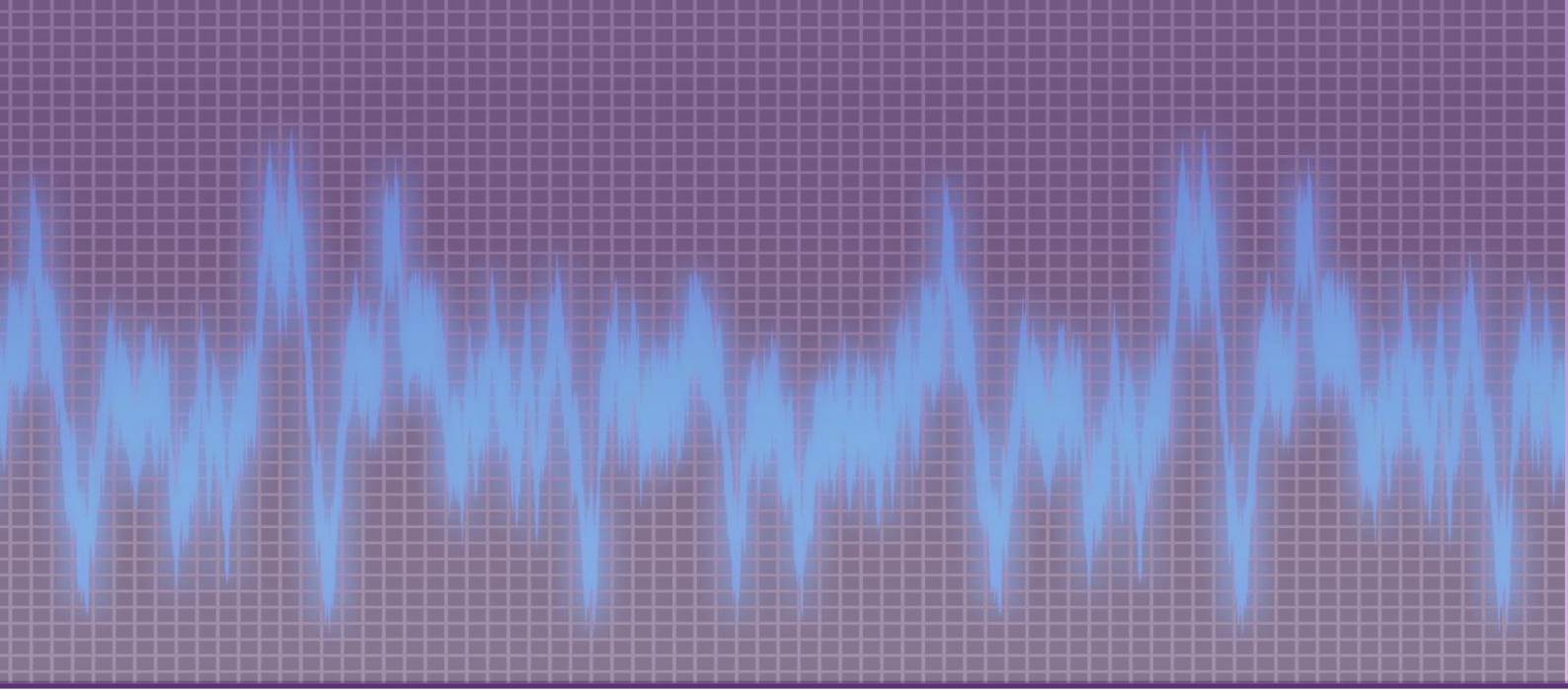


Compte Rendu TP3

Filtrage numérique d'un signal audio bruité

Réalisé par : Hachem Squalli ElHoussaini N°29

Dirigé par : Pr. H. TOUZANI

LAB 1 :

Exercice :

1. Analyse comparative d'un signal audio original et de sa version bruitée par un parasite haute fréquence (1 kHz).

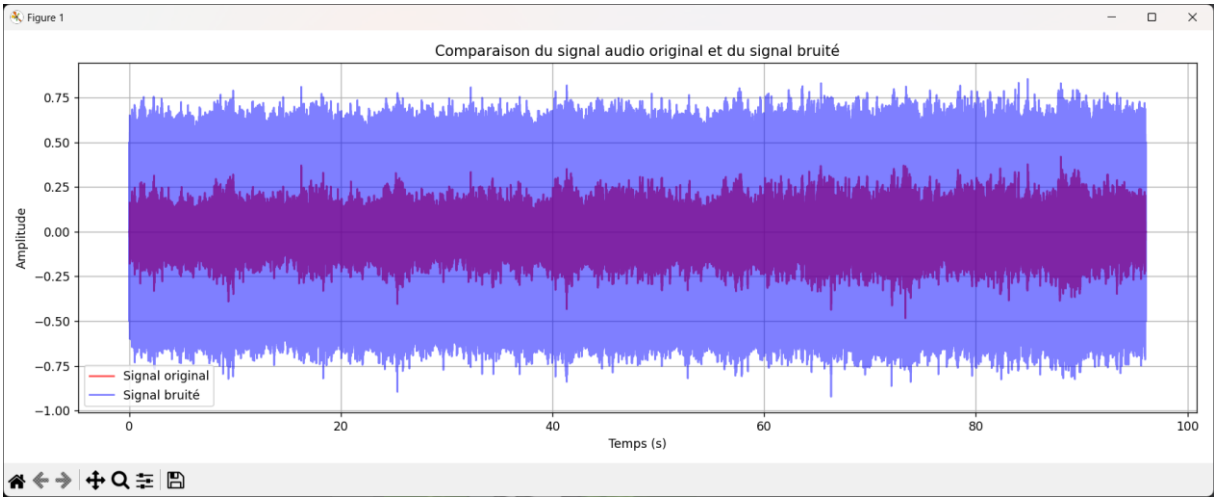
Points clés :

- Superposition temporelle des deux signaux (original rouge vs bruité bleu)
- Bruit additif sinusoidal à 1000 Hz d'amplitude 0.5
- Visualisation de l'impact spectral dans le domaine temporel

Paramètres :

- Fréquence du bruit : 1 kHz
- Amplitude du bruit : 50% du signal original
- Échelle temporelle linéaire pour comparaison visuelle

```python
import librosa
import matplotlib.pyplot as plt
import numpy as np

audio_file = "sample-1.wav"

# Load audio file
signal_audio, sr = librosa.load(audio_file, sr=None)
time = np.linspace(0, len(signal_audio)/sr, len(signal_audio))

# Create high frequency noise
f_bruit = 1000
amplitude_bruit = 0.5
bruit_haute_frequence = amplitude_bruit * np.sin(2 * np.pi * f_bruit * time)

# Create noisy signal
signal_bruite = signal_audio + bruit_haute_frequence

# Create figure
plt.figure(figsize=(14, 5))

# Plot both signals on the same graph
plt.plot(time, signal_audio, color="red", alpha=0.7, label='Signal original')
plt.plot(time, signal_bruite, color="blue", alpha=0.5, label='Signal bruité')

# Add title and labels
plt.title('Comparaison du signal audio original et du signal bruité')
plt.xlabel('Temps (s)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend()

# Adjust layout and show
plt.tight_layout()
plt.show()
```



2. Implémentation d'un système de génération et de lecture en temps réel d'un signal audio bruité par un parasite haute fréquence (1 kHz).

Points clés :

- Injection d'un bruit sinusoïdal à 1000 Hz (amplitude 0.5)
- Conversion du signal float32 en format PCM 16 bits
- Lecture audio interactive via Pygame
- Contrôle des limites (-1,1) avant conversion

Spécifications techniques :

- Fréquence d'échantillonnage préservée (sr originale)
- Normalisation et clipping pour éviter la saturation

```python
import librosa
import numpy as np
import pygame
import soundfile as sf  # enrigistrer WAV files
audio_file = "sample-1.wav"
sound = audio_file
# Load audio file
signal_audio, sr = librosa.load(audio_file, sr=None)
time = np.linspace(0, len(signal_audio)/sr, len(signal_audio))
# Create high frequency noise
f_bruit = 1000
amplitude_bruit = 0.5
bruit_haute_frequence = amplitude_bruit * np.sin(2 * np.pi * f_bruit * time)
# Create noisy signal
signal_bruite = signal_audio + bruit_haute_frequence



signal_bruit_int16 = np.int16(np.clip(signal_bruite, -1, 1) * 32767)
pygame.mixer.init(frequency=44100)
sound = pygame.mixer.Sound(buffer=signal_bruit_int16.tobytes())
sound.play()
Cody
while pygame.mixer.get_busy():
    pygame.time.Clock().tick(10)


# Save the noisy signal as a new WAV file
output_file = "code2.wav"
sf.write(output_file, sound, sr)
print(f"Noisy audio saved as: {output_file}")
```

3. Traitement audio complet avec bruitage haute fréquence (1 kHz) et filtrage passe-bas (Butterworth 4ème ordre, fc=500 Hz), incluant l'enregistrement du résultat.

Résultats attendus:

1. **Sorties audio:**
    - Signal original contaminé par un bruit à 1 kHz (audible comme un sifflement aigu)
    - Signal filtré avec atténuation marquée au-dessus de 500 Hz
    - Fichier WAV enregistré ("code2.wav") contenant le signal filtré
2. **Effets audibles:**
    - Avant filtrage : Présence distincte du bruit à 1 kHz
    - Après filtrage :
        - Élimination quasi-complète du bruit ajouté
        - Atténuation partielle des composantes fréquentielles >500 Hz du signal original
3. **Validation:**
    - Vérifier dans le spectre :
        - Forte atténuation à 1 kHz (>20dB)
        - Transition progressive entre 400-600 Hz
    - Contrôle audio :
        - Disparition du sifflement parasite
        - Conservation des fréquences graves/médiums

code5.py

```python
import librosa
import numpy as np
import pygame
from scipy.signal import butter,filtfilt
import soundfile as sf  # enrigistrer WAV files
audio_file = "sample-1.wav"
sound = audio_file
# Load audio file
signal_audio, sr = librosa.load(audio_file, sr=None)
time = np.linspace(0, len(signal_audio)/sr, len(signal_audio))
# Create high frequency noise
f_bruit = 1000
amplitude_bruit = 0.5
bruit_haute_frequence = amplitude_bruit * np.sin(2 * np.pi * f_bruit * time)
# Create noisy signal
signal_bruite = signal_audio + bruit_haute_frequence

cuttoff_frequency = 500
order = 4
```

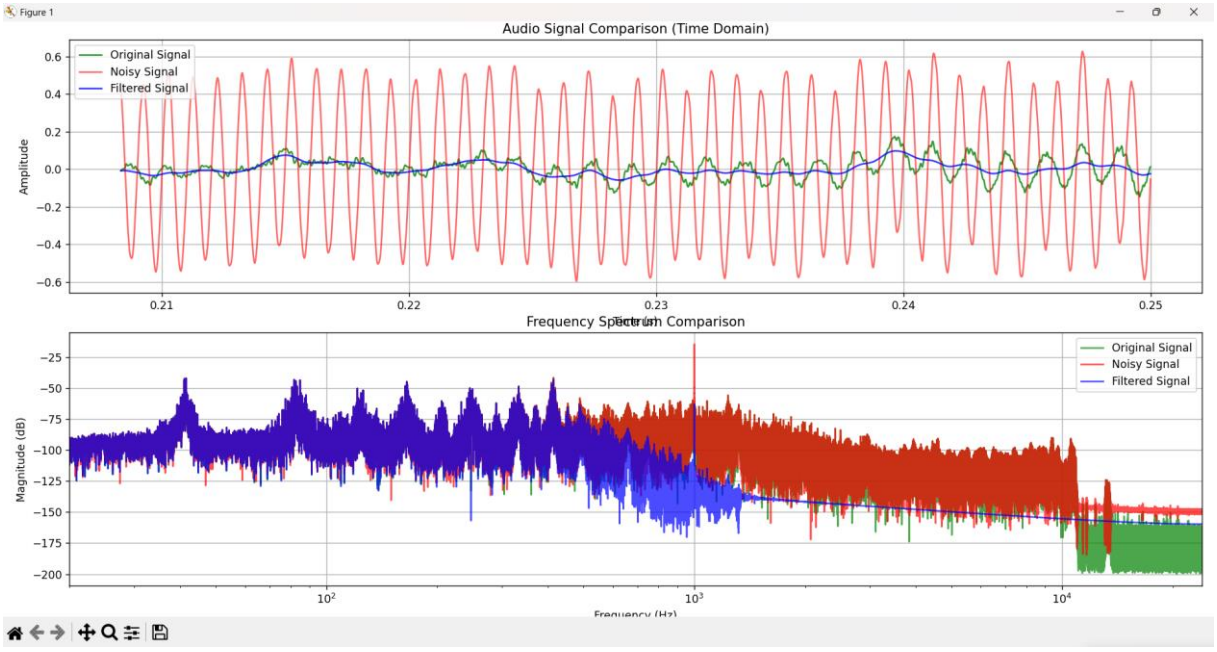
```
def butter_lowpass(cutoff, fs, order=5):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    from scipy.signal import butter
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    return b, a
b,a = butter_lowpass(cuttoff_frequency,sr, order)
signal_filter = filtfilt(b,a,signal_bruite) #filtre du signal bruité

signal_filter_int16 = np.int16(np.clip(signal_filter, -1, 1) * 32767)
pygame.mixer.init(frequency=sr)
sound = pygame.mixer.Sound(buffer=signal_filter_int16.tobytes())
sound.play()
Cody
while pygame.mixer.get_busy():
    pygame.time.Clock().tick(10)
output_file = "code2.wav"
sf.write(output_file, sound, sr)
print(f"Noisy audio saved as: {output_file}")
```

Création une visualisation montrant le signal audio d'origine, le signal bruyant et le signal filtré

```
import librosa
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import butter, filtfilt, freqz


audio_file = "sample-1.wav"
signal_audio, sr = librosa.load(audio_file, sr=None)
time = np.linspace(0, len(signal_audio)/sr, len(signal_audio))


f_bruit = 1000
amplitude_bruit = 0.5
bruit_haute_frequence = amplitude_bruit * np.sin(2 * np.pi * f_bruit * time)
          Alt+D to Document
signal_bruite = signal_audio + bruit_haute_frequence


cutoff_frequency = 500
order = 4
Cody
def butter_lowpass(cutoff, fs, order=5):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    return b, a


b, a = butter_lowpass(cutoff_frequency, sr, order)
signal_filter = filtfilt(b, a, signal_bruite)
```

```python
segment_length = 2000
start_index = 10000

end_index = min(start_index + segment_length, len(signal_audio))
segment_time = time[start_index:end_index]

plt.plot(segment_time, signal_audio[start_index:end_index], 'g-', label='Original Signal', alpha=0.8)
plt.plot(segment_time, signal_bruite[start_index:end_index], 'r-', label='Noisy Signal', alpha=0.6)
plt.plot(segment_time, signal_filter[start_index:end_index], 'b-', label='Filtered Signal', alpha=0.8)
plt.title('Audio Signal Comparison (Time Domain)')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.legend()
plt.grid(True)


plt.subplot(2, 1, 2)

Cody
def plot_spectrum(signal, sr, label, color, alpha=0.7):
    N = len(signal)
    freq = np.fft.rfftfreq(N, d=1/sr)
    spectrum = np.abs(np.fft.rfft(signal))/N
    plt.semilogx(freq, 20 * np.log10(spectrum + 1e-10), color=color, label=label, alpha=alpha)
    return freq, spectrum

plot_spectrum(signal_audio, sr, 'Original Signal', 'g')
plot_spectrum(signal_bruite, sr, 'Noisy Signal', 'r')
plot_spectrum(signal_filter, sr, 'Filtered Signal', 'b')

plt.title('Frequency Spectrum Comparison')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude (dB)')
plt.xlim(20, sr/2)
plt.legend()
plt.grid(True)

plt.tight_layout();plt.show()
```

Résultat avec cutoff_frequency = 500



Résultat avec cutoff_frequency = 1000