Optymalizacja IT gr. 2

Dariusz Homa

Wojciech Jurgielewicz

Michał Koleżyński

Projekt 2

Cel ćwiczenia

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

Wykonanie

Kody funkcji

Metoda Hooke'a-Jeevesa

```
try
{
    solution xOpt;
    solution XB = x0;
    solution XB_;

    do
    {
        xOpt = HJ_trial(ff, XB, s);

        xOpt.fit_fun(ff, ud1, ud2);
        XB.fit_fun(ff, ud1, ud2);

        if (xOpt.y < XB.y)
        {
            do
            {
                XB_ = XB;
                XB = xOpt;
                xOpt.x = 2.0 * XB.x - XB_.x;
                xOpt = HJ_trial(ff, xOpt, s);

                xOpt.fit_fun(ff, ud1, ud2);
                XB.fit_fun(ff, ud1, ud2);
```

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

            } while (xOpt.y < XB.y);
            xOpt = XB;
        }
        else
            s *= alpha;

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

    } while (s > epsilon);

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

Próba Hooke'a-Jeevesa

```
try
{
    matrix base(2, 2);
    base(0, 0) = 1.0;
    base(1, 1) = 1.0;
    solution tempSolution;

    for (int i = 0; i < get_len(XB.x); i++)
    {
        XB.fit_fun(ff, base, ud2);
        matrix y1 = XB.y;

        tempSolution.x = XB.x + s * base[i];
        if (tempSolution.fit_fun(ff, base, ud2) < y1)
            XB.x = XB.x + s * base[i];
```

```
        else
        {
            tempSolution.x = XB.x - s * base[i];
            if (tempSolution.fit_fun(ff, base, ud2) < y1)
                XB.x = XB.x = XB.x - s * base[i];
        }
    }

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

Metoda Rosenbrocka

```
const int n = 2;

try
{
    solution Xopt;

    matrix base(n, n);
    base(0, 0) = 1.0;
    base(1, 1) = 1.0;

    matrix lambda(n, new double[n]{ 0.0 });
    matrix p(n, new double[n] { 0.0 });
    solution XB = x0;
    solution XB2 = 0;

    matrix s = s0;
    double max_s = 0.0;

    int i = 0;
    do
    {
        for (int j = 0; j < n; j++)
        {
```

```
            XB2 = XB.x + s(j) * base[j];
            XB2.fit_fun(ff, ud1, ud2);
            XB.fit_fun(ff, ud1, ud2);
            if (XB2.y < XB.y)
            {
                XB.x = XB.x + s(j) * base[j];
                lambda(j) += s(j);
                s(j) *= alpha;
            }
            else
            {
                s(j) *= -beta;
                p(j) += 1;
            }
        }

        i++;
        Xopt.x = XB.x;

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

        if (changeBase)
        {
            matrix lambdaMatrix(n, n);
            int l = 0;
            for (int k = 0; k < n; ++k)
            {
                for (int j = 0; j <= k; ++j)
                    lambdaMatrix(k, j) = lambda(l);
                ++l;
```

```
            }

            matrix Q = base * lambdaMatrix;

            matrix v1 = Q[0];
            double v1_norm = norm(v1);
            v1 = v1 / v1_norm;

            base[0] = v1;

            matrix v2 = Q[1] - (trans(Q[1]) * base[0]) * base[0];
            double v2_norm = norm(v2);
            v2 = v2 / v2_norm;

            base[1] = v2;

            lambda = matrix(n, new double[n] { 0.0 });
            p = matrix(n, new double[n] { 0.0 });
            s = s0;
        }

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

        max_s = 0.0;
        for (int j = 0; j < n; j++)
            if (abs(s(j)) > max_s)
                max_s = abs(s(j));

    } while (max_s > epsilon);

    Xopt.fit_fun(ff, ud1, ud2);
    return Xopt;
}
catch (string ex_info)
{
    throw("solution Rosen(...):\n" + ex_info);
}
```

Testowa funkcja celu

$$f(x_1, x_2) = x_1^2 + x_2^2 - \cos(2{,}5\pi x_2) + 2$$

Cel

- Wykonanie 100 optymalizacji dla trzech różnych długości kroku startując z losowego punktu.
- Dla jednego przypadku naniesienie rozwiązania optymalnego uzyskanego po każdej iteracji na wykres poziomic funkcji celu.

Parametry

$$\alpha_{hook} = 0.1$$

$$\beta = 0.5$$

$$\varepsilon = 10^{-4}$$

$$N_{max} = 10000$$

$$\alpha_{rosen} = 1.1$$

Długość kroku dla testowej funkcji celu:

$$s_1 = 0.1;\ s_2 = 0.2;\ s_3 = 0.3$$

Przypadek do wykresu iteracji:

$$s = 0.2;\ x_1 = 0{,}2388;\ x_2 = -0{,}74822$$

Kod

Funkcja celu

```
matrix lab2_fun(matrix x, matrix ud1, matrix ud2)
{
    matrix y;

    y = pow(x(0), 2) + pow(x(1), 2) - cos(2.5 * M_PI * x(0)) -
cos(2.5 * M_PI * x(1)) + 2;

    return y;
}
```

100 optymalizacji na losowym przedziale

```
double alpha_1 = 0.1;
double alpha_2 = 1.1;
double beta = 0.50;
double epsilon = 1e-6;
int Nmax = 2000;

double s = 0.1;
```

```
matrix sv(2, 1);
sv(0) = s;
sv(1) = s;

for (int k = 1; k <= 3; k++)
{
    s = 0.1 * k;
    sv(0) = s;
    sv(1) = s;
    for (int i = 0; i < 100; i++)
    {
        matrix x(2, 1);
        x(0) = RandomNumberGenerator::Get().Double(-1.0, 1.0);
        x(1) = RandomNumberGenerator::Get().Double(-1.0, 1.0);
        solution::clear_calls();

        // Hooke
        solution hookeResult = HJ(lab2_fun, x, s, alpha_1, epsilon,
Nmax);
        solution::clear_calls();

        // Rosen
        solution rosenResult = Rosen(lab2_fun, x, sv, alpha_2, beta,
epsilon, Nmax);
        solution::clear_calls();
    }
}
```

Omówienie wyników

- Na podstawie tabeli 2 można wywnioskować, że dla większego rozmiaru zmniejsza się prawdopodobieństwo znalezienia minimum globalnego, gdyż algorytm staje się mniej precyzyjny.
- Z tabel 1 i 2 również można wywnioskować, że metoda Hooke'a-Jeevesa kończy działanie szybciej od metody Rosenbrocka. Ponadto, dla większego kroku algorytm Hooke'a-Jeevesa kończy działanie jeszcze szybciej, a algorytm Rosenbrocka jeszcze wolniej.
- Nie zauważono znaczącej różnicy w prawdopodobieństwie odnalezienia globalnego minimum między metodami.
- Z wykresu iteracji wynika, że metoda Hooke'a-Jeevesa szybciej zawęża przedział, w którym może być minimum.

Problem rzeczywisty

Cel

Znalezienie takich wartości współczynników k_1 oraz k_2 dla których funkcjonał $Q(k_1, k_2)$ przyjmuje najmniejszą wartość.

Parametry

- Metoda Hooke'a-Jeveesa
  - k_1 początkowe – 1.0
  - k_2 początkowe – 1.0
  - Długość kroku
    $$s = 0.4$$
  - Współczynnik zmniejszania kroku
    $$\alpha_{hooke} = 0.5$$
  - Dokładność:
    $$\epsilon = 10^{-6}$$
  - Maksymalna ilość wywołań funkcji
    $$N_{max} = 10000$$

- Metoda Rosenbrocka
  - k_1 początkowe – 1.0
  - k_2 początkowe – 1.0
  - Wektor długości kroków
    $$[s_1, s_2] = [0.4, 0.4]$$
  - Współczynnik ekspansji
    $$\alpha_{Rosen} = 1.3$$
  - Współczynnik kontrakcji:
    $$\beta = 0.5$$
  - Dokładność
    $$\epsilon = 10^{-6}$$
  - Maksymalna ilość wywołań funkcji
    $$N_{max} = 10000$$

Kod

Funkcja różniczkowa df2

```
matrix df2(double t, matrix Y, matrix ud1, matrix ud2)
{
    matrix dY(2, 1);

    double alpha_t = Y(0);
    double omega_t = Y(1);

    double b = 0.5;
```

```
    double mr = 1.0;
    double mc = 5.0;
    double l = 1.0;

    double alpha_ref = ud1(0);
    double omega_ref = ud1(1);
    double k1 = ud2(0);
    double k2 = ud2(1);

    double I = 1.0 / 3.0 * mr * pow(l, 2) + mc * pow(l, 2);
    double Mt = k1 * (alpha_ref - alpha_t) + k2 * (omega_ref -
omega_t);

    dY(0) = omega_t;
    dY(1) = (Mt - b * omega_t) / I;

    return dY;
}
```

Równanie różniczkowe f2R

```
matrix f2R(matrix x, matrix ud1, matrix ud2)
{
    matrix Y0(2, 1);
    matrix Yref(2, new double[2]{ M_PI, 0 });
    matrix* Y = solve_ode(df2, 0, 0.1, 100, Y0, Yref, x);

    double y = 0.0;
    int n = get_len(Y[0]);
    for (int i = 0; i < n; i++)
    {
        y += 10 * pow(Yref(0) - Y[1](i, 0), 2) +
            pow(Yref(1) - Y[1](i, 1), 2) +
            pow(x(0) * (Yref(0) - Y[1](i, 0)) + x(1) * (Yref(1) -
Y[1](i, 1)), 2);
    }

    y *= 0.1;
```

```
    matrix result(1, 1);
    result(0) = y;

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

    return result;
}
```

Symulacja

```
double alpha1 = 0.5; // for HJ
double alpha2 = 1.3; // for Rosen
double beta = 0.5; // for Rosen
double epsilon = 1e-6;
int Nmax = 10000;

double s = 0.4;
matrix sv(2, 1);
sv(0) = s;
sv(1) = s;

matrix ud1(2, 1);
ud1(0) = M_PI;
ud1(1) = 0;

matrix k(2, 1);
k(0) = 1.0;
k(1) = 1.0;

matrix y0(2, 1);
y0(0) = 0;
y0(1) = 0;

solution HJResult_sim = HJ(f2R, k, s, alpha1, epsilon, Nmax, ud1);
std::cout << "Simulation HJ result: " << HJResult_sim;
solution::clear_calls();
```

```
solution RosenResult_sim = Rosen(f2R, k, sv, alpha2, beta, epsilon,
Nmax, ud1);
std::cout << "Simulation Rosen result: " << RosenResult_sim;
solution::clear_calls();

matrix* simulationHooke = solve_ode(df2, 0, 0.1, 100, y0, ud1,
HJResult_sim.x);
matrix* simulationRosen = solve_ode(df2, 0, 0.1, 100, y0, ud1,
RosenResult_sim.x);
```

Omówienie wyników

Obie metody znajdują te same, optymalne wartości k_1 i k_2 i najmniejszą wartość funkcjonału $Q(k_1, k_2)$, metoda Hooke'a – Jeevesa robi to jednak zauważalnie szybciej niż metoda Rosenbrocka. Wykorzystanie metody Rosenbrocka jest nieoptymalne pod względem czasowym, w przypadku gdy nasz obszar poszukiwania jest tak duży (od 0 do 10).

Wnioski

- Wraz ze wzrostem długości kroków zmniejsza się efektywność czasowa algorytmu Rosenbrocka z jednoczesnym zmniejszeniem szansy znalezienia minimów globalnych. Metodę tą należy więc stosować w przypadku szukania minimum globalnego na mniejszych, dokładniejszych obszarach.
- Metoda Hooke'a Jeevesa, jest szybsza od metody Rosenbrocka, szczególnie przy większych krokach, kosztem dokładności – nie nadaje się więc do optymalizacji małych obszarów, lecz doskonale sprawdza się na dużych obszarach (jak w przypadku naszego zagadnienia rzeczywistego).
- Należy wykorzystywać odpowiednią metodę optymalizacji w zależności od potrzeb danego zagadnienia.