Optymalizacja IT gr. 2

Dariusz Homa

Wojciech Jurgielewicz

Michał Koleżyński

Projekt 1

Cel ćwiczenia

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

Wykonanie

Kody funkcji

Metoda ekspansji

```
double *expansion(
    matrix (*ff)(matrix, matrix, matrix),
    double x0, double d, double alpha, int Nmax,
    matrix ud1, matrix ud2)
{
    try {
        double *p = new double[2]{0, 0};

        int i = 0;

        solution sx0(x0), sx1(x0 + d);

        sx0.fit_fun(ff);
        sx1.fit_fun(ff);

        if (sx0.y == sx1.y) {
```

```
            p[0] = m2d(sx0.x);
            p[1] = m2d(sx0.y);

            return p;
        } else if (sx1.y > sx0.y) {
            d = -d;

            sx1 = sx0.x + d;

            if (sx1.y >= sx0.y) {
                p[0] = m2d(sx1.x);
                p[1] = m2d(sx0.x) - d;

                return p;
            }
        }

        solution sx2(x0 + d);
        sx2.fit_fun(ff);

        do {
            if (sx0.f_calls > Nmax)
                throw("Przekroczono limit wywolan funkcji :)");
            i++;

            sx0 = sx1;
            sx1 = sx2;

            sx2.x = m2d(sx0.x) + pow(alpha, i) * d;
            sx2.fit_fun(ff);
        } while (m2d(sx1.y) > m2d(sx2.y));

        if (d > 0) {
            p[0] = m2d(sx0.x);
            p[1] = m2d(sx2.x);

            return p;
        }
```

```
        p[0] = m2d(sx2.x);
        p[1] = m2d(sx0.x);

        return p;
} catch (string ex_info) {
        throw("double* expansion(...):\n" + ex_info);
    }
}
```

Metoda Fibonnaciego

```
solution fib(
    matrix(*ff)(matrix, matrix, matrix),
    double a, double b, double epsilon,
    matrix ud1, matrix ud2)
{
    try
    {
        std::vector<double> sigma = { 1, 1 };
        double ratio = (b - a) / epsilon;
        while (true)
        {
            if (sigma.back() > ratio)
                break;

            sigma.push_back(sigma[sigma.size() - 1] +
sigma[sigma.size() - 2]);
        }
        int k = static_cast<int>(sigma.size() - 1);

        double a0 = a;
        double b0 = b;
        double c0 = b0 - sigma[k - 1] / sigma[k] * (b0 - a0);
        double d0 = a0 + b0 - c0;
```

```
        solution c_sol, d_sol;
        for (int i = 0; i <= k - 3; ++i)
        {
            SAVE_TO_FILE("fib-b-a.txt") << b0 - a0;

            c_sol.x = c0;
            c_sol.fit_fun(ff, ud1, ud2);

            d_sol.x = d0;
            d_sol.fit_fun(ff, ud1, ud2);

            if (c_sol.y < d_sol.y)
                b0 = d0;
            else
                a0 = c0;

            c0 = b0 - sigma[k - i - 2] / sigma[k - i - 1] * (b0 -
a0);
            d0 = a0 + b0 - c0;
        }

        solution Xopt;
        Xopt.x = c0;
        Xopt.fit_fun(ff, ud1, ud2);

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

Metoda Lagrange’a

```
solution lag(
    matrix (*ff)(matrix, matrix, matrix),
    double a, double b, double epsilon, double gamma, int Nmax,
    matrix ud1, matrix ud2)
{
    try {
        solution Xopt;

        double ai = a;
        double bi = b;
        double ci = (a + b) / 2;
        double di{};

        int i = 0;
        double l{}, m{};
        solution ai_sol, bi_sol, ci_sol, di_sol;
        double l_prev{}, m_prev{}, di_prev{};
        do
        {
            SAVE_TO_FILE("lag-b-a.txt") << bi - ai;

            ai_sol.x = ai;
            ai_sol.fit_fun(ff, ud1, ud2);

            bi_sol.x = bi;
            bi_sol.fit_fun(ff, ud1, ud2);

            ci_sol.x = ci;
            ci_sol.fit_fun(ff, ud1, ud2);

            l = m2d(ai_sol.y) * (pow(bi, 2) - pow(ci, 2)) +
m2d(bi_sol.y) * (pow(ci, 2) - pow(ai, 2)) + m2d(ci_sol.y) * (pow(ai,
2) - pow(bi, 2));
            m = m2d(ai_sol.y) * (bi - ci) + m2d(bi_sol.y) * (ci -
ai) + m2d(ci_sol.y) * (ai - bi);
```

```
            if (m <= 0) {
                Xopt.flag = 0;
                break;
            }

            di = 0.5 * l / m;
            di_sol.x = di;
            di_sol.fit_fun(ff, ud1, ud2);

            if (ai < di && di < ci) {
                if (di_sol.y < ci_sol.y) {
                    bi = ci;
                    ci = di;
                } else
                    ai = di;
            } else {
                if (ci < di && di < bi) {
                    if (di_sol.y < ci_sol.y) {
                        ai = ci;
                        ci = di;
                    } else
                        bi = di;
                } else {
                    Xopt.flag = 0;
                    break;
                }
            }

            if (ai_sol.f_calls > Nmax) {
                Xopt.flag = 0;
                throw std::string("Error message!");
                break;
            }

            if (i > 0) {
                di_prev = 0.5 * l_prev / m_prev;
            }
```

```
            l_prev = l;
            m_prev = m;

            ++i;
        } while (!(bi - ai < epsilon || abs(di - di_prev) < gamma));

        Xopt.x = di;
        Xopt.fit_fun(ff, ud1, ud2);

        return Xopt;
    } catch (string &ex_info) {
        throw("solution lag(...):\n" + ex_info);
    }
}
```

Testowa funkcja celu

$$f(x) = -cos(0.1x) * e^{-(0.1x-2\pi)^2} + 0.002 * (0.1x)^2$$

Cel

- Wykonanie 100 optymalizacji dla 3 różnych współczynników ekspansji α dla losowych punktów startowych – zawężenie przedziałów metodą ekspansji, porównanie metody Fibonnacciego i Lagrange'a na zawężonych przedziałach pod względem dokładności i zbieżności.
- Dokonanie optymalizacji bez początkowego zawężania przedziału poszukiwań.

Parametry

$$\varepsilon = 10^{-20}$$

$$\gamma = 10^{-30}$$

$$N_{max} = 10000$$

Współczynniki ekspansji dla testowej funkcji celu:

$$\alpha_1 = 1.5;\ \alpha_2 = 3.0;\ \alpha_3 = 4.5$$

Kod

Funkcja celu

```
matrix lab1_fun(matrix x, matrix ud1, matrix ud2) {
    matrix fx(1, 1);
    double y = -cos(0.1 * m2d(x)) * exp(-pow((0.1 * m2d(x) - 2 *
3.14), 2)) + 0.002 * pow(0.1 * m2d(x), 2);
    fx(0) = y;
    return fx;
}
```

Optymalizacja bez początkowego zawężania przedziału poszukiwań

```
solution fibonacciResult = fib(lab1_fun, -100, 100, epsilon, Nmax);
std::cout << fibonacciResult << "\n";
solution::clear_calls();

solution lagrangeResult = lag(lab1_fun, -100, 100, epsilon, gamma,
Nmax);
std::cout << lagrangeResult << "\n";
solution::clear_calls();
```

100 optymalizacji z zawężeniem przedziałów

```
for (double alpha = 1.5; alpha <= 4.5; alpha += 1.5)
{
    for (int i = 0; i < 100; i++)
    {
        // Expansion
        double x0 = RandomNumberGenerator::Get().Double(0, 100);
        double* expansionResult = expansion(lab1_fun, x0, 1, alpha,
Nmax);

        SAVE_TO_FILE("expansion-" + std::to_string(alpha) + ".txt")
<< x0 << expansionResult[0] << expansionResult[1] <<
solution::f_calls;
        solution::clear_calls();

        // Fibonacci
        solution fibonacciResult = fib(lab1_fun, expansionResult[0],
expansionResult[1], epsilon, Nmax);

        SAVE_TO_FILE("fibonacci-" + std::to_string(alpha) + ".txt")
<< fibonacciResult.x << fibonacciResult.y << solution::f_calls;
        solution::clear_calls();

        // Lagrange
        solution lagrangeResult = lag(lab1_fun, expansionResult[0],
expansionResult[1], epsilon, gamma, Nmax);

        SAVE_TO_FILE("lagrange-" + std::to_string(alpha) + ".txt")
<< lagrangeResult.x << lagrangeResult.y << solution::f_calls;
        solution::clear_calls();

        delete[] expansionResult;
    }
}
```

Omówienie wyników

- Z analizy metody ekspansji dla różnych współczynników ekspansji wynika, iż dla większego współczynnika ekspansji rozmiar zakresu b-a staje się większy, a liczba wywołań funkcji celu maleje. Wynika to z tego, iż dla większego współczynnika zakres się przesuwa większymi krokami, toteż szybciej znajdzie minimum, lecz kosztem precyzji.
- Z tabel 1 i 2 również można wywnioskować, że dla wysokiego współczynnika ekspansji bardziej prawdopodobne stają się odstające wyniki. Dlatego też w tabeli dot. interpolacji Lagrange'a dla **α = 4.5** uśrednione wartości x oraz y tak bardzo różnią się od rzeczywistości - powiązane dane w tabeli 1 zawierają odstające wartości.
- Z wykresu przedziału b-a wynika, że interpolacja Lagrange'a szybciej zawęża przedział, w którym może być minimum.

Problem rzeczywisty

Cel

- Znalezienie optymalnego pola przekroju D_A , dla którego $T_B^{max} = 50℃$, z wykorzystaniem metody Fibonnaciego i Lagrange'a
- Przeprowadzenie symulacji dla uzyskanego D_A

Parametry

$$\varepsilon = 10^{-20}$$

$$\gamma = 10^{-30}$$

$$N_{max} = 10000$$

Parametry funkcji różniczkowej

- Pole podstawy zbiornika A: $P_A = 0.5\ [m^2]$
- Pole podstawy zbiornika B: $P_B = 1\ [m^2]$
- Temperatura zbiornika A: $T_A = 90\ [℃]$
- Pole przekroju otworu w zbiorniku B: $D_B = 0{,}00365665\ [m^2]$
- Temperatura wody wlewanej do B: $T_B^{in} = 20\ [℃]$
- Szybkość wlewania się wody do zbiornika B: $F_B^{in} = 0.01\ [m^3/s]$
- Współczynnik odpowiadający za lepkość cieczy: $a = 0.98$
- Współczynnik odpowiadający za zwężanie strumienia cieczy: $b = 0.63$

Kod

Funkcja różniczkowa df1

```
matrix df1(double t, matrix Y, matrix ud1, matrix ud2)
{
    matrix dY(3, 1);

    double Va = Y(0);
    double Vb = Y(1);
    double Tb = Y(2);

    double Pa = ud1(0);
    double Ta = ud1(1);
    double Pb = ud1(2);
    double Db = ud1(3);
    double F_in = ud1(4);
    double T_in = ud1(5);
    double a = ud1(6);
    double b = ud1(7);

    double FaOut = a * b * m2d(ud2) * sqrt(2 * EARTH_ACCELERATION *
(Va / Pa));
    double FbOut = a * b * Db * sqrt(2 * EARTH_ACCELERATION * (Vb /
Pb));
    double dTb_dt = FaOut / Vb * (Ta - Tb) + F_in / Vb * (T_in -
Tb);

    dY(0) = -FaOut;
    dY(1) = FaOut + F_in - FbOut;
    dY(2) = dTb_dt;

    return dY;
}
```

Równanie różniczkowe f1R

```
matrix f1R(matrix x, matrix ud1, matrix ud2) {
    matrix y(1, 1);
    matrix y0 = matrix(3, new double[3]{5, 1, 20});

    matrix *y_ptr = solve_ode(df1, 0, 1, 2000, y0, ud1, x);

    int n = get_len(y_ptr[0]);

    double max = y_ptr[1](0, 2);

    for (int i = 0; i < n; i++)
        if (y_ptr[1](i, 2) > max)
            max = y_ptr[1](i, 2);

    y(0) = abs(max - 50.0);

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

    return y;
}
```

Symulacja

```
double epsilon = 1e-20;
double gamma = 1e-30;
int Nmax = 10000;
double a = 1.0 * 0.01 * 0.01; // 1 cm2 = 0.0001 m2
double b = 100.0 * 0.01 * 0.01; // 100 cm2 = 0.01 m2

solution resultFib = fib(f1R, a, b, epsilon, ud1);
std::cout << resultFib << "\n";
solution::clear_calls();

solution resultLag = lag(f1R, a, b, epsilon, gamma, Nmax, ud1);
std::cout << resultLag << "\n";
solution::clear_calls();

matrix y(1, 1);
matrix y0 = matrix(3, new double[3] { 5.0, 1.0, 20.0 });

matrix* simulationFib = solve_ode(df1, 0, 1, 2000, y0, ud1,
resultFib.x);

SAVE_TO_FILE("wynik_proj1_fib.txt") << simulationFib[1];

matrix* simulationLag = solve_ode(df1, 0, 1, 2000, y0, ud1,
resultLag.x);

SAVE_TO_FILE("wynik_proj1_lag.txt") << simulationLag[1];
```

Omówienie wyników

Dla rzeczywistego problemu (tabela 3) interpolacja Lagrange'a osiąga wynik w znacznie mniejszą ilość wywołań. Na wykresach Tb(t), Va(t), Vb(t) wykazujących sytuację, gdy maksymalna temperatura wody w zbiorniku B wynosi 50°C.

Wnioski

Z danych wynika, że metoda Lagrange'a odnajduje minimum funkcji znacznie szybciej niż metoda Fibonacciego. Dla metody Lagrange'a przydatne jest, aby najpierw zawęzić zakres przeszukiwania za pomocą metody ekspansji, gdyż oszczędza to wiele wywołań funkcji celu.

Stosowanie zawężonego przedziału jednak nie oszczędza dużo pracy w przypadku metody Fibonacciego.