Numerical Analysis

Formative Assessment 2

By Romand Lansangan

Machine Exercises

Write your code, table of values, and final answer. (5 points each)

---

1. Neville's Method

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:

- $f(0) = 1$
- $f(0.25) = 1.64872$
- $f(0.5) = 2.71828$
- $f(0.75) = 4.48169$

Code Snippet

In [272…

```
import numpy as np
import pandas as pd

def neville(given, x):
    n=len(given)
    matrix = np.zeros((n, n), dtype=float)

    # add initial f(x)
    for i in range(n):
        matrix[i][0] = given[i][1]

    # nevilles method
    for r in range(1,n):
        for c in range(1,r+1):
            approx = (x-given[r-c][0])*matrix[r][c-1] - (x-given[r][0])*matrix[r-1][c-1]
            mult = 1/(given[r][0] - given[r-c][0])
            matrix[r][c] = approx*mult

    # turn to df
    x_s = [x for x, y in given]
    result = pd.DataFrame(matrix, index=x_s)
    result = result.replace(0, "")
    result.reset_index(names="x_n", inplace=True)
    return result
```

In [55]:

```
given = [(0, 1), (0.25,1.64872), (0.5, 2.71828), (0.75,4.48169)]
```

```
x=0.43
```

In [57]:
```
neville(given, x)
```

Out[57]:

	x_n	0	1	2	3
0	0.00	1.00000			
1	0.25	1.64872	2.115798		
2	0.50	2.71828	2.418803	2.376383	
3	0.75	4.48169	2.224525	2.348863	2.360605

---

2. Newton’s Divided Differences

Use the Newton’s Divided Differences algorithm to construct the interpolating polynomials of degree three and approximate $f(8.4)$ if:

- $f(8.1) = 16.94410$
- $f(8.3) = 17.56492$
- $f(8.6) = 18.50515$
- $f(8.7) = 18.82091$

In [286…
```
def newton_divided(given, x):
    n=len(given)
    matrix = np.zeros((n, n), dtype=float)

    # add initial f(x)
    for i in range(n):
        matrix[i][0] = given[i][1]

    # calculate for coeficients [F_{0,0}, F_{1,1}...]
    for r in range(1,n):
        for c in range(1,r+1):
            num = matrix[r][c-1] - matrix[r-1][c-1]
            denum = given[r][0] - given[r-c][0]
            matrix[r][c] = num/denum

    # get diagonal of the coeficients values
    diagonal = np.array([matrix[i][i] for i in range(n)])


    x_min_x_n = []
    for r in range(1,n+1):
        res = 1
        for c in range(1,r):
            res *= (x-given[c-1][0])
        x_min_x_n.append(res)

    x_min_x_n = np.array(x_min_x_n) # should be [1,(x-x_0), (x-x_0)(x_x_1),...]
```

```python
    #multiply matrix for approximation
    return (diagonal @ x_min_x_n.T)
```

In [288…

```python
given = [(8.1,16.94410), (8.3,17.56492), (8.6,18.50515), (8.7,18.82091)]
x=8.4
```

In [290…

```python
result = newton_divided(given, x)
result
```

Out[290…

```
17.877142499999998
```

Given the function $f(x) = x\cos x - 2x^2 + 3x - 1$ and the following data:

x	$f(x)$	$f'(x)$
0.1	(-0.62049958)	3.58502082
0.2	(-0.28398668)	3.14033271
0.3	0.00660095	2.66668043
0.4	0.24842440	2.16529366

---

3. Hermite Interpolation

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

In [308…

```python
def hermite_coef(given):
    n = len(given)
    matrix_q = np.zeros((2*n+1, 2*n+1), dtype=float)
    vector_z = np.zeros((2*n+1, 1), dtype=float)

    #step 1-3
    for i in range(n):
        vector_z[2*i][0] = given[i][0]
        vector_z[2*i+1][0] = given[i][0]
        matrix_q[2*i][0] = given[i][1]
        matrix_q[2*i+1][0] = given[i][1]
        matrix_q[2*i+1][1] = given[i][2]

        if i != 0:
            num = matrix_q[2*i][0] - matrix_q[2*i-1][0]
            denum = vector_z[2*i][0] - vector_z[2*i-1][0]
            matrix_q[2*i][1] = num / denum

    #step 4
    for i in range(2, 2*n+1):
        for j in range(2, i+1):
            num = matrix_q[i][j-1] - matrix_q[i-1][j-1]
            denum = vector_z[i][0] - vector_z[i-j][0]
            matrix_q[i][j] = num / denum
```

```
    # extract diagonal values for coefs
    diagonal = np.array([matrix_q[i][i] for i in range(2*n+1)])

    return diagonal
```

In [310…

```
# evaluate hermite
def hermite_approx(coefs, x, x_vals : list):
    result = coefs[0]
    product = 1.0
    x_vals = [x for x,y,z in given]
    for i in range(1, 2*n+1):
        product *= (x - x_vals[(i-1)//2])
        result += coefs[i] * product
    return result
```

In [364…

```
given = [(0.1,-0.62049958,3.58502082),(0.2,-0.28398668,3.14033271),(0.3,0.00660095,2.666
x=0.25
coefs = hermite_coef(given)
approximation = hermite_approx(coefs=coefs, x=x, x_vals=[x for x, y, z in given])
approximation
```

Out[364…

```
-0.13277189859765623
```

Checking

$f(x) = x\cos x - 2x^2 + 3x - 1$

In [366…

```
eq = "x*cos(x)-2*x**2 + 3*x -1"
p_0 = 0.25
true_val = sip.eq_solver(eq, p_0)
true_val
```

Out[366…

```
-0.13277189457233884
```

In [368…

```
abs_er = abs(true_val - approximation)
print("Absolute error: ", abs_er)
```

```
Absolute error:  4.025317384970251e-09
```

---

4. Natural Cubic Spline

Construct the natural cubic spline and approximate $f(0.25)$ and $f'(0.25)$. Find the absolute error.

In [318…

```
def cubic_spline_coef(given):
    x = [x for x,y in given]
    a = [y for x,y in given]

    n = len(x) - 1
    alpha = np.zeros(n)
    l = np.zeros(n+1)
    mu = np.zeros(n+1)
    z = np.zeros(n+1)
    c = np.zeros(n+1)
```

```python
    b = np.zeros(n)
    d = np.zeros(n)

    #step 1
    h = np.diff(x)

    #step 2
    for i in range(1, n):
        alpha[i] = (3/h[i] * (a[i+1] - a[i])) - (3/h[i-1] * (a[i] - a[i-1]))

    #step 3
    l[0] = 1
    mu[0] = 0
    z[0] = 0

    #step 4
    for i in range(1, n):
        l[i] = 2 * (x[i+1] - x[i-1]) - h[i-1] * mu[i-1]
        mu[i] = h[i] / l[i]
        z[i] = (alpha[i] - h[i-1] * z[i-1]) / l[i]

    #step 5
    l[n] = 1
    z[n] = 0
    c[n] = 0

    #step 6
    for j in range(n-1, -1, -1):
        c[j] = z[j] - mu[j] * c[j+1]
        b[j] = (a[j+1] - a[j]) / h[j] - h[j] * (c[j+1] + 2 * c[j]) / 3
        d[j] = (c[j+1] - c[j]) / (3 * h[j])

    return a[:-1], b, c[:-1], d


given = [(0.1,-0.62049958,3.58502082),(0.2,-0.28398668,3.14033271),(0.3,0.00660095,2.666
a_j, b_j, c_j, d_j = cubic_spline_coef([(x, y) for x,y,z in given])
print("a_j:", a_j)
print("b_j:", b_j)
print("c_j:", c_j)
print("d_j:", d_j)
```

```
a_j: [-0.62049958, -0.28398668, 0.00660095]
b_j: [3.4550856 3.1852158 2.6170671]
c_j: [ 0.        -2.698698 -2.982789]
d_j: [-8.99566 -0.94697  9.94263]
```

In [262…

```python
def cubic_spline_approx(x, x_values, a_j, b_j, c_j, d_j):
    x_values = np.sort(x_values)
    j = np.searchsorted(x_values, x) - 1
    if j < 0:
        j = 0
    elif j >= len(x_values) - 1:
        j = len(x_values) - 2

    dx = x - x_values[j]
    S_x = a_j[j] + b_j[j] * dx + c_j[j] * dx**2 + d_j[j] * dx**3

    S_prime_x = b_j[j] + 2 * c_j[j] * dx + 3 * d_j[j] * dx**2
```

```
    return S_x, S_prime_x
```

In [320…

```
x_val = np.sort([x for x,y,z in given])
x=0.25
S_x, S_prime_x = cubic_spline_approx(x=x, x_values=x_val, a_j=a_j, b_j=b_j, c_j=c_j, d_j
print(f"Approximated f({x}) = {S_x}")
print(f"Approximated f'({x}) = {S_prime_x}")
```

```
Approximated f(0.25) = -0.13159100624999998
Approximated f'(0.25) = 2.9082437250000006
```

Checking

$$f(x) = x\cos x - 2x^2 + 3x - 1$$

In [376…

```
eq = "x*cos(x)-2*x**2 + 3*x -1"
p_0 = 0.25
true_val = sip.eq_solver(eq, p_0)
true_val
```

Out[376…

```
-0.13277189457233884
```

In [378…

```
abs_er = abs(true_val - S_x)
print("Absolute error: ", abs_er)
```

```
Absolute error:  8.322259323384484e-05
```

$$f'(x) = -x\sin x + cosx - 4x + 3$$

In [382…

```
eq = "-x*sin(x) + cos(x) - 4*x + 3"
p_0 = 0.25
true_val = sip.eq_solver(eq, p_0)
true_val
```

Out[382…

```
2.907061431897014
```

In [384…

```
abs_er = abs(true_val - S_prime_x)
print("Absolute error: ", abs_er)
```

```
Absolute error:  0.0010038263537266445
```

---

5. Clamped Cubic Spline

Construct the clamped cubic spline and approximate $f(0.25)$ and $f'(0.25)$. Find the absolute error.

In [326…

```
def clamped_cubic_spline_coef(given):
    x = [x for x, y, z in given]
    a = [y for x, y, z in given]
    fp0 = given[0][2]
    fpn = given[-1][2]
```

```
    n = len(x) - 1
    l = np.zeros(n + 1)
    mu = np.zeros(n + 1)
    z = np.zeros(n + 1)
    c = np.zeros(n + 1)
    b = np.zeros(n)
    d = np.zeros(n)

    #step 1
    h = np.diff(x)

    #step 2
    alpha = np.zeros(n + 1)
    alpha[0] = (3 / h[0]) * (a[1] - a[0]) - 3 * fp0
    alpha[n] = 3 * fpn - (3 / h[-1]) * (a[n] - a[n - 1])

    #step 3
    for i in range(1, n):
        alpha[i] = (3 / h[i]) * (a[i + 1] - a[i]) - (3 / h[i - 1]) * (a[i] - a[i - 1])

    #step 4
    l[0] = 2 * h[0]
    mu[0] = 0.5
    z[0] = alpha[0] / l[0]

    #step 5
    for i in range(1, n):
        l[i] = 2 * (x[i + 1] - x[i - 1]) - h[i - 1] * mu[i - 1]
        mu[i] = h[i] / l[i]
        z[i] = (alpha[i] - h[i - 1] * z[i - 1]) / l[i]

    #step 6
    l[n] = h[-1] * (2 - mu[-1])
    z[n] = (alpha[n] - h[-1] * z[n - 1]) / l[n]
    c[n] = z[n]

    #step 7
    for j in range(n - 1, -1, -1):
        c[j] = z[j] - mu[j] * c[j + 1]
        b[j] = (a[j + 1] - a[j]) / h[j] - h[j] * (c[j + 1] + 2 * c[j]) / 3
        d[j] = (c[j + 1] - c[j]) / (3 * h[j])

    return a[:-1], b, c[:-1], d


given = [(0.1, -0.62049958, 3.58502082), (0.2, -0.28398668, 3.14033271), (0.3, 0.0066009
x_val = np.sort([x for x, y, z in given])
x = 0.25


a_j, b_j, c_j, d_j = clamped_cubic_spline_coef(given)
print("a_j:", a_j)
print("b_j:", b_j)
print("c_j:", c_j)
print("d_j:", d_j)
```

```
a_j: [-0.62049958, -0.28398668, 0.00660095]
b_j: [3.58502082 3.1416661  2.66133066]
c_j: [-2.16320744 -2.27033971 -2.5330147 ]
d_j: [-0.35710757 -0.8755833   1.01853077]
```

In [386…

```
# use same evaluation method as 4
S_x, S_prime_x = cubic_spline_approx(x=x, x_values=x_val, a_j=a_j, b_j=b_j, c_j=c_j, d_j
print(f"Approximated f({x}) = {S_x}")
print(f"Approximated f'({x}) = {S_prime_x}")
```

```
Approximated f(0.25) = -0.132688671979105
Approximated f'(0.25) = 2.9080652582507405
```

Checking

$$f(x) = x\cos x - 2x^2 + 3x - 1$$

In [388…

```
eq = "x*cos(x)-2*x**2 + 3*x -1"
p_0 = 0.25
true_val = sip.eq_solver(eq, p_0)
true_val
```

Out[388…

```
-0.13277189457233884
```

In [390…

```
abs_er = abs(true_val - S_x)
print("Absolute error: ", abs_er)
```

```
Absolute error:  8.322259323384484e-05
```

$$f'(x) = -x\sin x + cosx - 4x + 3$$

In [392…

```
eq = "-x*sin(x) + cos(x) - 4*x + 3"
p_0 = 0.25
true_val = sip.eq_solver(eq, p_0)
true_val
```

Out[392…

```
2.907061431897014
```

In [394…

```
abs_er = abs(true_val - S_prime_x)
print("Absolute error: ", abs_er)
```

```
Absolute error:  0.0010038263537266445
```

Code for function evaluation

In [339…

```
import math
import regex
import pandas as pd

class Sipnayan:
    def __init__(self, math_object, regex, pd):
        self.math_object = math_object
        self.reg = regex
        self.pd = pd

    def number_solver(self, equation):
```

```python
        return eval(equation)

    def eq_solver(self, equation, var_val, var="x"):
        """"var != e
        Note to future romand: make varaible a list instead for equations beyond 2d
        """
        # print(f"Original: {equation}")
        equation = equation.replace(var, str(var_val))
        # print(f"Parsed: {equation}")
        # Check for other variable other than specified var
        if (self.check_for_other_var(equation, var)):
            return f"Letters detected aside from independent variable ({var})"
        operations = ["e", "cos", "sin", "tan", "ln"]
        if any(operation in equation for operation in operations):
            return eval(self.nested_handler(equation))
        return self.number_solver(equation)

    def trigo(self, trig_op, arg):
        match str(trig_op):
            case "cos":
                return self.math_object.cos(arg)
            case "sin":
                return self.math_object.sin(arg)
            case "tan":
                return self.math_object.tan(arg)
            case default:
                return "invalid argument"

    def exp_solve(self, arg):
        """For e^x with x as arg"""
        return self.math_object.exp(arg)

    def ln_solve(self, arg):
        return self.math_object.log(arg)

    def nested_handler(self, equation):
        operations = ["e\\^", "cos", "sin", "tan", "ln"]
        ops = ["e", "cos", "sin", "tan", "ln"]
        # print(f"Processing equation: {equation}")

        pat = rf'({"|".join(operations)})\(((?:[^\(\)]+|(?R))*)\)'
        # print(f"Regex pattern: {pat}")

        while any(operation in equation for operation in ops):
            match = regex.search(pat, equation)
            if not match:
                break

            opp = match.group(1)
            ovr_expression = match.group(0)
            argument = match.group(2)
            # print(f"Matched operation: {ovr_expression}, Argument: {argument}")

            if any(operation in argument for operation in ops):
                argument = self.nested_handler(argument)

            equation = self.special_operations(opp, ovr_expression, argument, equation)
            # print(f"Updated equation: {equation}")
```

```
        return equation

    def special_operations(self, opp, ovr_expression, argument, equation):
        # print(f"Processing {opp} with argument: {argument}")
        try:
            if "e" in opp:
                result = self.exp_solve(eval(argument))
            elif "ln" in opp:
                result = self.ln_solve(eval(argument))
            else:
                result = self.trigo(opp, eval(argument))
        except Exception as e:
            print(f"Error in {opp}: {e}")
            return equation

        updated_equation = equation.replace(ovr_expression, str(result))
        return updated_equation



    def check_for_other_var(self, equation, var):
        remove = ['cos', 'sin', 'tan', 'e', "ln"]
        for opp in remove:
            equation = equation.replace(opp,"")
        for i in equation:
            if i.isalpha() and i != var:
                return True
        return False

sip = Sipnayan(math, regex, pd)
```