**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Информационные технологии   
и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа № 3   
по курсу «Численные методы»

Группа: М8О-308Б-21

Студент: Белоносов К. А.

Преподаватель: Ревизников Д. Л.

Дата: 22.05.2024

Москва, 2024

**ОГЛАВЛЕНИЕ**

[1 Тема 3](#_Toc158983147)

[2 Задание 3](#_Toc158983148)

[3 Теория 4](#_Toc158983149)

[4 Ход лабораторной работы 5](#_Toc158983150)

[5 Выводы 6](#_Toc158983151)

# **Тема**

Методы приближения функций. Численное дифференцирование и интегрирование

# **Задание**

3.1. Используя таблицу значений функции , вычисленных в точках построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки . Вычислить значение погрешности интерполяции в точке .

3.2. Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при и . Вычислить значение функции в точке .

1.5

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | 0.0 | 0.9 | 1.8 | 2.7 | 3.6 |
|  | 0.0 | 0.36892 | 0.85408 | 1.7856 | 6.3138 |

3.3. Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены a) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
|  | -0.9 | 0.0 | 0.9 | 1.8 | 2.7 | 3.6 |
|  | -0.36892 | 0.0 | 0.36892 | 0.85408 | 1.7856 | 6.3138 |

3.4. Вычислить первую и вторую производную от таблично заданной функции в точке .

 2.0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
|  | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 |
|  | 0.0 | 0.40547 | 0.69315 | 0.91629 | 1.0986 |

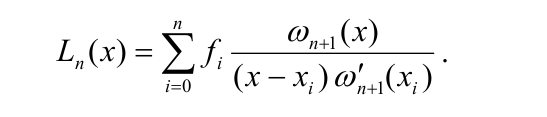
3.5. Вычислить определенный интеграл , методами прямоугольников, трапеций, Симпсона с шагами . Оценить погрешность вычислений, используя Ме­тод Рунге-Ромберга.



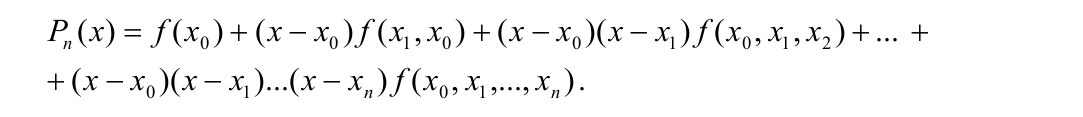


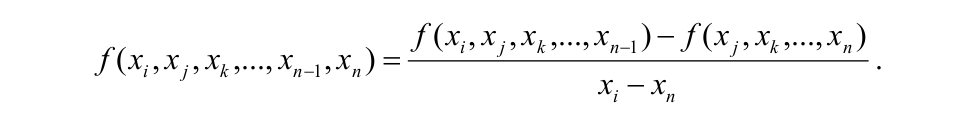
# **Теория**

Рассмотрим задачу приближения функции с помощью **интерполяционного многочлена Лагранжа:**

****

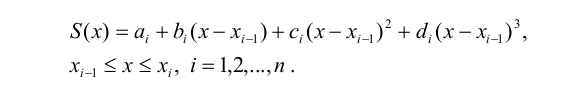
Недостатком интерполяционного многочлена Лагранжа является необходимость полного пересчета всех коэффициентов в случае добавления дополнительных интерполяционных узлов. Чтобы избежать указанного недостатка используют **интерполяционный многочлен в форме Ньютона**:

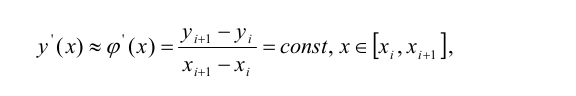
Разделенная разность порядка n − k + 2 определяется соотношениями:



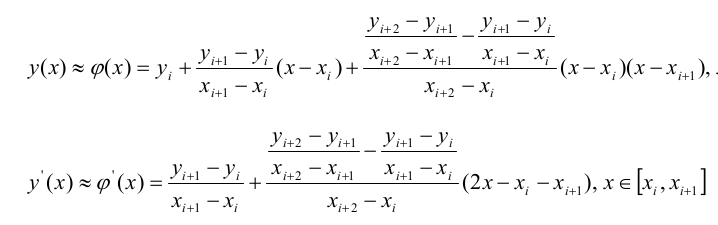
Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается в применении **сплайн-интерполяции.** Суть сплайн-интерполяции заключается в определении интерполирующей функции по формулам одного типа для различных непересекающихся промежутков и в стыковке значений функции и её производных на их границах.

На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как:

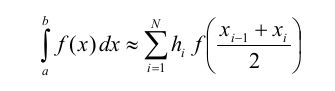


В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой:

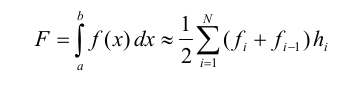
При использовании для аппроксимации таблично заданной функции интерполяционного многочлена второй степени имеем:

 При использовании интерполяционных многочленов различной степени, получают формулы численного интегрирования различного порядка точности.

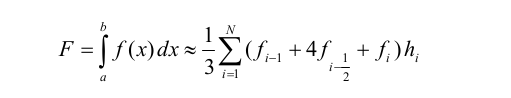
Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку xi, получим **формулу прямоугольников:**



В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т.е. заменить функцию f (x) многочленом Лагранжа первой степени, получим **формулу трапеции:**



Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой – интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования. Получим **формулу Симпсона (парабол):**



# **Х****од лабораторной работы**

Код был реализован на языке C++

#ifndef POLYNOMS\_H

#define POLYNOMS\_H

#include <vector>

#include <algorithm>

#include "../lab1/Matrix.h"

#include <matplotlibcpp.h>

namespace plt = matplotlibcpp;

using namespace numeric;

double lagrangePolynomial(const std::vector<double>& X, const std::vector<double>& Y, double x) {

size\_t n = X.size();

double result = 0.0;

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

double term = Y[i];

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

if (i != j) {

term \*= (x - X[j]) / (X[i] - X[j]);

}

}

result += term;

}

return result;

}

class NewtonPolynomial {

private:

std::vector<double> X;

std::vector<double> coefficients;

public:

NewtonPolynomial(const std::vector<double>& X, const std::vector<double>& Y) {

this->X = X;

computeDividedDifferences(Y);

}

void computeDividedDifferences(const std::vector<double>& Y) {

coefficients = Y;

int n = X.size();

for (int j = 1; j < n; j++) {

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

coefficients[i] = (coefficients[i] - coefficients[i - 1]) / (X[i] - X[i - j]);

}

}

}

double evaluate(double x) {

int n = X.size();

double result = coefficients[n - 1];

for (int i = n - 2; i >= 0; i--) {

result = result \* (x - X[i]) + coefficients[i];

}

return result;

}

};

class CubicSpline {

private:

struct SplineTuple {

double a, b, c, d, x;

};

std::vector<SplineTuple> splines;

public:

CubicSpline(const std::vector<double>& x, const std::vector<double>& y) : splines(x.size() - 1){

int n = x.size() - 1;

std::vector<double> h(n);

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

h[i] = x[i + 1] - x[i];

}

numeric::Matrix<double> matrix(n, n);

std::vector<double> b(n, 0);

for (int i = 1; i <= n - 1; ++i) {

if(i == 1) {

matrix[i][i] = 2 \* (h[i - 1] + h[i]);

matrix[i][i + 1] = h[i];

} else if(i == n - 1) {

matrix[i][i - 1] = h[i - 1];

matrix[i][i] = 2 \* (h[i - 1] + h[i]);

} else {

matrix[i][i - 1] = h[i - 1];

matrix[i][i] = 2 \* (h[i - 1] + h[i]);

matrix[i][i + 1] = h[i];

}

b[i] = 3 \* ((y[i + 1] - y[i]) / h[i] - (y[i] - y[i - 1]) / h[i - 1]);

}

matrix[0][0] = 1;

std::vector<double> c = numeric::tridiagonalSolve(matrix, b);

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

splines[i].a = y[i];

if(i == n - 1) {

splines[i].b = (y[n] - y[n - 1]) / h[i] - 2.0 / 3 \* h[i] \* c[i];

splines[i].d = - c[i] / (3 \* h[i]);

} else {

splines[i].b = (y[i + 1] - y[i]) / h[i] - h[i] \* (c[i + 1] + 2 \* c[i]) / 3;

splines[i].d = (c[i + 1] - c[i]) / (3 \* h[i]);

}

splines[i].c = c[i];

splines[i].x = x[i];

}

}

double evaluate(double x) const{

if (splines.empty()) {

throw std::invalid\_argument("Splines not constructed yet");

}

const auto segment = std::lower\_bound(splines.begin(), splines.end(), x, [](const SplineTuple &s, double x) {

return s.x < x;

}) - 1;

double dx = x - segment->x;

return segment->a + segment->b \* dx + segment->c \* dx \* dx + segment->d \* dx \* dx \* dx;

}

};

void plotCubicSpline(const CubicSpline& sp, const std::vector<double>& x\_points, const std::vector<double>& y\_points) {

std::vector<double> x\_vals, y\_vals;

double x\_min = \*min\_element(x\_points.begin(), x\_points.end());

double x\_max = \*max\_element(x\_points.begin(), x\_points.end());

double step = (x\_max - x\_min) / 1000; // Подробность графика

for (double x = x\_min; x <= x\_max; x += step) {

double y = sp.evaluate(x);

x\_vals.push\_back(x);

y\_vals.push\_back(y);

}

plt::scatter(x\_points, y\_points, 40);

plt::plot(x\_vals, y\_vals); // Рисуем сплайн

plt::title("Cubic Spline Interpolation");

plt::xlabel("X");

plt::ylabel("Y");

plt::grid(true);

plt::show();

}

template<typename T>

Matrix<T> vandermonde(const std::vector<T>& x, int degree) {

size\_t n = x.size();

Matrix<T> V(n, degree + 1);

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

T val = 1;

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

V[i][j] = val;

val \*= x[i];

}

}

return V;

}

template<typename T>

Matrix<T> leastSquares(const std::vector<T>& x, const std::vector<T>& y, int degree) {

Matrix<T> V = vandermonde(x, degree);

auto Vt = V.transpose();

Matrix<T> VtV = Vt \* V;

std::vector<T> VtY = Vt \* y;

// Создание аугментированной матрицы VtV | VtY

Matrix<T> augmented(VtV.rows(), VtV.cols() + 1);

for (size\_t i = 0; i < VtV.rows(); ++i) {

for (size\_t j = 0; j < VtV.cols(); ++j) {

augmented[i][j] = VtV[i][j];

}

augmented[i][VtV.cols()] = VtY[i];

}

LUMatrix<T> luSolver(augmented);

Matrix<T> ans = luSolver.solve();

return ans;

}

double derivativeUsingQuadraticInterpolation(const std::vector<double>& X, const std::vector<double>& Y, double x) {

int n = X.size();

if (n < 3) {

throw std::invalid\_argument("Need at least three points for quadratic interpolation.");

}

int i = 1;

for (; i < n - 1; i++) {

if (X[i] <= x && X[i+1] > x) {

break;

}

}

if (i == n - 1) {

throw std::invalid\_argument("x is out of bounds of the provided data points.");

}

double dy = (Y[i+1] - Y[i-1]) / (X[i+1] - X[i-1]) +

(2 \* x - X[i] - X[i+1]) \* (

(Y[i+1] - Y[i]) / (X[i+1] - X[i]) -

(Y[i] - Y[i-1]) / (X[i] - X[i-1])

) / (X[i+1] - X[i-1]);

return dy;

}

double linearDerivative(const std::vector<double>& X, const std::vector<double>& Y, double x) {

int n = X.size();

if (n < 2) {

throw std::invalid\_argument("Need at least two points for linear interpolation.");

}

int i = 0;

for (; i < n - 1; i++) {

if (x >= X[i] && x <= X[i+1]) {

break;

}

}

if (i == n - 1) {

throw std::invalid\_argument("x is out of the bounds of the provided data points.");

}

double dy = (Y[i+1] - Y[i]) / (X[i+1] - X[i]);

return dy;

}

double secondDerivativeAtX(const std::vector<double>& X, const std::vector<double>& Y, double x) {

int n = X.size();

if (n < 3) {

throw std::invalid\_argument("Need at least three points for quadratic interpolation.");

}

// Найти подходящий интервал i так, что x\_i <= x < x\_{i+1}

int i = 1;

for (; i < n - 1; i++) {

if (X[i] <= x && x < X[i + 1]) {

break;

}

}

if (i == 1 || i == n - 1) {

throw std::invalid\_argument("x is out of the interpolation bounds. Need more internal data points.");

}

double h1 = X[i] - X[i-1];

double h2 = X[i+1] - X[i];

double delta1 = (Y[i] - Y[i-1]) / h1;

double delta2 = (Y[i+1] - Y[i]) / h2;

double second\_derivative = 2 \* (delta2 - delta1) / (h2 + h1);

return second\_derivative;

}

double rectangleMethod(std::function<double(double)> f, double a, double b, int n) {

double h = (b - a) / n;

double sum = 0.0;

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

sum += f(a + i \* h);

}

return sum \* h;

}

double trapezoidMethod(std::function<double(double)> f, double a, double b, int n) {

double h = (b - a) / n;

double sum = 0.5 \* (f(a) + f(b));

for (int i = 1; i < n; i++) {

sum += f(a + i \* h);

}

return sum \* h;

}

double simpsonMethod(std::function<double(double)> f, double a, double b, int n) {

double h = (b - a) / n;

double sum = f(a) + f(b);

for (int i = 1; i < n; i++) {

sum += (i % 2 == 0 ? 2 : 4) \* f(a + i \* h);

}

return sum \* h / 3;

}

double rungeRomberg(double I1, double I2, int k) {

return (I1 - I2) / (pow(2, k) - 1);

}

#endif //POLYNOMS\_H

3.1) Интерполяционные многочлены Лагранжа и Ньютона:

#include <iostream>

#include <cmath>

#include <vector>

#define WITHOUT\_NUMPY

#include "Polynoms.h"

using namespace std;

double f(double x) {

return tan(x);

}

int main() {

vector<double> X1 = {0, M\_PI/8, 2\*M\_PI/8, 3\*M\_PI/8};

vector<double> Y1 = {0, 0.4142, 1, 2.4142};

vector<double> X2 = {0, M\_PI/8, M\_PI/3, 3\*M\_PI/8};

vector<double> Y2 = {0, 0.4142, 1.7320, 2.4142};

double x = 3 \* M\_PI / 16;

cout << "Lagrange first: ";

cout << lagrangePolynomial(X1, Y1, x);

cout << "\nNewton first: ";

cout << NewtonPolynomial(X1, Y1).evaluate(x);

cout << "\nLagrange second: ";

cout << lagrangePolynomial(X2, Y2, x);

cout << "\nNewton second: ";

cout << NewtonPolynomial(X2, Y2).evaluate(x);

cout << "\nReal value: ";

cout << f(x) << "\n";

return 0;

}

3.2) Кубический сплайн:

#include <iostream>

#include <cmath>

#include <vector>

#define WITHOUT\_NUMPY

#include "Polynoms.h"

using namespace std;

int main() {

CubicSpline spline({0.0, 0.9, 1.8, 2.7, 3.6}, {0.0, 0.36892, 0.85408, 1.7856, 6.3138});

cout << "\nSpline: ";

cout << spline.evaluate(1.5);

plotCubicSpline(spline, {0.0, 0.9, 1.8, 2.7, 3.6}, {0.0, 0.36892, 0.85408, 1.7856, 6.3138});

}

3.3) МНК:

#include <iostream>

#include <vector>

#define WITHOUT\_NUMPY

#include "matplotlibcpp.h"

#include "Polynoms.h"

using namespace std;

namespace plt = matplotlibcpp;

int main() {

vector<double> X = {0, 1.7, 3.4, 5.1, 6.8, 8.5};

vector<double> Y = {0.0, 1.3038, 1.8439, 2.2583, 2.6077, 2.9155};

auto coefficients1 = leastSquares(X, Y, 1);

auto coefficients2 = leastSquares(X, Y, 2);

vector<double> Y\_fit1, Y\_fit2, X\_cont;

for (double x = \*min\_element(X.begin(), X.end()); x <= \*max\_element(X.begin(), X.end()); x += 0.1) {

X\_cont.push\_back(x);

Y\_fit1.push\_back(coefficients1[0][0] + coefficients1[1][0] \* x);

Y\_fit2.push\_back(coefficients2[0][0] + coefficients2[1][0] \* x + coefficients2[2][0] \* x \* x);

}

plt::figure\_size(1200, 780);

plt::scatter(X, Y, 50);

plt::plot(X\_cont, Y\_fit1, "--g");

plt::plot(X\_cont, Y\_fit2, "--r");

plt::title("Least Squares Polynomial Fit");

plt::xlabel("X");

plt::ylabel("Y");

std::map<std::string, std::string> legend\_keywords;

legend\_keywords["title"] = "Legend";

plt::legend(legend\_keywords);

plt::show();

return 0;

}

3.4) Первая и вторая производные:

#include <iostream>

#include <vector>

#define WITHOUT\_NUMPY

#include "Polynoms.h"

using namespace std;

namespace plt = matplotlibcpp;

int main() {

vector<double> X = {1, 1.5, 2.0, 2.5, 3.0};

vector<double> Y = {0.0, 0.40547, 0.69315, 0.91629, 1.0986};

cout << linearDerivative(X, Y, 2.0) << "\n";

cout << derivativeUsingQuadraticInterpolation(X, Y, 2.0) << "\n";

cout << secondDerivativeAtX(X, Y, 2.0) << "\n";

return 0;

}

3.5) Метод прямоугольника, трапеций, Симпсона:

#include <iostream>

#include <vector>

#define WITHOUT\_NUMPY

#include "Polynoms.h"

using namespace std;

namespace plt = matplotlibcpp;

double f(double x) {

return x/pow(3\*x + 4, 3);

}

int main() {

double a = -1;

double b = 1;

int n = 10;

double I\_rect = rectangleMethod(f, a, b, n);

double I\_trap = trapezoidMethod(f, a, b, n);

double I\_simp = simpsonMethod(f, a, b, n);

double I\_rect\_double = rectangleMethod(f, a, b, 2 \* n);

double I\_trap\_double = trapezoidMethod(f, a, b, 2 \* n);

double I\_simp\_double = simpsonMethod(f, a, b, 2 \* n);

double err\_rect = rungeRomberg(I\_rect\_double, I\_rect, 1);

double err\_trap = rungeRomberg(I\_trap\_double, I\_trap, 1);

double err\_simp = rungeRomberg(I\_simp\_double, I\_simp, 4);

std::cout << "Integral using Rectangle Method: \n" << "\nFirst: "<< I\_rect << "\nSecond: " << I\_rect\_double << "\nError: " << err\_rect << "\n\n";

std::cout << "Integral using Trapezoid Method: \n" << "\nFirst: "<< I\_trap << "\nSecond: " << I\_trap\_double << "\nError: " << err\_trap << "\n\n";

std::cout << "Integral using Simpson Method: \n" << "\nFirst: "<< I\_simp << "\nSecond: " << I\_simp\_double << "\nError: " << err\_simp << "\n\n";

return 0;

}

# **Выводы**

В ходе лабораторной работы были рассмотрены методы приближения функций, численного дифференцирования и интегрирования. Построение интерполяционных многочленов Лагранжа и Ньютона для заданных точек продемонстрировало их точность и эффективность, что подтвердилось малой погрешностью в вычислении значений в конкретной точке.

Кубический сплайн с нулевой кривизной на концах интервала позволил создать гладкую функцию, проходящую через все заданные точки, и его значение в заданной точке оказалось близким к истинному.

Численное интегрирование методами прямоугольников, трапеций и Симпсона показало, что метод Симпсона дает наименьшую погрешность.