Week 5 Assignment

Question 1:

```
def BestfitPolynomial(input_data, degree):
    x = [input_data[i][0] for i in range(len(input_data))]  # x_values
    y = [input_data[i][1] for i in range(len(input_data))]  # y_values
    co_matrix = []  # array created for the co-effecient matrix

    r = []
    index = 0
    dummy_degree = degree
    while dummy_degree <= 2 * degree:
        # at each row of the matrix the difference between the highest
        # and lowest degrees of x_values is exactly equal to degree
        for i in range(index, dummy_degree + 1):
            s = [x[j] ** i for j in range(len(x))]

            r.append(sum(s))
        co_matrix.append(r)
        r = []
        index += 1
        dummy_degree += 1
```

- The following code depicts the calculation of the co-efficient matrix for a polynomial fit (e.g., the first row of the matrix is [n,$\sum x$, $\sum x^2$, … , $\sum x^n$])
- Also note the highest degree in such summation is 2n and hence we run the loop while we don't reach 2n

```
•  while y_index <= degree:
       s = sum([y[i] * (x[i] ** y_index) for i in range(len(y))])
       b.append(s)
       y_index += 1
   # formulating the equation Ax=b
   A = np.array(co_matrix)
   B = np.array(b)
   sol = np.linalg.solve(A, B)  # solving via linalg
   sol1 = [abs(round(i, 8)) for i in sol] # rounding off the values
   a = min(x)
   b = max(x)
   x1 = np.linspace(a, b,num=100) # x_values for the plot
   # setting up a polynomial object to valuate a value at points and
   # using it in a plot
   p = Polynomial(sol1) # creating a polynomial object with the co-
   effecients
```

- The above code depicts the solution of the linear system Ax=B where x is the co-efficient of our fitted polynomial
- We have used **numpy. linalg** to solve it and have created a polynomials object with the solution vector

Question 2:

```
def BestfitPolynomial(degree):
    co_matrix = []

    r = []
    index = 0
    dummy_degree = degree
    # similar to question 1,however instead of taking sum we integrate
to get the co-effecients
    while dummy_degree <= 2 * degree: # since highest degree of the
x_value could be 2*degree
        for i in range(index, dummy_degree + 1):
            s = lambda x: x ** i
            r.append(list(integrate.quad(s, 0, np.pi))[0]) # integrate
with quadrature rule
        co_matrix.append(r)
        r = []
        index += 1
        dummy_degree += 1
```

- Question 2 is exactly similar to question 1 the only difference being in this case the co-efficient matrix is populated by **integrating** in the given range instead of summing it and also, we have used **quadrature rule** to integrate the functions

Question 3:

```
# function to evaluate nth legendre polynomial
def LegendrePolynomial(degree):
    p = Polynomial([-1, 0, 1]) # defining (x^2-1)
    s = Polynomial([1])
    # Multiplying the polynomial n times
    for i in range(degree):
        s = s * p
    # differentiating the polynomial n times
    for i in range(degree):
        s = s.derivative()
    l = (1 / ((2 ** degree) * math.factorial(degree))) * s
    # printing the legendre polynomial
    print(l)
```

- Using the Polynomial class that already exists we evaluate the nth degree Legendre Polynomial by suitably differentiating n times and multiplying the polynomial n times. The function takes degree of the required polynomial as input.

Question 4:

```
def legendre_function(self, x):
    s = 0
    for i in range(len(self.list_co)):
        s += (x ** i) * self.list_co[i]
    return s
```

- We first create the above function to return a polynomial in the form $a_0 + a_1x + a_2x^2 + .. + a_nx^n$ from the list of co-efficient $[a_0, a_1, \ldots, a_n]$

```
def legend_approximation(self, degree): # the method to approximate using nth degree legendre polynomial
    p = Polynomial()
    aj = []
    for i in range(degree + 1):
        # calculation of cj starts with weight function w(x)=1
        s = lambda x: p.LegendrePolynomial(i).legendre_function(x) ** 2
        # calculation of aj
        r = lambda x: p.LegendrePolynomial(i).legendre_function(x) * np.exp(x)
        # cj values
        cj = list(integrate.quad(s, -1, 1))[0]
        # calculating the co-effecients i.e. aj
        tmp = (1 / cj) * list(integrate.quad(r, -1, 1))[0]
        aj.append(tmp) # creating a list of co-effecients
    # estimating the polynomial
    p_leg = Polynomial([0])
    for i in range(degree + 1):
        p_leg = p_leg + aj[i] * p.LegendrePolynomial(i)
```

- First, we assert that we can use Legendre polynomial to approximate the function e^xbecause of its **orthogonality** w.r.t weight function **w(x)=1 in [-1,1]**
- We calculate the co-efficient of the Polynomial by using the available formulas with the help of integration
- After evaluation the co-efficients of the Polynomial we evaluate the Polynomial $\sum a_j \Phi_j$

Question 5:

```
# creating the n th degree chebyshev polynomial
def ChebyshevPolynomial(degree):
    p1 = Polynomial([1]) # 1st chebyshev polynomial
    p2 = Polynomial([0, 1]) # 2nd chebyshev polynomial
    i = 1
    # recursive calculation for chebyshev's polynomial with T_(n+1)(x)=2xT_n(x) - T_(n-1)(x)
    while i < degree:
        p = 2 * Polynomial([0, 1]) * p2 - p1
        p1 = p2
        p2 = p
        i += 1
    if degree == 0:
        print(p1)
    elif degree == 1:
        print(p2)
    else:
        print(p)
```

- We calculate the Chebyshev Polynomial with the help of the recurrence relation $\boldsymbol{T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).}$
- Also note that the first two polynomials are initiated to start the process

Question 6 :

```
def OrthogonalCheck(self):
    c = []
    p = Polynomial()
    # storing the first 5 chebyshev polynomials
    for i in range(5):
        c.append(p.ChebyshevPolynomial(i))
    # Checking of orthogonality
    for i in range(5):
        for j in range(5):
            # taking the product with the weight function
w(x)=1/sqrt(1-x^2)
            s = lambda x: c[i].Chebyshevfunction(x) *
c[j].Chebyshevfunction(x) * 1 / math.sqrt(1 - x ** 2)
            int_val = integrate.quad(s, -1, 1)
            print("Intrgration of", i, "degree Polynomial and", j,
"degree Polynomial:\n",
                  round(abs(int_val[0]), 8))
```

- To check the orthogonality of the first 5 Chebyshev Polynomials we integrate then mutually w.r.t to the weight function $\boldsymbol{w(x) = \frac{1}{\sqrt{1-x^2}}}$

Question 7:

```
for i in range(degree + 1):
    s = lambda x: np.exp(x) * np.cos(i * x)  # calculation for ak
    r = lambda x: np.exp(x) * np.sin(i * x)  # calculation of bk
    a.append((1 / np.pi) * list(integrate.quad(s, -np.pi, np.pi))[0])
# storing ak values
    b.append((1 / np.pi) * list(integrate.quad(r, -np.pi, np.pi))[0])
# stroring bk values
```

- The above code depicts the calculation of $\boldsymbol{a_k \text{ and } b_k}$.

```
s = a[0] / 2 # since the 1st term in fourier series is a_0/2
sn = []
# calculation of S_n(x) for all x
for i in x:
    for j in range(1, degree + 1):
        s = s + a[j] * np.cos(i * j) + b[j] * np.sin(j * i)
    sn.append(s)
    s = a[0] / 2
```

- The above code depicts the value of the $\boldsymbol{S_n(x)}$ for a given set of x values.
- Notice that we initialize with $\boldsymbol{\frac{a_0}{2}}$ because the Fourier series starts with that value .