

Laboratorium 2

Otoczka Wypukła

Łukasz Kwinta

1 Dane Techniczne

Procesor: AMD Ryzen 7 5700U

System operacyjny: Ubuntu 20.04 w środowisku WSL 2 na Windows 11 x64

Pamięć ram: 32 GB DDR4

Środowisko i język: Python 3.9 + Jupyter Notebook w środowisku Anaconda
Wykresy tworzyłem przy pomocy narzędzia przygotowanego przez KN Bit, do obliczeń numerycznych używałem biblioteki numpy. Dane przechowywałem w zmiennych typu float – typ danych o rozmiarze 64 bitów, odpowiednik typu double w języku C.

2 Opis Realizacji Ćwiczenia

Celem ćwiczenia była implementacja algorytmów Grahama i Jarvisa do wyznaczania otoczki wypukłej chmury punktów oraz porównanie ich precyzji oraz wydajności.

2.1 Generowanie chmur punktów

Pierwszym krokiem było wygenerowanie czterech zbiorów danych, które posłużą za pierwotne przetestowania algorytmu oraz wizualizację jego działania. Wszystkie punkty mają współrzędne 2D, typu rzeczywistego:

- a) zawierający 100 losowo wygenerowanych punktów o współrzędnych z przedziału $[-100, 100]$
- b) zawierający 100 losowo wygenerowanych punktów leżących na okręgu o środku $(0, 0)$ i promieniu $R = 10$
- c) zawierający 100 losowo wygenerowanych punktów leżących na bokach prostokąta o wierzchołkach $(-10, 10), (-10, -10), (10, -10), (10, 10)$
- d) zawierający wierzchołki kwadratu $(0, 0), (10, 0), (10, 10), (0, 10)$ oraz punkty wygenerowane losowo w sposób następujący: po 25 punktów na dwóch bokach kwadratu leżących na osiach i po 20 punktów na przekątnych kwadratu

Liczby pseudolosowe generowałem przy pomocy funkcji bibliotecznej z biblioteki Pythona `random.uniform(left, right)` która generuje liczby typu float (odpowiednik `double` z języka C) z przedziału domkniętego $[left, right]$.

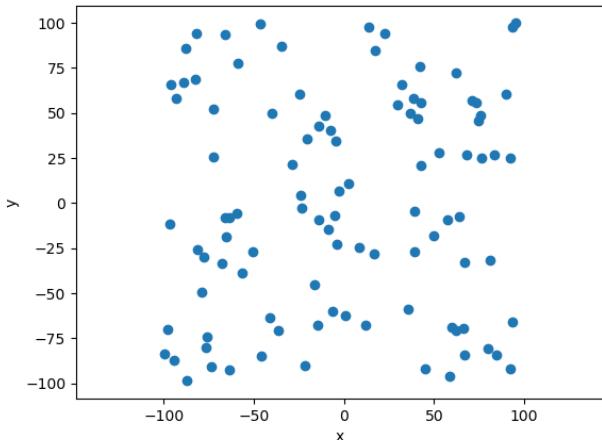
Punkty ze zbioru b) generowałem przy użyciu postaci parametrycznej okręgu postaci:

$$(x, y) = (R \cos(2\pi t), R \sin(2\pi t)) \quad \text{dla} \quad t \in [0, 1]$$

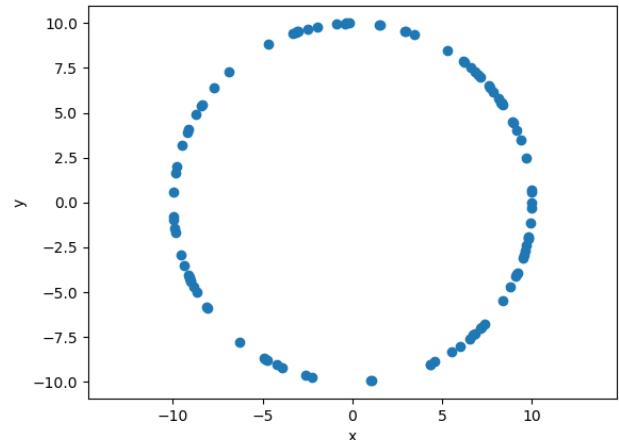
Użyłem funkcji trygonometrycznych z biblioteki `numpy`, a parametr t był zmienną losowaną.

Punkty ze zbioru c) losowałem poprzez wylosowanie najpierw jednej liczby z przedziału $[0, L]$, gdzie L to obwód prostokąta, następnie sprawdzałem na którym boku znalazła się liczba, po czym losowałem jeden punkt na odpowiednim boku.

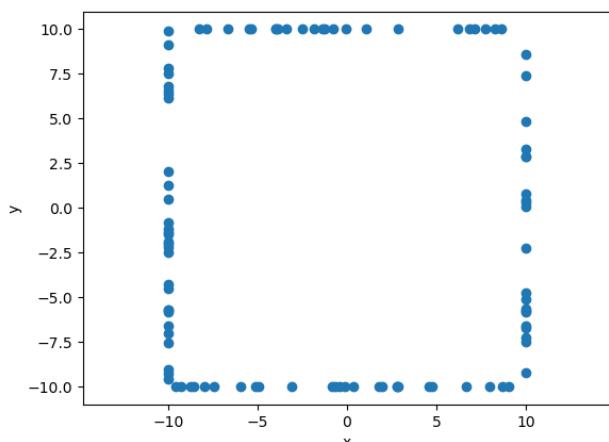
Do losowania punktów na bokach i przekątnych kwadratu użyłem funkcji generującej punkty na prostej przechodzącej przez zadane punkty zapożyczonej z *Laboratorium 1*.



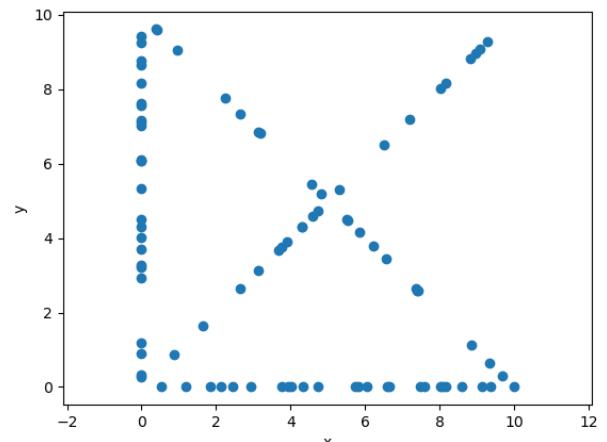
a Wizualizacja zbioru a)



b Wizualizacja zbioru b)



c Wizualizacja zbioru c)



d Wizualizacja zbioru d)

Tabela 1: Wizualizacje zbiorów

3 Algorytm Grahama

Implementując algorytm Grahama, do sortowania punktów względem kąta użyłem wyznacznika macierzy 2×2 ze wzoru z *Laboratorium 1*.

$$\det(a, b, c) = (b.x - a.x)(c.y - b.y) - (b.y - a.y)(c.x - b.x)$$

Aby zaoszczędzić czas na wykonywania operacji pierwiastkowania, nie obliczałem dokładnej odległości między punktami lecz porównywałem punkty na podstawie kwadratu odległości.

Do obliczeń przyjąłem dokładność zera: $\varepsilon = 10^{-32}$.

Dla tej dokładności algorytm dawał oczekiwane wyniki w testach przygotowanych przez KN Bit.

Uproszczony schemat działania algorytmu:

1. Znalezienie punktu p_0 o najmniejszej współrzędnej x i y
2. Posortowanie punktów względem kąta jaki tworzą z punktem p_0
3. Usunięcie wspólniowych punktów z posortowanego zbioru
4. Pamiętając 2 ostatnie punkty należące do otoczki szukamy punktów o najmniejszym kącie z nimi

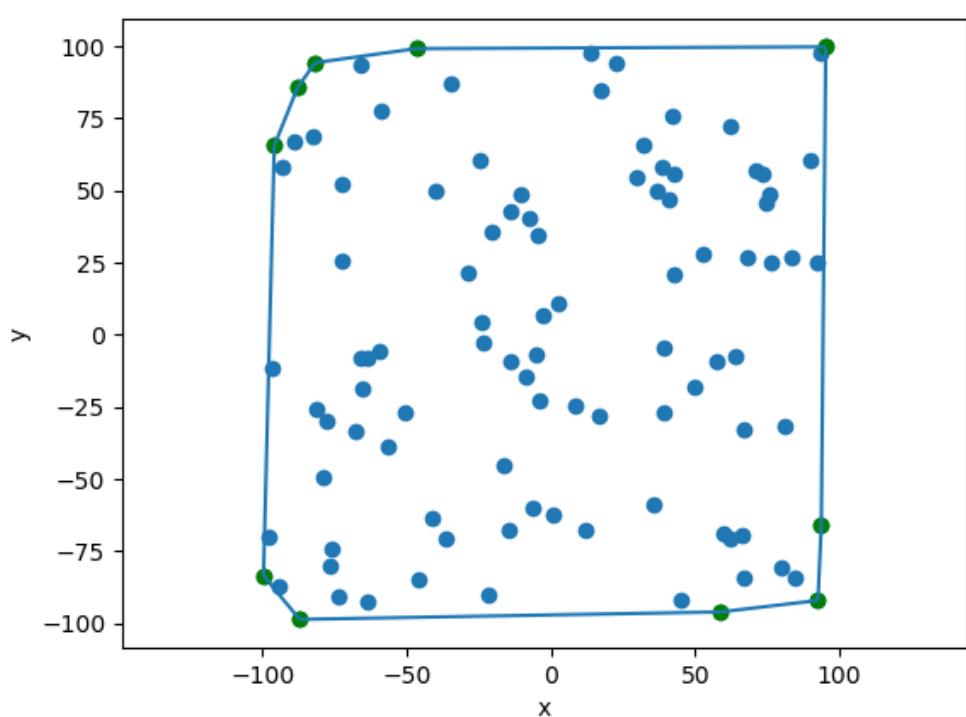
Do sortowania użyłem funkcję wbudowaną w Pythona `sorted()`, używając funkcji `cmp_to_key`, z biblioteki `functools`. Teoretyczna złożoność tego algorytmu to:

$$O(n \log n)$$

Poniżej prezentuję wyniki działania algorytmu na poszczególnych zbiorach. W animacjach na niebiesko zaznaczam punkty ze zbioru wejściowego, na czerwono punkty ze zbioru po usunięciu wspólniowych (punkty brane pod uwagę przy wyznaczaniu otoczki), a na zielono punkty należące do otoczki (są też one połączone linią).

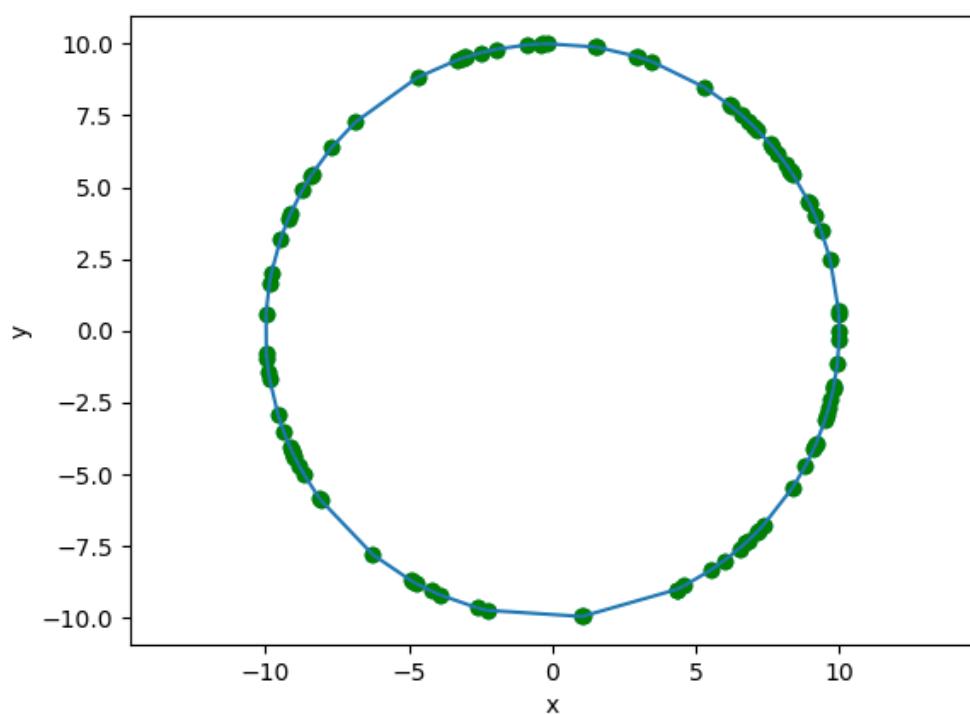
Na rysunkach pokazujących wynik wyznaczania otoczki, na niebiesko zaznaczone są punkty wejściowe, a na zielono punkty należące do otoczki (połączono je też linią).

Rysunek 1: Animacja pokazująca działanie algorytmu Grahama na zbiorze a)



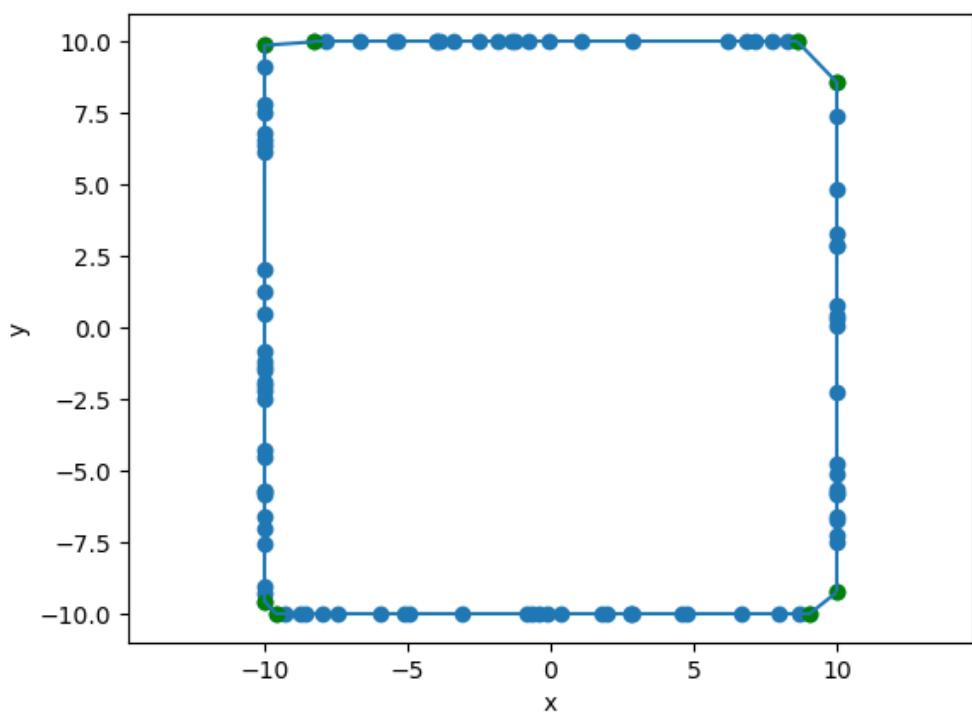
Rysunek 2: Wynik działania algorytmu Grahama na zbiorze a)

Rysunek 3: Animacja pokazująca działanie algorytmu Grahama na zbiorze b)



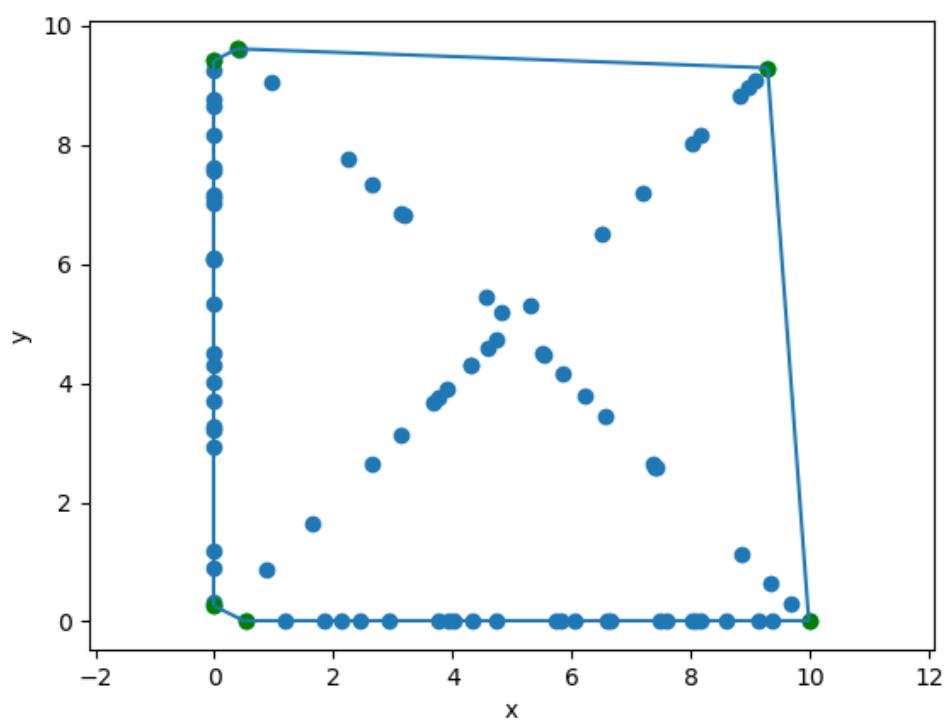
Rysunek 4: Wynik działania algorytmu Grahama na zbiorze b)

Rysunek 5: Animacja pokazująca działanie algorytmu Grahama na zbiorze c)



Rysunek 6: Wynik działania algorytmu Grahama na zbiorze c)

Rysunek 7: Animacja pokazująca działanie algorytmu Grahama na zbiorze d)



Rysunek 8: Wynik działania algorytmu Grahama na zbiorze d)

4 Algorytm Jarvisa

Wybrałem implementację algorytmu, która od razu wyznacza całą otoczkę, a nie osobno lewą i prawą stronę.

Do implementacji algorytmu Jarvisa, użyłem tak jak w przypadku algorytmu Graha-ma, użyłem wyznacznika 2×2 oraz funkcji obliczającej kwadrat odległości pomiędzy punktami.

Podobnie jak wcześniej, tutaj też za dokładność zera przyjąłem wartość: $\varepsilon = 10^{-32}$. Dla takiej wartości algorytm przechodził testy KN Bit.

Uproszczony schemat działania algorytmu:

1. Wybranie punktu z najmniejszymi współrzędnymi y i x
2. Wybranie kandydata na następny punkt otoczki
3. Znalezienie punktu który z ostatnimi punktami otoczki tworzy najmniejszy kąta
4. Znaleziony punkt jest kolejnym punktem otoczki
5. Wróć do punktu 2 o ile nie doszliśmy do pierwszego punktu otoczki

Algorytm Jarvisa ma teoretyczną złożoność:

$$O(n^2)$$

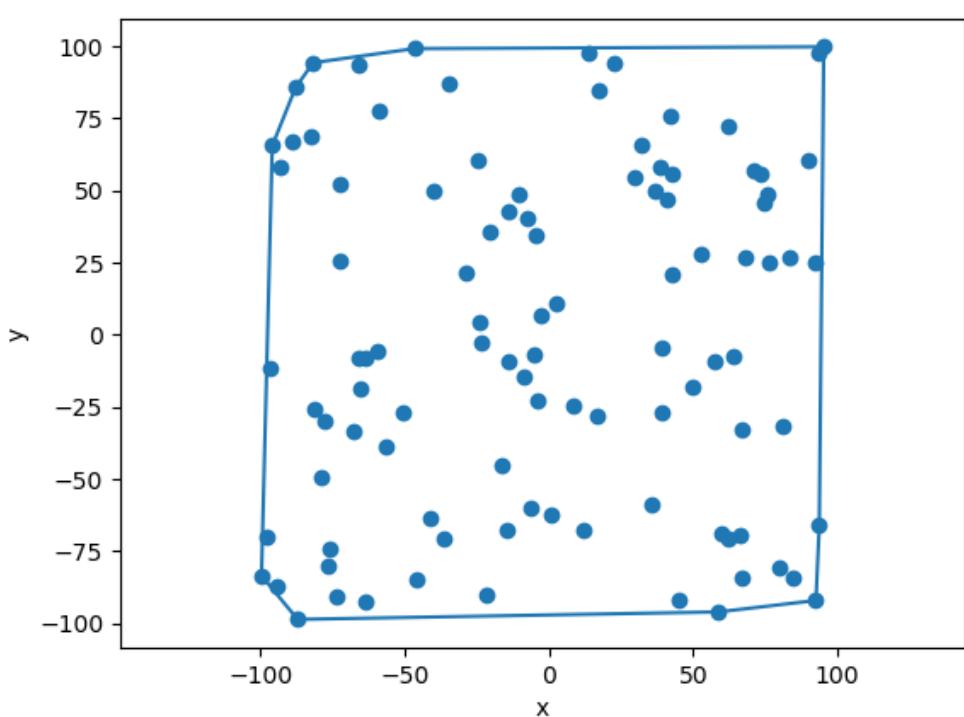
lecz jest on ciekawy w przypadkach gdy liczba punktów otoczki jest niewielka, wtedy możemy ograniczyć jego złożoność przez stałą k która oznacza liczbę punktów w otoczce:

$$O(kn)$$

Poniżej prezentuję animacje i wyniki działania algorytmu Jarvisa dla zadanych zbiorów danych. Na animacjach na niebiesko oznaczono punkty wejściowe, na zielono, połączone liniami oznaczyłem punkty tworzonej otoczki. Punkty które algorytm sprawdza w danym kroku też są oznaczone jakby to był punkt otoczki lecz można łatwo zauważyc, że są to kolejne próby znalezienia optymalnego punktu. Warto zwrócić uwagę, że o ile w przypadku animacji dla algorytmu Graha-ma, zaznaczona był każdy testowany punkt, tak w tym przypadku jako punkty przejściowe, które algorytm "testuje" zaznaczone są tylko kolejne potencjalne minima.

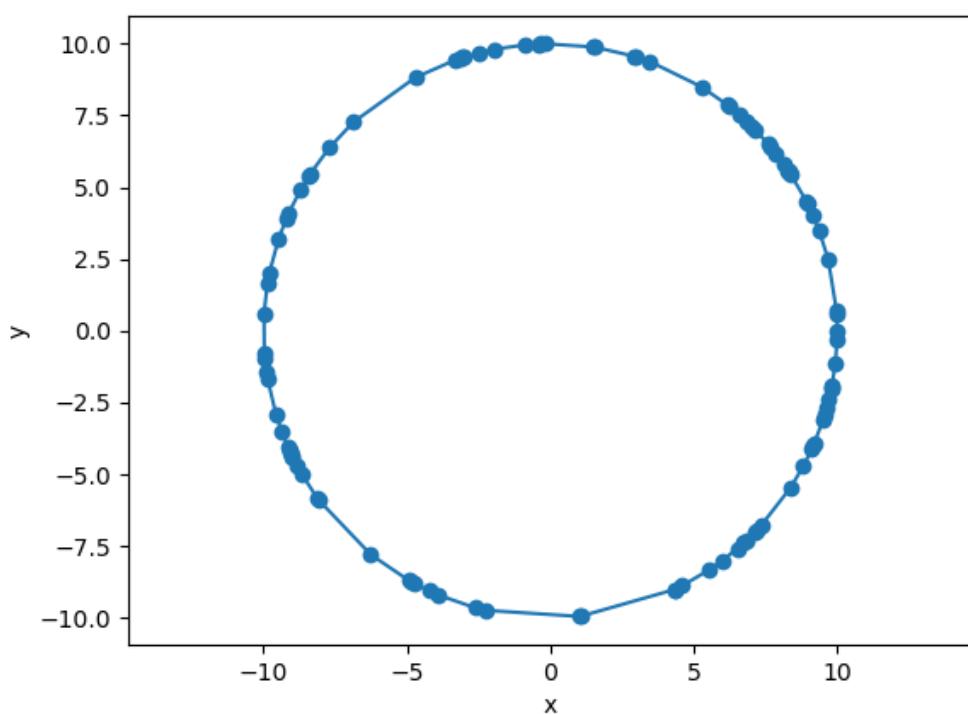
Na ilustracjach pokazujących wynik działania algorytmu na niebiesko zaznaczono punkty wejściowe, na zielono punkty otoczki , dodatkowo połączone liniami.

Rysunek 9: Animacja pokazująca działanie algorytmu Jarvisa na zbiorze a)



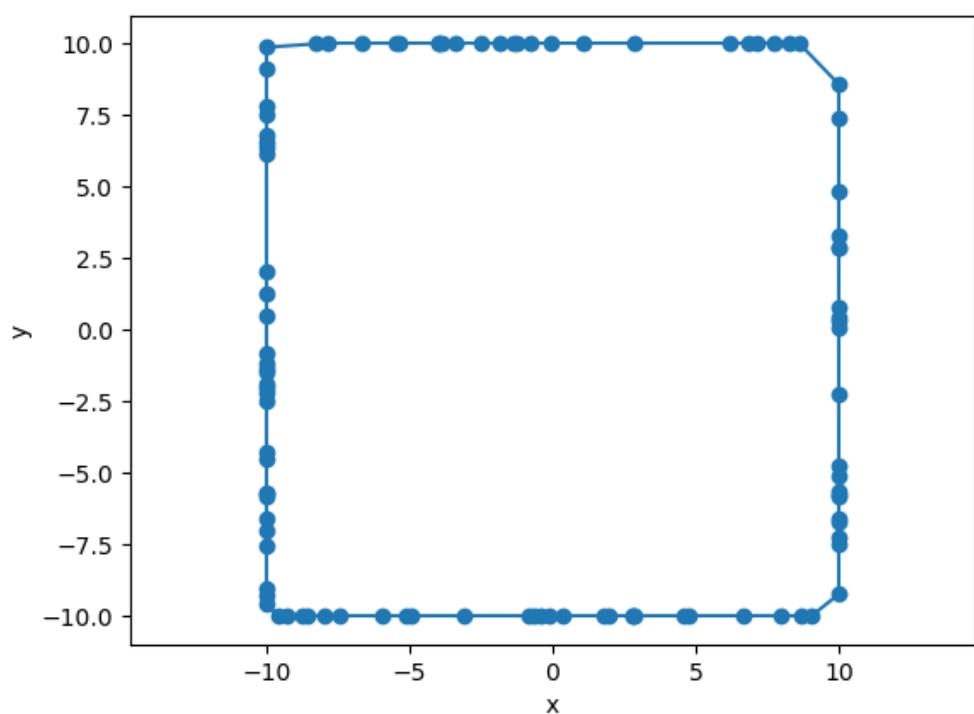
Rysunek 10: Wynik działania algorytmu Jarvisa na zbiorze a)

Rysunek 11: Animacja pokazująca działanie algorytmu Jarvisa na zbiorze b)



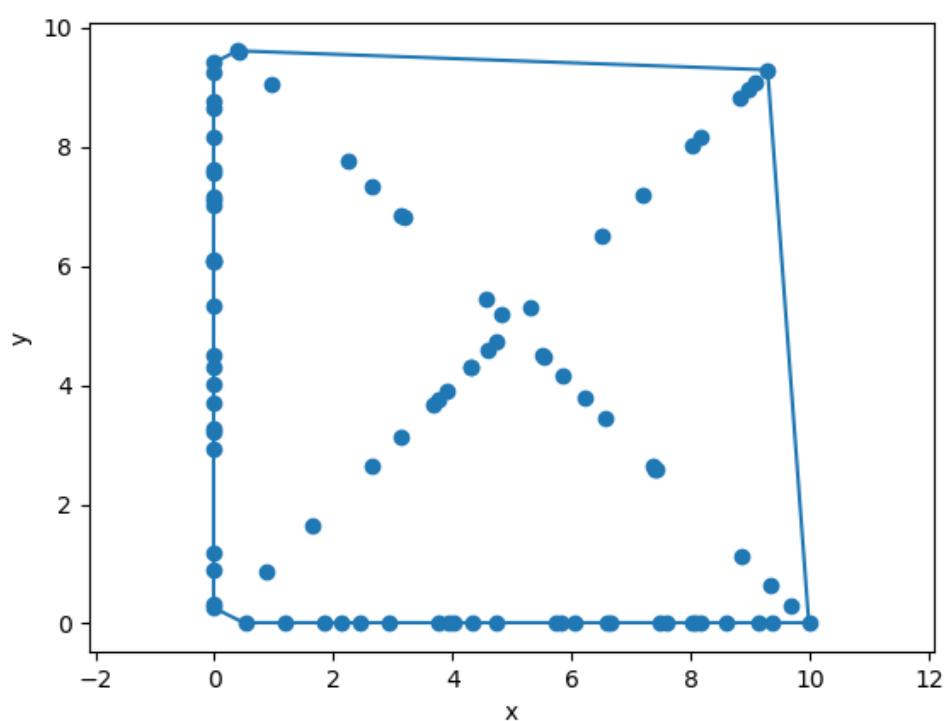
Rysunek 12: Wynik działania algorytmu Jarvisa na zbiorze b)

Rysunek 13: Animacja pokazująca działanie algorytmu Jarvisa na zbiorze c)



Rysunek 14: Wynik działania algorytmu Jarvisa na zbiorze d)

Rysunek 15: Animacja pokazująca działanie algorytmu Jarvisa na zbiorze d)



Rysunek 16: Wynik działania algorytmu Jarvisa na zbiorze d)

Już na pierwszy rzut oka widać różnicę w działaniu algorytmów. Algorytm Jarvisa wybiera punkty po kolejności ich ułożenia w tablicy, a algorytm Grahama, dzięki posortowaniu punktów, wybiera punkty względem własności. Najbardziej to widać na podstawie zbioru c), gdzie algorytm Grahama na Rysunku 3 wybiera po kolejności punkty i są one od razu kwalifikowane jako punkty otoczki, a algorytm Jarvisa na Rysunku 11 skacze "losowo" po kolejnych punktach zanim trafi na punkt będący częścią otoczki.

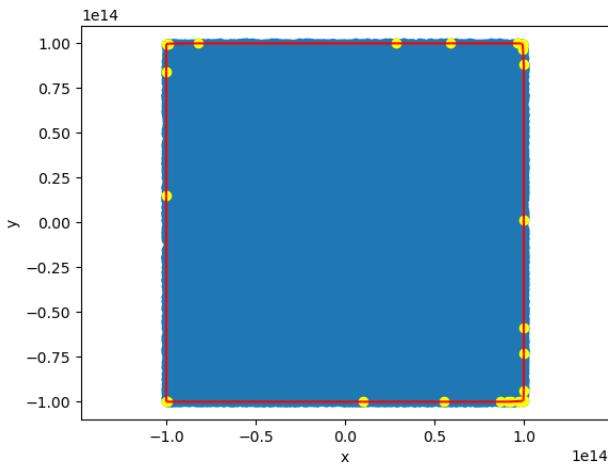
5 Modyfikacja Zbiorów

Modyfikując zbiory do testów, chciałem sprawdzić jak algorytmy zachowają się dla dużych chmur punktów oraz gdy współrzędne punktów są duże - prowadzi to do sytuacji gdzie wyznaczniki są duże, a to z kolei prowadzi do zmniejszenia precyzji liczb zmiennoprzecinkowych. Dlatego postanowiłem zwiększyć liczbę punktów i współrzędne punktów. Nowe zbiory danych przedstawiają się w następujący sposób.

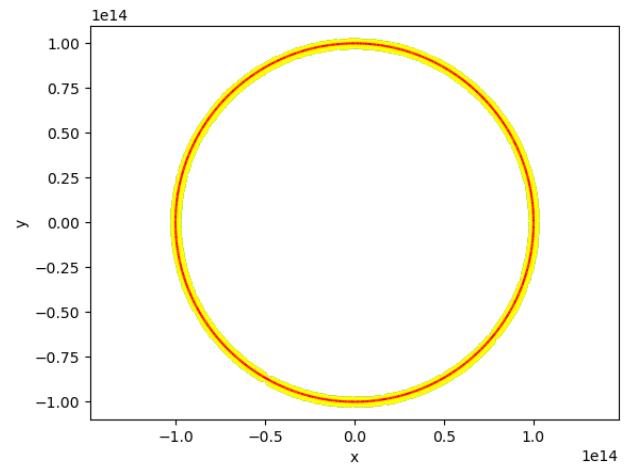
- a') zawierający 10^5 losowo wygenerowanych punktów o współrzędnych z przedziału $[-10^{14}, 10^{14}]$
- b') zawierający 10^3 losowo wygenerowanych punktów leżących na okręgu o środku $(0, 0)$ i promieniu $R = 10^{14}$
- c') zawierający 10^5 losowo wygenerowanych punktów leżących na bokach prostokąta o wierzchołkach $(-10^{14}, 10^{14}), (-10^{14}, -10^{14}), (10^{14}, -10^{14}), (10^{14}, 10^{14})$
- d') zawierający wierzchołki kwadratu $(0, 0), (10^{14}, 0), (10^{14}, 10^{14}), (0, 10^{14})$ oraz punkty wygenerowane losowo w sposób następujący: po 10^5 punktów na dwóch bokach kwadratu leżących na osiach i po 10^5 punktów na przekątnych kwadratu

Niestety dla zbioru b' nie udało mi się przetestować większej liczby punktów, gdyż algorytm Jarvisa działał tak wolno, że Jupyter Notebook się restartował po kilku minutach działania algorytmu.

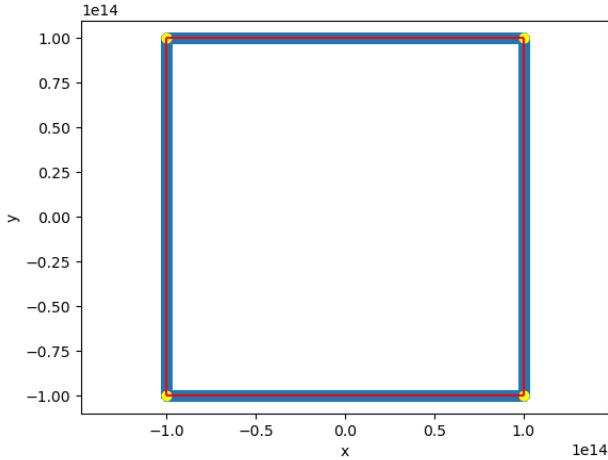
Poniżej zamieszczam wizualizacje wyników działania obu algorytmów (oba dały te same wyniki), dla nowych zbiorów testowych. Na niebiesko zaznaczone są punkty wejściowe, a na żółto punkty należące do otoczki (połączone czerwoną linią).



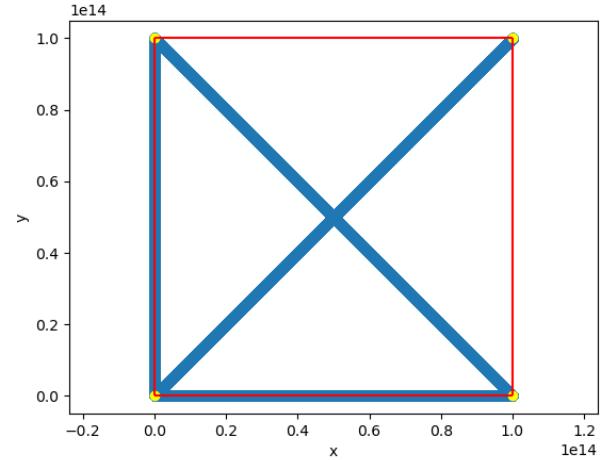
a Wizualizacja wyniku działania algorytmów dla testowego zbioru a')



b Wizualizacja wyniku działania algorytmów dla testowego zbioru $b')$



c Wizualizacja wyniku działania algorytmów dla testowego zbioru $c')$



d Wizualizacja wyniku działania algorytmów dla testowego zbioru $d')$

Tabela 2: Wizualizacje zbiorów testowych

6 Testy Wydajnosci

Poniżej prezentuję tabelę zestawiającą wyniki testów wydajnościowych obu algorytmów. Przy okazji testów chciałem też sprawdzić zachowanie algorytmu dla ekstremalnych danych, więc wypisałem ilości punktów zaklasyfikowanych do otoczki.

Zbiór	Czas [s]		Ilość punktów w otoczce	
	Graham	Jarvis	Graham	Jarvis
Zbiór A	0.000418	0.001040	8	8
Zbiór B	0.000982	0.016427	100	100
Zbiór C	0.000442	0.002085	8	8
Zbiór D	0.000312	0.000546	4	4
Zbiór A'	0.928114	2.826157	27	27
Zbiór B'	0.007917	1.275697	1000	1000
Zbiór C'	1.188857	1.126793	8	8
Zbiór D'	9.745127	1.795051	4	4

Tabela 3: Tabela zestawiająca czas obliczania otoczki dla danych zbiorów przez oba algorytmy

Jak widać w Tabeli 3 algorytm Grahama jest szybszy od algorytmu Jarvisa w przypadkach ogólnych o rzad wielkości.

Jednakże, dla zbiorów C' i D', gdzie liczba punktów w otoczce jest ograniczona, algorytm Jarvisa okazuje się szybszy przez ograniczenie złożoności obliczeniowej.

W przypadkach gdy wszystkie wierzchołki należą do otoczki algorytm Jarvisa okazuje się dużo wolniejszy od algorytmu Grahama co pokazuje zbiór B'.

Warto tutaj, też wspomnieć, że dzięki temu, że funkcja sortująca punkty w algorytmie Grahama jest funkcją wbudowaną w język Python i jest zaimplementowana w języku C, przez to działa ona dużo szybciej niż kod Pythona, który jest interpretowany. Powoduje to sytuację, że przewaga algorytmu Jarvisa, w przypadkach ograniczonej otoczki, nie jest aż tak duża jak wynikałoby to z teoretycznej złożoności obliczeniowej.

7 Wnioski

W moich testach ewidentnie szybszy okazał się algorytm Grahama, co zresztą przewiduje teoretyczna złożoność algorytmów. Algorytm Jarvisa błyszczy gdy liczba punktów w otoczce jest niewielka, lecz w pozostałych przypadkach, przeszukiwanie całego zbioru danych dla każdego punktu otoczki bardzo spowalnia działanie algorytmu, w tym miejscu pojawia się przewaga algorytmu Grahama, który sortuje punkty, dzięki czemu, później samą otoczkę może wyznaczyć liniowo.

Warto jeszcze poruszyć kwestię poprawności działania algorytmów. Oba algorytmy dały taki sam wynik dla skrajnych danych, oba dały wyniki oczekiwane w testach przygotowanych przez KN Bit i w końcu wyniki moich testów są racjonalne, więc z całą pewnością można stwierdzić, że algorytmy działają przewidywalnie. Na pewno nie do końca poprawnie, bo na pewno istnieją dane dla których przyjęta przeze mnie precyzja obliczeń nie jest odpowiednia i wynik działania programu.