CS448 - Lab 1: Forward and Inverse STFT

The purpose of this lab is to familiarize you with taking a sound to the time/frequency domain and back. You will code a spectrogram routine, its inverse, and then run some examples to see the effects of various parameters when performing such analyses. Although you can find existing functions to perform some of these calculations, you will have to develop your own version from scratch. This will allow you to perform some more complex processing later in the semester, and of course it will also give you a deeper understanding of how things work.

You will likely reuse a lot of this code in future labs, so this is not one of the labs to skip!

Part 1. The Forward Transform

You need to design a function that uses five different arguments as follows:

```
stft_output = stft( input_sound, dft_size, hop_size, zero_pad, window)
    input_sound is a 1d array that contains an input sound.
    dft_size is the DFT point size that you will use for this analysis.
    hop_size is the number of samples that your analysis frame will advance.
    zero_pad is the amount of zero-padding that you will use.
    window is a vector containing the analysis window that you will be using.
```

To complete this you need to perform the following steps:

- You need to segment the input array as shorter frames which are `dft_size` samples long. Each frame will start `hop_size` samples after the beginning of the previous one. In practice, `hop_size` will be smaller than `dft_size`, usually by a factor of 2 or 4. Feel free to add some zeros at the beginning and/or end of the input so that you have enough samples to compose the last frame at the desired length.
- You will then need to compute the Discrete Fourier Transform (DFT) of each frame. For each input frame you will get a complex-valued vector containing its spectrum. Take all of these vectors and concatenate them as columns of a matrix. The `{i,j}` element of this matrix will contain the coefficient for frequency `i` at input frame `j`. Note that there is a variety of Fourier options in numpy. Since we will be using real-valued signals you should use the `fft.rfft` routine.
- You might notice that by doing only the above the output is a little noisy-looking. This is because we are not using an analysis window. In order to apply a window you need to multiply each analysis frame with a function that smoothly tapers the edges down to zero. This function will be provided as the function input `window`, which will have to have the same length as the analysis frames (i.e. `dft_size` samples). Typical window shapes are the triangle window (goes from 0 to 1 to 0), the Hann window (see the incorrectly-named function `hanning`), the Hamming window (`hamming`), and the Kaiser window (`kaiser`).
- Finally, we will add the option to zero pad the input. Doing so will allow us to obtain smoother looking outputs when `dft_size` is small (remember that zero padding in the time domain results in interpolation in the frequency domain). To do so you can append `zero_pad` zeros at the end of each analysis frame. Alternatively you can use the `fft.rfft` function's `size` variable and ask it to perform a DFT of size `dft_size+zero_pad`, which will implicitly add zeros to its input.

You should now have a complete forward Short-Time Fourier Transform routine. Try it on the following example sounds:

- Drum clip: [https://drive.google.com/uc?export=download&id=1e-pLopbM4WyOFadoxu78E77EvXX4OtYb]
- Speech clip: [https://drive.google.com/uc?export=download&id=1dlOQHVi5po7S2CwIWvBsZMJpL17mbFgx]
- Piano clip: [https://drive.google.com/uc?export=download&id=1eEFfri_af_QXN4k7xQS2bntyS4VzHLsv]

and plot the magnitude of the result (you should use the `pcolormesh` function to plot it as an image). Try to find the best function parameters that allow you to see what's going on in the input sounds. You want to get a feel of what it means to change the DFT size, the hop size, the window and the amount of zero padding. Plot some results that demonstrate the effect of these parameters. As a rough guide, traditionally the hop size is 1/2, 1/4 or 1/8, of the DFT size, and historically the DFT size is almost always chosen to be a power of two (it's faster than othersize). Liekwise the zero padding is often a multiple of the DFT size (most of the time you just double the sequence length by zero padding).

Often, such plots lack significant contrast to make a good visualization. A good idea is to plot the log value of the magnitudes (beware of zeros), or to raise them to a small power, e.g. 0.3. This will create better looking plots where smaller differences are more visible. A good colormap is also essential, have a look at: [https://jakevdp.github.io/blog/2014/10/16/how-bad-is-your-colormap/] You want to use something with a linear luminance gradient.

Finally, I want you to make sure that the axes in your spectrogram plot are in terms of Hz on the y-axis and seconds on the x-axis.

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

In [ ]:
```python
# Make a sound player function that plays array "x" with a sample rate "rate", and labels it with "label"
def sound(x, rate=8000, label=''):
    from IPython.display import display, Audio, HTML
    display(
        HTML('<style> table, th, td {border: 0px; }</style> <table><tr><td>' +
             label + '</td><td>' + Audio(x, rate=rate)._repr_html_()[3:] +
             '</td></tr></table>'))
```

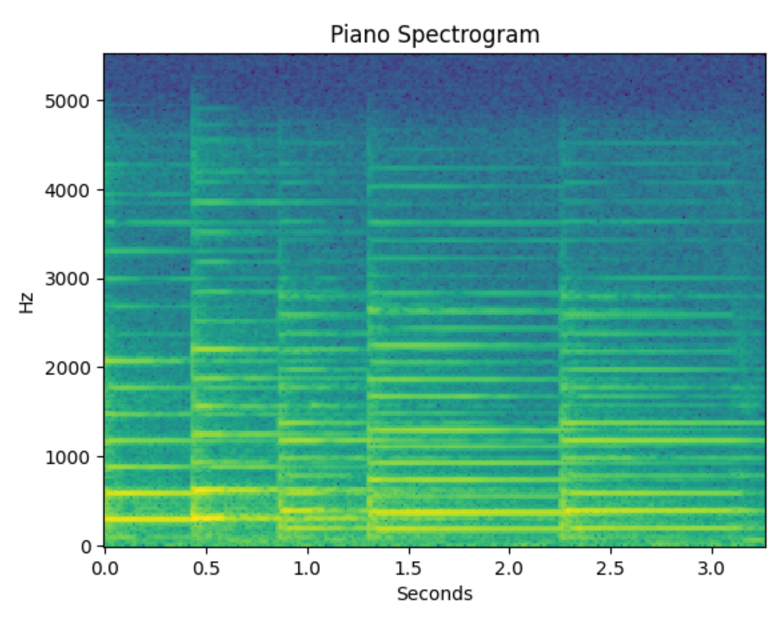
In [ ]:
```python
def plot_spectrogram(stft, input_sound, fs, title="Spectrogram"):
    # Taking the log of the spectrogram to make it more visible
    output = np.log(np.absolute(stft))
    X = np.linspace(0, len(input_sound) / fs, len(output))

    # Calculating the frequency axis
    freq = np.max(np.fft.fftfreq(len(input_sound), d=1 / fs))
    Y = np.linspace(0, freq, output.shape[1])

    plt.pcolormesh(X, Y, output.T)
    plt.title(title)
    plt.xlabel("Seconds")
    plt.ylabel("Hz")
    plt.show()
```

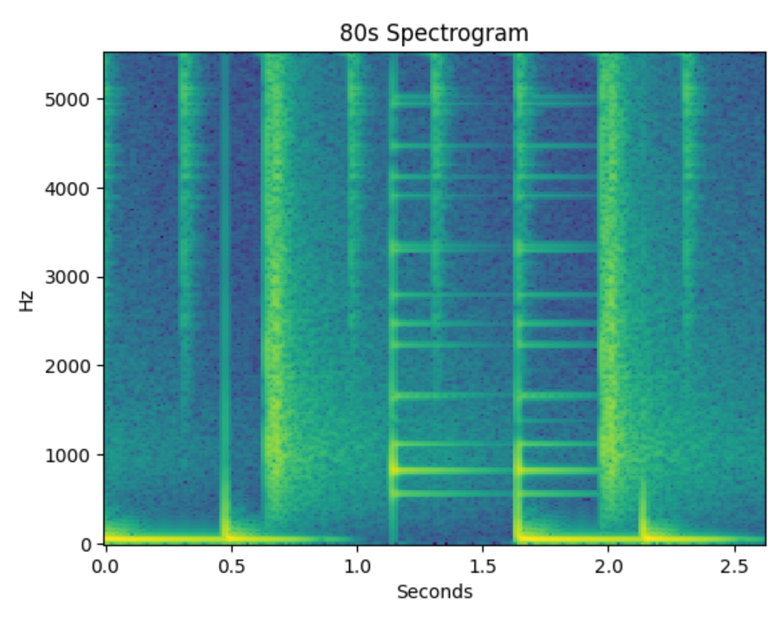
In [ ]:
```python
def stft(input_sound, dft_size, hop_size, zero_pad, window):
    # Creating the n-1 frames
    frames = []
    idx = 0
    for idx in range(0, len(input_sound) - dft_size, hop_size):
        frames.append(np.multiply(input_sound[idx:idx + dft_size], window))
    idx += hop_size

    # Creating the last frame accounting for padding
    last_frame = np.multiply(
        np.append(input_sound[idx:-1],
                  np.zeros(idx + dft_size - len(input_sound) + 1)), window)
    frames.append(last_frame)

    # Convert to numpy array
    frames = np.array(frames, dtype=float)

    # Compute the DFT of each frame
    dft_frames = np.fft.rfft(frames, dft_size + zero_pad)
    return dft_frames


# Load each sound
fs_piano, input_sound_piano = wavfile.read("./data/piano.wav")
sound(input_sound_piano, rate=fs_piano, label='piano.wav')

fs_80s, input_sound_80s = wavfile.read("./data/80s.wav")
sound(input_sound_80s, rate=fs_80s, label='80s.wav')

fs_speech, input_sound_speech = wavfile.read("./data/speech.wav")
sound(input_sound_speech, rate=fs_speech, label='speech.wav')

# STFT them
dft_size = 512
hop_size = dft_size // 4
zero_pad = 0
window = np.hanning(dft_size)

stft_piano = stft(input_sound_piano, dft_size, hop_size, zero_pad, window)
stft_80s = stft(input_sound_80s, dft_size, hop_size, zero_pad, window)
stft_speech = stft(input_sound_speech, dft_size, hop_size, zero_pad, window)

# Plot all the spectrograms
plot_spectrogram(stft_piano, input_sound_piano, fs_piano, "Piano Spectrogram")
plot_spectrogram(stft_80s, input_sound_80s, fs_piano, "80s Spectrogram")
plot_spectrogram(stft_speech, input_sound_speech, fs_piano, "Speech Spectrogram")
```

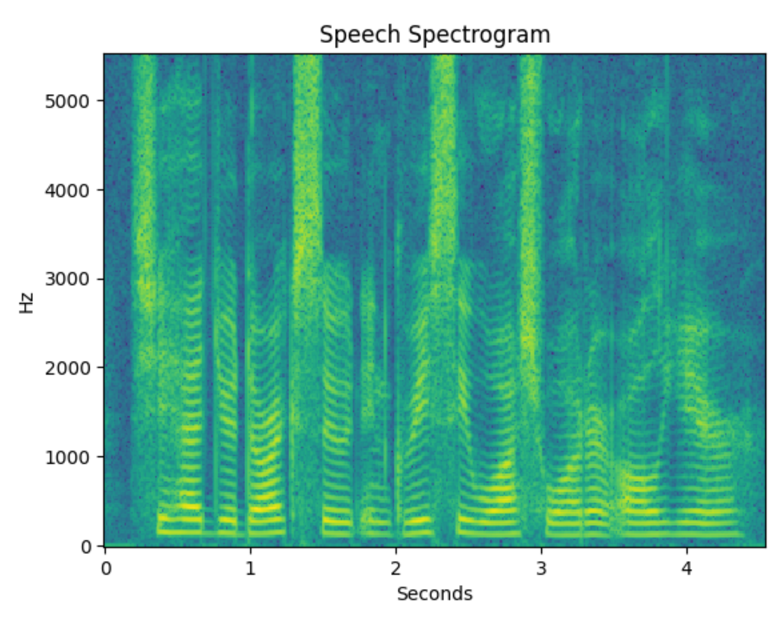
piano.wav -0:03

80s.wav -0:02

speech.wav -0:03







Part 2. The Inverse Transform

We will now implement a function that accepts the output of the function above, and returns the time-domain waveform that produces it. This is known as an inverse Short-Time Fourier Transform. This function will look as follows:

```
waveform = istft( stft_output, dft_size, hop_size, zero_pad, window)
    stft_output is the 2d array produced by the function you just did in part 1.
    dft_size is the DFT point size that you will use for the resynthesis.
    hop_size is the number of samples that your synthesis frames will advance.
    zero_pad is the amount of zero-padding that you have used originally.
    window is a vector containing the synthesis window that you will be using.
```

To perform the inverse transform you need to complete the following steps.

- Take each spectrum produced by the analysis and perform an inverse DFT on it. For each spectrum you should get back a small snippet of sound that was part of the original input.
- If the hop size you used is the same as the DFT size, you can simply concatenate the waveforms from above and that could recreate the original input (if you didn't use a window). However since the waveforms in the analysis frames are likely to overlap (which happens when the hop size is smaller than the DFT size), you will need to use an overlap-add procedure. Generate an output array which is as long as the desired output sound and set all its elements to zero. Each time you obtain a waveform frame by applying the inverse DFT on a spectrum from step 1, you will need to add the result at the indices from which the original frame input came. This will effectively superimpose parts of frames that overlap and thus not throw away any information.
- Finally you will need to add the option of a synthesis window. Some of the operations that we will be performing will result in significant changes in the time domain and might create some discontinuities at the ends of the outputs which will result in audible clicks. A good way to ensure that these artifacts go away is to use a synthesis window. This will be a function defined as before, but we will be applying it on the time-domain output of the inverse DFT.

Using the same sounds as above, verify that when you perform a forward transform and then take its inverse, that you get an output that sounds like the original (there might be minor numerical differences, you can ignore these). Try to get the resynthesized output to be as close to the input as possible, when using various settings.

Note that you can't always get perfect reconstruction depending on the parameters you choose. The hop size needs to be equal or smaller than the DFT size otherwise you will lose information (some samples won't be transformed). When you use a window, you also cause some information to be lost. In the case of the Hann window you should have an overlap of 1/2, 1/4, 1/8, etc. of the DFT size. If not you will get an unintended amplitude modulation. Likewise, if you use a Hann synthesis window, the hop size needs to be at most as large as 1/4 of the DFT size (try this with 1/2 the size and attempt to explain why this is a bad idea).

I suggest you start with no windowing, then use a Hand window for both synthesis and analysis. Make sure that the hop size and windows are such that the COLA principle holds (otherwise you won't get perfect reconstruction). To ensure that the COLA principle holds you need to make sure that when you add windows which are offset by the hop size they sum to a contant value. If you also use a synthesis window in addition to an analysis window, then you need to make sure that the square of these windows sums to one (because you effectively apply them twice, once for the forward transform, and once for the resynthesis). A sneaky way to guarantee that is to select a window, and square root its components before you apply it. Is the window is COLA, then this operation will guarantee that if you apply it for both analysis and synthesis it will still result in a constant modulation.

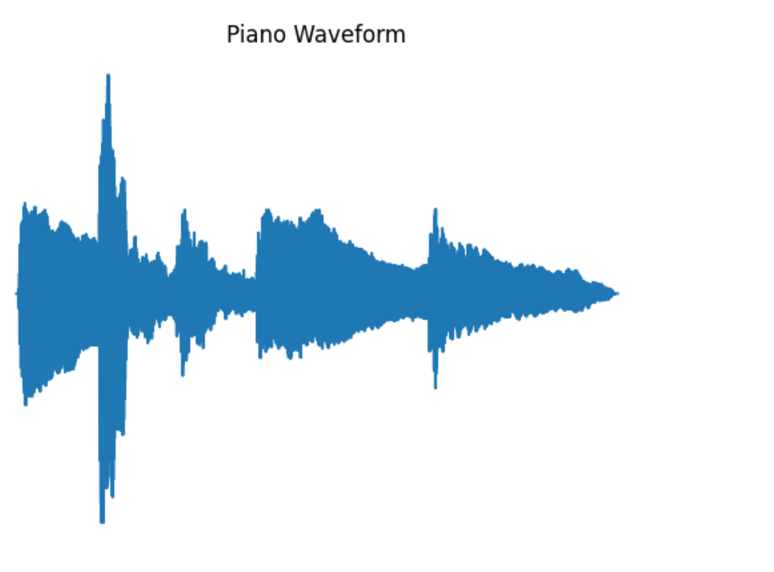
In [ ]:
```python
def plot_waveform(waveform, title):
    plt.plot(waveform)
    plt.title(title)

    ax = plt.gca()
    ax.xaxis.set_tick_params(length=0,labelbottom=False)
    ax.yaxis.set_tick_params(length=0,labelbottom=False)
    ax.axes.xaxis.set_ticklabels([])
    ax.axes.yaxis.set_ticklabels([])
    ax.spines['top'].set_visible(False)
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.spines['bottom'].set_visible(False)
    ax.spines['left'].set_visible(False)

    plt.show()
```

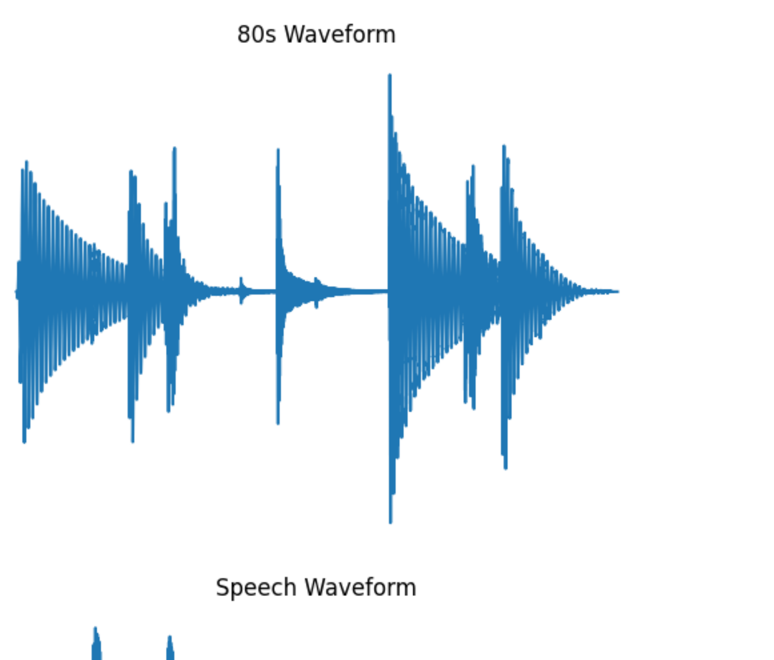
In [ ]:
```python
def istft(stft_output, dft_size, hop_size, zero_pad, window):
    # Initializing the signal length
    signal_length = (stft_output.shape[0] * hop_size) + dft_size + zero_pad
    signal = np.zeros(signal_length)

    for i in range(stft_output.shape[0]):
        original_signal = np.fft.irfft(stft_output[i, :], dft_size + zero_pad)
        start = i * hop_size
        end = start + original_signal.shape[0]
        signal[start:end] += original_signal

    return signal


# Invert all of the spectrograms from the previous assignment
inverse_piano = istft(stft_piano, dft_size, hop_size, zero_pad, window)
inverse_80s = istft(stft_80s, dft_size, hop_size, zero_pad, window)
inverse_speech = istft(stft_speech, dft_size, hop_size, zero_pad, window)

# Play the sounds to make sure they are correct (look out for unwanted clicks, wobbles, etc.)
sound(inverse_piano, rate=fs_piano, label='piano inverse')
sound(inverse_80s, rate=fs_80s, label='80s inverse')
sound(inverse_speech, rate=fs_speech, label='speech inverse')

# Plot the waveforms
plot_waveform(inverse_piano, "Piano Waveform")
plot_waveform(inverse_80s, "80s Waveform")
plot_waveform(inverse_speech, "Speech Waveform")
```

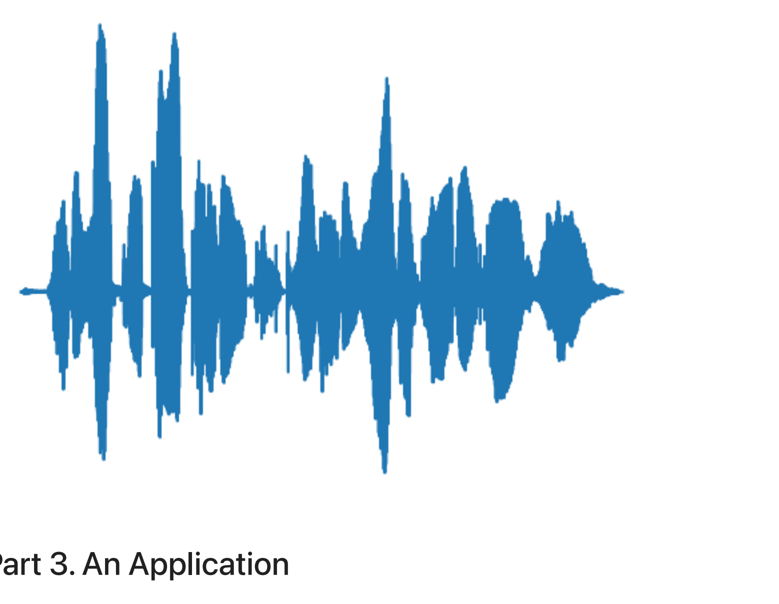
piano inverse -0:03

80s inverse -0:02

speech inverse -0:03







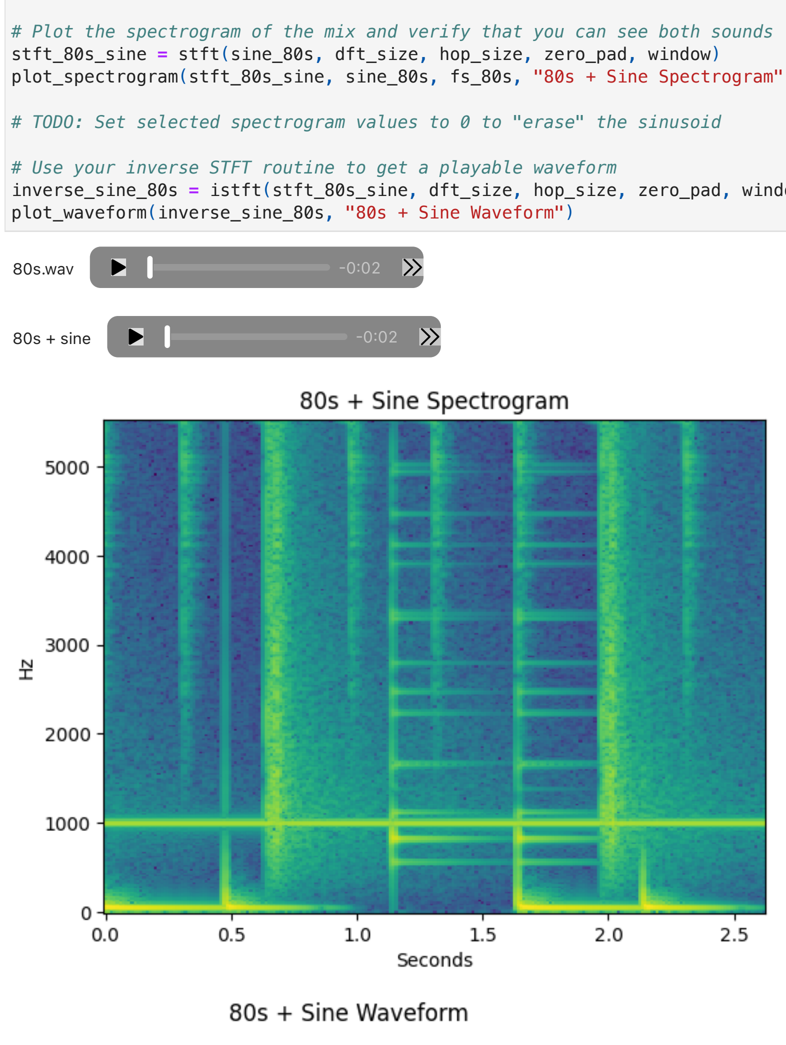
Part 3. An Application

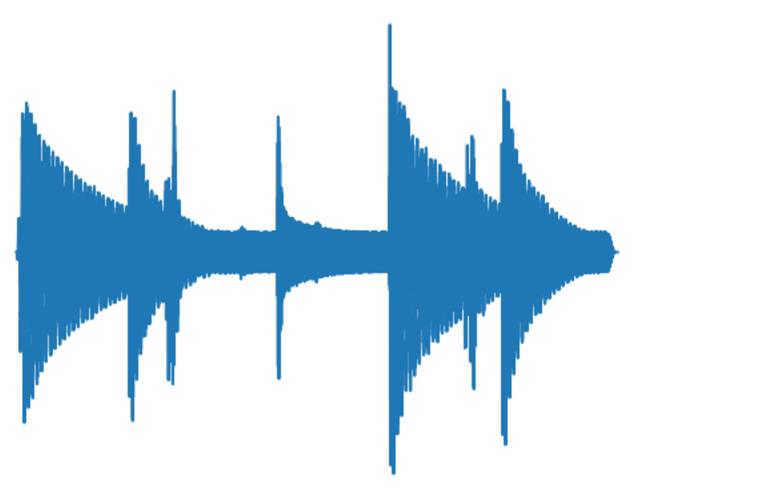
Just so you get an idea of how one might use these tools here is a simple example. Take one of the test sounds above and add to it a constant sinusoid with a frequency of 1kHz. When you plot the spectrogram of that sound you should be able to see the sinusoid. Using your code take the spectrogram matrix and set its values that correspond to the sinusoid to zero. Put that back to the inverse stft function and you should get a denoised version of the signal. FYI, this is not a textbook way to solve this problem (it's a little hacky), we'll cover the right way later.

In [ ]:
```python
# Load one sound and add to it a 1kHz sinusoid of the same length
fs_80s, input_sound_80s = wavfile.read("./data/80s.wav")
sound(input_sound_80s, rate=fs_80s, label='80s.wav')

sine_frequency = 1000
sine_duration = len(input_sound_80s) / fs_80s
sine_amplitude = 0.08

t = np.linspace(0, sine_duration, int(sine_duration * fs_80s), False)
sine_wave = sine_amplitude * np.sin(2 * np.pi * sine_frequency * t)
sine_wave = (sine_wave * (2**15 - 1)).astype(np.int16)

sine_80s = input_sound_80s + sine_wave
sound(sine_80s, rate=fs_80s, label='80s + sine')

# Plot the spectrogram of the mix and verify that you can see both sounds
stft_80s_sine = stft(sine_80s, dft_size, hop_size, zero_pad, window)
plot_spectrogram(stft_80s_sine, sine_80s, fs_80s, "80s + Sine Spectrogram")

# TODO: Set selected spectrogram values to 0 to "erase" the sinusoid

# Use your inverse STFT routine to get a playable waveform
inverse_sine_80s = istft(stft_80s_sine, dft_size, hop_size, zero_pad, window)
plot_waveform(inverse_sine_80s, "80s + Sine Waveform")
```

80s.wav -0:02

80s + sine -0:02





Part 4. (optional): Just for fun

Take a picture of yourself and load it in python. Pretend it is a spectrogram and put it into your inverse STFT function to get it's correspoding waveform. It'll likely sound horrible, but hey that was fun.