Laboratorium 1 - Wojciech Kołodziejak 310747

Treść zadania:

Zaimplementuj algorytm gradientu prostego oraz algorytm Newtona. Algorytm Newtona powinien móc działać w dwóch trybach:

- ze stałym parametrem kroku
- z adaptacją parametru kroku przy użyciu metody z nawrotami.

Następnie zbadaj zbieżność obu algorytmów, używając następującej funkcji:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \alpha^{\frac{i-1}{n-1}} x_i^2, \ \mathbf{x} \in [-100, 100]^n \subset \mathbb{R}^n.$$

Zbadaj wpływ wartości parametru kroku na zbieżność obu metod. W swoich badaniach rozważ następujące wartości parametru **α∈{1,10,100}** oraz dwie wymiarowości **n∈{10,20}**. Ponadto porównaj czasy działania obu algorytmów.

Opis implementacji

W implementacji zostały użyte następujące biblioteki:

- NumPy - macierze i działania na nich
- numdifftools - funkcje obliczające gradienty i hesjany funkcji
- matplotlib - sporządzanie wykresów

Implementacja algorytmu gradientu prostego w języku Python

```
def gradient_descent(fun: MathFunction, starting_point: np.ndarray, step_size:
float = 0.001, values: List[float] = None,precision: float = 1e-03) ->
Tuple[np.ndarray, int]:
        if values is None:
            values = []
        current_point = starting_point
        gradient = nd.Gradient(fun)
        iters = 0
        while iters < MAX_ITERS:
            iters += 1
            values.append(fun(current_point))
            gradient_at_point = gradient(current_point)
            next_point = current_point - step_size * gradient_at_point
            points_diff = np.abs(next_point - current_point)
            current_point = next_point
            if np.all(np.abs(points_diff) <= precision):
                return current_point, iters
        return None, iters
```

Algorytm gradientu prostego wykorzystuje gradient funkcji w punkcie, który decyduje (zależenie od znaku) o zwiększeniu lub zmniejszeniu argument. Aby wyliczyć minimum funkcji n zmiennych podjęte są następujące kroki:

1. Wybranie losowych (lub zadanych) początkowych argumentów [x_0, x_1, x_2...x_{n-1}]
2. Obliczenie gradientu w punkcie dla obecnego punktu - $\nabla f(x_0..x_{n-1})$

3. Wyznaczenie następnego argumentu ze wzoru $x_{k+1} = x_k - \alpha\nabla f(x_0..x_{n-1})$, gdzie α - podana wielkość kroku
4. Sprawdzanie warunku stopu $|x_{k+1} - x_k| <= \varepsilon$, gdzie ε - podana precyzja
5. Jeśli warunek stopu jest spełniony zwracana jest tablica końcowych argumentów [x_0, x_1, x_2...x_{n-1}] oraz liczba iteracji. W przeciwnym wypadku algorytm wraca do punktu 2.

Wielkość kroku należy wybrać zależnie od funkcji - zbyt duży może powodować wpadanie algorytmu w oscylacje i prowadzić do niedokładnego wyniku, a gdy będzie zbyt mały może prowadzić do długiego czasu znajdowania minimum. Z tego powodu zostało wprowadzone ograniczenie iteracji wynoszące 30000

Implementacja algorytmu Newton ze stałym krokiem w języku Python

```
def newton_constant_step(fun: MathFunction, starting_point: np.ndarray,
step_size: float = 0.001, values: List[float] = None, precision: float = 1e-03) -
> Tuple[np.ndarray, int]:
        if values is None:
            values = []
        current_point = starting_point
        hessian = nd.Hessian(fun)
        gradient = nd.Gradient(fun)
        iters = 0
        while iters < MAX_ITERS:
            iters += 1
            values.append(fun(current_point))
            gradient_at_point = gradient(current_point)
            hessian_at_pointt_inv = np.linalg.inv(hessian(current_point))
            next_point = current_point - step_size *
np.matmul(hessian_at_pointt_inv, gradient_at_point)
            points_diff = np.abs(next_point - current_point)
            current_point = next_point
            if np.all(np.abs(points_diff) <= precision):
                return current_point, iters
        return None, iters
```

Algorytm optymalizujący Newtona pozwala znaleźć minimum funkcji podwójnie różniczkowalnej funkcji. Do obliczenia kierunku poszukiwań wykorzystuje się rozwinięcia Taylora. W tym celu stosuje się następujące kroki:

1. Wybranie losowych (lub zadanych) początkowych argumentów [x_0, x_1, x_2...x_{n-1}]
2. Obliczenie gradientu oraz odwróconej macierzy Hessego w punkcie dla obecnego punktu - $\nabla f(x_0..x_{n-1})$ i $(\nabla^2 f(x_0..x_{n-1}))^{-1}$
3. Wyznaczenie następnego argumentu ze wzoru $x_{k+1} = x_k - \alpha(\nabla^2 f(x_0..x_{n-1}))^{-1}\nabla f(x_0..x_{n-1})$, gdzie α - podana wielkość kroku
4. Sprawdzanie warunku stopu $|x_{k+1} - x_k| <= \varepsilon$, gdzie ε - podana precyzja
5. Jeśli warunek stopu jest spełniony zwracana jest tablica końcowych argumentów [x_0, x_1, x_2...x_{n-1}] oraz liczba iteracji. W przeciwnym wypadku algorytm wraca do punktu 2.

Implementacja algorytmu Newtona ze zmiennym krokiem w języku Python

```
def newton_backtracking_step(fun: MathFunction, starting_point: np.ndarray,
values: List[float] = None, precision: float = 1e-03) -> Tuple[np.ndarray, int]:
        if values is None:
            values = []
        current_point = starting_point
        hessian = nd.Hessian(fun)
        gradient = nd.Gradient(fun)
        iters = 0
        step_size = 1
        while iters < MAX_ITERS:
            iters += 1
            values.append(fun(current_point))
            gradient_at_point = gradient(current_point)
            hessian_at_pointt_inv = np.linalg.inv(hessian(current_point))
            direction = np.matmul(hessian_at_pointt_inv, gradient_at_point)
            step_size = FunctionOptimization._backtracking_line_search(fun,
current_point, step_size, gradient, direction)
            next_point = current_point - direction * step_size
            points_diff = np.abs(next_point - current_point)
            current_point = next_point
            if np.all(np.abs(points_diff) <= precision):
                return current_point, iters
        return None, iters
```

Algorytm Newtona ze zmiennym krokiem rożni się od wersji ze stałym korkiem, tym że w każdej iteracji jest obliczany krok, który w danej iteracji jest optymalny. W tym celu używa się metodę nawrotu z wykorzystaniem warunku Armijo – Goldsteina. W celu wyliczenia optymalnej wielkości kroku wybiera się parametry alfa i beta takie, że 0<beta, alfa<1. Następnie odpowiednio zmniejszany jest krok dopóki prawdziwy jest warunek:

$$f(x - td) > f(x) + at(f'(x) \cdot -d)$$

gdzie: x - obecny punkt, t - wielkość korku, d - kierunek poszukiwań równy $(\nabla^2 f(x_0..x_{n-1}))^{-1} \nabla f(x_0..x_{n-1})$, a - parametr alfa f'(x) - pochodna (gradient) funkcji

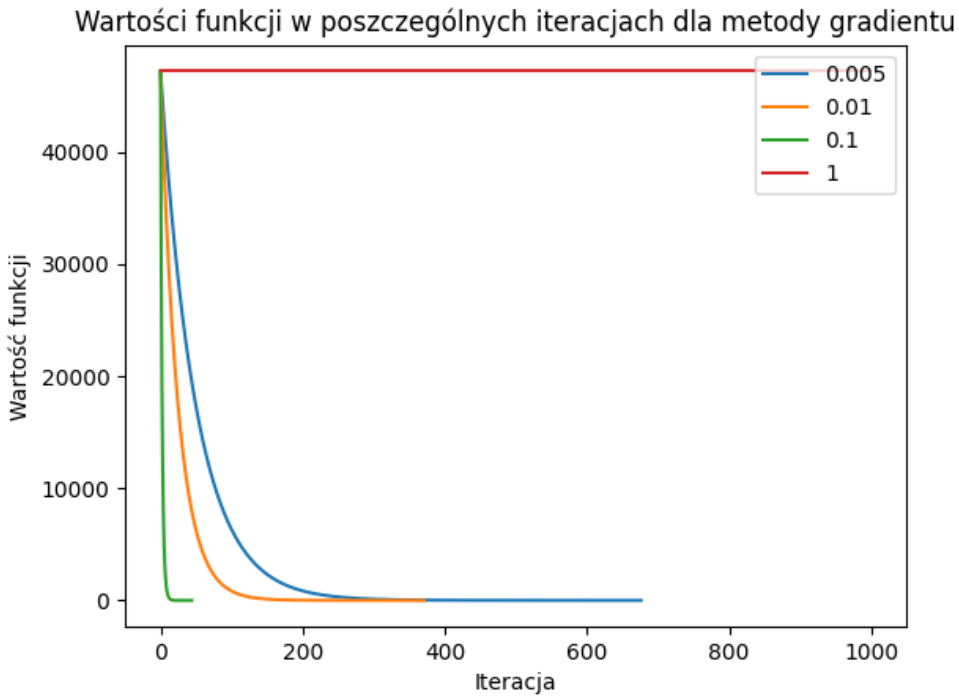
```
def _backtracking_line_search(fun: MathFunction, point: np.ndarray, step: float,
gradient: nd.Gradient, direction: np.ndarray) -> float:
        alpha = 0.3
        beta = 0.6
        while fun(point - step * direction) > fun(point) + alpha * step *
np.dot(np.transpose(gradient(point)), -direction):
            step *= beta
        return step
```

Dzięki tej metodzie algorytm znacznie skraca swój czas działania, ponieważ zamiast kosztownych obliczeń hesjanów wykonuje się prostsze obliczenia w celu wyznaczenia optymalnego kroku.

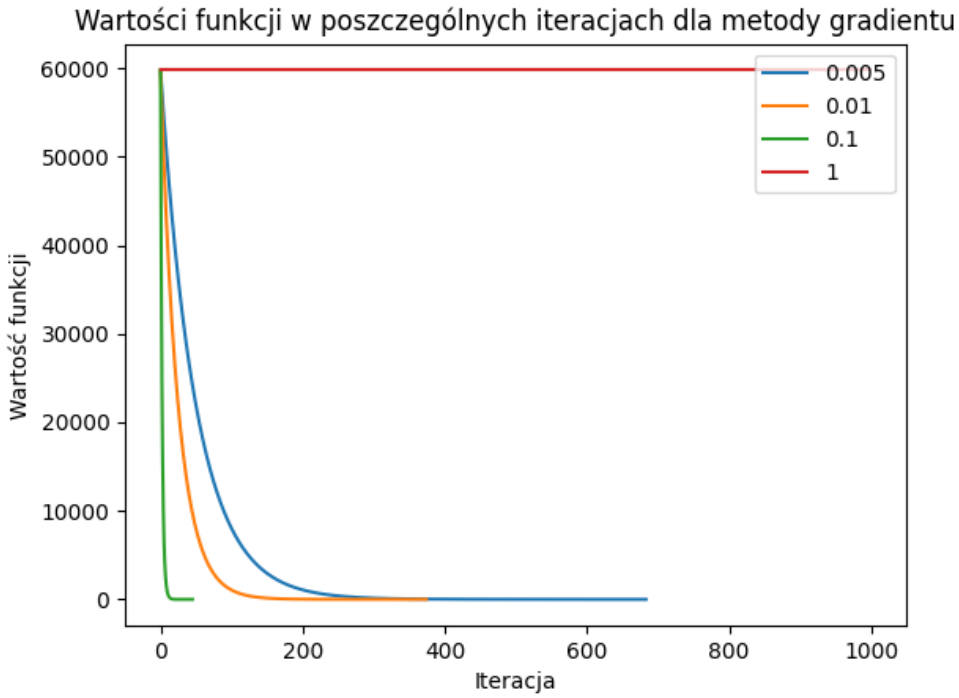
Eksperymenty numeryczne

W celu zbadania wpływu wielkości kroku na zbieżność metod zostały przeprowadzane eksperymenty numeryczne. Dla każdego parametru a (1, 10, 100) oraz n (10, 20) użyto obu metod do obliczenia minimum. Każdy przypadek był rozpatrywany dla różnej wielkości kroku (0.005, 0.01, 0.1, 1). Następnie zostały sporządzone wykresy pokazujące zbieganie funkcji do jej minimum w zależności od iteracji.

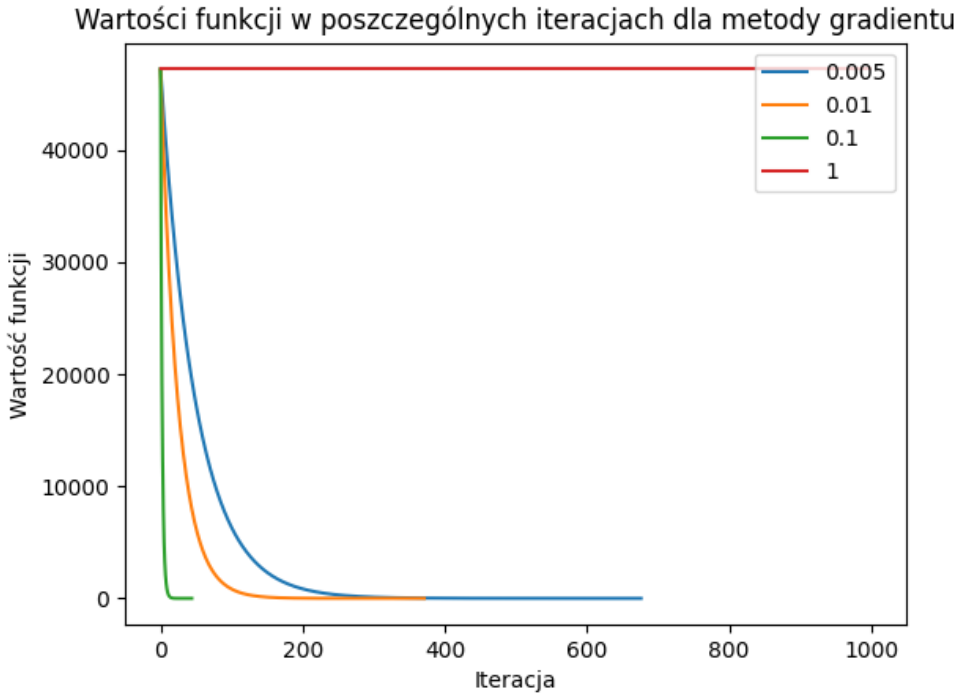
- Metoda gradientu prostego
  - n = 10
    - a = 1



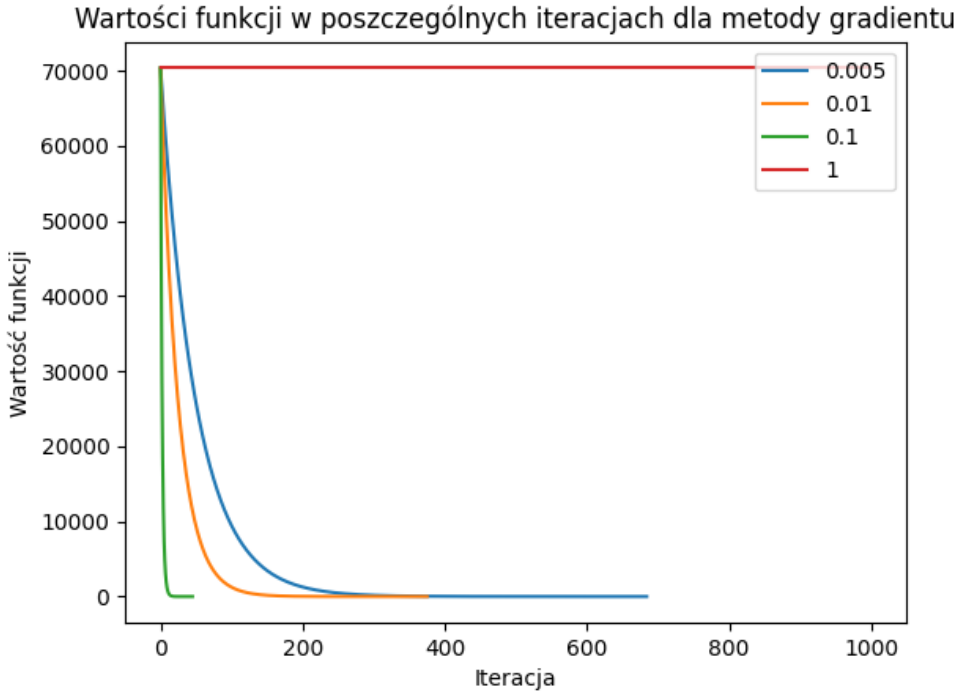
    - a = 10



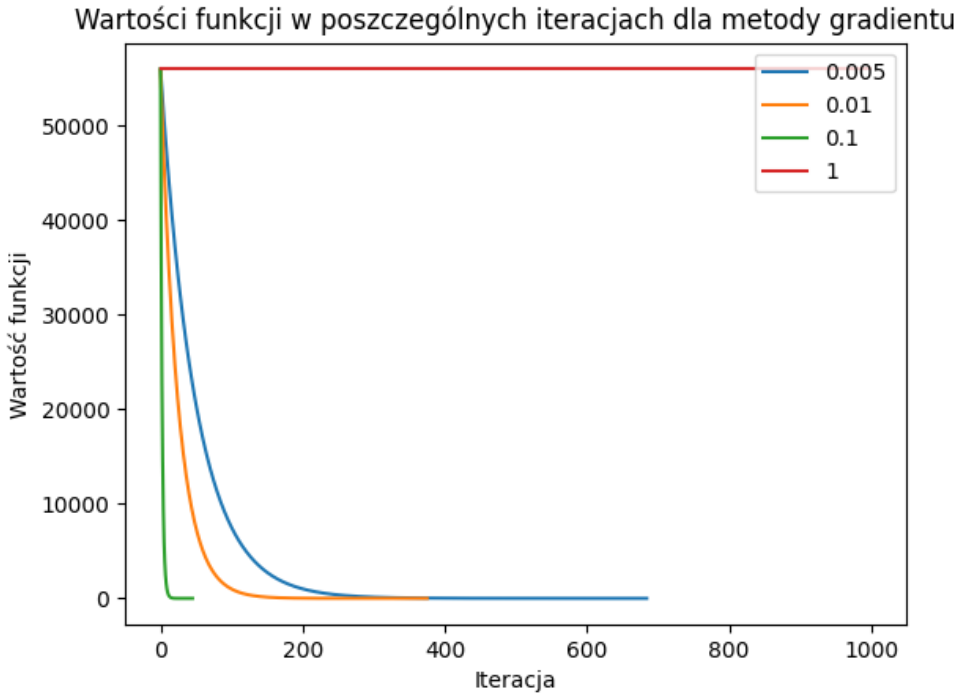
- a = 100



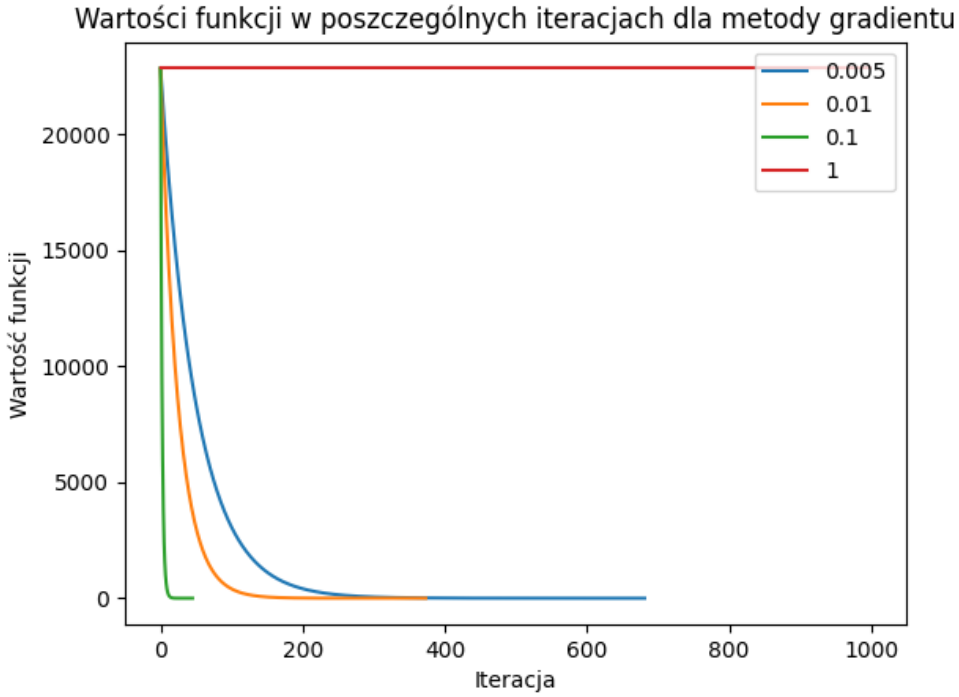
- n = 20
  - a = 1



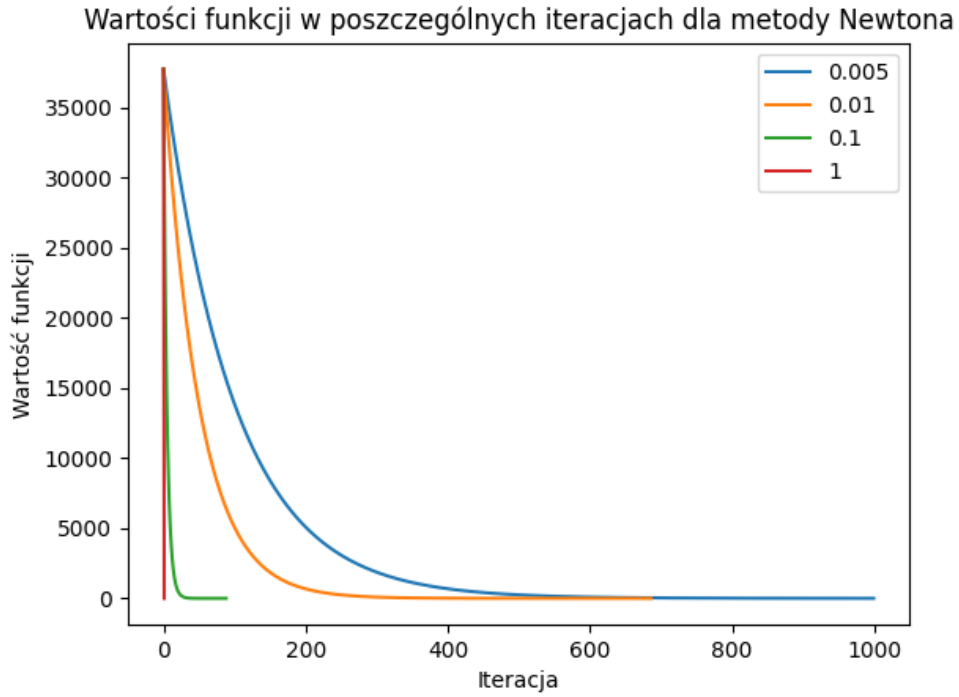
  - a = 10



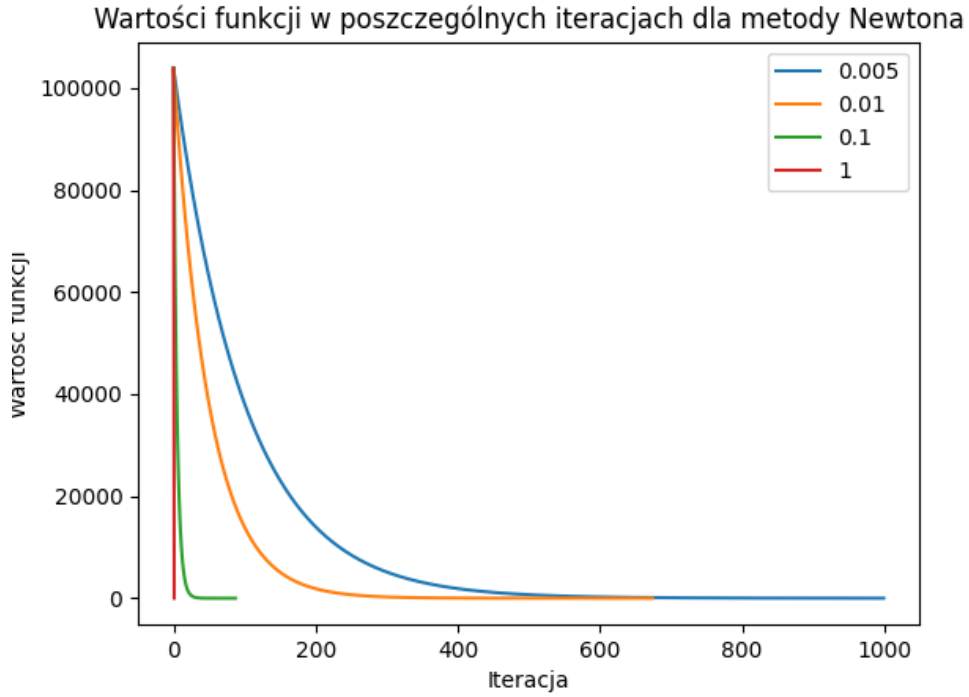
- a = 100



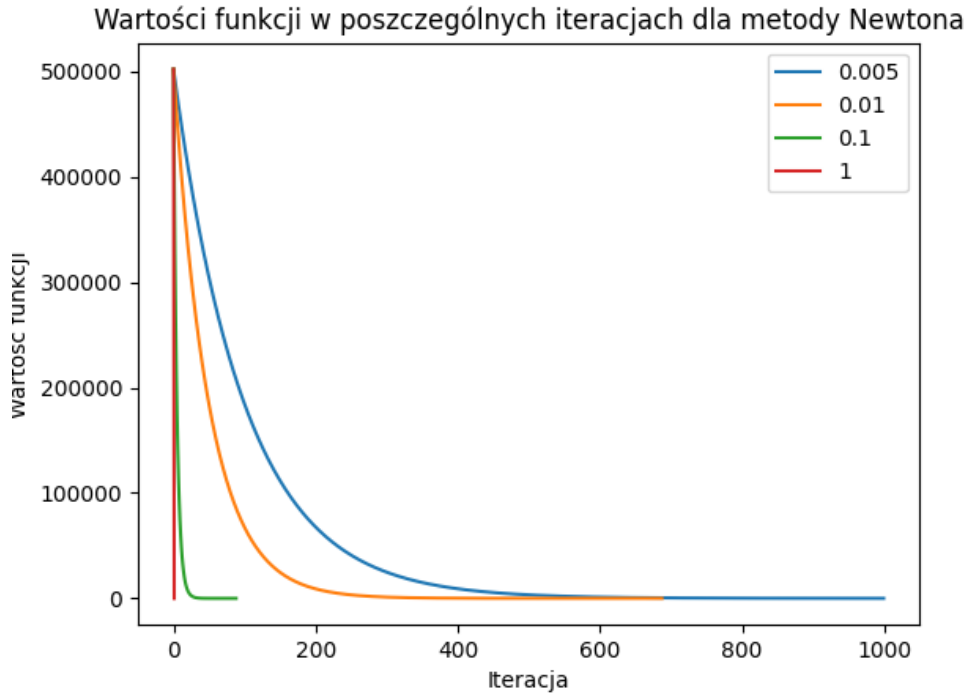
- Metoda Newtona
  - n = 10
    - a = 1


Wartości funkcji w poszczególnych iteracjach dla metody Newtona
0.005
0.01
0.1
1
Wartość funkcji
Iteracja
0
5000
10000
15000
20000
25000
30000
35000
0
200
400
600
800
1000

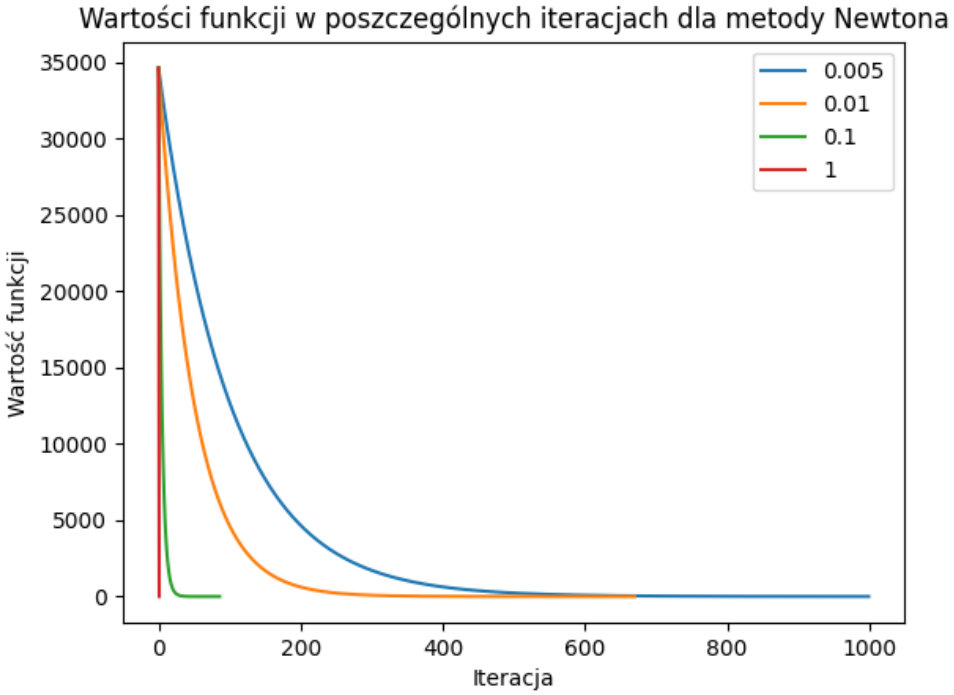
- a = 10


Wartości funkcji w poszczególnych iteracjach dla metody Newtona
0.005
0.01
0.1
1
Wartosc funkcji
Iteracja
0
20000
40000
60000
80000
100000
0
200
400
600
800
1000

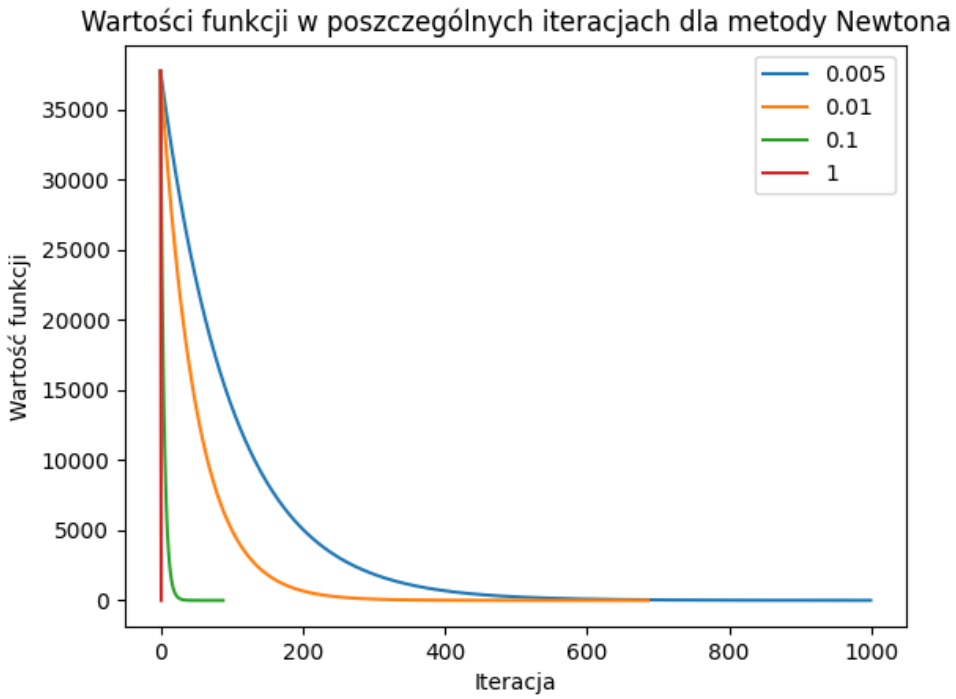
- a = 100


Wartości funkcji w poszczególnych iteracjach dla metody Newtona
wartosc funkcji
Iteracja
0.005
0.01
0.1
1
500000
400000
300000
200000
100000
0
0
200
400
600
800
1000

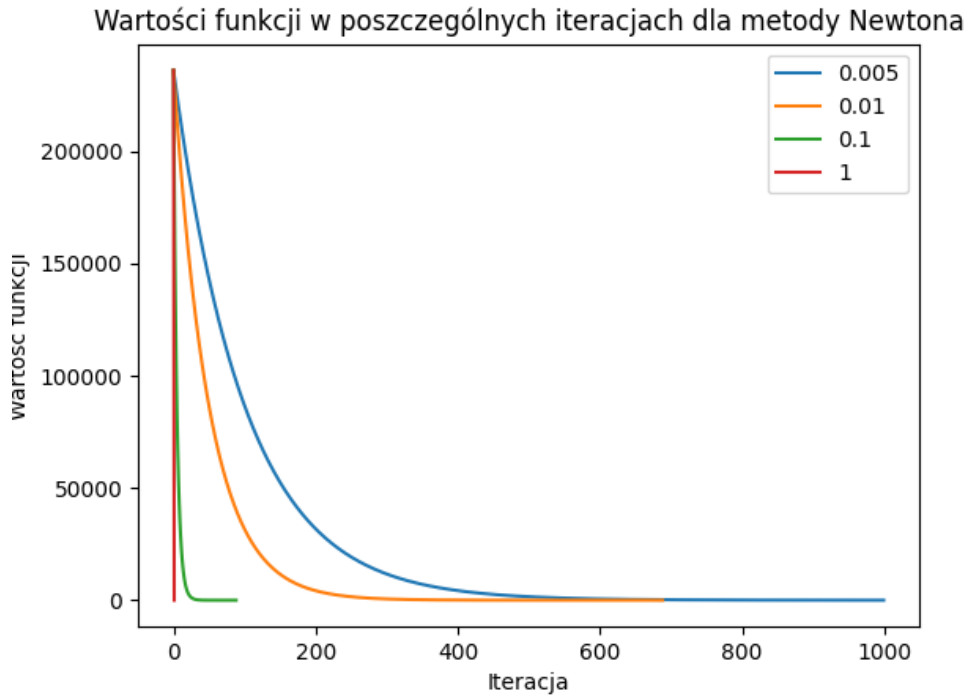
- n = 20
  - a = 1


Wartości funkcji w poszczególnych iteracjach dla metody Newtona
Wartość funkcji
Iteracja
0.005
0.01
0.1
1
35000
30000
25000
20000
15000
10000
5000
0
0
200
400
600
800
1000

  - a = 10



- a = 100



Z podanych wykresów można wywnioskować jaką wielkość kroku należy wybrać dla każdego rodzaju funkcji aby oba algorytmy wykonały jak najmniej iteracji i jak najszybciej znalazły minimum.

Ponadto zbadano również szybkość wykonywania się metody Newtona z nawrotami, który okazuje się najszybszym algorytmem znajdywania minimum. Również dla każdego typu funkcji wykonano eksperyment i otrzymano wyniki zbieżność algorytmu:

- Wyniki wykazały że algorytm Newtona z nawrotami od razu wyznacza optymalną wielkość kroku, która dla tej funkcji niezależnie od paramterów jest 1 i od razu w drugiej iteracji znajduje wynik

Porównania czasowe:

- Średni czas wykonywania się algorytmu gradientu prostego: 15.231s
- Średni czas wykonywania się algorytmu Newtona (bez nawrotów): 160.261s (Wynika to z konieczności obliczania hesjana w punkcie w każdej iteracji)

- Średni czas wykonywania się algorytmu Newtona (z nawrotami): 0.394s

Wnioski:

Z przeprowadzonych eksperymentów wynika, że najlepiej radzi sobie metoda Newtona z nawrotami. Dzięki adaptacyjnej wielkości kroku jest w stanie bardzo szybko znaleźć minimum przez brak skomplikowanych obliczeń macierzy Hessego. Dla innych metod ważnym parametrem jest wielkość kroku. Dobrze ustawiona jest w stanie zmniejszyć znacznie liczbę iteracji algorytmu. W metodzie Newtona bez nawrotów dla kroku = 1 algorytm niemal od razu znajduje właściwe minimum. Za to dla metody gradientu wpada w oscylacje i nie może znaleźć minimum. W dużej części zależy to od danej funkcji (w przypadku naszej jest to idealny krok). Dlatego należy uważnie wybierać krok algorytmu. W przypadku gdy zależy nam na czasie warto wybrać metodę Newtona z nawrotami lub ostatecznie metodę gradientu, ponieważ nie wymaga ona kosztownych obliczeń.