Marcin Brzózka gr. 1

Nr. albumu: 408932

Laboratorium 1, temat:

Optymalizacja funkcji jednej zmiennej metodami bezgradientowymi

Cel ćwiczenia:

Celem ćwiczenia jest zapoznanie się z metodami bezgradientowymi poprzez ich implementację oraz wykorzystanie do rozwiązania jednowymiarowego problemu optymalizacji. Do wykonania zadania wykorzystujemy metody Ekspansji, Fibonacciego oraz metody opartej na interpolacji Lagrange'a.

Wykonanie ćwiczenia:

Pierwszym etapem wykonywania ćwiczenia było uzupełnienie kodu funkcji potrzebnych do obliczania wartości za pomocą pseudokodu dostępnego w instrukcji do ćwiczenia. Następnym krokiem po uzupełnieniu kodu funkcji było uzupełnienie danych z 300 wywołań programu. Dane te były ułożone po 100 dla trzech różnych współczynników ekspansji. W tabeli pierwszej wypisywaliśmy współczynnik ekspansji, w moim przypadku przyjąłem: „1.7” , „2.5” , „3.6”. Następnie uzupełnialiśmy dane związane z metodą Ekspansji, Fibonacciego oraz metody opartej na interpolacji Lagrange'a: punkt startowy x0, który generował się losowo i zawierał się w przedziale od -100 do 100, przedział poszukiwań a, b w metodzie ekspansji oraz wyniki generowane za pomocą wyżej wymienionych metod. Zebranie takiej ilości danych pozwoliło obliczyć średnie w tabeli drugiej i wyciągnąć za ich pomocą pierwsze wnioski.

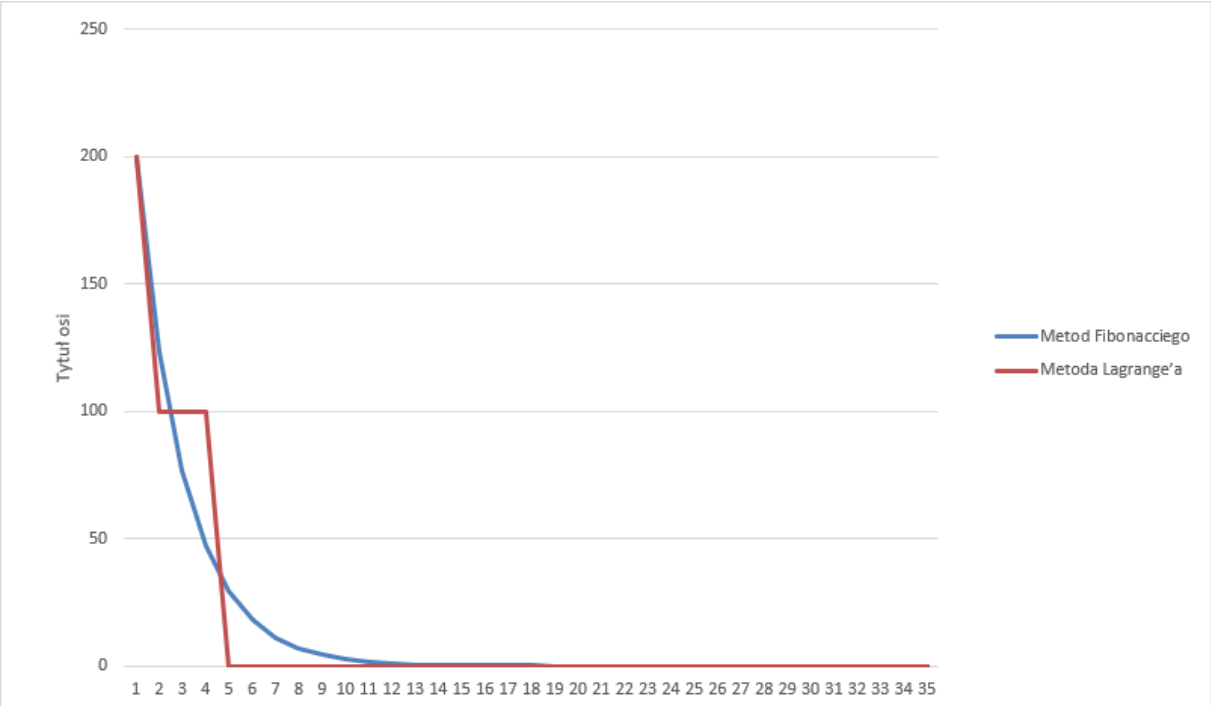
Tabela 2:

Współczynnik ekspansji	Metoda ekspansji		Rodzaj minimum	Metoda Fibonacciego				Metoda oparta na inperpolacji Lagrange'a			
	b-a	Liczba wywołań funkcji celu		x*	y*	Liczba wywołań funkcji celu	Liczba wystąpień	x*	y*	Liczba wywołań funkcji celu	Liczba wystąpień
			Minimum globalne	62,7482	-0,921148	62,8	30	62,74820769	-0,921148	31,69230769	26
			Minimum lokalne	2,99E-07	2,67E-17	64,88571429	70	1,78E-13	-7,16E-18	23,2173913	74
1.7	40,65109724	8,54	Brak zbieżności							4	4
			Minimum globalne	62,7482	-0,921148	65,125	32	62,74819643	-0,921148	34	28
			Minimum lokalne	-3,13E-08	2,60E-17	68	68	2,16E-13	-7,16E-18	26,01351351	72
2.5	76,34	6,72	Brak zbieżności							3,5	6
			Minimum globalne	62,7482	-0,921148	64,12903226	31	62,74821818	-0,921148	35,27272727	22
			Minimum lokalne	8,96E-01	2,37E-17	68,46376812	69	1,82E-13	-7,16E-18	26,73239437	78
3,6	86,29	5,68	Brak zbieżności							3,285714286	7

Tabela ta pokazuję zależność pomiędzy wybranymi współczynnikami ekspansji a zakresem obliczanym przez metodę ekspansji. Wraz ze wzrostem wartości współczynnika wzrasta również wartość b-a. Powoduję to zmniejszenie się dokładności i jakości wyników z danych metod. Analizując średnie wartości metody Fibonacciego oraz metody opartej na interpolacji Lagrange'a możemy dojść do wniosków, że wraz z wzrostem współczynnika wzrasta też możliwość występowania błędów oraz to że pierwsza z nich jest wolniejsza natomiast częściej oblicza minimum globalne.

Do wykonania pierwszego wykresu zmodyfikowałem kod metod Fibonacciego oraz metody opartej na interpolacji Lagrange'a w taki sposób aby podczas każdej iteracji wypisywany był zakres poszukiwań minimum z przedziału od -100 do 100. W przypadku pierwszej z metod ilość iteracji podczas których wypisywany był zakres okazała się znacznie większa niż w drugiej w nich.

Wykres ukazujący długość przedziału [a,b] dla obu funkcji:



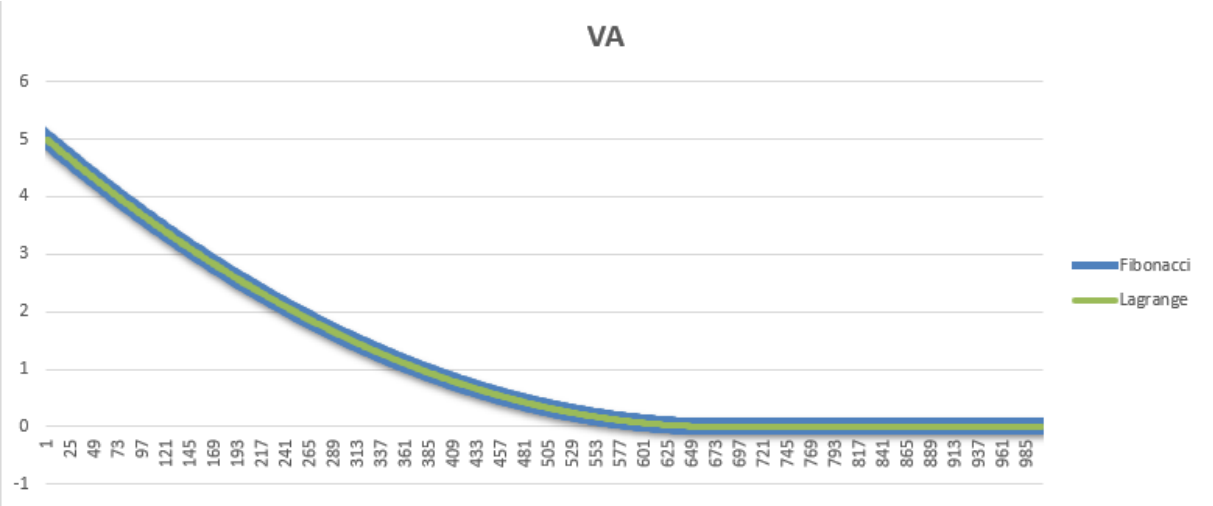
Kolejnym krokiem naszego ćwiczenia będzie rozwiązanie problemu rzeczywistego opisanego w instrukcji do zadania. Obie metody wygenerowały takie same wyniki.

Tabela 3:

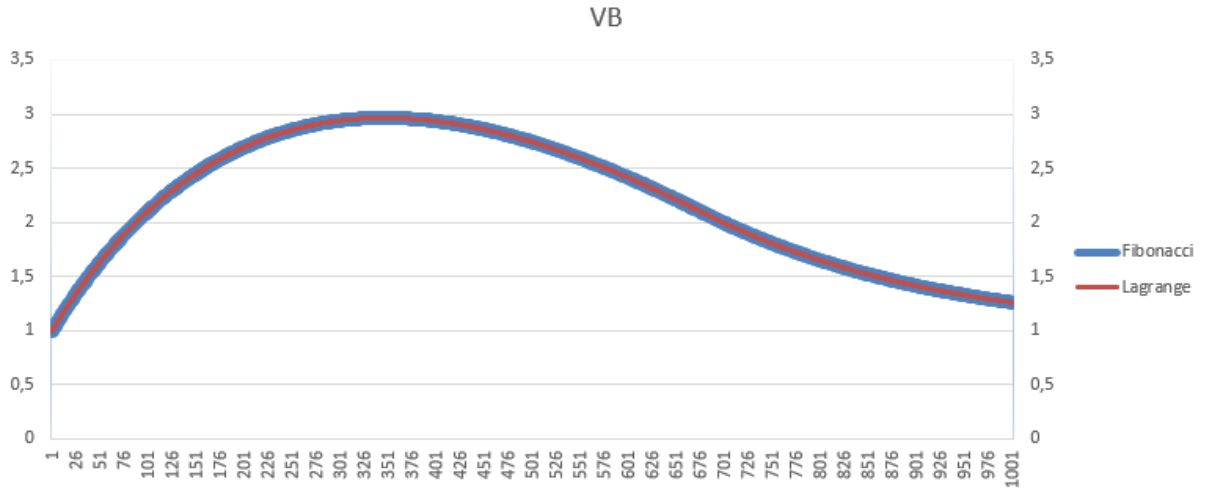
Metoda Fibonacciego			Metoda oparta na inperpolacji Lagrange'a		
DA*	y*	Liczba wywołań funkcji celu	DA*	y*	Liczba wywołań funkcji celu
0,00241201	1,33E-07	78	0,00241201	9,31E-08	120

Wykresy uzyskane za pomocą symulacji obliczania optymalnego pola przekroju otworu w zbiorniku A.

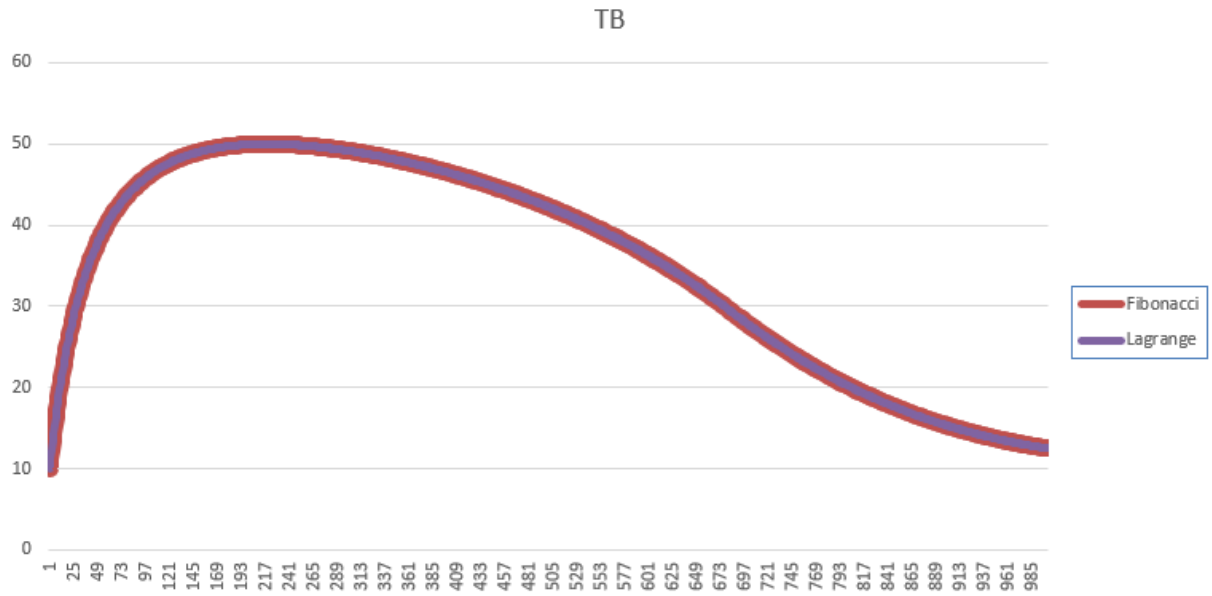
Wykres przedstawiający objętość wody w zbiorniku A:



Wykres przedstawiający objętość wody w zbiorniku B:



Wykres przedstawiający temperaturę wody w zbiorniku B:



Kod:

```
void lab1()
{
    ////Funkcja testowa
    double x0 = m2d(200 * rand_mat() - 100), d = 1, alpha = 2.8, epsilon = 1e-5, gamma = 1e-200;
    //x0 = 15;
    int Nmax = 1000;
    //double* ab;
    solution opt;
    //cout << x0 << endl << endl;
    //ab = expansion(ff1T, x0, d, alpha, Nmax);
    //cout << ab[0] << '\t' << ab[1] << endl << endl;
    //solution::clear_calls();
    //opt = fib(ff1T, ab[0], ab[1], epsilon);
    //opt = fib(ff1T, -100, 100, epsilon);
    //cout << opt << endl << endl;
    //solution::clear_calls();
    //opt = lag(ff1T, ab[0], ab[1], epsilon, gamma, Nmax);
    //opt = lag(ff1T, -100, 100, epsilon, gamma, Nmax);
    //cout << opt << endl;
    //solution::clear_calls();

    //Zbiorniki
    epsilon = 1e-10;
    opt = fib(ff1R, 1e-4, 1e-2, epsilon);
    int n = get_len(symulation[0]);
    for (int i = 0; i < n; ++i)
        cout << symulation[0](i, 0) << ";" << symulation[1](i, 0) << ";" << symulation[1](i, 1) << ";" << symulation[1](i, 2) << endl;
    cout << opt << endl << endl;
    solution::clear_calls();
    opt = lag(ff1R, 1e-4, 1e-2, epsilon, gamma, Nmax);
    n = get_len(symulation[0]);
    for (int i = 0; i < n; ++i)
        cout << symulation[0](i, 0) << ";" << symulation[1](i, 0) << ";" << symulation[1](i, 1) << ";" << symulation[1](i, 2) << endl;
    cout << opt << endl << endl;
    solution::clear_calls();
}
```

```
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 X0(x0), X1(x0 + d);
        X0.fit_fun(ff, ud1, ud2);
        X1.fit_fun(ff, ud1, ud2);
        if (X0.y == X1.y)
        {
            p[0] = m2d(X0.x);
            p[1] = m2d(X1.x);
            return p;
        }
        if (X0.y < X1.y)
        {
            d = -d;
            X1.x = X0.x + d;
            X1.fit_fun(ff, ud1, ud2);
            if (X1.y >= X0.y)
            {
                p[0] = m2d(X1.x);
                p[1] = m2d(X0.x - d);
                return p;
            }
        }
        solution X2;
        while (true)
        {
            ++i;
            X2.x = X0.x + pow(alpha, i) * d;
            X2.fit_fun(ff, ud1, ud2);
            if (X2.y > X1.y || solution::f_calls > Nmax)
                break;
            X0 = X1;
            X1 = X2;
        }
        if (d > 0)
        {
            p[0] = m2d(X0.x);
            p[1] = m2d(X2.x);
        }
        else {
            p[0] = m2d(X2.x);
            p[1] = m2d(X0.x);
        }
        cout << solution::f_calls << endl;
        return p;
    }
    catch (string ex_info)
    {
        throw ("double* expansion(...):\n" + ex_info);
    }
}
```

```
solution fib(matrix(*ff)(matrix, matrix, matrix), double a, double b, double epsilon, matrix ud1, matrix ud2)
{
    try
    {
        solution Xopt;
        Xopt.ud = b - a;
        int n = (int)(ceil(log2(sqrt(5) * (b - a) / epsilon) / log2((1 + sqrt(5)) / 2)));
        int* F = new int[n] {1, 1};
        for (int i = 2; i < n; ++i)
            F[i] = F[i - 2] + F[i - 1];
        solution A(a), B(b), C, D;
        C.x = B.x - 1.0 * F[n - 2] / F[n - 1] * (B.x - A.x);
        D.x = A.x + B.x - C.x;
        C.fit_fun(ff, ud1, ud2);
        D.fit_fun(ff, ud1, ud2);
        for (int i = 0; i <= n - 3; ++i)
        {
            cout << "a: " << A.x << endl;
            cout << "b: " << B.x << endl;
            if (C.y < D.y)
                B.x(0, 0) = D.x(0, 0);
            else
                A.x(0, 0) = C.x(0, 0);
            C.x = B.x - 1.0 * F[n - i - 2] / F[n - i - 1] * (B.x - A.x);
            D.x = A.x + B.x - C.x;
            C.fit_fun(ff, ud1, ud2);
            D.fit_fun(ff, ud1, ud2);

            Xopt.ud.add_row((B.x - A.x)());
        }
        Xopt = C;
        Xopt.flag = 0;
        return Xopt;
    }
    catch (string ex_info)
    {
        throw ("solution fib(...):\n" + ex_info);
    }
}
```

```
solution lag(matrix(*ff)(matrix, matrix, matrix), double a, double b, double epsilon, double gamma, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        solution Xopt;
        Xopt.ud = b - a;
        solution A(a), B(b), C, D, D_old(a);
        C.x = (b + a) / 2;
        double l, m;
        while (true)
        {
            cout << "a: " << A.x << endl;
            cout << "b: " << B.x << endl;
            A.fit_fun(ff, ud1, ud2);
            B.fit_fun(ff, ud1, ud2);
            C.fit_fun(ff, ud1, ud2);
            l = m2d(A.y * (pow(B.x, 2) - pow(C.x, 2)) + B.y * (pow(C.x, 2) - pow(A.x, 2)) + C.y * (pow(A.x, 2) - pow(B.x, 2)));
            m = m2d(A.y * (B.x - C.x) + B.y * (C.x - A.x) + C.y * (A.x - B.x));
            if (m <= 0)
            {
                Xopt = D_old;
                Xopt.flag = 2;
                return Xopt;
            }
            D.x = 0.5 * l / m;
            D.fit_fun(ff, ud1, ud2);
            if (A.x < D.x && D.x < C.x)
            {
                if (D.y < C.y)
                {
                    B.x = C.x;
                    C.x = D.x;
                }
                else {
                    A.x = D.x;
                }
            }
            else if (C.x < D.x && D.x < B.x)
            {
                if (D.y < C.y)
                {
                    A.x = C.x;
                    C.x = D.x;
                }
                else {
                    B.x = D.x;
                }
            }
            else
            {
                Xopt = D_old;
                Xopt.flag = 2;
                return Xopt;
            }
```

```
            Xopt.ud.add_row((B.x - A.x)());

            if (B.x - A.x < epsilon || abs(m2d(D.x) - m2d(D_old.x)) < gamma)
            {
                Xopt = D;
                Xopt.flag = 0;
                return Xopt;
            }
            if (solution::f_calls > Nmax)
            {
                Xopt = D;
                Xopt.flag = 1;
                break;
            }
            D_old = D;
        }
        return Xopt;
    }
    catch (string ex_info)
    {
        throw ("solution lag(...):\n" + ex_info);
    }
}
```

```
matrix ff1T(matrix x, matrix ud1, matrix ud2)
{
    matrix y;
    double& pomx = x(0, 0);
    double exponent = 0.1 * pomx - 2.0 * M_PI;
    exponent *= exponent;
    y = -cos(0.1 * pomx) * exp(-exponent) + 0.002 * pow(0.1 * pomx, 2.0);
    return y;
}

matrix* symulation;

matrix ff1R(matrix x, matrix ud1, matrix ud2)
{
    matrix y;
    matrix Y0 = matrix(3, new double[3]{ 5, 1, 10 });
    matrix* Y = solve_ode(df1, 0, 1, 1000, Y0, ud1, x);
    int n = get_len(Y[0]);
    double max = Y[1](0, 2);
    for (int i = 1; i < n; ++i)
        if (max < Y[1](i, 2))
            max = Y[1](i, 2);
    y = abs(max - 50);
    symulation = Y;
    return y;
}
```

Wnioski:

Podsumowując, wyniki pokazują że metoda Fibonacciego może być bardziej precyzyjna, natomiast wymaga ona większej ilości czasu obliczeniowego, podczas gdy metoda oparta na interpolacji Lagrange'a może wywoływać sporadyczne braki zbieżności za to do obliczeń potrzebuje ona mniejszej ilości czasu. Podczas przeprowadzania symulacji obie z metod wykazały bardzo zbliżone do siebie końcowe rezultaty podczas gdy metoda oparta na interpolacji Lagrange'a potrzebowała większej ilości wywołań funkcji celu. Wykorzystanie obu metod przy określonym przedziale czasowym ustalonym na (1-1000 sekund) wykazało bardzo duże podobieństwo wyników uzyskanych podczas obliczeń czego potwierdzeniem są wykresy przedstawione powyżej. Mimo drobnych różnic w wynikach, obie z przedstawionych metod są zgodne w rozwiązywaniu danego problemu optymalizacyjnego.