**ИНСТИТУТ ТРАНСПОРТА И СВЯЗИ**



ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ НАУК И ТЕЛЕКОММУНИКАЦИЙ

КАФЕДРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ КОМПЬЮТЕРНЫХ СИСТЕМ

**Лабораторная работа №2**

по дисциплине

„Численные методы”

Тема: «Метод приближения функции. Интерполяция и аппроксимация.»

Выполнил: Денис Белов, Андрей Савкин, Евгений Хрущ

Рига.

2020.

1. Формулировка задания

Необходимо реализовать аппроксимацию по методу наименьших квадратов(МНК) и сплайн интерполяцию. Графически отобразить результат работы методов. После оба метода надо сравнить с одними и теми же данными на одном графике, и сравнить полученные результаты.

1. Аппроксимация по методу наименьших квадратов (МНК)

class LeastSquaresApproximator:

    def \_\_init\_\_(self, vectorX: list, vectorY: list, k\_approx\_order: int = 2, ftype: str = "auto", makeplot=False, customfunc=None, resolution=10):

        # Initialize class fields

        self.k\_approx\_order = k\_approx\_order

        self.resolution = resolution

        self.vectorX = vectorX

        self.vectorY = vectorY

        # Set n to the size of known X vector

        self.n = len(vectorX)

        # Getting matrix and b coefficients vector from the power basis

        self.matrixA, self.vectorB = self.get\_power\_basis\_matrix()

        # Making the one whole matrix from the matrixA and vectorB to pass into the Gauss Elimination solving function

        self.whole\_matrix = mh.only\_append\_vectorB(

            self.matrixA.tolist(), mh.unpack\_vector(self.vectorB.tolist()))

        # Solving matrix with Gauss Elimination, getting the X vector of solutions for power basis matrix

        self.solution\_vectorX = ge.gauss\_elimination(

            matrix=self.whole\_matrix,

            vars=['x'+str(i) for i in range(0, self.n)],

            matrix\_name="")

        print("solution\_vectorX = ", self.solution\_vectorX)

        print("vectorX[0]=",self.vectorX[0])

        # Initializing vectorF and vector\_deltaF as empty lists

        self.vectorF = []

        self.vector\_deltaF = []

        # Then we need to calculate the approximation vector from the solution vector

        # using the choosen type of approximation function (auto by default)

        if ftype == "linear":

            def linfunc(a, x):

                print("a \* x + sum(solutions) = ", a, "\*", x, "+",

                sum(self.solution\_vectorX[:-1]), "=", a \* x + sum(self.solution\_vectorX[:-1]))

                return a \* x + sum(self.solution\_vectorX[:-1])

            self.vectorF = [linfunc(self.solution\_vectorX[-1], self.vectorX[i]) for i in range(0, self.n)]

            self.vector\_deltaF = [(self.vectorY[i] - self.vectorF[i]) \*\* 2 for i in range(0, self.n)]

        elif ftype == "exponential":

            exit

        elif ftype == "custom":

            self.vectorF = [customfunc(self.solution\_vectorX, self.vectorX[i]) for i in range(0, self.n)]

            self.vector\_deltaF = [(self.vectorY[i] - self.vectorF[i]) \*\* 2 for i in range(0, self.n)]

        elif ftype == "auto":

            print("Using auto (default) approximation basis function")

            self.vectorF = [self.autofunc(self.solution\_vectorX, self.vectorX[i]) for i in range(0, self.n)]

            self.vector\_deltaF = [(self.vectorY[i] - self.vectorF[i]) \*\* 2 for i in range(0, self.n)]

        print("Sum of vector\_deltaF =", sum(self.vector\_deltaF))

        print("vectorF:\n", self.vectorF, "\nvector\_deltaF:\n", self.vector\_deltaF)

        # Getting interpolated vectors of X and Y to build smooth curve (count of points depends on the resolution parameter)

        self.interpolated\_vectorX, self.interpolated\_vectorY = self.get\_interpolated\_xy\_vectors()

        if makeplot:

            self.make\_plot()

    def autofunc(self, solvec, x):

        return sum([solvec[i] \* (x \*\* i) for i in range(0, len(solvec))])

    def get\_interpolated\_xy\_vectors(self):

        # Initializing empty out vectors

        out\_vectorX = list()

        out\_vectorY = list()

        # Getting values interpolated between x[i] and x[i+1] points

        for i in range(0, len(self.vectorX) - 1):

            current\_step = (self.vectorX[i + 1] - self.vectorX[i]) / self.resolution

            current\_x = self.vectorX[i]

            for j in range(0, self.resolution):

                # Calculating interpolated Y point and adding it to the out Y vector

                out\_vectorY.append(self.autofunc(self.solution\_vectorX, current\_x))

                out\_vectorX.append(current\_x)

                # Adding the current step value to the current\_x to get the next x value

                current\_x += current\_step

        # Adding the last point X and Y coords to the out\_vectors

        out\_vectorX.append(self.vectorX[-1])

        out\_vectorY.append(self.vectorF[-1])

        return out\_vectorX, out\_vectorY

    def make\_plot(self):

        plt.title("Least Squares approximation with k = " + str(self.k\_approx\_order))

        plt.plot(self.vectorX, self.vectorY, 'bs', self.vectorX, self.vectorF, 'g--', self.vectorX, self.vectorF, 'g^')

        plt.plot(self.interpolated\_vectorX, self.interpolated\_vectorY, 'y-')

        plt.xlabel("X values")

        plt.ylabel("Y values")

        for i in range(0, len(self.vectorX) - 1):

            plt.plot([self.vectorX[i], self.vectorX[i]], [self.vectorY[i], self.vectorF[i]], 'r--')

        plt.show()

    def get\_power\_basis\_matrix(self):

        current\_k = self.k\_approx\_order

        variables\_count = len(self.vectorX)

        out\_matrix = np.ndarray(shape=(current\_k + 1, current\_k + 1))

        out\_b\_coefficients = np.ndarray(shape=(current\_k + 1, 1))

        # Filling the out\_matrix

        for i in range(0, current\_k + 1):

            # Iterating over the elements of a row

            for j in range(0, current\_k + 1):

                # Calculating each element of the matrix A

                out\_matrix[i, j] = sum([x \*\* (j + i) for x in self.vectorX])

                out\_b\_coefficients[i, 0] = sum([(self.vectorX[index] \*\* i) \* self.vectorY[index] for index in range(0, len(self.vectorX))])

        return out\_matrix, out\_b\_coefficients

3) Сплайн интерполяция

import matplotlib.pyplot as plt

import numpy as np

import copy

import os, sys

sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(\_\_file\_\_))))

import lab1\_gauss\_elimination.gauss\_elimination as ge

import lab1\_gauss\_seidel.gauss\_seidel as gs

import matrix\_helpers as mh

class CubicSplineInterpolator:

    def \_\_init\_\_(self, known\_vectorX: list, known\_vectorY: list, known\_points: list, vars = []):

        self.known\_points = known\_points

        self.points\_count = len(known\_points if known\_points is not None else known\_vectorX)

        self.spline\_count = self.points\_count - 1

        # Get the X and Y vectors from the known\_points, or use known\_vectorX and known\_vectorY if provided

        self.vectorX = known\_vectorX if known\_vectorX is not None else [elem[0] for elem in known\_points]

        self.vectorY = known\_vectorY if known\_vectorY is not None else [elem[1] for elem in known\_points]

        self.vectorH = [self.vectorX[i] - self.vectorX[i - 1] for i in range(1, self.points\_count)]

        # Initializing vector of undefined variables A

        self.coefficientsA = copy.deepcopy(self.vectorY)

        # Get the dirrerences of Y, to easily iterate

        self.deltaY = np.diff(self.vectorY)

        # Initializing matrix A and getting the calculated C coefficients from solving the system

        self.matrixA, self.vectorC = self.construct\_tridiagonal\_matrix()

        # Solving tridiagonal matrix and getting calculated unknown C coefficients vector

        self.coefficientsC = self.solve\_tridiagonal\_matrix(self.matrixA, self.vectorC, vars)

        self.coefficientsC = np.array(self.coefficientsC)

        self.coefficientsB = np.zeros(shape=(self.points\_count - 1, 1))

        self.coefficientsD = np.zeros(shape=(self.points\_count - 1, 1))

        # Iterating to calculate unknown B and D coefficients from the C coefficients vector

        for i in range(0, len(self.coefficientsD)):

            self.coefficientsD[i] = (self.coefficientsC[i + 1] - self.coefficientsC[i]) / (3 \* self.vectorH[i])

            self.coefficientsB[i] = (self.deltaY[i]/self.vectorH[i]) - (self.vectorH[i]/3) \* (2\*self.coefficientsC[i] + self.coefficientsC[i+1])

        self.print\_results\_table()

    def get\_xy(self, resolution=10, makeplot=False, showpoints=False):

        self.resolution = resolution

        out\_vectorX = list()

        out\_vectorY = list()

        for i in range(0, self.spline\_count):

            current\_step = self.vectorH[i] / resolution

            current\_x = self.vectorX[i]

            for j in range(0, resolution):

                out\_vectorY.append(self.get\_sx(current\_x, i))

                out\_vectorX.append(current\_x)

                current\_x += current\_step

        # Adding the last point X and Y coords to the out\_vectors

        out\_vectorX.append(self.vectorX[-1])

        out\_vectorY.append(self.vectorY[-1])

        if makeplot:

            if showpoints:

                plt.plot(out\_vectorX, out\_vectorY, "bo-", markersize=2)

            else:

                plt.plot(out\_vectorX, out\_vectorY, "b-")

            plt.plot(self.vectorX, self.vectorY, "yo")

            plt.show()

        return out\_vectorX, out\_vectorY

    def print\_sx(self, x: float, spline\_index: int):

        print(f'S{spline\_index}({x}) = {self.get\_sx(x, spline\_index)}')

    def get\_sx(self, x: float, spline\_index: int):

        i = spline\_index

        previous\_x = self.vectorX[spline\_index]

        h = x - previous\_x

        return self.coefficientsA[i] + self.coefficientsB[i][0] \* h + self.coefficientsC[i] \* (h \*\* 2) + self.coefficientsD[i][0] \* (h \*\* 3)

    def solve\_tridiagonal\_matrix(self, matrixA, vectorB, vars) -> list:

        n = len(matrixA)

        # Initialize two 1D vectors to hold Ai and Bi values (or ak and bk, as in the pdf)

        alphas = [0] \* n

        betas = [0] \* n

        alphas[0] = matrixA[0, 1] / matrixA[0, 0]

        betas[0] = vectorB[0][0] / matrixA[0, 0]

        for i in range(1, n - 1):

            alphas[i] = matrixA[i, i + 1] / (matrixA[i, i] - matrixA[i, i-1] \* alphas[i-1])

            betas[i] = (vectorB[i][0] - matrixA[i, i-1] \* betas[i-1]) / (matrixA[i, i] - matrixA[i, i-1] \* alphas[i - 1])

        # Initialize vector of out X results, where x\_n = beta\_n

        X = [0] \* n

        X[n - 1] = betas[n - 1]

        print("alphas=",alphas)

        print("betas=",betas)

        # Backward substitution

        for i in range(n - 2, -1, -1):

            X[i] = betas[i] - alphas[i] \* X[i + 1]

        return X

    def print\_results\_table(self):

        A = list(self.coefficientsA)

        B = list(self.coefficientsB.tolist())

        C = list(self.coefficientsC.tolist())

        D = list(self.coefficientsD.tolist())

        print(f'{"Spline number":<16} | {"ai":<16} | {"bi":<16} | {"ci":<16} | {"di":<16}')

        print('{:-<80}'.format(""))

        for i in range(0, self.spline\_count):

            a\_val = round(A[i], 5)

            a\_len = len(str(a\_val))

            b\_val = round(B[i][0], 5)

            b\_len = len(str(b\_val))

            c\_val = round(C[i], 5)

            c\_len = len(str(c\_val))

            d\_val = round(D[i][0], 5)

            d\_len = len(str(d\_val))

            print(

                f'{i:<16} | {a\_val}{"":<{16 - a\_len}} | {b\_val}{"":<{16 - b\_len}} | {c\_val}{"":<{16 - c\_len}} | {d\_val}{"":<{16 - d\_len}}')

    def construct\_tridiagonal\_matrix(self):

        # --- Constructing the tridiagonal matrix A as hi\*ci + 2\*(hi-1 + hi)\*ci + hi\*ci+1

        matrixA = np.zeros(shape=(self.points\_count, self.points\_count))

        vectorB = np.zeros(shape=(self.points\_count, 1))

        matrixA[0, 0] = 1

        matrixA[-1, -1] = 1

        for i in range(1, self.points\_count - 1):

            matrixA[i, i-1] = self.vectorH[i-1]

            matrixA[i, i+1] = self.vectorH[i]

            matrixA[i, i] = 2\*(self.vectorH[i-1]+self.vectorH[i])

            # Get vector B (C coefficients)

            vectorB[i, 0] = 3\*(self.deltaY[i]/self.vectorH[i] - self.deltaY[i-1]/self.vectorH[i-1])

        return matrixA, vectorB

4) Вывод