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.

2. **Przeprowadzenie ćwiczenia**
   a) 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

   b) 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

```cpp
#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

c) 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.

```cpp
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

d) 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

e) 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

f) 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

g) 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

h) 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

i) 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

3. Parametry algorytmów

a) Metoda ekspansji
   Nmax = 200
   d = 0.01
   alpha = 1.5, 2.8, 4.1

b) Metoda Fibonacciego
   Epsilon = 1e-18

c) Metoda Lagrange'a
   Nmax = 200
   Epsilon = 1e-18
   Gamma = 1e-30

4. Dyskusja wyników

a) 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).

b) 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. 0,00116768m^2. 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.

5. Wnioski

Dzięki optymalizacji funkcji jednej zmiennej za pomocą metod Lagrange'a i Fibonacciego udało nam się rozwiązać problem rzeczywisty. Pole DA powinno wynosić 0,00116768m^2, aby maksymalna temperatura w zbiorniku B wynosiła 50 stopni Celsjusza.