

Intelligent Signal Processing Python examples

May 25, 2023

1 Python for Signal Processing

Danilo Greco, PhD - danilo.greco@uniparthenope.it - University of Naples Parthenope

1.0.1 Example 1 - Audio Processing -

This is a Python code that does some audio processing. It first generates an audio signal and adds some noise to it using the normal distribution. Then, it designs a finite impulse response (FIR) filter https://en.wikipedia.org/wiki/Finite_impulse_response using the SciPy signal processing library. After that, it applies the filter to the noisy signal using the `lfilter` function from the same library. Finally, it plots both the noisy signal and filtered signal using Matplotlib for visualization. This is a simple example of signal processing that showcases the power of Python and its scientific libraries. The given Python code generates an audio signal using the NumPy library. It first creates a time array `t` using the `linspace` function, which generates 500 evenly spaced values between 0 and 1. It then sets the amplitude of the signal to 0.5. The resulting audio signal is a sine wave with a frequency of 1 Hz and an amplitude of 0.5. The code then imports the `firwin` and `lfilter` functions from the SciPy library, which are used to create and apply a finite impulse response (FIR) filter to the audio signal. The code imports the `pyplot` module from the Matplotlib library, which is used to plot the original and filtered audio signals

1. *import numpy as np*: This line imports the NumPy library and gives it the alias `np`, which is a commonly used abbreviation for NumPy.
2. *from scipy.signal import firwin, lfilter*: This line imports two functions, `firwin` and `lfilter`, from the `scipy.signal` module. These functions will be used for filtering the audio signal.
3. *import matplotlib.pyplot as plt*: This line imports the `pyplot` submodule from the `matplotlib` library and gives it the alias `plt`, which will be used for plotting the audio signal before and after filtering.
4. *t = np.linspace(0, 1, 500, False)*: This line generates an array of 500 equally spaced points between 0 and 1 using the `np.linspace()` function and assigns it to the variable `t`.
5. *x = 0.5 * np.sin(2 * np.pi * 50 * t) + np.sin(2 * np.pi * 120 * t)*: This line generates a audio signal consisting of two sine waves of frequencies 50 Hz and 120 Hz and assigns it to the variable `x`.
6. *noise = np.random.normal(0, 1, 500)*: This line generates an array of 500 normally distributed random numbers with a mean of 0 and standard deviation of 1 using the `np.random.normal()` function and assigns it to the variable `noise`.
7. *xn = x + noise*: This line adds the noise signal to the original audio signal `x` and assigns the resulting signal to the variable `xn`.
8. *b = firwin(51, 0.3)*: This line designs a low-pass filter with a cutoff frequency of 0.3 using the `firwin()` function and assigns the filter coefficients to the variable `b`.
9. *yn = lfilter(b, 1, xn)*: This line filters the noisy audio signal `xn` using the filter coefficients `b` and assigns the filtered signal to the variable `yn`.
10. *plt.plot(t, xn, 'b', label='Noisy Signal')*: This line plots the noisy audio signal `xn`

in blue color with a label of “Noisy Signal” using the `plot()` function from `pyplot`. 11. `plt.plot(t, yn, 'r', label='Filtered Signal')`: This line plots the filtered audio signal `yn` in red color with a label of “Filtered Signal” using the `plot()` function from `pyplot`. 12. `plt.legend()`: This line adds a legend to the plot using the `legend()` function from `pyplot`. 13. `plt.show()`: This line displays the plot on the screen using the `show()` function from `pyplot`.

```
[2]: import numpy as np
from scipy.signal import firwin, lfilter
import matplotlib.pyplot as plt

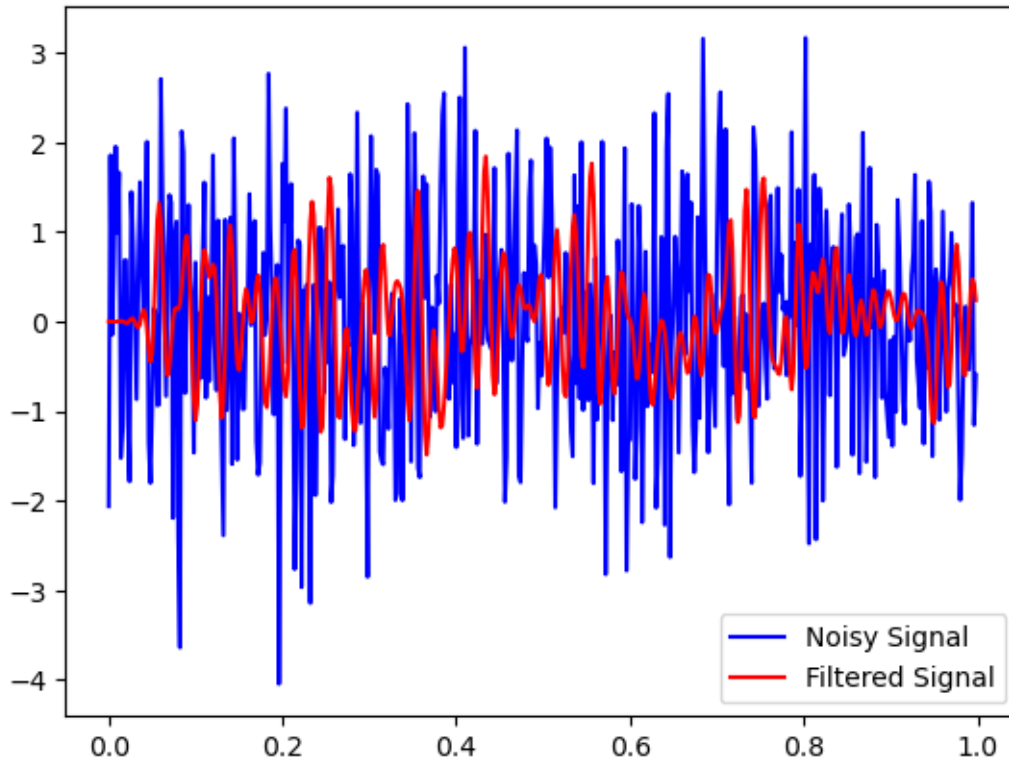
# Generating the audio signal
t = np.linspace(0, 1, 500, False)
x = 0.5 * np.sin(2 * np.pi * 50 * t) + np.sin(2 * np.pi * 120 * t)

# Adding noise to the audio signal
noise = np.random.normal(0, 1, 500)
xn = x + noise

# Designing the filter
b = firwin(51, 0.3)

# Filtering the audio signal
yn = lfilter(b, 1, xn)

# Plotting the audio signal before and after filtering
plt.plot(t, xn, 'b', label='Noisy Signal')
plt.plot(t, yn, 'r', label='Filtered Signal')
plt.legend()
plt.show()
```



1.0.2 Example 2 - FFT and IFFT -

This Python code generates a sine wave and applies the Fast Fourier Transform (FFT) to it. The code first imports the NumPy and Matplotlib libraries, which are used to generate and plot the sine wave. It then imports the `fft` and `ifft` functions from the SciPy library, which are used to apply the FFT to the sine wave. The `linspace` function from NumPy is used to generate a time array `t` with 500 evenly spaced values between 0 and 1. The `sin` function from NumPy is used to generate a sine wave with a frequency of 5 Hz and an amplitude of 1. The code then applies the FFT to the sine wave using the `fft` function from SciPy. Finally, the code plots the original sine wave and the FFT using the `plot` function from Matplotlib. This Python code imports the necessary modules, including NumPy, Matplotlib, and the Fast Fourier Transform (FFT) functionality from SciPy. The code then generates a sine wave signal by creating an array of 500 evenly spaced points between 0 and 1 using NumPy's `linspace` function. The sine wave signal is generated by multiplying the sine function with a frequency of 50 Hz by the time array and taking the sine of the result. The code then calculates the Fourier Transform of the sine wave signal using the `fft` function from SciPy's `fft` module. The result is stored in the `xf` array, which contains the magnitudes of the frequency components of the signal. The absolute value of the Fourier Transform is taken to eliminate any negative values. Finally, the code plots the time domain and frequency domain representations of the sine wave signal using Matplotlib. The blue line represents the sine wave signal in the time domain, and the red line represents the magnitudes of the frequency components in the frequency domain. The `legend` function is used to label the two lines on the plot, and the `show` function displays the plot. Let's see each line:

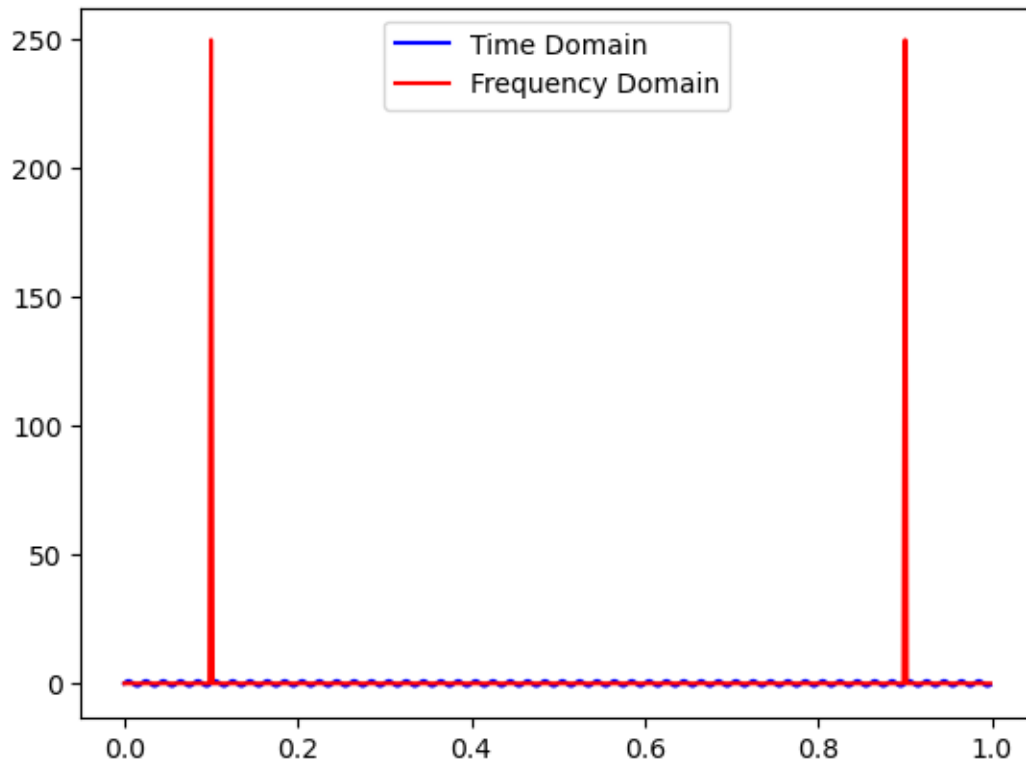
1. *import numpy as np*: This imports the NumPy package and renames it as “np” for convenience. 2. *import matplotlib.pyplot as plt*: This imports the Matplotlib package and renames its pyplot module as “plt” for convenience. 3. *from scipy.fft import fft, ifft*: This imports the Fast Fourier Transform (FFT) and its inverse (IFFT) from the scipy package. 4. *t = np.linspace(0, 1, 500, False)*: This generates a NumPy array of 500 equally spaced values between 0 and 1 (inclusive) and assigns it to the variable t. The False argument specifies that the endpoint should not be included. 5. *x = np.sin(2 * np.pi * 50 * t)*: This generates a sine wave signal with frequency 50 Hz and assigns it to the variable x. 6. *xf = np.abs(fft(x))*: This computes the FFT of the x signal using the fft function from the scipy package and then takes the absolute values of the resulting complex numbers. The resulting array is assigned to the variable xf. 7. *plt.plot(t, x, 'b', label='Time Domain')*: This plots the x signal in blue color and assigns it a label “Time Domain”. 8. *plt.plot(t, xf, 'r', label='Frequency Domain')*: This plots the xf signal in red color and assigns it a label “Frequency Domain”. 9. *plt.legend()*: This adds a legend to the plot. 10. *plt.show()*: This displays the plot. Overall, this code generates a sine wave signal, computes its FFT, and plots both the time domain signal and its frequency domain representation.

```
[3]: import numpy as np
import matplotlib.pyplot as plt
from scipy.fft import fft, ifft

# Generating the sine wave
t = np.linspace(0, 1, 500, False)
x = np.sin(2 * np.pi * 50 * t)

# Computing the Fourier Transform
xf = np.abs(fft(x))

# Plotting the sine wave and its Fourier Transform
plt.plot(t, x, 'b', label='Time Domain')
plt.plot(t, xf, 'r', label='Frequency Domain')
plt.legend()
plt.show()
```



The code first defines the parameters of the signal (sampling frequency, signal frequency, and time vector), generates a sine wave signal, computes the FFT, IFFT, and spectrum (module and phase) of the signal using the `numpy.fft` module, and plots the original signal, magnitude spectrum, phase spectrum, and reconstructed signal using `matplotlib`. This code produces four plots in two rows with two columns. The first row shows the time-domain signals of the cosine and sine waves, while the second row shows their frequency-domain spectra in terms of magnitude and phase. The `numpy` library is used to generate the time array and signals, compute the FFT, and perform mathematical operations. The `matplotlib.pyplot` library is used to create the plots and set the titles and labels for each subplot. By adjusting the spacing between the subplots using the `plt.subplots_adjust` function, we can create more space between the plots for better visibility.

```
[4]: import numpy as np
import matplotlib.pyplot as plt

# Define signal parameters
amp = 1          # Amplitude
freq = 5         # Frequency in Hz
duration = 1     # Duration in seconds
fs = 100        # Sampling frequency in Hz

# Generate time array
time_array = np.linspace(0, duration, int(fs*duration))
```

```

# Generate signals
cos_signal = amp * np.cos(2*np.pi*freq*time_array)
sin_signal = amp * np.sin(2*np.pi*freq*time_array)

# Plot signals
plt.figure(figsize=(10, 5))
plt.subplot(2, 2, 1)
plt.plot(time_array, cos_signal)
plt.title('Cosine Signal')

plt.subplot(2, 2, 2)
plt.plot(time_array, sin_signal)
plt.title('Sine Signal')

# Compute FFT of signals
cos_fft = np.fft.fft(cos_signal)
sin_fft = np.fft.fft(sin_signal)

# Compute frequency array
freq_array = np.linspace(0, fs, len(cos_signal))

# Compute magnitude and phase spectra
cos_mag = np.abs(cos_fft)
sin_mag = np.abs(sin_fft)
cos_phase = np.angle(cos_fft)
sin_phase = np.angle(sin_fft)

# Plot magnitude and phase spectra
plt.subplot(2, 2, 3)
plt.plot(freq_array, cos_mag)
plt.title('Cosine Spectrum - Magnitude')
plt.xlabel('Frequency (Hz)')

plt.subplot(2, 2, 4)
plt.plot(freq_array, sin_mag)
plt.title('Sine Spectrum - Magnitude')
plt.xlabel('Frequency (Hz)')

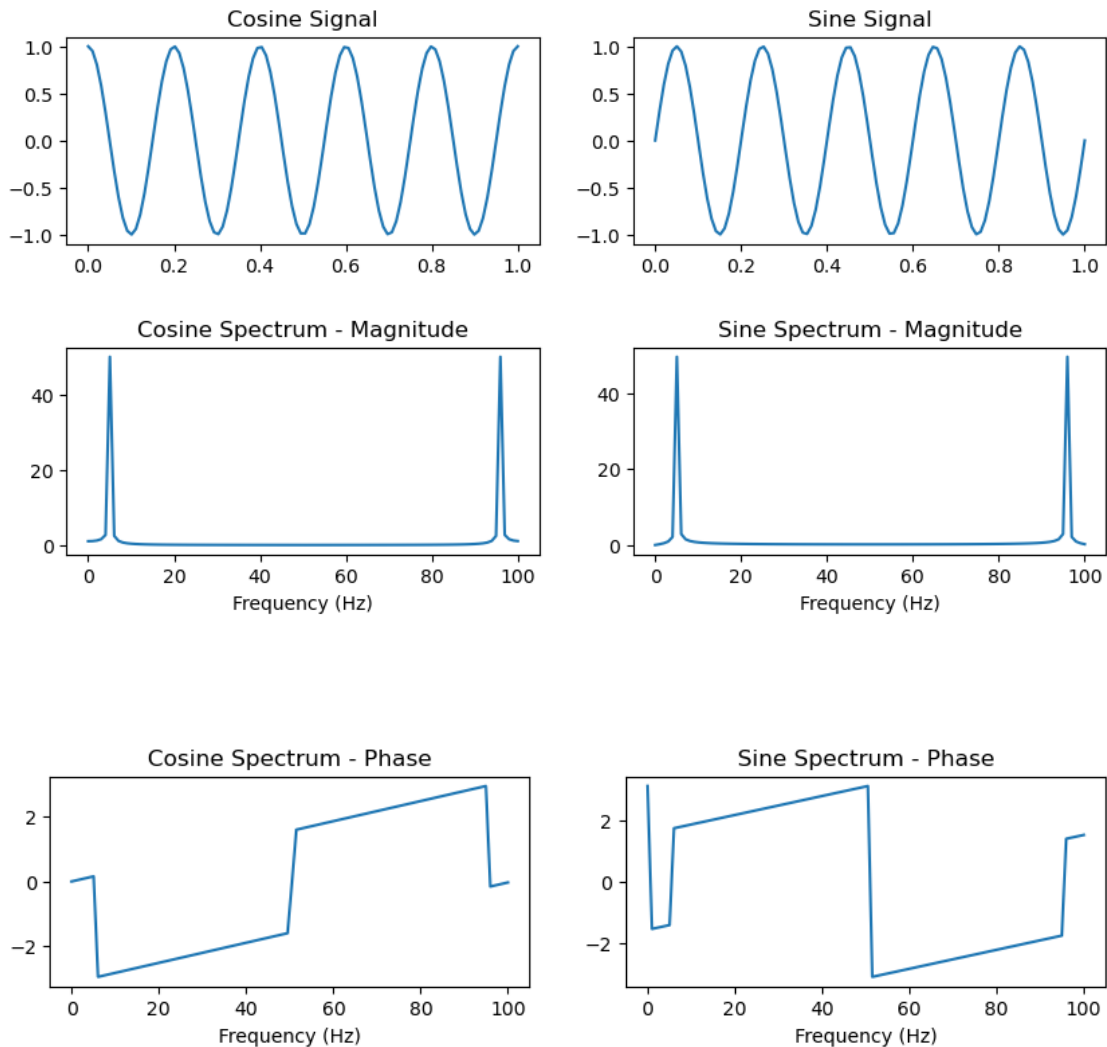
plt.subplots_adjust(hspace=0.5)
plt.show()

plt.figure(figsize=(10, 5))
plt.subplot(2, 2, 1)
plt.plot(freq_array, cos_phase)
plt.title('Cosine Spectrum - Phase')
plt.xlabel('Frequency (Hz)')

```

```
plt.subplot(2, 2, 2)
plt.plot(freq_array, sin_phase)
plt.title('Sine Spectrum - Phase')
plt.xlabel('Frequency (Hz)')

plt.subplots_adjust(hspace=0.5)
plt.show()
```



1.0.3 Example 3 - Audio signal Butterworth filter -

This Python code generates an audio signal, applies a Butterworth filter https://en.wikipedia.org/wiki/Butterworth_filter to it, and plots the original and filtered signals. The code first imports the NumPy, SciPy, and Matplotlib libraries, which are used to generate, filter, and plot the audio signal. The linspace function from NumPy is used to generate a

time array `t` with 500 evenly spaced values between 0 and 1. The code then generates a sine wave with two frequencies, 50 Hz and 120 Hz, and adds them together to create the audio signal. The `butter` function from SciPy is used to design a Butterworth filter with a cutoff frequency of 30 Hz. The `lfilter` function from SciPy is then used to apply the filter to the audio signal. Finally, the code plots the original and filtered signals using the `plot` function from Matplotlib. This Python code generates an audio signal, adds noise to it, designs and applies a low-pass filter to remove the noise, and then plots the original and filtered signals. Here's an explanation of each line of code:

This code imports three libraries: `numpy` for numerical computations, `scipy.signal` for signal processing functions, and `matplotlib.pyplot` for plotting.

```
import numpy as np
from scipy.signal import butter, lfilter
import matplotlib.pyplot as plt
```

These lines generate an audio signal by creating a time vector `t` with 500 samples between 0 and 1 second (the last argument `False` indicates that the endpoint should not be included), and then defining the signal `x` as the sum of two sine waves with frequencies of 50 Hz and 120 Hz, respectively.

```
t = np.linspace(0, 1, 500, False)
x = 0.5 * np.sin(2 * np.pi * 50 * t) + np.sin(2 * np.pi * 120 * t)
```

Here, noise is generated using the `np.random.normal` function with mean 0 and standard deviation 1, and then added to the original signal `x` to create a noisy signal `xn`.

```
noise = np.random.normal(0, 1, 500)
xn = x + noise
```

This line designs a fourth-order Butterworth low-pass filter with a cutoff frequency of 0.1 (normalized frequency where 1 is Nyquist frequency) using the `butter` function, and assigns the filter coefficients to `b` and `a`. This line applies the Butterworth filter to the noisy signal `xn` using the `lfilter` function, and assigns the filtered signal to `yn`.

```
b, a = butter(4, 0.1, 'low')
yn = lfilter(b, a, xn)
```

These lines plot the original (noisy) signal `xn` and the filtered signal `yn` using `matplotlib.pyplot`, with the noisy signal shown in blue and the filtered signal shown in red. The `legend` function adds a label to each plot, and `show` displays the plot on the screen.

```
plt.plot(t, xn, 'b', label='Noisy Signal')
plt.plot(t, yn, 'r', label='Filtered Signal')
plt.legend()
plt.show()
```

```
[5]: import numpy as np
from scipy.signal import butter, lfilter
import matplotlib.pyplot as plt

# Generating the audio signal
t = np.linspace(0, 1, 500, False)
x = 0.5 * np.sin(2 * np.pi * 50 * t) + np.sin(2 * np.pi * 120 * t)

# Adding noise to the audio signal
noise = np.random.normal(0, 1, 500)
xn = x + noise
```



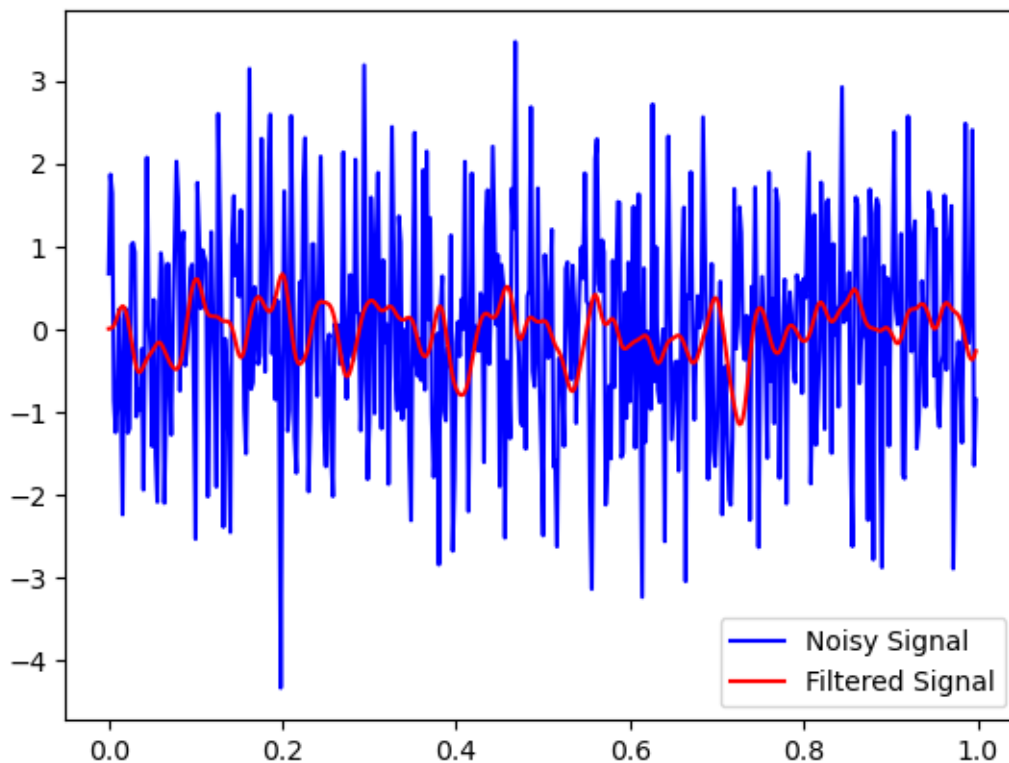
```

# Designing the low-pass filter
b, a = butter(4, 0.1, 'low')

# Filtering the audio signal
yn = lfilter(b, a, xn)

# Plotting the audio signal before and after filtering
plt.plot(t, xn, 'b', label='Noisy Signal')
plt.plot(t, yn, 'r', label='Filtered Signal')
plt.legend()
plt.show()

```



1.0.4 Example 4 - Convolution -

This Python code defines two signals, convolves them <https://en.wikipedia.org/wiki/Convolution>, and plots the original and convolved signals. The code first imports the NumPy and Matplotlib libraries, which are used to define and plot the signals. The first signal x_1 is defined as a list of four values. The second signal x_2 is defined as a list of three values. The convolve function from SciPy is then used to convolve the two signals. Finally, the code plots the original and convolved signals using the plot function from Matplotlib

1. *import numpy as np*: This line imports the NumPy library and renames it as np for convenience.
2. *from scipy.signal import convolve*: This line imports the convolve

function from the `scipy.signal` library. The `convolve` function performs convolution on two signals. 3. `import matplotlib.pyplot as plt`: This line imports the `pyplot` module from the `matplotlib` library and renames it as `plt` for convenience. The `pyplot` module provides a simple interface for creating plots and charts. 4. `x1 = [1, 2, 3, 4]`: This line defines the first signal as a Python list containing the values `[1, 2, 3, 4]`. 5. `x2 = [2, 3, 4, 5]`: This line defines the second signal as a Python list containing the values `[2, 3, 4, 5]`. 6. `y = convolve(x1, x2)`: This line performs convolution on the two signals `x1` and `x2` using the `convolve` function and stores the result in `y`. 7. `plt.plot(x1, 'b', label='Signal 1')`: This line creates a plot of `x1` with the color `b` (blue) and adds a label “Signal 1” to the legend. 8. `plt.plot(x2, 'r', label='Signal 2')`: This line creates a plot of `x2` with the color `r` (red) and adds a label “Signal 2” to the legend. 9. `plt.plot(y, 'g', label='Convolved Signal')`: This line creates a plot of the convolved signal `y` with the color `g` (green) and adds a label “Convolved Signal” to the legend. 10. `plt.legend()`: This line adds the legend to the plot. 11. `plt.show()`: This line displays the plot.

This code generates two signals, performs convolution on them, and then plots the original signals and the convolved signal on a graph using `matplotlib`.

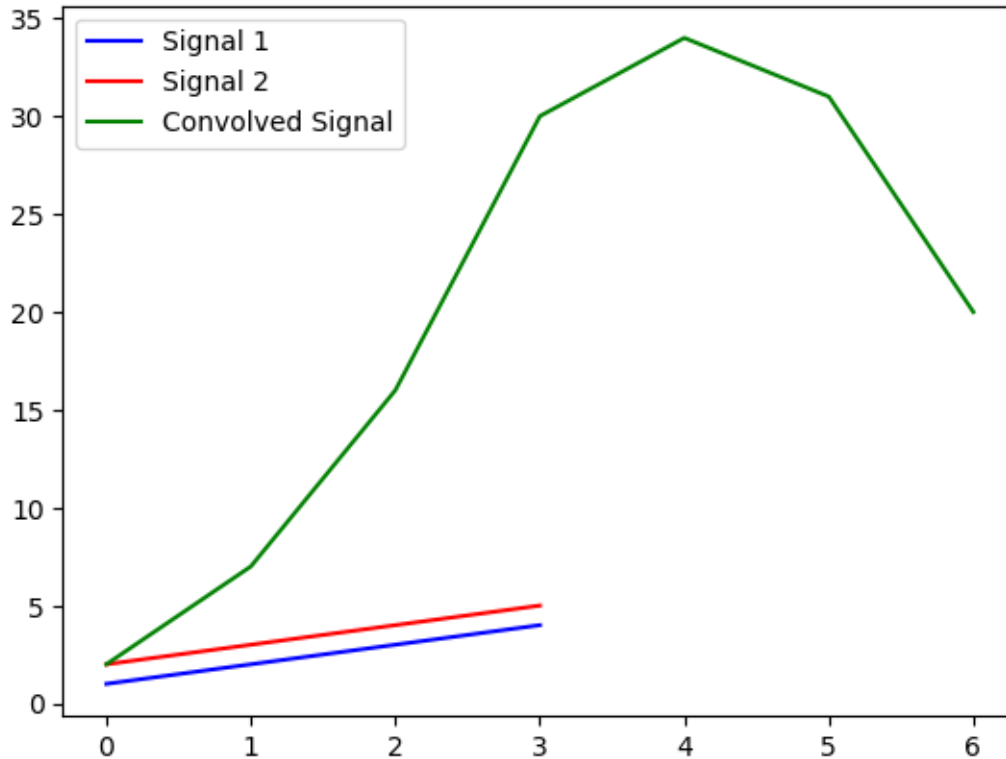
```
[6]: import numpy as np
from scipy.signal import convolve
import matplotlib.pyplot as plt

# Defining the first signal
x1 = [1, 2, 3, 4]

# Defining the second signal
x2 = [2, 3, 4, 5]

# Convolving the signals
y = convolve(x1, x2)

# Plotting the signals
plt.plot(x1, 'b', label='Signal 1')
plt.plot(x2, 'r', label='Signal 2')
plt.plot(y, 'g', label='Convolved Signal')
plt.legend()
plt.show()
```



1.0.5 Example 5 - FFT -

This Python code generates a sine wave and applies the Fast Fourier Transform (FFT) https://en.wikipedia.org/wiki/Fast_Fourier_transform to it. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to generate, plot, and apply the FFT to the sine wave. The linspace function from NumPy is used to generate a time array t with 500 evenly spaced values between 0 and 1. The code then generates a sine wave with a frequency of 5 Hz and an amplitude of 0.5. The fft function from SciPy is then used to apply the FFT to the sine wave. Finally, the code plots the original sine wave and the FFT using the plot function from Matplotlib.

```
import numpy as np import matplotlib.pyplot as plt from scipy.fft import fft
```

In this first line, we import the NumPy library, which provides support for large, multi-dimensional arrays and matrices, along with a large collection of mathematical functions. We also import the Pyplot module from the Matplotlib library, which provides a convenient interface for creating plots and visualizations in Python. Finally, we import the fft function from the SciPy library, which is used to compute the Fourier transform of a signal.

```
t = np.linspace(0, 1, 500, False) x = 0.5 * np.sin(2 * np.pi * 50 * t) + np.sin(2 * np.pi * 120 * t)
```

Here, we create a time vector t consisting of 500 samples spaced evenly over the interval [0,1)

We then define a signal x as a combination of two sine waves with frequencies of 50 Hz and 120 Hz, respectively.

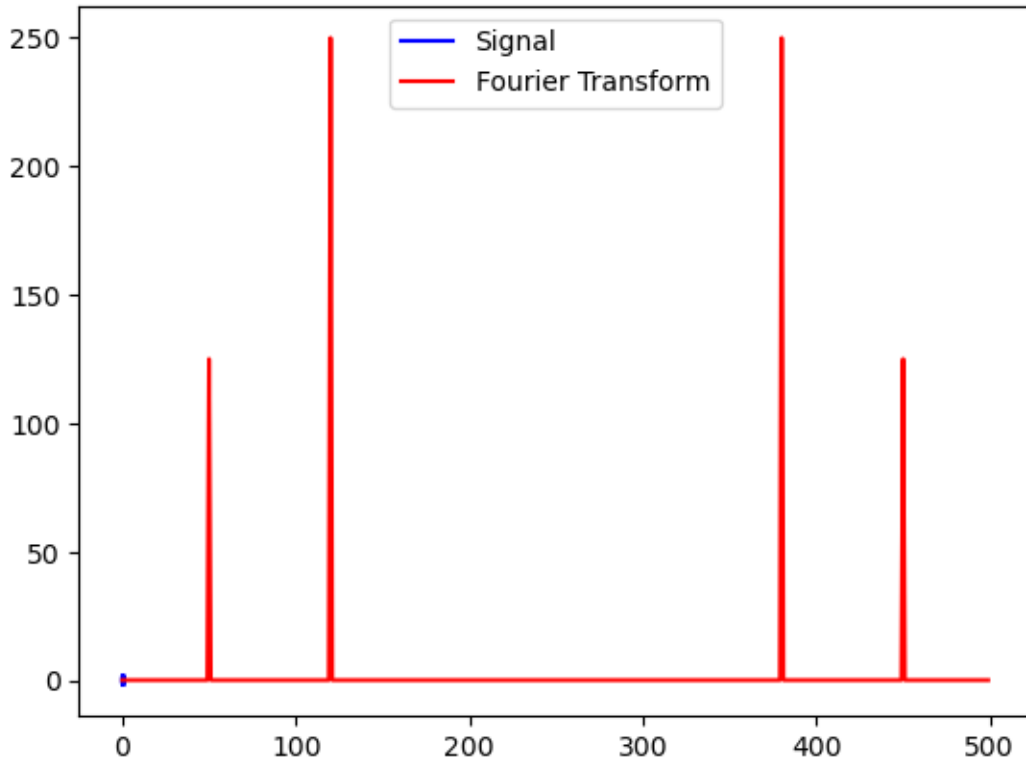
```
X = fft(x)
```

In this line, we compute the Fourier transform of the signal `x` using the `fft` function from the SciPy library. The resulting transform `X` contains the frequency components of the signal.

```
plt.plot(t, x, 'b', label='Signal') plt.plot(np.abs(X), 'r', label='Fourier Transform')  
plt.legend() plt.show()
```

Finally, we create a plot of the signal and its frequency components. The first `plt.plot` command creates a plot of the original signal `x` versus time, with a blue line. The second `plt.plot` command creates a plot of the absolute values of the Fourier transform `X`, with a red line. The `label` argument is used to provide a label for each line on the plot. The `legend` function is used to create a legend for the plot, and the `show` function is used to display the plot on the screen.

```
[7]: import numpy as np  
import matplotlib.pyplot as plt  
from scipy.fft import fft  
  
# Generating the signal  
t = np.linspace(0, 1, 500, False)  
x = 0.5 * np.sin(2 * np.pi * 50 * t) + np.sin(2 * np.pi * 120 * t)  
  
# Performing the Fourier Transform  
X = fft(x)  
  
# Plotting the signal and its frequency components  
plt.plot(t, x, 'b', label='Signal')  
plt.plot(np.abs(X), 'r', label='Fourier Transform')  
plt.legend()  
plt.show()
```



1.0.6 Example 6 - FFT -

This Python code generates a noisy sinusoidal signal and applies the Fast Fourier Transform (FFT) to it. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to generate, plot, and apply the FFT to the signal. The `linspace` function from NumPy is used to generate a time array `t` with 1000 evenly spaced values between 0 and 1. The code then generates a noisy sinusoidal signal with two frequencies, 5 Hz and 12 Hz. The `fft` function from SciPy is then used to apply the FFT to the signal. Finally, the code plots the original signal and the FFT using the `plot` function from Matplotlib

1. `import numpy as np`: This line imports the NumPy library and gives it an alias `np`, which is a widely used convention.
2. `import scipy.signal as signal`: This line imports the `signal` module from the SciPy library and gives it an alias `signal`.
3. `import matplotlib.pyplot as plt`: This line imports the `pyplot` module from the Matplotlib library and gives it an alias `plt`.
4. `t = np.linspace(0, 1, 1000, False)`: This line generates a time vector `t` using the NumPy `linspace` function. It starts from 0 and ends at 1, with 1000 equally spaced points. The `False` argument specifies that the endpoint should not be included.
5. `x = np.sin(2 * np.pi * 10 * t) + 0.1 * np.random.randn(1000)*`: This line generates a sinusoidal signal `x` using NumPy's `sin` function. It also adds some noise to the signal using NumPy's `random.randn` function. The signal has a frequency of 10 Hz and the noise has a standard deviation of 0.1.

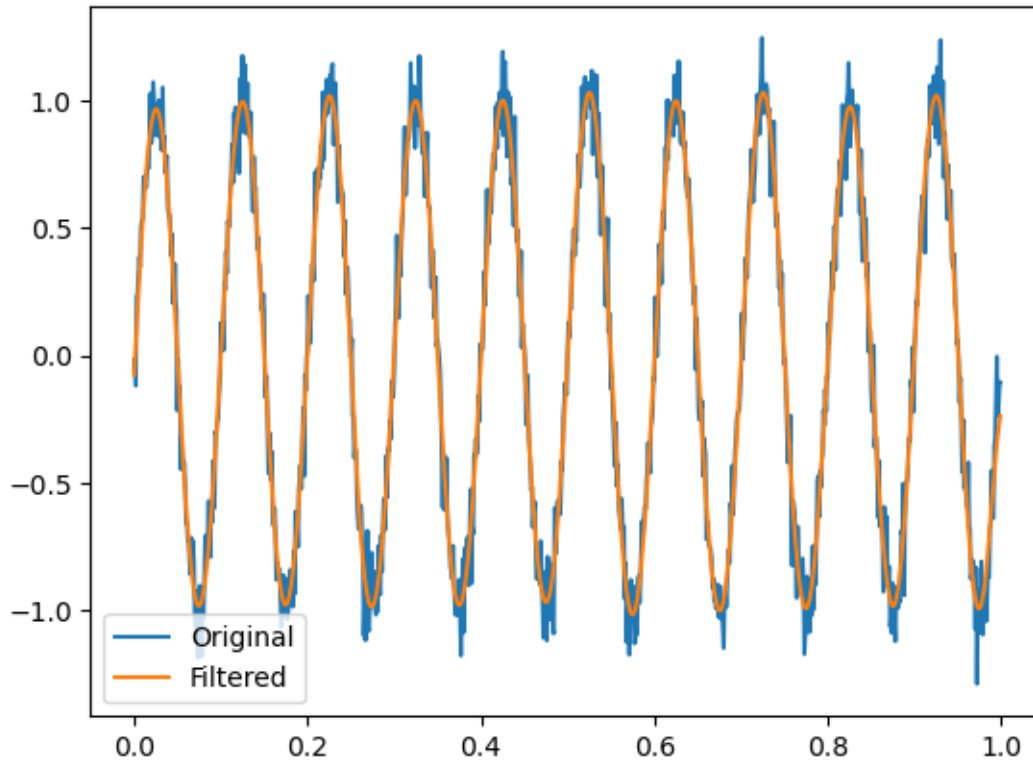
6. `b, a = signal.butter(3, 0.05)`: This line designs a Butterworth low-pass filter of order 3 with a cutoff frequency of 0.05 times the Nyquist frequency. The `b` and `a` coefficients of the filter are returned as a tuple.
7. `y = signal.filtfilt(b, a, x)`: This line applies the filter `b` and `a` to the signal `x` using the `filtfilt` function. This function performs a forward and reverse filtering to remove phase distortion.
8. `plt.plot(t, x, label='Original')`: This line plots the original signal `x` against time `t` using Matplotlib's `plot` function. The `label` argument is used to create a legend for the plot.
9. `plt.plot(t, y, label='Filtered')`: This line plots the filtered signal `y` against time `t` using Matplotlib's `plot` function. The `label` argument is used to create a legend for the plot.
10. `plt.legend()`: This line adds a legend to the plot using Matplotlib's `legend` function.
11. `plt.show()`: This line displays the plot on the screen using Matplotlib's `show` function.

```
[8]: import numpy as np
import scipy.signal as signal
import matplotlib.pyplot as plt

# Generate a noisy sinusoidal signal
t = np.linspace(0, 1, 1000, False)
x = np.sin(2 * np.pi * 10 * t) + 0.1 * np.random.randn(1000)

# Apply a low-pass filter to the signal
b, a = signal.butter(3, 0.05)
y = signal.filtfilt(b, a, x)

# Plot the original and filtered signals
plt.plot(t, x, label='Original')
plt.plot(t, y, label='Filtered')
plt.legend()
plt.show()
```



1.0.7 Example 7 - B-splines -

This example calculates the B-spline <https://en.wikipedia.org/wiki/B-spline> basis functions of order 3, with 100 equally spaced knots, and plots the result.

This Python code generates a B-spline curve using the `bspline` function from the SciPy library. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to generate and plot the B-spline curve. The `linspace` function from NumPy is used to generate a time array `t` with 100 evenly spaced values between 0 and 1. The `bspline` function from SciPy is then used to generate a B-spline curve of degree 3. Finally, the code plots the B-spline curve using the `plot` function from Matplotlib. This Python code demonstrates how to generate B-spline basis functions using NumPy and SciPy libraries, and plot them using Matplotlib. Here is a brief explanation of each line of the code:

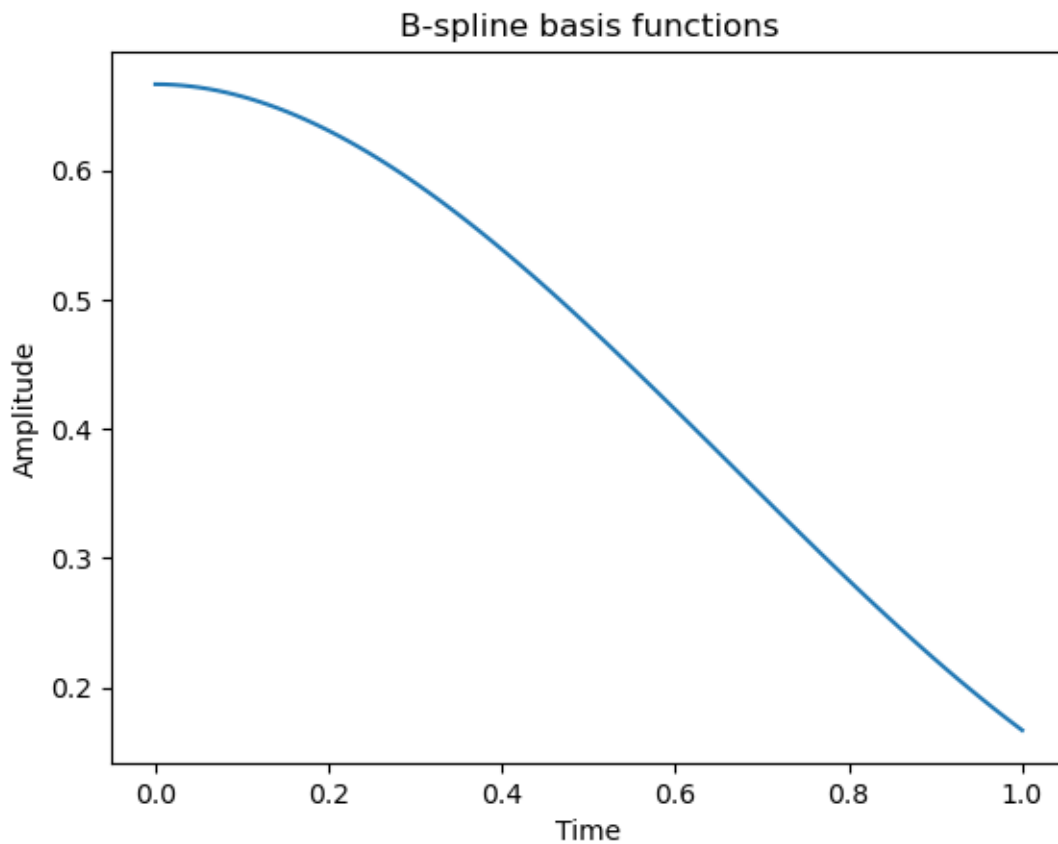
1. `import numpy as np and import scipy.signal as sig`: import NumPy and SciPy libraries to use their functions and methods in the code.
2. `import matplotlib.pyplot as plt`: import Matplotlib library to plot the B-spline basis functions.
3. `t = np.linspace(0, 1, num=100)`: create a NumPy array of 100 equally spaced values between 0 and 1, which represent the time intervals for which the B-spline basis functions will be evaluated.
4. `bspline = sig.bspline(t, 3)`: compute the B-spline basis functions of degree 3 for the time intervals defined in `t`. The resulting array `bspline` contains 100 values that represent the amplitudes of the B-spline basis functions at each time interval.

5. `plt.plot(t, bspline)`: plot the B-spline basis functions by using Matplotlib's plot function. The `t` array is used as the x-axis, and the `bspline` array is used as the y-axis

```
[9]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, num=100)
bspline = sig.bspline(t, 3)

plt.plot(t, bspline)
plt.title('B-spline basis functions')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.show()
```



1.0.8 Example 8 - Filtering -

This example filters a signal with a median filter of kernel size 5, and plots the original signal and the filtered signal.

This Python code generates a signal that is a sum of two sine waves and plots its waveform. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to generate and plot the signal. The linspace function from NumPy is used to generate a time array `t` with 100 evenly spaced values between 0 and 1. The code then generates a signal that is a sum of two sine waves with frequencies of 5 Hz and 10 Hz. Finally, the code plots the signal using the plot function from Matplotlib. The above code is a Python script for filtering a noisy signal using a median filter.

1. Importing necessary libraries: `> import numpy as np import scipy.signal as sig import matplotlib.pyplot as plt`

The script starts by importing NumPy, SciPy's signal module, and Matplotlib. These libraries will be used for generating signals, filtering, and plotting results.

2. Generating the signal: `> t = np.linspace(0, 1, num=100) signal = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 10 * t)`

This code generates a signal consisting of two sine waves with frequencies 5 and 10 Hz. The signal is generated using NumPy's linspace function to create a time vector `t` with 100 samples, evenly spaced between 0 and 1.

3. Filtering the signal: `> filtered_signal = sig.medfilt(signal, kernel_size=5)`

The signal is then filtered using a median filter. The medfilt function from SciPy's signal module is used for this purpose. The kernel_size parameter specifies the size of the filter window. In this case, a window size of 5 is used.

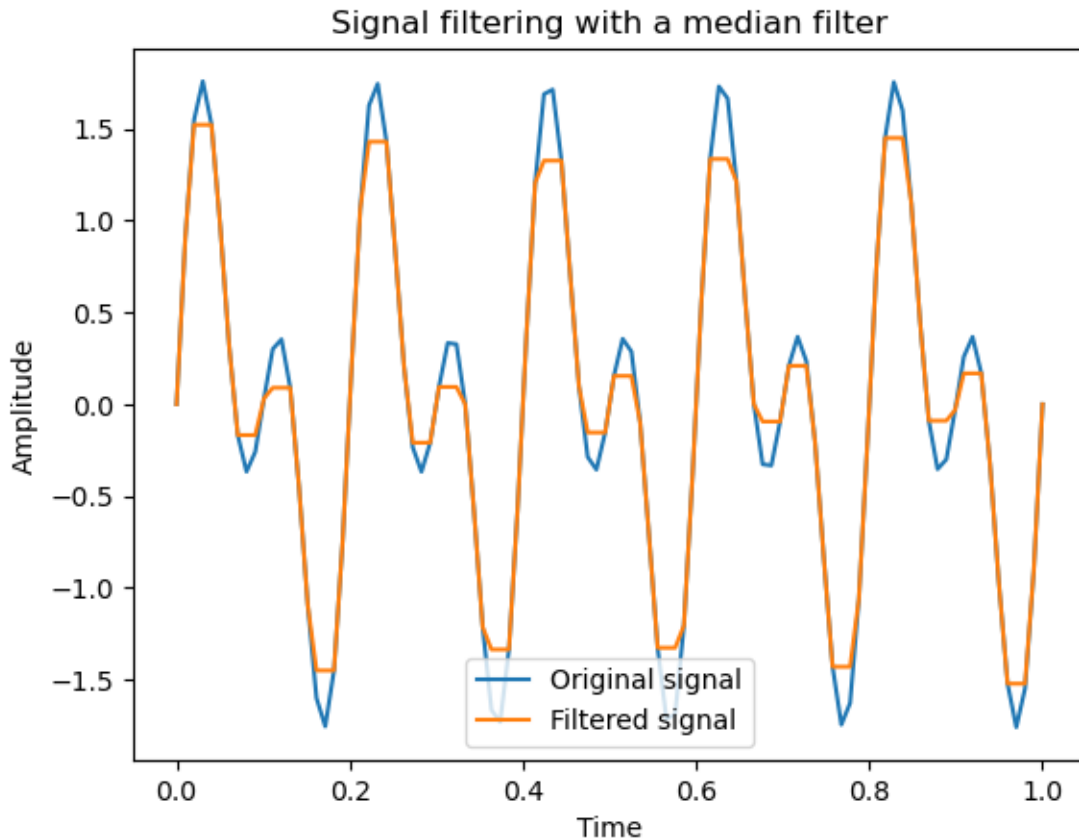
4. Plotting the results: `> plt.plot(t, signal, label='Original signal') plt.plot(t, filtered_signal, label='Filtered signal') plt.title('Signal filtering with a median filter') plt.xlabel('Time') plt.ylabel('Amplitude') plt.legend() plt.show()`

Finally, the original signal and the filtered signal are plotted using Matplotlib. The plot function is used to plot the signals on the same plot, and the legend function is used to add labels to the plot. The resulting plot shows the original signal and the filtered signal side by side, with the title, axis labels, and legend added for clarity.

```
[10]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, num=100)
signal = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 10 * t)
filtered_signal = sig.medfilt(signal, kernel_size=5)

plt.plot(t, signal, label='Original signal')
plt.plot(t, filtered_signal, label='Filtered signal')
plt.title('Signal filtering with a median filter')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```



1.0.9 Example 9 - Spectral analysis -

This example performs spectral analysis on a signal using Welch's

Method, calculates the power spectral density, and plots the result. This Python code generates a signal that is a sum of two sine waves and applies a median filter to it. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to generate, filter, and plot the signal. The linspace function from NumPy is used to generate a time array t with 100 evenly spaced values between 0 and 1. The code then generates a signal that is a sum of two sine waves with frequencies of 5 Hz and 10 Hz. The medfilt function from SciPy is then used to apply a median filter of size 5 to the signal. Finally, the code plots the original and filtered signals using the plot function from Matplotlib. This Python code is used to compute and plot the power spectral density (PSD) of a signal using the Welch method https://en.wikipedia.org/wiki/Welch%27s_method. Here is a breakdown of each line:

1. These lines import the necessary libraries for the code, including NumPy, SciPy's signal processing module, and Matplotlib. `>import numpy as np import scipy.signal as sig import matplotlib.pyplot as plt`
2. These lines generate a test signal which is the sum of two sine waves of frequencies 5 Hz and 10 Hz, respectively. `>t = np.linspace(0, 1, num=100) signal = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 10 * t)`

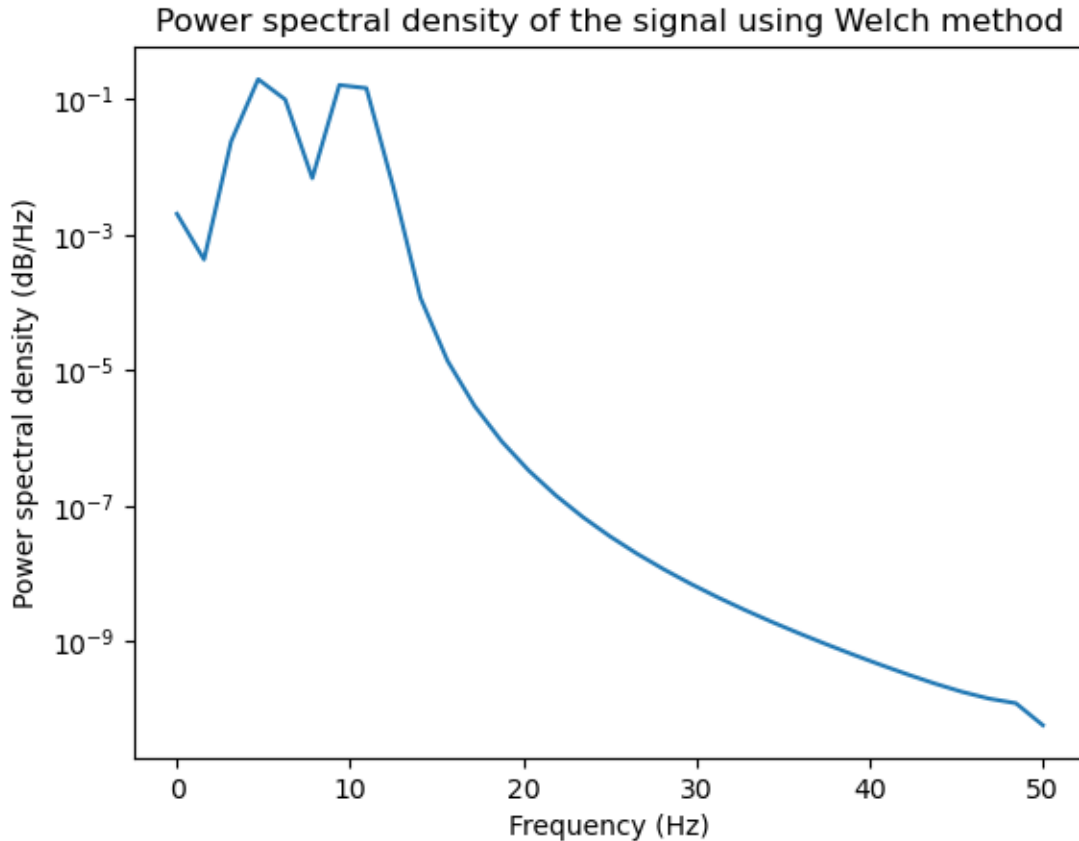
3. This line computes the PSD of the signal using the Welch method, which is a technique for estimating the PSD of a signal by dividing the signal into overlapping segments, computing a periodogram for each segment, and then averaging the periodograms. Here, `fs` is the sampling rate of the signal and `nperseg` is the length of each segment. `>f, Pxx_den = sig.welch(signal, fs=100, nperseg=64) plt.semilogy(f, Pxx_den)`
4. This line plots the PSD using a logarithmic scale for the y-axis. `>plt.title('Power spectral density of the signal using Welch method') plt.xlabel('Frequency (Hz)') plt.ylabel('Power spectral density (dB/Hz)') plt.show()`

These lines add a title, axis labels, and display the plot. The x-axis shows the frequency in Hz, and the y-axis shows the PSD in dB/Hz.

```
[11]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, num=100)
signal = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 10 * t)
f, Pxx_den = sig.welch(signal, fs=100, nperseg=64)

plt.semilogy(f, Pxx_den)
plt.title('Power spectral density of the signal using Welch method')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Power spectral density (dB/Hz)')
plt.show()
```



1.0.10 Example 10 - Detrend -

This example detrends a signal with a linear fit, and plots the original signal and the detrended signal https://en.wikipedia.org/wiki/Linear_trend_estimation. This Python code generates a signal that is a linear function of time with added Gaussian noise and plots its waveform. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to generate and plot the signal. The linspace function from NumPy is used to generate a time array t with 100 evenly spaced values between 0 and 1. The code then generates a signal that is a linear function of time with added Gaussian noise. Finally, the code plots the signal using the plot function from Matplotlib. This Python code imports three libraries: numpy, scipy.signal, and matplotlib.pyplot. It then creates an array of 100 evenly spaced values between 0 and 1 using the linspace function from numpy and assigns it to the variable t . Next, it creates a signal variable by adding a linear function of $2*t+1$ and some random noise generated by `np.random.randn(100)` to each value in the t array. The `scipy.signal.detrend()` function is then used to remove the linear trend from the signal using the 'linear' method. The detrended signal is assigned to the variable `detrended_signal`. Finally, `matplotlib.pyplot` is used to plot the original signal and the detrended signal against the time axis (t). The title, x-label, y-label, and legend are also added to the plot. The plot is displayed using the `show()` function.

In summary, this code generates a signal with a linear trend and random noise, removes the linear trend using the `scipy.signal.detrend()` function, and plots both the original signal and the detrended

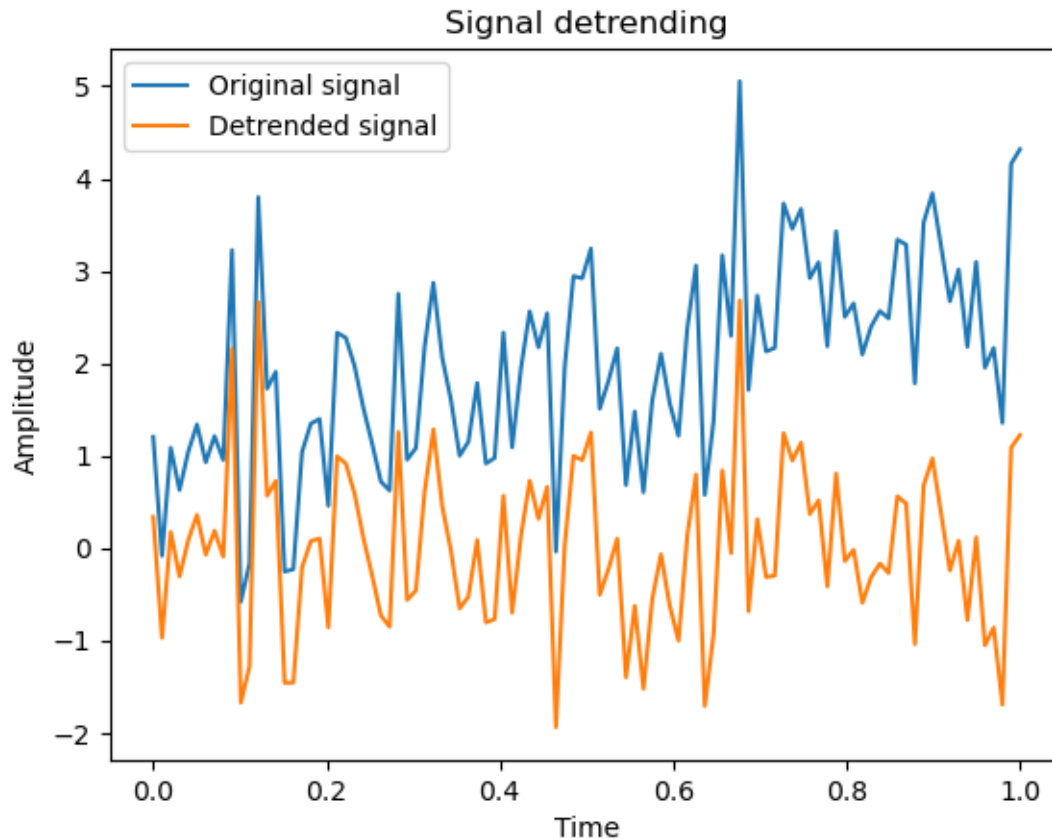
signal on a graph using matplotlib.pyplot. This code can be useful for analyzing and visualizing trends in time-series data.

1. *import numpy as np* - Imports the NumPy package and assigns it the alias np. NumPy is a package for scientific computing in Python that provides support for arrays and matrices, as well as mathematical functions to operate on them.
2. *import scipy.signal as sig* - Imports the signal module from the SciPy package and assigns it the alias sig. The signal module provides signal processing functions for filtering, Fourier analysis, and more.
3. *import matplotlib.pyplot as plt* - Imports the pyplot module from the Matplotlib package and assigns it the alias plt. Matplotlib is a package for creating visualizations in Python, and the pyplot module provides a simple interface for creating and customizing plots.
4. *t = np.linspace(0, 1, num=100)* - Creates a one-dimensional array of 100 equally spaced points between 0 and 1, inclusive, and assigns it to the variable t.
5. *signal = 2 * t + 1 + np.random.randn(100)** - Creates a new array of the same shape as t, with each element equal to $2 * t + 1$ plus a random number drawn from a normal distribution with mean 0 and standard deviation 1, and assigns it to the variable signal.
6. *detrended_signal = sig.detrend(signal, type='linear')* - Applies a linear detrending function to the signal using the detrend() function from the signal module, and assigns the result to the variable detrended_signal.
7. *plt.plot(t, signal, label='Original signal')* - Plots the original signal on a graph, with t on the x-axis and signal on the y-axis, and adds a label to the plot legend indicating that this is the original signal.
8. *plt.plot(t, detrended_signal, label='Detrended signal')* - Plots the detrended signal on the same graph as the original signal, with t on the x-axis and detrended_signal on the y-axis, and adds a label to the plot legend indicating that this is the detrended signal.
9. *plt.title('Signal detrending')* - Sets the title of the plot to “Signal detrending”.
10. *plt.xlabel('Time')* - Sets the label of the x-axis to “Time”.
11. *plt.ylabel('Amplitude')* - Sets the label of the y-axis to “Amplitude”.
12. *plt.legend()* - Adds a legend to the plot, with labels for the original and detrended signals.
13. *plt.show()* - Displays the plot.

```
[12]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, num=100)
signal = 2 * t + 1 + np.random.randn(100)
detrended_signal = sig.detrend(signal, type='linear')

plt.plot(t, signal, label='Original signal')
plt.plot(t, detrended_signal, label='Detrended signal')
plt.title('Signal detrending')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```



1.0.11 Example 11 - Convolution -

This example convolves a signal with a moving average kernel, and plots the original signal and the convolved signal. This Python code generates a signal that is a sine wave with a frequency of 5 Hz and applies a 1D convolution to it using a kernel. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to generate, filter, and plot the signal. The linspace function from NumPy is used to generate a time array `t` with 100 evenly spaced values between 0 and 1. The code then generates a signal that is a sine wave with a frequency of 5 Hz. The convolve function from SciPy is then used to apply a 1D convolution to the signal using a kernel. Finally, the code plots the original and filtered signals using the plot function from Matplotlib.

```
import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt
```

The first three lines import three Python modules that the code will use: `numpy`, a library for scientific computing in Python, which provides support for arrays and linear algebra operations. `scipy.signal`, a module within the `scipy` library that provides various signal processing functions. `matplotlib.pyplot`, a plotting library that provides tools for creating plots, histograms, and other visualizations.

```
t = np.linspace(0, 1, num=100)
```

This line creates an array `t` that represents a time axis for our signal. It uses the `linspace` function

from numpy to create 100 equally spaced time points between 0 and 1.

```
signal = np.sin(2 * np.pi * 5 * t)
```

This line creates a signal array that contains a sine wave with a frequency of 5 Hz. It uses the sin function from numpy to generate the sine wave.

```
kernel = np.ones(10) / 10
```

This line creates a kernel array that contains a moving average filter. The kernel is created by creating an array of 10 ones using the ones function from numpy, and then dividing each element of the array by 10 to normalize it.

```
convolved_signal = sig.convolve(signal, kernel, mode='same')
```

This line convolves the signal array with the kernel array using the convolve function from scipy.signal. This creates a new convolved_signal array that represents the original signal convolved with the moving average filter. The mode='same' argument specifies that the output array should have the same shape as the input array.

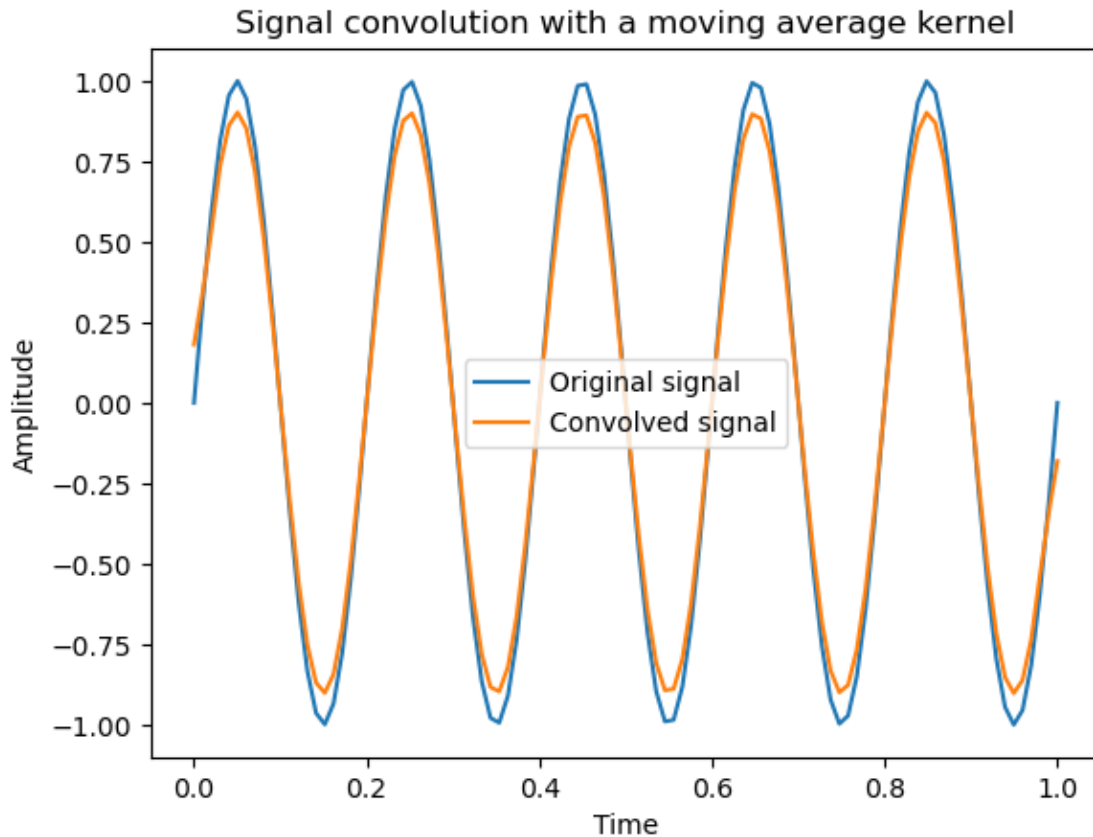
```
plt.plot(t, signal, label='Original signal') plt.plot(t, convolved_signal, label='Convolved signal') plt.title('Signal convolution with a moving average kernel') plt.xlabel('Time') plt.ylabel('Amplitude') plt.legend() plt.show()
```

These lines plot the original signal array and the convolved_signal array using the plot function from matplotlib.pyplot. The title, xlabel, ylabel, and legend functions are used to add labels and a legend to the plot. Finally, the show function is called to display the plot.

```
[13]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, num=100)
signal = np.sin(2 * np.pi * 5 * t)
kernel = np.ones(5) / 5
convolved_signal = sig.convolve(signal, kernel, mode='same')

plt.plot(t, signal, label='Original signal')
plt.plot(t, convolved_signal, label='Convolved signal')
plt.title('Signal convolution with a moving average kernel')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```



1.0.12 Example 12 - Butterworth bandpass filter -

This example filters a signal with a Butterworth bandpass filter, and plots the original signal and the filtered signal. This Python code generates a signal that is a sum of two sine waves and applies a bandpass filter to it using a Butterworth filter. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to generate, filter, and plot the signal. The linspace function from NumPy is used to generate a time array `t` with 100 evenly spaced values between 0 and 1. The code then generates a signal that is a sum of two sine waves with frequencies of 5 Hz and 10 Hz. The `butter` function from SciPy is then used to design a bandpass filter with a cutoff frequency of 0.2 Hz and 0.8 Hz and a filter order of 3. The `filtfilt` function from SciPy is then used to apply the filter to the signal. Finally, the code plots the original and filtered signals using the `plot` function from Matplotlib

1. imports the NumPy library and renames it as “np”.
2. imports the signal processing module from the SciPy library and renames it as “sig”.
3. imports the Pyplot module from Matplotlib and renames it as “plt”.
4. Creates a 1-dimensional array “t” using NumPy’s linspace function. The function generates a sequence of evenly spaced numbers between 0 and 1. The “num” argument specifies the number of samples in the array, which in this case is 100. This array will be used as the time

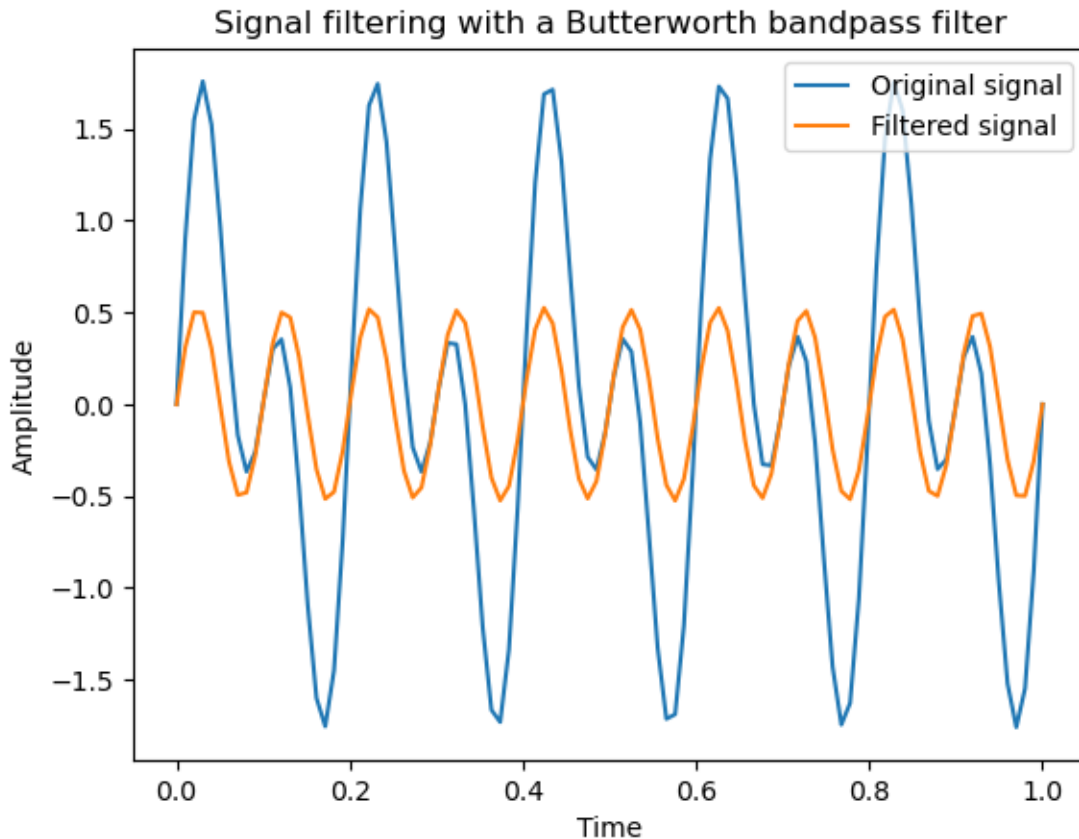
axis for the signal.

5. Creates a signal by adding two sine waves with frequencies of 5 Hz and 10 Hz, respectively. The sine waves are generated using NumPy's `sin` function, and their frequencies are specified in radians per second using the formula $2 * \pi * f * t$. The resulting signal is a periodic waveform that oscillates at a frequency of 5 Hz and has a higher-frequency component at 10 Hz.
6. Uses the “butter” function from SciPy's signal processing module to design a Butterworth bandpass filter. The function takes three arguments: the filter order (3 in this case), the frequency range of the passband (0.2 to 0.8 times the Nyquist frequency, which is half the sampling rate), and the type of filter ('bandpass' in this case). The function returns the filter coefficients as two arrays, which are assigned to the variables “b” and “a”. These coefficients will be used to filter the signal.
7. Applies the filter to the signal using the “filtfilt” function from SciPy's signal processing module. The function performs zero-phase filtering, which means that it applies the filter twice, once forward and once backward, to eliminate phase distortion. The filtered signal is assigned to the variable “filtered_signal”.
8. Plots the original signal and the filtered signal using Matplotlib's “plot” function. The time axis “t” is used as the x-axis, and the signals are used as the y-axis. The “label” argument is used to specify the legend labels for the two signals.
9. Sets the title of the plot using Matplotlib's “title” function.
10. Sets the label of the x-axis using Matplotlib's “xlabel” function.
11. Sets the label of the y-axis using Matplotlib's “ylabel” function.
12. Adds a legend to the plot using Matplotlib's “legend” function.
13. Displays the plot using Matplotlib's “show” function.

```
[14]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, num=100)
signal = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 10 * t)
b, a = sig.butter(3, [0.2, 0.8], 'bandpass')
filtered_signal = sig.filtfilt(b, a, signal)

plt.plot(t, signal, label='Original signal')
plt.plot(t, filtered_signal, label='Filtered signal')
plt.title('Signal filtering with a Butterworth bandpass filter')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```



1.0.13 Example 13 - High Pass Filter design -

This example designs a Butterworth highpass filter, calculates its frequency response, and plots the result. This Python code designs a highpass Butterworth filter with a cutoff frequency of 0.2 and plots its frequency response. The code first imports the NumPy, Matplotlib, and SciPy libraries, which are used to design and plot the filter. The `butter` function from SciPy is then used to design a highpass filter with a cutoff frequency of 0.2 Hz and a filter order of 3. The `freqz` function from SciPy is then used to compute the frequency response of the filter. Finally, the code plots the magnitude response of the filter using the `plot` function from Matplotlib. This Python code demonstrates the use of the NumPy, SciPy, and Matplotlib libraries to plot the frequency response of a Butterworth highpass filter.

1. imports the NumPy library and renames it as “np”.
2. imports the signal processing module from the SciPy library and renames it as “sig”.
3. imports the Pyplot module from Matplotlib and renames it as “plt”.
4. Uses the “butter” function from SciPy’s signal processing module to design a Butterworth highpass filter. The function takes three arguments: the filter order (3 in this case), the cutoff frequency of the filter (0.2 in this case, specified as a fraction of the Nyquist frequency), and the type of filter (‘highpass’ in this case). The function returns the filter coefficients as two

arrays, which are assigned to the variables “b” and “a”.

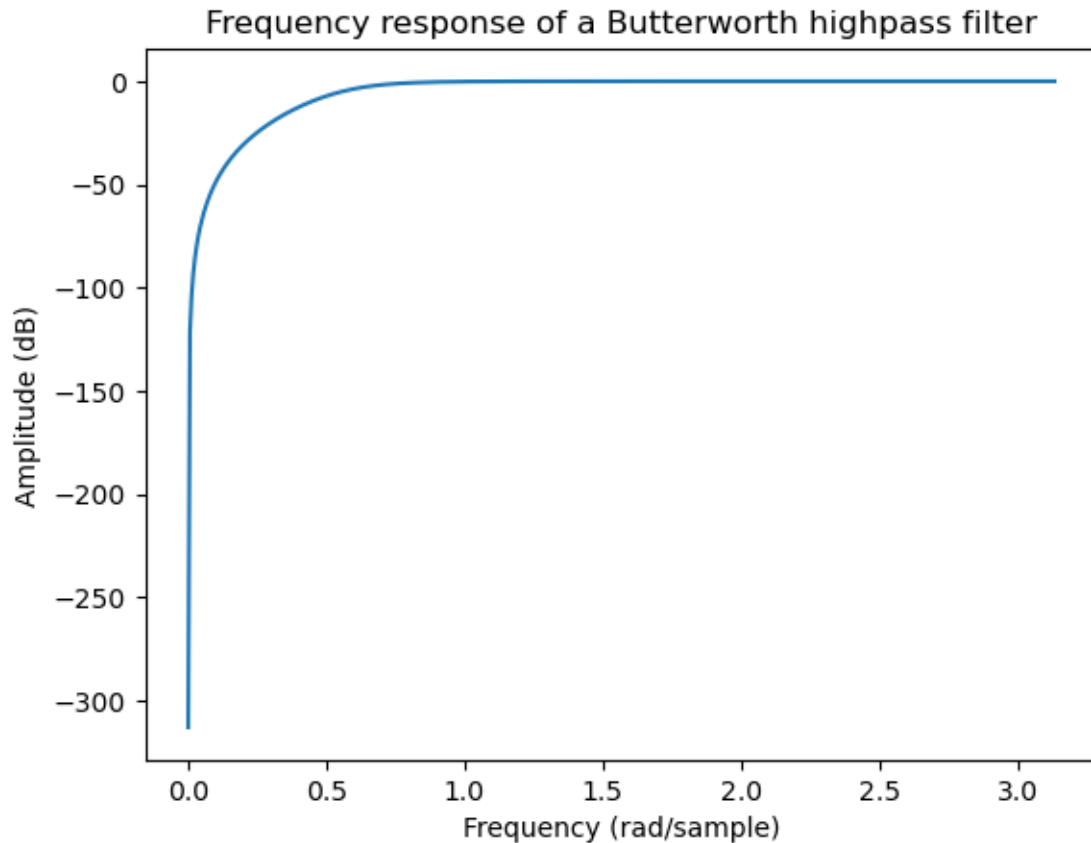
5. Uses the “freqz” function from SciPy’s signal processing module to compute the frequency response of the filter. The function takes the filter coefficients “b” and “a” as arguments and returns the frequency response as two arrays, which are assigned to the variables “w” and “h”. The “w” array contains the frequencies (in radians per sample) at which the frequency response is computed, and the “h” array contains the complex values of the frequency response.
6. Clips the values of “h” that are smaller than the machine epsilon (the smallest representable positive number for the given floating-point data type). This is done to avoid division by zero in the next step.
7. Computes the magnitude of the frequency response in decibels using the formula $20 \cdot \log_{10}(|H(w)|)$, where $|H(w)|$ is the absolute value of the complex frequency response. The result is assigned to the variable “db”.
8. Plots the frequency response using Matplotlib’s “plot” function. The “w” array is used as the x-axis, and the “db” array is used as the y-axis.
9. Sets the title of the plot using Matplotlib’s “title” function.
10. Sets the label of the x-axis using Matplotlib’s “xlabel” function.
11. Sets the label of the y-axis using Matplotlib’s “ylabel” function.
12. Displays the plot using Matplotlib’s “show” function.

```
[15]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

b, a = sig.butter(3, 0.2, 'highpass')
w, h = sig.freqz(b, a)

h[np.abs(h) < np.finfo(float).eps] = np.finfo(float).eps # avoid division by
↳ zero
db = 20 * np.log10(np.abs(h))

plt.plot(w, db)
plt.title('Frequency response of a Butterworth highpass filter')
plt.xlabel('Frequency (rad/sample)')
plt.ylabel('Amplitude (dB)')
plt.show()
```



1.0.14 Example 14 - Other filters -

This Python code demonstrates the use of the NumPy, SciPy, and Matplotlib libraries to filter a signal using a median filter https://en.wikipedia.org/wiki/Median_filter.

1. imports the NumPy library and renames it as “np”.
2. imports the signal processing module from the SciPy library and renames it as “sig”.
3. imports the Pyplot module from Matplotlib and renames it as “plt”.
4. Uses the “linspace” function from NumPy to generate a sequence of 100 equally spaced time points between 0 and 1, inclusive. The sequence is assigned to the variable “t”.
5. Generates a signal by summing two sine waves with frequencies of 5 and 10 Hz, respectively, and amplitudes of 1. The signal is assigned to the variable “signal”.
6. Applies a median filter to the signal using the “medfilt” function from SciPy’s signal processing module. The function takes two arguments: the signal to be filtered (“signal” in this case), and the size of the median filter window (5 in this case, which means that the median is computed over 5 samples at a time). The filtered signal is assigned to the variable “filtered_signal”.
7. Plots the original signal and the filtered signal using Matplotlib’s “plot” function. The “t” array is used as the x-axis, and the “signal” and “filtered_signal” arrays are used as the y-axis

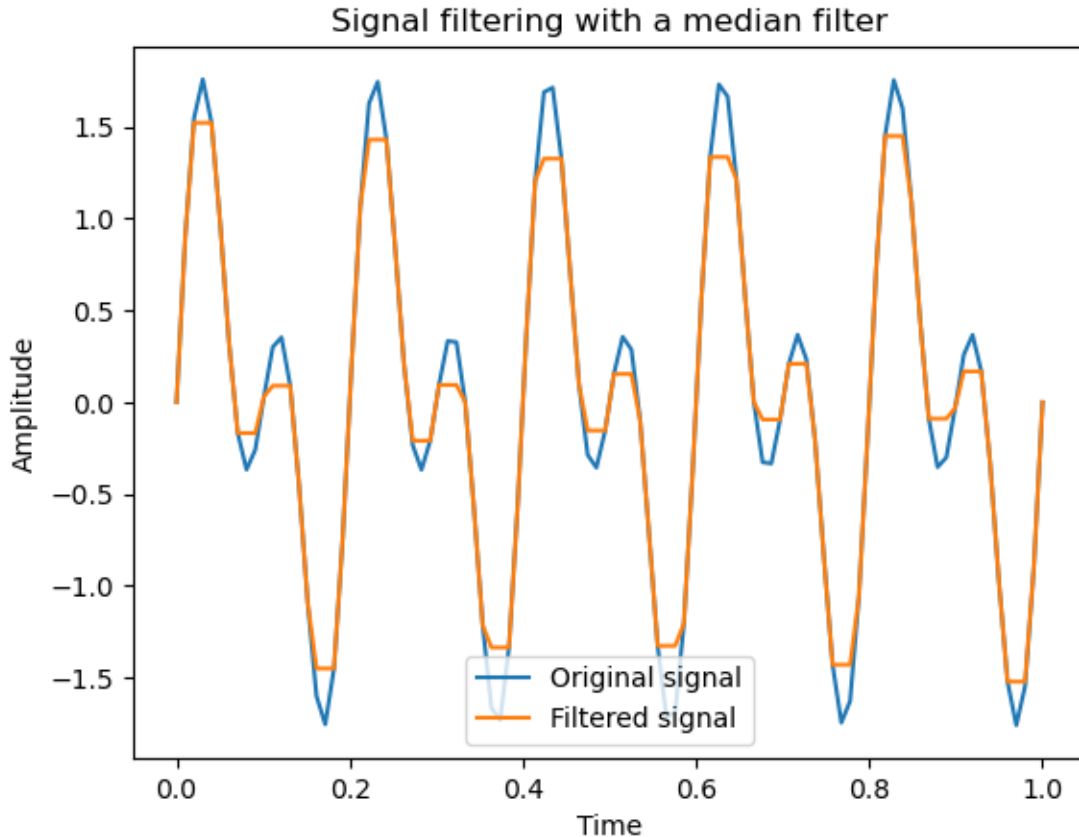
for the original and filtered signals, respectively. The “label” argument is used to provide a legend for the plot.

8. Sets the title of the plot using Matplotlib’s “title” function.
9. Sets the label of the x-axis using Matplotlib’s “xlabel” function.
10. Sets the label of the y-axis using Matplotlib’s “ylabel” function.
11. Creates a legend for the plot using Matplotlib’s “legend” function.
12. Displays the plot using Matplotlib’s “show” function.

```
[16]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, num=100)
signal = np.sin(2 * np.pi * 5 * t) + np.sin(2 * np.pi * 10 * t)
filtered_signal = sig.medfilt(signal, kernel_size=5)

plt.plot(t, signal, label='Original signal')
plt.plot(t, filtered_signal, label='Filtered signal')
plt.title('Signal filtering with a median filter')
plt.xlabel('Time')
plt.ylabel('Amplitude')
plt.legend()
plt.show()
```



1.0.15 Example 15 - Analog filter -

This Python code uses the SciPy library to design and plot the frequency response of an analog bandpass Chebyshev type II filter https://en.wikipedia.org/wiki/Chebyshev_filter.

Here is an explanation of the code line by line:

1. *import numpy as np*: Import the NumPy library with the alias np.
2. *import scipy.signal as sig*: Import the SciPy signal processing module with the alias sig.
3. *import matplotlib.pyplot as plt*: Import the pyplot module of the Matplotlib library with the alias plt.
4. *wp = 0.2*: Set the passband edge frequency to 0.2, which is a normalized frequency (ranging from 0 to 1) that specifies the frequency at which the filter's gain is 3 dB less than its maximum value in the passband.
5. *ws = 0.8*: Set the stopband edge frequency to 0.8, which is a normalized frequency that specifies the frequency at which the filter's gain is 3 dB or more less than its maximum value in the passband.
6. *rs = 60*: Set the stopband attenuation to 60 dB, which is a measure of how much the filter's gain is reduced in the stopband relative to the passband.
7. *b, a = sig.iirfilter(3, [wp, ws], rp=None, rs=rs, btype='bandpass', ftype='cheby2')*: Use the iirfilter function of the sig module to design a third-order Chebyshev type II bandpass filter

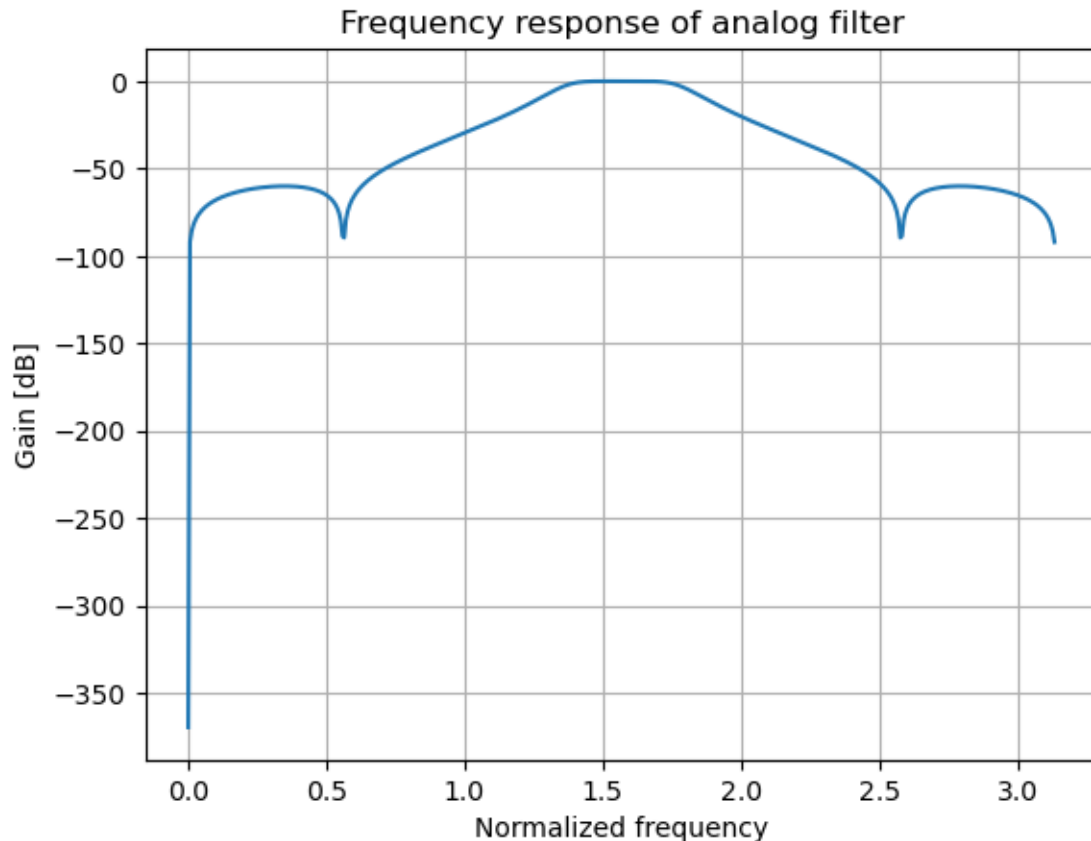
with passband edge frequency w_p , stopband edge frequency w_s , stopband attenuation r_s , band type 'bandpass', and filter type 'cheby2'. The function returns the filter coefficients b and a , which are used to compute the frequency response of the filter in the next line.

8. $w, h = \text{sig.freqz}(b, a)$: Use the `freqz` function of the `sig` module to compute the frequency response of the filter with coefficients b and a . The function returns the normalized frequency w (ranging from 0 to π) and the complex frequency response h .
9. $\text{plt.plot}(w, 20 \cdot \log_{10}(\text{np.abs}(h)))$: Plot the frequency response of the filter as a function of the normalized frequency w using Matplotlib's `plot` function. The y-axis is converted from the linear scale to the logarithmic scale in decibels (dB) using the formula $20 \cdot \log_{10}(\text{abs}(h))$, where $\text{abs}(h)$ is the magnitude of the frequency response.
10. $\text{plt.title}(\text{'Frequency response of analog filter'})$: Set the title of the plot to 'Frequency response of analog filter'.
11. $\text{plt.xlabel}(\text{'Normalized frequency'})$: Set the label of the x-axis to 'Normalized frequency'.
12. $\text{plt.ylabel}(\text{'Gain [dB]'})$: Set the label of the y-axis to 'Gain [dB]'.
13. $\text{plt.grid}()$: Add a grid to the plot.
14. $\text{plt.show}()$: Display the plot.

```
[17]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

wp = 0.2
ws = 0.8
rs = 60
b, a = sig.iirfilter(3, [wp, ws], rp=None, rs=rs, btype='bandpass',
                    ftype='cheby2')
w, h = sig.freqz(b, a)

plt.plot(w, 20 * np.log10(np.abs(h)))
plt.title('Frequency response of analog filter')
plt.xlabel('Normalized frequency')
plt.ylabel('Gain [dB]')
plt.grid()
plt.show()
```



1.0.16 Example 16 - Periodogram measurements (Welch's method) -

This example computes the power spectral density of a signal using Welch's method and plots the result.

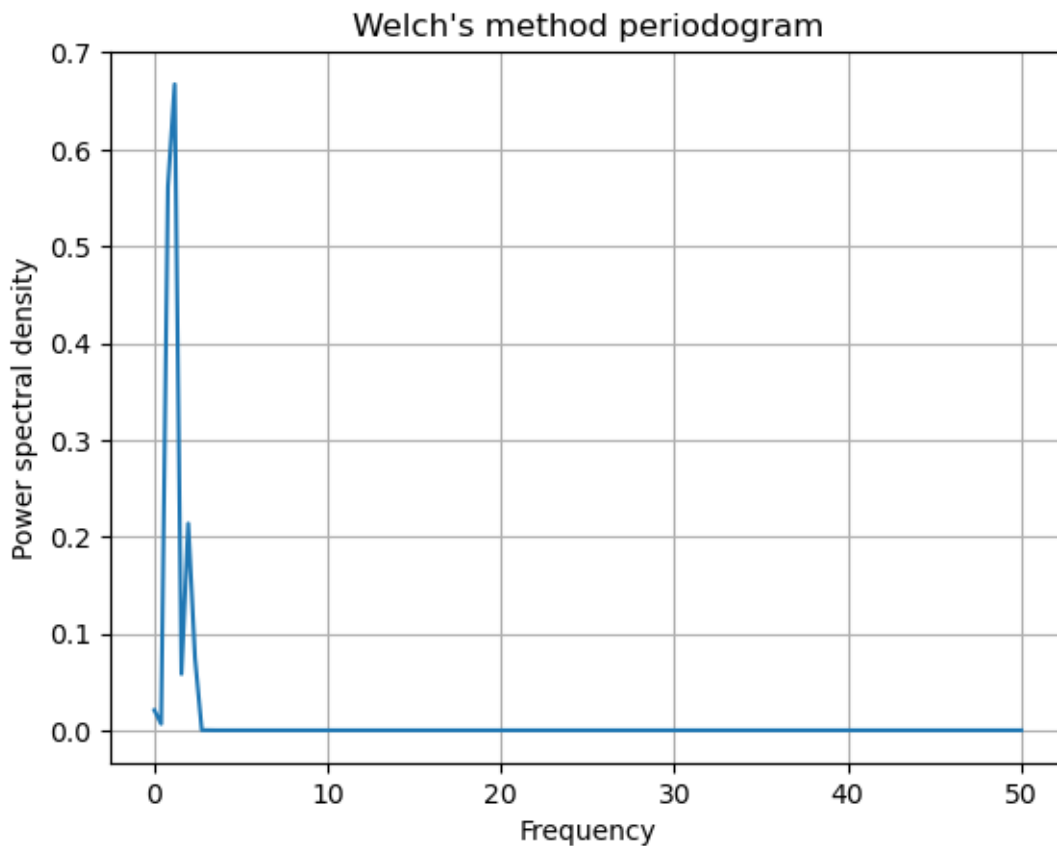
1. `import numpy as np`: Import the NumPy library and alias it as "np" for convenience.
2. `import scipy.signal as sig`: Import the SciPy signal processing module and alias it as "sig" for convenience.
3. `import matplotlib.pyplot as plt`: Import the Matplotlib library and alias it as "plt" for convenience, which is used for plotting.
4. `t = np.linspace(0, 1, 1000)`: Generate a 1-second time vector with 1000 evenly spaced points between 0 and 1.
- `signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)`: Create a time-domain signal that consists of a sine wave with a frequency of 10 Hz and a smaller amplitude sine wave with a frequency of 20 Hz.
5. `f, p = sig.welch(signal, fs=100, window='hann', nperseg=256, noverlap=128)`: Compute the Welch's method periodogram of the signal, which is a power spectral density estimate obtained by averaging modified periodograms of the signal segments with overlapping windows. `fs` is the sampling frequency, `window` is the window function applied to each segment, `nperseg` is the length of each segment, and `noverlap` is the number of points of overlap between segments. This returns the frequency vector `f` and the corresponding power spectral density vector `p`.
6. `plt.plot(f, p)`: Plot the power spectral density estimate against the frequency vector.
7. `plt.title("Welch's method periodogram")`: Add a title to the plot.
8. `plt.xlabel('Frequency')`: Add a label to the x-axis.
9. `plt.ylabel('Power spectral density')`: Add a label to the y-axis.
10. `plt.grid()`: Add grid lines to the plot.
11. `plt.show()`: Display the plot.


```
[18]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, 1000)
signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)

f, p = sig.welch(signal, fs=100, window='hann', nperseg=256, noverlap=128)

plt.plot(f, p)
plt.title("Welch's method periodogram")
plt.xlabel('Frequency')
plt.ylabel('Power spectral density')
plt.grid()
plt.show()
```



1.0.17 Example 17 - Lomb-Scargle periodograms -

This example computes the Lomb-Scargle periodogram https://en.wikipedia.org/wiki/Least-squares_spectral_analysis#The_Lomb-Scargle_periodogram of a signal and plots the re-

sult. This code generates a Lomb-Scargle periodogram for a signal consisting of two sinusoids with frequencies 10 Hz and 20 Hz. Here's a breakdown of the code line by line:

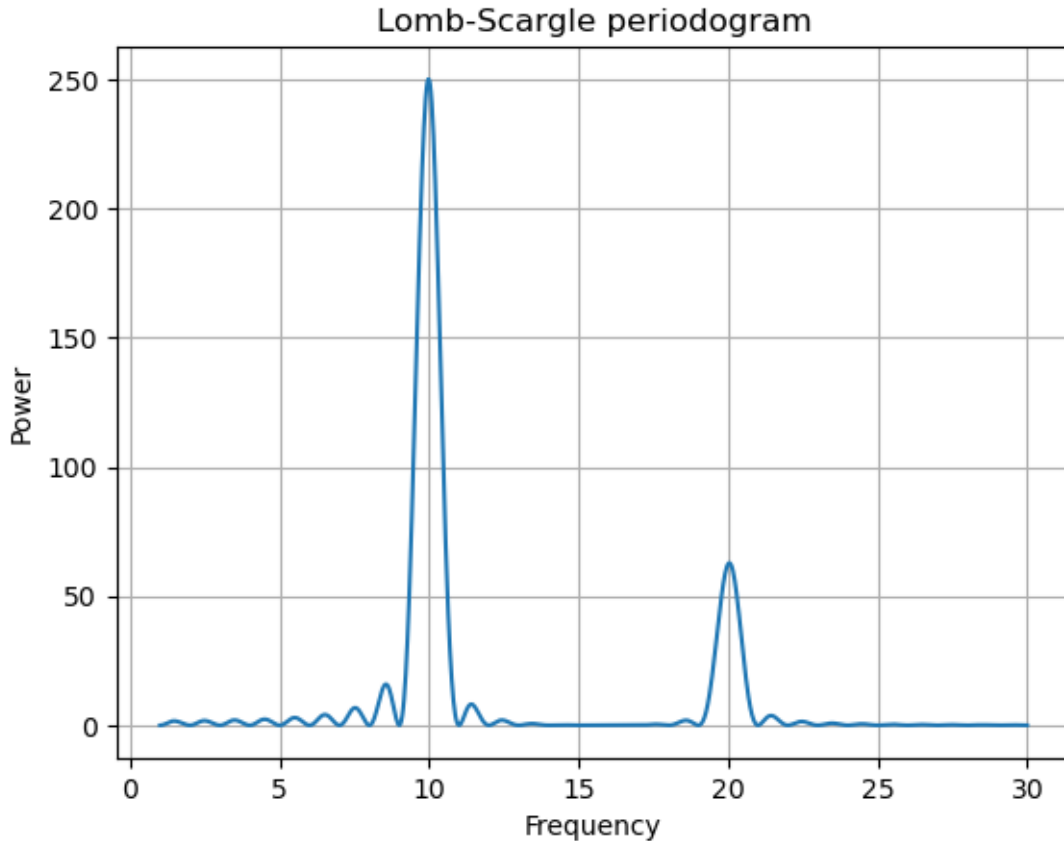
1. `import numpy as np`: Imports the NumPy library, which provides support for numerical operations on arrays and matrices.
2. `import scipy.signal as sig`: Imports the SciPy library, which provides a variety of signal processing functions.
3. `import matplotlib.pyplot as plt`: Imports the Matplotlib library, which provides functions for creating visualizations of data.
4. `t = np.linspace(0, 1, 1000)`: Creates an array of 1000 evenly spaced time values between 0 and 1.
5. `signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)`: Creates a signal consisting of two sinusoids with frequencies 10 Hz and 20 Hz, respectively. The sin function is applied to each frequency and the resulting signals are added together.
6. `f = np.linspace(1, 30, 1000)`: Creates an array of 1000 evenly spaced frequencies between 1 and 30 Hz.
7. `p = sig.lombscargle(t, signal, 2 * np.pi * f)`: *Calculates the Lomb-Scargle periodogram for the input signal signal. The function lombscargle computes the Lomb-Scargle periodogram, which is a frequency analysis method for irregularly sampled data. The inputs to the function are the time values t, the signal signal, and the frequencies f multiplied by 2pi to convert them to radians.*
8. `plt.plot(f, p)`: Plots the Lomb-Scargle periodogram. The frequency values are on the x-axis and the power values are on the y-axis.
9. `plt.title("Lomb-Scargle periodogram")`: Sets the title of the plot to "Lomb-Scargle periodogram".
10. `plt.xlabel('Frequency')`: Sets the label of the x-axis to "Frequency".
11. `plt.ylabel('Power')`: Sets the label of the y-axis to "Power".
12. `plt.grid()`: Adds a grid to the plot.
13. `plt.show()`: Displays the plot.

```
[19]: import numpy as np
import scipy.signal as sig
import matplotlib.pyplot as plt

t = np.linspace(0, 1, 1000)
signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)

f = np.linspace(1, 30, 1000)
p = sig.lombscargle(t, signal, 2 * np.pi * f)

plt.plot(f, p)
plt.title("Lomb-Scargle periodogram")
plt.xlabel('Frequency')
plt.ylabel('Power')
plt.grid()
plt.show()
```



1.0.18 Example 18 - Discrete Cosine Transform -

This code generates a discrete cosine transform (DCT) https://en.wikipedia.org/wiki/Discrete_cosine_transform plot for a given signal. The resulting plot shows the amplitude of each DCT coefficient, with the x-axis representing the index of each coefficient and the y-axis representing its amplitude. Since the signal has two frequency components, there are two non-zero coefficients in the DCT plot. The first coefficient corresponds to the DC component of the signal, while the second coefficient corresponds to the 20 Hz component of the signal. Here's how it works:

1. The numpy library is imported as np and the scipy.fftpack library is imported as fft.
2. A time vector t is created using numpy's linspace() function. This vector contains 1000 evenly spaced values between 0 and 1.
3. A signal is generated by adding two sine waves together: one with a frequency of 10 Hz and another with a frequency of 20 Hz. The resulting signal is stored in the variable signal.
4. The DCT of the signal is computed using fft.dct(). The type argument is set to 2 to indicate the use of the inverse DCT algorithm, which generates a type-II DCT.
5. The resulting DCT coefficients are stored in the variable C.
6. The matplotlib library is used to create a plot of the DCT coefficients. The plot() function is

called with the `C` variable as its argument. The plot is labeled with a title, axis labels, and a grid.

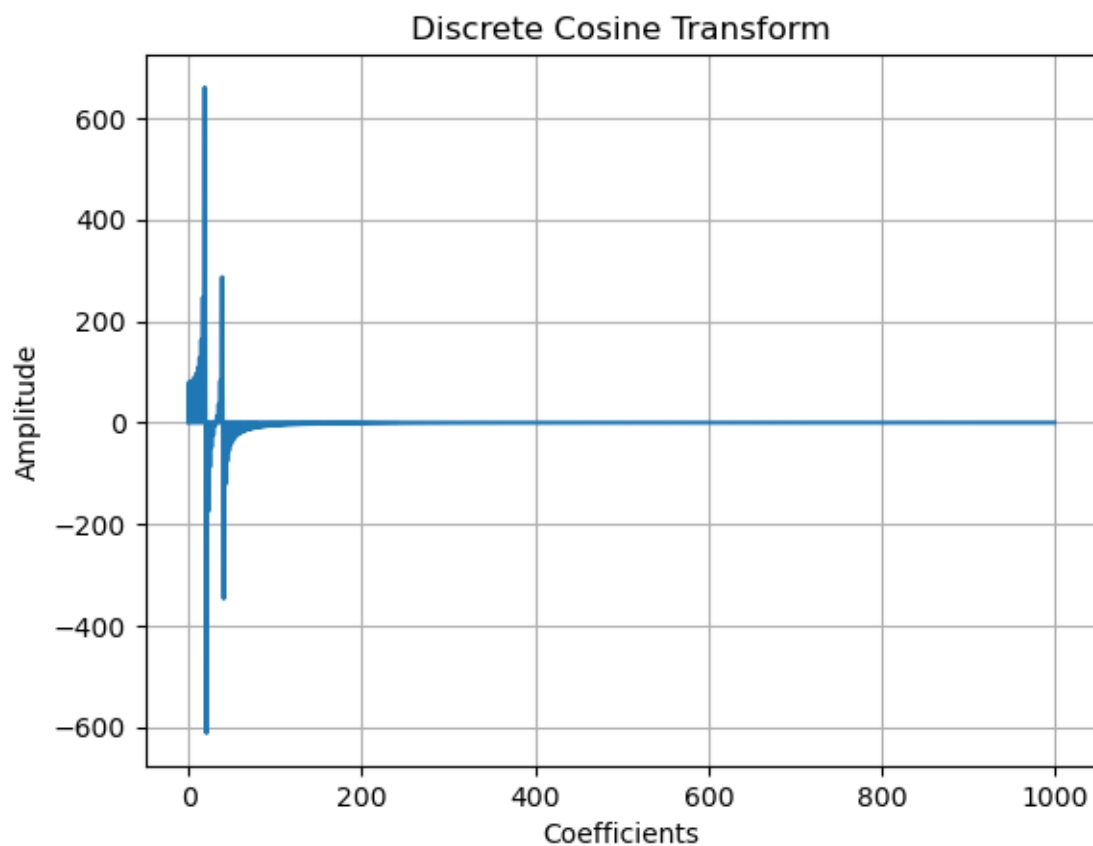
7. Finally, the plot is displayed using the `show()` function.

```
[20]: import numpy as np
import scipy.fftpack as fft
import matplotlib.pyplot as plt

t = np.linspace(0, 1, 1000)
signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)

C = fft.dct(signal, type=2)

plt.plot(C)
plt.title("Discrete Cosine Transform")
plt.xlabel('Coefficients')
plt.ylabel('Amplitude')
plt.grid()
plt.show()
```



1.0.19 Example 19 - Discrete Sine Transform -

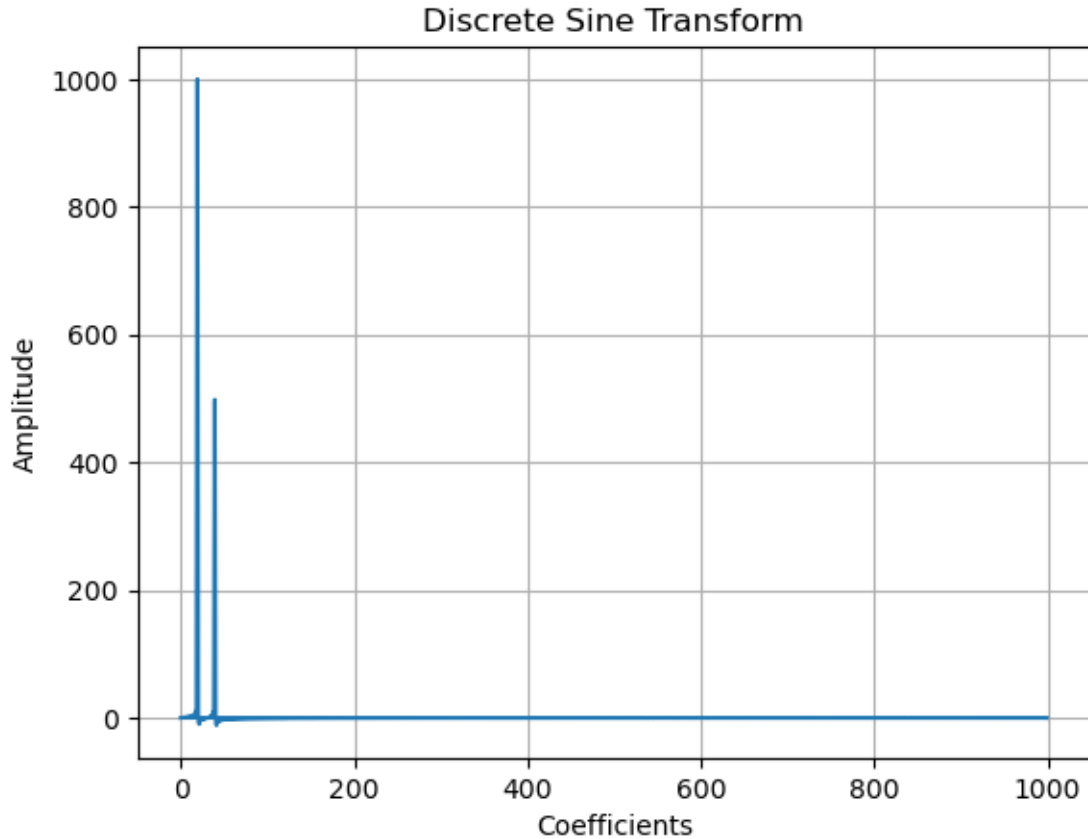
This code generates a signal consisting of two sine waves, computes its DST https://en.wikipedia.org/wiki/Discrete_sine_transform, and plots the DST coefficients using pyplot. The plot shows how the signal is decomposed into a set of sine functions with different frequencies and amplitudes. 1. *import numpy as np*: Imports the NumPy package and creates an alias 'np' for easy referencing in the code. 2. *import scipy.fftpack as fft*: Imports the FFTPACK module from SciPy package and creates an alias 'fft' for easy referencing in the code. 3. *import matplotlib.pyplot as plt*: Imports the pyplot module from the matplotlib package and creates an alias 'plt' for easy referencing in the code. 4. *t = np.linspace(0, 1, 1000)*: Generates a 1D array of 1000 equally spaced values between 0 and 1 and assigns it to the variable 't'. 5. *signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)*: Generates a signal by adding two sine waves with frequencies 10 Hz and 20 Hz, respectively, and assigns it to the variable 'signal'. 6. *S = fft.dst(signal)*: Computes the Discrete Sine Transform (DST) of the signal using the FFTPACK module and assigns it to the variable 'S'. 7. *plt.plot(S)*: Plots the DST coefficients of the signal using pyplot. 8. *plt.title("Discrete Sine Transform")*: Sets the title of the plot to "Discrete Sine Transform". 9. *plt.xlabel('Coefficients')*: Sets the label for the x-axis to "Coefficients". 10. *plt.ylabel('Amplitude')*: Sets the label for the y-axis to "Amplitude". 11. *plt.grid()*: Adds grid lines to the plot. 12. *plt.show()*: Displays the plot on the screen.

```
[21]: import numpy as np
import scipy.fftpack as fft
import matplotlib.pyplot as plt

t = np.linspace(0, 1, 1000)
signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)

S = fft.dst(signal)

plt.plot(S)
plt.title("Discrete Sine Transform")
plt.xlabel('Coefficients')
plt.ylabel('Amplitude')
plt.grid()
plt.show()
```



1.0.20 Example 20 - The Hilbert Transform -

In this code example, the `hilbert()` function from the `scipy.signal` module is used to perform the Hilbert Transform https://en.wikipedia.org/wiki/Hilbert_transform. The magnitude of the analytic signal is then calculated using `np.abs()` and plotted using `matplotlib`. First, let me give you a brief overview of what this code does. It generates a signal that is a sum of two sine waves with frequencies 10Hz and 20Hz, applies the Hilbert transform to obtain the analytic signal, calculates the absolute value of the analytic signal, and then plots the resulting amplitude values. This allows you to see the envelope of the original signal, which is the result of the Hilbert transform. Here's a line-by-line explanation of the code:

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

This code imports three Python libraries that will be used in the rest of the code. `numpy` is a library for numerical computing, `matplotlib` is a library for data visualization, and `scipy` is a library for scientific computing. `hilbert` is a function provided by the `scipy.signal` module that performs the Hilbert transform.

```
t = np.linspace(0, 1, 1000)
```

This code creates an array of 1000 evenly spaced values between 0 and 1. This array will be used as the time values for the signal.

```
signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)
```

This code generates a signal that is the sum of two sine waves with frequencies 10Hz and 20Hz. The signal is stored in the variable `signal`.

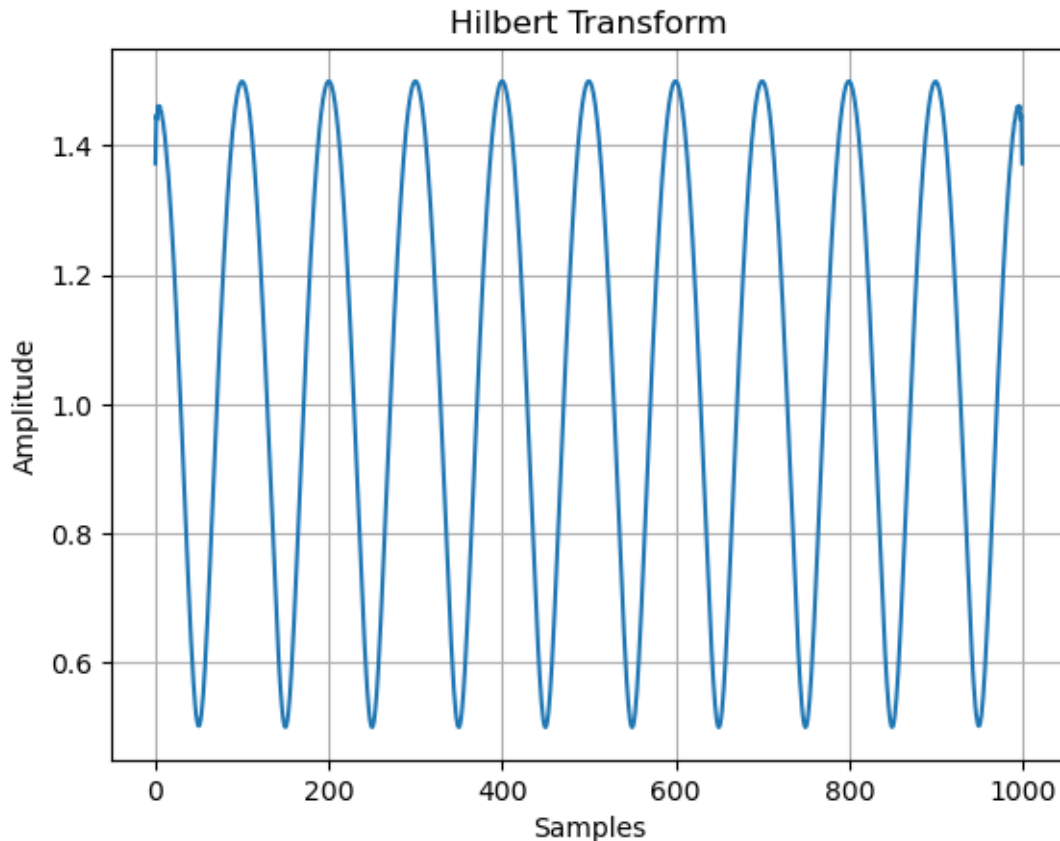
```
analytic_signal = hilbert(signal) H = np.abs(analytic_signal)
```

This code applies the Hilbert transform to the signal using the `hilbert` function provided by the `scipy.signal` module. The result is stored in the variable `analytic_signal`. The `np.abs` function is then used to calculate the absolute value of the analytic signal, which gives us the amplitude of the envelope of the original signal. The resulting amplitude values are stored in the variable `H`.

```
plt.plot(H) plt.title("Hilbert Transform") plt.xlabel('Samples') plt.ylabel('Amplitude')  
plt.grid() plt.show()
```

This code generates a plot of the amplitude values stored in `H`. The `plt.plot` function is used to create the plot, and the `plt.title`, `plt.xlabel`, and `plt.ylabel` functions are used to add a title and axis labels to the plot. The `plt.grid` function adds a grid to the plot. Finally, the `plt.show` function is used to display the plot on the screen.

```
[22]: import numpy as np  
import matplotlib.pyplot as plt  
from scipy.signal import hilbert  
  
t = np.linspace(0, 1, 1000)  
signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.sin(2 * np.pi * 20 * t)  
  
analytic_signal = hilbert(signal)  
H = np.abs(analytic_signal)  
  
plt.plot(H)  
plt.title("Hilbert Transform")  
plt.xlabel('Samples')  
plt.ylabel('Amplitude')  
plt.grid()  
plt.show()
```



1.0.21 Example 21 - Spectrogram and MFCC coefficients -

This code demonstrates how to compute and visualize the spectrogram and MFCC (Mel-frequency cepstral coefficients) https://en.wikipedia.org/wiki/Mel-frequency_cepstrum of an audio file using the librosa library. Let's break down the code step by step: Importing necessary libraries python

```
import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt
```

Loading the audio file: `> audio_file = 'm2_script1_iphone_livingroom1.wav'` audio, sr = librosa.load(audio_file, sr=None)

The librosa.load() function is used to load the audio file. It returns the audio data (audio) and the sample rate (sr) of the audio.

Computing the spectrogram: `> spectrogram = np.abs(librosa.stft(audio))`

The spectrogram is calculated by applying the Short-Time Fourier Transform (STFT) to the audio data. librosa.stft() computes the STFT, and np.abs() is used to get the magnitude of the STFT.

Displaying the spectrogram: `> plt.figure(figsize=(10, 4)) librosa.display.specshow(librosa.amplitude_to_db(spectrogram, ref=np.max), sr=sr,`


```
x_axis='time', y_axis='log') plt.colorbar(format='%+2.0f dB') plt.title('Spectrogram') plt.show()
>
```

This code visualizes the spectrogram using `librosa.display.specshow()`. `librosa.amplitude_to_db()` converts the spectrogram to dB scale for better visualization. The resulting spectrogram is displayed using `plt.imshow()`, and a colorbar and title are added.

```
Computing the MFCC coefficients: > mfcc = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13) >
```

The MFCC coefficients are computed using `librosa.feature.mfcc()`. The `y` parameter specifies the audio data, `sr` is the sample rate, and `n_mfcc` determines the number of MFCC coefficients to compute.

```
Displaying the MFCC coefficients: > plt.figure(figsize=(10, 4)) librosa.display.specshow(mfcc,
x_axis='time') plt.colorbar() plt.title('MFCC Coefficients') plt.show() >
```

This code displays the MFCC coefficients using `librosa.display.specshow()`. The resulting MFCC coefficients are visualized using `plt.imshow()`, and a colorbar and title are added.

1. Import the necessary libraries: `numpy` for numerical operations, `librosa` for audio processing, and `matplotlib` for plotting.
2. Provide the path to the audio file that you want to analyze.
3. Load the audio file using `librosa.load()`.
4. The `sr=None` argument ensures that the audio is loaded in its original sampling rate.
5. Compute the spectrogram using `librosa.stft()`. This function converts the audio signal from the time domain to the frequency domain.
6. Display the spectrogram using `librosa.display.specshow()`. The `librosa.amplitude_to_db()` function converts the amplitude values to decibels for better visualization. The `sr` parameter is set to the original sampling rate, and the `x_axis` is set to `'time'` to display the time on the x-axis. The `y_axis` is set to `'log'` to display the frequencies on a logarithmic scale.
7. Show a colorbar indicating the magnitude of the spectrogram in decibels.
8. Compute the MFCC (Mel-frequency cepstral coefficients) using `librosa.feature.mfcc()`. MFCCs are commonly used features for audio analysis and represent the spectral envelope of the audio signal.
9. Display the MFCC coefficients using `librosa.display.specshow()`. Since MFCCs are a 2D matrix, the function directly displays the coefficients as an image.
10. Show a colorbar indicating the magnitude of the MFCC coefficients.

By running this code with a valid audio file path, you will visualize the spectrogram and MFCC coefficients of the audio.

```
[23]: import numpy as np
import librosa
import librosa.display
import matplotlib.pyplot as plt

# Load audio file
audio_file = 'm2_script1_iphone_livingroom1.wav'
#audio_file = 'file_example_WAV_5MG.wav'
audio, sr = librosa.load(audio_file, sr=None)

# Compute spectrogram
spectrogram = np.abs(librosa.stft(audio))

# Display spectrogram
```

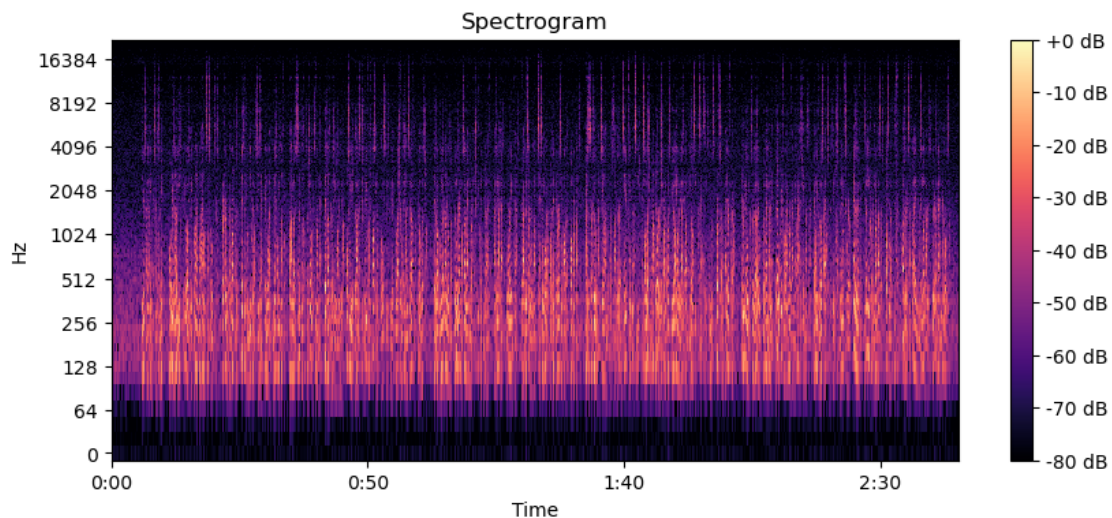
```

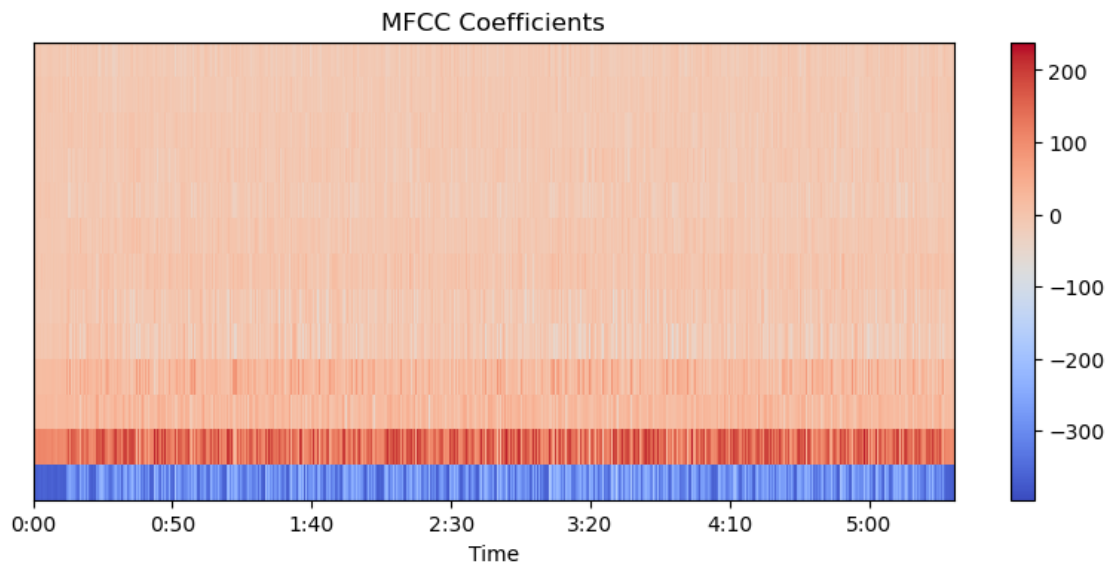
plt.figure(figsize=(10, 4))
librosa.display.specshow(librosa.amplitude_to_db(spectrogram, ref=np.max),
                        sr=sr, x_axis='time', y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Spectrogram')
plt.show()

# Compute MFCC coefficients
mfcc = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13)

# Display MFCC coefficients
plt.figure(figsize=(10, 4))
librosa.display.specshow(mfcc, x_axis='time')
plt.colorbar()
plt.title('MFCC Coefficients')
plt.show()

```





[]: