Отчет по дисциплине: «Численные методы»

Лабораторная работа №9

## «Метод Ньютона решения нелинейных уравнений»

Подготовил студент 3 курса 4 группы

Кондратович Артём

Цель работы:

Изучить основы метода деления отрезка пополам и метода Ньютона решения нелинейных уравнений; разработать (для модельного примера) программу, реализующую методы. Проверить некоторые положения (какие – в зависимости от промежуточных результатов), связанные с темами получения собственных значений и собственных векторов матрицы.

Задание:

Вычислить вещественные корни собственного многочлена четвертой степени P(λ), полученного из канонической формы Фробениуса лабораторной работы «Метод Данилевского». Проверить некоторые положения (далее уточняется какие), связанные с темами получения собственных значений и собственных векторов матрицы.

1. Произвести отделение корней многочлена P(λ).

Для определения промежутков монотонности функции P(λ) решить уравнение P'(λ)=0: а) применить метод деления отрезка пополам, б) применить метод Ньютона. Предварительно произвести отделение корней многочлена P'(λ). Если требуется, при отделении корней многочлена P'(λ) найти промежутки монотонности функции P'(λ): решить квадратное уравнение P''(λ)=0.

Если отделения корней многочлена P(λ) не произошло, то вещественных корней многочлен P(λ) не имеет. Тогда два (наибольших по модулю) корня образуют комплексно-сопряженную пару и имеет место Случай 4 файла «Степенной метод». Проверить, справедливо ли Замечание 9: Характерным признаком Случая 4 является колебательный характер последовательности приближений.

2. Приближенно вычислить вещественные корни собственного многочлена методом Ньютона.

3. Хотя бы для одного найденного собственного значения (корня собственного многочлена): с помощью матриц M3, M2, M1, полученных в лабораторной работе «Метод Данилевского», найти соответствующий собственный вектор u. Проверить Au–λu≈0.

Прокомментировать (можно устно) выходные данные и листинг программы.

Листинг программы:

#include "Operations.h"

#include <string>

#include <random>

const int n = 4;

const float epsilon = (float)std::pow(10, -8);

const double border = 1E-6;

const std::vector<float> frobenius = { -51.61526489f, -61.52075195f, 14011.79687500f, 3981367.25000000f };

static float GetRandomFromRange(float a, float b)

{

static std::random\_device rd;

static std::mt19937 generator(rd());

std::uniform\_real\_distribution<float> distribution(a, b);

return distribution(generator);

}

static std::vector<std::vector<float>> GenerateA()

{

std::vector<std::vector<float>> matrix(n, std::vector<float>(n));

for (auto& line : matrix)

{

for (auto& element : line)

{

element = GetRandomFromRange(-50, 50);

}

}

return matrix;

}

static float Trace(const std::vector<std::vector<float>>& matrix)

{

float trace = 0;

for (size\_t i = 0; i < n; i++)

{

trace += matrix[i][i];

}

return trace;

}

static std::vector<std::vector<float>> GenerateE()

{

std::vector<std::vector<float>> e(n, std::vector<float>(n));

for (size\_t i = 0; i < n; i++)

{

e[i][i] = 1;

}

return e;

}

static std::vector<std::vector<float>> InverseMatrixM(const std::vector<std::vector<float>>& a, int i)

{

auto e = GenerateE();

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

{

e[i - 1][j] = a[i][j];

}

return e;

}

static std::vector<std::vector<float>> MatrixM(const std::vector<std::vector<float>>& a, float leading, int i)

{

auto e = GenerateE();

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

{

if (j != i - 1)

{

e[i - 1][j] = -(a[i][j] / leading);

}

else

{

e[i - 1][j] = 1 / leading;

}

}

return e;

}

static float CheckResult(float sourceTrace, float resultTrace)

{

return std::fabs(sourceTrace - resultTrace);

}

static void Printreport(const std::vector<std::vector<float>>& frobenius, const std::vector<std::pair<std::string, std::vector<std::vector<float>>>>& matrices, const float& sourceTrace)

{

std::cout << "Frobenius" << std::endl;

std::cout << frobenius;

for (auto const& m : matrices)

{

std::cout << m.first << std::endl;

std::cout << m.second;

}

std::cout << "p1 = " << frobenius[0][0] << std::endl;

std::cout << "|sourceTrace - p1| = " << CheckResult(sourceTrace, frobenius[0][0]) << std::endl;

}

static bool DanilevskiMethod(std::vector<std::vector<float>> a, std::vector<std::pair<std::string, std::vector<std::vector<float>>>>& matrices, std::vector<std::vector<float>>& frobenius)

{

for (int i = n - 1; i >= 1; i--)

{

if (std::fabs(a[i][i - 1]) < epsilon)

{

return false;

}

auto m = MatrixM(a, a[i][i - 1], i);

matrices.emplace\_back("M" + std::to\_string(i), m);

auto m1 = InverseMatrixM(a, i);

a = m1 \* a \* m;

}

frobenius = a;

return true;

}

static std::vector<std::vector<float>> Solve()

{

std::vector<std::vector<float>> frobenius(n, std::vector<float>(n));

std::vector<std::pair<std::string, std::vector<std::vector<float>>>> matrices;

while (true)

{

auto a = GenerateA();

std::cout << "Start" << std::endl;

std::cout << "Source matrix A" << std::endl;

std::cout << a;

auto trace = Trace(a);

if (DanilevskiMethod(a, matrices, frobenius))

{

Printreport(frobenius, matrices, trace);

break;

}

std::cout << "Exception! One of the leading elements less than 10^-8" << std::endl;

if (!matrices.empty())

{

matrices.clear();

}

}

return frobenius;

}

static std::vector<float> GetPlambda()

{

std::vector<float> p\_lambda(n + 1);

p\_lambda[0] = 1.f;

for (int i = 0; i < frobenius.size(); i++)

{

p\_lambda[i + 1] = -frobenius[i];

}

return p\_lambda;

}

static double CountPlambda(double x0, const std::vector<float>& pLambda)

{

double result = 0.;

for (size\_t i = 0; i < pLambda.size(); i++)

{

result += pLambda[i] \* std::pow(x0, pLambda.size() - 1 - i);

}

return result;

}

static int DichotomyMethod(double x0, double x1, double& x, const std::vector<float>& pLambda) {

int rotation;

if (sgn(CountPlambda(x0, pLambda)) < 0 && sgn(CountPlambda(x1, pLambda)) > 0)

rotation = 1;

else if (sgn(CountPlambda(x0, pLambda)) > 0 && sgn(CountPlambda(x1, pLambda)) < 0)

rotation = -1;

else

return 0;

double delta\_k = (x1 - x0) / 2;

double x\_k = (x0 + x1) / 2;

int iterations = 0;

while (delta\_k > border) {

delta\_k /= 2;

x\_k -= delta\_k \* sgn(CountPlambda(x\_k, pLambda)) \* rotation;

++iterations;

}

x = x\_k;

return iterations;

}

static int NewtonMethod(double x0, double x1, double& x, const std::vector<float>& pLambda, const std::vector<float>& derivative) {

double x\_k = x1;

double x\_k\_1 = x0;

int iterations = 0;

while (std::abs(x\_k - x\_k\_1) > border) {

x\_k = x\_k\_1;

x\_k\_1 -= CountPlambda(x\_k, pLambda) / CountPlambda(x\_k, derivative);

++iterations;

}

x = x\_k\_1;

return iterations;

}

static std::vector<float> TakeDerivative(const std::vector<float>& polinom)

{

int k = 0;

std::vector<float> derivative(polinom.size() - 1);

for (int i = 0; i < polinom.size() - 1; i++)

{

derivative[i] = (polinom.size() - 1 - i) \* polinom[k++];

}

return derivative;

}

static std::vector<double> solveCubicEquation(double p1, double p2, double p3) {

std::vector<double> solutions;

// Calculate discriminant

double Q = (3 \* p2 - std::pow(p1, 2)) / 9;

double R = (9 \* p1 \* p2 - 27 \* p3 - 2 \* std::pow(p1, 3)) / 54;

double discriminant = std::pow(Q, 3) + std::pow(R, 2);

// Check discriminant for real solutions

if (discriminant > 0) {

double S = std::cbrt(R + std::sqrt(discriminant));

double T = std::cbrt(R - std::sqrt(discriminant));

// Calculate solutions

double x1 = S + T - p1 / 3;

solutions.push\_back(x1);

}

else if (discriminant == 0) {

double K = R >= 0 ? -std::cbrt(R) : std::cbrt(-R);

double x1 = 2 \* K - p1 / 3;

double x2 = -K - p1 / 3;

solutions.push\_back(x1);

solutions.push\_back(x2);

}

return solutions;

}

int main()

{

auto pLambda = GetPlambda();

auto derivative = TakeDerivative(pLambda);

double x1,x2,x3,x4;

auto it1 = DichotomyMethod(-68, -56, x1, pLambda);

auto it2 = DichotomyMethod(30, 40, x2, pLambda);

auto it3 = NewtonMethod(-70, -60, x3, pLambda, derivative);

auto it4 = NewtonMethod(30, 40, x4, pLambda, derivative);

std::cout << "Dichotomy it = " << it1 << "; x1(infinity) = " << x1 << std::endl;

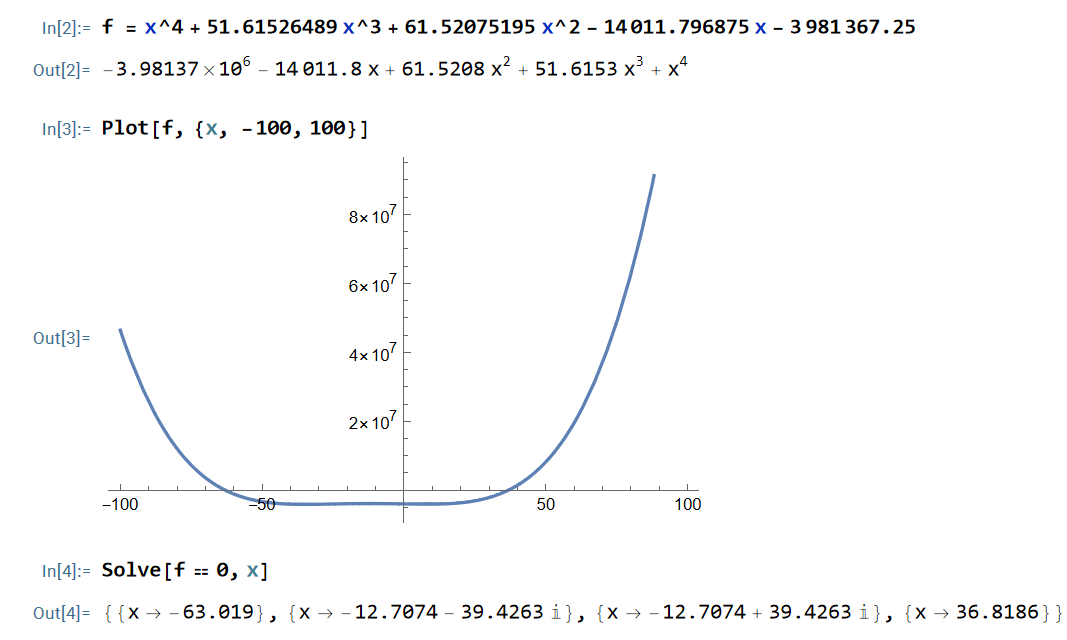
std::cout << "Dichotomy it = " << it2 << "; x2(infinity) = " << x2 << std::endl;

std::cout << "Newton it = " << it3 << "; x1(infinity) = " << x3 << std::endl;

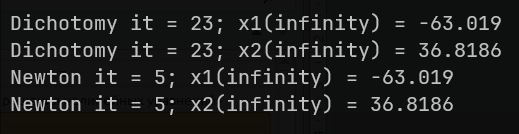
std::cout << "Newton it = " << it4 << "; x2(infinity) = " << x4 << std::endl;

}

Результаты:



2 вещественных корня соответственно выделим два интервала {-70,-60} {30,40}.



Можем заметить, что скорость сходимости метода Ньютона значительно выше, чем сходимость метода Дихотомии.