Jakub Latawiec

Rafał Malik

Optymalizacja

Laboratorium - optymalizacja funkcji wielu zmiennych metodami bezgradientowymi.

1. **Cel ćwiczenia**
   Celem ćwiczenia jest zapoznanie się z metodami bezgradientowymi poprzez ich implementację oraz wykorzystanie do rozwiązania problemu optymalizacji.

2. **Przeprowadzenie ćwiczenia**
   a) Implementacja metody Hooke'a-Jeevesa.

```
solution HJ(matrix(*ff)(matrix, matrix, matrix), matrix x0, double s, double
alpha, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
	try
	{
		//String builder do zapisu danych
		std::stringstream ss{};

		solution XT;

		//Wyliczanie punktu bazowego
		solution XB;
		XB.x = x0;
		XB.fit_fun(ff,ud1, ud2);

		while (s > epsilon)
		{
			//Poszukiwanie lepszego punktu
			XT = HJ_trial(ff, XB, s, ud1, ud2);

			//Znalezienie lepszego punktu
			if (XT.y < XB.y)
			{
				//Szukanie minimum przy stałym kroku do momentu
rośnięcia funkcji
				do
				{
					//Zapis danych do wykresu
					if (SAVE_CHART_DATA)
						ss << m2d(XB.x(0)) << ";" << m2d(XB.x(1))
<< "\n";

					solution XB_old = XB;
					XB = XT;
					XT.x = 2.0 * XB.x - XB_old.x;
					XT.fit_fun(ff, ud1, ud2);
					XT = HJ_trial(ff, XT, s, ud1, ud2);
					if (solution::f_calls > Nmax)
					{
						XT.flag = 0;
						throw std::string("Maximum amount of
f_calls reached!");
					}
				} while (XT.y < XB.y);
```

```
                        //Zastąpienie wyiczonego punktu poprzednim (XT to był
punkt w którym funkcja już rosła)
                        XT = XB;
                }
                //Brak lepszego punktu
                else
                        //Zmniejszenie kroku poszukiwań
                        s = s * alpha;

                if (solution::f_calls > Nmax)
                {
                        XT.flag = 0;
                        throw std::string("Maximum amount of f_calls reached!");
                }
        }

        //Zapis danych do pliku
        if (SAVE_CHART_DATA)
                save_to_file("HJ_chart.csv", ss.str());

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

Zaimplementowana metoda Hooke'a-Jeevesa w pliku opt_alg.cpp

b) Implementacja metody pomocniczej dla metody Hooke'a Jeevesa.

```
solution HJ_trial(matrix(*ff)(matrix, matrix, matrix), solution XB, double s,
matrix ud1, matrix ud2)
{
    try
    {
        int n = get_len(XB.x);

        //Baza szukania minimum
        matrix d = matrix(n, n);
        for (int i = 0; i < n; ++i)
            d(i, i) = 1.0;

        //Poszukiwanie lepszego minimum po każdym kierunku funkcji
        for (int j = 0; j < n; ++j)
        {
            //Wyliczanie nowego punktu
            solution X_trial;
            X_trial.x = XB.x + s * d[j];
            X_trial.fit_fun(ff, ud1, ud2);

            //Znalezienie lepszego punktu w kierunku dodatnim
            if (X_trial.y < XB.y)
                XB = X_trial;
            //Sprawdzenie kierunku ujemnego
            else
            {
                X_trial.x = XB.x - s * d[j];
                X_trial.fit_fun(ff, ud1, ud2);
                //Znalezienie lepszego punktu w kierunku ujemnym
                if (X_trial.y < XB.y)
                    XB = X_trial;
```

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

Zaimplementowana metoda pomocnicza Hooke'a-Jeevesa w pliku opt_alg.cpp

c) Implementacja metody Rosenbrocka.

```
solution Rosen(matrix(*ff)(matrix, matrix, matrix), matrix x0, matrix s0, double
alpha, double beta, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
    try
    {
        std::stringstream ss;

        solution Xopt;

        //Funkcja pomocnicza szukająca maksymalnej wartości bezwzględnej w
wektorze pionowym
        auto max = [&](matrix m) -> double
        {
            int len = get_len(m);
            double result = 0.0;
            for (int i = 0; i < len; ++i)
                if (abs(result) < abs(m(i)))
                    result = abs(m(i));

            return result;
        };

        int i = 0;
        //Ilość zmiennych funkcji
        int n = get_len(x0);

        //Baza szukania kierunku
        matrix d = matrix(n, n);
        for (int j = 0; j < n; ++j)
            d(j, j) = 1.0;

        //Wektory pomocnicze
        matrix lambda = matrix(n, new double[n] {0.0});
        matrix p = matrix(n, new double[n] {0.0});

        //Wektor pionowy kroku
        matrix s = s0;

        //Wyliczanie punktu bazowego
        solution XB;
        XB.x = x0;
        XB.fit_fun(ff, ud1, ud2);

        if (SAVE_CHART_DATA)
            ss << m2d(XB.x(0)) << ";" << m2d(XB.x(1)) << "\n";

        while (max(s) >= epsilon)
```

```
		{
			//Poszukiwanie lepszego minimum po każdym kierunku funkcji
			for (int j = 0; j < n; ++j)
			{
				//Wyliczanie nowego punktu
				solution XT;
				XT.x = XB.x + s(j) * d[j];
				XT.fit_fun(ff, ud1, ud2);

				//Ekspansja
				if (XT.y < XB.y)
				{
					XB = XT;
					lambda(j) += s(j);
					s(j) *= alpha;
				}
				//Kontrakcja
				else
				{
					s(j) *= -beta;
					p(j) += 1;
				}
			}
			++i;

			//Zapis danych do wykresu
			if (SAVE_CHART_DATA)
				ss << m2d(XB.x(0)) << ";" << m2d(XB.x(1)) << "\n";

			//Znaleziony punkt przypisywany do wynikowego
			Xopt = XB;

			//Sprawdzanie czy wszystkie możliwe kierunki zostały
sprawdzone
			bool changeDirectionBase = true;
			for (int j = 0; j < n; ++j)
			{
				if (lambda(j) == 0 || p(j) == 0)
				{
					changeDirectionBase = false;
					break;
				}
			}

			//Zmiana bazy kierunku
			if (changeDirectionBase)
			{
				//Macierz trójkątna dolna lambdy
				matrix lambdaMatrix(n, n);
				int l = 0;
				for (int k = 0; k < n; ++k)
				{
					for (int j = 0; j <= k; ++j)
						lambdaMatrix(k, j) = lambda(l);
					++l;
				}

				//Wyliczanie macierzy Q
				matrix Q = d * lambdaMatrix;

				//Wyznaczanie składowych wektora V
				matrix V = matrix(n, 1);
				V = Q[0];
```

```
                        d[0] = V / norm(V);
                        for (int j = 1; j < n; ++j)
                        {
                                matrix sum = matrix(n, new double[n] {0.0});
                                for (int k = 0; k < j; ++k)
                                {
                                        sum = sum + (trans(Q[j]) * d[k]) * d[k];
                                }
                                V = Q[j] - sum;
                                d[j] = V / norm(V);
                        }

                        //Zerowanie wektorów pomocniczych
                        lambda = matrix(n, new double[n] {0.0});
                        p = matrix(n, new double[n] {0.0});

                        //Zmiana długości kroku na początkowy
                        s = s0;
                }

                if (solution::f_calls > Nmax)
                {
                        Xopt.flag = 0;
                        throw std::string("Maximum amount of f_calls reached!");
                }
        }

        //Zapis danych do pliku
        if (SAVE_CHART_DATA)
                save_to_file("Rosen_chart.csv", ss.str());

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

Zaimplementowana metoda Rosenbrocka w pliku opt_alg.cpp

d) Implementacja funkcji celu dla problemu rzeczywistego

```
matrix ff2R(matrix x, matrix ud1, matrix ud2)
{
    matrix y;
    y = 0;

    //Warunki początkowe
    matrix Y0 = matrix(2, 1);

    //Symulacja
    matrix* Y = solve_ode(df2, 0.0, 0.1, 100, Y0, ud1, x);

    //Dane referencyjne
    double alpha_ref = ud1(4);
    double omega_ref = ud1(5);

    //Obliczanie funkjconału jakości metodą prostokątów
    int n = get_len(Y[0]);
    for (int i = 0; i < n; ++i)
```

```
	{
		y = y + 10 * pow(alpha_ref - Y[1](i, 0), 2) + pow(omega_ref -
Y[1](i, 1), 2) + pow(x(0) * (alpha_ref - Y[1](i, 0)) + x(1) * (omega_ref -
Y[1](i, 1)), 2);
	}
	y = 0.1 * y;

	return y;
}
```

Zaimplementowana funkcja celu dla problemu rzeczywistego w user_funs.cpp

e) Implementacja funkcji pochodnych dla problemu rzeczywistego

```
matrix df2(double t, matrix Y, matrix ud1, matrix ud2)
{
	//Wektor zmian po czasie
	matrix dY(2, 1);

	//Zmienne zadania
	double alpha = Y(0);
	double omega = Y(1);

	//Dane z zadania
	double l = ud1(0);
	double m_r = ud1(1);
	double m_c = ud1(2);
	double b = ud1(3);
	double alpha_ref = ud1(4);
	double omega_ref = ud1(5);

	//Współczynniki wzmocnienia
	double k1 = ud2(0);
	double k2 = ud2(1);

	//Moment bezwładności
	double I = (1.0 / 3.0) * m_r * pow(l, 2) + m_c * pow(l, 2);

	//Moment siły
	double Mt = k1 * (alpha_ref - alpha) + k2 * (omega_ref - omega);

	dY(0) = Y(1);
	dY(1) = (Mt - b * omega) / I;

	return dY;
}
```

Zaimplementowana funkcja pochodnych dla problemu rzeczywistego w user_funs.cpp

f) Implementacja funkcji pochodnych lab2

```
void lab2()
{

#ifdef SAVE_TO_FILE
	create_environment("lab02");
#endif

	//Dane dokładności wyników
```

```
	double s = 0.1;
	double alpha = 0.1;
	double beta = 0.1;
	double epsilon = 1E-6;
	double Nmax = 2000;
	double alphaRosen = 1.2;

#ifdef CALC_TEST

	//Generator liczb losowych
	std::random_device rd;
	std::mt19937 gen(rd());
	std::uniform_real_distribution<> x0_dist(-1.0, 1.0);

	//Stringstream do zapisu danych
	std::stringstream test_ss;

	//Rozwiązanie dla wyników testowych
	solution test_sol;

	//Dane dokładności dla testów
	double test_s = 0.1;

	//Punty startowe dla testów
	matrix test_x0{};

	//Liczenie testów
	for (int i = 0; i < 3; ++i)
	{
		for (int j = 0; j < 100; ++j)
		{
			test_x0 = matrix(2, new double[2] {x0_dist(gen),
x0_dist(gen)});
			test_sol = HJ(ff2T, test_x0, s, alpha, epsilon, Nmax);
			test_ss << test_x0(0) << ";" << test_x0(1) << ";" <<
m2d(test_sol.x(0)) << ";" << m2d(test_sol.x(1)) << ";" << m2d(test_sol.y) << ";"
<< test_sol.f_calls << ";" << (abs(m2d(test_sol.x(0))) < 0.01 &&
abs(m2d(test_sol.x(1))) < 0.01 ? "TAK" : "NIE") << ";";
			solution::clear_calls();
			test_sol = Rosen(ff2T, test_x0, matrix(2, new double[2] {s,
s}), alphaRosen, beta, epsilon, Nmax);
			test_ss << m2d(test_sol.x(0)) << ";" << m2d(test_sol.x(1)) <<
";" << m2d(test_sol.y) << ";" << test_sol.f_calls << ";" <<
(abs(m2d(test_sol.x(0))) < 0.01 && abs(m2d(test_sol.x(1))) < 0.01 ? "TAK" :
"NIE") << "\n";
			solution::clear_calls();
		}

		//Zapis do pliku
#ifdef SAVE_TO_FILE
		save_to_file("test_s_" + std::to_string(test_s) + ".csv",
test_ss.str());
#endif

		//Czyszczenie zawartości ss
		test_ss.str(std::string());

		//Zmiana długości kroku
		test_s += 0.6;
	}


	//Liczenie danych do wykresów
```

```
	SAVE_CHART_DATA = true;

	//Wspólny punkt dla obu metod
	test_x0 = matrix(2, new double[2] {0.45, 0.45});

	HJ(ff2T, test_x0, s, alpha, epsilon, Nmax);
	solution::clear_calls();

	Rosen(ff2T, test_x0, matrix(2, new double[2] {s, s}), alphaRosen, beta,
epsilon, Nmax);
	solution::clear_calls();

	SAVE_CHART_DATA = false;
#endif

#ifdef CALC_SIMULATION

	//Dane symulacji
	matrix ud1 = matrix(6, new double[6] {
		1.0, //Długość ramienia (l) [m]
		1.0, //Masa ramienia (m_r) [kg]
		5.0, //Masa ciężarka (m_c) [kg]
		0.5, //Współczynnik tarcia (b) [Nms]
		3.14, //Referencyjny kąt (alpha_ref) [rad]
		0.0 //Referencyjna prędkość kątowa (omega_ref) [rad/s]
	});

	//Początkowe wartości współczynników
	matrix k_0 = matrix(2, new double[2] {1.0, 1.0});

	//Warunki początkowe rónwe 0.0
	matrix Y0 = matrix(2, 1);

	//Wyliczanie optymalnych wartości współczynników metodą HJ
	solution opt = HJ(ff2R, k_0, s, alpha, epsilon, Nmax, ud1);
	std::cout << opt << "\n";
	solution::clear_calls();
	matrix* Y = solve_ode(df2, 0.0, 0.1, 100.0, Y0, ud1, opt.x);
#ifdef SAVE_TO_FILE
	save_to_file("simulation_HJ.csv", hcat(Y[0], Y[1]));
#endif

	//Wyliczanie optymalnych wartości współczynników metodą Rosenbrocka
	opt = Rosen(ff2R, k_0, matrix(2, new double[2]{s, s}), alphaRosen, beta,
epsilon, Nmax, ud1);
	std::cout << opt << "\n";
	solution::clear_calls();
	Y = solve_ode(df2, 0.0, 0.1, 100.0, Y0, ud1, opt.x);
#ifdef SAVE_TO_FILE
	save_to_file("simulation_Rosen.csv", hcat(Y[0], Y[1]));
#endif

#endif
}
```

Zaimplementowana funkcja lab2 w pliku main.cpp

3. **Parametry algorytmów**
   a) Metoda Hooke'a-Jeveesa
   Długość korku (s): 0.1

Współczynnik zmniejszania kroku (alpha): 0.1
Dokładność (epsilon): 1E-6
Maksymalna ilość wywołań funkcji (Nmax): 2000

b) Metoda Rosenbrocka
Wektor długości kroków (s): [0.1, 0.1]
Współczynnik ekspansji (alpha): 1.2
Współczynnik kontrakcji: 0.1
Dokładność (epsilon): 1E-6
Maksymalna ilość wywołań funkcji (Nmax): 2000

4. Dyskusja wyników

a) Wyniki testowe

Zarówno metoda Hooke'a Jeevesa jak i Rosenbrocka znalazła więcej minimów globalnych dla długości korku 1.3 (dane z tabeli 2). Nie zmalała przy tym dokładność wyników, wszystkie znalezione minima globalne są bliskie 0.0. Metoda Rosenbrocka cechuje się dużo większą liczbą wywołań funkcji celu. Podana testowa funkcja celu w zakresie [-1, 1] dla obu zmiennych, posiada 9 minimów w tym jedno globalne w punkcie (0,0). Z tego wynika niewielka liczba znalezionych minimów globalnych przez obie metody.

b) Wyniki problemu rzeczywistego

Metoda Hooke'a-Jeevesa i metoda Rosenbrocka znalazły bardzo zbliżone do siebie wyniki współczynników wzmocnienia ramienia. Jest to k1 = 2.8732 i k2 = 4,8817. Funkcjonał jakości dla obu metod wynosi Q = 193.938. Metoda Hooke'a-Jeevesa potrzebowała dużo mniej wywołań funkcji celu (171 w porównaniu z 903). Wyniki z symulacji pokazują że już w ok. 13 sekundzie ruchu ramię osiągnęło kąt równy PI dla obu metod. Z danych wynika że ramię wykonało większy kąt niż założono, ale potem wróciło do oczekiwanego kąta. Maksymalna prędkość kątowa została osiągnięta w pierwszych 5 sekundach ruchu i wynosiła ok. 1.07 rad/s.

5. Wnioski

Dzięki optymalizacji funkcji dwóch zmiennych za pomocą metod Hooke'a-Jeevesa i Rosenbrocka udało nam się rozwiązać problem rzeczywisty. Współczynniki wzmocnienia ramienia powinny wynosić ok. [2.8732, 4.8817], aby funkcjonał jakości był jak najmniejszy dla ruchu ramienia o PI stopni z prędkością kątową początkową równą 0 rad/s. Funkcjonał jakości wynosi wtedy: $Q(k_1, k_2) = 193.938$.