Sound Visualizer documentation

**1)About - What the app does?**

The Sound Visualizer is the app used for visualization of the music sample. The output of the program is a representation of note syllabus which the app detected. Typical use is catching the base tune of a solo or improvisation to which we have no note sheet. To find more information about how to use the app see the part 2).

Disclaimer: Results of the app are far from being precise note syllabus and are most accurate If

the sample is a recording of a single, one track instrument.

USER PART

**2)How to use the app?**

INPUT:

a) The input is selected through the UI (user interface) by

- either specifying the path to the input file As there is a possiblity of multiple files processed at the same time - every (also when there is only one) input file must be ended with semicolon and contain no blank spaces between files (e.g.:C:\\Users\Me\Music\Sample.wav;)

- clicking 'Browse' button and selecting the files via opened dialog

Please do not combine these two approaches.

Supported input format in current version is .wav.

b) Proceed with selecting the tuning of your instrument - this will be common for all selected files. (eg. C(0) for a typical piano or violin, Bes(-2) for a typical clarinet or tenor saxophone)

c)After a) and b) are completed, click 'proceed' button – This will start the process and

OUTPUT:

-For each input file there is output file created with suffix \_notes.ly.

(eg. input : input1.wav;input2.wav; output: input1\_notes.ly;input2\_notes.ly)

-The suffix .ly indicates LillyPond file which is a software that you need to install separately to transform text file into .pdf note sheet.

-Link to download LillyPond: <http://lilypond.org/download.html>

-After succesful instalation of LillyPond you need to doubleClick the .ly file and you get the final .pdf file.

DEVELOPER PART

**3) How does the app work?**

INPUT AND OUTPUT:

Input requirements: .wav file (no compression),

8/16/24 bit depth,

single or dual channelled,

selected through GUI

Output format: .ly file, further processed by LillyPond software

USED ALGORITHMS:

->Main -Header is read and proccessed

-Repeat: ReadBuffer ~ 4096 samples

ProcessBuffer - FFT, HPS

-Determine note lengths and filter errors

-Write output to the file

->FFT -Fast fourier transform, is done recursively with primitive complex roots of 1

->HPS -Harmonic product spectrum algorithm, applies the viewing function to the buffer in number of iterations, reduces error of double frequency appearing more dominant than the base one

->Hann's window -Applies window function to the buffer, to eliminate error that occurs when the wave captured in buffer does not start in phase with 0 amplitude (and does not end in phase with 0 amplitude)

IMPORTANT CLASSES/INTERFACES AND METHODS:

**Class Extensions**

-.MakeFFT() and .MakeHPS () – extends array of Complex values

**Class WavSoundReader :ISoundReader**

-reads and processes header of file in .wav format

-moves the buffer through the stream and returns array of byte values

**Interface IWindowFunction**

-has .Windowify() method that applies the function to the whole buffer

**Class LillyPondNoteWriter : INoteWriter**

-obtains List of Notes and creates text file further processed by LillyPond

**Class NoteDetector**

- when given a frequency and according to the configured transposition detects the closest note to the frequency

**Interface INoiseDetector**

-detects when there is no relevant sound input from the ISoundReader

**Interface IErrorCorrector**

-from a given List of Notes tries to detect those which are least likely that they‘ve truly been played

**Interface INoteLengthProcessor**

-szqeezes two adjacent notes of the same height of the length n and 1 to length n+1

**Class MainProcessor**

- executes the “main” algorithm

- grabs elements from the queue of files that are not yet processed

-there can be up to four processors working separately, each on the thread of its own

WORK ON THE PROJECT & FINAL WORDS

I started with implementation of the reader, which went smooth. When implementing FFT I got stuck for quite some time, because of problems with „missallign eror“ (Partially solved with Hann’s window) and also „octava error“ (supressed later by HPS correction). Then I chose the most fiting buffer size where I encountered something really similar to heisenberg principle. You’ll either have a good time resolution (quickly detecting short tones) or fine frequency resolution (accurate frequency detection) but you can’t have both. That is one of the reasons why the results of this app are only approximate. Nonetheless I’m satisfied with what the visualizer does and no matter if you are a developer or a musician I hope you’ll have some fun with my app.

Tomas Celko

24.8.2020