AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Sprawozdanie 4

Optymalizacja z ograniczeniami funkcji wielu zmiennych metodami gradientowymi

Autorzy: *Paulina Grabowska*
Filip Rak
Arkadiusz Sala

Kierunek studiów: *Inżynieria Obliczeniowa*

Kraków, 2024

Spis treści

Cel ćwiczenia 3

Optymalizowane problemy 3

Testowa funkcja celu 3

Problem rzeczywisty 4

Implementacja metod optymalizacji 7

Metody stałokrokowe 7

Metoda najszybszego spadku 8

Metoda gradientów sprzężonych 9

Metoda Newtona 10

Metoda zmiennokrokowa 11

Opracowanie wyników 13

Testowa funkcja celu 13

Problem rzeczywisty 21

Wnioski 24

Cel ćwiczenia

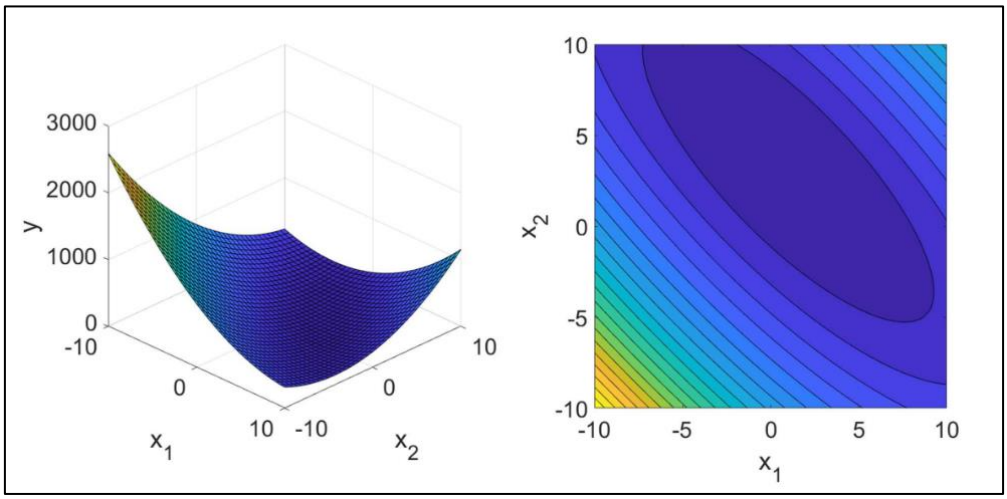
Celem przeprowadzonego ćwiczenia było pogłębienie wiedzy na temat gradientowych metod optymalizacji – metody najszybszego spadku, metody gradientów sprzężonych i metody Newtona oraz metody złotego podziału. Zostały one zaimplementowane i zastosowane do rozwiązywania problemów optymalizacyjnych dla funkcji wielu zmiennych, 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*. Funkcja ta ma wiele minimów globalnych znajdujących się na elipsie rozciągającej się mniej więcej od punktu (9, -5), do punktu (-4, 11). Obrany punkt startowy będzie należał do przedziału [-10, 10] (dla obu zmiennych).

$$f(x_1, x_2) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 \tag{1}$$



Rys.1. Wykres funkcji (1)

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

```
matrix ff4T(matrix x, matrix ud1, matrix ud2)
{
	if (isnan(ud2(0)))
		return (pow(x(0) + 2.0 * x(1) - 7.0) + pow(2.0 * x(0) + x(1) - 5.0));
	else
		return (pow((ud1(0) + x(0) * ud2(0)) + 2.0 * (ud1(1) + x(0) * ud2(1)) - 7.0)
			+ pow(2.0 * (ud1(0) + x(0) * ud2(0)) + (ud1(1) + x(0) * ud2(1)) -
5.0));

}
```

Fragment kodu 1: funkcja licząca testową funkcję celu

Dla testowej funkcji celu dodano również funkcje liczące gradient oraz hesjan testowej funkcji celu. Funkcja *gradff4T* oblicza gradient (wektor pochodnych cząstkowych) pewnej funkcji celu w punkcie *x*, a funkcja *Hff4t* oblicza hesjan funkcji (1).

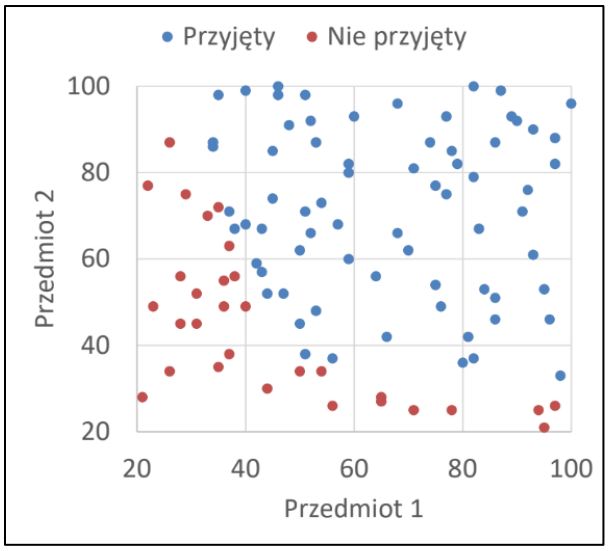
```
matrix gradff4T(matrix x, matrix ud1, matrix ud2)
{

	return matrix(2, new double[2] {
				10.0 * x(0) + 8.0 * x(1) - 34,
				8.0 * x(0) + 10.0 * x(1) - 38,
		});
}

matrix Hff4T(matrix x, matrix ud1, matrix ud2)
{
	matrix rt(2, 2);
	rt(0, 0) = 10.0;
	rt(0, 1) = 8.0;
	rt(1, 0) = 8.0;
	rt(1, 1) = 10;

	return rt;
}
```

Fragment kodu 2: funkcje liczące gradient i hesjan testowej funkcji celu

Problem rzeczywisty

Rozważany problem rzeczywisty polega na znalezieniu najlepszych parametrów klasyfikatora, który na podstawie uzyskanych ocen będzie przewidywał, czy dana osoba zostanie przyjęta na uczelnię.

W plikach *XData.txt* oraz *YData.txt* zawarto niezbędne informacje do wykonania zadania, tj. dane o ocenach z obu przedmiotów w postaci macierzy, dla każdego kandydata oraz informacja, czy kandydat został przyjęty czy odrzucony. Wszystkie te dane zostały przedstawione na *Rys.2,* gdzie czerwonymi kropkami zaznaczono kandydatury odrzucone, niebieskimi przyjęte.



Rys.2. Ilustracja problemu rzeczywistego

Klasyfikator korzysta z hipotezy przedstawionej wzorem (2). We wzorze tym, wektor θ zawiera szukane parametry klasyfikatora, natomiast wektor x zawiera oceny z obu przedmiotów. Aby kandydat został przyjęty na uczelnię wartość hipotezy musi wynosić co najmniej 0,5.

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}} \tag{2}$$

```
1. double get_h_l4(matrix theta, matrix X, int col)
2. {
3.         // Get z = thetha0 * 1 + theta1 * x1 + theta2 * x2
4.         double z = theta(0, 0) * X(0, col) + theta(1, 0) * X(1, col) + theta(2, 0) * X(2, col);
5.         double h = 1.0 / (1.0 + exp(-z));
6.
7.         return h;
8. }
```

Fragment kodu 3: funkcja licząca wzór (2)

Proces znajdowania najlepszych parametrów klasyfikatora polega na minimalizowaniu funkcji kosztu opisanej wzorem (3), natomiast jej pochodna cząstkowa opisana jest wzorem (4):

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left(y^i \cdot ln\left(h_\theta\left(x^i\right)\right) + \left(1 - y^i\right) \cdot ln\left(1 - h_\theta\left(x^i\right)\right)\right. \tag{3}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^i\right) - y_j^i\right) \tag{4}$$

W powyższych wzorach m to liczba danych, n to liczba współrzędnych wektora gradientu, a zmienna j należy do liczb naturalnych od 1 do n. Należy jednak pamiętać o tym, że wyniki klasyfikatora nie będą w idealny sposób oddawać rzeczywistości, ponieważ szukaną przez

klasyfikator granicą jest linia prosta, a żeby oddawać prawdziwe warunki zadania należałoby znaleźć zależność kwadratową czy nawet wielomianową.

```
matrix get_cost(matrix theta, matrix Y, matrix X)
{
	// Get size
	int* size_Y = get_size(Y);
	int m = size_Y[1];
	delete[] size_Y;

	double cost = 0.0;
	for (int j = 0; j < m; ++j)
	{
		double h = get_h_l4(theta, X, j);

		// Add to the cost
		if (Y(0, j) == 1)
		{
			// Slightly increase the value to avoid log(0)
			cost += -log(h + 1e-15);
		}
		else
		{
			cost += -log(1.0 - h + 1e-15);
		}
	}

	// Average cost value
	cost /= static_cast<double>(m);

	// Return the result as matrix
	return cost;
}

matrix get_gradient(matrix theta, matrix Y, matrix X)
{
	// Get size
	int* size_Y = get_size(Y);
	int m = size_Y[1];
	delete[] size_Y;

	// Initialize gradient as 3x1 matrix with zeros
	matrix grad(3, 1, 0.0);

	for (int j = 0; j < m; ++j)
	{
		double h = get_h_l4(theta, X, j);

		// Get the error
		double error = h - Y(0, j);

		// Update the graident
		grad(0, 0) += error * X(0, j);		// For theta0 (bias)
		grad(1, 0) += error * X(1, j);		// For theta1
		grad(2, 0) += error * X(2, j);		// For theta2
	}

	// Average gradient value
	for (int j = 0; j < 3; ++j)
	{
		grad(j, 0) /= static_cast<double>(m);
	}

	return grad;
}
```

Fragment kodu 4: funkcja licząca wzory (3) i (4)

Dodatkowo, zdefiniowano funkcję *get_accuracy*, która oblicza dokładność modelu klasyfikacyjnego na podstawie podanych danych wejściowych i etykiet. Dla każdej próbki porównywany jest wynik otrzymany za pomocą funkcji *get_h_l4* (*Fragment kodu 3*) z rzeczywistą informacją dotyczącą przyjęcia bądź odrzucenia kandydata.

```
1. matrix get_accuracy(matrix theta, matrix X, matrix Y, int cols)
 2. {
 3.       int guessed = 0;
 4.       for (int j = 0; j < cols; j++)
 5.       {
 6.                double h = get_h_l4(theta, X, j);
 7.
 8.                int prediction = (h >= 0.5) ? 1 : 0;
 9.                if (prediction == Y(0, j))
10.                {
11.                         guessed++;
12.                }
13.       }
14.
15.       return (static_cast<double>(guessed) / cols) * 100.0;
16. }
17.
```

Fragment kodu 5: funkcja licząca dokładność klasyfikatora

Implementacja metod optymalizacji

Metody stałokrokowe

Kierunek di wyznaczany będzie dla trzech różnych metod: metody najszybszego spadku (5), metody gradientów sprzężonych (6) oraz metody Newtona (7).

$$d^{(i)} = -\nabla f\left(x^{(i)}\right) \tag{5}$$

$$d^{(i)} = -\nabla f\left(x^{(i)}\right) + \beta \cdot d^{(i-1)} \qquad \text{dla } \beta = \frac{\left(\left\|\nabla f\left(x^{(i)}\right)\right\|_2\right)^2}{\left(\left\|\nabla f\left(x^{(i-1)}\right)\right\|_2\right)^2} \tag{6}$$

$$d^{(i)} = -H^{-1}\left(x^{(i)}\right)\nabla f\left(x^{(i)}\right) \tag{7}$$

Metoda najszybszego spadku

Funkcja *SD* implementuje metodę najszybszego spadku, iteracyjnie minimalizując funkcję celu z użyciem gradientu i kroku optymalizacyjnego. W kolejnych iteracjach aktualizuje rozwiązanie, oblicza kierunek optymalizacji jako przeciwny do gradientu, wyznacza długość kroku (*h0)* dynamicznie przy pomocy metody ekspansji i złotego podziału, aż do osiągnięcia zbieżności lub maksymalnej liczby iteracji.

```
 1. solution SD(matrix(*ff)(matrix, matrix, matrix), matrix(*gf)(matrix, matrix, matrix), matrix
x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)
 2. {
 3.       try
 4.       {
 5.                solution xi(x0), x_i;
 6.                xi.flag = 0;
 7.                bool h0_c = false;
 8.                if (h0 == 0.0)
 9.                         h0_c = true;
10.                int i = 0;
11.                bool save_prev_x = !isnan(ud1(0));
12.
13.                do
14.                {
15.                         matrix di = -gf(xi.x, NAN, NAN);
16.                         ++solution::g_calls;
17.                         if (h0_c) {
18.                                  double* range = expansion(ff,0,10.0,2.0, Nmax, xi.x, di);
19.                                  h0 = m2d(golden(ff,0,range[1],epsilon,Nmax, xi.x, di).x);
20.                                  delete[] range;
21.                         }
22.
23.                         x_i = xi;
24.                         if (save_prev_x)
25.                         {
26.                                  if (!i) {
27.                                           xi.ud = matrix(x_i.x);
28.                                           i++;
29.                                  }
30.                                  else {
31.                                           xi.ud.add_col(x_i.x);
32.                                  }
33.                         }
34.                         xi.x = xi.x + di * h0;
35.
36.                         xi.fit_fun(ff,NAN,NAN);
37.
38.                         if (solution::f_calls > Nmax) {
39.                                  xi.flag = -2;
40.                                  break;
41.                         }
42.
43.                } while (norm(xi.x - x_i.x) >= epsilon);
44.
45.                return xi;
46.       }
47.       catch (string ex_info)
48.       {
49.                throw ("solution SD(...):\n" + ex_info);
50.       }
51. }
```

Fragment kodu 6: metoda najszybszego spadku

Metoda gradientów sprzężonych

Funkcja *CG* implementuje metodę optymalizacji gradientów sprzężonych. Działa iteracyjnie, obliczając gradient, kierunek optymalizacji jako kombinację gradientu i poprzedniego kierunku (z wagą określoną współczynnikiem β), oraz dynamicznie ustalając długość kroku (*h0*). Proces kończy się, gdy osiągnięta zostanie zbieżność rozwiązania lub przekroczona zostanie maksymalna liczba iteracji *Nmax*.

```
 1. solution CG(matrix(*ff)(matrix, matrix, matrix), matrix(*gf)(matrix, matrix, matrix), matrix
x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)
 2. {
 3.       try
 4.       {
 5.                 solution xi(x0), x_i;
 6.                 xi.flag = 0;
 7.                 bool h0_c = false;
 8.                 if (h0 == 0.0)
 9.                           h0_c = true;
10.                 bool save_prev_x = !isnan(ud1(0));
11.                 int i = 0;
12.                 int k = 0;
13.                 matrix di;
14.                 double x_i_g_pow_norm =0.0;
15.                 do
16.                 {
17.                           // Debug output
18.                           if (solution::f_calls % 10000 == 0 && solution::f_calls != 0)
19.                                     std::cout << "|" << solution::f_calls / 10000 << "|";
20.
21.                           xi.grad(gf, ud1, ud2);
22.                           double xi_g_pow_norm = pow(norm(xi.g), 2);
23.
24.                           if (i != 0) {
25.
26.                                     double beta = xi_g_pow_norm / x_i_g_pow_norm;
27.                                     if (x_i.y < xi.y)
28.                                     {
29.                                               //std::cout << "i: " << i << " " << x0 <<
std::endl;
30.                                               di = -x_i.g;
31.                                     }
32.                                     else
33.                                     di = -xi.g + di * beta;
34.                           }
35.                           else {
36.                                     di = -xi.g;
37.
38.                           }
39.                           i++;
40.
41.                           x_i_g_pow_norm = xi_g_pow_norm;
42.
43.                           if (h0_c) {
44.                                     double* range = expansion(ff, 0, 10.0, 2.0, Nmax, xi.x,
di);
45.                                     h0 = m2d(golden(ff, max(range[0],0.0), range[1], epsilon,
Nmax, xi.x, di).x);
46.                                     delete[] range;
47.                           }
48.
49.                           x_i.x = xi.x;
```

```
50.                             x_i.y = xi.y;
51.                             x_i.g = xi.g;
52.                             if (save_prev_x)
53.                             {
54.                                     if (!k) {
55.                                             xi.ud = matrix(x_i.x);
56.                                             k++;
57.                                     }
58.                                     else {
59.                                             xi.ud.add_col(x_i.x);
60.                                     }
61.                             }
62.
63.                             xi.x = xi.x + di * h0;
64.                             xi.fit_fun(ff, ud1, ud2);
65.
66.
67.                             if (solution::f_calls > Nmax) {
68.                                     xi.flag = -2;
69.                                     break;
70.                             }
71.
72.                     } while (norm(xi.x - x_i.x) >= epsilon);
73.
74.                     return xi;
75.             }
76.             catch (string ex_info)
77.             {
78.                     throw ("solution CG(...):\n" + ex_info);
79.             }
80. }
```

Fragment kodu 7: metoda gradientów sprzężonych

Metoda Newtona

Funkcja *Newton* implementuje metodę optymalizacji, która w każdej iteracji obliczany jest gradient oraz hesjan, które służą do wyznaczenia kierunku optymalizacji poprzez rozwiązanie układu równań liniowych.

```
 1. solution Newton(matrix(*ff)(matrix, matrix, matrix), matrix(*gf)(matrix, matrix, matrix),
 2.         matrix(*Hf)(matrix, matrix, matrix), matrix x0, double h0, double epsilon, int Nmax,
matrix ud1, matrix ud2)
 3. {
 4.         try
 5.         {
 6.                 solution xi(x0), x_i;
 7.                 xi.flag = 0;
 8.                 bool h0_c = false;
 9.                 if (h0 == 0.0)
10.                         h0_c = true;
11.                 bool save_prev_x = !isnan(ud1(0));
12.                 int i = 0;
13.                 do
14.                 {
15.                         xi.hess(Hf, NAN, NAN);
16.                         xi.grad(gf, NAN, NAN);
17.
18.                         matrix di = -inv(xi.H)*xi.g;
19.
```

```
20.                             if (h0_c) {
21.                                     double* range = expansion(ff, 0, 10.0, 2.0, Nmax, xi.x,
di);
22.                                     h0 = m2d(golden(ff, max(range[0], 0.0), range[1],
epsilon, Nmax, xi.x, di).x);
23.                                     delete[] range;
24.                             }
25.
26.                             if (solution::f_calls > Nmax)
27.                             {
28.                                     xi.flag = -2;
29.                                     break;
30.                             }
31.
32.                             x_i = xi;
33.                             if (save_prev_x)
34.                             {
35.                                     if (!i) {
36.                                             xi.ud = matrix(xi.x);
37.
38.                                             i++;
39.                                     }
40.                                     else {
41.                                             xi.ud.add_col(xi.x);
42.                                     }
43.                             }
44.
45.                             xi.x = xi.x + di * h0;
46.
47.                             xi.fit_fun(ff, NAN, NAN);
48.
49.                             solution::f_calls++;
50.                             if (solution::f_calls > Nmax) {
51.                                     xi.flag = -2;
52.                                     break;
53.                             }
54.
55.                     } while (norm(xi.x - x_i.x) >= epsilon);
56.
57.                     return xi;
58.             }
59.             catch (string ex_info)
60.             {
61.                     throw ("solution Newton(...):\n" + ex_info);
62.             }
63. }
64.
```

Fragment kodu 8: metoda Newtona

Metoda zmiennokrokowa

Dodatkowo, dla zaimplementowanych wyżej metod, oprócz stałych długości kroku będzie badane też ich zachowanie dla metody zmiennokrokowej – czyli metody złotego podziału. Punkt z większą wartością funkcji jest odrzucany, a przedział jest zawężany, aż jego długość stanie się mniejsza niż zadana precyzja *epsilon*. Wynikowy punkt optymalny to środek końcowego przedziału, który zwracany jest jako rozwiązanie. Funkcja kończy działanie, jeśli liczba wywołań funkcji celu przekroczy *Nmax*.

```
 1. solution golden(matrix(*ff)(matrix, matrix, matrix), double a, double b, double epsilon, int
Nmax, matrix ud1, matrix ud2)
 2. {
 3.       try
 4.       {
 5.                solution Xopt;
 6.                Xopt.flag = 0;
 7.                //Tu wpisz kod funkcji
 8.                static const double alpha = (sqrt(5.0) - 1.0) * .5;
 9.                double ai = a, bi = b;
10.                double ay = m2d(ff(a, ud1, ud2)), by = m2d(ff(b, ud1, ud2));
11.                solution::f_calls+= 2;
12.                double ci  = (bi - alpha * (bi - ai)), di = (ai + alpha * (bi - ai));
13.                double cy = m2d(ff(ci, ud1, ud2)), dy = m2d(ff(di, ud1, ud2));
14.                solution::f_calls += 2;
15.                do
16.                {
17.                          if (cy < dy)
18.                          {
19.                                    bi = di;
20.                                    by = dy;
21.                                    di = ci;
22.                                    dy = cy;
23.                                    ci = bi - alpha * (bi - ai);
24.                                    cy = m2d(ff(ci, ud1, ud2));
25.                                    solution::f_calls++;
26.                          }
27.                          else
28.                          {
29.                                    ai = ci;
30.                                    ay = cy;
31.                                    ci = di;
32.                                    cy = dy;
33.                                    di = ai + alpha * (bi - ai);
34.                                    dy = m2d(ff(di, ud1, ud2));
35.                                    solution::f_calls++;
36.                          }
37.                          if (solution::f_calls > Nmax)
38.                          {
39.                                    Xopt.flag = -2;
40.                                    break;
41.                          }
42.                } while (bi-ai >= epsilon);
43.                Xopt.x = matrix((ai + bi) / 2.0);
44.                return Xopt;
45.       }
46.       catch (string ex_info)
47.       {
48.                throw ("solution golden(...):\n" + ex_info);
49.       }
50. }
51.
```

Fragment kodu 9: metoda złotego podziału

Opracowanie wyników

Zadeklarowano zmienne lokalne wykorzystywane podczas późniejszego testowania i dokładność oraz maksymalną liczbę iteracji (*Fragment kodu 10*).

```
1.      // Common arguments
2.      double epsilon = 1e-3;
3.      int Nmax = 1000;
```

Fragment kodu 10: zmienne lokalne niezbędne dla działania badanych funkcji

Testowa funkcja celu

Dla każdej metody zestawiano w tabeli optymalne wartości x_1 i x_2, y oraz liczba wywołań funkcji f i g. Dla każdej metody (najszybszego spadku, gradientów sprzężonych oraz Newtona) testowano długości kroku 0,05; 0,12 oraz wartość otrzymaną za pomocą metody zmiennokrokowej (metody złotego podziału).

```
 1.      // ---------- Table 1 and Table 2 ----------
 2.
 3.      std::cout << "Solving for Table 1 and Table 2...\n";
 4.
 5.      // Open file to save test values
 6.      ofstream tfun_file1(OUTPUT_PATH + "out_1_tfun.txt");
 7.      if (!tfun_file1.good()) return;
 8.
 9.      // Init random number generator
10.      srand(time(NULL));
11.
12.      // Range for testing on both axis
13.      double max_values[2]{ 10.0, 10.0 };
14.      double min_values[2]{ -10.0, -10.0 };
15.
16.      // Set step size for testing
17.      double steps[3]{ .05, .12, .0 };
18.      {
19.                // Test 100 random points with defined number of steps
20.                for (int step_i = 0; step_i < 3; step_i++)
21.                {
22.                          for (int i = 0; i < 100; i++)
23.                          {
24.                                    matrix test = matrix(2, new double[2] {
25.                                              min_values[0] + static_cast<double>(rand()) /
RAND_MAX * (max_values[0] - min_values[0]),
26.                                                        min_values[1] +
static_cast<double>(rand()) / RAND_MAX * (max_values[1] - min_values[1])
27.                                              });
28.                                    tfun_file1 << test(0) << delimiter << test(1) <<
delimiter;
29.
30.                                    // Steepest descent metohod -> step size determines the
convergence of the function
31.                                    solution SSD = SD(ff4T, gradff4T, test, steps[step_i],
epsilon, Nmax, NAN, NAN);
32.                                    SSD.fit_fun(ff4T, NAN, NAN);
```

```
                                    tfun_file1 << SSD.x(0) << delimiter << SSD.x(1) <<
delimiter << SSD.y(0) << delimiter;
                                    tfun_file1 << SSD.f_calls << delimiter << SSD.g_calls <<
delimiter;
                                    solution::clear_calls();

                                    // Conjugate gradient method -> guarantee of convergence,
no need to calculate the Hessian matrix
                                    solution SCG = CG(ff4T, gradff4T, test, steps[step_i],
epsilon, Nmax, NAN, NAN);
                                    SCG.fit_fun(ff4T, NAN, NAN);
                                    tfun_file1 << SCG.x(0) << delimiter << SCG.x(1) <<
delimiter << SCG.y(0) << delimiter;
                                    tfun_file1 << SCG.f_calls << delimiter << SCG.g_calls <<
delimiter;
                                    solution::clear_calls();

                                    // Newton's method -> guarantee of convergence, the
Hessian matrix must be provided as additional Hf function.
                                    solution SNewton = Newton(ff4T, gradff4T, Hff4T, test,
steps[step_i], epsilon, Nmax, NAN, NAN);
                                    SNewton.fit_fun(ff4T, NAN, NAN);
                                    tfun_file1 << SNewton.x(0) << delimiter << SNewton.x(1)
<< delimiter << SNewton.y(0) << delimiter;
                                    tfun_file1 << SNewton.f_calls << delimiter <<
SNewton.g_calls << delimiter << SNewton.H_calls << delimiter;
                                    solution::clear_calls();

                                    tfun_file1 << std::endl;
                            }
                    }
                    tfun_file1.close();
            }
            try{
                    matrix test_x = matrix(2, new double[2] {
                            min_values[0] + static_cast<double>(rand()) / RAND_MAX *
(max_values[0] - min_values[0]),
                            min_values[1] + static_cast<double>(rand()) / RAND_MAX *
(max_values[1] - min_values[1])
                    });

                    try {
                            ofstream SD_tfun(OUTPUT_PATH + "out_SD_tfun.txt");
                            if (!SD_tfun.good())  throw("FILE NO GOOD");
                            for (int step_i = 0; step_i < 3; step_i++)
                            {
                                    solution SSD = SD(ff4T, gradff4T, test_x, steps[step_i],
epsilon, Nmax, 0, NAN);
                                    int* size_sd = get_size(SSD.ud);
                                    for (int i = 0; i < size_sd[1]; i++)
                                    {
                                            SD_tfun << SSD.ud(0, i) << delimiter <<
SSD.ud(1, i) << std::endl;
                                    }
                                    SD_tfun << SSD.x(0) << delimiter << SSD.x(1) <<
std::endl;
                                    SD_tfun << std::endl;
                                    delete[] size_sd;
                                    solution::clear_calls();
                            }
                            SD_tfun.close();
                    }
                    catch (string ex_info)
                    {
                            throw ("SD_tfun:\n" + ex_info);
                    }

                    try {
                            ofstream CG_tfun(OUTPUT_PATH + "out_CG_tfun.txt");
                            if (!CG_tfun.good()) throw("FILE NO GOOD");
```

```
 88.                         for (int step_i = 0; step_i < 3; step_i++)
 89.                         {
 90.                                 solution SCG = CG(ff4T, gradff4T, test_x, steps[step_i],
epsilon, Nmax, 0, NAN);
 91.                                 int* size_sd = get_size(SCG.ud);
 92.                                 for (int i = 0; i < size_sd[1]; i++)
 93.                                 {
 94.                                         CG_tfun << SCG.ud(0, i) << delimiter <<
SCG.ud(1, i) << std::endl;
 95.                                 }
 96.                                 CG_tfun << SCG.x(0) << delimiter << SCG.x(1) <<
std::endl;
 97.                                 CG_tfun << std::endl;
 98.                                 delete[] size_sd;
 99.                                 solution::clear_calls();
100.                         }
101.                         CG_tfun.close();
102.                 }
103.                 catch (string ex_info)
104.                 {
105.                         throw ("CG_tfun:\n" + ex_info);
106.                 }
107.
108.                 try {
109.                         ofstream Newton_tfun(OUTPUT_PATH + "out_Newton_tfun.txt");
110.                         if (!Newton_tfun.good()) throw("FILE NO GOOD");
111.                         for (int step_i = 0; step_i < 3; step_i++)
112.                         {
113.                                 solution SNewton = Newton(ff4T, gradff4T, Hff4T, test_x,
steps[step_i], epsilon, Nmax, 0, NAN);
114.                                 int* size_sd = get_size(SNewton.ud);
115.                                 for (int i = 0; i < size_sd[1]; i++)
116.                                 {
117.                                         Newton_tfun << SNewton.ud(0, i) << delimiter <<
SNewton.ud(1, i) << std::endl;
118.                                 }
119.
120.                                 Newton_tfun << SNewton.x(0) << delimiter << SNewton.x(1)
<< std::endl;
121.                                 Newton_tfun << std::endl;
122.                                 delete[] size_sd;
123.                                 solution::clear_calls();
124.                         }
125.                         Newton_tfun.close();
126.                 }
127.                 catch (string ex_info)
128.                 {
129.                         throw ("Newton_tfun:\n" + ex_info);
130.                 }
131.
132.         }
133.         catch (string ex_info)
134.         {
135.                 throw ("Graph values:\n" + ex_info);
136.         }
```

Fragment kodu 11: fragment funkcji *main* (*lab4*) zawierający funkcjonalności dla *Tabeli 1* oraz *Tabeli 2*

Analizując zbieżność rozwiązania to metoda najszybszego spadku Dla stałego kroku h = 0,05; osiąga wynik bliski optimum, natomiast przy h = 0,12 wynik staje się niestabilny osiągając astronomiczne wartości. Dla zmiennego kroku osiąga bardzo dokładne rozwiązanie. W przypadku metody gradientów sprzężonych wszystkie testowane przypadki prowadzą do bardzo dokładnych wyników. W przypadku metody Newtona dla kroku 0,05 oraz zmiennokrokowej metoda osiąga bardzo dokładne wyniki z małym błędem. Natomiast przy h = 0,12 również wykazuje dużą precyzję, choć mniejszą niż przy pozostałych krokach.

Metoda najszybszego spadku wymaga największej liczby wywołań funkcji celu, a w przypadku metody gradientów sprzężonych zachowuje większą stabilność, również posiadając tym samym większą precyzję. Metoda Newtona natomiast jest najbardziej efektywna w liczbie wywołań funkcji celu przy metodzie zmiennokrokowej. W przypadku stałych kroków (0,05) już nie jest tak zbieżna jak pozostałe dwie.

Skupiając się teraz na liczbie wywołań gradientu, można zauważyć, że dla metody najszybszego spadku jest zbliżona do liczby wywołań funkcji celu, ale dla zmiennego kroku znacząco spada. W metodzie gradientów sprzężonych i Newtona liczba wywołań gradientu jest najmniejsza dla zmiennego kroku. Można też zauważyć, że w przypadku metody Newtona liczby wywołań hesjanu jest identyczna liczbie wywołań gradientu.

Tabela 2. Wartości średnie wyników optymalizacji dla trzech długości kroku dla trzech metod

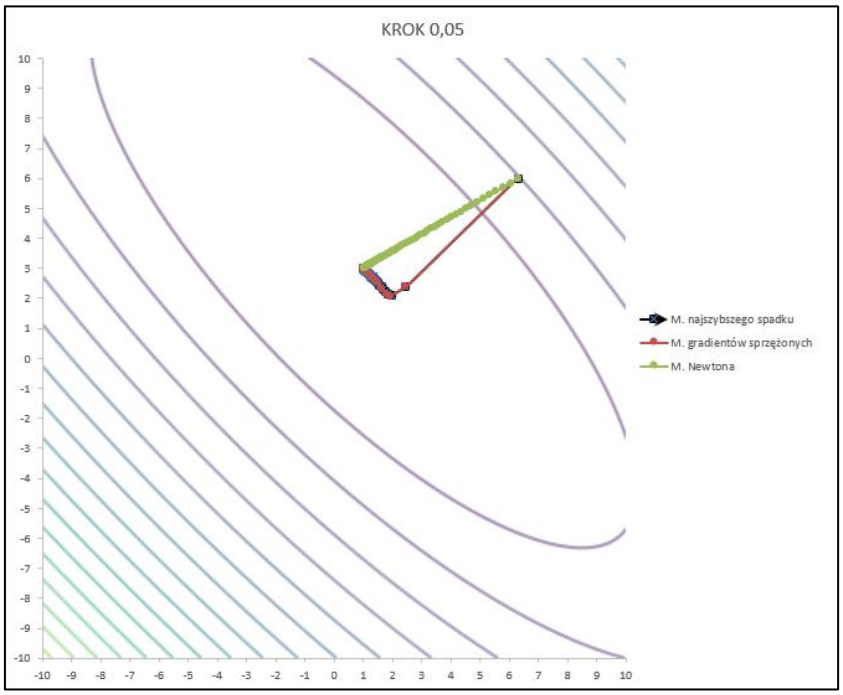
Długość kroku	Metoda najszybszego spadku				
	x_1*	x_2*	y*	f_calls	g_calls
0,05	1,002E+00	2,998E+00	7,31E-05	59,74	58,74
0,12	9,016E+64	9,016E+64	4,92E+131	1002,00	1001,00
M. zk.	1,000E+00	3,000E+00	1,90E-06	397,18	14,16

Długość kroku	Metoda gradientów sprzężonych				
	x_1*	x_2*	y*	f_calls	g_calls
0,05	1,000E+00	3,000E+00	3,30E-06	23,90	22,90
0,12	1,000E+00	3,000E+00	4,06E-06	55,12	54,12
M. zk.	1,000E+00	3,000E+00	4,62E-07	134,28	4,76

Długość kroku	Metoda Newtona					
	x_1*	x_2*	y*	f_calls	g_calls	H_calls
0,05	9,987E-01	2,995E+00	1,70E-03	236,88	117,94	117,94
0,12	9,993E-01	2,997E+00	2,48E-04	111,46	55,23	55,23
M. zk.	1,000E+00	3,000E+00	1,51E-14	76,69	2,61	2,61

Następnie przeprowadzono porównanie każdej z metod i każdej z długości kroków za pomocą wykresów poziomic.

Dla kroku 0,05 można zauważyć, że metoda gradientów sprzężonych jest najszybsza, co wynika z jej efektywnego wykorzystywania informacji o kierunku gradientu i oszczędności liczby iteracji. Metoda najszybszego spadku zbiega wolniej niż gradienty sprzężone, co wskazuje na jej większą zależność od dobrania odpowiedniej strategii krokowej. Natomiast metoda Newtona jest najwolniejszą z trzech metod, co można przypisać wysokim kosztom obliczeniowym związanym z wyznaczaniem i odwracaniem macierzy Hessego.

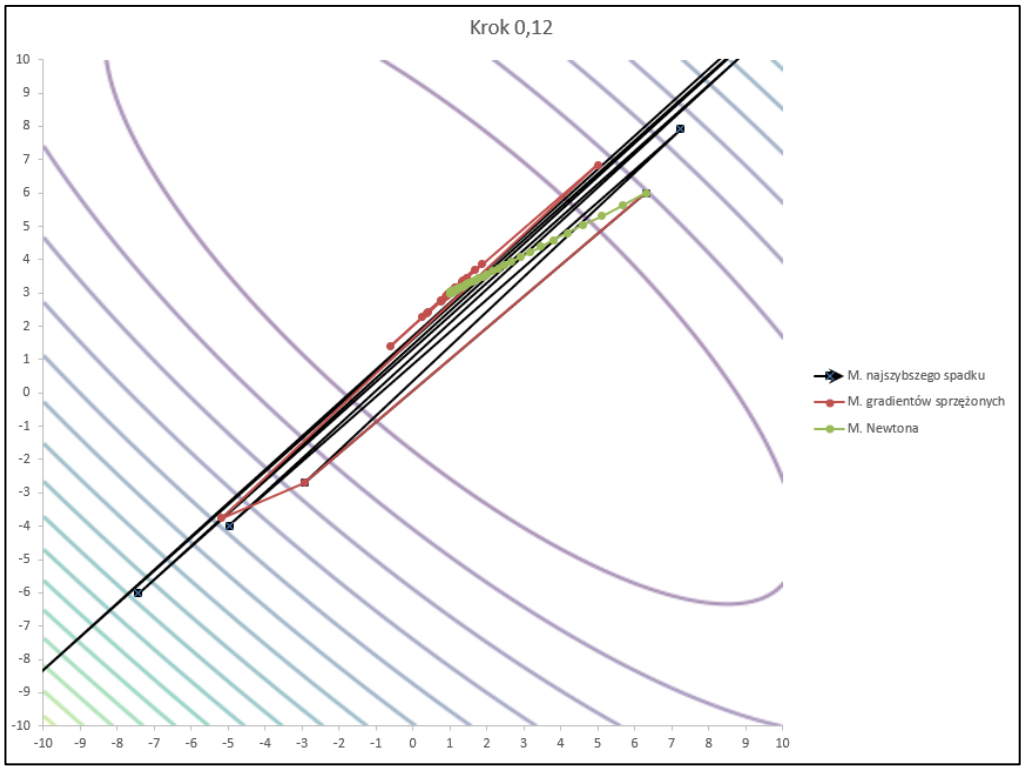


Wykres 1 – długość kroku 0,05 dla każdej z metod

Dla kroku 0,12 metoda najszybszego spadku nie działa – na wykresie pojawiają się spore oscylacje, a po porównaniu ich z danymi zebranymi do utworzenia wykresu można zauważyć, że funkcja oddala się od szukanego punktu. Wynika to prawdopodobnie z tego, że większy gradient sprawia, że odległość między punktem optymalnym ,a daną iteracją staje się coraz większa. Metoda Newtona wymaga dla tego kroku najmniejszej liczby iteracji i przewyższa gradienty sprzężone pod względem szybkości. Natomiast metoda gradientów sprzężonych jest

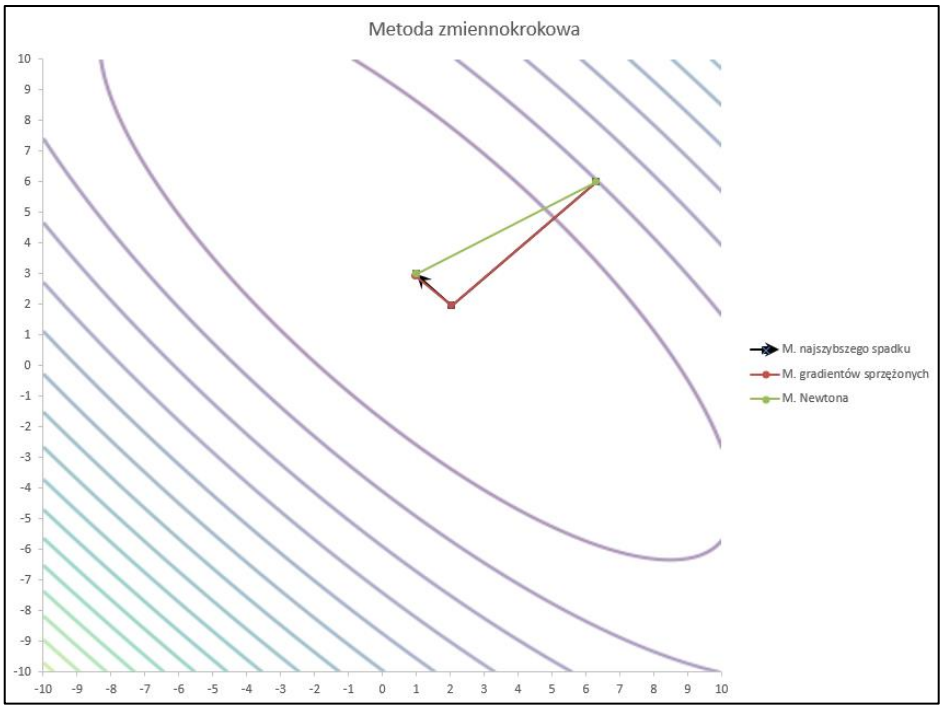
najwolniejsza w tej konfiguracji, co może być wynikiem jej większej wrażliwości na większe kroki.

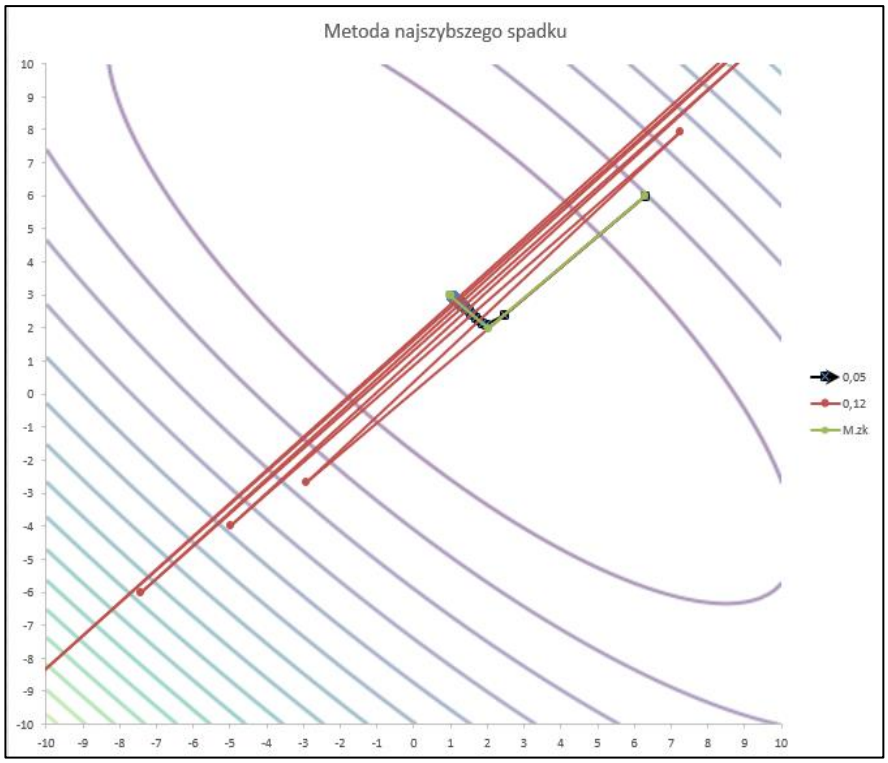
W przypadku zmiennego kroku metoda Newtona zbiega najszybciej, co potwierdza jej przewagę w dostosowywaniu się do lokalnej struktury problemu, szczególnie gdy krok zmienny dobrze balansuje szybkość i dokładność. Metoda gradientów sprzężonych jest nieco wolniejsza niż Newton, ale nadal bardzo szybka. Natomiast metoda najszybszego spadku jest najwolniejsza, choć różnica między metodami jest niewielka.



Wykres 2 – długość kroku 0,12 dla każdej z metod

Wykres 3 – wersja zmiennokrokowa dla każdej z metod





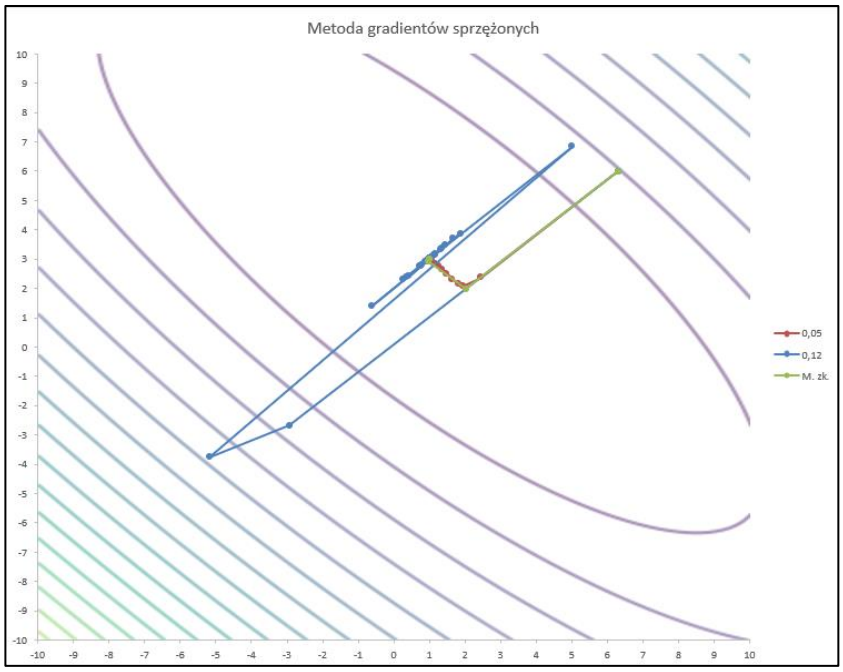
Wykres 4 – Metoda najszybszego spadku (3 długości koku)

Metoda najszybszego spadku dla kroku 0,05 jest stosunkowo wolna, ponieważ mały krok powoduje powolne zmiany w kierunku optymalnym. Chociaż wyniki zbliżają się do minimum, wymaga to dużej liczby iteracji. W przypadku kroku 0,12 metoda najszybszego spadku nie działa – rozwiązania oscylują i oddalają się od minimum, ale go nie osiągają w takim przybliżaniu, jakie ma miejsce dla kroku zmiennego czy równego 0,05. Zmienny krok to najefektywniejsza strategia dla metody

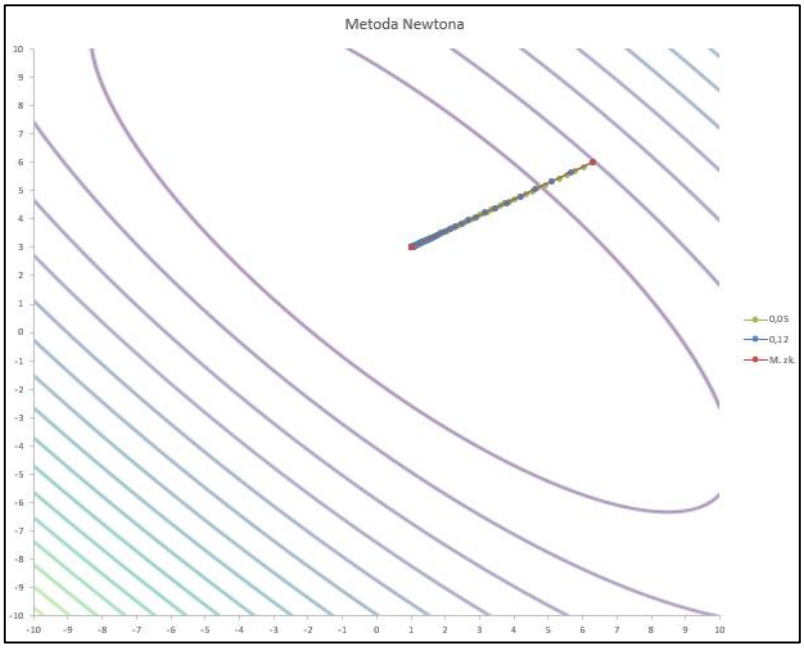
najszybszego spadku. Pozwala to algorytmowi adaptować krok w zależności od sytuacji, co umożliwia bardziej elastyczne i szybkie zbliżanie się do minimum.

W przypadku metody gradientów sprzężonych, dla kroku 0,05 jest bardzo efektywna - przy małym kroku szybko koncentruje się na kierunku optymalnym, nie wykonując niepotrzebnych zawirowań. Choć krok 0,05 jest mały, metoda szybko zbliża się do minimum, redukując liczbę iteracji w porównaniu do metody najszybszego spadku. Dla kroku 0,12 gradienty sprzężone stają się mniej skuteczne niż dla kroku 0,05, ale wciąż osiągają lepsze wyniki niż metoda najszybszego spadku. Zmienny krok daje gradientom sprzężonym dużą elastyczność, co pozwala na najlepszą efektywność. Algorytm potrafi dostosować się do zmieniających się warunków i optymalizować kierunek, co skutkuje szybszą zbieżnością

Metoda Newtona dla małego kroku 0,05 nie jest najbardziej efektywna. Choć jest to bardzo precyzyjna metoda, wymaga dużych obliczeń związanych z obliczaniem hesjanu i jego odwrotności. Mały krok prowadzi do długotrwałych obliczeń, które powodują, że metoda jest powolna w porównaniu do gradientów sprzężonych czy najszybszego spadku. Przy kroku 0,12 metoda Newtona działa nieco szybciej, ale wciąż jest stosunkowo kosztowna. Większy krok pozwala na szybsze osiąganie wyników, ale nadal koszty obliczeń są wyższe niż w metodzie gradientów sprzężonych. Zmienny krok umożliwia metodzie Newtona dużą elastyczność. Ponieważ zmienia się krok w zależności od potrzeby, metoda może wykorzystać większy krok w początkowych etapach, a mniejszy, gdy zbliża się do minimum. Chociaż tak naprawdę na wykresach te różnice nie są widoczne, ponieważ nakładają się one na siebie, to widać różnicę przy liczbie iteracji dla każdego kroku.



Wykres 5 – Metoda gradientów sprzężonych (3 długości kroku)



Wykres 6 – Metoda Newtona (3 długości kroku)

Problem rzeczywisty

Kod wczytuje dane z dwóch plików wejściowych (*XData.txt* i *YData.txt*), przetwarza je na macierze (*x_matrix* i *y_matrix*), a następnie wykorzystuje algorytm optymalizacji (metodą gradientów sprzężonych) do rozwiązania rzeczywistego problemu przy różnych długościach kroków. Iteruje po kilku krokach (0,01, 0,001, 0,0001) i oblicza rozwiązanie optymalizacyjne. Po każdej iteracji zapisuje rozwiązanie do pliku wyjściowego, w tym dokładność i niektóre związane obliczenia.

```
1.      // ---------- Real Problem ---------- //
2.      std::cout << "Solving for real problem...";
3.
4.      // Input files
5.      ifstream x_matrix_file(INPUT_PATH + "XData.txt");
6.      ifstream y_matrix_file(INPUT_PATH + "YData.txt");
7.
8.      // Matrix sizes
9.      const int cols = 100;
10.     const int x_rows = 3;
11.     const int y_rows = 1;
12.
13.     // Load data from files into matrices (all integers)
14.     matrix x_matrix(x_rows, cols);      // Data per col [a] = { x_theta, x_grade1, x_grade2
15.     matrix y_matrix(y_rows, cols);      // Data per col [a] = { has_passed }
16.
17.     x_matrix_file >> x_matrix;
18.     y_matrix_file >> y_matrix;
19.
20.     // Close the input files
21.     x_matrix_file.close();
22.     y_matrix_file.close();
23.
24.     matrix theta = {3, new double[3] {0.1f, 0.1f, 0.1f} };
25.
26.     // Debug print with test values
27.     // std::cout << "J: " << get_cost(theta, y_matrix, x_matrix) << "\n";
28.     // std::cout << "Gradient:\n" << get_gradient(x_matrix, y_matrix, theta) << "\n";
29.
30.     // Solve for various step lengths
31.     int real_n_max = 2e5;
32.     double real_epsilon = 1e-7;
33.     double step_length[] = { 0.01, 0.001, 0.0001 };
34.     int iterations = sizeof(step_length) / sizeof(double);
35.
36.     // File reference for data output
37.     ofstream output_file;
38.
39.     for (int i = 0; i < iterations; i++)
40.     {
41.             // Call the optimization function
42.             std::cout << "\n--- Step = " << step_length[i] << " # ";
43.             solution::clear_calls();
44.             solution opt_sol = CG(get_cost, get_gradient, theta, step_length[i],
real_epsilon, real_n_max, y_matrix, x_matrix);
45.             std::cout << "#";
46.
47.             // Open output file
48.             std::string out_number = to_string(i + 1);
49.             output_file.open(OUTPUT_PATH + "out_2_" + out_number + ".txt");
50.
51.             // Output the solution to file
52.             output_file << opt_sol << "\n";
53.             output_file<< "Accuracy: " << get_accuracy(opt_sol.x, x_matrix, y_matrix,
cols) << "\n";
54.
55.             for (int j = 0; j < cols; j++)
56.             {
57.                 output_file << x_matrix(1, j) << delimiter << x_matrix(2, j) << delimiter
58.               << y_matrix(0, j) << delimiter << get_h_l4(opt_sol.x, x_matrix, j) << "\n";
59.             }
60.
61.
62.             // Close output file
63.             output_file.close();
64.     }
65.
66.     std::cout << "\n";
```

Fragment kodu 12: fragment funkcji *main* zawierający funkcjonalności dla *Tabeli 3*

Dla długości kroku 0.01 funkcja kosztu wynosi 4.76, co wskazuje na stosunkowo dużą rozbieżność od rzeczywistego wyniku. W 30 przypadkach klasyfikator poprawnie przewidział wynik rekrutacji, a liczba wywołań funkcji gradientu to 100001, ale zbieżność nie została osiągnięta. Przy długości kroku 0.001 funkcja kosztu spada, co świadczy o lepszej jakości modelu. Poprawność przewidywania modelu rośnie do 84, a liczba wywołań gradientu spada do 8480 - przy tej długości kroku jest zbieżność.

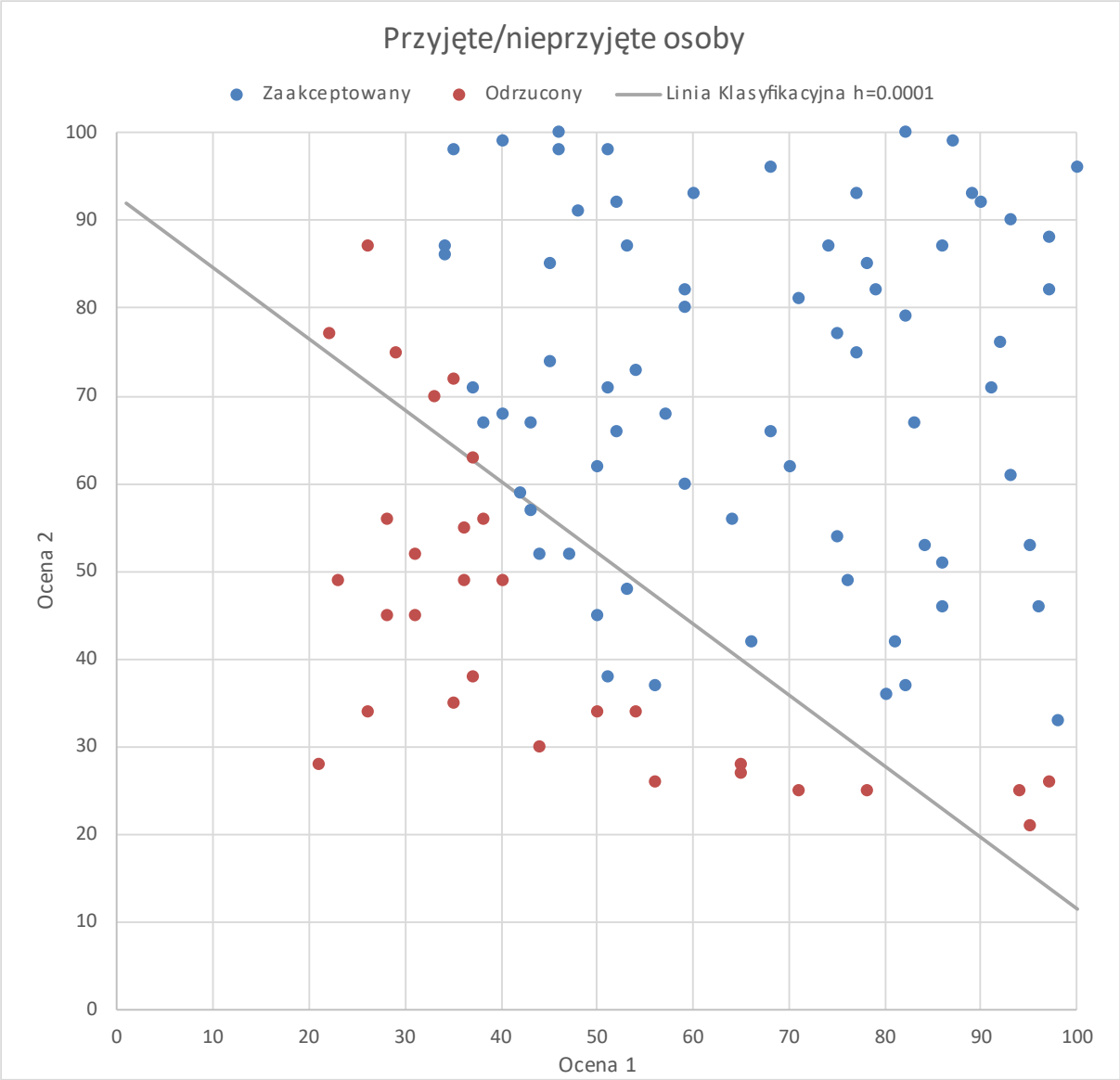
W przypadku kroku 0.0001 parametry są zbliżone do tych uzyskanych przy kroku 0.001 podobnie jak funkcja kosztu i poprawność przewidywania modelu. Ogólnie, najmniejszy krok (0.0001) daje dokładne wyniki, ale wymaga większej liczby iteracji, podczas gdy krok 0.001 daje najlepszy kompromis między czasem obliczeń a jakością modelu. Zbyt duży krok, jak w przypadku 0.01, może prowadzić do gorszej zbieżności, co jest widoczne w wysokiej wartości funkcji kosztu.

Tabela 3. Zestawienie wyników optymalizacji dla problemu rzeczywistego

Długość kroku	Metoda gradientów sprzężonych					
	θ_0*	θ_1*	θ_2*	J(θ*)	P(θ*)	g_calls
0,01	-20,6563	0,0791602	1,21E-01	4,76E+00	30	100001
0,001	-11,8209	0,10352	0,127354	0,271198	84	8480
0,0001	-1,18E+01	0,103504	0,127335	0,271198	84	23791

Wykres ilustruje, jak wartości ocen wpływają na klasyfikację. Dzięki małemu krokowi h = 0.0001, proces optymalizacji parametrów przebiega stabilnie, co pozwala na precyzyjne wyznaczenie granicy decyzyjnej.

Można zauważyć, że 16 punktów na wykresie znajduje się ze złej strony (tj, przyjęte osoby z przewidywaniem nieprzyjęcia i odwrotnie), co jest wynikiem spodziewanym i wynika z tego, że szukana prosta tworzy zależność liniową.



Rys. 3. Wykres klasyfikatora i danych rzeczywistych

Wnioski

Dla stałego kroku h = 0,05 metody różnią się pod względem efektywności i szybkości zbieżności. Metoda gradientów sprzężonych jest najbardziej efektywna, ponieważ szybko zbliża się do minimum dzięki skutecznemu wykorzystaniu kierunków gradientu. Mały krok umożliwia dokładne poszukiwanie optymalnego rozwiązania, zmniejszając liczbę iteracji i wywołań funkcji celu w porównaniu do innych metod. Z kolei metoda najszybszego spadku przy tym kroku działa wolniej, ponieważ mały krok powoduje stopniowe i powolne zmiany w

kierunku optymalnym, przez co wymaga większej liczby iteracji i wywołań funkcji celu. Metoda Newtona, mimo że jest bardzo precyzyjna, jest przy tym kroku najmniej efektywna ze względu na wysokie koszty obliczeniowe związane z obliczaniem hesjanu i jego odwrotności. Choć metoda ta jest dokładna, wymaga znacznych zasobów obliczeniowych, co powoduje jej wolniejsze tempo zbieżności.

Zwiększenie kroku do h = 0,12 dla metody Newtona działa przewyższa gradienty sprzężone pod względem szybkości, ponieważ może lepiej adaptować się do większych kroków. Gradienty sprzężone przy h = 0,12 są natomiast najwolniejsze. Metoda ta, mimo że początkowo może działać dobrze przy mniejszych krokach, staje się mniej skuteczna przy większym kroku, ponieważ jest bardziej wrażliwa na zmiany w wielkości kroku. W przypadku metody najszybszego spadku nie ma zbieżności dla tego kroku.

Zmienny krok okazał się najbardziej efektywną strategią w przypadku wszystkich trzech metod. Dla metody najszybszego spadku zmienny krok pozwala na adaptację w zależności od sytuacji, co sprawia, że algorytm może szybciej zbliżać się do minimum, zachowując stabilność. Metoda gradientów sprzężonych również korzysta na zmiennym kroku, dostosowując rozmiar kroku do bieżącej sytuacji w trakcie iteracji, co pozwala na uzyskanie lepszej zbieżności i oszczędność zasobów obliczeniowych. Metoda Newtona, dzięki zmiennemu krokowi, osiąga najlepsze wyniki pod względem szybkości zbieżności, ponieważ jest w stanie dostosować krok do lokalnej struktury problemu. Zmienny krok w tej metodzie pozwala na wykorzystanie większych kroków na początku procesu optymalizacji i mniejszych w pobliżu minimum, co przyspiesza konwergencję.

Metoda najszybszego spadku charakteryzuje się największą liczbą wywołań funkcji celu, szczególnie przy małym kroku, co skutkuje wolniejszą zbieżnością. Jednak przy większym kroku staje się mniej efektywna – gradient się zwiększa, a chociaż krok pozostaje ten sam, to odległość od minimum robi się coraz większa. Zmienny krok pozwala tej metodzie na szybsze osiąganie wyników, zachowując większą stabilność.

Metoda gradientów sprzężonych daje bardzo dokładne wyniki, wymagając mniejszej liczby iteracji niż metoda najszybszego spadku. Przy małym kroku jest szczególnie efektywna, jednak przy większym kroku jej skuteczność maleje, ponieważ staje się bardziej wrażliwa na zmiany w rozmiarze kroku. Zmienny krok również poprawia jej efektywność, pozwalając na szybszą i dokładniejszą konwergencję.

Metoda Newtona jest najbardziej efektywna w przypadku zmiennego kroku, szczególnie przy dynamicznej adaptacji do lokalnej struktury problemu. Przy stałym kroku 0,05 metoda ta nie jest najefektywniejsza, ponieważ obliczenia hesjanu są kosztowne i prowadzą do wolniejszego tempa zbieżności. Przy większym kroku jej efektywność rośnie, ale wciąż jest wyraźnie mniej skuteczna niż przy zmiennym kroku.

W przypadku problemu rzeczywistego, przy dużej długości kroku (h = 0.01) algorytm ma tendencję do oscylowania wokół minimum, zamiast skutecznie zbiegać do niego. Tego rodzaju zachowanie wynika z faktu, że duże kroki mogą prowadzić do przeskakiwania przez minimum, co skutkuje brakiem precyzyjnego podejścia do optymalnych wartości parametrów. W efekcie, algorytm wymaga znacznie większej liczby iteracji, by zbliżyć się do minimum, co nie tylko wydłuża czas obliczeń, ale również sprawia, że proces staje się mniej stabilny.

Z kolei w przypadku mniejszych kroków, takich jak h = 0.001 czy h = 0.0001, proces optymalizacji przebiega w sposób znacznie bardziej stabilny i dokładny. Mniejsze kroki umożliwiają stopniowe i precyzyjne zmniejszanie funkcji kosztu, eliminując ryzyko oscylacji i zapewniając stabilną eksplorację przestrzeni parametrów. Dodatkowo, mniejsze kroki zmniejszają ryzyko akumulacji błędów numerycznych, ponieważ zmiany w parametrach w każdej iteracji są na tyle niewielkie, że algorytm nie gubi precyzji obliczeń. W rezultacie, funkcja kosztu jest skutecznie minimalizowana, a parametry osiągają wartości realistyczne i sensowne, co skutkuje poprawną i stabilną klasyfikacją (z poprawnością przewidywania modelu równą 84).

Podsumowując, chociaż większe kroki mogą przyspieszyć proces, to ostatecznie mogą prowadzić do niestabilności i braku zbieżności. Z kolei mniejsze kroki wymagają więcej iteracji, ale zapewniają stabilność i precyzyjne dopasowanie parametrów, co prowadzi do lepszej jakości modelu. Najlepsze rezultaty uzyskuje się, wybierając optymalną długość kroku, która zapewnia dobrą równowagę między czasem obliczeń a jakością uzyskiwanego rozwiązania.