Jakub Latawiec

Rafał Malik

Optymalizacja

Laboratorium – Optymalizacja metodami niedeterministycznymi.

1. **Cel ćwiczenia**
   Celem ćwiczenia jest zapoznanie się z niedeterministycznymi metodami optymalizacji poprzez ich implementację oraz wykorzystanie do wyznaczenia minimum podanej funkcji celu.

2. **Przeprowadzenie ćwiczenia**
   a) Implementacja algorytmu ewolucyjnego

```
solution EA(matrix(*ff)(matrix, matrix, matrix), int N, matrix lb, matrix ub, int mi, int lambda, matrix sigma0, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
	try
	{
		solution* population = new solution[mi + lambda]; // Populacja (rodzice + potomkowie)
		solution* best_mu = new solution[mi]; // Najlepsi osobnicy

		matrix inv_fitness(mi, 1), temp_individual(N, 2);

		double r, cumulative_sum, total_inv_fitness;
		double tau = 1.0 / sqrt(2 * N), tau_prime = 1.0 / sqrt(2 * sqrt(N)); // Współczynniki mutacji

		int worst_idx; // Indeks najgorszego osobnika

		// Inicjalizacja populacji początkowej
		for (int i = 0; i < mi; ++i)
		{
			population[i].x = matrix(N, 2);

			for (int j = 0; j < N; ++j)
			{
				population[i].x(j, 0) = (ub(j) - lb(j)) * m2d(rand_mat()) + lb(j);
				population[i].x(j, 1) = sigma0(0);
			}

			population[i].fit_fun(ff, ud1, ud2);

			if (population[i].y < epsilon)
			{
				population[i].flag = 1;
				return population[i];
			}
		}

		// Główna pętla ewolucji
		while (true)
		{
			total_inv_fitness = 0;
```

```
                        for (int i = 0; i < mi; ++i)
                        {
                                inv_fitness(i) = 1 / population[i].y(0);
                                total_inv_fitness += inv_fitness(i);
                        }

                        // Selekcja rodziców
                        for (int i = 0; i < lambda; ++i)
                        {
                                r = total_inv_fitness * m2d(rand_mat());
                                cumulative_sum = 0;
                                for (int j = 0; j < mi; ++j)
                                {
                                        cumulative_sum += inv_fitness(j);
                                        if (r <= cumulative_sum)
                                        {
                                                population[mi + i] = population[j];
                                                break;
                                        }
                                }
                        }

                        // Mutacja potomków
                        for (int i = 0; i < lambda; ++i)
                        {
                                r = m2d(randn_mat());
                                for (int j = 0; j < N; ++j)
                                {
                                        population[mi + i].x(j, 1) *= exp(tau_prime * r +
tau * m2d(randn_mat()));
                                        population[mi + i].x(j, 0) += population[mi +
i].x(j, 1) * m2d(randn_mat());
                                }
                        }

                        // Krzyżowanie
                        for (int i = 0; i < lambda; i += 2)
                        {
                                r = m2d(rand_mat());
                                temp_individual = population[mi + i].x;
                                population[mi + i].x = r * population[mi + i].x + (1 -
r) * population[mi + i + 1].x;
                                population[mi + i + 1].x = r * population[mi + i + 1].x
+ (1 - r) * temp_individual;
                        }

                        // Ocena funkcji celu dla potomków
                        for (int i = 0; i < lambda; ++i)
                        {
                                population[mi + i].fit_fun(ff, ud1, ud2);
                                if (population[mi + i].y < epsilon)
                                {
                                        population[mi + i].flag = 1;
                                        return population[mi + i];
                                }
                        }

                        // Selekcja mi najlepszych osobników
                        for (int i = 0; i < mi; ++i)
                        {
                                worst_idx = 0;
                                for (int j = 1; j < mi + lambda; ++j)
                                {
```

```
                                    if (population[worst_idx].y > population[j].y)
                                          worst_idx = j;
                              }
                              best_mu[i] = population[worst_idx];
                              population[worst_idx].y = 1e10;
                        }

                        // Aktualizacja populacji bazowej
                        for (int i = 0; i < mi; ++i)
                              population[i] = best_mu[i];

                        if (solution::f_calls > Nmax)
                              break;
                  }

                  population[0].flag = 0;
                  return population[0];
            }
            catch (string ex_info)
            {
                  throw ("solution EA(...):\n" + ex_info);
            }
}
```

b) Implementacja testowej funkcji celu

```
matrix ff6T(matrix x, matrix ud1, matrix ud2)
{
      return pow(x(0), 2) + pow(x(1), 2) - cos(2.5 * M_PI * x(0)) - cos(2.5 *
M_PI * x(1)) + 2;
}
```

c) Implementacja funkcji różniczkowej dla problemu rzeczywistego

```
matrix df6(double t, matrix Y, matrix ud1, matrix ud2)
{
      matrix dY(4, 1);

      const double m_1 = 5.0;
      const double m_2 = 5.0;
      const double k_1 = 1.0;
      const double k_2 = 1.0;
      const double F = 1.0;

      double b_1 = ud2(0);
      double b_2 = ud2(1);

      dY(0) = Y(1);
      dY(1) = (-b_1 * Y(1) - b_2 * (Y(1) - Y(3)) - k_1 * Y(0) - k_2 * (Y(0) -
Y(2))) / m_1;
      dY(2) = Y(3);
      dY(3) = (F + b_2 * (Y(1) - Y(3)) + k_2 * (Y(0) - Y(2))) / m_2;

      return dY;
}
```

d) Implementacja funkcji celu dla problemu rzeczywistego

```
matrix ff6R(matrix x, matrix ud1, matrix ud2)
{
	matrix y = 0;

	matrix Y0 = matrix(4, new double[4] {0.0, 0.0, 0.0, 0.0});
	matrix* Y = solve_ode(df6, 0.0, 0.1, 100.0, Y0, ud1, x[0]);

	for (int i = 0; i < 1001; ++i)
	{
		y = y + abs(ud2(i, 0) - Y[1](i, 0)) + abs(ud2(i, 1) - Y[1](i, 2));
	}
	y(0) = y(0) / (2 * ud1(0));

	return y;
}
```

e) Implementacja funkcji lab6()

```
void lab6()
{
#ifdef SAVE_TO_FILE
	create_environment("lab06");
#endif

	//Dane dokładnościowe
	double epsilon = 1E-4;
	int Nmax = 10000;

#ifdef CALC_TEST
	//Stringstream do zapisu danych
	std::stringstream test_ss;

	//Rozwiązanie dla wyników testowych
	solution test_sol;

	//Punkty startowe
	matrix test_x0{};

	//Długości kroków
	double sigma_arr[] = { 0.01, 0.1, 1.0, 10.0, 100.0 };

	//Zakresy
	matrix lb_test = matrix(2, new double[2] {-5.0, -5.0});
	matrix ub_test = matrix(2, new double[2] {5.0, 5.0});

	//Liczba populacji
	int mi_test = 20;
	int lambda_test = 40;

	for (auto sigma : sigma_arr)
	{
		for (int i = 0; i < 100; ++i)
		{
			test_sol = EA(ff6T, 2, lb_test, ub_test, mi_test, lambda_test,
sigma, epsilon, Nmax);
			test_ss << test_sol.x(0) << ";" << test_sol.x(1) << ";" <<
test_sol.y(0) << ";" << test_sol.f_calls << ";" << (solution::f_calls > Nmax ?
"NIE" : "TAK") << "\n";
			solution::clear_calls();
		}
	}
```

```
#ifdef SAVE_TO_FILE
	save_to_file("test.csv", test_ss.str());
#endif
#endif

#ifdef CALC_SIMULATION
	//Parametry zadania
	matrix lb = matrix(2, new double[2] {0.1, 0.1});
	matrix ub = matrix(2, new double[2] {3.0, 3.0});
	matrix sigma0 = matrix(2, new double[2] {0.1, 0.1});
	int mi = 50;
	int lambda = 100;

	//Pobieranie danych z pliku
	matrix x1_x2_data = file_reader::fileToMatrix(1001, 2,
"../input_data/lab06/polozenia.txt");

	solution opt = EA(ff6R, 2, lb, ub, mi, lambda, sigma0, epsilon, Nmax, 1001,
x1_x2_data);
	std::cout << opt << "\n";

	matrix y;
	matrix Y0 = matrix(4, new double[4] {0.0, 0.0, 0.0, 0.0});
	matrix* Y = solve_ode(df6, 0.0, 0.1, 100.0, Y0, NAN, opt.x[0]);
#ifdef SAVE_TO_FILE
	save_to_file("simulation.csv", Y[1]);
#endif
#endif
}
```

3. **Parametry algorytmów**
   a) <u>Dla testów:</u>
   Dokładność (epsilon): 1E-4
   Maksymalna ilość wywołań funkcji (Nmax): 10 000
   Intensywność mutacji (sigma): [0.01, 0.1, 1.0, 10.0, 100.0]
   Ograniczenia: (lb i ub): [-5.0, 5.0]
   Liczba rodziców w populacji bazowej (mi): 20
   Liczba potomków (lambda): 40

   b) <u>Dla problemu rzeczywistego:</u>
   Dokładność (epsilon): 1E-4
   Maksymalna ilość wywołań funkcji (Nmax): 10 000
   Intensywność mutacji (sigma): 0.1
   Ograniczenia: (lb i ub): [0.1, 3.0]
   Liczba rodziców w populacji bazowej (mi): 50
   Liczba potomków (lambda): 100

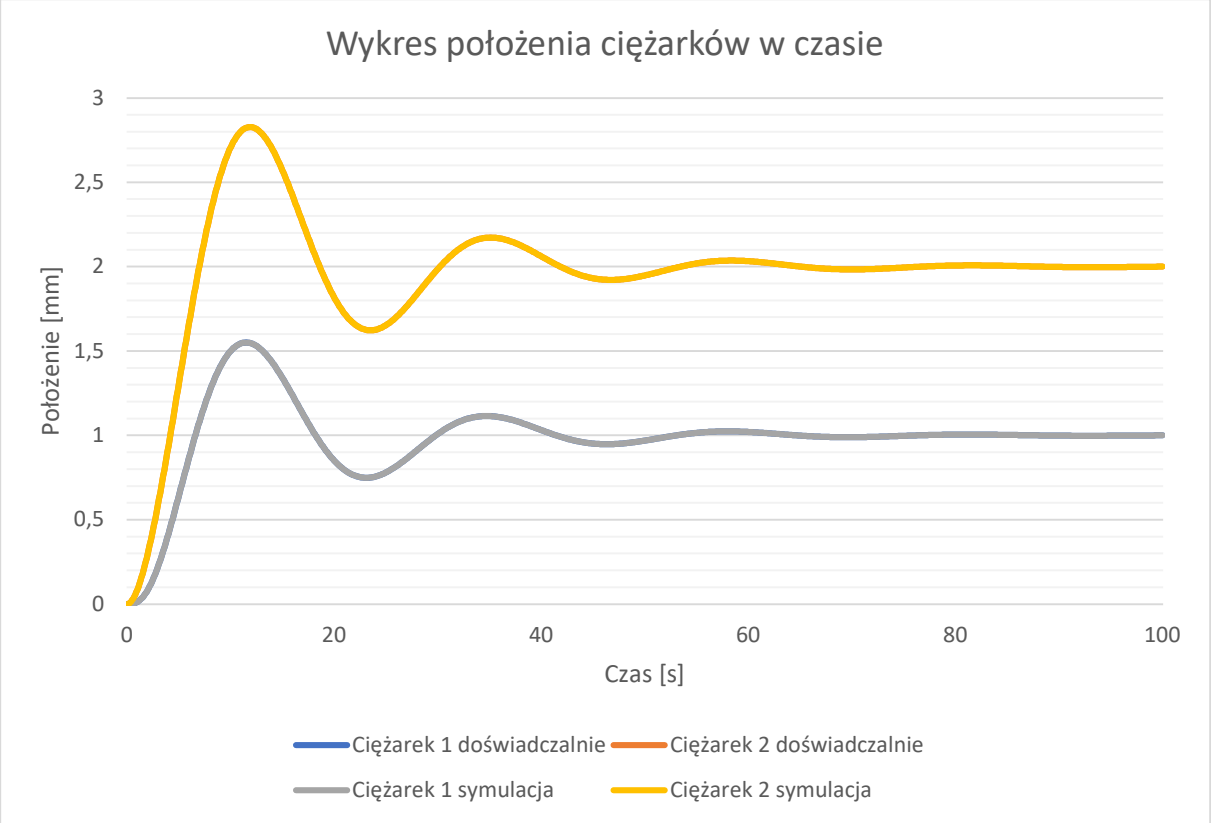
4. **Dyskusja wyników**
   a) <u>Wyniki testowe:</u>
   Po przeprowadzeniu 100 optymalizacji dla różnych wartości intensywności mutacji i odrzuceniu wyników nie znajdujących minimów, otrzymaliśmy następujące wartości średnie:

Początkowa wartość zakresu mutacji	x_1*	x_2*	y*	Liczba wywołań funkcji celu	Liczba minimów globalnych
0,01	-1,99E-05	-1,66E-05	4,82E-05	400	68
0,1	1,87E-04	-1,52E-04	4,95E-05	603	91
1	1,74E-05	1,30E-05	4,68E-05	984	99
10	-1,55E-04	-1,66E-05	5,19E-05	1884	97
100	1,18E-05	2,70E-05	4,86E-05	5544	84

Z powyższych danych widzimy że zwiększanie wartości epsilon, zwiększa ilość znajdowanych minimów globalnych kosztem zwiększenia ilości obliczeń. Wyjątkiem jest epsilon równy 100, gdzie skuteczność zaczyna maleć a ilość wywołań funkcji rośnie. Najbardziej optymalną wartością intensywności mutacji jest 10, w kontekście znalezienia najwięcej minimów. Dodatkowo średnia wartość funkcji jest najmniejsza dla epsilon = 10.

b) <u>Wyniki problemu rzeczywistego:</u>

Za pomocą algorytmu ewolucyjnego udało znaleźć się współczynniki oporu ruchu dla obu ciężarków. Wynoszą one: b1 = 1.5, b2 = 2.5. Po wykonaniu symulacji ruchu obu ciężarków dla znalezionych współczynników i porównaniu ich z wartościami z doświadczenia, otrzymaliśmy wykres:



Na wykresie nie są widoczne linie związane z danymi z doświadczenia, co oznacza że nasze dane z symulacji pokrywają się z danymi z doświadczenia z całkiem dużą dokładnością.

5. Wnioski

Skuteczność przeprowadzania optymalizacji niedeterministyczną metodą ewolucji zależy od dobrania odpowiedniego współczynnika intensywności mutacji. Na dokładność wyników może wpływać również ilość osobników w populacji ich potomków.