Report

Homework nr.3

Student: Macrii Danu
Gr.: FAF-222
Teacher: Bostan Viorel

Chișinău, Spring 2023

Problem 1

My code:

```
def lagrange_method(x, y, xp):
    n = 6
    yp = 0
    for i in range(n):
        p = 1
        for j in range(n):
            if i != j:
                p = p * (xp - x[j]) / (x[i] - x[j])

        yp = yp + p * y[i]
    return yp

# l0 = lambda x: ya[0]*(x-xa[1])*(x-xa[2])*(x-xa[3])*(x-xa[4])*(x-xa[5]))/\
#                ((xa[0]-xa[1])*(xa[0]-xa[2])*(xa[0]-xa[3])*(xa[0]-xa[4])*(xa[0]-xa[5]))
#
# l1 = lambda x: ya[1]*((x-xa[0])*(x-xa[2])*(x-xa[3])*(x-xa[4])*(x-xa[5]))/\
#                ((xa[1]-xa[0])*(xa[1]-xa[2])*(xa[1]-xa[3])*(xa[1]-xa[4])*(xa[1]-xa[5]))
#
# l2 = lambda x: ya[2]*((x-xa[0])*(x-xa[1])*(x-xa[3])*(x-xa[4])*(x-xa[5]))/\
#                ((xa[2]-xa[1])*(xa[2]-xa[0])*(xa[2]-xa[3])*(xa[2]-xa[4])*(xa[2]-xa[5]))
#
# l3 = lambda x: ya[3]*((x-xa[0])*(x-xa[1])*(x-xa[2])*(x-xa[4])*(x-xa[5]))/\
#                ((xa[3]-xa[1])*(xa[3]-xa[2])*(xa[3]-xa[0])*(xa[3]-xa[4])*(xa[3]-xa[5]))
#
# l4 = lambda x: ya[4]*((x-xa[0])*(x-xa[1])*(x-xa[2])*(x-xa[3])*(x-xa[5]))/\
#                ((xa[4]-xa[1])*(xa[4]-xa[2])*(xa[4]-xa[3])*(xa[4]-xa[0])*(xa[4]-xa[5]))
#
# l5 = lambda x: ya[5]*((x-xa[0])*(x-xa[1])*(x-xa[2])*(x-xa[3])*(x-xa[4]))/\
#                ((xa[5]-xa[1])*(xa[5]-xa[2])*(xa[5]-xa[3])*(xa[5]-xa[0])*(xa[5]-xa[4]))
```

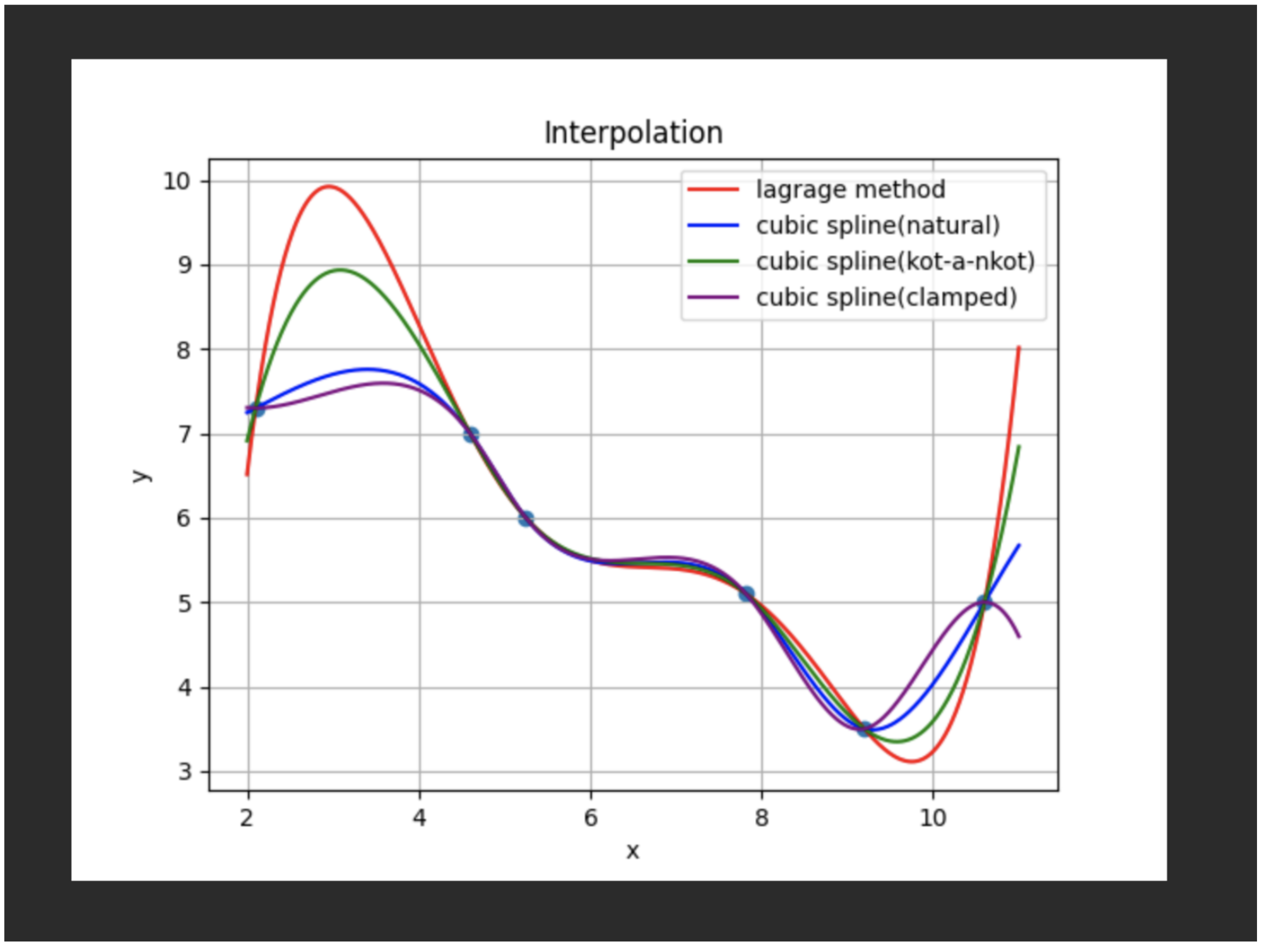
Explaining:

In this problem I exercise Lagrange method , did it manually and then created an algorithm for automatical compilation . After that used several cubic splines:

- natural $\implies$ second derivatives at end points is zero
- not-a-knot $\implies$ the first and second segment at a curve end are the same polynomial
- periodic $\implies$ points are repeating
- clamped $\implies$ the first derivative at end points is zero

Obviously we see that the best method for the shortest path is natural cubic spline.

Results



Problem 2

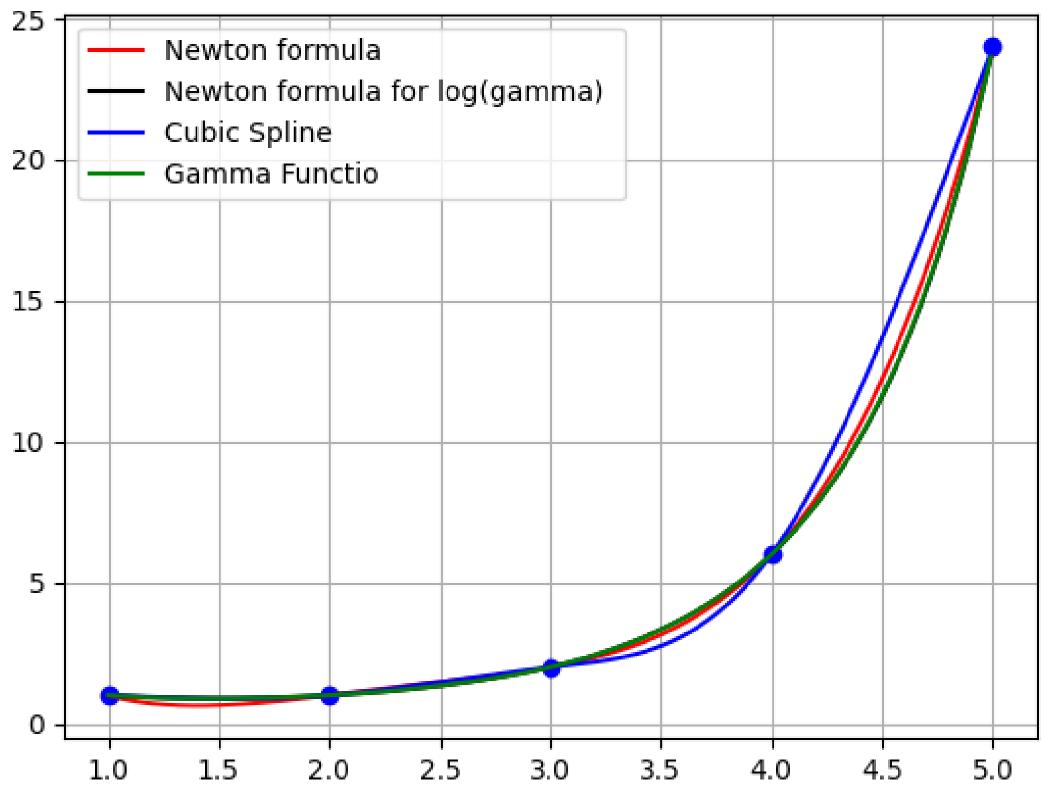
My code:
```
x = np.array([1, 2, 3, 4, 5])
y = np.array([1, 1, 2, 6, 24])
y1 = np.array([0, 0, log(2), log(6), log(24)])

def divided_diff(x, y):
    n = len(y)
    coef = np.zeros([n, n])
    coef[:, 0] = y

    for j in range(1, n):
        for i in range(n - j):
            coef[i][j] = (coef[i + 1][j - 1] - coef[i][j - 1]) / (x[i + j] - x[i])

    return coef

def newton_poly(coef, x_data, x):
    n = len(x_data) - 1
    p = coef[n]
    for k in range(1, n + 1):
        p = coef[n - k] + (x - x_data[n - k]) * p
    return p

x_new = np.arange(1, 5, 0.01)
a_s = divided_diff(x, y)[0, :]
y_new = newton_poly(a_s, x, x_new)
```

Explaining:

To create table bellow I used the function divideddiff , it creates a matrix and where the first column is y and next ones are the result.To plot the equation I used the function newtonpoly , it takes the matrix and combines it to create a function.I did newton method for both gamma function x, y and x, log(y). After computation found that , surprisingly log(y) gives the smallest error , maybe because it is similary to the e^x or maybe because gamma function uses e^{-t}.

Result

```
+------+-------+-------+--------------------+-------+
| f(x) | D1(x) | D2(x) |        D3(x)       | D4(x) |
+------+-------+-------+--------------------+-------+
| 1.0  |  0.0  |  0.5  | 0.3333333333333333 | 0.375 |
| 1.0  |  1.0  |  1.5  | 1.8333333333333333 |   -   |
| 2.0  |  4.0  |  7.0  |         -          |   -   |
| 6.0  |  18.0 |   -   |         -          |   -   |
| 24.0 |   -   |   -   |         -          |   -   |
+------+-------+-------+--------------------+-------+
Newton formula max error:  0.6843861302529195
Newton formula for log(gamma) max error:  0.066409634177389888
Cubic Spline max error:  2.244141371222291
```



Problem 3

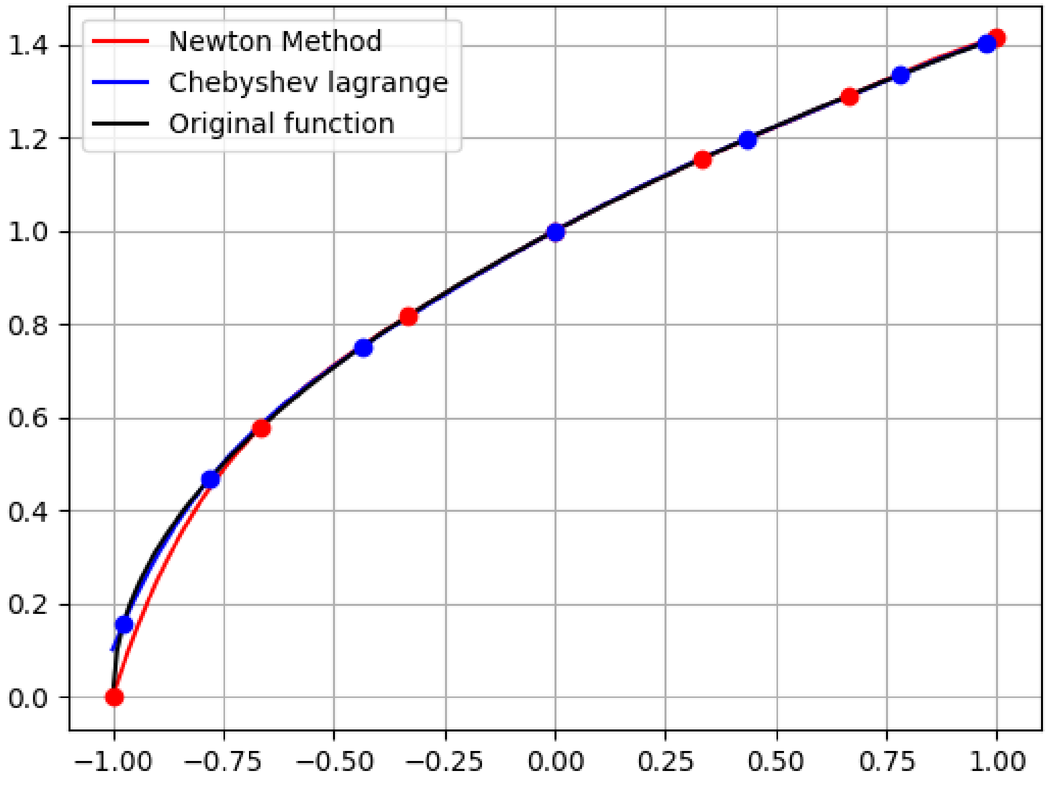
My code:

```
f = lambda x: sqrt(x+1)

i = array(range(7))
chebyshevroots = cos((2*i+1)*pi/(14))

x = linspace(-1, 1, 7)
y = [f(i) for i in x]

y_cheb = [f(i) for i in chebyshevroots]
x_new = arange(-1, 1, 0.01)

a_s = divided_diff(x, y)[0, :]
a_scheb = divided_diff(chebyshevroots, y_cheb)[0, :]

p7 = newton_poly(a_s, x, x_new)
m7 = newton_poly(a_scheb, chebyshevroots, x_new)
```

Explaining:

To find the near minimax polynomial approximation first we need to find the roots of Chebyshev Polynomial of degree 7. Find the f(cheb roots) to find the y and do computational using newton method. From the graph below we can see that choosing points to interpolate the function is also important. Ploting using uniformly distributed points gives us error near the bounds of the interval , but minimax approximation gives us much smaller maximal error , solving Runges Problem.

Result



Newton Method
Chebyshev lagrange
Original function
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

Appendix

1

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.interpolate import CubicSpline
def lagrange_method(x, y, xp): n=6
   yp = 0
   for i in range(n):
p=1
for j in range(n):
           if i != j:
               p = p * (xp - x[j]) / (x[i] - x[j])
       yp = yp + p * y[i]
   return yp
xa = [2.1, 4.6, 5.25, 7.82, 9.2, 10.6]
ya = [7.3, 7.0, 6.0, 5.1, 3.5, 5.0]
x_ew = np.linspace(2, 11, 1000)
# l0 = lambda x: ya[0]*(x-xa[1])*(x-xa[2])*(x-xa[3])*(x-xa[4])*(x-xa[5]))/\
#
((xa[0]-xa[1])*(xa[0]-xa[2])*(xa[0]-xa[3])*(xa[0]-xa[4])*(xa[0]-xa[5]))
#
# l1 = lambda x:
ya[1]*((x-xa[0])*(x-xa[2])*(x-xa[3])*(x-xa[4])*(x-xa[5]))/\
#
((xa[1]-xa[0])*(xa[1]-xa[2])*(xa[1]-xa[3])*(xa[1]-xa[4])*(xa[1]-xa[5]))
#
# l2 = lambda x:
ya[2]*((x-xa[0])*(x-xa[1])*(x-xa[3])*(x-xa[4])*(x-xa[5]))/\
#
((xa[2]-xa[1])*(xa[2]-xa[0])*(xa[2]-xa[3])*(xa[2]-xa[4])*(xa[2]-xa[5]))
#
# l3 = lambda x:
ya[3]*((x-xa[0])*(x-xa[1])*(x-xa[2])*(x-xa[4])*(x-xa[5]))/\
#
((xa[3]-xa[1])*(xa[3]-xa[2])*(xa[3]-xa[0])*(xa[3]-xa[4])*(xa[3]-xa[5]))
#
# l4 = lambda x:
ya[4]*((x-xa[0])*(x-xa[1])*(x-xa[2])*(x-xa[3])*(x-xa[5]))/\
#
((xa[4]-xa[1])*(xa[4]-xa[2])*(xa[4]-xa[3])*(xa[4]-xa[0])*(xa[4]-xa[5]))
#
# l5 = lambda x:
ya[5]*((x-xa[0])*(x-xa[1])*(x-xa[2])*(x-xa[3])*(x-xa[4]))/\
#
((xa[5]-xa[1])*(xa[5]-xa[2])*(xa[5]-xa[3])*(xa[5]-xa[0])*(xa[5]-xa[4]))
f1 = CubicSpline(xa, ya, bc_type='natural')
```

```python
f2 = CubicSpline(xa, ya)
f3 = CubicSpline(xa, ya, bc_type='clamped')
plt.scatter(xa, ya)
# plot the function
plt.plot(x_ew, lagrange_method(xa, ya, x_ew), 'r', label='lagrage method')
plt.plot(x_ew, f1(x_ew), 'b', label='cubic spline(natural)')
plt.plot(x_ew, f2(x_ew), 'g', label='cubic spline(kot-a-nkot)')
plt.plot(x_ew, f3(x_ew), 'purple', label='cubic spline(clamped)')
plt.title('Interpolation')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
plt.show()
```

2

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.interpolate import CubicSpline
from scipy.special import gamma
from numpy import log, exp
from prettytable import PrettyTable

table = PrettyTable()



x = np.array([1, 2, 3, 4, 5])
y = np.array([1, 1, 2, 6, 24])
y1 = np.array([0, 0, log(2), log(6), log(24)])


def divided_diff(x, y):
    n = len(y)
    coef = np.zeros([n, n])
    coef[:, 0] = y

    for j in range(1, n):
        for i in range(n - j):
            coef[i][j] = (coef[i + 1][j - 1] - coef[i][j - 1]) / (x[i + j] - x[i])

    return coef


def newton_poly(coef, x_data, x):
    n = len(x_data) - 1
    p = coef[n]
    for k in range(1, n + 1):
        p = coef[n - k] + (x - x_data[n - k]) * p
    return p


x_new = np.arange(1, 5, 0.01)
a_s = divided_diff(x, y)[0, :]
a1_s = divided_diff(x, y1)[0, :]
y_new = newton_poly(a_s, x, x_new)
y1_new = newton_poly(a1_s, x, x_new)
table.field_names=['f(x)', 'D1(x)', 'D2(x)', 'D3(x)', 'D4(x)']
for row in divided_diff(x, y):
    table.add_row(row)
print(table)
f = CubicSpline(x, y, bc_type='natural')
```

```
maxnf = 0
maxnlf = 0
maxcs = 0
xxx = [x/100. for x in range(100, 500)]

for i in x_new:
    maxnf = max(maxnf, abs(gamma(i)-newton_poly(a_s, x, i)))
    maxnlf = max(maxnlf, abs(gamma(i)-exp(newton_poly(a1_s, x, i))))
    maxcs = max(maxcs, abs(gamma(i)-f(i)))

print('Newton formula max error: ',maxnf)
print('Newton formula for log(gamma) max error: ',maxnlf)
print('Cubic Spline max error: ',maxcs)

plt.plot(x, y, 'bo')
plt.plot(x_new, y_new, 'r', label='Newton formula')
plt.plot(x_new, exp(y1_new), 'black', label='Newton formula for log(gamma) ')
plt.plot(x_new, f(x_new), 'b', label='Cubic Spline')
plt.plot(x_new, gamma(x_new), 'g', label='Gamma Functio')
plt.legend()
plt.grid()
plt.show()
```

3

```
from numpy import cos, pi, sqrt, linspace, array, arange, zeros
import matplotlib.pyplot as plt


def divided_diff(x, y):
    n = len(y)
    coef = zeros([n, n])
    coef[:, 0] = y

    for j in range(1, n):
        for i in range(n - j):
            coef[i][j] = (coef[i + 1][j - 1] - coef[i][j - 1]) / (x[i + j] - x[i])
    return coef


def newton_poly(coef, x_data, x):
    n = len(x_data) - 1
    p = coef[n]
    for k in range(1, n + 1):
        p = coef[n - k] + (x - x_data[n - k]) * p
    return p

f = lambda x: sqrt(x+1)
i = array(range(7))
chebyshevroots = cos((2*i+1)*pi/(14))

x = linspace(-1, 1, 7)
y = [f(i) for i in x]

y_cheb = [f(i) for i in chebyshevroots]

x_new = arange(-1, 1, 0.01)
a_s = divided_diff(x, y)[0, :]
a_scheb = divided_diff(chebyshevroots, y_cheb)[0, :]
p7 = newton_poly(a_s, x, x_new)
m7 = newton_poly(a_scheb, chebyshevroots, x_new)

plt.plot(x_new, p7, 'r', label='Newton Method')
plt.plot(x_new, m7, 'b', label='Chebyshev lagrange')
plt.plot(x_new, f(x_new), 'black', label='Original function')
plt.plot(x,y, 'ro')
plt.plot(chebyshevroots, y_cheb, 'bo')
plt.legend()
plt.grid()
plt.show()
```