A6: Harmonic Model

Audio Signal Processing for Music Applications

Introduction

This assignment on Harmonic model will help you better understand fundamental frequency (f_0) estimation with several sound examples with harmonic content. You will see a practical application of f_0 estimation in segmenting a melody. You can optionally also improve the f_0 estimation algorithm in sms-tools. There are four parts to this assignment: 1) Estimate fundamental frequency in an polyphonic audio file 2) Segmentation of stable note regions in an audio signal, 3) Compute amount of inharmonicity present in a sound 4) Improving the implementation of the two way mismatch f_0 estimation algorithm (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.

Relevant Concepts

Harmonic model parameters: Harmonic model is used for the analysis of harmonic sounds. The file `sms-tools/software/models/harmonicModel.py` provides the code for Harmonic model analysis and synthesis. The key component of the harmonic model is the estimation of the fundamental frequency (f_0) and its harmonics. Apart from the parameters such as the window, FFT size and the peak picking threshold, we have a few additional parameters used by the harmonic model.

- nH: Maximum number of harmonics. This is the number of harmonics estimated and returned by harmonicModelAnal().
- maxf0: Maximum f_0 frequency in Hz.
- minf0: Minimum f_0 frequency in Hz. The estimated f_0 will not be less than minf0. Setting the maxf0 and minf0 accurately help to narrow down the f_0 candidates used by TWM algorithm and lead to better f_0 estimation.
- f0et: Error threshold in the f_0 detection. This is the maximum error allowed in the TWM algorithm. If the TWM mismatch error is larger than f0et, no f_0 is detected and the TWM algorithm returns $f_0 = 0$ for the frame.
- harmDevSlope: Slope of harmonic deviation allowed in the estimated harmonic frequency, compared to a perfect harmonic frequency. This is used to compute the threshold to generate the harmonics.

Melody representation: For computational analysis, melody is represented typically by the pitch (fundamental frequency). The fundamental frequency (f_0) is usually estimated in Hz but for a musically meaningful representation, we convert f_0 from Hz to Cent. Cent is a logarithmic scale computed as

$$f_{0,\text{Cents}} = 1200 \log_2 \left(\frac{f_{0,\text{Hz}}}{55.0} \right) \qquad (1)$$

Assuming a tuning frequency of A4 = 440 Hz, the reference frequency used in the Cent scale is the frequency of the note A1 = 55Hz, i.e. 55Hz = 0 Cent.

Segmentation and transcription: Audio segmentation and transcription are two important music information retrieval tasks. Audio segmentation aims to segment the audio into musically meaningful entities. Music Transcription aims to automatically obtain a score-like representation from a music audio piece. Segmentation is often a preprocessing step in transcription. Both these tasks have several different approaches that have been explored.

In this assignment, we will consider a simple approach to note level segmentation of melodies. Given the audio file, we first estimate the pitch (fundamental frequency f_0) for the whole file. We then segment the pitch contour into stable regions. The stable regions most likely correspond to notes of the melody. We then have the start and end time stamps of each note played in the melody. A limitation of this approach to segmentation is that it might not work for notes with a vibrato.

You will only implement the segmentation as described above. However, additionally for each segment, given a tuning frequency (say A = 440 Hz), you can obtain the notes played in the melody by quantizing the pitch in each segment to a note - a note level transcription of the melody.

Inharmonicity: In music, inharmonicity is the degree to which the frequencies of the partials depart from integer multiples of the fundamental frequency (harmonic series). An ideal, homogeneous, infinitesimally thin or infinitely flexible string or column of air has exactly harmonic modes of vibration. However, in any real musical instrument, the resonant body that produces the music tone - typically a string, wire, or column of air—deviates from this ideal and has some small or large amount of inharmonicity. You can read more about inharmonicity at `http://en.wikipedia.org/wiki/Inharmonicity`.

A typical example of an instrument that exhibits inharmonicity is the piano. For the piano, several models have been proposed to obtain the partials of the piano, which can be used to estimate the inharmonicity. One of the models proposed by Fletcher (Harvey Fletcher, "Normal Vibration Frequencies of a Stiff Piano String", J. Acoust. Soc. Am. 36, 203 (1964); `http://dx.doi.org/10.1121/1.1918933`) is shown in Equation 2, where f_r is the frequency of the r^{th} partial, f_0 is the fundamental frequency and B is the inharmonicity coefficient.

$$f_r = r f_0 \sqrt{(1 + Br^2)} \tag{2}$$

In this assignment, you will measure the inharmonicity in a piano note using the harmonic model. With the estimates of the fundamental frequency f_0 and the harmonics $\mathbf{f}_{est}$ for a frame l, we can obtain a measure of inharmonicity as,

$$I[l] = \frac{1}{R}\sum_{r=1}^{R}\left(\frac{|f_{\text{est}}^{r}[l] - r\, f_0[l]|}{r}\right) \tag{3}$$

where R is the number of harmonics (the number of harmonics being used to compute inharmonicity), $f_0[l]$ is the fundamental frequency estimated at the frame l and $f_{\text{est}}^{r}[l]$ is the estimated frequency of the r^{th} harmonic at the frame. Note that the first harmonic is the fundamental.

We can then compute the mean inharmonicity in a specific time region between the frame indexes l_1 and l_2 as,

$$I_{\text{mean}} = \frac{1}{l_2 - l_1 + 1}\sum_{l=l_1}^{l_2} I[l] \tag{4}$$

TWM algorithm candidate selection: The two way mismatch algorithm implemented in sms-tools needs a set of f_0 candidates to start with. An easy choice of candidates are the peaks of the magnitude spectrum within a specific range of frequencies. However, this way of choosing f_0 candidates fails when there is no peak corresponding to the true f_0 value. The generation of f_0 candidates can be done better by also including the sub-harmonics of the peak frequencies as f_0 candidates.

Searching numpy arrays: Numpy provides an efficient way to search for a specific element(s) of an array that satisfy a given condition. You can use `np.where()` in such cases. e.g. Given a numpy array `a = array([ 0.9193727 , 0.6359579 , 0.8335968 , 0.20568055, 0.13874869])` and

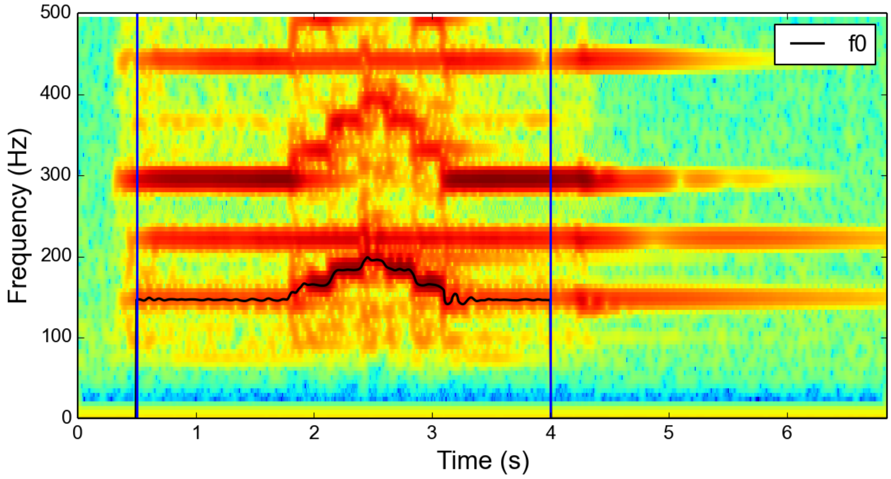

Figure 1: f_0 in the time segment 0.5 and 4 seconds for cello-double-2.wav

you want to extract the indexes of elements less than 0.5, you can use `np.where(a<0.5)[0]`. The function returns `array([3, 4])` corresponding the indexes of the elements in `a` less than 0.5.

Part-1: Estimate fundamental frequency in an polyphonic audio file (*3 points*)

Set the analysis parameters used within the function `estimateF0()` to obtain a good estimate of the fundamental frequency (f_0) corresponding to one melody within a complex audio signal. The signal is a cello recording `cello-double-2.wav`, in which two strings are played simultaneously. One string plays a constant drone while the other string plays a simple melody. You have to choose the analysis parameter values such that only the f_0 frequency of the simple melody is tracked.

The input argument to the function is the wav file name including the path (`inputFile`). The function returns a numpy array of the f_0 frequency values for each audio frame. For this question we take hopSize (`H`) = 256 samples.

`estimateF0()` calls `f0Detection()` function of the `harmonicModel.py`, which uses the two way mismatch algorithm for f_0 estimation.

`estimateF0()` also plots the f_0 contour on top of the spectrogram of the audio signal for you to visually analyse the performance of your chosen values for the analysis parameters. In this question we will only focus on the time segment between 0.5 and 4 seconds. So, your analysis parameter values should produce a good f_0 contour in this time region.

In addition to plotting the f_0 contour on the spectrogram, this function also synthesizes the f_0 contour (10 harmonics). You can also evaluate the performance of your chosen analysis parameter values by listening to this synthesized wav file named `synthF0Contour.wav`

Since there can be numerous combinations of the optimal analysis parameter values, the evaluation is done solely on the basis of the output f_0 sequence. Note that only the segment of the f_0 contour between time 0.5 to 4 seconds is used to evaluate the performance of f_0 estimation.

Your assignment will be tested on `inputFile` = '`../../sounds/cello-double-2.wav`'. So choose the analysis parameters using which the function estimates the f_0 frequency contour corresponding to the string playing simple melody and not the drone. There is no separate test case for this question. You can keep working with the wav file mentioned above and when you think the performance is satisfactory you can submit the assignment. The plots can help you achieve a good performance. Your aim should be to get a f_0 contour as close to the one shown in Figure 1.

Be cautious while choosing the window size. Window size should be large enough to resolve the spectral peaks and small enough to preserve the note transitions. Very large window sizes may smear the f_0 contour at note transitions.

Depending on the parameters you choose and the capabilities of the hardware you use, the function might take a while to run (even half a minute in some cases). For this part of the assignment please refrain from posting your analysis parameters on the discussion forum.

```
def estimateF0(inputFile = '../../sounds/cello-double-2.wav'):
    """
    Function to estimate fundamental frequency (f0) in an audio signal.
    This function also plots the f0 contour on the spectrogram and synthesize
    the f0 contour.
    Input:
        inputFile (string): wav file including the path
    Output:
        f0 (numpy array): array of the estimated fundamental frequency (f0) values
    """
    ### Change these analysis parameter values marked as XX
    window = XX
    M = XX
    N = XX
    f0et = XX
    t = XX
    minf0 = XX
    maxf0 = XX

    # Additional code follows...
```

Part-2: Segmentation of stable note regions in an audio signal (*4 points*)

Complete the function `segmentStableNotesRegions()` that identifies the stable regions of notes in a specific monophonic audio signal. The function returns an array of segments where each segment contains the starting and the ending frame index of a stable note.

The input argument to the function are the wav file name including the path (`inputFile`), threshold to be used for deciding stable notes (`stdThsld`), minimum allowed duration of a stable note (`minNoteDur`), number of samples to be considered for computing standard deviation (`winStable`), analysis window (`window`), window size (`M`), FFT size (`N`), hop size (`H`), error threshold used in the f_0 detection (`f0et`), magnitude threshold for spectral peak picking (`t`), minimum allowed f_0 (`minf0`) and maximum allowed f_0 (`maxf0`). The function returns a numpy array of shape (k,2), where `k` is the total number of detected segments and the two columns in each row contains the starting and the ending frame indexes of a stable note segment. The segments must be returned in the increasing order of their start times.

In order to facilitate the assignment we have configured the input parameters to work with a particular sound `'../../sounds/sax-phrase-short.wav'`. The code and parameters to estimate the fundamental frequency is completed. Thus you start from an f_0 curve obtained using the `f0Detection()` function and you will use that to obtain the note segments.

All the steps to be implemented in order to solve this question are indicated in `segmentStableNotesRegions()` as comments. Depending on the analysis parameters and the capabilities of the hardware you use, the function might take a while to run (even half a minute in some cases). These are the steps:

1. In order to make the processing musically relevant, the f_0 values should be converted first from Hertz to Cents, which is a logarithmic scale.

2. At each time frame (for each f_0 value) you should compute the standard deviation of the past winStable number of f_0 samples (including the f_0 sample at the current audio frame).

3. You should then apply a deviation threshold, `stdThsld`, to determine if the current frame belongs to a stable note region or not. Since we are interested in the stable note regions, the

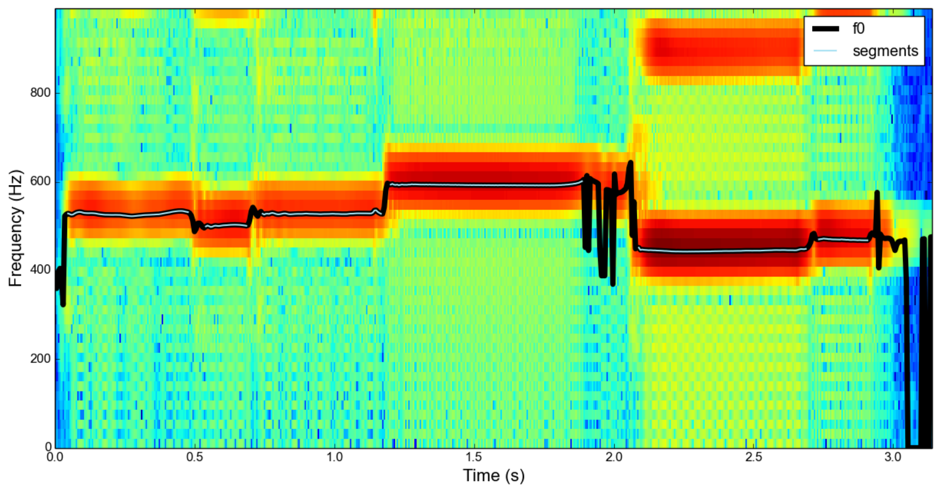

Figure 2: Note segments obtained for the default parameters on sax-phrase-short.wav

standard deviation of the previous `winStable` number of f_0 samples (including the current sample) should be less than `stdThsld` i.e. use the current sample and `winStable-1` previous samples. Ignore the first `winStable-1` samples in this computation.

4. All the consecutive frames belonging to the stable note regions should be grouped together into segments. For example, if the indexes of the frames corresponding to the stable note regions are 3,4,5,6,12,13,14, we get two segments, first 3-6 and second 12-14.

5. After grouping frame indexes into segments filter/remove the segments which are smaller in duration than `minNoteDur`. Return the segments in the increasing order of their start frame index.

Test case 1: Using `inputFile='../../sounds/cello-phrase.wav', stdThsld=10, minNoteDur=0.1, winStable = 3, window='hamming', M=1025, N=2048, H=256, f0et=5.0, t=-100, minf0=310, maxf0=650`, the function `segmentStableNotesRegions()` should return 9 segments. Please use `loadTestcases.load()` to check the expected segment indexes in the output.
Test case 2: Using `inputFile='../../sounds/cello-phrase.wav', stdThsld=20, minNoteDur=0.5, winStable = 3, window='hamming', M=1025, N=2048, H=256, f0et=5.0, t=-100, minf0=310, maxf0=650`, the function `segmentStableNotesRegions()` should return 6 segments. Please use `loadTestcases.load()` to check the expected segment indexes in the output.
Test case 3: Using `inputFile='../../sounds/sax-phrase-short.wav', stdThsld=5, minNoteDur=0.6, winStable = 3, window='hamming', M=1025, N=2048, H=256, f0et=5.0, t=-100, minf0=310, maxf0=650`, the function `segmentStableNotesRegions()` should return just one segment. Please use `loadTestcases.load()` to check the expected segment indexes in the output.

We also provide the function `plotSpectogramF0Segments()` to plot the f_0 contour and the detected segments on the top of the spectrogram of the audio signal in order to visually analyse the outcome of your function, see Figure 2 for example.

```
def segmentStableNotesRegions(inputFile = '../../sounds/sax-phrase-short.wav',
```

```
                stdThsld=10.0, minNoteDur=0.1, winStable = 3, window='hamming',
                M=1024, N=2048, H=256, f0et=5.0, t=-100, minf0=310, maxf0=650):
    """
    Function to segment the stable note regions in an audio signal
    Input:
      inputFile (string): wav file including the path
      stdThsld (float): threshold for detecting stable regions in the f0 contour
                        (in cents)
      minNoteDur (float): minimum allowed segment length (note duration)
      winStable (integer): number of samples used for computing standard deviation
      window (string): analysis window
      M (integer): window size used for computing f0 contour
      N (integer): FFT size used for computing f0 contour
      H (integer): Hop size used for computing f0 contour
      f0et (float): error threshold used for the f0 computation
      t (float): magnitude threshold in dB used in spectral peak picking
      minf0 (float): minimum fundamental frequency in Hz
      maxf0 (float): maximum fundamental frequency in Hz
    Output:
      segments (np.ndarray): Numpy array containing starting and ending frame
      indexes of every segment.
    """
    fs, x = UF.wavread(inputFile)                               #reading inputFile
    w  = get_window(window, M)                                  #analysis window
    f0 = HM.f0Detection(x, fs, w, N, H, t, minf0, maxf0, f0et)  #estimating F0

    ### Your code here

    # 1. convert f0 values from Hz to Cents (as described in pdf document)

    #2. create an array containing standard deviation of last winStable samples

    #3. apply threshold on standard deviation values to find indexes of the stable
        points in melody

    #4. create segments of continuous stable points such that consecutive stable
        points belong to same segment

    #5. apply segment filtering, i.e. remove segments with are < minNoteDur in length

    # Plot spectrogram and F0 if needed
    # plotSpectogramF0Segments(x, fs, w, N, H, f0, segments)

    # return segments
```

Part-3: Compute amount of inharmonicity present in a sound (*3 points*)

Complete the function `estimateInharmonicity()` that measures the amount of inharmonicity present in a pitched/harmonic sound. The function should measure the mean inharmonicity in the sound over the time interval `t1` to `t2`.

The input argument to the function are the wav file name including the path (`inputFile`), start (`t1`) and end time (`t2`) of the audio segment to compute inharmonicity, analysis window (`window`), window size (`M`), FFT size (`N`), hop size (`H`), error threshold used in the f_0 detection (`f0et`), magnitude threshold for spectral peak picking (`t`), minimum allowed f_0 (`minf0`), maximum allowed f_0 (`maxf0`) and number of harmonics to be considered in the computation of inharmonicity (`nH`). The function returns a single numpy float, which is the mean inharmonicity over time `t1` to `t2`.

A brief description of the method to compute inharmonicity is provided in the Relevant Concepts section of the assignment pdf. The steps to be done are:

1. Use `harmonicModelAnal` function in `harmonicModel` module for computing the harmonic frequencies and their magnitudes at each audio frame. The first harmonic is the fundamental frequency. For `harmonicModelAnal` use `harmDevSlope`=0.01, `minSineDur`=0.0. Use `harmonicModelAnal` to estimate harmonic frequencies and magnitudes for the entire audio signal.

2. For the computation of the inharmonicity choose the frames that are between the time interval `t1` and `t2`. Do not slice the audio signal between the time interval `t1` and `t2` before estimating harmonic frequencies.

3. Use the formula given in the Relevant section to compute the inharmonicity measure for the given interval. Note that for some frames some of the harmonics might not be detected due to their low energy. For handling such cases use only the detected harmonics to compute the inharmonicity measure. All the detected harmonics have a non zero frequency.

In this question we will work with a piano sound (`'../../sounds/piano.wav'`), a typical example of an instrument that exhibits inharmonicity (`http://en.wikipedia.org/wiki/Piano_acoustics#Inharmonicity_and_piano_size`).

Test case 1: If you run your code with `inputFile = '../../sounds/piano.wav', t1=0.2, t2=0.4, window='hamming', M=2047, N=2048, H=128, f0et=5.0, t=-90, minf0=130, maxf0=180, nH = 25`, the returned output should be 1.4543.

Test case 2: If you run your code with `inputFile = '../../sounds/piano.wav', t1=2.3, t2=2.55, window='hamming', M=2047, N=2048, H=128, f0et=5.0, t=-90, minf0=230, maxf0=290, nH = 15`, the returned output should be 1.4874.

Test case 3: If you run your code with `inputFile = '../../sounds/piano.wav', t1=2.55, t2=2.8, window='hamming', M=2047, N=2048, H=128, f0et=5.0, t=-90, minf0=230, maxf0=290, nH = 5`, the returned output should be 0.1748.

Optional/Additional tasks: An interesting task would be to compare the inharmonicities present in the sounds of different instruments.

```
def estimateInharmonicity(inputFile = '../../sounds/piano.wav', t1=0.1, t2=0.5,
                window='hamming', M=2048, N=2048, H=128, f0et=5.0, t=-90,
                minf0=130, maxf0=180, nH = 10):
    """
    Function to estimate the extent of inharmonicity present in a sound
    Input:
        inputFile (string): wav file including the path
        t1 (float): start time of the segment considered for computing
                    inharmonicity
        t2 (float): end time of the segment considered for computing
                    inharmonicity
        window (string): analysis window
        M (integer): window size used for computing f0 contour
        N (integer): FFT size used for computing f0 contour
        H (integer): Hop size used for computing f0 contour
        f0et (float): error threshold used for the f0 computation
        t (float): magnitude threshold in dB used in spectral peak picking
        minf0 (float): minimum fundamental frequency in Hz
        maxf0 (float): maximum fundamental frequency in Hz
        nH (integer): number of integers considered for computing inharmonicity
    Output:
```

```
    meanInharm (float or np.float): mean inharmonicity over all the frames
    between the time interval t1 and t2.
"""
### Your code here

# 0. Read the audio file and obtain an analysis window

# 1. Use harmonic model to compute the harmonic frequencies and magnitudes

# 2. Extract the time segment in which you need to compute the inharmonicity.

# 3. Compute the mean inharmonicity of the segment
```

Part-4: Improving the implementation of the two way mismatch f0 estimation algorithm (*Optional*)

Improve the performance of the current implementation of the two way mismatch algorithm in sms-tools used for fundamental frequency estimation. This is an optional open question and will not contribute towards the final grade. There is no definite answer for this question. Its main purpose is to understand the limitations of the current implementations of the TWM algorithm and to come up with some community driven solutions based on collective thinking.

In this question you will directly modify the core functions that implement the TWM algorithm in sms-tools. To assist you with this task, we have copied all the needed functions into this python file. Hence, you just need to modify the functions in this file and not anywhere else.

Estimating fundamental frequency from an audio signal is still a challenging and unsolved problem to a large extent. By this time you might have also realized that many times the performance of the TWM f_0 estimation algorithm falls short of the expectations. There can be a systematic explanation for the scenarios where TWM fails for specific categories or characteristics of the sounds. Some of the known scenarios where the current implementation of the TWM algorithm fails to estimate a correct fundamental frequency are:

1. *Missing fundamental frequency:* For many sounds the fundamental frequency component is very low and therefore during the spectral peak picking step we do not obtain any peak corresponding to the f_0. Since the TWM algorithm implemented in sms-tools considers only the detected spectral peaks as the f_0 candidates, we do not get any candidate corresponding to the f_0. This causes f_0 estimation to fail. For example, such a scenario is encountered in low pitched vocal sounds.
2. *Pseudo-harmonicity in the sound:* Many instruments such as piano exhibit some deviation from perfect harmonicity wherein their harmonic partials are not perfectly located at integral multiples of the fundamental frequency. Since the TWM algorithm computes error function assuming that the harmonic locations are at integral multiples, its performance is poorer when such deviations exist.

In this question we propose to work on these two scenarios. Go to freesound and download sound examples of low pitched vocal sounds and of piano. Run current implementation of TMW to identify the limitations and propose improvements to the code in order to obtain better f_0 estimation for those two particular scenarios.

The core TWM algorithm is implemented in the function `TWM_p()`, which takes in an array of f_0 candidates and detect the candidate that has the lowest error. `TWM_p()` is called by `f0Twm()`, which generates f_0 candidates. This function also implements a memory based prunning of the f_0 candidates. If the f_0 contour is found to be stable (no drastic transitions across frames) then only the f_0 candidates close to the stable f_0 value are retained. `f0Twm()` is called for every audio frame by `f0Detection()`.

You can use `computeAndPlotF0()`, which calls `f0Detection()` for estimating f_0 for every audio frame. In addition, it also plots the f_0 contour on the top of the spectrogram. If you set plot=1, it shows the plot, plot=2 saves the plot as can be seen in the code.

Once you implement your proposed enhancement, discuss and share your ideas on the discussion forum assigned for A6Part4 - `https://class.coursera.org/audio-002/forum/list?forum_id=10026`. Along with the text you should include 2 plots showing the f_0 contour before and after your changes. Use the same values of the analysis parameters while showing the improvement in the performance. in the discussion, also include a link to the sound in freesound.

TIP: An identified limitation of the current implementation for the case of low vocal sounds is that it can only find f_0 if there is a peak present in the magnitude spectrum. A possible improvement is to generate additional f_0 candidates from the identified peaks. Another identified limitation for the case of piano sounds is the assumption of perfect harmonicity. For these sounds you can think of modifying the generation of the ideal harmonic series that is computed in the code, incorporating the typical deviation from harmonicity encountered in piano sounds.

NOTE: Before you start making changes in the TWM implementation make sure you have reached the best possible performance that can be achieved by tuning the analysis parameters. If the analysis parameters are inappropriately set, it is not completely meaningful to just improve the TWM implementation.

To maintain the integrity of the sms-tools package for future assignments, please make changes only to the functions in this file and not the other files in sms-tools.

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.