Marcin Brzózka gr. 1

Nr. albumu: 408932

Laboratorium 2, temat:

Optymalizacja z ograniczeniami funkcji wielu zmiennych metodami bezgradientowymi.

Cel ćwiczenia:

Celem ćwiczenia jest zapoznanie się z metodami bezgradientowymi uwzględniając ograniczenia poprzez ich implementację oraz wykorzystanie do rozwiązania jednowymiarowego problemu optymalizacji. Do wykonania zadania wykorzystujemy metody sympleks Neldera-Meada oraz metody funkcji kary.

Wykonanie ćwiczenia:

Wykonywanie ćwiczenia rozpocząłem od uzupełnienia braków w udostępnionym razem z plikami kodem. Kod należało uzupełnić zgodnie z pseudokodem dostępnym w instrukcji do ćwiczenia. Część testowa ćwiczenia polegała na wykonaniu 100 wywołań funkcji dla trzech różnych wartości parametru a które podane były w instrukcji do ćwiczenia. Funkcja celu w przypadku tego zadania miała postać: \

$$f(x_1, x_2) = \frac{sin\left(\pi\sqrt{\left(\frac{x_1}{\pi}\right)^2 + \left(\frac{x_2}{\pi}\right)^2}\right)}{\pi\sqrt{\left(\frac{x_1}{\pi}\right)^2 + \left(\frac{x_2}{\pi}\right)^2}}$$

Część rzeczywista polegała natomiast na znalezieniu wartości przemieszczenia oraz prędkości piłki w celu symulacji trajektorii lotu.

Po wywołaniu zebraniu wyników z 300 wywołań naszego programu i uzupełnieniu nimi tabeli pierwszej mogłem wyliczyć średnie wartości wszystkich kolumn w zależności od parametru a i uzupełnić nimi tabelę drugą.

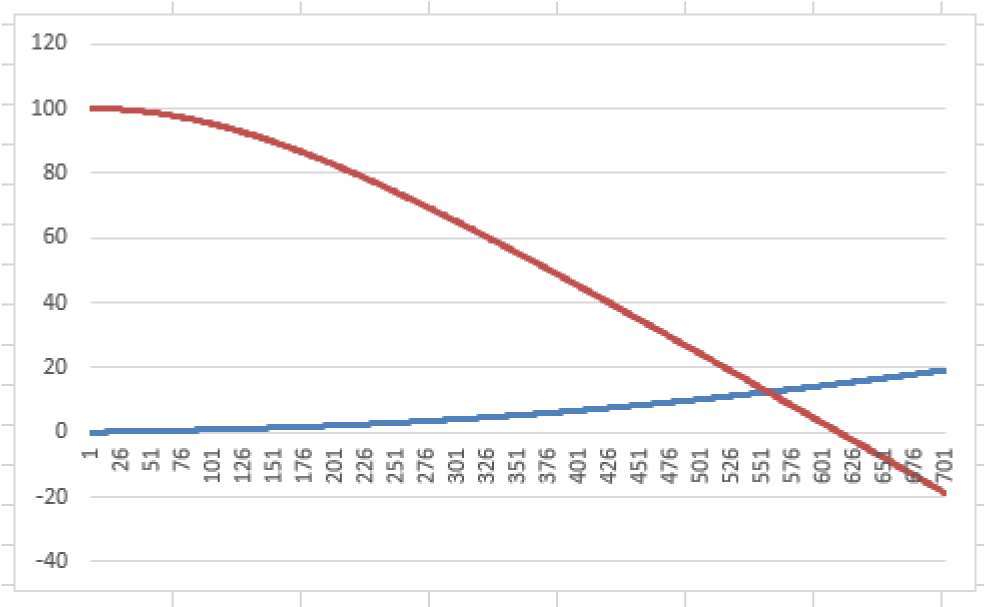
Parametr a	Zewnętrzna funkcja kary					Wewnętrzna funkcja kary				
	x_1*	x_2*	r*	y*	Liczba wywołań funkcji celu	x_1*	x_2*	r*	y*	Liczba wywołań funkcji celu
4	2,7978878	2,7431564	4,0001068	-0,18920314	361,7	2,75587369	2,701142	3,9435014	-149397,3763	170,84
4,4934	3,1076917	3,089099	4,4933539	-0,217234	74,62	3,08107619	3,06248369	4,4588171	-7,74E+04	210,48
5	3,1332131	3,01700242	4,4969968	-0,21710225	71,82	3,46710669	3,35089631	4,9585706	-73456,74146	280,8

Analizując otrzymane wyniki możemy dojść do wniosków że liczba wywołań funkcji w konkretnej funkcji kary zależy od parametru a. Dla zewnętrznej funkcji kary liczba ta jest większa dla mniejszej wartości a natomiast dla wewnętrznej funkcji kary liczba wywołań funkcji celu wzrasta wraz z wzrostem wartości parametru a. Możemy również zauważyć dużą różnicę w wartościach wyników y. Ma to związek z wynikami funkcji kary. W przypadku wewnętrznej funkcji kary powodowane jest to tym że uwzględnia ona odwrotność funkcji ograniczających. Są one często bliskie zeru co powoduje otrzymywanie niskich wartości $S(x_1, x_2)$ a to wpływa bezpośrednio na wartości „y".

Następnym krokiem było wykorzystanie przygotowanego programu do rozwiązania problemu rzeczywistego. Z racji że punkt startowy w każdym przypadku jest generowany losowo wyniki będą różniły się pomiędzy konkretnymi wywołaniami.

$v_{0x}^{(0)}$	$\omega^{(0)}$	v_{ox}^*	ω^*	x_{end}^*	Liczba wywołań funkcji celu
-2,10002	4,96671	0,187086	7,25382	18,1888	487

Po rozwiązaniu problemu rzeczywistego uzupełniłem tabelkę odpowiedzialną za symulację i wykonałem wykres toru lotu piłki:



Analizując wykres możemy zaobserwować że wartość y spada natomiast wartość x wzrasta. Wynika to z grawitacji oraz opisanego w instrukcji do zadania efektu Magnusa. Piłka poczas spadania zmienia swoje położenie względem osi x.

Kod wykorzystany do wykonania ćwiczenia:

```cpp
void lab3()
{
    //Funkcja testowa
        matrix x0, a(5);
        double c_ex = 1, c_in = 10, dc_ex = 2, dc_in = 0.5, epsilon = 1e-4;
        int Nmax = 10000;
        solution opt;
        do
            x0 = 5 * rand_mat(2, 1) + 1;
        while (norm(x0) > a);
        cout << x0 << endl << endl;
        saveResultToFile(x0, "wyniki.txt");
        opt = pen(ff3Ta, x0, c_ex, dc_ex, epsilon, Nmax, a);
        cout << opt << endl;
        saveResultToFile(opt, "wyniki.txt");
        cout << norm(opt.x) << endl << endl;
        saveResultToFile(norm(opt.x), "wyniki.txt");
        solution::clear_calls();
        opt = pen(ff3Tb, x0, c_in, dc_in, epsilon, Nmax, a);
        cout << opt << endl;
        saveResultToFile(opt, "wyniki.txt");
        cout << norm(opt.x) << endl << endl;
        saveResultToFile(norm(opt.x), "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";
        }
        //cout << testFun(opt.x) << endl;
        solution::clear_calls();

    //Rzut pilka
    x0 = matrix(2, 1);
    x0(0) = 20 * m2d(rand_mat()) - 10;
    x0(1) = 40 * m2d(rand_mat()) - 20;
    cout << x0 << endl << endl;
    opt = pen(ff3R, x0, c_ex, dc_ex, epsilon, Nmax);
    int n = get_len(symulation[0]);
    for (int i = 0; i < n; ++i)
        cout << symulation[1](i, 0) << "," << symulation[1](i, 2) << endl;
    opt.y = -opt.y;
    cout << opt << endl;
    solution::clear_calls();
}
```

```cpp
solution pen(matrix(*ff)(matrix, matrix, matrix), matrix x0, double c, double dc, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try {
        solution Xopt;
        double alpha = 1, beta = 0.5, gamma = 2, delta = 0.5, s = 0.5;
        solution X(x0), X1;
        while (true)
        {
            X1 = sym_NM(ff, X.x, s, alpha, beta, gamma, delta, epsilon, Nmax, ud1, c);
            if (norm(X1.x - X.x) < epsilon)
            {
                Xopt = X1;
                Xopt.flag = 1;
                break;
            }
            if (solution::f_calls > Nmax)
            {
                Xopt = X1;
                Xopt.flag = 0;
                break;
            }
            c = dc * c;
            X = X1;
        }
        return Xopt;
    }
    catch (string ex_info)
    {
        throw ("solution pen(...):\n" + ex_info);
    }
}
```

```cpp
solution sym_NM(matrix(*ff)(matrix, matrix, matrix), matrix x0, double s, double alpha, double beta, double gamma, double delta, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        solution Xopt;
        int n = get_len(x0);
        matrix D = ident_mat(n);
        int N = n + 1;
        solution* S = new solution[N];
        S[0].x = x0;
        S[0].fit_fun(ff, ud1, ud2);
        for (int i = 1; i < N; ++i)
        {
            S[i].x = S[0].x + s * D(i - 1);
            S[i].fit_fun(ff, ud1, ud2);
        }
        solution PO, PE, PZ;
        matrix pc;
        int i_min, i_max;
        while (true)
        {
            i_min = i_max = 0;
            for (int i = 1; i < N; ++i)
            {
                if (S[i_min].y > S[i].y)
                    i_min = i;
                if (S[i_max].y < S[i].y)
                    i_max = i;
            }
            pc = matrix(n, 1);
            for (int i = 0; i < N; ++i)
                if (i != i_max)
                    pc = pc + S[i].x;

            pc = pc / (N - 1.0);
            PO.x = pc + alpha * (pc - S[i_max].x);

            PO.fit_fun(ff, ud1, ud2);

            if (S[i_min].y <= PO.y && PO.y < S[i_max].y)
                S[i_max] = PO;
            else if (PO.y < S[i_min].y)
            {
                PE.x = pc + gamma * (PO.x - pc);
                PE.fit_fun(ff, ud1, ud2);

                if (PE.y < PO.y)
                    S[i_max] = PE;
                else
                    S[i_max] = PO;
            }
            else
```

```
                }
                else
                {
                    PZ.x = pc + beta * (S[i_max].x - pc);
                    PZ.fit_fun(ff, ud1, ud2);
                    if (PZ.y < S[i_max].y)
                        S[i_max] = PZ;
                    else
                    {
                        for (int i = 0; i < N; ++i)
                            if (i != i_min)
                            {
                                S[i].x = delta * (S[i].x + S[i_min].x);
                                S[i].fit_fun(ff, ud1, ud2);
                            }
                    }
                }

                double max_s = norm(S[0].x - S[i_min].x);
                for (int i = 1; i < N; ++i)
                    if (max_s < norm(S[i].x - S[i_min].x))
                        max_s = norm(S[i].x - S[i_min].x);
                if (max_s < epsilon)
                {
                    Xopt = S[i_min];
                    Xopt.flag = 1;
                    break;
                }
                if (solution::f_calls > Nmax)
                {
                    Xopt = S[i_min];
                    Xopt.flag = 0;
                    break;
                }
            }
            delete[]S;
            return Xopt;
        }
        catch (string ex_info)
        {
            throw ("solution sym_NM(...):\n" + ex_info);
        }
}
```

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

using namespace std;

double g(int i, double x1, double x2, double a)
{
    switch (i)
    {
    case 1:
        return -(x1 - 1.0);
    case 2:
        return -(x2 - 1.0);
    case 3:
        return sqrt(pow(x1, 2) + pow(x2, 2)) - a;
    }
}

matrix* symulation;

double testFun(matrix x)
{
    double symulation = M_PI * sqrt(pow(x(0) / M_PI, 2) + pow(x(1) / M_PI, 2));
    return sin(symulation) / symulation;
}

matrix ff3Ta(matrix x, matrix ud1, matrix ud2)
{
    double function = testFun(x);
    double penalty = 0.0;

    for (int i = 1; i <= 3; ++i) {
        double gValue = g(i, x(0), x(1), ud1(0));
        if (gValue > 0)
            penalty += pow(gValue, 2);
    }
    return function + ud2 * penalty;
}

matrix ff3Tb(matrix x, matrix ud1, matrix ud2)
{
    double function = testFun(x);
    double penalty = 0.0;

    for (int i = 1; i <= 3; ++i) {
        double gValue = g(i, x(0), x(1), ud1(0));
        penalty += 1 / gValue;
    }
    return function + ud2 * -penalty;
}
```

```cpp
matrix ff3R(matrix x, matrix ud1, matrix ud2)
{
    matrix y;
    matrix Y0(4, new double[4]{ 0, x(0), 100, 0 });
    matrix* Y = solve_ode(df3, 0, 0.01, 7, Y0, ud1, x(1));
    int n = get_len(Y[0]);
    int i50 = 0, i0 = 0;
    for (int i = 0; i < n; ++i)
    {
        if (abs(Y[1](i, 2) - 50) < abs(Y[1](i50, 2) - 50))
            i50 = i;
        if (abs(Y[1](i, 2)) < abs(Y[1](i0, 2)))
            i0 = i;
    }
    y = -Y[1](i0, 0);
    if (abs(x(0)) - 10 > 0)
        y = y + ud2 * pow(abs(x(0)) - 10, 2);
    if (abs(x(1)) - 20 > 0)
        y = y + ud2 * pow(abs(x(1)) - 20, 2);
    if (abs(Y[1](i50, 0) - 5) - 1 > 0)
        y = y + ud2 * pow(abs(Y[1](i50, 0) - 5) - 1, 2);
    symulation = Y;
    return y;
}

matrix df3(double t, matrix Y, matrix ud1, matrix ud2)
{
    double C = 0.47, r = 0.12, m = 0.6, ro = 1.2, g = 9.81;
    double S = 3.14 * r * r,
        Dx = 0.5 * C * ro * S * Y(1) * abs(Y(1)),
        Dy = 0.5 * C * ro * S * Y(3) * abs(Y(3)),
        FMx = 3.14 * ro * Y(3) * m2d(ud2) * pow(r, 3),
        FMy = 3.14 * ro * Y(1) * m2d(ud2) * pow(r, 3);
    matrix dY(4, 1);
    dY(0) = Y(1);
    dY(1) = (-Dx - FMx) / m;
    dY(2) = Y(3);
    dY(3) = (-m * g - Dy - FMy) / m;
    return dY;
}
```

```cpp
#pragma once

#include"ode_solver.h"

matrix ff0T(matrix, matrix = NAN, matrix = NAN);
matrix ff0R(matrix, matrix = NAN, matrix = NAN);
matrix df0(double, matrix, matrix = NAN, matrix = NAN);

matrix ff3Ta(matrix, matrix = NAN, matrix = NAN);
matrix ff3Tb(matrix, matrix = NAN, matrix = NAN);
matrix ff3R(matrix, matrix = NAN, matrix = NAN);
matrix df3(double, matrix, matrix = NAN, matrix = NAN);

extern matrix* symulation;
double testFun(matrix x);
```

Wnioski:

Funkcje kary których używamy w naszym kodzie są odpowiedzialne za monitorowanie i kary w przypadku w którym ograniczenia optymalizacji zostaną naruszone. Zadaniem tych funkcji jest ukaranie tych rozwiązań które nie spełniają warunków ograniczeń. Kary te możemy zauważyć bo rozbieżnościach w wartościach „y”. Analizując wyniki dochodzimy do wniosków że funkcje kary mogą różnić się dokładnością i szybkością. Wyłonienie szybszej funkcji zależy głównie od charakterystyki problemu i implementacji naszego kodu. Analizując mój przypadek doszedłem do wniosku że szybkość i wydajność funkcji kary zależna była od wielkości parametru „a” ponieważ dla wyższych wartości tego parametru szybciej radziła sobie z tym problemem funkcja zewnętrzna. Analizując wyniki problemu rzeczywistego możemy dojść do wniosku że funkcja spełnia powierzone jej zadanie i w prawidłowy sposób oblicza ruch piłki. W zależności od losowo przyjętych danych początkowych piłka osiąga różne wyniki końcowe które jednak zgadzają się z przeprowadzona symulacją.