**Міністерство освіти, науки, молоді та спорту України**

**Національний технічний університет України**

**«Київський політехнічний інститут»**

Факультет прикладної математики

Кафедра спеціалізованих комп’ютерних систем та системного програмування

**Лабораторна робота № 5**

З Алгоритмів і методів обчислень

**«СЕРЕДНЬОКВАДРАТИЧНЕ НАБЛИЖЕННЯ ФУНКЦІЙ»**

Виконав:

студент групи КВ-02

Стасюк А. В.

Перевірив:

« » \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2012 р.

\_\_\_\_\_\_\_\_ Зорін Ю. М.

\_\_\_\_ балів

І семестр

К И Ї В – 2 0 1 2

***Завдання***

1. Написати програму побудови узагальненого многочлена для заданої функції, яку необхідно апроксимувати
2. За допомогою програми визначити степінь узагальненого многочлена, який забезпечує для заданої функції на заданому проміжку середньоквадратичне відхилення не гірше ніж *O(10-2).*
3. За допомогою *Advanced Grapher* побудувати графік заданої функції та графік узагальненого многочлена степеня, визначеного в п.2.

***Тексти файлів***

// AbstractFunction.h

#ifndef ABSTRACT\_FUNCTION

#define ABSTRACT\_FUNCTION

class AbstractFunction {

public: virtual double initial(double x) = 0;

};

class FunctionsMultiplicationTerm: public AbstractFunction {

private: AbstractFunction \* firstFunction;

private: AbstractFunction \* secondFunction;

private: bool isDivision;

public: FunctionsMultiplicationTerm(AbstractFunction \* firstFunction, AbstractFunction \* secondFunction, bool isDivision = false) {

this->firstFunction = firstFunction;

this->secondFunction = secondFunction;

this->isDivision = isDivision;

}

public: double initial(double x) {

double multipilier = isDivision ? (1.0 / secondFunction->initial(x)) : secondFunction->initial(x);

return firstFunction->initial(x) \* multipilier;

}

};

class FunctionsSumTerm: public AbstractFunction {

private: AbstractFunction \* firstFunction;

private: AbstractFunction \* secondFunction;

private: bool isSubstraction;

public: FunctionsSumTerm(AbstractFunction \* firstFunction, AbstractFunction \* secondFunction, bool isSubstraction = false) {

this->firstFunction = firstFunction;

this->secondFunction = secondFunction;

this->isSubstraction = isSubstraction;

}

public: double initial(double x) {

double koeficient = isSubstraction ? -1.0 : 1.0;

return firstFunction->initial(x) + koeficient\*secondFunction->initial(x);

}

};

class FunctionPower: public AbstractFunction {

private: AbstractFunction \* function;

private: double power;

public: FunctionPower(AbstractFunction \* function, double power) {

this->function = function;

this->power = power;

}

public: double initial(double x) {

if((double) (int) power == power) {

int integerPower = (int) power;

return pow(function->initial(x), integerPower);

}

return pow(function->initial(x), power);

}

};

#endif

//AbstractFunctionalBasis.h

#ifndef ABSTRACT\_FUNCTIONAL\_BASIS

#define ABSTRACT\_FUNCTIONAL\_BASIS

#include "../integrating/AbstarctFunction.h"

class AbstractFunctionalBasis {

public: virtual AbstractFunction \* nthFunction(int functionOrder) = 0;

};

#endif

// AbstractIntegrator.h

#ifndef ABSTRACT\_INTEGRATOR

#define ABSTRACT\_INTEGRATOR

#define \_in

#define \_out

#include "AbstarctFunction.h"

class AbstractIntegrator {

public: virtual double integrate(\_in double left, \_in double right, \_in \_out int & ranges) = 0;

public: virtual AbstractFunction \* getFunction() = 0;

};

#endif

// AbstractEquationSystemSolver.h

#ifndef ABSTRACT\_EQUATION\_SYSTEM\_SOLVER

#define ABSTRACT\_EQUATION\_SYSTEM\_SOLVER

#include <vector>

#include "LinearEquationSystem.h"

using namespace std;

class AbstractEquationSystemSolver {

public: AbstractEquationSystemSolver() {}

public: virtual vector <double> solve(double precision, bool & wasSolved, string & message) = 0;

};

#endif

// DoubleRecalculationMethod.h

#ifndef DOUBLE\_RECALCULATION\_METHOD

#define DOUBLE\_RECALCULATION\_METHOD

#include <math.h>

#include <iostream>

#include "AbstractIntegrator.h"

using namespace std;

class DoubleRecalculationMethod {

private: double r;

private: AbstractIntegrator \* integrator;

public: DoubleRecalculationMethod(AbstractIntegrator \* integrator, double r) {

this->integrator = integrator;

this->r = r;

}

public: double integrate(\_in double left, \_in double right, \_in double precision, \_in int maximumRanges = 10000) {

const int mantissaLength = 52;

swapIfNeeded(left, right);

int rangesCount = (int)(1.0 / pow(precision, 1 / r)) + 1;

double ideal = 0.0;

double previousResult = 0.0, currentResult = 0.0;

double relativeError = 0.0;

do {

previousResult = currentResult;

currentResult = integrator->integrate(left, right, rangesCount);

double resultsDelta = currentResult - previousResult;

resultsDelta = resultsDelta < 0.0 ? -resultsDelta : resultsDelta;

relativeError = resultsDelta / (currentResult < 0.0 ? -currentResult : currentResult);

rangesCount <<= 1;

} while(relativeError > precision && (rangesCount >> 1) < maximumRanges);

return currentResult;

}

private: void swapIfNeeded(\_in \_out double & left, \_in \_out double & right) {

if(left > right) {

double temporary = left;

left = right;

right = temporary;

}

}

};

#endif

// TrapezeMethodIntegrator.h

#ifndef TRAPEZE\_METHOD\_INTEGRATOR

#define TRAPEZE\_METHOD\_INTEGRATOR

#include <vector>

#include "AbstractIntegrator.h"

#include "AbstarctFunction.h"

using namespace std;

class TrapezeMethodIntegrator: public AbstractIntegrator {

private: AbstractFunction \* function;

public: TrapezeMethodIntegrator(AbstractFunction \* function) {

this->function = function;

}

public: double integrate(\_in double left, \_in double right, \_in \_out int & ranges) {

swapIfNeeded(left, right);

vector <double> nodes(ranges + 1);

double step = (right - left) / ranges;

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

nodes[i] = left + i\*step;

double result = 0.0;

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

result += function->initial(nodes[i]) / ((i == 0 || i == ranges) ? 2.0 : 1.0);

result \*= (right - left) / ranges;

return result;

}

public: AbstractFunction \* getFunction() {

return function;

}

private: void swapIfNeeded(\_in \_out double & left, \_in \_out double & right) {

if(left > right) {

double temporary = left;

left = right;

right = temporary;

}

}

};

#endif

// LegendreFunctionalBasis.h

#ifndef LEGENDRE\_FUNCTIONAL\_BASIS

#define LEGENDRE\_FUNCTIONAL\_BASIS

#include "AbstractFunctionalBasis.h"

#pragma warning (disable: 4244)

#include <vector>

using namespace std;

class LegenderFunction: public AbstractFunction {

private: int order;

private: double \* resultsBuffer;

public: LegenderFunction(int order) {

this->order = order;

resultsBuffer = new double[order + 1];

}

public: ~LegenderFunction() {

delete [] resultsBuffer;

}

public: double initial(double x) {

if(order == 0)

return 1.0;

else if(order == 1)

return x;

else {

resultsBuffer[0] = 1.0;

resultsBuffer[1] = x;

for(int i = 2; i <= order; i++) {

double n = i - 1;

resultsBuffer[i] = ((2\*n + 1) / (n + 1) \* x \* resultsBuffer[i - 1]) -

(n / (n + 1))\*resultsBuffer[i - 2];

}

return resultsBuffer[order];

}

}

private: double calculate(double x, int currentOrder) {

if(currentOrder == 0)

return 1.0;

else if(currentOrder == 1)

return x;

else {

double n = (double)currentOrder - 1.0;

return ((2\*n + 1) / (n + 1)\*x\*calculate(n, x)) -

(n / (n + 1) \* calculate(n - 1, x));

}

}

};

class LegendreFunctionalBasis: public AbstractFunctionalBasis {

private: vector <AbstractFunction \*> functions;

public: LegendreFunctionalBasis(int maximumOrder = 32) {

functions.resize(maximumOrder, 0);

}

public: ~LegendreFunctionalBasis() {

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

delete functions[i];

}

public: AbstractFunction \* nthFunction(int functionOrder) {

if(functionOrder > functions.size() + 1)

functions.resize(functionOrder + 1);

AbstractFunction \* function = functions[functionOrder];

if(function == 0) {

function = new LegenderFunction(functionOrder);

functions[functionOrder] = function;

}

return function;

}

};

#endif

// TestFunctionalBasis.h

#ifndef TEST\_FUNCTIONAL\_BASIS

#define TEST\_FUNCTIONAL\_BASIS

#include "AbstractFunctionalBasis.h"

#pragma warning (disable: 4244)

class TestBasisFunction: public AbstractFunction {

private: int order;

public: TestBasisFunction(int order) {

this->order = order;

}

public: double initial(double x) {

return pow(x, order);

}

};

class TestFunctionalBasis: public AbstractFunctionalBasis {

private: vector <AbstractFunction \*> functions;

public: TestFunctionalBasis(int maximumOrder = 32) {

functions.resize(maximumOrder, 0);

}

public: ~TestFunctionalBasis() {

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

delete functions[i];

}

public: AbstractFunction \* nthFunction(int functionOrder) {

if(functionOrder > functions.size() + 1)

functions.resize(functionOrder + 1);

AbstractFunction \* function = functions[functionOrder];

if(function == 0) {

function = new TestBasisFunction(functionOrder);

functions[functionOrder] = function;

}

return function;

}

};

#endif

// MyFunction.h

#ifndef MY\_FUNCTION

#define MY\_FUNCTION

#include <math.h>

#include "AbstarctFunction.h"

class MyFunction: public AbstractFunction {

public: double initial(double x) {

return 0.2 \* exp(pow(x, 1.0 / 3.0)) \* log(x) \* sin(3.0\*x);

}

};

#endif

// TestFunction.h

#ifndef TEST\_FUNCTION

#define TEST\_FUNCTION

#include <math.h>

#include "AbstarctFunction.h"

class TestFunction: public AbstractFunction {

public: double initial(double x) {

return -2\*x\*x\*x\*x\*x +

3\*x\*x\*x\*x -

12\*x\*x\*x +

20\*x\*x -

14.3\*x +

15;

}

};

#endif

// LeastSquaresInterpolator.h

#ifndef LEAST\_SQUARES\_INTERPOLATOR

#define LEAST\_SQUARES\_INTERPOLATOR

#include "../integrating/TrapezeMethodIntegrator.h"

#include "../integrating/DoubleRecalculationMethod.h"

#include "../equations\_system/SingleDivisionSchemSolver.h"

#include "../equations\_system/LinearEquationSystem.h"

#include "../integrating/AbstarctFunction.h"

#include "AbstractFunctionalBasis.h"

#include <vector>

#include <iostream>

#include <string>

using namespace std;

class InterpolatedFunction: public AbstractFunction {

public: AbstractFunctionalBasis \* basis;

public: vector <double> koeficients;

public: InterpolatedFunction(AbstractFunctionalBasis \* basis, vector <double> & koeficients) {

this->basis = basis;

this->koeficients = koeficients;

}

public: double initial(double x) {

int order = koeficients.size() - 1;

double result = 0.0;

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

result += koeficients[i] \* basis->nthFunction(i)->initial(x);

return result;

}

};

class LeastSquaresInterpolator {

private: AbstractFunction \* function;

private: AbstractFunctionalBasis \* basis;

private: double left;

private: double right;

private: vector <LinearEquationSystem \*> previousSystemsPool;

public: LeastSquaresInterpolator(AbstractFunctionalBasis \* basis, AbstractFunction \* function, double left, double right,

int maximumOrder = 32) {

this->function = function;

this->basis = basis;

this->left = left;

this->right = right;

previousSystemsPool.resize(maximumOrder + 1, 0);

}

public: vector <double> interpolate(int order, \_out double & deviation, \_out AbstractFunction \*\* resultFunction) {

LinearEquationSystem normalSystem (order + 1);

fillEquationsSystem(order, normalSystem);

vector <double> resultKoeficients = solveEquationSystem(normalSystem);

saveEquationsSystemToPool(normalSystem);

(\* resultFunction) = new InterpolatedFunction(basis, resultKoeficients);

AbstractFunction \* deviationFunction = new FunctionPower(new FunctionsSumTerm(function, (\* resultFunction), true), 2);

deviation = integrateFunction(deviationFunction);

return resultKoeficients;

}

private: void saveEquationsSystemToPool(LinearEquationSystem & equationsSystem) {

if(previousSystemsPool.size() < equationsSystem.getEquationsCount() + 1)

previousSystemsPool.resize(equationsSystem.getEquationsCount() + 1);

previousSystemsPool[equationsSystem.getEquationsCount() - 1] = new LinearEquationSystem(equationsSystem);

}

private: vector <double> solveEquationSystem(LinearEquationSystem & normalSystem) {

bool wasSolved = false;

string message = "";

SingleDivisionSchemSolver systemSolver(& normalSystem);

return systemSolver.solve(0, wasSolved, message);

}

private: void fillEquationsSystem(int order, LinearEquationSystem & equationSystem) {

int filledRows = tryToFillParticulary(order, equationSystem);

for(int i = filledRows; i <= order; i++)

for(int j = 0; j <= order; j++) {

FunctionsMultiplicationTerm currentFunction(basis->nthFunction(i), basis->nthFunction(j));

equationSystem[i][j] = equationSystem[j][i] = integrateFunction(& currentFunction);

}

for(int i = filledRows; i <= order; i++) {

FunctionsMultiplicationTerm currentFunction(basis->nthFunction(i), this->function);

equationSystem[i][order + 1] = integrateFunction(& currentFunction);

}

}

private: double integrateFunction(AbstractFunction \* function) {

TrapezeMethodIntegrator trapezeMethodIntegrator(function);

int ranges = 10000;

return trapezeMethodIntegrator.integrate(left, right, ranges);

}

private: int tryToFillParticulary(int neededOrder, \_out LinearEquationSystem & equationsSystem) {

int theBestCandidateIndex = -1;

int currentPoolSize = previousSystemsPool.size();

for(int i = min(neededOrder, currentPoolSize); i > 0; i--)

if(previousSystemsPool[i] != 0) {

theBestCandidateIndex = i;

break;

}

if(theBestCandidateIndex < 1)

return 0;

LinearEquationSystem & theBestCandidate = \* previousSystemsPool[theBestCandidateIndex];

int equationsCount = theBestCandidate.getEquationsCount();

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

for(int j = i; j < equationsCount; j++)

equationsSystem[i][j] = equationsSystem[j][i] = theBestCandidate[i][j];

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

equationsSystem[i][neededOrder + 1] = theBestCandidate[i][equationsCount];

return equationsCount;

}

};

#endif

//main.cpp

#include "interpolating/LeastSquaresInterpolator.h"

#include "interpolating/LegendreFunctionalBasis.h"

#include "interpolating/TestFunctionalBasis.h"

#include "integrating/MyFunction.h"

#include "integrating/TestFunction.h"

#include "printing/TablePrinter.h"

#include <conio.h>

int main(int argumentsCount, char \* arguments []) {

setlocale(0, "");

const double left = 2.0;

const double right = 9.0;

LegendreFunctionalBasis myBasis;

MyFunction myFunction;

TestFunction testFunction;

TestFunctionalBasis testBasis;

LeastSquaresInterpolator interpolator(& myBasis, & myFunction, left, right);

//LeastSquaresInterpolator interpolator(& testBasis, & testFunction, 2.0, 9.0);

AbstractFunction \* resultFunction = 0, \* bestFunction = 0;

double deviation = 0.0, bestDeviation = 0.0;

char \* deviationHeaders [] = {"Порядок полiнома", "Сер. кв. вiдхилення"};

TablePrinter deviationPrinter(2, deviationHeaders);

int currentOrder = 1;

do {

cerr << "Поточний порядок: " << currentOrder << "..." << endl;

interpolator.interpolate(currentOrder, deviation, & resultFunction);

deviationPrinter.setCellData(currentOrder, 0, currentOrder, TablePrinter :: CENTER);

deviationPrinter.setCellData(currentOrder, 1, deviation, TablePrinter :: RIGHT);

if(bestFunction == 0 || (bestDeviation != 0.0 && bestDeviation > deviation)) {

bestFunction = resultFunction;

bestDeviation = deviation;

}

currentOrder++;

} while(deviation >= 1e-1 && currentOrder < 32);

cerr << endl;

deviationPrinter.print("Залежнiсть сер. кв. вiдхилення вiд порядку полiнома");

char \* valuesHeaders [] = {"x", "Pn(x)"};

TablePrinter valuesPrinter(2, valuesHeaders);

const int nodes = 15;

double x = left;

for(int i = 0; i <= nodes; i++) {

valuesPrinter.setCellData(i + 1, 0, x, TablePrinter :: RIGHT);

valuesPrinter.setCellData(i + 1, 1, bestFunction->initial(x), TablePrinter :: RIGHT);

x += (right - left) / nodes;

}

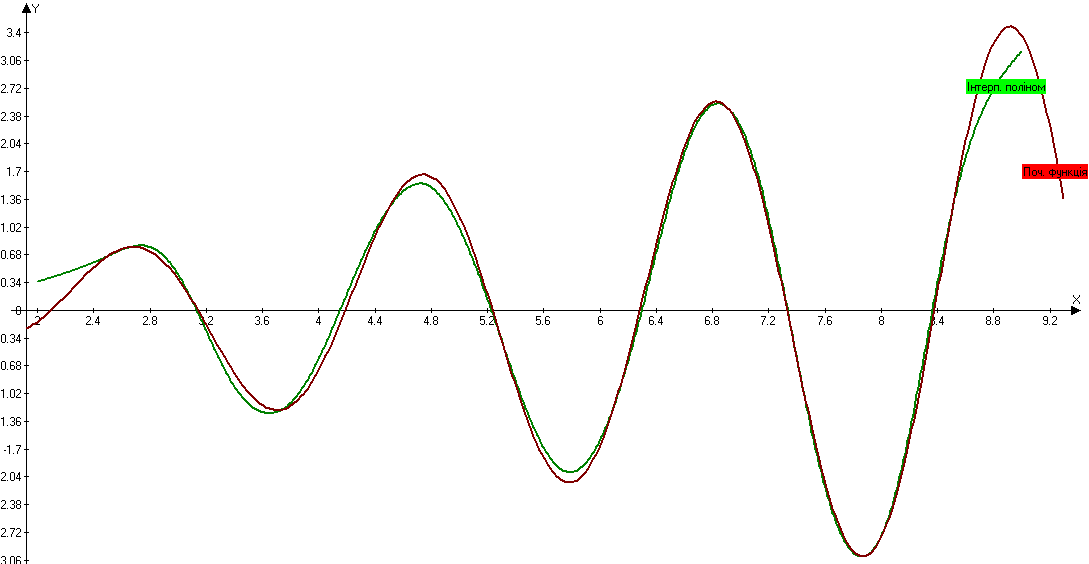
valuesPrinter.print("Таблиця значень полiнома");

getch();

return 0;

}

***Графіки початкової функції та інтерполяційного полінома***



***Результат роботи програми***

