ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Ψηφιακή Επεξεργασία Σήματος

2Η ΕΡΓΑΣΤΗΡΙΑΚΗ ΆΣΚΗΣΗ

Αναστάσιος Στέφανος Αναγνώστου 03119051
Σπυρίδων Παπαδόπουλος 03119058
5/6/2022

Πίνακας περιεχομένων

Μέρος 1

ΒΗΜΑ 1.0

Στο πρώτο βήμα, αφού μετατραπεί το σήμα από stereo σε mono, διαιρείται σε παράθυρα των 512 δειγμάτων. Αξιοσημείωτο είναι ότι, επειδή το πλήθος των δειγμάτων του αρχικού σήματος δεν είναι πολλαπλάσιο του 512, η διαίρεση είναι ατελής, με αποτέλεσμα το τελευταίο παράθυρο να έχει μήκος 149 δειγμάτων. Εντούτοις, αυτό δεν αποτελεί πρόβλημα, καθώς μπορεί κανείς είτε να συμπληρώσει τα υπολειπόμενα δείγματα με μηδενικά, είτε να συνεχίσει ανενόχλητος στο επόμενο βήμα υπολογισμού του φάσματος ισχύος 512 δειγμάτων του παραθύρου, όπου θα προστεθούν τα μηδενικά ούτως ή άλλως. Στην παρούσα εργασία ακολουθείται η δεύτερη προσέγγιση. Κατά την διάρκεια της επεξεργασίας χρησιμοποιείται το 780ο παράθυρο του σήματος για επισκόπηση των αποτελεσμάτων.

Αρχικά, ορίζονται κάποια βασικά μεγέθη για την ολοκλήρωση των ερωτημάτων και κάποιες συναρτήσεις:

```
music_signal, music_srate = soundfile.read("music.wav")

music_signal = np.transpose(music_signal)
if music_signal.ndim == 2:
    music_signal = (music_signal[1]+music_signal[0])/2
music_signal = music_signal / abs(max(music_signal))

music_length = len(music_signal)
N = 512
MUL_N = [N*x for x in list(range(len(music_signal)//N+1))]
NUM_WINDOWS = int(np.ceil(len(music_signal)/N))
PN = 90.302  # dB
M = 32
B = 16  # number of bits used for the encoding of each signal sample
R = 2**B  # number of intensity levels of the original signal
# central frequency of each of M=32 filters
Fk = [(2*k-1)*music_srate/4/M for k in range(1, M+1)]
```

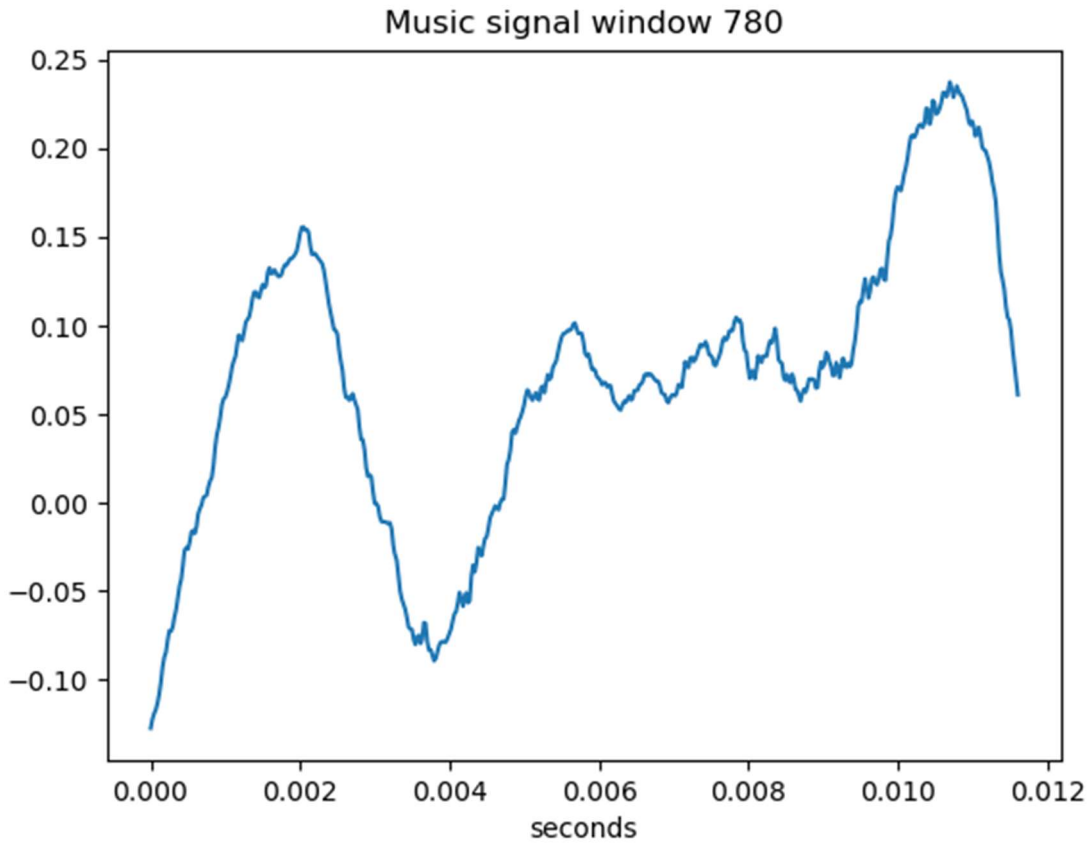
```
def ath(freq):

    """Calculate the Absolute threshold of hearing."""
    return (3.64*(freq/1000)**(-0.8) -
            6.5 * np.exp((-0.6)*(freq/1000-3.3)**2)
            + (0.001)*(freq/1000)**4)


def bark(freq):
    """Convert Hz to Bark."""
    return 13*np.arctan(0.00076*freq)+3.5*np.arctan((freq/7500)**2)


def itof(k, samples=N, s_rate=music_srate):
    """Convert discrete frequency to natural frequency."""
    return s_rate * (k+1) // samples
```

Για τον υπολογισμό των βοηθητικών πινάκων

```
# These arrays are helpful for speeding up some computations


itofr = [itof(k) for k in range(N//2)]  # index to natural frequency

aths = [ath(freq) for freq in itofr]    # absolute thresholds of hearing

barks = [bark(freq) for freq in itofr]  # index to bark frequency

domains = [[f for f in range(N//2)
            if ((2*k-1)*music_srate/4/M - music_srate/4/M <=
                itofr[f]
                <= (2*k-1)*music_srate/4/M + music_srate/4/M)]
           for k in range(1, M+1)]  # frequency domains for the filter in 2.3
```

οι οποίοι θα φανούν χρήσιμοι σε επόμενα ερωτήματα και για την επιτάχυνση του κώδικα.

Παρατίθεται το 780ο παράθυρο 512 δειγμάτων του κανονικοποιημένου μουσικού σήματος.



Εικόνα 1 Το 780 παράθυρο N=512 δειγμάτων του δεδομένου μουσικού σήματος.

ΒΗΜΑ 1.1

Στο βήμα αυτό υπολογίζεται το φάσμα ισχύος 512 δειγμάτων καθενός παραθύρου εκπεφρασμένο σε μονάδες SPL (Sound Pressure Level) σύμφωνα με τον τύπο:

$$P(k) = PN + 10 \cdot \log_{10} \left| \sum_{n=0}^{N-1} w(n) \cdot x(n) \cdot e^{-\frac{2\pi k n}{N}} \right|^2 , 0 \le k < \frac{N}{2}$$

Όπου $PN = 90.302\, dB$ και $N = 512$. Το χρησιμοποιούμενο παράθυρο $w(n)$ είναι το παράθυρο Hanning:

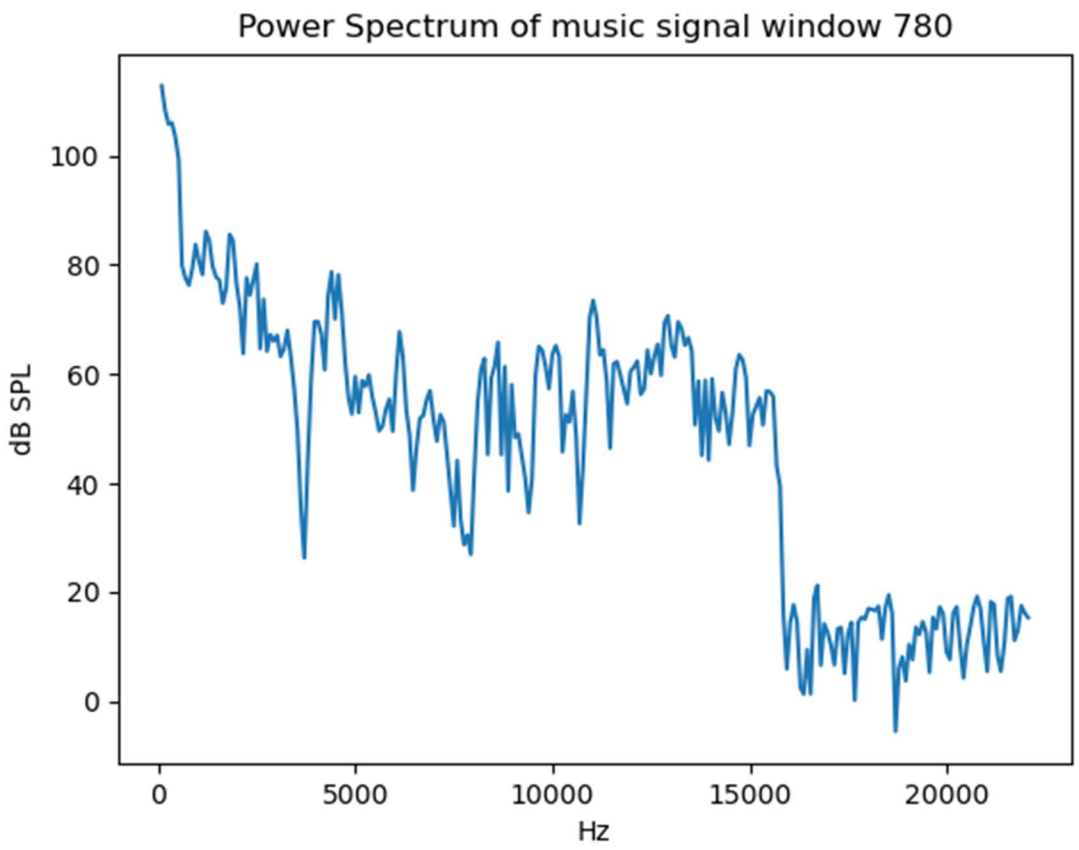
$$w(n) = \frac{1}{2} \cdot \left[1 - \cos\left(\frac{2\pi n}{N}\right)\right]$$

```
def power_spec(insig, points):

    """Calculate the power spectrum of insig using "points" points."""
    return halfit(PN+10*np.log10(
        abs(np.fft.fft(insig*np.hanning(len(insig)), points))**2))
```

```
def halfit(insig):

    """Half the insig."""
    return insig[0:len(insig)//2]
```

Η συνάρτηση `power_spec(insig, points)` υπολογίζει το φάσμα ισχύος του σήματος εισόδου και χρήσει της βοηθητικής `halfit(insig)` κρατάει το μισό του συμμετρικό αντί για ολόκληρο το σήμα.

Παρατίθεται το φάσμα ισχύος για το 780^{o} παράθυρο:



Εικόνα 2 Φάσμα ισχύος (μισό λόγω άρτιας συμμετρίας) του παραθύρου 780.

Φαίνεται από το φάσμα ισχύος ότι, περισσότερη ενέργεια σημειώνεται στην περιοχή συχνοτήτων $[0,15000]$ Hz, ενώ στις μεγαλύτερες συχνότητες η ενέργεια φθίνει αισθητά.

ΒΗΜΑ 1.2

Στη βήμα αυτό υπολογίζονται τα τοπικά μέγιστα, και οι θέσεις αυτών, των φασμάτων ισχύων, τα οποία εν προκειμένω καλούνται τονικές μάσκες.

Αρχικά, υλοποιούνται οι συναρτήσεις της εκφώνησης:

```
def mask_band(k):

    """
    Mask band.

    Return the distances away from "k"
    that should be checked in order to decide if there is
    a mask in "k" or not.
    """
    if 2 < k < 63:
        return [2]
    if 63 <= k < 127:
        return [2, 3]
    if 127 <= k < 250:
        return [2, 3, 4, 5, 6]


def ismask(power_spectrum, k):
    """Check if point is mask."""
    return ((not (k < 3 or k > 249)) and
            (power_spectrum[k] > power_spectrum[k-1] and
             power_spectrum[k] > power_spectrum[k+1] and
             (not (False in [power_spectrum[k] > power_spectrum[k+pos]+7 and
                             power_spectrum[k] > power_spectrum[k-pos]+7
                             for pos in mask_band(k)]))))


def find_mask_positions(power_spectrum):
    """Find positions of masks."""
    return [x for x in range(len(power_spectrum)) if ismask(power_spectrum, x)]


def mask_power(power_spectrum, k):
    """
    Mask power.

    Returns the power of the mask in "k".
    """
    if not ismask(power_spectrum, k):
        return 0
    return (10*np.log10(10**(0.1*power_spectrum[k-1]) +
                        10**(0.1*power_spectrum[k]) +
                        10**(0.1*power_spectrum[k+1])))
```

Η βοηθητική συνάρτηση `mask_band(k)` επιστρέφει την γειτονιά εντός της οποίας πρέπει να αποτιμηθεί η συνθήκη για να χαρακτηριστεί ένα σημείο μάσκα ή όχι. Η δε βοηθητική συνάρτηση `ismask(power_spectrum, k)` υλοποιεί ακριβώς αυτήν την συνθήκη, Η συνάρτηση `find_mask_positions(power_spectrum)` επιστρέφει τις θέσεις των τονικών μασκών και τέλος η `mask_power(power_spectrum, k)` υπολογίζει την ισχύ τους βάσει του δεδομένου τύπου.

Για το επισκοπούμενο παράθυρο 780^{ο} , τα αποτελέσματα είναι τα εξής:

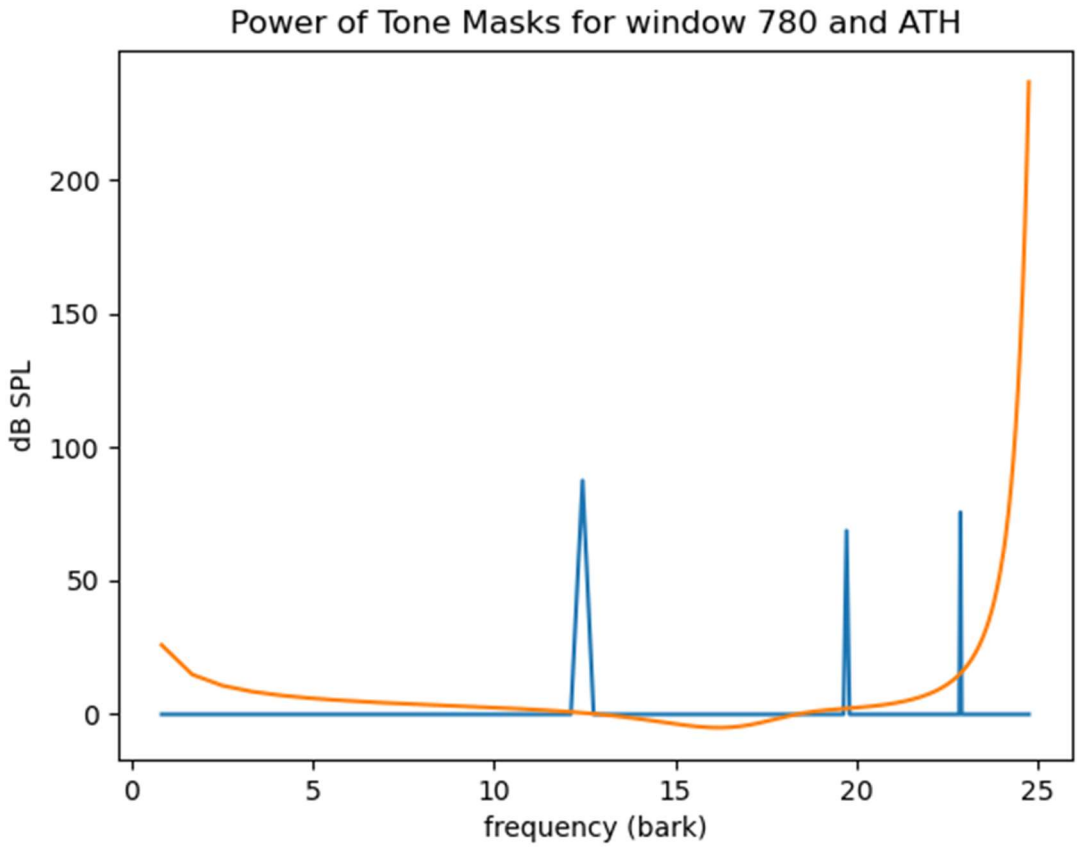
Θέσεις Τονικών Μασκών	Τιμές Τονικών Μασκών (dB SPL)
20	88.29428415267054
46	73.73635164910874
70	69.55844303697125
127	76.43969586776736

Οι δε δεδομένες τονικές μάσκες, κατόπιν μειώσεως είναι οι:

Θέσεις Τονικών Μασκών	Τιμές Τονικών Μασκών (dB SPL)
20	87.7408355727181
70	69.00499445701881
127	75.88624728781488

Απ' ό,τι φαίνεται, η τονική μάσκα 46 αφαιρείται με την διαδικασία μειώσεως των μασκών και οι υπόλοιπες μάσκες συμφωνούν τόσο στις τιμές όσο και στις θέσεις.

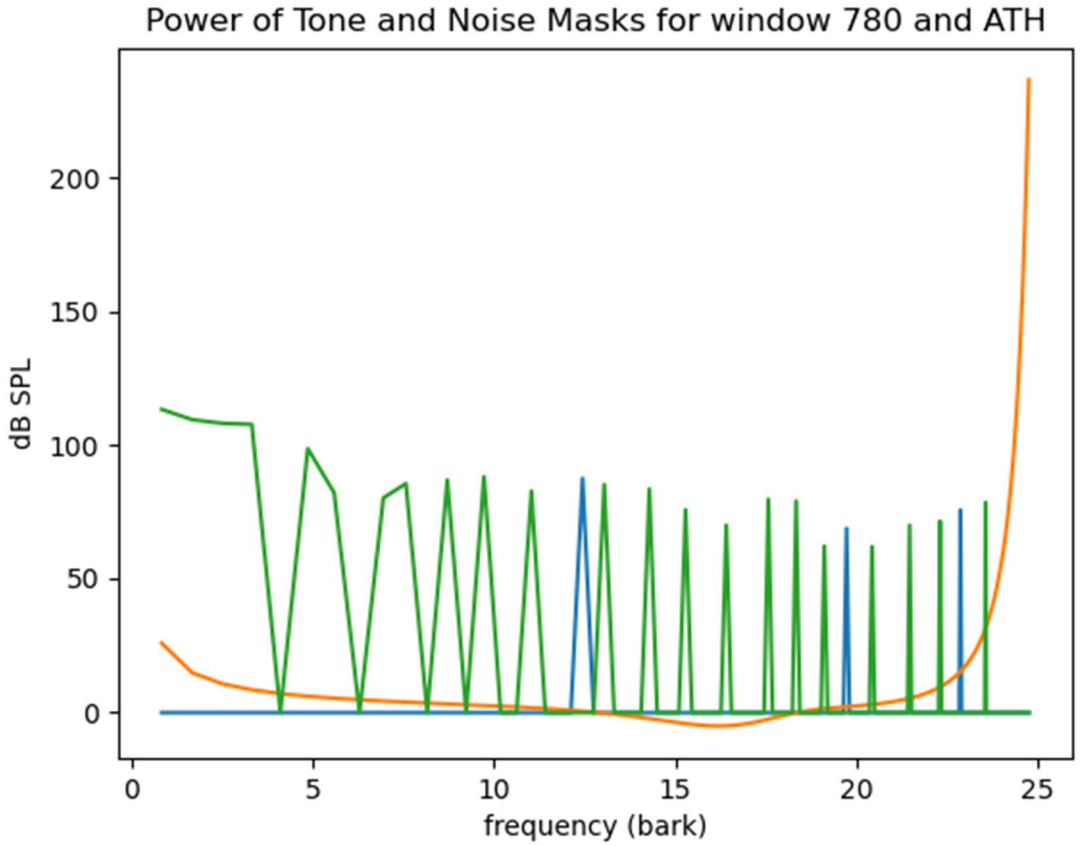
Παρατίθεται διάγραμμα ισχύος των μειωμένων τονικών μασκών του 780^{ου} παραθύρου εν συγκρίσει με το απόλυτο κατώφλι ακοής για τις συχνότητες σε κλίμακα bark.



Εικόνα 3 Ισχύς των μειωμένων τονικών μασκών του παραθύρου 780.

ΒΗΜΑ 1.3

Σε αυτό το βήμα δίνονται έτοιμοι οι πίνακες μειωμένων τονικών μασκών και μασκών θορύβου. Παρατίθεται το διάγραμμα των μασκών αυτών του 780ου παραθύρου μαζί με το απόλυτο κατώφλι ακοής των συχνοτήτων σε κλίμακα bark.



Εικόνα 4 Ισχύς μειωμένων τονικών και θορυβικών μασκών του παραθύρου 780.

ΒΗΜΑ 1.4

Στο βήμα αυτό υπολογίζονται τα διάφορα κατώφλια κάλυψης. Ο υπολογισμός γίνεται στο παρακάτω τμήμα κώδικα:

```
P_NM = np.load("P_NM.npy")
P_TMc = np.load("P_TMc.npy")
P_NMc = np.load("P_NMc.npy")

transposeP_TMc = np.transpose(P_TMc)
J_TM = [[j for j, toneMask in enumerate(transposeP_TMc[s]) if toneMask > 0]
        for s in range(NUM_WINDOWS)]    # indexes of tone masks

spectrarum_masks = [[row[s] for row in P_TMc] for s in range(NUM_WINDOWS)]
T_TM = [[[imt(i, j, spectrarum_masks[s], "TM") for j in J_TM[s]]
        for i in range(N//2)]
        for s in range(NUM_WINDOWS)]    # individual masking thresholds

transposeP_NMc = np.transpose(P_NMc)
J_NM = [[j for j, noiseMask in enumerate(transposeP_NMc[s]) if noiseMask > 0]
        for s in range(NUM_WINDOWS)]    # indexes of noise masks

spectrarum_masks = [[row[s] for row in P_NMc] for s in range(NUM_WINDOWS)]
T_NM = [[[imt(i, j, spectrarum_masks[s], "NM") for j in J_NM[s]]
        for i in range(N//2)]
        for s in range(NUM WINDOWS)]    # individual masking thresholds
```

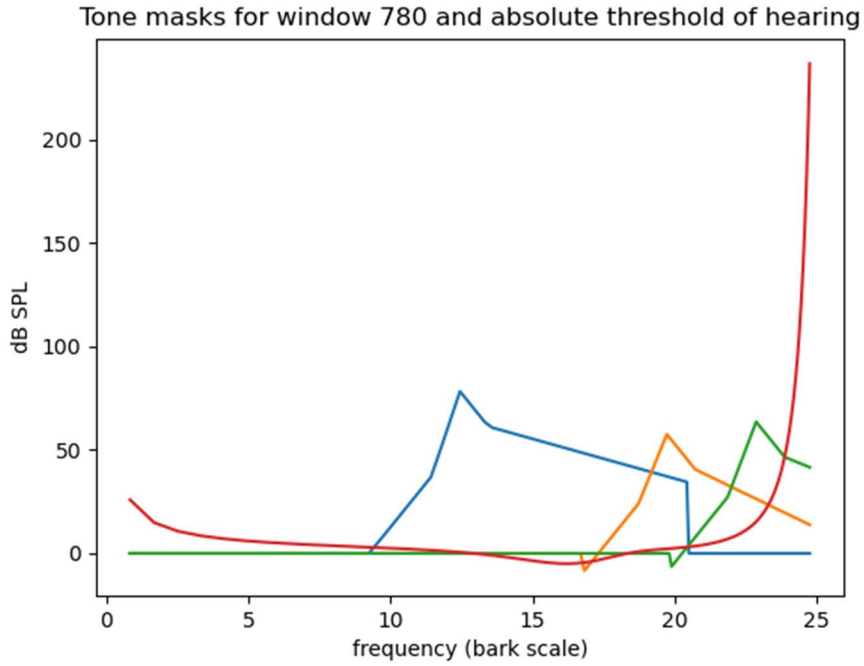
Κατ' αρχάς, φορτώνονται οι δεδομένοι πίνακες. Στην συνέχεια, βρίσκονται οι θέσεις των μη μηδενικών μασκών ανά παράθυρο. Χρησιμοποιούνται οι θέσεις αυτές για την αποθήκευση των ίδιων των μασκών στον πίνακα ... . Τέλος, ανά παράθυρο, ανά δείγμα, ανά τονική μάσκα, υπολογίζεται το ατομικό κατώφλι άλυψης.

Η ίδια διαδικασία ακριβώς επαναλαμβάνεται για τον υπολογισμό των θορυβικών μασκών.

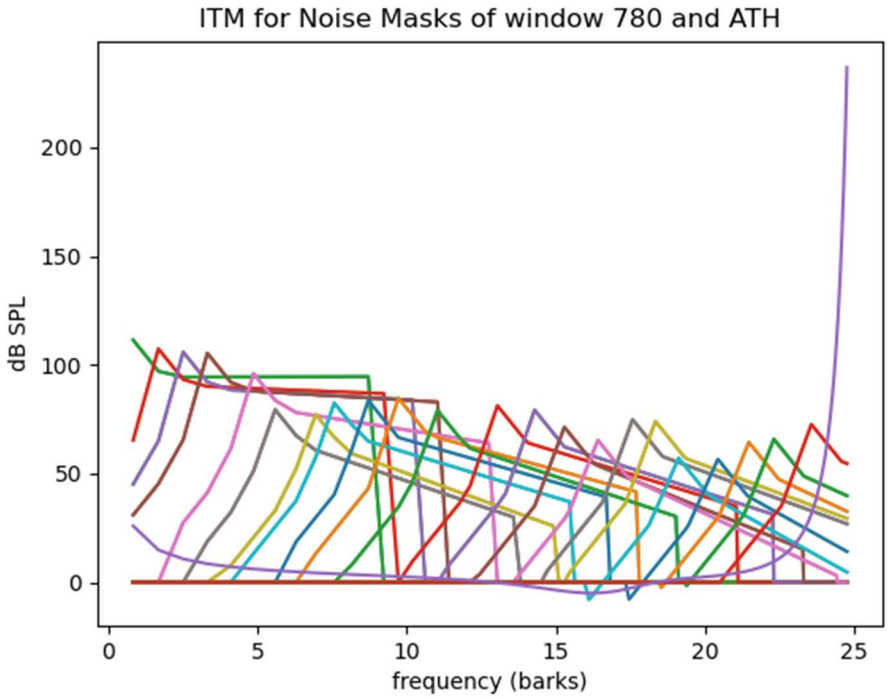
Ο υπολογισμός του κατωφλίου γίνεται από την αντίστοιχη συνάρτηση, η οποία υλοποιεί ακριβώς την αντίστοιχη συνάρτηση της εκφώνησης:

```
def imt(pos_i, pos_j, masks, flag):

    """
    Individual masking thresholds.

    Returns the amount of covering in point i from the tone or
    noise mask in point j.
    """
    if barks[pos_i]-barks[pos_j] > 8 or barks[pos_i]-barks[pos_j] < -3:
        return 0

    def sf(pos_i, pos_j, masks):
        """
        Help function.

        Minimum power level that neighbouring frequencies must have,
        so that both of them are perceptible by a human.
        """
        delta_bark = barks[pos_i]-barks[pos_j]
        if delta_bark >= 8 or delta_bark < -3:
            return 0
        if -3 <= delta_bark < -1:
            return 17*delta_bark-0.4*masks[pos_j]+11
        if -1 <= delta_bark < 0:
            return delta_bark*(0.4*masks[pos_j]+6)
        if 0 <= delta_bark < 1:
            return -17*delta_bark
        return delta_bark*(0.15*masks[pos_j]-17)-0.15*masks[pos_j]
    if flag == "TM":
        return masks[pos_j]-0.275*barks[pos_j]+sf(pos_i, pos_j, masks)-6.025
    return masks[pos_j]-0.175*barks[pos_j]+sf(pos_i, pos_j, masks)-2.025
```

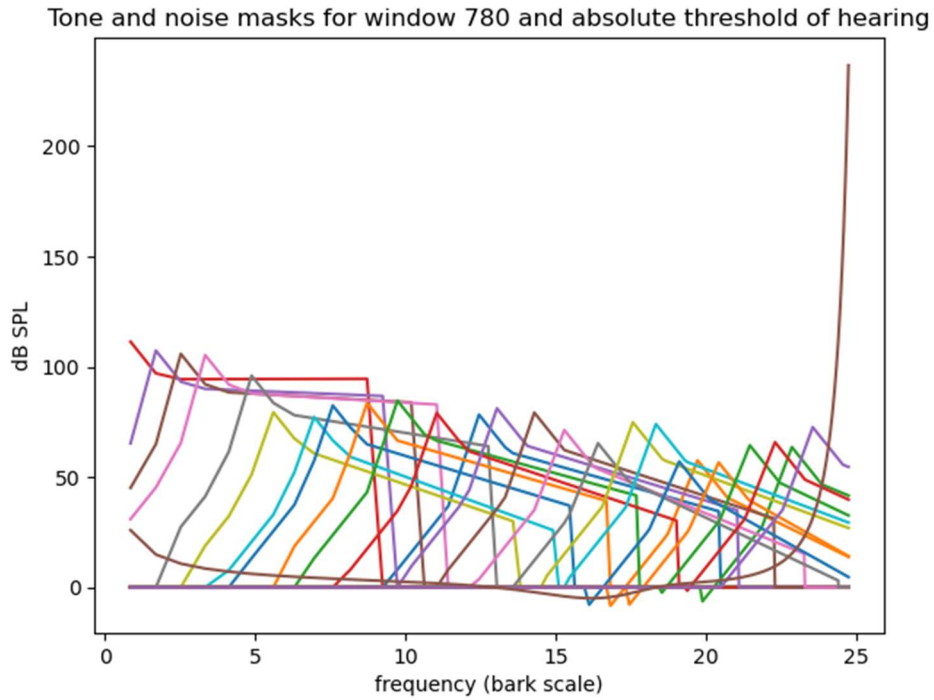
Παρατίθενται τα αποτελέσματα για το 780ο παράθυρο.



Εικόνα 5 Ατομικά Κατώφλια Κάλυψης για τις τονικές μάσκες παραθύρου 780 εν συγκρίσει με το απόλυτο κατώφλι ακοής.



Εικόνα 6 Ατομικά Κατώφλια Κάλυψης μασκών θορύβου για το παράθυρο 780 εν συγκρίσει με το απόλυτο κατώφλι ακοής.



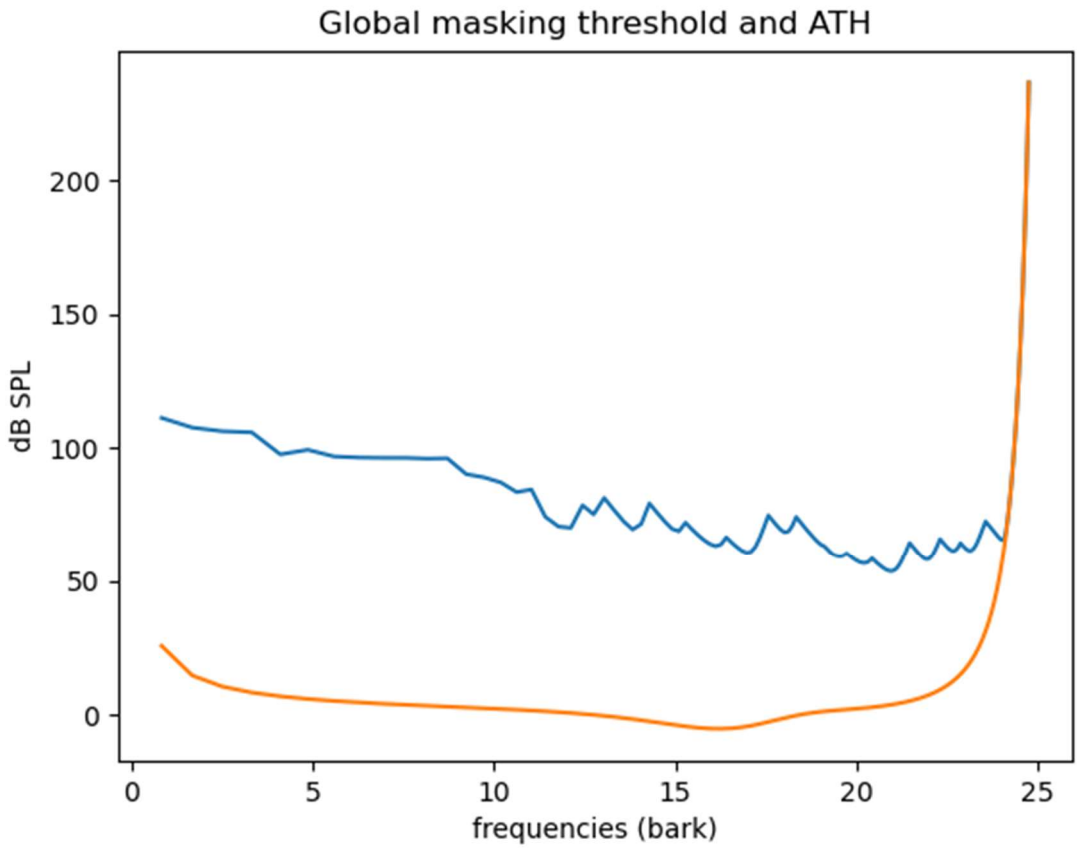
Εικόνα 7 Ατομικά Κατώφλια Κάλυψης για τις τονικές και θορυβικές μάσκες του παραθύρου 780 εν συγκρίσει με το απόλυτο κατώφλι ακοής.

ΒΗΜΑ 1.5

Στο βήμα αυτό συνδυάζονται τα υπολογισμένα ατομικά κατώφλια κάλυψης των τονικών και θορυβικών μασκών και το απόλυτο κατώφλι ακοής των συχνοτήτων ώστε να προκύψει ένα συνολικό κατώφλι κάλυψης για κάθε διακριτή συχνότητα ξεχωριστά.

```
def gbm(k, tone_thresholds, noise_thresholds):
    """Calculate the Global Masking Threshold."""
    return 10*np.log10(
        10**(0.1*aths[k]) +
        sum([10**(0.1*(tone_thresholds[k][q]))
             for q in range(len(tone_thresholds[k]))]) +
        sum([10**(0.1*(noise_thresholds[k][m]))
             for m in range(len(noise_thresholds[k]))]))


# global masking thresholds

spectrarum_thresholds = [[gbm(i, T_TM[s], T_NM[s])
                          for i in range(N//2)]
                          for s in range(NUM_WINDOWS)]
```

Παρατίθεται το αποτέλεσμα για το 780ο παράθυρο.



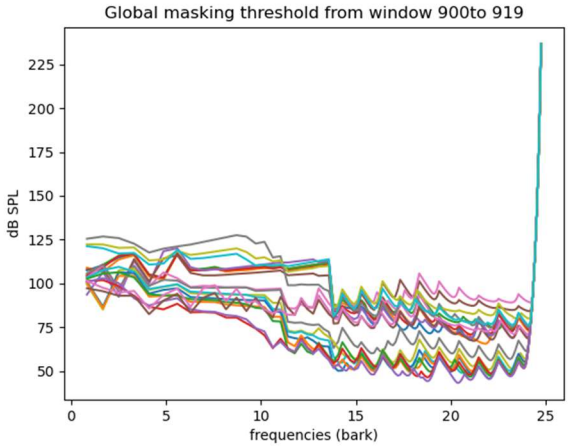
Εικόνα 8 Συνολικό κατώφλι κάλυψης κάθε διακριτής συχνότητας ξεχωριστά για το παράθυρο 780 εν συγκρίσει με το απόλυτο κατώφλι ακοής.

Παρατηρείται ότι, ενώ στα θεωρητικά διαγράμματα[1] σημειώνονται διακριτές κορυφές στην περιοχή των χαμηλών συχνοτήτων, δεν συμβαίνει το ίδιο στα παρόντα αποτελέσματα. Αυτό δικαιολογείται από το γεγονός ότι, οι θορυβικές μάσκες των θεωρητικών διαγραμμάτων είναι αφενός ασθενέστερες (γύρω στα 20-35 dB SPL) από τις αντίστοιχες τονικές (20-50 dB SPL), είναι αφετέρου συσσωρευμένες στις υψηλές συχνότητες αντί στις χαμηλές και τέλος είναι σχεδόν ισοπληθείς με τις αντίστοιχες τονικές.

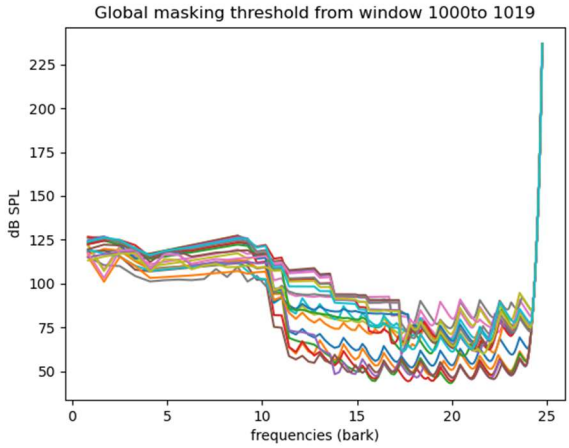
Εν προκειμένω, οι θορυβικές μάσκες είναι σαφώς περισσότερες στο πλήθος, είναι ισχυρότερες και είναι συσσωρευμένες στις χαμηλές συχνότητες, με αποτέλεσμα, αντί να διακρίνονται κορυφές, να ενώνονται κατά τέτοιον τρόπον ώστε να εκδηλώνεται μία ομαλότερη καμπύλη.

Σημειώνεται δε, ότι η ίδια συμπεριφορά εκδηλώνεται και σε άλλα παράθυρα με κάποιες διακυμάνσεις. Άλλοτε είναι εμφανείς κάποιες κορυφές, άλλοτε ενώνονται σε μία ομαλότερη καμπύλη.

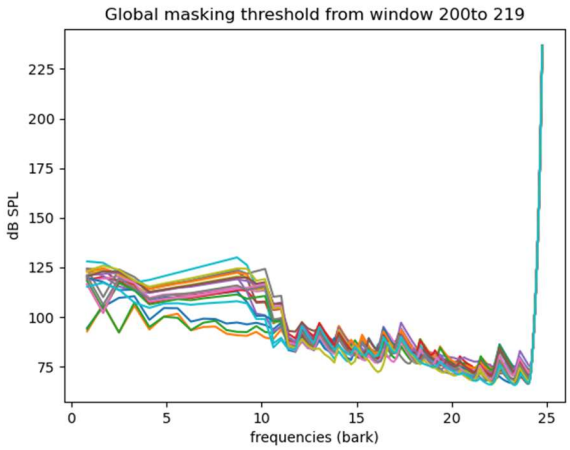
Συγκεκριμένα στα παράθυρα 900-919 παρατηρούνται κορυφές, ενώ στα 1000-1019 παρατηρείται σχετικά ομαλή καμπύλη. Στα παράθυρα 200-219 και 300-319 παρατηρείται ανάμεικτη συμπεριφορά.



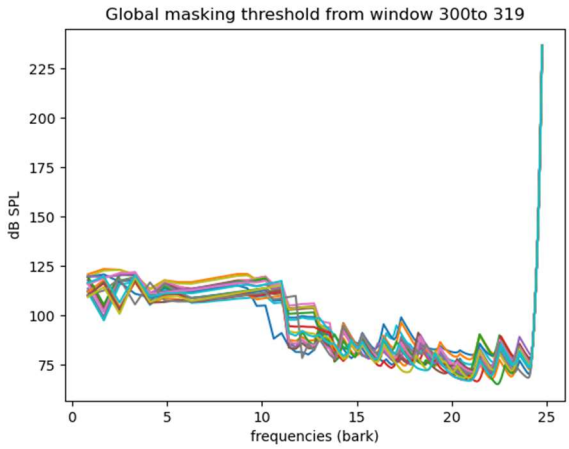
Εικόνα 12 Συνολικά κατώφλια κάλυψης για τα παράθυρα 900 έως 919



Εικόνα 11 Συνολικά κατώφλια κάλυψης για τα παράθυρα 1000 έως 1019



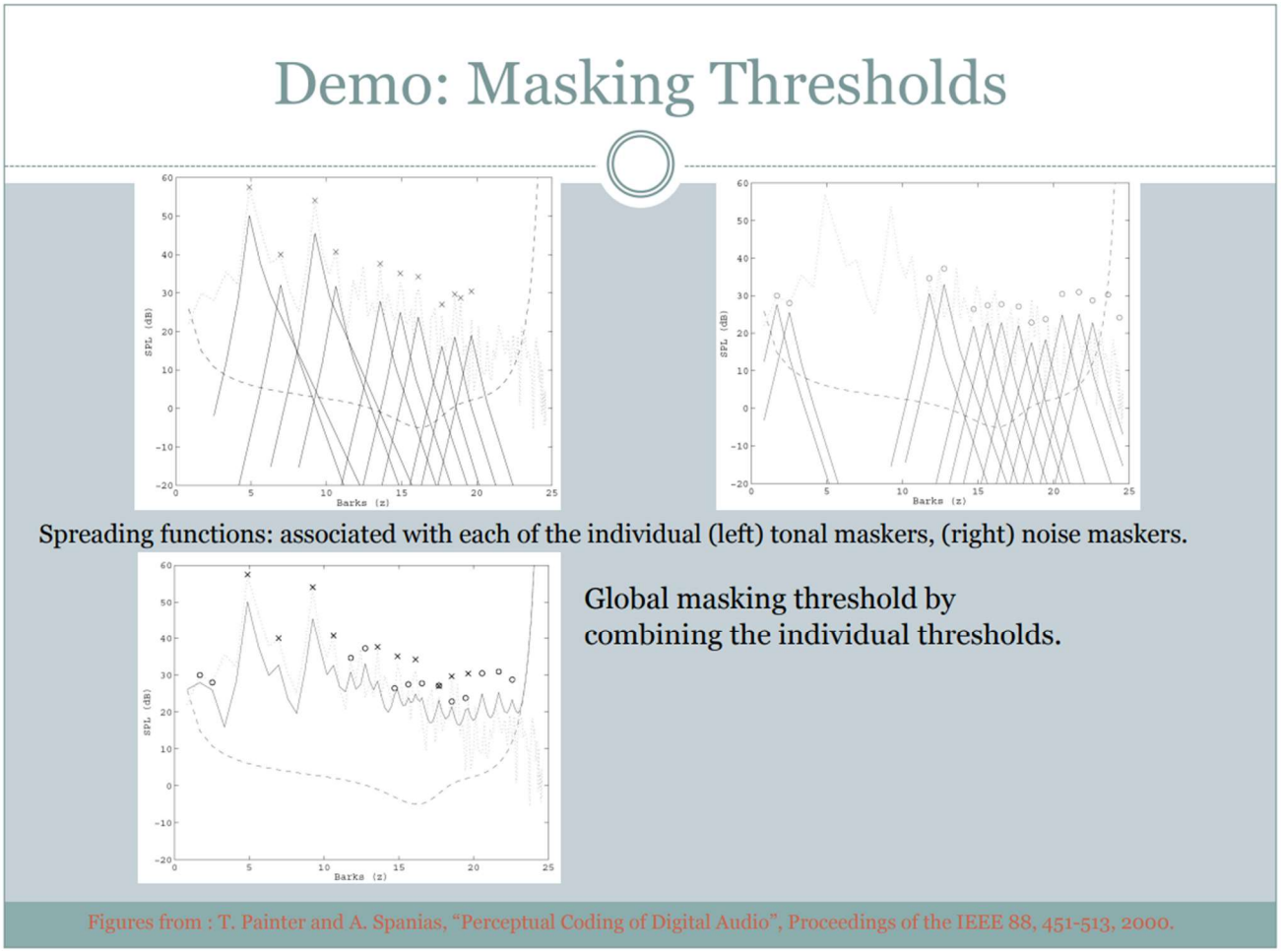
Εικόνα 10 Συνολικά κατώφλια κάλυψης για τα παράθυρα 200 έως 219



Εικόνα 9 Συνολικά κατώφλια κάλυψης για τα παράθυρα 300 έως 319

[1] Τα θεωρητικά διαγράμματα της παρουσίασης της προκείμενης εργαστηριακής άσκησης.

Παρακάτω βρίσκεται η χαρακτηριστική διαφάνεια της παρουσίασης του εργαστηρίου με τα αντίστοιχα θεωρητικά διαγράμματα.



Εικόνα 13 Χαρακτηριστική διαφάνεια από την παρουσίαση του εργαστηρίου

Μέρος 2

ΒΗΜΑ 2.0 ΚΑΙ ΒΗΜΑ 2.1

Στο βήμα αυτό υλοποιείται ο Modified Discrete Cosine Transform (MDCT) για το φιλτράρισμα καθενός παραθύρου του σήματος, μέσω της συνάρτησης `mdct(in_sig, k, m=M, flag="analysis")`.

```
def mdct(in_sig, k, m=M, flag="analysis"):
    """Perform the Modified Discrete Cosine Transform."""
    n = np.linspace(0, 2*m-1, 2*m)
    hk = (np.sin((n+0.5)*np.pi/(2*m)) *
          ((2/m)**(1/2)) *
          (np.cos((2*n+m+1)*(2*k+1)*np.pi/(4*m))))
    if flag == "synthesis":
        # gk = hk*(2*m-1-n)
        gk = hk[::-1]
        return np.convolve(in_sig, gk)
    elif flag == "analysis":
        return np.convolve(in_sig, hk)
    print("wrong flag: either 'analysis' or 'synthesis'")
```

Η συνάρτηση δέχεται ως παραμέτρους το παράθυρο εισόδου, την παράμετρο k, το πλήθος των σημείων m και μία παράμετρο «σημαία» για την επισήμανση της επιθυμητής λειτουργίας, αμφότερες με προκαθορισμένες τιμές “32” και “analysis”.

Στην συνάρτηση ορίζεται ένας γραμμικός χώρος $2 \cdot M = 64$ σημείων από το 0 έως το $2 \cdot M - 1$. Στην συνέχεια, υλοποιούνται οι δεδομένες από την εκφώνηση παραστάσεις και ελέγχεται η επιθυμητή λειτουργία, ανάλυση ή σύνθεση, μέσω της ειδικής μεταβλητής «σημαίας». Εάν έχει επιλεχθεί ανάλυση, τότε πραγματοποιείται η συνέλιξη του παραθύρου εισόδου με την ορισθείσα συνάρτηση, ενώ αν έχει επιλεχθεί η σύνθεση, τότε πραγματοποιείται η συνέλιξη του παραθύρου εισόδου με την αντίστοιχη συνάρτηση σύνθεσης, η οποία είναι η ανάποδη αυτής της ανάλυσης.

Για την υποδειγματοληψία του σήματος συνέλιξης και για την σαφήνεια του κώδικα, ορίζεται η βοηθητική συνάρτηση `downsample(insig, m)`, η οποία απλώς δέχεται το παράθυρο και τον παράγοντα υποδειγματοληψίας και επιστρέφει το προκύπτον από την είσοδο σήμα, εάν κρατούνται δείγματα ανά m. Φυσικά, εν προκειμένω η συνάρτηση καλείται με $m = M = 32$.

```
def downsample(insig, m):

    """Keep every m-th point of insig."""
    return insig[::m]
```

ΒΗΜΑ 2.2

Στο βήμα αυτό υλοποιούνται δύο διαφορετικοί κβαντιστές. Ο ένας είναι ομοιόμορφος κβαντιστής των 8 ψηφίων και λειτουργεί θεωρώντας πεδίο τιμών του σήματος το $[-1,1]$, ασχέτως αν αυτό πράγματι ισχύει ή όχι, και ως εκ τούτου έχει βήμα $\Delta = \frac{1-(-1)}{2^8} = 2^{-7}$. Ο άλλος είναι προσαρμοσμένος κβαντιστής ο οποίος, ανά παράθυρο, αποφασίζει πόσα ψηφία απαιτούνται για την κβάντιση του σήματος βάσει της δεδομένης συνάρτησης:

$$B_k = int\left(\log_2\left(\frac{R}{\min\left(T_g(i)\right)}\right) - 1\right)$$

Όπου $R = 2^B = 2^{16}$ τα επίπεδα κβαντισμού του αρχικού μουσικού σήματος και $T_g(i)$ το συνολικό κατώφλι κάλυψης ανά διακριτή συχνότητα του παραθύρου, εφόσον αυτή η συχνότητα βρίσκεται στο πεδίο ορισμού του φίλτρου. Το βήμα Δ του προσαρμοσμένου κβαντιστή προκύπτει από το πραγματικό πεδίο τιμών καθενός παραθύρου και από τα υπολογισθέντα ψηφία ως $\Delta_\kappa = \frac{\max(window) - \mathrm{mi}\ (window)}{2^{B_k}}$.

```
def bitsk(thresholds, i, j):
    """Calculate required bits for quantization."""
    if thresholds[i][j]:
        return int(np.ceil(np.log2(2**(16)/min(thresholds[i][j]))-1))
    return 0

def quantize(insig, bits, flag="adaptive"):
    """Perform quantization.

    Return the quantization level of each sample.
    """
    def find_best_match(sample, values):
        """Return value from values that minimizes difference from sample."""
        result = min([abs(value - sample) for value in values])
        if result <= 0:
            return -(result-sample)
        return result+sample
    if flag == "adaptive":
        return [find_best_match(sample,
                                np.linspace(min(insig), max(insig), 2**bits))
                for sample in insig]
    if flag == "8bit":
        return [find_best_match(sample, np.linspace(-1, 1, 2**8))
                for sample in insig]
    print("Flag either 'adaptive' or '8bit'")
```

Η συνάρτηση `bitsk(thresholds, i, j)` δέχεται ως ορίσματα τα συνολικά κατώφλια ανά παράθυρο ανά φίλτρο "thresholds" και το παράθυρο και φίλτρο "i", "j" .[2]

Τέλος, `quantize(insig, bits, flag="adaptive")` υπολογίζει την στάθμη στην οποίαν θα κβαντιστεί καθένα δείγμα του παραθύρου εισόδου, δεχόμενη ως ορίσματα τα αντιστοίχως υπολογισθέντα ψηφία κβαντισμού και βήμα κβαντισμού. Χρησιμοποιεί την βοηθητική `find_best_match(sample, values)` στην οποίαν μεταβιβάζεται το δείγμα και οι στάθμες του κβαντιστή και αναζητείται η στάθμη κατ' ελάχιστον απέχουσα από το δείγμα.

Για την υλοποίηση του προσαρμοσμένου κβαντιστή, καλείται η συνάρτηση `quantize(insig, bits, flag="adaptive")` με το κατάλληλο πλήθος ψηφίων και το κατάλληλο flag="adaptive".

Για την υλοποίηση του ομοιόμορφου κβαντιστή 8 ψηφίων, καλείται η ίδια συνάρτηση αλλά flag="8bit"

[2] Καθένα παράθυρο έχει έναν πίνακα συνολικών κατωφλίων με 256 κατώφλια, τόσα όσα και οι συχνότητες. Ουσιαστικά, πριν την κλήση της συνάρτησης, χωρίζονται αυτά τα κατώφλια στα αντίστοιχα 32 πεδία ορισμών, ένα για κάθε φίλτρο. Άρα, η παράσταση `thresholds[i][j]` επιστρέφει έναν κατάλογο με τα κατώφλια του παραθύρου `i` τα οποία αφορούν στο φίλτρο `j`.

ΒΗΜΑ 2.3

Σε αυτό το βήμα πρέπει να υπερδειγματοληπτηθεί καθεμία ακολουθία κβαντισμένων δειγμάτων, να διέλθουν από τα φίλτρα συνθέσεως και να προστεθούν τα 32 συντεθειμένα παράθυρα τα οποία αντιστοιχούν σε κάθε παράθυρο του μουσικού σήματος σε ένα. Τέλος, επειδή, λόγω των προηγουμένων συνελίξεων, τα τελικά παράθυρα έχουν μήκος μεγαλύτερο από τα αυθεντικά παράθυρα, πρέπει να προστεθούν με κατάλληλη αλληλεπικάλυψη για το τελικό αποτέλεσμα.

Το αποτέλεσμα θα έχει μία χρονική μετατόπιση, η οποία θα εντοπιστεί και υποστεί κατάλληλο χειρισμό ώστε να μπορέσει να υπολογιστεί το μέσον τετραγωνικό σφάλμα. Ακολουθούν οι συναρτήσεις:

```
def oversample(insig, m=M):

    """
    Oversample.

    Keep every m-th sample of insig and stuff the
blanks with zeroes.
    """
    result = [0 for _ in range(len(insig)*m)]
    for s in range(len(insig)*m):
        if s % m == 0:
            result[s] = insig[s//m]
    return result
```

Η `oversample(insig, m=M)` δέχεται ένα παράθυρο εισόδου και έναν παράγοντα υπερδειγματοληψίας. Κατασκευάζει έναν πίνακα μεγέθους $m * (\mu\acute{\varepsilon}\gamma\varepsilon\theta o\varsigma\ \pi\alpha\rho\alpha\theta\acute{\upsilon}\rho o\upsilon)$ και θέτει όλες τις τιμές μηδενικές. Τέλος, διατρέχει τις θέσεις του πίνακα και όταν βρίσκει θέση πολλαπλάσια του παράγοντα m, την θέτει ίση με την αντίστοιχη τιμή του παραθύρου εισόδου.

Η συνάρτηση `process(windows, spec_thresh, flag="adaptive")` υλοποιεί την επεξεργασία των αυθεντικών παραθύρων του μουσικού σήματος και επιστρέφει έναν πίνακα με ένα επεξεργασμένο παράθυρο ανά φίλτρο ανά αυθεντικό παράθυρο μουσικού σήματος. Με την παράμετρο «σημαία» επισημαίνεται εάν στο στάδιο του κβαντιστή επιλέγεται ο προσαρμοστικός κβαντιστής ή ο ομοιόμορφος 8 ψηφιών. Η παράμετρος `spec_thresh` κρατάει τα συνολικά κατώφλια κάλυψης ανά διακριτή συχνότητα ανά παράθυρο. Σε αμφότερες τις περιπτώσεις, η διαδικασία είναι όμοια ανά παράθυρο:

1. Υπολόγισε τις συνελίξεις του παραθύρου με καθένα από τα 32 φίλτρα.
2. Πραγματοποίησε σε καθένα από τα 32 παράθυρα υποδειγματοληψία κατά παράγοντα M.
3. Εάν έχει επιλεγεί προσαρμοσμένος κβαντιστής:
    a. Χώρισε τα κατώφλια των συχνοτήτων στα κατάλληλα πεδία ορισμού των φίλτρων.
    b. Υπολόγισε για κάθε παράθυρο το κατάλληλο πλήθος ψηφίων κβάντισης.
    c. Κβάντισε καθένα παράθυρο με το αντίστοιχο πλήθος ψηφίων.
    d. Πραγματοποίησε σε καθένα παράθυρο υπερδειγματοληψία κατά παράγοντα M.
    e. Υπολόγισε τις συνελίξεις καθενός παραθύρου με τα φίλτρα συνθέσεως και επίστρεψέ τες.
4. Εάν έχει επιλεγεί ομοιόμορφος κβαντιστής 8 ψηφίων:
    a. Κβάντισε καθένα παράθυρο με 8 ψηφία.
    b. Πραγματοποίησε σε καθένα παράθυρο υπερδειγματοληψία κατά παράγοντα M.
    c. Υπολόγισε τις συνελίξεις καθενός παραθύρου με τα φίλτρα συνθέσεως και επίστρεψέ τες.
5. Τύπωσε ενημερωτικό μήνυμα σχετικό με την χρήση της παραμέτρου «σημαίας».

```
def process(windows, spec_thresh, flag="adaptive"):
    """Process the given windows and perform adaptive quantization."""
    mdct_convolutions = [[mdct(window, k) for k in range(M)]
                         for window in windows]
    mdct_downsampled = [[downsample(conv, M) for conv in convols]
                        for convols in mdct_convolutions]
    if (flag == "adaptive"):
        valid_thresholds = [[[spec_thresh[s][f] for f in domains[k]]
                             for k in range(M)] for s in range(NUM_WINDOWS)]
        Bk = [[bitsk(valid_thresholds, s, k) for k in range(M)]
              for s in range(NUM_WINDOWS)]
        quantized = [[quantize(mdct_downsampled[s][k], Bk[s][k], flag)
                      for k in range(M)] for s in range(NUM_WINDOWS)]
        oversampled = [[oversample(quantized[s][k]) for k in range(M)]
                       for s in range(NUM_WINDOWS)]
        return [[mdct(oversampled[s][k], k, M, "synthesis")
                 for k in range(M)] for s in range(NUM_WINDOWS)]
    elif (flag == "8bit"):
        quantized = [[quantize(mdct_downsampled[s][k], 8, flag)
                      for k in range(M)] for s in range(NUM_WINDOWS)]
        oversampled = [[oversample(quantized[s][k]) for k in range(M)]
                       for s in range(NUM_WINDOWS)]
        return [[mdct(oversampled[s][k], k, M, "synthesis")
                 for k in range(M)] for s in range(NUM_WINDOWS)]
    print("Flag either 'adaptive' or '8bit'")
```

Η συνάρτηση `reconstruct(windows, filename, srate=44100)` ανακατασκευάζει το αρχικό μουσικό σήματα από τα παράθυρα εισόδου της και γράφει το αποτέλεσμα σε ένα αρχείο .wav. Δέχεται, λοιπόν, ως παραμέτρους τα παράθυρα προς ανασύνθεση, το επιθυμητό όνομα αρχείο και την συχνότητα δειγματοληψίας ώστε να πετύχει η εγγραφή.

Η τελική ανασύνθεση του σήματος υλοποιείται με την συνάρτηση `reconstruct(windows, filename, srate=44100)` η οποία χρησιμοποιεί την βοηθητική συνάρτηση `overlap_add(a, b, ai=0, bi=0)`.

```
def reconstruct(windows, filename, srate=44100):
    """Reconstruct the final signal using given windows and write to wav."""
    added = [np.zeros(len(windows[0][0])) for window in windows]
    for index, window in enumerate(windows):
        for filtered in window:
            added[index] = list(np.array(filtered)+np.array(added[index]))
    added = np.array(added)
    length = len(added[0])
    dif = length-N
    result = added[0]
    for i in range(1, NUM_WINDOWS):
        result = overlap_add(result, added[i], bi=len(result)-dif)
    soundfile.write(filename, result/abs(max(result)), srate)
    return result
```

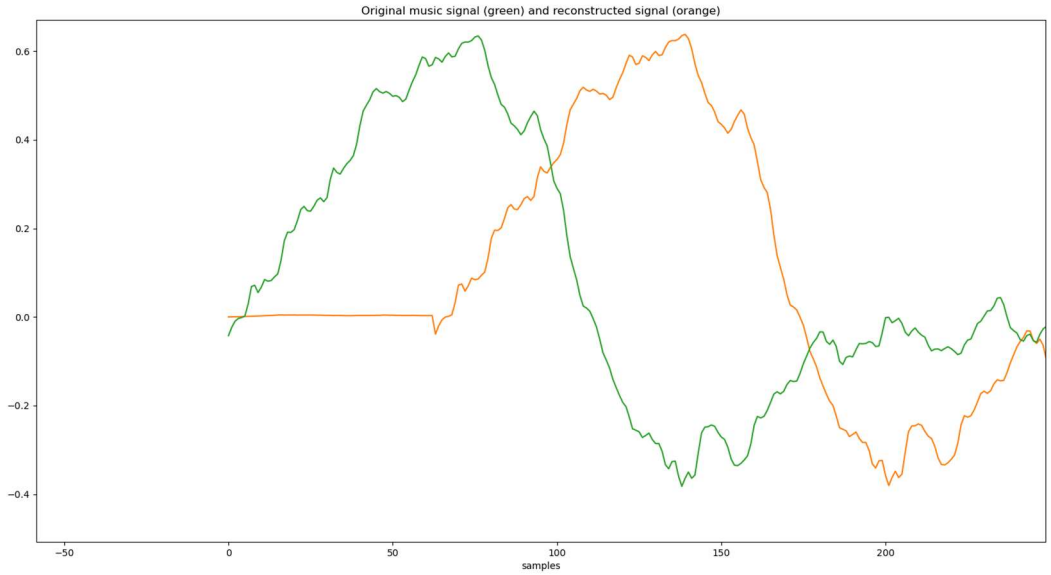
Αρχικά, προσθέτει τα 32 παράθυρα τα οποία αντιστοιχούν σε κάθε αυθεντικό παράθυρο του σήματος κατά στοιχεία και καταλήγει με τόσα παράθυρα όσα εξ αρχής. Στην συνέχεια, υπολογίζει την επιθυμητή αλληλεπικάλυψη κατά την άθροιση των παραθύρων και αρχικοποιεί το αποτέλεσμα. Στον επαναληπτικό βρόχο πραγματοποιεί την επικαλυπτική άθροιση overlap-add των παραθύρων, χρήσει της βοηθητικής συνάρτησης `overlap_add(a, b, ai=0, bi=0)` . Τέλος γράφει το κανονικοποιημένο αποτέλεσμα σε αρχείο .wav και επιστρέφει το ανακατασκευασμένο σήμα.

```
def overlap_add(a, b, ai=0, bi=0):

    """Add signals with specified overlap."""
    assert ai >= 0
    assert bi >= 0
    al = len(a)
    bl = len(b)
    cl = max(ai+al, bi+bl)
    c = np.zeros(cl)
    c[ai: ai+al] += a
    c[bi: bi+bl] += b
    return c
```

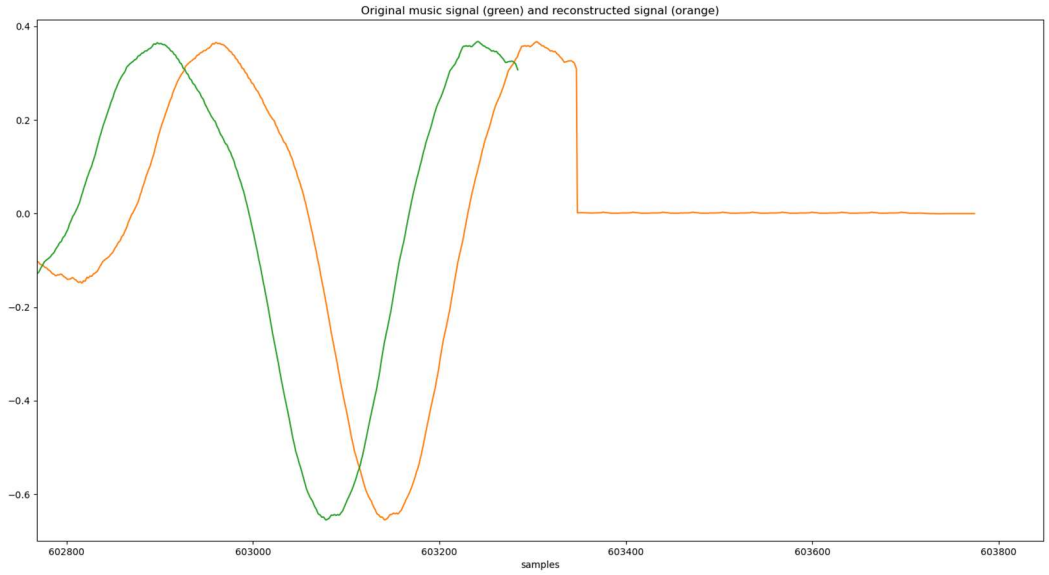
Η βοηθητική συνάρτηση `overlap_add(a, b, ai=0, bi=0)` δέχεται δύο πίνακες και δύο παραμέτρους για την επισήμανση της αλληλεπικάλυψης. Υπολογίζει το μέγεθος του τελικού πίνακα, τον αρχικοποιεί με μηδενικά και προσθέτει στο, οριζόμενο από την παράμετρο ai, εύρος τα στοιχεία του πίνακα a και στο, αντίστοιχο, εύρος του πίνακα b, τα στοιχεία του πίνακα b. Εν προκειμένω, καλείται πάντοτε με την παράμετρο $ai = 0$ και την παράμετρο $bi = len(result) - dif$.

Παρουσίαση Αποτελεσμάτων και Σύγκριση Μεθόδων

Σχεδιάζοντας το διάγραμμα με το αρχικό μουσικό σήμα και το ανακατασκευασμένο (με οποιονδήποτε κβαντιστή) διακρίνεται μία μετατόπιση 64 δειγμάτων αλλά και η ύπαρξη επιπλέον μηδενικών δειγμάτων στο τέλος.

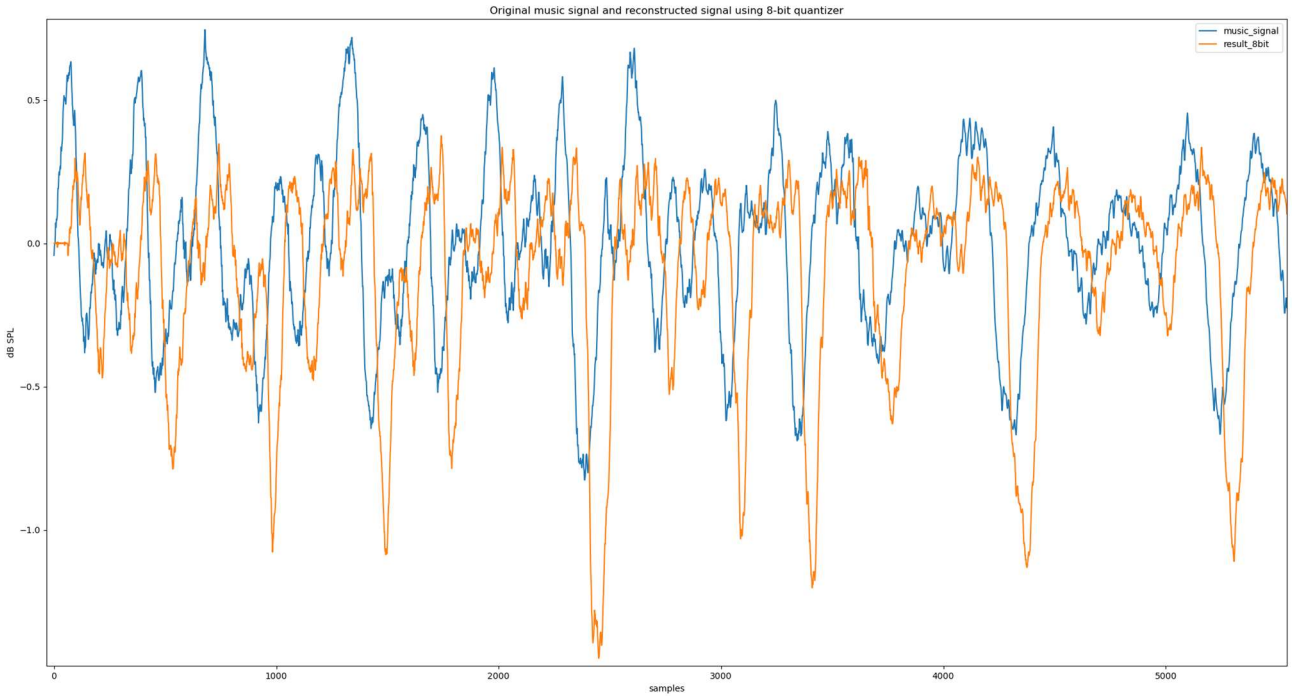


Εικόνα 14 Μετατόπιση και επιπλέον μηδενικά στην αρχή του ανακατασκευασμένου σήματος.

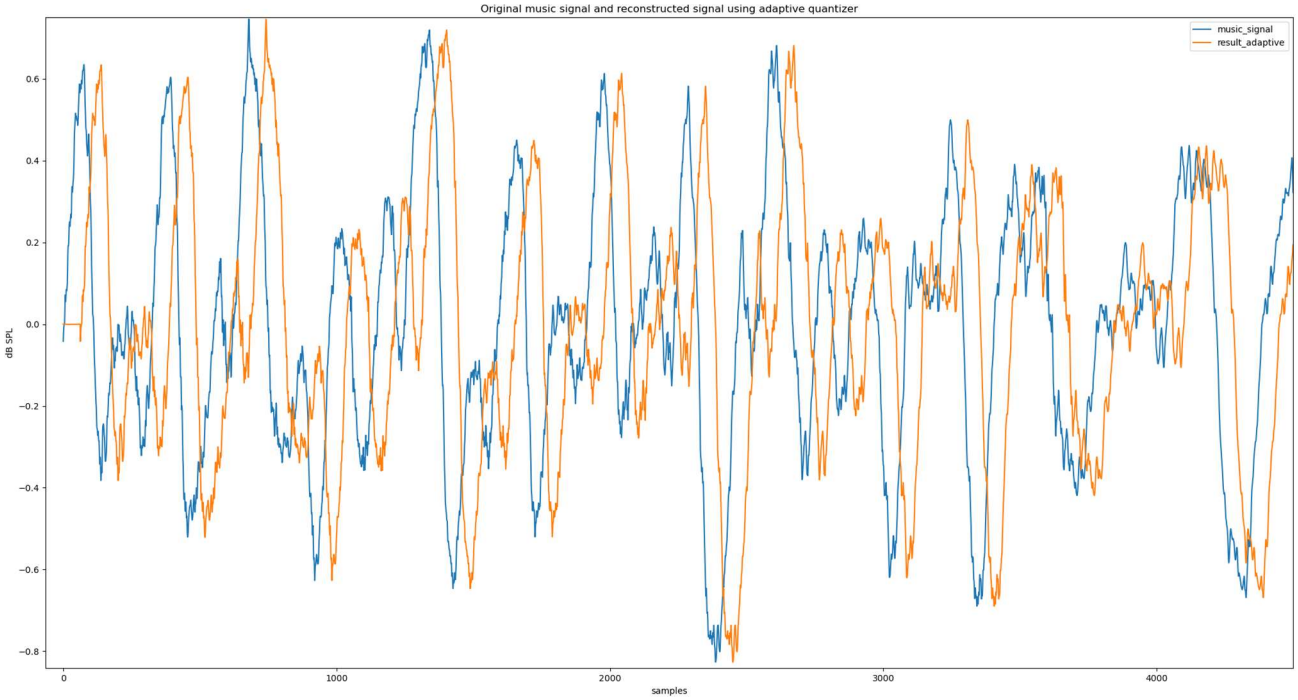


Εικόνα 15 Μετατόπιση και επιπλέον μηδενικά στο τέλος του ανακατασκευασμένου σήματος.

Σύγκριση των ανακατασκευασμένων σημάτων με το αυθεντικό

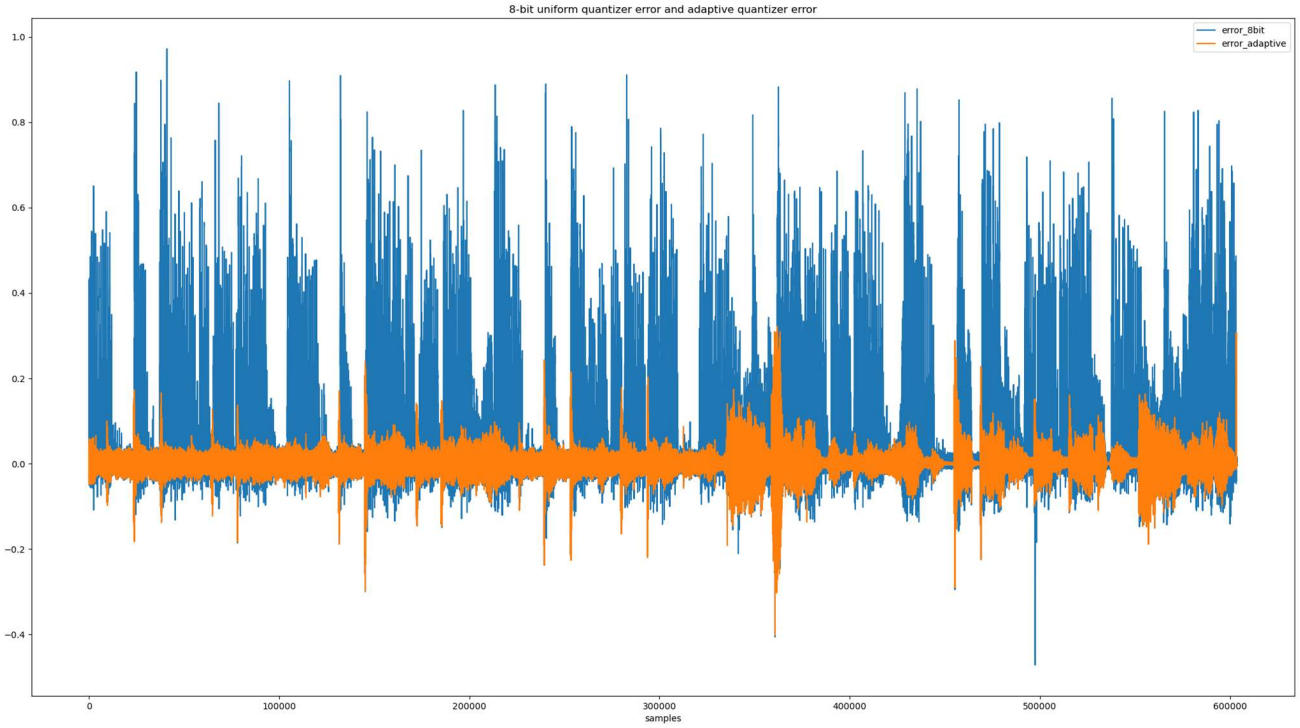


Εικόνα 16 Σύγκριση του ανακατασκευασμένου σήματος χρήσει 8ψηφίου ομοιόμορφου κβαντιστή με το αυθεντικό.



Εικόνα 17 Σύγκριση του ανακατασκευασμένου σήματος χρήσει προσαρμοζόμενου κβαντιστή με το αυθεντικό.

Παρατίθεται και η απεικόνιση των σφαλμάτων ανά τεχνική:



Για να υπολογισθεί, λοιπόν, το μέσον τετραγωνικό σφάλμα ορθώς, πρέπει να παρεμβληθούν $2 \cdot M = 64$ μηδενικά στην αρχή του μουσικού σήματος και $len(result) - len(music\ signal) - 2 \cdot M = 426$ μηδενικά στο τέλος.

Τα μέσα τετραγωνικά σφάλματα του προσαρμοσμένου και του ομοιόμορφου κβαντιστή 8 ψηφίων είναι:

	Προσαρμοσμένος Κβαντιστής	Ομοιόμορφος Κβαντιστής 8 ψηφίων
Μέσο τετραγωνικό σφάλμα	0.00038178998206995856= 0.0382 %	0.021478254678596275= 2%

Οι υπολογισμοί και τα διαγράμματα, λοιπόν, δείχνουν ότι ο προσαρμοσμένος κβαντιστής έχει καλύτερη επίδοση από τον ομοιόμορφο.

Πράγματι, αυτό επαληθεύεται και ακουστικά, ακούγοντας τα αντίστοιχα ανακατασκευασμένα μουσικά σήματα. Στην περίπτωση του προσαρμοσμένου κβαντιστή, δεν μπορεί κανείς να διακρίνει την διαφορά από το αρχικό σήμα. Στην περίπτωση, όμως, του ομοιόμορφου κβαντιστή, η ποιότητα είναι εμφανώς χειρότερη. Σαφώς, λοιπόν, αποδεικνύεται ότι ο προσαρμοσμένος κβαντιστής έχει καλύτερη επίδοση από τον ομοιόμορφο.

Όσον αφορά στα ποσοστά συμπίεσης, αυτά υπολογίζονται σε σχέση με το πλήθος των ψηφίων στην αναπαράσταση του αρχικού μουσικού σήματος και σε σχέση με το πλήθος των ψηφίων του ανακατασκευασμένου σήματος πριν την επικαλυπτόμενη πρόσθεση overlap-add.

Συγκεκριμένα, το ποσοστό για τον ομοιόμορφο κβαντιστή προκύπτει πολλαπλασιάζοντας το πλήθος των παραθύρων (1179) επί το πλήθος των δειγμάτων (639 ή 512)[3] επί το πλήθος των ψηφίων (8) διά το πλήθος των δειγμάτων επί το πλήθος των ψηφίων του αρχικού σήματος.

Για τον προσαρμοζόμενο κβαντιστή, το ποσοστό προκύπτει αθροίζοντας το γινόμενο του πλήθους των παραθύρων (1179) επί το πλήθος των μεταδιδόμενων ψηφίων για την αποκωδικοποίηση (16+16) με το γινόμενο του πλήθους των δειγμάτων (639) επί τα ψηφία κβάντισης στο συγκεκριμένο παράθυρο διά το πλήθος των δειγμάτων επί το πλήθος των ψηφίων του αρχικού σήματος.

	Προσαρμοσμένος Κβαντιστής	Ομοιόμορφος Κβαντιστής 8 ψηφίων
Ποσοστό συμπίεσης	1- 73.6% = 26.4%	1 -62.439%= 37.6% ή 1-50.03%=49.97%

Η επεξεργασία του σήματος, η παρεμβολή μηδενικών στο αρχικό σήμα και ο υπολογισμός των σφαλμάτων έγινε στο παρακάτω κομμάτι κώδικα χρήσει των αντιστοίχων συναρτήσεων:

```
def squared_error(sig1, sig2):
    """Calculate squared error between two signals."""
    return np.square(np.array(sig1)-np.array(sig2))

synthesized = process(windowed_music_signals,

                      spectrarum_thresholds, "adaptive")

synthesized_8bit = process(windowed_music_signals,
                           spectrarum_thresholds, "8bit")

result_adaptive = reconstruct(synthesized, "result_adaptive.wav")

result_8bit = reconstruct(synthesized_8bit, "result_8bit.wav")

# pad some zeros to the original music signal in order to calculate the
# mean squared error. The reason why this is necessary is that the
# the reconstructed signal contains some tiny values that could be regarded
# as zeroes, but they are not.
music_signal_altered = np.pad(
    music_signal, (2*M, len(result_adaptive)-2*M-music_length), 'constant')

squared_error_adaptive = squared_error(music_signal_altered, result_adaptive)

error_adaptive = np.array(music_signal_altered)-np.array(result_adaptive)

mse_adaptive = sum(squared_error_adaptive)/len(squared_error_adaptive)

squared_error_8bit = squared_error(music_signal_altered, result_8bit)

error_8bit = np.array(music_signal_altered)-np.array(result_8bit)

mse_8bit = sum(squared_error_8bit)/len(squared_error_8bit)
```

[3] Στην περίπτωση του ομοιόμορφου κβαντιστή, επειδή όλα τα παράθυρα χρησιμοποιούν ίδιο πλήθος ψηφίων, δεν δημιουργείται πρόβλημα από την επικάλυψή τους, άρα τα περίσσεια μπορούν να αμεληθούν και να θεωρηθεί πλήθος 512 ψηφίων αντί για 639. Το ίδιο δεν μπορεί να γίνει στον προσαρμοζόμενο κβαντιστή.