**Московский авиационный институт**

**(Национальный исследовательский университет)**

**Отчет по лабораторным работам**

по курсу «Численные методы»

Студент: Фирфаров А. С.

Группа: 8О-308Б

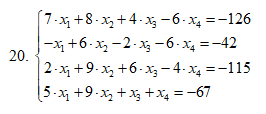
Москва, 2020

**Лабораторная работа 1.1**

1. **Задание ЛР**

Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

1. **Вариант: 20**



1. **Алгоритм**

В данной лабораторной работе используется алгоритм LU разложения матриц. LU – разложение матрицы **A** представляет собой разложение матрицы **A** в произведение нижней и верхней треугольных матриц. Задается исходная матрица и вектор правых частей. Далее производится разложение матрицы **A**. С помощью полученных матриц L и U решается система алгебраических уравнений. Далее с помощью LU разложения можно легко найти определитель и обратную матрицу.

1. **Код (Python)**

import json  
import argparse  
import sys  
import copy  
  
from utils import Matrix, Vector  
  
  
def get\_matrix\_permutations(matrix):  
 size = matrix.size  
 p = Matrix(size, single=True)  
 for i in range(0, size):  
 column = [matrix[j][i] for j in range(i, size)]  
 row\_idx = column.index(max(column, key=abs)) + i  
 if i != row\_idx:  
 p[i], p[row\_idx] = p[row\_idx], p[i]  
 return p  
  
  
def lup(matrix):  
 size = matrix.size  
  
 p = get\_matrix\_permutations(matrix)  
 pa = p \* matrix  
 l = Matrix(size, single=True)  
 u = copy.deepcopy(pa)  
  
 for k in range(0, size - 1):  
 m\_k = Matrix(size, single=True)  
 for i in range(k + 1, size):  
 mu\_i = -u[i][k] / u[k][k]  
  
 m\_k[i][k] = mu\_i  
 l[i][k] = -mu\_i  
 u = m\_k \* u  
 return l, u, p  
  
  
def lup\_solve(l, u, p, b):  
 size = b.size  
 n\_b = p \* b  
 z = Vector(size)  
 x = Vector(size)  
  
 z[0] = n\_b[0]  
 for i in range(1, size):  
 z[i] = n\_b[i] - sum([l[i][j] \* z[j] for j in range(0, i)])  
  
 x[size - 1] = z[size - 1] / u[size - 1][size - 1]  
 for i in range(size - 2, -1, -1):  
 x[i] = (z[i] - sum([u[i][j] \* x[j] for j in range(i + 1, size)])) / u[i][i]  
  
 return x  
  
  
def lup\_det(u, p):  
 det = 1.0  
 count\_perm = 0  
  
 for i in range(0, p.size):  
 if p[i][i] == 0:  
 count\_perm += 1  
 count\_perm = count\_perm - 1 if count\_perm > 0 else 0  
  
 det \*= (-1) \*\* count\_perm  
 for i in range(0, u.size):  
 det \*= u[i][i]  
 return det  
  
  
def inverse\_matrix(l, u, p):  
 size = l.size  
 inv = Matrix(size)  
  
 for i in range(0, size):  
 e = Vector(size)  
 e[i] = 1  
 column = lup\_solve(l, u, p, e)  
  
 for j in range(0, size):  
 inv[j][i] = column[j]  
 return inv  
  
  
def test(l, u, p, matrix, b, x, inv):  
 print('----------------Проверка----------------')  
 lu = l \* u  
 print('L \* U: ')  
 lu.print\_matrix()  
 pa = p \* matrix  
 print('P \* A: ')  
 pa.print\_matrix()  
 ax = matrix \* x  
 print('Ax: ')  
 ax.print\_vector()  
 print('B: ')  
 b.print\_vector()  
 print('A \* A^-1: ')  
 e = matrix \* inv  
 e.print\_matrix()  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 n = int(input('Размерность матрицы: '))  
 A = Matrix(n)  
 B = Vector(n)  
  
 A.matrix\_read\_file(args.input, 'matrix')  
 B.vector\_read\_file(args.input, 'vector')  
  
 L, U, P = lup(A)  
 X = lup\_solve(L, U, P, B)  
 det = lup\_det(U, P)  
 inv = inverse\_matrix(L, U, P)  
  
 for i in range(0, X.size):  
 print('x{0} = {1:5.2f}'.format(i + 1, X[i]))  
  
 print('\nОпределитель: {0:5.2f}'.format(det))  
 print('\nОбратная матрица ')  
 inv.print\_matrix()  
  
 test(L, U, P, A, B, X, inv)

1. **Файл с данными (data1.1)**

{  
 "matrix": [  
 [ 7, 8, 4, -6],  
 [ -1, 6, -2, -6],  
 [ 2, 9, 6, -4],  
 [ 5, 9, 1, 1]  
 ],  
 "vector": [-126, -42, -115, -67]  
}

1. **Результаты работы**

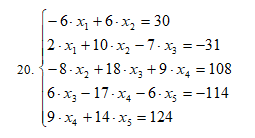
C:\Users\Admin\Desktop\Численные методы>python 1.1.py --input data1.1  
Размерность матрицы: 4  
x1 = -4.00  
x2 = -5.00  
x3 = -7.00  
x4 = 5.00  
  
Определитель: 2866.00  
  
Обратная матрица  
 0.14655 -0.05932 -0.12282 0.03210  
 -0.07048 0.05234 0.04955 0.08932  
 -0.00523 -0.10502 0.14724 -0.07258  
 -0.09316 -0.06943 0.02094 0.10816  
  
----------------Проверка----------------  
L \* U:  
 7.00000 8.00000 4.00000 -6.00000  
 2.00000 9.00000 6.00000 -4.00000  
 -1.00000 6.00000 -2.00000 -6.00000  
 5.00000 9.00000 1.00000 1.00000  
  
P \* A:  
 7.00000 8.00000 4.00000 -6.00000  
 2.00000 9.00000 6.00000 -4.00000  
 -1.00000 6.00000 -2.00000 -6.00000  
 5.00000 9.00000 1.00000 1.00000  
  
Ax:  
-126.00000 -42.00000 -115.00000 -67.00000  
B:  
-126.00000 -42.00000 -115.00000 -67.00000  
A \* A^-1:  
 1.00000 0.00000 -0.00000 0.00000  
 0.00000 1.00000 -0.00000 0.00000  
 0.00000 0.00000 1.00000 0.00000  
 -0.00000 -0.00000 -0.00000 1.00000

**Лабораторная работа 1.2**

1. **Задание ЛР**

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

1. **Вариант: 20**

****

1. **Алгоритм**

Метод прогонки является одним из эффективных методов решения СЛАУ с трех - диагональными матрицами. В программу подаются диагонали матрицы и вектор правых частей. Далее итерационно находим решение методом прогонки.

1. **Код (Python)**

import json  
import argparse  
import sys  
  
  
class TridiagonalMatrix:  
  
 def \_\_init\_\_(self, size):  
 self.A = []  
 self.B = []  
 self.C = []  
 self.size = size  
  
 def trimatrix\_read\_file(self, filename):  
 try:  
 with open(filename, 'r') as f:  
 data = json.load(f)  
 if len(data['A']) != self.size - 1 or len(data['C']) != self.size - 1 or len(data['B']) != self.size:  
 raise ValueError  
 self.A = [0.0] + list(map(float, data['A']))  
 self.B = list(map(float, data['B']))  
 self.C = list(map(float, data['C'])) + [0.0]  
 except Exception:  
 print('Некоректные входные данные')  
 sys.exit(1)  
  
  
class Vector:  
  
 def \_\_init\_\_(self, size):  
 self.size = size  
 self.vector = [0.0 for \_ in range(0, size)]  
  
 def \_\_getitem\_\_(self, idx):  
 return self.vector[idx]  
  
 def \_\_setitem\_\_(self, key, value):  
 self.vector[key] = value  
  
 def print\_vector(self):  
 for num in self.vector:  
 print('{0:5.2f} '.format(num), end='')  
 print()  
  
 def vector\_read\_file(self, filename, arg\_name):  
 try:  
 with open(filename, 'r') as f:  
 data = json.load(f)  
 if len(data[arg\_name]) != self.size:  
 raise ValueError  
 self.vector = list(map(float, data[arg\_name]))  
 except Exception:  
 print('Некоректные входные данные')  
 sys.exit(1)  
  
  
def rtm(m, d):  
 size = m.size  
 x = [0 for \_ in range(0, size)]  
 p, q = [], []  
  
 p.append(-m.C[0] / m.B[0])  
 q.append(d[0] / m.B[0])  
  
 for i in range(1, size):  
 p\_i = -m.C[i] / (m.B[i] + m.A[i] \* p[i - 1])  
 q\_i = (d[i] - m.A[i] \* q[i - 1]) / (m.B[i] + m.A[i] \* p[i - 1])  
  
 p.append(p\_i)  
 q.append(q\_i)  
  
 x[size - 1] = q[size - 1]  
 for i in range(size - 2, -1, -1):  
 x[i] = p[i] \* x[i + 1] + q[i]  
  
 return x  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 n = int(input('Размерность матрицы: '))  
 M = TridiagonalMatrix(n)  
 D = Vector(n)  
  
 M.trimatrix\_read\_file(args.input)  
 D.vector\_read\_file(args.input, 'D')  
  
 X = rtm(M, D)  
 for i in range(0, len(X)):  
 print('x{0} = {1:5.2f}'.format(i + 1, X[i]))

1. **Файл с данными (data1.2)**

{  
 "A": [2, -8, 6, 9],  
 "B": [-6, 10, 18, -17, 14],  
 "C": [6, -7, 9, -6],  
 "D": [30, -31, 108, -114, 124]  
}

1. **Результаты работы**

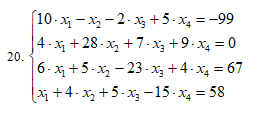
C:\Users\Admin\Desktop\Численные методы>python 1.2.py --input data1.2  
Размерность матрицы: 5  
x1 = -5.00  
x2 = 0.00  
x3 = 3.00  
x4 = 6.00  
x5 = 5.00

**Лабораторная работа 1.3**

1. **Задание ЛР**

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

1. **Вариант: 20**



1. **Алгоритм**

В программе реализован метод простых итераций и метод Зейделя. На вход программы подается матрица системы, вектор правых частей и точность, с которой нужно искать решение. Решение ищется в виде Начальное приближение Находим матрицу альфа и вектор бета, далее находим решение с помощью метода простых итераций и метода Зейделя. Метод Зейделя находит решение за меньшее число итераций.

1. **Код (Python)**

import argparse  
import copy  
import json  
  
from utils import Matrix, Vector  
  
  
def get\_alpha\_beta(a, b):  
 size = a.size  
 alpha = Matrix(size)  
 beta = Vector(size)  
  
 for i in range(0, size):  
 beta[i] = b[i] / a[i][i]  
 for j in range(0, size):  
 alpha[i][j] = -a[i][j] / a[i][i] if i != j else 0  
  
 return alpha, beta  
  
  
def method\_of\_simple\_iteration(alpha, beta, eps):  
 n\_iter = 0  
 x = copy.deepcopy(beta)  
 alpha\_norm = alpha.matrix\_norm()  
 k = alpha\_norm / (1 - alpha\_norm)  
  
 while True:  
 x\_prev = copy.deepcopy(x)  
 x = beta + alpha \* x  
 n\_iter += 1  
 if k \* (x - x\_prev).vector\_norm() < eps:  
 print('Итераций: ', n\_iter)  
 return x  
  
  
def seidel\_method(alpha, beta, eps):  
 n\_iter = 0  
 x = copy.deepcopy(beta)  
 alpha\_norm = alpha.matrix\_norm()  
 k = alpha\_norm / (1 - alpha\_norm)  
  
 while True:  
 x\_prev = copy.deepcopy(x)  
 for i in range(0, x.size):  
 x[i] = beta[i] + sum([alpha[i][j] \* x[j] for j in range(0, x.size)])  
 n\_iter += 1  
 if k \* (x - x\_prev).vector\_norm() < eps:  
 print('Итераций: ', n\_iter)  
 return x  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 while True:  
 method = int(input('Метод простых итераций - 1\nМетод Зейделя - 2\n'))  
 if method in (1, 2):  
 break  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 n = int(data['size'])  
 Eps = float(data['eps'])  
  
 A = Matrix(n)  
 B = Vector(n)  
  
 A.matrix\_read\_file(args.input, 'matrix')  
 B.vector\_read\_file(args.input, 'vector')  
  
 Alpha, Beta = get\_alpha\_beta(A, B)  
 print('Альфа:')  
 Alpha.print\_matrix()  
 print('Бета:')  
 Beta.print\_vector()  
 print('Норма матрицы Альфа: ', Alpha.matrix\_norm())  
 print()  
  
 X = Vector(n)  
 if method == 1:  
 X = method\_of\_simple\_iteration(Alpha, Beta, Eps)  
 if method == 2:  
 X = seidel\_method(Alpha, Beta, Eps)  
  
 for i in range(0, X.size):  
 print('x{0} = {1:8.5f}'.format(i + 1, X[i]))

1. **Файл с данными (data1.3)**

{  
 "matrix": [  
 [ 10, -1, -2, 5],  
 [ 4, 28, 7, 9],  
 [ 6, 5, -23, 4],  
 [ 1, 4, 5, -15]  
 ],  
 "vector": [-99, 0, 67, 58],  
 "size": 4,  
 "eps": 0.01  
}

1. **Результаты работы**

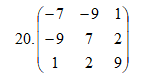
C:\Users\Admin\Desktop\Численные методы>python 1.3.py --input data1.3  
Метод простых итераций - 1  
Метод Зейделя - 2  
1  
Альфа:  
 0.00000 0.10000 0.20000 -0.50000  
 -0.14286 0.00000 -0.25000 -0.32143  
 0.26087 0.21739 0.00000 0.17391  
 0.06667 0.26667 0.33333 0.00000  
  
Бета:  
 -9.90000 0.00000 -2.91304 -3.86667  
Норма матрицы Альфа: 0.8  
  
Итераций: 10  
x1 = -8.00082  
x2 = 3.99861  
x3 = -4.99956  
x4 = -4.99950  
  
C:\Users\Admin\Desktop\Численные методы>python 1.3.py --input data1.3  
Метод простых итераций - 1  
Метод Зейделя - 2  
2  
Альфа:  
 0.00000 0.10000 0.20000 -0.50000  
 -0.14286 0.00000 -0.25000 -0.32143  
 0.26087 0.21739 0.00000 0.17391  
 0.06667 0.26667 0.33333 0.00000  
  
Бета:  
 -9.90000 0.00000 -2.91304 -3.86667  
Норма матрицы Альфа: 0.8  
  
Итераций: 5  
x1 = -8.00001  
x2 = 4.00007  
x3 = -5.00001  
x4 = -4.99998

**Лабораторная работа 1.4**

1. **Задание ЛР**

Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

1. **Вариант: 20**

****

1. **Алгоритм**

Метод вращений Якоби применим только для симметрических матриц и решает полную проблему собственных значений и собственных векторов таких матриц. На вход программы подается матрица и точность вычислений. На каждой итерации выбираем максимальный по модулю внедиагональный элемент матрицы. Далее находим соответствующую этому элементу матрицу вращения U и получаем новую матрицу В качестве критерия окончания итерационного процесса используется условие малости суммы квадратов вне диагональных элементов. Матрица собственных векторов V ищется одновременно с матрицей собственных значений. Изначально задает V единичной матрицей, далее на каждой итерации

1. **Код (Python)**

import argparse  
import copy  
import json  
import math  
  
from utils import Matrix, Vector  
  
  
def t(a):  
 sum\_el = 0.0  
 for j in range(1, a.size):  
 for i in range(0, j):  
 sum\_el += a[i][j] \*\* 2  
 return math.sqrt(sum\_el)  
  
  
def find\_max(a):  
 i\_max, j\_max = 0, 1  
 for j in range(1, a.size):  
 for i in range(0, j):  
 if abs(a[i][j]) > abs(a[i\_max][j\_max]):  
 i\_max, j\_max = i, j  
 return i\_max, j\_max  
  
  
def method\_jacobi\_rotations(a, eps):  
 n\_iter = 0  
 a\_k = copy.copy(a)  
 v = Matrix(a.size, single=True)  
  
 while True:  
 i\_max, j\_max = find\_max(a\_k)  
 fi = 0.5 \* math.atan(2 \* a\_k[i\_max][j\_max] / (a\_k[i\_max][i\_max] - a\_k[j\_max][j\_max]))  
  
 u = Matrix(a.size, single=True)  
 u[i\_max][i\_max] = math.cos(fi)  
 u[i\_max][j\_max] = -math.sin(fi)  
 u[j\_max][i\_max] = math.sin(fi)  
 u[j\_max][j\_max] = math.cos(fi)  
  
 u\_t = copy.copy(u)  
 u\_t.transpose()  
  
 a\_k = u\_t \* a\_k \* u  
 v = v \* u  
  
 n\_iter += 1  
  
 if t(a\_k) < eps:  
 eigenvalue = Vector(a\_k.size)  
 eigenvalue.vector = [a\_k[i][i] for i in range(0, a\_k.size)]  
  
 print('Итераций: ', n\_iter)  
 return eigenvalue, v  
  
  
def test(eigenvalue, v\_matrix, matrix):  
 print('Проверка')  
 vectors = []  
  
 for vector in zip(\*v\_matrix.matrix):  
 v = Vector(v\_matrix.size)  
 v.vector = vector  
 vectors.append(v)  
  
 for i in range(0, len(vectors) - 1):  
 print('v\_{0}: '.format(i + 1), end='')  
 vectors[i].print\_vector()  
 print('v\_{0}: '.format(len(vectors)), end='')  
 v.print\_vector()  
 print('(v\_{0}, v\_{1}): '.format(i + 1, len(vectors)), end='')  
 res = sum([vectors[i][j] \* v[j] for j in range(0, v.size)])  
 print('{0:8.20f}'.format(res))  
 print()  
  
 print('Проверка A \* x = a\_k \* x')  
  
 for i in range(0, len(vectors)):  
 print('a\_k = ', eigenvalue[i])  
 print('x = ', end='')  
 vectors[i].print\_vector()  
 print('A \* x = ', end='')  
 (matrix \* vectors[i]).print\_vector()  
 print('a\_k \* x = ', end='')  
 (vectors[i] \* eigenvalue[i]).print\_vector()  
 print('-------------------------------------------------------------')  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 n = int(data['size'])  
 Eps = float(data['eps'])  
  
 A = Matrix(n)  
 A.matrix\_read\_file(args.input, 'matrix')  
  
 Eigenvalue, V = method\_jacobi\_rotations(A, Eps)  
  
 print('Собственные значения: ')  
 for i in range(0, n):  
 print('a\_{0} = {1:8.5f}'.format(i + 1, Eigenvalue[i]))  
 print('\nМатрица собственных векторов: ')  
 V.print\_matrix()  
  
 test(Eigenvalue, V, A)

1. **Файл с данными (data1.4)**

{  
 "matrix": [  
 [ -7, -9, 1],  
 [ -9, 7, 2],  
 [ 1, 2, 9]  
 ],  
 "size": 3,  
 "eps": 0.01  
}

1. **Результаты работы**

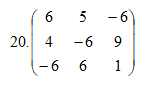
a\_1 = -11.55598  
a\_2 = 12.03659  
a\_3 = 8.51939  
  
Матрица собственных векторов:  
 0.89269 -0.36891 0.25885  
 0.44229 0.82737 -0.34617  
 -0.08646 0.42351 0.90175  
  
Проверка  
v\_1: 0.89269 0.44229 -0.08646  
v\_2: -0.36891 0.82737 0.42351  
(v\_1, v\_2): 0.00000000000000003469  
  
v\_1: 0.89269 0.44229 -0.08646  
v\_3: 0.25885 -0.34617 0.90175  
(v\_1, v\_3): -0.00000000000000004163  
  
v\_2: -0.36891 0.82737 0.42351  
v\_3: 0.25885 -0.34617 0.90175  
(v\_2, v\_3): 0.00000000000000011102  
  
Проверка A \* x = a\_k \* x  
a\_k = -11.555975526027277  
x = 0.89269 0.44229 -0.08646  
A \* x = -10.31595 -5.11112 0.99913  
a\_k \* x = -10.31595 -5.11112 0.99913  
-------------------------------------------------------------  
a\_k = 12.036590138386002  
x = -0.36891 0.82737 0.42351  
A \* x = -4.44046 9.95881 5.09745  
a\_k \* x = -4.44041 9.95873 5.09765  
-------------------------------------------------------------  
a\_k = 8.519385387641274  
x = 0.25885 -0.34617 0.90175  
A \* x = 2.20534 -2.94935 7.68230  
a\_k \* x = 2.20525 -2.94916 7.68240  
-------------------------------------------------------------

**Лабораторная работа 1.5**

1. **Задание ЛР**

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

1. **Вариант: 20**



1. **Алгоритм**

На вход программы подается матрица A и точность вычислений. Для нахождения собственных значений матрицы необходимо найти ее QR разложение, где Q - ортогональная матрица, а R - верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Матрицы Q и R находятся итерационно по формулам , где – соответствующая матрица Хаусхолдера. Изначально Q – единичная, R совпадает с матрицей A. После QR разложения матрицы A получаем новую матрицу с помощью перемножения R и Q: . Таким образом, каждая итерация реализуется в два этапа. На первом этапе осуществляется разложение матрицы в произведение матриц Q и R, а на втором – полученные матрицы перемножаются в обратном порядке. Последовательность сходится к верхней треугольной матрице или к верхней квазитреугольной матрице. Каждому вещественному собственному значению будет соответствовать столбец со стремящимися к нулю поддиагональными элементами. Каждой комплексно-сопряженной паре соответствует диагональный блок размерностью 2х2. Производится проверка, сходятся ли поддиагональные элементы к 0. Если сходятся, то считаем что диагональный элемент - вещественное собственное значение. Иначе проверяем наличие комплексно сопряженной пары. В конце программы полученные результаты сравниваются с результатами, полученными через numpy.

1. **Код (Python)**

import argparse  
import copy  
import json  
import math  
import numpy  
  
from utils import Matrix, Vector  
from numpy.linalg import eig  
  
  
def get\_matrix\_householder(matrix, col):  
 v = Vector(matrix.size)  
 e = Matrix(matrix.size, single=True)  
  
 sign = -1 if matrix[col][col] < 0 else 1 if matrix[col][col] > 0 else 0  
 v[col] = matrix[col][col] + sign \* math.sqrt(sum([matrix[j][col] \*\* 2 for j in range(col, matrix.size)]))  
  
 for i in range(col + 1, matrix.size):  
 v[i] = matrix[i][col]  
  
 v\_vt = Matrix(matrix.size)  
 for i in range(0, v\_vt.size):  
 for j in range(0, v\_vt.size):  
 v\_vt[i][j] = v[i] \* v[j]  
  
 vt\_v = sum([v[i] \*\* 2 for i in range(0, v.size)])  
  
 h = e - v\_vt \* (2 / vt\_v)  
 return h  
  
  
def qr(matrix):  
 q = Matrix(matrix.size, single=True)  
 r = copy.copy(matrix)  
  
 for i in range(0, matrix.size - 1):  
 h = get\_matrix\_householder(r, i)  
  
 q = q \* h  
 r = h \* r  
  
 return q, r  
  
  
def qr\_eigenvalues(matrix, eps):  
 size = matrix.size  
 a\_k = copy.deepcopy(matrix)  
  
 res = [None for \_ in range(0, size)]  
 iteration = 0  
  
 while True:  
 break\_flag = True  
 iteration += 1  
 q\_k, r\_k = qr(a\_k)  
 a\_k = r\_k \* q\_k  
  
 i = 0  
 while i < size - 1:  
 if math.sqrt(sum([a\_k[j][i] \*\* 2 for j in range(i + 1, size)])) < eps:  
 res[i] = a\_k[i][i]  
 i += 1  
 else:  
 roots = numpy.roots([1, -a\_k[i + 1][i + 1] - a\_k[i][i],  
 a\_k[i][i] \* a\_k[i + 1][i + 1] - a\_k[i][i + 1] \* a\_k[i + 1][i]]).tolist()  
 if not (isinstance(res[i], list) and \  
 abs(roots[0] - res[i][0]) < eps and \  
 abs(roots[1] - res[i][1]) < eps):  
 break\_flag = False  
 res[i] = copy.copy(roots)  
 res[i + 1] = None  
 i += 2  
 if not break\_flag:  
 break  
  
 if break\_flag:  
 answer = []  
 print('Итераций: ', iteration)  
 for val in res:  
 if val is None:  
 continue  
 elif isinstance(val, list):  
 answer.append(val[0])  
 answer.append(val[1])  
 else:  
 answer.append(val)  
 if len(answer) == size - 1:  
 answer.append(a\_k[size - 1][size - 1])  
 return answer  
  
  
def test(matrix):  
 print()  
 values, vectors = eig(matrix.matrix)  
 print(values)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 n = int(data['size'])  
 Eps = float(data['eps'])  
  
 A = Matrix(n)  
 A.matrix\_read\_file(args.input, 'matrix')  
  
 result = qr\_eigenvalues(A, Eps)  
  
 print('Собственные значения: ')  
 for i in range(0, n):  
 print('a\_{0} = {1}'.format(i + 1, result[i]))  
  
 test(A)

1. **Файл с данными (data1.5)**

{  
 "matrix": [  
 [ 6, 5, -6],  
 [ 4, -6, 9],  
 [ -6, 6, 1]  
 ],  
 "size": 3,  
 "eps": 0.00000001  
}

1. **Результаты работы**

C:\Users\Admin\Desktop\Численные методы>python 1.5.py --input data1.5  
Итераций: 24  
Собственные значения:  
a\_1 = -13.27629301103898  
a\_2 = 9.861542628176373  
a\_3 = 4.414750382862597  
  
[-13.27629301 9.86154262 4.41475039]

**Лабораторная работа 2.1**

1. **Задание ЛР**

Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

1. **Вариант: 20**

****

1. **Алгоритм**

В программе реализованы методы простой итерации и Ньютона решения нелинейных уравнений. Для использования метода простых итераций исходное уравнение заменяется эквивалентным уравнением с выделенным линейным членом: . Это уравнение подается на вход программы из файла. Решение ищется путем построения последовательности . Для критерия окончания нужно вычислить Желательно, чтобы функция на данном интервале была монотонной. Тогда достаточно вычислить производные на концах отрезка и выбрать наибольшее по модулю значение. Чтобы воспользоваться методом Ньютона, нужно найти начальную точку, в которой значение функции, умноженное на значение второй производной в этой точке больше 0. Далее находится решение согласно итерационной формуле метода. В конце программы производится подстановка полученного решения в исходную функцию и проверка равенству 0.

1. **Код (Python)**

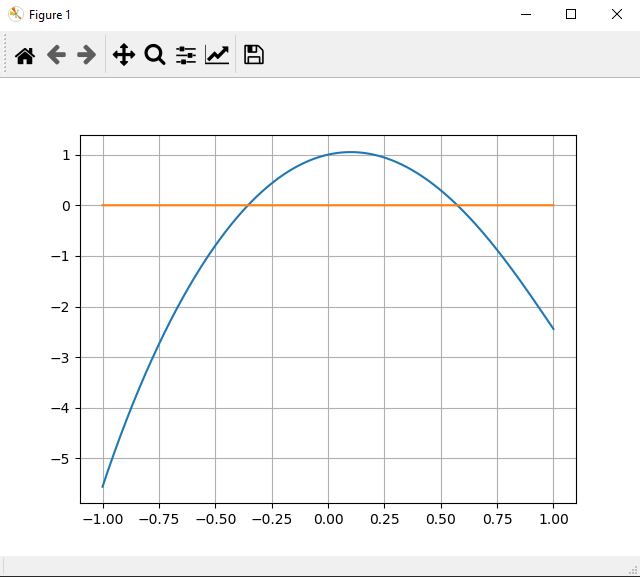
import argparse  
import json  
import sympy  
import matplotlib.pyplot as plt  
from numpy import \*  
from sympy.parsing.sympy\_parser import parse\_expr  
  
  
def newton\_method(equation, a, b, eps):  
 x = sympy.Symbol('x')  
 exp = parse\_expr(equation)  
 func = sympy.lambdify(x, exp)  
  
 exp\_der1 = exp.diff(x)  
 exp\_der2 = exp\_der1.diff(x)  
 func\_der1 = sympy.lambdify(x, exp\_der1)  
 func\_der2 = sympy.lambdify(x, exp\_der2)  
  
 iteration = 0  
 x\_prev = 0  
 if func(a) \* func\_der2(a) > 0:  
 x\_prev = a  
 elif func(b) \* func\_der2(b) > 0:  
 x\_prev = b  
  
 while True:  
 iteration += 1  
 x\_cur = x\_prev - func(x\_prev) / func\_der1(x\_prev)  
  
 if abs(x\_cur - x\_prev) < eps:  
 print('Итераций: ', iteration)  
 return x\_cur  
 x\_prev = x\_cur  
  
  
def method\_of\_simple\_iteration(fi, a, b, eps):  
 x = sympy.Symbol('x')  
 exp\_fi = parse\_expr(fi)  
 exp\_fi\_der1 = exp\_fi.diff(x)  
  
 func\_fi = sympy.lambdify(x, exp\_fi)  
 func\_fi\_der1 = sympy.lambdify(x, exp\_fi\_der1)  
  
 q = max(abs(func\_fi\_der1(a)), abs(func\_fi\_der1(b)))  
 assert q < 1  
  
 iteration = 0  
 x\_prev = (a + b) / 2  
  
 while True:  
 iteration += 1  
 x\_cur = func\_fi(x\_prev)  
  
 if q \* abs(x\_cur - x\_prev) / (1 - q) <= eps:  
 print('Итераций: ', iteration)  
 return x\_cur  
 x\_prev = x\_cur  
  
  
def graph(equation):  
 func = lambda x: eval(equation)  
 x0 = lambda x: x \* 0  
  
 plt.subplots()  
 x = linspace(-1.0, 1.0, 1000000)  
 plt.plot(x, func(x))  
 plt.plot(x, x0(x))  
 plt.grid()  
 plt.show()  
  
  
def test(eq, res):  
 print('Проверка')  
 x = sympy.Symbol('x')  
 exp = parse\_expr(eq)  
 func = sympy.lambdify(x, exp)  
  
 print('{0:.16f}'.format(func(res)))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 while True:  
 method = int(input('Метод простых итераций - 1\nМетод Ньютона - 2\n'))  
 if method in (1, 2):  
 break  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 Eps = float(data['eps'])  
 Equation = data['equation']  
 Interval = data['interval']  
  
 if method == 1:  
 Fi = data['fi']  
  
 graph(Equation)  
  
 if method == 1:  
 X = method\_of\_simple\_iteration(Fi, \*Interval, Eps)  
 elif method == 2:  
 X = newton\_method(Equation, \*Interval, Eps)  
  
 print('x = {0}'.format(X))  
 test(Equation, X)

1. **Файл с данными (data2.1)**

{  
 "equation": "tan(x) - 5 \* x \*\* 2 + 1",  
 "fi": "sqrt((tan(x) + 1) / 5)",  
 "interval": [0.5, 0.6],  
 "eps": 0.00000000001  
}

1. **Результаты работы**

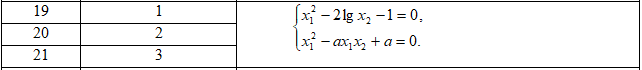
C:\Users\Admin\Desktop\Численные методы>python 2.1.py --input data2.1  
Метод простых итераций - 1  
Метод Ньютона - 2  
1  
Итераций: 16  
x = 0.5738256142161718  
Проверка  
0.0000000000195961  
  
C:\Users\Admin\Desktop\Численные методы>python 2.1.py --input data2.1  
Метод простых итераций - 1  
Метод Ньютона - 2  
2  
Итераций: 4  
x = 0.5738256142207074  
Проверка  
0.0000000000000003

****

**Лабораторная работа 2.2**

1. **Задание ЛР**

Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

1. **Вариант: 20**
2. **Алгоритм**

В программе реализованы методы простой итерации и Ньютона решения систем нелинейных уравнений. Программа строит график для выделения интервалов, на которых нужно искать решения. При использовании метода простой итерации система уравнений приводится к эквивалентной системе в векторной форме. Эти уравнения подаются на вход программы. Для критерия окончания нужно вычислить - матрица производных. Далее решение находится по итерационной формуле с заданной точностью. Методе Ньютона формулы записаны в разрешенном относительно и виде. В конце программы производится подстановка полученного решения в исходную функцию и проверка равенству 0.

1. **Код (Python)**

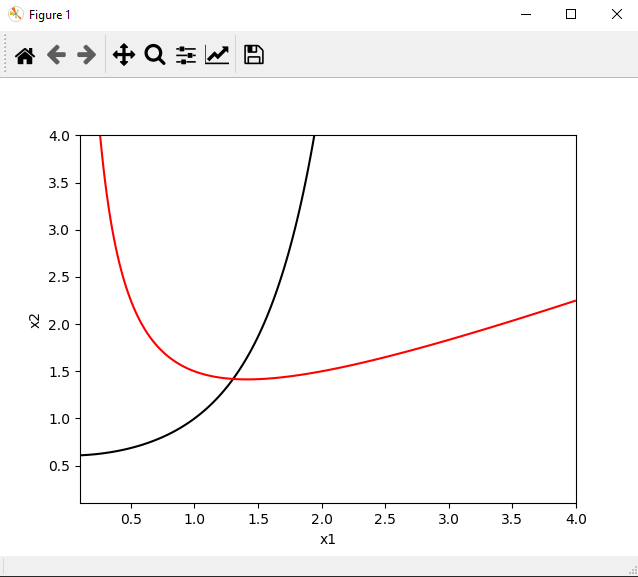
import argparse  
import json  
import sympy  
import matplotlib.pyplot as plt  
from numpy import \*  
from sympy.parsing.sympy\_parser import parse\_expr  
  
  
def method\_of\_simple\_iteration(fi1, fi2, a1, b1, a2, b2, eps):  
 x1 = sympy.Symbol('x1')  
 x2 = sympy.Symbol('x2')  
  
 exp\_fi1 = parse\_expr(fi1)  
 exp\_fi2 = parse\_expr(fi2)  
  
 func\_fi1 = sympy.lambdify([x1, x2], exp\_fi1)  
 func\_fi2 = sympy.lambdify([x1, x2], exp\_fi2)  
  
 exp\_fi1\_der1\_x1 = exp\_fi1.diff(x1)  
 exp\_fi1\_der1\_x2 = exp\_fi1.diff(x2)  
 exp\_fi2\_der1\_x1 = exp\_fi2.diff(x1)  
 exp\_fi2\_der1\_x2 = exp\_fi2.diff(x2)  
  
 func\_fi1\_der1\_x1 = sympy.lambdify([x1, x2], exp\_fi1\_der1\_x1)  
 func\_fi1\_der1\_x2 = sympy.lambdify([x1, x2], exp\_fi1\_der1\_x2)  
 func\_fi2\_der1\_x1 = sympy.lambdify([x1, x2], exp\_fi2\_der1\_x1)  
 func\_fi2\_der1\_x2 = sympy.lambdify([x1, x2], exp\_fi2\_der1\_x2)  
  
 x\_prev = [(a1 + b1) / 2, (a2 + b2) / 2]  
  
 q = None  
 for x\_1 in [a1, b1]:  
 for x\_2 in [a2, b2]:  
 q\_cur = max([abs(func\_fi1\_der1\_x1(x\_1, x\_2)) + abs(func\_fi1\_der1\_x2(x\_1, x\_2)),  
 abs(func\_fi2\_der1\_x1(x\_1, x\_2)) + abs(func\_fi2\_der1\_x2(x\_1, x\_2))])  
 if q is None or q\_cur > q:  
 q = q\_cur  
 assert q < 1  
  
 iteration = 0  
 while True:  
 iteration += 1  
 x\_cur = [func\_fi1(\*x\_prev), func\_fi2(\*x\_prev)]  
  
 if max([abs(x\_cur[0] - x\_prev[0]), abs(x\_cur[1] - x\_prev[1])]) \* q / (1 - q) <= eps:  
 print('Итераций: ', iteration)  
 return x\_cur  
 x\_prev = x\_cur  
  
  
def newton\_method(eq1, eq2, a1, b1, a2, b2, eps):  
 x1 = sympy.Symbol('x1')  
 x2 = sympy.Symbol('x2')  
  
 exp1 = parse\_expr(eq1)  
 exp2 = parse\_expr(eq2)  
  
 func1 = sympy.lambdify([x1, x2], exp1)  
 func2 = sympy.lambdify([x1, x2], exp2)  
  
 exp1\_der1\_x1 = exp1.diff(x1)  
 exp1\_der1\_x2 = exp1.diff(x2)  
 exp2\_der1\_x1 = exp2.diff(x1)  
 exp2\_der1\_x2 = exp2.diff(x2)  
  
 func1\_der1\_x1 = sympy.lambdify([x1, x2], exp1\_der1\_x1)  
 func1\_der1\_x2 = sympy.lambdify([x1, x2], exp1\_der1\_x2)  
 func2\_der1\_x1 = sympy.lambdify([x1, x2], exp2\_der1\_x1)  
 func2\_der1\_x2 = sympy.lambdify([x1, x2], exp2\_der1\_x2)  
  
 detA1 = lambda x\_1, x\_2: func1(x\_1, x\_2) \* func2\_der1\_x2(x\_1, x\_2) - func2(x\_1, x\_2) \* func1\_der1\_x2(x\_1, x\_2)  
 detA2 = lambda x\_1, x\_2: func2(x\_1, x\_2) \* func1\_der1\_x1(x\_1, x\_2) - func1(x\_1, x\_2) \* func2\_der1\_x1(x\_1, x\_2)  
 detJ = lambda x\_1, x\_2: func1\_der1\_x1(x\_1, x\_2) \* func2\_der1\_x2(x\_1, x\_2) - \  
 func2\_der1\_x1(x\_1, x\_2) \* func1\_der1\_x2(x\_1, x\_2)  
  
 x\_prev = [(a1 + b1) / 2, (a2 + b2) / 2]  
 iteration = 0  
  
 while True:  
 iteration += 1  
 x\_cur = [x\_prev[0] - detA1(\*x\_prev) / detJ(\*x\_prev), x\_prev[1] - detA2(\*x\_prev) / detJ(\*x\_prev)]  
  
 if max([abs(x\_cur[0] - x\_prev[0]), abs(x\_cur[1] - x\_prev[1])]) < eps:  
 print('Итераций: ', iteration)  
 return x\_cur  
 x\_prev = x\_cur  
  
  
def graph(equations):  
 func1 = lambda x1, x2: eval(equations[0])  
 func2 = lambda x1, x2: eval(equations[1])  
  
 plt.figure()  
 x1\_list = linspace(0.1, 4.0, 2000)  
 x2\_list = linspace(0.1, 4.0, 2000)  
 plt.xlabel('x1')  
 plt.ylabel('x2')  
 x1, x2 = meshgrid(x1\_list, x2\_list)  
 plt.contour(x1, x2, func1(x1, x2), [0], colors='k')  
 plt.contour(x1, x2, func2(x1, x2), [0], colors='r')  
 plt.show()  
  
  
def test(eq1, eq2, x\_1, x\_2):  
 print('Проверка')  
 x1 = sympy.Symbol('x1')  
 x2 = sympy.Symbol('x2')  
  
 exp1 = parse\_expr(eq1)  
 exp2 = parse\_expr(eq2)  
  
 func1 = sympy.lambdify([x1, x2], exp1)  
 func2 = sympy.lambdify([x1, x2], exp2)  
  
 print('{0:.16f}'.format(func1(x\_1, x\_2)))  
 print('{0:.16f}'.format(func2(x\_1, x\_2)))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 while True:  
 method = int(input('Метод простых итераций - 1\nМетод Ньютона - 2\n'))  
 if method in (1, 2):  
 break  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 Eps = float(data['eps'])  
 Equations = data['equations']  
 Intervals = data['intervals']  
  
 if method == 1:  
 Fi = data['fi']  
  
 graph(Equations)  
  
 if method == 1:  
 X = method\_of\_simple\_iteration(\*Fi, \*Intervals, Eps)  
 elif method == 2:  
 X = newton\_method(\*Equations, \*Intervals, Eps)  
  
 for i in range(0, len(X)):  
 print('x{0} = {1}'.format(i + 1, X[i]))  
 test(\*Equations, \*X)

1. **Файл с данными (data2.2)**

{  
 "equations": ["x1 \*\* 2 - 2 \* log(x2) - 1",  
 "x1 \*\* 2 - 2 \* x1 \* x2 + 2"],  
 "fi": ["sqrt(2 \* log(x2) + 1)",  
 "(x1 \*\* 2 + 2) / (2 \* x1)"],  
 "intervals": [1.25, 1.35, 1.4, 1.45],  
 "eps": 0.0000000001  
}

1. **Результаты работы**

C:\Users\Admin\Desktop\Численные методы>python 2.2.py --input data2.2  
Метод простых итераций - 1  
Метод Ньютона - 2  
1  
Итераций: 14  
x1 = 1.3037464213468484  
x2 = 1.4188935327409418  
Проверка  
0.0000000000103975  
0.0000000000083336  
  
C:\Users\Admin\Desktop\Численные методы>python 2.2.py --input data2.2  
Метод простых итераций - 1  
Метод Ньютона - 2  
2  
Итераций: 3  
x1 = 1.3037464213446917  
x2 = 1.4188935327443284  
Проверка  
0.0000000000000000  
-0.0000000000000004



**Лабораторная работа 3.1**

1. **Задание ЛР**

Используя таблицу значений функции , вычисленных в точках построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки . Вычислить значение погрешности интерполяции в точке .

1. **Вариант: 20**
2. **Алгоритм**

Программа строит интерполяционные многочлены Лагранжа и Ньютона по заданным точкам. На вход программы подается заданная функция, 2 набора точек и точка, в которой требуется вычислить абсолютную погрешность. Так как функция задана в 4 точках, то для построения многочлена Лагранжа 3 степени нужно вычислить Многочлен Ньютона строится через разделенные разности. В конце считается абсолютная погрешность в заданной точке.

1. **Код (Python)**

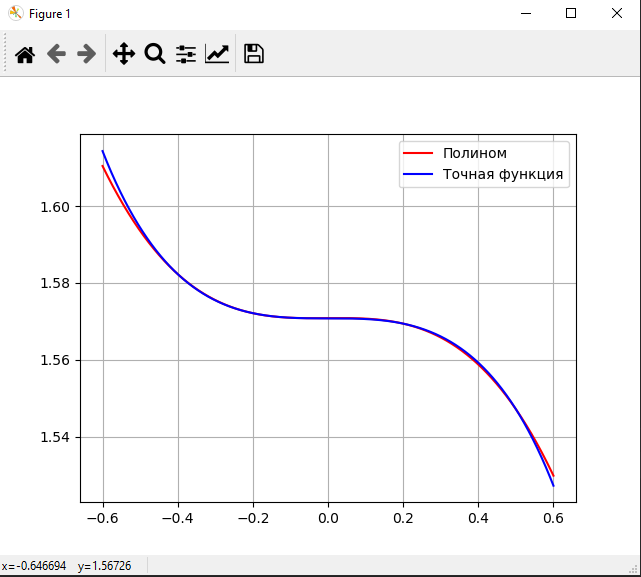
import argparse  
import json  
import sympy  
import operator  
import matplotlib.pyplot as plt  
from numpy import \*  
from pprint import pprint  
from functools import reduce  
from sympy.parsing.sympy\_parser import parse\_expr  
  
  
def get\_w(points, idx):  
 w = ''  
 for i in range(0, len(points)):  
 w += '(x {0} {1}) \* '.format('-' if points[i] > 0 else '+', abs(points[i])) if i != idx else ''  
 return w[0:-2]  
  
  
def lagrange\_interpolation(func, points):  
 size = len(points)  
  
 lagrange = ''  
 for i in range(0, size):  
 w\_n = get\_w(points, i)  
 w\_i = reduce(operator.mul, [points[i] - points[j] for j in range(0, size) if i != j])  
 k = func(points[i]) / w\_i  
 if k > 0:  
 lagrange += '+ '  
 lagrange += '{0} \* {1}'.format(k, w\_n)  
  
 if lagrange[0] == '+':  
 lagrange = lagrange[1:]  
 return lagrange  
  
  
def newton\_interpolation(func, points):  
 size = len(points)  
 divided\_div = [func(point) for point in points]  
  
 newton = '{0}'.format(divided\_div[0])  
 k = ''  
  
 for i in range(1, size):  
 for j in range(size - 1, i - 1, -1):  
 divided\_div[j] = (divided\_div[j - 1] - divided\_div[j]) / (points[j - i] - points[j])  
 k += '(x {0} {1}) '.format('-' if points[i - 1] > 0 else '+', abs(points[i - 1]))  
 if divided\_div[i] > 0:  
 newton += '+'  
 newton += '{0} \* {1}'.format(divided\_div[i], k)  
 k += '\* '  
  
 if newton[0] == '+':  
 newton = newton[1:]  
 return newton  
  
  
def graph(f\_poly, func):  
  
 plt.subplots()  
 x = linspace(-0.6, 0.6, 100000)  
 plt.plot(x, f\_poly(x), color='r', label='Полином')  
 plt.plot(x, func(x), color='b', label='Точная функция')  
 plt.legend()  
 plt.grid()  
 plt.show()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 while True:  
 method = int(input('Лагранж - 1\nНьютон - 2\n'))  
 if method in (1, 2):  
 break  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 Function = data['function']  
 Points = data['points']  
 X\_p = data['x\_p']  
  
 x = sympy.Symbol('x')  
 exp = parse\_expr(Function)  
 function = sympy.lambdify(x, exp)  
  
 for points\_set in Points:  
 if method == 1:  
 poly = lagrange\_interpolation(function, points\_set)  
 elif method == 2:  
 poly = newton\_interpolation(function, points\_set)  
  
 x = sympy.Symbol('x')  
 exp\_poly = sympy.expand(parse\_expr(poly))  
 func\_poly = sympy.lambdify(x, exp\_poly)  
  
 print('Точки: ', points\_set)  
 print('Многочлен {0}: '.format('Лагранжа' if method == 1 else 'Ньютона'))  
 pprint(poly, compact=True)  
 print('Раскрытый вид: ')  
 pprint(exp\_poly)  
 print('Абсолютная погрешность в точке {0}: '.format(X\_p))  
  
 print('{0:.16f}'.format(abs(func\_poly(X\_p) - function(X\_p))))  
 print('----------------------------------------------------')  
  
 graph(func\_poly, function)

1. **Файл с данными (data3.1)**

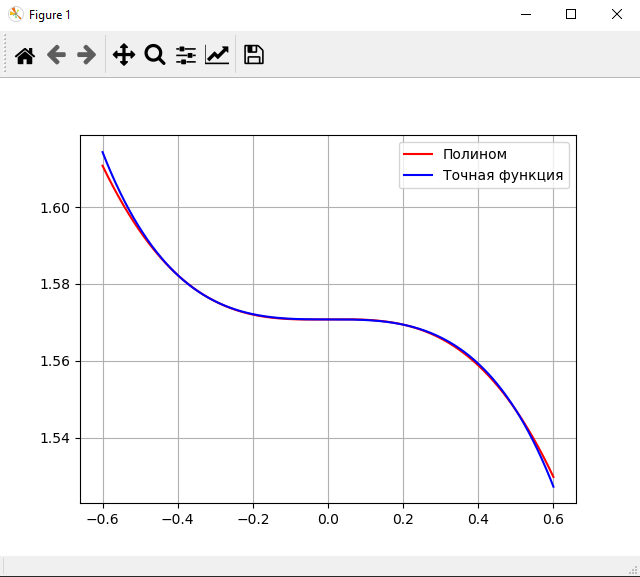
{  
 "function": "acos(x) + x",  
 "points": [[-0.4, -0.1, 0.2, 0.5],  
 [-0.4, 0.0, 0.2, 0.5]],  
 "x\_p": 0.1  
}

1. **Результаты работы**

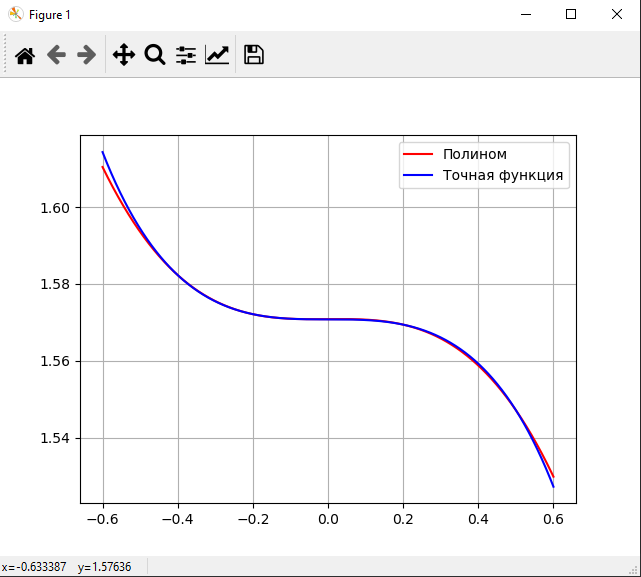
C:\Users\Admin\Desktop\Численные методы>python 3.1.py --input data3.1  
Лагранж - 1  
Ньютон - 2  
1  
Точки: [-0.4, -0.1, 0.2, 0.5]  
Многочлен Лагранжа:  
('-9.767365264582617 \* (x + 0.1) \* (x - 0.2) \* (x - 0.5) + 29.09192125845289 \* '  
 '(x + 0.4) \* (x - 0.2) \* (x - 0.5) -29.063674185269733 \* (x + 0.4) \* (x + '  
 '0.1) \* (x - 0.5) + 9.550602167880234 \* (x + 0.4) \* (x + 0.1) \* (x - 0.2) ')  
Раскрытый вид:  
-0.18851602351922592\*x\*\*3 - 0.0019765684222251868\*x\*\*2 + 0.00076866437483173733\*x + 1.5708718640546424  
Абсолютная погрешность в точке 0.1:  
0.0001115431510472  
----------------------------------------------------

****

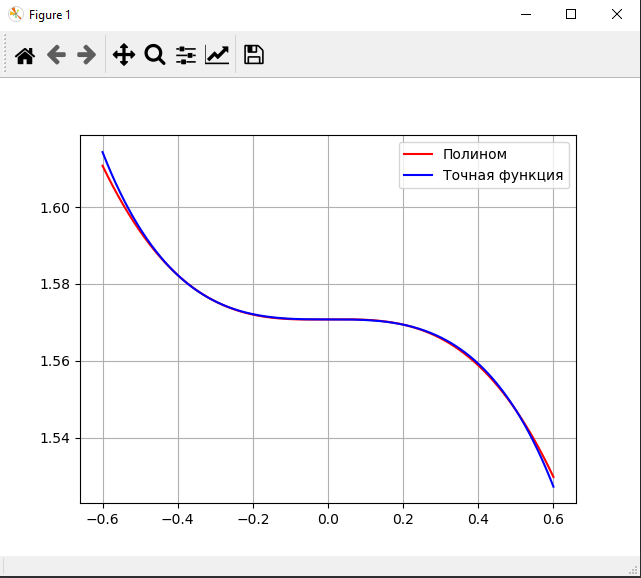
Точки: [-0.4, 0.0, 0.2, 0.5]  
Многочлен Лагранжа:  
('-7.325523948436963 \* (x + 0.0) \* (x - 0.2) \* (x - 0.5) + 39.26990816987241 \* '  
 '(x + 0.4) \* (x - 0.2) \* (x - 0.5) -43.5955112779046 \* (x + 0.4) \* (x + 0.0) '  
 '\* (x - 0.5) + 11.460722601456279 \* (x + 0.4) \* (x + 0.0) \* (x - 0.2) ')  
Раскрытый вид:  
-0.19040445501287262\*x\*\*3 - 0.0014100389741299457\*x\*\*2 + 0.0011085820436871688\*x + 1.570796326794897  
Абсолютная погрешность в точке 0.1:  
0.0000737745211747  
----------------------------------------------------



C:\Users\Admin\Desktop\Численные методы>python 3.1.py --input data3.1  
Лагранж - 1  
Ньютон - 2  
2  
Точки: [-0.4, -0.1, 0.2, 0.5]  
Многочлен Ньютона:  
('1.5823131728623845-0.0378314163530935 \* (x + 0.4) +0.0545782386335415 \* (x + '  
 '0.4) \* (x + 0.1) -0.18851602351922772 \* (x + 0.4) \* (x + 0.1) \* (x - 0.2) ')  
Раскрытый вид:  
-0.18851602351922772\*x\*\*3 - 0.001976568422226821\*x\*\*2 + 0.00076866437483091502\*x + 1.5708718640546426  
Абсолютная погрешность в точке 0.1:  
0.0001115431510472  
----------------------------------------------------



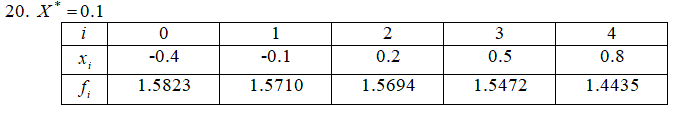
Точки: [-0.4, 0.0, 0.2, 0.5]  
Многочлен Ньютона:  
('1.5823131728623845-0.028792115168719823 \* (x + 0.4) +0.03667085202844385 \* '  
 '(x + 0.4) \* (x + 0.0) -0.190404455012878 \* (x + 0.4) \* (x + 0.0) \* (x - '  
 '0.2) ')  
Раскрытый вид:  
-0.190404455012878\*x\*\*3 - 0.001410038974131755\*x\*\*2 + 0.0011085820436879606\*x + 1.5707963267948966  
Абсолютная погрешность в точке 0.1:  
0.0000737745211743  
----------------------------------------------------



**Лабораторная работа 3.2**

1. **Задание ЛР**

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при и . Вычислить значение функции в точке .

1. **Вариант: 20**
2. **Алгоритм**

Программа строит кубический сплайн для функции, заданной в узлах интерполяции. На вход программы подается набор точек, значений в точках и точка, в которой необходимо найти значение функции. Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить 4n неизвестных Для их нахождения нужно решить систему линейных алгебраических уравнений с трехдиагональной матрицей. Для этого используется метод прогонки из предыдущей лабораторной работы. После нахождения всех коэффициентов, строится сплайн на каждом из участков. В конце вычисляется значение функции в заданной точке.

1. **Код (Python)**

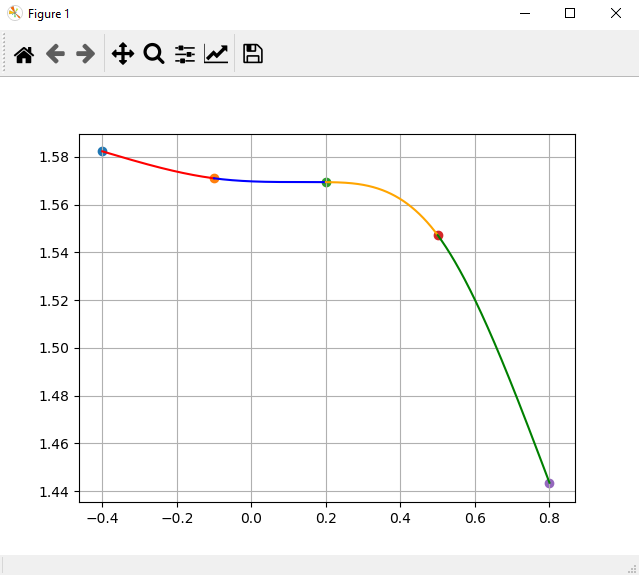
import argparse  
import json  
import sympy  
import matplotlib.pyplot as plt  
from utils import TridiagonalMatrix, Vector  
from rtm import rtm  
from numpy import \*  
from sympy.parsing.sympy\_parser import parse\_expr  
  
  
def find\_interval(points, x):  
 for i in range(1, len(points)):  
 if points[i - 1] < x <= points[i]:  
 return i  
  
  
def get\_c(x, f, h):  
 a\_diag = [0.0] + [h[i - 1] for i in range(3, len(x))]  
 b\_diag = [2 \* (h[i - 1] + h[i]) for i in range(2, len(x))]  
 c\_diag = [h[i] for i in range(2, len(x) - 1)] + [0.0]  
 d = [3 \* ((f[i] - f[i - 1]) / h[i] - ((f[i - 1] - f[i - 2]) / h[i - 1])) for i in range(2, len(x))]  
  
 matrix = TridiagonalMatrix(len(x) - 2)  
 matrix.A = a\_diag  
 matrix.B = b\_diag  
 matrix.C = c\_diag  
  
 vec = Vector(len(x) - 2)  
 vec.vector = d  
  
 c = [0.0, 0.0] + rtm(matrix, vec)  
 return c  
  
  
def get\_a(f):  
 return [0.0] + [f[i - 1] for i in range(1, len(f))]  
  
  
def get\_b(f, h, c):  
 last = len(f) - 1  
 b = [0.0]  
 b.extend([(f[i] - f[i - 1]) / h[i] - 1 / 3 \* h[i] \* (c[i + 1] + 2 \* c[i]) for i in range(1, last)])  
 b.append((f[last] - f[last - 1]) / h[last] - 2 / 3 \* h[last] \* c[last])  
 return b  
  
  
def get\_d(h, c):  
 last = len(h) - 1  
 d = [0.0]  
 d.extend([(c[i + 1] - c[i]) / (3 \* h[i]) for i in range(1, last)])  
 d.append(-c[last] / (3 \* h[last]))  
 return d  
  
  
def spline\_interpolation(x, f):  
 h = [0.0] + [x[i] - x[i - 1] for i in range(1, len(x))]  
 c = get\_c(x, f, h)  
 a = get\_a(f)  
 b = get\_b(f, h, c)  
 d = get\_d(h, c)  
  
 return a, b, c, d  
  
  
def make\_splines\_exp(a, b, c, d, pt):  
 splines\_exp = []  
 for i in range(1, len(pt)):  
 splines\_exp.append(  
 f'{a[i]} + {b[i]} \* (x - {pt[i - 1]}) + {c[i]} \* (x - {pt[i - 1]})\*\*2 + {d[i]} \* (x - {pt[i - 1]})\*\*3')  
 return splines\_exp  
  
  
def print\_splines(splines\_exp, points):  
 for i in range(0, len(points) - 1):  
 print('Отрезок [{0},{1}]'.format(points[i], points[i + 1]))  
 print(sympy.expand(parse\_expr(splines\_exp[i])))  
 print()  
  
  
def make\_splines\_func(splines\_exp):  
 x = sympy.Symbol('x')  
 splines\_func = []  
 for spline\_exp in splines\_exp:  
 splines\_func.append(sympy.lambdify(x, spline\_exp))  
 return splines\_func  
  
  
def graph(splines\_f, points):  
 colors = ['red', 'blue', 'orange', 'green']  
 size = len(points)  
  
 plt.subplots()  
 for i in range(0, size - 1):  
 x = linspace(points[i], points[i + 1], 1000000)  
 plt.plot(x, splines\_f[i](x), color=colors[i])  
 plt.scatter(points[i], splines\_f[i](points[i]))  
 plt.scatter(points[size - 1], splines\_f[i](points[size - 1]))  
 plt.grid()  
 plt.show()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 with open(args.input, 'r') as file:  
 data = json.load(file)  
 Points = data['points']  
 Values = data['values']  
 X\_p = data['x\_p']  
  
 A, B, C, D = spline\_interpolation(Points, Values)  
  
 Splines = make\_splines\_exp(A, B, C, D, Points)  
 Splines\_f = make\_splines\_func(Splines)  
 print\_splines(Splines, Points)  
 graph(Splines\_f, Points)  
  
 idx = find\_interval(Points, X\_p)  
 print('Значение в точке {0}:'.format(X\_p))  
 print(Splines\_f[idx](X\_p))

1. **Файл с данными (data3.2)**

{  
 "points": [-0.4, -0.1, 0.2, 0.5, 0.8],  
 "values": [1.5823, 1.5710, 1.5694, 1.5472, 1.4435],  
 "x\_p": 0.1  
}

1. **Результаты работы**

C:\Users\Admin\Desktop\Численные методы>python 3.2.py --input data3.2  
Отрезок [-0.4,-0.1]  
0.09682539682539713\*x\*\*3 + 0.1161904761904766\*x\*\*2 + 9.5238095237935741e-5\*x + 1.5699444444444444  
Отрезок [-0.1,0.2]  
-0.12486772486772513\*x\*\*3 + 0.049682539682539896\*x\*\*2 - 0.0065555555555557197\*x + 1.5697227513227513  
Отрезок [0.2,0.5]  
-0.7195767195767185\*x\*\*3 + 0.406507936507936\*x\*\*2 - 0.07792063492063493\*x + 1.5744804232804232  
Отрезок [0.5,0.8]  
0.7476190476190464\*x\*\*3 - 1.794285714285711\*x\*\*2 + 1.0224761904761887\*x + 1.3910809523809526  
  
Значение в точке 0.1:  
1.5700338624338623

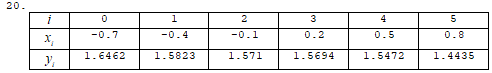


**Лабораторная работа 3.3**

1. **Задание ЛР**

Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены a) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

1. **Вариант: 20**

****

1. **Алгоритм**

Программа находит приближающие многочлены 1-ой и 2-ой степени для заданной табличной функции. На вход программе подается набор точек и значений. Многочлен 1-ой степени ищется в виде , многочлен 2-ой степени в виде Для нахождения неизвестных коэффициентов нужно записать нормальную систему МНК и решить ее. В программе система решается методом LU разложения из предыдущей лабораторной работы. В конце программа считает сумму квадратов ошибок и строит графики.

1. **Код (Python)**

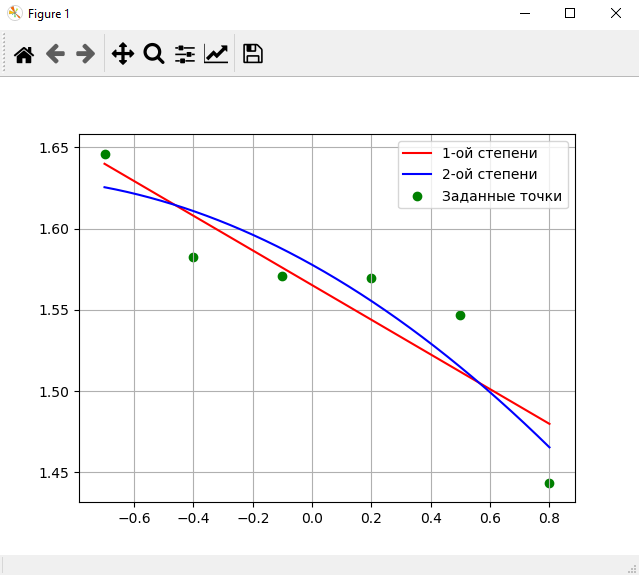
import argparse  
import json  
import sympy  
import matplotlib.pyplot as plt  
from utils import Matrix, Vector  
from lup import lup, lup\_solve  
from numpy import \*  
from sympy.parsing.sympy\_parser import parse\_expr  
  
  
def get\_poly\_1\_deg(points, values):  
 sum\_points = sum(points)  
 sum\_points\_2 = sum([point \*\* 2 for point in points])  
 sum\_values = sum(values)  
 sum\_v\_p = sum([values[i] \* points[i] for i in range(0, len(points))])  
  
 a = Matrix(2)  
 a.matrix = [[len(points), sum\_points], [sum\_points, sum\_points\_2]]  
 b = Vector(2)  
 b.vector = [sum\_values, sum\_v\_p]  
  
 l, u, p = lup(a)  
 coefficients = lup\_solve(l, u, p, b).vector  
  
 return parse\_expr(f'{coefficients[0]} + {coefficients[1]} \* x')  
  
  
def get\_poly\_2\_deg(points, values):  
 sum\_points = sum(points)  
 sum\_points\_2 = sum([point \*\* 2 for point in points])  
 sum\_points\_3 = sum([point \*\* 3 for point in points])  
 sum\_points\_4 = sum([point \*\* 4 for point in points])  
 sum\_values = sum(values)  
 sum\_v\_p = sum([values[i] \* points[i] for i in range(0, len(points))])  
 sum\_v\_p2 = sum([values[i] \* points[i] \*\* 2 for i in range(0, len(points))])  
  
 a = Matrix(3)  
 a.matrix = [  
 [len(points), sum\_points, sum\_points\_2],  
 [sum\_points, sum\_points\_2, sum\_points\_3],  
 [sum\_points\_2, sum\_points\_3, sum\_points\_4]]  
 b = Vector(3)  
 b.vector = [sum\_values, sum\_v\_p, sum\_v\_p2]  
  
 l, u, p = lup(a)  
 coefficients = lup\_solve(l, u, p, b).vector  
  
 return parse\_expr(f'{coefficients[0]} + {coefficients[1]} \* x + {coefficients[2]} \* x \*\* 2')  
  
  
def get\_poly\_f(poly\_exp):  
 x = sympy.Symbol('x')  
 return sympy.lambdify(x, poly\_exp)  
  
  
def get\_sum\_err\_squares(poly\_f, points, values):  
 return sum([(poly\_f(points[i]) - values[i]) \*\* 2 for i in range(0, len(points))])  
  
  
def graph(poly\_1\_f, poly\_2\_f, points, values):  
 size = len(points)  
  
 plt.subplots()  
 x = linspace(points[0], points[size - 1], 100000)  
 plt.plot(x, poly\_1\_f(x), color='r', label='1-ой степени')  
 plt.plot(x, poly\_2\_f(x), color='b', label='2-ой степени')  
 for i in range(0, size - 1):  
 plt.scatter(points[i], values[i], color='g')  
 plt.scatter(points[size - 1], values[size - 1], color='g', label='Заданные точки')  
 plt.legend()  
 plt.grid()  
 plt.show()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 with open(args.input, 'r') as file:  
 data = json.load(file)  
 Points = data['points']  
 Values = data['values']  
  
 poly\_1 = get\_poly\_1\_deg(Points, Values)  
 poly\_2 = get\_poly\_2\_deg(Points, Values)  
  
 poly\_f\_1 = get\_poly\_f(poly\_1)  
 poly\_f\_2 = get\_poly\_f(poly\_2)  
  
 poly\_l = [poly\_1, poly\_2]  
 poly\_l\_f = [poly\_f\_1, poly\_f\_2]  
 for i in range(2):  
 print(f'Многочлен степени {i + 1}: ')  
 print(poly\_l[i])  
 print(f'Сумма квадратов ошибок: {get\_sum\_err\_squares(poly\_l\_f[i], Points, Values)}')  
 print()  
 graph(\*poly\_l\_f, Points, Values)

1. **Файл с данными (data3.3)**

{  
 "points": [-0.7, -0.4, -0.1, 0.2, 0.5, 0.8],  
 "values": [1.6462, 1.5823, 1.571, 1.5694, 1.5472, 1.4435]  
}

1. **Результаты работы**

C:\Users\Admin\Desktop\Численные методы>python 3.3.py --input data3.3  
Многочлен степени 1:  
1.5652685714285715 - 0.10670476190476182\*x  
Сумма квадратов ошибок: 0.003940351047619036  
  
Многочлен степени 2:  
-0.04813492063492142\*x\*\*2 - 0.10189126984126967\*x + 1.5777836507936511  
Сумма квадратов ошибок: 0.0032396991428571406

****

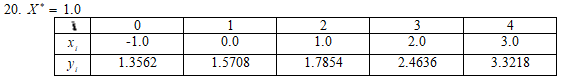
**Лабораторная работа 3.4**

1. **Задание ЛР**

Вычислить первую и вторую производную от таблично заданной функции

в точке .

1. **Вариант: 20**

****

1. **Алгоритм**

Программа считает 1-ую и 2-ую производную для таблично заданной функции. На вход программы поступает набор точек, набор значений и точка, в которой нужно вычислить производные. Для вычисления производных нужно определить интервал, которому принадлежит заданная точка. Первая и вторая производная вычисляется со вторым порядком точности.

1. **Код (Python)**

import argparse  
import json  
  
  
def find\_interval(points, x\_p):  
 for i in range(0, len(points) - 2):  
 if points[i] < x\_p <= points[i + 1]:  
 return i  
 return -1  
  
  
def get\_derivative\_2(points, val, x\_p):  
 i = find\_interval(points, x\_p)  
 if i == -1:  
 raise Exception("Недостаточно точек")  
  
 k\_1 = (val[i + 2] - val[i + 1]) / (points[i + 2] - points[i + 1])  
 k\_2 = ((val[i + 1] - val[i]) / (points[i + 1] - points[i]))  
  
 return 2 \* (k\_1 - k\_2) / (points[i + 2] - points[i])  
  
  
def get\_derivative\_1(points, val, x\_p):  
 i = find\_interval(points, x\_p)  
 if i == -1:  
 raise Exception("Недостаточно точек")  
  
 k\_1 = (val[i + 1] - val[i]) / (points[i + 1] - points[i])  
 k\_2 = (val[i + 2] - val[i + 1]) / (points[i + 2] - points[i + 1])  
 k\_3 = (2 \* x\_p - points[i] - points[i + 1]) / (points[i + 2] - points[i])  
  
 return k\_1 + (k\_2 - k\_1) \* k\_3  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 Points = data['points']  
 Values = data['values']  
 X\_p = data['x\_p']  
  
 dev1 = get\_derivative\_1(Points, Values, X\_p)  
 dev2 = get\_derivative\_2(Points, Values, X\_p)  
  
 print('Точка x = ', X\_p)  
 print('Первая производная: ', dev1)  
 print('Вторая производная: ', dev2)

1. **Файл с данными (data3.4)**

{  
 "points": [-1.0, 0.0, 1.0, 2.0, 3.0],  
 "values": [1.3562, 1.5708, 1.7854, 2.4636, 3.3218],  
 "x\_p": 1.0  
}

1. **Результаты работы**

C:\Users\Admin\Desktop\Численные методы>python 3.4.py --input data3.4  
Точка x = 1.0  
Первая производная: 0.4464  
Вторая производная: 0.4635999999999998

**Лабораторная работа 3.5**

1. **Задание ЛР**

Вычислить определенный интеграл , методами прямоугольников, трапеций, Симпсона с шагами . Оценить погрешность вычислений, используя Ме­тод Рунге-Ромберга.

1. **Вариант: 20**

****

1. **Алгоритм**

Программа находит значение определенного интеграла методами прямоугольников, трапеций и Симпсона. В зависимости от метода используется своя формула нахождения решения. Решение находится для двух значений шагов, далее оценивается погрешность вычислений по методу Рунге-Ромберга.

1. **Код (Python)**

import argparse  
import json  
import sympy  
from numpy import arange  
from sympy.parsing.sympy\_parser import parse\_expr  
  
  
def get\_points(x\_0, x\_k, step):  
 return arange(x\_0, x\_k, step).tolist() + [x\_k]  
  
  
def rectangle\_method(func, x, step):  
 return step \* sum([func((x[i] + x[i + 1]) / 2) for i in range(0, len(x) - 1)])  
  
  
def trapeze\_method(func, x, step):  
 size = len(x)  
 return step \* (func(x[0]) / 2 + sum([func(x[i]) for i in range(1, size - 1)]) + func(x[size - 1]) / 2)  
  
  
def simpson\_method(func, x, step):  
 size = len(x)  
 return step / 3 \* (func(x[0]) + sum([4 \* func(x[i]) for i in range(1, size - 1, 2)]) +  
 sum([2 \* func(x[i]) for i in range(2, size - 2, 2)]) + func(x[size - 1]))  
  
  
def runge\_romberg\_method(res):  
 k = res[0]['H'] / res[1]['H']  
  
 r\_rec = abs((res[1]['Rectangle'] - res[0]['Rectangle'])) / (k \*\* 2 - 1)  
 r\_tr = abs((res[1]['Trapeze'] - res[0]['Trapeze'])) / (k \*\* 2 - 1)  
 r\_simp = abs((res[1]['Simpson'] - res[0]['Simpson'])) / (k \*\* 4 - 1)  
  
 return {'r\_rec': r\_rec, 'r\_tr': r\_tr, 'r\_simp': r\_simp}  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 Function = data['function']  
 X\_0 = data['x\_0']  
 X\_k = data['x\_k']  
 Steps = data['steps']  
  
 x = sympy.Symbol('x')  
 exp = parse\_expr(Function)  
 function = sympy.lambdify(x, exp)  
  
 results = []  
 for h in Steps:  
 X = get\_points(X\_0, X\_k, h)  
  
 res\_rectangle = rectangle\_method(function, X, h)  
 res\_trapeze = trapeze\_method(function, X, h)  
 res\_simpson = simpson\_method(function, X, h)  
  
 print('Шаг: ', h)  
 print('Метод прямоугольников: ', res\_rectangle)  
 print('Метод трапеций: ', res\_trapeze)  
 print('Метод Симпсона: ', res\_simpson)  
 print()  
  
 results.append({  
 'Rectangle': res\_rectangle,  
 'Trapeze': res\_trapeze,  
 'Simpson': res\_simpson,  
 'H': h  
 })  
 r\_result = runge\_romberg\_method(results)  
  
 print('Ошибки по методу Рунге-Ромберга'.format(Steps[0]))  
 print('Метод прямоугольников: ', r\_result['r\_rec'])  
 print('Метод трапеций: ', r\_result['r\_tr'])  
 print('Метод Симпсона: ', r\_result['r\_simp'])

1. **Файл с данными (data3.5)**

{  
 "function": "sqrt(x) / (4 + 3 \* x)",  
 "x\_0": 1,  
 "x\_k": 5,  
 "steps": [1.0, 0.5]  
}

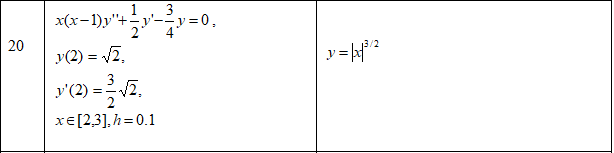
1. **Результаты работы**

C:\Users\Admin\Desktop\Численные методы>python 3.5.py --input data3.5  
Шаг: 1.0  
Метод прямоугольников: 0.5318189388349329  
Метод трапеций: 0.5299284993159105  
Метод Симпсона: 0.5308999037021467  
  
Шаг: 0.5  
Метод прямоугольников: 0.5313885217114334  
Метод трапеций: 0.5308737190754217  
Метод Симпсона: 0.5311887923285921  
  
Ошибки по методу Рунге-Ромберга  
Метод прямоугольников: 0.0001434723744998534  
Метод трапеций: 0.00031507325317040963  
Метод Симпсона: 1.9259241763025513e-05

**Лабораторная работа 4.1**

1. **Задание ЛР**

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки . С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

1. **Вариант: 20**

Условие не соответствует заданному точному решению. Пришлось заменить на .

1. **Алгоритм**

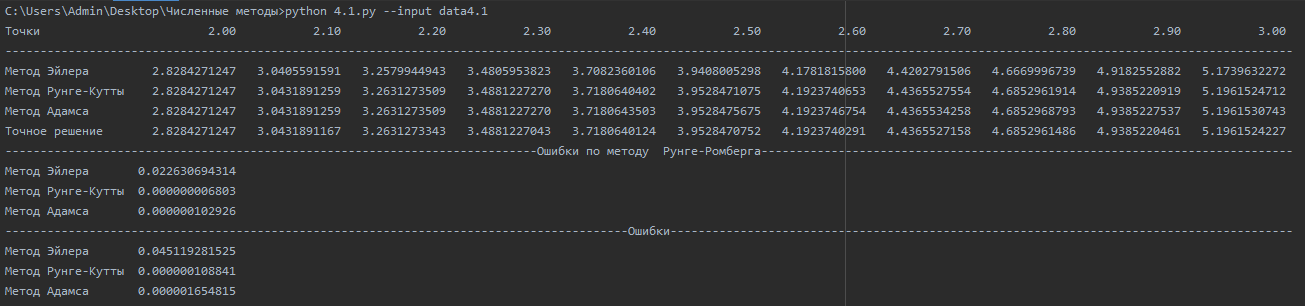
Программа решает задачу Коши для ОДУ 2-го порядка методами Эйлера, Рунге-Кутты и Адамса 4-го порядка. Метод Адамса использует метод Рунге-Кутты для получения первых четырех точек. На вход программы подаются коэффициенты уравнения, начальные условия, интервал, шаг сетки и точное решение. Исходное уравнение переписывается в виде системы ОДУ 1-го порядка и решается всеми 3 способами. Задача решается с 2 значениями шага для оценки погрешности по методу Рунге-Ромберга. В конце программа выводит результаты и строит графики.

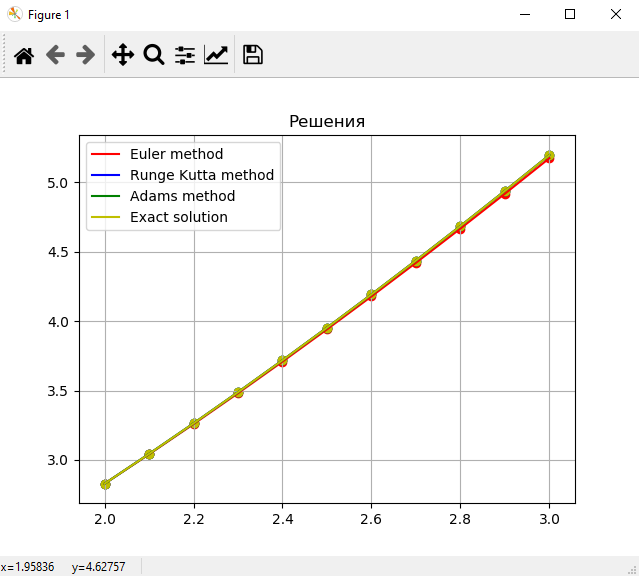
1. **Код (Python)**

import argparse  
import json  
import sympy  
import matplotlib.pyplot as plt  
from numpy import \*  
from sympy.parsing.sympy\_parser import parse\_expr  
  
  
def exact\_values(points, sol\_f):  
 return [sol\_f(p) for p in points]  
  
  
def method\_runge\_kutta(points, start\_cond\_0, start\_cond\_1, step, f, g):  
 values = [start\_cond\_0]  
 values\_der = [start\_cond\_1]  
  
 for i in range(0, len(points) - 1):  
 x\_k = points[i]  
 y\_k = values[i]  
 z\_k = values\_der[i]  
  
 k\_1 = step \* f(x\_k, y\_k, z\_k)  
 l\_1 = step \* g(x\_k, y\_k, z\_k)  
  
 k\_2 = step \* f(x\_k + 0.5 \* step, y\_k + 0.5 \* k\_1, z\_k + 0.5 \* l\_1)  
 l\_2 = step \* g(x\_k + 0.5 \* step, y\_k + 0.5 \* k\_1, z\_k + 0.5 \* l\_1)  
  
 k\_3 = step \* f(x\_k + 0.5 \* step, y\_k + 0.5 \* k\_2, z\_k + 0.5 \* l\_2)  
 l\_3 = step \* g(x\_k + 0.5 \* step, y\_k + 0.5 \* k\_2, z\_k + 0.5 \* l\_2)  
  
 k\_4 = step \* f(x\_k + step, y\_k + k\_3, z\_k + l\_3)  
 l\_4 = step \* g(x\_k + step, y\_k + k\_3, z\_k + l\_3)  
  
 dy\_k = 1 / 6 \* (k\_1 + 2 \* k\_2 + 2 \* k\_3 + k\_4)  
 dz\_k = 1 / 6 \* (l\_1 + 2 \* l\_2 + 2 \* l\_3 + l\_4)  
  
 values.append(y\_k + dy\_k)  
 values\_der.append(z\_k + dz\_k)  
  
 return values, values\_der  
  
  
def method\_euler(points, start\_cond\_0, start\_cond\_1, step, f, g):  
 values = [start\_cond\_0]  
 values\_der = [start\_cond\_1]  
  
 for i in range(0, len(points) - 1):  
 x\_k = points[i]  
 y\_k = values[i]  
 z\_k = values\_der[i]  
  
 values.append(y\_k + step \* f(x\_k, y\_k, z\_k))  
 values\_der.append(z\_k + step \* g(x\_k, y\_k, z\_k))  
  
 return values, values\_der  
  
  
def method\_adams(x, start\_cond\_0, start\_cond\_1\_, step, f, g):  
 y, z = method\_runge\_kutta(x[:4], start\_cond\_0, start\_cond\_1\_, step, f, g)  
  
 for i in range(3, len(x) - 1):  
 y.append(  
 y[i] + step / 24 \* (55 \* f(x[i], y[i], z[i]) -  
 59 \* f(x[i - 1], y[i - 1], z[i - 1]) +  
 37 \* f(x[i - 2], y[i - 2], z[i - 2]) -  
 9 \* f(x[i - 3], y[i - 3], z[i - 3]))  
 )  
 z.append(  
 z[i] + step / 24 \* (55 \* g(x[i], y[i], z[i]) -  
 59 \* g(x[i - 1], y[i - 1], z[i - 1]) +  
 37 \* g(x[i - 2], y[i - 2], z[i - 2]) -  
 9 \* g(x[i - 3], y[i - 3], z[i - 3]))  
 )  
 return y, z  
  
  
def get\_sol(points, coeffs, start\_cond\_0, start\_cond\_1, step, solution):  
 f\_exp = parse\_expr('z')  
 g\_exp = parse\_expr(f'-({coeffs[1]} \* z + {coeffs[2]} \* y + {coeffs[3]}) / ({coeffs[0]})')  
 sol\_exp = parse\_expr(solution)  
  
 x = sympy.Symbol('x')  
 y = sympy.Symbol('y')  
 z = sympy.Symbol('z')  
 f = sympy.lambdify([x, y, z], f\_exp)  
 g = sympy.lambdify([x, y, z], g\_exp)  
 sol\_f = sympy.lambdify(x, sol\_exp)  
  
 val\_euler, val\_der\_euler = method\_euler(points, start\_cond\_0, start\_cond\_1, step, f, g)  
 val\_runge\_kutta, val\_der\_runge\_kutta = method\_runge\_kutta(points, start\_cond\_0, start\_cond\_1, step, f, g)  
 val\_adams, val\_der\_adams = method\_adams(points, start\_cond\_0, start\_cond\_1, step, f, g)  
 val\_exact = exact\_values(points, sol\_f)  
  
 return {'euler': val\_euler, 'runge\_kutta': val\_runge\_kutta, 'adams': val\_adams, 'exact': val\_exact}  
  
  
def err\_runge\_romberg(res\_1, res\_2, p):  
 return (sum([(res\_1[i] - res\_2[2 \* i]) \*\* 2 for i in range(0, len(res\_1))]) \*\* 0.5) / (2 \*\* p - 1)  
  
  
def get\_err\_runge\_romberg(res\_1, res\_2):  
 err\_euler = err\_runge\_romberg(res\_1['euler'], res\_2['euler'], 1)  
 err\_runge\_kutta = err\_runge\_romberg(res\_1['runge\_kutta'], res\_2['runge\_kutta'], 4)  
 err\_adams = err\_runge\_romberg(res\_1['adams'], res\_2['adams'], 4)  
  
 return {'euler': err\_euler, 'runge\_kutta': err\_runge\_kutta, 'adams': err\_adams}  
  
  
def err\_exact(res, exc\_val):  
 return sum([(res[i] - exc\_val[i]) \*\* 2 for i in range(0, len(res))]) \*\* 0.5  
  
  
def get\_err\_exact(res):  
 exc\_val = res['exact']  
  
 exc\_err\_euler = err\_exact(res['euler'], exc\_val)  
 exc\_err\_runge\_kutta = err\_exact(res['runge\_kutta'], exc\_val)  
 exc\_err\_adams = err\_exact(res['adams'], exc\_val)  
  
 return {'euler': exc\_err\_euler, 'runge\_kutta': exc\_err\_runge\_kutta, 'adams': exc\_err\_adams}  
  
  
def get\_points(begin, end, step):  
 return arange(begin, end, step).tolist() + [end]  
  
  
def print\_results(res, err\_rr, err\_exc, pt):  
 print('{0:18s}'.format('Точки'), end='')  
 for point in pt:  
 print('{0:15.2f}'.format(point), end='')  
 print('\n' + '-' \* 184)  
  
 print('{0:18s}'.format('Метод Эйлера'), end='')  
 for val in res['euler']:  
 print('{0:15.10f}'.format(val), end='')  
  
 print('\n{0:18s}'.format('Метод Рунге-Кутты'), end='')  
 for val in res['runge\_kutta']:  
 print('{0:15.10f}'.format(val), end='')  
  
 print('\n{0:18s}'.format('Метод Адамса'), end='')  
 for val in res['adams']:  
 print('{0:15.10f}'.format(val), end='')  
  
 print('\n{0:18s}'.format('Точное решение'), end='')  
 for val in res['exact']:  
 print('{0:15.10f}'.format(val), end='')  
  
 print('\n' + '-' \* 76 + 'Ошибки по методу Рунге-Ромберга' + '-' \* 76)  
 print('{0:18s}'.format('Метод Эйлера'), end='')  
 print('{0:15.12f}'.format(err\_rr['euler']), end='')  
  
 print('\n{0:18s}'.format('Метод Рунге-Кутты'), end='')  
 print('{0:15.12f}'.format(err\_rr['runge\_kutta']), end='')  
  
 print('\n{0:18s}'.format('Метод Адамса'), end='')  
 print('{0:15.12f}'.format(err\_rr['adams']), end='')  
  
 print('\n' + '-' \* 89 + 'Ошибки' + '-' \* 89)  
 print('{0:18s}'.format('Метод Эйлера'), end='')  
 print('{0:15.12f}'.format(err\_exc['euler']), end='')  
  
 print('\n{0:18s}'.format('Метод Рунге-Кутты'), end='')  
 print('{0:15.12f}'.format(err\_exc['runge\_kutta']), end='')  
  
 print('\n{0:18s}'.format('Метод Адамса'), end='')  
 print('{0:15.12f}'.format(err\_exc['adams']), end='')  
  
  
def graph(res, pt):  
 plt.subplots()  
 plt.plot(pt, res['euler'], color='r', label='Euler method')  
 plt.plot(pt, res['runge\_kutta'], color='b', label='Runge Kutta method')  
 plt.plot(pt, res['adams'], color='g', label='Adams method')  
 plt.plot(pt, res['exact'], color='y', label='Exact solution')  
 for i in range(0, len(pt)):  
 plt.scatter(pt[i], res['euler'][i], color='r')  
 plt.scatter(pt[i], res['runge\_kutta'][i], color='b')  
 plt.scatter(pt[i], res['adams'][i], color='g')  
 plt.scatter(pt[i], res['exact'][i], color='y')  
 plt.legend()  
 plt.title('Решения')  
 plt.grid()  
 plt.show()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 Coefficients = data['coefficients']  
 Start\_cond\_0 = data['condition\_0']  
 Start\_cond\_1 = data['condition\_1']  
 Interval = data['interval']  
 Step = data['step']  
 Solution = data['solution']  
  
 points = get\_points(\*Interval, Step)  
 points\_2 = get\_points(\*Interval, Step / 2)  
  
 coeffs = [parse\_expr(coeff) for coeff in Coefficients]  
 cond\_0 = float(parse\_expr(Start\_cond\_0))  
 cond\_1 = float(parse\_expr(Start\_cond\_1))  
  
 result = get\_sol(points, coeffs, cond\_0, cond\_1, Step, Solution)  
 result\_2 = get\_sol(points\_2, coeffs, cond\_0, cond\_1, Step / 2, Solution)  
  
 err\_runge\_romberg = get\_err\_runge\_romberg(result, result\_2)  
 err\_exact = get\_err\_exact(result)  
  
 graph(result, points)  
 print\_results(result, err\_runge\_romberg, err\_exact, points)

1. **Файл с данными (data4.1)**

{  
 "coefficients": ["x\*(x-1)", "0.5", "-0.75", "0"],  
 "condition\_0": "2 \*\* 1.5",  
 "condition\_1": "1.5 \* sqrt(2)",  
 "interval": [2, 3],  
 "step": 0.1,  
 "solution": "(abs(x))\*\*1.5"  
}

1. **Результаты работы**

****

**Лабораторная работа 4.2**

1. **Задание ЛР**

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

1. **Вариант: 20**



1. **Алгоритм**

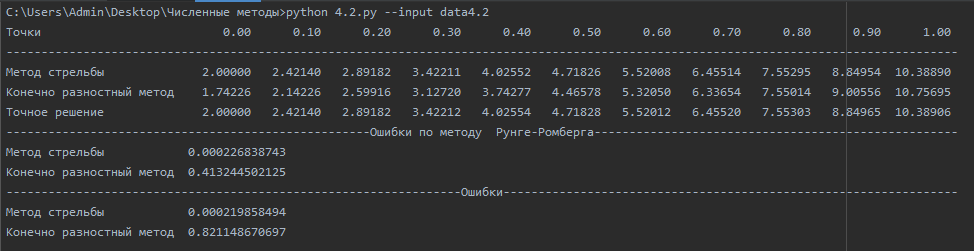
В программе реализован метод стрельбы и конечно разностный метод для решения краевой задачи для ОДУ. Суть метода заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи. Для решения задачи Коши используется метод Рунге-Кутты из предыдущей лабораторной. Следующее значение параметра вычисляется методом секущих. В конечно-разностном методе нужно решить систему с трехдиагональной матрицей. Для решения системы используется метод прогонки. В конце программа выводит результаты и рисует графики.

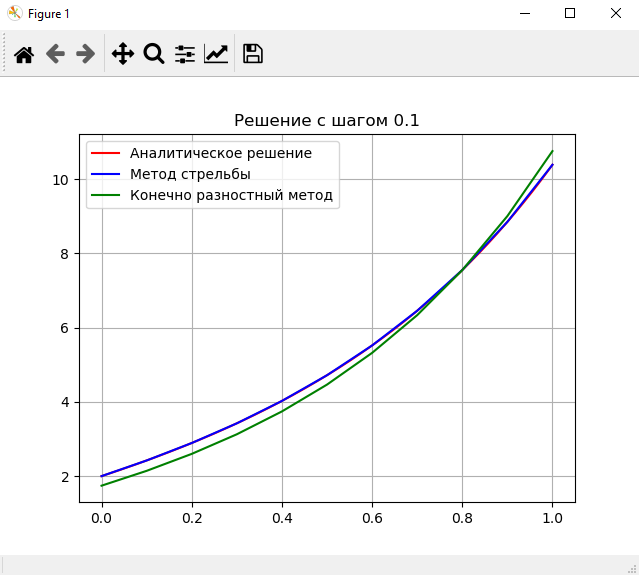
1. **Код (Python)**

import argparse  
import json  
import sympy  
import matplotlib.pyplot as plt  
from numpy import \*  
from sympy.parsing.sympy\_parser import parse\_expr  
from utils import TridiagonalMatrix, Vector  
from runge\_kutta import method\_runge\_kutta, get\_points  
from rtm import rtm  
  
  
def get\_n(n\_prev, n, res\_prev, res\_der\_prev, res, res\_der, delta, gamma, cond\_1):  
 y\_der = res\_der\_prev[-1]  
 y = (cond\_1 - gamma \* y\_der) / delta  
 phi\_prev = y - res\_prev[-1]  
  
 y\_der = res\_der[-1]  
 y = (cond\_1 - gamma \* y\_der) / delta  
 phi = y - res[-1]  
  
 return n - phi \* (n - n\_prev) / (phi - phi\_prev)  
  
  
def check\_end(res, res\_der, delta, gamma, cond\_1, eps):  
 y\_der = res\_der[-1]  
 y = (cond\_1 - gamma \* y\_der) / delta  
 phi = y - res[-1]  
 return abs(phi) < eps  
  
  
def finite\_difference(pt, coeffs, alpha, beta, delta, gamma, step, cond\_0, cond\_1):  
 x = sympy.Symbol('x')  
 p\_exp = parse\_expr(f'({coeffs[1]}) / ({coeffs[0]})')  
 q\_exp = parse\_expr(f'({coeffs[2]}) / ({coeffs[0]})')  
  
 p = sympy.lambdify(x, p\_exp)  
 q = sympy.lambdify(x, q\_exp)  
  
 sz = len(pt) - 1  
 a = [0.0] + [1 - p(pt[i]) \* step / 2 for i in range(0, sz - 1)] + [- gamma]  
 b = [alpha \* step - beta] + [-2 + q(pt[i]) \* step \*\* 2 for i in range(0, sz - 1)] + [delta \* step + gamma]  
 c = [beta] + [1 + p(pt[i]) \* step / 2 for i in range(0, sz - 1)] + [0.0]  
 d = [cond\_0 \* step] + [0 for \_ in range(0, sz - 1)] + [cond\_1 \* step]  
  
 matr = TridiagonalMatrix(sz + 1)  
 matr.A = a  
 matr.B = b  
 matr.C = c  
  
 vec = Vector(sz + 1)  
 vec.vector = d  
  
 return rtm(matr, vec)  
  
  
def shooting\_method(points, coeffs, alpha, beta, delta, gamma, cond\_0, cond\_1, step, eps):  
 f\_exp = parse\_expr('z')  
 g\_exp = parse\_expr(f'-({coeffs[1]} \* z + {coeffs[2]} \* y + {coeffs[3]}) / ({coeffs[0]})')  
  
 if coeffs[0] == 'x' and points[0] == 0.0:  
 points[0] += eps  
  
 x = sympy.Symbol('x')  
 y = sympy.Symbol('y')  
 z = sympy.Symbol('z')  
 f = sympy.lambdify([x, y, z], f\_exp)  
 g = sympy.lambdify([x, y, z], g\_exp)  
  
 n\_prev, n = 1.0, 0.8  
  
 y\_der = (cond\_0 - alpha \* n\_prev) / beta  
 res\_prev, res\_der\_prev = method\_runge\_kutta(points, n\_prev, y\_der, step, f, g)  
  
 y\_der = (cond\_0 - alpha \* n) / beta  
 res, res\_der = method\_runge\_kutta(points, n, y\_der, step, f, g)  
  
 while not check\_end(res, res\_der, delta, gamma, cond\_1, eps):  
 n\_prev, n = n, get\_n(n\_prev, n, res\_prev, res\_der\_prev, res, res\_der, delta, gamma, cond\_1)  
 res\_prev = res  
 y\_der = (cond\_0 - alpha \* n) / beta  
 res, res\_der = method\_runge\_kutta(points, n, y\_der, step, f, g)  
 return res  
  
  
def get\_err\_rr(res\_1, res\_2):  
 return (sum([(res\_1[i] - res\_2[2 \* i]) \*\* 2 for i in range(0, len(res\_1))]) \*\* 0.5) / (2 \*\* 1 - 1)  
  
  
def get\_err(res, val):  
 return sum([(res[i] - val[i]) \*\* 2 for i in range(0, len(res))]) \*\* 0.5  
  
  
def print\_results(shooting, fin\_diff, exact, err\_rr\_d, err\_d, points):  
 print('{0:25s}'.format('Точки'), end='')  
 for point in points:  
 print('{0:10.2f}'.format(point), end='')  
 print('\n' + '-' \* 136)  
  
 print('{0:25s}'.format('Метод стрельбы'), end='')  
 for val in shooting:  
 print('{0:10.5f}'.format(val), end='')  
  
 print('\n{0:25s}'.format('Конечно разностный метод'), end='')  
 for val in fin\_diff:  
 print('{0:10.5f}'.format(val), end='')  
  
 print('\n{0:25s}'.format('Точное решение'), end='')  
 for val in exact:  
 print('{0:10.5f}'.format(val), end='')  
  
 print('\n' + '-' \* 52 + 'Ошибки по методу Рунге-Ромберга' + '-' \* 52)  
 print('{0:25s}'.format('Метод стрельбы'), end='')  
 print('{0:15.12f}'.format(err\_rr\_d['shooting']), end='')  
  
 print('\n{0:25s}'.format('Конечно разностный метод'), end='')  
 print('{0:15.12f}'.format(err\_rr\_d['finite\_difference']), end='')  
  
 print('\n' + '-' \* 65 + 'Ошибки' + '-' \* 65)  
 print('{0:25s}'.format('Метод стрельбы'), end='')  
 print('{0:15.12f}'.format(err\_d['shooting']), end='')  
  
 print('\n{0:25s}'.format('Конечно разностный метод'), end='')  
 print('{0:15.12f}'.format(err\_d['finite\_difference']), end='')  
  
  
def graph(points, val\_shooting, val\_finite\_difference, f, step):  
 plt.subplots()  
 x = linspace(0, 1, 100000)  
 plt.plot(x, f(x), color='r', label='Аналитическое решение')  
 plt.plot(points, val\_shooting, color='b', label='Метод стрельбы')  
 plt.plot(points, val\_finite\_difference, color='g', label='Конечно разностный метод')  
 plt.title(f'Решение с шагом {step}')  
 plt.legend()  
 plt.grid()  
 plt.show()  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 parser = argparse.ArgumentParser()  
 parser.add\_argument('--input', required=True)  
 args = parser.parse\_args()  
  
 with open(args.input, 'r') as f:  
 data = json.load(f)  
 Coefficients = data['coefficients']  
 Alpha, Beta = data['coefficients\_1']  
 Delta, Gamma = data['coefficients\_2']  
 Cond\_0 = data['condition\_0']  
 Cond\_1 = data['condition\_1']  
 Interval = data['interval']  
 H = data['step']  
 Eps = data['eps']  
 Solution = data['solution']  
  
 Cond\_0 = float(parse\_expr(Cond\_0))  
 Cond\_1 = float(parse\_expr(Cond\_1))  
  
 x = sympy.Symbol('x')  
 f\_exp = parse\_expr(Solution)  
 f = sympy.lambdify(x, f\_exp)  
  
 Points = get\_points(\*Interval, H)  
 Points2 = get\_points(\*Interval, H / 2)  
  
 Exact = []  
 for Point in Points:  
 Exact.append(f(Point))  
  
 res\_shooting = shooting\_method(Points, Coefficients, Alpha, Beta, Delta, Gamma, Cond\_0, Cond\_1, H, Eps)  
 res\_finite\_difference = finite\_difference(Points, Coefficients, Alpha, Beta, Delta, Gamma, H, Cond\_0, Cond\_1)  
  
 res\_shooting2 = shooting\_method(Points2, Coefficients, Alpha, Beta, Delta, Gamma, Cond\_0, Cond\_1, H / 2, Eps)  
 res\_finite\_difference2 = finite\_difference(Points2, Coefficients, Alpha, Beta, Delta, Gamma, H / 2, Cond\_0, Cond\_1)  
  
 err\_rr\_shooting = get\_err\_rr(res\_shooting, res\_shooting2)  
 err\_rr\_finite\_difference = get\_err\_rr(res\_finite\_difference, res\_finite\_difference2)  
  
 err\_shooting = get\_err(res\_shooting, Exact)  
 err\_finite\_difference = get\_err(res\_finite\_difference, Exact)  
  
 err\_rr = {'shooting': err\_rr\_shooting, 'finite\_difference': err\_rr\_finite\_difference}  
 err = {'shooting': err\_shooting, 'finite\_difference': err\_finite\_difference}  
  
 graph(Points, res\_shooting, res\_finite\_difference, f, H)  
 print\_results(res\_shooting, res\_finite\_difference, Exact, err\_rr, err, Points)

1. **Файл с данными (data4.2)**

{  
 "coefficients": ["x", "-(2\*x+1)", "2", "0"],  
 "coefficients\_1": [0, 1],  
 "coefficients\_2": [-2, 1],  
 "condition\_0": "4",  
 "condition\_1": "-4",  
 "interval": [0, 1],  
 "step": 0.1,  
 "eps": 0.000001,  
 "solution": "2 \* x + 1 + exp(2 \* x)"  
}

1. **Результаты работы**

****