Kierunek Inżynieria Obliczeniowa	**Temat laboratoriów** Ćwiczenie 2 - Optymalizacja funkcji wielu zmiennych metodami bezgradientowymi	**Data ćwiczenia** 23.10.2024, 30.10.2024
Przedmiot Optymalizacja	Karolina Kurowska Szymon Majdak Amelia Nalborczyk	**Grupa** 2

Cel

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

Przebieg ćwiczenia

Zadanie 1 funkcja testowa celu

Na ćwiczeniach zaimplementowano dwie metody bezgradientowe:

1. Kod metody Hooke'a-Jeevesa.

```
solution HJ (matrix (*ff)(matrix, matrix, matrix), matrix x0, double s, double alpha, double epsilon, int Nmax, matrix ud1, matrix ud2) {
    try {
        solution::clear_calls ();
        solution Xopt = x0;
        solution xB;
        solution x_B;
        while (true) {
            cout << Xopt.x (0) << ";" << Xopt.x (1) << endl;
            xB = Xopt;
            xB.fit_fun (ff, ud1, ud2);
            Xopt = HJ_trial (ff, xB, s, ud1, ud2);
            if (Xopt.y < xB.y) {
                while (true) {
                    x_B = xB;
                    xB = Xopt;
                    Xopt = 2 * xB.x - x_B.x;
                    Xopt.fit_fun (ff, ud1, ud2);
                    Xopt = HJ_trial (ff, Xopt, s, ud1, ud2);
                    if (solution::f_calls > Nmax) {
                        Xopt.flag = 0;
                        cout << "Error" << endl;
                        break;
                    }
                    if (Xopt.y >= xB.y) {
                        break;
                    }
                }
                Xopt = xB;
            }
            else {
                s = alpha * s;
            }
            if (solution::f_calls > Nmax) {
                cout << "Error" << endl;
                Xopt.flag = -1;
                break;
            }
            if (s < epsilon) {
                Xopt.flag = 1;
                break;
            }
        }
        return Xopt;

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

```
281  solution HJ_trial (matrix (*ff)(matrix, matrix, matrix), solution XB, double s, matrix ud1, matrix ud2) {
282      try {
283          solution xB(XB);
284          int n = get_dim (XB);
285          matrix e = ident_mat (n);
286          for (int j = 0; j < n; j++) {
287              xB = XB.x + s * e[j];
288              xB.fit_fun (ff, ud1, ud2);
289              if (xB.y < XB.y) {
290                  XB = xB;
291              }
292              else {
293                  xB = XB.x - s * e[j];
294                  xB.fit_fun (ff, ud1, ud2);
295                  if (xB.y < XB.y) {
296                      XB = xB;
297                  }
298              }
299          }
300          return XB;
301
302      }
303      catch (std::string ex_info) {
304          throw ("solution HJ_trial(...):\n" + ex_info);
305      }
306  }
307
```

2. Kod metody Rosenbrocka.

```
solution Rosen(matrix(*ff)(matrix, matrix, matrix), matrix x0, matrix s0, double alpha, double beta, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        solution::clear_calls();
        int n = get_dim(x0);
        double max;
        matrix D = ident_mat(n);
        matrix lambda(n, 1);
        matrix p(n, 1);
        matrix s(s0);
        solution xB(x0);
        solution xB_i;
        matrix Q(n, n);
        matrix sum_sum_v(n, 1);
        bool U_turn = true;
        xB.fit_fun(ff, ud1, ud2);
        while (true) {
            //cout << /*xB.x(0) << ";" << */ xB.x(1) << endl;
            for (int j = 0; j < n; j++) {
                xB_i = xB.x + s(j) * D[j];
                xB_i.fit_fun(ff, ud1, ud2);
                if (xB_i.y < xB.y) {
                    xB = xB_i;
                    lambda(j) += s(j);
                    s(j) *= alpha;
                }
                else {
                    s(j) = s(j) * (- beta);
                    p(j)++;
                }
            }
            U_turn = true;
            for (int j = 0; j < n; j++) {
                if (lambda(j) == 0 || p(j) == 0) {
                    U_turn = false;
                    break;
                }
            }
```

```
        if (U_turn) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k <= j; k++) {
                    Q(j, k) = lambda(j);
                }
            }
            Q = Q * D;
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < j; k++) {
                    sum_sum_v = sum_sum_v + (trans(Q[j]) * D[k]) * D[k];
                }
                D.set_col((Q[j] - sum_sum_v) / norm(Q[j] - sum_sum_v), j);
                sum_sum_v = matrix(n, 1);
            }
            Q = matrix(n, n);
            s = s0;
            lambda = matrix(n, 1);
            p = matrix(n, 1);
        }
        if (solution::f_calls > Nmax) {
            cout << "Error" << endl;
            xB.flag = -1;
            return xB;
        }
        max = abs(s(0));
        for (int j = 1; j < n; ++j) {
            if (max < abs(s(j))) max = abs(s(j));
        }
        if (max < epsilon) {
            xB.flag = 1;
            break;
        }
    }
    return xB;
}
catch (string ex_info)
{
    throw ("solution Rosen(...):\n" + ex_info);
}
```

W pierwszej kolejności rozwiązujemy dwuwymiarowy problem optymalizacyjny, mający na celu znalezienie minimum. Zadanie to jest sformułowane poprzez funkcję celu daną wzorem $f(x_1, x_2) = x_1^2 + x_2^2 \ - \cos(2{,}5\pi x_1) - \cos(2{,}5\pi x_2) + 2$ zaimplementowaną jako:

```
matrix ff2T(matrix x, matrix ud1, matrix ud2)
{
    matrix y;
    y = pow(x(0), 2) + pow(x(1), 2) - cos(2.5 * 3.1415 * x(0)) - cos(2.5 * 3.1415 * x(1)) + 2;
    return y;
}
```

Część funkcji main oraz fragment funkcji lab2() odpowiedzialna za obliczanie danych do tej części zadania:

```
int main()
{
    try
    {
        lab2();
    }
    catch (string EX_INFO)
    {
        cerr << "ERROR:\n";
        cerr << EX_INFO << endl << endl;
    }
    system("pause");
    return 0;
}
```

```
// Tabele 1 i 2
ofstream file ("lab2_tab1.csv");
if (!file.is_open()) {
    cout << "file error" << endl;
    return;
}
file << "x1,x2,x1_hj,x2_hj,y_hj,f_calls_hj,min_type_hj,x1_r,x2_r,y_r,f_calls_r,min_type_r\n";

srand(time(nullptr));
double s = 0.5, alpha_HJ = 0.4, alpha_R = 1.2, beta_R = 0.3, epsilon = 0.001 , Nmax = 1000;
matrix x0(2, 1, 0.0);
matrix s0(2, 1, s);
double f_calls_HJ = 0.0, f_calls_R = 0.0;
//solution hj = HJ(ff2T, x0, s, alphaHJ, epsilon, Nmax);
for (int i = 0; i < 100; i++){
    x0(0) = (rand() % 201 - 100) * 0.01;
    x0(1) = (rand() % 201 - 100) * 0.01;
    solution solution_HJ = HJ(ff2T, x0, s, alpha_HJ, epsilon, Nmax);
    f_calls_HJ = solution::f_calls;
    solution solution_R = Rosen(ff2T, x0, s0, alpha_R, beta_R, epsilon, Nmax);
    f_calls_R = solution::f_calls;

  // Sprawdzanie czy to minimum globalne, czy lokalne
    string minimum_type_R;
    if (solution_R.y (0) < epsilon) {
        minimum_type_R = "globalne";
    }
    else {
        minimum_type_R = "lokalne";
    }
```

```
    string minimum_type_HJ;
    if (solution_HJ.y(0) < epsilon) {
        minimum_type_HJ = "globalne";
    }
    else {
        minimum_type_HJ = "lokalne";
    }

// Zapis do .csv
    file
        << x0(0) << "," << x0(1) << ","
        << solution_HJ.x(0) << ","
        << solution_HJ.x(1) << ","
        << solution_HJ.y(0) << ","
        << f_calls_HJ << ","
        << minimum_type_HJ << ","
        << solution_R.x(0) << ","
        << solution_R.x(1) << ","
        << solution_R.y(0) << ","
        << f_calls_R << ","
        << minimum_type_R << "\n";
}
file.close ();
```

Problem zostaje rozwiązany przy użyciu opisanych wcześniej metod dla losowych punktów początkowych. Dla każdego przypadku, każda z metod jest uruchamiana 100 krotnie. Rozważamy 3 przypadki, w których zmieniamy wartość długości kroku: 0.5, 0.35 i 0.2. Podczas każdego eksperymentu zapisujemy dane, takie jak: współrzędne

początkowe, współrzędne uzyskanego minimum, wartość minimum, liczba wywołań funkcji celu oraz informację, czy uzyskane minimum jest lokalne, czy globalne. Na podstawie zebranych danych obliczamy wartości średnie współrzędnych, liczby wywołań funkcji celu uwzględniając tylko minimum globalne. Wyniki tych analiz przedstawiono w Tabeli 1 i Tabeli 2, które znajdują się w pliku Excel.

Na podstawie danych widzimy dokładność wyników jest w zakresie 10^{-5} do 10^{-7}, co wskazuje na bardzo wysoką dokładność porównując tą wartość do założonej przez nas dokładności 0.001 Przy długości kroku 0,35 metoda Hooke'a-Jeevesa osiąga najniższą wartość, co sugeruje lepsze dopasowanie tego kroku dla tej metody. Metoda Rosenbrocka natomiast osiąga najlepszą wartość $y*$ dla kroku 0,5.

Analizując długość kroku i średnią rozwiązania poszczególnych metod, nie można określić konkretnej zależności. Z tego możemy wyciągnąć wniosek, że długość kroku powinna być ustalana eksperymentalnie, biorąc pod uwagę różne czynniki, takie jak treść zadania, metoda optymalizacyjna oraz parametry zadania.

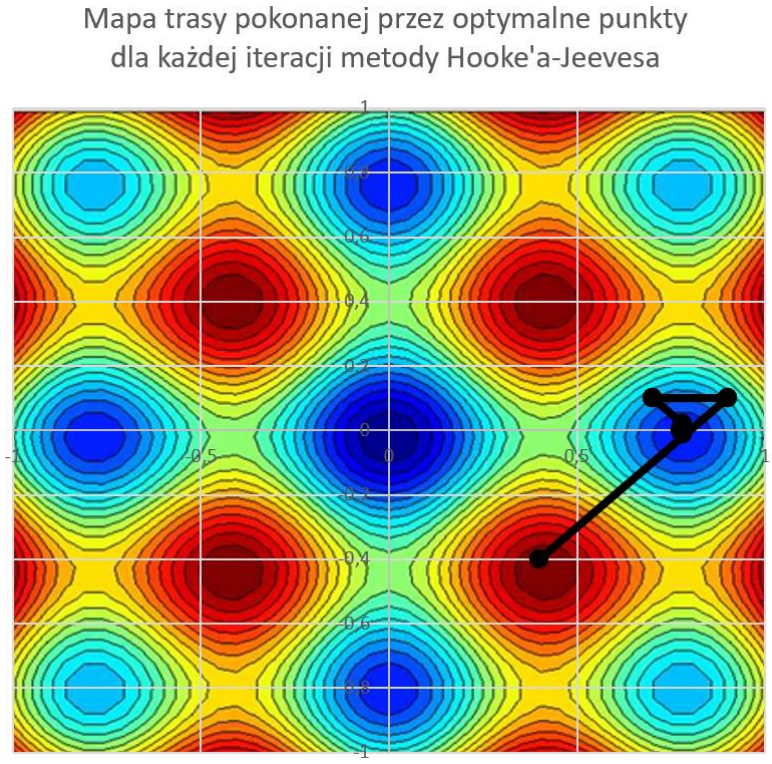
Metoda Rosenbrocka wymaga mniejszej liczby wywołań funkcji celu w porównaniu do metody Hooke'a-Jeevesa przy wszystkich długościach kroku, co świadczy o jej większej efektywności w tym zadaniu. Najmniejszą średnią liczbę wywołań dla Rosenbrocka obserwujemy to 51,24 wywołań, podczas gdy dla Hooke'a-Jeevesa ta wartość wynosi 82,14.

Porównując wyniki działania obu metod, metoda Rosenbrocka wydaje się bardziej efektywna, ponieważ osiąga porównywalne wyniki przy średnio mniejszej liczbie wywołań funkcji celu. Natomiast metoda Hooke'a-Jeevesa znalazła średnio więcej minimum globalnych, co jest pożądaną cechą.

W kolejnej części zadania obie metody uruchamiamy jednokrotnie dla losowej pary punktów oraz zapisujemy kolejne współrzędne wyznaczane przez algorytm. Realizuje to dalsza część funkcji lab2() zamieszczona poniżej.

```
  //Arkusz 3 (Wykres)
     double s = 0.5, alpha_HJ = 0.4, alpha_R = 1.2, beta_R = 0.3, epsilon = 0.001 , Nmax = 1000;
matrix x0(2, 1, 0.0);
x0(0) = 0.4;
x0(1) = -0.4;
matrix s0(2, 1, s);
solution solution_HJ = HJ(ff2T, x0, s, alpha_HJ, epsilon, Nmax);
solution solution_R = Rosen(ff2T, x0, s0, alpha_R, beta_R, epsilon, Nmax);
```

W kolejnym etapie dla metody Hooke'a-Jeevesa na wykresie poziomic funkcji celu przedstawiamy rozwiązania optymalne, uzyskane kolejno po każdej iteracji, pokazujące przebieg optymalizacji. Otrzymane rezultaty znajdują się w zakładce "Wykres" w arkuszu Excel oraz poniżej jako zrzut ekranu:



Wykres pokazuje, jak metoda Hooke'a-Jeevesa wyznacza kolejne punktu w procesie wyszukiwania minimum. W przeprowadzonym wywołaniu metoda nie znajduje minimum globalnego, które analitycznie wyznaczone znajduje się w pkt. (0,0), a znajduje minimum lokalne.

Zadanie 2 problem rzeczywisty

Kolejnym zadaniem jest przeprowadzenie symulacji. Problem polega na optymalizacji sterowania ramieniem robota, które ma umieścić ciężarek na platformie poprzez obrót o kąt π radianów i zatrzymanie się. Optymalizacja ma znaleźć takie wartości współczynników k1 i k2 regulatora, aby zminimalizować funkcjonał jakości Q(k1, k2), który ocenia, jak dokładnie i efektywnie ramię osiąga żądany kąt oraz prędkość.

W celu rozwiązania tego problemu stworzyliśmy funkcje odwzorowujące dane równania różniczkowe, które zostały zapisane w pliku user_funs.cpp. Funkcje te zamieszczamy poniżej:

```cpp
// Moment bezwładności ramienia
double mom_of_i() {
    const double l = 1.0;      // Długość ramienia [m]
    const double m_r = 1.0;     // Masa ramienia [kg]
    const double m_c = 5.0;     // Masa ciężarka [kg]
    return (1.0 / 3.0) * m_r * l * l + m_c * l * l;
}

// Funkcja celu
matrix ff2R (matrix x, matrix ud1, matrix ud2) {
    matrix y = 0;
    matrix Y_start (2, 1);
    matrix Y_ref (2, new double[2] {M_PI, 0});
    matrix* Y = solve_ode(df2, 0, 0.1, 100, Y_start, Y_ref, x);

    int* size = get_size(Y[0]);
    int rows = size[0]; // Liczba wierszy

    for (int i = 0; i < rows; ++i) {
        //oblcizmay Q value
        double M = x (0) * (Y_ref (0) - Y[1] (i, 0)) + x (1) * (Y_ref (1) - Y[1] (i, 1));
        double alpha_dif = Y_ref (0) - Y[1] (i, 0);
        double omega_dif = Y_ref (1) - Y[1] (i, 1);

        y = y + (10 * pow(alpha_dif, 2) + pow(omega_dif, 2) + pow(M, 2));
    }

    y = y * 0.1;
    Y[0].~matrix();
    Y[1].~matrix();
    return y;
}

// Równanie różniczkowe
matrix df2 (double t, matrix Y, matrix ud1 /*k1*/, matrix ud2 /*k2*/) {
    double I = mom_of_i(); // Moment bezwładności
    double b = 0.5;        //Współczynnik tarcia
    double alpha = Y(0);   // Aktualny kąt
    double omega = Y(1);   // Aktualna prędkość kątowa

    matrix dY (2, 1);
    dY(0) = omega;  // dα/dt = ω
    dY(1) = (ud2(0) * (ud1(0) - alpha) + ud2(1) * (ud1(1) - omega) - b * omega) / I; // dω/dt = (M - bω) / I

    return dY;
}
```

Funkcja mom_of_i oblicza moment bezwładności ramienia (I), zgodnie z wzorem podanym w konspekcie: $I = \frac{1}{3} m_r l^2 + m_c l^2$.

Funkcja ff2R oblicza wartość funkcji celu, która służy do oceny jakości parametrów (k_1) oraz (k_2) regulatora. Początkowo funkcja wywołuje solve_ode, która korzysta z funkcji df2 do symulacji ruchu ramienia w czasie, rozwiązując równania różniczkowe dla układu dynamicznego. Następnie po rozwiązaniu równań funkcja oblicza moment siły z wzoru podanego w konspekcie: $M(t) = k_1\left(\alpha_{ref} - \alpha(t)\right) + k_2\left(\omega_{ref} - \omega(t)\right)$. Następnie obliczamy funkcjonał jakości metodą prostokątów znów bazując na równaniach dostarczonych w konspekcie:

$$Q(k_1, k_2) = \int_0^{t_{end}} \left(10 \cdot \left(\alpha_{ref} - \alpha(t)\right)^2 + \left(\omega_{ref} - \omega(t)\right)^2 + \left(M(t)\right)^2\right) dt$$

Po zakończeniu pętli, wartość funkcji celu jest pomnożona przez 0.1, aby uwzględnić krok czasowy w całkowaniu.

Funkcje są wywoływane za pomocą odpowiednich instrukcji w kodzie. Część kodu funkcji lab2() odpowiedzialna za tworzenie symulacji:

```
double s = 0.5, alpha_HJ = 0.4, alpha_R = 1.2, beta_R = 0.3, epsilon = 0.001, Nmax = 1000;
matrix x0(2, 1, 5.0);
matrix s0(2, 1, s);
double t_start = 0.0, dt = 0.1, t_end = 100.0;

// PROBLEM RZECZYWISTY

// Testowanie ff2R
cout << "Problem rzeczywisty:" << endl;
cout << "ff2R tostowe: " << ff2R(x0) << endl << endl;

// Szukanie optymalnych k1 i k2 i wypisanie (Arkusz 4)
solution solution_HJ = HJ(ff2R, x0, s, alpha_HJ, epsilon, Nmax);
cout << "HJ: " << endl << solution_HJ << endl;

solution solution_R = Rosen(ff2R, x0, s0, alpha_R, beta_R, epsilon, Nmax);
cout << "Rosen: " << endl << solution_R << endl;

// Symulacja dla znalezionych k1 i k2
matrix Y_start(2, 1);
matrix Y_ref(2, new double[2] {3.14, 0});

matrix* solution_result_HJ = solve_ode(df2, t_start, dt, t_end, Y_start, Y_ref, solution_HJ.x);
matrix* solution_result_R = solve_ode(df2, t_start, dt, t_end, Y_start, Y_ref, solution_R.x);


// Zapis do .csv (Arkusz 5)
ofstream file_Sym("symulacja.csv");
if (!file_Sym.is_open()) {
    cout << "file error" << endl;
    return;
}
file_Sym << "t,alfa_HJ,omega_HJ,alfa_Rosen,omega_Rosen\n";
for (int i = 0; i <= 1000; ++i) {
    file_Sym << solution_result_HJ[0](i) << "," << solution_result_HJ[1](i, 0) << "," << solution_result_HJ[1](i, 1) << ","
        << solution_result_R[1](i, 0) << "," << solution_result_R[1](i, 1) << "\n";
}
file_Sym.close();
```

Wyniki symulacji zostały zamieszczone w zakładkach "Tabela 3" oraz "Symulacja" w pliku Excel, który dołączono do sprawozdania. W Tabeli 3 zamieszczamy wyniki symulacji optymalizacja dla długości kroku 0.5. W arkuszu “Symulacja” zamieszczamy symulacje problemu rzeczywistego dla obliczonych optymalnych współczynników k1 i k2 oraz 2 wykresy.

Analizując wyniki optymalizacji w Tabeli 3 możemy zauważyć, że metoda Rosenbrocka jest bardziej wydajna pod względem liczby wywołań funkcji celu, osiągając tę samą jakość rozwiązania co metoda Hooke’a-Jeevesa, ale przy mniejszym nakładzie obliczeń. Obie metody dają zbliżone wyniki parametrów k1 i k2.

Pierwszy wykres przedstawia położenie ramienia w czasie dla poszczególnych algorytmów. Drugi wykres przedstawia prędkość kątową ramienia w czasie dla poszczególnych algorytmów optymalizacyjnych. Wykresy zamieszczone są poniżej:

Na pierwszym wykresie na początku widoczny jest szybki wzrost kąta obrotu, z lekkim przeregulowaniem, zanim robot ustabilizuje się wokół wartości docelowej. Widać, że obie metody doprowadzają ramię robota do stabilizacji w podobnym czasie.

Wykres drugi przedstawia zmiany prędkości kątowej ramienia w czasie. Na początku prędkość gwałtownie rośnie, osiągając szczyt, po czym zmniejsza się do zera w miarę, jak ramię osiąga wartość docelowego kąta.

Wnioski

Celem ćwiczenia było wdrożenie metod bezgradientowych, tj. Hooke’a-Jeevesa i Rosenbrocka, oraz ich zastosowanie do optymalizacji funkcji celu i sterowania ramieniem robota.

W podsumowaniu można stwierdzić, że metoda Rosenbrocka wypadła lepiej niż metoda Hooke’a-Jeevesa. W obu częściach zadania – zarówno w optymalizacji funkcji celu, jak i w optymalizacji sterowania ramieniem robota – metoda Rosenbrocka uzyskała porównywalną jakość rozwiązania, jednocześnie wymagając mniejszej liczby wywołań funkcji celu. To oznacza, że była bardziej efektywna obliczeniowo, co może mieć znaczenie przy większych lub bardziej złożonych problemach optymalizacyjnych.

Wyniki potwierdzają, że metody bezgradientowe są użytecznymi narzędziami do optymalizacji zadań, gdzie brak jest łatwego dostępu do gradientu, a dobór odpowiednich parametrów oraz metod jest kluczowy dla osiągnięcia efektywnych wyników.