Kierunek Inżynieria Obliczeniowa	**Temat laboratoriów** Ćwiczenie 1 - Optymalizacja funkcji jednej zmiennej metodami bezgradientowymi	**Data ćwiczenia** 09.10.2024, 16.10.2024
Przedmiot Optymalizacja	Karolina Kurowska Szymon Majdak Amelia Nalborczyk	**Grupa** 2

Cel

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

Przebieg ćwiczenia

Zadanie 1 funkcja testowa celu

Na ćwiczeniach zaimplementowano 3 metody bez gradientowe:

1. Kod metody ekspansji

```
33
34  double* expansion(matrix(*ff)(matrix, matrix, matrix), double x0, double d, double alpha, int Nmax, matrix ud1, matrix ud2){
35      try{
36          double* p = new double[2]{0,0}; //tam bedzie nasz przedzial
37
38          //Tu wpisz kod funkcji
39          solution X_0(x0), X_1(x0 + d);
40          X_0.fit_fun(ff, ud1, ud2);
41          X_1.fit_fun(ff, ud1, ud2);
42          if(X_0.y == X_1.y){
43              p[0] = m2d(X_0.x);
44              p[1] = m2d(X_1.x);
45              return p;
46          }
47
48          if(X_1.y > X_0.y){
49              d = -d;
50              X_1.x = X_0.x + d;
51              X_1.fit_fun(ff, ud1, ud2);
52              if (X_1.y>=X_0.y){
53                  p[0]=m2d(X_1.x);
54                  p[1]=m2d(X_0.x-d);
55                  return p;
56              }
57          }
58
59          int i = 0;
60          solution X_i = X_0;
61          while(true){
62              if(X_0.f_calls > Nmax){
63                  return NULL;
64              }
65              i = i+1;
66              if(i>1){
67                  X_i=X_1;
68              }
69              X_1.x = X_0.x + pow(alpha, i)*d;
70              X_1.fit_fun(ff, ud1, ud2);
71              X_i.fit_fun(ff,ud1,ud2);
72              if (X_i.y<=X_1.y){
73                  break;
74              }
75          }
76
77          solution X_im;
78          X_im.x = X_0.x + pow(alpha, i-2)*d;
79
80          if (d>0){
81              p[0] =m2d(X_im.x);
82              p[1] = m2d(X_1.x);
83              return p;
84          }
85
86          p[0]= m2d(X_1.x);
87          p[1] = m2d(X_im.x);
88          return p;
89      }
90
91      catch (string ex_info){
92          throw ("double* expansion(...):\n" + ex_info);
93      }
94  }
```

2. Kod metody Fibonacciego

```
solution fib(matrix(*ff)(matrix, matrix, matrix), double a, double b, double epsilon, matrix ud1, matrix ud2){
    try{
        solution Xopt;
        solution a_0(a), b_0(b), c_0, d_0;
        int k = 0;

        while(fibo_help(k) <= (b-a)/epsilon){
            k++;
        }

        c_0.x = b_0.x - ((fibo_help(k-1)/(fibo_help(k)))*(b_0.x-a_0.x));
        d_0.x = a_0.x + b_0.x - c_0.x;

        for(int i = 0; i<k-3;i++){
            c_0.fit_fun(ff,ud1,ud2);
            d_0.fit_fun(ff,ud1,ud2);

            if(c_0.y<d_0.y){
                a_0.x = a_0.x;
                b_0.x = d_0.x;
            }

            else{
                b_0.x = b_0.x;
                a_0.x = c_0.x;
            }

            c_0.x = b_0.x - ((fibo_help(k-i-2)/(fibo_help(k-i-1)))*(b_0.x-a_0.x));
            d_0.x = a_0.x + b_0.x - c_0.x;
            cout << " a_0: " << a_0.x << "\t";
            cout << " b_0: " << b_0.x << "\t";
            cout << " c_0: " << c_0.x << "\t";
            cout << " d_0: " << d_0.x << "\t";
            cout << endl;

        }

        Xopt = c_0;
        Xopt.fit_fun(ff, ud1, ud2);
        return Xopt;
    }

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

```
double fibo_help(int k){
    double x = 0;
    double y = 1;

    for(int i = 0; i < k; i++){
        y = y + x;
        x = y - x;
    }
    return y;
}
```

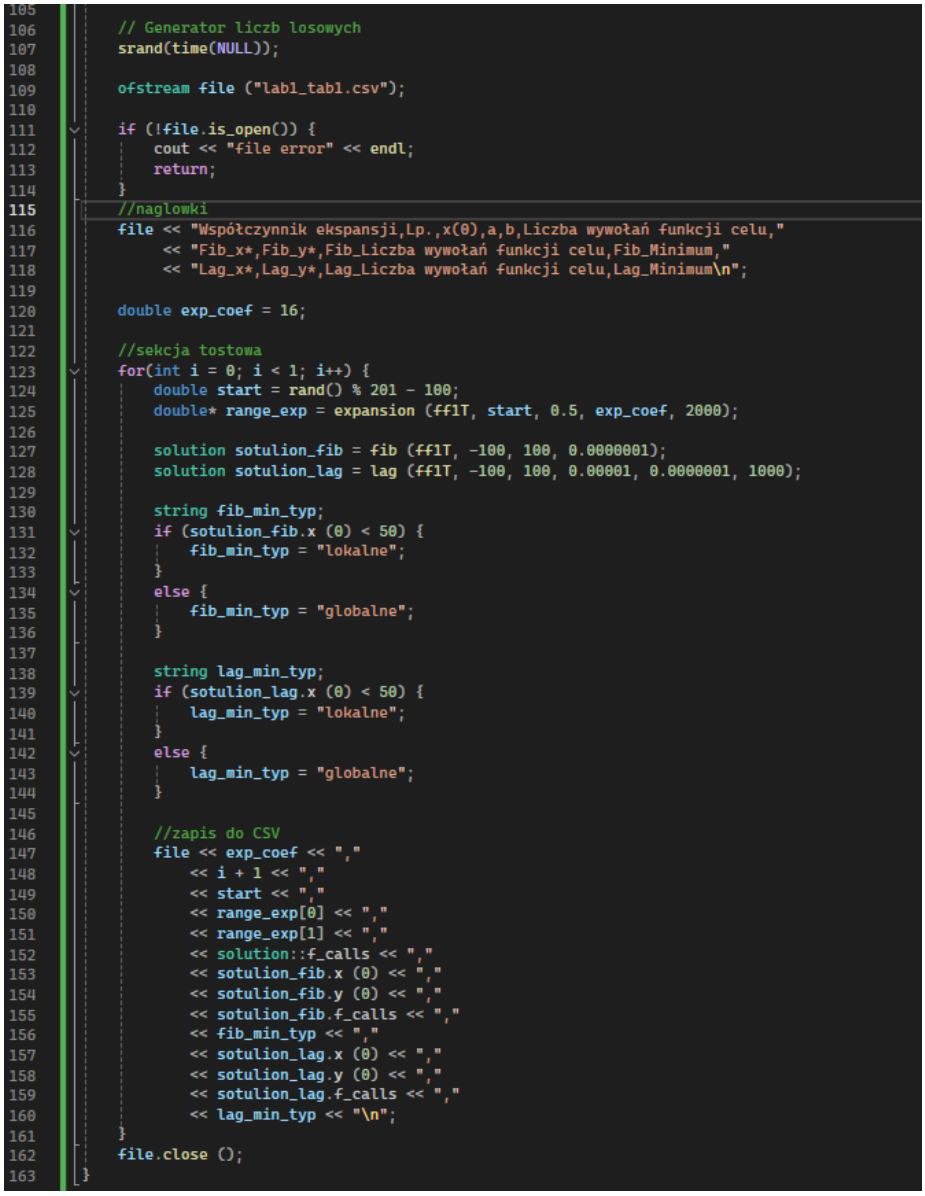
3. Kod metody Lagrange’a

```
solution lag (matrix (*ff)(matrix, matrix, matrix), double a, double b, double epsilon, double gamma, int Nmax, matrix ud1, matrix ud2) {
    try {
        solution::clear_calls ();
        solution A_i = a, B_i = b, C_i = (a + b) / 2,
            Xopt = C_i, D_i = 0;
        int i = 0;
        double l, m;
        cout << " a_0: " << A_i.x << "\t";
        cout << " b_0: " << B_i.x << "\t";
        cout << " c_0: " << C_i.x << "\t";
        cout << " d_0: " << D_i.x << "\t";
        cout << endl;


        do {
            A_i.fit_fun (ff, ud1, ud2); B_i.fit_fun (ff, ud1, ud2); C_i.fit_fun (ff, ud1, ud2);
            l = A_i.y(0) * (pow (B_i.x(0), 2) - pow(C_i.x (0), 2))
                + B_i.y(0) * (pow (C_i.x(0), 2) - pow (A_i.x(0), 2))
                + C_i.y(0) * (pow (A_i.x(0), 2) - pow(B_i.x(0), 2));
            m = A_i.y(0) * (B_i.x(0) - C_i.x(0))
                + B_i.y(0) * (C_i.x(0) - A_i.x(0))
                + C_i.y(0) * (A_i.x(0) - B_i.x(0));
            if (m <= 0) {
                Xopt.flag = 0;
                break;
            }
            D_i = l/2/m;
            D_i.fit_fun (ff, ud1, ud2);

            if (A_i.x(0) < D_i.x(0) && D_i.x(0) < C_i.x(0)){
                if (D_i.y(0) < C_i.y(0)) {
                    B_i = C_i;
                    C_i = D_i;
                }
                else A_i = D_i;
            }

            else{
                if (C_i.x(0) < D_i.x(0) && D_i.x(0) < B_i.x(0)) {
                    if (D_i.y(0) < C_i.y(0)) {
                        A_i = C_i;
                        C_i = D_i;
                    }
                    else B_i = D_i;
                }
                else{
                    Xopt.flag = 0;
                    break;
                }
            }

            if (Xopt.f_calls > Nmax){
                Xopt.flag = 0;
                break;
            }

            Xopt = D_i;
            cout << " a_0: " << A_i.x << "\t";
            cout << " b_0: " << B_i.x << "\t";
            cout << " c_0: " << C_i.x << "\t";
            cout << " d_0: " << D_i.x << "\t";
            cout << endl;
        } while (B_i.x(0) - A_i.x(0) >= epsilon || abs(D_i.x(0) - Xopt.x(0)) >= gamma);

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

W pierwszej kolejności rozwiązujemy jednowymiarowy problem optymalizacyjny, mający na celu znalezienie minimum. Zadanie to jest sformułowane poprzez następującą funkcję celu:

```
matrix ff1T (matrix x, matrix ud1, matrix ud2) {
    double arg = m2d (x);
    double result = -cos (0.1 * arg) * exp (-pow (0.1 * arg - 2 * 3.14, 2)) + 0.002 * pow (0.1 * arg, 2);
    matrix res = matrix (result);
    return res;
}
```

Część main odpowiedzialna za obliczanie danych do tej części zadania:



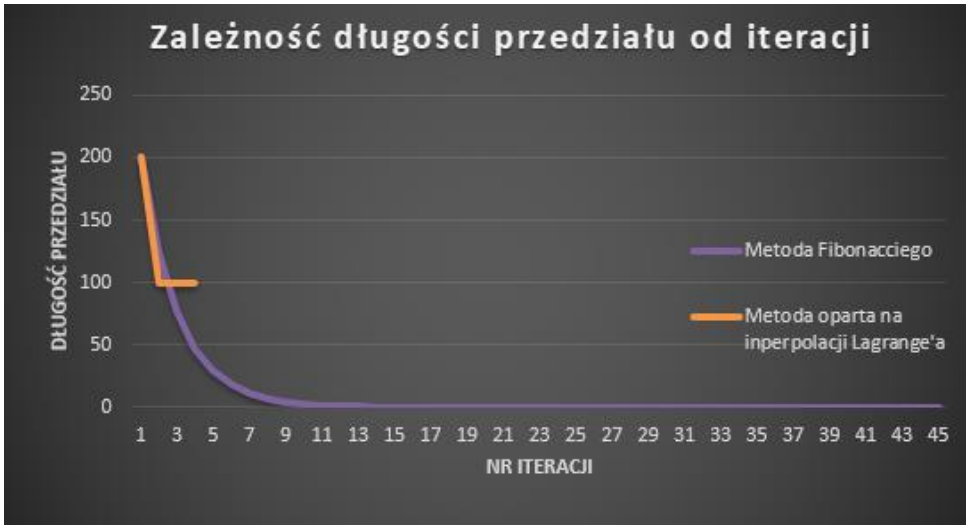
```cpp
    // Generator liczb losowych
    srand(time(NULL));

    ofstream file ("lab1_tab1.csv");

    if (!file.is_open()) {
        cout << "file error" << endl;
        return;
    }
    //naglowki
    file << "Współczynnik ekspansji,Lp.,x(0),a,b,Liczba wywołań funkcji celu,"
         << "Fib_x*,Fib_y*,Fib_Liczba wywołań funkcji celu,Fib_Minimum,"
         << "Lag_x*,Lag_y*,Lag_Liczba wywołań funkcji celu,Lag_Minimum\n";

    double exp_coef = 16;

    //sekcja tostowa
    for(int i = 0; i < 1; i++) {
        double start = rand() % 201 - 100;
        double* range_exp = expansion (ff1T, start, 0.5, exp_coef, 2000);

        solution sotulion_fib = fib (ff1T, -100, 100, 0.0000001);
        solution sotulion_lag = lag (ff1T, -100, 100, 0.00001, 0.0000001, 1000);

        string fib_min_typ;
        if (sotulion_fib.x (0) < 50) {
            fib_min_typ = "lokalne";
        }
        else {
            fib_min_typ = "globalne";
        }

        string lag_min_typ;
        if (sotulion_lag.x (0) < 50) {
            lag_min_typ = "lokalne";
        }
        else {
            lag_min_typ = "globalne";
        }

        //zapis do CSV
        file << exp_coef << ","
            << i + 1 << ","
            << start << ","
            << range_exp[0] << ","
            << range_exp[1] << ","
            << solution::f_calls << ","
            << sotulion_fib.x (0) << ","
            << sotulion_fib.y (0) << ","
            << sotulion_fib.f_calls << ","
            << fib_min_typ << ","
            << sotulion_lag.x (0) << ","
            << sotulion_lag.y (0) << ","
            << sotulion_lag.f_calls << ","
            << lag_min_typ << "\n";
    }
    file.close ();
}
```

Problem zostaje rozwiązany przy użyciu opisanych wcześniej metod, każda z nich stosowana jest 100 razy, z zastosowaniem współczynnika ekspansji równego 1,5. Następnie przeprowadzamy analogiczne obliczenia, stosując trzy różne metody, również 100-krotnie, ale tym razem z współczynnikami ekspansji równymi 9 oraz 16. Podczas każdego eksperymentu zapisujemy dane, takie jak: współrzędne uzyskanego minimum, zakres przedziału, liczba wywołań funkcji celu oraz informację, czy uzyskane minimum jest lokalne, czy globalne. Na podstawie zebranych danych obliczamy wartości średnie. Wyniki tych analiz przedstawiono w Tabeli 1 i Tabeli 2, które znajdują się w pliku Excel.

Na podstawie danych możemy stwierdzić, że wraz ze wzrostem współczynnika ekspansji rośnie średnia długość przedziału poszukiwań oraz liczba wywołań funkcji celu. Wyższy współczynnik zwiększa szansę na znalezienie minimum globalnego. Można też zauważyć, że metoda Fibonacciego charakteryzuje się bardziej stabilną liczbą wywołań funkcji celu i regularnym znalezieniem minimum globalnego. Metoda Lagrange’a

wykazuje większą zmienność, szczególnie dla wyższych współczynników ekspansji, gdzie częściej dochodzi do znalezienia minimów lokalnych.

W kolejnym etapie tworzymy wykres ilustrujący zależność długości przedziału od liczby iteracji, uwzględniając poszczególne kroki metod Fibonacciego i Lagrange'a. Na tym etapie nie stosujemy metody ekspansji. Otrzymane rezultaty znajdują się w zakładce "Wykres" w arkuszu Excel oraz poniżej jako zrzut ekranu:



Wykres pokazuje, jak długość przedziału zmniejsza się w kolejnych iteracjach dla Metody Fibonacciego i Metody Lagrange'a. Metoda Fibonacciego szybciej zawęża przedział na początku, co wskazuje na szybszą zbieżność, podczas gdy Metoda Lagrange'a redukuje długość przedziału wolniej na starcie, ale z czasem osiąga podobną efektywność. Ostatecznie obie metody prowadzą do podobnych wyników, natomiast metoda interpolacji Lagrange'a znajduje rozwiązanie w dużo mniejszej ilości iteracji w porównaniu do metody Fibonacciego.

Zadanie 2 problem rzeczywisty

Kolejnym zadaniem jest przeprowadzenie symulacji przepływu wody między dwoma zbiornikami. Celem symulacji jest znalezienie takiego pola przekroju otworu DA, aby temperatura wody w zbiorniku B nie przekroczyła 50 stopni. Zapisujemy następujące dane: objętości w zbiornikach A i B oraz temperaturę wody w zbiorniku B.

W celu rozwiązania tego problemu stworzyliśmy funkcje odwzorowujące równania różniczkowe, które zostały zapisane w pliku user_funs.c. Funkcje te opisują:

1. Zmianę objętości wody w zbiorniku, spowodowaną jej wypływem przez otwór o polu przekroju D.

```
matrix df1 (double t, matrix Y, matrix ud1, matrix ud2) { //zmiana objetosci wody w zbiorniku
    //predkosci wyplywow wody
    double outf_A = 0;
    double outf_B = 0;

    //do zbiornika B wlewa się woda o predkosci 10 l/s = 0.01m3/s
    double B_stream = 0.01;

    //temperatury wody w zbiornikach
    double B_temp = 10.0;
    double A_temp = 90.0;
    matrix d_Y (3, 1);

    if (Y (0) > 0) {
        outf_A = 0.98 * 0.63 * m2d (ud2) * sqrt (2.0 * 9.81 * Y (0) / 0.5);
    }

    if (Y (1) > 0) {
        outf_B = 0.98 * 0.63 * 0.00365665 * sqrt (2.0 * 9.81 * Y (1) / 1.0);
    }

    d_Y(0) = -outf_A;
    d_Y(1) = (outf_A + B_stream - outf_B);
    d_Y(2) = (B_stream / Y(1)) * (B_temp - Y(2)) + (outf_A / Y(1)) * (A_temp - Y(2));

    return d_Y;
}
```

2. Zmianę temperatury wody w zbiorniku B.

```
matrix ff1R (matrix x, matrix ud1, matrix ud2) { // zmiana temperatury w zbiorniku
    matrix y;
    matrix Y_0 = matrix(3, new double[3] {5, 1, 20}); //Kolejno V zbiornika A, V zbiornika B oraz temperatura zbiornika B
    matrix* Y = solve_ode (df1, 0, 1, 1000, Y_0, ud1, x);

    int n = get_len (Y[0]);
    double max = Y[1] (0, 2);

    for (int i = 0; i < n; i++) {
        if (max < Y[1] (i, 2)) max = Y[1] (i, 2);
    }

    y = abs(max - 50); //bezwzgledna między max a chcianym przez nas 50stop
    return y;
}
```

Funkcje są wywoływane za pomocą odpowiednich instrukcji w kodzie:

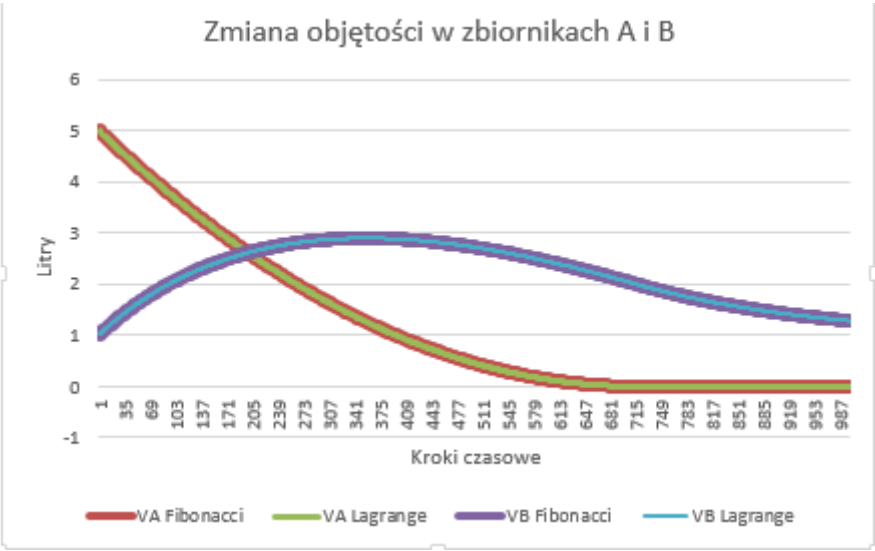
☺ Część kodu main odpowiedzialna za tworzenie symulacji

```
void lab1 (){ //sekcja rzeczywista
    ofstream file ("lab1_tab2_lag.csv");
    //ofstream file ("lab1_tab2_fib.csv");

    if (!file.is_open ()) {
        cout << "file error" << endl;
        return;
    }

    file << "Da;y;T\n";

    solution sotulion = lag (ff1R, 0.0001, 0.01, 0.0000001, 0.0000001, 1000);
    //solution sotulion = fib(ff1R, 0.0001, 0.01, 0.0000001);
    matrix YO = matrix(3, new double[3] {5, 1, 20});
    matrix* Y = solve_ode(df1, 0, 1, 1000, YO, NAN, sotulion.x (0));

    //arkusz_4
    cout << sotulion.x(0) << endl << sotulion.y(0) << endl << sotulion.f_calls << endl;
    //arkusz_5
    cout << Y[1] << endl;

    file << Y[1] << "\n";

    file.close ();
```

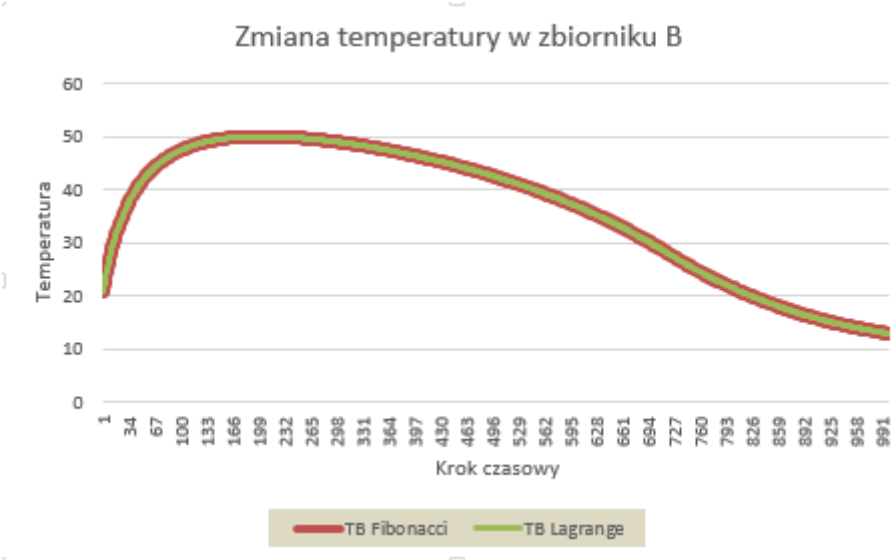
Wyniki symulacji zostały zamieszczone w zakładkach "Tabela 3" oraz "Symulacja" w pliku Excel, który dołączono do sprawozdania. W Tabeli 3 zamieszczamy rozwiązanie symulacji - wartość przekroju poprzecznego rury oraz ilość wywołań funkcji celu.

Metoda Fibonacciego osiągnęła wynik optymalny $DA*=0.00161518$ przy wartości $y*=0.00034583$ i wymagała 45 wywołań funkcji celu. Z kolei metoda oparta na interpolacji Lagrange'a uzyskała wynik $DA*=0.00161523$ z $y*=0.000110244$, ale wymagała aż 96 wywołań funkcji celu. Metoda Fibonacciego była więc bardziej efektywna pod względem liczby wywołań funkcji, podczas gdy metoda interpolacji Lagrange'a dała dokładniejszy wynik dla $y*$.

W zakładce “Symulacja” zamieszczamy zmieniające się wartości objętości płynu w zbiornikach A i B oraz temperaturę płynu w zbiorniku B oraz tworzymy na ich podstawie wykresy.



Wykres przedstawia zmianę objętości wody w dwóch zbiornikach (A i B) podczas trwania symulacji, dla metody Fibonacciego i metody Lagrange’a. W zbiorniku A objętość wody początkowo maleje, natomiast w zbiorniku B rośnie, co wskazuje na przepływ wody między zbiornikami. Po osiągnięciu punktu maksymalnego, objętość w zbiorniku B zaczyna maleć, a w A stabilizuje się. Obie metody dają podobne wyniki.



Na wykresie przedstawiono zmianę temperatury w zbiorniku B podczas trwania symulacji. Na osi pionowej oznaczono temperaturę w stopniach Celsjusza, a na osi poziomej numery kroków czasowych. Krzywe pokazują dwie różne metody obliczania temperatury: metodę Fibonacciego (czerwona linia) i metodę Lagrange'a (zielona linia). Obie metody dają bardzo podobne wyniki. Temperatura początkowo rośnie, osiągając szczyt w okolicach 150 kroku czasowego, a następnie stopniowo maleje.

Wnioski

Metoda Fibonacciego jest bardziej stabilna i efektywna pod względem liczby wywołań funkcji celu, regularnie prowadząc do znalezienia minimum globalnego. Metoda Lagrange'a charakteryzuje się większą zmiennością i częściej prowadzi do lokalnych minimów przy wyższych współczynnikach ekspansji, ale potrafi zbiegać szybciej niż metoda Fibonacciego.

Wzrost współczynnika ekspansji zwiększa długość przedziału poszukiwań i liczbę wywołań funkcji, co zwiększa szanse na znalezienie minimum globalnego, ale może powodować większe fluktuacje.

W zadaniu symulacji przepływu wody metoda Fibonacciego osiągnęła optymalne wyniki przy mniejszej liczbie wywołań, natomiast metoda Lagrange'a dała bardziej precyzyjny wynik, choć wymagała większej liczby wywołań.

Ostatecznie obie metody dają podobne rezultaty w symulacjach, zarówno w zakresie zmiany objętości wody, jak i temperatury, co wskazuje na ich równą skuteczność w rozwiązywaniu tego typu problemów.