Optymalizacja IT gr. 2

Dariusz Homa

Wojciech Jurgielewicz

Michał Koleżyński

Projekt 3

Cel ćwiczenia

Celem ćwiczenia jest wykorzystanie bezgradientowych metod optymalizacji do wyznaczenia minimum funkcji celu uwzględniając ograniczenia.

Wykonanie

Kody funkcji

Metoda Neldera-Meada

```
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 = 3;
        matrix base(3, 3);
        base(0, 0) = 1.0;
        base(1, 1) = 1.0;
        base(2, 2) = 1.0;

        std::vector<solution> p;
        p.reserve(n);
        p.push_back(x0);

        for (int i = 1; i < n; i++)
        {
            double* coords = new double[2] {
                p[0].x(0) + s * base(i, i),
                p[0].x(1) + s * base(i, i)
            };
```

```
        solution newP(2, coords);
        p.push_back(newP);
    }

    int max_index{};
    int min_index{};

    solution p_(matrix(2, 1, 0.0));

    solution p_odb;
    solution p_z;
    solution p_e;

    double max = 0;

    do
    {
        for (int i = 0; i < n; i++)
            p[i].fit_fun(ff, ud1, ud2);

        min_index = 0;
        max_index = 0;

        for (int i = 0; i < n; i++)
        {
            if (p[i].y > p[max_index].y)
                max_index = i;

            if (p[i].y < p[min_index].y)
                min_index = i;
        }

        for (int i = 0; i < n; i++)
            if (i != max_index)
                p_.x = p_.x + p[i].x;

        p_.x = p_.x / n;
```

```
            p_odb.x = p_.x + alpha * (p_.x - p[max_index].x);
            p_odb.fit_fun(ff, ud1, ud2);

            if (p_odb.y < p[min_index].y)
            {
                p_e.x = p_.x + gamma * (p_odb.x - p_.x);
                p_e.fit_fun(ff, ud1, ud2);

                if (p_e.y < p_odb.y)
                    p[max_index] = p_e;
                else
                    p[max_index] = p_odb;
            }
            else
            {
                if (p[min_index].y <= p_odb.y && p_odb.y <
p[max_index].y)
                {
                    p[max_index] = p_odb;
                }
                else
                {
                    p_z.x = p_.x + beta * (p[max_index].x - p_.x);
                    p_z.fit_fun(ff, ud1, ud2);

                    if (p_z.y >= p[max_index].y)
                    {
                        for (int i = 0; i < n; i++)
                        {
                            if (i != min_index)
                            {
                                p[i].x = delta * (p[i].x +
p[min_index].x);
                                p[i].fit_fun(ff, ud1, ud2);
                            }
                        }
                    }
```

```
                    else
                    {
                        p[max_index] = p_z;
                    }
                }
            }

            max = 0;

            for (int i = 0; i < n; i++)
            {
                double value = abs(m2d(p[min_index].y) -
m2d(p[i].y));

                if (value > max)
                    max = value;
            }

            if (solution::f_calls > Nmax)
            {
                throw std::string("Przekroczono limit wywolan
funkcji :)");
            }
        } while (max > epsilon);

        return p[min_index];
    }
    catch (string ex_info)
    {
        throw("solution sym_NM(...):\n" + ex_info);
    }
}
```

Funkcja kary

```
solution pen(matrix(*ff)(matrix, matrix, matrix), matrix x0, double
c, double dc, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
```

```
        solution XB;
        XB.x = x0;
        XB.fit_fun(ff, ud1, c);

        solution XT;
        XT = XB;

        double s = ud2(0);
        double alpha = ud2(1);
        double beta = ud2(2);
        double gamma = ud2(3);
        double delta = ud2(4);

        while (true)
        {
            XT = sym_NM(ff, XB.x, s, alpha, beta, gamma, delta,
epsilon, Nmax, ud1, c);
            c *= dc;

            if (solution::f_calls > Nmax)
                throw std::string("Przekroczono limit wywolan
funkcji :)");

            if (norm(XT.x - XB.x) < epsilon)
                break;

            XB = XT;
        };

        return XT;
    }
    catch (string ex_info)
    {
        throw("solution pen(...):\n" + ex_info);
    }
}
```

Testowa funkcja celu

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

Ograniczenia:

$$g_1(x_1) = -x_1 + 1 \leq 0$$

$$g_2(x_2) = -x_2 + 1 \leq 0$$

$$g_3(x_1, x_2) = \sqrt{x_1^2 + x_2^2} - a \leq 0$$

Cel

- Wykonanie 100 optymalizacji

Parametry

- długość boku sympleksu początkowego $s = 0.5$
- współczynnik odbicia $\alpha = 1.0$
- współczynnik zawężenia $\beta = 0.5$
- współczynnik ekspansji $\gamma = 2.0$
- współczynnik redukcji $\delta = 0.5$
- dokładność $\epsilon = 10^{-4}$
- maksymalna liczba wywołań funkcji $N_{max} = 10\text{^}4$
- współczynniki skalowania $a \in \{4, 4.4934, 5.0\}$
- współczynniki c dla
    - wewnętrznej funkcji kary: $c_{in} = 10,\ dc_{in} = 0.2$
    - zewnętrznej funkcji kary: $c_{out} = 1,\ dc_{out} = 1.5$

Kod

```
double s = 0.5;
double alpha = 1.0;
double beta = 0.5;
double gamma = 2.0;
double delta = 0.5;

matrix userData(5, 1);
userData(0) = s;
userData(1) = alpha;
userData(2) = beta;
userData(3) = gamma;
userData(4) = delta;

double epsilon = 1e-4;
int Nmax = 10000;

double c_inner = 10.0;
double dc_inner = 0.2;

double c_outer = 1.0;
```

```
double dc_outer = 1.5;

double a_tab[] = {
    4.0,
    4.4934,
    5.0
};

for (double a : a_tab)
{
    for (int i = 0; i < 1; i++)
    {
        matrix x(2, 1);
        x(0) = RandomNumberGenerator::Get().Double(1.5, 5.5);
        x(1) = RandomNumberGenerator::Get().Double(1.5, 5.5);

        SAVE_TO_FILE("x-" + std::to_string(a) + ".txt") << x(0) <<
";" << x(1) << "\n";

        solution penResultOut = pen(lab3_fun_outer, x, c_outer,
dc_outer, epsilon, Nmax, a, userData);
        solution::clear_calls();

        solution penResultIn = pen(lab3_fun_inner, x, c_inner,
dc_inner, epsilon, Nmax, a, userData);
        solution::clear_calls();
    }
}
```

Funkcja celu

```
matrix lab3_fun_help(matrix x, matrix ud1, matrix ud2)
{
    return sin(M_PI * sqrt(pow(x(0) / M_PI, 2) + pow(x(1) / M_PI,
2))) / (M_PI * sqrt(pow(x(0) / M_PI, 2) + pow(x(1) / M_PI, 2)));
}

matrix lab3_fun_outer(matrix x, matrix ud1, matrix ud2)
{
```

```
    matrix y = lab3_fun_help(x, ud1, ud2);

    if (-x(0) + 1 > 0)
        y = y + ud2 * pow(-x(0) + 1, 2);

    if (-x(1) + 1 > 0)
        y = y + ud2 * pow(-x(1) + 1, 2);

    if (norm(x) - ud1 > 0)
        y = y + ud2 * pow(norm(x) - ud1, 2);

    return y;
}

matrix lab3_fun_inner(matrix x, matrix ud1, matrix ud2)
{
    matrix y = lab3_fun_help(x, ud1, ud2);

    if (-x(0) + 1 >= 0)
        y = 1e10;
    else
        y = y - m2d(ud2) / (-x(0) + 1);

    if (-x(1) + 1 >= 0)
        y = 1e10;
    else
        y = y - m2d(ud2) / (-x(1) + 1);

    if (norm(x) - m2d(ud1) > 0)
        y = 1e10;
    else
        y = y - m2d(ud2) / (norm(x) - ud1);

    return y;
}
```

100 optymalizacji na losowym przedziale

Omówienie wyników

Analizując tabelę nr 2 można dostrzec, że podanie mniejszych współczynników skalowania a sprawia, że wynik znalezionego r jest znacząco dokładniejszy, kosztem dwukrotnego zwiększenia liczby wywołań funkcji celu. Ponadto, widać że podanie wartości współczynnika skalowania jako liczbę w przedziale $[a, b]$, sprawia, że wynik jest bardzo zbliżony do tego niedokładniejszego – dla większego współczynnika skalowania - b. W celu znalezienia najbardziej dokładnego wyniku należy podawać współczynniki a jako jak najmniejsze liczby całkowite.

Problem rzeczywisty

Cel

Znalezienie takich wartości $V_{0x} \in [-10{,}10]\frac{m}{s}$ oraz $\omega = [-15{,}15]\frac{rad}{s}$ które zapewnią największą wartość x_{end}.

Parametry

- Prędkość liniowa pozioma, początkowa: $V_{0x} = 5\frac{m}{s}$
- Prędkość kątowa początkowa: $\omega = 10\frac{rad}{s}$
- Masa piłki: $m = 0.6kg$
- Promień piłki: $r = 0.12m$
- Wysokość początkowa: $y_0 = 100m$
- Współczynnik oporu: $C = 0.47$
- Gęstość powietrza: $\rho = 1.2kg/m^3$
- Przyśpieszenie grawitacyjne: $g = 9.81\frac{m}{s^2}$
- Parametry funkcji *pen(...) i sym_NM()* – jak wyżej

Kod

Funkcja różniczkowa df3

```
matrix df3(double t, matrix Y, matrix ud1, matrix ud2)
{
    matrix dY(4, 1);

    double C = 0.47;
    double rho = 1.2;
    double r = 12 * 0.01;
    double m = 0.6;
    double omega = ud2(0);

    double S = M_PI * pow(r, 2);
    double Dx = 0.5 * C * rho * S * Y(1) * abs(Y(1));
    double Dy = 0.5 * C * rho * S * Y(3) * abs(Y(3));
    double Fmx = rho * Y(3) * omega * M_PI * pow(r, 3);
    double Fmy = rho * Y(1) * omega * M_PI * pow(r, 3);
```

```
    dY(0) = Y(1);
    dY(1) = (-Dx - Fmx) / m;
    dY(2) = Y(3);
    dY(3) = (-m * EARTH_ACCELERATION - Dy - Fmy) / m;

    return dY;
}
```

Równanie różniczkowe f3R

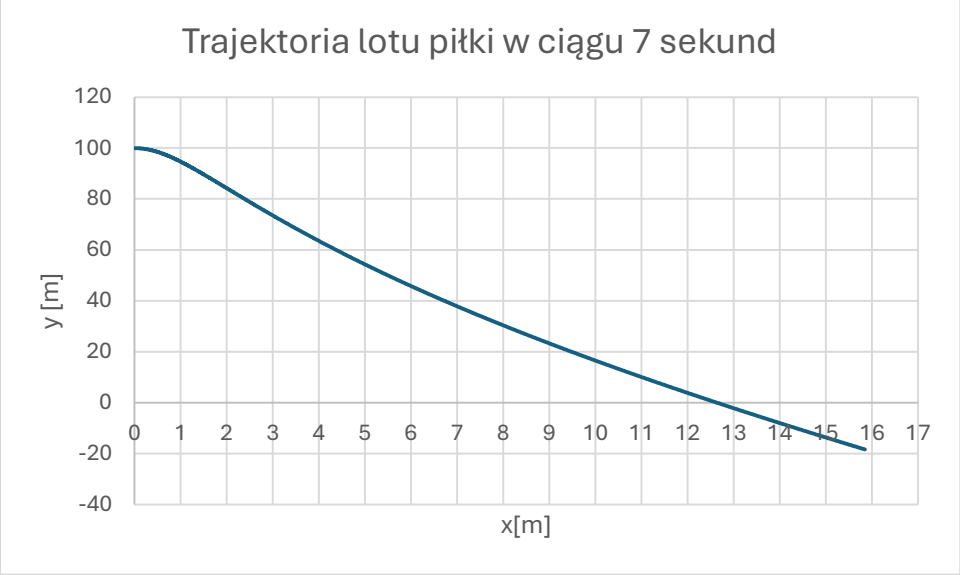
```
matrix f3R(matrix x, matrix ud1, matrix ud2)
{
    matrix Y0(4, new double[4] {0, x(0), 100, 0 });
    matrix* Y = solve_ode(df3, 0, 0.01, 7, Y0, ud1, x(1));

    int n = get_len(Y[0]);
    int i0 = 0;
    int i50 = 0;
    matrix result;

    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;
    }

    result = -Y[1](i0, 0);

    if (abs(x(0)) - 10 > 0)
        result = result + ud2 * pow(abs(x(0)) - 10, 2); // ud2 = c

    if (abs(x(1)) - 15 > 0)
        result = result + ud2 * pow(abs(x(1)) - 15, 2);

    if (abs(Y[1](i50, 0) - 5) - 0.5 > 0)
        result = result + ud2 * pow(abs(Y[1](i50, 0) - 5) - 0.5, 2);

    Y[0].~matrix();
    Y[1].~matrix();

    return result;
}
```

Symulacja

```
    {
        matrix x0(2, 1);
        x0(0) = 5.0;
        x0(1) = 10.0;

        double c = 1.0;
        double dc = 1.5;

        solution optimal = pen(f3R, x0, c, dc, epsilon, Nmax, NULL,
userData);
        std::cout << optimal << "\n";
        solution::clear_calls();

        matrix Y0(4, new double[4] { 0.0, optimal.x(0), 100, 0 });
        matrix* Y = solve_ode(df3, 0.0, 0.01, 7.0, Y0, NULL,
optimal.x(1));

        std::cout << hcat(Y[0], Y[1]);
        SAVE_TO_FILE("symulacja.txt") << hcat(Y[0], Y[1]);
    }
```

Omówienie wyników

Z optymalizacji problemu rzeczywistego o powyższych warunkach zadania, dowiadujemy się że dla początkowej wartości prędkości liniowej $V_{0x} = 0.896399\ m/s$ i prędkości kątowej $\omega = 3.07097 \frac{rad}{s}$ odległość piłki osiągnie **największą możliwą wartość**, równą $x_{max} = 12.6442\ m$. Uruchamiając optymalizację dla tych wartości początkowych widzimy, że piłka dotknie powierzchni ziemi po $t = 6.14\ s$.



Wnioski

- Użycie współczynników skalowania w funkcjach kary ma kluczowe znaczenie dla precyzji wyniku i liczby iteracji. Mniejsze współczynniki prowadzą do większej dokładności, ale zwiększają liczbę wywołań funkcji celu.
- Optymalizacja metodą sympleks Neldera-Meada dla zewnętrznej funkcji kary znajduje wynik znacząco szybciej niż dla wewnętrznej funkcji kary. Gdy porównamy wyniki dla obu przypadków widzimy, że dla tego samego współczynnika a znajdują podobne wartości.