Jakub Latawiec

Rafał Malik

Kacper Krok

Optymalizacja

Laboratorium - optymalizacja funkcji jednej zmiennej metodami bezgradientowymi

1. **Cel ćwiczenia**

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

1. **Przeprowadzenie ćwiczenia**
2. Stworzenie dodatkowego pliku configuration.h

Plik configuration.h zawiera szereg zmiennych globalnych oraz dyrektyw definiujących które części kodu się wykonają. Nie ma on wpływu na sposób w jaki liczone są dane, jest on tylko ze względu na wygodę pracy dla użytkownika rozwiązującego problem.

#pragma once

#include <string>

//OBLICZENIA

#define CALC\_TEST

#define CALC\_SIMULATION

//ZAPIS DO PLIKU

#define SAVE\_TO\_FILE

inline bool SAVE\_CHART\_DATA = false;

inline const std::string DATA\_PATH = "../data/";

inline std::string FILE\_PATH{};

*zawartość pliku configuration.h*

1. Stworzenie dodatkowych plików environment.h i environment.cpp

Pliki te stworzone zostały aby w szybki sposób dokonywać zapisu do pliku. Funkcja create\_environment(string) w odpowiednim miejscu projektu tworzy folder o podanej nazwie. Wszystkie pliki z danymi będą zapisywane w folderze o dacie uruchomienia programu. Czyli w naszym przypadku stworzyliśmy środowisko „lab1” i wszystkie dane wynikowe zostaną zapisane w data/lab1/YYYY-MM-DD\_HH-MM-SS. Funkcja save\_to\_file(string, string) jest abstrakcją zapisywania do plików wynikowych aktualnego środowiska. Te pliki również nie mają wpływu na obliczenia wykonywane przez program, są wykorzystywane do wygodniejszego zarządzania wyjściowymi wynikami.

#pragma once

#include <string>

#include "matrix.h"

void create\_environment(std::string env\_name);

void save\_to\_file(std::string filename, std::string& data);

void save\_to\_file(std::string filename, matrix& data);

*zawartość pliku environment.h*

#include "environment.h"

#include "configuration.h"

#include <filesystem>

#include <sstream>

#include <fstream>

void create\_environment(std::string env\_name)

{

//Tworzenie folderu jeśli ten nie istnieje

if (!std::filesystem::exists(DATA\_PATH + env\_name))

std::filesystem::create\_directory(DATA\_PATH + env\_name);

//Tworzenie stringa który jest czasem i datą wykonywania programu

auto now = std::chrono::system\_clock::now();

std::time\_t now\_time = std::chrono::system\_clock::to\_time\_t(now);

std::tm\* local\_time = std::localtime(&now\_time);

std::ostringstream date;

date << std::put\_time(local\_time, "%Y-%m-%d\_%H-%M-%S");

//Tworzenie folderu z datą

std::filesystem::create\_directory(DATA\_PATH + env\_name + "/" + date.str());

//Przypisanie folderu z plikami dla stworzonego środowiska

FILE\_PATH = DATA\_PATH + env\_name + "/" + date.str() + "/";

}

void save\_to\_file(std::string filename, std::string& data)

{

std::ofstream file(FILE\_PATH + filename);

file << data;

file.close();

}

void save\_to\_file(std::string filename, matrix& data)

{

std::ofstream file(FILE\_PATH + filename);

file << data;

file.close();

}

*zawartość pliku environment.cpp*

1. Implementacja metody ekspansji

Wykorzystując klasy solution i matrix, oraz pseudokod znajdujący się w skrypcie, została zaimplementowana funkcja expansion(). Zawęża ona przedział funkcji na jakim znajduje się jej minimum.

double\* expansion(matrix(\*ff)(matrix, matrix, matrix), double x0, double d, double alpha, int Nmax, matrix ud1, matrix ud2)

{

try

{

int i = 0;

solution xi\_sol, xi\_next\_sol;

double xi, xi\_next;

xi = x0;

xi\_next = xi + d;

xi\_sol.x = xi;

xi\_sol.fit\_fun(ff, ud1);

xi\_next\_sol.x = xi\_next;

xi\_next\_sol.fit\_fun(ff, ud1);

if (xi\_next\_sol.y == xi\_sol.y)

return new double[3] {xi, xi\_next, (double)solution::f\_calls};

if (xi\_next\_sol.y > xi\_sol.y)

{

d = -d;

xi\_next = xi + d;

xi\_next\_sol.x = xi\_next;

xi\_next\_sol.fit\_fun(ff, ud1);

if (xi\_next\_sol.y >= xi\_sol.y)

return new double[3] {xi\_next, xi - d, (double)solution::f\_calls};

}

solution::clear\_calls();

double xi\_prev{};

double f\_xi = m2d(xi\_sol.y);

do

{

if (solution::f\_calls > Nmax)

{

xi\_next\_sol.flag = 0;

throw std::string("Maximum amount of f\_calls reached!");

}

++i;

xi\_next = xi + pow(alpha, i) \* d;

xi\_next\_sol.x = xi\_next;

xi\_next\_sol.fit\_fun(ff, ud1);

if (!(f\_xi > xi\_next\_sol.y))

break;

xi\_prev = xi;

xi = xi\_next;

f\_xi = m2d(xi\_next\_sol.y);

} while (true);

if (d > 0)

return new double[3] {xi\_prev, xi\_next, (double)solution::f\_calls};

return new double[3] {xi\_next, xi\_prev, (double)solution::f\_calls};

}

catch (string ex\_info)

{

throw ("double\* expansion(...):\n" + ex\_info);

}

}

*zaimplementowana metoda ekspansji w pliku opt\_alg.cpp*

1. Implementacja metody interpolacji Fibbonacciego.

solution fib(matrix(\*ff)(matrix, matrix, matrix), double a, double b, double epsilon, matrix ud1, matrix ud2)

{

try

{

std::stringstream ss;

std::vector<double> sigma = { 1, 1 };

double ratio = (b - a) / epsilon;

while (true)

{

if (sigma.back() > ratio)

break;

sigma.push\_back(sigma[sigma.size() - 1] + sigma[sigma.size() - 2]);

}

int k = sigma.size() - 1;

double a0 = a;

double b0 = b;

double c0 = b0 - sigma[k - 1] / sigma[k] \* (b0 - a0);

double d0 = a0 + b0 - c0;

solution c\_sol, d\_sol;

for (int i = 0; i <= k - 3; ++i)

{

c\_sol.x = c0;

c\_sol.fit\_fun(ff, ud1);

d\_sol.x = d0;

d\_sol.fit\_fun(ff, ud1);

if (c\_sol.y < d\_sol.y)

b0 = d0;

else

a0 = c0;

c0 = b0 - sigma[k - i - 2] / sigma[k - i - 1] \* (b0 - a0);

d0 = a0 + b0 - c0;

if (SAVE\_CHART\_DATA)

ss << i << ";" << b0 - a0 << ";\n";

}

solution Xopt;

Xopt.x = c0;

Xopt.fit\_fun(ff, ud1);

if (SAVE\_CHART\_DATA)

save\_to\_file("fibonacci\_chart.csv", ss.str());

return Xopt;

}

catch (string ex\_info)

{

throw ("solution fib(...):\n" + ex\_info);

}

}

*zaimplementowana metoda interpolacji Fibonacciego w pliku opt\_alg.cpp*

1. Implementacja metody interpolacji Lagrange’a

solution lag(matrix(\*ff)(matrix, matrix, matrix), double a, double b, double epsilon, double gamma, int Nmax, matrix ud1, matrix ud2)

{

try

{

std::stringstream ss;

solution Xopt;

double ai = a;

double bi = b;

double ci = (a + b) / 2;

double di{};

int i = 0;

double l{}, m{};

solution ai\_sol, bi\_sol, ci\_sol, di\_sol;

double l\_prev{}, m\_prev{}, di\_prev{};

do

{

ai\_sol.x = ai;

ai\_sol.fit\_fun(ff, ud1);

bi\_sol.x = bi;

bi\_sol.fit\_fun(ff, ud1);

ci\_sol.x = ci;

ci\_sol.fit\_fun(ff, ud1);

l = m2d(ai\_sol.y) \* (pow(bi, 2) - pow(ci, 2)) + m2d(bi\_sol.y) \* (pow(ci, 2) - pow(ai, 2)) + m2d(ci\_sol.y) \* (pow(ai, 2) - pow(bi, 2));

m = m2d(ai\_sol.y) \* (bi - ci) + m2d(bi\_sol.y) \* (ci - ai) + m2d(ci\_sol.y) \* (ai - bi);

if (m <= 0)

{

Xopt.flag = 0;

break;

}

di = 0.5 \* l / m;

di\_sol.x = di;

di\_sol.fit\_fun(ff, ud1);

if (ai < di && di < ci)

{

if (di\_sol.y < ci\_sol.y)

{

bi = ci;

ci = di;

}

else

ai = di;

}

else

{

if (ci < di && di < bi)

{

if (di\_sol.y < ci\_sol.y)

{

ai = ci;

ci = di;

}

else

bi = di;

}

else

{

Xopt.flag = 0;

break;

}

}

if (ai\_sol.f\_calls > Nmax)

{

Xopt.flag = 0;

throw std::string("Error message!");

break;

}

if (i > 0)

{

di\_prev = 0.5 \* l\_prev / m\_prev;

}

l\_prev = l;

m\_prev = m;

if (SAVE\_CHART\_DATA)

ss << i << ";" << bi - ai << ";\n";

++i;

} while (!(bi - ai < epsilon || abs(di - di\_prev) < gamma));

Xopt.x = di;

Xopt.fit\_fun(ff, ud1);

if (SAVE\_CHART\_DATA)

save\_to\_file("lagrange\_chart.csv", ss.str());

return Xopt;

}

catch (string& ex\_info)

{

throw ("solution lag(...):\n" + ex\_info);

}

}

*zaimplementowana metoda interpolacji Lagrange’a w pliku opt\_alg.cpp*

1. Implementacja testowej funkcji celu

matrix ff1T(matrix x, matrix ud1, matrix ud2)

{

matrix y;

y = -cos(0.1 \* m2d(x)) \* exp(-1.0 \* pow((0.1 \* m2d(x) - 2 \* M\_PI), 2)) + 0.002 \* pow(0.1 \* m2d(x), 2);

return y;

}

*zaimplementowana testowa funkcja celu w pliku user\_funs.cpp*

1. Implementacja funkcji celu dla problemu rzeczywistego

matrix ff1R(matrix x, matrix ud1, matrix ud2)

{

matrix y;

matrix Y0 = matrix(3, new double[3] {5.0, 1.0, 20.0});

matrix\* Y = solve\_ode(df1, 0, 1, 2000, Y0, ud1, x);

int n = get\_len(Y[0]);

double max = Y[1](0, 2);

for (int i = 0; i < n; ++i)

{

if (max < Y[1](i, 2))

max = Y[1](i, 2);

}

y = abs(max - 50.0);

Y[0].~matrix();

Y[1].~matrix();

return y;

}

*zaimplementowana funkcja celu dla problemu rzeczywistego w pliku user\_funs.cpp*

1. Implementacja funkcji pochodnych dla problemu rzeczywistego

matrix df1(double t, matrix Y, matrix ud1, matrix ud2)

{

matrix dY(3, 1);

const double a = 0.98;

const double b = 0.63;

const double g = 9.81;

double Va = Y(0);

double Vb = Y(1);

double Tb = Y(2);

double Pa = ud1(0);

double Ta = ud1(1);

double Pb = ud1(2);

double Db = ud1(3);

double F\_in = ud1(4);

double T\_in = ud1(5);

double Da = m2d(ud2(0));

double Fa\_out = a \* b \* Da \* sqrt(2 \* g \* Va / Pa);

if (Va <= 0.0)

Fa\_out = 0.0;

double Fb\_out = a \* b \* Db \* sqrt(2 \* g \* Vb / Pb);

if (Vb <= 0.0)

Fb\_out = 0.0;

dY(0) = -Fa\_out;

dY(1) = Fa\_out + F\_in - Fb\_out;

if (Vb > 0)

dY(2) = (F\_in / Vb) \* (T\_in - Tb) + (Fa\_out / Vb) \* (Ta - Tb);

else

dY(2) = 0;

if (Y(0) + dY(0) < 0)

dY(0) = -Y(0);

if (Y(1) + dY(1) < 0)

dY(1) = -Y(1);

return dY;

}

*zaimplementowana funkcja pochodnych dla problemu rzeczywistego w pliku user\_funs.cpp*

1. Implementacja funkcji lab1()

W funkcji lab1 liczone są dane testowe jak i problem rzeczywisty. Można włączać i wyłączać dane bloki obliczeń ustawiając odpowiednie dane w pliku configuration.h.

void lab1()

{

#ifdef SAVE\_TO\_FILE

create\_environment("lab01");

#endif

//Dane dokładności wyników

double epsilon = 1e-18;

double gamma = 1e-30;

int Nmax = 200;

double d = 0.01;

double alpha = 1.1;

//Funkcja testowa

#ifdef CALC\_TEST

//Generator losowania liczb

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> x0\_dist(0.0, 100.0);

//Stringstream do zapisu danych

std::stringstream test\_ss;

//Solution dla testów

solution test\_opt;

double test\_alpha = 1.5;

for (int j = 0; j < 3; ++j)

{

//Liczenie ekspansji, fibonacciego i lagrange'a dla danego współczynnika alpha

for (int i = 0; i < 100; ++i)

{

double x0 = x0\_dist(gen);

double\* bounds = expansion(ff1T, x0, d, test\_alpha, Nmax);

test\_ss << x0 << ";" << bounds[0] << ";" << bounds[1] << ";" << bounds[2] << ";";

solution::clear\_calls();

test\_opt = fib(ff1T, bounds[0], bounds[1], epsilon);

test\_ss << m2d(test\_opt.x) << ";" << m2d(test\_opt.y) << ";" << test\_opt.f\_calls << ";" << (test\_opt.x > -1 && test\_opt.x < 1 ? "lokalne" : "globalne") << ";";

solution::clear\_calls();

test\_opt = lag(ff1T, bounds[0], bounds[1], epsilon, gamma, Nmax);

test\_ss << m2d(test\_opt.x) << ";" << m2d(test\_opt.y) << ";" << test\_opt.f\_calls << ";" << (test\_opt.x > -1 && test\_opt.x < 1 ? "lokalne" : "globalne") << ";\n";

solution::clear\_calls();

}

//Zapis do pliku

#ifdef SAVE\_TO\_FILE

save\_to\_file("test\_alpha\_" + std::to\_string(test\_alpha) + ".csv", test\_ss.str());

#endif

//Czyszczenie zawartości ss

test\_ss.str(std::string());

//Zmiana alfy

test\_alpha += 1.3;

}

//Czyszczenie zawartości ss

test\_ss.str(std::string());

//Obliczanie minimum metodą Fibonacci'ego

SAVE\_CHART\_DATA = true;

test\_opt = fib(ff1T, -100.0, 100.0, epsilon);

test\_ss << m2d(test\_opt.x) << ";" << m2d(test\_opt.y) << ";" << test\_opt.f\_calls << ";" << (test\_opt.x > -1 && test\_opt.x < 1 ? "lokalne" : "globalne") << ";";

solution::clear\_calls();

//Obliczanie minimum metodą Lagrange'a

test\_opt = lag(ff1T, -100, 100, epsilon, gamma, Nmax);

test\_ss << m2d(test\_opt.x) << ";" << m2d(test\_opt.y) << ";" << test\_opt.f\_calls << ";" << (test\_opt.x > -1 && test\_opt.x < 1 ? "lokalne" : "globalne") << ";\n";

solution::clear\_calls();

SAVE\_CHART\_DATA = false;

//Zapis do pliku

#ifdef SAVE\_TO\_FILE

save\_to\_file("test\_no\_expansion.csv", test\_ss.str());

#endif

#endif

#ifdef CALC\_SIMULATION

//Dane do problemu rzeczywistego

matrix ud1 = matrix(6, 1);

ud1(0) = 0.5; //Pa

ud1(1) = 90.0; //Ta

ud1(2) = 1.0; //Pb

ud1(3) = 36.5665 \* 0.0001; //Db

ud1(4) = 10 \* 0.001; //F\_in

ud1(5) = 20.0; //T\_in

//Zakres szukania Da

double Da\_0\_s = 1.0 \* 0.0001;

double Da\_0\_f = 100 \* 0.0001;

//Szukanie minimum

solution opt = fib(ff1R, Da\_0\_s, Da\_0\_f, epsilon, ud1);

std::cout << opt;

solution::clear\_calls();

//Warunki początkowe

matrix Y0 = matrix(3, 1);

Y0(0) = 5.0; //Poczatkowa objetosc w a

Y0(1) = 1.0; //Poczatkowa objetosc w b

Y0(2) = 20.0;//Poczatkowa temperatura w b

//Symulacja

matrix\* Y = solve\_ode(df1, 0, 1, 2000, Y0, ud1, opt.x);

#ifdef SAVE\_TO\_FILE

save\_to\_file("simulation\_fibonacci.csv", hcat(Y[0], Y[1]));

#endif

//Szukanie maksymalnej temperatury

int n = get\_len(Y[0]);

double Tb\_max = Y[1](0, 2);

for (int i = 0; i < n; ++i)

{

if (Tb\_max < Y[1](i, 2))

Tb\_max = Y[1](i, 2);

}

std::cout << "Tb\_max (fibonacci): " << Tb\_max << "\n\n";

//Szukanie minimum

opt = lag(ff1R, Da\_0\_s, Da\_0\_f, epsilon, epsilon, Nmax, ud1);

std::cout << opt;

solution::clear\_calls();

//Symulacja

Y = solve\_ode(df1, 0, 1, 2000, Y0, ud1, opt.x);

#ifdef SAVE\_TO\_FILE

save\_to\_file("simulation\_lagrange.csv", hcat(Y[0], Y[1]));

#endif

//Szukanie maksymalnej temperatury

n = get\_len(Y[0]);

Tb\_max = Y[1](0, 2);

for (int i = 0; i < n; ++i)

{

if (Tb\_max < Y[1](i, 2))

Tb\_max = Y[1](i, 2);

}

std::cout << "Tb\_max (lagrange): " << Tb\_max << "\n";

Y[0].~matrix();

Y[1].~matrix();

#endif

}

*zaimplementowana funkcja lab1 w pliku main.cp*p

1. **Parametry algorytmów**
2. Metoda ekspansji

Nmax = 200

d = 0.01

alpha = 1.5, 2.8, 4.1

1. Metoda Fibonacciego

Epsilon = 1e-18

1. Metoda Lagrange’a

Nmax = 200

Epsilon = 1e-18

Gamma = 1e-30

1. **Dyskusja wyników**
2. Wyniki testowe

Ze zwiększeniem współczynnika ekspansji, rośnie również długość zakresu wynikowego, lecz maleje liczba wywołań funkcji celu (im mniejszy współczynnik tym dokładniejsze wyniki i więcej obliczeń). Wyliczone przedziały miały wpływ na znajdowanie minimum przez metody Fibonacciego i Lagrange’a. Dla wyższych wartości współczynnika ekspansji, wyniki były mniej dokładne. Ponadto metoda Lagrange’a cechuje się występowaniem braku zbieżności. W naszym przypadku były to dwa wyniki dla alfy równej 4,1. Jedno minimum wynosiło (x: -3298,95 y: 217,6620) a drugie (x: -353,117 y: 2,4938).  
Szukanie minimum bez metody ekspansji oboma sposobami, przyniosło podobne rezultaty, zostało znalezione minimum lokalne w okolicach x=0,0. Metoda Lagrange’a potrzebowała jednak dużo mniej iteracji w liczeniu zbieżności niż metoda Fibonacciego (chodzi o iteracje „i” a nie f\_calls).

1. Wyniki problemu rzeczywistego

Metody Lagrange’a i Fibonacciego znalazły bardzo zbliżony wynik w szukanym przez nas polu DA. W obu przypadkach, dla podanych w poprzednim punkcie dokładności, DA wyniosło ok. 0,00116768m2. Przeglądając dane symulacji możemy wywnioskować że maksymalna temperatura w zbiorniku B dla wyliczonego DA wyniosła 50 stopni Celsjusza w 255 sekundzie. Pod koniec symulacji temperatura wody w zbiorniku B wynosiła ok. 20 stopni Celsjusza, ponieważ cała woda ze zbiornika A wylała się ok. 988 sekundy i zbiornik otrzymywał już tylko wodę o temperaturze 20 stopni Celsujsza z kranu.

1. **Wnioski**

Dzięki optymalizacji funkcji jednej zmiennej za pomocą metod Lagrange’a i Fibonacciego udało nam się rozwiązać problem rzeczywisty. Pole DA powinno wynosić 0,00116768m2, aby maksymalna temperatura w zbiorniku B wynosiła 50 stopni Celsjusza.