|  |  |
| --- | --- |
| Optymalizacja funkcji wielu zmiennej metodami bezgradientowymi | |
| Autorzy | Kozłowski Bartosz, Kopeć Jakub, Kobyłecki Emil |
| Data wysłania | 1.12.2020 |

**1. Wejściowe parametry algorytmu sympleks Neldera – Meada dla testowej funkcji celu:**

**Funkcje kary:**

-100 punktów startowych o losowych współrzędnych, lecz nie mniejszych od 1 oraz nie większych od współczynnika a,

-początkowa wartość współczynnika kary: 1,

-mnożnik współczynnika kary: 0.5,

-dopuszczalny błąd optymalizacji: 0.001,

-maksymalna ilość wywołań funkcji celu: 1000

**2. Wejściowe parametry algorytmu sympleks Neldera – Meada dla problemu rzeczywistego:**

**Funkcja kary:**

-współrzędne punktu startowego: [2,2],

-początkowa wartość współczynnika kary: 1,

-mnożnik współczynnika kary: 2,

-dopuszczalny błąd optymalizacji: 0.001,

-maksymalna ilość wywołań funkcji celu: 1000

**Funkcja sym\_SM:**  
-współrzędne punktu startowego są wyznaczane przez funkcję kary,

**-**początkowy bok simpleksu: 0.5  
-współczynnik odbicia: 1

**-**współczynnik zawężenia: 0.5

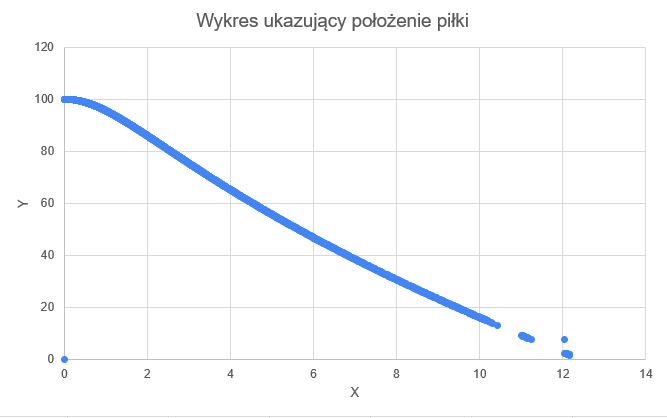
**-**współczynnik ekspansji: 2

**-**współczynnik redukcji: 0.5  
-dopuszczalny błąd optymalizacji: 0.001

-maksymalna ilość wywołań funkcji celu: 1000

**3. Dyskusja wyników oraz wnioski:**

Obie funkcje kary pomimo różnych oraz losowych punktów startowych zwróciły zbliżone do siebie wyniki. Niemal wszystkie zwrócone parametry obu funkcji są niemal identyczne dla wszystkich trzech współczynników a, poza ilością wywołań funkcji celu. Dla zdecydowanej większości iteracji, wewnętrzna funkcja kary potrzebowała kilkakrotnie więcej wywołań funkcji celu niż zewnętrzna funkcja kary.



Wedle danych z symulacji, wypuszczona piłka przemieściła się 12,41 metrów w kierunku poziomym, zanim opadła na ziemię. W momencie, w którym była ona na wysokości 50 metrów, zmieniła swoje położenie w kierunku poziomym o 5,61 metra. Część punktów w symulacji została wycięta z powodu ciągłego formatowania danych na daty. Ich ręczna zmiana byłaby bardzo żmudnym procesem, przez co wygodniejszym sposobem było pozbycie się wadliwych danych.

W metodzie Neldera-Meada wybrany sympleks jest przekształcany za pomocą operacji geometrycznych takich jak: ekspansja, redukcja, odbicie, zawężenie. Ich wynikiem jest stopniowe przybliżanie się do znalezienia rozwiązania, ponieważ punkty, dla których funkcja przyjmuje największe wartości są odrzucane, a w ich miejsce dołączane są kolejne punkty.

Fragment pliku main.cpp dla funkcji testowej:

#elif LAB\_NO == 4

//dla funkcji testowej !

//ifstream dla pktow startowych wczytywanych, ofstream dla losowanych

ifstream x01("x01.txt");    //punkty startowe

ifstream x02("x02.txt");

ofstream x1("x1.txt");  //wyniki

ofstream x2("x2.txt");

ofstream y("y.txt");

ofstream rad("rad.txt");    //promien

ofstream f\_calls("f\_calls.txt");    //wywolania funkcji

    matrix wsp\_a(3, 1);

    wsp\_a(0) = 40.0;

    wsp\_a(1) = 4.4934;

    wsp\_a(2) = 5;

    matrix x0(2, 1);

    x0(0) = 1;

    x0(1) = 1;

    double c0 = 1;

    //dla kary zewnetrznej

    double dc = 2;

    //dla kary wewnetrznej

    //double dc = 0.5;

    double epsilon = 0.001;

    int Nmax = 1000;

    double r = 0;

    int sukcesy = 0;

    double rWyniku = 0;

    solution::clear\_calls();

    //pętla działa dopóki nie wylosuje sie 100 pktow startowych spełniających warunki g

    while(sukcesy != 1)

    {

        //losowanie punktow startowych

        //x0(0) = fRand(1, wsp\_a(0));

        //x0(1) = fRand(1, wsp\_a(0));

        //wczytywanie punktow startowych z pliku

        x01 >> x0(0);

        x02 >> x0(1);

        x0(0) = 1;

        x0(1) = 1;

        r = sqrt(pow(x0(0), 2) + pow(x0(1), 2));

        //sprawdzenie czy g3 < 0 dla punktow startowych

        if (r <= wsp\_a(0)) {

            sukcesy++;

            //x01 << x0(0) << endl;

            //x02 << x0(1) << endl;

            solution sympleks = pen(x0, c0, dc, epsilon, Nmax, wsp\_a);

            sympleks.fit\_fun();

            rWyniku = sqrt(pow(sympleks.x(0), 2) + pow(sympleks.x(1), 2));

            //wypisywanie do plików

            x1 << sympleks.x(0) << endl;

            x2 << sympleks.x(1) << endl;

            rad << rWyniku << endl;

            y << sympleks.y << endl;

            f\_calls << solution::f\_calls << endl;

            cout << sympleks;

            solution::clear\_calls();

        }

    }

    cout << "sukcesow: " << sukcesy << endl;

#endif

Fragment pliku main.cpp dla problemu rzeczywistego:

#elif LAB\_NO == 4

//poczatkowe wartosc vox oraz omega

matrix x0(2,1);

x0(0) = 2;

x0(1) = 2;

//współczynniki do kary

double C = 1;

double dc = 2;

double epsilon = 0.00001;

int Nmax = 1000;

solution wynik = pen(x0, C, dc, epsilon, Nmax);

cout << "vox= " << wynik.x(0) << "\nOmega: " << wynik.x(1)<<endl;

cout << endl << "xend:" << (-1) \* wynik.y(0) << endl;//xend z konspektu,musi być mnożeno przez -1

ofstream f\_calls("f\_calls.txt");    //wywolania funkcji

f\_calls << solution::f\_calls << endl;//zapis wywolań funkcji do pliku

solution::clear\_calls();

Funkcja kary oraz Simpleks Neldera-Maeda:

solution pen(matrix x0, double c0, double dc, double epsilon, int Nmax, matrix O)

{

    double alfa = 1, beta = 0.5, gama = 2, delta = 0.5, s = 0.5;

    matrix A(new double[2]{ c0,O(0) }, 2);

    solution X, X1;

    X.x = x0;

    while (true)

    {

        X1 = sym\_NM(X.x, s, alfa, beta, gama, delta, epsilon, Nmax, A);

        if (solution::f\_calls > Nmax || norm(X1.x - X.x) < epsilon)

            return X1;

        A(0) \*= dc;

        X = X1;

    }

}

//s - poczatkowy bok simpleksu

//alfa - wsp odbicia = 1

//beta - wsp zawezenia = 1/2

//gamma - wsp ekspansji = 2

//delta - wsp redukcji = 1/2

//epsilon = 0.01

solution sym\_NM(matrix x0, double s, double alfa, double beta, double gama, double delta, double epsilon, int Nmax, matrix O)

{

    int\* n = get\_size(x0);

    //D = ident\_mat(n[0]);

    //liczba wierzcholkow sympleksu

    int N = n[0] + 1;

    //sympleks

    solution\* S = new solution[N];

    S[0].x = x0;

    S[0].fit\_fun(O);

    matrix D(n[0],n[0]);

    for (int i = 0; i < n[0]; i++) {

        D(i, i) = 1;

    }

    //cout << "wierzcholki sympleksu:\n";

    //cout << "s = " << s << endl;

    //cout << S[0].x << endl;

    //sprawdzic

    //cout << "test1\n";

    for (int i = 1; i < N; ++i)

    {

        //cout << "czy to tutaj\n";

        S[i].x = S[0].x + D[i-1]\*s;

        //cout << S[i].x << endl;

        S[i].fit\_fun(O);

    }

    cout << endl;

    solution p\_o, p\_e, p\_z;

    matrix p\_sr;

    int i\_min, i\_max;

    while (true)

    {

        i\_min = i\_max = 0;

        //cout << "test2\n";

        for (int i = 1; i < N; ++i)

        {

            if (S[i].y < S[i\_min].y)

                i\_min = i;              //najlepszy punkt

            if (S[i].y > S[i\_max].y)

                i\_max = i;              //najgorszy punkt

        }

        //punkt srodkowy do symetrii odbicia

        p\_sr = matrix(n[0], 1);

        //cout << "test3\n";

        for (int i = 0; i < N; ++i)

        {

            if (i != i\_max)

                p\_sr = p\_sr + S[i].x;

        }

        p\_sr = p\_sr / (N-1);

        //odbicie

        //cout << "test4\n";

        p\_o.x = p\_sr + alfa\*(p\_sr - S[i\_max].x);

        p\_o.fit\_fun(O);

        //akceptacja odbicia

        //cout << "test5\n";

        //cout <<"punkt odbicia  "<< p\_o.y << endl;

        //cout << "S[i\_max].y  " << S[i\_max].y << endl;

        //cout << "S[i\_min].y  " << S[i\_min].y << endl;

        if (p\_o.y >= S[i\_max].y && p\_o.y < S[i\_min].y)

        {

            //cout << "tutaj\n";

            S[i\_max] = p\_o;

            //cout << "tutaj\n";

        }

        //ekspansja

        else if (p\_o.y < S[i\_min].y)

        {

            //cout << "test6\n";

            p\_e.x = p\_sr + gama\*(p\_o.x - p\_sr);

            p\_e.fit\_fun(O);

            //akceptacja ekspansji

            if (p\_e.y < p\_o.y)

                S[i\_max] = p\_e;

            //odrzucenie ekspansji i powrot do wyniku odbicia

            else

                S[i\_max] = p\_o;

            //cout << "test7\n";

        }

        //zawężenie

        else

        {

            //cout << "test yo mama\n";

            p\_z.x = p\_sr + beta\*(S[i\_max].x - p\_sr);

            p\_z.fit\_fun(O);

            //akceptacja zawężenia

            if (p\_z.y < S[i\_max].y)

                S[i\_max] = p\_z;

            //odrzucenie zaweżenia i redukcja

            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(O);

                    }

            }

        }

        //cout << "test8\n";

        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)

            return S[i\_min];

    }

}

Funkcja fit function dla problemu rzeczywistego:

void solution::fit\_fun(matrix O)

{

    //PROBLEM RZECZYWISTY

    //Poczatkowe wartości położenia x, prędkości x,położenia y

    matrix Y0(4, 1);

    Y0(0) = 0;

    Y0(1) = x(0);

    Y0(2) = 100;

    Y0(3) = 0;

    matrix \*Y = solve\_ode(0, 0.01, 7, Y0, x(1));

    double x0 = 0, x50 = 0; //x50 - położenie piłki w osi x na wysokości 50, x0 analogicznie dla wys. 0

    for (int i=0;i<\*get\_size(Y[1]);i++)

    {

        if(fabs(Y[1](i,2)-50)<fabs(Y[1](x50,2)-50))

        {

            x50=i;

        }

        if(fabs(Y[1](i,2))<fabs(Y[1](x0,2)))

        {

            x0=i;

        }

    }

    y=-Y[1](x0,0); // y = -x0 z wyżej

    //Funkcje g

    if(x(0)<-10)y=y+pow(-x(0)-10,2); //g1

    if(x(0)>10) y=y+pow(x(0)-10,2); //g2

    if(x(1)<-20)y=y+pow(-x(1)-20,2);//g3

    if(x(1)>20)y=y+pow(x(1)-20,2);//g4

    if(Y[1](x50,0)<4)y=y+pow(-Y[1](x50,0)+4,2);//g5

    if(Y[1](x50,0)>6)y=y+pow(Y[1](x50,0)-6,2);//g6

    ++f\_calls;

}

Funkcja diff

matrix diff(double t, const matrix &Y, matrix P) //return pochodne of all variables

{

#elif LAB\_NO == 4

//DLA PROBLEMU RZECZYWISTEGO

    //współczynniki z konspektu

    double C=0.47,m=0.6,r=0.12,rho=1.2,g=9.81,S=3.14\*r\*r;

    //przemieszczenie w osi x oraz y

    double Dx=0.5\*C\*rho\*S\*Y(1)\*abs(Y(1)); //Y(1)- prędkość pozioma

    double Dy=0.5\*C\*rho\*S\*Y(3)\*abs(Y(3)); //Y(3)- prędkość pionowa

    //Siły Magnusa

    double Fmx=3.14\*rho\*Y(3)\*P(0)\*pow(r,3); //Siła Magnusa x

    double Fmy=3.14\*rho\*Y(1)\*P(0)\*pow(r,3); //Siła Magnus y

    //Macierz z rozwiązaniami

    matrix dY(4,1);

    dY(0)=Y(1);

    dY(1)=(-Dx-Fmx)/m;

    dY(2)=Y(3);

    dY(3)=(-m\*g-Dy-Fmy)/m;

    //Wypisywanie kolejnych położeń piłki w osi Y oraz X

    cout<<Y(2)<<" "<<Y(1)<<endl;

    return dY;

#endif

}