Algorytmy geometryczne
Sprawozdanie z Laboratorium 4

Adam Naumiec

naumiec@student.agh.edu.pl
410936
Grupa 4 – czwartek, 11:20-12:50, tydzień B

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Informatyki, Elektroniki i Telekomunikacji

Grudzień, MMXXII

Spis treści

1. Dane techniczne komputera, na którym wykonywano obliczenia 3

2. Cel i plan ćwiczenia 4

3. Tolerancje dla zera 5

4. Wyznacznik 6

5. Generacja, wizualizacja, przechowywanie, zapisywanie i wczytywanie odcinków 7

5.1. Generacja odcinków 7
5.2. Wizualizacja odcinków 7
5.3. Przechowywanie odcinków 7
5.4. Zapisywanie odcinków 7
5.5. Wczytywanie odcinków 7

6. Definicje 8

6.1. Struktura stanu 8
6.2. Struktura zdarzeń 8
6.3. Warunki losowego generowania odcinków 9
6.4. Wizualizacja działania algorytmów 9

7. Algorytmy 10

7.1. Algorytmy generacji losowych odcinków 10
7.2. Algorytm zamiatania – oba przypadki 10
7.2.1. Opis i działanie algorytmu pierwszego – sprawdzanie, czy w zbiorze występuje choć jedno przecięcie 10
7.2.2. Opis i działanie algorytmu drugiego – wykrywanie wszystkich przecięć w zbiorze. 11

8. Działanie algorytmu 12

8.1. Losowy zbiór linii $\boldsymbol{ZR}$ 12
8.2. Testowy zbiór linii $\boldsymbol{Z1}$ 15
8.3. Testowy zbiór linii $\boldsymbol{Z2}$ 18

9. Wnioski i podsumowanie 20

1. Dane techniczne komputera, na którym wykonywano obliczenia

Wykorzystano:

- komputer z 64-bitowym systemem *macOS* 13 *Ventura*;
- czterordzeniowy procesor *Intel Core i*5;
- środowisko *Jupyter Notebook*, *JetBrains PyCharm*;
- język programowania *Python* 3 (3.11);
- wykorzystane biblioteki: *math* (wartość π), *matplotlib* (rysowanie wykresów), *time* (sprawdzanie czasu wykonania obliczeń);
- do przygotowania sprawozdania wykorzystano programy *Microsoft Word*, *Microsoft Excel*;
- do rysowania wielokątów: dostarczone na laboratoriach narzędzie graficzne oparte na bibliotekach *matplotlib* i *numpy*.

2. Cel i plan ćwiczenia

W laboratorium przygotowano aplikację graficzną tak, aby można było zadawać w sposób interaktywny kolejne odcinki (reprezentowane przez pary wierzchołków), a także generować losowo zadaną liczbę odcinków z podanego zakresu współrzędnych 2D. Program umożliwia także zapis i odczyt zbioru odcinków.

W ćwiczeniu zaimplementowano:

- procedurę pozwalającą na wprowadzanie odcinków;
- funkcję generowania losowego zbioru odcinków o zadanej liczbie na płaszczyźnie 2D;
- algorytm zamiatania sprawdzający, czy choć jedna para odcinków się przecina;
- algorytm zamiatania wyznaczający wszystkie przecięcia odcinków;
- wizualizację działania algorytmów zamiatania.

Na koniec przetestowano działanie programu na różnych zestawach danych.

3. Tolerancje dla zera

Klasyfikacji figur geometrycznych dokonywano z przyjętą tolerancją dla zera (oznaczaną jako *epsilon*). Wykorzystanie tolerancji związane jest z reprezentacją liczb rzeczywistych w komputerze i wynikającym z tego obarczeniem wyników obliczeń niedokładnością przy wykorzystaniu liczb zmiennoprzecinkowych (w języku *Python* typ zmiennej $float$ – 64-bitowa liczba zmiennoprzecinkowa).

Dokonano obliczeń dla tolerancji:

$$\varepsilon = 10^{-12}.$$

4. Wyznacznik

Do obliczania wyznacznika przygotowano własną implementację funkcji wykonującej stosowne obliczenia odpowiednimi sposobami, w laboratorium wykorzystano własny wyznacznik 2x2.

Punkty w zbiorach sklasyfikowano na te znajdujące się po lewej, po prawej oraz współliniowe (znajdujące się na jednej prostej) na podstawie wartości obliczonego wyznacznika.

Klasyfikacji dokonano na podstawie wartości wyznacznika dla punktów a, b i badanego punktu c. Wykorzystano następującą zależność:

Punkt c jest wobec punktów (prostej łączącej punkty) a i b:

$$\det(a,b,c)\begin{cases}(-\infty,0) \Rightarrow po\ prawej,\\ 0 \Rightarrow współliniowy,\\ (0,+\infty) \Rightarrow po\ lewej.\end{cases}$$

5. Generacja, wizualizacja, przechowywanie, zapisywanie i wczytywanie odcinków

5.1. Generacja odcinków

Odcinki generowane były w przestrzeni $\mathbb{R}^2$ z metryką euklidesową (współrzędne punktów wyrażone były liczbami rzeczywistymi). Odcinki można było zadawać za pomocą myszki w zmodyfikowanym programie.

5.2. Wizualizacja odcinków

Do wizualizacji odcinków na płaszczyźnie kartezjańskiej wykorzystano dostarczone na laboratoriach narzędzie graficzne korzystające z bibliotek *matplotlib* oraz *numpy*. Pierwsza z nich pozwala na rysowanie wykresów, druga dostarcza wiele narzędzi przydatnych w algorytmice i obliczeniach komputerowych.

5.3. Przechowywanie odcinków

Odcinki przechowywane są jako dwa punkty – początkowy i końcowy. Punkt reprezentowany jest jako para liczb rzeczywistych odpowiadających kolejno współrzędnej x i y.

5.4. Zapisywanie odcinków

Klasa *Plot* posiadała metodę *toJson*(), która zwracała łańcuch znaków zawierający listę scen w formacie *JSON*. Taki łańcuch można zapisać do pliku stosując standardowe sposoby dostępne w *Pythonie*. Do zapisu skorzystano z metody *dumps* z biblioteki *json*.

5.5. Wczytywanie odcinków

Kolejne etapy budowy wielokąta zapisywane były jako sceny. Wczytanie listy odcinków z pliku dokonywało się poprzez podanie parametru *json* w wywołaniu klasy *Plot*, która przygotowywała listę (która implementowała zbiór odcinków) poprzez jej wczytanie z użyciem metody *load* z biblioteki *json*.

6. Definicje

6.1. Struktura stanu

W implementacji struktury stanu została użyta struktura *SortedSet* która w *Pythonie* występuje w bibliotece *sortedcontainers*.

Struktura ta jest reprezentowana przez drzewo czerwono-czarne, dzięki czemu możliwe jest wstawianie, usuwanie oraz znajdowanie poprzednika/następnika elementu w czasie $O(\log n)$.

Struktura ta została użyta jako struktura stanu miotły, która przechowuje posortowane odcinki względem współrzędnej y.

6.2. Struktura zdarzeń

Do struktury zdarzeń została użyta struktura z biblioteki *heapq*, na której wywoływane są funkcje *heapify* oraz *heapop*. Struktura ta reprezentuje kolejkę priorytetową typu *min*, gdzie jej elementami są współrzędne x punktów, które należą do aktualnego zdarzenia.

Wykorzystanie takiej struktury jest spowodowane faktem, iż algorytm będzie po kolei przeglądał zdarzenia, ale do naszego *heapq* nigdy nie zostanie dodane kolejne (inne) nowe zdarzenie, jeżeli algorytm (pierwszy raz) wykryje przecięcie się dwóch odcinków.

W drugiej wersji algorytmu do struktury zdarzeń wykorzystano *SortedSet* zarówno do struktury stanu jak i struktury zdarzeń. Wynika to z żądanej funkcjonalności jaką jest znajdowanie wszystkich punktów przecięcia, a nie tylko sprawdzanie, czy choć jedno przecięcie się odcinków występuje w zbiorze. Z tego faktu wynika, że struktura zrealizowana z użyciem *SortedSet* działałaby poprawnie w obu przypadkach, natomiast *heapq* daje gwarancję poprawności tylko dla pierwszej wersji działania algorytmu, ale jednocześnie pozwala na szybsze (w przypadku tych funkcji) wykonanie algorytmu. Modyfikacja ta pozwala na prawidłowe działanie algorytmu w obu wersjach działania programu.

6.3. Warunki losowego generowania odcinków

Odcinki pionowe powinny być eliminowane (odcinki o tej samej współrzędnej x w punkcie na początku i końcu odcinka – w tolerancji dla liczb zmiennoprzecinkowych) oraz żadna para odcinków nie powinna mieć końców odcinków o tej samej współrzędnej x (również wartości równych w tolerancji).

6.4. Wizualizacja działania algorytmów

W wizualizacji algorytmu przyjęto następujące oznaczenia:

- *niebieskie linie* – zbiór odcinków,
- *czerwona linia pionowa* – reprezentacja miotły,
- *pomarańczowy punkt* – obecnie sprawdzany punkt,
- *czerwony punkt* – punkt zakwalifikowany jako punkt przecięcia odcinków,
- *czerwona linia* – obecnie sprawdzany odcinek w zbiorze.

7. Algorytmy

7.1. Algorytmy generacji losowych odcinków

Algorytm losuje dwa razy dwie liczby losowe wyznaczające dwa końce odcinka. Jeżeli odcinek jest pionowy lub jedna ze współrzędnych jest końcem innego odcinka to losowanie jest powtarzane. Odcinek jest zapisywano jako krotka dwóch punktów – początku i końca, przy czym początek jest mniejszy leksykograficznie. Sprawdzanie, czy współrzędna już wystąpiła w zbiorze jest realizowane za pomocą zbioru (*set*).

7.2. Algorytm zamiatania – oba przypadki

7.2.1. Opis i działanie algorytmu pierwszego – sprawdzanie, czy w zbiorze występuje choć jedno przecięcie

Pierwszym algorytmem, który należało zaimplementować jest algorytm sprawdzający czy dowolne dwie pary odcinków w zadanym zbiorze przecinają się.

Algorytm szuka punktu przecięcia w oparciu o algorytm zamiatania. Polega to na przeglądaniu zdarzeń i sprawdzaniu, czy dodanie lub usuniecie odcinka ze struktury zdarzeń spowoduje znalezienie przecięcia. Jeśli zdarzenie jest początkiem odcinka, to wykorzystując metodę *add_line*, miotła dodaje do struktury stanu nowy odcinek, a następnie sprawdza czy istnieje przecięcie tego odcinka z innym. Jeśli zdarzenie jest końcem odcinka to wykonuje procedurę *remove_line*, która polega najpierw na sprawdzeniu czy linia ta przecina się z jakąś inną, a następnie usuniecie tego odcinka ze struktury. Jeżeli natomiast jest to zdarzenie przecięcia się odcinków (występuje ono podczas procedur *add_line* oraz *remove_line*) to program kończy działanie, ponieważ znalazł dwa dowolne odcinki które się przecinają.

7.2.2. Opis i działanie algorytmu drugiego – wykrywanie wszystkich przecięć w zbiorze

Algorytm opiera się na bardzo podobnej procedurze co algorytm sprawdzający czy dowolne dwie pary odcinków w zadanym zbiorze przecinają się.

Struktura stanu miotły jest również zaimplementowana w taki sam sposób, czyli poprzez strukturę *SortedSet* z biblioteki *sortedcontainers*. Jednak w tym przypadku, struktura ta została użyta również do struktury zdarzeń. Zmiana to była konieczna i dzięki niej unikamy wielokrotnego dodania tego samego punktu przecięcia do zbioru. W poprzednim algorytmie nie miało to znaczenia, ponieważ wykonywał się do momentu znalezienia pierwszego punktu przecięcia. Również dzięki tej strukturze w łatwy i szybki sposób możemy wstawiać i usuwać elementy.

Algorytm szuka punktu przecięcia w oparciu o algorytm zamiatania. Polega to na przeglądaniu zdarzeń i sprawdzaniu, czy dodanie lub usunięcie odcinka ze struktury zdarzeń spowoduje znalezienie przecięcia. Jeśli zdarzenie jest początkiem odcinka, to aktualizujemy strukturę stanu w taki sposób, że nowy klucz staje się współrzędną x aktualnego zdarzenia. Kolejnym krokiem jest wykonanie procedury *add_line*, która wstawia odcinek do struktury stanu i sprawdza czy istnieje przecięcie między nią a jego sąsiadem ze struktury. Jeżeli zdarzenie jest końcem odcinka, to aktualizujemy strukturę stanu w taki sposób, że nowy klucz staje się współrzędną x aktualnego zdarzenia. Następnie wykonujemy procedurę *remove_line*, która na początku szuka ewentualnych przecięć pomiędzy aktualnie rozpatrywanym odcinkiem a jego sąsiadami w strukturze. Po wykonaniu tej operacji odcinek jest usuwany ze struktury stanu miotły. Ostatnim możliwym zdarzeniem jest przecięcie się odcinków. Procedura ta wykonywana jest w funkcji *state* wtedy, jeżeli nasze zdarzenie nie zachodzi. Procedura ta polega na dodaniu znalezionego przecięcia do zbioru przecięć i zmianie kolejności przecinających się odcinków w strukturze zdarzeń poprzez aktualizację klucza tej struktury (zwiększenie jego wartości o $\varepsilon = 10^{-12}$).

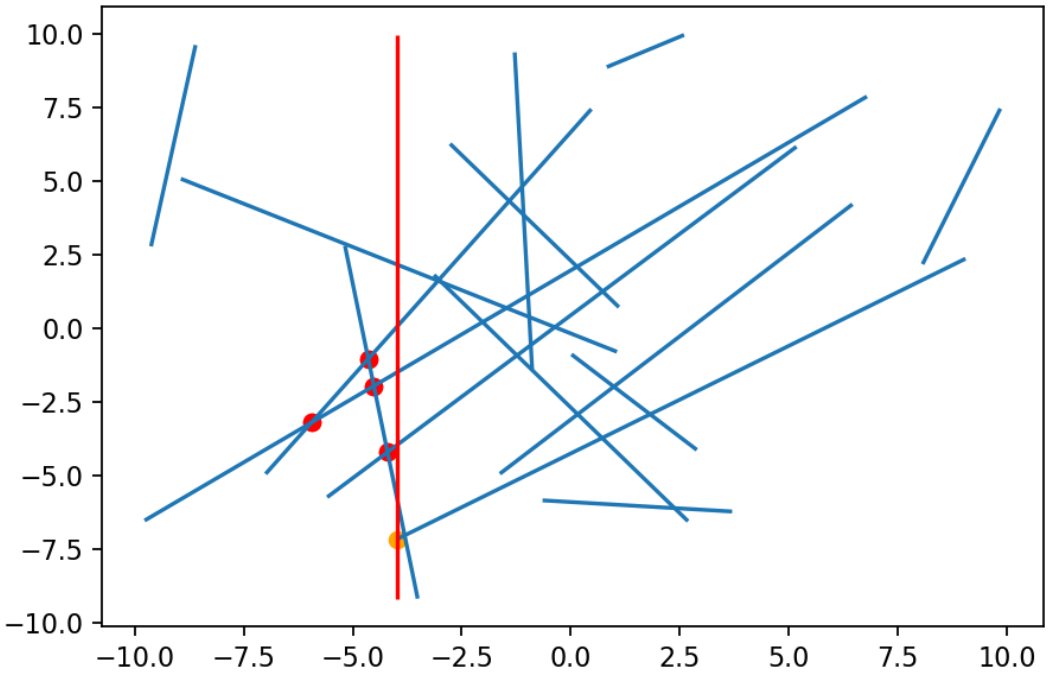
W implementacji zostały użyte również struktury (klasy) służące do przechowywania odcinków (*Line*) oraz punktów (*Point*). Przy ich implementacji konieczne było zaimplementowanie funkcji haszującej do prawidłowego identyfikowania instancji klasy.

Dodanie przecięcia w wizualizacji odbywa się wtedy, kiedy miotła dojdzie do punktu przecięcia się dwóch odcinków, które to przecięcie wcześniej znalazła.

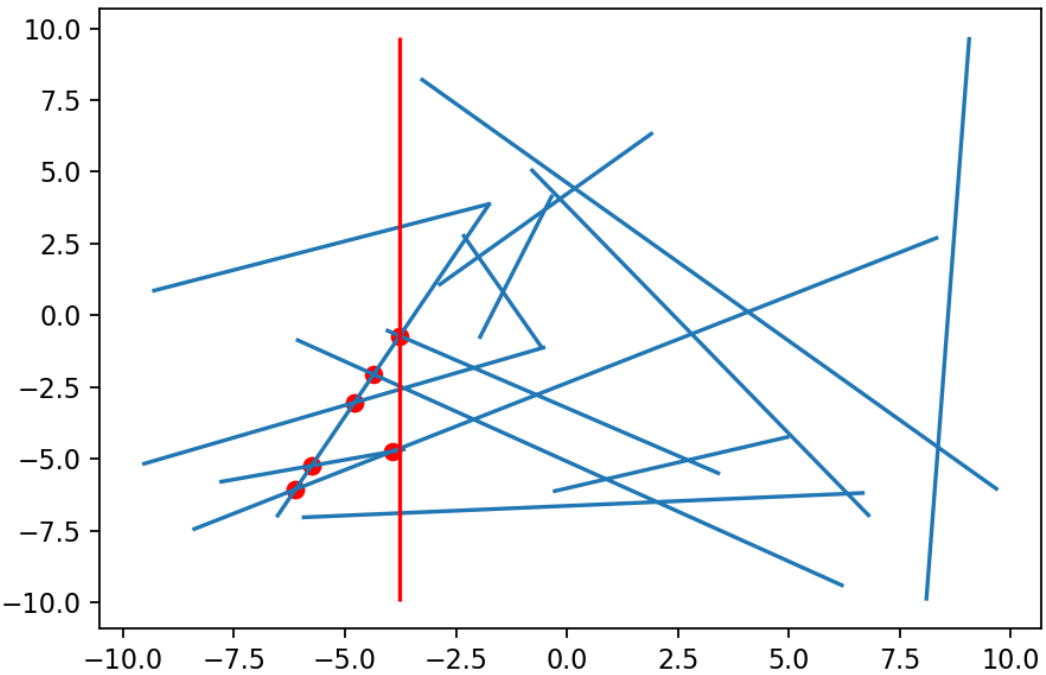
8. Działanie algorytmu

Przedstawiono wybrane kroki działania algorytmu wykrywania wszystkich przecięć odcinków (linii) na zbiorach.

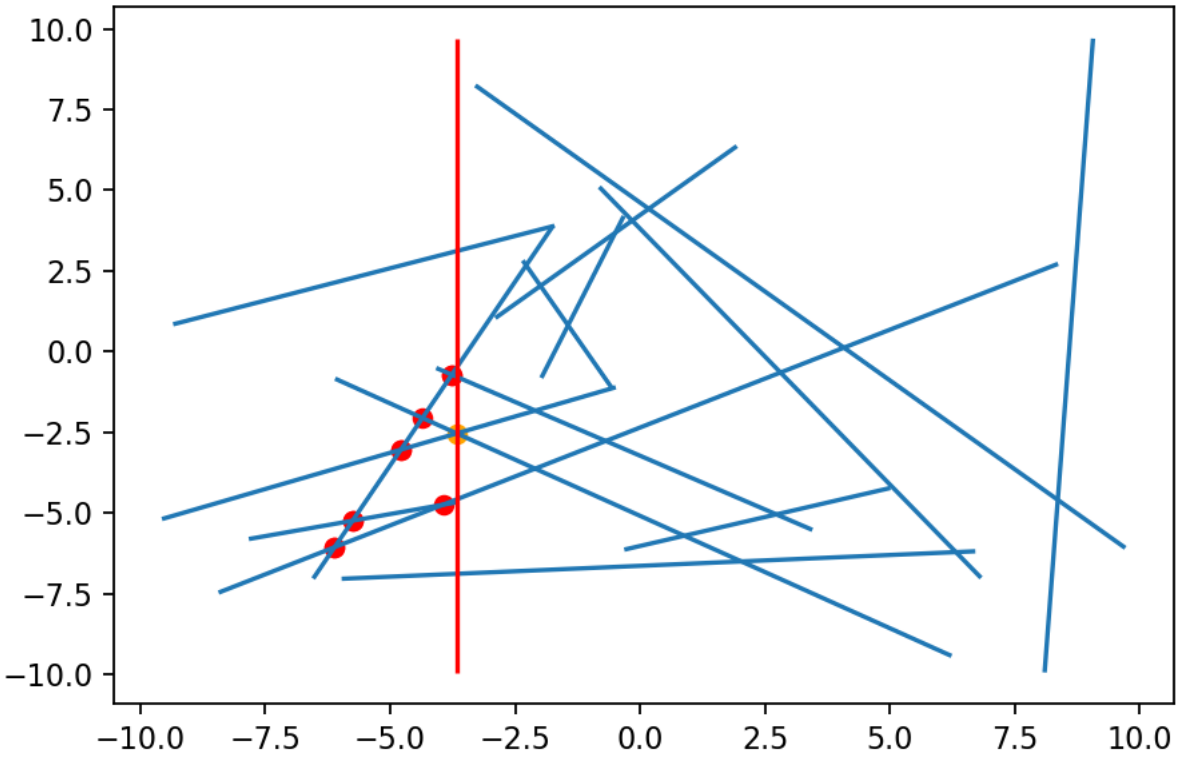
8.1. Losowy zbiór linii $\mathcal{ZR}$



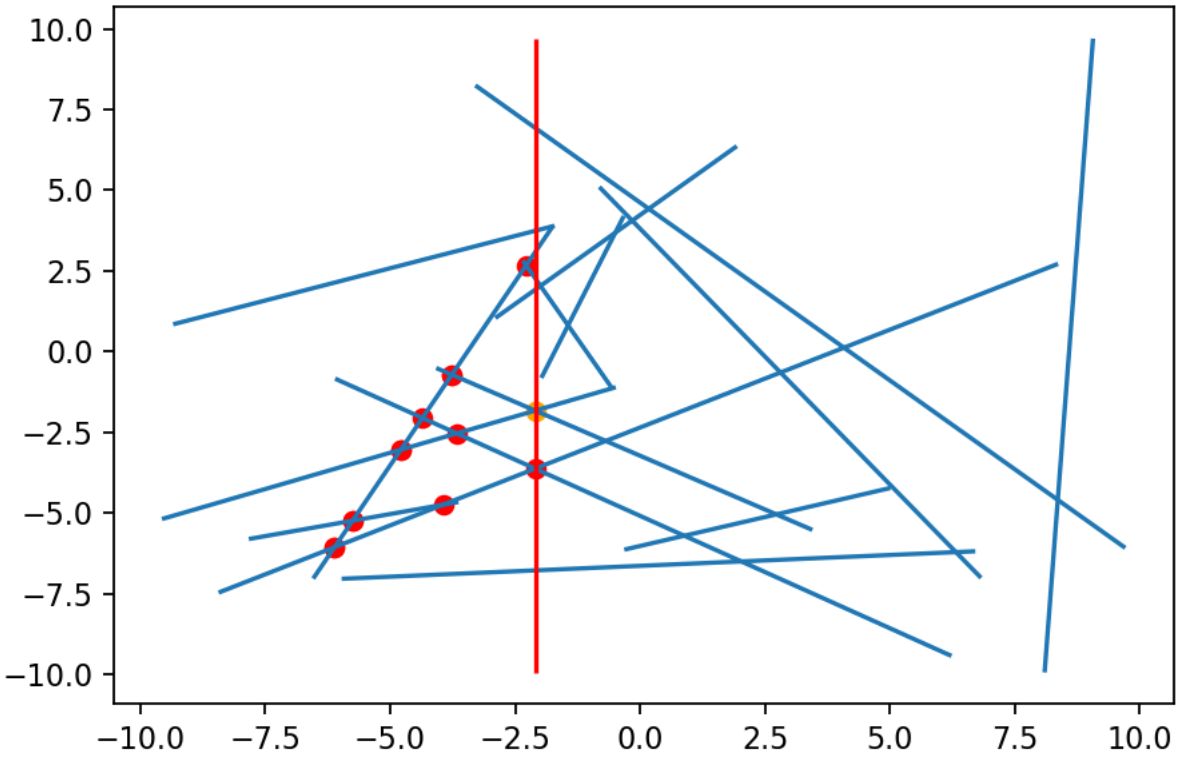
Rys. 8.1. a) Krok 1



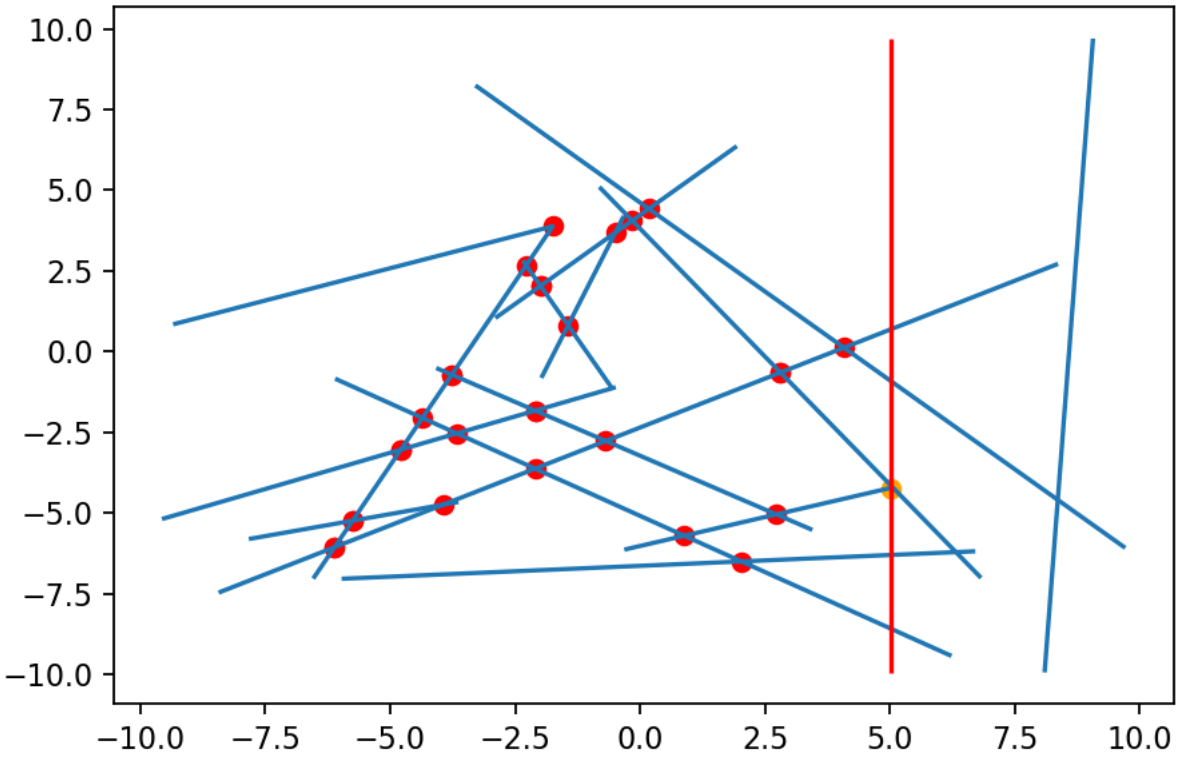
Rys. 8.1. b) Krok 2



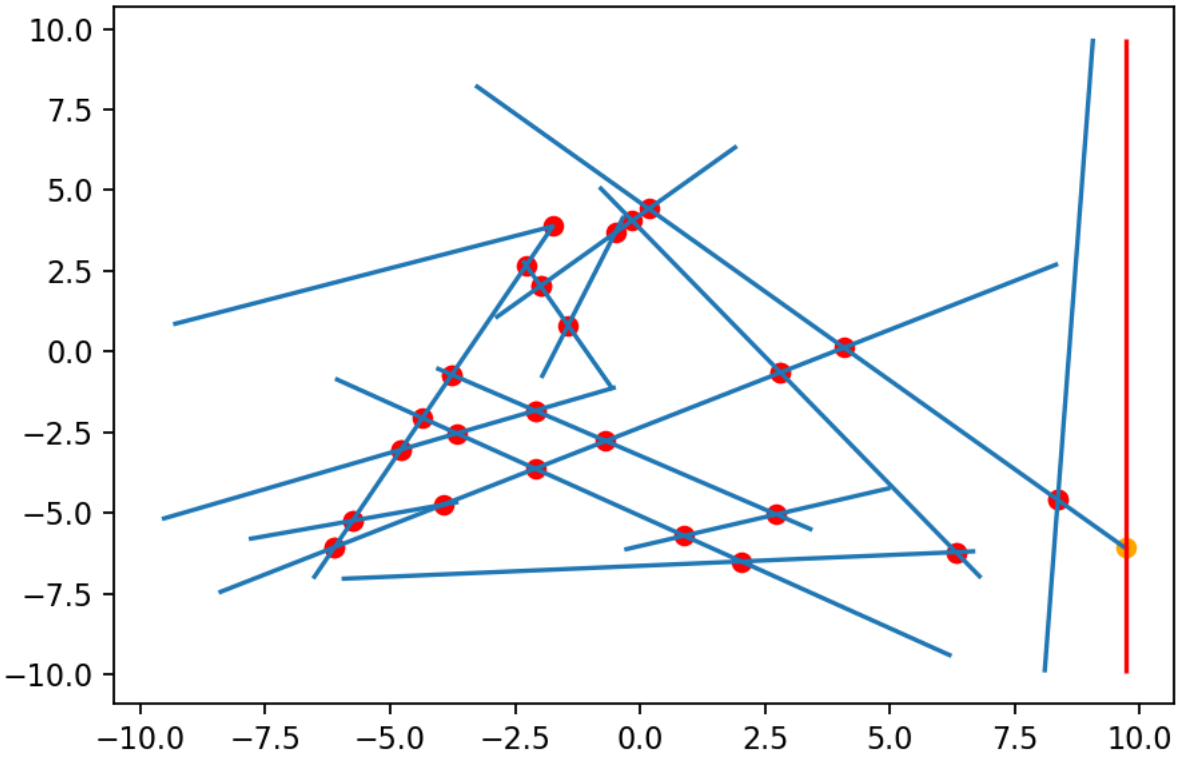
Rys. 8.1. c) Krok 3



Rys. 8.1. d) Krok 4

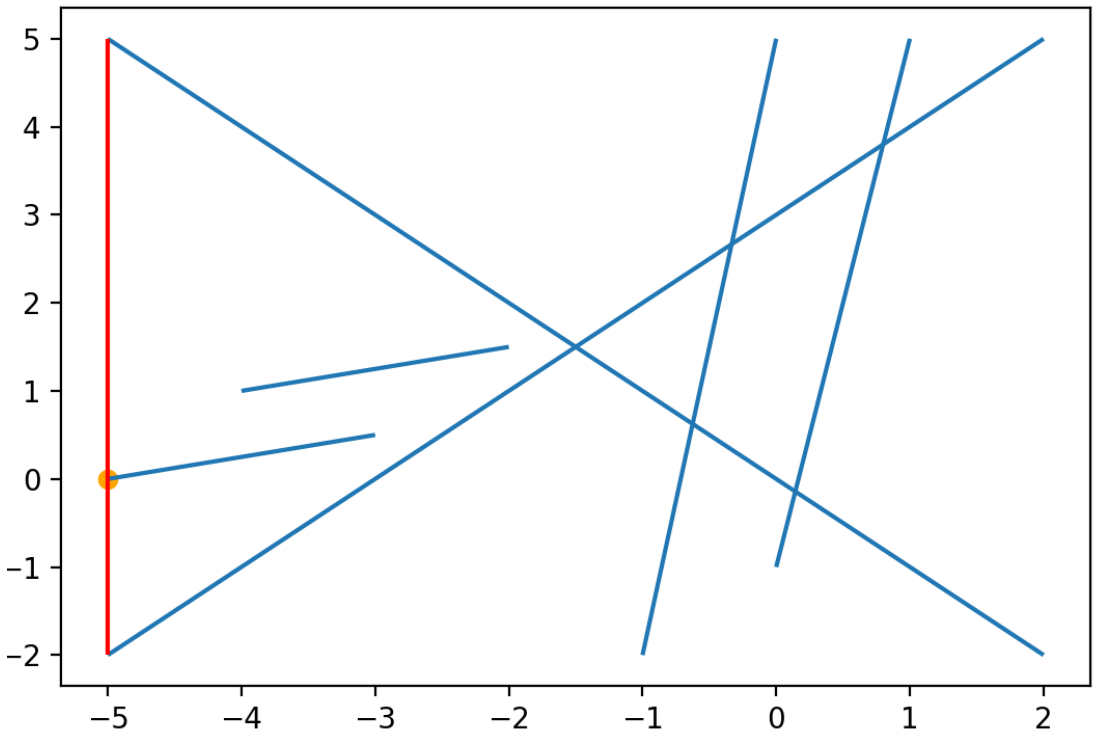


Rys. 8.1. e) Krok 5

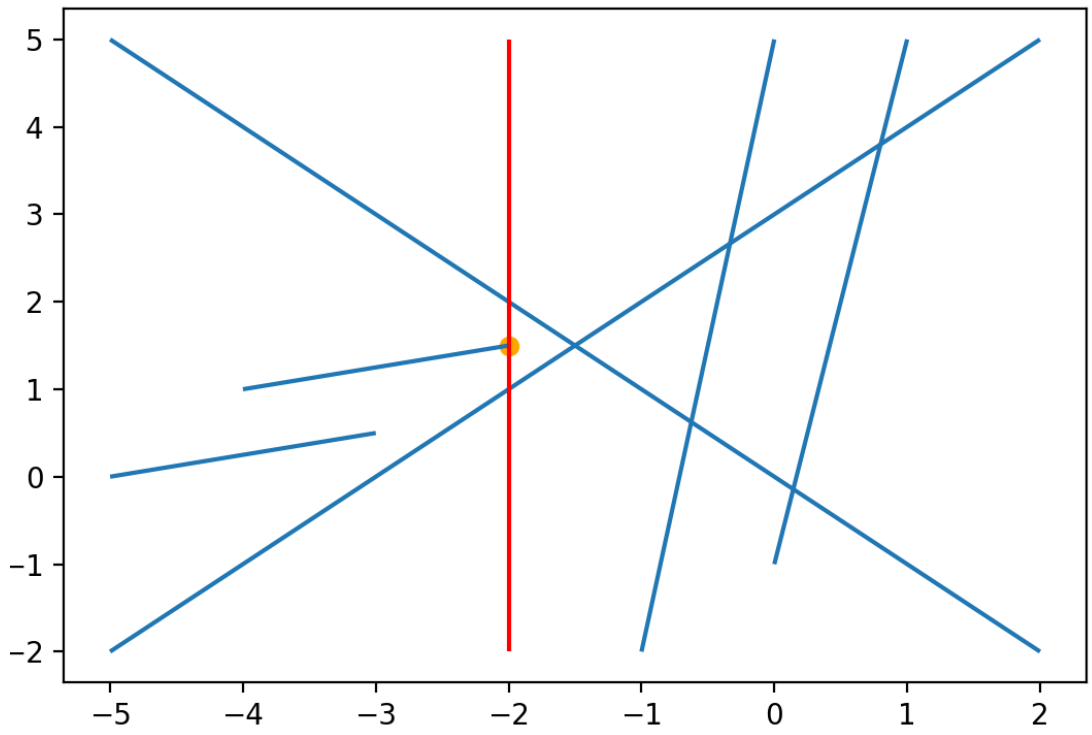


Rys. 8.1. f) Krok 6

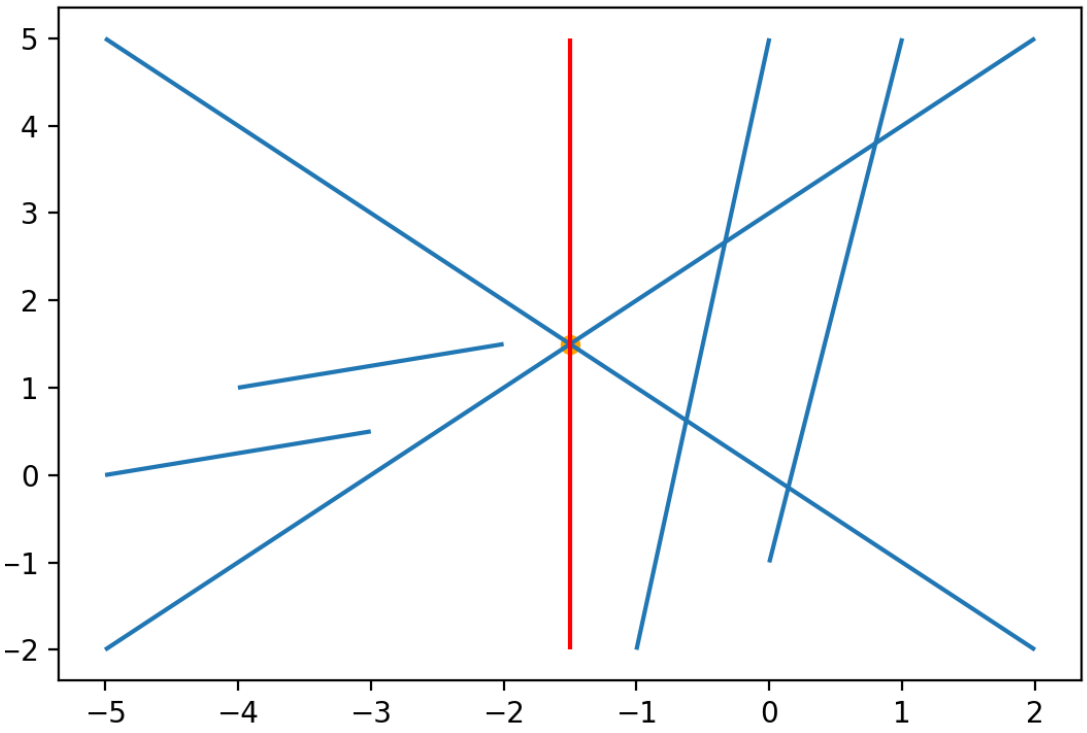
8.2. Testowy zbiór linii $\mathcal{Z}1$



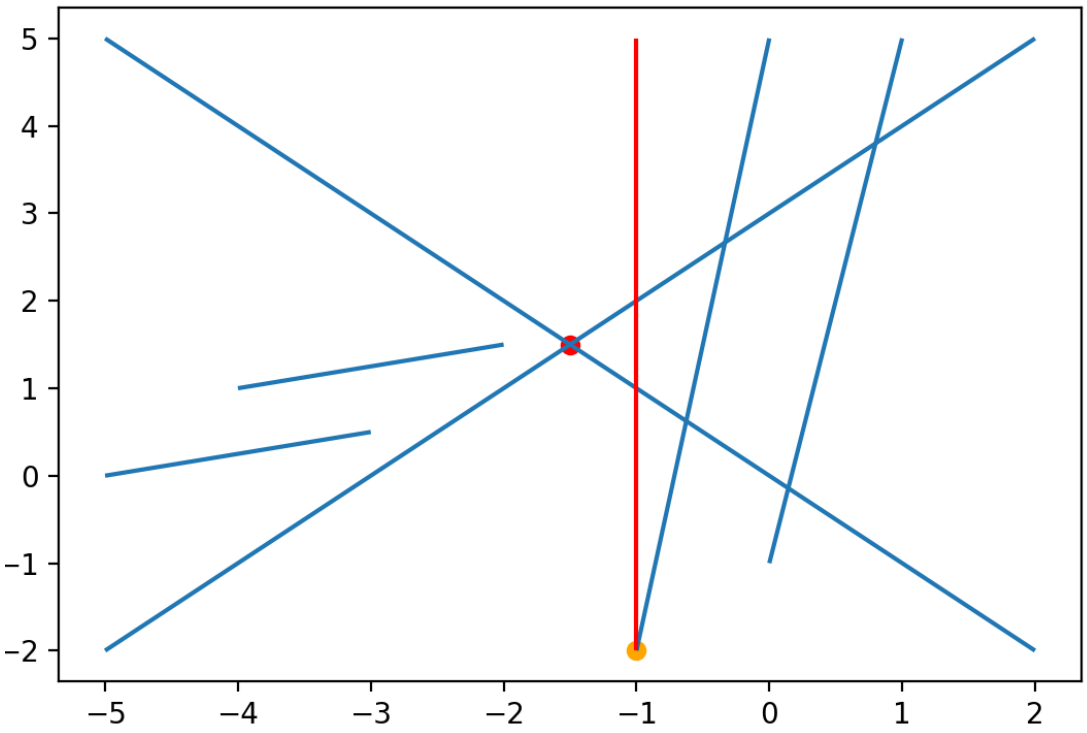
Rys. 8.2. a) Krok 1



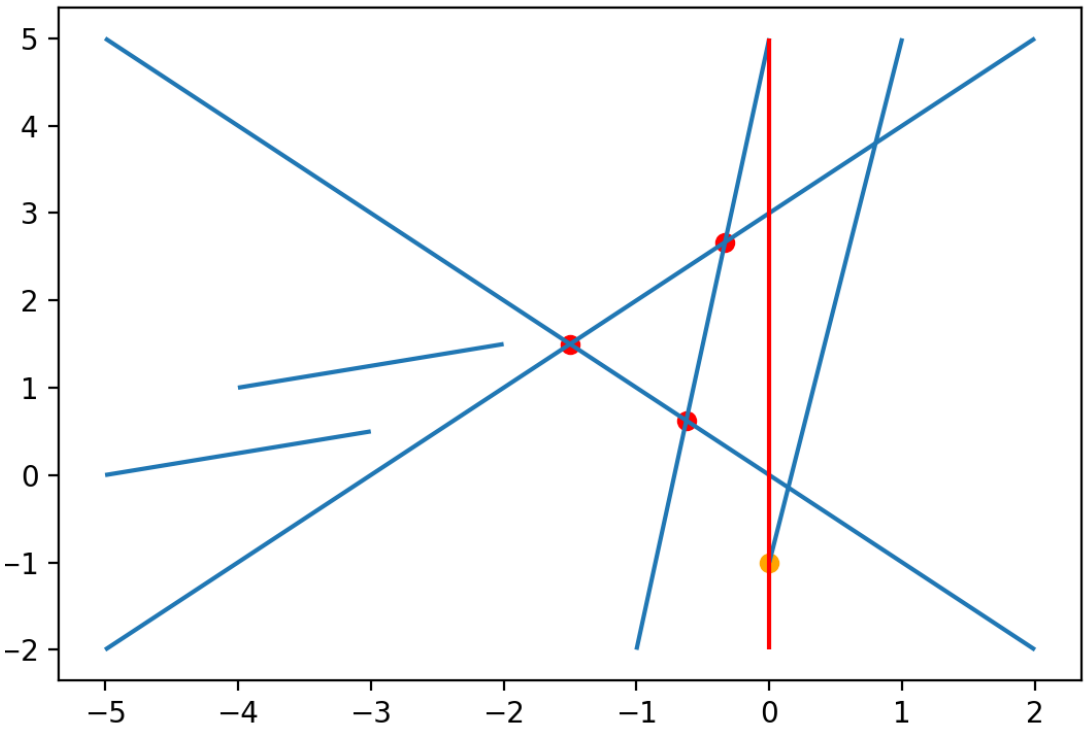
Rys. 8.2. b) Krok 2



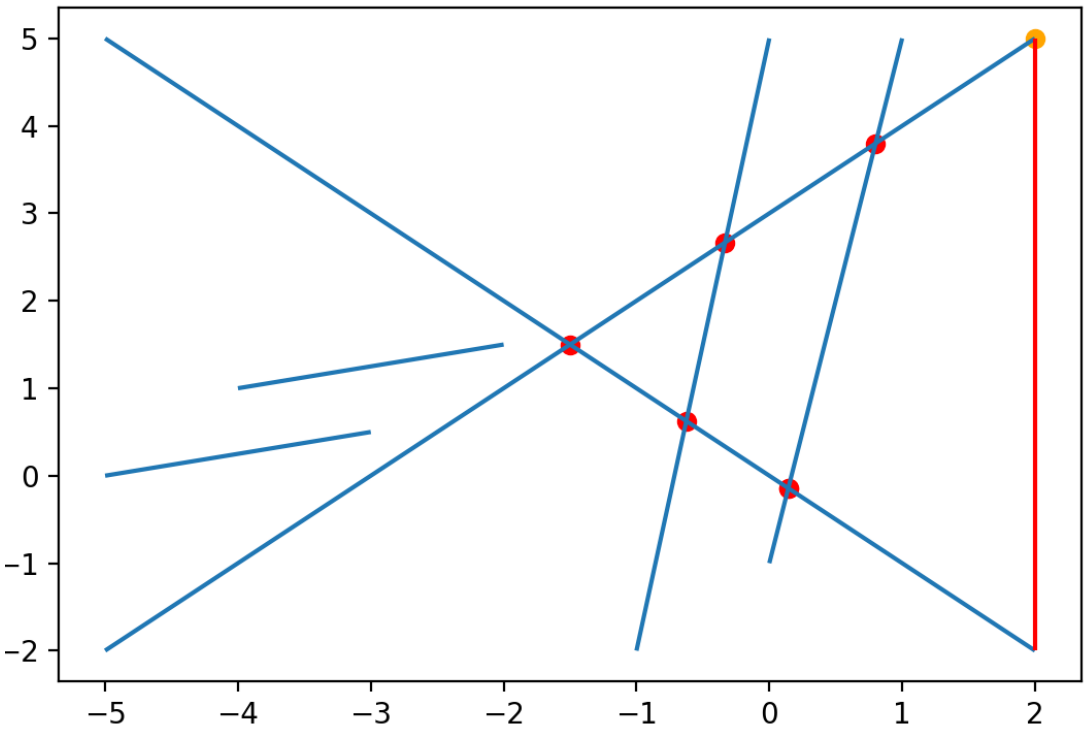
Rys. 8.2. c) Krok 3



Rys. 8.2. d) Krok 4

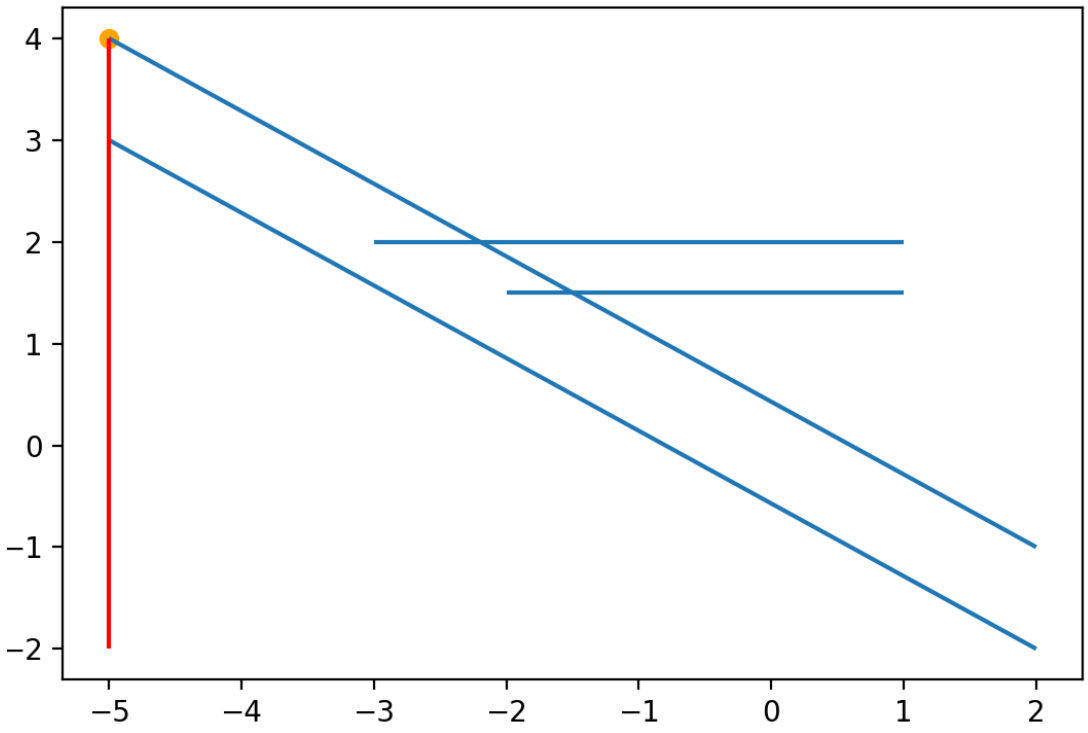


Rys. 8.2. e) Krok 5

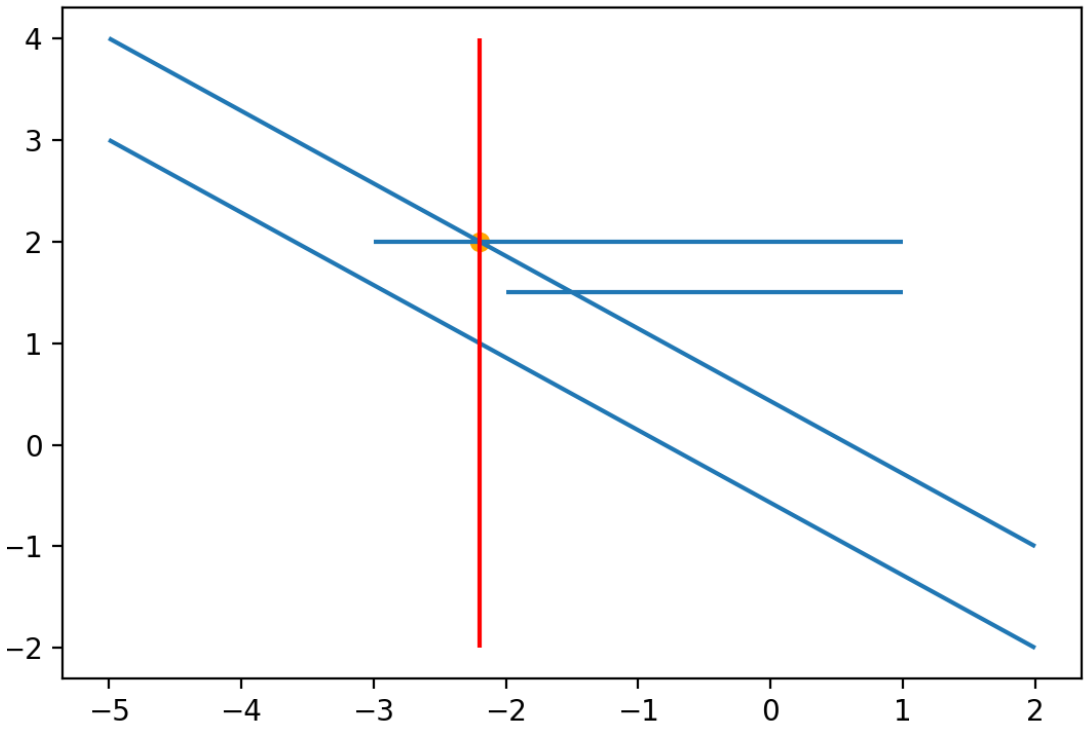


Rys. 8.2. f) Krok 6

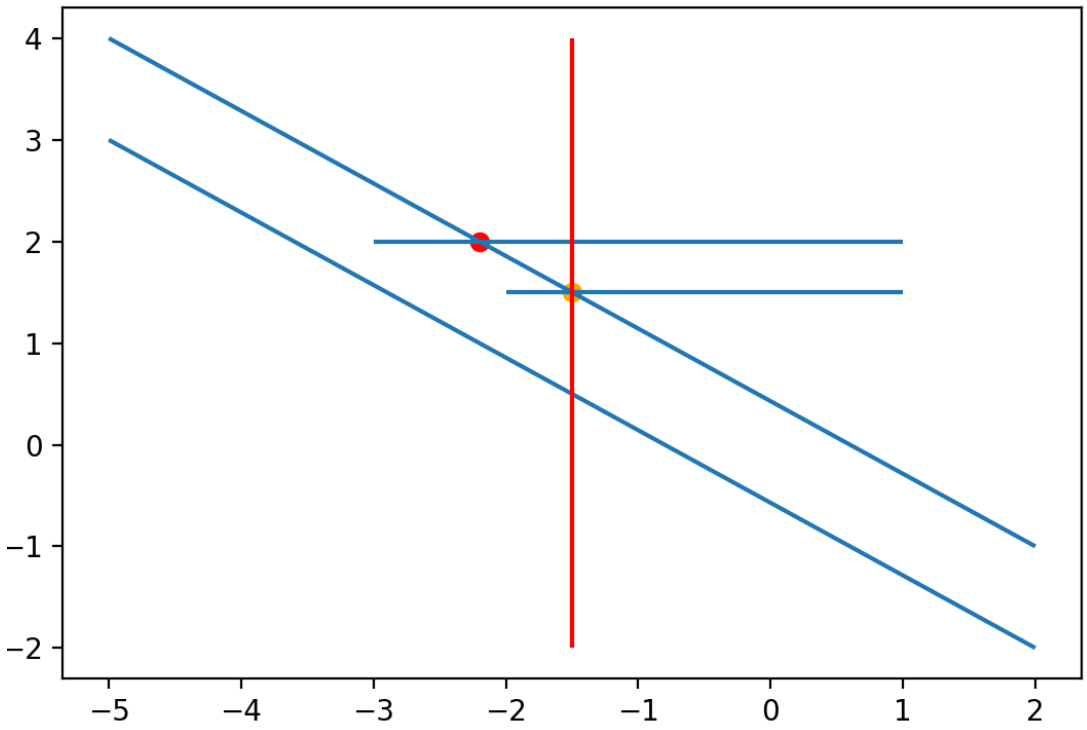
8.3. Testowy zbiór linii $\mathcal{Z}2$



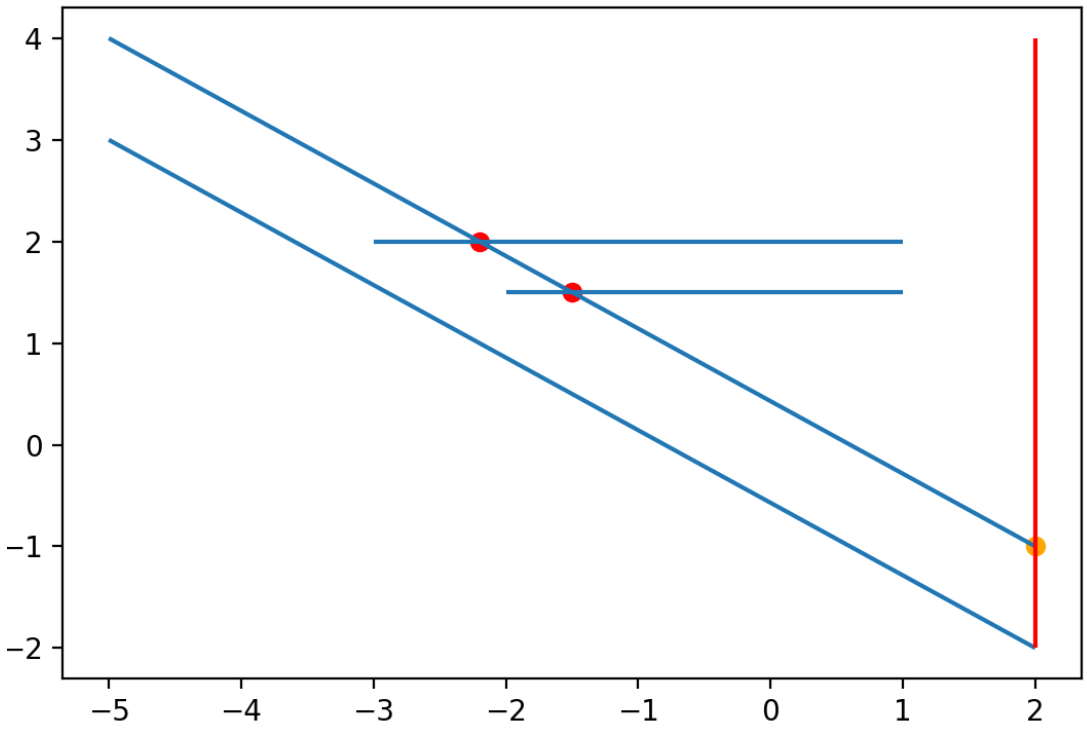
Rys. 8.3. a) Krok 1



Rys. 8.3. b) Krok 2



Rys. 8.3. c) Krok 3



Rys. 8.4. d) Krok 4

9. Wnioski i podsumowanie

Programy zostały przetestowane na różnych zbiorach danych, w szczególności uwagę skupiono na przypadkach brzegowych, w których sprawdzana była poprawność działania dodatkowych instrukcji sprawdzających dane (zbiory $Z1$ i $Z2$). Poprzez przeanalizowanie otrzymanych rezultatów można stwierdzić, że programy działają poprawnie w każdym przypadku.

Testowano także wykrywanie przez algorytm odcinków pionowych i o tej samej współrzędnej x by mieć pewność właściwego działania. Strukturami przechowującymi odcinki były kolekcje punktów przechowywane w listach. Pozwalało to na łatwą modyfikację danych, dodawanie i usuwanie odcinków. Umożliwiało to także łatwo generować odcinki oraz dawało dostęp do wszystkich elementów w każdym kroku. Pozwoliło to w pełni wykorzystać funkcjonalność list w języku *Python*.

Do zapisu i odczytu skorzystano z biblioteki *json*, która pozwalała na zapis i odczyt zadanych wielokątów w popularnym formacie JSON, który jest wszechstronny i wygodny w użyciu. Umożliwia to łatwą pracę z programem i wykorzystanie przetworzonych danych w innych projektach, a program jest przyjazny dla użytkownika i pozwala na implementację wybranych funkcjonalności w innych projektach.

Wnioski te są szczególnie cenne dla projektów informatycznych, w których precyzja obliczeń i identyfikacja pewnych danych z wykorzystaniem algorytmów geometrycznych są sprawą kluczową.

* * *