Gábor Transorms (Short-Time Fourier Transform)

Brandon Goldney[1]

[1]*Department of Applied Mathematics, University of Washington, Email: Goldney@uw.edu*

Abstract

The Gábor Filter shows the frequency of a signal in the time domain. This is an advancement from the Fourier Transform which shows a signal's amplitude in the frequency domain. The purpose of this paper is to demonstrate the advantages and limitations of the Gábor Filter. An 8.9 second audio clip of Handel's "Messiah" is analyzed to determine the frequencies present in the time domain. Subsequently, a rendering of the children's classic song "Mary Had a Little Lamb" is passed through a Gábor Filter to assess the frequencies present.

1 Section I: Introduction and Overview

Heisenberg's Uncertainty Principle asserts that the position and velocity of an object cannot be measured exactly, at the same time, even in theory. One example of this is the Fourier Transform, which decomposes a function into its constituent frequencies. Fourier Series are powerful because they are able to represent a function as a series of sin and cos waves. However, the cost of performing a Fourier Transform is that all time information is discarded. In other words, the signal which was originally in the time domain will now be represented in the frequency domain.

A Fourier Transform is very good at determining the constituent frequencies of a signal, but that signal must be stationary (i.e. the frequencies are constant throughout time). Consider a song where the frequencies change throughout time. A Fourier Transform will fail to ascertain the frequencies used in a non-stationary signal. Gábor Wavelets are designed to overcome this challenge. Wavelets are effectively windows dragged over the signal, at each iteration a Fourier Transform is performed on the windowed section of the signal. Sliding the window across the signal highlights the Heisenberg Uncertainty Principle. The width of the window can be narrow or wide, based off the desired accuracy. A wider window will capture more frequencies, but not localize the time well.

This leads us to the concepts of oversampling and undersampling. Oversampling will occur when the window is too narrow. The downside to oversampling is the Gábor Transformation will be computationally expensive, and can be prohibitively time consuming for real-world applications. Undersampling occurs when the window is too wide. In this scenario, the Fourier Transform is performed over a wide range of frequencies. The multitude of frequencies will cause the Gabor Transform to become blurry because too many frequencies are being represented.

2 Section II: Theoretical Background

In order to localize with respect to both time and frequency Gabor introduced a kernel:

$$g_{t,\omega}(\tau) = e^{i\omega\tau} g(\tau - t)$$

Incorporating Gabor's kernel with the Fourier Transform results in the following:

$$G[f](t,\omega) = \tilde{f}_g(t,\omega) = \int_{-\infty}^{\infty} f(\tau)\bar{g}(\tau - t)e^{i\omega\tau}d\tau = (f, \bar{g}_{t,\omega})$$

The bar denotes the complex conjugate. While this formula appears intimidating at first, it can be broken down into its components to better understand it. Notice, we're integrating over the term τ, which effectively slides the window over the signal. The new term, $\bar{g}(\tau - t)$ was conceptually explained earlier, it localizes the signal with respect to time and frequency. The final term, $e^{-i\omega\tau}$, is the standard Fourier Transform.

The Gábor transform is computed by discretizing the time and frequency domain. Consider the following sample points:

$$v = m\omega_0 \tau = nt_0$$

where m and n are integers and $w_0, \tau_0 > 0$ are constants. In this case, the discrete version of Gábor's kernel becomes:

$$g_{m,n}(t) = e^{i2\pi m\omega_0 t} g(t - nt_0)$$

Accordingly, the Gábor Transform becomes:

$$\tilde{f}(m,n) = \int_{-\infty}^{\infty} f(t)\bar{g}_{m,n}(t)dt = (f, g_{m,n})$$

3 Section III: Algorithm Implementation and Development

Since the process is very similar for both Handel's "Messiah" and "Mary Had a Little Lamb", the process below is generalized to be accurate to both methods.

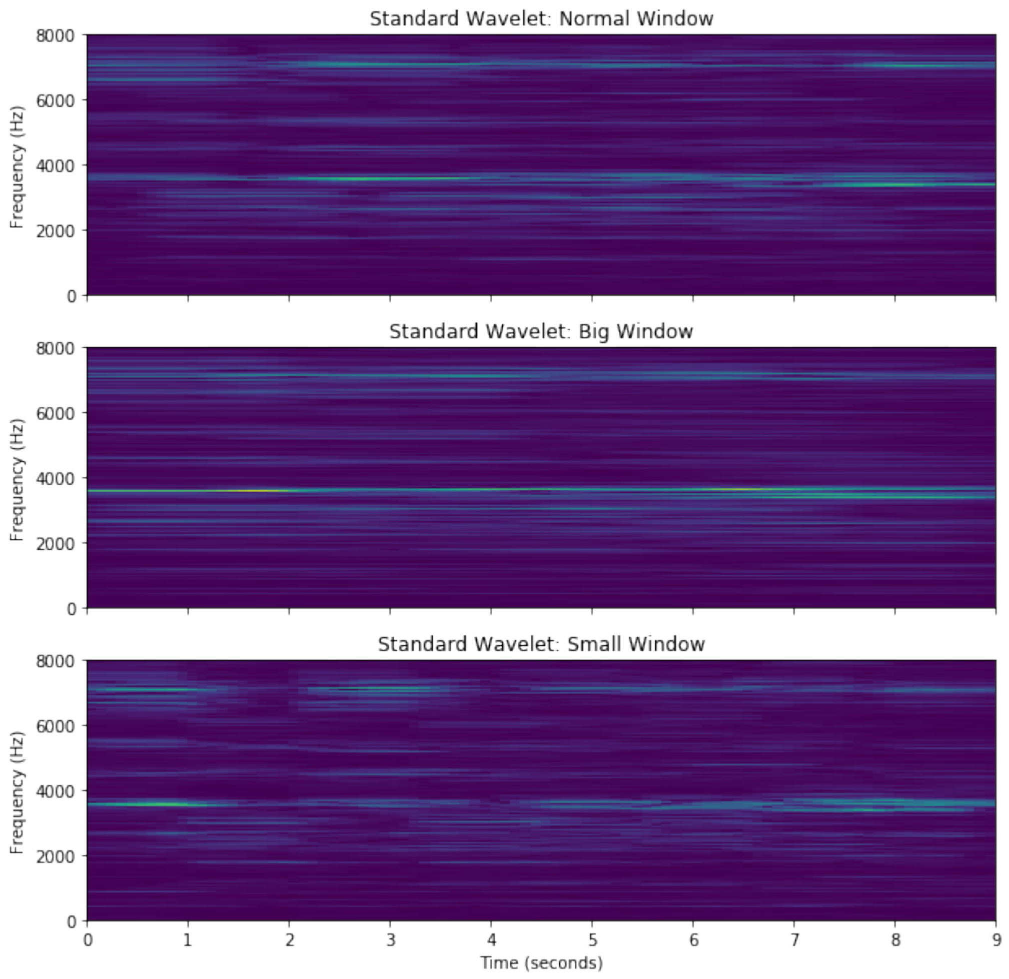
After loading the data, the first step is to define the length of the data and then to discretize it into n data points. Next, we rescale the n data points by $(2 * \pi)/L$, where L equals the duration of the song. L is calculated by dividing the number of data points by the sampling frequency. We know in later steps that we will be performing a Fourier Transform, which rearranges X by shifting the zero-frequency component to the center of the array. With that in mind, we shift the n data points (i.e. use np.fft.fftshift) prior to any further steps.

Now that we have the initial variables declared and created the discretized points , we can begin working with the data from the sound file by setting up a for loop. Within the for loop, a window (e.g. Gaussian, Shannon, etc.) is created. It's important to recall that setting up the window width (i.e. *t* - *tslide*) is important because that will impact the trade-off in accuracy between time and frequency. Each time the data passes through the window,the signal is multiplied by the window. The Fourier Transform is taken on that new data, and then it is shifted (i.e. np.fft.fftshift) because the FFT algorithim shifts the data, and finally it's saved. The data is saved so the spectrogram can be constructed. To create the graphs, we plot the *time* variable on the x-axis and the wavenumbers (rescaled to be $2 * \pi$ periodic) are shifted and put on the y-axis.

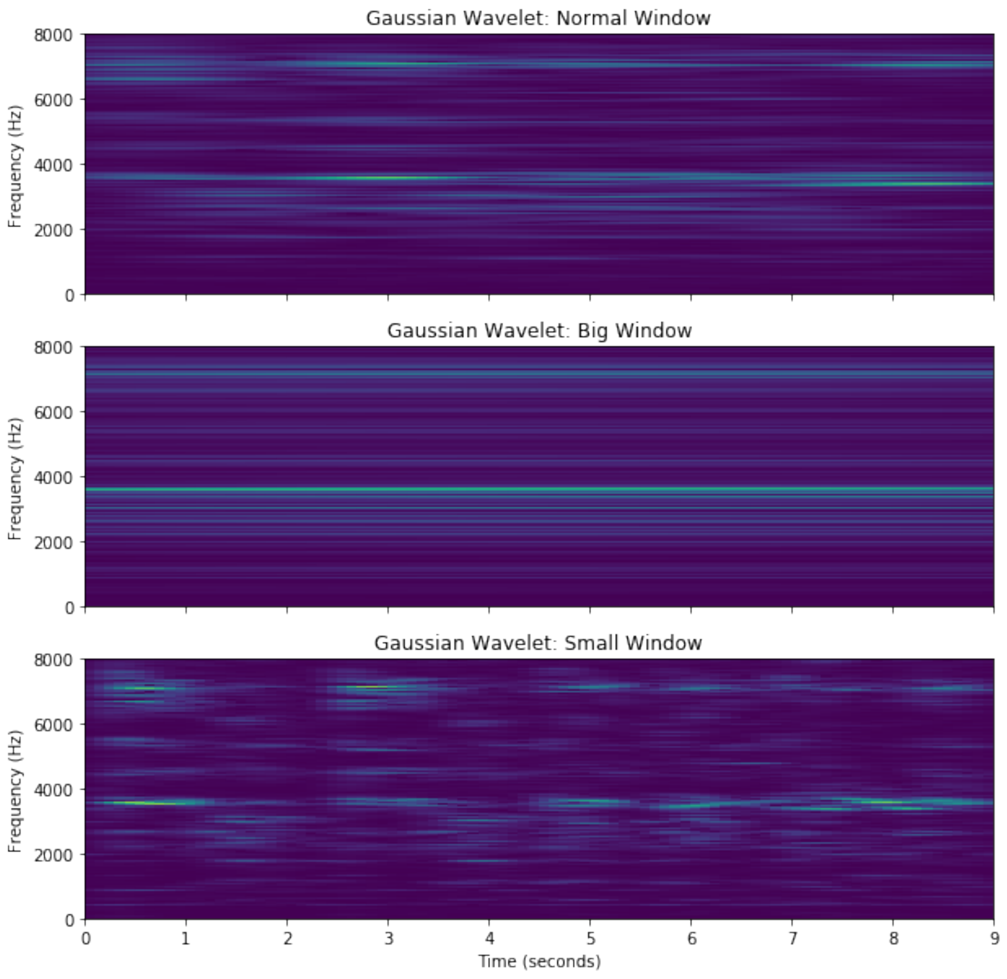
4 Section IV: Computational Results

This section lists each type of wavelet applied to the data, and three spectrograms for each wavelet. Each spectrogram shows a different window width (i.e. wide, narrow, and standard).

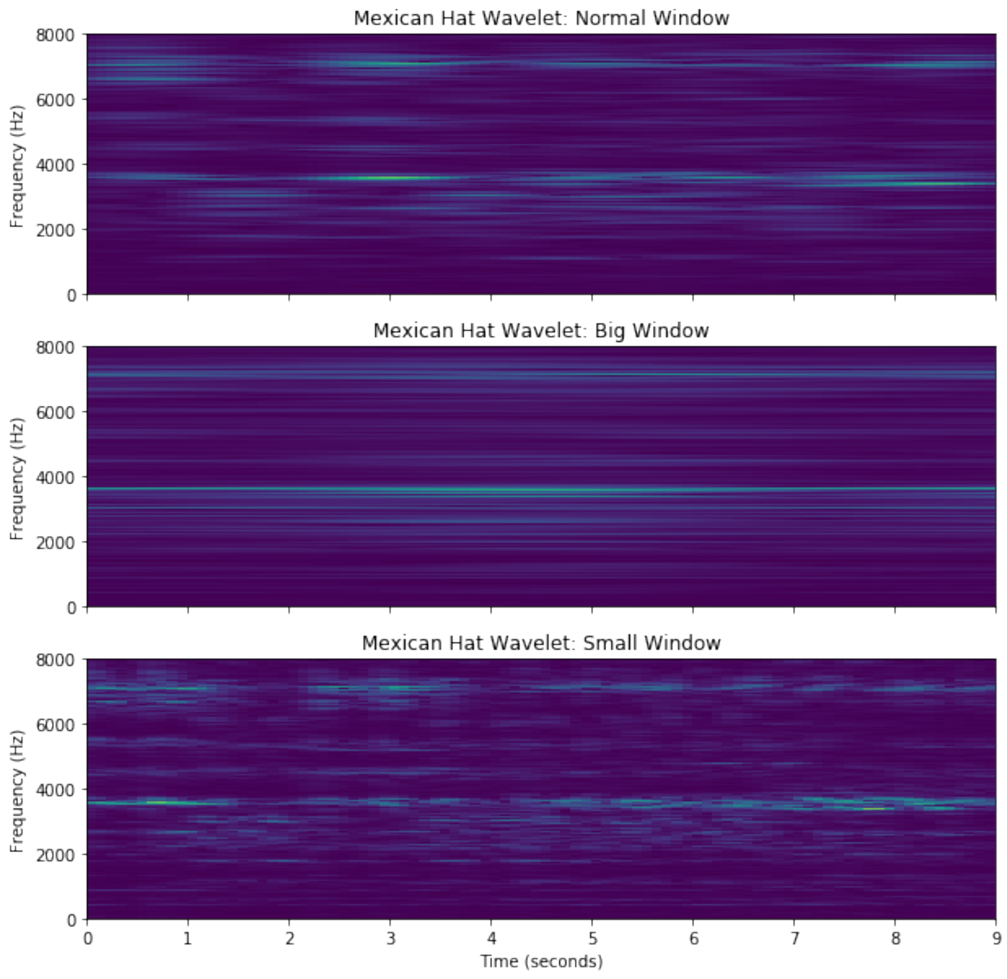
$$\text{Standard Wavelet} = \begin{cases} \text{normal window} & = e^{-1*(t-\tau)^{10})} \\ \text{big window} & = e^{-0.01*(t-\tau)^{10})} \\ \text{small window} & = e^{-100*(t-\tau)^{10})} \end{cases} \tag{1}$$



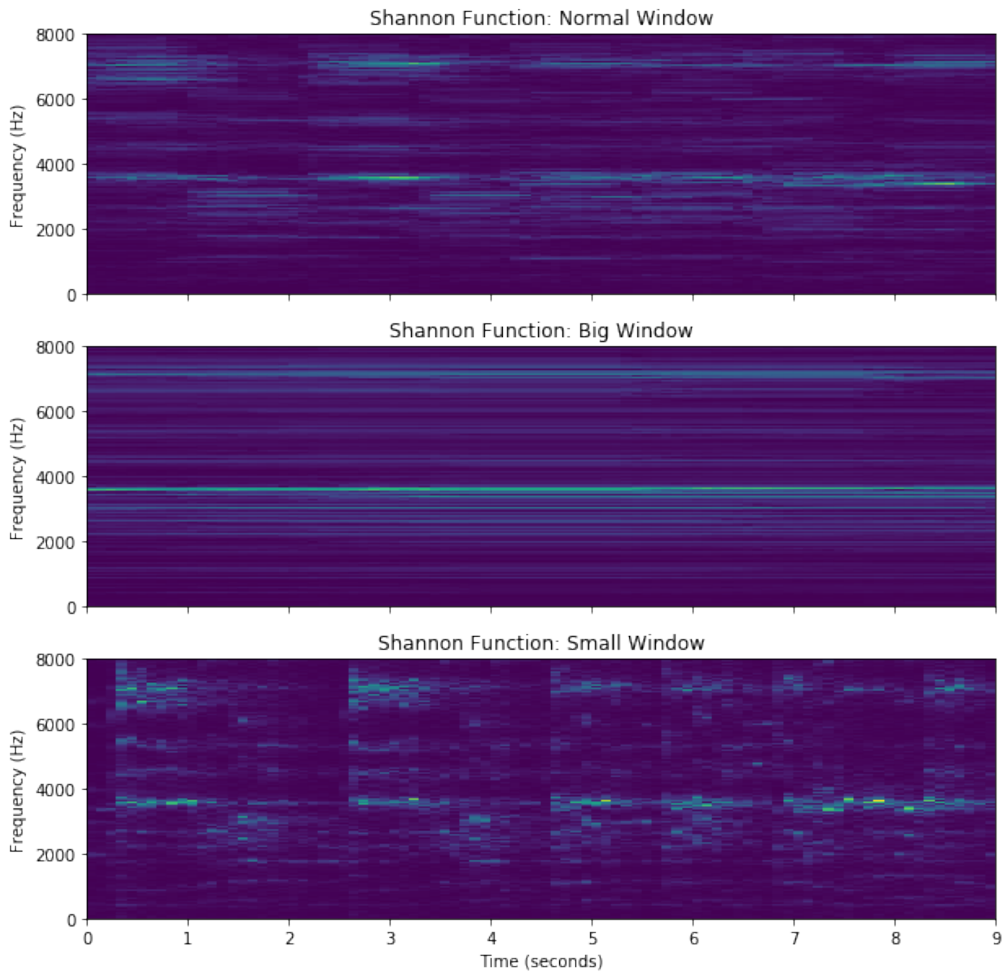
$$\text{Gaussian Wavelet} = \begin{cases} \text{normal window} & = e^{-1*(t-\tau)^{2})} \\ \text{big window} & = e^{-0.01*(t-\tau)^{2})} \\ \text{small window} & = e^{-10*(t-\tau)^{2})} \end{cases} \tag{2}$$



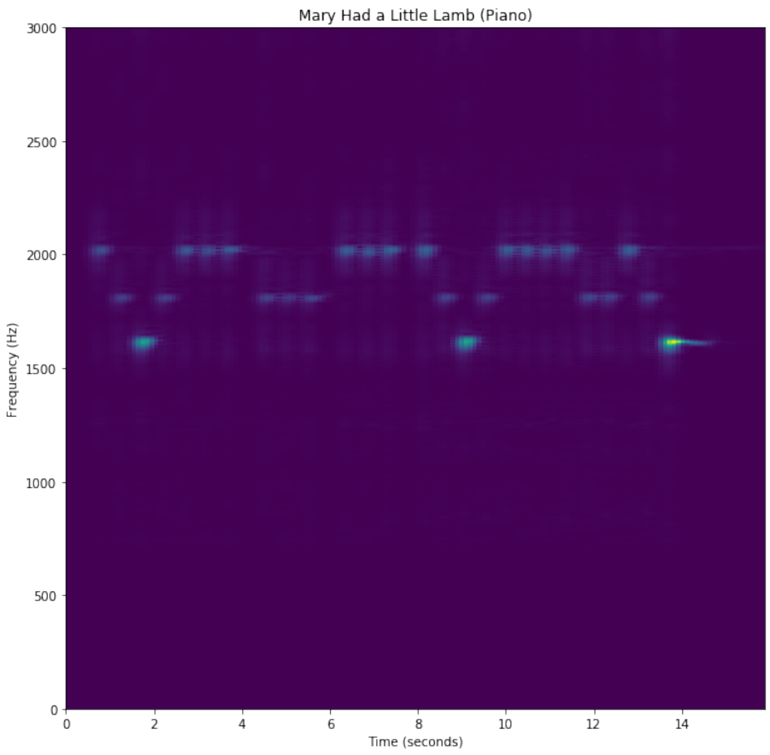
$$\text{Mexican Hat Wavelet} = \begin{cases} \text{normal window} & = (1 - (t-\tau)^2 * e^{-1*(t-\tau)^2} \\ \text{big window} & = ((1-0.1)*(t-\tau)^2)*e^{-0.1*(t-\tau)^2} \\ \text{small window} & = ((1-10)*(t-\tau)^2)*e^{-10*(t-\tau)^2} \end{cases} \tag{3}$$

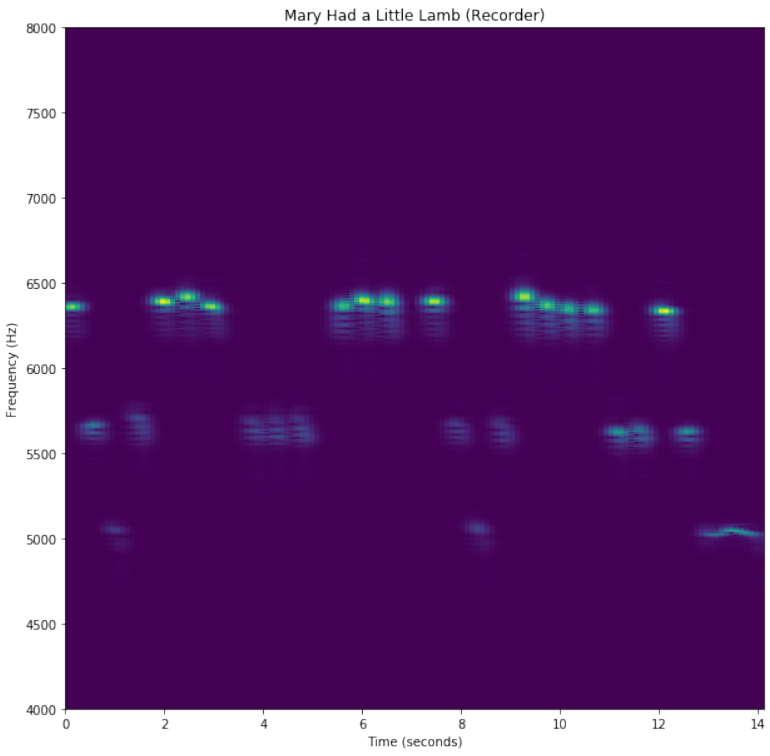


$$\text{Shannon Function Wavelet} = \begin{cases} \text{normal window} & = |t - \tau| \leq 0.5 \\ \text{big window} & = 0.1 * |t - \tau| \leq 0.5 \\ \text{small window} & = 10 * |t - \tau| \leq 0.5 \end{cases} \tag{4}$$



Part 2: Mary Had a Little Lamb





5 Section V: Summary and Conclusions

Recall, as we increase the width of the window, we will capture a broader range of frequencies; however, we will not be able to localize the frequencies with respect to time. The reverse is true for narrow windows. That being said, in graphs with larger windows, we can trust the y-axis (i.e. frequency) more, and we lose faith in the x-axis (i.e time).

Graphically, we can see this in the spectrograms. Graphs with wider windows tend to blur more in the horizontal direction, since larger windows do not localize well. The Gaussian Wavelet, Mexican Hat, and Shannon Function clearly demonstrate the difference between large and small window sizes. In those functions, we can see dark spaces between the color on the x-axis. Presumably this is where there are slight pauses in music. In Handel's Messiah, we can clearly see a higher and lower frequency, presumably the difference between men and women's singing.

Mary Had a Little Lamb Creating spectrograms for a piano and a recorder creates an interesting plot. Pianos create overtones; whereas, recorders create single frequencies. We can see more color in the y-axis direction for the piano than the recorder.

6 Appendix A: Python functions used and brief implementation explanation

- **np.load**: Load arrays or pickled objects
- **sd.play**: Assuming you have a NumPy array named myarray holding audio data with a sampling frequency of fs (in the most cases this will be 44100 or 48000 frames per second), you can play it back with play()
- **IPython.display import Image**: Loads images from local source
- **Matplotlib**: Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms
- **np.linspace**: Return evenly spaced numbers over a specified interval
- **np.fft.fftshift**: Shift the zero-frequency component to the center of the spectrum
- **np.fft.ffti**: Compute the one-dimensional discrete Fourier Transform
- **np.argmax**: Returns the indices of the maximum values along an axis
- **np.colormesh**: Create a pseudocolor plot with a non-regular rectangular grid (good for creating spectrograms)

7 Appendix B: Python code

HW_2_Code_Only vF

March 7, 2020

Github
Link: https://github.com/b-goldney/AMATH_582
Username: b-goldney

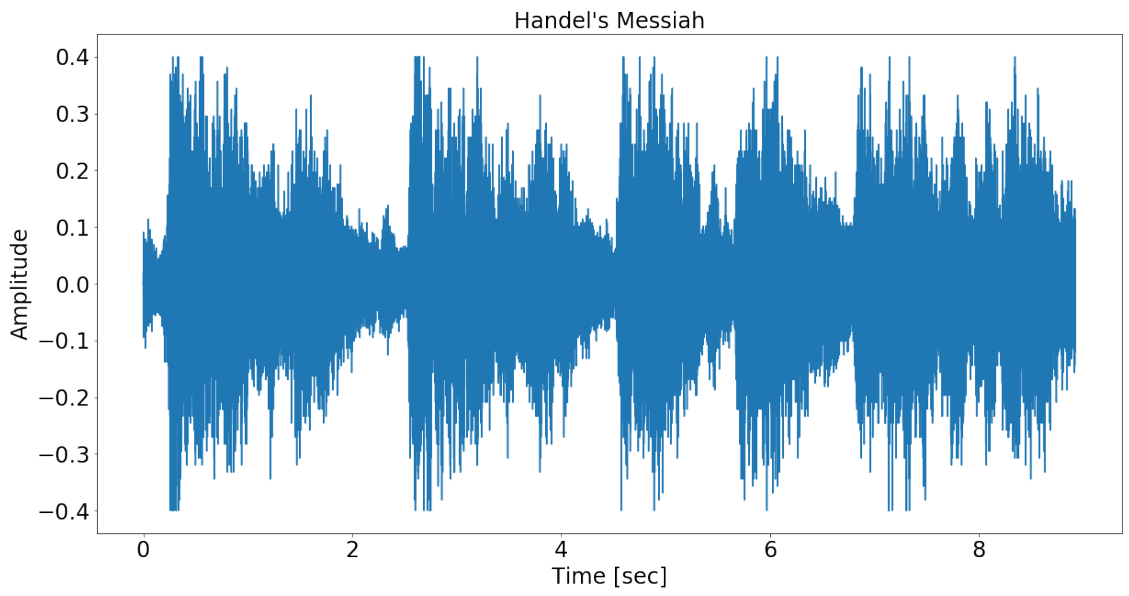
Appendix B: Python Code

Part I: Handel's Messiah

[2]:
```
import sounddevice as sd
import numpy as np
import matplotlib. pyplot as plt
from scipy import signal
#from scipy.fft import fftshift
import matplotlib.pyplot as plt

y = np.load('y.npy')
Fs = np.load('Fs.npy') # The elements of a 0d array can be accessed via: Fs[()]
sd.play(y,Fs)
```

Note, on the above:
- Fs is sampling frequency. The sampling frequency (or sample rate) is the number of samples per second in a Sound. For example: if the sampling frequency is 44100 hertz, a recording with a duration of 60 seconds will contain 2,646,000 samples. - y is an array with 73,113 elements. It's 8.9 seconds long (73,113 samples / 8192 samples per second = 8.924 seconds)

[3]:
```
# Incorporate given code from the homework prompt
v = y/2; # v is simply the original amplitude divided by 2

plt.figure(figsize=(20,10)), plt.tick_params(axis='both', which='major',␣
 ↪labelsize=24)
plt.xlabel('Time [sec]', fontsize=24), plt.ylabel('Amplitude',fontsize=24)
plt.title('Handel\'s Messiah', fontsize=24)

plt.plot(range(0, len(v))/Fs,v);
```



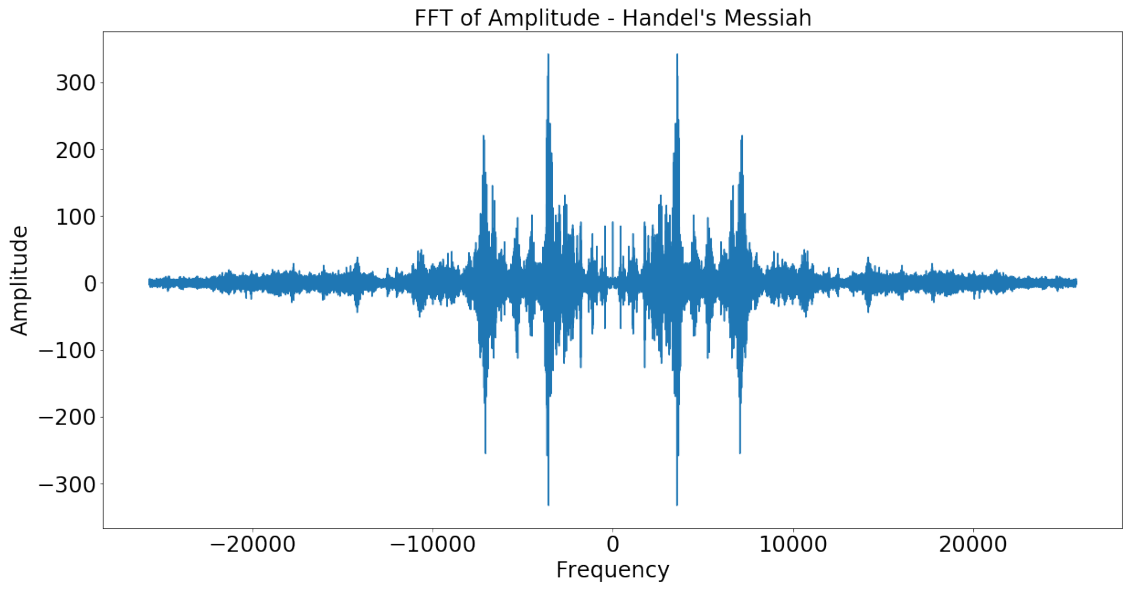
[4]:
```
L = len(y)/Fs
n = len(v) # v is simply the original amplitude divided by 2
t2 = range(0,n+1)/ Fs # this is length of amplitude divided by 8,192 (sampling␣
 ↪frequency).
t = t2[0:n+1]

# Create k variable
list1 = np.linspace(0,n/2-1,num=(n/2-1*.0000001)) # n is length of amplitude
list2 = np.linspace(-n/2,-1,num=(n/2-1*.0000001))
list3 = []

for i in list1:
    list3.append(i)

for i in list2:
    list3.append(i)

# Convert list3 to a numpy array so it can be multiplied
array3 = np.asarray(list3)

# Rescale wavenumbers because FFT assumes 2*pi periodic signals
k = (2*np.pi / L)* array3
```

Note on the above: - variable t2 is the length of amplitude divided by 8,192 (i.e. sampling frequency). The purpose of this is to create frame size. Frame size is the total time (T) to acquire one block of data. - The block size (N) is the total number of time data points that are captured to perform a Fourier transform. A block size of 2000 means that two thousand data points are acquired, then a Fourier transform is performed. - For example, with a block size of 2000 data points and a sampling rate of 1000 samples per second, the total time to acquire a single data block is 2 seconds. It takes two seconds to collect 2000 data points. - Source: Digital Signal Processing: Sampling Rates, Bandwidth, Spectral Lines, and more... - In other words, t2 tells us how much time is required to obtain the ith element of time in t2

[5]:
```
%matplotlib inline

# Take FFT of data and and fftshift to create plot of the entire song
ks = np.fft.fftshift(k)
vt = np.fft.fft(v[0:-1]) # v is the amplitude divided by 2

plt.figure(figsize=(20,10)), plt.tick_params(axis='both', which='major',␣
 ↪labelsize=24)
plt.xlabel('Frequency', fontsize=24), plt.ylabel('Amplitude',fontsize=24)
plt.title('FFT of Amplitude - Handel\'s Messiah', fontsize=24)

plt.plot(ks,np.fft.fftshift(vt[0:73112]))
```

```
/opt/anaconda3/lib/python3.7/site-packages/numpy/core/_asarray.py:85:
ComplexWarning: Casting complex values to real discards the imaginary part
  return array(a, dtype, copy=False, order=order)
```

[5]:
```
[<matplotlib.lines.Line2D at 0x121bdc890>]
```



Set up the Gabor Filter

[6]:
```
tslide = np.linspace(0,9,91)

window_spc = []
```

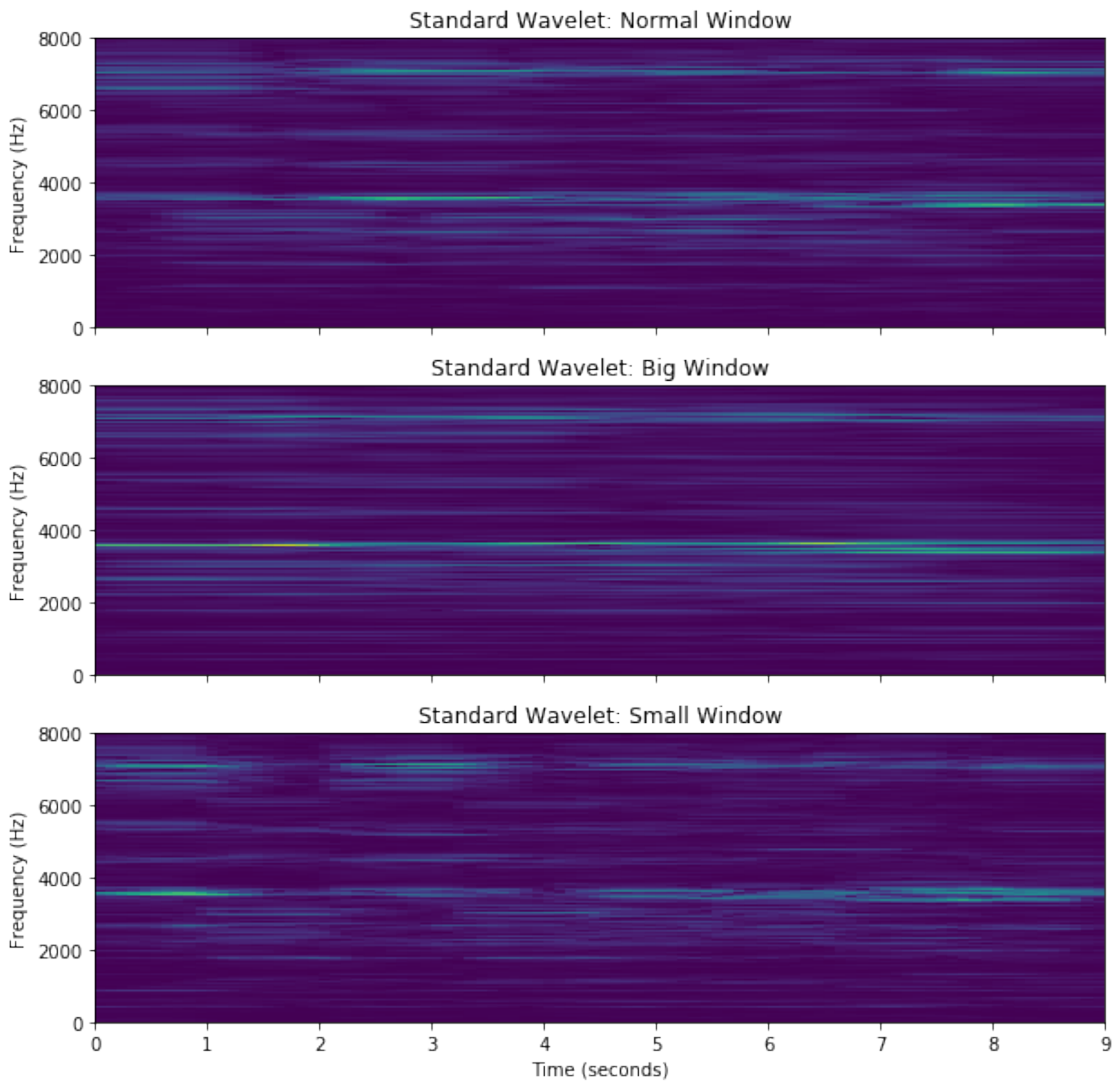
```
big_window_spc = []
small_window_spc = []

gauss_spc = []
gauss_big_spc = []
gauss_small_spc = []

mexican_hat_spc = []
mexican_big_spc = []
mexican_small_spc = []

shannon_spc = []
shannon_big_spc = []
shannon_small_spc = []

for i in range(len(tslide)):
    window = np.exp(-1*(t-tslide[i])**10) # t is the length of amplitude␣
 ↪divided by 8,192 (sampling frequency)
    big_window = np.exp(-0.01*(t-tslide[i])**10)
    small_window = np.exp(-100*(t-tslide[i])**10)
    gauss_window = np.exp(-(t-tslide[i])**2)
    gauss_big = np.exp(-0.01*(t-tslide[i])**2)
    gauss_small = np.exp(-10*(t-tslide[i])**2)
    mexican_hat = (1-(t-tslide[i])**2)*np.exp(-(t-tslide[i])**2)
    mexican_hat_big = ((1-0.1)*(t-tslide[i])**2)*np.exp(-0.1*(t-tslide[i])**2)
    mexican_hat_small = ((1-10)*(t-tslide[i])**2)*np.exp(-10*(t-tslide[i])**2)
    shannon_fcn = abs(t-tslide[i]) <= 0.5
    shannon_fcn_big = 0.1 * abs(t-tslide[i]) <= 0.5
    shannon_fcn_small = 10 * abs(t-tslide[i]) <= 0.5

    window_vf = window[0:-1] * v # v is simply the original amplitude divided␣
 ↪by 2
    window_vft = np.fft.fft(window_vf)
    window_spc.append(abs(np.fft.fftshift(window_vft[0:-2])))

    big_window_vf = big_window[0:-1] * v # v is simply the original amplitude␣
 ↪divided by 2
    big_window_vft = np.fft.fft(big_window_vf)
    big_window_spc.append(abs(np.fft.fftshift(big_window_vft[0:-2])))

    small_window_vf = small_window[0:-1] * v # v is simply the original␣
 ↪amplitude divided by 2
    small_window_vft = np.fft.fft(small_window_vf)
    small_window_spc.append(abs(np.fft.fftshift(small_window_vft[0:-2])))

    gauss_window_vf = gauss_window[0:-1] * v # v is simply the original␣
 ↪amplitude divided by 2
```

```
    gauss_window_vft = np.fft.fft(gauss_window_vf)
    gauss_spc.append(abs(np.fft.fftshift(gauss_window_vft[0:-2])))

    gauss_big_vf = gauss_big[0:-1] * v # v is simply the original amplitude␣
↪divided by 2
    gauss_big_vft = np.fft.fft(gauss_big_vf)
    gauss_big_spc.append(abs(np.fft.fftshift(gauss_big_vft[0:-2])))

    gauss_small_vf = gauss_small[0:-1] * v # v is simply the original amplitude␣
↪divided by 2
    gauss_small_vft = np.fft.fft(gauss_small_vf)
    gauss_small_spc.append(abs(np.fft.fftshift(gauss_small_vft[0:-2])))

    mexican_vf = mexican_hat[0:-1] * v # v is simply the original amplitude␣
↪divided by 2
    mexican_vft = np.fft.fft(mexican_vf)
    mexican_hat_spc.append(abs(np.fft.fftshift(mexican_vft[0:-2])))

    mexican_big_vf = mexican_hat_big[0:-1] * v # v is simply the original␣
↪amplitude divided by 2
    mexican_big_vft = np.fft.fft(mexican_big_vf)
    mexican_big_spc.append(abs(np.fft.fftshift(mexican_big_vft[0:-2])))

    mexican_small_vf = mexican_hat_small[0:-1] * v # v is simply the original␣
↪amplitude divided by 2
    mexican_small_vft = np.fft.fft(mexican_small_vf)
    mexican_small_spc.append(abs(np.fft.fftshift(mexican_small_vft[0:-2])))

    shannon_vf = shannon_fcn[0:-1] * v # v is simply the original amplitude␣
↪divided by 2
    shannon_vft = np.fft.fft(shannon_vf)
    shannon_spc.append(abs(np.fft.fftshift(shannon_vft[0:-2])))

    shannon_big_vf = shannon_fcn_big[0:-1] * v # v is simply the original␣
↪amplitude divided by 2
    shannon_big_vft = np.fft.fft(shannon_big_vf)
    shannon_big_spc.append(abs(np.fft.fftshift(shannon_big_vft[0:-2])))

    shannon_small_vf = shannon_fcn_small[0:-1] * v # v is simply the original␣
↪amplitude divided by 2
    shannon_small_vft = np.fft.fft(shannon_small_vf)
    shannon_small_spc.append(abs(np.fft.fftshift(shannon_small_vft[0:-2])))
```

note, in order for the graphs below to work, the "c" value argument to pcolormesh needs to be transposed for some reason

[14]:
```
fig, (ax1,ax2,ax3) = plt.subplots(nrows=3, ncols=1, figsize=(10,10),␣
 ↪sharex=True, sharey=True)

# Create graphs
ax1.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(window_spc).
 ↪transpose())
ax2.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(big_window_spc).
 ↪transpose())
ax3.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(small_window_spc).
 ↪transpose())

# Set titles
ax1.title.set_text('Standard Wavelet: Normal Window')
ax2.title.set_text('Standard Wavelet: Big Window')
ax3.title.set_text('Standard Wavelet: Small Window')

# Note, I intentionally did not label each x-axis because it makes everything␣
 ↪too cluttered
ax3.set_xlabel('Time (seconds)')

ax1.set_ylabel('Frequency (Hz)')
ax2.set_ylabel('Frequency (Hz)')
ax3.set_ylabel('Frequency (Hz)')

# Set y-axis limits
ax1.set_ylim([0,8000])
```

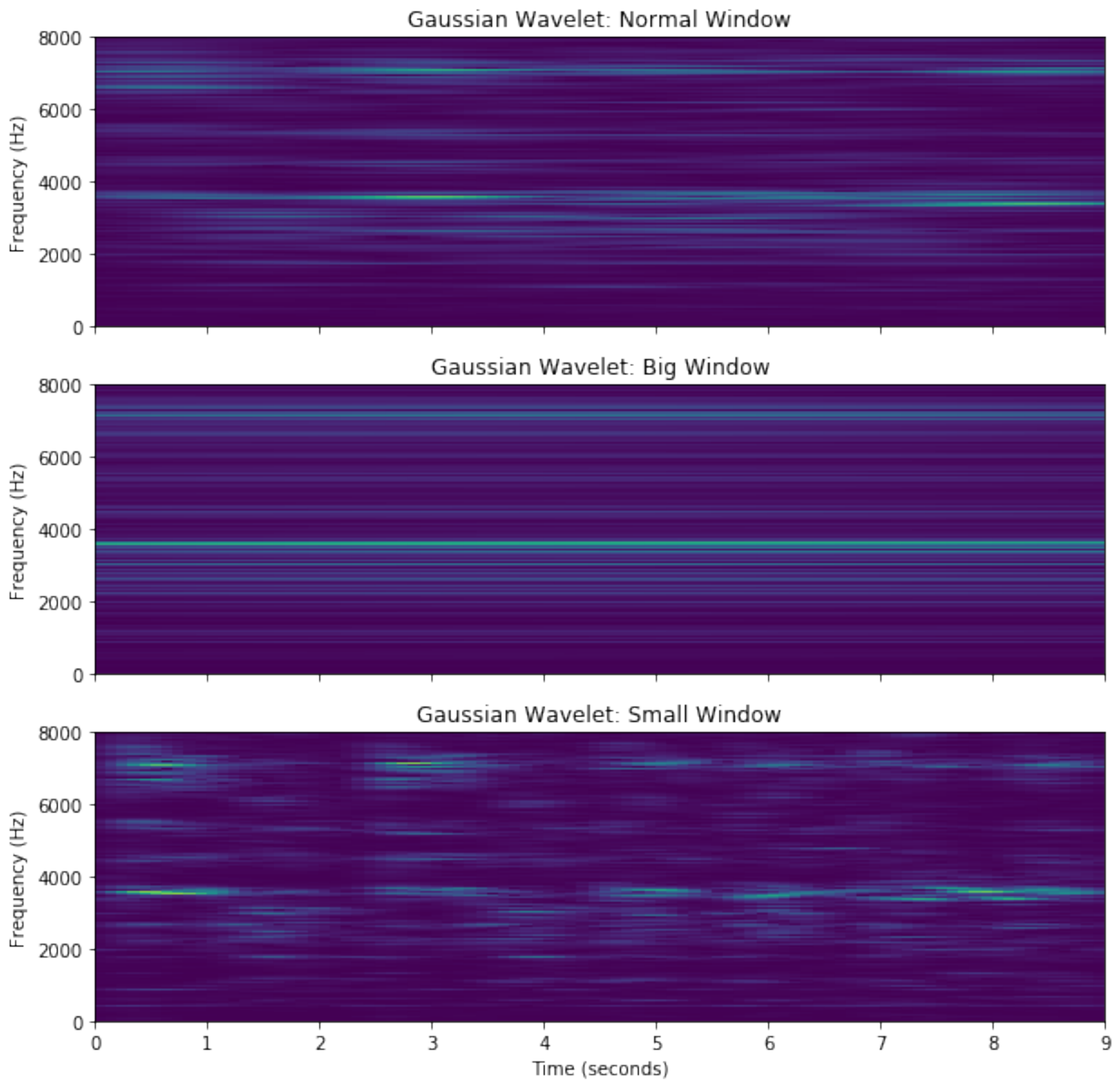
[14]:
```
(0, 8000)
```



[42]:
```
fig, (ax1,ax2,ax3) = plt.subplots(nrows=3, ncols=1, figsize=(10,10),␣
↪sharex=True, sharey=True)

# Create graphs
ax1.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(gauss_spc).transpose())
ax2.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(gauss_big_spc).
↪transpose())
ax3.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(gauss_small_spc).
↪transpose())

# Set titles
ax1.title.set_text('Gaussian Wavelet: Normal Window')
ax2.title.set_text('Gaussian Wavelet: Big Window')
```

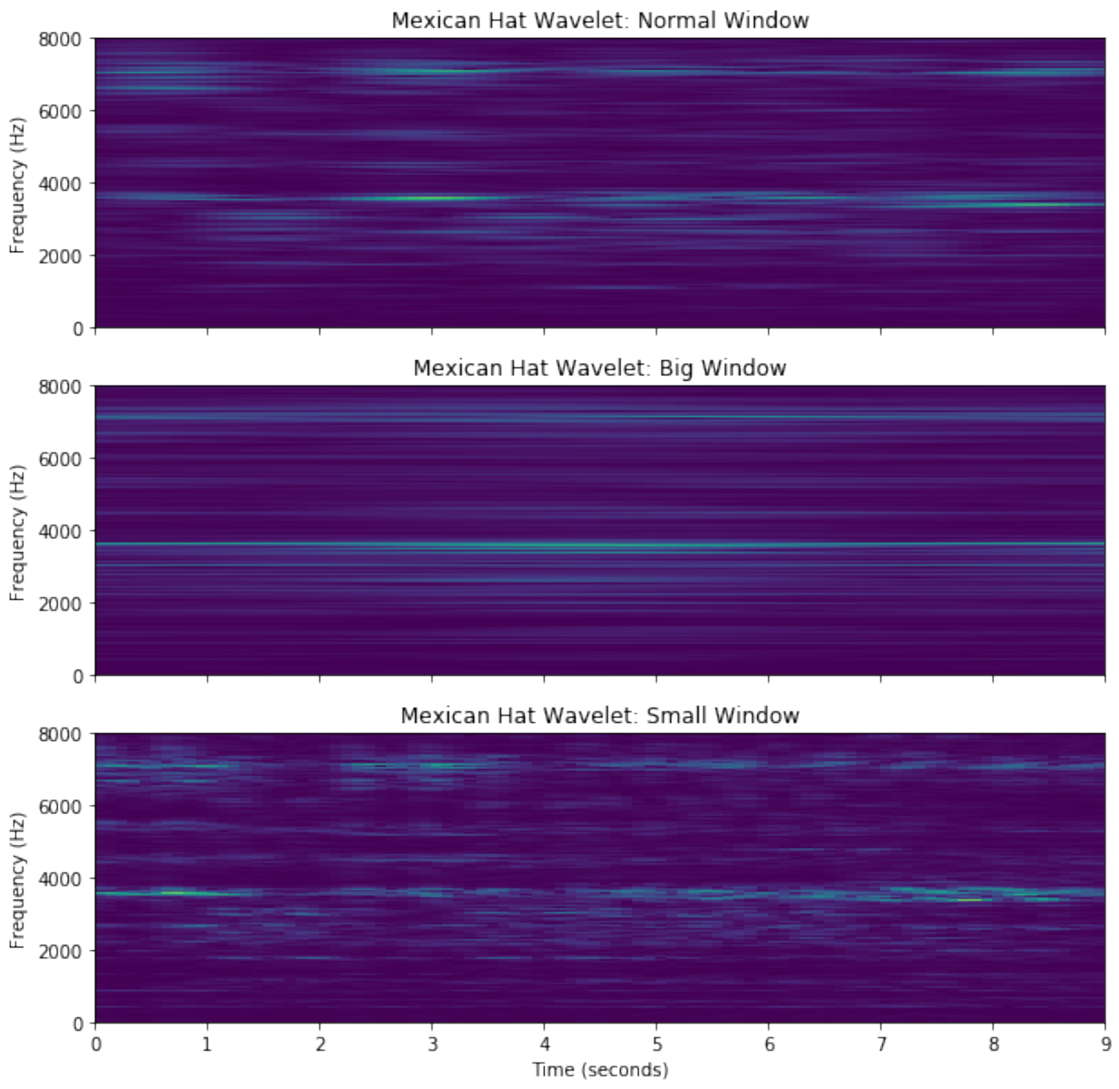
```
ax3.title.set_text('Gaussian Wavelet: Small Window')

# Note, I intentionally did not label each x-axis because it makes everything␣
↪too cluttered
ax3.set_xlabel('Time (seconds)')

ax1.set_ylabel('Frequency (Hz)')
ax2.set_ylabel('Frequency (Hz)')
ax3.set_ylabel('Frequency (Hz)')

# Set y-axis limits
ax1.set_ylim([0,8000])
```

```
[42]: (0, 8000)
```



```
[36]: fig, (ax1,ax2,ax3) = plt.subplots(nrows=3, ncols=1, figsize=(10,10),␣
       ↪sharex=True, sharey=True)

      # Create graphs
      ax1.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(mexican_hat_spc).
       ↪transpose())
      ax2.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(mexican_big_spc).
       ↪transpose())
      ax3.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(mexican_small_spc).
       ↪transpose())

      # Set titles
      ax1.title.set_text('Mexican Hat Wavelet: Normal Window')
      ax2.title.set_text('Mexican Hat Wavelet: Big Window')
      ax3.title.set_text('Mexican Hat Wavelet: Small Window')

      # Note, I intentionally did not label each x-axis because it makes everything␣
       ↪too cluttered
      ax3.set_xlabel('Time (seconds)')

      ax1.set_ylabel('Frequency (Hz)')
      ax2.set_ylabel('Frequency (Hz)')
      ax3.set_ylabel('Frequency (Hz)')

      # Set y-axis limits
      ax1.set_ylim([0,8000])
```

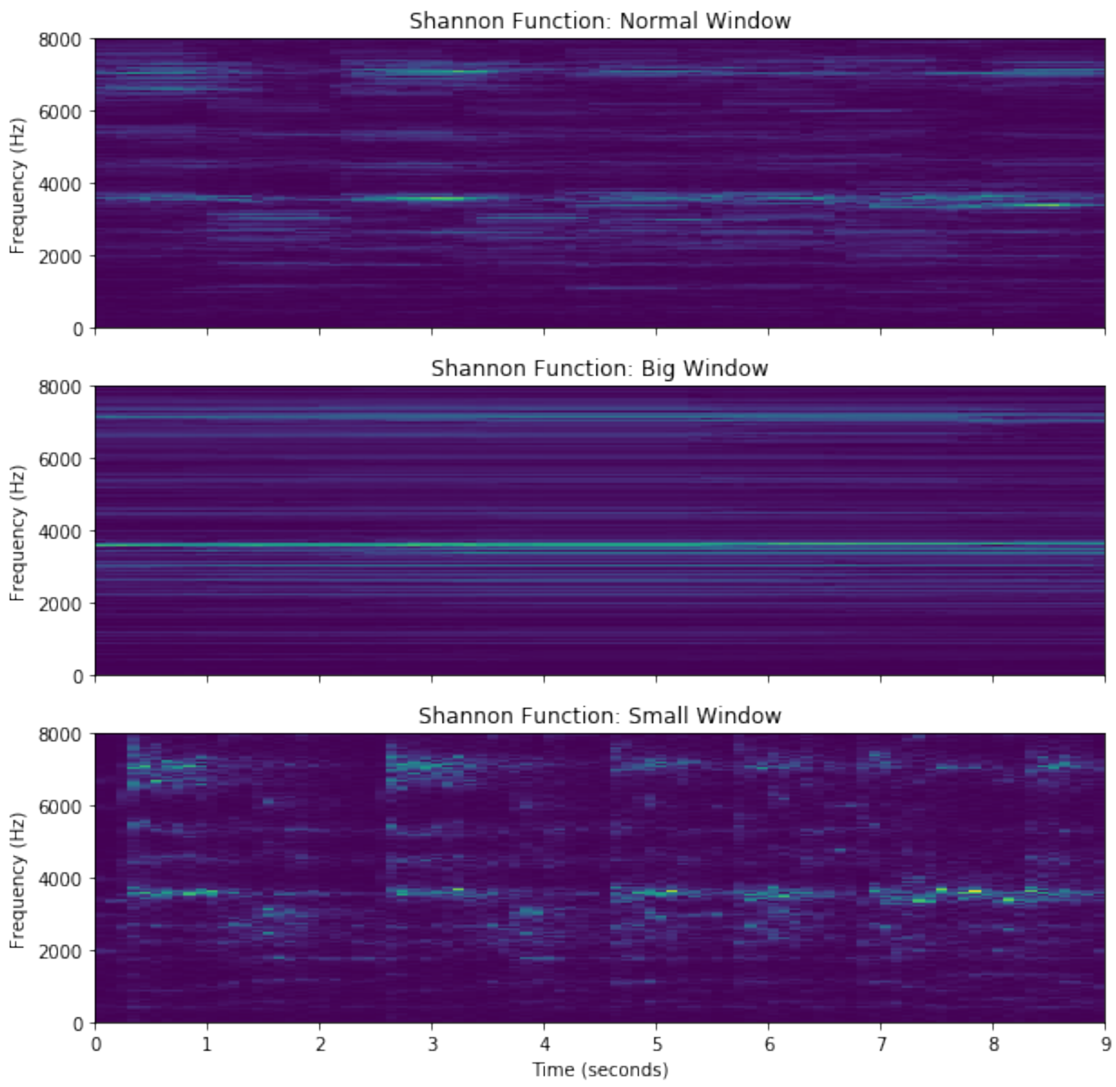
```
[36]: (0, 8000)
```



```
[37]: fig, (ax1,ax2,ax3) = plt.subplots(nrows=3, ncols=1, figsize=(10,10),␣
       ↪sharex=True, sharey=True)

      # Create graphs
      ax1.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(shannon_spc).
       ↪transpose())
      ax2.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(shannon_big_spc).
       ↪transpose())
      ax3.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(shannon_small_spc).
       ↪transpose())

      # Set titles
      ax1.title.set_text('Shannon Function: Normal Window')
```

```
ax2.title.set_text('Shannon Function: Big Window')
ax3.title.set_text('Shannon Function: Small Window')

# Note, I intentionally did not label each x-axis because it makes everything␣
↪too cluttered
ax3.set_xlabel('Time (seconds)')

ax1.set_ylabel('Frequency (Hz)')
ax2.set_ylabel('Frequency (Hz)')
ax3.set_ylabel('Frequency (Hz)')

# Set y-axis limits
ax1.set_ylim([0,8000])
```

```
[37]: (0, 8000)
```



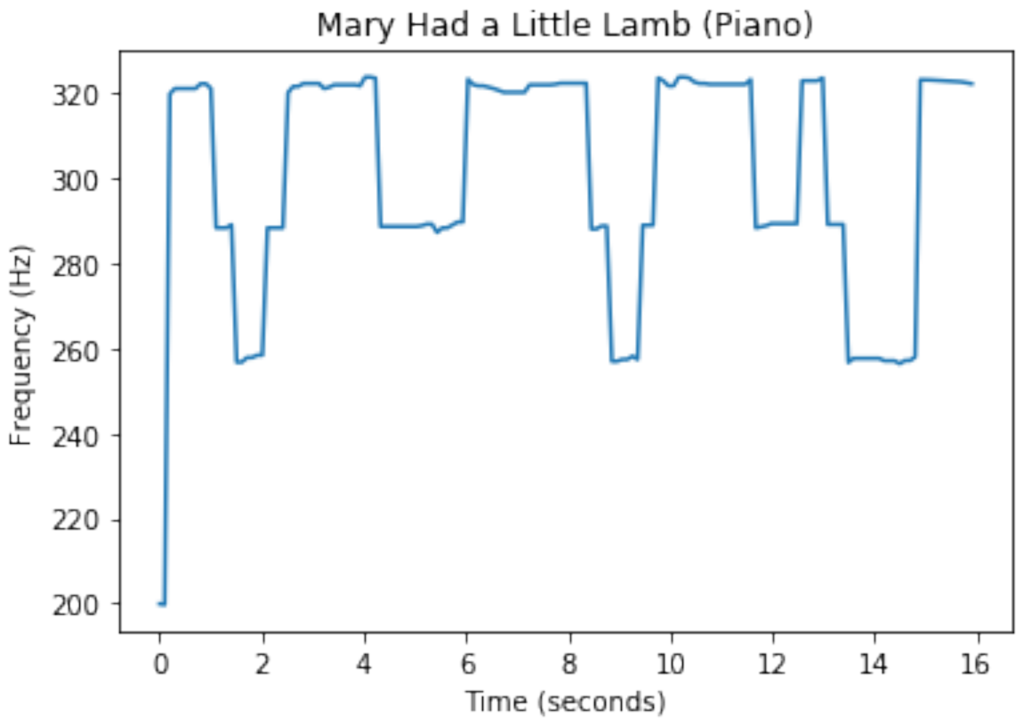
Part II: Mary Had a Little Lamb (Piano)

[11]:
```
import wave
import soundfile as sf

#tr_piano = 16
data, samplerate = sf.read('music1.wav')

L = len(data) / samplerate
n = 701440
t2 = np.linspace(0,L, n+1)
t = t2[0:n]

# Create k variable
list1 = np.linspace(0,n/2-1,num=(n/2-1*.0000001)) # n is length of amplitude
list2 = np.linspace(-n/2,-1,num=(n/2-1*.0000001))
list3 = []

for i in list1:
    list3.append(i)

for i in list2:
    list3.append(i)

# Convert list3 to a numpy array so it can be multiplied
array3 = np.asarray(list3)

k = (2*np.pi / L)* array3 # Rescale wavenumbers because FFT assumes 2*pi␣
 ↪periodic signals
```

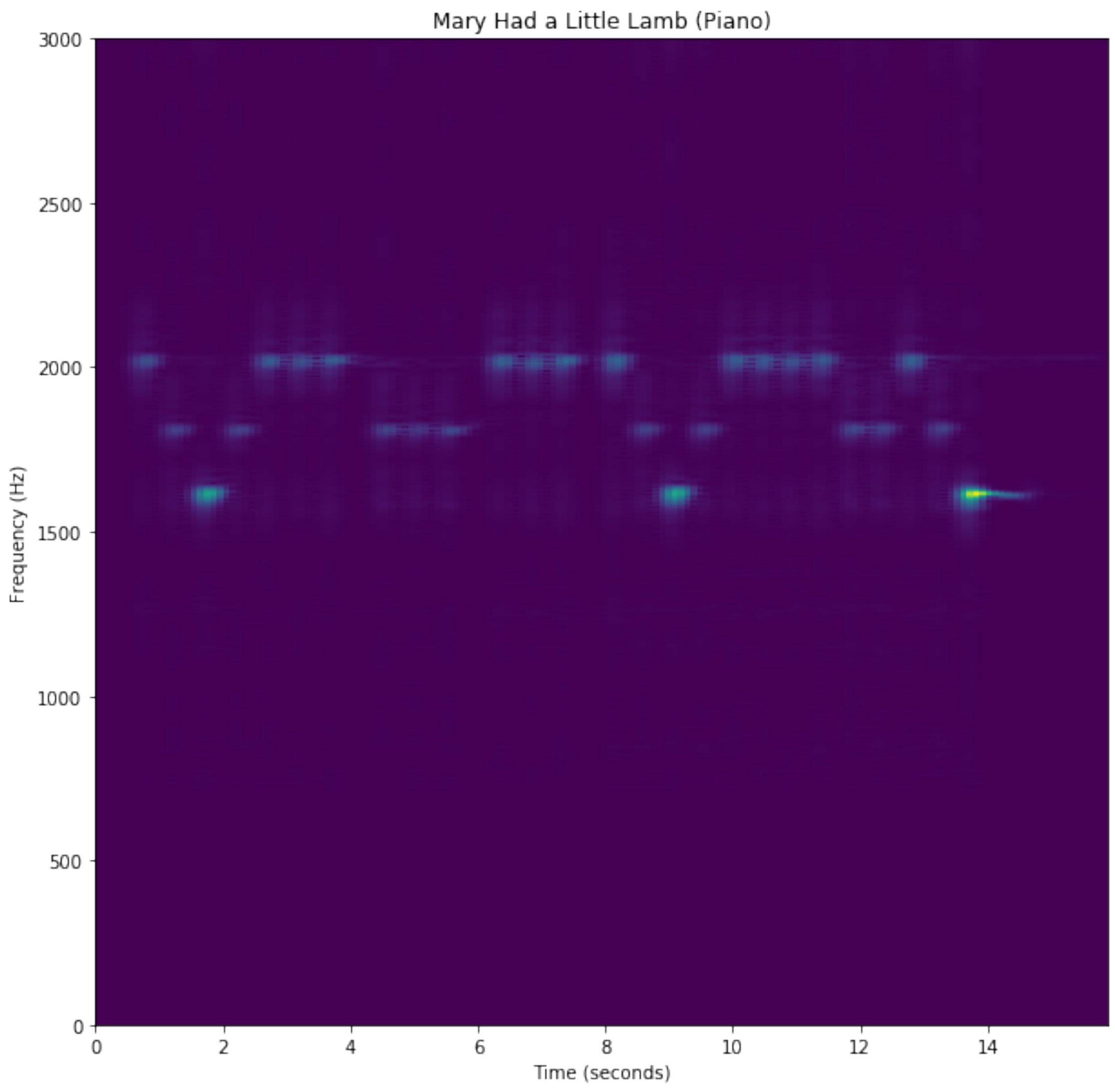
[12]:
```
t_slide = np.linspace(0,16,16/.1)
Max = []
spc = []

for i in range(1,len(t_slide)):
    g = np.exp(-20*(t-t_slide[i])**2)
    Sf = g[1:]*(data[0:-1]); # signal filter
    Sft = np.fft.fft(Sf);
    f_max = max(Sft)
    index_max = np.argmax(np.real(Sft))
    Max.append(k[index_max])
    spc.append(abs(np.fft.fftshift(Sft)))
```

[13]:
```
plt.plot(t_slide[0:-1], abs(np.array(Max)/(2*np.pi)))
plt.title('Mary Had a Little Lamb (Piano)')
plt.xlabel('Time (seconds)'), plt.ylabel('Frequency (Hz)')
```

[13]:
```
(Text(0.5, 0, 'Time (seconds)'), Text(0, 0.5, 'Frequency (Hz)'))
```



[14]:
```
fig, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(10,10), sharex=True,␣
 ↪sharey=True)
ax1.pcolormesh(t_slide[0:-1], np.fft.fftshift(k), np.array(spc).transpose()[0:
 ↪-1,:])
#ax1.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(mexican_hat_spc).
 ↪transpose())
plt.ylim(0,3000)
plt.title("Mary Had a Little Lamb (Piano)")
plt.xlabel("Time (seconds)")
plt.ylabel('Frequency (Hz)')
plt.show()
```



Part II: Mary Had a Little Lamb (Recorder)

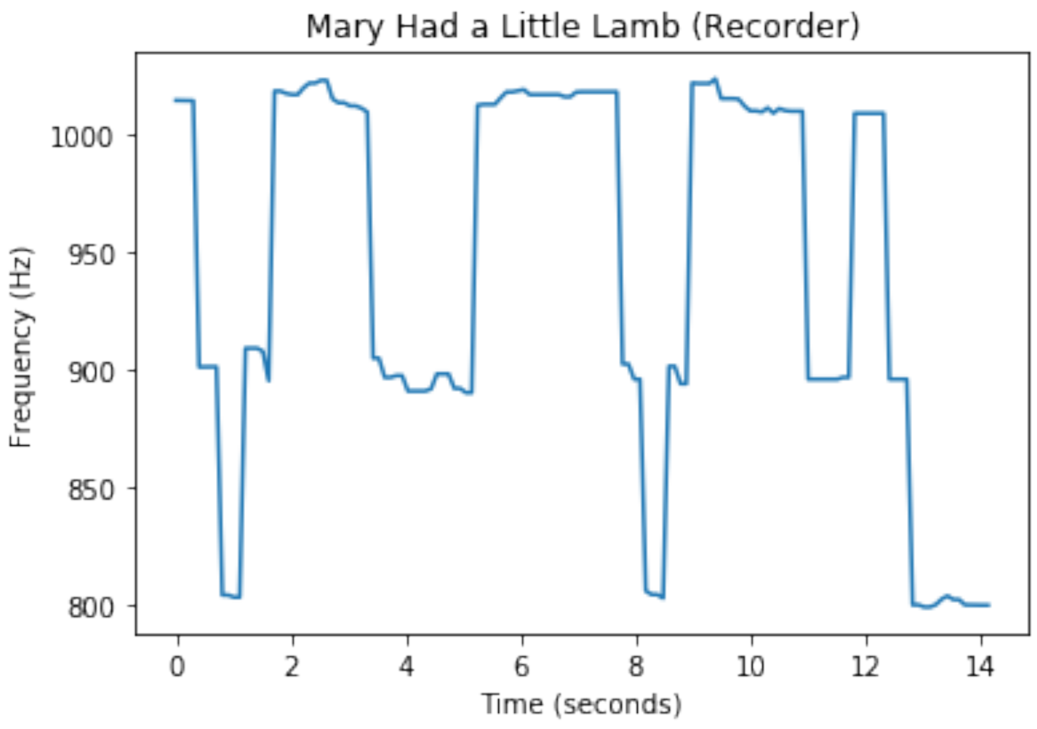
[15]:
```
tr_rec = 14
data, samplerate = sf.read('music2.wav')

L = len(data) / samplerate
n = 627712
t2 = np.linspace(0,L, n+1)
t = t2[0:n]

# Create k variable
list1 = np.linspace(0,n/2-1,num=(n/2-1*.0000001)) # n is length of amplitude
list2 = np.linspace(-n/2,-1,num=(n/2-1*.0000001))
list3 = []
```

```
for i in list1:
    list3.append(i)

for i in list2:
    list3.append(i)

# Convert list3 to a numpy array so it can be multiplied
array3 = np.asarray(list3)

k = (2*np.pi / L)* array3 # Rescale wavenumbers because FFT assumes 2*pi␣
 ↪periodic signals
```

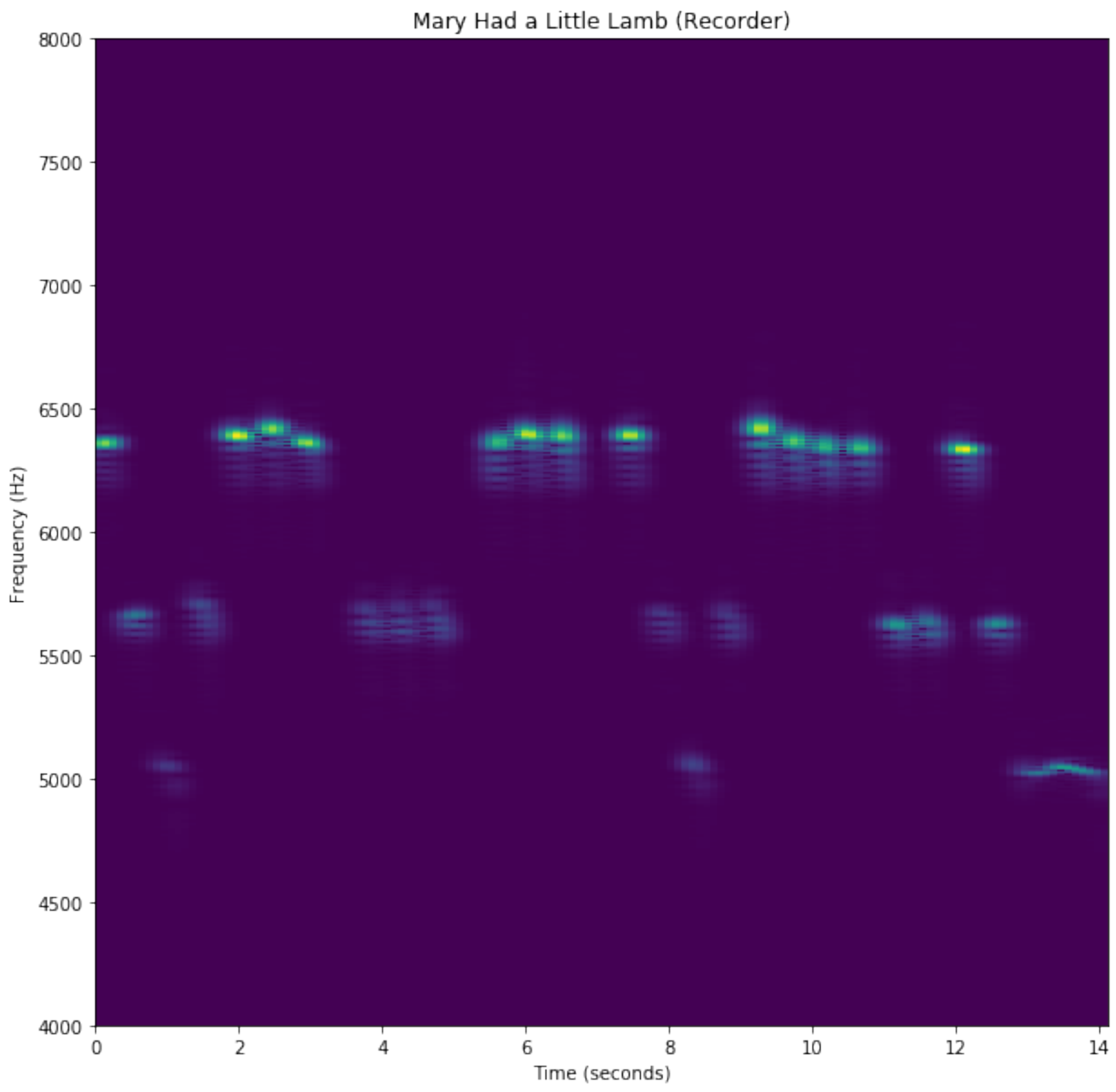
[16]:
```
t_slide = np.linspace(0,L,L/.1)
Max = []
spc = []

for i in range(1,len(t_slide)):
    g = np.exp(-20*(t-t_slide[i])**2)
    Sf = g[1:]*(data[0:-1]); # signal filter
    Sft = np.fft.fft(Sf);
    f_max = max(Sft)
    index_max = np.argmax(np.real(Sft))
    Max.append(k[index_max])
    spc.append(abs(np.fft.fftshift(Sft)))
```

[17]:
```
plt.plot(t_slide[0:-1], abs(np.array(Max)/(2*np.pi)))
plt.title('Mary Had a Little Lamb (Recorder)')
plt.xlabel('Time (seconds)'), plt.ylabel('Frequency (Hz)')
```

[17]:
```
(Text(0.5, 0, 'Time (seconds)'), Text(0, 0.5, 'Frequency (Hz)'))
```



[18]:
```
fig, ax1 = plt.subplots(nrows=1, ncols=1, figsize=(10,10), sharex=True,␣
 ↪sharey=True)
ax1.pcolormesh(t_slide[0:-1], np.fft.fftshift(k), np.array(spc).transpose()[0:
 ↪-1,:])
#ax1.pcolormesh(tslide, np.fft.fftshift(k[0:-1]),np.array(mexican_hat_spc).
 ↪transpose())
plt.ylim(4000,8000)
plt.title("Mary Had a Little Lamb (Recorder)")
plt.xlabel("Time (seconds)")
plt.ylabel('Frequency (Hz)')
plt.show()
```


Mary Had a Little Lamb (Recorder)
Frequency (Hz)
Time (seconds)
8000
7500
7000
6500
6000
5500
5000
4500
4000
0
2
4
6
8
10
12
14

```
[ ]:
```

```
[ ]:
```