Optymalizacja funkcji wielu zmiennych metodami gradientowymi	Data wykonania: 13.12.2025 r.	Optymalizacja
Dominika Myszka Tatsiana Merzianiova	Gr. 2	ITE

1. Cel ćwiczenia

Celem ćwiczenia było zapoznanie się z gradientowymi metodami optymalizacji (metoda najszybszego spadku, metoda gradientów sprzężonych, metoda Newtona) poprzez ich implementację oraz praktyczne zastosowanie. Zadanie polegało na znalezieniu minimum testowej funkcji wielu zmiennych oraz optymalizacji parametrów klasyfikatora logicznego dla problemu rzeczywistego (rekrutacja na studia).

2. Testowa funkcja celu

2.1. Opis problemu

Rozważano funkcję celu daną wzorem:

$$f(x_1, x_2) = {}^1\!/_6\, x_1^6 - 1{,}05x_1^4 + 2x_1^2 + x_2^2 + x_1x_2$$

Funkcja ta posiada minimum globalne w punkcie(0,0) o wartości 0 oraz minima lokalne. Poszukiwania prowadzono w dziedzinie $x_1, x_2 \in [-2,2]$.

2.2. Porównanie metod optymalizacji

Przeprowadzono serię 100 prób optymalizacji dla każdego wariantu algorytmu, startując z losowych punktów początkowych. Przetestowano trzy metody (Najszybszego Spadku - MS, Gradientów Sprzężonych - MGS, Newtona - MN) przy zastosowaniu kroku stałego (h=0,05, h=0,25) oraz kroku zmiennego wyznaczanego metodą Złotego Podziału.

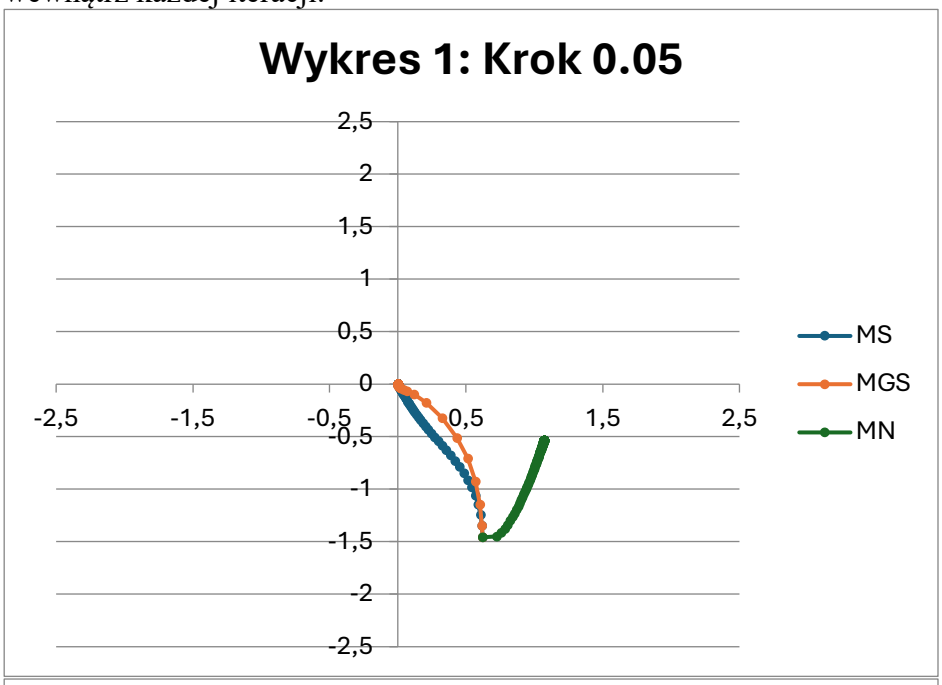
Wyniki zbiorcze przedstawiono w Tabeli 1 i Tabeli 2

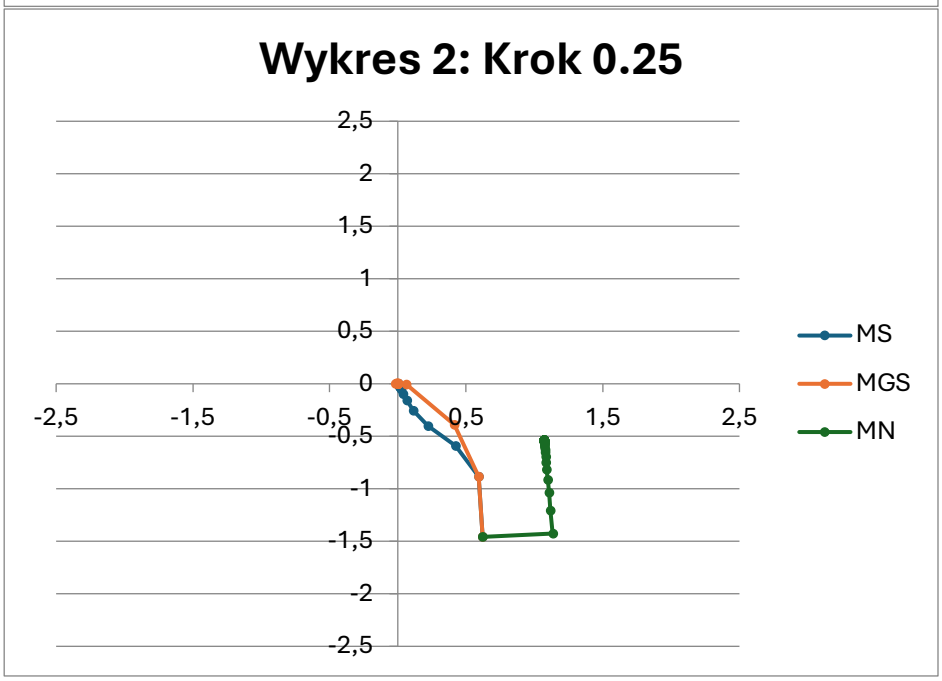
2.3. Analiza trajektorii (Wnioski z wykresów)

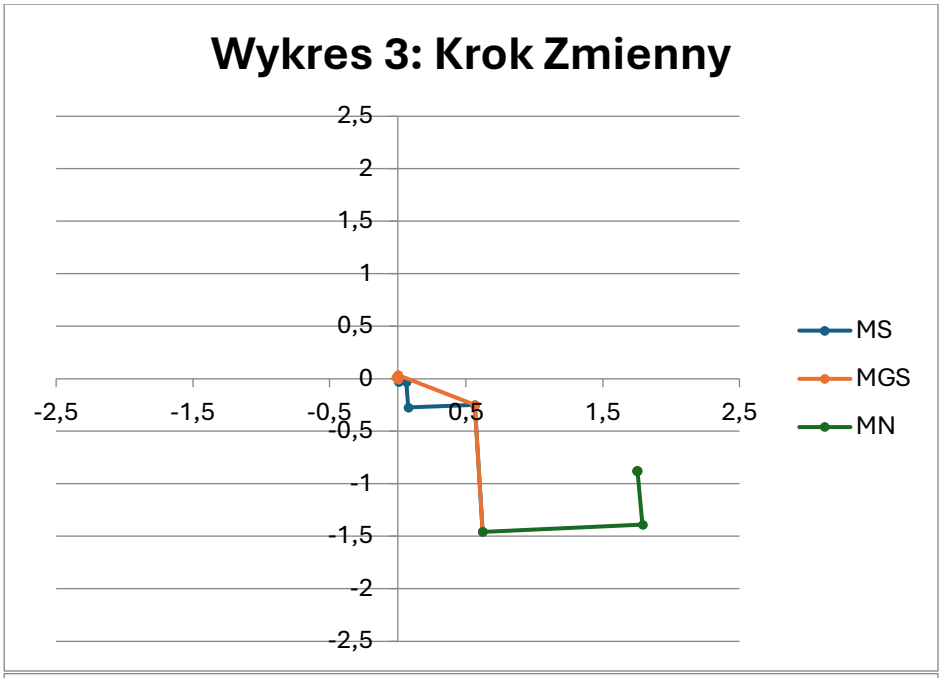
Na podstawie wygenerowanych wykresów trajektorii optymalizacji (Wykresy 1-6) zaobserwowano następujące zależności:

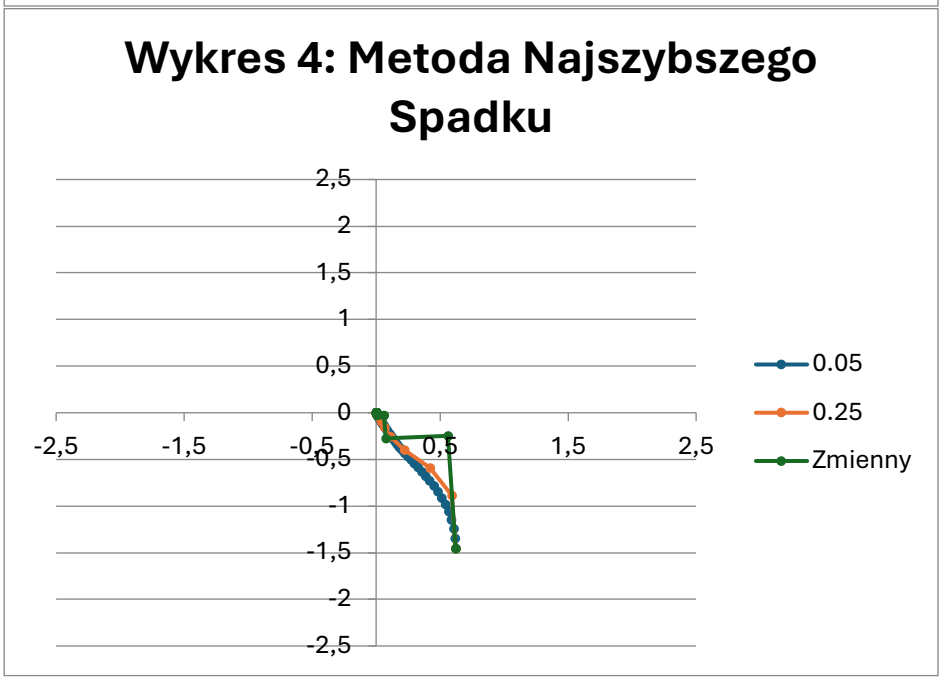
1. **Metoda Najszybszego Spadku (MS):** Charakteryzuje się powolną zbieżnością w pobliżu optimum. Trajektoria często przyjmuje kształt "zygzaka" (oscylacje), szczególnie przy stałym kroku. Metoda ta jest najmniej efektywna obliczeniowo pod względem liczby iteracji potrzebnych do osiągnięcia zadanej dokładności.

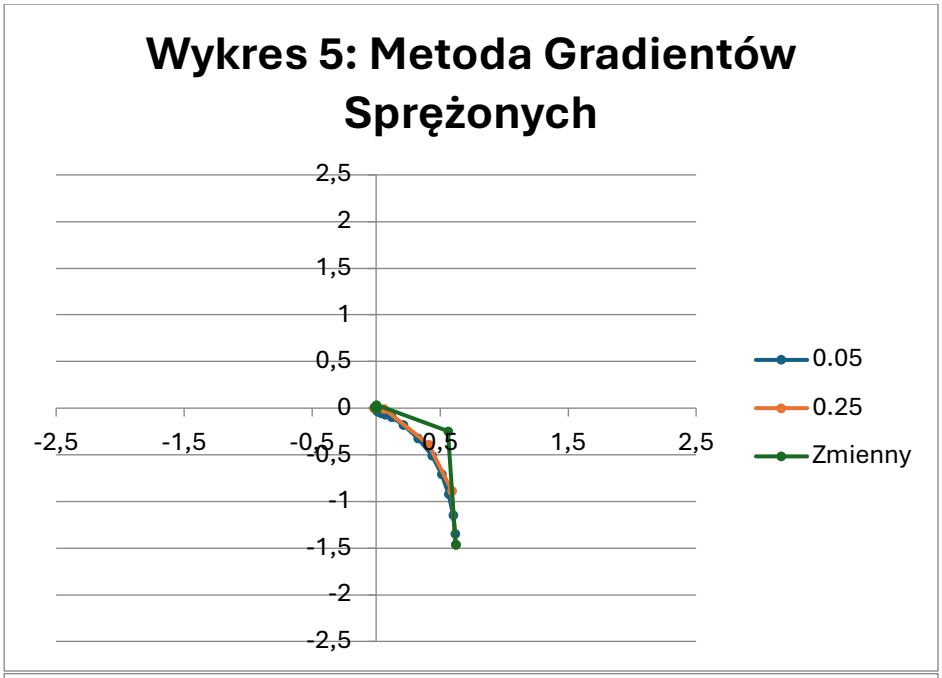
2. **Metoda Newtona (MN):** Wykazuje najszybszą zbieżność (najmniejsza liczba kroków), ponieważ wykorzystuje informacje o krzywiźnie funkcji (Hesjan). Ścieżka do minimum jest niemal bezpośrednia.
3. **Metoda Gradientów Sprzężonych (MGS):** Stanowi kompromis między metodą MS a MN. Zbiega szybciej niż MS, unikając oscylacji dzięki uwzględnieniu kierunku z poprzedniej iteracji (parametr).
4. **Wpływ długości kroku:**
    - **Krok 0,05:** Zapewnia stabilną, ale powolną zbieżność.
    - **Krok 0,25:** Przyspiesza proces, ale w przypadku metody MS zwiększa ryzyko oscylacji.
    - **Krok zmienny (M. Złotego Podziału):** Gwarantuje największą stabilność i zazwyczaj prowadzi do znalezienia minimum w najmniejszej liczbie iteracji algorytmu głównego, choć kosztem dodatkowych obliczeń funkcji celu wewnątrz każdej iteracji.

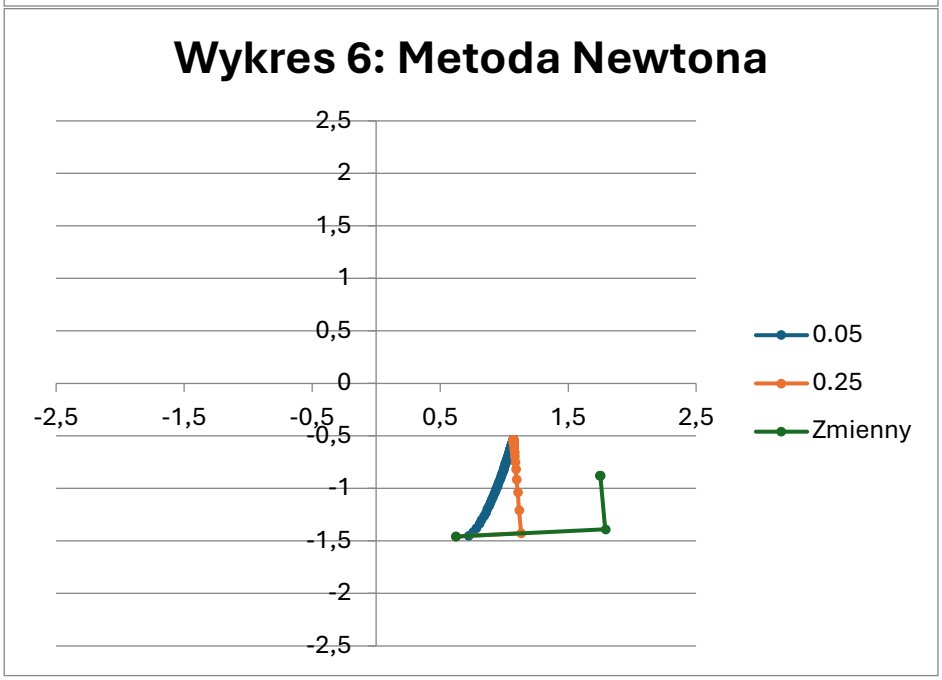





Wykres 3: Krok Zmienny
2,5
2
1,5
1
0,5
0
-0,5
-1
-1,5
-2
-2,5
-2,5
-1,5
-0,5
0,5
1,5
2,5
MS
MGS
MN


Wykres 4: Metoda Najszybszego Spadku
2,5
2
1,5
1
0,5
0
-0,5
-1
-1,5
-2
-2,5
-2,5
-1,5
-0,5
0,5
1,5
2,5
0.05
0.25
Zmienny





3. Problem rzeczywisty – Klasyfikator

3.1. Opis problemu

Celem było wytrenowanie klasyfikatora (regresja logistyczna), który na podstawie dwóch ocen przewiduje przyjęcie kandydata na studia. Funkcja hipotezy dana jest wzorem:

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

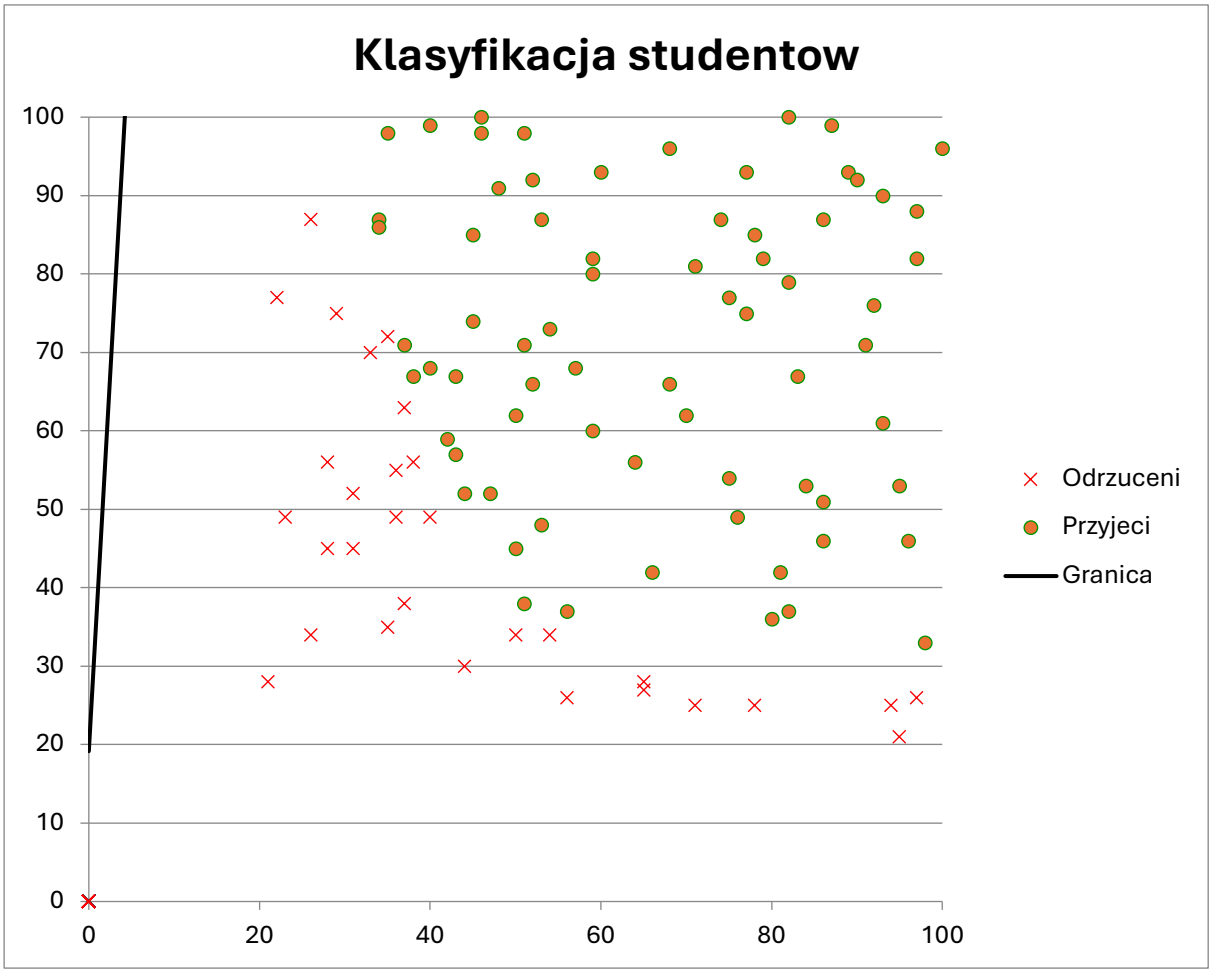
Optymalizację przeprowadzono metodą Gradientów Sprzężonych dla różnych długości kroku stałego, startując z punktu $\theta=[0,0,0]^T$.

3.2. Wyniki optymalizacji

Wyniki dla różnych długości kroku zestawiono w Tabele 3.

3.3. Dyskusja wyników

1. **Długość kroku h=0,01 :** Algorytm okazał się rozbieżny. Wartości funkcji kosztu rosły lawinowo, a parametry osiągnęły absurdalnie wysokie wartości. Świadczy to o "przestrzeleniu" minimum funkcji kosztu przez zbyt duży krok (overshooting). Skuteczność klasyfikacji wyniosła jedynie ok. 60%, co jest wynikiem losowym.
2. **Długość kroku h=0,00001 oraz h= 0,001 :** Algorytm zbiegł poprawnie do minimum. Uzyskano skuteczność klasyfikacji na poziomie **83-89%**
3. **Granica decyzyjna:** Wyznaczona granica decyzyjna (linia $x_2 = -\frac{\theta_0 + \theta_1 x_1}{\theta_2}$) poprawnie separuje większość punktów ze zbioru uczącego, co widać na załączonym wykresie.



4. Wnioski końcowe

- Dobór metody: Metoda Newtona jest bezkonkurencyjna pod względem szybkości zbieżności w sensie liczby iteracji, jednak wymaga kosztownego obliczania i odwracania macierzy Hesjanu. Dla problemów o dużej liczbie zmiennych lepszym wyborem może być metoda Gradientów Sprzężonych.

- Dobór kroku: Jest to kluczowy parametr metod gradientowych. Zbyt mały krok powoduje niepotrzebnie długi czas obliczeń, natomiast zbyt duży (jak w przypadku h=0,01 w zadaniu 2) prowadzi do braku zbieżności. Zastosowanie metod zmiennokrokowych (np. Złoty Podział) eliminuje ten problem, automatyzując dobór h.

- Minima lokalne: Wyniki dla funkcji testowej potwierdzają, że metody gradientowe są metodami lokalnymi – algorytm może utknąć w minimum lokalnym w zależności od punktu startowego. Aby znaleźć minimum globalne funkcji niewypukłej, konieczne jest wielokrotne uruchomienie algorytmu z różnych punktów (Multistart).

5. Dodatek - Kod źródłowy

Poniżej przedstawiono kod źródłowy funkcji przygotowanych specjalnie na potrzeby niniejszego ćwiczenia: definicje testowej funkcji celu oraz funkcji kosztu dla klasyfikatora (plik `user_funs.cpp`), a także funkcję sterującą przebiegiem eksperymentu (plik `main.cpp`).

5.1. Definicja problemów optymalizacyjnych (user_funs.cpp)

```
matrix func_test(matrix x, matrix ud1, matrix ud2) {
    double x1 = x(0);
    double x2 = x(1);
    // f(x) = 1/6*x1^6 - 1.05*x1^4 + 2*x1^2 + x2^2 + x1*x2
    double val = (1.0/6.0)*pow(x1, 6) - 1.05*pow(x1, 4) + 2.0*pow(x1, 2) + pow(x2,
2) + x1*x2;
    return matrix(val);
}

matrix grad_test(matrix x, matrix ud1, matrix ud2) {
    matrix g(2, 1);
    double x1 = x(0);
    double x2 = x(1);
    // Gradient analityczny
    g(0) = pow(x1, 5) - 4.2*pow(x1, 3) + 4.0*x1 + x2;
    g(1) = 2.0*x2 + x1;
    return g;
}

matrix hess_test(matrix x, matrix ud1, matrix ud2) {
    matrix H(2, 2);
    double x1 = x(0);
    // Hesjan analityczny
    H(0, 0) = 5.0*pow(x1, 4) - 12.6*pow(x1, 2) + 4.0;
```

```
    H(0, 1) = 1.0;
    H(1, 0) = 1.0;
    H(1, 1) = 2.0;
    return H;
}

// ========================================
// 2. PROBLEM RZECZYWISTY
// ========================================

// Pomocnicza funkcja sigmoidalna
double sigmoid(double z) {
    return 1.0 / (1.0 + exp(-z));
}

matrix func_real(matrix theta, matrix X, matrix Y) {
    int m = 100;

    double cost = 0.0;

    for (int i = 0; i < m; ++i) {
        // Oblicz hipotezę h(x) = theta^T * x
        double z = 0.0;
        for (int j = 0; j < 3; ++j) {
            z += theta(j) * X(j, i);
        }
        double h = sigmoid(z);

        // Zabezpieczenie logarytmu
        if(h < 1e-15) h = 1e-15;
        if(h > 1.0 - 1e-15) h = 1.0 - 1e-15;

        double y_val = Y(0, i);
        cost += y_val * log(h) + (1.0 - y_val) * log(1.0 - h);
    }

    return matrix(-1.0 / m * cost);
}

matrix grad_real(matrix theta, matrix X, matrix Y) {

    int m = 100;
    matrix g(3, 1);

    g(0) = 0; g(1) = 0; g(2) = 0;

    for (int i = 0; i < m; ++i) {
        double z = 0.0;
        for (int j = 0; j < 3; ++j) {
            z += theta(j) * X(j, i);
        }
```

```
        double h = sigmoid(z);

        double error = h - Y(0, i);

        for (int j = 0; j < 3; ++j) {
            g(j) += error * X(j, i);
        }
    }

    return (1.0 / m) * g;
}
```

5.2. Logika sterująca eksperymentem (main.cpp)

```
void lab4() {
    // 1. Wczytywanie danych
    matrix X(3, 100);
    matrix Y(1, 100);
    for (int i = 0; i < 100; ++i) X(0, i) = 1.0;

    try {
        ifstream fileX("XData.txt");
        ifstream fileY("YData.txt");
        if (!fileX.is_open() || !fileY.is_open()) throw runtime_error("Brak plikow
danych.");

        char dummy; double val;
        for (int row = 1; row <= 2; ++row) {
            for (int col = 0; col < 100; ++col) {
                fileX >> val; X(row, col) = val; fileX >> dummy;
            }
        }
        for (int col = 0; col < 100; ++col) {
            fileY >> val; Y(0, col) = val; fileY >> dummy;
        }
    } catch (...) { }

    // 2. Tabela 1 i 2 (Funkcja Testowa)
    ofstream f1("wyniki_lab4_tabela1.csv");
    ofstream f2("wyniki_lab4_tabela2.csv");

    f1 << "Dlugosc kroku,Lp.,x1(0),x2(0),Metoda najszybszego spadku,,,,,,Metoda
gradientow sprzezonych,,,,,,Metoda Newtona,,,,,," << endl;
    f1 <<
",,,,x1*,x2*,y*,f_calls,g_calls,MinGlob,x1*,x2*,y*,f_calls,g_calls,MinGlob,x1*,x2*,
y*,f_calls,g_calls,H_calls,MinGlob" << endl;

    f2 << "Dlugosc kroku,Metoda najszybszego spadku,,,,,,Metoda gradientow
sprzezonych,,,,,,Metoda Newtona,,,,,," << endl;
```

```
    f2 << ",x1*,x2*,y*,f_calls,g_calls,Liczba
MinGlob,x1*,x2*,y*,f_calls,g_calls,Liczba
MinGlob,x1*,x2*,y*,f_calls,g_calls,H_calls,Liczba MinGlob" << endl;

    random_device rd; mt19937 gen(rd());
    uniform_real_distribution<> dis(-2.0, 2.0);

    vector<matrix> start_points(100);
    for(int i=0; i<100; ++i) { start_points[i] = matrix(2, 1);
start_points[i](0)=dis(gen); start_points[i](1)=dis(gen); }

    struct RunConfig { double h; string name; };
    vector<RunConfig> configs = { {0.05, "0.05"}, {0.25, "0.25"}, {-1.0, "M. zk."}
};

    for (const auto& cfg : configs) {
        double sd_sum[5] = {0}; int sd_glob = 0;
        double cg_sum[5] = {0}; int cg_glob = 0;
        double new_sum[6] = {0}; int new_glob = 0;

        for (int i = 0; i < 100; ++i) {
            matrix x0 = start_points[i];

            if (i == 0) f1 << cfg.name;
            f1 << "," << (i + 1) << "," << x0(0) << "," << x0(1) << ",";

            // --- SD ---
            solution::clear_calls(); g_calls_cnt = 0;
            solution sol_sd = SD(func_test, grad_test, x0, cfg.h, 1e-3, 10000);
            bool sd_is_glob = (abs(sol_sd.y(0)) < 1e-2);
            f1 << sol_sd.x(0) << "," << sol_sd.x(1) << "," << sol_sd.y(0) << "," <<
sol_sd.f_calls << "," << g_calls_cnt << "," << (sd_is_glob ? "TAK" : "NIE") << ",";

            if (sd_is_glob) {
                sd_sum[0]+=sol_sd.x(0); sd_sum[1]+=sol_sd.x(1);
sd_sum[2]+=sol_sd.y(0);
                sd_sum[3]+=sol_sd.f_calls; sd_sum[4]+=g_calls_cnt; sd_glob++;
            }

            // --- CG ---
            solution::clear_calls(); g_calls_cnt = 0;
            solution sol_cg = CG(func_test, grad_test, x0, cfg.h, 1e-3, 10000);
            bool cg_is_glob = (abs(sol_cg.y(0)) < 1e-2);
            f1 << sol_cg.x(0) << "," << sol_cg.x(1) << "," << sol_cg.y(0) << "," <<
sol_cg.f_calls << "," << g_calls_cnt << "," << (cg_is_glob ? "TAK" : "NIE") << ",";

            if (cg_is_glob) {
                cg_sum[0]+=sol_cg.x(0); cg_sum[1]+=sol_cg.x(1);
cg_sum[2]+=sol_cg.y(0);
                cg_sum[3]+=sol_cg.f_calls; cg_sum[4]+=g_calls_cnt; cg_glob++;
            }
```

```
            // --- Newton ---
            solution::clear_calls(); g_calls_cnt = 0; H_calls_cnt = 0;
            solution sol_new = Newton(func_test, grad_test, hess_test, x0, cfg.h,
1e-6, 10000);
            bool new_is_glob = (abs(sol_new.y(0)) < 1e-2);
            f1 << sol_new.x(0) << "," << sol_new.x(1) << "," << sol_new.y(0) << ","
<< sol_new.f_calls << "," << g_calls_cnt << "," << H_calls_cnt << "," <<
(new_is_glob ? "TAK" : "NIE");

            if (new_is_glob) {
                new_sum[0]+=sol_new.x(0); new_sum[1]+=sol_new.x(1);
new_sum[2]+=sol_new.y(0);
                new_sum[3]+=sol_new.f_calls; new_sum[4]+=g_calls_cnt;
new_sum[5]+=H_calls_cnt; new_glob++;
            }

            f1 << endl;
        }

        auto avg = [](double sum, int count) { return count > 0 ? sum/count : 0; };

        f2 << cfg.name << ",";
        f2 << avg(sd_sum[0], sd_glob) << "," << avg(sd_sum[1], sd_glob) << "," <<
avg(sd_sum[2], sd_glob) << "," << avg(sd_sum[3], sd_glob) << "," << avg(sd_sum[4],
sd_glob) << "," << sd_glob << ",";
        f2 << avg(cg_sum[0], cg_glob) << "," << avg(cg_sum[1], cg_glob) << "," <<
avg(cg_sum[2], cg_glob) << "," << avg(cg_sum[3], cg_glob) << "," << avg(cg_sum[4],
cg_glob) << "," << cg_glob << ",";
        f2 << avg(new_sum[0], new_glob) << "," << avg(new_sum[1], new_glob) << ","
<< avg(new_sum[2], new_glob) << "," << avg(new_sum[3], new_glob) << "," <<
avg(new_sum[4], new_glob) << "," << avg(new_sum[5], new_glob) << "," << new_glob <<
endl;
    }
    f1.close(); f2.close();

    // 2.5 Generowanie danych do wykresów (dla jednego punktu startowego)
    ofstream f_graphs("wyniki_lab4_wykresy.csv");
    f_graphs << "Nr iteracji,Metoda najszybszego spadku,,,,,,Metoda gradientow
sprzezonych,,,,,,Metoda Newtona,,,,,," << endl;
    f_graphs << ",0.05,,0.25,,M. zk.,,0.05,,0.25,,M. zk.,,0.05,,0.25,,M. zk.," <<
endl;
    f_graphs <<
",x1*,x2*,x1*,x2*,x1*,x2*,x1*,x2*,x1*,x2*,x1*,x2*,x1*,x2*,x1*,x2*,x1*,x2*" << endl;

    matrix x0_plot = start_points[0]; // Wybrany punkt startowy
    vector<matrix> plot_results[9]; // 9 wariantów: 3 metody * 3 kroki

    int plot_idx = 0;
    double steps_arr[] = {0.05, 0.25, -1.0};
```

```
    // SD
    for(double h : steps_arr) {
        g_path_trace = &plot_results[plot_idx];
        solution::clear_calls();
        SD(func_test, grad_test, x0_plot, h, 1e-6, 10000);
        g_path_trace = nullptr;
        plot_idx++;
    }
    // CG
    for(double h : steps_arr) {
        g_path_trace = &plot_results[plot_idx];
        solution::clear_calls();
        CG(func_test, grad_test, x0_plot, h, 1e-6, 10000);
        g_path_trace = nullptr;
        plot_idx++;
    }
    // Newton
    for(double h : steps_arr) {
        g_path_trace = &plot_results[plot_idx];
        solution::clear_calls();
        Newton(func_test, grad_test, hess_test, x0_plot, h, 1e-6, 10000);
        g_path_trace = nullptr;
        plot_idx++;
    }

    size_t max_iter = 0;
    for(int i=0; i<9; ++i) if(plot_results[i].size() > max_iter) max_iter =
plot_results[i].size();

    for(size_t i=0; i<max_iter; ++i) {
        f_graphs << i;
        for(int j=0; j<9; ++j) {
            f_graphs << ",";
            if(i < plot_results[j].size()) {
                f_graphs << plot_results[j][i](0) << "," << plot_results[j][i](1);
            } else {
                f_graphs << ",";
            }
        }
        f_graphs << endl;
    }
    f_graphs.close();

    // 3. Klasyfikator
    ofstream f3("wyniki_lab4_klasyfikator.csv");
    f3 << "Dlugosc kroku,Metoda gradientow sprzezonych,,,,," << endl;
    f3 << ",Theta0*,Theta1*,Theta2*,J(Theta*),P(Theta*),g_calls" << endl;

    vector<double> steps = { 0.01, 0.001, 0.0001 };
    matrix theta_start(3, 1); theta_start(0)=0; theta_start(1)=0; theta_start(2)=0;
```

```
    for (double step : steps) {
        solution::clear_calls(); g_calls_cnt = 0;
        solution sol = CG(func_real, grad_real, theta_start, step, 1e-6, 10000, X,
Y);
        double acc = calculate_accuracy(sol.x, X, Y);

        f3 << step << "," << sol.x(0) << "," << sol.x(1) << "," << sol.x(2) << ","
<< sol.y(0) << "," << acc << "," << g_calls_cnt << endl;
    }
    f3.close();

    cout << "Wygenerowano pliki:\n- wyniki_lab4_tabela1.csv\n-
wyniki_lab4_tabela2.csv\n- wyniki_lab4_wykresy.csv\n- wyniki_lab4_klasyfikator.csv"
<< endl;
}
```

5.3. Implementacja algorytmów optymalizacji (opt_alg.cpp)

```
solution SD(matrix(*ff)(matrix, matrix, matrix), matrix(*gf)(matrix, matrix,
matrix), matrix x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    solution Xopt;
    Xopt.x = x0;
    Xopt.fit_fun(ff, ud1, ud2);

    // Zapis punktu startowego
    if (g_path_trace) g_path_trace->push_back(Xopt.x);

    matrix d(x0);
    solution X1;

    while (solution::f_calls < Nmax)
    {
        matrix grad = gf(Xopt.x, ud1, ud2);
        g_calls_cnt++;

        if (norm(grad) < epsilon) break;
        d = -grad;

        double h;
        if (h0 < 0) h = line_search_golden(ff, Xopt.x, d, ud1, ud2);
        else h = h0;

        X1.x = Xopt.x + h * d;
        X1.fit_fun(ff, ud1, ud2);

        if (g_path_trace) g_path_trace->push_back(X1.x);

        if (norm(X1.x - Xopt.x) < epsilon) { Xopt = X1; break; }
        Xopt = X1;
```

```
    }
    return Xopt;
}

solution CG(matrix(*ff)(matrix, matrix, matrix), matrix(*gf)(matrix, matrix,
matrix), matrix x0, double h0, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    solution Xopt;
    Xopt.x = x0;
    Xopt.fit_fun(ff, ud1, ud2);

    // Zapis punktu startowego
    if (g_path_trace) g_path_trace->push_back(Xopt.x);

    matrix grad = gf(Xopt.x, ud1, ud2);
    g_calls_cnt++;

    matrix d = -grad;
    matrix grad_old = grad;
    solution X1;

    while (solution::f_calls < Nmax)
    {
        double h;
        if (h0 < 0) h = line_search_golden(ff, Xopt.x, d, ud1, ud2);
        else h = h0;

        X1.x = Xopt.x + h * d;
        X1.fit_fun(ff, ud1, ud2);

        if (g_path_trace) g_path_trace->push_back(X1.x);

        if (norm(X1.x - Xopt.x) < epsilon) { Xopt = X1; break; }
        Xopt = X1;

        matrix grad_new = gf(Xopt.x, ud1, ud2);
        g_calls_cnt++;

        if (norm(grad_new) < epsilon) break;

        double beta = pow(norm(grad_new), 2) / pow(norm(grad_old), 2);
        d = -grad_new + beta * d;
        grad_old = grad_new;
    }
    return Xopt;
}

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)
{
```

```
    solution Xopt;
    Xopt.x = x0;
    Xopt.fit_fun(ff, ud1, ud2);

    // Zapis punktu startowego
    if (g_path_trace) g_path_trace->push_back(Xopt.x);

    solution X1;

    while (solution::f_calls < Nmax)
    {
        matrix grad = gf(Xopt.x, ud1, ud2);
        g_calls_cnt++;
        if (norm(grad) < epsilon) break;

        matrix H = Hf(Xopt.x, ud1, ud2);
        H_calls_cnt++;

        matrix d = -inv(H) * grad;

        double h;
        if (h0 < 0) h = line_search_golden(ff, Xopt.x, d, ud1, ud2);
        else {
             h = 1.0;
             if (h0 != 1.0) h = h0;
        }

        X1.x = Xopt.x + h * d;
        X1.fit_fun(ff, ud1, ud2);

        if (g_path_trace) g_path_trace->push_back(X1.x);

        if (norm(X1.x - Xopt.x) < epsilon) { Xopt = X1; break; }
        Xopt = X1;
    }
    return Xopt;
}
```