Kierunek Inżynieria Obliczeniowa	**Temat laboratoriów** Ćwiczenie 6 - Optymalizacja metodami niedeterministycznymi	**Data ćwiczenia** 08.01.2025, 15.01.2025
Przedmiot Optymalizacja	Karolina Kurowska Szymon Majdak Amelia Nalborczyk	**Grupa** 2

Cel

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

Przebieg ćwiczenia

Zadanie 1 funkcja testowa celu

Na ćwiczeniach zaimplementowano algorytmy optymalizacji metodami niedeterministycznymi:

1. Algorytm ewolucyjny

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

        vector<solution> P(mi + lambda);
        vector<solution> selected_P(mi);

        matrix phi(mi, 1), temp_P(N, 2);

        double r, phi_sum_sum, phi_sum;
        double alpha = pow(2 * N, -0.5);
        double beta = pow(2 * pow(N, 0.5), -0.5);

        // populacja poczatkowa
        for (int i = 0; i < mi; ++i) {
            P[i].x = matrix(N, 2);
            P[i].x(0, 0) = lb(0) + (lb(1) - lb(0)) * m2d(rand_mat());
            P[i].x(0, 1) = sigma(0);
            P[i].x(1, 0) = ub(0) + (ub(1) - ub(0)) * m2d(rand_mat());
            P[i].x(1, 1) = sigma(0);
            P[i].fit_fun(ff, ud1, ud2);

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

        while (true) {

            // wagi selekcji
            phi_sum = 0;
            for (int i = 0; i < mi; ++i) {
                phi(i) = 1 / P[i].y(0);
                phi_sum += phi(i);
```

```
        }

        // generowanie potomkow
        for (int i = 0; i < lambda; ++i) {
            r = phi_sum * m2d(rand_mat());
            phi_sum_sum = 0;

            for (int j = 0; j < mi; ++j) {
                phi_sum_sum += phi(j);
                if (r <= phi_sum_sum) {
                    P[mi + i] = P[j];
                    break;
                }
            }
        }

        // krzyzowanie par potomkow
        for (int i = 0; i < lambda; i += 2) {
            r = m2d(rand_mat());
            temp_P = P[mi + i].x;

            P[mi + i].x = r * P[mi + i].x + (1 - r) * P[mi + i + 1].x;
            P[mi + i + 1].x = r * P[mi + i + 1].x + (1 - r) * temp_P;
        }

        // mutacja potomkow
        for (int i = 0; i < lambda; ++i) {
            double rand_gauss = m2d(randn_mat());

            for (int j = 0; j < N; ++j) {
                P[mi + i].x(j, 1) *= exp(beta * rand_gauss + alpha * m2d(randn_mat()));
                P[mi + i].x(j, 0) += P[mi + i].x(j, 1) * m2d(randn_mat());
            }
        }
```

```
        // obliczanie funkcji celu dla nowych potomków
        for (int i = 0; i < lambda; ++i) {
            P[mi + i].fit_fun(ff, ud1, ud2);

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

        // wybor najlepszych osobników
        for (int i = 0; i < mi; ++i) {

            int j_min = 0;
            for (int j = 1; j < mi + lambda; ++j) {
                if (P[j].y < P[j_min].y) {
                    j_min = j;
                }
            }

            selected_P[i] = P[j_min];
            P[j_min].y = 1e20;
        }

        for (int i = 0; i < mi; ++i) {
            P[i] = selected_P[i];
        }

        if (solution::f_calls > Nmax) {
            P[0].flag = -1;
            cout << "f_calls > Nmax" << endl;
            return P[0];
        }
    }
```

2. Funkcja testowa

```
matrix ff6T(matrix x, matrix ud1, matrix ud2) {

    matrix y;

    y = pow(x(0), 2) + pow(x(1), 2) - cos(2.5 * M_PI * x(0)) - cos(2.5 * M_PI * x(1)) + 2;

    return y;

}
```

3. Fragment funkcji main zapewniającej zapis wszystkich potrzebnych danych z problemu testowego do jednego pliku csv.

```
//SEKCJA TOSTOWA
ofstream file("lab_06_tostowe.csv");
if (!file.is_open())throw string("PLIK CSV NIE OTWARTY");
file << "x1, x2, y, f_calls\n";

double sigma[] = { 0.01, 0.1, 1, 10, 100 };
int sigma_size = 5;
int N = 2;

matrix lb(N, 1);
lb(0) = -5;
lb(1) = -5;

matrix ub(N, 1);
ub(0) = 5;
ub(1) = 5;

int mi = 25;
int lambda = 50;
double epsilon = 1e-5;
int Nmax = 100000;

for (int j = 0; j < sigma_size; j++) {
    for (int i = 0; i < 100; i++)
    {
        solution ea = EA(ff6T, N, lb, ub, mi, lambda, sigma[j], epsilon, Nmax);
        cout << ea;

        file
            << ea.x(0) << ","
            << ea.x(1) << ","
            << ea.y(0) << ","
            << solution::f_calls << ","
            << "\n";
    }
}

file.close();
```

Wykonujemy zadanie polegające na wykonaniu 100 optymalizacji dla pięciu różnych wartości początkowych współczynnika mutacji (σ=0.01, 0.1, 1, 10, 100). Punkt startowy jest losowany z przedziału $x_1 \in [-5; 5]$, $x_2 \in [-5; 5]$. Funkcja testowa jest postaci:

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

Wykonujemy optymalizację metodą ewolucyjną. Wyniki optymalizacji zebrano w arkuszu "Tabela 1" w pliku Excel, a wyniki średnie znajdują się w arkuszu “Tabela 2” oraz poniżej.

Początkowa wartość zakresu mutacji	x_1*	x_2*	y*	Liczba wywołań funkcji celu	Liczba minimów globalnych
0,01	-0,00005141	0,00772824	0,00620944	3517,77000000	99
0,1	0,00001120	0,00001904	0,00000566	1910,14000000	100
1	0,00001017	0,00771885	0,00620957	3754,21000000	99
10	0,00002712	0,00775300	0,00620974	3246,20000000	99
100	0,00003624	0,00003728	0,00000462	3743,88000000	100

Najlepsze wyniki osiągnięto dla zakresu mutacji 0.1, gdzie wartość $y*$ była najbliższa zeru, liczba wywołań funkcji celu najniższa, a liczba minimów globalnych wynosiła 100, co wskazuje na pełną stabilność. Dla zakresu 0.01, choć wyniki były poprawne, optymalizacja wymagała większej liczby wywołań funkcji celu. Zakresy 1 i 10 generowały podobne wartości, ale z większą liczbą wywołań funkcji celu, co obniżało efektywność. Dla zakresu 100 uzyskano dobrą wartość i 100 minimów globalnych, jednak liczba wywołań funkcji celu była wysoka. Optymalnym wyborem pozostaje zakres 0.1 dzięki równowadze między precyzją a efektywnością.

Zadanie 2 problem rzeczywisty

Następnym zadaniem jest przeprowadzenie symulacji. Dwa ciężarki o masach m1=5kg, m2=5kg są zawieszone na sprężynach o współczynnikach k1=k2=1N/m. Na drugi ciężarek działa siła F=1N. Ruch układu opisują równania różniczkowe uwzględniające siły sprężystości, opór ruchu, zdefiniowany przez współczynniki b1, b2 oraz wzajemne oddziaływanie ciężarków. Celem jest znalezienie wartości b1, b2 $\in$ [0.1,3.0] Ns/m, które najlepiej odwzorują dane eksperymentalne zapisane w pliku polozenia.txt. Na ich podstawie stworzyliśmy wykres wykres przedstawiający położenie
ciężarków naniesione na wykres położenia ciężarków uzyskany w doświadczeniu.

W celu rozwiązania tego problemu stworzyliśmy funkcje odwzorowujące dane równania różniczkowe, które zostały zapisane w pliku user_funs.cpp. Funkcje te zamieszczamy poniżej razem z resztą kodów używanych do przeprowadzenia symulacji:

1. Funkcja main

```
// SEKCJA PRAWDZIWEGO PROBLEMU
int mi = 5;
int lambda = 10;
double epsilon = 1e-1;
int N_max = 100000;

double sigma[] = {0.01, 0.1, 1, 10, 100};
int N = 2;

matrix lb_real (N, 1), ub_real (N, 1);

matrix lb (N, 1);
lb (0) = 0.1;
lb (1) = 0.1;

matrix ub (N, 1);
ub (0) = 3.0;
ub (1) = 3.0;
matrix sigma_for_real (N, 1, sigma[0]);

solution::clear_calls ();
matrix ud1;
matrix ud2_positions = load_from_txt("polozenia.txt");

solution sotulion = EA(ff6R, N, lb, ub, mi, lambda, sigma_for_real, epsilon, N_max, ud1, ud2_positions);

cout << "b1 = " << sotulion.x (0, 0) << ", b2 = " << sotulion.x (1, 0) <<
    ", Y = " << sotulion.y (0, 0) << ", f_calls = " << solution::f_calls << endl;
```

2. Funkcja pomocnicza

```
matrix load_file (const std::string& filename){
    std::ifstream input_file (filename);
    if (!input_file){
        std::cerr << "Nie udało się otworzyć pliku: " << filename << std::endl;
        return matrix ();
    }

    std::vector<double> first_column, second_column;
    std::string line_content;

    while (std::getline (input_file, line_content)){
        std::transform (line_content.begin (), line_content.end (), line_content.begin (), [](char c) {
            return (c == ',' ? '.' : (c == ';' ? ' ' : c));
            });

        std::istringstream line_stream (line_content);
        double value1 = 0.0, value2 = 0.0;
        if (line_stream >> value1 >> value2){
            first_column.push_back (value1);
            second_column.push_back (value2);
        }
    }

    input_file.close ();

    size_t total_rows = first_column.size ();
    matrix output_matrix (total_rows, 2);

    for (size_t row = 0; row < total_rows; ++row){
        output_matrix (row, 0) = first_column[row];
        output_matrix (row, 1) = second_column[row];
    }

    return output_matrix;
}
```

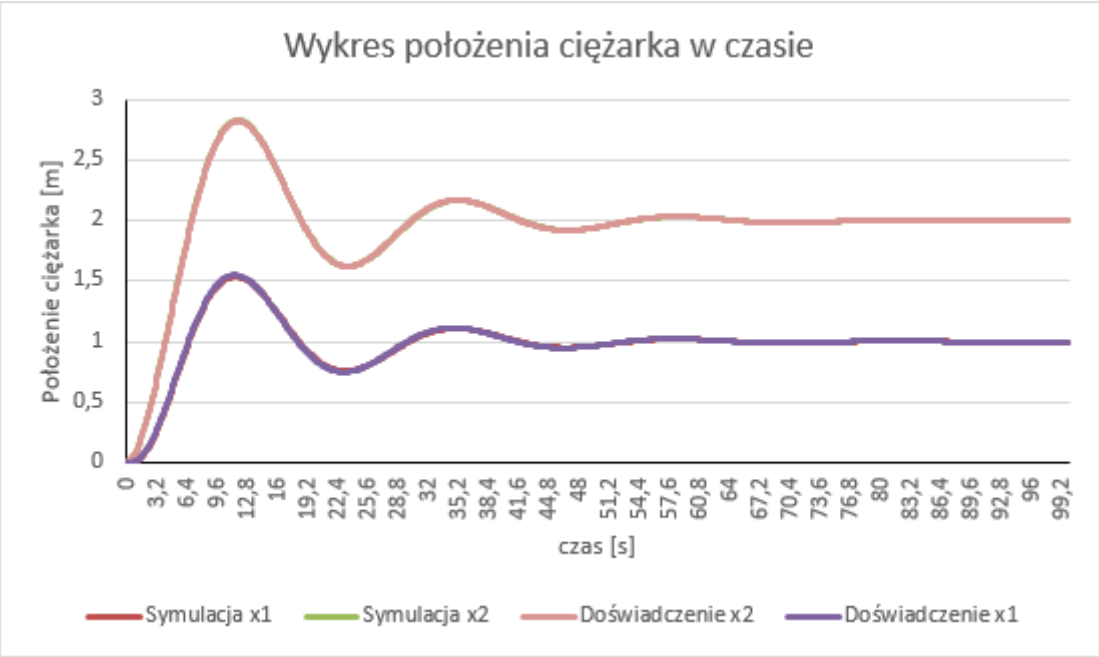
3. Funkcje pochodnych rozwiązywanych podczas symulacji

```
matrix ff6R (matrix x, matrix ud1, matrix ud2)
{
    double b_1 = x (0, 0);
    double b_2 = x (1, 0);
    double t_0 = 0.0;
    double delta_t = 0.1;
    double T = 100.0;

    matrix Y_0 (4, 1);
    Y_0 (0) = 0.0; // position1 (x1)
    Y_0 (1) = 0.0; // velocity 1
    Y_0 (2) = 0.0; // position2 (x2)
    Y_0 (3) = 0.0; // velocity 2

    matrix* Simulation = solve_ode (df6, t_0, delta_t, T, Y_0, x);

    matrix time_stamps = Simulation[0];
    matrix positions = Simulation[1];

    int numPoints = get_len (time_stamps);
    double error = 0.0;

    for (int i = 0; i < numPoints; ++i)
    {
        double s_x1 = positions (i, 0);
        double s_x2 = positions (i, 2);
        double m_x1 = ud2 (i, 0);
        double m_x2 = ud2 (i, 1);

        error += pow (s_x1 - m_x1, 2) + pow (s_x2 - m_x2, 2);
    }

    static double minError = std::numeric_limits<double>::max ();
    static matrix bestStates;
    static matrix bestTimeStamps;


    if (error < minError)
    {
        minError = error;
        bestStates = positions;
        bestTimeStamps = time_stamps;
    }

    std::ofstream resultsFile ("symulacja.csv");
    resultsFile << "time, x1, x2\n";
    for (int idx = 0; idx < get_len (bestTimeStamps); ++idx)
    {
        resultsFile << bestTimeStamps (idx, 0) << "," << bestStates (idx, 0) << "," << bestStates (idx, 2) << "\n";
    }
    resultsFile.close ();

    delete[] Simulation;

    matrix result (1, 1);
    result (0, 0) = minError;
    return result;
}
```

```
matrix df6 (double t, matrix Y, matrix ud1, matrix ud2) {
    double b_1 = ud1 (0);
    double b_2 = ud1 (1);
    double m_1 = 5.0;
    double m_2 = 5.0;
    double k_1 = 1.0;
    double k_2 = 1.0;
    double F = 1.0;

    double x_1 = Y (0);
    double x_2 = Y (2);

    double v_1 = Y (1);
    double v_2 = Y (3);

    matrix dY (4, 1);
    dY (0) = v_1;
    dY (1) = (-b_1 * v_1 - b_2 * (v_1 - v_2) - k_1 * x_1 - k_2 * (x_1 - x_2)) / m_1;
    dY (2) = v_2;
    dY (3) = (F + b_2 * (v_1 - v_2) + k_2 * (x_1 - x_2)) / m_2;

    return dY;
}
```

Wyniki wyznaczenia współczynników b1 i b2 znajdują się w arkuszu “Tabela 3” w pliku Excel. Ich wartości wynoszą b1=1,60544 b2=2.70651 i są wykorzystywane w dalszych

obliczeniach. Natomiast wyniki symulacji oraz wykres położenia ciężarków znajduje się w arkuszu “Symulacja” w pliku Excel oraz poniżej.



Na wykresie przedstawiono porównanie wyników symulacji i doświadczenia dla położenia dwóch ciężarków x1, x2 w czasie. Obie krzywe symulacyjnex1, x2 dobrze odzwierciedlają wyniki eksperymentalne, pokazując oscylacyjny ruch ciężarków z tłumieniem. Różnice między symulacją a doświadczeniem są minimalne, co wskazuje na dobrą zgodność modelu z rzeczywistością.

Wnioski

Optymalizacja wykazała skuteczność metod niedeterministycznych w rozwiązywaniu problemów testowych i rzeczywistych. Właściwy dobór parametrów, takich jak zakres mutacji czy współczynniki modelu, pozwolił na osiągnięcie wysokiej dokładności i zgodności wyników symulacji z eksperymentem.