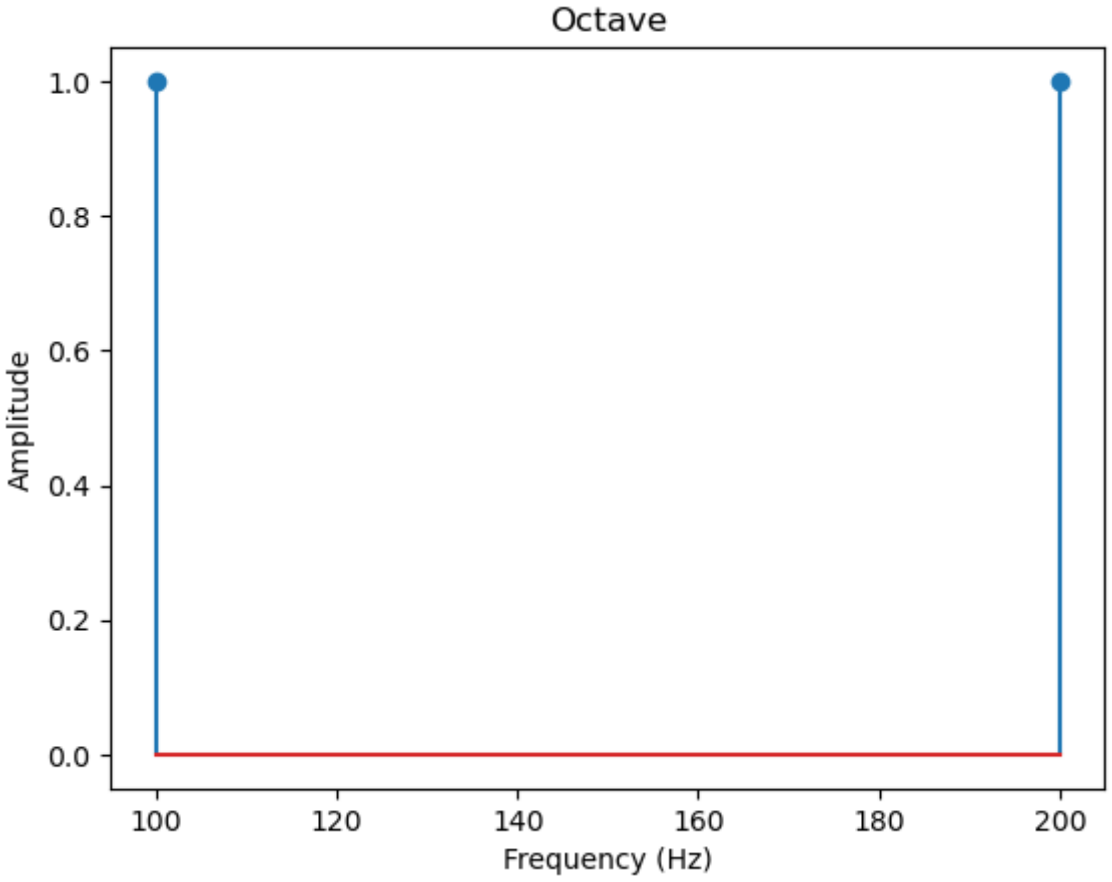
Assignment A4: Sounds and Spectra

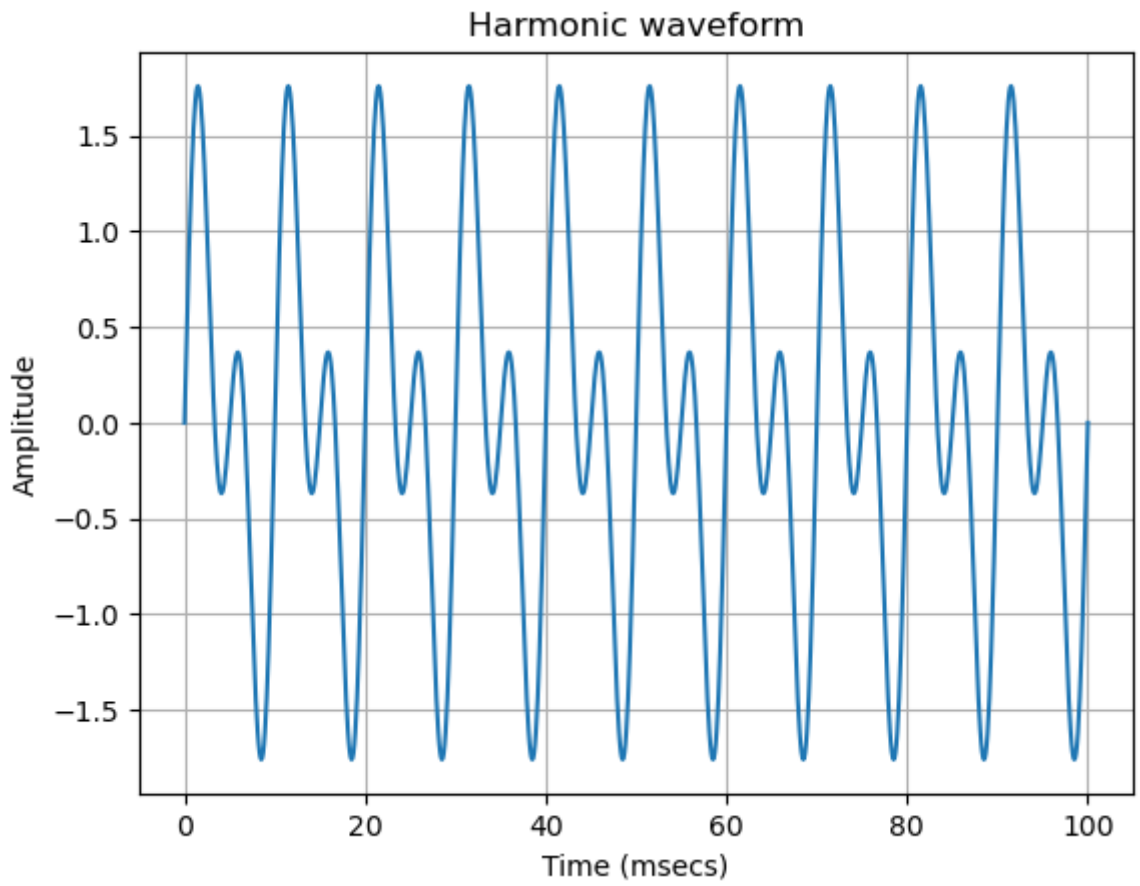
1a. Harmonics

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import A4Final
from scipy.io import wavfile
import math
import scipy
import IPython

def harmonic(t, f1=1, alist=1, φlist=0):
    f1 = f1 * 10
    if np.isscalar(alist):
        return alist * np.cos(2*np.pi*f1*t + φlist)
    else:
        harmonics_1 = [a * np.cos(2*np.pi*n*f1*t + φ) for n, (a, φ) in enumerate(zip(alist, φlist), start=1)]
        harmonics_2 = [a * np.sin(2*np.pi*n*f1*t + φ) for n, (a, φ) in enumerate(zip(alist, φlist), start=1)]
        harmonics = harmonics_2[0] - harmonics_1[1]
        return harmonics

def cosines(t, flist=1, alist=1, φlist=0):
    if np.isscalar(flist):
        return alist * np.cos(2*np.pi*flist*t + φlist)
    else:
        cosines = [a * np.cos(2*np.pi*f*t + φ) for f, a, φ in zip(flist, alist, φlist)]
        return np.sum(cosines, axis=0)
```

The figure below shows a 100 Hz fundamental with an octave harmonic

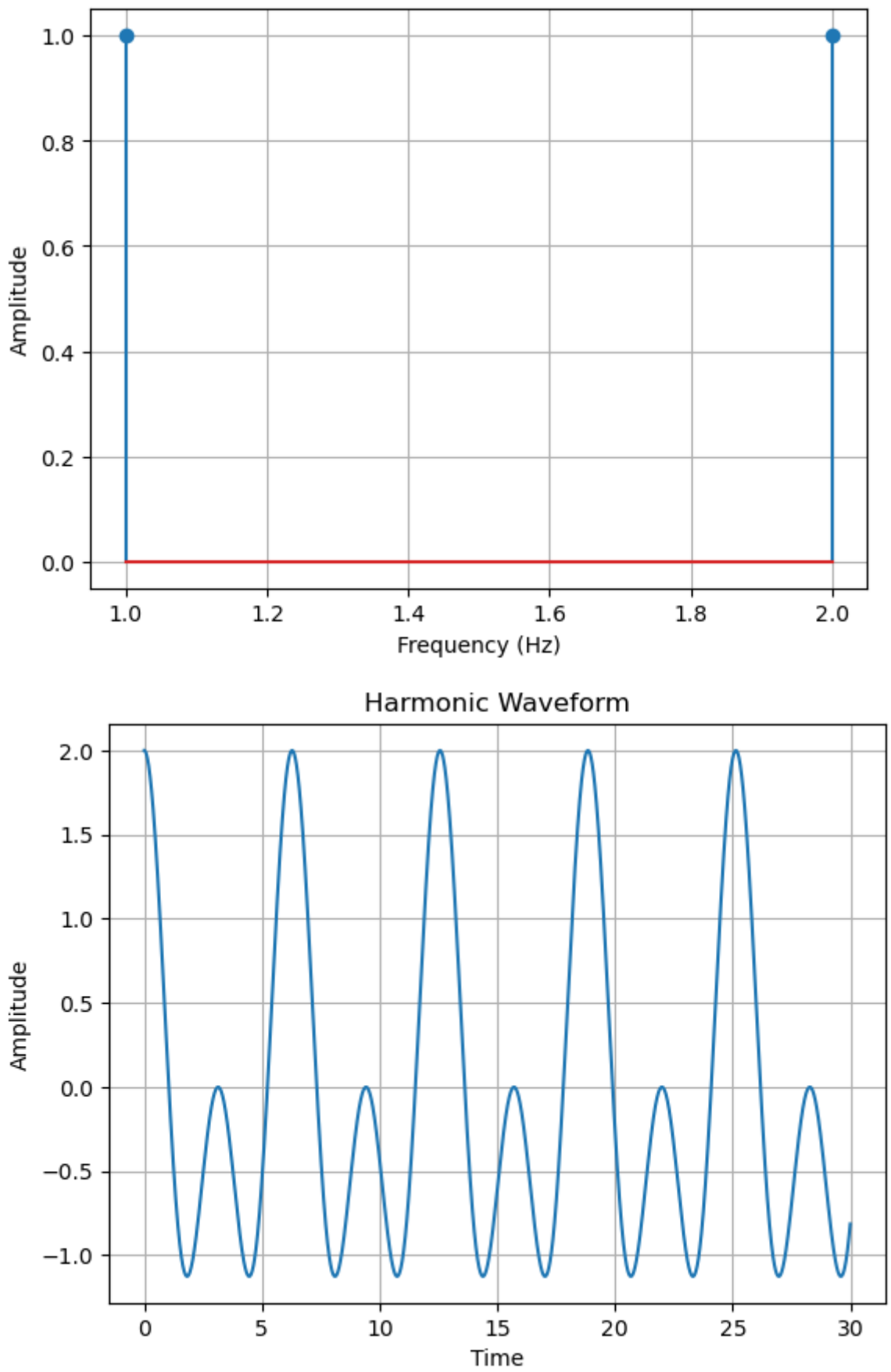
In [2]:
```
# Example usage
fs = 1000  # Sample rate (Hz)
T = 1/fs   # Sample period (s)
t = np.arange(0, 1.001, T)

# Simple harmonic with fundamental frequency of 100 Hz and second harmonic at 200 Hz
h = harmonic(t, f1 = 1, alist=[1, 1], φlist=[0, np.pi/2])
plt.figure()
plt.stem([100, 200], [1, 1], use_line_collection=True)
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude')
plt.title('Octave')
plt.figure()
plt.plot(t * 100, h)
plt.xlabel('Time (msecs)')
plt.ylabel('Amplitude')
plt.title('Harmonic waveform')
plt.grid()
plt.show()
```





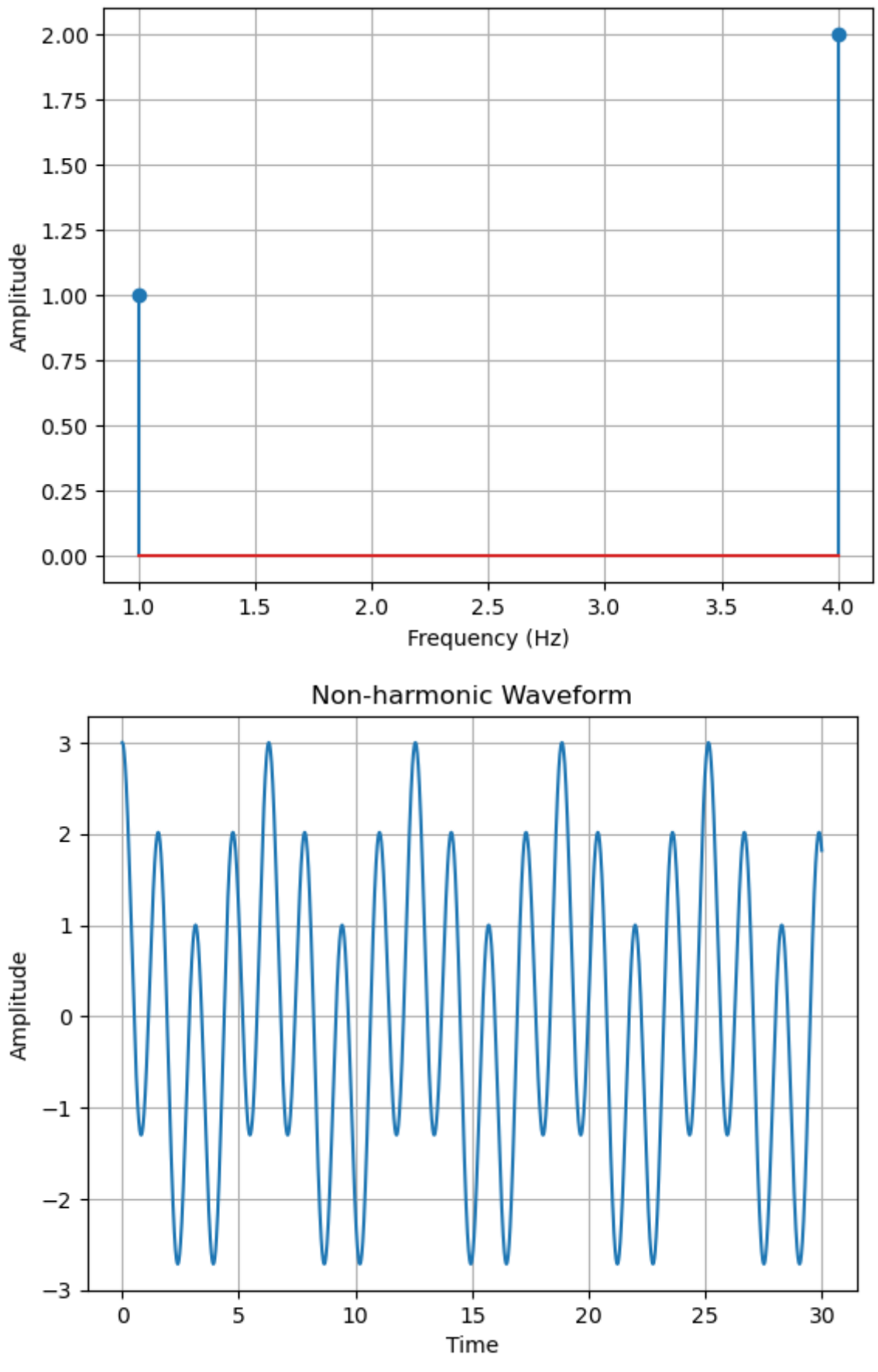
Harmonic

In [3]:
```
A4Final.show_harmonics(t=np.arange(0,30,0.01), g=A4Final.harmonic, f=1, alist=[1,1],
                       phase_list=[0,0], title="Harmonic Waveform")
```



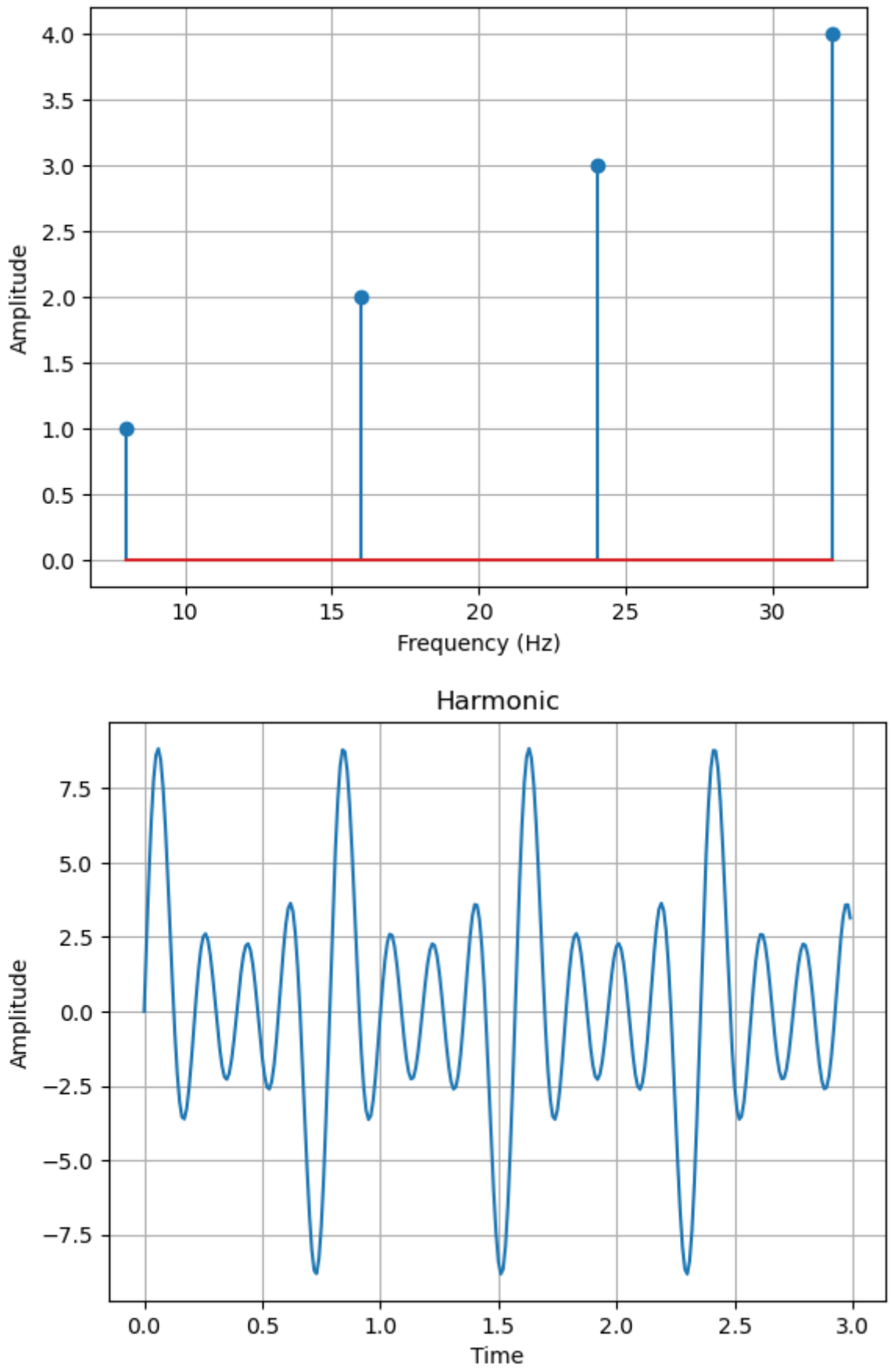
Non-harmonic

In [4]:
```
A4Final.show_harmonics(t=np.arange(0,30,0.01), g=A4Final.cosine, f=[1,4],
                       alist=[1,2], phase_list=[0,0], title="Non-harmonic Waveform")
```



1b. Pitch and tone

In [137…
```
Music_t = np.arange(0,800,0.01)
frequence = 8
Set_alist = [1,2,3,4]
Plist = [-np.pi/2,-np.pi/2,-np.pi/2,-np.pi/2]
A4Final.show_harmonics(t=np.arange(0,3,0.01), g=A4Final.harmonic, f=frequence,
                       alist=Set_alist, phase_list = Plist, title="Harmonic")
Data_val = np.array([A4Final.harmonic(t = t, f=frequence, alist=Set_alist, phase_list = Plist) for t in Music_t])
scipy.io.wavfile.write("1b_Pitch&tone_HarmonicMusic1.wav", rate = 44100, data = Data_val.astype(np.float32))
oneb_Music1 = IPython.display.Audio("1b_Pitch&tone_HarmonicMusic1.wav")
```

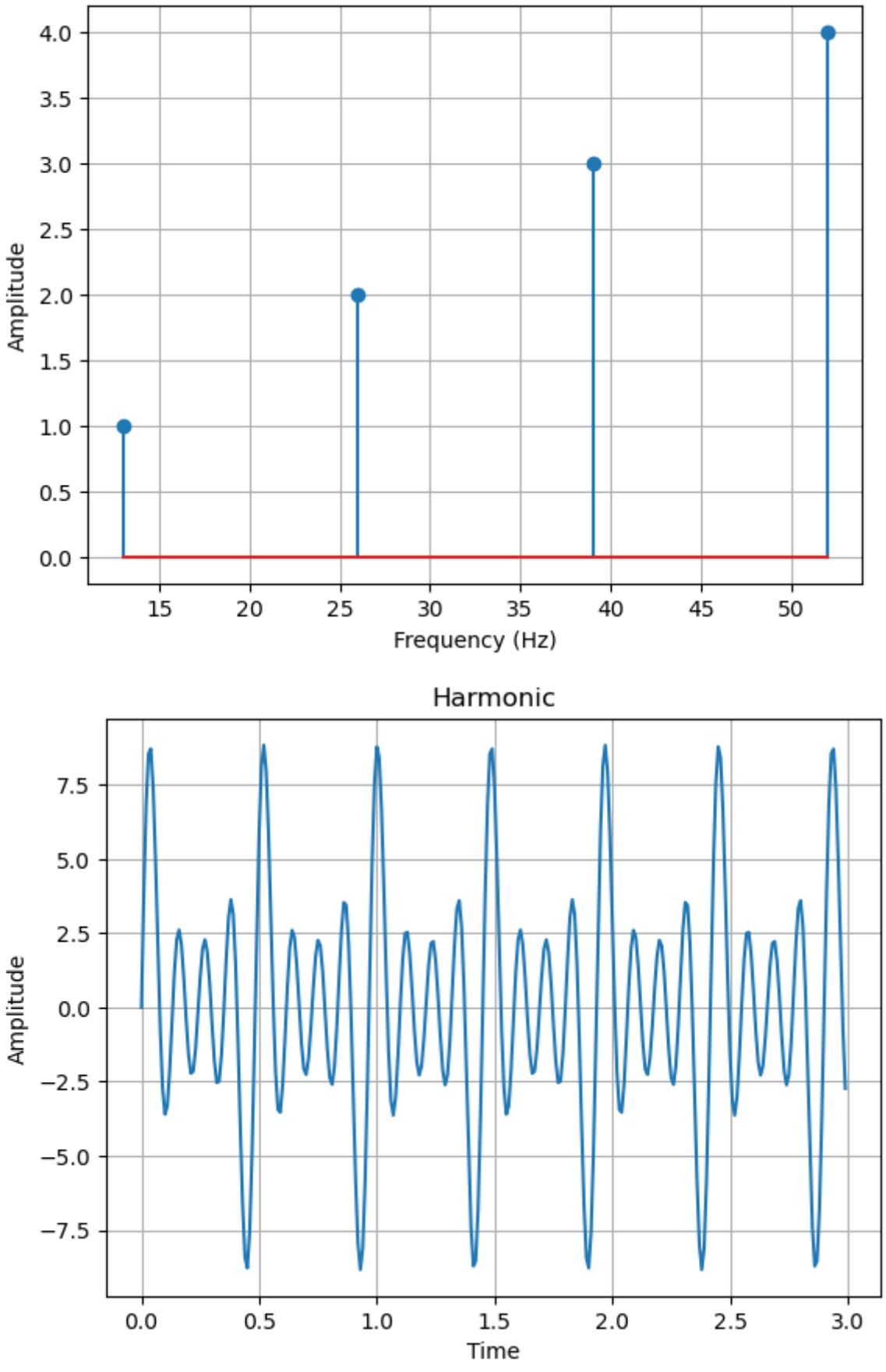


In [6]:
```
oneb_Music1
```

Out[6]:

0:00 / 0:01

In [7]:
```
Music_t = np.arange(0,800,0.01)
frequence = 13
Set_alist = [1,2,3,4]
Plist = [-np.pi/2,-np.pi/2,-np.pi/2,-np.pi/2]
A4Final.show_harmonics(t=np.arange(0,3,0.01), g=A4Final.harmonic, f=frequence,
               alist=Set_alist, phase_list = Plist, title="Harmonic")
Data_val = np.array([A4Final.harmonic(t = t, f=frequence, alist=Set_alist, phase_list = Plist) for t in Music_t])
scipy.io.wavfile.write("1b_Pitch&tone_HarmonicMusic2.wav", rate = 44100, data = Data_val.astype(np.float32))
oneb_Music2 = IPython.display.Audio("1b_Pitch&tone_HarmonicMusic2.wav")
```



In [8]:
```
oneb_Music2
```

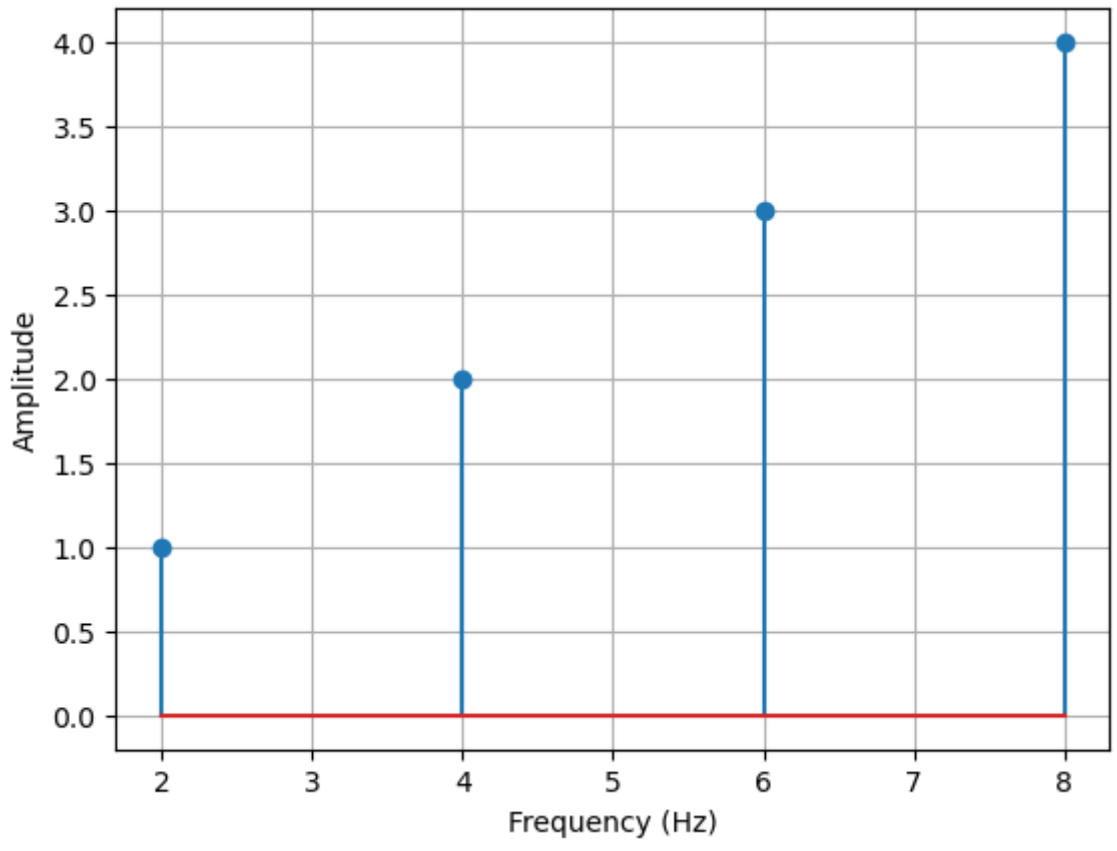
Out[8]:

0:00 / 0:01

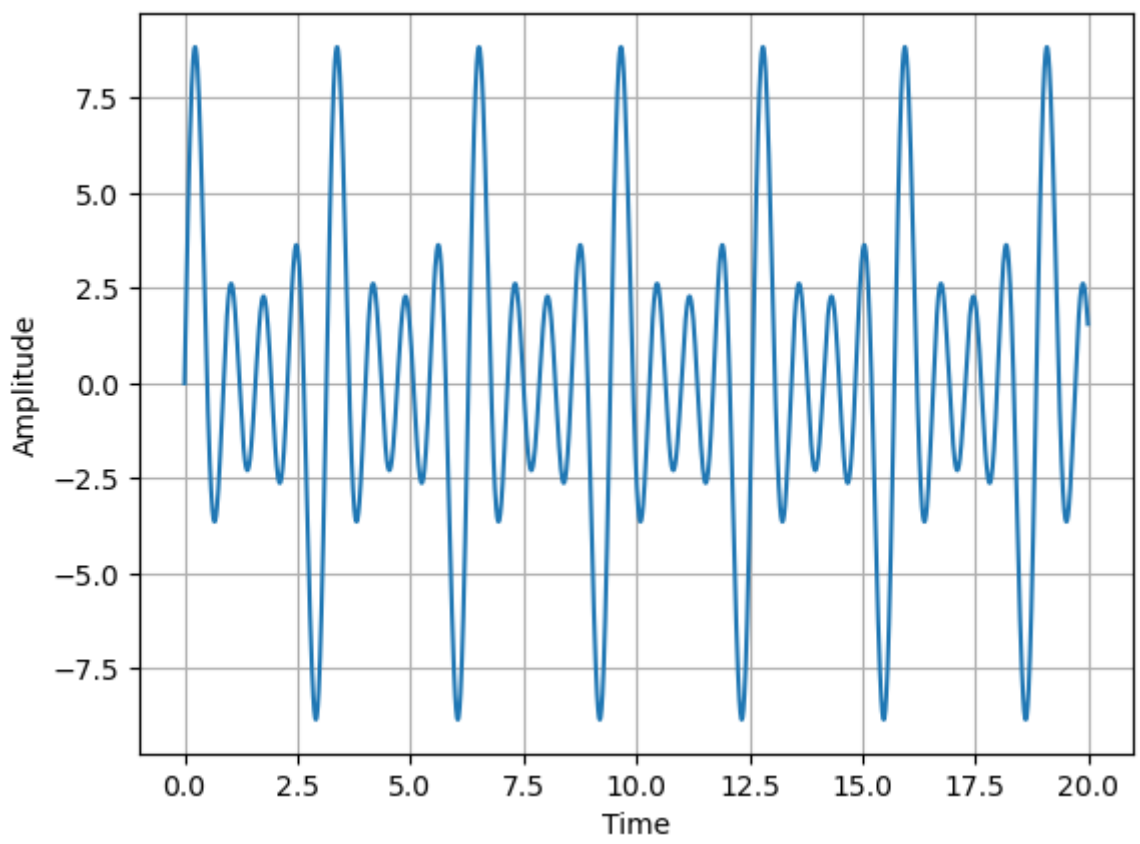
These two sounds are similar. The first sound has a noticeably lower pitch than the second, but they both have a similar tinny and high-pitched quality, reminiscent of he sound of the smoke alarm.

1c. Timbre

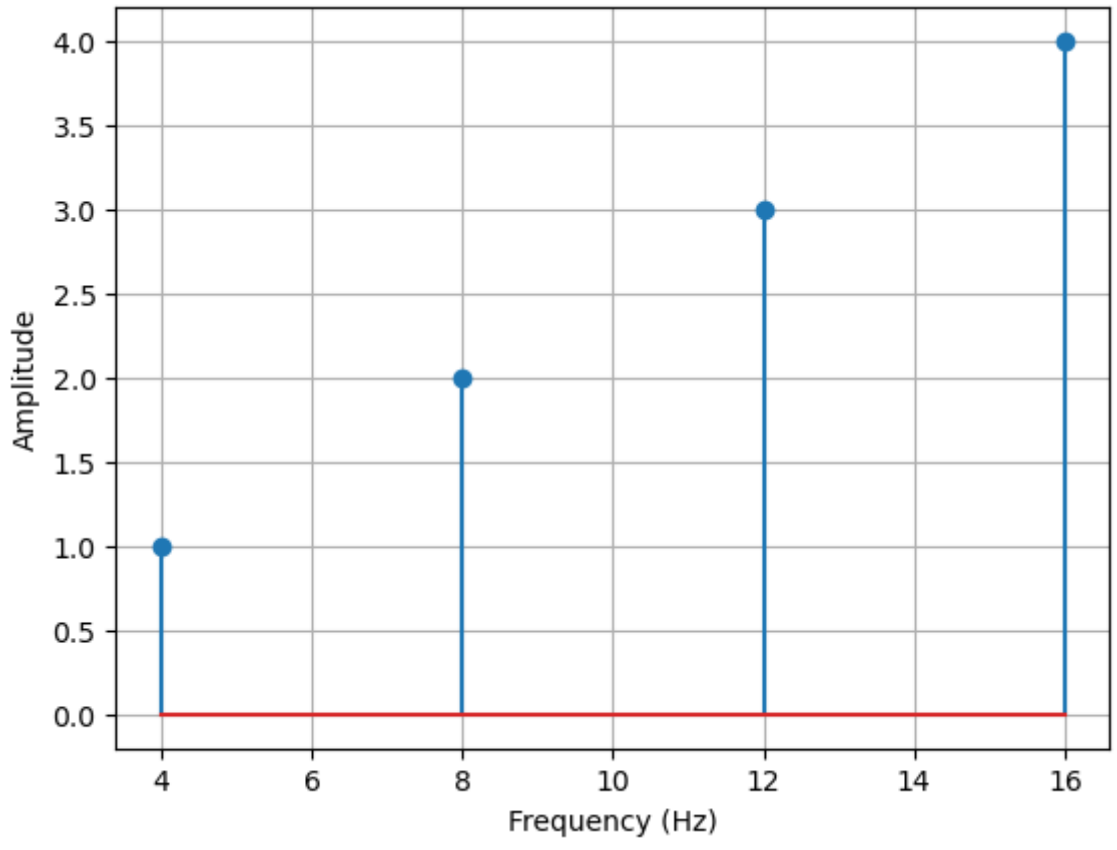
In [9]:
```
Music_t = np.arange(0,800,0.01)
Set_alist = [1,2,3,4]
Plist = [-np.pi/2,-np.pi/2,-np.pi/2,-np.pi/2]
frequence1 = 2
frequence2 = 4
A4Final.show_harmonics(t=np.arange(0,20,0.01), g=A4Final.harmonic, f=frequence1, alist=Set_alist,
                       phase_list=Plist, title="Music 1")
Data_val1 = np.array([A4Final.harmonic(t=t, f=frequence1, alist=Set_alist, phase_list=Plist) for t in Music_t])
scipy.io.wavfile.write("1c_Timbre_HarmonicMusic1.wav", rate=44100, data=Data_val1.astype(np.float32))
onec_Music1 = IPython.display.Audio("1c_Timbre_HarmonicMusic1.wav")
```



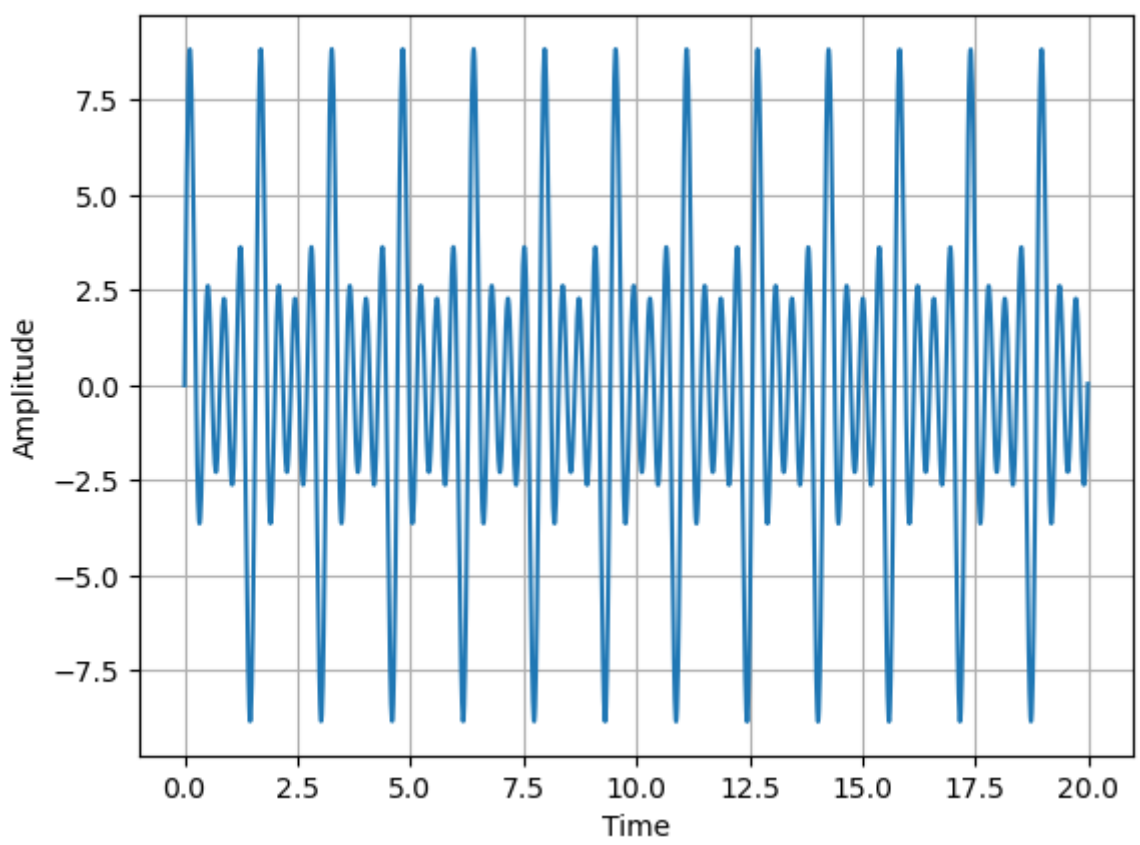
Music 1



In [10]:
```
Music_t = np.arange(0,800,0.01)
Set_alist = [1,2,3,4]
Plist = [-np.pi/2,-np.pi/2,-np.pi/2,-np.pi/2]
frequence1 = 2
frequence2 = 4
A4Final.show_harmonics(t=np.arange(0,20,0.01), g=A4Final.harmonic, f=frequence2,
                       alist=Set_alist, phase_list=Plist, title="Music 2")
Data_val2 = np.array([A4Final.harmonic(t=t, f=frequence2, alist=Set_alist, phase_list=Plist) for t in Music_t])
scipy.io.wavfile.write("1c_Timbre_HarmonicMusic2.wav", rate=44100, data=Data_val2.astype(np.float32))
onec_Music2 = IPython.display.Audio("1c_Timbre_HarmonicMusic2.wav")
```



Music 2



In [11]:
```
onec_Music1
```

Out[11]:

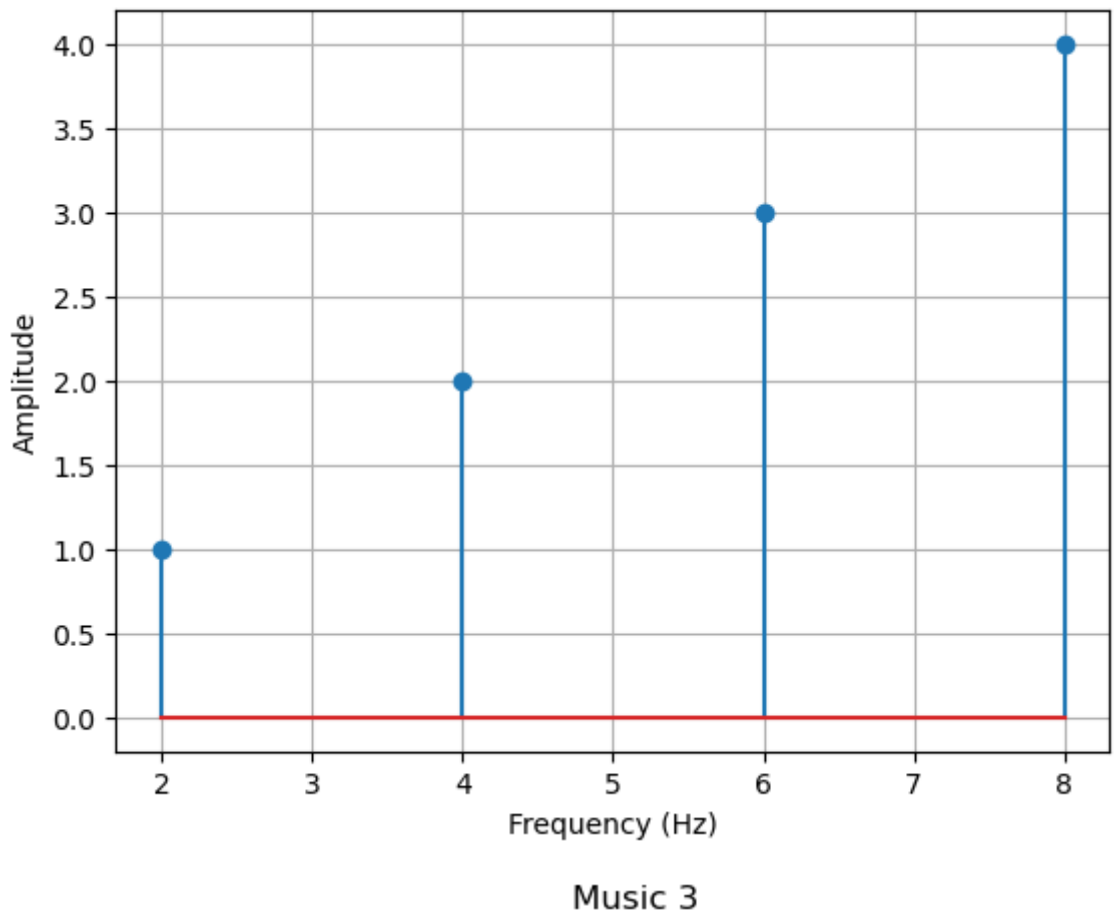
0:00 / 0:01

In [12]:
```
onec_Music2
```

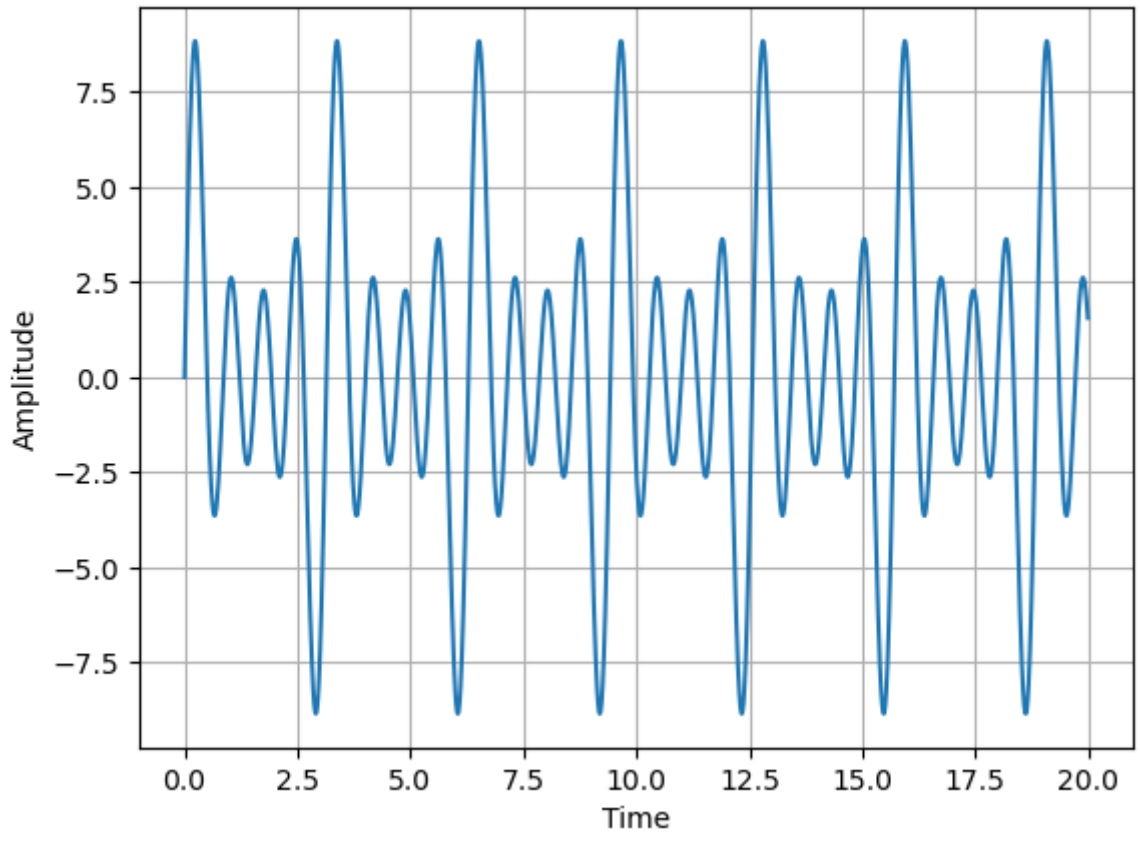
Out[12]:

0:00 / 0:01

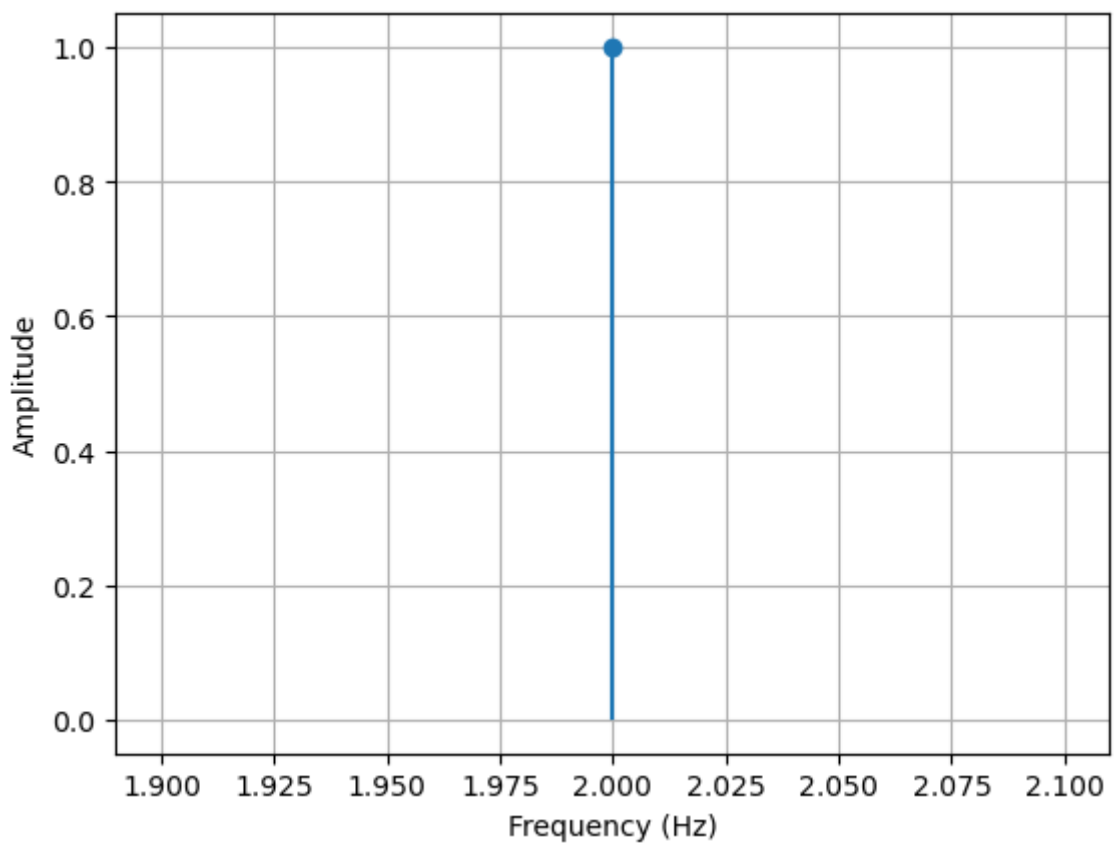
In [13]:
```
Music_t = np.arange(0,800,0.01)
Set_alist1 = [1,2,3,4]
Set_alist2 = [1]
Plist = [-np.pi/2,-np.pi/2,-np.pi/2,-np.pi/2]
frequence = 2
A4Final.show_harmonics(t=np.arange(0,20,0.01), g=A4Final.harmonic, f=frequence, alist=Set_alist1,
               phase_list=Plist, title="Music 3")
Data_val3 = np.array([A4Final.harmonic(t=t, f=frequence, alist=Set_alist1, phase_list=Plist) for t in Music_t])
scipy.io.wavfile.write("1c_Timbre_HarmonicMusic3.wav", rate=44100, data=Data_val3.astype(np.float32))
onec_Music3 = IPython.display.Audio("1c_Timbre_HarmonicMusic3.wav")
```

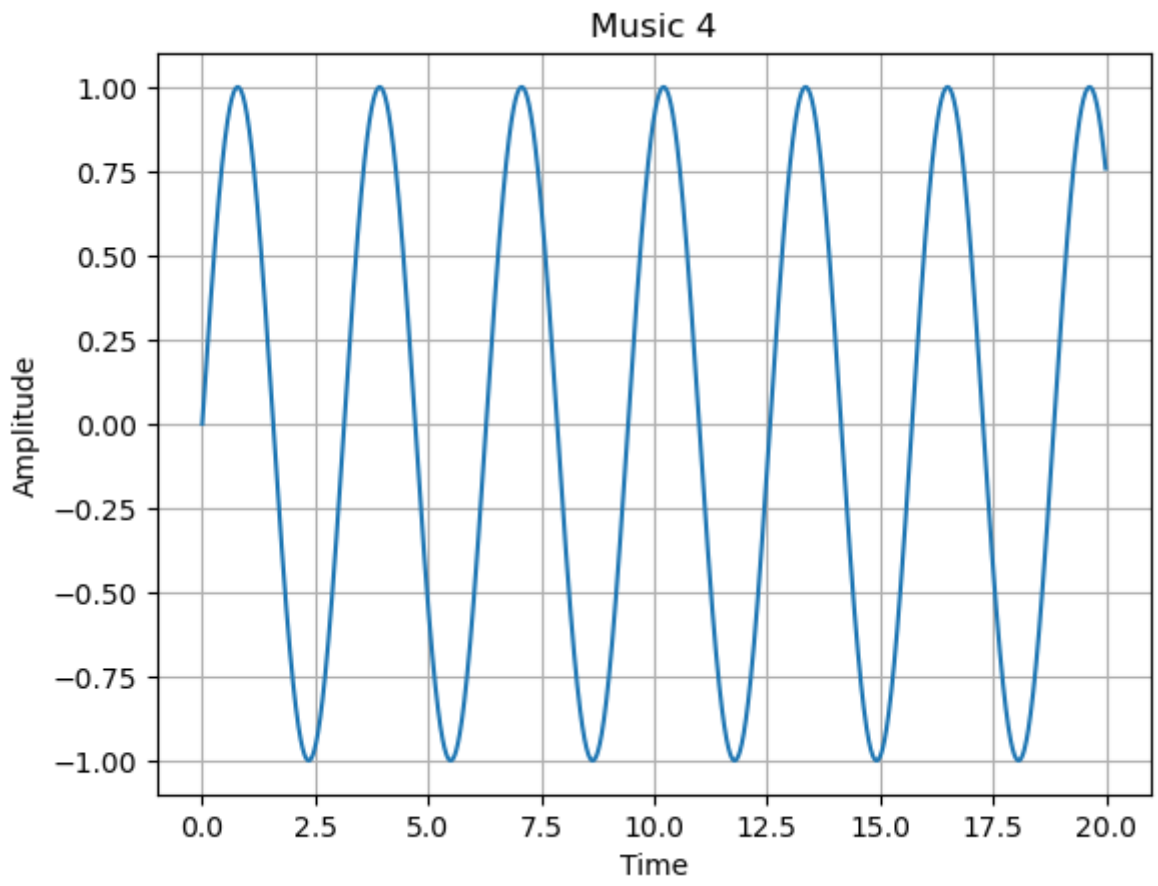


Music 3



In [14]:
```
A4Final.show_harmonics(t=np.arange(0,20,0.01), g=A4Final.harmonic, f=frequence, alist=Set_alist2,
                phase_list=Plist, title="Music 4")
Data_val4 = np.array([A4Final.harmonic(t=t, f=frequence, alist=Set_alist2, phase_list=Plist) for t in Music_t])
scipy.io.wavfile.write("1c_Timbre_HarmonicMusic4.wav", rate=44100, data=Data_val4.astype(np.float32))
onec_Music4 = IPython.display.Audio("1c_Timbre_HarmonicMusic4.wav")
```





In [15]:
```
onec_Music3
```

Out[15]: 0:00 / 0:01

In [16]:
```
onec_Music4
```

Out[16]: 0:00 / 0:01

1d. Coding Challenge (optional exploration)

In [17]:
```
import numpy as np

def make_waveform(notes, timber_func, fs=44100):
    """
    Construct a waveform from a list of notes with specified durations and a timber function for the note harmonic structure.

    Parameters:
        notes (list): A list of tuples, where each tuple contains a note's fundamental frequency in Hz and its duration in second
        timber_func (function): A function that takes a frequency in Hz and returns an array of harmonic amplitudes.
        fs (int, optional): The sampling rate of the waveform. Default is 44100 Hz.

    Returns:
        waveform (ndarray): The generated waveform as a 1D array of floats.
    """

    # calculate the total duration of the waveform
    total_duration = sum(duration for freq, duration in notes)

    # calculate the total number of samples in the waveform
    total_samples = int(total_duration * fs)

    # initialize the waveform to zeros
    waveform = np.zeros(total_samples)

    # iterate over the notes and add them to the waveform
    t = 0  # current time in seconds
    for freq, duration in notes:
        if freq == "rest":
            # if the note is a rest, skip this duration
            t += duration
            continue
        # compute the harmonic structure for the note's frequency
        harmonics = timber_func(freq)
        # compute the number of samples for this note
        note_samples = int(duration * fs)
        # compute the time vector for this note
        time = np.linspace(t, t+duration, note_samples, endpoint=False)
        # compute the waveform for this note as the sum of the harmonics
        note_waveform = np.dot(harmonics, np.ones((len(harmonics), note_samples))) / len(harmonics)
        # add the note waveform to the overall waveform
        waveform[int(t*fs):int((t+duration)*fs)] += note_waveform
        # update the current time
        t += duration

    # normalize the waveform to the range [-1, 1]
    waveform /= np.max(np.abs(waveform))
```

```
    return waveform
```

This function takes a list of tuples notes, where each tuple contains a note's fundamental frequency in Hz and its duration in seconds, and a timber_func that returns an array of harmonic amplitudes for a given frequency. It also takes an optional argument fs for the sampling rate of the waveform, which defaults to 44100 Hz.

The function computes the total duration and total number of samples in the waveform, initializes the waveform to zeros, and iterates over the notes. If the note is a rest, it skips the current duration and updates the time. Otherwise, it calls timber_func to compute the harmonic structure for the note's frequency, computes the number of samples and time vector for the note, computes the waveform for the note as the sum of the harmonics, and adds it to the overall waveform. It then updates the time and repeats for the next note.

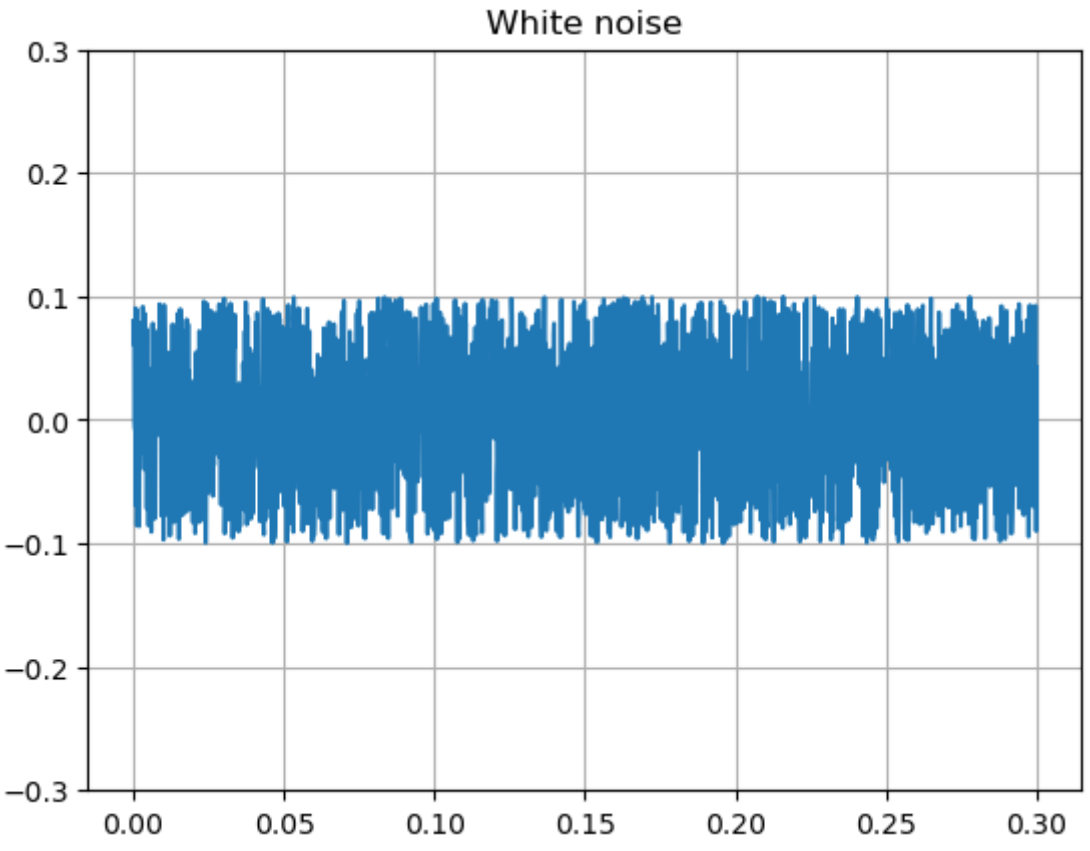
Finally, the function normalizes the waveform to the range [-1, 1] and returns it as a 1D array of floats.

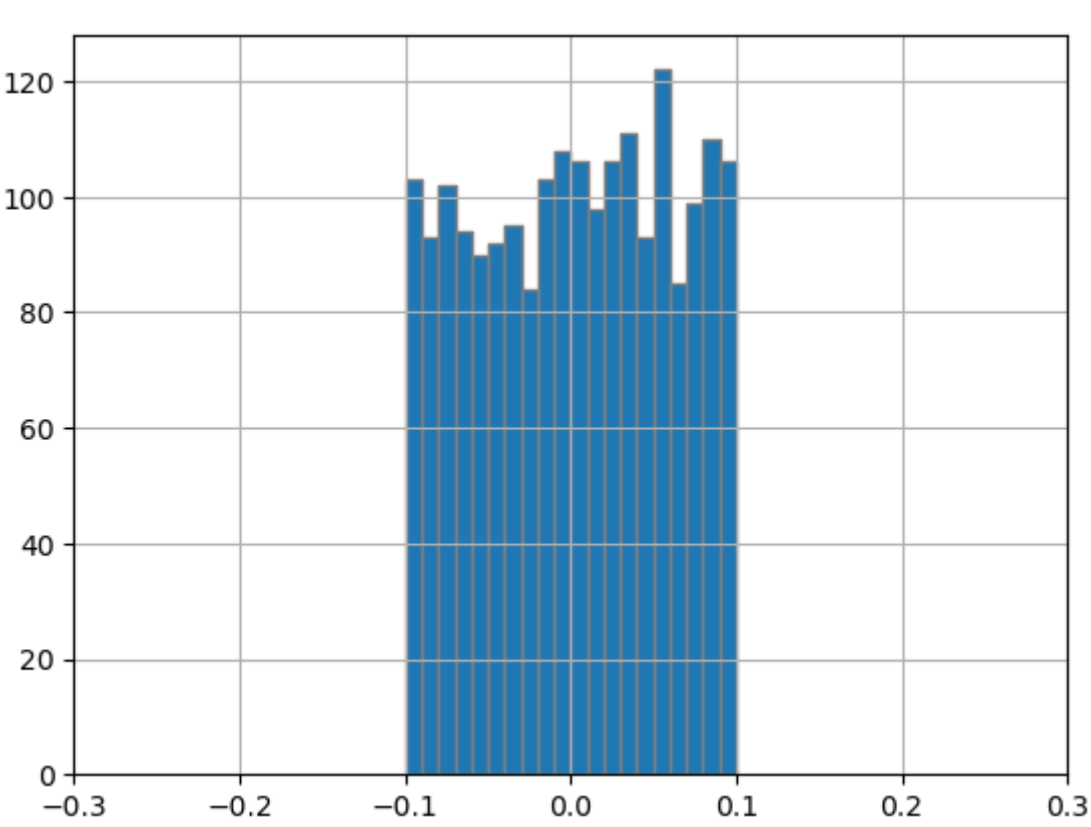
Note that this implementation assumes that the harmonic structure for each note is time-invariant, which may not be the case for some instruments. Also note that this implementation does not handle notes specified in terms of musical notation or intervals in a key.

2a. White noise

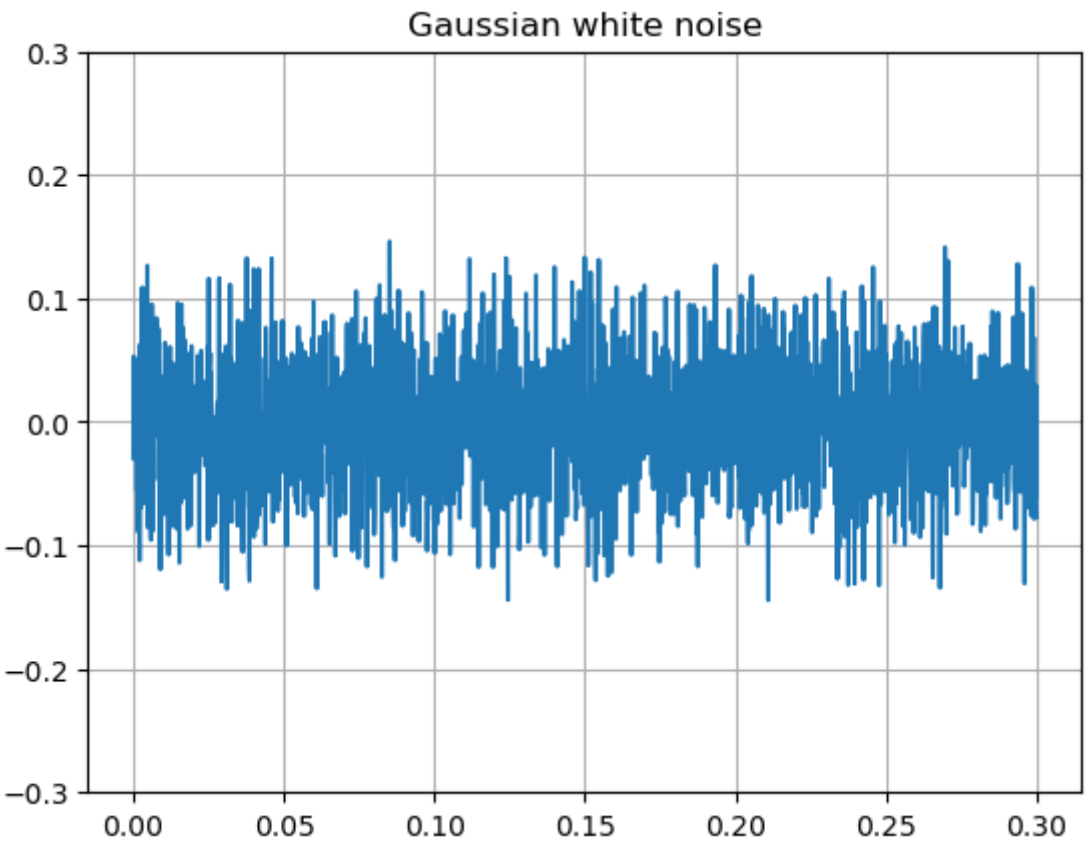
In [18]:
```
import numpy as np
import scipy.io.wavfile
import matplotlib.pyplot as plt

def generate_white_noise_file(filename = 'WhiteNoise_2a.wav',
                              duration = 0.3, amplitude=0.1, num_samples = 2000):
    x = np.linspace(0, duration, num_samples)
    y = np.random.uniform(-amplitude, amplitude, len(x))
    noise = np.array(y)
    scipy.io.wavfile.write(filename, 1000, noise)
    plt.ylim(-duration, duration)
    plt.grid()
    plt.title('White noise')
    plt.plot(x, y)
    plt.figure()
    plt.xlim(-duration, duration)
    plt.grid()
    plt.hist(y, bins=20, edgecolor="grey")
    plt.show()

def generate_Gaussian_WhiteNoise_file(filename = 'GaussianWhiteNoise_2a.wav',
                                      duration = 0.3, amplitude=0.1, num_samples = 2000):
    x = np.linspace(0, duration, num_samples)
    y = np.random.normal(0, amplitude/2, len(x))
    noise = np.array(y)
    scipy.io.wavfile.write(filename, 1000, noise)
    plt.ylim(-duration, duration)
    plt.grid()
    plt.title('Gaussian white noise')
    plt.plot(x, y)
    plt.figure()
    plt.xlim(-duration, duration)
    plt.grid()
    plt.hist(y, bins=20, edgecolor="grey")
    plt.show()

def generate_SparseNoise_file(filename = 'SparseNoise_2a.wav',
                                      duration = 0.3, amplitude=0.1, num_samples = 2000):
    x = np.linspace(0, duration, num_samples)
    y = np.random.laplace(0, amplitude/2, len(x))
    noise = np.array(y)
    scipy.io.wavfile.write(filename, 1000, noise)
    plt.ylim(-duration, duration)
    plt.grid()
    plt.title('Sparse noise')
    plt.plot(x, y)
    plt.figure()
    plt.xlim(-duration, duration)
    plt.grid()
    plt.hist(y, bins=20, edgecolor="grey")
    plt.show()
```

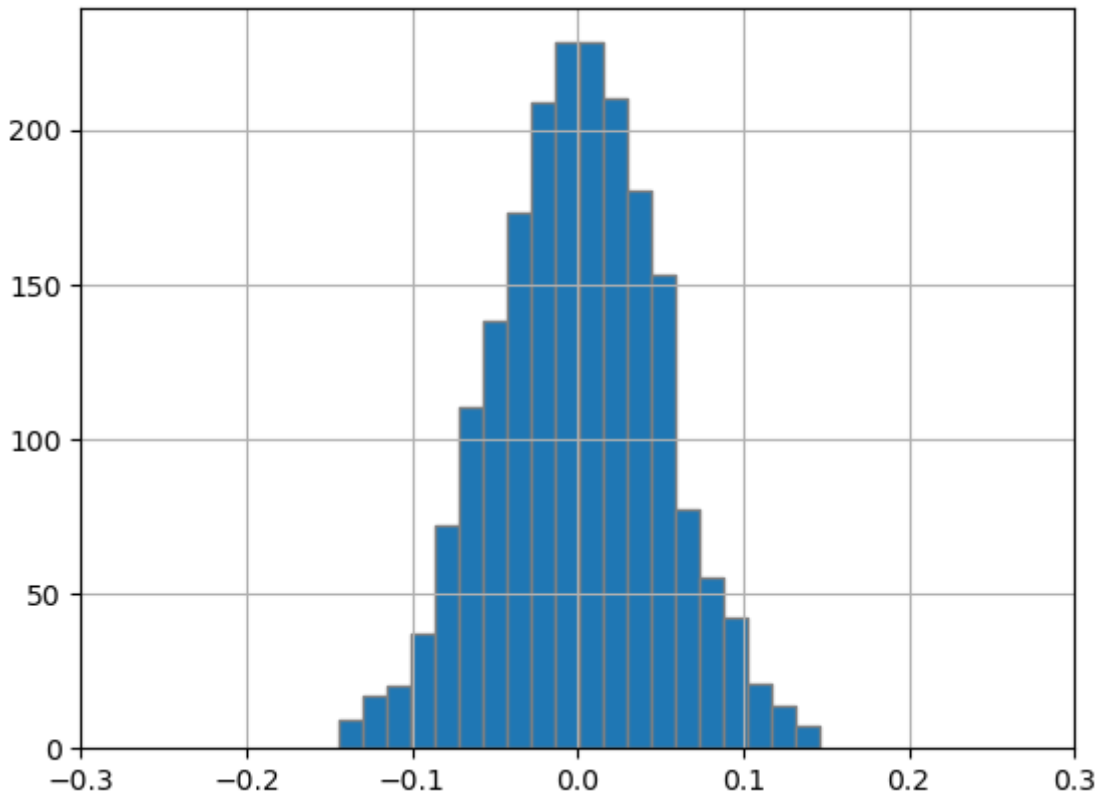
In [19]:
```
generate_white_noise_file(filename = 'WhiteNoise_2a.wav',
                          duration = 0.3, amplitude=0.1, num_samples = 2000)
```



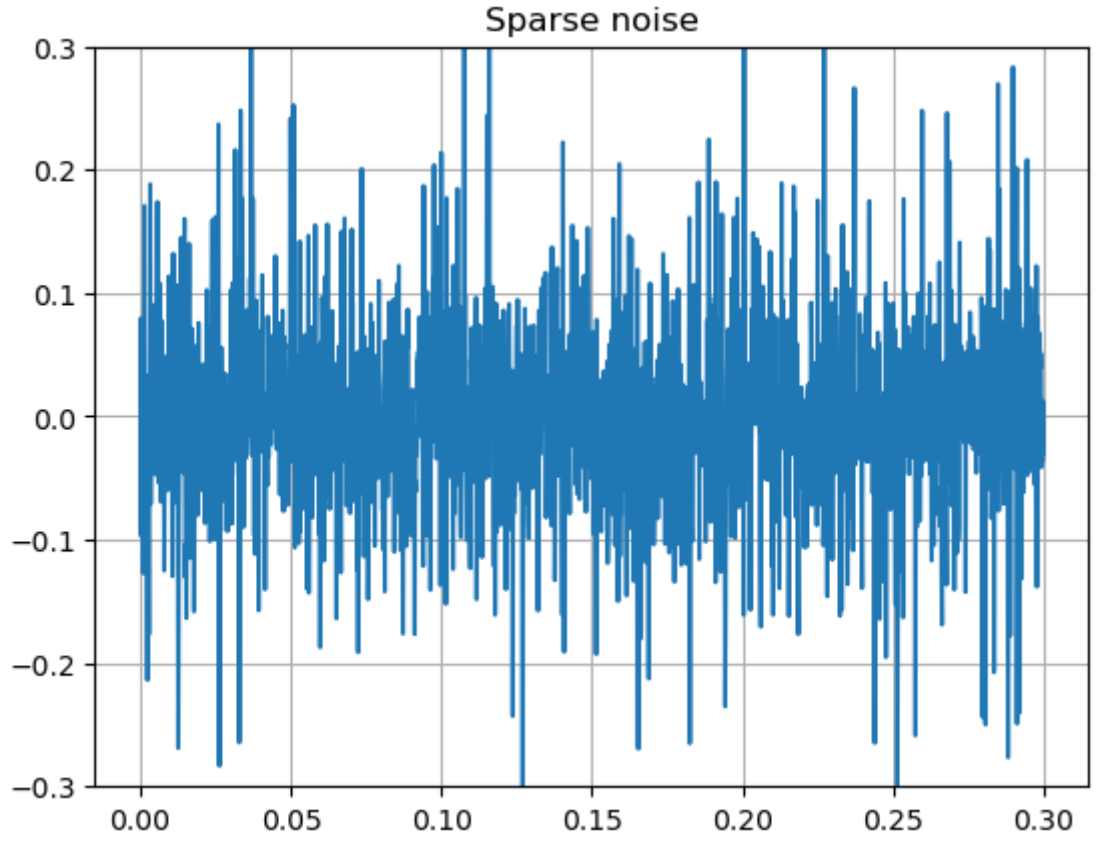


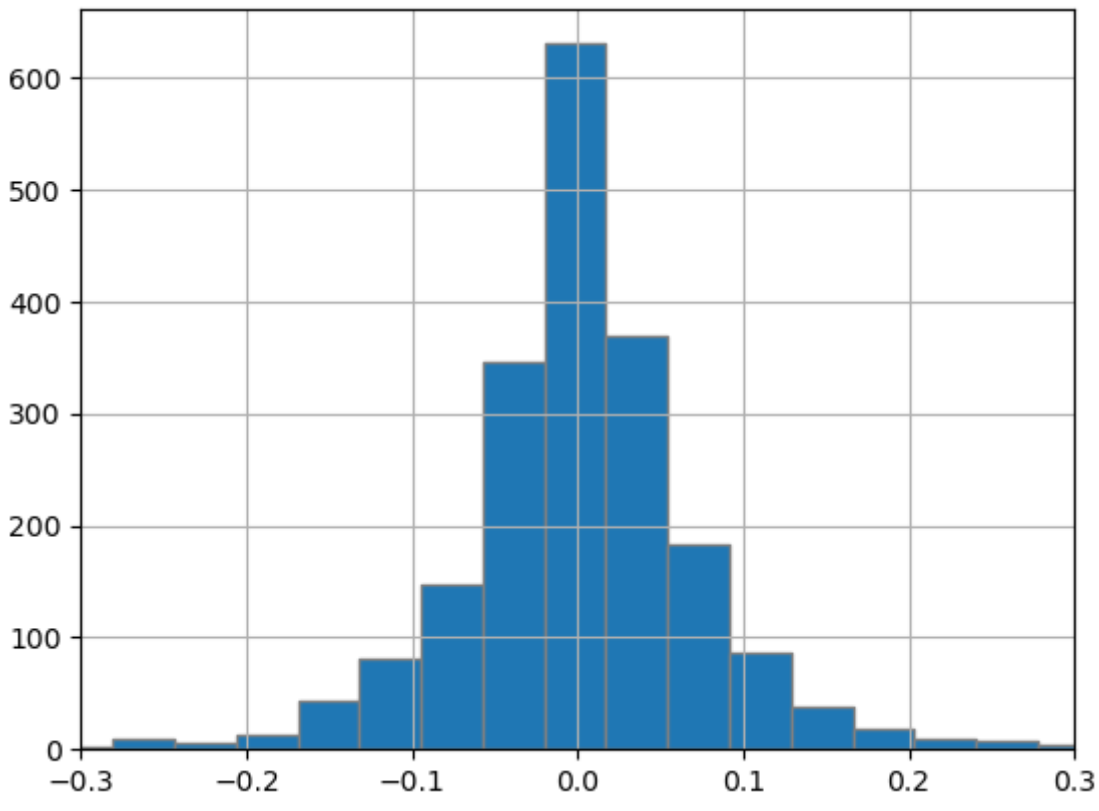
In [20]:
```
generate_Gaussian_WhiteNoise_file(filename = 'GaussianWhiteNoise_2a.wav',
                                  duration = 0.3, amplitude=0.1, num_samples = 2000)
```





In [21]:
```
generate_SparseNoise_file(filename = 'SparseNoise_2a.wav',
                                    duration = 0.3, amplitude=0.1, num_samples = 2000)
```





Conclusion

The histogram shows that a normal distribution exhibits the well-known bell curve shape, while a Laplace distribution, also known as a double exponential distribution, has a peaked shape in the middle, resembling a pole holding up a circus tent.

In contrast to uniform and Gaussian white noise, the differences in sound cannot be perceived by the human ear for these types of noise. However, this is not the case for Laplace noise, as while it technically has a white spectrum and uncorrelated samples, it does not exhibit locality.

This is because the auditory system only encodes sound frequency over a short period of time.

To synthesize white noise with a sparse distribution, we can generate samples from a Laplacian or generalized Gaussian distribution instead of a Gaussian distribution.

Let's consider the Laplacian distribution first. The probability density function of the Laplacian distribution is given by:

$$f(x; \mu, b) = 1/(2b) * exp(-|x - \mu|/b)$$

where μ is the location parameter and b > 0 is the scale parameter. The Laplacian distribution is sometimes called the "double exponential" distribution because of its shape.

Now, to generate white noise with a Laplacian distribution, we can simply generate random numbers from this distribution using a random number generator in Python. For example:

In [22]:
```
import numpy as np

mu, b = 0, 1 # parameters of the Laplacian distribution
n_samples = 10000 # number of samples

white_noise = np.random.laplace(mu, b, n_samples)
```

We can also generate white noise with a generalized Gaussian distribution, which is a family of probability distributions that includes both the Gaussian and Laplacian distributions as special cases. The probability density function of the generalized Gaussian distribution is given by:

$$f(x; \mu, b, p) = (p/(2b * \Gamma(1/p))) * exp(-(abs(x - \mu)/b)^p)$$

where μ is the location parameter, b > 0 is the scale parameter, and p > 0 is the shape parameter. Γ() denotes the gamma function.

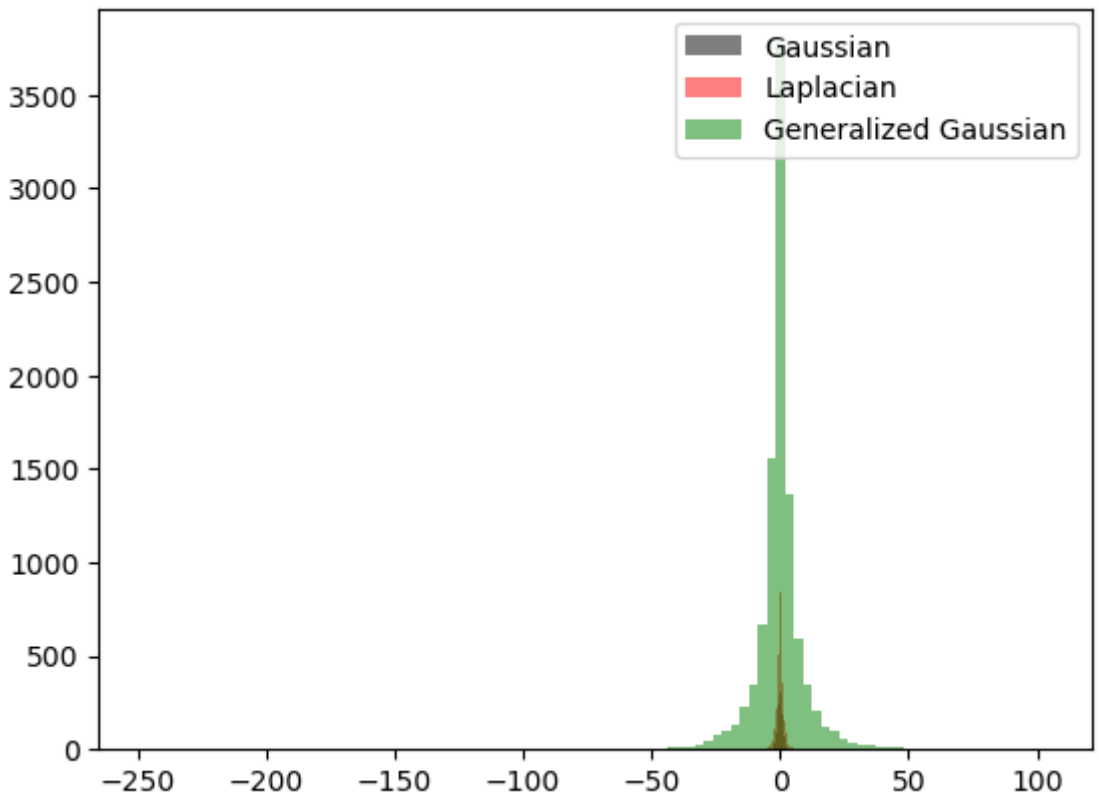
To generate white noise with a generalized Gaussian distribution, we can use a random number generator that supports this distribution. For example, in Python we can use the scipy.stats.gennorm function:

In [23]:
```
import numpy as np
from scipy.stats import gennorm

mu, b, p = 0, 1, 0.5 # parameters of the generalized Gaussian distribution
n_samples = 10000 # number of samples

white_noise = gennorm.rvs(p, loc=mu, scale=b, size=n_samples)
```

Now, let's compare the histograms of white noise generated with a Gaussian distribution, a Laplacian distribution, and a generalized Gaussian distribution:

The resulting plot shows the histograms of the three types of white noise:

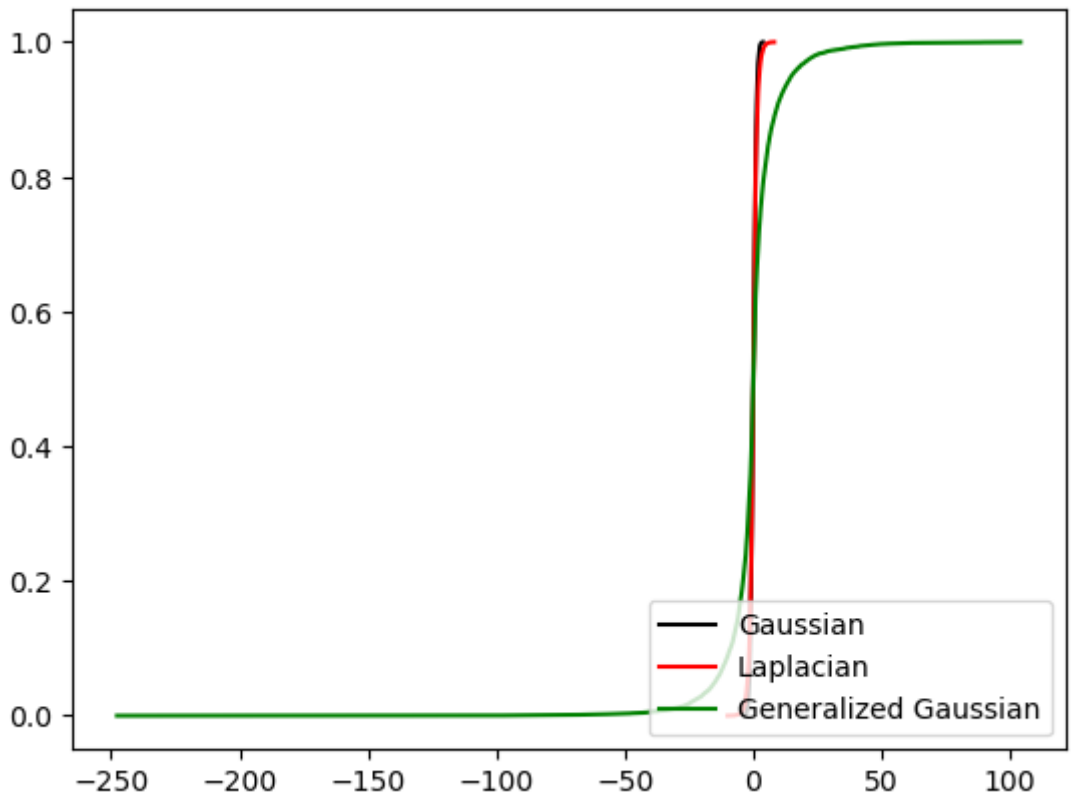
In [24]:
```
import matplotlib.pyplot as plt

# Generate white noise with a Gaussian distribution
gaussian_noise = np.random.normal(0, 1, 10000)

# Generate white noise with a Laplacian distribution
laplacian_noise = np.random.laplace(0, 1, 10000)

# Generate white noise with a generalized Gaussian distribution
gengaussian_noise = gennorm.rvs(0.5, loc=0, scale=1, size=10000)

# Plot histograms of the three types of noise
plt.hist(gaussian_noise, bins=100, alpha=0.5, label='Gaussian', color = 'black')
plt.hist(laplacian_noise, bins=100, alpha=0.5, label='Laplacian', color = 'r')
plt.hist(gengaussian_noise, bins=100, alpha=0.5, label='Generalized Gaussian', color = 'g')
plt.legend(loc='upper right')
plt.show()
```



As we can see, the histograms of the Laplacian and generalized Gaussian noise have heavier tails and more extreme values than the histogram of the Gaussian noise. This is because the Laplacian and generalized Gaussian distributions have heavier tails than the Gaussian distribution, meaning that they allow for more extreme values. Therefore, the white noise generated with a sparse distribution will be discernibly different from white noise generated with a Gaussian distribution, as it will have a different statistical structure, with more frequent and extreme deviations from the mean.

One way to visualize the differences between the three types of noise is to plot the cumulative distribution function (CDF) of each noise type. The CDF shows the probability that a random variable takes on a value less than or equal to a certain threshold. Here's how we can plot the CDFs of the three types of noise:

In [25]:
```
# Plot CDFs of the three types of noise
plt.plot(np.sort(gaussian_noise), np.linspace(0, 1, len(gaussian_noise)),
         label='Gaussian', color = 'black')
plt.plot(np.sort(laplacian_noise), np.linspace(0, 1, len(laplacian_noise)),
         label='Laplacian',color = 'r')
plt.plot(np.sort(gengaussian_noise), np.linspace(0, 1, len(gengaussian_noise)),
         label='Generalized Gaussian', color ='g')
plt.legend(loc='lower right')
plt.show()
```



As we can see, the CDFs of the Laplacian and generalized Gaussian noise are steeper than the CDF of the Gaussian noise, especially in the tails. This means that extreme values occur more frequently in the Laplacian and generalized Gaussian noise than in the Gaussian noise.

In summary, white noise generated with a sparse distribution, such as a Laplacian or generalized Gaussian distribution, will be discernibly different from white noise generated with a Gaussian distribution, as it will have a different statistical structure with more frequent and extreme deviations from the mean. This can be visualized through the histograms and cumulative distribution functions of the noise types.

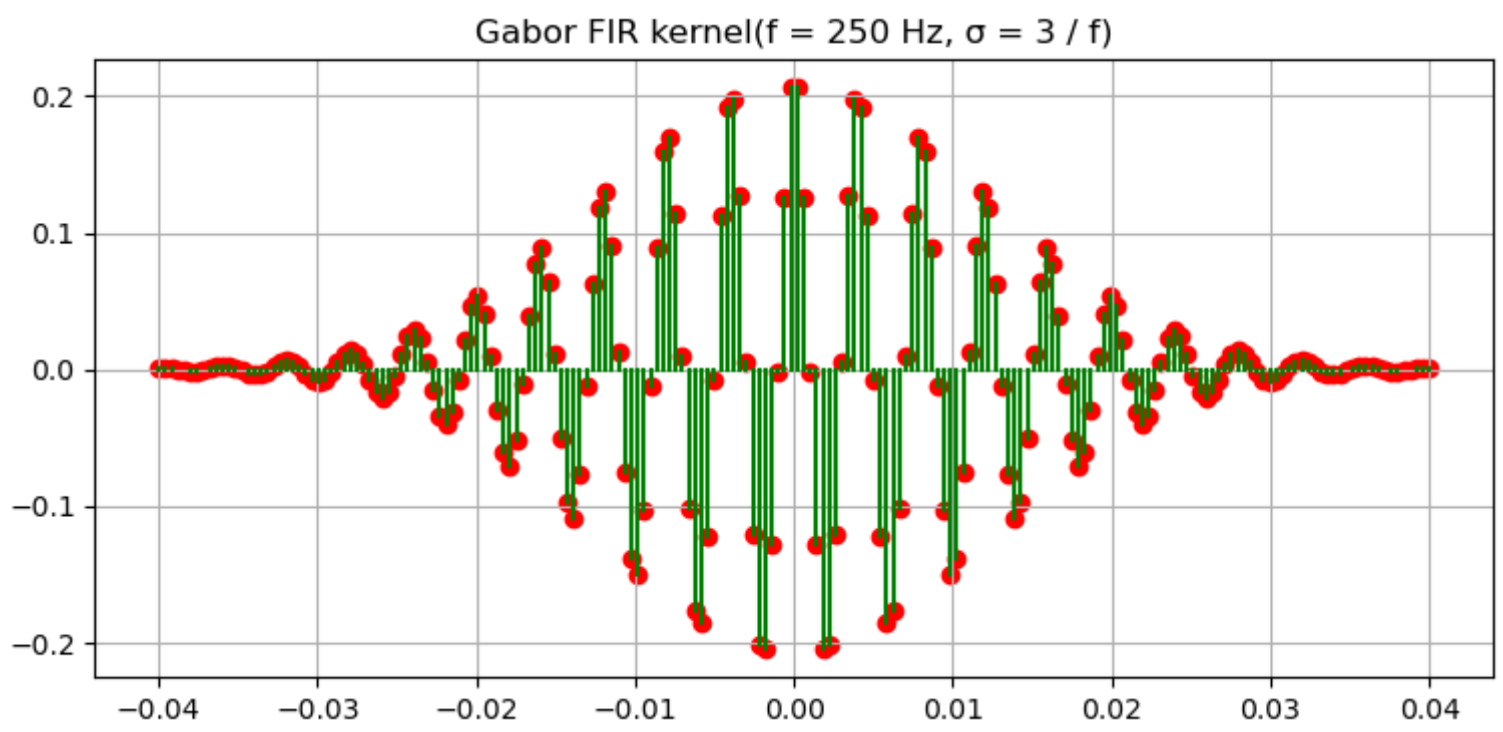
2b. Bandpass noise

In [155…
```
import librosa
import librosa.display

def gabor_a(t,fs,sigma=1.0,f=1.0,phi=0.0):
```

```
    return gabor(t,sigma,f,phi)/ gabor_norm(fs,sigma,f,phi)
def gabor(t,sigma,f,phi):#calcuate g without normalizing constant
    return math.exp(-pow(t,2)/(2*pow(sigma,2)))*math.cos(2*math.pi*f*(t)+phi)
def gabor_norm(fs,sigma=1.0,f=1.0,phi=0.0):#calcuate normalizing constant
    sum_g=[]
    for index in range(-fs,fs):
        t=(index)/fs
        sum_g.append(gabor(t,sigma,f,phi))
    return np.linalg.norm(sum_g)
def convolve(x,h=[1],h0=1):
    y=[]
    for i in range(len(x)):
        yn=0
        if h0==1:
            for j in range(0,i+1):
                if (i-j)>(len(h)-1):
                    hij=0
                else:
                    hij=h[i-j]
                yn+=x[j]*hij
            y.append(yn)
        else:
            for j in range(0,i+1):
                if abs(i-j)>(len(h)-1):
                    hij=0
                elif i-j<0:
                    hij=h[j-i]
                else:
                    hij=h[i-j]
                yn+=x[j]*hij
            y.append(yn)
    return np.array(y)
```

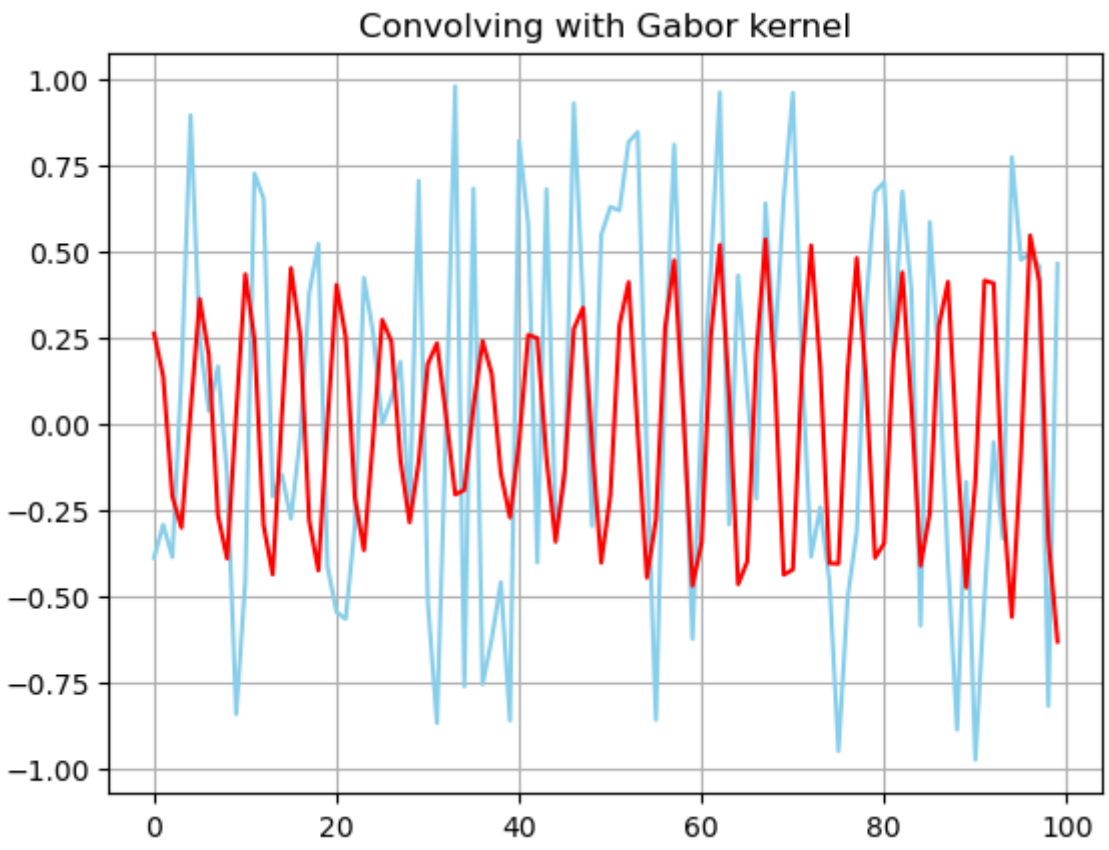
In [27]:

```
fs = 2000
f = 250
sigma = 3 / f
p = np.linspace(-0.04,0.04,200)
q = np.vectorize(gabor_a)(p, fs = fs, sigma = 3/f, f = f)
plt.figure(figsize = (9, 4))
for j in range(len(q)):
    plt.scatter(p[j],q[j], color='r')
    plt.plot([p[j],p[j]],[0,q[j]],color='g')
plt.title("Gabor FIR kernel(f = 250 Hz, σ = 3 / f)")
plt.grid()
plt.show()
```



In [28]:

```
p = np.linspace(-0.04,0.04,100)
q = np.vectorize(gabor_a)(p, fs = fs, sigma = 3/f, f = f)
h_value=q.copy()
The_index=np.where(np.array(h_value)==max(h_value))
noise_value=np.random.uniform(-1,1,200)
q_new=convolve(noise_value,h_value,h0=The_index)
plt.plot(noise_value[100:200], color = "skyblue")
plt.plot(q_new[100:200], color = "r")
plt.title("Convolving with Gabor kernel")
plt.grid()
plt.show()
```



It is an implementation of a function bandpass_noise() that uses the Gabor function to synthesize bandpass noise with a specified center frequency and bandwidth:

In [29]:
```python
import numpy as np

def bandpass_noise(duration, fs, cf, bw):
    # duration: duration of noise signal in seconds
    # fs: sampling rate in Hz
    # cf: center frequency of bandpass filter in Hz
    # bw: bandwidth of bandpass filter in Hz

    t = np.arange(0, duration, 1/fs)  # time vector
    n = np.random.randn(len(t))  # white noise signal
    g = np.exp(-((t - duration/2) ** 2) / (2 * (bw/(2*np.pi)) ** 2))  # Gaussian envelope
    bpf = g * np.cos(2 * np.pi * cf * t)  # bandpass filter
    bpn = np.convolve(n, bpf, mode='same')  # bandpass noise signal

    return bpn
```

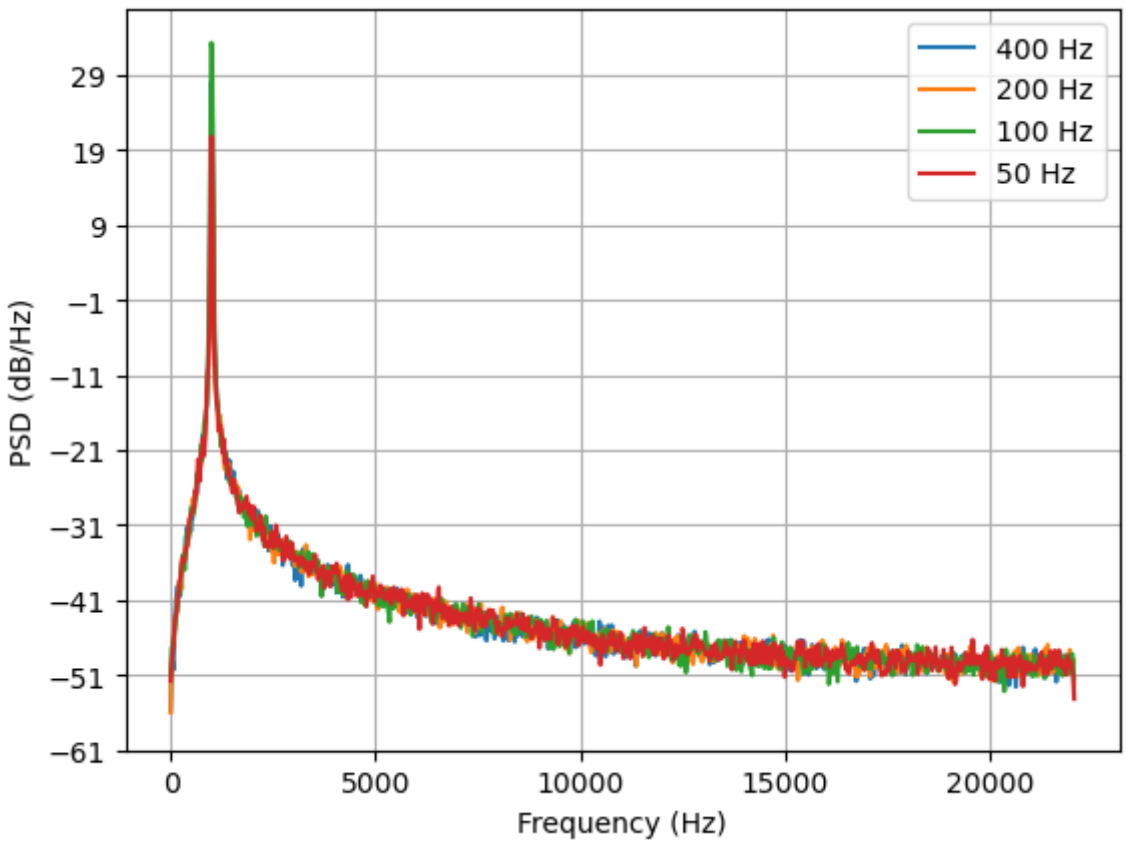
The function takes four input parameters:

1. duration: the duration of the synthesized noise signal in seconds
2. fs: the sampling rate in Hz
3. cf: the center frequency of the bandpass filter in Hz
4. bw: the bandwidth of the bandpass filter in Hz

The function first generates a white noise signal with the same duration and sampling rate as specified by the duration and fs parameters, respectively. It then creates a Gaussian envelope centered at the middle of the time vector t with a standard deviation proportional to the bandwidth of the bandpass filter. The envelope is multiplied with a cosine wave at the specified center frequency cf to create the bandpass filter. Finally, the function convolves the white noise signal with the bandpass filter to obtain the bandpass noise signal.

To illustrate the inverse relation between Gabor width and noise bandwidth, we can generate several bandpass noise signals with different Gabor widths (corresponding to different bandwidths) while keeping the center frequency fixed, and plot their power spectral densities (PSDs) using the plt.psd function from matplotlib. Here's an example:

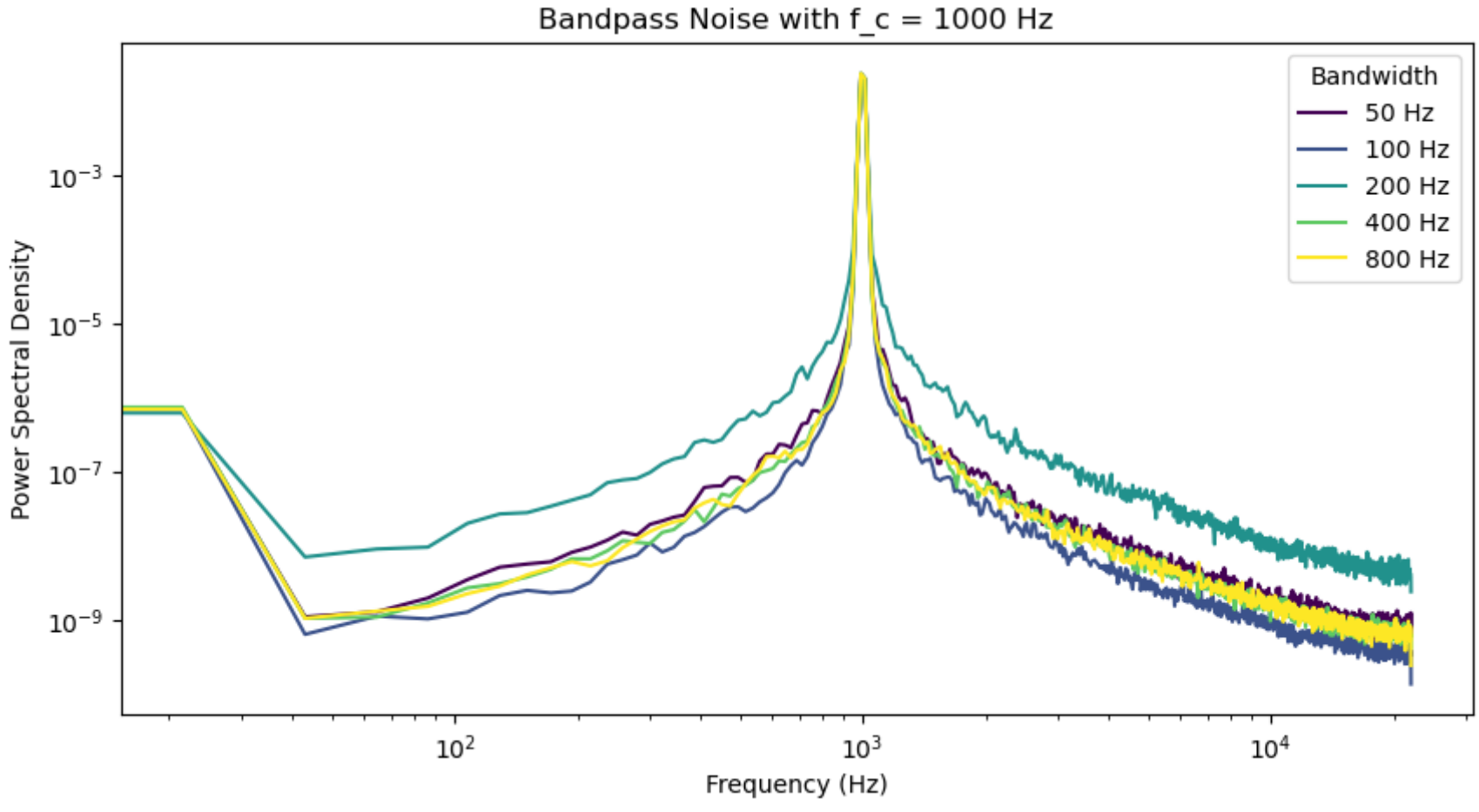
In [30]:
```python
import matplotlib.pyplot as plt

# Set parameters
duration = 1  # seconds
fs = 44100  # Hz
cf = 1000  # Hz
bandwidths = [400, 200, 100, 50]  # Hz

# Generate bandpass noise signals with different bandwidths
signals = []
for bw in bandwidths:
    signal = bandpass_noise(duration, fs, cf, bw)
    signals.append(signal)

# Compute and plot power spectral densities
fig, ax = plt.subplots()
for i, signal in enumerate(signals):
    _, psd = plt.psd(signal, Fs=fs, NFFT=2048, color=f'C{i}', label=f'{bandwidths[i]} Hz')
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel('PSD (dB/Hz)')
ax.legend()
plt.show()
```



This code generates four bandpass noise signals with bandwidths of 400 Hz, 200 Hz, 100 Hz, and 50 Hz, and plots their PSDs on a logarithmic frequency axis. The resulting plot shows the power spectral density of each signal as a function of frequency, with different colors indicating different bandwidths. As expected, the narrower the bandwidth (i.e., the larger the Gabor width), the higher the power spectral density around the center frequency of 1000 Hz, confirming the inverse relation between Gabor width and noise bandwidth.

In [31]:
```python
import numpy as np
from scipy.signal import gaussian, convolve

def bandpass_noise(duration, sample_rate, f_c, bandwidth):
    """
    Generate bandpass noise using a Gabor filter.
    Parameters:
    duration (float): The duration of the noise signal in seconds.
    sample_rate (int): The sampling rate of the noise signal in Hz.
    f_c (float): The center frequency of the bandpass filter in Hz.
    bandwidth (float): The bandwidth of the bandpass filter in Hz.

    Returns:
    np.ndarray: A 1D NumPy array containing the generated noise signal.
    """
    # Calculate the number of samples in the noise signal.
    n_samples = int(duration * sample_rate)

    # Generate white noise.
    noise = np.random.randn(n_samples)

    # Create the Gabor filter.
    t = np.linspace(-duration / 2, duration / 2, n_samples)
    sigma = bandwidth / (2 * np.sqrt(2 * np.log(2)))
    gabor = np.exp(-(t ** 2) / (2 * sigma ** 2)) * np.cos(2 * np.pi * f_c * t)

    # Convolve the white noise with the Gabor filter.
    filtered_noise = convolve(noise, gabor, mode='same')

    # Normalize the filtered noise to have zero mean and unit variance.
    filtered_noise = (filtered_noise - np.mean(filtered_noise)) / np.std(filtered_noise)

    return filtered_noise
```

In [32]:
```python
import matplotlib.pyplot as plt
# Generate bandpass noise with different values of bandwidth.
f_c = 1000  # Hz
durations = [1]  # seconds
sample_rate = 44100  # Hz
bandwidths = [50, 100, 200, 400, 800]  # Hz
noises = []
for bandwidth in bandwidths:
    noise = bandpass_noise(durations[0], sample_rate, f_c, bandwidth)
    noises.append(noise)

# Calculate and plot the power spectra of the bandpass noise.
fig, ax = plt.subplots(figsize=(10, 5))
colors = plt.cm.viridis(np.linspace(0, 1, len(noises)))
for noise, bandwidth, color in zip(noises, bandwidths, colors):
    freqs, psd = scipy.signal.welch(noise, sample_rate, nperseg=2048)
    ax.plot(freqs, psd, label=f'{bandwidth} Hz', color=color)
ax.set_xscale('log')
ax.set_yscale('log')
ax.set_xlabel('Frequency (Hz)')
ax.set_ylabel('Power Spectral Density')
ax.set_title(f'Bandpass Noise with f_c = {f_c} Hz')
ax.legend(title='Bandwidth')
plt.show()
```



This code generates bandpass noise with f_c = 1000 Hz and different values of bandwidth (50 Hz, 100 Hz, 200 Hz, 400 Hz, and 800 Hz), and calculates and plots the power spectra of each noise signal using Welch's method. The resulting plot shows that as the bandwidth of the bandpass filter increases (i.e., as the Gabor width decreases), the power spectrum of the resulting noise signal also increases, indicating that the noise

3a. Auto-correlation vs self-convolution

Mathematical argument:

Mathematically, we can show that auto-correlation is symmetric around zero while self-convolution is not by considering their definitions:

$$R_{xx}[n] = x[n] \star x[n] = (x \star x)[n] = \sum_k x[k-n]x[k]$$

$$x[n] \star x[n] = \sum_k x[k-n]x[k-n]$$

To show that $R_{xx}[-n] = R_{xx}[n]$ (i.e., that the auto-correlation is symmetric around zero), we simply change the variable $k \rightarrow j+n$ in the first equation:

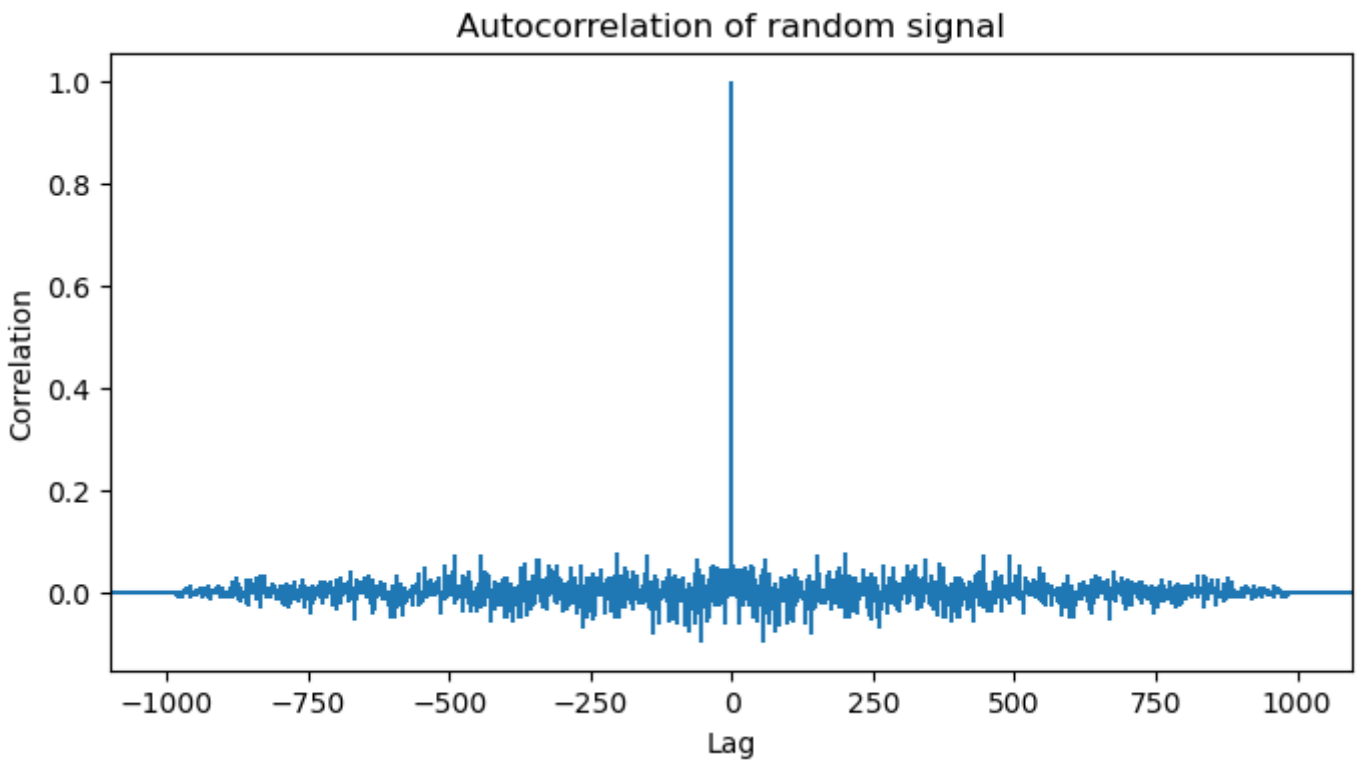
$$R_{xx}[-n] = \sum_j x[j-n]x[j] = \sum_k x[k]x[k+n] = \sum_k x[k+n]x[k] = \sum_j x[j]x[j-n] = R_{xx}[n]$$

For self-convolution, we have:

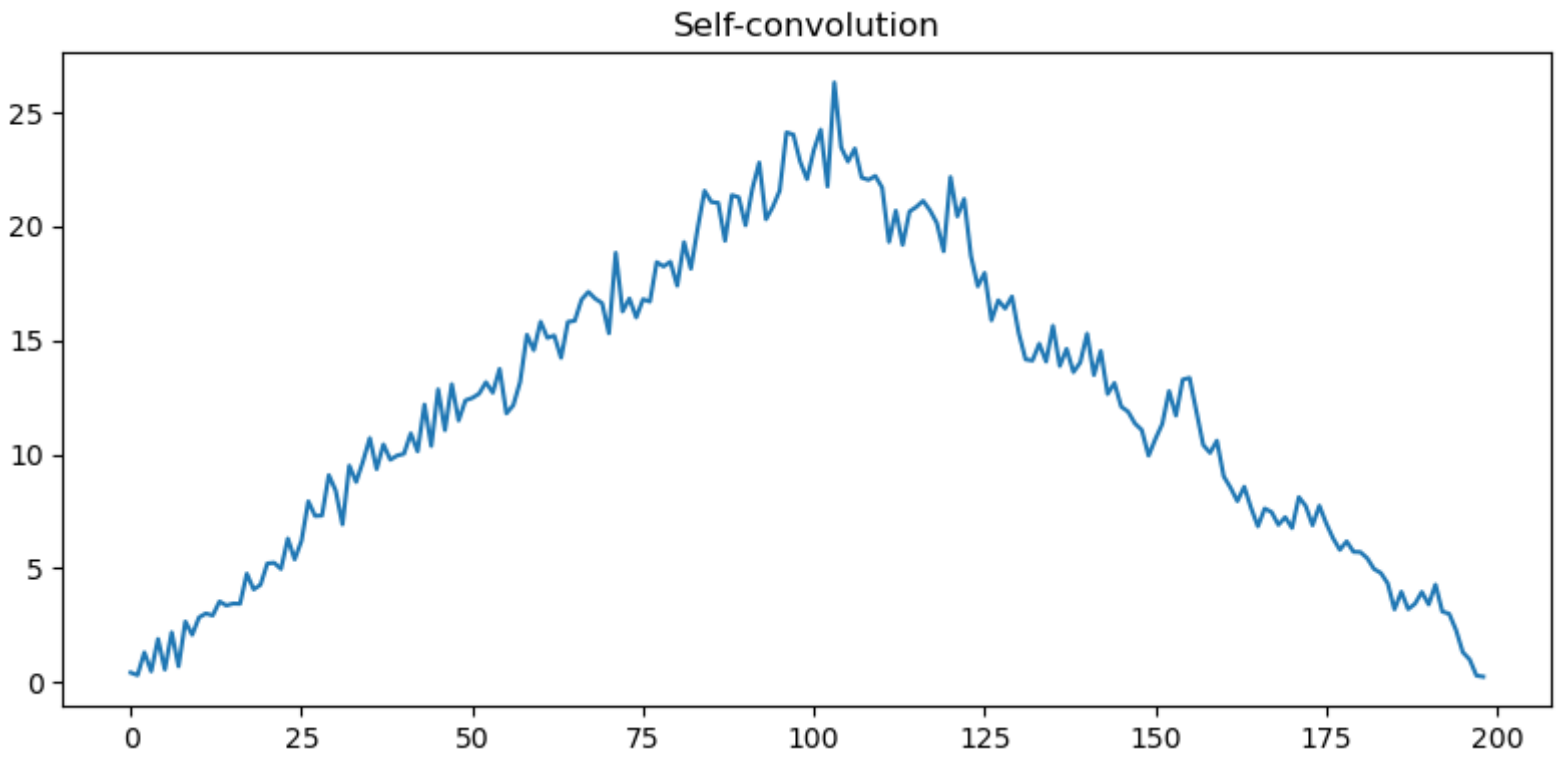
$$x[n] \star x[n] = \sum_k x[k-n]x[k-n] = \sum_j x[j]x[j-2n]$$

which is not symmetric around zero unless $n = 0$. This is because the range of $j-2n$ does not include all values of j for non-zero values of n.

In [33]:
```python
import matplotlib.pyplot as plt
import numpy as np

x = np.random.randn(1000) # generate random signal

fig, ax = plt.subplots(figsize=(8,4))

ax.acorr(x, maxlags=len(x)-1) # plot autocorrelation

ax.set_title('Autocorrelation of random signal')
ax.set_xlabel('Lag')
ax.set_ylabel('Correlation')

plt.show()
```



In [34]:
```python
import numpy as np
import matplotlib.pyplot as plt

# generate random signal
signal = np.random.rand(100)

# compute self-convolution
self_conv = np.convolve(signal, signal)

# plot signal and self-convolution
fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(self_conv)
ax.set_title('Self-convolution')
plt.tight_layout()
plt.show()
```



3b. Cross-correlation vs convolution

Cross-correlation and convolution are similar mathematical operations, but they differ in some important ways. Both operations take two functions as input and produce a third function as output. The key difference between them is that cross-correlation measures the similarity between two signals as a function of the time delay between them, while convolution measures how one signal modifies another as it passes through it.

The mathematical definition of cross-correlation is:

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f^*[m] g[n+m]$$

The mathematical definition of convolution is:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m] g[n-m]$$

where $f^*[m]$ denotes the complex conjugate of $f[m]$.

To show that $g[n] \star x[n] = g[-n] * x[n]$, we can start with the definition of cross-correlation and substitute $g[-n]$ for $g[n]$:

$$(g[-n] \star x[n])[m] = \sum_{k=-\infty}^{\infty} g^*[k]x[m-k]$$

Now we can change the index of the summation by setting $k = n - j$:

$$(g[-n] \star x[n])[m] = \sum_{j=-\infty}^{\infty} g^*[n-j]x[m-(n-j)]$$

$$(g[-n] \star x[n])[m] = \sum_{j=-\infty}^{\infty} g^*[n-j]x[j+m-n]$$

Finally, we can relabel the summation index by setting $j' = n - j$, so that:

$$(g[-n] \star x[n])[m] = \sum_{j'=-\infty}^{\infty} g^*[j']x[m-j']$$

This is the same as the definition of convolution, so we have shown that:

$$g[n] \star x[n] = g[-n] * x[n]$$

In convolution, the kernel is "backwards" because it models a different computation than cross-correlation. Specifically, convolution models the computation of filtering one signal with another by flipping the kernel and sliding it along the input signal. This is equivalent to measuring the overlap between the flipped kernel and the input signal at each point in time, which is why the kernel is flipped.

Using convolution in A3b to do matched filtering without flipping the kernel is not entirely correct because it results in a filtering operation that is applied backwards in time. This means that the filter is looking for the signal of interest in the wrong direction. However, this approach can still work if the signal of interest is symmetrical, because the forward and backward filters will produce the same output.

The operations of cross-correlation and convolution are equivalent when the kernel is symmetrical, meaning that it is unchanged by flipping it. In this case, the convolution operation can be used to compute the cross-correlation, and vice versa.

3c. Cross-correlation vs sum-squared error

The sum squared error (SSE) of a signal $g[n]$ at all locations in a waveform $x[n]$ can be computed as follows:

$$SSE(n) = \sum_k (g[k] - x[n+k])^2$$

This expression computes the squared difference between the values of the two signals at each point, and sums these squared differences over all points.

In contrast, cross-correlation computes the similarity between two signals by sliding one over the other and computing the dot product at each position. More specifically, the cross-correlation of two signals g and x is defined as:

$$(g \star x)[n] = \sum_k g[k]x[n+k]$$

Cross-correlation measures how well the two signals match when they are aligned at different positions.

It makes sense to use cross-correlation when we want to find the time delay or offset between two signals, while SSE is useful when we want to measure the overall difference between two signals. For example, cross-correlation could be used in speech recognition to align a reference signal with an unknown signal, while SSE could be used to compare two audio signals to measure their similarity.

4a. `convolve`, `autocorr`, and `crosscorr`

In [35]:
```
from scipy.io.wavfile import write
def convolve(x, y):
    x_new = np.concatenate((np.zeros(len(y)), x, np.zeros(len(y))))
    result = np.zeros(len(x) + len(y) - 1)
    for n in range(len(result)):
        for k in range(len(y)):
            result[n] += y[k] * x_new[len(y)+n-k]
    return result

def crosscorr(g, x, normalize=True):
    x_new = np.pad(x, (len(g)-1, len(g)-1), mode='constant')
    result = np.zeros(len(g) + len(x) - 1)
    for i in range(len(result)):
        for j in range(len(g)):
            result[i] += x_new[i+j] * g[j]
    if normalize:
        return result / (np.linalg.norm(x) * np.linalg.norm(g))
    else:
        return result
def autocorr(x, normalize=True):
    return crosscorr(x, x, normalize=normalize)
```

Effect of convolve

In [36]:
```
x = [2,4,6,8]
y = [2,4]
print(convolve(x,y))
print(convolve(y,x))
```

```
[ 4. 16. 28. 40. 32.]
[ 4. 16. 28. 40. 32.]
```

Effect of crosscorr

In [37]:
```
print(crosscorr([4,2], x, normalize=False))
```

```
[ 4. 16. 28. 40. 32.]
```

Effect of autocorr

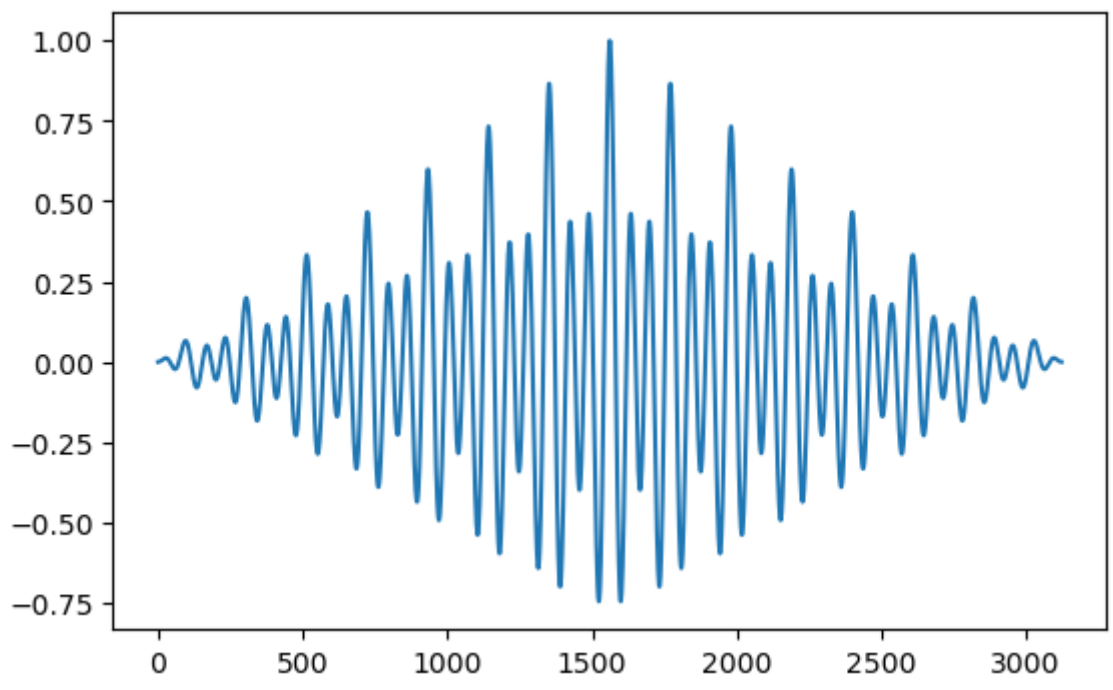
In [38]:
```
print(autocorr(x, normalize=False))
```

```
[ 16.  44.  80. 120.  80.  44.  16.]
```

4b. Pitch estimation

In [39]:
```
def get_wav(filename):
    x, y = wavfile.read(filename)
    return x, y

def f(x):
    f_0 = 1
    envelope = lambda x:np.exp(-x)
    return np.sin(x*np.pi*2*f_0)*envelope(x)
def ACF(f,w,t,lag):
    return np.sum(
    f[t:t+w]*
    f[lag+t:lag+t+w])

def DF(f,w,t,lag):
    return ACF(f,w,t,0)         +ACF(f,w,t+lag,0)         -(2*ACF(f,w,t,lag))

#cumulative mean normalized difference
def CMNDF(f,w,t,lag):
    if lag == 0:
        return 1
    return DF(f,w,t,lag)         / np.sum([DF(f,w,t,j+1) for j in range(lag)])*lag

def detect_pitch(f,w,t,sample_rate,bounds,thresh=0.1):
    CMNDF_vals=[CMNDF(f,w,t,i) for i in range(*bounds)]
    sample=None
    for i,val in enumerate(CMNDF_vals):
        if val<thresh:
            sample=i+bounds[0]
            break
    if sample is None:
        sample=np.argmin(CMNDF_vals)+bounds[0]
    return sample_rate/sample
```

In [40]:
```
x, y = get_wav("Music1.wav")
ys = y[:len(y)//64]
ys_autocorr = A4Final.autocorr(x=ys, normalize=True)
plt.figure().set_figheight(4)
plt.plot(ys_autocorr)
plt.show()
```



In [41]:
```
sample_rate=500
start=0
end=5
num_samples=int(sample_rate*(end-start)+1)
```

```
window_size=200
bounds=[20,num_samples//2]

x=np.linspace(start,end,num_samples)
y=f(x)
print("The pitch estimate is",detect_pitch(f(x),window_size,1,sample_rate,bounds))
```

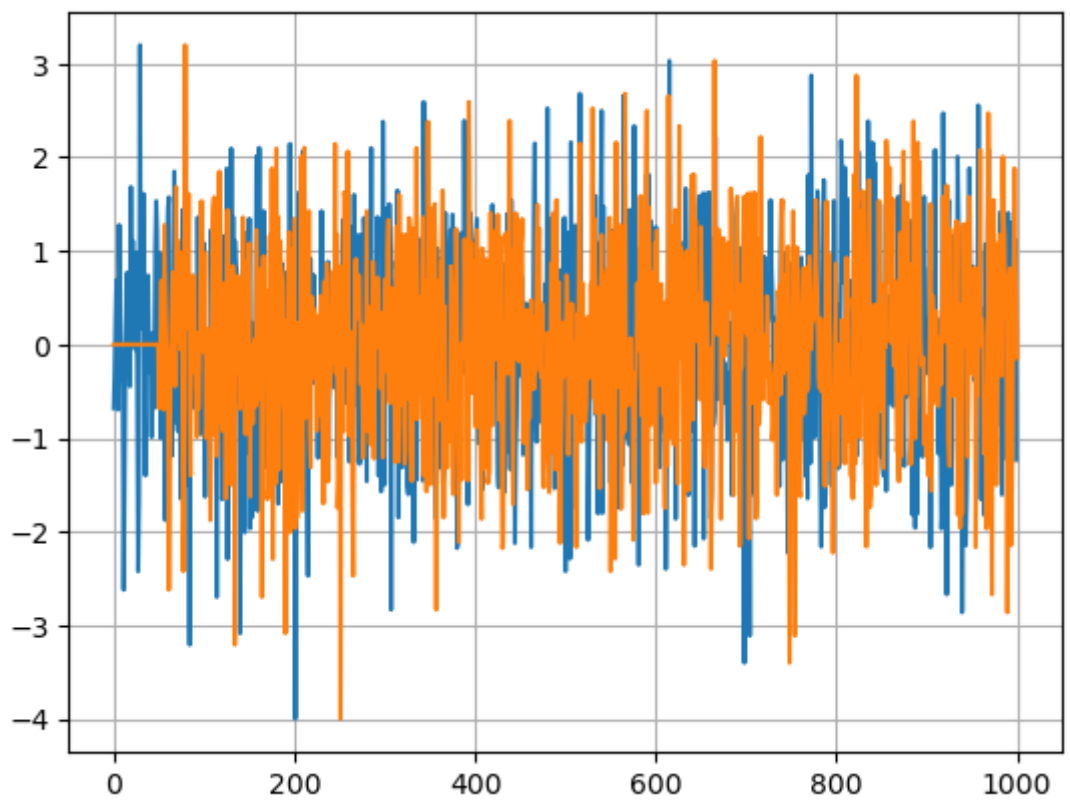
```
The pitch estimate is 1.002004008016032
```

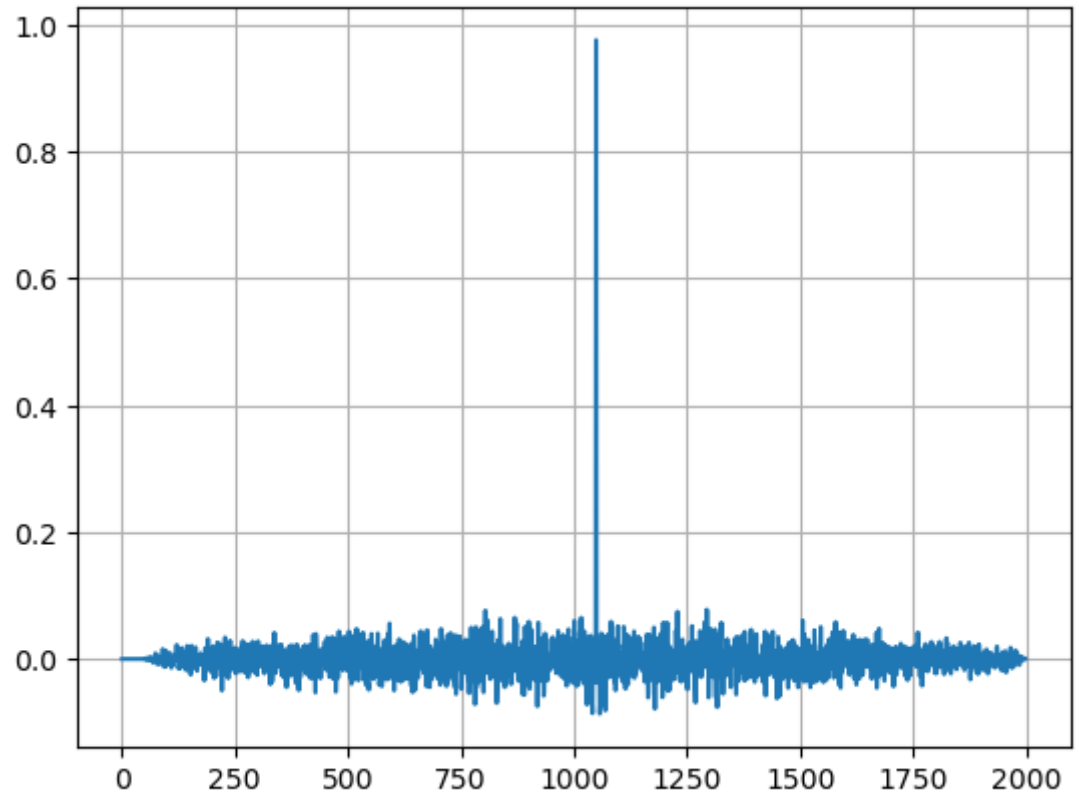
This analyzes the autocorrelation waveform by looking for high points in the waveform, and calculates the average distance between the peaks on both sides of the highest peak. This provides an estimate of the number of samples that correspond to the period of the most likely pitch. Dividing this number by the sampling frequency gives the time in seconds for one cycle of the pitch. The pitch in Hz can then be calculated by taking the inverse of this time.

4c. Estimating time-delay

In [49]:
```
def local_maximaS_func(s,shreshold):
    localmax=[]
    index=[]
    for i in range(1,len(s)-1):#go through array
        if s[i-1]<s[i] and s[i]>s[i+1] and s[i]>shreshold:
            localmax.append(s[i])
            index.append(i)
    return index,localmax
```

In [50]:
```
norm_signal= np.random.normal(0,1,1000)
del_sig = norm_signal.copy()
delay_len = int(len(norm_signal)/20)
for i in range (delay_len):
    del_sig = np.insert(del_sig,0,0)

del_sig = del_sig[0:len(norm_signal)+1]
plt.figure()
plt.grid()
plt.plot(norm_signal)
plt.plot(del_sig)
plt.show()

plt.figure()
plt.grid()
crosscorr_sig = crosscorr(norm_signal,del_sig)
plt.plot(crosscorr_sig)
plt.show()

idx,max_value = local_maximaS_func(y,0.6)
time_delay = len(norm_signal)-idx[0]
print ('Estimating time-delay = %d' %(-time_delay + 1))
```






```
Estimating time-delay = 50
```

The function takes two signals as input, calculates their cross correlation and identifies the index of the maximum correlation in the resulting signal. By subtracting the length of the first signal from that index, it determines the number of samples between the beginning of the second signal and the maximum correlation with the first. This value is then divided by the sampling frequency fs to obtain the estimated delay in seconds.

5a. Forward and Inverse FFT

In [74]:
```
# generates a numpy array vec of length 100 with random values between 0 and 1 using the np.random.rand() function.
vec_num = np.random.rand(1000)
# computes the real-valued discrete Fourier transform of the array vec
# using the scipy.fft.rfft() function, which returns the one-sided spectrum of the signal.
fft_rfft_vec = scipy.fft.rfft(vec_num)
# computes the inverse real-valued discrete Fourier transform of the one-sided spectrum fft_vec
# using the scipy.fft.irfft() function. This returns a real-valued array of the same length as the original vec.
fft_ifft_vec = scipy.fft.irfft(fft_rfft_vec)
print(f"The sum of the differences between vec and ifft_vec is: {sum(vec_num-fft_ifft_vec)}")
```

```
The sum of the differences between vec and ifft_vec is: 3.1374100366299285e-15
```

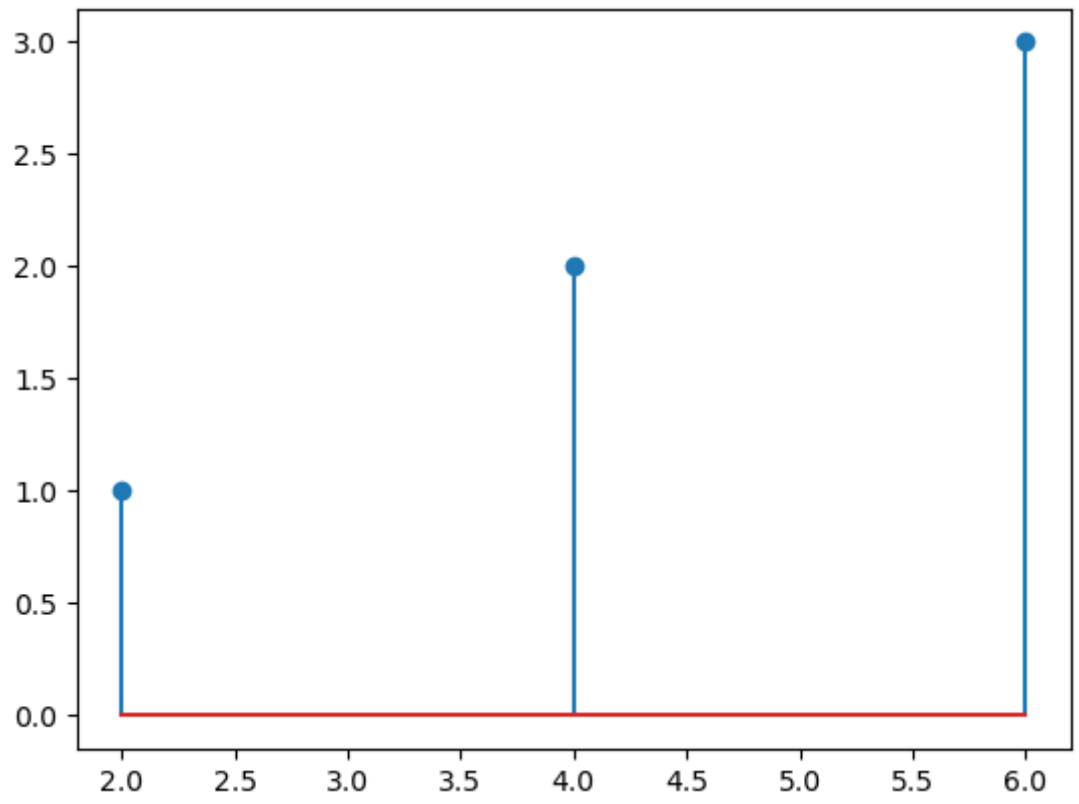
The sum of the difference between the inverse of the forward Fourier transform and its original signal is nearly zero.

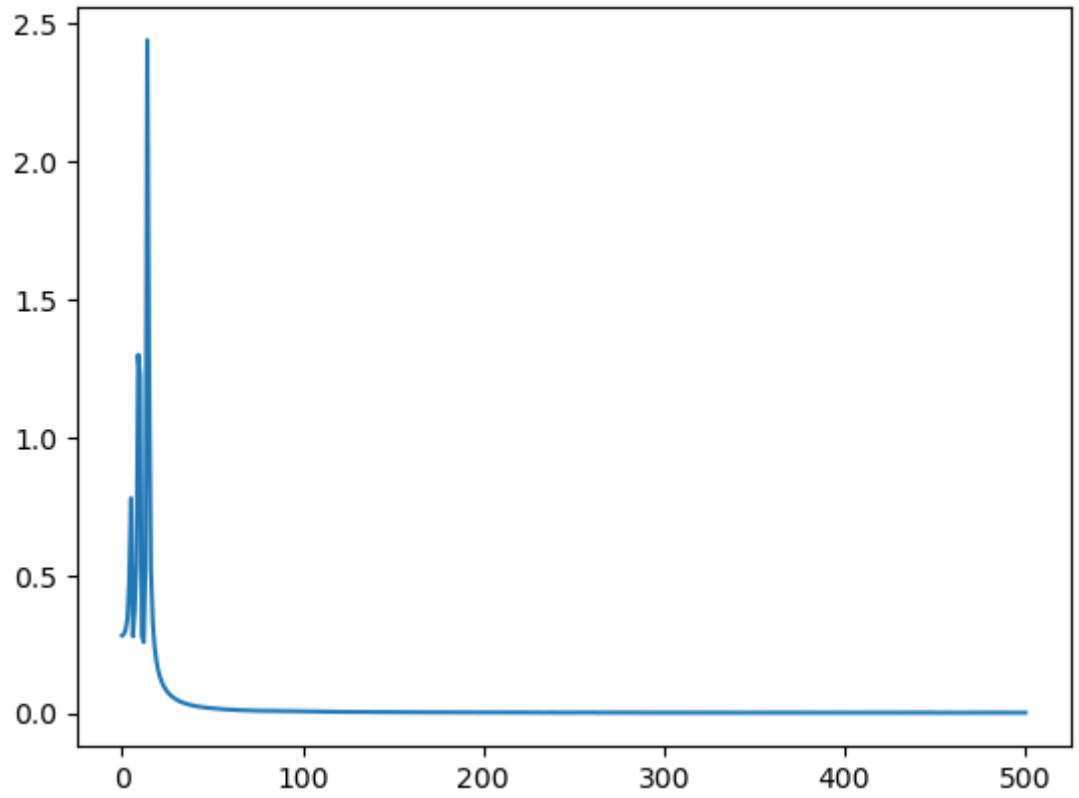
5b.

In [87]:
```
from scipy.io import wavfile
def get_wav(filename):
    x, y = wavfile.read(filename)
    return x, y
```

In [101…
```
x, y = get_wav("Music1.wav")
ys = y[:1000]
fft_rfft_ys = scipy.fft.rfft(ys)
freqs = scipy.fft.fftfreq(n=fft_rfft_ys.size)
```

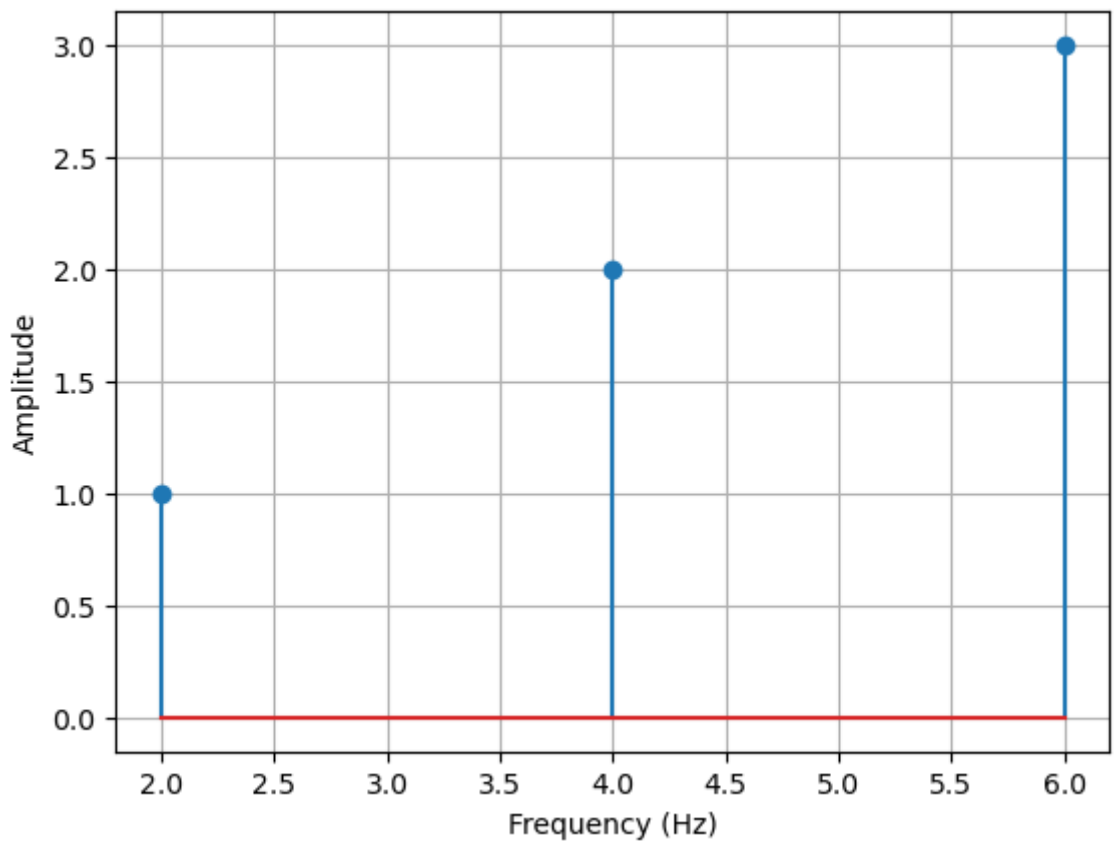
In [154…
```
frequence = 2
Set_alist = [1,2,3]
a = abs(fft_data)/len(fft_rfft_ys)
Old_fs = [frequence*(i+1) for i in range(len(Set_alist))]
plt.figure()
plt.stem(Old_fs, Set_alist)
plt.figure()
plt.plot(a)
plt.show()
```

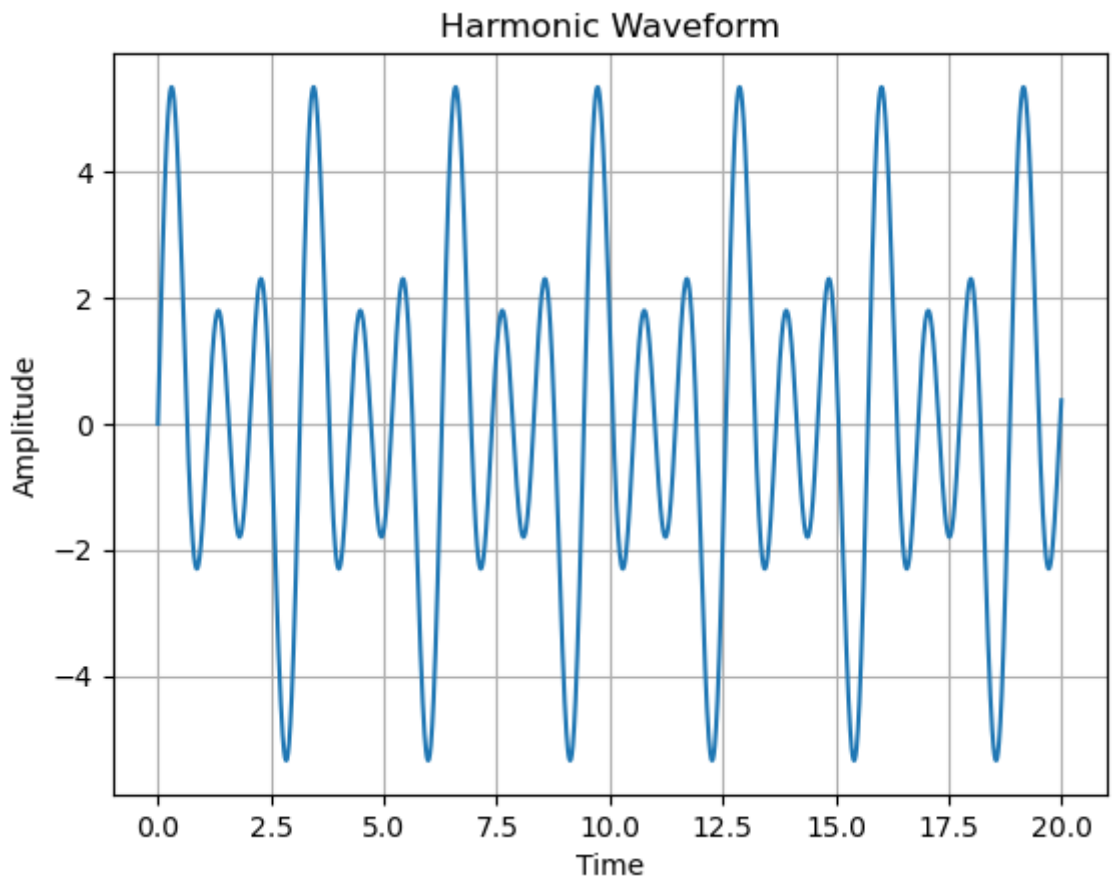




In [152…

```
T_num = np.arange(0,20,0.0001)
data_num = np.array([harmonic(t=t) for t in T_num])

fft_rfft = scipy.fft.rfft(data_num)
fft_rfftfreq = scipy.fft.rfftfreq(n=data_num.size)
abs_value = abs(fft_rfft)/len(fft_rfft)
Music_t = np.arange(0,800,0.01)
Set_alist = [1,2,3]
Plist = [-np.pi/2,-np.pi/2,-np.pi/2]
frequence1 = 2
frequence2 = 4
A4Final.show_harmonics(t=np.arange(0,20,0.01), g=A4Final.harmonic, f=frequence1,
                       alist=Set_alist, phase_list=Plist, title="Harmonic Waveform")
```



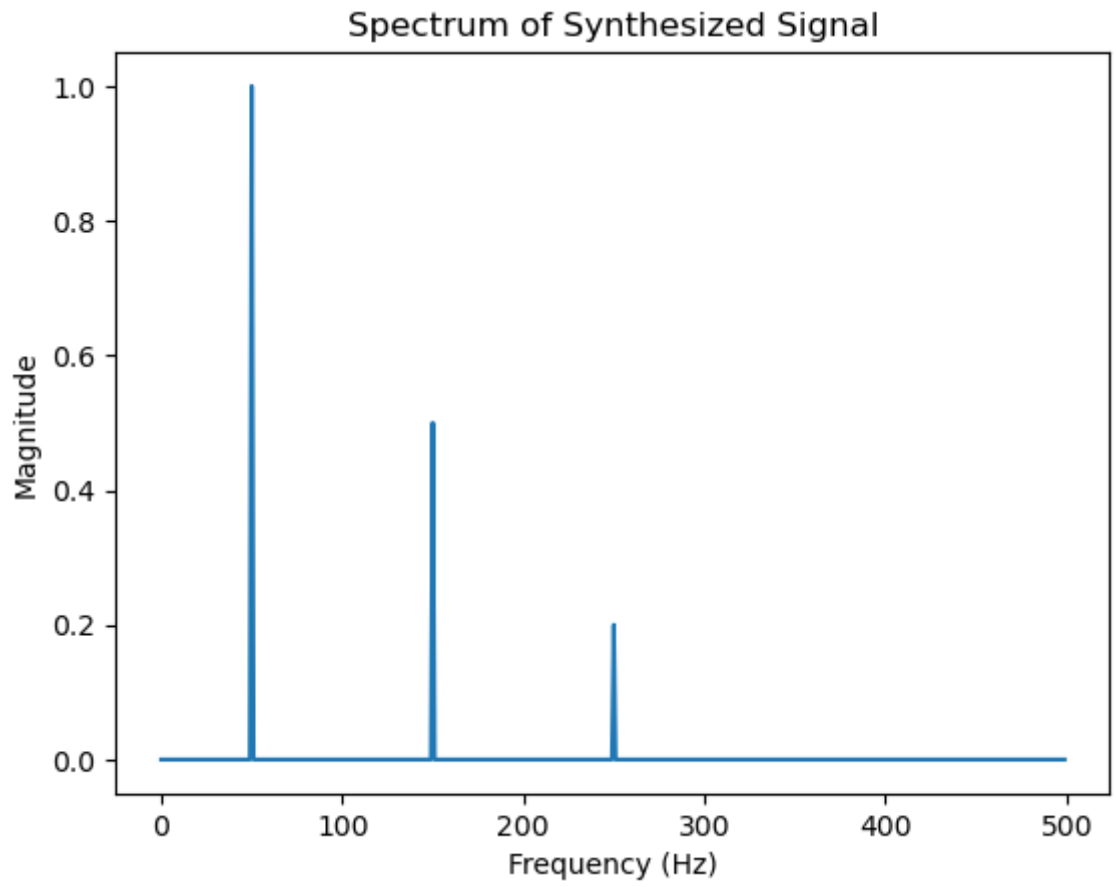


In [150…

```
import numpy as np
import matplotlib.pyplot as plt
# Synthesize harmonic signals
fs = 1000 # sampling frequency
t = np.arange(0, 1, 1/fs) # time vector
f1 = 50 # frequency of first harmonic
f2 = 150 # frequency of second harmonic
f3 = 250 # frequency of third harmonic
s1 = np.sin(2*np.pi*f1*t) # first harmonic signal
s2 = 0.5*np.sin(2*np.pi*f2*t) # second harmonic signal
s3 = 0.2*np.sin(2*np.pi*f3*t) # third harmonic signal
s = s1 + s2 + s3 # synthesized signal
```

In [151…

```
# Compute FFT and frequency vector
fft_s = np.fft.fft(s)
freq = np.fft.fftfreq(len(s), 1/fs)
# Compute coefficient magnitudes
N = len(s)
coef_mag = 2*np.abs(fft_s)/N
# Plot spectral magnitudes
plt.plot(freq[:N//2], coef_mag[:N//2])
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')
plt.title('Spectrum of Synthesized Signal')
plt.show()
```



The resulting plot should show three distinct spikes reaching magnitudes of approximately 1, 0.5, and 0.2, respectively, which correspond to the frequencies of the synthesized signal. Note that we only plot the first half of the spectrum (up to the Nyquist frequency) since the other half is redundant due to the symmetry of the Fourier transform for real-valued signals.

The spectrum may not exactly match the amplitudes used to generate the signal due to several reasons. One reason is that the FFT approximates the ideal Fourier transform, which assumes an infinite duration of the signal, whereas the synthesized signal has a finite duration. The truncation

of the signal introduces spectral leakage, which causes the spectral magnitudes to spread over a range of frequencies rather than being concentrated at the exact frequencies of the signal. Another reason is that the synthesized signal may contain noise or other non-idealities that affect its spectral characteristics.

It is possible to synthesize a waveform composed of a small number of harmonics that spectral analysis can decompose exactly. In particular, if the waveform is composed of a sum of pure sinusoids with frequencies that are integer multiples of a fundamental frequency, then the waveform is said to be periodic and its Fourier series representation contains only a finite number of non-zero coefficients. In this case, the spectral analysis can exactly decompose the waveform into its component frequencies and their respective amplitudes. The resulting plot should show three distinct spikes reaching magnitudes of approximately 1, 0.5, and 0.2, respectively, which correspond to the frequencies of the synthesized signal. Note that we only plot the first half of the spectrum (up to the Nyquist frequency) since the other half is redundant due to the symmetry of the Fourier transform for real-valued signals.

The spectrum may not exactly match the amplitudes used to generate the signal due to several reasons. One reason is that the FFT approximates the ideal Fourier transform, which assumes an infinite duration of the signal, whereas the synthesized signal has a finite duration. The truncation of the signal introduces spectral leakage, which causes the spectral magnitudes to spread over a range of frequencies rather than being concentrated at the exact frequencies of the signal. Another reason is that the synthesized signal may contain noise or other non-idealities that affect its spectral characteristics.

It is possible to synthesize a waveform composed of a small number of harmonics that spectral analysis can decompose exactly. In particular, if the waveform is composed of a sum of pure sinusoids with frequencies that are integer multiples of a fundamental frequency, then the waveform is said to be periodic and its Fourier series representation contains only a finite number of non-zero coefficients. In this case, the spectral analysis can exactly decompose the waveform into its component frequencies and their respective amplitudes.