AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Sprawozdanie 6

Optymalizacja metodami niedeterministycznymi

Autorzy: *Paulina Grabowska*
Filip Rak
Arkadiusz Sala

Kierunek studiów: *Inżynieria Obliczeniowa*

Kraków, 2025

Spis treści

Cel ćwiczenia

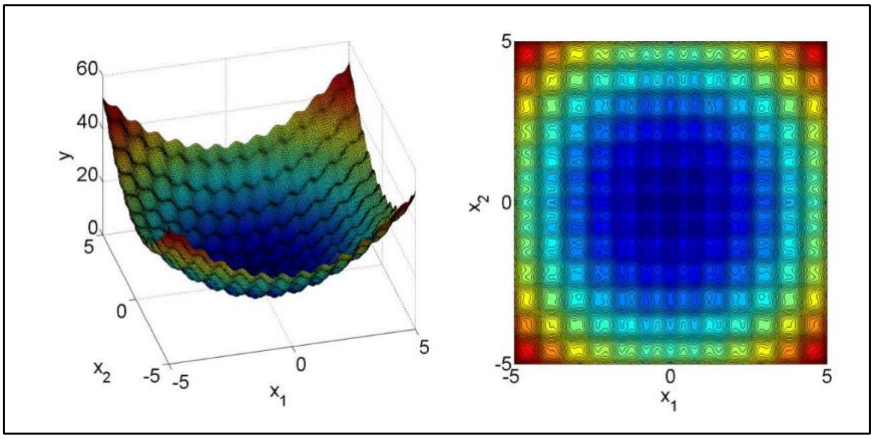
Celem przeprowadzonego ćwiczenia było pogłębienie wiedzy na temat niedeterministycznych metod optymalizacji oraz algorytmu ewolucyjnego. Zostały one zaimplementowane i zastosowane do rozwiązywania problemów optymalizacyjnych dla funkcji wielu zmiennych, umożliwiając praktyczne zrozumienie ich działania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

Optymalizowane problemy

Testowa funkcja celu

Celem praktycznego wykorzystania zaimplementowanych algorytmów badano funkcję celu podaną wzorem (1) zwizualizowaną na *Rys. 1*. Obrany punkt startowy będzie należał do przedziału [-5, 5] (dla obu zmiennych).

$$f(x_1, x_2) = x_1^2 + x_2^2 - cos(2{,}5\pi x_1) - cos(2{,}5\pi x_2) + 2 \qquad (1)$$



Rys. 1: wykres funkcji (1).

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

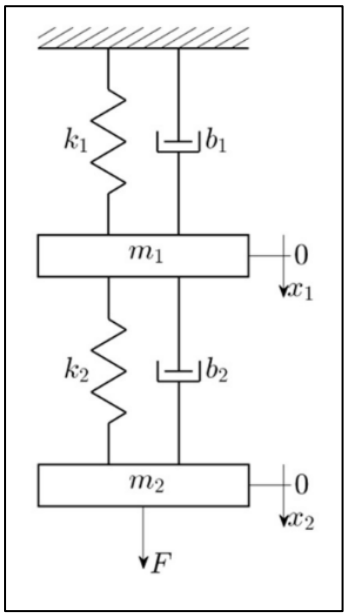
```
matrix ff6_T(matrix x, matrix ud1, matrix ud2)
{
	static double d_2o5_pi = 2.5 * pi;
	return x(0) * x(0) + x(1) * x(1) - cos(d_2o5_pi * x(0)) - cos(d_2o5_pi * x(1)) + 2.0;
}

```

Fragment kodu 1: funkcja *ff6_T*

Problem rzeczywisty

Analizowany problem rzeczywisty dotyczy wyznaczenia wartości współczynników tłumienia b_1 oraz b_2, odpowiadających za opór ruchu w układzie dwóch ciężarków (*Rys. 2*), dla których przeprowadzono symulację ruchu. Celem jest znalezienie wartości b_1 (odpowiadającego za opór ruchu ciężarka górnego) oraz b_2 (odpowiadającego za tłumienie interakcji między ciężarkami) należących do przedziału [0.1, 3].



Rys. 2: wizualizacja problemu rzeczywistego

Układ składa się z dwóch ciężarków o masach m_1 oraz m_2 równych 5 kg. Każdy ciężarek jest zawieszony na sprężynie o współczynnikach sprężystości k_1 oraz k równych 1 N/m. Położenie ciężarków oznaczone jest zmiennymi x_1 oraz x_2.

Do dolnego ciężarka została przyłożona siła *F* równa 1N. Ruch górnego ciężarka w układzie można opisać równaniem (2), a ruch dolnego ciężarka równaniem (3).

$$m_1\ddot{x}_1 + b_1\dot{x}_1 + b_2(\dot{x}_1 - \dot{x}_2) + k_1x_1 + k_2(x_1 - x_2) = 0 \quad (2)$$

$$m_2\ddot{x}_2 - b_2(\dot{x}_1 - \dot{x}_2) - k_2(x_1 - x_2) = F \quad (3)$$

Używane we wzorach (2) i (3) pochodne stanu układu w danym momencie czasowym są obliczane za pomocą funkcji *ff6R_derivative* (*Fragment kodu 2a*). Stan układu jest opisany wektorem *Y*, który zawiera pozycje i prędkości ciężarków

```
matrix ff6R_derivative(double time, matrix Y, matrix dampness, matrix ud2)
{
	// Physical constants
	double upper_mass = 5.0, lower_mass = 5.0;
	double upper_spring = 1.0, spring_coupling = 1.0;
	double external_force = 1.0;

	// Damping coefficients
	double upper_damping = dampness(0, 0);
	double damping_coupling = dampness(1, 0);

	// State
	double upper_position = Y(0, 0), lower_position = Y(2, 0);
	double upper_velocity = Y(1, 0), lower_velocity = Y(3, 0);

	// Get acceleration
	double upper_acceleration = (-upper_damping * upper_velocity - damping_coupling
		* (upper_velocity - lower_velocity) - upper_spring
		* upper_position - spring_coupling * (upper_position - lower_position))
		/ upper_mass;

	double lower_acceleration = (external_force + damping_coupling
		* (upper_velocity - lower_velocity) + spring_coupling
		* (upper_position - lower_position))
		/ lower_mass;

	// Pack derivatives
	matrix dY(4, 1);
	dY(0, 0) = upper_velocity;
	dY(1, 0) = upper_acceleration;
	dY(2, 0) = lower_velocity;
	dY(3, 0) = lower_acceleration;

	// Return derivatives
	return dY;
}

```

Fragment kodu 2a: funkcja *ff6R_derivative*

Symulacja ruchu układu dla podanych współczynników tłumienia b_1 oraz b_2 została zaimplementowana w funkcji *ff6R_motion* (*Fragment kodu 2b*). Funkcja wywołuje solver równań różniczkowych korzystający z funkcji *ff6R_derivative*. Wynik symulacji to macierz zawierająca położenia obu ciężarków w czasie.

```
matrix ff6R_motion(matrix dampness)
{
	// Damping coefficients
	double upper_damping = dampness(0, 0);
	double damping_coupling = dampness(1, 0);

	// Time parameters
	double start_time = 0.0;
	double stop_time = 100.0;
	double time_step = 0.1;
```

```
11.
12.       // Initial state
13.       double upper_position = 0.0, lower_position = 0.0;
14.       double upper_velocity = 0.0, lower_velocity = 0.0;
15.       matrix Y0(4, new double[4]{upper_position, upper_velocity, lower_position,
lower_velocity});
16.
17.       // Solve differential equation
18.       matrix* solution = solve_ode(ff6R_derivative, start_time, time_step, stop_time, Y0,
dampness);
19.
20.       // Only copy the positions
21.       int rows = 1001;
22.       matrix positions(rows, 2);
23.
24.       for (int i = 0; i < rows; i++)
25.       {
26.                 positions(i, 0) = solution[1](i, 0);
27.                 positions(i, 1) = solution[1](i, 2);
28.       }
29.
30.       delete[] solution;
31.
32.       // Return the positions
33.       return positions;
34. }
35.
```

Fragment kodu 2b: funkcja *ff6R_motion*

Dodatkowo zaimplementowano funkcję *ff6R_error* (*Fragment kodu 2c*), która oblicza błąd pomiędzy wynikami symulacji a rzeczywistymi danymi z eksperymentu. Wykorzystuje różnicę w położeniach ciężarków (x_1 oraz x_2) w każdym kroku czasowym. Funkcja pozwala na ocenę, jak dobrze model odzwierciedla rzeczywisty ruch układu.

```
 1. double ff6R_error(matrix simulation_results, matrix reference_data)
 2. {
 3.       // Get the size of the matrix
 4.       int *size = get_size(simulation_results);
 5.       int rows = size[0];
 6.       delete[] size;
 7.
 8.       // Calculate error
 9.       double error = 0.0;
10.       for (int i = 0; i < rows; ++i)
11.       {
12.                 double sim_upper_pos = simulation_results(i, 0);
13.                 double sim_lower_pos = simulation_results(i, 1);
14.                 double ref_upper_pos = reference_data(i, 0);
15.                 double ref_lower_pos = reference_data(i, 1);
16.
17.                 error += pow(sim_upper_pos - ref_upper_pos, 2) + pow(sim_lower_pos -
ref_lower_pos, 2);
18.       }
19.
20.       return error;
21. }
22.
```

Fragment kodu 2c: funkcja *ff6R_error*

Funkcja *ff6R* (*Fragment kodu 2d*) integruje wszystkie funkcje, by przeprowadzić optymalizację parametrów tłumienia.

```
matrix ff6R(matrix x, matrix ud1, matrix ud2)
{
	// Debug: Print the number of function calls
	static int calls = 0;
	calls += 1;
	if (calls % 100 == 0)
	{
		std::cout << "ff6R calls: " << calls << "\n";
	}

	// Run simulation for given dampness coefficients
	matrix simulation_results = ff6R_motion(x);

	// Return the error
	return ff6R_error(simulation_results, ud2);
}

```

Fragment kodu 2d: funkcja *ff6R*

Implementacja metod optymalizacji

Metoda ewolucyjna

Algorytmy ewolucyjne są inspirowane procesami biologicznej ewolucji, takimi jak selekcja naturalna, rekombinacja i mutacja. Są używane do rozwiązywania problemów optymalizacyjnych, zwłaszcza gdy funkcja celu jest nieciągła, wielowymiarowa, lub trudna do analitycznego rozwiązania. Populacja potencjalnych rozwiązań ewoluuje w kolejnych pokoleniach, dążąc do maksymalizacji (lub minimalizacji) wartości funkcji celu.

Funkcja *EA* (*Fragment kodu 3*) optymalizuje funkcję celu w przestrzeni wielu wymiarów. Proces rozpoczyna się od losowej inicjalizacji populacji rozwiązań w dopuszczalnych granicach, gdzie każde rozwiązanie opisane jest przez wektor parametrów, odchylenia standardowe oraz wartość funkcji celu. Następnie przeprowadzana jest selekcja, w której wybierane są najlepsze osobniki na podstawie ich wyników.

Wybrane osobniki poddaje się rekombinacji, łącząc ich parametry, oraz mutacji, wprowadzając losowe zmiany w celu eksploracji przestrzeni poszukiwań. Każdy nowo powstały osobnik jest oceniany za pomocą funkcji celu, a nowa populacja powstaje przez zastąpienie najgorszych osobników najlepszymi z połączonego zbioru starej i nowej populacji. Proces ten powtarza się

iteracyjnie, aż zostanie spełnione kryterium zakończenia, takie jak osiągnięcie określonego błędu lub maksymalnej liczby iteracji.

```
solution EA(matrix(*ff)(matrix, matrix, matrix), int N, matrix lb, matrix ub, int mi, int lambda, double sigma0,
	double epsilon, int Nmax, matrix ud1, matrix ud2)
{
	try
	{

		static double alpha = std::sqrt((double)N);
		static double beta = std::pow(2.0 * (double)N, -0.25);

		static std::random_device rd{};
		static std::mt19937 gen{ rd() };

		static std::normal_distribution<> dst{ 0.0, 1.0 };
		static double max_gen = static_cast<double>(gen.max());

		static struct x_y_sigma {
			matrix x;
			matrix sigma;
			double y;
		};

		const auto init_f = [](const x_y_sigma& a, const x_y_sigma& b) {
			return (a.y) < (b.y);
			};

		static auto randU = []() -> double {
			return (double)rand() / (double)RAND_MAX;
			};

		static auto randN = [&]() -> double {
			double u1 = randU();
			double u2 = randU();
			return sqrt(-2.0 * log(u1)) * cos(2.0 * 3.1415 * u2);
			};


		std::set<x_y_sigma, decltype(init_f)> P_x_y(init_f);
		for (int i = 0; i < mi; i++)
		{

			x_y_sigma init_P = { matrix(N,1), matrix(N,1), 0.0 };

			for (int d = 0; d < N; d++)
			{
				//double r = (double)(gen())/max_gen;
				double r = randU();
				double xd = lb(d, 0) + r * (ub(d, 0) - lb(d, 0));
				init_P.x(d, 0) = xd;
				init_P.sigma(d, 0) = sigma0;
			}

			init_P.y = m2d(ff(init_P.x, ud1, ud2));
			solution::f_calls++;
			P_x_y.insert(init_P);
		}
		int i = 0;
		//std::cout << "pp\n";
		while (true)
		{

			if (P_x_y.begin()->y <= epsilon)
			{
				solution Xopt(P_x_y.begin()->x);
```

```
                    Xopt.y = P_x_y.begin()->y;
                    Xopt.flag = 0;
                    return Xopt;
            }

            if (solution::f_calls >= Nmax)
            {
                    solution Xopt(P_x_y.begin()->x);
                    Xopt.y = P_x_y.begin()->y;
                    Xopt.flag = 0;
                    return Xopt;
            }

            auto first = P_x_y.begin();

            vector<double> fi(mi), q(mi + 1, 0.0);

            double sumFi = 0.0;
            for (int j = 0; j < mi; j++)
            {
                    sumFi += fi[j] = 1.0 / first->y;
                    first++;
            }

            for (int j = 1; j <= mi; j++)
            {
                    q[j] = q[j - 1] + (fi[j - 1] / sumFi);
            }

            std::set<x_y_sigma, decltype(init_f)> T_x_y(init_f);

            for (int off = 0; off < lambda; off++)
            {

                    //double rA = (double)(gen()) / max_gen;
                    double rA = randU();
                    int idA = 0;
                    for (int j = 1; j <= mi; j++)
                    {
                            if (rA <= q[j])
                            {
                                    idA = j - 1;
                                    break;
                            }
                    }

                    //double rB = (double)(gen()) / max_gen;
                    double rB = randU();
                    int idB = 0;
                    for (int j = 1; j <= mi; j++)
                    {
                            if (rB <= q[j])
                            {
                                    idB = j - 1;
                                    break;
                            }
                    }

                    x_y_sigma child = { matrix(N,1), matrix(N,1), 0.0 };

                    //double rC = (double)(gen()) / max_gen;
                    double rC = randU();

                    auto A = P_x_y.begin();
                    std::advance(A, idA);
                    auto B = P_x_y.begin();
                    std::advance(B, idB);

                    //double rG = dst(gen);
                    double rG = randN();
```

```
134.
135.                                     for (int d = 0; d < N; d++)
136.                                     {
137.                                               double xA = A->x(d, 0);
138.                                               double xB = B->x(d, 0);
139.                                               double sA = A->sigma(d, 0);
140.                                               double sB = B->sigma(d, 0);
141.
142.                                               child.x(d, 0) = rC * xA + (1.0 - rC) * xB;
143.
144.                                               child.sigma(d, 0) = 0.5 * (sA + sB);
145.                                     }
146.
147.                                     for (int d = 0; d < N; d++)
148.                                     {
149.                                               //double rL = dst(gen);
150.                                               double rL = randN();
151.                                               double sigma_old = child.sigma(d, 0);
152.
153.                                               double sigma_new = sigma_old * exp(alpha * rG +
beta * rL);
154.                                               child.sigma(d, 0) = sigma_new;
155.
156.                                               //double step = sigma_new * dst(gen);
157.                                               double step = randN();
158.                                               child.x(d, 0) += step;
159.
160.                                               // Granice
161.                                               /*if (child.x(d, 0) < lb(d, 0))
162.                                                         child.x(d, 0) = lb(d, 0);
163.                                               if (child.x(d, 0) > ub(d, 0))
164.                                                         child.x(d, 0) = ub(d, 0);*/
165.                                     }
166.
167.                                     child.y = m2d(ff(child.x, ud1, ud2));
168.                                     solution::f_calls++;
169.                                     T_x_y.insert(child);
170.
171.                           }
172.
173.                           std::set<x_y_sigma, decltype(init_f)> new_P_x_y(init_f);
174.
175.
176.                           new_P_x_y.insert(P_x_y.begin(), P_x_y.end());
177.                           new_P_x_y.insert(T_x_y.begin(), T_x_y.end());
178.
179.                           P_x_y.clear();
180.                           P_x_y.insert(new_P_x_y.begin(), std::next(new_P_x_y.begin(),mi));
181.
182.                 }
183.       }
184.       catch (string ex_info)
185.       {
186.                 throw("solution EA(...):\n" + ex_info);
187.       }
188. }
189.
```

Fragment kodu 3: funkcja *EA*

Opracowanie wyników

Zadeklarowano zmienne lokalne dla problemu testowego i rzeczywistego wykorzystywane podczas późniejszego testowania – dokładność *epsilon*, liczba osobników populacji bazowej *mi*, liczba potomków dla każdej iteracji *lambd* oraz maksymalną liczbę iteracji *Nmax* (*Fragment kodu 4*).

```
1.       matrix
2.                lb(2, std::unique_ptr<double[]>(new double[2] { -5.0, -5.0 }).get()),
3.                ub(2, std::unique_ptr<double[]>(new double[2] { 5.0, 5.0 }).get());
4.       //return;
5.       double epsilon = 1e-2;
6.       int mi = 5;
7.       int lambd = 10;
8.       int Nmax = 1e+5;
9.
10.  . . .
11.
12.      // Optimizer settings
13.      double rp_lower_bound = 0.f;
14.      double rp_upper_bound = 3.f;
15.      matrix rp_lb(2, std::unique_ptr<double[]>(new double[2] {rp_lower_bound,
rp_lower_bound}).get());
16.      matrix rp_ub(2, std::unique_ptr<double[]>(new double[2] {rp_upper_bound,
rp_upper_bound}).get());
17.
18.      double rp_epsilon = 1e-2, rp_sigma = 0.1f;
19.      int rp_mi = 5, rp_lambda = 10;
20.      int rp_n_max = 1e4, rp_n = 2;
21.
```

Fragment kodu 4: zmienne lokalne niezbędne dla działania badanych funkcji

Testowa funkcja celu

Analiza obejmuje przeprowadzenie 100 optymalizacji dla każdej z pięciu różnych wartości początkowych współczynnika mutacji ($\sigma \in \{0.01, 0.1, 1, 10, 100\}$). Dla optymalizacji zakończonych sukcesem (tj. osiągnięciem minimum globalnego), obliczono wartości średnie i zaprezentowano je w *Tabeli 2*. Wyniki każdej optymalizacji, takie jak wartości parametrów, wynik funkcji celu, liczba wywołań funkcji oraz status powodzenia, są zapisywane do pliku tekstowego (*Fragment kodu 5*).

```
1.                  ofstream tfun_file1(OUTPUT_PATH + "out_1_tfun.txt");
2.                  if (!tfun_file1.good()) return;
3.                  int pop = 0;
4.                  double sigma_v[5] = { 0.01, 0.1, 1., 10., 100. };
5.                  for (int i = 0; i < 5; i++)
6.                  {
7.                            //std::cout << sigma_v[i] << std::endl;
8.                            for (int j = 0; j < 100; j++)
9.                            {
10.                                     solution is1 = EA(ff6_T, 2, lb, ub, mi, lambd,
sigma_v[i], epsilon, Nmax);
11.                                     tfun_file1 << is1.x(0) << delimiter
12.                                               << is1.x(1) << delimiter
13.                                               << is1.y(0) << delimiter
14.                                               << solution::f_calls << delimiter
15.                                               << (is1.y(0) < epsilon) << std::endl;
16.                                     if (solution::f_calls > Nmax) pop++;
17.                                     solution::clear_calls();
18.
19.                           }
20.                 std::cout << "POP: " << pop << std::endl;
```

Fragment kodu 5: fragment funkcji *main* zawierający funkcjonalności do analizy problemu testowego

Algorytm ewolucyjny w każdej konfiguracji początkowego zakresu mutacji (σ) był w stanie znaleźć minimum globalne, co jest świadectwem jego wysokiej skuteczności. Liczba minimów globalnych wynosi zawsze 100, co wskazuje, że algorytm dobrze radzi sobie z zadaniem eksploracji przestrzeni rozwiązań, niezależnie od wartości σ.

Wartości x_1^*, x_2^* oraz y^* są bardzo bliskie siebie dla różnych wartości σ, co sugeruje stabilność rozwiązania. Wartość y^*, będąca wartością funkcji celu w minimum, utrzymuje się w zakresie od 0,0047 do 0,0054. Najmniejsza liczba wywołań funkcji celu została osiągnięta dla σ = 0,1, wynosząc 5205,8. Największa liczba wywołań wystąpiła dla σ = 100, co może sugerować, że zbyt duży zakres mutacji prowadzi do wolniejszej zbieżności.

Tabela 2: Wartości średnie optymalizacji zakończonych sukcesem dla problemu rzeczywistego

Początkowa wartość zakresu mutacji	x_1^*	x_2^*	y^*	Liczba wywołań funkcji celu	Liczba minimów globalnych
0,01	0,000534266	0,000295957	0,004698715	6012,8	100
0,1	0,001520279	-0,000377255	0,005130077	5205,8	100
1	-0,000351079	0,000383564	0,005350854	5925,5	100
10	-0,000974889	0,000764726	0,004741474	5918,9	100
100	0,000870511	-0,001411263	0,005035358	7066,3	100

Problem rzeczywisty

W ramach analizy problemu rzeczywistego przeprowadzono pojedynczą optymalizację, a otrzymane wyniki zestawiono w *Tabeli 3*. Dla wyznaczonych wartości optymalnych b_1 oraz b_2 wykonano wykonać symulację, na podstawie której sporządzono sporządzić wykres przedstawiający położenie ciężarków.

Poniższy kod (*Fragment kodu 6*) wykorzystuje algorytm ewolucyjny do minimalizacji błędu między wynikami symulacji a danymi referencyjnymi (*ref_data*). Następnie, na podstawie zoptymalizowanych wartości tłumienia, wykonywana jest symulacja (*ff6R_motion*), której wyniki są zapisywane i porównywane z danymi referencyjnymi, a obliczony błąd (*ff6R_error*) jest wypisywany na konsolę.

```
1.        // Problem data
 2.       int rows = 1001, cols = 2;
 3.       matrix ref_data(rows, cols);
 4.       ref_file >> ref_data;
 5.
 6.       matrix rp_ud1 = NAN;
 7.       matrix rp_ud2 = ref_data;
 8.
 9.       // Optimize dampness values (b)
10.       solution::clear_calls();
11.       solution rp_solution = EA(ff6R, rp_n, rp_lb, rp_ub, rp_mi, rp_lambda, rp_sigma,
rp_epsilon, rp_n_max, rp_ud1, rp_ud2);
12.       rp_opt_file << rp_solution;
13.
14.       // Run simulation with optimized dampness
15.       matrix sim_res = ff6R_motion(rp_solution.x);
16.       sim_file << sim_res;
17.
18.       // Print error of simulation
19.       matrix error = ff6R_error(sim_res, ref_data);
20.       std::cout << "RP_error: " << error << "\n";
21.
```

Fragment kodu 6: fragment funkcji *main* zawierający funkcjonalności do analizy problemu rzeczywistego

Tabela 3: Wyniki uzyskane w wyniku optymalizacji dla problemu rzeczywistego

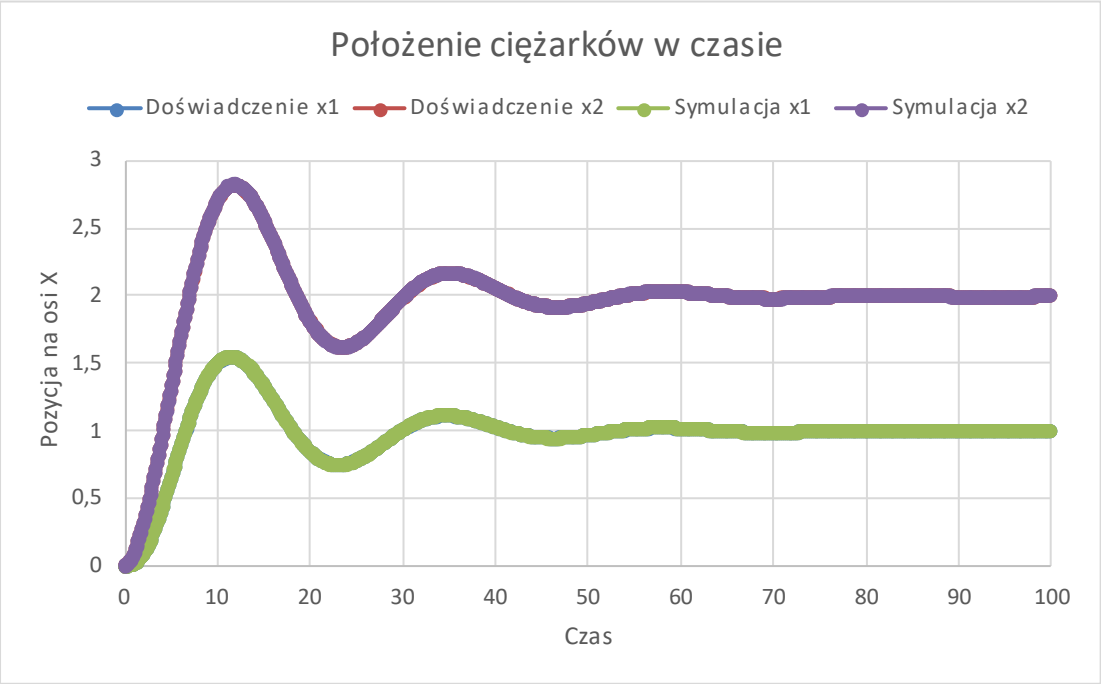
b_1*	b_2*	y*	Liczba wywołań funkcji celu
2,5106	1,48957	0,00109436	245

Wykres (*Rys. 3*) przedstawia porównanie wyników symulacji z wynikami uzyskanymi w doświadczeniu dla układu dwóch ciężarków. Linie zielona i fioletowa przedstawiają wyniki

przeprowadzanej symulacji dla x_1 oraz x_2, natomiast niebieska i czerwona – dane wyniki doświadczenia.

Analiza wykresu pokazuje, że symulacja dobrze odwzorowuje ogólne zachowanie układu, zwłaszcza w zakresie dynamiki początkowych drgań. W przypadku x_1 (górny ciężarek) trajektoria symulacji niemal dokładnie pokrywa się z wynikami eksperymentalnymi, wskazując na dobrze zoptymalizowany współczynnik tłumienia b_1, odpowiadający za opór ruchu górnego ciężarka. Również dla x_2 (dolny ciężarek) trajektoria symulacji fioletowa pokrywa się z trajektorią czerwoną, co świadczy o poprawnym doborze b_2, tłumiącego interakcję między ciężarkami.

Wyniki symulacji bardzo dobrze odwzorowują wyniki eksperymentalne, co świadczy o skutecznej optymalizacji współczynników tłumienia b_1 oraz b_2. Model jest szczególnie skuteczny w odwzorowaniu dynamiki początkowej układu, co ma kluczowe znaczenie dla analizy układów dynamicznych.



Wnioski

W przypadku problemu testowego niezależnie od początkowej wartości zakresu mutacji, metoda potrafi znaleźć podobne rozwiązanie w przestrzeni przeszukiwań. Różnice w y^* między różnymi σ są minimalne, co może świadczyć o tym, że dobór zakresu mutacji nie wpływa znacząco na jakość końcowego rozwiązania w tym przypadku.

Dla σ = 0,1 osiągnięto najniższą liczbę wywołań funkcji celu, co może oznaczać, że mniejsze mutacje sprzyjają szybszej zbieżności algorytmu, szczególnie w obszarze bliskim optimum globalnego. W przypadku σ = 100, liczba wywołań funkcji celu jest największa, co wynika prawdopodobnie z dużych kroków mutacji, które utrudniają precyzyjną eksplorację bliskiego otoczenia minimum globalnego i wydłużają proces poszukiwania.

Zakres mutacji σ jest kluczowym parametrem w algorytmach ewolucyjnych, odpowiadającym za balans między eksploracją (poszukiwaniem nowych rozwiązań w odległych częściach przestrzeni), a eksploatacją (precyzyjnym dostrajaniem rozwiązań w pobliżu minimum).

Małe wartości σ (np. 0,01; 0,1) skutkują mniejszymi perturbacjami, co sprzyja szybkiemu dostrajaniu rozwiązania w obszarach bliskich optimum. Jednak zbyt małe wartości mogą ograniczać zdolność algorytmu do eksploracji przestrzeni i prowadzić do utknięcia w lokalnych minimach (co jednak w tym przypadku nie miało miejsca). Duże wartości σ (np. 100) powodują większe kroki w przestrzeni przeszukiwań, co zwiększa zdolność eksploracyjną algorytmu, ale może spowolnić konwergencję, zwłaszcza w zaawansowanych etapach optymalizacji, gdy potrzebna jest precyzyjna eksploracja bliskiego otoczenia rozwiązania.

Wyniki symulacji oraz analiza wykresu pozwalają na stwierdzenie dotyczących skuteczności zastosowanego modelu optymalizacji współczynników tłumienia b_1 oraz b_2. Model bardzo dobrze odwzorowuje dynamikę układu w początkowej fazie ruchu, co świadczy o poprawnym sformułowaniu równań opisujących dynamikę ciężarków oraz skuteczności metody ewolucyjnej w optymalizacji parametrów. Położenie ciężarków w symulacji niemal idealnie pokrywa się z wynikami eksperymentalnymi.