Optymalizacja funkcji wielu zmiennych metodami bezgradientowymi	Data wykonania: 14.11.2025	Optymalizacja
Dominika Myszka Tatsiana Merzianiova	Gr.2	ITE

1. Cel ćwiczenia.
Celem ćwiczenia jest zapoznanie się z metodami bezgradientowymi poprzez ich implementację oraz wykorzystanie do rozwiązania problemu optymalizacji.

2. Wykonanie ćwiczenia, opis wyników, kody:
Funkcja celu dana jest wzorem:

$$f(x\ ,\ x\ ) = x + x - cos(2{,}5\pi x\ ) - cos(2{,}5\pi x\ ) + 2$$

Jest to funkcja multimodalna (posiadająca wiele minimów lokalnych), której minimum globalne wynosi i znajduje się w punkcie$(0{,}0)$. Punkt startowy losowano z przedziału x $\in$[-1,1].

Wyniki eksperymentów

Przeprowadzono po 100 prób optymalizacji dla różnych długości kroków startowych, startując z punktów losowych. Kryterium sukcesu (znalezienie minimum globalnego) przyjęto osiągnięcie wartości funkcji celu bliskiej 0.

Tabela 1. Skuteczność i koszt obliczeniowy metod (Dane uśrednione)

Długość kroku	Metoda	Średnia liczba wywołań funkcji	Skuteczność (Liczba minimów globalnych na 100)
1.0	Hooke-Jeeves	96.79	47%
0.1	Hooke-Jeeves	74.48	23%
0.01	Hooke-Jeeves	80.33	24%
0.5	Rosenbrock	72.64	22%
0.1	Rosenbrock	58.83	23%
0.05	Rosenbrock	52.04	23%

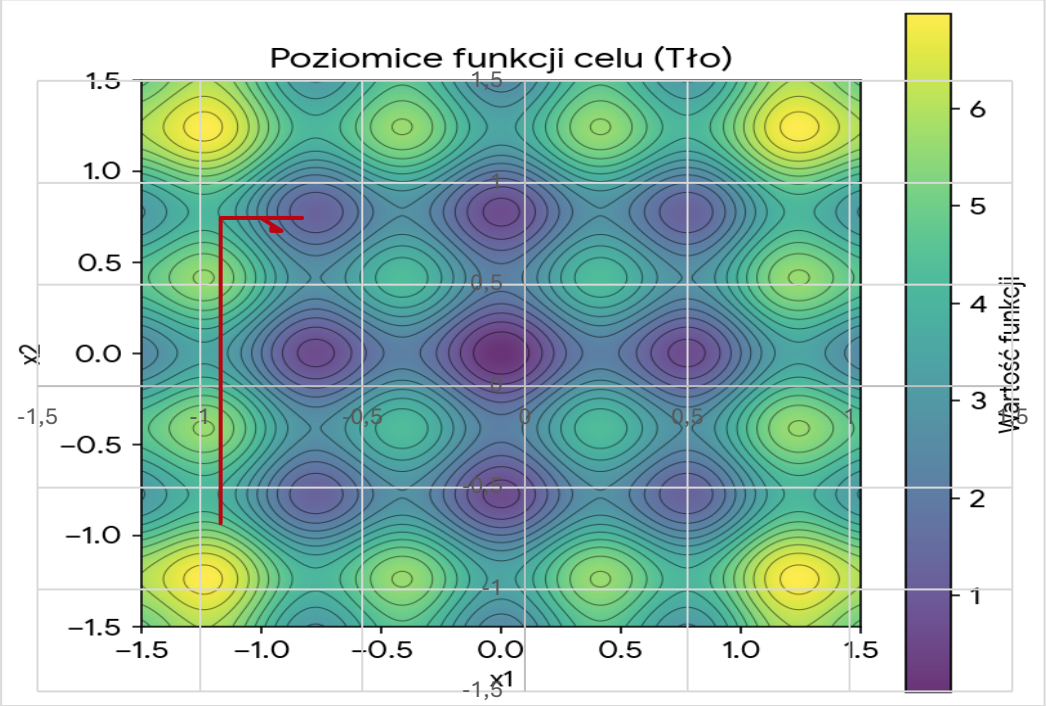
Dyskusja wyników zadania testowego

- **Wpływ długości kroku:** Dla metody Hooke'a-Jeevesa największa długość kroku (1.0) dała najwyższą skuteczność (47%). Wynika to ze specyfiki funkcji testowej –

duży krok pozwala algorytmowi „przeskoczyć” nad lokalnymi minimami (tzw. „dołkami”) i trafić w obszar przyciągania minimum globalnego. Mniejsze kroki (0.1, 0.01) powodowały częstsze utknięcie w optimach lokalnych (skuteczność ok. 23-24%).

- **Porównanie metod:** Metoda Rosenbrocka wykazała się mniejszą liczbą wywołań funkcji celu (średnio ok. 50-70 wywołań) w porównaniu do metody Hooke’a-Jeevesa (75-97 wywołań), jednak jej skuteczność w znajdowaniu minimum globalnego dla tej konkretnej funkcji była niska i stabilna na poziomie ok. 22-23% niezależnie od kroku.
- **Wniosek:** Obie metody deterministyczne mają trudności z funkcjami wielomodalnymi przy losowym punkcie startowym, chyba że parametry początkowe (wielkość kroku) są dobrane tak, aby dopasować się do "krajobrazu" funkcji.

Wykres poziomic funkcji celu dla metody Hooke-Jeeves



Problem rzeczywisty (Ramię robota)

Opis problemu

Zadanie polegało na doborze nastaw regulatora PD () dla ramienia robota, aby zminimalizować wskaźnik jakości uwzględniający uchyb regulacji kąta , prędkości oraz koszt sterowania .

Model matematyczny dynamiki: $I = \frac{1}{3} m_r l^2 + m_c l^2.$

Gdzie to moment bezwładności, a to moment sterujący:

$$M(t) = k_1 \left(\alpha_{ref} - \alpha(t) \right) + k_2 \left(\omega_{ref} - \omega(t) \right),$$

Wyniki optymalizacji

Na podstawie Tabeli 3 wyznaczono optymalne parametry sterownika. Obie metody zbiegły się do praktycznie identycznego rozwiązania.

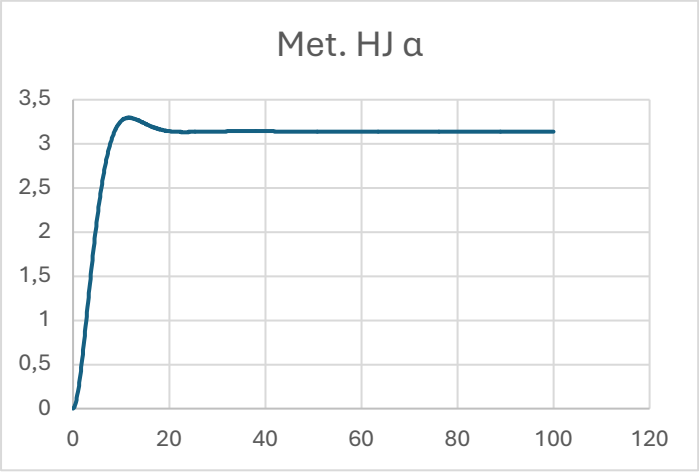
Tabela 2. Wyniki optymalizacji sterownika (Długość kroku 0.5)

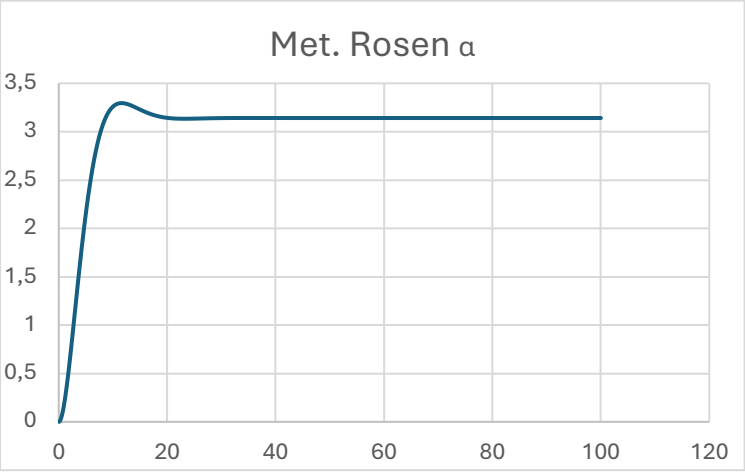
Parametr	Metoda Hooke'a-Jeevesa	Metoda Rosenbrocka
(Wzmocnienie członu proporcjonalnego)	3.005859	3.006239
(Wzmocnienie członu różniczkującego)	10.839844	10.839984
(Wartość funkcji celu)	373.656	373.656
Liczba wywołań funkcji	143	143

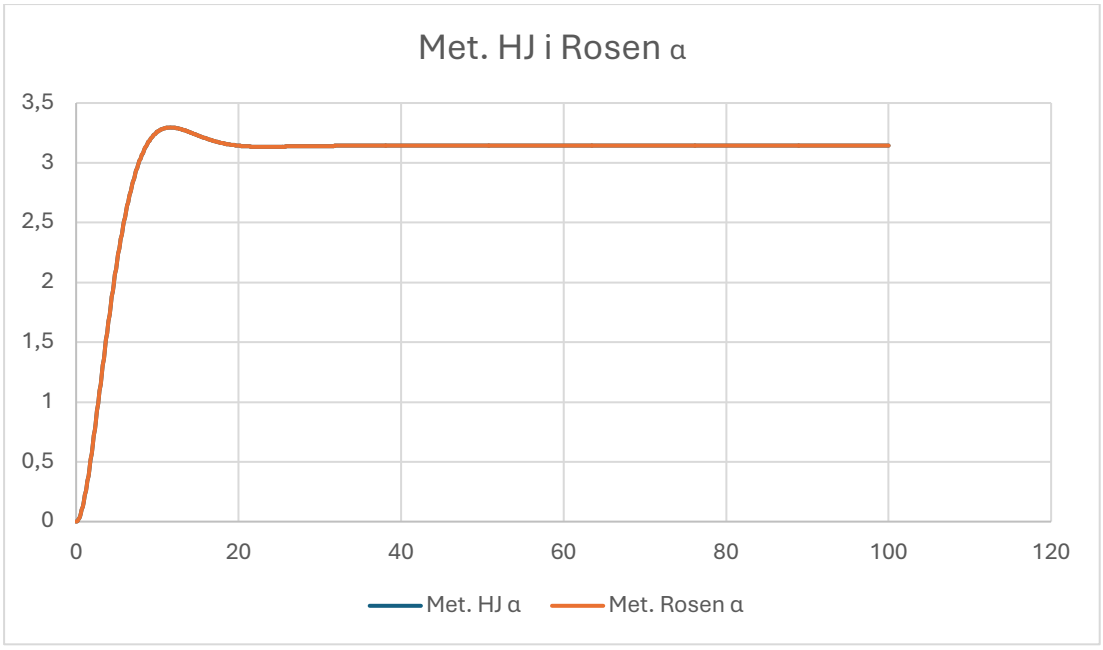
Analiza symulacji

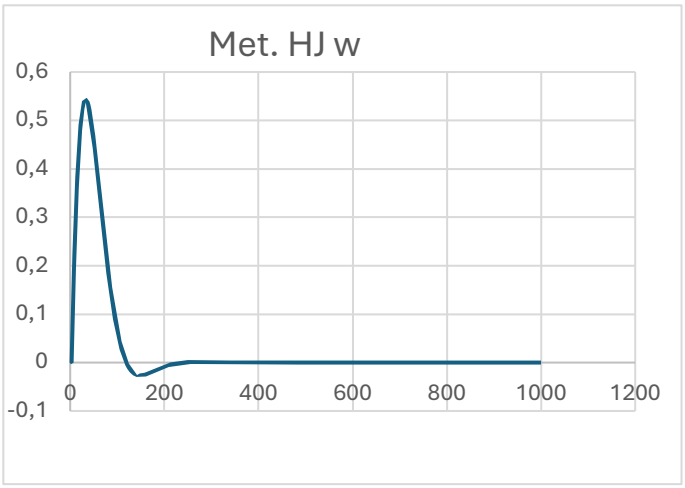
Na podstawie wyznaczonych parametrów przeprowadzono symulację ruchu ramienia. Poniżej przedstawiono analizę przebiegów czasowych:

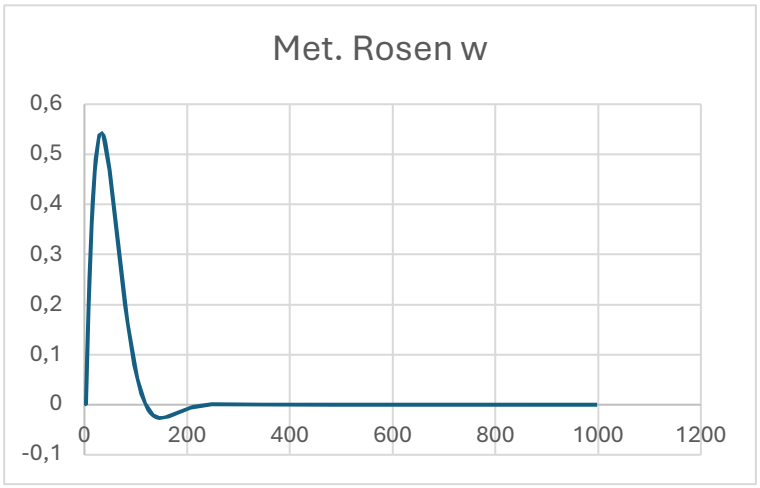
- **Położenie (alfa):** Ramię startuje z pozycji 0 rad i płynnie dąży do wartości zadanej pi (około 3.14 rad). Wartość docelowa osiągana jest w przybliżeniu w 8. sekundzie symulacji. Przebieg ma charakter aperiodyczny – nie występuje przeregulowanie (ramię nie przekracza wartości zadanej), co jest korzystne ze względów bezpieczeństwa. W stanie ustalonym (dla czasu t = 100 s) kąt alfa wynosi około 3.141593.
- **Prędkość (omega):** Prędkość narasta do około 3. sekundy, osiągając wartość maksymalną około 0.54 rad/s, a następnie łagodnie spada do zera, co potwierdza stabilne zatrzymanie ramienia w punkcie docelowym.

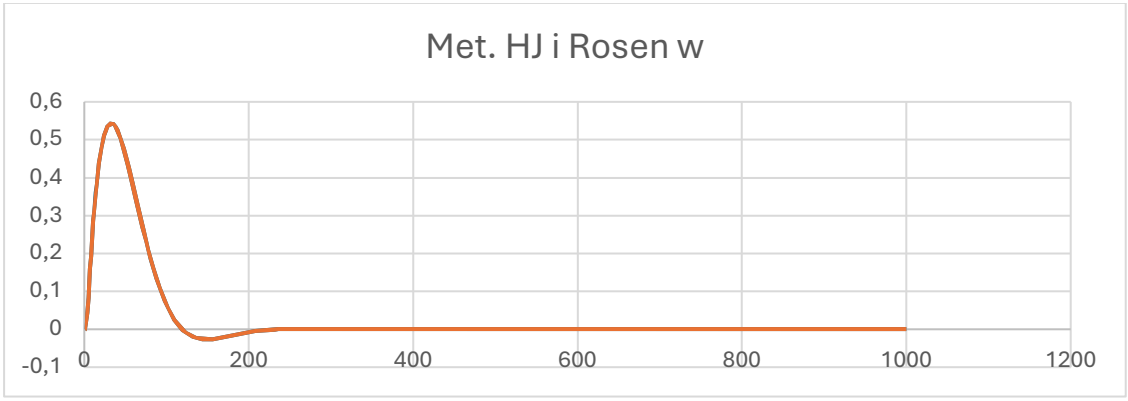





Met. HJ i Rosen α
3,5
3
2,5
2
1,5
1
0,5
0
0
20
40
60
80
100
120
Met. HJ α
Met. Rosen α


Met. HJ w
0,6
0,5
0,4
0,3
0,2
0,1
0
-0,1
0
200
400
600
800
1000
1200


Met. Rosen w
0,6
0,5
0,4
0,3
0,2
0,1
0
-0,1
0
200
400
600
800
1000
1200


Met. HJ i Rosen w
0,6
0,5
0,4
0,3
0,2
0,1
0
-0,1
0
200
400
600
800
1000
1200

Wnioski końcowe

1. **Efektywność:** Metoda Rosenbrocka okazała się bardziej ekonomiczna (mniej wywołań funkcji celu) w zadaniu testowym, zachowując podobną skuteczność dla małych kroków jak metoda Hooke'a-Jeevesa.
2. **Zbieżność w problemie inżynierskim:** W przypadku problemu sterowania (funkcja celu o gładszym przebiegu, prawdopodobnie unimodalna w badanym obszarze), obie metody dały niemal identyczne, bardzo precyzyjne wyniki.
3. **Zastosowanie:** Dla funkcji silnie nieliniowych i wielomodalnych (Zadanie 1), proste metody bezgradientowe są ryzykowne i silnie zależne od punktu startowego. Dla problemów inżynierskich o charakterystyce zbliżonej do kwadratowej (Zadanie 2), metody te sprawdzają się doskonale.

Załącznik: Implementacja kluczowych funkcji (Kod C++)

Funkcja testowa:

```
matrix ff_lab2_T(matrix x, matrix ud1, matrix ud2)
{
    double x1 = x(0);
    double x2 = x(1);
    double y_val = x1 * x1 + x2 * x2 - cos(2.5 * M_PI * x1) - cos(2.5 * M_PI * x2)
+ 2.0;
    return matrix(y_val);
}
```

Równania różniczkowe modelu ramienia

```
matrix df_lab2_R(double t, matrix Y, matrix ud1, matrix ud2)
{
    double mr = 1.0;   // masa ramienia [kg]
    double mc = 5.0;   // masa ciężarka [kg]
    double l = 2.0;    // długość ramienia [m]
    double b = 0.25;   // współczynnik tarcia [Nms]

    // Moment bezwładności [cite: 45]
    double I = (1.0 / 3.0) * mr * pow(l, 2) + mc * pow(l, 2);

    double k1 = ud1(0);
    double k2 = ud1(1);

    double alpha_ref = M_PI; // pi rad
    double omega_ref = 0.0;

    double alpha = Y(0);
    double omega = Y(1);

    // Moment sterujący M(t) [cite: 47]
    double M_t = k1 * (alpha_ref - alpha) + k2 * (omega_ref - omega);
```

```
    matrix dY(2, 1);
    dY(0) = omega;
    // Równanie dynamiki: I * alpha'' + b * alpha' = M(t) -> alpha'' = (M(t) -
b*omega)
    dY(1) = (M_t - b * omega) / I;

    return dY;
}

// Funkcja celu Q
// x(0) = k1, x(1) = k2
matrix ff_lab2_R(matrix x, matrix ud1, matrix ud2)
{
    matrix Y0(2, 1); // Warunki początkowe: alpha=0, omega=0

    // Parametry symulacji
    double t0 = 0;
    double dt = 0.1;
    double tend = 100;

    // Rozwiązanie ODE. Przekazujemy x (czyli k1, k2) jako ud1 do funkcji df_lab2_R
    matrix* Y = solve_ode(df_lab2_R, t0, dt, tend, Y0, x);

    int n = get_len(Y[0]); // liczba kroków czasowych
    double Q = 0;

    double alpha_ref = M_PI;
    double omega_ref = 0.0;

    // Całkowanie metodą prostokątów funkcji podcałkowej
    // Q = Integral( 10*(ref-a)^2 + (ref-w)^2 + M(t)^2 ) dt
    for (int i = 0; i < n; ++i)
    {
        double alpha = Y[1](i, 0);
        double omega = Y[1](i, 1);

        // Odtworzenie momentu M(t) dla danego kroku (ponieważ nie jest zwracany
przez solve_ode)
        double k1 = x(0);
        double k2 = x(1);
        double M_t = k1 * (alpha_ref - alpha) + k2 * (omega_ref - omega);

        double component = 10 * pow(alpha_ref - alpha, 2) + pow(omega_ref - omega,
2) + pow(M_t, 2);

        Q = Q + component * dt;
    }

    // Czyszczenie pamięci po solverze
    Y[0].~matrix();
```

```
    Y[1].~matrix();

    return matrix(Q);
}
```

Metoda Rosen

```
solution Rosen(matrix(*ff)(matrix, matrix, matrix), matrix x0, matrix s0, double
alpha, double beta, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        // --- LOGIKA ZAPISU DO PLIKU ---
        bool record = false;
        if (get_len(ud1) > 0 && !isnan(ud1(0, 0)) && ud1(0, 0) == 1.0) record =
true;

        ofstream Traj;
        if (record) Traj.open("trajektoria_Rosen.csv");
        // ------------------------------

        int n = get_len(x0);
        matrix D = ident_mat(n);
        matrix s = s0;
        matrix lambda(n, 1);
        matrix p(n, 1);

        solution X(x0);
        X.fit_fun(ff, ud1, ud2);
        solution XB = X;

        while (true)
        {
            // Zapis punktu
            if (record) Traj << X.x(0) << ";" << X.x(1) << endl;

            matrix s_prev = s;
            XB = X;

            for (int j = 0; j < n; ++j)
            {
                matrix dj = get_col(D, j);
                solution X_test = X;
                X_test.x = X.x + s(j) * dj;
                X_test.fit_fun(ff, ud1, ud2);

                if (X_test.y < X.y)
                {
                    X = X_test;
                    lambda(j) = lambda(j) + s(j);
                    s(j) = alpha * s(j);
```

```
            }
            else
            {
                s(j) = -beta * s(j);
                p(j) = p(j) + 1;
            }
        }

        if (solution::f_calls > Nmax)
        {
            X.flag = 0;
            break;
        }

        bool change_basis = true;
        for (int j = 0; j < n; ++j)
            if (lambda(j) == 0.0 || p(j) == 0.0)
            {
                change_basis = false;
                break;
            }

        if (change_basis)
        {
            matrix Q(n, n);
            for (int j = 0; j < n; ++j)
            {
                matrix Qj(n, 1);
                for (int k = j; k < n; ++k)
                    Qj = Qj + get_col(D, k) * lambda(k);
                Q.set_col(Qj, j);
            }

            matrix D_new(n, n);
            matrix v = get_col(Q, 0);
            D_new.set_col(v / norm(v), 0);

            for (int j = 1; j < n; ++j)
            {
                v = get_col(Q, j);
                for (int k = 0; k < j; ++k)
                {
                    matrix dk = get_col(D_new, k);
                    v = v - (trans(get_col(Q, j)) * dk) * dk;
                }
                D_new.set_col(v / norm(v), j);
            }
            D = D_new;
            lambda = matrix(n, 1);
            p = matrix(n, 1);
            s = s0;
```

```
            }

            double max_s = 0;
            for (int j = 0; j < n; ++j)
                if (abs(s_prev(j)) > max_s) max_s = abs(s_prev(j));

            if (max_s < epsilon)
            {
                X.flag = 1;
                break;
            }
        }

        if (record) Traj.close();
        return X;
    }
    catch (string ex_info)
    {
        throw ("solution Rosen(...):\n" + ex_info);
    }
}
```

Metoda HJ

```
solution HJ(matrix(*ff)(matrix, matrix, matrix), matrix x0, double s, double alpha,
double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        // --- LOGIKA ZAPISU DO PLIKU ---
        bool record = false;
        // Jeśli ud1 nie jest puste i ma wartość 1, włączamy nagrywanie
        if (get_len(ud1) > 0 && !isnan(ud1(0, 0)) && ud1(0, 0) == 1.0) record =
true;

        ofstream Traj;
        if (record) Traj.open("trajektoria_HJ.csv");
        // -------------------------------

        solution X(x0), XB(x0);
        X.fit_fun(ff, ud1, ud2);
        XB.fit_fun(ff, ud1, ud2);

        while (true)
        {
            // Zapis punktu bieżącego
            if (record) Traj << XB.x(0) << ";" << XB.x(1) << endl;

            XB = X;
            X = HJ_trial(ff, XB, s, ud1, ud2);
```

```
            if (X.y < XB.y)
            {
                while (true)
                {
                    // Zapis punktu w pętli pattern move
                    if (record) Traj << XB.x(0) << ";" << XB.x(1) << endl;

                    solution X_prev = XB;
                    XB = X;
                    X.x = 2.0 * XB.x - X_prev.x;
                    X.fit_fun(ff, ud1, ud2);
                    X = HJ_trial(ff, X, s, ud1, ud2);

                    if (solution::f_calls > Nmax)
                    {
                        X.flag = 0;
                        if (record) Traj.close();
                        return X;
                    }
                    if (X.y >= XB.y) break;
                }
                X = XB;
            }
            else
            {
                s = alpha * s;
            }

            if (solution::f_calls > Nmax)
            {
                X.flag = 0;
                break;
            }
            if (s < epsilon)
            {
                X.flag = 1;
                break;
            }
        }

        if (record) Traj.close();
        return X;
    }
    catch (string ex_info)
    {
        throw ("solution HJ(...):\n" + ex_info);
    }
}
```

Implementacja w main

```cpp
void lab2()
{
    // ================================================================================
    // CZĘŚĆ 1: Zadanie 5a - Statystyka (Tabela 1 i 2)
    // ================================================================================

    ofstream Sout("wyniki_lab2_partA.csv");
    if (!Sout.is_open()) { cerr << "Blad zapisu partA" << endl; return; }

    cout << "--- Zadanie 5a: Funkcja Testowa (Statystyka) ---" << endl;
    Sout << fixed << setprecision(6);
    cout << fixed << setprecision(6);

    int N_runs = 100;
    int Nmax = 20000;
    double epsilon = 1e-3;
    matrix lb(2, 1, -1.0);
    matrix ub(2, 1, 1.0);

    // Generowanie 100 stałych punktów startowych dla rzetelnego porównania
    std::vector<matrix> start_points;
    for (int i = 0; i < N_runs; ++i)
    {
        matrix x0 = rand_mat(2, 1);
        x0(0) = (ub(0) - lb(0)) * x0(0) + lb(0); // Skalowanie do [-1, 1]
        x0(1) = (ub(1) - lb(1)) * x0(1) + lb(1); // Skalowanie do [-1, 1]
        start_points.push_back(x0);
    }

    double s_hj_all[] = { 1.0, 0.1, 0.01 };
    double alpha_hj = 0.5;

    matrix s0_rosen_all[] = { matrix(2, 1, 0.5), matrix(2, 1, 0.1), matrix(2, 1,
0.05) };
    double alpha_rosen = 2.0;
    double beta_rosen = 0.5;

    // --- Metoda Hooke'a-Jeevesa (Statystyka) ---
    for (double s_hj : s_hj_all)
    {
        Sout << "\nMetoda Hooke'a-Jeevesa, s = " << s_hj << "\n";
        Sout << "x1_start;x2_start;x1_opt;x2_opt;y_opt;f_calls;flag\n";
        int global_found = 0;
        for (const auto& x0 : start_points)
        {
            solution::clear_calls();
            solution opt = HJ(ff_lab2_T, x0, s_hj, alpha_hj, epsilon, Nmax);

            Sout << x0(0) << ";" << x0(1) << ";" << opt.x(0) << ";" << opt.x(1) <<
";"
                << opt.y << ";" << solution::f_calls << ";" << opt.flag << "\n";
```

```
            // Kryterium sukcesu: blisko (0,0) i wartość bliska 0
            if (opt.flag == 1 && norm(opt.x) < 0.1 && opt.y < 0.01) global_found++;
        }
        cout << "HJ (s=" << s_hj << "): Globalne = " << global_found << "/" <<
N_runs << endl;
    }

    // --- Metoda Rosenbrocka (Statystyka) ---
    for (const auto& s0_rosen : s0_rosen_all)
    {
        double s_val = s0_rosen(0);
        Sout << "\nMetoda Rosenbrocka, s = " << s_val << "\n";
        Sout << "x1_start;x2_start;x1_opt;x2_opt;y_opt;f_calls;flag\n";
        int global_found = 0;
        for (const auto& x0 : start_points)
        {
            solution::clear_calls();
            solution opt = Rosen(ff_lab2_T, x0, s0_rosen, alpha_rosen, beta_rosen,
epsilon, Nmax);

            Sout << x0(0) << ";" << x0(1) << ";" << opt.x(0) << ";" << opt.x(1) <<
";"
                << opt.y << ";" << solution::f_calls << ";" << opt.flag << "\n";

            if (opt.flag == 1 && norm(opt.x) < 0.1 && opt.y < 0.01) global_found++;
        }
        cout << "Rosenbrock (s=" << s_val << "): Globalne = " << global_found <<
"/" << N_runs << endl;
    }
    Sout.close();


    // ========================================================================
    // CZĘŚĆ 2: Zadanie 5b - Problem Rzeczywisty (Robot)
    // ========================================================================

    ofstream SoutR("wyniki_lab2_partB.csv");
    if (!SoutR.is_open()) { cerr << "Blad zapisu partB" << endl; return; }

    cout << "\n--- Zadanie 5b: Problem Rzeczywisty ---" << endl;
    SoutR << fixed << setprecision(6);

    // Punkt startowy k1=10, k2=10 (środek przedziału [0,20])
    matrix x0_robot(2, 1);
    x0_robot(0) = 10.0;
    x0_robot(1) = 10.0;

    double s_robot = 0.5; // Krok startowy dla robota
    matrix s0_robot(2, 1, s_robot);
```

```
    // --- Optymalizacja HJ ---
    solution::clear_calls();
    solution opt_HJ = HJ(ff_lab2_R, x0_robot, s_robot, alpha_hj, epsilon, Nmax);
    int hj_f_calls = solution::f_calls; // Zapamiętujemy liczbę wywołań
    cout << "HJ zakonczone. y = " << opt_HJ.y << endl;

    // --- Optymalizacja Rosenbrocka ---
    solution::clear_calls();
    solution opt_Ros = Rosen(ff_lab2_R, x0_robot, s0_robot, alpha_rosen,
beta_rosen, epsilon, Nmax);
    cout << "Rosenbrock zakonczony. y = " << opt_Ros.y << endl;

    // Zapis do Tabeli 3
    SoutR << "Metoda;k1_start;k2_start;k1_opt;k2_opt;Q_min;f_calls\n";
    SoutR << "HJ;" << x0_robot(0) << ";" << x0_robot(1) << ";"
        << opt_HJ.x(0) << ";" << opt_HJ.x(1) << ";" << opt_HJ.y << ";" <<
hj_f_calls << "\n";
    SoutR << "Rosenbrock;" << x0_robot(0) << ";" << x0_robot(1) << ";"
        << opt_Ros.x(0) << ";" << opt_Ros.x(1) << ";" << opt_Ros.y << ";" <<
solution::f_calls << "\n";
    SoutR.close();

    // --- Symulacja dla najlepszego wyniku ---
    // Wybieramy metodę, która dała mniejszy błąd Q
    solution best_opt = (opt_HJ.y < opt_Ros.y) ? opt_HJ : opt_Ros;
    cout << "Generowanie symulacji dla lepszego wyniku (Metoda: "
        << ((opt_HJ.y < opt_Ros.y) ? "HJ" : "Rosenbrock") << ")..." << endl;

    matrix Y0(2, 1); // Warunki początkowe
    matrix* Y_sim = solve_ode(df_lab2_R, 0, 0.1, 100, Y0, best_opt.x);

    ofstream SimOut("symulacja_lab2.csv");
    if (SimOut.is_open())
    {
        SimOut << fixed << setprecision(6);
        // Zapis: czas; pozycja; prędkość
        int n_steps = get_len(Y_sim[0]);
        for (int i = 0; i < n_steps; ++i)
        {
            SimOut << Y_sim[0](i, 0) << ";" << Y_sim[1](i, 0) << ";" << Y_sim[1](i,
1) << "\n";
        }
        SimOut.close();
        cout << "Dane symulacji zapisane w 'symulacja_lab2.csv'. (Wklej do arkusza
Symulacja)" << endl;
    }
    Y_sim[0].~matrix(); Y_sim[1].~matrix();


    // ========================================================================
    // CZĘŚĆ 3: GENEROWANIE TRAJEKTORII (Dla arkusza "Wykres" - wykres poziomic)
```

```
    // ========================================================================
    cout << "\n--- Generowanie trajektorii dla wybranego przypadku (Zad 5a Wykres)
---" << endl;

    // Wybieramy pierwszy punkt startowy z naszej listy (dla powtarzalności)
    matrix x0_traj = start_points[0];

    // Ustawiamy flagę zapisu (ud1 = 1.0) - to uruchomi kod zapisu w opt_alg.cpp
    matrix flag_record(1, 1, 1.0);

    // 1. Trajektoria HJ (najczęściej wymagana do wykresu poziomic)
    solution::clear_calls();
    // Używamy przykładowego kroku 0.5
    HJ(ff_lab2_T, x0_traj, 0.5, alpha_hj, epsilon, Nmax, flag_record);
    cout << "Utworzono plik 'trajektoria_HJ.csv' z punktami posrednimi." << endl;
    cout << "Skopiuj zawartosc tego pliku na wykres poziomic w Excelu." << endl;
}
```