

Kompresija EEG signala

Seminarski rad iz predmeta **Biomedicinski signali i sistemi**

Studenti:

- Nedžad Džindo 1701/17904
- Marko Nedić 1660/17983
- Petar Pejović 1666/17765

Sarajevo, avgust 2020.

Uvod	2
1. O korištenom EEG signalu	3
2. Kvantizacija originalnog signala	4
2.1. Praktična izvedba kvantizacije	4
3. Huffmanovo kodiranje	9
3.1. Teoretska osnova	9
3.2. Praktična izvedba	11
4. Shannon-Fano kodiranje	17
4.1. Teoretska osnova	17
4.2. Praktična izvedba	18
5. Delta modulacija	23
5.1. Teoretska osnova	23
5.2. Praktična izvedba	24
Zaključak	31

Uvod

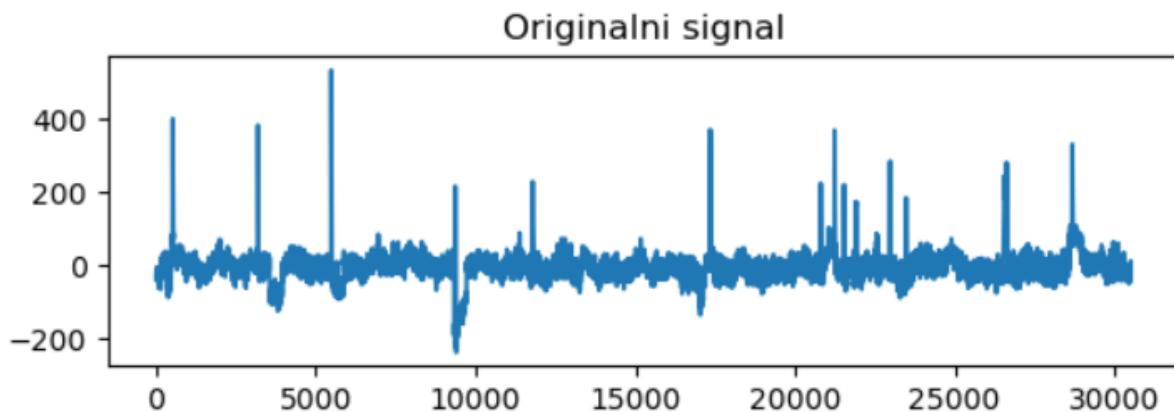
EEG (elektroencefalograf) predstavlja jednu od najkorisnijih, kao i najzastupljenijih tehnika korištenih u različitim oblastima medicine i srodnih nauka. U svojoj osnovi se sastoji od monitoringa (posmatranja) moždane aktivnosti mjerene putem elektroda koje se raspoređuju ravnomjerno na glavu čovjeka. Bitno je napomenuti da je neinvazivna metoda. Pomenuta tehnika se uspješno koristi za detektovanje bolesti nervnog sistema, motoričkih poteškoća, detektovanja posljedica uslijed moždanog udara, bolesti epilepsije i sl., kao i generalno svih poteškoća vezanih za ljudski mozak.

Budući da je tehnika EEG-a jedan od temelja današnje medicine u brojnim sferama liječenja pacijenata, za temu rada je izabran pristup optimiziranja snimanja i same pohrane EEG snimaka. Pošto je poznato da smo digitalnom revolucijom postigli same granice memorije i memorijskih komponenti, te da se ne predviđa enormno poboljšanje u skorijem periodu (prestanak važenja Moore-ovog zakona), memorijska optimizacija i danas predstavlja jednu od aktuelnijih tema u svijetu tehnologije. Optimizacija u vidu komprimiranja podataka (u koju svrstavamo i sadržaj ovog rada) se može svrstati kao jedna od optimizacijskih tehnika koja je prisutna u velikoj mjeri u današnjem svijetu multimedije (kompresija slike, zvuka, teksta, videa, i sl.). Pošto EEG snimci ne predstavljaju ništa drugo nego skupove podataka (datasetove), na njih je moguće primijeti tehnike komprimiranja s ciljem uštede vidno ograničenih resursa. Naravno, same podatke je prvobitno potrebno detaljno analizirati, sagledati sve moguće opcije (algoritme) koji će biti korišteni u razradi, kao i pripremiti same skupove tehnikama normalizacije, transformacije, uklanjanjem nepodobnosti (outliera) i slično.

Možemo zaključiti da je postupak pripreme podataka bitniji od samih tehnika komprimiranja, ali ne smijemo izostaviti ni ulogu algoritama putem kojih će se samo komprimiranje vršiti. Istraživači trebaju biti vođeni osnovnom ulogom algoritama kompresije - sažimanjem podataka, kao i uspješnim vraćanjem podataka u izvorno stanje u što bližem obliku (sa što manje gubitaka). Budući da razlikujemo algoritme sa i bez gubitaka prilikom kompresije, uvijek treba razmatrati njihove dobre i loše karakteristike kao što su (jednostavnost implementacije, stepen kompresije, sfera korištenja, gubici) te na osnovu toga donijeti konačni izbor. U sklopu ovog rada će se pristupiti kompresiji EEG skupa podataka (dataseta) korištenjem dvije poznate tehnike: Shannon-Fano algoritam te Huffmanov algoritam (sa i bez prethodne modifikacije diskretnom kosinusnom transformacijom - DCT).

1. O korištenom EEG signalu

Za testiranje i prikaz rada implementiranih algoritama koji vrše kompresiju EEG signala, korišten je signal koji se nalazi u sklopu EEGLAB alata namjenjen za procesuiranje istih. Ekstraktovali smo prvi signal (jedna od 32 lokacije). Svaki signal se sastoji od 30504 snimljenih vrijednosti u periodu od 238.305 sekundi. Grafički prikaz korištenog signala prikazan je na slici 1.1.



Slika 1.1: Grafički prikaz originalnog EEG signala

Vrijedi napomenuti da su tokom kompresije signala korištene generisane tačke u vremenu. Navedeni signal se generiše sljedećom funkcijom:

```
x = np.linspace(0, len(signali[0]), len(signali[0]))
```

Pri čemu se generiše 30504 vrijednosti u periodu od 0 do 30504. Ovim nije narušen izgled samog signala.

2. Kvantizacija originalnog signala

Budući da su prva dva algoritma koja su obrađena u ovom seminarskom radu Shannon-Fano i Huffmanov algoritam kompresije podataka, prije same kopresije potrebno je izvršiti kvantizaciju podataka. Kvantizaciju je potrebno izvršiti za ova dva algoritma zbog toga što algoritmi rade samo za konačan broj vrijednosti. Iako je originalni signal na ulazu u sistem digitaliziran to znači da već postoji konačan broj vrijednosti u signalu te se spomenuti algoritmi mogu pozvati bez prethodnog kvantiziranja. Kako su digitalne vrijednosti predstavljene float tipom podataka, za pretpostaviti je da su sve vrijednosti u signalu unikatne pa zbog toga pozivanje pomenutih algoritama za kompresiju nad takvim signalom ne bi rezultiralo ukupnom memorijskom kompresijom tj. stepen kompresije bi bio manji od jedan. Bitno je naglasiti da što je manji broj pragova na koje se kvantiziraju originalni digitalni podaci, to je stepen kompresije veći i upravo je to razlog zbog čega je algoritam kvantizacije implementiran.

2.1. Praktična izvedba kvantizacije

Praktično su implementirana dva algoritma za kvantizaciju ulaznog signala.

Oba algoritma kao ulaz primaju dvije varijable:

- signal - ulazni originalni signal opisan nizom vrijednosti predstavljenih float tipom podataka,
- brojPragova - predefinisani cijeli broj koji označava na koliko će se pragova ulazni signal kvantizirati, a opisan je int tipom podataka.

Izlaz algoritma kvantizacije su vrijednosti:

- noviSignal - signal kojemu su sve vrijednosti zaokružene na najbliži prag, a nad takvim signalom se pozivaju Shannon-Fano i Huffman algoritmi kompresije. Opisan je nizom vrijednosti int tipa podataka.
- pragovi - niz int vrijednosti na koje su zaokružene vrijednosti funkcije iz originalnog signala.

Za isti ulaz implementirana dva algoritma kvantizacije dat će isti izlaz, a jedina razlika je u vremenska kompleksnost izvršavanja istih. Prva (jednostavnija) implementacija ima vremensku kompleksnost $O(n^2)$, dok druga (složenija) implementacija je opisana sa vremenskom kompleksnošću $O(n \cdot \log(n))$. Brža verzija je na kraju implementirana i korištena u finalnom programu. Sada slijedi prikaz koda algoritma kvantizacije koji je korišten u programu.

```
def kvantizirajLog(signal, brojPragova):
```

```

noviSignal = copy.deepcopy(signal)
minSig = min(signal)
maxSig = max(signal)
ukupniRaspon = abs(maxSig - minSig)
jedinicniRaspon = ukupniRaspon / brojPragova

pragovi = []
for i in range(brojPragova):
    minPragTmp = minSig + (i * jedinicniRaspon) - (jedinicniRaspon / 2)
    maxPragTmp = minSig + (i * jedinicniRaspon) + (jedinicniRaspon / 2)
    meanPragTmp = int(round((maxPragTmp + minPragTmp) / 2))
    pragovi.append(meanPragTmp)

for i in range(len(noviSignal)):
    rasponIndeksaZaSkokPretrage = int(brojPragova / 4)
    indeksPretrage = int(brojPragova / 2)
    if(noviSignal[i] >= pragovi[-1]):
        noviSignal[i] = int(pragovi[-1])
    while(True):
        # da li je u dobrom opsegu
        if(noviSignal[i] >= pragovi[indeksPretrage - 1]-1 and noviSignal[i] <=
pragovi[indeksPretrage]+1):
            # zaokruzi na manje
            if(noviSignal[i] - pragovi[indeksPretrage - 1] < pragovi[indeksPretrage] -
noviSignal[i]):
                noviSignal[i] = int(pragovi[indeksPretrage - 1])
            # zaokruzi na vece
            else:
                noviSignal[i] = int(pragovi[indeksPretrage])
            break
        # pretrazi za vece pragove
        if(noviSignal[i] > pragovi[indeksPretrage - 1]):
            indeksPretrage = indeksPretrage + rasponIndeksaZaSkokPretrage
        # pretrazi za manje pragove
        else:
            indeksPretrage = indeksPretrage - rasponIndeksaZaSkokPretrage
            rasponIndeksaZaSkokPretrage = int(rasponIndeksaZaSkokPretrage / 2)
            if(rasponIndeksaZaSkokPretrage == 0 and indeksPretrage >= 2):
                rasponIndeksaZaSkokPretrage = rasponIndeksaZaSkokPretrage + 1

return noviSignal, pragovi

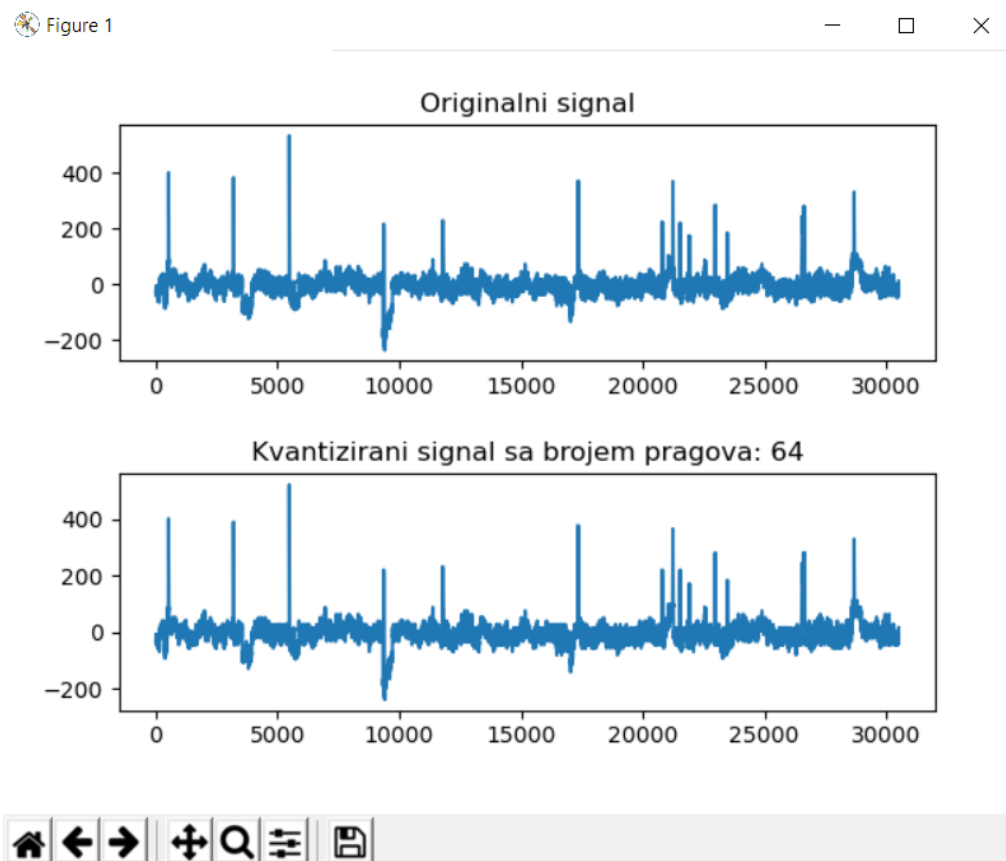
```

Nakon pisanja opisanog koda ostao je još problem određivanja broja pragova na koje će se svesti vrijednosti originalnog signala. Jasno je da vrijedi sljedeće pravilo:

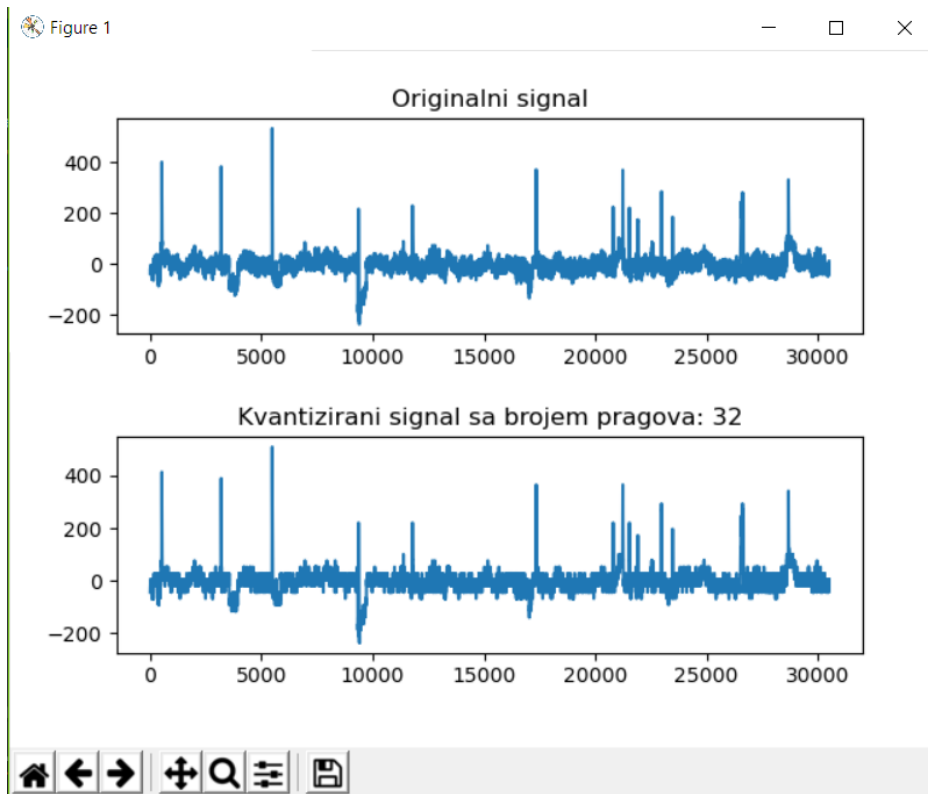
- Što je veći broj pragova to je kvadratna greška odstupanja manja, a stepen kompresije manji.

- Što je manji broj pragova to je kvadratna greška odstupanja veća, a i stepen kompresije veći.

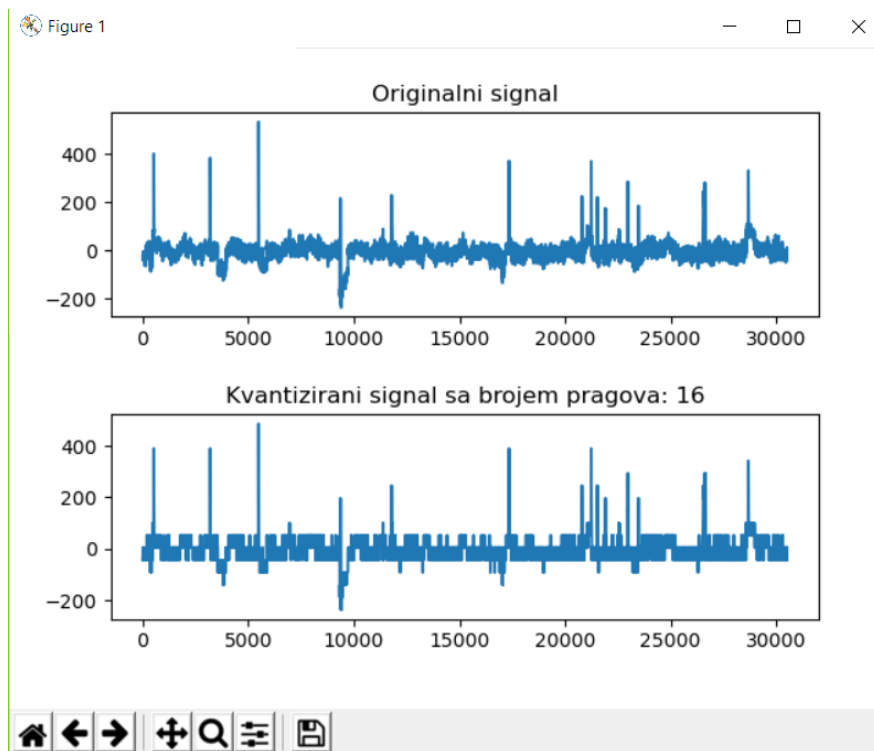
Zbog toga određivanje broja pragova je zaseban problem koji se mora riješiti. Problem je riješen plotanjem kvantiziranog signala za različite vrijednosti broja pragova. U finalnom projektu uzet je broj pragova 32 jer za taj broj kvantizirani signal nije izgubio mnogo na preciznosti, a za ostale manje vrijednosti algoritam "na oko" generira značajnije greške. Sada slijedi tabelarni prikaz spomenutih plotova za vrijednosti broja pragova 64, 32 i 24 respektivno.



Slika 2.1: Kvantizirani signal sa brojem pragova: 64



Slika 2.2: Kvantizirani signal sa brojem pragova: 32



Slika 2.3: Kvantizirani signal sa brojem pragova: 16

Vrijedi još prikazati na koje su to vrijednosti svedeni ulazni podaci ukoliko se radi o opisanom testnom signalu i broju pragova jednakim 32.

Vrijednosti iz ulaznog signala u rasponu:	Vrijednost nakon kvantizacije:
$[-\infty, -224.15]$	-236
$(-224.15, -200.06]$	-212
$(-200.06, -176.98]$	-188
...	...
$(-31.47, -7.38]$	-19
$(-7.38, 16.69]$	5
$(16.69, 40.78]$	29
...	...
$(450.22, 474.30]$	462
$(474.30, 498.39]$	486
$(498.39, +\infty]$	510

3. Huffmanovo kodiranje

Nakon osnovnog opisa signala i njegove kvantizacije, potrebno je opisati i korištene algoritme kodiranja. U sklopu ovog poglavlja i narednog (*Poglavlje 4*) detaljno će se objasniti dva korištena algoritma - Huffmanov i Shannon-Fano. Ukazat ćemo njihove principe, prirodu, sličnosti i razlike na teoretskoj osnovi, te kroz praktične primjere objasniti implementirani sadržaj rada i dobijene rezultate.

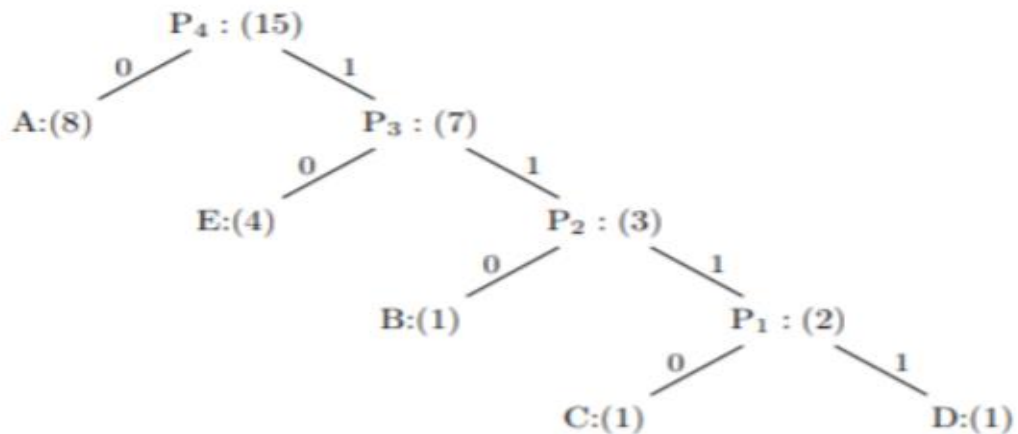
3.1. Teoretska osnova

Huffmanov algoritam kodiranja predstavlja jedan od osnovnih algoritama koji vrše kompresiju po principu od dna prema vrhu (*bottom-up*). Suština algoritma se sastoji u rekurzivnoj izgradnji stabla, a prilikom svake iteracije izgradnje stabla određenim simbolima se dodjeljuju određeni kodni znaci, što ćemo objasniti u sadržaju ovog podpoglavlja. Inače, pomenuti algoritam se većinski koristi u komercijalnim aplikacijama, a i dobro poznatim procesima kompresije - JPEG i MPEG standardima kodiranja.

Sada ćemo opisati sve korake algoritma. Osnovna procedura algoritma je sljedeća:

- Za listu simbola utvrditi frekvencije učestalosti (pojavljivanja)
- Sortirati listu simbola u opadajućem poretku - od najučestalijih prema najmanje učestalnim simbolima
- Rekurzivno ponavljati idući skup akcija - sve dok u listama ne ostane samo po jedan element (do uslova terminiranja algoritma):
 - a) Uzeti dva simbola sa najmanjom frekvencijom i spojiti ih kao djecu jednog roditeljskog čvora.
 - b) Roditeljskom čvoru dodijeliti vrijednost frekvencije jednako sumi frekvencija djece, te takav roditeljski čvor umetnuti u listu bez narušavanja poretka.
 - c) Obrisati čvorove djece iz liste.
- Granama u izgrađenom stablu koje spajaju roditelje i lijeva podstabla dodijeliti 0, a onima koje spajaju roditelje i desna podstabla dodijeliti 1.
- Dodijeliti kodne riječi za svaki list u finalnom stablu. Kodne riječi možemo dobiti tako što pratimo simbole 0 i 1 od korijena ka listu.

Na narednoj slici (3.1) vidimo primjer jednog finalnog stabla kodiranja, te način očitavanja dodijeljene kodne riječi.



Slika 3.1.1: Huffmanovo stablo kodiranja

Primjer očitavanja kodne riječi za npr. slovo 'C' bi bilo: 1110.

Potrebno je naglasiti još dvije osobine Huffmanovog kodiranja koje ga čine pogodnim za kompresijske algoritme. To su svojstva jedinstvenog prefiksa i svojstvo optimalnosti.

- Svojstvo jedinstvenog prefiksa garantuje da nijedna riječ dobijena Huffmanovim kodiranjem nije prefiks niti jedne druge riječi. Ta osobina direktno proizilazi da se pri izgradnji stabla svi simboli nalaze u listovima. Pomenuta osobina važiti će i za Shannon-Fano algoritam (*Poglavlje 4*).
- Svojstvo optimalnosti garantuje da Huffmanov algoritam generira kodove s minimalnom redundancijom. Detaljnije to možemo razgraničiti na naredni par tvrdnji:
 - a) Kodne riječi s najmanjom frekvencijom pojavljivanja razlikuju se samo po posljednjem bitu.
 - b) Simboli koji imaju veće frekvencije pojavljivanja istovremeno imaju i kraće kodne riječi.

Osim klasičnog Huffmanovog kodiranja, prethodno opisanog u sklopu ovog poglavlja, bitno je za napomenuti da postoji i adaptivno Huffmanovo kodiranje. Osnovna razlika klasičnog i adaptivnog kodiranja je ta da se u slučaju adaptivnog kodiranja ne znaju unaprijed svi ulazni znakovi, te je potrebno razviti strategije manipulacija nad strukturom stabla kodnih simbola. Manipulacije obuhvataju rotiranje stabla i zamjenu simbola, sve uz očuvanje osnovne osobine stabla (poretka). Iako je pomenuta tema poprilično korisna, u sklopu ovog rada nisu korištene adaptivne tehnike tako da u njen detaljniji način rada nećemo ulaziti.

Pomenut ćemo još i izraz za stepen kompresije. On je osnovna karakteristika kompresijskih algoritama i predstavlja mjeru kvalitete samog algoritma. Računa se po formuli: $B1/B2$ (gdje $B1$ predstavlja broj bita prije kompresije, a $B2$ broj bita nakon kompresije).

3.2. Praktična izvedba

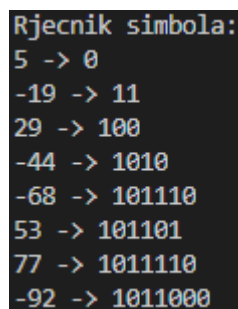
Implementacija huffmanog algoritma za rad sa EEG signalom sastoji se od tri ključne funkcije, a to su kodiraj(), dekodiraj() i izvjestaj(). Nakon što se uspješno izvrši kvantizacija ulaznog EEG signala, ovaj signal se dodjeljuje varijabli ulazniSignal, te se proslijeđuje funkciji kodiraj():

```
kodiraniSignal = kodiraj(ulazniSignal)
```

Funkcija **kodiraj()** generiše binarni kod (za ulazni EEG signal) koji funkcija vraća. Prvo se kreira rječnik gdje su vrijednosti poredane po frekvenciji pojavljivanja od najveće do najmanje. Nakon toga ove vrijednosti se dodjeljuju u red pomoću kojeg se generiše stablo sa svim vrijednostima. Generisano stablo se nakon toga koristi za kodiranje vrijednosti tako što se pronalazi svaki list(vrijednost) i dodjeljuje im se binarni kod. Ovaj dio implementiran je pomoću funkcije kodirajVrijednostiStabla() kojoj se proslijeđuje stablo koje se nalazi na stacku i prazan string. Implementacija funkcije kodirajVrijednostiStabla() prikazana je u sljedećem kodu:

```
def kodirajVrijednostiStabla(trenutni_cvor, dekompresovani_string):  
    if trenutni_cvor is None:  
        return  
    if trenutni_cvor.lijevi is None and trenutni_cvor.desni is None:  
        rjecnikStabla[trenutni_cvor.karakter] = dekompresovani_string  
        return  
    kodirajVrijednostiStabla(trenutni_cvor.lijevi, dekompresovani_string+"0")  
    kodirajVrijednostiStabla(trenutni_cvor.desni, dekompresovani_string+"1")
```

Primjer generisanog rječnika prikazan je na slici ispod:



```
Rjecnik simbola:  
5 -> 0  
-19 -> 11  
29 -> 100  
-44 -> 1010  
-68 -> 101110  
53 -> 101101  
77 -> 1011110  
-92 -> 1011000
```

Slika 3.2.1. Rječnik simbola

Nakon što se uspješno generiše rječnik, kodira se signal tako što se svaka vrijednost signala zamijeni sa njenim odgovarajućim binarnim kodom i ovaj rezultat se vraća iz funkcije. Kompletna implementacija funkcije kodiraj() prikazana je u kodu ispod:

```
def kodiraj(signal):  
    frekvencija=Counter(signal).most_common()  
    Qsize = len(frekvencija)  
    Q = PriorityQueue()  
  
    print("Broj pojavljivanja vrijednosti:")
```

```

print(frekvencija)
for key,value in frekvencija:
    element = cvor(key, value)
    Q.put(element)

#Kreiranje stabla
while(Qsize!= 1):
    novicvor = cvor()
    lijevi = Q.get()
    desni = Q.get()
    novicvor.lijevi = lijevi
    novicvor.desni = desni
    novicvor.frekvencija = lijevi.frekvencija + desni.frekvencija
    Q.put(novicvor)
    Qsize-=1

#Pretvaranje stabla u rjecnik
for i in range (0,len(frekvencija)):
    dekompresovani_karakter = {frekvencija[i][0]:0}
    rjecnikStabla.update(dekompresovani_karakter)
    kodirajVrijednostiStabla(Q.get(), '')

kodiraniSignal = ''
for karakter in signal:
    kodiraniSignal += rjecnikStabla[karakter]

return kodiraniSignal

```

Nakon završenog kodiranja, kodirani signal je proslijeđen funkciji dekodiraj(). Ova funkcija kao rezultat vraća dekodirani signal što je u biti ulazni/izvorni signal. Proces dekodiranja kodiranog signala je relativno jednostavan i sastoji se od sljedećih koraka:

1. Generiši novi rječnik stabla sa zamijenjenim vrijednostima ključa i vrijednosti (umjesto npr. 45-> '00110', u ovom stablu će biti '00110'->45)
2. Sve dok je dužina kodiranog signala veća od nule, uzmi novi simbol i dodjeli ga stringu, provjeri da li string postoji u invertovanom rječniku
3. Ako postoji, string obriši iz kodiranog signala, vrijednost iz rječnika za taj binarni kod dodjeli u niz za dekodirani signal i vrati se na korak 2
4. Ako ne postoji, pređi na sljedeći simbol i vrati se na korak 2

Kod za funkciju dekodiraj je prikazan ispod:

```
def dekodiraj(kodiraniSignal):
    inverzniRijecnik = dict([(value, key) for key,value in rjecnikStabla.items()])

    zaDekodiranje = kodiraniSignal

    dekodiraniSignal=[]
    trenutniSimboli = ""
    indexZadnjeg = 0
    while len(zaDekodiranje)>0:
        trenutniSimboli = trenutniSimboli + zaDekodiranje[indexZadnjeg]
        if(trenutniSimboli in inverzniRijecnik):
            dekodiraniSignal.append(inverzniRijecnik.get(trenutniSimboli))
            zaDekodiranje=zaDekodiranje[indexZadnjeg+1:]
            indexZadnjeg = 0
            trenutniSimboli=""
        else :
            indexZadnjeg =indexZadnjeg +1

    return dekodiraniSignal
```

Nakon dekodiranja, generišu se rezultati pomoću funkcije izvjestaj(). U sklopu izvještaja prikazuju se sljedeći podaci:

1. Rječnik simbola (u formatu 'Vrijednost' -> 'binarna vrijednost'),
2. Broj bita kodiranog signala,
3. Stepen kompresije,
4. Ulazni signal,
5. Kodirani signal (koji se zbog veličine upisuje u datoteku kodiraniSignal.txt),
6. Grafički prikaz ulaznog i dekompresovanog signala te
7. Provjeru da li su ulazni i dekodirani signal identični.

Treba napomenuti da je prije generisanja izvještaja, u funkciji kodiraj() ispisuje niz sa frekvencijama (brojem pojavljivanja) za svaku vrijednosti signala. Implementacija funkcije izvjestaj() prikazana je sljedećem kodu:

```
def izvjestaj():

    print('Rjecnik simbola:')
    for karakter in rjecnikStabla:
        print(karakter,'->',rjecnikStabla[karakter])

    duzinaKodiranogSignala = len(kodiraniSignal)
    print("Broj bita kodiranog signala: ", duzinaKodiranogSignala)
```

```

    #Pošto se za ascii reprezentaciju uzima 8bita tako da ćemo stepen računati
na sljedeći način
    stepen_kompresije = 32*len(ulazniSignal)/duzinaKodiranogSignala
    print('Stepen kompresije:', stepen_kompresije)

    print('Ulazni signal:', ulazniSignal)
    datoteka = open('kodiraniSignal.txt','a+')
    datoteka.write(kodiraniSignal)
    datoteka.close()
    print('Kodirani signal je upisan u datoteku kodiraniSignal.txt')

    if(ulazniSignal == dekodiraniSignal).all():
        print('Ulazni i dekodirani signali su identicni.')

    fig, pt = plt.subplots(2)
    fig.tight_layout(pad=3.0)

    fig.suptitle('Huffman')
    pt[0].plot(x, ulazniSignal)
    pt[0].set_title('Ulazni signal')

    pt[1].plot(x, dekodiraniSignal)
    pt[1].set_title('Dekodirani signal: ')

    plt.show()

```

```

Broj pojavljivanja vrijednosti:
[(5, 11114), (-19, 10122), (29, 3847), (-44, 2660), (-68, 658), (53, 643), (77, 437), (-92, 286), (-140, 130), (101, 110),
4), (173, 38), (197, 27), (-164, 27), (149, 25), (270, 18), (-212, 18), (366, 13), (294, 12), (-236, 8), (318, 7), (510, 4)]
Rjecnik simbola:
5 -> 0
-19 -> 11
29 -> 100
-44 -> 1010
-68 -> 101110
53 -> 101101
77 -> 1011110
-92 -> 1011000
-140 -> 10110010
101 -> 101111110
-116 -> 101100110
221 -> 1011111110
246 -> 1011111001
-188 -> 1011111000
125 -> 1011001111
173 -> 1011001110
197 -> 10111110110
-164 -> 10111110111
149 -> 10111110101
270 -> 101111111110
-212 -> 101111111111
366 -> 101111111100
294 -> 101111101000
-236 -> 1011111111010
318 -> 1011111010011
510 -> 10111111110110
342 -> 10111111110111
390 -> 10111110100100
414 -> 101111101001011
438 -> 101111101001010
Broj bita kodiranog signala: 73629
Stepen kompresije: 13.257384997759035
Ulazni signal: [-44 -19 -19 ... 5 5 5]
Kodirani signal je upisan u datoteku kodiraniSignal.txt
Ulazni i dekodirani signali su identicni.

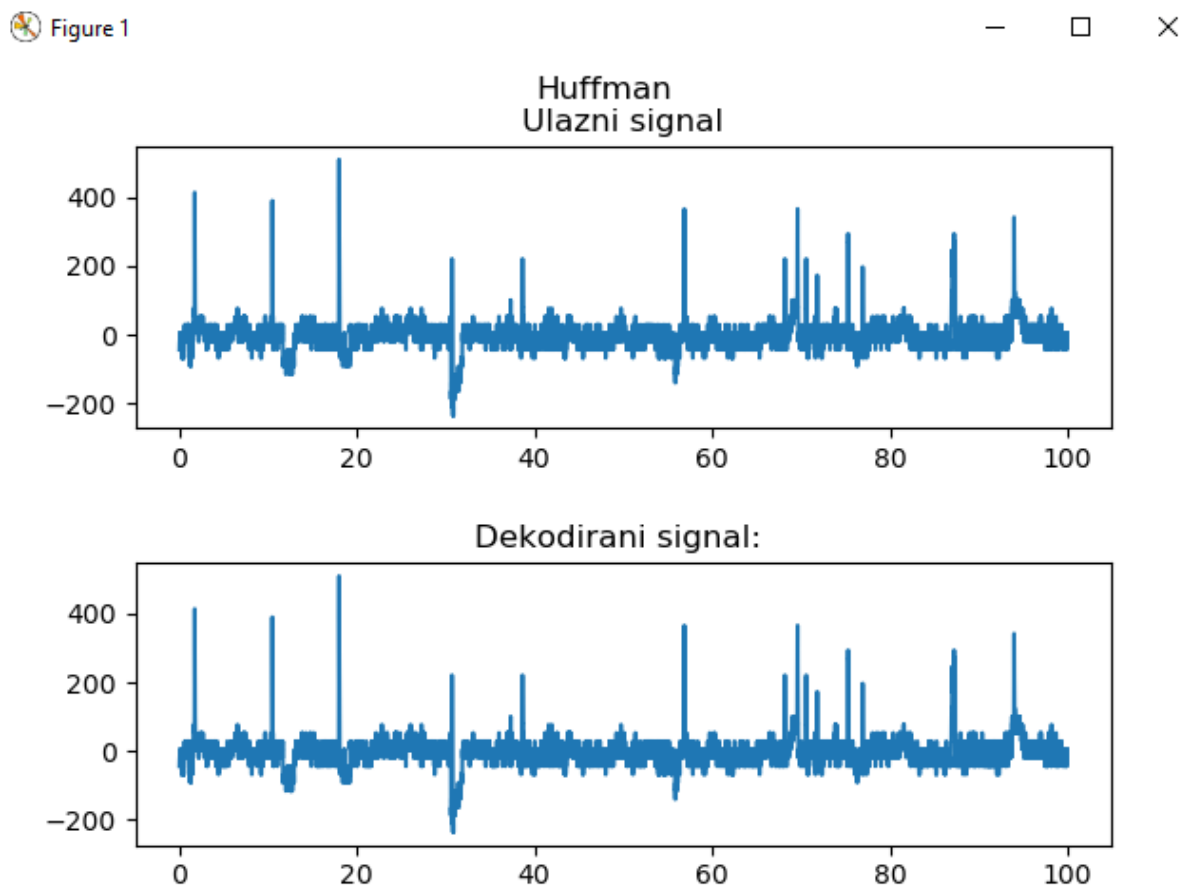
```

Također ne smijemo zaboraviti ulazni signal koji se upisuje u tekstualnu datoteku kodiraniSignal.txt:

[illegible]

15

Grafički prikaz ulaznog i dekodiranog signala prikazan je na slici ispod:



Slika 3.2.4. Grafički prikaz ulaznog i dekodiranog signala

Zbog veličine kodiranog signala prikazan je samo jedan dio. Stepen kompresije se računa kao:

$$\begin{aligned} \text{stepen kompresije} &= \frac{\text{veličina ulaznog signala} * 4B}{\text{broj bita kodiranog signala}} = \frac{976128}{73629} \\ &= 13.257384997759035 \end{aligned}$$

Što znači da smo uspjeli smanjiti veličinu ulaznog signala približno 13.2578 puta.

4. Shannon-Fano kodiranje

Shannon-Fano kodiranje je algoritam u osnovi sličan prethodno opisanom Huffmanovom, koji se također vodi karakteristikom da frekventnijim simbolima treba dodijeliti kraće kodne riječi. Za razliku od Huffmanovog, Shannon-Fano je baziran na principu top-down, gdje se stablo gradi od vrha prema dnu. Sada ćemo dati teoretske osnove za pomenuti algoritam, kao i implementacijski primjer i analizu rezultata.

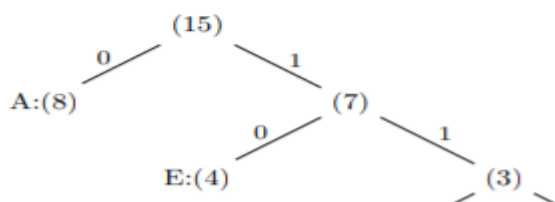
4.1. Teoretska osnova

Algoritam možemo najbolje opisati kroz sljedeći niz koraka:

- Za listu simbola utvrditi frekvencije učestalosti (pojavljivanja)
- Sortirati listu simbola u opadajućem poretku - od najučestalijih prema najmanje učestalnim simbolima
- Podijeliti listu na dva dijela, tako da frekvencije pojavljivanja elemenata u oba dijela trebaju biti što je moguće sličnije
- Lijevi dio liste dobija kodni simbol 0, a desni kodni simbol 1
- Rekurzivno ponavljati prethodna dva koraka, sve dok svi dobijeni dijelovi liste ne budu sadržavali samo jedan element.

Prilikom implementacije Shannon-Fano algoritma također se gradi binarno stablo. Konvencijom je utvrđeno da se svim granama koje povezuju roditeljski čvor i lijevo dijete pridružuje znak 0, a onima koji povezuju roditeljski čvor i desno dijete pridružuje broj 1. Dobijene kodne riječi nisu uvijek jednoznačne. To je čista posljedica koraka 3 iz opisa algoritma. Nekada je moguće elemente unutar skupina pregrupisati na drugačiji način što će na kraju rezultovati i drugačijim finalnim kodnim riječima. Shannon-Fano također zadovoljava osobine jedinstvenog prefiksa i minimalne redundancije.

U nastavku slijedi dio stabla kodiranja i očitavanje kodne riječi za simbol iz primjera (E).



Slika 4.1: Shannon-Fano kodiranje

Primjer očitavanja za 'E' je: 10 (kao i za Huffmana). Stepen kompresije se računa na isti način.

4.2 Praktična izvedba

Slično kao kod Huffman algoritma kompresije EEG signala, i kod ovog signala se nakon kvantizacije signala kreira lista sa jedinstvenim vrijednostima iz signala. Nakon kreiranja liste jedinstvenih vrijednosti, kreira se i tuple u formatu ('Vrijednost', ''), gdje se u drugo polje upisuje binarni kod za svaki signal. Ažuriranje binarnog koda za svaki signal se obavlja putem funkcije Shannon_Fano_Pomocna() kojoj se proslijeđuje index donjeg i gornjeg dijela liste, kao i kompletan tuple (varijabla lista). Implementacija ove pomoćne funkcije prikazana je kodom ispod:

```
def Shannon_Fano_Pomocna(donjiDio, gornjiDio, lista):

    size = gornjiDio - donjiDio + 1
    if size > 1:
        mid = int(size / 2 + donjiDio)
        for i in range(donjiDio, gornjiDio + 1):
            tup = lista[i]
            if i < mid:
                lista[i] = (tup[0], tup[1] + '0')
            else:
                lista[i] = (tup[0], tup[1] + '1')
        Shannon_Fano_Pomocna(donjiDio, mid - 1, lista)
        Shannon_Fano_Pomocna(mid, gornjiDio, lista)
```

Nakon završetka ove funkcije svakoj vrijednosti u signalu je dodijeljen binarni kod:

```
lista:
[(510.0, '0000'), (438.0, '00010'), (414.0, '00011'), (390.0, '00100'), (366.0, '00101'), (342.0, '00110'), (318.0, '00111'),
(149.0, '01110'), (125.0, '01111'), (101.0, '1000'), (77.0, '10010'), (53.0, '10011'), (29.0, '10100'), (5.0, '10101'),
), (-164.0, '11100'), (-188.0, '11101'), (-212.0, '11110'), (-236.0, '11111')]
```

Slika 4.2.1. Izgled liste

Ova lista se radi lakše manipulacije konvertuje u rječnik, te nakon konvertovanja vrši se operacija kodiranja ulaznog signala. Cijeli proces dobivanja binarnog koda za svaku vrijednost kao i kodiranje signala implementirano je funkcijom Shannon_Fano koja kao ulazni argument prima izvorni signal:

```
def Shannon_Fano(signal):
    znakovi = ctr(signal)
    lista = sorted([(b, '') for b,a in znakovi.items()], key=itemgetter(0), reverse=True)
    Shannon_Fano_Pomocna(0, len(lista)-1, lista)
    rjecnikKodovaFun = dict((key, value) for key,value in lista)

    kodiraniSignal = ''
    for karakter in signal:
        kodiraniSignal += rjecnikKodovaFun[karakter]

    #print(kodiraniSignal)
```

```
return kodiraniSignal, rjecnikKodovaFun
```

Operacija dekodiranja je identična kao kod Huffmanovog algoritma, više informacija o tome kako funkcioniše operacija dekodiranja kao i kod može se pronaći u tehničkoj implementaciji u sklopu poglavlja "Huffmanovo kodiranje".

Prilikom generisanja izvještaja u konzoli se prikazuju sljedeće informacije:

1. Rječnik vrijednosti (U formatu: "Vrijednost: " <Vrijednost>", Kod:" <Binarni kod>),
2. Broj bita za prikaz nekodiranog (ulaznog) signala,
3. Broj bita za prikaz kodiranog signala,
4. Stepen kompresije,
5. Provjera da li su ulazni i dekodirani signali identični,
6. Ulazni signal i
7. Poruka o tome da je kodirani signal upisan u tekstualnu datoteku.

Kod funkcije izvjestaj(), koja generiše sam izvještaj, prikazan je ispod:

```
def izvjestaj():

    brojBitaOriginalnogSignala = len(ulazniSignal)*32
    brojBitaKodiranogSignala = len(kodiraniSignal)
    znakovi = ctr(ulazniSignal)
    lista_ponavljanja = [a for b,a in znakovi.items()]

    print('Rjecnik vrijednosti:')
    for karakter in rjecnikKodova:
        print("Vrijednost:" + str(karakter) + ", Kod: " + rjecnikKodova[karakter])

    print(" ")
    print("Broj bita potreban za prikaz nekodiranog signala: " + str(brojBitaOriginalnogSignala))
    print("Broj bita potreban za prikaz kodiranog signala: " + str(brojBitaKodiranogSignala))
    print("Stepen kompresije iznosi: " + str(brojBitaOriginalnogSignala/brojBitaKodiranogSignala))

    if(ulazniSignal == dekodiraniSignal).all():
        print("Ulazni i dekodirani signal su identicni.")
    else:
        print("Ulazni i dekodirani signal nisu identicni.")

    print('Ulazni signal:', ulazniSignal)
    datoteka = open('kodiraniSignalShannonFano.txt','a+')
    datoteka.write(kodiraniSignal)
    datoteka.close()
    print('Kodirani signal je upisan u datoteku kodiraniSignalShannonFano.txt')
)
```

```

fig, pt = plt.subplots(2)
fig.tight_layout(pad=3.0)

fig.suptitle('Shannon-Fano kompresija')
pt[0].plot(x, ulazniSignal)
pt[0].set_title('Ulazni signal')

pt[1].plot(x, dekodiraniSignal)
pt[1].set_title('Dekodirani signal: ')

plt.show()

```

Prije samog ispisivanja sadržaja proračunavaju se količine bita sa spašavanjem ulaznog i kodiranog signala. Za čuvanje jedne vrijednosti ulaznog signala potrebna su 4 bajta, tako da se broj bita za originalni signal računa kao:

$$\begin{aligned}
 \text{broj bita za originalni signal} &= \text{veličina originalnog signala} * 4B \\
 &= \text{veličina originalnog signala} * 32 \text{ bita}
 \end{aligned}$$

Potom, stepen kompresije se računa kao:

$$\text{stepen kompresije} = \frac{\text{broj bita za originalni signal}}{\text{broj bita za kodirani signal}} = \frac{976128}{152406} = 6.404787213101846$$

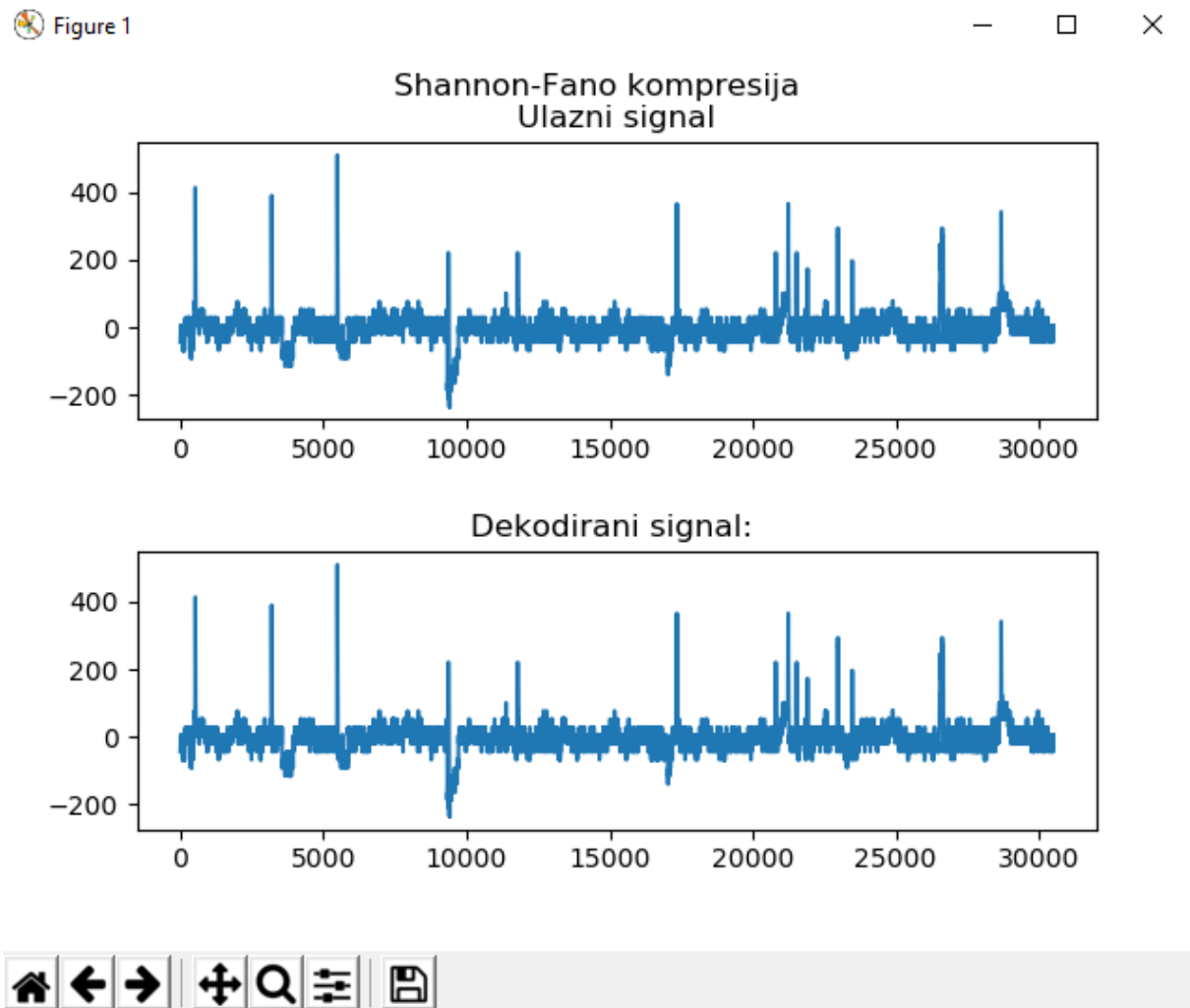
Pored poruka koje se ispišu u konzoli, izvještaj generiše i tekstualnu datoteku kodiraniSignalShannonFano.txt, u kojoj se nalazi kodirani signal:



The image shows a text editor window with the file name 'kodiraniSignalShannonFano.txt'. The content of the file is a single line of binary code, starting with a line number '1' in the left margin. The binary sequence is long and appears to be a compressed representation of the original signal.

Slika 4.2.2. Dio kodiranog signala Shannon-Fano algoritmom

Također prikazan je i plot ulaznog i dekodiranog signala koji su isti jer se radi o algoritmu bez gubitaka:



Slika 4.2.3. Plot ulaznog i dekodiranog signala

Rječnik vrijednosti a koji se ispisiuje u konzoli, prikazan je na sljedećoj slici:

```
Rjecnik vrijednosti:
Vrijednost:510.0, Kod: 0000
Vrijednost:438.0, Kod: 00010
Vrijednost:414.0, Kod: 00011
Vrijednost:390.0, Kod: 00100
Vrijednost:366.0, Kod: 00101
Vrijednost:342.0, Kod: 00110
Vrijednost:318.0, Kod: 00111
Vrijednost:294.0, Kod: 01000
Vrijednost:270.0, Kod: 01001
Vrijednost:246.0, Kod: 01010
Vrijednost:221.0, Kod: 01011
Vrijednost:197.0, Kod: 01100
Vrijednost:173.0, Kod: 01101
Vrijednost:149.0, Kod: 01110
Vrijednost:125.0, Kod: 01111
Vrijednost:101.0, Kod: 1000
Vrijednost:77.0, Kod: 10010
Vrijednost:53.0, Kod: 10011
Vrijednost:29.0, Kod: 10100
Vrijednost:5.0, Kod: 10101
Vrijednost:-19.0, Kod: 10110
Vrijednost:-44.0, Kod: 10111
Vrijednost:-68.0, Kod: 11000
Vrijednost:-92.0, Kod: 11001
Vrijednost:-116.0, Kod: 11010
Vrijednost:-140.0, Kod: 11011
Vrijednost:-164.0, Kod: 11100
Vrijednost:-188.0, Kod: 11101
Vrijednost:-212.0, Kod: 11110
Vrijednost:-236.0, Kod: 11111
```

Slika 4.2.4. Rječnik vrijednosti

Ostatak izvještaja kao što su broj bita potrebnih da se sačuva ulazni i kodirani signal, stepen kompresije, itd. prikazan je na slici ispod:

```
Broj bita potreban za prikaz nekodiranog signala: 976128
Broj bita potreban za prikaz kodiranog signala: 152406
Stepen kompresije iznosi: 6.404787213101846
Ulazni i dekodirani signal su identicni.
Ulazni signal: [-44. -19. -19. ... 5. 5. 5.]
Kodirani signal je upisan u datoteku kodiraniSignalShannonFano.txt
```

Slika 4.2.5. Ostatak generisanog sadržaja

5. Delta modulacija

Prethodno opisane impulsno kodne modulacije mogu se i dodatno unaprijediti koristeći alternativne PCM tehnike od kojih je najpoznatija tehnika *delta modulacije*. Delta modulacija je algoritam kompresije sa gubicima. Za potrebe delta modulacije nije potrebno kvantizirati ulazne podatke signala, što je jedna od prednosti ovog algoritma vidljiva u vremenu izvršavanja istog.

5.1. Teoretska osnova

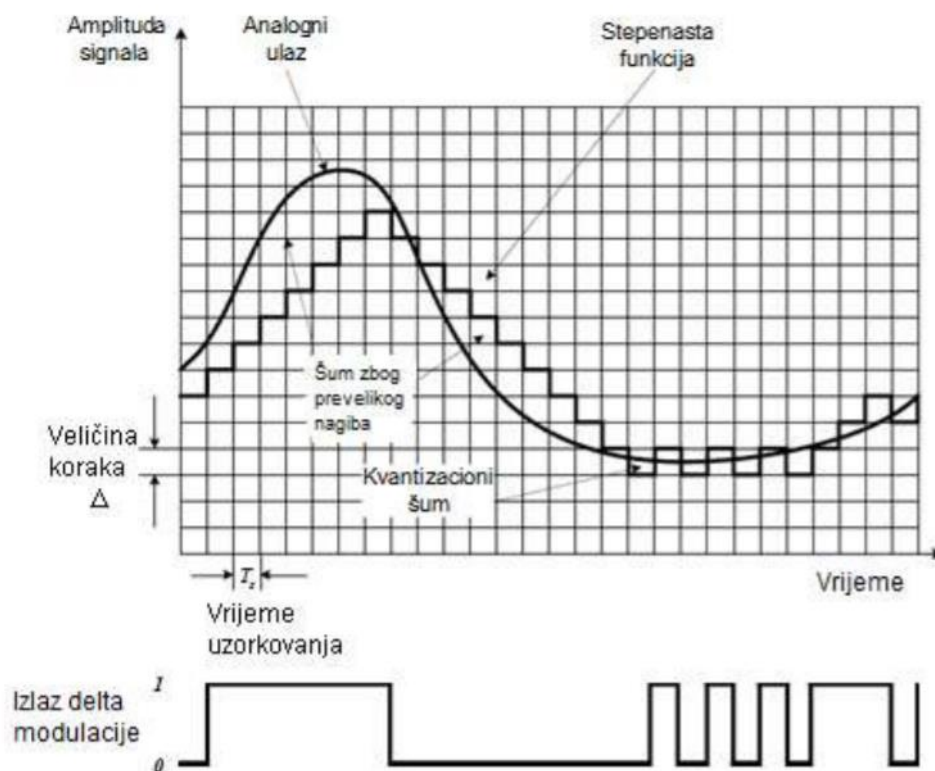
Kod delta modulacije se ulazni digitalni signal aproksimira stepeničastom funkcijom karakterističnog izgleda. Stepeničasti signal, koji je ustvari komprimirani oblik ulaznog signala, se pomjera ili naviše ili naniže za svaku novu vrijednost u originalnom signalu.

Delta modulator generira niz bita dužine ulaznog signala minus jedan. Biti u tom nizu predstavljaju da li će se sljedeća vrijednost komprimiranog signala pomjeriti naviše (označeno sa bitom 1), ili naniže (označeno sa bitom 0). Veličina tog pomjeranja određena je koeficijentom *delta* (Δ).

Najpogodniji koeficijent delta se najčešće nalazi ekperimentalnim putem za date oblike signala. U ovisnosti o delte, komprimirani signal će biti bolji ili lošiji, ali ne postoji deterministički način za otkrivanje najpogodnije delte. U objašnjenju praktične izvedbe ovog algoritma bit će objašnjeno na koji način je dobijena pogodna delta. Na slici 5.1.1. se može primijetiti da se veće greške stvaraju kada se originalni signal brzo mijenja. Uzrok tome je koeficijent delta koji je mal da "stigne" taj brzi rast originalne funkcije. Ukoliko bi koeficijent delta bio postavljen na neki veći broj, takve greške bi postale manje ali bi se javile kvantizacijske greške jer delta ne bi bila precizna.

Glavne značajke delta modulacije su:

- Digitalni signal nije potrebno kvantitizirati.
- Stepeničasti pragovi se kreiraju za svaki novi digitalni uzorak pa se ne može dogoditi Nyquist-ova pogreška.
- Visok stepen kompresije kod signala koji izgledaju kao šum i nemaju vrlo nagle skokove.
- Greška varira u odnosu na vrijednost delte.



Slika 5.1.1: Princip rada delta modulacije

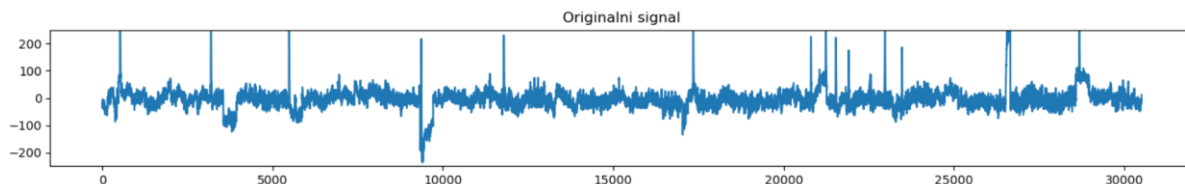
Na slici 5.1.1. može se vidjeti:

- Analogni signal (u slučaju obrađenog EEG dataseta na kojim su vršene algoritmi kompresije, ovaj signal je digitalan),
- Stepeničasti delta signal (komprimirani oblik ulaznog signala)
- Veličina koraka *delta* (Δ),
- Šum zbog prevelikog nagiba koji nastaje zbog toga što je delta premala da “stigne” strmi nagib ulaznog signala,
- Kvantizacioni šum koji nastaje zbog toga što je delta prevelika i komprimirani signal titra oko originalnog signala.

5.2. Praktična izvedba

Kao što je već rečeno, algoritam delta modulacija relativno je dobar algoritam kompresije za signale koji izgledaju kao šum i nemaju vrlo nagle skokove. Upravo takva je većina EEG signala i zbog toga je za pretpostaviti da će ovaj algoritam dati relativno dobre rezultate kompresije.

Kao što je i u prethodno opisanim algoritmima kompresije i ovaj algoritam će praktično biti prikazan na istom signalu (slika 5.1.1.). Kod učitavanja ovog signala u varijablu *signal* nije potrebno opet prikazivati.



Slika 5.2.1: Prikaz originalnog signala na kojem je vršena kompresija

Nakon učitano signala u *main-u* se poziva funkcija *deltaModulacija*, kojoj se kao parametar šalje prethodno prikazani učitani signal.

```
pocetnaVrijednost, delta, nizBita = deltaModulacija(signal)
```

Kod pozvane funkcije slijedi u nastavku. Funkcija kao rezultat vraća tri varijable:

- *pocetnaVrijednost* - prva digitalna vrijednost zaokružena na najbliži cijeli broj
- *delta* - iako je delta globalna varijabla o kojoj će posebno biti riječi u nastavku, ipak se vraća kao rezultat iz funkcije jer se delta može eksperimentalno računati u ovom dijelu koda
- *nizBita* - ova varijabla je srž kompresije a predstavlja “izlaz delta modulacije” na slici 5.1.1.

```
def deltaModulacija(signal):
    pocetnaVrijednost = int(round(signal[0]))
    nizBita = bytearray()
    deltaSignal = [pocetnaVrijednost]

    for i in range(len(signal) - 1):
        if(signal[i+1] > deltaSignal[i]):
            deltaSignal.append(deltaSignal[i] + delta)
            nizBita.append(1)
        else:
            deltaSignal.append(deltaSignal[i] - delta)
            nizBita.append(0)

    return pocetnaVrijednost, delta, nizBita
```

Na osnovu dobivene tri vrijednosti moguće je iskonstruirati stepeničasti signal koji ustvari predstavlja kompresiju originalnog signala sa gubitkom. Ove tri vrijednosti predstavljaju komprimirani oblik koji je potrebno poslati sa jednog uređaja na drugi. Na drugoj strani sistema (npr. uređaj koji treba prikazati EEG signal) može se pozvati funkcija na sljedeći način. Funkcija kao parametre prima prethodno spomenute tri varijable.

```
noviSignal = inverznaDeltaModulacija(pocetnaVrijednost, delta, nizBita)
```

Kod pozvane funkcije slijedi u nastavku. Funkcija kao rezultat vraća jednu varijablu pod nazivom *noviSignal*. To je varijabla koja predstavlja stepeničastu funkciju tj. signal.

```
def inverznaDeltaModulacija(pocetnaVrijednost, delta, nizBita):
    noviSignal = [pocetnaVrijednost]

    for i in range(len(nizBita)):
        if(nizBita[i] == True):
            noviSignal.append(noviSignal[i] + delta)
        else:
            noviSignal.append(noviSignal[i] - delta)

    return noviSignal
```

Dosadašnji dio opisanog programa predstavlja srž ovog algoritma, ali bitno je skrenuti pažnju na nekoliko bitnih stavki. U programu varijabla *delta* je korištena kao globalna varijabla sa vrijednosti dvanaest (12). Postavlja se pitanje, zašto baš taj broj. Kao što je već rečeno u poglavlju koji se bavi teoretskim opisom delta modulacije, pomenuti koeficijent se nalazi eksperimentalnim putem. Taj način se svodi na pokretanje programa sa različitim vrijednostima delte sve dok se ne pronađe delta koja najbolje (ili gotovo najbolje) odgovara za kompresiju početnog signala.

Naravno da postoje i egzaktno metode kojima se može pronaći delta za koju kompresovani signal ima npr. najmanju kvadratnu grešku ili najmanju apsolutnu grešku itd. Ali specifičnost EEG signala je to što razlika u vrijednosti uzoraka sa istim rednim brojem kod originalnog i komprimiranog signala ne igra veliku ulogu. Glavnu ulogu u kvaliteti koprimiranog signala igra sličnost oblika ta dva signala u pojedinim dijelovima koji se promatraju.

Kod funkcije koja pokreće opisani program za deltu u rasponu [0, 30] slijedi u nastavku. Iako delta ne funkcije ako ima vrijednost nula, prikazan je i taj slučaj.

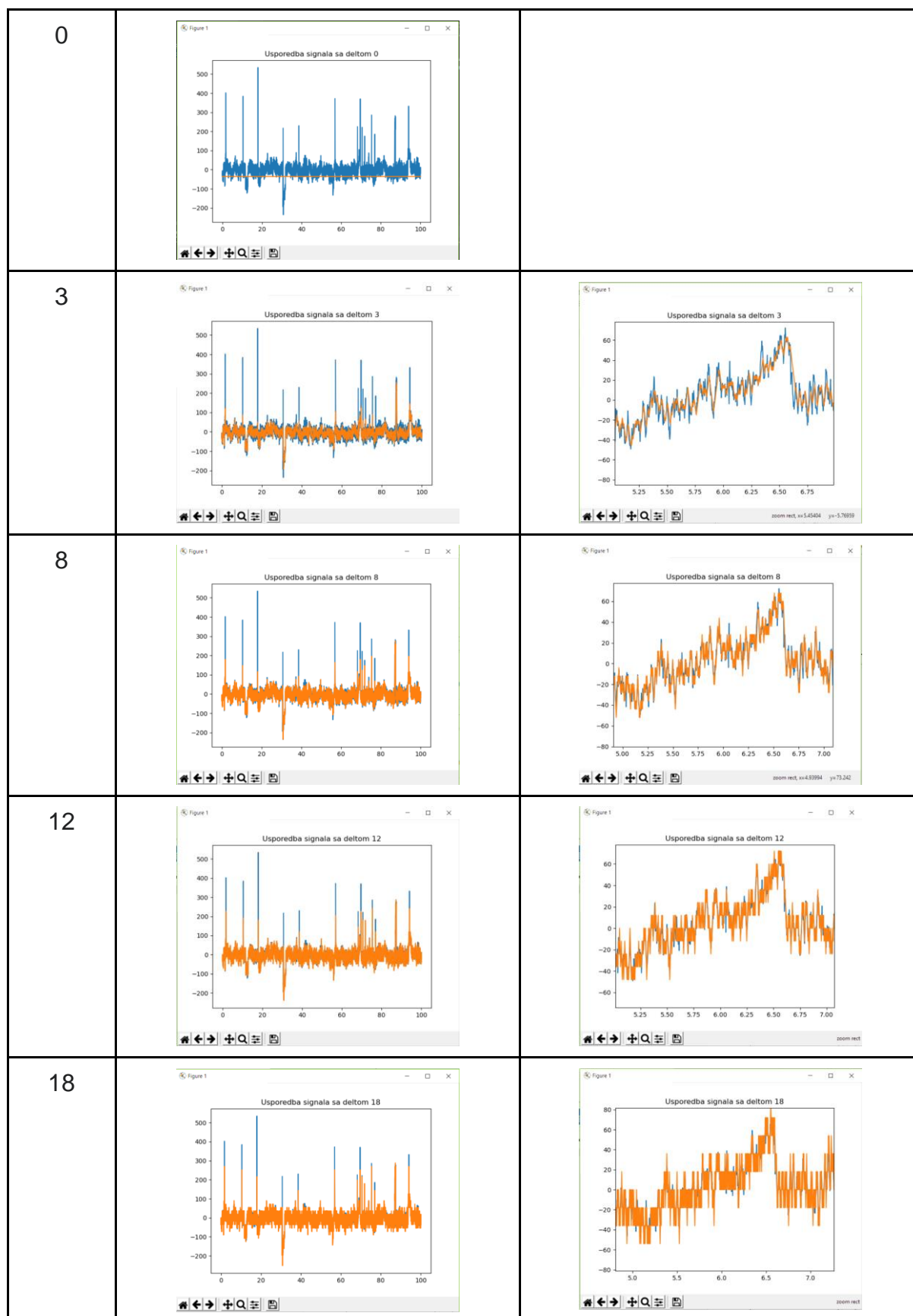
```
for i in range(30):
    pocetnaVrijednost, delta, nizBita, deltaSignal = deltaModulacija(signal, i)
    noviSignal = inverznaDeltaModulacija(pocetnaVrijednost, delta, nizBita)

    x = np.linspace(0, 100, len(signali[0]))
    fig, pt = plt.subplots(1)
    plt.title('Usporedba signala sa deltom ' + str(i))
```

```
plt.plot(x, signal)
plt.plot(x, deltaSignal)
plt.show()
```

U nastavku su prikazani neki od originalnih i komprimiranih signala sa različitim deltama. Prikazani su originalni i približeni plotovi. Približeni plotovi služe da se preciznije odredi “nabolja” delta.

Delta Δ	Originalni plot	Približeni plot
----------------	-----------------	-----------------



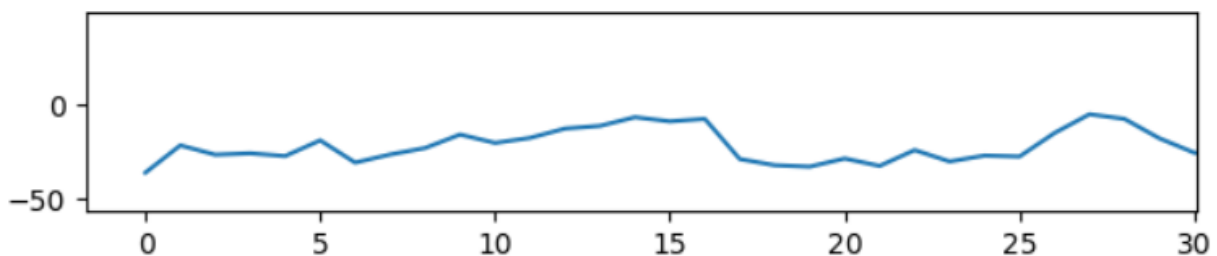
Kako bi se povećao stepen kompresije (koji će biti detaljno objašnjen u nastavku), ubačen je dodatni dio koda koji smanjuje frekvenciju uzorkovanja originalnog signala. Koeficijent smanjenja frekvencije uzorkovanja u kodu je nazvan

koeficijentDesetkovanja, a predstavljen je kao globalna varijabla. U programu ovaj koeficijent je postavljen na četiri (4), a to te vrijednosti se došlo na isti način kao i do vrijednosti koeficijenta delte. Kodovi funkcije smanjenja frekvencije uzorkovanja signala u koderu odnosno funkcije povećavanja frekvencije uzorkovanja signala u dekoderu dati su u nastavku.

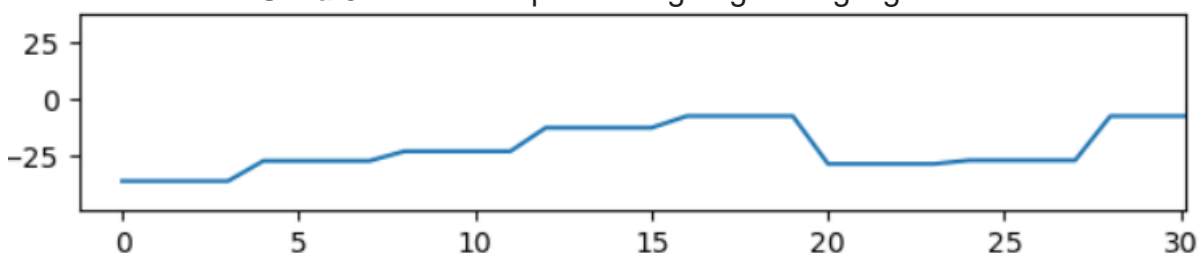
```
def smanjiFrekvencijuUzorkovanjaSignala(signal, koeficijentDesetkovanja):  
    noviSignal = []  
  
    for i in range(len(signal)):  
        if(i % koeficijentDesetkovanja == 0):  
            noviSignal.append(signal[i])  
  
    return noviSignal
```

```
def povecajFrekvencijuUzorkovanjaSignala(signal, koeficijentDesetkovanja):  
    noviSignal = []  
  
    for i in range(len(signal)):  
        noviSignal.append(signal[i])  
        for j in range(koeficijentDesetkovanja - 1):  
            noviSignal.append(noviSignal[-1])  
  
    return noviSignal
```

Kako je *koeficijentDesetkovanja* postavljen na vrijednost četiri (4), to znači da 4 uzastopne vrijednosti u signalu se svode na prvu vrijednost (prvu od te četiri). Dio približenog originalnog i desetkovanog signala slijedi u nastavku

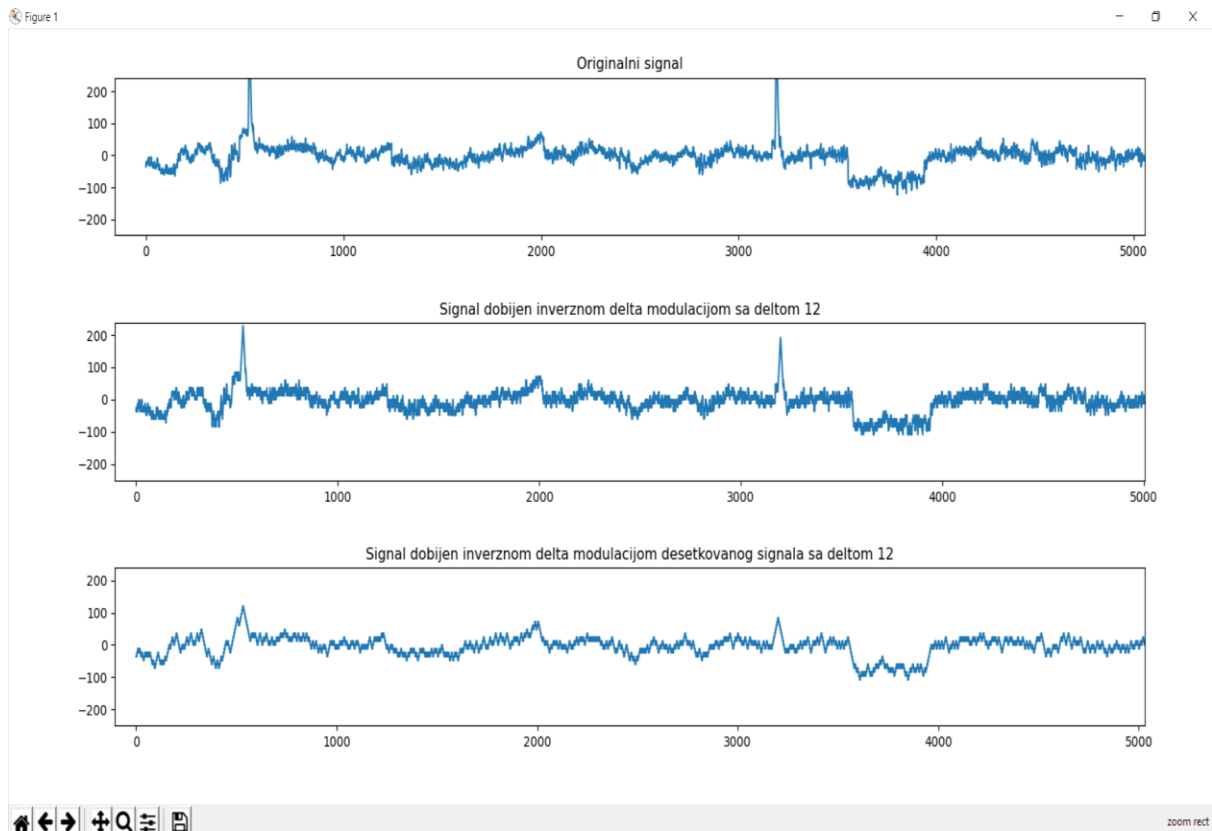


Slika 5.2.2: Prikaz približenog originalnog signala



Slika 5.2.3: Prikaz približenog desetkovanog signala

Prije nego bude analiziran stepen kompresije algoritma, slijedi prikaz tri grafa i to samo prvih oko 5000 vrijednosti (da bi se detaljnije vidjeli signali). Uzevši pomenuti signal te koeficijente *delta* i *koeficijentDesetkovanja* respektivno 12 i 4 dobijamo tri sljedeća grafa na slici 5.2.4.



Slika 5.2.4: Prikaz originalnog, komprimiranog i desetkovanog pa komprimiranog signala

Sada prokomentirajmo postignuti stepen kompresije za opisani proces. Originalni signal sadrži 30504 float podataka. Jedan float vrijednost zauzima 4 bajta memorije. Desetkovani signal (sa koeficijentom desetkovanja četiri) sadrži 7626 float podataka. Primjenom delta modulacije na desetkovani signal, dobijamo 3 vrijednosti:

- Pocetna vrijednost - jedna vrijednost od 4 bajta
- Delta koeficijent - jedna vrijednost od 4 bajta
- Niz bita - niz od 7625 bita

$$\text{stepen kompresije} = \frac{30504 * 4B}{7625 + 4B + 4B} = \frac{976128}{7657} = 127,48$$

Ovakav stepen kompresije je relativno vrlo dobar, ali bitno je za napomenuti da se proces smanjenja frekvencije ulaznog signala mogao izvršiti i na prva dva obrađena

algoritma. Smanjenje frekvencije doprinosi kompresiji (u ovom slučaju kompresija je uvećana 4 puta) ali i znatno umanjuje preciznost izlaznog signala što se može vidjeti i na slici 5.2.4.

Stepen kompresije opisanog algoritma bez smanjenja frekvencije bi iznosio 31.87 što je i dalje nekoliko puta uspješnija kompresija nego kada su korišteni Shannon-Fano ili Huffmanovi algoritmi. Mnogo je parametara koji utiču na to da li će kompresija određenog signala biti uspješna ili ne, ali na kraju se može zaključiti da od predstavljenih algoritama, za kompresiju EEG signala, najpogodniji je algoritam delta modulacija.

Zaključak

Osnovna tema ovog seminarskog rada bila je implementacija i praktična izvedba Shannon-Fano, Huffman i delta modulacija memorijske kompresije podataka EEG signala. Također izvršena je i osnovna teoretska analiza svih bitnih dijelova spomenutih algoritama te prikaz koda programa napisanog u Python programskom jeziku. Uz to prikazane su prednosti, mane i razlike implementiranih algoritama.

Na samom kraju prethodnog poglavalja prokomentirana je uspješnost delta modulacija algoritma za kompresiju EEG signala. Ukoliko je kvaliteta izlaznog signala tog algoritma zadovoljavajuća, s porastom broja podataka stepen kompresije teži broju 128. Ako uzmemo primjer da se signal snima sa 32 elektrode te da jedan snimak sadrži 30504 *float* vrijednosti (kao što je to slučaj sa setom podataka odakle je izdvojen testni signal za ovaj seminarski rad) to znači da koristeći delta modulacija implementirani algoritam ukupna memorija bi se komprimirala sa oko 4MB na oko 30KB.

Ovaj rad je rezultirao uspješnom implementacijom spomenuta tri algoritma te može poslužiti kao osnova za daljnji rad na poboljšanju kompresije EEG i sličnih signala.