# signal_processing_basics

January 14, 2024

- Use sudo apt-get install portaudio19-dev and install the package to be able to hear the sounds

[1]:

```
pip install numpy matplotlib sounddevice
```

```
Requirement already satisfied: numpy in
/home/akhil/anaconda3/lib/python3.9/site-packages (1.25.1)
Requirement already satisfied: matplotlib in
/home/akhil/anaconda3/lib/python3.9/site-packages (3.7.2)
Collecting sounddevice
  Downloading sounddevice-0.4.6-py3-none-any.whl (31 kB)
Requirement already satisfied: contourpy>=1.0.1 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (1.3.2)
Requirement already satisfied: packaging>=20.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (23.2)
Requirement already satisfied: pillow>=6.2.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (9.0.1)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (3.0.4)
Requirement already satisfied: python-dateutil>=2.7 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: importlib-resources>=3.2.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (6.1.0)
Requirement already satisfied: CFFI>=1.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from sounddevice) (1.15.0)
Requirement already satisfied: pycparser in
/home/akhil/anaconda3/lib/python3.9/site-packages (from CFFI>=1.0->sounddevice)
(2.21)
Requirement already satisfied: zipp>=3.1.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from importlib-
resources>=3.2.0->matplotlib) (3.7.0)
Requirement already satisfied: six>=1.5 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from python-
dateutil>=2.7->matplotlib) (1.16.0)
```

```
Installing collected packages: sounddevice
Successfully installed sounddevice-0.4.6

[notice] A new release of pip is
available: 23.3.1 -> 23.3.2
[notice] To update, run:
pip install --upgrade pip
Note: you may need to restart the kernel to use updated packages.
```

[1]:

```python
import numpy as np
import matplotlib.pyplot as plt
import sounddevice as sd

# Function to generate and plot a signal
def generate_and_plot_signal(signal, title):
    plt.figure(figsize=(10, 4))
    plt.plot(t, signal)
    plt.title(title)
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.grid(True)
    plt.show()

# Function to listen to the signal
def listen_to_signal(signal, sampling_frequency):
    sd.play(signal, samplerate=sampling_frequency)
    sd.wait()
```

# 1 Signal characteristics

1. Plotting:

- **Discrete signal**: A signal that is sampled at specific points in time, resulting in a sequence of discrete values. All the plots given below are discrete signals.
- **Continuous-Time Signal**: A signal that varies continuously with respect to time, like an analog signal from the physical world.
- In the plots given below, each signal represents a different type of chirp (a sinusoidal signal whose frequency is dependent on time).
- In the plots given below for various chirps, **amplitude of the wave is on y-axis** and **x-axis contains time points (in seconds)**.
- **Rayleigh frequency**: It is the minimum frequency that can be resolved in a finite duration of time window. If the time window is T seconds long, then minimum frequency that can be resolved is **1/T Hz**.
- **Resolution of a signal**: It's the minimum frequency that we can distinguish in a signal. It can be thought of as a bin size in the sampled signal.
- **Sampling Rate**: Refers to the number of samples (data points) captured or recorded per unit of time. It is a critical parameter in the process of converting a continuous-time signal

into a discrete-time signal.

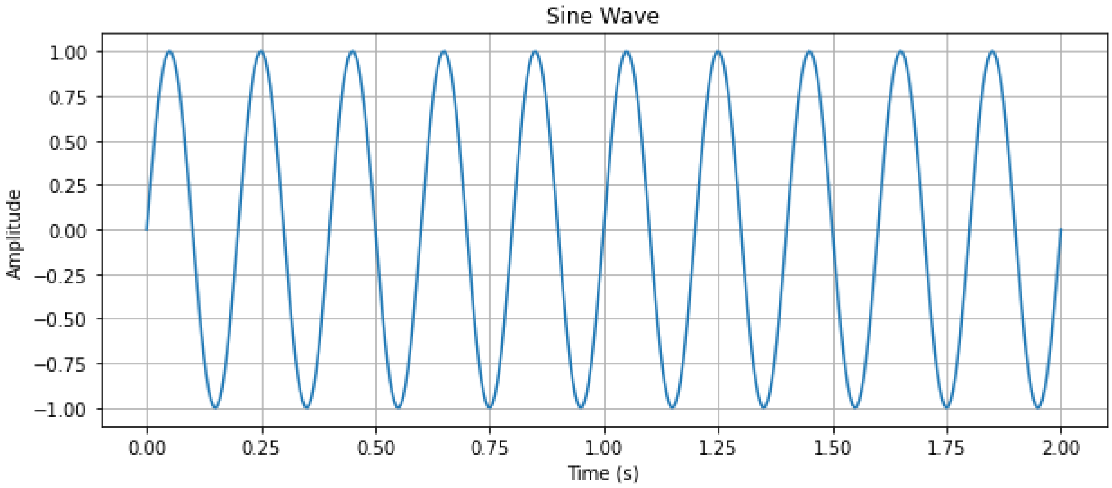
- **Nyquist Frequency**: The Nyquist frequency is defined as half of the sampling rate.
- **Nyquist-Shannon sampling theorem**: States that in order to accurately represent a continuous-time signal in a discrete form, the sampling rate must be at least twice the maximum frequency present in the signal. If the sampling rate is less than twice the maximum frequency, aliasing can occur, leading to errors in the representation of the signal.
- **Amplitude range of a signal**: It is the range of values that span from the minimum amplitude to the maximum amplitude a signal can take over a time interval.
- **Auditory hearing range of humans**: Humans can hear signals that contain frequencies in the range between 20Hz to 20kHz.

### 1.1 Sine wave

- Sampling frequency: 200 Hz
- Natural frequency of sine wave = 5 Hz
- Range of signal: -1 units to 1 units
- Nyquist frequency: 100 Hz
- Rayleigh frequency: 0.5 Hz
- Frequencies in wave: 5 Hz

[203]:

```
# Generate time values
t = np.linspace(0, 2, 400)  # 2 seconds, 400 samples
freq = 5
sine_wave = np.sin(2 * np.pi * freq * t)
generate_and_plot_signal(sine_wave, 'Sine Wave')
print(f"Range of signal (In terms of amplitude): {min(sine_wave)} units to␣
 ↪{max(sine_wave)} units\n")
print(f"Sampling frequency: {200}Hz, Frequency of wave: {freq} Hz\n")

listen_to_signal(sine_wave, 7000)
```



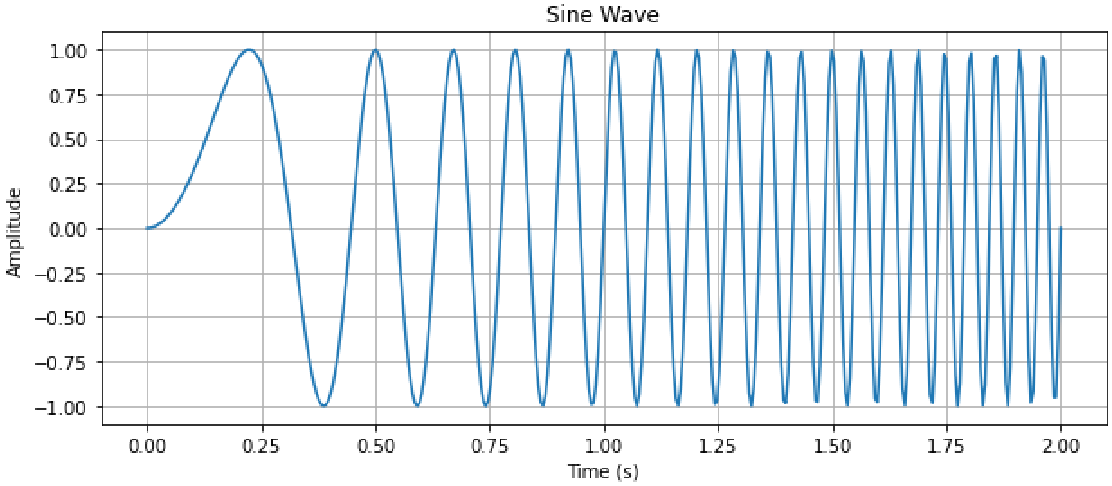
```
Range of signal (In terms of amplitude): -0.9999922506833704 units to
0.9999922506833704 units

Sampling frequency: 200Hz, Frequency of wave: 5 Hz
```

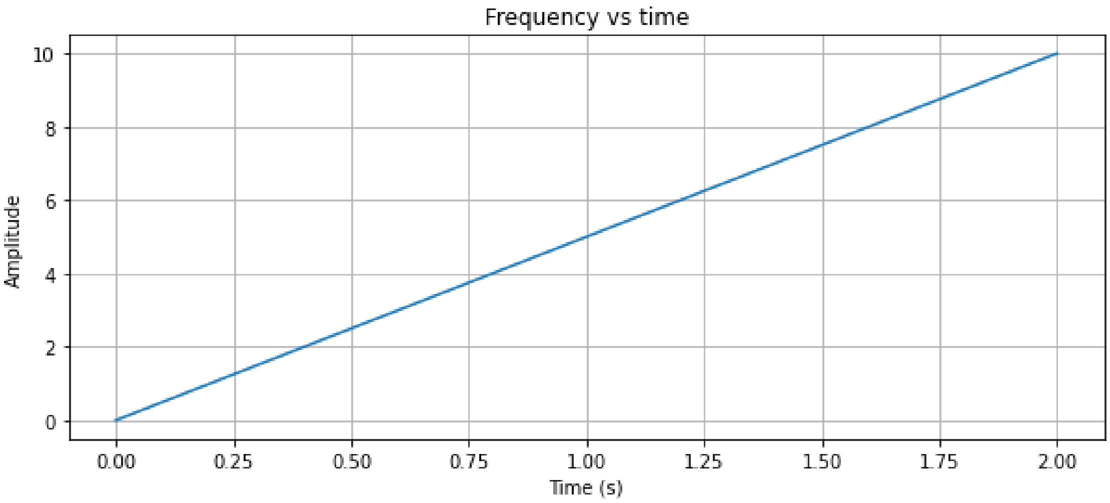
- Wave is a normal sinusoidal wave sampled for a time interval of 2 seconds. Wave can't be heard as it contains 5 Hz frequency which falls outside the human auditory hearing range (25 Hz - 25 kHz).

### 1.2 Linear chirp

- Sampling frequency: 200 Hz
- Natural frequency of sine wave = (5 * t) Hz, where t is an array of time intervals
- Range of signal: -1 units to 1 units
- Nyquist frequency: 100 Hz
- Rayleigh frequency: 0.5 Hz
- Frequencies in wave: 0 to 10 Hz

```
[204]: t = np.linspace(0, 2, 400)  # 5 seconds, 5000 samples
freq = 5 * t
linear_chirp = np.sin(2 * np.pi * freq * t)
generate_and_plot_signal(linear_chirp, 'Sine Wave')
listen_to_signal(linear_chirp, 7000)
generate_and_plot_signal(freq, 'Frequency vs time')
print(f"Range of signal (In terms of amplitude): {min(linear_chirp)} units to␣
  ↪{max(linear_chirp)} units\n")
print(f"Sampling frequency: {200}Hz, Frequency of wave: {min(freq)} Hz to␣
  ↪{max(freq)} Hz\n")
```





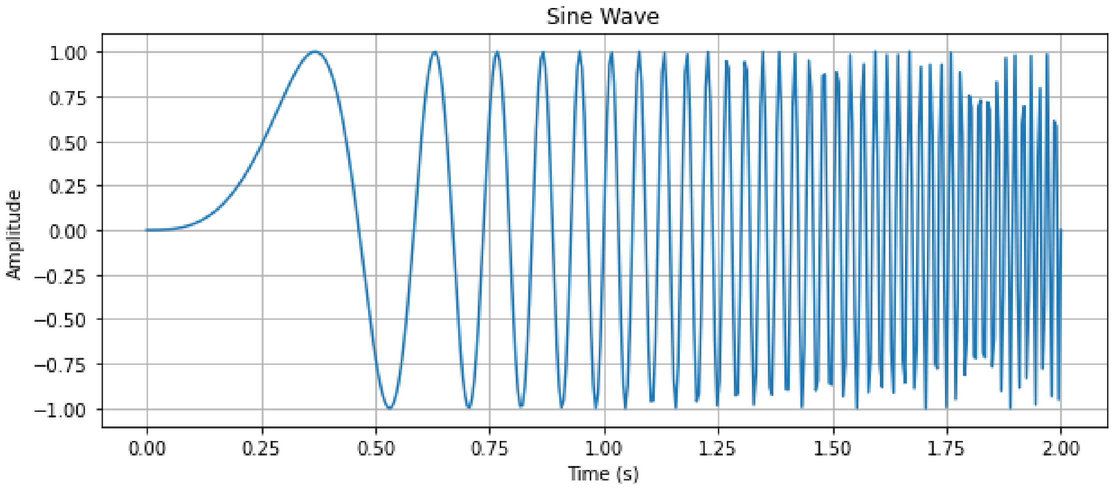
```
Range of signal (In terms of amplitude): -0.9999884554964279 units to
0.9998584999152674 units

Sampling frequency: 200Hz, Frequency of wave: 0.0 Hz to 10.0 Hz
```

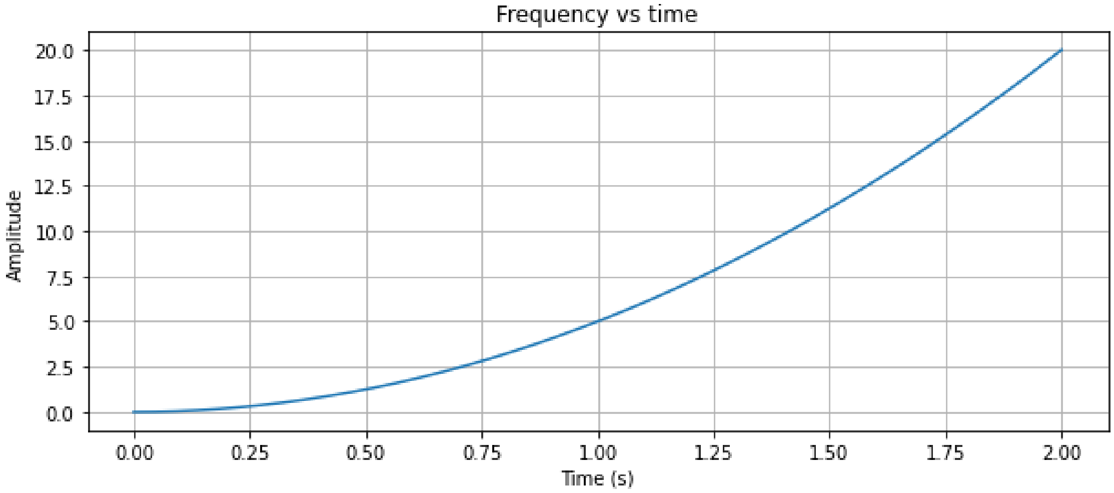
- Wave is a sinusoidal wave that has a linearly time dependent frequency, sampled for a time interval of 2 seconds. Wave can't be heard as its frequency range (0 to 10 Hz) falls outside the human auditory hearing range (20 Hz - 20 kHz).

### 1.3 Quadratic chirp

- Sampling frequency: 200 Hz
- Natural frequency of sine wave = (5 * (t^2)) Hz, where t is an array of time intervals
- Range of signal: -1 units to 1 units
- Nyquist frequency: 100 Hz
- Rayleigh frequency: 0.5 Hz
- Frequencies in wave: 0 to 20 Hz

```
[205]: t = np.linspace(0, 2, 400)  # 5 seconds, 5000 samples
freq = 5 * (t ** 2)
quadratic_chirp = np.sin(2 * np.pi * freq * t)
generate_and_plot_signal(quadratic_chirp, 'Sine Wave')
listen_to_signal(quadratic_chirp, 7000)
generate_and_plot_signal(freq, 'Frequency vs time')
print(f"Range of signal (In terms of amplitude): {min(quadratic_chirp)} units␣
 ↪to {max(quadratic_chirp)} units\n")
print(f"Sampling frequency: {200}Hz, Frequency of wave: {min(freq)} Hz to␣
 ↪{max(freq)} Hz\n")
```





```
Range of signal (In terms of amplitude): -0.9999999996712015 units to
0.9999997603058647 units

Sampling frequency: 200Hz, Frequency of wave: 0.0 Hz to 20.0 Hz
```

- Wave is a sinusoidal wave whose frequency depends quadratically with time, sampled for a time interval of 2 seconds. Wave can't be heard as its frequency range (0 to 20 Hz) falls outside the human auditory hearing range (20 Hz - 20 kHz).

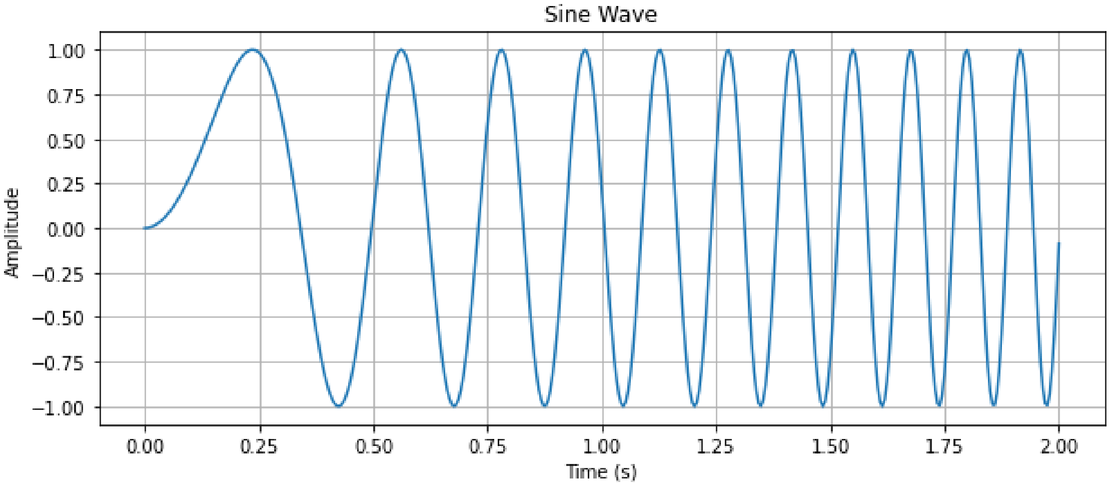
### 1.4 Logarithmic chirp

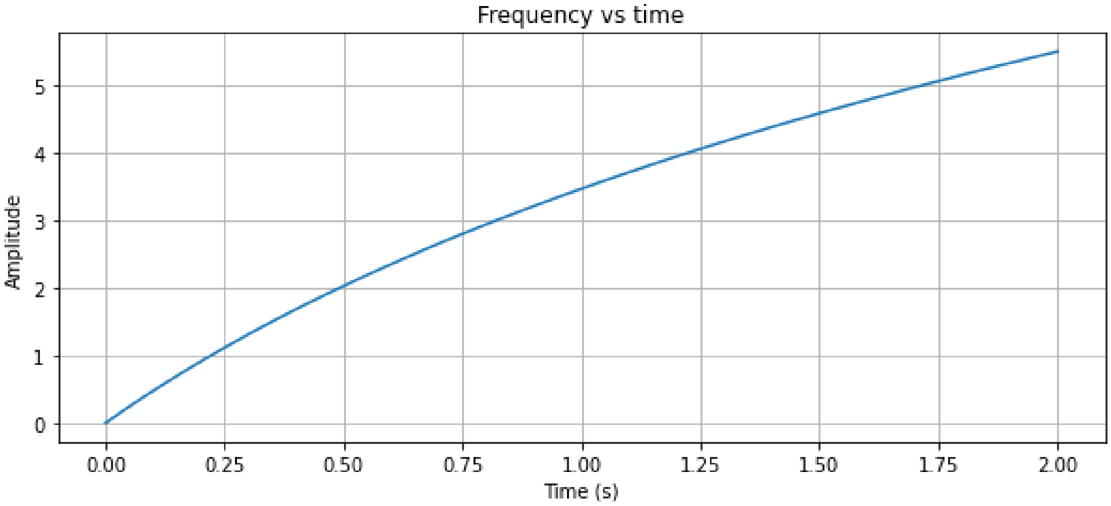
- Sampling frequency: 200 Hz
- Natural frequency of sine wave = (5 * (log(1 + t))) Hz, where t is an array of time intervals

- Range of signal: -1 units to 1 units
- Nyquist frequency: 100 Hz
- Rayleigh frequency: 0.5 Hz
- Frequencies in wave: 0 to ~5.5 Hz

[207]:

```
t = np.linspace(0, 2, 400)  # 5 seconds, 5000 samples
freq = 5 * (np.log(1 + t))
logarithmic_chirp = np.sin(2 * np.pi * freq * t)
generate_and_plot_signal(logarithmic_chirp, 'Sine Wave')
listen_to_signal(logarithmic_chirp, 7000)
generate_and_plot_signal(freq, 'Frequency vs time')
print(f"Range of signal (In terms of amplitude): {min(logarithmic_chirp)} units␣
 ↪to {max(logarithmic_chirp)} units\n")
print(f"Sampling frequency: {200}Hz, Frequency of wave: {min(freq)} Hz to␣
 ↪{max(freq)} Hz\n")
```





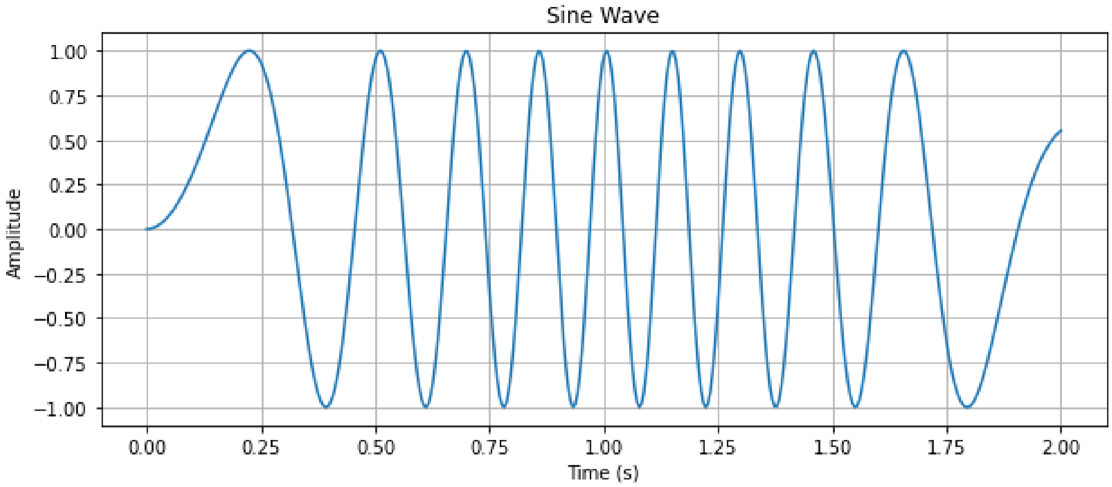
```
Range of signal (In terms of amplitude): -0.9999867943242016 units to
0.9999884421538878 units

Sampling frequency: 200Hz, Frequency of wave: 0.0 Hz to 5.493061443340549 Hz
```

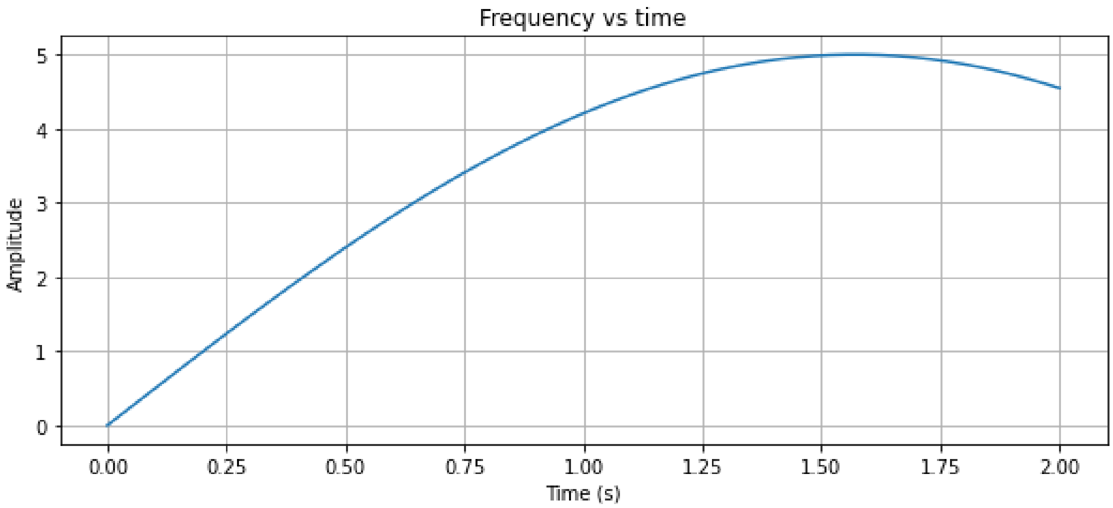
- Wave is a sinusoidal wave whose frequency depends logarithmically with time, sampled for a time interval of 2 seconds. Wave can't be heard as it falls outside the human auditory hearing range (20 Hz - 20 kHz).

## 1.5 Concave chirp

- Sampling frequency: 200 Hz
- Natural frequency of sine wave = (5 * (sin(t))) Hz, where t is an array of time intervals
- Range of signal: -1 units to 1 units
- Nyquist frequency: 100 Hz
- Rayleigh frequency: 0.5 Hz
- Frequencies in wave: 0 to ~5 Hz

[209]:
```
t = np.linspace(0, 2, 400)  # 5 seconds, 5000 samples
freq = 5 * (np.sin(t))
concave_chirp = np.sin(2 * np.pi * freq * t)
generate_and_plot_signal(concave_chirp, 'Sine Wave')
listen_to_signal(concave_chirp, 7000)
generate_and_plot_signal(freq, 'Frequency vs time')
print(f"Range of signal (In terms of amplitude): {min(concave_chirp)} units to␣
 ↪{max(concave_chirp)} units\n")
print(f"Sampling frequency: {200}Hz, Frequency of wave: {min(freq)} Hz to␣
 ↪{max(freq)} Hz\n")
```





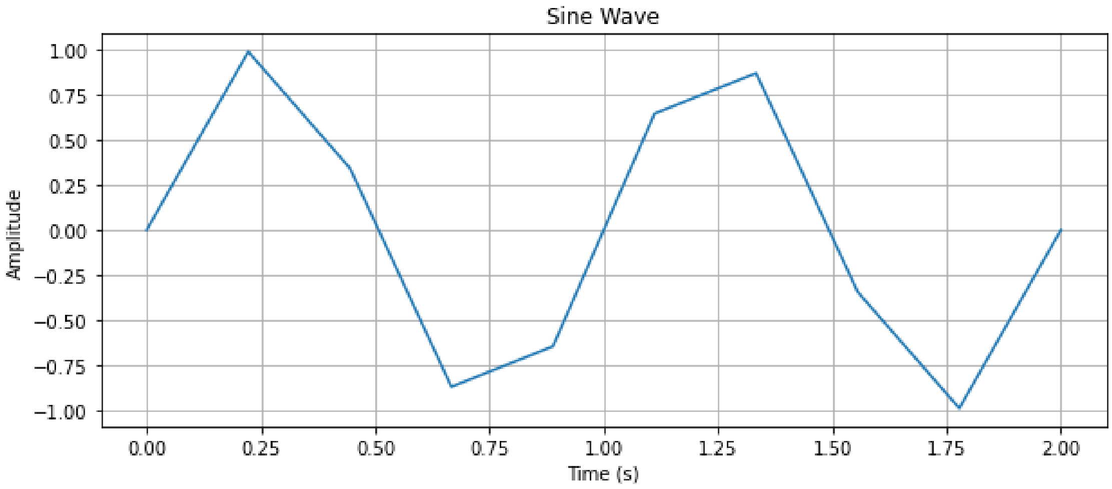
```
Range of signal (In terms of amplitude): -0.9999892489770762 units to
0.9999866417656452 units

Sampling frequency: 200Hz, Frequency of wave: 0.0 Hz to 4.999991220115513 Hz
```

# 2 Frequency spectra

## 2.1 Wrap around frequency example

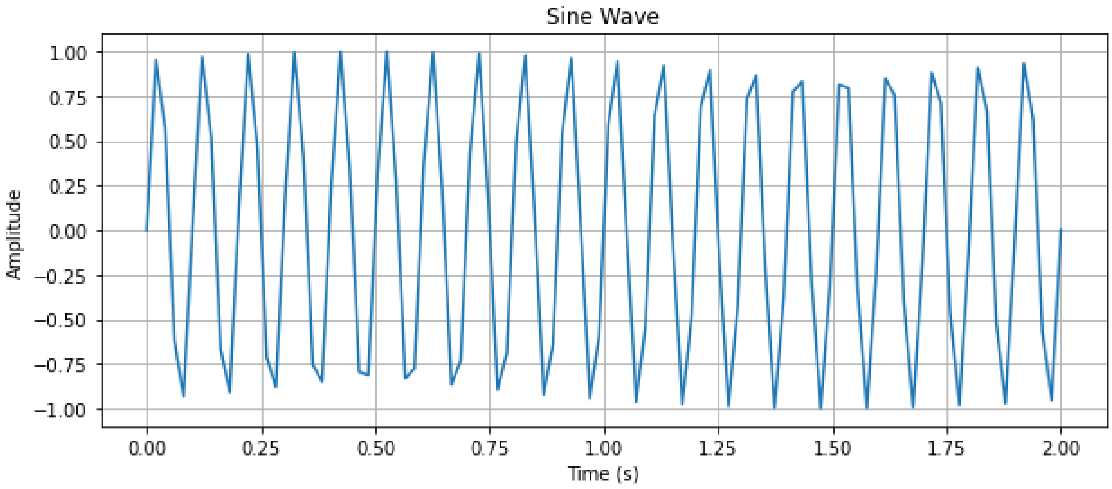
- We can observe aliasing if sampling frequency is less than 2 * max frequency of a signal from the below below as the sin wave is in a poor shape

[225]:
```
t = np.linspace(0, 2, 10)  # 5 seconds, 5000 samples
freq = 10
wrap_around = np.sin(2 * np.pi * freq * t)
generate_and_plot_signal(wrap_around, 'Sine Wave')
```



[226]:

```
t = np.linspace(0, 2, 100)  # 5 seconds, 5000 samples
freq = 10
wrap_around = np.sin(2 * np.pi * freq * t)
generate_and_plot_signal(wrap_around, 'Sine Wave')
```



## 2.2 FFT

- When FFT is applied to a real-valued signal, the resulting spectrum is symmetric, the positive frequencies up to the Nyquist frequency are mirrored in the negative frequencies. Hence, the FFT output for a real signal provides information about the positive frequencies up to the Nyquist frequency.
- **X-axis (in plots given below)**: 0 Hz to Nyquist frequency (100 Hz as sampling frequency = 200 Hz)

- **Y-axis (in plots given below)**: Amplitude in decibels (dB)

### 2.3 Linear chirp

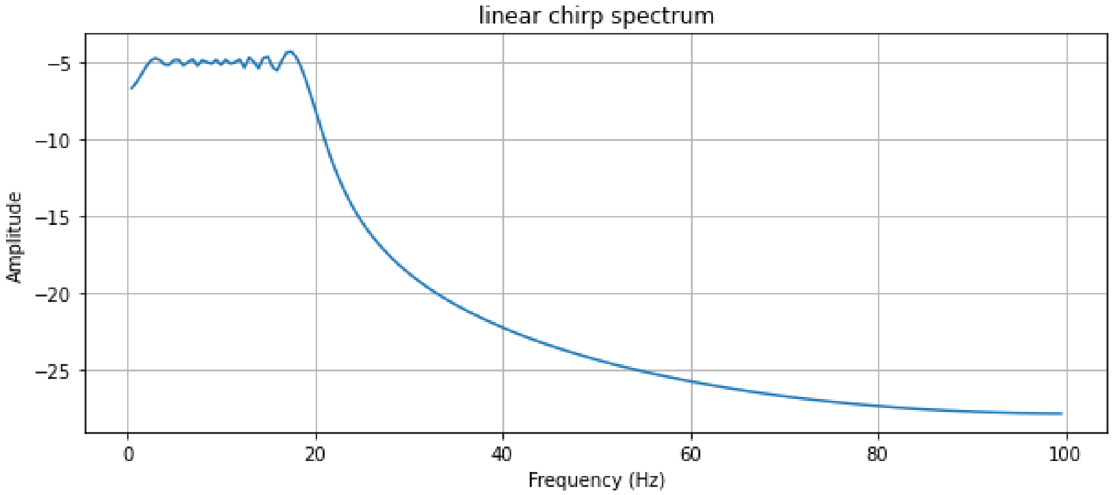
[210]:

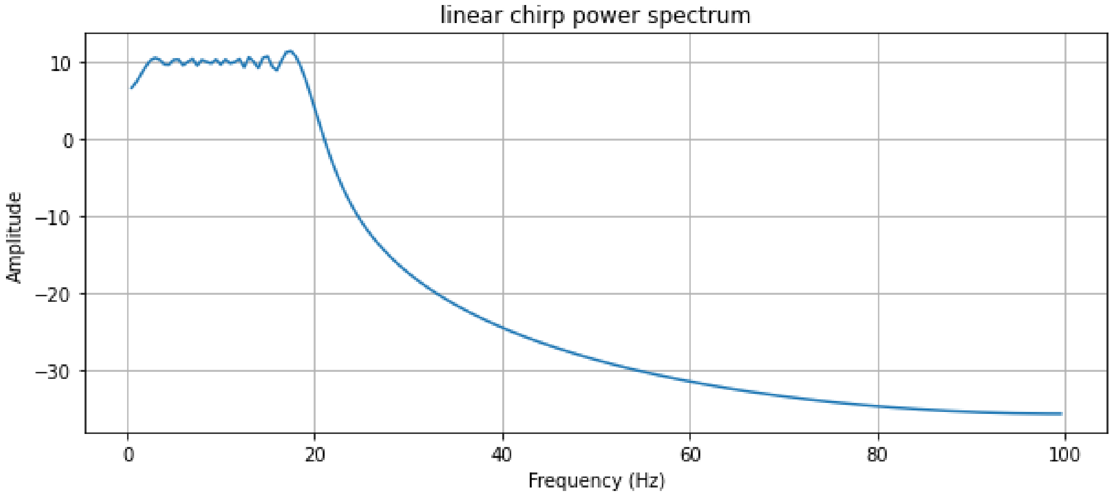
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import welch

def pwelch(signal, fs):
    # Generate a sample signal (replace this with your own signal)
      # Sampling frequency in Hz
    # Calculate the power spectral density using Welch method
    frequencies, psd = welch(signal, fs=fs, nperseg=256)

    # Plot the results
    plt.semilogy(frequencies, psd)
    plt.title('Power Spectral Density (PSD) - Welch Method')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Power/Frequency (dB/Hz)')
    plt.grid(True)
    plt.show()
```

[211]:

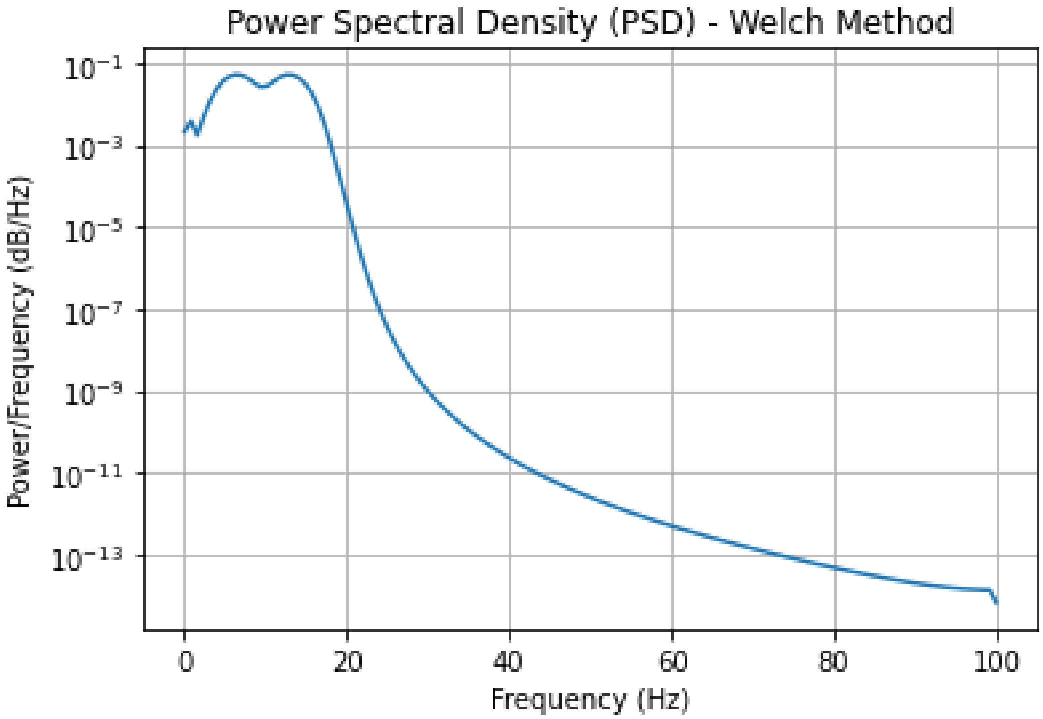
```
def compute_spectrum(signal, title, sampling_frequency):
    spectrum = np.fft.fft(signal)
    frequencies = np.fft.fftfreq(len(signal), d=1/sampling_frequency)
    pos = frequencies > 0
    fft_mag = 2 * np.abs(spectrum[pos]/sampling_frequency)
    return frequencies[pos], fft_mag, pos

def plot_spectrum(frequencies, title, spectrum):
    plt.figure(figsize=(10, 4))
    plt.plot(frequencies, 10 * np.log10(spectrum))
    plt.title(title)
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Amplitude')
    plt.grid(True)

def compute_power_spectrum(signal, title, sampling_frequency):
    power_spectrum = np.fft.fft(signal)
    power_frequencies = np.fft.fftfreq(len(signal), d=1/sampling_frequency)
    power_pos = power_frequencies > 0
    power_mag = 2 * np.abs(power_spectrum[power_pos]**2/sampling_frequency)
    return power_frequencies[power_pos], power_mag, power_pos

sampling_frequency = 200
linear_frequencies, linear_spectrum, linear_pos =␣
 ↪compute_spectrum(linear_chirp, 'linear_chirp', sampling_frequency)
```

```
linear_power_frequencies, linear_power_spectrum, linear_power_pos =␣
 ↪compute_power_spectrum(linear_chirp, 'linear_chirp', sampling_frequency)

plot_spectrum(linear_frequencies, 'linear chirp spectrum', linear_spectrum)
plot_spectrum(linear_power_frequencies, 'linear chirp power spectrum',␣
 ↪linear_power_spectrum)
```



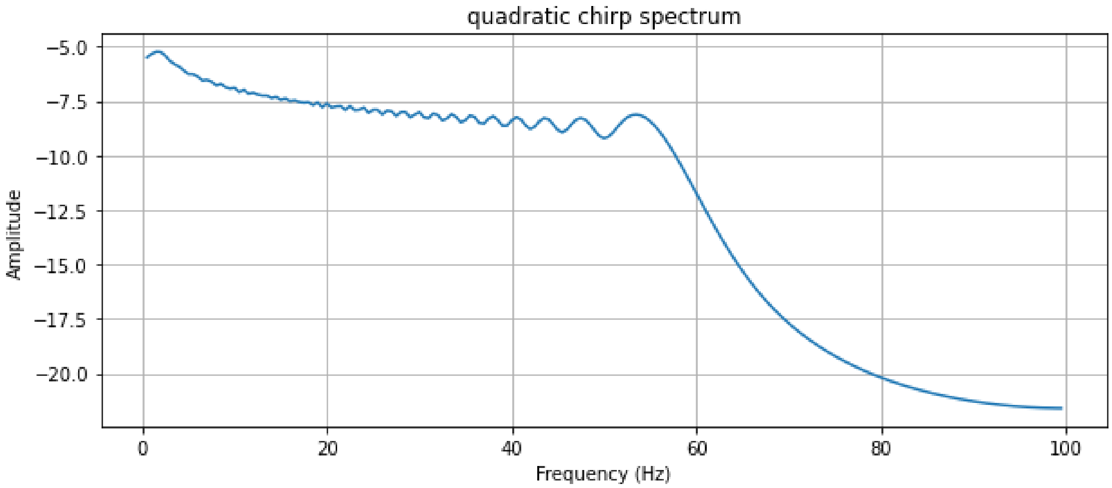


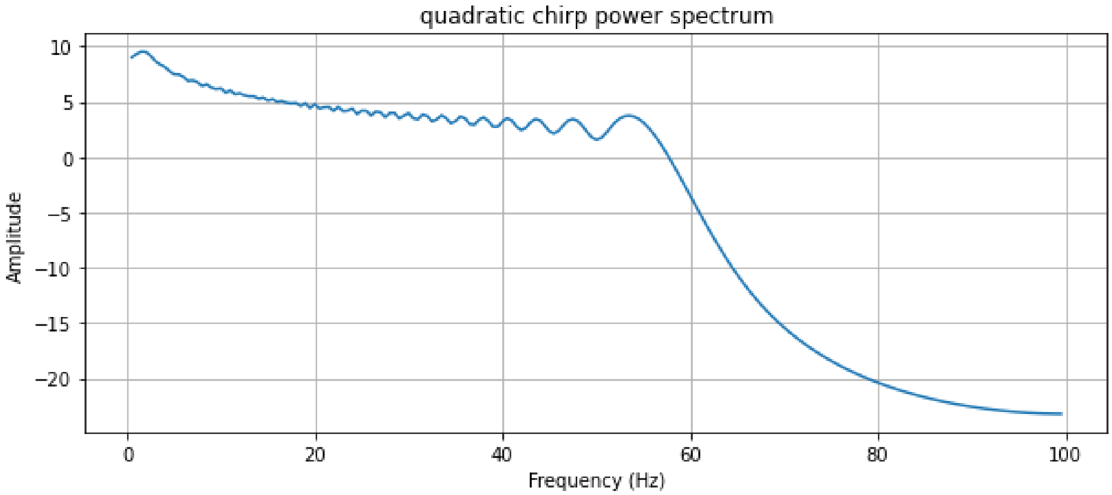
[212]:
```
pwelch(linear_chirp, 200)
```



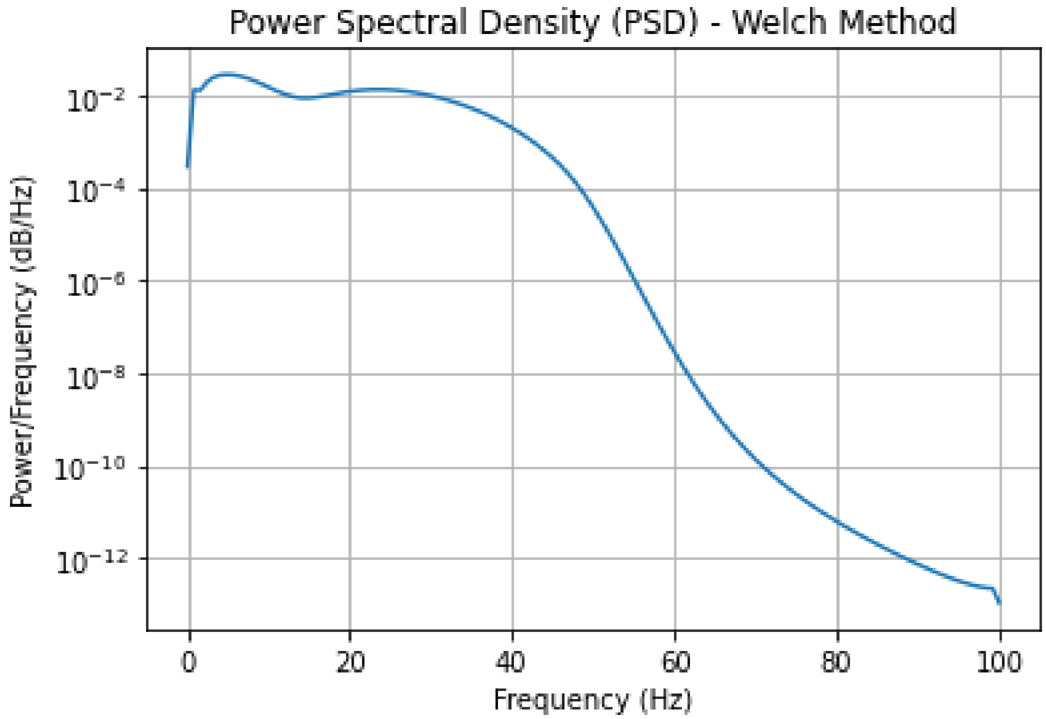
- From the above plots for power spectra of the linear chirp signal, it can be seen that frequencies have positive amplitudes from 0 to 20 Hz whereas the signal has frequencies ranging from 0 Hz to 10 Hz (twice the max frequency of signal as fft is symmetric and gives equal response for both positive and negative frequencies).

[213]:
```
quadratic_frequencies, quadratic_spectrum, quadratic_pos =␣
 ↪compute_spectrum(quadratic_chirp, 'quadratic_chirp', sampling_frequency)
quadratic_power_frequencies, quadratic_power_spectrum, quadratic_power_pos =␣
 ↪compute_power_spectrum(quadratic_chirp, 'quadratic_chirp',␣
 ↪sampling_frequency)

plot_spectrum(quadratic_frequencies, 'quadratic chirp spectrum',␣
 ↪quadratic_spectrum)
plot_spectrum(quadratic_power_frequencies, 'quadratic chirp power spectrum',␣
 ↪quadratic_power_spectrum)
```



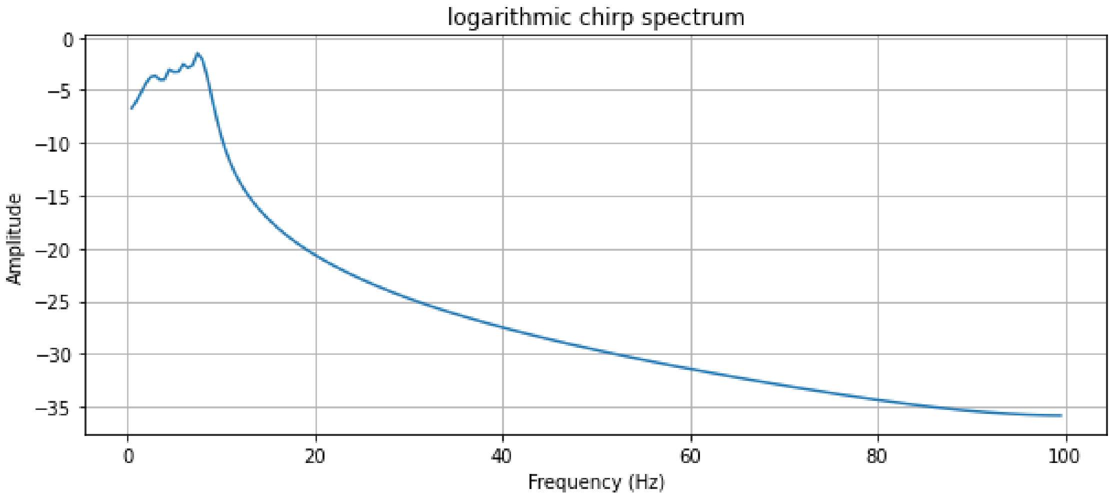


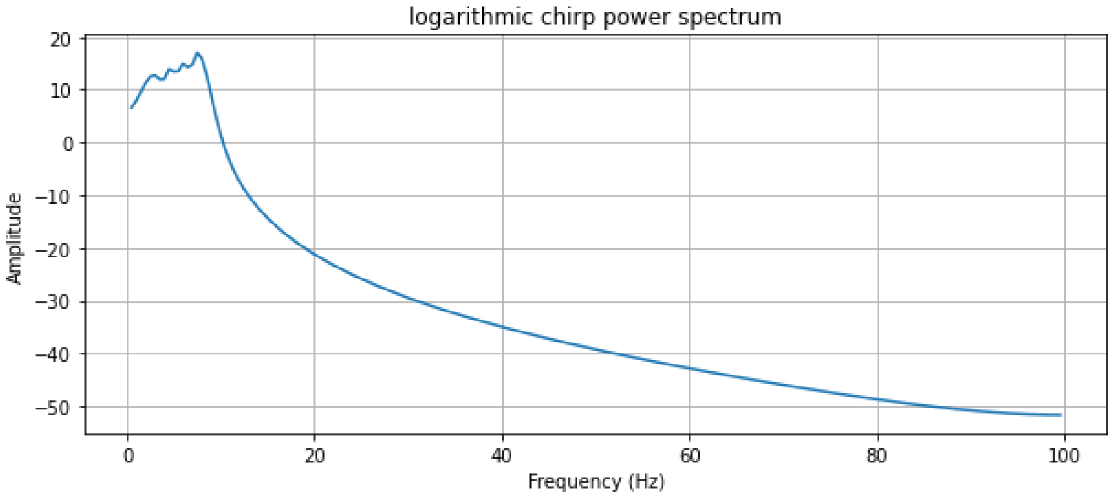
[214]:
```
pwelch(quadratic_chirp, 200)
```



- From the above plots for power spectra of the quadratic chirp signal, it can be seen that frequencies have positive amplitudes from 0 to 40 Hz whereas the signal has frequencies ranging from 0 Hz to 20 Hz (twice the max frequency of signal as fft is symmetric and gives equal response for both positive and negative frequencies).

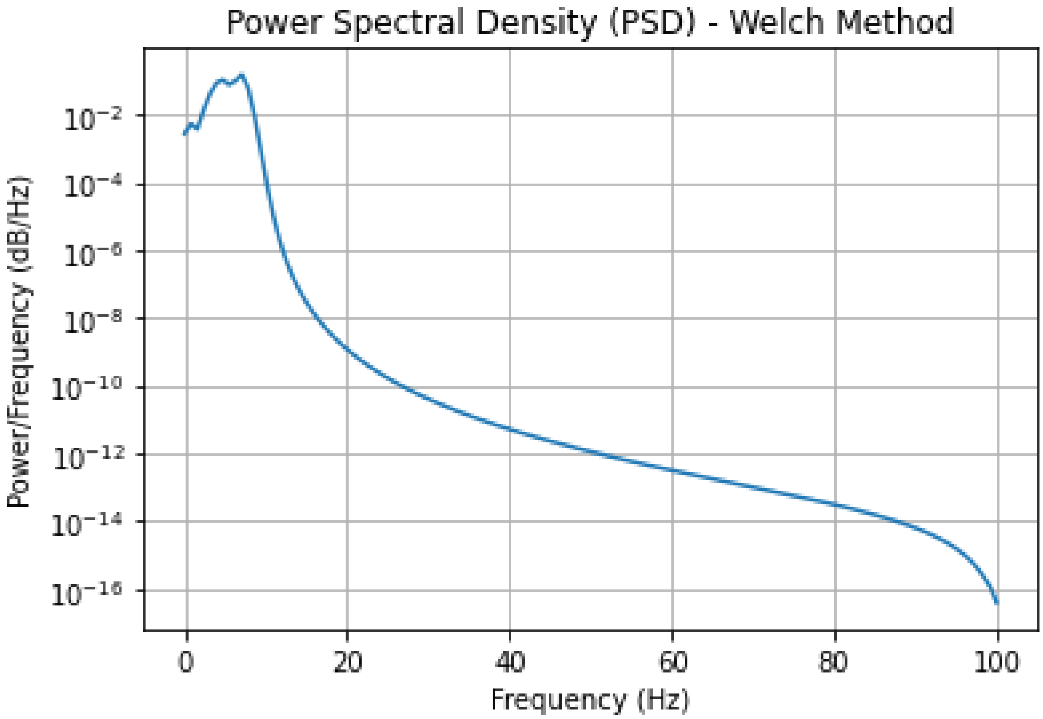
[215]:
```
logarithmic_frequencies, logarithmic_spectrum, logarithmic_pos =␣
 ↪compute_spectrum(logarithmic_chirp, 'logarithmic_chirp', sampling_frequency)
logarithmic_power_frequencies, logarithmic_power_spectrum,␣
 ↪logarithmic_power_pos = compute_power_spectrum(logarithmic_chirp,␣
 ↪'logarithmic_chirp', sampling_frequency)

plot_spectrum(logarithmic_frequencies, 'logarithmic chirp spectrum',␣
 ↪logarithmic_spectrum)
plot_spectrum(logarithmic_power_frequencies, 'logarithmic chirp power␣
 ↪spectrum', logarithmic_power_spectrum)
```





[216]:

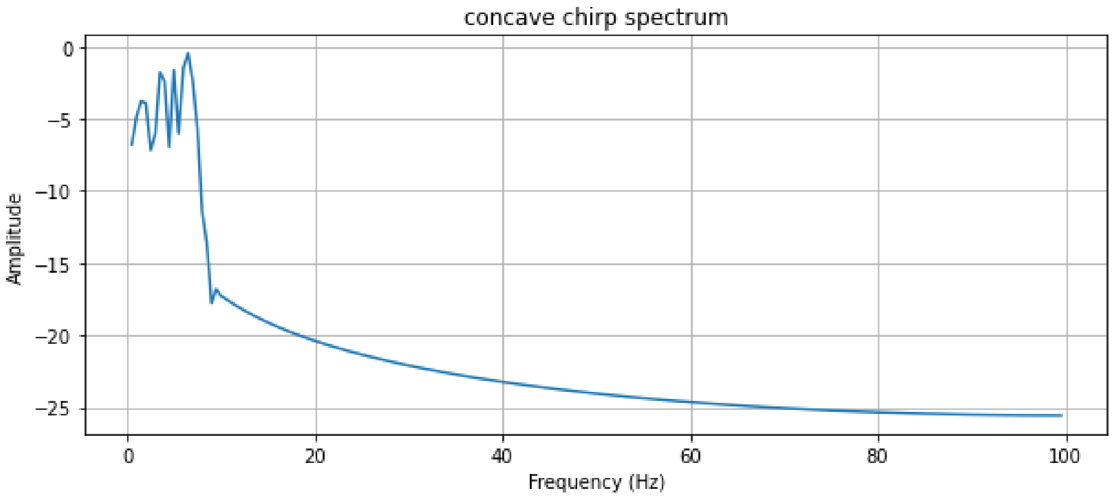
```
pwelch(logarithmic_chirp, 200)
```

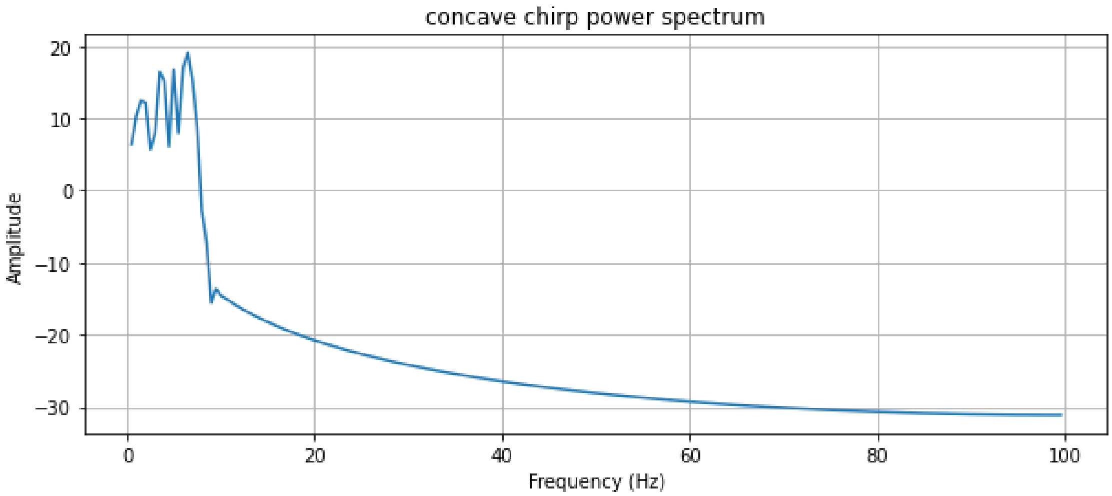


- From the above plots for power spectra of the logarithmic chirp signal, it can be seen that frequencies have positive amplitudes from 0 to 11 Hz whereas the signal has frequencies ranging from 0 Hz to 5.5 Hz (twice the max frequency of signal as fft is symmetric and gives equal response for both positive and negative frequencies).

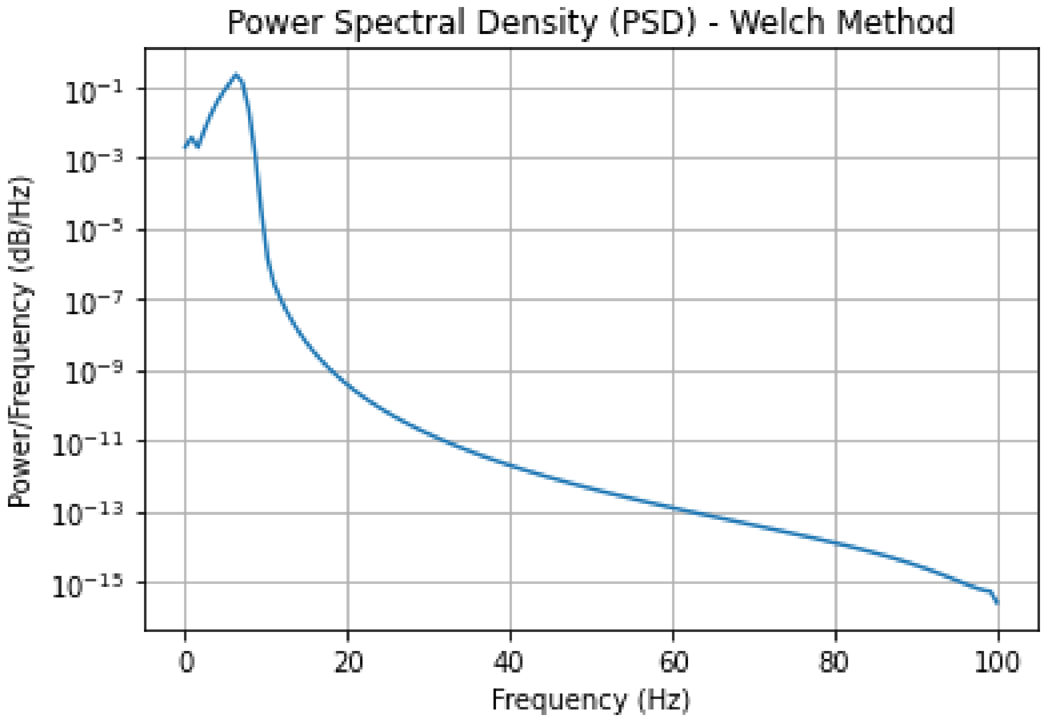
[217]:

```
concave_frequencies, concave_spectrum, concave_pos =␣
 ↪compute_spectrum(concave_chirp, 'concave_chirp', sampling_frequency)
concave_power_frequencies, concave_power_spectrum, concave_power_pos =␣
 ↪compute_power_spectrum(concave_chirp, 'concave_chirp', sampling_frequency)

plot_spectrum(concave_frequencies, 'concave chirp spectrum', concave_spectrum)
plot_spectrum(concave_power_frequencies, 'concave chirp power spectrum',␣
 ↪concave_power_spectrum)
```





[218]:
```
pwelch(concave_chirp, 200)
```



- From the above plots for power spectra of the quadratic chirp signal, it can be seen that frequencies have positive amplitudes from 0 to 10 Hz whereas the signal has frequencies ranging from 0 Hz to 5 Hz (twice the max frequency of signal as fft is symmetric and gives equal response for both positive and negative frequencies).

## 3 Tapers

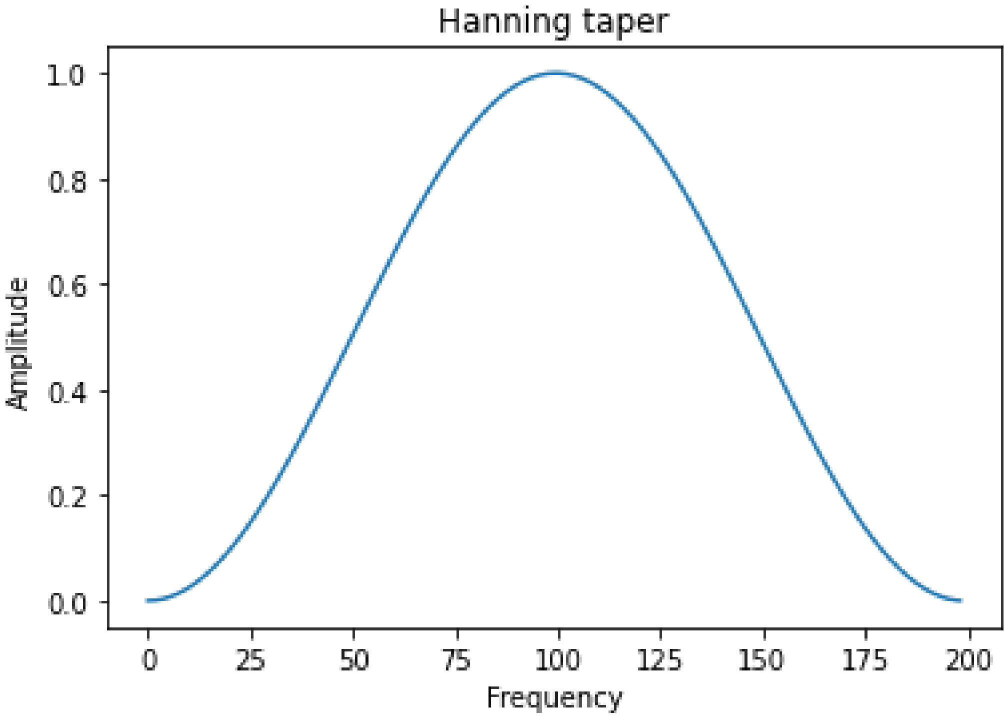
- In the below plots we can observe that the rectangular taper preservers the signal and accounts for all the frequencies and the amplitude while the hamming taper modifies the amplitude of the signal according to normal distribution resulting in the low amplitude of leftmost and rightmost part of signal. Hanning taper also does the same as hamming the only difference is that hamming window accounts for the leftmost and rightmost part of signal with some positive probability while the hanning taper takes it as zero and totally cut off those parts.

[219]:

```
# Compute Spectra with Hanning, Rectangular, and Hamming tapers
import numpy as np
from scipy.signal import get_window
def wave_stats(data):
    n = len(data)
    mean = sum(data) / n
    variance = sum((x - mean) ** 2 for x in data) / (n - 1)
    return mean, variance
```

```python
# Number of points for FFT
nfft = 199
hanning_taper = get_window('hann', nfft)
rectangular_taper = get_window('boxcar', nfft)
hamming_taper = get_window('hamming', nfft)


plt.plot(hanning_taper)
plt.title("Hanning taper")
plt.xlabel('Frequency')
plt.ylabel('Amplitude');
mean, variance = wave_stats(hanning_taper)
print("Mean of wave: "+str(mean)+" , variance of wave: "+str(variance))
```

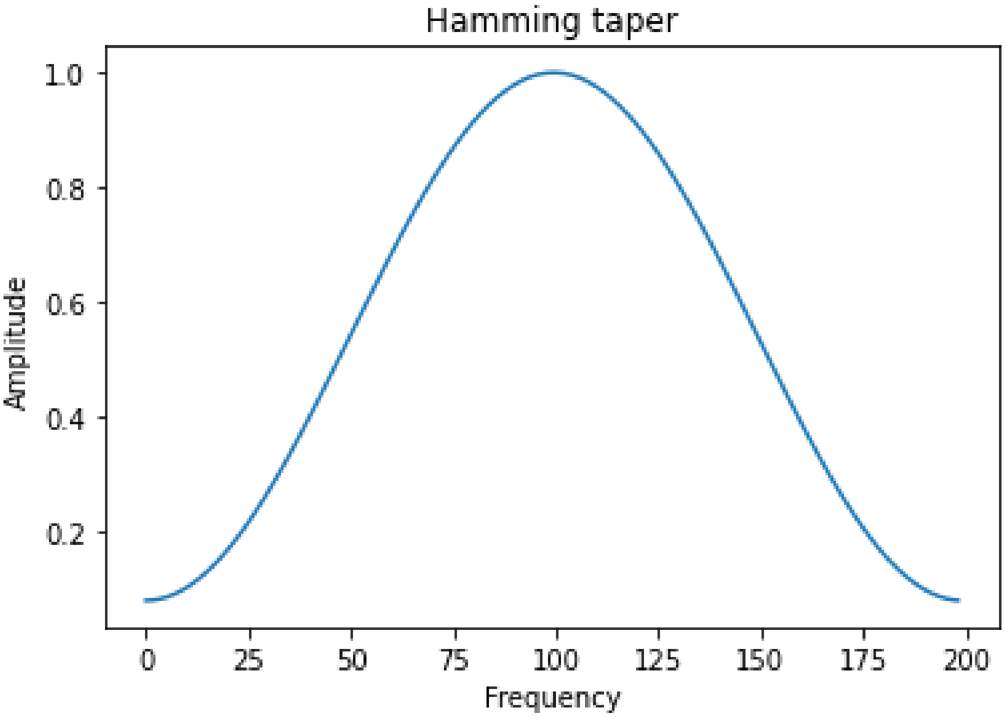
```
Mean of wave: 0.5 , variance of wave: 0.12563131313131307
```



[220]:

```python
plt.plot(hamming_taper)
plt.title("Hamming taper")
plt.xlabel('Frequency')
plt.ylabel('Amplitude');
mean, variance = wave_stats(hamming_taper)
print("Mean of wave: "+str(mean)+" , variance of wave: "+str(variance))
```

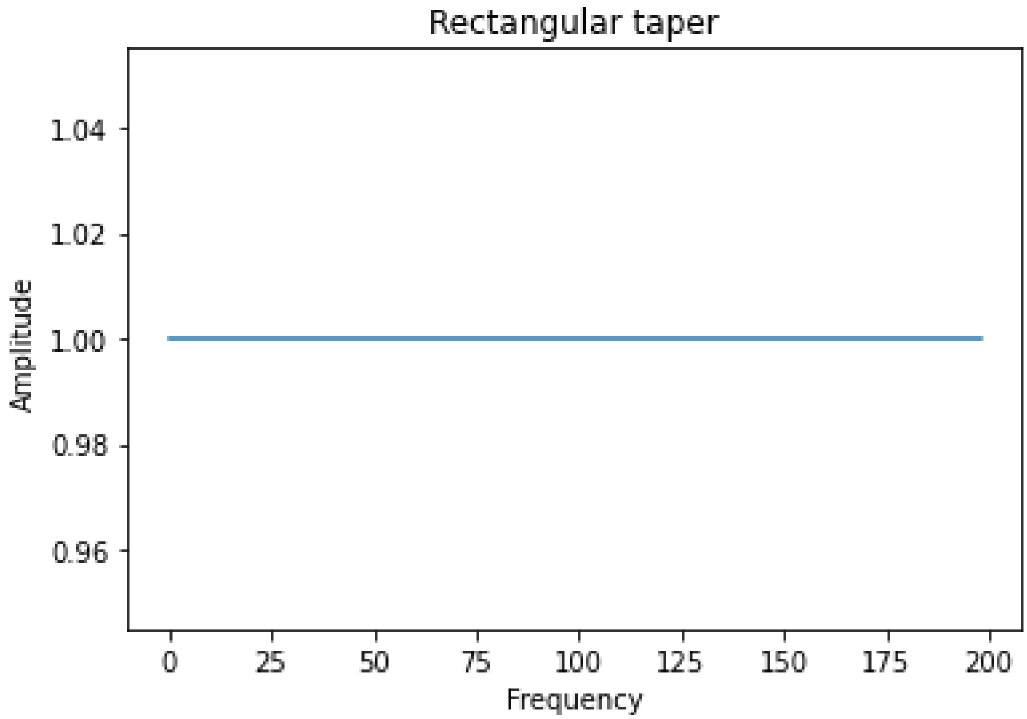
```
Mean of wave: 0.5400000000000003 , variance of wave: 0.10633434343434339
```



[221]:

```
plt.plot(rectangular_taper)
plt.title("Rectangular taper")
plt.xlabel('Frequency')
plt.ylabel('Amplitude');
mean, variance = wave_stats(rectangular_taper)
print("Mean of wave: "+str(mean)+" , variance of wave: "+str(variance))
```

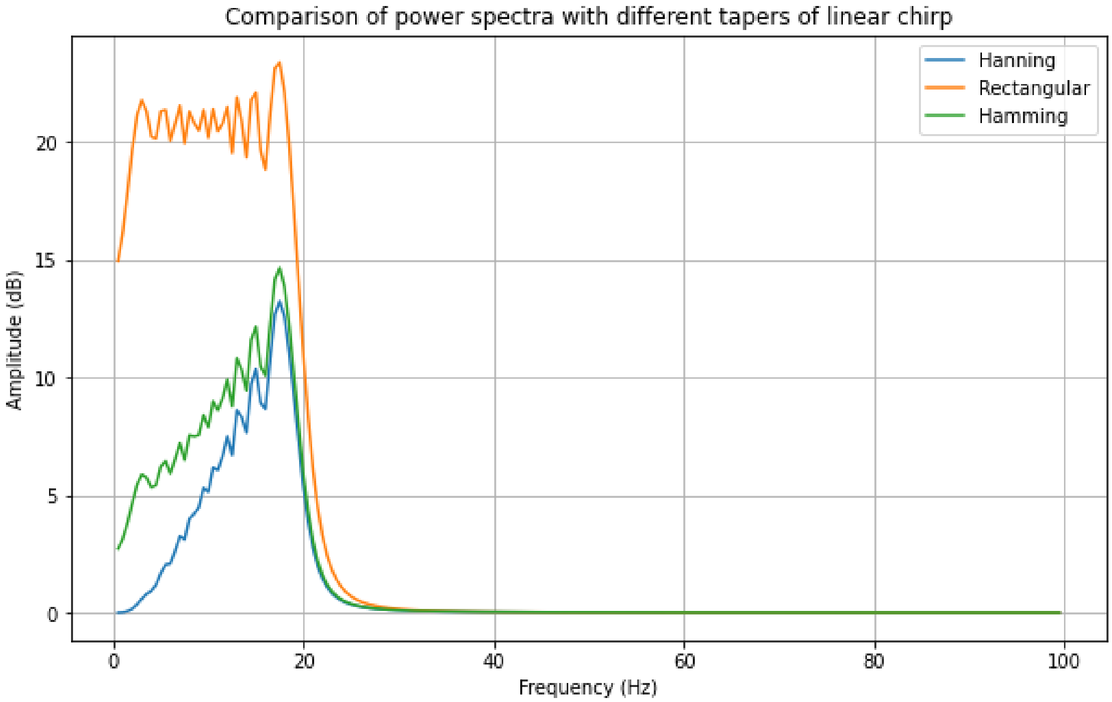
```
Mean of wave: 1.0 , variance of wave: 0.0
```



[222]:

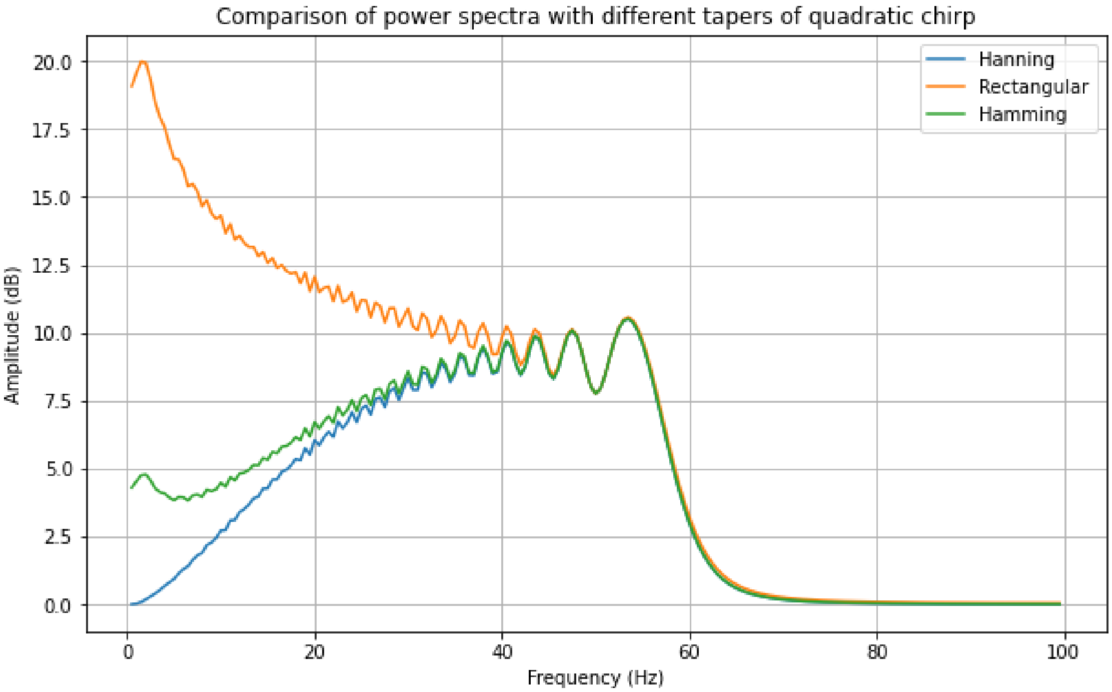
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import get_window, welch




hanning_spectrum = np.multiply(linear_power_spectrum, hanning_taper)
rectangular_spectrum = np.multiply(linear_power_spectrum, rectangular_taper)
hamming_spectrum = np.multiply(linear_power_spectrum, hamming_taper)

# Frequency axis
#freq = np.fft.fftfreq(nfft, 1/fs)

# Plot the spectra
plt.figure(figsize=(10, 6))
plt.plot(linear_frequencies, 20 * np.log10(1 + hanning_spectrum),␣
 ↪label='Hanning')
plt.plot(linear_frequencies, 20 * np.log10(1 + rectangular_spectrum),␣
 ↪label='Rectangular')
plt.plot(linear_frequencies, 20 * np.log10(1 + hamming_spectrum),␣
 ↪label='Hamming')
```

```
plt.title('Comparison of power spectra with different tapers of linear chirp')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude (dB)')
plt.legend()
plt.grid(True)
plt.show()
```



[223]:
```
hanning_spectrum = np.multiply(quadratic_power_spectrum, hanning_taper)
rectangular_spectrum = np.multiply(quadratic_power_spectrum, rectangular_taper)
hamming_spectrum = np.multiply(quadratic_power_spectrum, hamming_taper)

# Frequency axis
#freq = np.fft.fftfreq(nfft, 1/fs)

# Plot the spectra
plt.figure(figsize=(10, 6))
plt.plot(quadratic_power_frequencies, 20 * np.log10(1 + hanning_spectrum),␣
 ↪label='Hanning')
plt.plot(quadratic_power_frequencies, 20 * np.log10(1 + rectangular_spectrum),␣
 ↪label='Rectangular')
plt.plot(quadratic_power_frequencies, 20 * np.log10(1 + hamming_spectrum),␣
 ↪label='Hamming')
```

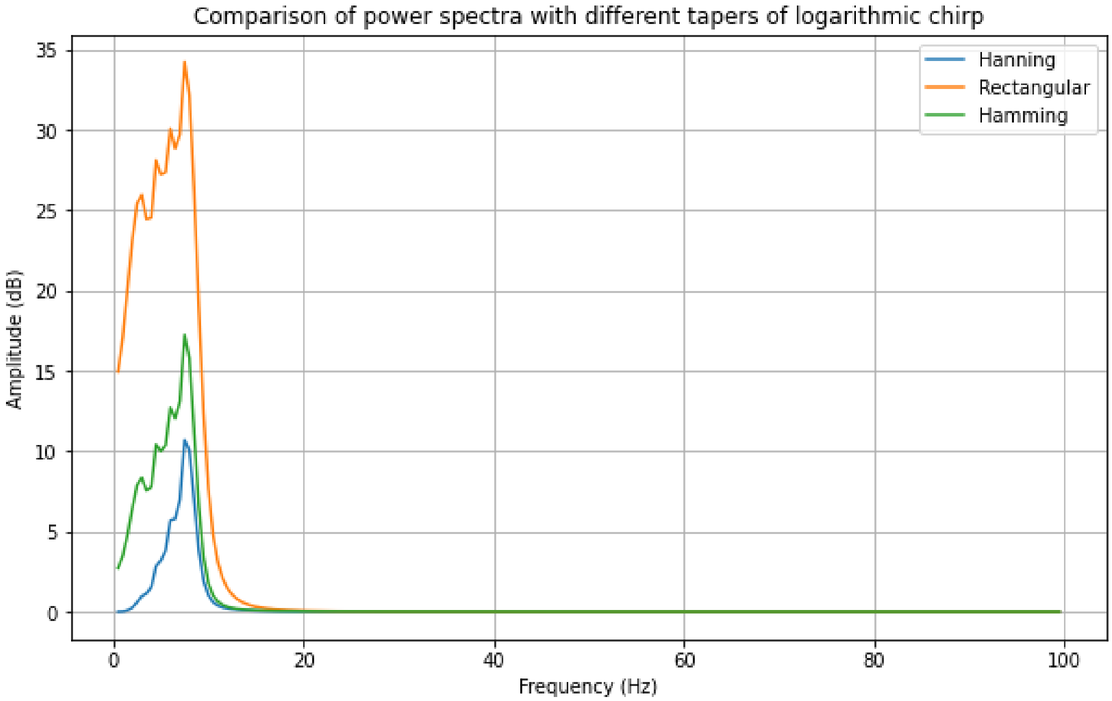
```
plt.title('Comparison of power spectra with different tapers of quadratic␣
  ↪chirp')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude (dB)')
plt.legend()
plt.grid(True)
plt.show()
```



[224]:

```
hanning_spectrum = np.multiply(logarithmic_power_spectrum, hanning_taper)
rectangular_spectrum = np.multiply(logarithmic_power_spectrum,␣
  ↪rectangular_taper)
hamming_spectrum = np.multiply(logarithmic_power_spectrum, hamming_taper)

# Frequency axis
#freq = np.fft.fftfreq(nfft, 1/fs)

# Plot the spectra
plt.figure(figsize=(10, 6))
plt.plot(logarithmic_power_frequencies, 20 * np.log10(1 + hanning_spectrum),␣
  ↪label='Hanning')
plt.plot(logarithmic_power_frequencies, 20 * np.log10(1 +␣
  ↪rectangular_spectrum), label='Rectangular')
plt.plot(logarithmic_power_frequencies, 20 * np.log10(1 + hamming_spectrum),␣
  ↪label='Hamming')
```

```
plt.title('Comparison of power spectra with different tapers of logarithmic␣
 ↪chirp')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Amplitude (dB)')
plt.legend()
plt.grid(True)
plt.show()
```



## 4 Time frequency resolution

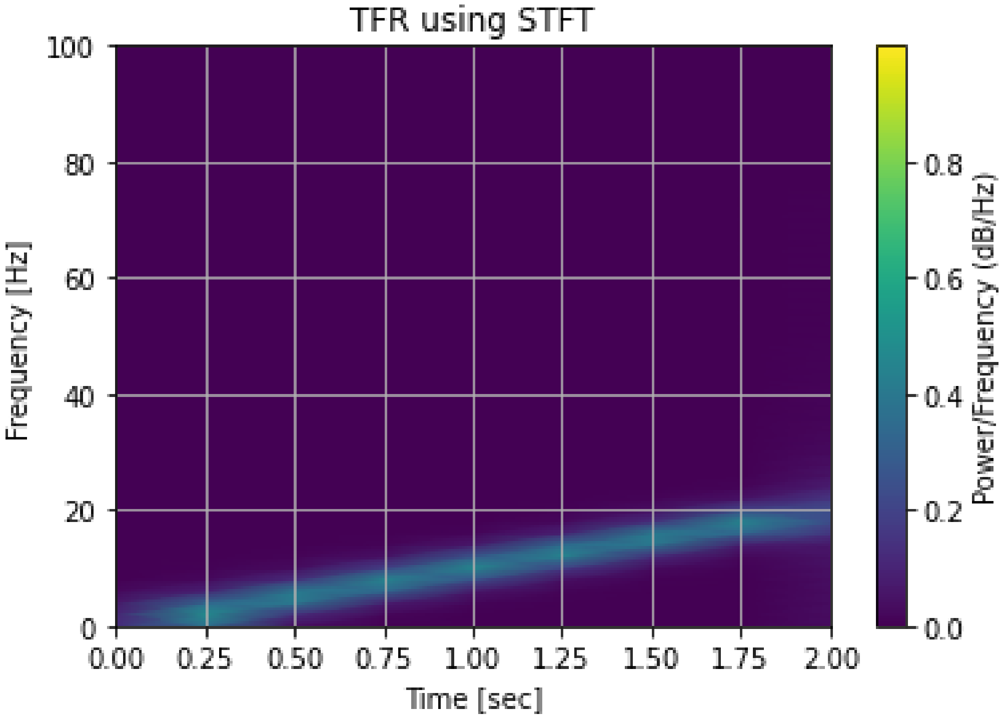
- Maximum frequency can be observed at t = 2 seconds for linear, quadratic and logarithmic chirps as they are monotonically increasing functions, whereas the concave chirp has max frequency at t = 1 second.
- The power spectrums are comparable across all the methods.
- Frequency resolution = **nyquist_frequency / np.abs(Z).shape**[0] where Z is magnitude of tfr.
- From the plots, if window size is increased, the frequency resolution is increased. As time window increases, the time resolution decreases, hence, as frequency = **1/time**, frequency resolution increases.

### 4.1 STFT

```
[72]: from scipy import signal
def plot_stft(wave, fs, window_size):
    A_max = max(wave)
    f, t, Z = signal.stft(wave, fs, nperseg=window_size)
    plt.pcolormesh(t, f, np.abs(Z), vmin=0, vmax=A_max, shading='gouraud')
    nyquist_frequency = fs / 2
    frequency_resolution_stft = nyquist_frequency / np.abs(Z).shape[0]

    print(f'Frequency Resolution (STFT): {frequency_resolution_stft} Hz')
    plt.title('TFR using STFT')
    plt.ylabel('Frequency [Hz]')
    plt.xlabel('Time [sec]')
    plt.colorbar(label='Power/Frequency (dB/Hz)')
    plt.grid()
    plt.show()
```

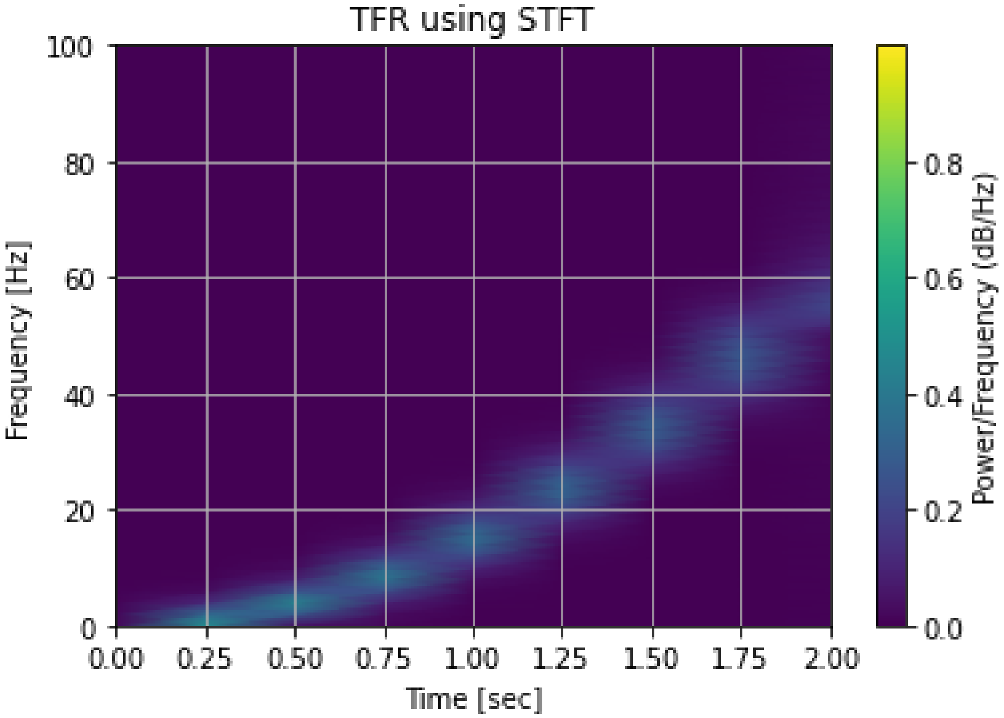
```
[19]: plot_stft(linear_chirp,200, 100)
```

```
Frequency Resolution (STFT): 1.9607843137254901 Hz
```



```
[22]: plot_stft(quadratic_chirp, 200, 100)
```

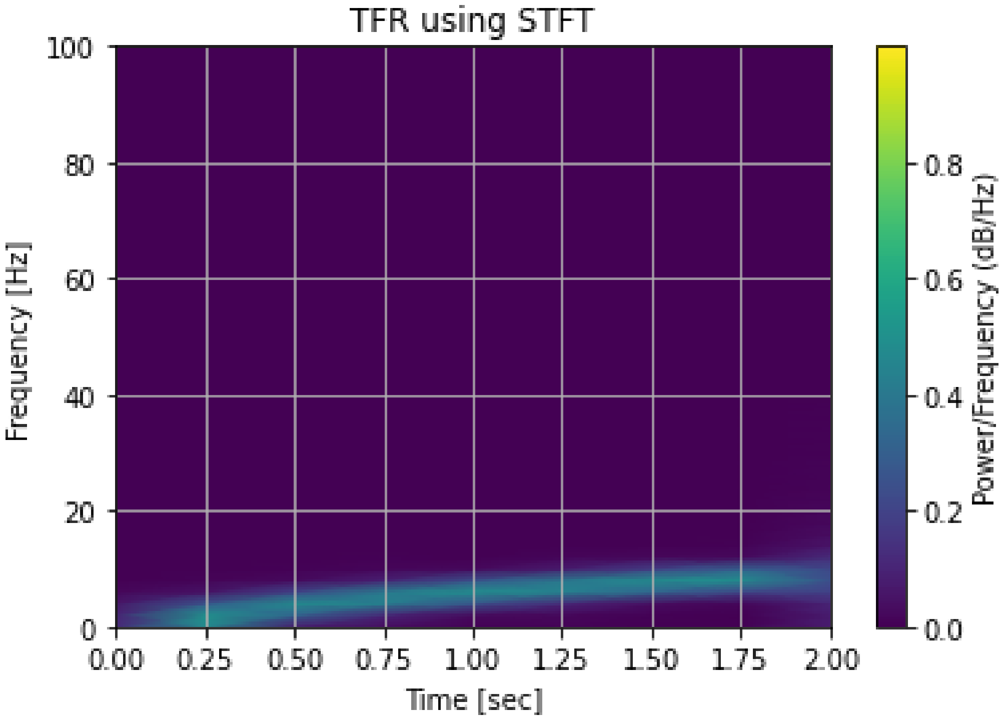
```
Frequency Resolution (STFT): 1.9607843137254901 Hz
```



- Maximum frequency can be observed at t = 2 seconds

[227]:
```
plot_stft(logarithmic_chirp, 200, 100)
```

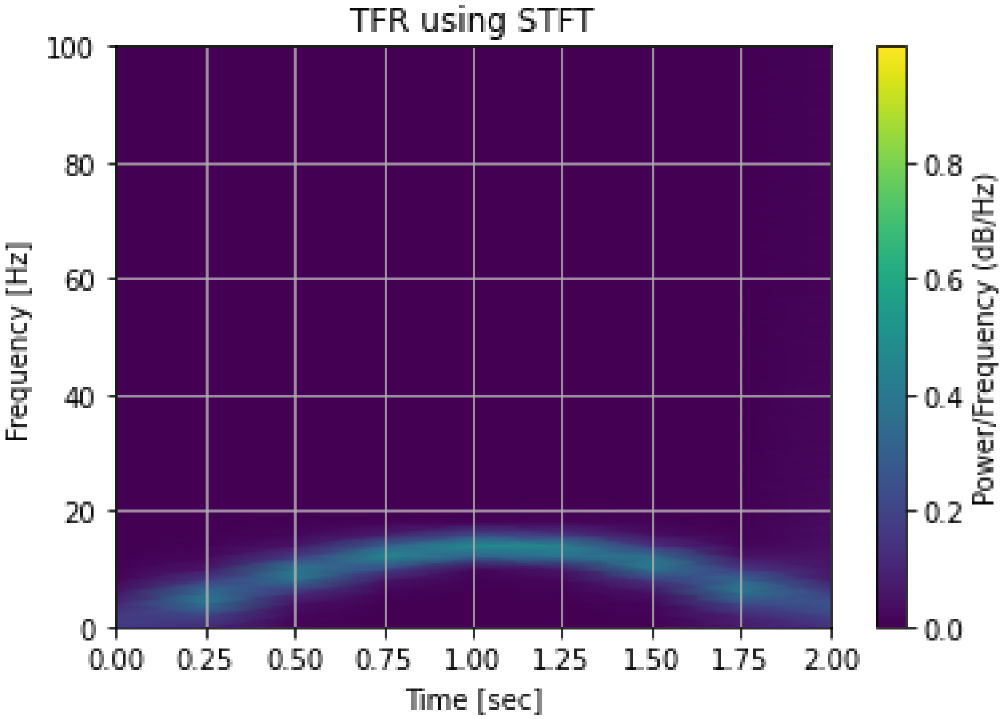
```
Frequency Resolution (STFT): 1.9607843137254901 Hz
```



- Maximum frequency can be observed at t = 2 seconds

[24]:
```
plot_stft(concave_chirp, 200, 100)
```

```
Frequency Resolution (STFT): 1.9607843137254901 Hz
```



### 4.2 Multitaper

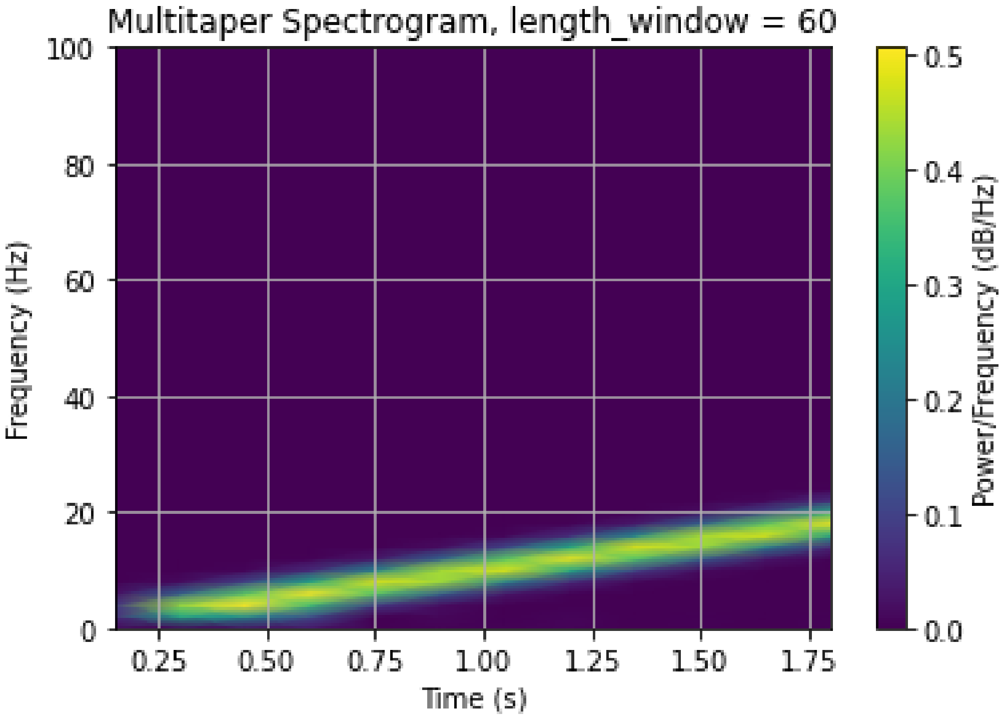
[129]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import spectrogram, windows

# Generate a sample signal
fs = 150  # Sampling frequency
t = np.arange(0, 2, 1/fs)  # 5 seconds of data
f_signal = 5 * (t**2)  # Signal frequency
signal = np.sin(2 * np.pi * f_signal * t)

def multitaper_decomposition(signal, fs, nperseg):
# Multitaper decomposition parameters
      # Length of each segment
    noverlap = nperseg // 2  # Overlap between segments
    nfft = 100  # Number of points for the FFT
    window = windows.dpss(nperseg, 2)  # DPSS window with time-bandwidth␣
 ↪product 4.0

    # Compute the multitaper spectrogram
    frequencies, times, Sxx = spectrogram(signal, fs=fs, nperseg=nperseg,␣
 ↪noverlap=noverlap, nfft=nfft, window=window, scaling='spectrum')
```

```
    nyquist = fs/2
    freq_res = nyquist/nperseg
    print("Frequency resolution: " + str(freq_res)+"Hz")
    # Plot the results
    plt.pcolormesh(times, frequencies, np.abs(Sxx), shading='gouraud')
    plt.ylabel('Frequency (Hz)')
    plt.xlabel('Time (s)')
    plt.title('Multitaper Spectrogram, length_window = '+str(nperseg))
    plt.colorbar(label='Power/Frequency (dB/Hz)')
    plt.grid()
    plt.show()
```

[132]:

```
multitaper_decomposition(linear_chirp, 200, 60)
```

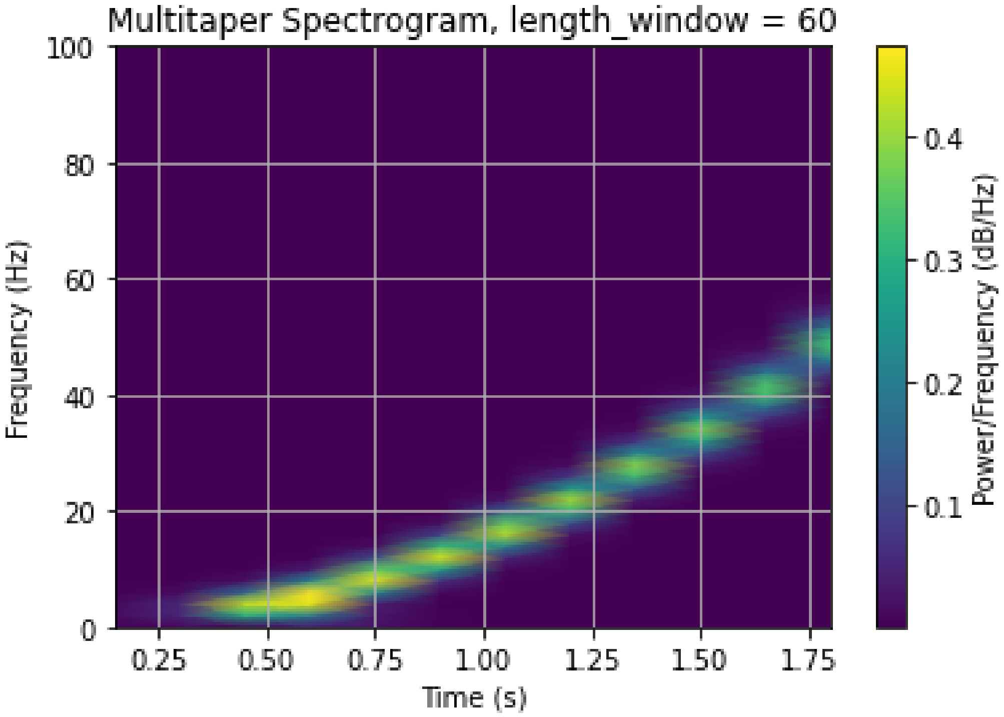
```
Frequency resolution: 1.6666666666666667Hz
```



[133]:

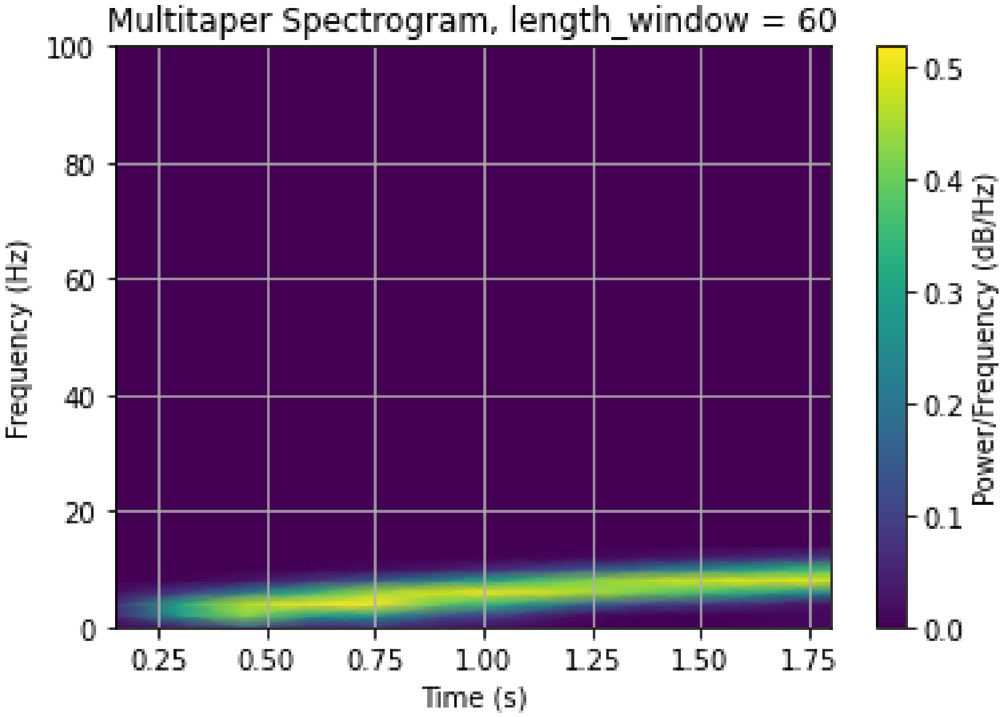
```
multitaper_decomposition(quadratic_chirp, 200, 60)
```

```
Frequency resolution: 1.6666666666666667Hz
```



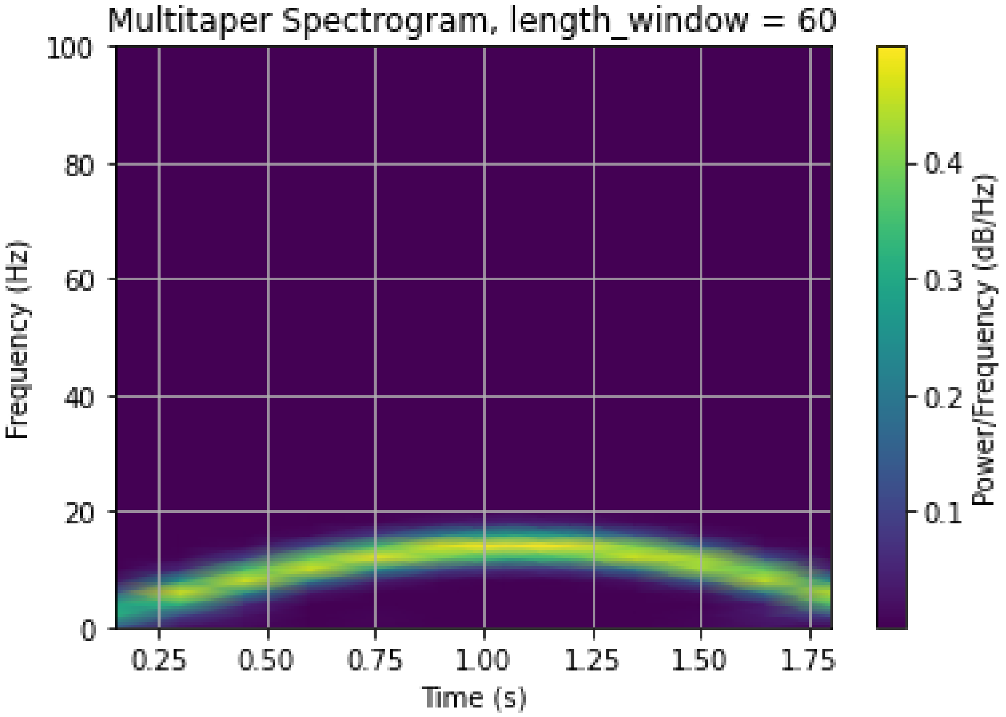
[134]:
```
multitaper_decomposition(logarithmic_chirp, 200, 60)
```

```
Frequency resolution: 1.6666666666666667Hz
```



```
[135]: multitaper_decomposition(concave_chirp, 200, 60)
```

```
Frequency resolution: 1.6666666666666667Hz
```



### 4.3 Wavelet decomposition

[25]:

```
!pip install PyWavelets matplotlib
```

```
Requirement already satisfied: PyWavelets in
/home/akhil/anaconda3/lib/python3.9/site-packages (1.3.0)
Requirement already satisfied: matplotlib in
/home/akhil/anaconda3/lib/python3.9/site-packages (3.7.2)
Requirement already satisfied: numpy>=1.17.3 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from PyWavelets) (1.25.1)
Requirement already satisfied: contourpy>=1.0.1 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (1.3.2)
Requirement already satisfied: packaging>=20.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (23.2)
Requirement already satisfied: pillow>=6.2.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (9.0.1)
Requirement already satisfied: pyparsing<3.1,>=2.3.1 in
```

```
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (3.0.4)
Requirement already satisfied: python-dateutil>=2.7 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (2.8.2)
Requirement already satisfied: importlib-resources>=3.2.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from matplotlib) (6.1.0)
Requirement already satisfied: zipp>=3.1.0 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from importlib-
resources>=3.2.0->matplotlib) (3.7.0)
Requirement already satisfied: six>=1.5 in
/home/akhil/anaconda3/lib/python3.9/site-packages (from python-
dateutil>=2.7->matplotlib) (1.16.0)

[notice] A new release of pip is
available: 23.3.1 -> 23.3.2
[notice] To update, run:
pip install --upgrade pip
```

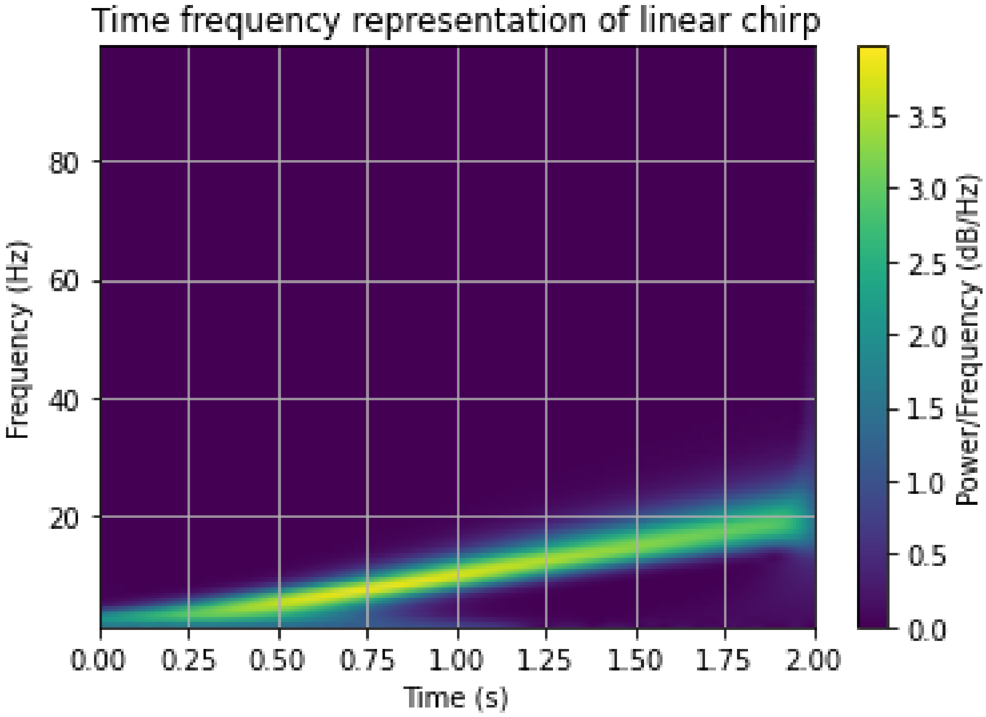
[136]:

```python
from scipy import signal
w = 6.
sig = linear_chirp

def wavelet_decomposition(sig, w):
    fs = 199.0
    t = np.linspace(0, 2, 400)
    freq = np.linspace(1, fs/2, 100)
    widths = w*fs / (2*freq*np.pi)
    cwtm = signal.cwt(sig, signal.morlet2, widths, w=w)
    nyquist = fs/2
    freq_res = nyquist/np.abs(cwtm).shape[0]
    print("Frequency resolution: " + str(freq_res)+"Hz")
    plt.pcolormesh(t, freq, np.abs(cwtm), cmap='viridis', shading='gouraud')
    plt.colorbar(label='Power/Frequency (dB/Hz)')
    plt.title('Time frequency representation of linear chirp')
    plt.ylabel('Frequency (Hz)')
    plt.xlabel('Time (s)')
    plt.grid()
    plt.show()
```

[137]:

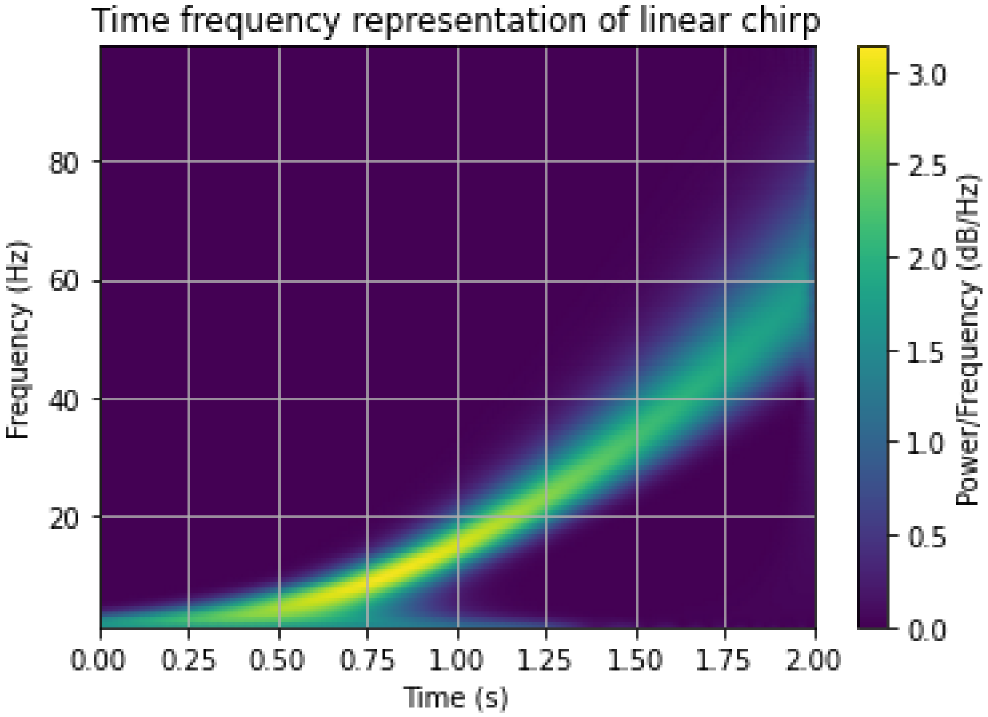
```python
wavelet_decomposition(linear_chirp, w)
```

```
Frequency resolution: 0.995Hz
```



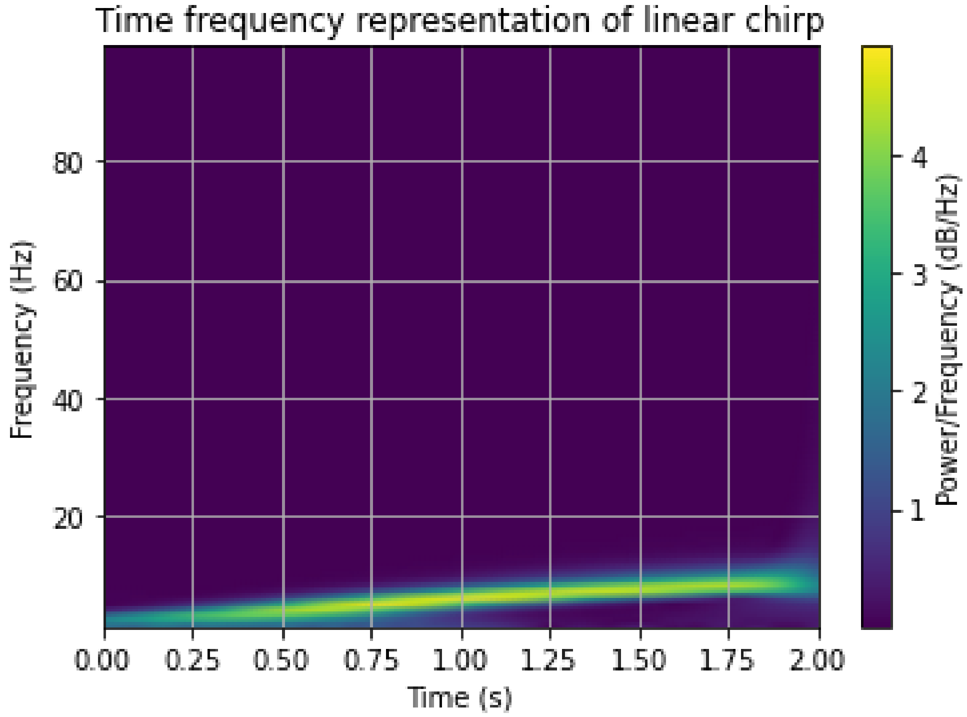
[138]:
```
wavelet_decomposition(quadratic_chirp, w)
```

```
Frequency resolution: 0.995Hz
```



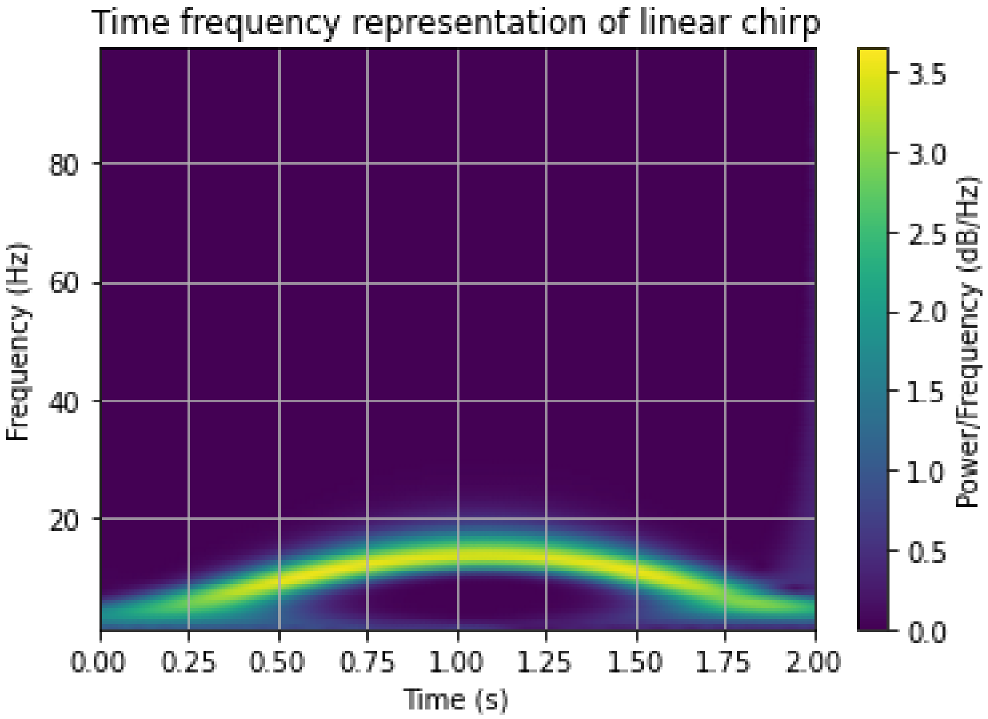
```
[139]: wavelet_decomposition(logarithmic_chirp, w)
```

```
Frequency resolution: 0.995Hz
```



[140]:
```
wavelet_decomposition(concave_chirp, w)
```

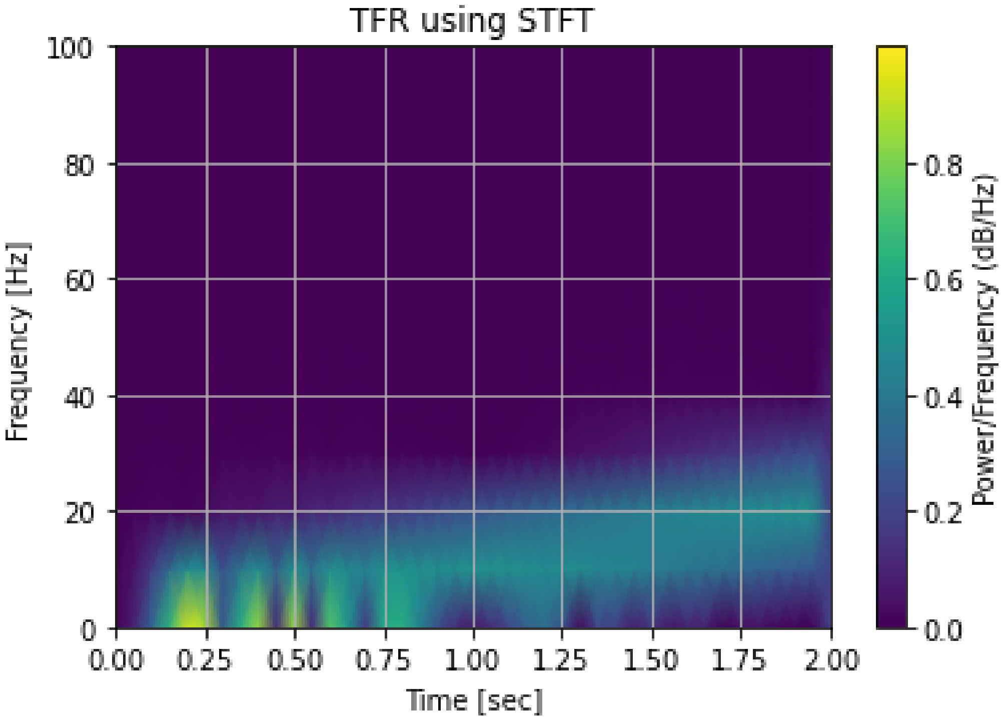
```
Frequency resolution: 0.995Hz
```



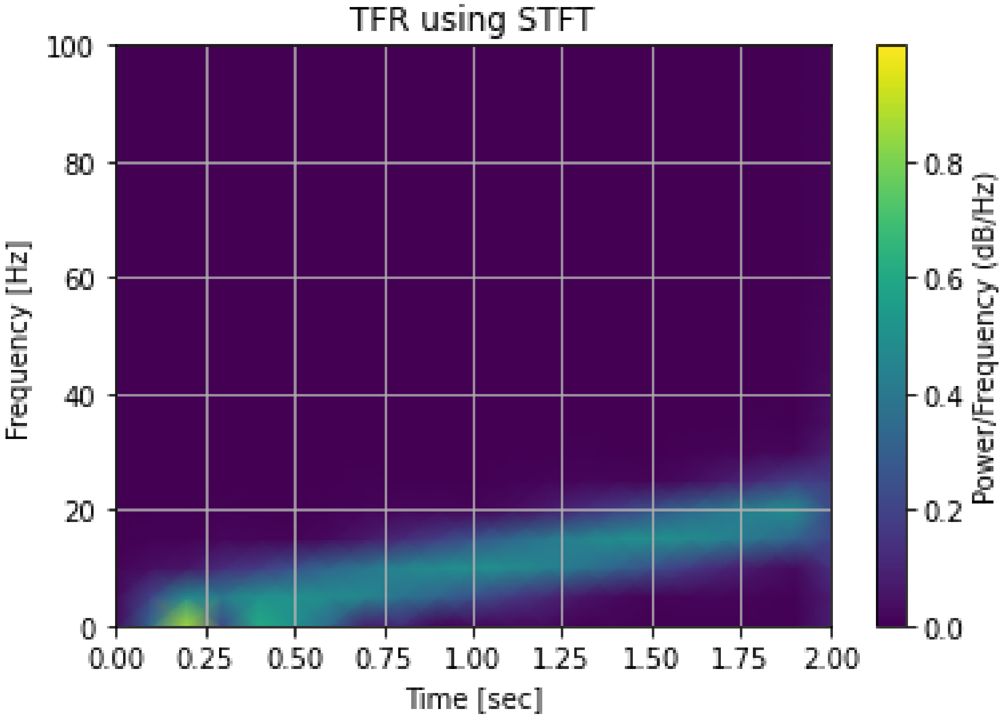
### 4.4 Changing window sizes for different methods for linear chirp

[141]:
```
window_sizes = [20, 40, 60, 80, 100]
for window_size in window_sizes:
    plot_stft(linear_chirp, 200, window_size)
```

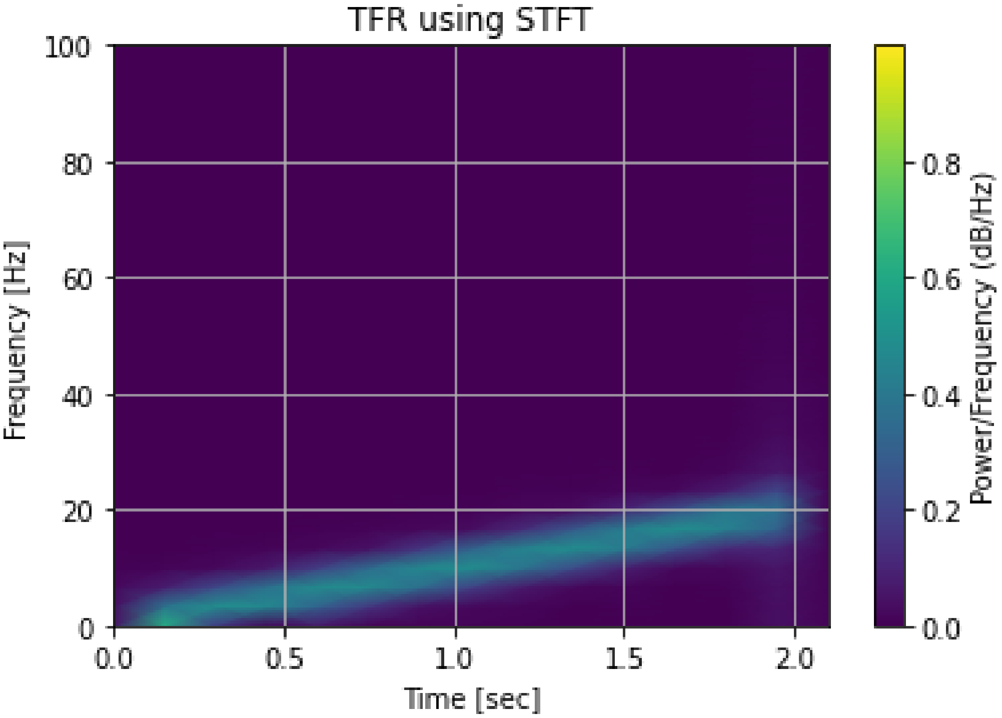
```
Frequency Resolution (STFT): 9.090909090909092 Hz
```



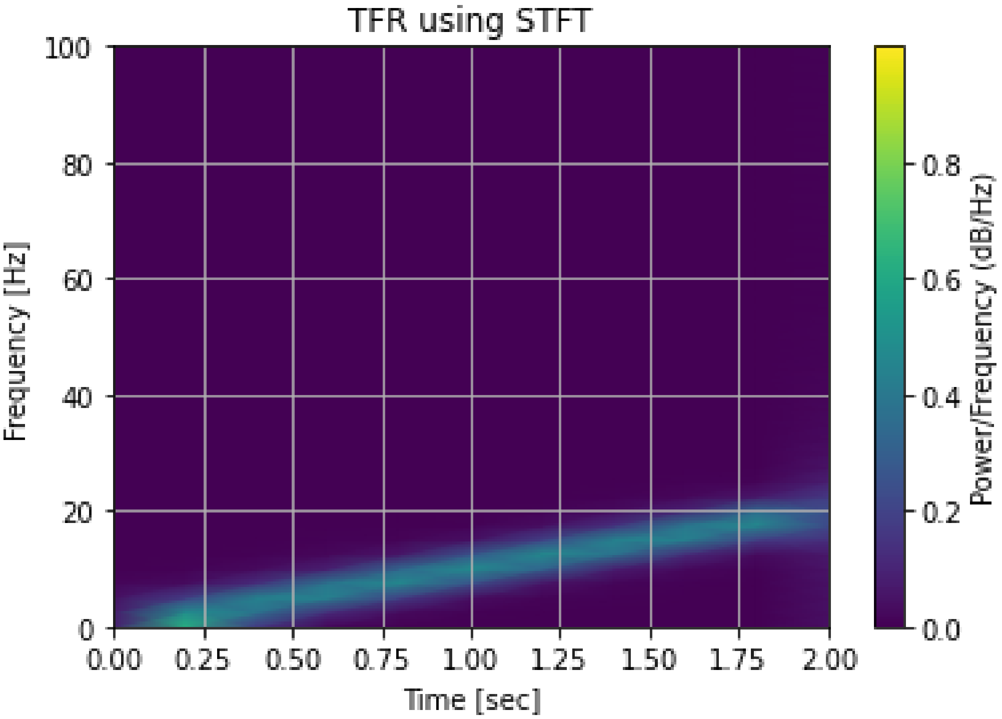
```
Frequency Resolution (STFT): 4.761904761904762 Hz
```



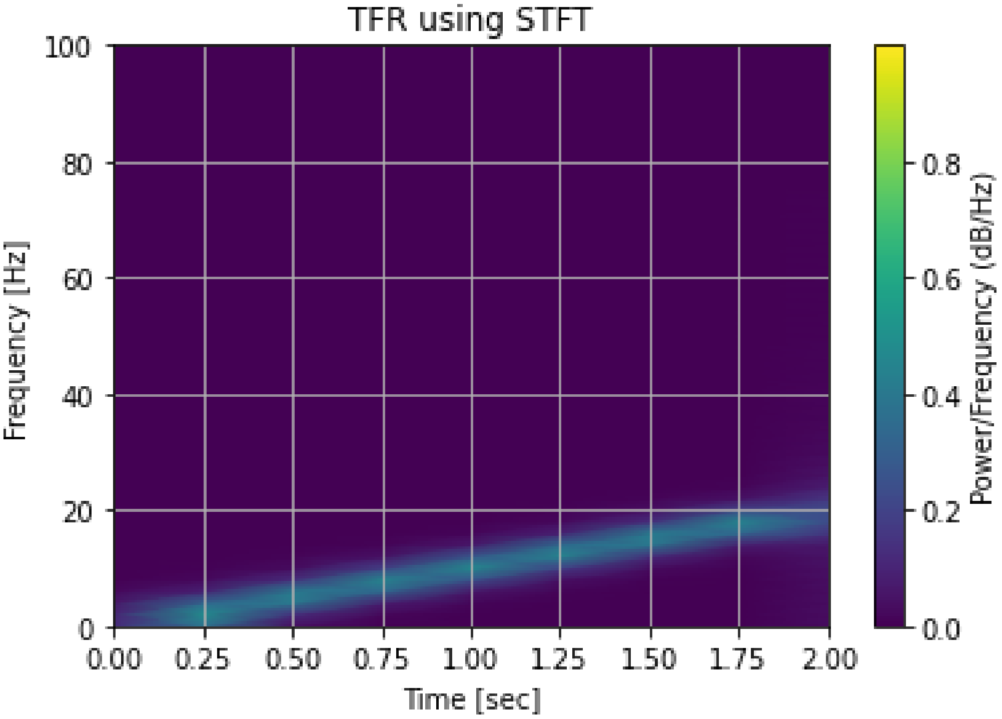
```
Frequency Resolution (STFT): 3.225806451612903 Hz
```



```
Frequency Resolution (STFT): 2.4390243902439024 Hz
```

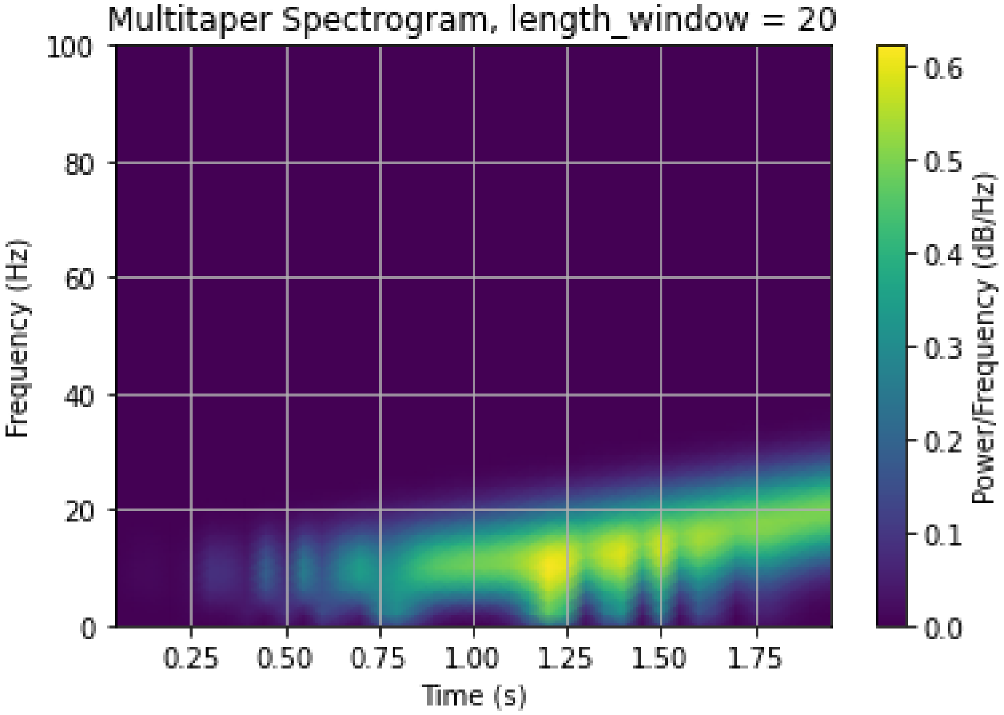


```
Frequency Resolution (STFT): 1.9607843137254901 Hz
```

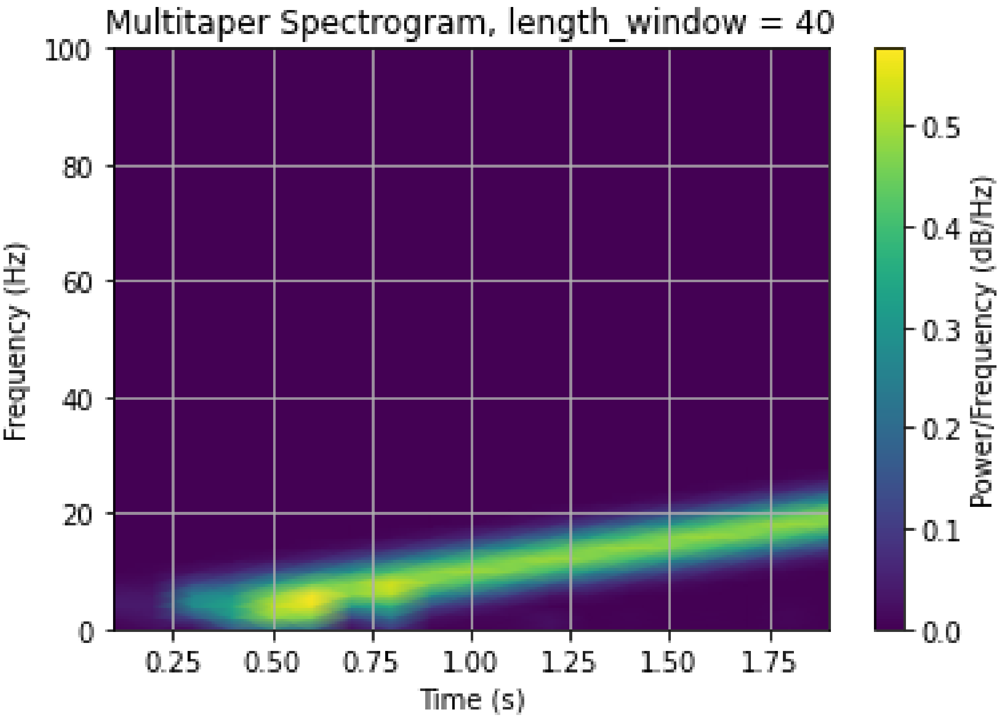


```
[142]: for window_size in window_sizes:
           multitaper_decomposition(linear_chirp, 200, window_size)
```

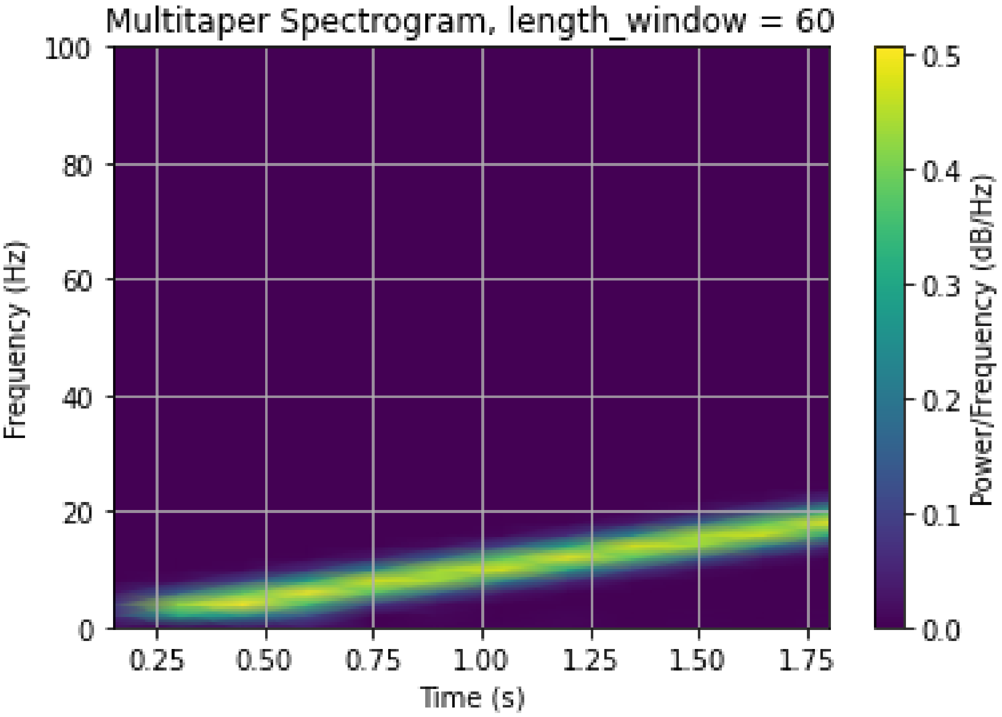
```
Frequency resolution: 5.0Hz
```



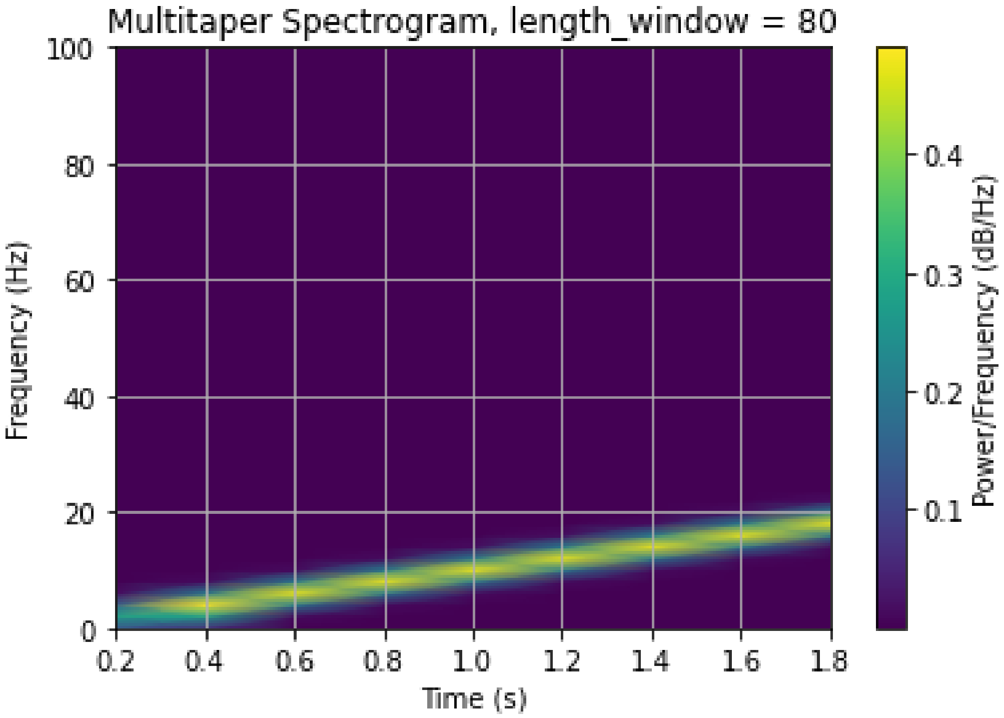
```
Frequency resolution: 2.5Hz
```



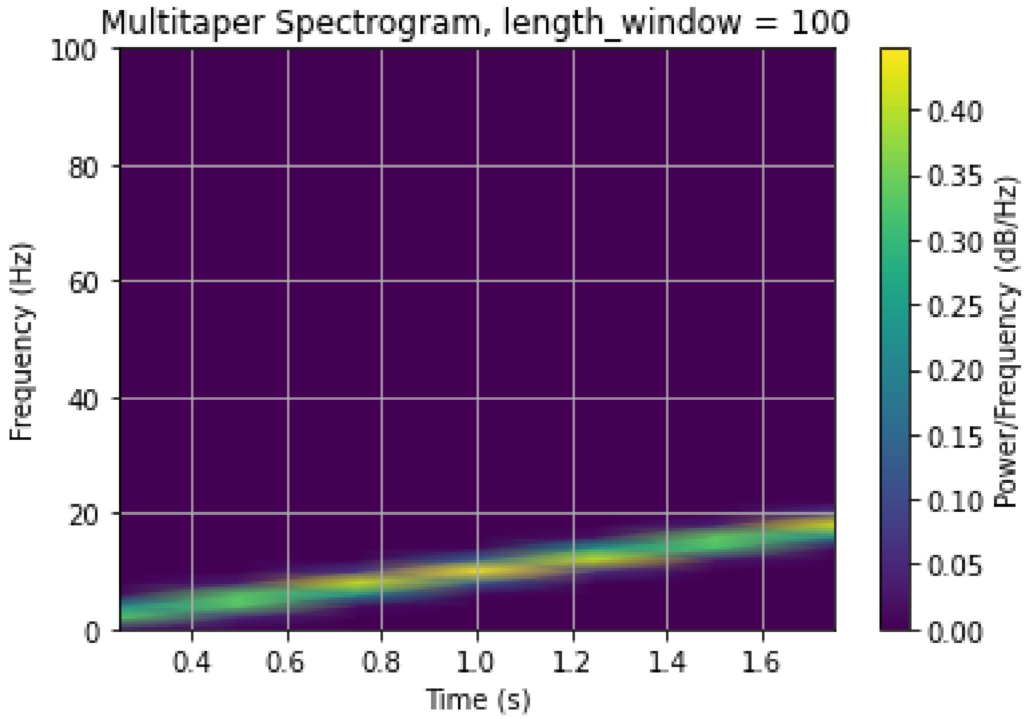
```
Frequency resolution: 1.6666666666666667Hz
```



```
Frequency resolution: 1.25Hz
```



```
Frequency resolution: 1.0Hz
```

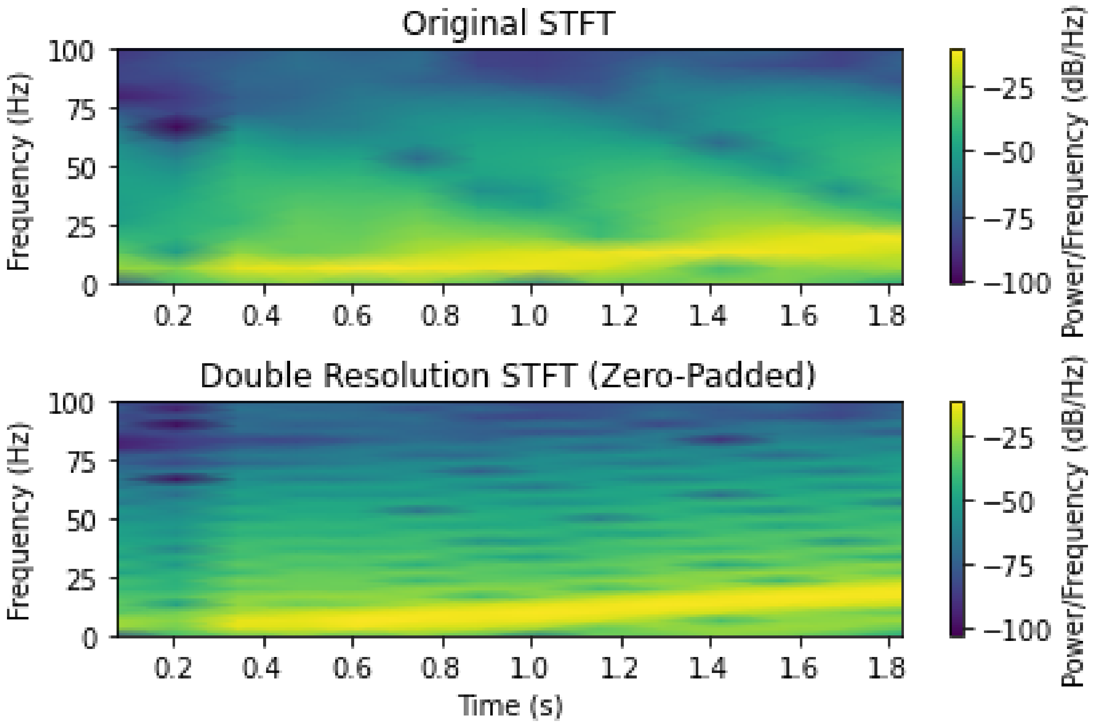


### 4.5 Doubling frequency resolution by padding

[151]:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import spectrogram

def stft_double_frequency_resolution(signal, fs, nperseg):
    # Calculate the STFT with zero-padding
    nfft = nperseg * 2  # Double the number of points in the FFT
    f, t, Sxx = spectrogram(signal, fs=fs, nperseg=nperseg, nfft=nfft)

    nfft_orig = nperseg  # Double the number of points in the FFT
    f_old, t_old, Sxx_old = spectrogram(signal, fs=fs, nperseg=nperseg,␣
 ↪nfft=nfft_orig)
    # Plot the original and double resolution STFT
    plt.subplot(2, 1, 1)
    plt.pcolormesh(t_old, f_old, 10 * np.log10(Sxx_old), shading='gouraud',␣
 ↪cmap='viridis')
    plt.title('Original STFT')
    plt.ylabel('Frequency (Hz)')
    plt.colorbar(label='Power/Frequency (dB/Hz)')
    nyquist_frequency = fs / 2
    frequency_resolution_stft_old = nyquist_frequency / np.abs(Sxx_old).shape[0]
    print("Frequency resolution without padding: " + str(frequency_resolution_stft_old))
    plt.subplot(2, 1, 2)
    plt.pcolormesh(t, f, 10 * np.log10(Sxx), shading='gouraud', cmap='viridis')
    plt.title('Double Resolution STFT (Zero-Padded)')
    plt.xlabel('Time (s)')
    plt.ylabel('Frequency (Hz)')
    plt.colorbar(label='Power/Frequency (dB/Hz)')
    frequency_resolution_stft = nyquist_frequency / np.abs(Sxx).shape[0]
    print("Frequency resolution after padding: " + str(frequency_resolution_stft))
    plt.tight_layout()
    plt.show()

# Example usage:
# Replace the following with your actual signal, sample rate, and desired window length
# signal = ...
# fs = ...
# nperseg = ...

# Call the function
stft_double_frequency_resolution(linear_chirp, 200, 30)
```

```
Frequency resolution without padding: 6.25
Frequency resolution after padding: 3.225806451612903
```


Original STFT
Frequency (Hz)
Power/Frequency (dB/Hz)
Double Resolution STFT (Zero-Padded)
Frequency (Hz)
Time (s)
Power/Frequency (dB/Hz)