

ФАКУЛЬТЕТ: Информатика и системы управления

КАФЕДРА: Программное обеспечение ЭВМ и информационные технологии

**Лабораторная работа №1**

**Тема** Построение и программная реализация алгоритма полиномиальной интерполяции табличных функций.

**Студент** Зайцева А. А.

**Группа** ИУ7 – 42Б

**Оценка (баллы) \_\_\_\_\_\_\_**

**Преподаватель** Градов В. М.

Москва.

2021 г

**Цель работы.** Получение навыков построения алгоритма интерполяции таблично заданных функций полиномами Ньютона и Эрмита.

1. Исходные данные
2. Таблица функции и её производных

|  |  |  |
| --- | --- | --- |
| x | y | y` |
| 0 | 1.000000 | -1.000000 |
| 0.15 | 0.838771 | -1.14944 |
| 0.30 | 0.655336 | -1.29552 |
| 0.45 | 0.450447 | -1.43497 |
| 0.60 | 0.225336 | -1.56464 |
| 0.75 | -0.018310 | -1.68164 |
| 0.90 | -0.278390 | -1.78333 |
| 1.05 | -0.552430 | -1.86742 |

1. Степень аппроксимирующего полинома - n.
2. Значение аргумента, для которого выполняется интерполяция.
3. Код программы

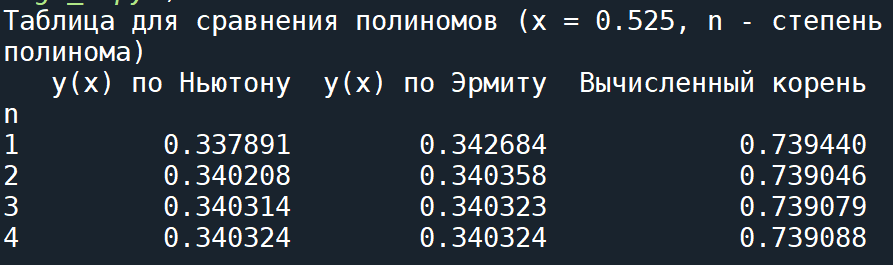
functions.py

#Синхронная сортировка двух массивов по возрастанию элементов в первом  
#Возвращает отсортированные массивы  
def sync\_sort2(x, y):  
    indices = sorted(range(len(x)), key=lambda i: x[i])  
    x = [x[i] for i in indices]  
    y = [y[i] for i in indices]  
    return x, y  
  
#Синхронная сортировка трех массивов по возрастанию элементов в первом  
#Возвращает отсортированные массивы  
def sync\_sort3(x, y, yd):  
    indices = sorted(range(len(x)), key=lambda i: x[i])  
    x = [x[i] for i in indices]  
    y = [y[i] for i in indices]  
    yd = [yd[i] for i in indices]  
    return x, y, yd  
  
#Выбор узлов из массивов x, y для полинома Ньютона степени n и вычисления y(x0)  
#Возвращает массивы выбранных узлов  
def prepare\_arrays\_newton(x, y, n, x0):  
    #необходимо выбрать n + 1 узлов  
    need\_to\_take = n + 1  
    if need\_to\_take > len(x):  
        print('ОШИБКА: не хватает точек для построения полинома Ньютона')  
    #сортируем (чтобы верно выбрать ближайшие узлы)  
    x, y = sync\_sort2(x, y)  
    #находим точку в таблице, которая ближе всего к x0,   
    #чтобы выбирать узлы вокруг неё  
    closest\_to\_x0\_i = (sorted(range(len(x)), key=lambda i: abs(x[i] - x0)))[0]  
    #определяем индексы необходимых узлов в исходных массивах  
    #если не удаетсся равномерно распределить узлы вокруг точки, выбираем из  
    #того, что есть  
    from\_i = closest\_to\_x0\_i - need\_to\_take // 2   
    if from\_i < 0:  
        from\_i = 0     
    to\_i = from\_i + need\_to\_take   
    if to\_i > len(x):  
        to\_i = len(x)  
        from\_i = to\_i - need\_to\_take  
    #формируем массивы из выбранных узлов      
    x\_new = x[from\_i : to\_i]  
    y\_new = y[from\_i : to\_i]  
    return x\_new, y\_new  
  
#Выбор узлов из массивов x, y, yd для полинома Эрмита степени n и вычисления y(x0)  
#Возвращает массивы выбранных узлов  
def prepare\_arrays\_ermit(x, y, yd, n, x0):  
    #при построении полинома Эрмита используются как значения функции, так и   
    #значения производных, поэтому необходимое количество точек вычисляется так  
    need\_to\_take = (n // 2) + 1  
    if need\_to\_take > len(x):  
        print('ОШИБКА: не хватает точек для построения полинома Эрмита')  
    #сортируем (чтобы верно выбрать ближайшие узлы)  
    x, y, yd = sync\_sort3(x, y, yd)  
    #находим точку в таблице, которая ближе всего к x0,   
    #чтобы выбирать узлы вокруг неё  
    closest\_to\_x0\_i = (sorted(range(len(x)), key=lambda i: abs(x[i] - x0)))[0]  
    #определяем индексы необходимых узлов в исходных массивах  
    #если не удаетсся равномерно распределить узлы вокруг точки, выбираем из  
    #того, что есть  
    from\_i = closest\_to\_x0\_i - need\_to\_take // 2   
    if from\_i < 0:  
        from\_i = 0     
    to\_i = from\_i + need\_to\_take  
    if to\_i > len(x):  
        to\_i = len(x)  
        from\_i = to\_i - need\_to\_take  
    #формируем массивы из выбранных узлов  
    #при этом таблицу из x, y преобразуем к виду, удобному для отыскания  
    #разделенных разностей согласно процедуре обработки полинома Ньютона.  
    x\_new = []  
    y\_new = []  
    yd\_new = []  
    for i in range(from\_i, to\_i):  
        x\_new.append(x[i])  
        x\_new.append(x[i])  
        y\_new.append(y[i])  
        y\_new.append(y[i])  
        yd\_new.append(yd[i])  
    return x\_new, y\_new, yd\_new  
  
#Поиск коэффициентов для интерполяционного полинома Ньютона n-й степени  
#с использованием масивов узлов x, y  
#Возвращает массив коэффициентов  
def find\_coeffs\_newton(x, y, n):  
    coeffs = [y[0]] #первое слагаемое - y(x[0])  
    #step - шаг (номер столбца в таблице после y)  
    for step in range(n):  
        #i - строка столбца  
        for i in range(n - step):  
            #вычисление разделенной разности для y от (step + 2) переменных  
            y[i] = (y[i + 1] - y[i])/(x[i + step + 1] - x[i])   
        coeffs.append(y[0])  
    return coeffs  
  
#Поиск коэффициентов для интерполяционного полинома Эрмита n-й степени  
#с использованием масивов узлов x, y, yd  
#Формально строим полином Ньютона по (n // 2) + 1 узлам, каждый из которых   
#повторяется дважды  
#Возвращает массив коэффициентов  
def find\_coeffs\_ermit(x, y, yd, n):  
    coeffs = [y[0]] #первое слагаемое - y(x[0])  
    #step - шаг (номер столбца в таблице после y)  
    for step in range(n):  
        #i - строка столбца  
        for i in range(n - step):  
            #вычисление разделенной разности для y от (step + 2) переменных  
            #на нулевом шаге (кратность 2) формулы для разделенных разностей   
            #получаются предельным переходом  
            if (step == 0) and (i % 2 == 0):  
                y[i] = yd[i // 2]  
            else:  
                y[i] = (y[i + 1] - y[i])/(x[i + step + 1] - x[i])  
        coeffs.append(y[0])  
    return coeffs  
  
#Вычисление полинома степени n с коэффициентами coeffs по массиву x в точке x0  
#Возвращает значение полинома в точке x0  
def count\_polynom(x, coeffs, n, x0):  
    summ = 0  
    #вычисление очередного слагаемого  
    for stage in range(n + 1):  
        summand = coeffs[stage]  
        for i in range(stage):  
            summand \*= (x0 - x[i])  
        #Формирование ответа  
        summ += summand  
    return summ  
  
#Приближенное вычисление y(x0) c помощью полинома Ньютона n-й степени  
#(объединение функций prepare\_arrays\_newton,find\_coeffs\_newton и count\_polynom)   
#Возвращает значение полинома Ньютона в точке x0     
def approximate\_newton(x, y, n, x0):  
    x\_newton, y\_newton = prepare\_arrays\_newton(x, y, n, x0)  
    coeffs = find\_coeffs\_newton(x\_newton, y\_newton, n)  
    return count\_polynom(x\_newton, coeffs, n, x0)  
  
#Нахожление корня функции с помощью обратной интерполяции   
#используя полином Ньютона.  
def find\_root\_back\_interp(x, y, n):  
    return approximate\_newton(y, x, n, 0)  
  
#Приближенное вычисление y(x0) c помощью полинома Эрмита n-й степени  
#(объединение функций prepare\_arrays\_ermit,find\_coeffs\_ermit и count\_polynom)   
#Возвращает значение полинома Эрмита в точке x0     
def approximate\_ermit(x, y, yd, n, x0):  
    x\_ermit, y\_ermit, yd\_ermit = prepare\_arrays\_ermit(x, y, yd, n, x0)  
    coeffs = find\_coeffs\_ermit(x\_ermit, y\_ermit, yd\_ermit, n)  
    return count\_polynom(x\_ermit, coeffs, n, x0)

main.py

import functions  
import pandas as ad  
  
def main():  
    #исходные данные  
    x = [0.00, 0.15, 0.30, 0.45, 0.60, 0.75, 0.90, 1.05]  
    y = [1.000000, 0.838771, 0.655336, 0.450447,  
         0.225336, -0.018310, -0.278390, -0.552430]  
    yd = [-1.000000, -1.14944, -1.29552, -1.43497,  
          -1.56464, -1.68164, -1.78333, -1.86742]  
    n\_range = range(1, 5)  
    x0 = 0.525  
      
    #получение результатов  
    comp\_table = []  
    columns = ['n','y(x) по Ньютону', 'y(x) по Эрмиту', 'Вычисленный корень']  
    for n in n\_range:  
        res\_newton = functions.approximate\_newton(x, y, n, x0)  
        res\_ermit = functions.approximate\_ermit(x, y, yd, n, x0)  
        root = functions.find\_root\_back\_interp(x, y, n)  
        comp\_table.append([n, res\_newton, res\_ermit, root])  
      
    #Вывод сравнительной таблицы  
    print('Таблица для сравнения полиномов (x = 0.525, n - степень полинома)')  
    df = pd.DataFrame(data = comp\_table, columns = columns)  
    df.index = df['n']  
    df = df.drop('n', axis = 1)  
    print(df)  
      
if \_\_name\_\_ == "\_\_main\_\_":  
    main()

1. Результаты работы
2. Значения y(x) при степенях полиномов Ньютона и Эрмита n= 1, 2, 3 и 4 при фиксированном x, например, x=0.525 (середина интервала 0.45- 0.60). Результаты свести в таблицу для сравнения полиномов.
3. Найти корень заданной выше табличной функции с помощью обратной интерполяции, используя полином Ньютона.



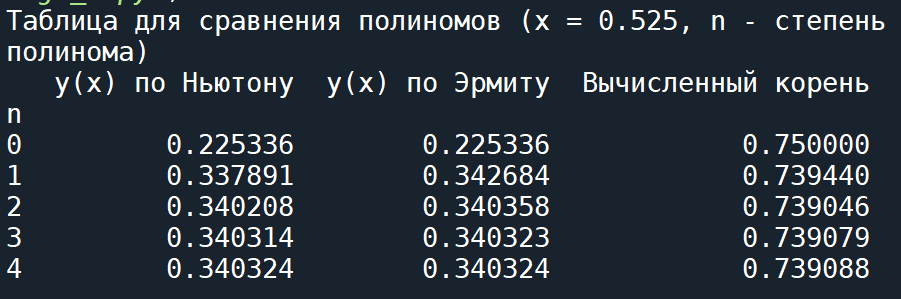
1. Вопросы при защите лабораторной работы

Ответы на вопросы дать письменно в Отчете о лабораторной работе.

1. Будет ли работать программа при степени полинома n=0?

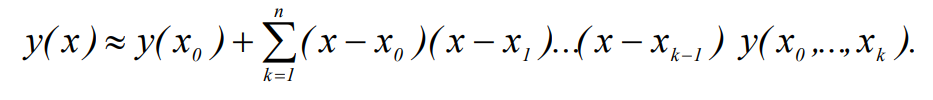
Да, тогда при вычислении y(x) с помощью полиномов Ньютона и Эрмита она вернет значение функции из таблицы в точке, которая ближе всего к x, а при вычислении корня с помощью обратной интерполяции – значение точки x из таблицы, в которой y ближе всего к нулю

Вывод программы при тех же исходных данных, с добавленным полиномом 0 степени:

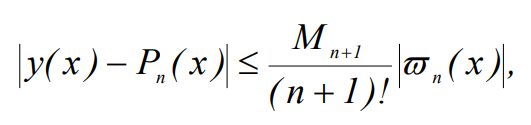


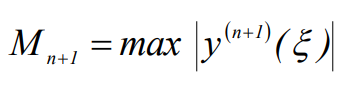
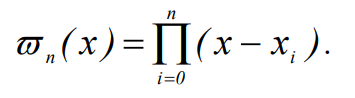
1. Как практически оценить погрешность интерполяции? Почему сложно применить для этих целей теоретическую оценку?

Чтобы практически оценить погрешность интерполяции можно воспользоваться оценкой первого отброшенного члена при вычислении многочлена Ньютона

, в котором оставляют только те члены, которые больше заданной погрешности расчетов.

Теоретическую оценку сложно применить, так как производные интерполируемой функции обычно неизвестны, а она нужна в формуле для вычисления погрешности:



где  - максимальное значение производной интерполируемой функции на отрезке между наименьшим и наибольшим из значений x0, x1, x2, … xn, а полином

1. Если в двух точках заданы значения функции и ее первых производных, то полином какой минимальной степени может быть построен на этих точках?

На этих точках можно построить полиномы Эрмита 0, 1, 2 и 3 степени или полиномы Ньютона 0 и 1 степени. Минимальная степень – 0.

1. В каком месте алгоритма построения полинома существенна информация об упорядоченности аргумента функции (возрастает, убывает)?

На первом этапе, когда необходимо сформировать конфигурацию из (n+1)-го узлов, по возможности симметрично расположенных относительно значения x.

1. Что такое выравнивающие переменные и как их применить для повышения точности интерполяции?

Если разделенные разности функции значительно меняются на протяжении нескольких интервалов сетки, то интерполяция обобщенным многочленом обычно недостаточно точна для дифференцирования этой функции. Чтобы повысить точность используется квазилинейная интерполяция, производимая при помощи выравнивающих переменных.

Оказывается, что преобразованием переменных η=η( y ) и ξ = ξ(x) можно добиться того, чтобы в новых переменных график η(ξ ) был близок к прямой хотя бы на отдельных участках. В этом случае интерполяцию проводят в переменных (η, ξ) , а затем обратным интерполированием находят yi = y(ηi).

Преобразования η(y) и ξ(x) должны быть достаточно простыми (логарифмическая, экспоненциальная, тригонометрические и некоторые другие функции). При этом надо заботиться о том, чтобы и обратное преобразование y (η) оказалось несложным. Часто встречается степенная зависимость функции от своих аргументов. В этом случае удобны преобразования типа логарифмирования.

Если  — выравнивающие переменные, то для искомой производной справедливо соотношение

https://scask.ru/archive/arch.php?path=../htm/stu.sernam/book_dig_m/files.book&file=dig_m_28.files/image1.gif

Выравнивающие преобразования подбирают несложными, чтобы их производные находились точно. Остается только численно найти  .