Szymon Budziak
Informatyka II rok, grupa poniedziałek 14:40 - 16:10

Algorytmy geometryczne, laboratorium 4 - sprawozdanie

1.Opis ćwiczenia

Zadaniem które należy wykonać na laboratorium 4 jest implementacja oraz przetestowanie działania algorytmu zamiatania, który wyznacza punkty przecięcia odcinków na płaszczyźnie. Należało zaimplementować:

- funkcję umożliwiającą generowanie w losowy sposób zadanej ilości odcinków we współrzędnych 2D,
- algorytm zamiatania sprawdzający, czy dowolne dwie pary odcinków w zadanym zbiorze przecinają się,
- algorytm wyznaczający wszystkie przecięcia odcinków w zbiorze, zwrócenie liczby wszystkich punktów przecięcia, ich współrzędne oraz odcinki, które się przecinają.

2. Środowisko, biblioteki oraz użyte narzędzia

Ćwiczenie zostało wykonane w Jupyter Notebook i napisane w języku Python. Do rysowania wykresów zostało użyte dostarczone na laboratorium narzędzie graficzne, które oparte jest o bibliotekę matplotlib. Umożliwiało to również wspomniane w opisie ćwiczenia zadawanie odcinków przy pomocy myszki, ich zapis oraz odczyt. Wszystko było wykonywane na systemie operacyjnym Linux Ubuntu 20.04 oraz na procesorze Intel Core i5-7300HQ 2.50GHz.

3. Plan, sposób wykonania ćwiczenia, opisy algorytmów oraz wizualizacja ich działania na przykładach

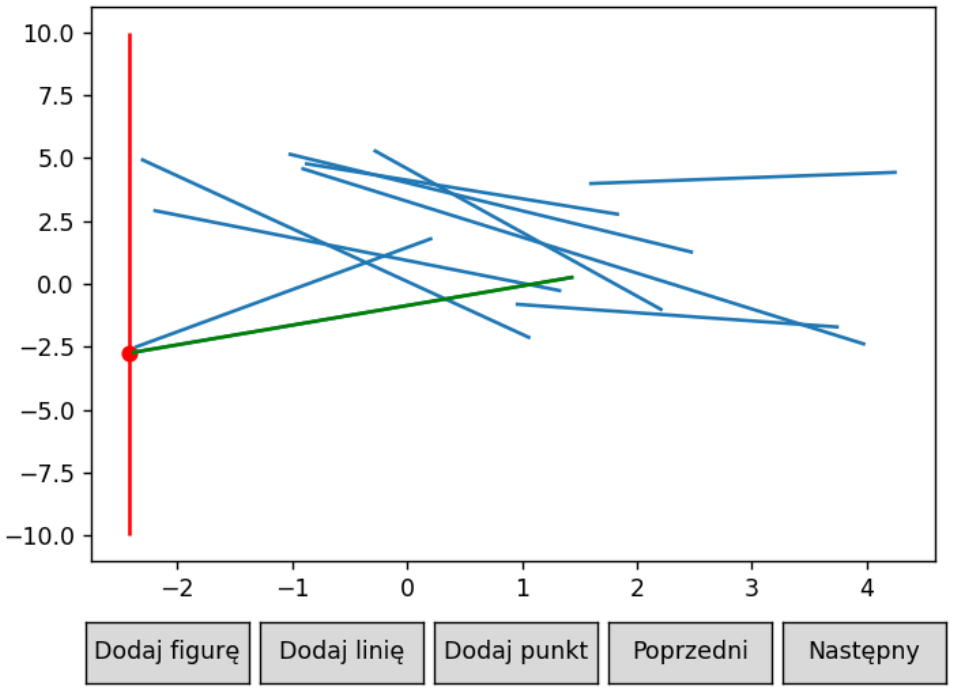
3.1 Algorytm sprawdzający czy dowolne dwie pary odcinków w zadanym zbiorze przecinają się

Pierwszym algorytmem, który należało zaimplementować jest algorytm sprawdzający czy czy dowolne dwie pary odcinków w zadanym zbiorze przecinają się. W implementacji tej została użyta struktura SortedSet która w Pythonie występuje w bibliotece sortedcontainers. Klasa ta jest reprezentowana przez drzewo czerwono-czarne, dzięki czemu możliwe jest wstawianie, usuwanie oraz znajdowanie poprzednika/ następnika elementu w czasie O(logn).

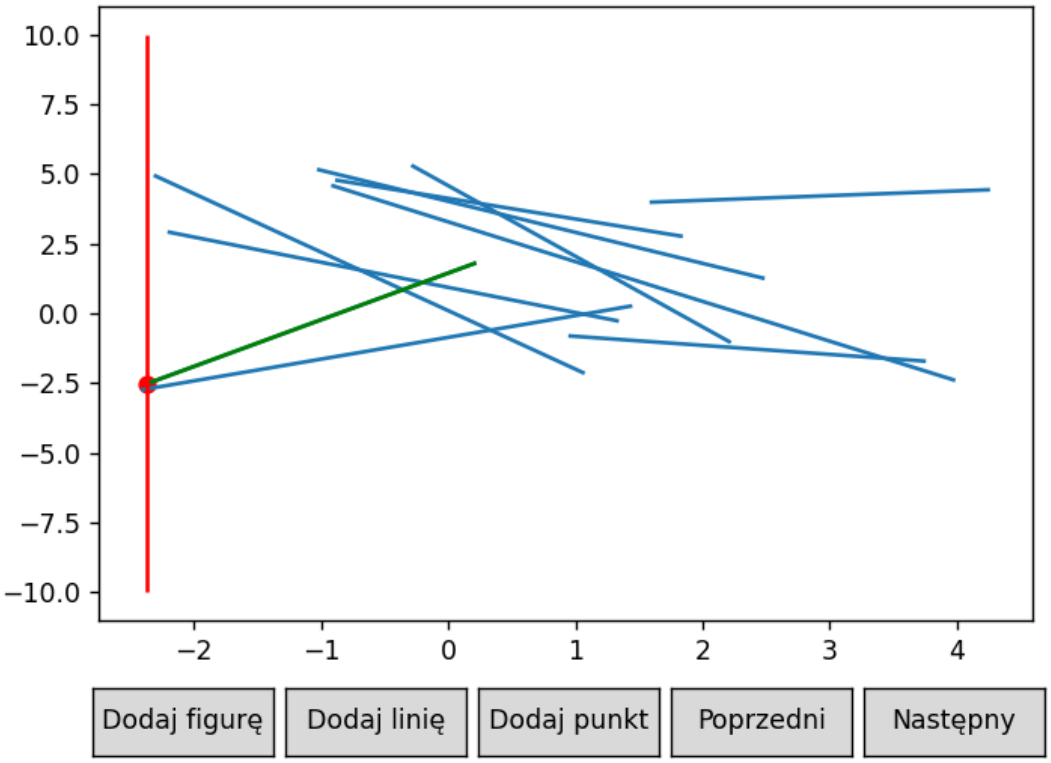
Struktura ta została użyta jako struktura stanu miotły, która przechowywuje posortowane odcinki względem współrzędnej y. Do struktury zdarzeń została użyta struktura z biblioteki heapq na której wywoływane są funkcje heapify oraz heapop. Struktura ta reprezentuje kolejkę priorytetową typu min, gdzie 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 nowe zdarzenie, jeżeli algorytm wykryje przecięcie się dwóch odcinków.

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 wykorzystując metodę add_line dodaję do struktury stanu nowy odcinek, a następnie sprawdzam 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 usunięcie tego odcinka ze struktury. Jeżeli natomiast jest to zdarzenie przecięcia się odcinków (występuje ono podczas procedur add_line oraz remove_line) to program przestaje działać, ponieważ znalazł dwa dowolne odcinki które się przecinają.
Przykład działania algorytmu:

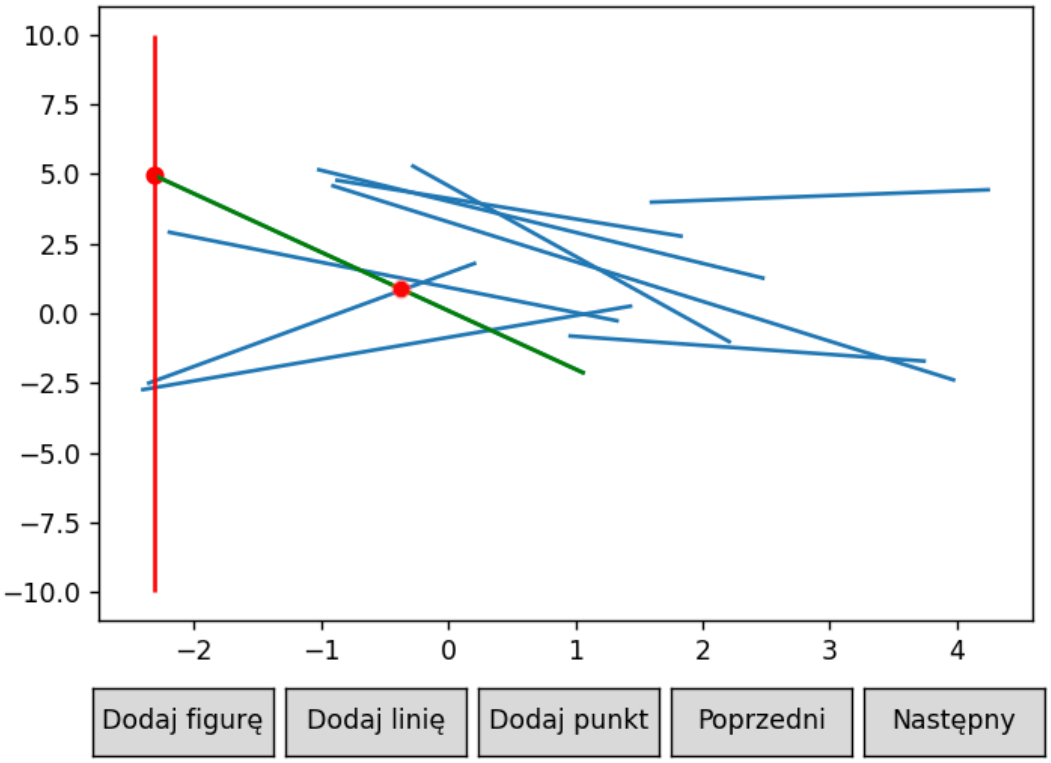
Wykres_1: Początek działania algorytmu (miotła jest koloru czerwonego, aktualnie rozpatrywana linia koloru zielonego)



Wykres_2: Działanie algorytmu po kolejnym kroku (miotła jest koloru czerwonego, aktualnie rozpatrywana linia koloru zielonego)



Wykres_3: Ostatni krok algorytmu, znalezienie przecięcia (miotła jest koloru czerwonego, aktualnie rozpatrywana linia koloru zielonego, znalezione przecięcie jest zaznaczone czerwoną kropką)

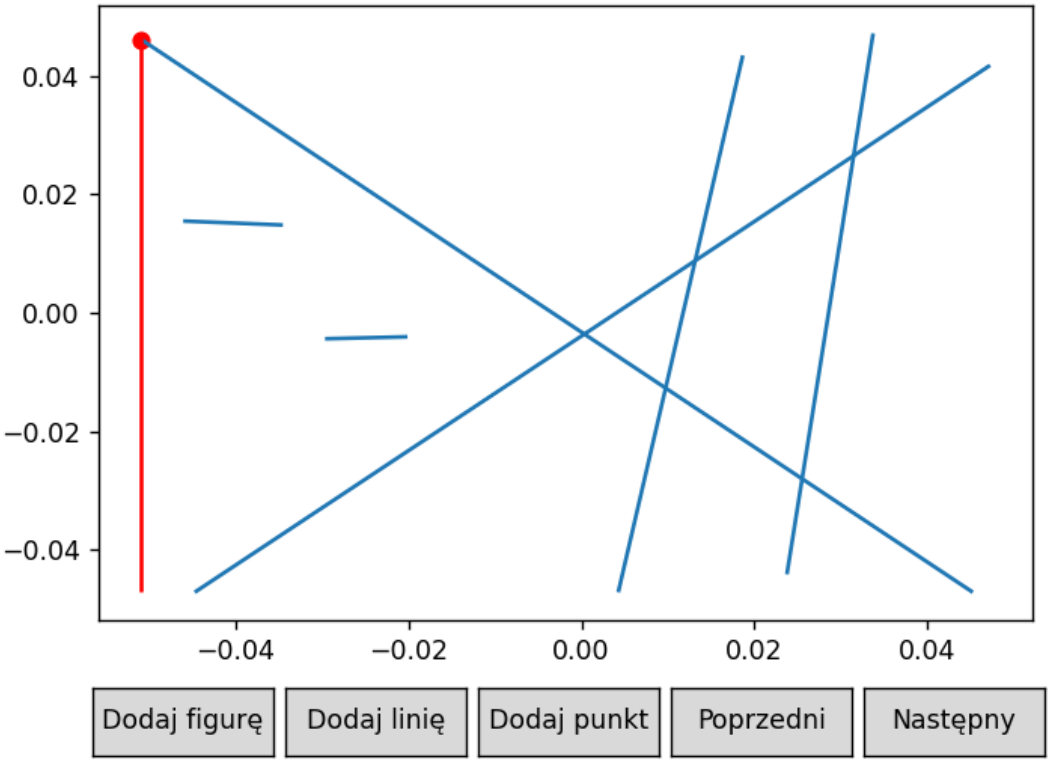


3.2 Algorytm wyznaczający wszystkie przecięcia odcinków w zbiorze

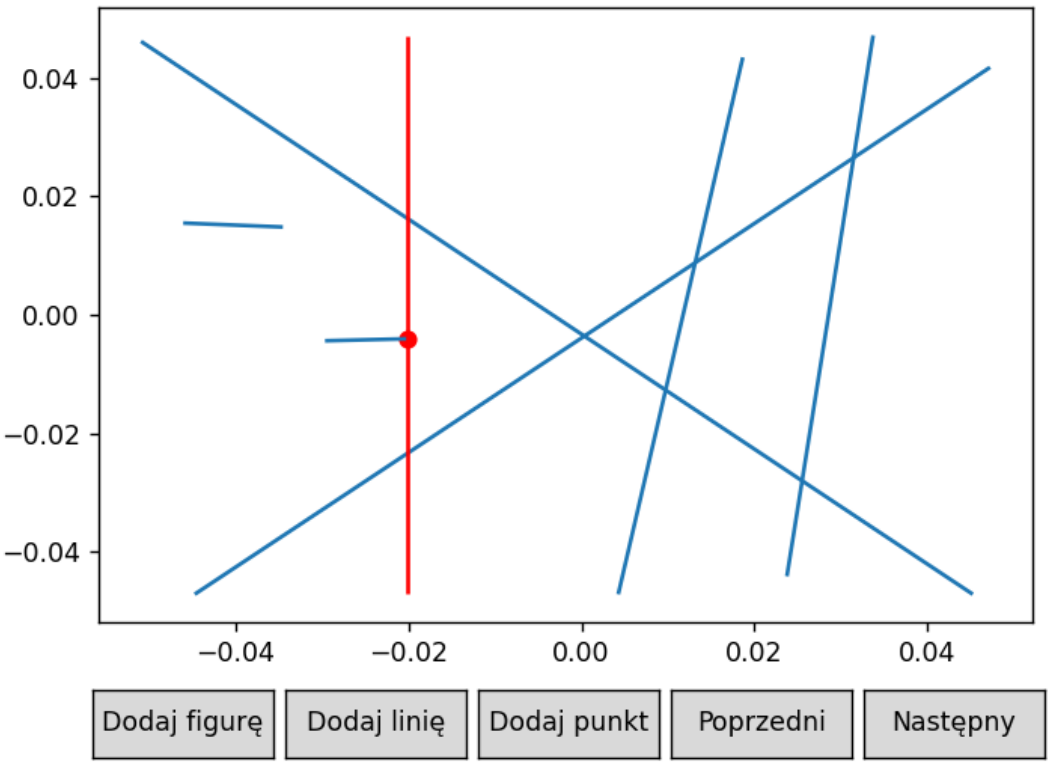
Algorytm opiera się na bardzo podobnej procedurze co algorytm sprawdzający czy dowolne dwie pary odcinków w zadanym zbiorze przecinają się. Struktura stanu miotły jest również zaimplementowana w taki sam sposób, czyli poprzez strukturę SortedSet z biblioteki sortedcontainers. Jednak w tym przypadku, 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 epsilon= 10**(-12)).
W implementacji zostały użyte również struktury służące do przechowywania odcinków (Line) oraz punktów (Point). Przy ich implementacji konieczne było zaimplementowanie funkcji hasz. Dodanie

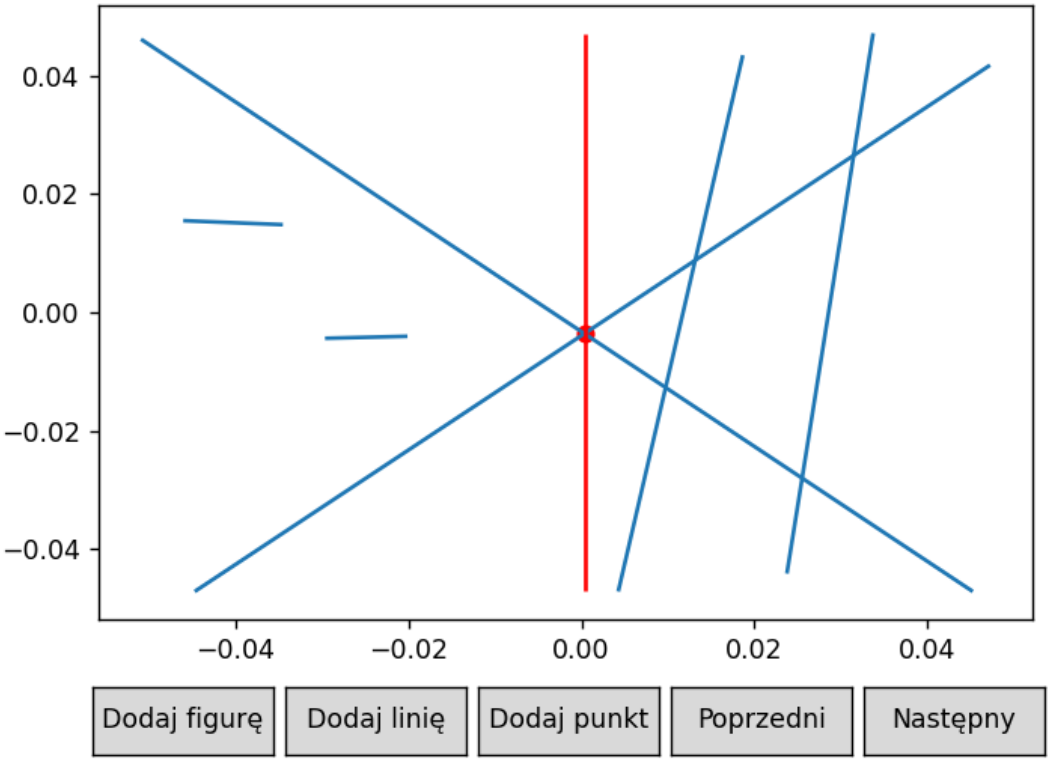
przecięcia w wizualizacji odbywa się wtedy, kiedy miotła dojdzie do przecięcia które wcześniej znalazła.
Przykład działania algorytmu:
Wykres_4: Początek działania algorytmu (miotła jest koloru czerwonego)



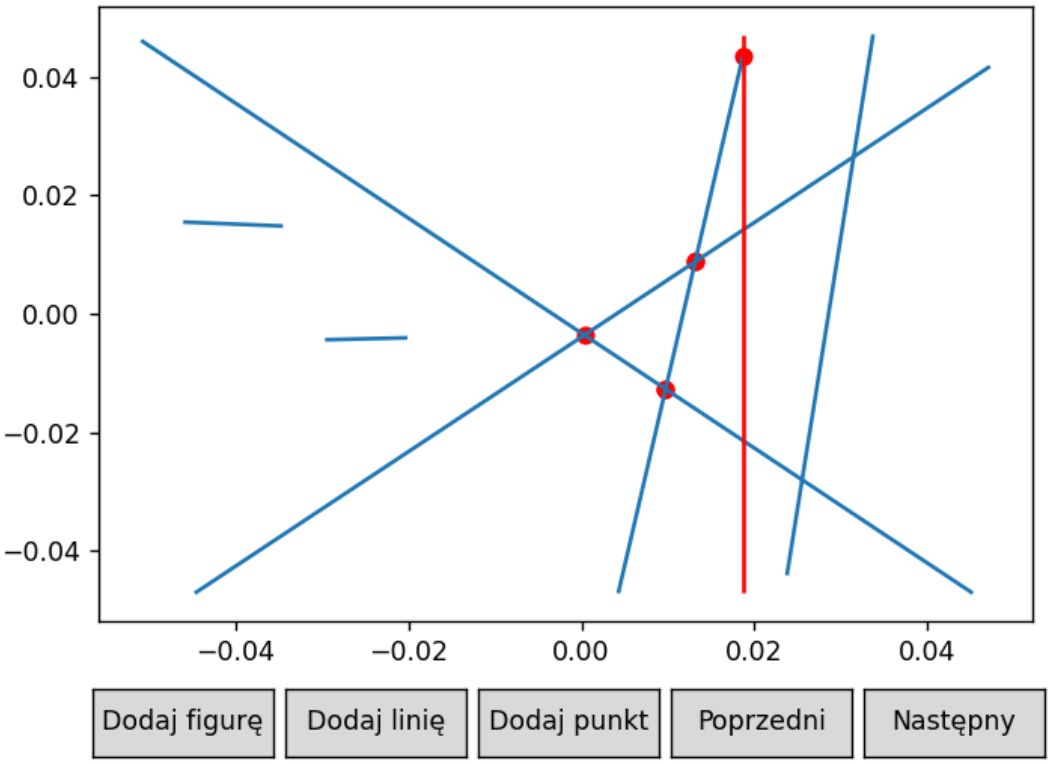
Wykres_5: Działanie algorytmu po kolejnym kroku (miotła jest koloru czerwonego)



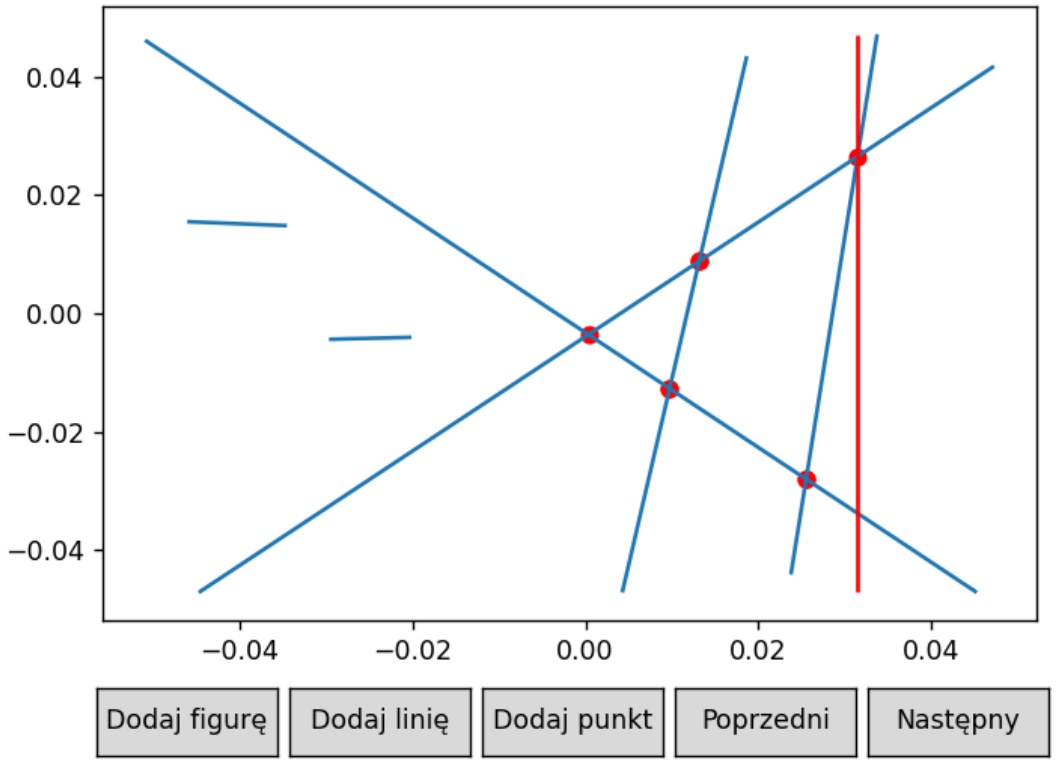
Wykres_6: Działanie algorytmu po kolejnym kroku, dodanie pierwszego przecięcia (miotła jest koloru czerwonego, punkt przecięcia jest koloru czerwonego)



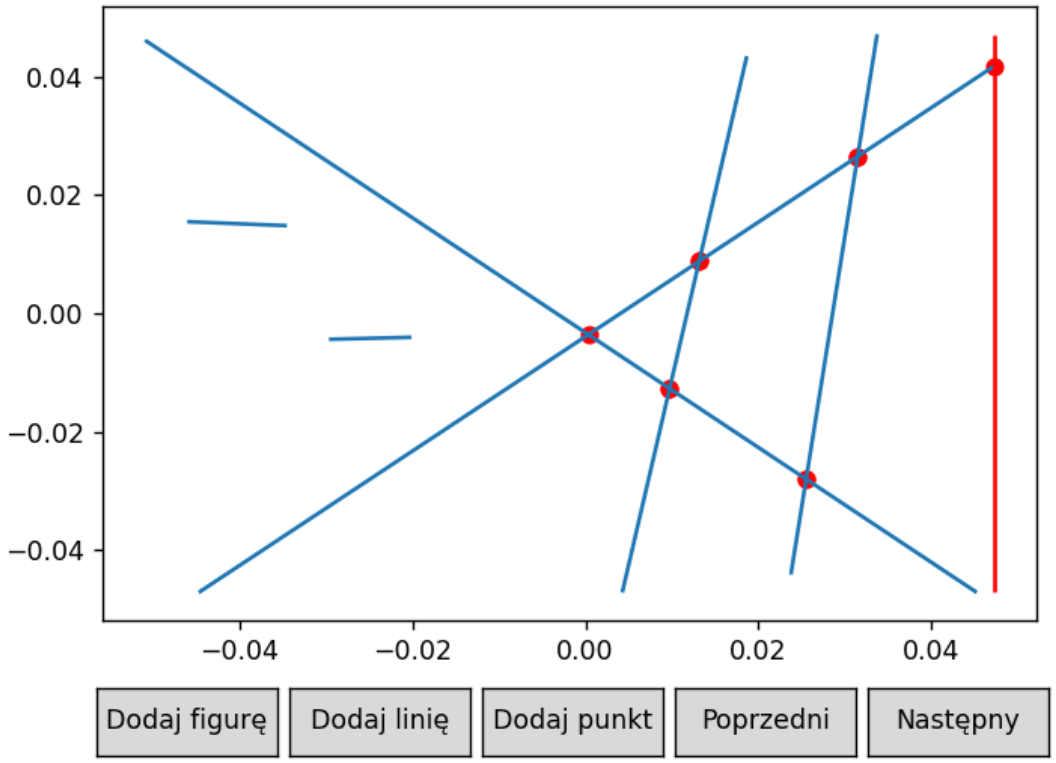
Wykres_7: Działanie algorytmu po kolejnym kroku, dodanie kolejnych przecięć (miotła jest koloru czerwonego, punkty przecięcia są koloru czerwonego)



Wykres_8: Działanie algorytmu po kolejnym kroku, dodanie kolejnych przecięć (miotła jest koloru czerwonego, punkty przecięcia są koloru czerwonego)



Wykres_9: Ostatni krok algorytmu, przejście po wszystkich odcinkach i znalezienie wszystkich przecięć(miotła jest koloru czerwonego, punkty przecięcia są koloru czerwonego)



4. Wnioski

Obydwa algorytmy po ich przetestowaniu na powyższych zbiorach (oraz innch, które są w jupyter notebook) działają poprawnie i nie zauważono w nich żadnych błędów w działaniu. Można zatem stwierdzić, że są one zaimplementowane i działają w sposób poprawny. Struktury stanu oraz zdarzeń pozwalają na uzyskanie dobrej złożoności całego algorytmu. Dzięki zastosowaniu SortedSet możemy w łatwy sposób uzyskać dostęp do kolejnych elementów. W przypadku drugiego algorytmu, użycie struktury SortedSet pozwoliło na wyeliminowanie wielokrotnego wykrywania tego samego punktu.