowanych krawędzi trójkąta. Pierwszy napotkany element spełniający ten warunek przerywa pętlę i staje się podstawą do dalszej retriangulacji.

  Wydajność i złożoność
  W metodzie każdorazowe wstawienie punktu wymaga sprawdzenia średnio $O(n)$ trójkątów, gdzie n to liczba elementów aktualnie znajdujących się w siatce. Prowadzi to do całkowitej złożoności obliczeniowej $O(N^2)$ dla chmury N punktów.

  Przebieg retriangulacji:

```
if found:
    add_point_to_triangulation(triangulation, p, found, vis=vis)
```

Po znalezieniu trójkąta zawierającego punkt, program uruchamia funkcję `add_point_to_triangulation`. Warto zauważyć, że w tym wariancie nie ma potrzeby zapamiętywania ostatnio znalezionego elementu, ponieważ każda kolejna iteracja przeszukuje zbiór od początku.

Zakończenie i czyszczenie:

```
final_tris = clean_super_triangle(triangulation, st)
return final_tris
```

Finalnym etapem jest usunięcie wierzchołków super-trójkąta. Funkcja `clean_super_triangle` filtruje zbiór trójkątów, usuwając te, które współdzielą punkty z supertrójkątem, pozostawiając jedynie triangulację Delaunay'a właściwego obszaru danych.

- **Metoda przeszukiwania sąsiadów topologicznych (`walkingSearch`)**
  Jest to zoptymalizowana technika lokalizacji punktu, która zamiast przeszukiwania całego zbioru danych, wykorzystuje grafową strukturę triangulacji. Proces ten można przyrównać do nawigacji po siatce, gdzie algorytm przemieszcza się między przyległymi trójkątami w stronę celu.

  Inicjalizacja i przygotowanie struktur:

```
triangulation = set()
st = SuperTriangle(points)
triangulation.add(st)
last_found = st
```

Tworzona jest początkowa triangulacja poprzez wywołanie funkcji (`SuperTriangle`), który obejmuje wszystkie punkty. Zmienna `last_found` zostaje zainicjowana jako ten trójkąt – będzie ona punktem startowym dla pierwszego procesu wyszukiwania.

Mechanizm nawigacji topologicznej
Algorytm rozpoczyna pracę od trójkąta startowego. Punktem wyjścia dla każdego nowego punktu jest trójkąt znaleziony w poprzedniej iteracji (`last_found`).

Dla bieżącego trójkąta sprawdzana jest orientacja punktu p względem jego krawędzi. Wykorzystywany jest tu predykat `is_outside`: ponieważ trójkąty mają wymuszoną orientację CCW, jeśli punkt leży po prawej stronie krawędzi, oznacza to, że cel znajduje się w kierunku sąsiada przylegającego do tej krawędzi.

```
while curr:
  if vis:
      path.append(curr)
      record_state(triangulation, p, path[-1:], 'orange', f"Sąsiedztwo topologiczne: Punkt
{i+1} (Krok {len(path)})")

  if curr in visited:
      for t in triangulation:
          if t.is_inside(p): found = t; break
      break
  visited.add(curr)

  if curr.is_inside(p):
      found = curr
      break
```

Jeżeli punkt p znajduje się wewnątrz `curr` (test `is_inside`), nawigacja kończy się sukcesem. W przeciwnym razie iterujemy po krawędziach – znalezienie krawędzi, względem której punkt jest „na zewnątrz", pozwala na natychmiastowe przeskoczenie do odpowiedniego sąsiada dzięki liście `neighbours`.

Wydajność i złożoność
W przeciwieństwie do metody podstawowej ($O(n)$), średnia złożoność tej metody wynosi $O(\sqrt{n})$. Wynika to z faktu, że liczba odwiedzonych trójkątów jest proporcjonalna do liczby elementów przeciętych przez odcinek łączący punkt startowy z punktem p.

Zastosowanie spójności przestrzennej:

```
new_tris = add_point_to_triangulation(triangulation, p, found, vis=vis)
if new_tris:
    last_found = new_tris[0]
```

Po poprawnym wstawieniu punktu i retriangulacji, program aktualizuje `last_found`. Dzięki temu, jeśli punkty wejściowe leżą blisko siebie, kolejny proces wyszukiwania rozpocznie się niemal w tym samym miejscu, co redukuje złożoność do poziomu bliskiego $O(1)$.

Zapobieganie błędom
Algorytm został zaprojektowany tak, aby radzić sobie z błędami numerycznymi oraz specyficzną geometrią brzegów siatki. Implementacja zawiera dwa kluczowe zabezpieczenia :

1. **Detekcja cykli**: Program śledzi odwiedzone trójkąty w zbiorze `visited`. Jeśli algorytm wróci do tego samego miejsca (co może się zdarzyć przy błędach precyzji zmiennoprzecinkowej), przerywa przeszukiwanie sąsiadów topologicznych i dokonuje szukania punktu metodą podstawową.
2. **Obsługa minimów lokalnych i krawędzi**: Jeśli punkt znajduje się poza krawędzią brzegową (brak sąsiada) lub nie można wykonać dalszego ruchu (`not moved`), algorytm przewiduje bezpieczny powrót do metody podstawowej.

```
if curr in visited or not moved:
    for t in triangulation:
        if t.is_inside(p):
            found = t
            break
    break
visited.add(curr)
```

Zakończenie i czyszczenie:

```
final_tris = clean_super_triangle(triangulation, st)
return final_tris
```

Po przetworzeniu wszystkich punktów wywoływana jest funkcja `clean_super_triangle`, która usuwa wierzchołki supertrójkąta i wszystkie przyległe do nich krawędzie, pozostawiając finalną, czystą triangulację Delaunay'a obszaru właściwego.

4.10. Sposób pracy z przygotowanym programem

Struktura projektu

Folder `projekt/` zawiera dwa pliki. Pierwszy `delaunay.py` jest biblioteką zawierającą funkcje oraz klasy opisane w dokumentacji, natomiast user.ipynb to notatnik jupyter zawierający aplikacje okienkową `tniap` oraz demonstracje działania biblioteki `delaunay.py`.

projekt/
├── delaunay.py
└── user.ipynb

By korzystać z biblioteki we własnym programie, wystarczy umieścić plik delaunay.py w folderze roboczym i zaimportować funkcje poleceniem `from delaunay import *`. Zalecane jest jednak importowanie konkretnych klas i funkcji, np. `from delaunay import Triangle`.

5. Bibliografia

[1] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf, „Geometria obliczeniowa. Algorytmy i zastosowania", Wydawnictwa Naukowo-Techniczne.
[2] Wykład: „Wielościany Voronoi i triangulacja Delaunay'a" dr Barbary Głut.
[3] Dokumentacja biblioteki Matplotlib (użyta wersja 3.10.8).
[4] https://en.wikipedia.org/wiki/Delaunay_triangulation