Assignment 1: Dealing with Time-Domain Audio

CS 4347: Sound and Music Computing

due Wednesday 4 February 2015, 11:59 pm

0. This assignment will make use of the "Music & Speech" dataset of Marsyas:
   - `http://opihi.cs.uvic.ca/sound/music_speech.tar.gz`
   - This dataset has two copies of each song; delete the `music/` and `speech/` directories and use the files in `music-wav/` and `speech-wav/` directories. There are 64 music and 64 speech files; each file is 30 seconds of audio stored as 16-bit signed integers, 22050 Hz.
   - Ground truth data for this dataset:
     `http://www.comp.nus.edu.sg/~duanzy/music_speech.mf`
     Format of the file is `filename` `\t` (tab) `label` `\n` (newline), one song per line:
     ```
     filename1\tlabel1\n
     filename2\tlabel2\n
     ...
     filename128\tlabel128\n
     ```
     The `label` will be `music` or `speech`.

1. Write a python program that will:
   - Read the ground-truth `music_speech.mf` file
   - Load each wav file and convert the data to floats by dividing the samples by 32768.0. Hint: use `scipy.io.wavfile.read()`
   - Calculate 4 features for each file according to the given formulae. Use only one vector per file (don't use multiple buffers for each file).
     Given $X = \{x_0, x_1, x_2, \ldots x_{N-1}\}$,
     (a) Root-mean-squared (RMS)
     $$X_{\text{RMS}} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1} x_i^2}$$
     (b) Peak-to-average-ratio (PAR)
     $$X_{\text{PAR}} = \frac{\arg\max_i |x_i|}{X_{\text{RMS}}}$$
     (c) Zero crossings (ZCR)
     $$X_{\text{ZCR}} = \frac{1}{N-1}\sum_{i=1}^{N-1}\begin{cases}1 & \text{if } (x_i \cdot x_{i-1}) < 0\\ 0 & \text{else}\end{cases}$$
     (d) Median absolute deviation (MAD)
     $$X_{\text{MAD}} = \operatorname*{median}_i\left(\left|x_i - \operatorname*{median}_j(x_j)\right|\right)$$
     Hint: numpy has a built-in `numpy.median()` function!

- Output the data to a comma separated value (CSV) text file in the format:

  ```
  filename1,RMS1,PAR1,ZCR1,MAD1\n
  filename2,RMS2,PAR2,ZCR2,MAD2\n
  ...
  filename128,RMS128,PAR128,AC128,MAD128\n
  ```

  Concretely, the beginning and ending of the file should be:

  ```
  music_wav/bagpipe.wav,0.063492,8.149929,0.191660,0.031769
  music_wav/ballad.wav,0.029699,7.320233,0.039395,0.012695
  ...
  speech_wav/voice.wav,0.070688,4.163124,0.082435,0.031982
  ```

  To pass our automated grading system, the format of your file must match this exactly. The order of filenames must match the order in the `music_speech.mf` file.

2. Upload your CSV file to:

   ```
   http://cs4347.smcnus.org
   ```

   This will automatically grade the values you calculated. If any mistake is found, please check your program and resubmit – you are welcome to submit as many versions as you wish before the submission deadline.

   Submit a zip file containing your program's source code (as a `.py` file), the CSV file, and an optional README.txt file to the same website.

   - You may use anything in the python standard library, numpy (including pylab / matplotlib), and scipy libraries. No other libraries are permitted.

If you are familiar with python and understood the lecture, this should take about 1 hour.
Grading scheme:

- **3/6 marks**: correct CSV file (automatically graded by computer).
- **3/6 marks**: readable source code (good variable names, clean functions, comments when needed).

Assignment 2: Machine Learning with Time-Domain Audio

CS 4347: Sound and Music Computing

due Wednesday 11 February 2015, 11:59 pm

0. This assignment will use the same "music / speech" dataset that we used in asn 1.

1. Make a copy of your python program from assignment 1 and begin modifying it. Your program should begin by doing the same steps as assignment 1, namely:
   - Read the ground-truth `music_speech.mf` file
   - Load each wav file and convert the data to floats by dividing the samples by 32768.0.
   - Calculate 4 features for each file: RMS, PAR, ZCR, MAD. Use only one vector per file (don't use multiple buffers for each file).
   - (optional) You may wish to improve your program's speed. Well-written python+numpy can calculate all these features on the entire dataset in less than a minute on a modern desktop computer (mine completes assignment 1 in 9.97 seconds).
     The key is to avoid any loops for calculating features. Everything can be done with built-in numpy commands. Some features can be calculated with a single line, for example:

     ```
     def rms(time_domain_data):
         return numpy.sqrt(numpy.mean(numpy.square(time_domain_data)))
     ```

   Your program should output different data, however:

   - Output the data to an `ARFF` file with this format:

     ```
     @RELATION music_speech
     @ATTRIBUTE RMS NUMERIC
     @ATTRIBUTE PAR NUMERIC
     @ATTRIBUTE ZCR NUMERIC
     @ATTRIBUTE MAD NUMERIC
     @ATTRIBUTE class {music,speech}

     @DATA
     RMS1,PAR1,ZCR1,MAD1,music
     RMS2,PAR2,ZCR2,MAD2,music
     ...
     RMS128,PAR128,ZCR128,MAD128,speech
     ```

     To pass our automated grading system, the format of your file must match this exactly. The order of filenames must match the order in the `music_speech.mf` file. Note that the the ARFF format does not include the filenames, merely the label.
   - Output the follow pairs of features as a graph (use `pylab.savefig` to save as PNG files):
     - ZCR (x-axis) and PAR (y-axis)
     - MAD and RMS

     Each graph must have axis labels, clearly distinguishable markers & colors for music and speech, and a legend.

Hint: don't try to do the plotting in the main loop of your program! Intead, fill two matrices of features for music and speech:

```
features_music  = numpy.zeros((num_audio_files, num_features))
features_speech = numpy.zeros((num_audio_files, num_features))
...
for i in range(num_audio_files):
    ...
    if label == "music":
        features_music[i]  = features
    else:
        features_speech[i] = features
```

These can then be plotted very easily by extracting the relevant columns (e.g., column 2 is ZCR, column 1 is PAR):

```
pylab.plot(features_music[:,2], features_music[:,1])
pylab.plot(features_speech[:,2], features_speech[:,1])
```

The resulting plots do not clearly distinguish between these sets of features, so make sure you improve the plots before submitting the assignment. This hint gives you the beginning of the plotting task, not the whole answer!

2. Load the ARFF file in Weka: `http://www.cs.waikato.ac.nz/ml/weka/`

   Classify the data with `trees.J48` with 10-fold cross-validation and save the results. In detail, load weka, go to the Explorer, and open your ARFF file. Click on the "Classify" tab, then click "Choose" under "Classifier". Select the `trees.J48` option, then press "Start". Right-click on the value in the "Result list" and save the result buffer (the text containing the Classifier model, Summary, and Confusion Matrix).

   In addition to classifying with `trees.J48`, select 2 other classification algorithms and record their output. Write a file called `classifications.txt` which compares the results of these three algorithms. The file should begin with 1-3 sentences stating which was the best algorithm and summarizing its performance in comparison to the other two algorithms. This should be followed by the Weka results ordered from best to worst.

3. Upload your ARFF file to:

   `http://cs4347.smcnus.org`

   This will automatically grade the values you calculated. If any mistake is found, please check your program and resubmit – you are welcome to submit as many versions as you wish before the submission deadline.

   Submit a zip file containing your program's source code (as a `.py` or `ipynb` file), the ARFF file, the two PNG files, the `classifications.txt` file comparing different machine learning algorithms, and an optional README.txt file to the same website.

   - You may use anything in the python standard library, numpy (including pylab / matplotlib), and scipy libraries. No other libraries are permitted.

If you are familiar with python, this should take about 1 hour.
Grading scheme:

- **1/6 marks**: correct ARFF file (automatically graded by computer).
- **2/6 marks**: readable source code (good variable names, clean functions, comments when needed).
- **2/6 marks**: visual quality of plots (axis labels, markers, colors, legend)
- **1/6 marks**: Discussion of weka output

Assignment 3: Machine Learning and Buffer-based Time-Domain Audio

CS 4347: Sound and Music Computing

due Wednesday 18 February 2015, 11:59 pm

0. This assignment will use the same "music / speech" dataset that we used in asn 1.

1. Make a copy of your python program from assignment 2 and add 1 new feature. Your program should still calculate features based on the entire audio file at once, and output an ARFF file, but it does not need to output any PNG files.

   Mean Absolute Deviation (MEAN-AD):

$$X_{\text{MEAN\_AD}} = \frac{1}{N} \sum_{i=0}^{N-1} \left(\left| x_i - \frac{1}{N} \sum_{j=0}^{N-1} x_j \right| \right)$$

2. Make a copy of the previous program. Modify it such that it will:

   - Read the collection file
   - Load each wav file (divide by 32768.0) and split the data into buffers of length 1024 with 50% overlap (or a hopsize of 512) Only include complete buffers; if the final buffer has 1020 samples, omit that buffer.

     Hint: the starting and ending indices for the first few buffers are:

Buffer number	start index	end index
		(not included in array)
0	0	1024
1	512	1536
2	1024	2048
...		

     I recommend that you use the numpy "array slice" feature:

```
for i in range(num_buffers):
    start = ...
    end = ...
    buffer_data = whole_file_data[start:end]
```

     This should give you 1290 buffers of length 1024. If you are comfortable with linear algebra and numpy, the program will run significantly faster if you store this as a single 1290x1024 matrix and avoid any loops during the feature calculations.
   - Calculate 5 features for each buffer: RMS, PAR, ZCR[1], MAD, MEAN_AD. This means that $N = 1024$ in the equations, instead of the previous $N = 661500$. You should end up with 1290 feature vectors of length 5, or a single 1290x5 matrix.

[1] You may be concerned about the boundaries between buffers when calculating the ZCR: "if buffer i ends with a negative value and buffer $i + 1$ begins with a positive value, what happens to that zero-crossing?". The answer is that for this assignment, we strictly use the formula from assignment 1. This means that zero-crossings that are exactly between buffers are never counted. This is not an ideal solution since it means that the "total" number of zero-crossings can change based on the buffer size. However, this vastly simplifies the programming, so we adopt this solution. Some well-known audio analysis tools such as Marsyas do the same thing.

- After calculating the features for each buffer, calculate the mean and uncorrected sample standard deviation for each feature over all buffers for each file. You should end up with a single feature vector of length 10, or a 1x10 matrix for each file.
- Output the data to a new ARFF file (with a different filename from the whole-song ARFF file) with the same format as before, but the header should be:

  ```
  @RELATION music_speech
  @ATTRIBUTE RMS_MEAN NUMERIC
  @ATTRIBUTE PAR_MEAN NUMERIC
  @ATTRIBUTE ZCR_MEAN NUMERIC
  @ATTRIBUTE MAD_MEAN NUMERIC
  @ATTRIBUTE MEAN_AD_MEAN NUMERIC
  @ATTRIBUTE RMS_STD NUMERIC
  @ATTRIBUTE PAR_STD NUMERIC
  @ATTRIBUTE ZCR_STD NUMERIC
  @ATTRIBUTE MAD_STD NUMERIC
  @ATTRIBUTE MEAN_AD_STD NUMERIC
  @ATTRIBUTE class {music,speech}
  ```

  and the data lines should be:

  ```
  @DATA
  RMS_MEAN1,PAR_MEAN1,ZCR_MEAN1,MAD_MEAN1,MEAN_AD_MEAN1,RMS_STD1,PAR_STD1,ZCR_STD1,MAD_STD1,MEAN_AD_STD1,music
  ...
  ```

  Concretely, the @DATA section should be:

  ```
  @DATA
  0.057447,3.234547,0.191595,0.037308,0.044607,0.027113,0.397827,0.036597,0.018317,0.021898,music
  0.026583,2.590328,0.039416,0.016488,0.018977,0.013284,0.384978,0.015575,0.011143,0.012273,music

  ...
  0.062831,2.822102,0.082504,0.042271,0.049270,0.032323,0.799688,0.070962,0.027540,0.029103,speech
  ```

3. Classify the results of the whole-song and buffer-based ARFF files in Weka with `trees.J48` and save the results. Choose at least 1 other algorithm, and classify both ARFF files with that algorithm, again saving the output.

   Write a file called `classifications.txt` which compares the results of these 2 algorithms on the two ARFF files. The file should begin with 1-3 sentences stating which was the best results and summarizing its performance in comparison to the other results. This should be followed by the Weka results for the four cases.

4. Upload your ARFF file to:

   `http://cs4347.smcnus.org`

   This will automatically grade the values you calculated. If any mistake is found, please check your program and resubmit – you are welcome to submit as many versions as you wish before the submission deadline.

   Submit a zip file containing your program's source code (as a `.py` file), the ARFF file, and an optional README.txt file to the same website.

   - You may use anything in the python standard library, numpy (including pylab / matplotlib), and scipy libraries. No other libraries are permitted.

If you are familiar with python and understood the lecture, this should take about 1 hour.
Grading scheme:

- **3/6 marks**: correct ARFF file (automatically graded by computer).
- **2/6 marks**: readable source code (good variable names, clean functions, comments when needed).
- **1/6 marks**: Discussion of weka output

Assignment 4: Machine Learning and Spectral Features of Audio

CS 4347: Sound and Music Computing

due Wednesday 4 March 2015, 11:59 pm

0. This assignment will use the same "music / speech" dataset that we used in asn 1.

1. Make a copy of your python program from assignment 3 and modify it so that it:
   - Read the collection file
   - Load each wav file and split the data into buffers of length 1024 with 50% overlap. Only include complete buffers; if the final buffer has 1020 samples, omit that buffer.
   - Multiplies each buffer with a Hamming window. Hint: `scipy.signal.hamming()`
   - Performs a Discrete Fourier Transform. Hint: `scipy.fftpack.fft()`
   - Remember that the DFT gives you "positive" and "negative" frequencies, whose values are mirrored around the Nyquist frequency. Discard the negative frequencies (array indices above $N/2$ for an FFT of length N).
   - Calculate the following features for each spectral buffer. Since all these features use the absolute value of the spectrum, you may find it useful to convert the entire spectrum into absolute values (hint: `numpy.abs()`).
     Given a spectral buffer X,

     (a) Spectral Centroid (SC): the "brightness" of a sound

     $$\text{SC} = \frac{\sum_{k=0}^{N-1} k \cdot |X[k]|}{\sum_{k=0}^{N-1} |X[k]|}$$

     (b) Spectral Roll-Off (SRO): measures how the energy is concentrated throughout the spectrum. Specifically, SRO is the smallest bin index R such that L energy is less than the sum it. For audio analysis, we will use $L = 0.85$.

     $$\sum_{k=0}^{R-1} |X[k]| \geq L \cdot \sum_{k=0}^{N-1} |X[k]|$$

     (c) Spectral Flatness Measure (SFM): how "spiky" the spectrum is, calculated by dividing the geometric mean by the arithmetic mean

     $$\text{SFM} = \frac{\exp\left(\frac{1}{N}\sum_{k=0}^{N-1} \ln|X[k]|\right)}{\frac{1}{N}\sum_{k=0}^{N-1} |X[k]|}$$

     Using the log-scale is useful for avoiding multiplications which may exceed the bounds of double floating-point arithmetic.

     (d) Peak-to-average ratio (PARFFT): as defined in assignment 1.

(e) Spectral Flux (SF): difference between two successive buffers of DFT data. Let $|X[k]|_n$ be "the n^{th} analysis buffer".

$$\text{Flux}_n = \sum_{k=0}^{N-1} H(|X[k]|_n - |X[k]|_{n-1})$$
$$H(y) = \begin{cases} y & \text{if } y > 0 \\ 0 & \text{else} \end{cases}$$

We define $|X[k]|_{-1}$ as an array of zeros.

- After calculating the features for each buffer, calculate the mean and uncorrected sample standard deviation for each feature over all buffers for each file.
- Outputs the data to an `ARFF` file with the same format as before, but the headers should be `SC`, `SRO`, `SFM`, `PARFFT`, `SF`; each of them with a corresponding `MEAN_` and `STD_`.

  Concretely, the `@DATA` section should be:

```
@DATA
128.656296,239.404651,0.329993,9.164712,65.645735,13.206525,27.957121,0.087828,1.748588,45.773579,music
35.920625,63.362016,0.070470,14.030788,9.253576,5.640378,12.961779,0.017258,2.971727,11.300829,music
...
78.481380,145.886047,0.198849,13.112994,73.083275,39.388633,66.942115,0.133545,3.576990,64.046635,speech
```

  The header names must be:

```
SC_MEAN, SRO_MEAN, SFM_MEAN, PARFFT_MEAN, FLUX_MEAN,
SC_STD, SRO_STD, SFM_STD, PARFFT_STD, FLUX_STD
```

2. Classify the data with `trees.J48` with 10-fold cross-validation and save the results. In addition, select 2 other classification algorithms and record their output. Write a file called `classifications.txt` which compares the results of these three algorithms. The file should begin with 1-3 sentences stating which was the best algorithm and summarizing its performance in comparison to the other two algorithms. This should be followed by the Weka results ordered from best to worst.

3. Upload your ARFF file to:

   `http://cs4347.smcnus.org`

   This will automatically grade the values you calculated. If any mistake is found, please check your program and resubmit – you are welcome to submit as many versions as you wish before the submission deadline.

   Submit a zip file containing your program's source code (as a `.py` or `ipynb` file), the ARFF file, the `classifications.txt` file comparing different machine learning algorithms, and an optional README.txt file to the same website.

   - You may use anything in the python standard library, numpy (including pylab / matplotlib), and scipy libraries. No other libraries are permitted.

If you are familiar with python and understood the lecture, this should take 1–2 hours.
Grading scheme:

- **3/6 marks**: correct ARFF file (automatically graded by computer).
- **2/6 marks**: readable source code (good variable names, clean functions, comments when needed).
- **1/6 marks**: Discussion of weka output

Assignment 5: Perceptual audio features

CS 4347: Sound and Music Computing

due 11 March 2015, 11:59pm SGT

0. This assignment will make use of the "Music Speech" dataset of Marsyas we used for assignments 1–4.

   `http://opihi.cs.uvic.ca/sound/music_speech.tar.gz`

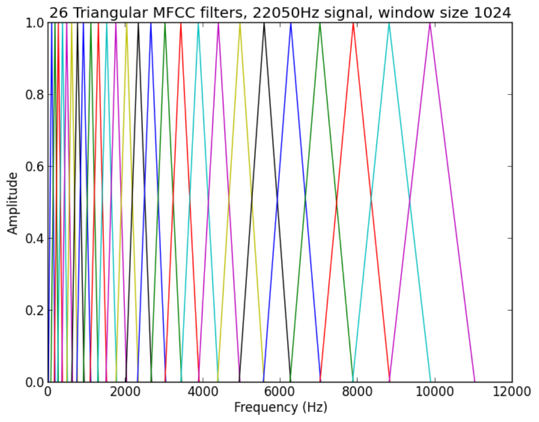
1. Write a program that:
   - Reads a collection file (format: filename \t (tab) label).
   - Load each wav file and split the data into buffers of length 1024 with 50% overlap. Only include complete buffers; if the final buffer has 1020 samples, omit that buffer.
   - Calculates the MFCCs for each window as specified in the lecture notes. A few more details:
     - Given input $x(t)$ and output $y(t)$, the pre-emphasis filter should be

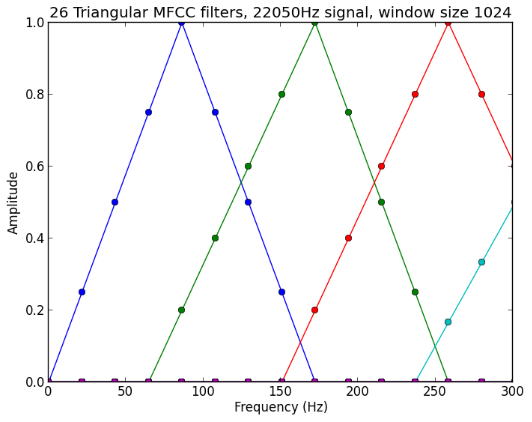
       $$y(t) = x(t) - 0.95x(t-1)$$
     - Use a Hamming window before the mag-spectrum calculation
     - Mel-scale of frequency f is:

       $$Mel(f) = 1127 \ln(1 + \frac{f}{700})$$
     - Calculate 26 mel-frequency filters, covering the entire frequency range (from 0 Hz to the Nyquist limit). To calculate the filters,
       * find the X-axis points of the filters (left side, top, right side). All points must be convereted into integer FFT bins; the left side should use the `floor()` operation; the top point should use `round()`; the right point should use `ceil()`.
       * assign the left bin to be 0, top bin to be 1.0, right bin to be 0; linearly interpolate between the rest
     - the log step should be log base 10.
     - scipy has DCT built-in: `scipy.fftpack.dct()`
     - do not calculate any delta-features
   - Calculates the mean and standard deviation for each MFCC bin over the entire file. So if there are M MFCC bins each each buffer, you will end up with a feature vector of length $2M$ for each song.
   - Writes the data to an arff file.
   - Classify the data with `trees.J48` with 10-fold cross-validation and save the results. In addition, select 2 other classification algorithms and record their output. Write a file called `classifications.txt` which compares the results of these three algorithms.

The file should begin with 1-3 sentences stating which was the best algorithm and summarizing its performance in comparison to the other two algorithms. This should be followed by the Weka results ordered from best to worst.

- Make two plots: the overall range of the triangular windows, and the triangular windows from 0 to 300 Hz. They should match the examples below.





2. Submit: the 2 PNGs, the ARFF file, the discussion of weka output, and your source code.

Grading scheme:

- **2/6 marks**: PNG files showing the triangular windows.
- **2/6 marks**: Discussion of weka output.
- **2/6 marks**: readable source code (good variable names, clean functions, comments when needed).

Assignment 6: Beat Tracking

CS 4347: Sound and Music Computing

due Wednesday 18 March 2015, 11:59 pm

NOTE: For inquiries on this assignment, please contact ZHU Shenggao (a0107950@nus.edu.sg).

0. In this activity you will learn to design a basic beat tracking algorithm. You can download an excerpt of the song 'Moskau', by Deschingus Khan, of approximately 15 seconds in length from the following link:

   `http://www.comp.nus.edu.sg/~duanzy/media/Moskau.wav`

   Over the course of this assignment, you will take this file and calculate an accent signal, estimate its periodicity, and finally determine the locations of its beats.

1. You may begin by making a copy of your previous Python assignment. Modify this program to do the following:
   - Read the file 'Moskau.wav'.
   - Load the wav file and split the data into frames. Each frame should be 1024 samples long and have 50% overlap. Only include complete buffers (e.g., if the final buffer has 1020 samples, omit that buffer).

2. You will now implement the first major component of the beat tracker, the accent signal:
   - First, for each frame, apply a Hamming window first, then use a Fast Fourier Transform operation to calculate its spectrogram, $S[n]$.
     - Remember that this operation gives you "positive" and "negative" frequencies, whose values are mirrored around the Nyquist frequency. Discard the negative frequencies (array indices above N/2 for an FFT of length N).
   - Next, calculate the value of an accent signal A according to the following equation:

   $$A[n] = \sum_{k=0}^{N/2} HWR(|S[n]| - |S[n-1]|), n = 1, 2, 3, ... \qquad (1)$$

   - Where k is the index of each frequency bin, and HWR is an operation in which all negative values are set to 0. Its full name is 'half-wave rectification'.
   - In other words, you will first calculate the difference between the magnitude spectrograms in frame n and frame $n-1$. You will then set any negative values to 0, and then sum over all frequencies in the frame. The result will be a single value, which is the accent signal value for that frame.
   - Note: when calculating $A[n]$, start at $n = 1$, and ignore $n = 0$. I.e., the length of $A[n]$ array will be one less than the # of frames.

3. As the next step, estimate the periodicity of the accent signal.

- Do this by first autocorrelating the accent signal, and get the resultant *autocorrelation of the accent signal* (AAS).
  Hint: the autocorrelation function in Python is as follows:

  ```
  def autocorr(x):
      result = numpy.correlate(x, x, mode='full')
      return result[result.size / 2:]
  ```

- Next, you will find the index of a local maximum of the autocorrelation of the accent signal (AAS). This index is related to the accent signal's periodicity.
  - The period of the accent signal means the same as the period of the audio signal. Given the audio period p in seconds, the audio tempo T in Beats Per Minute (BPM) can be calculated as $T = 60/p$.
  - The relationship between an autocorrelation index (an index in AAS) and the tempo T of the audio is

    $$Index = \frac{60}{T * L} \tag{2}$$

    Where L is the time interval between successive frames in seconds, and the tempo is in Beats Per Minute (BPM). Each frame is shifted by 512 samples at 44,100 samples/second, or equivalently, by about .0116 seconds. Therefore, L = .0116 seconds (approximately).
  - Note that AAS has several maxima. You will therefore need to find a sensible range of AAS within which to search for a maximum.
  - You may assume the true tempo of this music is between 60 and 180 BPM. Convert 60 and 180 BPM to autocorrelation indices, and use these two indices to determine the sensible range of AAS.
  - Find the index of the local maximum value of the AAS within this range. This index is the approximate number of frames between beats in this audio. This index will be called the *tempo index*, or t for short.
    Hint: In order to find the index of the maximum of a numpy array x, you may use the argmax function:

    ```
    index_max = x.argmax()
    ```

4. You will now create the algorithm to determine the locations of the beats in this piece of music.

   - Make your program search within the first t values of the original accent signal (not AAS) for a maximum value. For example, if your tempo index is 110, then search among the first 110 values. The index of this maximum value will be your first beat index.
   - Starting at the first beat index, your system should skip ahead by the t. In other words, if your first beat index is 50 and your tempo index is 110, your system should skip to index 160 of the accent signal.
   - Your system should then look for a maximum within 10 values on both sides of its new index. For instance, if your system's position is index 160, then the system should check for a maximum from index 150 to 170. The index of the maximum that you find will be your next beat index.
   - Repeat the previous two steps until you have stepped through the entire accent signal and have recorded a list of all of the beat indices.
   - When this has completed and you have all of your beat indices (in frames), you must convert them to times in seconds. This is a simple multiplication operation:

$$Time = Index * .0116 \tag{3}$$

- This equation works because the first time frame ranges from 0 seconds to .0223 seconds, with an average value of .0116 seconds, and each subsequent frame is shifted forwards in time by .0116 seconds.
- Write your beat times to file using the following line of code:

```
beat_time.tofile('beat_time.csv', sep=',')
```

5. Write a file called beattrack.txt and discuss the following:
   - In 1-2 sentences, describe the output. Did the system beat track the music correctly? (Hint: you can plot out the audio/accent signal in time and your calculated beats times. Then see if the beat times are reasonable or not?)
   - In 2-3 sentences, describe two assumptions that this system made about the music. For example: what would have happened if the tempo of the music had changed in the middle of the piece?
   - In 1-2 sentences, explain why you think the 'beat estimation' step searched for maxima within a narrow range instead of simply incrementing by the beat index. Hint: consider the resolution of .0116 seconds/frame. Is it likely that the period is exactly an integer number of frames? Is it likely that the musicians will play to a perfectly exact tempo?
6. Hint: you can compare your own outputs with the following results (but you should not copy and hard-code these results directly in your program):
   - Total # of frames: 1307
   - Total # of accent signal values: 1306
   - Tempo index: 39
   - Total # of beats: 33
   - First beat index: 33
7. Upload your result file beat_time.csv to

   `http://cs4347.smcnus.org`

   This will automatically grade the values you calculated. If any mistake is found, please check your program and resubmit. You are welcome to submit as many versions as you wish before the submission deadline.

   Submit a zip file containing your program's source code (as a .py), your discussion file, and an optional README.txt file to the same website. You may use anything in the python standard library, numpy (including pylab / matplotlib), and scipy libraries. No other libraries are permitted.

   Grading scheme:

   - **3/6 marks**: correct submission file (automatically graded by computer).
   - **2/6 marks**: readable source code (good variable names, clean functions, comments when needed).
   - **1/6 marks**: Discussion

Assignment 7: Audio Synthesis with Sine Waves

CS 4347: Sound and Music Computing

due Thursday 26 March 2015, 11:59 pm

NOTE: For inquiries on this assignment, please contact FANG Jiakun (a0123777@nus.edu.sg).

1. Additive synthesis

   Write a program that plays notes with these MIDI pitches:

   ```
   midis = [60, 62, 64, 65, 67, 69, 71, 72, 72, 0, 67, 0, 64, 0, 60]
   ```

   Hint: write a function which takes a MIDI number as input argument, and returns an array containing the synthesized sine waves. Use `numpy.concatenate()` to combine the notes together.

   - A "pitch" of 0 means a rest (also known as "silence", `numpy.zeros()`).
   - The fundamental frequency of each non-rest note `m` is:

     $$f_0 = 440 \cdot 2^{\frac{m-69}{12}}$$

   - Each note (including rests) should be 0.25 seconds long.
   - Use 16-bit samples and sampling rate of either 8000, 16000, or 32000 Hz.
   - Each note must use at least 4 sine waves with harmonic additive synthesis (frequencies are $n \cdot f_0$. You can choose your own n).
   - If any frequencies are aliased (due to going above the maximum frequency), you will lose a mark.
   - Save the audio to the file `notes.wav`.
     Hint: `scipy.io.wavfile.write()`
   - Create a spectrogram (window size 512, 50% overlap, blackman window, log-scale with constant of 10^{-10}) and save it to the file `spectrogram-notes.png`. The X-axis must be time, and the Y-axis must be frequency. You may display time in analysis frames, and frequency in bins (instead of converting to seconds and Hz).
     Hint: `pylab.imshow()`

2. ADSR Envelope

   Update your python program to apply an ADSR envelope to each note. You have the freedom to choose any ADSR envelope, but the resulting wav file should be audibly distinguishable from the one without envelope.

   - Save the audio to the file `notes-adsr.wav`.
   - Create a spectrogram and save it to the file `spectrogram-notes-adsr.png`.

3. Submit a zip file containing your program's source code (as a `.py` file), the `.wav` files, and the `.png` files to:

   `http://cs4347.smcnus.org`

- You may use anything in the python standard library, numpy (including pylab / matplotlib), and scipy libraries. No other libraries are permitted.

Grading scheme:

- **3/6 marks**: correct additive synthesis output (`.wav` and `.png`)
- **3/6 marks**: correct ADSR output (`.wav` and `.png`)

Assignment 8: Synthesis Beyond Two Sine Waves

CS 4347: Sound and Music Computing

due Wednesday 01 April 2015, 11:59 pm

NOTE: For inquiries on this assignment, please contact FANG Jiakun (a0123777@nus.edu.sg).

0. Each section (numbers 1–2) should have its own source code file.
1. Use additive synthesis to construct a band-limited sawtooth wave at $f = 1000$ Hz lasting 1.0 seconds with a maximum amplitude of 0.5, using $F_s = 44100$.

   The formulae for sawtooth wave is:

   $$x_{sawtooth}(t) = -\frac{2A}{\pi}\sum_{k=1}^{M}\frac{1}{k}\sin(k2\pi\frac{f}{F_s}t)$$

   M is the maximum possible number of sine waves without aliasing.

   Submit:

   - a png showing the time-domain perfect sawtooth wave and your reconstructed one. Only include 5-6 cycles in this figure. The title of your plot should state how many sine waves you used, which should be the maximum possible without aliasing.
     You may use `scipy.signal.sawtooth()` to create the perfect sawtooth wave.
   - a png showing the dB-magnitude FFT (not a spectrogram!) of a perfect sawtooth wave and your reconstructed one. Use FFT length of 8192 (so ignore the remaining $44100 - 8192 = 35908$ samples).
   - write 1 paragraph contrasting what you hear when you listen to a perfect sawtooth wave and your reconstructed one. You do not need to submit the `wav` files, but you must submit the paragraph.
   - your python source code.

2. Generate sine waves at different frequencies using a look-up table.

   - Create 1 look-up table with 16384 samples. This must contain a single cycle.
   - Use that look-up table to create sine waves at $f = 100.0$ Hz and $f = 1234.56$ Hz. Use $F_s = 44100$, and generate 1.0 seconds of audio. (hint: before proceeding, plot the look-up table to ensure that it only contains 1 cycle of a sine wave. 1 cycle means that the final value in this array should *not* be 0; it should be slightly below 0)
     For each sine wave, create 3 versions:
     - no interpolation (using the look-up table)
     - linear interpolation (using the look-up table)
     - perfect version (using `numpy.sin()` directly)
     - Calculate the maximum error of the look-up table versions: given `LUT_sine_wave` and `perfect_sine_wave`, do:

       ```
       max_error = numpy.max(numpy.abs(LUT_sine_wave - perfect_sine_wave))
       max_audio_file_error = 32767*max_error
       ```

- Repeat the above using a look-up table with 2048 samples.
  The above `max_audio_file_error` is not a completely accurate representation of potential errors in the `wav` files, but it is close enough. If you use `round()` for the "no interpolation", the error should be approximately 2π in the worst case for 16384 samples, and approximately 50 in the worst case for 2048 samples. If you did not use `round()`, then these errors will likely be twice as big.
  With interpolation, the best result should have an error less than 0.001.
- Submit:
  - a text file giving the `max_audio_file_error` of the 2 sine waves using no interpolation and linear interpolation for 16384 and 2048 samples. Your text file should be formatted as follows:

    ```
    Frequency  Interpolation    16384-sample    2048-sample
    100Hz      No               err_1           err_2
               Linear           err_2           err_4
    1234.56Hz  No               err_3           err_6
               Linear           err_4           err_8
    ```
  - your python source code.

Grading scheme:
- **3/6 marks**: files for 1. additive synthesis of a band-limited sawtooth
- **3/6 marks**: files for 2. the look-up table

Assignment 9: Analysis / Synthesis of Speech

CS 4347: Sound and Music Computing

due Sunday 12 April 2015, 11:59pm SGT

NOTE: For inquiries on this assignment, please contact ZHU Shenggao (a0107950@nus.edu.sg).

0. Download this file; note that it's sampling rate is Fs = 22050 Hz.
   `http://www.comp.nus.edu.sg/~duanzy/clear_d1.wav`

1. Analyze the above audio file.
   - Load the wav file and split the data into frames. Each frame has a window size $N = 128$ samples and has *no* overlap. Only include complete frames (e.g., if the final frame has less than N samples, omit that frame). Hint: total # of frames = 463
   - For each frame, apply a Hamming window first, and then calculate the magnitude spectrum using FFT. Remember to discard the "negative frequencies" of the spectrum (i.e., array indices above $N/2$).
   - For each frame's magnitude spectrum, find the maximum value (noted as M) of the spectrum, as well as the index of M. The corresponding frequency in Hz (noted as f) of that index is given by: f = index / N * Fs. Store the frequency f and amplitude M in an array.
     - Hint: first create an array to store these values for all frames:
       ```
       freq_amp = numpy.zeros((n_frames, 2)).
       ```
   - Output that array to a csv file `freq_amp.csv`. To make the output more legible, use:
     ```
     numpy.savetxt('freq_amp.csv', freq_amp, fmt='%.6g', delimiter=',')
     ```
     The file should begin (small variations due to floating-point inaccuracy may occur):
     ```
     10335.9,0.00139136
     10335.9,0.00117254
     11025,0.0013192
     ...
     ```

2. Reconstruct that audio using sine waves.
   - For each frame, based on its frequency f and amplitude M calculated above, generate a sine wave with N samples. The sine wave should also have frequency f and amplitude M. Hint: sample code for reference
     ```
     sine_wav = M * numpy.sin(2 * numpy.pi * f / Fs * numpy.arange(N))
     ```
   - Reconstruct the audio. Concatenate the generated sine waves of all frames into an array (say, `re_wav`), and save it as an audio file `reconstructed.wav`. Hint: normalize the array and convert it to 16-bit integer first. Sample code:
     ```
     re_wav_norm = re_wav / re_wav.max() * 32767
     re_wav_norm = re_wav_norm.astype(numpy.int16)
     scipy.io.wavfile.write('reconstructed.wav', Fs, re_wav_norm)
     ```

- When listening to the reconstructed audio, you may want to lower your computer's volume at first (just in case some surprising sound is generated)!

3. Plot the spectrograms of the original audio and the reconstructed audio.
   - Hint: for each audio, save the magnitude spectrum of all frames into an array (463 rows by 65 columns). This array is called the spectrogram. Remember window size $N = 128$, *no* overlap, Hamming windowing.
   - Plot the two spectrograms (say `spectrogram` and `re_spectrogram`) as two subplots in the same figure. Normalize each spectrogram (so that the maximum becomes 1) before plotting. The x-axis should be the frames, and y-axis the frequency bins. Also remember to add axis labels and titles to make a good figure. Save the figure as `spectrogram.png`. Hint: use `pylab.imshow()`. Sample code:

```
    pylab.subplot(2,1,1)
    pylab.imshow(spectrogram.T/spectrogram.max(),
                 origin='lower', aspect='auto')
    pylab.subplot(2,1,2)
    pylab.imshow(re_spectrogram.T/re_spectrogram.max(),
                 origin='lower', aspect='auto')
```

   - Figure 1 shows an example of the spectrograms:

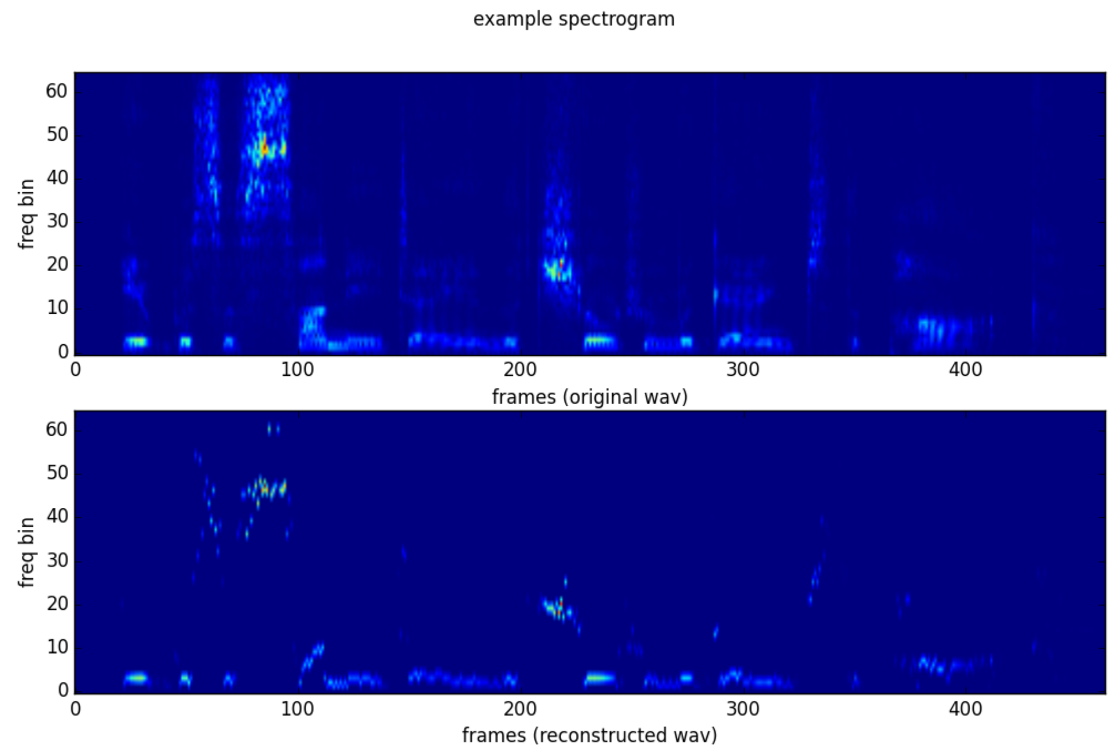


Figure 1: Example spectrogram

4. Write a file called `discussion.txt` and answer the following questions:
   - Listen to your reconstructed audio. Describe the difference between it and the original audio. Briefly explain the reasons for this difference.
   - Compare the two spectrograms in your plot. What difference can you observe regarding the frequency distribution for each frequency bin?

5. Upload your result csv file `freq_amp.csv` to

    `http://cs4347.smcnus.org`

   This will automatically grade the values you calculated. If any mistake is found, please check your program and resubmit. You are welcome to submit as many versions as you wish before the submission deadline.

   Submit a zip file containing (1) your program's source code (as a .py), (2) the reconstructed audio file `reconstructed.wav`, (3) the figure `spectrogram.png`, (4) your discussion file `discussion.txt`, and an optional README.txt file to the same website. You may use anything in the python standard library, numpy (including pylab / matplotlib), and scipy libraries. No other libraries are permitted.

   Grading scheme:

   - **2/6 marks**: automatic grading of `freq_amp.csv`
   - **1/6 marks**: reconstructed audio file `reconstructed.wav`
   - **1/6 marks**: the figure `spectrogram.png`
   - **1/6 marks**: discussion file `discussion.txt`
   - **1/6 marks**: source code quality