coerchon_habasque_dlts_tp1

16 Octobre 2024

[9]:
```
from IPython.core.display import HTML
HTML("""
<style>
.consignes{
  font-weight: bold;
  color: #3256a8;
  background-color: #edebdf
}
</style>
""")
```

[9]:
```
<IPython.core.display.HTML object>
```

#

Deep Learning et traitement du signal TP 1

L'objectif de ce TP est de prendre en main les outils de modélisation et d'analyse du signal présentés dans le premier cours et d'introduire la problématique de détection dont on parlera dans le cours numéro 3

Deadline : 16 octobre 2024, 13h59, par mail à deepetsignal.mva@gmail.com Effort estimé : 2 à 3 heures maximum

Le rendu de ce TP n'est pas obligatoire. Il permet d'obtenir un bonus de 1 (minimum syndical) à 3 (votre notebook servira de correction l'an prochain) points sur la moyenne des TP

Listez les noms des étudiants (2 au maximum) ayant participé à ce notebook dans la cellule suivante (prénom, nom). Au moment du rendu, le notebook doit être nommé nom1_nom2_dlts_tp1.ipynb

Colin Coërchon et Chloé Habasque

Si vous installez des paquets supplémentaires, merci de les lister dans la cellule suivante avec la syntaxe

```
!pip install \< nom_du_paquet \>
```

[ ]:
```
```

[10]:
```
import numpy as np
import matplotlib.pylab as plt
import scipy
```

```
import IPython.display as ipd
```

##

Partie 1: Audio

###

Visualisation

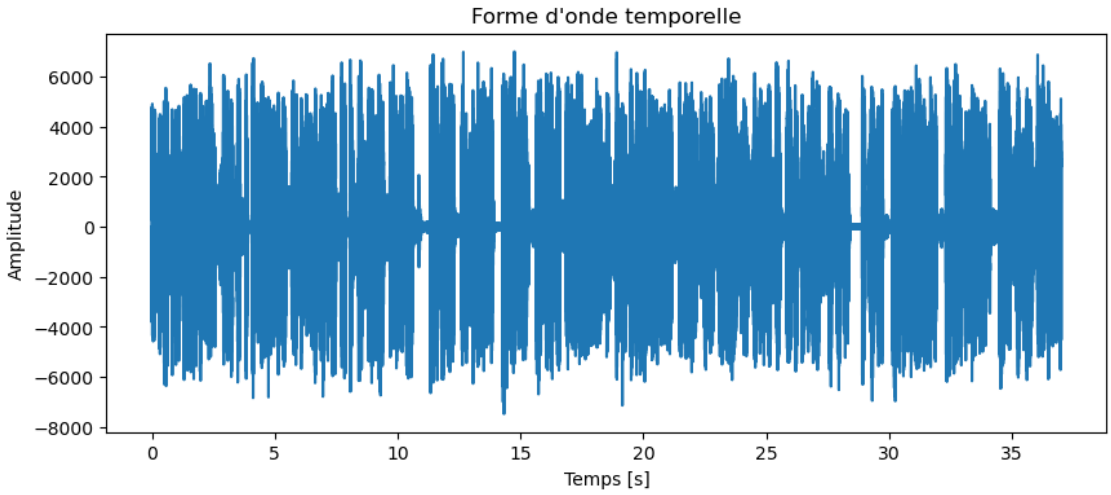
Enregistrez un fichier wav de quelques secondes de votre voix. Importez le avec scipy.io.wavfile.read et écoutez le dans le notebook avec IPython.display.Audio

[11]:
```
# Nous avons finalement choisi d'importer un fichier audio de Francis Huster.
# (Cet audio est plutôt drôle malgré les sujets abordés assez lourds, et il␣
 ↪s'est beaucoup retrouvé dans les tendances Twitter de ces derniers jours.)

fe, data = scipy.io.wavfile.read('FrancisHuster_extract.wav')

# Écouter l'extrait
ipd.Audio('FrancisHuster_extract.wav')
```

[11]:
```
<IPython.lib.display.Audio object>
```

Visualisez la forme d'onde temporelle de ce signal audio. Estimez et affichez sa Densité Spectrale de Puissance. Donnez une interprétation de ce que vous observez.

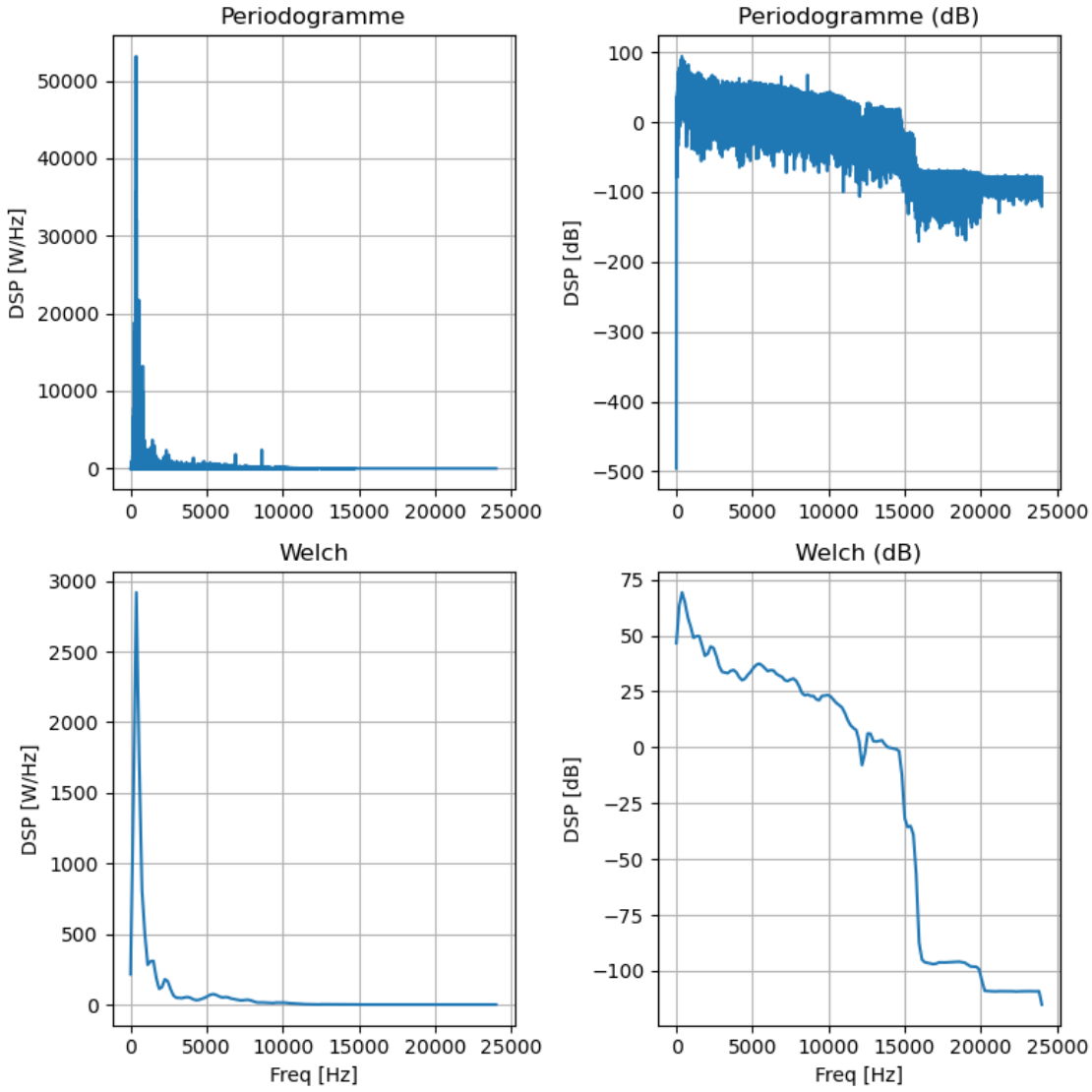
[12]:
```
duree = len(data) / fe  # Durée en secondes
time = np.linspace(0, duree, len(data))

plt.figure(figsize=(10, 4))
plt.plot(time, data)
plt.title('Forme d\'onde temporelle')
plt.xlabel('Temps [s]')
plt.ylabel('Amplitude')
plt.show()
```



On essaye ensuite d'estimer la Densité Spectrale de Puissance (DSP) du signal. Pour cela, on fait 2 méthodes. - Tracer le périodogramme - Utiliser la méthode de Welch

[13]:
```
fw, welch = scipy.signal.welch(data, fe)  # Méthode de Welch
ff, S = scipy.signal.periodogram(data, fe)  # Périodogramme

plt.figure(figsize=(8, 8))

# Sous-graphe 1 : Périodogramme (DSP en W/Hz)
plt.subplot(221)
plt.plot(ff, np.abs(S))
plt.ylabel('DSP [W/Hz]')
plt.title('Periodogramme')
plt.grid('on')

# Sous-graphe 2 : Périodogramme (DSP en dB)
plt.subplot(222)
plt.plot(ff, 20 * np.log10(np.abs(S)))
plt.ylabel('DSP [dB]')
plt.title('Periodogramme (dB)')
plt.grid('on')

# Sous-graphe 3 : Méthode de Welch (DSP en W/Hz)
plt.subplot(223)
plt.plot(fw, np.abs(welch))
plt.ylabel('DSP [W/Hz]')
plt.xlabel('Freq [Hz]')
plt.title('Welch')
plt.grid('on')
```

```python
# Sous-graphe 4 : Méthode de Welch (DSP en dB)
plt.subplot(224)
plt.plot(fw, 20 * np.log10(np.abs(welch)))
plt.xlabel('Freq [Hz]')
plt.ylabel('DSP [dB]')
plt.title('Welch (dB)')
plt.grid('on')

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



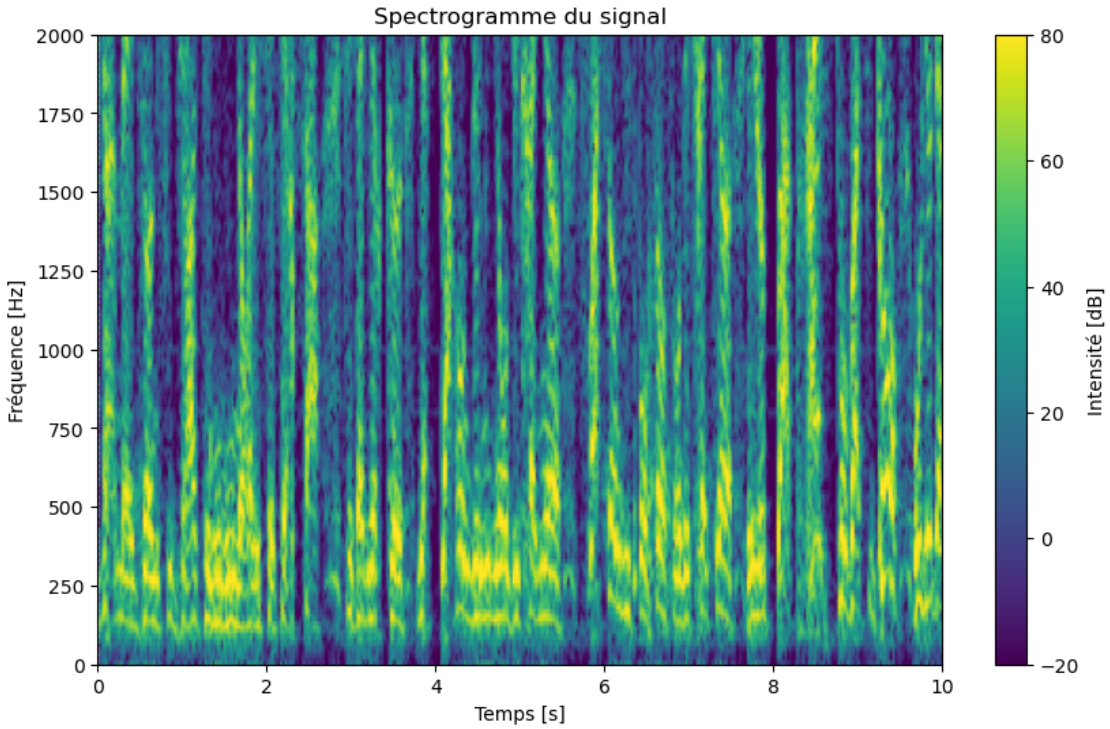
- On remarque que la fréquence maximale se situe proche de 24 kHz. C'est plutôt logique quand on remarque que la fréquence d'échantillonnage choisie est de 48 kHz. Cela corrobore bien le

théorème de Nyquist-Shannon.

- De plus, la majorité du spectre fréquentiel se situe en dessous de 10 à 15 kHz, avec des amplitudes de plus de 30 dB dans la DSP. Cela est tout à fait normal puisque nous étudions un signal d'une voix humaine.
- Les fréquences au dessus de 15 kHz sont très peu présentes (amplitudes vers les -100 dB), cela pourrait davantage s'apparenter à du bruit.

Calculez et affichez le spectrogramme du signal. Justifiez du choix des réglages que vous avez faits. Sélectionnez une portion pertinente du spectrogramme pour estimer visuellement le pitch de votre voix.

[14]:
```
duree_segment= 0.05 # sec
nperseg= int(duree_segment *fe)
noverlap = nperseg // 2  # Chevauchement de 50%

# Calculer le spectrogramme
frequencies, times, Sxx = scipy.signal.spectrogram(data, fs=fe, nperseg=nperseg,␣
 ↪noverlap=noverlap)

# Afficher le spectrogramme
plt.figure(figsize=(10, 6))
plt.pcolormesh(times, frequencies, 20 * np.log10(Sxx), vmax = 80,  vmin=-20,␣
 ↪shading='gouraud')
plt.colorbar(label='Intensité [dB]')
plt.ylabel('Fréquence [Hz]')
plt.xlabel('Temps [s]')
plt.title('Spectrogramme du signal')
plt.xlim(0, 10)
plt.ylim(0, 2000)
plt.show()
```



Notre audio est un peu long (37 secondes). Nous avons donc pris uniquement les 10 premières secondes pour un meilleur rendu visuel du spectrogramme. Après différents essais, nous avons finalement affiché le spectrogramme uniquement pour les fréquences de moins de 2000 Hz et pris 0.5 secondes pour la durée d'un segment. Nous avons également pris `noverlap = nperseg // 2` pour lisser les transitions entre les fenêtres.

Visuellement, on trouve un pitch (la fréquence fondamentale) vers les 200 Hz.

###

Calcul du Pitch

Proposez une méthode simple pour estimer automatiquement le Pitch de votre voix (cette méthode ne doit pas faire intervenir d'implémentations externes).

Utilisez cette méthode pour estimer les variations du Pitch le long du signal, estimez le pitch toutes les 20 ms et présentez vos résultats sous forme visuelle.

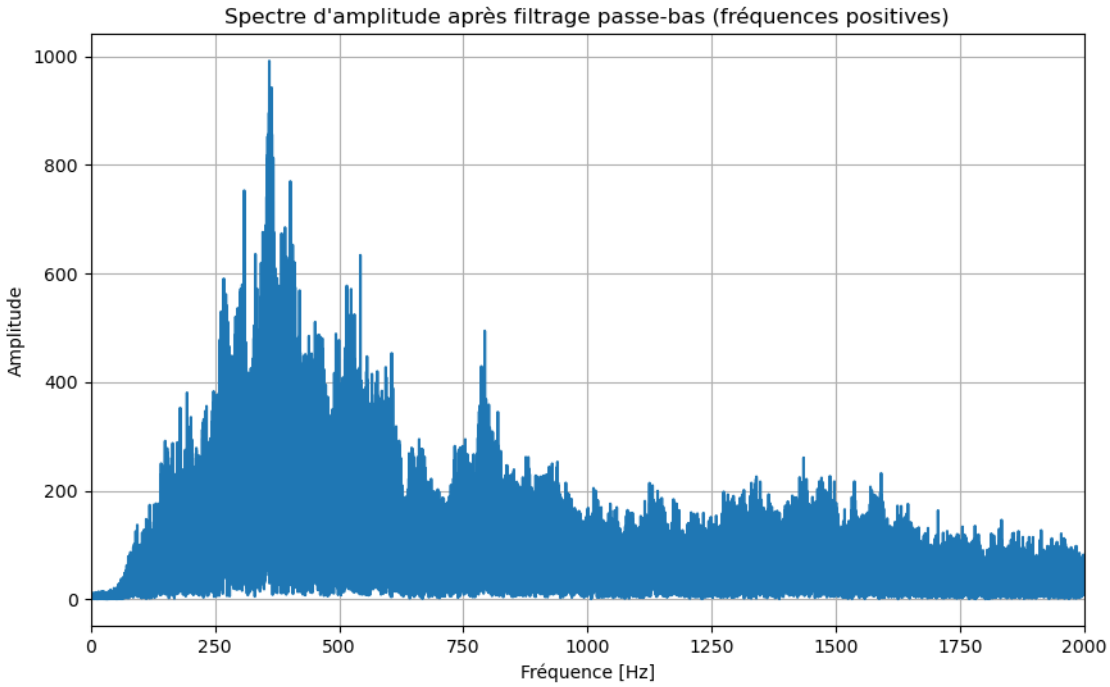
Ne pas utiliser de méthode "toute faite" que vous pourriez par exemple trouver dans la bibliothèque librosa.

Commentez vos résultats.

[15]:
```
# On fait d'abord un filtre passe-bas, car il y a beaucoup de bruit au dessus de␣
↪15 kHz dans notre signal.

def butter_lowpass(cutoff, fe, order=5):
```

```
    fmax = 0.5 * fe  # Fréquence de Nyquist
    normal_cutoff = cutoff / fmax  # Normalisation de la fréquence de coupure
    b, a = scipy.signal.butter(order, normal_cutoff, btype='low', analog=False)
    return b, a

def butter_lowpass_filter(data, cutoff, fs, order=5):
    b, a = butter_lowpass(cutoff, fs, order=order)
    y = scipy.signal.filtfilt(b, a, data)
    return y

cutoff_frequency = 10000
order = 4

# On filtre le signal extrait (avec notre filtre passe-bas)
filtered_data = butter_lowpass_filter(data, cutoff_frequency, fe, order)

# Calcul de la FFT sur le signal filtré
fft_spectrum = np.fft.fft(filtered_data) / fe
frequencies = np.fft.fftfreq(len(filtered_data), d=1/fe)

# On prend uniquement les valeurs positives
positive_frequencies = frequencies[frequencies >= 0]
positive_amplitude_spectrum = np.abs(fft_spectrum[frequencies >= 0])

# On cherche la fréquence fondamentale (fréquence avec la plus grande amplitude)
index_max = np.argmax(positive_amplitude_spectrum[1:]) + 1  # On exclut la␣
 ↪composante continue (DC)
fundamental_freq_estimated = positive_frequencies[index_max]

print(f"La fréquence fondamentale estimée est : {fundamental_freq_estimated:.2f}␣
 ↪Hz")

plt.figure(figsize=(10, 6))
plt.plot(positive_frequencies, positive_amplitude_spectrum)
plt.xlim(0, 2000)
plt.title("Spectre d'amplitude après filtrage passe-bas (fréquences positives)")
plt.xlabel("Fréquence [Hz]")
plt.ylabel("Amplitude")
plt.grid(True)
plt.show()
```

```
La fréquence fondamentale estimée est : 359.35 Hz
```



On effectue en premier lieu un filtre passe bas pour enlever le bruit très présent au dessus de 15 kHz. Puis, on calcule la transformée de Fourier du signal et ne prend QUE les valeurs positives des coefficients de la FFT.

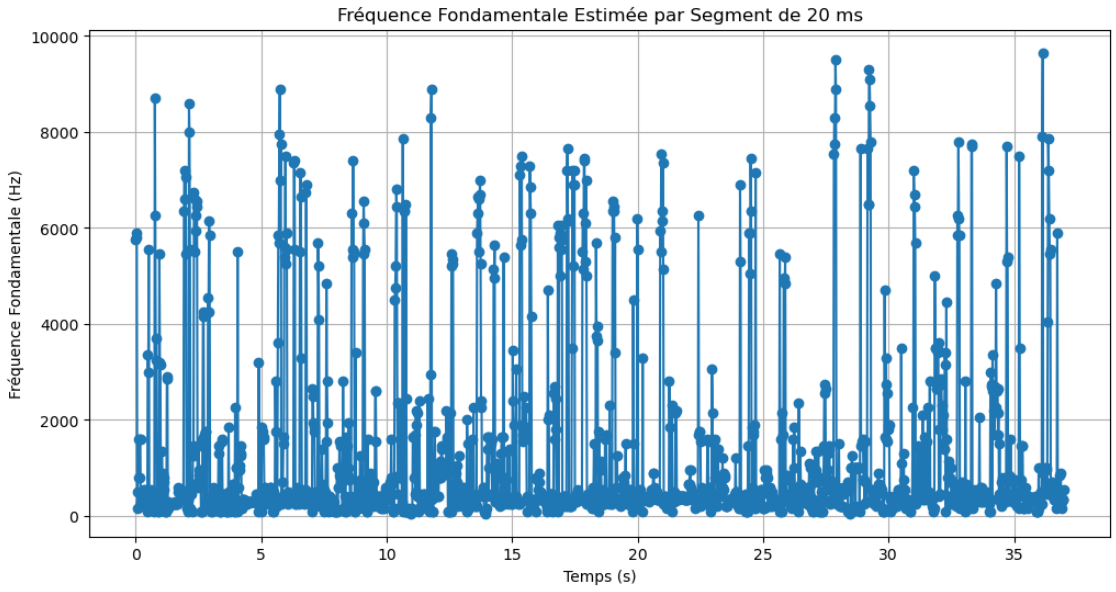
On trouve finalement un pitch à 359 Hz. Ce résultat nous semble un peu bizarre puisqu'une voix d'homme a une fréquence fondamentale proche de 100 à 150 Hz.

On passe maintenant à la découpe du signal en segments de 20 ms.

[16]:
```
segment_duration = 0.02  # Durée du segment en secondes (20 ms)
segment_samples = int(segment_duration * fe)  # Nombre d'échantillons par segment

num_segments = len(filtered_data) // segment_samples

fundamental_frequencies = []

for i in range(num_segments):
    # Découper le signal
    segment = filtered_data[i * segment_samples:(i + 1) * segment_samples]

    # Appliquer la FFT
    N = len(segment)
    fft_result = np.fft.fft(segment)
    freq = np.fft.fftfreq(len(fft_result), d=1/fe)
    amplitude = np.abs(fft_result)
```

```
    half_N = N // 2
    freq = freq[:half_N]
    amplitude = amplitude[:half_N]

    # Trouver la fréquence fondamentale
    index_max = np.argmax(amplitude[1:]) + 1  # On exclut le 0 en début de␣
 ↪spectre

    fundamental_freq = np.abs(freq[index_max])  # On prend la valeur absolue de␣
 ↪la fréquence
    fundamental_frequencies.append(fundamental_freq)

# Affichage des moyennes des fréquences filtrées
print(f"Fréquence moyenne (filtrée < 1000 Hz) : {np.
 ↪mean(fundamental_frequencies):.2f} Hz")

# Visualisation
plt.figure(figsize=(12, 6))
plt.plot(np.arange(len(fundamental_frequencies)) * segment_duration,␣
 ↪fundamental_frequencies, marker='o')
plt.title('Fréquence Fondamentale Estimée par Segment de 20 ms')
plt.xlabel('Temps (s)')
plt.ylabel('Fréquence Fondamentale (Hz)')
plt.grid()
plt.show()
```

```
Fréquence moyenne (filtrée < 1000 Hz) : 1253.49 Hz
```

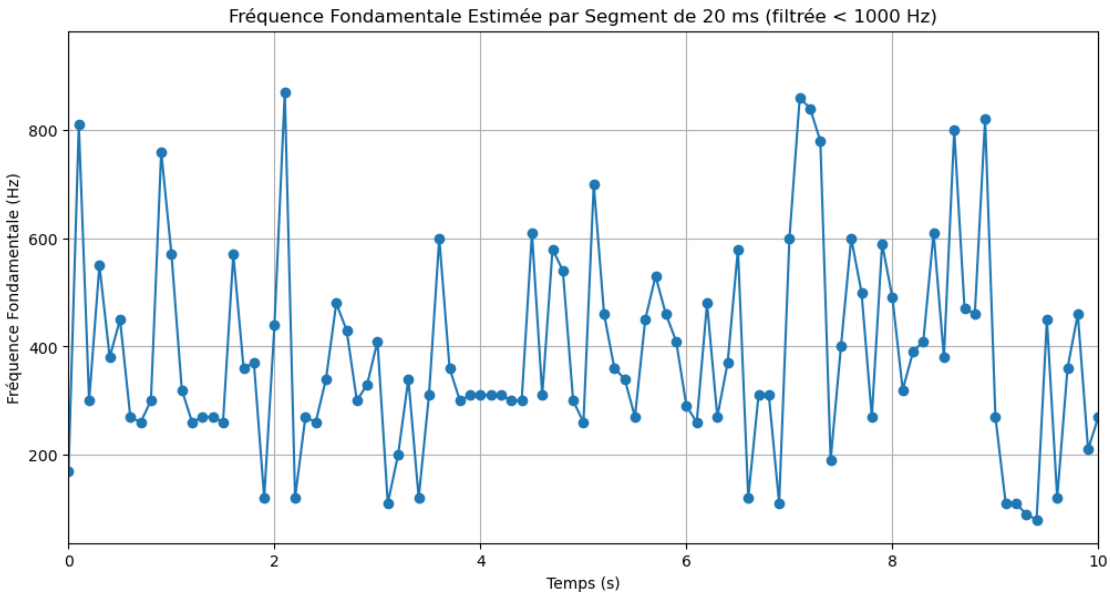


Il y a trop de valeurs aberrantes (au dessus de 1000 ou 2000 Hz), alors que l'on remarque que la majorité des valeurs se situent vers les 350 Hz. Nous avons donc choisi d'effectuer le même calcul, mais en supprimant toutes les valeurs au dessus de 1000 Hz, jugées non pertinentes.

[17]:
```
segment_duration = 0.1  # Durée du segment en secondes (20 ms)
segment_samples = int(segment_duration * fe)  # Nombre d'échantillons par segment

num_segments = len(filtered_data) // segment_samples

fundamental_frequencies = []

for i in range(num_segments):
    # Découper le signal
    segment = filtered_data[i * segment_samples:(i + 1) * segment_samples]

    # Appliquer la FFT
    N = len(segment)
    fft_result = np.fft.fft(segment)
    freq = np.fft.fftfreq(len(fft_result), d=1/fe)
    amplitude = np.abs(fft_result)

    half_N = N // 2
    freq = freq[:half_N]
    amplitude = amplitude[:half_N]

    # Trouver la fréquence fondamentale
    index_max = np.argmax(amplitude[1:]) + 1  # On exclut le 0 en début de␣
 ↪spectre

    fundamental_freq = np.abs(freq[index_max])  # On prend la valeur absolue de␣
 ↪la fréquence

    # Filtrer les fréquences au-dessus de 1000 Hz
    if fundamental_freq < 1000:
        fundamental_frequencies.append(fundamental_freq)

# Affichage des moyennes des fréquences filtrées
print(f"Fréquence moyenne (filtrée < 1000 Hz) : {np.
 ↪mean(fundamental_frequencies):.2f} Hz")

# Visualisation
plt.figure(figsize=(12, 6))
plt.plot(np.arange(len(fundamental_frequencies)) * segment_duration,␣
 ↪fundamental_frequencies, marker='o')
plt.title('Fréquence Fondamentale Estimée par Segment de 20 ms (filtrée < 1000␣
 ↪Hz)')
plt.xlabel('Temps (s)')
```

```
plt.xlim(0,10)
plt.ylabel('Fréquence Fondamentale (Hz)')
plt.grid()
plt.show()
```

```
Fréquence moyenne (filtrée < 1000 Hz) : 416.35 Hz
```



(Seulement les 10 premières secondes du signal sont affichées.) On trouve donc, en moyenne, un pitch à 381 Hz. On remarque par ailleurs que les coefficients de la transoformée de Fourier sont échantillonnées de 50 Hz en 50 Hz, car on a pris une longueur de segment très faible (20 ms).

##

Partie 2: Détection d'impulsion

Un signal de durée 1 seconde et échantillonné à 1000 Hz est composé d'un bruit blanc gaussien de puissance inconnue et éventuellement d'une impulsion à une fréquence f0 comprise entre 100 et 200 Hz. Une série de 1000 signaux est enregistrée dans le fichier signaux_impulsions.npz. Ouvez ce fichier avec numpy (cf code plus bas). Le fichier contient une tableau signaux 10000 x 1000 dont chaque ligne contient un signal de durée 1000. Le fichiez contient aussi un tableau labels de taille 10000 dont la ligne i est à TRUE si le signal i contient une impulsion et à 0 sinon.

[18]:
```
donnees = np.load('signaux_impulsions.npz')
signaux = donnees['data']
labels = donnees['labels']
```

##

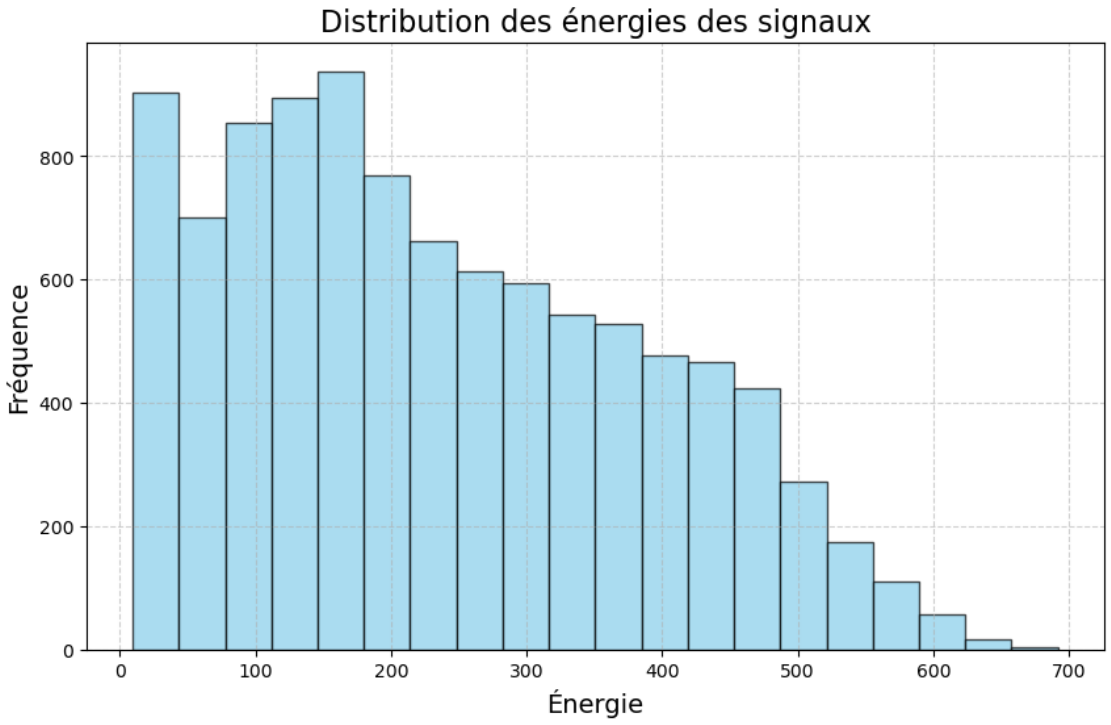
Première méthode

Proposez une méthode simple pour décider si un signal contient une impulsion ou non à partir du calcul de l'énergie du signal. Cette méthode fera intervenir un seuil:

```
def contient_impulsion_energie(signal: np.ndarray, seuil: float) -> bool:
```

[19]:
```
def contient_impulsion_energie(signal: np.ndarray, seuil: float) -> bool:

    energie = np.sum(signal ** 2)
    if energie > seuil:
        return 1
    else:
        return 0
```

[20]:
```
energies = []
for signal in signaux:
    energie = np.sum(signal ** 2)
    energies.append(energie)

plt.figure(figsize=(10, 6))
plt.hist(energies, bins=20, color='skyblue', edgecolor='black', alpha=0.7)

plt.title('Distribution des énergies des signaux', fontsize=16)
plt.xlabel('Énergie', fontsize=14)
plt.ylabel('Fréquence', fontsize=14)
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```



Utilisez votre méthode avec un seuil fixé sur tous les signaux pour prédire s'ils contiennent une impulsion ou non.

[21]:
```
seuil = 200
predictions = []
for signal in signaux:
    contient_imp = contient_impulsion_energie(signal, seuil)
    predictions.append(contient_imp)
```

Calculez: - VP = le nombre de signaux que vous détectez comme contenant une impulsion qui contiennet effectivement une impulsion - FP = le nombre de signaux que vous détectez comme contenant une impulsion qui ne contiennet en fait PAS une impulsion - VN = le nombre de signaux que vous détectez comme ne contenant PAS une impulsion qui ne contiennet effectivement PAS une impulsion - FN = le nombre de signaux que vous détectez comme ne contenant PAS une impulsion mais qui contiennet en fait une impulsion

[22]:
```
VP_eng=0
FP_eng=0
VN_eng=0
FN_eng=0
for i in range (len(predictions)):
    if predictions[i] == 1:
        if labels[i] == predictions[i]:
            VP_eng+=1
        else:
            FP_eng+=1
    else:
        if labels[i] == predictions[i]:
            VN_eng+=1
        else:
            FN_eng+=1

print(f"VP (Vrai Positif): {VP_eng}")
print(f"FP (Faux Positif): {FP_eng}")
print(f"VN (Vrai Négatif): {VN_eng}")
print(f"FN (Faux Négatif): {FN_eng}")
```

```
VP (Vrai Positif): 3154
FP (Faux Positif): 2118
VN (Vrai Négatif): 2878
FN (Faux Négatif): 1850
```

##

Deuxième méthode

Proposez une deuxième méthode faisant par exemple intervenir le spectrogramme du signal. Cette méthode fera encore intervenir un seuil

```
def contient_impulsion_spectrogramme(signal: np.ndarray, seuil: float) -> bool:
```

[23]:
```
def contient_impulsion_spectrogramme(signal: np.ndarray, fe: int, seuil: float):
    nperseg = 256
    freqs, times, Sxx = scipy.signal.spectrogram(signal, fs=fe, nperseg=nperseg)

    Sxx_dB = 20 * np.log10(Sxx + 1e-10)  # Ajout d'un petit offset pour éviter␣
 ↪log(0)

    # On se concentre uniquement sur la plage de fréquences 100 Hz - 200 Hz␣
 ↪(comme dit dans l'énoncé)
    freq_range = (freqs >= 100) & (freqs <= 200)
    Sxx_dB_filtered = Sxx_dB[freq_range, :]

    # On vérifie si une valeur du spectrogramme dans cette plage dépasse le seuil
    if np.any(Sxx_dB_filtered > seuil):
        return True
    else:
        return False
```

Calculez pour cette nouvelle méthode, pour un certain seuil les valeurs de VP, FP, VN, FN

[24]:
```
# Paramètres
seuil = -50  # Exemple de seuil en dB, à ajuster
fe = 1000  # Fréquence d'échantillonnage

predictions_spectrogramme = []
for signal in signaux:
    contient_imp = contient_impulsion_spectrogramme(signal, fe, seuil)
    predictions_spectrogramme.append(contient_imp)

# Calcul des métriques (VP, FP, VN, FN)
VP_spectro, FP_spectro, VN_spectro, FN_spectro = 0, 0, 0, 0

for i in range(len(predictions_spectrogramme)):
    if predictions_spectrogramme[i] == 1:
        if labels[i] == 1:
            VP_spectro += 1
        else:
            FP_spectro += 1
    else:
        if labels[i] == 0:
            VN_spectro += 1
        else:
            FN_spectro += 1

# Afficher les résultats
print(f"VP (Vrai Positif) : {VP_spectro}")
```

```
print(f"FP (Faux Positif) : {FP_spectro}")
print(f"VN (Vrai Négatif) : {VN_spectro}")
print(f"FN (Faux Négatif) : {FN_spectro}")
```

```
VP (Vrai Positif) : 4569
FP (Faux Positif) : 1176
VN (Vrai Négatif) : 3820
FN (Faux Négatif) : 435
```

##

Comparaison des méthodes

Pour une méthode de détection et un seuil donné, la précision est définie comme:

$$\frac{\sharp\text{Signaux détectés comme contenant une impulsion qui en contiennent effectivement une}}{\sharp\text{Signaux détectés comme positifs}}$$

et le rappel comme:

$$\frac{\sharp\text{Signaux détectés comme contenant une impulsion qui en contiennent effectivement une}}{\sharp\text{Signaux contenant une impulsions}}$$

[25]:
```
def calculer_precision_rappel(VP, FP, FN):
    if (VP + FP) > 0:
        precision = VP / (VP + FP)
    else:
        precision = 0.0

    if (VP + FN) > 0:
        rappel = VP / (VP + FN)
    else:
        rappel = 0.0

    return precision, rappel
```

[26]:
```
precision_eng, rappel_eng = calculer_precision_rappel(VP_eng, FP_eng, FN_eng)

print(f"Précision: {precision_eng:.2f}")
print(f"Rappel: {rappel_eng:.2f}")
```

```
Précision: 0.60
Rappel: 0.63
```

[27]:
```
precision_spect, rappel_spect = calculer_precision_rappel(VP_spectro,␣
↪FP_spectro, FN_spectro)

print(f"Précision: {precision_spect:.2f}")
print(f"Rappel: {rappel_spect:.2f}")
```

```
Précision: 0.80
Rappel: 0.91
```

Donnez une interprétation de ces deux métriques

Precision : Représente le pourcentage de signaux détectés comme contenant une impulsion et qui en contient effectivement une.

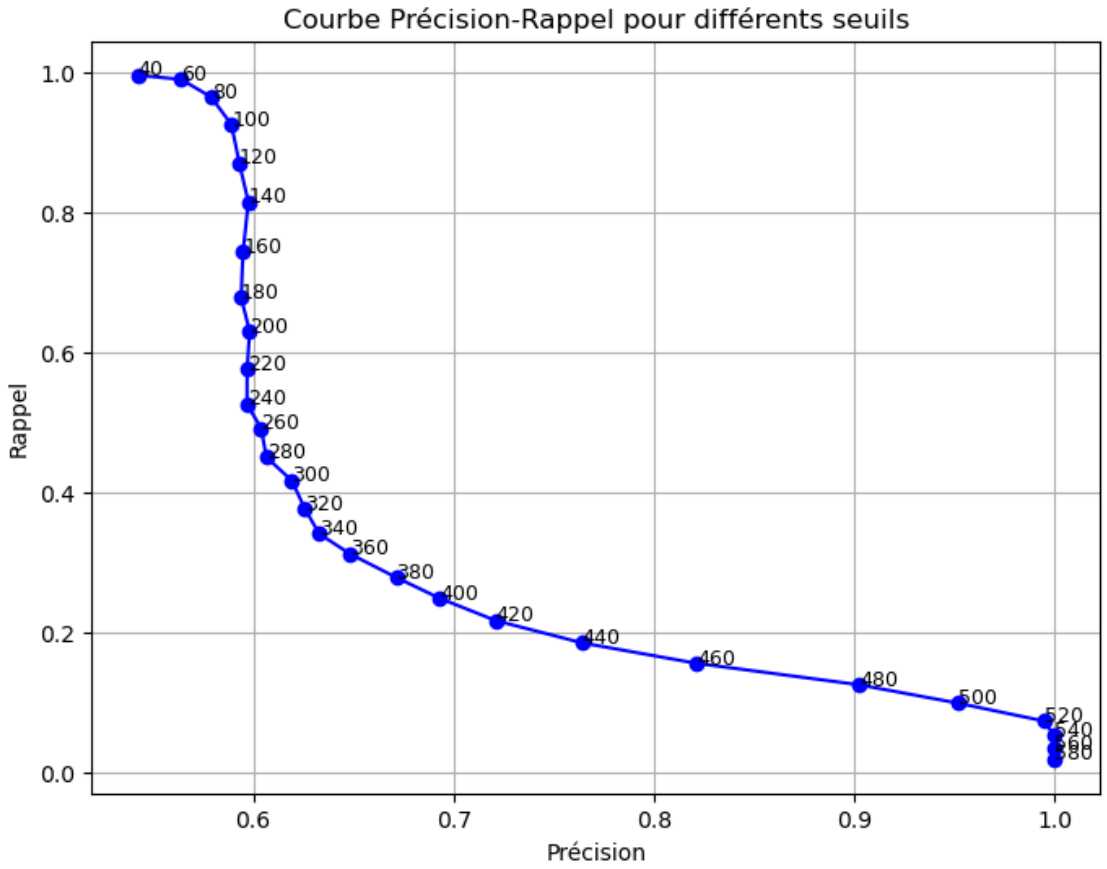
Rappel : Représente le pourcentage d'impulsions correctement détectées par rapport au nombre de signaux contenant des impulsions.

```
Pour chacune des deux méthodes proposées, faites varier le seuil sur une dizaine de valeurs et c
```

Affichez dans le plan (précision , rappel) les points de fonctionnement des deux méthodes pour différents seuils. Commentez le résultat.

[28]:
```python
### Première méthode

# Définir une gamme de seuils à tester
seuils = range(40, 600, 20)

precisions = []
rappels = []

for seuil in seuils:
    predictions = []

    for signal in signaux:
        contient_imp = contient_impulsion_energie(signal, seuil)
        predictions.append(contient_imp)

    # Calcul de VP, FP, VN, FN pour chaque seuil
    VP_eng, FP_eng, VN_eng, FN_eng = 0, 0, 0, 0
    for i in range(len(predictions)):
        if predictions[i] == 1:
            if labels[i] == predictions[i]:
                VP_eng += 1
            else:
                FP_eng += 1
        else:
            if labels[i] == predictions[i]:
                VN_eng += 1
            else:
                FN_eng += 1

    precision, rappel = calculer_precision_rappel(VP_eng, FP_eng, FN_eng)

    precisions.append(precision)
    rappels.append(rappel)
```

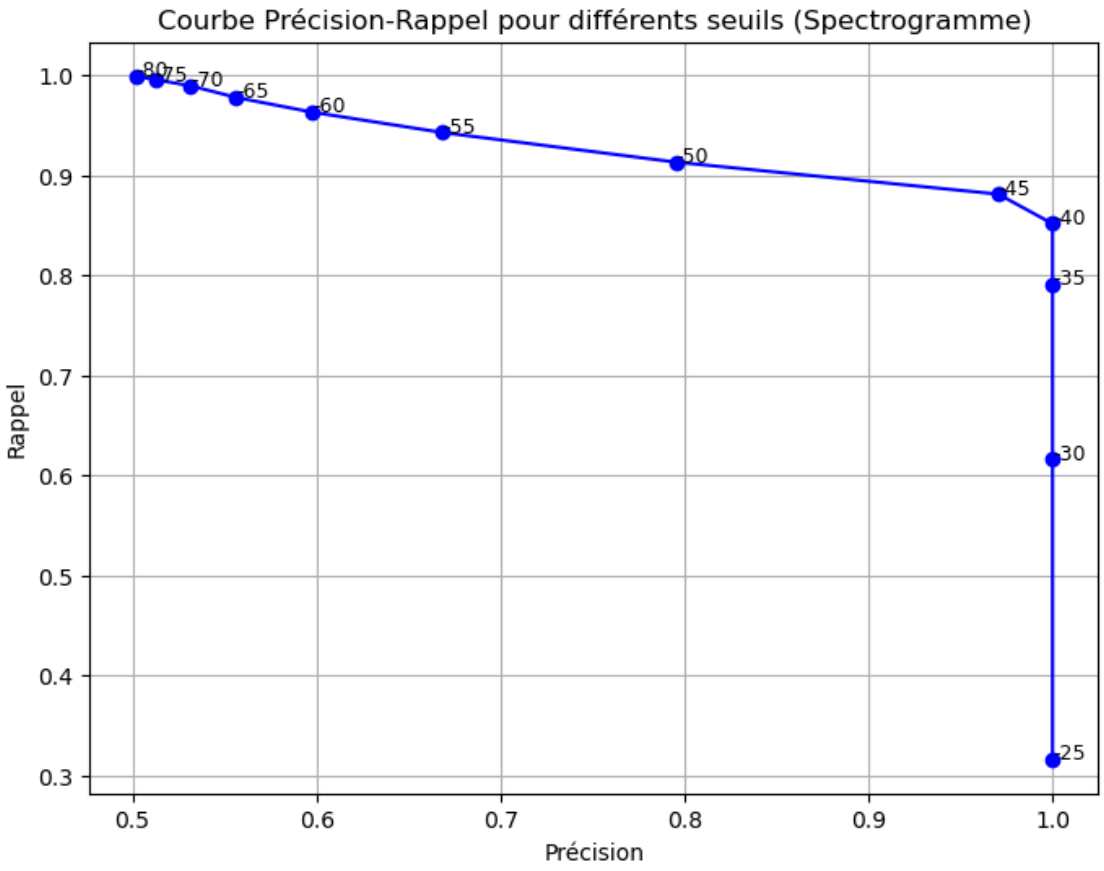
```
plt.figure(figsize=(8, 6))
plt.plot(precisions, rappels, marker='o', linestyle='-', color='b')

# Pour rendre tout ceci plus joli
for i, seuil in enumerate(seuils):
    plt.text(precisions[i], rappels[i], f"{seuil}", fontsize=9)

plt.title("Courbe Précision-Rappel pour différents seuils")
plt.xlabel("Précision")
plt.ylabel("Rappel")
plt.grid(True)
plt.show()
```



On cherche à maximiser en même temps la précision et le rappel. Visuellement, on peut prendre un seuil à 100 qui donne une precision de 59% pour un très bon rappel de 93%.

[29]:
```
### Deuxième méthode

# Définir une gamme de seuils à tester
seuils = range(-80, -20, 5)
```

```
precisions = []
rappels = []

for seuil in seuils:
    predictions = []

    for signal in signaux:
        contient_imp = contient_impulsion_spectrogramme(signal, fe, seuil)
        predictions.append(contient_imp)

    # Calcul de VP, FP, VN, FN pour chaque seuil
    VP_spectro, FP_spectro, VN_spectro, FN_spectro = 0, 0, 0, 0
    for i in range(len(predictions)):
        if predictions[i] == 1:
            if labels[i] == 1:
                VP_spectro += 1
            else:
                FP_spectro += 1
        else:
            if labels[i] == 0:
                VN_spectro += 1
            else:
                FN_spectro += 1

    precision, rappel = calculer_precision_rappel(VP_spectro, FP_spectro,␣
 ↪FN_spectro)

    precisions.append(precision)
    rappels.append(rappel)

plt.figure(figsize=(8, 6))
plt.plot(precisions, rappels, marker='o', linestyle='-', color='b')

# Pour rendre tout ceci plus joli
for i, seuil in enumerate(seuils):
    plt.text(precisions[i], rappels[i], f"{seuil}", fontsize=9)

plt.title("Courbe Précision-Rappel pour différents seuils (Spectrogramme)")
plt.xlabel("Précision")
plt.ylabel("Rappel")
plt.grid(True)
plt.show()
```



On cherche à maximiser en même temps la précision et le rappel. Visuellement, on peut prendre un seuil à -40 qui donne une precision de 100% pour un très bon rappel de 85%.

1 Conclusion

En testant différents seuils, nous avons pu identifier les valeurs optimales maximisant simultanément la précision et le rappel.

1.0.1 Première méthode (basée sur l'énergie) :

Avec un seuil à 100, nous obtenons un rappel de 93% et une précision de 59%. Cependant, pour améliorer la précision, il fallait fortement augmenter le seuil, ce qui réduisait le rappel. Un seuil entre 100 et 120 offrait le meilleur compromis.

1.0.2 Deuxième méthode (basée sur le spectrogramme) :

Les résultats obtenus avec cette méthode sont nettement supérieurs. En explorant une gamme de seuils entre -50 dB et -35 dB, les performances sont bien meilleures. Avec un seuil à -40 dB, nous atteignons par exemple une précision de 100% pour un rappel de 85%. Cette méthode est plus

adaptée grâce à son analyse temporelle et fréquentielle des impulsions, permettant de mieux isoler les impulsions du bruit.

Ainsi, la méthode spectrogramme se révèle plus performante et mieux adaptée à la structure du signal.