# Praca domowa 3
# Porównanie metody gradientowej z symulowanym wyżarzaniem oraz szukaniem przypadkowym

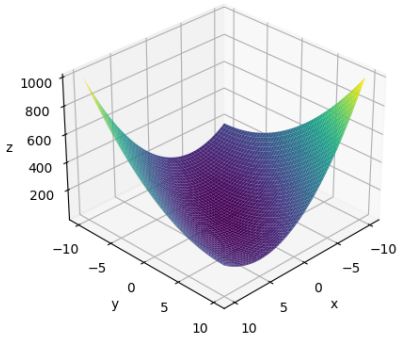
Damian Jankowski s188597

23 kwietnia 2023

## 1 Wstęp

Celem pracy domowej jest porównanie trzech metod szukania minimum funkcji: metody gradientowej, symulowanego wyżarzania oraz z szukaniem przypadkowym. W tym celu wybrałem funkcję $f(x, y) = 4x^2 + 2 * y^2 - 4xy$. Minimum funkcji to $f(0, 0) = 0$.

Poszukiwanie minimum funkcji zaczęłem od wybrania losowego punktu startowego $X_0$ z zakresu $-10 \leq x \leq 10$ oraz $-10 \leq y \leq 10$.




Rysunek 1: Wykres funkcji $f(x, y)$

## 2 Metoda symulowanego wyżarzania

Metoda symulowanego wyżarzania polega na losowaniu kolejnych punktów z pewnego otoczenia punktu $X_k$ i sprawdzaniu czy wartość funkcji w tym punkcie jest mniejsza niż w poprzednim. Jeśli tak to punkt $X_{k+1}$ staje się nowym punktem startowym. Jeśli nie to punkt $X_{k+1}$ jest losowany ponownie. Wraz z kolejnymi iteracjami otoczenie punktu $X_k$ zmniejsza się. W ten sposób metoda symulowanego wyżarzania przeszukuje coraz mniejsze obszary wokół punktu $X_k$.

### 2.1 Opis kroków

1. Wybranie losowego punktu startowego $X_0$
2. Wyznaczenie wartości funkcji $f(X_k)$
3. Wyznaczenie nowego punktu $w' = w + \Delta w$ z otoczenia punktu $X_k$, gdzie $\Delta w$ jest realizacją zmiennej losowej o rozkładzie normalnym $g(\Delta w, T) = (2\pi T)^{-\frac{n}{2}} e^{-\frac{\Delta w^2}{2T}}$

4. Wyznaczenie wartości funkcji $f(w')$
5. Podstawienie $w'$ za $w$ jeśli $f(w') < f(w)$ lub gdy $r < \frac{1}{1+e^{\frac{\Delta f}{T}}}$, gdzie $r$ jest realizacją zmiennej losowej o rozkładzie jednostajnym na przedziale $[0, 1]$
6. Zmniejszenie temperatury $T' = \alpha T$
7. Zwiększenie licznika iteracji $k = k + 1$. Zakończenie gdy $k = k_{max}$ lub gdy $f(X_k) < f_{min}$

# 3 Metoda szukania przypadkowego

Metoda szukania przypadkowego podobnie jak metoda symulowanego wyżarzania polega na losowaniu kolejnych punktów z pewnego otoczenia punktu $X_k$. W tym przypadku zakres losowania jest stały i nie zmniejsza się wraz z kolejnymi iteracjami.

## 3.1 Opis kroków

1. Wybranie losowego punktu startowego $X_0$
2. Wyznaczenie wartości funkcji $f(X_k)$
3. Wyznaczenie nowego punktu $w' = w + \Delta w$ z otoczenia punktu $X_k$, gdzie $\Delta w$ jest realizacją zmiennej losowej o rozkładzie normalnym $g(\Delta w, \sigma) = (2\pi\sigma)^{-\frac{n}{2}} e^{-\frac{\Delta w^2}{2\sigma}}$ (w tym przypadku $\sigma = const$)
4. Wyznaczenie wartości funkcji $f(w')$
5. Podstawienie $w'$ za $w$ jeśli $f(w') < f(w)$
6. Zwiększenie licznika iteracji $k = k + 1$. Zakończenie gdy $k = k_{max}$ lub gdy $f(X_k) < f_{min}$

# 4 Zadanie

W przypadku symulowanego wyżarzania przyjąłem:

- $T_0 = 10$
- $\alpha = 0.99$

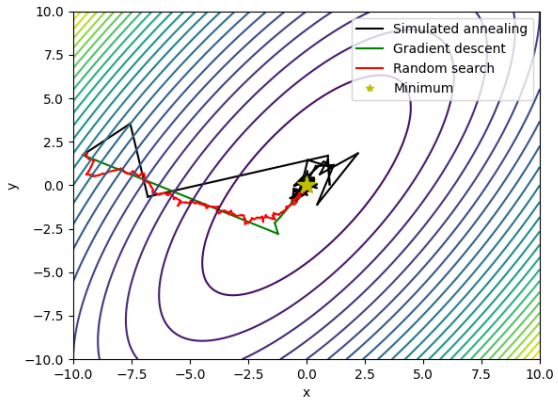
Natomiast dla metody szukania przypadkowego:

- $\sigma = 0.1$

Wszystkie 3 metody kończyły się gdy wartość funkcji w punkcie $X_k$ była mniejsza niż $f_{min} = 10^{-4}$ lub gdy liczba iteracji osiągnęła wartość $k_{max} = 10000$.

Symulację przeprowadziłem dla 1000 losowo wybranych punktów startowych. Średnia liczba iteracji potrzebna do osiągnięcia wartości funkcji mniejszej niż $f_{min}$ wyniosła:

- Metoda gradientowa: 34.729
- Metoda symulowanego wyżarzania: 764.728
- Metoda szukania przypadkowego: 647.866




Rysunek 2: Rzut 2D funkcji $f(x, y)$ oraz kolejne kroki minimalizacji metodą gradientową, z wyżarzaniem oraz z szukaniem przypadkowym

# 5 Wnioski

Wyniki symulacji pokazują, że metoda gradientowa jest znacznie szybsza od pozostałych metod. Metoda symulowanego wyżarzania potrzebowała średnio 22 razy więcej iteracji niż metoda gradientowa.

W porównaniu metod gradientowych, wyżarzania oraz szukania przypadkowego można wyróżnić kilka wniosków:

Metoda gradientowa jest najskuteczniejsza, gdyż pozwala na znalezienie globalnego minimum, jeśli funkcja jest różniczkowalna i nie ma zbyt wiele minimów lokalnych. Metoda ta działa dobrze dla prostych funkcji, ale może mieć problemy z funkcjami nieliniowymi, gdzie minimum globalne znajduje się w dolinie.

Metoda wyżarzania jest skuteczna, ale wymaga więcej czasu niż metoda gradientowa. W przeciwieństwie do metody gradientowej, metoda wyżarzania nie zawsze znajduje globalne minimum, ale może znaleźć rozwiązanie optymalne dla funkcji, które mają wiele minimów lokalnych. Metoda ta działa dobrze, jeśli funkcja jest nieregularna i ma wiele lokalnych minimów.

Metoda szukania przypadkowego jest najmniej skuteczna, ale jest najprostsza do zastosowania. Ta metoda działa dobrze dla funkcji, które mają wiele minimów lokalnych, ale może trwać bardzo długo, zanim znajdzie się globalne minimum. W rzeczywistości, jeśli funkcja ma więcej niż kilka wymiarów, szansa na znalezienie globalnego minimum jest bardzo niska. W związku z powyższymi wnioskami, wybór odpowiedniej metody optymalizacji zależy od charakterystyki funkcji, której minimum poszukujemy, a także od czasu, jaki mamy na wykonanie obliczeń.

W przypadku, gdy mamy do czynienia z funkcjami nieregularnymi i wieloma minimami lokalnymi, a czas nie jest czynnikiem decydującym, zaleca się zastosowanie metody wyżarzania. Natomiast, gdy czas jest ważny, a funkcja jest różniczkowalna i nie ma wielu minimów lokalnych, najlepiej wykorzystać metodę gradientową.

Metoda szukania przypadkowego może być przydatna, gdy mamy do czynienia z funkcją, która ma wiele minimów lokalnych, ale nie jest to zalecane dla złożonych funkcji wielowymiarowych.

# 6 Kod programu

```
import numpy as np
import matplotlib.pyplot as plt

RANGE = 10


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


def gradient(x, y):
```

```
    return np.array([8 * x - 4 * y, 4 * y - 4 * x])


def gradient_descent(gradient, w0):
    w = w0
    w_hist = [w0]
    k = 1
    while True:
        w = w - 0.1 * gradient(w[0], w[1])
        w_hist.append(w)
        if f(*w) < 1e-4 or k > 10000:
            break
        k += 1
    return w_hist, k


def simulated_annealing(f, w0, Tmax, n):
    w = w0
    k = 1
    f_value = f(*w)
    w_hist = [w]
    T = Tmax
    while True:
        delta_w = np.random.normal(0, T, len(w))
        w_prime = w + delta_w
        f_prime = f(*w_prime)
        delta_f = f_prime - f_value
        if delta_f < 0 or np.random.rand() < 1 / (1 + np.exp(delta_f / T)):
            w, f_value = w_prime, f_prime
            w_hist.append(w)
        T *= n
        if f(*w) < 1e-4 or k > 10000:
            break
        k += 1
    return w_hist, k


def random_search(f, w0, sigma):
    w = w0
    k = 1
    f_value = f(*w)
    w_hist = [w]
    while True:
        delta_w = np.random.normal(0, sigma, len(w))
        w_prime = w + delta_w
        f_prime = f(*w_prime)
        delta_f = f_prime - f_value
        if delta_f < 0:
            w = w_prime
            f_value = f_prime
            w_hist.append(w)
        if f(*w) < 1e-4 or k > 10000:
            break
        k += 1
    return w_hist, k


x = np.linspace(-RANGE, RANGE, 100)
y = np.linspace(-RANGE, RANGE, 100)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis', edgecolor='none')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
ax.view_init(30, 45)
plt.savefig('function.png')
plt.show()

k1_hist = []
k2_hist = []
k3_hist = []

w_0 = np.array(np.array([
    np.random.uniform(-RANGE, RANGE),
    np.random.uniform(-RANGE, RANGE)]))

w_hist, k1 = simulated_annealing(f, w_0, 10, 0.99)
w_hist2, k2 = gradient_descent(gradient, w_0)
w_hist3, k3 = random_search(f, w_0, 0.1)

k1_hist.append(k1)
k2_hist.append(k2)
k3_hist.append(k3)

for i in range(len(w_hist) - 1):
    plt.plot([w_hist[i][0], w_hist[i + 1][0]],
             [w_hist[i][1], w_hist[i + 1][1]], 'k-')
for i in range(len(w_hist2) - 1):
    plt.plot([w_hist2[i][0], w_hist2[i + 1][0]],
             [w_hist2[i][1], w_hist2[i + 1][1]], 'g-')
for i in range(len(w_hist3) - 1):
    plt.plot([w_hist3[i][0], w_hist3[i + 1][0]],
             [w_hist3[i][1], w_hist3[i + 1][1]], 'r-')


print('Simulated annealing: ', np.mean(k1_hist))
print('Gradient descent: ', np.mean(k2_hist))
print('Random search: ', np.mean(k3_hist))

cp = plt.contour(X, Y, Z, 25)
plt.xlabel('x')
plt.ylabel('y')

plt.plot(0, 0, 'y*', markersize=15)

plt.plot([], [], 'k-', label='Simulated annealing')
plt.plot([], [], 'g-', label='Gradient descent')
plt.plot([], [], 'r-', label='Random search')
plt.plot([], [], 'y*', label='Minimum')
plt.legend()

plt.savefig('plot.png')
plt.show()
```