Algorytmy geometryczne
Sprawozdanie z laboratorium 2.

Michał Nożkiewicz
gr. Czw_11:20_B

1. Użyte narzędzia i oprogramowanie

Ćwiczenie realizowałem na komputerze z systemem Windows 10 x64**,**
Procesor: Intel Core i5-4460 @3.2 GHz
Pamięc RAM: 8GB

Kod programu pisałem w środowisku Jupyter notebook, w języku Python, z wykorzystaniem bibliotek *numpy, math, random, time* oraz *matplotlib.*

2. Wstęp teoretyczny

Zadaniem tego laboratorium było zaimplementowanie algorytmu znajdowania otoczki wypukłej danego zbioru danych, używając algorytmów Grahama oraz Jarvisa.

2.1 Algorytm Grahama

Opis algorytmu

- Pierwszym krokiem algorytmu jest znaliezienie punktu p_0 o najmniejszej współrzędnej na osi y, w przypadku kilku takich punktów, wybieramy ten o najmniejszej współrzędnej x.
- Następnie pozostałe punkty sortujemy ze względu na kąt, jaki tworzy wektor (p_0, p) z dodatnik kierunkiem osi x. Jeśli kilka punktów tworzy ten sam kąt, wiadomo, że w otoczce wypukłej będzie mógł znaleźć się jedynie ten z tych punktów, który jest najbardziej oddalony od p_0, więc pozostałe może usunąć. Oznaczmy uzyskany ciąg punktów jako p_1, p_2, …, p_n.

- Kolejnym krokiem jest zainicjalizowanie stosu, do którego początkowo wrzucamy punkty p_0, p_1. Następnie przechodzimy kolejne po punktach i wrzucamy je na stos, przy czym jeśli aktualnie rozważany punkt znajduje się na prawo od linii, którą tworzą dwa punkty na szczycie stosu, to zrzucamy ze stosu punkt na jego szczycie.
- Punkty, które po wykonaniu tej procedury znajdują się na stosie tworzą otoczkę wypukłą.

Złożoność

- Znajdowanie p_0 – O(n)
- Sortowanie – O(nlogn)
- Inicjalizacja stosu – O(1)
- Suma operacji na stosie jest liniowa, gdyż w najgorszym przypadku, każdy punkt raz zostanie na niego wrzucony i z niego zrzucony – O(n)
- **Wieć złożoność to O(nlogn)**

2.2 Algorytm Jarvisa(owijanie prezentu)

Opis algorytmu

- Na początku znajdujemy punkt o najmniejsze współrzędnej y, w przypadku kilku o tym samym y, bierzemy ten z najmniejszym x. Dodajemy ten punkt do otoczki
- Następnie znajdujemy punkt, który tworzy najmniejszy kąt z prostą na którą składają się dwa ostatnio dodane punkty. (w przypadku gdy mamy dopiero jeden punkt za tą prostą można przyjąć $y = y_p$,) i dodajemy ten punkt do otoczki. Procedurę powtarzamy, aż wrócimy do początkowego punktu.

Złożoność

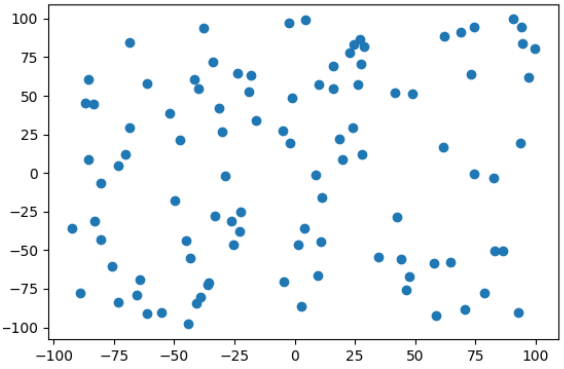
- Znalezienie kolejnego punkty otoczki – O(n)
- Kolejny punkt będziemy znajdować tyle razy ile jest wierzchołków w otoczce – pesymistycznie O(n)
- **Więc złożoność tego algorytmu to $O(n^2)$, jeśli liczba wierzchołków otoczki jest ograniczona przez stałą k to złożoność to O(nk)**

3. Generowanie zbiorów punktów testowych

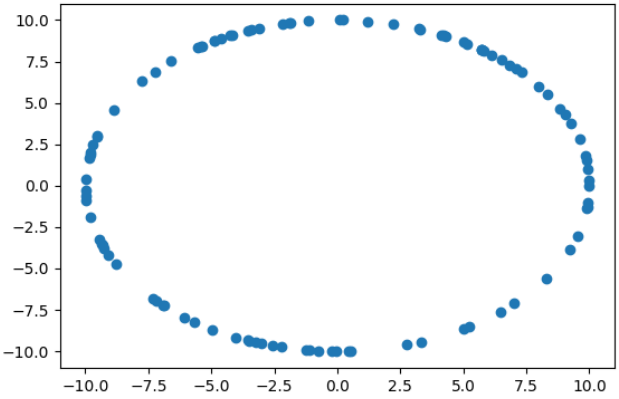
Pierwszym krokiem było wygenerowanie czterech zbiorów punktów.

- Zbiór A
  100 punktów o współrzędnych z przedziału [-100, 100],
- Zbiór B
  100 punktów leżących na okręgu o środku (0, 0) i promieniu R = 10,
- Zbiór C
  100 punktów leżących na bokach prostokąta o wierzchołkach (-10, 10), (-10, -10), (10, -10), (10, 10),
- Ziór 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.

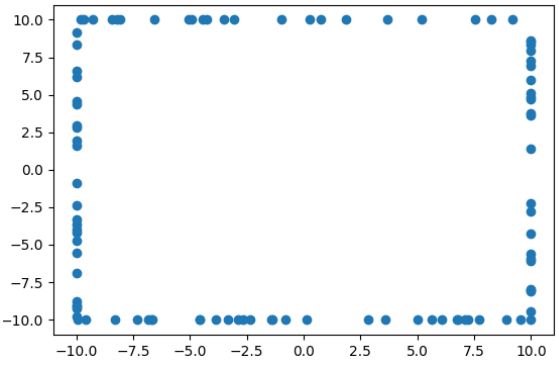
Wykres 3.1 Zbiór A



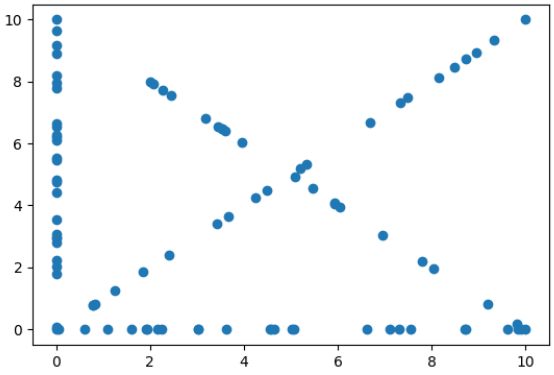
Wykres 3.2 Zbiór B



Wykres 3.3 Zbiór C



Wykres 3.4 Zbiór D



Następnie zaimplementowałem funkcje do generowania zbiorów takich jak zbiory A, B, C i D, lecz przyjmujące dodatkowe parametry.

- generate_a
  Funkcja jako parametry przyjmuje n, czyli ilość punktów do wylosowania, a także przedział, z którego powinny być losowane punkty, a następnie losuje n punktów z tego przedziału.
- generate_b
  Funkcja losuje dowolną ilość punktów na okręgu, o podanym promieniu i współrzędnych środka.
- generate_c
  Funkcja losuje dowolną ilość punktów na prostokącie. Przyjmuje współrzędne lewego dolnego wierzchołka oraz współrzędne prawego górnego wierzchołka. Aby punkty były wylosowane jednostajnie losowałem liczbę z przedziału [0, obwód prostokąta], a następnie odpowiednio przydzielałem jej współrzędne.
- generate_d
  Funkcja generuje zbiory analogiczne do Zbioru D, lecz dodatkowo przyjmuje, lewy dolny wierzchołek kwadratu, bok kwadratu, a także parametry mówiące po ile punktów ma być na bokach, a ile na przekątnych.

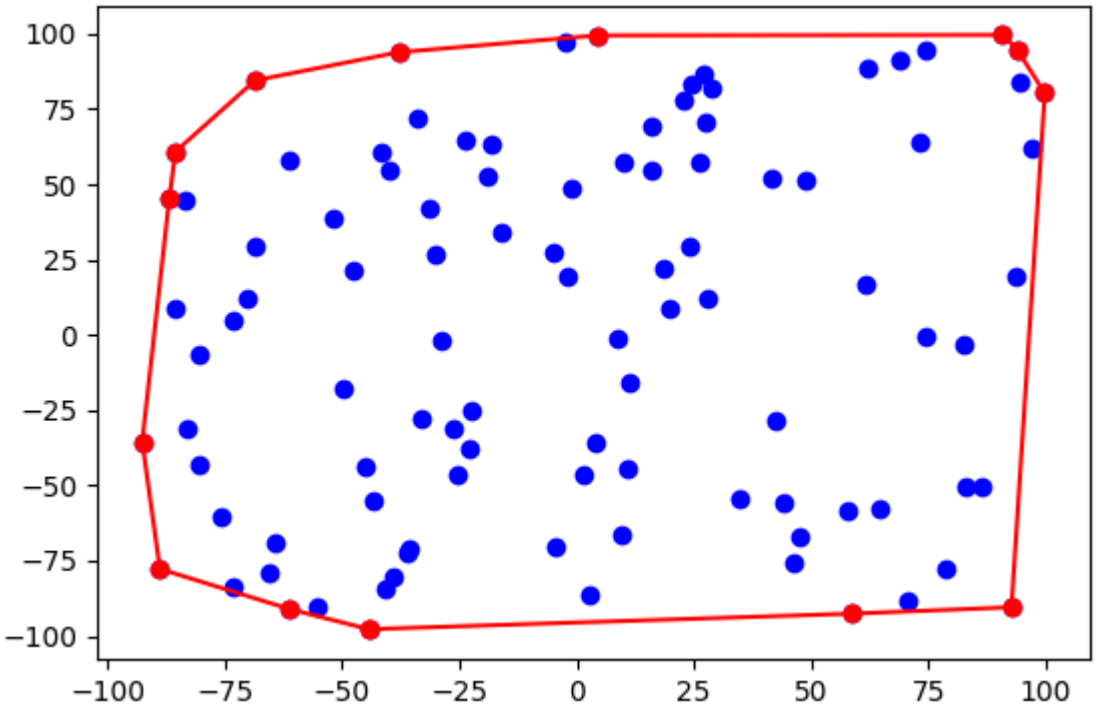
4. Metoda obliczenia wyznacznika

Wzorując się wynikami z poprzedniego laboratorium zdecydowałem, że najefektywniejszy, a także najbardziej dokładny jest wyznacznik 3x3 własnej implementacji, a jako tolerancje dla zera przyjąłem liczbę 10^{-10}.

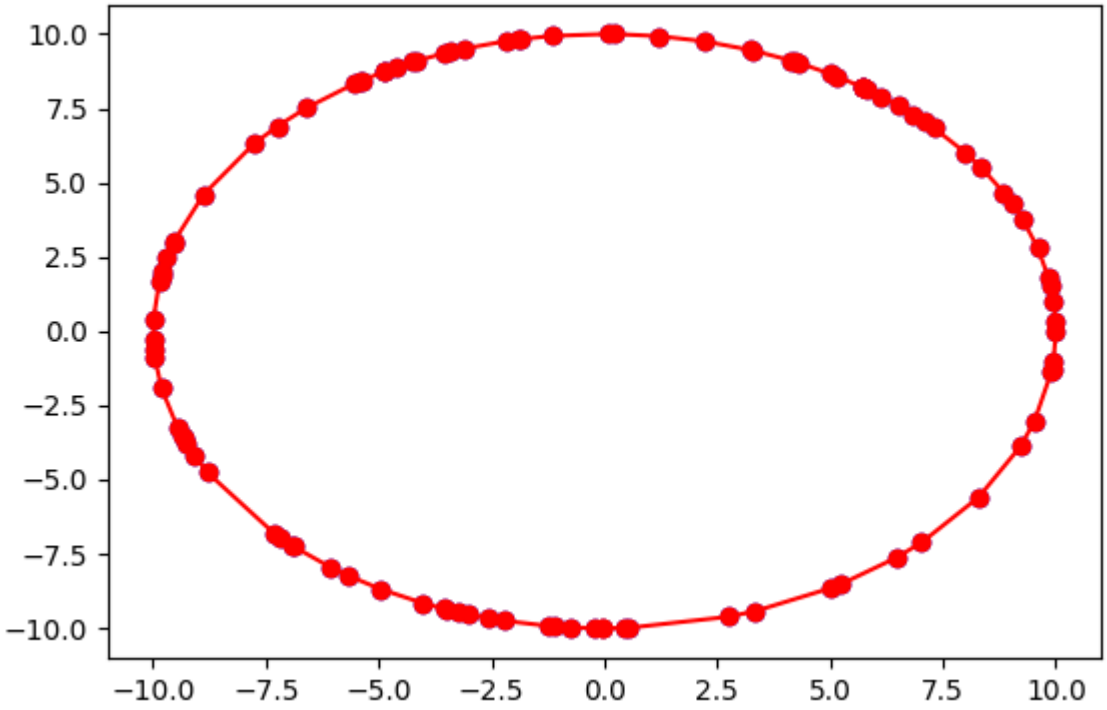
5. Wyniki działania obu algorytmów

Algorytmy testowałem na tych samych zbiorach danych. Używałem zbiorów A, B, C i D, a także kilku innych zbiorów wygenerowanych według funkcji z punktu 3. Dla wszystkich zbiorów algorytmy zwróciły te same poprawne wyniki. Poniżej znajdują się ostateczne wyniki działania algorytmów, wizualizacja poszczególnych kroków, jest dostępna w Jupyter Notebooku.

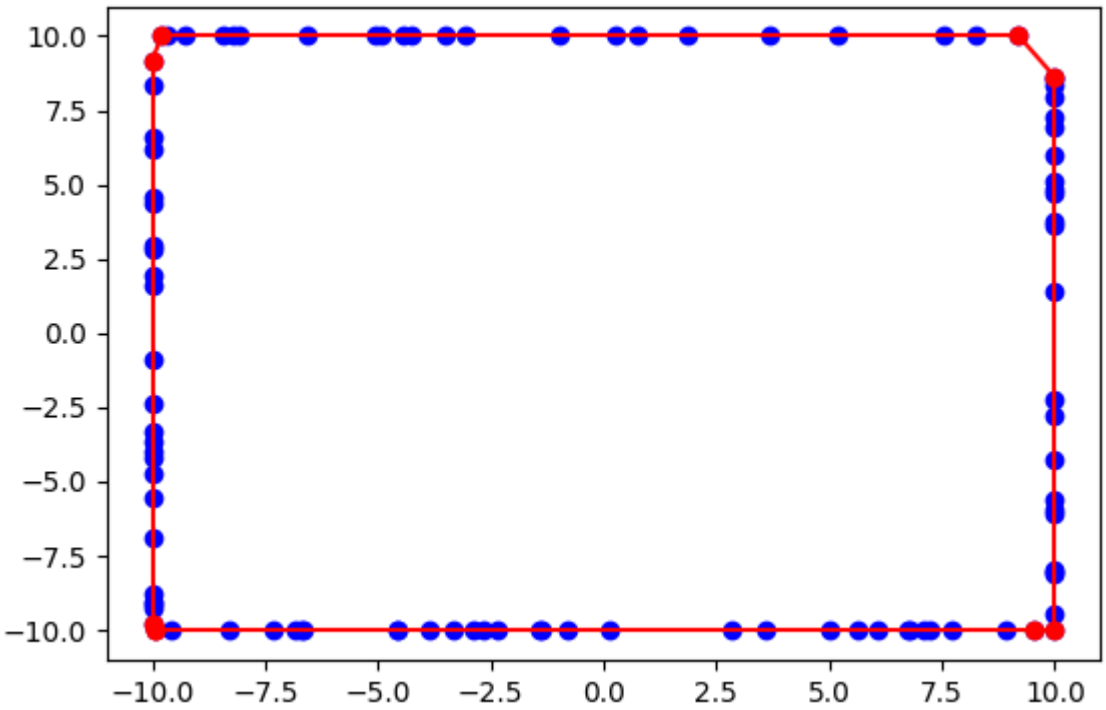
Wykres 3.1 Otoczka Zbioru A



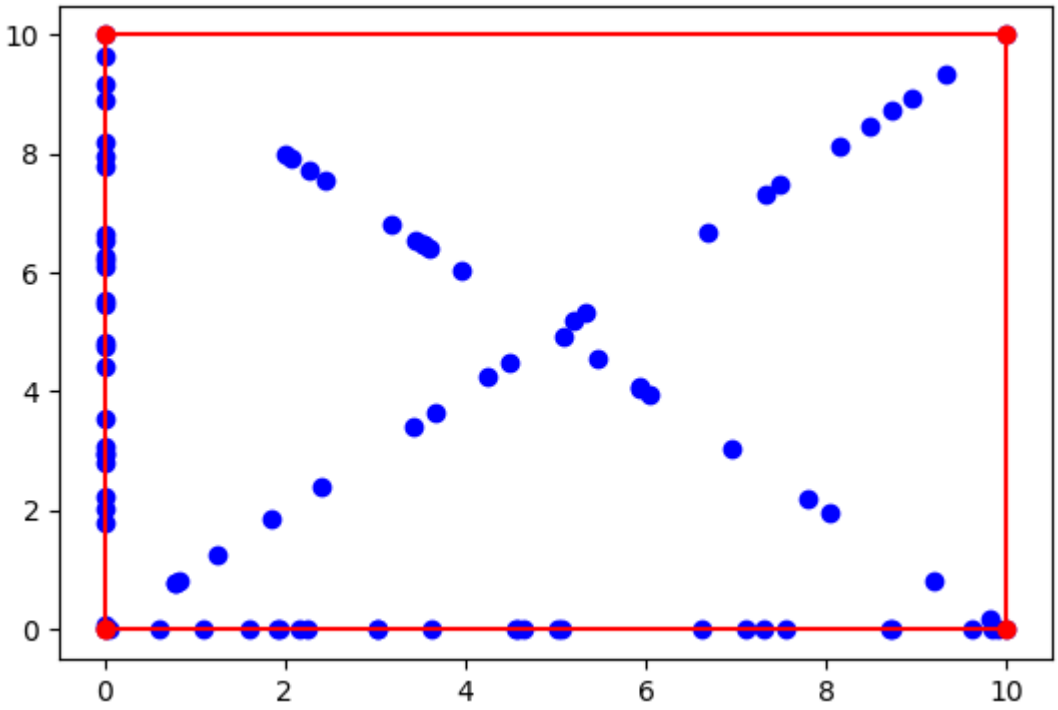
Wykres 3.2 Otoczka Zbioru B



Wykres 3.3 Otoczka Zbioru C



Wykres 3.4 Otoczka Zbioru D



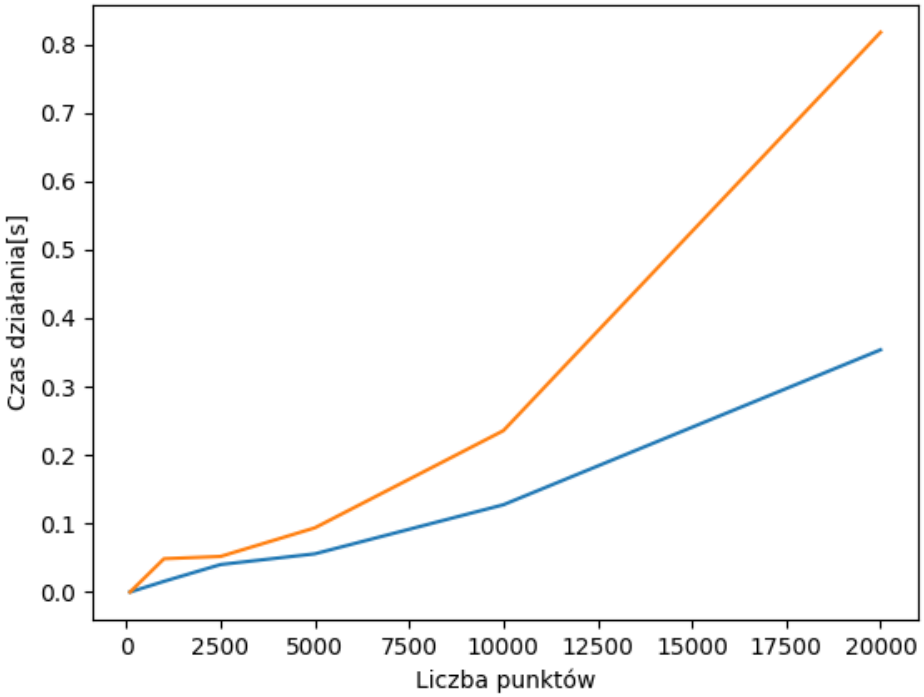
6. Porównanie czasu działania algorytmów

6.1 Zbiory punktów typu A

Tabela 6.1 Czas wykonywania algorytmów Grahama i Jarvisa w zależności od liczby punktów

Liczba punktów	Graham [s]	Jarvis [s]
100	0	0
1000	0,0156	0,0486
2500	0,0401	0,0519
5000	0,0558	0,0937
10000	0,1275	0,2357
20000	0,3537	0,8174

Wykres 6.1 wykres zależności z tabeli 6.1



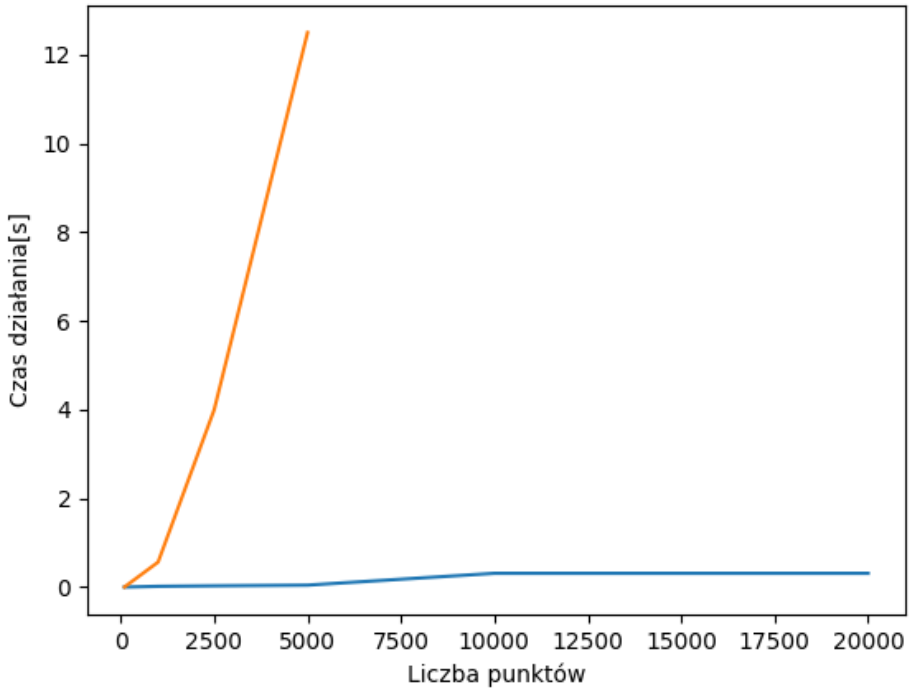
Algorytm Grahama okazał się lepszy dla losowo wygenerowanego zbioru punktów, w każdym przypadku był mniej więcej dwa razy szybszy, widać jednak, że mimo dużo gorszej złożoności pesymistycznej, algorytm jarvisa radzi sobie bardzo dobrze dla losowych danych, gdyż ilość punktów na otoczce jest mała.

6.2 Zbiory punktów typu B

Tabela 6.2 Czas wykonywania algorytmów Grahama i Jarvisa w zależności od liczby punktów

Liczba punktów	Graham [s]	Jarvis [s]
100	0,0029	0,0049
1000	0,0199	0,5659
2500	0,0312	3,996
5000	0,0468	12,4992
10000	0,3131	-
20000	0,3129	-

Wykres 6.2 wykres zależności z tabeli 6.2



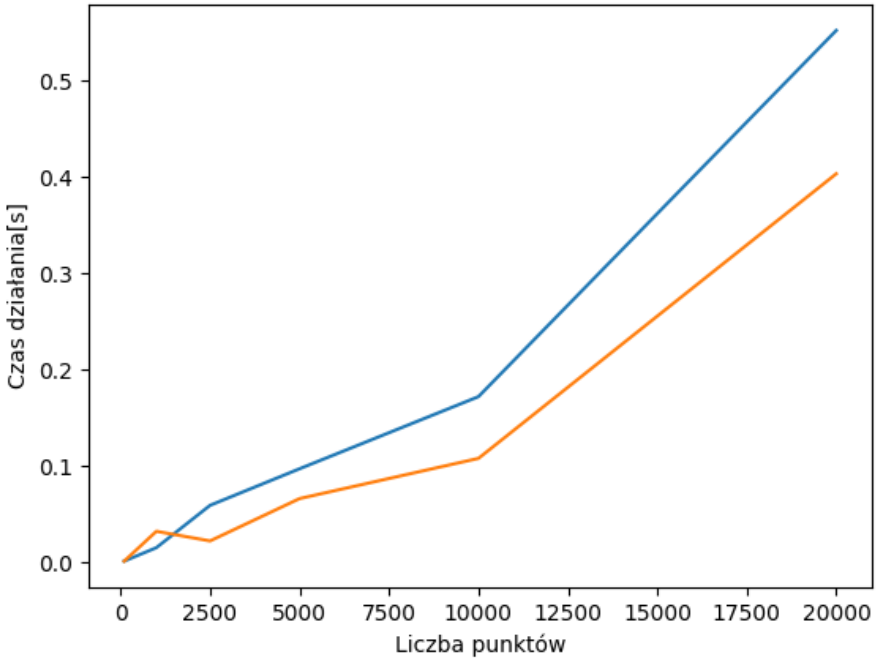
W przypadku okręgu wszystkie punkty należą do otoczki, więc algorytm Jarvisa staje się bardzo nieefektywny i dla kilku tysięcu punktów działa kilkanaście sekund. Czas trwania algorytmu grahama nie różni się wiele od zbioru B.

6.3 Zbiory punktów typu C

Tabela 6.3 Czas wykonywania algorytmów Grahama i Jarvisa w zależności od liczby punktów

Liczba punktów	Graham [s]	Jarvis [s]
100	0,001	0
1000	0,0149	0,0319
2500	0,0589	0,0219
5000	0,0969	0,0659
10000	0,1719	0,1077
20000	0,5525	0,4036

Wykres 6.3 wykres zależności z tabeli 6.3



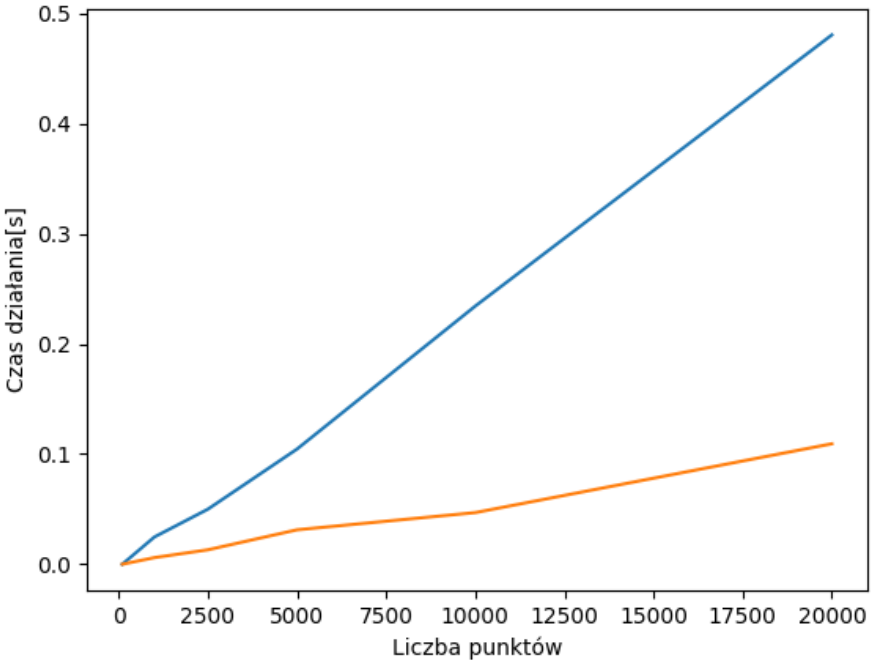
Dla prostokąta nieznacznie szybszy okazał się Jarvis, gdyż otoczka jest ma maksymalnie osiem punktów. Algorytm Grahama działa z podobną szybkością jak w poprzednich przykładach.

6.4 Zbiory punktów typu D

Tabela 6.4 Czas wykonywania algorytmów Grahama i Jarvisa w zależności od liczby punktów

Liczba punktów	Graham [s]	Jarvis [s]
100	0	0
1000	0,0247	0,0059
2500	0,0499	0,0131
5000	0,1045	0,0312
10000	0,2344	0,0468
20000	0,4806	0,1093

Wykres 6.4 wykres zależności z tabeli 6.4



Podobnie jak w 6.3 Jarvis jest szybszy od Grahama, jednak teraz do otoczki zawsze trafiają cztery punkty, więc przewaga Jarvisa jest nawet większa.

7. Wnioski

Jak widać oba algorytmy mogą posłużyć, aby poprawnie wyznaczyć otoczkę wypukłą. Algorytm Grahama dla każdych zbiorów punktów działał w podobnym czasie. Inaczej to wyglądało w przypadku algorytmu Grahama, którego czas zależał od tego jaki zbiór punktów dobraliśmy. Jak widać w punkcie 6.2, łatwo można dobrać zbiór punktów dla którego ten algorytm okaże się bardzo nieefektywny.

Według mnie lepszym wyborem jest jednak algorytm Grahama, gdyż jest on stabilniejszy i dla każdych zbiorów danych będzie działał równie szybko. Być może nie jest tak intuicyjny jak algorytm Jarvis, lecz nie jest od niego trudniejszy implementacyjnie. Jedyna rzecz na którą należy zwrócić szczególną uwagę w przypadku implementacji Grahama jest napisanie dobrego comparatora podczas sortowania punktów.