**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** Малышев И. А.

**Группа** ИУ7-41Б

**Оценка (баллы) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Преподаватель** Градов В.М.

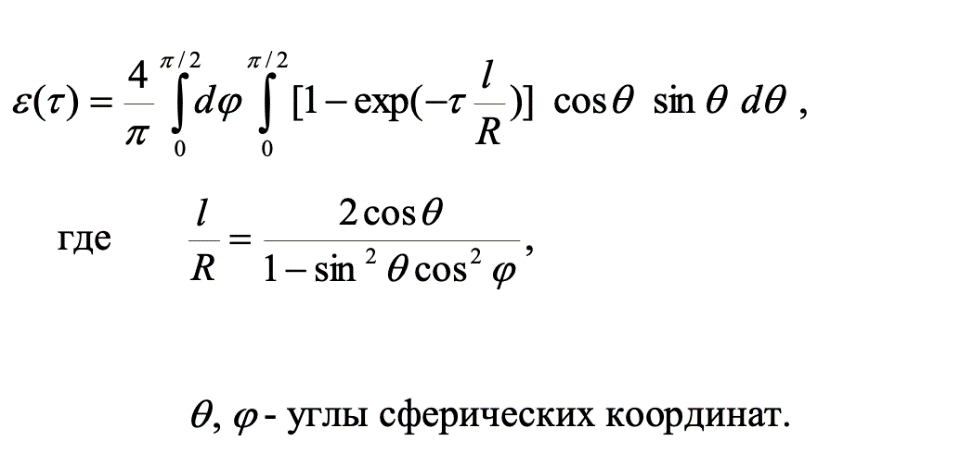
Москва.

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 :

****

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

1. **Код программы**

from math import pi, cos, sin, exp

from numpy import arange

import matplotlib.pyplot as plt

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

def derivative(pol):

dst = []

for i in range(1, len(pol)):

dst.append(pol[i] \* i)

return dst

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

def get\_pol\_value(pol, arg):

val = 0

for i in range(len(pol)):

val += pol[i] \* (arg \*\* i)

return val

def half\_div\_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\_div\_method(pol, left, mid, grad)

elif test < 0:

return half\_div\_method(pol, mid, right, grad)

else:

if test < 0:

return half\_div\_method(pol, left, mid, grad)

elif test > 0:

return half\_div\_method(pol, mid, right, grad)

return mid

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\_div\_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\_div\_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])

return roots

def solve\_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

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\_Gauss(matrix, len(nodes))

return res

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

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

def gauss(func, a, b, num\_of\_nodes):

leg = find\_leg(num\_of\_nodes)

args = find\_roots(leg)

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)

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])

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

def main():

end = False

while not end:

param = float(input("Tau: "))

mode = bool(int(input("external method (0 - Gauss; 1 - Simpson): ")))

func1 = simpson if mode else gauss

mode = bool(int(input("internal method (0 - Gauss; 1 - Simpson): ")))

func2 = simpson if mode else gauss

N = int(input("number of nodes for 1st method: "))

M = int(input("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:", 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("Tau")

plt.show()

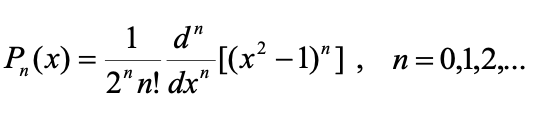
if \_\_name\_\_ == "\_\_main\_\_":

main()

1. **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:



Из свойств полиномов Лежандра известно, что полином Pn(x) имеет n различных и действительных корней, расположенных на интервале [-1;1]. Однако заметим, что функция (x2 - 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.

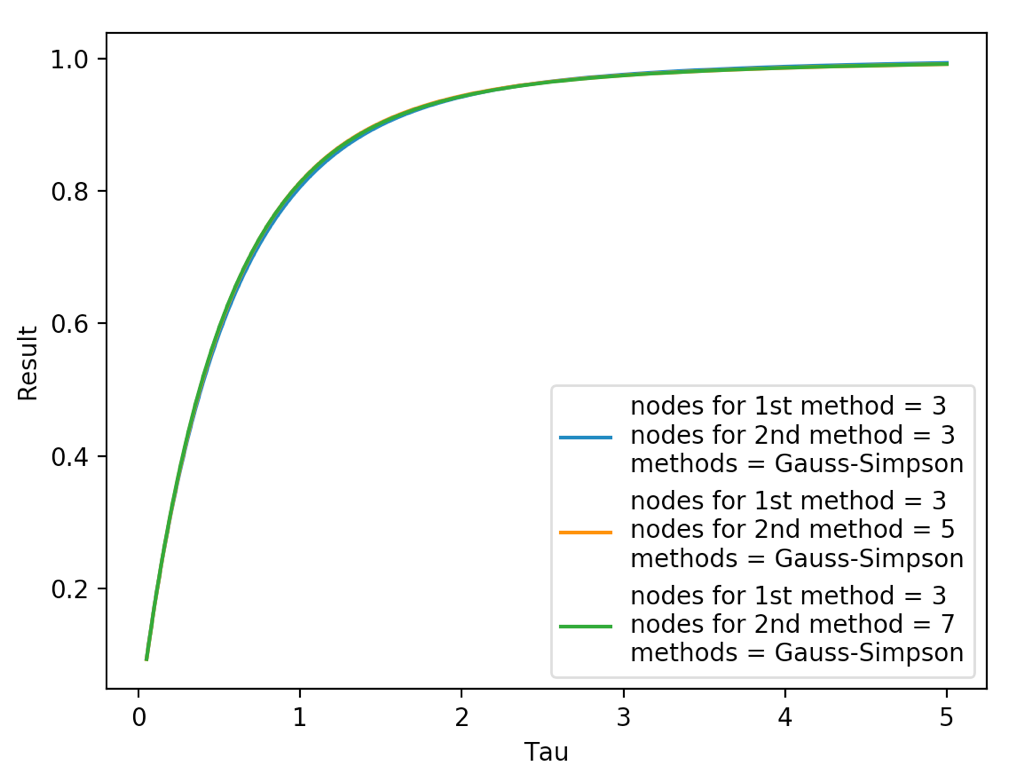
Идея метода половинного деления состоит в следующем: если на концах отрезка значения функции имеют разный знак, то корень находится внутри этого отрезка. Изначально мы точно знаем, что отрезок содержит корень, поэтому делим отрезок пополам и проверяем, какой половине принадлежит корень. Таким образом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

*2. Исследовать влияние количества выбираемых узлов сетки по каждому*

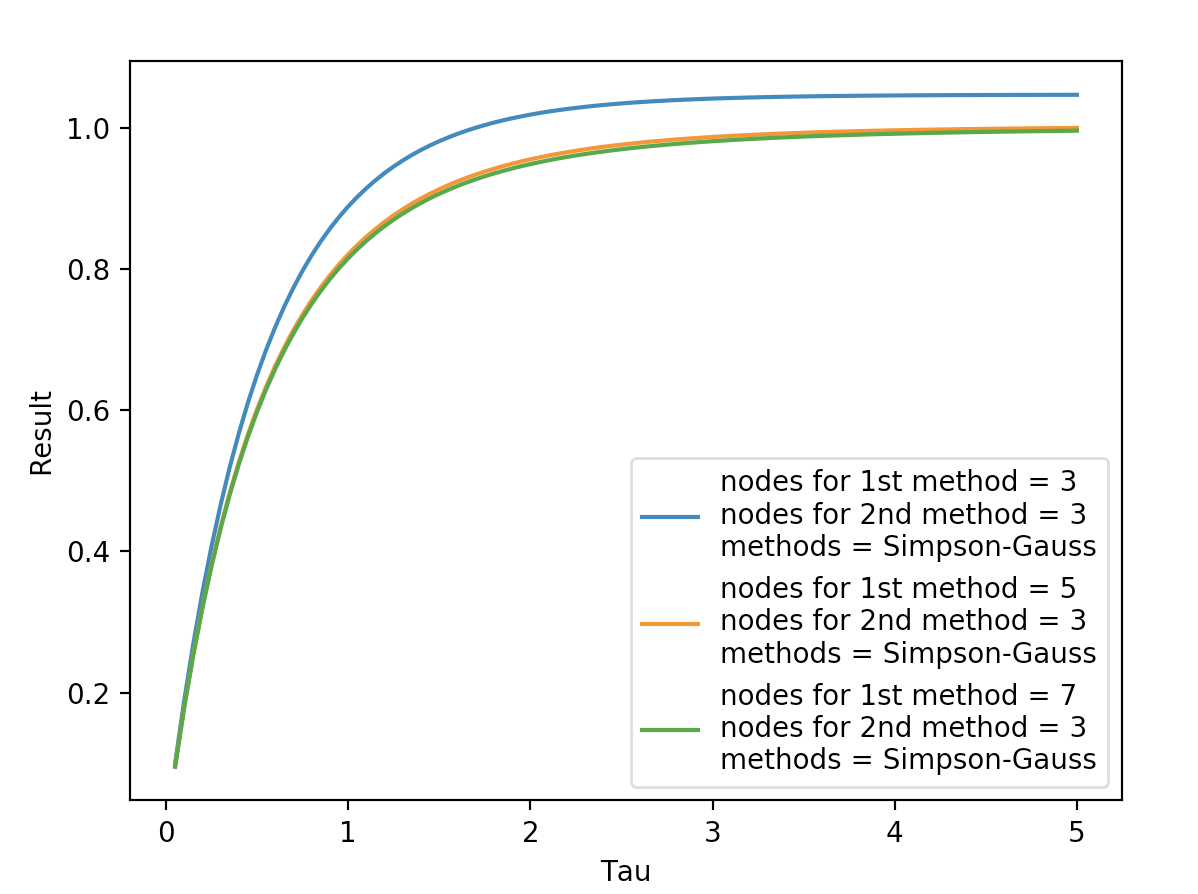
*направлению на точность расчетов.*

*Исследование метода Симпсона*

1. Внешний – Гаусс; Внутренний - Симпсон



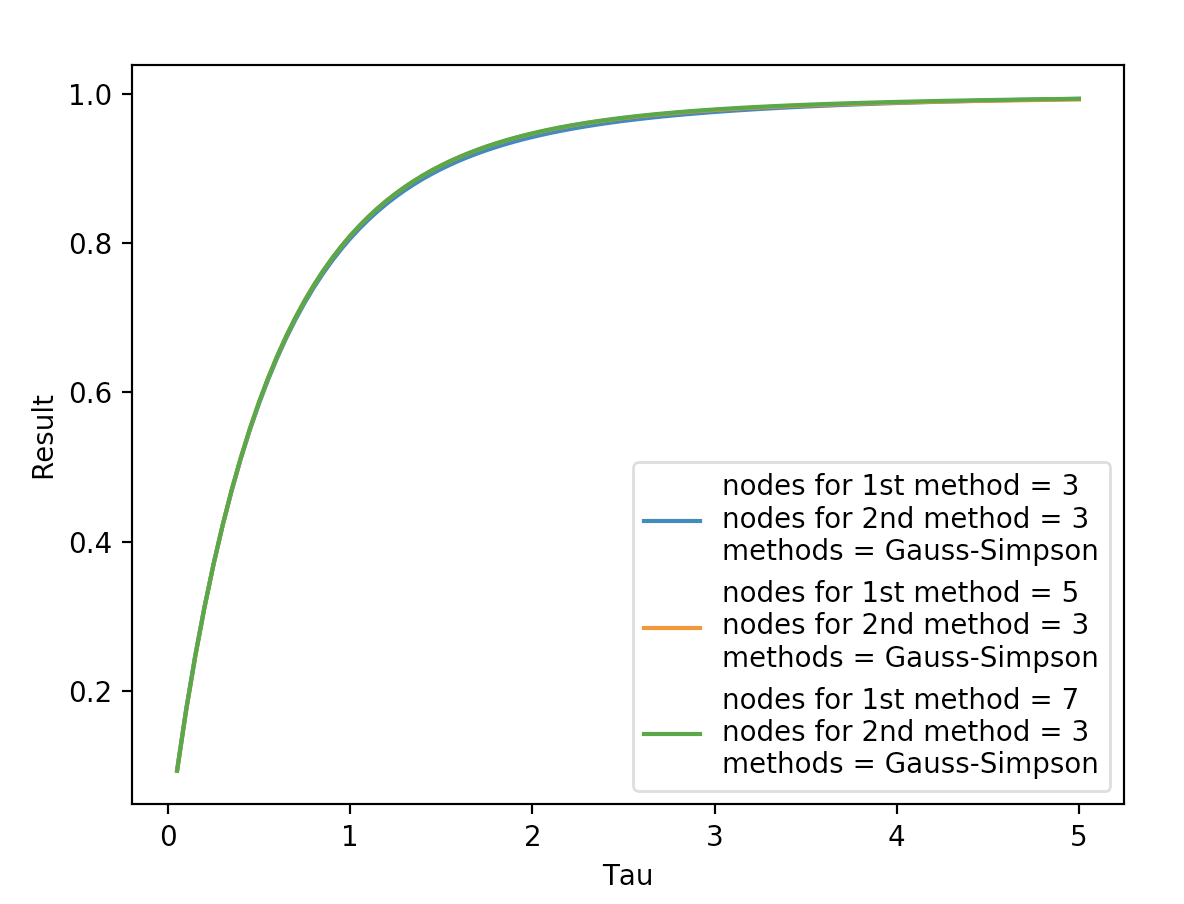
1. Внешний – Симпсон; Внутренний - Гаусс



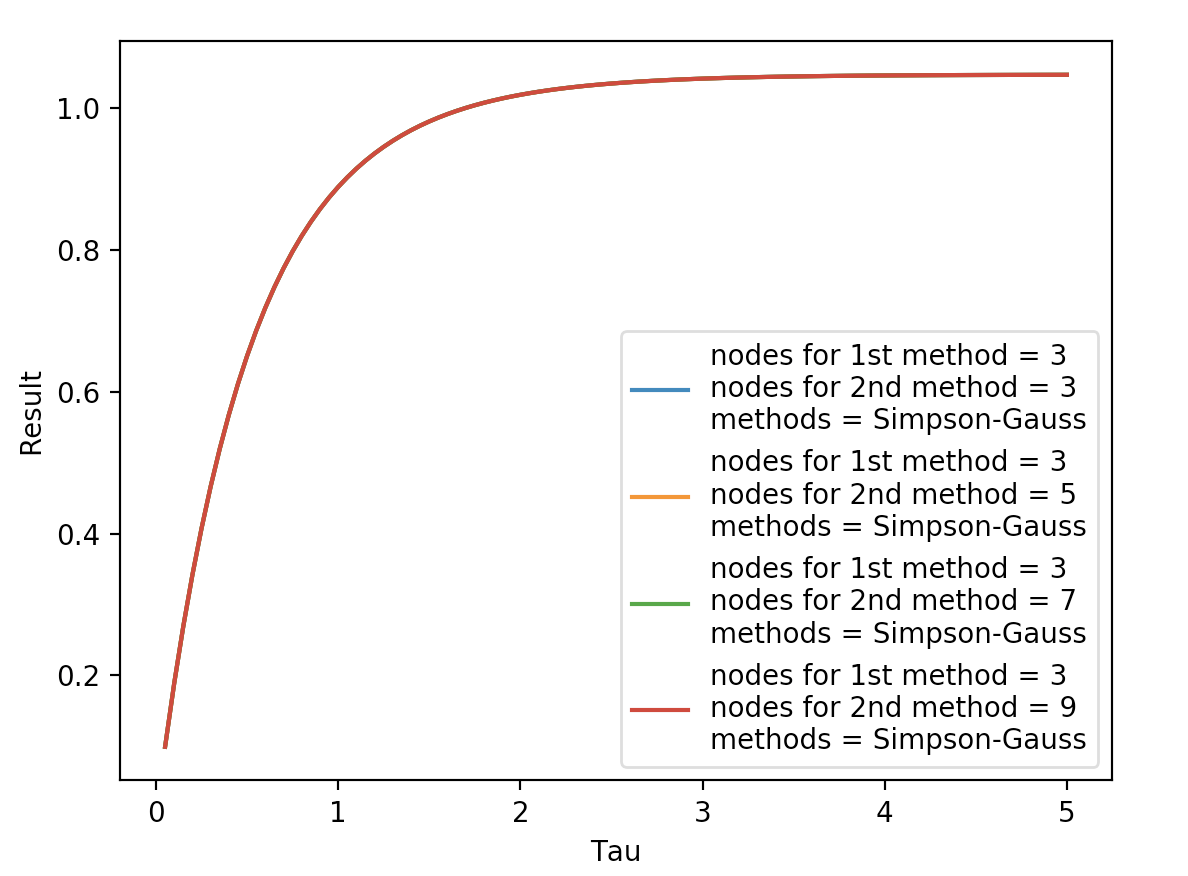
Из полученных зависимостей очевидно, что метод Симпсона даёт неточные результаты при небольшом количестве узлов (при этом результат интегрирования наиболее заметен на внешнем направлении).

*Исследование метода Гаусса*

1. Внешний – Гаусс; Внутренний - Симпсон



1. Внешний – Симпсон; Внутренний - Гаусс

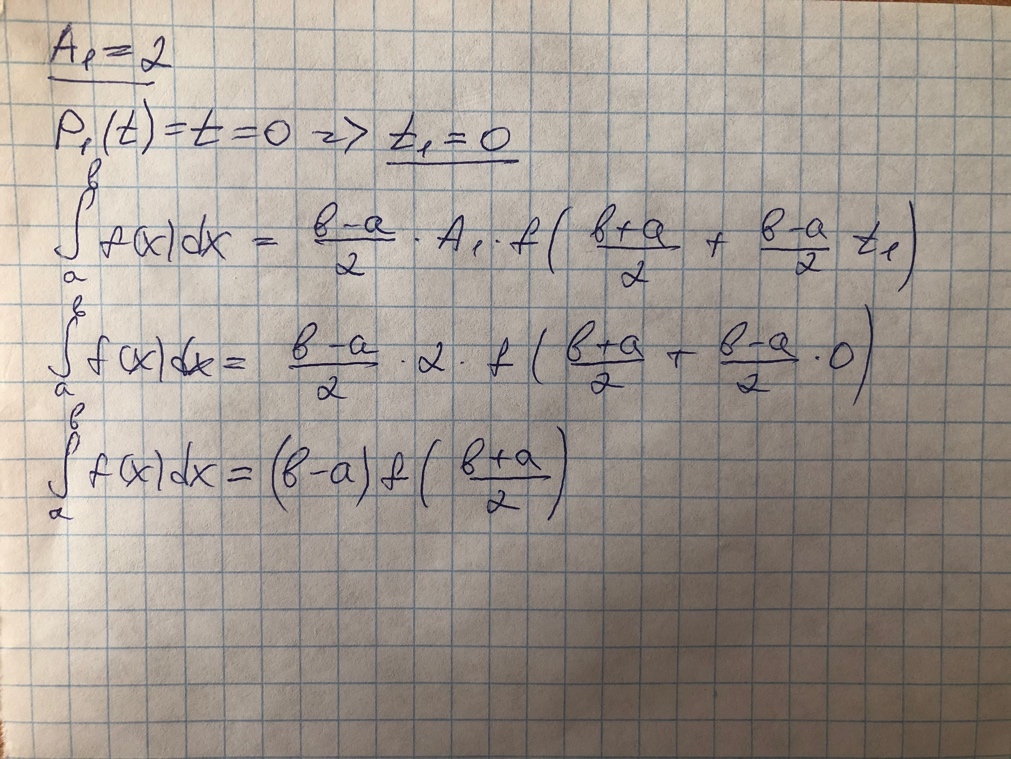


Из полученных зависимостей очевидно, что как на внешнем, так и на внутреннем направлениях при любом количестве узлов метод Гаусса остаётся достаточно точным.

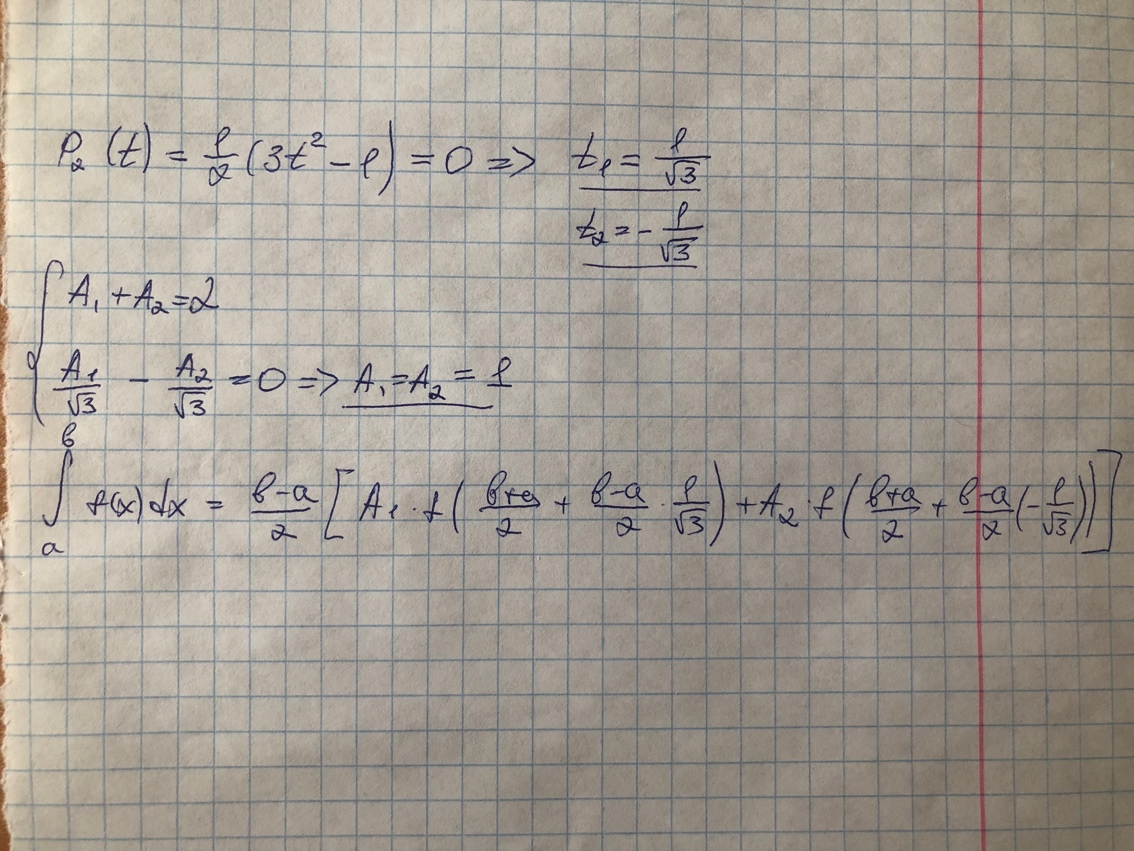
*1. В каких ситуациях теоретический порядок квадратурных формул численного интегрирования не достигается.*

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.*

