Otoczka wypukła dla zbioru punktów w przestrzeni dwuwymiarowej

Temat 2, Grupa 2

Autorzy: Remigiusz Babiarz, Jakub Własiewicz

6 stycznia 2026

1. Wstęp

Ćwiczenie polegało na implementacji algorytmów wyznaczania otoczki wypukłej zaprezentowanych na wykładzie. Zaimplementowane zostały algorytmy:

- **Grahama**,
- **Jarvisa**,
- przyrostowy wyznaczania otoczki wypukłej (dalej **przyrostowy**)
- **górnej i dolnej otoczki**,
- **dziel i rządź**,
- **quickhull**,
- **Chana**.

Powyższe algorytmy następnie przetestowano na uprzednio przygotowanych danych testowych oraz porównano ich działanie. Przygotowano wizualizację działania każdego z algorytmów.

2. Szczegóły techniczne

2.1. Dane sprzętowe

- **System operacyjny**: Fedora Linux 43
- **Środowisko**: NeoVim/Visual Studio Code
- **Język**: Python
- **Procesor**: 12th Gen Intel Core i5-12450H $\times$ 12

2.2. Użyte biblioteki

W projekcie wykorzystano funkcjonalności zarówno z biblioteki standardowej języka, jak i bibliotek zewnętrznych. Poniżej znajduje się lista importowanych modułów wraz z opisem zastosowania:

- **itertools** - użyto do grupowania punktów względem współrzędnej,
- **time** - użyto do mierzenia czasu wykonywania algorytmów,
- **numpy** - użyto do ułatwienia zapisu danych, losowania punktów, tasowania zbiorów. Również wszystkie funkcje matematyczne zostały zaczerpnięte z tej biblioteki,
- **matplotlib** - umożliwia wizualizację działania algorytmów,
- **os** - użyto do obsługi systemu plików podczas zapisu i odczytu danych oraz animacji.

2.3. Struktura plików

Kod źródłowy projektu został podzielony na moduły.

Moduły realizujące rozwiązanie ćwiczenia znajdują się w katalogu **/src/**. Poniżej znajduje się lista modułów wraz z opisem każdego z nich:

- **tests.py** - zawiera generatory punktów losowych, oraz funkcję umożliwiającą przeprowadzenie analizy czasu działania zbioru algorytmów na zbiorach punktów generowanych przez zadany generator. Opisy poszczególnych generatorów znajdują się w sekcji,
- **drawing.py** - pozwala na wprowadzanie zbioru punktów przez użytkownika, zawiera funkcje umożliwiające wizualizację zbioru punktów oraz otoczki tego zbioru. Ponadto zawiera klasę *Visualization* pozwalającą na wizualizację kroków algorytmu. Każdy z algorytmów posiada wersję wykorzystującą tą klasę do prezentacji graficznej poszczególnych kroków.
- **pozostałe pliki** - realizacje poszczególnych algorytmów wyznaczania otoczki wypukłej. W sekcji 3.3 przy opisie każdego z algorytmów znajduje się informacja, który plik zawiera jego kod źródłowy.

Ponadto plik **main.py**, znajdujący się poza katalogiem **/src/**, został przygotowany w celu realizacji warstwy użytkownika opisanej w następnej sekcji dokumentacji.

2.4. Warstwa użytkownika

Program należy uruchomić używając pliku **main.py**. Plik wykorzystuje funkcjonaności kodu źródłowego do prezentacji działania algorytmów wyznaczania otoczki wypukłej. Całość interfejsu użytkownika realizowana jest przez konsolę. Poprzez wybór różnych opcji użytkownik może:

- przeprowadzić analizę czasu działania algorytmów dla różnych zbiorów danych,
- wprowadzić własny zbiór punktów,
- wygenerować zbiór punktów z użyciem generatora z pliku **tests.py**,
- wczytać zbiór punktów z pliku,
- zapisać zbiór punktów do pliku,
- zapisać wynik działania wybranego algorytmu - otoczkę wypukłą zbioru punktów,
- wyświetlić wizualizację działania wybranego algorytmu w oknie *matplotlib*,
- zapisać wizualizację działania wybranego algorytmu do pliku *.gif*.

Jeżeli program jest uruchamiany z poziomu **main.py**:

- zbiory punktów zapisywane są w katalogu **/data/** (są również z niego wczytywane),
- otoczki wypukłe zapisywane są w katalogu **/hulls/**,
- wizualizacje działania algorytmów zapisywane są w katalogu **/gifs/**.

3. Realizacja ćwiczenia

3.1. Dane wejściowe

Przyjmujemy, że zbiorem danych wejściowych jest zbiór punktów na płaszczyźnie. Punkty są krotkami zawierającymi dwie liczby zmiennoprzecinkowe reprezentujące ich współrzędne. Przyjmujemy, że w zbiorze punktów nie ma żadnych duplikatów (punktów o tych samych współrzędnych). Mogą natomiast wystąpić pary punktów o tej samej jednej ze współrzędnych.

3.2. Oczekiwany wynik działania algorytmów

Każdy z zaimplementowanych algorytmów ma w założeniu wyznaczyć otoczkę wypukłą punktów z danych wejściowych. Za poprawną otoczkę przyjmuje się listę punktów taką, że:

- punkty z tej listy są wierzchołkami wielokąta następującymi po sobie w kolejności przeciwnej do ruchu wskazówek zegara,
- wielokąt ten jest otoczką wypukłą zbioru danych wejściowych.

3.3. Omówienie implementacji algorytmów

3.3.1. Elementy wspólne

3.3.1.1. Wyznacznik - *det*

Funkcja *det(a,b,c)* obliczająca wyznacznik macierzy:

$$\det(a,b,c) = \begin{pmatrix} x_a & y_a & 1 \\ x_b & y_b & 1 \\ x_c & y_c & 1 \end{pmatrix}$$

została wielokrotnie wykorzystana w zaimplementowanych algorytmach. Poprzez badanie znaku tego wyznacznika można określić orientację trzech następujących po sobie punktów.

Z racji na niedokładność obliczeń, za każdym razem, gdy badano znak wartości funkcji *det*, użyto wartości $\varepsilon = 10^{-12}$ jako tolerancji dla 0.

3.3.1.2. Sortowanie z usuwaniem punktów współliniowych - *x_sort*

Funkcja *x_sort* została zaimplementowana na potrzebę sortowania punktów względem ich współrzędnej ***x***, która to funkcjonalność znajduje zastosowanie w algorytmach: ***przyrostowym***, ***górnej i dolnej otoczki*** oraz ***dziel i rządź***.

Funkcja poza sortowaniem punktów dodatkowo grupuje punkty o tej samej współrzędnej ***x***. Jeżeli w danej grupie występują co najmniej 3 punkty funkcja dodatkowo usuwa ze zbioru wynikowego wszystkie punkty poza tymi o najmniejszej i największej współrzędnej ***y***. Następnie łącząc grupy i zwracaja posortowaną listę.

Złożoność takiego sortowania wynosi $O(n log(n))$, gdzie n to liczba punktów.

3.3.2. Algorytm Grahama

Plik **graham.py**.

3.3.2.1. Przebieg algorytmu

Algorytm rozpoczyna działanie od przygotowania danych. Najpierw znajduje w zbiorze wejściowym punkt ***lowest_point*** o najniższej drugiej współrzędnej, lub - w przypadku remisu - punkt o najniższych współrzędnych. Następne sortuje pozostałe punkty na podstawie kątu jaki tworzy odcinek tworzony przez dany punkt oraz punkt ***lowest_point*** z osią ***OX***. Kąt ten jest obliczany z pomocą funkcji *atan2* z biblioteki *numpy*. W przypadku remisu punkty porządkowane są w kierunku rosnącej odległości od puntu ***lowest_point***. Ostatnim krokiem przygotowującym dane jest usunięcie z posortowanej listy punktów współliniowych - w przypadku trójki (***lowest_point***, ***p***, ***q***), gdzie punkt ***p*** leży na odcinku (***lowest_point***, ***q***), punkt ***p*** jest usuwany ze zbioru danych.

Następnie algorytm iteracyjnie wyznacza otoczkę wypukłą przygotowanego zbioru. Tworzy stos ***hull***, na którym umieszcza ***lowest_point***. Następnie, dla każdego kolejnego punktu ***p*** z posortowanej listy wykonuje:

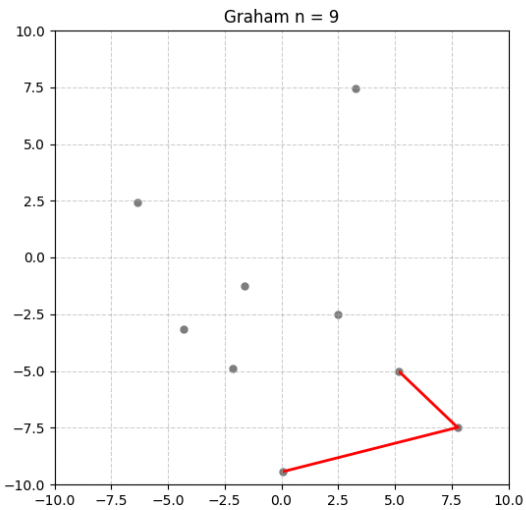
- dopóki na stosie są co najmniej 2 punkty sprawdź relację dwóch ostatnich punktów na stosie z punktem ***p*** z użyciem funkcji *det*:

– jeśli *det(**przedostatni punkt**, **ostatni punkt**, **p**) <= 0* (skręt w prawo), usuń ostatni punkt ze stosu,
– w przeciwnym wypadku, dodaj punkt ***p*** na górę stosu.

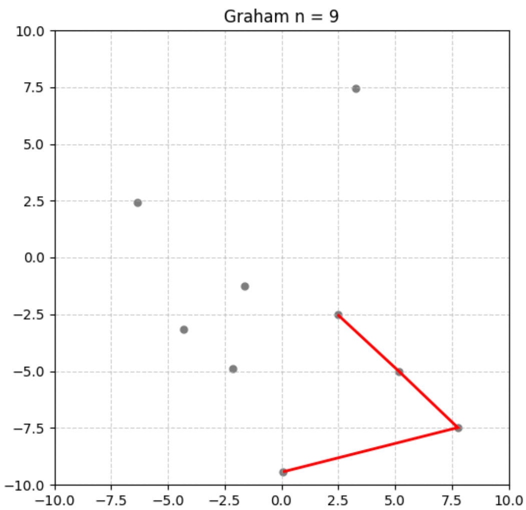
Po przetworzeniu wszystkich punków na stosie pozostaje lista wynikowa będąca otoczką wypukłą zbioru punktów z danych wejściowych.

3.3.2.2. Prezentacja działania

Dopóki warunek wypukłości nie jest naruszony algorytm dodaje punkty jeden po drugim (rys. 1, rys. 2).

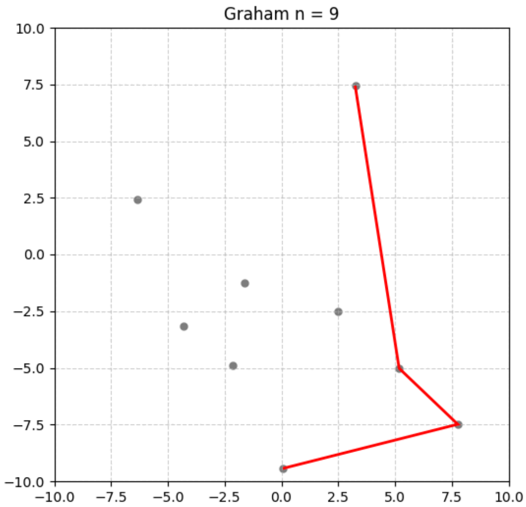


Rysunek 1: dodanie punktu

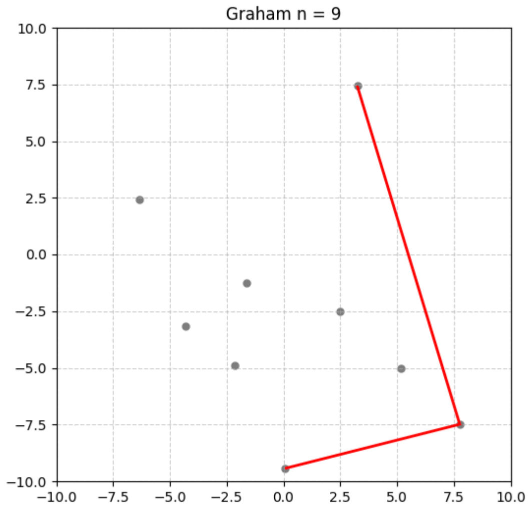


Rysunek 2: dodanie punktu

Kiedy kąt wewnętrzny okazuje się być rozwarty po dodaniu punktu algorytm usuwa dodane punkty aż warunek wypukłości będzie spełniony (rys. 3, rys. 4).



Rysunek 3: naruszenie warunku wypukłości



Rysunek 4: usunięcie punktu wewnętrznego

W ten sposób algorytm buduje całą otoczkę.

3.3.2.3. Analiza złożoności obliczeniowej

Cały algorytm ma złożoność obliczeniową *O(nlog(n))*, gdzie n to liczba punktów na płaszczyźnie. Najbardziej kosztownym etapem algorytmu jest sortowanie punktów - wykonuje się ono w czasie *O(nlog(n))*. Następny etap algorytmu jest iteracją po punktach i wymaga czasu $O(n)$. Algorytm Grahama jest bardzo uniwersalnym algorytmem o przewidywalnym czasie działania dla każdego zbioru danych.

3.3.3. Algorytm Jarvisa

Plik **jarvis.py**.

3.3.3.1. Przebieg algorytmu

Algorytm jest inaczej nazywany algorytmem *owijania prezentu* (ang. *gift wrapping*). Podobnie jak algorytm Grahama (3.3.2) rozpoczyna działanie od znalezienia punktu ***lowest_point*** o najmniejszej pierwszej współrzędnej, lub - w przypadku remisu - o najmniejszych obu współrzędnych. Następnie algorytm znajduje następny punkt należący do otoczki z pomocą funkcji *det* - iteruje po wszystkich punktach znajdując taki punkt ***best***, dla którego wszystkie inne punkty leżą po lewej stronie odcinka (***last***, ***best***), gdzie ***last*** jest ostatnim znalezionym punktem należącym do otoczki - początkowo ***lowest_point***. Po przetworzniu wszystkich puntów punkt ***best*** staje się punktem ***last*** i jest dodawany do otoczki. Algorytm kroki te powtarza, aż znaleziony zostanie punkt startowy, co reprezentuje zamknięcie otoczki.

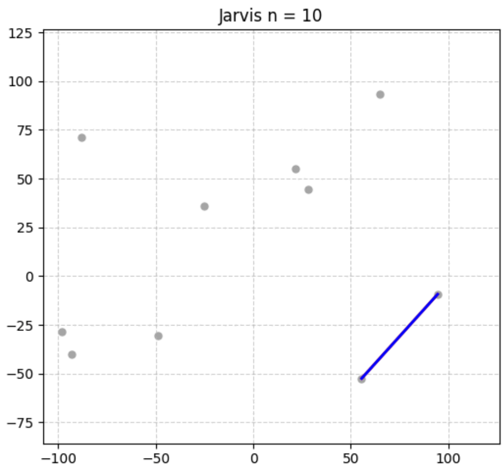
3.3.3.2. Analiza złożoności obliczeniowej

Algorytm Jarvisa ma złożoność *O(nk)*, gdzie *n* to liczba punktów na płaszczyźnie, oraz *k* to liczba punktów należących do otoczki. Wynika ona z prostego faktu znajdowania jednego punktu należącego do otoczki w każdym kroku algorytmu, która objemuje iteracje po wszystkich punktach ze zbioru wejściowego. Faktyczny czas działania algorytmu może być nieprzewidywalny i jest bardzo wrażliwy na różne dane wejściowe - w oczywisty algorytm

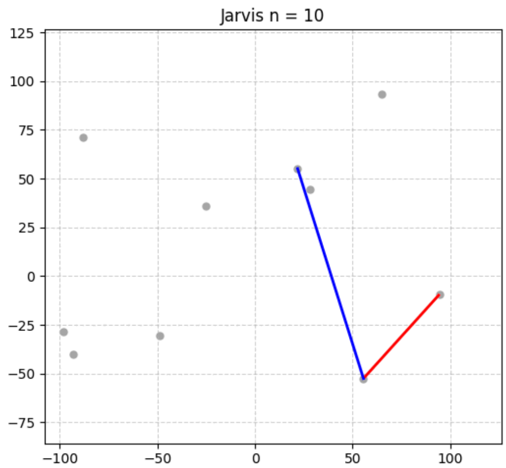
Jarvisa nie jest najlepszym wyborem do wyznaczania otoczek zbiorów punktów o potencjalnie wielu punktach należących do otoczki.

3.3.3.3. Prezentacja działania

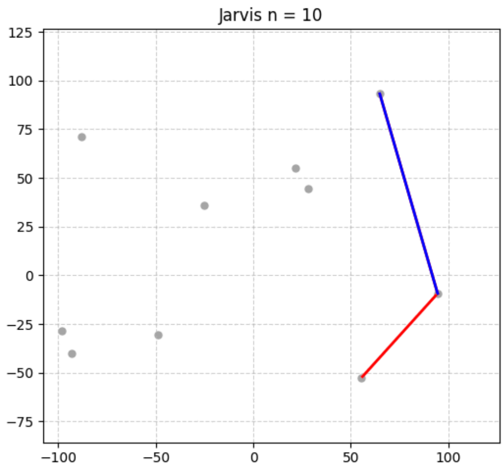
Na rysunkach 5-8 zaprezentowano działanie algorytmu.



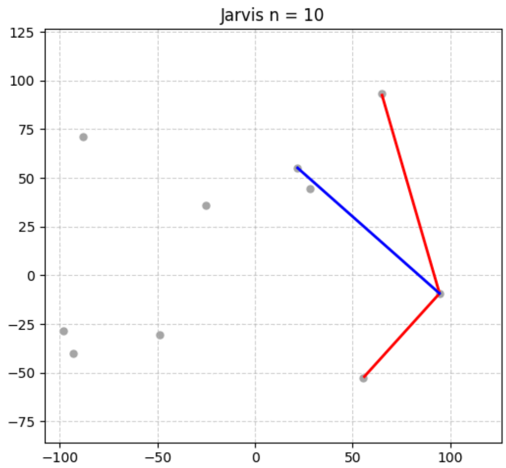
Rysunek 5: sprawdzanie punktów jeden po drugim



Rysunek 6: algorym pamięta najodleglejszy w sensie biegunowym punkt



Rysunek 7: wszystkie punkty zostały przetworzone



Rysunek 8: powtarzanie kroków dla nowego punktu otoczki

W ten sposób algorytm buduje całą otoczkę.

3.3.4. Algorytm przyrostowy

Plik **incremental.py**.

3.3.4.1. Przebieg algorytmu

Algorytm najpierw sortuje punkty z użyciem funkcji *x_sort*.

Algorytm tworzy pierwszą otoczkę na podstawie dwóch pierwszych punktów posortowanego zbioru. Następnie iteracyjnie dodaje każdy z pozostałych punktów do otoczki. Dołączanie punktu opiera się na znalezieniu stycznych do otoczki przechodzących przez ten punkt i usunięcie wszystkich punktów otoczki, które znajdowałyby się wewnątrz otoczki po dodaniu do niej rozważanego punktu.

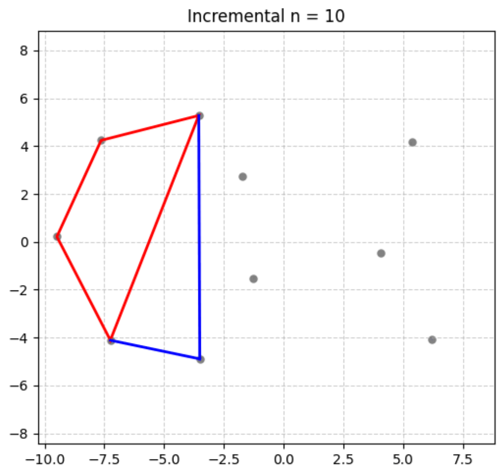
Styczne znajdowane są z pomocą funkcji *det*.

3.3.4.2. Analiza złożoności algorytmu

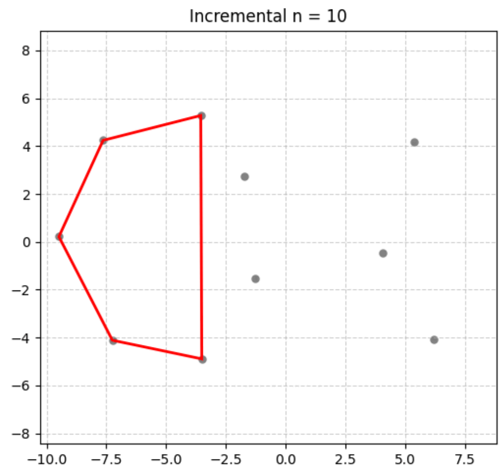
Algorytm ma złożoność *O(nlog(n))*, gdzie *n* to liczba punktów na płaszczyźnie. Samo sortowanie punktów zajmuje *O(nlog(n))* czasu procesora. Podczas iteracyjnego dołączania punktów do otoczki każdy punkt jest dodawany do otoczki raz i maksymalnie raz z niej usuwany, złożoność tego kroku wynosi więc *O(n)*, a finalna złożoność algorytmu przyrostowego to faktycznie *O(nlog(n))*.

3.3.4.3. Prezentacja działania=

Tak długo jak nie występują punkty wewnętrzne algorytm dodaje każdy punkt w kolejności sortowania (rys. 9, rys. 10).

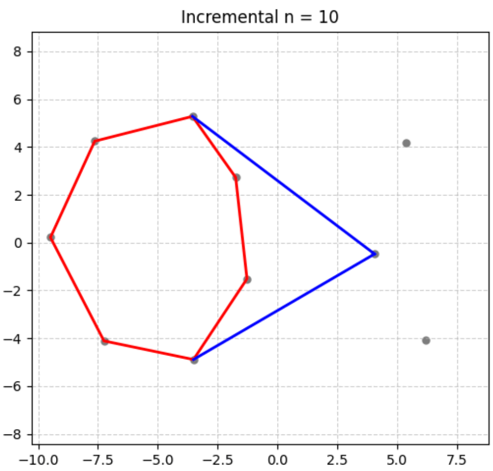


Rysunek 9: dodawanie punktu

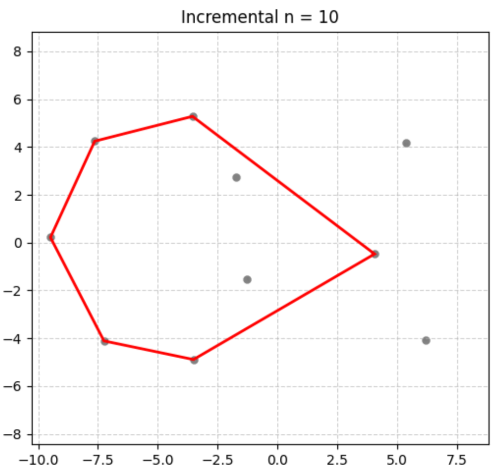


Rysunek 10: brak punktów wewnętrznych

W momencie wystąpienia punktów wewnętrznych algorytm usuwa wszystkie takie punkty (rys. 11, rys. 12).



Rysunek 11: dodawanie punktu



Rysunek 12: punkty wewnętrzne usunięte z otoczki

W ten sposób algorytm buduje całą otoczkę.

3.3.5. Algorytm górnej i dolnej otoczki

Plik **monochain.py**.

3.3.5.1. Przebieg algorytmu

Algorytm rozpoczyna pracę od posortowania zbioru punktów z użyciem funkcji *x_sort*.

Następnie algorytm iteracyjnie konstuuje górną otoczkę punktów. Początkowa górna otoczka składa się z dwóch pierwszych punktów w posortowanym zbiorze. Każdy kolejny punkt jest dodawany do górnej otoczki, po uprzednim usunięciu z niej wszystkich punktów, których obecność naruszyła by warunek wypukłości, który sprawdzany jest z użyciem funkcji *det*. Dolna otoczka wyznaczana jest analogicznie.

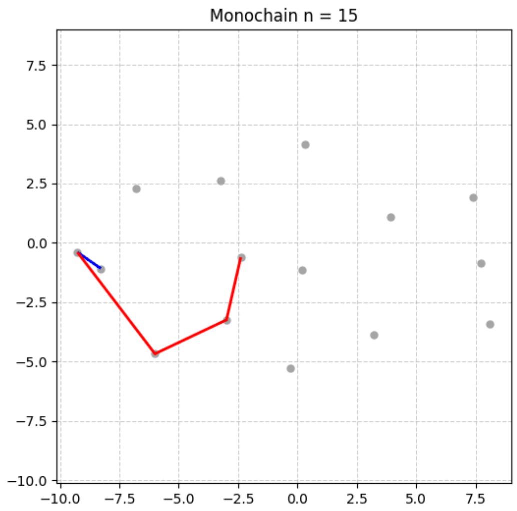
Ostatnim krokiem algorytmu jest połączenie górnej i dolnej otoczki z uwagą na warunek prawoskrętności otoczki.

3.3.5.2. Analiza złożoności obliczeniowej

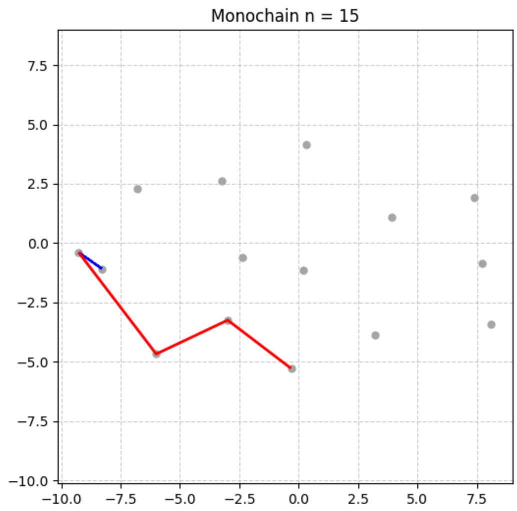
Sortowanie punktów wykonywane jest w czasie *O(nlog(n))*, gdzie *n* to liczba punktów na płaszczyźnie. Każdy punkt jest przetwarzany w iteracyjnej części algorytmu stałą liczbę razy. Finalna złożoność algorytmu górnej i dolnej otoczki to więc *O(nlog(n))*.

3.3.5.3. Prezentacja działania algorytmu

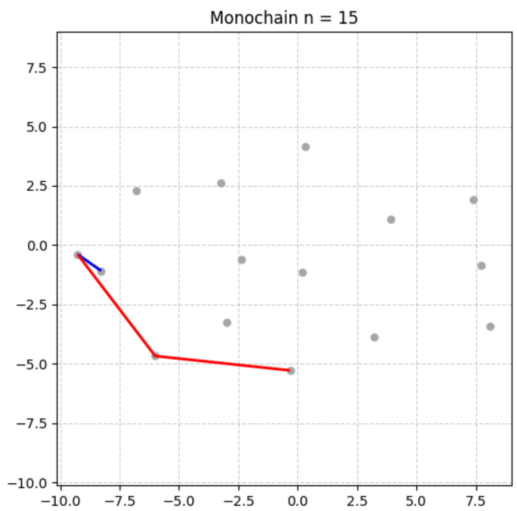
Na rysunkach 13, 14, 15 zaprezentowano kroki budowy dolnej otoczki. Górna otoczka jest budowana analogicznie i razem z dolną tworzy otoczkę wypukłą zbioru (rys. 16).



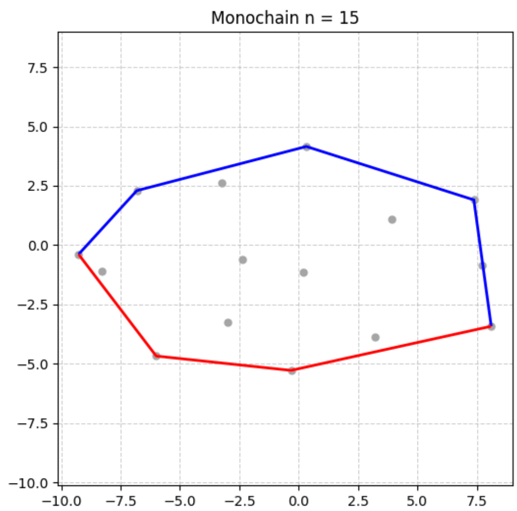
Rysunek 13: budowanie dolnej otoczki



Rysunek 14: naruszenie warunku wypukłości



Rysunek 15: usunięcie punktów wewnętrznych



Rysunek 16: otoczka wypukła będąca sumą otoczki górnej i dolnej

3.3.6. Algorytm dziel i rządź

Plik **divide_and_conquer.py**.

3.3.6.1. Przebieg algorytmu

Algorytm opiera się na utworzeniu zbioru otoczek, które w sumie obejmują cały zbiór punktów wejściowych, oraz na późniejszym łączeniu ich w czasie stałym do momentu otrzymania jednej otoczki obejmującej cały zbiór.

Pierwszym krokiem jest posortowanie punktów z użyciem funkcji *x_sort*. Następnie tak posortowana lista jest dzielona na części. Każda z tych części jest listą kolejnych $\boldsymbol{k}$ punktów należących do posortowanej listy wejściowej, gdzie ***k*** jest małą stałą będącą parametrem algorytmu. Następnie dla każdego z tych podzbiorów punktów wyznaczana jest otoczka wypukła z użyciem algorytmu Grahama.

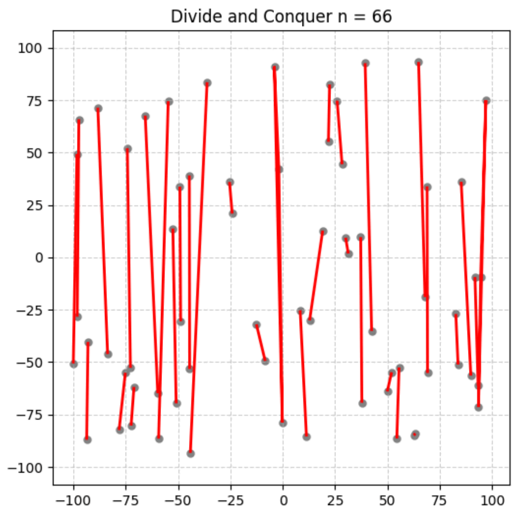
Tak powstałe sąsiednie otoczki są łączone poprzez znajdowanie stycznych z użyciem funkcji *det*. Łączenie to jest powtarzane do momentu otrzymania jednej otoczki będącej sumą wszystkich otoczek.

3.3.6.2. Analiza złożoności obliczeniowej

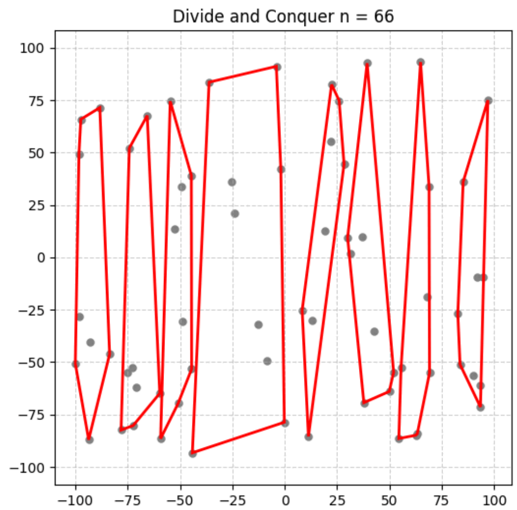
Samo sortowanie zbioru wejściowego wykonuje się w czasie *O(nlog(n))*. Wyznaczanie otoczek podzbiorów dla małej stałej ***k*** zajmuje stały czas O(k). Ponieważ ten krok powtarzany jest dla *n/k* otoczek zajmuje on w sumie $O\left(k \times \frac{n}{k}\right) = O(n)$ czasu procesora. Łączenie 2 otoczek zajmuje stały czas, a samych otoczek do połączenia jest $\frac{n}{k}$. Czas poświęcany na ten krok wynosi więc $O\left(\frac{n}{k} log\left(\frac{n}{k}\right)\right) = O(nlog(n))$. Finalna złożoność algorytmu wynosi więc *O(nlogn)*.

3.3.6.3. Prezentacja działania algorytmu

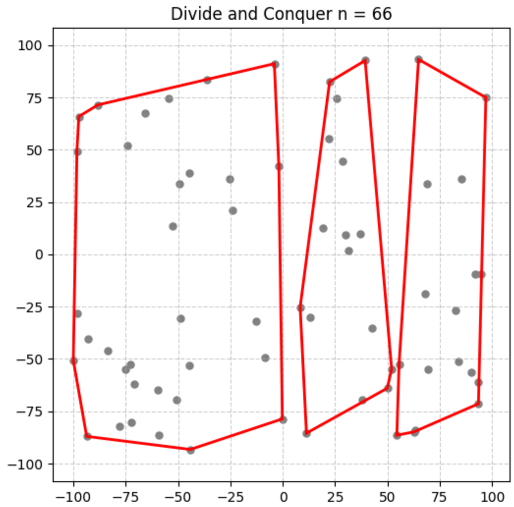
Na rysunkach 17-20 zaprezentowano wybrane kroki algorytmu, na których widać, jak otoczki łączą się. Połączenie wszystkich otoczek jest otoczką wypukłą zbioru.



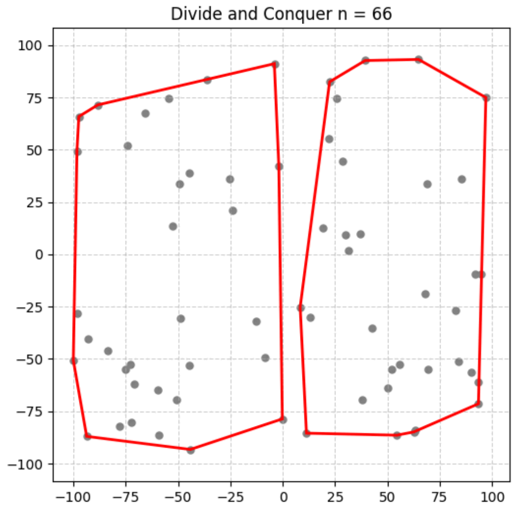
Rysunek 17: początkowy stan dla k=2



Rysunek 18: połączone otoczki



Rysunek 19: trzeci od końca krok algorytmu



Rysunek 20: przedostatni krok algorytmu

3.3.7. Algorytm Quickhull

Plik **quickhull.py**

3.3.7.1. Przebieg algorytmu

Algorytm rozpoczyna pracę od znalezienia 4 punktów o skrajnych współrzędnych:
- maksymalnej współrzędnej ***x***,
- minimalnej współrzędnej ***x***,
- maksymalnej współrzędnej ***y***,
- minimalnej współrzędnej ***y***.

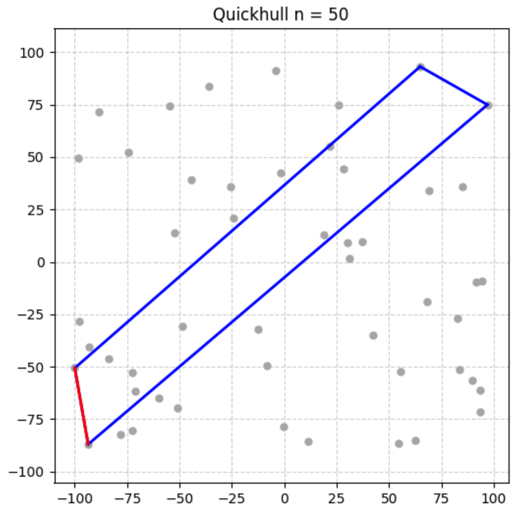
Punkty te definiują 4 odcinki, które tworzą wielokąt. Wielokąt ten określamy otoczką, którą będziemy rekurencyjnie rozszerzać. Wszystkie punkty wewnątrz otoczki są usuwane. Pozostałe punkty przetwarzane są rekurencyjnie z użyciem metody *dziel i rządź* poprzez wywoływanie funkcji ***rec_hull***, która działa następująco:
Dla danego odcinka znajdowany jest punkt znajdujący się najdalej od niego i będący na zewnątrz wielokąta tworzonego przez aktualne punkty otoczki. Punkt ten wraz z końcami rozpatrywanego odcinka definiuje 2 kolejne odcinki, które dodawane są do otoczki. Na tych nowych odcinkach rekurencyjnie wywoływana jest funckja ***rec_hull*** tak długo jak istnieją punkty poza otoczką.

3.3.7.2. Analiza złożoności obliczeniowej

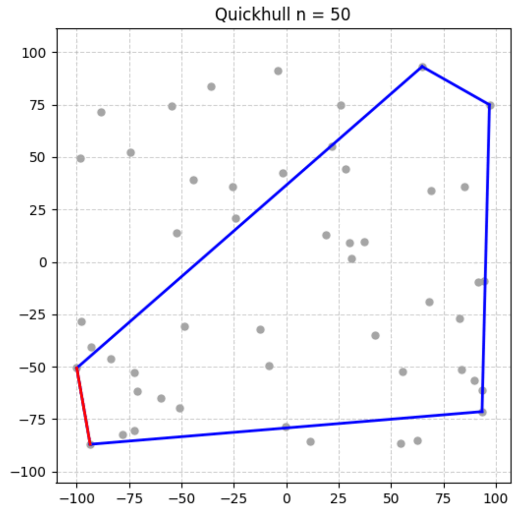
Każde rekurencyjne wywołanie fukcji ***rec_hull*** iteruje się po zbiorze punktów, którego rozmiar jest proporcjonalny do rozmiaru zbioru wejściowego. Ilość takich wywołań w pełni zależy od charakterystyki zbioru wejściowego. W pesymistycznym przypadku w każdym wywołaniu ***rec_hull*** jedyny usuwany punkt jest tym najbardziej odległym od rozpatrywanego odcinka, wtedy liczba wywołań wynosi *n*, a pesymistyczna złożoność obliczeniowa wynosi *O(n^2)*. Przypadek ten zachodzi gdy wszystkie punkty, lub ich większość należy do otoczki. Realistycznie jednak, zakładając względnie równomierne rozłożenie punktów, przy każdym „powiększaniu” otoczki przez funkcję ***rec_hull*** punkty należące do obszaru proporcjonalnego do długości odcinka są usuwane. Zamortyzowana złożoność obliczeniowa wynosi więc *O(nlog(n))*.

3.3.7.3. Prezentacja działania algorytmu

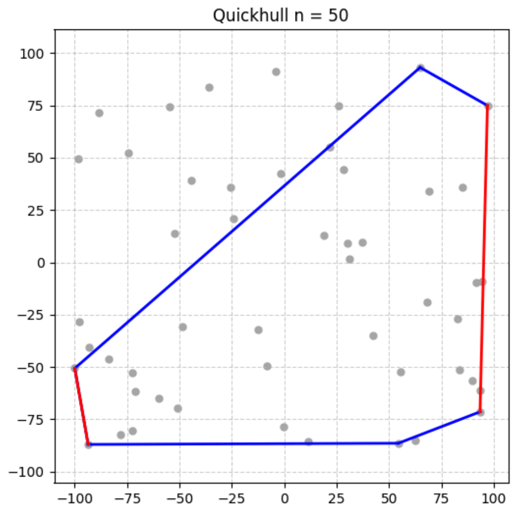
Na rysunkach 21-24 zaprezentowano wybranekroki algorytmu quickhull. Na czerwono zostały oznaczone odcinki, które na danym etapie algorytmu zostały przetworzone i na pewno należą do otoczki.



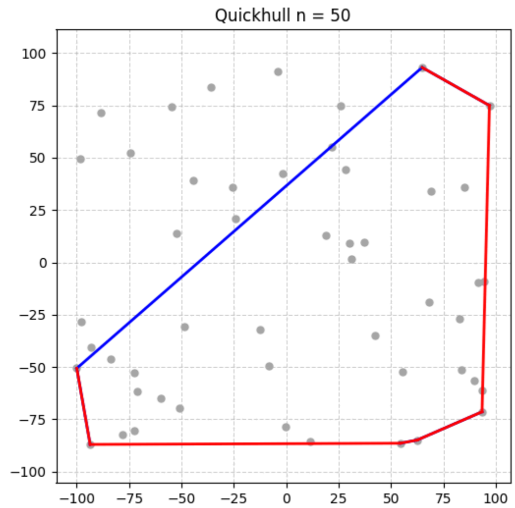
Rysunek 21: rozpoczęcie rekurencji



Rysunek 22: rozbicie odcinka



Rysunek 23: jedna z gałęzi zakończyła rekurencję



Rysunek 24: stan otoczki na moment przetworzenia 3 z 4 odcinków startowych

Przetworzenie ostatniego odcinka (rys. 24 - niebieski odcinek) skutkuje wyznaczeniem otoczki wypukłej.

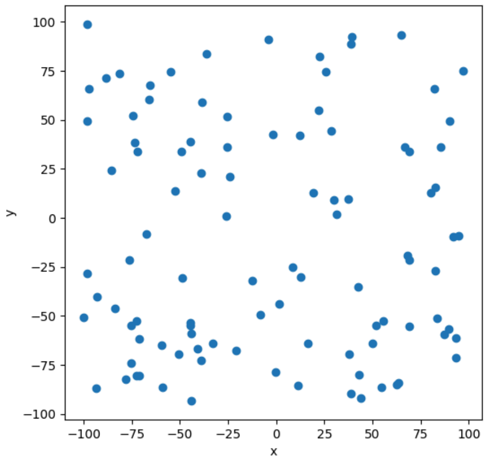
3.4. Przygotowane generatory zbiorów testowych

Plik **tests.py**.

3.4.1. generate_uniform_points

Generuje zbiór **n** losowych punktów leżących w obszarze [**left**,**right**] × [**left**,**right**], gdzie **left**, **right** oraz **n** to parametry generatora.

Poniżej, na rysunku 25, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_uniform_points** i parametrów **n** = 100, **left** = −100, **right** = 100.

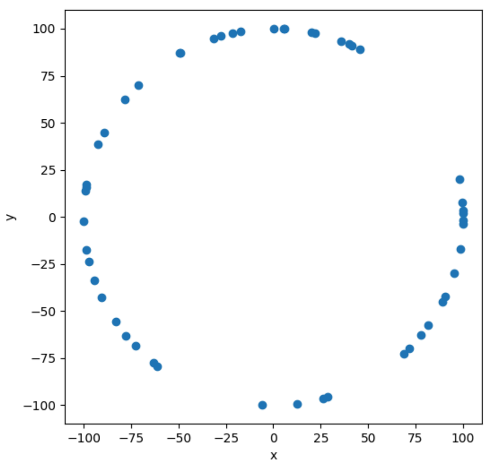


Rysunek 25: przykładowy zbiór punktów

3.4.2. generate_circle_points

Generuje zbiór **n** losowych punktów leżących na kole o środku w punkcie **O** oraz promieniu **R**, gdzie **O**, **R** oraz **n** to parametry generatora.

Poniżej, na rysunku 26, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_circle_points** i parametrów **n** = 50, **O** = (0,0), **R** = 100.

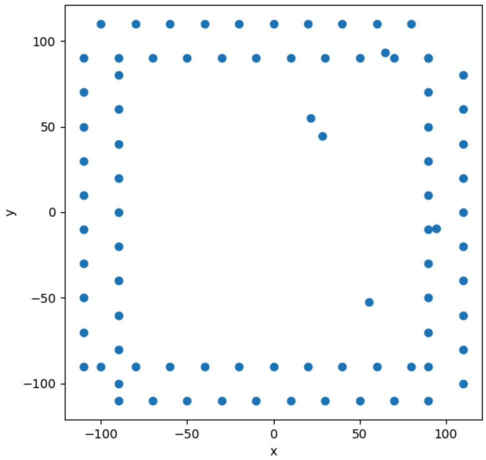


Rysunek 26: przykładowy zbiór punktów

3.4.3. generate_zigzag_points

Generuje zbiór **n** losowych punktów leżących na prostokącie o środku w początku układu współrzędnych, oraz o długościach boków **width** oraz **height**, które są parametrami generatora. Punkty dodatkowo otoczone są naprzemienną obramówką punktów, tak jak widać na rysunku 27. Szerokość obramówki definiuje parametr **amplitude**, a częstość punktów na niej parametr **period**.

Poniżej, na rysunku 27, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_zigzag_points** i parametrów **n** = 5, **width** = 200, **height** = 200, **amplitude** = 10, **period** = 10.

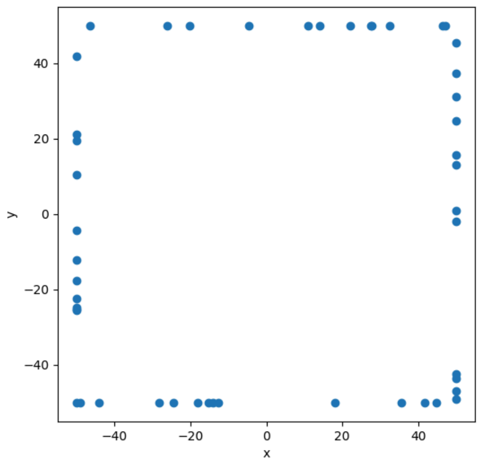


Rysunek 27: przykładowy zbiór punktów

3.4.4. generate_square_points

Generuje zbiór **n** losowych punktów leżących na kwadracie o środku w początku układu współrzędnych i o boku długości **a**, gdzie **a** to parametr generatora.

Poniżej, na rysunku 28, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_square_points** i parametrów **n** = 50, **a** = 100.

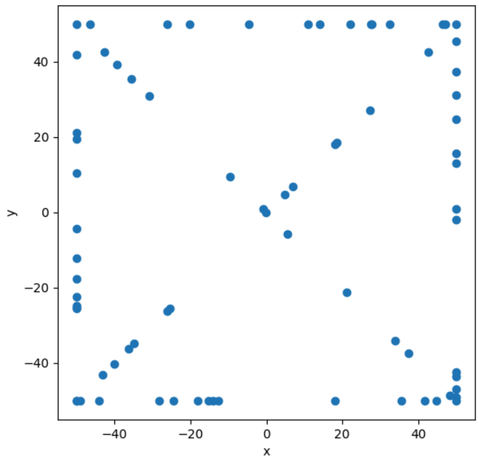


Rysunek 28: przykładowy zbiór punktów

3.4.5. generate_x_square_points

Generuje zbiór **n** losowych punktów leżących na kwadracie o środku w początku układu współrzędnych i o boku długości **a**, lub na jego przekątnych gdzie, **a** to parametr generatora. Ponadto generator dodaje do zbioru wynikowego 4 punkty będące wierzchołkami kwadratu.

Poniżej, na rysunku 28, znajduje się wizualizacja przykładowego zbioru punktów wygenerowana z użyciem generatora **generate_x_square_points** i parametrów **n** = 50, **a** = 100.



Rysunek 29: przykładowy zbiór punktów

3.5. Testy algorytmów na przygotowanych zbiorach