UNIVERSITEIT VAN AMSTERDAM

Processing live audiostreams: FOQAPS

Wouter Vrielink (10433597) & Bram van den Akker (10434100)

Date:
December 21, 2015

Contents

1 Design FOQAPS class

The goal of FOQAPS is to implement a Python module that can apply filters to a real time audio stream. The FOQAPS class uses the alsaaudio, scipy, and numpy libraries. The alsaaudio library can be found on `https://larsimmisch.github.io/pyalsaaudio/libalsaaudio.html`, and can be installed through pip with the following command: *pip install pyalsaaudio*.

1.1 __init__(self)

Initialise input and output streams.

1.2 create_input_stream(self)

Create an input stream and store it in the object.

1.3 create_output_stream(self)

Create an output stream and store it in the object.

1.4 execute_filter(self, b, a, buffer, pre_buffer)

Execute the actual filter over the buffer.
b: The numerator coefficient vector in a 1-D sequence.
a: The denominator coefficient vector in a 1-D sequence. If a[0] is not 1, then both a and b are normalised by a[0].
buffer: The current buffer from the input stream (1d array).
pre_buffer: The previous buffer from the input stream (1d array).

Returns a 1d output with the length of a single buffer.

1.5 rt_lfilter(self, b, a, run_time)

Execute a filter over the audio captured with the microphone.
b: The numerator coefficient vector in a 1-D sequence.
a: The denominator coefficient vector in a 1-D sequence. If a[0] is not 1, then both a and b are normalised by a[0].
run_time: Time to run filter in seconds

Method ‘rt_lfilter‘ is based on the scipy.signal.lfilter method, but uses a timeframe instead of input stream. The input stream will be generated with the realtime data.

2 Details

This section will go over some details regarding customisability and workings.

2.1 Audiostream

The audiostream methods are provided by the alsaaudio library. Herein, it is possible to change the number of channels, the sample size, the sample rate (frame rate), and the buffer size (period size).

The FOQAPS class can currently not apply filters on more than one channel or data with a sample size bigger than one byte. (This would require some changes in the code). Sample rate has to be the sample rate of the stream; if the sample rate is incorrect, the output stream will be corrupt. Changing the buffer size will affect the delay and accuracy of the output stream. When a small buffer size is chosen (less then 1/200th of the sample rate), it becomes difficult to properly apply a filter (FFT becomes inaccurate). Buffer size will directly influence the delay as the output stream will always be at least one buffer behind. If the buffer size is one tenth of the sample rate, the delay will be one tenth of a second. We found period sizes below 320 for a sample frequency of 44.1 kHz to be very lossy.

The audiostream is based on Pulse-code modulation (PCM), which means that it is a digital signal that was discretised from an analog signal. This signal can be read from the stream in chunks with the set buffer size. The audiostream has been configured to be blocking. This means that if there is not enough data in the audiostream buffer, the function will wait until there is enough data. This is why the buffer size affects the delay directly. The data from the audiostream is composed of a string of values, which we unpack to an array of floats.

There is one last issue with the audiostream. The inputstream will buffer all samples that have taken place. This means that if the amount of time it takes to modify your data and put the results in the output stream is higher than the buffer time, you will slowly go out of synchronisation; the outputstream will be lagging behind. To prevent this from happening we have chosen to make our internal buffer size variable, so that it can hold all buffers from the inputstream that are available at that time. We then sub-sample this data back to the size of our original buffer. This way, the input and outputstreams will stay synchronized over time.

2.2 Filters

The filters applied are provided by the Scipy.signals library. Herein, it is possible to change the order and the critical frequencies. These critical frequencies are the points at which the gain drops to $1/\sqrt{2}$ that of the passband. The critical frequencies must be given as a float ranging from zero to one. One being the niquist frequency (or half of the sample rate).

The use of lfilter to apply our filter to the buffer requires us to remember at least a part of the prior buffer. This is necessary because lfilter takes a couple of samples to "catch up"; it is always a couple of samples behind. We have chosen to keep the whole previous buffer and concatenate it with the

current buffer, and then apply lfilter to that. When this is done we get the second half of the results and use that as result.

3 Using FOQAPS

The FOQAPS module can be found at `https://github.com/WouterVrielink/DSP/tree/master/FOQAPS`. It is possible to either download or clone the repository.

Import the module class in python.

```
from foqaps import FOQAPS
```

Initiate the python object.

```
foqaps = FOQAPS()
```

Use any tool to create the numerator coefficient vector and the denominator coefficient vector. It is possible to use either FIR or IIR filters (see `http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.signal.lfilter.html#scipy.signal.lfilter`). Example using scipy.signal.butter (see `http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.signal.butter.html`):

```
# Determine the numerator and denominator.
b, a = signal.butter(order, freq, btype = 'lowpass', analog=
    False)
```

Start the realtime linear filter with the run time (in seconds), calculated numerator and denominator

```
foqaps.rt_lfilter(b, a, 60)
```

3.1 Examples

To demonstrate the use of the FOQAPS library, we provide you with a few extra examples.

lowpass.py: Example use of the FOQAPS library using a lowpass filter.

bandpass.py: Example of the FOQAPS library using a bandpass filter.