Fourier Analysis using Python

Valentina Watanabe

April 2024

1 Introduction

In this computational lab, the primary objective was to use Fourier analysis techniques in Python, including tools like Simpson's rule for numerical integration and Discrete Fourier Transforms. Then various periodic and non-periodic series were analysed and computed along with their discrete Fourier transforms. Fourier Analysis makes it easier to break down both periodic and aperiodic signals into sinusoidal components so that they can be used to reconstruct the original signal.

2 Theory

Fourier Analysis is an important tool used for deconstructing signals into simpler components. Usually, signs are not periodic, in which case Fourier Transforms are used, where signals are converted from the time into the frequency domain.

For periodic functions, Simpson's rule is a numerical method for approximating integrals between limits a and b. It follows the equation,

$$\int_a^b f(x)\,dx \approx \tfrac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right]$$

where n is an even number and h = (b-a)/n. We also use the coefficient equations,

Non-periodic functions can also be extended with a combination of cosine and sine functions over a continuous range of frequencies ω. Fourier transforms for non-periodic functions can be represented using,

$$a_0 = \frac{1}{T}\int_0^T dt\ f(t)$$

$$a_k = \frac{2}{T}\int_0^T dt\ f(t)\cos(k\omega t) \qquad \text{k=1,2, ...}$$

$$b_k = \frac{2}{T}\int_0^T dt\ f(t)\sin(k\omega t) \qquad \text{k=1,2, ...}$$

$$\begin{aligned} f(t) &= \int_0^\infty (a(\omega)\cos(\omega t) + b(\omega)\sin(\omega t))\, d\omega \\ a(\omega) &= \frac{1}{\pi}\int_{-\infty}^\infty f(t)\cos(\omega t)\, dt \\ b(\omega) &= \frac{1}{\pi}\int_{-\infty}^\infty f(t)\sin(\omega t)\, dt. \end{aligned}$$

These equations are often replaced using Discrete Fourier Transforms with finite sums over a specified interval. Taking N sampled of a signal with h intervals we can compute the Fourier Transform as,

$$F_n = \sum_{m=0}^{N-1} f_m e^{-i\omega_n t_m} = \sum_{m=0}^{N-1} f_m e^{\frac{-2\pi imn}{N}}$$

and

$$f_m = \frac{1}{N}\sum_{n=0}^{N-1} F_n e^{+i\omega_n t_m} = \frac{1}{N}\sum_{n=0}^{N-1} F_n e^{\frac{+2\pi imn}{N}}$$

3 Methodology

3.1 Exercise 1

Simpson's rules were incorporated into a constructed script to solve definite integrals over a set boundary by defining a function with the appropriate equations. This was then extended to compute the Fourier series for any input function.

The equations for periodic functions were solved for a range of k numbers, obtaining coefficients a_k and b_k. The following functions were then evaluated and graphed. Their Fourier series coefficients were also calculated and graphed as discrete points.

$$
\begin{gathered}
f(t) = \sin(\omega t) \\
f(t) = \cos(\omega t) + 3\cos(2\omega t) - 4\cos(3\omega t) \\
f(t) = \sin(\omega t) + 3\sin(3\omega t) + 5\sin(5\omega t) \\
f(t) = \sin(\omega t) + 2\cos(3\omega t) + 3\sin(5\omega t).
\end{gathered}
\tag{1}
$$

3.2 Exercise 2

This part consisted of graphing the square and rectangular functions and using Fourier analysis to reconstruct them and study the respective accuracy. For the square function, the following equations were used.

$$
f(\theta) = \begin{cases} 1 & 0 \leq \theta \leq \pi \\ -1 & \pi < \theta \leq 2\pi \end{cases}
$$

$$
\begin{aligned}
a_k &= 0 \qquad\qquad k = 1,3,5\ldots \\
b_k &= \begin{cases} 4/\pi k & k = 1,3,5\ldots \\ 0 & k = 2,4,6\ldots \end{cases}
\end{aligned}
$$

The rectangular function was represented using,

$$
f(\omega t) = \begin{cases} 1 & 0 < \omega t < \omega\tau \\ -1 & \omega\tau < \omega t < 2\pi \end{cases}
$$

$$
\begin{aligned}
a_0 &= (2/\alpha) - 1 \\
a_k &= (2/k\pi)\sin(2k\pi/\alpha) \qquad k = 1,2,3\ldots \\
b_k &= (2/k\pi)(1 - \cos(2k\pi/\alpha)) \qquad k = 1,2,3\ldots
\end{aligned}
$$

3.3 Exercise 3

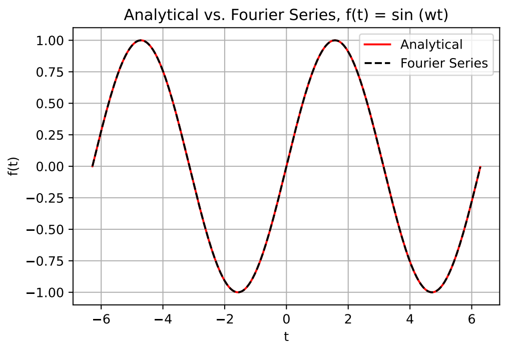
The focus of exercise three was to apply Discrete Fourier Transforms. The script is adjusted to calculate the real and imaginary parts of the Fourier components, denoted as $F_n real$ and $F_n imaginary$, respectively. The program is also used to evaluate and print the sampling rate ν_s (samples per unit time) and the fundamental frequency ω_1 to verify the accuracy of calculated Fourier components. Then, for a given function $f(t) = \sin(0.45\pi t)$ with N=128 and h = 0.1, the plot is built over a total time τ = N x h along with the sampled points. This was repeated with ideal sampling, choosing a value of h, and keeping N fixed. The back-transform was also calculated and plotted.

4 Results

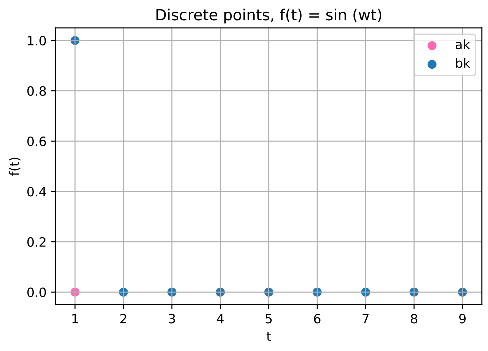
4.1 Exercise 1

The first integral evaluated was e^x from 0 to 1 with n = 8 steps, giving a result of 1.7182818284650119. This highly matches the analytical result of 1.71828183. The then extended program evaluated the Fourier Coefficients of ak and bk of the mentioned functions. Figures 1, 2, 3, and 4 show the series of their respective plots and coefficients.

For the first function, we expect to obtain b1 = 1, for the second a1 = 1, a2 = 3, a3 = -4 and so on, because these numbers represent the coefficients of their respective terms. bk coefficients are zero for the first function as it has no cosine terms, and the same of ak for the second function, as it has no sine terms.

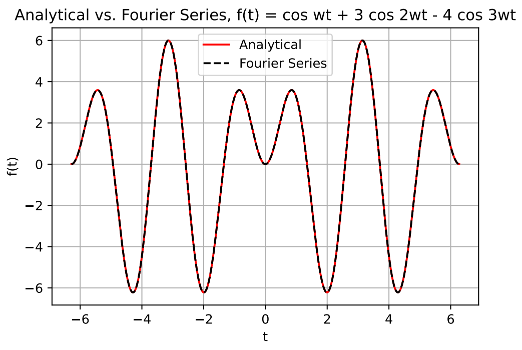


Analytical vs. Fourier Series

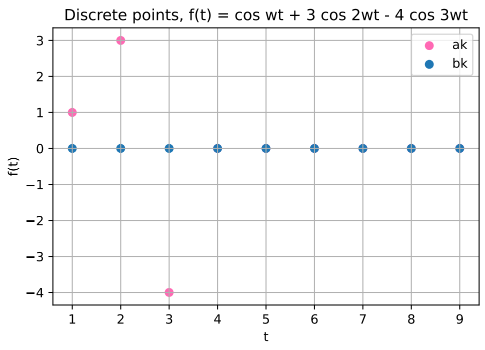


Fourier coefficients

Figure 1: Series and discrete points for first periodic function.

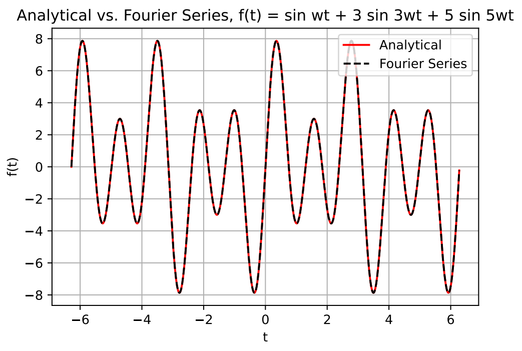


Analytical vs. Fourier Series

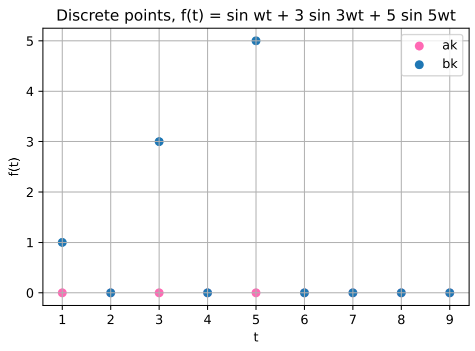


Fourier coefficients

Figure 2: Series and discrete points for second periodic function.

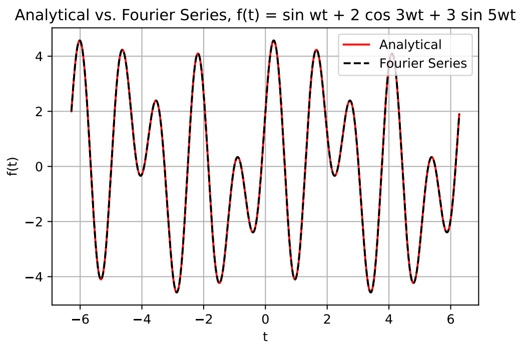


Analytical vs. Fourier Series

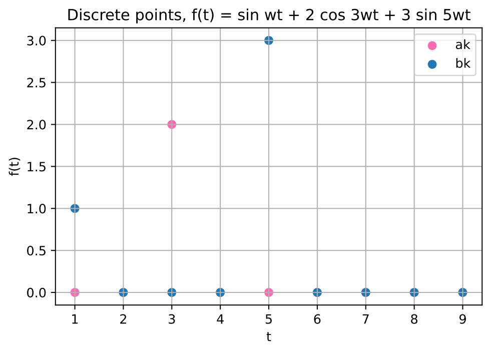


Fourier coefficients

Figure 3: Series and discrete points for third periodic function.



Analytical vs. Fourier Series

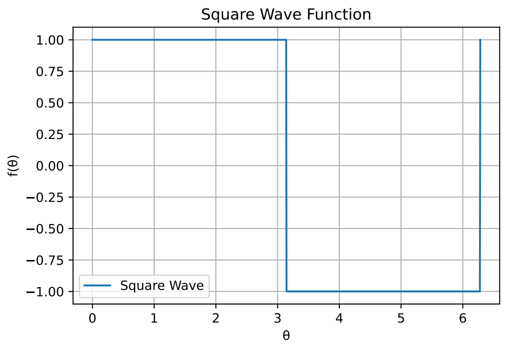


Fourier coefficients

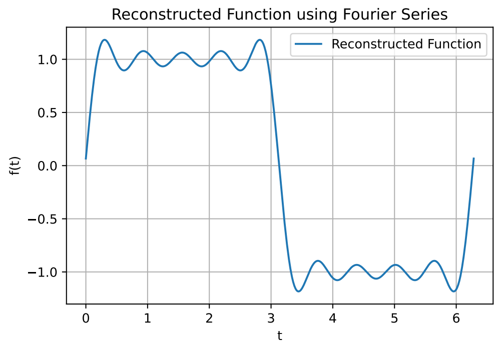
Figure 4: Series and discrete points for fourth periodic function.

4.2 Exercise 2

For this exercise, the previous code was used and a definition of the square wave function was added. Its Fourier coefficients were also defined to construct the square wave function using Fourier analysis. In Figure 5, we can see the square wave function with the reconstruction. Figure 7 is the reconstructed function with various increasing n terms.



Square Function



Reconstructed Square Function

Figure 5: Square function and Reconstructed function

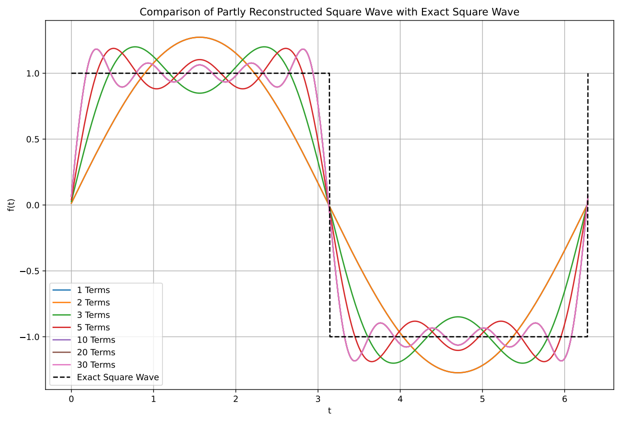


Figure 6: Square wave functions and Reconstructed function

The same was done for the respective rectangular function. As we can see for both cases, the increasing number of terms is related directly to the increasing accuracy of the reconstructed function.

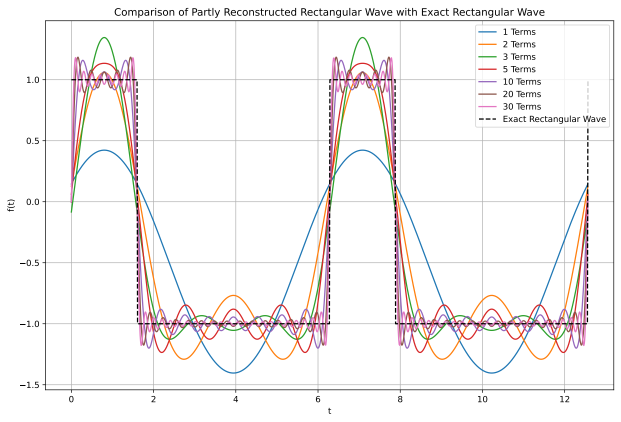


Figure 7: Rectangular wave functions and Reconstructed function

4.3 Exercise 3

The sampling rate ν_s (samples per unit time) and the fundamental frequency ω_1 were given to be 0.1 and 0.4908738521234052 respectively. Function $f(t) = \sin(0.45\pi t)$ was used to plot,

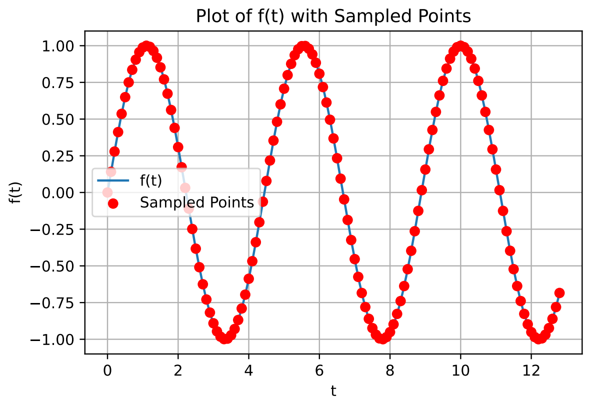


Figure 8: Plot of $f(t) = \sin(0.45\pi t)$

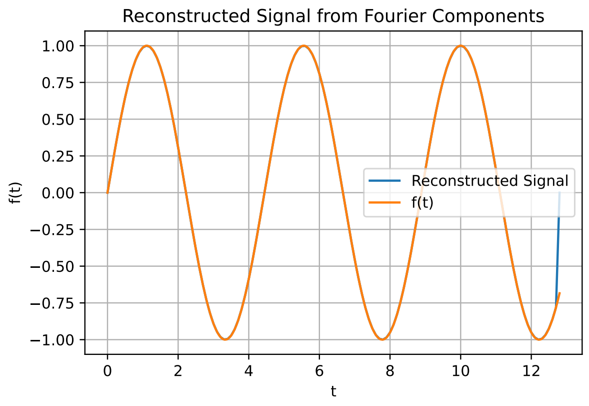


Figure 9: Plot of f(t) with reconstructed signal

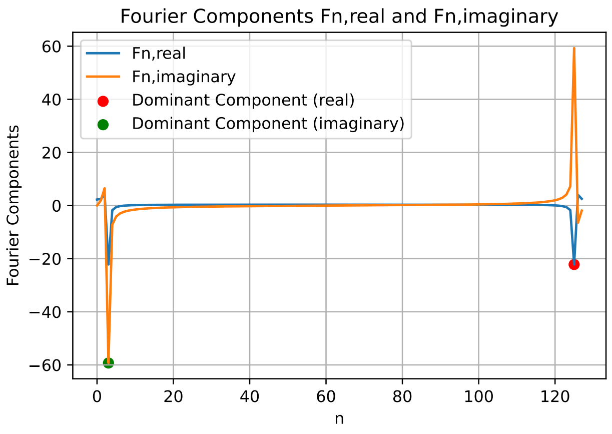


Figure 10: Fourier components F(n)

The dominant component index of the real part is 125, and the dominant component index for the imaginary part is 3. The expected value of n based on the function is 2.8800000000000003.

For the next section, the Fourier is performed with the ideal sampling time for the periodic signal. The frequency ω of f1 is 0.45π. N was kept at 128. h can be chosen to be h $= \frac{2\pi}{(\omega * N)} = \frac{2}{0.45*128} = 0.034722222222222224$. The expected value of n based on the function is 1.0.

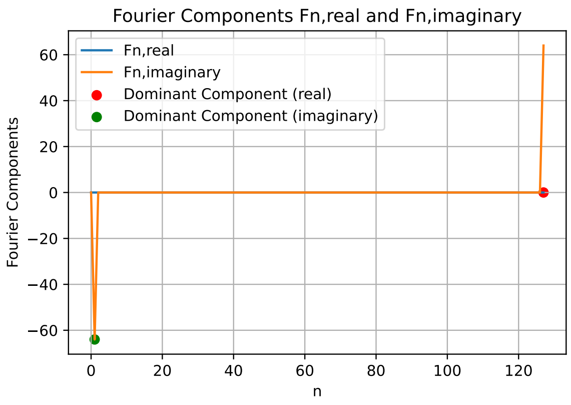
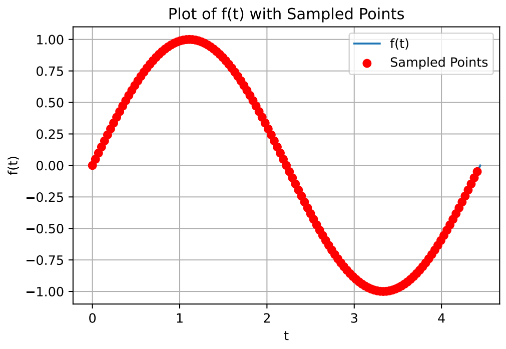
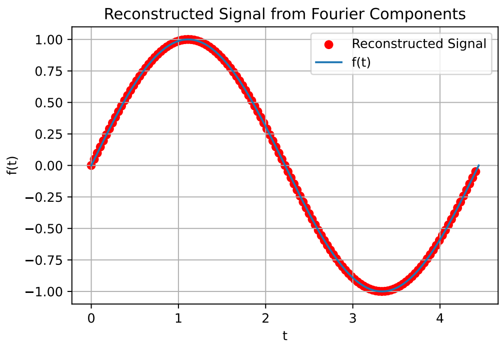


Figure 11: Fourier Components



Plot of f(t)



Reconstructed f(t)

Figure 12: Plot of f(t) with sampled points and reconstructed signal

5 Conclusion

Fourier series for periodic signals and Fourier Transform for aperiodic signals serve as a method of analysis by dissecting signals into their component harmonic vibrations. In this lab, Fourier analysis techniques were applied in Python, using tools like Simpson's rule for integration to reconstruct periodic and non-periodic series. The importance of Fourier Analysis can be shown through its ability to break down signals into simpler sinusoidal components for their analysis. This was done through the evaluation of Fourier coefficients, plotting of functions, and scrutiny of reconstructed signals. Furthermore, the use of Discrete Fourier Transforms allowed for the accurate computation of Fourier components. As demonstrated through the exercises, the increasing accuracy of reconstructed functions with an augmented number of Fourier components proved the efficacy of Fourier analysis.

6 Appendix

```
Exercise 1:
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 5 13:36:15 2024

@author: vwitch
"""

import numpy as np
import matplotlib.pylab as plt

def function_to_integrate(x):
    return np.exp(x)

def simpsons_rule(func, a, b, n):
    h = (b - a) / n
    x = [a + i * h for i in range(n+1)]
    y = [func(x_val) for x_val in x]
    int = h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-2:2]) + y[-1])
    return int

# Test the function
a = 0  # Lower limit
b = 1  # Upper limit
n = 8  # Number of steps (even)

integral = simpsons_rule_integration(function_to_integrate, a, b, n)
print("Integral:", integral)

def simp(f, a, b, n):
    h = (b - a) / n
    x = np.linspace(a, b, n+1)
    y = f(x)
    return h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) +
    y[-1])

def simpa0(f, a, b, n):
    h = (b - a) / n
    x = np.linspace(a, b, n+1)
    y = fun(x)
    return (1/b) * h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) + y[-1])

def simpak(f, a, b, n, k):
    h = (b - a) / n
    x = np.linspace(a, b, n+1)
    y = fun(x) * np.cos(k * x)
    return (2/b) * h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) + y[-1])

def simpbk(f, a, b, n, k):
    h = (b - a) / n
    x = np.linspace(a, b, n+1)
    y = fun(x) * np.sin(k * x)
    return (2/b) * h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) + y[-1])

def fourierana(n, x, f, k):
    k_array = np.arange(1, k)
    fourier_transform = simpa0(f, 0, 2*np.pi, n)
    for k_in_the_array in k_array:
        fourier_transform+=simpak(f, 0, 2*np.pi, n, k_in_the_array) *
        np.cos(k_in_the_array * x) + simpbk(f, 0, 2*np.pi, n,
        k_in_the_array) * np.sin(k_in_the_array * x)
    return fourier_transform


def f1 (x):
    f = np.sin(x)
    return f

x = np.arange(-2*np.pi, 2*np.pi, 0.01)
plt.plot(x, np.sin(x), 'r',label = 'Analytical')
plt.plot(x, fourierana(8, x, f1, 3), 'k', linestyle = 'dashed', label
= 'Fourier Series')

plt.legend()
plt.grid(True)
plt.title('Analytical vs. Fourier Series, f(t) = sin (wt)')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.show()


def f2 (x):
    f = np.cos(x) + 3*np.cos(2*x) - 4*np.cos(3*x)
    return f

x = np.arange(-2*np.pi, 2*np.pi, 0.01)
plt.plot(x, np.cos(x) + 3*np.cos(2*x) - 4*np.cos(3*x), 'r',label =
'Analytical')
plt.plot(x, fourierana(50, x, f2, 5), 'k', linestyle = 'dashed', label
= 'Fourier Series')
plt.legend()
plt.grid(True)
plt.title('Analytical vs. Fourier Series, f(t) = cos wt + 3 cos 2wt - 4 cos 3wt')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.show()

def f3 (x):
    f = np.sin(x) + 3*np.sin(3*x) + 5*np.sin(5*x)
    return f

x = np.arange(-2*np.pi, 2*np.pi, 0.01)
plt.plot(x, np.sin(x) + 3*np.sin(3*x) + 5*np.sin(5*x), 'r',label =
'Analytical')
plt.plot(x, fourierana(50, x, f3, 6), 'k', linestyle = 'dashed', label
= 'Fourier Series')
plt.legend()
plt.grid(True)
plt.title('Analytical vs. Fourier Series, f(t) = sin wt + 3 sin 3wt +
5 sin 5wt')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.show()

def f4 (x):
    f = np.sin(x) + 2*np.cos(3*x) + 3*np.sin(5*x)
    return f

x = np.arange(-2*np.pi, 2*np.pi, 0.01)
plt.plot(x, np.sin(x) + 2*np.cos(3*x) + 3*np.sin(5*x), 'r',label =
'Analytical')
plt.plot(x, fourierana(50, x, f4, 6), 'k', linestyle = 'dashed', label
= 'Fourier Series')
plt.legend()
plt.grid(True)
plt.title('Analytical vs. Fourier Series, f(t) = sin wt + 2 cos 3wt +
3 sin 5wt')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.show()


k_array = np.arange(1, 10, 1)
ak_array = []
bk_array = []
```

```
for k in k_array:
    ak = simpak(f1, 0, 2 * np.pi, n, k)
    bk = simpbk(f1, 0, 2 * np.pi, n, k)
    ak_array.append(ak)
    bk_array.append(bk)

plt.scatter(k_array, ak_array, color = 'hotpink', label = 'ak')
plt.scatter(k_array, bk_array, label = 'bk')
plt.ylabel('f(t)')
plt.xlabel('t')
plt.title('Discrete points, f(t) = sin (wt)')
plt.legend()
plt.grid(True)
plt.show()


k_array = np.arange(1, 10, 1)
ak_array = []
bk_array = []

for k in k_array:
    ak = simpak(f2, 0, 2 * np.pi, n, k)
    bk = simpbk(f2, 0, 2 * np.pi, n, k)
    ak_array.append(ak)
    bk_array.append(bk)

plt.scatter(k_array, ak_array, color = 'hotpink',label = 'ak')
plt.scatter(k_array, bk_array, label = 'bk')
plt.legend()
plt.grid(True)
plt.title('Discrete points, f(t) = cos wt + 3 cos 2wt - 4 cos 3wt')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.show()

k_array = np.arange(1, 10, 1)
akray = []
bkray = []

for k in k_array:
    ak = simpak(f3, 0, 2 * np.pi, n, k)
    bk = simpbk(f3, 0, 2 * np.pi, n, k)
    akray.append(ak)
    bkray.append(bk)
```

```
plt.scatter(k_array, akray, color = 'hotpink',label = 'ak')
plt.scatter(k_array, bkray, label = 'bk')
plt.legend()
plt.grid(True)
plt.title('Discrete points, f(t) = sin wt + 3 sin 3wt + 5 sin 5wt')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.show()
#

k_array = np.arange(1, 10, 1)
akray = []
bkray = []

for k in k_array:
    ak = simpak(f4, 0, 2 * np.pi, n, k)
    bk = simpbk(f4, 0, 2 * np.pi, n, k)
    akray.append(ak)
    bkray.append(bk)

plt.scatter(k_array, akray, color = 'hotpink', label = 'ak')
plt.scatter(k_array,bkray, label = 'bk')
plt.legend()
plt.grid(True)
plt.title('Discrete points, f(t) = sin wt + 2 cos 3wt + 3 sin 5wt')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.show()
```

Exercise 2:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sun Apr 14 21:40:56 2024

@author: vwitch
"""

import numpy as np
import matplotlib.pylab as plt


def simpa0(f, a, b, n):
    h = (b - a) / n
    x = np.linspace(a, b, n+1)
    y = f(x)
    return (1/b) * h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) + y[-1])

def simpak(f, a, b, n, k):
    h = (b - a) / n
    x = np.linspace(a, b, n+1)
    y = f(x) * np.cos(k * x)
    return (2/b) * h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) + y[-1])

def simpbk(f, a, b, n, k):
    h = (b - a) / n
    x = np.linspace(a, b, n+1)
    y = f(x) * np.sin(k * x)
    return (2/b) * h/3 * (y[0] + 4*np.sum(y[1:-1:2]) + 2*np.sum(y[2:-1:2]) + y[-1])

def fourierana(n, x, f, k):
    k_array = np.arange(1, k)
    fourier_transform = simpa0(f, 0, 2*np.pi, n)
    for k_in_the_array in k_array:
        fourier_transform+=simpak(f, 0, 2*np.pi, n, k_in_the_array) *
        np.cos(k_in_the_array * x) + simpbk(f, 0, 2*np.pi, n, k_in_the_array)
        * np.sin(k_in_the_array * x)
    return fourier_transform

def square_wave(theta):
    result = np.zeros_like(theta)  # Initialize result array
    # Define the square wave function
    for i, value in enumerate(theta):
        norm_theta = value % (2*np.pi)  # Normalizing theta to [0, 2pi)
        if 0 <= norm_theta <= np.pi:
            result[i] = 1
        else:
            result[i] = -1
    return result

def analytical_ak(k):
    return 0

def analytical_bk(k):
    if k % 2 == 0:
        return 0
    else:
        return 4 / (k * np.pi)

# Define the range of k values
k_values = np.arange(1, 10)
```

```
# Compute the numerical Fourier coefficients
ak_array_numerical = []
bk_array_numerical = []
for k in k_values:
    ak_numerical = simpak(square_wave, 0, 2 * np.pi, 200, k)
    if k % 2 != 0:
        bk_numerical = simpbk(square_wave, 0, 2 * np.pi, 200, k)
    else:
        bk_numerical = 0
    ak_array_numerical.append(ak_numerical)
    bk_array_numerical.append(bk_numerical)

# Convert to numpy array
ak_array_numerical = np.array(ak_array_numerical)
bk_array_numerical = np.array(bk_array_numerical)

# Compute the analytical Fourier coefficients
ak_array_analytical = np.array([analytical_ak(k) for k in k_values])
bk_array_analytical = np.array([analytical_bk(k) for k in k_values])

# Print and compare the results
print("k\tAnalytical a_k\t\tNumerical a_k\tAnalytical b_k\t\tNumerical b_k")
for k, ak_analytical, ak_numerical, bk_analytical, bk_numerical in
zip(k_values, ak_array_analytical, ak_array_numerical, bk_array_analytical,
bk_array_numerical):
    print(f"{k}\t{ak_analytical:.6f}\t\t{ak_numerical:.6f}\t{bk_analytical:.6f}\t\t{bk
    _numerical:.6f}")

    # Define the Fourier series reconstruction function
def reconstruct_fourier_series(t, ak, bk):
    omega = 1  # Assuming omega = 1 for simplicity
    n_terms = min(len(ak), len(bk))

    f_t = np.zeros_like(t)
    for k in range(n_terms):
        f_t += ak[k] * np.cos((k + 1) * omega * t) + bk[k] * np.sin((k + 1) * omega * t)

    return f_t

# Define the range of t values for reconstruction
t_values = np.linspace(0, 2 * np.pi, 1000)

# Reconstruct the original function using Fourier series

reconstructed_function = reconstruct_fourier_series(t_values,
ak_array_numerical, bk_array_numerical)
```

```
# Define the range of theta values for plotting
theta_values = np.linspace(0, 2 * np.pi, 1000)

# Compute the square wave function values
square_wave_values = square_wave(theta_values)

# Plot the square wave function
plt.plot(theta_values, square_wave_values, label='Square Wave')
plt.xlabel('')
plt.ylabel('f()')
plt.title('Square Wave Function')
plt.grid(True)
plt.legend()
plt.savefig('Lab 4 sqr.pdf')
plt.show()
# Plot the reconstructed function
plt.plot(t_values, reconstructed_function, label='Reconstructed Function')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Reconstructed Function using Fourier Series')
plt.grid(True)
plt.legend()
plt.savefig('Rec sqr.pdf')
plt.show()

# Define the Fourier series reconstruction function with limited terms
def reconstruct_fourier_series_limited_terms(t, ak, bk, num_terms):
    omega = 1  # Assuming omega = 1 for simplicity
    n_terms = min(len(ak), len(bk), num_terms)

    f_t = np.zeros_like(t)
    for k in range(n_terms):
        f_t += ak[k] * np.cos((k + 1) * omega * t) + bk[k] * np.sin((k + 1) *
        omega * t)

    return f_t

# Define the range of t values for reconstruction
t_values = np.linspace(0, 2 * np.pi, 1000)

# Define the number of terms for reconstruction
num_terms_list = [1, 2, 3, 5, 10, 20, 30]

# Plot the reconstructed square wave function for each number of terms
plt.figure(figsize=(12, 8))
```

```
for num_terms in num_terms_list:
    reconstructed_function =
    reconstruct_fourier_series_limited_terms(t_values, ak_array_numerical,
    bk_array_numerical, num_terms)
    plt.plot(t_values, reconstructed_function, label=f'{num_terms} Terms')

plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Reconstructed Square Wave Function with Limited Terms')
plt.grid(True)
plt.legend()
plt.show()


# Define the Fourier series reconstruction function with limited terms
def reconstruct_fourier_series_limited_terms(t, ak, bk, num_terms):
    omega = 1  # Assuming omega = 1 for simplicity
    n_terms = min(len(ak), len(bk), num_terms)

    f_t = np.zeros_like(t)
    for k in range(n_terms):
        f_t += ak[k] * np.cos((k + 1) * omega * t) + bk[k] * np.sin((k + 1) *
        omega * t)

    return f_t

# Define the range of t values for reconstruction
t_values = np.linspace(0, 2 * np.pi, 1000)

# Define the number of terms for reconstruction
num_terms_list = [1, 2, 3, 5, 10, 20, 30]

# Plot the reconstructed square wave function for each number of terms

plt.figure(figsize=(12, 8))
for num_terms in num_terms_list:
    reconstructed_function =
    reconstruct_fourier_series_limited_terms(t_values, ak_array_numerical,

    bk_array_numerical, num_terms)

    plt.plot(t_values, reconstructed_function, label=f'{num_terms} Terms')

# Plot the exact square wave function
exact_square_wave = square_wave(t_values)
```

```
plt.plot(t_values, exact_square_wave, linestyle='--', color='black',
label='Exact Square Wave')

plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Comparison of Partly Reconstructed Square Wave with Exact Square
Wave')
plt.grid(True)
plt.legend()
plt.savefig('Lab 4 Ex2.1.pdf')
plt.show()


For rectangular wave:
import numpy as np
import matplotlib.pyplot as plt

# Define the rectangular wave function
def rectangular_wave(omega, tao, theta):
    result = np.zeros_like(theta)  # Initialize result array

    # Define the rectangular wave function
    for i, value in enumerate(theta):
        norm_theta = value % (2 * np.pi)  # Normalize theta to [0, 2pi)
        if 0 <= norm_theta <= omega * tao:
            result[i] = 1
        elif omega * tao < norm_theta <= 2 * np.pi:
            result[i] = -1

    return result

# Define the exact Fourier coefficients for the rectangular wave
def exact_ak(k, alpha, tao):
    return (2 / (k * np.pi)) * np.sin(2 * k * np.pi * tao / alpha)

def exact_bk(k, alpha, tao):
    return (2 / (k * np.pi)) * (1 - np.cos(2 * k * np.pi *tao / alpha))

def exact_a0(alpha, tao):
    return 2 / alpha * (2*tao - alpha)

# Define the Fourier series reconstruction function with limited terms
def reconstruct_fourier_series_limited_terms(t, a0, ak, bk, num_terms):
    omega = 1  # Assuming omega = 1 for simplicity
    n_terms = min(len(ak), len(bk), num_terms)
    f_t = np.zeros_like(t)
    f_t += a0 / 2  # Add a0 / 2 for the constant term
    for k in range(n_terms):
        f_t += ak[k] * np.cos((k + 1) * omega * t) + bk[k] * np.sin((k + 1) * omega * t)

    return f_t

# Define tao
tao = 1.6

# Define the range of t values for reconstruction
t_values = np.linspace(0, 4 * np.pi, 1000)

# Define the number of terms for reconstruction
num_terms_list = [1, 2, 3, 5, 10, 20, 30]

# Compute the exact Fourier coefficients for the rectangular wave
alpha = 2 * np.pi  # Total period of the rectangular wave
exact_ak_array = np.array([exact_ak(k, alpha, tao) for k in range(1,
num_terms_list[-1] + 1)])
exact_bk_array = np.array([exact_bk(k, alpha, tao) for k in range(1,
num_terms_list[-1] + 1)])
exact_a0_value = exact_a0(alpha, tao)

# Plot the reconstructed rectangular wave function for each number of terms
plt.figure(figsize=(12, 8))

for i, num_terms in enumerate(num_terms_list):
    reconstructed_function =
    reconstruct_fourier_series_limited_terms(t_values, exact_a0_value,
    exact_ak_array[:num_terms], exact_bk_array[:num_terms], num_terms)
    plt.plot(t_values, reconstructed_function, label=f'{num_terms} Terms')

# Plot the exact rectangular wave function
exact_rectangular_wave = rectangular_wave(1, tao, t_values)  # Adjust omega
and tao as needed
plt.plot(t_values, exact_rectangular_wave, linestyle='--', color='black',
label='Exact Rectangular Wave')

plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Comparison of Partly Reconstructed Rectangular Wave with Exact
Rectangular Wave')
plt.grid(True)
plt.legend()
plt.savefig('Lab 4 Ex 2.2.pdf')
plt.show()

Exercise 3:
Part 1-2:
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 17 16:08:40 2024

@author: vwatanab
"""

import numpy as np
import matplotlib.pyplot as plt

# Compute the DFT
def discrete_fourier_transform(data):
    # Compute Fourier coefficients
    coefficients = []
    for n in range(N):
        Fn_real = 0
        Fn_imaginary = 0
        for m in range(N):
            Fn_real += data[m] * np.cos(2 * np.pi * m * n / N)
            Fn_imaginary -= data[m] * np.sin(2 * np.pi * m * n / N)
        coefficients.append((Fn_real, Fn_imaginary))

    return coefficients


# Parameters
N = 128
h = 0.1

# Calculations
tau = N * h
nu_s = tau / N  #Sampling rate
omega_1 = 2 * np.pi / (N * h)  # Fundamental frequency

# Generate time values
t_values = np.linspace(0, tau, num=N+1)     ##### DATA LIST FOR THE EXERCISE


# Calculate Fourier coefficients
fourier_comps = discrete_fourier_transform(t_values)

# Extract real and imaginary parts of Fourier coefficients
```

```
Fn_real_values = [Fn_real for Fn_real, _ in fourier_comps]
Fn_imaginary_values = [Fn_imaginary for _, Fn_imaginary in fourier_comps]


# Plot Fourier components Fn,real and Fn,imaginary as a function of n
plt.figure()
plt.plot(Fn_real_values, label='Fn,real')
plt.plot(Fn_imaginary_values, label='Fn,imaginary')
plt.xlabel('n')
plt.ylabel('Fourier Components')
plt.title('Fourier Components Fn,real and Fn,imaginary')
plt.legend()
plt.grid(True)
plt.savefig('Lab 4 3.1.pdf')
plt.show()


print("Sampling Rate (nu_s):", nu_s)
print("Fundamental Frequency (omega_1):", omega_1)
```

Part 3-4-5:

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 17 16:08:40 2024

@author: vwatanab
"""

import numpy as np
import matplotlib.pyplot as plt

# Define f(t) = sin(0.45*pi*t)
def f(t):
    return np.sin(0.45 * np.pi * t)


# Compute the Discrete Fourier Transform (DFT)
def discrete_fourier_transform(data):
    # Compute Fourier coefficients
    coefficients = []
    for n in range(N):
        Fn_real = 0
        Fn_imaginary = 0
        for m in range(N):
            Fn_real += data[m] * np.cos(2 * np.pi * m * n / N)
            Fn_imaginary -= data[m] * np.sin(2 * np.pi * m * n / N)
        coefficients.append((Fn_real, Fn_imaginary))
    return coefficients



# Define a function to reconstruct the original signal from Fourier
components
def reconstruct_signal(Fn_real_values, Fn_imaginary_values, t):
    signal_values = np.zeros_like(t)
    for n, (Fn_real, Fn_imaginary) in enumerate(zip(Fn_real_values,
    Fn_imaginary_values)):
        signal_values += (Fn_real * np.cos(2 * np.pi * n * t / tau) -
        Fn_imaginary * np.sin(2 * np.pi * n * t / tau))/N

    return signal_values


# Parameters
N = 128
h = 0.1

# Calculations
tau = N * h
nu_s = tau / N  # Sampling rate
omega_1 = 2 * np.pi / tau  # Fundamental frequency

# Generate time and values for the function f(t) = sin(0.45*pi*t)
t_values = np.linspace(0, tau, 10000)
f_values = f(t_values)

# Generate sampling time
sampled_t_values = np.arange(0, tau + h, h)
# Calculate function values
sampled_f_values = f(sampled_t_values)

# Calculate Fourier coefficients
fourier_comps = discrete_fourier_transform(sampled_f_values)

# Extract real and imaginary parts of Fourier coefficients
Fn_real_values = [Fn_real for Fn_real, _ in fourier_comps]
Fn_imaginary_values = [Fn_imaginary for _, Fn_imaginary in fourier_comps]

# Find the dominant component
dominant_component_index_real = np.argmax(np.abs(Fn_real_values))
dominant_component_index_imaginary = np.argmax(np.abs(Fn_imaginary_values))
```

```
# Print the indices of n corresponding to the dominant components
print("Dominant component index (real part):", dominant_component_index_real)
print("Dominant component index (imaginary part):",
dominant_component_index_imaginary)

# Plot the function f(t) along with the sampled points
plt.figure()
plt.plot(t_values, f_values, label='f(t)')
plt.scatter(sampled_t_values, sampled_f_values, color='red', label='Sampled
Points', zorder=5)
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Plot of f(t) with Sampled Points')
plt.legend()
plt.grid(True)
plt.savefig('Lab 4 3.2.pdf')
plt.show()

# Plot Fourier components Fn,real and Fn,imaginary as a function of n
plt.figure()
plt.plot(Fn_real_values, label='Fn,real')
plt.plot(Fn_imaginary_values, label='Fn,imaginary')
plt.scatter(dominant_component_index_real,
Fn_real_values[dominant_component_index_real], color='red', label='Dominant
Component (real)')
plt.scatter(dominant_component_index_imaginary,
Fn_imaginary_values[dominant_component_index_imaginary], color='green',
label='Dominant Component (imaginary)')

plt.xlabel('n')
plt.ylabel('Fourier Components')
plt.title('Fourier Components Fn,real and Fn,imaginary')
plt.legend()
plt.grid(True)
plt.savefig('Lab 4 3.3.pdf')
plt.show()

print("Sampling Rate (nu_s):", nu_s)
print("Fundamental Frequency (omega_1):", omega_1)

# Calculate the expected frequency corresponding to this value of n
# Using the relationship 0.45*pi*t = 2*pi*n*m/N and tm=m*h
# Solving for n
expected_n  = 0.45 * N * h /2
print("Expected value of n based on the function:", expected_n)
```

```
# Reconstruct the signal using the Fourier components
reconstructed_signal_values = reconstruct_signal(Fn_real_values,
Fn_imaginary_values, sampled_t_values)

# Plot the reconstructed signal
plt.figure()
plt.plot(sampled_t_values, reconstructed_signal_values, label='Reconstructed

Signal')
plt.plot(t_values, f_values, label='f(t)')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Reconstructed Signal from Fourier Components')
plt.legend()
plt.grid(True)
plt.savefig('lab 4 3.4.pdf')
plt.show()
```

3.6:

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 17 16:08:40 2024

@author: vwatanab
"""

import numpy as np
import matplotlib.pyplot as plt


# Define f(t) = sin(0.45*pi*t)
def f(t):
    return np.sin(0.45 * np.pi * t)

# Compute the Discrete Fourier Transform (DFT)
def discrete_fourier_transform(data, x_values):
    # Compute Fourier coefficients
    coefficients = []
    for n in range(N):
        Fn_real = 0
        Fn_imaginary =0
        for m in range(N):
            Fn_real += data[m] * np.cos(2 * np.pi * m * n / N)
            Fn_imaginary -= data[m] * np.sin(2 * np.pi * m * n / N)
        coefficients.append((Fn_real, Fn_imaginary))
    return coefficients

# Define a function to reconstruct the original signal from Fourier components
def reconstruct_signal(Fn_real_values, Fn_imaginary_values, t):
    signal_values = np.zeros_like(t)
    for i, ti in enumerate(t):
        signal_value = 0
        for n in range(len(Fn_real_values)):
            signal_value += (1/N) * (Fn_real_values[n] * np.cos(2 * np.pi * n
            * ti / tau) - Fn_imaginary_values[n] * np.sin(2 * np.pi * n * ti
            / tau))
            signal_values[i] = signal_value
    return signal_values

# Parameters
N = 128    #128
h = 2 / (0.45 * N)
print ("frequency = 0.45*pi  --> h = (2pi) / (freq*N))= 2/(0.45*128) = ", h)
# Calculations
tau = N * h

nu_s = tau / N  # Sampling rate
omega_1 = 2 * np.pi / tau  # Fundamental frequency

# Generate time and values for the function f(t) = sin(0.45*pi*t)
t_values = np.linspace(0, tau, 100000)     ##### DATA LIST FOR THE EXERCISE
f_values = f(t_values)

# Generate sampling time
sampled_t_values = np.arange(0, tau, h)
# Calculate function values
sampled_f_values = f(sampled_t_values)

# Calculate Fourier coefficients
fourier_comps = discrete_fourier_transform(sampled_f_values, sampled_t_values)

# Extract real and imaginary parts of Fourier coefficients
Fn_real_values = [Fn_real for Fn_real, _ in fourier_comps]
Fn_imaginary_values = [Fn_imaginary for _, Fn_imaginary in fourier_comps]

# Find the dominant component
dominant_component_index_real = np.argmax(np.abs(Fn_real_values))
dominant_component_index_imaginary = np.argmax(np.abs(Fn_imaginary_values))
```

```
# Plot the function f(t) along with the sampled points
plt.figure()
plt.plot(t_values, f_values, label='f(t)')
plt.scatter(sampled_t_values, sampled_f_values, color='red', label='Sampled
Points', zorder=5)
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Plot of f(t) with Sampled Points')
plt.legend()
plt.grid(True)
plt.savefig('Lab 4 Ex 3.601.pdf')
plt.show()

# Plot Fourier components Fn,real and Fn,imaginary as a function of n
plt.figure()
plt.plot(Fn_real_values, label='Fn,real')
plt.plot(Fn_imaginary_values, label='Fn,imaginary')
plt.scatter(dominant_component_index_real,
Fn_real_values[dominant_component_index_real], color='red', label='Dominant
Component (real)')
plt.scatter(dominant_component_index_imaginary,
Fn_imaginary_values[dominant_component_index_imaginary], color='green',
label='Dominant Component (imaginary)')

plt.xlabel('n')
plt.ylabel('Fourier Components')
plt.title('Fourier Components Fn,real and Fn,imaginary')
plt.legend()
plt.grid(True)
plt.savefig('Lab 4 Ex 3.602.pdf')
plt.show()


# Calculate the expected frequency corresponding to this value of n
# Using the relationship 0.45*pi*t = 2*pi*n*m/N and tm=m*h
# Solving for n
expected_n  = 0.45 * N * h /2
print("Expected value of n based on the function:", expected_n)

# Reconstruct the signal using the Fourier components
reconstructed_signal_values = reconstruct_signal(Fn_real_values,
Fn_imaginary_values, sampled_t_values)

# Plot the reconstructed signal
plt.figure()
plt.scatter(sampled_t_values, reconstructed_signal_values,
label='Reconstructed Signal', color='red')
plt.plot(t_values, f_values, label='f(t)')
plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('Reconstructed Signal from Fourier Components')
plt.legend()
plt.grid(True)
plt.savefig('Lab 4 Ex 3.603.pdf')
plt.show()
```