

# Analizzatore di Spettro Real-Time

Antonio Bernardini

21 gennaio 2025

# Indice

|          |  |          |
|----------|--|----------|
| <b>1</b> | <b>Analizzatore di Spettro Real-Time</b> | <b>2</b> |
| 1.1      | Introduzione                             | 2        |
| 1.2      | Teoria dei segnali                       | 3        |
| 1.2.1    | Introduzione                             | 3        |
| 1.2.2    | Conversione analogico-digitale           | 3        |
| 1.2.3    | Sistemi digitali                         | 7        |
| 1.2.4    | Acquisizione del segnale audio           | 8        |
| 1.2.5    | Analisi del segnale audio                | 8        |
| 1.2.6    | Trasformata di Fourier (DFT e FFT)       | 12       |
| 1.3      | Progettazione dell'architettura hardware | 19       |
| 1.3.1    | Introduzione                             | 19       |
| 1.3.2    | Microfono KY-037                         | 19       |
| 1.3.3    | Sistema digitale                         | 22       |
| 1.3.4    | Display OLED                             | 25       |
| 1.3.5    | Schema di montaggio finale               | 26       |
| 1.4      | Progettazione del firmware               | 27       |
| 1.4.1    | Introduzione                             | 27       |
| 1.4.2    | Organizzazione del progetto              | 27       |
| 1.5      | Conclusioni                              | 32       |

# Capitolo 1

## Analizzatore di Spettro Real-Time

Questo progetto ha l'obiettivo di sviluppare un analizzatore di spettro utilizzando una scheda ESP32, un display OLED  $128 \times 64$  e un sensore KY-037 per l'acquisizione del suono. L'analizzatore consente di visualizzare la distribuzione delle frequenze sonore, rendendolo utile per scopi didattici, hobbistici e applicazioni audio.

### 1.1 Introduzione

L'analizzatore di spettro è uno strumento fondamentale nell'ambito dell'analisi e dell'elaborazione dei segnali, progettato per rappresentare la distribuzione delle componenti in frequenza di un segnale nel dominio spettrale. La sua funzione principale è scomporre un segnale complesso, generalmente nel dominio temporale, nelle sue componenti sinusoidali elementari, mostrando l'ampiezza o la potenza associata a ciascuna frequenza. Questo processo si basa sulla teoria della trasformata di Fourier, che stabilisce che qualsiasi segnale periodico o non periodico può essere rappresentato come una somma di funzioni sinusoidali di diverse frequenze, ampiezze e fasi.

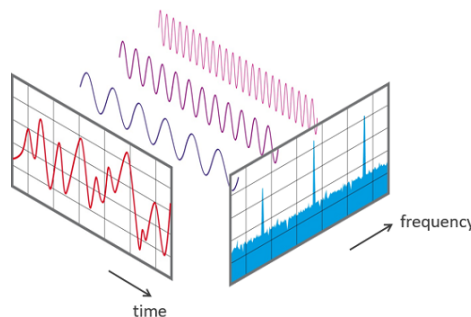


Figura 1.1: Trasformata di Fourier

La trasformata di Fourier permette di passare dal dominio temporale, dove il segnale è descritto in funzione del tempo, al dominio della frequenza, offrendo una visione dettagliata delle caratteristiche spettrali del segnale. Nel contesto pratico, l'analizzatore di spettro trova applicazione in diversi ambiti, tra cui:

- **Diagnostica dei segnali audio:** per identificare distorsioni, rumori o componenti non desiderate.

- **Ingegneria delle telecomunicazioni:** per analizzare la banda occupata da un segnale e ottimizzare la trasmissione.
- **Ingegneria acustica:** per studiare le caratteristiche sonore in ambienti specifici.

Un analizzatore di spettro real-time, come quello implementato nel presente progetto, consente di aggiornare continuamente l'analisi del segnale, fornendo una rappresentazione dinamica e interattiva. Questo è possibile grazie all'utilizzo di algoritmi efficienti per il calcolo della Fast Fourier Transform (FFT), una versione ottimizzata della trasformata di Fourier discreta (DFT), che riduce significativamente il costo computazionale rispetto all'approccio diretto. Nella lettura che segue, verrà spiegato ogni aspetto del progetto: dalla scelta e descrizione dei componenti hardware utilizzati, all'implementazione pratica della trasformata di Fourier tramite firmware e del rendering grafico sul display OLED, fino alle possibili estensioni future del sistema. Questo approccio dettagliato permetterà di comprendere non solo il funzionamento dell'analizzatore di spettro, ma anche i principi teorici e pratici che ne stanno alla base.

## 1.2 Teoria dei segnali

### 1.2.1 Introduzione

L'obiettivo principale dell'elettronica è il processamento dei segnali. Un segnale è una funzione matematica che rappresenta la variazione di una grandezza fisica (per esempio la temperatura, la pressione, ...) rispetto ad un'altra quantità (generalmente il tempo). L'elettronica analogica si concentra sullo studio di segnali continui, detti *segnali analogici*, in termini di tensione e corrente, mentre l'elettronica digitale si concentra sullo studio di segnali discreti, detti *segnali digitali*. Generalmente i segnali digitali si ottengono partendo da segnali analogici attraverso il processo che viene chiamato *digitalizzazione* di un segnale analogico, riassumibile in tre passi fondamentali: il *campionamento*, la *quantizzazione* e la *codifica*.

### 1.2.2 Conversione analogico-digitale

In generale, un segnale analogico è definito sull'intero asse  $x$  e può assumere un qualsiasi valore sull'asse  $y$ , proprio come una funzione  $y = f(x)$ . Il *campionamento* è il processo di riduzione di un segnale continuo in un segnale discreto. In particolare, il segnale discreto è ottenuto considerando i valori del segnale continuo solo a tempo discreto come mostrato in Figura 1.2.

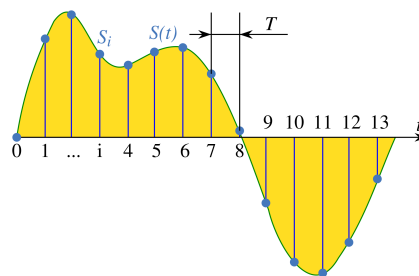


Figura 1.2: Campionamento di un segnale analogico

Il tempo tra due valori (chiamati anche *campioni* o *samples*) viene chiamato *tempo di campionamento* e viene indicato con la lettera  $T$ . L'inverso del tempo di campionamento definisce la *frequenza di campionamento*  $f_s = \frac{1}{T}$ . Tuttavia il campionamento deve rispettare il *teorema di Nyquist-Shannon*. In particolare, data la frequenza massima  $f_{\max}$  di un segnale analogico, per il teorema di Nyquist-Shannon la frequenza di campionamento (o *sample rate*)  $f_s$  deve essere almeno il doppio della frequenza massima del segnale analogico, cioè  $f_s > 2 \cdot f_{\max}$ . Se viene rispettata questa condizione, è allora possibile ricostruire, con l'utilizzo di apposite funzioni interpolatrici, il segnale analogico senza perderne alcuna informazione. Qualora invece non venga rispettata tale condizione, si riscontra un effetto conosciuto con il nome di *aliasing*, che comporta una distorsione del segnale analogico ricostruito. Generalmente per una buona e fedele ricostruzione del segnale analogico è richiesta una frequenza di campionamento che sia 5 ÷ 10 volte maggiore della frequenza massima contenuta nel segnale campionato.

Cerchiamo di capire meglio il concetto di campionamento e il teorema di Nyquist-Shannon con un esempio pratico. Come già sappiamo esso consiste nel prelievo di *campioni*, da un segnale analogico e continuo nel tempo, ogni  $\Delta t$  secondi<sup>1</sup>. Pertanto, più nel dettaglio, il teorema di Nyquist-Shannon stabilisce che, dato un segnale analogico  $x(t)$  la cui banda di frequenze sia limitata dalla frequenza  $f_{\max}$ , e dato  $n \in \mathbb{Z}$ , il segnale  $x(t)$  può essere univocamente ricostruito a partire dai suoi campioni  $x(n \cdot \Delta t)$  presi a frequenza  $f_s = \frac{1}{\Delta t}$  se  $f_s > 2 \cdot f_{\max}$  mediante la seguente formula:

$$x(t) = \sum_{k=-\infty}^{+\infty} x(k \cdot \Delta t) \cdot \text{sinc} \left( \frac{t}{\Delta t} - k \right), \quad \forall t \in \mathbb{R}$$

dove  $\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$  è la funzione seno cardinale. Ma, dato che la teoria non è sufficiente per capire appieno il concetto, vediamo concretamente. Utilizzando il seguente codice scritto in linguaggio Python :

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Signal parameters
5 amplitude = 1 # Signal amplitude
6 f_max = 10 # Signal maximum frequency
7 t = np.arange(0, 1, 0.01) # Time axis
8 x = amplitude * np.sin(2 * np.pi * f_max * t) # x(t)
9
10 plt.figure(figsize=(12, 10))
11
12 plt.subplot(2, 2, 1)
13 plt.plot(t, x)
14 plt.title(r"$x(t) = \sin(2 \pi f_{\text{max}} t)$")
15 plt.xlabel(r"$t$")
16 plt.ylabel(r"$x(t) = \sum_{k=-\infty}^{+\infty} x(k \cdot \Delta t) \cdot \text{sinc} \left( \frac{t}{\Delta t} - k \right)$")
17
18 # (1) f_s > 2 * f_max
19 k = 2
20 f_s = k * 2 * f_max
21 sampling_time = 1 / f_s
22 t = np.arange(0, 1 + sampling_time, sampling_time) # k * \Delta t
23 x = amplitude * np.sin(2 * np.pi * f_max * t) # x(t)
24
25 plt.subplot(2, 2, 2)
26 plt.stem(t, x)

```

<sup>1</sup>In questo caso viene usata la notazione  $\Delta t$  al posto della notazione  $T$  per indicare il tempo di campionamento. Questo va bene in quanto sarà utile nella definizione più generale del teorema di Nyquist-Shannon.

```

27 plt.title(r"$f_s > 2 f_{\text{max}}$")
28 plt.xlabel(r"$t = k \cdot \Delta t$")
29 plt.ylabel(r"$x(t) = \sin(2 \pi f_{\text{max}} t)$")
30
31 # (2)  $f_s < 2 \cdot f_{\text{max}}$ 
32  $f_s = 10$ 
33  $\text{sampling\_time} = 1 / f_s$ 
34  $t = \text{np.arange}(0, 1 + \text{sampling\_time}, \text{sampling\_time})$  #  $k \cdot \Delta t$ 
35  $x = \text{amplitude} \cdot \text{np.sin}(2 \cdot \text{np.pi} \cdot f_{\text{max}} \cdot t)$  #  $x(t)$ 
36
37 plt.subplot(2, 2, 3)
38 plt.stem(t, x)
39 plt.title(r"$f_s < 2 f_{\text{max}}$")
40 plt.xlabel(r"$t = k \cdot \Delta t$")
41 plt.ylabel(r"$x(t) = \sin(2 \pi f_{\text{max}} t)$")
42
43 # (3)  $f_s = 2 \cdot f_{\text{max}}$ 
44  $k = 1$ 
45  $f_s = k \cdot 2 \cdot f_{\text{max}}$ 
46  $\text{sampling\_time} = 1 / f_s$ 
47  $t = \text{np.arange}(0, 1 + \text{sampling\_time}, \text{sampling\_time})$  #  $k \cdot \Delta t$ 
48  $x = \text{amplitude} \cdot \text{np.sin}(2 \cdot \text{np.pi} \cdot f_{\text{max}} \cdot t)$  #  $x(n)$ 
49
50 plt.subplot(2, 2, 4)
51 plt.stem(t, x)
52 plt.title(r"$f_s = 2 f_{\text{max}}$")
53 plt.xlabel(r"$t = k \cdot \Delta t$")
54 plt.ylabel(r"$x(t) = \sin(2 \pi f_{\text{max}} t)$")
55
56 plt.tight_layout()
57 plt.show()

```

si ottengono i seguenti grafici:

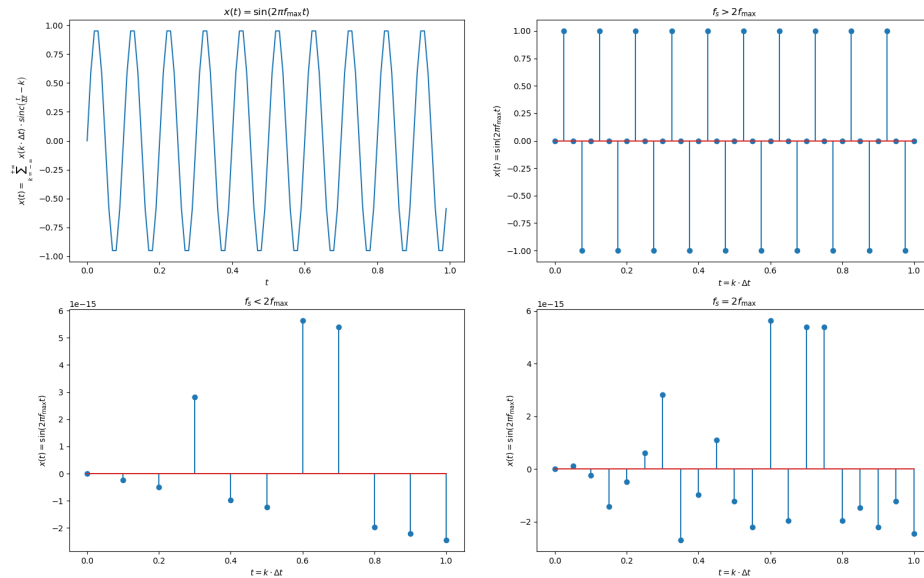


Figura 1.3: Campionamento di un segnale analogico

Il risultato ottenuto mostra chiaramente come il segnale analogico  $x(t) = \sin(2\pi f_{\text{max}} t)$  possa essere ricostruito correttamente solo se la frequenza di campionamento  $f_s > 2 \cdot f_{\text{max}}$ , mentre se  $f_s \leq 2 \cdot f_{\text{max}}$  si verifica il fenomeno dell'*aliasing*, che comporta una distorsione nel segnale ricostruito.

Infine possiamo introdurre e definire la frequenza di Nyquist come  $f_N = \frac{f_s}{2}$ , dove  $f_s$  è la frequenza di campionamento. Quando un segnale analogico contiene componenti in frequenza superiori alla frequenza di Nyquist, la loro ricostruzione porta a un segnale in

cui queste componenti risultano *riflesse*, cioè hanno una frequenza specchiata rispetto alla frequenza di Nyquist e rispetto al segnale analogico originale.

**Esempio.** Se il segnale analogico è una sinusoidale con frequenza  $f = 12\text{Hz}$  e la frequenza di campionamento è  $f_s = 20\text{Hz}$ , allora si ottiene  $f_r = f_N - (f - f_N) = 8\text{Hz}$ , dove  $f_r$  è la frequenza della sinusoidale ricostruita. L'obiettivo di questo esempio è illustrare il fenomeno dell'aliasing che si verifica quando il segnale analogico contiene componenti a frequenza più alta rispetto alla frequenza di Nyquist e quindi non può essere campionato correttamente. Poiché la frequenza del segnale  $f = 12\text{Hz}$  è superiore alla frequenza di Nyquist  $f_N = 10\text{Hz}$ , durante il campionamento il segnale verrà *aliasato*, cioè la sua frequenza verrà *riflessa* rispetto alla frequenza di Nyquist. In particolare, il segnale ricostruito avrà una frequenza di  $f_r = 8\text{Hz}$ , che è la frequenza specchiata rispetto alla frequenza di Nyquist.

Pertanto per evitare il fenomeno dell'aliasing è necessario:

1. adottare una frequenza di campionamento  $f_s$  maggiore se non si vogliono perdere le informazioni contenute nelle componenti ad alta frequenza del segnale analogico acquisito.
2. adottare un *filtraggio anti-aliasing* (filtro passa-basso) così da eliminare dal segnale analogico le frequenze maggiori della frequenza di Nyquist del campionatore.

Una volta che il segnale è stato campionato, il passo successivo è la *quantizzazione*, che consiste nel mappare i valori campionati su un insieme discreto di valori. In particolare, l'asse  $y$  del segnale analogico viene diviso in  $n$  livelli, chiamati *livelli di quantizzazione*. Ogni valore campionato viene quindi mappato sul livello di quantizzazione più vicino, generando un segnale discreto. Il risultato si può visualizzare in Figura 1.4, dove viene raffigurata una griglia che combina campionamento e quantizzazione.

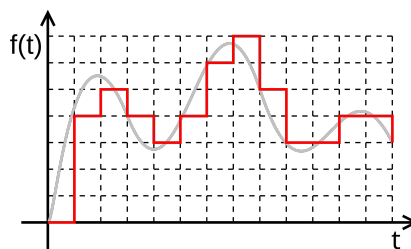


Figura 1.4: Quantizzazione di un segnale analogico

La quantizzazione introduce un errore noto come *errore di quantizzazione*, che è la differenza tra il valore campionato e il valore quantizzato. Per diminuire l'errore di quantizzazione, è possibile aumentare il numero di livelli di quantizzazione, riducendo la distanza tra i livelli e quindi l'errore. In particolare, gli  $n$  livelli di quantizzazione sono legati ai bit che vengono utilizzati per rappresentare il segnale digitale tramite la formula  $n = 2^k$ , dove  $k$  è il numero di bit. Più bit si usano migliore sarà il rapporto segnale-rumore (SNR) ed in particolare ad ogni aumento di 1 bit si ha un incremento di circa 6dB dell'SNR.

**Perchè proprio 6dB?** Supponiamo di avere un segnale digitale che può essere rappresentato da un certo numero di livelli discreti. Se aumentiamo di 1 bit, il numero di livelli possibili e l'ampiezza del segnale raddoppiano (infatti  $n = 2^{k+1} = 2 \cdot 2^k$ ). Quindi, dato che la variazione in decibel è data dalla formula:

$$\Delta L = 10 \cdot \log_{10} \left( \frac{P_1}{P_0} \right) = 10 \cdot \log_{10} \left( \frac{\frac{V_1^2}{R}}{\frac{V_0^2}{R}} \right) = 20 \cdot \log_{10} \left( \frac{V_1}{V_0} \right)$$

dove  $P_1$  e  $P_0$  sono le potenze del segnale in uscita e in ingresso,  $V_1$  e  $V_0$  sono le tensioni del segnale in uscita e in ingresso ed infine  $R$  è la resistenza di carico. Quindi, considerando un generico segnale sinusoidale di tensione elettrica nella forma  $v(t) = V_0 \cdot \sin(2\pi ft)$  dove  $V_0$  è l'ampiezza del segnale,  $f$  è la frequenza e  $t$  è il tempo, sapendo che  $V_1 = 2 \cdot V_0$  (perchè con  $k + 1$  bit raddoppia l'ampiezza) si ha  $\Delta L \approx 6.02$  dB.

Infine ogni livello di quantizzazione viene codificato usando dei numeri in rappresentazione binaria. Pertanto ogni numero viene codificato utilizzando combinazioni di 0 ed 1 e, come già detto precedentemente, più bit implicano più livelli di quantizzazione, che a loro volta implicano maggiore precisione.

### 1.2.3 Sistemi digitali

In generale, un sistema elettronico è composto dai seguenti elementi:



Figura 1.5: Elementi di un sistema elettronico.

dove l'*input* è il segnale analogico in ingresso al sistema, l'ADC (*Analog-to-Digital Converter*) è il convertitore analogico-digitale che trasforma il segnale analogico in un segnale digitale, il *sistema digitale* è il sistema che elabora il segnale digitale che può essere un microprocessore, un microcontrollore, un FPGA, un DSP, un ASIC (per esempio una GPU), il DAC (*Digital-to-Analog Converter*) è il convertitore digitale-analogico che trasforma il segnale digitale in un segnale analogico e infine l'*output* è il segnale analogico in uscita dal sistema.

Tuttavia, per questo progetto, occorre considerare un sistema elettronico meno complesso rispetto a quello di Figura 1.5, riconducibile al seguente schema:



Figura 1.6: Elementi del sistema elettronico per l'analizzatore di spettro.

infatti, come si può notare, l'ADC non è presente perchè esso è già integrato all'interno del sistema digitale (di cui parleremo più avanti) ed inoltre non c'è la necessità di ricostruire



il segnale analogico in uscita (quindi l'onda sonora acquisita dal microfono), ma solo di visualizzarlo su un display sfruttando la trasformata di Fourier, quindi non è presente nemmeno il DAC.

## 1.2.4 Acquisizione del segnale audio

Per acquisire il segnale audio, è necessario utilizzare un microfono che trasforma le onde sonore in segnali elettrici<sup>2</sup>. Per adesso ci limitiamo a mostrare come è possibile acquisire il segnale audio attraverso il linguaggio `C++` per dispositivi embedded. Pertanto possiamo sfruttare la funzione `loop()` ed in particolare la funzione `analogRead()` per leggere il valore del segnale audio acquisito dal microfono. Tuttavia questa lettura deve essere fatta ad una frequenza di campionamento  $f_s$  adeguata per evitare il fenomeno dell'aliasing e quindi ogni  $T = \frac{1}{f_s}$  microsecondi. Come verrà spiegato nella sezione 1.4, avremmo accesso alla variabile `engine.samplingPeriod` che rappresenta il tempo di campionamento espresso in microsecondi. Pertanto possiamo scrivere il seguente codice:

```
1  const uint32_t interval = engine.samplingPeriod;
2  uint32_t lastUpdate = 0;
3
4  void setup() {
5      pinMode(AUDIO_IN_PIN, INPUT);
6      lastUpdate = micros();
7  }
8
9  void loop() {
10     // Acquisition of audio samples
11     if (micros() - lastUpdate >= interval) {
12         lastUpdate = micros();
13         for (uint8_t i = 0; i < SAMPLES; ++i) {
14             uint16_t sample = analogRead(AUDIO_IN_PIN);
15             engine.re[i] = sample;
16             engine.im[i] = 0.0;
17         }
18
19         // ...
20     }
21 }
```

in particolare si noti la presenza delle variabili `engine.re[i]` e `engine.im[i]` che rappresentano rispettivamente la parte reale e immaginaria del segnale acquisito. Questa suddivisione è dovuta al fatto che la trasformata di Fourier è una trasformata complessa e quindi necessita di una parte reale e di una parte immaginaria. Si noti infine che grazie alla condizione `if (micros() - lastUpdate ≥ interval)` siamo in grado di acquisire il segnale audio ad una frequenza di campionamento  $f_s$  adeguata.

## 1.2.5 Analisi del segnale audio

Una volta che il segnale audio è stato acquisito dal microfono<sup>3</sup>, il passo successivo è l'analisi del segnale stesso. In particolare uno dei passi fondamentali è la preparazione per l'analisi spettrale, che consiste nella preparazione del segnale per potergli applicare la trasformata di Fourier.

---

<sup>2</sup>Per maggiori informazioni riguardo al microfono utilizzato si rimanda alla sezione 1.3.

<sup>3</sup>Non avendo fornito espliciti dettagli sui componenti hardware utilizzati, per adesso immaginiamo che l'acquisizione del segnale tramite il microfono avvenga per magia.

## Filtraggio del segnale

Il primo passo fondamentale è il filtraggio del segnale audio acquisito ed in particolare, esistono due tipologie principali di filtraggio: il *filtraggio hardware* e il *filtraggio software*. La prima tipologia ci permette di modificare l'hardware aggiungendo un filtro passa-basso analogico per eliminare le frequenze superiori alla frequenza di Nyquist. Per realizzare un filtro passa-basso si possono utilizzare componenti passivi come resistenze, condensatori e induttori, oppure componenti attivi come amplificatori operazionali. In particolare, indipendentemente dal tipo di componenti che scegliamo di usare<sup>4</sup>, la realizzazione del filtro dipende dal dimensionando dei componenti  $R$  e  $C$ , con i quali si ottiene un filtro passa-basso con una frequenza di taglio  $f_t$  pari alla frequenza di Nyquist, ovvero tale da rispettare la seguente relazione:

$$f_t = \frac{1}{2\pi RC} = f_N$$

dove  $R$  è la resistenza e  $C$  è il condensatore. Mentre, per quanto riguarda il filtraggio software, si può utilizzare un filtro digitale per eliminare le frequenze superiori alla frequenza di Nyquist. Un filtro digitale può essere implementato in vari modi, tra i quali abbiamo il *filtro FIR* (*Finite Impulse Response*) e il *filtro IIR* (*Infinite Impulse Response*). In particolare, un semplice filtro IIR può essere implementato con la seguente equazione ricorsiva:

$$y[n] = \alpha \cdot x[n] + (1 - \alpha) \cdot y[n - 1]$$

dove  $y[n]$  è l'uscita del filtro al campione  $n$ -esimo,  $x[n]$  è l'ingresso del filtro e  $\alpha$  è un parametro che determina la frequenza di taglio del filtro ed è definito come:

$$\alpha = \frac{T}{T + \tau}$$

dove  $T$  è il tempo di campionamento e  $\tau$  è la costante di tempo del filtro che è correlata alla frequenza di taglio. Entrambe le tecniche di filtraggio hanno vantaggi e svantaggi. Il filtraggio hardware è generalmente più efficiente e veloce, ma richiede modifiche all'hardware e può essere meno flessibile. Il filtraggio software è più flessibile e può essere facilmente modificato, ma richiede più risorse computazionali e può introdurre ritardi nel segnale. Per questo progetto, si è scelto di utilizzare un filtro software per la sua flessibilità e facilità di implementazione. A tal proposito, riporto di seguito, l'implementazione di un filtro IIR passa-basso in linguaggio `C++` per dispositivi embedded:

```
1 float EngineFFT::antiAliasingLowPassFilter(uint16_t sample) {
2     // Normalize the sample value to a range between 0 and 1
3     float currentValue = sample / ((1 << ADC_RESOLUTION) - 1.0f);
4
5     // IIR Low-Pass Filter
6     float filteredValue = alpha * currentValue + (1.0f - alpha) * this->lastFilteredValue;
7     this->lastFilteredValue = filteredValue;
8
9     return filteredValue;
10 }
```

---

<sup>4</sup>Questa affermazione, ovviamente, è vera per i circuiti elettronici che sfruttano unicamente componenti passivi. Ma, allo stesso tempo, è del tutto valida anche nel caso di circuiti che usano componenti attivi. Infatti questi ultimi a loro volta dovranno fare uso di componenti passivi per rendere il circuito stabile e quindi funzionare correttamente.

dove `alpha`<sup>5</sup> è il parametro che determina la frequenza di taglio del filtro IIR, mentre `this->lastFilteredValue` è il valore del campione precedente. Per utilizzare il filtro IIR passa-basso è sufficiente chiamare la funzione `engine.antiAliasingLowPassFilter(sample)` passando il valore del campione acquisito dal microfono. Pertanto modifichiamo il codice mostrato nella sezione 1.2.4 come segue:

```

1  const uint32_t interval = engine.samplingPeriod;
2  uint32_t lastUpdate = 0;
3
4  void setup() {
5      pinMode(AUDIO_IN_PIN, INPUT);
6      lastUpdate = micros();
7  }
8
9  void loop() {
10     // Acquisition of audio samples
11     if (micros() - lastUpdate >= interval) {
12         lastUpdate = micros();
13         for (uint8_t i = 0; i < SAMPLES; ++i) {
14             uint16_t sample = analogRead(AUDIO_IN_PIN);
15             engine.re[i] = engine.antiAliasingLowPassFilter(sample);
16             engine.im[i] = 0.0;
17         }
18
19         // ...
20     }
21 }

```

## Rimozione del DC offset

Un altro passo fondamentale è la rimozione del *DC offset* dal segnale audio. Il DC offset è una componente costante del segnale audio, come mostrato in Figura 1.7, che può essere causata da vari fattori, come rumore elettrico, interferenze, errori di calibrazione o errori di acquisizione. La presenza del DC offset può influenzare l'analisi spettrale del segnale audio, introducendo errori e distorsioni. Sostanzialmente si tratta di una traslazione del segnale audio lungo l'asse verticale, che può essere facilmente identificata visualizzando il segnale nel dominio del tempo.

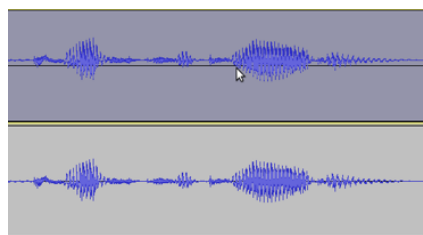


Figura 1.7: DC offset nel programma Audacity.

Pertanto è necessario rimuovere il DC offset dal segnale audio prima di procedere con l'analisi spettrale. Per rimuovere il DC offset si può implementare una funzione che sottrae la media del segnale dal segnale stesso. In particolare, la media del segnale può essere calcolata utilizzando la media aritmetica:

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

<sup>5</sup>Per maggiori informazioni consultare la sezione 1.4.

dove  $x[n]$  è il campione  $n$ -esimo del segnale audio e  $N$  è il numero totale di campioni. Una volta calcolata la media del segnale, è possibile rimuovere il DC offset sottraendo la media da ciascun campione:

$$y[n] = x[n] - \mu$$

dove  $y[n]$  è il campione  $n$ -esimo del segnale audio senza il DC offset. Pertanto possiamo scrivere la seguente funzione in linguaggio `C++` per dispositivi embedded:

```
1 void EngineFFT::removeDC(void) {
2     double mean = 0.0;
3     for (uint16_t i = 0; i < SAMPLES; ++i) {
4         mean += this->re[i];
5     }
6     mean /= SAMPLES;
7     for (uint16_t i = 0; i < SAMPLES; ++i) {
8         this->re[i] -= mean;
9     }
10 }
```

che implementa la rimozione del DC offset dal segnale audio acquisito.

## Windowing del segnale

Un altro passo fondamentale è il *windowing* del segnale audio. Nell'elaborazione numerica dei segnali una *funzione finestra* è una funzione che vale zero al di fuori di un certo intervallo. Quando un'altra funzione è moltiplicata per una funzione finestra, anche il prodotto assume valori nulli al di fuori dell'intervallo: tutto ciò che resta è la “vista” attraverso la finestra. Esistono vari tipi di funzioni finestra, tra cui la *finestra rettangolare*, la *finestra di Hann*, la *finestra di Hamming*, la *finestra di Blackman*, e molte altre.

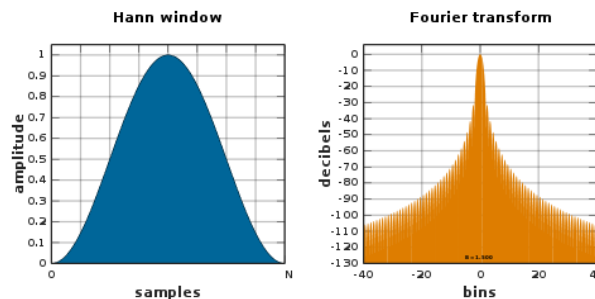


Figura 1.8: Finestra di Hann applicata al segnale audio.

L'obiettivo principale dell'utilizzo di una funzione finestra è quello di minimizzare l'*effetto di leakage* nelle frequenze risultanti dalla trasformata di Fourier. L'effetto di leakage si verifica quando il segnale campionato non corrisponde esattamente a un numero intero di cicli della frequenza che si sta cercando di analizzare. Questo porta a una distorsione nei risultati, causando la diffusione dell'energia del segnale su frequenze adiacenti. In altre parole, se la frequenza del segnale non è esattamente un multiplo intero della frequenza di campionamento, il segnale apparirà *diluuito* in frequenza, rendendo difficile identificare con precisione la sua frequenza e ampiezza nello spettro risultante. Per questo progetto si è scelto di utilizzare la *finestra di Hann*, una delle finestre più comuni e ampiamente utilizzate in applicazioni audio, definita come:

$$w[n] = 0.5 \cdot \left[ 1 - \cos \left( \frac{2\pi n}{N-1} \right) \right]$$

dove  $w[n]$  è il valore della finestra di Hann al campione  $n$ -esimo ed  $N$  è il numero totale di campioni. Il risultato dell'applicazione della finestra di Hann al segnale audio è mostrato in Figura 1.8. Quindi implementiamo la funzione di windowing in linguaggio `c++` per dispositivi embedded:

```

1 void EngineFFT::windowing(void) {
2     double samplesMinusOne = SAMPLES - 1.0;
3     for (uint16_t i = 0; i < (SAMPLES >> 1); ++i) {
4         double ratio = i / samplesMinusOne;
5
6         // Hann Window factor
7         double weighingFactor = 0.5 * (1.0 - cos(TWO_PI * ratio));
8
9         // Applying Hann Window to samples
10        this->re[i] *= weighingFactor;
11        this->re[SAMPLES - (i + 1)] *= weighingFactor;
12    }
13 }
```

## 1.2.6 Trasformata di Fourier (DFT e FFT)

La trasformata di Fourier discreta (DFT) è uno strumento matematico fondamentale per analizzare segnali nel dominio tempo trasformandoli in segnali nel dominio della frequenza. Questo approccio permette di rappresentare un segnale come somma di sinusoidi (o esponenziali complessi) di diverse frequenze, ampiezze e fasi. La DFT, di una sequenza di  $N$  campioni  $x[n]$ , è definita come:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi kn}{N}}, \quad k = 0, 1, \dots, N-1$$

dove  $X[k]$  è il  $k$ -esimo campione della DFT,  $x[n]$  è l' $n$ -esimo campione del segnale in ingresso,  $j$  è l'unità immaginaria,  $N$  è il numero totale di campioni ed infine  $e^{-j\frac{2\pi kn}{N}}$  è l'esponenziale complesso.

Tuttavia la definizione della DFT, in termini di informatica teorica, ha un costo computazionale elevato, pari a  $O(N^2)$ , che la rende inefficiente per dispositivi embedded con risorse limitate. Infatti, dal punto di vista puramente pratico, implementare la definizione della funzione `dft()` in linguaggio `c`, comporta un codice simile al seguente:

```

1 void dft(float in[], float complex out[], size_t N) {
2     for (size_t k = 0; k < N; ++k) {
3         out[k] = 0;
4         for (size_t n = 0; n < N; ++n) {
5             double angle = -2.0 * M_PI * k * n / N;
6             out[k] += in[n] * cexp(angle * I);
7         }
8     }
9 }
```

dove, per ogni campione  $k$ -esimo della DFT, si calcola la somma di tutti i campioni  $n$ -esimi del segnale moltiplicati per l'esponenziale complesso, rendendo il costo computazionale della DFT pari a  $O(N^2)$ . Inoltre nei dispositivi embedded non esiste il tipo di dato `complex`, quindi occorre gestire separatamente la parte reale e la parte immaginaria del segnale. Proprio per questi motivi, ma anche per molti altri, è stata introdotta la *Fast Fourier Transform* (FFT), un algoritmo efficiente per calcolare la DFT con un costo computazionale pari a  $O(N \log_2 N)$ .

L'algoritmo FFT è stato inventato da Cooley e Tukey nel 1965 e da allora è diventato uno degli algoritmi più importanti e ampiamente utilizzati in vari campi, tra cui l'elaborazione

numerica dei segnali, la teoria dei numeri, la crittografia, e molti altri. Per questo motivo, in questo progetto, si è scelto di utilizzare l'algoritmo di Cooley-Tukey, in particolare la versione *radix-2*, che è la più comune e ampiamente utilizzata. Questo algoritmo si basa sulla tecnica del *divide et impera*, suddividendo ricorsivamente il segnale in ingresso in sottoinsiemi di dimensioni pari e dispari, e sulle *butterfly operations*. Tuttavia prima di procedere con l'implementazione dell'algoritmo FFT, è necessario introdurre il concetto di *bit reversal*, ossia una tecnica utilizzata per riordinare i campioni del segnale in ingresso prima di applicare l'algoritmo FFT.

## Bit Reversal Algorithm

L'algoritmo di *bit reversal* è una tecnica che permette di separare i campioni con indice dispari dai campioni con indice pari, cosicché nella prima metà (da 0 a  $\frac{N}{2} - 1$ ) si trovino i campioni con indice pari e nella seconda metà (da  $\frac{N}{2}$  a  $N - 1$ ) si trovino i campioni con indice dispari<sup>6</sup>. Per implementare tale algoritmo in linguaggio `c++` per dispositivi embedded, si può utilizzare il seguente codice:

```
1 uint16_t j = 0;
2 for (uint16_t i = 0; i < (SAMPLES - 1); ++i) {
3     if (i < j) this->swap(&this->re[i], &this->re[j]);
4     uint16_t k = SAMPLES >> 1;
5     while (k <= j) {
6         j -= k;
7         k >>= 1;
8     }
9     j += k;
10 }
```

e, trattandosi di un singolo ciclo `for`, questo algoritmo può essere già implementato all'interno della funzione `fft()`<sup>7</sup> senza creare una funzione appositamente dedicata. Questo non vale per la funzione `swap()` che, invece, è necessario implementarla a parte:

```
1 void EngineFFT::swap(double *x, double *y) {
2     double temp = *x;
3     *x = *y;
4     *y = temp;
5 }
```

## Algoritmo di Cooley-Tukey

Iniziamo col definire i concetti di *butterfly operation* e *twiddle factor*. Una *butterfly operation* è una tecnica utilizzata nell'algoritmo FFT per calcolare la DFT di un segnale diviso in due parti (pari e dispari). In particolare, una *butterfly operation* è definita come segue:

$$\begin{cases} X[k] = E[k] + W_N^k \cdot O[k] \\ X[k + \frac{N}{2}] = E[k] - W_N^k \cdot O[k] \end{cases}$$

dove  $X[k]$  è il  $k$ -esimo campione della DFT,  $X[k + \frac{N}{2}]$  è il  $k + \frac{N}{2}$ -esimo campione della DFT,  $E[k]$  è il  $k$ -esimo campione della DFT della parte pari del segnale,  $O[k]$  è il  $k$ -esimo campione della DFT della parte dispari del segnale,  $W_N^k$  è il *twiddle factor* definito come  $W_N^k = e^{-j\frac{2\pi k}{N}}$ ,  $k = 0, 1, \dots, 7$  ed  $N$  è il numero totale di campioni.

Il *twiddle factor* è un fattore complesso che ruota il segnale di un certo angolo, in modo

<sup>6</sup>dove con  $N$  si indica il numero di campioni.

<sup>7</sup>Per maggiori informazioni riguardo l'organizzazione del codice sorgente del progetto è possibile consultare la sezione 1.4.

da calcolare la DFT di un segnale diviso in due parti. Immaginiamo di considerare il caso in cui il numero di campioni totali sia  $N = 8^8$ . In questo caso, il twiddle factor  $W_8^k$  è definito come segue:

$$W_8^k = e^{-j\frac{\pi k}{4}}$$

il quale, essendo un fattore complesso, si può rappresentare sul piano complesso come mostrato in Figura 1.9.

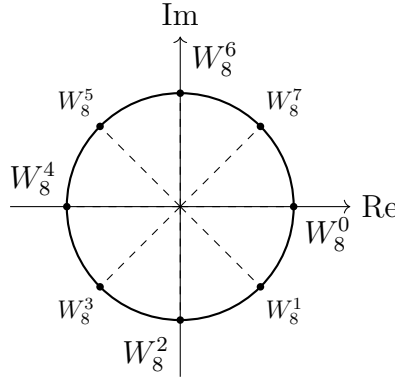


Figura 1.9: Rappresentazione dei twiddle factors  $W_8^i$  con  $i = 0, \dots, 7$ .

Come si evince dalla Figura 1.9, i twiddle factors  $W_8^i$  sono disposti uniformemente sulla circonferenza unitaria del piano complesso, con un angolo di  $\frac{\pi}{4}$  tra ciascun fattore. Pertanto è facile ricavare i seguenti legami fondamentali:

$$W_8^0 = -W_8^4, \quad W_8^1 = -W_8^5, \quad W_8^2 = -W_8^6, \quad W_8^3 = -W_8^7$$

tali legami valgono anche nel caso generale in cui il numero di campioni è  $N$ .

**Esempio.** Per definizione dell'algoritmo FFT il twiddle factor di partenza è  $W_8^4 = -1$ . Supponiamo di considerare il twiddle factor  $W_8^1 = e^{-j\frac{\pi}{4}}$ . Come sappiamo l'opposto di  $W_8^1$  è  $-W_8^5 = -e^{-j\frac{5\pi}{4}} = -\frac{\sqrt{2}}{2} - j\frac{\sqrt{2}}{2}$ . Voglio mostrare come, preso il twiddle factor  $W_8^4 = -1$ , valga la relazione  $W_8^4 \cdot W_8^1 = -W_8^5$ . Quindi, calcoliamo il prodotto dei due twiddle factors:

$$W_8^4 \cdot W_8^1 = -1 \cdot e^{-j\frac{\pi}{4}} = -\cos\left(\frac{\pi}{4}\right) - j \cdot \sin\left(\frac{\pi}{4}\right) = -\frac{\sqrt{2}}{2} - j\frac{\sqrt{2}}{2} = -W_8^5$$

Tuttavia non abbiamo ancora definito un modo per capire come aggiornare tale twiddle factor quando si passa da un generico livello  $l$  ad un livello  $l + 1$  dell'algoritmo FFT o, più nel dettaglio, come aggiornare il twiddle factor  $W_N^k$  quando la dimensione della *sub-DFT* raddoppia.

Tornando alle *butterfly operations*, una volta che il segnale è stato riordinato tramite l'algoritmo di *bit reversal*, possiamo applicare l'algoritmo FFT per calcolare la DFT di tale segnale. In particolare, facciamo riferimento alla Figura 1.10, la quale rappresenta le *butterfly operations* per un segnale con  $N = 8$  campioni. Come si può notare in input ci sono i campioni  $x[0], \dots, x[7]$ , opportunamente riordinati dal *bit reversal*, mentre in output

<sup>8</sup>Consideriamo un numero abbastanza piccolo per far comprendere al meglio il concetto. Infatti, nella pratica il numero di campioni è tendenzialmente  $N \geq 128$ .

ci sono i campioni  $X[0], \dots, X[7]$  che rappresentano la DFT del segnale in ingresso. In particolare, sia alla prima metà dei campioni che alla seconda metà viene applicata la DFT (anche detta *sub-DFT*), la quale, lavorando con un numero di campioni più piccolo rispetto ai campioni totali, permette di ridurre il costo computazionale. Quindi ci sono 4 campioni per la parte pari e 4 campioni per la parte dispari.

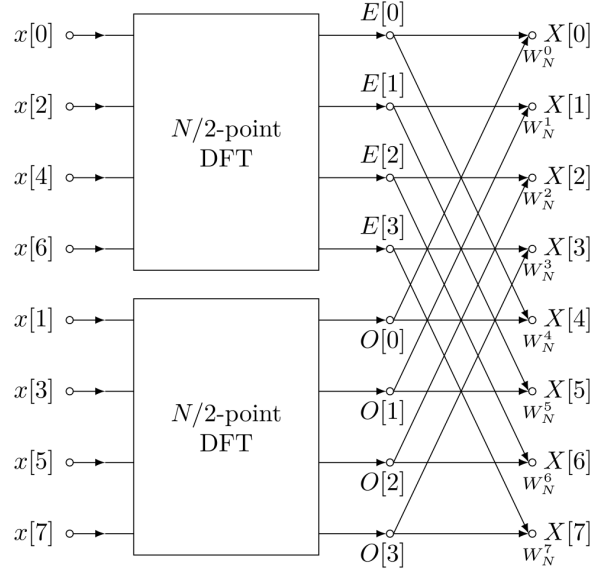


Figura 1.10: Butterfly operation per un segnale con  $N = 8$  campioni.

Terminate le *sub-DFT* si ottegono 4 campioni  $E[0], \dots, E[3]$  e 4 campioni  $O[0], \dots, O[3]$  (rispettivamente *even* e *odd*). La potenza delle *butterfly operations* è rappresentata dal fatto che i campioni  $E[0], \dots, E[3]$  e  $O[0], \dots, O[3]$  vengono combinati insieme per calcolare i campioni  $X[0], \dots, X[7]$ , come mostrato in Figura 1.10. Tutto questo ragionamento ha senso nel momento in cui ci troviamo ad uno specifico livello<sup>9</sup>  $l$  dell'algoritmo FFT, in cui il numero di campioni è  $N = 2^l$ . Pertanto occorre prestare attenzione al fatto che, ad ogni iterazione dell'algoritmo FFT, la grandezza delle *sub-DFT* viene raddoppiata e quindi all'aumentare del livello  $l \in [0, \log_2 N]$ , occorre dimezzare l'angolo tra i fattori di twiddle. Per fare ciò occorre calcolare la radice quadrata del twiddle factor utilizzando la seguente relazione:

$$(W_N^k)_{l+1} = \sqrt{(W_N^k)_l}$$

quindi, nel passaggio dal livello  $l$  al livello  $l + 1$ , il twiddle factor viene calcolato come la radice quadrata del twiddle factor precedente. Pertanto, dato che il twiddle factor  $(W_N^k)_l$  non è altro che un numero complesso, è possibile calcolarne la radice quadrata utilizzando la seguente formula:

$$(W_N^k)_{l+1} = \sqrt{(W_N^k)_l} = \sqrt{\frac{1 + \operatorname{Re}[(W_N^k)_l]}{2}} - j \cdot \sqrt{\frac{1 - \operatorname{Re}[(W_N^k)_l]}{2}} \quad (1.1)$$

<sup>9</sup>Con il termine livello, si intende la grandezza che assume una delle *sub-DFT* pari al massimo a  $\log_2 N$ , dove  $N$  è il numero totale dei campioni.



**Approfondimento.** L'equazione (1.1) può essere ottenuta considerando il twiddle factor  $(W_N^k)_l$  come un numero complesso nella forma  $z = a + j \cdot b$ . Per semplicità di notazione scriviamo  $(W_N^k)_l = z$ . Definendo  $\sqrt{(W_N^k)_l} = w = x + j \cdot y = \sqrt{z}$ , possiamo scrivere la seguente equazione:

$$(x + j \cdot y)^2 = a + j \cdot b \implies x^2 - y^2 + j \cdot 2xy = a + j \cdot b$$

dalla quale otteniamo il seguente sistema di equazioni:

$$\begin{cases} x^2 - y^2 = a \\ 2xy = b \end{cases}$$

ora, considerando l'equazione della circonferenza goniometrica, cioè  $x^2 + y^2 = 1$ , e considerando l'equazione  $x^2 - y^2 = a$ , possiamo scrivere:

$$\begin{cases} x^2 - y^2 = a \\ x^2 + y^2 = 1 \end{cases} \implies \begin{cases} x^2 = \frac{1+a}{2} \\ y^2 = \frac{1-a}{2} \end{cases} \implies \begin{cases} x = \pm \sqrt{\frac{1+a}{2}} \\ y = \pm \sqrt{\frac{1-a}{2}} \end{cases}$$

quindi poichè il twiddle factor deve avere una rotazione in senso antiorario, si considera la radice quadrata positiva per la parte reale  $x$  e la radice quadrata negativa per la parte immaginaria  $y$ . Pertanto si ottiene:

$$\sqrt{(W_N^k)_l} = \sqrt{\frac{1 + \text{Re}[(W_N^k)_l]}{2}} - j \cdot \sqrt{\frac{1 - \text{Re}[(W_N^k)_l]}{2}}$$

Per implementare l'algoritmo FFT in linguaggio `c++` per dispositivi embedded, possiamo procedere per step. Iniziamo riportando il codice del *bit reversal* all'interno della funzione `fft()`:

```
1 void EngineFFT::fft(void) {
2     // Bit Reversal Algorithm
3     uint16_t j = 0;
4     for (uint16_t i = 0; i < (SAMPLES - 1); ++i) {
5         if (i < j) this->swap(&this->re[i], &this->re[j]);
6         uint16_t k = SAMPLES >> 1;
7         while (k <= j) {
8             j -= k;
9             k >>= 1;
10        }
11        j += k;
12    }
13    // ...
14 }
15 }
```

a questo punto definiamo il numero complesso  $(W_N^k)_l$  che, per definizione, ha valore iniziale  $(W_N^k)_l = -1$ , quindi si ottiene:

```
1 void EngineFFT::fft(void) {
2     // Bit Reversal Algorithm
3     uint16_t j = 0;
4     for (uint16_t i = 0; i < (SAMPLES - 1); ++i) {
5         if (i < j) this->swap(&this->re[i], &this->re[j]);
6         uint16_t k = SAMPLES >> 1;
7         while (k <= j) {
8             j -= k;
9             k >>= 1;
10        }
11        j += k;
12    }
13    // ...
14 }
15 }
```

```

10     }
11     j += k;
12 }
13
14 // Compute the FFT with Cooley-Tukey Algorithm
15
16 // W_l = (re_W_l + j * im_W_l)
17 double re_W_l = -1.0;
18 double im_W_l = 0.0;
19
20 // ...
21 }

```

ora definiamo le variabili `next_size` e `curr_size` che rappresentano rispettivamente la dimensione della successiva e della corrente *sub-DFT* e creiamo un ciclo `for` per scorrere i  $\log_2 N$  livelli dell'algoritmo FFT:

```

1 void EngineFFT::fft(void) {
2     // Bit Reversal Algorithm
3     uint16_t j = 0;
4     for (uint16_t i = 0; i < (SAMPLES - 1); ++i) {
5         if (i < j) this->swap(&this->re[i], &this->re[j]);
6         uint16_t k = SAMPLES >> 1;
7         while (k <= j) {
8             j -= k;
9             k >>= 1;
10        }
11        j += k;
12    }
13
14    // Compute the FFT with Cooley-Tukey Algorithm
15
16    // W_l = (re_W_l + j * im_W_l)
17    double re_W_l = -1.0;
18    double im_W_l = 0.0;
19
20    // size of the next sub-DFT
21    // after log2(samples) iterations => next_size = samples = N
22    uint16_t next_size = 1;
23
24    for (uint8_t l = 0; l < this->levels; ++l) { // levels = log2(samples)
25        // size of the current sub-DFT
26        // after log2(samples) iterations => curr_size = samples / 2 = N / 2
27        uint16_t curr_size = next_size;
28
29        // ...
30    }
31 }

```

ora occorre aumentare la dimensione della variabile `next_size` di un fattore 2, calcolare il twiddle factor  $(W_N^k)_l$  per ogni campione della *sub-DFT* corrente (quindi del livello  $l$  corrente) ed infine usare le *butterfly operations* per calcolare i campioni  $X[k]$  e  $X[k + \frac{N}{2}]$ <sup>10</sup>:

```

1 void EngineFFT::fft(void) {
2     // Bit Reversal Algorithm
3     uint16_t j = 0;
4     for (uint16_t i = 0; i < (SAMPLES - 1); ++i) {
5         if (i < j) this->swap(&this->re[i], &this->re[j]);
6         uint16_t k = SAMPLES >> 1;
7         while (k <= j) {
8             j -= k;
9             k >>= 1;
10        }
11        j += k;
12    }
13
14    // Compute the FFT with Cooley-Tukey Algorithm
15

```

<sup>10</sup>I passaggi sono spiegati nel dettaglio nei commenti del codice.

```

16 // W_l = (re_W_l + j * im_W_l)
17 double re_W_l = -1.0;
18 double im_W_l = 0.0;
19
20 // size of the next sub-DFT
21 // after log2(samples) iterations => next_size = samples = N
22 uint16_t next_size = 1;
23
24 for (uint8_t l = 0; l < this->levels; ++l) { // levels = log2(samples)
25     // size of the current sub-DFT
26     // after log2(samples) iterations => curr_size = samples / 2 = N / 2
27     uint16_t curr_size = next_size;
28     next_size <= 1;
29
30     // Twiddle Factor: W_{N}^{k} = exp(-j * (2 * pi * k) / N)
31     double re_W = 1.0;
32     double im_W = 0.0;
33     for (j = 0; j < curr_size; ++j) { // foreach sub-DFT
34         for (uint16_t k = j; k < SAMPLES; k += next_size) { // Butterfly Operations
35             uint16_t i = k + curr_size; // i = k + N / 2
36
37             // W_{N}^{k} * O_k = (a + j * b) * (c + j * d)
38             // = (a * c - b * d) + j * (a * d + b * c)
39             // = re_temp + j * im_temp
40             double re_temp = re_W * this->re[i] - im_W * this->im[i];
41             double im_temp = re_W * this->im[i] + im_W * this->re[i];
42
43             // X_{k + N / 2} = E_k - W_{N}^{k} * O_k
44             // => X_i = E_k - (re_temp + j * im_temp)
45             // => Re[X_i] = Re[E_k] - re_temp
46             // => Im[X_i] = Im[E_k] - im_temp
47             this->re[i] = this->re[k] - re_temp;
48             this->im[i] = this->im[k] - im_temp;
49
50             // X_k = E_k + W_{N}^{k} * O_k
51             this->re[k] += re_temp;
52             this->im[k] += im_temp;
53         }
54     }
55     // ...
56 }
57 }
58 }

```

infine occorre calcolare il twiddle factor opposto a quel del livello corrente, cioè l'opposto di  $(W_N^k)_l$ , ed infine occorre calcolare il twiddle factor per il livello successivo, cioè  $(W_N^k)_{l+1}$ , utilizzando la formula (1.1):

```

1 void EngineFFT::fft(void) {
2     // Bit Reversal Algorithm
3     uint16_t j = 0;
4     for (uint16_t i = 0; i < (SAMPLES - 1); ++i) {
5         if (i < j) this->swap(&this->re[i], &this->re[j]);
6         uint16_t k = SAMPLES >> 1;
7         while (k <= j) {
8             j -= k;
9             k >>= 1;
10        }
11        j += k;
12    }
13
14    // Compute the FFT with Cooley-Tukey Algorithm
15
16    // W_l = (re_W_l + j * im_W_l)
17    double re_W_l = -1.0;
18    double im_W_l = 0.0;
19
20    // size of the next sub-DFT
21    // after log2(samples) iterations => next_size = samples = N
22    uint16_t next_size = 1;
23
24    for (uint8_t l = 0; l < this->levels; ++l) { // levels = log2(samples)

```

```

25 // size of the current sub-DFT
26 // after log2(samples) iterations => curr_size = samples / 2 = N / 2
27 uint16_t curr_size = next_size;
28 next_size <= 1;
29
30 // Twiddle Factor:  $W_N^{\{k\}} = \exp(-j * (2 * \pi * k) / N)$ 
31 double re_W = 1.0;
32 double im_W = 0.0;
33 for (j = 0; j < curr_size; ++j) { // foreach sub-DFT
34     for (uint16_t k = j; k < SAMPLES; k += next_size) { // Butterfly Operations
35         uint16_t i = k + curr_size; //  $i = k + N / 2$ 
36
37         //  $W_N^{\{k\}} * O_k = (a + j * b) * (c + j * d)$ 
38         //  $= (a * c - b * d) + j * (a * d + b * c)$ 
39         //  $= re\_temp + j * im\_temp$ 
40         double re_temp = re_W * this->re[i] - im_W * this->im[i];
41         double im_temp = re_W * this->im[i] + im_W * this->re[i];
42
43         //  $X_{\{k + N / 2\}} = E_k - W_N^{\{k\}} * O_k$ 
44         // =>  $X_i = E_k - (re\_temp + j * im\_temp)$ 
45         // =>  $Re[X_i] = Re[E_k] - re\_temp$ 
46         // =>  $Im[X_i] = Im[E_k] - im\_temp$ 
47         this->re[i] = this->re[k] - re_temp;
48         this->im[i] = this->im[k] - im_temp;
49
50         //  $X_k = E_k + W_N^{\{k\}} * O_k$ 
51         this->re[k] += re_temp;
52         this->im[k] += im_temp;
53     }
54
55     // Calculation of the opposite twiddle factor to apply
56     // butterfly operations to the next iteration
57     double temp = re_W * re_W_l - im_W * im_W_l;
58     im_W = re_W * im_W_l + im_W * re_W_l;
59     re_W = temp;
60 }
61
62 // Before start the next level, we need to calculate
63 // the square root of the twiddle factor
64 im_W_l = -sqrt((1.0 - re_W_l) / 2.0);
65 re_W_l = sqrt((1.0 + re_W_l) / 2.0);
66 }
67 }

```

Pertanto siamo riusciti ad implementare l'algoritmo FFT di Cooley-Tukey in linguaggio `C++` per dispositivi embedded, il quale permette di calcolare la DFT di un segnale con un costo computazionale pari a  $O(N \log_2 N)$ .

## 1.3 Progettazione dell'architettura hardware

### 1.3.1 Introduzione

In questa sezione si discutono le scelte progettuali relative all'hardware del progetto. In particolare, si discute della scelta del sistema digitale e delle periferiche utilizzate per l'acquisizione e la visualizzazione del segnale audio.

### 1.3.2 Microfono KY-037

In un sistema di acquisizione dati, i *trasduttori* sono dispositivi che forniscono in uscita una grandezza elettrica funzione della grandezza fisica da rilevare. Nel caso dell'analizzatore di spettro real-time la grandezza fisica che si deve rilevare è il suono, proveniente dall'ambiente circostante, che successivamente deve essere convertito in una tensione elettrica e per farlo possiamo utilizzare il sensore KY-037. Tale sensore, riportato in Figura

1.11, presenta un microfono, che permette di rilevare un range di frequenze da 50Hz fino a 10kHz. L'onda sonora acquisita dal microfono viene elaborata dal circuito integrato ad alta precisione LM393. Il sensore fornisce due uscite: un segnale analogico, disponibile sul pin **A0**, che viene collegato al sistema digitale, e un segnale digitale, disponibile sul pin **D0**, che può essere utilizzato per rilevare la presenza o l'assenza di suono superando una determinata soglia.



Figura 1.11: Pinout del sensore KY-037.

Il particolare, il circuito integrato LM393 presenta al suo interno due comparatori di tensione invertenti, i quali permettono di confrontare due tensioni e di generare un segnale in uscita in base al risultato del confronto.

### Circuito Elettronico del sensore KY-037

Andando più nel dettaglio possiamo fare riferimento al circuito mostrato in Figura 1.12. In particolare notiamo la presenza del microfono, contraddistinto dai terminali *S* e *G*, dove il terminale *S* è il terminale di segnale, mentre il terminale *G* è il terminale di massa. Infatti, il segnale audio rilevato viene fatto “passare” attraverso la resistenza  $R_3 = 150\Omega$ , giungendo ad un bivio (in gergo tecnico chiamato *nodo*), dove una parte del segnale viene inviata al pin **A0** del sensore KY-037, mentre l'altra parte del segnale viene inviata all'ingresso invertente del primo comparatore di tensione presente nel circuito integrato LM393. Invece l'ingresso non invertente viene collegato al partitore di tensione formato dalle resistenze  $R_2 = R_6 = 100k\Omega$ .

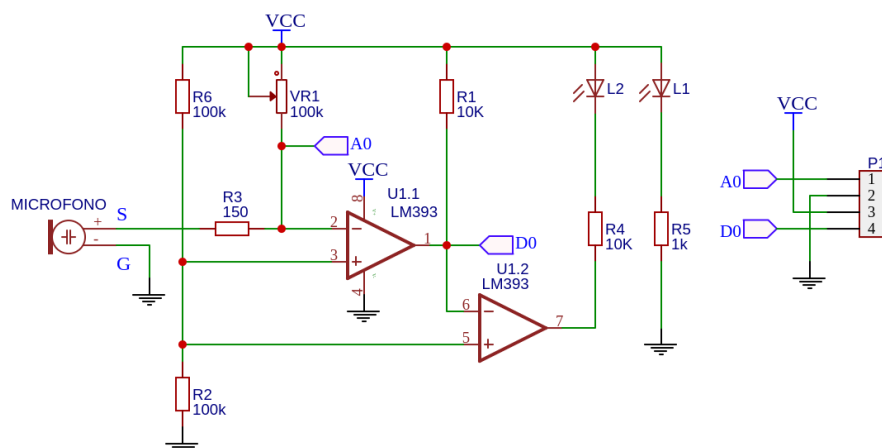


Figura 1.12: Circuito elettronico del sensore KY-037.

Si noti inoltre che il primo comparatore lavora unicamente con una tensione di alimentazione positiva, cioè  $V_{cc}$ , perchè la tensione di alimentazione negativa, cioè  $-V_{cc}$ , è collegata a massa.

Pertanto valgono le seguenti relazioni:

$$V_{cc} = 5V, \quad -V_{cc} = 0V$$

In generale, un comparatore di tensione è un circuito elettronico che ha due ingressi, uno invertente e uno non invertente, un'uscita e due alimentazioni, come mostrato in Figura 1.13.

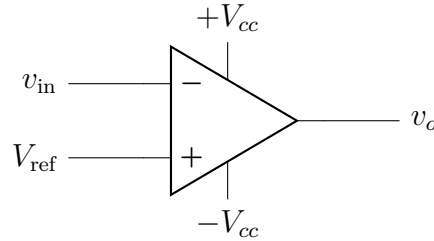


Figura 1.13: Comparatore di tensione invertente.

Il funzionamento del comparatore è molto semplice. Si definisco due tensioni di riferimento,  $V_{ref}$  all'ingresso non invertente e  $v_{in}$  all'ingresso invertente, dove  $V_{ref}$  è la tensione di riferimento e  $v_{in}$  è la tensione in ingresso. Si definisce inoltre la tensione di saturazione dell'uscita del comparatore come  $\pm V_{sat}$  che risulta essere sempre più piccola della tensione di alimentazione  $\pm V_{cc}$ . Quindi, se la tensione in ingresso  $v_{in}$  è minore della tensione di riferimento  $V_{ref}$ , allora l'uscita del comparatore sarà  $+V_{sat}$ . Mentre, se la tensione in ingresso  $v_{in}$  è maggiore della tensione di riferimento  $V_{ref}$ , allora l'uscita del comparatore sarà  $-V_{sat}$ . In formule otteniamo:

$$v_o = \begin{cases} +V_{sat} & \text{se } v_{in} < V_{ref} \\ -V_{sat} & \text{se } v_{in} > V_{ref} \end{cases} \quad (1.2)$$

Pertanto, come mostrato in Figura 1.14, il comparatore invertente effettua una vera e propria conversione analogico-digitale del segnale in ingresso  $v_{in}$  in un segnale digitale  $v_o$ .

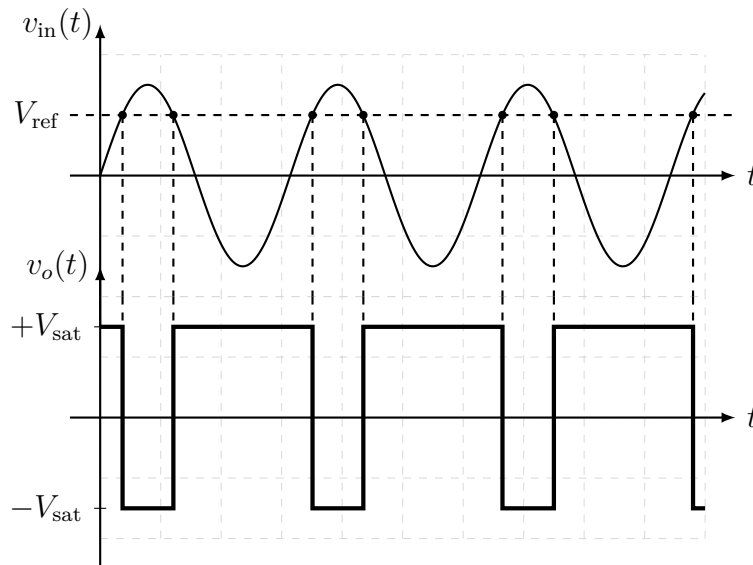


Figura 1.14: Forme d'onda del comparatore invertente.

Tuttavia le relazioni (1.2) valgono per un comparatore di tensione invertente avente due tensioni di alimentazione  $\pm V_{cc} \neq 0$ . Invece, nel caso del sensore KY-037, il comparatore di tensione invertente ha la tensione di alimentazione positiva  $+V_{cc} \neq 0$  e la tensione di alimentazione negativa  $-V_{cc} = 0V$ . Pertanto le relazioni precedenti si possono riscrivere come segue:

$$v_o = \begin{cases} V_{\text{sat}} & \text{se } v_{\text{in}} < V_{\text{ref}} \\ 0 & \text{se } v_{\text{in}} > V_{\text{ref}} \end{cases}$$

quindi, se immaginiamo di considerare la tensione di saturazione  $V_{\text{sat}}$  come un livello logico alto e la tensione 0 come un livello logico basso, allora possiamo ottenere i grafici mostrati in Figura 1.14 con l'uscita  $v_o$  traslata di  $V_{\text{sat}}$  verso l'alto, quindi una vera e propria onda digitale con valori logici 0 e  $V_{\text{sat}}$ .

Tuttavia, è importante sottolineare che il pin A0, che corrisponde al segnale  $v_{\text{in}}$ , viene inviato al sistema digitale (di cui parleremo a breve). Pertanto, il segnale  $v_o$  rappresenta il segnale audio digitalizzato che rimane all'interno del sensore KY-037 e che viene utilizzato insieme al diodo led  $L_2$  per indicare la presenza o l'assenza di un segnale audio.

Ora, facendo riferimento alla Figura 1.12, quando la tensione sul pin 2 (del primo comparatore) proveniente dal ramo formato dal microfono e regolata dal potenziometro  $V_{R_1}$  da  $10k\Omega$ , è inferiore a quella presente sul pin 3 (del primo comparatore), si avrà in uscita, sul pin 1, un segnale  $V_{\text{sat}}$ . Quando, invece, la tensione sul pin 2 è maggiore, si avrà un livello 0. Il secondo comparatore dell'integrato LM393 viene utilizzato per pilotare l'accensione del diodo led  $L_2$ , che segnala la rilevazione di un suono che supera la soglia impostata. L'accensione del diodo led  $L_1$ , invece, indicherà la presenza della tensione di alimentazione, poichè il sensore è collegato a  $V_{cc}$  e a massa tramite la resistenza  $R_5 = 1k\Omega$ . Inoltre, come si può intuire, è necessario regolare preventivamente il potenziometro  $V_{R_1}$  per ottenere il segnale  $V_{\text{sat}}$  solo quando il rumore supera la soglia desiderata. La taratura è relativamente semplice, facilitata dalla presenza del diodo led  $L_2$ . Infatti, agendo in senso orario sulla vite del potenziometro fino a quando il diodo led  $L_2$  si spegne, ed emettendo un suono, se il led si accende nuovamente, significa che il sensore è in grado di riconoscere il livello di rumore e quindi di produrre un segnale  $V_{\text{sat}}$  in uscita.

### 1.3.3 Sistema digitale

Alcuni anni fa, la scheda ESP8266 ha rivoluzionato il mondo dell'IoT Embedded essendo un microcontrollore programmabile, abilitato al Wi-Fi ed in grado di monitorare e controllare sensori da qualsiasi parte del mondo. Dopo lo straordinario successo ottenuto, Espressif, l'azienda di semiconduttori che ha creato tale scheda, ha rilasciato un perfetto aggiornamento potenziato: l'ESP32. Quest'ultima incorpora non solo il Wi-Fi ma anche il Bluetooth 4.0 (BLE/Bluetooth Smart), rendendola la scheda ideale per qualsiasi applicazione Internet of Things (IoT). Esistono numerose schede della famiglia ESP32 ed in particolare quella che è stata utilizzata per la realizzazione di questo progetto è la scheda ESP32 DevKit V1, che presenta numerose caratteristiche hardware come vedremo a breve.

#### Modulo ESP-WROOM-32

La scheda ESP32 DevKit V1 è dotata del modulo ESP-WROOM-32, che contiene il microprocessore Tensilica Xtensar Dual-Core a 32 bit LX6. Questo processore è simile

a quello utilizzato nell'ESP8266, ma ha due core CPU (che possono essere controllati individualmente), funziona con una frequenza di clock regolabile da 80Hz a 240MHz e può funzionare fino a 600DMIPS (Dhrystone Million Instructions Per Second).

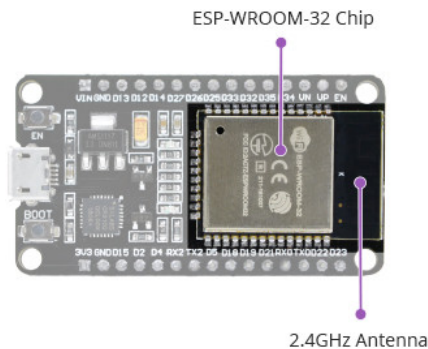


Figura 1.15: Modulo ESP-WROOM-32

La scheda presenta anche 448KB di ROM, 520KB di SRAM e 4MB di memoria Flash (per programmi e archiviazione dati), sufficienti per gestire le lunghe stringhe che compongono le pagine Web o i dati JSON/XML. L'ESP32 incorpora un ricetrasmittitore Wi-Fi 802.11b/g/n HT40, che gli consente di connettersi a una rete Wi-Fi per accedere a Internet (Station mode) o per creare la propria rete wireless Wi-Fi (Soft access point mode) a cui possono connettersi altri dispositivi. L'ESP32 supporta anche il Wi-Fi Direct, che è una buona opzione per le connessioni peer-to-peer che non richiedono un access point. Il Wi-Fi Direct è più semplice da configurare e offre velocità di trasferimento dati molto più elevate rispetto al Bluetooth. Infine la scheda supporta sia il Bluetooth 4.0 (BLE/Bluetooth Smart) che il Bluetooth Classic (BT), il che la rende ancora più versatile.

## Alimentazione elettrica

Poiché l'intervallo di tensione operativa dell'ESP32 è compreso tra 2.2V e 3.6V, la scheda include un regolatore di tensione LDO per mantenere la tensione stabile a 3.3V, in modo da fornire fino a 600mA di corrente.

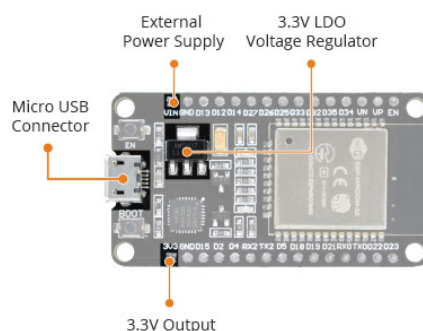


Figura 1.16: Alimentazione elettrica e regolatore di tensione LDO

L'uscita del regolatore da 3.3V viene distribuita al pin della scheda etichettato con `3V3`, che può essere utilizzato per alimentare circuiti esterni. Inoltre tale scheda è generalmente alimentata dal connettore Micro USB integrato, tuttavia se si dispone di un alimentatore da 5V, è possibile utilizzare il pin etichettato con `VIN` per alimentare direttamente l'ESP32 e le sue periferiche. Quest'ultimo metodo è quello che è stato utilizzato per l'alimentazione del display OLED (di cui parleremo più avanti) e del sensore KY-037.



## Periferiche e I/O

Sebbene l'ESP32 abbia in totale 30 pin, solo 25 di essi sono GPIO (General Purpose Input Output) e quindi accessibili tramite i rispettivi pin che si trovano sulla scheda. La piedinatura è la seguente:

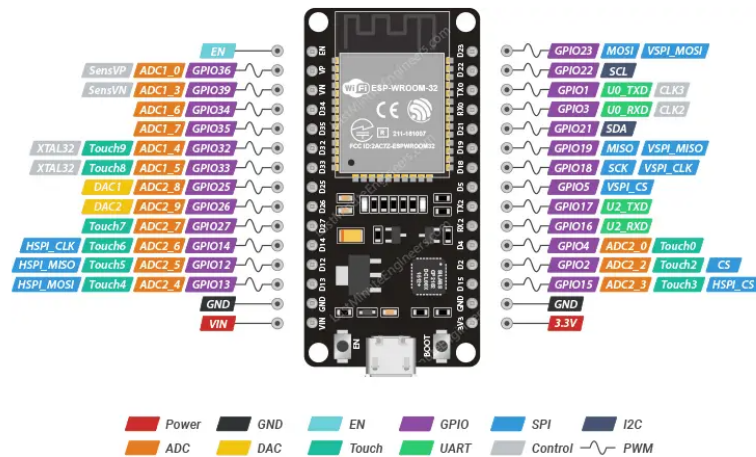


Figura 1.17: Pinout della scheda

A questi pin può essere assegnata una varietà di compiti, tra cui 15 canali di ADC a 12 bit con intervalli selezionabili di  $[0, 1]V$ ,  $[0, 1.4]V$ ,  $[0, 2]V$  o  $[0, 4]V$ , due interfacce UART (Universal Asynchronous Receiver-Transmitter) con controllo di flusso e supporto IrDA (Infrared Data Association), 25 pin PWM (Pulse Width Modulation) per controllare la velocità di un motore o la luminosità di un diodo led, due DAC a 8 bit per generare tensioni analogiche, tre interfacce SPI (Serial Peripheral Interface), un'interfaccia I2C (Inter-Integrated Circuit) per collegare vari sensori e periferiche ed infine due interfacce I2S<sup>11</sup> per collegare microfoni e altoparlanti, ed infine 9 Touch Pads GPIO con rilevamento tattile capacitivo.

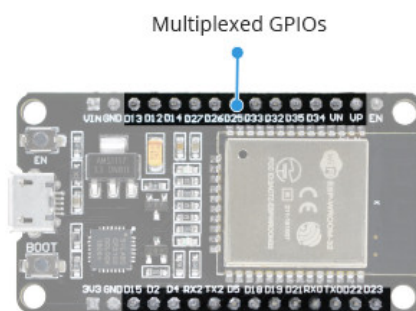


Figura 1.18: Periferiche e I/O

In particolare la funzione multiplexing, dei pin dell'ESP32, consente a più periferiche di condividere un singolo pin GPIO, infatti, un singolo pin GPIO può fungere da ingresso

<sup>11</sup>Il protocollo I2S è molto complesso e richiede il framework nativo di Espressif nella configurazione dell'ecosistema PlatformIO. In questo progetto si è scelto di utilizzare il framework di Arduino per garantire maggiore compatibilità con entrambe le schede. Per maggiori informazioni è possibile consultare la sezione 1.5.

ADC, uscita DAC o Touch Pad. Infine i pin `GPIO34` , `GPIO35` , `GPIO36(VP)` e `GPIO39(VN)` non possono essere configurati come uscite. Possono essere utilizzati come ingressi digitali o analogici o per altri scopi. Inoltre non dispongono di resistori pull-up e pull-down interni, a differenza degli altri pin GPIO.

## Interruttori e indicatori LED

Sono presenti due pulsanti sulla scheda ESP32. Il pulsante Reset, etichettato `EN` , viene utilizzato per ripristinare il chip ESP32. L'altro pulsante è il pulsante Boot, che viene utilizzato per scaricare nuovi software o programmi.

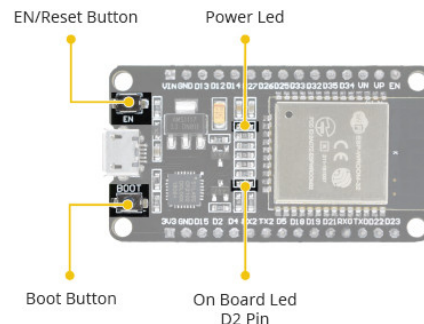


Figura 1.19: Interruttori e indicatori LED

La scheda include anche due indicatori led. Il diodo led rosso indica che la scheda è accesa, mentre il diodo led blu è programmabile ed è collegato al pin `D2` della scheda.

## Comunicazione seriale

La scheda include il controller bridge USB-to-UART CP2102 di Silicon Labs, che converte i segnali USB in seriali e consente di programmare il chip ESP32.

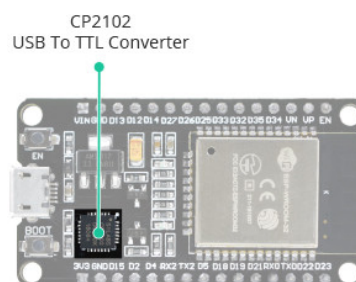


Figura 1.20: Comunicazione seriale

### 1.3.4 Display OLED

I display OLED sono disponibili in una vasta gamma di dimensioni (per esempio  $128 \times 64$  o  $128 \times 32$ ) e colori (per esempio OLED bianco, blu e bicolore). Alcuni display OLED hanno un'interfaccia I2C, mentre altri hanno un'interfaccia SPI. Una cosa che hanno tutti in comune, tuttavia, è la presenza di un potente controller driver OLED CMOS a chip singolo: l'SSD1306. Quest'ultimo gestisce tutto il buffering della RAM. Il display OLED che è stato utilizzato per il progetto ha dimensioni  $128 \times 64$  ed utilizza un'interfaccia I2C,

rispettivamente con collegamenti rispettivamente ai pin SCL ( `GPIO22` ) e SDA ( `GPIO21` ), ed è molto simile a quello mostrato in Figura 1.21.



Figura 1.21: Display OLED

Un display OLED, a differenza di un display LCD a caratteri, non richiede retroilluminazione perché genera luce propria. Ciò spiega l'elevato contrasto del display, l'angolo di visione estremamente ampio e la capacità di visualizzare livelli di nero profondo. L'assenza di retroilluminazione riduce significativamente il consumo energetico. Il display utilizza in media circa 20mA, anche se questo varia a seconda della quantità di display illuminato. Il controller SSD1306 lavora tra 1.65V e 3.3V, mentre il pannello OLED richiede una tensione di alimentazione compresa tra 7V e 15V. Tutti i requisiti di alimentazione sono soddisfatti dal circuito interno della pompa di carica<sup>12</sup>. Ciò rende possibile collegare il display OLED alla scheda ESP32 o a qualsiasi altro microcontrollore logico da 5V senza utilizzare un convertitore di livello logico.

## Mappatura della memoria

Per controllare il display è fondamentale comprendere la mappatura della memoria. Indipendentemente dalle dimensioni<sup>13</sup> del display OLED, il driver SSD1306 include una RAM di dati di visualizzazione grafica (GDDRAM) da 1KB che memorizza il pattern di bit da visualizzare sullo schermo. Quest'area di memoria  $M = 1\text{KB}$  è divisa in  $p = 8$  pagine (da 0 a 7). Ogni pagina ha  $s = 128$  colonne o segmenti (blocco da 0 a 127). Inoltre, ciascuna colonna può memorizzare  $n = 8$  bit di dati (da 0 a 7). Pertanto è facile dimostrare che la memoria  $M$  vale:

$$M = p \cdot s \cdot n = 8 \cdot 128 \cdot 8 = 8192 \text{ bit} = 1024 \text{ byte} = 1\text{KB}$$

L'intera memoria da 1KB, incluse pagine, segmenti e dati, è evidenziata in Figura 1.22.

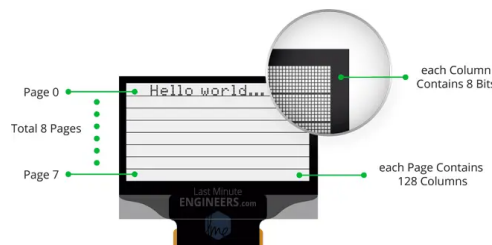


Figura 1.22: Mappatura della memoria  $M = 1\text{KB}$

<sup>12</sup>La pompa di carica è un circuito elettronico che usa dei condensatori per immagazzinare energia in maniera da ottenere delle sorgenti con tensioni più elevate o più basse di quelle disponibili dall'alimentazione.

<sup>13</sup>Ogni modulo contiene  $M = 1\text{KB}$  di RAM. Il modulo OLED  $128 \times 64$  visualizza l'intero contenuto di  $M = 1\text{KB}$  di RAM (tutte le 8 pagine), mentre il modulo OLED  $128 \times 32$  visualizza solo metà della RAM (le prime 4 pagine).

In particolare, ogni bit rappresenta un singolo pixel sullo schermo che può essere attivato o disattivato direttamente dal codice.

### 1.3.5 Schema di montaggio finale

In conclusione, l'intero sistema, dal punto di vista hardware, è composto da un microfono KY-037, un display OLED  $128 \times 64$  e una scheda ESP32 DevKit V1. Pertanto collegando adeguatamente i vari componenti, come mostrato in Figura 1.23, è possibile realizzare un analizzatore di spettro audio real-time.

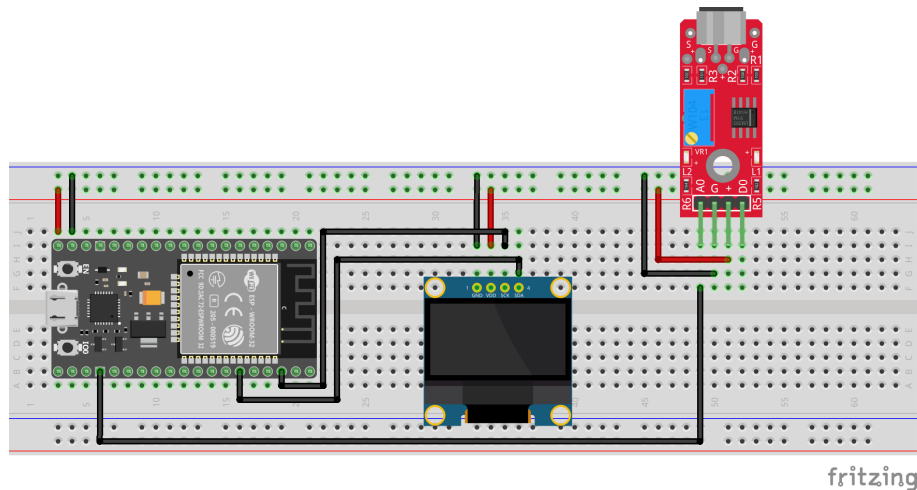


Figura 1.23: Schema di montaggio finale

Tuttavia occorre finalizzare ancora il progetto con il firmware, che permetterà di acquisire il segnale audio dal sensore KY-037, di calcolare la DFT del segnale e di visualizzare lo spettro del segnale sul display OLED.

## 1.4 Progettazione del firmware

### 1.4.1 Introduzione

Per realizzare il firmware del progetto è stato utilizzato il framework di Arduino, che permette di programmare la scheda ESP32 DevKit V1 utilizzando il linguaggio di programmazione `C++`. A tale scopo si è utilizzato l'ecosistema open-source di PlatformIO, che permette di programmare microcontrollori e microprocessori in modo professionale, fornendo un ambiente di sviluppo integrato (IDE) per la scrittura del codice, la compilazione e il caricamento del firmware sul microcontrollore. In questa sezione si discutono le scelte progettuali relative al firmware del progetto, con particolare attenzione all'acquisizione del segnale audio, al calcolo della DFT e alla visualizzazione dello spettro del segnale. Alcune parti di codice sono già state discusse nelle sezioni precedenti, pertanto si discuteranno solo le parti di codice più significative.

### 1.4.2 Organizzazione del progetto

Il progetto è organizzato in diverse cartelle e file, come mostrato in Figura 1.24. In particolare la cartella `src/` contiene i file `main.cpp` e `fft.cpp` mentre la cartella `include/`

contiene i file `fft.h` e `settings.h`. Inoltre è presente anche il file `platformio.ini` che contiene le impostazioni del progetto relative al framework di Arduino, alla scheda ESP32 DevKit V1 e ad altre librerie esterne utilizzate nel progetto.

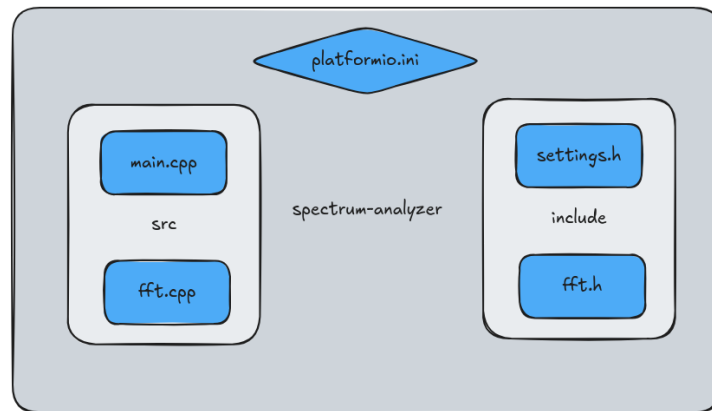


Figura 1.24: Struttura del progetto

In particolare, per utilizzare il display OLED con la scheda ESP32, è necessario installare la libreria `Adafruit_SSD1306` e la libreria `Adafruit_GFX`. Pertanto occorre modificare il file `platformio.ini` come mostrato di seguito:

```
1 [env:esp32doit-devkit-v1]
2 platform = espressif32
3 board = esp32doit-devkit-v1
4 framework = arduino
5 lib_deps =
6     adafruit/Adafruit SSD1306@^2.5.13
7     adafruit/Adafruit GFX Library@^1.11.11
```

Ora, facendo riferimento alla Figura 1.24, il file `settings.h` contiene le impostazioni del progetto, come ad esempio la dimensione del display OLED e la frequenza di campionamento del segnale audio. Vediamolo nel dettaglio:

```
1 #pragma once
2
3 #include <Arduino.h>
4 #include <Wire.h>
5
6 // External Libraries
7 #include <Adafruit_GFX.h>
8 #include <Adafruit_SSD1306.h>
9
10 // Serial Communication Variables
11 constexpr uint32_t BAUD_RATE = 115200;
12 constexpr uint8_t ADC_RESOLUTION = 12;
13
14 // Audio Input Variables
15 constexpr uint8_t AUDIO_IN_PIN = 34;
16
17 // Adafruit OLED Display Variables
18 #define SCREEN_WIDTH 128
19 #define SCREEN_HEIGHT 64
20 #define OLED_RESET 4
21 #define SCREEN_ADDRESS 0x3C
22
23 // FFT Variables
24 constexpr uint16_t SAMPLE_RATE = 22050; // 22.05kHz (using Nyquist-Shannon Theorem)
25 constexpr uint16_t SAMPLES = 128;
26 constexpr uint8_t BAR_WIDTH = 2;
27 constexpr uint8_t BAR_SPACING = 2;
```

dove alcune variabili sono autoesplicative, come ad esempio la variabile `BAUD_RATE` che rappresenta la velocità di trasmissione dati seriale, la variabile `ADC_RESOLUTION` che rappresenta la risoluzione del convertitore analogico-digitale, la variabile `AUDIO_IN_PIN` che rappresenta il pin di ingresso del segnale audio, la variabile `SAMPLE_RATE` che rappresenta la frequenza di campionamento del segnale audio e la variabile `SAMPLES` che rappresenta il numero di campioni del segnale audio. Altre variabili, come ad esempio la variabile `BAR_WIDTH` che rappresenta la larghezza delle barre dello spettro del segnale audio e la variabile `BAR_SPACING` che rappresenta lo spazio tra le barre dello spettro del segnale audio, sono state scelte in modo arbitrario per ottenere un risultato visivo gradevole.

In particolare, il display OLED ha 128 colonne, è possibile visualizzare su di esso solo 128 frequenze dello spettro del segnale audio. Tuttavia, avendo definito le variabili `BAR_WIDTH` e `BAR_SPACING`, è possibile visualizzare un numero minore di frequenze (proprio perchè vengono aggiunti degli spazi vuoti, ossia delle colonne del display che non saranno mai accese). Quindi è possibile calcolare il numero di frequenze visualizzate con la seguente formula:

$$\text{Frequenze visualizzate} = \frac{\text{SCREEN WIDTH}}{\text{BAR WIDTH} + \text{BAR SPACING}} = \frac{128}{2 + 2} = 32 \text{ frequenze}$$

mentre la frequenza  $f_b$  associata a ciascuna barra dello spettro è data dalla formula:

$$f_b = \frac{f_s}{N} \cdot n$$

dove  $f_s$  è la frequenza di campionamento,  $N$  è il numero di campioni e  $n$  rappresenta l'indice della barra che rappresenta una specifica banda di frequenze.

**Esempio.** Per  $n = 1$  si ha che la frequenza associata alla prima barra dello spettro è data da:

$$f_b = \frac{22050}{128} \cdot 1 = 172.27\text{Hz}$$

per  $n = 2$  si ha che la frequenza associata alla seconda barra dello spettro è data da:

$$f_b = \frac{22050}{128} \cdot 2 = 344.54\text{Hz}$$

e così via fino a  $n = 32$ .

Continuiamo dunque con il file `main.cpp`, il cui contenuto è già stato spiegato in parte nella sezione 1.2.4. Dunque, per chiarezza, riporto il codice completo del file `main.cpp`:

```
1 #include "fft.h"
2
3 EngineFFT engine = EngineFFT();
4
5 const uint32_t interval = engine.samplingPeriod;
6 uint32_t lastUpdate = 0;
7
8 void setup() {
9     engine.displayInit();
10    pinMode(AUDIO_IN_PIN, INPUT);
11    lastUpdate = micros();
12 }
13
14 void loop() {
15     // Acquisition of audio samples
16     if (micros() - lastUpdate >= interval) {
17         lastUpdate = micros();
```

```

18     for (uint8_t i = 0; i < SAMPLES; ++i) {
19         uint16_t sample = analogRead(AUDIO_IN_PIN);
20         engine.re[i] = engine.antiAliasingLowPassFilter(sample);
21         engine.im[i] = 0.0;
22     }
23
24     // Real-Time FFT
25     engine.removeDC();
26     engine.windowing();
27     engine.fft();
28     engine.drawBigBars();
29 }
30 }

```

come si può notare, il codice è diviso in due parti principali: la funzione `setup()` e la funzione `loop()`. La funzione `setup()` inizializza il display OLED e imposta il pin `AUDIO_IN_PIN` come ingresso. La funzione `loop()` acquisisce i campioni del segnale audio, calcola la DFT del segnale e visualizza lo spettro del segnale sul display OLED. In particolare, è stata creata una classe `EngineFFT` che contiene tutti i metodi necessari per l'acquisizione del segnale audio, il calcolo della DFT e la visualizzazione dello spettro del segnale. Quindi tutti i metodi della classe `EngineFFT` e la classe stessa, sono stati implementati nei file `fft.h` e `fft.cpp`. Pertanto analizziamo nel dettaglio il contenuto del file `fft.h`:

```

1  #pragma once
2
3  #include "settings.h"
4
5  class EngineFFT {
6  public:
7      EngineFFT(void);
8      void displayInit(uint32_t baudrate = BAUD_RATE);
9      float antiAliasingLowPassFilter(uint16_t sample);
10     void removeDC(void);
11     void windowing(void); // only Hann Window type
12     void fft(void); // only forward direction
13     void drawBigBars(void);
14
15     uint32_t samplingPeriod; // microseconds
16     double re[SAMPLES];
17     double im[SAMPLES];
18
19 private:
20     void swap(double *x, double *y);
21     void createMagnitudes(void);
22     double getPeak(void);
23
24     uint8_t levels;
25     uint8_t fontSize;
26     Adafruit_SSD1306 display;
27     float lastFilteredValue;
28
29     const float alpha = 0.15f;
30     const uint16_t maxBars = SCREEN_WIDTH / (BAR_WIDTH + BAR_SPACING);
31 };

```

dove, poichè una buona parte dei metodi sono stati già discussi nelle sezioni [1.2.4](#), [1.2.5](#), [1.2.5](#) e [1.2.6](#), in questa sezione, ci concentreremo unicamente sul costruttore della classe `EngineFFT` e sui metodi `displayInit()` e `drawBigBars()`. In particolare, il costruttore della classe `EngineFFT` è definito come segue:

```

1  EngineFFT::EngineFFT(void) {
2      this->levels = log2(SAMPLES);
3      this->samplingPeriod = round(1000000 * (1.0 / SAMPLE_RATE)); // T[us] = 1 / f => T *
4      10^(-6) = 1 / f => T = 10^6 / f
5      this->fontSize = 1;
6      this->lastFilteredValue = 0.0f;
7  }

```



dove vengono calcolati i livelli delle *sub-DFT*, il periodo di campionamento del segnale audio ed inoltre viene inizializzata la dimensione del font del display OLED e il valore dell'ultimo campione filtrato. Continuando, il metodo `displayInit()` che si occupa di inizializzare il display OLED, è definito come segue:

```

1 void EngineFFT::displayInit(uint32_t baudrate) {
2     Serial.begin(baudrate);
3     Wire.begin();
4
5     this->display = Adafruit_SSD1306(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);
6     if (!this->display.begin(SSD1306_SWITCHCAPVCC, SCREEN_ADDRESS)) {
7         Serial.println(F("SSD1306 allocation failed"));
8         for (;;);
9     }
10    this->display.setTextColor(WHITE);
11    this->display.clearDisplay();
12    this->display.display();
13 }

```

dove vengono inizializzati i pin di comunicazione seriale, il display OLED e il colore del testo. Infine, il metodo `drawBigBars()` disegna le barre dello spettro del segnale sul display OLED e viene definito come segue:

```

1 void EngineFFT::createMagnitudes(void) {
2     for (uint16_t i = 0; i < SAMPLES; ++i) {
3         this->re[i] = sqrt(sq(this->re[i]) + sq(this->im[i]));
4     }
5 }
6
7 double EngineFFT::getPeak(void) {
8     double peak = 0.0;
9     for (uint16_t i = 0; i < SAMPLES; ++i) {
10        peak = (peak > this->re[i]) ? peak : this->re[i];
11    }
12    return peak;
13 }
14
15 void EngineFFT::drawBigBars(void) {
16     this->createMagnitudes();
17     double peak = this->getPeak();
18
19     this->display.fillRect(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, BLACK);
20
21     for (uint16_t i = 0; i < maxBars && i < SAMPLES; ++i) {
22         double magnitude = this->re[i];
23         double logMagnitude = magnitude > 0.0 ? log(magnitude + 1.0) : 0.0;
24         double normalizedHeight = (logMagnitude / log(peak + 1.0)) * SCREEN_HEIGHT;
25         uint16_t barHeight = static_cast<uint16_t>(normalizedHeight);
26
27         uint16_t x = i * (BAR_WIDTH + BAR_SPACING);
28         uint16_t y = SCREEN_HEIGHT - barHeight;
29
30         this->display.fillRect(x, y, BAR_WIDTH, barHeight, WHITE);
31     }
32
33     this->display.display();
34 }

```

dove sono riportati anche i metodi `createMagnitudes()` e `getPeak()` che vengono utilizzati per calcolare le magnitudini e il picco dello spettro del segnale. In particolare, la funzione `drawBigBars()` disegna le barre dello spettro del segnale sul display OLED, calcolando la magnitudine normalizzata di ciascuna barra e disegnando la barra sul display. Per maggiori informazioni riguardo al codice sorgente del progetto e al funzionamento pratico è possibile consultare il repository GitHub del progetto all'indirizzo <https://github.com/AntonioBerna/spectrum-analyzer>.



## 1.5 Conclusioni

Il progetto dell'analizzatore di spettro audio rappresenta un esempio concreto di come sia possibile coniugare conoscenze teoriche e capacità pratiche per sviluppare uno strumento versatile e innovativo. Questo lavoro ha offerto l'opportunità di esplorare a fondo l'elaborazione digitale dei segnali, l'ottimizzazione delle risorse hardware e software, e la progettazione di un sistema in grado di operare in tempo reale. Le funzionalità attualmente implementate costituiscono una solida base per futuri sviluppi, che potrebbero ampliare le possibilità di utilizzo del dispositivo. Alcune delle principali estensioni individuate includono:

- L'adozione del protocollo I2S, per migliorare la qualità e la fedeltà dell'acquisizione del segnale audio.
- L'integrazione di una scheda microSD, per memorizzare tracce audio e consentire analisi successive con il dispositivo stesso o altri strumenti.
- L'utilizzo di un microfono MEMS, che semplificherebbe l'acquisizione audio diretta, garantendo al contempo compattezza e precisione.
- L'implementazione di un display TFT a colori, per migliorare la chiarezza e l'intuitività della rappresentazione visiva dello spettro audio.

Questi miglioramenti, oltre a rafforzare l'efficacia e la fruibilità del sistema, aprono la strada a molteplici applicazioni pratiche, dall'uso educativo e didattico alla progettazione audio professionale. Inoltre, consentirebbero di soddisfare esigenze sempre più complesse, mantenendo al contempo una struttura compatta e accessibile.

In sintesi, il progetto ha dimostrato il valore di un approccio integrato alla progettazione di strumenti elettronici, combinando creatività, ricerca e soluzioni tecniche efficaci. Le estensioni prospettate rappresentano un naturale completamento di questo lavoro, permettendo di rendere il dispositivo un punto di riferimento per applicazioni pratiche nel campo dell'analisi audio.