Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Marcin Permus

Nr albumu: 1102368

Algorytmy geometryczne w języku Python

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2018

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

Pragnę złożyć serdeczne podziękowania Promotorowi Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za wsparcie, okazaną pomoc, wielkie zaangażowanie i cierpliwość podczas powstawania niniejszej pracy, a także za motywację do samodoskonalenia w pisaniu coraz lepszego kodu.
Chciałbym także podziękować moim kochającym Rodzicom za wsparcie, bez którego niniejsza praca nie mogłaby powstać.

Streszczenie

Geometria obliczeniowa zajmuje się badaniem algorytmów i struktur danych rozwiązujących problemy zdefiniowane z użyciem obiektów geometrycznych (punkty, odcinki, proste, wielokąty). W pracy przedstawiono implementację w języku Python wybranych algorytmów geometrii obliczeniowej na płaszczyźnie. Zbudowano szereg klas reprezentujących obiekty geometryczne, a także zebrano narzędzia często używane w algorytmach geometrycznych. Przykładowe operacje to znajdowanie zorientowanego pola równoległoboku, sprawdzanie przecinania się odcinków, sortowanie punktów względem kąta.

Przygotowano trzy algorytmy wyznaczające otoczkę wypukłą dla skończonego zbioru punktów: algorytm Grahama, algorytm Jarvisa i algorytm quickhull.

Zaimplementowano dwa algorytmy sprawdzające zawieranie się punktu w wielokącie: metoda liczby przecięć i metoda liczby nawinięć. Metoda rotujących suwmiarek została wykorzystana do wygenerowania wszystkich par punktów antypodycznych, które z kolei pozwalają szybko wyznaczyć parę najdalszych punktów. Zaimplementowano algorytm wyznaczający orientację wielokąta prostego. Są algorytmy testujące, czy wielokąt jest prosty (metoda siłowa) i czy wielokąt prosty jest wypukły.

Słowa kluczowe: geometria obliczeniowa, wielokąty, otoczka wypukła, obracające się suwmiarki, dwa najdalsze punkty

English title: Geometric algorithms with Python

Abstract

Computational geometry is devoted to study algorithms and data structures solving problems involving geometric objects (points, segments, lines, polygons). Python implementation of selected computational geometry algorithms in the plane is presented. Separate clases representing geometric object were built and tools used in geometric algorithms were collected. Exemplary operations: finding the oriented area of a parallelogram, testing segments intersections, sorting points by angles.

Three algorithms for computing the convex hull for a finite set of points are presented: the Graham scan algorithm, the gift wrapping (Jarvis march) algorithm, and the quickhull algorithm.

Two algorithms solving the point in polygon problem are implemented: the crossing number method and the winding number method. The method of rotating calipers is used to generate all antipodal pairs of points for a convex polygon. Antipodal pairs are used to find a pair of furthest points. The algorithm for finding the orientation of a simple polygon is also implemented. There are algorithms for testing if a polygon is simple (brute force) and if a simple polygon is convex.

Keywords: computational geometry, polygons, convex hull, rotating calipers, two furthest points

Spis treści

Spis tabel 4
Spis rysunków 5
Listings 6
1. Wstęp 7
1.1. Cele pracy 9
1.2. Organizacja pracy 9
2. Geometria obliczeniowa 10
2.1. Wielokąty 10
2.2. Otoczka wypukła 11
2.2.1. Algorytm Grahama 12
2.2.2. Algorytm Jarvisa 13
2.2.3. Algorytm inkrementacyjny 13
2.2.4. Algorytm "dziel i zwyciężaj" 13
2.2.5. Algorytm quickhull 14
2.2.6. Algorytm łańcucha monotonicznego 15
2.2.7. Algorytmy czułe na wyjście 15
2.2.8. Heurystyka Akla-Toussainta 15
2.3. Średnica zbioru punktów 15
2.4. Przecinanie się zbioru odcinków 16
3. Implementacja 18
3.1. Założenia implementacyjne 18
3.2. Podstawowe obiekty geometryczne 18
3.3. Przykładowe obliczenia 19
4. Algorytmy 24
4.1. Wyznaczanie orientacji trzech punktów 24
4.2. Sortowanie punktów względem kąta 24
4.3. Należenie punktu do odcinka 25
4.4. Należenie punktu do trójkąta 26
4.5. Należenie punktu do wielokąta 26
4.6. Należenie punktu do wielokąta wypukłego 29
4.7. Przecinanie się dwóch odcinków 29
4.8. Orientacja wielokąta prostego 30
4.9. Rozpoznawanie wielokątów prostych 31
4.10. Rozpoznawanie wielokątów wypukłych 31
4.11. Kontenery ograniczające 31
4.12. Algorytm Grahama 32
4.13. Algorytm Jarvisa 33
4.14. Algorytm quickhull 33
4.15. Generowanie par punktów antypodycznych 35
4.16. Wyznaczenie pary najdalszych punktów 36

5. Podsumowanie 37
A. Testy algorytmów 38
A.1. Testy wielokątów 38
A.1.1. Test wyznaczania orientacji wielokątów 38
A.1.2. Test rozpoznawania wielokątów prostych 38
A.1.3. Test rozpoznawania wielokątów wypukłych 38
A.2. Testy otoczki wypukłej 38
Bibliografia 45

Spis tabel

3.1. Interfejs punktów na płaszczyźnie. 20
3.2. Interfejs odcinków na płaszczyźnie. 20
3.3. Interfejs wielokątów. 21
3.4. Interfejs prostokątów. 21
3.5. Interfejs trójkątów. 22
3.6. Interfejs okręgów. 22

Spis rysunków

2.1. Wielokąt prosty wklęsły. 11
2.2. Wielokąt złożony. 12

A.1. Wykres wydajności wyznaczania orientacji. 39
A.2. Wydajność testu czy wielokąt jest prosty. 39
A.3. Wykres wydajności sprawdzania wypukłości. 41
A.4. Wykres wydajności algorytmu Grahama (kwadrat). 41
A.5. Wykres wydajności algorytmu Grahama (okrąg). 42
A.6. Wykres wydajności algorytmu Jarvisa (kwadrat). 42
A.7. Wykres wydajności algorytmu Jarvisa (okrąg). 43
A.8. Wykres wydajności algorytmu quickhull (kwadrat). 43
A.9. Wykres wydajności algorytmu quickhull (okrąg). 44

Listings

4.1 Orientacja trzech punktów. 24
4.2 Należenie punktu do odcinka. 25
4.3 Należenie punktu do trójkąta. 26
4.4 Obliczanie liczby przecięć (orientacja). 27
4.5 Obliczanie liczby przecięć (punkt przecięcia). 27
4.6 Obliczanie liczby nawinięć. 28
4.7 Należenie punktu do wielokąta. 28
4.8 Przecinanie się dwóch odcinków. 29
4.9 Orientacja wielokąta prostego. 30
4.10 Prostokąt ograniczający. 31
4.11 Algorytm Grahama z listą cykliczną. 32
4.12 Algorytm Jarvisa z listą cykliczną. 33
4.13 Algorytm *quickhull* z listą cykliczną. 34
4.14 Generator punktów antypodycznych. 35
4.15 Wyznaczanie pary najdalszych punktów. 36

1. Wstęp

Tematem niniejszej pracy są algorytmy geometrii obliczeniowej (ang. *computational geometry*) [1]. Jest to dział algorytmiki, zajmujący się algorytmami i strukturami danych, które pozwalają efektywnie rozwiązywać problemy związane z obiektami geometrycznymi, czyli punktami, odcinkami, wielokątami, okręgami, itp. Geometria obliczeniowa dzieli się na dwie główne gałęzie [1]. *Kombinatoryczna geometria obliczeniowa* zajmuje się obiektami geometrycznymi jako obiektami dyskretnymi. *Numeryczna geometria obliczeniowa* zajmuje się przede wszystkim reprezentowaiem obiektów ze świata rzeczywistego w formie odpowiedniej dla obliczeń komputerowych w systemach CAD/CAM. Niniejsza praca dotyczy kombinatorycznej geometrii obliczeniowej na płaszczyźnie.

W języku polskim ważną pozycją na temat geometrii obliczeniowej jest książka Berga i in. [2]. Opisuje ona algorytmy z tej dziedziny w odniesieniu do zastosowań w robotyce, grafice komputerowej, systemach CAD/CAM i innych. Inną klasyczną pozycją jest książka Preparaty i Shamosa [3]. Książki czy artykuły z geometrii obliczeniowej z reguły prezentują algorytmy w postaci pseudokodu, który pomija szczegóły implementacyjne i pewne przypadki graniczne. Osoba starająca się zaimplementować dany algorytm napotyka na spore trudności, aby otrzymać w pełni poprawny kod, działający również dla nietypowych zestawów danych. Naszym celem będzie próba rozwiązania tego problemu, ale rozpoczniemy od krótkiego przeglądu problemów i technik.

Przykładowe problemy z obszaru geometrii obliczeniowej [1]:

— Wyznaczanie pary najbliższych lub najdalszych punktów.
— Wyznaczenie wszystkich par różnych punktów odległych od siebie nie dalej niż ustalony promień (ang. *fixed-radius near neighbor problem*).
— Wyznaczanie otoczki wypukłej (ang. *convex hull*).
— Wyznaczanie wszystkich przecięć zbioru odcinków (ang. *intersections of segments*), wykrywanie kolizji (ang. *collision detection*).
— Sprawdzanie należenia punktu do wielokąta (ang. *point in polygon problem*).
— Triangulacja wielokąta (ang. *polygon triangulation*).
— Generowanie siatki (ang. *mesh/grid generation*).
— Planowanie ruchu robota (ang. *motion planning*).
— Okienkowanie (ang. *windowing*).

Przykładowe techniki i metodologie tworzenia algorytmów geometrycznych [3]:

— Dziel i zwyciężaj (np. w *quickhull*).
— Rekurencja.
— Programowanie dynamiczne.

— Wymiatanie płaszczyzny (ang. *sweep line algorithms*).
— Obracające się suwmiarki (ang. *rotating calipers*).
— Diagram Woronoja (ang. *Voronoi diagram*).

Przykładowe struktury danych używane w algorytmach geometrii obliczeniowej [3]:

— Zbiór, słownik.
— Stos, kolejka, kolejka priorytetowa.
— Drzewo przedziałów.
— Lista podwójnie powiązana (np. lista krawędzi dla grafów płaskich, lista punktów dla otoczki wypukłej).

Typowe trudności pojawiające się przy rozwiązywaniu problemów z geometrii obliczeniowej:

— Degeneracja, np. punkty współliniowe (przy otoczce wypukłej), punkty leżące na jednym okręgu (triangulacja Delaunaya). Dla zbioru punktów mówi się o *pozycji ogólnej* (ang. *general position*), kiedy nie ma degeneracji [6]. Przy konstrukcji i analizie algorytmów często najpierw analizuje się obiekty w pozycji ogólnej, a potem rozważa się przypadki zdegenerowane.
— Błędy wynikające z ograniczeń liczb zmiennoprzecinkowych, używanych w obliczeniach komputerowych. Inny problem to możliwe przekroczenie zakresu liczb całkowitych. Błędy mogą dotyczyć samych danych wejściowych lub późniejszych obliczeń komputerowych. Rozwiązaniem może być (1) prowadzenie obliczeń dokładnych w zbiorze liczb wymiernych, (2) staranna analiza propagacji błędów operacji zmiennoprzecinkowych, (3) małe zaburzanie danych wejściowych, które będzie gwarantować uzyskanie sensownego rozwiązania przy obliczeniach zmiennoprzecinkowych. Biblioteki geometryczne często mają własne typy liczbowe, aby rozwiązać problem dokładności obliczeń.

Analizując algorytmy w ramach niniejszej pracy korzystaliśmy z dokumentacji kilku istniejących bibliotek geometrycznych, oraz z informacji z serwisów internetowych:

— Biblioteka CGAL, napisana w języku C++, korzysta z bibliotek Boost, pakiety w Debianie to libcgal-demo, libcgal-dev, libcgal9, libcgal-ipelets [7].
— Biblioteka LEDA, napisana w języku C++, dostarcza wydajne typy danych i algorytmy, dostępna w wersji *Free Edition* [8].
— *GeeksforGeeks*, portal z kolekcją algorytmów, struktur danych i problemów, z przykładowymi implementacjami w kilku językach programowania (Python, C/C++, Java) [9].
— *GeomAlgorithms.com*, serwis poświęcony praktycznym algorytmom geometrycznym, z przykładowymi implementacjami w C++ [10].
— *Algorytmy i struktury danych*, serwis z opisami i implementacjami algorytmów w różnych językach programowania [11]

1.1. Cele pracy

Celem pracy jest przedstawienie wybranych algorytmów geometrii obliczeniowej na płaszczyźnie. Obok opisu teoretycznego pokazana będzie implementacja algorytmów w języku Python, ponieważ ten język pozwala na czytelne przedstawienie algorytmów zamiast pseudokodu, a z drugiej strony pozwala na uruchomienie kodu i analizę algorytmu w działaniu. Taka prezentacja algorytmów ma duże zalety dydaktyczne, szczególnie przy starannym zwróceniu uwagi na możliwe źródła błędów.

Mamy nadzieję na przygotowanie bazy, która rozwinie się w większą bibliotekę algorytmów geometrycznych. Chcemy zidentyfikować podstawowe struktury danych i operacje często wykorzystywane przy budowie algorytmów geometrycznych. Spójna implementacja algorytmów z pewnością pomoże w lepszym zrozumieniu znanych algorymów, a być może przyczyni się do znalezienia nowych rozwiązań.

1.2. Organizacja pracy

Praca została zorganizowana w następujący sposób. Rozdział 1 zawiera wprowadzenie do geometrii obliczeniowej, cele, oraz organizację pracy. W rozdziale 2 zebrano potrzebne definicje obiektów geometrycznych i omówiono wybrane algorytmy geometryczne. Rozdział 3 opisuje założenia implementacyjne i interfejs obiektów geometrycznych. W rozdziale 4 omówiono implementacje algorytmów rozważanych w pracy. Rozdział 5 jest podsumowaniem pracy. W dodatku A zostały przedstawione wyniki testów dla zaimplementowanych algorytmów.

2. Geometria obliczeniowa

W tym rozdziale zostaną przypomniane podstawowe definicje i twierdzenia geometrii obliczeniowej, ktore będą wykorzystywane w niniejszej pracy.

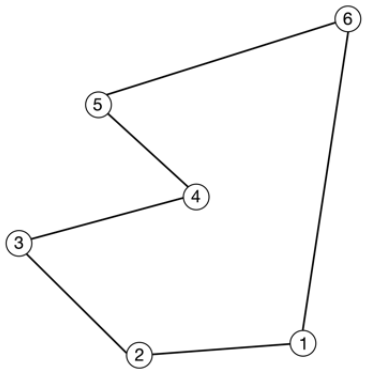
2.1. Wielokąty

Definicja: *Wielokąt/wielobok* (ang. *polygon*) jest to figura płaska ograniczona łamaną zamkniętą, czyli skończonym ciągiem połączonych końcami odcinków [12]. Koniec jednego odcinka jest początkiem następnego odcinka. Czasem zakłada się, że żadne dwa następujące po sobie odcinki nie leżą na jednej prostej. Odcinki nazywamy bokami lub krawędziami wielokąta, a końce odcinków wierzchołkami wielokąta. *Wielokąt prosty* (ang. *simple polygon*) nie ma przecinających się boków. Wielokąt, który nie jest prosty, nazywamy *wielokątem złożonym*.

Definicja: *Wielokąt wypukły* (ang. *convex polygon*) jest to wielokąt prosty, w którym dowolne dwa punkty można połączyć odcinkiem zawartym w całości w tym wielokącie [13]. Przykłady wielokątów wypukłych: trójkąt, prostokąt, kwadrat, romb, równoległobok, trapez. Wielokąt, który nie jest wypukły, nazywamy *wklęsłym* (ang. *concave*). Część wspólna wielokątów wypukłych jest wielokątem wypukłym. Zauważmy, że w definicji wielokąta wypukłego jest mowa o wielokącie prostym, prawdopodobnie ze względu na możliwe patologie.

Twierdzenie: Wielokąt jest wypukły wtedy i tylko wtedy, gdy jest prosty i nie ma kątów wewnętrznych rozwartych. Sprawdzenie rozwartości kątów można wykonać w czasie $O(n)$ przez obliczenie orientacji dla wszystkich trzech kolejnych wierzchołków wielokąta prostego.

Twierdzenie: Wielokąt jest wypukły, jeżeli w standardowej postaci kąty krawędzi są niemalejące. W postaci standardowej wierzchołki wielokąta są podawane zaczynając od tego o najmniejszej współrzędnej y, a przy niejednoznaczności tego o najmniejszej współrzędnej x. Wtedy krawędzie można interpretować jako wektory skierowane zgodnie z kolejnością wierzchołków. Rozważamy kąty między dodatnim kierunkiem osi X, a wektorami krawędzi. W przypadku wielokąta wypukłego kąty te rosną w przedziale od 0 do 2π, co można sprawdzić w czasie $O(n)$. Zauważmy, że nie musimy obliczać samych kątów, wystarczy używać pewnej monotonicznej funkcji kątów.



Rysunek 2.1. Wielokąt prosty wklęsły. Orientacja przeciwna do ruchu wskazówek zegara (wartość -1).

Twierdzenie: Każdy wielokąt wypukły o n wierzchołkach można striangulować w czasie liniowym $O(n)$ poprzez triangulację wachlarzową (ang. *fan triangulation*). Z dowolnego wierzchołka należy narysować odcinki łączące go z pozostałymi (niesąsiednimi) wierzchołkami wielokąta. Jest istotnie, aby dwa wierzchołki sąsiednie wybranego wierzchołka nie miały kątów wewnętrznych równych π, bo wtedy trójkąty byłyby zdegenerowane.

Twierdzenie Krejna-Milmana (1940): Wielokąt wypukły jest otoczką wypukłą swoich wierzchołków [13].

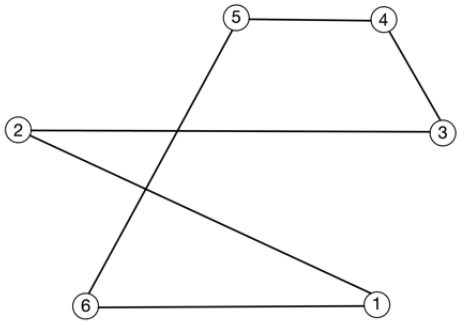
Definicja: Wielokąt jest *monotoniczny* względem prostej L (ang. *monotone polygon with respect to L*), jeżeli każda prosta prostopadła do L przecina brzeg wielokąta najwyżej dwa razy [14]. Wielokąty wypukłe są monotoniczne względem dowolnej prostej. Podział wielokąta prostego na wielokąty monotoniczne może być wykonany w czasie $O(n \log n)$.

Twierdzenie: Każdy wielokąt monotoniczny o n wierzchołkach można striangulować w czasie liniowym $O(n)$.

Twierdzenie (Chazelle, 1991): Każdy wielokąt prosty o n wierzchołkach można striangulować w czasie liniowym $O(n)$ [15]. Algorytm jest trudny. Triangulację można potraktować jako podział na wielokąty monotoniczne.

2.2. Otoczka wypukła

Definicja: *Otoczka wypukła* $\mathrm{CH}(P)$ (ang. *convex hull*) zbioru punktów P w przestrzeni euklidesowej jest to najmniejszy w sensie inkluzji zbiór wypukły zawierający zbiór P [16].



Rysunek 2.2. Wielokąt złożony. Orientacja zgodna z ruchem wskazówek zegara (wartość +1).

Stwierdzenie: Dla dowolnego skończonego zbioru punktów płaszczyzny P, gdzie $|P| > 2$, otoczka wypukła jest wielokątem wypukłym o wierzchołkach należących do zbioru P. W przypadku zdegenerowanym otoczka wypukła będzie odcinkiem. W przypadku trzech punktów niewspółliniowych otoczka wypukła będzie trójkątem o wierzchołkach w zbiorze P.

Jest znanych wiele algorytmów wyznaczających otoczkę wypukłą, przedstawimy kilka z nich [17]. W ogólnym przypadku otoczka wypukła dla zbioru n punktów nie może być wyznaczona szybciej niż sortowanie n liczb, czyli w czasie $O(n \log n)$. W szczególnych przypadkach sortowanie może być wykonane szybciej, co przekłada się na szybsze wyznaczanie otoczki wypukłej.

2.2.1. Algorytm Grahama

Algorytm został opublikowany przez Ronalda Grahama w roku 1972 [18]. Algorytm wyznacza otoczkę wypukłą $\mathrm{CH}(P)$ dla skończonego zbioru n punktów P na płaszczyźnie. Jest to *algorytm inkrementacyjny (przyrostowy)*, kolejne punkty są dodawane pojedyńczo do tymczasowego rozwiązania. Oryginalny algorytm ma pięć kroków [19].

Krok 1: Znaleźć punkt A we wnętrzu $\mathrm{CH}(P)$. Może to być *centroid*, czyli środek masy punktów. [W pewnych implementacjach za punkt A przyjmowany jest punkt o najmniejszej współrzędnej y, a przy kilku takich punktach ten z najmniejszą współrzędną x. Wtedy kąt biegunowy punktów należy do przedziału od 0 do 180 stopni.]

Krok 2: Przenieść środek układu współrzędnych do punktu A.

Krok 3: Posortować wszystkie punkty wg kąta we współrzędnych biegunowych.

Krok 4: Jeżeli dwa punkty mają ten sam kąt biegunowy, to należy usunąć punkt bliższy A.

Krok 5: Startujemy z trzech kolejnych punktów $B_k B_{k+1} B_{k+2}$ o różnych kątach biegunowych. Jeżeli B_{k+1} należy do trójkąta AB_kB_{k+2}, to usuwamy B_{k+1}

(B_{k+1} nie należy do otoczki), a następnie powtarzamy krok 5 dla punktów $B_{k-1}B_kB_{k+2}$. Jeżeli B_{k+1} **nie** należy do trójkąta AB_kB_{k+2}, to powtarzamy krok 5 dla punktów $B_{k+1}B_{k+2}B_{k+3}$ (B_{k+1} może należeć do otoczki).

Zauważmy, że każde zastosowanie kroku 5 albo zmniejsza liczbą możliwych punktów w tymczasowej otoczce, albo zwiększa liczbę kandydatów możliwych w otoczce. Po zastosowaniu $2n$ razy kroku 5 na pewno uzyskamy prawidłową otoczkę wypukłą. Złożoność obliczeniowa czasowa algorytmu Grahama wynosi $O(n \log n)$, co jest konsekwencją sortowania.

Algorytm Grahama dla wielokątów prostych: Jeżeli wiemy, że ciąg punktów P tworzy wielokąt prosty, to w algorytmie Grahama można pominąć etap sortowania punktów. W efekcie otrzymamy algorytm o złożoności liniowej $O(n)$ znajdujący otoczkę wypukłą.

Problem tworzenia wielokąta prostego: Rozważmy problem budowy wielokąta prostego przy zadanym zbiorze punktów. Można wykorzystać pierwsze trzy kroki algorytmu Grahama do rozwiązania tego problemu w czasie $O(n \log n)$. Po posortowaniu punktów względem kąta w układzie związanym z centroidem wystarczy połączyć kolejne punkty. Punkty z tym samym kątem łączymy w kolejności od najbliższego centroidu do najdalszego.

2.2.2. Algorytm Jarvisa

Algorytm Jarvisa nazywany jest również algorytmem owijania prezentów (ang. *gift wrapping algorithm*) [20]. Złożoność obliczeniowa algorytmu wynosi $O(hn)$, gdzie n to liczba punktów w zbiorze P, natomiast h to liczba punktów należących do otoczki wypukłej $\mathrm{CH}(P)$ (Chand, Kapur, 1970; Jarvis, 1973).

Krok 1: Znaleźć punkt A_1 w zbiorze P o najmniejszej współrzędnej y, a jeżeli takich punktów jest więcej, to wybrać ten o najmniejszej współrzędnej x. Punkt A_1 zaliczyć do otoczki wypukłej. [Czasem wybiera się punkt o najmniejszej współrzędnej x].

Krok 2: Powtarzać następujące kroki, aż dojdziemy do punktu A_1. (a) Wybrać punkt A_{i+1} taki, że dla każdego punktu B punkty (A_i, A_{i+1}, B) dają skręt w lewo. (b) Zaliczyć punkt A_{i+1} do otoczki wypukłej.

Powyższy opis nie uwzględnia degeneracji (punkty współliniowe). Pętla w drugim kroku wykona się h razy, a wybór każdego kolejnego punktu A_i należącego do otoczki zajmuje czas $O(n)$. Warto zauważyć, że algorytm Jarvisa przypomina sortowanie przez wybór, bo kolejno szukamy następnego punktu otoczki wypukłej.

2.2.3. Algorytm inkrementacyjny

Idea algorytmu inkrementacyjnego jest opisana w pracy [30] (Kallay, 1984). Złożoność obliczeniowa czasowa algorytmu wynosi $O(n \log n)$.

2.2.4. Algorytm "dziel i zwyciężaj"

Algorytm "dziel i zwyciężaj" ma strukturę rekurencyjną [9]. Załóżmy, że znamy otoczkę wypukłą $\mathrm{CH}(P_1)$ dla lewej połowy punktów P_1, oraz otoczkę

wypukłą CH(P_2) dla prawej połowy punktów P_2. Należy połączyć obie otoczki wypukłe w jedną otoczkę wypukłą CH($P_1 \cup P_2$). Można to zrobić znajdując dolną i górną styczną do lewej i prawej otoczki. Wypadkowa otoczka będzie zawierała fragmenty lewej i prawej otoczki, oraz odcinki dolnej i górnej stycznej.

Pozostaje problem znalezienia otoczki dla lewej i prawej połowy punktów. W tym miejscu wykorzystujemy rekurencję i dzielimy zbiory punktów na mniejsze podzbiory. Dla małych zbiorów, np. pięcioelementowych, można znaleźć otoczkę wypukłą algorytmem siłowym. Złożoność obliczeniowa czasowa algorytmu wynosi $O(n \log n)$ (Preparata, Hong, 1977).

Osobnym problemem do rozwiązania jest znalezienie stycznej pomiędzy dwoma rozłącznymi wypukłymi wielokątami [9]. Możliwe są cztery takie styczne: T_{uu}, T_{ud}, T_{du}, T_{dd}. W algorytmie "dziel i zwyciężaj" korzystamy z T_{uu} i T_{dd}.

2.2.5. Algorytm quickhull

Algorytm *quickhull* również stosuje metodę "dziel i zwyciężaj", ale jego działanie przypomina algorytm sortowania *quicksort* [21], [9]. Wyznaczanie rozwiązania odbywa się rekurencyjnie - w każdym kroku rozpatrywany zbiór punktów dzielony jest na podzbiory, po czym dla każdego podzbioru wyznaczane są punkty należące do otoczki. Średnia złożoność obliczeniowa wynosi $O(n \log n)$, natomiast w przypadku pesymistycznym wynosi $O(n^2)$ (Eddy, 1977; Bykat, 1978; Green, Silverman, 1979).

Opisując działanie algorytmu w sposób obrazowy - za zbiór punktów weźmy posadzone kwiaty w ogrodzie. Mają one zostać otoczone niskim płotkiem, który zabezpieczałby je przed gryzoniami w taki sposób, by zajmował on możliwie jak najmniejszą część ogrodu. Wybieramy dwie rośliny, które rosną najbardziej na lewo (A) i najbardziej na prawo (B), a następnie "łączymy" je wzrokiem. Rośliny A i B zaliczamy do otoczki.

Zbiór roślin został podzielony na dwa podzbiory, na prawo od odcinka AB i na lewo od odcinka AB (czyli na prawo od odcinka BA). Każdy podzbiór rozpatrujemy osobno. Wyszukujemy kwiat C, który leży w największej odległości na prawo od odcinka AB. Kwiat C zaliczamy do otoczki. Rośliny ABC tworzą trójkąt, w którego środku rosnące kwiaty na pewno nie należą do otoczki. Dalej należy rozważyć kwiaty leżące na prawo od odcinka AC i kwiaty leżące na prawo od odcinka CA. Czynności powtarzamy rekurencyjnie dla mniejszych podzbiorów, aż na prawo od badanego odcinka nie będzie już kwiatów. Całość można opisać pseudokodem.

```
def find_convex_hull(Q):
    A = skrajny lewy punkt
    B = skrajny prawy punkt
    Q1 = zbior punktow po prawej stronie AB
    Q2 = zbior punktow po lewej stronie AB
    return [A] + quickhull(A, B, Q1) + [B] + quickhull(B, A, Q2)

def quickhull(A, B, Q):
    if Q pusty: return []
    C = najbardziej odlegly punkt od AB
    Q1 = zbior punktow po prawej stronie AC
```

```
Q2 = zbior punktow po prawej stronie CB
return quickhull(A, C, Q1) + [C] + quickhull(C, B, Q2)
```

2.2.6. Algorytm łańcucha monotonicznego

Algorytm łańcucha monotonicznego (ang. *monotone chain algorithm*) bazuje na leksykograficznym sortowaniu punktów (Andrew, 1979). Złożoność obliczeniowa wynosi $O(n \log n)$. Pseudokod algorytmu i implementację w C++ można znaleźć w serwisie *GeomAlgorithms.com* [10].

W skrócie można powiedzieć, że algorytm wyznacza górny i dolny łańcuch punktów należących do otoczki wypukłej, rozciągający się od punktu położonego najbardziej na lewo do punktu położonego najbardziej na prawo.

2.2.7. Algorytmy czułe na wyjście

Algorytmy czułe na wyjście (ang. *output-sensitive algorithms*) mają czas działania zależny nie tylko od wielkości danych wejściowych, ale również od wielkości danych wyjściowych. W przypadku problemu wyznaczania otoczki wypukłej rozmiar danych wejściowych to liczba punktów n w zbiorze P, a rozmiar danych wyjściowych to liczba punktów h należących do otoczki wypukłej $\mathrm{CH}(P)$.

Najlepsze algorytmy czułe na wyjście mają złożoność $O(n \log h)$. Jest to algorytm Chana (1996) [22], [23], algorytm Kirkpatricka-Seidela (1986) [24]. Algorytm Jarvisa również należy do algorytmów czułych na wyjście, ale ma gorszą złożoność obliczeniową $O(nh)$.

2.2.8. Heurystyka Akla-Toussainta

Algorytmy wyznaczania otoczki wypukłej mogą być przyspieszone przez szybkie usunięcie punktów, które nie będą należały do otoczki [17]. Heurystyka Akla-Toussainta (1978) znajduje dwa punkty z najmniejszą i największą wartością współrzędnej x, oraz dwa punkty z najmniejszą i największą wartością współrzędnej y [zajmuje to czas $O(n)$]. Te cztery punkty tworzą czworokąt wypukły, a wszystkie punkty w nim zawarte (oprócz wierzchołków) nie są częścią otoczki. Znalezienie takich punktów zajmuje czas $O(n)$.

Opcjonalnie można znaleźć dwa punkty z najmniejszą i największą sumą współrzędnych $x + y$, oraz dwa punkty z najmniejszą i największą różnicą współrzędnych $x - y$. Dostaniemy wtedy wypukły ośmiokąt, którego punkty mogą być bezpiecznie usunięte (oprócz wierzchołków). Taki preprocessing może prowadzić nawet do algorytmów działających w czasie liniowym, mimo że w pesymistycznym przypadku czas jest kwadratowy.

2.3. Średnica zbioru punktów

Definicja: *Średnica* (ang. *diameter*) zbioru punktów P jest to największa odległość między dwoma punktami należącymi do zbioru P. Średnica jest miarą rozrzucenia punktów, a wykorzystuje się ją m.in. przy klastrowaniu

punktów [3]. Najprostszy sposób znalezienia średnicy zbioru n punktów polega na sprawdzeniu odległości pomiędzy każdą parą punktów. Prowadzi to do algorytmu działającego w czasie $O(n^2)$. Istnieje lepsze rozwiązanie.

Twierdzenie (Hocking, Young, 1961): Średnica zbioru P jest równa średnicy jego otoczki wypukłej CH(P) [3]. Algorytm Grahama pozwala wyznaczyć otoczkę wypukłą zbioru punktów w czasie $O(n \log n)$. Otoczka wypukła jest wielokątem wypukłym, co jest zasadniczym uproszczeniem problemu.

Twierdzenie (Yaglom, Bolyanskii, 1961): Średnica figury wypukłej jest największą odległością między parą równoległych prostych wspierających [3]. Proste wspierające dotykają figurę wypukłą (wielokąt wypukły) z obu stron. Ważne jest, że proste wspierające nie mogą przechodzić przez każdą parę punktów, a jedynie przez *punkty antypodyczne*. Technika obracających się suwmiarek (ang. *rotating calipers*) pozwala wyznaczyć wszystkie pary punktów antypodycznych w czasie $O(n)$, ponieważ unika się sprawdzania wszystkich par punktów [25], [26], [27], [28]. Prowadzi to do następującego wyniku końcowego:

Twierdzenie: Średnica wielokąta wypukłego o n bokach może być wyznaczona w czasie $O(n)$. Dla dowolnego zbioru n punktów optymalny algorytm wyznacza średnicę w czasie $O(n \log n)$.

Twierdzenie: Średnica wielokąta prostego o n bokach może być wyznaczona w czasie $O(n)$. Wykorzystuje się wersję algorytmu Grahama wyznaczającą otoczkę wypukłą wielokąta prostego w czasie $O(n)$.

2.4. Przecinanie się zbioru odcinków

W wielu sytuacjach zachodzi potrzeba sprawdzenia, czy w zbiorze odcinków znajdują się odcinki przecinające się. Przykładem może być testowanie, czy wielokąt jest prosty, kiedy sprawdza się przecinanie par niesąsiednich krawędzi. Już jedno przecięcie krawędzi powoduje, że wielokąt nie jest prosty. Drugim przykładem jest wyznaczanie części wspólnej lub sumy dwóch wielokątów prostych lub grafów planarnych. W tej sytuacji należy wyznaczyć wszystkie punkty przecięcia krawędzi.

W ogólnym przypadku dla n odcinków może być $O(n^2)$ punktów przecięcia. Najprostszy algorytm siłowy polega na sprawdzaniu wszystkich par krawędzi, co daje złożoność $O(n^2)$. Jeżeli jednak liczba punktów przecięcia k jest o wiele mniejsza od n^2, to można szukać lepszej metody rozwiązania problemu przecięć. Wspomnimy dwa algorytmy wykorzystujące technikę wymiatania płaszczyzny.

Algorytm Shamosa-Hoeya (1976): Algorytm znajduje co najmniej jedno istniejące przecięcie w zbiorze odcinków w czasie $O(n \log n)$, korzystając z pamięci $O(n)$ [10].

Algorytm Bentleya-Ottmana (1979): Algorytm wyznacza wszystkie k przecięcia odcinków w czasie $O((n+k)\log n)$, korzystając z pamięci $O(n+k)$ [29]. Jest to algorytm czuły na wyjście, przez co dla $k = n^2$ zajmuje czas $O(n^2 \log n)$, co jest wynikiem gorszym od algorytmu siłowego.

3. Implementacja

Rozdział poświęcony jest przedstawieniu założeń naszej implementacji, interfejsu obiektów geometrycznych, oraz przykładowym obliczeniom.

3.1. Założenia implementacyjne

Znane są ograniczenia związane z przechowywaniem liczb w maszynie cyfrowej, oraz z wykonywaniem działań matematycznych. Naszym celem jest unikanie występowania tego rodzaju błędów. Python automatycznie korzysta z liczb całkowitych dowolnej wielkości, więc nie ma zagrożenia przekroczenia zakresu. Obliczenia będziemy prowadzić przy wykorzystaniu liczb wymiernych z klasy Fraction, aby przy danych wejściowych całkowitych lub wymiernych, wyniki obliczeń też były liczbami wymiernymi. Jeżeli dane wejściowe będą typu **float**, wtedy obliczenia automatycznie będą dawały wyniki typu **float** (tutaj będzie trudniej kontrolować dokładność obliczeń).

3.2. Podstawowe obiekty geometryczne

Przedstawimy podstawowe obiekty geometryczne na płaszczyźnie i sposób ich implementacji. Implementacja jest rozszerzeniem kodu wykorzystywanego podczas kursu *Język Python* na Wydziale FAIS UJ. Wszystkie obiekty geometryczne są niezmienne i hashowalne.

Punkt: Instancja klasy Point (moduł points). Punkt jest utożsamiany z wektorem łączącym początek układu współrzędnych z danym punktem na płaszczyźnie.

Odcinek: Instancja klasy Segment (moduł segments). Odcinek ma wyróżniony początek i koniec (atrybuty pt1 i pt2), czyli jest to odcinek skierowany.

Wielokąt: Instancja klasy Polygon (moduł polygons). Wewnętrznie wielokąt przechowuje listę minimum trzech punktów w atrybucie point_list. Punkty są uporządkowane wzdłuż brzegu wielokąta przeciwnie do ruchu wskazówek zegara (lub zgodnie z ruchem). Orientację wielokąta można wyznaczyć za pomocą osobnej metody Polygon.orientation().

Prostokąt: Instancja klasy Rectangle (moduł rectangles). Boki prostokąta są równoległe do osi układu współrzędnych. Wewnętrznie prostokąt przechowuje lewy dolny i prawy górny róg w atrybutach pt1 i pt2.

Trójkąt: Instancja klasy Triangle (moduł triangles). Wewnętrznie trójkąt przechowuje trzy niewspółliniowe punkty w atrybutach pt1, pt2, pt3. Orientację trójkąta można wyznaczyć za pomocą osobnej metody Triangle.orientation().

Okrąg: Instancja klasy Circle (moduł circles). Wewnętrznie okrąg przechowuje swój środek i promień w atrybutach pt i radius.

Zauważmy, że dla wielu figur polecenie s.move(0, 0) jest równoważne poleceniu s.copy(), ponieważ dostajemy nową instancję klasy.

Warto zatrzymać się nad sposobem przechowywania wielokąta, który jest wyznaczony przez uporządkowany ciąg kolejnych wierzchołków. W naszej implementacji wielokąt wewnętrznie przechowuje listę (tablicę) instancji klasy Point. W literaturze można spotkać rozwiązania nieobiektowe z dwiema tablicami zawierającymi osobno współrzędne x i y. Jednak w pewnych algorytmach zachodzi potrzeba dodawania lub usuwania wierzchołków. Wtedy lepszą strukturą danych do przechowywania wierzchołków jest lista cykliczna podwójnie powiązana. Jeden węzeł listy może przechowywać instancję klasy Point lub parę liczb (x, y). Transformację między tablicą a listą cykliczną można wykonać w czasie $O(n)$, a więc nie jest to problem przy analizie złożoności algorytmów.

Podane sposoby przechowywania wielokąta mają dwie niejednoznaczności, które pozwalają reprezentować wielokąt na $2n$ sposobów. Po pierwsze mamy dowolność wyboru pierwszego wierzchołka wielokąta. Po drugie są dwie orientacje z jakimi można przebiegać wierzchołki. W literaturze (Shamos, 1978) określa się postać kanoniczną wielokąta, gdzie wierzchołki są listowane z orientacją przeciwną do ruchu wskazówek zegara, pierwszy wierzchołek ma najmniejszą współrzędną y, a przy niejednoznaczności wybieramy wierzchołek z najmniejszą współrzędną x. W naszej implementacji nie jest wymagane użycie postaci kanonicznej, a orientację czy lewy dolny wierzchołek można wyznaczyć w razie potrzeby.

Algorytmy wyznaczające otoczkę wypukłą dla danego zbioru punktów przygotowano w dwóch wersjach. W pierwszej wersji otoczka ma postać zwykłej listy punktów, a w drugiej wersji otoczka jest listą cykliczną przechowującą punkty. Struktura listy cyklicznej podwójnie powiązanej również została przygotowana i przetestowana w ramach niniejszej pracy.

3.3. Przykładowe obliczenia

Przedstawimy przykładowe obliczenia wykonane za pomocą stworzonego oprogramowania.

Przykład 1: Wyznaczanie otoczki wypukłej dla zbioru punktów.

```
# Przygotowanie zestawu 50 punktow z kwadratu o boku 1.
>>> from points import Point
>>> from random import random
>>> plist = [Point(random(), random()) for _ in range(50)]

# Sortowanie punktow wzgledem kata nachylenia.
>>> plist.sort(key=Point.alpha)
```

Tabela 3.1. Interfejs punktów na płaszczyźnie (klasa Point). p i q są punktami, a to liczba.

Operacja	**Znaczenie**	**Metoda**
p = Point(x, y)	tworzenie punktu/wektora	__init__
print p	wyświetlanie punktu	__repr__
cmp(p, q)	porównywanie punktów	__cmp__
p + q	dodawanie wektorów	__add__
p − q	odejmowanie wektorów	__sub__
p ∗ q, p ∗ a	mnożenie wektorów	__mul__
a ∗ p	mnożenie wektorów	__rmul__
+p	operator jednoargumentowy	__pos__
−p	operator jednoargumentowy	__neg__
p.cross(q)	mnożenie wektorowe	cross
p.copy()	kopiowanie punktu	copy
abs(p)	długość wektora	__abs__
p.length()	długość wektora	length
p.alpha()	funkcja kąta nachylenia	alpha
hash(p)	hashowanie punktu	__hash__
p.gnu()	polecenie gnuplota	gnu

Tabela 3.2. Interfejs odcinków na płaszczyźnie(klasa Segment). s, t są odcinkami, p i q są punktami.

Operacja	**Znaczenie**	**Metoda**
s = Segment(x1, y1, x2, y2)	tworzenie odcinka	__init__
t = Segment(p, q)	tworzenie odcinka	__init__
print s	wyświetlanie odcinka	__repr__
s == t	porównywanie odcinków	__eq__
s != t	porównywanie odcinków	__ne__
s.center()	środek odcinka	center
s.length()	długość odcinka	length
p **in** s, q **not in** s	punkt w odcinku	__contains__
s. intersect (t)	przecinanie odcinków	__intersect__
s.move(x, y)	odcinek przesunięty	move
s.move(p)	odcinek przesunięty	move
s.copy()	kopiowanie odcinka	copy
hash(s)	hashowanie prostokąta	__hash__
s.gnu()	polecenie gnuplota	gnu

Tabela 3.3. Interfejs wielokątów (klasa Polygon). s, t są wielokątami, p, q, r są punktami. Ustawienie parametru na wartość test_is_simple=False pozwala pominąć powolny test wielokąta prostego.

Operacja	**Znaczenie**	**Metoda**
s = Polygon(x1, y1, x2, y2, x3, y3)	tworzenie wielokąta	__init__
t = Polygon(p, q, r)	tworzenie wielokąta	__init__
print s	wyświetlanie wielokąta	__repr__
s == t	porównywanie wielokątów	__eq__
s != t	porównywanie wielokątów	__ne__
s.move(x, y)	wielokąt przesunięty	move
s.move(p)	wielokąt przesunięty	move
s.copy()	kopiowanie wielokąta	copy
s.is_simple()	czy wielokąt prosty	is_simple
s.is_convex(test_is_simple=True)	czy wielokąt wypukły	is_convex
s.orientation(test_is_simple=True)	orientacja wielokąta prostego	orientation
p **in** s, q **not in** s	punkt w wielokącie	__contains__

Tabela 3.4. Interfejs prostokątów (klasa Rectangle). s, t są prostokątami, p i q są punktami.

Operacja	**Znaczenie**	**Metoda**
s = Rectangle(x1, y1, x2, y2)	tworzenie prostokąta	__init__
t = Rectangle(p, q)	tworzenie prostokąta	__init__
print s	wyświetlanie prostokąta	__repr__
s == t	porównywanie prostokątów	__eq__
s != t	porównywanie prostokątów	__ne__
s.center()	środek prostokąta	center
s.area()	pole powierzchni	area
s.make4()	podział prostokąta	make4
p **in** s, q **not in** s	punkt w prostokącie	__contains__
s.cover(t)	prostokąt pokrywający	cover
s.intersection(t)	część wspólna	intersection
s.is_square()	czy kwadrat	is_square
s.move(x, y)	prostokąt przesunięty	move
s.move(p)	prostokąt przesunięty	move
s.copy()	kopiowanie prostokąta	copy
hash(s)	hashowanie prostokąta	__hash__
s.gnu()	polecenie gnuplota	gnu

Tabela 3.5. Interfejs trójkątów (klasa Triangle). s, t są trójkątami, p, q, r są punktami.

Operacja	**Znaczenie**	**Metoda**
s = Triangle(x1, y1, x2, y2, x3, y3)	tworzenie trójkąta	__init__
t = Triangle(p, q, r)	tworzenie trójkąta	__init__
print s	wyświetlanie trójkąta	__repr__
s == t	porównywanie trójkątów	__eq__
s != t	porównywanie trójkątów	__ne__
s.center()	środek trójkąta	center
s.area()	pole powierzchni	area
s.make4()	podział trójkąta	make4
p **in** s, q **not in** s	punkt w trójkącie	__contains__
s.move(x, y)	trójkąt przesunięty	move
s.move(p)	trójkąt przesunięty	move
s.copy()	kopiowanie trójkąta	copy
s.orientation()	orientacja trójkąta	orientation
hash(s)	hashowanie trójkąta	__hash__
s.gnu()	polecenie gnuplota	gnu

Tabela 3.6. Interfejs okręgów (klasa Circle). s, t są okręgami, p, q, r są punktami, a to liczba dodatnia.

Operacja	**Znaczenie**	**Metoda**
s = Circle(x, y, a)	tworzenie okręgu	__init__
t = Circle(p, a)	tworzenie okręgu	__init__
print s	wyświetlanie okręgu	__repr__
s == t	porównywanie okręgów	__eq__
s != t	porównywanie okręgów	__ne__
s.area()	pole powierzchni	area
p **in** s, q **not in** s	punkt w okręgu	__contains__
s.cover(t)	okrąg pokrywający	cover
s.move(x, y)	okrąg przesunięty	move
s.move(p)	okrąg przesunięty	move
s.copy()	kopiowanie okręgu	copy
hash(s)	hashowanie okręgu	__hash__
s.gnu()	polecenie gnuplota	gnu

```
# Sortowanie punktow wzgledem kata nachylenia, potem promienia.
>>> plist.sort(key=lambda p: (p.alpha(), p*p))
# Sortowanie punktow wzgledem y, potem x.
>>> plist.sort(key=lambda p: (p.y, p.x))

# Algorytm Grahama.
>>> from graham import GrahamScan
>>> algorithm = GrahamScan(plist)
>>> algorithm.run()
>>> print algorithm.convex_hull   # lista punktow otoczki
[ ... ]

# Algorytm Jarvisa.
>>> from jarvis import JarvisMarch
>>> algorithm = JarvisMarch(plist)
>>> algorithm.run()
>>> print algorithm.convex_hull   # lista punktow otoczki
[ ... ]

# Algorytm quickhull.
>>> from quickhull import QuickHull
>>> algorithm = QuickHull(plist)
>>> algorithm.run()
>>> print algorithm.convex_hull   # lista punktow otoczki
[ ... ]
```

Przykład 2: Badanie wielokątów.

```
>>> from points import Point
>>> from polygons import Polygon
>>> from polygons import bounding_box
>>> polygon = Polygon(1, 0, 2, 1, 1, 2, 0, 1)   # kwadrat obrocony
>>> polygon
Polygon(Point(1, 0), Point(2, 1), Point(1, 2), Point(0, 1))
>>> polygon.is_simple()
True
>>> polygon.orientation(test_is_simple=False)   # pomijanie testu
1       # orientacja przeciwna do ruchu wskazowek zegara
>>> polygon.is_convex(test_is_simple=False)   # pomijanie testu
True
>>> Point(1, 1) in polygon
True
>>> Point(0, 0) in polygon
False
>>> bounding_box(polygon.point_list)
Polygon(Point(0, 0), Point(2, 0), Point(2, 2), Point(0, 2))
```

4. Algorytmy

Kod stworzony w ramach niniejszej pracy pokazuje typowe struktury danych i techniki stosowane w geometrii obliczeniowej. W tym rozdziale zwrócimy uwagę na najważniejsze z nich.

4.1. Wyznaczanie orientacji trzech punktów

Wyznaczanie orientacji trzech uporządkowanych punktów na płaszczyźnie jest wykorzystywane w wielu algorytmach geometrii obliczeniowej. Użycie liczb zmiennoprzecinkowych w obliczeniach może prowadzić do różnego typu błędów [30]. Orientację wyznacza funkcja orientation(p, q, r), która zwraca 0 dla punktów współliniowych, 1 przy skręcie w lewo, -1 przy skręcie w prawo. Orientacja jest równa znakowi wyznacznika zbudowanego ze współrzędnych punktów,

$$\text{orientation}(p,q,r) = \text{sign}\left(\det\begin{bmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{bmatrix}\right). \tag{4.1}$$

W implementacji wykorzystujemy własności instancji klasy Point (listing 4.1). Dla wymiernych współrzędnych punktów obliczenia są dokładne. Dla współrzędnych typu **float** mogą wystąpić błędy zaokrągleń.

Listing 4.1. Orientacja trzech punktów.

```
def orientation(p, q, r):
    """Orientation of 3 ordered points."""
    result = (q - p).cross(r - p)
    if result == 0:    # points are colinear
        return 0
    elif result > 0:    # left turn (counterclockwise)
        return 1
    else:    # right turn (clockwise)
        return -1
```

W wielu sytuacjach do sprawdzenia orientacji wystarcza prostsza funkcja oriented_area(), która zwraca zorientowane pole równoległoboku zbudowanego na wektorach $q - p$ i $r - p$.

4.2. Sortowanie punktów względem kąta

W algorytmie Grahama pojawia się problem sortowania punktów względem kątów nachylenia ich wektorów wodzących. Przy równych kątach bierze

się pod uwagę odległość punktu od początku układu współrzędnych. Obliczanie wprost kątów wymaga użycia kosztownych funkcji trygonometrycznych, dlatego zwykle stosuje się inne rozwiązania. Oprócz tego obliczanie odległości od początku układu wymaga kosztownego pierwiastkowania, którego również można uniknąć wykorzystując kwadrat odległości.

Pierwszy sposób uniknięcia funkcji trygonometrycznych polega na stosowaniu podczas sortowania funkcji orientation(origin, pt1, pt2), która wyznaczy nam wzajemne uporządkowanie dwóch punktów. Punkt origin oznacza początek układu współrzędnych.

Drugi sposób polega na wykorzystaniu pewnej funkcji monotonicznej względem kąta nachylenia. Jedna z możliwych takich funkcji monotonicznych ma postać ($d = |x| + |y|$) [11]:

$$f(x,y) = \begin{cases} y/d & \text{gdy } x \geqslant 0 \text{ oraz } y \geqslant 0 \text{ (I ćwiartka)}, \\ 2 - y/d & \text{gdy } x < 0 \text{ oraz } y \geqslant 0 \text{ (II ćwiartka)}, \\ 2 + y/d & \text{gdy } x < 0 \text{ oraz } y < 0 \text{ (III ćwiartka)}, \\ 4 - y/d & \text{gdy } x \geqslant 0 \text{ oraz } y < 0 \text{ (IV ćwiartka)}. \end{cases} \tag{4.2}$$

Dla punktu w początku układu współrzędnych można przyjąć $f(x,y) = 0$ lub rzucić wyjątkiem. Funkcja f została zaimplementowana jako metoda Point.angle(). Dla punktów o współrzędnych wymiernych funkcja zwraca wartości wymierne, w innych przypadkach zwraca **float**.

4.3. Należenie punktu do odcinka

Chcemy odpowiedzieć na pytanie, czy dany punkt należy do odcinka. Przyjmujemy, że odcinek jest domknięty, czyli zawiera swoje końce. Najpierw sprawdzamy, czy dany punkt jest współliniowy z odcinkiem, a następnie sprawdzamy zakresy współrzędnych. Algorytm jest zawarty w metodzie Segment.__contains__(), dzięki czemu można z niego korzystać przez test point **in** segment.

Listing 4.2. Należenie punktu do odcinka.

```
class Segment:
    """The class defining a segment."""

    def __contains__(self, other):
        """Test if a point is in a segment."""
        if isinstance(other, Point):
            if orientation(self.pt1, self.pt2, other) != 0:
                return False  # not collinear
            if (other.x <= max(self.pt1.x, self.pt2.x) and
                other.x >= min(self.pt1.x, self.pt2.x) and
                other.y <= max(self.pt1.y, self.pt2.y) and
                other.y >= min(self.pt1.y, self.pt2.y)):
                return True
            else:
                return False
        else:
            raise ValueError("not a point")
```

4.4. Należenie punktu do trójkąta

Chcemy odpowiedzieć na pytanie, czy dany punkt należy do trójkąta. Przyjmujemy, że trójkąt jest domknięty, czyli zawiera swój brzeg. Dla każdego boku trójkąta sprawdzamy, czy dany punkt znajduje się po tej samej stronie tego boku co przeciwległy wierzchołek trójkąta. Korzystamy z funkcji orientation(). Algorytm jest zawarty w metodzie Triangle.__cointains__(), dzięki czemu można z niego korzystać przez test point **in** triangle. Algorytm sugeruje uogólnienie na dowolne wielokąty wypukłe.

Listing 4.3. Należenie punktu do trójkąta.

```
class Triangle:
    """The class defining a triangle."""

    def __contains__(self, other):
        """Test if a point is in a triangle."""
        if isinstance(other, Point):
            a12 = orientation(self.pt1, self.pt2, self.pt3)
            b12 = orientation(self.pt1, self.pt2, other)
            a23 = orientation(self.pt2, self.pt3, self.pt1)
            b23 = orientation(self.pt2, self.pt3, other)
            a31 = orientation(self.pt3, self.pt1, self.pt2)
            b31 = orientation(self.pt3, self.pt1, other)
            return (a12*b12 >= 0) and (a23*b23 >= 0) and (a31*b31 >= 0)
        else:
            raise ValueError("not a point")
```

4.5. Należenie punktu do wielokąta

Interesuje nas odpowiedź na pytanie, czy dany punkt należy do wielokąta, przy czym wielokąt jest dany jako ciąg kolejnych wierzchołków. Jest to problem w geometrii obliczeniowej pod nazwą *punkt w wielokącie* (ang. *point-in-polygon problem*) [31]. Problem ma zastosowanie m.in. w grafice komputerowej, w systemach informacji geograficznej (GIS), systemach CAD. Typowo stosuje się dwie metody o złożoności liniowej $O(n)$ [10]:

— *Metoda liczby przecięć* cn (ang. *crossing number method*). Zlicza się ile razy promień wypuszczony z punktu P przecina krawędzie ograniczające wielokąt. Punkt P jest na zewnątrz wielokąta, jeżeli liczba przecięć jest parzysta, w przeciwnym wypadku punkt P jest wewnątrz wielokąta [32].
— *Metoda liczby nawinięć* wn (ang. *winding number method*). Zlicza się ile razy wielokąt owija się wokół punktu P. Punkt P jest na zewnątrz wielokąta tylko dla wn $= 0$. Dla innych wartości wn punkt P jest wewnątrz wielokąta.

Dla wielokąta prostego obie metody dają taki sam wynik. W przypadku wielokątów złożonych wyniki dla pewnych punktów mogą być różne, np. w obszarach nakładania się wielokąta na siebie. Generalnie zaleca się stosowanie metody liczby nawinięć, ponieważ daje bardziej intuicyjną odpowiedź [10]. W niniejszej pracy przedstawimy obie metody, ponieważ udało się opracować

wydajny kod, który dodatkowo podkreśla podobieństwo obu metod (Sunday, 2001). W obliczeniach korzystamy z funkcji orientation() i dla liczb wymiernych obliczenia są dokładne.

Przy obliczaniu liczby przecięć należy obsłużyć szczególne przypadki, kiedy promień przechodzi przez wierzchołek wielokąta, albo cała krawędź zawiera się w promieniu. Ponadto trzeba ustalić, czy punkt na brzegu wielokąta leży wewnątrz, czy na zewnątrz. Zwykle przyjmuje się konwencję, że punkt na lewej lub dolnej krawędzi jest wewnątrz wielokąta, a punkt na prawej lub górnej krawędzi jest na zewnątrz wielokąta. Dzięki temu, jeżeli dwa wielokąty mają wspólną krawędź, to punkt należy tylko do jednego wielokąta. Ta konwencja pozwala uniknąć wielu problemów w grafice komputerowej [10].

Typowy algorytm obliczający liczbę przecięć wykonuje petlę po krawędziach wielokąta i sprawdza przecięcia. Szczególne przypadki są prawidłowo obsługiwane, jeżeli stosowane są następujące reguły przecięć krawędzi (promień biegnie poziomo w prawo).

1. Dla krawędzi wznoszącej się zaliczamy punkt startowy i pomijamy końcowy.
2. Dla krawędzi opadającej pomijamy punkt startowy i zaliczamy końcowy.
3. Krawędzie równoległe pomijamy.
4. Punkt przecięcia krawędzi z promieniem musi leżeć ściśle na prawo od początku promienia.

Zwykle algorytmy obliczające liczbę przecięć wyznaczają współrzędne punktu przecięcia krawędzi i promienia, ale nie jest to konieczne, co pokazuje nasza implemnetacja, wykorzystująca pomysły z kodu dla liczby nawinięć. Po ustaleniu, czy krawędź przecinająca promień jest wznosząca się, czy opadająca, wystarczy sprawdzić orientację początku promienia względem krawędzi.

Listing 4.4. Obliczanie liczby przecięć (orientacja).

```
def crossing_number(polygon, point):
    """Return the crossing number for a point."""
    cn = 0
    n = len(polygon.point_list)
    for i in xrange(n):
        a = polygon.point_list[i]
        b = polygon.point_list[(i+1) % n]
        if a.y <= point.y:
            if b.y > point.y and orientation(a, b, point) > 0:
                cn += 1   # upward edge
        else:   # a.y > point.y
            if b.y <= point.y and orientation(a, b, point) < 0:
                cn += 1   # downward edge
    return cn
```

Dla porównania przedstawimy także drugą wersję funkcji obliczającej liczbę przecięć, gdzie wyznaczany jest punkt przecięcia. Ta wersja była przedstawiona w roku 1962 przez Shimrata [32].

Listing 4.5. Obliczanie liczby przecięć (punkt przecięcia).

```
def crossing_number2(polygon, point):
    """Return the crossing number for a point."""
```

```
    cn = 0
    n = len(polygon.point_list)
    for i in xrange(n):
        a = polygon.point_list[i]
        b = polygon.point_list[(i+1) % n]
        if ((a.y <= point.y and b.y > point.y) or
            (a.y > point.y and b.y <= point.y)):
            vt = float(point.y - a.y) / (b.y - a.y)
            xmid = a.x + vt * (b.x - a.x)
            if point.x < xmid:
                cn += 1
    return cn
```

Liczba nawinięć dla wielokąta może być obliczana tak jak dla zamkniętej krzywej ciągłej, ale jest to kosztowne obliczeniowo. Prościej jest sprawdzać w którym kierunku krawędzie przecinają ustalony promień wychodzący z badanego punktu. Przecięcie zgodne z kierunkiem wskazówek zegara jest ujemne (-1), a przecięcie przeciwne do ruchu wskazówek zegara jest dodatnie $(+1)$. Należy stosować te same reguły przecięć krawędzi, jak dla liczby przecięć, co prowadzi do bardzo zbliżonego kodu.

Listing 4.6. Obliczanie liczby nawinięć.

```
def winding_number(polygon, point):
    """Return the winding number for a point."""
    wn = 0
    n = len(polygon.point_list)
    for i in xrange(n):
        a = polygon.point_list[i]
        b = polygon.point_list[(i+1) % n]
        if a.y <= point.y:
            if b.y > point.y and orientation(a, b, point) > 0:
                wn += 1   # upward edge
        else:   # a.y > point.y
            if b.y <= point.y and orientation(a, b, point) < 0:
                wn -= 1   # downward edge
    return wn
```

Funkcje obliczające liczbę przecięć i liczbę nawinięć są umieszczone w module polygons. Metoda Polygon.__contains__() wykorzystuje liczbę nawinięć, dzięki czemu sprawdzenie należenia punktu do wielokąta można wykonać przez test point **in** polygon.

Listing 4.7. Należenie punktu do wielokąta.

```
class Polygon:
    """The class defining a polygon."""

    def __contains__(self, other):
        """Test if a point is in a polygon."""
        if isinstance(other, Point):
            # cn = crossing_number(self, other)
            # return cn % 2 != 0
            wn = winding_number(self, other)
            return wn != 0
        else:
            raise ValueError("not a point")
```

4.6. Należenie punktu do wielokąta wypukłego

Problemy określone dla wielokątów zwykle łatwiej rozwiązuje się dla wielokątów wypukłych. Tak jest w przypadku problemu należenia punktu do wielokąta. Niech A i B będą wierzchołkami wielokąta leżącymi odpowiednio najbardziej na lewo i najbardziej na prawo. Te dwa punkty dzielą wielokąt na łańcuch górny i łańcuch dolny. Każda prosta lub promień przecina wielokąt najwyżej dwa razy, a każda pionowa prosta lub promień przecina każdy łańcuch najwyżej raz.

Za pomocą przeszukiwnia binarnego można szukać współrzędnej x dla punktów w obu łańcuchach, aby znaleźć krawędzie przecinane przez promień. Daje to złożoność $O(\log n)$. Podane uzasadnienie pochodzi z wykładu Davida Eppsteina, przy czym wydaje się, że łańcuchy punktów uważa się za dane na początku. W przeciwnym razie wyznaczanie łańcuchów prawdopodobnie zajmie czas $O(n)$, dopiero potem jest etap $O(\log n)$.

4.7. Przecinanie się dwóch odcinków

Jednym z podstawowych problemów geometrycznych jest zbadanie wzajemnego położenia dwóch odcinków. Chcemy dostać odpowiedź na pytanie czy dwa odcinki przecinają się. Przyjmujemy, że odcinki są domknięte, czyli zawierają swoje końce. Przez przecięcie rozumiemy także sytuację, kiedy koniec jednego odcinka leży na drugim odcinku.

Podany algorytm wykorzystuje funkcję orientation() do wyznaczenia orientacji końców jednego odcinka względem drugiego odcinka. W ogólnym przypadku odcinki nie są współliniowe, a wtedy sprawdzamy, czy końce jednego odcinka mają różne orientacje względem drugiego odcinka (i na odwrót). W szczególnym przypadku, gdy odcinki są współliniowe, sprawdzamy należenie pewnego końca jednego odcinka do drugiego odcinka (cztery przypadki).

Algorytm jest zawarty w metodzie Segment.intersect(), dzięki czemu korzystamy z niego przez test segment1.intersect(segment2).

Listing 4.8. Przecinanie się dwóch odcinków.

```
class Segment:
    """The class defining a segment."""

# Algorytm na bazie materialow ze strony
# https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/
    def intersect(self, other):
        """Test if two segments intersect."""
        o1 = orientation(self.pt1, self.pt2, other.pt1)
        o2 = orientation(self.pt1, self.pt2, other.pt2)
        o3 = orientation(other.pt1, other.pt2, self.pt1)
        o4 = orientation(other.pt1, other.pt2, self.pt2)
        # General case.
        if (o1 != o2) and (o3 != o4):
            return True
        # Special cases.
        if o1 == 0 and other.pt1 in self:
            return True
```

```
        if o2 == 0 and other.pt2 in self:
            return True
        if o3 == 0 and self.pt1 in other:
            return True
        if o4 == 0 and self.pt2 in other:
            return True
        return False
```

4.8. Orientacja wielokąta prostego

Wielokąt prosty może mieć kolejność wierzchołków zgodną lub przeciwną do ruchu wskazówek zegara. Przyjmujemy konwencję zgodną z funkcją orientation(), kiedy kolejność przeciwna do ruchu wskazówek zegara daje orientację równą 1. W przypadku trójkąta triangle orientacja może być wyznaczona wprost przez orientation(triangle.pt1, triangle.pt2, triangle.pt3), albo przez wywołanie metody triangle.orientation().

W przypadku ogólnego wielokąta prostego problem wyznaczenia orientacji ma co najmniej dwa rozwiązania [10]. Pierwszy sposób to obliczenie zorientowanego pola powierzchni wielokąta, przy czym znak pola powierzchni daje szukaną orientację. Drugi (szybszy) sposób polega na znalezieniu wierzchołka o najmniejszej współrzędnej y i wysuniętego najbardziej na prawo. Wtedy orientację wyznaczy ten punkt i jego sąsiedzi na liście wierzchołków. Złożoność algorytmu wynosi $O(n)$.

Listing 4.9. Orientacja wielokąta prostego.

```
class Polygon:
    """The class defining a polygon."""

# Algorytm na bazie kodu C++ ze strony
# http://geomalgorithms.com/a01-_area.html
    def orientation(self):
        """Simple polygon orientation."""
        # First find rightmost lowest vertex of the polygon.
        rmin = 0
        xmin = self.point_list[0].x
        ymin = self.point_list[0].y
        for i in range(1, len(self.point_list)):
            if self.point_list[i].y > ymin:
                continue
            if self.point_list[i].y == ymin:
                if self.point_list[i].x < xmin:
                    continue
            rmin = i
            xmin = self.point_list[i].x
            ymin = self.point_list[i].y
        # Test orientation at the rmin vertex.
        if rmin == 0:
            return orientation(self.point_list[-1],
                               self.point_list[0],
                               self.point_list[1])
        elif rmin == len(self.point_list)-1:
            return orientation(self.point_list[-2],
```

```
                                self.point_list[-1],
                                self.point_list[0])
        else:
            return orientation(self.point_list[rmin-1],
                                self.point_list[rmin],
                                self.point_list[rmin+1])
```

4.9. Rozpoznawanie wielokątów prostych

Najprostsza metoda testowania, czy dany wielokąt jest prosty, polega na sprawdzeniu ewentualnego przecinania się wszystkich par niesąsiednich krawędzi wielokąta. Złożoność algorytmu wynosi $O(n^2)$. To podejście zostało zaimplementowane w metodzie Polygon.is_simple().

4.10. Rozpoznawanie wielokątów wypukłych

W pierwszym kroku sprawdzamy, czy wielokąt jest prosty, bo tylko dla takich wielokątów działa nasz algorytm. Wykorzystujemy metodę Polygon.is_simple(), przy czym ten krok można pominąć, jeżeli wiemy z góry, że wielokąt jest prosty. W drugim kroku wyznaczamy orientację wielokąta w czasie $O(n)$ za pomocą metody Polygon.orientation(). W trzecim kroku w czasie $O(n)$ sprawdzamy, czy wszystkie kąty wewnętrzne wielokąta są wklęsłe (dopuszczamy też kąt półpełny). Można to łatwo zrobić sprawdzając ułożenie wszystkich kolejnych trójek wierzchołków, które powinny skręcać w lewo dla orientacji wielokąta przeciwnej do ruchu wskazówek zegara lub w prawo dla orientacji zgodnej z ruchem wskazówek zegara. Korzystamy przy tym z funkcji orientation(). Algorytm jest zawarty w metodzie Polygon.is_convex().

4.11. Kontenery ograniczające

W wielu zastosowaniach przydatne jest posiadanie kontenera ograniczającego (ang. *bounding container*) skończony obiekt geometryczny. Prosty kontener może przyspieszyć obliczenia dotyczące kolizji obiektów, *ray tracing*, czy wykrywania obiektów ukrytych [10]. Stosuje się zwykle dwa typy kontenerów: liniowe (prostokąty, wielokąty wypukłe), oraz kwadratowe (okręgi, elipsy). Otoczka wypukła jest najmniejszym liniowym kontenerem ograniczającym.

Najprostszym kontenerem liniowym jest prostokąt ograniczający (ang. *bounding box*). Boki prostokąta ograniczającego są równoległe do osi układu, co jest zgodne z klasą Rectangle z obecnej pracy. Dla zbioru n punktów można wyznaczyć prostokąt ograniczający w czasie $O(n)$.

Listing 4.10. Prostokąt ograniczający.

```
def bounding_box(point_list):
    """Return the bounding box for a point set."""
    xmin = xmax = point_list[0].x
    ymin = ymax = point_list[0].y
    for point in point_list:
```

```
            xmin = min(point.x, xmin)
            xmax = max(point.x, xmax)
            ymin = min(point.y, ymin)
            ymax = max(point.y, ymax)
        return Rectangle(xmin, ymin, xmax, ymax)
        #return Polygon(xmin, ymin, xmax, ymin, xmax, ymax, xmin, ymax)
```

4.12. Algorytm Grahama

Implementacja algorytmu Grahama na wejściu ma listę punktów, a wynik to lista cykliczna zawierająca punkty należące do otoczki wypukłej (listing 4.11). W drugiej implementacji punkty należące do otoczki są ustawione kolejno na początku listy punktów, przez co nie ma potrzeby używania dodatkowej pamięci na otoczkę.

Na początku wszystkie posortowane punkty trafiają do otoczki, a następnie niepotrzebne punkty są usuwane. Lista cykliczna dobrze oddaje fakt, że wielokąt wypukły (otoczka) nie ma jakiegoś wyróżnionego wierzchołka.

Listing 4.11. Algorytm Grahama z listą cykliczną.

```
class GrahamScan:
    """Graham's scan algorithm for finding the convex hull of points."""

    def __init__(self, point_list):
        """The algorithm initialization."""
        if len(point_list) < 3:
            raise ValueError("small number of points")
        self.point_list = point_list
        self.convex_hull = CyclicList()

    def run(self):
        """Executable pseudocode."""
        p_min = min(self.point_list, key=lambda p: (p.y, p.x))
        self.point_list.sort(key=lambda p:
            ((p-p_min).alpha(), (p-p_min)*(p-p_min)))
        for pt in self.point_list:
            self.convex_hull.insert_tail(Node(pt))
        # Po sortowaniu p_min bedzie na poczatku listy.
        # Wstawiamy punkty do listy cyklicznej.
        # potem bedziemy usuwac niepotrzebne punkty.
        node1 = self.convex_hull.head
        node2 = node1.next   # zaliczony tymczasowo do otoczki
        node3 = node2.next
        while node2 != self.convex_hull.head:
            if orientation(node1.data, node2.data, node3.data) > 0:
                # node2 na zewnatrz, przesuwamy sie dalej
                node1 = node2
                node2 = node3
                node3 = node3.next
            else:   # node2 wewnatrz lub brzeg
                if node1 != self.convex_hull.head:
                    self.convex_hull.remove(node2)
                    node2 = node1
                    node1 = node1.prev
```

```
                # node3 bez zmian
            else: # nie mozna ruszyc node1
                self.convex_hull.remove(node2)
                node2 = node3
                node3 = node3.next
```

4.13. Algorytm Jarvisa

Implementacja algorytmu Jarvisa również ma dwie wersje, przy czym listing 4.12 przedstawia wersję z listą cykliczną. Algorytm znajduje kolejne punkty należące do otoczki i umieszcza je na liście cyklicznej. Koniec następuje wtedy, gdy ponownie trafiamy na pierwszy punkt zaliczony do otoczki.

Listing 4.12. Algorytm Jarvisa z listą cykliczną.

```
class JarvisMarch:
    """Jarvis march for finding the convex hull of points."""

    def __init__(self, point_list):
        """The algorithm initialization."""
        if len(point_list) < 3:
            raise ValueError("small number of points")
        self.point_list = point_list
        self.convex_hull = CyclicList()

    def run(self):
        """Executable pseudocode."""
        p_min = min(self.point_list, key=lambda p: (p.y, p.x))
        self.convex_hull.insert_head(Node(p_min))
        last = p_min   # ostatni punkt zaliczony do otoczki
        for pt1 in self.point_list:
            for pt2 in self.point_list:
                if pt2 == pt1:
                    continue
                orient = oriented_area(last, pt1, pt2)
                if orient < 0:
                    pt1 = pt2   # nowy kandydat do otoczki
                elif orient == 0:   # degeneracja
                    segment = Segment(last, pt2)
                    if pt1 in segment:
                        pt1 = pt2
            if pt1 == self.convex_hull.head.data:
                break   # doszlismy do p_min
            else:
                last = pt1
                self.convex_hull.insert_tail(Node(pt1))
```

4.14. Algorytm quickhull

Implementacja algorytmu *quickhull* także ma dwie wersje, a listing 4.13 przedstawia wersję z listą cykliczną. Implementacja naśladuje pseudokod podany wcześniej. Na początku nieuporządkowane punkty trafiają na listę

cykliczną. Dalsze etapy algorytmu operują całymi węzłami, przepinając je pomiędzy tymczasowymi listami. Jest ważne, że operacja łączenia dwóch list cyklicznych działa w stałym czasie.

Listing 4.13. Algorytm *quickhull* z listą cykliczną.

```
class QuickHull:
    """Quickhull algorithm for finding the convex hull of points."""

    def __init__(self, point_list):
        """The algorithm initialization."""
        if len(point_list) < 3:
            raise ValueError("small number of points")
        self.point_list = point_list
        self.convex_hull = CyclicList()

    def run(self):
        """Executable pseudocode."""
        clist = CyclicList()
        for pt in self.point_list:   # O(n) time
            clist.insert_head(Node(pt))
        node1 = clist.find_min(key_func=lambda node:
            (node.data.y, node.data.x))
        node2 = clist.find_max(key_func=lambda node:
            (node.data.y, node.data.x))
        clist.remove(node1)
        clist.remove(node2)
        clist1 = self._points_on_the_right(node1, node2, clist)
        clist2 = self._points_on_the_right(node2, node1, clist)
        # Po tych operacjach czesc wezlow przeszla z clist
        # do clist1 lub clist2.
        self.convex_hull.insert_tail(node1)
        self.convex_hull.merge( self._quickhull(node1, node2, clist1) )
        self.convex_hull.insert_tail(node2)
        self.convex_hull.merge( self._quickhull(node2, node1, clist2) )
        clist.clear()

    def _points_on_the_right(self, node1, node2, clist):
        """Return the list of points on the right side of the oriented
        line from node1 to node2."""
        if clist.is_empty():
            return CyclicList()
        new_clist = CyclicList()
        node = clist.head
        counter = clist.length
        while counter > 0:
            if oriented_area(node1.data, node2.data, node.data) < 0:
                current = node   # wezel do przeniesienia
                node = node.next
                clist.remove(current)
                new_clist.insert_head(current) # lub insert_tail
            else:
                node = node.next
            counter -= 1
        # Tu nie wolno zrobic clist.clear(), bo punkty beda potrzebne.
        return new_clist

    def _quickhull(self, node1, node2, clist):
```

```
        """Find points on convex hull from the list that are
        on the right side of the oriented line from node1 to node2."""
        if clist.is_empty():
            return CyclicList()
        # Szukam wezla z punktem najbardziej oddalonym od odcinka node1_node2.
        node3 = clist.find_max(key_func=lambda node:
            oriented_area(node1.data, node.data, node2.data))
        clist.remove(node3)
        clist1 = self._points_on_the_right(node1, node3, clist)
        clist2 = self._points_on_the_right(node3, node2, clist)
        hull = CyclicList()
        hull.merge( self._quickhull(node1, node3, clist1) )
        hull.insert_tail(node3)
        hull.merge( self._quickhull(node3, node2, clist2) )
        clist.clear()
        return hull
```

4.15. Generowanie par punktów antypodycznych

Parą punktów antypodycznych są dwa punkty leżące na dwóch prostych wspierających stycznych do wielokąta wypukłego. Znajomość par punktów antypodycznych jest potrzebna m.in. do wyznaczenia średnicy zbioru punktów. Przedstawiony algorytm wykorzystuje technikę obracających się suwmiarek i działa w czasie $O(n)$ [generator iter_all_antipodal_pairs()].

Listing 4.14. Generator punktów antypodycznych.

```
def iter_all_antipodal_pairs(point_list):
    """Generate all antipodal pairs for a convex polygon."""
    L = point_list
    n = len(L)   # liczba punktow wielokata wypuklego
    i = n-1   # ostatni
    j = 0   # nastepny po i
    # Szukamy punktu polozonego najdalej od odcinka L[n-1]L[0].
    # Prosta wspierajaca zawiera ten odcinek.
    while (oriented_area(L[i], L[(i+1) % n], L[(j+1) % n]) >
        oriented_area(L[i], L[(i+1) % n], L[j])):
        j = (j + 1) % n
    k = j   # najdalszy punkt
    while True: # to while zmienia i
        i = (i + 1) % n   # przejscie na koniec odcinka i print
        yield L[i], L[j]   # raportuje pierwsza pare (0, k), potem nastepne
        # To while zmienia j, znowu szukamy najdalszego punktu.
        # Petla moze sie zatrzymac na krawedziach rownoleglych.
        while (oriented_area(L[i], L[(i+1) % n], L[(j+1) % n]) >
            oriented_area(L[i], L[(i+1) % n], L[j])):
            j = (j + 1) % n
            if j == 0: # nie mozna przekroczyc n-1
                return
            yield L[i], L[j]
        # Obsluga krawedzi rownoleglych.
        if (oriented_area(L[i], L[(i+1) % n], L[(j+1) % n]) ==
            oriented_area(L[i], L[(i+1) % n], L[j])):
            if j != (n-1): # nie chcemy wypisac j=0
                # Bierzemy nastepne j, ale nie przesuwamy j.
```

```
yield L[i], L[(j+1) % n]
```

4.16. Wyznaczenie pary najdalszych punktów

Rozważamy problem znalezienia pary najdalszych punktów dla danego zbioru n punktów P. Odległość między tymi dwoma punktami jest równa średnicy zbioru punktów.

Prosta metoda polega na sprawdzeniu każdej pary punktów ze zbioru P, co daje algorytm działający w czasie $O(n^2)$ [funkcja find_two_furthest_points1()].

Optymalna metoda polega na wyznaczeniu otoczki wypukłej $\mathrm{CH}(P)$ w czasie $O(n \log n)$, a następnie na sprawdzeniu w czasie $O(n)$ wszystkich par punktów antypodycznych należących do otoczki wypukłej $\mathrm{CH}(P)$ Funkcja find_two_furthest_points2() działa dla listy wierzchołków wielokąta wypukłego. Zauważmy, że wewnątrz funkcji zamiast odległości porównywane są kwadraty odległości, co pozwala uniknąć czasochłonnego obliczania pierwiastka kwadratowego i dla liczb wymiernych daje wyniki dokładne.

Listing 4.15. Wyznaczanie pary najdalszych punktów.

```
def find_two_furthest_points2(point_list):
    """Find two furthest points in O(n) time for a convex polygon."""
    dist2 = 0
    pair = None
    for (pt1, pt2) in iter_all_antipodal_pairs(point_list):
        vec = pt2 - pt1
        new_dist2 = vec * vec
        if new_dist2 > dist2:
            dist2 = new_dist2
            pair = pt1, pt2
    return pair
```

5. Podsumowanie

W ramach pracy przygotowano kilka klas odpowiadających podstawowym obiektom geometrycznym na płaszczyźnie, takim jak punkt (wektor 2D), odcinek, trójkąt, prostokąt (boki równoległe do osi układu), wielokąt, okrąg. Metody klas zawierają wiele przydatnych algorytmów, np. sprawdzenie należenia punktu do figury, przecinanie się dwóch odcinków, obliczanie pola powierzchni.

Przygotowano trzy algorytmy wyznaczające otoczkę wypukłą dla danego zbioru punktów: algorytm Grahama (dwie wersje), algorytm Jarvisa (dwie wersje), algorytm quickhull (dwie wersje). Algorytmy prezentują różne techniki programistyczne zastosowane do jednego problemu, przez co mają wartość dydaktyczną.

Najtrudniejsze algorytmy związane są z wielokątami, które są zadane przez ciągi punktów (wierzchołków). W celu sprawdzenia należenia punktu do wielokąta złożonego przygotowano dwie metody: metodę liczby przecięć i metodę liczby nawinięć. Mamy test sprawdzający, czy wielokąt jest prosty, a następnie test czy wielokąt prosty jest wypukły. Zaimplementowano szybki algorytm wyznaczający orientację wielokąta prostego. Technikę obracających się suwmiarek zastosowano do generowania wszystkich par punktów antypodycznych wielokąta wypukłego, co z kolei wykorzystano w optymalnej metodzie wyznaczania pary najdalszych punktów i średnicy zbioru punktów.

Powstały kod w języku Python został sprawdzony pod względem poprawności (testy z modułem unittest) i pod względem wydajności obliczeniowej (testy z modułem timeit). Planowane jest w przyszłości powiększanie bazy algorytmów o nowe elementy i nowe techniki stosowane w geometrii obliczeniowej.

A. Testy algorytmów

Dodatek zawiera wyniki testów algorytmów działających na zbiorach punktów i na wielokątach. Stosowano następujące zbiory punktów:

— punkty losowo rozmieszczone w kwadracie,
— punkty rozmieszczone w węzłach sieci kwadratowej bez rogów,
— punkty rozmieszczone w równych odstępach na okręgu,
— punkty rozmieszczone nierównomiernie na okręgu.

W przypadku algorytmu Grahama ważne było wstępne wymieszanie punktów, ponieważ sortowanie w Pythonie potrafi wykorzystać częściowy porządek w danych, co prowadzi do przyspieszenia działania algorytmu, a nam chodziło o uzyskanie niekorzystanego stanu początkowego.

A.1. Testy wielokątów

Sprawdzono wybrane metody związane z wielokątami.

A.1.1. Test wyznaczania orientacji wielokątów

Testy sprawdzające algorytm wyznaczania orientacji wielokąta prostego wykonano dla wielokąta foremnego z liczbą boków równą n. Potwierdzono złożoność $O(n)$ metody wyznaczającej orientację (wykres A.1).

A.1.2. Test rozpoznawania wielokątów prostych

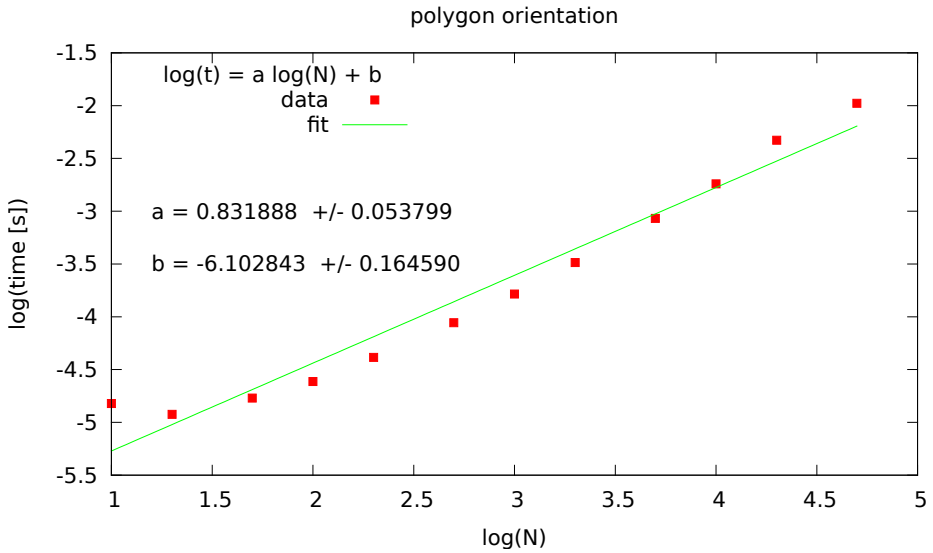
Testy sprawdzające czy wielokąt jest prosty wykonano dla wielokąta foremnego z liczbą boków równą n (metoda Polygon.is_simple()). Potwierdzono złożoność $O(n^2)$ prostej metody rozpoznawania wielokątów prostych (wykres A.2).

A.1.3. Test rozpoznawania wielokątów wypukłych

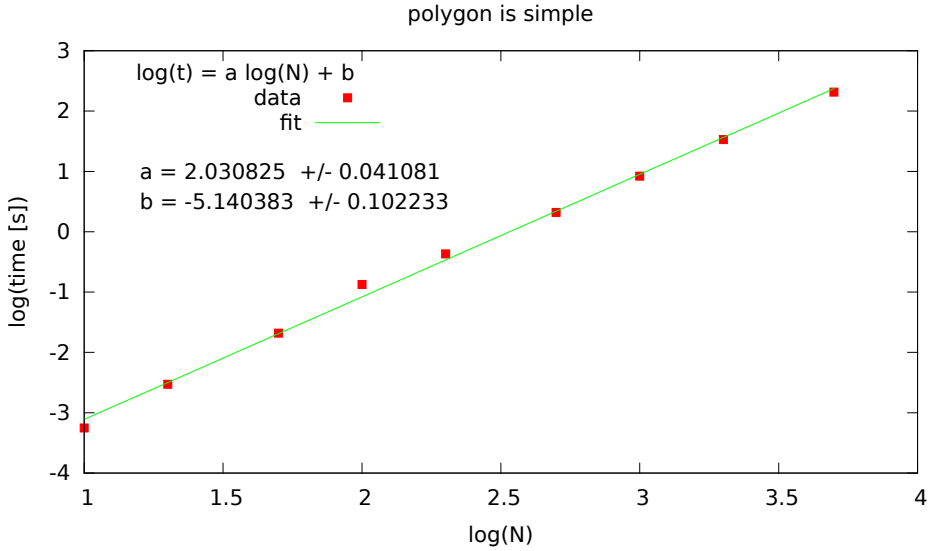
Testy sprawdzające czy wielokąt prosty jest wypukły wykonano dla wielokąta foremnego z liczbą boków równą n (metoda Polygon.is_convex()). Na potrzeby testu wyłączyliśmy sprawdzanie czy wielokąt jest prosty. Potwierdzono złożoność $O(n)$ metody sprawdzającej wypukłość (wykres A.3).

A.2. Testy otoczki wypukłej

Testy wykonano dla kilku typów zbiorów punktów, aby sprawdzić łatwe i trudne przypadki. W pierwszym zbiorze punkty leżały w węzłach sieci $s \times s$,



Rysunek A.1. Wykres wydajności algorytmu wyznaczania orientacji wielokąta prostego na przykładzie wielokąta foremnego. Współczynnik a bliski 1 potwierdza złożoność liniową $O(n)$.

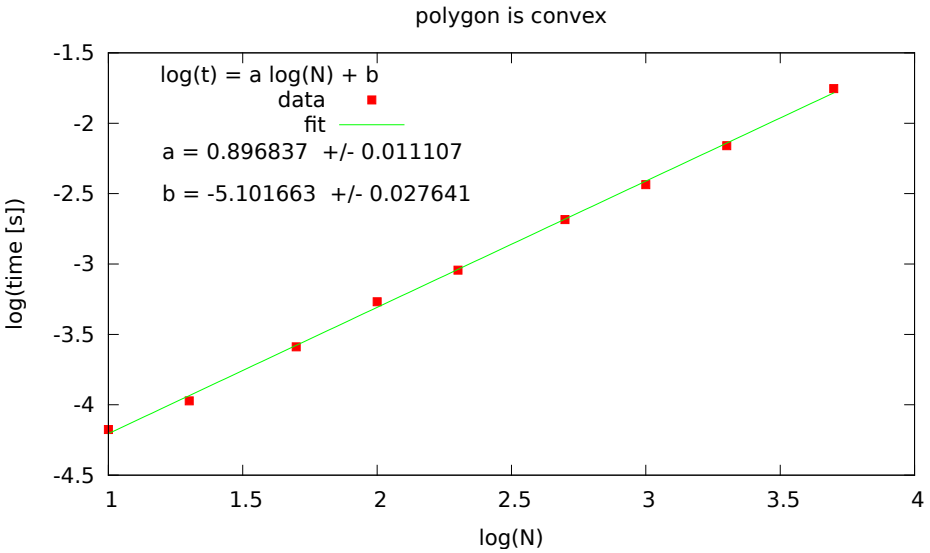


Rysunek A.2. Wykres wydajności testu sprawdzającego czy wielokąt jest prosty na przykładzie wielokąta foremnego. Współczynnik a bliski 2 potwierdza złożoność kwadratową $O(n^2)$.

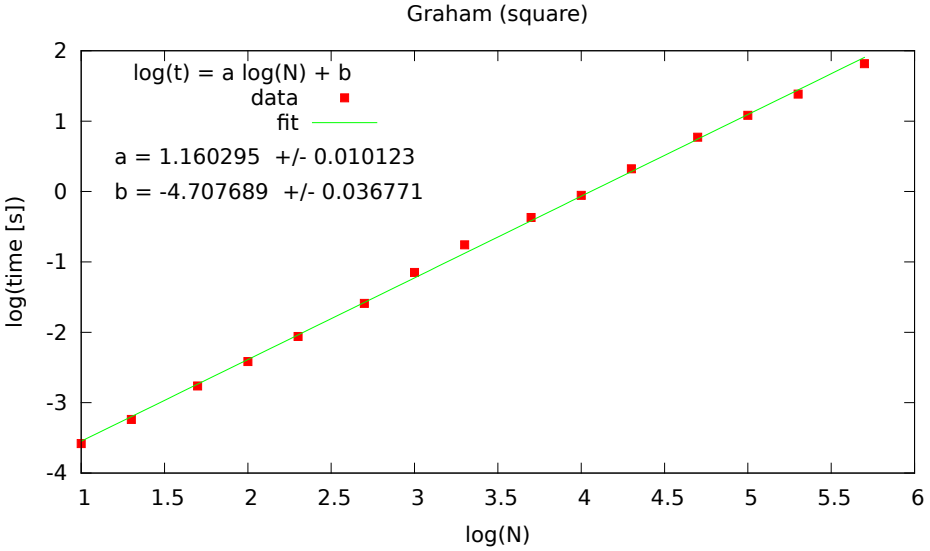
z pustymi narożnikami. Liczba punktów wynosi $n = s^2 - 4$, a liczba punktów należących do otoczki wypukłej wynosi $h = 8$, czyli jest stała. W drugim zbiorze n punktów było równomiernie rozłożonych na okręgu, przez co wszystkie punkty należą do otoczki ($h = n$). W trzecim zbiorze punkty również były rozłożone na okręgu, ale zagęszczały się przy jednym punkcie. Trzeci zbiór punktów był przygotowany z myślą o algorytmie *quickhull*, aby wymusić niekorzystne podziały punktów na podzbiory.

Wyniki testów przedstawiają wykresy:
— Algorytm Grahama (wykresy A.4, A.5),
— Algorytm Jarvisa (wykresy A.6, A.7),
— Algorytm *quickhull* (wykresy A.8, A.9).

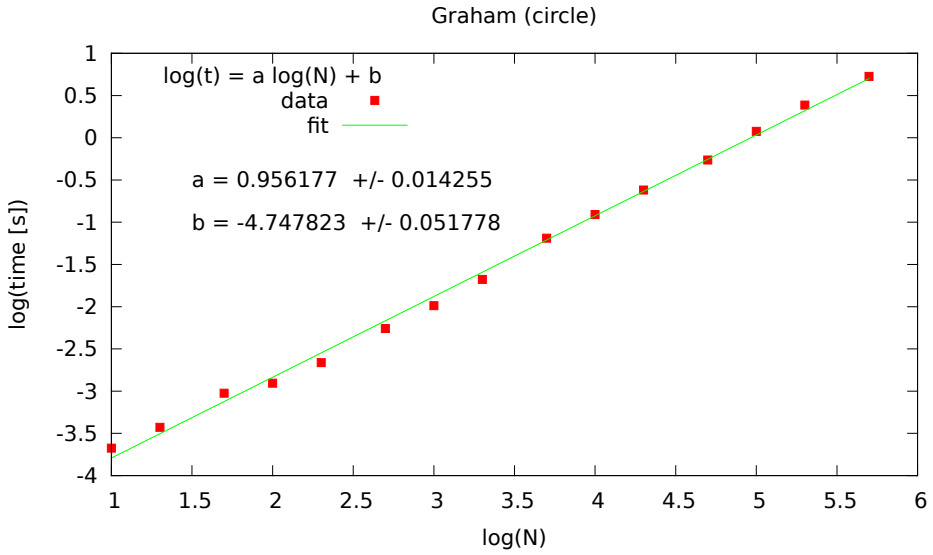
Dla punktów na sieci kwadratowej najszybszy był algorytm *quickhull*, a trochę wolniejszy algorytm Jarvisa. Czas pracy tych algorytmów był praktycznie liniowy z liczbą punktów. Dla punktów rozłożonych równomiernie na okręgu najszybszy był algorytm Grahama, a zaraz za nim *quickhull*. Oba algorytmy miały w praktyce złożoność $O(n \log n)$. Najwolniejszy algorytm Jarvisa ujawnił kwadratową zależność od liczby punktów.



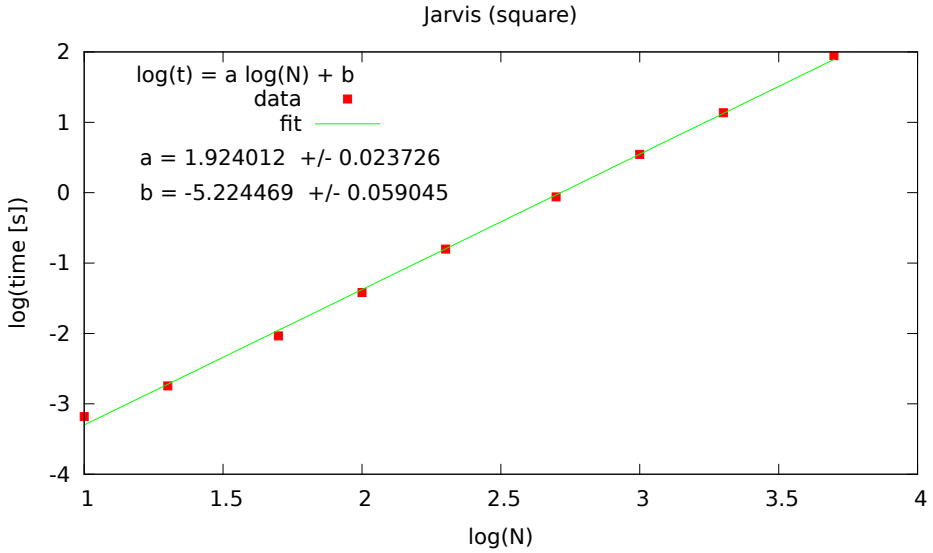
Rysunek A.3. Wykres wydajności algorytmu sprawdzającego wypukłość wielokąta prostego na przykładzie wielokąta foremnego. Współczynnik a bliski 1 potwierdza złożoność liniową $O(n)$.



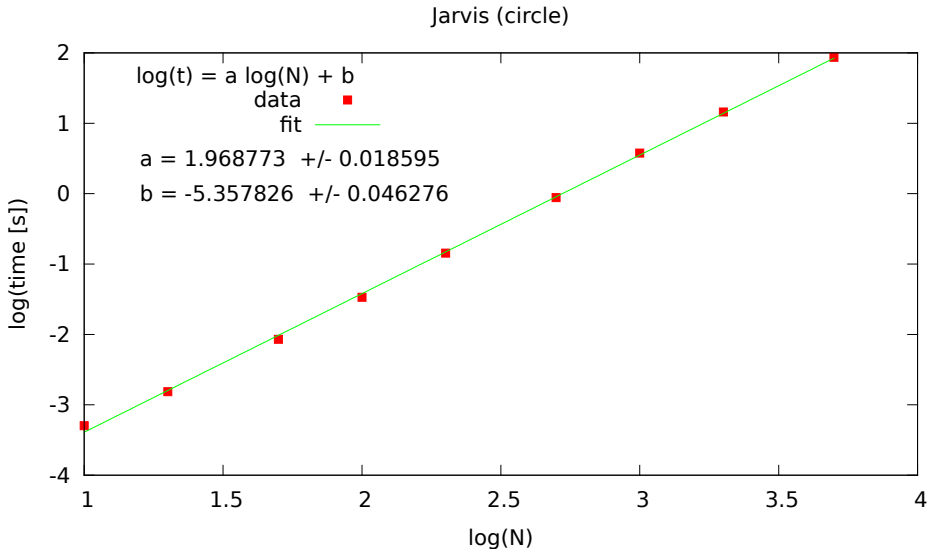
Rysunek A.4. Wykres wydajności algorytmu Grahama dla punktów na kwadracie. Współczynnik a lekko przekraczający 1 jest zgodny z teoretyczną złożonością $O(n \log n)$.



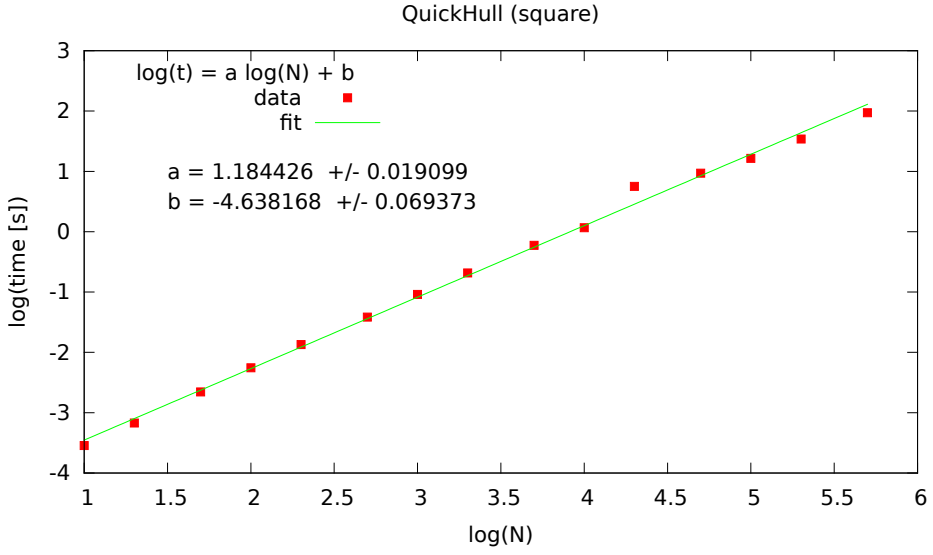
Rysunek A.5. Wykres wydajności algorytmu Grahama dla punktów na okręgu. Współczynnik a bliski 1 jest zgodny z teoretyczną złożonością $O(n \log n)$.



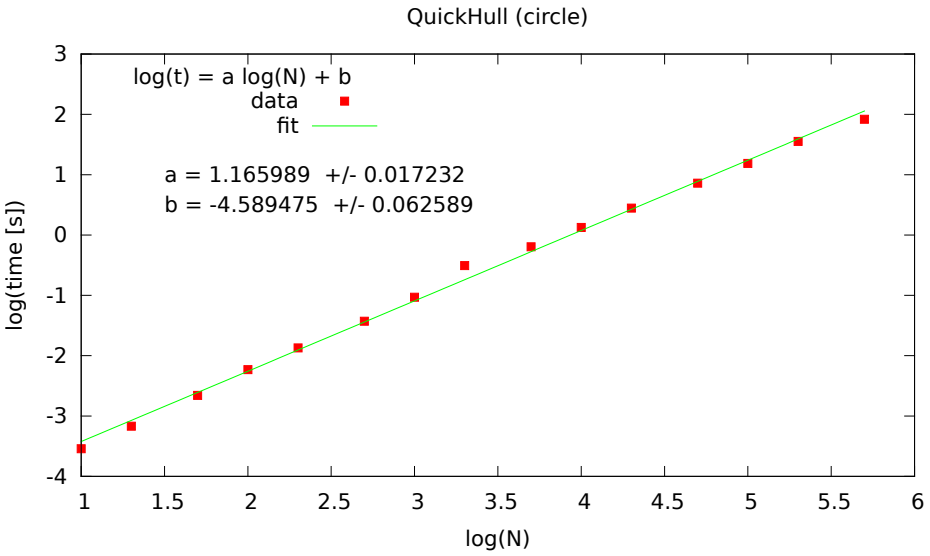
Rysunek A.6. Wykres wydajności algorytmu Jarvisa dla punktów na kwadracie. Współczynnik a bliski 2 jest zgodny z teoretyczną złożonością $O(n^2)$ dla niekorzystnego przypadku rosnącej z n liczby punktów należących do otoczki.



Rysunek A.7. Wykres wydajności algorytmu Jarvisa dla punktów na okręgu. Współczynnik a bliski 2 jest zgodny z teoretyczną złożonością $O(hn) = O(n^2)$.



Rysunek A.8. Wykres wydajności algorytmu quickhull dla punktów na kwadracie. Współczynnik a przekraczający 1 wskazuje na złożonością $O(n \log n)$. Inne testy pokazywały nawet złożoność $O(n)$.



Rysunek A.9. Wykres wydajności algorytmu quickhull dla punktów na okręgu. Współczynnik a przekraczający 1 wskazuje na złożoność $O(n \log n)$ dla tego typu zbioru punktów. Kolejne wywołania rekurencyjne korzystnie dzieliły punkty na równe podzbiory.

Bibliografia

[1] Wikipedia, Computational geometry, 2018, https://en.wikipedia.org/wiki/Computational_geometry.

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

[3] Franco P. Preparata, Michael Ian Shamos, *Geometria obliczeniowa: wprowadzenie*, Helion, Gliwice 2003.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

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

[6] Wikipedia, General position, 2018, https://en.wikipedia.org/wiki/General_position.

[7] The Computational Geometry Algorithms Library (CGAL), 2018, https://www.cgal.org/.

[8] Library of Efficient Data types and Algorithms (LEDA), 2018, http://www.algorithmic-solutions.com/leda/.

[9] GeeksforGeeks, A computer science portal for geeks, 2018, https://www.geeksforgeeks.org/.

[10] Dan Sunday, Geometry Algorithms, 2018, http://geomalgorithms.com/.

[11] Algorytmy i struktury danych, 2018, http://www.algorytm.org/.

[12] Wikipedia, Polygon, 2018, https://en.wikipedia.org/wiki/Polygon.

[13] Wikipedia, Convex polygon, 2018, https://en.wikipedia.org/wiki/Convex_polygon.

[14] Wikipedia, Monotone polygon, 2018, https://en.wikipedia.org/wiki/Monotone_polygon.

[15] Bernard Chazelle, *Triangulating a Simple Polygon in Linear Time*, Discrete & Computational Geometry 6, 485-524 (1991).

[16] Wikipedia, Convex hull, 2018, https://en.wikipedia.org/wiki/Convex_hull.

[17] Wikipedia, Convex hull algorithms, 2018, https://en.wikipedia.org/wiki/Convex_hull_algorithms.

[18] Wikipedia, Graham scan, 2018, https://en.wikipedia.org/wiki/Graham_scan.

[19] R. L. Graham, *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*, Information Processing Letters 1, 132-133 (1972).

[20] Wikipedia, Gift wrapping algorithm, 2018, https://en.wikipedia.org/wiki/Gift_wrapping_algorithm.

[21] Wikipedia, Quickhull, 2018, https://en.wikipedia.org/wiki/Quickhull.

[22] Wikipedia, Chan's algorithm, 2018,
https://en.wikipedia.org/wiki/Chan%27s_algorithm.
[23] T. M. Chan, *Optimal output-sensitive convex hull algorithms in two and three dimensions*, Discrete Comput Geom 16, 361-368 (1996).
[24] Wikipedia, Kirkpatrick-Seidel algorithm, 2018,
https://en.wikipedia.org/wiki/Kirkpatrick-Seidel_algorithm.
[25] Wikipedia, Rotating calipers, 2018,
https://en.wikipedia.org/wiki/Rotating_calipers.
[26] Michael Ian Shamos, *Computational Geometry*, Yale University, Connecticut, USA, 1978.
[27] Godfried T. Toussaint, *Solving geometric problems with the rotating calipers*, Proceedings of IEEE MELECON'83, Athens, Greece (1983).
[28] Hormoz Pirzadeh, *Computational geometry with the rotating calipers*, McGill University, Montreal, Quebeck, Canada, 1999.
[29] Wikipedia, Bentley-Ottmann algorithm, 2018,
https://en.wikipedia.org/wiki/Bentley-Ottmann_algorithm.
[30] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, Chee Yap, *Classroom examples of robustness problems in geometric computations*, Computational Geometry 40, 61-78 (2008).
[31] Wikipedia, Point in polygon, 2018,
https://en.wikipedia.org/wiki/Point_in_polygon.
[32] M. Shimrat, *Algorithm 112: Position of point relative to polygon*, Communications of the ACM 5(8), 434 (1962).