

# Drum'Assist



Projet Interdisciplinaire 2017

**Shariatian Dario**  
**Terminale 4**

## Sommaire

### **I) Introduction au projet**

<u>1) Rappels sur les sons</u>	4
<u>2) Contexte et Besoin</u>	7
<u>3) Travail préliminaire : mesures de référence au Minimal Studio (rigueur et qualité des mesures)</u>	8
<u>4) Principe de fonctionnement général du système</u>	9

### **II) Mise en Place et Délimitations du Projet**

<u>1) Décomposition fonctionnelle</u>	10
<u>2) Cahier des Charges</u>	11

### **III) Études de Performances, Détermination des Écarts**

<u>1) Relever le flux audio entrant</u>	13
<u>2) Étude du microphone</u>	15
<u>3) Analyser le flux audio</u>	
<u>3.1) Mise en place de la transformée de Fourier</u>	21
<u>3.2) Fonction de fenêtrage</u>	28
<u>3.3) Reconnaître une frappe</u>	30
<u>4) Communication avec le microcontrôleur</u>	36
<u>5) Présentation du programme final et de ses performances</u>	37

### **IV) Présentation du système final**

<u>Bibliographie</u>	41
<u>Annexes</u>	42

## **Remerciements**

Je tiens tout d'abord à remercier mes professeurs de Sciences de l'Ingénieur M. Rabeau et M. Pescheux, dont les aides et recommandations ont été capitales à l'avancement du projet, ainsi que Mme Berthier, professeur d'éducation sportive, dont le regard neuf sur le sujet a pu apporter d'excellents conseils.

Je tiens également à remercier mes camarades dans l'élaboration de ce projet : Aymeric Schneider, Jean-Charles Lévy et Clémence Lemercier.

Enfin, je remercie les professionnels du Minimal Studio (Alex Finkin et Benjamin Petit) qui ont eu la gentillesse de nous accorder de leur temps afin de nous renseigner sur les aspects sonores du projet et qui nous ont permis d'enregistrer des sons sans parasites servant de matière première aux recherches, ainsi que l'artiste Mamady Keita, avec qui nous avons échangé des mails éclairants.

## I) Introduction au Projet

### 1) Rappels sur le son

Le son et ses particularités constituent un thème central de ce projet. Ainsi avant de présenter ce dernier il est essentiel d'expliquer les caractéristiques de base du son.

Un son est une vibration mécanique d'un fluide (comme l'air) se propageant sous forme d'onde. En tant qu'humain, nous percevons ces vibrations à l'aide de nos tympans qui se mettent eux aussi à vibrer, ce qui sera ensuite traduit par notre cerveau en un son.

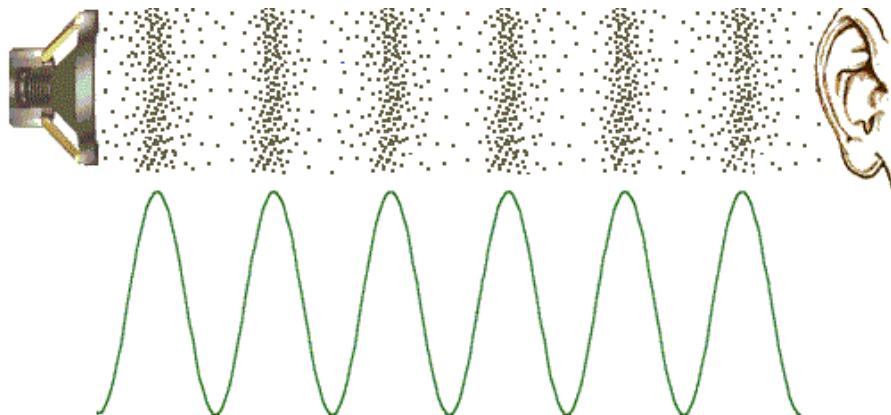
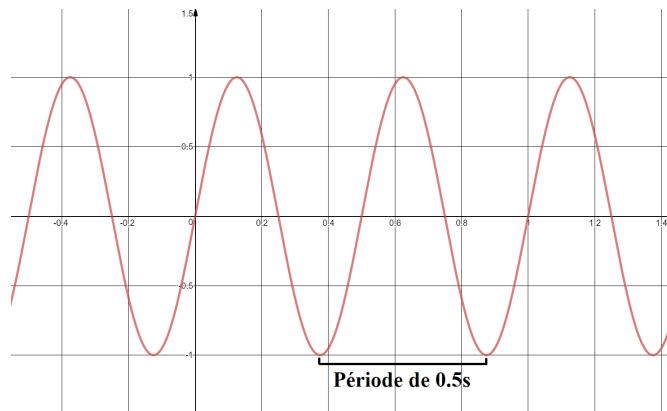


Schéma représentant la propagation d'onde sonore dans l'air. Nos tympans se mettront à vibrer au même rythme

Un son peut être périodique, c'est-à-dire qu'il peut se répéter identique à lui-même à intervalles de temps réguliers. On peut alors définir la période de ce son, i.e la durée de cet intervalle.



Graphe d'une onde sonore périodique de période 0.5s

De plus, on définit la fréquence d'un tel son (périodique) comme le nombre de périodes par seconde. Dans l'exemple précédent, puisqu'une période dure 0.5s, alors en une seconde on aura 2 périodes, soit une fréquence de 2Hz (on exprime la fréquence en Hertz (Hz)). Plus généralement, si on note  $f$  la fréquence et  $T$  la période, une relation extrêmement simple les lie, découlant de la définition :

$$f = \frac{1}{T}$$

Si la fréquence est faible (par exemple 100Hz) on entend un son grave, si elle est élevée (par exemple 4000Hz) on entend un son aigu.

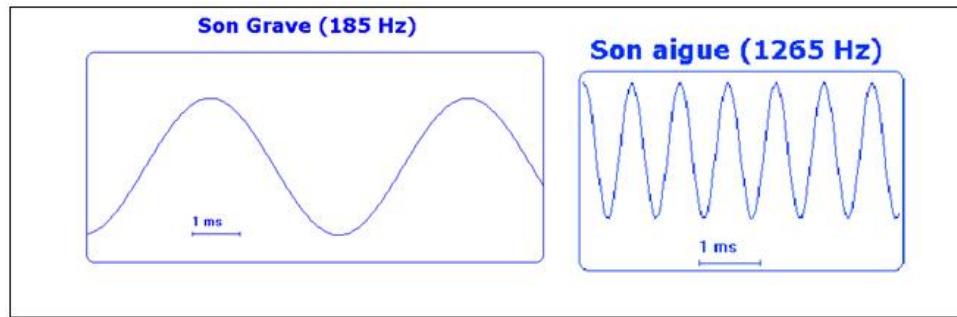


Schéma montrant la différence entre son grave et son aigu

Il s'avère que toute onde sonore, aussi complexe soit-elle, périodique ou non, peut être décomposée en une somme d'ondes périodiques. Ainsi il est possible de caractériser une onde sonore quelconque à l'aide d'un spectre fréquentiel : ce dernier est un outil qui représente les fréquences et amplitudes (l'amplitude est l'intensité, la force du son) des différentes ondes périodiques simples qui constituent cette onde sonore.

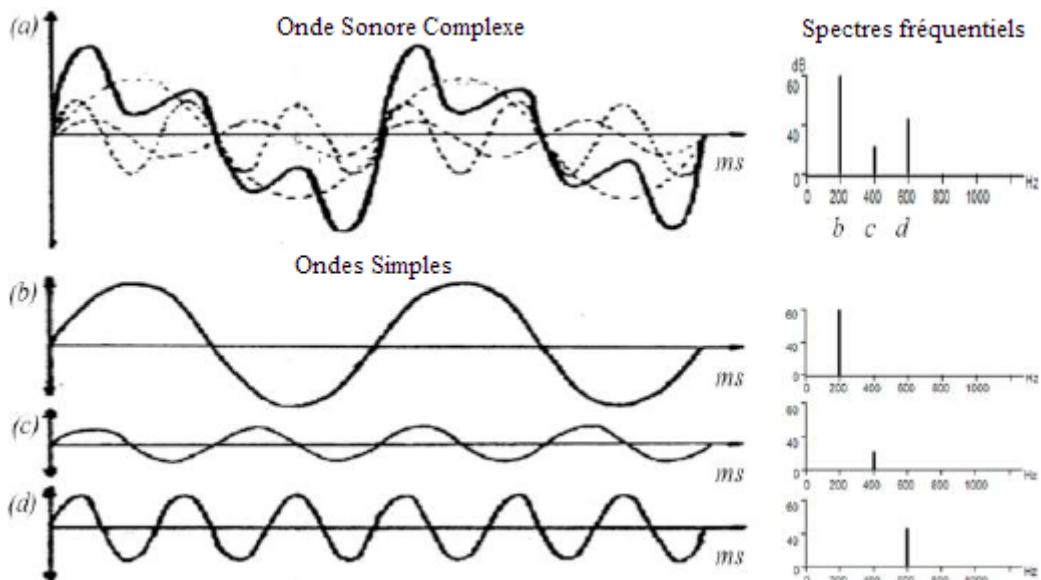
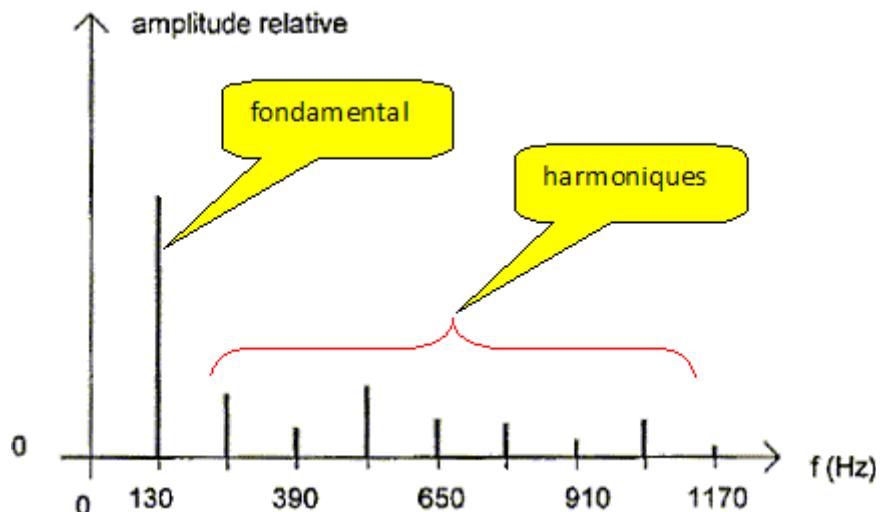
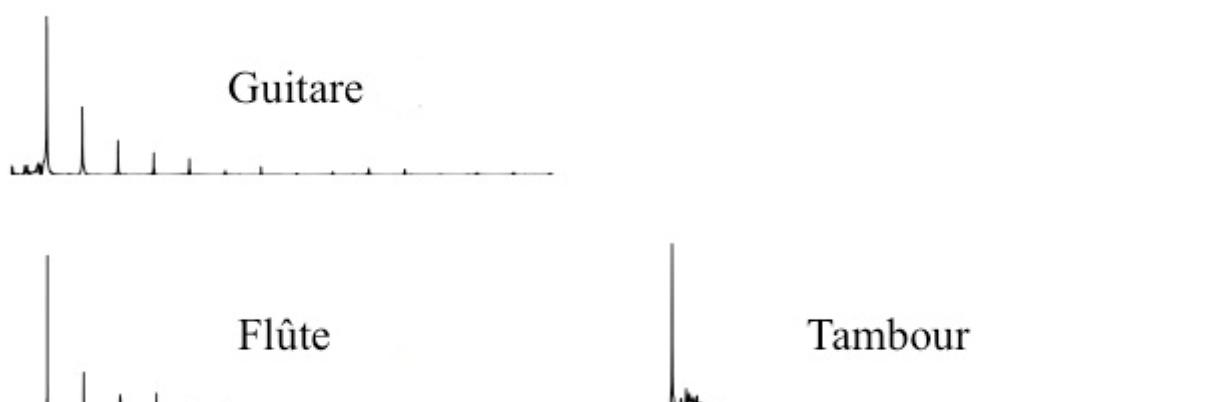


Schéma montrant la décomposition d'une onde sonore complexe en ondes simples, ainsi que leurs spectres fréquentiels respectifs

Enfin, cas particulier, lorsque l'on joue d'un instrument, les notes émises ont quelques caractéristiques supplémentaires intéressantes : si on étudie leur spectre fréquentiel, on pourra y trouver une fréquence dite fondamentale d'amplitude la plus élevée : c'est la fréquence qui caractérise la note jouée, qu'on appelle aussi hauteur. Par exemple si elle est de 440Hz il s'agira d'un LA3. On y trouvera aussi des fréquences dites harmoniques qui sont multiples de la fréquence fondamentale : ils forment le timbre de la note. En effet, le LA d'une guitare et le même LA d'un violon sonnent bien différemment : bien qu'ils aient la même hauteur, leurs timbres (i.e les fréquences harmoniques) sont différents.



Spectre fréquentiel d'une note avec sa fondamentale et ses harmoniques

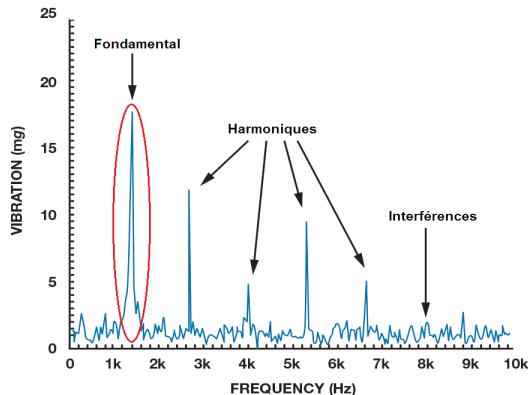


Spectres fréquentiels d'une même note jouée sur différents instruments

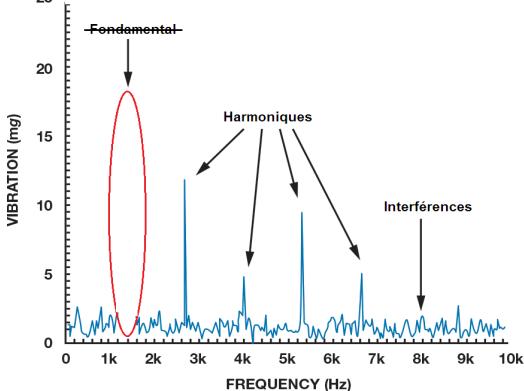
## 2) Contexte et Besoin

Dans le jeu d'une percussion quelconque, l'instrument peut émettre deux types de notes :

-Des notes normales issues de frappes normales : les spectres fréquentiels de ces notes n'ont pas de caractéristiques particulières, on y distingue une fondamentale accompagnée d'harmoniques :



-Des « harmoniques », qui sont des notes dont la fréquence fondamentale a été étouffée et dont on ne garde que les fréquences harmoniques. Ces dernières sont les notes les plus belles et les plus appréciées :



Bien que cela puisse paraître trivial, obtenir une note harmonique sur un certain nombre de percussions se révèle très compliqué, si bien que nombreux de musiciens ne savent pas en jouer une correctement même après des années d'expérience. En effet, produire une telle note requiert un angle, une position et une vitesse de frappe bien précis, paramètres qui sont dans leur ensemble très difficiles à communiquer pour un maître vers son élève comme a pu nous l'expliquer Jean-Claude Nardon (maître du djembé entraîné par les plus grands tel Mamady Keita).

Il s'agit alors de permettre de manière pédagogique à un néophyte (voire même quelqu'un de relativement expérimenté) d'obtenir des « harmoniques » parfaites.

C'est à ce besoin que répond ce système. Il sera une aide parallèle à un apprentissage classique avec un professeur (qui comme nous l'avons expliqué se révèle être insuffisant). Il s'agira de montrer indéfiniment le mouvement parfait à effectuer sur la percussion avec un bras robotisé, mais également d'indiquer des informations de correction en écoutant les frappes de l'utilisateur. L'enjeu pédagogique inhérent à la réalisation du projet nous permet d'en définir quelques caractéristiques supplémentaires, en particulier automatiser et faciliter au maximum la commande du système par l'utilisateur (avec la création d'un programme informatique sur ordinateur).

### 3) Travail préliminaire : mesures de référence au Minimal Studio (rigueur et qualité des mesures)

Pour débuter notre projet, nous nous sommes rendus au Minimal Studio, studio d'enregistrement dans le 14ème arrondissement de Paris, accompagnés d'un professionnel des percussions pour faire nos mesures. Au studio, des ingénieurs du son ont pu répondre à nos questions concernant la viabilité et la mise en place du projet, puis nous avons procédé à des relevés audios et vidéos de frappes sur une percussion. En effet, ce genre de studio offre une salle insonorisée pour effectuer les relevés, ce qui élimine la majorité des parasites, ainsi que du matériel d'enregistrement de très haute qualité.



L'image ci-dessus est une capture d'écran d'une vidéo que nous avons prise du professionnel des percussions jouant sur une caisse claire (c'est la percussion que nous utilisons pour notre système). Cette vidéo a été réalisée à l'aide d'une caméra haute fréquence afin de pouvoir mieux analyser la vitesse de la baguette lors de la frappe. On peut également voir sur cette image le **microphone** utilisé (entouré en **rouge**) ainsi qu'une autre **caméra** (entouré en **jaune**) filmant de haut la caisse claire (pour étudier la position de la baguette sur la caisse claire lors de la frappe et ses répercussions sur le son émis).

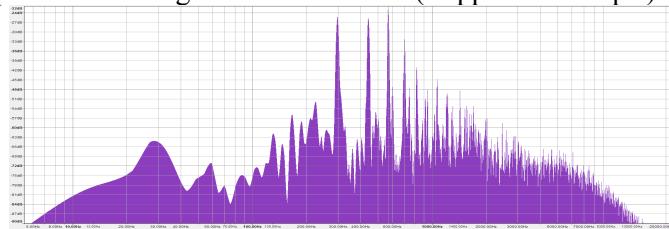


Mise en place du matériel avant de procéder aux mesures

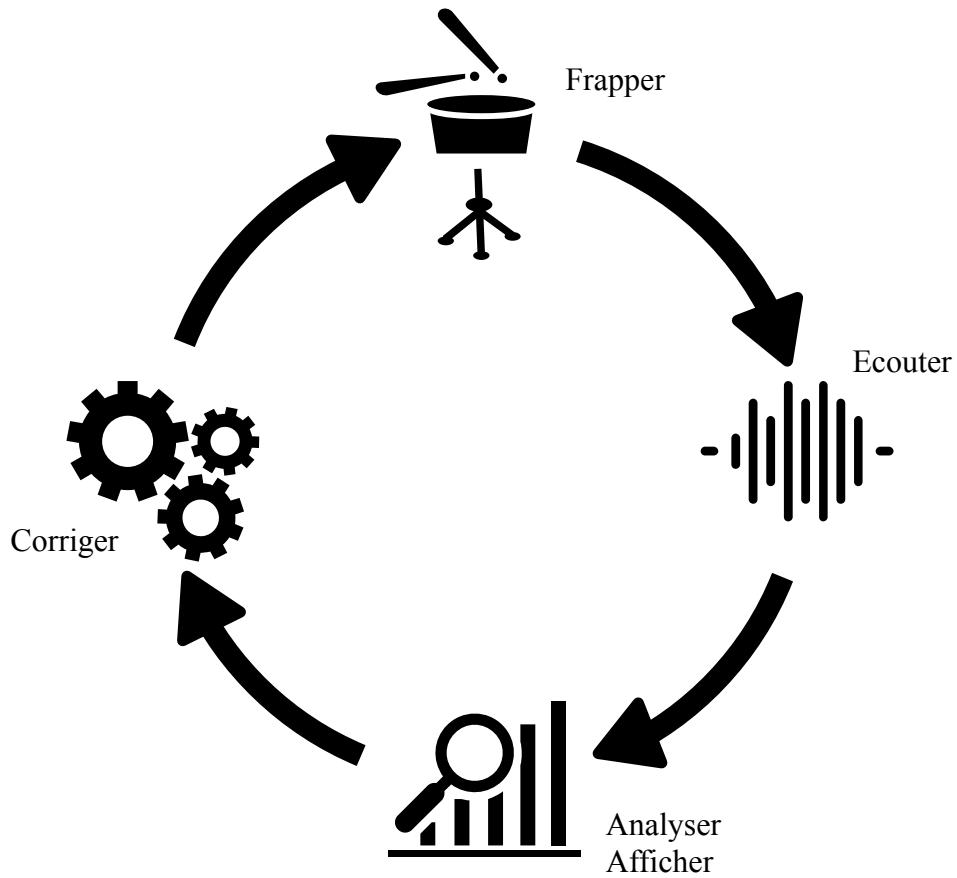


Utilisation d'un mètre sur les vidéos pour pouvoir étalonner les mesures

Exemple d'un spectre fréquentiel obtenu grâce à ces mesures (frappe harmonique) :



#### 4) Principe de fonctionnement général du système



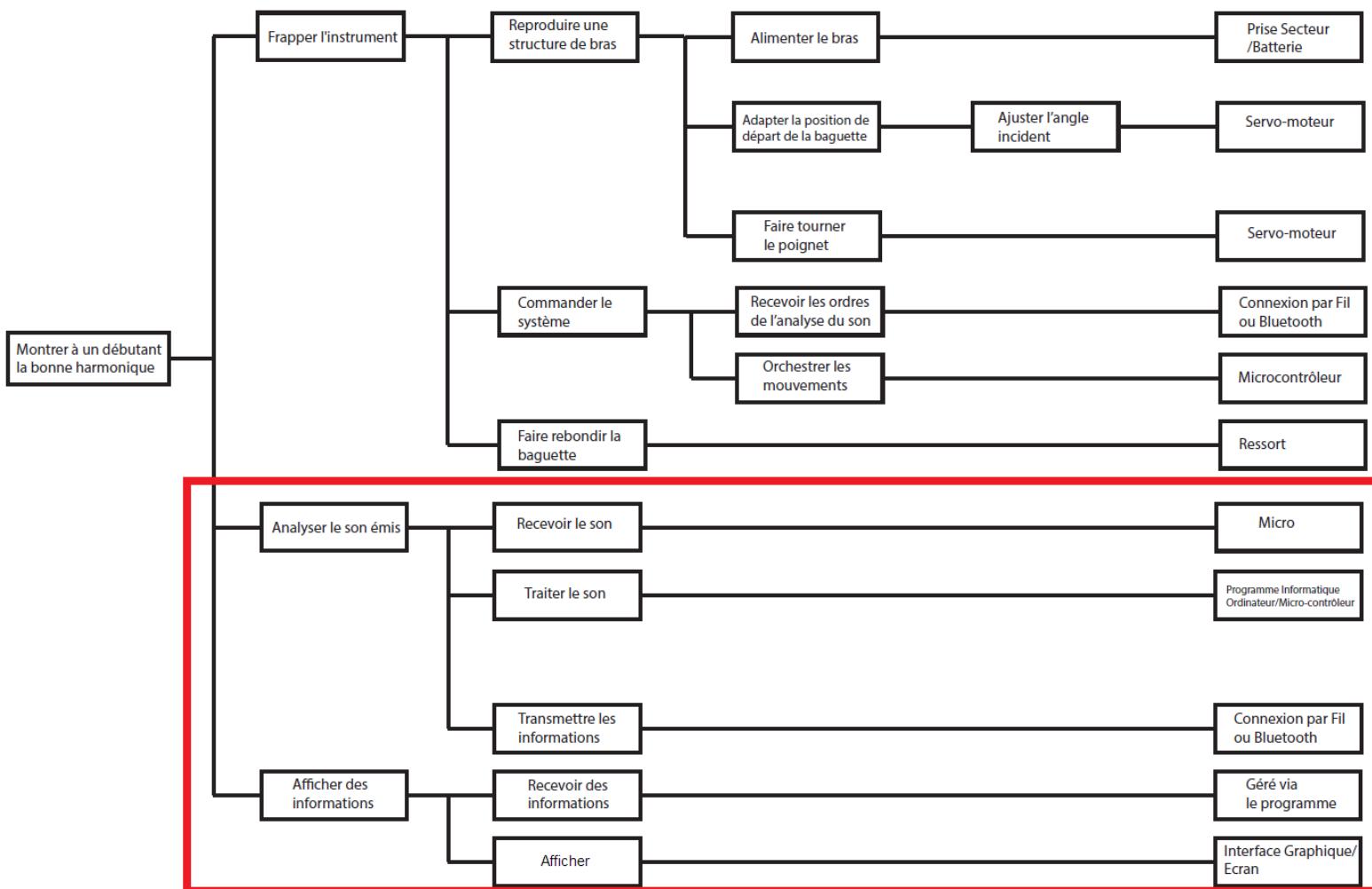
Le cycle décrit ci-dessus est simple. Il s'agit de :

- **Frapper** la percussion. Cela peut se faire soit par le débutant (s'il veut savoir si sa frappe était harmonique ou pas), soit par le bras robotisé (si on veut que le système reproduise une frappe parfaite à montrer au débutant) ;
- **Écouter** et enregistrer le son émis à l'aide d'un microphone ;
- **Analyser** le son reçu grâce au microphone afin d'en déduire les caractéristiques (on pourra en faire le spectre fréquentiel puis le comparer à celui d'une frappe harmonique par exemple), que l'on pourra **Afficher**, ce qui permettra à l'utilisateur de savoir si la frappe était harmonique ou non ;
- **Corriger** le mouvement du bras robotisé, dans le cas où ce dernier effectue la frappe, pour se rapprocher le plus possible de la frappe harmonique.

## II) Mise en Place et Délimitations du Projet

### 1) Décomposition fonctionnelle

Nous avons réalisé un diagramme FAST. Ce dernier développe les fonctions de service du système (fonctions demandées pour répondre au besoin) en fonctions techniques (fonctions permettant de réaliser l'action) pour finalement présenter les solutions techniques choisies. Le FAST nous permet d'avoir une bonne vision globale du système et surtout des tâches qu'il faut réaliser. En outre, on peut y discerner clairement la partie qui me concerne dans la réalisation du projet.



La partie qui m'incombe est encadrée en rouge.

## 2) Cahier des charges

Détermination des performances attendues/théoriques.

Fonction	Recevoir le son
Objectif	Relever et stocker le flux audio entrant dans l'ordinateur en vue de son analyse
Description	Utilisation d'un microphone pour transformer le son en données numériques et les transmettre à l'ordinateur. Stockage du flux audio entrant dans la mémoire de l'ordinateur.
Contraintes	<p>Le microphone doit fournir un ensemble de données suffisamment proches du signal original : <b>les valeurs du microphone doivent être comprises entre 15 % et 85 % de l'amplitude maximal, et sa courbe de réponse ne doit pas varier de plus de 4 dB sur la plage de fréquence importante (entre 250 et 1500 Hz)</b></p> <p>La mise en place du stockage du flux audio ne doit pas dégrader les données et permettre une précision suffisamment bonne pour l'analyse qui suit. <b>La résolution des échantillons doit être suffisante et la fréquence d'échantillonnage doit permettre de capturer les fréquences importantes les plus hautes émises lors d'une frappe de percussion (1500 Hz)</b></p>
Niveau de priorité	Priorité haute

Fonction	Traiter le son
Objectif	Déterminer les caractéristiques du flux audio entrant à l'aide d'une analyse (algorithmes...)
Description	Lancer l'analyse sur l'ensemble de données correspondant au flux audio. Optimisation des données, mise en place du spectre fréquentiel, comparaison avec la base de données pour conclure quant au son relevé.
Contraintes	<p>La durée de l'analyse doit absolument être inférieure à la durée du relevé audio (autrement on ne pourrait pas traiter tous les relevés, ce qui serait problématique) quel que soit l'ordinateur sur lequel est lancé le programme. Ainsi, il faut que l'analyse soit très rapide. Sur mon ordinateur, pour avoir une marge suffisante, on fixe : <b>un temps de traitement inférieur à 10% de la durée du relevé audio, une utilisation processeur moyenne inférieure à 5%.</b></p> <p><b>L'analyse doit être précise. On s'impose que le spectre fréquentiel soit précis au hertz près.</b></p> <p>L'analyse devrait pouvoir reconnaître une frappe et <b>donner un résultat cohérent au moins 1 fois sur 2.</b></p>
Niveau de priorité	Priorité haute

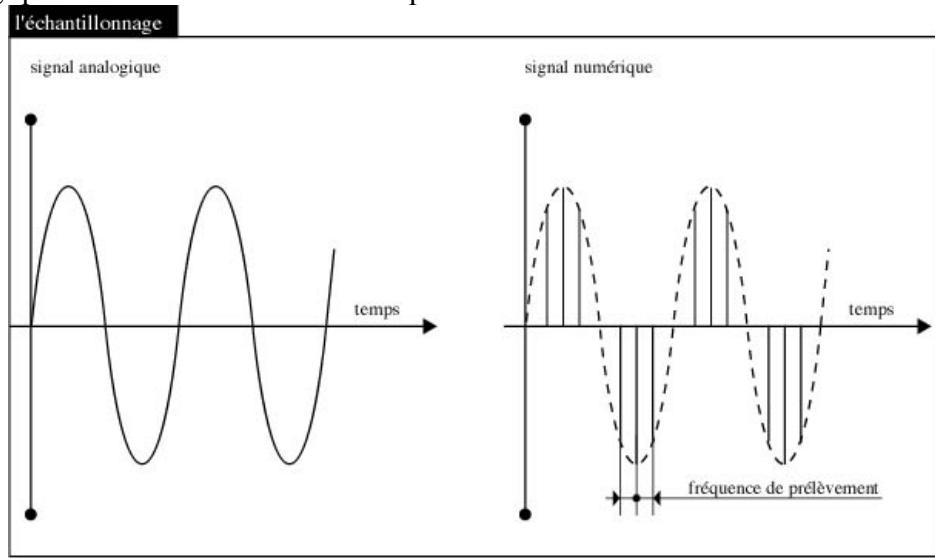
Fonction	Afficher les informations
Objectif	Montrer les caractéristiques audios du son relevé
Description	Le système relève en continu le flux audio entrant. Il en fait en temps réel l'analyse, qu'il affiche. Ainsi, il est possible de visualiser la qualité d'une frappe sur la percussion. Cela permet également de vérifier que tout fonctionne bien dans le traitement du son.
Contraintes	L'affichage doit être clair. Il doit être intégré à l'interface utilisateur.
Niveau de priorité	Priorité Normale

Fonction	Transmettre les informations
Objectif	Transmettre les résultats de l'analyse audio au microcontrôleur après une frappe.
Description	Lorsqu'une frappe est détectée, le programme informatique doit transmettre les résultats de l'analyse sonore au microcontrôleur pour une éventuelle correction du mouvement du bras robotisé.
Contraintes	<p>La transmission se fait par fil.</p> <p>Le programme doit reconnaître automatiquement la présence d'un microcontrôleur sur l'ordinateur.</p> <p style="color: red;"><b>La transmission ne doit quasiment jamais échouer (seuil de 95%)</b></p>
Niveau de priorité	Priorité Normale

### III) Études de Performances, Détermination des Écarts

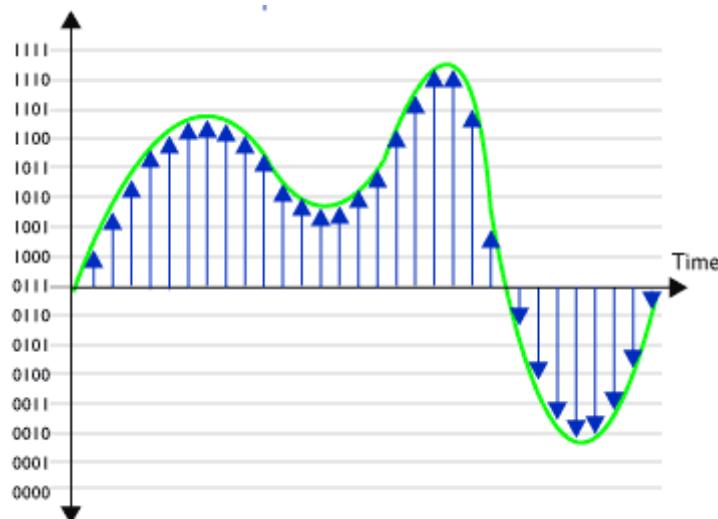
#### 1) Relever le flux audio entrant

Pour que le son puisse être analysé par un ordinateur, il doit être converti en données numériques. Cela se fait en 2 étapes : tout d'abord, un microphone transforme, à l'aide d'une membrane, l'onde sonore en une onde analogique (les variations de pressions sont converties en variation de tension). A partir de là, un CAN (Convertisseur Analogique-Numérique) transforme ce signal analogique continu en valeurs numériques discrètes.



L'échantillonage s'effectue sur la dimension du temps

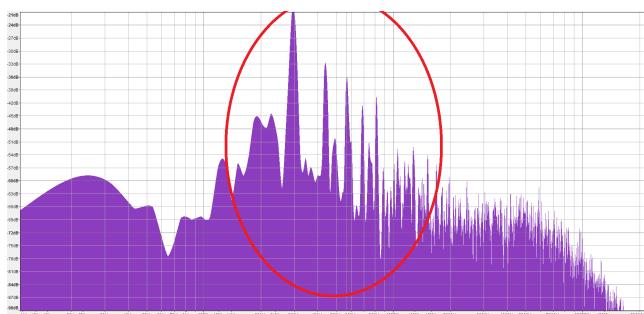
La période d'échantillonnage, i.e la durée entre deux échantillons, est constante. On définit la **fréquence échantillonnage** comme l'inverse de cette durée. Elle est de 44.1kHz pour un CD par exemple. Un autre facteur entre en compte dans la conversion, il s'agit de la **Résolution** des échantillons : en effet, chaque échantillon est codé sur un nombre fini de bits. Plus le nombre de bits sur lequel il est codé est grand, plus de valeurs il pourra prendre et donc plus le relevé sera précis. Exemple ci-dessous avec une résolution de 4 bits ( $2^4 = 16$  valeurs possibles) :



La réalisation du projet requiert l'analyse de données audio correspondant à une frappe de percussion pour en déduire ses caractéristiques et ainsi déterminer les éventuelles corrections à appliquer.

Il a fallu réaliser des études afin de déterminer les impacts de la qualité du microphone sur les relevés en temps réels, ainsi que pour le choix du microphone. Les protocoles de tests qui suivent ont pour but de faire une rédaction illustrée, schématisée des successions de tâches à réaliser en vue de caractériser les performances du microphone (préparer, faire, puis conclure par rapport au cahier des charges).

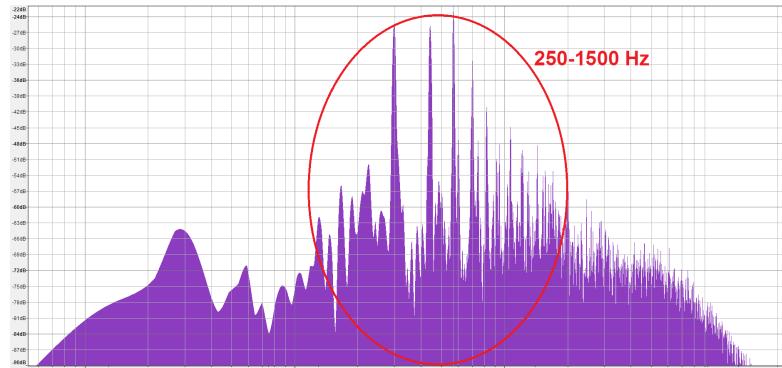
Suite aux relevés audio du Minimal Studio, il a été possible, grâce à un logiciel de traitement du son (Audacity), de déterminer les spécificités sonores d'une frappe en fonction de sa position sur la percussion (distance au centre, si elle touche l'arceau ou non...). Son spectre fréquentiel est obtenu grâce à une transformée de Fourier : c'est un procédé mathématique permettant de représenter en fréquences un signal sonore. Cela permettra de créer un bon algorithme.



Spectre fréquentiel d'une frappe excentrée



Spectre fréquentiel d'une frappe centrée



Ce spectre fréquentiel correspond à celui d'une frappe harmonique (obtenu avec Audacity).

Les fréquences à étudier pour différencier chaque type de frappe de percussion sont situées dans une plage s'étalant de 250 à 1500 Hz. On notera que les pics de fréquence caractéristiques peuvent être très fins, ne s'étalant parfois que sur quelques hertz (pour certains environ 4-5Hz).

## 2) Étude du microphone

L'importance du microphone dans la réussite du projet est évidente : en effet, pour analyser correctement le son émis par la frappe, il est nécessaire que les données sonores relevées et fournies par le microphone soient de suffisamment bonne qualité, et donc qu'elles ressemblent suffisamment au signal original. Nous avons pour notre projet choisi le microphone "**H2N Zoom**". Il va falloir procéder à des tests sur ce microphone pour confirmer ou non ses performances, sa robustesse et sa fiabilité ; nous le comparerons au microphone '**Windows**' fourni par défaut sur notre ordinateur.

Avant de procéder aux tests, il s'agit de comprendre ce qui caractérise la qualité d'un microphone :

- **Directivité** : La directivité d'un microphone est sa sensibilité aux sons selon la direction ou l'angle de la source sonore, ou plus simplement la manière dont le microphone «entend» le son en fonction des différentes directions.

- **Pression acoustique maximale admissible (SPL)** : mesurée en dB, c'est le niveau sonore maximum pouvant être supporté par la membrane avant saturation/dégradation.

- **Bandé Passante et Courbe de Réponse** : La bande passante correspond à l'aptitude à restituer les fréquences captées. Elle est schématisée par une courbe de réponse.

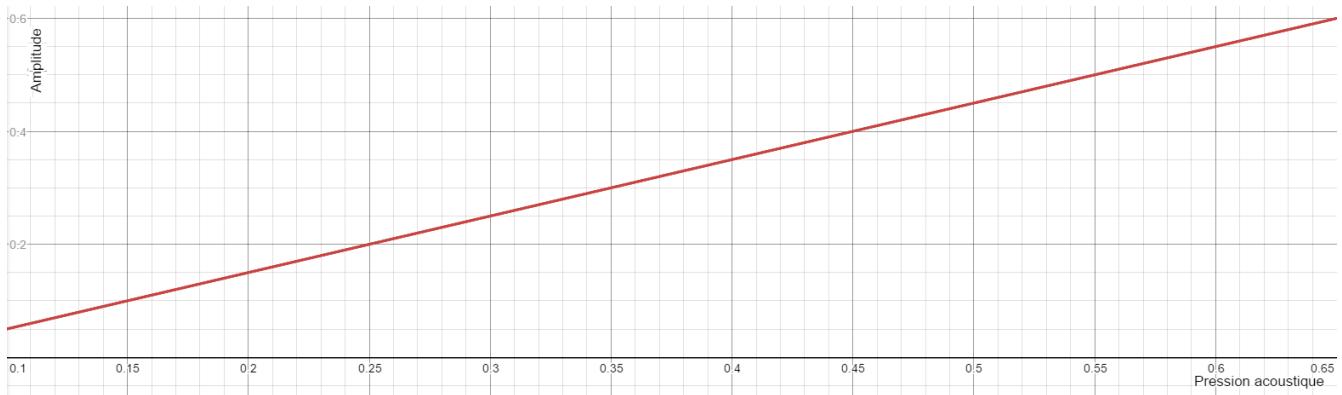
- **Sensibilité** : c'est la tension de sortie du microphone par rapport à la tension acoustique à laquelle il est soumis, mesurée en mV/Pa. Elle varie selon la technologie utilisée par le micro. Si on doit par exemple enregistrer une source sonore très faible, il faudra impérativement utiliser un micro ayant une sensibilité élevée, pour obtenir un meilleur rapport signal/bruit. Une bonne sensibilité assure une meilleure restitution de la bande passante.

Dans nos tests, nous nous intéresserons seulement à la sensibilité et à la bande passante du micro. En effet, la SPL nous importe peu : elle est au minimum de 115 dB sur n'importe quel microphone, et jamais un microphone ne sera soumis à de tels volumes sonores dans le cadre de ce projet. La directivité, quant à elle, est une donnée intéressante lorsqu'il s'agit d'éviter les larsens ou de s'isoler des bruits environnants (bien que tous captent de manière optimale les sons à l'avant, ils atténuent plus ou moins les sons provenant des autres directions), néanmoins nous n'effectuerons pas de test sur ce dernier paramètre car nous considérons que la frappe sera suffisamment proche et forte pour étouffer les bruits indésirables environnants, et nous dirigerons le microphone droit devant la source sonore.

Ainsi le protocole sera le suivant : tester la sensibilité du micro en le soumettant à des frappes répétées de percussion pour savoir si elle fera des relevés correctes au volume sonore d'une frappe de percussion, puis tester sa bande passante, c'est-à-dire sa capacité à restituer certaines fréquences, pour savoir si elle relèvera correctement les fréquences qui nous intéressent dans la frappe d'une percussion.

### Sensibilité

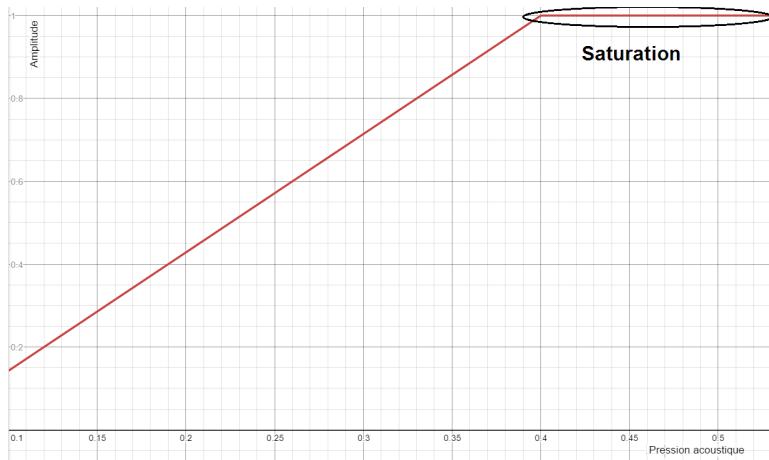
En réalité, la valeur de la sensibilité du microphone en tant que tel nous importe peu. Nous voulons seulement savoir si elle est adaptée aux mesures que nous voulons faire, c'est-à-dire pour une frappe de percussion. Ainsi nous nous intéressons seulement à la valeur normalisée du volume donnée par le micro (rapport de l'amplitude sonore relevée sur l'amplitude maximale que peut relever le micro, ce qui donne une valeur entre 0 et 1. Cette échelle est sensée être linéaire, et c'est ce que nous supposerons ; dans les faits, il se peut qu'elle ne le soit pas parfaitement pour certains volumes ou pour certaines fréquences, mais les effets sont, dans le cadre de nos études, négligeables). Par exemple on aurait :



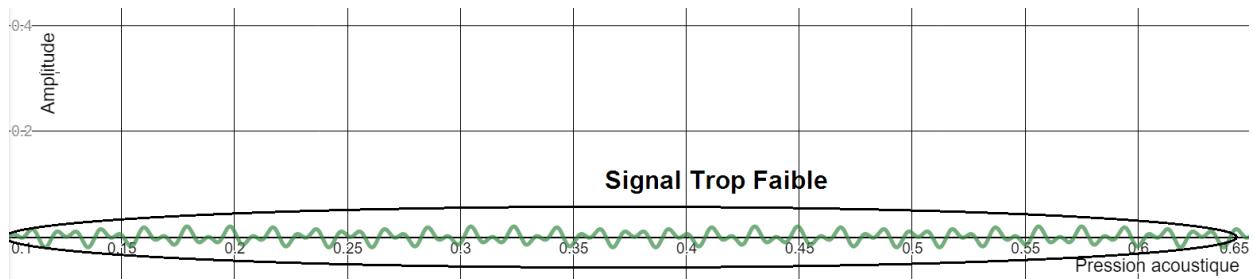
La pression acoustique, en Pa, aurait ici été multipliée par un facteur arbitraire. Il s'agit seulement de voir la relation linéaire liant la pression acoustique qu'enregistre le micro et la valeur d'amplitude qu'elle délivre.

Les expériences doivent montrer que le micro relève les données sonores de manière assez fiable du point de vue sonore et donc :

- Que le micro ne sature pas lors d'une frappe (Les systèmes d'enregistrement, de traitement et de restitution du son tel un micro ont tous un volume maximum — un espace limite — au-delà duquel le signal est perdu ou déformé : on parle alors de saturation). Dans un tel cas, le volume normalisé donné par le micro sera égal ou proche de 1 ;



- Que le micro soit suffisamment sensible pour relever avec une précision correcte le son issu de la frappe ;



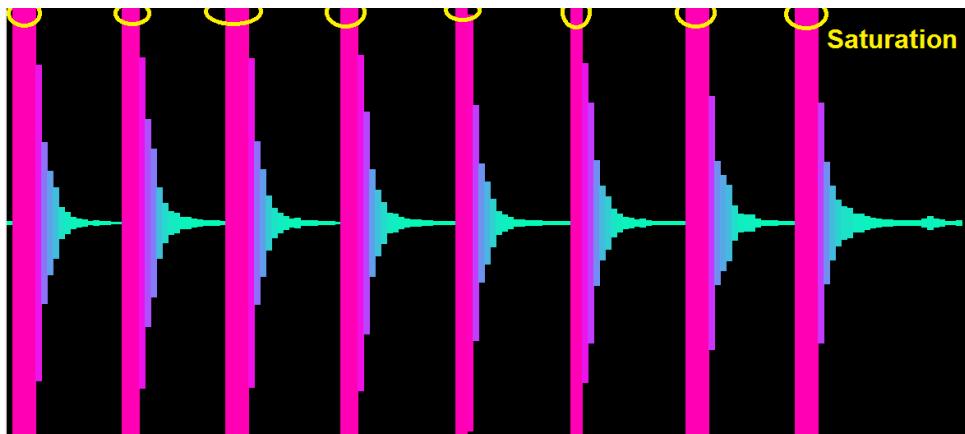
Le protocole est alors simple. Il faut :

- Brancher le microphone à l'ordinateur sur lequel on exécute le programme accompagnant le projet. Ce dernier affiche directement les amplitudes sonores : chaque barre correspond à l'amplitude sonore maximale dans un relevé audio de 40ms. La hauteur de la barre correspond à son amplitude entre 0 (ligne du milieu, hauteur nulle) et 1 (la barre touche alors les extrémités du cadre). L'échelle des hauteurs est linéaire. *Un lien menant vers une version récente du programme est en annexe* ;
- Positionner la percussion à une distance faible du micro ;
- Frapper plusieurs fois la percussion sur toutes ses parties, afin de répliquer tous les types de frappes possibles, puis regarder le graphe fourni par le programme affichant les amplitudes relatives relevées par le micro. Si l'amplitude relative est suffisamment élevée sans mettre en évidence un signe de saturation, alors la sensibilité du micro est satisfaisante pour cette distance. En se référant au cahier des charges, une amplitude relevée entre 0.15 et 0.85 est considérée satisfaisante (garantissant bonne précision et protection contre la saturation).
- Répéter l'opération en éloignant de plus en plus la percussion, ce qui permet de déterminer une distance optimale d'utilisation du micro.

Nous procédons à ces tests sur nos deux microphones : le microphone "**Windows**" installé par défaut sur notre ordinateur et le "**H2N Zoom**".

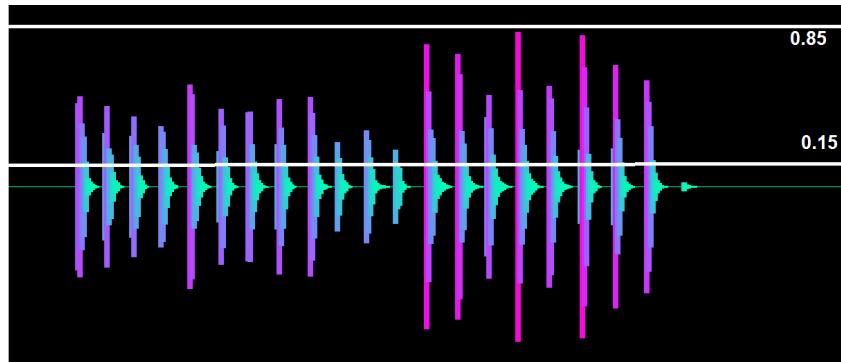
### Test du "H2N Zoom"

*Pour 20 cm (très proche) :*



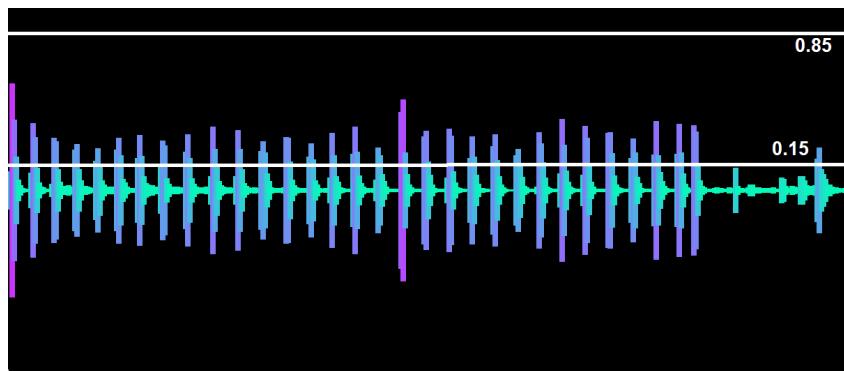
Ici, quel que soit l'endroit où on a frappé sur la percussion, apparaît le phénomène de saturation. En effet, les barres représentant l'amplitude sonore relative relevée par le micro sont à leur hauteur maximale (elles touchent le cadre). Il s'agit donc d'une amplitude relative de 1 : il y a phénomène de saturation sur le micro à cette distance.

**Pour 60 cm (bonne distance) :**



Toutes les valeurs d'amplitudes sont dans les limites imposées, quel que soit le type de frappe effectué. Les valeurs d'amplitude sont suffisamment élevées : le rapport signal/bruit devrait être très bon. Cette distance semble idéale.

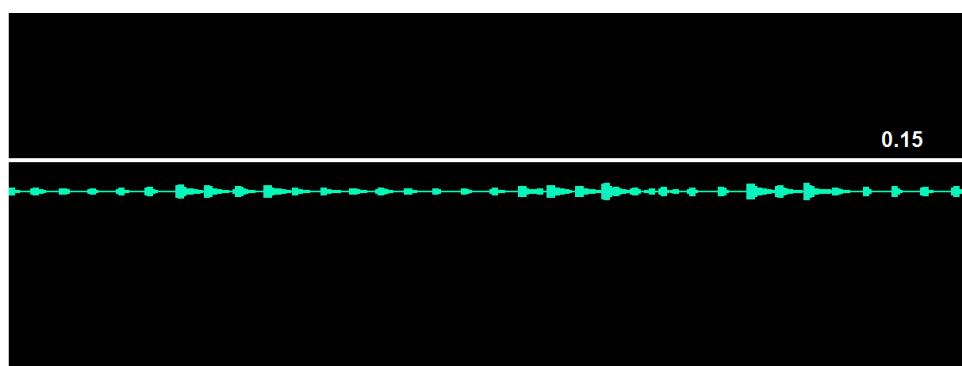
**Pour 150 cm (éloigné) :**



Les valeurs sont dans les limites imposées, mais sont relativement faibles. On pourrait se placer à cette distance, mais on préférera la distance précédente de 60 cm, optimale.

#### Test du microphone par défaut "Windows"

**Dès 50 cm :**



Les valeurs sont bien en dessous de la limite imposée dès une distance de 50 cm. Les performances du microphone par défaut "Windows" sont moins bonne que celle du "H2N Zoom" de ce point de vue là.

### **Bandé passante**

La bande passante du microphone est sa sensibilité aux différentes fréquences à relever. Les relevés du studio d'enregistrement nous indiquent que le microphone devra relever avec précision et homogénéité les fréquences s'étalant de 250 à 1500 Hz.

Le microphone est soumis à des sons de même amplitude mais de fréquences différentes de manière à recouvrir une portion du spectre fréquentiel. Il faut :

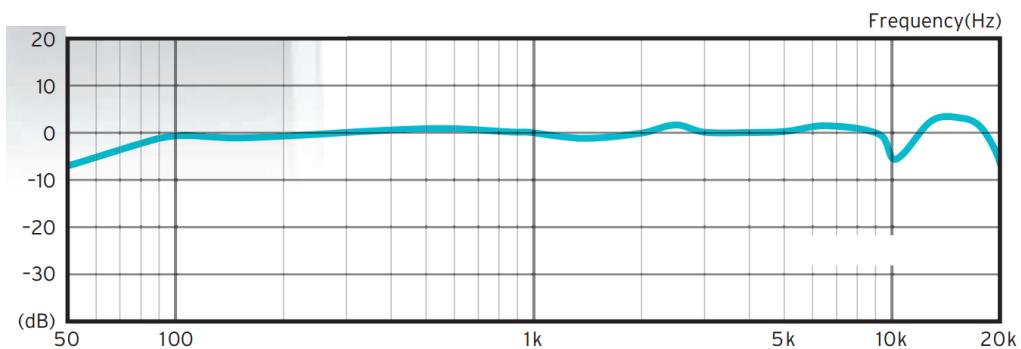
- Se munir d'une enceinte de bonne qualité dont sont négligés les effets sur la qualité de la sortie audio (dans notre cas BOSE SOUNDLINK MINI II);

- Si on dispose d'une enceinte de très haute qualité dans un environnement parfait (chambre anéchoïque), générer un bruit rose (possédant une énergie spectrale constante par bande d'octave) : ainsi, une fois capté par le micro, il suffit d'en tirer le spectre fréquentiel pour obtenir la courbe de réponse. Il faut, sinon, relever l'amplitude sonore relative donnée par le microphone pour chaque fréquence incluse dans une plage définie (par exemple dans notre cas entre 20 et 20kHz). Dans la pratique, les relevés s'effectuent en commençant par la fréquence correspondant au début de la plage à étudier, puis incrémentés entre chaque relevé par une certaine valeur et ce jusqu'à la fin de la plage à étudier (dans notre cas, nous commençons les relevés avec une fréquence de 20Hz que l'on incrémente de 10Hz entre chaque relevé et ce jusqu'à 20kHz). Il est possible d'utiliser le site internet <http://www.szynalski.com/tone-generator/> pour générer les tons de fréquences voulues. Pour ce test, nous avons écrit spécifiquement un programme informatique, s'appuyant sur le générateur de ton qui pourra être trouvé en annexe.

- Avant de tracer la courbe de réponse à partir des données relevées, appliquer à chaque amplitude relevée  $x$  la fonction  $f$  tel que  $f(x)=20 \log_{10}(\frac{x}{x_0})$  avec  $x_0$  l'amplitude en 1000Hz. Cela donnera des valeurs en dB relatifs à celle en 1000Hz (utilisée par convention).

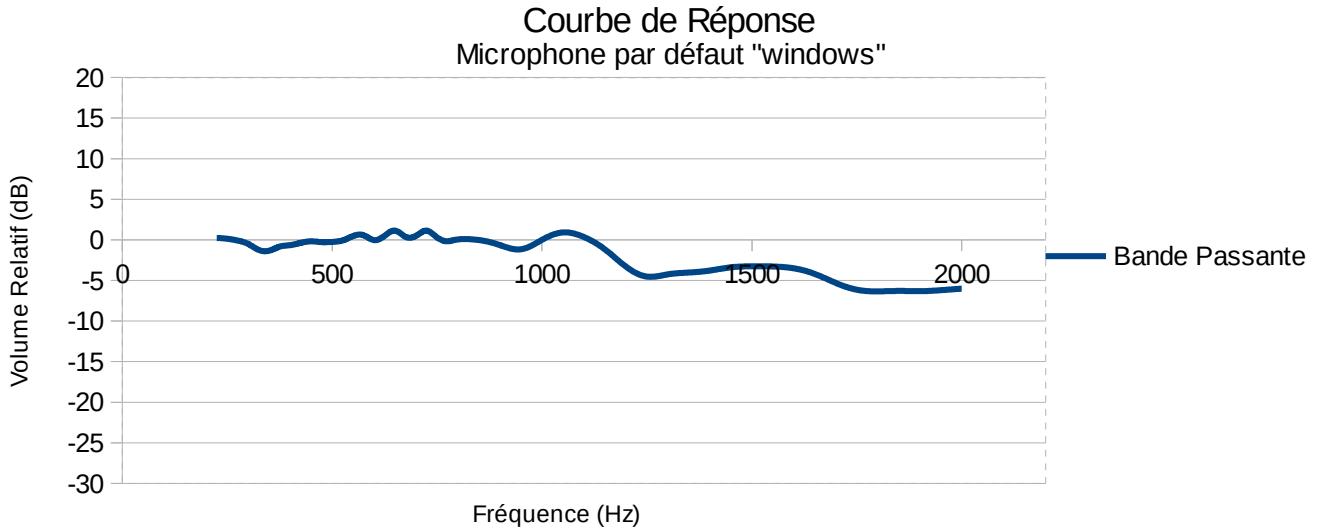
-Enfin, tracer la courbe de réponse.

Voici les résultats pour le micro "H2N Zoom":



On constate une restitution bonne et homogène dans les fréquences qui nous intéressent (de 250 à 1500 Hz). La réponse aux fréquences environnantes étant similaire voir plus faible (pour les basses par exemple) nous sommes sûrs que les fréquences non étudiées ne perturberont pas les mesures.

En revanche, le microphone par défaut "**Windows**" s'en tire moins bien :



La restitution des fréquences entre 200 Hz et 1100 Hz n'est pas mauvaise, mais elle se dégrade au delà (chute à -5dB de 1100 Hz à 2000Hz). La restitution est donc imparfaite et hétérogène dans les fréquences qui nous intéressent (de 250 à 1500Hz).

Les performances du microphone par défaut "**Windows**" sont systématiquement inférieures à celles du "**H2N Zoom**". Placé à 60 cm de la percussion, le "**H2N Zoom**" remplit tous les critères imposés par le cahier des charges. Le choix du micro "**H2N Zoom**" est donc validé, que nous placerons à 60cm de la percussion.

#### Précision des échantillons

Le programme informatique utilise 16 bits pour coder chaque échantillon audio. Il est possible d'en calculer le rapport signal-bruit maximal, qui est égal :

$$RSB = 20 \log_{10}(2^{16}) = 96.33 \text{ dB}$$

Cela correspond à la différence maximale de volume sonore qu'il peut y avoir dans le relevé audio avant perdre en qualité. Cette valeur est pour nous bien satisfaisante : on ne compte pas descendre en dessous de 10dB ni monter au dessus de 90dB. Ainsi cette résolution ne devrait pas poser de problème.

#### Fréquence d'échantillonnage

Par défaut dans le programme est mis en place une fréquence d'échantillonnage de 48kHz. Le théorème de Nyquist nous assure donc que nous pourrons capturer toutes les fréquences inférieures à 2 fois cette valeur, i.e 24kHz, ce qui est bien au dessus des 1500Hz imposés par le cahier des charges.

### 3) Analyser le flux audio

#### 3.1) Mise en place de la transformée de Fourier

La Transformée de Fourier Discrète (TFD, ou DFT en anglais) nous permettra de convertir notre relevé audio en son spectre fréquentiel. En effet, la DFT transforme une séquence finie de valeurs également espacés dans le domaine temporel en une séquence de taille égale et de valeurs complexes également espacés dans le domaine fréquentiel. Ces valeurs complexes images de la DFT sont fonction de la fréquence qu'elles représentent.

La DFT se calcule à l'aide de la relation suivante, pour un signal  $s$  de  $N_s$  échantillons, en notant  $S$  la fonction de la fréquence image :

$$S(k) = \sum_{n=0}^{N_s-1} s(n) \times e^{-2\pi i k \frac{n}{N_s}}$$

avec  $0 \leq k < N_s$  ( $k$  représente chaque échantillon)

Notons  $F_{éch}$  la fréquence d'échantillonnage du relevé audio,  $\Delta t$  la durée du relevé audio et  $\Delta f$  la précision de la DFT.

Le signal décrit est périodique de période  $\Delta t$  (donc sur ses  $N_s$  échantillons), et renseigne sur les fréquences comprises entre  $-\frac{F_{éch}}{2}$  et  $\frac{F_{éch}}{2}$  ;  
la DFT varie donc sur une plage égal à  $F_{éch}$ .

Cela va nous permettre de prouver que la précision de la DFT est égal à l'inverse de la durée correspondant au relevé audio, i.e que  $\Delta f = \frac{1}{\Delta t}$ .

On connaît la relation de base qui lie fréquence échantillonnage, nombre d'échantillon et durée du relevé audio :

$$N_s = \Delta t \times F_{éch} \quad (\text{E})$$

De plus, on sait dans le cas du résultat de la DFT que l'intervalle d'échantillonnage multiplié par le nombre d'échantillon doit forcément être égal à la fréquence d'échantillonnage (puisque les valeurs images sur une plage égal à  $F_{éch}$ ). Ainsi :

$$N_s \times \Delta f = F_{éch} \quad (\text{E'})$$

(E) et (E') nous donnent par substitution :

$$\begin{aligned} N_s &= \Delta t \times (N_s \times \Delta f) \\ \Leftrightarrow \Delta t \times \Delta f &= 1 \\ \Leftrightarrow \Delta f &= \frac{1}{\Delta t} \end{aligned}$$

On retrouve bien la propriété énoncé précédemment.

On remarquera une propriété supplémentaire : dans la pratique, on ne pourra exploiter que les fréquences inférieures à la fréquence de Nyquist  $F_{Ny} = \frac{F_{éch}}{2}$ . En effet, c'est la fréquence maximale que peut contenir un signal pour permettre sa description non ambiguë par un échantillonnage à intervalles réguliers.

En outre, la précision de la DFT dépend de la durée du relevé audio, et la plage de fréquence qu'elle représente dépend de la fréquence d'échantillonnage.

L'implémentation de la DFT ne se fait pas en utilisant la formule classique mais en employant des

algorithmes bien plus efficaces et rapides que l'on regroupe sous le terme de FFT (Fast Fourier Transform ou Transformée de Fourier Rapide). Leurs utilisations sont si répandues que les termes FFT et DFT s'utilisent de façon interchangeable. La complexité de ces algorithmes varient en  $O(N_s \log(N_s))$  avec  $N_s$  le nombre de points, quel que soit  $N_s$ , alors que la complexité de l'algorithme « naïf » s'exprime en  $O(N_s^2)$ . Ainsi, pour  $N_s = 1\ 024$ , le temps de calcul de l'algorithme rapide peut être 100 fois plus court que le calcul utilisant la formule de définition de la DFT. En effet, la FFT est particulièrement rapide pour  $N_s$  une puissance de 2. Montrons comment fonctionne l'optimisation de la DFT pour  $N_s$  puissance de 2 :

Écrivons la DFT d'un signal  $s$  quelconque, en notant  $W_{N_s}^k = e^{-2\pi i k \frac{n}{N_s}}$  :

$$\begin{aligned} S(k) &= \sum_{n=0}^{N_s-1} s(n) \times (W_{N_s}^k)^n \\ &= \sum_{n=0}^{(N_s/2)-1} s(2n) \times (W_{N_s}^k)^{2n} + s(2n+1) \times (W_{N_s}^k)^{2n+1} \\ &= \sum_{n=0}^{(N_s/2)-1} s(2n) \times (W_{N_s}^k)^{2n} + \sum_{n=0}^{(N_s/2)-1} s(2n+1) \times (W_{N_s}^k)^{2n+1} \\ S(k) &= \sum_{n=0}^{(N_s/2)-1} s(2n) \times (W_{N_s}^k)^{2n} + (W_{N_s}^k)^{\sum_{n=0}^{(N_s/2)-1} s(2n+1) \times (W_{N_s}^k)^{2n}} \end{aligned}$$

On remarque que  $(W_{N_s}^k)^2 = e^{-2\pi i \frac{2}{N_s}} = e^{-2\pi i \frac{1}{N_s/2}} = (W_{N_s/2}^k)$ . Ainsi  $(W_{N_s}^k)^2 = (W_{N_s/2}^k)$ . On continue :

$$S(k) = \sum_{n=0}^{(N_s/2)-1} s(2n) \times (W_{N_s/2}^k)^n + (W_{N_s}^k)^{\sum_{n=0}^{(N_s/2)-1} s(2n+1) \times (W_{N_s/2}^k)^n}$$

Ici, nous retrouvons deux DFT de taille  $N_s/2$  sur les termes pairs d'une part et les termes impairs d'autre part. Notons  $E(k)$  la DFT de taille  $N_s/2$  sur les termes pairs, et  $O(k)$  celle sur les termes impairs. Alors :

$$S(k) = E(k) + (W_{N_s}^k)O(k)$$

Ici, on va pouvoir tirer parti de la périodicité de la DFT, comme nous l'avions vu précédemment. En effet, puisque  $E(k)$  et  $O(k)$  sont deux DFT de taille  $N_s/2$ , on sait que :

$$\begin{cases} E(k + \frac{N_s}{2}) = E(k) \\ O(k + \frac{N_s}{2}) = O(k) \end{cases}$$

De plus,

$$\begin{aligned} W_{N_s}^{k+N_s/2} &= e^{-2\pi i \frac{(k+N_s/2)}{N_s}} \\ W_{N_s}^{k+N_s/2} &= e^{-2\pi i \frac{k}{N_s} + (-2\pi i \frac{N_s/2}{N_s})} \\ W_{N_s}^{k+N_s/2} &= e^{-2\pi i \frac{k}{N_s}} \times e^{(-2\pi i \frac{N_s/2}{N_s})} \\ W_{N_s}^{k+N_s/2} &= e^{-2\pi i \frac{k}{N_s}} \cdot e^{-\pi i} \end{aligned}$$

$$W_{N_s}^{k+N_s/2} = -e^{-2\pi i \frac{k}{N_s}} \text{ car } e^{-\pi i} = e^{\pi i} = -1$$

$$W_{N_s}^{k+N_s/2} = -W_{N_s}^k$$

Cela va nous permettre de calculer  $S(k)$  de manière bien plus efficace. En effet :

$$S(k) = E(k) + (W_{N_s}^k)O(k)$$

Ce que l'on peut diviser en 2 intervalles au niveau de  $N_s/2$  :

$$\begin{cases} S(k) = E(k) + (W_{N_s}^k)O(k) & 0 \leq k < \frac{N_s}{2} \\ S(k + \frac{N_s}{2}) = E(k + \frac{N_s}{2}) + (W_{N_s/2}^{k+N_s/2})O(k + \frac{N_s}{2}) = E(k) - (W_{N_s}^k)O(k) & \frac{N_s}{2} \leq k < N_s \end{cases}$$

Ainsi, il ne suffit de faire que les deux DFT  $E(k)$  et  $O(k)$ , qui sont de taille deux fois moindre que la DFT originale, dont on utilisera les valeurs deux fois (dans chaque moitié). puisqu'on a supposé  $N_s$  une puissance de 2, on peut continuer de même avec  $E(k)$  et  $O(k)$  (qui sont de taille  $N_s/2$ , toujours une puissance de deux) en les décomposant en deux nouvelles DFT de taille  $N_s/4$ , etc. Logiquement, on aura autant d'étapes que la puissance à laquelle a été élevé 2 pour trouver  $N_s$ , soit exactement  $\log_2(N_s)$  ou  $\log(N_s)$  (en effet, en algorithmie, lorsque l'on ne précise pas la base du logarithme, on considère que c'est un logarithme de base 2).

Voilà une implémentation possible très simple de la FFT, écrite en C++ :

```
// Remplace l'appellation "std::vector<Complex>",
// i.e tableau de valeurs complexes,
// par "CArray"
typedef std::vector<Complex> CArray

/*
 * Fonction récursive qui calcule la FFT.
 * Prend en argument d'entrée un tableau de valeurs complexes,
 * sur lequel sera stocké la FFT après calcul.
 * Dans le cas de données audios, toutes ces valeurs sont réelles
 * ce qui peut laisser place à de l'optimisation, mais je ne le fais pas ici.
 */
void fft(CArray &x)
{
    // Evaluer la taille du tableau de valeurs. Si elle est de 1
    // i.e qu'on ne peut plus faire plus de DFT, on quitte la fonction.
    const int N = x.size();
    if(N <= 1) return;

    // On découpe la DFT en 2 DFT 2 fois plus petites.
    CArray even = x[slice(0,N/2,2)]; // On ne prend que les éléments aux rangs pairs
    CArray odd = x[slice(1,N/2,2)]; // On ne prend que les éléments aux rangs impairs

    // On lance une nouvelle DFT sur les éléments aux rangs pairs, puis impairs.
    fft(even);
    fft(odd);

    // Enfin, on combine la DFT de deux moitiés en une DFT entière.
    for(int k = 0 ; k < N/2 ; k++)
    {
        // La fonction polar est semblable à notre notation W.
        // En effet, polar(float a, float b) = a*e^(ib)
        Complex t = polar(1.0,-2 * PI * k / N) * odd[k];
        x[k] = even[k] + t;
        x[k+N/2] = even[k] - t;
    }
}
```

Il s'agit ici d'une fonction récursive, ce qui n'en fait pas l'implémentation la plus rapide de l'algorithme. De plus, il faut que la taille du tableau de valeurs d'entrée soit une puissance de 2.

En pratique, dans le programme créé, est utilisée une bibliothèque externe pour effectuer le calcul de la FFT (*FFTW*, pour Fastest Fourier Transform in the West). Cette dernière, écrite en C, est hautement optimisée. Voici comment elle est implémentée dans le programme (il suffit seulement d'appeler les fonctions qu'elle nous fournit) :

```
// Fonction utilisant la bibliothèque FFTW pour le calcul de la FFT.
// Les arguments d'entrées sont :
// std::vector<float> samples : un tableau de réel appelé "samples" représentant les données audios
// std::vector<float> spectreFrequenziel : un tableau de réel représentant le spectre fréquentiel
void fft(std::vector<int16_t> &samples, std::vector<float> &spectreFrequenziel) {

    // Réserver l'espace mémoire pour les arguments de la FFT à réaliser avec FFTW.
    float* in = (float*)fftwf_malloc(sizeof(float)*samples.size());
    fftwf_complex* out = (fftwf_complex*)fftwf_malloc(sizeof(fftwf_complex)*samples.size());

    // Remplir le tableau de valeurs d'entrées pour la FFT. On pourra mettre en place à cet endroit la fonction de fenêtrage...
    for (int i = 0; i < samples.size(); i++) {
        in[i] = static_cast<float>(samples[i]);
    }

    /* Initialiser la FFT avec les bons paramètres, à savoir :
     * Le nombre de données audios,
     * un pointeur vers le tableau de valeurs réels des données d'entrées,
     * un pointeur vers le tableau de valeurs complexes des données images de la FFT (fonction de la fréquence),
     * un dernier paramètre pour indiquer le mode de fonctionnement de la FFT :
     * Ici, FFTW_ESTIMATE permet à FFTW de trouver rapidement un algorithme (probablement sub-optimal),
     * sans écraser les tableaux in et out à la création du fftwf_plan.
    */
    fftwf_plan plan = fftwf_plan_dft_r2c_1d(samples.size(), in, out, FFTW_ESTIMATE);

    // Exécuter l'algorithme de la FFT
    fftwf_execute(plan);

    /* Ici on pourra remplir le spectre fréquentiel, le résultat de la FFT étant stocké dans "out"
     * ...
     * ...
     * ...
    */

    // Libérer l'espace mémoire précédemment alloué
    fftwf_free(in);
    fftwf_free(out);
    fftwf_destroy_plan(plan);
}
```

L'algorithme est particulièrement rapide pour des ensembles de taille puissance de 2. Comparons la vitesse de ces deux algorithmes sur un ensemble de 65536 échantillons (relevé audio d'1.37s avec une fréquence d'échantillonnage de 48kHz) en utilisant le type « float » (nombre à virgule flottante à précision simple). On montrera aussi que le temps de calcul peut être très rapide sur un nombre d'échantillons autre qu'une puissance de 2, mais l'algorithme est alors moins efficace :

	Temps de Calcul Moyen (ms) 65536 Échantillons	Temps de Calcul Moyen (ms) 48000 Échantillons
Algorithme Simple Récursif	55.6	Ø
FFTW	1.6	2.5

En effet, FFTW utilise de nombreuses astuces pour arriver à une telle vitesse : emploi d'une boucle à la place d'une récursion (en réarrangeant les éléments en « bit-reverse »), valeurs pré-calculées... Ainsi la FFT qui sera utilisée est extrêmement rapide, avec une valeur compatible au cahier des charges (temps de calcul de 2.5 ms contre les 100 ms maximum égaux à 10 % d'1s). De plus,

FFTW ne nous limite pas à des ensembles de taille puissance de 2 mais permet de traiter efficacement des ensembles de taille quelconque (au détriment d'un peu de vitesse).

Enfin, on veut s'assurer que le spectre fréquentiel puisse être précis au Hz près (critère du cahier des charges). Sur ordinateur, on représente les nombres réels soit par un nombre à virgule flottante à précision simple (aussi appelé « float ») dans plusieurs langages de programmation, abréviation de l'anglais « floating-point number ») codé sur 32 bits, soit par un nombre à virgule flottante à précision double (aussi appelé « double ») codé sur 64 bits. Tandis qu'un float permet des calculs plus rapides, il est bien moins précis qu'un double. En effet, leurs imprécisions maximales notées  $\epsilon$  sont :

	$\epsilon$
Float	$2^{-23} \approx 1.19 \cdot 10^{-7}$
Double	$2^{-52} \approx 2.22 \cdot 10^{-16}$

L'imprécision maximale d'un spectre fréquentiel réalisé sur  $N$  échantillons est majorée par  $\epsilon \log(N)$ . Ainsi, pour  $N$  égal à la fréquence d'échantillonnage, on devrait avoir une précision fréquentielle de 1 Hz puisque le relevé durerait 1s. Voyons comment les deux types de variables pourraient faire dériver la précision, en prenant plusieurs fréquences d'échantillonnage caractéristiques :

	$\epsilon \log(8000)$	$\epsilon \log(44100)$	$\epsilon \log(48000)$	$\epsilon \log(96000)$
Float	1.54292e-6	1.83599e-6	1.85053e-6	1.96953e-6
Double	2.8784041e-15	3.425125e-15	3.4522658e-15	3.6742658e-15

L'imprécision est réellement négligeable. On préférera donc l'utilisation du float pour sa vitesse d'exécution, comme on a pu le voir dans le test de vitesse des algorithmes précédent.

Enfin, pour vérifier les performances de la FFT, cette dernière a été implémentée dans l'interface utilisateur support du programme, sur lequel il est possible de faire 2 choses :

- Générer un ton d'une certaine fréquence et voir quel est le résultat de la FFT (cela permet d'analyser un son parfait). *La fonction qui permet de générer le ton voulu est en annexe.*
- Analyser le flux audio entrant venant du microphone connecté à l'ordinateur (on peut gérer la durée du relevé audio via le programme).

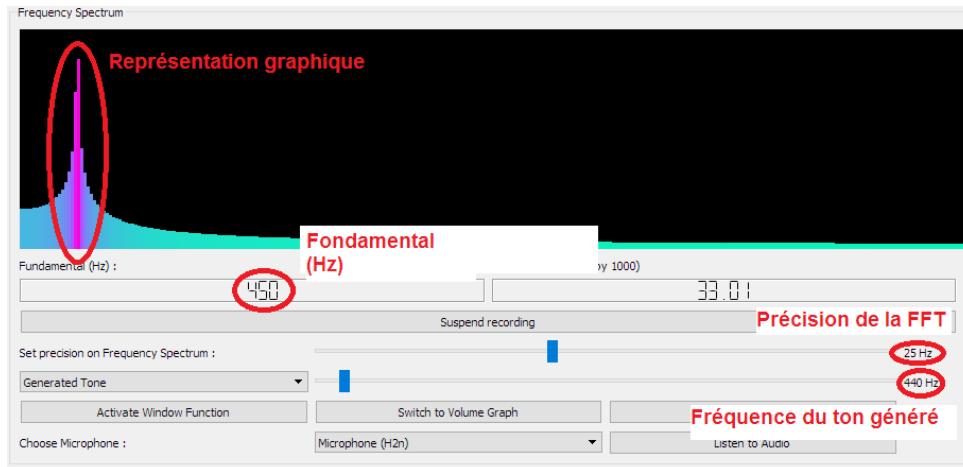
Avant de continuer, il est nécessaire de préciser comment le spectre fréquentiel est construit une fois la FFT réalisée. Comme nous l'avions vu précédemment, les valeurs complexes  $z_f = a_f + ib_f$  résultats de la FFT sont fonctions de la fréquence qu'elles représentent. Leurs modules correspondent à l'amplitude des fréquences qu'elles représentent, tandis que leurs arguments correspondent aux déphasages de ces fréquences dans le signal original. Ainsi, en notant  $A_f$  l'amplitude de la fréquence  $f$  et  $P_f$  son déphasage on a :

$$A_f = |z_f| \Leftrightarrow A_f = \sqrt{a_f^2 + b_f^2}$$

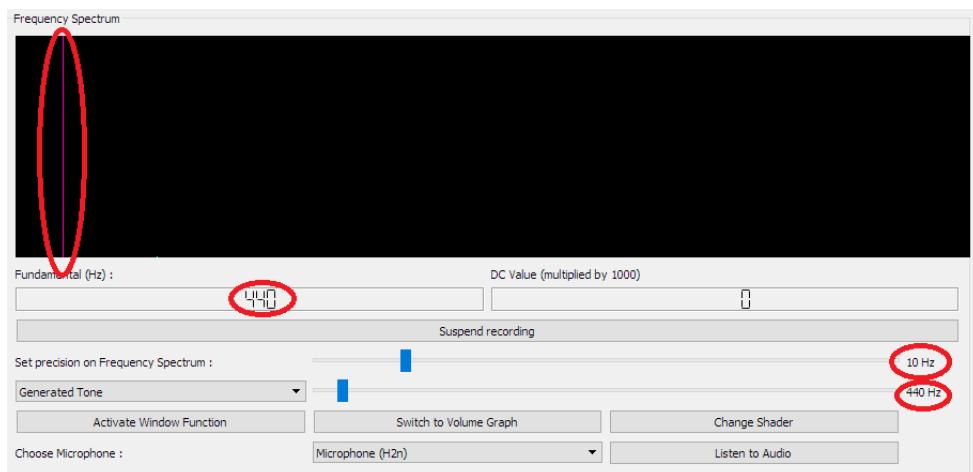
$$P_f = \arg(z_f) \Leftrightarrow P_f = \arctan\left(\frac{b_f}{a_f}\right)$$

Pour faire le spectre fréquentiel, seul l'amplitude de la fréquence nous intéresse.

On peut alors aisément vérifier les performances réels de la FFT mise en place :  
**Ton généré de 440 Hz dans le programme :**

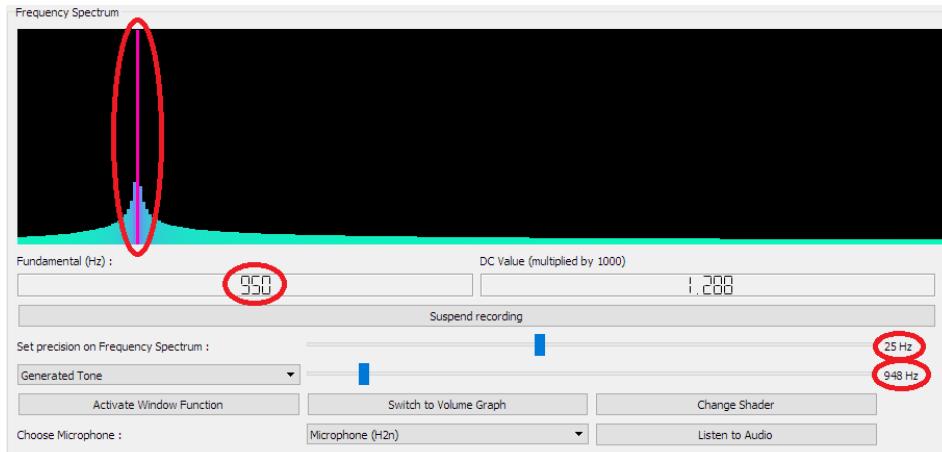


*Toutes les informations concordent bien avec le résultat de la FFT*

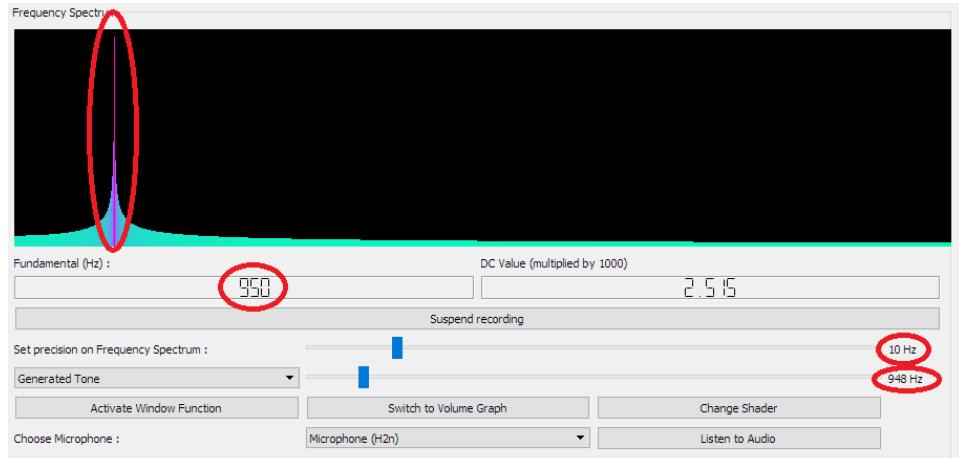


*De même ici, mais nous avons augmenté la précision de la FFT qui est maintenant de 10Hz (le ton généré simule donc un relevé audio durant 1/10 = 0.1s soit 100ms)*

**Ton généré de 948 Hz :**

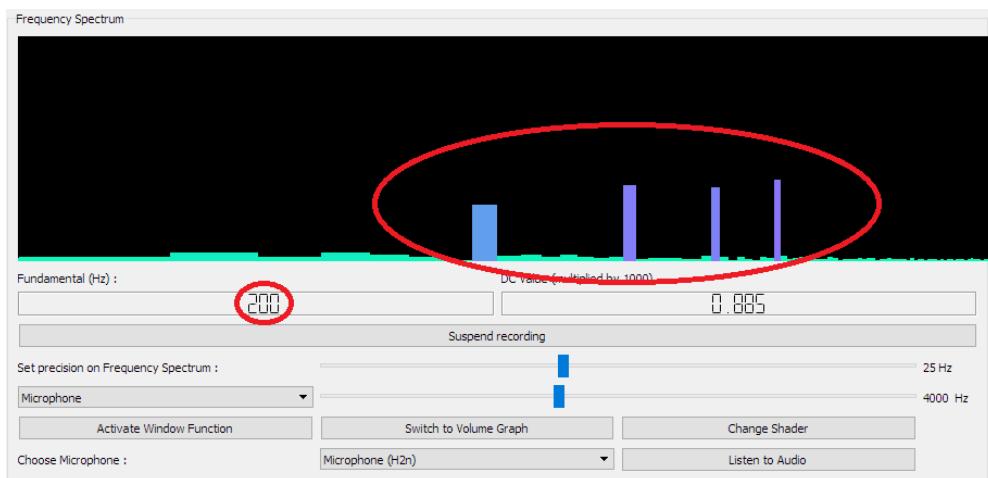


*Après avoir changé la fréquence (948 Hz), la FFT nous donne toujours un résultat correct*



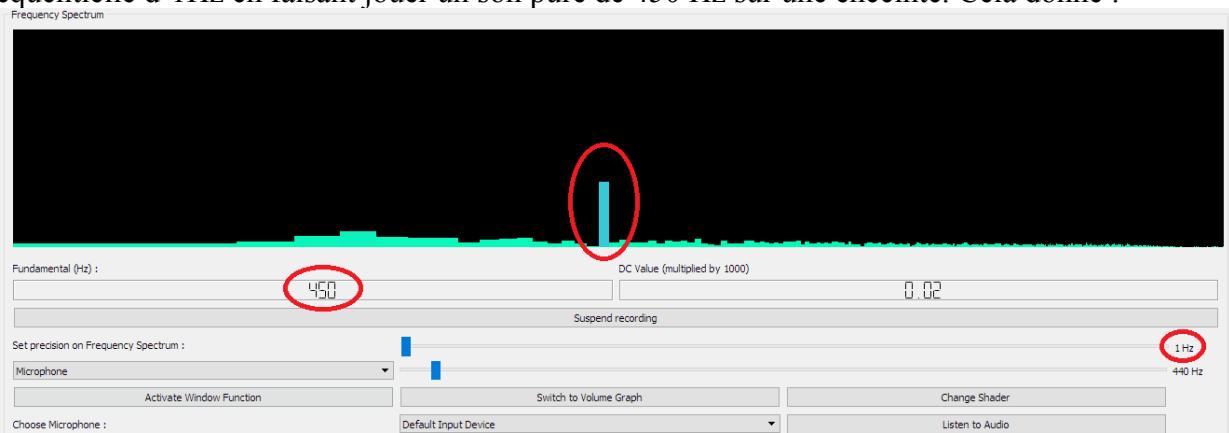
*De même après avoir changé la précision une nouvelle fois*

**Ton comportant des fréquences de 200, 400, 600 et 800 Hz reçu via le microphone :**



*Ici l'échelle utilisée pour représenter les fréquences est logarithmique (pour mieux mettre en avant ces petites fréquences). Les 4 fréquences sont bien retrouvées par la FFT.*

Enfin, on vérifie que la FFT, en situation d'utilisation réelle, peut garantir une précision fréquentielle d'1Hz en faisant jouer un son pure de 450 Hz sur une enceinte. Cela donne :



Ainsi, les résultats expérimentaux concordent avec les résultats simulés ainsi que le critère du cahier des charges imposant une imprécision fréquentielle inférieure à 1 Hz lorsque nécessaire.

### 3.2) Fonction de fenêtrage

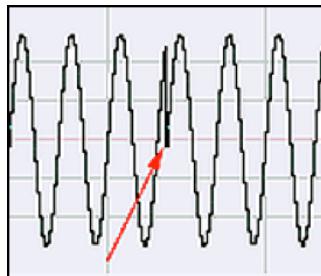
La technique du 'fenêtrage' sera appliquée sur la FFT ; c'est une technique servant à minimiser les distorsions qui provoquent une fuite spectrale de la FFT. En utilisant correctement les fonctions de fenêtrage, la résolution spectrale des résultats dans le domaine fréquentiel s'en trouve accrue. Le principe est simple :

Pour observer un signal sur une durée finie, on le multiplie par une fonction fenêtre d'observation (également appelée fenêtre de pondération). La plus simple est la fenêtre rectangulaire (ou porte), définie telle que :

$$w(t) = \begin{cases} 1, & t \in [T_1, T_2] \\ 0, & \text{otherwise} \end{cases}$$

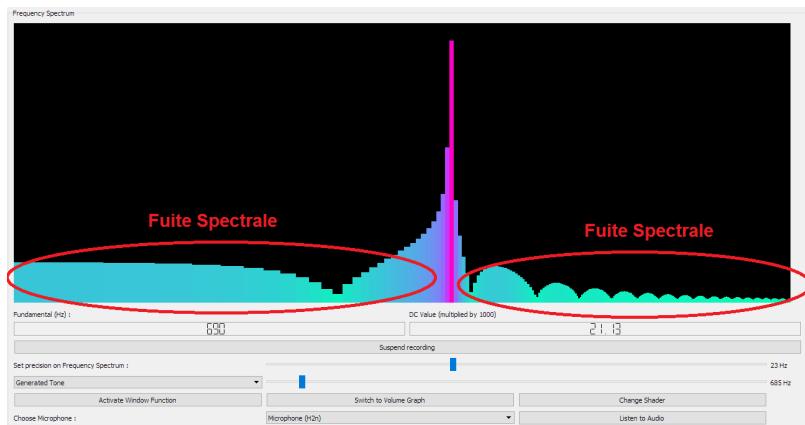
Ainsi, quand on multiplie un signal  $s(t)$  par cette fenêtre, on n'obtient plus que la partie comprise entre  $T_1$  et  $T_2$  de ce signal : on l'« observe » sur une durée allant de  $T_1$  à  $T_2$ . Toute observation étant de durée limitée, on applique forcément une fenêtre par rapport à un signal théorique infini ; on utilise donc au moins une fenêtre, même si on l'applique sans s'en rendre compte. Au lieu d'étudier le signal  $s(t)$ , on étudie le signal tronqué :  $s_w(t) = s(t)w(t)$ . L'utilisation d'une fenêtre de pondération va donc changer la transformée de Fourier du signal.

Prenons un exemple simple. Lorsque l'on fait la FFT d'un signal fini, la formule « considère » que ce signal se répète identique à lui-même à l'infini, comme nous l'avions vu précédemment. Or cela pose problème si on a un signal périodique qui est « coupé » avant d'avoir pu finir sa période :



C'est cela qui va donner naissance à des fuites spectrales. Générons un exemple avec mon programme. On trouve un signal susceptible de donner naissance à un tel phénomène en générant un ton qui se répétera un nombre non entier de fois (ce qui peut être imposé à l'aide de la fréquence du ton généré ainsi que de la "précision" de la FFT, qui est je le rappelle strictement équivalente à sa fréquence de rafraîchissement, i.e l'inverse de la durée des données audios générées) ; par exemple on peut prendre un ton de 685 Hz avec une fréquence de rafraîchissement de 23Hz, soit un signal qui se répétera  $623/23 = 29.78$  fois.

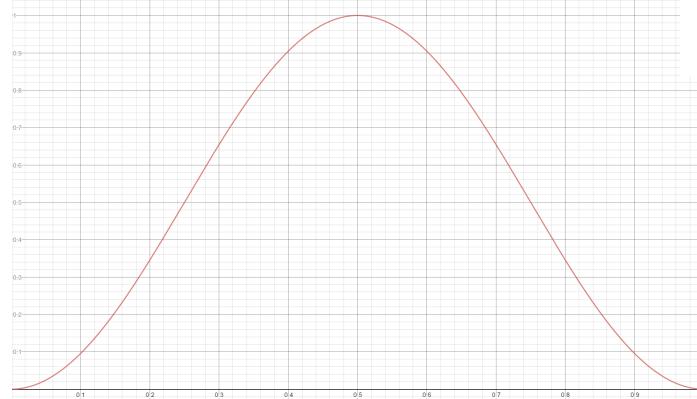
**FFT non fenêtrée :**



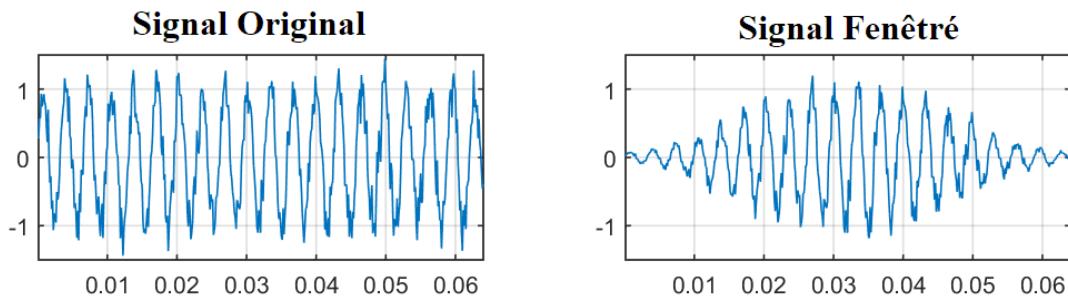
On comprend tout l'intérêt d'une fonction de fenêtrage. Ici, on utilisera une fenêtre de Hann. Celle-ci est définie pour chaque échantillon  $k$  comme suit, pour un ensemble de taille  $N$  :

$$w(k) = \frac{1}{2} - \frac{1}{2} \times \cos\left(2\pi \frac{k}{N}\right) \quad \text{avec } 0 \leq k < N$$

Voici à quoi elle ressemble graphiquement :

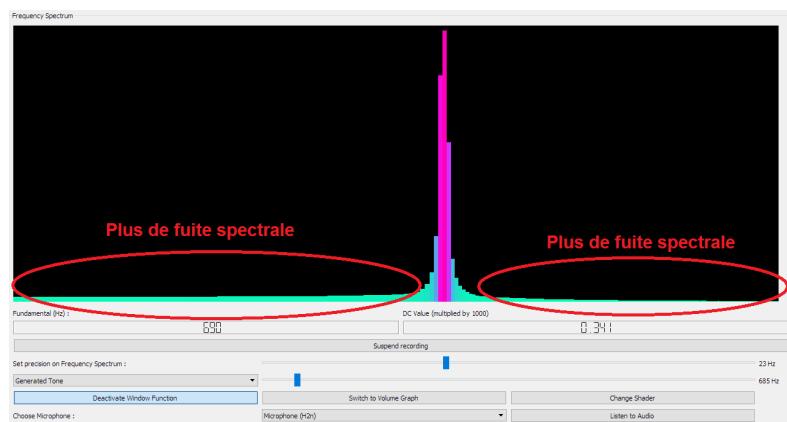


Et voici ce que donne cette fenêtre appliquée à un signal quelconque :



Comme on peut le voir, la fonction de fenêtrage « périodise » le signal. Dans le cas du signal généré précédemment sur le programme, l'utilisation d'une fonction de fenêtrage règle très clairement le problème :

#### **FFT fenêtré (Hann) :**

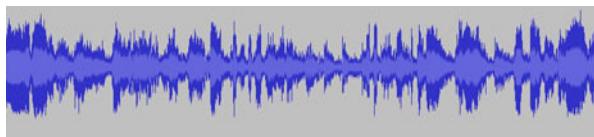


Ainsi la fonction de fenêtrage permet de réduire fortement l'impact de certains artefacts qui seraient susceptibles d'apparaître dans les relevés audios qui risquerait de les fausser. L'imprécision légère qu'elle génère autour des pics de fréquences principales n'est pas assez importante pour dérégler l'algorithme qui permettra de différencier les différents types de frappe. Son usage est validé.

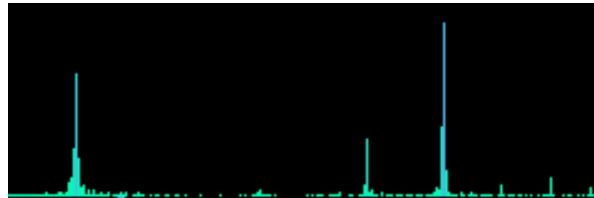
### 3.3) Reconnaître une frappe

Maintenant que nous savons comment effectuer efficacement nos spectres fréquentiels, il s'agit de mettre tout cela en œuvre afin de trouver un algorithme qui puisse reconnaître une frappe. Pour se faire, nous allons comparer la FFT du signal entrant avec la FFT d'une frappe déjà enregistrée sur l'ordinateur (par exemple les données audios du Minimal Studio) : si ces deux FFT sont suffisamment semblables, alors on pourra considérer qu'une frappe a été effectuée. On pourra alors transmettre leur « taux de ressemblance » au micro-contrôleur pour lui indiquer à quel point la frappe relevée était proche d'une frappe harmonique.

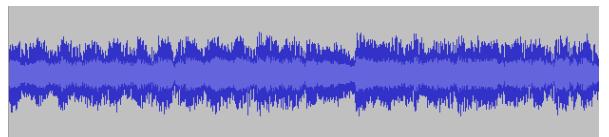
Données audios de la base de données  
Correspond ici à une frappe harmonique parfaite



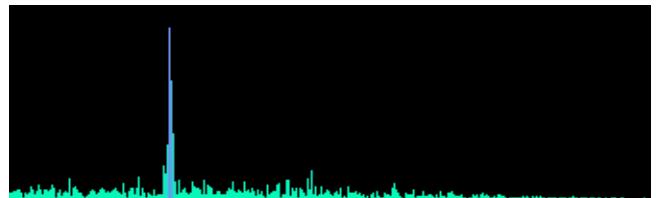
FFT



Données audios correspondant au flux audio entrant



FFT



*Comparaison*

**Similarité des deux FFT déterminées.  
Il ne s'agit pas du même son.**

Mais comment quantifier ce « taux de ressemblance » ? Nous allons utiliser le procédé mathématique de corrélation croisée. Ce dernier va nous fournir un taux de corrélation (taux de ressemblance) entre les deux FFT. De plus, la corrélation croisée nous permet de trouver aussi un taux de corrélation entre les deux FFT en prenant en compte un décalage en fréquence : cela peut être utile en permettant de compenser d'éventuelles imprécisions.

### Modèle pour la reconnaissance de frappe (Simulé)

Soient  $u$  et  $v$  les fonctions qui associent respectivement à une fréquence  $f$  son amplitude dans la 1ère FFT et son amplitude dans la 2nde FFT ;

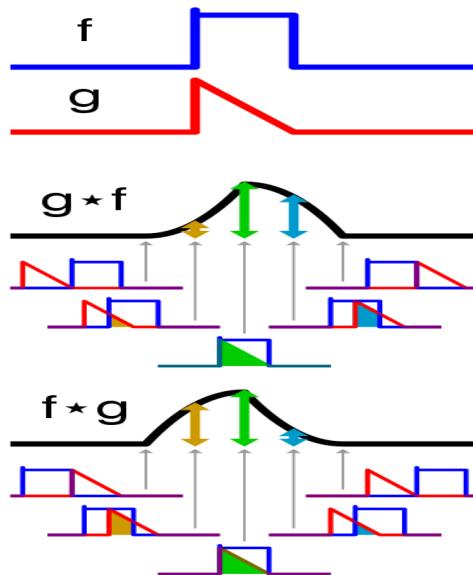
Soit  $F$  l'ensemble des valeurs de fréquence que recouvrent conjointement  $u$  et  $v$  (La précision fréquentielle  $\delta f$  d'une FFT doit donc être un multiple de l'autre, autrement cet ensemble serait vide. Dans la pratique, on s'arrangera pour que les deux FFT aient la même précision fréquentielle) ;

Soit  $\Delta f$  le décalage en fréquence (forcément multiple de la précision fréquentielle :  $\Delta f = k \delta f$ ,  $k \in \mathbb{Z}$ ). On considérera que les valeurs pour lesquelles ne sont pas définies  $u$  et  $v$  après un certain décalage ont pour image 0.

Alors la corrélation croisée, pour nos données réels, s'écrit :

$$(u * v)(\Delta f) = \sum_{f \in F} u(f)v(f + \Delta f)$$

Exemple sur deux fonctions continues quelconques :



Normalement, on se sert de la corrélation croisée pour étudier la similarité de deux signaux mais aussi leur décalage (ici, le paramètre  $\Delta f$ ). Comme nous voulons que les deux FFT soient similaires sur les mêmes fréquences, ce décalage devrait être nul.

Mais dans le programme informatique, on fera varier  $\Delta f$  entre  $-\delta f$  et  $\delta f$  puis on prendra la valeur maximale. Ainsi, nous serons certains d'avoir remédié à toute forme d'imprécision fréquentielle, tout en limitant le risque qu'une FFT décalée provenant d'un son totalement différent soit considéré comme semblable. Dans la pratique, lors d'une frappe, le maximum sera donc presque toujours atteint pour  $\Delta f = 0$ .

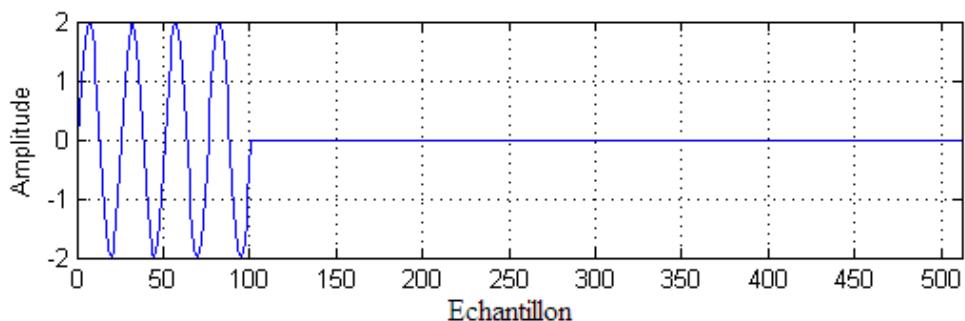
### Technique du « zero-padding » (bourrage de zéro)

Les deux FFT à comparer doivent avoir la même précision fréquentielle. Comme nous l'avions vu, cela est équivalent à dire que les relevés audios qu'ils représentent sont de même durées. Or les relevés stockés dans notre base de données sont de tailles fixes (et donc de durée fixe) ;

Si la durée du flux audio entrant est inférieur à cette durée, alors il n'y a pas de problème : on n'en sélectionnera qu'une partie pour créer notre FFT de référence.

Sinon, on ajoute un nombre suffisant d'échantillons pour simuler un relevé de la durée souhaitée. Ces échantillons ajoutés valent tous 0, d'où le terme de zero-padding ou bourrage de zéro.

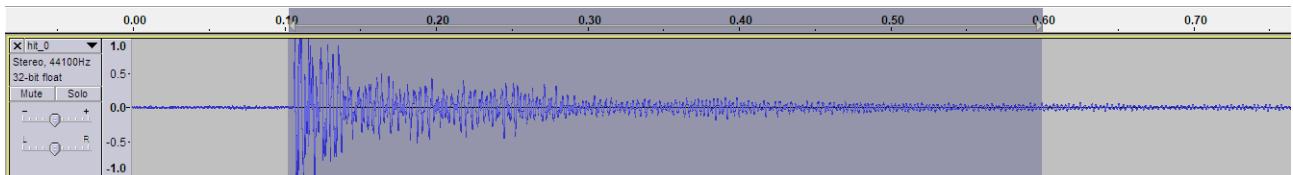
Voici un exemple de l'application de cette technique. Le signal est « bourré de zéro » de l'échantillon 100 jusqu'à l'échantillon 512.



En outre, cette technique permet aussi, dans le cas général, d'accélérer le calcul de la FFT en faisant en sorte que le nombre d'échantillons du signal devienne une puissance de 2, ce qui a aussi pour effet d'améliorer la précision de la FFT (plus d'échantillons, donc virtuellement un relevé plus long, donc une meilleure précision fréquentielle). Bien que cette technique introduise également quelques problèmes comme des fuites spectrales, ceux-ci ont été en réalité déjà réglés avec la mise en place d'une fonction de fenêtrage.

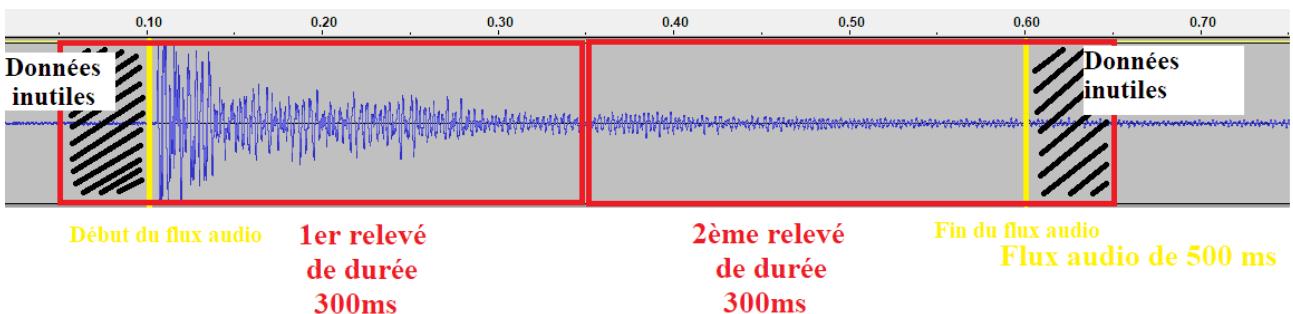
On pourra trouver en annexes les codes, écrit en C++, permettant d'effectuer la FFT (avec fonction de fenêtrage) ainsi que la corrélation croisée. Ils sont également accompagnés de la fonction permettant d'effectuer le zero-padding et d'une fonction optimisée permettant de trouver la puissance de 2 supérieure la plus proche d'un entier.

Comme nous l'avions au début du dossier, les pics de fréquence caractéristiques d'une frappe peuvent être très fins, s'étalant parfois sur seulement 4-5Hz. Notre modèle mathématique nous dit donc que la FFT devrait être au moins aussi précise, i.e que la durée du relevé audio soit d'au moins  $1/4 = 0.250\text{s}$  soit 250ms pour une précision de 4Hz ou  $1/5 = 0.2\text{s}$  soit 200ms pour une précision de 5Hz. Mais la durée moyenne d'une frappe est d'environ 500ms :



Flux audio correspondant à une frappe harmonique, ouvert sur Audacity.

Pour être certain de capturer uniquement les données audios correspondant à une frappe, il faut que la durée du relevé soit la moitié de la durée de la frappe, soit 250ms. Ainsi, il y aura nécessairement au moins un relevé qui contiendra uniquement les données audio d'une frappe. En effet, pour une durée de relevé plus longue, il se pourrait qu'aucun relevé ne contienne uniquement les données audios d'une frappe. Ce cas spécial peut arriver si les données audios d'une frappe chevauchent en leur milieu deux relevés consécutifs ; alors chaque relevé contient la moitié des données audios de la frappe, et de plus des données inutiles en début ou en fin de relevé :



#### Écarts avec le cahier des charges (théorique) :

Ainsi, on ne pourrait descendre qu'à une précision maximale de 4Hz. Le critère du cahier des charges qui impose une précision au Hz près est donc, d'après le modèle utilisé, quelque peu irréalisable : non seulement le flux audio entrant après une frappe pourrait se corrompre en se chargeant de données inutiles, mais de plus, pour une fréquence de rafraîchissement de 1Hz, i.e un relevé d'1s, cela entraînerait un zero-padding très important sur les données audios de la frappe de référence stockées dans la base de données, ce qui pourrait là aussi corrompre la FFT résultante.

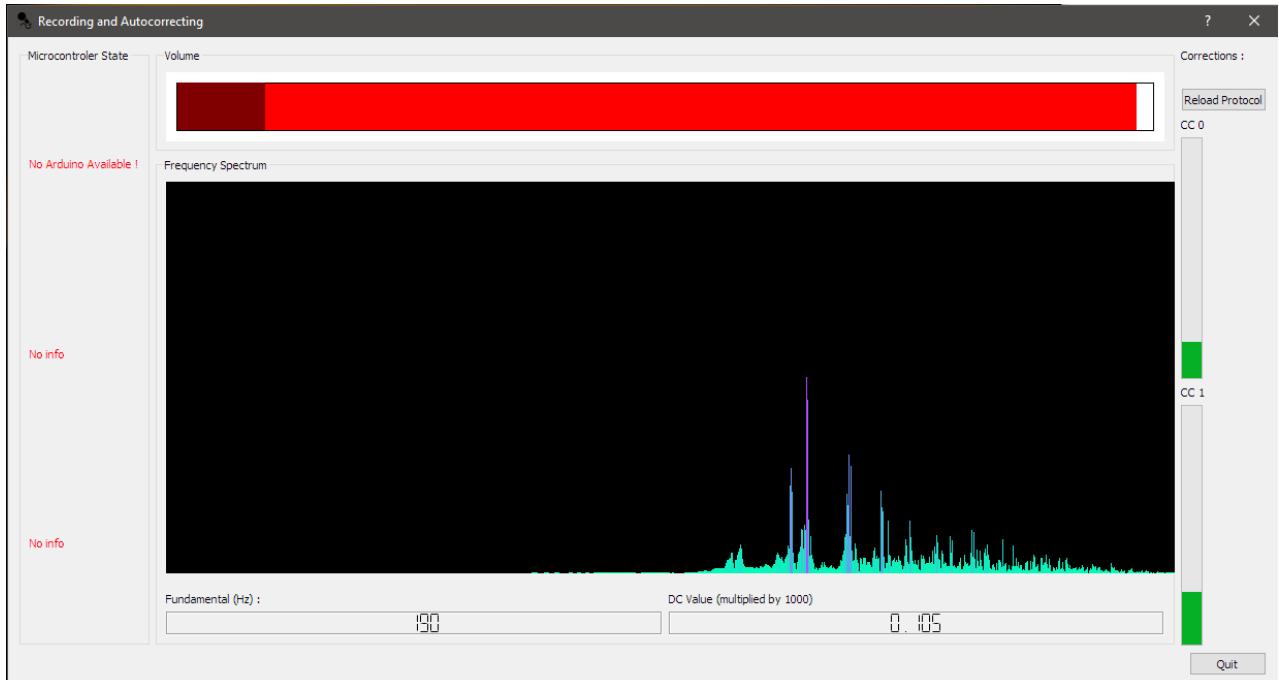
En revanche, le critère du cahier des charges imposant une reconnaissance au moins une fois sur deux de la frappe est en accord avec le modèle utilisé : avec une précision de 4Hz, le programme informatique devrait reconnaître chaque frappe et ne pas être trop perturbé par les bruits environnants.

### Résultats expérimentaux sur la reconnaissance de frappe (Réel)

Voyons comment se comporte le programme informatique en situation réelle. Pour vérifier ses performances, nous allons, tout en faisant varier la précision de la FFT à chaque étape :

- Ne pas faire de bruit et vérifier que le programme ne détecte rien ;
- Jouer une harmonique parfaite sur la percussion à distance idéale du microphone ;
- Créer des bruits parasites (discussions humaines, bruit de fond ambiant, claquer ses mains...) pour vérifier que le programme ne génère pas, ou peu de faux-positifs ;
- Jouer une harmonique parfaite sur la percussion à distance idéale du microphone mais avec des bruits parasites autours pour vérifier que le programme reconnaît tout de même la frappe.

Voici, dans le programme, ce à quoi ressemble la fenêtre ouverte durant la phase d'écoute et de corrections :



Les barres CC0 et CC1 affichent les résultats des corrélations croisées entre la FFT du flux audio entrant et celles d'une frappe harmonique parfaite. Il y en a deux pour être certain de combler toute forme d'imprécision ; ainsi dans la pratique elles sont lors d'une frappe pratiquement toujours au même niveau. Aux niveaux où on peut les voir ci-dessus on peut considérer qu'une frappe a eu lieu (ce qui était effectivement le cas). L'échelle utilisée pour la représentation du spectre fréquentiel est logarithmique.

Voici les résultats de l'expérience (taux de corrélations notés CC sur 5 mesures) :

1 Hz	CC Mesure 1	CC Mesure 2	CC Mesure 3	CC Mesure 4	CC Mesure 5	Moyenne
Pas de bruit	0.00452	0.00503	0.00197	0.00256	0.00500	0.00382
Note parfaite	0.68014	0.14206	0.75532	0.09502	0.78994	0.49250
Bruits ambients	0.27434	0.19877	0.18282	0.15311	0.31978	0.22577
Note parfaite sur bruits ambients	0.76683	0.68126	0.24605	0.66463	0.12143	0.55604

4 Hz	CC Mesure 1	CC Mesure 2	CC Mesure 3	CC Mesure 4	CC Mesure 5	Moyenne
Pas de bruit	0.00312	0.00294	0.00628	0.00265	0.00348	0.00369
Note parfaite	0.81321	0.69617	0.73204	0.65074	0.74836	0.72811
Bruits ambiants	0.28024	0.63126	0.19273	0.23315	0.40621	0.40727
Note parfaite sur bruits ambients	0.72051	0.80641	0.69521	0.86056	0.75103	0.76674

10 Hz	CC Mesure 1	CC Mesure 2	CC Mesure 3	CC Mesure 4	CC Mesure 5	Moyenne
Pas de bruit	0.00877	0.00392	0.01168	0.00771	0.00362	0.00714
Note parfaite	0.77748	0.82064	0.68023	0.78611	0.74021	0.76093
Bruits ambiants	0.23118	0.71421	0.32835	0.34377	0.79406	0.48231
Note parfaite sur bruits ambients	0.83861	0.85734	0.71544	0.85804	0.75103	0.80410

Dans tous les relevés, peu importe la fréquence, lorsqu'il n'y pas de bruit, le programme fonctionne correctement et le résultat de la corrélation croisée est proche de 0.

Les résultats expérimentaux quant aux autres situations sont plus intéressantes et nous éclairent sur plusieurs points. Passons en revue les résultats pour chaque fréquence avant de conclure :

- Pour 1 Hz

- Note parfaite : les valeurs ne sont pas très hautes et sont très dispersées. Il semblerait que l'analyse échoue à reconnaître certaines frappes, et qu'elle ne reconnaisse que partiellement les frappes « entendues ». Cela peut s'expliquer aisément à l'aide du modèle que nous avons utilisé : avec une précision de 1Hz, la durée du relevé audio doit être d'1s. Or le son émis par une frappe ne dure qu'environ 500ms. Comme nous l'avions expliqué plus haut, non seulement le flux audio entrant après une frappe se corrompt en se chargeant de données inutiles, mais de plus cela entraîne un zero-padding trop important sur les données audios de la frappe de référence stockées dans la base de données, ce qui là aussi corrompt la FFT résultante. De plus, il se peut que les données audios d'une frappe chevauchent deux relevées (par exemple les premières 200ms de la frappe à la fin d'un premier relevé d'1s, les 300ms suivantes au début du relevé suivant). De ce point de vue, le modèle que nous avons développé est confirmé.

- Bruits ambiants : les valeurs sont relativement basses, l'analyse ne fait pas d'erreurs ;
- Note parfaite sur bruits ambiants : là aussi les résultats sont chaotiques. On peut faire la même remarque que pour la note parfaite seule.

- Pour 4 Hz

- Note parfaite : les résultats sont hauts et relativement homogènes. La reconnaissance est donc bonne.

- Bruits ambients : les résultats sont hétérogènes, et certaines valeurs sont relativement hautes. Un bruit de fond suffisamment fort aux fréquences suffisamment proches de celles d'une frappe harmoniques pourrait entraîner un faux-positif, comme on peut le voir avec le 2ème relevé ;

- Note parfaite sur bruit ambiant : La-aussi les résultats sont hauts et homogènes. On voit que le bruit de fond influe toujours légèrement sur les mesures, mais rien de problématique.

- Pour 10 Hz

- Note parfaite : les résultats sont hauts et homogènes. La reconnaissance par algorithme paraît bonne.

- Bruits ambients : l'effet remarqué avec une précision de 4Hz est ici fortement accentué, de telle sorte que les bruits ambients puissent entraîner des faux-positifs fréquemment (presque un relevé sur deux ici).

- Note parfaite sur bruits ambients : même remarque que pour la fréquence 4Hz.

#### Écarts avec le cahier des charges :

Ainsi, les résultats expérimentaux nous permettent de conclure quant au cahier des charges : la précision au Hz près imposé par ce dernier n'est donc pas une contrainte retenue pour les raisons expliquées. En revanche, avec une précision fréquentielle de 4Hz, la contrainte du cahier des charges qui impose un résultat cohérent au moins une fois sur deux est respecté.

#### Écarts avec le modèle (simulé) :

De manière générale, les résultats expérimentaux concordent relativement bien avec le modèle utilisé. La fréquence optimale prévue de 4Hz s'est révélée la meilleure de celles testées. Comme prévu par le modèle, les imprécisions de calcul (nombre à virgule flottante etc.) se révèlent négligeable. Néanmoins, même avec une précision fréquentielle de 4Hz, il est possible avec des bruits environnants particuliers d'obtenir un faux-positif comme on peut le voir avec le deuxième relevé à cette fréquence.

#### 4) Communication avec le microcontrôleur

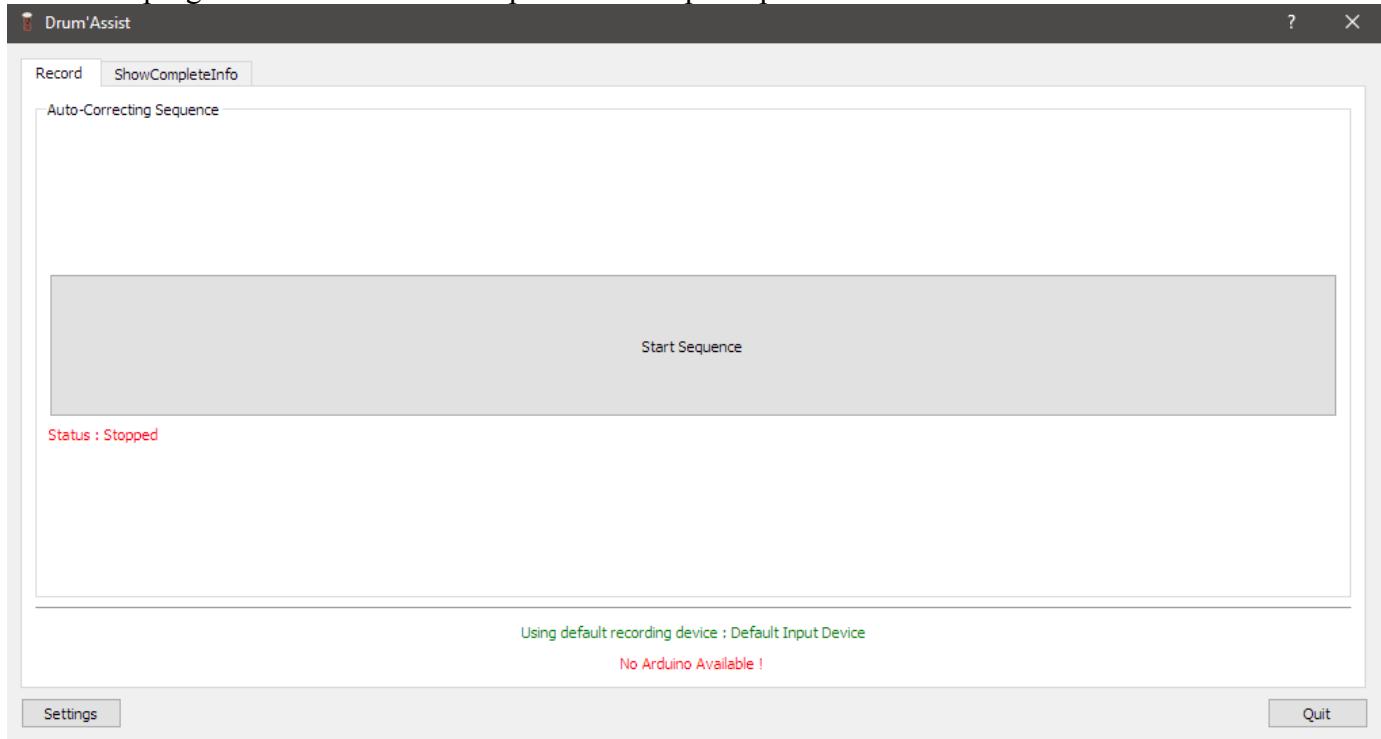
Une carte Arduino est utilisée dans notre projet pour contrôler le bras robotisé, mais aussi pour recevoir les informations résultant de l'analyse de la frappe et corriger le mouvement du bras en conséquence. La communication entre l'ordinateur et cette carte Arduino se fait tout simplement avec un câble. On utilise le port COM pour communiquer. On lui envoie à chaque frappe détectée un octet image de la correspondance avec une frappe harmonique : plus la frappe détectée est proche d'une frappe harmonique, plus la valeur envoyée est proche de 255, valeur entière maximal stockable sur un octet non signé (8 bits, soit  $2^8 = 256$  valeurs possibles, de 0 à 255).

*Le code pour gérer cette communication est en annexe.*

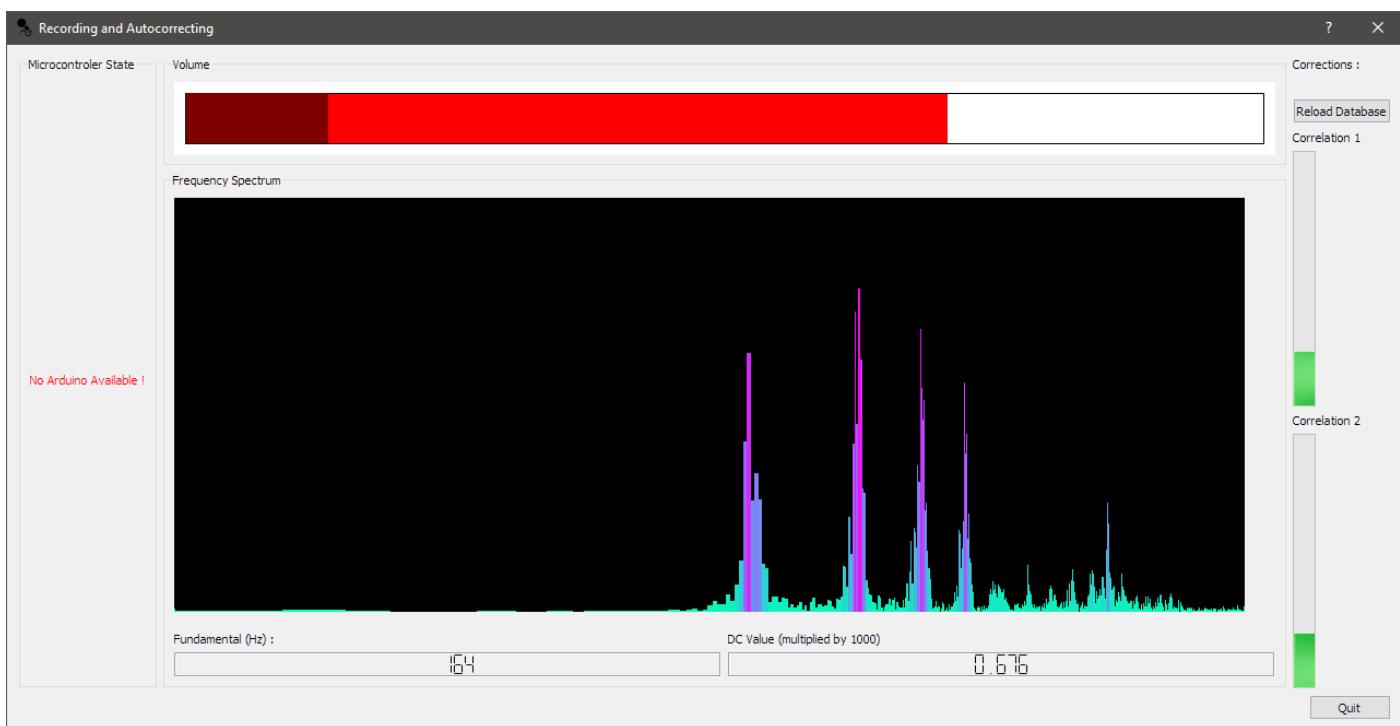
Ce type de communication est fiable (jusqu'ici il n'a jamais échoué) et le programme informatique reconnaît bien l'Arduino lorsqu'il est connecté à l'ordinateur. Le code écrit (simulation) est donc en accord avec le réel, et tous deux répondent positivement à la contrainte imposée par le cahier des charges, à savoir une communication ordinateur/Arduino réussie au moins 95 % du temps.

### 5) Présentation du programme final et de ses performances

Le programme final est très simple. Le menu principal est comme suit :



Les informations sont très claires. Une fois que l'utilisateur décide de lancer le processus d'écoute et de corrections en appuyant sur le bouton « Start Sequence » voici la fenêtre qui apparaît :



A partir de là, si les corrélations sont suffisamment élevées le programme enverra un message à l'Arduino ...

### Écarts sur les performances du programme

Théorique : le cahier des charges nous impose une utilisation processeur moyenne inférieure à 5%.

Simulé : nous pouvons estimer mathématiquement l'utilisation processeur du programme. Nous savons que la fréquence de rafraîchissement de la FFT est de 4 Hz et que l'on utilise une fréquence d'échantillonnage de 48kHz. Ainsi Il y aura  $48000/4 = 12000$  échantillons à analyser tous les 250ms. Il y a deux FFT à réaliser avec autant d'échantillons (la fréquence d'échantillonnage des relevés audios de la base de données est également de 48 kHz) soit très grossièrement  $2 \times 12000 \log_2(12000)$  opérations. On aura également 12000 multiplications avant le calcul de la FFT (application de la fonction de fenêtrage, on fait l'hypothèse qu'elle n'est calculée qu'une seule fois étant donné que ce sera toujours le même nombre d'échantillons à traiter), 2\*6000 multiplications et 6000 calcul de racine carrée à la fin de chaque FFT pour en faire le spectre fréquentiel et 2\*6000 multiplications et additions lors du calcul des deux corrélations croisées. Cela nous donne en tout :

$$2 \times 12000 \log_2(12000) + 12000 + 2 \times 6000 + 6000 + 2 \times 6000 \approx 367218 \text{ opérations.}$$

Autant le calcul d'une addition ne prend qu'un seul cycle processeur, le calcul d'une multiplication de 2 flottants peut en prendre jusqu'à une dizaine et celle d'une racine carrée plusieurs dizaines. Dans les faits, le nombre de cycles par opérations dépend du processeur mais également du niveau d'optimisation engagé sur le compilateur. Notre programme a été compilé avec les options -O2 et -fp:fast entre autres, ce qui permet une optimisation agressive. On majorera le nombre de cycles par opération par 7. Ainsi, les calculs (et seulement les calculs) vont utiliser approximativement :

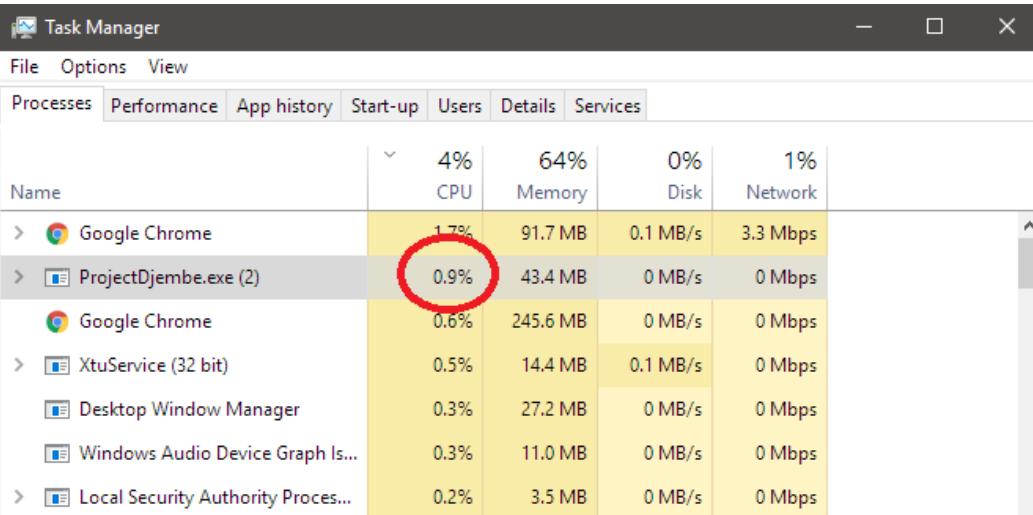
$$367218 \times 7 = 2570526 \text{ cycles.}$$

Le processeur sur mon ordinateur est un I7-6700HQ cadencé à 3GHz, soit 3 milliards de cycles par seconde. On considérera que le processeur n'effectue qu'une seule instruction par cycle (aucune optimisation de type SSE etc.). Ainsi, en une seconde (4 fois plus de temps) le calcul devrait prendre :

$$\frac{2570526 \times 4}{3000000000} = 0.003427 \text{ soit environ } \mathbf{0.34\% \text{ d'utilisation processeur.}}$$

Écart avec le cahier des charges : cela est bien inférieur au 5 % imposé par le cahier de charges. En effet, les calculs engagés ne sont pas si gourmands en ressources.

Réel : il suffit d'ouvrir le gestionnaire des tâches lors de l'exécution du programme pour savoir quelle est son utilisation processeur.



Name	CPU	Memory	Disk	Network
> Google Chrome	1.7%	91.7 MB	0.1 MB/s	3.3 Mbps
> ProjectDjembe.exe (2)	0.9%	43.4 MB	0 MB/s	0 Mbps
Google Chrome	0.6%	245.6 MB	0 MB/s	0 Mbps
> XtuService (32 bit)	0.5%	14.4 MB	0.1 MB/s	0 Mbps
Desktop Window Manager	0.3%	27.2 MB	0 MB/s	0 Mbps
Windows Audio Device Graph Is...	0.3%	11.0 MB	0 MB/s	0 Mbps
> Local Security Authority Proces...	0.2%	3.5 MB	0 MB/s	0 Mbps

On voit que le programme utilise **0.9% du processeur**.

Écart avec le cahier des charges : cela est inférieur à la limite imposée par le cahier des charges (qui est de 5%). Après tout cela est logique, le programme est loin d'engager des calculs d'une complexité extraordinaire, le processeur est puissant, et de plus le programme a été écrit en C++, langage qui permet une bonne vitesse d'exécution et la mise en place d'optimisations puissantes. Bien que je sois loin de maîtriser ce langage, il m'a quand même été possible de mettre en place un programme relativement rapide et peu gourmand en ressources.

Écart avec le modèle simulé : notre modèle simulé prévoyait une utilisation de **0.34 % du processeur**. Il se révèle incorrect pratiquement par un facteur 3. Une partie de cet écart pourrait s'expliquer par le fait que nous n'avons pas pris en compte l'ensemble des calculs à effectuer par le processeur. En réalité, cet écart s'explique principalement par un facteur extrêmement important que nous n'avons pas pris en compte : **le temps mis par le processeur pour accéder à la mémoire pour lire ou écrire des données**. En effet, aller chercher des données stockées directement dans la RAM (mémoire vive) est un processus extrêmement long (le processeur peut passer 200-300 cycles à attendre ces données). Heureusement il existe tout un système de « mémoire cache » intégré au processeur qui remédie partiellement à ce problème et il aura fallu faire attention dans le code à mettre à profit cette technologie (principalement en disposant les données à étudier dans des cases mémoires adjacentes et en les parcourant linéairement).

Les optimisations liées à l'utilisation de données stockées en mémoire et leurs interactions avec la mémoire cache sont aujourd'hui les principaux facteurs influant la vitesse d'exécution dans une très grande partie des programmes informatiques.

## **IV) Présentation du système final**

Le système final, dans son ensemble, fonctionne correctement. En voici une photo :



Le système reconnaît les frappes suffisamment fréquemment comme l'ont montré les études, et est en mesure de se repositionner automatiquement pour pouvoir montrer la frappe harmonique parfaite.

Le système a été présenté aux **Olympiades des Sciences de l'Ingénieur**.

## **Bibliographie**

<https://en.wikipedia.org/wiki/Cross-correlation>

[https://fr.wikipedia.org/wiki/Transformation\\_de\\_Fourier\\_discr%C3%A8te](https://fr.wikipedia.org/wiki/Transformation_de_Fourier_discr%C3%A8te)

[https://fr.wikipedia.org/wiki/Transformation\\_de\\_Fourier\\_rapide](https://fr.wikipedia.org/wiki/Transformation_de_Fourier_rapide)

[https://en.wikipedia.org/wiki/Fast\\_Fourier\\_transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform)

[https://en.wikipedia.org/wiki/Discrete\\_Fourier\\_transform](https://en.wikipedia.org/wiki/Discrete_Fourier_transform)

<https://fr.wikipedia.org/wiki/Fen%C3%AAtrage>

[https://en.wikipedia.org/wiki/Window\\_function](https://en.wikipedia.org/wiki/Window_function)

[https://en.wikipedia.org/wiki/Hann\\_function](https://en.wikipedia.org/wiki/Hann_function)

<http://www.fftw.org/>

<http://www.drdobbs.com/cpp/a-simple-and-efficient-fft-implementatio/199500857>

<https://www.codeproject.com/Articles/9388/How-to-implement-the-FFT-algorithm>

[https://www.dsprelated.com/freebooks/mdft/Zero\\_Padding.html](https://www.dsprelated.com/freebooks/mdft/Zero_Padding.html)

<https://www.zoom-na.com/fr/products/enregistrement-terrain-vidéo/enregistrement-de-terrain/zoom-h2n-handy-recorder>

<http://www.ni.com/white-paper/4844/fr/>

[https://en.wikipedia.org/wiki/Spectral\\_leakage](https://en.wikipedia.org/wiki/Spectral_leakage)

[https://fr.wikipedia.org/wiki/Epsilon\\_d'une\\_machine](https://fr.wikipedia.org/wiki/Epsilon_d'une_machine)

<http://calcul.math.cnrs.fr/IMG/pdf/20120207-binet-opt-cxx.pdf>

[http://www.agner.org/optimize/optimizing\\_cpp.pdf](http://www.agner.org/optimize/optimizing_cpp.pdf)

<http://deveniringeson.com/les-caracteristiques-du-micro/>

<http://www.audio-technica.com/cms/site/e4b90aaa592b9d1a/index.html>

<http://doc.qt.io/qt-5/classes.html>

<https://www.sfml-dev.org/tutorials/2.4/#graphics-module>

## ANNEXES

### Générateur de Ton

```

#define M_PI 3.14159265358979323846
typedef float d_real;

/** Fonction donnant la valeur de l'onde sinusoïdale de fréquence frequency pour le point t (temps en s)
 * t peut être obtenu en divisant le rang de l'échantillon par la fréquence d'échantillonage.
 */
inline d_real sinFrequency(d_real frequency, d_real t) {
    return sin(2 * M_PI*frequency*t);
}

/** Cette fonction génère une onde sinusoïdale d'amplitude "amplitude"
 * et dont la fréquence croît linéairement de initialFrequency à finalFrequency.
 * Elle est ensuite stocké dans le tableau pointé par data.
 * Les autres arguments nous donnent les informations complémentaires nous permettant
 * de savoir comment stocker ces données :
 * simuler une fréquence d'échantillonage sampleRate sur bytesToWrite octets.
 */
template<typename T>
bool toneGen(T* data, unsigned int bytesToWrite, unsigned int sampleRate,           // Où et comment écrire le ton générés
            d_real initialFrequency, d_real finalFrequency, d_real amplitude)        // Le ton à générer
{
    // On vérifie que le nombre d'octets à écrire est bien multiple du type de donnée utilisé
    if (bytesToWrite % sizeof(T) != 0) {
        return false;
    }
    // Le nombre d'échantillon à écrire
    int samples = bytesToWrite / sizeof(T);
    d_real changingFrequency = initialFrequency;
    d_real shift = (finalFrequency - initialFrequency) / samples;
    for (int i = 0; i < samples; i++) {
        changingFrequency += shift;
        // Ecrire le ton généré dans data
        data[i] = static_cast<T>(sinFrequency(changingFrequency, (d_real)i / (d_real)sampleRate)
                                  * amplitude
                                  * (std::numeric_limits<T>::max)());
    }
    return true;
}

```

**Code pour gérer l'Arduino**Arduino.h

```
#ifndef ARDUINOMANAGER_H
#define ARDUINOMANAGER_H

#include <QObject>
#include <QSerialPort>
#include <QSerialPortInfo>
#include <QDebug>
#define ARDUINO_DEBUG

class ArduinoManager : public QObject
{
    Q_OBJECT
public:
    ArduinoManager();
    ~ArduinoManager();

    bool initialize();
    bool scanForArduino();
    bool configureSerialPort();
    bool checkSerialPortError();
    bool isAvailable() { return m_arduino_is_available; }

public slots :
    bool writeByteToArduino(char);

signals :
    void arduinoPortName(QString) const;

private :
    static const quint16    m_arduino_uno_vendor_id = 9025;
    static const quint16    m_arduino_uno_product_id = 66;
    QSerialPort*           m_arduino;
    QString                m_arduino_port_name;
    bool                   m_arduino_is_available;
};

#endif // ARDUINOMANAGER_H
```

Arduino.cpp

```
#include "arduinomanager.h"

ArduinoManager::ArduinoManager()
    : m_arduino_is_available(false)
    , m_arduino_port_name("")
    , m_arduino(new QSerialPort)
{
#ifdef ARDUINO_DEBUG
    qWarning() << "Number of available ports: " << QSerialPortInfo::availablePorts().length();
#endif
}

ArduinoManager::~ArduinoManager() {
    m_arduino->close();
    delete m_arduino;
}

bool ArduinoManager::initialize(){
    if (scanForArduino())
        if(configureSerialPort())
            return true;

    return false;
}
```

```

bool ArduinoManager::scanForArduino() {
    m_arduino_is_available = false;
    foreach(const QSerialPortInfo &serialPortInfo, QSerialPortInfo::availablePorts()){
        if(serialPortInfo.hasVendorIdentifier() && serialPortInfo.hasProductIdentifier()){
#ifdef ARDUINO_DEBUG
            qWarning() << "Serial port found. Specs :"
            << "Vendor ID : " << serialPortInfo.vendorIdentifier()
            << ".Product ID :" << serialPortInfo.productIdentifier();
#endif
    }
    if(serialPortInfo.vendorIdentifier() == m_arduino_uno_vendor_id){
        if(serialPortInfo.productIdentifier() == m_arduino_uno_product_id){
            m_arduino_port_name = serialPortInfo.portName();
            emit arduinoPortName(m_arduino_port_name);
            m_arduino_is_available = true;
            return true;
        }
    }
}
emit arduinoPortName("<font color='red'>No Arduino Available !</font color>");
return false;
}

bool ArduinoManager::configureSerialPort() {
    if(m_arduino_is_available){
        // open and configure the serialport
        m_arduino->setPortName(m_arduino_port_name);
        m_arduino->open(QSerialPort::ReadWrite);
        m_arduino->setBaudRate(QSerialPort::Baud9600);
        m_arduino->setDataBits(QSerialPort::Data8);
        m_arduino->setParity(QSerialPort::NoParity);
        m_arduino->setStopBits(QSerialPort::OneStop);
        m_arduino->setFlowControl(QSerialPort::NoFlowControl);
        return true;
    }
    return false;
}

bool ArduinoManager::checkSerialPortError(){
    if (m_arduino->error()) {
#ifdef ARDUINO_DEBUG
        qWarning() << "An error occurred during serial port operation. Error code :" << m_arduino->error();
#endif
        m_arduino->close();
        emit arduinoPortName("<font color='red'>No Arduino connected !</font color>");
        return true;
    }
    return false;
}

bool ArduinoManager::writeByteToArduino(char value) {
    // Check if there is an error code (port disconnected for instance...)
    if(checkSerialPortError()){
        qWarning() << "Serial Port Error : Arduino not connected or Special Error";
        return false;
    }

    if (m_arduino->isOpen() && m_arduino->isWritable()) {
#ifdef ARDUINO_DEBUG
        qWarning() << "Arduino is open and writable.";
        qWarning() << "Writing" << QString::number(value) << "to the Arduino";
#endif
        m_arduino->write(&value, 1);
        if (m_arduino->bytesToWrite() > 0) {
            if(m_arduino->flush()) {
#ifdef ARDUINO_DEBUG
                qWarning() << "Data has been written to Arduino";
#endif
            }
        }
        return true;
    }
    return false;
}

```

## Code pour l'Analyse

### Zero-padding

```
// Un QByteArray est un tableau de byte, i.e des char codé sur un octet
inline int zeroPad(QByteArray& tab, int totalBytes) {
    int bytesToadd = totalBytes - tab.size();
    // On vérifie que le nombre d'octet à ajouter est supérieur à 0
    if (bytesToadd > 0)
        tab.append(bytesToadd, 0); // On ajoute le nombre de 0 nécessaire
    return tab.size(); // la valeur de retour est la taille nouvelle du QByteArray
}
```

### Puissance de 2 supérieure la plus proche d'un entier

```
// Entier non signé
template<typename T>
unsigned long long next_power_of_2(T value, typename std::enable_if<std::is_unsigned<T>::value && std::is_integral<T>::value, bool>::type* = 0) {
    for (int i = sizeof(T)*8 - 1; i >= 0; i--) {
        if (value & (T)1<<i) {
            if (value > (T)1<<i)
                return 1ull << (i+1ull);
            else {
                return (1ull << i);
            }
        }
    }
    // Value is 0 : return 2^(-max) =approx 0.
    return 0;
}

// Entier signé
template<typename T>
unsigned long long next_power_of_2(T value, typename std::enable_if<std::is_signed<T>::value && std::is_integral<T>::value, bool>::type* = 0) {
    if (value > 0) {
        for (int i = sizeof(T)*8 - 1; i >= 0; i--) {
            if (value & (T)1<<i) {
                if (value > (T)1<<i)
                    return 1ull << (i+1ull);
                else {
                    return (1ull << i);
                }
            }
        }
    }
    return 0;
}
```

### Puissance de 2 supérieure la plus proche d'un nombre

```
// double
template<typename T>
T next_power_of_2(T value, typename std::enable_if<std::is_floating_point<T>::value, bool>::type* = 0) {
    if (value > 0) {
        return pow(2.0, ceil(log2(value)));
    }
    return 0;
}
```

### Puissance de 2 la plus proche d'un nombre

```
inline unsigned long long closest_power_of_2(double value) {
    unsigned long long up = next_power_of_2(value);
    unsigned long long down = pow(2.0, floor(log2(value)));
    return (up - value) < (value - down) ? up : down;
}
```

### Calcul de la FFT

```

bool AudioInfo::performFrequencySpectrumCalculation(){
    const int tailleEchantillon = 2; // 16 bits is 2 bytes
    const int nbEchantillons = (m_dataBytes/tailleEchantillon); // nombre d'échantillons

    // Calcul de la fonction de fenêtrage, stockée dans m_window qui est un vector<float>
    m_window.resize(nbEchantillons);
    for (int i=0; i< nbEchantillons; ++i) {
        m_window[i] = static_cast<float>(0.5f * (1.0f - cosf(static_cast<float>(2.0f * M_PI * i) / static_cast<float>(nbEchantillons - 1))));
    }

    // in et out sont des fftwf_complex*
    in = (float*) fftwf_malloc(sizeof(float)*nbEchantillons);
    out = (fftwf_complex*) fftwf_malloc(sizeof(fftwf_complex)*nbEchantillons);

    // pointeur vers les données audios
    const qint8* ptr = reinterpret_cast<const qint8*>(m_data);

    for (int i = 0; i < nbEchantillons; i++){
        // Appliquer la fonction de fenêtrage
        in[i] = *reinterpret_cast<const qint16*>(ptr) * m_window[i];
        ptr += tailleEchantillon;
    }

    my_plan = fftwf_plan_dft_r2c_1d(nbEchantillons, in, out, FFTW_ESTIMATE);
    fftwf_execute(my_plan);

    // Mettre à l'échelle les résultats
    float facteurReduction = 2.0f/(float)nbEchantillons;
    facteurReduction *= (1.0f/0.5f); // Du fait de la fenêtre de Hann utilisée

    float valeur = 0;
    // On stocke le spectre fréquentiel dans m_frequencySpectrum. Il s'agit d'un vector<float>
    m_frequencySpectrum.resize(nbEchantillons/2);
    for (int i = 0; i < nbEchantillons/2; i++){
        valeur = sqrtf(out[i][0]*out[i][0] + out[i][1]*out[i][1])*facteurReduction/std::numeric_limits<qint16>::max();
        m_frequencySpectrum[i] = valeur;
    }

    fftwf_free(in);
    fftwf_free(out);
    fftwf_destroy_plan(my_plan);

    return true;
}

```

### Calcul de la corrélation croisée

```

// Faire la corrélation croisée
// Les deux FFT, de type vector<float>, sont :
// "databaseFFT" pour celle correspondant à la base de donnée
// "fs" pour celle correspondant au flux audio entrant
// On s'est assuré qu'ils soient de même taille etc.
int elements = fs.size();
float correlationCroisee = 0.0f;
for (int decalage = -1; decalage <= 1; decalage++) {
    float valeur = 0.0f;
    for (int i = 1; i < elements - 1; i++) {
        valeur += fs[i] * databaseFFT[i + decalage];
    }
    if (valeur > correlationCroisee) correlationCroisee = valeur;
}
// La valeur de la correlation croisée est maintenant
// stockée dans la variable correlationCroisee.

```

Lien vers une version récente du code :

<https://www.dropbox.com/s/cefta2brbl6ksdo/Project%20Djembe%20Nouvelle%20version%20%28Code%20seul%29.zip?dl=0>