Kierunek Inżynieria Obliczeniowa	**Temat laboratoriów** Ćwiczenie 3 - Optymalizacja funkcji wielu zmiennych metodami gradientowymi	**Data ćwiczenia** 27.11.2024, 04.12.2024
Przedmiot Optymalizacja	Karolina Kurowska Szymon Majdak Amelia Nalborczyk	**Grupa** 2

Cel

Celem ćwiczenia jest zapoznanie się z gradientowymi metodami optymalizacji poprzez ich implementację oraz wykorzystanie do wyznaczenia minimum podanej funkcji celu.

Przebieg ćwiczenia

Zadanie 1 funkcja testowa celu

Na ćwiczeniach zaimplementowano algorytmy metod gradientowych:

1. Metoda najszybszego spadku

```
solution SD(matrix(*ff)(matrix, matrix, matrix), matrix(*gf)(matrix, matrix, matrix), matrix x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        ofstream file("lab_04_wykresowe_sd.csv");
        if (!file.is_open())throw string("Nie da sie otworzyc pliku csv");
        file << "x0, x1\n";


        solution::clear_calls();
        solution X0, Xi;
        X0.x = x0;
        int n = get_len(x0);
        matrix di(n, 1), P(n, 2);
        solution h;
        double* exp_result;
        while (true) {
            di = -X0.grad(gf, ud1, ud2);

            if (h0 == 2137) {
                P.set_col(X0.x, 0);
                P.set_col(di, 1);
                exp_result = expansion(ff, 0, 1, 1.2, Nmax, ud1, P);
                h = golden(ff, exp_result[0], exp_result[1], epsilon, Nmax, ud1, P);
                Xi.x = X0.x + h.x * di;
            }
            else {
                Xi.x = X0.x + h0 * di;
            }

            file << Xi.x(0) << "," << Xi.x(1) << "\n";
            //cout << X1.x(0) << endl;
            //cout << X1.x(1) << endl;

            if (solution::g_calls > Nmax) {
                Xi.flag = -1;
                cout << "Error: f_calls > Nmax" << endl;
                break;
            }
            if (solution::g_calls > Nmax) {
                Xi.flag = -1;
                cout << "Error: g_calls > Nmax" << endl;
                break;
            }

            if (norm(Xi.x - X0.x) < epsilon) {
                Xi.fit_fun(ff, ud1, ud2);
                Xi.flag = 1;
                file.close();
                return Xi;
            }
            X0 = Xi;
        }
    }
    catch (string ex_info)
    {
        throw ("solution SD(...):\n" + ex_info);
    }
}
```

2. Metoda gradientów sprzężonych

```
solution CG(matrix(*ff)(matrix, matrix, matrix), matrix(*gf)(matrix, matrix, matrix), matrix x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        ofstream file("lab_04_wykresowe_cg.csv");
        if (!file.is_open())throw string("Nie da sie otworzyc pliku csv");
        file << "x0, x1\n";


        //cout << x0(0) << endl;
        //cout << x0(1) << endl;
        solution::clear_calls();
        int n = get_len(x0);
        solution X0, Xi;
        X0.x = x0;
        matrix d0(n, 1), di(n, 1), P(n, 2);
        solution h;
        double* exp_result, beta;
        d0 = -X0.grad(gf, ud1, ud2);
        di = d0;
        while (true) {
            if (h0 == 2137) {
                P.set_col(X0.x, 0);
                P.set_col(di, 1);
                exp_result = expansion(ff, 0, 1, 1.2, Nmax, ud1, P);
                h = golden(ff, exp_result[0], exp_result[1], epsilon, Nmax, ud1, P);
                Xi.x = X0.x + h.x * di;
            }
            else {
                Xi.x = X0.x + h0 * di;
            }

            file << Xi.x(0) << "," << Xi.x(1) << "\n";
            //cout << X1.x(0) << endl;
            //cout << X1.x(1) << endl;

            if (solution::g_calls > Nmax) {
                Xi.flag = -1;
                cout << "Error: f_calls > Nmax" << endl;
                break;
            }
            if (solution::g_calls > Nmax) {
                Xi.flag = -1;
                cout << "Error: g_calls > Nmax" << endl;
                break;
            }

            if (norm(Xi.x - X0.x) < epsilon) {
                Xi.fit_fun(ff, ud1);
                Xi.flag = 1;
                file.close();
                return Xi;
            }
            Xi.grad(gf);
            beta = pow(norm(Xi.g), 2) / pow(norm(X0.g), 2);
            di = -Xi.g + beta * d0;
            d0 = di;
            X0 = Xi;
        }
    }
    catch (string ex_info)
    {
        throw ("solution CG(...):\n" + ex_info);
    }
}
```

3. Metoda Newtona

```
solution Newton(matrix(*ff)(matrix, matrix, matrix), matrix(*gf)(matrix, matrix, matrix),
    matrix(*Hf)(matrix, matrix, matrix), matrix x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        ofstream file("lab_04_wykresowe_newton.csv");
        if (!file.is_open())throw string("Nie da sie otworzyc pliku csv");
        file << "x0, x1\n";

        //cout << x0(0) << endl;
        //cout << x0(1) << endl;
        solution::clear_calls();
        int n = get_len(x0);
        solution X0, Xi;
        X0.x = x0;
        matrix d(n, 1), P(n, 2);
        solution h;
        double* exp_result;
        while (true) {
            X0.grad(gf);
            X0.hess(Hf);
            d = -inv(X0.H) * X0.g;
            if (h0 == 2137) {
                P.set_col(X0.x, 0);
                P.set_col(d, 1);
                exp_result = expansion(ff, 0, 1, 1.2, Nmax, ud1, P);
                h = golden(ff, exp_result[0], exp_result[1], epsilon, Nmax, ud1, P);
                Xi.x = X0.x + h.x * d;
            }
            else {
                Xi.x = X0.x + h0 * d;
            }

            file << Xi.x(0) << "," << Xi.x(1) << "\n";
            //cout << X1.x(0) << endl;
            //cout << X1.x(1) << endl;

            if (solution::g_calls > Nmax) {
                Xi.flag = -1;
                cout << "Error: f_calls > Nmax" << endl;
                break;
            }
            if (solution::g_calls > Nmax) {
                Xi.flag = -1;
                cout << "Error: g_calls > Nmax" << endl;
                break;
            }
            if (solution::H_calls > Nmax) {
                Xi.flag = -1;
                cout << "Error: H_calls > Nmax" << endl;
                break;
            }

            if (norm(Xi.x - X0.x) < epsilon) {
                Xi.fit_fun(ff, ud1);
                Xi.flag = 1;
                file.close();
                return Xi;
            }
            X0 = Xi;
        }
    }
    catch (string ex_info)
    {
        throw ("solution Newton(...):\n" + ex_info);
    }
}
```

4. Metoda złotego podziału

```
solution golden(matrix(*ff)(matrix, matrix, matrix), double a, double b, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        solution A, B, C, D;

        double alfa = (sqrt(5) - 1) / 2;
        A.x = a;
        B.x = b;
        C.x = B.x - alfa * (B.x - A.x);
        D.x = A.x + alfa * (B.x - A.x);

        C.fit_fun(ff, ud1, ud2);
        D.fit_fun(ff, ud1, ud2);

        while (true) {
            if (C.y < D.y) {
                B = D;
                D = C;
                C.x = B.x - alfa * (B.x - A.x);
                C.fit_fun(ff, ud1, ud2);
            }
            else {
                A = C;
                C = D;
                D.x = A.x + alfa * (B.x - A.x);
                D.fit_fun(ff, ud1, ud2);
            }

            if (solution::f_calls > Nmax) {
                A.flag = -1;
                cout << "Error: f_calls > Nmax" << endl;
                break;
            }

            if (B.x - A.x < epsilon) {
                A.x = (A.x + B.x) / 2;
                A.fit_fun(ff, ud1, ud2);
                A.flag = 1;
                return A;
            }
        }

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

5. Funkcja testowa

```
matrix ffT4(matrix x, matrix ud1, matrix ud2) {
    matrix y;

    if (isnan(ud2(0, 0))) {
        y = pow(x(0) + 2 * x(1) - 7, 2) + pow(2 * x(0) + x(1) - 5, 2);
    }
    else {
        matrix trans_x = x;

        while (!isnan(ud2(0, 0))) {
            trans_x = ud2[0] + trans_x * ud2[1];

            if (isnan(ud1(0, 0))) {
                break;
            }
            ud2 = ud1;
        }
        y = pow(trans_x(0) + 2 * trans_x(1) - 7, 2) + pow(2 * trans_x(0) + trans_x(1) - 5, 2);
    }
    return y;
}
```

6. Funkcja gradientu

```
matrix gradient(matrix x, matrix ud1, matrix ud2) {
    matrix G(2, 1);

    G(0) = x(0) * 10 + 8 * x(1) - 34; //pochodna cząstkowa względem x1
    G(1) = x(0) * 8 + 10 * x(1) - 38; //pochodna cząstkowa względem x2

    return G;
}
```

7. Funkcja hesjanu

```
matrix hesjan(matrix x, matrix ud1, matrix ud2) {
    matrix H(2, 2); //macierz pochodnych cząstkowych drugiego rzędu

    H(0, 0) = 10;
    H(1, 0) = 8;
    H(0, 1) = 8;
    H(1, 1) = 10;

    return H;
}
```

Wykonujemy zadanie polegające na 100-krotnej optymalizacji każdego z algorytmów dla kroku 0,05. Zadanie powtarzamy dla kroku 0,12 oraz dla zmiennej wartości kroku wyznaczonej na podstawie metody złotego podziału. Wyniki zamieszczam w Excelu w arkuszu Tabela 1. Wartości średnie wyników zamieszczamy w arkuszu Tabela 2 oraz poniżej.

Długość kroku	Metoda najszybszego spadku					Metoda gradientów sprzężonych					Metoda Newtona					
	x_1*	x_2*	y*	f_calls	g_calls	x_1*	x_2*	y*	f_calls	g_calls	x_1*	x_2*	y*	f_calls	g_calls	H_calls
0,05	1,0009	2,9991	0,0001	1,000	58,2000	1,0002	2,9998	0,0000	1,000	22,6300	0,9974	2,9941	0,0015	1,000	115,9400	115,9400
0,12	-	-	-	1,000	10001,0000	0,9999	3,0001	0,0000	1,000	1124,2500	0,9990	2,9978	0,0002	1,000	54,3200	54,3200
M. zk.	1,0001	2,9999	0,0000	370,6000	16,8000	0,9999	3,0001	0,0000	116,0600	5,2300	1,0000	3,0000	0,0000	59,6000	2,9300	2,9300

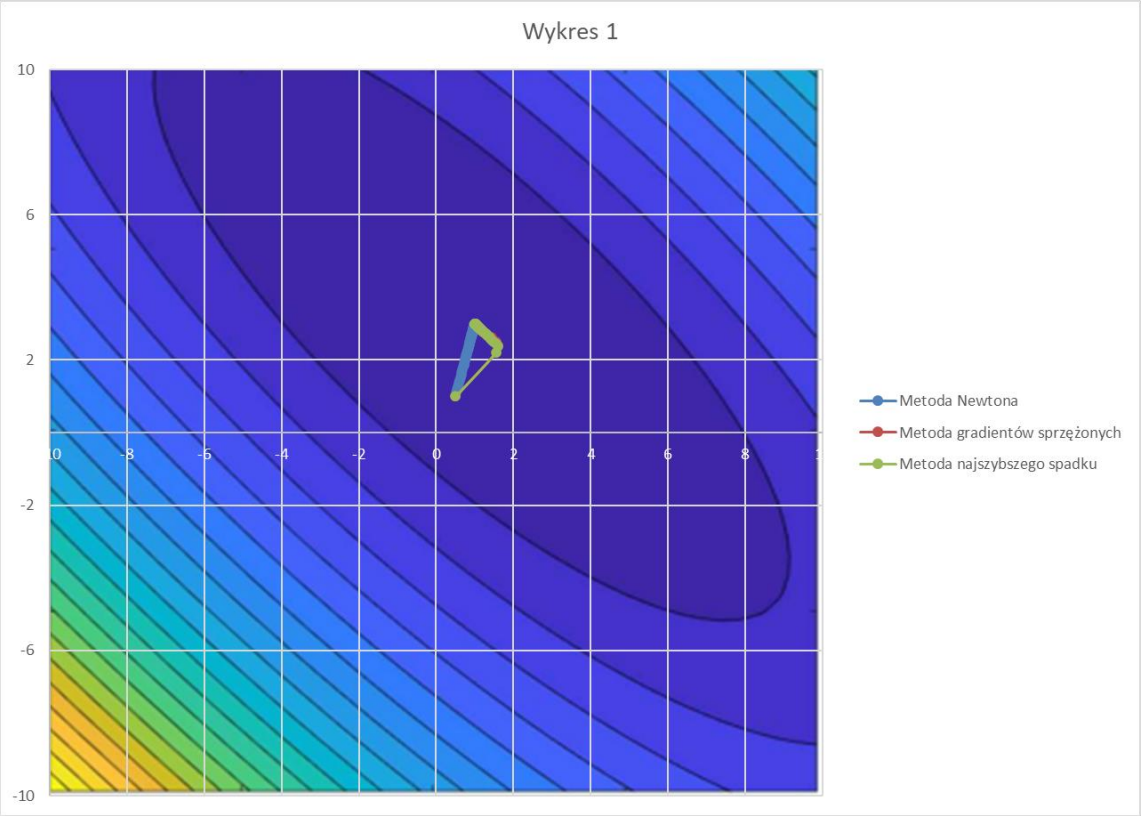
Docelowe wartości zmiennych x1∗, x2∗ oraz y∗ są zbliżone do x1∗=1, x2∗=3, oraz y∗=0, co oznacza, że wszystkie metody były w stanie znaleźć zbliżone minimum problemu. Dla metody najszybszego spadku przy długości kroku 0.05 liczba wywołań funkcji celu oraz liczba wywołań gradientu była stosunkowo niska. Przy długości kroku 0.12 niektóre wyniki są "brakujące", co może oznaczać, że metoda nie zbiegała do rozwiązania przy tej długości kroku. Metoda gradientów sprzężonych osiąga bardzo dobre wyniki, a wartości liczby wywołań funkcji celu oraz liczby wywołań gradientu są znacznie mniejsze niż dla metody najszybszego spadku. Metoda Newtona osiąga wyniki bardzo bliskie

rzeczywistego minimum, jednak liczba wywołań hesjanu i liczba wywołań gradientu wskazują na większy koszt obliczeniowy w porównaniu do gradientów sprzężonych.

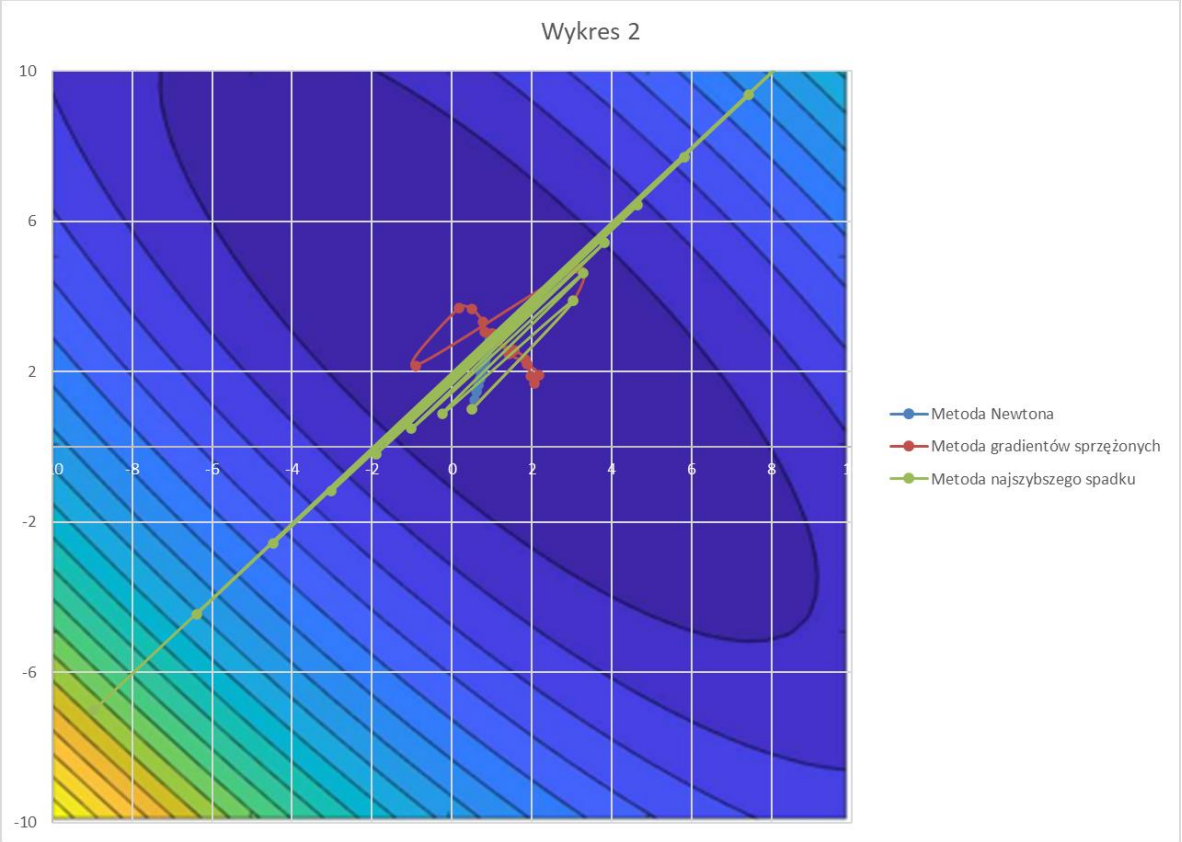
Podsumowując metoda Newtona oraz gradienty sprzężone wyróżniają się największą precyzją, wymaga jednak dodatkowego obliczenia hesjanu, co zwiększa jej koszt obliczeniowy. Metoda gradientów sprzężonych jest najbardziej efektywna pod względem liczby wywołań funkcji celu i gradientu. Przy większym kroku (0.12) metoda najszybszego spadku może nie zbiegać prawidłowo do rozwiązania, co podkreśla znaczenie właściwego doboru długości kroku.

Następnie dla jednego wybranego punktu startowego zapisujemy wartości znalezionego x dla każdej iteracji i wykonujemy 6 wykresów wyznaczenia rozwiazania optymalnego na wykresie poziomic. Wykresy znajdują się w arkuszu Wykresy oraz poniżej.

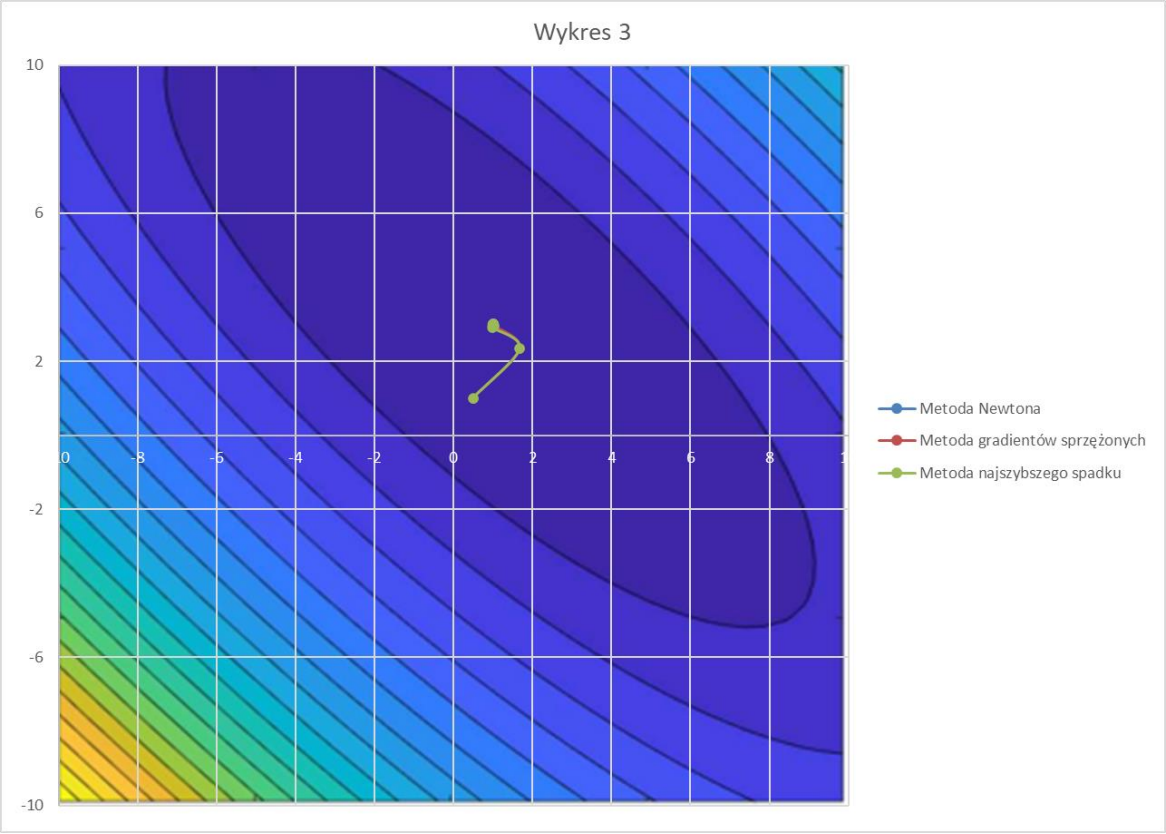
Wykres 1: Wykres dla długości kroku równej 0,05 rozwiązania otrzymane każdą z metod



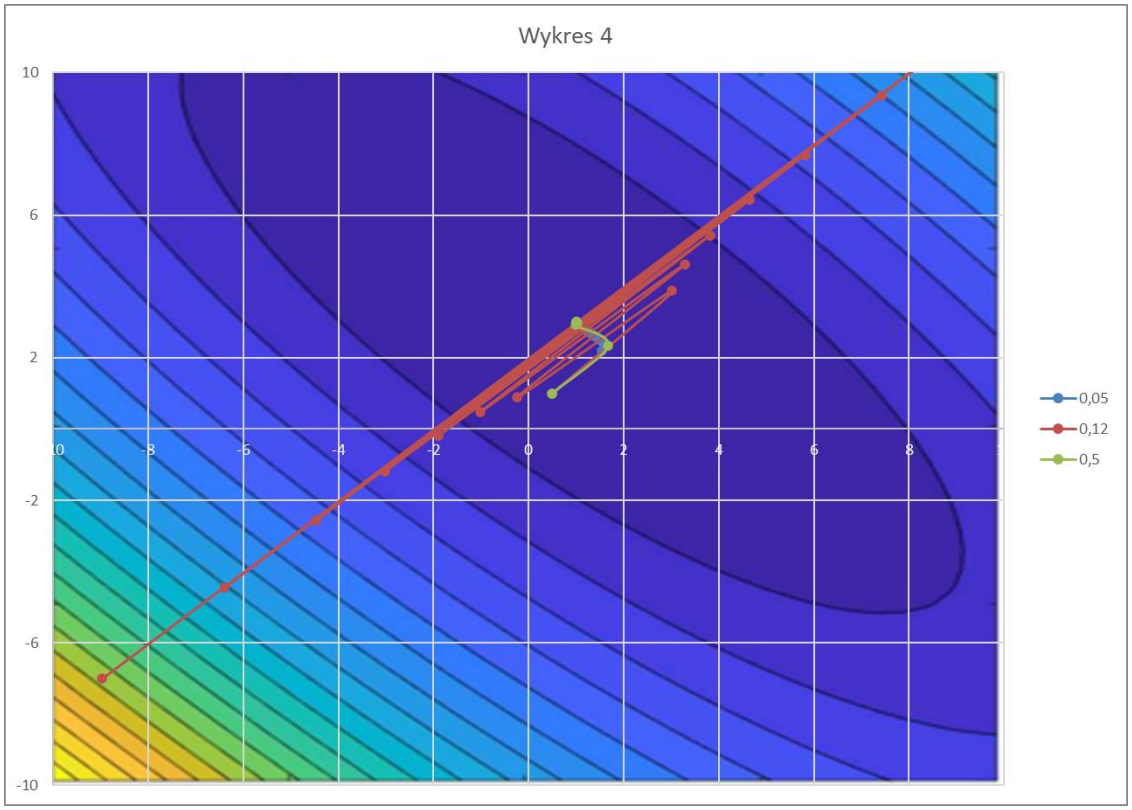
Wykres 2: Wykres dla długości kroku równej 0,12 rozwiązania otrzymane każdą z metod



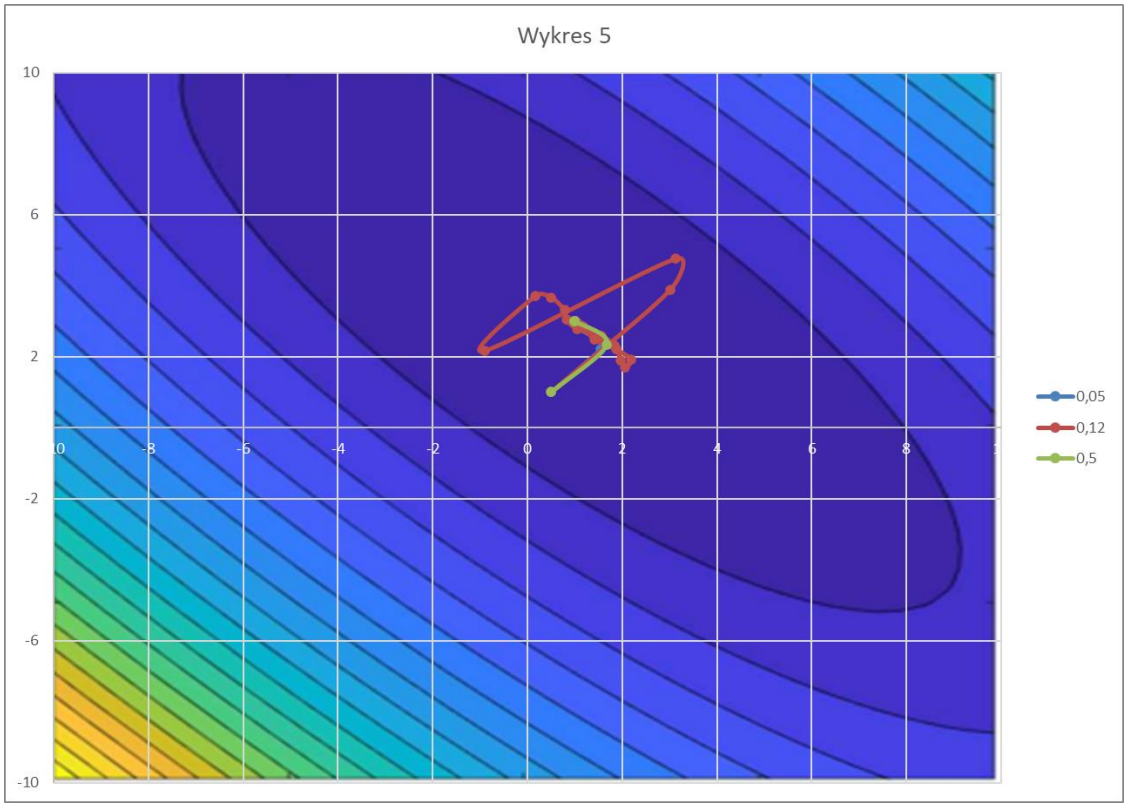
Wykres 3: Wykres dla wersji zmiennokrokowej rozwiązania otrzymane każdą z metod



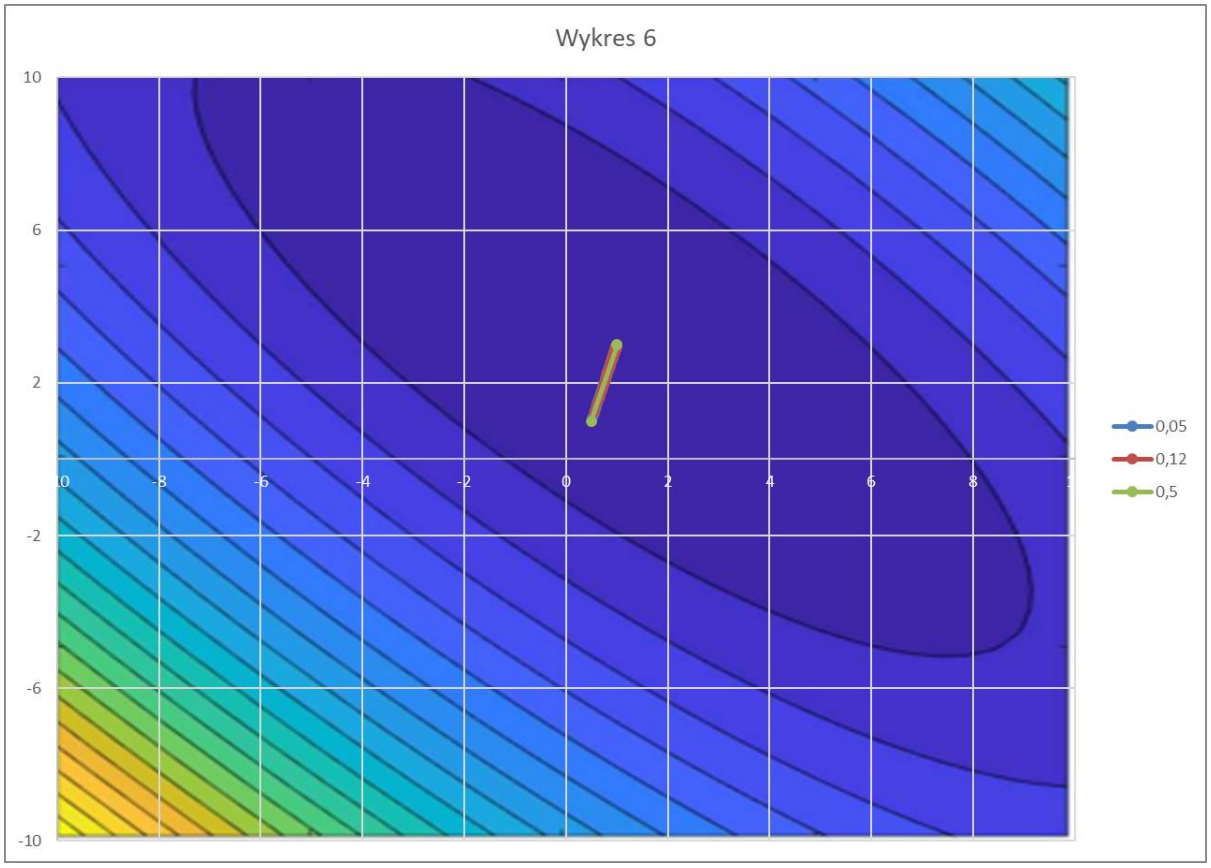
Wykres 4: Wykres dla metody najszybszego spadku rozwiązania otrzymane dla każdej długości kroku



Wykres 5: Wykres dla metody gradientów sprzężonych rozwiązania otrzymane dla każdej długości kroku



Wykres 6: Wykres dla metody Newtona rozwiązania otrzymane dla każdej długości kroku



Zadanie 2 problem rzeczywisty

Następnym zadaniem jest przeprowadzenie symulacji. W tym zadaniu optymalizacji analizujemy proces przyjęcia kandydatów na uczelnię na podstawie ocen z dwóch przedmiotów. Dane wejściowe obejmują oceny oraz decyzje o przyjęciu lub odrzuceniu kandydatów. Celem jest stworzenie modelu, który przewiduje, czy kandydat zostanie przyjęty, bazując na tych ocenach.

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

```
matrix diff_data(matrix x, matrix ud1, matrix ud2){
  const int n = get_len(x); //Liczba współrzędnych wektora gradientu
  const int m = 100; //Liczba optymalizacji

  matrix G(n, 1);
  static matrix X(n, m);
  static matrix Y(1, m);

  //Dane do plików pierwsze wywołanie
  if(solution::g_calls == 1){
    ifstream in;

    in.open("XData.txt");
    if( in.is_open() ){
      in >> X;
      in.close();
    }

    in.open("YData.txt");
    if(in.is_open()){
      in >> Y;
      in.close();
    }
  }

  for(int i = 0; i < n; ++i){
    double sum = 0.0;

    for(int j = 0; j < m; ++j){
      double h = m2d(trans(x) * X[j]);
      h = 1 / (1 + exp(-h)); //Funkcja hipotezy
      sum = sum + (X(i, j) * (h - Y(0, j)));
    }

    G(i) = sum / m;
  }

  return G;
}
```

Do rozwiązania tej części implementujemy funkcje celu, zamieszczoną poniżej:

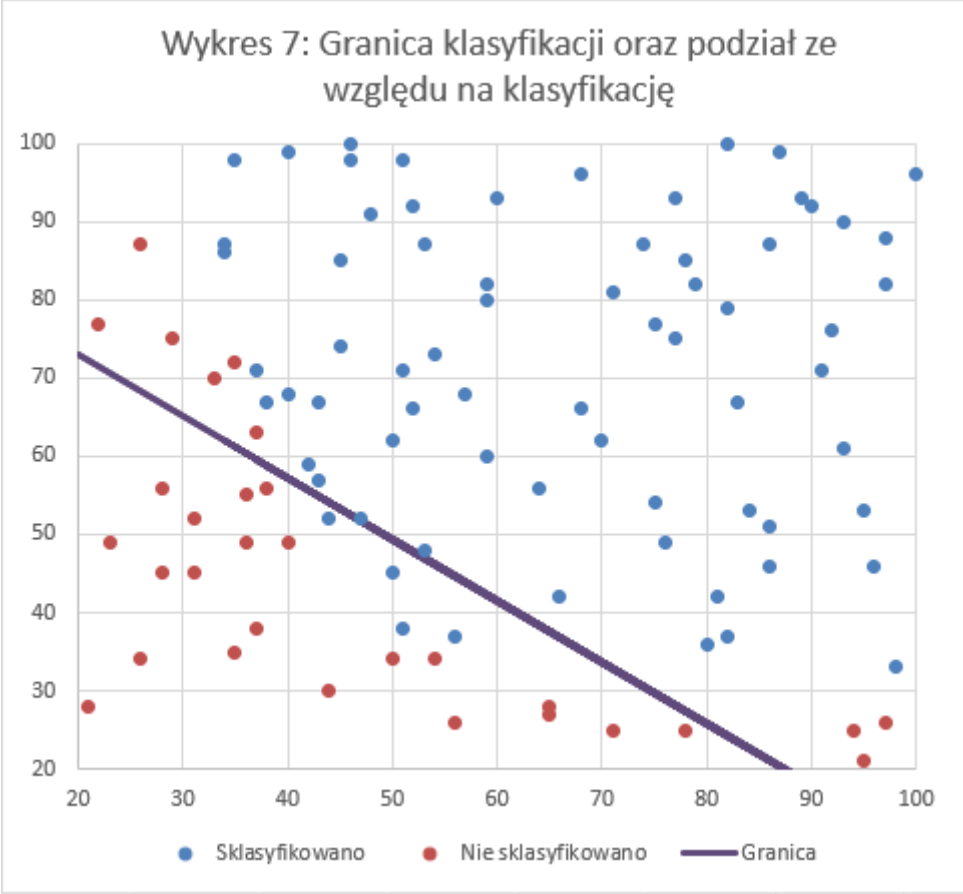
```
matrix ff4R( matrix x, matrix ud1, matrix ud2 ){
  const int n = get_len(x); //Liczba współrzędnych wektora gradientu
  const int m = 100; //Liczba optymalizacji

  matrix Y;
  static matrix X(n, m);
  static matrix Y(1, m);

  //Dane do plików pierwsze wywołanie
  if( solution::f_calls == 1 )
  {
    ifstream in;

    in.open( "XData.txt" );
    if( in.is_open() ){
      in >> X;
      in.close();
    }

    in.open( "YData.txt" );
    if(in.is_open()){
      in >> Y;
      in.close();
    }
  }

  double h;
  Y = 0;

  for(int i = 0; i < m; ++i){
    h = m2d(trans(x) * X[i]);
    h = 1.0 / (1.0 + exp(-h)); //Funkcja hipotezy
    Y -= Y(0, i) * log(h) + (1 - Y(0, i)) * log(1 - h); //Koszt logarytmiczny
  }

  Y = Y/ m; //Normalizacja
  return Y;
}
```

W tym celu wykonaliśmy optymalizację dla metody gradientów sprzężonych dla trzech różnych wartości kroku (0,1; 0,001; 0,0001) zaczynając od punktu [0;0;0].

Wyniki tej optymalizacji zamieszczamy w arkuszu Klasyfikator oraz poniżej.

Długość kroku	Metoda gradientów sprzężonych					
	θ_0^*	θ_1^*	θ_2^*	$J(\theta^*)$	$P(\theta^*)$	g_calls
0,01	-1649,59824	15216,15732	-10954,56077	8,749823339	62	1001
0,001	-7,02483719	0,062238751	0,079270621	0,293966551	87	1001
0,0001	-2,025860179	0,020427792	0,032311862	0,419454612	76	1001

Wybierając wartość kroku, dla której procent poprawnego sklasyfikowania był najlepszy, sporządziliśmy wykres przedstawiający liniową granicę podziału, na osoby sklasyfikowane oraz niesklasyfikowane. Wykres ten razem z danymi znajduje się w arkuszu Wykres, a wykres znajdziemy również poniżej.



Wykres przedstawia proces klasyfikacji kandydatów na podstawie ocen z dwóch przedmiotów, gdzie ocena z jednego przedmiotu znajduje się na osi poziomej (X), a ocena z drugiego przedmiotu na osi pionowej (Y). Punkty na wykresie są podzielone na dwie grupy: sklasyfikowanych kandydatów (oznaczonych niebieskimi kropkami) oraz niesklasyfikowanych kandydatów (oznaczonych czerwonymi kropkami). Widoczna jest również granica klasyfikacji w postaci fioletowej linii, która rozdziela obszar na dwie części.

Granica ta wskazuje, które kombinacje ocen prowadzą do decyzji o przyjęciu kandydata. Punkty znajdujące się powyżej granicy odpowiadają kandydatom sklasyfikowanym pozytywnie, czyli przyjętym, natomiast punkty poniżej granicy to kandydaci niesklasyfikowani, czyli odrzuceni. Model liniowy skutecznie rozdzielił dwie grupy, co wskazuje na możliwość stosowania metod gradientowych do klasyfikacji liniowej w prostych problemach decyzyjnych.

Wnioski

W ramach ćwiczeń skutecznie zaimplementowano oraz przetestowano gradientowe metody optymalizacji. Analiza wyników pozwoliła zidentyfikować ich zalety i ograniczenia. Metody gradientowe znalazły szerokie zastosowanie zarówno w minimalizacji funkcji testowych, jak i w praktycznym problemie klasyfikacji kandydatów, potwierdzając swoją użyteczność w procesach optymalizacji i modelowania decyzyjnego.