Błażej Kapkowski, Konrad Konsek 13.06.2024

**„Laboratorium” 11**

**Optymalizacja**

**Dane techniczne:**

Język: Python

Translator: Visual Studio Code

Procesor: AMD Ryzen 7 5800H

System operacyjny: Windows 11

**Realizacja ćwiczenia:**

**Zadanie 1:**

**Cel:** Wyznaczenie punktów krytycznych funkcji oraz ich klasyfikacja jako minimum, maksimum lub punkt siodłowy.

**Kroki:**

1. **Definiowanie funkcji:** W pierwszym kroku zdefiniowaliśmy funkcje matematyczne, dla których chcemy znaleźć punkty krytyczne:

f1 = x\*\*2 - 4\*x\*y + y\*\*2

f2 = x\*\*4 - 4\*x\*y + y\*\*4

f3 = 2\*x\*\*3 - 3\*x\*\*2 - 6\*x\*y\*(x - y - 1)

f4 = (x - y)\*\*4 + x\*\*2 - y\*\*2 - 2\*x + 2\*y + 1

1. **Wyznaczanie gradientu:** Gradient funkcji określa punkty, w których pochodna cząstkowa jest równa zero (punkty krytyczne). Używamy SymPy do obliczenia gradientu:

def find\_critical\_points(f):

    grad = sp.Matrix([sp.diff(f, var) for var in (x, y)])

    stationary\_points = sp.solve(grad, (x, y), dict=True)

    return stationary\_points

1. **Klasyfikacja punktów krytycznych:** Aby określić, czy punkt krytyczny jest minimum, maksimum czy punktem siodłowym, używamy macierzy Hessego. Wykorzystujemy wartości własne macierzy Hessego:

def classify\_critical\_points(f, points):

    hessian = sp.Matrix([[sp.diff(f, var1, var2) for var1 in (x, y)] for var2 in (x, y)])

    classifications = []

    for point in points:

        hessian\_at\_point = hessian.subs(point)

        det\_hessian = hessian\_at\_point.det()

        eigenvals = hessian\_at\_point.eigenvals()

        real\_eigenvals = [sp.re(val) for val in eigenvals]

        if det\_hessian == 0:

            classification = 'indeterminate due to zero determinant'

        elif all(val > 0 for val in real\_eigenvals):

            classification = 'minimum'

        elif all(val < 0 for val in real\_eigenvals):

            classification = 'maximum'

        else:

            classification = 'saddle point'

        classifications.append((point, classification))

    return classifications

1. **Zbieranie i wyświetlanie wyników:** Wyniki są zbierane i wyświetlane w formacie DataFrame:

flat\_results = []

for func\_name, points in critical\_points\_and\_classifications.items():

    for point, classification in points:

        flat\_results.append({

            'Function': func\_name,

            'Point': point,

            'Classification': classification

        })

df\_results = pd.DataFrame(flat\_results)

**Wyniki:**

Function Point Point Classification

0 Function 1 {x: 0, y: 0} saddle point

1 Function 2 {x: -1, y: -1} minimum

2 Function 2 {x: 0, y: 0} saddle point

3 Function 2 {x: 1, y: 1} minimum

4 Function 2 {x: -I, y: I} maximum

5 Function 2 {x: I, y: -I} maximum

6 Function 2 {x: sqrt(2)\*(-1 - I)/2, y: sqrt(2)/2 - sqrt(2)\*I/2} saddle point

7 Function 2 {x: sqrt(2)\*(-1 + I)/2, y: sqrt(2)/2 + sqrt(2)\*I/2} saddle point

8 Function 2 {x: sqrt(2)\*(1 - I)/2, y: -sqrt(2)/2 - sqrt(2)\*I/2} saddle point

9 Function 2 {x: sqrt(2)\*(1 + I)/2, y: -sqrt(2)/2 + sqrt(2)\*I/2} saddle point

10 Function 3 {x: -1, y: -1} maximum

11 Function 3 {x: 0, y: -1} saddle point

12 Function 3 {x: 0, y: 0} saddle point

13 Function 3 {x: 1, y: 0} minimum

14 Function 4 {x: 1, y: 1} saddle point

**Wnioski:**

 Wszystkie cztery funkcje mają różne rozmieszczenia punktów krytycznych, w tym minima, maksima i punkty siodłowe.

 Funkcje f2 i f3 mają zarówno minima, jak i maksima lokalne, co sugeruje bardziej złożoną strukturę krajobrazu funkcji w porównaniu do f1 i f4

 Funkcja f2 ma maksima w punktach zespolonych (−i, i) oraz (i, −i) W rzeczywistej przestrzeni sugeruje to, że funkcja może mieć oscylacyjne zachowanie w okolicach tych punktów.

 Punkty zespolone, które są punktami siodłowymi, wskazują na bardziej złożony krajobraz funkcji. Nawet w przestrzeni rzeczywistej, analiza tych punktów może pomóc zrozumieć zachowanie funkcji w okolicach rzeczywistych punktów krytycznych.

**Zadanie 2:**

**Cel:** Znalezienie najkrótszej ścieżki robota pomiędzy dwoma punktami przy uwzględnieniu przeszkód za pomocą metody największego spadku.

**Kroki:**

1. **Definiowanie parametrów:** Ustaliliśmy parametry, takie jak liczba odcinków, liczba przeszkód, punkty startowe i końcowe, itp.:

n = 20

k = 50

x\_start = np.array([0, 0])

x\_end = np.array([20, 20])

r = np.random.uniform(0, 20, (k, 2))

lambda\_1 = 1

lambda\_2 = 1

epsilon = 1e-13

iterations = 400

1. **Inicjalizacja punktów ścieżki:** Punkty ścieżki są inicjalizowane liniowo pomiędzy punktami startowymi i końcowymi:

x = np.linspace(x\_start, x\_end, n+1)

x[0] = x\_start

x[-1] = x\_end

1. **Definiowanie funkcji celu:** Funkcja celu składa się z dwóch części: oddziaływanie przeszkód oraz długość ścieżki:

def F(x, r, lambda\_1, lambda\_2, epsilon):

    term1 = lambda\_1 \* np.sum([1 / (epsilon + np.linalg.norm(x[i] - r[j])\*\*2) for i in range(n+1) for j in range(k)])

    term2 = lambda\_2 \* np.sum([np.linalg.norm(x[i+1] - x[i])\*\*2 for i in range(n)])

    return term1 + term2

1. **Gradient funkcji celu:** Gradient funkcji celu jest niezbędny do algorytmu największego spadku:

def grad\_F(x, r, lambda\_1, lambda\_2, epsilon):

    grad = np.zeros\_like(x)

    for i in range(1, n):

        grad\_term1 = lambda\_1 \* np.sum([-2 \* (x[i] - r[j]) / (epsilon + np.linalg.norm(x[i] - r[j])\*\*2)\*\*2 for j in range(k)], axis=0)

        grad\_term2 = lambda\_2 \* (2 \* (x[i] - x[i-1]) - 2 \* (x[i+1] - x[i]))

        grad[i] = grad\_term1 + grad\_term2

    return grad

1. **Metoda złotego podziału:** Wykorzystujemy metodę złotego podziału do przeszukiwania liniowego:

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

1. **Algorytm największego spadku:** Implementujemy algorytm największego spadku:

def gradient\_descent(x, r, lambda\_1, lambda\_2, epsilon, iterations):

    alpha = 1

    values = []

    for \_ in range(iterations):

        grad = grad\_F(x, r, lambda\_1, lambda\_2, epsilon)

        f = lambda a: F(x - a \* grad, r, lambda\_1, lambda\_2, epsilon)

        alpha = golden\_section\_search(f, 0, alpha)

        x -= alpha \* grad

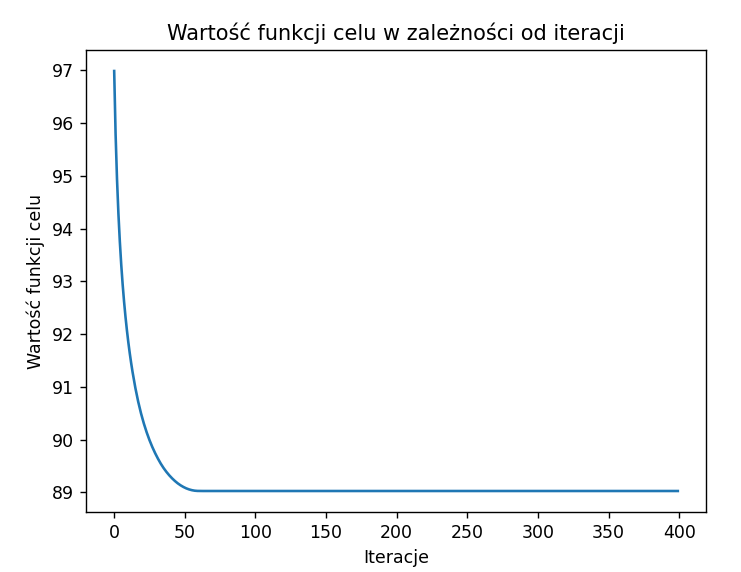
        values.append(F(x, r, lambda\_1, lambda\_2, epsilon))

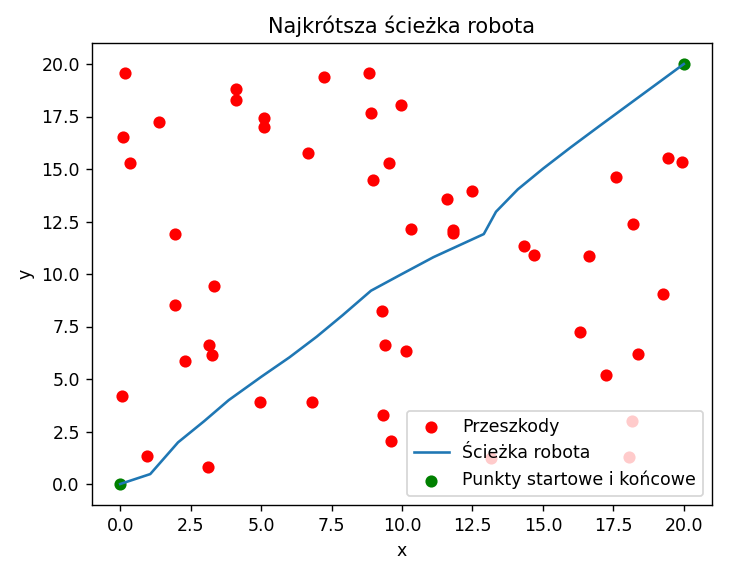
    return x, values

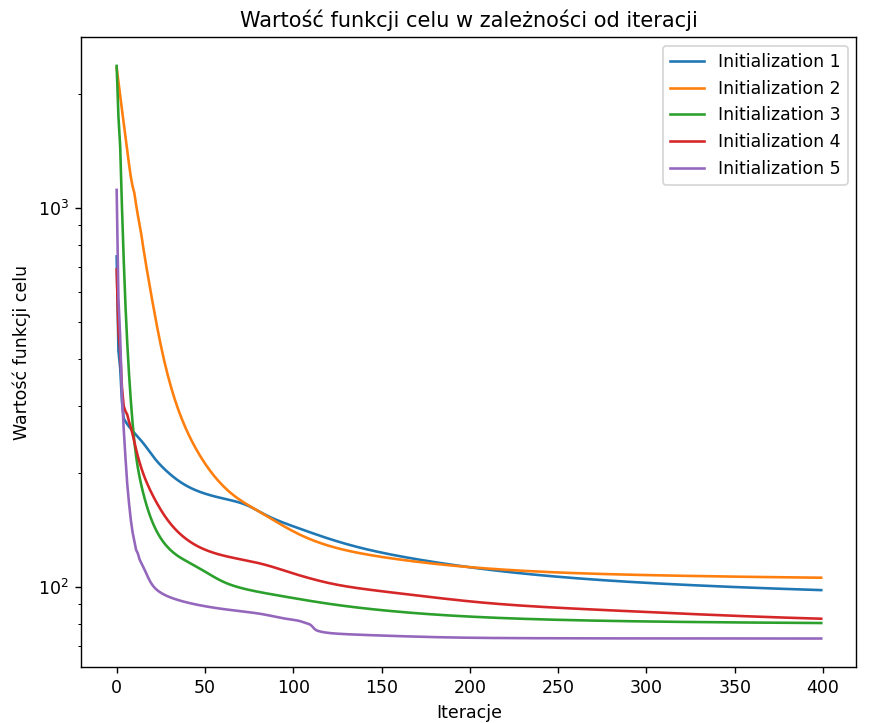
1. **Znajdowanie najkrótszej ścieżki:** Przeprowadzamy obliczenia dla pięciu różnych losowych inicjalizacji punktów wewnątrz ścieżki:

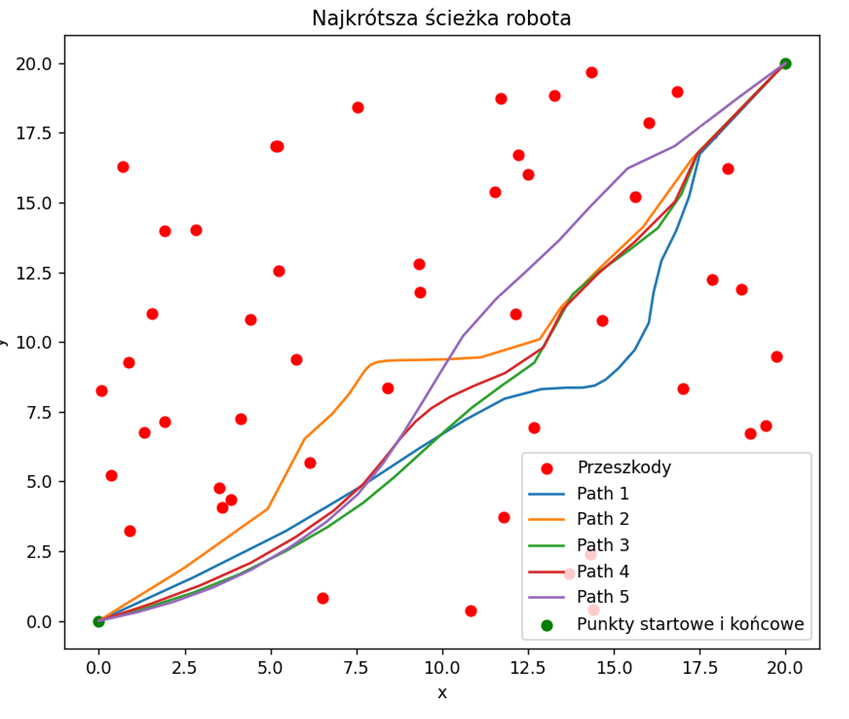
x\_opt, values = gradient\_descent(x, r, lambda\_1, lambda\_2, epsilon, iterations)

**Wyniki:**

****

****

****

****

**Wnioski:**

#### **Wartość funkcji celu w zależności od iteracji**

* **Szybki spadek na początku**: Na początku algorytmu widzimy szybki spadek wartości funkcji celu, co wskazuje na to, że algorytm szybko zbliża się do optymalnego rozwiązania.
* **Stabilizacja wartości**: Po około 50 iteracjach wartość funkcji celu stabilizuje się, co oznacza, że algorytm zbliżył się do lokalnego minimum i dalsze iteracje nie przynoszą znaczącej poprawy.
* **Końcowa wartość funkcji celu**: Ostateczna wartość funkcji celu po 400 iteracjach wynosi około 89. Wskazuje to na osiągnięcie stabilnego rozwiązania, które minimalizuje zarówno długość trasy, jak i unikanie przeszkód.

**Analiza trasy robota**

* **Efektywne omijanie przeszkód**: Trasa robota skutecznie omija przeszkody, co jest wynikiem uwzględnienia tych przeszkód w funkcji celu.
* **Gładka ścieżka**: Trasa jest względnie gładka, co wskazuje na to, że algorytm z powodzeniem minimalizował długość trasy, jednocześnie unikając gwałtownych zmian kierunku.