**Лабораторная работа №2**

**Численное решение интегральных уравнений**

**Вариант 11**

**Чеботаревский Никита**

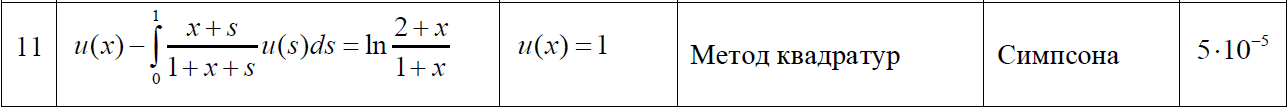
**3 курс, 8 группа**

**Постановка задачи**

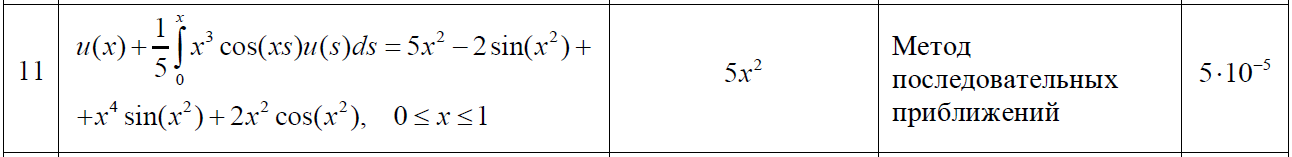
1) Найти решение интегрального уравнения Фредгольма второго рода с заданной точностью ε методом, указанным в варианте задания. Для вычисления интегралов использовать составную квадратурную формулу, указанную в варианте задания. Сравнить полученное с заданной точностью

приближенное решение {,…,} с точным решением *u*(*x*). В одной

системе координат построить графики точного и приближенных решений.



2) Найти решение интегрального уравнения Вольтерра второго рода с заданной точностью ε методом, указанным в варианте задания, использую квадратурную формулу трапеций. Сравнить полученное с заданной точностью приближенное решение {,…,} с точным решением *u*(*x*). В одной системе координат построить графики точного и приближенных решений.



Также в отчёте должна быть приведена следующая информация:

1) алгоритмы, применяемые для численного решения поставленных задач.

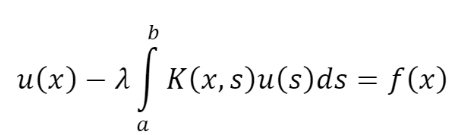
2) результаты решения поставленных задач.

3) выводы.

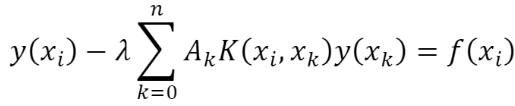
4) листинг программы с комментариями.

**Теоретические сведения**

**1) Метод квадратур для решения ИУФ-2:**



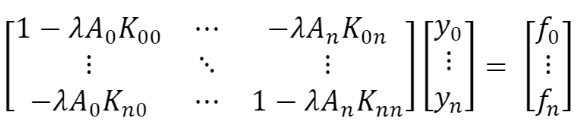
Для решения необходимо найти значение u(x) в узлах . Вместо интеграла подставим квадратурную сумму и получим:



y(x) является приближением искомой функции u(x),

также наше выражение можно записать в виде СЛАУ, где

Тогда можно это всё записать в следующем виде:



Так как мы используем квадратурную формулу Симпсона, то для чётного числа разбиений, получим:

Правило Рунге для практической оценки погрешности ИУФ-2:

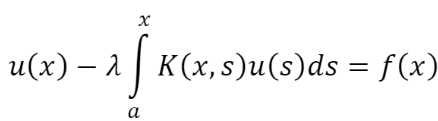
1) Строим сетку узлов для числа разбиений соответственно с шагом h и ().

2) для данной сетки узлов считаем, по алгоритму приведённому выше,

3) Считаем погрешность по формуле , где m – порядок точности квадратурной формулы. Если данное условие не выполняется, то

, после чего переходим опять к шагу (1).

**2) Метод последовательных приближений для ИУВ – 2:**



В данной варианте мы используем квадратурную формулу трапеций, поэтому находить приближённое значение функции u(x) будем по следующим формулам:

Правило Рунге для практической оценки погрешности ИУВ-2:

1) Строим сетку узлов для числа разбиений соответственно с шагом h и ().

2) для данной сетки узлов считаем, по алгоритму приведённому выше,

при этом считаем до тех пор, пока не выполнится условие:

и аналогичное условие для второй сетки:

3) Считаем погрешность по формуле , где m – порядок точности квадратурной формулы. Если данное условие не выполняется, то

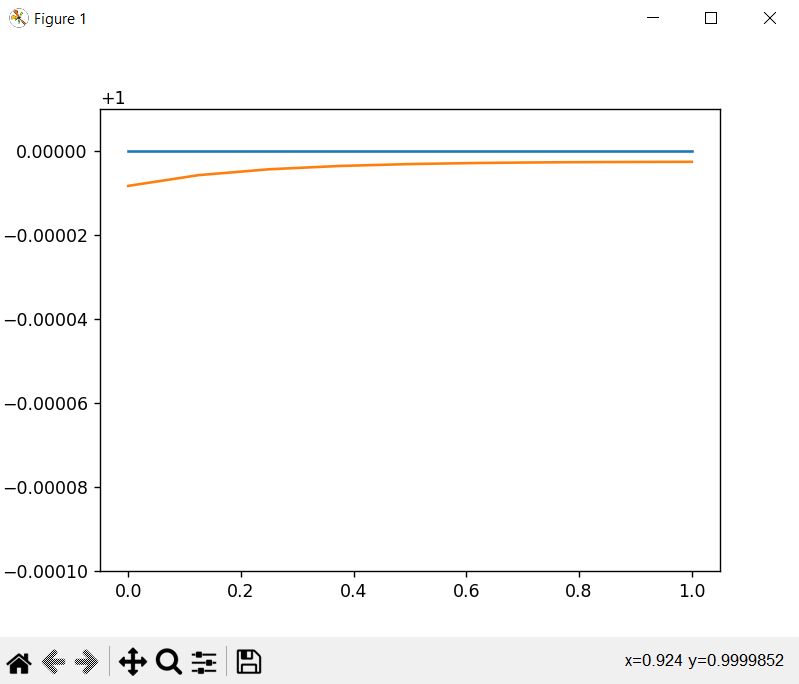
, после чего переходим опять к шагу (1).

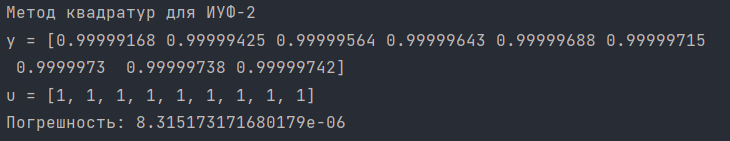
**Листинг программы**

*import* numpy *as* np  
*import* math  
*import* matplotlib.pyplot *as* plt  
  
  
*class* Iuf:  
 *def* \_\_init\_\_(*self*, a, b, lamb, eps):  
 *self*.a = a  
 *self*.b = b  
 *self*.lamb = lamb  
 *self*.eps = eps  
  
 @staticmethod  
 *def* accuracy(y, y2, m):  
 result = 0  
 *for* i *in* range(len(y)):  
 tmp = abs(y2[0 + i \* 2] - y[i])  
 *if* tmp > result:  
 result = tmp  
  
 *return* result / (2 \*\* m - 1)  
  
 @staticmethod  
 *def* count\_accuracy(y, u):  
 result = 0  
 *for* i *in* range(len(y)):  
 tmp = abs(y[i] - u[i])  
 *if* tmp > result:  
 result = tmp  
 *return* result  
  
 *def* runge\_rule(*self*, func, m):  
 n1, n2 = 2, 4  
 *while True*:  
 y1 = func(n1)  
 y2 = func(n2)  
 acc = *self*.accuracy(y1, y2, m)  
 *if* acc <= *self*.eps:  
 *return* y2  
 *else*:  
 n1 = n2  
 n2 = n2 \* 2  
  
 *def* h(*self*, n):  
 *return* (*self*.b - *self*.a) / n  
  
 *def* x(*self*, n):  
 *return* [*self*.a + i \* *self*.h(n) *for* i *in* range(n + 1)]  
  
 *def* show\_result(*self*, y, u):  
 print(f"y = {y}")  
 print(f"u = {u}")  
 print(f"Погрешность: {*self*.count\_accuracy(y, u)}", end="\n" \* 3)  
  
  
*class* QuadratureMethod(Iuf):  
 *def* \_\_init\_\_(*self*, a, b, lamb, eps):  
 super().\_\_init\_\_(a, b, lamb, eps)  
  
 @staticmethod  
 *def* f(x):  
 *return* np.log((2 + x) / (1 + x))  
  
 @staticmethod  
 *def* k(x, s):  
 *return* (x + s) / (1 + x + s)  
  
 *def* count\_a(*self*, n):  
 coefficients = []  
 step = *self*.h(n)  
 *for* i *in* range(n + 1):  
 *if* i == 0 *or* i == n:  
 coefficients.append(step / 3)  
 *elif* i % 2 == 1:  
 coefficients.append(4 \* step / 3)  
 *else*:  
 coefficients.append(2 \* step / 3)  
  
 *return* coefficients  
  
 *def* count\_y(*self*, n):  
 coefficients = *self*.count\_a(n)  
 x = *self*.x(n)  
 matrix, fxi = [], []  
 *for* i *in* range(n + 1):  
 tmp = []  
 *for* j *in* range(n + 1):  
 *if* i == j:  
 tmp.append(1 - *self*.lamb \* coefficients[i] \* *self*.k(x[i], x[i]))  
 *else*:  
 tmp.append(-*self*.lamb \* coefficients[j] \* *self*.k(x[i], x[j]))  
  
 matrix.append(tmp)  
 fxi.append(*self*.f(x[i]))  
  
 *return* np.linalg.solve(matrix, fxi)  
  
  
*class* SuccessiveApproximations(Iuf):  
 *def* \_\_init\_\_(*self*, a, b, lamb, eps):  
 super().\_\_init\_\_(a, b, lamb, eps)  
  
 @staticmethod  
 *def* f(x):  
 *return* 5 \* x \*\* 2 - 2 \* math.sin(x \*\* 2) + (x \*\* 4) \* math.sin(x \*\* 2) + 2 \* (x \*\* 2) \* math.cos(x \*\* 2)  
  
 @staticmethod  
 *def* k(x, s):  
 *return* (x \*\* 3) \* math.cos(x \* s)  
  
 *def* count\_y(*self*, y, n):  
 step = *self*.h(n)  
 x = *self*.x(n)  
 result = []  
 *for* i *in* range(len(x)):  
 tmp = *self*.lamb \* (step / 2) \* (*self*.k(x[i], x[0]) \* y[0] + *self*.k(x[i], x[i]) \* y[i]) + *self*.f(x[i])  
 s = 0  
 *for* j *in* range(1, i):  
 s += (*self*.k(x[i], x[j]) \* y[j])  
  
 s \*= (*self*.lamb \* step)  
 result.append(tmp + s)  
  
 *return* result  
  
 *def* count\_y\_with\_accuracy(*self*, n):  
 y = [0] \* (n + 1)  
  
 *while True*:  
 y1 = *self*.count\_y(y, n)  
  
 *if self*.count\_accuracy(y1, y) <= *self*.eps / 100:  
 *return* y1  
 *else*:  
 y = y1  
  
  
print("Метод квадратур для ИУФ-2")  
qm = QuadratureMethod(0, 1, 1, 5 \* 10 \*\* (-5))  
y\_res = qm.runge\_rule(qm.count\_y, 4)  
x\_res = qm.x(len(y\_res) - 1)  
u\_res = [1] \* len(y\_res)  
qm.show\_result(y\_res, u\_res)  
plt.plot(x\_res, u\_res)  
plt.plot(x\_res, y\_res)  
plt.ylim(0.9999, 1.00000999)  
plt.show()  
  
  
print("Метод последовательных приближений для ИУВ-2")  
sa = SuccessiveApproximations(0, 1, -1 / 5, 5 \* 10 \*\* (-5))  
y\_res = sa.runge\_rule(sa.count\_y\_with\_accuracy, 2)  
x\_res = sa.x(len(y\_res) - 1)  
u\_res = [5 \* elem \*\* 2 *for* elem *in* x\_res]  
sa.show\_result(y\_res, u\_res)  
plt.plot(x\_res, u\_res, x\_res, y\_res)  
plt.show()

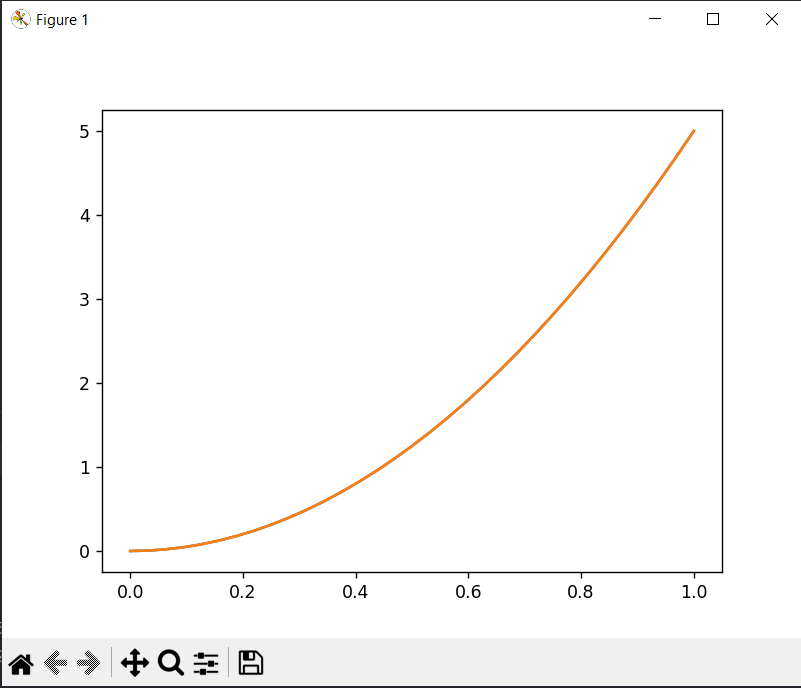
**Результаты программы**

**Задание 1:**

****

****

**Задание 2:**

****

**Метод последовательных приближений для ИУВ-2**

y = [0.0, 0.004882812344779719, 0.019531247516512213, 0.043945299928151625, 0.0781249602736386, 0.12207021554728538, 0.17578104909261974, 0.23925744070234034, 0.3124993667970962, 0.39550680071667726, 0.4882797131627533, 0.5908180728373272, 0.7031218473253075, 0.8251910042727304, 0.9570255129137601, 1.0986253459991917, 1.2499904821762535, 1.4111209088634653, 1.5820166256545694, 1.7626776482715787, 1.9531040130682422, 2.153295782061407, 2.363253048438693, 2.5829759424566783, 2.8124646376049474, 3.0517193568686434, 3.3007403788769376, 3.559528043678707, 3.8280827578417904, 4.1064049985306745, 4.394495316181774, 4.692354335367642, 4.999982753423197]

u = [0.0, 0.0048828125, 0.01953125, 0.0439453125, 0.078125, 0.1220703125, 0.17578125, 0.2392578125, 0.3125, 0.3955078125, 0.48828125, 0.5908203125, 0.703125, 0.8251953125, 0.95703125, 1.0986328125, 1.25, 1.4111328125, 1.58203125, 1.7626953125, 1.953125, 2.1533203125, 2.36328125, 2.5830078125, 2.8125, 3.0517578125, 3.30078125, 3.5595703125, 3.828125, 4.1064453125, 4.39453125, 4.6923828125, 5.0]

Погрешность: 4.226882129287901e-05

**Вывод**

Сравнивая полученные значения для МК и МПП можно сделать вывод, что задача решается верно, так как для различных методов получаем близкие значения в узлах.

В сравнении с МПП МК является более трудоёмким, так как при использовании МК необходимо решить СЛАУ.