Algorytmy geometryczne
Sprawozdanie z laboratorium 4.

Michał Nożkiewicz
gr. Czw_11:20_B

1. Użyte narzędzia i oprogramowanie

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

Kod programu pisałem w środowisku Jupyter notebook, w języku Python, z wykorzystaniem bibliotek *numpy, random, matplotlib, queue, sortedcontainers* oraz *functools.*

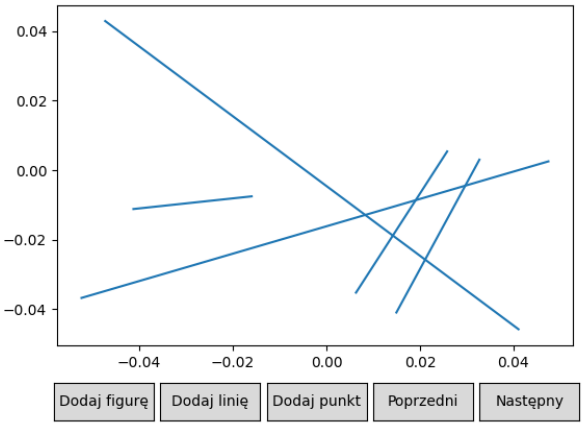
2. Opis ćwiczenia

Zadaniem tego laboratorium było zaimplementowanie algorytmu stwierdzającego czy zadanym zbiorze odcinków znanjdują się dwa, które się przecinają, a także algorytmu, który znajduje wszystkie przecięcia dla zadanego zbioru punktów.

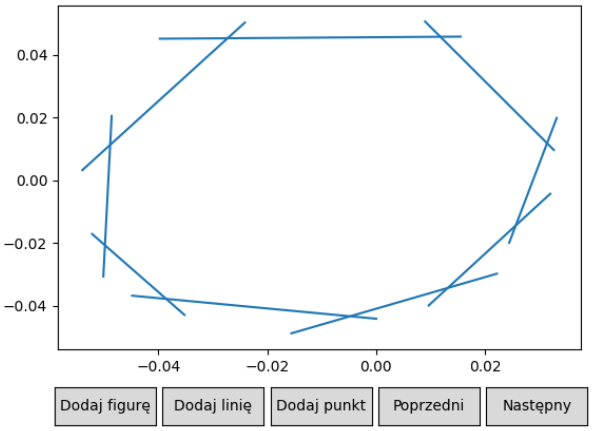
3. Zbiory testowe

W zadaniu zaimplementowałem możliwość zadawaniu własnych zbiorów odcinków, a także generowanie losowej ilości odcinków. Zadawane zbiory były zapisywane do plików formatu json. Funkcja generująca odcinki jako parametry przyjmowała ilość odcinków do wygenerowania, a także przedziały z jakich powinny być losowane współrzędne punktów będących końcami odcinków. Punkty były losowane z rozkładu jednostajnego na zadanym przedziale.
Poniżej znajduje się zestawienie zbiorów, które wprowadziłem. Wśród nich znajduje się jeden zbiór w którym nie ma żadnych przecięć, a także taki w którym kilkukrotnie pod kątem przecięcia mogą być sprawdzane dwa odcinki.

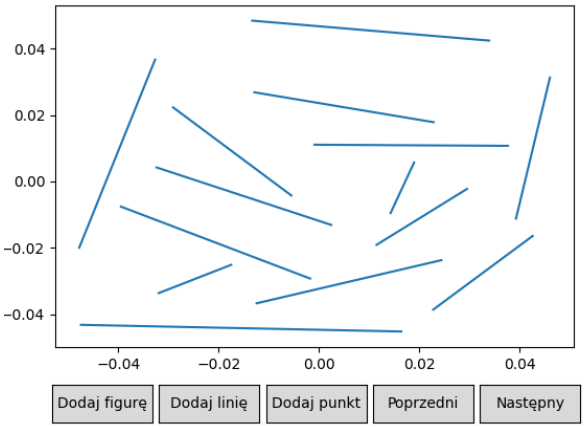
Wykres 3.1 Zbiór A



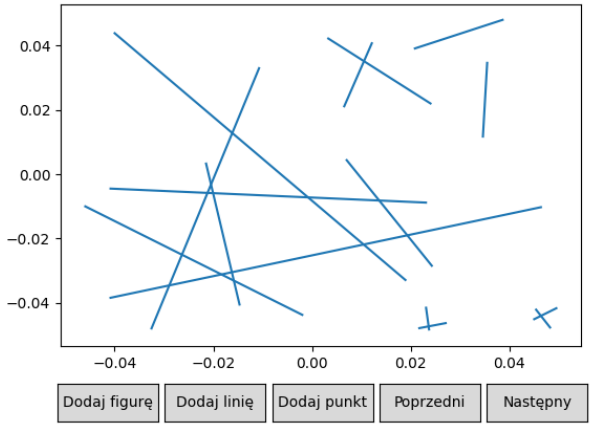
Wykres 3.2 Zbiór B



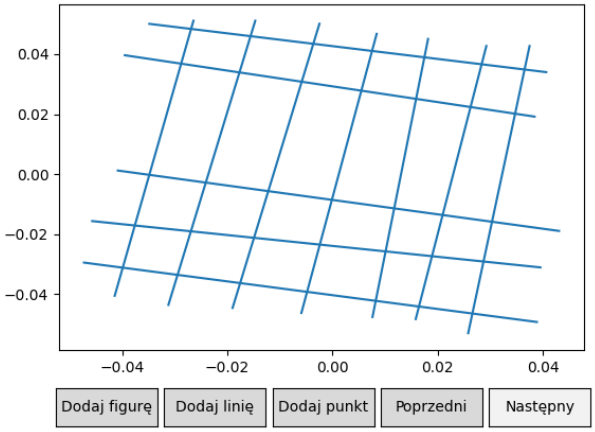
Wykres 3.3 Zbiór C



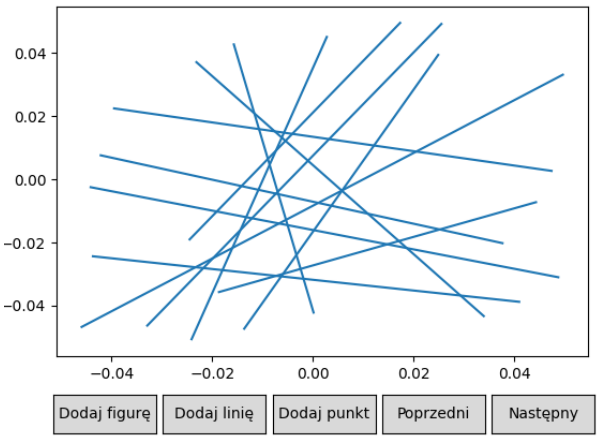
Wykres 3.1 Zbiór D



Wykres 3.1 Zbiór E



Wykres 3.1 Zbiór F



4. Implementacja algorytmów

Zarówno algorytm sprawdzający czy istnieje choć jedna para przecinających się odcinków jak i ten znajdujący wszystkie przecięcia opierają się na technice zamiatania. W obu algorytmach potrzebna będzie struktura zdarzeń. Będzie ona przechowywać kolejne zdarzenia, które należy rozważać w trakcie „zamiatania". W przypadku algorytmu stwierdzającego czy istnieje choć jedno przecięcie będą to końce i początki wszystkich odcinków, a w algorytmie znajdującym wszystkie przecięcia będą to także znalezione punkty przecięć. W przypadku napotkaniu początku odcinka będą go wrzucać do struktury stanu, a przy napotkaniu końca z niej wyrzucać. Struktura stanu będzie przechowywać odcinki posortowane według współrzędnej y. (współrzędna y jest to wartość jaką daje funkcja liniowa opisująca dany odcinka dla współrzędnej x, która jest rozważana w danym zdarzeniu). Przy szukaniu ewentualnych przecięć wystarczy wtedy porównywać odcinki sąsiadujace ze sobą, a dokładniej pary sąsiadów którzy pojawili się przy wrzucaniu lub wyrzucaniu nowego odcnika do struktury stanu. Można zauważyć, że tak zdefiniowany porządek może doprowadzić do sytuacji, że dla pewnych dwóch odcinków A i B najpierw w pewnym zdarzeniu zachodzi A > B, a w kolejnym B > A. Jednak jeśli najpierw odcinek A znajduje się nad odcinkiem B, a potem na odwrót i oba odcinki dalej znajdują się w strukturze to znaczy, że w pewnym momencie się przecinają. Dlatego przy analizie zdarzenia, które jest przecięciem dwóch odcinków, należy je zamienić miejscami w strukturze stanu, a także sprawdzić czy nowopowstałe pary sąsiadów się nie przecinają.

Dodatkowo w implementacji obu algorytmów użyłem klasy Line. Obiekt tej klasy tworzony jest dla każdego odcinka. Klasa ta przechowuje współrzędne odcinka, współczynniki prostej na której leży, a także metodę, która dla danej współrzędnej x oblicza wartość y tej prostej. Ponadto klasa ta posiada metodę find_intersect, która przyjmuje inny obiekt klasy Linia i sprawdza czy dwa odcinki się przecinają i w przypadku znalezienia przecięcia zwraca ten punkt, a w przeciwnym razie None.

4.1 Algorytm stwierdzający czy istnieje przynajmiej jedno przecięcie

W tym algorytmie w przypadku znalezienie przecięcia, w momencie znalezienia przecięcia algorytm powinien zwrócić True i zakończyć działanie, więc jedyne zdarzenia jakie wystarczy rozważać to początki i końce odcinków. Wszystkie zdarzenia, które będą rozważane są znane już na początku, więc strukturą zdarzeń może być posortowana lista wszystkich zdarzeń. Do implementacji struktury stanu używam struktury SortedSet dostępnej w bibliotece sortedcontainers. Jest ona

zaimplementowana z użyciem drzewa czerwono-czarnego. W węzłach drzewa przechowuje linie jako indeksy. Każdy indeks wskazuje na pozycje na jakiej dany punkt znajduje się w liście przechowującej obiekty klasy Line. Domyślny komparator porównałby jedynie wartości tych indeksów, więc należy zdefiniować własny komparator. W pythonie można to zrealizować używając funkcji cmp_to_key dostępnej w module functools. Pozwala ona zdefiniować własny komparator, który dostaje dwa indeksy i jeśli są one równe zwraca 0, jeśli pierwszy argument jest większy od drugiego dowolną dodatnią wartość, a jeśli drugi większy od pierwszego to dowolną ujemną wartość. Struktura SortedSet realizuje wszystkie potrzebne operacje (czyli znajdowanie konretnego obiektu, jego poprzednika, następnika, usuwanie oraz wstawianie) w czasie O(log(n)) zachowując przy tym zdefiowany porządek.

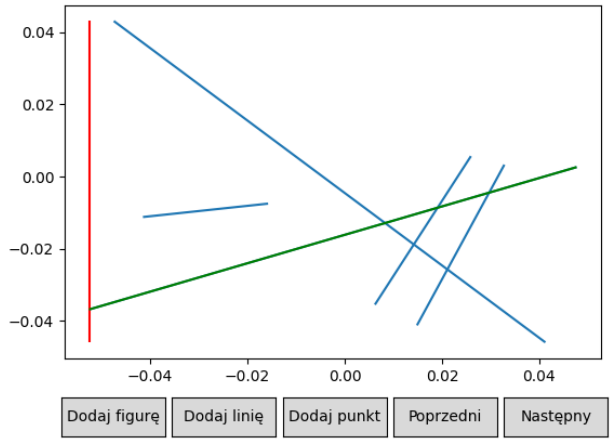
4.2 Algorytm znajdujący wszystkie przecięcia

W przypadku tego algorytmu struktura stanu nie ulega zmienia, lecz rozważane są także zdarzenia przecięcia dwóch odcinków. Oznacza to, że do struktury zdarzeń będą dynamicznie dochodzić nowe zdarzenia. Nie może już ona być listą, gdyż wtedy wstawianie nowego elementu będzie miało złożoność O(n). Dla struktury zdarzeń potrzebne są jedynie dwie operacje: wstawienie nowego zdarzenia oraz wzięcie najwcześniejszego zdarzenia, czyli punktu o najmniejszej współrzędnej x. Strukturę zdarzeń można więc zrealizować używając PriorityQueue z biblioteki Queue zaimplementowanej z użyciem kopca binarnego. W trakcie obsługi zdarzenia przecięcia dwóch odcinków, należy zamienić kolejność dwóch przecinających się odcinków w strukturze stanu. W rozważanym punkcie jednak mają one tą samą wartość, a tak naprawdę na względu na dokładność obliczeń te wartości prawie zawsze będą się minimalnie różnić, więc należy także zaaktualizować funkcje porównującą dwa odcinki w strukturze stanu, aby oddzielnie rozważała moment w którym porównuje takie odcinki. Kolejnym możliwym utrudnieniem w przypadku tego algorytmu jest to, że w trakcie „zamiatania” pewne odcinki wielokrotnie mogą stać się nowymi sąsiadami w strukturze stanu przez co wielokrotnie sprawdzałbym te samy pary odcinków. Więc za każdym razem, gdy sprawdzam czy odcinek x przecina y, dodaje do zbioru krotkę (x, y). Zbiór jest w pythonie zaimplementowany z użyciem tablicy haszującej, więc sprawdzenie czy dana para odcinków była sprawdzana działa w czasie O(1).

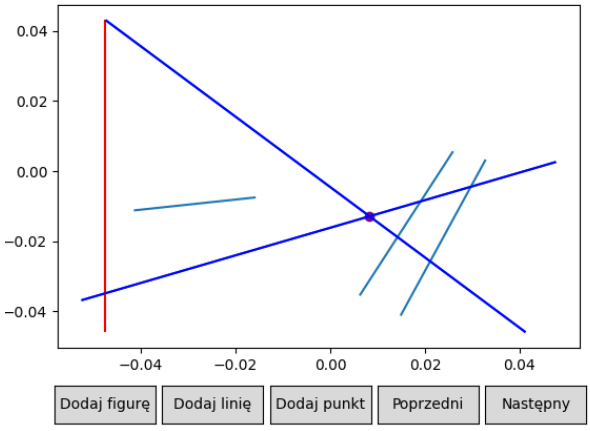
5. Wizualizacja procesu zamiatania.

Pełna wizualizacja dla dowolnego zbioru jest dostępna w pliku ipynb w funkcji *visualize_line_sweep*. Poniżej zamieszczam kolejne (wybrane) kroki algorytmu dla zbioru A.
Czerwona jest to przemieszczająca się w dodatnim kierunku osi x miotła. Na ciemno-niebieskie zaznaczam linie, dla których w danym kroku sprawdzam czy się ze sobą przecinają, na fioletowo znalezione punkty przecięć, na zielono nowo dodane linie i także na czerwono linie właśnie usunięte.

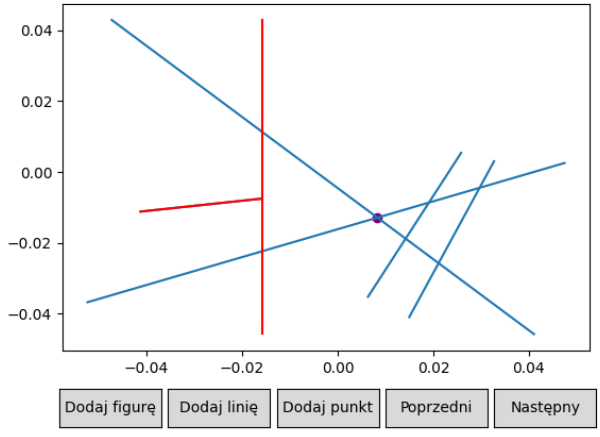
Wykres 5.1 Dodanie nowej linii



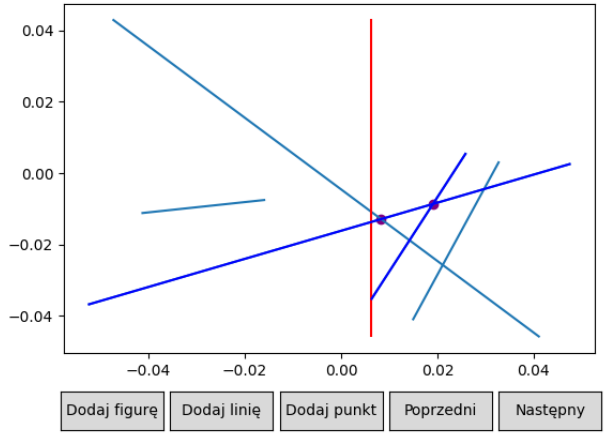
Wykres 5.2 Znalezienie przecięcia



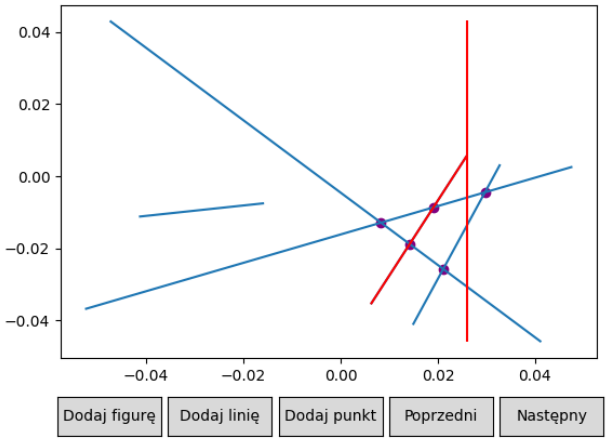
Wykres 5.3 Usunięcie odcinka



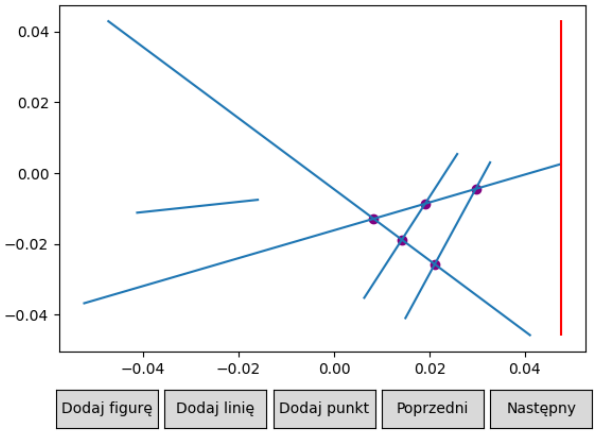
Wykres 5.4 Znalezienie przecięcia



Wykres 5.5 Usunięcie odcinka



Wykres 5.6 Zakończenie działania



6. Wnioski

Algorytm zadziałał poprawnie dla wszystkich testowanych zbiorów danych. Ponadto dla ułatwienia sprawdzania poprawności algorytmu na większych zbiorach danych zaimplementowałem naiwny algorytm szukający przecięć poprzez sprawdzania każdej pary odcinków. Algorytm naiwny i optymalny w każdym przypadku testowym znalazły tą samą ilość punktów, więc można stwierdzić, że algorytm został zaimplementowany poprawnie.