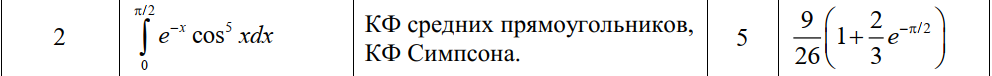
**Лабораторная работа №3**

**«**Приближенное вычисление интегралов**»**

Башев Ян 11 группа

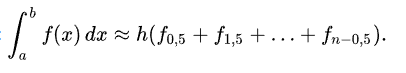
Вариант 2



Задание 1

Были использованы следующие КФ:

Формула Средних прямоугольников(СП):



Формула Симпсона:

, где

Правило Рунге:

Правило Рунге является критерием оценки точности вычислений определенного интеграла с использованием составных численных квадратурных формул (КФ). Согласно этому правилу, вычисления продолжаются с удвоением количества узлов до тех пор, пока не будет выполняться следующее условие:

, где – КФ представляет собой значение КФ, вычисленное с использованием вдвое большего количества узлов, чем изначально, а - значение КФ, вычисленное с использованием начального количества узлов., где p - порядок точности использованного численного метода. Для метода трапеций p = 2, для метода Симпсона p = 4. Для метода трапеций p равняется 2, а для метода Симпсона p равняется 4. означает величину погрешности, а ε - заданную точность вычислений.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Квадратурная формула | Число разбиений | Шаг | Приближенное значение интеграла | Оценка погрешности | Абсолютная погрешность |
| Средних прямоугольников | 2  4  8  16  32  64  128  256  512 | 0.785398  0.392699  0.19635  0.0981748  0.0490874  0.0245437  0.0122718  0.00613592  0.00306796 | 0.358944  0.387263  0.392494  0.393723  0.394026  0.394101  0.39412  0.394124  0.394126 | -1  0.00943963  0.00174351  0.000409653  0.000100895  2.51306e-05  6.27686e-06  1.56885e-06  3.9219e-07  9.80458e-08 | 9.80458e-08 |
| Симпсона | 2  4  8  16  32 | 0.785398  0.392699  0.19635  0.0981748  0.0490874 | 0.346203  0.391297  0.394  0.394119  0.394126 | -1  0.00300628  0.000180205  7.91655e-06  4.61578e-07 | 2.82601e-08 |

Выводы

Благодаря высшему порядку точности метод Симпсона проявил себя намного лучше, на 32 узлах точнее, чем метод средних прямоугольников на 512.

Задание 2

Для КФ НАСТ использовались следующие формулы:

Решена СЛАУ с целью нахождения :

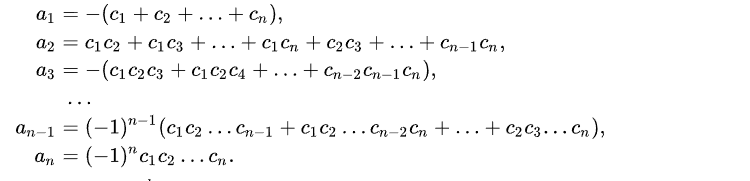
СЛАУ была построена следующим образом:

Получили коэффициенты для уравнения, которое решаем программно с помощью метода Ньютона (смотреть листинг – SolveLegendre()).

Решив получаем набор узлов

Находим по формуле :

Очевидно, что коэффициенты перед в произведении n-1 числа двучленов равны сумме попарных произведений с чередующимся знаком – Формула Виета:



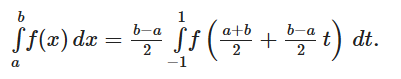
В данном случае брались только те коэффициенты, стоящие перед , где k – нечетное.

Таким образом, весовые коэффициенты были найдены в функции getWeights(см. листинг).

Чтобы получить квадратурную формулу Гаусса для произвольного отрезка [a, b], следует сделать замену переменной



в результате которой



nodes: 0.785398 1.20831 0.362485 1.49711 0.0736862

weights: 0.446804 0.375914 0.375914 0.186082 0.186082

correct integral value: 0.394126

NAST value: 0.394139

measurement: 1.2792e-05

Выводы

В рамках данного задания было вычислено приближенное значение интеграла из задания 1, используя квадратурную формулу наивысшей алгебраической степени точности (КФ НАСТ) с 4 узлами. КФ НАСТ даже небольшой степени позволяет относительно точно вычислять значения интегралов.

Исходный код

#define \_USE\_MATH\_DEFINES

#include <iostream>

#include <cmath>

#include <vector>

#include <Eigen/Dense>

#include <complex>

#include <algorithm>

const double a = 0;

const double b = M\_PI / 2;

const double EPS = 1e-7;

const int N = 5;

double I()

{

return 9. / 26 \* ( 1 + 2./3 \* pow(M\_E, -M\_PI / 2));

}

double f(double x)

{

return pow(M\_E, -x) \* pow(cos(x), 5);

}

double middleRect(int n, double h)

{

double res = 0;

for (int i = 0; i <= n - 1; i++)

{

double x = a + (double)i \* h + h / 2;

res += f(x);

}

return res \* h;

}

double simpson(int n, double h)

{

// n % 2 == 0 every time

double summOdd = 0, summEven = 0;

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

{

double x = a + (double)i \* h;

summOdd += (i % 2 == 1) \* f(x);

summEven += (i % 2 == 0) \* f(x);

}

return h / 3 \* (f(a) + f(b) + 4. \* summOdd + 2. \* summEven);

}

Eigen::VectorXd getLegendreKoeffs()

{

Eigen::MatrixXd A(N, N);

Eigen::VectorXd b(N);

for (int startC = 0; startC < N; startC++)

{

for (int i = N + startC; i >= startC; i--)

{

if (i != startC + N)

{

A(startC, N + startC - i - 1) = 1. \* (i % 2 == 0) \* (1. / (i + 1));

}

else

{

b(startC) = -1. \* (i % 2 == 0) \* (1. / (i + 1));

}

}

}

Eigen::VectorXd x = A.colPivHouseholderQr().solve(b);

return x;

}

std::vector<double> SolveLegendre()

{

Eigen::VectorXd coeffs = getLegendreKoeffs();

Eigen::MatrixXd companionMatrix(N, N);

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

{

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

{

companionMatrix(i, j) = 0;

}

}

for (int i = 1; i < N; ++i)

{

companionMatrix(i, i - 1) = 1;

}

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

{

companionMatrix(i, N - 1) = -coeffs[N - i - 1];

}

Eigen::EigenSolver<Eigen::MatrixXd> es(companionMatrix);

Eigen::VectorXcd roots = es.eigenvalues();

std::vector<double>res(N);

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

{

res[i] = roots[i].real();

}

return res;

}

std::vector<double> getWeights(std::vector<double>& nodes)

{

std::vector<double> weights(N);

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

{

std::vector<double> nodesW;

double del = 1;

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

{

if (j == i)continue;

nodesW.push\_back(nodes[j]);

del \*= (nodes[i] - nodes[j]);

}

for (int num0 = 0; num0 < N; num0+=2)

{

double res = 0;

std::vector<int>bitmask(nodesW.size(), 1);

for (int j = 0; j < num0; j++)

{

bitmask[j] = 0;

}

do

{

double umn = 1;

for (int j = 0; j < bitmask.size(); j++)

{

if (bitmask[j])

umn \*= nodesW[j];

}

res += umn;

} while (std::next\_permutation(bitmask.begin(), bitmask.end()));

weights[i] += (2. / (num0 + 1)) \* res;

}

weights[i] = weights[i] / del;

if (N % 2 == 0)weights[i] \*= -1;

}

return weights;

}

int main()

{

std::vector<std::pair<double(\*) (int, double), int >> kfs = { {middleRect, 2}, {simpson, 4} };

for (auto values : kfs)

{

int m = values.second;

std::cout << ((m == 2) ? "Middle rectangles method\n" : "Simpson's method\n");

double h = (b - a);

int n = 1;

double r = NULL;

do

{

n \*= 2;

h /= 2;

double qh = values.first(n, h);

if (r == NULL)

std::cout << n << ' ' << h << ' ' << qh << ' ' << " - " << '\n';

else

std::cout << n << ' ' << h << ' ' << qh << ' ' << r << '\n';

//std::cout << n << '\n';

double qh2 = values.first(n \* 2, h / 2);

r = abs((qh2 - qh) / (pow(2, m) - 1));

} while (abs(r) > EPS);

//std::cout << values.first(n \* 2, h / 2) << '\n';

std::cout << abs(I() - values.first(n \* 2, h / 2)) << '\n';

}

std::vector<double> nodes = SolveLegendre();

std::vector<double> weights = getWeights(nodes);

std::cout << "nodes: ";

for (auto& i : nodes)

{

i = i \* (b - a) / 2 + (a + b) / 2;

std::cout << i << ' ';

}

std::cout << '\n';

std::cout << "weights: ";

for (auto& i : weights)

{

i \*= (b - a) / 2;

std::cout << i << ' ';

}

std::cout << '\n';

double integral = 0;

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

{

integral += f(nodes[i]) \* weights[i];

}

std::cout << "correct integral value: " << I() << '\n';

std::cout << "NAST value: " << integral << '\n';

std::cout << "measurement: " << abs(I() - integral) << '\n';

}