Group 2 - Audio Programming in Python

Playing Audio

Here we will describe the Python (Python 3) files we have made for creating a user interface to play audio as well as displaying decibel and rms.

File: *PlayMusic*

PlayMusic is the Python file to play up a sound file specified in the terminal. Important libraries, classes, methods and functions are outlined here.

Libraries:

- *Pyaudio*
- *wave*
- *PyQt5,* tested on versions <= PyQt5 (5.7.1)
- *threading* (standard Python library)

Class: *MusicPlayer*

Class for graphical interface inheriting from PyQt5 QWidget. Features include a “Pause Play” button; a progress bar tracking showing the current position in the song being played; and finally a slider allowing the user to effectively choose what part of the song they want to play.

Methods:

- *set_progress*
- *change_value*
- *handle_clicks*

Class: *Worker*

Worker class continuously updating the progress bar, inherits from QThread (PyQt5:s threading module).

Methods:

- *run*

Class: *Buffering*

Class implementing double event driven buffering by utilizing threading.

Methods:

- *callback*
- *load_buffers*
- *Flush_buffers*

Functions:

- *Initialize_stream* - function to initialize Pyaudio *stream* object. The Pyaudio *stream* will play continuously until it is paused or the song playing ends.

General Comments:
The wav file is instantiated utilizing the python library *wave*. This library allows us to find out format width, channels, framerate etc for the given file. Using this information we can use *Pyaudio* to instantiate an audio *stream*. The audio *stream* uses a callback function that reads frames and sends to the data to *Pyaudio*. To implement double (or more) event driven buffering the callback function is modified to queue up additional frames ahead of time. In order to implement this we utilize the standard python library *queue* as well as *threading*. Every time the callback function is called a new thread is started to queue up future frames. This allows the program to queue up new frames parallel to playing the song. In order to not start up too many new threads the maximum queue size is limited to some arbitrary number.

To allow the user to easily control the audio stream the class *MusicPlayer* is used. The progress of the song is displayed in a progress bar. To allow the progress bar to update continuously without interrupting the user interface it is run in a separate thread. If the user decides to rewind or fast forward using the slider, the program will react when the slider is released. To allow thread-safe modification of the progress bar as well the audio stream, both of these threads are paused on release, and restarted once the song position as well as the progress bar have been put in their new place. The Buffered frames described above will also be flushed at this point for obvious reasons.

File: *ComputeRmsDB*

The code in this Python file is imported to the PlayMusic file and used to display real-time dBFS and RMS-meters in the terminal.
Libraries:

- *pyaudio*

Class: *Meter*
This is a class that functions as a meter display. Every time that the RMS and dBFS values have been calculated they are passed to the class instantiation object.
Methods:

- *__call_*
- _ - the Meter class is callable as function. This is how the display is being continuously updated during playback.

Functions:

- *get_rms* - function that calculates and returns the *RMS* value and *dBFS* value of each buffer cycle.

Recording Audio

Here we will outline the Python (Python 3) files we have made for creating a user interface to record and save audio.

File: *SoundRecorder*

Soundrecorder is the python file to record a long or short sound file. Important libraries, classes, methods and functions are outlined here.

Libraries:
Same as in the file *PlayMusic* described above.

Class: *RecordGUI*
Class for graphical interface inheriting from PyQt5 QWidget. Features include a record button and a save button.
Methods:

- *start_pause*
- *save*

Class: *Worker*
Worker class (inheriting from threading) for continuously collecting data without interrupting the user interface.
Methods:

- *run*
- *pause*
- *resume*
- *get_frames*

Functions:

- *initialize_stream*
- *disk_usage* - returns the current amount of free disk space
- *save_as_handler* - function to handle user interaction through terminal to save file, ask about overwriting etc.

General Comments:
In this case the *Pyaudio stream* is instantiated before there is any wav file. This means that parameters need to specified explicitly in the program. The user interface allows the user to start recording by pressing record. This start a new thread continuously feeding the *stream* with audio

frames. The frames are thus converted to data which is added to a list. With every appended frame the program will retrieve the current disk space and ask if it is more than a specified threshold. If free disk space is below this threshold the thread will pause and inform the user how much disk space is left, and prompt him or her to press to record button again. After this future warnings will be suppressed.

The user is allowed to pause and resume the recording as many times as he or she wishes.

Once the user presses save the dialog window closes and communication is moved to the terminal. There the user is asked to specify as filename. If this filename overlaps with another file in the same directory the user will be confirm overwrite of choose a new filename.

Real Time Modification of audio

PURE DATA

We have programmed a pd-patch that takes your singing voice and converts it to MIDI notes and then plays back the corresponding pitch of the voice with a real-time modifiable waveform. The audio to midi conversion is done with the sigmund object in pure data. Then we integer-divide and round the numbers as to make them into 12-tone chromatic tones. Then we convert the MIDI notes back into a sinusoidal which then is played back. The sinusoidal waveform can be modified in realtime when singing. Pressing any key on the computer toggles the sound output on/off. Each and every step of the path is commented on in the pd-patch itself. Since the pd is a graphical language itself, one can argue that it makes more sense to look at the comments themselves than trying to understand this summary of the patch.

Function:

The pure data patch performs the following steps:

1. It takes the microphone input
2. Converts it to MIDI
3. Rounds values to 12-tone chromatic notes
4. Converts the signal back to a sinusoidal
5. Outputs this signal with the ability to alter the waveform in real time.

Files:

- *Dt2410_voice_to_synth.pd*
- *output~.pd*