Περικλής Αλεξίου 03122079
Πηνελόπη Άννα Τσιμπλάκη 03122097

Ψηφιακή Επεξεργασία Σημάτων

2η Εργαστηριακή Άσκηση

Πρόλογος

Για τα 2 μέρη της εργασίας χρησιμοποιήθηκαν οι βιβλιοθήκες που φαίνονται στην παρακάτω φωτογραφία.

```
import numpy as np
import IPython.display as ipd
import matplotlib.pyplot as plt
import scipy as sp
pi = np.pi
```

Και τα δύο μέρη υλοποιήθηκαν σε κοινό jupyter script καθώς το μέρος 2 χρειαζόταν δεδομένα που βρέθηκαν στο μέρος 1. Συνολικός σκοπός της άσκησης ήταν η συμπίεση ενός σήματος μουσικής με βάση το ψυχοακουστικό μοντέλο. Στο 1ο μέρος βρίσκεται το κατώφλι κάλυψης Tg, το οποίο είναι κομβικό στοιχείο για τον υπολογισμό των αριθμών των bits της κωδικοποίησης που λαμβάνει χώρα στο 2ο μέρος.

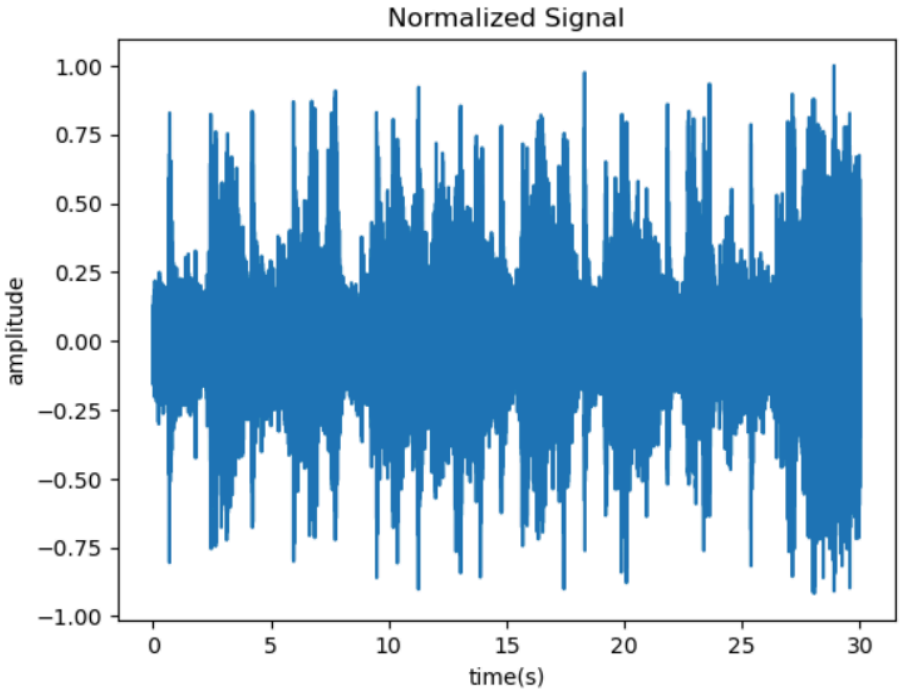
Μέρος 1. Ψυχοακουστικό Μοντέλο 1

Στο 1ο μέρος της εργαστηριακής άσκησης θέλουμε να βρούμε το συνολικό κατώφλι κάλυψης Tg, το οποίο υπολογίζεται με βάση το απόλυτο κατώφλι ακοής (ATH) και τα φασματικά χαρακτηριστικά του σήματος. Πιο συγκεκριμένα, αν στο σήμα προς επεξεργασία, το περιεχόμενο του σε μία συχνότητα υπερτερεί σε μεγάλο βαθμό (tonal masker TN) επί των γειτονικών του, τότε τα δεύτερα μπορούν να παραλειφθούν. Σε συχνοτικές περιοχές που δεν υπάρχουν τέτοια peaks, χρησιμοποιούνται οι noise masker (NM) για τον καθορισμό των συχνοτήτων που επισκιάζονται από θορυβώδεις συνιστώσες με ευρύ φάσμα.
*Η υλοποίηση του πρώτου μέρους δεν έγινε σε μορφή συνάρτησης καθώς πολλά δεδομένα πάρθηκαν έτοιμα από το πρόσθετο υλικό και δεν βρισκόντουσαν εσωτερικά της υλοποίησης.

Βήμα 1.0: Προεπεξεργασία του σήματος

Σε αυτή την άσκηση θα επεξεργαστούμε το ηχητικό σήμα που απεικονίζεται στο διάγραμμα.

```
fs, signal = sp.io.wavfile.read(r"C:\Users\tspen\Documents\DSP25_LabProject2_Material\music_dsp.wav")
normal = signal/np.max(np.abs(signal))
plotter(normal, fs)
```



Το σήμα αναλύεται και επεξεργάζεται σε παράθυρα. Για το μέρος 1 χρησιμοποιούμε hanning παράθυρα μήκους N = 512 για να τεμαχίσουμε το σήμα μας, αφού αυτό έχει κανονικοποιηθεί.

```
windows = []
step = 512
hann = np.hanning(step)
for i in range(0, size, step):
    xi = normal[i:i+step]
    if len(xi) < step:
        z = np.zeros(step-len(xi))
        xi = np.concatenate((xi, z))
    win = xi*hann
    windows.append(win)

windows = np.array(windows)
Nw = len(windows)
```

Βήμα 1.1: Φασματική Ανάλυση

Καθώς κάποιες συχνότητες είναι ισχυρότερα αντιληπτές στο ανθρώπινο αυτί (πράγμα που φαίνεται και στο ATH), εργαζόμαστε στη μη γραμμική κλίμακα bark. Η κλίμακα bark χωρίζει το ακουστικό φάσμα σε 24 κρίσιμες ζώνες, αφιερώνοντας περισσότερες σε χαμηλές συχνότητες.

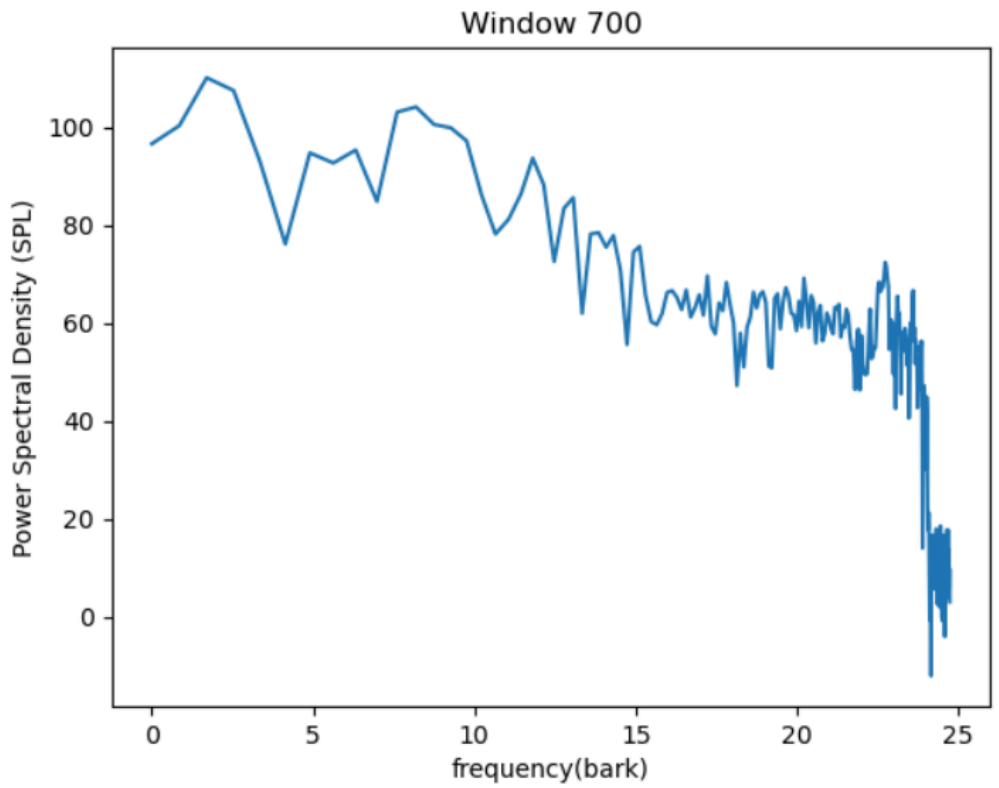
```
def b(f):
    return 13*np.arctan(0.00076*f) + 3.5*np.arctan((f/7500)**2)
```

Για τον εντοπισμό των TM πρέπει πρώτα να βρούμε την μορφή του φασματικού περιεχομένου του σήματος μας μέσω της φασματικής ισχύος P(f), που επειδή βρισκόμαστε σε χώρο διακριτής συχνότητας και χρόνου γίνεται P(k). Η P(k) εκφράζεται σε μονάδες SPL. Το σήμα μας είναι πραγματικό οπότε παρουσιάζει άρτια συμμετρία, πράγμα που μας επιτρέπει να εργαζόμαστε στα k εντός του [0, N/2] = [0, 256], αντί για μέχρι το N.

```
N = 512
N2 = N//2
def spectral_density(win):
    PN = 90.302
    sp = np.fft.fft(win, N)
    sp = sp[:N//2]
    res = PN + 10*np.log10(np.abs(sp)**2 + 1e-12)
    return res
```

```
P = [spectral_density(w) for w in windows]
f = np.linspace(0, fs//2, N2)
bark = b(f)
plt.plot(bark, P[700])
```

Παρακάτω φαίνεται το φάσμα ισχύος του 700ου παραθύρου.



Βήμα 1.2: Εντοπισμός μασκών τόνων και θορύβου (Maskers)

Τις τονικές μάσκες τις αναζητούμε σε συχνοτικές γειτονιές, μελετώντας το φάσμα ισχύος. Εντοπίζουμε μια τονική μάσκα στη συχνότητα k, εάν υπάρχει τοπικό μέγιστο του P(k), το οποίο υπερτερεί από τα γειτονικά του σημεία κατά 7dB. Το εύρος της γειτονιάς είναι μεταβλητό ανά συχνότητα, και καθώς αυτή αυξάνεται, το εύρος μεγαλώνει.

Η συνάρτηση Dk λαμβάνει ως παράμετρο τη διακριτή συχνότητα k και επιστρέφει τις αποστάσεις των σημείων που ελέγχονται για τον εντοπισμό τοπικής μάσκας σε αυτή.

```
def Dk(k):
    res = [2]
    if(63 <= k):
        res.append(3)
    if(127 <= k):
        for i in range(4, 7):
            res.append(i)
    return res
```

Από τις παραπάνω συγκρίσεις προκύπτει μία λογική συνάρτηση St(k) που γίνεται 1 μόνο αν στο k αυτό έχουμε TM. Για τα k που St(k) = 1 υπολογίζουμε την ισχύ της τονικής μάσκας P_TM(k) ενώ οπουδήποτε αλλού την θεωρούμε 0.

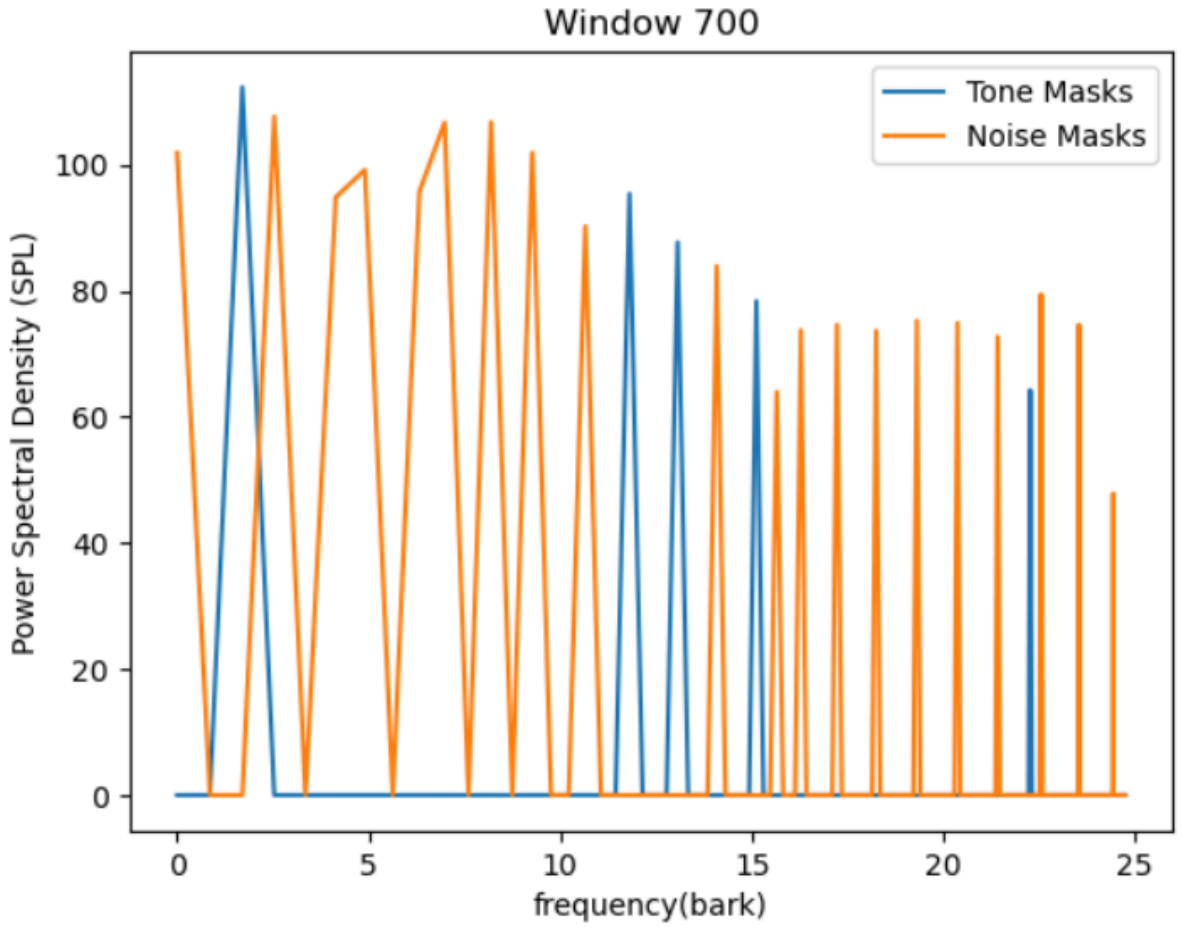
```
def St(win, k):
    if(k<2 or k>=250):
        return False
    x1 = (P[win][k] > P[win][k+1]) and (P[win][k] > P[win][k-1])
    for dk in Dk(k):
        x1 = x1 and ((P[win][k]>P[win][k+dk] + 7) and (P[win][k]>P[win][k-dk] + 7))
    return x1
```

```
P_TM = []
for w in range(Nw):
    temp = np.zeros(N2)
    for k in range(N2):
        if(St(w,k)):
            temp[k] = 10*np.log10(10**(0.1*P[w][k-1]) + 10**(0.1*P[w][k]) + 10**(0.1*P[w][k+1]))
    temp = np.array(temp)
    P_TM.append(temp)

P_TM = np.array(P_TM)
P_NM = np.load(r"C:\Users\tspen\Documents\DSP25_LabProject2_Material\P_NM-25.npy")
P_NM = np.transpose(P_NM)
```

Οι ισχείς των NM δίνονται έτοιμες.

Παρακάτω φαίνονται οι τονικές μάσκες και οι μάσκες θορύβου στο παράθυρο 700.



Βήμα 1.3: Μείωση και αναδιοργάνωση των μασκών

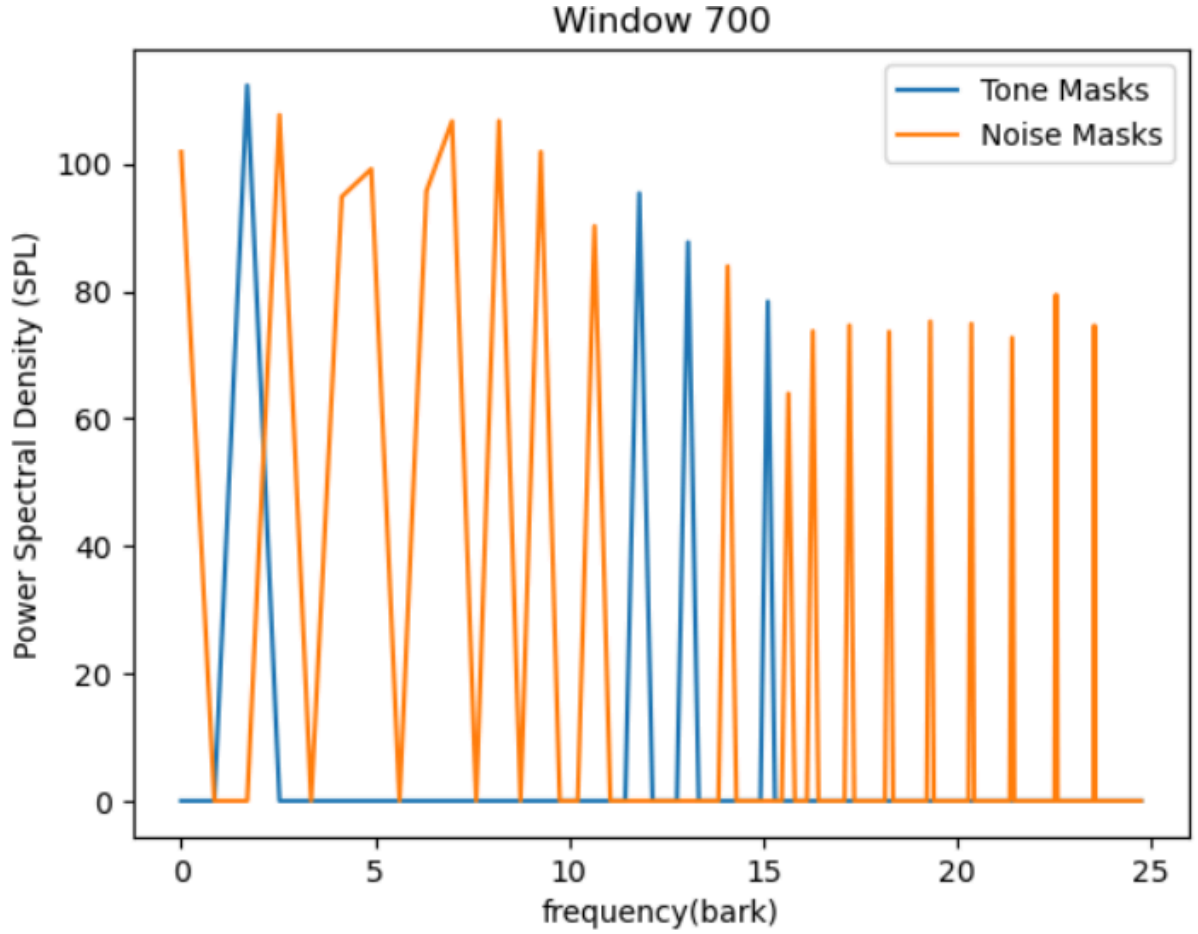
Σε αυτό που βήμα γίνεται μείωση και αναδιοργάνωση των μασκών. Πιο αναλυτικά, τις μειώνουμε με βάση δύο κριτήρια.

ι) Θέλουμε οι ισχείς των μασκών να ξεπερνούν το ATH για να θεωρηθεί ότι γίνονται από το ανθρώπινο αυτί,

ιι) Ομαδοποιώντας τις μάσκες ανά 0.5 bark, κρατάμε μόνο αυτή με την μεγαλύτερη ισχύ.

Στο βήμα αυτό, οι επιλεγμένες μάσκες δίνονται έτοιμες ως P_TMc και P_NMc.

Στο παρακάτω σχήμα φαίνονται οι τελικές μάσκες.



Βήμα 1.4: Υπολογισμός των δύο διαφορετικών κατωφλίων κάλυψης (Individual Masking Thresholds)

Για κάθε Τονική Μάσκα στην θέση j, υπολογίζουμε το ποσοστό κάλυψης T_TM(i, j) στη θέση i. Θεωρούμε πως η έκταση κάθε τονικής μάσκας είναι 12 Bark, δηλαδή φτάνει μέχρι και την θέση i όπου b(i) ∈ [b(j) − 3, b(j) + 8]. Στον υπολογισμό των T_TN κάθε τονικής μάσκας χρησιμοποιούμε και την συνάρτηση SF(i, j), η οποία μοντελοποιεί τον βαθμό επισκίασης των συχνοτήτων της θέσης i από την μάσκα στην θέση j.

Τα T_NM(i, j) των Μασκών Θορύβου υπολογίζονται με αντίστοιχο τρόπο.

```
def SF(w, i, j, t):
    Db = bark[i] - bark[j]
    if(Db >= -3 and Db < -1):
        res = 17*Db - 0.4*t +11
    elif(Db >= -1 and Db < 0):
        res = (0.4*t + 6)*Db
    elif(Db >= 0 and Db < 1):
        res = -17*Db
    elif(Db >= 1 and Db < 8):
        res = (0.15*t - 17)*Db - 0.15*t
    else:
        res = -np.inf
    return res
```

```
T_TM = []
T_NM = []
for w in range(Nw):

    temp1 = np.full((N2, N2), -np.inf)
    temp2 = np.full((N2, N2), -np.inf)

    for j in range(N2):
        if P_TMc[w][j] > 0:
            for i in range(N2):
                if (bark[i] >= bark[j] - 3) and (bark[i] <= bark[j] + 8):
                    temp1[i][j] = P_TMc[w][j] - 0.275 * bark[j] + SF(w, i, j, P_TMc[w][j]) - 6.025
        if P_NMc[w][j] > 0:
            for i in range(N2):
                if bark[i] >= bark[j] - 3 and bark[i] <= bark[j] + 8:
                    temp2[i][j] = P_NMc[w][j] - 0.175 * bark[j] + SF(w, i, j, P_NMc[w][j]) - 2.025
    T_TM.append(temp1)
    T_NM.append(temp2)
```

Βήμα 1.5: Υπολογισμός του συνολικού κατωφλίου κάλυψης (Global Masking Threshold)

Για τον τελικό υπολογισμό του κατωφλιού Tg που θέλουμε να βρούμε, χρησιμοποιούμε τον τύπο:

$$T_g(i) = 10\log_{10}\left(10^{0.1T_q(i)} + \sum_{l=0}^{255} 10^{0.1T_{TM}(i,\ell)} + \sum_{m=0}^{255} 10^{0.1T_{NM}(i,m)}\right) \text{ dB SPL}$$

Παρατηρούμε πως για μία διακριτή συχνότητα i, το Tg είναι συνάρτηση του ATH αλλά και όλων των T_TM και T_NM σε θέσεις l και m που φτάνουν μέχρι το i. Για την υπολογιστική ορθότητα του προγράμματος μας έχουμε αρχικοποιήσει όλα τα T_TM και T_NM σε πλην άπειρο και τα έχουμε αλλάξει μόνο εκεί που ορίζονται τιμές, ώστε όταν γίνεται το άθροισμα με τα εκθετικά, να έχουμε μηδενική συνεισφορά από τα σημεία j που δεν έχουν κάλυψη στα σημεία i.

```
def T_q(fhz):
    t = fhz/1000
    return 3.65*(t**(-0.8)) - 6.5*np.exp(-0.6*((t-3.3)**2)) + 0.001*(t**4)
```

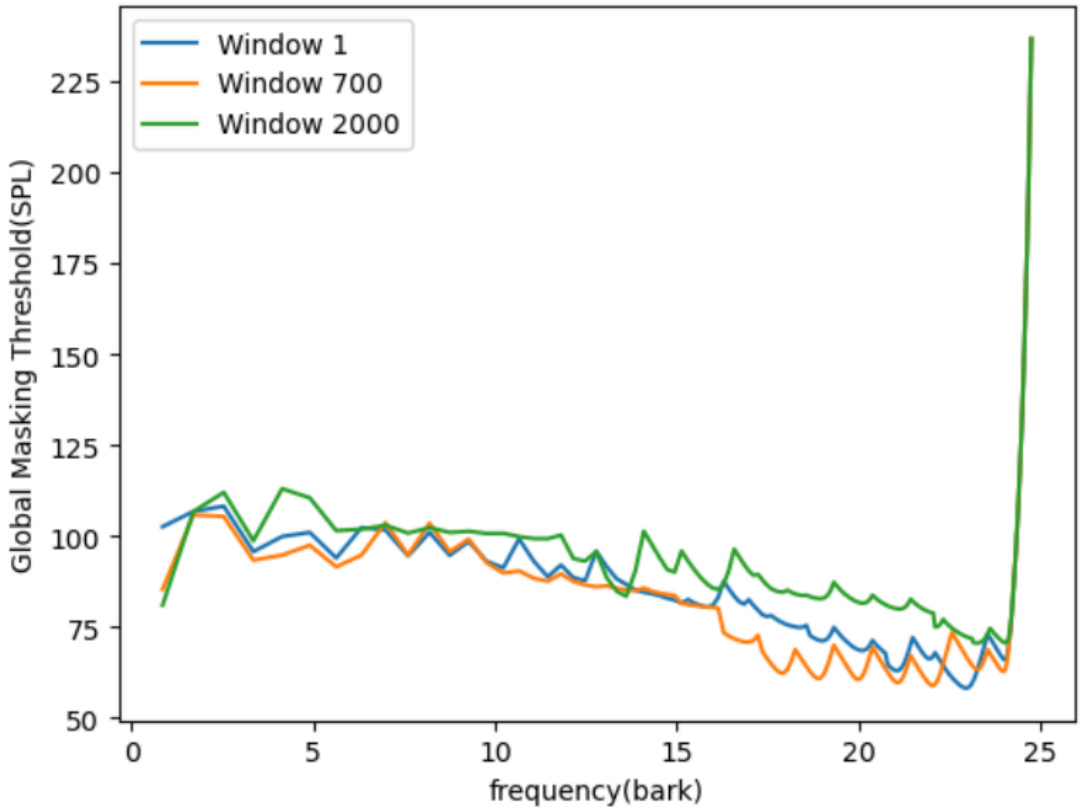
```
f = np.linspace(0, fs//2, 256)

Tq = T_q(f)
```

```
Tg = []

for w in range(Nw):
    temp = np.zeros(N2)
    for i in range(N2):
        t1 = np.sum(10**(0.1*T_TM[w][i]))
        t2 = np.sum(10**(0.1*T_NM[w][i]))
        t3 = 10**(0.1*Tq[i])
        t = 10*np.log10(t1+t2+t3)
        temp[i] = t
    Tg.append(temp)
Tg = np.array(Tg)
```

Στο παρακάτω σχήμα φαίνεται το global masking threshold για κάποια παράθυρα. Όπως φαίνεται, στις ψηλές συχνότητες βλέπουμε το κατώφλι να αυξάνεται απότομα καθώς αυτές είναι οι συχνότητες που είναι πιο δύσκολα αντιληπτές από το ανθρώπινο αυτί.



Μέρος 2. Χρονο-Συχνοτική Ανάλυση με Συστοιχία Ζωνοπερατών Φίλτρων

Σκοπός του δεύτερου μέρους είναι η κβαντοποίηση και κωδικοποίηση του σήματος μουσικής που μας δίνεται. Η επεξεργασία γίνεται σε τετραγωνικά παράθυρα των N = 512 δειγμάτων μέσω της συνάρτησης compress (compress1 for the adaptive, compress2 for the not-adaptive) που επιστρέφει το ανακατασκευασμένο παράθυρο και τα bits που χρειάστηκαν για τον κβαντισμό του.
Το κάθε παράθυρο περνάει από 32 ημιτονοειδή φίλτρα (filterbank) που το χωρίζουν σε 32 διαφορετικές συχνοτικές περιοχές. Συνολικά λοιπόν, η επεξεργασία μας είναι χρονο-συχνοτική.
Για την κβαντοποίηση χρησιμοποιούμε το Tg που βρήκαμε στο Μέρος 1, το οποίο είναι παράγοντας στον καθορισμό των bits που χρησιμοποιούμε για την κωδικοποίηση κάθε subband κάθε παραθύρου στον προσαρμοσμένο κβαντιστή.

Ξεκινάμε την εργασία μας χωρίζοντας το σήμα σε τετραγωνικά παράθυρα. Για να εξασφαλίσουμε ότι τα παράθυρα είναι ισομήκη, επεκτείνουμε το σήμα με μηδενικά. Υπολογίζουμε τον συνολικό αριθμό bit που χρησιμοποιείται για την απεικόνιση του παραθυρομένου σήματος.

```
x = []
step = 512
for i in range(0, size, step):
    xi = signal[i:i+step]
    if len(xi) < step:
        z = np.zeros(step-len(xi))
        xi = np.concatenate((xi, z))
    x.append(xi)
x = np.array(x)
Nx = len(x)
bits_initial = step * Nx * 16
```

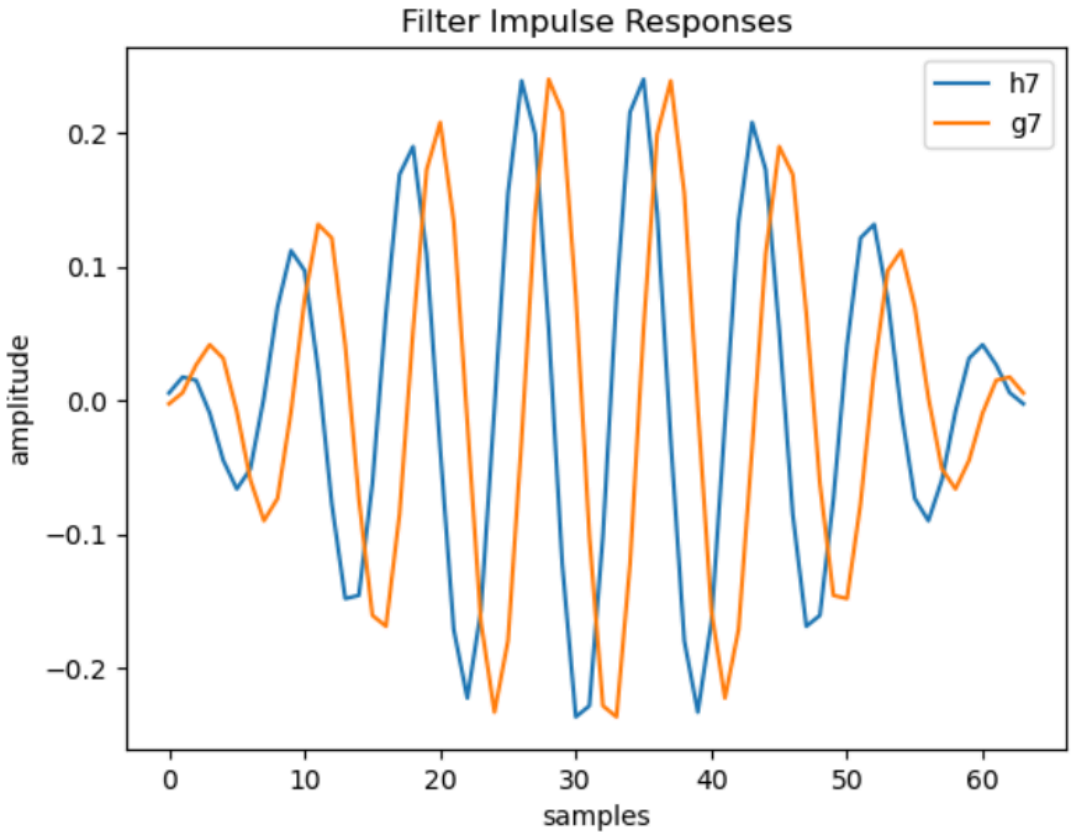
Βήμα 2.0: Συστοιχία Ζωνοπερατών Φίλτρων (Filterbank)

Αρχικά πρέπει να φτιάξουμε τα φίλτρα hk[n] και gk[n] τα οποία χρησιμοποιούμε για την διάσπαση του παραθύρου σε σήματα διαφορετικών συχνοτικών ζωνών και την αντίστοιχη ανακατασκευή του.

```
def sin_win(n, k, M):
    h = np.sin((n+0.5)*(pi/(2*M)))*np.sqrt(2/M)*np.cos(((2*n+M+1)*(2*k+1)*pi)/(4*M))
    return h
```

```
M = 32
L = 2*M
n = np.arange(L)
n2 = 2*M - 1 - n
h = []
g = []
for k in range(M):
    th = sin_win(n, k, M)
    tg = sin_win(n2, k, M)
    h.append(th)
    g.append(tg)
```

Το κάθε φίλτρο g προκύπτει από το συμμετρικό του αντίστοιχου φίλτρου h ως προς τον άξονα y, μετά από μετατόπιση κατά M σημεία προς τα δεξιά. Αυτό γίνεται εμφανές στο παρακάτω σχήμα.



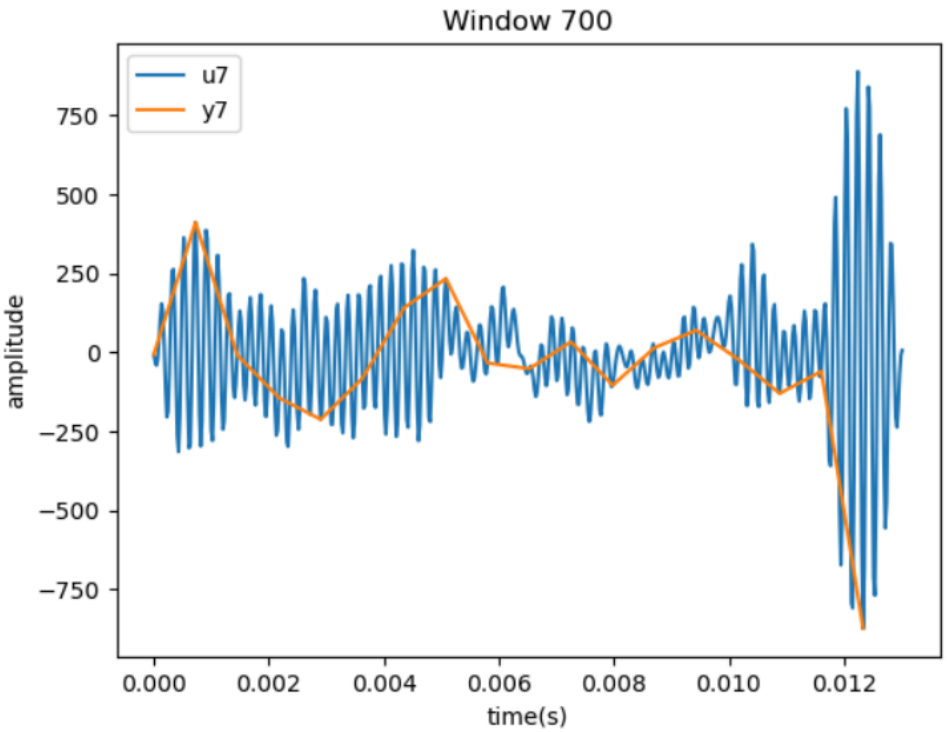
Βήμα 2.1: Ανάλυση με Συστοιχία Φίλτρων

Μετά τη δημιουργία των φίλτρων, το κάθε παράθυρο περνάει και από τα 32 με αποτέλεσμα να προκύπτουν 32 σήματα uk[n]. Αυτά τα σήματα καταλαμβάνουν μόνο το 1/32 του ακουστικού φάσματος , οπότε τα κάνουμε decimate με M = 32, ώστε να καταλαμβάνουν όλο το διαχειρίσιμο φάσμα. Φαινόμενα επικάλυψης λόγω μη-ιδανικότητας των φίλτρων που δεν κόβουν κάθετα τις συχνότητες αμελούνται.

Για την πραγματοποίηση του φίλτρου και του αποδεκατισμού φτιάξαμε την συνάρτηση conv_dec(x, h, M) που παίρνει σαν παραμέτρους το παράθυρο, το φίλτρο για κάποιο k και το M =32 για το decimation και επιστρέφει το σήμα y.

```
def conv_dec(x, h, M):
    y = []
    u = []
    for k in range(M):
        temp = sp.signal.convolve(x, h[k])
        temp2 = temp[::M]
        u.append(temp)
        y.append(temp2)
    u = np.array(u)
    y = np.array(y)
    return u, y
```

Φαίνεται στο σχήμα το παράθυρο 700, μετά από φιλτράρισμα στο 7ο subband σε σύγκριση με το αποτέλεσμα του decimation. Ο αριθμός δειγμάτων του σήματος που διατηρούμε φαίνεται να ακολουθεί ικανοποιητικά το αρχικό σήμα.



Βήμα 2.2: Κβαντοποίηση

*Για το κομμάτι της κβαντοποίησης, υλοποιούμε δύο εναλλακτικές συναρτήσεις συμπίεσης.

Η πρώτη είναι με προσαρμοζόμενο ομοιόμορφο κβαντιστή. Σε αυτή την υλοποίηση, ο αριθμός Bk των bits βρίσκεται σύμφωνα με το Tg του συγκεκριμένου παραθύρου και στις συχνότητες που μας ενδιαφέρουν σε κάθε subband μέσω του τύπου
"*Bk = int(log2(R/min(Tg)) - 1)*", όπου R το πλήθος των βαθμίδων του αρχικού σήματος. Λαμβάνει υπόψη και την μέγιστη και ελάχιστη τιμή του παραθύρου για το βήμα
Δ= (xmax - xmin)/2^Bk.

Η δεύτερη είναι με μη-προσαρμοζόμενο ομοιόμορφο κβαντιστή με σταθερό αριθμό bits B = 8 και στα σταθερό βήμα Δ = *(xmax - xmin)*/(2^B).
Μετά την κβαντοποίηση παίρνουμε το σήμα yq.

Η κβαντοποίηση υλοποιείται εντός των συναρτήσεων συμπίεσης, για κάθε παράθυρο και κβαντιστή. Επιτυγχάνεται με τη βοηθητική συνάρτηση quant η οποία αντιστοιχίζει κάθε δείγμα στην πλησιέστερη στάθμη κβαντισμού, εάν γνωρίζει το βήμα κβαντισμού D και την ελάχιστη τιμή του σήματος στο παράθυρο. Στην περίπτωση του μη-προσαρμοζόμενου κβαντιστή, το ελάχιστο λαμβάνει default ίση με την ελάχιστη του σήματος.

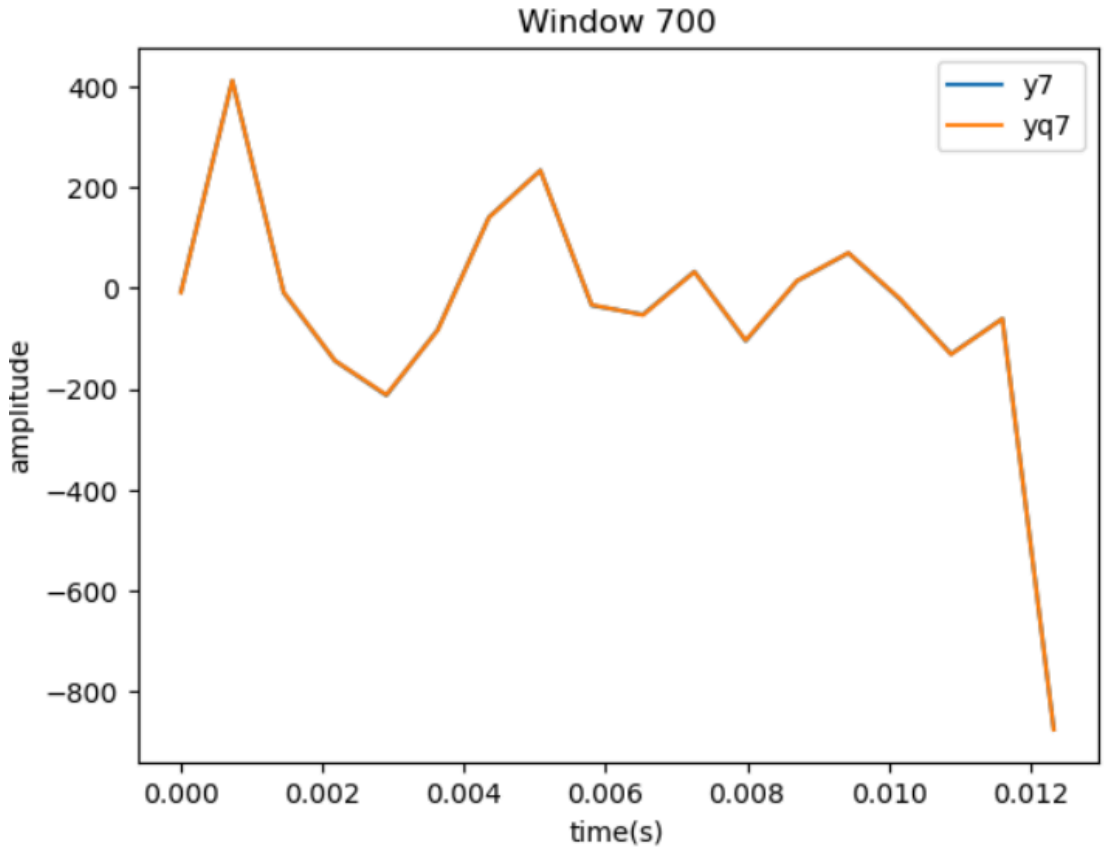
```
def quant(s, D, mv = np.min(signal)):
    return mv + D*np.round((s-mv)/D)
```

Για την εύρεση των bits, χρησιμοποιούμε τον παραπάνω τύπο, και συγκεκριμένα, για το min(Tg) ορίζουμε τον πίνακα index που αντιστοιχεί στις κεντρικές συχνότητες κάθε subband $f_k = \frac{(2k-1)F_s}{4M}$. Κάθε διάστημα στο οποίο μας ενδιαφέρει το Tg έχει πλάτος Fs/2M.

Προσαρμοζόμενος κβαντιστής:

```
#2.2 quantize
index = [round((2*k-1)*N/(4*M)) for k in range(1,M+1)]
A = N//(4*M)
R = 2**(signal.itemsize * 8)
bits = np.zeros(M, dtype= np.int16)
Delta = np.zeros(M)
yq = []
for i in range(M):
    d = np.min(Tg[(index[i]-A):(index[i]+A)])
    bits[i] = np.int16(np.log2(R/d)-1)
    Delta[i] = (np.max(y[i]) - np.min(y[i])) / (2**bits[i])
    t = quant(y[i], Delta[i], np.min(y[i]))
    yq.append(t)
yq = np.array(yq)
```

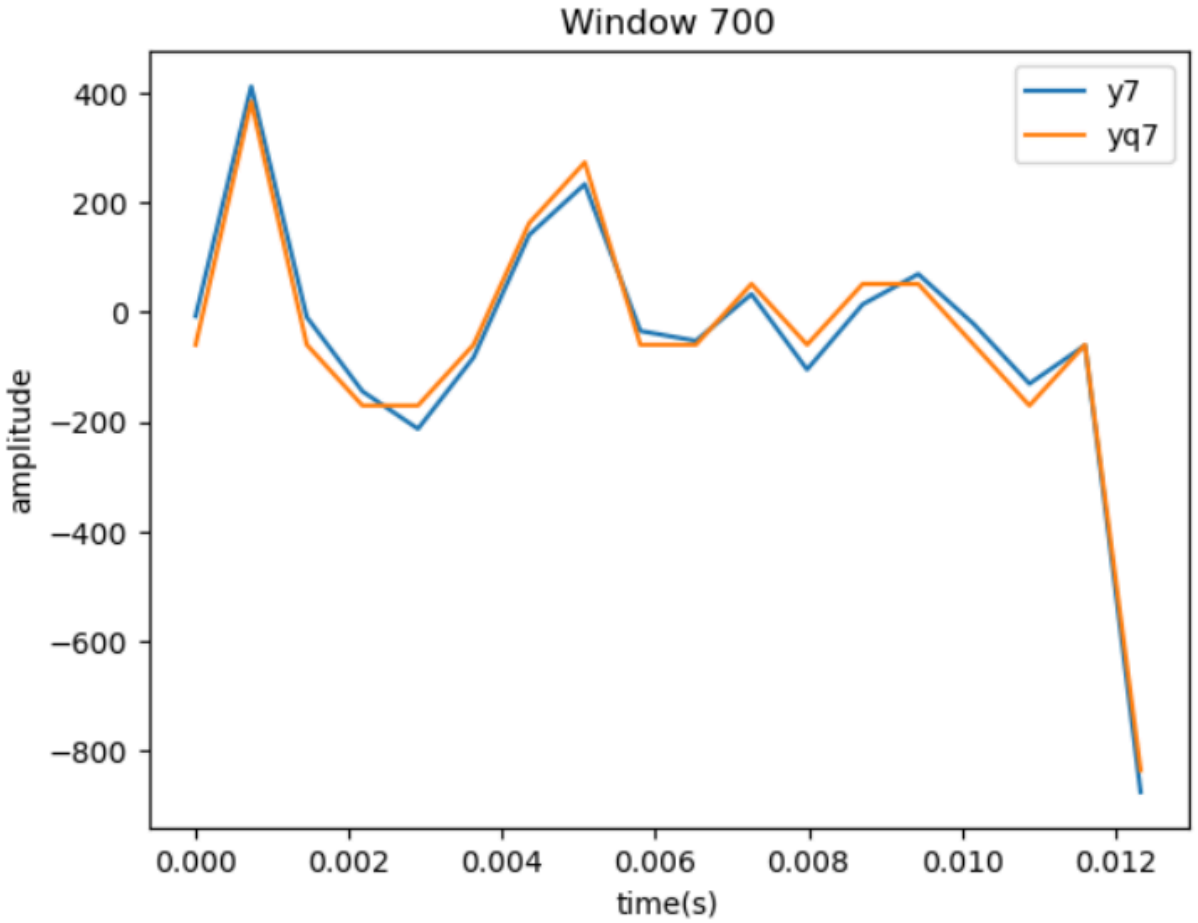
Η έξοδος του προσαρμοζόμενου κβαντιστή για το 700ο παράθυρο, φαίνεται να ταυτίζεται με το σήμα πριν την κβάντιση.



Μη-προσαρμοζόμενος κβαντιστής:

```
#2.2 quantize
bits = 8
Delta = (y.max()-y.min())/(2**bits)
yq = []
for i in range(M):
    t = quant(y[i], Delta)
    yq.append(t)
yq = np.array(yq)
```

Ο μη-προσαρμοζόμενος κβαντιστής δεν ακολουθεί το σήμα όσο πιστά ο προσαρμοζόμενος. Αυτό είναι λογικό, εφόσον έχουμε λιγότερα επίπεδα κβαντισμού.



Βήμα 2.3: Σύνθεση

Η σύνθεση του σήματος μετά την κωδικοποίηση συμβαίνει σε δύο στάδια. Το πρώτο στάδιο πραγματοποιείται εντός του παραθύρου και αφορά την υπερδειγματοληψία του σήματος yq ώστε να μειωθεί ξανά το εύρος ζώνης του, και από όπου παίρνουμε το σήμα w, και το πέρασμα του w από το φίλτρο σύνθεσης g που φτιάξαμε στην αρχή. Έτσι προκύπτει το σήμα what.

Μόλις αθροίσουμε όλα τα αποτελέσματα των συνελίξεων με τα σήματα gk για k = 1,...32 παίρνουμε το ανακατασκευασμένο σήμα xr από το κάθε παράθυρο x.

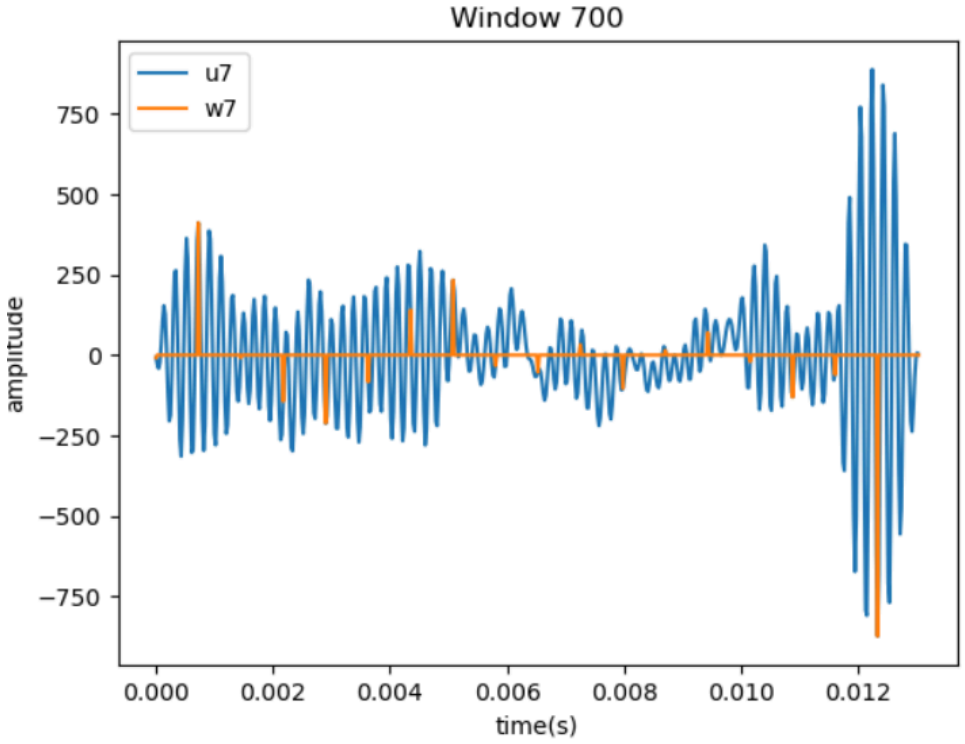
Για να εκτελέσουμε αυτή τη λειτουργία, χρησιμοποιούμε τη συνάρτηση int_conv.

```
def int_conv(yq, g, M):
    w = []
    what = []
    for i in range(M):
        t = np.zeros(len(yq[i])*M)
        t[::M] = yq[i]
        w.append(t)
        t2 = sp.signal.convolve(w[i], g[i])
        what.append(t2)
    what = np.array(what)
    w = np.array(w)
    xr = np.sum(what, axis=0).astype(np.int16)
    return xr, what, w
```

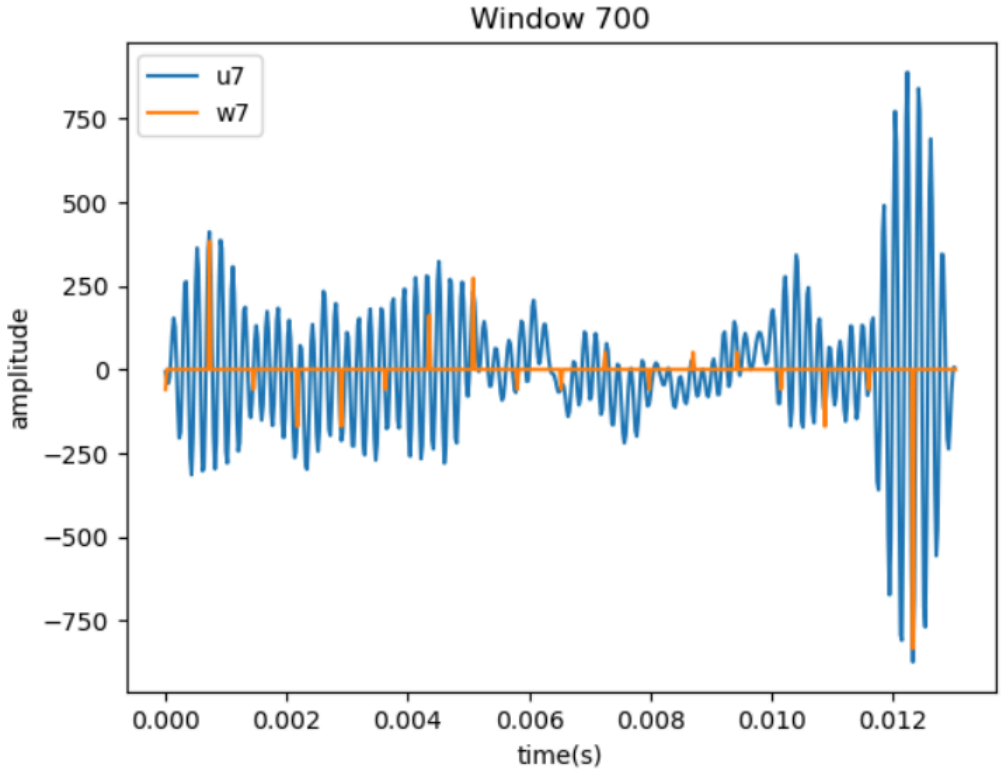
Φροντίζουμε το ανακατασκευασμένο σήμα να έχει ίδιο data type με το αρχικό, ώστε μετά την “εικόνικη” διαδικασία κβάντισης, να μην αυξηθεί ο αριθμός των bit ανά δείγμα. Πριν επαναφέρουμε το datatype σε int16, είχε μετατραπεί σε float64 κατά τον υπολογισμό του βήματος κβάντισης που είναι δεκαδικός αριθμός.

Στο παρακάτω σχήμα, φαίνεται το σήμα του 7ου subband του 700ου παραθύρου πριν το decimation, σε αντιπαράθεση με την κατάσταση του μετά το interpolation.

Προσαρμοζόμενος Κβαντιστής



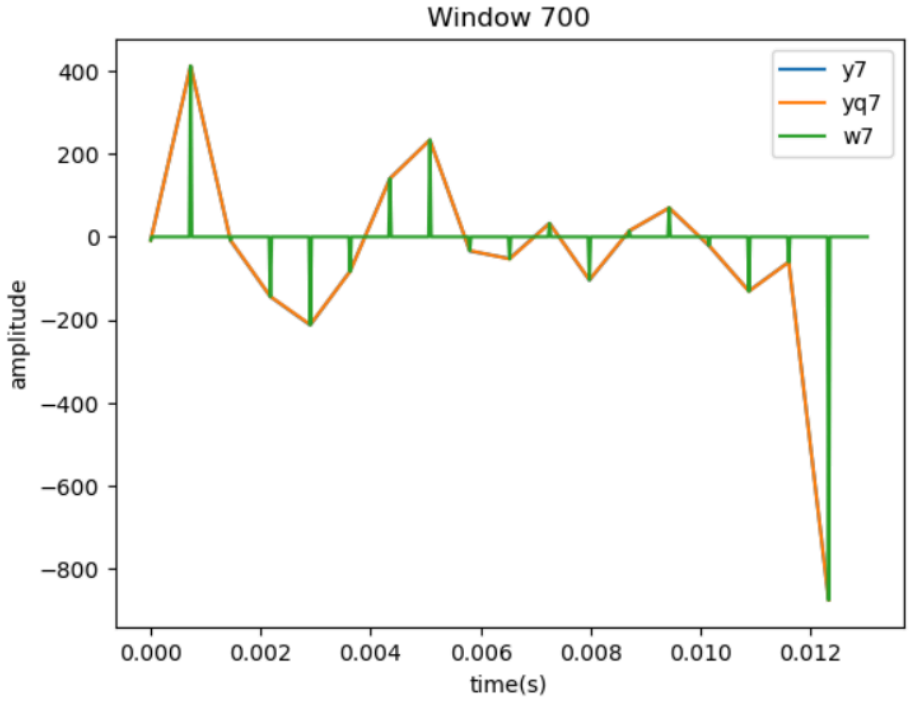
Μη προσαρμοζόμενος κβαντιστής



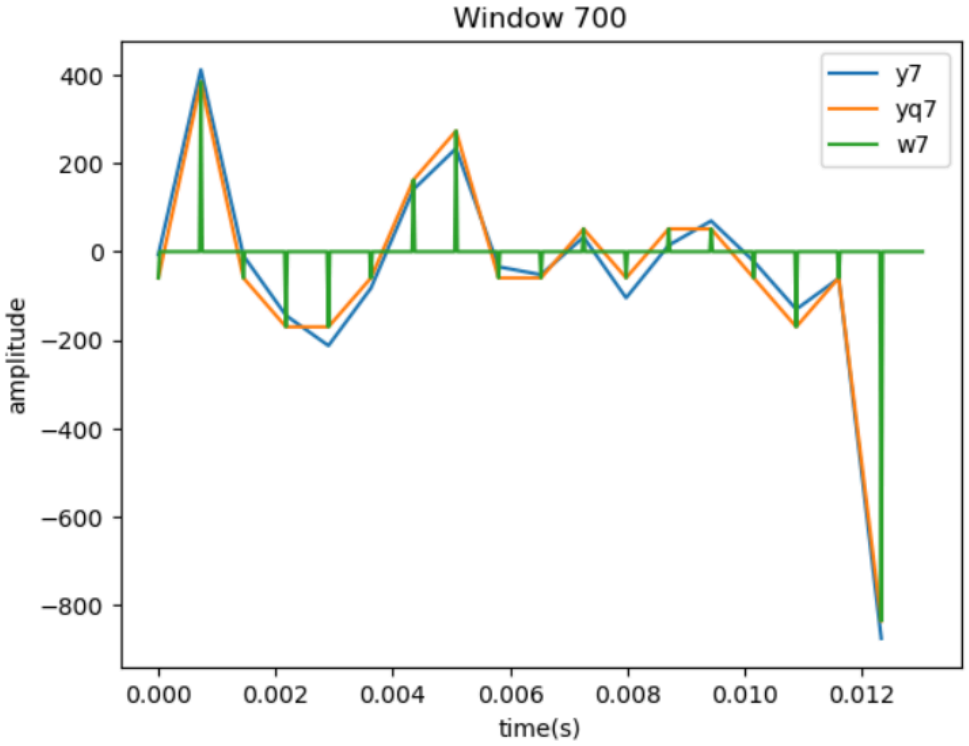
Όπως είναι λογικό, οι αποκλίσεις είναι μεγαλύτερες

Παρακάτω, το σήμα μετά το interpolation φαίνεται σε σύγκριση με το κβαντισμένο σήμα, από του οποίου την υπερδειγματοληψία προήλθε.

Προσαρμοζόμενος κβαντιστής



Μη προσαρμοζόμενος



Με τη χρήση των συναρτήσεων compress , αποθηκεύουμε σε πίνακες τα ανακατασκευασμένα παράθυρα.

Στην περίπτωση του προσαρμοζόμενου κβαντιστή, ανακτούμε και τον αριθμό bit που χρειάστηκε για την κωδικοποίηση κάθε παραθύρου.
Υπολογίζουμε, παράλληλα, και τον συνολικό αριθμό των bit που χρειάστηκαν για την κωδικοποίηση, ανά παράθυρο και συχνοτική μπάντα.

Μη προσαρμοζόμενος κβαντιστής	`xr2 = []`
	`for win in range(Nx):`
	`  xwinr = compress2(x[win], h, g, Tg[win], M, win)`
	`  xr2.append(xwinr)`
	`xr2 = np.array(xr2)`

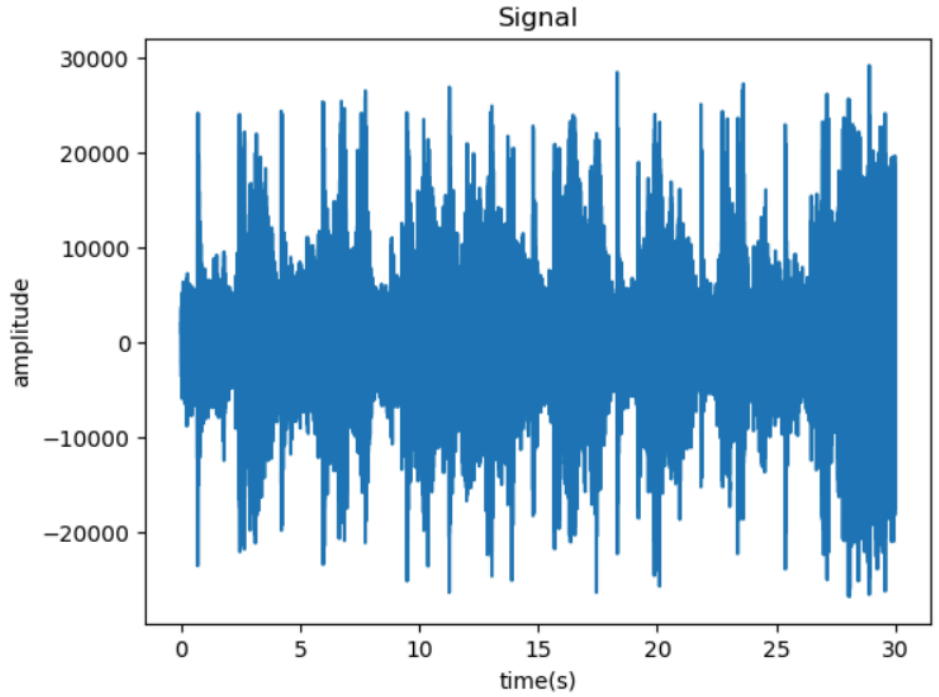
Προσαρμοζόμενος κβαντιστής:	`xr1 = []`
	`Bk1 = []`
	`bits_final1 = 0`
	`for win in range(Nx):`
	`  xwinr, b = compress1(x[win], h, g, Tg[win], M, win)`
	`  for i in range(M):`
	`  bits_final1 += (step/M)* b[i]`
	`  xr1.append(xwinr)`
	`  Bk1.append(b)`
	`xr1 = np.array(xr1)`

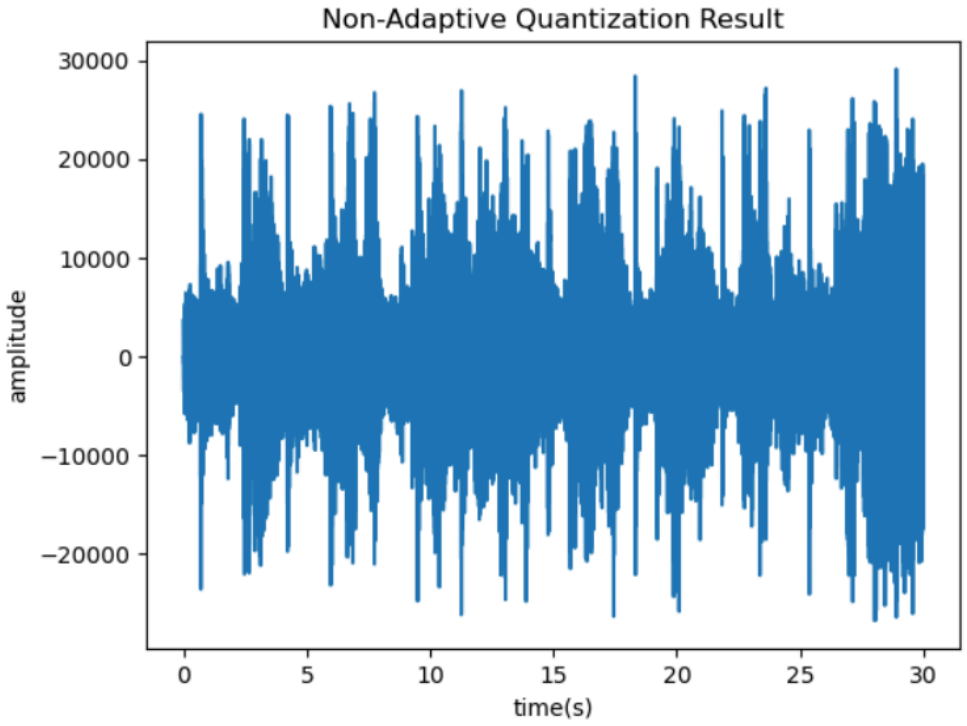
Οδηγούμαστε λοιπόν στο δεύτερο στάδιο της σύνθεσης που αφορά την στοίχιση όλων των ανακαστευασμένων παραθύρων xr, η οποία υλοποιείται με Overlap Add.

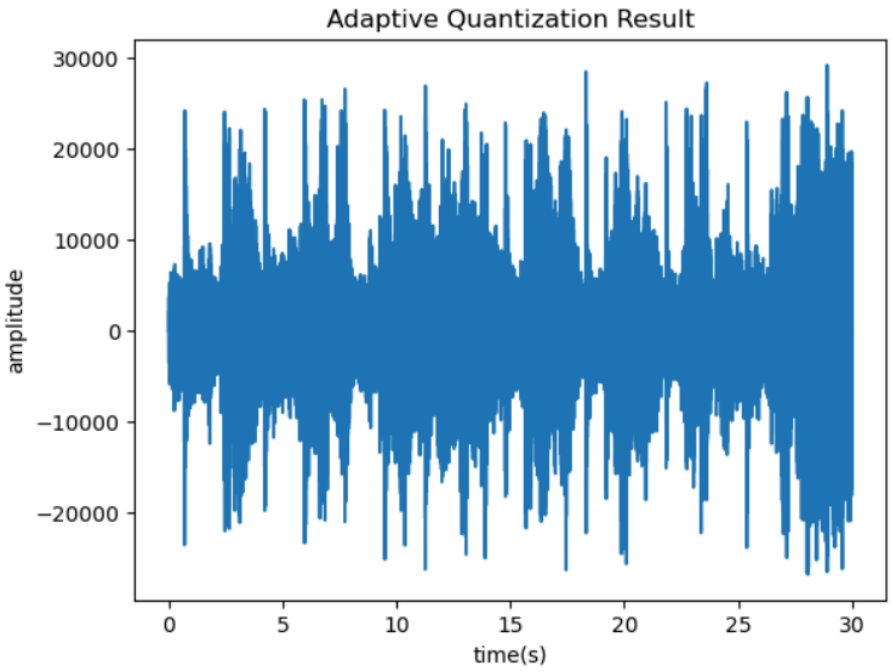
Προσαρμοζόμενος κβαντιστής:	`step = N`
	`ola = len(xr1[0])`
	`final1 = np.zeros(Nx*step+len(g[0])+len(h[0]), dtype=np.int16)`
	`for win in range(Nx):`
	`  final1[win*step:win*step+ola] += xr1[win]`
	`final1.astype(np.int16)`

Μη-προσαρμοζόμενος κβαντιστής:
 	
Υπολογίζουμε τον συνολικό αριθμό bit που χρειάστηκαν σε αυτή την υλοποίηση.
	`step = N`
	`ola = len(xr2[0])`
	`final2 = np.zeros(Nx*step+len(g[0])+len(h[0]), dtype=np.int16)`
	`for win in range(Nx):`
	`  final2[win*step:win*step+ola] += xr2[win]`
	`final2.astype(np.int16)`
	`bits_final2 = Nx * step * 8`

Με τη συνάρτηση Audio της IPython display ακούμε τα ανακατασκευασμένα σήματα. Τα τελικά σήματα ακούγονται όμοια με το αρχικό. Οι γραφικές τους παραστασεις φαίνονται επίσης όμοιες.

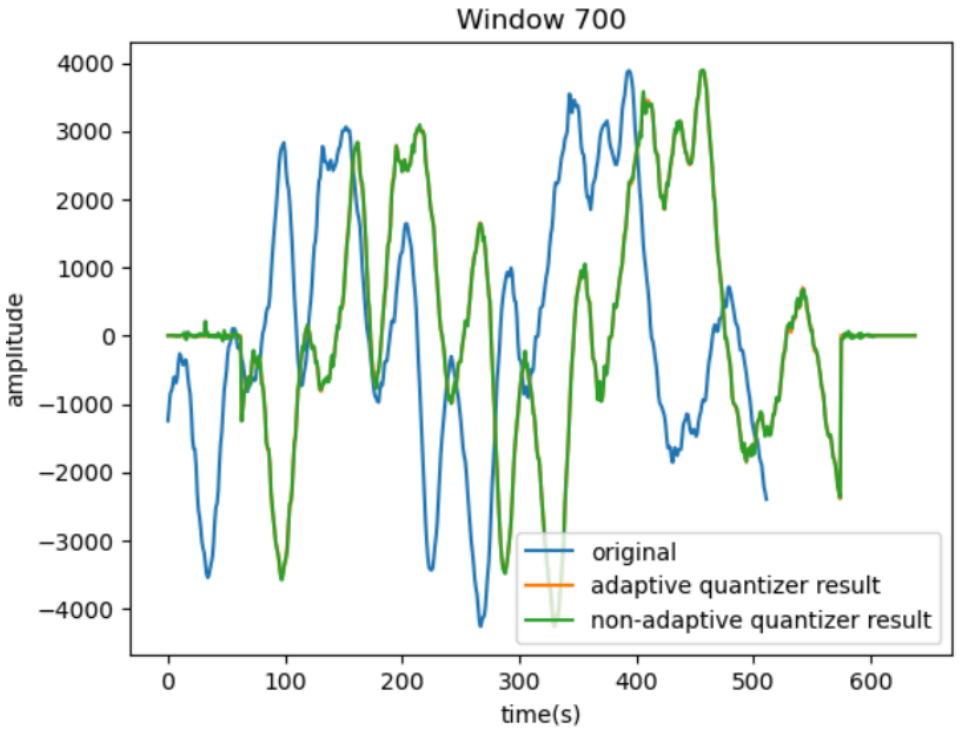






Παρατηρούμε, όμως, μελετώντας το παράθυρο 700, πριν και μετά την επεξεργασία, παρατηρούμε ότι τα φίλτρα σύνθεσης εισάγουν μια καθυστέρηση 64 δειγμάτων.

```
w = 700
plt.plot(x[w])
plt.plot(xr1[w])
plt.plot(xr2[w])
```



Αποτελέσματα σύγκρισης

Ποσοστό συμπίεσης

Υπολογίζουμε τα ποσοστό συμπίεσης με τη συνάρτηση compr_rate.

```
def compr_rate(bits_initial, bits_final):
    return 1- bits_final/bits_initial
```

```
The compression rate using the non-adaptive compressor is 0.5
```

Αυτό είναι αναμενόμενο, εφόσον στον μη-προσαρμοζόμενο κβαντιστή έχουμε δείγματα μεγέθους
8 bit, ίσου με το ½ του αρχικού μεγέθους.

```
The compression rate using the adaptive compressor is 0.5017596265479876
```

Το compression rate είναι λίγο μεγαλύτερο. Αυτό είναι λογικό γιατί ο συνολικός αριθμός bit είναι μικρότερος. Οι συχνοτικές μπάντες που δε γίνονται αντιληπτές από το ανθρώπινο αυτί κωδικοποιούνται με λιγότερα bit.

Μέσο τετραγωνικό λάθος

Λαμβάνοντας υπόψη τη καθυστέρηση, υπολογίζουμε το μέσο τετραγωνικό σφάλμα για κάθε κβαντιστή.

```
start = 63
end = strt+len(signal)
mse2 = np.mean((final2[strt:end]-signal)**2)
```

```
The MSE using the non-adaptive compressor is 3637.973558578987
```

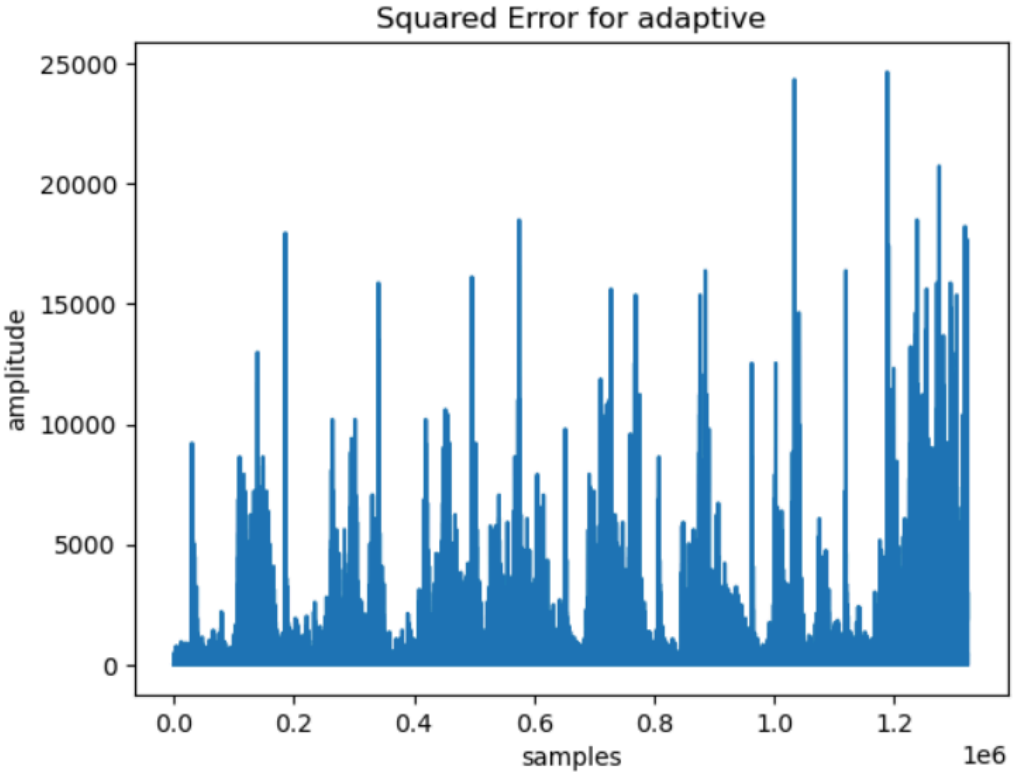
```
start = 63
end = start+len(signal)
mse1 = np.mean((final1[start:end]-signal)**2)
```

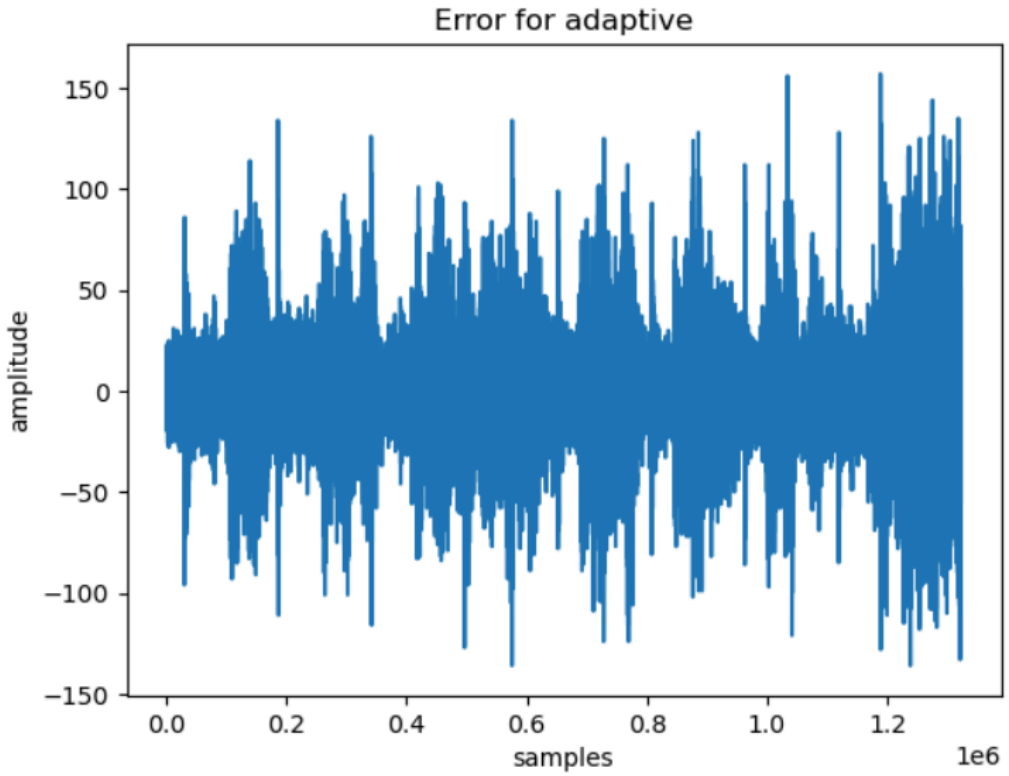
```
The MSE using the adaptive compressor is 340.47830007558576
```

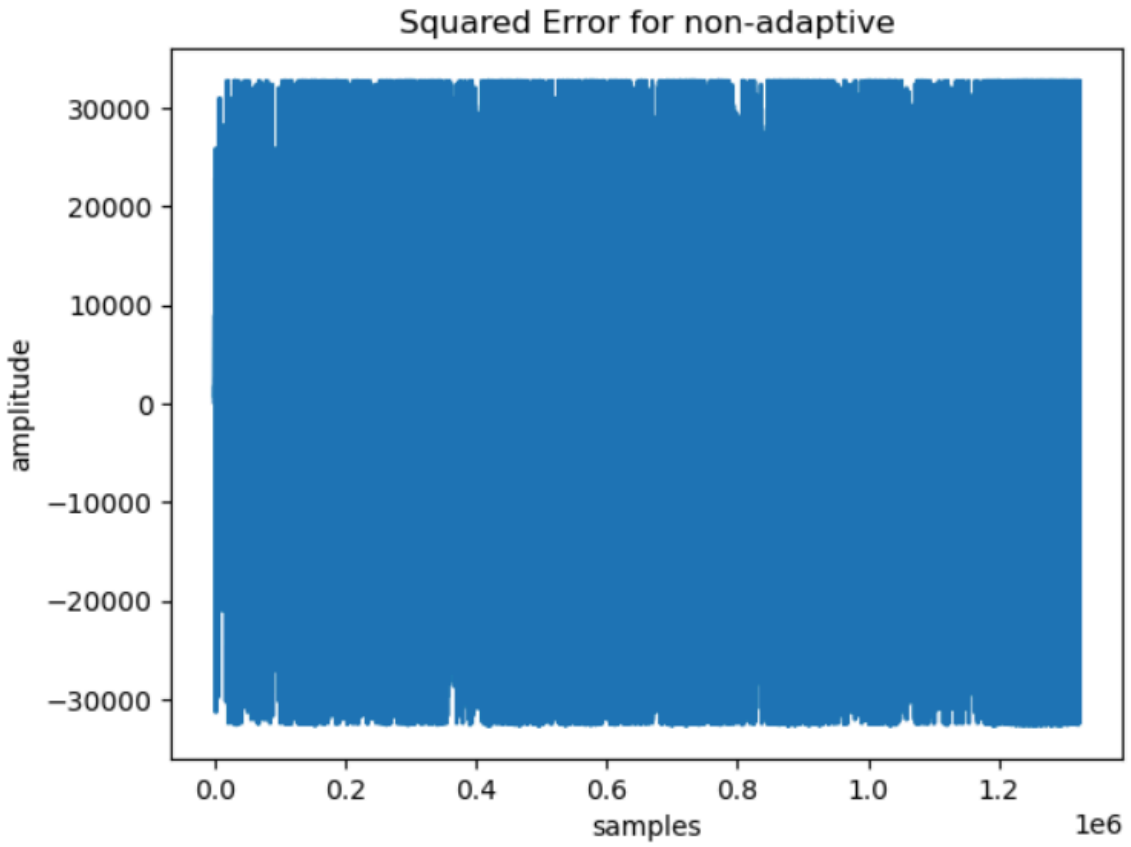
Η αύξηση του μέσου τετραγωνικού σφάλματος στον μη-προσαρμοζόμενο κβαντιστή οφείλεται στο ότι οι συχνοτικές ζώνες που είναι πιο σημαντικές με βάση την ανάλυση του ψυχοακουστικού μοντέλου κωδικοποιούνται με λιγότερες στάθμες. Έτσι, οδηγούμαστε σε μεγαλύτερα σφάλματα.

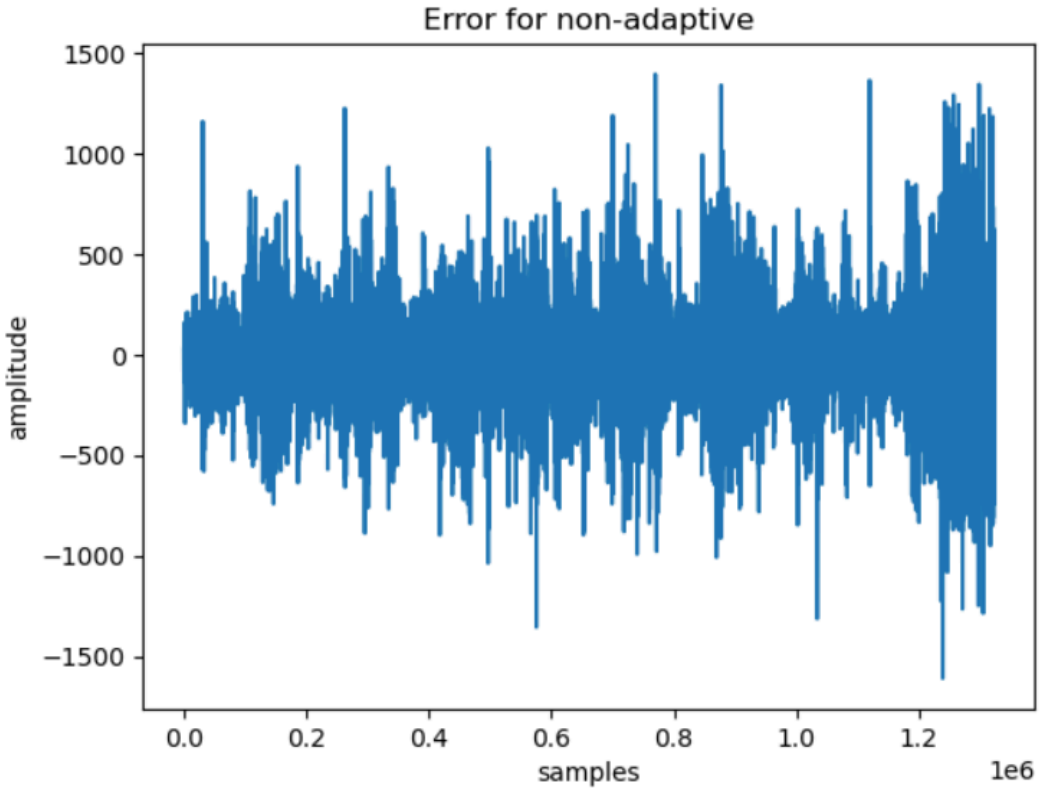
Απεικόνιση Σφάλματος

Φαίνεται το τετραγωνικό και το άμεσο σφάλμα των ανακατασκευασμένων σημάτων, σε σύγκριση με το αρχικό σήμα μουσικής. Στην περίπτωση του προσαρμοζόμενου κβαντιστή, το σφάλμα είναι αισθητά μικρότερο.






Squared Error for non-adaptive
30000
20000
10000
0
−10000
−20000
−30000
amplitude
0.0
0.2
0.4
0.6
0.8
1.0
1.2
samples
1e6


Error for non-adaptive
1500
1000
500
0
−500
−1000
−1500
amplitude
0.0
0.2
0.4
0.6
0.8
1.0
1.2
samples
1e6