Marcin Brzózka gr. 1

Nr. albumu: 408932

Laboratorium 2, temat:

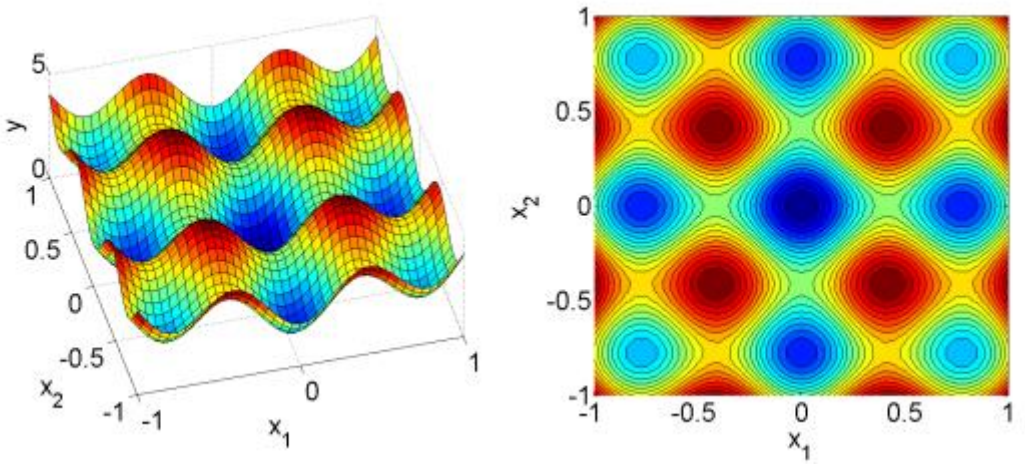
Optymalizacja funkcji wielu zmiennych metodami bezgradientowymi

Cel ćwiczenia:

Celem ćwiczenia jest zapoznanie się z metodami bezgradientowymi poprzez ich implementację oraz wykorzystanie do rozwiązania jednowymiarowego problemu optymalizacji. Do wykonania zadania wykorzystujemy metody Hooke'a-Jeevesa oraz Rosenbrocka.

Wykonanie ćwiczenia:

Pierwszym krokiem wykonywania ćwiczenia było wypełnienie braków w udostępnionym nam kodzie przy pomocy pseudokodu zamieszczonego w instrukcji do ćwiczenia. W tym kroku należało uzupełnić wzór funkcji oraz brakujące elementy metod wymienionych powyżej funkcji. Następnym etapem wykonywania ćwiczenia było uzupełnienie danych uzyskanych z 300 wywołań gotowego programu. Dane te były ułożone po 100 dla trzech różnych kroków. W pierwszej kolumnie pierwszej tabeli uzupełnialiśmy wybrane przez nas długości kroku. W moim przypadku były to: „0,9", „1,7", „2,4". Następnie uzupełnialiśmy pozostałe dane związane z metodami Hooke'a-Jeevesa oraz Rosenbrocka. Minimum globalne ustaliłem przy pomocy dostępnego wykresu funkcji.



Możemy zauważyć że minimum którego szukamy znajduję się w x_1=0, x_2=0, y=0

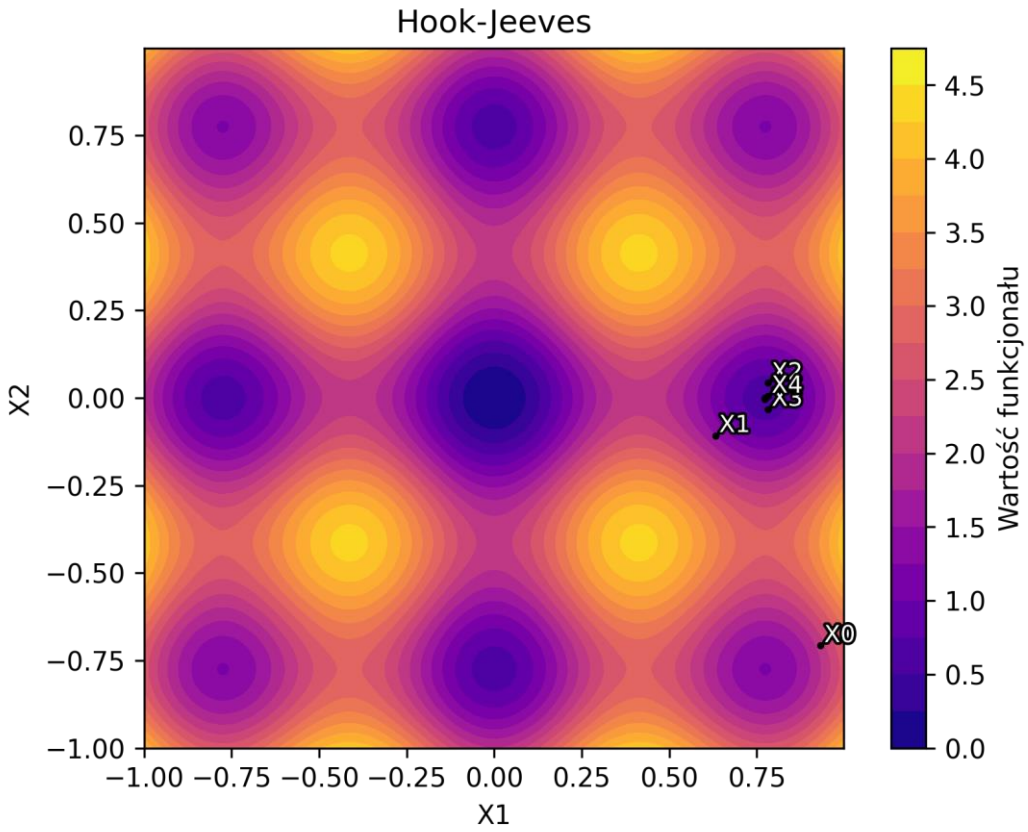
Zebranie tych wszystkich wyników i wyłonienie spośród nich minimum globalnych pozwoliło nam uzupełnić tabelę drugą w której znajdowały się średnie wartości uzupełnionych danych.

Długość kroku	Metoda Hooke'a-Jeevesa					Metoda Rosenbrocka				
	x_1^*	x_2^*	y^*	Liczba wywołań funkcji celu	Liczba minimów globalnych	x_1^*	x_2^*	y^*	Liczba wywołań funkcji celu	Liczba minimów globalnych
0,9	-6,43E-05	-1,66E-04	1,66E-05	97,14545455	55	4,91E-05	-2,50E-05	2,15E-05	100,4117647	51
1,7	-2,41E-05	-6,44E-05	1,60E-05	122,3088235	68	7,69E-06	-4,38E-05	2,65E-05	109,0266667	75
2,4	4,69E-05	4,40E-06	9,55E-06	101,5909091	22	4,69E-06	-1,69E-05	9,78E-06	122,0909091	33

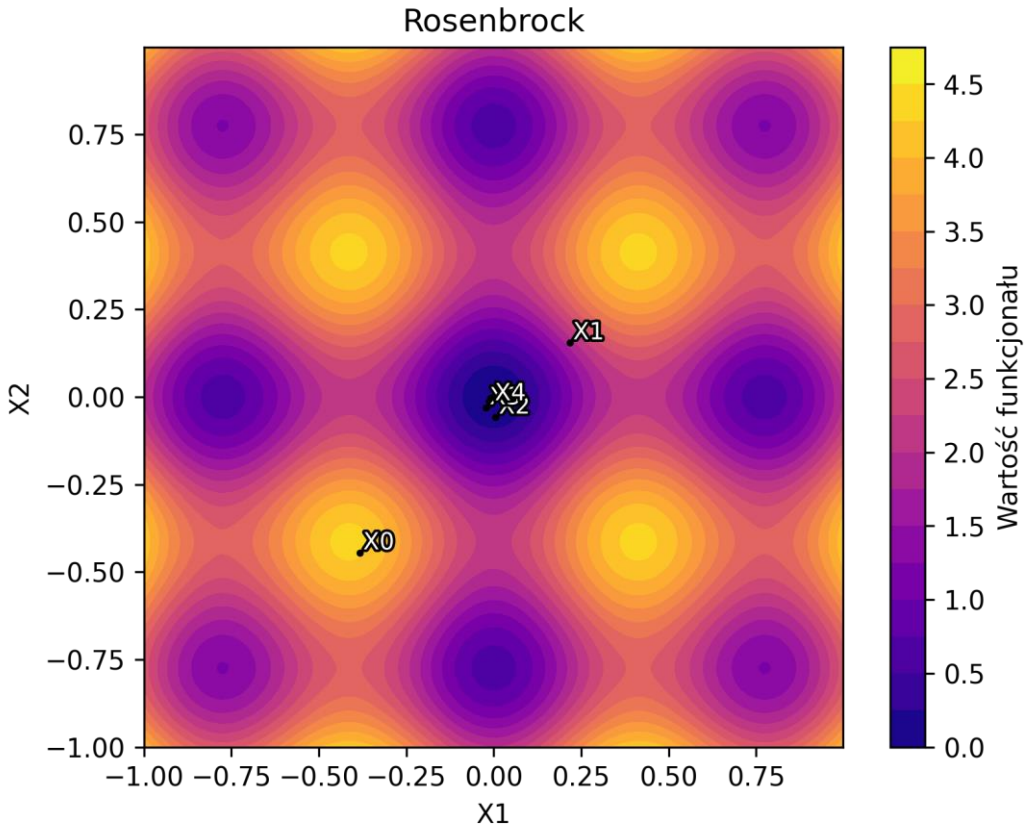
Z tabeli tej możemy wywnioskować że wraz z zmniejszeniem się długości kroku liczba wywołań funkcji w metodzie Rosenbrocka zmniejsza się, natomiast w metodzie Hooke'a-Jeevesa różni się ona z każdą długością kroku.

Wykres poziomic:

Hooke'a-Jeeves:



Rosenbrock:

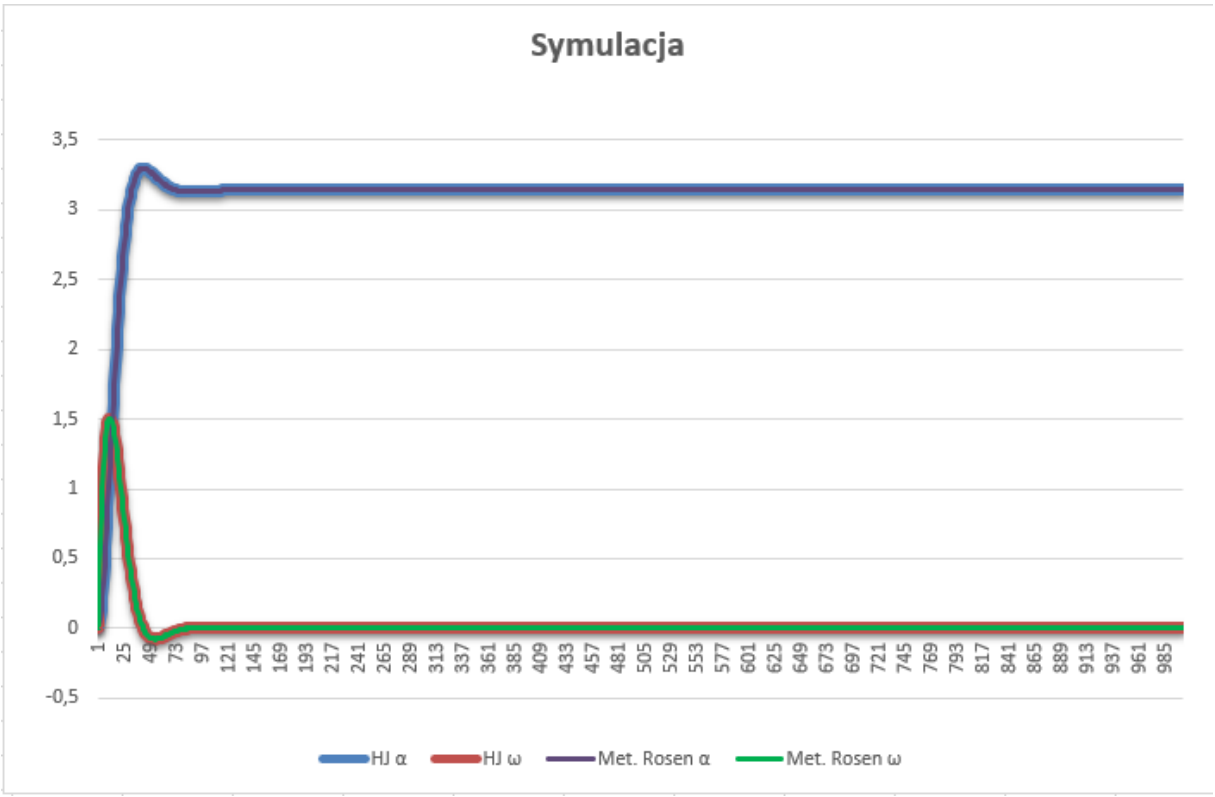


Następnym krokiem było wykorzystanie gotowego programu do rozwiązania problemu rzeczywistego przedstawionego w instrukcji do zadania. Do wykonania tego etapu potrzebowaliśmy określić pole położenia robota oraz prędkość jego ramienia. Wymienione wyżej dane umieszczaliśmy w tabeli trzeciej:

Długość kroku	Metoda Hooke'a-Jeevesa				Metoda Rosenbrocka			
	k_1*	k_2*	Q*	Liczba wywołań funkcji celu	k_1*	k_2*	Q*	Liczba wywołań funkcji celu
2	2,77007	3,1845	140,16	98	2,77053	3,18456	140,16	193

W moim przypadku wartość Q* wyniosła 140,16 natomiast wartości k_1 oraz k_2 minimalnie różnią się w zależności od wybranej metody.

Ostatnim krokiem ćwiczenia było zebranie danych i przeprowadzenie symulacji. Wykres uzyskany przy pomocy danych prezentuję się następująco:



Przedstawia on położenie kątowe ramienia i jego prędkość kątową w okresie czasu 100s i krokiem czasowym 0,1s. Daję nam to po 1000 wyników na każdą z metod. Jak możemy zauważyć dla obu metod wykres wskazuję praktycznie identyczne wyniki. Jest to spowodowane tym że metody te szukają wspólnego minimum ze skutecznością będącą na bardzo podobnym poziomie.

Kod wykorzystany do wykonania ćwiczenia:

```cpp
void lab2()
{
    //Funkcja testowa
    double s = 0.5, alphaHJ = 0.5, alphaR = 2, beta = 0.5, epsilon = 1e-3;
    int Nmax = 1000;
    solution opt;
    matrix x0, s0;
    /*s0 = matrix(2, 1, s);
    x0 = 2 * rand_mat(2, 1) - 1;
    cout << x0 << endl << endl;
    saveResultToFile(x0, "wyniki.txt");
    opt = HJ(ff2T, x0, s, alphaHJ, epsilon, Nmax);
    cout << opt << endl << endl;
    saveResultToFile(opt, "wyniki.txt");
    solution::clear_calls();
    opt = Rosen(ff2T, x0, s0, alphaR, beta, epsilon, Nmax);
    cout << opt << endl << endl;
    saveResultToFile(opt, "wyniki.txt");
    ofstream file("wyniki.txt", ios::app);
    if (file.is_open()) {
        file << SEPARATOR << "\n";
        file.close();
    }
    else {
        cerr << "Błąd otwarcia pliku do zapisu.\n";
    }
    solution::clear_calls();*/

    //Ramie robota
    s = 2;

    x0 = 10 * rand_mat(2, 1);
    cout << x0 << endl << endl;
    opt = HJ(ff2R, x0, s, alphaHJ, epsilon, Nmax);
    int n = get_len(symulation[0]);
    for (int i = 0; i < n; ++i)
        cout << symulation[1](i, 0) << "," << symulation[1](i, 1) << endl;
    cout << opt << endl << endl;
    solution::clear_calls();
    s0 = matrix(2, 1, s);
    opt = Rosen(ff2R, x0, s0, alphaR, beta, epsilon, Nmax);
    n = get_len(symulation[0]);
    for (int i = 0; i < n; ++i)
        cout << symulation[1](i, 0) << "," << symulation[1](i, 1) << endl;
    cout << opt << endl << endl;
    solution::clear_calls();
}
```

```cpp
#pragma once

#include"ode_solver.h"

matrix ff2T(matrix, matrix = NAN, matrix = NAN);
matrix ff2R(matrix, matrix = NAN, matrix = NAN);
matrix df2(double, matrix, matrix = NAN, matrix = NAN);

extern matrix* symulation;
```

```cpp
#include"user_funs.h"
#define _USE_MATH_DEFINES
#include <math.h>

using namespace std;
matrix* symulation;

matrix ff2T(matrix x, matrix ud1, matrix ud2)
{
    matrix y;
    double& x1 = x(0);
    double& x2 = x(1);
    y = pow(x1, 2) + pow(x2, 2) - cos(2.5 * M_PI * x1) - cos(2.5 * M_PI * x2) + 2;
    return y;
}

matrix ff2R(matrix x, matrix ud1, matrix ud2)
{
    matrix y;
    matrix Y0(2, 1), Y_ref(2, new double[2]{ 3.14,0 });
    matrix* Y = solve_ode(df2, 0, 0.1, 100, Y0, Y_ref, x);
    int n = get_len(Y[0]);
    y = 0;
    for (int i = 0; i < n; ++i)
    {
        y = y + 10 * pow(Y_ref(0) - Y[1](i, 0), 2) +
            pow(Y_ref(1) - Y[1](i, 1), 2) +
            pow(x(0) * (Y_ref(0) - Y[1](i, 0)) + x(1) * (Y_ref(1) - Y[1](i, 1)), 2);
    }
    y = y * 0.1;
    symulation = Y;
    return y;
}

matrix df2(double t, matrix Y, matrix ud1, matrix ud2)
{
    double mr = 1, mc = 10, l = 0.5, b = 0.5;
    double I = mr * l * l / 3 + mc * l * l;
    matrix dY(2, 1);
    dY(0) = Y(1);
    dY(1) = (ud2(0) * (ud1(0) - Y(0)) + ud2(1) * (ud1(1) - Y(1)) - b * Y(1)) / I;
    return dY;
}
```

```
solution HJ(matrix(*ff)(matrix, matrix, matrix), matrix x0, double s, double alpha, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        //int i = 1;
        solution Xopt;
        Xopt.ud = trans(x0);
        solution XB(x0), XB_old, X;
        XB.fit_fun(ff, ud1, ud2);
        while (true)
        {
            //cout << "i: " << i << endl;
            //i++;
            //cout << "xb: " << XB.x << endl;
            X = HJ_trial(ff, XB, s, ud1, ud2);
            if (X.y < XB.y)
            {
                while (true)
                {
                    XB_old = XB;
                    XB = X;
                    Xopt.ud.add_row(trans(XB.x));
                    X.x = 2 * XB.x - XB_old.x;
                    X.fit_fun(ff, ud1, ud2);
                    X = HJ_trial(ff, X, s, ud1, ud2);
                    if (X.y >= XB.y)
                        break;
                    if (solution::f_calls > Nmax)
                    {
                        Xopt = XB;
                        Xopt.flag = 0;
                        return Xopt;
                    }
                }
            }
            else
                s *= alpha;
            if (s < epsilon)
            {
                Xopt = XB;
                Xopt.flag = 1;
                break;
            }
            if (solution::f_calls > Nmax)
            {
                Xopt = XB;
                Xopt.flag = 0;
                break;
            }
        }
        return Xopt;
    }
    catch (string ex_info)
    {
        throw ("solution HJ(...):\n" + ex_info);
    }
}
```

```
solution HJ_trial(matrix(*ff)(matrix, matrix, matrix), solution XB, double s, matrix ud1, matrix ud2)
{
    try
    {
        int n = get_dim(XB);
        matrix D = ident_mat(n);
        solution X;
        for (int i = 0; i < n; ++i)
        {
            X.x = XB.x + s * D[i];
            X.fit_fun(ff, ud1, ud2);
            if (X.y < XB.y)
                XB = X;
            else
            {
                X.x = XB.x - s * D[i];
                X.fit_fun(ff, ud1, ud2);
                if (X.y < XB.y)
                    XB = X;
            }
        }
        return XB;
    }
    catch (string ex_info)
    {
        throw ("solution HJ_trial(...):\n" + ex_info);
    }
}
```

```
solution Rosen(matrix(*ff)(matrix, matrix, matrix), matrix x0, matrix s0, double alpha, double beta, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        //int i = 1;
        solution Xopt;
        Xopt.ud = trans(x0);
        solution XB(x0), X;
        int n = get_dim(XB);
        matrix l(n, 1), p(n, 1), s(s0), D = ident_mat(n);
        XB.fit_fun(ff, ud1, ud2);
        while (true)
        {
            //cout << "i: " << i << endl;
            //i++;
            //cout << "xb: " << XB.x << endl;*/
            for (int i = 0; i < n; ++i)
            {
                X.x = XB.x + s(i) * D[i];
                X.fit_fun(ff, ud1, ud2);
                if (XB.y > X.y)
                {
                    XB = X;
                    l(i) += s(i);
                    s(i) *= alpha;
                }
                else
                {
                    s(i) *= -beta;
                    ++p(i);
                }
            }

            Xopt.ud.add_row(trans(XB.x));

            bool change = true;
            for (int i = 0; i < n; ++i)
                if (p(i) == 0 || l(i) == 0)
                {
                    change = false;
                    break;
                }
            if (change)
            {
                matrix Q(n, n), v(n, 1);
                for (int i = 0; i < n; ++i)
                    for (int j = 0; j <= i; ++j)
                        Q(i, j) = l(i);
                Q = D * Q;
                v = Q[0] / norm(Q[0]);
                D.set_col(v, 0);
                for (int i = 1; i < n; ++i)
                {
                    matrix temp(n, 1);
                    for (int j = 0; j < i; ++j)
                        temp = temp + trans(Q[i]) * D[j] * D[j];
                    v = (Q[i] - temp) / norm(Q[i] - temp);
                    D.set_col(v, i);
```

```
                D.set_col(v, i);
            }
            s = s0;
            l = matrix(n, 1);
            p = matrix(n, 1);
        }
        double max_s = abs(s(0));
        for (int i = 1; i < n; ++i)
            if (max_s < abs(s(i)))
                max_s = abs(s(i));
        if (max_s < epsilon)
        {
            Xopt = XB;
            Xopt.flag = 1;
            break;
        }
        if (solution::f_calls > Nmax)
        {
            Xopt = XB;
            Xopt.flag = 0;
            break;
        }
    }
    return Xopt;
}
catch (string ex_info)
{
    throw ("solution Rosen(...):\n" + ex_info);
}
}
```

Wnioski:

Analizując wyżej przedstawione wyniki możemy dojść do wniosku że metody wykorzystane do wykonania zadania przynoszą rzeczywiste rezultaty poprzez rozwiązanie problemu przedstawionego w instrukcji do ćwiczenia. Metody metody Hooke'a-Jeevesa oraz Rosenbrocka różnią się od siebie głównie prędkością wykonywania obliczeń wraz ze zmianą długości kroku. Druga z nich jest bardziej zaawansowana i wraz z zmniejszeniem się długości kroku czas realizacji obliczeń również się zmniejsza. Pierwsza z wymienionych jest prostsza a liczba wywołań funkcji celu nieznacznie różni się w zależności od długości kroku. Ćwiczenie to pokazuję że dobór odpowiedniej długości kroku ma bardzo istotny wpływ na skuteczność i szybkość wykonywanych obliczeń. Właściwy wybór długości kroku i wykorzystywanej metody może mieć kluczowy wpływ na efekty naszej optymalizacji.