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.

1. **Przeprowadzenie ćwiczenia**
2. 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);

}

}

1. 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);

}

}

1. 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;

}

1. 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;

}

1. 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;

}

1. 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;

}

1. 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

}

1. **Parametry algorytmów**
2. Ogólne

Dokładność (epsilon): 1E-3

Maksymalna ilość wywołań funkcji celu (nmax): 10 000

1. Funkcje z karą zewnętrzną

Wartość kary (c): 1.0

Współczynnik kary (dc): 1.5

1. Funkcje z karą wewnętrzną

Wartość kary (c): 100

Współczynnik kary (dc): 0.2

1. Metoda sympleks Neldera-Meada

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

1. **Dyskusja wyników**
2. 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.

1. 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.

1. **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.