Ministerul Educaţiei și Cercetării

al Republicii Moldova

Universitatea Tehnică a Moldovei

**RAPORT**

Despre lucrările de laborator

la Metode Numerice

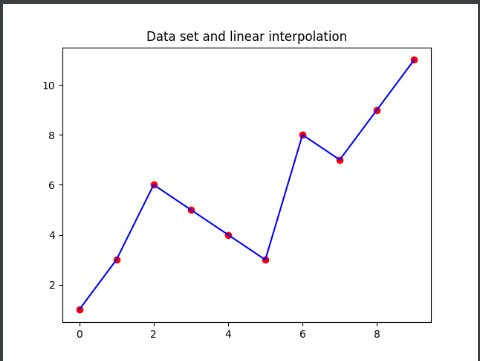
A realizat st.gr. 212 Lupașcu Felicia

A verificat: Cristofor Fistic

**1. Spline Interpolation**

Write a program that will compute spline interpolation

from scipy import interpolate  
import matplotlib.pyplot as plt  
import numpy as np  
  
y = [1, 3, 6, 5, 4, 3, 8, 7, 9, 11]  
n = len(y)  
x = range(0, n)  
  
plt.plot(x, y, 'ro')  
plt.plot(x, y, 'b')  
plt.title("Data set and linear interpolation")  
plt.show()



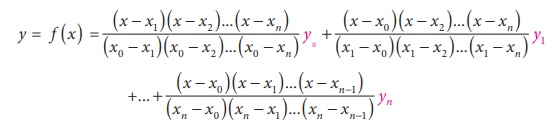
# 2. Lagrange's interpolation

Write a program that will compute Lagrange's interpolation.

# Python3 program for implementation Lagrange's Interpolation  
  
# To represent a data point corresponding to x and y = f(x)  
class Data:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
# function to interpolate the given data points  
# using Lagrange's formula  
# xi -> corresponds to the new data point  
# whose value is to be obtained  
# n -> represents the number of known data points  
def interpolate(f: list, xi: int, n: int) -> float:  
 # Initialize result  
 result = 0.0  
 for i in range(n):  
 # Compute individual terms of above formula  
 term = f[i].y  
 for j in range(n):  
 if j != i:  
 term = term \* (xi - f[j].x) / (f[i].x - f[j].x)  
 # Add current term to result  
 result += term  
  
 return result  
  
# Driver Code  
if \_\_name\_\_ == "\_\_main\_\_":  
 # creating an array of 4 known data points  
 f = [Data(0, 10), Data(1, 11), Data(2, 22), Data(5, 55)]  
 # Using the interpolate function to obtain a data point  
 # corresponding to x=1.25  
 print("Value of f(1.25) is :", interpolate(f, 1.25, 4))



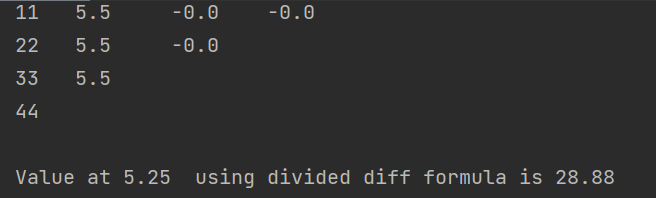
We can apply Lagrange’s interpolation formula to get our solution.   
The Lagrange’s Interpolation formula:   
If, y = f(x) takes the values y0, y1, … , yn corresponding to x = x0, x1 , … , xn then,

  
  
We can use interpolation techniques to find an intermediate data point say at x = 1.25.

# 3. Divide differences

Write a program that executes the divide differences method.

#ex3  
# Python3 program for implementing  
# divided difference formula  
# Function to find the product term  
def proterm(i, value, x):  
 pro = 1;  
 for j in range(i):  
 pro = pro \* (value - x[j]);  
 return pro;  
# Function for calculating  
# divided difference table  
def dividedDiffTable(x, y, n):  
 for i in range(1, n):  
 for j in range(n - i):  
 y[j][i] = ((y[j][i - 1] - y[j + 1][i - 1]) /  
 (x[j] - x[i + j]));  
 return y;  
# Function for applying  
# divided difference formula  
def applyFormula(value, x, y, n):  
 sum = y[0][0];  
 for i in range(1, n):  
 sum = sum + (proterm(i, value, x) \* y[0][i]);  
 return sum;  
# Function for displaying divided  
# difference table  
def printDiffTable(y, n):  
 for i in range(n):  
 for j in range(n - i):  
 print(round(y[i][j], 4), "\t",  
 end=" ");  
 print("");  
# Driver Code  
# number of inputs given  
n = 4;  
y = [[0 for i in range(10)]  
 for j in range(10)];  
x = [2, 4, 6, 8];  
# y[][] is used for divided difference  
# table where y[][0] is used for input  
y[0][0] = 11;  
y[1][0] = 22;  
y[2][0] = 33;  
y[3][0] = 44;  
  
# calculating divided difference table  
y = dividedDiffTable(x, y, n);  
  
# displaying divided difference table  
printDiffTable(y, n);  
  
# value to be interpolated  
value = 5.25;  
  
# printing the value  
print("\nValue at", value, " using divided diff formula is",  
 round(applyFormula(value, x, y, n), 2))



* Through the difference table, we can find out the differences in higher order.

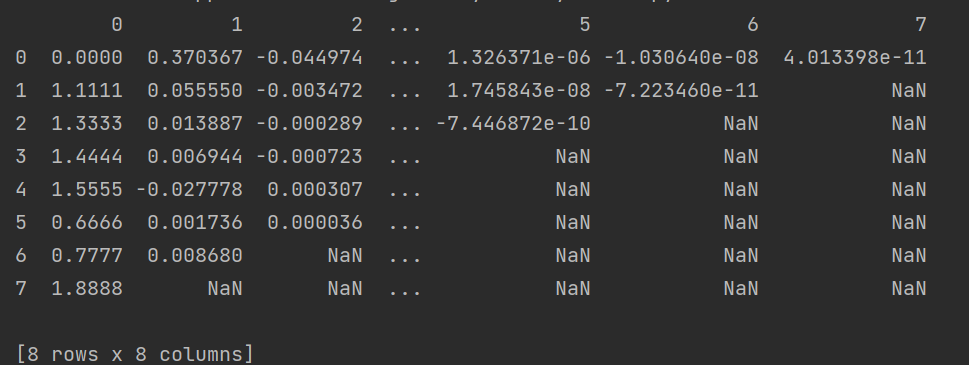
Differences at each stage in each of the columns are easily measured by subtracting the previous value from its immediately succeeding value.

The differences are found successively between the two adjacent values of the y variable till the ultimate difference vanishes or becomes a constant.

# 4. Newton method

Write a program that will compute Newton's form of interpolation.

#ex4  
import pandas as pd  
import matplotlib.pyplot as plt  
import numpy as np  
def newton\_interpolation(x, y, xi):  
 # length/number of datapoints  
 n = len(x)  
 # divided difference initialization  
 fdd = [[None for x in range(n)] for x in range(n)]  
 # f(X) values at different degrees  
 yint = [None for x in range(n)]  
 # error value  
 ea = [None for x in range(n)]  
 # finding divided difference  
 for i in range(n):  
 fdd[i][0] = y[i]  
 for j in range(1, n):  
 for i in range(n - j):  
 fdd[i][j] = (fdd[i + 1][j - 1] - fdd[i][j - 1]) / (x[i + j] - x[i])  
 # just printing dd here  
 fdd\_table = pd.DataFrame(fdd)  
 print(fdd\_table)  
 # interpolating xi  
 xterm = 1  
 yint[0] = fdd[0][0]  
 for order in range(1, n):  
 xterm = xterm \* (xi - x[order - 1])  
 yint2 = yint[order - 1] + fdd[0][order] \* xterm  
 ea[order - 1] = yint2 - yint[order - 1]  
 yint[order] = yint2  
 # returning a map for pandas dataframe  
 return map(lambda yy, ee: [yy, ee], yint, ea)  
x = [1,4,8,16,32,64,128,256]  
y = [0, 1.1111, 1.3333, 1.4444, 1.5555 , 0.6666, 0.7777, 1.8888]  
a = newton\_interpolation(x, y, 2)  
df = pd.DataFrame(a, columns=['f(x)','error'])



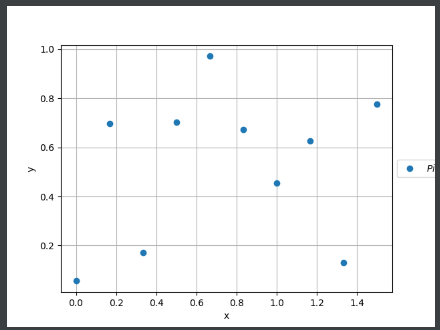
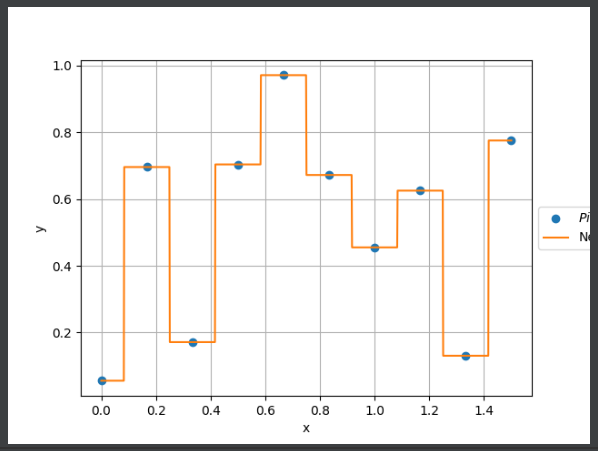
# 5. Piecewise linear interpolation

Write a program that will compute the piecewise linear interpolation

## Nearest (aka. piecewise) interpolation

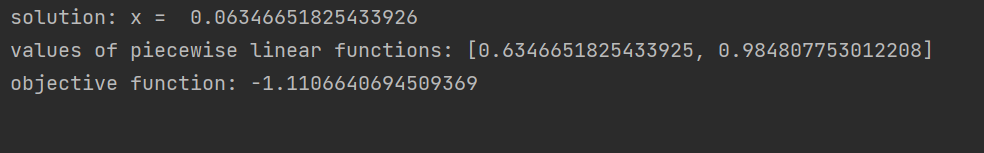
Function y(x)y(x) takes the value yiyi of the nearest point PiPi on the xx direction.

#ex5  
import numpy as np  
from matplotlib import pyplot as plt  
from scipy import interpolate  
N = 10  
xmin, xmax = 0., 1.5  
xi = np.linspace(xmin, xmax, N)  
yi = np.random.rand(N)  
plt.plot(xi,yi, 'o', label = "$Pi$")  
plt.grid()  
plt.xlabel("x")  
plt.ylabel("y")  
plt.legend(loc='center left', bbox\_to\_anchor=(1, 0.5))  
plt.show()  
x = np.linspace(xmin, xmax, 1000)  
interp = interpolate.interp1d(xi, yi, kind = "nearest")  
y\_nearest = interp(x)  
plt.plot(xi,yi, 'o', label = "$Pi$")  
plt.plot(x, y\_nearest, "-", label = "Nearest")  
plt.grid()  
plt.xlabel("x")  
plt.ylabel("y")  
plt.legend(loc='center left', bbox\_to\_anchor=(1, 0.5))  
plt.show()

**** ****

***Method2***

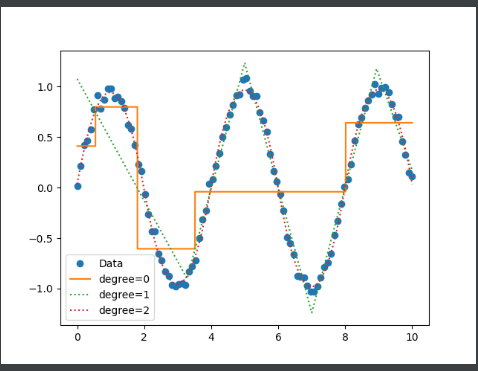
import xpress as xp  
import math  
import numpy as np  
x = xp.var(ub=4)  
# Piecewise linear, continuous concave function  
pw1 = xp.pwl({(0, 1): 10\*x,  
 (1, 2): 10 + 3\*(x-1),  
 (2, 3): 13 + 2\*(x-2),  
 (3, 4): 15 + (x-3)})  
# Approximate sin(freq \* x) for x in [0, 2\*pi]  
N = 100 # Number of points of the approximation  
freq = 27.5 # frequency  
step = 2 \* math.pi / (N - 1) # width of each x segment  
breakpoints = np.array([i \* step for i in range(N)])  
values = np.sin(freq \* breakpoints) # value of the function  
slopes = freq \* np.cos(freq \* breakpoints) # derivative  
# Piecewise linear, discontinuous function over N points: over the  
# i-th interval, the function is equal to v[i] + s[i] \* (y - b[i])  
# where v, s, b are value, slope, and breakpoint.  
pw2 = xp.pwl({(breakpoints[i], breakpoints[i+1]):  
 values[i] + slopes[i] \* (x - breakpoints[i]) for i in range(N - 1)})  
p = xp.problem(x) # create a problem and add variable x  
p.setObjective (pw1 - pw2)  
p.solve()  
print("solution: x = ", p.getSolution(x))  
print("values of piecewise linear functions:", xp.evaluate([pw1, pw2], problem=p))  
print("objective function:", p.getObjVal())

****

# 6. Piecewise quadratic

Write a program that will compute piecewise quadratic interpolation.

import numpy as np  
import pwlf as pwlf  
from matplotlib import pyplot as plt  
x = np.linspace(0, 10, num=100)  
y = np.sin(x \* np.pi / 2)  
# add noise to the data  
y = np.random.normal(0, 0.05, 100) + y  
# initialize piecewise linear fit with our x and y data  
# pwlf lets you fit continuous model for many degree polynomials  
# degree=0 constant # degree=1 linear (default)# degree=2 quadratic  
my\_pwlf\_0 = pwlf.PiecewiseLinFit(x, y, degree=0)  
my\_pwlf\_1 = pwlf.PiecewiseLinFit(x, y, degree=1) # default  
my\_pwlf\_2 = pwlf.PiecewiseLinFit(x, y, degree=2)  
# fit the data for four line segments  
res0 = my\_pwlf\_0.fitfast(5, pop=50)  
res1 = my\_pwlf\_1.fitfast(5, pop=50)  
res2 = my\_pwlf\_2.fitfast(5, pop=50)  
# predict for the determined points  
xHat = np.linspace(min(x), max(x), num=10000)  
yHat0 = my\_pwlf\_0.predict(xHat)  
yHat1 = my\_pwlf\_1.predict(xHat)  
yHat2 = my\_pwlf\_2.predict(xHat)  
# plot the results  
plt.figure()  
plt.plot(x, y, 'o', label='Data')  
plt.plot(xHat, yHat0, '-', label='degree=0')  
plt.plot(xHat, yHat1, ':', label='degree=1')  
plt.plot(xHat, yHat2, ':', label='degree=2')  
plt.legend()  
plt.show()

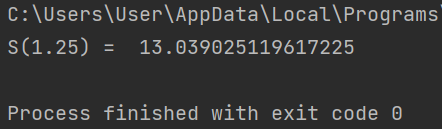
****

**7. Cubic Spline Interpolation**

Write a program that will compute spline interpolation

Method 1

#Spline Interpolation  
#Write a program that will compute spline interpolation  
import numpy as np  
from scipy.interpolate import CubicSpline  
# calculate 5 natural cubic spline polynomials for 6 points  
# (x,y) = (0,10) (1,11) (2,22) (3,33) (4,33) (5,55)  
x = np.array([0, 1, 2, 3, 4, 5])  
y = np.array([10,11,22,33,44,55])  
# calculate natural cubic spline polynomials  
cs = CubicSpline(x,y,bc\_type='natural')  
# show values of interpolation function at x=1.25  
print('S(1.25) = ', cs(1.25))



!!! The SciPy library already has a CubicSpline class !!!

Method 2

import numpy as np  
from math import sqrt  
def cubic\_interp1d(x0, x, y):  
 *"""Interpolate a 1-D function using cubic splines.  
 x0 : a float or an 1d-array  
 x : (N,) array\_like  
 A 1-D array of real/complex values.  
 y : (N,) array\_like  
 A 1-D array of real values. The length of y along the  
 interpolation axis must be equal to the length of x.  
 Implement a trick to generate at the first step the Cholesky matrice of  
 the tridiagonal matrices A (thus L is a bidiagonal matrice that  
 can be solved in two distinct loops). """* x = np.asfarray(x)  
 y = np.asfarray(y)  
 # remove non-finite values  
 # indexes = np.isfinite(x)  
 # x = x[indexes] # y = y[indexes]  
 # check if sorted  
 if np.any(np.diff(x) < 0):  
 indexes = np.argsort(x)  
 x = x[indexes]  
 y = y[indexes]  
 size = len(x)  
 xdiff = np.diff(x)  
 ydiff = np.diff(y)  
 # allocate buffer matrices  
 Li = np.empty(size)  
 Li\_1 = np.empty(size-1)  
 z = np.empty(size)  
 # fill diagonals Li and Li-1 and solve [L][y] = [B]  
 Li[0] = sqrt(2\*xdiff[0])  
 Li\_1[0] = 0.0  
 B0 = 0.0 # natural boundary  
 z[0] = B0 / Li[0]  
 for i in range(1, size-1, 1):  
 Li\_1[i] = xdiff[i-1] / Li[i-1]  
 Li[i] = sqrt(2\*(xdiff[i-1]+xdiff[i]) - Li\_1[i-1] \* Li\_1[i-1])  
 Bi = 6\*(ydiff[i]/xdiff[i] - ydiff[i-1]/xdiff[i-1])  
 z[i] = (Bi - Li\_1[i-1]\*z[i-1])/Li[i]  
 i = size - 1  
 Li\_1[i-1] = xdiff[-1] / Li[i-1]  
 Li[i] = sqrt(2\*xdiff[-1] - Li\_1[i-1] \* Li\_1[i-1])  
 Bi = 0.0 # natural boundary  
 z[i] = (Bi - Li\_1[i-1]\*z[i-1])/Li[i]  
 # solve [L.T][x] = [y]  
 i = size-1  
 z[i] = z[i] / Li[i]  
 for i in range(size-2, -1, -1):  
 z[i] = (z[i] - Li\_1[i-1]\*z[i+1])/Li[i]  
 # find index  
 index = x.searchsorted(x0)  
 np.clip(index, 1, size-1, index)  
 xi1, xi0 = x[index], x[index-1]  
 yi1, yi0 = y[index], y[index-1]  
 zi1, zi0 = z[index], z[index-1]  
 hi1 = xi1 - xi0  
 # calculate cubic  
 f0 = zi0/(6\*hi1)\*(xi1-x0)\*\*3 + \  
 zi1/(6\*hi1)\*(x0-xi0)\*\*3 + \  
 (yi1/hi1 - zi1\*hi1/6)\*(x0-xi0) + \  
 (yi0/hi1 - zi0\*hi1/6)\*(xi1-x0)  
 return f0  
if \_\_name\_\_ == '\_\_main\_\_':  
 import matplotlib.pyplot as plt  
 x = np.linspace(0, 10, 11)  
 y = np.sin(x)  
 plt.scatter(x, y)  
 x\_new = np.linspace(0, 10, 201)

plt.plot(x\_new, cubic\_interp1d(x\_new, x, y))  
 plt.show()

