A5: Sinusoidal model

Audio Signal Processing for Music Applications

Introduction

In this assignment you will experiment with the sinusoidal model, measuring and tracking sinusoids in different kinds of audio signals. You will use the sinusoidal model to analyze short synthetic sounds with the goal to better understand various aspects of sinusoid estimation and tracking. You will experiment with different parameters and enhancements of the sinusoidal modeling approach. There are five parts in this assignment: 1) Minimizing the frequency estimation error of a sinusoid 2) Tracking a two component chirp 3) Tracking sinusoids of different amplitudes 4) Tracking sinusoids using the phase spectrum 5) Sinusoidal modeling of a multicomponent signal (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

Chirp signals: A chirp is a signal whose frequency varies with time. In an up-chirp, the frequency increases in time. In Part 2 of the assignment, we will use a synthetically generated linear chirp with two frequency components. The frequency components are very close to each other and to resolve the two components, you need to use a large window. However, the frequency of a chirp continuously changes, which implies that we need a shorter window for analysis to capture this continuously changing frequency. Hence, there exists a tradeoff between the best set of sinusoidal analysis parameters to achieve a good tracking of the two components of the chirp.

Sinusoidal modeling and sine tracking: Sinusoidal modeling aims to model each frame of audio with a set of sinusoids, from which we can reconstruct the input audio with minimum reconstruction error. The basic peak detection task performed in the spectrum of a frame can be enhanced in many ways to obtain the most compact representation possible and the most meaningful one for a particular task. One of the enhancements implemented in sms-tools is the tracking of the estimated sinusoids over time. In sounds with stable notes, the sinusoids tend to last over several frames and this can be used to discard spurious sinusoids that have been estimated. There are two functions in `sineModel.py` that together perform sine tracking. The function `sineTracking()` tracks peaks from one frame to the next, to give tracks of the sinusoids in time. The function `cleaningSineTracks()` then cleans up the tracks by discarding short spurious sinusoids based on a length threshold. These functions use the following parameters to do sine tracking.

1. `maxnSines`: Maximum number of sines tracked per frame.
2. `minSineDur`: Minimum duration of a sinusoidal track in seconds.
3. `freqDevOffset`: The minimum frequency deviation at 0Hz. Since the frequency of sinusoidal tracks can change slowly over time, it is necessary to have a margin of allowed deviation to track the change over time.
4. `freqDevSlope`: Slope increase of minimum frequency deviation. The common deviations are more pronounced at higher frequency and we compensate for that using a scaling factor, which provides a higher deviation allowance at higher frequencies.

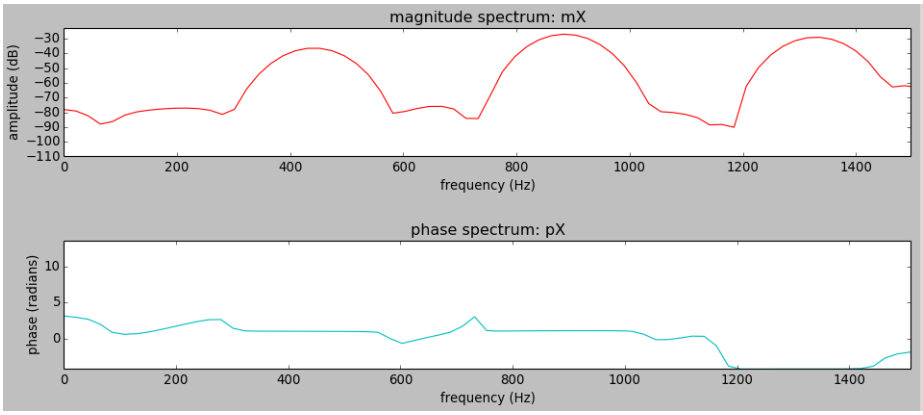

Figure 1: A part of the magnitude and phase spectrum of oboe-A4.wav. Observe how the phase is flat across the width of the mainlobes.

If we set `minSineDur` = 0, it retains all the spectral peaks detected. `maxnSines` can be set according to our prior knowledge about the number of sinusoids present in the signal. `freqDevOffset` intuitively can be set to be the minimum distance between two consecutive sinusoids (or harmonics in the case of harmonic sounds).

Tracking low amplitude sinusoids: Tracking low amplitude sinusoids present among other high amplitude sinusoids presents several challenges. Let us consider a signal that has two sinusoidal components with a very large difference in their amplitude. In such a signal, the analysis window we use is critical. If we use a window that has high sidelobe levels, the mainlobe of the sinusoid with low amplitude might get buried under the sidelobes of the dominant high amplitude sinusoid. When the amplitude difference is huge you need to choose a window that will ensure that the sidelobe levels of the louder sinusoid is lower than the mainlobe level of the fainter sinusoid.

Using Phase to track sinusoid peaks: Recall that zero-phase windowing of sinusoid signal frame leads to a phase spectrum that is flat around the bins corresponding to frequency of the sinusoid. This property of flatness of the phase spectrum can be used to identify and track sinusoidal peaks. This can be an alternative, or a complementary way to the identification of sinusoids by just measuring local maxima in the magnitude spectrum. We can pick local maxima in the magnitude spectrum and also measure the variance of the phase spectrum around the local maximum to select or discard a particular spectral peak. However, it is to be noted that this condition of flatness is satisfied only when the sinusoid is not time varying, as shown in Figure 1 for oboe-A4.wav. For strongly time-varying sinusoids, the condition fails.

Part-1: Minimizing the frequency estimation error of a sinusoid (*3 points*)

Complete the function `minFreqEstErr(inputFile, f)` that estimates the frequency of a sinusoidal signal at a given time instant. The function should return the estimated frequency in Hz, together with the window size and the FFT size used in the analysis.

The input arguments to the function are the wav file name including the path (inputFile) containing the sinusoidal signal, and the frequency of the sinusoid in Hz (`f`). The frequency of the input sinusoid can range between 100 Hz and 2000 Hz. The function should return a three element tuple of the estimated frequency of the sinusoid (`fEst`), the window size (`M`) and the FFT size (`N`) used.

The input wav file is a stationary audio signal consisting of a single sinusoid of length $\geq$ 1 second. Since the signal is stationary you can just perform the analysis in a single frame, for example in the middle of the sound file (time equal to 0.5 seconds). The analysis process would be to first select a fragment of the signal equal to the window size, `M`, centered at 0.5 seconds, then

compute the DFT using the `dftAnal` function, and finally use the `peakDetection` and `peakInterp` functions to obtain the frequency value of the sinusoid.

Use a Blackman window for analysis and a magnitude threshold $t = -40\,\text{dB}$ for peak picking. The window size and FFT size should be chosen such that the difference between the true frequency (`f`) and the estimated frequency (`fEst`) is less than 0.05 Hz for the entire allowed frequency range of the input sinusoid. The window size should be the minimum positive integer of the form $100 * k + 1$ (where k is a positive integer) for which the frequency estimation error is $< 0.05\,\text{Hz}$. For a window size `M`, take the FFT size (`N`) to be the smallest power of 2 larger than `M`.

HINT: Computing `M` theoritically using a formula might be complex in such cases. Instead, you need to follow a heuristic approach to determine the optimal value of `M` and `N` for a particular `f`. You can iterate over all allowed values of window size `M` and stop when the condition is satisfied (i.e. the frequency estimation error ¡ 0.05 Hz).

Test case 1: If you run your code with `inputFile = '../../sounds/sine-490.wav'`, `f` = 490.0, the optimal values are `M` = 1101, `N` = 2048, `fEst` = 489.963 and the freqency estimation error is 0.037 Hz.

Test case 2: If you run your code with `inputFile = '../../sounds/sine-1000.wav'`, `f` = 1000.0, the optimal values are `M` = 1101, `N` = 2048, `fEst` = 1000.02 and the freqency estimation error is 0.02 Hz.

Test case 3: If you run your code with `inputFile = '../../sounds/sine-200.wav'`, `f` = 200.0 Hz, the optimal values are `M` = 1201, `N` = 2048, `fEst` = 200.038 and the freqency estimation error is 0.038 Hz.

```
def minFreqEstErr(inputFile, f):
    """
    Inputs:
      inputFile (string) = wav file including the path
      f (float) = frequency of the sinusoid present in the input audio signal (Hz)
    Output:
      fEst (float) = Estimated frequency of the sinusoid (Hz)
      M (int) = Window size
      N (int) = FFT size
    """
    # analysis parameters:
    window = 'blackman'
    t = -40

    ### Your code here
```

Part-2: Tracking a two component chirp (*2 points*)

Perform a "good" sinusoidal analysis of a two component chirp by specifying the parameter window-size in the function `chirpTracker(inputFile)` below. The estimation and tracking of the two varying frequencies should result in a mean error smaller than 2 Hz. The function returns the parameters used, true and estimated tracks of frequency components, and their associated time stamps.

In this part, you will use the sound `chirp-150-190-linear.wav`, which is a linear chirp with two sinusoids of 150 Hz and 190 Hz, whose frequency increases in two seconds to 1400 Hz and 1440 Hz respectively. Listen to the sound and use sms-tools GUI or sonic visualizer to see its spectrogram.

You will not write any additional code in this question, but modify the parameters of the code to obtain the best possible results. There are three functions we have written for you. Please go through each function and understand it, but do not modify any of it. 1. `chirpTracker()`: This is the main function. Uses `sineModel.py` for sinusoidal analysis of the input sound. It takes an input audio file and uses the function `sineModel.sineModelAnal()` to obtain the two frequency tracks (`fTrackEst`) in the chirp and computes the estimation error (`meanErr`) using the true frequency tracks obtained using `genTrueFreqTracks()`.

`chirpTracker()` calls the following two functions:

1. `genTimeStamps()`: Generates the time stamps at which the frequencies of sinusoids are estimated (one value per frame)
2. `genTrueFreqTracks()`: For the input sound `chirp-150-190-linear.wav`, the function generates the true frequency values, so that we can compare the true and the estimated frequency values.

We will use the sinusoidal model to analyse this sound and extract the two components. We will use the `sineModel.sineModelAnal()` function for analysis. The code for analysis is already provided below with some parameters we have fixed. For analysis, we will use a blackman window. Since we need only two frequency component estimates every frame, we set `maxnSines` = 2.

Choose the parameters window-size (`M`) and hop-size (`H`) such that the mean estimation error (`meanErr`) of both the frequency components is less than 2 Hz. There is a range of values of `M` and `H` for which this is true and all of those values will be considered correct answers. You can plot the estimated and true frequency tracks to visualize the accuracy of estimation.

We have written the function `chirpTracker()` for you and you just have to edit `M`. It is marked as `XX` and you can edit its value as needed.

As an example, choosing `M`=1023, the mean estimation error is `[13.677, 518.409]` Hz, which as you can see do not give us the desired estimation errors.

```
def chirpTracker(inputFile='../../sounds/chirp-150-190-linear.wav'):
    """
    Input:
      inputFile (string) = wav file including the path
    Output:
      M (int) = Window length
      H (int) = hop size in samples
      tStamps (numpy array) = A Kx1 numpy array of time stamps at which the
                              frequency components were estimated
      fTrackEst (numpy array) = A Kx2 numpy array of estimated frequency values,
                                one row per time frame, one column per component
      fTrackTrue (numpy array) = A Kx2 numpy array of true frequency values, one
                                 row per time frame, one column per component
      K is the number of frames
    """
    # Analysis parameters: Modify values of the parameters marked XX
    M = XX                                    # Window size in samples

    # More code follows
```

Part-3: Tracking sinusoids of different amplitudes (*2 points*)

Perform a "good" sinusoidal analysis of a signal with two sinusoidal components of different amplitudes by specifying a window type(`window`) and a peak picking threshold (`t`) in the function `mainlobeTracker()` below. The function should return the parameters used, true and estimated tracks of frequency components, and their associated time stamps.

We will consider a signal that has two sinusoidal components with a very large difference in their amplitude. We will use a synthetically generated signal with frequency components 440 Hz and 602 Hz, $s = \sin(2\,\pi\,440\,t) + 2 \times 10^{-3}\,\sin(2\,\pi\,602\,t)$. As you see the amplitude difference is large. You will use the sound `sines-440-602-hRange.wav`. Listen to the sound and use sms-tools GUI or sonic visualizer to see its spectrogram. Notice the difference in the amplitudes of its components.

You will not write any additional code in this question, but modify the parameters of the code to obtain the best possible results. There are three functions we have written for you. Please go through each function and understand it, but do not modify any of it.

1. `mainlobeTracker()`: This is the main function. It uses `sineModel.py` for sinusoidal analysis of the input sound. It takes an input audio file and uses `sineModel.sineModelAnal()` function, tracks the mainlobes of the two sinusoids to obtain the two frequency tracks (`fTrackEst`) in the signal. It also computes the estimation error (`meanErr`) in frequency using the true frequency tracks obtained using `genTrueFreqTracks()`. `mainlobeTracker()` calls the following two functions:

2. `genTimeStamps()`: Generates the time stamps at which the sinuosid frequencies are estimated (one value per audio frame)

3. `genTrueFreqTracks()`: For the input sound `sines-440-602-hRange.wav`, the function generates the true frequency values, so that we can compare the true and the estimated frequency values.

We will use sinusoidal Model to analyse this sound and extract the two components. We will use the `sineModel.sineModelAnal()` function for analysis. The code for analysis is already provided below with some parameters we have fixed. For analysis, we will use a window length (`M`) of 2047 samples, an FFT size (`N`) of 4096 samples and hop size (`H`) of 128 samples. For sine tracking, we set the minimum sine duration (`minSineDur`) to 0.02 seconds, `freqDevOffset` to 10 Hz and `freqDevSlope` to its default value of 0.001. Since we need only two frequency component estimates at every frame, we set `maxnSines` = 2.

Choose the parameters `window` and the peak picking threshold (`t`) such that the mean estimation error of both the frequency components is less than 2 Hz. There is a range of values of `M` and `t` for which this is true and all of those values will be considered correct answers. You can plot the estimated and true frequency tracks to visualize the accuracy of estimation. The output is the set of parameters you used: `window`, `t`, the time stamps and, estimated and the true frequency tracks. Note that choosing the wrong window might lead to tracking of one of the sidelobes of the high amplitude sinusoid instead of the mainlobe of the low amplitude sinusoid.

We have written the function `mainlobeTracker()` and you have to edit the `window` and `t` values. For the window, choose one of 'boxcar', 'hanning', 'hamming', 'blackman', or 'blackman-harris'. `t` is specified in negative dB. These two parameters are marked as `XX` and you can edit the values as needed.

As an example, choosing `window`='boxcar', `t`=-80, the mean estimation error is `[0.142, 129.462]` Hz.

```
def mainlobeTracker(inputFile='../../sounds/sines-440-602-hRange.wav'):
    """
    Input:
      inputFile (string): wav file including the path
    Output:
      window (string): The window type used for analysis
      t (float) = peak picking threshold (negative dB)
      tStamps (numpy array) = A Kx1 numpy array of time stamps at which the
                              frequency components were estimated
      fTrackEst = A Kx2 numpy array of estimated frequency values, one row
                              per time frame, one column per component
      fTrackTrue = A Kx2 numpy array of true frequency values, one row per
                   time frame, one column per component
    """
    # Analysis parameters: Modify values of the parameters marked XX
    window = XX                                  # Window type
    t = XX                                       # threshold (negative dB)

    # More code follows
```

Part-4: Tracking sinusoids using the phase spectrum (*3 points*)

Complete the function `selectFlatPhasePeak()` that selects a sinusoid peak based on the flatness of the phase spectrum around the frequency of the sinusoid. The function will be used for tracking sinusoids in audio signals, as an alternate method to tracking the mainlobe peaks of the magnitude spectrum.

In this question, you will implement an alternate way of tracking mainlobe of a sinusoid, using the phase spectrum. Recall that zero-phase windowing of sinusoid signal frame leads to a phase spectrum that is flat around the bins corresponding to frequency of the sinusoid. We will use this property of flatness of the phase spectrum as an alternative method to track the sinusoids. Note that this condition of flatness is satisfied only when the sinusoid is not time varying. For time-varying sinusoids, the condition fails.

We will consider a signal that has two sinusoid components and has a transient in the middle of the audio file. You will use the sound `sines-440-602-transient.wav`. Listen to the sound and use sms-tools GUI or sonic visualizer to see its spectrogram. Notice the transient that occurs in the middle of the sound file, where tracking using phase is likely to fail. We also recommend you to use the sms-tools GUI and DFT model to plot the spectrum at different parts of the signal to see if you indeed observe that the phase spectrum is flat around the sinusoid frequencies.

We will use sinusoidal model for analysis. We have modified the code for sinusoidal modeling from `sineModel.sineModelAnal()` to create a new function `sineModelAnalEnhanced()` which does a modified sine Tracking based on phase spectrum. Once we have the peaks estimated from the magnitude spectrum, we use a phase spectrum flatness measure around each peak location to select or reject the peak.

You will implement the function `selectFlatPhasePeak()` that checks for the flatness of the phase spectrum around the peak location. Given the peak location (`p`), the positive half of the phase spectrum (`pX`) and a threshold (`phaseDevThres`), you will compute the standard deviation of 5 samples of `pX` around the peak location (two samples either side and the sample at `p` itself) and compare it with the threshold. Based on the comparison, return a boolean variable `selectFlag`, which is `True` if the standard deviation is less than the threshold (and hence the phase is flat), else False (phase is not flat). We will use a small phase deviation threshold of 0.01 radian. In short, `selectFlatPhasePeak()` that returns `True` if the standard deviation of five samples of the phase spectrum `pX` around the input index `p` is less than the given threshold, else `False`.

Read through the function `sineModelAnalEnhanced()` and understand it thoroughly before implementing `selectFlatPhasePeak()` function. The function `sineModelAnalEnhanced()` takes an input audio file and uses phase based sinusoid tracking to obtain the two frequency tracks (`fTrackEst`) in the signal. Since we need only two sinusoids every frame, we only consider the frames where we get two selected peaks, and ignore the other frames. You can plot the estimated and true frequency tracks to visualize the accuracy of estimation (code provided).

Test case 1: With `pX = np.array([1.0, 1.2, 1.3, 1.4, 0.9, 0.8, 0.7, 0.6, 0.7, 0.8])`, `p` = 3, and `phaseDevThres` = 0.25, the function `selectFlatPhasePeak()` should return `True`.

Test case 2: With `pX = np.array([1.0, 1.2, 1.3, 1.4, 0.9, 0.8, 0.7, 0.6, 0.7, 0.8])`, `p` = 3, and `phaseDevThres` = 0.1, the function `selectFlatPhasePeak()` should return `False`.

Test case 3: With `pX = np.array([2.39, 2.40, 2.40, 2.41, 3.37, 2.45, 2.46, 2.46, 2.29, 1.85, 2.34, 2.18, 2.93, 2.10, 3.39, 2.41, 2.41, 2.40, 2.40, 2.40, 1.46, 0.23, 0.98, 0.41, 0.37, 0.40, 0.41, 0.87, 0.51, 0.67])`, `p` = 17, and `phaseDevThres` = 0.01, the function `selectFlatPhasePeak()` should return `True`.

As an example, when you run your code using `inputFile= '../../sounds/sines_440-602_transient.wav'`, if you have implemented `selectFlatPhasePeak()` function correctly, you will see two sinusoid tracks in the beginning and end of the audio file, while there are no tracks in the middle of the audio file. This is due to the transients present in the middle of the audio file, where phase based tracking of sinusoids fails.

```
def selectFlatPhasePeak(pX, p, phaseDevThres):
    """
    Function to select a peak index based on phase flatness measure.
    Input:
      pX (numpy array) = The phase spectrum of the frame
      p (positive integer) = The index of peak in the magnitude spectrum
      phaseDevThres (float) = The threshold value to measure flatness of phase
    Output:
    selectFlag (Boolean) = True, if the peak at index p is a mainlobe,
                           False otherwise
    """
    ### Your code here
```

Part-5: Sinusoidal modeling of a multicomponent signal (optional)

Perform a sinusoidal analysis of a complex synthetic signal, exploring the different parameters of the model. Use the sound `multiSines.wav` and post your comments on the forum thread specific for this topic.

This is an open question without a single specific answer. We will use the sound `multiSines.wav`, which is a synthetic audio signal with sharp attacks, close frequency components with a wide range of amplitudes, and time varying chirps with frequency tracks that cross over. All these characteritics make this signal difficult to analyze with `sineModel`. Listen to the sound and use sms-tools GUI or sonic visualizer to see its spectrogram.

We have written a basic code for sinusoidal analysis, you are free to modify the code. The function saves the output sound which you can listen to and visualize.

You will get credit for this question by just attempting the question and submitting it. Share your thoughts on the forum thread (`https://class.coursera.org/audio-002/forum/list?forum_id=10022`) on the parameter choices you made, tradeoffs that you encountered, and quality of the reconstruction you achieved.

```
def exploreSineModel(inputFile='../../sounds/multiSines.wav'):
    """
    Input:
      inputFile (string) = wav file including the path
    Output:
      return True
      Discuss on the forum!
    """
    # More code follows
```

Grading

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