Lab 2: Sub-band Coding based on FFT

Adrià Hermini Gonzalez Arnés NIA: 182497

1. Introducció

L'objectiu d'aquesta pràctica és simular la codificació que tindria un senyal d'àudio a l'enviar-se, comprimir-se/descomprimir-se i altres processos. Basada en la DFT (Discrete Fourier Transform) per bandes.

2. Procés

En els següents passos es detalla com s'ha fet aquest procés:

- Ex 1: Transformació del Bloc
- Ex 2: Disseny de les bandes de freqüència
- Ex 3: Assignació i quantificació de bits fixes
- Ex 4: Enfinestrat i Solapament
- Ex 5: Variabilitat dels bits

2.1. Transformació del Bloc

La primera part consisteix a implementar les bases del programa on inicialment s'aplicarà la FFT i la IFFT per a recuperar la senyal original.

Llegirem la senyal inicial i la seva freqüència, guardant-les en el nostre cas amb les variables *audio* i *fsaudio*. Com l'àudio no és constant, es defineix la mida de les finestres en les quals l'àudio es dividirà per a fer la FFT (és important tenir en compte que la mida de les finestres ha de ser un nombre en potència de dos perquè la FFT ho requereix). En el cas d'aquest programa una bona mesura és de 1024 mostres.

En la següent imatge es pot apreciar de quina manera recuperem la senyal original o àudio, on h equival a un frame de 1024 mostres on finalment se sumen entre ells i es recupera la senyal original.

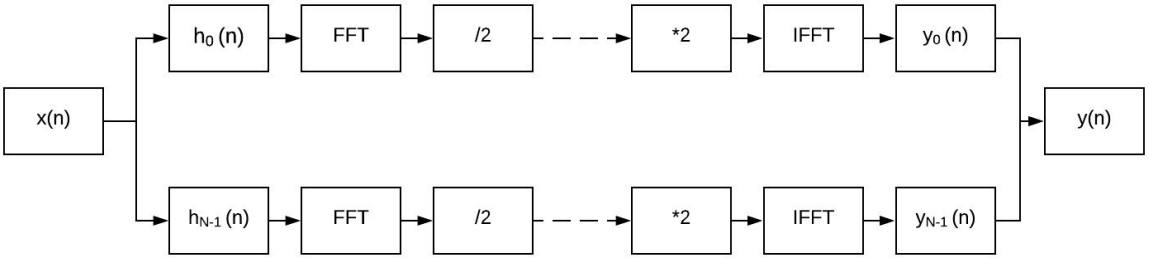

Figura 1: *El diagrama de blocs de la transformació del audio.*

En el codi aquesta implementació es troba en la classe: *block_transform* funció *transform*, sense tenir la part de la quantització en la qual s'entrarà més endavant.
I a continuació es pot apreciar la forma que té el frame número 55, tant en temps com magnitud i fase.

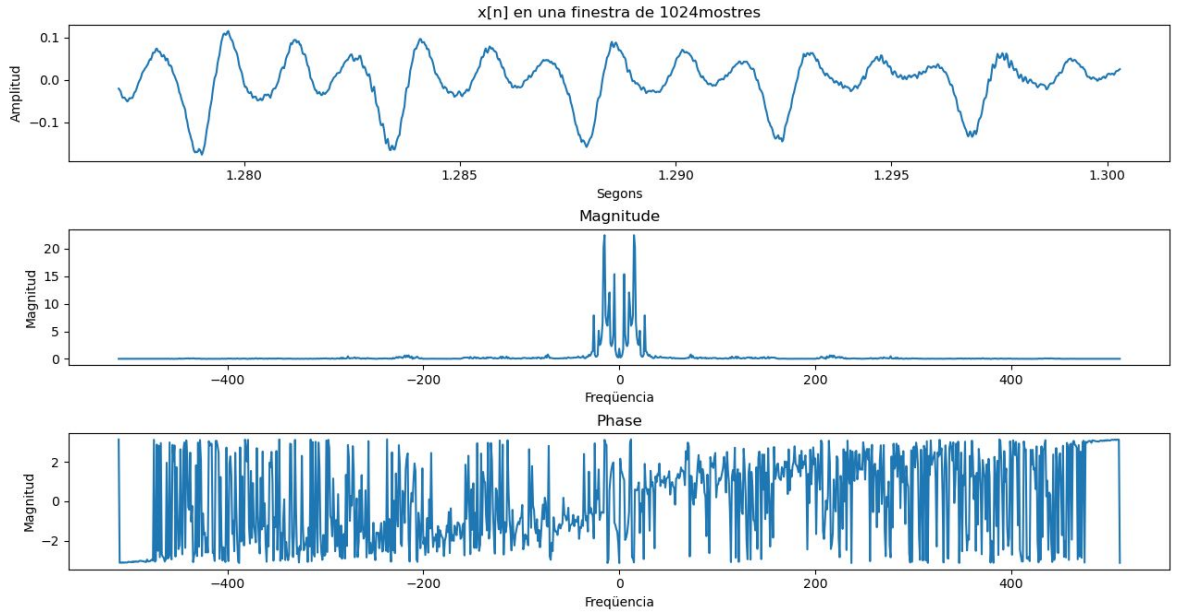

Figura 2: *Plot d'un frame de la senyal, en temps, magnitud i phase.*

2.2. Disseny de les bandes de freqüència

Ara comença la quantització i codificació del programa, ja que fins al moment podem dir que l'àudio només es transforma fins al domini freqüencial. En aquest apartat dividirem la senyal del frame en diferents bandes per a poder-les codificar sense gran problema.

Com diu a l'enunciat, en el cas de treballar amb àudio, les bandes es divideixen seguint una escala logarítmica. Això és degut al que la nostra oïda no és lineal, sinó logarítmica.
En aquest cas, `N = len(freqFram),` ja que `freqFrame` és el midareduït de la FFT.

Per a dissenyar les bandes de freqüència, s'utilitza les següents línies de codi:

```
beginBand = 0

for iBand in range(0, nBands):
    endBand = int(N/pow(2, nBands-iBand))
```

2.3. Assignació i quantització de Bits

Seguint l'apartat anterior, per a cada banda, hi ha una amplitud de codificació determinada, que va disminuint en funció que canvia de banda. A continuació es veu la línea de codi que correspon a l'amplitud de cada banda:

```
        max_amplitude = np.sqrt(N/pow(2, iBand))
```

Aquests dos últims apartats es troben en la funció `quantization` de la classe `quantization`.

Els resultats que he obtingut són bastant acurats al resultat final tot i que amb lleugeres modificacions, per a poder escolar el resultat i apreciar la diferència només s'ha de reproduir el fitxer WaveOut_quantizet.wav.

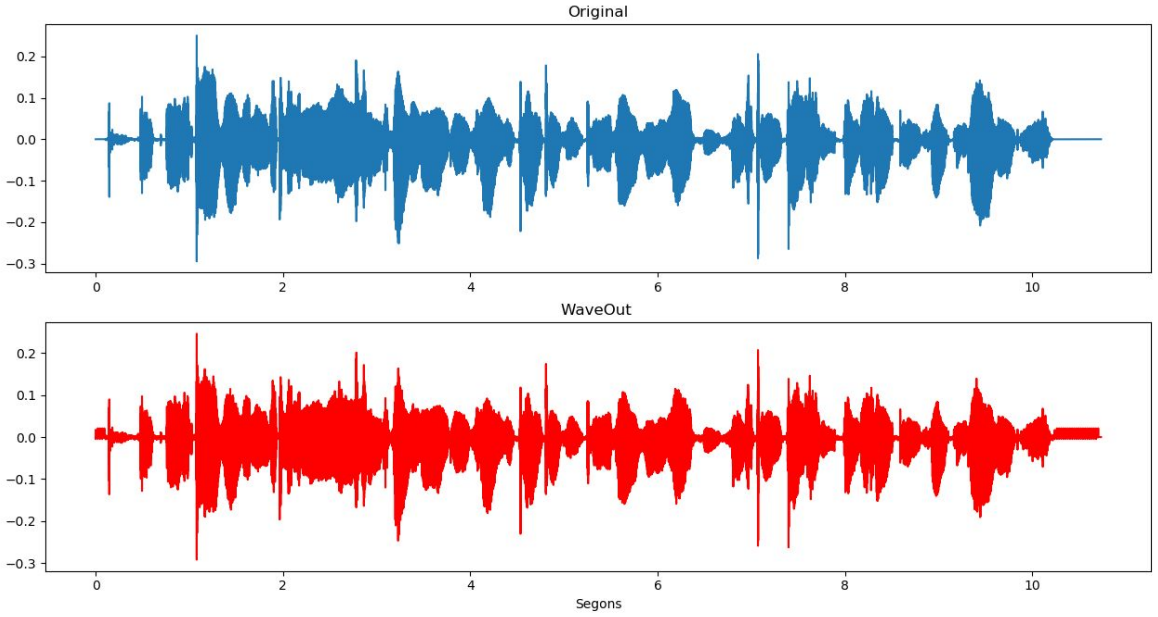

Figura 3: *Ona original (blau) i de la seva reconstrucció (vermell). Amb 8 bits i finestra de 1024.*

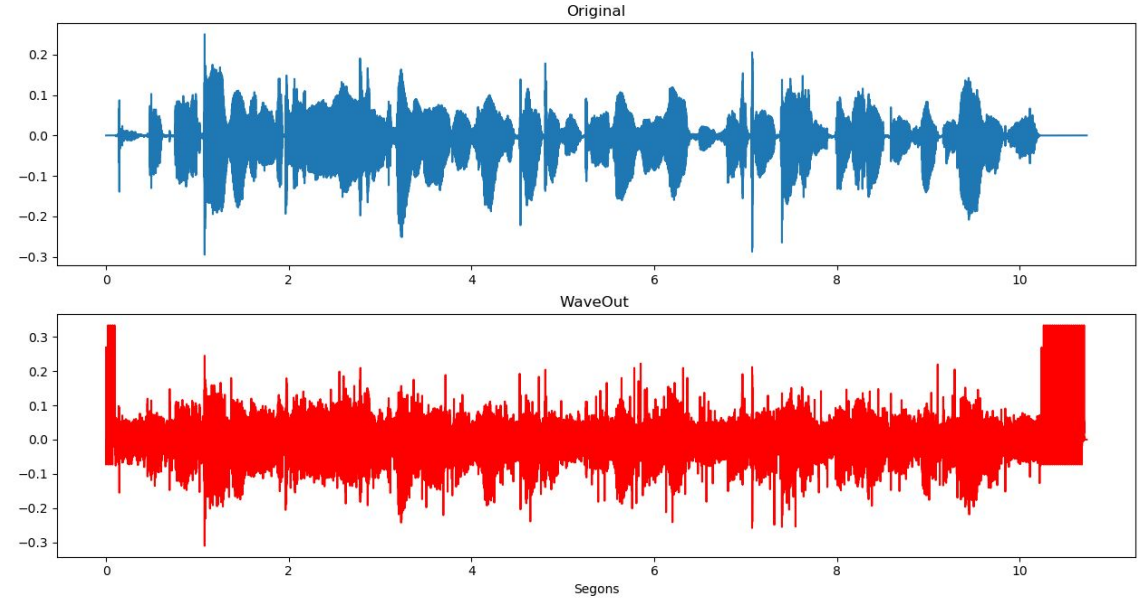

Figura 4: *Ona original (blau) i de la seva reconstrucció (vermell). Amb 4 bits i finestra de 1024.*

En les dues últimes figures es pot observar perfectament que codificar els àudios amb 4 bits és molt pitjor que fer-ho amb 8 bits. I tot i que amb 8 bits el resultat obtingut sembla bastant bo encara es pot millorar, sobretot en l'inici i el final de l'audio.

2.4. Enfinestrat i Overlap o (solapament)

Per a evitar els artefactes que se senten fins ara, usarem un overlap factor a l'hora de passar la finestra, la qual no serà rectangular. En aquest cas, s'obtenen bons resultats amb una finestra hamming i un overlap factor major a 0,5.
L'overlap factor indica quant de tros del senyal enfinestrat en el frame previ estarà inclòs en l'enfinestrament del frame actual. Aquestes repeticions en la informació faran que el senyal es reconstrueixi de forma molt més fidel.

Primer de tot hem de tornar a computar el nombre de frames, tenint en compte l'overlap factor:

```
nFrames = int(len(audio)/(winL*(1-overlap)))-int(1/(1-overlap))
```

Amb aquest càlcul ens assegurem que el nombre de frames no es passi, per a poder mantenir la longitud de l'audio original.

A més, com a conseqüència lògica d'usar "repeticions de data", es comptarà amb més informació per cada mostra.

Finalment, una vegada hem dut a terme el mateix procés que a l'exercici anterior (quantització, ...), es farà un desenfinestrament multiplicant la finestra de sortida per una nova finestra hamming, que en aquest cas serà la mateixa que la finestra inicial.

```
# Overlap-add
frameOut_w = frameOut*window
waveOut[beginFrame:endFrame] = waveOut[beginFrame:endFrame]+frameOut_w
```

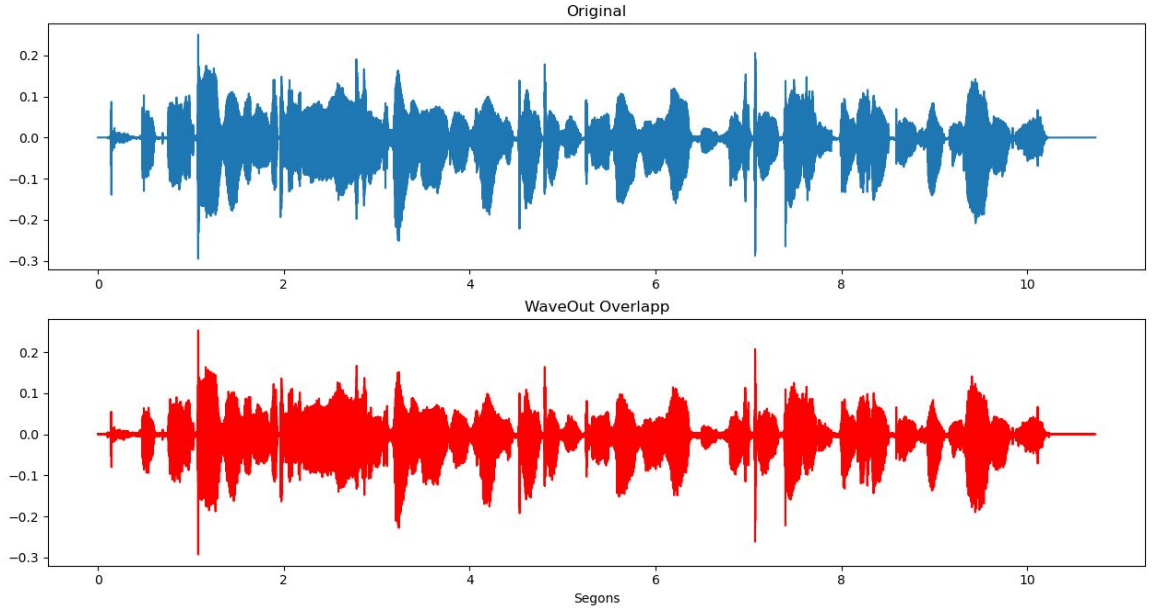

Figura 5: *Ona original (blau) i de la seva reconstrucció (vermell). Amb 8 bits i finestra de 1024 i overlap 0.5.*

2.5. Variabilitat de bits

Per estalviar bits, memòria, l'exercici 5 proposa no codificar les bandes que tinguessin menys informació o informació nulla, posant un llindar anomenat `energyThreshold` que a partir de l'amplada de les bandes de codificació descartava si valia la pena o no codificar-lo amb la següent línia de codi:

```
# Coder
if max(np.abs(freqFrame)) > (max_amplitude/energyThreshold):
```

Amb `energyThreshold = 6`:

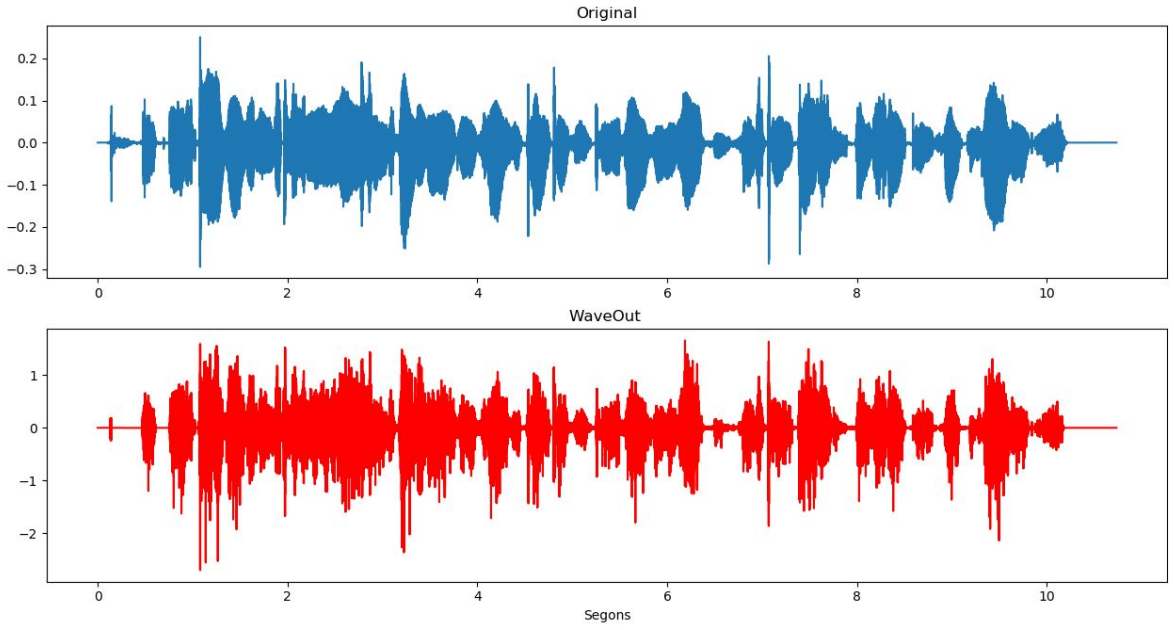

Figura 6: *Ona original (blau) i de la seva reconstrucció (vermell). Amb 8 bits i finestra de 1024 i overlap 0.5.*

I el recompte de bits dona:

```
BITRATE exercici 3: 352.8  Kbits/s
BITRATE exercici 4 (Overlap-Add): 704.41 Kbits/s
BITRATE exercici 5 (Variable): 288.58 Kbits/s
```

El resultat no és el esperat, ja que no acaba de tenir una bona reconstrucció. Però la reducció de bits comparat amb l'exercici 4 és considerable.

La implementació del codi es troba en la funció `variable_quantization` de la classe `quantization`.

3. Problemes

Aquesta pràctica l'havia fet prèviament amb matlab durant el curs i aquest és un treball per a recuperar l'assignatura ja que m'ha quedat penjada, com a tal, a la hora de traduir el codi i reformular-lo m'he trobat amb dos problemes que fins el moment enacra no he reslot i m'hagradaria obtenir resposta.

- Primer: En python hi han dues maneres de calcular la FFTi la IFFT, a traves de la classe `np.fft` o a través de la classe `skipy.fftpack`. La gran diferencia entre una i l'altre es que les funcions de la classe `skipy.fftpack` són més acurades, i contra la lógica que això implica els resultats són molt pitjors que no pas amb la np.fft. I per aquesta raó he utilitzat la classe `numpy`.

- Segon: La manera en la que he calculat els bits que he utilitzat durant el Ex 5:

```
# Coder
if max(np.abs(freqFrame)) > (max_amplitude/energyThreshold):
    # Real part: real(fft)
    data_real = np.real(freqFrame[beginBand:endBand])
    _, Qlevel_real = quantimaxmin(data_real, n_bits, max_amplitude, -max_amplitude)
    numbits = numbits + n_bits*len(data_real)

    # Imaginary part: img(fft)
    data_img = np.imag(freqFrame[beginBand:endBand])
    _, Qlevel_img = quantimaxmin(data_img, n_bits, max_amplitude, -max_amplitude)
    numbits = numbits + n_bits*len(data_img)

    reciveBand = True
else:
    numbits = numbits+1
    reciveBand = False
```

numbits → el recompte dels bits que es necessiten per a codificar l'àudio. És correcte, o en el decoder també hauria de fer el mateix?