**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,

    print\_matrix=False, customstep=None):

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

        self.resolution = resolution

        self.vectorX = vectorX

        self.vectorY = vectorY

        self.customfunc = customfunc

        self.customstep = customstep

        self.n = len(vectorX)

        # Getting matrix and b coefficients vector from the power basis

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

        self.matrixAB = mh.append\_vectorB\_to\_matrixA(

            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.GaussElimination(

            matrixAB=self.matrixAB,

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

            matrix\_name="",

            print\_only\_results=True).solution\_vectorX

        self.vectorF = []

        self.vector\_deltaF = []

        if ftype == "linear\_deprecated":

            def linfunc(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 in ("auto", "", None):

            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)]

        else:

            raise ValueError("You need to choose ftype!")

        # 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()

        # Constructing the plot

        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\_x\_from\_y\_estimated(self, y, lower\_border = -0.5, upper\_border = 0.5, max\_iterations=500, tolerance=0.0001):

        return roots.RootFinder(

            self.autofunc if self.customfunc is None else self.customfunc,

            solution\_vectorX=self.solution\_vectorX

        ).bisection(

            lower=y + lower\_border,

            upper=y + upper\_border,

            max\_iterations=max\_iterations,

            tolerance=tolerance

        )

    def get\_x\_from\_y\_closest(self, y, vectorX=None, vectorY=None):

        # Get the closest known x from the vectorX and the corresponding y from the vectorY

        closest\_list = []

        for v in self.interpolated\_vectorY if vectorY is None else vectorY:

            closest\_list.append(abs(y - v))

        closest\_y\_index = closest\_list.index(min(closest\_list))

        closest\_x = self.interpolated\_vectorX[closest\_y\_index] if vectorX is None else vectorX[closest\_y\_index]

        return closest\_x, closest\_y\_index

    def get\_closest\_boundary\_points(self, y):

        closest\_x = self.get\_x\_from\_y\_closest(y)

        index = 0

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

            if self.vectorX[i] > closest\_x[0]:

                index = i

                break

        return self.vectorX[index - 1], self.vectorF[index - 1], self.vectorX[index], self.vectorF[index]

    def get\_x\_from\_y\_interpolated(self, y, resolution = 100):

        # Get the closest known x from the reinterpolated vector X and the corresponding y from the reinterpolated vector Y

        xvec, yvec = self.get\_interpolated\_xy\_vectors(resolution)

        new\_closest = self.get\_x\_from\_y\_closest(y, xvec, yvec)

        return xvec[new\_closest[1]]

    def interpolate(self, currentX, deltaX, resolution):

        interpolated\_vectorX = list()

        interpolated\_vectorY = list()

        step = deltaX / resolution

        for i in range(0, resolution):

            new\_y = self.autofunc(self.solution\_vectorX, currentX) if self.customfunc is None else self.customfunc(self.solution\_vectorX, currentX)

            interpolated\_vectorY.append(new\_y)

            interpolated\_vectorX.append(currentX)

            currentX += step

        return interpolated\_vectorX, interpolated\_vectorY

    def get\_y\_from\_x(self, x):

        if self.customfunc is None:

            return self.autofunc(self.solution\_vectorX, x)

        else:

            return self.customfunc(self.solution\_vectorX, x)

    def get\_interpolated\_xy\_vectors(self, custom\_resolution=None):

        # Initializing empty out vectors

        out\_vectorX = list()

        out\_vectorY = list()

        resolution = self.resolution if custom\_resolution is None else custom\_resolution

        # 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]) / resolution if self.customstep is None else self.customstep

            current\_x = self.vectorX[i]

            # Adding each point to the out vector

            for j in range(0, resolution):

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

                    if self.customfunc is None else self.customfunc(self.solution\_vectorX, current\_x))

                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.vectorF[-1])

        return out\_vectorX, out\_vectorY

    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))

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

            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])

                # Calculating each B coefficient

                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

График y=sin(5x) \* ex

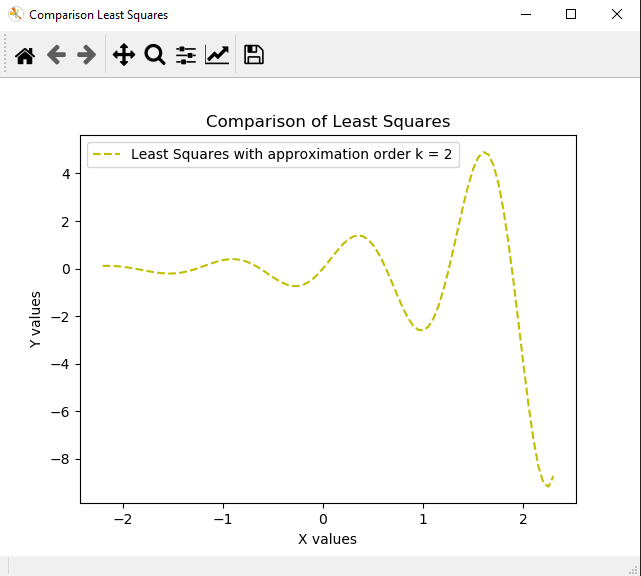
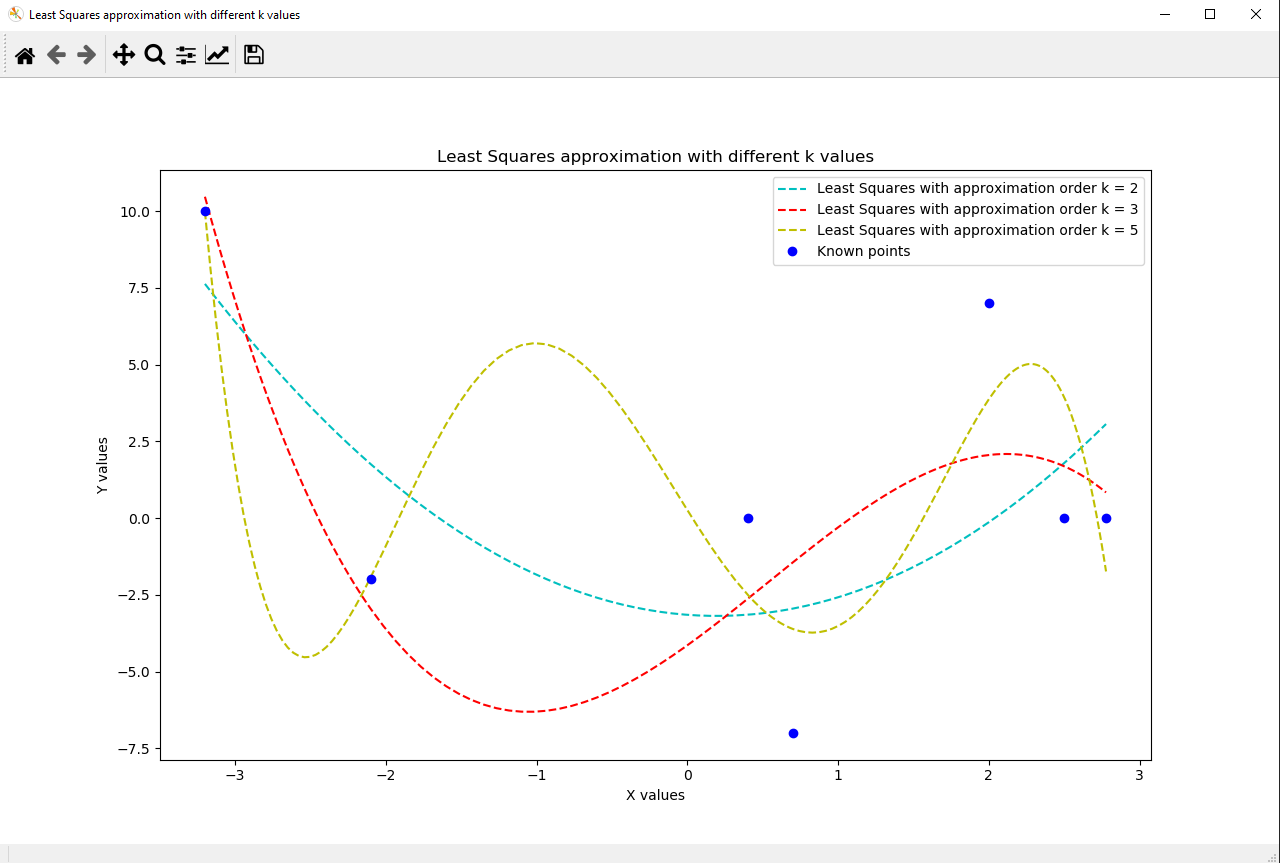


График одной функции при трёх различных значениях порядка полинома:



3) **Сплайн интерполяция**

class CubicSplineInterpolator:

    def \_\_init\_\_(self, known\_vectorX: list, known\_vectorY: list, known\_points: list=None, 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

        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)]

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

        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()

        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])

    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])

        return out\_vectorX, out\_vectorY

    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)

        alphas = [0] \* n

        betas = [0] \* n

        # Initializing a1 and b1

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

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

        # Calculate values for each alpha and beta

        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])

        X = [0] \* n

        X[n - 1] = betas[n - 1]

        # Backward substitution

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

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

        return X

    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))

        # Set the first element (the most upper-left element in the matrix) to 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]

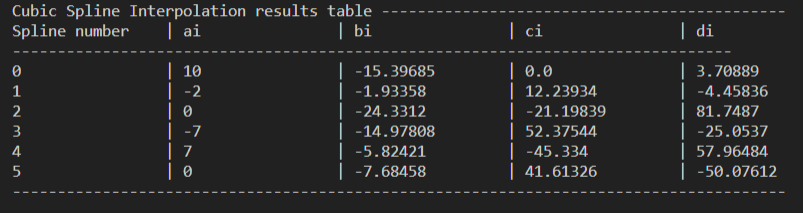
            matrixA[i, i] = 2\*(self.vectorH[i-1]+self.vectorH[i])

            # Get vector B (C coefficients)

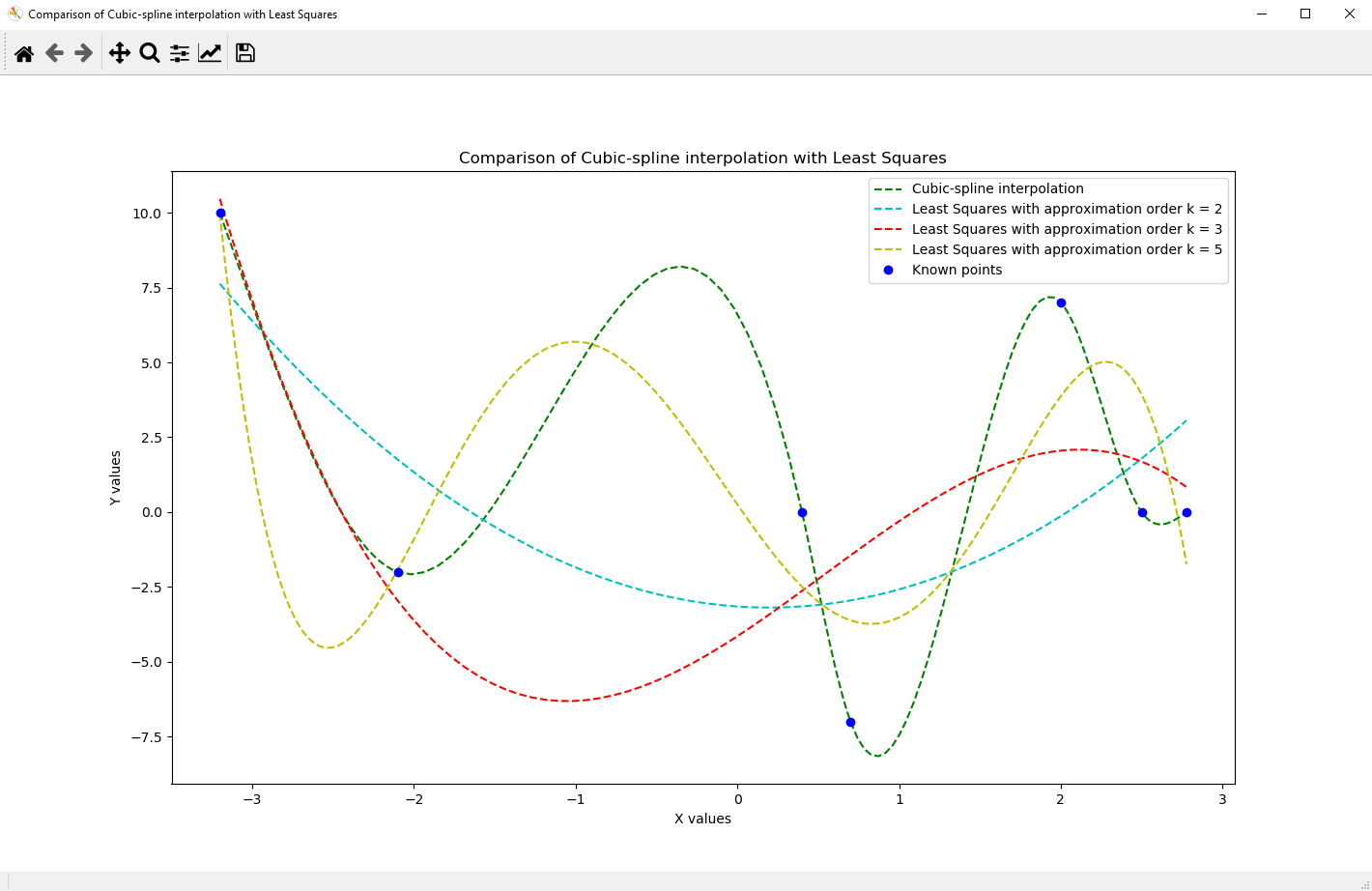
            vectorB[i, 0] = 3\*(self.deltaY[i]/self.vectorH[i] - self.deltaY[i-1]/self.vectorH[i-1])

        return matrixA, vectorB

Информация для вывода:



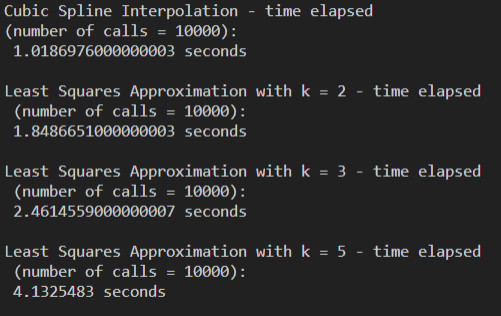
**4) Графическое сравнение двух методов**



**5) Выводы:**

Сложность метода МНК равняется O(n3), что больше, если сравнивать сложность сплайн интерполяции, которая равна O(n). На реализацию метода МНК потребовалось меньше времени, чем на сплайн интерполяция.

Время затраченное на выполнение:



Таким образом получается, что время МНК сильно возрастает.

Сплайн интерполяция более гладка, тянется от точки к точке, гибкая. Используется для анализа сигналов, например, георадаров, или отображения интенсивности, например, света.

МНК используется в статистике для описания данных, для техник предсказания в и получения максимального правдоподобия. Для выявления корреляции. Метод позволяет проверить функцию на то как она входит в данные.