AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

**WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ**

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

# Sprawozdanie 5

## *Optymalizacja wielokryterialna*

Autorzy: *Paulina Grabowska*
*Filip Rak*
*Arkadiusz Sala*

Kierunek studiów: *Inżynieria Obliczeniowa*

Kraków, 2025

# Spis treści

# Cel ćwiczenia

Celem przeprowadzonego ćwiczenia było pogłębienie wiedzy na temat optymalizacji wielokryterialnego oraz metody Powella. Zostały one zaimplementowane i zastosowane do rozwiązywania problemów optymalizacyjnych dla funkcji wielu zmiennych, umożliwiając praktyczne zrozumienie ich działania oraz efektywności.

# Optymalizowane problemy

## Testowa funkcja celu

Celem praktycznego wykorzystania zaimplementowanych algorytmów badano funkcje celu podane wzorami (1). Obrany punkt startowy będzie należał do przedziału [-10, 10] (dla obu zmiennych).

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

$$f_2(x_1, x_2) = \frac{1}{a}((x_1 + 2)^2 + (x_2 + 2)^2) \tag{1}$$

W pliku *user_funs.cpp* dodano implementację funkcji (1), jak przedstawiono we *Fragmencie kodu 1*. Ponieważ optymalizacja będzie przeprowadzana dla trzech parametrów $a$ – o wartościach 1, 10 oraz 100, to każdą funkcję oraz zależność wag zaimplementowano w sposób analogiczny.

```
matrix ff5T1_1(matrix x, matrix ud1, matrix ud2)
{
	if (isnan(ud2(0)))
		return (pow( x(0) - 2.0 )+pow( x(1) - 2.0 ));
	else
		return (pow(ud1(0) + x(0) * ud2(0) - 2.0) + pow(ud1(1) + x(0) * ud2(1) - 2.0));
}
matrix ff5T2_1(matrix x, matrix ud1, matrix ud2)
{
	if (isnan(ud2(0)))
		return (pow(x(0) + 2.0) + pow(x(1) + 2.0));
	else
		return (pow(ud1(0) + x(0) * ud2(0) + 2.0) + pow(ud1(1) + x(0) * ud2(1) + 2.0));
}
matrix ff5T3_1(matrix x, matrix ud1, matrix ud2)
{
	return weights[weight_i] * ff5T1_1(x, ud1, ud2) + (1.0 - weights[weight_i]) * ff5T2_1(x, ud1, ud2);
}
```

```
19. matrix ff5T1_10(matrix x, matrix ud1, matrix ud2)
20. {
21.       if (isnan(ud2(0)))
22.                 return 10.0*(pow(x(0) - 2.0) + pow(x(1) - 2.0));
23.       else
24.                 return 10.0*(pow(ud1(0) + x(0) * ud2(0) - 2.0) + pow(ud1(1) + x(0) * ud2(1) -
2.0));
25. }
26. matrix ff5T2_10(matrix x, matrix ud1, matrix ud2)
27. {
28.       if (isnan(ud2(0)))
29.                 return 0.1*(pow(x(0) + 2.0) + pow(x(1) + 2.0));
30.       else
31.                 return 0.1*(pow(ud1(0) + x(0) * ud2(0) + 2.0) + pow(ud1(1) + x(0) * ud2(1) +
2.0));
32. }
33. matrix ff5T3_10(matrix x, matrix ud1, matrix ud2)
34. {
35.       return weights[weight_i] * ff5T1_10(x, ud1, ud2) + (1.0 - weights[weight_i]) *
ff5T2_10(x, ud1, ud2);
36. }
37. matrix ff5T1_100(matrix x, matrix ud1, matrix ud2)
38. {
39.       if (isnan(ud2(0)))
40.                 return 100.0 * (pow(x(0) - 2.0) + pow(x(1) - 2.0));
41.       else
42.                 return 100.0 * (pow(ud1(0) + x(0) * ud2(0) - 2.0) + pow(ud1(1) + x(0) *
ud2(1) - 2.0));
43. }
44. matrix ff5T2_100(matrix x, matrix ud1, matrix ud2)
45. {
46.       if (isnan(ud2(0)))
47.                 return 0.01 * (pow(x(0) + 2.0) + pow(x(1) + 2.0));
48.       else
49.                 return 0.01 * (pow(ud1(0) + x(0) * ud2(0) + 2.0) + pow(ud1(1) + x(0) * ud2(1)
+ 2.0));
50. }
51. matrix ff5T3_100(matrix x, matrix ud1, matrix ud2)
52. {
53.       return weights[weight_i] * ff5T1_100(x, ud1, ud2) + (1.0 - weights[weight_i]) *
ff5T2_100(x, ud1, ud2);
54. }
55.
```

**Fragment kodu 1:** funkcje liczące testowe funkcję celu dla różnych parametrów *a*.

## Problem rzeczywisty

Analizowany problem rzeczywisty dotyczy optymalizacji konstrukcji belki o przekroju kołowym, która jest obciążona siłą. Celem jest zaprojektowanie belki o minimalnej masie i minimalnym ugięciu, jednocześnie spełniając określone wymagania dotyczące jej wytrzymałości i odkształceń.

Belka ma przekrój kołowy o średnicy $d$ i długości $l$ (*Rys.1*). Na belkę działa siła $P = 1kN$, która powoduje jej ugięcie i naprężenie. Pod wpływem działania siły belka się ugina zgodnie ze wzorem (2), który wyraża jej ugięcie $u$. Użyty we wzorze parametr $E$ to wartość modułu Younga wynosząca 207 GPa.

$$u = \frac{64 \cdot P \cdot l^3}{3 \cdot E \cdot \pi \cdot d^4} \tag{2}$$

```
// Constants
const double density = 7800.0; // kg/m^3 (rho)
const double force = 1000.0; // 1 kN (P)
const double youngs_modulus = 207e9; // E in Pa (GPa -> Pa)
const double pi = 3.141592653;


matrix ff5R_deflection(matrix x)
{
	double beam_length = x(0, 0);
	double csd = x(1, 0);		// cross-sectional diameter

	// Deflection in meters
	double deflection = (64.0 * force * pow(beam_length, 3)) / (3.0 * youngs_modulus * pi * pow(csd, 4));
	return matrix(deflection);
}

```

**Fragment kodu 2a:** stałe parametry problemu rzeczywistego oraz funkcja licząca wzór (2)

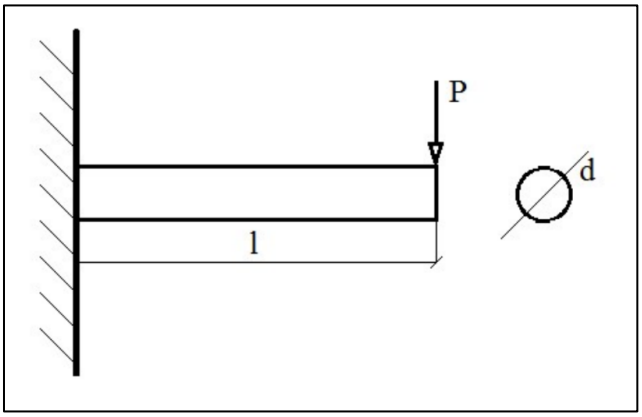
Ponadto, w belce występuje naprężenie $\sigma$ zgodne ze wzorem (3).

$$\sigma = \frac{32 \cdot P \cdot l}{\pi \cdot d^3} \tag{3}$$

```
matrix ff5R_stress(matrix x)
{
	double beam_length = x(0, 0);
	double csd = x(1, 0);		// cross-sectional diameter

	// Calculate stress
	double stress = (32.0 * force * beam_length) / (pi * pow(csd, 3));
	return matrix(stress);
}
```

**Fragment kodu 2b:** funkcja licząca wzór (3)




**Rys.1**. Ilustracja problemu rzeczywistego

Badana belka jest zrobiona z materiału, którego gęstość wynosi $\rho = 7800^{kg}/_{m^3}$. Przeprowadzane zadanie polega na znalezieniu takich wartości l ∈ [200 mm, 1000 mm] oraz d ∈ [10 mm, 50 mm], że masa belki – wzór (4) oraz ugięcie – wzór (2) – osiągają wartości optymalne przy ograniczeniach, że maksymalne ugięcie $u_{max}$ nie przekracza 5mm, a maksymalne naprężenie $\sigma_{max}$ nie przekracza 300 MPa.

$$m = \rho \cdot \frac{\pi \cdot d^2}{4} \cdot l \tag{4}$$

```
matrix ff5R_mass(matrix x)
{
	double beam_length = x(0);
	double csd = x(1);  // cross-sectional diameter

	// Cross-sectional area
	double csa = pi * pow(csd / 2.0, 2); // m^2

	// Calculate mass
	double mass = density * csa * beam_length; // kg
	return matrix(mass);
}

```

**Fragment kodu 2c:** funkcja licząca wzór (4)

Funkcja *ff5R_penalty* (*Fragment kodu 3*) została zaimplementowana, aby obliczać karę dla określonych parametrów belki konstrukcyjnej - długość belki (*beam_length*) i średnica przekroju poprzecznego (*csd*). Funkcja uwzględnia ograniczenia projektowe i dodaje kary, gdy parametry nie spełniają określonych kryteriów.

```
matrix ff5R_penalty(matrix x)
{
	double beam_length = x(0, 0);
	double csd = x(1, 0);		// cross-sectional diameter

	// Deflection in meters
	double deflection = m2d(ff5R_deflection(x));

	// Stress in Pa
	double stress = m2d(ff5R_stress(x));

	// Penalty and constraints
	double penalty = 0.0;
	const double penalty_modifier = 1e15;
	const double penalty_base = 1e5;
	const double max_length = 1;  // Meters
	const double min_length = 0.200;	// Meters
	const double csd_max = 0.050; // Meters
	const double csd_min = 0.010; // Meters
	const double reflection_max = 5.0e-3;	// Meters
	const double stress_max = 300e6;  // MPa -> Pa

	/* Apply penalties as proportion of offset */

	// Beam length
	if (beam_length < min_length)
	{
```

```
28.                 double diff = min_length - beam_length;
29.                 penalty += penalty_modifier * (diff / min_length) + penalty_base;
30.         }
31.
32.         if (beam_length > max_length)
33.         {
34.                 double diff = beam_length - max_length;
35.                 penalty += penalty_modifier * (diff / max_length) + penalty_base;
36.         }
37.
38.         // Cross sectionnal diameter
39.         if (csd < csd_min)
40.         {
41.                 double diff = csd_min - csd;
42.                 penalty += penalty_modifier * (diff / csd_min) + penalty_base;
43.         }
44.
45.         if (csd > csd_max)
46.         {
47.                 double diff = csd - csd_max;
48.                 penalty += penalty_modifier * (diff / csd_max) + penalty_base;
49.         }
50.
51.         // Deflection
52.         if (deflection > reflection_max)
53.                 penalty += penalty_modifier * (deflection - reflection_max) / reflection_max
+ penalty_base;
54.
55.         if (deflection < 0.f)
56.                 penalty += penalty_modifier * (-deflection);
57.
58.         // Stress
59.         if (stress > stress_max)
60.                 penalty += penalty_modifier * (stress - stress_max) / stress_max +
penalty_base;
61.
62.         if (stress < 0.f)
63.                 penalty += penalty_modifier * (-stress) + penalty_base;
64.
65.         // Return the total penalty
66.         return matrix(penalty);
67. }
68.
```

**Fragment kodu 3:** funkcja nakładająca kary według złożeń problemu rzeczywistego

Funkcja *ff5R* (*Fragment kodu 4*) oblicza wartość funkcji celu jako ważoną kombinację masy, ugięcia i kary dla parametrów wejściowych belki. Jeśli *ud2* nie jest *NaN*, wprowadza modyfikacje do parametrów *x* za pomocą *ud1* i *ud2*. Następnie oblicza masę, ugięcie i karę, a wynik zwraca jako macierz zawierającą wartość funkcji celu.

```
 1. matrix ff5R(matrix x, matrix ud1, matrix ud2)
 2. {
 3.         // If ud2 is not NaN apply adjustments from ud1 and ud2
 4.         matrix xi = x;
 5.         if (!isnan(ud2(0)))
 6.                 xi = ud1 + m2d(x) * ud2;
 7.
 8.         // Debug; Set [entry] to true to enable
 9.         static bool entry = false;
10.         if (entry)
11.         {
12.                 std::cout << "----------\n";
13.                 std::cout << "ff5R args:\n";
14.                 std::cout << "x vals:\n" << x << "\n";
```

```
15.                 std::cout << "xi vals:\n" << x << "\n";
16.                 std::cout << "ud1 vals:\n" << ud1 << "\n";
17.                 std::cout << "ud2 vals:\n" << ud2 << "\n";
18.                 std::cout << "----------\n";
19.
20.                 entry = false;
21.         }
22.         // Get the weight
23.         double weight = weights[weight_i];
24.
25.         // Get mass, deflection and penalty
26.         matrix mass = ff5R_mass(xi);
27.         matrix deflection = ff5R_deflection(xi);
28.         matrix penalty = ff5R_penalty(xi);
29.
30.         // Objective function calculation (weighted combination)
31.         double objective = weight * m2d(mass) + (1.0 - weight) * m2d(deflection) +
m2d(penalty);
32.         return matrix(objective);
33. }
34.
```

**Fragment kodu 4:** funkcja *ff5R*

# Implementacja metod optymalizacji

## Metoda Powella

Funkcja Powell implementuje metodę Powella do znajdowania minimum funkcji wielu zmiennych bez obliczania pochodnych. Przyjmuje wskaźnik do funkcji celu, początkowy punkt optymalizacji, tolerancję błędu, maksymalną liczbę iteracji oraz dodatkowe parametry. Działa iteracyjnie, optymalizując wzdłuż ustalonych kierunków, które są aktualizowane na podstawie wyników. Wykorzystuje metodę złotego podziału i ekspansji do znajdowania minimum wzdłuż kierunków. Warunki stopu to osiągnięcie tolerancji lub przekroczenie limitu wywołań funkcji. W przypadku sukcesu zwraca znalezione minimum, a w przypadku problemów odpowiednie flagi błędu.

```
 1. solution Powell(matrix(*ff)(matrix, matrix, matrix), matrix x0, double epsilon, int Nmax,
matrix ud1, matrix ud2)
 2. {
 3.         try
 4.         {
 5.                 solution Xopt;
 6.                 int DIM = get_len(x0);
 7.                 matrix e = ident_mat(DIM);
 8.                 matrix d = matrix(e);
 9.                 matrix xi = x0;
10.                 do
11.                 {
12.                         matrix p = matrix(DIM,DIM+1);
13.                         p.set_col(xi,0);
14.                         for(int j = 1; j <= DIM; j++)
15.                         {
16.                                 //wyznacz h
17.                                 //std::cout << " range: \n";
```

```
18.                                     double* range = expansion(ff, -100.0, 100.0, 2.0, Nmax,
p[j - 1], d[j-1]);
19.                                     double h0 = m2d(golden(ff, range[0], range[1], epsilon,
Nmax, p[j - 1], d[j - 1]).x);
20.                                     p.set_col(p[j - 1] + d[j - 1] * h0,j);
21.                                     delete[] range;
22.                             }
23.                             if (norm(p[DIM] - xi) < epsilon)
24.                             {
25.                                     Xopt.x = xi;
26.                                     Xopt.flag = 0;
27.                                     break;
28.                             }
29.                             for (int j = 0; j < DIM-1; j++)
30.                             {
31.                                     d.set_col(d[j+1],j);
32.                             }
33.                             d.set_col(p[DIM] - p[0], DIM - 1);
34.
35.                             double* range = expansion(ff, 0, 10.0, 2.0, Nmax, p[DIM], d[DIM -
1]);
36.                             double h0 = m2d(golden(ff, range[0], range[1], epsilon, Nmax,
p[DIM], d[DIM - 1]).x);
37.                             xi = p[DIM] + h0 * d[DIM - 1];
38.                             delete[] range;
39.                             if (solution::f_calls > Nmax)
40.                             {
41.                                     Xopt.x = xi;
42.                                     Xopt.flag = -2;
43.                                     break;
44.                             }
45.                     } while(true);
46.                     return Xopt;
47.             }
48.             catch (string ex_info)
49.             {
50.                     throw ("solution Powell(...):\n" + ex_info);
51.             }
52. }
53.
```

**Fragment kodu 5:** metoda Powella

## Metoda kryterium ważonego

Kryterium ważone to metoda łączenia dwóch lub więcej funkcji celu w jedną funkcję agregowaną, co umożliwia przeprowadzenie optymalizacji w przypadku problemów wielokryterialnych. Polega ono na obliczaniu kombinacji liniowej funkcji celu za pomocą odpowiednio dobranych wag, zgodnie we wzorem (5).

$$f(x) = w \cdot f_1(x) + (1 - w) \cdot f_2(x) \qquad \text{dla } w \in [0,1] \qquad (5)$$

Jeżeli wartości funkcji różnią się znacząco, to funkcja o większych wartościach zdominuje wynik optymalizacji. Aby temu zapobiec, należy przeskalować obie funkcje tak, aby przyjmowały wartości w podobnym zakresie, np. przez standaryzację.

We fragmencie kodu (6) parametry *static double weights[101]* oraz *static int weight_i* są używane do sterowania liniowym połączeniem funkcji celu *ff5T1* i *ff5T2*. Tablica *weights[101]* zawiera wartości wagowe z przedziału od 0.0 do 1.0 w krokach co 0.01. Wagi te są używane do skalowania wkładu dwóch funkcji celu w funkcji złożonej *ff5T3*, a *weight_i* to indeks aktualnie wybranej wagi w tablicy *weights*. Określa, jakie wagi są używane w danym momencie przy tworzeniu kombinacji liniowej w funkcjach *ff5T3*. Funkcja *set_weight(int i)* służy do ustawienia indeksu wagi na podaną wartość, a funkcja *init_weights()* inicjalizuje tablicę *weights* wartościami od 0.0 do 1.0.

```
1. static double weights[101];
2. static int weight_i = -1;
3.
4. void set_weight(int i)
5. {
6.       weight_i = i;
7. }
8.
9. void init_weights()
10. {
11.       for (int i = 0; i < 101; i++)
12.       {
13.                 weights[i] = static_cast<double>(i) * 0.01;
14.       }
15. }
16.
```

**Fragment kodu 6:** definicja wag

Minimalizacja na kierunku zostanie przeprowadzona metodą złotego podziału, a początkowy przedział zostanie wyznaczony metodą ekspansji. Podczas analizy problemu rzeczywistego uwzględniona zostanie zewnętrzna funkcja kary.

## Opracowanie wyników

Zadeklarowano zmienne lokalne wykorzystywane podczas późniejszego testowania i dokładność oraz maksymalną liczbę iteracji (*Fragment kodu 7*).

```
1.        // Common arguments
2.        double epsilon = 1e-3;
3.        int Nmax = 1000;
4.
```

**Fragment kodu 7:** zmienne lokalne niezbędne dla działania badanych funkcji

## Testowa funkcja celu

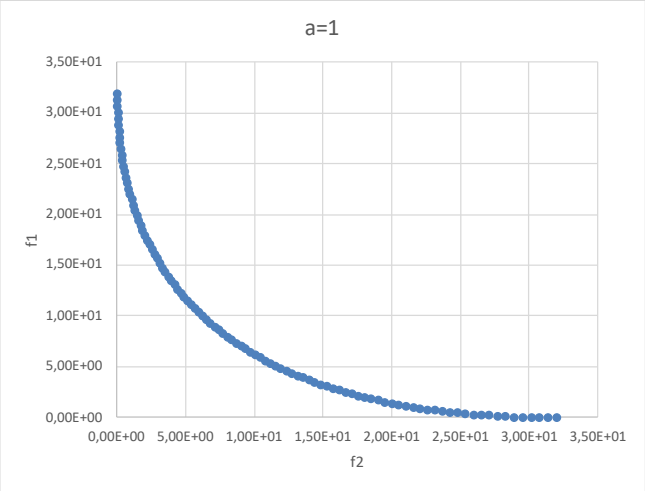
Przeprowadzone zostanie 101 optymalizacji dla różnych wartości parametru w (od 0 do 1) przy różnych wartościach parametru a, zaczynając z losowego punktu początkowego. Wyniki zostaną zapisane w tabeli, a dla każdej wartości a (1, 10 oraz 100) zostanie stworzony wykres przedstawiający minimalne rozwiązania Pareto.

Kod generuje dane dla tabeli w pliku tekstowym, przeprowadzając optymalizację funkcji przy użyciu metody Powell'a. Dla każdej z 101 iteracji losowo generowane są wartości zmiennych w zadanym zakresie. W każdej iteracji obliczane są wyniki optymalizacji dla trzech funkcji celów o różnych wagach, a następnie zapisywane do pliku *out_1_tfun.txt*. Wyniki obejmują współrzędne optymalne, wartości funkcji celów oraz liczbę wywołań funkcji w każdej optymalizacji.

```
1.      std::cout << "Solving for Table 1...\n";
2.
3.      double max_values[2]{ 10.0, 10.0 };
4.      double min_values[2]{ -10.0, -10.0 };
5.
6.      {
7.              matrix test = matrix(2, new double[2] {
8.                      min_values[0] + static_cast<double>(rand()) / RAND_MAX *
(max_values[0] - min_values[0]),
9.                                      min_values[1] + static_cast<double>(rand()) / RAND_MAX *
(max_values[1] - min_values[1])
10.                     });
11.
12.             ofstream tfun_file1(OUTPUT_PATH + "out_1_tfun.txt");
13.             if (!tfun_file1.good()) return;
14.
15.             for (int i = 0; i < 101; i++)
16.             {
17.                     set_weight(i);
18.                     tfun_file1 << test(0) << delimiter << test(1) << delimiter;
19.                     solution is1 = Powell(ff5T3_1, test, epsilon, Nmax, NAN, NAN);
20.                     tfun_file1 << is1.x(0) << delimiter << is1.x(1) << delimiter <<
ff5T1_1(is1.x, NAN, NAN) << delimiter << ff5T2_1(is1.x, NAN, NAN) << delimiter <<
solution::f_calls << delimiter;
21.                     solution::clear_calls();
22.                     solution is10 = Powell(ff5T3_10, test, epsilon, Nmax, NAN, NAN);
23.                     tfun_file1 << is10.x(0) << delimiter << is10.x(1) << delimiter <<
ff5T1_10(is10.x, NAN, NAN) << delimiter << ff5T2_10(is10.x, NAN, NAN) << delimiter <<
solution::f_calls << delimiter;
24.                     solution::clear_calls();
25.                     solution is100 = Powell(ff5T3_100, test, epsilon, Nmax, NAN, NAN);
26.                     tfun_file1 << is100.x(0) << delimiter << is100.x(1) << delimiter <<
ff5T1_100(is100.x, NAN, NAN) << delimiter << ff5T2_100(is100.x, NAN, NAN) << delimiter <<
solution::f_calls << delimiter;
27.                     solution::clear_calls();
28.                     tfun_file1 << std::endl;
29.             }
30.
31.             tfun_file1.close();
32.     }
33.
```

**Fragment kodu 11**: fragment funkcji *main* (*lab5*) zawierający funkcjonalności dla *Tabeli 1*

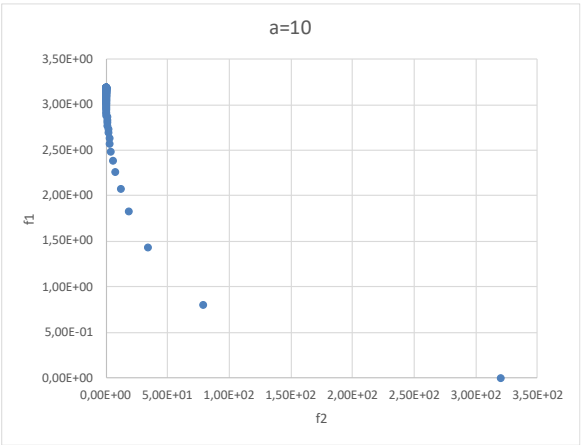
W przypadku wartości *a* równej 1 rozwiązania Pareto (*Rys. 2a*) układają się na gładkiej krzywej hiperbolicznej, co oznacza równowagę między funkcjami celu *f1* i *f2*. Zmiana wartości jednej funkcji wiąże się ze znaczną zmianą drugiej. Krzywa pokazuje dobrze rozłożone kompromisy między *f1* a *f2*.



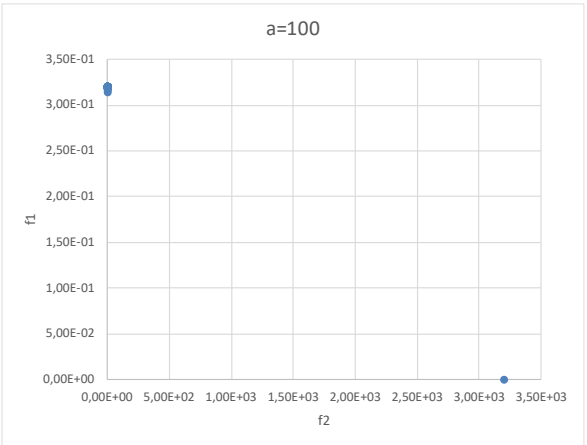

**Rys. 2a:** wykres rozwiązań minimalnych w sensie Pareto dla a = 1

Kiedy zwiększona zostaje wartość *a*, do 10 (*Rys. 2b*), to można wyraźnie zauważyć jak punkty Pareto zaczynają przesuwać się w stronę dominacji *f1*, ponieważ wyższe wartości a wzmacniają wpływ tej funkcji. Rozkład punktów staje się mniej równomierny, a *f2* przyjmuje znacznie większe wartości w porównaniu do *f1*. Dla wartości *a* równej 100 (*Rys. 2c*) dominacja *f1* jest jeszcze bardziej widoczna. Rozwiązania Pareto są skrajnie przesunięte – wartości *f2* są bardzo wysokie, a *f1* jest zminimalizowane. Funkcja *f1* niemal całkowicie determinuje proces optymalizacji.

Im większa wartość *a*, tym większe znaczenie przypisuje się funkcji *f1*, co prowadzi do przesunięcia rozwiązań Pareto w kierunku minimalizacji *f1*, co wynika bezpośrednio ze wzorów (1). Przy *a* = 1 punkty Pareto są równomiernie rozłożone, co wskazuje na balans między funkcjami. Wartości *a* = 10 i *a* = 100 przesuwają rozwiązania w stronę dominacji jednej funkcji kosztem drugiej.




**Rys. 2b:** wykres rozwiązań minimalnych w sensie Pareto dla a = 10




**Rys. 2c:** wykres rozwiązań minimalnych w sensie Pareto dla a = 100

## Problem rzeczywisty

Poniższy fragment kodu realizuje optymalizację problemu rzeczywistego, gdzie długość belki i średnica przekroju poprzecznego są losowo generowane w zadanych zakresach, a następnie optymalizowane metodą Powella dla różnych wag funkcji celu. Kod iteruje 101 razy (dla różnych wag), wykonuje optymalizację dla każdego zestawu parametrów startowych, a wyniki (w tym zoptymalizowane parametry, masę, ugięcie i liczbę wywołań funkcji celu) zapisuje do pliku tekstowego *out2.txt*.

```
1.      /* Real Problem */
2.      std::cout << "Solving for Table 2...";
3.
4.      // Output file
5.      ofstream rp_file(OUTPUT_PATH + "out2.txt");
6.
7.      // Optimizer settings
8.      double real_epsilon = 1e-8;
9.      double real_nmax = 20000;
10.
11.     // Constraints in meters
12.     // Beam length (l)
13.     const double beam_length_min = 0.200f;
14.     const double beam_length_max = 1.000f;
15.
16.     // Cross sectional diameter (d)
17.     const double csd_min = 0.010;
18.     const double csd_max = 0.050;
19.
20.     // Loop over all weights
21.     const int weight_number = 101;
22.     for (int i = 0; i < weight_number; i++)
23.     {
24.             // Randomize beam's properties
25.             double beam_length = beam_length_min + static_cast<double>(rand()) / RAND_MAX
* (beam_length_max - beam_length_min);
26.             double csd = csd_min + static_cast<double>(rand()) / RAND_MAX * (csd_max -
csd_min);
27.
28.             matrix x0 = matrix(2, new double[2] {beam_length, csd});
29.
30.             // Set the weight
31.             set_weight(i);
32.
33.             // Call the optimizer
34.             solution opt_res = Powell(ff5R, x0, real_epsilon, real_nmax, NULL, NULL);
35.
36.             // Output data to the file
37.             rp_file
38.                     << get_weight() << delimiter                    // Weight used for
calculation
39.                     << beam_length * 1000.f << delimiter    // Random beam's length (l)
in mm
40.                     << csd * 1000.f << delimiter;                   // Random cross-
sectional diameter (d) in mm
41.
42.             rp_file
43.                     << opt_res.x(0) * 1000.f << delimiter   // Optimized beam's length
(l*) in mm
44.                     << opt_res.x(1) * 1000.f << delimiter   // Optimized cross-sectional
diameter (d*) in mm
45.                     << ff5R_mass(opt_res.x)(0) << delimiter// Mass for optimized X in
kg
```

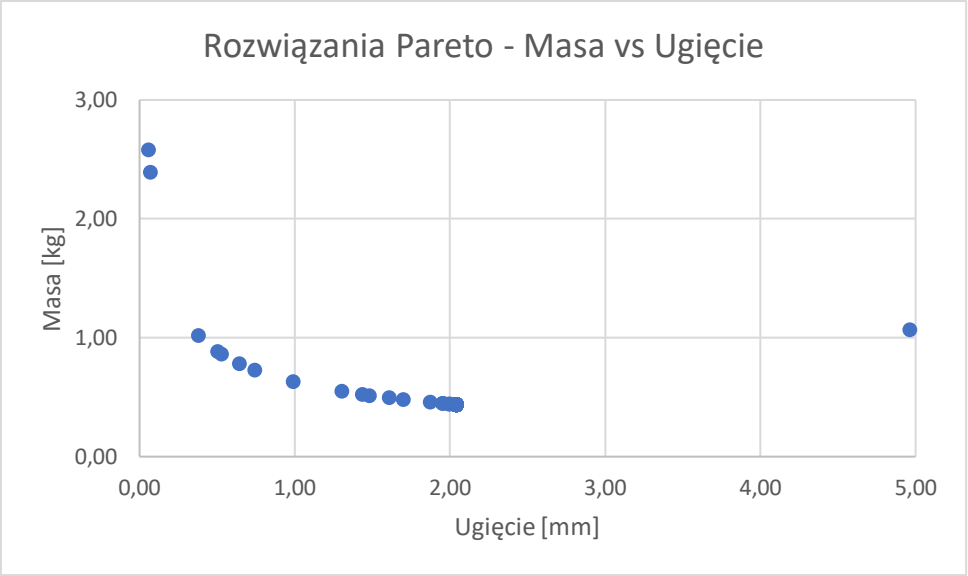
```
46.                             << ff5R_deflection(opt_res.x)(0) * 1000.f << delimiter     //
Deflection for optimized X in mm
47.                             << opt_res.f_calls << delimiter                            // Fit
function calls to reach the solution
48.                             << opt_res.flag;
          // Solution's flag
49.
50.                 // Add newline character between data
51.                 rp_file << "\n";
52.
53.                 // Clear f_calls after saving to output
54.                 solution::clear_calls();
55.       }
56.
57.       // Close the output file
58.       rp_file.close();
59.
```

**Fragment kodu 12:** fragment funkcji *main* zawierający funkcjonalności dla *Tabeli 3*

Poniższy wykres (*Rys. 3*) przedstawia rozwiązania Pareto dla problemu optymalizacji konstrukcji belki o przekroju kołowym. Na osi poziomej znajduje się ugięcie belki w milimetrach, a na osi pionowej masa belki w kilogramach. Wykres pokazuje front rozwiązań w sensie Pareto, który ilustruje kompromis między minimalizacją masy a minimalizacją ugięcia belki. Punkty na tym froncie reprezentują optymalne rozwiązania, gdzie nie można poprawić jednego kryterium bez pogorszenia drugiego.

Ugięcie belki waha się od około 0 mm do 5 mm, a masa belki waha się od około 0.5 kg do 2.5 kg. Punkty po lewej stronie wykresu (niskie ugięcie) mają wyższą masę, co oznacza, że aby zminimalizować ugięcie, konieczne jest zwiększenie masy belki. Punkty po prawej stronie wykresu (wysokie ugięcie) mają niższą masę, co oznacza, że aby zminimalizować masę, konieczne jest zaakceptowanie większego ugięcia.

Punkt znajdujący się ze skrajnie prawej strony reprezentuje ekstremalne rozwiązanie, w którym belka ma minimalną masę, ale największe dopuszczalne ugięcie. Rozwiązanie to wynika wprost z użycia funkcji kary. Większość punktów znajduje się w lewej połowie wykresu, ponieważ optymalne rozwiązania zazwyczaj wymagają kompromisu między masą a ugięciem, a skrajne wartości są rzadziej osiągane ze względu na ograniczenia projektowe i wymagania wytrzymałościowe.




**Rys. 3.** Wykres rozwiązań minimalnych w sensie Pareto dla problemu rzeczywistego

## Wnioski

Wybór *a* powinien być uzależniony od priorytetów w problemie optymalizacyjnym. Niskie wartości *a* wspierają równowagę między funkcjami celu, natomiast wyższe wartości *a* pozwalają na skupienie się na minimalizacji jednej z funkcji, zazwyczaj kosztem drugiej. W praktyce, wybór wartości *a* powinien być zależny od tego, która funkcja celu jest bardziej istotna w danym problemie, ponieważ może to wpłynąć na dystrybucję punktów Pareto i na sposób, w jaki kompromisy są realizowane.

Również w przypadku problemów rzeczywistych, wybór optymalnego rozwiązania zależy od priorytetów projektowych. Jeśli priorytetem jest minimalizacja ugięcia, należy wybrać punkt z lewej strony frontu Pareto, co wiąże się z większą masą belki. Jeśli priorytetem jest minimalizacja masy, należy wybrać punkt z prawej strony frontu Pareto, co wiąże się z większym ugięciem. Optymalne rozwiązania w przypadku szukania minimalnych wartości, są najczęściej zlokalizowane w lewej części wykresu, co wskazuje, że kompromisy bliższe zrównoważeniu są preferowane w rzeczywistych zastosowaniach, z uwagi na ograniczenia konstrukcyjne i wytrzymałościowe.

Należy pamiętać o ograniczeniach projektowych, takich jak maksymalne dopuszczalne ugięcie i maksymalne dopuszczalne naprężenie. Wszystkie punkty na wykresie powinny spełniać te

ograniczenia. W praktyce, wybór konkretnego rozwiązania może zależeć od dodatkowych czynników, takich jak koszty materiałów, łatwość produkcji, czy specyficzne wymagania problemu.

Metoda Powella jest jedną z technik optymalizacji bez ograniczeń, która nie wymaga obliczania gradientów, co czyni ją szczególnie użyteczną w przypadku złożonych funkcji celu, które mogą być nieciągłe lub trudne do zróżniczkowania. Dzięki iteracyjnej naturze metody możliwe jest skuteczne poszukiwanie minimum w wielowymiarowej przestrzeni, co jest kluczowe w przypadku optymalizacji wielokryterialnej, jak w analizowanym problemie rzeczywistym.

Wykresy Pareto, takie jak te przedstawione w analizie, dostarczają wizualnej interpretacji kompromisów między funkcjami celu. Pozwalają także na łączenie różnych właściwości szukanych rozwiązań – czy to obu wartości minimalnych, obu wartości maksymalnych czy jednej wartości minimalnej, a drugiej maksymalnej. W szczególności dla problemów inżynierskich, jak optymalizacja konstrukcji belki, wykresy te pozwalają zrozumieć, jakie zmiany w parametrach projektowych są konieczne do osiągnięcia pożądanych właściwości konstrukcyjnych.