Image Processing - Exercise 2

Hadar Tal, hadar.tal, 207992728

Introduction

In this exercise, the goal was to implement a denoising algorithm for audio signals, specifically focusing on the transition from the time domain to the frequency domain through Fourier transforms. Leveraging tools and libraries commonly used in Python for audio processing, the primary domains included popular Python libraries such as *SciPy* and *NumPy* for handling audio files, performing Fourier transforms, and managing numerical computations. Additionally, *Matplotlib* was employed for visualizing frequency histograms over time, aiding in the analysis and refinement of the denoising algorithm.

The two types of noises present in the audio signals presented distinct challenges and required tailored approaches for effective denoising. The first noise exhibited a consistent pattern throughout the entire duration of the audio, allowing for the application of statistical tools. By identifying and manipulating specific frequencies associated with the persistent noise, a targeted denoising strategy was developed. Contrastingly, the second noise lacked a discernible pattern, requiring a trial-and-error approach based on the auditory characteristics of the sound. This distinction in noise patterns influenced the algorithmic choices and methods employed during the denoising process.

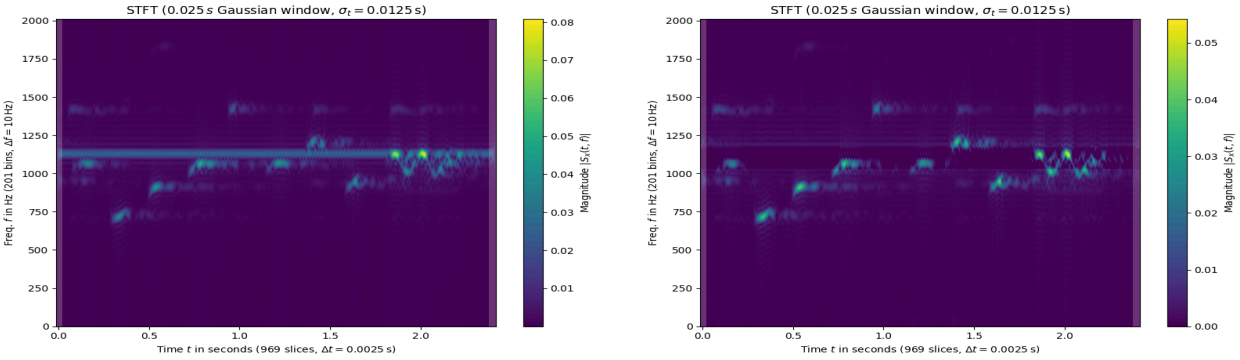
Algorithm

Noise 1:

1. Read the input - read the wav format input as a tuple of 1D array and sampling rate.
2. Perform a short time fast Fourier transform.
3. Identify noisy frequencies and find their average magnitude based on probability and magnitude thresholds
4. Reduce the amplitude of the noisy frequencies in each splice
5. Perform the Inverse short time Fourier transform

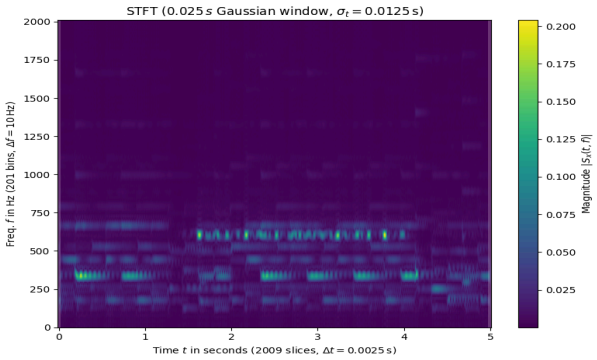
Noise 2:

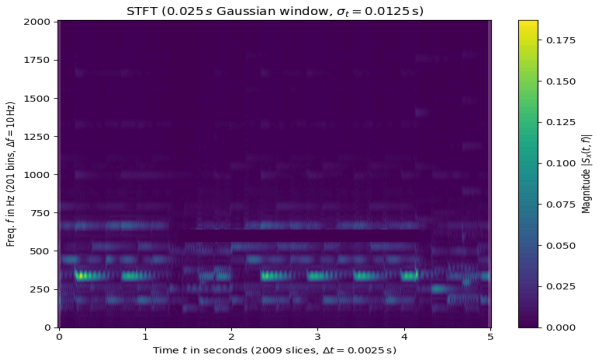
1. Read the input.
2. Perform a short time fast Fourier transform.
3. Remove noisy frequencies in specific predefined frequencies and splices.
4. Perform the Inverse short time Fourier transform



STFT Computation: Initially, I used *NumPy*'s FFT and implemented the STFT computation from scratch for learning purposes, then switched to *SciPy*'s ShortTimeFFT for better performance and accuracy in inverse computation. The initial manual code is in the Python file.
Identifying and Marking Noisy Frequencies: The function analyzes a 2D array of frequency magnitudes over time in an audio signal, marking frequencies as noisy based on probability and magnitude thresholds. With *probability_threshold* = 0.5 and *magnitude_threshold* = 0.05, it returns frequencies that, at least half of the time, have a magnitude 5% from the maximum. This ensures effective identification of consistent noise throughout the audio. The resulting array contains mean magnitudes of frequencies marked for denoising. Threshold values were established through iterative testing and analysis. The depicted figures showcase the approach's success in detecting persistent noise around 1170Hz.
Amplitude Reduction: Implemented functions for targeted denoising by selectively reducing the amplitude of identified noisy frequencies while preserving phase information. This approach allows nuanced control over amplitude reduction, retaining overlapping frequencies with the noise to preserve shared sound components. The method effectively mitigates noise impact while maintaining the audio signal's integrity (e.g., preserving frequencies around 1170Hz at the 2nd second).

Unlike Q1, Q2's varied noise patterns demanded a nuanced strategy. A detailed analysis, including histogram examination and auditory inspection, precisely matched noise characteristics with specific frequencies. Two key functions, *calculate_slice_index* and *calculate_slice_frequency*, were crucial. The former converted real-time values to window slices, adjusting for *window_step* deviations, while the latter ensured precise identification and removal of noisy frequencies. Combined with tailored parameter settings (min_sec, max_sec, min_frequency, max_frequency) = (1.4, 4.2, 565, 645), these calculations formed an effective and customized denoising strategy for Q2's unique challenges (Figures Below: STFTs of Q2's Noise - Left Before Denoising, Right After Denoising).





Hyperparameters and Thresholds: the params were set set as follows:
* *window_length*=400: the length (in samples) of the window for STFT.
* *window_step*=10: the step size (in samples) between consecutive windows.
* *gaussian_window*=100: the size of the symmetric Gaussian window applied to each segment of the input signal during STFT computation.
* *gaussian_std*=50: the standard deviation of the symmetric Gaussian window.

In Fig (1) below, the tuned STFT histogram of the first noise is presented. Precision in adjusting the *gaussian_std* parameter is crucial for crafting a spectrogram that seamlessly represents the underlying signal, avoiding both excessive blurring of frequency details and undue noise. A small std (Fig (2), set to 10) yields a narrow Gaussian window, offering high time localization and the ability to capture intricate details. However, it introduces noise and artifacts, perceptibly blurring along the frequency axis. Conversely, a large std (Fig (3), set to 900) results in a broader Gaussian window, enhancing time resolution but compromising frequency localization. Overlapping windows make it challenging to distinctly discern individual frequencies. In each time frame, the dominant frequency in its respective window becomes prominently apparent, contributing to the overall "compressed" visual representation of the spectrogram.

The *window_step* parameter sets the time gap between consecutive windows in the STFT. A smaller *window_step* allows for a more detailed analysis of rapidly changing frequency components, but comes at the cost of increased computational load. Conversely, a larger *window_step* reduces computational requirements but sacrifices temporal precision. In Fig (4), an example with a too large *window_step* (set to 1000) is shown. There's not much overlap between windows, causing information loss. The STFT misses quick events or changes in the signal since the windows are less likely to capture them. As a result, the histogram becomes blurry across the time axis, and we don't see the relatively small changes in frequencies over time.

The selection of *window_length* in STFT involves a balancing act between time and frequency resolution. Short windows, denoted by smaller *window_length* values, offer heightened accuracy at specific time points. This is because a shorter window

captures a smaller time segment, yielding more detailed information about the frequency components present during that interval. However, this advantage comes at the cost of frequency resolution. In shorter windows, the frequency "resolution" decreases, meaning the ability to distinguish closely spaced frequencies diminishes. Narrower frequency bins result in less precise identification of individual frequencies, leading to a loss of detail in the frequency domain. In Fig (5), with a *window_length* set to 40, the histogram displays a discernible "quantization" effect among various frequency bins.

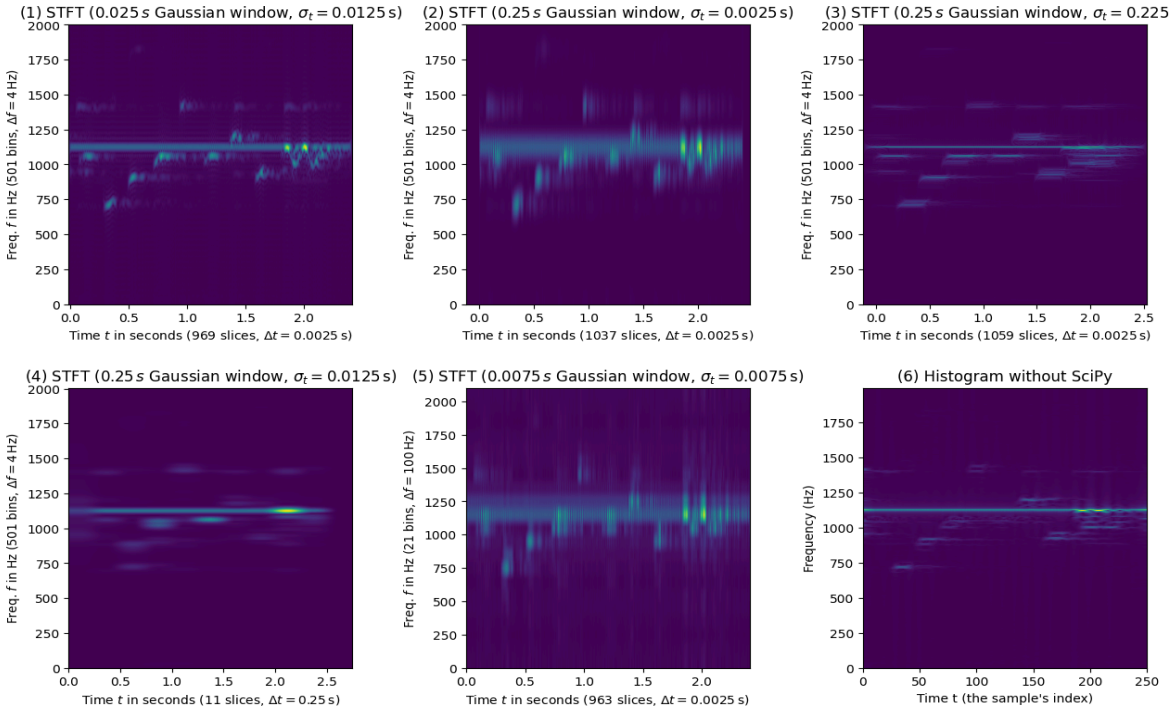


Fig (6) shows an histogram which was created without using SciPy lib.

Conclusion

In summary, the denoising process offered valuable insights into the practical application of signal processing techniques. Notably, systematic algorithms proved effective in targeted denoising, demonstrated by the reduction of consistent noise in Q1. The challenge of addressing non-patterned noise in Q2 emphasized the necessity for adaptability and versatility in real-world audio processing. The journey underscored the application of theoretical knowledge, emphasizing tools like the Fast Fourier Transform (FFT) for addressing complex problems. Overall, the experience highlighted the crucial role of flexibility in crafting solutions when dealing with diverse and unpredictable patterns in real-world signals.