Fourier Analysis for Signal Processing

Luke Keely

Trinity College Dublin,
keelyl@tcd.ie

Abstract. In this lab exercise, we look at applying Fourier analysis for signal processing. Using Simpson's Rule for numerical integration, we investigate the Fourier series analysis by deconstructing and reconstructing periodic and non-periodic signals as well as non-sinusoidal waves. The Discrete Fourier Transform (DFT) is also examined, where we confirm the significance of appropriate sampling rates, as dictated by the Nyquist-Shannon theorem, to capture signal frequency content accurately.
Overall, this study acts as a basis for Fourier analysis in practical signal processing applications, showcasing numerical methods and confirming theoretical knowledge.

Table of Contents

1 Introduction

Fourier analysis is a core idea in signal processing. It breaks down complex signals by decomposing them into fundamental components of sine waves and cosine waves.
This lab exercise focuses on applying Fourier analysis to periodic and non-periodic signals. We work with Fourier series, Fourier transforms, and discrete Fourier transforms (DFT), using the numerical method Simpson's rule for integration.

Fourier series are used to represent periodic signals. Mathematically, a Fourier series for a function $f(t)$ with period $T = \frac{2\pi}{\omega}$ is expressed as

$$f(t) = a_0 + \sum_{n=1}^{\infty}(a_n \cos(n\omega t) + b_n \sin(n\omega t)),$$

where $\omega_n = n\omega$ are the discrete frequencies, with $\omega_1 = \omega$ being the fundamental frequency. The coefficients a_n and b_n are the magnitudes of the cosine and sine components[2].

The Fourier transform develops into Fourier analysis for non-periodic signals. It transforms a time-domain signal into a frequency-domain representation. The Fourier transform of a function $f(t)$ is defined as

$$F(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-i\omega t}dt,$$

where $F(\omega)$ is the frequency spectrum of $f(t)$[3].

The Discrete Fourier Transform (DFT) is needed to analyze discrete, finite-length signals. The DFT is defined as

$$F_k = \sum_{n=0}^{N-1} f_n e^{-i2\pi kn/N},$$

where, F_k are the DFT coefficients, f_n represents the sampled values of the signal, and N is the number of samples. This formula enables the transformation of signals from the time domain to the frequency domain and is a core part of digital signal analysis[4].

Simpson's rule is used to compute the integrals needed for our Fourier coefficients. It is a numerical integration method that approximates the integral of a function by dividing it into sections and using parabolic arcs to approximate the area under the curve. This method is similar to the trapezoidal rule for integration but improves upon it by using quadratic functions instead of straight lines to connect the segments[5].

The uses of Fourier analysis go far beyond a computational exercise. It has many real-world applications in fields such as signal processing, digital image processing, statistics, and cryptography. This broad applicability comes from the many useful properties of the transforms[6].

2 Methodology

2.1 Simpson's Rule for Integration

In the first part of the experiment, we look at numerical methods for integration. This will be used in the following parts of our analysis. The method chosen for this lab was Simpson's Rule because of its efficiency and accuracy for approximating definite integrals.

Simpson's Rule was implemented by dividing the integration interval into an even number of segments and then applying a weighted sum of the function's values at different points across these segments. By combining these values, we approximate the area under the curve. The script calculates this using the formula

$$\text{Integral} \approx \frac{h}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + \cdots + 4f(x_{n-1}) + f(x_n)],$$

where $h = \frac{b-a}{n}$ is the width of each segment.

```
def simpsons_rule(a, b, n):
    h = (b - a) / n
    integral = f(a) + f(b)
    for i in range(1, n):
        k = a + i * h
        if i % 2 == 0:
            integral += 2 * f(k)
        else:
            integral += 4 * f(k)
    integral = integral * h / 3
    return integral
```

To test the code, we compute the integral of the exponential function $f(x) = e^x$ across the interval from 0 to 1 using 12 steps.

In order to check the accuracy of the Simpson's Rule integration, we plotted the approximation against the actual function $f(x) = e^x$.

```
def plot(a, b, n):
    sns.set_style("darkgrid", {"axes.facecolor": ".9"})
    sns.set_palette("magma")
    plt.figure(figsize=(6, 4), dpi=300)
    plt.plot(x_values, y_values, label=function_label)
    plt.plot(approx_x, approx_y, linestyle='—', label="Simpson Rule")
    plt.legend(loc='upper right')
    plt.savefig("integration_plot.png", format='png')
    plt.show()
```

This plotting style was used for the rest of the scripts going forward.

2.2 Fourier Series Analysis

In the second part of the experiment, we move to Fourier series analysis. This technique is used to break down periodic signals into a series of sine and cosine functions, allowing us to look at the frequency components of the signal. This is particularly useful for decomposing complicated periodic waveforms. We used Simpson's Rule to calculate the Fourier coefficients for the analysis. These coefficients are required to reconstruct the original function from our sine and cosine components.

The Fourier coefficients, a_n and b_n, are defined by integrals that measure the signal's correlation to the sine and cosine functions of different frequencies. These integrals are given by

$$a_n = \frac{2}{T}\int_0^T f(t)\cos(n\omega t)dt,$$

$$\text{and } b_n = \frac{2}{T}\int_0^T f(t)\sin(n\omega t)dt,$$

where T is the period of the function, and ω is the angular frequency.

```
def fourier_coefficients(f, T, n, k_max):
    omega = 2 * np.pi / T
    a0 = simpsons_rule(f, 0, T, n) / T
    ak = np.zeros(k_max)
    bk = np.zeros(k_max)
    for k in range(1, k_max + 1):
        ak_func = lambda t: f(t) * np.cos(k * omega * t)
        bk_func = lambda t: f(t) * np.sin(k * omega * t)
        ak[k-1] = 2 * simpsons_rule(ak_func, 0, T, n) / T
        bk[k-1] = 2 * simpsons_rule(bk_func, 0, T, n) / T
    return a0, ak, bk
```

The script then reconstructs the original signal from the calculated coefficients. By summing up the individual sine and cosine waves and taking their weights into account, we can see how signals may be reconstructed, which allows us to check the accuracy of the analysis.

```
def fourier_series(t, a0, ak, bk, T):
    omega = 2 * np.pi / T
    series = a0 / 2
    for k in range(1, len(ak) + 1):
        series += ak[k-1] * np.cos(k * omega * t) + bk[k-1] * np.sin(k
            * omega * t)
    return series
```

In order to test the script, it was applied to various functions with increasing complexity.

Test Case	Chosen Function
Simple sine wave	$\sin(\omega t)$
Combination of cosine waves	$\cos(\omega t) + 3\cos(2\omega t) - 4\cos(3\omega t)$
Combination of sine waves	$\sin(\omega t) + 3\sin(3\omega t) + 5\sin(5\omega t)$
Complex waveform	$\sin(\omega t) + 2\cos(3\omega t) + 3\sin(5\omega t)$

2.3 Rectangular Wave Fourier Analysis

In the third part of the experiment, we expand to the Fourier series analysis of a rectangular wave, similar to digital signals. Moving from traditional to rectangular waveforms allows us to transition from analog to analog and digital processing techniques.

The rectangular wave function was represented by alternating between 1 and -1. This was implemented using an if returning either a 1 or a -1 based on the position in the wave's period.

The script then reconstructs the original signal from the calculated coefficients. By summing up the individual sine and cosine waves and taking their weights into account, we can see how signals may be reconstructed, which allows us to check the accuracy of the analysis.

```
def rectangular_wave(t, w=1, alpha=2):
    tau = 2 * np.pi / (alpha * w)
    theta = (w * t) % (2 * np.pi)
    return 1 if 0 <= theta < tau else -1
```

Introducing a piecewise constant function made computing its Fourier series more complex. Simpson's Rule was again used to calculate the Fourier coefficients for the rectangular wave.

2.4 DFT Analysis and Sampling

In the final part of the experiment, we look at the Discrete Fourier Transform (DFT). It is a fundamental technique in analyzing discrete, finite-length signals. DFT is especially important for digital signal processing, transforming signals between the time and frequency domains, and can be applied to a wide range of applications such as signal compression, noise reduction, and data analysis.

```
def discrete_fourier_transform(signal):
    F = np.fft.fft(signal)
    return F.real, F.imag
```

The DFT analysis used a Fast Fourier Transform (FFT) to convert time-domain signals into frequency-domain signals. The real and imaginary parts of the transformed signal were extracted in order to analyze the frequency content.

2.5 Challenges and Modifications

Initially, I had little understanding of Fourier analysis and Fourier transform.
To solve this, I spent some time reading online resources in order to implement them in Python and explain their uses.

Another challenge was representing and analyzing the data from the Fourier transformations and numerical integrations clearly on a small plot.
To solve this, I used many graphs when iterating through step sizes and reduced the information on the plot to only what was necessary.

3 Results

3.1 Simpson's Rule for Integration

Using Simpson's Rule for integration returned excellent results. This numerical method worked well over many functions of varying complexity and for our test case return a very small error.

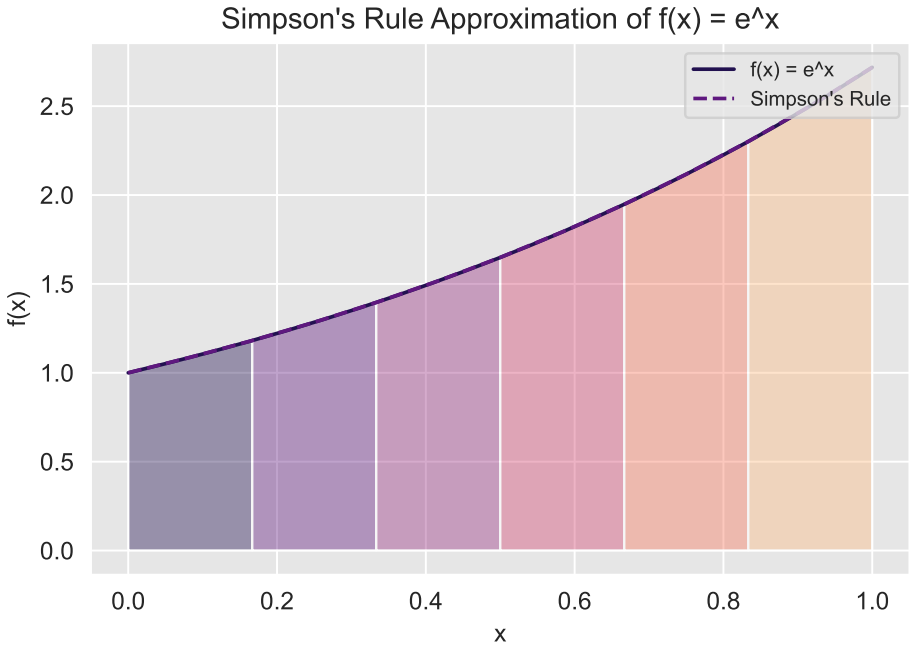


Fig. 3.1: Plot of function $f(x) = e^x$ and Simpson's Rule integration with steps

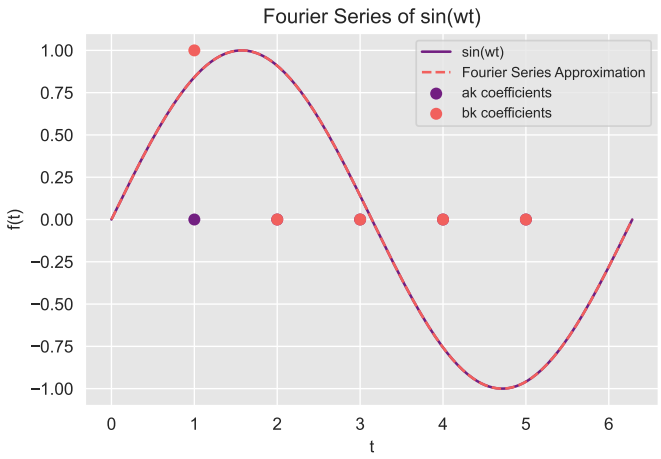
Comparing the numerical result from Simpson's Rule to the exact analytical result for the integral of $f(x) = e^x$ over the interval $[0, 1]$, using a step size of 12, we see excellent accuracy. The analytical calculation results in $e - 1$, while the numerical integration results in 1.7182822884380207.

Function	Interval	Analytical Result	Numerical Result (Simpson's Rule)	Error
e^x	$[0, 1]$	$e - 1$	1.7182822884380207	$\approx -4.6 \times 10^{-7}$

This negligible error confirms that Simpson's Rule approximations are suitable for this experiment.

3.2 Fourier Series Analysis

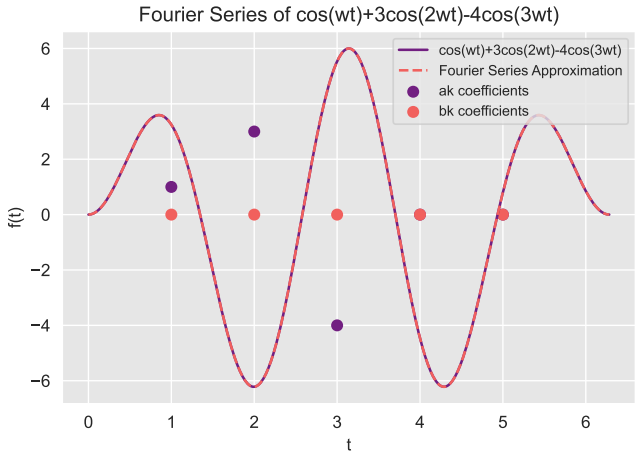
The Fourier series analysis and reconstruction performed well over every test case.



Simple Sine Wave $\sin(\omega t)$

The Fourier series matches the waveform exactly, with the Fourier coefficients matching the expected values, so our plotted lines overlap completely.

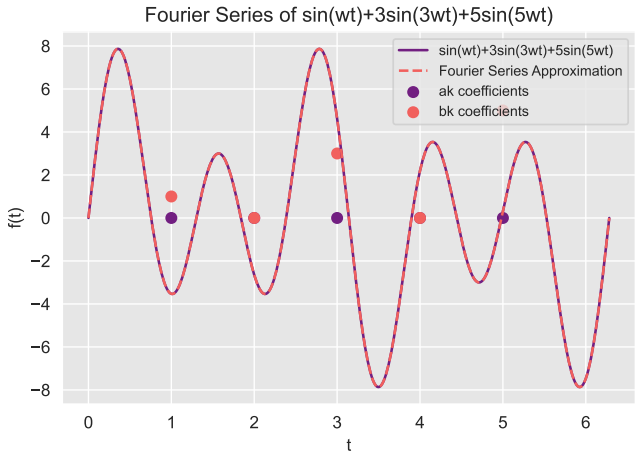
Fig. 3.2: Plot of Fourier Analysis Reconstruction for $\sin(\omega t)$ over Original Function with Coefficients.



Combination of Cosine Waves $\cos(\omega t) + 3\cos(2\omega t)$

The Fourier series accurately captures the combination waveform, even with the added complexity of many cosine waves added together.

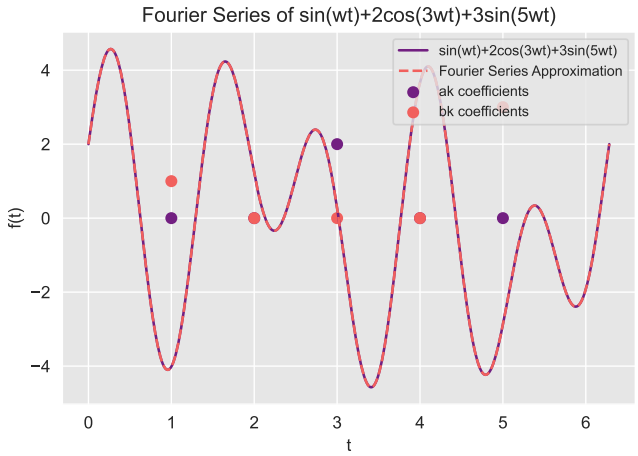
Fig. 3.3: Plot of Fourier Analysis Reconstruction for $\cos(\omega t) + 3\cos(2\omega t) - 4\cos(3\omega t)$ over Original Function with Coefficients.



Combination of Sine Waves $\sin(\omega t) + 3\sin(3\omega t) + 5\sin(5\omega t)$

The Fourier series analysis again decomposed the function into its basic sinusoidal components successfully, leading to an excellent waveform approximation.

Fig. 3.4: Plot of Fourier Analysis Reconstruction for $\sin(\omega t) + 3\sin(3\omega t) + 5\sin(5\omega t)$ over Original Function with Coefficients.



Complex Waveform $\sin(\omega t) + 2\cos(3\omega t) + 3\sin(5\omega t)$

This more complex waveform showed the Fourier series method's ability to deal with complicated functions, again with the analysis providing an accurate reconstruction of the original signal.

Fig. 3.5: Plot of Fourier Analysis Reconstruction for $\sin(\omega t) + 2\cos(3\omega t) + 3\sin(5\omega t)$ over Original Function with Coefficients.

3.3 Rectangular Wave Fourier Analysis

The analysis of the rectangular wave using the Fourier series showed the method's ability to decompose the characteristics of non-sinusoidal functions.

1 Term Reconstruction Using only 1 term in the Fourier series resulted in a sinusoidal approximation that captures the fundamental frequency of the rectangular wave, but no other information is given. The waveform is smooth and has no sharp corners, the characteristic of a rectangular wave.

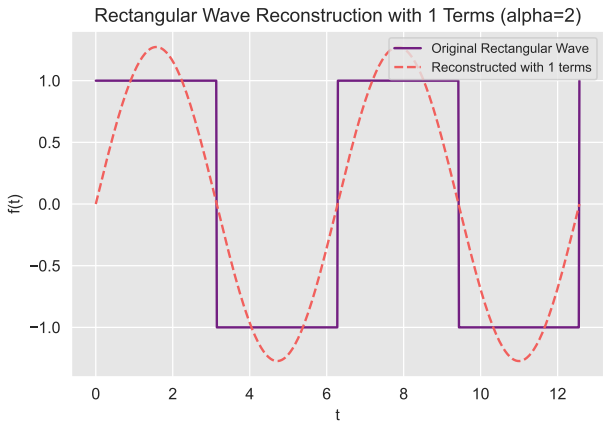


Fig. 3.6: Plot of Rectangular Wave with $\alpha = 2$ and initial 1 Term Approximation

5 Term Reconstruction Introducing 5 terms, we start to see the sharpness characteristic of a rectangular wave. The approximation begins to show the square waves more clearly. However, the edges are still round, and it is not a great wave reconstruction.

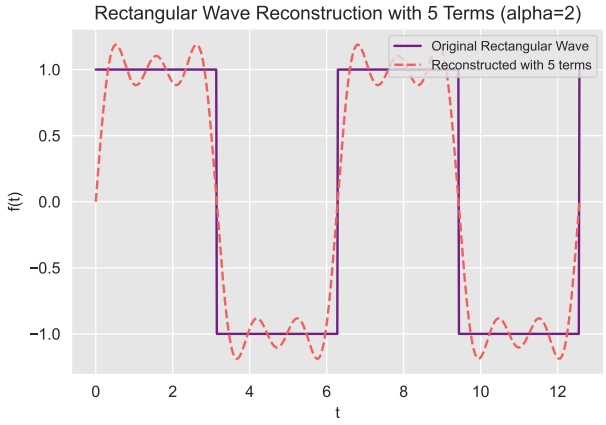


Fig. 3.7: Plot of Rectangular Wave with $\alpha = 2$ and acceptable fit 5 Term Approximation

30 Term Reconstruction Reaching 30 terms, the Fourier series closely reconstructs the rectangular wave. The approximation shows clear highs and lows separated by steep steps, similar to the original waveform. The plot with 30 terms shows how increasing the number of terms in a Fourier series improves the approximation even in discontinuous functions, which is essential for analyzing digital signals.

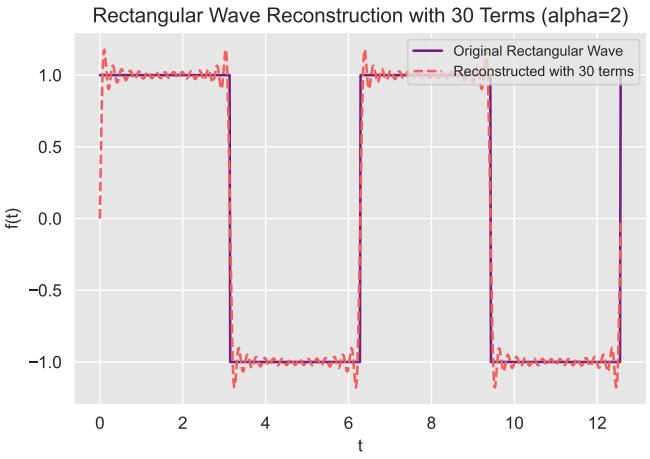
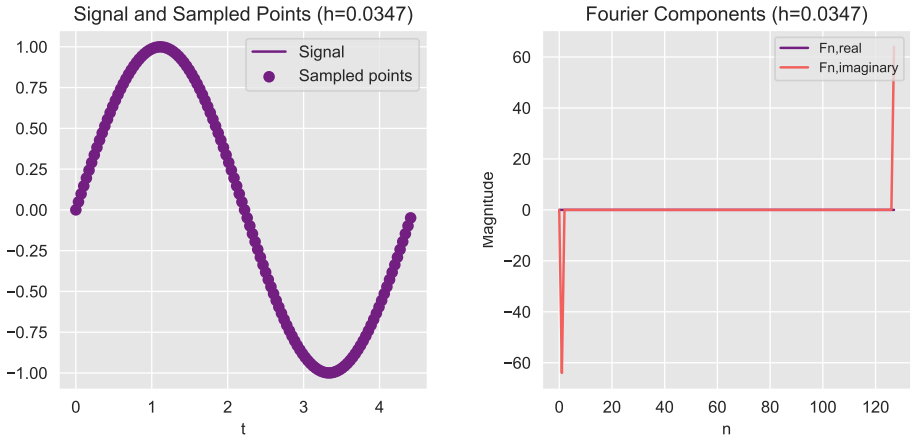


Fig. 3.8: Plot of Rectangular Wave with $\alpha = 2$ and well fit 30 Term Approximation

3.4 DFT Analysis and Sampling

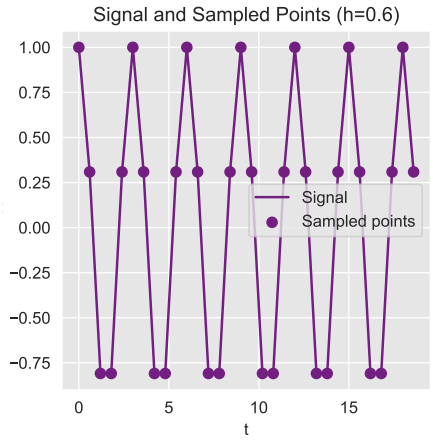


(a) Sine Signal with Sampled Points

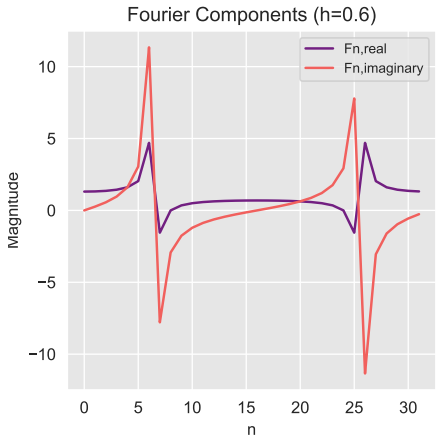
(b) Fourier Components

Fig. 3.9: DFT on Sine Signal with ideal sample rate h=0.0347

When sampled at an ideal rate, the sine signal shows a clear DFT spectrum with sharp peaks characteristic of an excellent scan.

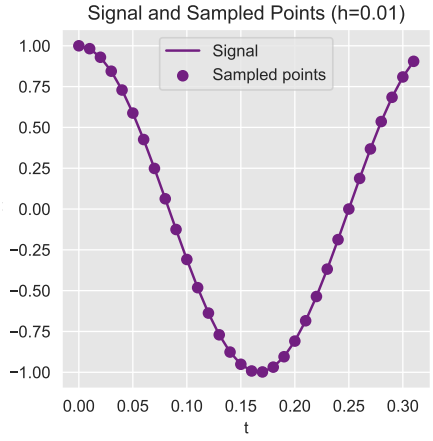


(a) Cosine Signal with Sampled Points

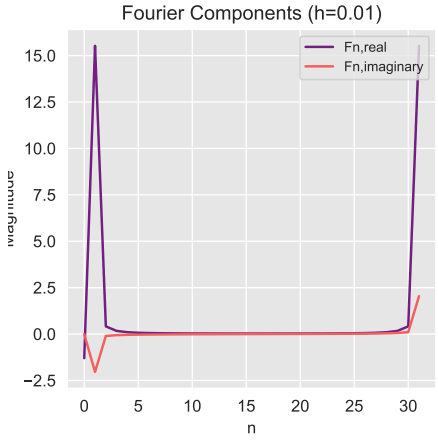


(b) Fourier Components

Fig. 3.10: DFT on Cosine Signal showing aliasing h=0.6



(a) Cosine Signal with Sampled Points



(b) Fourier Components

Fig. 3.11: DFT on Cosine Signal with better sample rate h=0.01

The cosine signal was sampled at two rates. The higher sampling rate (h=0.01) captures the cosine wave well.
However, the lower sampling rate (h=0.6) shows signs of aliasing due to the sampling rate being too low. This obscures the frequency, resulting in sharp points.

The difference in the DFT results from the cosine wave shows how important the sampling rate is in signal analysis. To avoid artifacts such as aliasing in the scans, it is important to sample at a rate at least twice that of the maximum frequency present in the signal, according to the Nyquist-Shannon sampling theorem.

4 Conclusion

In the lab exercise, we looked at Fourier Analysis for Signal Processing. We focused on Simpson's Rule for Integration, Fourier Series Analysis, Rectangular Wave Fourier Analysis, and DFT Sampling.

The Fourier analysis was applied to periodic and non-periodic signals, showing how sine and cosine waves can be found in complex signals and reconstructed from their Fourier coefficients. Using Simpson's Rule, we were able to calculate Fourier coefficients and reconstruct the signals with a high degree of accuracy.

The Fourier analysis was further applied to non-sinusoidal functions. While these were harder to reconstruct, increasing the terms to 30 allowed for an accurate reconstruction of the signal that could be used to understand and transform the original rectangular wave.

The investigation of the Discreet Fourier Transform showed us that the choice of sampling rate is essential for accurate and reliable scans. Ideal sampling rates result in sharp, clear peaks on the DFT spectrum, while poor rates lead to aliasing and distortion of the true signal. This is in line with the Nyquist-Shannon sampling theorem, which states that in order to capture the true nature of a signal, the sampling rate should be at least twice the highest frequency present in the signal.

In conclusion, this lab exercise not only looks at the theoretical background of Fourier analysis but highlights the need for fundamental theorems when applying these techniques.

A Supplemental Material

References

1. Department of Physics, Trinity College Dublin, *SF Computational Lab Handbook*.
2. Lamar University, *Introduction to Fourier Series*.
3. Wolfram MathWorld, *Fourier Transform*.
4. Wolfram MathWorld, *Discrete Fourier Transform*.
5. Wolfram MathWorld, *Simpson's Rule*.
6. Wikipedia, *Fourier Analysis*.
7. Documentation for NumPy, Matplotlib, and Seaborn.
8. Python Software Foundation, *Python Documentation*.

B Code

B.1 simpsons_rule.py

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os

# Define integral and accuracy
a = 0  # Lower limit of integration
b = 1  # Upper limit of integration
n = 12  # Number of steps (must be even)
function_label = "f(x) = e^x"
def f(x):
    return np.exp(x)

def simpsons_rule(a, b, n):
    h = (b - a) / n
    integral = f(a) + f(b)

    for i in range(1, n):
        k = a + i * h
        if i % 2 == 0:
            integral += 2 * f(k)
        else:
            integral += 4 * f(k)

    integral = integral * h / 3
    return integral

def plot(a, b, n):
    x_values = np.linspace(a, b, 1000)
    y_values = f(x_values)
    approximation_x = np.linspace(a, b, n + 1)
    approximation_y = f(approximation_x)
    sns.set_style("darkgrid", {"axes.facecolor": ".9"})
    sns.set_palette("magma")
    plt.figure(figsize=(6, 4), dpi=300)
    plt.plot(x_values, y_values, label=function_label)
    plt.plot(approximation_x, approximation_y, linestyle='--', label="
        Simpson's Rule")

    for i in range(1, len(approximation_x) - 1, 2):
        plt.fill_between(approximation_x[i-1:i+2], approximation_y[i-1:
            i+2], alpha=0.4)

    plt.title(f'Simpson\'s Rule Approximation of {function_label}')
    plt.xlabel("x")
    plt.ylabel("f(x)")
    plt.legend(loc='upper right', fontsize='small')
    folder_name = "Simpsons_Rule_Integration"
    os.makedirs(folder_name, exist_ok=True)
    plt.savefig(f"{folder_name}/integration_plot.png", format='png')
    plt.savefig(f"{folder_name}/integration_plot.pdf", format='pdf')
    plt.show()

```

```
result = simpsons_rule(a, b, n)
print("The integral result is:", result)
plot(a, b, n)
```

B.2 fourier_simpsons.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import os

# Constants
T = 2 * np.pi  # Period
n = 100  # Steps
k_max = 5  # Number of Fourier coefficients

# Cases
functions = {
    "sin(wt)": lambda t, w=1: np.sin(w * t),
    "cos(wt)+3cos(2wt)-4cos(3wt)": lambda t, w=1: np.cos(w * t) + 3 *
        np.cos(2 * w * t) - 4 * np.cos(3 * w * t),
    "sin(wt)+3sin(3wt)+5sin(5wt)": lambda t, w=1: np.sin(w * t) + 3 *
        np.sin(3 * w * t) + 5 * np.sin(5 * w * t),
    "sin(wt)+2cos(3wt)+3sin(5wt)": lambda t, w=1: np.sin(w * t) + 2 *
        np.cos(3 * w * t) + 3 * np.sin(5 * w * t)
}

sns.set_style("darkgrid", {"axes.facecolor": ".9"})
sns.set_palette("magma", n_colors=2)

def simpsons_rule(func, a, b, n):
    h = (b - a) / n
    integral = func(a) + func(b)
    for i in range(1, n):
        k = a + i*h
        if i % 2 == 0:
            integral += 2 * func(k)
        else:
            integral += 4 * func(k)
    return integral * h / 3

def fourier_coefficients(f, T, n, k_max):
    omega = 2 * np.pi / T
    a0 = simpsons_rule(f, 0, T, n) / T
    ak = np.zeros(k_max)
    bk = np.zeros(k_max)
    for k in range(1, k_max + 1):
        ak_func = lambda t: f(t) * np.cos(k * omega * t)
        bk_func = lambda t: f(t) * np.sin(k * omega * t)
        ak[k-1] = 2 * simpsons_rule(ak_func, 0, T, n) / T
        bk[k-1] = 2 * simpsons_rule(bk_func, 0, T, n) / T
    return a0, ak, bk

def fourier_series(t, a0, ak, bk, T):
    omega = 2 * np.pi / T
```

```
    series = a0 / 2
    for k in range(1, len(ak) + 1):
        series += ak[k-1] * np.cos(k * omega * t) + bk[k-1] * np.sin(k
            * omega * t)
    return series


for function_name, function in functions.items():
    folder_name = f"fourier_simpsons_{function_name}"
    os.makedirs(folder_name, exist_ok=True)
    a0, ak, bk = fourier_coefficients(function, T, n, k_max)

    t_values = np.linspace(0, T, 400)
    f_values = [function(t) for t in t_values]
    fourier_values = [fourier_series(t, a0, ak, bk, T) for t in
        t_values]
    df = pd.DataFrame({'t': t_values, 'f(t)': f_values, 'Fourier
        Approximation': fourier_values})
    df.to_csv(f"{folder_name}/fourier_data_{function_name}.csv", index=
        False)

    plt.figure(figsize=(6, 4), dpi=300)
    plt.plot(t_values, f_values, label=f"{function_name}")
    plt.plot(t_values, fourier_values, linestyle='--', label="Fourier
        Series Approximation")
    plt.scatter(np.arange(1, k_max + 1), ak, label='ak coefficients')
    plt.scatter(np.arange(1, k_max + 1), bk, label='bk coefficients')
    plt.title(f"Fourier Series of {function_name}")
    plt.xlabel("t")
    plt.ylabel("f(t)")
    plt.legend(loc='upper right', fontsize='small')
    plt.savefig(f"{folder_name}/fourier_plot_{function_name}.png",
        format='png')
    plt.savefig(f"{folder_name}/fourier_plot_{function_name}.pdf",
        format='pdf')
    plt.show()
```

B.3 fourier_rectwave.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import os

# Constants
T = 2 * np.pi
n = 100
k_max = 30
alpha = 2

def rectangular_wave(t, w=1, alpha=2):
    tau = 2 * np.pi / (alpha * w)
    theta = (w * t) % (2 * np.pi)
    return 1 if 0 <= theta < tau else -1

```

```
def simpsons_rule(func, a, b, n):
    h = (b - a) / n
    integral = func(a) + func(b)
    for i in range(1, n):
        k = a + i * h
        integral += 2 * func(k) if i % 2 == 0 else 4 * func(k)
    return integral * h / 3

def fourier_coefficients(f, T, n, k_max, alpha):
    a0 = (2 / T) * simpsons_rule(lambda t: f(t, alpha=alpha), 0, T, n)
    ak = [(2 / (k * np.pi)) * np.sin(2 * k * np.pi / alpha) for k in
        range(1, k_max + 1)]
    bk = [(2 / (k * np.pi)) * (1 - np.cos(2 * k * np.pi / alpha)) for k
         in range(1, k_max + 1)]
    return a0, np.array(ak), np.array(bk)

def reconstruct_fourier_series(t, a0, ak, bk, w, k_max):
    series = a0 / 2
    for k in range(1, k_max + 1):
        series += ak[k-1] * np.cos(k * w * t) + bk[k-1] * np.sin(k * w
            * t)
    return series

a0, ak, bk = fourier_coefficients(rectangular_wave, T, n, k_max, alpha)

t_values = np.linspace(0, 2 * T, 800)
f_values = [rectangular_wave(t, w=1, alpha=alpha) for t in t_values]

folder_name = "fourier_rectangular_wave_analysis"
os.makedirs(folder_name, exist_ok=True)

term_numbers = [1, 2, 3, 5, 10, 20, 30]
sns.set_style("darkgrid", {"axes.facecolor": ".9"})
sns.set_palette("magma", n_colors=2)

for num_terms in term_numbers:
    reconstructed_values = [reconstruct_fourier_series(t, a0, ak, bk,
        1, num_terms) for t in t_values]

    plt.figure(figsize=(6, 4), dpi=300)
    plt.plot(t_values, f_values, label="Original Rectangular Wave")
    plt.plot(t_values, reconstructed_values, linestyle='--', label=f"
        Reconstructed with {num_terms} terms")
    plt.title(f"Rectangular Wave Reconstruction with {num_terms} Terms
        (alpha={alpha})")
    plt.xlabel("t")
    plt.ylabel("f(t)")
    plt.legend(loc='upper right', fontsize='small')
    plt.savefig(f"{folder_name}/rectangular_wave_reconstruction_{
        num_terms}_terms.png", format='png')
    plt.savefig(f"{folder_name}/rectangular_wave_reconstruction_{
        num_terms}_terms.pdf", format='pdf')
    plt.show()

coefficients_df = pd.DataFrame({
    'k': np.arange(1, k_max + 1),
    'ak': ak,
```

```
    'bk': bk
})
coefficients_df.to_csv(f"{folder_name}/fourier_coefficients.csv", index=False)
```

B.4 discrete_fourier_transform.py

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import os

# Constants
N = 128
h = 0.1
T = N * h

def generate_signal(func, T, N):
    t_values = np.linspace(0, T, N, endpoint=False)
    return t_values, [func(t) for t in t_values]

def discrete_fourier_transform(signal):
    F = np.fft.fft(signal)
    return F.real, F.imag

def save_data_and_plot(t_values, f_values, F_real, F_imag, h, title_suffix):
    folder_name = f"DFT_analysis_{title_suffix.strip('()').replace('=', '_')}"
    os.makedirs(folder_name, exist_ok=True)

    # Save data
    df = pd.DataFrame({'t': t_values, 'f(t)': f_values, 'F_real': F_real, 'F_imag': F_imag})
    print(title_suffix, df)
    df.to_csv(f"{folder_name}/{title_suffix}_data.csv", index=False)

    # Plot
    plt.figure(figsize=(6, 4), dpi=300)
    plt.plot(t_values, f_values, label='Signal')
    plt.scatter(t_values, f_values, label='Sampled points')
    plt.title(f'Signal and Sampled Points {title_suffix}')
    plt.xlabel('t')
    plt.ylabel('f(t)')
    plt.legend()
    plt.savefig(f"{folder_name}/signal.png")
    plt.show()
    plt.figure(figsize=(6, 4), dpi=300)
    plt.plot(F_real, label='Fn,real')
    plt.plot(F_imag, label='Fn,imaginary')
    plt.title(f'Fourier Components {title_suffix}')
    plt.xlabel('n')
    plt.ylabel('Magnitude')
    plt.legend(loc='upper right', fontsize='small')
    plt.savefig(f"{folder_name}/{title_suffix}fourier_components.png")
```

```
    plt.savefig(f"{folder_name}/{title_suffix}fourier_components.pdf")
    plt.show()

sns.set_style("darkgrid", {"axes.facecolor": ".9"})
sns.set_palette("magma", n_colors=2)

# Signal generation and DFT calculation for sin(0.45 t )
sin_func = lambda t: np.sin(0.45 * np.pi * t)
t_values, f_values = generate_signal(sin_func, T, N)
F_real, F_imag = discrete_fourier_transform(f_values)
save_data_and_plot(t_values, f_values, F_real, F_imag, h, "(h=0.1)")

# Ideal sampling interval calculation and analysis for omega_1 = 0.45
h_ideal = 2 * np.pi / (0.45 * np.pi * N)
t_values_ideal, f_values_ideal = generate_signal(sin_func, N * h_ideal,
    N)
F_real_ideal, F_imag_ideal = discrete_fourier_transform(f_values_ideal)
save_data_and_plot(t_values_ideal, f_values_ideal, F_real_ideal,
   F_imag_ideal, h_ideal, "(Ideal h)")

# Additional cases with f(t) = cos(6 t ), varying h
cos_func = lambda t: np.cos(6 * np.pi * t)
hs = [0.6, 0.2, 0.1, 0.04]
for h in hs:
    t_values, f_values = generate_signal(cos_func, N * h, N)
    F_real, F_imag = discrete_fourier_transform(f_values)
    save_data_and_plot(t_values, f_values, F_real, F_imag, h, f"(h={h})
        ")
```