Przeszukiwanie przestrzeni – wsi lab 1

Autor: Patryk Zdziech

Nr.a 311028

opis implementacji algorytmów:

Zanim przejdę do konkretnych algorytmów chciałbym omówić rozwiązania wspólne dla wszystkich algorytmów.

Dla ułatwienia pracy utworzona została klasa `Optymalization`

Przy inicjacji wprowadzam do niej funkcje „f” na której będziemy pracować:

```
def __init__(self, f):
     self.f = f
```

By następnie wywołać na niej jeden z zaimplementowanych algorytmów:

```
def gradient_algorithm(self, x):

def newton_algorithm(self, x):

def adaptative_algorithm(self, x):
```

Używam też dwóch innych zmiennych:

tablicy `function_values = []` po wykonaniu algorytmu móc obejrzeć zapisane wartość funkcji „f” występujące podczas działania algorytmu

oraz wartości stałej `EPSILON = 0.00001` której używam do aproksymacji pochodnej i jako wymagana dokładność w warunku stopu.

Używam własnej funkcji do liczenia aproksymacji gradientu. Pochodna w tym celu jest aproksymowana przez `(f(x + EPSILON) - f(x)) / EPSILON`.

Algorytm gradientu:

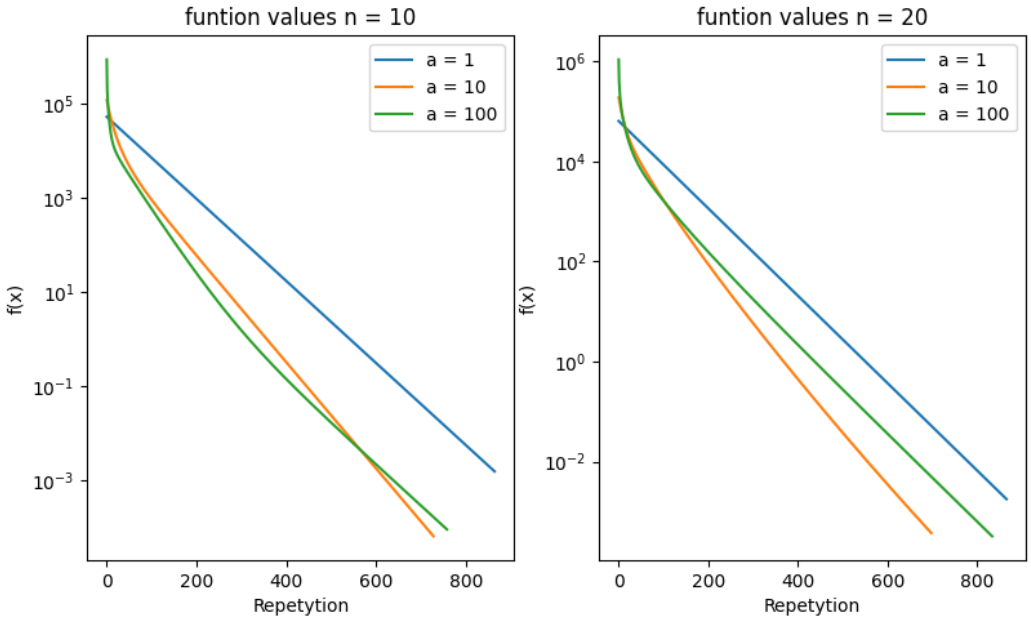
Trzymałem się standardowych założeń podczas mutacji x, użyłem gradientu przemnożonego przez wartość kroku.

$d \leftarrow \nabla q(x)$

$x \leftarrow x+\beta t\, d$

Wartość kroku `step = 0.005` została dobrana metodą prób i błędów. Nie możemy oczywiście ustawić zbyt małego parametru kroku gdyż wpłynie to negatywnie na czas działania algorytmu. Jednak wartość 0.001 okazała się największą wartością zapewniającą mi pewność że algorytm nie będzie nieustannie przeskakiwał minima i w końcu zatrzyma się.

Sprawdźmy jak sprawdził się algorytm gradientowy:



Wymagał on sporej liczby powtórzeń, ale okazał się dość stały niezależnie od złożoności funkcji potrzebował podobnej liczby iteracji. Osiągnęliśmy też liczbę stosunkowo bliską zera. Oczywiście moglibyśmy uzyskać dokładniejszy wynik manipulując wartością `EPSILON`

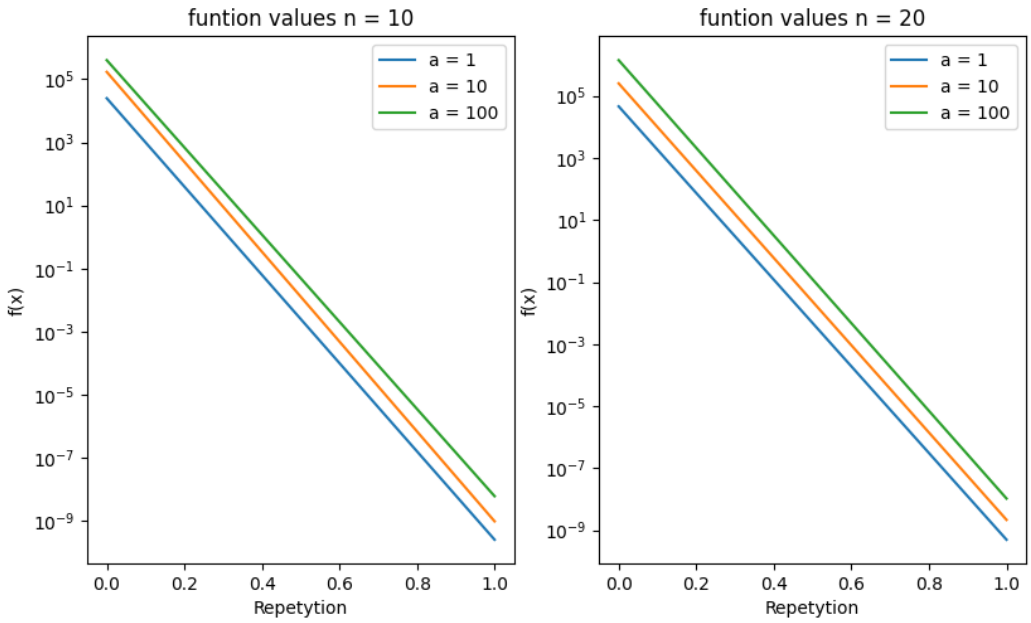
Algorytm Newtona:

Trzymałem się założeń dotyczących mutacji. Użyłem gradientu przemnożonego przez odwrotność hesianu przemnożonego przez wartość kroku wartość kroku.

$d \leftarrow H_q^{-1}(x) \cdot \nabla q(x)$

$x \leftarrow x + \beta d$

Okazało się jednak że algorytm dla parametru skoku równego jeden zadziałał w jednym powtórzeniu, co oznacza że mogłem wyeliminować parametr skoku.

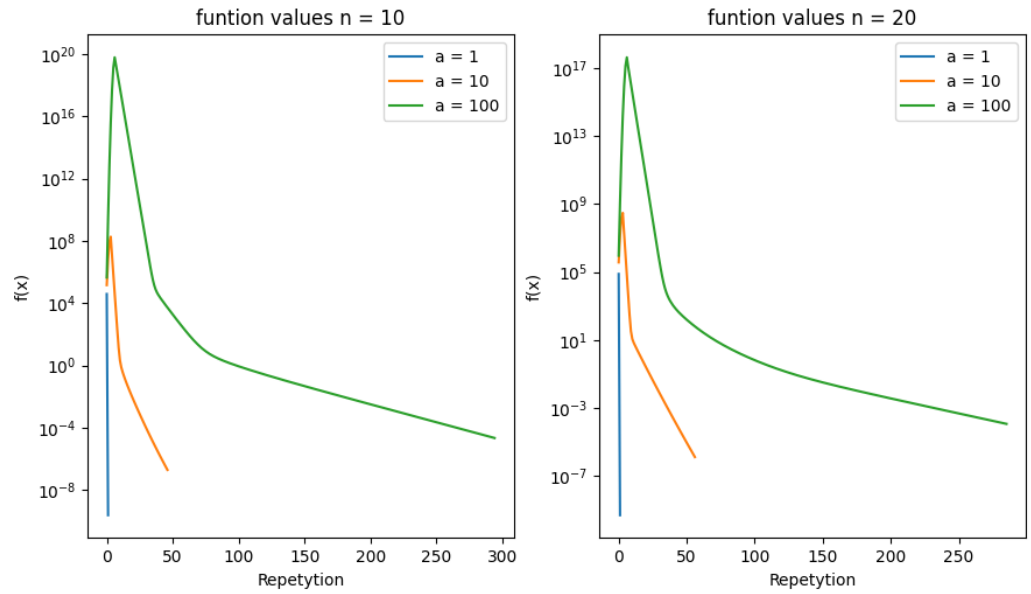


W związku z tym zastosowałem metodę adaptacji parametru kroku do metody gradientu.

Metoda z adaptacją kroku:

Przyjąłem tym razem znacznie większy początkowy parametr kroku `step = 0.5` , a następnie za każdym razem gdy algorytm mijał minimum zmniejszałem go o połowę.

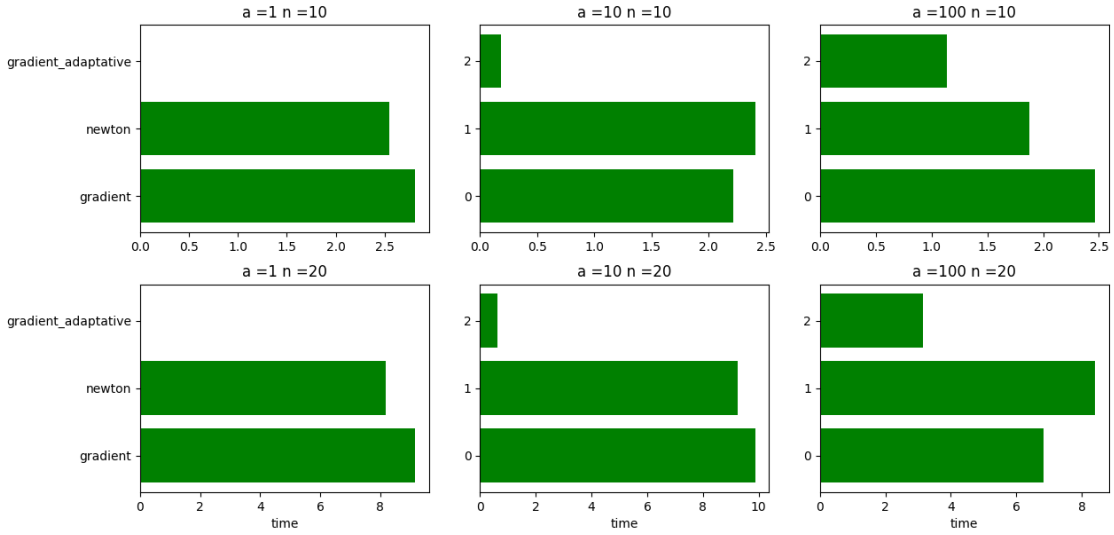
```
if f(x2) - f(x) > 0:
    step /= 2
```



Widać że na początku algorytm przeskakuje minimum dla funkcji o większym zakresie wartości, jednak mimo to liczba iteracji znacząco zmalała. Dla prostszych funkcji algorytm znajduje minimum bardzo szybko, jednak widzimy też że liczba iteracji dużo szybciej wzrasta wraz z parametrem „a” w porównaniu do innych algorytmów

Podsumowanie:

Na koniec przedstawiam porównanie czasu wykonywania algorytmów:



Jak widzimy dla funkcji o mniejszej złożoności algorytm adaptacyjny ma zdecydowaną przewagę, natomiast jego czas działania zdaje się rosnąć wykładniczo.