MAT 5153 Mathematical Foundations of Data Analytics

Homework 1

1. **Step 1:** Define a time series function. e.g., $f(t) = sin(t)$

```python
import numpy as np
def create_time_series_data(t):
"""
Create a time series dataset with noise.

Parameters:
time (ndarray): An array of time values.

Returns:
ndarray: An array of time series data.
"""
signal = np.sin(t)
return signal

```

2. **Step 2:** Using numpy add random noise to simulate real world data.

```python
noise = 0.5 * np.random.normal(a,b,N)
noisy_signal = signal + noise

```

3. **Step 3:** Design integrator functions (Trapezoid rule and Simpson's rule) to implement numerical integration

```python
def trapezoidal_rule(f, a, b, n):
"""
```

```
    Approximate the integral of f from a to b using the trapezoidal rule with n intervals.

    Parameters:
    f (function): The function to integrate.
    a (float): The start point of the interval.
    b (float): The end point of the interval.
    n (int): The number of subintervals.

    Returns:
    float: The approximate integral of f from a to b.
    """
    h = (b - a) / n # width of each subinterval
    x = np.linspace(a,b,n+1)
    y = f(x)

    integral = (h/2) * (y[0] + 2*np.sum(y[1:-1]) + y[-1])
    return integral

def simpsons_rule(f, a, b, n):
    """
    Approximate the integral of f from a to b using Simpson's rule with n intervals.

    Parameters:
    f (function): The function to integrate.
    a (float): The start point of the interval.
    b (float): The end point of the interval.
    n (int): The number of subintervals (must be even).

```

```
        Returns:
        float: The approximate integral of f from a to b.
        """

        h = (b - a) / n # width of each subinterval
        x = np.linspace(a,b,n+1)
        y = f(x)

        integral = y[0] + y[-1]
        integral += 4 * np.sum(y[1:-1:2])
        integral += 4 * np.sum(y[2:-1:2])
        integral *= h/3
        return integral
```

4. **step 4:** Design basis functions

```
        def cosine_basis_function(n):
        """
        Returns a cosine basis function: cos(n * t).

        Parameters:
            n (int): Frequency parameter of the cosine function.

        Returns:
            function: A function `f(t)` representing cos(n*t).
        """
        return lambda t: np.cos(n * t)

    def polynomial_exponential_basis_function(n):
        """
```

```
        Returns a polynomial exponential basis function: t^n * exp(nt).

        Parameters:
            n (int): Degree of the polynomial.

        Returns:
            function: A function `f(t)` representing t^n * exp(nt).
        """
        return lambda t:  np.exp(t/n)

```

5. **Step 5:** Create a function to compute inner product of two basis functions using either of the numerical integrators.

```
        def inner_product(f,g, a, b, N, integrator):
        """
        Compute the inner product of two functions f and g over the interval [a,b].

        Parameters:
        f (function): The first function.
        g (function): The second function.
        a (float): The lower bound of the interval.
        b (float): The upper bound of the interval.
        N (int): The number of points to use in the approximation.
        integrator (function): The numerical integration function to use.

        Returns:
        float: The inner product of f and g over the interval [a,b].
        """
        integrand = lambda t: np.multiply(f(t),g(t), dtype=object)
```

```python
        inner_product = integrator(integrand, a, b, N)
        return inner_product

```

6. **Step 6:** Create a Gram matrix function for the linear system $G_{ij} =< u_i, uj >$

```python
        def create_gram_matrix(basis_list, a, b, integrator):
        """
        Create the Gram matrix for a given set of basis functions.

        Parameters:
        basis_list (list): A list of basis functions.
        a (float): The lower bound of the interval.
        b (float): The upper bound of the interval.
        N (int): The number of points to use in the approximation.
        integrator (function): The numerical integration function to use.

        Returns:
        ndarray: The Gram matrix for the basis functions.
        """
        M = len(basis_list)
        gram_matrix = np.zeros((M, M))
        for i in range(M):
            for j in range(M):
                f = basis_list[i]
                g = basis_list[j]
                gram_matrix[i, j] = inner_product(f, g, a, b, M, integrator)
        return gram_matrix

```

7. **Step 7:** Create a right-hand side vector for the linear system $r_i =< u_i, f >$

```python
    def right_hand_vector(basis_list, signal,a,b, integrator):
    """
    Create the right-hand vector for the linear system
    parameters:
    basis_list (list): A list of basis functions.
    signal (function): The signal function.
    a (float): The lower bound of the interval.
    b (float): The upper bound of the interval.
    N (int): The number of points to use in the approximation.
    integrator (function): The numerical integration function to use.

    Returns:
    ndarray: The right-hand vector.
    """
    N = len(basis_list)
    right_hand_vector = np.zeros(N)
    for i in range(N):
        right_hand_vector[i] = inner_product(basis_list[i], signal, a, b, N, integrator)
    return right_hand_vector

```

8. **Step 8:** Solve the Linear system $Gc = r$ to find the coefficients c. Write a approximation function using the coefficients found from solving the linear system.

```python
    def function_approximator(t,c, basis):
    """
    Approximates a function value at a given point using a linear combination of basis functions.

    Parameters:
```

```
        t (float): The point at which to evaluate the approximated function.
        c (list of float): Coefficients for the linear combination of basis functions.
        basis (list of callable): List of basis functions, each of which takes a single argument (t).

        Returns:
        float: The approximated function value at point t.
        """
        return sum(c[k] * basis[k](t) for k in range(len(basis)))

```

9. **Step 9:** Using different combinations of basis functions and integrators observe the approximation.

```
        a = args.initial_value
    b = args.final_value
    N = args.interval
    time = np.linspace(a,b,N)
    signal = create_time_series_data(time)

    noise = 0.5 * np.random.normal(a,b,N)
    noisy_signal = signal + noise

    integrator_01 = trapezoidal_rule
    basis_01 = cosine_basis_function

    integrator_02 = trapezoidal_rule
    basis_02 = polynomial_exponential_basis_function

    integrator_03 =  simpsons_rule
    basis_03 = cosine_basis_function
```

```

integrator_04 =  simpsons_rule
basis_04 = polynomial_exponential_basis_function



basis_list_01 = [basis_01(n) for n in range(1,len(time))]
G_01 = create_gram_matrix(basis_list_01, a, b, integrator_01)
r_01 = right_hand_vector(basis_list_01, lambda t: noisy_signal, a, b,
integrator_01)
c_01 = np.linalg.solve(G_01, r_01)
approximator_01 = function_approximator(time,c_01,basis_list_01)

basis_list_02 = [basis_02(n) for n in range(1,len(time))]
G_02 = create_gram_matrix(basis_list_02, a, b, integrator_02)
r_02 = right_hand_vector(basis_list_02, lambda t: noisy_signal, a, b,
integrator_02)
c_02 = np.linalg.solve(G_02, r_02)
approximator_02 = function_approximator(time,c_02,  basis_list_02)

basis_list_03 = [basis_03(n) for n in range(1,len(time))]
G_03 = create_gram_matrix(basis_list_03, a, b, integrator_03)
r_03 = right_hand_vector(basis_list_03, lambda t: noisy_signal, a, b,
integrator_03)
c_03= np.linalg.solve(G_03, r_03)
approximator_03 = function_approximator(time, c_03, basis_list_03)

basis_list_04 = [basis_04(n) for n in range(1,len(time))]
G_04 = create_gram_matrix(basis_list_04, a, b, integrator_04)
r_04 = right_hand_vector(basis_list_04, lambda t: noisy_signal, a, b,
integrator_04)
```

```
c_04 = np.linalg.solve(G_04, r_04)
approximator_04 = function_approximator(time, c_04, basis_list_04)

```

10. **Step 10:** Plot the different results

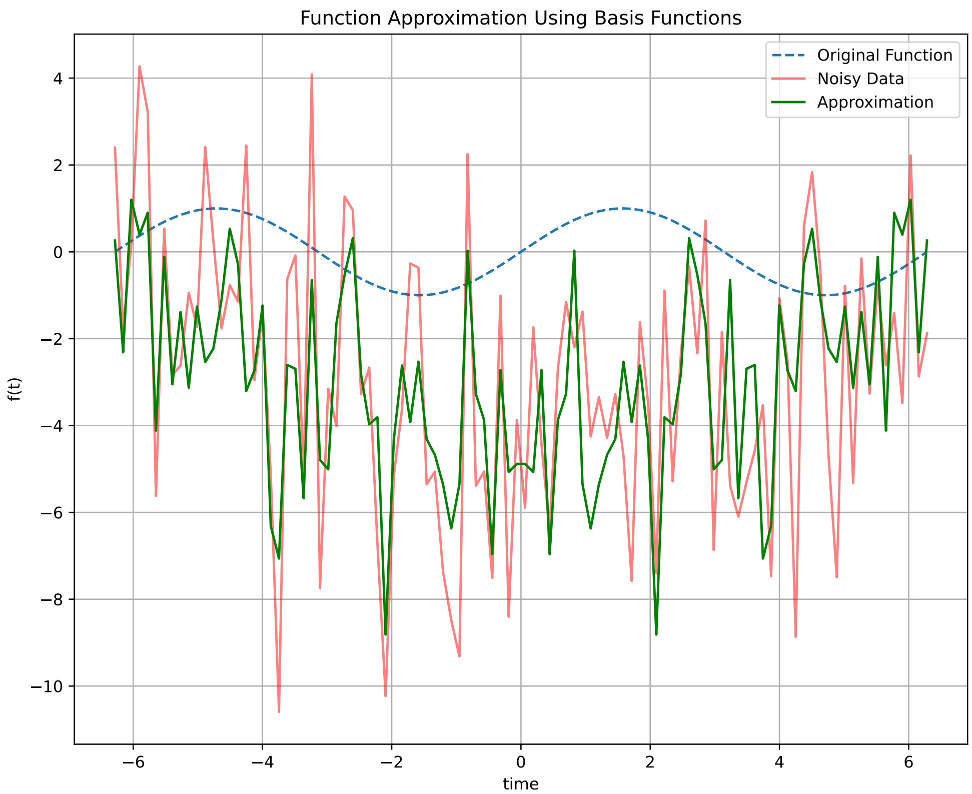
    (a) Basis function: Cosine Basis Function, Integrator: Trapezoid rule

```
            plt.figure(1, figsize=(10, 8))
            plt.plot(time, signal, label="Original Function", linestyle="--")
            plt.plot(time, noisy_signal, label="Noisy Data", color="red", alpha=0.5)
            plt.plot(time, approximator_01 , label="Approximation",color="green")
            plt.xlabel("time"), plt.ylabel("f(t)")
            plt.title("Function Approximation Using Basis Functions")
            plt.legend()
            plt.grid(True)
            plt.savefig(os.path.join(fig_path,'fig_01.pdf'))

```

    (b) Basis function: Exponential Basis Function, Integrator: Trapezoid rule

```
            plt.figure(2, figsize=(10, 8))
            plt.plot(time, signal, label="Original Function", linestyle="--")
            plt.plot(time, noisy_signal, label="Noisy Data", color="red", alpha=0.5)
            plt.plot(time, approximator_02 , label="Approximation",color="blue")
            plt.xlabel("time"), plt.ylabel("f(t)")
            plt.title("Function Approximation Using Basis Functions")
            plt.legend()
            plt.grid(True)
```

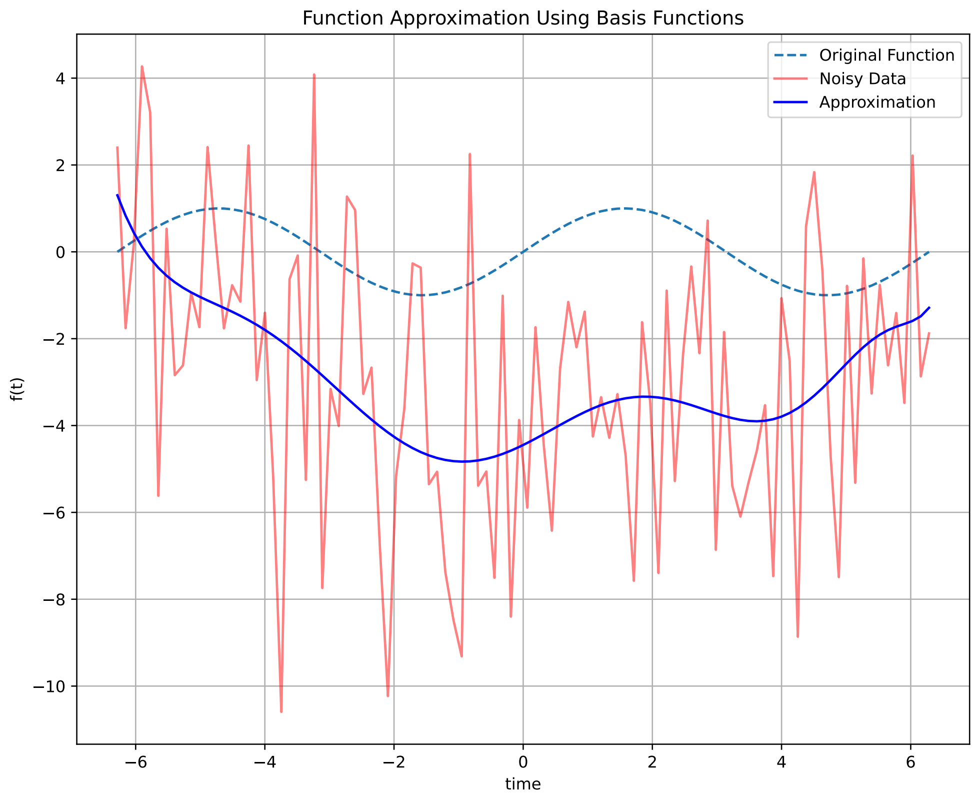



```
            plt.savefig(os.path.join(fig_path,'fig_02.pdf'))

```

(c) Basis function: Cosine Basis Function, Integrator = Simpson's rule

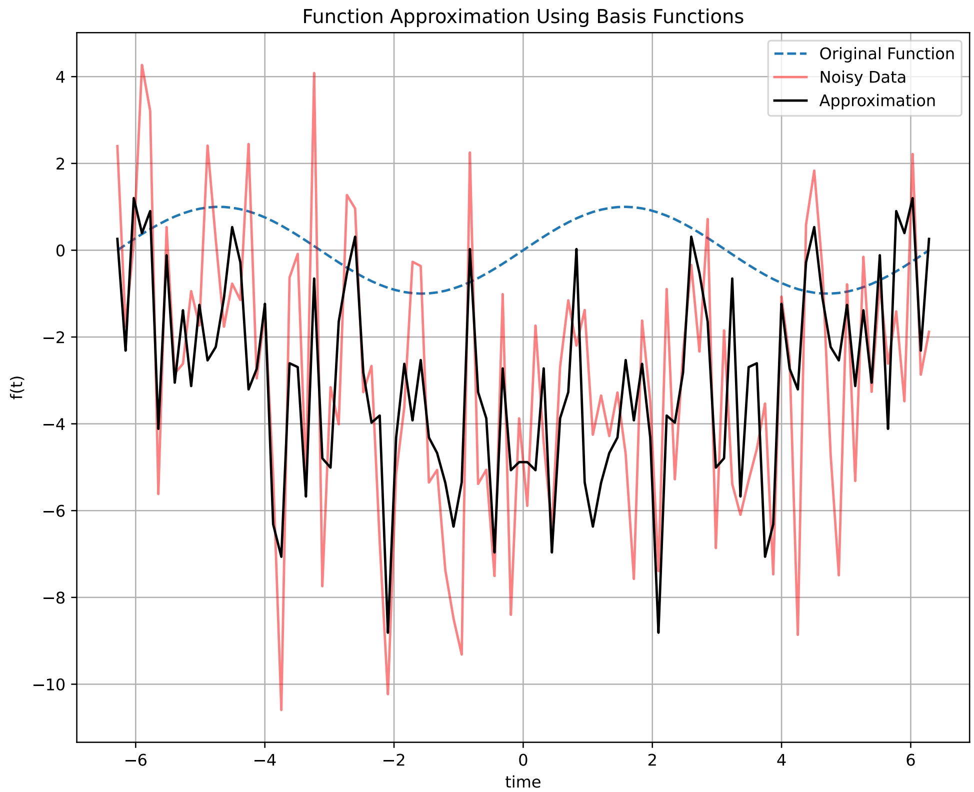
```
            plt.figure(3, figsize=(10, 8))
            plt.plot(time, signal, label="Original Function", linestyle="--")
            plt.plot(time, noisy_signal, label="Noisy Data", color="red", alpha=0.5)
            plt.plot(time, approximator_03 , label="Approximation",color="black")
            plt.xlabel("time"), plt.ylabel("f(t)")
            plt.title("Function Approximation Using Basis Functions")
            plt.legend()
            plt.grid(True)
            plt.savefig(os.path.join(fig_path,'fig_03.pdf'))
```

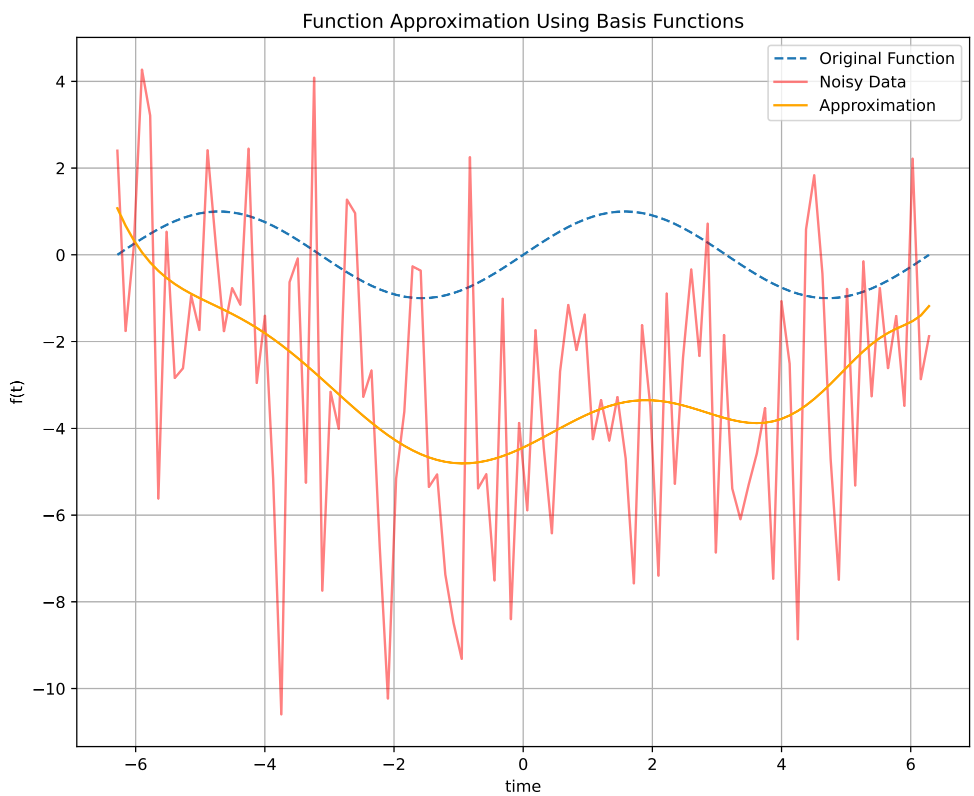


(d) Basis function: Exponential Basis Function, Integrator = Simpson's rule

```
            plt.figure(4, figsize=(10, 8))
            plt.plot(time, signal, label="Original Function", linestyle="--")
            plt.plot(time, noisy_signal, label="Noisy Data", color="red", alpha=0.5)
            plt.plot(time, approximator_04 , label="Approximation",color="orange")
            plt.xlabel("time"), plt.ylabel("f(t)")
            plt.title("Function Approximation Using Basis Functions")
            plt.legend()
            plt.grid(True)
            plt.savefig(os.path.join(fig_path,'fig_04.pdf'))

```


Function Approximation Using Basis Functions
Original Function
Noisy Data
Approximation
f(t)
time
4
2
0
−2
−4
−6
−8
−10
−6
−4
−2
0
2
4
6


Function Approximation Using Basis Functions
Original Function
Noisy Data
Approximation
f(t)
time
4
2
0
−2
−4
−6
−8
−10
−6
−4
−2
0
2
4
6