Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»_____

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 5

Тема Построение и программная реализация алгоритмов численного интегрирования.

Студент Варламова Е.А.

Группа ИУ7-41Б

Оценка (баллы) _____

Преподаватель Градов В.М.

Москва.
2021 г

Цель работы. Получение навыков построения алгоритма вычисления двукратного интеграла с использованием квадратурных формул Гаусса и Симпсона.

1. Исходные данные

Функция, значение которой необходимо найти при фиксированном τ:

$$\varepsilon(\tau) = \frac{4}{\pi} \int_0^{\pi/2} d\varphi \int_0^{\pi/2} [1 - \exp(-\tau \frac{l}{R})] \cos\theta \sin\theta \, d\theta ,$$

где $$\frac{l}{R} = \frac{2\cos\theta}{1 - \sin^2\theta \cos^2\varphi},$$

θ, φ - углы сферических координат.

Применяется метод последовательного интегрирования. По одному направлению используется формула Гаусса, а по другому - формула Симпсона.

2. Код программы

```
from math import pi, cos, sin, exp
from numpy import arange
import matplotlib.pyplot as plt

# умножает полиномы, заданные в виде коэффициентов,
# т.е. x^2 - 1 === [-1, 0, 1]
def mul_polynomes(src, mult):
    dst = []
    for i in range(len(src) + len(mult) - 1):
        dst.append(0)
    for i in range (len(src)):
        for j in range (len(mult)):
            dst[i + j] += src[i] * mult[j]
    return dst

# вычисляет производную полинома
def derivative(pol):
    dst = []
    for i in range(1, len(pol)):
        dst.append(pol[i] * i)
    return dst

# находит полином Лежандра Pn по определению
def find_leg(n):
    st = 1
    fact = 1
    pol = [1]
    mult = [-1, 0, 1]

    for i in range (1, n + 1):
        st *= 2
        fact *= i
        pol = mul_polynomes(pol, mult)

    for i in range(n):
        pol = derivative(pol)

    for i in range(len(pol)):
        pol[i] *= 1 / (st * fact)
    return pol

# Вычисляет значение полинома
def get_pol_value(pol, arg):
    val = 0
    for i in range(len(pol)):
        val += pol[i] * (arg ** i)
    return val
```

```
# находит все отрезки, на которых есть корни, и вычисляет корни
# с помощью метода половинного деления
def find_roots(pol):
    n = len(pol) - 1
    k = 0
    if get_pol_value(pol, -1) * get_pol_value(pol, 0) > 0:
        k = 0
    else:
        k = 1
    segments = [[-1, 0]]
    t = n - n / 2
    while (k < t):
        seg_tmp = []
        k = 0
        for i in range(len(segments)):
            mid = (segments[i][1] + segments[i][0]) / 2
            seg_tmp.append( [segments[i][0], mid] )
            seg_tmp.append( [mid, segments[i][1]] )
            if get_pol_value(pol, mid) == 0:
                k += 1
            else:
                if get_pol_value(pol, segments[i][0]) *
                   get_pol_value(pol, mid) <= 0:
                    k += 1
                if get_pol_value(pol, segments[i][1]) *
                   get_pol_value(pol, mid) <= 0:
                    k += 1
        segments = seg_tmp
    roots = []
    for seg in segments:
        left = get_pol_value(pol, seg[0])
        right = get_pol_value(pol, seg[1])
        if left == 0:
            roots.append(seg[0])
        if right == 0:
            continue
        if get_pol_value(pol, seg[0]) < 0 and
           get_pol_value(pol, seg[1]) > 0:
            roots.append(half_division_method(pol,   seg[0],   seg[1],
True))
        if get_pol_value(pol, seg[0]) > 0 and
           get_pol_value(pol, seg[1]) < 0:
            roots.append(half_division_method(pol,   seg[0],   seg[1],
False))
    if get_pol_value(pol, segments[len(segments) - 1][1]) == 0:
        roots.append(segments[len(segments) - 1][1])
    t = int(n / 2)
    for i in range (t):
        roots.append(-roots[i])
    print(roots)
    return roots
```

```
# метод половинного деления, работает корректно только тогда,
# когда на концах отрезка значения функции разного знака
def half_division_method(pol, left, right, grad):
    mid = (left + right) / 2
    if abs(left - right) < 1e-5:
        return mid
    test = get_pol_value(pol, mid)
    if grad:
        if test > 0:
            return half_division_method(pol, left, mid, grad)
        elif test < 0:
            return half_division_method(pol, mid, right, grad)
    else:
        if test < 0:
            return half_division_method(pol, left, mid, grad)
        elif test > 0:
            return half_division_method(pol, mid, right, grad)
    return mid

# решение СЛАУ с помощью метода Гаусса
def solve_slau_Gauss(matrix, n):
    for k in range(n):
        for i in range(k + 1, n):
            coeff = -(matrix[i][k] / matrix[k][k])
            for j in range(k, n + 1):
                matrix[i][j] += coeff * matrix[k][j]

    a = [0 for i in range(n)]
    for i in range(n - 1, -1, -1):
        for j in range(n - 1, i, -1):
            matrix[i][n] -= a[j] * matrix[i][j]
        a[i] = matrix[i][n] / matrix[i][i]
    return a
# находит коэффициенты в основной системе (узлы найдены с помощью полиномов
Лежандра)
def find_coefs(nodes):
    matrix = []
    for i in range(len(nodes)):
        array = []
        for j in range(len(nodes)):
            array.append(nodes[j] ** i)
        if i % 2 == 0:
            array.append(2 / (i + 1))
        else:
            array.append(0)
        matrix.append(array)
    res = solve_slau_Gauss(matrix, len(nodes))
    return res
```

```
# функция из задания
def main_function(param):
    subfunc = lambda x, y: 2 * cos(x) / (1 - (sin(x) ** 2) * (cos(y) **
2))
    func = lambda x, y: (4 / pi) * (1 - exp(-param * subfunc(x, y))) *
cos(x) * sin(x)
    return func
# метод Симпсона
def simpson(func, a, b, num_of_nodes):
    if (num_of_nodes < 3 or num_of_nodes & 1 == 0):
        raise ValueError

    h = (b - a) / (num_of_nodes - 1)
    x = a
    res = 0

    for _ in range((num_of_nodes - 1) // 2):
        res += func(x) + 4 * func(x + h) + func(x + 2 * h)
        x += 2 * h

    return res * (h / 3)
def t_to_x(t, a, b):
    return (b + a) / 2 + (b - a) * t / 2

# интегрирование с использованием формул Гаусса
def gauss(func, a, b, num_of_nodes):
    leg = find_leg(num_of_nodes)
    args = find_roots(leg, -1, 1)
    coeffs = find_coefs(args)
    res = 0
    for i in range(num_of_nodes):
        res += (b - a) / 2 * coeffs[i] * func(t_to_x(args[i], a, b))
    return res
def func_2_to_1(func2, value):
    return lambda y: func2(value, y)
# вычисление двукратного интеграла
def integrate(func, limits, num_of_nodes, integrators):
    inner = lambda x: integrators[1](func_2_to_1(func, x), limits[1][0],
limits[1][1], num_of_nodes[1])
    return    integrators[0](inner,    limits[0][0],    limits[0][1],
num_of_nodes[0])
# отрисовка графика
def tao_graph(integrate_func, ar_params, label):
    X = list()
    Y = list()
    for  t  in  arange(ar_params[0],  ar_params[1]  +  ar_params[2],
ar_params[2]):
        X.append(t)
        Y.append(integrate_func(t))
    plt.plot(X, Y, label=label)
```

```
def generate_label(n, m, func1, func2):
    res = "nodes for 1st method = " + str(n) + "\nnodes for 2nd method =
" + str(m) + "\nmethods = "
    res += "Simpson" if func1 == simpson else "Gauss"
    res += "-Simpson" if func2 == simpson else "-Gauss"
    return res

# main
end = False
while not end:
    param = float(input("Enter param (tao): "))
    mode  =  bool(int(input("Enter  external  method  (0  -  Gauss;  1  -
Simpson): ")))
    func1 = simpson if mode else gauss
    mode  =  bool(int(input("Enter  internal  method  (0  -  Gauss;  1  -
Simpson): ")))
    func2 = simpson if mode else gauss
    N = int(input("Enter number of nodes for 1st method: "))
    M = int(input("Enter number of nodes for 2nd method: "))

    param_integrate = lambda tao: integrate(main_function(tao), [[0, pi
/ 2], [0, pi / 2]], [N, M], [func1, func2])
    print("Result with your parameter:", param_integrate(param))
    try:
        tao_graph(param_integrate, [0.05, 5, 0.05], generate_label(N, M,
func1, func2))
    except ValueError:
        print("in simpson method argument should be > 2 and not even;")
    end = bool(int(input("End? (0 - No, 1 - Yes): ")))

plt.legend()
plt.ylabel("Result")
plt.xlabel("Tao")
plt.show()
```

3. Результат работы программы

1. Описать алгоритм вычисления n корней полинома Лежандра n-ой степени Pn (x) при реализации формулы Гаусса.

Для вычисления коэффициентов полинома Лежандра n-ой степени была использована формула из определения:

$$P_n(x) = \frac{1}{2^n n!}\frac{d^n}{dx^n}[(x^2-1)^n]\ , \quad n = 0,1,2,...$$

Из свойств полиномов Лежандра известно, что полином $P_n(x)$ имеет n различных и действительных корней, расположенных на интервале [-1;1]. Однако заметим, что функция $(x^2 - 1)^n$ чётная, следовательно, её n-ая производная будет чётной или нечётной функцией, а тогда достаточно найти корни на отрезке [-1;0] (на отрезке [0;1] они будут равны по модулю найденным и иметь противоположный знак).

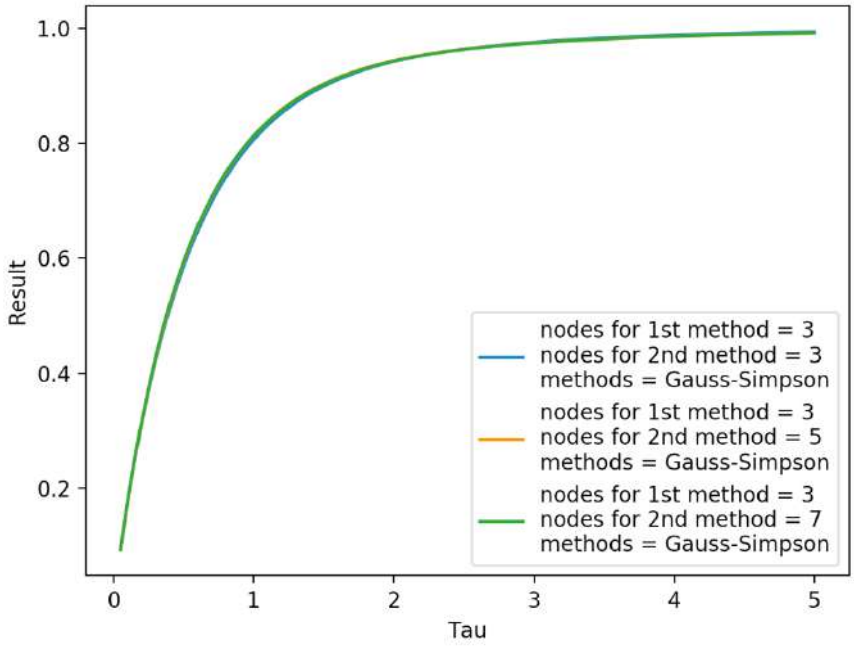
К отрезку [-1;0] применяется следующий метод: пока не будет найдено n / 2 корней, все текущие отрезки разбиваются пополам. Изначально текущим является отрезок [-1; 0]. Тогда в некоторый момент будут найдены все отрезки, на каждом из которых ровно один корень, далее к каждому отрезку применяется метод половинного деления.

Идея метода половинного деления состоит в следующем: если на концах отрезка значения функции имеют разный знак, то корень находится внутри этого отрезка. Изначально мы точно знаем, что отрезок содержит корень, поэтому делим отрезок пополам и проверяем, какой половине принадлежит корень. Таким образом мы итеративно уточняем значение корня.

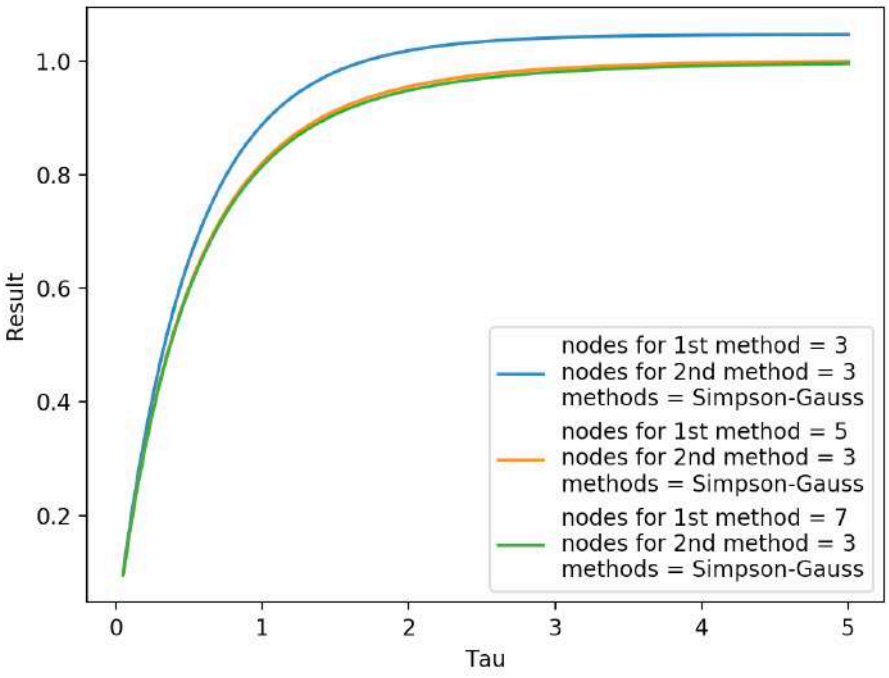
2. Исследовать влияние количества выбираемых узлов сетки по каждому направлению на точность расчетов.

Исследование метода Симпсона

1. Внешний – Гаусс; Внутренний - Симпсон



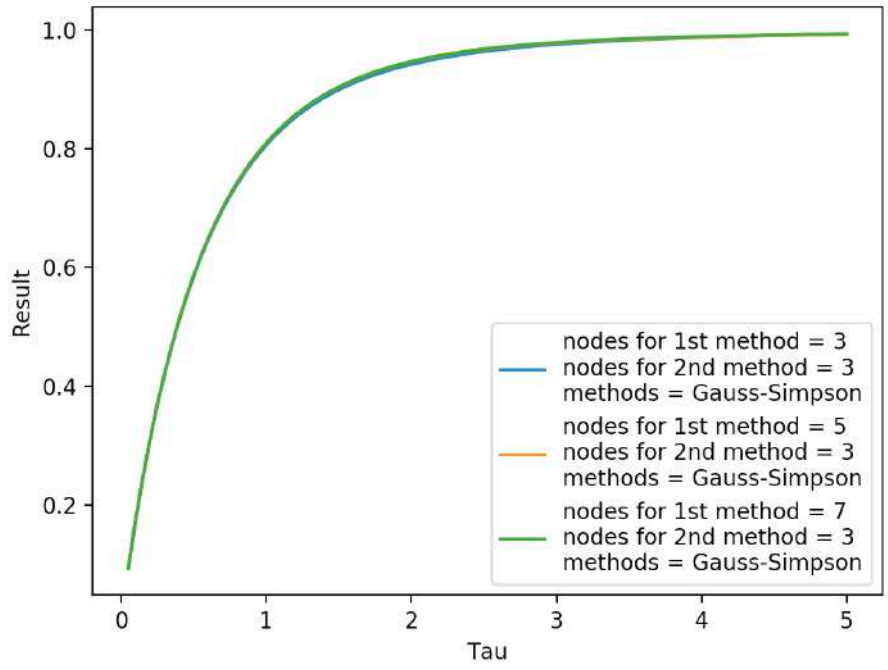
2. Внешний – Симпсон; Внутренний - Гаусс



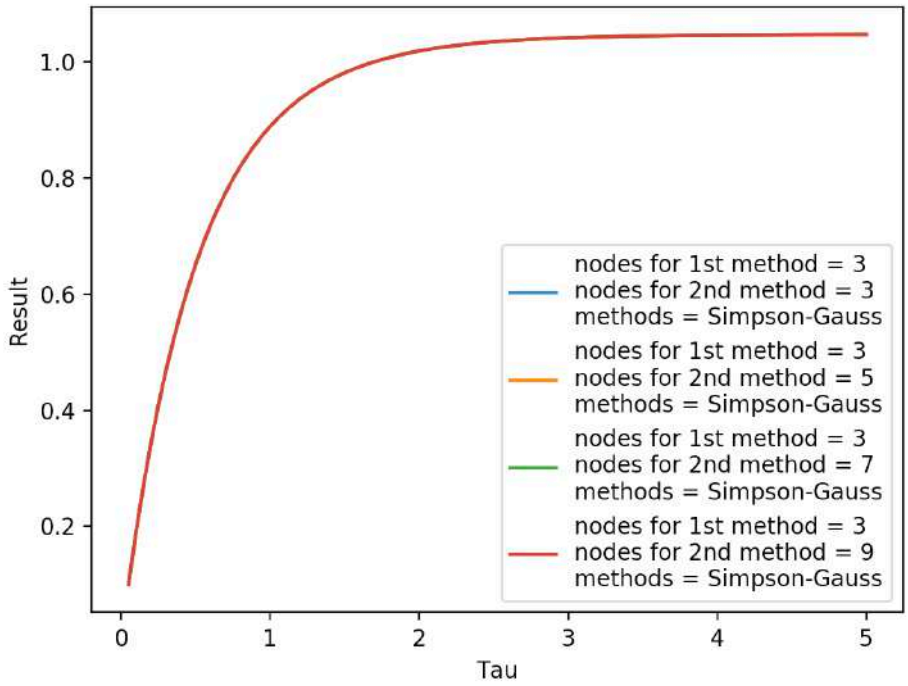
Из полученных зависимостей очевидно, что метод Симпсона даёт неточные результаты при небольшом количестве узлов (при этом результат интегрирования наиболее заметен на внешнем направлении).

Исследование метода Гаусса

3. Внешний – Гаусс; Внутренний - Симпсон



4. Внешний – Симпсон; Внутренний - Гаусс

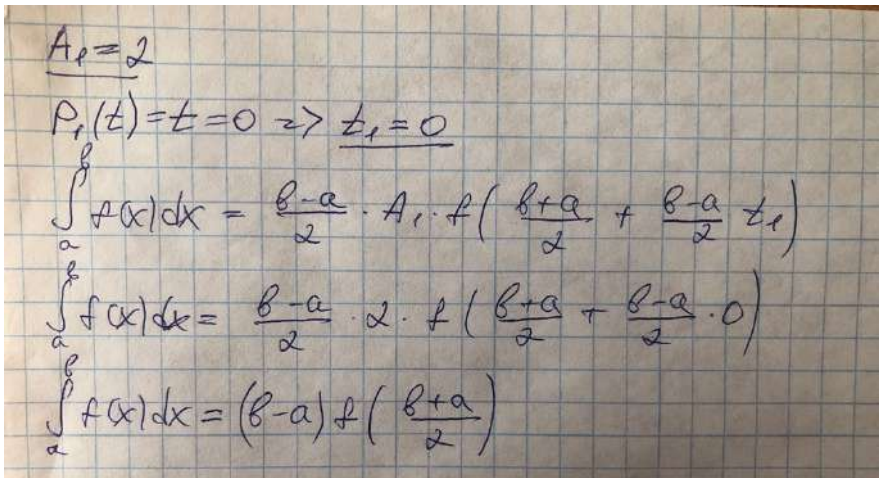


Из полученных зависимостей очевидно, что как на внешнем, так и на внутреннем направлениях при любом количестве узлов метод Гаусса остаётся достаточно точным.

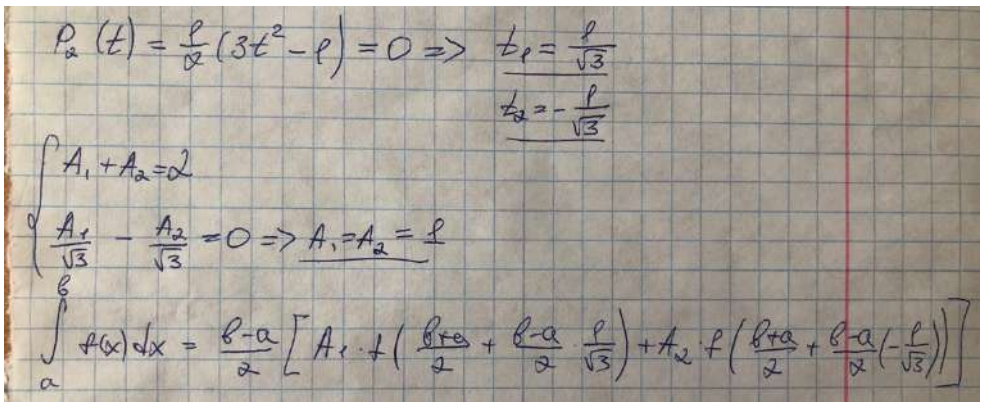
1. В каких ситуациях теоретический порядок квадратурных формул численного интегрирования не достигается.

Если подинтегральная функция не имеет соответствующих производных, то теоретический порядок точности не достигается. Так, если на отрезке интегрирования не существуют 3-я и 4-я производные, то порядок точности формулы Симпсона будет только 2-ой.

2. Построить формулу Гаусса численного интегрирования при одном узле.



3. Построить формулу Гаусса численного интегрирования при двух узлах.



4. Получить обобщенную кубатурную формулу для вычисления двойного интеграла методом последовательного интегрирования на основе формулы трапеций с тремя узлами по каждому направлению.