Noise cancellation using spectral gating in python

Author: *Nijat Hamidov L2, CS-19 Group-1*

Supervisor: *Prof. Javid Khalilov, javid.khalilov@ufaz.az*

Submission Date: *12/11/2021*

Table of Contents

Abstract

One of the main techniques that can be applied to reduce background noise at low level pitch is by using the noise reduction algorithm called *Fourier analysis*[1]. The "frequency spectrum" of the sound is determined by locating the spectrum of

pure tones that make up the background noise in the selected quite sound segment. That forms a fingerprint of the static background noise in the given sound file. When the noise is reduced from the sound as a whole, the algorithm finds the frequency spectrum of each short segment of sound. Any pure tones that aren't sufficiently louder than their average levels in the fingerprint are reduced in volume by using the general technique called *spectral gating*[2]. The first pass of noise reduction is done over just noise. For each windowed sample of the sound, we take a *Short-time Fourier Transform (STFT)*[3] using a Hann window and then statistics, including *the mean power*, are tabulated for each frequency band.

Used Libraries

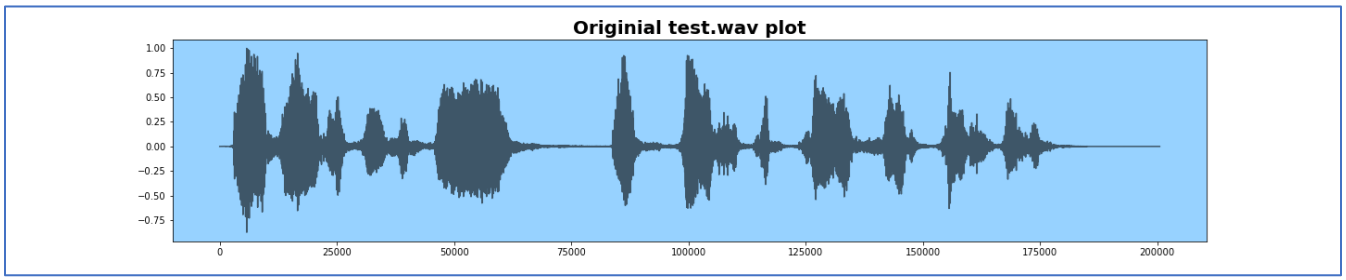
All of the code below were run in *jupyter notebook* with imported *IPython* module and enabled *%matplotlib inline* magic. Reading the original test.wav file was accomplished using the *scipy.io wavfile module*. Plotting the graphs of signal and noise along with the spectrum image was done by using *matplotlib's pyplot* module. All the numerical computations on arrays was provided by *numpy* library and its functions except for the few one (e.x for short-time Fourier transformation) which were used from *librosa* library.

Reading the original test.wav file

```
wav_loc = "test.wav"
rate, data = wavfile.read(wav_loc)

print(type(data)) # nd.array
print(data.ndim) # Data is 1-D array for 1-channel WAV
print(data.size) # 200542

# Standard 44.1 kHz sample rate for 16-bit wav file
print(rate) # 44100

# Assuming that the wav-file is 16 bit integer, the range is [-32768, 32767],
# thus dividing by 32768 (2^15) will give the proper twos-complement range of [-1, 1]

data = data / 32768
```

After reading test.wav file its data and sampling rate was stored in *data* and *rate* variable, respectively. Plot of the *data* variable which represents the wave for test.wav file is plotted on *Figure 1.1* .

Figure 1.1



Preparing noise for subsequent imposition

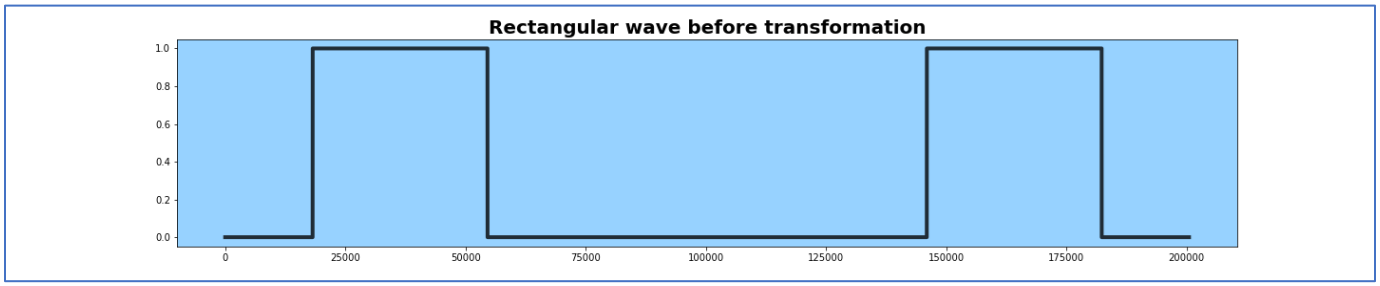
After that the noise will be prepared from rectangular wave which in turn was transformed using inverse Fourier transformation. For that two function were used.

Rectangular wave before IFFT

```
def band_limited_noise(min_freq, max_freq, samples=1024, samplerate=1):
# Sample spacing (inverse of the sampling rate) parameter 1 / samplerate

sample spacing is in seconds

#   Array of length n containing the absolute values of sample frequencies.
    freqs = np.abs(np.fft.fftfreq(samples, 1 / samplerate))

# frequency unit is in cycles/second


    f = np.zeros(samples) # 1D array of 200542 zeros


# Rectangular wave to be transformed
    f[np.logical_and(freqs >= min_freq, freqs <= max_freq)] = 1


    return fftnoise(f)
```

Before returning transformed sampled noise **`band_limited_noise`** functions will create a rectangular wave (*Figure 1.2*) which itself will transformed using IFFT by the **`fftnoise`** function.

Figure 1.2

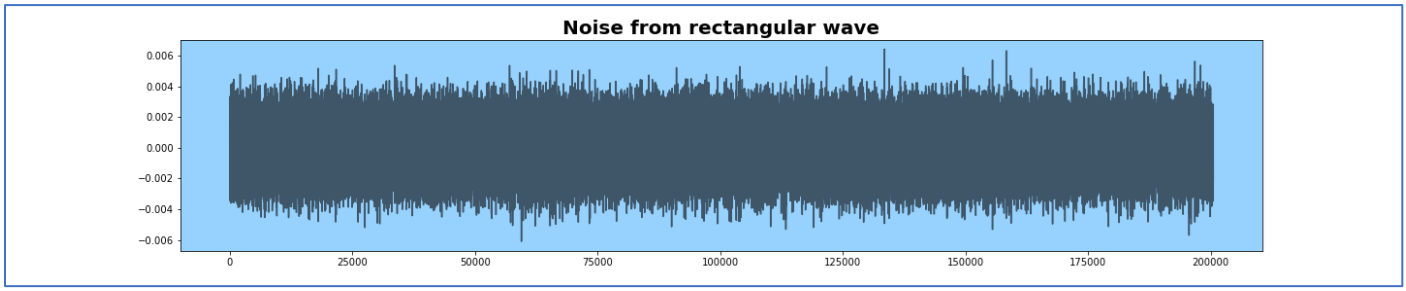


Getting Noise (Rectangular wave after IFFT)

```
def fftnoise(f):
    # array of complx numbers with size len(data)=200542
    f = np.array(f, dtype="complex")


    Np = (len(f) - 1) // 2
    phases = np.random.rand(Np) * 2 * np.pi

    # e^(iwt) = cos(wt) + i*sin(wt)
    phases = np.cos(phases) + 1j * np.sin(phases)
    f[1 : Np + 1] *= phases
    f[-1 : -1 - Np : -1] = np.conj(f[1 : Np + 1])

#   Applying IFFT
    return np.fft.ifft(f).real
```

Applying IFFT gives the noise (*Figure 1.3*) which can be imposed to our original signal (test.wav). **Note:** noise array will be multiplied by 10 in order to increase amplitude.

Figure 1.3

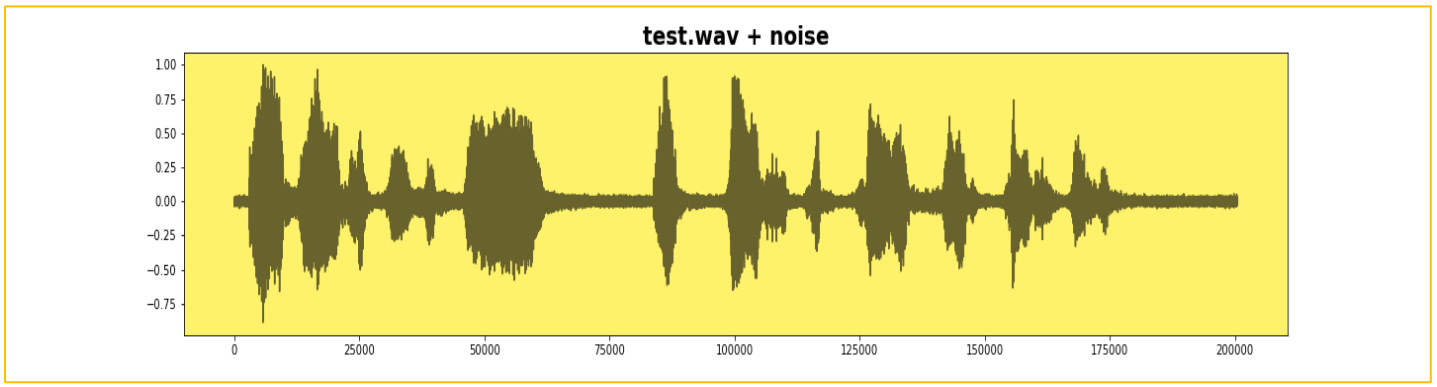


Imposing noise to the signal

```
noise_len = 2 # seconds

# multiplied by ten for amplitude to be 1 order high
noise = band_limited_noise(min_freq=4000, max_freq = 12000, samples=len(data), samplerate=rate)*10


# get two seconds of the clip with sampling rate=44.1kHz
noise_clip = noise[:rate*noise_len]

# Adding noise
audio_clip_band_limited = data+noise
```

Adding noise to the original signal has the effect of thickening the graph of the original signal. The modified original signal is show on *Figure 1.4* .

Figure 1.4



Denoising modified signal

After modifying original wave file the first thing in order to remove imposed noise is to calculate apply Short-time Fourier Transformation over the noise audio clip. There will be needed some additional functions given below to do STFT over the noise clip. Given that functions from librose.core module, spectral gating algorithm's steps are followed in order to reproduce the original data.

Additional functions

```
import time
from datetime import timedelta as td


def _stft(y, n_fft, hop_length, win_length):
    return librosa.stft(y=y, n_fft=n_fft, hop_length=hop_length,
win_length=win_length)


def _istft(y, hop_length, win_length):
    return librosa.istft(y, hop_length, win_length)


def _amp_to_db(x):
    return librosa.core.amplitude_to_db(x, ref=1.0, amin=1e-20,
top_db=80.0)


def _db_to_amp(x,):
    return librosa.core.db_to_amplitude(x, ref=1.0)


def plot_spectrogram(signal, title):
    fig, ax = plt.subplots(figsize=(20, 4))
    cax = ax.matshow(
        signal,
        origin="lower",
        aspect="auto",
        cmap=plt.cm.seismic,
        vmin=-1 * np.max(np.abs(signal)),
        vmax=np.max(np.abs(signal)),
    )
    fig.colorbar(cax)
    ax.set_title(title)
    plt.tight_layout()
    plt.show()
```

An STFT is calculated over the noise audio clip

```
# STFT over noise
 noise_stft = _stft(noise_clip, n_fft, hop_length, win_length)
 noise_stft_db = _amp_to_db(np.abs(noise_stft))  # convert to dB
```

Statistics are calculated over SFFT of the the noise (in frequency)

```
# Calculate statistics over noise

# Mean of the stft noise
    mean_freq_noise = np.mean(noise_stft_db, axis=1)
# Standard deviation of the stft noise
    std_freq_noise = np.std(noise_stft_db, axis=1)
```

A threshold is calculated based upon the statistics of the noise (and the desired sensitivity of the algorithm)

```
# Threshold of the stft noise
    noise_thresh = mean_freq_noise + std_freq_noise * n_std_thresh
```

An FFT is calculated over the signal

```
# STFT over signal
 sig_stft = _stft(audio_clip, n_fft, hop_length, win_length)

 sig_stft_db = _amp_to_db(np.abs(sig_stft))
```

Obtaining the value of the mask and creating smoothing filter

The minimum value is taken as the mask (in dB) from the array of absolute values of amplitudes ,converted to decibels, which in turn are taken from STFT signal.

```
# Calculate value to mask dB to
    mask_gain_dB = np.min(_amp_to_db(np.abs(sig_stft)))
    print(noise_thresh, mask_gain_dB)
    # Create a smoothing filter for the mask in time and frequency
    smoothing_filter = np.outer(
        np.concatenate(
            [
                np.linspace(0, 1, n_grad_freq + 1, endpoint=False),
                np.linspace(1, 0, n_grad_freq + 2),
            ]
        )[1:-1],
        np.concatenate(
            [
                np.linspace(0, 1, n_grad_time + 1, endpoint=False),
                np.linspace(1, 0, n_grad_time + 2),
            ]
        )[1:-1],
    )
    smoothing_filter = smoothing_filter / np.sum(smoothing_filter)
    # calculate the threshold for each frequency/time bin
    db_thresh = np.repeat(
        np.reshape(noise_thresh, [1, len(mean_freq_noise)]),
        np.shape(sig_stft_db)[1],
        axis=0,
    ).T
```

A mask is determined by comparing the signal STFT to the threshold

```
    # mask if the signal is above the threshold
    sig_mask = sig_stft_db < db_thresh
```

The mask is smoothed with a filter over frequency and time

```
  # convolve the mask with a smoothing filter
sig_mask = scipy.signal.fftconvolve(sig_mask, smoothing_filter, mode="same")

sig_mask = sig_mask * prop_decrease
```

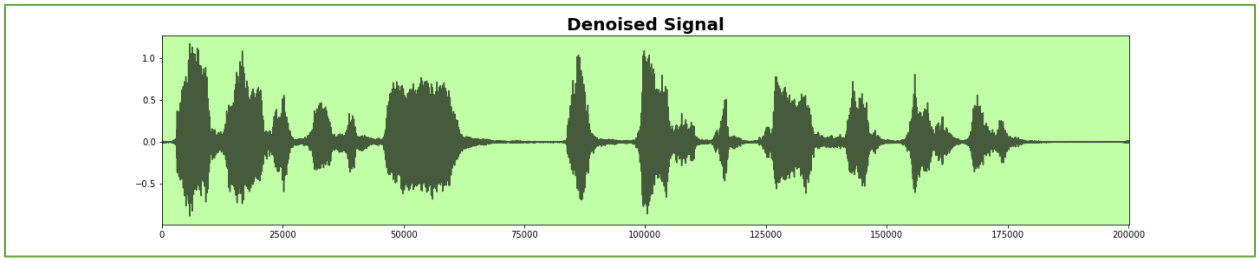
The mask is appled to the STFT of the signal, and is inverted

```
# mask the signal
    sig_stft_db_masked = (
        sig_stft_db * (1 - sig_mask)
        + np.ones(np.shape(mask_gain_dB)) * mask_gain_dB * sig_mask
    )  # mask real
    sig_imag_masked = np.imag(sig_stft) * (1 - sig_mask)
    sig_stft_amp = (_db_to_amp(sig_stft_db_masked) * np.sign(sig_stft)) + (
        1j * sig_imag_masked
    )
```

```
    # recover the signal
    recovered_signal = _istft(sig_stft_amp, hop_length, win_length)
    recovered_spec = _amp_to_db(
        np.abs(_stft(recovered_signal, n_fft, hop_length, win_length))
```

Denoised signal

Applying all the steps of the algorithm gives a way to reproduce original wave file the graph of which is shown on *Figure 1.5* .



References:

1. https://en.wikipedia.org/wiki/Fourier_analysis
2. https://en.wikipedia.org/wiki/Noise_gate
3. https://nl.wikipedia.org/wiki/Short-time_Fourier_transform
4. https://www.youtube.com/watch?v=spUNpyF58BY
5. https://www.youtube.com/watch?v=3gjJDuCAEQQ