FA2_INTERPOLATION_KHAFAJI

March 3, 2025

[1]:
```python
import numpy as np
import polars as pl
from sympy import *
import scipy.interpolate as spi
```

0.1 1.

Use Neville's Method algorithm to generate the table of approximations for Lagrange interpolating polynomials of degree one, two, and three to approximate $f(0.43)$

if $f(0) = 1, f(0.25) = 1.64872, f(0.5) = 2.71828,$ and $f(0.75) = 4.48169$

[2]:
```python
def nevilles(x, values):

    n = len(values[0])

    for col in range(1, n): # iteration for creation of columns

        temp_list = []
        # temporary list

        for j in range(col, n): # iteration for getting j (larger n)
            # j-col is the distance (for Q2, i is 1 below j, in Q3, i is 2␣
↪below j).
            # j and i refers to x values.
            i = j-col

            # j  and j-1 functions as indexes to the P(x) values.
            # use the Neville's method formula
            numerator = ((x-values[0][i])*values[col][j]) -␣
↪((x-values[0][j])*values[col][j-1])
            p = numerator/(values[0][j] - values[0][i])

            # append result to temp list
            temp_list.append(p)

        # append np.nan multiple times to the left in temporary list
        temp_list = [np.nan]*col + temp_list
```

```python
        # append temp list to values
        values = np.append(values, [temp_list], axis=0)

    # create list of column names
    column_names = ["x"] + [f"Q{n}" for n in range(values.shape[0]-1)]

    # turn rows into columns
    values_T = np.transpose(values)

    return  pl.DataFrame(values_T, schema=column_names)
```

[3]:
```python
given_values = np.array([
    [0, 0.25, 0.5, 0.75],
    [1, 1.64872, 2.71828, 4.48169]
])

x = 0.43
```

[4]:
```python
nevilles(x, given_values)
```

[4]: shape: (4, 5)

x	Q0	Q1	Q2	Q3
---	---	---	---	---
f64	f64	f64	f64	f64
0.0	1.0	NaN	NaN	NaN
0.25	1.64872	2.1157984	NaN	NaN
0.5	2.71828	2.4188032	2.376383	NaN
0.75	4.48169	2.224525	2.348863	2.360605

0.1.1 Answer for 1

$f(0.43) \approx 2.360605$

0.2 2.

Use the Newton Divided Differences Algorithm to construct the interpolating polynomials of degree three and approximate $f(8.4)$

given $f(8.1) = 16.94410, f(8.3) = 17.56492, f(8.6) = 18.50515, and f(8.7) = 18.82091 $

[5]:
```python
given_values_divdiff = np.array([
    [8.1, 8.3, 8.6, 8.7],
    [16.94410, 17.56492, 18.50515, 18.82091]
])
```

```
x_divdiff = 8.4
```

[6]:
```
def newtonDivDiff(x, values):

    n = len(values[0])

    b_vals = [values[1][0]]
    for col in range(1, n): # iteration for creation of columns

        temp_list = []
        # temporary list

        for j in range(col, n): # iteration for getting j (larger n)
            # j-col is the distance (for Q2, i is 1 below j, in Q3, i is 2␣
↪below j).
            # j and i refers to x values.
            i = j-col

            # j  and j-1 functions as indexes to the P(x) values.
            # use the Newton's Divided Difference Method
            numerator = values[col][j] - values[col][j-1]
            p = numerator/(values[0][j] - values[0][i])

            # append result to temp list
            temp_list.append(p)

        # append first value on retrieved values to our list of b values
        # before we insert np.nan
        b_vals.append(temp_list[0])

        # append np.nan multiple times to the left in temporary list
        temp_list = [np.nan]*col + temp_list

        # append temp list to values
        values = np.append(values, [temp_list], axis=0)

    # creating dataframe
    # create list of column names
    column_names = ["x"] + ["f(x)"] + [f"{n}th division" for n in␣
↪range(1,values.shape[0]-1)]

    # turn rows into columns
    values_T = np.transpose(values)

    df = pl.DataFrame(values_T, schema=column_names)

    # create polynomial for approximation
```

```python
    polynomial_sum = 0
    for idx, b_value in enumerate(b_vals): # iterating through each b values␣
↪retrieved
        if idx == 0:
            polynomial_sum = polynomial_sum + b_value
        else:
            temp_prod = b_value
            for p in range(idx): # creating iterator through current index to␣
↪iterate
                temp_prod = temp_prod*(x-values[0][p])
            polynomial_sum = polynomial_sum + temp_prod

    return df, polynomial_sum
```

[7]:
```python
divdiff_df, divDiff_approx = newtonDivDiff(x_divdiff, given_values_divdiff)
divdiff_df
```

[7]: shape: (4, 5)

x	f(x)	1th division	2th division	3th division
---	---	---	---	---
f64	f64	f64	f64	f64
8.1	16.9441	NaN	NaN	NaN
8.3	17.56492	3.1041	NaN	NaN
8.6	18.50515	3.1341	0.06	NaN
8.7	18.82091	3.1576	0.05875	-0.002083

[8]:
```python
print(f"The approximation yielded by the formula for f({x_divdiff}) is:␣
↪{divDiff_approx}")
```

```
The approximation yielded by the formula for f(8.4) is: 17.877142499999998
```

1 Machine Exercise

given $f(x) = xcos(x) - 2x^2 + 3x - 1$

and the data:

[9]:
```python
data_example_3 = {
    "x" : [0.1, 0.2, 0.3, 0.4],
    "fx" : [-0.62049958, -0.28398668, 0.00660095, 0.24842440],
    "f'x" : [3.58502082, 3.14033271, 2.66668043, 2.16529366]
}

df_machine_excercise = pl.DataFrame(data_example_3)
```

```
df_machine_excercise
```

[9]:
```
shape: (4, 3)

x      fx          f'x
---    ---         ---
f64    f64         f64

0.1    -0.6205     3.585021
0.2    -0.283987   3.140333
0.3    0.006601    2.66668
0.4    0.2484244   2.165294
```

1.1 3.

Use Hermite Interpolation to construct an approximating polynomial to approximate $f(0.25)$ and find the absolute error.

[10]:
```
x = symbols('x')
func_3 = x*cos(x) -2*(x**2) +3*x -1
```

[11]:
```
def lagrange_range(x_s):

    x=symbols('x')
    lagrange_list = []
    lagrange_derivative_list = []

    for x_val in x_s:
        usable_x = [xnot for xnot in x_s if xnot != x_val]
        lag_expr = 1
        for p in usable_x:
            lag_expr = lag_expr * (x-p)/(x_val - p)
        lagrange_list.append(lag_expr)
        lagrange_derivative_list.append(diff(lag_expr, x))

    return lagrange_list, lagrange_derivative_list
```

[12]:
```
def hermite_approx(x_approx, x_s, fx_s, fpx_s):

    x = symbols('x')
    lagrange_list, lagrange_derivative_list = lagrange_range(x_s)

    H_s = []
    h_hat_s = []
```

```
    for x_val, lagra, diff_lagra in zip(x_s, lagrange_list,␣
↪lagrange_derivative_list):

        # solve for hn
        h = (1-2*(x-x_val)*diff_lagra.subs(x, x_val))*(lagra**2)
        H_s.append(h)

        # solve for h hat n
        h_hat = (x-x_val)*(lagra**2)
        h_hat_s.append(h_hat)

    hermite_polynomial = 0
    for fx, fpx, h, h_hat in zip(fx_s, fpx_s, H_s, h_hat_s):
        hermite_polynomial = hermite_polynomial + fx*h + fpx*h_hat

    hermite_polynomial = simplify(hermite_polynomial)
    print("The hermite polynomial is:", hermite_polynomial, end="\n\n")

    print("The approximation is:", hermite_polynomial.subs(x, x_approx))

    return hermite_polynomial, hermite_polynomial.subs(x, x_approx)
```

[13]:
```
polynomial, approximation = hermite_approx(
    0.25,
    data_example_3["x"],
    data_example_3["fx"],
    data_example_3["f'x"]
)
```

```
The hermite polynomial is: 0.00296296301530674*x**7 - 0.00726851855870336*x**6 +
0.0466490743565373*x**5 - 0.00180268517578952*x**4 - 0.499630898146279*x**3 -
2.00004254629857*x**2 + 4.00000255777809*x - 1.00000005866667

The approximation is: -0.132771890847391
```

[14]:
```
print("The absolute error is:", abs(func_3.subs(x,0.25)-approximation))
```

```
The absolute error is: 3.72494743383633e-9
```

1.2 4.

Construct the Natural cubic spline and approximate $f(0.25)$ and $f'(0.25)$ and find the absolute error.

[15]:
```
x_vals = np.array(data_example_3["x"])
y_vals = np.array(data_example_3["fx"])

spline = spi.CubicSpline(x_vals, y_vals, bc_type='natural')
```

```
x_target = 0.25
f_approx = spline(x_target)
f_prime_approx = spline.derivative()(x_target)

print(f"The approximation of f({x_target}) is {f_approx}, and it's first␣
 ↪derivative approximation is {f_prime_approx}")
```

```
The approximation of f(0.25) is -0.13159115625, and it's first derivative
approximation is 2.908242058333334
```

[16]:
```
print(f"The absolute error for the function approximation is {abs(func_3.
 ↪subs(x, x_target)-f_approx)}")

func_3_diff = diff(func_3)
print(f"The absolute error for the derivative approximation is {abs(func_3_diff.
 ↪subs(x, x_target)-f_approx)}")
```

```
The absolute error for the function approximation is 0.00118073832233881
The absolute error for the derivative approximation is 3.03865258814701
```

1.3 5.

Construct the Clamped cubic spline and approximate $f(0.25)$ and $f'(0.25)$ and find the absolute error.

[17]:
```
y_prime_vals = np.array(data_example_3["f'x"])

# Construct Clamped Cubic Spline
spline_clamped = spi.CubicSpline(
    x_vals, y_vals,
    bc_type=((1, y_prime_vals[0]), (1, y_prime_vals[-1]))
)

# Approximate f(0.25) and f'(0.25)
x_target = 0.25
f_approx = spline_clamped(x_target)
f_prime_approx = spline_clamped.derivative()(x_target)
```

[18]:
```
print(f"The approximation of f({x_target}) is {f_approx}, and it's first␣
 ↪derivative approximation is {f_prime_approx}")
```

```
The approximation of f(0.25) is -0.1327722135833333, and it's first derivative
approximation is 2.9070627590000004
```

[19]:
```
print(f"The absolute error for the function approximation is {abs(func_3.
 ↪subs(x, x_target)-f_approx)}")
```

```python
func_3_diff = diff(func_3)
print(f"The absolute error for the derivative approximation is {abs(func_3_diff.
    subs(x, x_target)-f_approx)}")
```

```
The absolute error for the function approximation is 3.19010994481728E-7
The absolute error for the derivative approximation is 3.03983364548035
```