

PROJET Surveillance sonore P2

Module : Chaine d'acquisition et traitement numérique 2

Sommaire

Sommaire	1
1. Organisation du code (analyse)	1
2. Configuration et test de l'écran Oled	3
3. Réception des données audio	3
4. Traitement du signal audio	8



⚠️ La notation de cette activité sera dépendante de l'historique de vos commits ⚠️

Les commentaires dans les fichiers d'entêtes seront au format Doxygen

1. Organisation du code (analyse)

Vous utiliserez la POO pour structurer le code. Vous suivrez l'organisation du diagramme de classe présenté Figure 1.

- En bleu clair, les classes des bibliothèques nécessaires au programme.
- En jaune, les classes à créer.
- En rouge le programme principal.

Description des classes à réaliser :

- **CSurvSon** n'est pas une classe, c'est le programme principal, composé sous Arduino de la fonction **setup** et **loop**. Les compositions sur **CSurvSon** seront ici des instantiations globales dans le fichier source principal.
- **CCom** est la classe qui est chargée de la communication des informations via http en wifi.
- **CSon** est la classe qui est chargé d'acquérir les échantillons sonores et de faire les traitements mathématiques.

Dans cette première version, afin d'alléger le code, l'encapsulation est limitée.

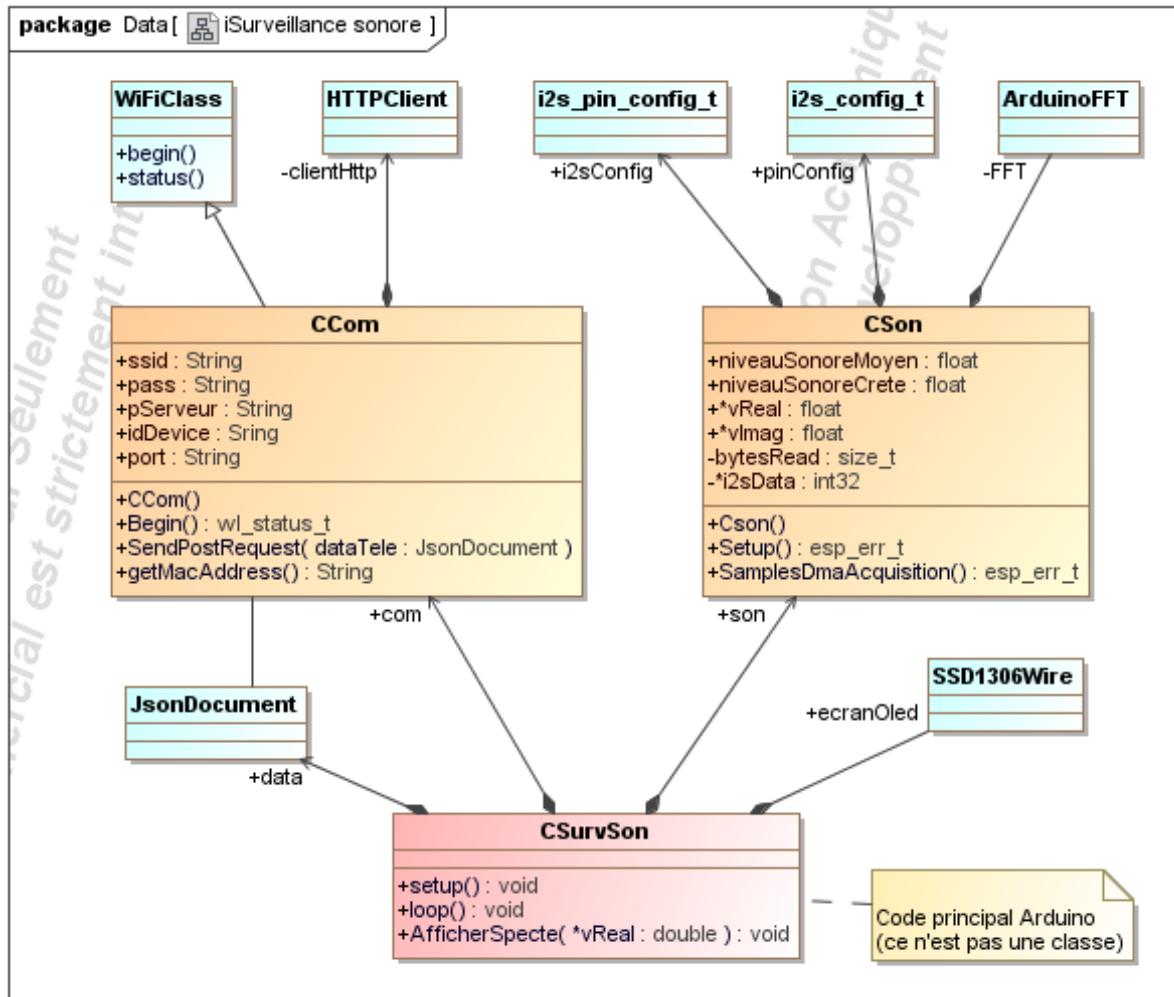


Figure 1 Diagramme de classe du programme de surveillance sonore

☛ Que représente en POO les symboles et noms suivants :

- La flèche avec un losange plein :
composition
- La flèche avec un triangle blanc :
héritage/généralisation
- CSon :
Nom de la classe
- CCom() :
constructeur de la classe CCom
- +data :
member public de la classe



Autoévaluation

2. Configuration et test de l'écran Oled

- ⌚ Inclure les fichiers d'entêtes `Wire.h` et `SSD1306.h`
- ⌚ Instancier un objet `ecranOled` de classe `SSD1306Wire` :
`SSD1306 ecranOled(0x3C, 5, 4);`
- ⌚ Quel est le protocole utilisé entre le µC et l'afficheur ? A quoi correspondent les paramètres du constructeur ?

le protocole utilisé est I2C
Le constructeur définit l'adresse I2C (0x3C), la broche SDA (5) et la broche SCL (4) pour communiquer avec l'écran OLED

- ⌚ Dans le `setup`, initialiser l'écran et afficher une présentation avec un texte indiquant le nom de l'application et sa version. Utiliser `init`, `clear`, `setFont`, `drawString`, `display`. Donner le code ci-dessous.

```
void setup()
{
    ecranOled.init();
    ecranOled.clear();
    ecranOled.setFont(ArialMT_Plain_16);
    ecranOled.drawString(0, 10, "Mon Appli v1.0");
    ecranOled.display();
}
```



3. Réception des données audio

3.1.Utilisation du DMA

Voir doc ESP : https://docs.espressif.com/projects/esp-idf/en/v5.0.1/esp32/api-reference/system/mem_alloc.html?highlight=dma

Quel est le principe et l'intérêt du DMA ?

Le DMA permet aux périphériques d'accéder directement à la mémoire sans passer par le processeur, ce qui accélère les transferts de données et libère les ressources du système.

3.2.Création de la classe CSon

- ☛ Créer le fichier d'entête pour la classe CSon.h et y inclure les fichiers d'entête :

```
#include "arduinoFFT.h"  
#include <driver/i2s.h>
```

- ☛ Des constantes seront définies en constantes de préprocesseur dans ce fichier d'entête :

```
#define SAMPLES 512           // Nombre d'échantillons  
#define SAMPLING_FREQUENCY 44100 // Fréquence d'échantillonnage  
#define DMA_BUF_LEN 512         // Taille du buffer DMA : 512 échantillons  
#define DMA_BUF_COUNT 8          // Nombre de buffers DMA : 8
```

- ☛ Coder le prototype de la classe dans le fichier d'entête CSon.h tel qu'elle est modélisée dans le diagramme de classe.

i2sData est un tableau d'entier sur 32bits de taille fixe égale à SAMPLES

- ☛ Ecrire les commentaires en Doxygen.



3.3. Initialisation I2S et FFT

Dans le constructeur de CSon, on initialise les attributs de la classe.

- ☛ Pour le paramétrage des entrées/sorties, recopier le code en ajoutant les commentaires précisant les signaux concernés. Ici, pinConfig est une structure que l'on initialise.

```
this->pinCConfig = {  
    .bck_io_num = 14,    // ?  
    .ws_io_num = 13,    // ?  
    .data_out_num = I2S_PIN_NO_CHANGE, // Pas de data out  
    .data_in_num = 12   // ?  
};
```

- ☛ Pour le paramétrage du bus i2S, recopier le code en complétant les commentaires.

```
this->i2sConfig = {  
    .mode = i2s_mode_t(I2S_MODE_MASTER | I2S_MODE_RX), // ?  
    .sample_rate = SAMPLING_FREQUENCY, // ?  
    .bits_per_sample = I2S_BITS_PER_SAMPLE_32BIT, // Bits par échantillon  
    .channel_format = I2S_CHANNEL_FMT_ONLY_RIGHT, // ?  
    .communication_format = i2s_comm_format_t(I2S_COMM_FORMAT_STAND_I2S),  
    .intr_alloc_flags = 0, //ESP_INTR_FLAG_LEVEL1, // Niveau d'interruption  
    .dma_buf_count = 8, // Nombre de buffers DMA  
    .dma_buf_len = DMA_BUF_LEN, // Taille du buffer DMA  
    .use_apll = false,  
    .tx_desc_auto_clear = false,  
    .fixed_mclk = 0  
};
```

- ☛ Pour l'initialisation FFT, recopier le code.

```
this->FFT = ArduinoFFT<double>(this->vReal, this->vImag, SAMPLES, SAMPLING_FREQUENCY);
```

Commit #3

3.4. Setup I2S

Dans la méthode `CSon::Setup()`, des fonctions de la bibliothèque i2s vont être appelées pour configurer le driver.

- ☛ Ajouter le code suivant pour la définition de cette méthode :

```
result = i2s_driver_install(I2S_NUM_0, &this->i2sConfig, 0, NULL);
result = i2s_set_pin(I2S_NUM_0, &this->pinConfig);
result = i2s_zero_dma_buffer(I2S_NUM_0);
return result;
```

- ☛ `result` sera ajouter en attribut public de la classe `CSon`
- ☛ Quel est le type de `result` et quelles sont les valeurs qu'il peut prendre en retour de `CSon::Setup()` ?

```
float result = i2s_driver_install(I2S_NUM_0, &this->i2sConfig, 0, NULL);
result = i2s_set_pin(I2S_NUM_0, &this->pinConfig);
result = i2s_zero_dma_buffer(I2S_NUM_0);
return result;
```

Commit #4

3.5.Acquisition en DMA d'un échantillon

La fonction de la bibliothèque i2s utilisée pour récupérer en mémoire DMA une partie des données reçues sur le bus i2s est `i2s_read`.

- ☛ Décrire les paramètres et la valeur de retour de cette fonction en indiquant s'il s'agit d'une entrée ou d'une sortie (ou les deux) :

Paramètres :

Port I2S : Identifie le port I2S utilisé (entrée).

Buffer de destination : Pointeur vers la mémoire où les données seront stockées (sortie).

Taille du buffer : Nombre de bytes à lire (entrée).

Bytes lus : Pointeur vers une variable où sera stocké le nombre de bytes effectivement lus (sortie).

Timeout : Temps maximal d'attente pour l'opération (entrée).

Valeur de retour :

La fonction retourne un code d'erreur indiquant si l'opération a réussi ou échoué. Par exemple :

`ESP_OK` : Succès.

`SD / BTS CIEL`

Diffusion non autorisée

`ESP FAIL` : Échec.

- ☛ Recopier l'implémentation de la méthode `SamplesDmaAcquisition` ci-dessous et vérifier qu'il n'y ai pas d'erreur de compilation.

```
esp_err_t CSon::SamplesDmaAcquisition()
{
    // Nombre d'octets lues en mémoire DMA
    size_t bytesRead;
    // Capture des données audio
    result = i2s_read(I2S_NUM_0, &this->i2sData, sizeof(this->i2sData), &bytesRead,
portMAX_DELAY);

    if (result == ESP_OK)
    {
        int16_t samplesRead = bytesRead/ 4; // ?
        if (samplesRead > 0)
        {
            float mean = 0;
            for (int16_t i = 0; i < samplesRead; ++i)
            {
                i2sData[i]= i2sData[i]>> 8; // ?
                mean += abs(i2sData[i]);
                if (abs(i2sData[i])>niveauSonoreCrete) niveauSonoreCrete=abs(i2sData[i]);
            }
            this->niveauSonoreMoyen = mean/samplesRead; // ?
        }
    }
    return result;
}
```

- ☛ Compléter les commentaires du code (`// ?`).



3.6. Tests de l'acquisition du son

- ☛ Dans le programme principal, instancier l'attribut son.
- ☛ Dans la fonction `setup`, appeler la méthode `Setup` de `CSon`.
- ☛ Dans la fonction `loop`, appeler la méthode `SamplesDmaAcquisition` puis afficher les valeurs de `niveauSonoreMoyen` et de `niveauSonoreCrete`.
- ☛ Tester



3.7.Affichage sur Serial Plotter du signal

- ➲ Donner les lignes de code permettant d'envoyer sur le port série les valeurs du niveau sonore moyen et crête, en utilisant une chaîne de caractère permettant à Serial Plotter de tracer les courbes de ces niveaux.

```
void loop()
{
    son.SamplesDmaAcquisition();
    Serial.print("nvSonoreMoyen: ");
    Serial.println(son.niveauSonoreMoyen);
    sleep(1);
    Serial.print("nvSonoreCrete: ");
    Serial.println(son.niveauSonoreCrete);
    sleep(1);
}
```

- ➲ Envoyer ces données sur le port série à chaque lecture en mémoire.
- ➲ Relever les courbes.

```
nvSonoreMoyen: 107686.70
nvSonoreCrete: 299443.00
nvSonoreMoyen: 34952.08
nvSonoreCrete: 299443.00
nvSonoreMoyen: 20364.94
nvSonoreCrete: 299443.00
nvSonoreMoyen: 3148.78
nvSonoreCrete: 299443.00
nvSonoreMoyen: 1080.30
nvSonoreCrete: 299443.00
```



Commit #7



Autoévaluation

4. Traitement du signal audio

4.1.Détermination du niveau sonore

La **valeur efficace** du signal sonore permet d'avoir une **représentation de l'énergie** de ce signal qui est proche du **volume sonore ressentie** par l'oreille humaine.

- Quelle est la formule qui donne la valeur efficace d'un signal périodique temporel ?

- En électricité, c'est la valeur efficace du courant qui est utilisée pour obtenir la puissance moyenne dissipée dans une résistance. Retrouver la formule dans le cas d'un signal périodique.

Pour un signal discret, la valeur efficace RMS (Root Mean Square) se calcule par la formule suivante :

$$RMS = \sqrt{\frac{1}{N} \sum_{i=1}^N x_i^2}$$

Pour réduire la charge de calcul, on va calculer la valeur moyenne du signal redressé :

$$Moyenne = \frac{1}{N} \sum_{i=1}^N |x_i|$$

La moyenne du signal redressé est plus rapide à calculer mais elle sous-estime les valeurs comparées au calcul RMS, car la méthode RMS pondère plus fortement les pics.

Pour un signal sinusoïdal on a :

$$\begin{aligned} \text{Valeur Moyenne : } & V_{moy} = 2/\pi V_{max} = 0.637V_{max} \\ \text{Valeur efficace } & V_{eff} = V_{max}/\sqrt{2} = 0.707V_{max} = \pi/2\sqrt{2}V_{moy} = 1.11.V_{moy} \end{aligned}$$

Pour un signal carré on a

$$\begin{aligned} \text{Valeur Moyenne : } & V_{moy} = V_{max} \\ \text{Valeur efficace } & V_{eff} = V_{max} = V_{moy} \end{aligned}$$

Dans notre application, nous n'avons pas besoin d'une mesure juste de la puissance sonore, cette approximation sera donc utilisée.

4.2.Adaptation de la période de moyennage

- Précédemment avec la méthode **SamplesDmaAcquisition**, sur quel intervalle de temps est calculé la moyenne du signal redressé et la valeur crête ? Quelle est la fréquence la plus basse que l'on peut intégrer dans la moyenne sur cette période ?

Pour l'application désirée, le niveau sonore moyen et crête sera évalué sur des temps de l'ordre de plusieurs secondes. Pour cela, dans une itération dans la fonction loop, on calculera un nouvelle moyenne `niveauSonoreMoy` à partir de celle fourni par `SamplesDmaAcquisition`.

Pour la valeur crête, il suffira de faire la mise à zéro de `niveauSonoreCrete` avant l'itération.

- ⌚ Ajouter la constante de préprocesseur dans main.cpp :

```
define PERIODE_RELEVE 10000 // période relevé et envoi en ms
```

- ⌚ Ajouter les variables suivantes dans loop :

```
float niveauSonoreMoy=0;  
int periodeReleve = PERIODE_RELEVE/son.tempsEchantillon;
```

- ⌚ Ajouter `tempsEchantillon` en attribut dans Cson et lui donner la valeur correcte dans le constructeur.

- ⌚ Tester le fonctionnement.



Commit #8



Autoévaluation

4.1. Mesure des niveaux sonores en échelle logarithmique

Le **niveau de pression acoustique (SPL)** est le niveau de pression d'un son, mesuré en décibels (dB). Il est égal à $20 \times \log_{10}$ du rapport de la valeur RMS de la pression acoustique à la référence de la pression acoustique.

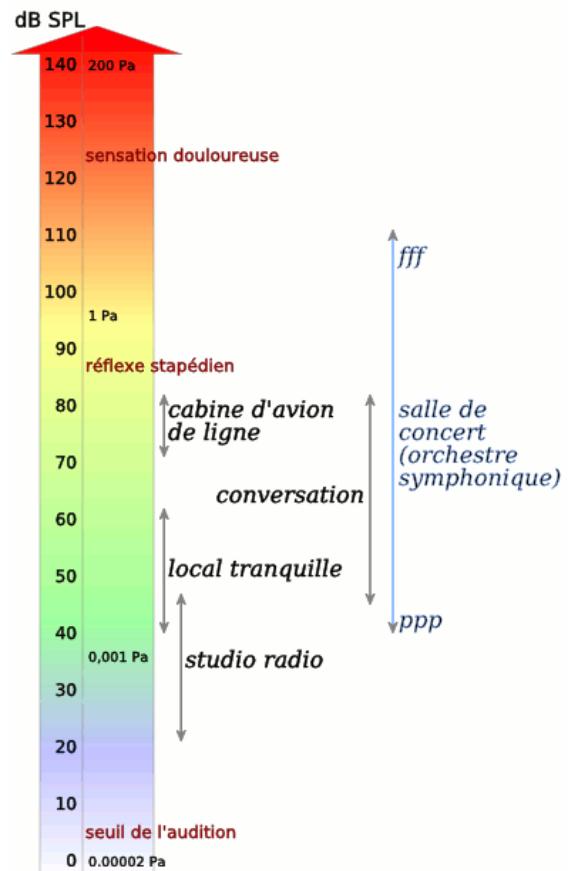
La pression acoustique de référence dans l'air est de 2×10^{-5} N/m² soit **20 µPa**. Ce qui correspond au seuil de l'audition humaine, **0 dB SPL**

80dB → dommage de l'oreille suivant le temps d'exposition (seuil d'activation de la protection par le muscle stapédiien)
94 dB SPL → pression acoustique de 1 Pa.

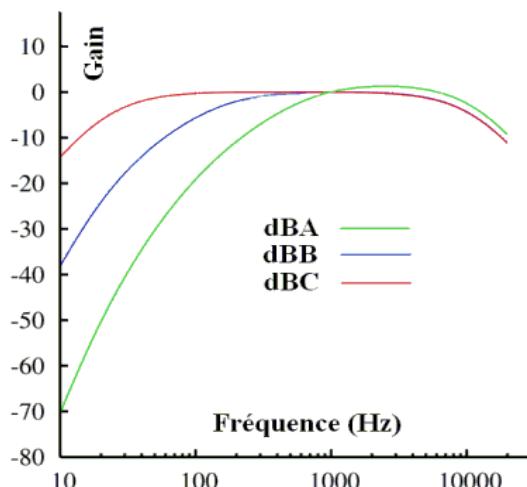
191dB SPL → niveau maximum (au niveau de la mer)
120 et 130 dB → Douleur

[https://fr.wikipedia.org/wiki/D%C3%A9cibel_\(bruit\)](https://fr.wikipedia.org/wiki/D%C3%A9cibel_(bruit))

<https://culturesciencesphysique.ens-lyon.fr/ressource/QSDecibelsSol.xml>



Des **pondérations** fréquentielles existent pour tenir compte de la sensibilité de l'oreille humaine aux différentes fréquences : **dBA**, **dBB** et **dBC**.



La **sensibilité** du microphone s'exprime en **dBFS** (décibels pleines échelles numériques) pour un niveau de pression acoustique de **94 dB SPL** soit 1 Pascal.

➊ Montrer que 1 Pascal correspond bien 94dB :

0 dBFS correspond à la valeur maximale possible du signal numérique pour une onde sinusoïdale, soit la valeur $2^{23} - 1$ pour un échantillonnage en 24 bits.

L'INMP441 a une **sensibilité** typique d'environ **-26 dBFS** c'est-à-dire qu'à 0dBFS on a $94+26 = 120$ dB SPL, ce qui est bien la valeur donnée dans les caractéristiques comme la valeur maximum (Maximum Acoustic Input).

Dans la documentation du microphone, il est indiqué que :

*An acoustic input signal of a 1 kHz sine wave **at 94 dB SPL** applied to the INMP441 results in an output signal with a **-26 dBFS level**. This means that the output digital word peaks at -26 dB below the digital full-scale level.*

*A common misunderstanding is that the output has an RMS level of **-29 dBFS**; however, this is not the case because of the definition of a 0 dBFS sine wave.*

⇒ Pourquoi as-t-on une différence de 3 dB entre la bonne et la mauvaise interprétation du dBFS ?

⇒ Ajouter dans loop les variables suivantes :

```
float niveauSonoreMoyenDB=0;  
float niveauSonoreCreteDB=0;
```

⇒ Actualiser ces variables en utilisant les formules suivantes :

```
niveauSonoreMoyenDB = 20 * log10( niveauSonoreMoy )-14.56 ;  
niveauSonoreCreteDB = 20 * log10( son.niveauSonoreCrete )-18.474 ;
```

⇒ Modifier le code pour afficher ces nouvelles variables



4.2.Analyse fréquentielle

On va utiliser ici la transformé de Fourier rapide (FFT) pour obtenir le spectre du signal.

⇒ Dans CSon, inclure l'entête `#include "arduinoFFT.h"`

⇒ La FFT réalise les calcul sur des nombres complexes, d'où les deux attributs tableaux de CSon :

```
/// @brief FFT Partie réelle  
double vReal[SAMPLES];  
/// @brief FFT Partie imaginaire  
double vImag[SAMPLES];
```

- ⇒ Instancier dans le constructeur de CSon, l'attribut `FFT` (`ArduinoFFT<double> FFT;`) en utilisant les deux attributs précédent, `SAMPLES` et `SAMPLING_FREQUENCY`

Dans `SamplesDmaAcquisition`, on remplit le tableau `VReal` par les acquisitions et on mets à 0 `vImag` :

```
vReal[i] = (double)i2sData[i]; // Partie réelle du signal  
vImag[i] = 0.0; // Partie imaginaire initialisée à zéro
```

Puis on réalise les opérations suivantes pour le calcul de la FFT :

```
FFT.windowing(this->vReal, SAMPLES, FFT_WIN_TYP_HAMMING, FFT_FORWARD); //  
FFT.compute(this->vReal, this->vImag, SAMPLES, FFT_FORWARD);  
FFT.complexToMagnitude(this->vReal, this->vImag, SAMPLES);
```

- ⇒ Ajouter le code précédent au bon endroit dans `CSon`.

- ⇒ Pourquoi utilise-t-on une fenêtre de Hamming ?

- ⇒ Quelle opération `complexToMagnitude` réalise, quelles sont les paramètres d'entrées et de sorties et comment obtient-t-on l'amplitude du spectre ? Visualiser le code de cette méthode.



4.3.Affichage du spectre sur l'écran Oled

⇒ Dans la fonction main, implémenter la fonction `AfficherSpectre` :

```
// @brief Affiche le spectre sur l'écran Oled
/// @param [in] vRe Amplitude du signal fréquentiel
void AfficherSpectre(double * vRe)
{
    // Définir les valeurs des barres du bargraphe (0 à 100)
    int barWidth = 1; // Largeur de chaque barre
    int spacing = 0; // Espace entre les barres
    int maxHeight = 63; // Hauteur maximale des barres
    ecranOled.clear();

    // Dessiner le bargraphe résolution de l'écran 128x64
    for (int i = 1; i < 128; i++) {
        // Calculer la hauteur de la barre en fonction de sa valeur
        // pour 512 échantillons 128 permet de voir les fréquences jusqu'à 22kHz
        (recouvrement spectre à partir de 512/2)
        double val = vRe[i]; // (vReal[i*2]+vReal[i*2+1])/2;
        int barHeight = map(val, 0, 700000, 0, maxHeight);

        // Dessiner le cadre de la barre (optionnel)
        // ecranOled.drawRect(i * (barWidth + spacing) + 10, 64 - maxHeight, barWidth,
        maxHeight);

        // Dessiner la barre pleine
        ecranOled.fillRect(i * (barWidth + spacing), 64 - barHeight, barWidth,
        barHeight);
    }
}
```

⇒ Appeler `AfficherSpectre` dans `loop`. Et vérifier le bon fonctionnement.

⇒ Faire en sorte que `AfficherSpectre` soit appelée qu'une itération sur 2.



4.4.Filtre numérique pour la pondération fréquentielle

Pour obtenir la compensation A, il faut réaliser un **filtre numérique passe-bande**. Celui-ci atténue les basses et les hautes fréquences.

Pour obtenir ce filtre on utilise un outil mathématique qui permet de transformer un signal du domaine temporel en un signal représenté par une série complexe. C'est la transformée en z.

Cela permet de trouver l'équation de récurrence et donc l'algorithme à implémenter pour obtenir la fonction de filtrage voulue.

Fonction de filtrage :

```
#include <iostream>
```

```
// Coefficients du filtre IIR pour la pondération A
const double b[] = { 0.255741125204258, -0.511482250408515, 0.255741125204258 };
```

```

const double a[] = { 1.0, -1.734725768809275, 0.766006600943264 };

// Fonction pour appliquer le filtre IIR (pondération A) sur un tableau
void applyAWeighting(double* signal, double* output, size_t length) {
    // Initialisation des variables pour les valeurs passées du filtre
    double x1 = 0.0, x2 = 0.0;
    double y1 = 0.0, y2 = 0.0;

    for (size_t i = 0; i < length; ++i) {
        // Calcul du filtre IIR
        double x0 = signal[i];
        double y0 = b[0] * x0 + b[1] * x1 + b[2] * x2 - a[1] * y1 - a[2] * y2;

        // Mise à jour des retards
        x2 = x1;
        x1 = x0;
        y2 = y1;
        y1 = y0;

        // Stocker la sortie filtrée dans le tableau de sortie
        output[i] = y0;
    }
}

int main() {
    // Exemple d'un signal audio (remplacer par vos données audio réelles)
    const size_t length = 10; // Taille du signal
    double audioSignal[length] = { 0.2, 0.4, 0.3, 0.1, -0.2, -0.4, -0.3, 0.0, 0.1, 0.3 }; // Signal audio fictif
    double filteredSignal[length]; // Tableau pour stocker le signal pondéré

    // Appliquer la pondération A
    applyAWeighting(audioSignal, filteredSignal, length);

    // Afficher le signal pondéré
    for (size_t i = 0; i < length; ++i) {
        std::cout << filteredSignal[i] << std::endl;
    }

    return 0;
}

```

⇒ A partir de ce code, réaliser la méthode permettant d'appliquer ce filtre pour obtenir la pondération A

