Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Лабораторная работа № 11

«Решение краевых задач. Методы коллокаций, наименьших квадратов и Галеркина»

Вариант 10

Выполнил:

студент гр. 953506

Кондрашов И.Д.

Проверил:

Анисимов В.Я.

Минск, 2021

**Цель работы:**

* изучить методы коллокаций, наименьших квадратов и Галеркина, составить алгоритмы методов и программы их реализаций, составить алгоритм решения краевых задач указанными методами, применимыми для организации вычислений на ПЭВМ;
* составить программу решения краевых задач по разработанным алгоритмам;
* выполнить тестовые примеры и проверить правильность работы программ.
* получить численное решение заданной краевой задачи.

**Краткие теоретические сведения**

**Краевая задача** (граничная задача) — задача о нахождении решения заданного [дифференциального уравнения](https://ru.wikipedia.org/wiki/%D0%94%D0%B8%D1%84%D1%84%D0%B5%D1%80%D0%B5%D0%BD%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%83%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5) (системы дифференциальных уравнений), удовлетворяющего [краевым (граничным) условиям](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%BD%D1%8B%D0%B5_%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%B8%D1%8F) в концах интервала или на границе области. Краевые задачи для [гиперболических](https://ru.wikipedia.org/wiki/%D0%93%D0%B8%D0%BF%D0%B5%D1%80%D0%B1%D0%BE%D0%BB%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D1%83%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5) и [параболических уравнений](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B1%D0%BE%D0%BB%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D1%83%D1%80%D0%B0%D0%B2%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5) часто называют **начально-краевыми** или **смешанными**, потому что в них задаются не только граничные, но и [начальные условия](https://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%87%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D1%83%D1%81%D0%BB%D0%BE%D0%B2%D0%B8%D1%8F).

**Ищем приближенное решение в виде**

𝑦𝑛(𝑥)=𝜙0(𝑥)+∑𝑁𝑖=0𝜙𝑖(𝑥)∗𝑎𝑖 , где 𝜙𝑖 - базисная система, 𝑎𝑖 - неизвестные коэффициенты

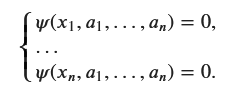
**Рассматриваем невязку:**

𝜓(𝑥1,𝑎1,...,𝑎𝑛)=𝑝(𝑥)𝑦″𝑛(𝑥)+𝑞(𝑥)𝑦′𝑛(𝑥)+𝑟(𝑥)𝑦𝑛(𝑥)−𝑓(𝑥)

**Все способы решения задачи сводятся к решению СЛАУ относительно коэффициентов 𝑎𝑖**

**Способы решения краевых задач**

**Метод коллокаций**

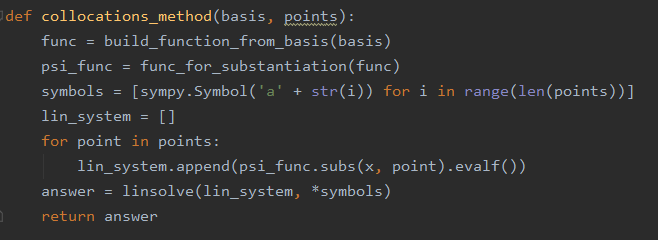


Коэффициенты в матрице:

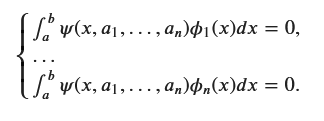
**

Вектор свободных членов:

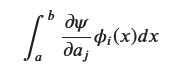
**



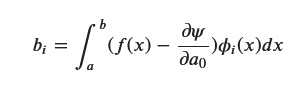
**Метод Галеркина**

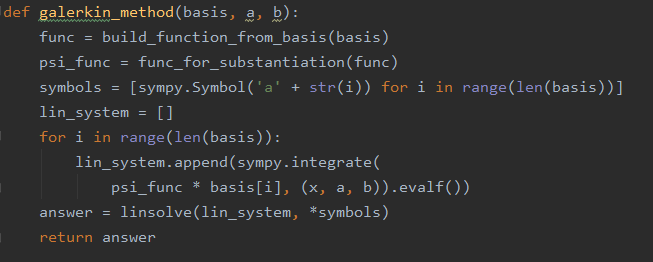


Коэффициенты в матрице:

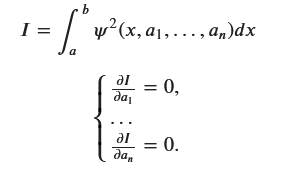


Вектор свободных членов:

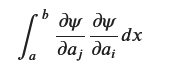
**

****

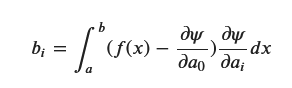
**Интегральный МНК**

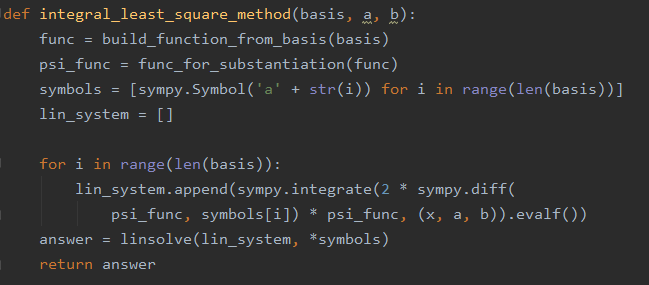
**

Коэффициенты в матрице:

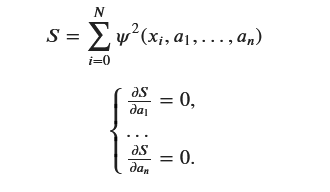


Вектор свободных членов:

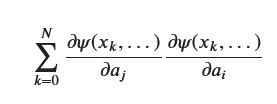
****



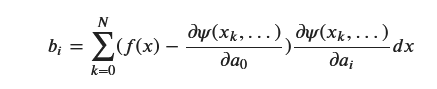
**Дискретный МНК**

****

Коэффициенты в матрице:

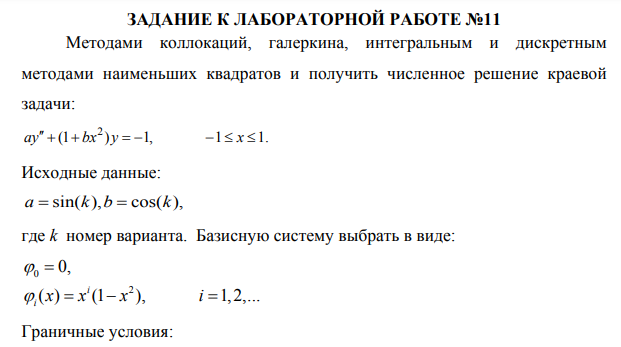


Вектор свободных членов:

****

**Text

Description automatically generated**



y(-1) = 0,

y(1) = 0.

**Результат выполнения**

**Text

Description automatically generated**

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

**Листинг кода**

import math  
  
import numpy  
import pylab  
import sympy  
from sympy.solvers.solveset import linsolve  
from numpy import linspace  
import functools  
  
x = sympy.Symbol('x')  
  
VARIANT = 10  
VARIABLES = 5  
DIFF\_POINTS = 5  
  
a = -1  
b = 1  
  
  
def func\_for\_substantiation(subs):  
 func = sympy.sin(VARIANT) \* sympy.diff(subs, x, x) + ((1 + sympy.cos(VARIANT) \* x \*\* 2) \* subs + 1)  
 return func  
  
  
def generate\_basis\_sequence(n):  
 sequence = []  
 for i in range(n):  
 sequence.append((x \*\* i) \* (1 - x \*\* 2))  
 return sequence  
  
  
def build\_function\_from\_basis(basis):  
 result = 0  
 for i in range(len(basis)):  
 current\_a = sympy.Symbol('a' + str(i))  
 result += current\_a \* basis[i]  
 return result  
  
  
def collocations\_method(basis, points):  
 func = build\_function\_from\_basis(basis)  
 psi\_func = func\_for\_substantiation(func)  
 symbols = [sympy.Symbol('a' + str(i)) for i in range(len(points))]  
 lin\_system = []  
 for point in points:  
 lin\_system.append(psi\_func.subs(x, point).evalf())  
 answer = linsolve(lin\_system, \*symbols)  
 return answer  
  
  
def integral\_least\_square\_method(basis, a, b):  
 func = build\_function\_from\_basis(basis)  
 psi\_func = func\_for\_substantiation(func)  
 symbols = [sympy.Symbol('a' + str(i)) for i in range(len(basis))]  
 lin\_system = []  
 for i in range(len(basis)):  
 lin\_system.append(sympy.integrate(2 \* sympy.diff(  
 psi\_func, symbols[i]) \* psi\_func, (x, a, b)).evalf())  
 answer = linsolve(lin\_system, \*symbols)  
 return answer  
  
  
def discrete\_least\_square\_method(basis, points\_num, a, b):  
 func = build\_function\_from\_basis(basis)  
 psi\_func = func\_for\_substantiation(func)  
 seq = [psi\_func.subs(x, point) \*\* 2 for point in linspace(a + 0.05, b - 0.05, points\_num)]  
 psi\_sqr\_sum = functools.reduce((lambda a, b: a + b), seq)  
 symbols = [sympy.Symbol('a' + str(i)) for i in range(len(basis))]  
 lin\_system = []  
 for i in range(len(basis)):  
 lin\_system.append(sympy.diff(psi\_sqr\_sum, symbols[i]).evalf())  
 answer = linsolve(lin\_system, \*symbols)  
 return answer  
  
  
def galerkin\_method(basis, a, b):  
 func = build\_function\_from\_basis(basis)  
 psi\_func = func\_for\_substantiation(func)  
 symbols = [sympy.Symbol('a' + str(i)) for i in range(len(basis))]  
 lin\_system = []  
 for i in range(len(basis)):  
 lin\_system.append(sympy.integrate(  
 psi\_func \* basis[i], (x, a, b)).evalf())  
 answer = linsolve(lin\_system, \*symbols)  
 return answer  
  
  
def get\_basis\_function(n):  
 if not n:  
 return lambda x: 0  
 return lambda x: x \*\* (n - 1) \* (1 - x \*\* 2)  
  
  
def get\_basis\_system(num\_of\_basis\_functions):  
 return [get\_basis\_function(n) for n in range(num\_of\_basis\_functions)]  
  
  
def numerical\_integration(f, a, b):  
 n = 100  
 dx = (b - a) / n  
 xlist = numpy.arange(a, b, dx)  
 ylist = [f(p) for p in xlist]  
 return numpy.trapz(ylist, dx=dx)  
  
  
def numerical\_diff(f, x, n):  
 h = 0.001  
 if not n:  
 return f(x)  
 elif n == 1:  
 return (f(x + h) - f(x - h)) / (2 \* h)  
 elif n == 2:  
 return (f(x - h) - 2 \* f(x) + f(x + h)) / (h \*\* 2)  
 else:  
 raise NotImplementedError  
  
  
def integral\_LSM(coefficients, f, num\_of\_basis\_functions, a, b):  
 basis = get\_basis\_system(num\_of\_basis\_functions)  
  
 def resudial\_part\_diff(a):  
 return lambda x: sum(coefficients[i](x) \* numerical\_diff(basis[a], x, i)  
 for i in range(len(coefficients)))  
  
 part\_diffs = [resudial\_part\_diff(i) for i in range(num\_of\_basis\_functions)]  
  
 matrix = [[numerical\_integration(lambda x: part\_diffs[i](x) \* part\_diffs[j](x), a, b)  
 for j in range(1, num\_of\_basis\_functions)]  
 for i in range(1, num\_of\_basis\_functions)]  
 right\_side = [numerical\_integration(  
 lambda x: (f(x) - part\_diffs[0](x)) \* part\_diffs[i](x), a, b)  
 for i in range(1, num\_of\_basis\_functions)]  
 answer = numpy.linalg.solve(numpy.matrix(matrix), numpy.array(right\_side))  
 return lambda x: basis[0](x) + sum(answer[i - 1] \* basis[i](x)  
 for i in range(1, num\_of\_basis\_functions))  
  
  
import time  
  
  
def discrete\_LSM(coefficients, f, num\_of\_basis\_functions, points):  
 basis = get\_basis\_system(num\_of\_basis\_functions)  
  
 def resudial\_part\_diff(a, x):  
 return sum(coefficients[i](x) \* numerical\_diff(basis[a], x, i)  
 for i in range(len(coefficients)))  
  
 part\_diffs\_x = {(i, x): resudial\_part\_diff(i, x) for i in range(num\_of\_basis\_functions)  
 for x in points}  
  
 matrix = [[sum(part\_diffs\_x[(i, x)] \* part\_diffs\_x[(j, x)] for x in points)  
 for j in range(1, num\_of\_basis\_functions)]  
 for i in range(1, num\_of\_basis\_functions)]  
 right\_side = [sum([(f(x) - part\_diffs\_x[(0, x)]) \* part\_diffs\_x[(i, x)] for x in points])  
 for i in range(1, num\_of\_basis\_functions)]  
 answer = numpy.linalg.solve(numpy.matrix(matrix), numpy.array(right\_side))  
 return lambda x: basis[0](x) + sum(answer[i - 1] \* basis[i](x)  
 for i in range(1, num\_of\_basis\_functions))  
  
  
def Galerkin\_method(coefficients, f, num\_of\_basis\_functions, a, b):  
 basis = get\_basis\_system(num\_of\_basis\_functions)  
  
 def resudial\_part\_diff(a):  
 return lambda x: sum(coefficients[i](x) \* numerical\_diff(basis[a], x, i)  
 for i in range(len(coefficients)))  
  
 part\_diffs = [resudial\_part\_diff(i) for i in range(num\_of\_basis\_functions)]  
  
 def optimal\_numerical\_integration(f1, f2, points):  
 dx = 2 / len(points);  
 return sum([(f1[i] \* f2[i] \* dx) for i in range(0, len(points))])  
  
 points = numpy.linspace(a, b, 100)  
 points = [(points[i] + points[i - 1]) / 2 for i in range(1, len(points))]  
  
 opt\_part\_diffs = [None] \* num\_of\_basis\_functions  
 opt\_basis = [None] \* num\_of\_basis\_functions  
 for i in range(1, num\_of\_basis\_functions):  
 opt\_part\_diffs[i] = []  
 opt\_basis[i] = []  
 for point in points:  
 opt\_part\_diffs[i].append(part\_diffs[i](point))  
 opt\_basis[i].append(basis[i](point))  
  
 matrix = [[optimal\_numerical\_integration(opt\_part\_diffs[j], opt\_basis[i], points)  
 for j in range(1, num\_of\_basis\_functions)]  
 for i in range(1, num\_of\_basis\_functions)]  
 right\_side = [numerical\_integration(  
 lambda x: (f(x) - part\_diffs[0](x)) \* basis[i](x), a, b)  
 for i in range(1, num\_of\_basis\_functions)]  
 answer = numpy.linalg.solve(numpy.matrix(matrix), numpy.array(right\_side))  
 return lambda x: basis[0](x) + sum(answer[i - 1] \* basis[i](x)  
 for i in range(1, num\_of\_basis\_functions))  
  
  
def collocation\_method(coefficients, f, collocation\_points):  
 num\_of\_basis\_functions = len(collocation\_points) + 1  
 basis = get\_basis\_system(num\_of\_basis\_functions)  
  
 def resudial\_part\_diff(a):  
 return lambda x: sum(coefficients[i](x) \* numerical\_diff(basis[a], x, i)  
 for i in range(len(coefficients)))  
  
 matrix = [[resudial\_part\_diff(i)(point) for i in range(1, num\_of\_basis\_functions)]  
 for point in collocation\_points]  
 right\_side = [f(point) - resudial\_part\_diff(0)(point)  
 for point in collocation\_points]  
 answer = numpy.linalg.solve(numpy.matrix(matrix), numpy.array(right\_side))  
 return lambda x: basis[0](x) + sum(answer[i - 1] \* basis[i](x)  
 for i in range(1, num\_of\_basis\_functions))  
  
  
print('collocations method: \n',  
 collocations\_method(generate\_basis\_sequence(VARIABLES),  
 linspace(a + 0.2, b - 0.2, VARIABLES)))  
  
print('integral least square method: \n',  
 integral\_least\_square\_method(generate\_basis\_sequence(VARIABLES), a, b))  
  
print('discrete least square method: \n',  
 discrete\_least\_square\_method(generate\_basis\_sequence(VARIABLES),  
 VARIABLES + DIFF\_POINTS, a, b))  
  
print('galerkin method: \n',  
 galerkin\_method(generate\_basis\_sequence(VARIABLES), a, b))  
  
coeffs = [lambda x: 1 + math.cos(VARIANT) \* x \*\* 2,  
 lambda x: 0,  
 lambda x: math.sin(VARIANT)]  
f = lambda x: -1  
  
  
def show\_plots(functions, start\_x, end\_x, dx, title):  
 for function in functions:  
 x\_list = numpy.arange(start\_x, end\_x, dx)  
 y\_list = [function(p) for p in x\_list]  
 pylab.plot(x\_list, y\_list)  
 pylab.title(title)  
 pylab.grid(True)  
 pylab.show()  
  
  
collocation\_points = numpy.linspace(a, b, 100)  
answer\_2\_with\_collocation\_method = \  
 collocation\_method(coeffs, f, collocation\_points)  
show\_plots([answer\_2\_with\_collocation\_method], a, b, 0.01, "Collocations method")  
  
answer\_2\_with\_integral\_LSM\_method = \  
 integral\_LSM(coeffs, f, 50, a, b)  
show\_plots([answer\_2\_with\_integral\_LSM\_method], a, b, 0.01, "Integral least squared method")  
  
points = numpy.linspace(a, b, 200)  
answer\_2\_with\_discrete\_LSM\_method = \  
 discrete\_LSM(coeffs, f, 100, points)  
show\_plots([answer\_2\_with\_discrete\_LSM\_method], a, b, 0.01, "Discrete least squared method")  
  
answer\_2\_with\_Galerkin\_method = Galerkin\_method(coeffs, f, 50, a, b)  
show\_plots([answer\_2\_with\_Galerkin\_method], a, b, 0.01, "Galerkin method")

**Выводы**

В данной лабораторной работе мной были изучены методы коллокаций, наименьших квадратов и Галёркина, составлены алгоритмы методов и программы их реализаций. Составлен алгоритм решения краевых задач указанными методами, применимыми для организации вычислений на ПЭВМ. Составлена программа решения краевых задач по разработанным алгоритмам. Получено численное решение заданной краевой задачи.