Jakub Latawiec

Rafał Malik

Optymalizacja

Laboratorium - optymalizacja z ograniczeniami funkcji wielu zmiennych metodami bezgradientowymi.

1. **Cel ćwiczenia**
   Celem ćwiczenia jest wykorzystanie bezgradientowych metod optymalizacji do wyznaczenia minimum funkcji celu uwzględniając ograniczenia.

2. **Przeprowadzenie ćwiczenia**
   a) Implementacja metody funkcji kary

```
solution pen(matrix(*ff)(matrix, matrix, matrix), matrix x0, double c,
double dc, double epsilon, int Nmax, matrix ud1, matrix ud2)
{
	try {
		solution XB;
		XB.x = x0;
		XB.fit_fun(ff, ud1, ud2);

		solution XT;
		XT = XB;

		double s = 0.5; //Długość boku trójkąta
		double alpha = 1.0; //Współczynnik odbicia
		double beta = 0.5; //Współczynnik zwężenia
		double gamma = 2.0; //Współczynnik ekspansji
		double delta = 0.5; //Współczynnik redukcji

		do
		{
			XT.x = XB.x;
			XT = sym_NM(ff, XB.x, s, alpha, beta, gamma, delta,
epsilon, Nmax, ud1, c);
			c *= dc;

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

			if (norm(XT.x - XB.x) < epsilon)
				break;

			XB = XT;
		} while (true);

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

b) Implementacja metody sympleks Neldera-Meada

```
solution sym_NM(matrix(*ff)(matrix, matrix, matrix), matrix x0, double s,
double alpha, double beta, double gamma, double delta, double epsilon, int
Nmax, matrix ud1, matrix ud2)
{
	try
	{
		//Funkcja pomocnicza do znajdywania maksymum normy
		auto max = [&](std::vector<solution> sim, int i_min) -> double
		{
			double result = 0.0;
			for (int i = 0; i < sim.size(); ++i)
			{
				double normal = norm(sim[i_min].x - sim[i].x);
				if (result < normal)
					result = normal;
			}
			return result;
		};

		int n = get_len(x0);

		//Tworzenie bazy ortogonalnej
		matrix d = matrix(n, n);
		for (int i = 0; i < n; ++i)
			d(i, i) = 1.0;

		//Tworzenie simplexu i uzupełnianie go danymi
		std::vector<solution> simplex;
		simplex.resize(n + 1);
		simplex[0].x = x0;
		simplex[0].fit_fun(ff, ud1, ud2);
		for (int i = 1; i < simplex.size(); ++i)
		{
			simplex[i].x = simplex[0].x + s * d[i - 1];
			simplex[i].fit_fun(ff, ud1, ud2);
		}

		//Indeks najmniejszej wartości wierzchołka simplexu
		int i_min{};
		//Indeks największej wartości wierzchołka simplexu
		int i_max{};

		while (max(simplex, i_min) >= epsilon)
		{
			//Wyznaczanie maksymalnego i minimalnego indeksu
			i_min = 0;
			i_max = 0;
			for (int i = 1; i < simplex.size(); ++i)
			{
				if (simplex[i].y < simplex[i_min].y)
					i_min = i;
				if (simplex[i].y > simplex[i_max].y)
					i_max = i;
			}

			//Wyznaczenie środka ciężkości
			matrix simplex_CoG{};
			for (int i = 0; i < simplex.size(); ++i)
			{
				if (i == i_max)
					continue;
```

```
                        simplex_CoG = simplex_CoG + simplex[i].x;
                }
                simplex_CoG = simplex_CoG / simplex.size();

                //Obliczanie wartości funkcji odbitego simplexu
                solution simplex_reflected{};
                simplex_reflected.x = simplex_CoG + alpha * (simplex_CoG
- simplex[i_max].x);
                simplex_reflected.fit_fun(ff, ud1, ud2);

                if (simplex_reflected.y < simplex[i_min].y)
                {
                        //Obliczanie wartości funkcji powiększonego
simplexu
                        solution simplex_expansion{};
                        simplex_expansion.x = simplex_CoG + gamma *
(simplex_reflected.x - simplex_CoG);
                        simplex_expansion.fit_fun(ff, ud1, ud2);
                        if (simplex_expansion.y < simplex_reflected.y)
                                simplex[i_max] = simplex_expansion;
                        else
                                simplex[i_max] = simplex_reflected;
                }
                else
                {
                        if (simplex[i_min].y <= simplex_reflected.y &&
simplex_reflected.y < simplex[i_max].y)
                                simplex[i_max] = simplex_reflected;
                        else
                        {
                                //Obliczanie wartości funkcji
pomniejszonego simplexu
                                solution simplex_narrowed{};
                                simplex_narrowed.x = simplex_CoG + beta *
(simplex[i_max].x - simplex_CoG);
                                simplex_narrowed.fit_fun(ff, ud1, ud2);
                                if (simplex_narrowed.y >= simplex[i_max].y)
                                {
                                        for (int i = 0; i < simplex.size();
++i)
                                        {
                                                if (i == i_min)
                                                        continue;
                                                simplex[i].x = delta *
(simplex[i].x + simplex[i_min].x);
                                                simplex[i].fit_fun(ff, ud1,
ud2);
                                        }
                                }
                                else
                                        simplex[i_max] = simplex_narrowed;
                        }
                }

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

        }
```

```
			return simplex[i_min];
		}
		catch (string ex_info)
		{
			throw ("solution sym_NM(...):\n" + ex_info);
		}
}
```

c) Implementacja testowej funkcji celu z zewnętrzną karą

```
matrix ff3T_outside(matrix x, matrix ud1, matrix ud2)
{
	matrix y;
	y = (sin(M_PI * sqrt(pow(x(0) / M_PI, 2) + pow(x(1) / M_PI, 2)))) /
(M_PI * sqrt(pow(x(0) / M_PI, 2) + pow(x(1) / M_PI, 2)));

	double a = m2d(ud1);
	double c = m2d(ud2);

	//g1
	if (-x(0) + 1 > 0)
		y = y + c * pow(-x(0) + 1, 2);
	//g2
	if (-x(1) + 1 > 0)
		y = y + c * pow(-x(1) + 1, 2);
	//g3
	if (norm(x) - a > 0)
		y = y + c * pow(norm(x) - a, 2);

	return y;
}
```

d) Implementacja testowej funkcji celu z wewnętrzną karą

```
matrix ff3T_inside(matrix x, matrix ud1, matrix ud2)
{
	matrix y;
	y = (sin(M_PI * sqrt(pow(x(0) / M_PI, 2) + pow(x(1) / M_PI, 2)))) /
(M_PI * sqrt(pow(x(0) / M_PI, 2) + pow(x(1) / M_PI, 2)));

	double a = m2d(ud1);
	double c = m2d(ud2);

	//g1
	if (-x(0) + 1 > 0)
		y = 1E10;
	else
		y = y - c / (-x(0) + 1);
	//g2
	if (-x(1) + 1 > 0)
		y = 1E10;
	else
		y = y - c / (-x(1) + 1);
	//g3
	if (norm(x) - a > 0)
		y = 1E10;
	else
		y = y - c / (norm(x) - a);

	return y;
}
```

e) Implementacja funkcji celu dla problemu rzeczywistego

```
matrix ff3R(matrix x, matrix ud1, matrix ud2)
{
	matrix y;
	matrix Y0(4, new double[4] {0.0, x(0), 100, 0});
	matrix* Y = solve_ode(df3, 0.0, 0.01, 7.0, Y0, ud1, x(1));
	int n = get_len(Y[0]);
	int i50 = 0;
	int i_0 = 0;
	for (int i = 0; i < n; ++i)
	{
		if (abs(Y[1](i, 2) - 50.0) < abs(Y[1](i50, 2) - 50.0))
			i50 = i;
		if (abs(Y[1](i, 2)) < abs(Y[1](i_0, 2)))
			i_0 = i;
	}

	y = -Y[1](i_0, 0);

	if (abs(x(0)) - 10 > 0)
		y = y + ud2 * pow(abs(x(0)) - 10, 2);
	if (abs(x(1)) - 15 > 0)
		y = y + ud2 * pow(abs(x(1)) - 15, 2);
	if (abs(Y[1](i50, 0) - 5.0) - 0.5 > 0)
		y = y + ud2 * pow(abs(Y[1](i50, 0) - 5.0) - 0.5, 2);

	return y;
}
```

f) Implementacja funkcji pochodnych dla problemu rzeczywistego

```
matrix df3(double t, matrix Y, matrix ud1, matrix ud2)
{
	//Wektor zmian po czasie
	matrix dY(4, 1);

	//Zmienne dane
	double x = Y(0);
	double v_x = Y(1);
	double y = Y(2);
	double v_y = Y(3);

	//Dane zadania
	double C = ud1(0);
	double rho = ud1(1);
	double r = ud1(2);
	double m = ud1(3);
	double g = ud1(4);

	double s = M_PI * pow(r, 2);

	double D_x = (1.0 / 2.0) * C * rho * s * v_x * abs(v_x);
	double D_y = (1.0 / 2.0) * C * rho * s * v_y * abs(v_y);
	double FM_x = rho * v_y * m2d(ud2) * M_PI * pow(r, 3);
	double FM_y = rho * v_x * m2d(ud2) * M_PI * pow(r, 3);

	dY(0) = v_x;
	dY(1) = (-D_x - FM_x) / m;
	dY(2) = v_y;
	dY(3) = ((- m * g) - D_y - FM_y) / m;

	return dY;
}
```

g) Implementacja funkcji lab3

```
void lab3()
{

#ifdef SAVE_TO_FILE
	create_environment("lab03");
#endif

	//Dane dokładnościowe
	double epsilon = 1E-3;
	int Nmax = 10000;
	double c_inside = 100;
	double dc_inside = 0.2;
	double c_outside = 1.0;
	double dc_outside = 1.5;


#ifdef CALC_TEST
	//Generator liczb losowych
	std::random_device rd;
	std::mt19937 gen(rd());
	std::uniform_real_distribution<> x0_dist(1.5, 5.5);

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

	//Rozwiązanie dla wyników testowych
	solution test_sol;

	//Dane a dla testów
	matrix a = matrix(4.0);

	//Punty startowe dla testów
	matrix test_x0{};

	for (int i = 0; i < 3; ++i)
	{
		if (i == 0)
			a = matrix(4.0);
		else if (i == 1)
			a = matrix(4.4934);
		else
			a = matrix(5.0);

		for (int j = 0; j < 100; ++j)
		{
			test_x0 = matrix(2, new double[2] {x0_dist(gen),
x0_dist(gen)});
			test_ss << test_x0(0) << ";" << test_x0(1) << ";";
			test_sol = pen(ff3T_outside, test_x0, c_outside,
dc_outside, epsilon, Nmax, a);
			test_ss << test_sol.x(0) << ";" << test_sol.x(1) << ";"
<< sqrt(pow(test_sol.x(0), 2) + pow(test_sol.x(1), 2)) << ";" << test_sol.y
<< test_sol.f_calls << ";";
			solution::clear_calls();
			test_sol = pen(ff3T_inside, test_x0, c_inside,
dc_inside, epsilon, Nmax, a);
			test_ss << test_sol.x(0) << ";" << test_sol.x(1) << ";"
<< sqrt(pow(test_sol.x(0), 2) + pow(test_sol.x(1), 2)) << ";" << test_sol.y
<< test_sol.f_calls << "\n";
			solution::clear_calls();
		}
```

```
#ifdef SAVE_TO_FILE
            save_to_file("test_a_" + std::to_string(m2d(a)) + ".csv",
test_ss.str());
#endif

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

#endif

#ifdef CALC_SIMULATION
      //Dane zadania
      matrix ud1 = matrix(5, new double[5] {
            0.47, //Współczynnik oporu (C) [-]
            1.2, //Gęstość powietrza (rho) [kg/m^3]
            0.12, //Promień piłki (r) [m]
            0.6, //Masa piłki (m) [kg]
            9.81 //Przyśpieszenie ziemskie (g) [m/s^2]
      });

      //Początkowe wartości szukania minimum
      matrix x0 = matrix(2, new double[2] {-5.0, 5.0});

      //Szukanie optymalnej prędkości początkowej po osi x i początkowej
prędkości obrotowej
      solution opt = pen(ff3R, x0, c_outside, dc_outside, epsilon, Nmax,
ud1);
      std::cout << opt << "\n";

      //Symulacja lotu piłki dla wyznaczonych ograniczeń
      matrix Y0(4, new double[4] {0.0, opt.x(0), 100, 0});
      matrix* Y = solve_ode(df3, 0.0, 0.01, 7.0, Y0, ud1, opt.x(1));

#ifdef SAVE_TO_FILE
      save_to_file("simulation.csv", hcat(Y[0], Y[1]));
#endif

#endif
}
```

3. **Parametry algorytmów**
   a) <u>Ogólne</u>
      Dokładność (epsilon): 1E-3
      Maksymalna ilość wywołań funkcji celu (nmax): 10 000

   b) <u>Funkcje z karą zewnętrzną</u>
      Wartość kary (c): 1.0
      Współczynnik kary (dc): 1.5

   c) <u>Funkcje z karą wewnętrzną</u>
      Wartość kary (c): 100
      Współczynnik kary (dc): 0.2

   d) <u>Metoda sympleks Neldera-Meada</u>
      Długość boku trójkąta (s): 0.5
      Współczynnik odbicia (alpha): 1.0

Współczynnik zwężenia (beta): 0.5
Współczynnik ekspansji (gamma): 2.0
Współczynnik redukcji (delta): 0.5

4. Dyskusja wyników

a) Wyniki testowe

Wraz ze wzrostem parametru „a" wzrasta dokładność szukania minimów metodami z karą zewnętrzną i wewnętrzną. W przypadku funkcji z karą zewnętrzną, malała również liczba wywołań funkcji w przeciwieństwie do funkcji z karą wewnętrzną. Dla a = 4.0 i a = 4.4934 funkcja z karą wewnętrzną parę razy zwróciła wyniki gdzie wartość funkcji wynosiła 1E10. Wynika to z wyjścia poza zakres funkcji (ustawiamy wtedy wartość kary na 1E10) i spełnieniu warunku wyjścia z funkcji pen(). Jedynie w przypadku gdy „a" było równe 5.0 wszystkie znalezione minima znajdowały się w zakresie.

b) Wyniki problemu rzeczywistego

Dla początkowych wartości prędkości początkowej po osi x oraz prędkości kątowa równych [-5.0, 5.0] zostały znalezione wartości równe [1.48, 0.29]. Liczba wywołań funkcji celu wynosiła 832. Znalezione wartości spełniają dwa zadane warunki: prędkość początkowa po osi x jest w zakresie [-10, 10] oraz prędkość kątowa jest w zakresie [-15, 15]. Po przeprowadzeniu symulacji na znalezionych wartościach, możemy zauważyć że spełniony jest trzeci warunek: dla y = 50m, x znajduje się w zakresie [4.5, 5.5]. Dokładnie x wynosi: 5.48 dla y = 50.12 i 5.50 dla y = 4.92.

5. Wnioski

Dzięki optymalizacji funkcji dwóch zmiennych z ograniczeniami zewnętrznej funkcji kary udało się nam wyznaczyć początkowe wartości prędkości początkowej po osi x (równej 1.48 m/s) i początkowej prędkości kątowej (równej 0.29 rad/s). Wszystkie zadane warunki są spełnione. Po przeprowadzeniu symulacji jesteśmy w stanie narysować trajektorię lotu piłki.