AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Sprawozdanie 1

Optymalizacja funkcji jednej zmiennej metodami bezgradientowymi

Autorzy: *Paulina Grabowska*
Filip Rak
Arkadiusz Sala

Kierunek studiów: *Inżynieria Obliczeniowa*

Kraków, 2024

Spis treści

Cel ćwiczenia

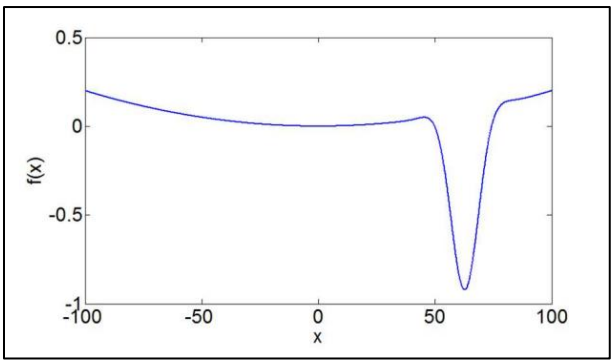
Celem przeprowadzonego ćwiczenia było pogłębienie wiedzy na temat bezgradientowych metod optymalizacji, w tym metody Fibonacciego, metody Lagrange'a oraz pomocniczej metody ekspansji. Zostały one zaimplementowane i zastosowane do rozwiązywania jednowymiarowych problemów optymalizacyjnych, umożliwiając praktyczne zrozumienie ich działania oraz efektywności.

Optymalizowane problemy

Testowa funkcja celu

Celem praktycznego wykorzystania zaimplementowanych algorytmów badano funkcję celu podaną wzorem (1), której wykres został załączony na *Rys.1*. Minimum globalne badanej funkcji na przedziale [-100; 100] znajduje się około punktu 67, a minimum lokalne w okolicach punktu 0.

$$f(x) = -cos\,(0{,}1x) \cdot e^{-(0.1x-2\pi)^2} + 0{,}002\,\cdot (0{,}1x)^2 \qquad (1)$$



Rys.1. Wykres funkcji (1)

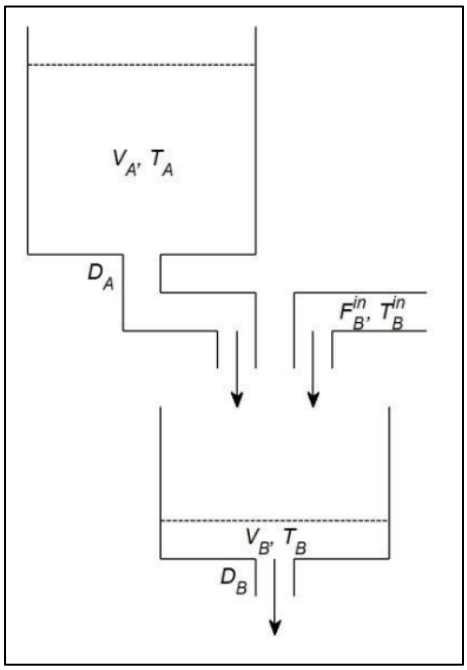
W pliku *user_funs.cpp* dodano implementację funkcji (1) z użyciem macierzy, jak przedstawiono we *Fragmencie kodu 1*.

```
1. matrix ff1T(matrix x, matrix ud1, matrix ud2)
2. {
3.         double t = -pow(0.1 * m2d(x) - 2 * M_PI,2);
4.         return -cos(0.1 * m2d(x)) * pow(M_E, t) + 0.00002 * pow(m2d(x),2);
5. }
6.
```

Fragment kodu 1: funkcja *matrix ff1T* licząca testową funkcję celu

Problem rzeczywisty

W analizowanym problemie mamy dwa zbiorniki z wodą – zbiornik *A*, będący górnym zbiornikiem, oraz zbiornik *B*, będący dolnym zbiornikiem (*Rys.2.).* Zbiornik *A* ma pole podstawy wynoszące 0,5 m² i początkowo zawiera 5 m³ wody o temperaturze 90°C. Z kolei zbiornik *B* ma większe pole podstawy, wynoszące 1 m², i początkowo mieści 1 m³ wody o temperaturze 20°C.



Rys.2. Ilustracja problemu rzeczywistego

Woda z górnego zbiornika *A* przepływa do dolnego zbiornika *B* poprzez otwór o polu przekroju D_A . Jednocześnie do zbiornika *B* dopływa woda o temperaturze 20°C z prędkością 10 litrów na sekundę. Ze zbiornika *B* woda wypływa przez otwór o polu przekroju $D_B = 36.5665$ cm^2

Zmiany objętości wody w zbiorniku są opisane równaniem (2), które uwzględnia wpływ parametrów takich jak lepkość cieczy (a = 0.98), współczynnik zwężenia strumienia (b = 0.63), oraz przyspieszenie ziemskie (g = 9.81 m/s^2). Równanie to pokazuje, że szybkość wypływu wody zależy od pola przekroju otworu i poziomu wody w zbiorniku.

$$\frac{dv}{dt} = -a \cdot b \cdot D \cdot \sqrt{2g\frac{v}{p'}} \qquad (2)$$

Zmiana temperatury wody w zbiorniku *B* jest opisana równaniem (3), które uwzględnia wpływ objętości i temperatury wody wpływającej oraz znajdującej się już w zbiorniku. W ten sposób możemy śledzić, jak wraz z dopływem zimnej wody i odpływem z dolnego zbiornika zmieniają się objętość oraz temperatura wody w zbiorniku *B* w czasie.

$$\frac{dT}{dt} = \frac{V^{in}}{v} \cdot \left(T^{in} - T\right) \tag{3}$$

Ponieważ do rozwiązania problemu rzeczywistego niezbędne jest rozwiązanie dwóch równań różniczkowych w pliku user_funs.cpp napisano dwie funkcje: *flow_and_temp* oraz *simulate_flow_temp*. Funkcja *flow_and_temp* (*Fragment kodu 2*) symuluje zmiany objętości i temperatury w dwóch zbiornikach w czasie. Zawiera modele fizyczne opisujące przepływ cieczy między zbiornikami oraz zmiany temperatury w zbiorniku *B*.

Zadeklarowano niezbędne stałe fizyczne oraz parametry zbiorników i wpływu. Następnie obliczany jest przepływ cieczy, kolejno w zbiornikach *A* oraz *B*, jak również liczona jest zmiana objętości w tych zbiorniach oraz zmiana temperatury w zbiorniku *B*. Funkcja *flow_and_temp* zwraca macierz zawierającą zmiany objętości w zbiorniku *A* i *B* oraz zmianę temperatury w zbiorniku *B*.

```
matrix flow_and_temp(double t, matrix Y, matrix ud1, matrix ud2)
{
	// Result vector - changes in volume and temp
	matrix dY(3, 1);

	// Physical coefficients: viscosity, contraction, gravitational acceleration
	double a = 0.98, b = 0.63, g = 9.81;

	// A tank parameters
	double area_a = 0.5;
	double temp_a = 90.0;

	// B tank parameters
	double area_b = 1.0;
	double outflow_area_b = 0.00365665;

	// Inflow parameters
	double inflow_rate = 0.01;
	double temp_increase = 20.0;

	// Get outflows from each tank
	double outflow_a = Y(0) > 0 ? a * b * m2d(ud2) * sqrt(2 * g * Y(0) / area_a) : 0;
	double outflow_b = Y(1) > 1 ? a * b * outflow_area_b * sqrt(2 * g * Y(1) / area_b) : 0;

	// Changes in volume and temp:
	dY(0) = -outflow_a; // Change in volume of tank A
	dY(1) = outflow_a + inflow_rate - outflow_b;	// Change in volume of tank B
	dY(2) = inflow_rate / Y(1) * (temp_increase - Y(2)) + outflow_a / Y(1) * (temp_a -
Y(2));	// Change in temp in tank B

	return dY;
}
```

Fragment kodu 2: funkcja *flow_and_temp*

Natomiast funkcja *simulate_flow_temp* symuluje proces przepływu cieczy oraz zmianę temperatury w systemie dwóch zbiorników w określonym czasie, a także oblicza maksymalną temperaturę w zbiorniku *B*. Warunki początkowe zostały ustawione zgodnie z treścią problemu początkowego.

Ustawione zostały również parametry symulacji czasu. Następnie numerycznie rozwiązywane jest równanie różniczkowe, korzystając z wcześniej zdefiniowanej funkcji *flow_and_temp*. Kolejno obliczana jest maksymalna temperatura w zbiorniku *B* oraz odchylenie tej temperatury od 50 °C.

```
matrix simulate_flow_temp(matrix x, matrix ud1, matrix ud2)
{
	// Initial conditions
	double volume_a = 5, volume_b = 1, temp_b = 20;
	double conditions[3] = { volume_a, volume_b, temp_b };
	matrix Y0 = matrix(3, conditions);

	// Time
	double start_time = 0.0;
	double end_time = 2000.0;
	double time_step = 1.0;

	// Solve differential equation
	matrix* results = solve_ode(flow_and_temp, start_time, time_step, end_time, Y0, ud1,
x);

	// Get max temp in B tank
	double max_temp_b = results[1](0, 2);

	for (int i = 1; i < get_len(results[0]); i++)
	{
		if (max_temp_b < results[1](i, 2))
			max_temp_b = results[1](i, 2);
	}

	// Get dabsolute deviation from the temp of 50
	double temp_deviation = abs(max_temp_b - 50);
	return temp_deviation;
}
```

Fragment kodu 3: funkcja *simulate_flow_temp*

Implementacja metod optymalizacji

Metoda ekspansji

Funkcja *expansion* (*Fragment kodu 4*) to algorytm optymalizacyjny, który ma na celu znalezienie przedziału w przestrzeni wartości zmiennej, gdzie istnieje minimum funkcji celu. Funkcja zaczyna od utworzenia dwóch instancji klasy *solution* z początkową wartością *x0* oraz *x0 + d*. Instancje te przechowują wartości zmiennej wejściowej oraz wynik funkcji celu po jej wywołaniu. Jeśli wartości funkcji celu w punktach *x0* i *x0 + d* są równe, algorytm zakłada, że znalazł przedział i zwraca wynik. W przeciwnym przypadku, jeżeli wartość funkcji w punkcie *x0 + d* jest większa niż w punkcie *x0*, kierunek poszukiwań zostaje odwrócony poprzez zmianę znaku *d*.

Następnie algorytm kontynuuje proces rozszerzania przedziału, dopóki wartości funkcji celu w kolejnych krokach nie zaczynają rosnąć, co oznacza, że algorytm przekroczył minimum. Przy każdym kroku *alpha* jest mnożone przez siebie, co powoduje zwiększanie kroku ekspansji, a funkcja celu jest obliczana w nowych punktach. Proces ten powtarza się, aż funkcja *ff* zacznie zwracać większe wartości, co oznacza, że znaleziono minimum w określonym przedziale.

Po zakończeniu procesu, funkcja zwraca wskaźnik na tablicę *p* z dwoma wartościami. Są to granice przedziału, w którym spodziewamy się istnienia minimum funkcji celu. Jeśli w trakcie działania liczba wywołań funkcji przekroczy maksymalny limit (*Nmax*), funkcja zgłasza wyjątek i kończy działanie.

Funkcja *expansion* jest typowym przykładem metody optymalizacyjnej, w której kluczowe jest znalezienie odpowiedniego przedziału, w którym istnieje minimum. Algorytm działa adaptacyjnie, zmieniając kierunek oraz szybkość poszukiwań w zależności od wyników funkcji celu, co pozwala efektywnie lokalizować optymalny zakres.

```
 1. double* expansion(matrix(*ff)(matrix, matrix, matrix), double x0, double d, double alpha, int
Nmax, matrix ud1, matrix ud2)
 2. {
 3.       try
 4.       {
 5.                double* p = new double[2]{ 0,0 };
 6.
 7.                solution X0(x0), X1(x0 + d);
 8.                X0.fit_fun(ff, ud1, ud2);
 9.                X1.fit_fun(ff, ud1, ud2);
10.
11.                if (X0.y == X1.y)
12.                {
13.                          p[0] = m2d(X0.x);
14.                          p[1] = m2d(X1.x);
15.                          return p;
16.                }
17.
18.                if (X1.y > X0.y)
19.                {
20.                          d = -d;
21.
22.                          X1.x = X0.x + d;
23.                          X1.fit_fun(ff, ud1, ud2);
24.                          if (X1.y >= X0.y)
25.                          {
26.                                    p[0] = m2d(X1.x);
27.                                    p[1] = m2d(X0.x)-d;
28.
29.                                    return p;
30.                          }
31.                }
32.                double prev;
33.                do
34.                {
35.                          if (solution::f_calls > Nmax) {
36.
37.                                    throw string(string("Nie znaleziono przedzialu po " +
Nmax) + " probach");
38.                          }
39.                          prev = m2d(X0.x);
40.                          X0 = X1;
41.                          X1.x = x0 + alpha * d;
42.                          X1.fit_fun(ff, ud1, ud2);
43.                          alpha *= alpha;
44.
45.                } while (X1.y <= X0.y);
46.
47.                if (prev < X1.x) {
48.                          p[0] = prev;
49.                          p[1] = m2d(X1.x);
50.                }
51.                else
52.                {
53.                          p[0] = m2d(X1.x);
54.                          p[1] = prev;
55.                }
56.
57.                return p;
58.       }
59.       catch (string ex_info)
60.       {
61.                throw ("double* expansion(...):\n" + ex_info);
62.       }
63. }
```

Fragment kodu 4: funkcja *expansion*

Metoda Fibonacciego

Funkcja *fib* (*Fragment kodu 5*) implementuje metodę optymalizacji wykorzystującą algorytm Fibonacciego do minimalizacji funkcji celu w zadanym przedziale *[a,b]* z tolerancją ϵ. Algorytm ten wykorzystuje podział przedziału w każdym kroku opierając się na liczbach Fibonacciego, co zapewnia znalezienie minimum w przypadku funkcji jednokierunkowej.

Algorytm zaczyna od obliczenia minimalnej liczby kroków *k*, tak aby *k*-ty element ciągu Fibonacciego był większy niż stosunek długości przedziału *[a,b]* do ϵ. Jest to warunek pozwalający zredukować przedział do oczekiwanej dokładności. Pętla, w której generowane są kolejne elementy ciągu Fibonacciego, trwa dopóki odpowiedni element nie przekroczy wartości *deps*.

Po obliczeniu wartości *k*, algorytm tworzy początkowe punkty *c0* i *d0* wewnątrz przedziału *[a,b]* zgodnie z formułami opartymi na liczbach Fibonacciego. Algorytm iteracyjnie zmniejsza przedział, porównując wartości funkcji celu w punktach *c0* i *d0*. W każdym kroku, jedna z wartości funkcji jest mniejsza, co pozwala wyeliminować część przedziału, w której nie występuje minimum. Proces ten trwa do $k-3$ iteracji. Każdy krok zmniejsza długość przedziału poszukiwań, aż osiągnie ona akceptowalny poziom wyznaczony przez epsilon.

Po zakończeniu iteracji punkt *c0* staje się optymalnym punktem *Xopt*, który zawiera minimum funkcji celu w przedziale *[a,b]*. Algorytm zwraca rozwiązanie *Xopt*, które - oprócz wartości funkcji celu w punkcie minimalnym - zawiera informacje o przebiegu optymalizacji.

Metoda Fibonacciego to efektywna metoda optymalizacji, która wykorzystuje własności ciągu Fibonacciego do sukcesywnego zmniejszania przedziału, w którym znajduje się minimum funkcji celu. Algorytm ten jest szczególnie przydatny w optymalizacji funkcji jednokierunkowych, gdzie minimalizujemy jedną zmienną w zadanym przedziale.

```
 1. solution fib(matrix(*ff)(matrix, matrix, matrix), double a, double b, double epsilon, matrix ud1, matrix
ud2)
 2. {
 3.         try
 4.         {
 5.                   solution Xopt;
 6.
 7.
 8.                   //znajdz najmniejsza k, tak aby k-ty element ciagu fibonaciego byl wiekszy od dlugosci
przedzialu przez epsylon
 9.                   double deps = (b - a) / epsilon;
10.                   //std::cout <<"DEPS:" << deps << std::endl;
11.                   int k = 2;
12.                   int Mk = 10;
13.                   const int ak = 5;
14.                   int Pk = 10;
15.                   long* tQ = new long[Mk];
16.                   tQ[0] = 0; tQ[1] = 1; tQ[2] = 1; tQ[3] = 2;  tQ[4] = 3;
17.                   tQ[5] = 5; tQ[6] = 8; tQ[7] = 13; tQ[8] = 21; tQ[9] = 34;
18.                   while (tQ[k] <= deps)
19.                   {
20.                             k++;
21.
22.                             if (k != Mk)
23.                             {
24.                                       Mk += ak;
25.                                       long* tmpQ = new long[Mk];
26.                                       std::copy(tQ, tQ + Mk - ak, tmpQ);
27.                                       delete[] tQ;
28.                                       tQ = tmpQ;
29.                             }
30.
31.                             if (k == Pk) {
32.
33.                                       tQ[k] = tQ[k - 1] + tQ[k - 2];
34.                                       Pk++;
35.                             }
36.
37.                   }
38.
39.                   solution a0(a), b0(b);
40.                   solution c0(b0.x - (static_cast<double>(tQ[k - 1]) / tQ[k]) * (b0.x - a0.x));
41.                   solution d0(a0.x + b0.x - c0.x);
42.                   Xopt.ud = b - a;
43.                   for (int i = 1; i <= k - 3; i++)
44.                   {
45. //std::cout << a0.x << b0.x;
46. //std::cout << c0.x << d0.x << std::endl;
47.                             if (c0.fit_fun(ff, ud1, ud2) < d0.fit_fun(ff, ud1, ud2))
48.                                       b0 = d0;
49.                             else
50.                                       a0 = c0;
51.
52.                             c0.x = b0.x - (static_cast<double>(tQ[k - i - 2]) / tQ[k - i - 1]) * (b0.x -
a0.x);
53.                             d0.x = a0.x + b0.x - c0.x;
54. //std::cout << a0.x << b0.x;
55. //std::cout << c0.x << d0.x << std::endl;
56.                             Xopt.ud.add_row(m2d(b0.x - a0.x));
57.                   }
58.
59.                   Xopt = c0;
60.                   Xopt.flag = 0;
61.                   //std::cout << Xopt.ud << std::endl << std::endl;
62.                   delete[] tQ;
63.
64.                   return Xopt;
65.         }
66.         catch (string ex_info)
67.         {
68.                   solution::clear_calls();
69.                   throw ("solution fib(...):\n" + ex_info);
70.         }
71.
72. }
```

Fragment kodu 5: funkcja *fib*

Metoda Lagrange'a

Funkcja *lag* (*Fragment kodu 6*) implementuje algorytm optymalizacji metodą interpolacji kwadratowej Lagrange'a. Algorytm opiera się na konstrukcji parabol opartych na trzech punktach i iteracyjnej aktualizacji tego podziału, aż do zbliżenia się do optymalnego rozwiązania.

Algorytm rozpoczyna od zainicjowania trzech punktów *ai, bi* oraz *ci* w przedziale *[a,b]*, gdzie *ci* znajduje się pośrodku przedziału. Następnie dla tych punktów obliczane są wartości funkcji celu, co będzie wykorzystywane do konstrukcji interpolacji. W każdej iteracji algorytm konstruuje parabolę na podstawie wartości funkcji w punktach *ai, bi* oraz *ci*. Następnie wyznaczany jest nowy punkt *di*, czyli minimum paraboli. Formuły do obliczenia *l* i *m* opierają się na równaniach interpolacji Lagrange'a. Kolejno, na podstawie tych wartości obliczany jest nowy punkt *di*. Jeśli nowy punkt *di* różni się od poprzedniego o wartość mniejszą niż γ, algorytm kończy swoje działania.

Jeśli zbieżność nie została osiągnięta, algorytm porównuje wartość funkcji w nowym punkcie *di* z wartością funkcji w pozostałych punktach *ai, bi* oraz *ci*. Na tej podstawie aktualizowany jest przedział, w którym znajduje się minimum, poprzez zastępowanie punktu o większej wartości funkcji nowym punktem *di*. Algorytm wybiera odpowiedni przedział na podstawie porównania wartości funkcji. Proces ten jest powtarzany, dopóki długość przedziału nie spadnie poniżej ϵ lub liczba iteracji nie przekroczy *Nmax*. Gdy warunek zbieżności zostanie spełniony, algorytm zwraca optymalny punkt *Xopt*, który zawiera wartość minimalną funkcji i dodatkowe informacje pomocnicze.

Metoda interpolacji kwadratowej Lagrange'a jest skuteczną techniką optymalizacji, która wykorzystuje parabole do znajdowania minimum funkcji. Proces ten iteracyjnie zawęża przedział poszukiwań poprzez obliczanie nowych punktów minimalnych i aktualizację przedziału, aż do osiągnięcia odpowiedniego stopnia dokładności.

```
 1. solution lag(matrix(*ff)(matrix, matrix, matrix), double a, double b, double epsilon, double gamma, int Nmax,
matrix ud1, matrix ud2)
 2. {
 3.         try
 4.         {
 5.                     solution Xopt;
 6.
 7.
 8.                     solution ai(a), bi(b), ci((a + b) * .5);
 9.                     ai.fit_fun(ff, ud1, ud2);
10.                     bi.fit_fun(ff, ud1, ud2);
11.                     ci.fit_fun(ff, ud1, ud2);
12.                     long double di_1 = 0;
13.                     int i = 0;
```

```
14.                     solution di(0);
15.                     Xopt.flag = 0;
16.                     do
17.                     {
18.                             if (!i)
19.                             {
20.                                     i++;
21.                                     Xopt.ud = b - a;
22.                             }
23.                             else
24.                                     Xopt.ud.add_row(m2d(bi.x - ai.x));
25.                             matrix l = ai.y * (pow(bi.x, 2) - pow(ci.x, 2)) + bi.y * (pow(ci.x, 2) - pow(ai.x,
2))
26.                                     + ci.y * (pow(ai.x, 2) - pow(bi.x, 2));
27.                             matrix m = ai.y * (bi.x - ci.x) + bi.y * (ci.x - ai.x) + ci.y * (ai.x - bi.x);
28.                             if (m <= 0)
29.                             {
30.                                     Xopt.flag = -1;
31.                                     return Xopt;
32.                             }
33.
34.                             di_1 = m2d(di.x);
35.                             di = solution( 0.5 * m2d(l) / m2d(m));
36.                             if (abs(m2d(di.x) - di_1) < gamma)
37.                                     break;
38.                             di.fit_fun(ff, ud1, ud1);
39.                             if (ai.x < di.x && di.x < ci.x)
40.                             {
41.                                     if (di.y < ci.y)
42.                                     {
43.                                             bi.x = ci.x; bi.y = ci.y;
44.                                             ci.x = di.x; ci.y = di.y;
45.
46.                                     }
47.                                     else
48.                                     {
49.                                             ai.x = di.x; ai.y = di.y;
50.                                     }
51.
52.                             }
53.                             else if (ci.x < di.x && di.x < bi.x)
54.                             {
55.                                     if (di.y < ci.y)
56.                                     {
57.                                             ai.x = ci.x; ai.y = ci.y;
58.                                             ci.x = di.x; ci.y = di.y;
59.                                     }
60.                                     else
61.                                     {
62.                                             bi.x = di.x; bi.y = di.y;
63.                                     }
64.
65.                             }
66.                             else {
67.                                     Xopt.flag = -1;
68.                                     Xopt.x = di_1;
69.                                     Xopt.fit_fun(ff, ud1, ud2);
70.                                     return Xopt;
71.                                     //throw string("di poza zakresem\n");
72.                             }
73.                             if (solution::f_calls > Nmax) {
74.
75.                                     throw string(string("Nie znaleziono przedzialu po " + Nmax) + "
probach");
76.                             }
77.
78.                     } while (m2d(bi.x) - m2d(ai.x) >= epsilon);
79.
80.                     Xopt.x = di.x;
81.                     Xopt.y = di.y;
82.
83.                     Xopt.fit_fun(ff, ud1, ud2);
84.                             return Xopt;
85.             }
86.
87.             catch (string ex_info)
88.             {
89.                     solution::clear_calls();
90.                     throw ("solution lag(...):\n" + ex_info);
91.             }
92. }
```

Fragment kodu 6: funkcja *lag*

Opracowanie wyników

Zadeklarowano zmienne lokalne wykorzystywane podczas późniejszego testowania (*Fragment kodu 7*). Składają się na nie:

- dokładność obliczeń *epsilon* równa 1e-05
- maksymalna liczba wywołań funkcja *n_max* równa 1000
- wartości *x_min* = -100 oraz *x_max* = 100 będące granicami przedziału
- parametr *d* = 2 będący krokiem w metodzie ekspansji
- parametr *alpha* będący współczynnikiem ekspansji
- parametr *gamma* = 1e-200 zapewniający większą stabliność w metodzie Lagrange'a

```
1. // Common arguments
2.       double epsilon = 1e-05;
3.       int n_max = 1000;
4.
5.       // Expansion-specific arguments
6.       int x_min = -100, x_max = 100;  // Boundries for random number generation
7.       double d = 2, alpha = 13;
8.
9.       // Lagrangea-specific arguments
10.      double gamma = 1e-200;
```

Fragment kodu 7: zmienne lokalne niezbędne dla działania badanych funkcji

Testowa funkcja celu

Obrano trzy współczynniki ekspansji, kolejno: 2, 13 oraz 17. Badania przeprowadzono dla każdej z dwóch metod optymalizacji (metoda Lagrange'a i metoda Fibonacciego), po 100 optymalizacji dla każdego współczynnika. Wartości uzyskanych wyników zestawiono w *Tabeli 1*, a następnie w Tabeli 2 zebrano ich uśrednione wartości.

Celem uzyskania danych do obu tych tabel, w funkcji *main* zawarto pętlę (*Fragment kodu 8*), w której dla 100 losowych punktów startowych przeprowadzane są różne metody numeryczne w celu znajdowania minimum funkcji.

```
// ---------- Table 1 and Table 2 ----------

	// Init random number generator
	srand(time(NULL));

	for (int i = 0; i < 100; i++)
	{
		// Narrow the range using expansion with random x0
		double x0 = x_min + (double)rand() / RAND_MAX * (x_max - x_min);
		double* range = expansion(ff1T, x0, d, alpha, n_max);

		if (exp_file.is_open())
			exp_file << x0 << delimiter << range[0] << delimiter << range[1]
			<< delimiter << solution::f_calls << delimiter << "\n";

		// Use Fibonnaci's method
		solution::clear_calls();
		solution fib_result = fib(ff1T, range[0], range[1], epsilon);

		if (fib_file.is_open())
			fib_file << m2d(fib_result.x) << delimiter << m2d(fib_result.y)
			<< delimiter << solution::f_calls << delimiter << fib_result.flag
<< delimiter << "\n";

		// Use Lagrange's method
		solution::clear_calls();
		solution lag_result = lag(ff1T, range[0], range[1], epsilon, gamma, n_max);

		if (lag_file.is_open())
			lag_file << m2d(lag_result.x) << delimiter << m2d(lag_result.y)
			<< delimiter << solution::f_calls << delimiter << lag_result.flag
<< delimiter<< "\n";


		// Deallocate memory
		delete[] range;
	}

	// Close the files
	exp_file.close();
	fib_file.close();
	lag_file.close();
```

Fragment kodu 8: fragment funkcji *main* zawierający funkcjonalności dla *Tabeli 1* oraz *Tabeli 2*

Analizując liczbę wywołań funkcji celu, można zauważyć, że metoda interpolacji Lagrange'a wykazuje się mniejszą jej liczbą w porównaniu do metody Fibonacciego. Jest to zauważalne szczególnie przy współczynnikach ekspansji 2 i 17. Metoda Fibonacciego wymaga większej liczby wywołań, szczególnie dla minimów lokalnych.

Wszystkie metody ogólnie dobrze radzą sobie ze znajdowaniem zarówno minimum globalnego, jak i lokalnego, ale metoda Lagrange'a ma pewne przypadki braku zbieżności. Najmniej przypadków braku zbieżności występuje dla współczynnika ekspansji 17, gdzie występuje on 5 razy w metodzie Lagrange'a. Wyższy współczynnik ekspansji (17) daje lepszą stabilność z mniejszą liczbą przypadków braku zbieżności, ale kosztem precyzji minimów lokalnych.

Obie metody skutecznie znajdowały minima globalne dla testowanych współczynników ekspansji. W przypadku minimów lokalnych, metoda Lagrange'a często dawała wyniki bliskie

zeru, co zbliża się do wartości rzeczywistych. Natomiast metoda Fibonacciego dawała wyniki odchylone od oczekiwanych.

Tabela 2. Wartości średnie wyników optymalizacji dla trzech współczynników ekspansji

Współczynnik ekspansji	Metoda ekspansji	
	b - a	Liczba wywołań funkcji celu
2	330,559986	15,21
13	236,3599488	15,43
17	396,5399757	14,22

Współczynnik ekspansji	Rodzaj minimum	Metoda Fibonacciego			
		x*	y*	Liczba wywołań funkcji celu	Liczba wystąpień
2	Minimum globalne	62,7482	-0,921148	65,61290323	31
	Minimum lokalne	0,909394576	-0,013349971	67,68115942	69
	Brak zbieżności			-	0
13	Minimum globalne	62,7482	-0,921148	65,69230769	39
	Minimum lokalne	7,74804E-08	2,99928E-16	67,24590164	61
	Brak zbieżności			-	0
17	Minimum globalne	62,7482	-0,921148	66,34482759	29
	Minimum lokalne	0,883777381	-0,012973915	66,34482759	71
	Brak zbieżności			-	0

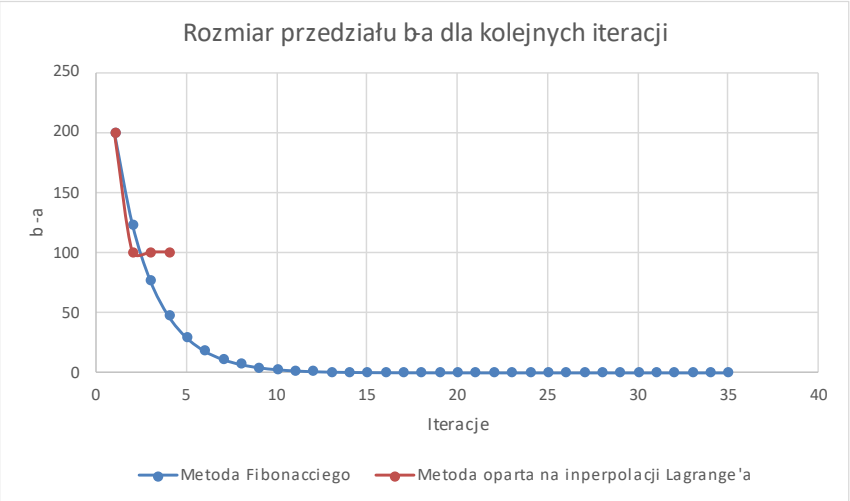
Współczynnik ekspansji	Rodzaj minimum	Metoda oparta na interpolacji Lagrange'a			
		x*	y*	Liczba wywołań funkcji celu	Liczba wystąpień
2	Minimum globalne	62,74719167	-0,921147833	11,16666667	12
	Minimum lokalne	2,2092E-13	-7,15717E-18	10,9516129	82
	Brak zbieżności			3,5	6
13	Minimum globalne	62,73135385	-0,921103923	12,46153846	13
	Minimum lokalne	2,2505E-13	-0,921103923	12,97101449	79
	Brak zbieżności			4,75	8
17	Minimum globalne	62,737025	-0,921039	12,75	8
	Minimum lokalne	1,567798413	-0,003294333	12,11111111	87
	Brak zbieżności			3,2	5

Następnie narysowano wykres, bez wstępnego zawężania przedziału, który przedstawia długość przedziału *[a, b]* jako funkcję numeru iteracji (*Rys.3*). W tym celu w funkcji *main* (*Fragment kodu 9*) zawarto funkcjonalności pozwalające na bezpośrednie zapisanie do pliku danych dotyczących minimum oraz liczby wywołań dla metody Fibonacciego oraz Lagrange'a

```
// ---------- Graph ----------

        // Open new files for output
        fib_file.open(OUTPUT_PATH + "out_2_1_fib.txt");
        lag_file.open(OUTPUT_PATH + "out_2_2_lag.txt");

        // Fixed range
        double a = -100, b = 100;

        // Fibonacci function call
        solution::clear_calls();
        solution fib_result = fib(ff1T, a, b, epsilon);
        fib_file << m2d(fib_result.x) << delimiter << m2d(fib_result.y)
                 << delimiter << solution::f_calls << delimiter << fib_result.flag
                 << delimiter << "\n\n" << fib_result.ud << "\n";

        // Lagrange Functions call
        solution::clear_calls();
        solution lag_result = lag(ff1T, a, b, epsilon, gamma, n_max);
        lag_file << m2d(lag_result.x) << delimiter << m2d(lag_result.y)
                 << delimiter << solution::f_calls << delimiter << lag_result.flag
                 << delimiter << "\n\n" << lag_result.ud << "\n";

        // Close the files
        fib_file.close();
        lag_file.close();
```

Fragment kodu 9: fragment funkcji *main* pozwalający na zebranie odpowiednich danych do utworzenia grafu funkcji długości przedziału od numeru iteracji.

Przebieg metody Fibonacciego jest bardziej płynny, co oznacza, że redukcja przedziału $b-a$ zachodzi systematycznie w każdej iteracji. W miarę postępu iteracji, wielkość przedziału szybko się zmniejsza i po około 15 iteracjach osiąga stabilny, bardzo niski poziom. Metoda Lagrange'a wykazuje szybsze zmniejszenie przedziału, co można zauważyć po gwałtownych spadkach wartości $b-a$ w pierwszych kilku iteracjach.



Rys.3. Wykres funkcji długości przedziału od numeru iteracji

Problem rzeczywisty

W przypadku problemu rzeczywistego program został uruchomiony jedynie raz, a jego wyniki zostały zebrane w *Tabeli 3*. Celem uzyskania wyników wykorzystano kod załączony na *Fragmencie kodu 10*, będący częścią funkcji *main*. Dla każdej z metod wywoływana jest funkcja dla zadanego zakresu, a wyniki zapisywane są do pliku.

```
// ---------- Table 3 ----------

      // Open new files
      fib_file.open(OUTPUT_PATH + "out_3_1_fib.txt");
      lag_file.open(OUTPUT_PATH + "out_3_2_lag.txt");

      // Table specific function arguments
      double range[] = { 1e-4, 1e-2 };

      // Call Fibonacci
      solution::clear_calls();
      fib_result = fib(simulate_flow_temp, range[0], range[1], epsilon);
      fib_file << m2d(fib_result.x) << delimiter << m2d(fib_result.y) << delimiter
               << solution::f_calls << delimiter << fib_result.flag << delimiter << "\n";

      // Call Lagrange
      solution::clear_calls();
      lag_result = lag(simulate_flow_temp, range[0], range[1], epsilon, gamma, n_max);
      lag_file << m2d(lag_result.x) << delimiter << m2d(lag_result.y) << delimiter
               << solution::f_calls << delimiter << lag_result.flag << delimiter << "\n";

      // Close the files
      fib_file.close();
      lag_file.close();
```

Fragment kodu 10: fragment funkcji main zawierający funkcjonalności dla *Tabeli 3*

Uzyskane wyniki, obiema metodami, nie różnią się zbytnio od siebie. Różnica między otrzymanymi w wyniku optymalizacji wartościami *DA** jest rzędu 1e-06 rzędu; co oznacza że obie metody dały porównywalne wyniki. Wartą zauważenia różnicą natomiast jest mniejsza liczba wywołań funkcji celu podczas korzystania z metody opartej na interpolacji Lagrange'a.

Tabela 3. Zestawienie wyników optymalizacji dla problemu rzeczywistego

Metoda Fibonacciego			Metoda oparta na interpolacji Lagrange'a		
DA*	y*	Liczba wywołań funkcji celu	DA*	y*	Liczba wywołań funkcji celu
0,00116322	0,0554825	28	0,00116788	0,0017075	21

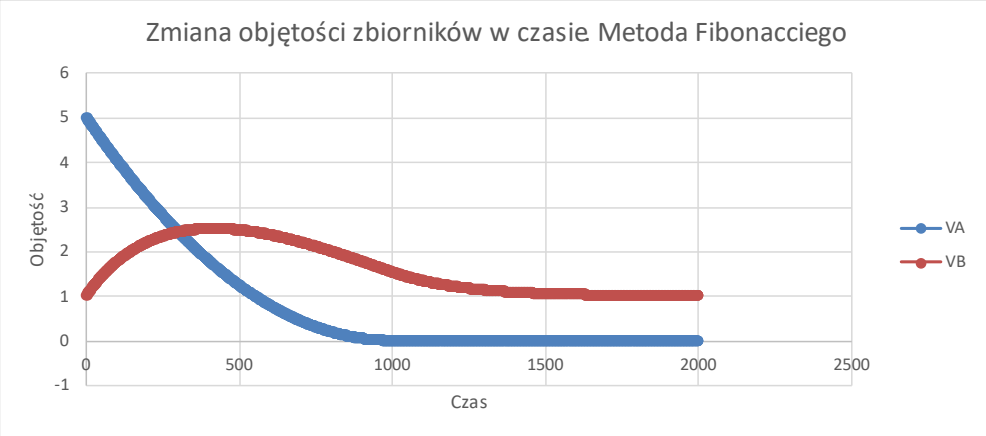
Następnie przystąpiono do przeprowadzenia symulacji, dla uzyskanego przed chwilą optymalnego pola przekroju *DA**. Kod, który pozwala na uzyskanie odpowiednich danych – tj.

Objętości wody w zbiornikach *A* oraz *B*, a także temperaturę wody w zbiorniku *B* wprowadzono do funkcji *main* funkcjonalności zawarte we *Fragmencie kodu 11*.

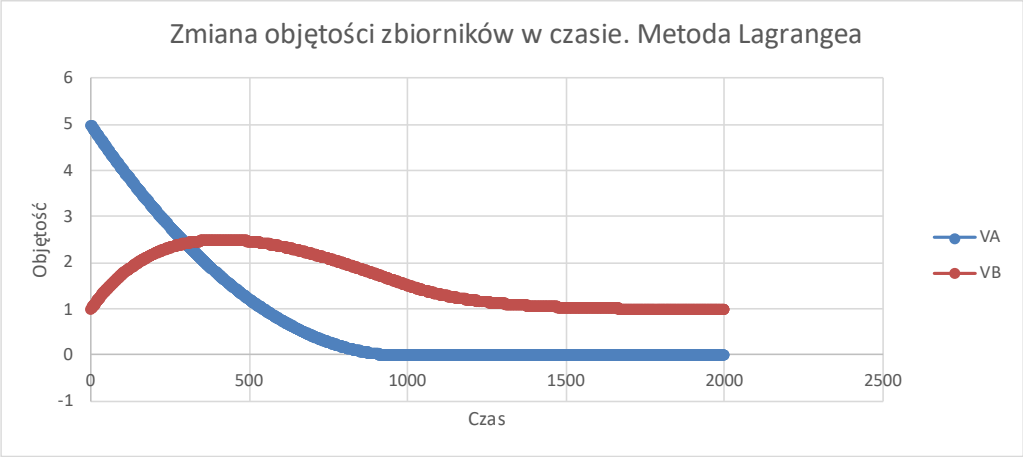
```
// ---------- Simulation ----------
        // Initial conditions
        double volume_a = 5, volume_b = 1, temp_b = 20;
        double conditions[3] = { volume_a, volume_b, temp_b };
        matrix Y0 = matrix(3, conditions);

        // Time
        double start_time = 0.0;
        double end_time = 2000.0;
        double time_step = 1.0;

        // Solve differential equation with result from Fibonacci
        matrix* ode_fib_result = solve_ode(flow_and_temp, start_time, time_step, end_time, Y0,
NULL, fib_result.x(0));

        // Solve differential equation with result from Lagrange
        matrix* ode_lag_result = solve_ode(flow_and_temp, start_time, time_step, end_time, Y0,
NULL, lag_result.x(0));

        // Open files and save results
        fib_file.open(OUTPUT_PATH + "out_4_1_fib.txt");
        lag_file.open(OUTPUT_PATH + "out_4_2_lag.txt");

        fib_file << ode_fib_result[1];
        lag_file << ode_lag_result[1];

        // Close the files
        fib_file.close();
        lag_file.close();
```

Fragment kodu 11: fragment funkcji *main* pozwalający na zebranie odpowiednich danych do utworzenia odpowiednich wykresów

Na *Rys.4* oraz *Rys.5* przedstawiono wykresy zmiany objętości zbiorników *A* (niebieska krzywa) oraz *B* (czerwona krzywa) w czasie dwóch tysięcy sekund (z krokiem czasowym 1s) dla obu metod optymalizacji. Wykresy te wyglądają identycznie i gołym okiem nie widać pomiędzy nimi żadnych różnic. Minimalne różnice są po prawdzie widoczne w tabeli utworzonej na potrzeby symulacji, ale nie są one bardzo znaczące w całokształcie rozwiązania.

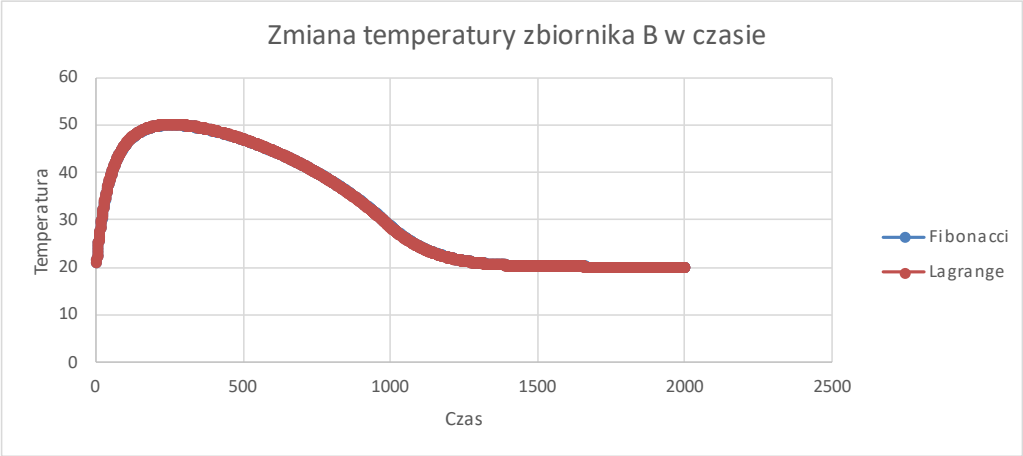


Rys. 4. Wykres zmiany objętości w czasie uzyskany metodą Fibonacciego



Rys. 5. Wykres zmiany objętości w czasie uzyskany metodą Lagrange'a

Wykonano również wykres zmiany temperatury w zbiorniku *B* w czasie (*Rys. 6*). Obie krzywe narysowane na wykresie, reprezentujące każdą metodę, pokrywają się - co wskazuje, że różnice między metodami są na tyle niewielkie, że nie są dostrzegalne.



Rys. 6. Wykres zmiany temperatury w zbiorniku *B*
uzyskany obiema metodami optymalizacji

Wnioski

W przypadku testowej funkcji celu analiza wyników optymalizacji wykazała, że metoda interpolacji Lagrange'a charakteryzuje się mniejszą liczbą wywołań funkcji celu w porównaniu do metody Fibonacciego. Choć obie metody skutecznie znajdowały zarówno minima globalne, problem pojawił się przy minimach lokalnych – metoda Fibonacciego dawała wyniki odchylone i niespójne. Metoda Lagrange'a wykazywała pewne przypadki braku zbieżności, przy czym najmniej takich przypadków zaobserwowano dla współczynnika ekspansji 17, co sugeruje lepszą stabilność w tym zakresie, choć kosztem precyzji w lokalnych minimach.

Metoda Fibonacciego zapewniała bardziej płynny przebieg optymalizacji, z systematycznym zmniejszaniem przedziału w każdej iteracji, osiągając stabilny poziom po około 15 iteracjach. Z kolei metoda Lagrange'a pozwalała na szybsze zmniejszenie przedziału, co było widoczne w gwałtownych spadkach wartości przedziału w początkowych iteracjach.

W odniesieniu do problemu rzeczywistego wyniki uzyskane przy obu metodach były porównywalne, z różnicą w wartościach *DA** rzędu 1e-06, co sugeruje, że obie metody dostarczają zbliżonych rezultatów. Warto jednak podkreślić, że mniejsza liczba wywołań funkcji celu przy użyciu metody Lagrange'a może czynić ją bardziej efektywną w praktycznych zastosowaniach.

Wizualizacje wyników, przedstawione na wykresach zmiany objętości zbiorników *A* i *B* oraz temperatury w zbiorniku *B*, wskazują na niemal identyczne przebiegi obu metod, co potwierdza niewielkie różnice między nimi. Minimalne różnice są zauważalne jedynie w tabeli wyników, jednak nie mają one istotnego wpływu na całościową ocenę skuteczności zastosowanych metod.