

## Faculté Polytechnique

### Projet de Traitement de signal

Localisation sonore en temps réel

Rapport de projet  
23 décembre 2024

Axel FOUCART, Claire D'HAENE, Thomas TOMSEN



Sous la direction de :  
Monsieur Thierry DUTOIT (Professeur)  
Monsieur François MARELLI (Assistant)

Année académique 2024-2025

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Système hors ligne</b>	<b>2</b>
2.1	Génération de données et jeu de données . . . . .	2
2.1.1	Génération de signaux sinusoïdaux . . . . .	2
2.1.2	Lecture de jeux de données . . . . .	3
2.2	Buffering . . . . .	4
2.2.1	Création d'un <i>buffer</i> circulaire . . . . .	4
2.2.2	Visualisation du contenu d'un <i>buffer</i> circulaire . . . . .	5
2.3	Pre-processing . . . . .	6
2.3.1	Normalisation . . . . .	6
2.3.2	Downsampling . . . . .	7
2.4	Cross-correlation . . . . .	14
2.5	Localisation . . . . .	15
2.5.1	TDOA . . . . .	15
2.5.2	Système d'équations de localisation . . . . .	15
2.6	Précision et vitesse du système . . . . .	17
2.6.1	Précision de notre programme . . . . .	18
2.6.2	Vitesse de notre programme . . . . .	19
2.6.3	Conclusion des observations . . . . .	20
<b>3</b>	<b>Localisation en temps réel</b>	<b>21</b>
3.1	Application pratique possible et son fonctionnement . . . . .	21
3.2	Feedback visuel et impression de temps réel du système . . . . .	21
<b>4</b>	<b>Conclusion</b>	<b>23</b>

# Chapitre 1

## Introduction

Dans le cadre du cours de traitement du signal, nous avons dû réaliser un projet. Ce dernier a pour but de construire un système qui traite les signaux audio pour estimer la position d'un son.

Nous avons commencé par une étape hors ligne afin de vérifier les performances de l'application puisqu'il s'agit d'une application en temps réel. Pour ce faire, nous avons utilisé un jeu de données nommé "LocateClaps" contenant des sons de claquements de mains enregistrés à l'aide de trois microphones.

Des sons aigus sont enregistrés simultanément à l'aide de ces microphones. Chaque son est positionné en fonction de son angle avec l'axe x, tout en restant à environ 3 mètres du centre du réseau de microphones. Ces données sont ajoutées au nom du fichier correspondant. C'est-à-dire que les noms de fichiers seront de la forme  $Mx_{\theta}.wav$  avec  $x$  l'identifiant du micro et signifie que la source du bruit se trouve à  $\theta^{\circ}$ .

Après cela, nous avons terminé par une étape lors de l'exécution, pour nous assurer que l'application fonctionne correctement en temps réel. Dans cette optique, nous ferons usage de l'équipement suivant : un 'Raspberry Pi 3 model B' et un 'Respeaker 6-Mic Circular Array'. Dans ce projet, il n'est requis que trois microphones, alors que six sont disponibles sur la carte audio. Nous avons donc sélectionné les données des micros n°1, n°3 et n°5.

Chaque étape pour y parvenir sera détaillée dans les prochaines sections.

# Chapitre 2

## Système hors ligne

Comme mentionné dans la section précédente, cette section nous a permis de vérifier les performances de l'application en utilisant le jeu de données 'LocateClaps'.

Dans cette partie, nous allons vous donner des explications sur la façon dont nous avons géré les étapes suivantes :

- génération de données et utilisation du jeu de données,
- gestion du buffer,
- pre-processing,
- normalisation,
- downsampling,
- cross-correlation,
- localisation,
- TDOA,
- système d'équation de localisation, et,
- la précision et la rapidité des systèmes.

### 2.1 Génération de données et jeu de données

Tout d'abord, afin de garantir le bon fonctionnement de notre programme, il est indispensable de le tester sur des signaux qui donnent des résultats prédictibles, tels que des sinusoïdes ou des jeux de données déjà préparés. Pour accomplir cela, nous avons dû mettre en place 2 fonctions.

#### 2.1.1 Génération de signaux sinusoïdaux

Nous avons commencé par une simple fonction capable de générer  $N$  échantillons d'un signal sinusoïdal avec une fréquence de signal  $f$ , une fréquence d'échantillonnage  $fs$  et une amplitude  $A$ . Il est important de noter que la phase ( $\varphi$ ) sera toujours nulle ici.

Nous savons qu'un échantillon pris à un temps  $t$  dans cette fonction sinus a pour valeur,  $A \sin(\omega t + \varphi)$  avec  $\omega = 2\pi f$  :

$$\begin{aligned}y &= A \cdot \sin(\omega t + \varphi) \\&= A \cdot \sin(2\pi f t + \varphi) \\&= A \cdot \sin(2\pi f t)\end{aligned}$$

N'oublions pas qu'un échantillonnage est la prise de mesure à un intervalle régulier de la valeur du signal sinusoïdal. Cet intervalle s'appelle la période d'échantillonnage et il vaut  $T_s = \frac{1}{f_s}$ .

La prise de mesure commence à l'instant 0 et se termine quand on a obtenu tous les échantillons demandés  $N$ , et vu que le temps entre chaque mesure est la période  $T_s$ , le temps total sera de  $N \cdot T_s$ .

Nous avons donc utilisé toutes ces affirmations pour créer la fonction "create\_sine\_wave" ci-dessous, qui retourne un tuple composé des temps de mesure pour chaque échantillon, ainsi que leurs valeurs mesurées.

```

1 def create_sine_wave(f, A, fs, N):
2     sampling = np.arange(0, N/fs, 1/fs)
3     sin_samples = np.sin(2 * np.pi * sampling * f)
4     return sampling, A * sin_samples

```

Nous avons pris un exemple afin de mieux pouvoir visualiser son résultat en utilisant  $f_s = 44.1 * 1000$  (44.1 kHz = 441000 Hz) en fréquence d'échantillonnage,  $N = 8000$  échantillons,  $f = 20$  (Hz) en fréquence de signal et  $A = 8$  (V) en amplitude.

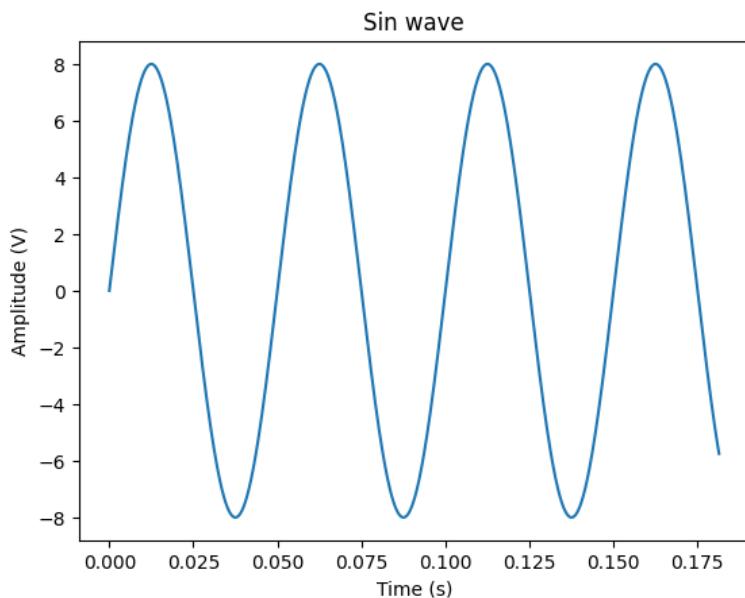


FIGURE 2.1 – Visualisation du résultat de l'appel à la fonction : `create_sine_wave(20, 8, 44.1 * 1000, 8000)`

Nous pouvons observer que nous obtenons bien un signal avec une amplitude de 8V en temps discret  $t(s)$ . On peut aussi noter que la période est de 0.05 s.

### 2.1.2 Lecture de jeux de données

Dans cette section, nous avons simplement utilisé *wavfile* de la librairie *scipy* afin de lire nos fichiers *.wav* et ensuite, la fonction *glob* permettant d'enregistrer plusieurs chemins d'accès à

des dossiers et/ou à des fichiers dans la même liste.

```
1 def read_wavefile(path):  
2     a,b=wf.read(path)  
3     return b,a
```

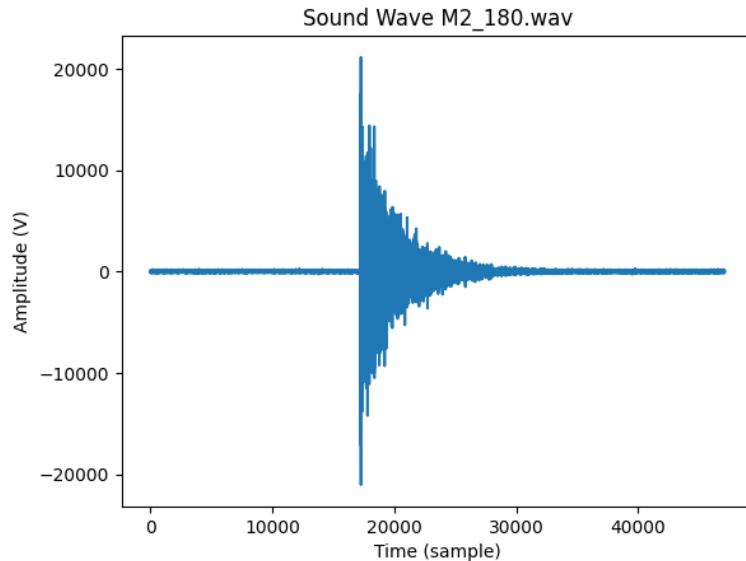


FIGURE 2.2 – Visualisation de la forme du signal contenue dans le fichier `M2_180.wav`

```
1 LocateClaps = "./LocateClaps"  
2 files = glob(f"{LocateClaps}/*.wav")
```

Nous utiliserons plus tard la variable `files` dans la section 2.6.

## 2.2 Buffering

Afin de pouvoir manipuler une arrivée continue de données, il est nécessaire d'utiliser un *Buffer*. Ce dernier nous a permis de mettre en attente les nouvelles valeurs qui arrivaient le temps que les précédentes soient traitées.

Etant donné qu'une fois traitée, une valeur n'est plus utile à conserver, il est utile d'opter pour un *buffer* circulaire (ou en anglais, un *ring buffer*) afin d'avoir un *buffer* avec une taille limitée dans lequel les anciennes données seront écrasées par les plus récentes lorsque la taille maximum de ce dernier est dépassée.

### 2.2.1 Crédation d'un *buffer* circulaire

Heureusement, Python dispose déjà d'une implémentation de ce dernier, connue sous le nom de `deque`.

Pour l'initialisation de notre *buffer* circulaire, nous avons donc créé une fonction `create_ringbuffer(maxlen)` qui retourne une instance de `deque` de taille maximum, `maxlen`.

```
1 def create_ringbuffer(maxlen) -> deque:
2     return deque maxlen=maxlen)
```

## 2.2.2 Visualisation du contenu d'un *buffer* circulaire

Nous allons utiliser pour les exemples suivants un *buffer* circulaire d'une taille maximale de 750.

Afin de vérifier si le buffer fonctionne correctement, nous avons ajouté un par un les échantillons de notre signal sinusoïdal comme expliqué dans l'exemple de la section 2.1.1 afin de simuler une arrivée de données continue échantillonnée à une fréquence de 44.1 kHz. Ceci dans le but d'afficher le contenu du buffer après avoir ajouté l'équivalent de 0.1s et 0.15s de données.

Afin d'observer le contenu d'un buffer après  $t$  secondes, nous avons dû trouver un  $i$  tel qu'il représente le nombre d'échantillons ajoutés dans le *buffer* à une fréquence de  $f_s$  après  $t$  secondes. Nous avons donc utilisé la formule suivante, en sachant que  $t = \frac{i}{f_s}$  :

$$i = t \cdot f_s$$

En d'autres termes, après avoir rajouté  $i$  échantillons dans notre buffer (à une fréquence de  $f_s$ ),  $t$  secondes se seront écoulées.

Nous pouvons aussi trouver le temps total d'attente dans notre *buffer* avec le calcul suivant :

$$t_{\max} = \frac{\text{Taille max}}{f_s} = \frac{750}{44.1\text{kHz}} \approx 0.017s$$

Visualisons maintenant le contenu du buffer après 0.1s et 0.15s, c'est-à-dire après avoir rajouté 4410 ( $= 0.1 \cdot 44.1$  kHz) et 6615 ( $= 0.15 \cdot 44.1$  kHz) échantillons respectivement.

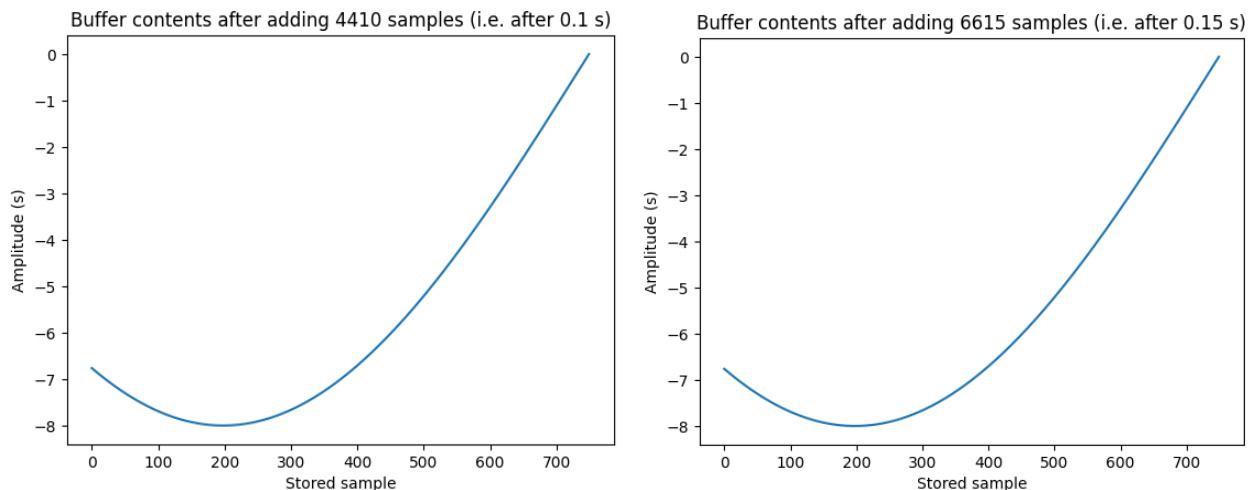


FIGURE 2.3 – Affichage du contenu du *buffer* à 0.1s et 0.15s respectivement

On remarque que le contenu paraît identique, ce qui peut être surprenant, mais si on regarde la visualisation du signal d'entrée à 0.1 s et 0.15 s, on remarque que ce comportement est normal

puisque que la période de ce signal est de 0.05 s et que le temps entre nos deux mesures est aussi de 0.05s ( $= 0.15 - 0.1$ ).

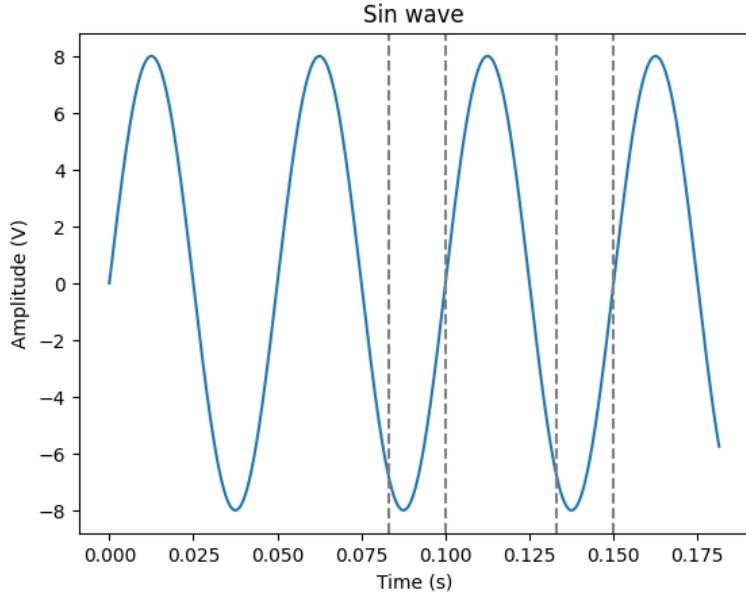


FIGURE 2.4 – Visualisation du signal d’entrée avec des lignes représentant les moments de début et de fin de l’affichage du contenu du *buffer* à 0.1 s et 0.15 s respectivement.

## 2.3 Pre-processing

La plupart des signaux ne sont pas traitables tels quels, il est donc nécessaire d’effectuer plusieurs opérations dessus pour pouvoir utiliser leurs données. Nous allons donc ici normaliser et échantillonner les signaux afin de pouvoir les analyser au mieux.

### 2.3.1 Normalisation

L’étape de normalisation va nous permettre de contenir notre amplitude entre  $[-1; 1]$  tout en conservant la fréquence originelle.

En effet, les signaux peuvent avoir des amplitudes complètement différentes. De ce fait, la normalisation nous permet donc de ramener les signaux sur une échelle commune facilitant leur comparaison.

Afin de procéder à l’opération de normalisation, nous avons tout d’abord créé une fonction nous permettant de récupérer la valeur maximum parmis jeu de données reçu. Après avoir récupéré cette valeur, il nous suffit donc de diviser toutes les données par cette dernière.

```

1 def get_max(s):
2     max = float('-inf')
3     for i in s:
4         if i > max:
```

```

5         max = i
6     return max
7
8
9 def normalise(s):
10    max = get_max(s)
11    return s / max

```

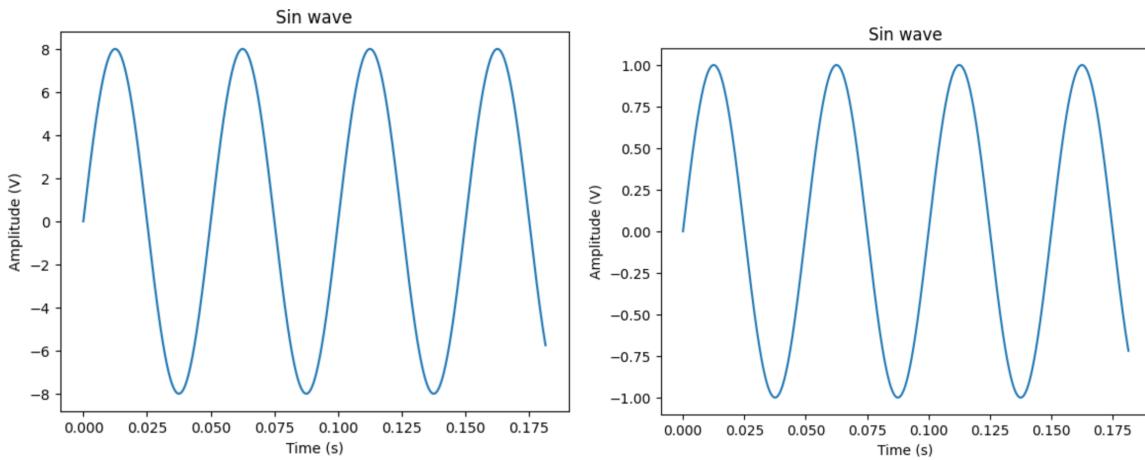


FIGURE 2.5 – Visualisation d'un signal sinusoïdal créé et de sa forme normalisée.

Nous remarquons qu'entre les deux signaux, l'apparence ne change en rien. La fréquence n'est pas altérée mais l'amplitude si, les signaux sont maintenant limités à une amplitude comprise dans l'intervalle  $[1-; 1]$ , ce qui correspond bien à une opération de normalisation.

### 2.3.2 Downsampling

#### Signal analysis

Pour visualiser le spectrogramme du signal, il suffira juste d'utiliser la fonction prédéfinie dans plt. Dans un spectrogramme, la fréquence est ordonnée, donc il est nécessaire de mesurer sur cet axe, là où le signal est le plus intense. Soit, là où il est le plus haut et le plus "clair".

```

1 f, data = read_wavefile("LocateClaps/M1_0.wav")
2 x = plt.specgram(normalise(data), Fs=f)

```

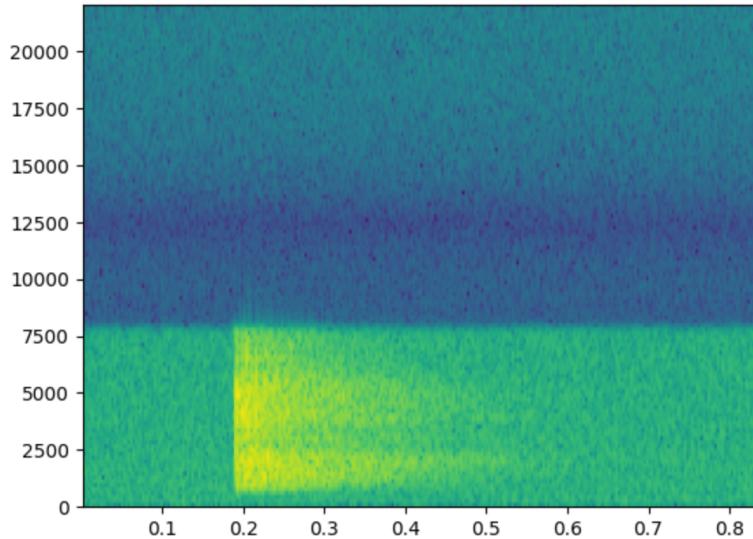


FIGURE 2.6 – Spectrogramme de la piste audio M1\_0.

Nous remarquons que la plus grande fréquence ici est de 8 kHz. Nous savons que par le théorème de Shannon une bonne valeur d'échantillonnage est bornée par :  $f_s \geq 2.f_e$ . Nous en déduisons alors qu'une bonne valeur d'échantillonnage optimale est de 16 kHz. De cette manière, on retient un maximum des valeurs utiles, sans en avoir de trop afin de limiter le nombre d'opérations à effectuer pour analyser le signal et ainsi, optimiser la vitesse de traitement.

### Anti-aliasing filter

Les filtres Anti-aliasings vont nous permettre de ne garder que les fréquences importantes de notre signal, par exemple, ils permettent de retirer le bruit qui peut s'ajouter à notre signal. Nous avons dans ce projet le choix entre deux filtres, **Chebychev** et **Cauer**. Tous deux sont déjà implémentés dans la librairie Signal de Scipy par le biais de deux fonctions.

Les fonctions `create_filter_cheby` ainsi que `create_filter_cauer`, serviront de raccourcis afin de ne pas devoir faire appel à deux fonctions pour générer un filtre.

```

1 def create_filter_cheby(wp, ws, gpass, gstop, fs):
2     N, wn = signal.cheb1ord(wp, ws, gpass, gstop, fs=fs)
3     return signal.cheby1(N, gpass, wn, "lowpass", False, fs=fs)
4
5 def create_filter_cauer(wp, ws, gpass, gstop, fs):
6     N, wn = signal.ellipord(wp, ws, gpass, gstop, fs=fs)
7     return signal.ellip(N, gpass, gstop, wn, 'lowpass', False, fs=
    fs)
```

### Exemple d'utilisation :

Afin de tester ces deux filtres, nous allons prendre un signal qui contient deux ondes sinusoïdales, une avec une amplitude de 1000 V et une fréquence de 8500 Hz, ainsi qu'une autre avec

une amplitude de 20 V et une fréquence de 7500 Hz. Ce signal sera utilisé comme exemple dans le reste de cette section.

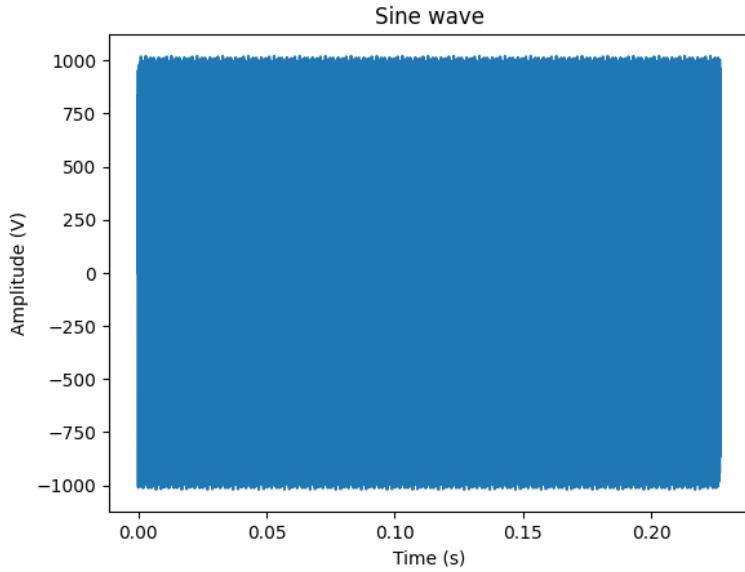


FIGURE 2.7 – Signal constitué de deux ondes sinusoïdales d'amplitude de 1000 et 20 V, et de fréquence à 8500 Hz et 7500 Hz respectivement.

Ici, nous avons décidé d'éliminer les fréquences au-delà de 8000 Hz, car le spectrogramme dans la figure 2.6 montre que la majorité de l'information utile se trouve en dessous de cette borne pour notre application finale.

```
1 wp = 7700
2 ws = 8500
3 rp = 0.1
4 rs = 70
5 (B_cauer, A_cauer) = create_filter_cauer(wp, ws, rp, rs,
   44100)
6
7 (B_cheby, A_cheby) = create_filter_cheby(wp, ws, rp, rs,
   44100)
```

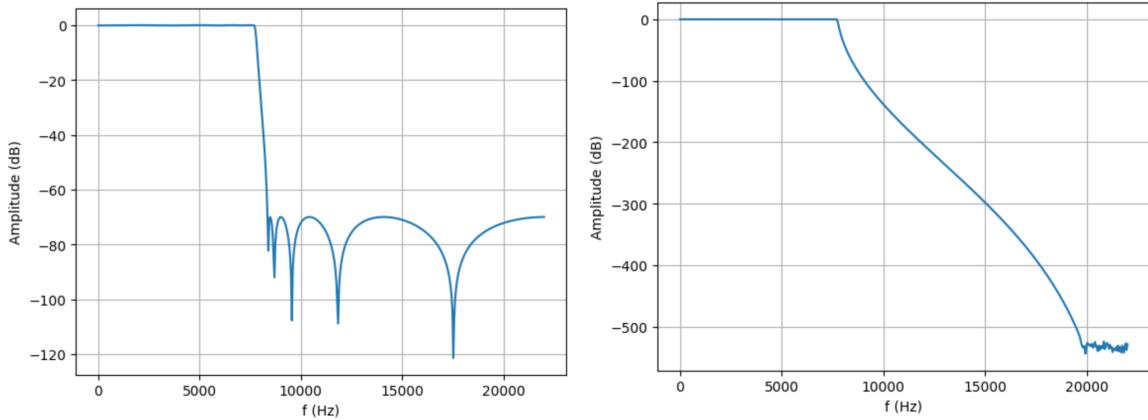


FIGURE 2.8 – Filtres de Cauer et de Chebychev

Nous avons choisi ces valeurs car nous voulions conserver les valeurs autour de la fréquence que nous avons déterminée de 8000Hz, d'où le fait que nous entourions par 7700 et 8500. Ensuite, notre objectif était que la pente soit la plus abrupte possible, tout en conservant un nombre de lobes pas trop important pour que notre filtre soit assez complexe et stable. La visualisation de la complexité du filtre peut-être visualisée avec la quantité de lobes formés.

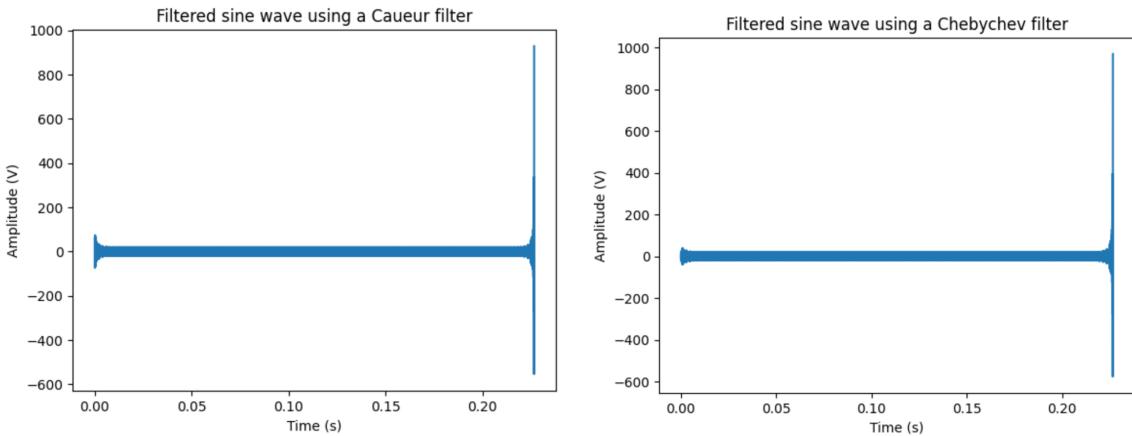


FIGURE 2.9 – Application du filtre de Cauer et de Chebychev sur notre signal.

Nous remarquons que les filtres s'appliquent bien, mais qu'il y a une grande instabilité à la fin de notre signal. Cela se justifie ici par le fait que notre signal n'est pas infini puisqu'il est généré artificiellement et possède un nombre d'échantillons discret.

Pour choisir quel filtre nous conviendrait le mieux, nous les avons tout d'abord analysés graphiquement. Les deux graphiques étant relativement similaires, nous n'avons su en tirer de conclusion déterminante. Nous nous sommes alors demandés comment l'appliquer à notre problème, quel serait l'élément majeur qui les différencierait ? Nous en sommes arrivés à nous dire que c'était le temps étant donné que nous sommes dans un système en temps réel. Le filtre le plus efficace le plus tôt possible serait donc celui que l'on choisirait.

Pour appliquer les filtres à notre signal, nous utilisions la fonction `filtfilt` qui applique le filtre une fois en avant et en arrière, nous avons donc essayé avec une autre fonction qui est `lfilter` et qui n'applique qu'une seule fois le filtre en avant et voici le résultat observé :

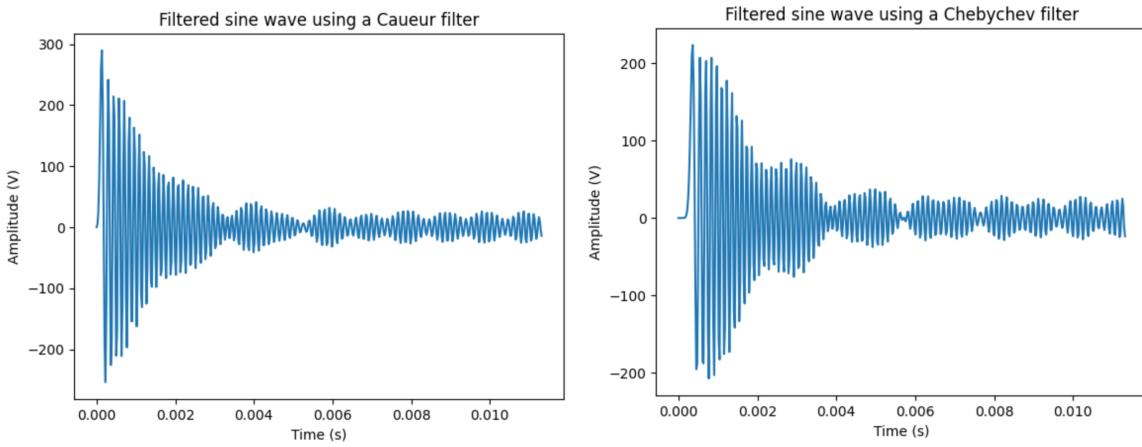


FIGURE 2.10 – Application du filtre de Cauer et de Chebychev (avec la fonction `lfilter`) sur les 500 premiers échantillons de notre signal.

Nous observons que le filtre de Cauer s'applique un peu plus tôt à notre signal que celui de Chebychev. C'est donc la raison pour laquelle nous avons décidé d'utiliser le filtre de Cauer pour la suite du projet.

## Decimation

La décimation est une étape importante de l'analyse d'un signal. En effet, il se peut que la fréquence d'échantillonage soit trop élevée et que nous recevons donc un jeu de données bien trop important. Cette opération nous permet donc de réduire la quantité de données à traiter pour le signal reçu, tout en baissant *artificiellement* la fréquence d'échantillonnage, car le nouveau signal est en fait le signal original, mais échantillonné à une fréquence inférieure. Cette étape est très simple en Python grâce aux opérations de *slicing* de liste comme montré dans le code ci-dessous.

```
1 def decimation(data, M=3):
2     return data[::M]
```

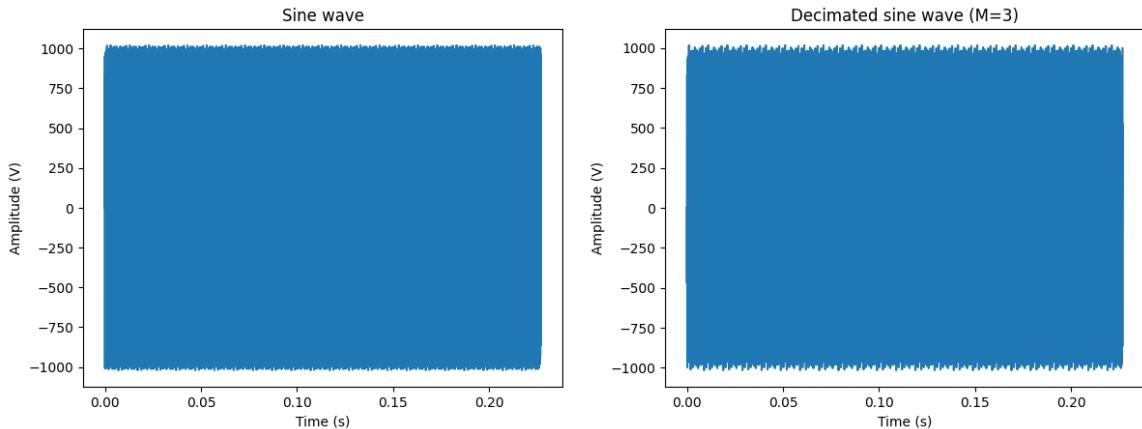


FIGURE 2.11 – Visualisation du signal original et de l'application d'une décimation avec  $M = 3$

Visualisons aussi l'effet de la décimation sur le domaine fréquentiel de notre signal :

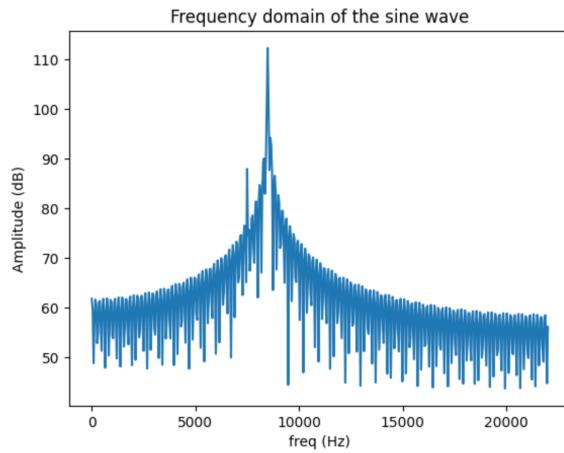


FIGURE 2.12 – Visualisation du signal en domaine de fréquence

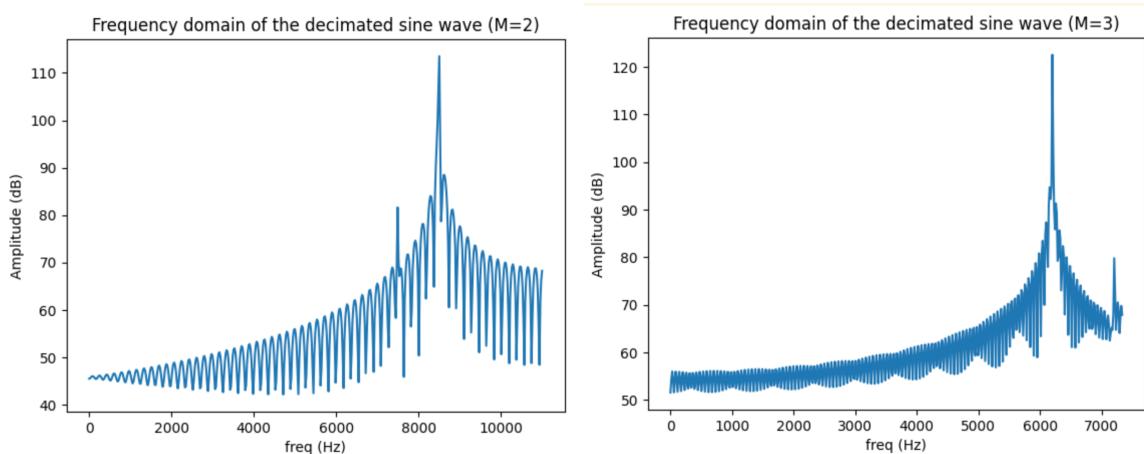


FIGURE 2.13 – Visualisation du signal décimé d'un facteur M=2 et M=3

Nous remarquons une grande similarité entre le signal de base et lorsqu'il est décimé d'un facteur 2. Cependant, pour un facteur de décimation de 3, nous remarquons que les "pics" ne sont pas placés au même endroit, n'ont pas la même allure. Cette incohérence pourrait s'expliquer qu'avec un facteur 3, nous avons une fréquence d'échantillonage de 14700 Hz, ce qui ne respecte pas le théorème de Shannon dans notre cas (nous devrions prendre une fréquence d'échantillonage supérieure à 16000 Hz). Le phénomène observé sur la figure M=3 est donc de l'**Aliasing**.

### Filtre et décimation :

Nous avons maintenant tous les éléments nécessaires pour créer une fonction appliquant le *downsampling*, celle-ci va d'abord appliquer un filtre donné et appliquer une décimation de  $M$  sur le signal résultant :

```

1 def downsampling(sig, B, A, M):
2     # Apply filter
3     sig = sc.filtfilt(B, A, sig)
4     # Apply decimation
5     return decimation(sig, M)

```

Observons maintenant l'effet du *downsampling* sur notre signal pour les deux filtres ainsi que pour  $M = 2$  et  $M = 3$  :

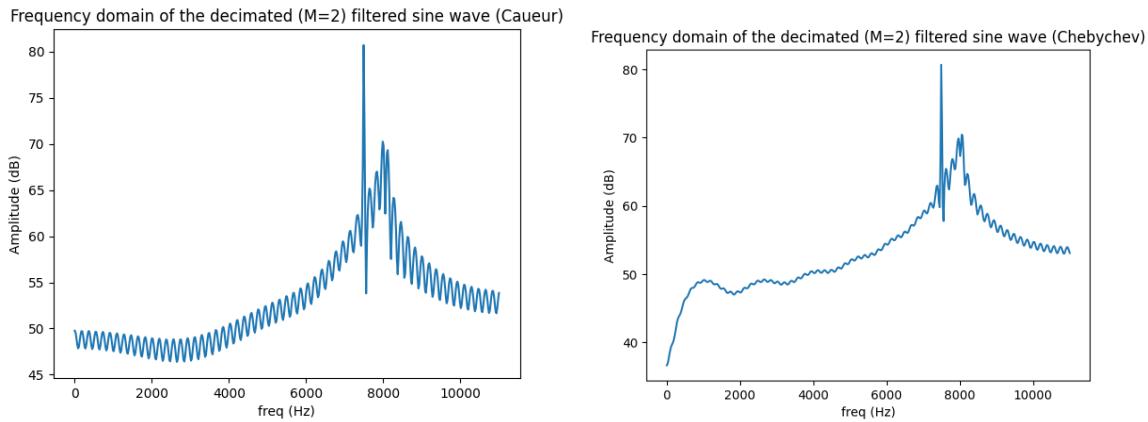


FIGURE 2.14 – Visualisation du domaine fréquentiel du signal filtré par Cauer et Chebychev, ainsi que décimé d'un facteur  $M=2$

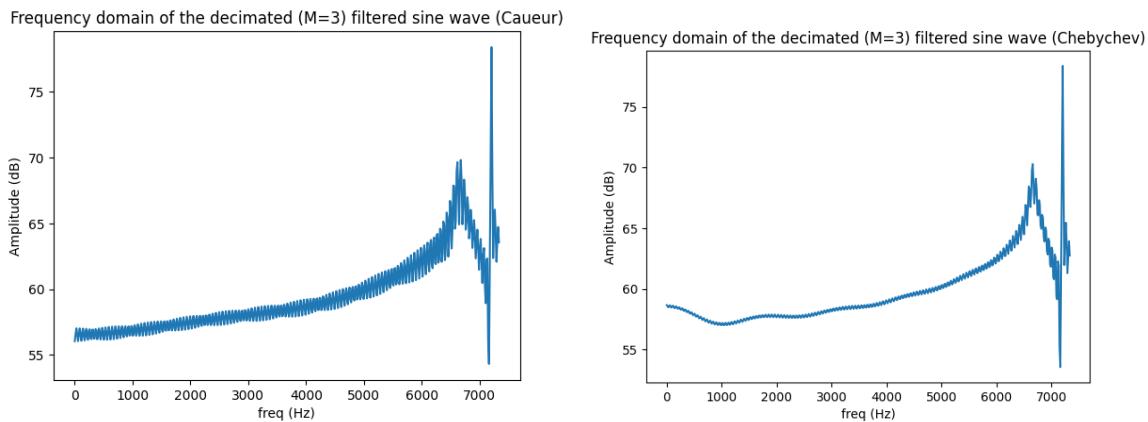


FIGURE 2.15 – Visualisation du domaine fréquentiel du signal filtré par Cauer et Chebychev, ainsi que décimé d'un facteur  $M=3$

Nous remarquons, notamment sur le graphe décimé d'un facteur 2, que le pic présent aux alentours de 8500Hz a été considérablement réduit, montrant bien que le filtre s'applique

correctement. Nous observons d'ailleurs toujours l'effet d'*aliasing* lorsque le graphe est décimé d'un facteur 3.

## 2.4 Cross-correlation

Cette étape nous permet de mesurer à quel point deux signaux sont similaires en utilisant le décalage observé entre les signaux.

Plus précisément, celle-ci nous permettra dans la section 2.5.1 d'identifier quel micro est utilisé grâce au décalage observé (entre plusieurs signaux), c'est-à-dire, identifier qui a reçu le signal en premier dans le domaine temporel.

Pour procéder, nous avons remarqué que la cross-correlation est étroitement liée à la convolution grâce aux formules suivantes :

$$\text{Convolution} : \{g * h\} = F^{-1}\{F\{g\} \cdot F\{h\}\}$$

$$\text{Cross-correlation} : \{g(t) * h(t)\}(t) = g(t) * h(-t) = F^{-1}\{F\{g\} \cdot \overline{F\{h\}}\}$$

Nous avons appliqué la formule de *cross-correlation* dans la fonction suivante avec `in1` et `in2` représentant  $g(t)$  et  $h(t)$  respectivement :

```

1 def fftxcorr(in1, in2):
2     n_1, n_2 = len(in1), len(in2)
3     n = n_1 + n_2 - 1
4
5     x = np.fft.fft(in1, n) * np.conjugate(np.fft.fft(in2, n))
6     return np.fft.fftshift(np.fft.ifft(x, n))

```

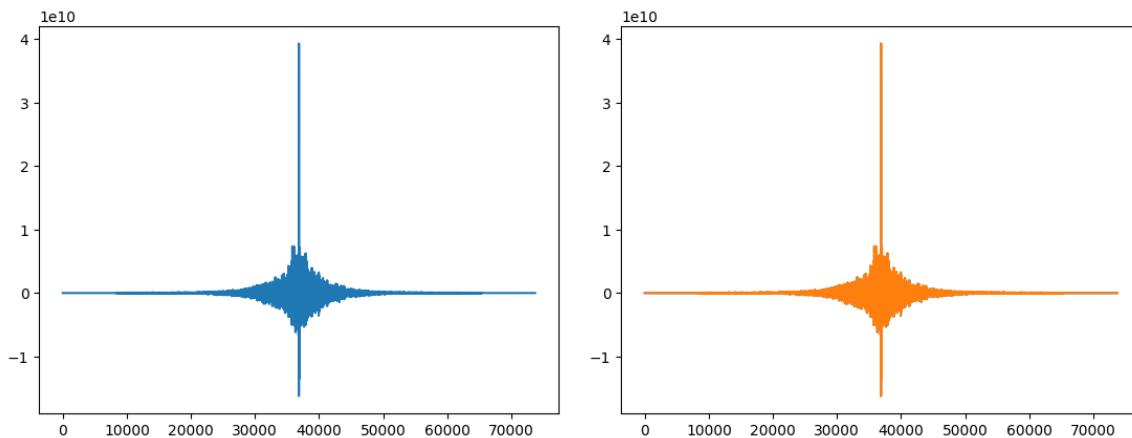
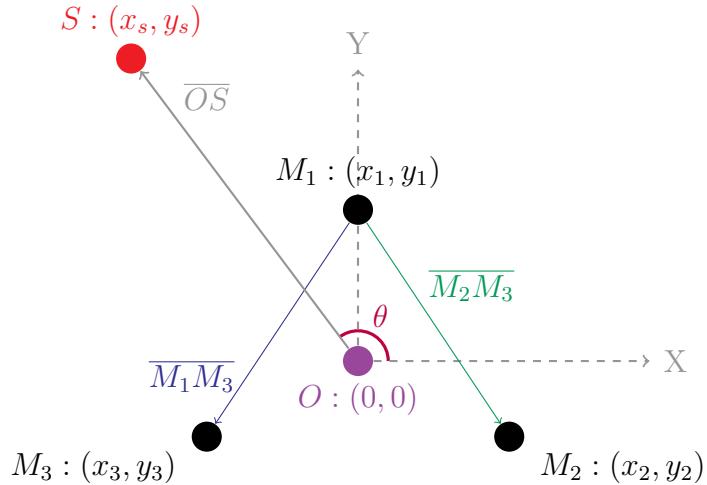


FIGURE 2.16 – Comparaison du résultat obtenu avec notre fonction sur les signaux contenus les fichiers "M1\_0.wav" et "M2\_0.wav" avec celui de `fftconvolve` de `scipy.signal` sur ces mêmes signaux.

Grâce à ces graphiques, nous pouvons observer que notre fonction est bien correcte, mais, nous pouvons également nous rendre compte de la similarité des deux signaux (axe des ordonnées) à chaque décalage dans le temps (en échantillons) sur l'axe des abscisses. Nous voyons donc un pic à l'endroit où la similarité entre les deux signaux est la plus élevée.

## 2.5 Localisation

Pour rappel, nous sommes dans la situation suivante, nous avons 3 micros, ( $M_1, M_2$  et  $M_3$ ) chacun placés à une certaine distance d'un point d'origine ( $O$ ). Ensuite, un bruit est produit à une position source ( $S$ ) qui n'est pas fixe. Le but de cette section est de retrouver cette position et plus particulièrement l'angle  $\theta$ .



### 2.5.1 TDOA

L'opération de TDOA consiste à récupérer un signal depuis différentes sources et à appliquer la cross-correlation pour pouvoir mesurer le délai auquel a été capturé ce signal.

```

1 def TDOA(xcorr, fs=44.1 * 1000):
2     # Get the sample index with the highest value
3     (i_max, sample_max) = (-float("inf"), -float("inf"))
4     for i, sample in enumerate(xcorr):
5         if sample_max < sample:
6             i_max, sample_max = i, sample
7     # Get the middle index
8     m = len(xcorr) // 2
9     return (i_max - m) / fs

```

Une façon de le calculer est de récupérer l'indice de l'échantillon avec la plus grande valeur et calculer la différence entre ce dernier et l'échantillon central. Pour ensuite convertir cette différence en seconde, en divisant par la fréquence d'échantillonnage comme expliqué dans la section 2.2.

### 2.5.2 Système d'équations de localisation

On va utiliser TDOA, afin de trouver quel micro reçoit le signal en premier (et est donc le plus proche). Avec deux *time-shifts* (que nous trouvons en utilisant TDOA), il est possible de trianguler la position de la source du bruit.

## Trouver la source

Nous savons que le *time-shift* entre deux microphones multiplié par la vitesse du son dans l'air est la projection de  $\overline{OS}$  sur  $\overline{M_i M_j}$ , c'est à dire :

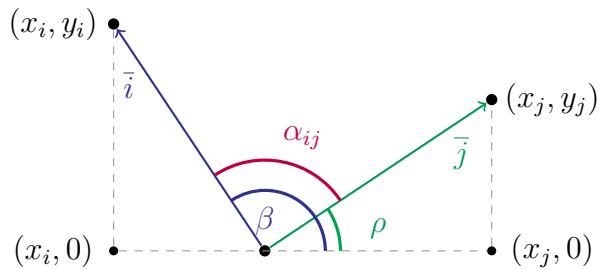
$$v \times D_{i,j} = \overline{M_i M_j} \times \overline{OS}$$

Avec  $D_{i,j}$  qui est le *time-shift* entre deux microphones  $i$  et  $j$ .

Et la projection d'un vecteur  $\bar{i}$  sur un vecteur  $\bar{j}$  est donnée par la relation suivante :

$$\bar{i} \cdot \bar{j} = ||\bar{i}|| \times ||\bar{j}|| \times \cos \alpha_{ij}$$

Avec  $||\bar{v}||$ , la norme du vecteur  $\bar{v}$ , et  $\alpha_{ij}$ , l'angle entre  $\bar{i}$  et  $\bar{j}$ , que nous pouvons retrouver en utilisant les relations fondamentales de la trigonométrie. Afin de mieux comprendre, prenons un exemple de situation possible :



Nous pouvons donc déduire que  $\alpha_{ij} = \beta - \rho$ . Avec  $\beta = \arctan\left(\frac{y_j}{x_j}\right)$  et  $\rho = \arctan\left(\frac{y_i}{x_i}\right)$ . (Nous rappellerons aussi comment trouver ces angles dans la sous-section suivante)

Il est maintenant possible de calculer  $\bar{i} \cdot \bar{j}$  :

$$\bar{i} \cdot \bar{j} = \sqrt{x_i^2 + y_i^2} \times \sqrt{x_j^2 + y_j^2} \times \cos \left( \arctan\left(\frac{y_i}{x_i}\right) - \arctan\left(\frac{y_j}{x_j}\right) \right)$$

Avec tout ceci, il est possible de trouver la position de source du bruit  $S$ , en résolvant l'équation suivante, qui nécessitera 2 TDOAs, étant donné qu'on a deux valeurs inconnues à trouver qui sont  $x_s$  et  $y_s$  :

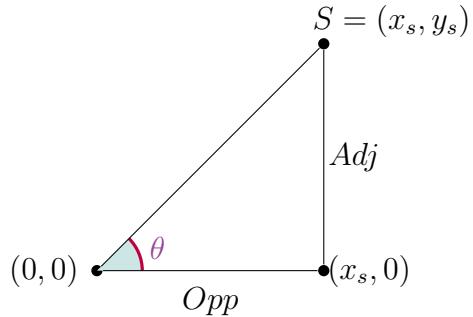
$$\begin{aligned} v \times D_{1,2} &= \overline{M_1 M_2} \times \overline{OS} \\ v \times D_{1,3} &= \overline{M_2 M_3} \times \overline{OS} \end{aligned}$$

La fonction python `localize_sound` qui nous a été fournie résout cette équation et nous retourne en sortie le point  $S$ , en prenant en entrée  $D_{1,2}$  ainsi que  $D_{1,3}$

## Trouver l'angle de la source

Maintenant que nous avons trouvé le point  $S$ , il est maintenant possible de trouver l'angle  $\theta$  en utilisant les relations fondamentales de la trigonométrie. En effet, nous recherchons la

valeur de l'angle  $\theta$  en sachant que le point  $S$  a comme coordonnée  $(x_s, y_s)$ . Nous pouvons donc généraliser la situation comme ceci :



Et en déduire que :

$$\tan \theta_r = \frac{Opp}{Adj}$$

Avec  $\theta_r$  qui est la valeur en radian de l'angle  $\theta$ . Et pour trouver ce dernier, il suffit de résoudre l'équation

- avec  $O$  la distance entre l'origine (le point  $(0, 0)$ ) et le point  $(x_s, 0)$ ,
- et  $A$  la distance entre le point  $S$  et le point  $(x_s, 0)$ .

$$\begin{aligned} \tan \theta_r &= \frac{\sqrt{(x_s - x_s)^2 + (0 - y_s)^2}}{\sqrt{(0 - x_s)^2 + (0 - 0)^2}} \\ &= \frac{y_s}{x_s} \\ \theta_r &= \arctan \theta \end{aligned}$$

(Notons que c'est aussi de cette façon que nous avons retrouvé les angles  $\beta$  et  $\rho$  dans la section précédente.) Et pour avoir  $\theta$ , l'angle en degrés, il suffit de faire :

$$\theta = \theta_r \cdot \frac{180}{\pi}$$

Nous appliquons donc ce que nous venons d'expliquer avec la fonction suivante :

```

1 def source_angle(coordinates):
2     xs, ys = coordinates
3     out = np.arctan2(ys, xs) * 180/np.pi
4     if out < 0:
5         out += 360
6     return out

```

## 2.6 Précision et vitesse du système

Dans cette section, nous allons juger la précision et la vitesse des fonctions décrites ci-dessus sur le jeu de données expliqué dans la section 1.

## 2.6.1 Précision de notre programme

Commençons par parler de la précision obtenue avec notre code. Pour cela, nous utiliserons la fonction `accuracy(pred_angle, gt_angle, threshold)` suivante :

```
1 def accuracy(pred_angle, gt_angle, threshold):  
2     return abs(pred_angle - gt_angle) < threshold
```

Cette fonction va vérifier si l'angle prédit (`pred_angle`) est assez proche du vrai angle (`gt_angle`) en regardant si leur différence est plus petite que la distance maximum acceptable (appelée le `threshold`).

Nous pouvons donc maintenant vérifier nos résultats obtenus sur l'ensemble du jeu de données de test en utilisant un `threshold` de 5.

Dans les sections suivantes, nous allons utiliser la notation suivante :

```
1 Real angle: (x) | Computed angle: (y)  
2 B
```

Elle signifie la comparaison d'un vrai angle  $x$  et de la prédiction calculée  $y$ . Avec  $B$ , le retour de la fonction `accuracy(y, x, 5)`.

### Sans preprocessing

Tout d'abord, calculons la précision de notre programme **sans appliquer** de `preprocessing` :

```
1 Real angle: (0) | Computed angle: (0.0)  
2     True  
3 Real angle: (30) | Computed angle: (29.970421179794062)  
4     True  
5 Real angle: (60) | Computed angle: (59.97040343760042)  
6     True  
7 Real angle: (90) | Computed angle: (84.17557978606705)  
8     False  
9 Real angle: (120) | Computed angle: (120.02959656239959)  
10    True  
11 Real angle: (150) | Computed angle: (150.02957882020596)  
12    True  
13 Real angle: (180) | Computed angle: (180.0)  
14    True  
15 Real angle: (210) | Computed angle: (209.97042117979407)  
16    True  
17 Real angle: (240) | Computed angle: (239.9704034376004)  
18    True  
19 Real angle: (270) | Computed angle: (270.0)  
20    True  
21 Real angle: (300) | Computed angle: (300.0295965623996)  
22    True  
23 Real angle: (330) | Computed angle: (324.214893078145)  
24    False
```

Ce qui donne une précision moyenne de 83.33 %.

### Avec preprocessing

Ensuite, recalculons la précision de notre programme, mais cette fois en appliquant le *preprocessing* comme vu dans la section 2.3 :

```
1 Real angle: (0) | Computed angle: (5.210792772829357e-15)
2   True
3 Real angle: (30) | Computed angle: (29.970421179794055)
4   True
5 Real angle: (60) | Computed angle: (70.8722507744857)
6   False
7 Real angle: (90) | Computed angle: (90.00000000000001)
8   True
9 Real angle: (120) | Computed angle: (109.12774922551431)
10  False
11 Real angle: (150) | Computed angle: (150.02957882020596)
12  True
13 Real angle: (180) | Computed angle: (180.0)
14  True
15 Real angle: (210) | Computed angle: (209.97042117979404)
16  True
17 Real angle: (240) | Computed angle: (250.87225077448568)
18  False
19 Real angle: (270) | Computed angle: (270.0)
20  True
21 Real angle: (300) | Computed angle: (289.1277492255143)
22  False
23 Real angle: (330) | Computed angle: (330.0295788202059)
24  True
```

On obtient une précision moyenne de 66.66 %. Nous pouvons observer qu'à cause du *preprocessing* notre programme est un peu moins précis, mais ce comportement était attendu. En effet, l'étape du *downsampling*, a affecté les données d'entrée de notre programme et peut avoir retiré de l'information utile.

### 2.6.2 Vitesse de notre programme

Parlons maintenant de la vitesse des différentes fonctions implémentées lors de ce projet. On parlera des fonctions `normalize`, `fftxcorr`, `fftconvolve`, `TDOA`, `localize_sound` et `source_angle`. Il est important de noter que les résultats ci-dessous ont été obtenus après une seule exécution de l'application. Néanmoins, beaucoup de choses peuvent influencer le calcul du temps, surtout en nanosecondes, ce qui peut le rendre imprécis.

### Sans *downsampling*

```
1 Without downsampling:  
2 normalize in 4293458 ns  
3 fftxcorr in 43964299 ns  
4 fftconvolve in 3832665 ns  
5 TDOA in 8029589 ns  
6 localize_sound in 224605 ns  
7 source_angle in 23434 ns
```

### Avec *downsampling*

```
1 With downsampling:  
2 normalize in 2927313 ns  
3 downsampling in 1100992 ns  
4 fftxcorr in 28791909 ns  
5 fftconvolve in 2098198 ns  
6 TDOA in 5108369 ns  
7 localize_sound in 338764 ns  
8 source_angle in 11001 ns
```

### Observations

Nous remarquons plusieurs choses, premièrement dans les deux cas, notre fonction `fftxcorr` est moins rapide que la fonction `fftconvolve` ceci est normal pour plusieurs raisons, par exemple Scipy peut faire appel à des fonctions extérieures en C plus rapide que le Python ou a tout simplement des techniques pour optimiser ce calcul que nous ne connaissons pas.

On remarque aussi une amélioration notable dans la vitesse d'exécution des fonctions `TDOA`, `fftxcorr` ainsi que `fftconvolve` après l'application du *downsampling*, étant donné qu'il y a moins de données à traiter après l'étape de la **décimation**.

### 2.6.3 Conclusion des observations

On peut conclure que l'étape de *preprocessing* est très importante et va beaucoup influencer le comportement de notre programme.

- Sans le *preprocessing*, nous obtiendrons des résultats plus précis, mais le temps d'exécution sera plus long.
- Avec le *preprocessing*, le programme tournera plus rapidement, mais au détriment de la qualité de nos prédictions.

Étant donné que nous devrons par la suite traiter des données en temps réel, il sera nécessaire d'appliquer le *preprocessing*, surtout que ce dernier sera important dans des conditions réelles auquelles du bruit peut se rajouter à l'information utile.

# Chapitre 3

## Localisation en temps réel

Dans cette section, nous allons utiliser le matériel détaillé dans la section "Introduction" afin de mettre en pratique les étapes précédemment expliquées.

### 3.1 Application pratique possible et son fonctionnement

Grâce à sa flexibilité et aux nombreuses possibilités de développement, il est possible d'imaginer de nombreux systèmes à l'aide d'un Raspberry Pi. Nous avons beaucoup échangé et l'idée qui revenait le plus fréquemment était dans le domaine animalier. Si cet outil était utilisé comme un capteur ?

Le but de ces capteurs serait de délimiter la zone de vie de certaines espèces que nous voulons protéger, par exemple, de toute nuisance causée par l'Homme. Cela permettrait également d'en apprendre plus sur le mode de vie de ces animaux. Restent-ils toujours dans la même zone ? Si ce sont des prédateurs, quelles zones semblent être favorisées pour leur chasse ?

Ces capteurs pourraient être placés dans une zone calculée au préalable par de nombreuses observations, ou encore, sur les animaux soignés et ensuite, relâchés (afin de conserver les espèces).

Un problème s'est posé dans notre réflexion au niveau de l'écologie mais nous en sommes venus à la conclusion, que les capteurs étant placés dans une zone calculée et à des endroits précis, il serait plutôt facile de les récupérer. En ce qui concerne les dispositifs placés sur les animaux, on pourrait imaginer un mécanisme se déclenchant lorsque la batterie devient faible afin que le capteur se décroche de l'animal et émette sa position pour ne pas perturber d'avantage ce dernier.

### 3.2 Feedback visuel et impression de temps réel du système

Il y a 12 LEDs dans le système. Les conditions sont les suivantes :

1. lorsqu'une LED particulière affiche une couleur brillante, la source se trouve dans la même direction,

2. et lorsque deux (ou trois) LED adjacentes sont allumées, la source est située entre ces deux (ou trois) LED.

Nous allons maintenant comparer nos résultats avec différentes tailles de buffer et fréquences d'échantillonnage.

1. **Cas de base (buffer de 0.18s et fréquence d'échantillonnage de 16kHz) :**
  - Nous avons pu observer que nos résultats semblent corrects et rapides (de l'ordre de la seconde), les LEDs réagissaient de façon adéquate aux bruits (signaux) reçus et s'allument rapidement.
2. **Buffer de 0.46s et fréquence d'échantillonnage de 16kHz**
  - On remarque que la position des leds est encore correcte mais le système s'exécute plus lentement que précédemment.
3. **Buffer de 1.11s et fréquence d'échantillonnage de 16kHz**
  - On remarque que la position des leds est encore correcte mais le système s'exécute encore plus lentement que précédemment. On remarque même que si nous n'effectuons pas un bruit assez long, les LEDs ne réagissent pas.
4. **Fréquence d'échantillonnage de 24kHz et buffer de 0.18s**
  - Dans ce cas, nous aurons une décimation de deux au lieu de trois. On remarque clairement que la position des LEDs est imprécise.
5. **Cas supplémentaire : pas de décimation et buffer de 0.18s**
  - Le programme plante complètement.

Il a été observé que la latence augmente à mesure que la taille du buffer augmente. Ce qui est logique car plus le buffer est grand, plus une donnée doit attendre longtemps avant d'être traitée (le temps que le buffer se remplisse). Aussi plus le nombre de données augmente plus le temps de calcul de nos fonctions augmente aussi comme vu dans la section 2.6.2.

Nous avons également remarqué que plus le facteur de décimation diminue, plus les résultats sont imprécis.

# **Chapitre 4**

## **Conclusion**

En utilisant pas à pas différentes méthodes de traitement du signal et en faisant des simulations sur un jeu de données, nous avons été amenés à appliquer tout ce travail sur un système temps-réel afin de détecter de quelle direction provenait un signal.

Ce projet nous a donné l'opportunité de mettre en pratique ces différents concepts vus en cours de façon ludique. Nous avons également pu observer nos résultats et les comparer au comportement attendu de l'application.

Toutes ces étapes combinées nous ont permis d'en apprendre davantage sur la façon de gérer le traitement des signaux.