Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет

информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы численного анализа

**ОТЧЁТ**

к лабораторной работе

на тему

Решение краевых задач. Метод коллокаций, наименьших квадратов и Галеркина, стрельбы и разностных аппроксимаций

Выполнил: студентка группы 153501

Тимофеев Кирилл Андреевич

Проверил: Анисимов Владимир Яковлевич

Минск 2023

**СОДЕРЖАНИЕ**

1. ВВЕДЕНИЕ………………………………………………………………..3
2. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ………………………….....4
3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ……………………………………….8

3.1 Код программы………………………………………………………..8

3.2 Результаты…………………………………………………………….10

4. АНАЛИЗ ПОЛУЧЕННЫХ РЕЗУЛЬТАТОВ…………………………….13

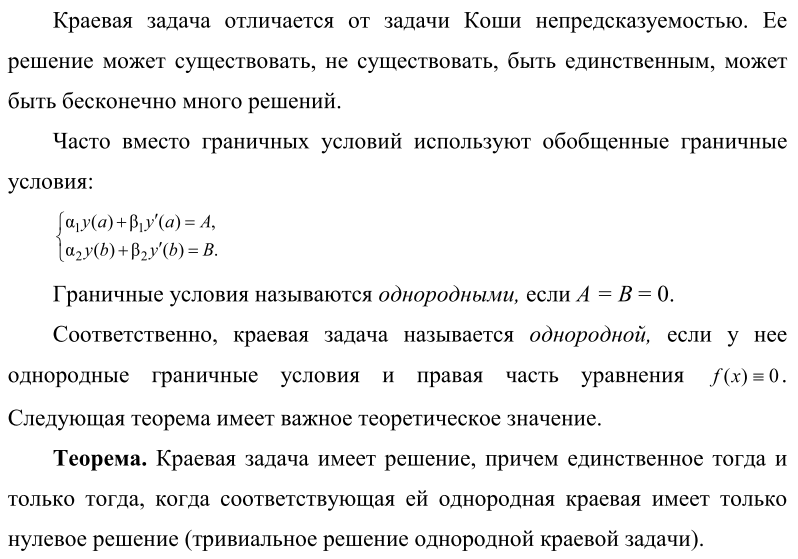
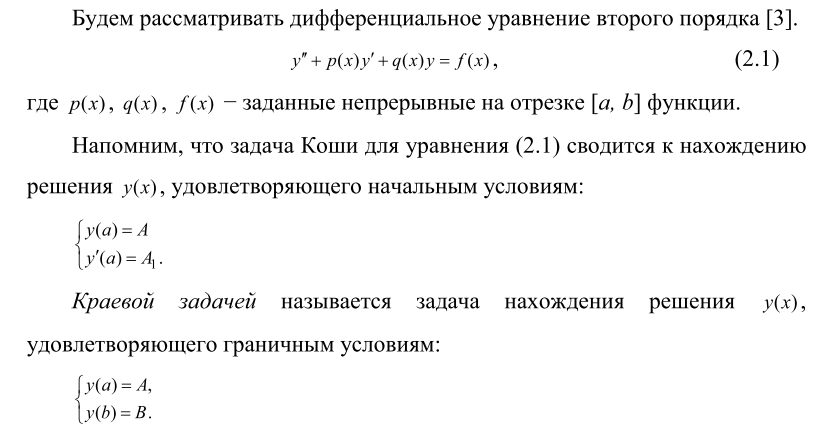
**1.** **ВВЕДЕНИЕ**

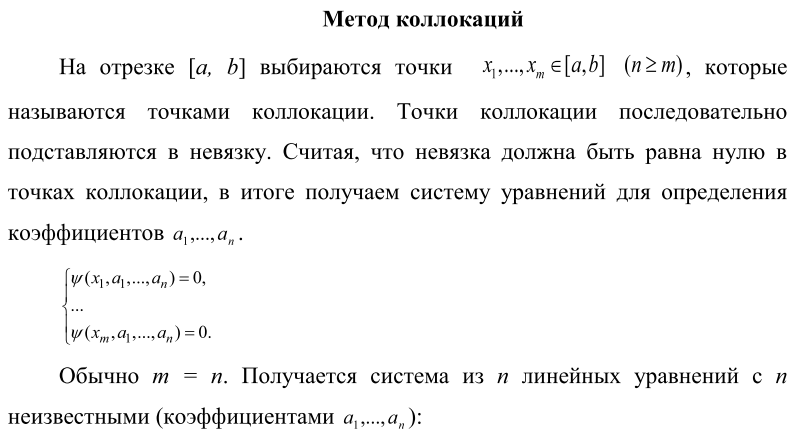
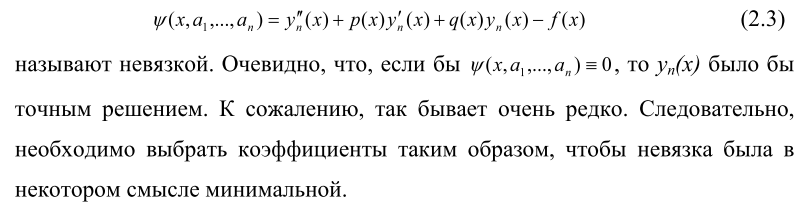
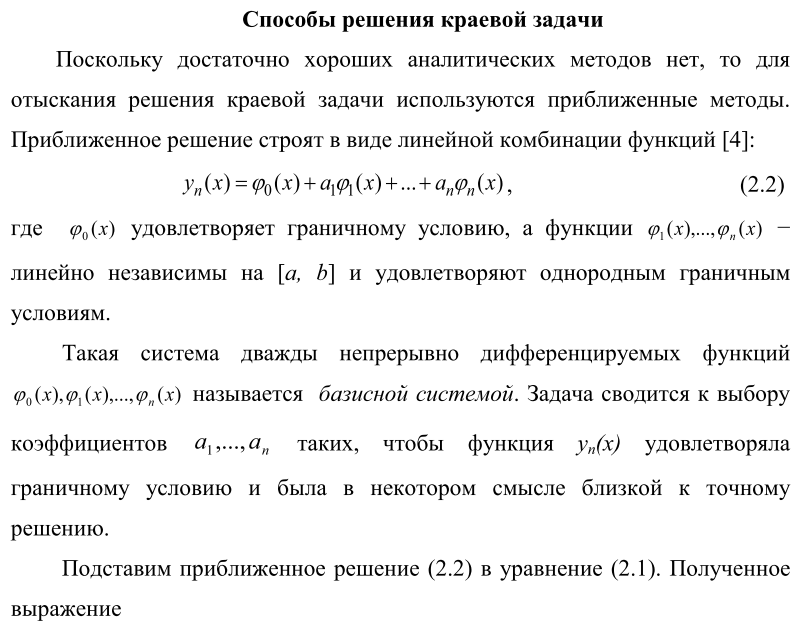
***Цели выполнения задания***:

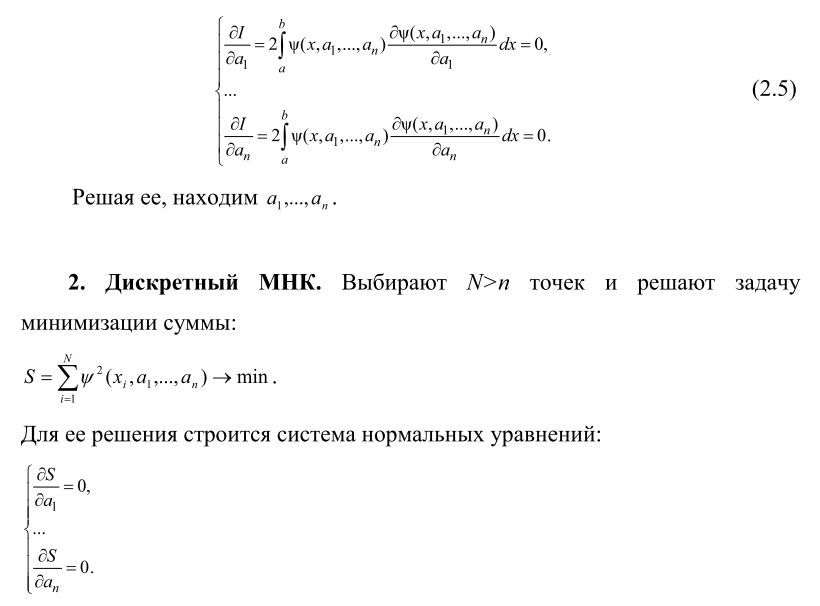
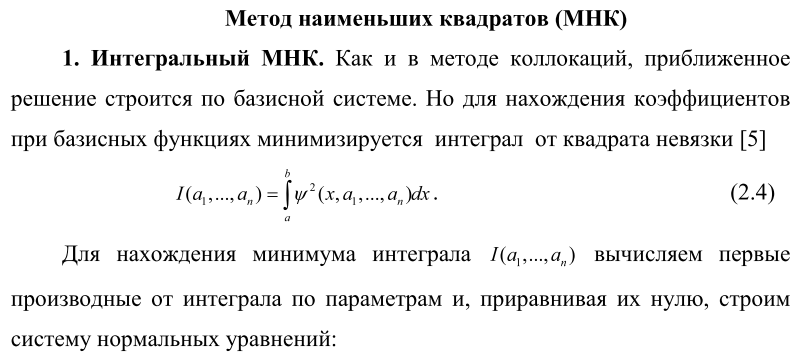
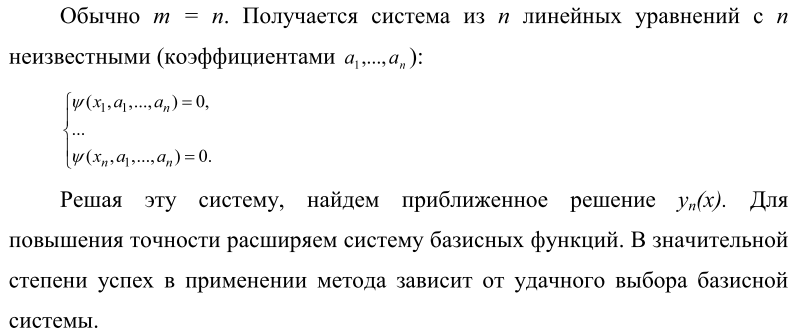
- изучить метод коллокаций, наименьших квадратов и Галеркина, стрельбы и разностных аппроксимаций;

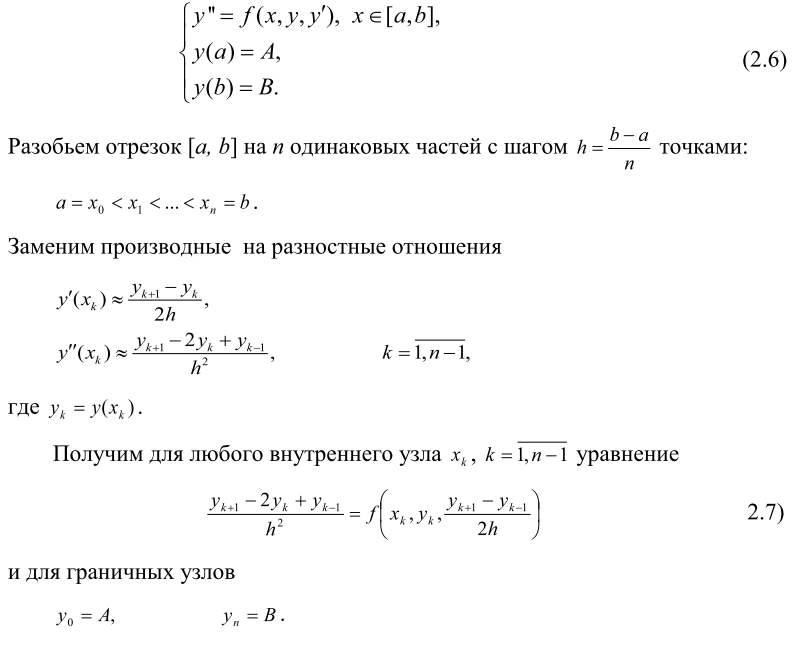
- написать программу для решения поставленных задач;

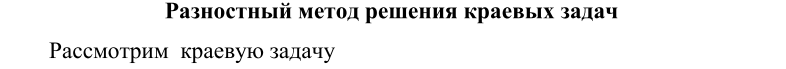
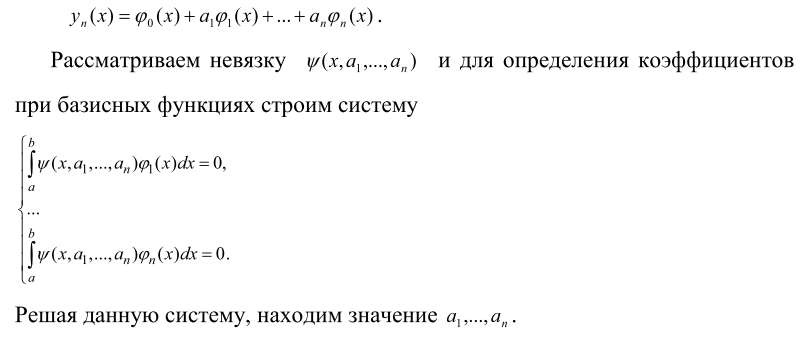
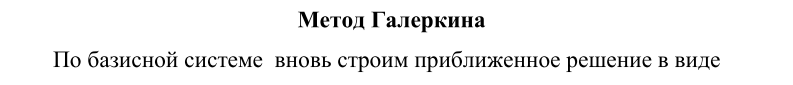
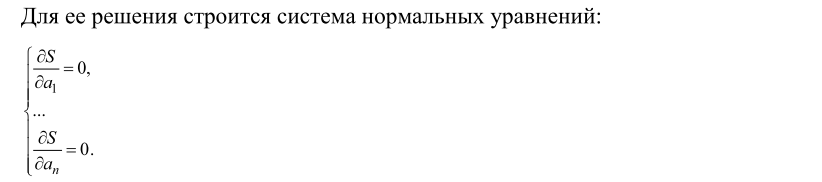
- решить тестовые примеры.

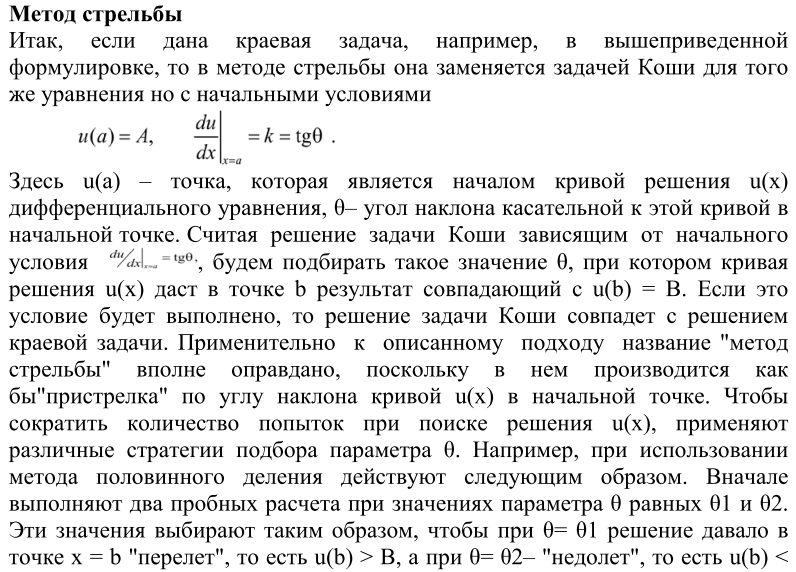
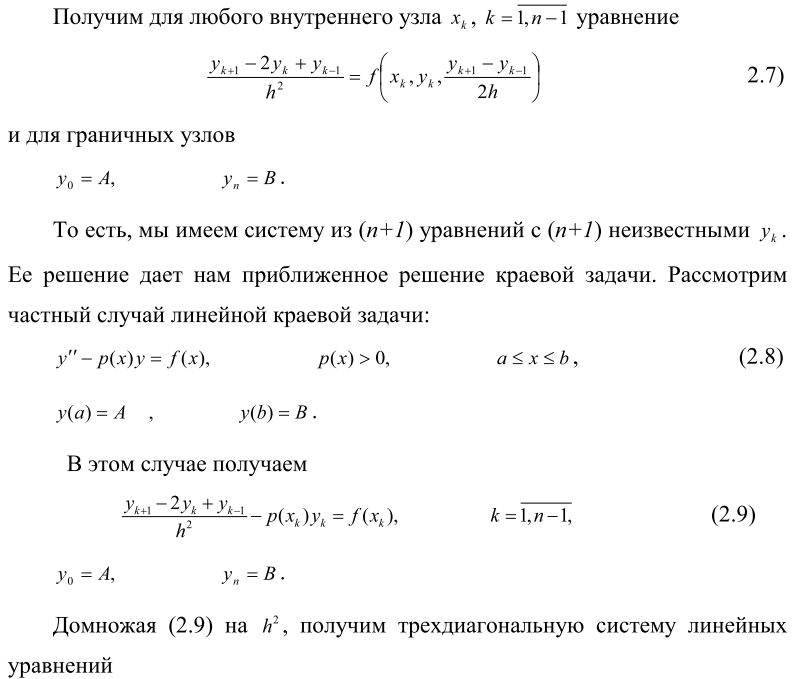
**2.** **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

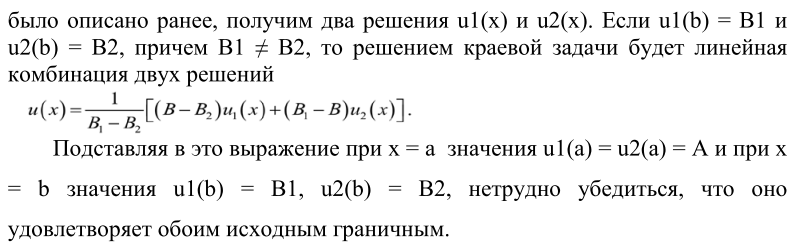
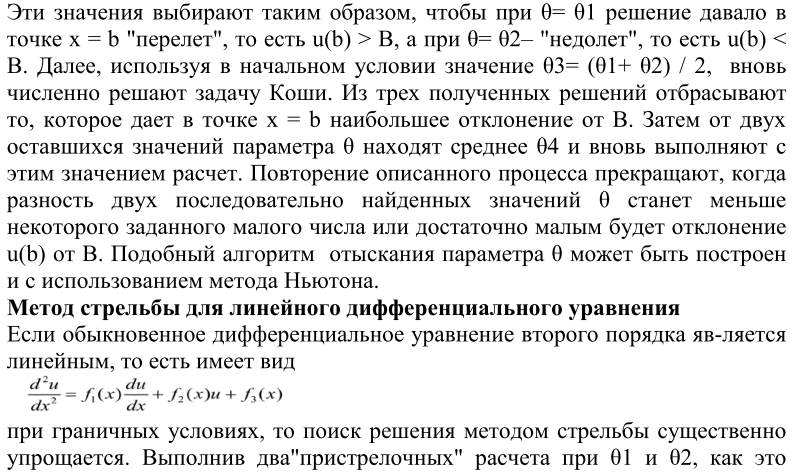












1. **ПРОГРАММНАЯ РЕАЛИЗАЦИЯ**
   1. **Код программы**

main.py

from collocation\_method import collocations\_method  
from galerkin\_method import galerkin\_method  
from min\_squares\_methods import integral\_least\_square\_method, discrete\_least\_square\_method  
from additional\_functions import generate\_basis\_sequence  
from constants import \*  
from difference\_method import task2, task3\_1, task3\_2, task5, task6, task7  
from shooting\_method import task4  
  
  
# 1.)  
print('collocations method: \n',  
 collocations\_method(generate\_basis\_sequence(3), [-1/2, 0, 1/2]))  
# print('collocations method: \n',  
# collocations\_method(generate\_basis\_sequence(3), [-1/2, 0, 1/2]))  
print('galerkin method: \n',  
 galerkin\_method(generate\_basis\_sequence(3), a, b))  
print('integral lsm method: \n',  
 integral\_least\_square\_method(generate\_basis\_sequence(N), a, b))  
print('discrete lsm method: \n',  
 discrete\_least\_square\_method(generate\_basis\_sequence(N), N + 1, a, b))  
  
# 2.)  
task2()  
  
# 3.)  
task3\_1()  
task3\_2()  
  
# 4.)  
task4()  
  
# 5.)  
task5()  
  
# 6.)  
task6()  
  
# 7.)  
task7()

collocation\_method.py

from sympy import linsolve  
from constants import x  
from additional\_functions import build\_function\_from\_basis, func\_for\_substantiation  
import sympy  
  
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

additional\_functions.py

from constants import \*  
import sympy  
  
def func\_for\_substantiation(subs):  
 func = sympy.sin(k) \* sympy.diff(subs, x, x) + ((1 + sympy.cos(k) \* x \*\* 2) \* subs + 1)  
 return func  
# def func\_for\_substantiation(subs):  
# func = sympy.diff(subs, x, x) + ((1 + 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

constants.py

import sympy as sp  
  
x = sp.Symbol('x')  
  
N = 3  
a = -1  
b = 1  
k = 2  
epsilon = 0.001

difference\_method.py

import math  
from abc import abstractmethod  
from math import \*  
from kostily import kostil1  
import matplotlib.pyplot as plt  
import numpy as np  
from kostily import kostil2  
  
class Metrics:  
 @staticmethod  
 def l\_1\_norm(first\_vector, second\_vector):  
 return np.sum(np.abs(first\_vector - second\_vector))  
  
 @staticmethod  
 def l\_2\_norm(first\_vector, second\_vector):  
 return np.sum((first\_vector - second\_vector) \*\* 2)  
  
 @staticmethod  
 def l\_infinity\_norm(first\_vector, second\_vector):  
 return np.max(np.abs(first\_vector - second\_vector))  
  
def solve\_tridigional\_matrix(upper\_diagonal, middle\_diagonal, lower\_diagonal, result\_vector):  
 c\_prime = np.zeros(middle\_diagonal.size - 1)  
 d\_prime = np.zeros(middle\_diagonal.size)  
  
 c\_prime[0] = upper\_diagonal[0] / middle\_diagonal[0]  
 for i in range(1, middle\_diagonal.size - 1):  
 c\_prime[i] = upper\_diagonal[i] / (  
 middle\_diagonal[i] - lower\_diagonal[i - 1] \* c\_prime[i - 1]  
 )  
  
 d\_prime[0] = result\_vector[0] / middle\_diagonal[0]  
 for i in range(1, middle\_diagonal.size):  
 d\_prime[i] = (result\_vector[i] - lower\_diagonal[i - 1] \* d\_prime[i - 1]) / (  
 middle\_diagonal[i] - lower\_diagonal[i - 1] \* c\_prime[i - 1]  
 )  
  
 solution = np.zeros(middle\_diagonal.size)  
 solution[-1] = d\_prime[-1]  
  
 for i in reversed(range(middle\_diagonal.size - 1)):  
 solution[i] = d\_prime[i] - c\_prime[i] \* solution[i + 1]  
 return solution  
  
class AbstractDifferenceSchemeSolver:  
 def *\_\_init\_\_*(  
 self,  
 p\_function,  
 q\_function,  
 f\_function,  
 left\_bound,  
 right\_bound,  
 h=0.1,  
 amount\_points=None,  
 left\_condition=None,  
 right\_condition=None,  
 ):  
 self.p = p\_function  
 self.q = q\_function  
 self.f = f\_function  
  
 self.left\_condition = left\_condition  
 self.right\_condition = right\_condition  
  
 self.left\_bound = left\_bound  
 self.right\_bound = right\_bound  
  
 if (  
 amount\_points is not None  
 and left\_condition is not None  
 and right\_condition is not None  
 ):  
 self.set\_h((right\_bound - left\_bound) / (amount\_points - 1))  
 elif h is not None:  
 self.set\_h(h)  
 else:  
 raise ValueError  
  
 def set\_h(self, new\_h):  
 self.h = new\_h  
 self.x\_points = np.arange(self.left\_bound, self.right\_bound + self.h, self.h)  
 self.y\_points = None  
  
 @abstractmethod  
 def \_get\_difference\_scheme\_vectors(self):  
 raise NotImplemented  
  
 def get\_accuracy(self, metrics):  
 if self.y\_points is None:  
 raise ValueError  
 double\_h\_solver = self.\_\_class\_\_(  
 self.p,  
 self.q,  
 self.f,  
 self.left\_bound,  
 self.right\_bound,  
 h=self.h \* 2.0,  
 left\_condition=self.left\_condition,  
 right\_condition=self.right\_condition,  
 )  
 double\_h\_solver.solve()  
 return metrics(  
 self.y\_points[::2],  
 double\_h\_solver.y\_points  
 if self.y\_points.size % 2 == 1  
 else double\_h\_solver.y\_points[:-1],  
 ) / (2 \*\* self.\_order - 1)  
  
 def get\_solution\_according\_accuracy(  
 self, metrics=Metrics.l\_infinity\_norm, eps=0.001, step=0.001  
 ):  
 current\_accuracy = self.get\_accuracy(metrics)  
 while current\_accuracy >= eps:  
 self.set\_h(self.h - step)  
 self.solve()  
 current\_accuracy = self.get\_accuracy(metrics)  
 return self.h, current\_accuracy  
  
 def solve(self):  
 self.y\_points = solve\_tridigional\_matrix(\*self.\_get\_difference\_scheme\_vectors())  
  
 def plot\_solution(self):  
 plt.plot(self.x\_points, self.y\_points, color='deeppink')  
  
  
class SecondOrderSchemeSolver(AbstractDifferenceSchemeSolver):  
 def *\_\_init\_\_*(  
 self,  
 p\_function,  
 q\_function,  
 f\_function,  
 left\_bound,  
 right\_bound,  
 h=0.1,  
 amount\_points=None,  
 left\_condition=None,  
 right\_condition=None,  
 ):  
 super().*\_\_init\_\_*(  
 p\_function,  
 q\_function,  
 f\_function,  
 left\_bound,  
 right\_bound,  
 h=h,  
 amount\_points=amount\_points,  
 left\_condition=left\_condition,  
 right\_condition=right\_condition,  
 )  
 self.\_order = 2  
  
 def \_get\_difference\_scheme\_vectors(self):  
 diagonal = np.ones(self.x\_points.size)  
 upper\_diagonal = np.zeros(self.x\_points.size - 1)  
 lower\_diagonal = np.zeros(self.x\_points.size - 1)  
  
 diagonal[1:-1] = np.vectorize(lambda x: (self.h \*\* 2) \* self.q(x) - 2)(  
 self.x\_points[1:-1]  
 )  
 upper\_diagonal[1:] = np.vectorize(lambda x: 1 + self.h \* self.p(x) / 2.0)(  
 self.x\_points[1:-1]  
 )  
 lower\_diagonal[:-1] = np.vectorize(lambda x: 1 - self.h \* self.p(x) / 2.0)(  
 self.x\_points[1:-1]  
 )  
  
 result = (self.h \*\* 2) \* np.vectorize(self.f)(self.x\_points)  
 result[0] = (  
 self.left\_condition(self.h)  
 if callable(self.left\_condition)  
 else self.left\_condition  
 )  
 result[-1] = (  
 self.right\_condition(self.h)  
 if callable(self.right\_condition)  
 else self.right\_condition  
 )  
 return upper\_diagonal, diagonal, lower\_diagonal, result  
  
class SecondOrderSchemeSolverBoundaryConditions(SecondOrderSchemeSolver):  
 def *\_\_init\_\_*(  
 self,  
 p\_function,  
 q\_function,  
 f\_function,  
 left\_bound,  
 right\_bound,  
 h=0.1,  
 amount\_points=None,  
 left\_condition=None,  
 right\_condition=None,  
 ):  
 super().\_\_init\_\_(  
 p\_function,  
 q\_function,  
 f\_function,  
 left\_bound,  
 right\_bound,  
 h=h,  
 amount\_points=amount\_points,  
 left\_condition=left\_condition[0],  
 right\_condition=right\_condition[0],  
 )  
 self.left\_condition\_function = left\_condition[1]  
 self.left\_condition\_derivative\_function = left\_condition[2]  
  
 self.right\_condition\_function = right\_condition[1]  
 self.right\_condition\_derivative\_function = right\_condition[2]  
  
 def get\_accuracy(self, metrics):  
 if self.y\_points is None:  
 raise ValueError  
 left\_condition = (  
 self.left\_condition,  
 self.left\_condition\_function,  
 self.left\_condition\_derivative\_function,  
 )  
 right\_condition = (  
 self.right\_condition,  
 self.right\_condition\_function,  
 self.right\_condition\_derivative\_function,  
 )  
  
 double\_h\_solver = self.\_\_class\_\_(  
 self.p,  
 self.q,  
 self.f,  
 self.left\_bound,  
 self.right\_bound,  
 h=self.h \* 2.0,  
 left\_condition=left\_condition,  
 right\_condition=right\_condition,  
 )  
 double\_h\_solver.solve()  
 return metrics(  
 self.y\_points[::2],  
 double\_h\_solver.y\_points  
 if self.y\_points.size % 2 == 1  
 else double\_h\_solver.y\_points[:-1],  
 ) / (2 \*\* self.\_order - 1)  
  
 def \_get\_difference\_scheme\_vectors(self):  
 (  
 upper\_diagonal,  
 diagonal,  
 lower\_diagonal,  
 result,  
 ) = super().\_get\_difference\_scheme\_vectors()  
  
 diagonal[0] = (  
 (-3 / 2.0 + lower\_diagonal[0] / (2 \* upper\_diagonal[1]))  
 \* self.left\_condition\_derivative\_function  
 + self.left\_condition\_function \* self.h  
 )  
 upper\_diagonal[0] = (  
 2 + diagonal[1] / (2 \* upper\_diagonal[1])  
 ) \* self.left\_condition\_derivative\_function  
 result[0] += (  
 result[1]  
 \* self.left\_condition\_derivative\_function  
 / (2 \* upper\_diagonal[1])  
 )  
  
 lower\_diagonal[-1] = (  
 -2 - diagonal[-2] / (2.0 \* lower\_diagonal[-2])  
 ) \* self.right\_condition\_derivative\_function  
 diagonal[-1] = (  
 (3 / 2 - upper\_diagonal[-1] / (2.0 \* lower\_diagonal[-2]))  
 \* self.right\_condition\_derivative\_function  
 + self.right\_condition\_function \* self.h  
 )  
 result[-1] -= (  
 result[-2] / (2.0 \* lower\_diagonal[-2])  
 ) \* self.right\_condition\_derivative\_function  
  
 return upper\_diagonal, diagonal, lower\_diagonal, result  
  
def task2():  
 a = sin(2)  
 b = cos(2)  
 A = 0  
 B = 0  
 eps = 0.001  
 # h = 0.07799999999999963  
 # h = 0.1  
 h = kostil1()  
 solver = SecondOrderSchemeSolver(  
 lambda x: 0,  
 lambda x: (1 + b \* x \*\* 2) / a,  
 lambda x: -1 / a,  
 -1,  
 1,  
 h,  
 left\_condition=A,  
 right\_condition=B,  
 )  
 solver.solve()  
 solver.plot\_solution()  
 solver.get\_accuracy(Metrics.l\_infinity\_norm)  
  
 # h, error = solver.get\_solution\_according\_accuracy(Metrics.l\_infinity\_norm)  
 # print(f"the step at which accuracy {eps} is achieved, equals {h=}, and {error=}")  
 print(f"\nthe step at which accuracy {eps} is achieved, equals {h=}")  
 plt.title(f"h = {h}\neps = {eps}")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 # solver.plot\_solution(mode="smooth")  
 plt.show()  
 plt.title(f"difference with galerkin method")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 solver.plot\_solution()  
 x = np.linspace(-1, 1, 1000)  
 plt.plot(x, .963545669335502 \* (-(x \*\* 2) + 1) - .102864438561445 \* (x \*\* 2) \* (-(x \*\* 2) + 1))  
 plt.show()  
  
  
def task3\_1():  
 p = lambda x: -x  
 q = lambda x: 2  
 f = lambda x: x + 1  
  
 left\_bound = 0.9  
 right\_bound = 1.2  
  
 left\_condition = (lambda h: 2 \* h, 1, -0.5)  
 right\_condition = (lambda h: h, 1, 0)  
 eps = 0.001  
  
 solver = SecondOrderSchemeSolverBoundaryConditions(  
 p,  
 q,  
 f,  
 left\_bound,  
 right\_bound,  
 h=0.1,  
 left\_condition=left\_condition,  
 right\_condition=right\_condition,  
 )  
 solver.solve()  
 h, error = solver.get\_solution\_according\_accuracy(Metrics.l\_infinity\_norm, eps=eps)  
 print(f"the step at which accuracy {eps} is achieved {h=}")  
 plt.title(f"h = {h}\neps = {eps}")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 solver.plot\_solution()  
 plt.show()  
  
 # kostil2()  
  
def task3\_2():  
 p\_test = lambda x: -x / (x - 1)  
 q\_test = lambda x: 1 / (x - 1)  
 f\_test = lambda x: (x - 1)  
 left\_bound = 2  
 left\_condition = (lambda h: h \* 2, 0, 1)  
 right\_bound = 4  
 right\_condition = (lambda h: h \* 5, 2, 1)  
 analytic\_solution = lambda x: -0.072 \* exp(x) + 7.532 \* x - x \*\* 2 - x - 1  
 solver = SecondOrderSchemeSolverBoundaryConditions(  
 p\_test,  
 q\_test,  
 f\_test,  
 left\_bound,  
 right\_bound,  
 amount\_points=100,  
 left\_condition=left\_condition,  
 right\_condition=right\_condition,  
 )  
 solver.solve()  
 plt.title("test")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 x = np.linspace(2, 4, 1000)  
 plt.plot(x, np.vectorize(analytic\_solution)(x), color='black')  
 solver.plot\_solution()  
 plt.show()  
  
  
def task5():  
 A = 0  
 B = 5  
 eps = 0.05  
 h = 0.039  
 solver = SecondOrderSchemeSolver(  
 lambda x: -2\*x,  
 lambda x: -5\*(2+math.sin(2\*x))/math.exp(-(x\*\*2)),  
 lambda x: -(math.exp(x)\*(1+math.sin(2\*x)))/math.exp(-(x\*\*2)),  
 0,  
 2,  
 h,  
 left\_condition=A,  
 right\_condition=B,  
 )  
 solver.solve()  
 solver.plot\_solution()  
 solver.get\_accuracy(Metrics.l\_infinity\_norm)  
  
 # h, error = solver.get\_solution\_according\_accuracy(Metrics.l\_infinity\_norm)  
 # print(f"the step at which accuracy {eps} is achieved, equals {h=}, and {error=}")  
 print(f"\nthe step at which accuracy {eps} is achieved, equals {h=}")  
 plt.title(f"h = {h}\neps = {eps}")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 solver.plot\_solution()  
 plt.show()  
  
  
def task6():  
 p = lambda x: 1  
 q = lambda x: 2\*x  
 f = lambda x: x\*\*2 + 1  
  
 left\_bound = 0.3  
 right\_bound = 2.7  
  
 left\_condition = (lambda h: 3 \* h, 1, 0.5)  
 right\_condition = (lambda h: h, 1, 0)  
 eps = 0.03  
  
 solver = SecondOrderSchemeSolverBoundaryConditions(  
 p,  
 q,  
 f,  
 left\_bound,  
 right\_bound,  
 h=0.1,  
 left\_condition=left\_condition,  
 right\_condition=right\_condition,  
 )  
 solver.solve()  
 h, error = solver.get\_solution\_according\_accuracy(Metrics.l\_infinity\_norm, eps=eps)  
 print(f"the step at which accuracy {eps} is achieved {h=}")  
 plt.title(f"h = {h}\neps = {eps}")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 solver.plot\_solution()  
 plt.show()  
  
  
def task7():  
 p = lambda x: 0  
 q = lambda x: -3.2/0.4  
 f = lambda x: 8\*x\*(x\*\*2-2)/0.4  
  
 left\_bound = 0  
 right\_bound = 1.275  
  
 left\_condition = (0, 0.5, -0.4)  
 right\_condition = (0, 0.5, 0.4)  
 eps = 0.0001  
  
 solver = SecondOrderSchemeSolverBoundaryConditions(  
 p,  
 q,  
 f,  
 left\_bound,  
 right\_bound,  
 h=0.1,  
 left\_condition=left\_condition,  
 right\_condition=right\_condition,  
 )  
 solver.solve()  
 h, error = solver.get\_solution\_according\_accuracy(Metrics.l\_infinity\_norm, eps=eps)  
 print(f"the step at which accuracy {eps} is achieved {h=}")  
 plt.title(f"h = {h}\neps = {eps}")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 solver.plot\_solution()  
  
 p = lambda x: 0  
 q = lambda x: -12 / 0.4  
 f = lambda x: 8 \* x \* (x \*\* 2 - 2) / 1.4  
  
 left\_bound = 1.275  
 right\_bound = 1.8  
  
 left\_condition = (0, 0.5, -1.4)  
 right\_condition = (0, 0.5, 1.4)  
 eps = 0.0001  
  
 solver = SecondOrderSchemeSolverBoundaryConditions(  
 p,  
 q,  
 f,  
 left\_bound,  
 right\_bound,  
 h=0.1,  
 left\_condition=left\_condition,  
 right\_condition=right\_condition,  
 )  
 solver.solve()  
 h, error = solver.get\_solution\_according\_accuracy(Metrics.l\_infinity\_norm, eps=eps)  
 print(f"the step at which accuracy {eps} is achieved {h=}")  
 plt.title(f"h = {h}\neps = {eps}")  
 plt.xlabel("x")  
 plt.ylabel("y")  
 solver.plot\_solution()  
  
 plt.show()

galerkin\_method.py

from sympy import linsolve  
from constants import \*  
from additional\_functions import build\_function\_from\_basis, func\_for\_substantiation  
import sympy  
  
  
def galerkin\_method(basis, A, B):  
 func = build\_function\_from\_basis(basis)  
 # print(f"Functions from basics: {func}")  
 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

kostily.py

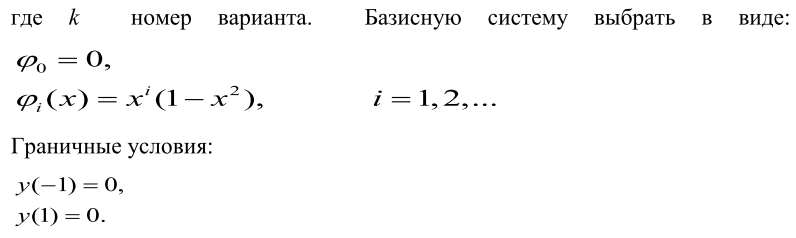
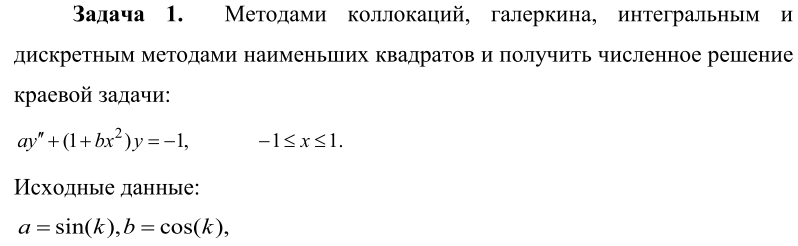
import math  
import numpy as np  
import matplotlib.pyplot as plt  
  
def sweep\_method(a, b, c, d):  
 AlphaS = [-c[0] / b[0]]  
 BetaS = [d[0] / b[0]]  
 GammaS = [b[0]]  
 n = len(d)  
 result = [0 for i in range(n)]  
 for i in range(1, n - 1):  
 GammaS.append(b[i] + a[i] \* AlphaS[i - 1])  
 AlphaS.append(-c[i] / GammaS[i])  
 BetaS.append((d[i] - a[i] \* BetaS[i - 1]) / GammaS[i])  
 GammaS.append(b[n - 1] + a[n - 1] \* AlphaS[n - 2])  
 BetaS.append((d[n - 1] - a[n - 1] \* BetaS[n - 2]) / GammaS[n - 1])  
 result[n - 1] = BetaS[n - 1]  
 for i in reversed(range(n - 1)):  
 result[i] = AlphaS[i] \* result[i + 1] + BetaS[i]  
 return result  
  
  
def check\_eps(current, prev, eps):  
 eps\_t = max([math.fabs(current[i \* 2] - prev[i]) for i in range(len(prev))])  
 if eps\_t > eps:  
 return False  
 return True  
  
  
def deep\_copy(system):  
 return [item for item in system]  
  
  
def get\_h(a, b, n):  
 return (b - a) / n  
  
  
def kostil1():  
 iteration\_count = 0  
 prev = []  
 current = []  
 eps = 1e-3  
 h = 0  
 n = 32  
 A = -1  
 B = 1  
 first = 0  
 last = 0  
 while True:  
 h = get\_h(A, B, n)  
 points = list(np.linspace(A, B, n + 2))  
 del points[0]  
 del points[-1]  
 a = [1 for x in points]  
 b = [-(2 + (h \*\* 2) \* (1 + math.sin(2) \* (x \*\* 2)) / math.cos(2)) for x in points]  
 c = [1 for x in points]  
 d = [-(h \*\* 2) for x in points]  
 d[0] = d[0] - a[0] \* first  
 d[-1] = d[-1] - c[-1] \* last  
 current = sweep\_method(a, b, c, d)  
 if iteration\_count != 0 and check\_eps(current, prev, eps):  
 break  
 prev = deep\_copy(current)  
 iteration\_count += 1  
 n \*= 2  
 # print("\nh = ", h)  
 return h  
  
  
def kostil2():  
 INTERVAL: tuple = (0.9, 1.2)  
  
 def get\_norm(values: list, step: float) -> float:  
 summ: float = 0.0  
  
 for value in values:  
 summ += value \* value \* step  
  
 return math.sqrt(summ)  
 return max(\*values[0:-1])  
  
 def get\_3\_task\_solution\_with\_number\_of\_subranges(n: int) -> tuple:  
 step: float = (INTERVAL[1] - INTERVAL[0]) / n  
  
 y\_kprev = lambda x: 2 + x \* step  
 y\_k = lambda x: 0.4 \* step \* step - 4  
 y\_knext = lambda x: 2 - x \* step  
  
 b\_k = lambda x: (x + 1) \* 2 \* step \* step # for free coefficients vector  
  
 # coefficients matrices with first initial condition already  
 free\_coefficients: list[float] = [4 \* step]  
 main\_coefficients: list[list[float]] = [[0] \* (n + 1)]  
 main\_coefficients[0][0] = 2 \* step + 1.5  
 main\_coefficients[0][1] = -2  
 main\_coefficients[0][2] = 0.5  
  
 x\_array: list[float] = [INTERVAL[0]]  
  
 for k in range(1, n):  
 new\_row: list[float] = [0] \* (n + 1)  
 new\_row[k - 1] = y\_kprev(x\_array[-1])  
 x\_array.append(x\_array[-1] + step)  
 new\_row[k] = y\_k(x\_array[-1])  
 new\_row[k + 1] = y\_knext(x\_array[-1] + step)  
  
 main\_coefficients.append(new\_row)  
 free\_coefficients.append(b\_k(x\_array[-1]))  
  
 x\_array.append(INTERVAL[1])  
  
 main\_coefficients.append([0] \* (n + 1))  
 main\_coefficients[-1][-1] = 1  
 free\_coefficients.append(1)  
  
 y\_array: list[float] = list(np.linalg.solve(main\_coefficients, free\_coefficients))  
  
 plt.plot(x\_array, y\_array, mew=2, ms=10)  
 plt.show()  
  
 return x\_array, y\_array, get\_norm(y\_array, step)  
  
 def get\_3\_task\_solution(epsilon: float) -> None:  
 number\_of\_ranges: int = 3  
  
 first\_solution = get\_3\_task\_solution\_with\_number\_of\_subranges(number\_of\_ranges)  
 number\_of\_ranges \*= 2  
 second\_solution = get\_3\_task\_solution\_with\_number\_of\_subranges(number\_of\_ranges)  
  
 while abs(first\_solution[2] - second\_solution[2]) > epsilon:  
 number\_of\_ranges \*= 2  
 first\_solution = second\_solution  
 second\_solution = get\_3\_task\_solution\_with\_number\_of\_subranges(number\_of\_ranges)  
  
 def solve\_task3() -> None:  
 epsilon: float = 0.001  
 get\_3\_task\_solution(epsilon)  
  
 solve\_task3()  
  
def kostyl3():  
 pass

min\_squares\_methods.py

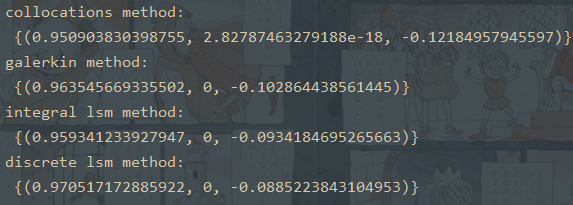
import functools  
from numpy import linspace  
from sympy import linsolve  
from constants import \*  
from additional\_functions import build\_function\_from\_basis, func\_for\_substantiation  
import sympy  
  
  
def integral\_least\_square\_method(basis, A, B):  
 func = build\_function\_from\_basis(basis)  
 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

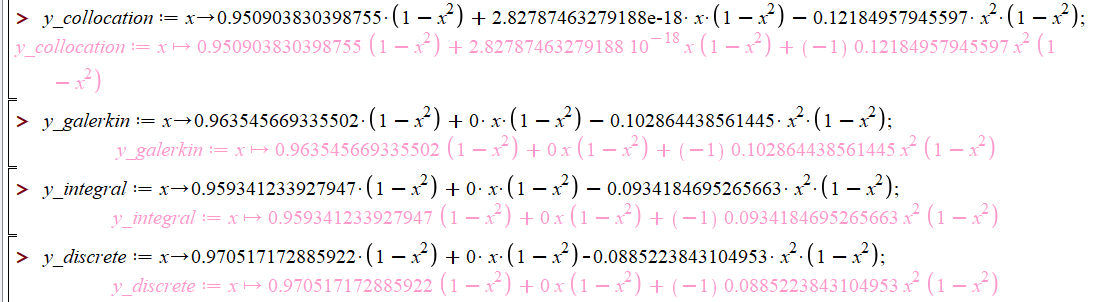
shooting\_method.py

import numpy as np  
from scipy.integrate import solve\_ivp  
import matplotlib.pyplot as plt  
  
def test():  
 x = np.linspace(0, 2 \* np.pi, 100)  
 y\_exact = 7 \* np.cos(np.sqrt(3) \* x) - 7 \* np.cos(2 \* np.pi \* np.sqrt(3)) / np.sin(2 \* np.pi \* np.sqrt(3)) \* np.sin(  
 np.sqrt(3) \* x)  
  
 def equations(x, y):  
 yprime = np.zeros(2)  
  
 yprime[0] = y[1]  
 yprime[1] = -3 \* y[0]  
  
 return yprime  
  
 tol = 1e-6  
 max\_iters = 100  
 low = -10  
 high = 10  
 count = 0  
  
 while count <= max\_iters:  
 count = count + 1  
 xspan = (x[0], x[-1])  
  
 # Use the midpoint between high and low as our guess  
 yprime0 = np.mean([low, high])  
  
 # Set the initial condition vector to be passed into the solver  
 y0 = [7, yprime0]  
  
 # Solve the system using our guess  
 sol = solve\_ivp(equations, xspan, y0, t\_eval=x)  
  
 # For ease of use, extract the function values from the solution object.  
 y\_num = sol.y[0, :]  
  
 # Check to see if we within our desire tolerance  
 if np.abs(y\_num[-1]) <= tol:  
 break  
  
 # Adjust our bounds if we are not within tolerance  
 if y\_num[-1] < 0:  
 high = yprime0  
 else:  
 low = yprime0  
  
 # print(count, y\_num[-1])  
  
 # Plot the solution and compare it to the analytical form defined above  
 plt.plot(x, y\_exact, 'k', label='Exact')  
 plt.plot(x, y\_num, 'b.', label='Numeric')  
 plt.plot([0, 2 \* np.pi], [7, 0], 'bo')  
 plt.grid(True)  
 plt.xlabel('x')  
 plt.ylabel('y')  
 plt.show()  
  
def task4():  
 test()

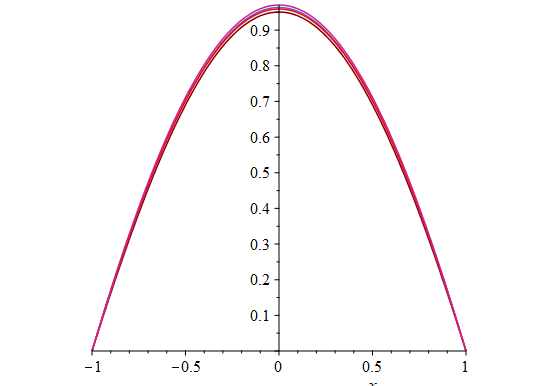
* 1. **Результаты (Вариант 2)**

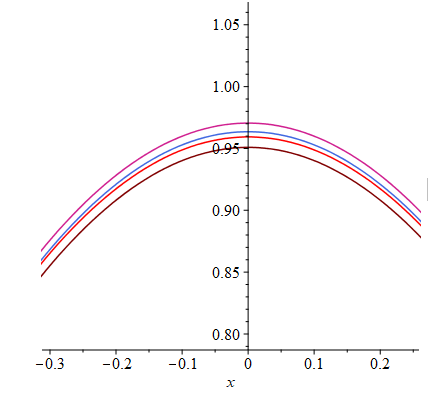
Значения коэффициентов в каждом методе при взятии трёх базисных функций:

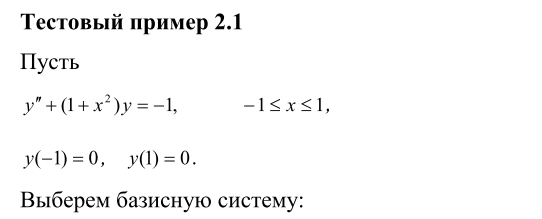


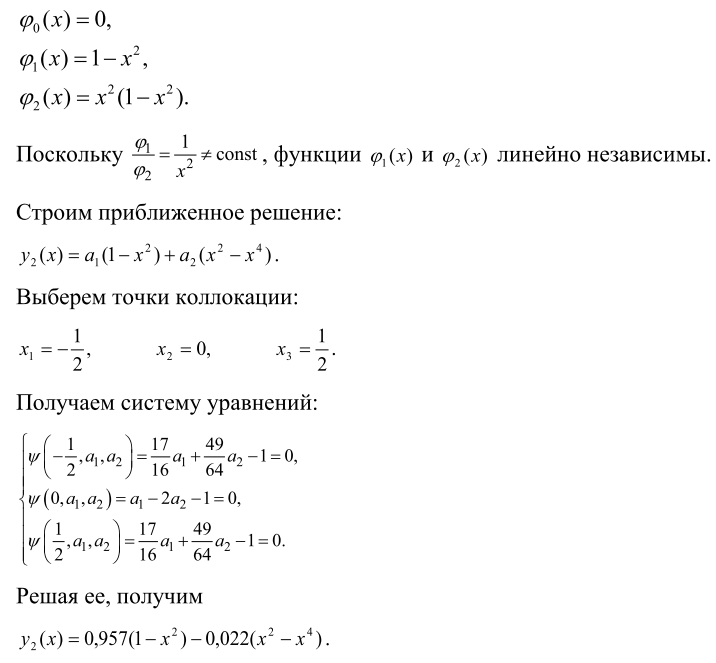
Полученные уравнения: 

Их графики в одной системе координат (метод коллокаций – коричневый цвет, метод галеркина – голубой цвет, интегральный МНК – красный цвет, дискретный МНК – фиолетовый цвет):

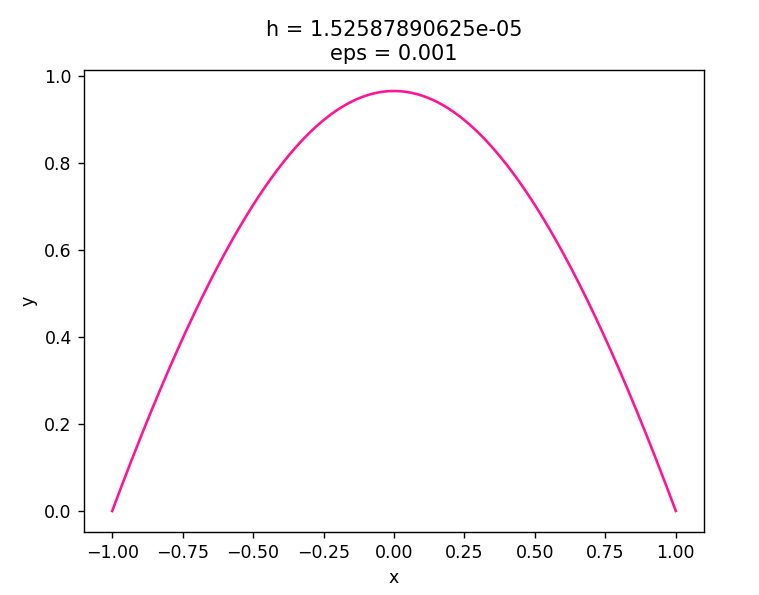
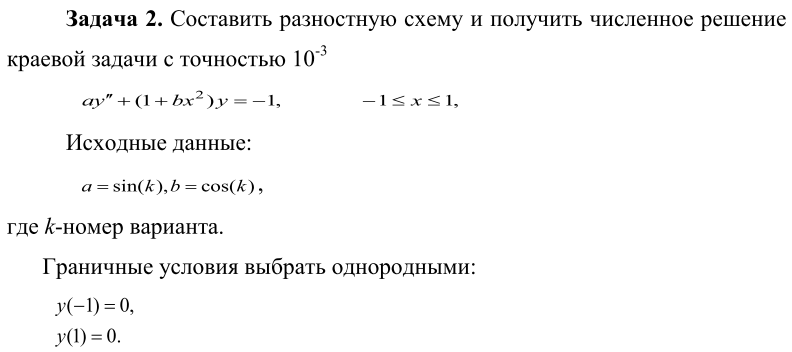
****

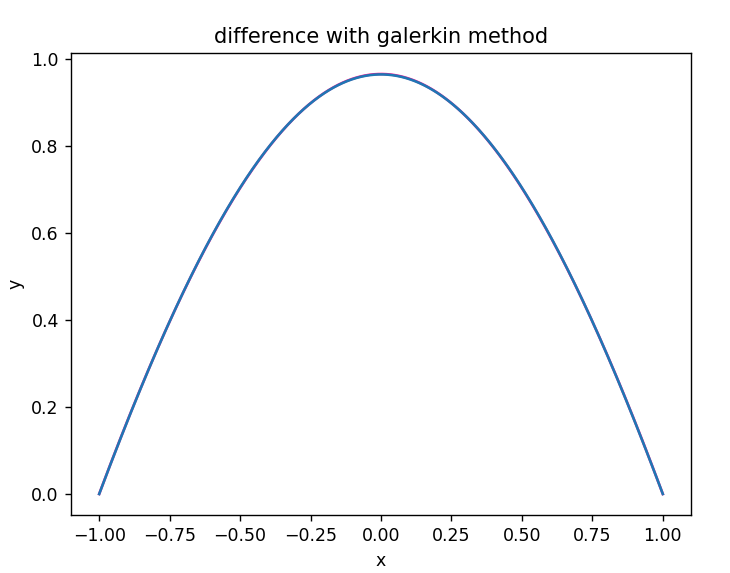
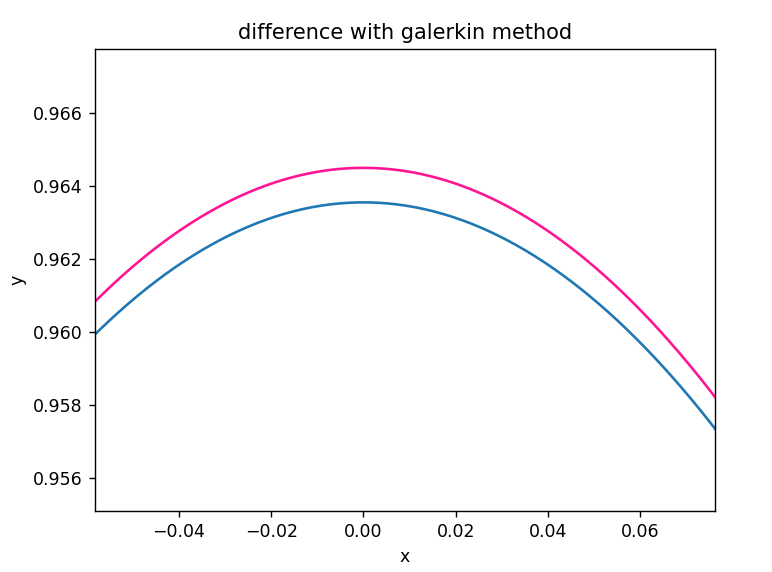


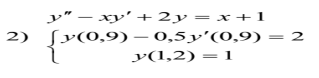
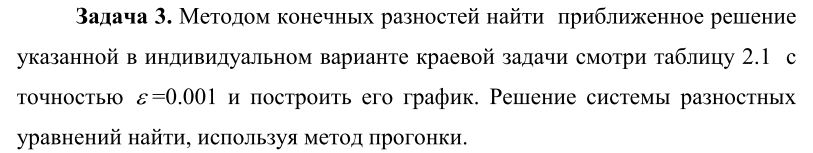
Возьмём тестовый пример из методического пособия:

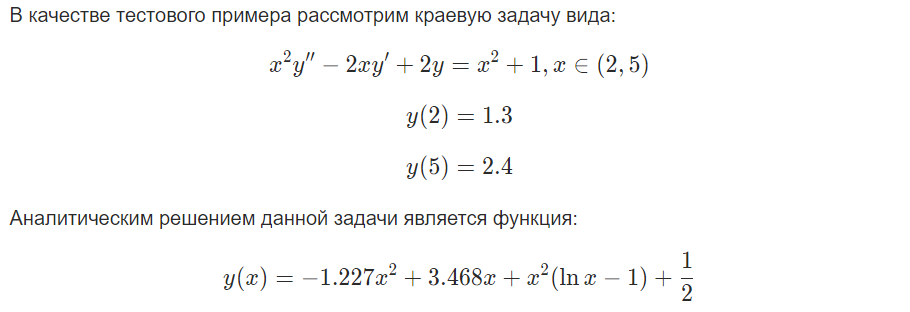


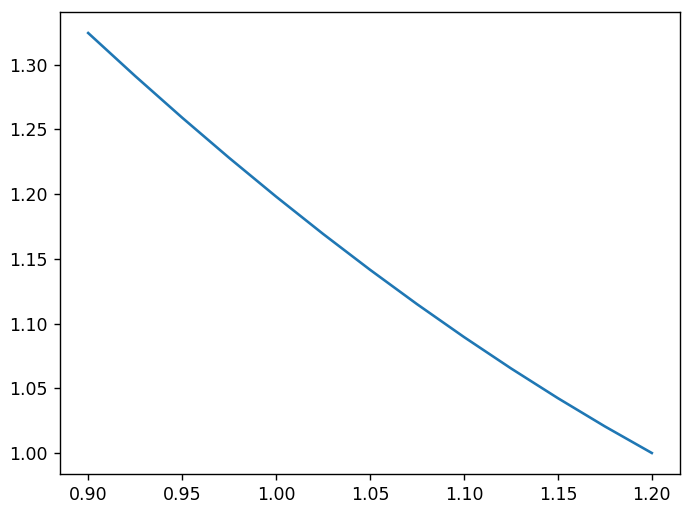


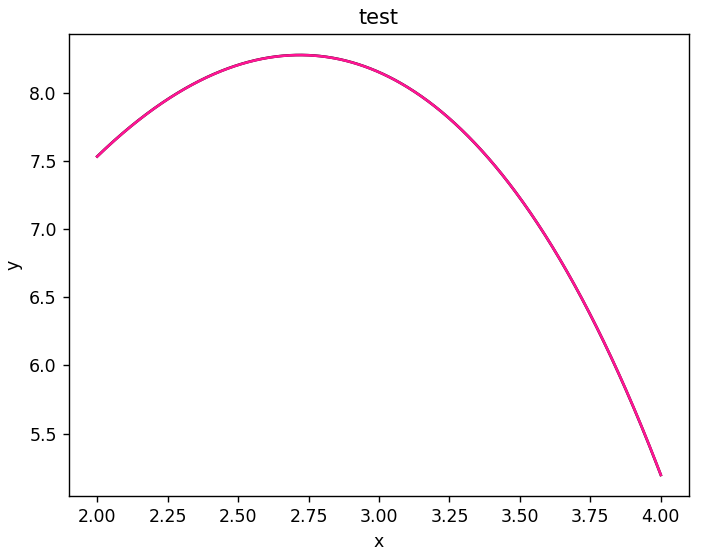


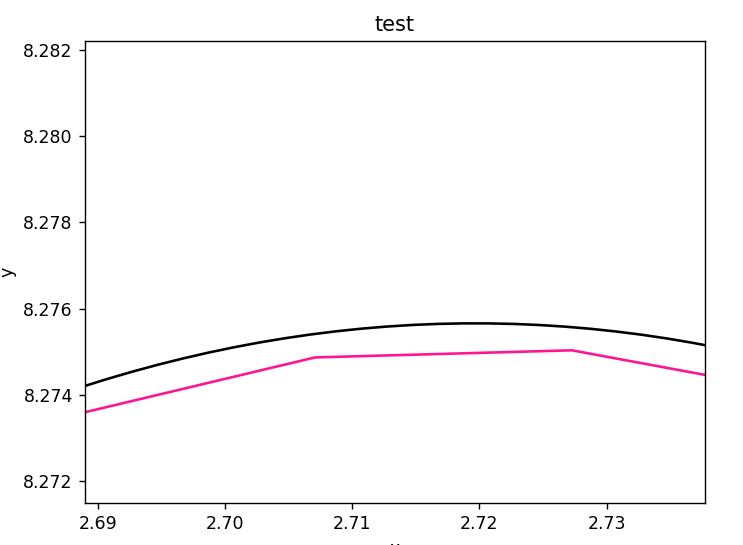
****



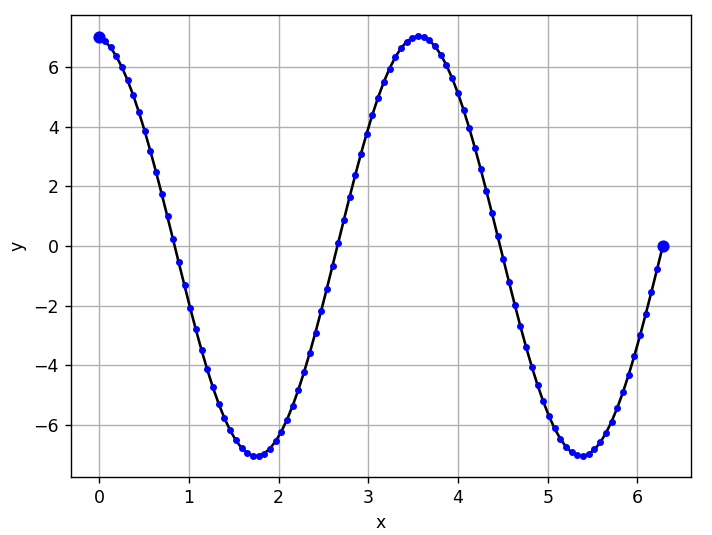
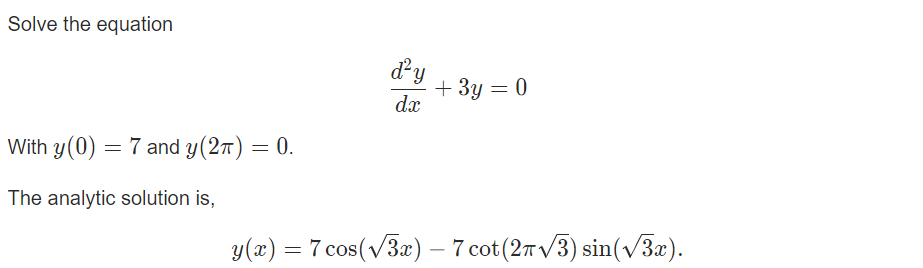
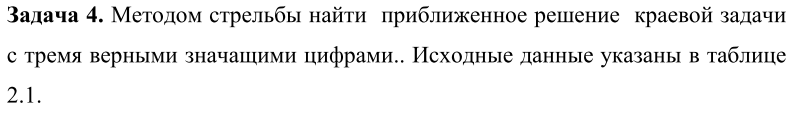


**-**

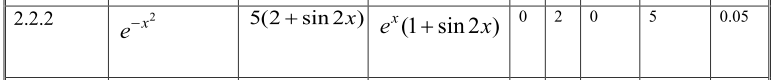
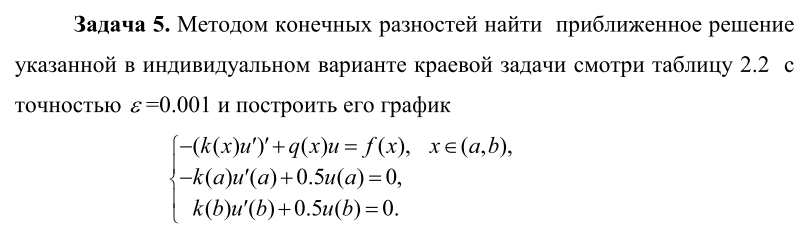


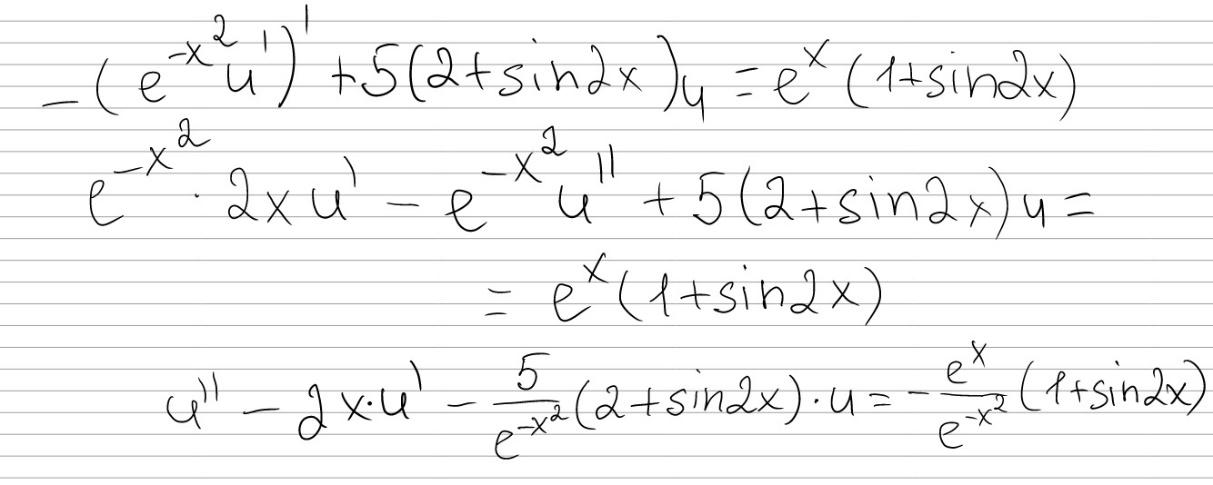


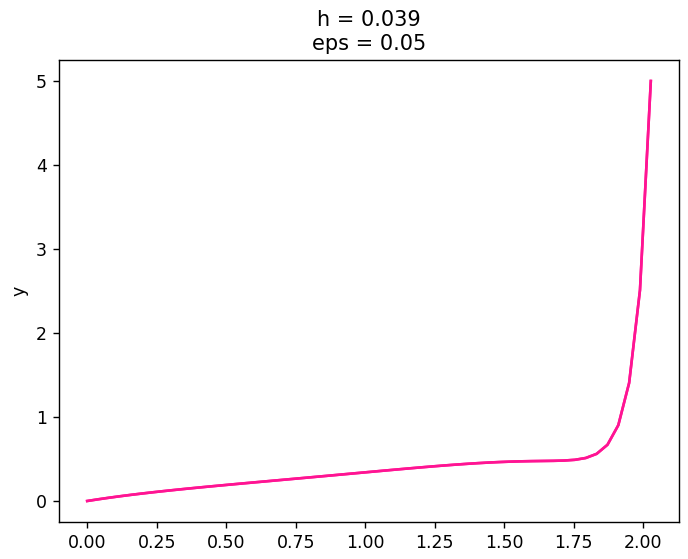
Аналитическое решение представлено чёрным цветом, а решение, полученное программой – розовым.



Чёрным цветом представлено аналитическое решений, а синим – точки, полученные методом стрельбы.

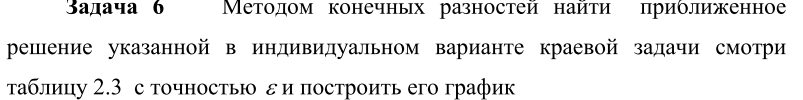
Для начала проведём пару вычислений для получения искомого по условии уравнения:

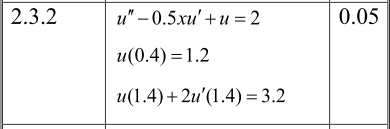


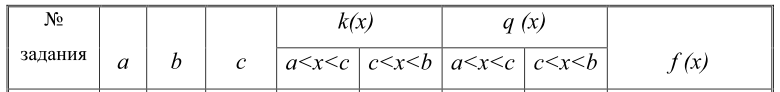
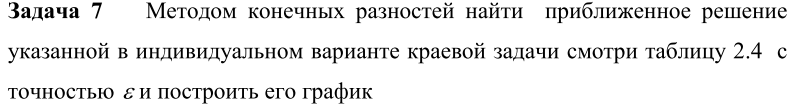


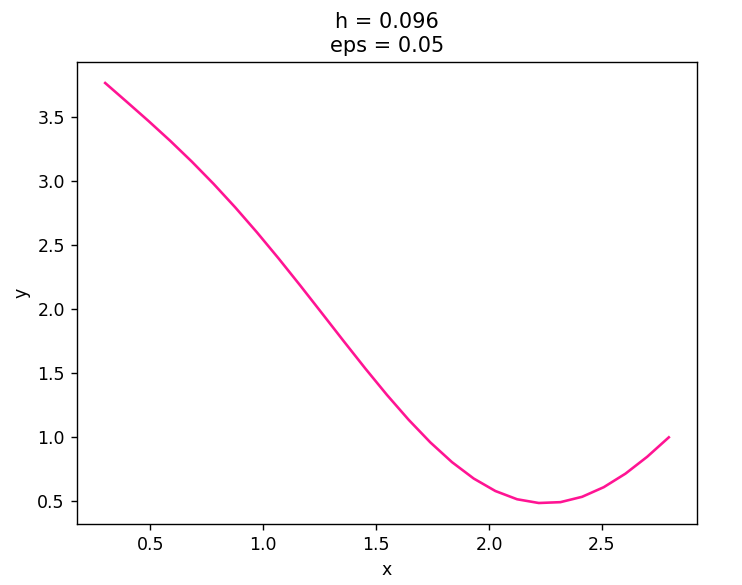
Итого получим следующие уравнение:











Заметим, что даже при таком маленьком шаге, краевые значения (особенно в месте разрыва) все равно имеют неточности, которые заметны невооружённым взглядом.

***Вывод:*** в данной лабораторной работе были изучены изучить метод коллокаций, наименьших квадратов и Галеркина, стрельбы и разностных аппроксимаций для решения краевых задач; была написана программа по данным методам и решены тестовые примеры.