AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Sprawozdanie 3

Optymalizacja z ograniczeniami funkcji wielu zmiennych metodami bezgradientowymi

Autorzy: *Paulina Grabowska*
Filip Rak
Arkadiusz Sala

Kierunek studiów: *Inżynieria Obliczeniowa*

Kraków, 2024

Spis treści

Cel ćwiczenia

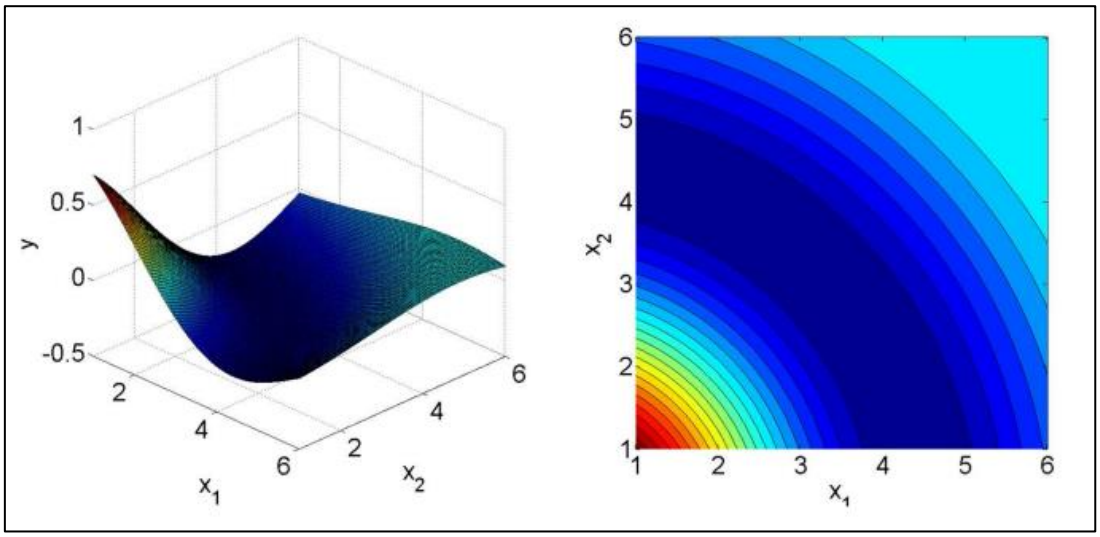
Celem przeprowadzonego ćwiczenia było pogłębienie wiedzy na temat bezgradientowych metod optymalizacji z ograniczeniami – metoda sympleksu Neldera-Meada oraz metody funkcji kary. Zostały one zaimplementowane i zastosowane do rozwiązywania problemów optymalizacyjnych dla funkcji wielu zmiennych, umożliwiając praktyczne zrozumienie ich działania oraz efektywności.

Optymalizowane problemy

Testowa funkcja celu

Celem praktycznego wykorzystania zaimplementowanych algorytmów badano funkcję celu podaną wzorem (1), której wykres został załączony na *Rys.1*. Funkcja ta ma nieskończenie wiele minimów globalnych na łuku znajdującym się około 4.5 jednostki od punktu (0, 0, 0), ponieważ jest funkcją radialną. Podczas optymalizacji uzyskane minimum będzie zależało od obranego punktu startowego.

$$f(x_1, x_2) = \frac{sin\left(\pi\sqrt{\left(\frac{x_1}{\pi}\right)^2 + \left(\frac{x_2}{\pi}\right)^2}\right)}{\pi\sqrt{\left(\frac{x_1}{\pi}\right)^2 + \left(\frac{x_2}{\pi}\right)^2}} \tag{1}$$



Rys.1. Wykres funkcji (1)

W pliku *user_funs.cpp* dodano implementację funkcji (1), jak przedstawiono we *Fragmencie kodu 1*.

```
matrix ff3T(matrix x, matrix ud1, matrix ud2) {
        double tk = M_PI * std::sqrt(m2d(pow(x(0) * M_1_PI) + pow(x(1) * M_1_PI)));
        return sin(tk)/tk;
}
```

Fragment kodu 1: funkcja licząca testową funkcję celu

Funkcji nadano ograniczenia zaprezentowane równaniami (2). Pierwsze dwa równania ograniczają zmienne x_1 oraz x_2 do przedziału [0, 1], natomiast trzecie ograniczenie, jest normą wektora związanym z parametrem a. Odległość między punktami x_1 oraz x_2 nie może przekroczyć wartości parametru a. W przypadku obrania za parametr a wartości 4, minima będą znajdować się poza obszarem dopuszczalnym, a dla wartości 4.4934 będzie zaraz na granicy.

$$\begin{cases} g_1(x_1) = -x_1 + 1 \leq 0 \\ g_2(x_2) = -x_2 + 1 \leq 0 \\ g_3(x_1, x_2) = \sqrt{x_1^2 + x_2^2} - a \leq 0 \end{cases} \tag{2}$$

W pliku *user_funs.cpp* dodano implementację funkcji (1) oraz ograniczenia (2), jak przedstawiono we *Fragmencie kodu 1*.

```
// the boundary g functions
matrix g1(matrix x, matrix ud1)
{
        return -x(0) + 1;
}
matrix g2(matrix x, matrix ud1)
{
        return -x(1) + 1;
}
matrix g3(matrix x, matrix a)
{
        return sqrt(pow(x(0), 2) + pow(x(1), 2)) - m2d(a);
}

matrix ff3T(matrix x, matrix ud1, matrix ud2) {
        double tk = M_PI * std::sqrt(m2d(pow(x(0) * M_1_PI) + pow(x(1) * M_1_PI)));
        return sin(tk)/tk;
}
```

Fragment kodu 1: funkcja implementująca testową funkcję celu oraz jej ograniczenia

Problem rzeczywisty

Problem dotyczy ruchu piłki, która spada z wysokości 100 metrów, posiadając jednocześnie poziomą prędkość początkową oraz rotację. Ruch piłki jest opisywany przez układ równań, które uwzględniają nie tylko grawitację, ale także siłę oporu powietrza oraz efekt Magnusa,

wynikający z rotacji piłki. Celem przeprowadzonej optymalizacji będzie znalezienie takich wartości v_{0x} (początkowa prędkość piłki w kierunku poziomym) zawierająca się w przedziale [-10, 10] $^{m}/_{s}$ oraz wartości ω (początkowej rotacji piłki) zawierającej się w przedziale [-15, 15] $^{rad}/_{s}$, które zapewnią największą wartość x_{end}.

Parametry piłki obrane w symulacji:

- masa piłki m = 600g = 0.6kg;
- promień piłki r = 12cm = 0.12 m;
- początkowa wysokość y_0 = 100m;
- początkowa prędkość w kierunku pionowym $v_{0y} = 0\ ^{m}/_{s}$;

Na spadającą piłkę działają trzy siły: siła grawitacji, siła oporu powietrza oraz siła Magnusa. Siła oporu powietrza (3) działa przeciwnie w kierunkach pionowym i poziomym. We wzorach (3) C jest współczynnikiem oporu powietrza dla piłki (uzależniony od jej kształtu i równy 0.47); ρ jest gęstością powietrza (1,2 $^{kg}/_{m^3}$); S jest powierzchnią przekroju poprzecznego piłki (πr^2), a v_x oraz v_y to składowe prędkości piłki w kierunkach odpowiednio poziomym i pionowym.

$$D_x = \frac{1}{2} C\rho S v_x |v_x| \qquad D_y = \frac{1}{2} C\rho S v_y |v_y| \tag{3}$$

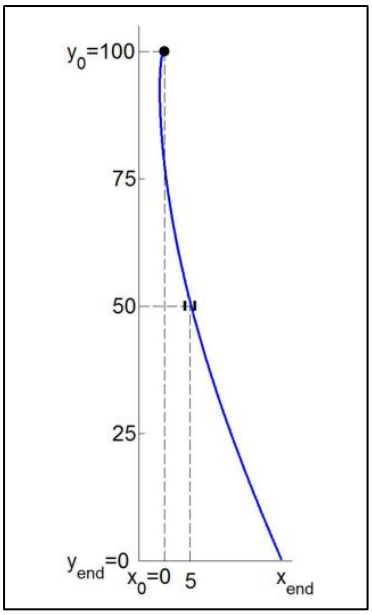
Siła Magnusa to siła boczna działająca na obiekt w ruchu obrotowym, wynikająca z różnicy ciśnienia po obu stronach obiektu. W przypadku piłki, siła Magnusa która powoduje skręcenie trajektorii piłki (jak widać na *Rys.2*) oraz zależy od prędkości piłki i jej rotacji (wzory (4)).

$$F_{Mx} = \rho v_y \omega \pi r^3 \qquad F_{My} = \rho v_x \omega \pi r^3 \tag{4}$$

Łącząc ze sobą wszystkie siły działające na piłkę oraz równania (3) i (4) można zapisać równanie ruchu piłki (5):

$$\begin{cases} m\dfrac{d^2x}{dt^2} + D_x + F_{Mx} = 0 \\ m\dfrac{d^2y}{dt^2} + D_y + F_{My} = -mg \end{cases} \tag{5}$$

Ograniczeniami nakładanymi na przeprowadzaną symulację jest zajście sytuacji, że środek piłki minie punkt (5, 50) w odległości nie większej niż 0,5m (ma to symulować przelecenie piłki przez kosz). Symulacja została przeprowadzona dla 7s, z krokiem czasowym 0,01s.



Rys.2. Ilustracja problemu rzeczywistego

```
matrix df3(double t, matrix Y, matrix ud1, matrix ud2)
{
	// Parameters
	const double m = 0.6;			// Mass [kg]
	const double r = 0.12;			// Radius [m]
	const double g = 9.81;			// Gravitational acceleration [m/s^2]
	const double c = 0.47;			// Air resistance coefficient
	const double rho = 1.2;			// Air density [kg/m^3]
	const double s = M_PI * pow(r, 2);	// Cross-section area [m2]

	// Movement vectors
	double v_x = Y(1);				// Horizontal speed
	double v_y = Y(3);				// Vertical speed
	double v = sqrt(v_x * v_x + v_y * v_y);		// Total speed

	// Forces
	double d_x = 0.5 * c * rho * s * v * v_x;		// Horizontal resistance
	double d_y = 0.5 * c * rho * s * v * v_y;		// Vertical resistance
	double m_x = rho * v_y * ud1(0) * 2 * M_PI * pow(r, 3);	// Magnus's force horizontal
	double m_y = rho * v_x * ud1(0) * 2 * M_PI * pow(r, 3);	// Magnus's force vertical

	// Differential euqations describing motion
	matrix dY(4, 1);
	dY(0) = v_x;				// dx / dt
	dY(1) = -(d_x + m_x) / m;		// dvx / dt
	dY(2) = v_y;				// d / dt
	dY(3) = -(m * g + d_y + m_y) / m;	// dvy / dt

	// Return the state
	return dY;
}
```

Fragment kodu 2: funkcja *df3*

```
matrix ff3R(matrix x, matrix ud1, matrix ud2)
{
	// Starting parameters
	const double x_0 = 0;			// Starting horizontal position
	const double vx_0 = x(0);		// Starting horizontal speed
	const double y_0 = 100;			// Starting vertical position
	const double vy_0 = 0;			// Starting vertical speed
	const double omega_0 = x(1);		// Starting rotation
```

```
 9.        matrix Y_0(4, new double[4] {x_0, vx_0, y_0, vy_0});
10.
11.        // Time
12.        const double start_time = 0.0;
13.        const double end_time = 7.0;
14.        const double time_step = 0.01;
15.
16.        // Resolve differential equation
17.        matrix* result = solve_ode(df3, start_time, time_step, end_time, Y_0, omega_0, NULL);
18.
19.        // Find indecies of y = 0 and y = 50
20.        int y_end_index = 0, y_50_index = 0;
21.        for (int i = 0; i < get_len(result[0]); i++)
22.        {
23.                  double current_y = result[1](i, 2);
24.
25.                  // Check if current Y is closer to 50
26.                  double dist_to_50 = abs(current_y - 50);
27.                  double best_dist_to_50 = abs(result[1](y_50_index, 2) - 50);
28.
29.                  if (dist_to_50 < best_dist_to_50)
30.                            y_50_index = i;
31.
32.                  // Check if current Y is closer to 0
33.                  double dist_to_0 = abs(current_y);
34.                  double best_dist_to_0 = abs(result[1](y_end_index, 2));
35.
36.                  if (dist_to_0 < best_dist_to_0)
37.                            y_end_index = i;
38.        }
39.
40.        // Assign closest values
41.        double x_end = result[1](y_end_index, 0);
42.        double x_50 = result[1](y_50_index, 0);
43.
44.        // Apply penalties
45.        const double scaling_factor = 600000;
46.        double total_penalty = 0.0;
47.
48.        // Penalty for exceeding starting speed
49.        if (abs(vx_0) > 10)
50.                  total_penalty += scaling_factor * pow(abs(vx_0) - 10, 2);
51.
52.        // Penalty for exceeding starting rotation
53.        if (abs(omega_0) > 15)
54.                  total_penalty += scaling_factor * pow(abs(omega_0) - 15, 2);
55.
56.        // Penalty for missing the target at y = 50
57.        if (x_50 < 4.5)
58.                  total_penalty += pow(4.5 - x_50, 2) * scaling_factor;
59.        else if (x_50 > 5.5)
60.                  total_penalty += pow(x_50 - 5.5, 2) * scaling_factor;
61.
62.        // Give a final score
63.        double score = -x_end + total_penalty;
64.
65.        // Free dynamic memory
66.        result[0].~matrix();
67.        result[1].~matrix();
68.
69.        // Return the score
70.        return score;
71. }
72.
```

Fragment kodu 3: funkcja *ff2R*

Implementacja metod optymalizacji

Metoda sympleksu Neldera-Meada

Funkcja *sym_NM* implementuje algorytm sympleksowy Neldera-Meada do znajdowania minimum funkcji celu. Algorytm rozpoczyna się od zainicjowania sympleksu, który składa się z punktu początkowego oraz dodatkowych wierzchołków wygenerowanych wzdłuż podstawowych kierunków skalowanych przez parametr *s*. Wartości funkcji celu są obliczane dla każdego wierzchołka i na ich podstawie algorytm iteracyjnie modyfikuje kształt i położenie sympleksu, dążąc do zbliżenia się do minimum funkcji.

```
 1. solution sym_NM(matrix(*ff)(matrix, matrix, matrix), matrix x0, double s, double alpha,
double beta, double gamma, double delta, double epsilon, int Nmax, matrix ud1, matrix ud2)
 2. {
 3.     try
 4.     {
 5.
 6.         solution Xopt;
 7.         int g_min = 0;
 8.         Xopt.flag = 0;
 9.         // set function's dimension
10.         const int DIM = 2;
11.         // set initial matrixes
12.         matrix p(DIM, 1 + DIM); // DIM (points' dimension) x DIM+1 (points' amount)
square point matrix
13.         matrix e = ident_mat(DIM); // DIM x DIM square identity matrix
14.
15.         p.set_col(x0, 0); // p0 = x0
16.         for (int i = 1; i <= DIM; i++)
17.             p.set_col(p[0] + e[i-1] * s, i); //pi = p0 + ei*s
18.
19.         solution::f_calls += 1 + DIM;
20.         matrix p_f(DIM+1, 1);
21.         for (int i = 0; i <= DIM; i++)
22.             p_f(i) = m2d(ff(p[i], ud1, NAN)); // returns matrix 1x1
23.
24.         double max_norm;
25.         do {
26.             max_norm = 0.0;
27.             int p_max = 0, p_min = 0;
28.             for (int i = 1; i <= DIM; i++) {
29.                 if (p_f(p_max) < p_f(i)) p_max = i;
30.                 if (p_f(p_min) > p_f(i)) p_min = i;
31.             }
32.             if (p_max == p_min)
33.                 p_max = (p_max+1)%(DIM+1);
34.
35.             matrix p_s(DIM,1); // _p
36.             for (int i = 0; i <= DIM; i++)
37.             {
38.                 if (i == p_max) continue;
39.                 p_s.set_col(p_s[0] + p[i], 0); // _p = E(i!=max) pi
40.             }
41.
42.             p_s.set_col(p_s[0] / DIM, 0);  // _p /= n
43.             matrix p_odb = p_s[0] + (p_s[0] - p[p_max]) * alpha; // p_odb = _p
+ a(_p - p_max)
44.
45.             solution::f_calls++;
```

```
                        double p_odb_f = m2d(ff(p_odb, ud1, NAN)); // returns matrix 1x1

                        if (m2d(p_odb_f) < p_f(p_max))
                        {
                                matrix p_e = p_s + (p_odb[0] - p_s[0]) * gamma;

                                solution::f_calls++;
                                double p_e_f = m2d(ff(p_e, ud1, NAN));

                                if (ff(p_e, ud1, NAN) < p_odb_f)
                                {
                                        p.set_col(p_e[0], p_max);
                                        p_f(p_max) = p_e_f;
                                }
                                else
                                {
                                        p.set_col(p_odb[0], p_max);
                                        p_f(p_max) = p_odb_f;
                                }
                        }
                        else
                        {
                                if (p_f(p_min) <= p_odb_f && p_odb_f < p_f(p_max))
                                {
                                        p.set_col(p_odb[0], p_max);
                                        p_f(p_max) = p_odb(0);
                                }
                                else
                                {
                                        matrix p_z = p_s[0] + (p[p_max] - p_s[0])*beta;

                                        solution::f_calls++;
                                        double p_z_f = m2d(ff(p_z, ud1, NAN));

                                        if (p_z_f >= p_f(p_max))
                                        {
                                                for (int i = 0; i <= DIM; i++)
                                                {
                                                   if (i == p_min) continue;
                                                   p.set_col((p[i]+p[p_min])*delta,i);
                                                   solution::f_calls++;
                                                   p_f(i) = m2d(ff(p[i], ud1, NAN));
                                                }
                                        }
                                        else
                                        {
                                                p.set_col(p_z[0], p_max);
                                                p_f(p_max) = p_z_f;
                                        }
                                }
                        }
                        g_min = p_min;
                        if (solution::f_calls > Nmax)
                        {
                                Xopt.flag = -2;
                                break;
                                //throw("Nie znaleziono przedzialu po Nmax probach
(f_calls > Nmax)\n");
                        }
                        for (int i = 0; i <= DIM; i++)
                        {
                                if (i == p_min) continue;
                                double i_norm = norm(p[p_min] - p[i]);
                                if (i_norm > max_norm)
                                        max_norm = i_norm;
                        }
                        //std::cout << max_norm << std::endl;
                } while (max_norm >= epsilon);
                Xopt.x = p[g_min];
                Xopt.y = p_f(g_min);
```

```
115. //std::cout << "NM: " << Xopt << std::endl;
116.                 return Xopt;
117.         }
118.         catch (string ex_info)
119.         {
120.                 throw ("solution sym_NM(...):\n" + ex_info);
121.         }
122. }
```

Fragment kodu 4: funkcja *sym_NM*

Metoda funkcji kary

Funkcja *pen* implementuje metodę kar w połączeniu z algorytmem Neldera-Meada w celu znalezienia minimum funkcji celu, która może zawierać ograniczenia. Proces iteracyjnie modyfikuje wagę kar i optymalizuje funkcję celu, dążąc do znalezienia rozwiązania zgodnego z ograniczeniami.

```
 1. solution pen(matrix(*ff)(matrix, matrix, matrix), matrix x0, double c, double dc, double
epsilon, int Nmax, matrix ud1, matrix ud2)
 2. {
 3.         try {
 4.                 solution xi(x0), x_i;
 5.                 xi.flag = 0;
 6.                 matrix init_v_S(2, 1);
 7.                 init_v_S(0) = c;
 8.                 init_v_S(1) = ud2(0);
 9.                 double nm = 0;
10.                 //double tmp1 = m2d(ff3T(xi.x)), tmp2;
11.                 do {
12.                         x_i = xi;
13.                         //tmp2 = tmp1;
14.                         xi = sym_NM(ff,xi.x,ud1(0),ud1(1),ud1(2),ud1(3), ud1(4), ud1(5),
Nmax, init_v_S);
15.                         //tmp1 = m2d(ff3T(xi.x));
16. //std::cout << "PEN:\nx:" << x_i.x(0) << " " << x_i.x(1) << " y: " << xi.y << "\nx:" <<
xi.x(0) << " " << xi.x(1) << " y: " << xi.y << "\n";
17.                         init_v_S(0) = init_v_S(0) * dc;
18.                         if (solution::f_calls > Nmax){
19.                                 xi.flag = -2;
20.                                 break;
21.                         }
22.                         if (dc < 1.0)
23.                         {
24.                                 double sum = 0.0;
25.                                 sum += 1 / m2d(g1(xi.x, NAN));
26.                                 sum += 1 / m2d(g2(xi.x, NAN));
27.                                 sum += 1 / m2d(g3(xi.x, init_v_S(1)));
28.                                 if (c * fabs(sum) < epsilon)
29.                                         break;
30.                         }
31.                         nm = norm(xi.x - x_i.x);
32.                 } while (nm >= epsilon);
33.                 return xi;
34.         }
35.         catch (string ex_info) { throw ("matrix pen(...):\n" + ex_info); }
36. }
```

Fragment kodu 5: funkcja *pen*

Podczas wykonywanych obliczeń korzystano zarówno z funkcji kary wewnętrznej, jak i zewnętrznej danymi poniższymi wzorami (6), a następnie dodano ich implementację w pliku *user_fun.h*, jak pokazano we *Fragmencie kodu 6*.

- dla funkcji kary zewnętrznej: $S(x_1, x_2) = \sum_{i=1}^{n} \left(max\left(0, g_i(x_1, x_2)\right)\right)^2$
- dla funkcji kary wewnętrznej: $S(x_1, x_2) = -\sum_{i=1}^{n} \frac{1}{g_i(x_1, x_2)}$ (6)

```
1. template <int k, matrix(*f)(matrix, matrix, matrix) >
2. matrix SEF(matrix x, matrix c_and_other, matrix ud1 = NAN)
3. {
4.       if (k >= n_sets || k < 0)
5.            throw("matrix SEF(matrix x, matrix ud1, matrix ud2): nie ma takiego zbioru" + k);
6.       double sum = 0;
7.       for (int i = 0; i < n_fun; i++)
8.       {
9.                if (gl_g_tab[k][i] != nullptr)
10.               sum += pow(max(0.0,m2d(gl_g_tab[k][i](x, c_and_other(1)))),2);
11.      }
12.      //std::cout << "SEF: " << (f(x, NAN, NAN) + sum * c_and_other(0)) << endl;
13.      return (f(x, NAN, NAN) + sum * c_and_other(0));
14. }
15.
16. template <int k, matrix(*f)(matrix, matrix, matrix)>
17. matrix SIF(matrix x, matrix c_and_other, matrix ud1 = NAN)
18. {
19.      if (k >= n_sets || k < 0)
20.          throw("matrix SIF(matrix x, matrix ud1, matrix ud2): nie ma takiego zbioru " + k);
21.      double sum = 0;
22.      for (int i = 0; i < n_fun; i++)
23.      {
24.               if(gl_g_tab[k][i] != nullptr)
25.               sum += 1.0/fabs(m2d(gl_g_tab[k][i](x, c_and_other(1))));
26.      }
27.      //std::cout << "SIF: " << sum * c_and_other(0) << endl;
28.
29.      return (f(x, NAN, NAN) - sum * c_and_other(0));
30. }
```

Fragment kodu 6: funkcja *SEF* oraz *SIF*

Opracowanie wyników

Zadeklarowano zmienne lokalne wykorzystywane podczas późniejszego testowania (*Fragment kodu 7*). Składają się na nie:

- parametry zakresu początkowego:
  - *x0_min*, *x1_min* – dolne ograniczenia dla generowania losowych wartości początkowych.
- parametry kary:

- *sif_c* – początkowa wartość parametru kary dla wewnętrznych ograniczeń;
- *sif_dc* – mnożnik, przez który wartość *sif_c* jest skalowana w kolejnych iteracjach, aby stopniowo zbliżać się do zera;
- *sef_c* – początkowa wartość parametru kary dla zewnętrznych ograniczeń;
- *sef_dc* – mnożnik, przez który *sef_c* jest skalowany, aby zwiększać wpływ kar, dążąc do nieskończoności

- ogólne parametry:
  - *epsilon* – zadana dokładność;
  - *Nmax* – maksymalna liczba iteracji.

```
1.      // Range for defined local minimums
2.      double x0_min = 1.0, x1_min = 1.0;  // Boundries for random number generation
3.
4.      double sif_c = 5.0;          // c size for internal
5.      double sif_dc = 0.5;         // c -> 0
6.
7.      double sef_c = 5.0;          // c size for external
8.      double sef_dc = 2.0;         // c -> inf
9.
10.     // Common arguments
11.     double epsilon = 1e-3;
12.     int Nmax = 2000;
```

Fragment kodu 7: zmienne lokalne niezbędne dla działania badanych funkcji

Testowa funkcja celu

Badania przeprowadzono dla funkcji kary wewnętrznej zewnętrznej. Wartości uzyskanych wyników zestawiono w *Tabeli 1*, a następnie w *Tabeli 2* zebrano ich uśrednione wartości. Dla każdego z trzech promieni (*value_a*) generuje 100 losowych punktów początkowych *x0*, które są odpowiednio ograniczane do danego promienia. Wyniki optymalizacji, takie jak współrzędne początkowe i końcowe, wartości funkcji celu, liczba wywołań funkcji celu i odległość od środka, są zapisywane do pliku tekstowego.

```
1. //initial values for sym_NM
2. matrix init_v_sym_NM = matrix(6, 1);
3.
4. init_v_sym_NM(0) = 1.0;            //double side_size = 0.5;
5. init_v_sym_NM(1) = 1.0;            //double reflection_fator = 1.0;
6. init_v_sym_NM(2) = 0.5;            //double narrowing_factor = 0.5;
7. init_v_sym_NM(3) = 2.0;            //double expansion_factor = 2.0;
8. init_v_sym_NM(4) = 0.5;            //double reduction_factor = 0.5;
9. init_v_sym_NM(5) = epsilon;
10. const char delimiter = '\t';
11. const string OUTPUT_PATH = "Output/lab_3/";
12. ofstream tfun_file(OUTPUT_PATH + "out_1_tfun.txt");
13. if (!tfun_file.is_open())
14.       return;
15.
```

```
16.
17. const int k = 3;
18. matrix value_a[k] = {matrix(1,1,4.0),matrix(1,1,4.4934),matrix(1,1,5)};
19.
20. for (int i = 0; i < k; i++) {
21.       for (int o = 0; o < 100; o++) {
22.                 matrix x0(2, 1);
23.                 x0(0)=x0_min+static_cast<double>(rand()/ RAND_MAX*(m2d(value_a[i]) - x0_min);
24.                 x0(1)x1_min+ static_cast<double>(rand())/RAND_MAX*(m2d(value_a[i]) - x1_min);
25.                 double r_init = sqrt(pow(x0(0), 2) + pow(x0(1), 2));
26.                 if (r_init > value_a[i])
27.                 {
28.                           if (x0(0) > x0(1))
29.                                     x0(0) -= r_init - m2d(value_a[i]);
30.                           else
31.                                     x0(1) -= r_init - m2d(value_a[i]);
32.                 }
33.
34.                 tfun_file << x0(0) << delimiter << x0(1) << delimiter;
35.                 solution k = pen(SEF<0, ff3T >, x0, sef_c, sef_dc, epsilon, Nmax,
                    init_v_sym_NM, value_a[i]);
36.                 k.fit_fun(ff3T);
37.                 tfun_file << k.x(0) << delimiter << k.x(1) << delimiter << sqrt(pow(k.x(0),
                    2) + pow(k.x(1), 2)) << delimiter;
38.                 tfun_file << m2d(k.y) << delimiter << solution::f_calls << delimiter;
39.                 solution::clear_calls();
40.
41.                 k=pen(SIF<0,ff3T>,x0,sif_c,sif_dc, epsilon, Nmax, init_v_sym_NM, value_a[i]);
42.                 k.fit_fun(ff3T);
43.                 tfun_file << k.x(0) << delimiter << k.x(1) << delimiter << sqrt(pow(k.x(0),
                    2) + pow(k.x(1), 2)) << delimiter;
44.                 tfun_file << m2d(k.y) << delimiter << solution::f_calls << std::endl;
45.                 solution::clear_calls();
46.       }
47. }
```

Fragment kodu 8: fragment funkcji *main* zawierający funkcjonalności dla *Tabeli 1* oraz *Tabeli 2*

Dla testowej funkcji celu badano zarówno karę zewnętrzną, jak i wewnętrzną. W przypadku zewnętrznej funkcji kary, dla każdego przypadku wartość promienia r^* jest bardzo bliska parametrowi a. Wartości y^* są ujemne, a ich moduł rośnie wraz z większym a. Liczba wywołań jest najniższa dla $a = 4.4934$.

Natomiast w przypadku wewnętrznej funkcji kary optymalizacje również znajdują punkty bliskie a, ale r^* jest bardziej odległe od a niż w przypadku funkcji kary zewnętrznej, szczególnie dla mniejszych wartości a. Tylko dla $a = 4$ wewnętrza funkcja kary wymaga mniejszej liczby wywołań funkcji celu.

Funkcja zewnętrzna kary skuteczniej prowadzi do lepszych wyników (niższe y^*) i wymaga mniej wywołań funkcji celu (w dwóch z trzech przypadków), co czyni ją bardziej efektywną. Obie funkcje kary prowadzą do rozwiązań bliskich wartościom a, jednak funkcja zewnętrzna lepiej spełnia wymagania optymalizacji.

Tabela 2. Wartości średnie wyników optymalizacji dla trzech współczynników ekspansji

Parametr *a*	Zewnętrzna funkcja kary				
	x_1*	x_2*	r*	y*	Liczba wywołań funkcji celu
4	2,803384767	2,797861977	4,00088186	-0,189302965	537,8023256
4,4934	3,0819254	3,007871	4,4933994	-0,217234	209,66
5	3,1031989	2,998815	4,49341	-0,217234	213,32

Parametr *a*	Wewnętrzna funkcja kary				
	x_1*	x_2*	r*	y*	Liczba wywołań funkcji celu
4	2,52163589	2,69187109	4,0105827	-0,18274811	263,73
4,4934	2,978723818	2,866566232	4,487195758	-0,217131919	256,3131313
5	3,2482873	3,2482941	4,9993657	-0,19178905	249,94

Problem rzeczywisty

W przypadku problemu rzeczywistego program został uruchomiony jedynie raz, a jego wyniki zostały zebrane w *Tabeli 3*. Celem uzyskania wyników wykorzystano kod załączony na *Fragmencie kodu 8*, będący częścią funkcji *main*. Dla problemu rzeczywistego użyto tylko funkcji kary zewnętrznej.

```
1.      // Starting conditions
2.      double start_velocity = 5.0;  // [m/s]
3.      double omega = 10.0;          // [rad/s]
4.      matrix x0(2, new double[2]{start_velocity, omega});
5.
6.      // Optimization
7.      double penalty_start = 100.0;
8.      double penalty_adjustment = 0.1;
9.      double penalty_multiplier = 1e7;
10.     solution opt_res = pen(ff3R, x0, penalty_start, penalty_adjustment, epsilon, Nmax,
        init_v_sym_NM, penalty_multiplier);
```

Fragment kodu 9: fragment funkcji *main* zawierający funkcjonalności dla *Tabeli 3*

Zestawione w *Tabeli 3* wyniki pokazuję, że optymalizacja znalazła wartości prędkości i rotacji, które najlepiej maksymalizują odległość przelotu piłki. Negatywna wartość v_{0x} jest zgodna z przewidywaniami – najpierw piłka leci w przeciwną stronę, aby potem, pod wpływem rotacji zmienić kierunek na przeciwny. Wysoka rotacja (14,9987 rad/s) ma duży wpływ na zmianę

trajektorii piłki, co w połączeniu z siłą oporu powietrza daje pożądany rezultat, tzn. piłka ląduje w odpowiednim miejscu na boisku, jednocześnie przelatując przez kosz (co pokazuje *Rys. 3*).

Tabela 3. Zestawienie wyników optymalizacji dla problemu rzeczywistego

$v_{0x}^{(0)}$	$\omega^{(0)}$	v_{ox}*	ω*	x_{end}*	Liczba wywołań funkcji celu
5	10	-4,14847	14,9987	28,7895	198

Następnie przygotowano kod do przeprowadzenia symulacji dla obu metod optymalizacji (*Fragment kodu 9*). Piłka spada z wysokości 100 metrów i porusza się z początkową prędkością poziomą oraz rotacją. Na początku kod ustawia początkowe parametry: pozycję poziomą (*x_0*), początkową wysokość (*y_0*), początkową prędkość pionową (*vy_0*), oraz początkową rotację piłki (*omega_0*) i początkową prędkość poziomą (*vx_0*), które są wynikiem wcześniejszej optymalizacji (zapisane w *opt_res*). Zebrane dane zapisywane są do plików z których następnie będzie możliwe wykonanie wykresu trajektorii piłki. Symulacja została przeprowadzona dla 7s, z krokiem czasowym 0.01s.

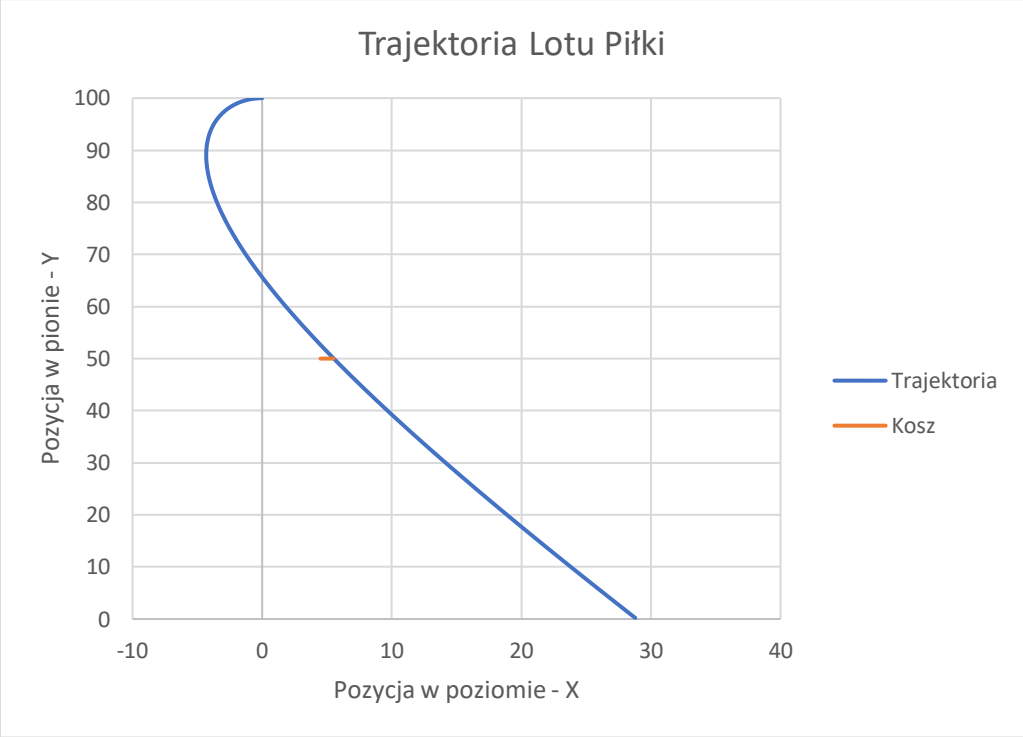
```
1.      // ----- Simulation -----
2.
3.      // Save optimized parameters
4.      double opt_velocity = opt_res.x(0);
5.      double opt_rotation = opt_res.x(1);
6.
7.      // Starting parameters
8.      const double x_0 = 0;                   // Starting horizontal position
9.      const double vx_0 = opt_res.x(0);       // Starting horizontal speed
10.     const double y_0 = 100;                 // Starting vertical position
11.     const double vy_0 = 0;                  // Starting vertical speed
12.     const double omega_0 = opt_res.x(1);    // Starting rotation
13.     matrix Y_0(4, new double[4] {x_0, vx_0, y_0, vy_0});
14.
15.     // Time
16.     const double start_time = 0.0;
17.     const double end_time = 7.0;
18.     const double time_step = 0.01;
19.
20.     // Resolve differential equation
21.     matrix* result = solve_ode(df3, start_time, time_step, end_time, Y_0, omega_0, NULL);
22.
23.     // Save the result
24.     for (int i = 0; i < get_len(result[0]); ++i)
25.     {
26.             double t_i = result[0](i, 0);
27.             double X_i = result[1](i, 0);
28.             double Y_i = result[1](i, 2);
29.
30.             sim_file << t_i << "\t" << X_i << "\t" << Y_i << "\n";
31.     }
```

Fragment kodu 9: fragment funkcji *main* pozwalający na zebranie odpowiednich danych do utworzenia odpowiednich wykresów

Niebieska linia pokazuje trajektorię lotu piłki. Pika zostaje rzucona w płaszczyźnie poziomej w lewą stronę (w kierunku ujemnej osi OX), natomiast potem zakręca w prawo, co wynika z efektu rotacji piłki (siła Magnusa), która powoduje skręcanie trajektorii.

Piłka przelatuje przez punkt na wysokości około 50 m (czerwona kreska na wykresie zawierająca dopuszczalny zakres kosza), czyli symulacja spełnia wymaganie przelecenia przez kosz. Również patrząc na dane zebrane podczas tworzenia symulacji, można zauważyć, że piłka przelatuje przez kosz – w punkcie (5,49991; 50,0711) około 3,83 sekundy.

Pomimo że czas symulacji został ustawiony na 7 sekund, to piłka uderza w ziemię po 6 sekundach. Algorytm dalej zwraca wyniki, ale wówczas wartości *y* są już ujemne, co nie ma oddania w rzeczywistości.



Rys. 3. Wykres trajektorii lotu piki

Wnioski

Wyniki z przeprowadzonej analizy wskazują, że zarówno zewnętrzna, jak i wewnętrzna funkcja kary prowadzą do wyników bliskich optymalnym wartościom parametru a, jednak zewnętrzna funkcja kary okazuje się bardziej efektywna w kontekście optymalizacji. Dla zewnętrznej funkcji kary, wartości promienia r^* są bardzo bliskie wartości parametru a, a liczba wywołań funkcji celu jest najniższa dla $a = 4.4934$. W przypadku wewnętrznej funkcji kary, r^* jest bardziej oddalone od a, szczególnie przy mniejszych wartościach a, a liczba wywołań funkcji celu nie jest tak korzystna jak w przypadku funkcji zewnętrznej.

Optymalizacja przy użyciu funkcji zewnętrznej kary prowadzi do uzyskania lepszych wyników (niższe wartości y^*), co sugeruje, że funkcja ta skuteczniej spełnia wymagania optymalizacji. Uzyskane wyniki zmierzają w kierunku granicy, co wynika z bardzo zbliżonych wartości optymalizacji dla obu funkcji kary. W miarę zbliżania się do granicy, wartość sumy funkcji kary dąży do nieskończoności. Powoduje to przybliżanie rozwiązania w kierunku granicy, nawet jeśli w tym obszarze nie występuje żadne rzeczywiste minimum.

Minimum funkcji celu jest stosunkowo płaskie. W rezultacie, rozwiązania znajdujące się na granicy wydają się być dominujące. Ponadto, różnica między wartością funkcji celu a sumą funkcji kary staje się coraz mniejsza w miarę zbliżania się do granicy. To dodatkowo wzmacnia tendencję do „wypychania” rozwiązania w pobliże granicy, niezależnie od położenia rzeczywistego minimum.

W przypadku optymalizacji trajektorii lotu piłki, algorytm wyznacza wartości prędkości i rotacji, które maksymalizują odległość przelotu piłki, co jest zgodne z założeniami modelu. Negatywna wartość początkowej prędkości poziomej ($v0x$) jest uzasadniona, a wysoka rotacja w połączeniu z siłą oporu powietrza skutkuje zakrzywieniem trajektorii, co pozwala piłce przelecieć przez punkt na wysokości 50 m, co jest zgodne z wymaganiami symulacji.