

# Ψηφιακή Επεξεργασία Σημάτων



Τμήμα Ηλεκτρολογικών Μηχανικών &  
Μηχανικών Τεχνολογιών

## Εργαστηριακή Αναφορά 2:

**Θέμα: Κωδικοποίηση Σημάτων Μουσικής Βάσει του Ψυχοακουστικού μοντέλου.**

Φοιτητές: Δωροθέα Κουμίδου 03119712  
Γιώργος Χαραλάμπους 03119706

### Μέρος 1

#### Ψυχοακουστικό Μοντέλο 1

Αρχικά εισάγουμε το σήμα μας και αποθηκεύουμε τα πλάτη του σήματος ως `sgn` ενώ την συχνότητα δειγματοληψίας `fs`. Στην συνέχεια μετατρέπουμε το σήμα μας από στέρεο σε μονό αθροίζοντας τα δύο channels και το κανονικοποιούμε διαιρώντας το με το απόλυτο μέγιστο πλάτος του αθροισμένου σήματος.

```
# Εισαγωγή αρχείων
# Δωροθέα Κουμίδου 03119712
# Γιώργος Χαραλάμπους 03119706

# Πάθος 1
# Όνομα 1.0

import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import soundfile as sf

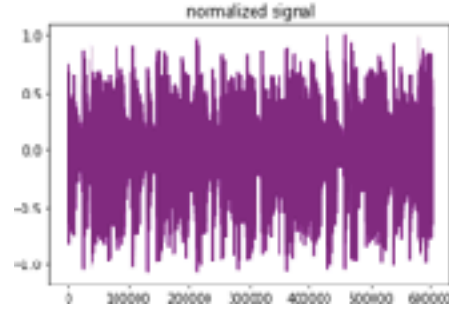
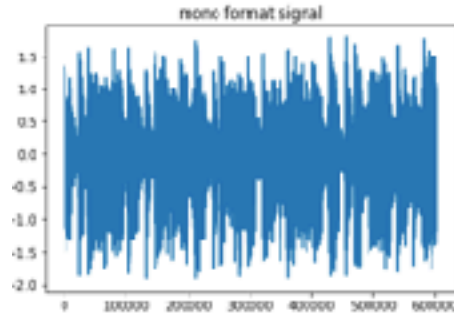
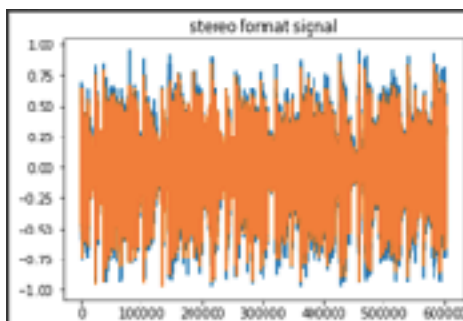
# σgn, fs = sf.read('src/101.wav') # stored also for the next steps as needed
print(fs)
print(len(sgn))

plt.figure(0)
plt.plot(sgn)
plt.title('stereo format signal')

# μετατρέπουμε stereo σε mono
mono_sgn = np.sum(sgn, axis=0)
plt.figure(1)
plt.plot(mono_sgn)
plt.title('mono format signal')

# κανονικοποιούμε το σήμα
mono_norm_sgn = mono_sgn / np.max(np.abs(mono_sgn))

plt.figure(2)
plt.plot(mono_norm_sgn, color='purple')
plt.title('normalized signal')
```



Μετά παραθυροποιούμε το σήμα μας σε πλαίσια των 512 δειγμάτων όμως με μια ματιά παρατηρήσαμε ότι το τελευταίο παράθυρο έχει λιγότερα δείγματα έτσι προσθέσαμε μηδενικά στο τέλος. Η εντολή concatenate είναι αυτή που μας βοήθησε να ενώσουμε τα τελευταία δείγματα με τα μηδενικά και να αναγνωρίζεται από το πρόγραμμα ως ένα παράθυρο των 512.

```

def main():
    # Load the signal
    signal = load('signal.mat')
    # Resample the signal to 1000 Hz
    sr = 1000
    signal = resample(signal, sr)
    # Split the signal into frames of 512 samples
    frames = signal.reshape((-1, 512))
    # Add a window to each frame
    window = hanning(512)
    frames *= window
    # Compute the magnitude spectrum for each frame
    spectra = []
    for i in range(frames.shape[0]):
        spectrum = abs(np.fft.fft(frames[i, :]))
        spectra.append(spectrum)
    # Average the spectra
    avg_spectrum = np.mean(spectra, axis=0)
    # Plot the magnitude spectrum
    plt.plot(avg_spectrum)
    plt.show()

```

Έπειτα, δημιουργούμε μια συνάρτηση για την κλίμακα Bark και συμπληρώνουμε τον πίνακα Pk που αποτελεί το φάσμα ισχύος με τρόπο ώστε τελικά να κρατάμε το μονόπλευρο σήμα όπως φαίνεται παρακάτω.

```

[2] #Bark 1.1
import math

def bark2hz(b):
    hz = 1000 * math.exp(0.00068 * b) * 0.5 * math.exp(-0.0001 * b)
    return hz

# Create the bark scale
bark = np.zeros(512)

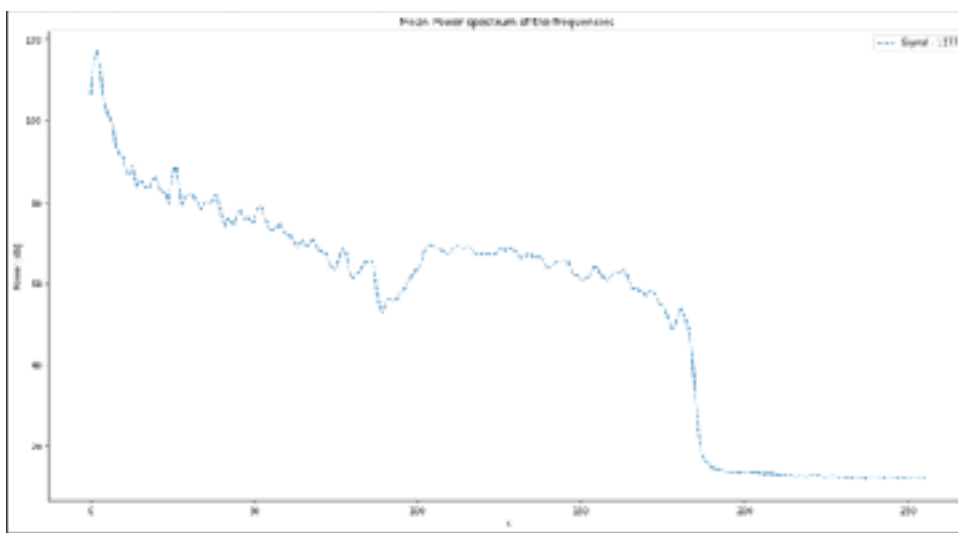
#Bark 1.1
Pk = []
for n in range(1):
    pk = np.zeros(512)
    for k in range(512//2):
        pk[k] = np.sqrt(2 * np.log10(avg_spectrum[k] + 1e-12) * np.pi * k * (k+1) * 0.0001)
    Pk.append(pk)

[4] print(np.array(Pk).shape)
print(np.transpose(Pk).shape)

summedPk = np.mean(Pk, axis=0)
print(summedPk.shape)
print(summedPk[256])

```

\* Για να βοηθηθούμε αποθηκεύσαμε το μέσο όρο όλων των πλαισίων για κάθε συχνότητα και αυτό παραστήσαμε. Ουσιαστικά το αρχικό power spectrum είχε διαστάσεις [256, 1179], τώρα είναι [256,1].



Στο βήμα 1.2 μας ζητήθηκε να βρούμε τις τονικές μάσκες οι οποίες είναι μεγαλύτερα από τις γειτονικές τους συχνότητες τουλάχιστον κατά 7 dB. Για να το καταφέρουμε αυτό, αρχικά φτιάξαμε ένα πίνακα με αποθηκευμένες τις τιμές του  $\Delta k$  και ορίσαμε τον πίνακα  $St$  ως ένα πίνακα γεμάτο μηδενικά και κάθε φορά που βρίσκει τονική μάσκα το μηδέν να αντικαθιστάται από 1, έτσι μας δείχνει την θέση που υπάρχει μάσκα. Άρα περιμένουμε να έχουμε ένα πίνακα και πάλι [256,1]

```

def select_Dk(k):
    if k%40:
        return [0]
    elif k%127:
        return [2,2]
    elif k%256:
        return np.arange(2,73)

def findSelect_Dk(128):
    St = np.zeros(256).astype(int)
    for k in range(5,256):
        if cummedPsk(k) > cummedPsk(k-4) and cummedPsk(k) > cummedPsk(k-4):
            mask = 1
            for Dk in select_Dk(k):
                if cummedPsk(k) > cummedPsk(k-Dk) and cummedPsk(k) > cummedPsk(k-Dk):
                    mask += 1
            St[k] = mask
    return St

# Plot the St matrix
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.imshow(St)

```

Αφού βρήκαμε τις θέσεις των τονικών μαस्कών, υπολογίσαμε την ισχύ τους με βάσει τον τύπο:

$$P_{TAR}(k) = \begin{cases} 10 \log_{10}(10^{0.3(P(k-1))} + 10^{0.3(P(k))} + 10^{0.3(P(k+1))}) \text{ (dB)}, & \text{αν } S_T(k) = 1 \\ 0, & \text{αν } S_T(k) = 0 \end{cases}$$

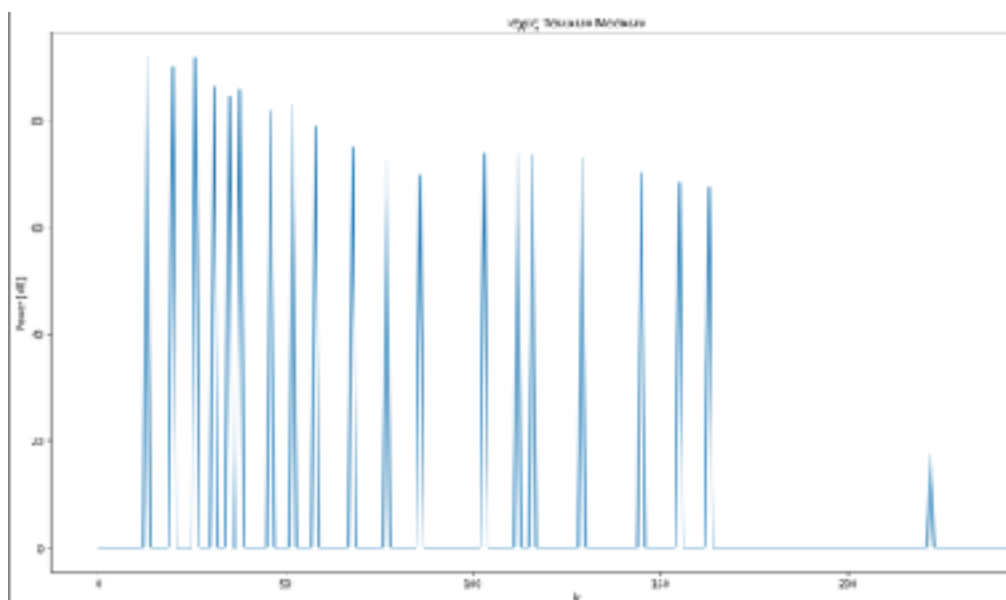
```

(7) # Power Tonics Maskes
P_TM=0
for k in range(10,256):
    if St[k]==1:
        p1=0.0001*10**(0.3*cummedPsk(k-4)+0.3*cummedPsk(k-4))
        if St[k]==1:
            p2=0.0001*10**(0.3*cummedPsk(k-4)+0.3*cummedPsk(k))
        if St[k]==0:
            p3=0
        else:
            p3=0.0001*10**(0.3*cummedPsk(k-4)+0.3*cummedPsk(k)+0.3*cummedPsk(k+4))
        P_TM.append(p1+p2+p3)

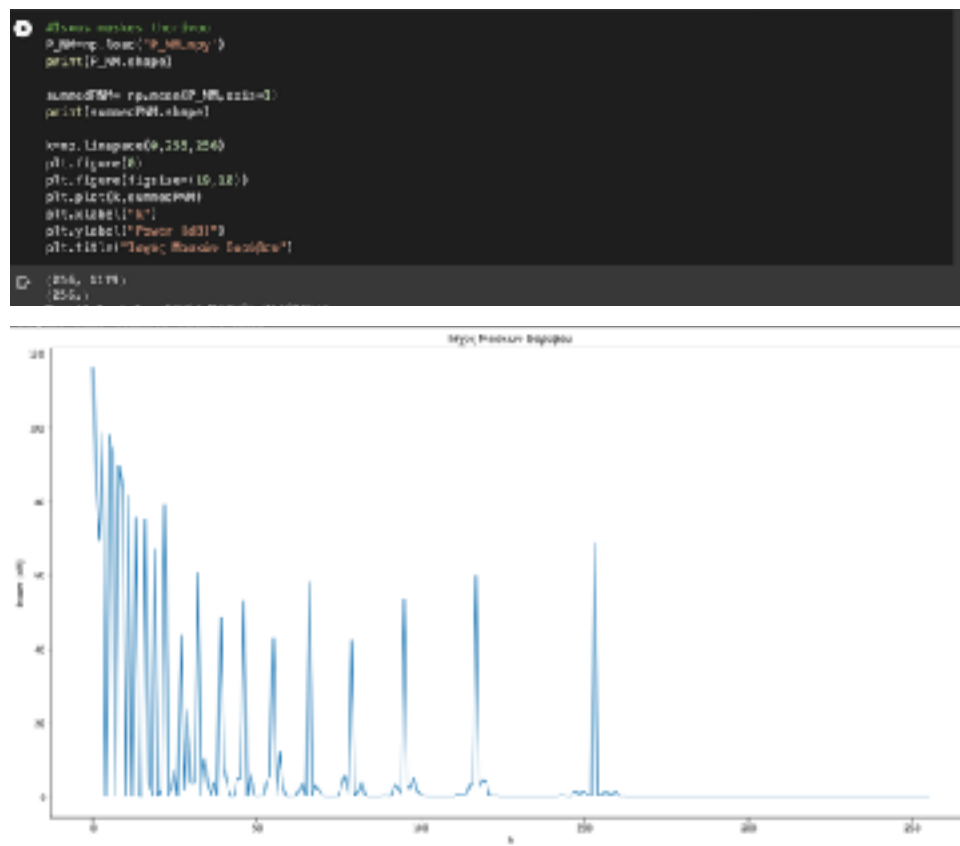
# Plot the P_TM matrix
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.imshow(P_TM)

(8) # Power Tonics Maskes
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(P_TM)
ax.set_xlabel('k')
ax.set_ylabel('Power [dB]')
ax.set_title('Power Tonics Maskes')

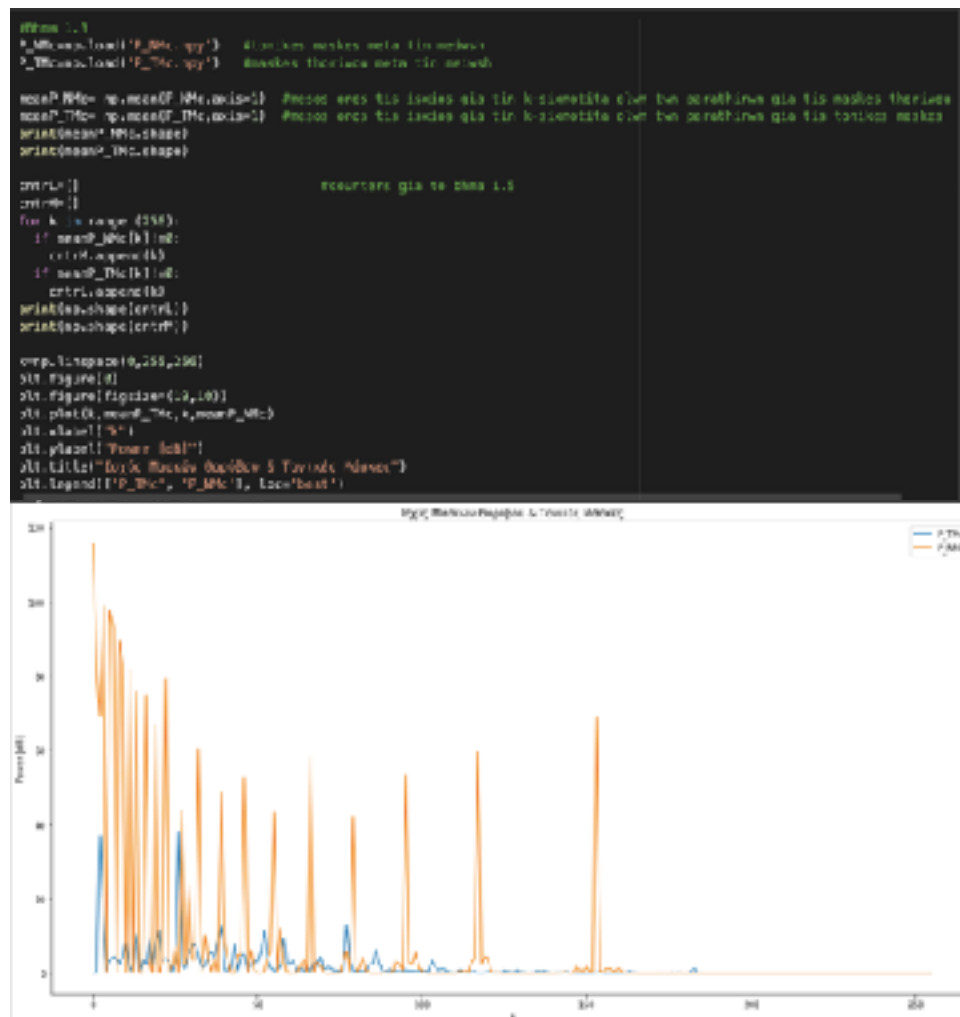
```



Παρακάτω φαίνονται και οι μάσκες θορβού.



Στο βήμα 1.3 εισάγαμε τα δύο αρχεία και βρήκαμε το μέσο όρο του καθενός ξεχωριστά. Στον κώδικα φαίνεται επίσης ο counter για το βήμα 1.5



Στο επόμενο βήμα τα δύο διαφορετικά κατώφλια κάλυψης για την μάσκα τόνου και θορύβου με βάσει τους τύπους

$$T_{TM}(i, j) = P_{TM}(j) - 0.275b[j] + SF(i, j) - 6.025(\text{dB SPL})$$

$$T_{NMF}(i, j) = P_{NMF}(j) - 0.175b(j) + SF(i, j) - 2.025 \text{ [dB SPL]}$$

Αρχικά όπως φαίνεται πιο κάτω καλέσαμε την συνάρτηση που φτιάξαμε πιο πάνω για την κλίμακα Bark. Μετά ορίσαμε πίνακες SF για τις συνάρτησεις SF των τονικών μασκών και των μασκών θορύβου ξεχωριστά, και δουλέψαμε με τον ίδιο τρόπο για την εύρεση των ST.

```

nNodes = 8
dx = 0.5
for i in range(1,257):
    tdata[256dx:256i]
    tdata[i]
    tdata.append(0)

SP = zeros(256)
QF = zeros(256)
for i in range(256):
    for j in range(256):
        Bdata[i][j]
        if Bdata and Bdata:
            SP_Te[i][j] = 17400 + 0.4 * meanP_Te[i][j] + 1
            SP_Bn[i][j] = 17400 + 0.4 * meanP_Bn[i][j] + 1
        if Bdata and Bdata:
            SP_Te[i][j] = 18400 + meanP_Te[i][j] + 0.5
            SP_Bn[i][j] = 18400 + meanP_Bn[i][j] + 0.5
        if Bdata and Bdata:
            SP_Te[i][j] = 0.5 * (1 - T)
            SP_Bn[i][j] = 0.5 * (1 - T)
        if Bdata and Bdata:
            SP_Te[i][j] = 0.5 * meanP_Te[i][j] + 0.5 * meanP_Te[i][j]
            SP_Bn[i][j] = 0.5 * meanP_Bn[i][j] + 0.5 * meanP_Bn[i][j]

print('Shape: ', SP.shape)

T_Temp = zeros(256)
T_Bn = zeros(256)
for i in range(256):
    for j in range(256):
        T_Temp[i][j] = meanP_Te[i][j] - 0.25 * Bdata[i][j] - SP_Te[i][j] - 0.5
        T_Bn[i][j] = meanP_Bn[i][j] - 0.25 * Bdata[i][j] - SP_Bn[i][j] - 0.5
print('T_Temp')
print('T_Bn')

```

Με βάση τα προηγούμενα, στο βήμα 1.5 βρίσκουμε το συνολικό κατώφλι κάλυψης χρησιμοποιώντας τον τύπο:

$$T_p(i) = 10 \lg_{10} \left( 10^{0.1 T_p(i)} + \sum_{k=1}^L 10^{0.1 T_{\text{rad}}(i,k)} + \sum_{m=1}^M 10^{0.1 T_{\text{non}}(i,m)} \right) \text{ dB SPL}$$

```
def ath(t):
    t0=0.0;mp.power(t/1000,-0.5)-0.5*mp.e+1-0.5*(mp.power((t/2000)-2.2,2))+(200-2)*mp.power(t/1000,t)
    return t0

Tq=[]
for i in range(0,255):
    t0=22850/256
    t=ath(t0)
    Tq.append(t)
print(Tq)
T0=[]
for i in range(256):
    sum_T0=i
    sum_M0=
    for j in range(1):
        sum_T0+=100*(0.5-T0[i]/11)
    T0=0.5*(0.5-T0)
    sum_M0+=100*(0.5-T0[i]/11)
    t0=0.5*mp.log(1+100*(0.5-T0[i]/11)-sum_T0-sum_M0)
    Tq.append(t0)

plt.figure(0)
plt.figure(figsize=(10,10))
plt.plot(Tq,color='b')
plt.xlabel('Time/s')
plt.ylabel('Magnitude/dB')
plt.title('Mikael Backing Threshold')
print(Tq)
```



Για το βήμα 2.1, χρησιμοποιήσαμε την μεταβλητή `filterNm` και κάναμε την υποδειγματοληψία του σήματος κατά παράγοντα  $M=32$ .

```
filterNm2                                     #Number of filters
for k in range(FilterNm):                     #for each filter do
    # filter with M
    #Mm= 2.1
    uk = [sp.convolve(row,Hk[k], 'same') for row in sgsplit] #([3279, 512] convolution of the splitted signal with each
    # Downsample, Decimation
    yk = [k[::32] for k in uk] #([3279, 16]
```

Σε αυτό το βήμα, μας ζητείται να κβαντοποιήσουμε το σήμα με δύο τρόπους:

α) adaptive(Δ εξαρτημένο απο Bits κωδικοποίησης και το range του εκάστοτε παραθύρου)

β) non-adaptive(  $\Delta$  =σταθερό και bits κωδικοποίησης=8)

Παρακάτω φαίνεται ο κώδικας για την παραθυροποίηση του  $Tg$ ( absolute threshold) σε 32 συχνοτικές μπάντες. Αρχικά ορίσαμε την οριακή τιμή συχνότητας για καθένα από τα 32 φίλτρα και βρήκαμε τις κεντρικές συχνότητες( $F_k$ ) για κάθε φίλτρο. Στην συνέχεια φαίνεται η συνάρτηση κβαντοποίησης η οποία δέχεται ουσιαστικά 4 παραμέτρους (σήμα προς κβαντοποίηση, ελάχιστο και μέγιστο πλάτος και αριθμό των επιπέδων) και επιστρέφει το κβαντοποιημένο σήμα.

```
#Step 2.2
M=32
FilterNum=32/32 #number of filters for each filter 1 to M, M=32
print(M)
for k in range(1,M+1): #number of filters for each of 32 filters
    #Mm= 2.1
    #Mm= 2.1

    length=512/32
    Tg_applied=[Tg.length] #applied for Tg to division for 32 filters
    start=length
    stop=length
    for k in range(1,M+1):
        Tg_applied.append(Tg.length)
        start=length
        stop=length

    #print(Tg_applied)
    #print('Tg_applied: ', Tg_applied)
```

```
Quantization function
import numpy as np
from matplotlib import pyplot as plt
import os
from IPython.display import Radio
import sys
sys.path.append('../')
matplotlib.rcParams['font.family'] = 'serif'

def quantize_uniform(x, quant_min, quant_max, quant_level):
    x_normalize = (x - quant_min) / (quant_max - quant_min)
    x_normalize[x_normalize < 0] = 0
    x_normalize[x_normalize > 1] = 1
    x_quantize = np.floor(x_normalize * quant_level)
    x_quant = (x_quantize + 1) * (quant_max - quant_min) / quant_level + quant_min
    return x_quant
```

[illegible]

Μετά υπερδειγματοληπτούμε τα δύο κβαντησμένα σήματα και τα αποθηκεύουμε ως  $ws$  και  $ws2$ . Στην συνέχεια συνελίσσουμε το καθένα από αυτά με τα φίλτρα ανάλυσης  $G_k$ .

```

c1 = 1
c2 = 1
P1 = 0
for k in range(P1 + 1):
    # Bilinear Interpolate
    w1 = (np.interp(rpa.mage(rpa.shape[0], 1 + 100), rpa.mage(100, row), row) for row in pgs)
    w2 = (np.interp(rpa.mage(rpa.shape[0], 100), rpa.mage(100, row), row) for row in pgs)
    # P1 = 0
    c1.append(np.convolve(row, c1) for row in w1)
    c2.append(np.convolve(row, c2) for row in w2)
    P1 += 1

```

```

output = np.sum(x_n,axis=0)
output2 = np.sum(x_n,axis=0)

print(x_n.shape[0])
print(x_n.shape[1])
print(x_n.shape[2])
print(x_n.shape[3])
print(x_n.shape[4])
print(x_n.shape[5])
print(x_n.shape[6])
print(x_n.shape[7])
print(x_n.shape[8])
print(x_n.shape[9])
print(x_n.shape[10])
print(x_n.shape[11])
print(x_n.shape[12])
print(x_n.shape[13])
print(x_n.shape[14])

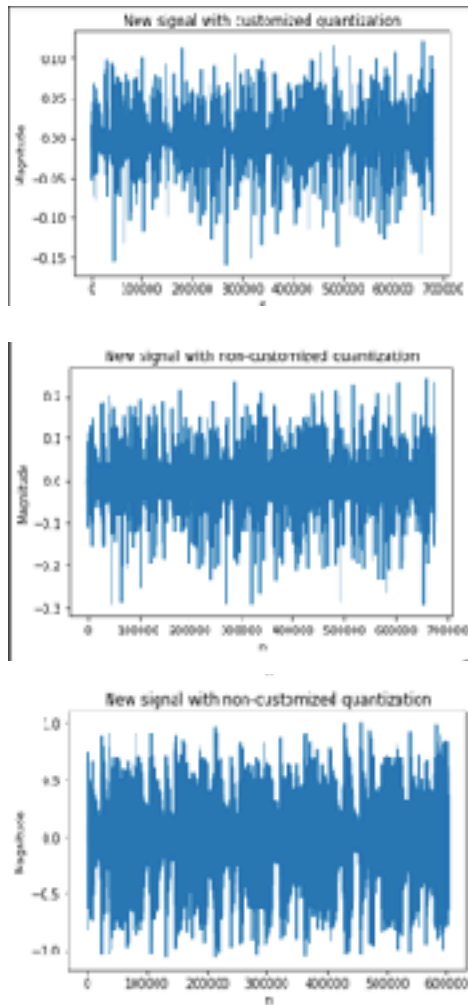
output=output.reshape(1,-1,10)
output2=output2.reshape(1,-1,10)

print(x_n.shape[15])
print(x_n.shape[16])

```



Παρακάτω απεικονίζονται τα δύο σήματα εξόδου και το αρχικό για σύγκριση.



Στο άκουσμα των δύο σημάτων είναι αισθητή η διαφορά με το non-adaptive να προσεγγίζει καλύτερα το αρχικό.