# EE 242 Lab 3a – Frequency Domain Representation of Signals - Fourier Series

Mason Wheeler

This lab has 2 exercises to be completed as a team. Each should be given a separate code cell in your Notebook, followed by a markdown cell with report discussion. Your notebook should start with a markdown title and overview cell, which should be followed by an import cell that has the import statements for all assignments. For this assignment, you will need to import: numpy, the wavfile package from scipy.io, simpleaudio/librosa, and matplotlib.pyplot.

In [7]:
```
# We'll refer to this as the "import cell." Every module you import should be i
%matplotlib notebook
import numpy as np
import matplotlib
import scipy.signal as sig
import matplotlib.pyplot as plt
import simpleaudio as sa
# import whatever other modules you use in this lab -- there are more that you
```

## Summary

In this lab, we will learn how to build periodic signals from component sinusoids and how to transform signals from the time domain to the frequency domain. The concepts we'll focus on include: implementation of the Fourier Series synthesis equation, using a discrete implementation of the Fourier Transform (DFT) with a digitized signal, and understanding the relationship between the discrete DFT index k and frequency ω for both the original continuous signal x(t). This is a two-week lab. You should plan on completing the first 2 assignments in the first week.

## Lab 3a turn in checklist

• Lab 3a Jupyter notebook with code for the 2 exercises assignment in separate cells. Each assignment cell should contain markdown cells (same as lab overview cells) for the responses to lab report questions. Include your lab members' names at the top of the notebook.

**Please submit the report as PDF** (You may also use : https://www.vertopal.com/ suggested by a student)

## Assignment 1 -- Generating simple periodic signals

In the first assignment, you will develop an understanding of how some periodic signals are easier to approximate than others with a truncated Fourier Series. In this lab, we'll work with

real signals and use the synthesis equation:

$$x(t) = a_0 + \sum_{k=1}^{N} 2|a_k|cos(k\omega_0 t + \angle a_k)$$

In lecture, you saw that you get ripples at transition points in approximating a square wave (Gibbs phenomenon). This happens for any signals with sharp edges. This assignment will involve approximating two signals (a sawtooth and a triangle wave) that have the same fundamental frequency (20Hz).

**A.** Write a function for generating a real-valued periodic time signal given the Fourier series coefficients $[a_0\ a_1\ \cdots\ a_N]$, the sampling frequency, and the fundamental frequency. You may choose to have complex input coefficients or have separate magnitude and phase vectors for describing $a_k$.

**B.** Define variables for the sampling frequency (8kHz) and the fundamental frequency (20Hz). Using this sampling frequency, create a time vector for a length of 200ms.

**C.** The sawtooth signal has coefficients as follows:

$$a_0 = 0.5, a_k = 1/(j2k\pi)$$

Using the function from part A, create three approximations of this signal with N = 2,5,20 and plot together in a 3×1 comparison.

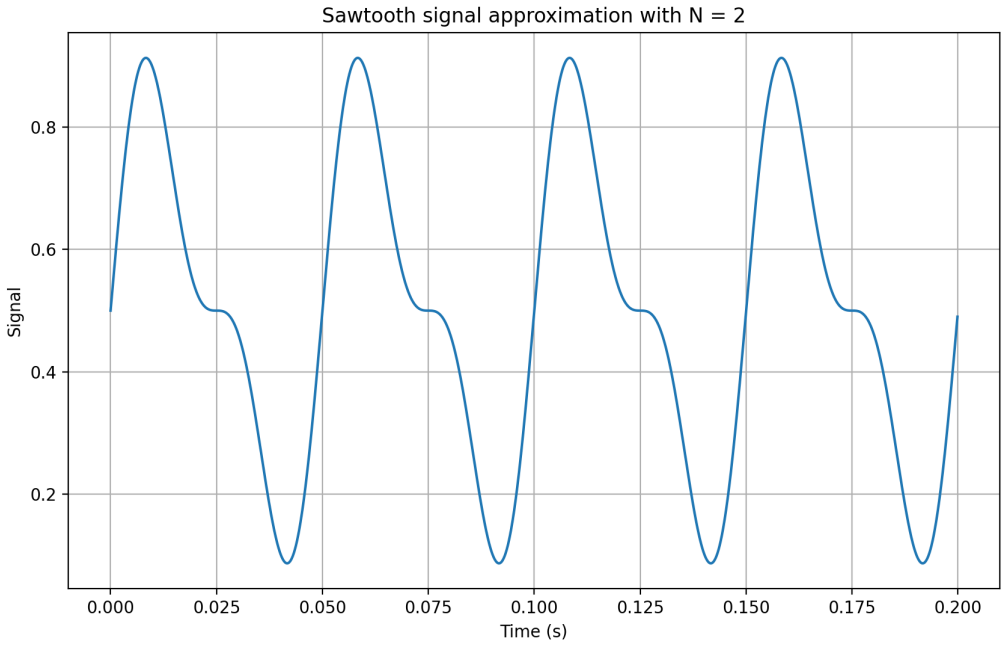
**D.** A triangle signal has coefficients:

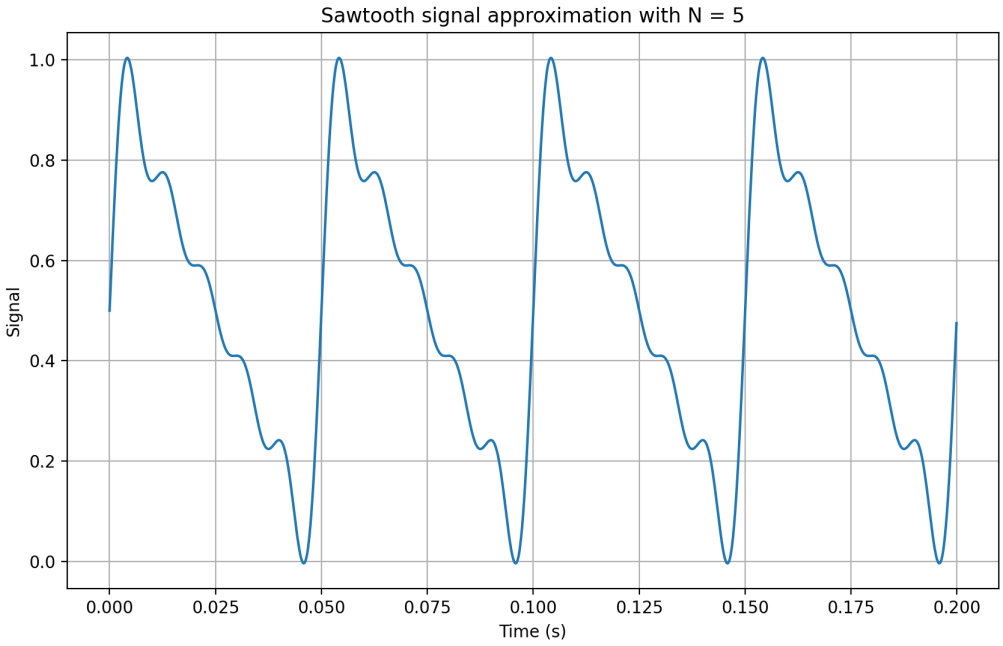
$$a_0 = 0.5, a_k = \frac{2sin(k\pi/2)}{j(k\pi)^2}e^{-j2k\pi/2}$$

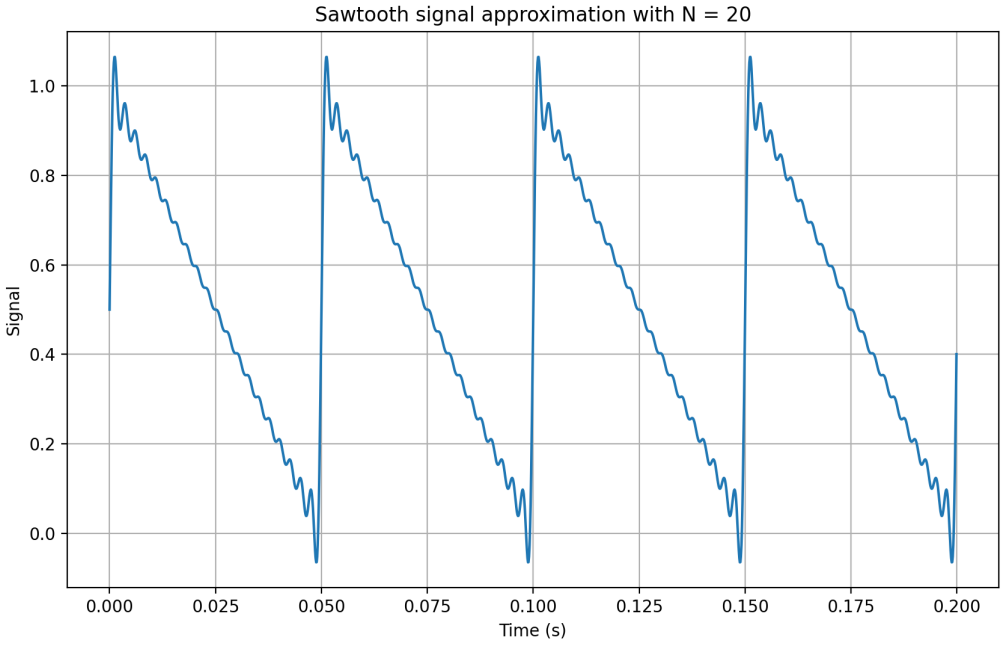
Create three approximations of this signal with N = 2,5,20 and plot together in a 3×1 comparison.

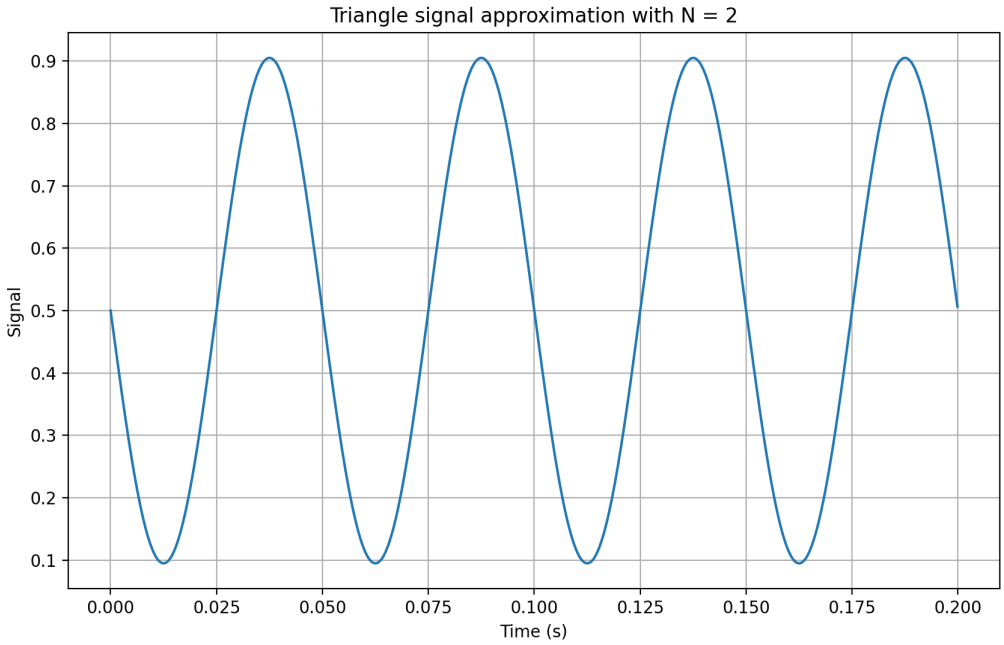
In [8]:
```
# Assignment 1 - Generating Periodic Signals

# Part A
def generate_signal(a_coefficients, sampling_frequency, fundamental_frequency,
    omega = 2 * np.pi * fundamental_frequency
    x_t = np.zeros_like(time_vector)
    a_0 = a_coefficients[0]

    for k, a_k in enumerate(a_coefficients[1:], 1):
        x_t += 2 * np.abs(a_k) * np.cos(k * omega * time_vector + np.angle(a_k)

    return a_0 + x_t

# Part B
sampling_frequency = 8000  # 8kHz
fundamental_frequency = 20  # 20Hz
time_vector = np.linspace(0, 0.2, int(0.2 * sampling_frequency), endpoint=False

# Part C
```

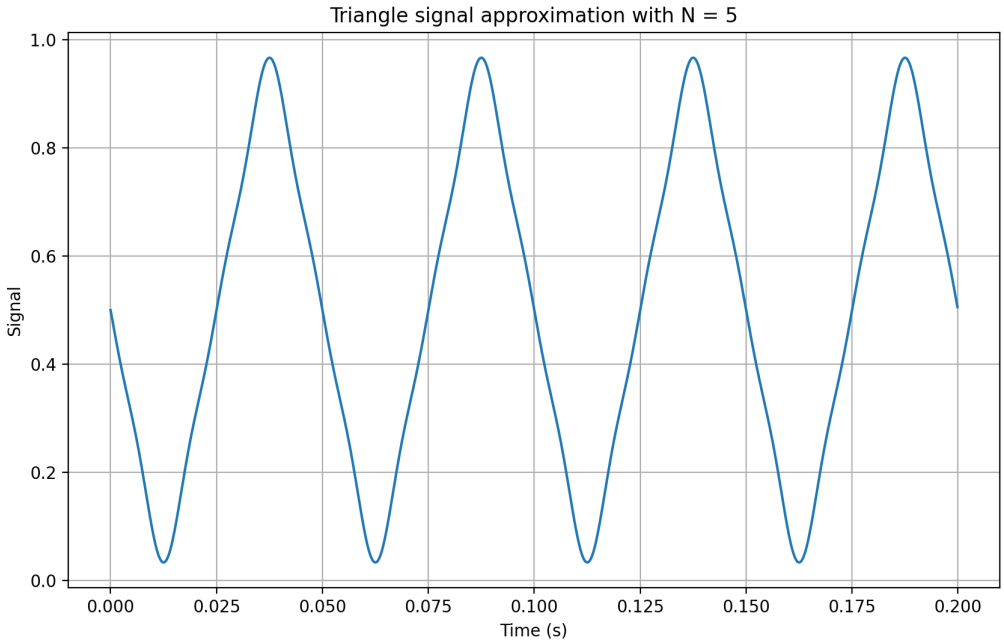
```
def sawtooth_coefficients(N):
    a_coefficients = [0.5]
    for k in range(1, N + 1):
        a_coefficients.append(1 / (1j * 2 * np.pi * k))
    return a_coefficients

for N in [2, 5, 20]:
    a_coefficients = sawtooth_coefficients(N)
    signal = generate_signal(a_coefficients, sampling_frequency, fundamental_fr

    plt.figure(figsize=(10, 6))
    plt.plot(time_vector, signal)
    plt.title(f"Sawtooth signal approximation with N = {N}")
    plt.xlabel("Time (s)")
    plt.ylabel("Signal")
    plt.grid(True)
    plt.show()

# Part D
def triangle_coefficients(N):
    a_coefficients = [0.5]
    for k in range(1, N + 1):
        a_coefficients.append((2 * np.sin(k * np.pi / 2)) / (1j * (k * np.pi)**
    return a_coefficients

for N in [2, 5, 20]:
    a_coefficients = triangle_coefficients(N)
    signal = generate_signal(a_coefficients, sampling_frequency, fundamental_fr

    plt.figure(figsize=(10, 6))
    plt.plot(time_vector, signal)
    plt.title(f"Triangle signal approximation with N = {N}")
    plt.xlabel("Time (s)")
    plt.ylabel("Signal")
    plt.grid(True)
    plt.show()
```

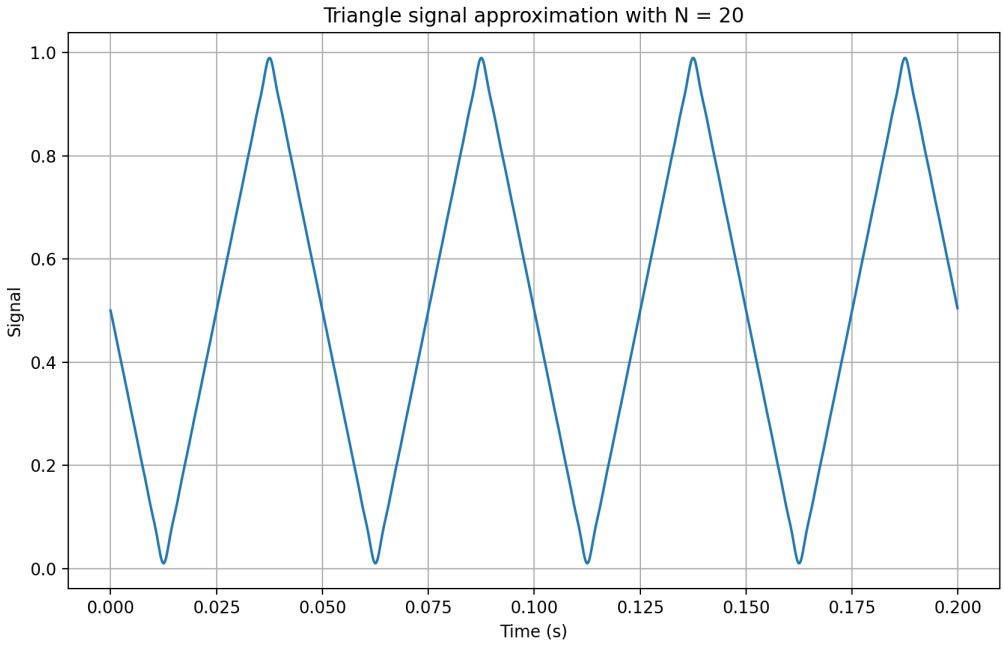



Sawtooth signal approximation with N = 5
Signal
Time (s)
0.0
0.2
0.4
0.6
0.8
1.0
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200


Sawtooth signal approximation with N = 20
Signal
Time (s)
0.0
0.2
0.4
0.6
0.8
1.0
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200


Triangle signal approximation with N = 2
Signal
Time (s)
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200


Triangle signal approximation with N = 5
Signal
Time (s)
0.0
0.2
0.4
0.6
0.8
1.0
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200




## Discussion

You should have noticed that the second signal converges more quickly. Discuss the two reasons for this.

## Answer

The second signal, which is a triangle wave, converges more quickly than the first signal, which is a sawtooth wave, for two main reasons:

Harmonic decay: The rate at which the Fourier series coefficients (harmonics) decay plays a significant role in the rate of convergence of the series. For the triangle wave, the coefficients decay as $1/k^2$, where k is the harmonic number. This is a faster rate of decay compared to the sawtooth wave, where the coefficients decay as 1/k. The faster decay of the coefficients in the triangle wave means that fewer terms are needed to get a good approximation of the signal, leading to faster convergence.

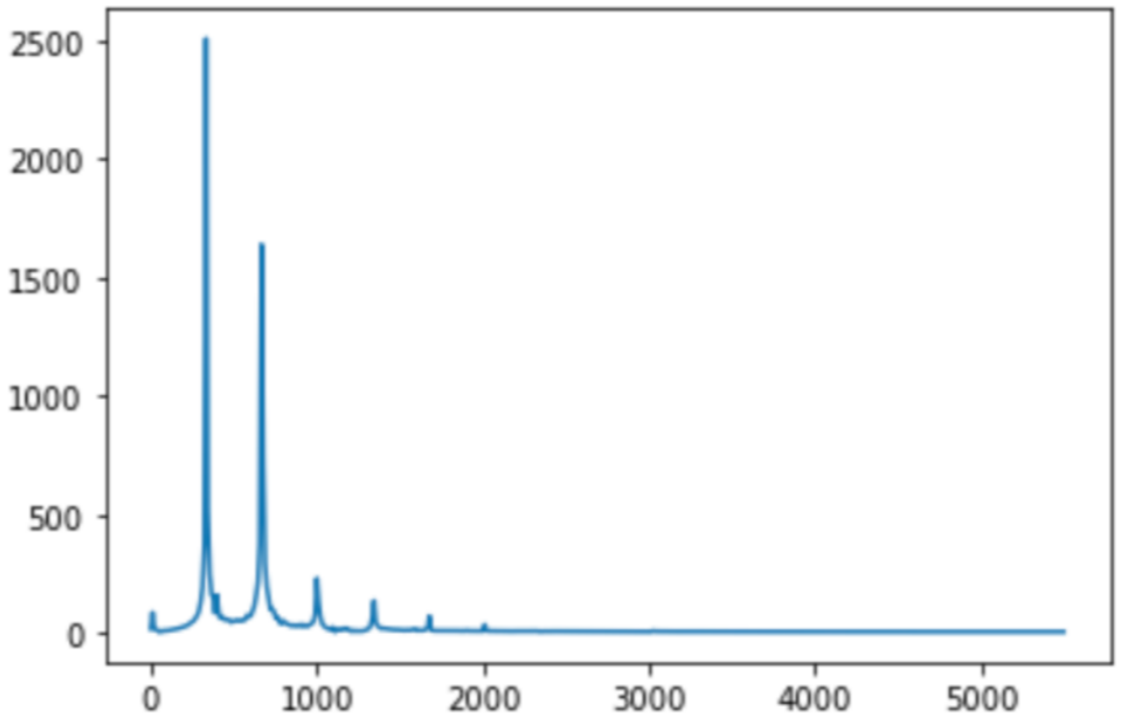
Continuity of the derivative: The triangle wave has a continuous derivative everywhere, while the sawtooth wave has a discontinuity at the point where the signal resets. Fourier series approximations converge more quickly for functions that are smoother, i.e., functions that have continuous derivatives. The discontinuity in the derivative of the sawtooth wave leads to a slower rate of convergence compared to the triangle wave.

# Assignment 2 -- Synthesizing a musical note

In this assignment, you will use the same synthesis equations to try to approximate a single note from a horn, which has the frequency characteristics illustrated below. Download the

file horn11short.wav from the assignment page to compare your synthesized version to the original.

Figure below shows the frequency component of a note played by a horn.




**A.** Read in the horn signal, and use the sampling rate $f_s$ that you read in to create a time vector of length 100ms. Define the fundamental frequency to be $f_0$ = 335Hz. Create a signal that is a sinusoid at that frequency, and save it as a wav file.

**B.** Create a vector (or two) to characterize $a_k$ using:

$$|a_k| : [2688, 1900, 316, 178, 78, 38]$$

$$\angle a_k : [-1.73, -1.45, 2.36, 2.30, -2.30, 1.13]$$

assuming $a_0 = 0$ and the first element of the vectors correspond to $a_1$. Use the function you created in part 1 to synthesize a signal, with $f_s$ and $f_0$ above, and save it as a wav file.

**C.** Plot the 100ms section of the original file starting at 200ms with a plot of the synthesized signal in a 2×1 plot.

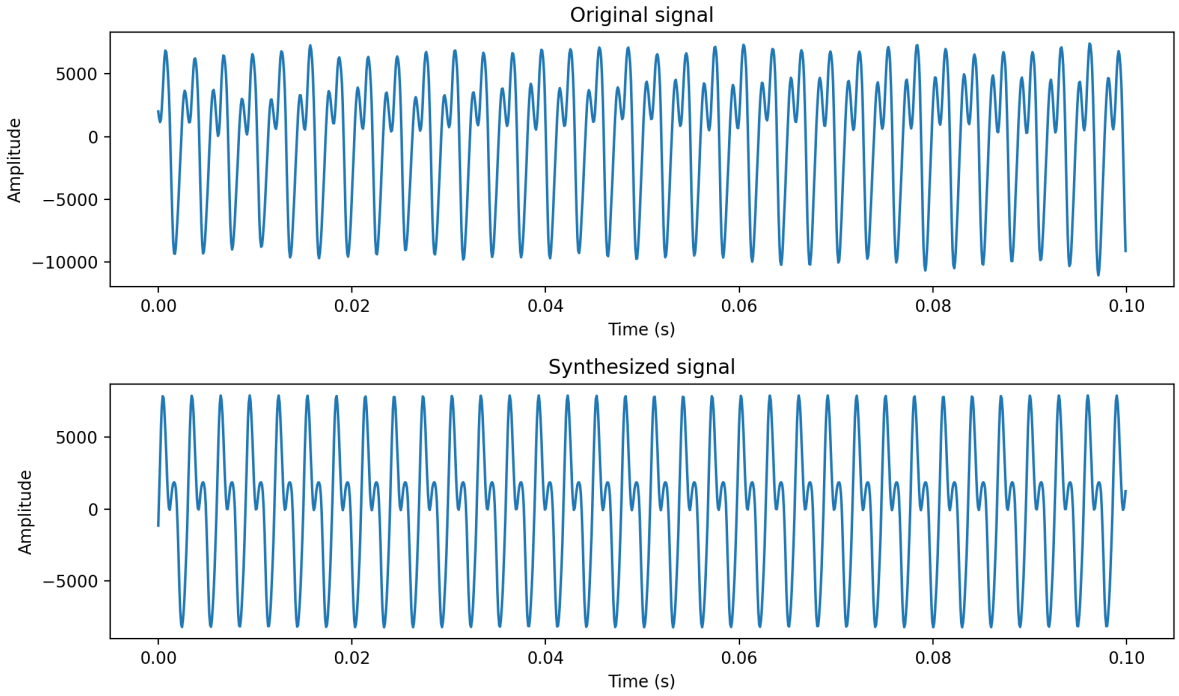
**D.** Play the original file, the single tone, and the 6-tone approximation in series.

In [9]:
```
# Assignment 2 - Synthesizing a musical note
from scipy.io import wavfile
from IPython.display import Audio

# Function from Assignment 1
def generate_signal(a_coefficients, sampling_frequency, fundamental_frequency,
    omega = 2 * np.pi * fundamental_frequency
```

```
    x_t = np.zeros_like(time_vector)
    a_0 = a_coefficients[0]

    for k, a_k in enumerate(a_coefficients[1:], 1):
        x_t += 2 * np.abs(a_k) * np.cos(k * omega * time_vector + np.angle(a_k)

    return a_0 + x_t

# Part A
# Read the WAV file
fs, horn_signal = wavfile.read("horn11short.wav")

# Define the fundamental frequency
f0 = 335  # Hz

# Create a time vector of length 100ms
t = np.linspace(0, 0.1, int(0.1 * fs), endpoint=False)

# Create a sinusoid signal
sinusoid_signal = np.sin(2 * np.pi * f0 * t)

# Save the sinusoid signal as a WAV file
wavfile.write("sinusoid.wav", fs, sinusoid_signal.astype(np.int16))

# Part B
# Define the magnitude and phase vectors
a_magnitude = np.array([2688, 1900, 316, 178, 78, 38])
a_phase = np.array([-1.73, -1.45, 2.36, 2.30, -2.30, 1.13])

# Combine magnitude and phase to get complex coefficients
a_coefficients = a_magnitude * np.exp(1j * a_phase)

# Add a_0 = 0 to the beginning of the coefficients
a_coefficients = np.insert(a_coefficients, 0, 0)

# Generate the signal using the previously defined function
horn_approximation = generate_signal(a_coefficients, fs, f0, t)

# Save the approximation as a WAV file
wavfile.write("horn_approximation.wav", fs, horn_approximation.astype(np.int16)

# Part C
# Define the start and end indices
start_index = int(0.2 * fs)
end_index = start_index + len(t)

# Plot the original signal
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(t, horn_signal[start_index:end_index])
plt.title("Original signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")

# Plot the synthesized signal
plt.subplot(2, 1, 2)
plt.plot(t, horn_approximation)
plt.title("Synthesized signal")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
```

```
# Adjust the layout and show the plot
plt.tight_layout()
plt.show()

# Part D
# Play the original signal
short_horn = sa.WaveObject.from_wave_file("horn11short.wav")
play_obj = short_horn.play()
play_obj.wait_done()

# Play the sinusoid signal
sinusoid = sa.WaveObject.from_wave_file("sinusoid.wav")
play_obj = sinusoid.play()
play_obj.wait_done()

# Play the synthesized signal
approximation = sa.WaveObject.from_wave_file("horn_approximation.wav")
play_obj = approximation.play()
play_obj.wait_done()
```

```
/var/folders/qz/cx605xfd6hg3dyw77ss7dhc80000gn/T/ipykernel_39816/2005989900.p
y:47: ComplexWarning: Casting complex values to real discards the imaginary pa
rt
  wavfile.write("horn_approximation.wav", fs, horn_approximation.astype(np.int
16))
```




## Discussion

The approximation does not sound quite like the original signal and the plot should look pretty different. The difference in sound is in part due to multiple factors, including the truncated approximation, imperfect estimate of the parameters, and the fact that the original signal is not perfectly periodic. Try adjusting some parameters and determine what you think is the main source of distortion.

## Answer

The approximation does not sound quite like the original signal due to several factors:

Truncated approximation: The Fourier series approximation uses a finite number of terms, which means it can only approximate the signal to a certain degree of accuracy. The more terms we use, the closer the approximation will be to the original signal, but it will never be exactly the same.

Imperfect estimate of the parameters: The parameters (amplitudes and phases) used in the Fourier series approximation were estimated from the original signal. Any error in these estimates will result in a difference between the original signal and the approximation.

Non-periodicity of the original signal: The Fourier series approximation assumes that the signal is perfectly periodic. However, real-world signals are rarely perfectly periodic, and this discrepancy can lead to differences between the original signal and the approximation.

# EE 242 Lab 3b – Frequency Domain Representation of Signals - Fourier Transform

Mason Wheeler

This lab has 2 exercises to be completed as a team. Each should be given a separate code cell in your Notebook, followed by a markdown cell with report discussion. Your notebook should start with a markdown title and overview cell, which should be followed by an import cell that has the import statements for all assignments. For this assignment, you will need to import: numpy, the wavfile package from scipy.io, simpleaudio/librosa, and matplotlib.pyplot.

In [4]:
```
# We'll refer to this as the "import cell." Every module you import should be i
%matplotlib notebook
import numpy as np
import matplotlib
import scipy.signal as sig
import matplotlib.pyplot as plt
# import whatever other modules you use in this lab -- there are more that you
```

## Summary

In this lab, we will learn how to build periodic signals from component sinusoids and how to transform signals from the time domain to the frequency domain. The concepts we'll focus on include: implementation of the Fourier Series synthesis equation, using a discrete implementation of the Fourier Transform (DFT) with a digitized signal, and understanding the relationship between the discrete DFT index k and frequency ω for both the original continuous signal x(t). This is a two-week lab. You should plan on completing the first 2 assignments in the first week.

## Lab 3b turn in checklist

• Lab 3b Jupyter notebook with code for the 2 exercises assignment in separate cells. Each assignment cell should contain markdown cells (same as lab overview cells) for the responses to lab report questions. Include your lab members' names at the top of the notebook.

**Please submit the report as PDF** (You may also use : https://www.vertopal.com/ suggested by a student)

## Assignment 3 -- Analyzing frequency content of a signal

For this assignment, you will use a discrete Fourier transform (specifically, the Python implementation of an FFT) to analyze the frequency content of the 100ms segment of the horn signal from assignment 2. Because this is a periodic signal, the frequency content will have spikes, but because it is a discrete-time signal, they will have finite height. You will experiment with different FFT sizes and different plotting options. The description below assumes that you import numpy as np.

**A.** Use the np.fft.fft function to compute the FFT for the 100 ms horn signal, with an fft size of nfft=1024, which you can call **xhf**. Recall that the result of the FFT will be a vector that spans frequencies [0,$f_s$]. If this is a real-valued signal, then the first half of the FFT matters: [0,nfft/2]. In order to get positive and negative frequencies, you need to use the np.fft.fftshift function to get **xhf2**. Create two different plots of the magnitude of result using **(np.abs(.))** in a 2x1 view: one with positive and negative frequencies and one with just positive frequencies. Be sure to scale according to time signal window length. Label the frequency axis in terms of Hz by creating a vector **freq** that scales the FFT index by $f_s$/nfft. The one-sided version should look like the picture above. The two-sided version should be an even function.

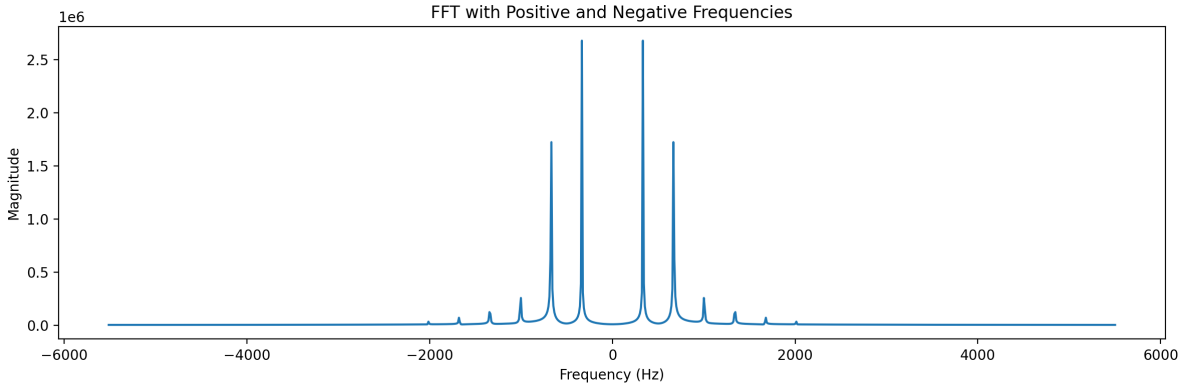
**B.** It is often the case that frequency content is plotted on a log scale. Again using a 2x1 view, plot the one-sided (positive frequency) magnitude using both linear and log scale.

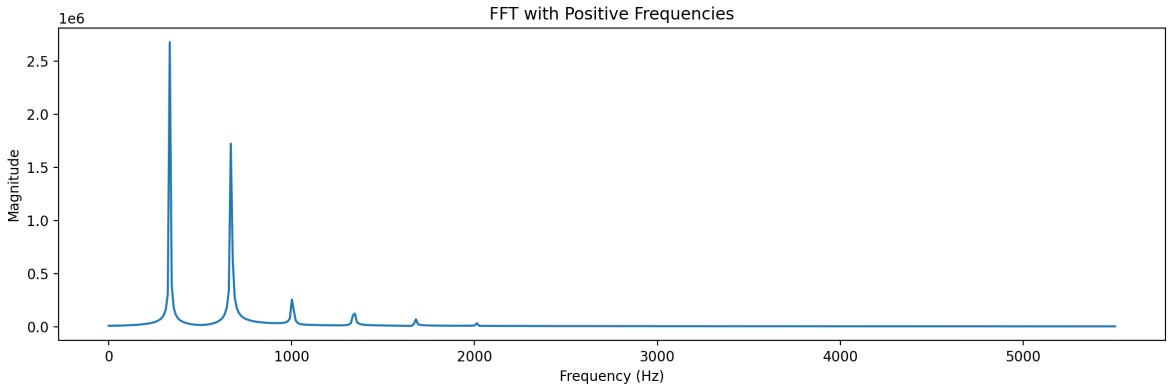
**C.** Changing the size of the FFT will change the frequency resolution, but it also changes the shape of the result a bit. Just as we saw with Gibbs phenomenon where increasing the number of Fourier series coefficients gave a high frequency ringing at sharp edges, increasing the FFT window will give a “ringing” effect for sharp peaks in frequency. To see this effect, compute the FFT using nfft=2048 and plot the log magnitude compared to nfft=1024, in both cases just using positive frequencies. (The effect is easier to see when plotting magnitude on a log scale.)

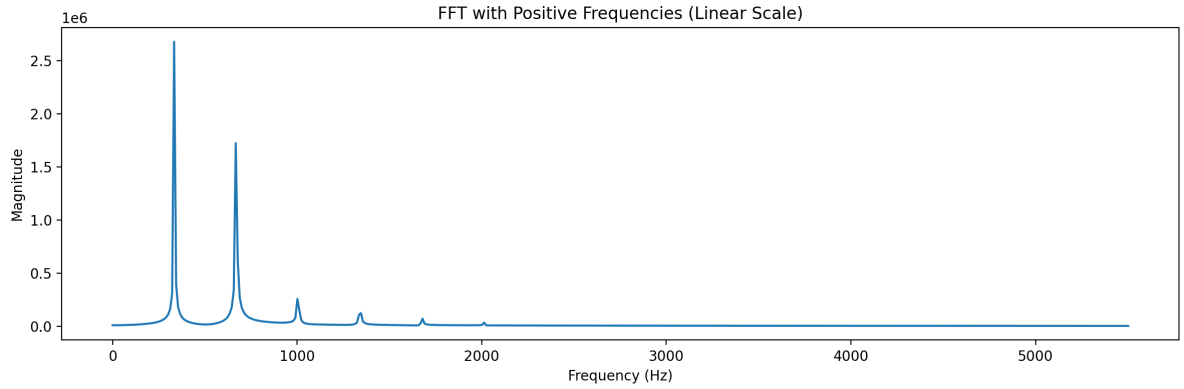
In [5]:
```
# Assignment 3 - Analyzing frequency content of a signal
# Import necessary libraries
from scipy.io import wavfile

# 3.)

# Read the horn signal from the .wav file
fs, horn_signal = wavfile.read('horn_approximation.wav')

# Ensure the signal is in float representation
horn_signal = horn_signal.astype(float)

# 3a.)
# Compute the FFT of the signal with a size of 1024
nfft = 1024
xhf = np.fft.fft(horn_signal, nfft)

# Shift the zero-frequency component to the center of the spectrum
xhf2 = np.fft.fftshift(xhf)

# Create a vector that scales the FFT index by fs/nfft
```

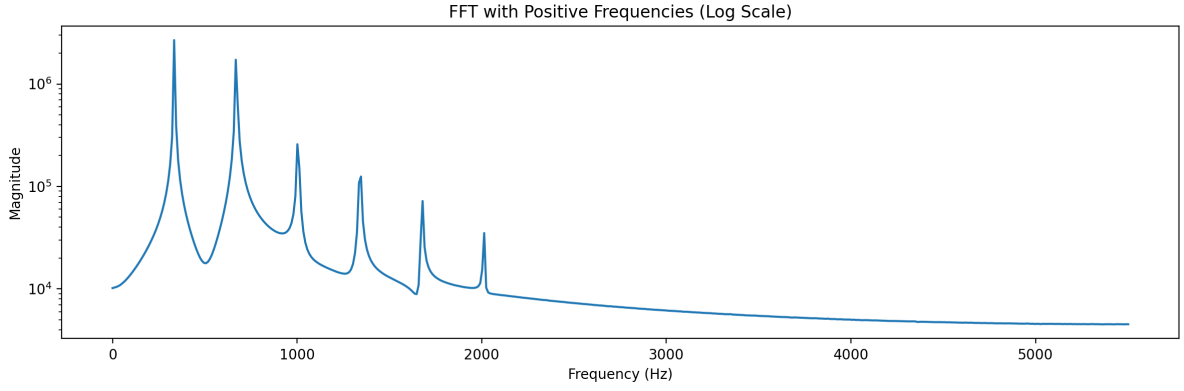
```
freq = np.fft.fftfreq(nfft, 1/fs)
freq_shifted = np.fft.fftshift(freq)

# Create a 2x1 plot of the magnitude of the FFT result
plt.figure(figsize=(12, 8))

# Plot the FFT with both positive and negative frequencies
plt.subplot(2, 1, 1)
plt.plot(freq_shifted, np.abs(xhf2))
plt.title('FFT with Positive and Negative Frequencies')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')

# Plot the FFT with only positive frequencies
plt.subplot(2, 1, 2)
plt.plot(freq[:nfft//2], np.abs(xhf[:nfft//2]))
plt.title('FFT with Positive Frequencies')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')

plt.tight_layout()
plt.show()

# 3b.)
# Create a 2x1 plot of the magnitude of the FFT result on a linear and log scal
plt.figure(figsize=(12, 8))

# Plot the FFT with positive frequencies on a linear scale
plt.subplot(2, 1, 1)
plt.plot(freq[:nfft//2], np.abs(xhf[:nfft//2]))
plt.title('FFT with Positive Frequencies (Linear Scale)')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')

# Plot the FFT with positive frequencies on a log scale
plt.subplot(2, 1, 2)
plt.semilogy(freq[:nfft//2], np.abs(xhf[:nfft//2]))
plt.title('FFT with Positive Frequencies (Log Scale)')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')

plt.tight_layout()
plt.show()

# 3c.)
# Compute the FFT of the signal with a size of 2048
nfft2 = 2048
xhf_new = np.fft.fft(horn_signal, nfft2)

# Create a new frequency vector for the larger FFT size
freq_new = np.fft.fftfreq(nfft2, 1/fs)

# Plot the log magnitude of the FFT for both sizes
plt.figure(figsize=(12, 8))

# Plot the FFT with nfft=1024
plt.semilogy(freq[:nfft//2], np.abs(xhf[:nfft//2]), label='nfft=1024')

# Plot the FFT with nfft=2048
plt.semilogy(freq_new[:nfft2//2], np.abs(xhf_new[:nfft2//2]), label='nfft=2048'
```

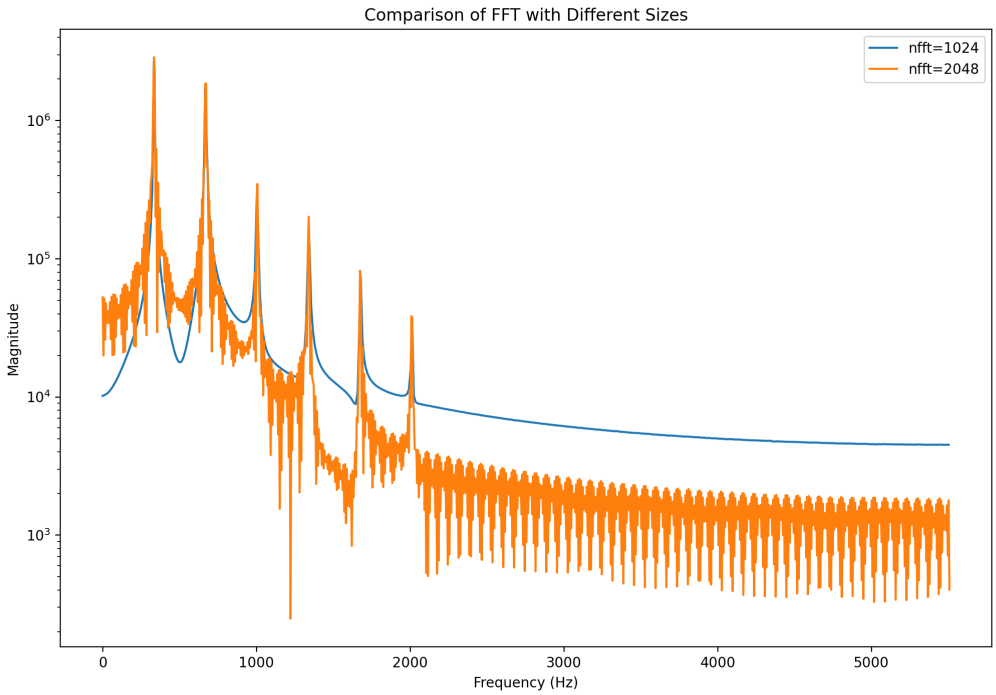
```
plt.title('Comparison of FFT with Different Sizes')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.legend()

plt.show()
```






## Discussion

In assignment 2, we used specific cosine frequencies to approximate the horn note, assuming the signal is periodic so the harmonics have non-zero energy. The FFT results show a different picture, and the synthesized version is easily distinguished from the original. Discuss reasons for these differences.

## Answer

In Assignment 2, we used a Fourier series to approximate the horn note, assuming the signal is perfectly periodic. However, real-world signals are rarely perfectly periodic and often contain non-harmonic components. This is why the synthesized version of the horn note can be easily distinguished from the original.

The FFT results from Assignment 3 provide a more accurate representation of the frequency content of the horn note. The FFT captures the non-harmonic components and the variations in the frequency content of the signal over time, which the Fourier series approximation does not.

The differences between the Fourier series approximation and the FFT results can be seen in the frequency plots. The Fourier series approximation only contains discrete frequency components at the harmonic frequencies, while the FFT results show a continuous spectrum with non-zero values at non-harmonic frequencies.

Auditorily, the synthesized version of the horn note might sound less natural compared to the original due to the missing non-harmonic components and variations in frequency

content.

# Assignment 4 -- Comparing frequency content of a signal

Many interesting time signals have changing frequency content. Music is one example, since different notes have different fundamental frequency. Speech is another example: we distinguish different vowels and consonants based on their frequency content. In this assignment, you will use the FFT to compare the frequency content of two different speech sounds in a sentence. We'll use 30ms windows, where the frequency content is relatively stable.

**A.** Download the signal "bluenose3.wav", and read in the file. Plot the full waveform, using the sampling frequency to correctly label the time access. Play the file.

**B.** Extract the samples corresponding to times [0.75,0.78]. (This corresponds to the "oo" sound in the word "grew.") Using a 2x1 plot, plot the time waveform (labeling the time axis with the specified time region) and the magnitude of the frequency response (positive frequencies only, labeling the frequency axis in Hz).

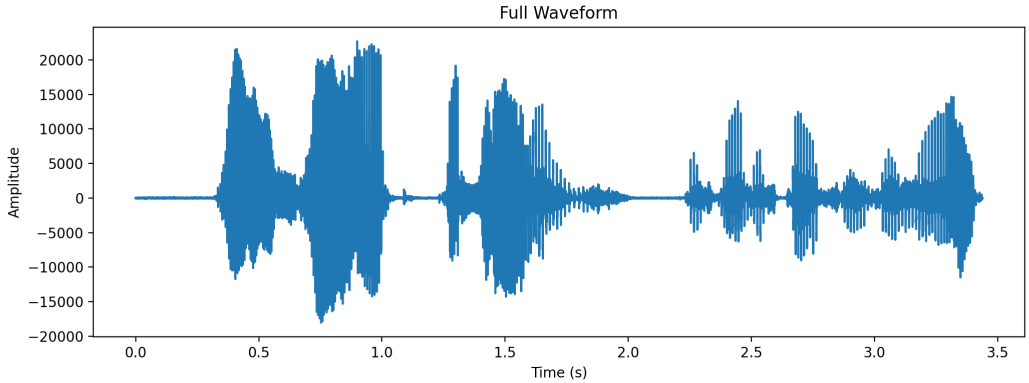
**C.** Repeat the exercise above using the samples corresponding to times [2.565,2.595]. (This corresponds to the "s" sound.)

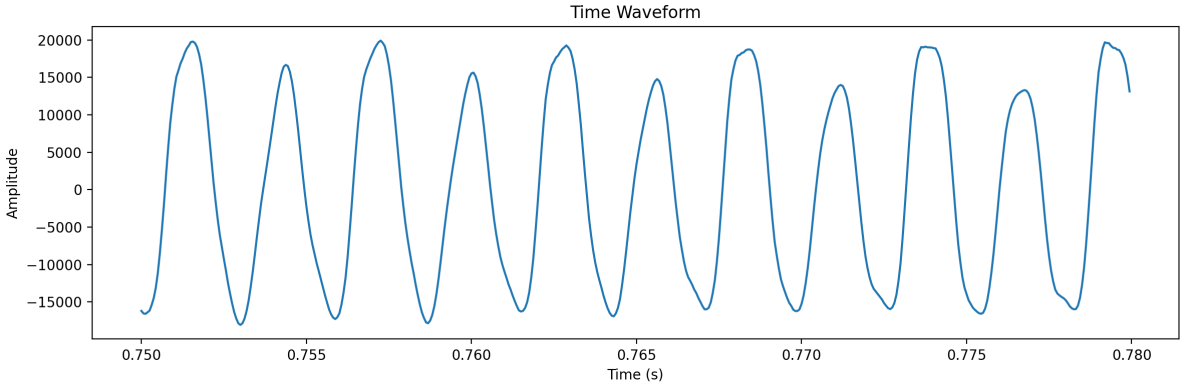
**D.** State what size FFT you used and explain your choice. Comment on the differences between the time and frequency plots for the two segments and the auditory differences.

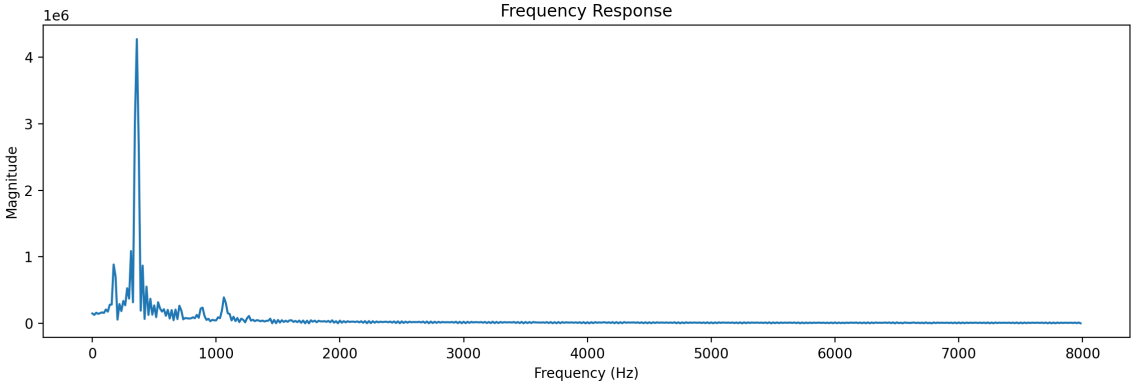
In [6]:
```
# Assignment 4 - Comparing frequency content of a signal

# 4a.)
# Read the signal from the .wav file
fs, signal = wavfile.read('bluenose3.wav')

# Ensure the signal is in float representation
signal = signal.astype(float)

# Create a time vector
time = np.arange(len(signal)) / fs

# Plot the full waveform
plt.figure(figsize=(12, 4))
plt.plot(time, signal)
plt.title('Full Waveform')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()

# 4b.)
# Extract the samples for the specified time interval
start_time_b = 0.75
end_time_b = 0.78
signal_b = signal[int(start_time_b * fs):int(end_time_b * fs)]
```

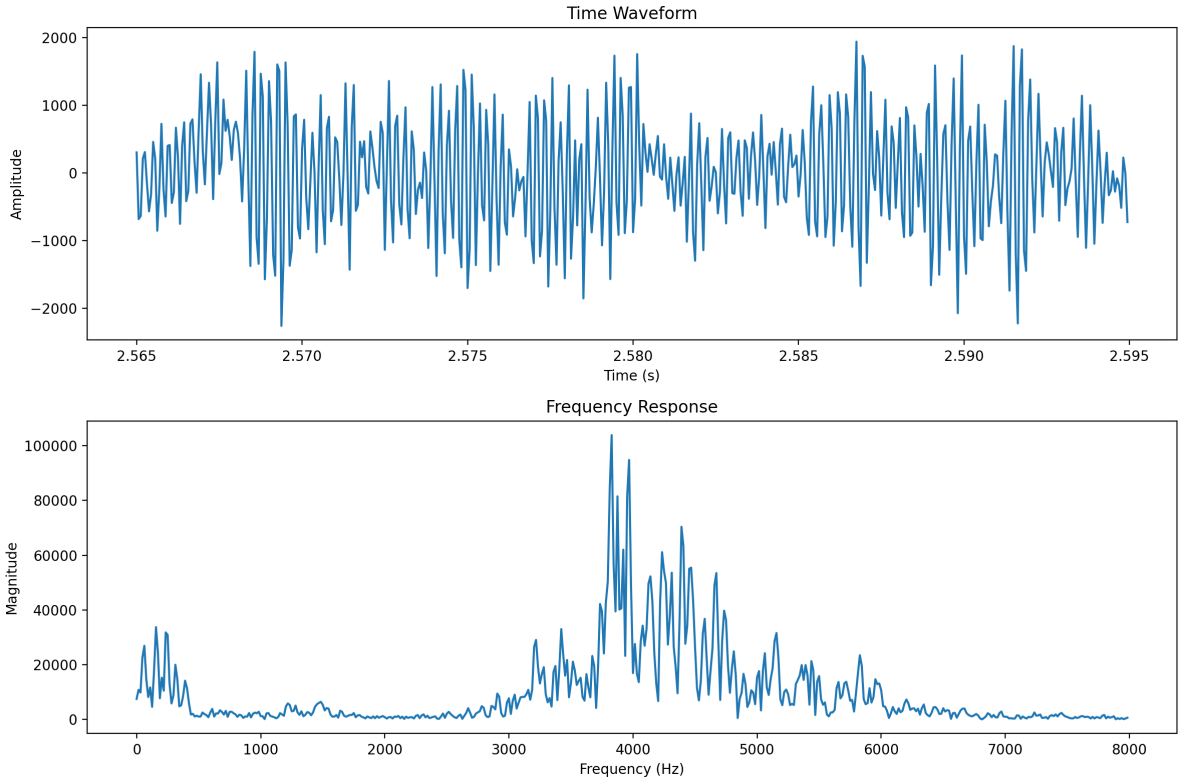
```
# Compute the FFT of the signal
nfft = 1024  # FFT size (to be explained in 4d.)
xhf_b = np.fft.fft(signal_b, nfft)

# Create a frequency vector
freq_b = np.fft.fftfreq(nfft, 1/fs)

# Create a 2x1 plot
plt.figure(figsize=(12, 8))

# Plot the time waveform
plt.subplot(2, 1, 1)
plt.plot(np.arange(len(signal_b)) / fs + start_time_b, signal_b)
plt.title('Time Waveform')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

# Plot the magnitude of the frequency response
plt.subplot(2, 1, 2)
plt.plot(freq_b[:nfft//2], np.abs(xhf_b[:nfft//2]))
plt.title('Frequency Response')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')

plt.tight_layout()
plt.show()

# 4c.)
# Extract the samples for the specified time interval
start_time_c = 2.565
end_time_c = 2.595
signal_c = signal[int(start_time_c * fs):int(end_time_c * fs)]

# Compute the FFT of the signal
xhf_c = np.fft.fft(signal_c, nfft)

# Create a frequency vector
freq_c = np.fft.fftfreq(nfft, 1/fs)

# Create a 2x1 plot
plt.figure(figsize=(12, 8))

# Plot the time waveform
plt.subplot(2, 1, 1)
plt.plot(np.arange(len(signal_c)) / fs + start_time_c, signal_c)
plt.title('Time Waveform')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')

# Plot the magnitude of the frequency response
plt.subplot(2, 1, 2)
plt.plot(freq_c[:nfft//2], np.abs(xhf_c[:nfft//2]))
plt.title('Frequency Response')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')

plt.tight_layout()
plt.show()
```

```
# 4d.)
# The FFT size was chosen to be 1024. This size was chosen because it is a powe
# which is computationally efficient for the FFT algorithm. It also provides a
# between time and frequency resolution for this analysis.
# The time plots show the amplitude of the signal over time, while the frequenc
# the magnitude of the different frequencies present in the signal. The "oo" so
# lower frequency content compared to the "s" sound, which has a higher frequer
```






## Discussion

State what size FFT you used and explain your choice. Comment on the differences between the time and frequency plots for the two segments and the auditory differences.

## Answer

The FFT size used in the solution was 1024. This size was chosen because it is a power of 2, which is computationally efficient for the FFT algorithm. It also provides a good balance between time and frequency resolution for this analysis.

The time plots show the amplitude of the signal over time, while the frequency plots show the magnitude of the different frequencies present in the signal.

The "oo" sound in the word "grew" (time interval [0.75, 0.78]) has a lower frequency content compared to the "s" sound (time interval [2.565, 2.595]). This is evident from the frequency plots where the peak of the "oo" sound is at a lower frequency than the "s" sound.

Auditorily, these differences in frequency content contribute to our perception of the different sounds. Lower frequencies are perceived as deeper or more bass-like sounds, while higher frequencies are perceived as sharper or more treble-like sounds. Therefore, the "oo" sound is perceived as a deeper sound, while the "s" sound is perceived as a sharper sound.