Artur Gęsiarz,
Czerwiec 14, 2024

Laboratorium nr 11
MOwNiT – Optymalizacja

1. Treść zadań

1.1. Zadanie pierwsze

Wyznacz punkty krytyczne każdej z poniższych funkcji. Scharakteryzuj każdy znaleziony punkt jako minimum, maksimum lub punkt siodłowy. Dla każdej funkcji zbadaj, czy posiada minimum globalne lub maksimum globalne na zbiorze R^2.

$$f_1(x,y) = x^2 - 4xy + y^2$$

$$f_2(x,y) = x^4 - 4xy + y^4$$

$$f_3(x,y) = 2x^3 - 3x^2 - 6xy(x - y - 1)$$

$$f_4(x,y) = (x-y)^4 + x^2 - y^2 - 2x + 2y + 1$$

1.2. Zadanie drugie

Należy wyznaczyć najkrótsza ścieżkę robota pomiędzy dwoma punktami x(0) i x(n). Problemem są przeszkody usytuowane na trasie robota, których należy unikać. Zadanie polega na minimalizacji funkcja kosztu, która sprowadza problem nieliniowej optymalizacji z ograniczeniami do problemu nieograniczonej optymalizacji. Macierz $X \in R^{(n+1)\,x\,2}$ opisuje sciezke złozona z $n + 1$ punktów $x^{(0)};\ x^{(1)};\ x^{(2)};\ \dots\ x^{(n)}$. Kazdy punkt posiada 2 współrzedne, $x^{(i)} \in R^2$. Punkty początkowy i końcowy ścieżki, $x^{(0)}$ i $x^{(n)}$, sa ustalone.
Punkty z przeszkodami (punkty o 2 współrzędnych), r(i) dane są w macierzy przeszkód $R \in R^{k\,x\,2}$.
W celu optymalizacji ścieżki robota należy użyć metody największego spadku. Funkcja celu użyta do optymalizacji $F\left(x^{(0)};\ x^{(1)};\ x^{(2)};\ \dots\ x^{(n)}\right)$zdefiniowana jest jako:

$$F\left(x^{(0)};\ x^{(1)};\ x^{(2)};\ \dots\ x^{(n)}\right) = \lambda_1 \sum_{i=0}^{n} \sum_{j=1}^{k} \frac{1}{\varepsilon + ||x^{(i)} - r^{(j)}||_2^2} + \lambda_2 \sum_{i=0}^{n-1} ||x^{(i+1)} - x^{(i)}||_2^2$$

Symbole użyte we wzorze maja następujące znaczenie:

- Stałe λ_1 i λ_2 okreslaja wpływ każdego członu wyrażenia na wartość F(X).

  – λ_1 określa wagę składnika zapobiegającego zbytniemu zblizaniu sie do przeszkody

  – λ_2 określa wagę składnika zapobiegającego tworzeniu bardzo długich ścieżek

- n jest liczba odcinków, a n + 1 liczba punktów na trasie robota
- k jest liczba przeszkód, których robot musi unikać
- Dodanie ε w mianowniku zapobiega dzieleniu przez zero.

(a) Wyprowadź wyrażenie na gradient ∇F funkcji celu F względem $x^{(i)}$

(b) Opisz matematycznie i zaimplementuj kroki algorytmu największego spadku z przeszukiwaniem liniowym, który służy do minimalizacji funkcji celu F. Do przeszukiwania liniowego (ang. line search) użyj metody złotego podziału (ang. golden section search). W tym celu załóż, ze F jest unimodalna (w rzeczywistości tak nie jest) i ze można ustalić początkowy przedział, w którym znajduje się minimum.

(c) Znajdź najkrótszą ścieżkę robota przy użyciu algorytmu zaimplementowanego w poprzednim punkcie.

Przyjmij następujące wartości parametrów:

$$n = 20, k = 50$$

$$x^{(0)} = [0,0], x^{(n)} = [20,20]$$

$$r^{(i)} \sim U(0,20)\ x\ U(0,20)$$

$$\lambda_1 = \lambda_2 = 1$$

$$\varepsilon = 10^{-13}$$

Liczba iteracji = 400

Ponieważ nie chcemy zmieniać położenia punktu początkowego i końcowego, $x^{(0)}, \dots, x^{(n)}$wyzeruj gradient funkcji F wzgledem tym punktów.
Obliczenia przeprowadź dla 5 różnych losowych inicjalizacji punktów wewnątrz ścieżki $x^{(0)};\ x^{(1)};\ x^{(2)}; \dots\ x^{(n)}$.
Narysuj przykładowy wykres wartości funkcji F w zależności od iteracji.

2. Rozwiązanie zadań:

2.1. Implementacja zadania pierwszego:

2.1.1. Implementacja funkcji

```
def function_1(x,y):
    return x ** 2 - 4 * x * y + y ** 2

def function_2(x,y):
    return x ** 4 - 4 * x * y + y ** 4

def function_3(x,y):
    return 2 * x ** 3 - 3 * x ** 2 - 6 * x * y * (x - y - 1)

def function_4(x,y):
    return (x - y) ** 4 + x ** 2 - y ** 2 - 2 * x + 2 * y + 1
```

2.1.2. Obliczenia gradientów kolejnych funkcji

```
def grad_function_1(x, y):
    return np.array([2 * x - 4 * y,
                     2 * y - 4 * x])

def grad_function_2(x, y):
    return np.array([4 * x ** 3 - 4 * y,
                     4 * y ** 3 - 4 * x])

def grad_function_3(x, y):
    return np.array([6 * x ** 2 - 6 * x - 12 * x * y + 6 * y ** 2 + 6 * y,
                     -6 * x ** 2 + 12 * x * y + 6 * x])

def grad_function_4(x, y):
    return np.array([4 * (x - y) ** 3 + 2 * x - 2,
                     -4 * (x - y) ** 3 - 2 * y + 2])
```

2.1.3. Obliczanie punktów krytycznych

```
critical_points = {
    "f1" : np.array([(0, 0)]),
    "f2" : np.array([(-1, -1), (0, 0), (1, 1)]),
    "f3" : np.array([(0, 0), (0, -1), (1, 0), (-1, -1)]),
    "f4" : np.array([(5/2, 1)]),
}
```

2.1.4. Obliczanie hesjanów dla poszczególnych funkcji

```
def hessian_f1(x, y):
    return np.array([[2, -4],
                     [-4, 2]])
```

```
def hessian_f2(x, y):
    return np.array([[12 * x ** 2, -4],
                     [-4, 12 * y ** 2]])
```

```
def hessian_f3(x, y):
    return np.array([[12 * x - 12 * y - 6, -12 * x + 12 * y + 6],
                     [-12 * x + 12 * y + 6, 12 * x]])
```

```
def hessian_f4(x, y):
    return np.array([[12 * (x - y) ** 2 + 2, -12 * (x - y) ** 2],
                     [-12 * (x - y) ** 2, 12 * (x - y) ** 2 - 2]])
```

2.1.5. Funkcja kwalifikująca punkty krytyczne

```
def qualifications_points(hessian, critical_points_array):
    for (x,y) in critical_points_array:
        hessian_matrix = hessian(x,y)
        det_hessian_2x2 = np.linalg.det(hessian_matrix)

        if hessian_matrix[0][0] > 0 and det_hessian_2x2 > 0:
            print(f"Minimum lokalne jest w punkcie: ({x},{y})")

        elif hessian_matrix[0][0] < 0 and det_hessian_2x2 > 0:
            print(f"Maximum lokalne jest w punkcie: ({x},{y})")

        else:
            print(f"Punkt siodłowy w punkcie: ({x},{y})")
```

```
def critical_points_characteristics_all():
    hessians = [hessian_f1,
                hessian_f2,
                hessian_f3,
                hessian_f4]

    for function_number in range(1,5):
        function_ = "f" + str(function_number)
        print(f"Charakterystyka dla funkcji nr. {function_number}:\n")
        qualifications_points(hessians[function_number - 1], critical_points[function_])
        print("\n")
```

2.1.6. Charakterystyka punktów krytycznych

```
critical_points_characteristics_all()
```

2.2. Implementacja zadania drugiego:

2.2.1. Funkcja gradientów

```
def gradient(X, R, lambda1, lambda2, epsilon):
    grad = np.zeros_like(X)
    n = X.shape[0] - 1
    k = R.shape[0]

    for i in range(1, n):
        for j in range(k):
            d_ij = np.linalg.norm(X[i] - R[j]) ** 2
            grad[i] += lambda1 * (-2 * (X[i] - R[j]) / (epsilon + d_ij) ** 2)

        grad[i] += lambda2 * (2 * (X[i] - X[i-1]) - 2 * (X[i+1] - X[i]))

    return grad
```

2.2.2. Metoda złotego podziału

```
def golden_section_search(f, a, b, tol=1e-5):
    gr = (np.sqrt(5) + 1) / 2
    c = b - (b - a) / gr
    d = a + (b - a) / gr
    while abs(c - d) > tol:
        if f(c) < f(d):
            b = d
        else:
            a = c
        c = b - (b - a) / gr
        d = a + (b - a) / gr
    return (b + a) / 2
```

2.2.3. F(x)

```
def objective(X, R, lambda1, lambda2, epsilon):
    n = X.shape[0] - 1
    k = R.shape[0]
    term1 = sum([sum([1 / (epsilon + np.linalg.norm(X[i] - R[j]) ** 2) for j in range(k)]) for i in range(n +
    1)])
    term2 = sum([np.linalg.norm(X[i+1] - X[i]) ** 2 for i in range(n)])
    return lambda1 * term1 + lambda2 * term2
```

2.2.4. Przeszukiwanie liniowe

```
def line_search(X, dX, R, lambda1, lambda2, epsilon):
    def f(alpha):
        return objective(X - alpha * dX, R, lambda1, lambda2, epsilon)
    return golden_section_search(f, 0, 1)
```

2.2.5. Algorytm największego spadku gradientu

```python
def gradient_descent(X, R, lambda1, lambda2, epsilon, max_iter):
    F_values = []

    for _ in range(max_iter):
        # Obliczam gradient
        grad = gradient(X, R, lambda1, lambda2, epsilon)

        # Zretuje gradient na poczatku i na koncu aby nie zmieniac polozenia punktu poczatkowego oraz koncoweg
        grad[0] = grad[-1] = 0

        # Obliczam wspolczynnik przesukiwaniem liniowym uzywajac metody zlotego podzialu
        alpha = line_search(X, grad, R, lambda1, lambda2, epsilon)

        # Przesuniecie robota
        X -= alpha * grad

        # Dodanie wartosci pola na ktore poruszyl sie robot
        F_values.append(objective(X, R, lambda1, lambda2, epsilon))

    return X, F_values
```

2.2.6. Poszczególne parametry

```python
n = 20
k = 50
x0 = np.array([0, 0])
xn = np.array([20, 20])
np.random.seed(0)
R = np.random.uniform(0, 20, (k, 2))
lambda1, lambda2 = 1, 1
eps = 1e-13

X = np.zeros((n + 1, 2))
X[0] = x0
X[-1] = xn
```

2.2.7. Wykonanie algorytmu dla 5 róznych iteracji

```python
for _ in range(5):
    # Losuje wspolrzedne X
    X[1:-1] = np.random.uniform(0, 20, (n-1, 2))

    # Stosuje algortym najwiekszego spadku gradientu, aby robot znalazl wlasciwa grode dla 400 iteracji
    X_opt, F_values = gradient_descent(X, R, lambda1, lambda2, eps, 400)

    # Pierwszy wykres - pokazujacy zmiane wartosc funkcji F w zaleznosci od iteracji
    plt.plot(F_values)
    plt.xlabel('Iteracja')
    plt.ylabel('Wartość funkcji F')
    plt.title('Zmiana wartości funkcji F w zależności od iteracji')
    plt.yscale('log')
    plt.show()

    # Drugi wykres - pokazujacy sciezke naszego robota
    plt.plot(X_opt[:, 0], X_opt[:, 1], 'o-', label='Ścieżka')
    plt.scatter(R[:, 0], R[:, 1], c='r', marker='x', label='Przeszkody')
    plt.legend()
    plt.title('Ścieżka robota')
    plt.show()
```

3. Tabele

3.1. Tabela dla zadania pierwszego dla kwalifikacji puntków

Nr. funkcji	Punkty siodłowe	Punkty maksymalne	Punkty minimalne
1	$(0,0)$	$-$	$-$
2	$(0,0)$	$-$	$(-1,-1),(1,1)$
3	$(0,0),(0,-1)$	$(-1,-1)$	$(1,0)$
4	$(2.5,1.0)$	$-$	$-$

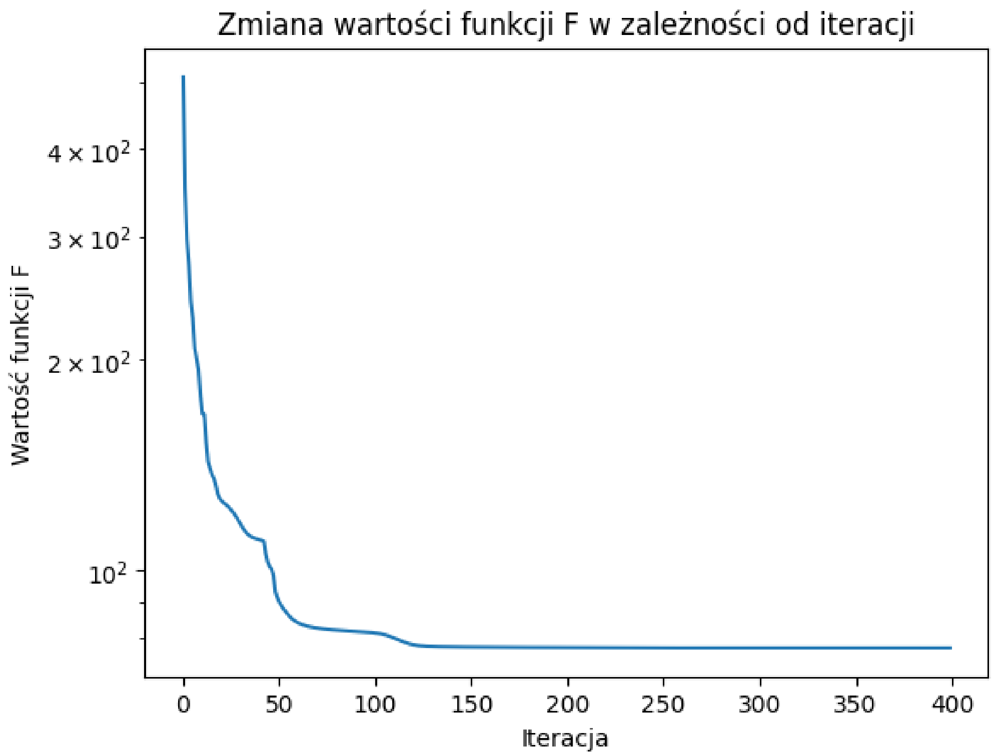
Tabela 1. Tabela opisująca kwalifikacje punktów

4. Wykresy

4.1. Wykresy dla zadania drugiego

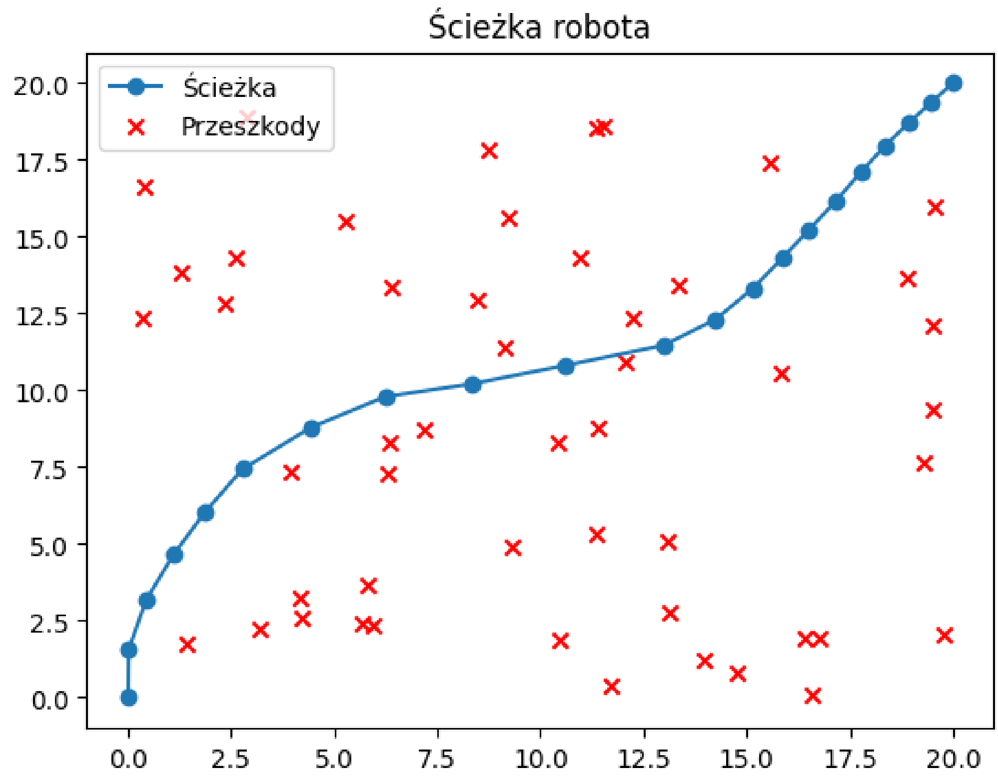
4.1.1. Pierwsza iteracja

4.1.1.1. Zmiana wartości funkcji F w zależności od iteracji



Wykres 1. Wykres zmiany wartości funkcji F w zależności od pierwszej iteracji

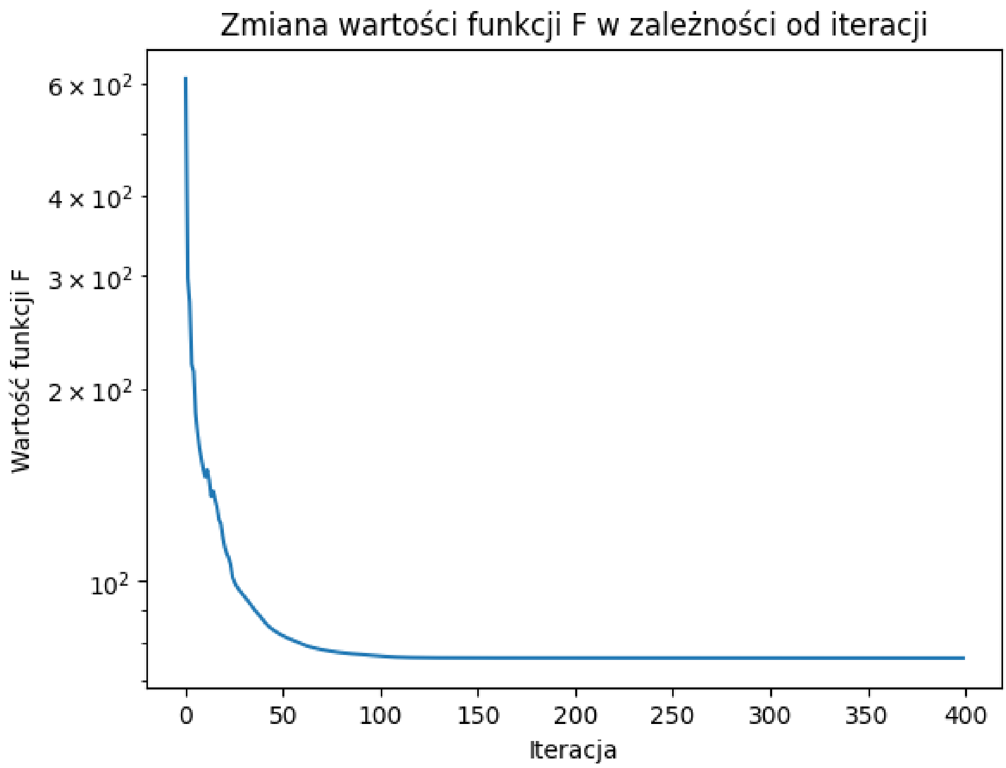
4.1.1.2. Ścieżka robota



Wykres 2. Wykres ścieżki robota w pierwszej iteracji

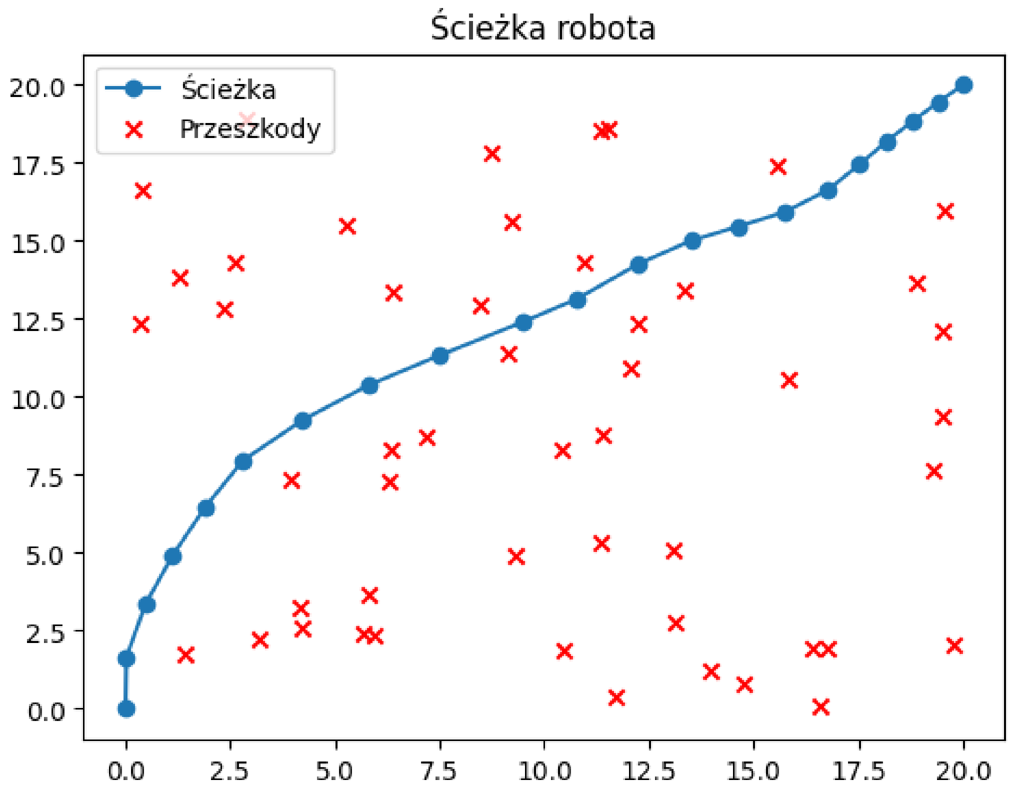
4.1.2. Druga iteracja

4.1.2.1. Zmiana wartości funkcji F w zależności od iteracji



Wykres 3. Wykres zmiany wartości funkcji F w zależności od drugiej iteracji

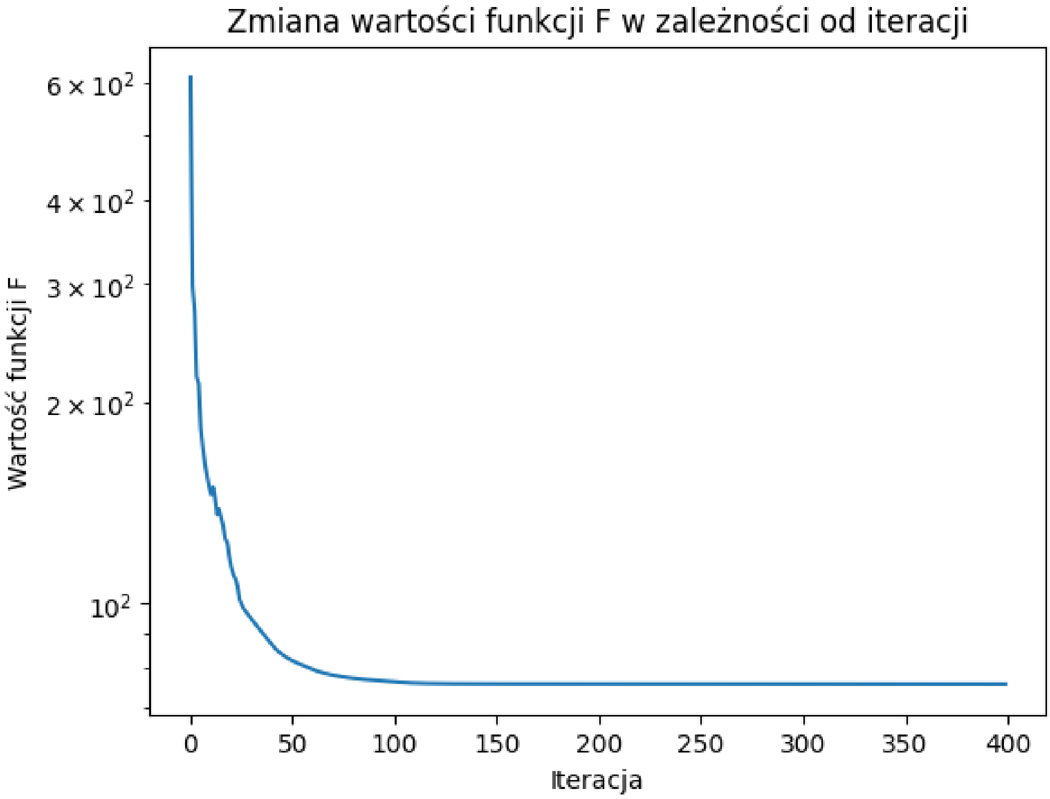
4.1.2.2. Ścieżka robota



Wykres 4. Wykres ścieżki robota w drugiej iteracji

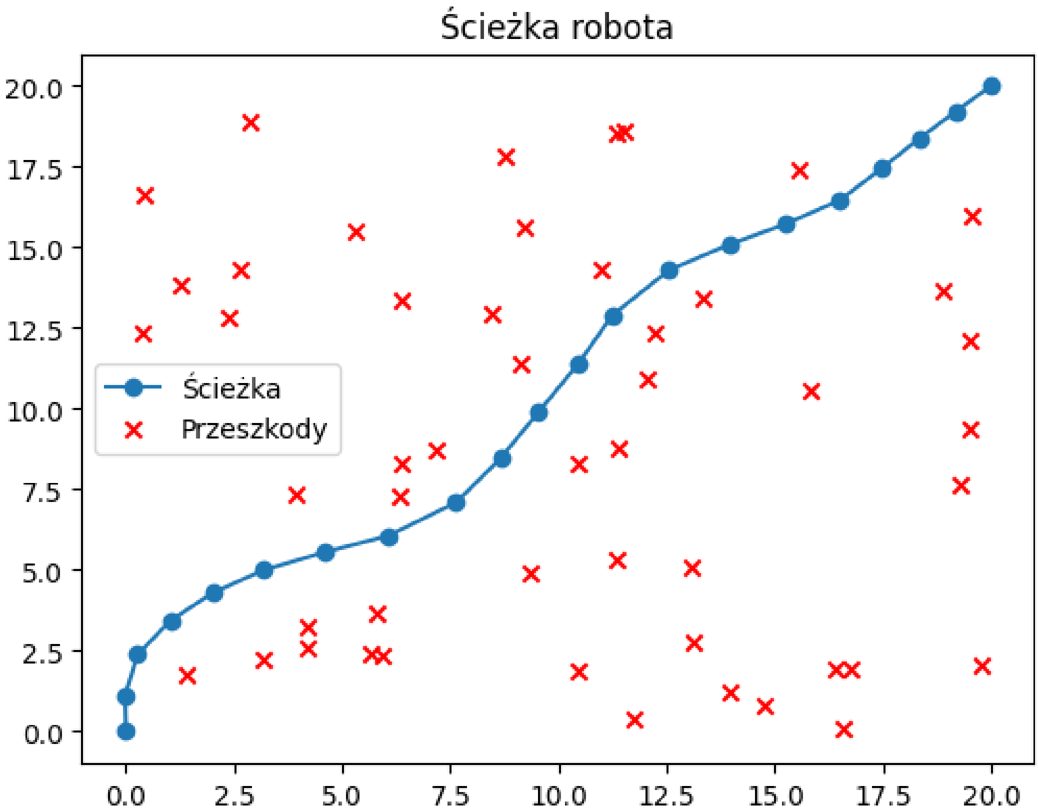
4.1.3. Trzecia iteracja

4.1.3.1. Zmiana wartości funkcji F w zależności od iteracji



Wykres 5. Wykres zmiany wartości funkcji F w zależności od trzeciej iteracji

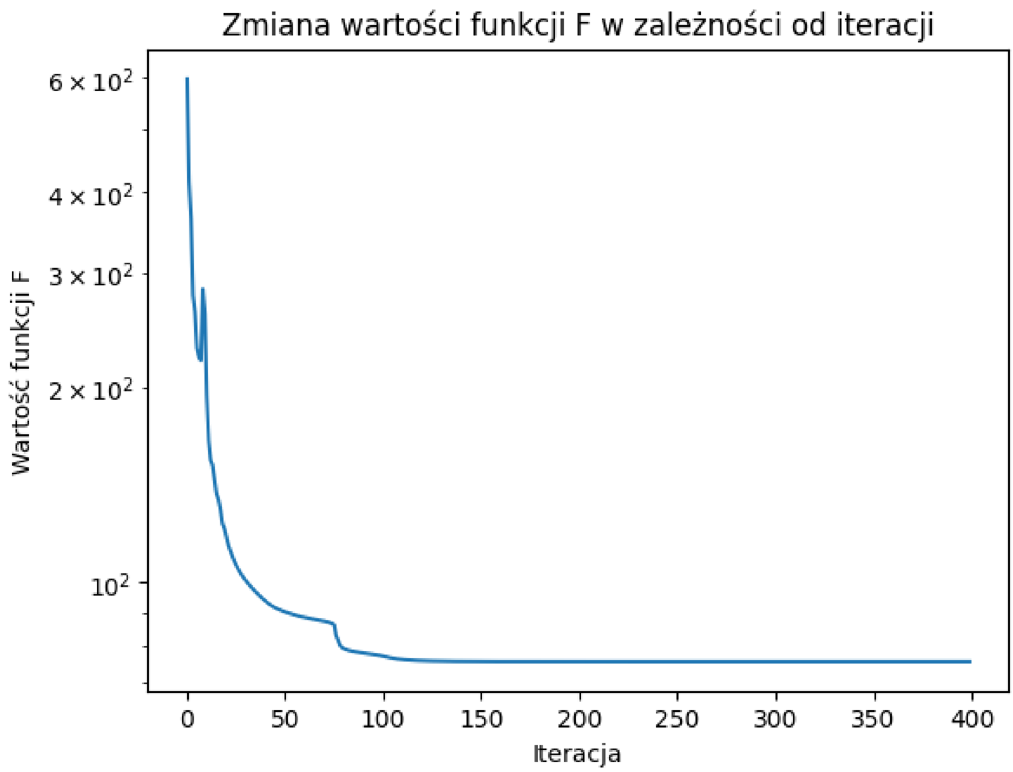
4.1.3.2. Ścieżka robota



Wykres 6. Wykres ścieżki robota w trzeciej iteracji

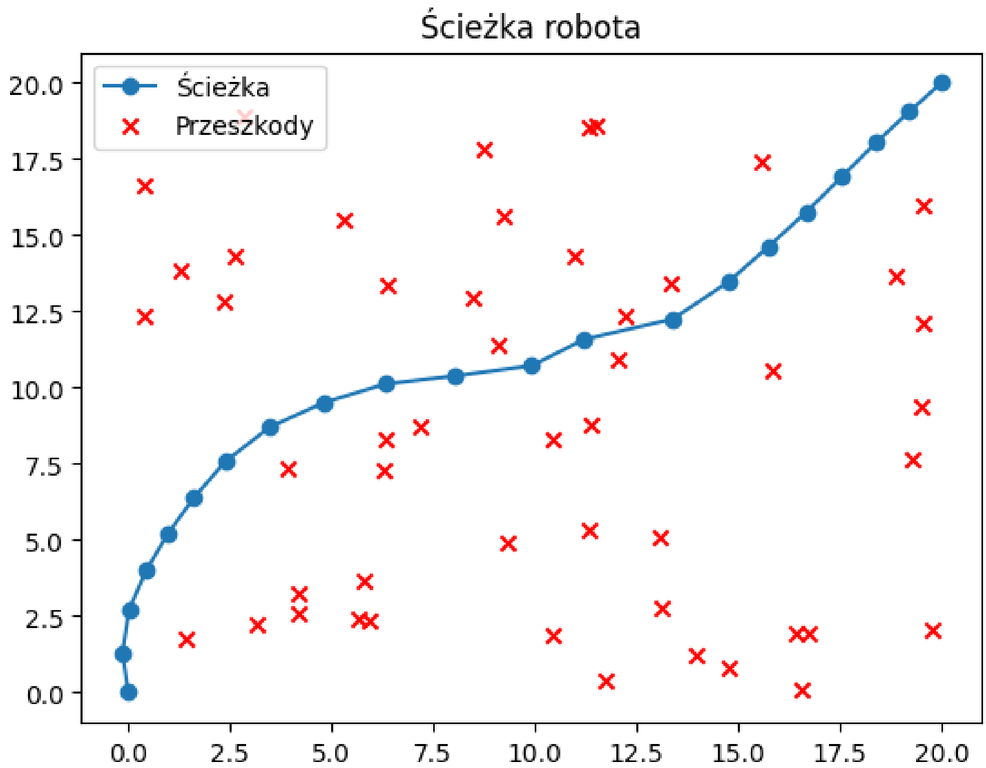
4.1.4. Czwarta iteracja

4.1.4.1. Zmiana wartości funkcji F w zależności od iteracji



Wykres 7. Wykres zmiany wartości funkcji F w zależności od czwartej iteracji

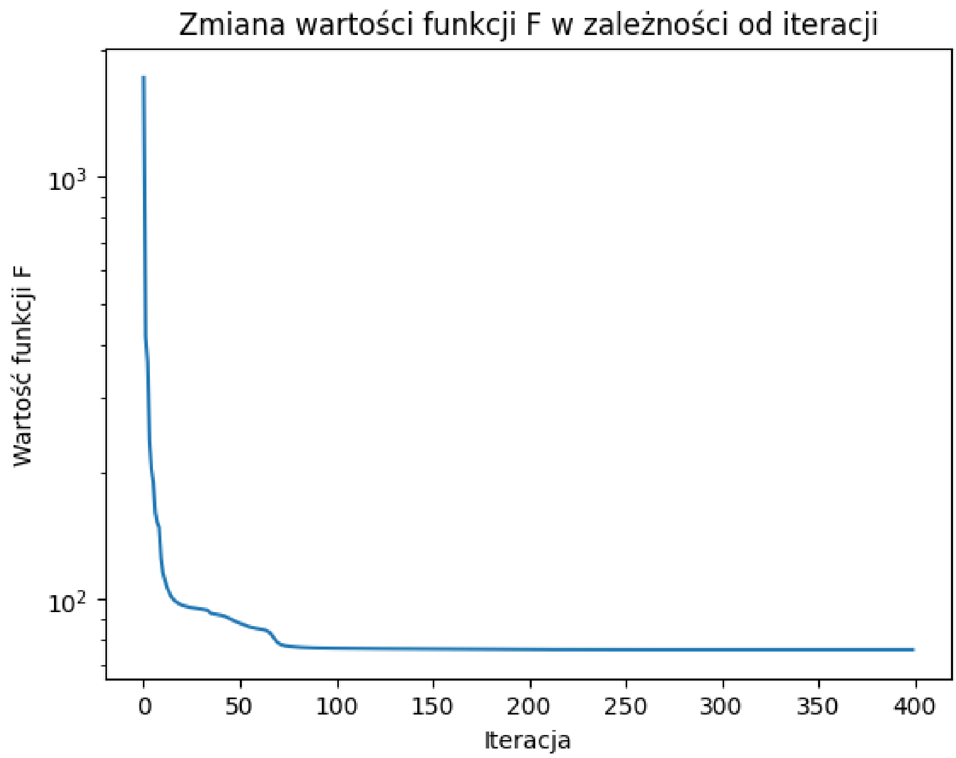
4.1.4.2. Ścieżka robota



Wykres 8. Wykres ścieżki robota w czwartej iteracji

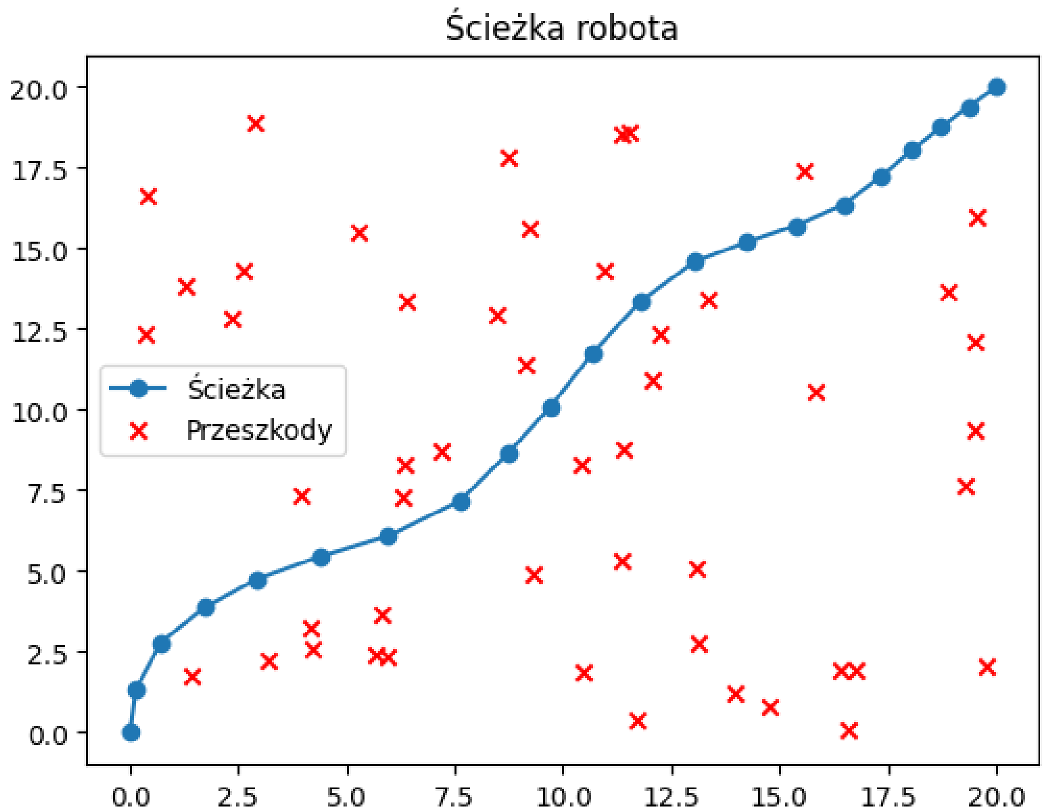
4.1.5. Piąta iteracja

4.1.5.1. Zmiana wartości funkcji F w zależności od iteracji



Wykres 9. Wykres zmiany wartości funkcji F w zależności od piątej iteracji

4.1.5.2. Ścieżka robota



Wykres 10. Wykres ścieżki robota w piątej iteracji

5. Wnioski
Metoda największego spadku wzdłuż gradientu okazała się skuteczna w nawigacji robota w środowisku z przeszkodami. Robot był w stanie znaleźć bezkolizyjną ścieżkę, minimalizując odległość do celu przy jednoczesnym omijaniu przeszkód, ale jednak, poruszamy się bardzo blisko przeszkód co przy nie których wypadkach może być niebezpieczne, kiedy chodzi o robota, który ma je unikać a on się bardzo blisko nich porusza.

Dzięki wykorzystaniu gradientu potencjału, robot dynamicznie dostosowywał swoją trasę w czasie rzeczywistym. Pozwoliło to na elastyczne reagowanie na zmieniające się warunki otoczenia oraz na skuteczne omijanie przeszkód.

Algorytm gradientu jest relatywnie mało wymagający obliczeniowo, co czyni go odpowiednim do zastosowań w czasie rzeczywistym. Robot był w stanie przetwarzać dane sensoryczne i aktualizować swoją trasę na bieżąco bez znaczących opóźnień.

Główne ograniczenie metody największego spadku wzdłuż gradientu polega na możliwości utkwienia w lokalnych minimach. W takich sytuacjach robot może nie znaleźć globalnie optymalnej trasy do celu, szczególnie w bardziej złożonych środowiskach z wieloma przeszkodami.

Podsumowując, metoda największego spadku wzdłuż gradientu jest efektywnym narzędziem do nawigacji robota w środowisku z przeszkodami, oferując zarówno prostotę implementacji, jak i dobrą wydajność w czasie rzeczywistym. Jednakże, jej skuteczność może być zwiększona poprzez integrację z innymi technikami optymalizacji, aby lepiej radzić sobie z lokalnymi minimami i bardziej złożonymi scenariuszami nawigacyjnymi.

6. Bibliografia

Wykład MOwNiT - prowadzony przez dr. Inż. K. Rycerz
Prezentacje – dr. Inż. M. Kuta

7. Dodatkowe informacje
Rozwiązania obu zadań znajdują się odpowiednio w plikach ex1.ipynb oraz ex2.ipynb