Laboratorium nr 5		Data 10.01.25r.
Temat:	**Optymalizacja wielokryterialna**	
Wykonanie:	Wojciech Zacharski, Karol Woda, Paweł Socała	

Sprawozdanie:

1. Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z problematyką optymalizacji wielokryterialnej oraz wyznaczenie rozwiązań minimalnych w sensie Pareto. W ramach zadania przeprowadzono optymalizację dla testowych funkcji celu oraz rozwiązano problem rzeczywisty dotyczący belki obciążonej siłą.

---

2. Testowe funkcje celu

Testowe funkcje celu zostały zdefiniowane jako:

$$f_1(x_1, x_2) = a \cdot ((x_1 - 2)^2 + (x_2 - 2)^2)$$

$$f_2(x_1, x_2) = {}^{1}/_{\mathrm{a}} \cdot ((x_1 + 2)^2 + (x_2 + 2)^2)$$

gdzie parametr a przyjmował wartości: $a = 1$, $a = 10$, $a = 100$. Punkt startowy został losowo wybrany z przedziałów:

$$x_1(0) \in [-10,10],\ x_2(0) \in [-10,10].$$

---

3. Problem rzeczywisty

Dla problemu rzeczywistego analizowano belkę o przekroju kołowym i długości l oraz średnicy d, obciążoną siłą $P = 1\,kNP$. Funkcje celu definiowały:

- Masę belki:

$$f_{\mathrm{m}} = \rho \cdot \pi \cdot \left(\frac{d}{2}\right)^2 \cdot l$$

- Ugięcie belki:

$$f_{\mathrm{u}} = \frac{64 \cdot P \cdot l^3}{3 \cdot E \cdot \pi \cdot d^4}$$

Dodatkowe ograniczenia to:

- Maksymalne ugięcie $u_{max} = 5\,mm$,
- Maksymalne naprężenie $\sigma_{max} = 300\,MPa$

Przedziały zmiennych optymalizacyjnych:

$$l \in [200\,mm, 1000\,mm],\ d \in [10\,mm, 50\,mm].$$

---

4. Algorytmy optymalizacji

Problemy wielokryterialne sprowadzono do jednokryterialnych, stosując metodę kryterium ważonego:

$$f(x) = w \cdot f_1(x) + (1 - w) \cdot f_2(x) + \text{penalty}$$

Do wyznaczenia minimum funkcji celu wykorzystano:

- **Metodę Powella** – do optymalizacji wielokryterialnej,
- **Metodę ekspansji** – do ustalenia przedziału początkowego wzdłuż pojedynczego kierunku,
- **Metodę złotego podziału** – do minimalizacji wzdłuż kierunku.

Ograniczenia uwzględniono poprzez zastosowanie funkcji kary zewnętrznej:

$$S(x) = \sum_{\mathrm{j}=1}^{\mathrm{k}} (max(0, g_{\mathrm{j}}(x)))^2 + \sum_{\mathrm{j}=1}^{\mathrm{m}} (h_{\mathrm{j}}(x))^2$$

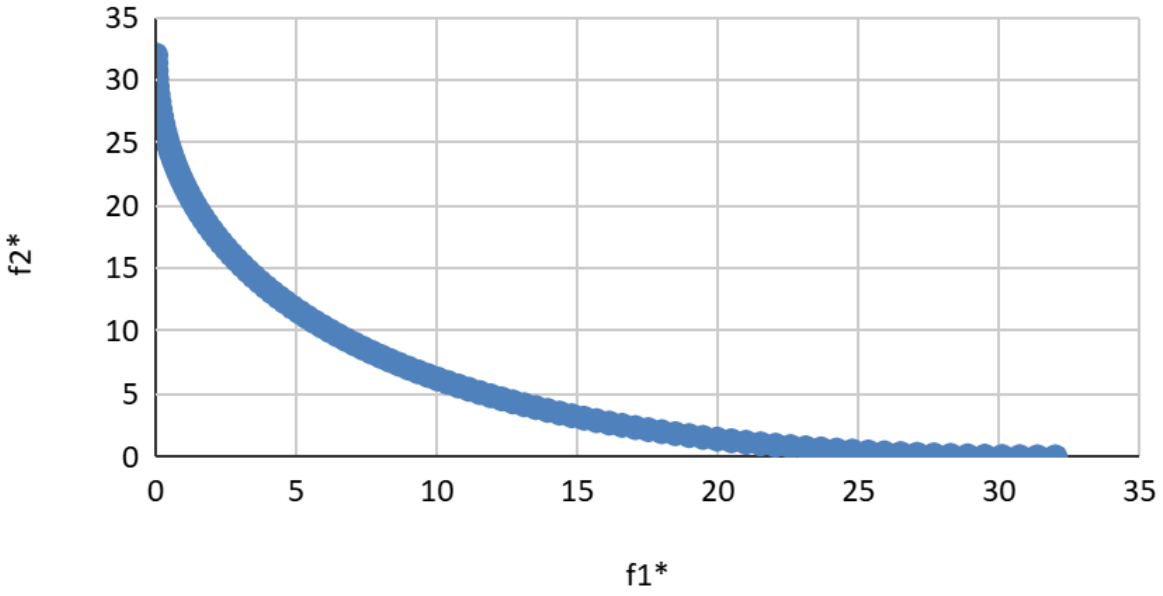
---

5. Wyniki

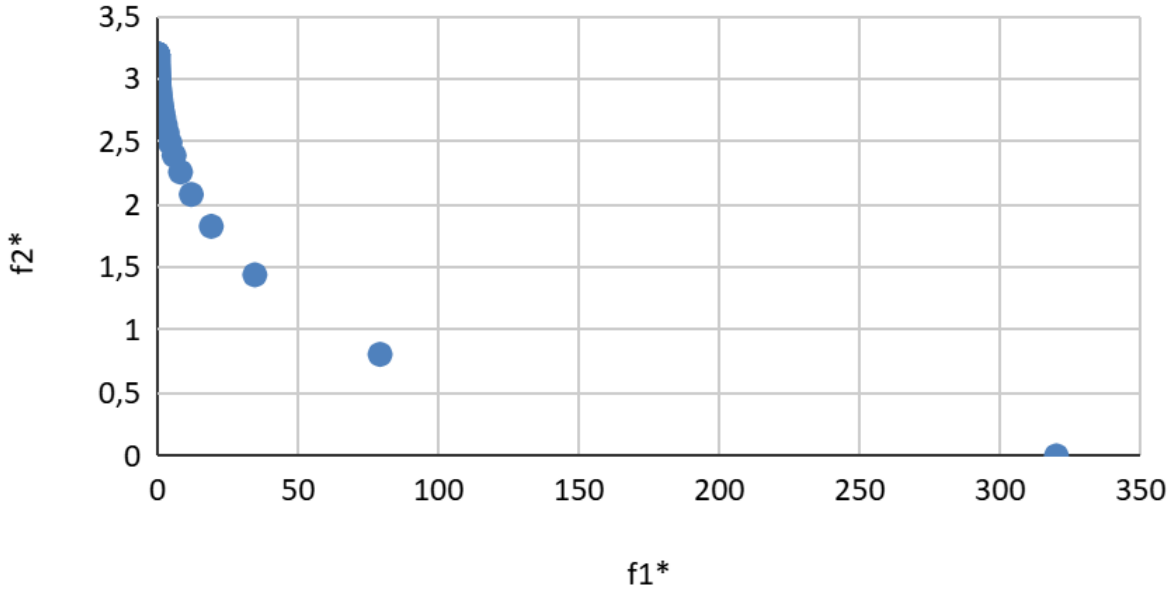
a. Testowe funkcje celu

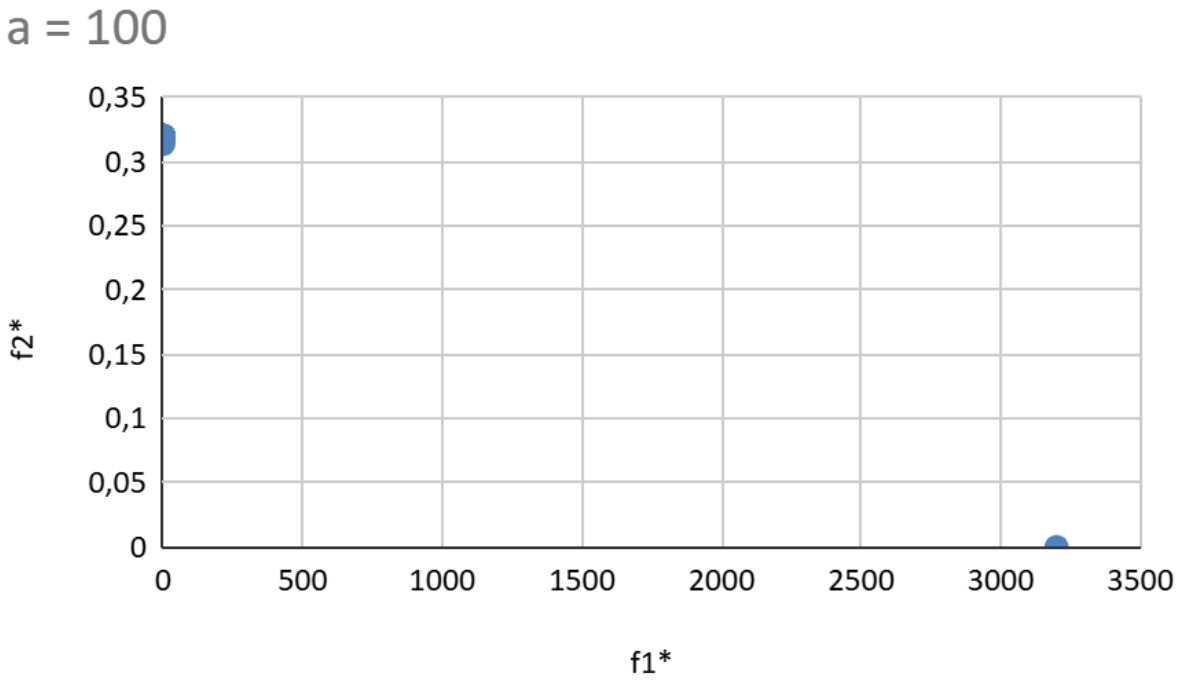
- Przeprowadzono 101 optymalizacji dla każdej wartości a i różnych wartości w.
- Wyniki zestawiono w tabeli (plik xlsx), wskazując optymalne wartości dla każdej konfiguracji.
- Na podstawie wyników wykreślono wykresy rozwiązań minimalnych dla każdego a.
- Na wykresach zauważamy wyraźnie zarysowane krzywe rozkładu Pareto:

a = 1


35
30
25
20
15
10
5
0
f2*
0
5
10
15
20
25
30
35
f1*

a = 10

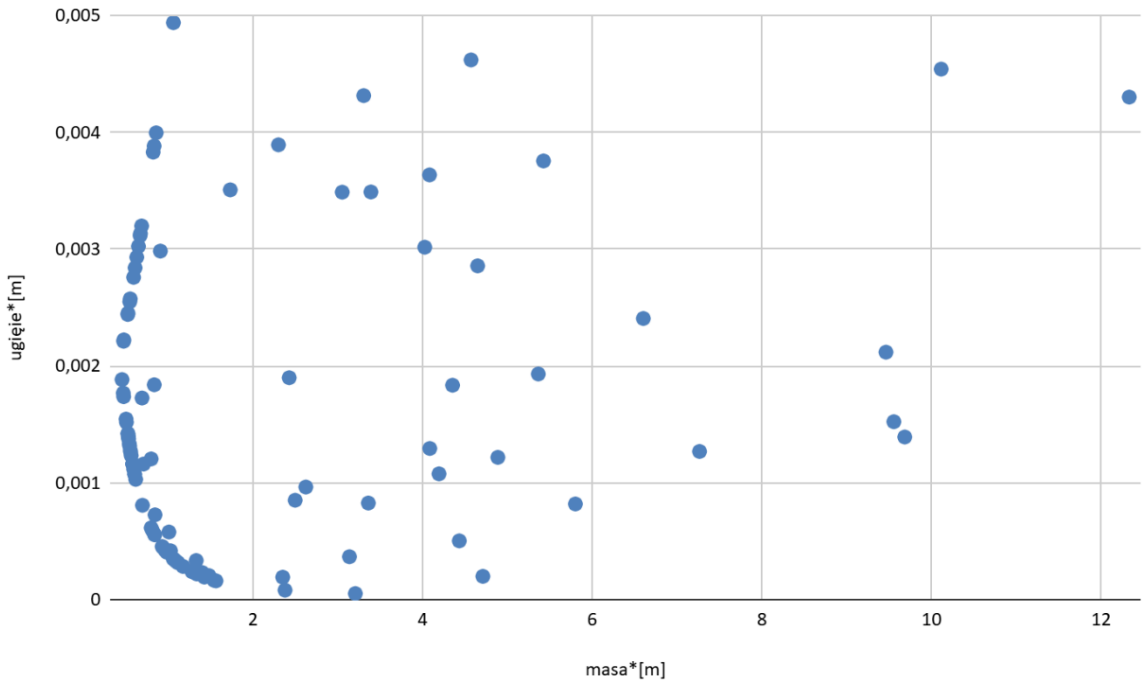

3,5
3
2,5
2
1,5
1
0,5
0
f2*
0
50
100
150
200
250
300
350
f1*



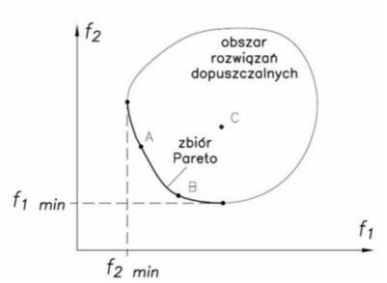
Występowanie krzywych Pareto na wykresach funkcji testowych wskazuje na poprawność implementacji metod optymalizacyjnych

b. Problem rzeczywisty

- Przeprowadzono 101 optymalizacji, dokumentując wartości.
- Wykresy rozwiązań minimalnych:



Na przedstawionym wykresie widać, że większa część wyników układa się na krzywej Pareto. Odstępstwa od krzywej i tak mieszczą się w większości w zakresie wyników dopuszczalnych:



Na tej podstawie stwierdzamy poprawność działania programu i przeprowadzonej optymalizacji.

---

6. Wnioski

1. Optymalizacja wielokryterialna pozwala zrozumieć kompromisy między różnymi funkcjami celu.
2. Front Pareto wskazuje zestaw rozwiązań, które są optymalne w sensie Pareto, tj. nie ma możliwości poprawy jednej funkcji celu bez pogorszenia drugiej.
3. Stosowane metody (Powella, złotego podziału, ekspansji) skutecznie wyznaczały minima przy uwzględnieniu ograniczeń.

---

7. Kod zaimplementowanych metod:

- Plik nagłówkowy:

```
#pragma once
#include <iostream>
#include <random>
#include <cmath>
#include <vector>
#include <functional>
#include <fstream>

// Stałe globalne
extern const double P = 1000.0;                    // [N] Siła działająca
extern const double E = 207e9;                     // [Pa] Moduł Younga
extern const double rho = 7800.0;                  // [kg/m^3] Gęstość materiału
extern const double u_max = 5e-3;                  // [m] Maksymalne ugięcie
extern const double sigma_max = 300e6;             // [Pa] Maksymalne naprężenie
extern const double PI = 3.14159265358979323846;   // Liczba PI

extern int function_call_count = 0;                // Licznik wywołań funkcji
celu
extern const double l_min_g = 0.2;  // [m]
extern const double l_max_g = 1.0;  // [m]
extern const double d_min_g = 0.01; // [m]
extern const double d_max_g = 0.05; // [m]

// ==========================   [ Metody obliczeniowe ] ==========================
// Znajdowanie przedziału występowania minimum funkcji liniowej - metoda
ekspansji
```

```
std::pair<double, double> metoda_ekspansji(const std::function<double(double)>&
f, double x1, double x2, double alfa, int N_max);

// Znajdowanie minimum funkcji liniowej - metoda złotego podziału
double metoda_zlotego_podzialu(const std::function<double(double)>& f, double a,
double b, int N_max, const double epsilon);
// Znajdowanie optymalnego minimum funkcji - metodą Powella
std::vector<double> metoda_Powella_v2(const std::function<double(double,
std::vector<double>, double)>& f, std::vector<double> x_0, double w, double c,
int N_max, const double epsilon);
// ================================================================================

// ====================   [ Funkcje problemu testowego ]   =======================

// Funkcja pierwsza, wg dostarczonego wzoru
double f1(double x1, double x2, double a);

// Funkcja druga, wg dostarczonego wzoru
double f2(double x1, double x2, double a);

// Funkcja ważona problemu testowego
double problem_jednokryterialny2(double w, std::vector<double> x, double a);

// Wywołanie optymalizacji dla problemu rzeczywistego
void problem_testowy();

// ================================================================================

// ==================   [ Funkcje problemu rzeczywistego ]   ====================

// Funkcja obliczająca masę
double f_m(double l, double d);

// Funkcja obliczająca ugięcie
double f_u(double l, double d);

// Funkcja obliczająca naprężenie
double sigma(double l, double d);

// Sprawdzanie czy są spełnione zadane ograniczenia
bool sprawdzanie_ograniczen(double l, double d);

// Funkcja obliczająca karę zewnętrzną
double funkcja_kary(double l, double d, double c);

// Funkcja ważona problemu rzeczywistego
double funkcja_wazona(double w, std::vector<double> x, double c);

// Wywołanie optymalizacji dla problemu rzeczywistego
void problem_rzeczywisty();

// ================================================================================

// ====================   [ Dodatkowe metody pomocnicze ]   =======================

// Generowanie wektora o długości ilosc ze zmiennoprzecinkowymi wartościami
losowymi z przedziału [a, b]
std::vector<double> generowanie_wektora_wartosci_losowych(double a, double b, int
ilosc);

// Generowanie zmiennoprzecinkowej wartości losowej z przedziału [a, b]
double generowanie_wartosci_losowej(double a, double b);
```

```
// ============================================================================
```

- Plik główny:

```
#include "Main.h"


// ========================   [ Metody obliczeniowe ]   ========================
// Znajdowanie przedziału występowania minimum funkcji liniowej - metoda
ekspansji
std::pair<double, double> metoda_ekspansji(const std::function<double(double)>&
f, double x1, double x2, double alfa, int N_max = 1000) {
    int i = 0;
    double x_poprzedni = NULL, x_obecny = x1, x_nastepny = x2;
    double f_obecny = f(x_obecny);
    double f_nastepny = f(x_nastepny);
    if (f_obecny == f_nastepny) return { x_obecny, x_nastepny };
    else if (f_obecny < f_nastepny) {
        x_poprzedni = x_nastepny;
        x_nastepny = -x_poprzedni;
        f_nastepny = f(x_nastepny);
        if (f_nastepny >= f_obecny) return { x_poprzedni, x_nastepny };
    }
    while (f_obecny > f_nastepny)
    {
        x_poprzedni = x_obecny;
        x_obecny = x_nastepny;
        f_obecny = f_nastepny;
        if (i > N_max)
        {
            //throw std::runtime_error("Nie udało się ustalić przedziału w N_max
krokach!");
            std::cout << "[Ekspansja] Nie udalo sie ustalic przedzialu w N_max
krokach!" << std::endl;
            return { NULL, NULL };
        }
        i++;
        x_nastepny = x_obecny * alfa;
        f_nastepny = f(x_nastepny);
    }
    if (x_poprzedni < x_nastepny) return { x_poprzedni, x_nastepny };
    else return { x_nastepny, x_poprzedni };
}

// Znajdowanie minimum funkcji liniowej - metoda złotego podziału
double metoda_zlotego_podzialu(const std::function<double(double)>& f, double a,
double b, int N_max = 1000, const double epsilon = 1e-6)
{
    int i = 0;
    const double alfa = (std::sqrt(5) - 1.0) / 2.0;
    double c = b - (alfa * (b - a));
    double d = a + (alfa * (b - a));
    while (b - a >= epsilon)
    {
        if (f(c) < f(d)) {
            b = d;
            d = c;
            c = b - (alfa * (b - a));
        }
        else {
            a = c;
            c = d;
            d = a + (alfa * (b - a));
```

```
        }
        i++;
        if (i > N_max)
        {
            //throw std::runtime_error("Nie udało się ustalić przedziału w N_max
krokach!");
            std::cout << "[Zloty podzial] Nie udalo sie ustalic przedzialu w
N_max krokach!" << std::endl;
            return NULL;
        }
    }
    return (a + b) / 2.0;
}
// Znajdowanie optymalnego minimum funkcji - metodą Powella
std::vector<double> metoda_Powella_v2(const std::function<double(double,
std::vector<double>, double)>& f, std::vector<double> x_0, double w,const double
c, int N_max = 10000, const double epsilon = 1e-6)
{
    auto minimum_kierunkowe = [&](std::vector<double> x, std::vector<double> dx)
{
        auto funkcja_liniowa = [&](double alfa) {
            std::vector<double> temp;
            for (int i = 0; i < x.size(); i++) temp.push_back(x[i] + alfa *
dx[i]);
            return f(w, temp, c);
            };
        auto przedzial = metoda_ekspansji(funkcja_liniowa, 0, 0.1, 1.5, N_max);
        //std::cout << "(" << przedzial.first << ", " << przedzial.second << ")"
<< std::endl;
        double alfa_opt = metoda_zlotego_podzialu(funkcja_liniowa,
przedzial.first, przedzial.second, N_max, epsilon);
        std::vector<double> p_opt = x;
        for (int i = 0; i < x.size(); i++)
        {
            p_opt[i] += alfa_opt * dx[i];
        }
        return p_opt;
        };


    int n = x_0.size();
    std::vector<std::vector<double>> d(n, std::vector<double>(n, 0.0));
    //for (int j = 0; j < n; ++j) d[j][j] = 1.0; // Początkowe kierunki
jednostkowe
    for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) d[i][j] =
generowanie_wartosci_losowej(0, 1);
    std::vector<std::vector<double>> p;
    p.push_back(x_0);
    while (function_call_count <= N_max)
    {
        for (int j = 0; j < n; j++)
        {
            //std::cout << "p: [" << p[p.size()-1][0] << ", " << p[p.size()-1][1]
<< "]; dp: [" << d[j][0] << ", " << d[j][1] << "]" << std::endl;
            std::vector<double> temp = minimum_kierunkowe(p.back(), d[j]);
            p.push_back(temp);
            if (p.size() > n + 1) p.erase(p.begin());
        }

        //if (abs(f(w, p.back(), c) - f(w, p[0], c)) < epsilon) return p.back();
        double norma_euklidesowa = 0.0;
        for (int element = 0; element < p.back().size(); element++)
norma_euklidesowa += pow((p[p.size()-1][element] - p[0][element]), 2);
```

```
        norma_euklidesowa = sqrt(norma_euklidesowa);
        if (norma_euklidesowa < epsilon) return p[0];


        std::vector<double> help = p.back();
        for (int element = 0; element < p.back().size(); element++) help[element]
-= p[0][element];

        // normalizacja wektora
        double norma = 0.0;
        for (int element = 0; element < p.back().size(); element++) norma +=
pow(help[element], 2);
        norma = sqrt(norma);
        for (int element = 0; element < p.back().size(); element++) help[element]
/= norma;

        d.push_back(help);
        d.erase(d.begin());
        //std::cout << "p: [" << p[p.size() - 1][0] << ", " << p[p.size() - 1][1]
<< "]; dp: [" << d[d.size() - 1][0] << ", " << d[d.size() - 1][1] << "]" <<
std::endl;
        std::vector<double> temp = minimum_kierunkowe(p.back(), d.back());
        p.push_back(temp);
        p.erase(p.begin());
    }
    //throw std::runtime_error("Nie udało się ustalić przedziału w N_max
krokach!");
    std::cout << "[Powell] więcej wywołan funkcji niz N_max!" << std::endl;
}
// ================================================================================

// ====================   [ Funkcje problemu testowego ]   ====================

// Funkcja pierwsza, wg dostarczonego wzoru
double f1(double x1, double x2, double a)
{
      double result = a * (pow((x1 - 2), 2) + pow((x2 - 2), 2));
      return result;
}

// Funkcja druga, wg dostarczonego wzoru
double f2(double x1, double x2, double a)
{
      double result = (1.0/a) * (pow((x1 + 2), 2) + pow((x2 + 2), 2));
      return result;
}

// Funkcja ważona problemu testowego
double problem_jednokryterialny2(double w, std::vector<double> x, double a)
{
    function_call_count++;

    double result = (w * f1(x[0], x[1], a)) + ((1 - w) * f2(x[0], x[1], a));
    return result;
}

// Wywołanie optymalizacji dla problemu rzeczywistego
void problem_testowy()
{
    double e = 1e-5;
    int N_max = 100000;
    std::vector<double> a = { 1, 10, 100 };
```

```
    std::ofstream results1("optimization_results_1.csv");
    results1 << "a;w;x1_0;x2_0;x1;x2;f1;f2;f_calls\n";
    std::ofstream results2("optimization_results_2.csv");
    results2 << "a;w;x1_0;x2_0;x1;x2;f1;f2;f_calls\n";
    std::ofstream results3("optimization_results_3.csv");
    results3 << "a;w;x1_0;x2_0;x1;x2;f1;f2;f_calls\n";

    for (int i = 0; i < 101; i++)
    {
        std::vector<double> x_0 = generowanie_wektora_wartosci_losowych(-10, 10,
2);
        double w = i * 0.01;
        for (double a : a)
        {
            function_call_count = 0;
            auto wynik = metoda_Powella_v2(problem_jednokryterialny2, x_0, w, a,
N_max, e);
            std::vector<double> x_opt = wynik;
            int f_calls = function_call_count;
            double f1_val = f1(x_opt[0], x_opt[1], a);
            double f2_val = f2(x_opt[0], x_opt[1], a);

            if (a == 1) results1 << a << ";" << w << ";" << x_0[0] << ";" <<
x_0[1] << ";" << x_opt[0] << ";" << x_opt[1] << ";" << f1_val << ";" << f2_val <<
";" << f_calls << "\n";
            else if (a == 10) results2 << a << ";" << w << ";" << x_0[0] << ";"
<< x_0[1] << ";" << x_opt[0] << ";" << x_opt[1] << ";" << f1_val << ";" << f2_val
<< ";" << f_calls << "\n";
            else if (a == 100) results3 << a << ";" << w << ";" << x_0[0] << ";"
<< x_0[1] << ";" << x_opt[0] << ";" << x_opt[1] << ";" << f1_val << ";" << f2_val
<< ";" << f_calls << "\n";
        }
    }
    results1.close();
    results2.close();
    results3.close();
    std::cout << "Optimization complete. Results saved\n";
}

// ============================================================================

// ===================   [ Funkcje problemu rzeczywistego ]   ===================

// Funkcja obliczająca masę
double f_m(double l, double d)
{
    double m = rho * PI * pow((d / 2.0), 2) * l;
    return m;
}

// Funkcja obliczająca ugięcie
double f_u(double l, double d)
{
    double u = (64.0 * P * pow(l, 3)) / (3 * E * PI * pow(d, 4));
    return u;
}

// Funkcja obliczająca naprężenie
double sigma(double l, double d)
{
    double sigma = (32.0 * P * l) / (PI * pow(d, 3));
    return sigma;
}
```

```cpp
// Sprawdzanie czy są spełnione zadane ograniczenia
bool sprawdzanie_ograniczen(double l, double d)
{
    double u = f_u(l, d);
    double sig = sigma(l, d);
    return (u <= u_max && sig <= sigma_max && l <= l_max_g && d <= d_max_g && l
>= l_min_g && d >= d_min_g);
}

// Funkcja obliczająca karę zewnętrzną
double funkcja_kary(double l, double d, double c)
{
    double penalty = 0.0;
    //if (!sprawdzanie_ograniczen(l, d)) {
        //penalty = c * (pow(std::max(0.0, f_u(l, d) - u_max), 2) +
pow(std::max(0.0, sigma(l, d) - sigma_max), 2) + pow(std::max(0.0, l - l_max_g),
2) + pow(std::max(0.0, d - d_max_g), 2) + pow(std::max(0.0, -(l - l_min_g)), 2) +
pow(std::max(0.0, -(d - d_min_g)), 2));
        double kara_u = pow(std::max(0.0, f_u(l, d) - u_max), 2);
        double kara_sigma = pow(std::max(0.0, sigma(l, d) - sigma_max), 2);
        double kara_l_min = pow(std::max(0.0, -(l - l_min_g)), 2);
        double kara_l_max = pow(std::max(0.0, l - l_max_g), 2);
        double kara_d_min = pow(std::max(0.0, -(d - d_min_g)), 2);
        double kara_d_max = pow(std::max(0.0, d - d_max_g), 2);
        penalty = c * (kara_sigma + kara_u + kara_l_min + kara_l_max + kara_d_min
+ kara_d_max);
    //}
    return penalty;
}

// Funkcja ważona problemu rzeczywistego
double funkcja_wazona(double w, std::vector<double> x, double c)
{
    function_call_count++;
    double l = x[0], d = x[1];
    double penalty = funkcja_kary(l, d, c);
    double wynik = (w * f_m(l, d)) + ((1 - w) * f_u(l, d)) + penalty;
    //std::cout << "Dla l = " << l << "; d = " << d <<"; w = "<< w << "; c = " <<
c << ": penalty = " << penalty << "; f = " << wynik <<"; f_m = "<<f_m(l,d)<<";
f_u = "<<f_u(l,d)<<"; f_sigma = "<<sigma(l,d) << std::endl;
    return wynik;
}

// Wywołanie optymalizacji dla problemu rzeczywistego
void problem_rzeczywisty()
{
    const double l_min = l_min_g; // [m]
    const double l_max = l_max_g; // [m]
    const double d_min = d_min_g; // [m]
    const double d_max = d_max_g; // [m]

    const double epsilon = 1e-9;
    const int N_max = 10000000;

    std::vector<std::pair<double, double>> starting_values;
    std::vector<std::pair<double, double>> results;
    std::vector<double> f_calls;

    for (double w = 0.0; w <= 1.01; w += 0.01) {
        function_call_count = 0;
        double l = generowanie_wartosci_losowej(l_min, l_max);
        double d = generowanie_wartosci_losowej(d_min, d_max);
```

```
        starting_values.emplace_back(l, d);
        std::cout << "Optymalizacja dla w = " << w << ": l = "<<l<<", d = "<<d <<
std::endl;
        double l_opt;
        double d_opt;
        int i = 10;
        do {
            double c = pow(10, i);
            auto wynik = metoda_Powella_v2(funkcja_wazona, { l, d }, w, c, N_max,
epsilon);
            l_opt = wynik[0];
            d_opt = wynik[1];
            //i++;
        } while (!sprawdzanie_ograniczen(l_opt, d_opt) /*&& i <= 100*/);
        results.emplace_back(l_opt, d_opt);
        f_calls.emplace_back(function_call_count);
        std::cout << "Zakonczono optymalizacje dla w = " << w << "; [WYNIK] = ["
<< l_opt << ", " << d_opt << "]" << std::endl;
    }

    std::ofstream file("results.csv");
    file << "w;l_0;d_0;l_opt;d_opt;mass;deflection;function_calls" << std::endl;
    for (size_t i = 0; i < results.size(); ++i) {
        double l_opt = results[i].first;
        double d_opt = results[i].second;
        file << i * 0.01 << ";" << starting_values[i].first << ";" <<
starting_values[i].second << ";" << l_opt << ";" << d_opt << ";" << f_m(l_opt,
d_opt) << ";" << f_u(l_opt, d_opt) << ";" << f_calls[i] << std::endl;
    }
    file.close();

    std::cout << "Optymalizacja zakończona. Wyniki zapisano w pliku results.csv"
<< std::endl;
}

// ================================================================================

// ====================   [ Dodatkowe metody pomocnicze ]   ====================

// Generowanie wektora o długości ilosc ze zmiennoprzecinkowymi wartościami
losowymi z przedziału [a, b]
std::vector<double> generowanie_wektora_wartosci_losowych(double a, double b, int
ilosc)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dist(a, b);
    std::vector<double> output;
    for (int i = 0; i < ilosc; i++) output.push_back(dist(gen));
    return output;
}

// Generowanie zmiennoprzecinkowej wartości losowej z przedziału [a, b]
double generowanie_wartosci_losowej(double a, double b)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dist(a, b);
    double output = dist(gen);
    return output;
}
// Dodatkowa funkcja do testowania
```

```cpp
double test(double w, std::vector<double> x, double c)
{
    return pow((x[0] - 150), 2) + pow((x[1] + 150), 2);
}

//================================================================================

int main()
{
    //problem_testowy();
    problem_rzeczywisty();
      return 0;
}
```