A4: Short-time Fourier Transform (STFT)

Audio Signal Processing for Music Applications

Introduction

Doing this assignment you will learn about the concept of the main lobe width of the spectrum of a window and you will understand in practice the short-time Fourier transform (STFT). You will also use STFT to extract basic rhythm related information from an audio signal. Specifically, you will write snippets of code to compute an onset detection function, which is one of the rhythm descriptors often used in music information retrieval to detect onsets of different acoustic events. There are four parts in this assignment. 1) Extracting the main lobe of the spectrum of a window, 2) Measuring noise in the reconstructed signal using the STFT model, 3) Computing band-wise energy envelopes of a signal, 4) Computing an onset detection function (optional). The last part is optional and will not count towards the final grade.
A brief description of the relevant concepts required to solve this assignment is given below. Subsequently, each part of this assignment is described in detail.

Relevant Concepts

Main lobe of the spectrum of a window: The width of the main lobe of the magnitude spectrum of a window is an important characteristic used in deciding which window type is best for the analysis of an audio excerpt. There exists a tradeoff between the main lobe width and the side lobe attenuation. Typically for windows with a narrower main lobe, the side lobes are less attenuated. An example of the Blackman window is given below. Figure 1 shows the plot of the time domain window (blue) together with its magnitude spectrum (red). Both the time domain function and its spectrum are centered around 0. The equation of the time domain function describing the window is:

$$w[n] = 0.42 - 0.5\cos(2\pi n/M) + 0.08\cos(4\pi n/M), \quad 0 \leq n \leq M-1 \tag{1}$$

where M is the size of the window and n is the discrete time index.
An interesting fact is that changing the length of a window M doesn't affect the main lobe width of the spectrum of the window in samples. Note that if you use zero-padding for computing the spectrum of a window, the main lobe width will be multiplied by the zero-padding factor.

Fast Fourier Transform (FFT): An efficient way to compute the discrete Fourier transform of a signal is the fast Fourier transform. The FFT algorithm factorizes the DFT matrix in order to exploit the symmetries in the DFT equation. FFT computation is specially very efficient when the FFT size is a power of 2. Therefore, whenever possible we use an FFT size that is a power of 2.

Energy of a signal: The energy of a signal $x[n]$ of length N can be computed in the discrete time domain as follows:

$$E = \sum_{n=0}^{N-1} |x\,[n]|^2 \tag{2}$$

Energy in a frequency band: Given the DFT spectrum of the signal $X[k]$, the energy E in a

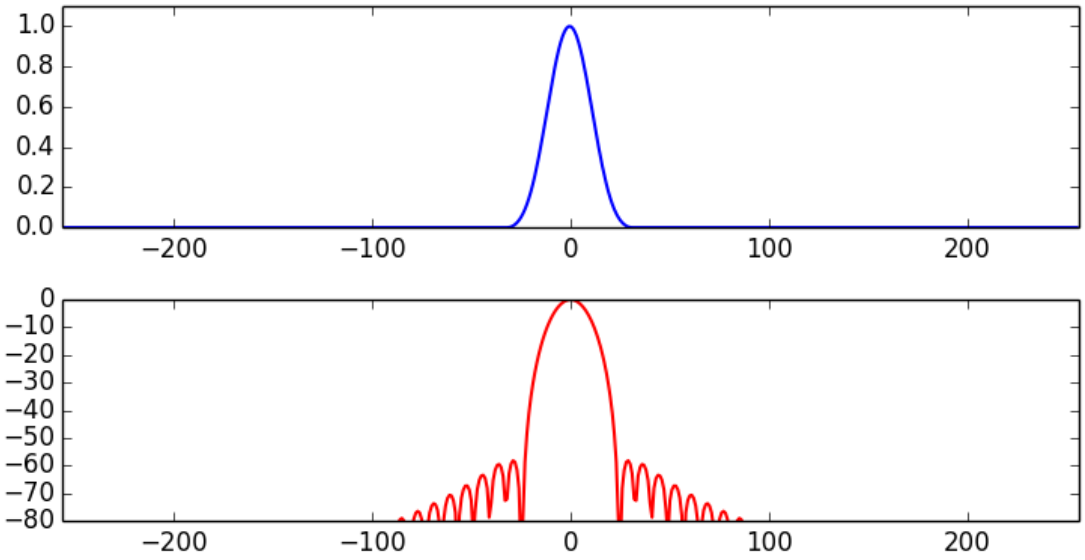

Figure 1: Plot of the Blackman window (blue) and its magnitude spectrum (red)

specific frequency band spanning the bin index k_1 to k_2 can be computed as:

$$E = \sum_{k=k_1}^{k_2} |X[k]|^2 \tag{3}$$

Note that in this computation the $X[k]$ values are not in decibels (dB). `stftAnal` function returns spectrum in dB scale, which should be converted to linear scale before the energy computation. However, once the energy is computed it can be converted back to the dB scale as:

$$E_{dB} = 10 \log_{10}(E) \tag{4}$$

Signal to noise ratio (SNR): Signal to noise ratio (SNR) is a frequently used measure to quantify the amount of noise present/added in a signal. In the context of this assignment it can be computed in decibels (dB) as:

$$\text{SNR} = 10 \log_{10}\left(\frac{E_{\text{signal}}}{E_{\text{noise}}}\right) \tag{5}$$

where, E_{signal} and E_{noise} is the energy of the signal and the noise respectively.
For our case of analysis and synthesis using a specific model (for example, STFT) noise can be thought of as the difference between the input signal and the output signal of the model.

Onset detection function: An onset detection function (ODF) refers to a continuous function (one value per audio frame) often used for detecting acoustic events in an audio stream. In music information retrieval (MIR), ODFs are typically used for detecting onsets of musical notes and percussion strokes. An ODF generally has high values at the onsets of acoustic events. As can be imagined, a simple ODF can be computed by taking the difference between the energy values of consecutive frames, as shown below:

$$O(l) = E(l) - E(l-1), \quad l \geq 1 \tag{6}$$

where, $O(l)$ is the ODF computed at frame index l and E is the energy of the signal in a particular frequency band in decibels (dB). Often, multiple ODFs are computed with different frequency bands across the spectrum.

In order to detect only the onsets of the events and not the offsets, it is a common practice to half wave rectify the ODF and obtain $\bar{O}(l)$. Half wave rectification of the ODF is given by:

$$\bar{O}(l) = \begin{cases} O(l), \text{ if } O(l) > 0 \\ 0, \quad \text{if } O(l) \leq 0 \end{cases} \tag{7}$$

Part-1: Extracting the main lobe of the spectrum of a window (*3 points*)

Complete the function `extractMainLobe(window, M)` in the file `A4Part1.py` that extracts the main lobe of the magnitude spectrum of a window given a window type and its length (M). The function should return the samples corresponding to the main lobe in decibels (dB).
To compute the spectrum, take the FFT size (N) to be 8 times the window length ($N = 8\,M$) (For this part, N need not be a power of 2).
The input arguments to the function are the window type (`window`) and the length of the window (`M`). The function should return a numpy array containing the samples corresponding to the main lobe of the window.
In the returned numpy array you should include the samples corresponding to both the local minimas across the main lobe.
The possible window types that you can expect as input are rectangular (`'boxcar'`), `'hamming'` or `'blackmanharris'`.
You can approach this question in two ways:

1. You can write code to find the indices of the local minimas across the main lobe.
2. You can manually note down the indices of these local minimas by plotting and a visual inspection of the spectrum of the window. If done manually, the indices have to be obtained for each possible window types separately (as they differ across different window types).

Tip: $\log_{10}(0)$ is not well defined, so its a common practice to add a small value such as $\epsilon = 10^{-16}$ to the magnitude spectrum before computing it in dB. This is optional and will not affect your answers. If you find it difficult to concatenate the two halves of the main lobe, you can first center the spectrum using `fftshift()` and then compute the indexes of the minimas around the main lobe.
Test case 1: If you run your code using `window` = `'blackmanharris'` and `M` = 100, the output numpy array should contain 65 samples.
Test case 2: If you run your code using `window` = `'boxcar'` and `M` = 120, the output numpy array should contain 17 samples.
Test case 3: If you run your code using `window` = `'hamming'` and `M` = 256, the output numpy array should contain 33 samples.

```
def extractMainLobe(window, M):
    """
    Input:
      window (string): Window type to be used (Either rectangular ('boxcar'),
          'hamming' or 'blackmanharris')
      M (integer): length of the window to be used
    Output:
      The function should return a numpy array containing the main lobe of
      the magnitude spectrum of the window in decibels (dB).
    """
    w = get_window(window, M)         # get the window

    ### Your code here
```

Part-2: Measuring noise in the reconstructed signal using the STFT model (*3 points*)

Complete the function `computeSNR(inputFile, window, M, N, H)` in the file `A4Part2.py` that measures the amount of noise introduced during the analysis and synthesis of a signal using the STFT model. Use SNR (signal to noise ratio) in dB to quantify the amount of noise. Use the `stft()` function in the stft model to do an analysis followed by a synthesis of the input signal. A brief description of the SNR computation is given above in the Relevant Concepts section. Use the time domain energy definition to compute the SNR.
With the input signal and the obtained output, compute two different SNR values for the following cases:

1. SNR1: Over the entire length of the input and the output signals.
2. SNR2: For the segment of the signals left after discarding M samples from both the start and the end, where M is the analysis window length. Note that this computation is done after STFT analysis and synthesis.

The input arguments to the function are the wav file name including the path (`inputFile`), window type (`window`), window length (`M`), FFT size (`N`), and hop size (`H`). The function should return a python tuple of both the SNR values in decibels: (`SNR1`, `SNR2`). Both `SNR1` and `SNR2` are float values.
Test case 1: If you run your code using `piano.wav` file (in the sounds folder of sms-tools) with `'blackman'` window, `M` = 513, `N` = 2048 and `H` = 128, the output SNR values should be around: (67.57, 304.68)
Test case 2: If you run your code using `sax-phrase-short.wav` file with `'hamming'` window, `M` = 512, `N` = 1024 and `H` = 64, the output SNR values should be around: (89.51, 306.19)
Test case 3: If you run your code using `rain.wav` file with `'hann'` window, `M` = 1024, `N` = 2048 and `H` = 128, the output SNR values should be around: (74.63, 304.27)
Due to precision differences on different machines/hardware, compared to the expected SNR values, your output values can differ by ±10 dB for `SNR1` and ±100 dB for `SNR2`.

```
def computeSNR(inputFile, window, M, N, H):
    """
    Input:
      inputFile (string): input sound file (monophonic with sampling rate of 44100)
      window (string): analysis window type (choice of rectangular, triangular,
          hanning, hamming, blackman, blackmanharris)
      M (integer): analysis window length (odd positive integer)
      N (integer): fft size (power of two, such that N > M)
      H (integer): hop size for the stft computation
    Output:
      The function should return a python tuple of both the SNR values (SNR1, SNR2)
      SNR1 and SNR2 are floats.
    """
    ### your code here
```

Part-3: Computing band-wise energy envelopes of a signal (*4 points*)

Complete the function `computeEngEnv(inputFile, window, M, N, H)` in the file `A4Part3.py` that computes band-wise energy envelopes of a given audio signal by using the STFT. Consider two frequency bands for this question, low and high. The low frequency band is the set of all the frequencies between `0` and `3000` Hz and the high frequency band is the set of all the frequencies between `3000` and `10000` Hz (excluding the boundary frequencies in both the cases). At a given frame, the value of the energy envelope of a band can be computed as the sum of

squared values of all the frequency coefficients in that band. Compute the energy envelopes in decibels. Further details to compute band-wise energy can be found in the Relevant Concepts section of this document.

The input arguments to the function are the wav file name including the path (`inputFile`), window type (`window`), window length (`M`), FFT size (`N`) and hop size (`H`). The function should return a numpy array with two columns, where the first column is the energy envelope of the low frequency band and the second column is that of the high frequency band.

Use `stft.stftAnal()` to obtain the STFT magnitude spectrum for all the audio frames. Then compute two energy values for each frequency band specified. While calculating frequency bins for each frequency band, consider only the bins that are within the specified frequency range. For example, for the low frequency band consider only the bins with frequency f, $0 < f < 3000$ Hz (you can use `np.where()` to find those bin indexes). This way we also remove the DC offset in the signal in energy envelope computation. The frequency corresponding to the bin index k can be computed as $k \times f_s/N$, where f_s is the sampling rate of the signal.

To get a better understanding of the energy envelope and its characteristics you can plot the envelopes together with the spectrogram of the signal. You can use matplotlib plotting library for this purpose. To visualize the spectrogram of a signal, a good option is to use colormesh. You can reuse the code in `sms-tools/lectures/4-STFT/plots-code/spectrogram.py`. Either overlay the envelopes on the spectrogram or plot them in a different subplot. Make sure you use the same range of the x-axis for both the spectrogram and the energy envelopes.

Example: Running your code on `piano.wav` file with window = `'blackman'`, `M` = 513, `N` = 1024, `H` = 128, in the plots you can clearly notice the sharp attacks and decay of the piano notes (Figure 2 shows the spectrogram and the energy envelopes). In addition, you can also visually analyse which of the two energy envelopes is better for detecting onsets of the piano notes.

Test case 1: Use `piano.wav` file with `window = 'blackman'`, `M` = 513, `N` = 1024 and `H` = 128 as input. The bin indexes of the low frequency band span from 1 to 69 (69 samples) and of the high frequency band span from 70 to 232 (163 samples). To numerically compare your output, use `loadTestCases.py` script to obtain the expected output.

Test case 2: Use `piano.wav` file with `window = 'blackman'`, `M` = 2047, `N` = 4096 and `H` = 128 as input. The bin indexes of the low frequency band span from 1 to 278 (278 samples) and of the high frequency band span from 279 to 928 (650 samples). To numerically compare your output, use `loadTestCases.py` script to obtain the expected output.

Test case 3: Use `sax-phrase-short.wav` file with `window = 'hamming'`, `M` = 513, `N` = 2048 and `H` = 256 as input. The bin indexes of the low frequency band span from 1 to 139 (139 samples) and of the high frequency band span from 140 to 464 (325 samples). To numerically compare your output, use `loadTestCases.py` script to obtain the expected output.

In addition to comparing results with the expected output, you can also plot your output for these test cases. You can clearly notice the sharp attacks and decay of the piano notes for test case 1 (See Figure 2). You can compare this with the output from test case 2 that uses a larger window. You can infer the influence of window size on sharpness of the note attacks and discuss it on the forums.

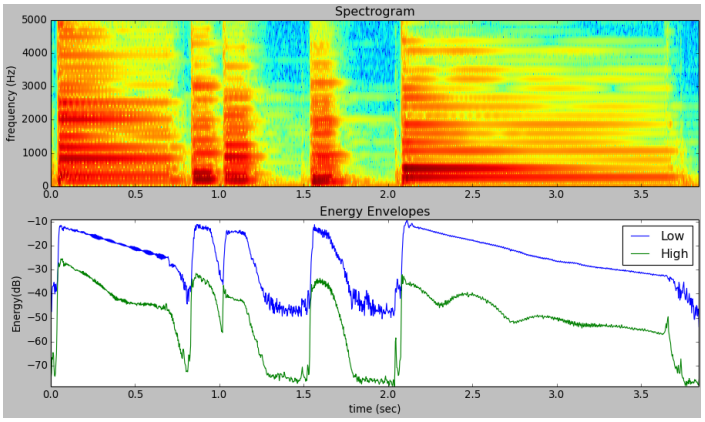

Figure 2: The spectrogram and the energy envelopes of piano.wav

```
def computeEngEnv(inputFile, window, M, N, H):
    """
    Inputs:
      inputFile (string): input sound file (monophonic with sampling rate of 44100)
      window (string): analysis window type (choice of rectangular, triangular,
          hanning, hamming, blackman, blackmanharris)
      M (integer): analysis window size (odd positive integer)
      N (integer): FFT size (power of 2, such that N > M)
      H (integer): hop size for the stft computation
    Output:
      The function should return a numpy array engEnv with shape Kx2,
      K = Number of frames
      containing energy envelop of the signal in decibles (dB) scale
      engEnv[:,0]: Energy envelope in band 0 < f < 3000 Hz (in dB)
      engEnv[:,1]: Energy envelope in band 3000 < f < 10000 Hz (in dB)
    """
    ### your code here
```

Part-4: Computing onset detection function (*Optional*)

Complete the function `computeODF(inputFile, window, M, N, H)` in the file `A4Part4.py` that computes a simple onset detection function (ODF) using the STFT. Compute two ODFs one for each of the frequency bands, low and high. The low frequency band is the set of all the frequencies between `0` and `3000` Hz and the high frequency band is the set of all the frequencies between `3000` and `10000` Hz (excluding the boundary frequencies in both the cases).

A brief description of the onset detection function is provided in the Relevant Concepts section of this document. Start with an initial condition of ODF(0) = 0 in order to make the length of the ODF same as that of the energy envelope. Remember to apply a half wave rectification on the ODF.

The input arguments to the function are the wav file name including the path (`inputFile`), window type (`window`), window length (`M`), FFT size (`N`), and hop size (`H`). The function should return a numpy array with two columns, where the first column is the ODF computed on the low frequency band and the second column is the ODF computed on the high frequency band.

Use `stft.stftAnal()` to obtain the STFT magnitude spectrum for all the audio frames. Then compute two energy values for each frequency band specified. While calculating frequency bins for each frequency band, consider only the bins that are within the specified frequency range. For example, for the low frequency band consider only the bins with frequency f, $0 < f < 3000$ Hz (you can use `np.where()` to find those bin indexes). This way we also remove the DC offset in

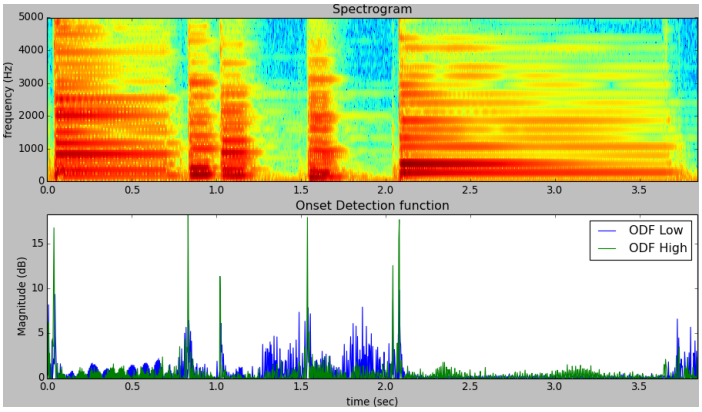

Figure 3: The spectrogram and the Onset Detection Functions of piano.wav

the signal in energy envelope computation. The frequency corresponding to the bin index k can be computed as $k \times f_s/N$, where f_s is the sampling rate of the signal.

To get a better understanding of the energy envelope and its characteristics you can plot the envelopes together with the spectrogram of the signal. You can use matplotlib plotting library for this purpose. To visualize the spectrogram of a signal, a good option is to use colormesh. You can reuse the code in `sms-tools/lectures/4-STFT/plots-code/spectrogram.py`. Either overlay the envelopes on the spectrogram or plot them in a different subplot. Make sure you use the same range of the x-axis for both the spectrogram and the energy envelopes.

Test case 1: Use `piano.wav` file with `window = 'blackman'`, `M` = 513, `N` = 1024 and `H` = 128 as input. The bin indexes of the low frequency band span from 1 to 69 (69 samples) and of the high frequency band span from 70 to 232 (163 samples). To numerically compare your output, use `loadTestCases.py` script to obtain the expected output.

Test case 2: Use `piano.wav` file with `window = 'blackman'`, `M` = 2047, `N` = 4096 and `H` = 128 as input. The bin indexes of the low frequency band span from 1 to 278 (278 samples) and of the high frequency band span from 279 to 928 (650 samples). To numerically compare your output, use `loadTestCases.py` script to obtain the expected output.

Test case 3: Use `sax-phrase-short.wav` file with `window = 'hamming'`, `M` = 513, `N` = 2048 and `H` = 256 as input. The bin indexes of the low frequency band span from 1 to 139 (139 samples) and of the high frequency band span from 140 to 464 (325 samples). To numerically compare your output, use `loadTestCases.py` script to obtain the expected output.

In addition to comparing results with the expected output, you can also plot your output for these test cases. For test case 1, you can clearly see that the ODFs have sharp peaks at the onset of the piano notes (See (Figure 3). You will notice exactly 6 peaks that are above 10 dB value in the ODF computed on the high frequency band.

```
def computeODF(inputFile, window, M, N, H):
    """
    Inputs:
      inputFile (string): input sound file (monophonic with sampling rate of 44100)
      window (string): analysis window type (choice of rectangular, triangular,
          hanning, hamming, blackman, blackmanharris)
      M (integer): analysis window size (odd integer value)
      N (integer): fft size (power of two, such that N > M)
      H (integer): hop size for the STFT computation
    Output:
      The function should return a numpy array with two columns, where the first
      column is the ODF computed on the low frequency band and the second column
      is the ODF computed on the high frequency band.
      ODF[:,0]: ODF computed in band 0 < f < 3000 Hz
```

```
  ODF[:,1]: ODF computed in band 3000 < f < 10000 Hz
"""
### your code here
```

Grading

Only the first three parts of this assignment are graded and the fourth part is optional. The total points for this assignment is 10.