Rapport TP

TL Traitement Du Signal

Réalisé par:
Nabil ABDELOUAHED
Yann MARTIN

June 4, 2025

Contents

1 Partie 1: Quelques opérations de base sur les signaux

1.1 Signal numérique de synthèse

1.1.1 Génération du signal

Un signal sinusoïdal de fréquence f_0 est généré par la fonction suivante:

$$x[n] = \sin\left(2\pi f_0 \frac{n}{f_e}\right)$$

où f_e est la fréquence d'échantillonnage et N le nombre d'échantillons.

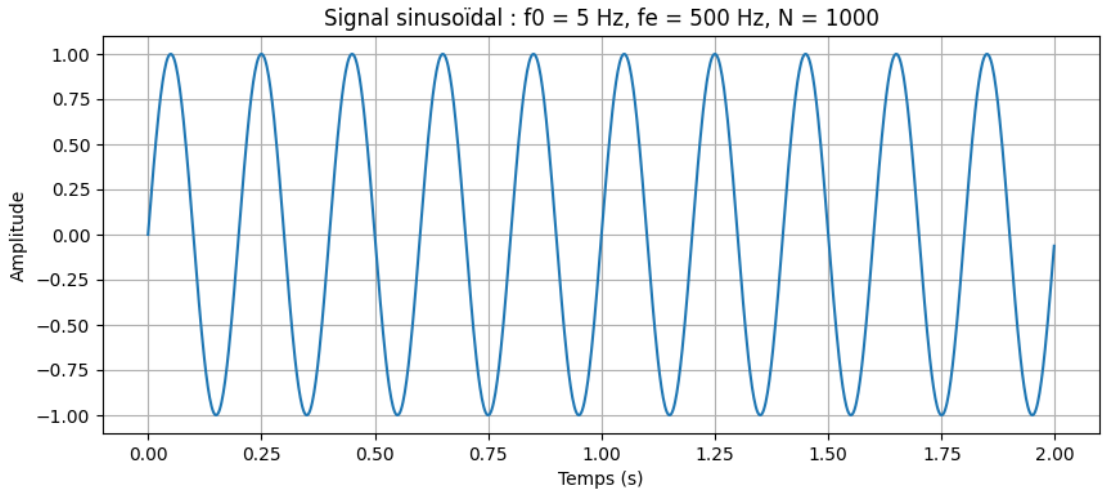


Figure 1: Signal sinusoïdal échantillonné

1.1.2 Énergie et puissance

L'énergie d'un signal discret $x[n]$ est donnée par :

$$E = \sum_{n=0}^{N-1} x[n]^2$$

Et la puissance moyenne par :

$$P = \frac{1}{N} \sum_{n=0}^{N-1} x[n]^2$$

Considérons un signal sinusoïdal discret de la forme :

$$x[n] = A \cdot \sin\left(2\pi f_0 \frac{n}{f_e}\right)$$

La puissance moyenne théorique d'un signal périodique est calculée par la formule:

$$P = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} x[n]^2$$

En utilisant l'identité trigonométrique :

$$\sin^2(\theta) = \frac{1 - \cos(2\theta)}{2}$$

on obtient :

$$x[n]^2 = A^2 \cdot \frac{1 - \cos\left(4\pi f_0 \frac{n}{f_e}\right)}{2}$$

Ainsi, la puissance devient :

$$P = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} A^2 \cdot \frac{1 - \cos\left(4\pi f_0 \frac{n}{f_e}\right)}{2} = \lim_{N\to\infty} \frac{A^2}{2} - \frac{A^2}{2N} \sum_{n=0}^{N-1} \cos\left(4\pi f_0 \frac{n}{f_e}\right) = \frac{A^2}{2}$$

Donc dans notre cas la puissance moyenne théorique est égale à 0.5.

Pour la puissance moyenne calculée numériquement pour le signal échantillonné on a la méme formule mais sans la limite:

$$P = \frac{A^2}{2} - \frac{A^2}{2N} \sum_{n=0}^{N-1} \cos\left(4\pi f_0 \frac{n}{f_e}\right)$$

Cas idéal : si N est un multiple entier de la période du signal (i.e., N couvre un nombre entier de périodes), alors la somme des cosinus s'annule :

$$\sum_{n=0}^{N-1} \cos\left(4\pi f_0 t\right) = 0 \quad \Rightarrow \quad P = \frac{A^2}{2}$$

Cas général : si N n'est pas un multiple exact de la période, la somme ne s'annule pas et on observe une légère déviation de la puissance par rapport à $\frac{A^2}{2}$. Cela est dû au fait que le signal est tronqué entre deux points non symétriques.

En variant N on trouve plusieurs valeurs de la puissance moyenne qui restent proches de la valeur théorique:

```
(.venv) deappool@deadpool-laptop:~/Desktop/td1_TS$ /home/deappool/De
puissance moyenne du signal echantillonné :  0.49999999999999895
(.venv) deappool@deadpool-laptop:~/Desktop/td1_TS$ /home/deappool/De
puissance moyenne du signal echantillonné :  0.5
(.venv) deappool@deadpool-laptop:~/Desktop/td1_TS$ /home/deappool/De
puissance moyenne du signal echantillonné :  0.5000000000000001
(.venv) deappool@deadpool-laptop:~/Desktop/td1_TS$ /home/deappool/De
puissance moyenne du signal echantillonné :  0.4999999999999995
```

Figure 2: Énergie et puissance du signal

1.1.3 Quantification

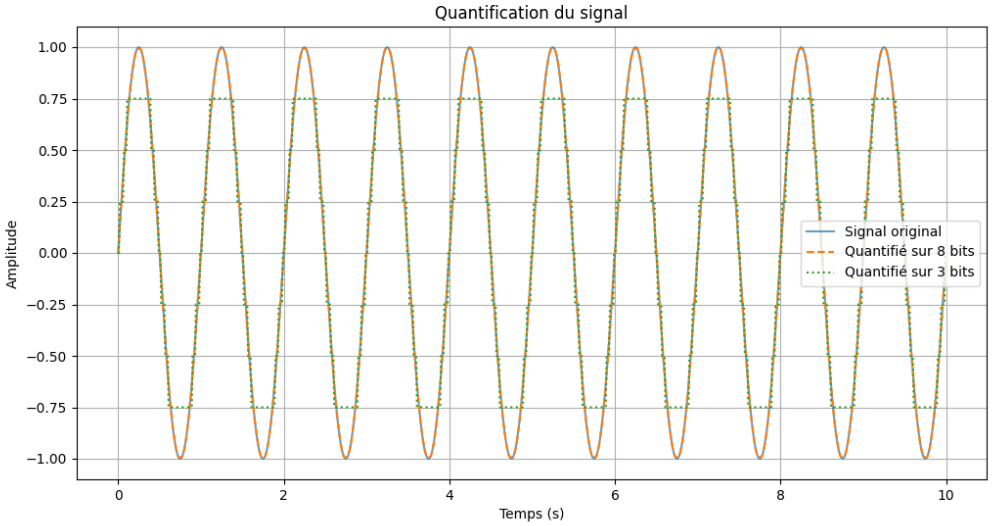


Figure 3: Quantification du signal à 3 et 8 bits

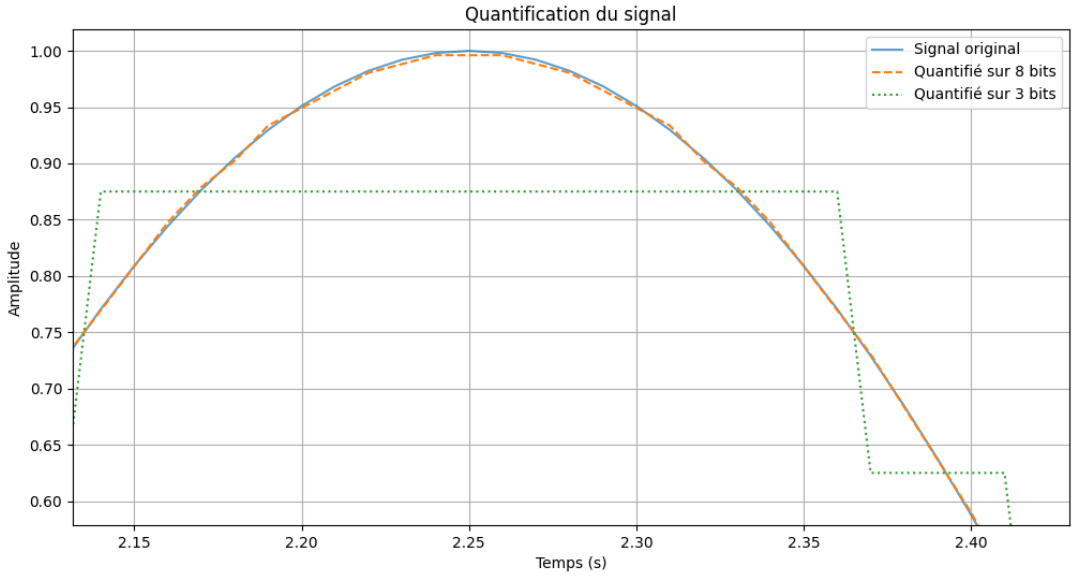


Figure 4: Zoom sur la quantification du signal à 3 et 8 bits

Pour la quantification à 3 bits, on retrouve bien 8 niveaux de quantification.
Le signal à 8 bits suit mieux la forme continue du signal d'origine mais on voit quand méme quelques erruers. À 3 bits, les marches sont plus visibles et le signal produit est significativement moins fidéle au signal d'origine.

SNR (Signal-to-Noise Ratio) :

$$\mathrm{SNR} = 10 \log_{10} \left(\frac{E_{\text{signal}}}{E_{\text{bruit}}} \right)$$

```
Énergie du bruit de quantification (8 bits): 0.006400
SNR (8 bits): 49,92 dB

Énergie du bruit de quantification (3 bits): 6.236655
SNR (3 bits): 19,82 dB
```

Figure 5: SNR pour chaque niveau de quantification

On remarque que l'energie du bruit est plus élevée pour le signal quantifié à 3 bits. Le résultat est logique car on voit sur le graphe que ce signal est plus éloigné du signal d'origine comparé au signal quantifié à 8 bits. On a la méme conclusion en raisonant sur le SNR: SNRq8 $>$ SNRq3.

1.2 Signal audio

1.2.1 Enregistrement

Les mots « Bonjour » et « ChatGpt » ont été enregistrés via Audacity (voir figure 6).

1.2.2 Restitution à différentes fréquences

L'audio est lu à f_e, $2f_e$ et $\frac{f_e}{2}$.
Effets observés :

- **Durée :** doubler la fréquence de restitution divise la durée par deux (voix accélérée), tandis que la diviser par deux double la durée (voix ralentie).

- **Hauteur :** multiplier la fréquence de restitution rend la voix plus aiguë (fréquences doublées), la diminuer la rend plus grave (fréquences divisées).

Lorsque la fréquence de restitution est trop grande ou trop petite comparée à la fréquence d'échantillonnage, le son devient incompréhensible. Le son est trop rapide ou trop lent, mais aussi on ne reconnaît plus les mots. Cela s'explique notamment par le déplacement des formants, c'est-à-dire des pics de résonance caractéristiques des voyelles et consonnes, qui changent de position dans le spectre. Leur modification rend la parole méconnaissable, car ce sont eux qui permettent d'identifier les sons du langage.

1.2.3 Quantification du signal audio

- À 3 bits : le son devient rugueux et très bruité, fortement altéré.
- À 8 bits : la voix reste compréhensible mais moins naturelle.

1.2.4 Extraction et séparation de mots

Après repérage visuel, les deux mots ont été extraits via tranches temporelles.

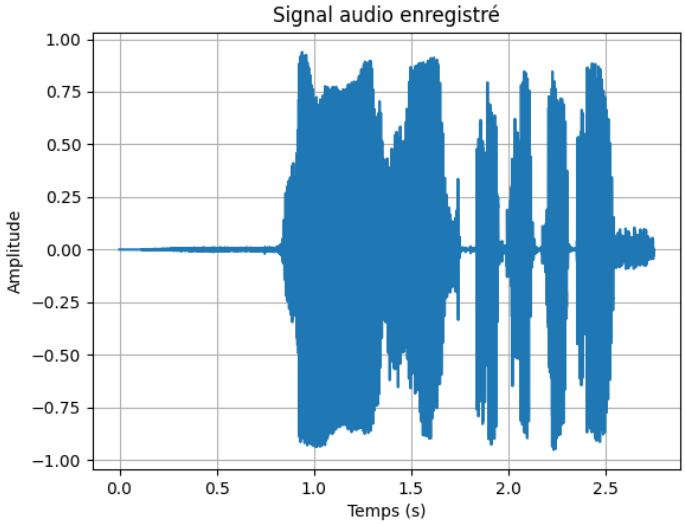
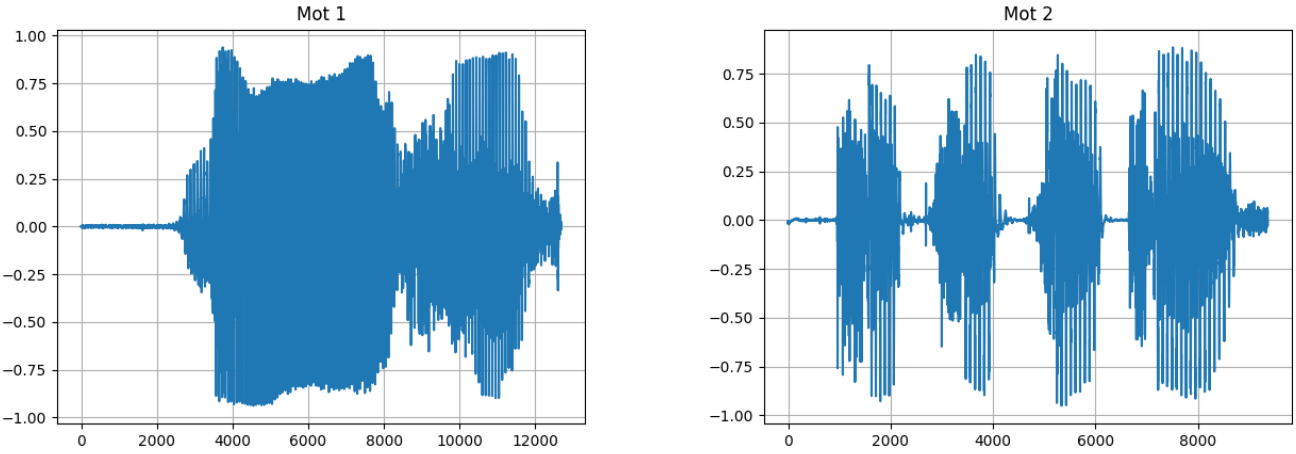


Figure 6: Signal audio enregistré



(a) Mot 1: "Bonjour"

(b) Mot 2: "ChatGpt"

Figure 7: Séparation des mots dans le signal audio

2 Partie 2: Classification des signaux

2.1 Exemple de calcul théorique

Soit un signal sinusoïdal discret défini par :

$$x[n] = A\sin(2\pi f n T_e)$$

L'autocorrélation théorique du signal discret est définie par :

$$R_{xx}[k] = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} x[n] \cdot x[n+k]$$

En remplaçant l'expression de $x[n]$:

$$R_{xx}[k] = \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} A\sin(2\pi f n T_e) \cdot A\sin(2\pi f (n+k) T_e)$$

$$= A^2 \cdot \lim_{N\to\infty} \frac{1}{N} \sum_{n=0}^{N-1} \sin(2\pi f n T_e) \cdot \sin(2\pi f n T_e + 2\pi f k T_e)$$

En utilisant l'identité trigonométrique :

$$\sin(a)\sin(b) = \frac{1}{2}[\cos(a-b) - \cos(a+b)]$$

On a :

$$\sin(2\pi f n T_e) \cdot \sin(2\pi f n T_e + 2\pi f k T_e) = \frac{1}{2}\left[\cos(2\pi f k T_e) - \cos(4\pi f n T_e + 2\pi f k T_e)\right]$$

$$\Rightarrow R_{xx}[k] = \frac{A^2}{2}\cos(2\pi f k T_e)$$

car la somme de $\cos(4\pi f n T_e + \cdot)$ sur une grande fenêtre N tend vers 0.
Conclusion : l'autocorrélation théorique du signal sinusoïdal discret est donnée par :

$$\boxed{R_{xx}[k] = \frac{A^2}{2}\cos(2\pi f k T_e)}$$

où :

- A est l'amplitude du signal,
- f est la fréquence du signal en Hz,
- $T_e = \frac{1}{f_e}$ est la période d'échantillonnage,
- $k \in \mathbb{Z}$ est le décalage (lag) discret.

2.2 Programmation

On calcule l'autocorrélation pour un signal sinusoïdal échantillonné en utilisant la formule :

$$R_{xx}[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] \cdot x[n+k]$$

```
def autocorrelation_manual(x):
N = len(x)
r = np.zeros(2*N - 1)
lags = np.arange(-N + 1, N)
for k in range(-N + 1, N):
    somme = 0
    for n in range(N - abs(k)):
        somme += x[n] * x[n + k] if k >= 0 else x[n - k] * x[n]
    r[k + N - 1] = somme
return lags, r
```

On compare avec l'autocorrelation calculée avec numpy et on trace la différence entre les deux résultats pour visualiser les écarts. Pour convertir les abcisses en secondes, on utilise la formule :

$$\mathrm{t} = \mathrm{k}.T_e = \frac{\mathrm{k}}{f_e}$$

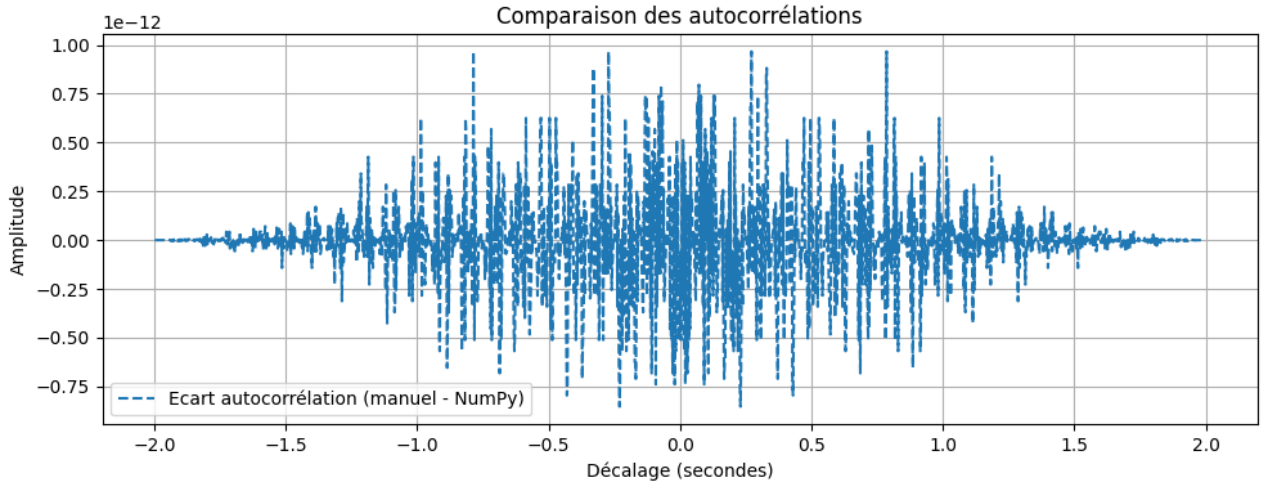


Figure 8: Ecart entre l'autocorrélation manuelle et celle de numpy

L'ecart est trop petit entre les deux méthodes (de l'ordre de 10^{-12}). Ceci est dû à la précision numérique des calculs et au fait que l'autocorrelation numpy est calculée différemment en utilisant plusieurs techniques d'optimisation.

On calcule le rapport des énergies entre la différence entre les deux méthodes et l'autocorrelation de numpy:

$$\text{rapport} = \frac{\sum_{k=0}^{N-1} \text{diff}[k]^2}{\sum_{k=0}^{N-1} \text{autocorr}[k]^2}$$

On trouve plusieurs valeurs de rapport pour différentes valeurs de N. Ceci est dû à la variation de l'énergie du signal en fonction de N, car l'autocorrelation et l'énergie sont sensibles à la longueur du signal (somme de 0 à N-1). On rajoute donc des valeurs au numérateur et au dénominateur dans le calcul de ce rapport (qui ne sont pas équivalentes vu l'écart entre les deux méthodes) ce qui expliques des valeurs de rapport différentes.

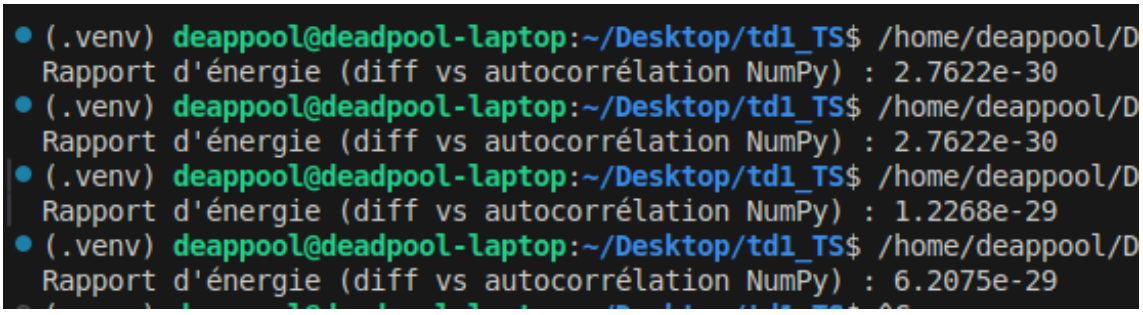


Figure 9: Valeurs du rapport pour plusieurs valeurs de N

2.3 Application à la classification de quelques signaux simples

On génére les signaux demandés avec le code suivant :

```python
# Parametres communs
fe = 11000           # frequence d'echantillonnage
duration = 1      # duree en secondes
t = np.linspace(0, duration, int(fe * duration), endpoint=False)

# 1. Signal sinusoidal de 200 Hz
f = 200
sinus = np.sin(2 * np.pi * f * t)

# 2. Signal triangulaire centre en 0 (serie de Fourier avec 10 termes)
tri = np.zeros_like(t)
for k in range(1, 11):
    n = 2 * k - 1  # uniquement les harmoniques impaires
    tri += ((-1)**((k+1)) / n**2) * np.sin(2 * np.pi * n * f * t)
tri *= (8 / (np.pi**2))  # normalisation serie de Fourier

# 3. Bruit blanc gaussien
taille = fe  # 1 seconde
bruit = np.random.randn(taille)
bruit /= np.max(np.abs(bruit))  # normalisation pour eviter la saturation
```

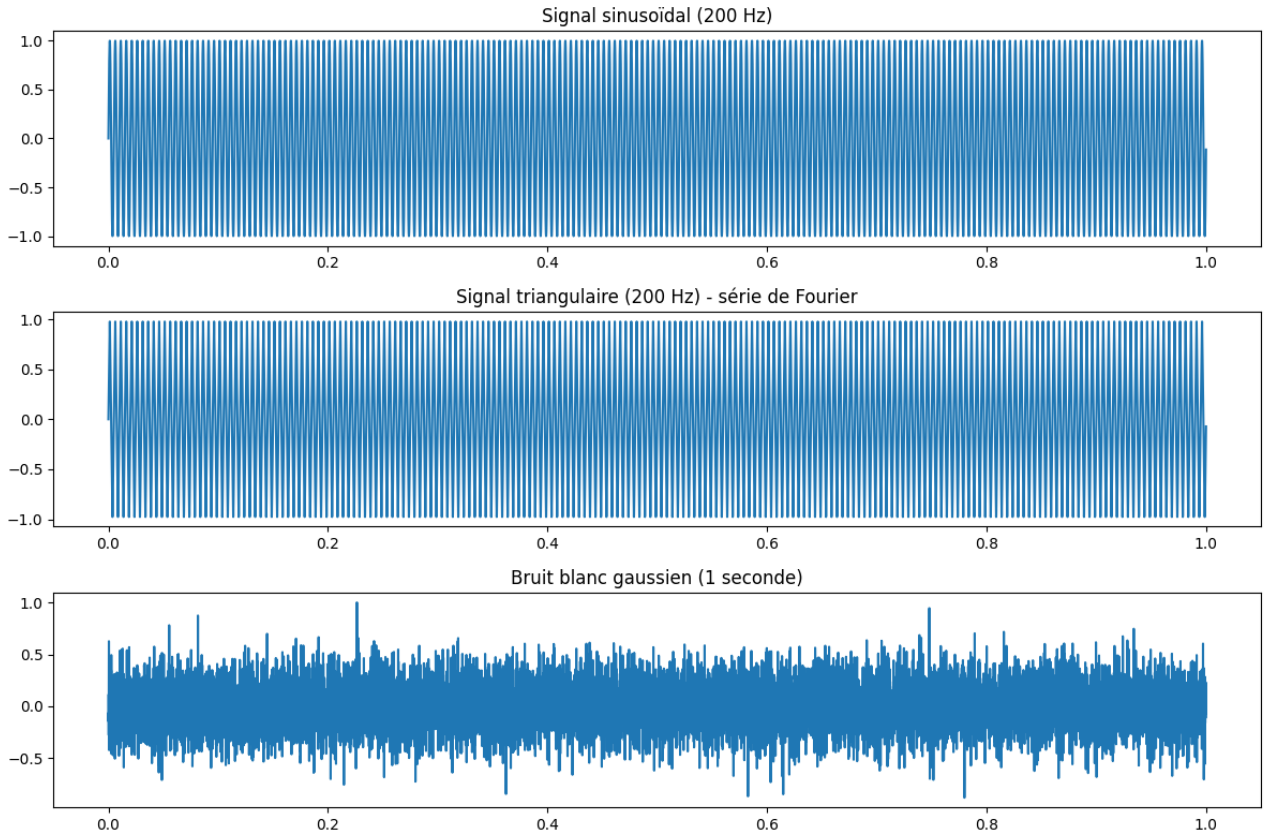


Figure 10: Signaux générés

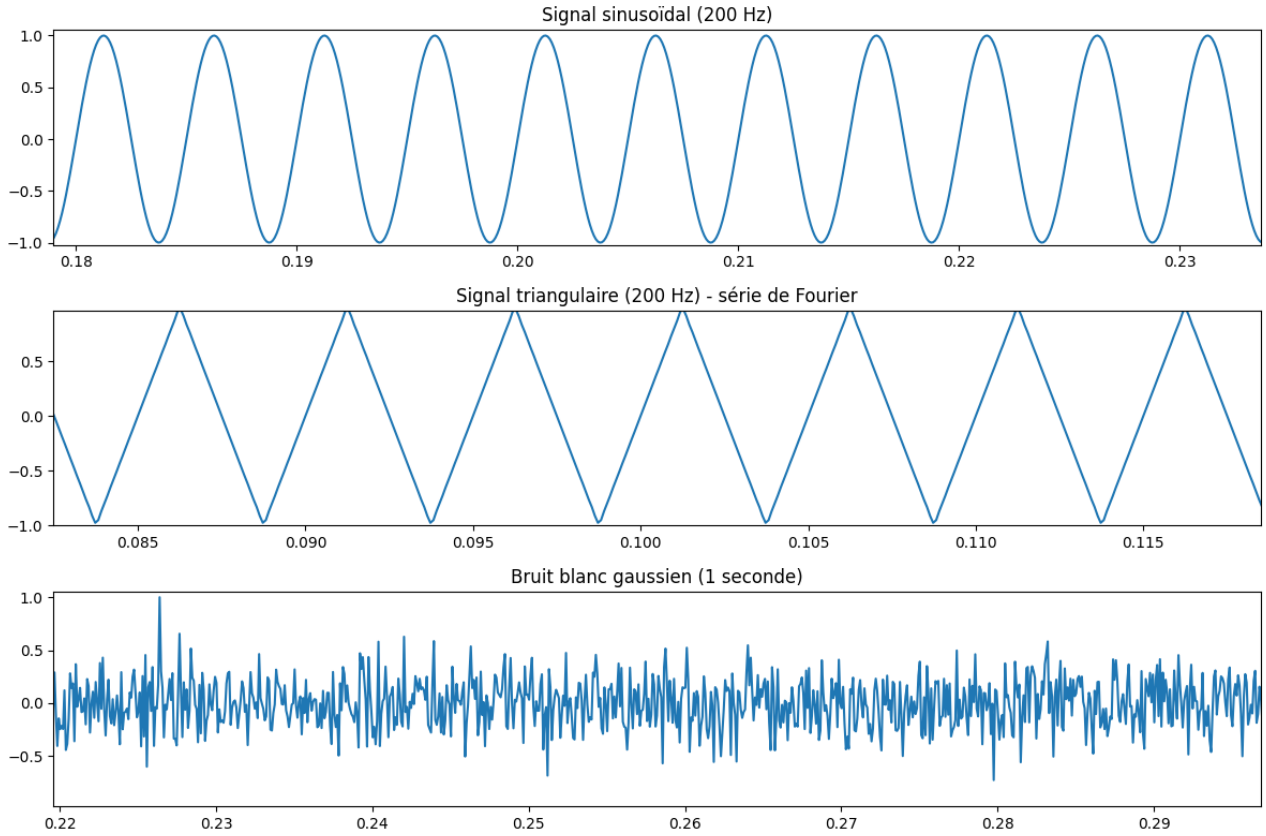


Figure 11: Signaux générés (zoom)

On calcule l'autocorrélation pour chaque signal sur une partie de durée de 30ms. C'est une durée typique en traitement du signal pour analyser la parole (équilibre entre résolution temporelle et fréquence). Elle est suffisante pour voir une structure (formants, périodicité...) sans mélanger des parties trop différentes du signal. On trace également l'autocorrélation du signal enregistré sur deux tranches de 30ms (et 80ms) à des moments différents (1000ms et 1500ms). Cela nous permettera de de comparer l'autocorrélation à différents moments afin de mettre en oeuvre la non-stationnarité du signal.

Un signal est dit **stationnaire** si ses propriétés statistiques, telles que la moyenne, la variance et l'autocorrélation, ne varient pas dans le temps.

L'analyse des autocorrélations montre que :

- Le signal vocal « aa » est **non stationnaire** car ses caractéristiques changent entre différentes tranches temporelles (voix humaine évolue dans le temps).
- Le signal de bruit blanc gaussien est **stationnaire**, ses propriétés statistiques étant constantes dans le temps (Pic au centre, décroissance rapide).
- Les signaux sinusoïdal et triangulaire, synthétisés sur quelques périodes, sont également **stationnaires** car ils sont périodiques et leurs statistiques ne varient pas dans le temps.

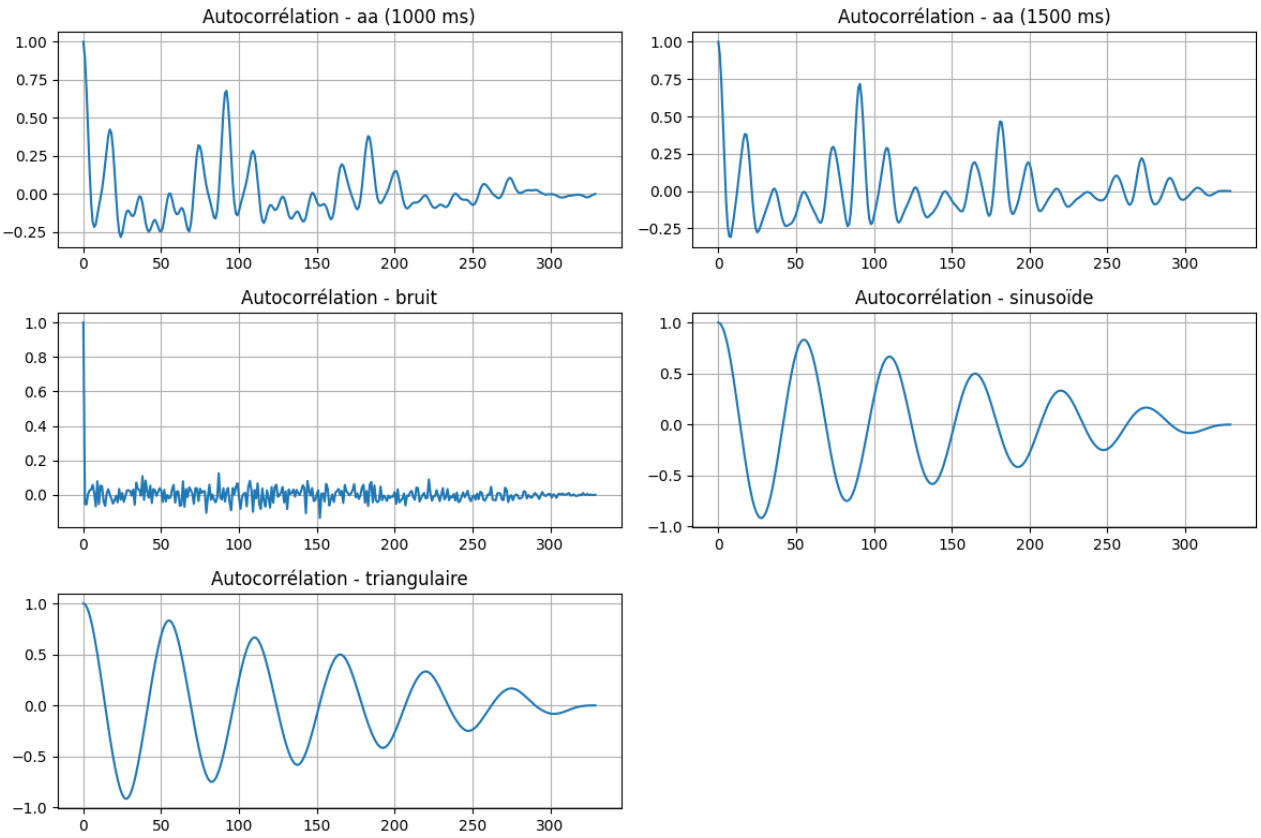


Figure 12: autocorrélation des signaux sur 30ms

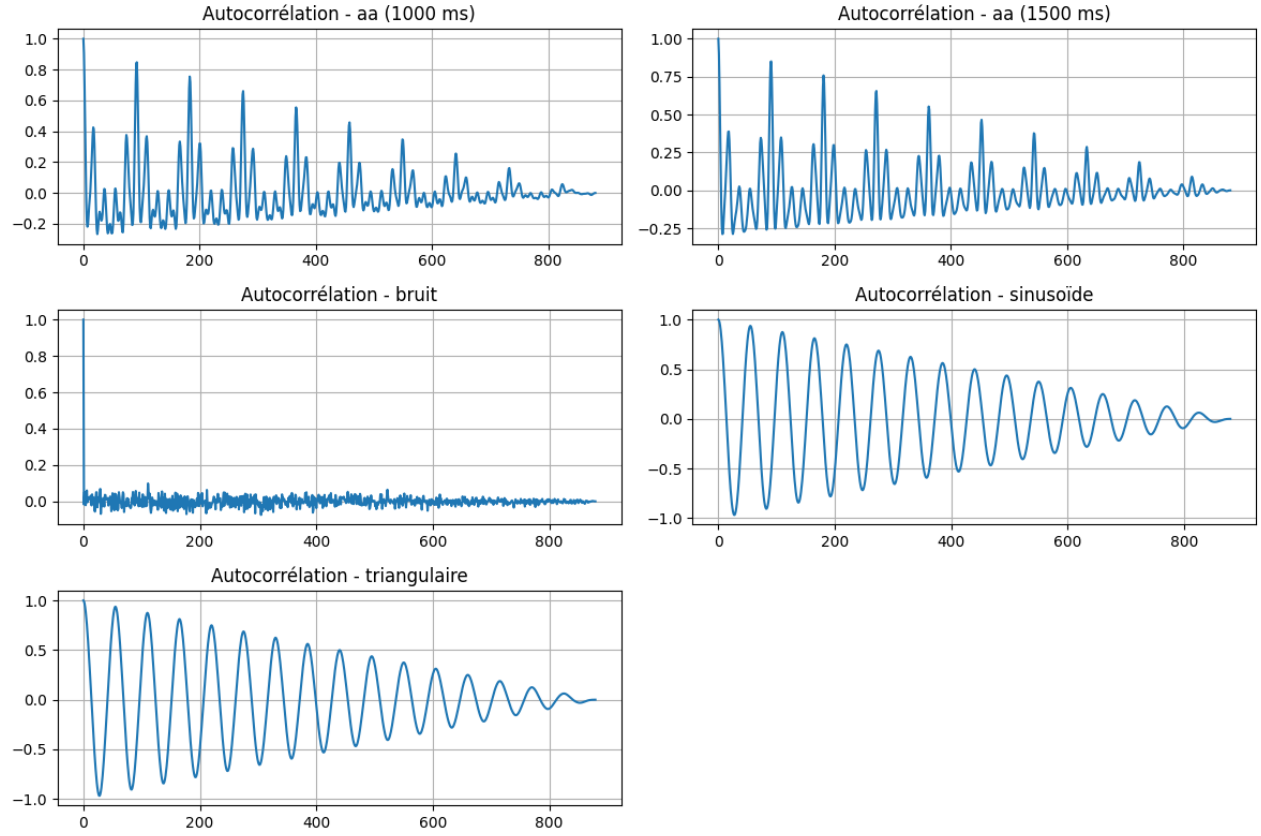


Figure 13: autocorrélation des signaux sur 80ms

2.4 Classification de signaux de parole voisés ou non voisés

Le signal est découpé en tranches de 30 ms pour analyser leur autocorrélation. Les résultats montrent clairement deux types de comportements :

- **Parties non voisées** (« chhhhh ») : l'autocorrélation est bruitée, sans périodicité marquée, caractéristique des sons fricatifs.
- **Parties voisées** (« aaaa ») : l'autocorrélation présente des pics réguliers, révélant une structure périodique due à la vibration des cordes vocales.

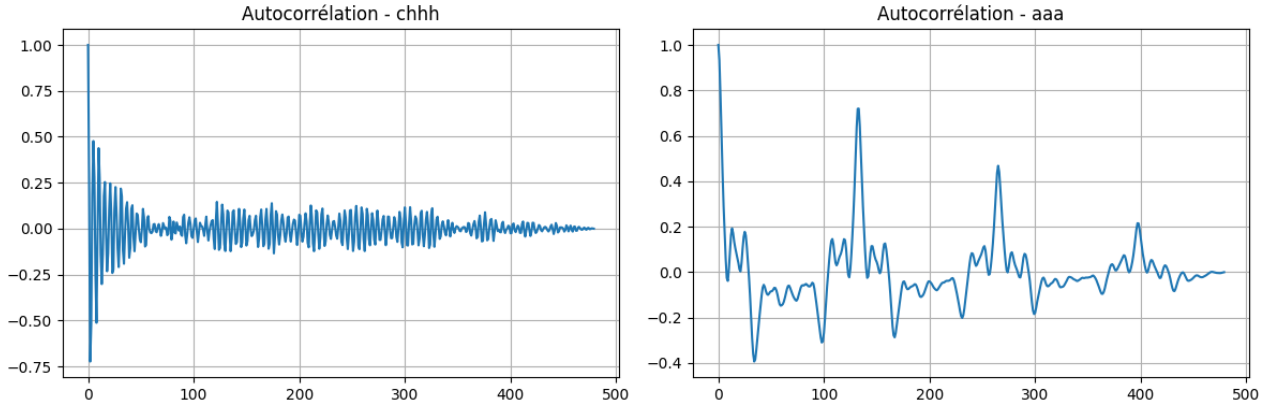


Figure 14: Autocorrélation de différentes tranches du mot « chat » : (gauche) non voisée, (milieu) transition, (droite) voisée.

L'autocorrélation permets d'estimer la fréquence fondamentale f_0 d'un signal voisé. On s'attend à ce qu'un signal périodique présente un maximum secondaire au retard correspondant à sa période fondamentale $T_0 = \frac{1}{f_0}$.

Pour identifier les pics secondaires dans l'autocorrélation, nous utilisons la fonction `find_peaks` de `scipy.signal`. Afin d'éviter de détecter des pics non significatifs dus au bruit ou à des résonances (formants), le paramètre **height** qui correspond à la hauteur minimale doit être soigneusement choisis pour ignorer les pics trop faibles dus au bruit. Nous avons fixé ce seuil à 0,3 après normalisation.

Pour la tranche correspondant à la voyelle *« a »* dans l'enregistrement (partie voisée), nous obtenons après traitement:

- Indices du premier pic détecté : 133
- On en déduit une fréquence fondamentale :

$$f_0 = \frac{f_e}{\text{lag}} = \frac{16\,000}{133} \approx 120{,}3 \text{ Hz}$$

Ce résultat est conforme aux plages typiques pour une voix masculine (85–180 Hz). Il est important de noter que la fréquence fondamentale ne caractérise pas la voyelle elle-même, mais le locuteur.

3 Partie 3: Aspects fréquentiels

3.1 Echantillonnage

Le théorème de Nyquist-Shannon stipule qu'un signal bande limitée, dont la fréquence maximale est f_0, peut être parfaitement reconstruit à partir de ses échantillons si la fréquence d'échantillonnage vérifie :

$$f_e > 2f_0$$

La fréquence $f_e = 2f_0$ est appelée **fréquence de Nyquist** et constitue la limite minimale pour éviter le phénomène de repliement spectral (*aliasing*).

On échantillone donc le signal selon trois cas distincts :

- **Cas 1 :** $f_e = 2f_0$ (fréquence de Nyquist),
- **Cas 2 :** $f_e = 5f_0$ (fréquence supérieure),
- **Cas 3 :** $f_e = 0{,}8f_0$ (fréquence inférieure).

On vois que pour les deux premiers cas, le signal échantillonné est fidèle à la forme du signal d'origine. En revanche, pour le troisième cas, l'échantillonnage à une fréquence inférieure à la fréquence de Nyquist entraîne un repliement spectral, rendant le signal méconnaissable.

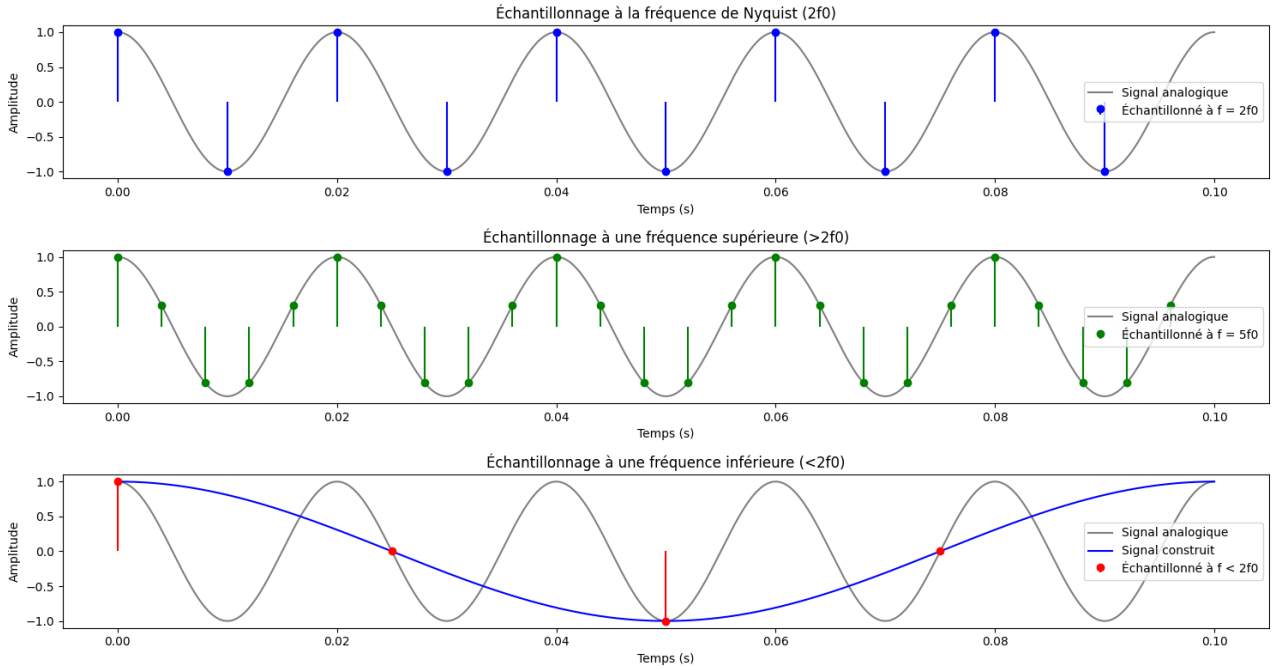


Figure 15: Signaux échantillonnés

3.2 La Transformée de Fourier discrète (TFD)

Nous avons implémenté une fonction en `Python` permettant de calculer et tracer la densité spectrale d'énergie d'un signal extrait d'un fichier `.wav`. La fonction lit le fichier audio, extrait une tranche du signal à partir d'un indice donné, applique une fenêtre de Hanning pour atténuer les effets de bord, puis calcule la transformée de Fourier rapide (`FFT`) sur 2^n points. La densité spectrale est obtenue en prenant le module au carré des coefficients de la FFT (représentant l'énergie en fréquence), normalisée et convertie en décibels (dB) selon la formule :

$$P_{\text{dB}}(f) = 10 \log_{10} \left(\frac{|\text{FFT}(f)|^2}{\max(|\text{FFT}(f)|^2)} + \varepsilon \right),$$

avec ε un terme petit (10^{-12}) pour éviter les problèmes numériques liés au logarithme de zéro.

La courbe obtenue est symétrique par rapport à l'axe des fréquences, car la FFT calcule les coefficients complexes pour les fréquences positives et négatives. On ne garde que la moitié positive du spectre.

Le choix de 2^n (taille de la FFT) influence la **résolution fréquentielle**. Plus n est grand, plus les fréquences sont discrétisées finement, ce qui permet de mieux isoler les pics correspondant aux composantes harmoniques du signal.

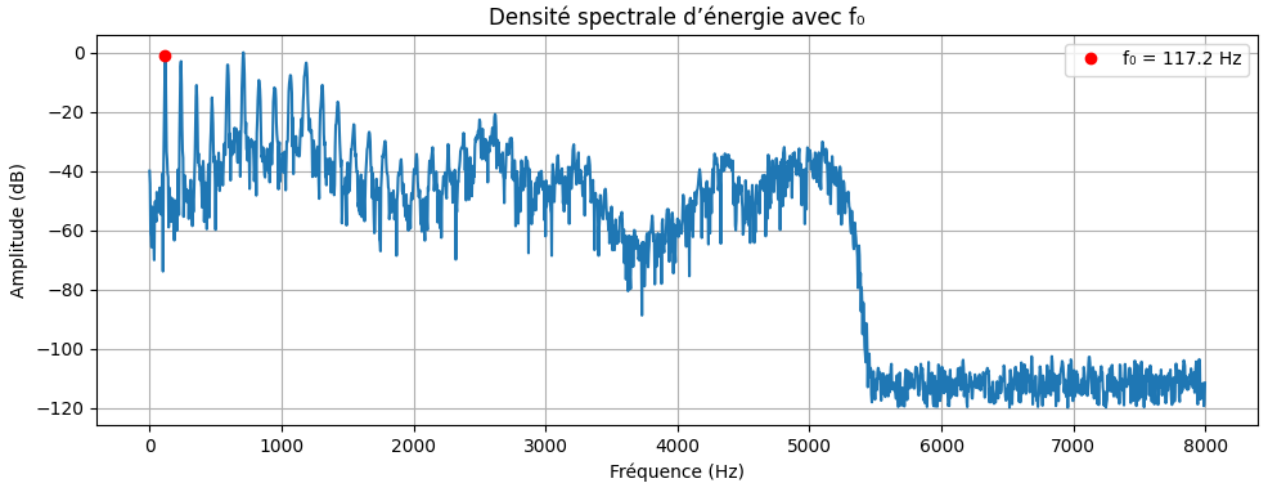


Figure 16: FFT

Sur le spectre obtenu, on observe une structure en pics caractéristiques d'un signal périodique voisé (ici le phonème *aaa*). La fréquence fondamentale correspond au premier pic significatif au-dessus du bruit. À partir du spectre, nous détectons une fréquence fondamentale de **117,2 Hz**.

Ce résultat est très proche de celui obtenu par l'autocorrélation dans le domaine temporel, où nous avions trouvé une fréquence de **120,3 Hz** (3.1 Hz de différence). Cette légère différence peut s'expliquer par :

- la résolution fréquentielle limitée de la FFT (dépendante de la taille de la tranche),
- les effets de la fenêtre de Hanning qui modifie légèrement le contenu fréquentiel,
- et les arrondis ou approximations dans la localisation du pic.

Malgré cela, les deux méthodes convergent vers une estimation très cohérente de la fréquence fondamentale, typique d'une voix d'homme (comprise entre 85 et 180 Hz).

Le *zero-padding* consiste à compléter un signal par des zéros à la fin. Cette opération n'ajoute aucune nouvelle information, mais elle permet d'augmenter artificiellement la longueur du signal, ce qui améliore la **résolution fréquentielle** du spectre après transformation de Fourier.

Nous avons implémenté une fonction `zero_padding` en `Python`, prenant en entrée le signal initial et le nombre de zéros à ajouter, et renvoyant le signal étendu.

```
def zero_padding(signal, nb_zeros):
return np.hstack((signal, np.zeros(nb_zeros)))
```

Nous avons ensuite comparé les densités spectrales d'énergie (obtenues par FFT) avant et après zero-padding. Le résultat montre que :

- les **positions des pics** dans le spectre restent inchangées (le contenu fréquentiel est le même),
- mais les courbes sont **plus lisses** comme on peut le voir dans la figure 18 et les pics **mieux localisés** après padding, ce qui facilite l'estimation des fréquences fondamentales,
- le zero-padding agit donc comme une **interpolation en fréquence**.

Cela est particulièrement utile pour distinguer des pics proches ou évaluer plus précisément une fréquence fondamentale.

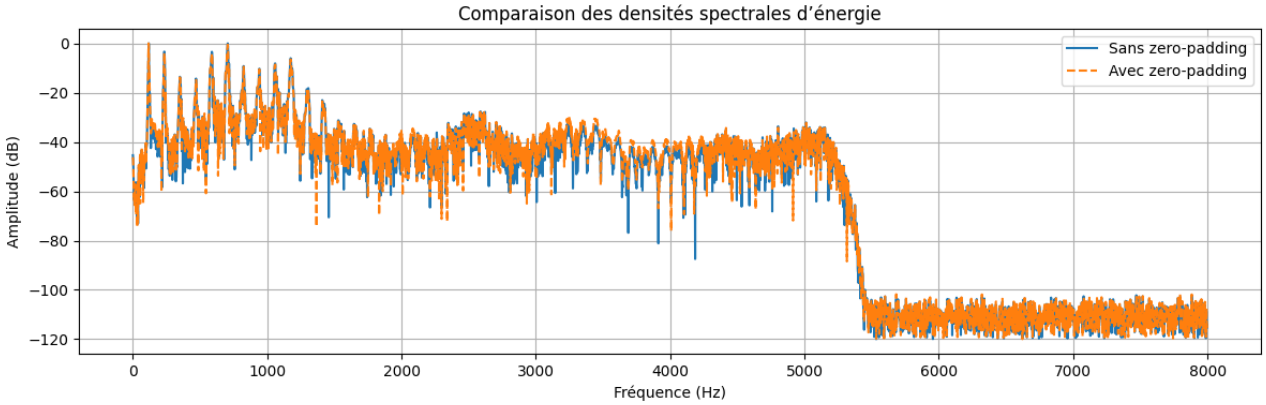


Figure 17: FFT avec et sans zero-padding

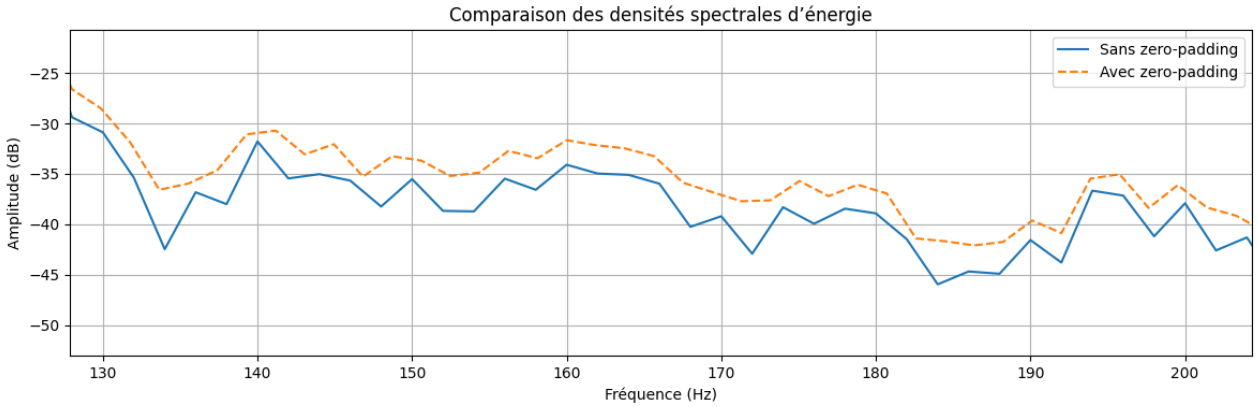


Figure 18: FFT avec et sans zero-padding (zoom)

On peut faire directement du zero-padding dans la fonction **np.fft.fft(x, n)**.

- Si n est supérieur à la longueur de x, la fonction ajoute des zéros à la fin de x jusqu'à atteindre la longueur n.
- Si n est inférieur ou égal à la longueur de x, la fonction tronque x à la longueur n.

3.3 Changement de fréquence d'échantillonnage

Il est essentiel de bien distinguer deux opérations proches mais conceptuellement différentes :

- **Changer la cadence (changer la vitesse de lecture)** consiste à lire un signal audio plus vite ou plus lentement, sans modifier son contenu en échantillons. Cela revient à modifier la *fréquence d'échantillonnage effective* sans changer les données elles-mêmes. Par exemple :
  - Lire un signal échantillonné à 44,100 Hz comme s'il était à 22,050 Hz revient à le lire deux fois plus lentement (et donc plus grave).
  - Inversement, le lire à 88,200 Hz le rend deux fois plus rapide (et plus aigu).
- **Le rééchantillonnage** (*resampling*) modifie le *nombre total d'échantillons* d'un signal, généralement en interpolant ou décimant les données, pour obtenir un nouveau signal correspondant à une nouvelle fréquence d'échantillonnage, mais sans changer la vitesse de lecture. Cela permet de :

Et donc Pour :

- **Réduction de cadence (ralentissement)** : on insère des échantillons ou on répète les échantillons existants. Cela allonge la durée du signal et rend la voix plus grave.
- **Augmentation de cadence (accélération)** : on sous-échantillonne le signal, par exemple en prenant un échantillon sur deux, ce qui réduit la durée et rend la voix plus aiguë.

Dans les deux cas, si l'on garde la même fréquence d'échantillonnage f_e, alors la durée du signal est directement modifiée.

Le facteur n de changement de cadence détermine le degré de ralentissement ou d'accélération :

- Si $n > 1$, on applique un ralentissement. Le signal devient plus lent, plus long et plus grave.
- Si $n < 1$ (ou bien $n = 1/k$ avec $k > 1$), on accélère le signal. Il devient plus court et plus aigu.

Par exemple :

- Pour $n = 2$, chaque échantillon est répété deux fois $\Rightarrow$ la durée est doublée.
- Pour $n = 1/3$ ($k = 3$), on garde un échantillon sur trois $\Rightarrow$ la durée est divisée par 3.

On utilise le code suivant :

```
    def reduce_cadence(filename, tranche_size, start_index, factor, out_filename):
    """
    Accelere le signal d'un facteur <factor> :
    - On prend 1 echantillon sur factor (decimation).
    - On conserve fs identique (donc la duree est divisee par factor).
    """
    # Lecture du wav
    fs, signal = wavfile.read(filename)
    # On extrait la tranche demandee
    tranche = signal[start_index:start_index + tranche_size]
    # On decime : on prend un echantillon sur 'factor'
    reduced = tranche[::factor]
    # On garde la meme frequence d'echantillonnage
    new_fs = fs
    # On ecrit le resultat
    wavfile.write(out_filename, new_fs, reduced.astype(np.int16))
    return reduced, new_fs


def increase_cadence(filename, tranche_size, start_index, factor, out_filename):
    """
    Ralentit le signal d'un facteur `factor` :
    - On repete chaque echantillon `factor` fois (np.repeat).
    - On conserve fs identique (donc la duree est multipliee par factor).
    """
    # Lecture du wav
    fs, signal = wavfile.read(filename)
    # On extrait la tranche demandee
    tranche = signal[start_index:start_index + tranche_size]
    # On repete chaque echantillon factor fois
    upsampled = np.repeat(tranche, factor)
    # On garde la meme frequence d'echantillonnage
    new_fs = fs
    # On ecrit le resultat
    wavfile.write(out_filename, new_fs, upsampled.astype(np.int16))
    return upsampled, new_fs


def generate_cadence_variations(filename, tranche_size, start_index):
    """
    Genere plusieurs fichiers .wav acceleres (x2, x3, x4) et ralentis (%2, %3, %4)
    dans le dossier 'outputs_cadence/'. Pour chaque facteur f :
      - accelere : fichier "faster_x{f}.wav"
      - ralenti  : fichier "slower_x{f}.wav"
    """
    acceleration_factors = [2, 3, 4, 5, 6]  # 2x, 3x, 4x plus rapides
    slowing_factors      = [2, 3, 4, 5, 6]  # %2, %3, %4 plus lents

    for f in acceleration_factors:
        out_file = f"outputs_cadence/faster_x{f}.wav"
```

```
        reduce_cadence(filename, tranche_size, start_index, f, out_file)
        print(f"Fichier genere : {out_file} (accelere x{f})")

    for f in slowing_factors:
        out_file = f"outputs_cadence/slower_x{f}.wav"
        increase_cadence(filename, tranche_size, start_index, f, out_file)
        print(f"Fichier genere : {out_file} (ralenti %{f})")

generate_cadence_variations(filename= "audio-sig.wav", tranche_size= 25600,
    start_index = 500)
```

Lorsque le facteur de modification est trop important, le signal devient difficilement intelligible :

- **Trop accéléré ($n \gg 1$)** : la durée est trop courte, les mots sont comprimés, et la fréquence fondamentale est augmentée au point que la voix devient méconnaissable.
- **Trop ralenti ($n \gg 1$ en ralentissement)** : la durée devient très longue, la voix semble traînante, et les syllabes sont déformées voire hachées.

Ainsi, au-delà de certains seuils (par exemple $n >= 3$ ou $n <= 0.33$), il devient difficile, voire impossible, de reconnaître les mots prononcés. Cela est dû au fait que les formants (zones de concentration d'énergie dans le spectre) sont déplacés ou trop étalés pour être perçus correctement.

Une augmentation ou réduction naïve de la cadence (comme avec ce code) modifie à la fois le *tempo* et la *hauteur*. Pour conserver uniquement l'un ou l'autre, des techniques plus avancées comme la transformée de Fourier à court terme (STFT) sont utilisées.

En traçant la transformée de Fourier des signaux accélérés et ralentis en comparaison avec le signal original, on observe les comportements suivants (Voir figure 19):

- **Original** : spectre de référence, avec ses pics de formants localisés autour des fréquences caractéristiques de la voix. Ces pics traduisent la présence des voyelles et consonnes, et leur position reflète la structure du signal vocal.
- **Faster x2, Faster x4** : le spectre est étiré vers les hautes fréquences. Autrement dit, les pics de formants sont décalés vers la droite, ce qui correspond à une montée en hauteur (pitch) — la voix devient plus aiguë. Ce phénomène est dû à la réduction de la durée du signal sans interpolation, ce qui contracte les cycles et augmente leur fréquence apparente.
- **Slower x2, Slower x4** : à l'inverse, le spectre est compressé vers les basses fréquences. Les formants se déplacent vers la gauche, traduisant une voix plus grave. Cette compression résulte de l'introduction de zéros (échantillons nuls) qui allongent artificiellement la période des signaux périodiques.
- **Amplitude** : on observe également une modification de l'amplitude du spectre. Lors de l'accélération, la suppression d'échantillons réduit l'énergie totale, ce qui diminue généralement les amplitudes du spectre. En revanche, lors du ralentissement, l'ajout de zéros augmente la longueur du signal traité par la FFT, ce qui peut amplifier certaines composantes fréquentielles, en particulier à basse fréquence, et conduire à une amplitude spectrale plus élevée.

Contrairement aux méthodes naïves d'augmentation ou de réduction de cadence par décimation (suppression d'échantillons) ou zéro-padding (ajout de zéros), la fonction `resample` de SciPy utilise une interpolation efficace basée sur la transformée de Fourier pour reconstruire le signal. Cette approche conserve bien mieux les caractéristiques spectrales du signal, notamment les formants vocaux.

- **À l'écoute**, les signaux transformés avec `resample` sont plus fluides, plus naturels, et conservent mieux l'intelligibilité du contenu vocal, même pour des facteurs élevés.
- En comparaison, la première méthode introduit des distorsions, un effet robotique ou haché, et altère la hauteur et le timbre de manière plus brutale.

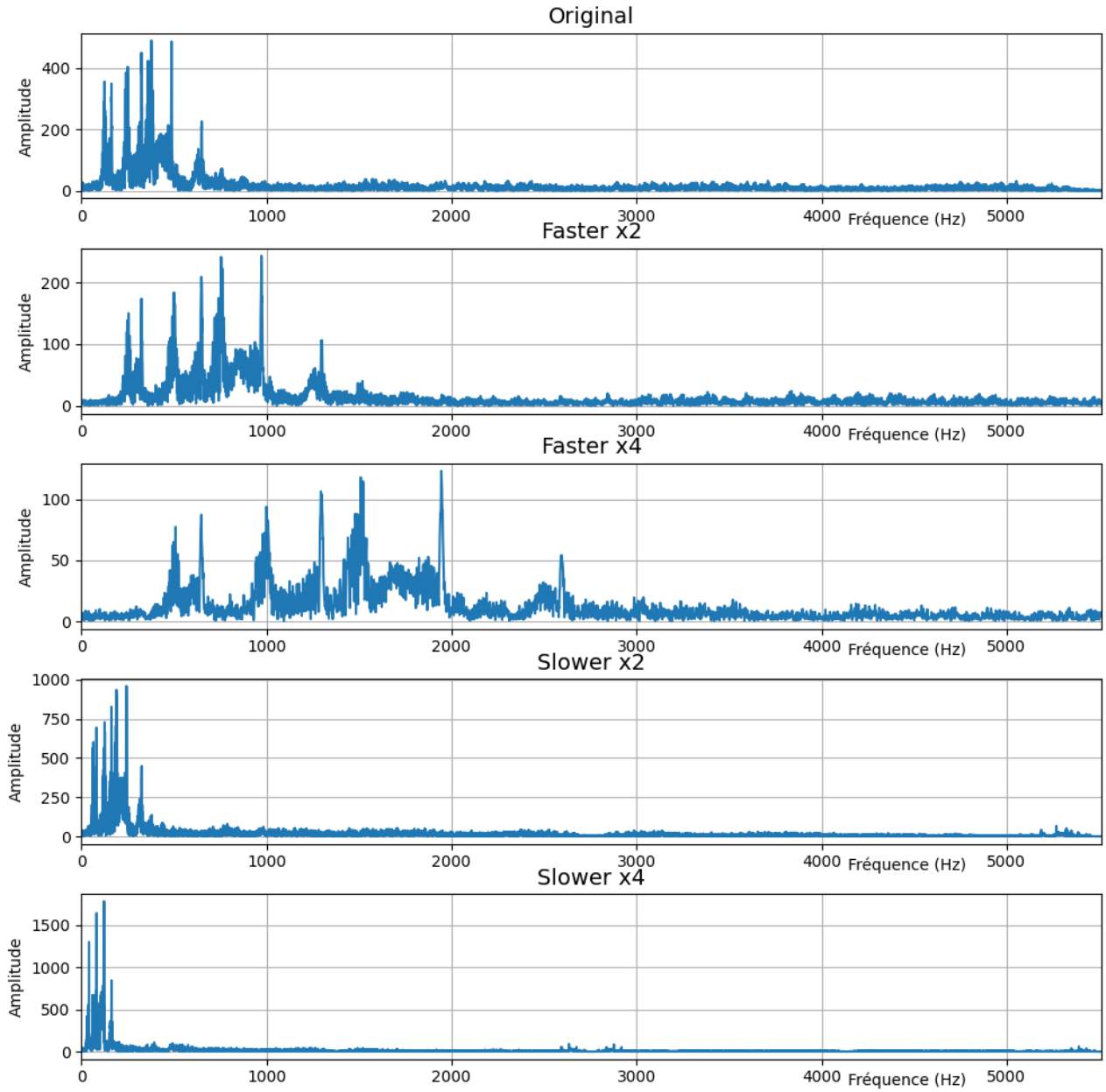


Figure 19: FFT cadence modifiée

3.4 Analyse spectrale

3.4.1 Analyse d'une tranche de signal par TFD

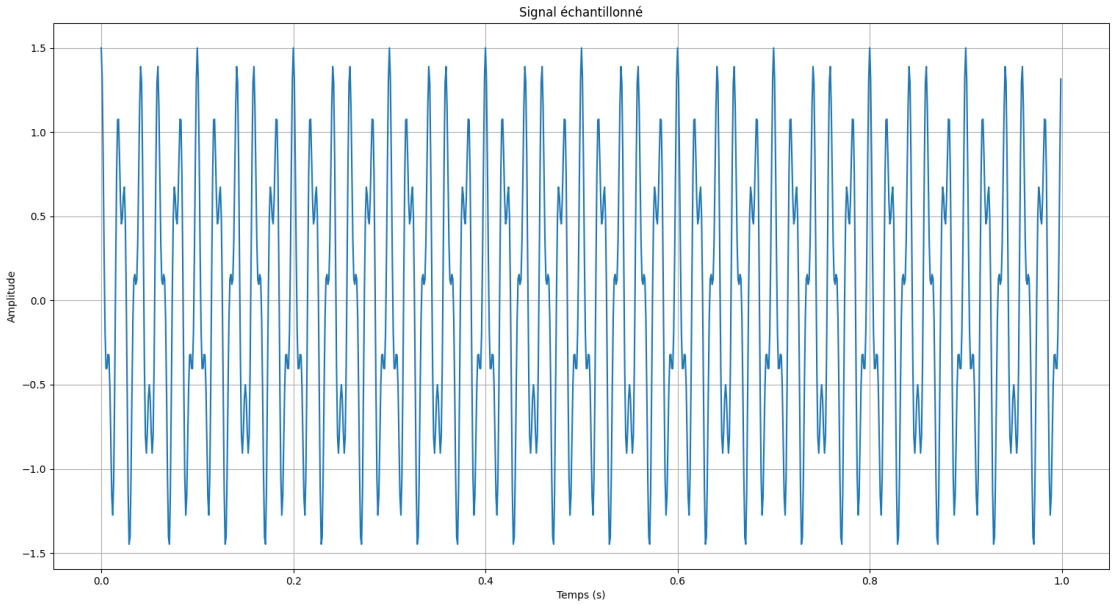


Figure 20: exemple d'un signal généré avec: N = 1000 ; fe = 1000 ; f0 = 50 ; f1 = 120 ; A0 = 1.0 ; A1 = 0.5

La transformée de Fourier discrète d'une fenêtre de Hanning de longueur N présente un module en forme de triangle centré autour de la fréquence zéro. Par analyse expérimentale :

- Le lobe principal est bien approximé par une forme triangulaire.
- La largeur de ce lobe est d'environ $\frac{4}{N}$ en fréquence normalisée (de $-2/N$ à $+2/N$).
- Après normalisation, la surface sous ce lobe principal (approximation de l'intégrale) est proche de 1, ce qui montre que la majorité de l'énergie spectrale est concentrée dans cette bande.

Cela justifie l'utilisation fréquente de la fenêtre de Hanning dans les analyses spectrales pour limiter les effets de repliement spectral tout en gardant une bonne localisation fréquentielle.

Lorsque la longueur N de la fenêtre de Hanning augmente, on observe que le module de sa transformée de Fourier devient plus "fin". Cela s'explique par le compromis temps–fréquence inhérent à la transformée de Fourier : une plus grande durée d'observation (fenêtre plus longue) permet une meilleure résolution fréquentielle. Ainsi :

- Le lobe principal devient plus étroit, avec une largeur approximative de $\frac{4}{N}$.
- Le spectre conserve sa forme triangulaire mais est plus concentré autour de la fréquence zéro.
- L'amplitude du lobe principal augmente légèrement pour compenser la réduction de largeur (la surface reste ≈ 1).

Cela confirme que la fenêtre de Hanning agit comme un filtre passe-bas doux, et que plus elle est longue, plus elle isole finement une bande de fréquences autour de zéro.

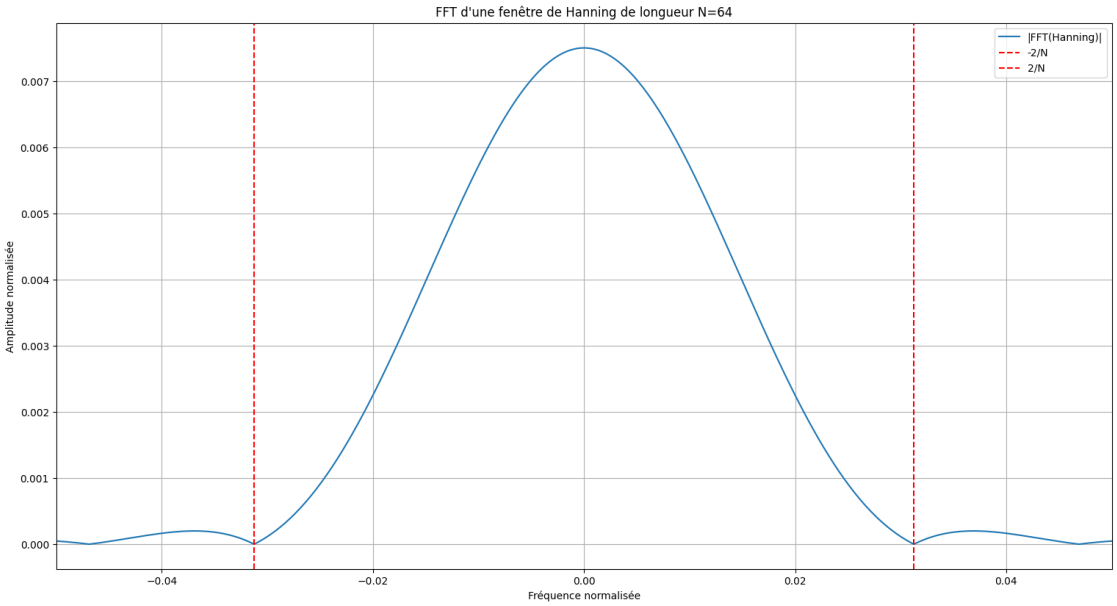


Figure 21: FFT fenêtre de Hanning N = 64

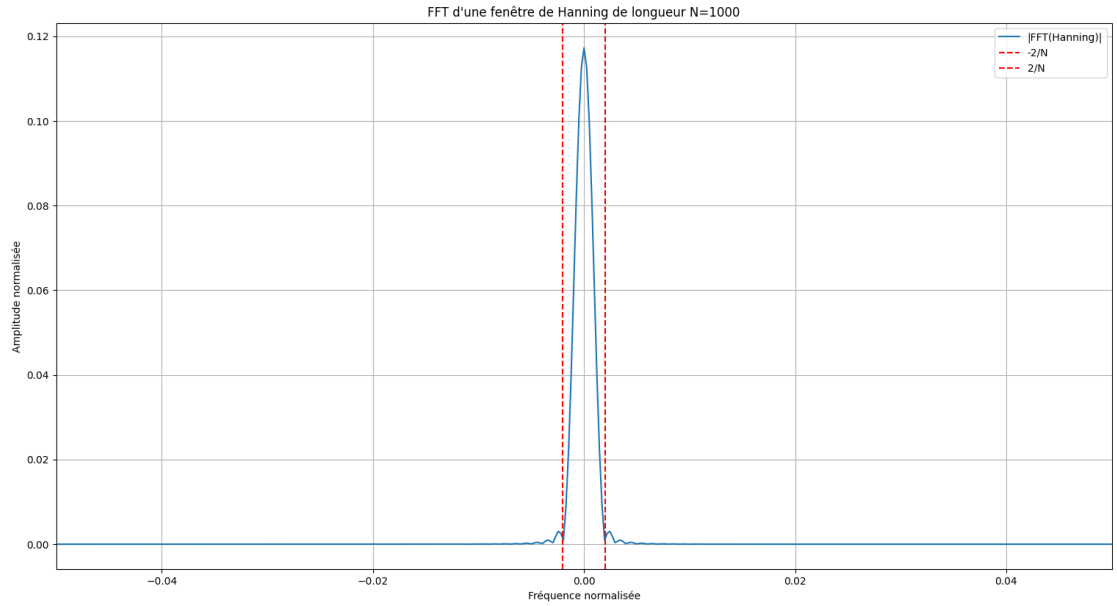


Figure 22: FFT fenêtre de Hanning N = 1000

On considère le signal :

$$x(t) = A\cos(2\pi f_0 t) + A\cos(2\pi f_1 t)$$

avec $f_0 = 40$ kHz, $f_1 = 61$ kHz, $f_e = 512$ kHz et $A = 1$. Le signal est échantillonné et une tranche de $N = 256$ points est extraite pour calculer la TFD après pondération par une fenêtre de Hanning.

La transformée de Fourier discrète d'une sinusoïde après pondération par une fenêtre de Hanning est égale à la convolution de la TFD idéale (un pic) avec la TFD de la fenêtre (un triangle de surface unité et de largeur $\frac{4}{N}$).

Chaque sinusoïde donne donc lieu à un lobe triangulaire :

- centré sur f_0 et f_1 ;

- de largeur en fréquence réelle :

$$\Delta f = \frac{4f_e}{N} = \frac{4 \cdot 512000}{256} = 8000 \text{ Hz}$$

Le module de la TFD est donc la somme de deux triangles :

- chacun de largeur 8 kHz (en fréquence) ;

- et de surface égale à 1 (normalisation par la fenêtre).

Le spectre montre deux pics nets élargis autour de 40 kHz et 61 kHz, ce qui permet d'estimer les fréquences dominantes du signal malgré la résolution limitée par la fenêtre.

La fenêtre de Hanning réduit le leakage spectral (fuites autour des pics), rendant les pics plus nets par rapport à une TFD sans fenêtre.

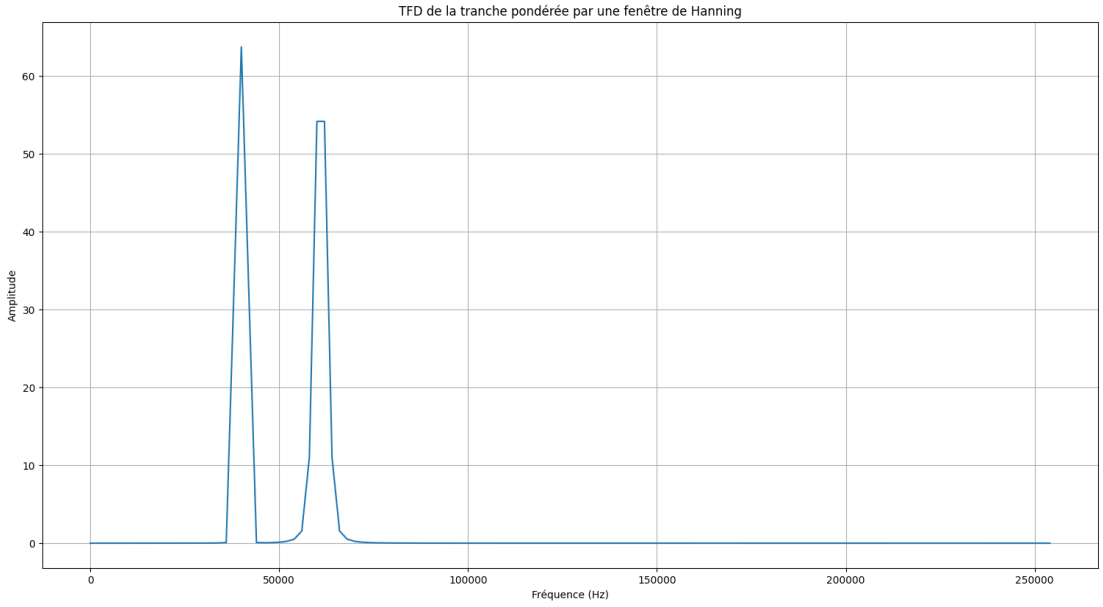


Figure 23: TFD avec fenêtre de Hanning

3.4.2 Effets de quelques fenêtres

Dans les trois cas on détecte bien les deux fréquences dominantes du signal (995 Hz et 1200 Hz), mais avec des différences notable dans l'amplitude en raison de la différence entre A_0 et A_1. On remarque également une légère différence sur la forme du pic autour de 995 Hz entre les types des fenêtres appliquées.

L'utilisation de la fenêtre de Hanning réduit efficacement les lobes secondaires par rapport à la fenêtre rectangulaire, tout en conservant une bonne résolution en fréquence, ce qui permet de distinguer les deux composantes spectrales. La fenêtre de Hamming offre une réduction encore plus importante des lobes secondaires, mais au prix d'un élargissement du lobe principal, ce qui peut rendre la séparation des deux fréquences plus difficile si elles sont proches. Quant à la fenêtre de Blackman, elle présente les plus faibles lobes secondaires, mais au détriment d'une résolution encore plus réduite. Ainsi, le choix de la fenêtre représente un compromis entre la résolution fréquentielle (finesse de séparation des pics) et la réduction des effets de fuite spectrale (lobes secondaires), particulièrement important ici pour détecter correctement la faible composante à 1200 Hz en présence d'une composante dominante à 995 Hz.

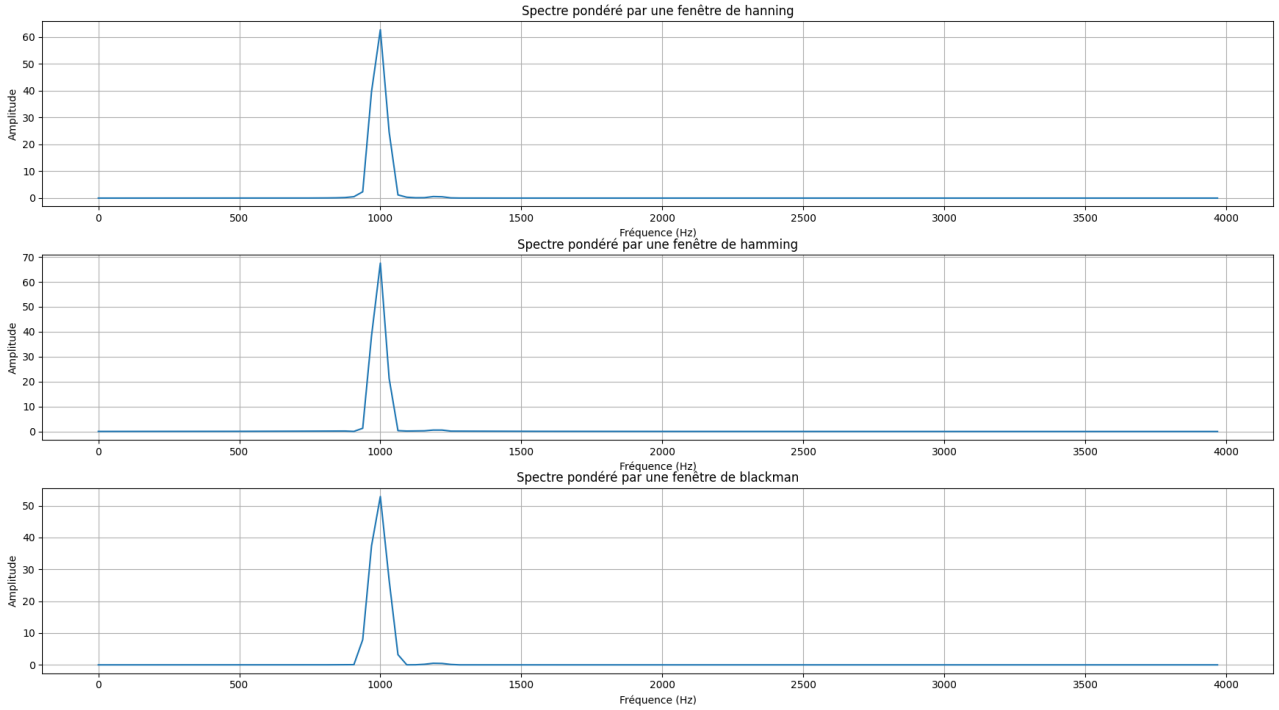


Figure 24: TFD avec différentes fenêtres

Dans ce cas, on analyse un signal de courte durée, composé de 256 échantillons, contenant deux fréquences proches : $f_0 = 995$ Hz et $f_1 = 1200$ Hz. La résolution fréquentielle théorique, sans zero-padding, est donnée par :

$$\Delta f = \frac{f_e}{N} = \frac{8000}{256} = 31{,}25 \text{ Hz}$$

Cela signifie que les fréquences ne peuvent être détectées qu'à des positions multiples de 31,25 Hz (par exemple : 968,75 Hz, 1000 Hz, 1031,25 Hz, etc.), ce qui est insuffisant pour distinguer précisément les composantes situées à 995 Hz et 1200 Hz.

L'ajout de *zero-padding*, par exemple en prolongeant le signal à 1024 points, n'apporte aucune information supplémentaire sur le contenu fréquentiel réel du signal, mais permet de raffiner la grille fréquentielle. La nouvelle résolution de l'axe fréquentiel devient :

$$\Delta f_{\text{interp}} = \frac{f_e}{1024} = \frac{8000}{1024} = 7{,}8125 \text{ Hz}$$

Cette interpolation plus fine permet d'observer les pics spectraux de manière plus précise (autour de 995 Hz et 1200 Hz), même si la capacité à les distinguer — c'est-à-dire la résolution effective — reste inchangée. En résumé, le zero-padding améliore la lecture du spectre (interpolation), mais n'améliore pas la séparation entre fréquences proches (résolution).

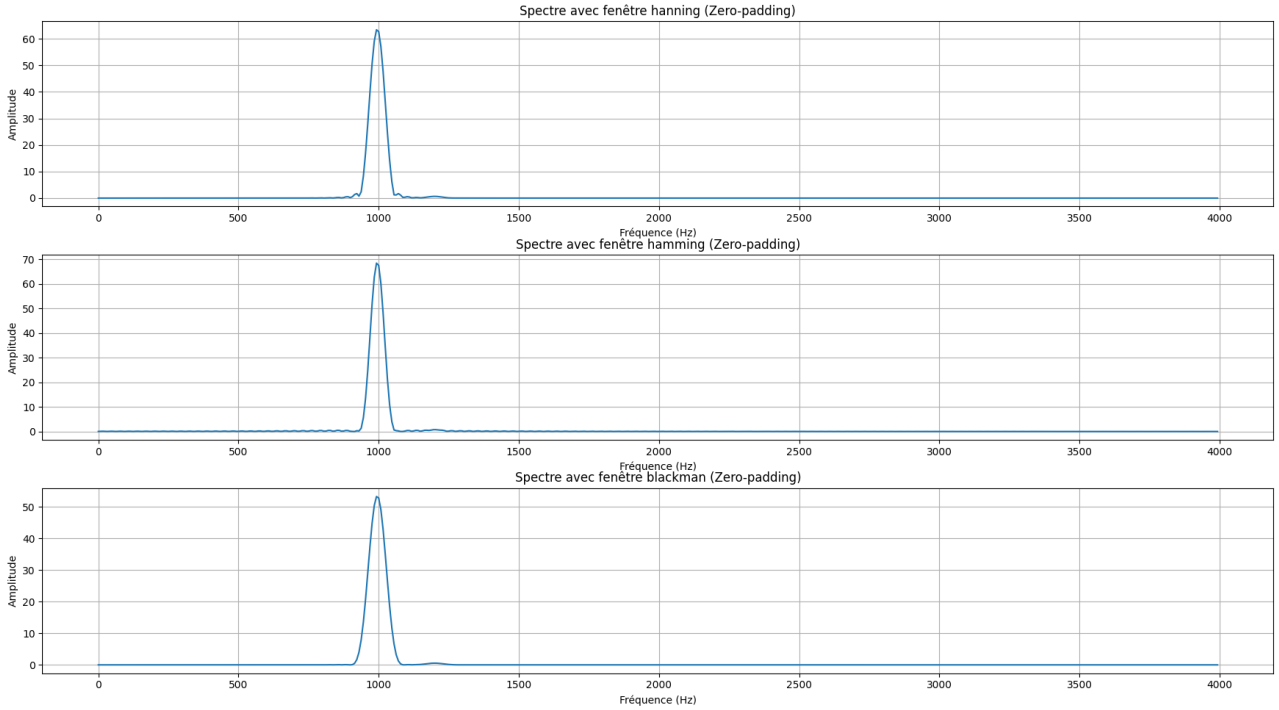


Figure 25: TFD avec différentes fenêtres + zero-padding

4 Partie 4: Application : détection de pitch

4.1 Code python

On a fait le code suivant :

```
    import numpy as np
import scipy.io.wavfile as wav
import scipy.signal as signal
import os


def is_voiced(frame, energy_threshold):
    """
    Determine si une trame est voisee selon son energie.
    """
    energy = np.sum(frame ** 2) / len(frame)
    return energy > energy_threshold

def autocorrelation_pitch(frame, fs, fmin, fmax):
    """
    Estime la frequence fondamentale d'une trame via l'autocorrelation.
    Retourne None si la recherche echoue.
    """
    # Soustraire la moyenne pour centrer
    frame = frame - np.mean(frame)
    # Calculer l'autocorrelation
    corr = np.correlate(frame, frame, mode='full')
    corr = corr[len(corr)//2:]  # ne garder que la partie positive
    # Determiner les bornes de lag correspondant a fmin et fmax
    lag_min = int(fs / fmax)
    lag_max = int(fs / fmin)
    if lag_max >= len(corr):
        lag_max = len(corr) - 1
    # Chercher le pic dans la plage de lag
    segment = corr[lag_min:lag_max+1]
    if len(segment) == 0:
        return None
```

```
    peak_idx = np.argmax(segment) + lag_min
    if corr[peak_idx] <= 0:
        return None
    # Convertir le lag en frequence
    freq = fs / peak_idx
    return freq

def freq_to_midi(freq):
    """
    Convertit une frequence en numero de note MIDI.
    Si freq est None ou <= 0, retourne None.
    """
    if freq is None or freq <= 0:
        return None
    return int(np.round(69 + 12 * np.log2(freq / 440.0)))

def detect_pitch_sequence(
    wav_filename,
    frame_duration=0.030,
    pitch_range=(80.0, 400.0),
    min_note_duration=0.1
):
    """
    Fonction principale de detection de pitch et de notes.

    Parametres :
    - wav_filename : chemin vers le fichier .wav
    - frame_duration : duree (en secondes) de chaque trame
    - pitch_range : (fmin, fmax) domaine de recherche du pitch en Hz
    - min_note_duration : duree minimale d'une note en secondes

    Retourne :
    - matrice numpy de shape (N, 3) avec colonnes [temps, frequence, note_MIDI]
    """
    # Lecture du signal
    fs, signal_data = wav.read(wav_filename)

    # Si le signal est stereo, le convertir en mono
    if signal_data.ndim == 2:
        signal_data = signal_data.mean(axis=1)

    # Normaliser le signal entre -1 et 1 si necessaire
    if signal_data.dtype != np.float32 and signal_data.dtype != np.float64:
        max_val = np.iinfo(signal_data.dtype).max
        signal_data = signal_data.astype(np.float32) / max_val

    # Parametres de tramage
    frame_len = int(frame_duration * fs)
    hop_len = frame_len  # trames sans recouvrement
    num_frames = int(np.floor(len(signal_data) / hop_len))

    # Calcul du seuil d'energie (par exemple, 10% de l'energie max d'une trame)
    energies = []
    for i in range(num_frames):
        start = i * hop_len
        frame = signal_data[start : start + frame_len]
        energies.append(np.sum(frame ** 2) / len(frame))
    energy_threshold = 0.05 * np.max(energies)

    # Allocation des resultats
    times = np.zeros(num_frames)
    freqs = np.zeros(num_frames)
    midis = np.zeros(num_frames, dtype=int)

    fmin, fmax = pitch_range
```

```
    # Parcours des trames
    for i in range(num_frames):
        start = i * hop_len
        frame = signal_data[start : start + frame_len]
        times[i] = (start + frame_len / 2) / fs  # temps au centre de la trame

        if is_voiced(frame, energy_threshold):
            freq = autocorrelation_pitch(frame, fs, fmin, fmax)
            freqs[i] = freq if freq is not None else 0.0
            midi = freq_to_midi(freq) if freq is not None else 0
            midis[i] = midi if midi is not None else 0
        else:
            freqs[i] = 0.0
            midis[i] = 0

    # Detection des notes : regrouper les trames de meme note MIDI consecutives
    min_frames_per_note = int(np.ceil(min_note_duration / frame_duration))
    i = 0
    while i < num_frames:
        if midis[i] > 0:
            j = i + 1
            while j < num_frames and midis[j] == midis[i]:
                j += 1
            length = j - i
            if length < min_frames_per_note:
                # Pas assez long : considerer comme bruit -> zero
                freqs[i:j] = 0.0
                midis[i:j] = 0
            i = j
        else:
            i += 1

    # Construire la matrice resultat
    result = np.column_stack((times, freqs, midis))
    return result

# Utilisation
for file in os.listdir("./fichierzip/"):
    if file.endswith(".wav") or file.endswith(".WAV"):
        print(f"Traitement du fichier : {file}")

        result = detect_pitch_sequence(
            f"./fichierzip/{file}",
            frame_duration=0.025,
            pitch_range=(80.0, 300.0),
            min_note_duration=0.1
        )
        np.savetxt(f"./partie4_output/pitch_detection_{file[:-4]}.csv", result,
            delimiter=",",
                header="time,frequency,midi", comments='')
```

4.2 Analyse du code

Le code implémente l'analyse d'un signal audio (`.wav`) pour en extraire, trame par trame, la fréquence fondamentale (pitch) et la convertir en note MIDI. Les grandes étapes sont les suivantes :

1. **Lecture et prétraitement du signal**
   - Charger le fichier `.wav` (échantillonnage f_s) et, si nécessaire, convertir un signal stéréo en mono (moyenne des deux canaux).
   - Normaliser le signal dans $[-1, 1]$ si les échantillons sont stockés en entiers (int16, etc.).
2. **Découpage en trames (framing)**
   - Choisir une *durée de trame* (T_{frame}, typiquement 20–30 ms).

- Déterminer :
  - $N_{\text{frame}} = \lfloor T_{\text{frame}} \times f_{\text{s}} \rfloor$: nombre d'échantillons par trame,
  - overlap : taux de recouvrement (45 %–50 % recommandé),
  - $N_{\text{hop}} = \lfloor N_{\text{frame}} \times (1 - \text{overlap}) \rfloor$: décalage entre trames successives.
- Calculer le nombre total de trames N_{trames} de façon à couvrir tout le signal (on peut éventuellement tronquer la fin si elle ne remplit pas une trame complète).

3. **Calcul du seuil d'énergie**

   - Pour chaque trame, on calcule l'énergie moyenne :

   $$E_i \;=\; \frac{1}{N_{\text{frame}}} \sum_{n=0}^{N_{\text{frame}}-1} x_i^2[n]\,,$$

   où x_i est la trame i.
   - Le *seuil d'énergie* est fixé à

   $$E_{\text{seuil}} \;=\; \alpha \times \max_i E_i,$$

   avec $\alpha \approx 0{,}01$–$0{,}05$.
   - Toute trame dont l'énergie est inférieure à E_{seuil} est immédiatement classée « non-voisée » (silence, bruit faible) et reçoit $f = 0$ et MIDI $= 0$.

4. **Estimation du pitch par FFT (fonction `fft_pitch`)**

   - Pour chaque trame *voisée*, on applique d'abord une fenêtre de Hamming $w[n]$:

   $$x_{\text{win}}[n] = \big(x[n] - \bar{x}\big) \times w[n], \quad w[n] = 0{,}54 - 0{,}46\cos\!\Big(\tfrac{2\pi n}{N_{\text{frame}}-1}\Big)\,,$$

   afin de réduire les effets de discontinuités aux bords.
   - On calcule la FFT réelle (rFFT) $\mathcal{F}\{x_{\text{win}}\}$ et son module $M[k]$.
   - On détermine l'indice k^* correspondant à la plus grande amplitude dans la plage de fréquences $[f_{\min}, f_{\max}]$. La fréquence estimée vaut alors

   $$f^* \;=\; f_s \times \frac{k^*}{N_{\text{frame}}}\,.$$

   - Si l'amplitude maximale est trop faible (artefact, bruit), on renvoie $f =$ None.

5. **Conversion en note MIDI**

   $$\text{MIDI} \;=\; 69 + 12\,\log_2\!\Big(\tfrac{f^*}{440}\Big)\,, \quad \text{arrondi à l'entier le plus proche.}$$

   Si f^* est ≤ 0 ou None, on assigne MIDI $= 0$.

6. **Regroupement trame par trame en notes continues**

   - Définir une *durée minimale de note* $T_{\min}$ (par exemple 50 ms).
   - Calculer le nombre de trames minimal $N_{\min} = \lceil T_{\min}/(T_{\text{frame}} \times (1 - \text{overlap})) \rceil$.
   - Parcourir la séquence de MIDI$[i]$. Lorsqu'un segment contigu de même note MIDI dure moins de $N_{\min}$ trames, on l'étiquette comme artefact et on le ramène à zéro ($f = 0,\ \text{MIDI} = 0$).

7. **Assemblage du résultat** Le code retourne une matrice de taille $N_{\text{trames}} \times 3$:

   $$\big[t_i,\ f_i,\ \text{MIDI}_i\big]_{i=1..N_{\text{trames}}}, \quad t_i = \frac{(i-1)N_{\text{hop}} + N_{\text{frame}}/2}{f_s}\,.$$

   On peut ensuite exporter vers un CSV ou un tableur pour obtenir colonnes `time, frequency, midi`.

4.3 Variables clés à ajuster et leur rôle

- `frame_duration` (T_{frame}) : durée d'une trame en secondes (20–30 ms).
  - $\downarrow T_{\text{frame}} \rightarrow$ meilleure résolution temporelle (utile pour parole rapide), mais pic spectral moins précis.
  - $\uparrow T_{\text{frame}} \rightarrow$ meilleure précision fréquentielle (utile pour signaux musicaux stables), mais moins de flexibilité temporelle.
- `overlap` : fraction de recouvrement (souvent 0,5).
  - $\uparrow$ recouvrement $\rightarrow$ moins de trous temporels, meilleurs résultats sur signaux courts, mais coût de calcul plus élevé.
  - $\downarrow$ recouvrement $\rightarrow$ gain de temps, mais plus de « trames isolées » et de zéros.
- `energy_factor` (α) : proportion de l'énergie maximale fixant le seuil (1% à 5%).
  - $\uparrow \alpha \rightarrow$ seuil plus haut $\rightarrow$ on filtre davantage (conserve moins de trames voisées), risque d'ignorer des sons faibles.
  - $\downarrow \alpha \rightarrow$ seuil plus bas $\rightarrow$ on conserve plus de trames, mais on risque d'interpréter du bruit comme un pitch.
- `pitch_range` ($f_{\min}, f_{\max}$) : plage de fréquences (Hz) à analyser.
  - Choisir $[f_{\min}, f_{\max}]$ en fonction du signal :
    - Voix masculine adulte $\approx 80 - 180Hz$
    - Voix féminine $\approx 150 - 270Hz$
    - Flûte traversière $\approx 250 - 2000Hz$
  - Plage trop étroite $\rightarrow$ on rate le pitch si la fondamentale du signal est en dehors.
  - Plage trop large $\rightarrow$ on peut capter des harmoniques ou du bruit comme faux pic.
- `min_note_duration` ($T_{\min}$) : durée minimale d'une note (à partir de 0,03 s).
  - $\uparrow T_{\min} \rightarrow$ on supprime les segments très courts (bruits, ouvertures de consonnes), mais on risque de découper des notes légères ou rapides.
  - $\downarrow T_{\min} \rightarrow$ on conserve même des notes très brèves, mais on accepte davantage de faux pitch (artefacts).

4.4 Résultats

En analysant les fichiers CSV générés, on remarque que cette methode marche bien pour certains enregistrements mais pas d'autres. On remarque également que certains intervalles sont mal détéctés avec f = 0 et midi = 0 alors qu'il n'y a pas de silence dans l'audio.

Ceci est dû au différents paramètres de la méthode, qui peuvent être ajustés pour améliorer la détection du pitch. Par exemple, en réduisant le seuil d'énergie ou en élargissant la plage de fréquences, on peut obtenir de meilleurs résultats sur certains signaux.

On a mis deux exemples de quelques lignes des fichiers CSV générés (page 28) et (page 29). Pour le fichier contenant la détection de pitch pour la flute (page 29), on a enlevé quelques lignes à l'affichage pour montrer les différentes notes détéctées.

4.5 Sources fréquentes d'erreurs (intervalles avec $f = 0$)

On a discuté précédémment du role des variables clés à ajuster pour la détection de pitch. Cependant, il est fréquent de rencontrer des intervalles où la fréquence détectée est $f = 0$ ou MIDI $= 0$ malgré un signal audible. Voici les principales raisons possibles et leurs remèdes :

1. **Seuil d'énergie trop élevé** Même un signal stable et régulier (voix, instrument, etc.) peut présenter une énergie inférieure à E_{seuil} s'il est enregistré avec un faible niveau sonore.
   - *Symptôme* : toutes les trames ont $f = 0$ malgré un son continu clairement perceptible.
   - *Remède* : abaisser `energy_factor` (par exemple de 0,05 à 0,01 ou 0,005).

2. **Plage de fréquences mal adaptée** Si la fondamentale du signal est située en dehors de l'intervalle $[f_{\min}, f_{\max}]$, l'algorithme basé sur la FFT risque de ne détecter aucun pic significatif.

   - *Symptôme* : toutes les trames ont $f = 0$, bien que le signal soit périodique, car sa fréquence fondamentale est hors de la plage spécifiée.
   - *Remède* : élargir $[f_{\min}, f_{\max}]$ pour inclure la fréquence réelle attendue.

3. **Durée de note trop courte par rapport à la durée de trame** Si un son est extrêmement bref (inférieur à T_{frame}), il se peut qu'aucune trame complète ne soit disponible pour l'analyse, entraînant l'absence de détection de fréquence.

   - *Symptôme* : sons très courts $\rightarrow$ trames vides ou $f = 0$.
   - *Remède* : réduire `frame_duration` (par exemple de 25 ms à 10 ms) et ajuster `min_note_duration` en conséquence.

4. **Fenêtrage insuffisant ou absence de recouvrement** Sans application d'une fenêtre (comme Hamming) et sans recouvrement entre les trames, l'analyse spectrale peut être dégradée (effets de discontinuité), réduisant la clarté des pics de fréquence.

   - *Symptôme* : certaines trames produisent un spectre bruité sans pic dominant $\rightarrow f = 0$.
   - *Remède* :
     - Appliquer une fenêtre de Hamming sur chaque trame.
     - Utiliser un recouvrement de overlap $\approx 0{,}5$ pour que chaque portion du signal soit analysée plusieurs fois.

5. **Segments non périodiques (parole, bruit, consonnes)** Certains segments de signal, notamment les bruits transitoires, consonnes occlusives ou affriquées dans la parole, ne présentent pas de périodicité stable. L'algorithme retourne alors $f = \text{None}$, soit $f = 0$.

   - *Symptôme* : portions du signal (notamment en parole) avec $f = 0$, surtout sur les consonnes.
   - *Remède* : ce comportement est normal ; seules les portions périodiques (voyelles, instruments tenus) doivent donner une fréquence définie.

6. **Artifacts liés au post-traitement des notes** Si la durée minimale de note (`min_note_duration`) est trop longue, des notes courtes mais valides peuvent être éliminées.

   - *Symptôme* : séquences de trames avec $f = 0$ à l'intérieur de passages pourtant musicaux ou chantés.
   - *Remède* : réduire `min_note_duration` proportionnellement à T_{frame} et à overlap pour permettre la détection de notes brèves.

4.6 Conclusion et remarques pertinentes

En ayant connaissance du type de signal analysé (voix, instrument, bruit), on peut ajuster les paramètres de l'algorithme pour obtenir une détection de pitch plus fiable. Voici quelques recommandations générales :

- **Approche FFT vs. autocorrélation** : l'estimation par FFT est plus robuste sur des signaux sinusoïdaux purs (même très courts), car elle détecte directement la raie fondamentale dans le domaine fréquentiel.
- **Importance du fenêtrage et du recouvrement** : un hachage correct (Hamming) et un recouvrement (environ 50 %) sont essentiels pour éviter des artefacts de bord qui génèrent des zéros « fausses détections ».
- **Réglage fin des paramètres** :
  - T_{frame} : 15–30 ms selon la nature du signal (voix vs instrument vs sinus très court).
  - α (energy_factor) : 0,005–0,05 : à adapter en fonction de l'amplitude relative du signal et du bruit de fond.
  - $[f_{\min}, f_{\max}]$: couvrir exactement le registre attendu (une plage plus large influe plus de bruit, une plage trop étroite ratera la fondamentale).
  - $T_{\min}$: réguler la suppression des artefacts très courts (consonnes, clics) sans éliminer les notes souhaitées.

- **Limites connues** :
  - Sur les plages de parole (consonnes, transitions), on ne récupère pas de pitch (zéros), mais ceci est normal : il n'y a pas de fondamentale stable.
  - Les instruments très riches en harmoniques (pianos, cordes pincées) peuvent conduire à un « faux pic » si la première harmonique est plus forte que la fondamentale : on peut alors détecter la seconde harmonique comme pitch. Pour corriger, on implante un post-filtrage (cherche le « plus bas » des pics récurrents).
  - Bruits de fond forts ou distorsion du signal se traduisent par des pics peu fiables : parfois $f = 0$ même si le bruit est audible.

Table 1: Extrait du fichier CSV pour la détection de pitch KATCHATU

rownum	time	frequency	midi
1	$1.25 \cdot 10^{-2}$	0	0
2	$3.75 \cdot 10^{-2}$	0	0
3	$6.25 \cdot 10^{-2}$	0	0
4	$8.75 \cdot 10^{-2}$	0	0
5	0.11	0	0
6	0.14	0	0
7	0.16	0	0
8	0.19	0	0
9	0.21	0	0
10	0.24	0	0
11	0.26	219.18	57
12	0.29	222.22	57
13	0.31	222.22	57
14	0.34	222.22	57
15	0.36	0	0
16	0.39	0	0
17	0.41	0	0
18	0.44	148.84	50
19	0.46	150.94	50
20	0.49	148.84	50
21	0.51	148.84	50
22	0.54	0	0
23	0.56	0	0
24	0.59	0	0
25	0.61	0	0
26	0.64	0	0
27	0.66	0	0
28	0.69	0	0
29	0.71	187.13	54
30	0.74	187.13	54
31	0.76	186.05	54
32	0.79	187.13	54
33	0.81	0	0
34	0.84	0	0
35	0.86	0	0
36	0.89	0	0
37	0.91	0	0
38	0.94	0	0
39	0.96	0	0

Table 2: Extrait du fichier CSV pour la détection de pitch fluteircam

rownum	time	frequency	midi
31	0.76	219.18	57
32	0.79	219.18	57
33	0.81	219.18	57
34	0.84	219.18	57
35	0.86	219.18	57
36	0.89	219.18	57
37	0.91	219.18	57
38	0.94	219.18	57
39	0.96	219.18	57
40	0.99	219.18	57
41	1.01	219.18	57
42	1.04	219.18	57
43	1.06	220.69	57
44	1.09	248.06	59
45	1.11	246.15	59
46	1.14	246.15	59
47	1.16	246.15	59
48	1.19	246.15	59
49	1.21	246.15	59
50	1.24	246.15	59
61	1.51	248.06	59
62	1.54	248.06	59
63	1.56	246.15	59
64	1.59	246.15	59
65	1.61	250	59
66	1.64	262.3	60
67	1.66	262.3	60
68	1.69	262.3	60
69	1.71	262.3	60
70	1.74	262.3	60
81	2.01	264.46	60
82	2.04	262.3	60
83	2.06	262.3	60
84	2.09	262.3	60
85	2.11	260.16	60
86	2.14	0	0
87	2.16	220.69	57
88	2.19	220.69	57
89	2.21	220.69	57