Univerzitet u Sarajevu

Elektrotehnički fakultet u Sarajevu

Odsjek za računarstvo i informatiku

Digitalno procesiranje signala

Projekat

MPEG-1 Layer 1 Enkoder GUI

**Student: Ajdin Šuta**

**Broj Indeksa: 19163**

Sadržaj

[Popis slika 3](#_Toc139578363)

[1. Uvod 4](#_Toc139578364)

[2. Osnovne informacije 5](#_Toc139578365)

[3. MP3 Enkoder GUI 6](#_Toc139578366)

[3.1. Preduslovi 6](#_Toc139578367)

[3.2. Parametri programa 6](#_Toc139578368)

[3.3. Čitanje datoteke 7](#_Toc139578369)

[3.4. Filtriranje podopsega 8](#_Toc139578370)

[3.5. Filtriranje uzoraka 9](#_Toc139578371)

[3.6. Psihoakustični model 10](#_Toc139578372)

[3.6.1. Efekat maskiranja 10](#_Toc139578373)

[3.6.2. Kritični opsezi 12](#_Toc139578374)

[3.6.3. Izračunavanje pragova maskiranja 14](#_Toc139578375)

[3.7. Skaliranje i kvantizacija 16](#_Toc139578376)

[3.8. Bitstream formatiranje 17](#_Toc139578377)

[4. Optimizacije 17](#_Toc139578378)

[5. Grafičko korisničko sučelje 21](#_Toc139578379)

[6. Rezultati 23](#_Toc139578380)

[7. Reference 24](#_Toc139578381)

# Popis slika

[Slika 1: Prikaz MP3 enkodera 5](#_Toc139228515)

[Slika 2: Mogući bitrate-ovi 7](#_Toc139228516)

[Slika 3: Djelimični prikaz čitača zaglavlja 8](#_Toc139228517)

[Slika 4: Broj poziva funkcija 18](#_Toc139228518)

[Slika 5: Optimizacija psihoakustičkog modela 18](#_Toc139228519)

[Slika 6: Optimizovana vremena izvršavanja 20](#_Toc139228520)

[Slika 7: Izgled GUI 21](#_Toc139228521)

# Uvod

Ovaj projekt je implementacija MPEG-1 Layer 1 enkodera i snimača koristeći Python. Projekat se sastoji od nekoliko funkcija koji omogućavaju korisniku snimanje zvuka, spremanje snimljenog zvuka u datoteku, reprodukciju snimljenog zvuka i kodiranje zvuka u MP3 format datoteke.

Odabran je MPEG-1 Layer 1 standard naspram Layer 3 jer je Layer 3 znatno kompleksniji (ali i efikasniji) dok istovremeno sadrže iste dijelove što se tiče Digitalnog Procesiranja Signla (DPS) i Layer 1 je znatno jednostavniji po drugim aspektima (nema kodiranje entropije...). Kako je glavna uloga ovog projekta edukativne namjere, enkoder nije značajno brz, te je preporučljivo kodirati datoteke manjih veličina.

Glavna funkcija projekta je funkcija kodiranja MP3, koja uzima audio bafer i kodira ga u MP3 format datoteke. Proces kodiranja uključuje nekoliko koraka kao filtriranje, uzorkovanje podopsega, psihoakustički model proračun i kvantizacija.

Projekt također uključuje funkcije za snimanje zvuka, pohranjivanje snimljenog zvuka u datoteku i reprodukciju snimljenog audio. Ove funkcije koriste PyAudio biblioteku za interakciju sa audio hardverom računara.

Korisničko sučelje projekta implementirano je pomoću Tkinter biblioteke, koja korisniku pruža jednostavan GUI kako bi jednostavnije pristupio mogućostima projekta.

Da biste pokrenuli glavni program, jednostavno pokrenite datoteku main.py i stupite u interakciju sa GUI.

# Osnovne informacije

MPEG-1 Audio koding standard je definisan u **Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 3: Audio** (JTC, 1993). Standard je sačinjen od tri sloja, gdje Layer 1 je najjednostavniji. Osnovni princip je prikazan na sljedećoj slici:

A picture containing text, screenshot, font, line

Description automatically generated

Slika 1: Prikaz MP3 enkodera

**Layer I**

Ovaj sloj sadrži osnovno mapiranje digitalnog audio ulaza u 32 podopsega, fiksnu segmentaciju za formatiranje podataka u blokove, psihoakustički model za određivanje adaptivne alokacije bitova i kvantizaciju pomoću kompandiranja (smanjenja signal-buka omjera) i formatiranja blokova. Teoretski minimalno kašnjenje kodiranja/dekodiranja za sloj I je oko 19 ms.

**Layer II**

Ovaj sloj pruža dodatno kodiranje alokacije bitova, faktora razmjera i uzoraka. Koriste se različiti okviri. Teoretski minimalno kašnjenje kodiranja/dekodiranja za sloj II je oko 35 ms.

**Layer III**

Ovaj sloj uvodi povećanu rezoluciju frekvencije zasnovanu na hibridnoj banci filtera. Dodaje drugačiji (neuniformni) kvantizator, adaptivnu segmentaciju i entropijsko kodiranje kvantiziranih vrijednosti. Teoretski minimalno kašnjenje kodiranja/dekodiranja za sloj III je oko 59 ms.

Zajedničko stereo kodiranje može se dodati kao dodatna karakteristika na bilo koji od slojeva.

# MP3 Enkoder GUI

## Preduslovi

Za pokretanje projekta, potrebne su sljedeće biblioteke:

* Numpy – Koristi se za većinu kalkulacije unutar projekta za efikasnije izračunavanje
* PyAudio – Koristi se za snimanje i reproduciranje snimljenog audio po šablonu koji će kasnije biti opisan
* Wave – Koristi se za pristup audio datotekama I/O
* Tkinter – Daje osnove za grafičko korisničko sučelje čime olakšava pristup korisnika

## Parametri programa

Kao što već znamo, frekvencija uzorkovanja, poznata i kao brzina uzorkovanja, je osnovni koncept u digitalnoj audio obradi i odnosi se na broj uzoraka uzetih u sekundi iz analognog audio signala kako bi se pretvorio u digitalni prikaz. Određuje brzinu kojom se originalni kontinuirani audio valni oblik diskretizira ili uzorkuje. U sastavu programa podržano je nekoliko frekvencija koje su definisane ISO/IEC 11172-3 standardom: 44.1 kHz, 48 kHz i 32 kHz.

Odabirom jednog od ponuđenih frekvencija uzorkovanja, koriste se odgovarajuće tabele u obliku datoteka koje sadrže kritične brzine opsega, apsolutne pragove i kritične granice opsega. Ovi parametri su predefinisani u sastavu ISO/IEC 11172. Annex B Tables i Annex D Psychoacoustic model 1. Za više informacija, konsultovati se sa literaturom.

U slučaju MP3 datoteka, bitrate se odnosi na broj kilobita podataka koji se koriste za predstavljanje jedne sekunde zvuka. Određuje nivo kompresije primijenjen na audio podatke, kao i rezultujuću veličinu datoteke i kvalitet zvuka.

Veće brzine u bitovima rezultiraju većim veličinama datoteka i boljim kvalitetom zvuka jer izdvajaju više podataka za predstavljanje audio signala. To znači da veći bitrate omogućava precizniju reprodukciju originalnog zvuka, hvatajući suptilnije detalje i nijanse.

Mogući izbori bitrate pri enkodiranju su navedeni na sljedećoj slici:

A screenshot of a computer

Description automatically generated

Slika 2: Mogući bitrate-ovi

Kako je projekt napravljen da bude što jednostavniji, trenutno su podržani PCM WAVE audio datoteke koje mogu biti 8, 16 ili 32 bita kodirane. Samim ovim praćen je standard za MPEG-1 Layer 1, gdje varijabilno kodiranje je podržano tek od Layer 3.

Tabele koje su uključene u osnovnoj direktoriji projekta, predstavljaju sljedeće. Zaviseći od frekvencije uzorkovanja, bivaju odabrane tabele koje predstavljaju vrijednosti za odgovarajuće kritične opsege i apsolutne pragove, koji će nam biti potrebni kasnije u psihoakustičkom modelu. Jedan od ovih apsolutnih pragova jeste Barkova skala, koja predstavlja psihoakustičku skalu koja je korištena da predstavi frekvencijske opsege ljudskog sluha.

## Čitanje datoteke

U sastavu projekta napravljena je posebna WavRead klasa koja će da izdvoji sve potrebne informacije iz zaglavlja PCM Wave audio datoteke. Kada je objekat kreiran, samo je zaglavlje WAVE datoteke pročitano da bi se odredili odgovarajući parametri te da li je ikako validna WAVE datoteka. Previše pažnje neće biti posvećeno ovom dijelu, jer nema posebne veze sa Digitalnim Procesiranjem Signala. Pored WavRead također su implementirane BitStream i CircBuffer klase koje pomažu pri efektivnijem manipuliranju podataka kako bi sačuvali prozor fiksne veličine audio uzoraka za procesiranje.

A screen shot of a computer program

Description automatically generated with medium confidence

Slika 3: Djelimični prikaz čitača zaglavlja

Nakon što se pročitaju podaci iz zaglavlja datoteke, sačuvani su u input\_buffer varijablu, koja će biti proslijeđena EncoderParameters funkciji da dobijemo sve potrebne parametre za uspješno kreiranje MP3 datoteke.

## Filtriranje podopsega

Prvi glavni dio enkodera jeste filtriranje podopsega. Specificirano je u ISO/IEC standardu i mora se pratiti za pravljenje validnog MP3 datoteke.

# Precompute the cosine window

        n = np.arange(512)

        cosine\_window = np.cos((2 \* np.arange(32)[:, None] + 1) \* (n - 16) \* np.pi / 64)

        # Compute the filterbank using broadcasting

        filterbank = cosine\_window \* filter\_coeffs()[None, :]

        # Allocate the subband\_samples array

        subband\_samples = np.zeros((params.nch, 32, 12), dtype='float32')

Kao što vidimo u kodu iznad, najprije je potrebno izračunati kosinusni prozor. U ovom slučaju, korišteno je da smanji spektralno curenje i poboljša frekvencijsku rezoluciju. Zatim potrebno je izračunati filterbank pomoću filter\_coeffs() funkcije koja prosto učitava tabelu vrijednosti pronađene u datoteci LPfilterprototype u niz tipa nparray. Također je korištena metoda broadcastinga koja je sastavni dio NumPy što omogućava izvršavanje operacija nad nizovima različitih dužina i oblika.

Na kraju, potrebno je izračunati uzorke podopsega gdje params.nch predstavlja broj audio kanala, 32 predstavlja broj podopsega i treći parametar predstavlja broj okvira procesiran za jedan blok. Ovo je dalje objašnjeno u sjedećem koraku.

Ovim smo dobili filterbank sa 32 bandpass filtera koji su korišteni da podjele audio signal na 32 podjednako odvojena podopsega. Svaki od ovih filtera ima 512 uzoraka impulsnog odziva. Ove filterbanke nam daju primarne separacije frekvencija za enkoder, i obnovljive filtere za dekoder. Izlazni uzorci filterbanke su kvantizirani.

## Filtriranje uzoraka

MPEG-1 standardi specificira veličinu okvira od 512 audio uzoraka za Layer 1 enkoder.

# In each block 12 frames are processed, which equals 12x32=384 new samples per block.

            for frm in range(FRAMES\_PER\_BLOCK):

                samples\_read = input\_buffer.read\_samples(SHIFT\_SIZE)

                # If all samples have been read, perform zero padding.

                if samples\_read < SHIFT\_SIZE:

                    for ch in range(params.nch):

                        input\_buffer.audio[ch].insert(np.zeros(SHIFT\_SIZE - samples\_read))

                # Filtering = dot product with reversed buffer.

                for ch in range(params.nch):

                    subband\_samples[ch, :, frm] = np.dot(filterbank, input\_buffer.audio[ch].reversed())

Ovaj dio koda čita iz ulaznog buffera, te provjerava da ako su svi uzorci pročitani, biva dodan nula padding nad cirkularnim bufferom. Što se tiče samog filtriranja, ono je izvršeno izračunavanjem dot produkta filtrebanke sa obrnutim audio podacima. Obrnuti audio podaci su korišteni jer filterbanka je dizajnirana da radi sa obrnutim signalom. Samo filtriranje može biti izvršeno pomoću standardne konvolucije ili ekvivalentno, obrtanjem signalom i uzimanjem dot produkta. Sa standardnom konvolucijom, može izgledati ovako:

subband\_samples[ch, :, frm] = np.convolve(filterbank, input\_buffer.audio[ch].reversed(), mode='valid')

## Psihoakustični model

Glavni cilj MP3 enkodiranja jeste postizanje visokih omjera kompresije dok istovremeno treba smanjiti degradaciju audio kvaliteta. Ovo je postignuto iskorištavanjem određenih karakteristika ljudskog slušnog sistema. Neke od tih karakteristika su bile primijenjene u sklopu psihoakustičnog modela u projektu te će biti opisani u sljedećim odjeljcima. Krajnji rezultat modela treba biti signal-mask omjer (SMR) za svaki opseg. Generalno, psihoakustični model može biti podjeljen u nekoliko koraka:

1. Izračunavanje FFT-a za konverziju vremena u frekvenciju.
2. Određivanje nivoa zvučnog pritiska u svakom podopsegu.
3. Određivanje praga u tišini (apsolutni prag).
4. Pronalaženje tonalnih (sličnije sinusoidima) i netonalnih (više nalik šumu) komponenti audio signala.
5. Decimacija maskera, da bi se dobili samo relevantni maskeri.
6. Proračun pojedinačnih pragova maskiranja.
7. Određivanje globalnog praga maskiranja.
8. Određivanje minimalnog praga maskiranja u svakom podopsegu.
9. Izračunavanje omjera signal-maska ​​u svakom podopsegu.

### Efekat maskiranja

Jedna od mana, ali u ovom slučaju korist, ljudskog slušnog sistema jeste što ne može da razlikuje frekvencije koje su previše blizu jedna drugoj. Iz ovoga slijedi da će čuti samo onu koja je jača. Ovim kažemo da je jača frekvencija masker, dok maskira sve ostale frekvencije oko sebe. Pomoću ove činjenice, moguće je odbaciti sve nepotrebne frekvencije koje čovjek u većini slučajeva neće ni čuti te samim tim dobijamo manje ukupno zauzeće same audio datoteke. Kako bi primijenili ovu činjenicu, potrebno je izračunati funkciju maskiranja koja nam daje nivo zvučnog pritiska za svaku frekvenciju blizu maskera gdje sve frekvencije ispod nje su nečujne.

table = params.table

    x = np.abs(np.fft.rfft(samples \* table.hann) / FFT\_SIZE)

    np.log10(x + EPS, out=x)

    x \*= 20

    # Since the dynamic range of the FFT is 96 dB, we scale the magnitudes so that the maximum is 96 dB.

    x -= np.max(x)

    x += 96

    scf = table.scalefactor[sfindices]

    subband\_spl = np.zeros(N\_SUBBANDS)

    for sb in range(N\_SUBBANDS):

        subband\_spl[sb] = np.max(x[int(1 + sb \* SUB\_SIZE): int(1 + sb \* SUB\_SIZE + SUB\_SIZE)])

        subband\_spl[sb] = np.maximum(subband\_spl[sb], 20 \* np.log10(scf[0, sb] \* 32768) - 10)

    peaks = np.where((x[3:-6] >= x[2:-7]) & (x[3:-6] > x[4:-5]))[0] + 3

U gornjem kodu, najprije se učitavaju potrebni parametri iz već unaprijed definisane tabele iz standarda, gdje pomoću nje izračunavamo FFT audio signala pomoću NumPy rfft funkcije. Ova funkcija izračunava DFT od realnih vrijednosti ulaznog signala koristeći brzu Fourierovu transformaciju (FFT). Rezultujući FFT koeficijenti zatim bivaju konvertovani u logaritamsku skalu, te skalirani sa faktorom 20 kako bi bili konvertovani u decibele (dB).

Sljedeći blok koda skalira magnitude podopsega kako bi maksimum bio 96 dB. Ovo je urađeno jer je dinamički opseg FFT 96 dB, te skaliranjem magnitude osiguravamo da nivo zvučnog pritiska je unutar opsega. Zatim izračunava se nivo zvučnog pritiska pojedinačnih podopsega tako što se traži maksimalna magnituda unutar svakog podopsega, koje kasnije budu konvertovane u dB pomoću log10 funkcije.

Sljedeći korak jeste pronalaženje tonalnih i netonalnih komponenti signala. Najprije ćemo objasniti za pronalazak tonalnih komponenti:

for k in peaks:

        if 2 < k < 63:

            testj = np.array([-2, 2])

        elif 63 <= k < 127:

            testj = np.array([-3, -2, 2, 3])

        else:

            testj = np.array([-6, -5, -4, -3, -2, 2, 3, 4, 5, 6])

        if np.all(tonal.spl[k] - tonal.spl[k + testj] >= 7):

            tonal.spl[k] = add\_db(tonal.spl[k - 1:k + 2])

            tonal.flag[k + testj] = IGNORE

            tonal.flag[k] = TONE

            tonal.tonal\_components.append(k)

Kako smo u prošlom bloku koda pronašli sve maksimume unutar magnituda podopsega (vrijednosti koje su veće i od prijašnje i sljedeće vrijednosti), potrebno je provjeriti za svaki od tih maksimuma da li su tonalne komponente. Ako su maksimumi označeni kao tonalnim, onda susjedi su označeni kao netonalnim komponentama.

### Kritični opsezi

Također još jedan termin usko povezan sa psihoakustičkim modelom jeste kritični opsezi. Kritični opseg jeste frekvencijski opseg u kojem ljudsko uho može samo da percipira jednu frekvenciju. Ako se još jedna pojavi unutar istog opsega, zbog efekta maskiranja, nečujna je. Ovo nas dovodi do izračunavanja netonalnih komponenti audio signala. Ovo je izvršeno na sljedeći način:

for i in range(table.cbnum - 1):

        weight = 0.0

        msum = DBMIN

        for j in range(table.cbound[i], table.cbound[i + 1]):

            if tonal.flag[i] == UNSET:

                msum = add\_db((tonal.spl[j], msum))

                weight += np.power(10, tonal.spl[j] / 10) \* (table.bark[table.map[j]] - i)

        if msum > DBMIN:

            index = weight / np.power(10, msum / 10.0)

            center = table.cbound[i] + int(index \* (table.cbound[i + 1] - table.cbound[i]))

            if tonal.flag[center] == TONE:

                center += 1

            tonal.flag[center] = NOISE

            tonal.spl[center] = msum

            tonal.noise\_components.append(center)

Ovaj kod iterira kroz svaki kritični opseg unutar audio signala. Za svaki kritični opseg, težinska suma od nivoa zvučnog pritiska podopsega je izračunata, i rezultujuća suma je korištena kako bi odredili indeks centralne frekvencije netonalne komponente. Centralna frekvencija zatim biva označena kao netonalnom, i njen nivo zvučnog pritiska je postavljen kao težinska suma za buduće iteracije.

Prva petlja iterira kroz svaki pojedinačni kritični opseg audio signala. Varijabla table.cbnum sadrži broj kritičnih opsega i niz table.cbound sadrži indekse podopsega koje pripadaju svakom pojedinačnom kritičnom opsegu.

Druga petlja iterira kroz svaki podopseg u trenutnom kritičnom opsegu. Niz tonal.flag je korišten da se odredi da li je podopseg tonalna komponenta. Ako je opseg tonalna komponenta, njegov nivo zvučnog pritiska nije dodan na težinsku sumu.

Na kraju, centralna frekvencija netonalnog komponenta je označena kao netonalna komponenta pomoću tonal.flag niza. Nivo zvučnog pritiska netonalne komponente je zatim postavljen kao težinska suma podopsega, i indeks netonalne komponente je dodan u listu tonal.noise\_components.

Sada kako smo izračunali potrebne tonalne i netonalne komponente, možemo logički zaključiti da će neki od ovih komponenti se nalaziti ispod minimalnog praga čujnosti ljudskog uha te moraju biti odbačeni. Do ovog odbacivanja može doći u dva slučaja: kada se tonalne ili netonalne komponente manje od apsolutnog praga i kada su dvije ili više tonalne komponente unutar udaljenosti manje od 0.5 Bark. Ovaj postupak možemo vidjeti u sljedećem kodu:

# Removes tonal components that are below the hearing threshold

    tonal.tonal\_components = [k for k in tonal.tonal\_components if tonal.spl[k] >= table.hear[table.map[k]]]

    # Removes non-tonal components that are below the hearing threshold

    tonal.noise\_components = [k for k in tonal.noise\_components if tonal.spl[k] >= table.hear[table.map[k]]]

    # If the tonal components are closer than 0.5 bark, the one with the lower SPL is removed

    tonal\_components = tonal.tonal\_components

    map\_table = table.map

    bark\_table = table.bark

    for i in range(len(tonal\_components) - 1):

        this = tonal\_components[i]

        nx = tonal\_components[i + 1]

        if bark\_table[map\_table[this]] - bark\_table[map\_table[nx]] < 0.5:

            if tonal.spl[this] > tonal.spl[nx]:

                tonal.flag[nx] = IGNORE

            else:

                tonal.flag[this] = IGNORE

    tonal.tonal\_components = [k for k in tonal\_components if tonal.flag[k] != IGNORE]

### Izračunavanje pragova maskiranja

Nakon uspješnog izračunavanja tonalnih i netonalnih komponenti, potrebno je izračunati maskiranje uzrokovano od strane svih ostalih tonalnih i netonalnih komponenti. Po standardu, za tonalne maskere, indeks maskiranja je dat sa avtm = -1.525 - 0.275 \* zj - 4.5, dok za netonalne maskere je avnm = -1.525 - 0.175 \* zj - 0.5. U sljedećem kodu možemo vidjeti to izvršavanje:

# Indiviudal masking thresholds are calculated for each tonal component.

    masking\_tonal = []

    masking\_noise = []

    # Vectorized operations for tonal masking

    for i in range(table.subsize):

        zi = table.bark[i]

        zj = table.bark[table.map[tonal.tonal\_components]]

        dz = zi - zj

        avtm = -1.525 - 0.275 \* zj - 4.5

        vf = np.where(dz < -1, 17 \* (dz + 1) - (0.4 \* x[tonal.tonal\_components] + 6),

                      np.where(dz < 0, dz \* (0.4 \* x[tonal.tonal\_components] + 6),

                               np.where(dz < 1, -17 \* dz, -(dz - 1) \* (17 - 0.15 \* x[tonal.tonal\_components]) - 17)))

        masking\_tonal.append(x[tonal.tonal\_components] + vf + avtm)

Za tonalne komponente, prag maskiranja je izračunat korištenjem tonal.tonal\_components niza, koji sadrži indekse tonalnih komponenti audio signala prijašnje izračunatu. Prag maskiranja je izračunat korištenjem Barkove skale, gdje je izračunat kao funckije razlike Barkovih vrijednosti trenutnog podopsega i tonalnih komponenti, te i amplituda tih istih tonalnih komponenti. Prag maskiranja zatim biva dodan na amplitudu tonalnih komponenti kako bi dobili ukupni prag maskiranja za taj podopseg.

# Vectorized operations for noise masking

    for i in range(table.subsize):

        zi = table.bark[i]

        zj = table.bark[table.map[tonal.noise\_components]]

        dz = zi - zj

        avnm = -1.525 - 0.175 \* zj - 0.5

        vf = np.where(dz < -1, 17 \* (dz + 1) - (0.4 \* x[tonal.noise\_components] + 6),

                      np.where(dz < 0, dz \* (0.4 \* x[tonal.noise\_components] + 6),

                               np.where(dz < 1, -17 \* dz, -(dz - 1) \* (17 - 0.15 \* x[tonal.noise\_components]) - 17)))

        masking\_noise.append(x[tonal.noise\_components] + vf + avnm)

Za netonalne komponente ili bučne komponente, prag maskiranja je izračunat na sličan način kao kod tonalnog praga maskiranja, samo sa drukčijim parametrima.

U ovoj sekciji, korištene su odgovarajuće optimizacije u kodu za brže izvršavanje te se samim tim smanjila čitljivost i razumijevanje samog koda, međutim ove optimizacije će kasnije biti objašnjene u kasnijem odjeljku.

Nakon svega, možemo izračunati globalni prag maskiranja kao sumu individualnih opsega maskiranja za svaki podopseg audio signala. Svaki pojedinačni prag maskiranja je izračunat u prethodno objašnjenom kodu.

# Global masking threshold

    masking\_global = np.zeros(table.subsize)

    for i in range(table.subsize):

        maskers = np.concatenate(([table.hear[i]], masking\_tonal[i], masking\_noise[i]))

        masking\_global[i] = add\_db(maskers)

Sada imamo pragove maskiranja svih opsega, ali ti podopsezi nisu oni koji nas interesuju za kvantizaciju. Potrebno je da izračunamo efekt maskiranja za 32 podjednako-odvojena podopsega za koje smo izračunali uzorke. Kako bi ovo uradili, moramo za sva 32 podopsega naći minimalni prag među svim opsezima koji pripadaju tom podopsegu.

mask = np.zeros(N\_SUBBANDS)

    for sb in range(N\_SUBBANDS):

        first = table.map[int(sb \* SUB\_SIZE)]

        after\_last = table.map[int((sb + 1) \* SUB\_SIZE - 1)] + 1

        mask[sb] = np.min(masking\_global[first:after\_last])

Kao zadnji korak psihoakustičkog modela, potrebno je izračunati signal-mask omjer za svaki podopseg:

# Compute the signal-to-mask ratio

    smr = subband\_spl – mask

Pomoću ovog omjera, dobijamo informacije o razlici između nivoa zvučnog pritiska i praga maskiranja svakog podopsega te će kasnije biti korišten za broj bita u kvantizaciji.

subband\_bit\_allocation = smr\_bit\_allocation(params, smr)

Način na koji radi ova funckija jeste da pronalazimo najmanji signal-mask omjer, alociramo još jedan bit na to i izračunavamo novi signal-mask omjer za taj podopseg.

## Skaliranje i kvantizacija

Kao što smo mogli vidjeti iz prijašnjeg koda psihoakustičkog modela, potrebno mu je proslijediti faktore skaliranja, o kojima ćemo nešto više reći.

Kvantizacija uzoraka podopsega će biti uniformna, što znači da floating point brojevi između -1 i 1 će biti zaokruženi na najbliži diskretni nivo kvantizacije.

sfactorindices = np.zeros(sbsamples.shape[0:-1], dtype='uint8')

    sbmaxvalues = np.max(np.absolute(sbsamples), axis=1)

    for sb in range(N\_SUBBANDS):

        i = 0

        while sftable[i + 1] > sbmaxvalues[sb]:

            i += 1

        sfactorindices[sb] = i

    return sfactorindices

Kod će iterirati kroz svaki podopseg audio signala pomoću for petlje, te za svaki podopseg će pronaći indeks faktora skaliranja koji odgovara maksimumu apsolutne vrijednosti tog podopsega. Varijabla sftable će sadržati te faktore skaliranja koje će prijašnje biti učitana iz datoteke. Kasnije ovi indeksi faktora skaliranja su korišteni za kvantizaciju uzoraka podopsega u audio signalu, čime smanjujemo ukupnu veličinu audio datoteke.

## Bitstream formatiranje

Sada imamo sve podatke potrebne za formiranje .mp3 datoteke koji možemo reprodukovati. Ovaj dio nema posebne veze sa digitalnim procesiranjem signala, te će biti objašnjen u kratkim crtama.

Sama varijabla params će sadržati sve informacije potrebne o postavljanju odgovarajućih bitova i šta oni predstavljaju unutar MP3 zaglavlja. Raspored ovih bitova možemo vidjeti u sljedećem kodu.

self.header = (self.syncword << 21 | self.mpegversion << 19 |

                       self.layer << 17 | self.crc << 16 |

                       self.bitrate << 7 | self.fscode << 10 |

                       self.padbit << 9 | self.chmode << 6 |

                       self.modext << 4 | self.copyright << 3 |

                       self.original << 2 | self.emphasis)

Nakon zaglavlja, bit alokacije su prenesene, te zatim indeksi faktora skaliranja. Na kraju, kvantizirani uzorci podopsega bivaju preneseni sa malim izmjenama. Funkcija koja izvršava ovo izgleda ovako:

bitstream\_formatting(outputmp3file,

                                 params,

                                 subband\_bit\_allocation,

                                 scfindices,

                                 subband\_samples\_quantized)

# Optimizacije

Kako je već navede, ova implementacija MP3 enkodera je znatno neefikasna te samim tim možemo pretpostaviti zašto je kasnije ovaj Layer 1 proširen sa dodatnim Layer 2 i Layer 3 sa većim parametrima pri enkodiranju. Pri pregledu koda, vidimo da najveće vrijeme izvršavanja ima pri primjeni psihoakustičkog modela gdje u prosječnom slučaju, vremenska kompleksnost main funkcije O(n^4) što je znatno neefikasno. Pored toga, izvršava se ogromna količina izračunavanja u tom modelu nad matricama što je također neefikasno.

Pošto je glavni cilj ovog enkodera da bude što jednostavniji, teško je popraviti ukupnu vremensku kompleksnost programa, no možemo uvesti optimizacije pri izračunavanju određenih vrijednosti unutar programa.

Kako Python sam po sebi nije veoma brz jezik, potrebno je koristiti optimizacijske tehnike koje nam on omogućava za znatno brže izvođenje. Ovdje nam pomažu tehnike vektorizacije i broadcasting iz NumPy biblioteke.

Vektorizacija je tehnika korištena u programima da optimiziramo performanse koda koje uključuju matematičke operacije nad nizovima ili matricama. Kako u ovom projektu ima znatna količina takvog izračunavanja, veoma je pogodno iskoristiti ovakav tip optimizacije. Ona radi na način da umjesto da iteriramo kroz svaki element u nizu ili matrici i izvršimo izračunavanje nad tim elementom, neke biblioteke i funkcije izvršavaju te operacije nad više elemenata niza ili matrice simultano. Samim ovim možemo skratiti nekoliko sekundi pri izvršavanju samog programa.

Primjer ovoga možemo vidjeti u sljedećem bloku koda:

def add\_db(values):

    # Add power magnitude values

    powers = np.power(10.0, np.array(values) / 10.0)

    return 10 \* np.log10(np.sum(powers) + EPS)

Ovdje su korištene NumPy funckije za izvršavanje ove optimizacije jer su one znatno efikasnije od prostog iteriranja i primjene operacije. Vidimo da umjesto iteriranja, prosto napravimo niz sa elementima proslijeđenim kroz parametar funkcije, te primijenimo operaciju dijeljenja nad tim nizom. Isto vrijedi i za sum i log10 funkcije. Iako je količina vremena dobivenog ovim tipom optimizacije mala, svaka sekunda je bitna pri cjelokupnog utisku korisnika te može biti od velike koristi kad se moraju neke vrijednosti više puta izračunati kao u ovom programu zbog O(n^4) kompleksnosti.

Ako iskoristimo ugrađeni profajler u PyCharm IDE, možemo vidjeti koliko se prosječno poziva izvrši u jednoj konverziji.

A picture containing screenshot, text

Description automatically generated

Slika 4: Broj poziva funkcija

Kao što vidimo, model1 funkcija nam zauzima najviše vremena izvršavanja, tj. 71.2% ukupnog vremena ili 44989 ms. Zbog ovoga, od velike je važnosti optimizirati što je više moguće navedene funkcije. Neke izvršene optimizacije će biti navedene u sljedećem kodu:

A picture containing text, font, screenshot, number

Description automatically generated



Slika 5: Optimizacija psihoakustičkog modela

U prvom bloku, u prvobitnoj verziji programa se iteriralo kroz date vrijednosti kako bi odredili maksimalne vrijednosti unutar podopsega. Ovo biva zamijenjeno ugrađenom funkcijom where iz NumPy biblioteke.

A picture containing text, screenshot, font

Description automatically generated

A picture containing text, screenshot, font

Description automatically generated

Također isto možemo uraditi i u pronalaženju tonalnih komponenti gdje gornja slika predstavlja neoptimizovanu verziju a donja optimizovanu. Umjesto da iteriramo i provjeramao da li je uslov ispunje za svaku vrijednost, prosto koristimo NumPy funkciju all gdje odma provjeravamo za sve elemente.

Izvršene su i optimizacije pri pronalaženju pojedinačnih pragova maskiranja za svaku tonalnu komponentu. Na primjer, za tonalno maskiranje operacije bivaju zamijenjene vektoriziranom verzijom umjesto prostih provjera kroz petlju čime bi dodatno povećali vremensku kompleksnost programa:

A picture containing text, screenshot, font, software

Description automatically generated

A picture containing text, screenshot, font

Description automatically generated

Nakon još dodatnih optimizacija, možemo vidjeti efekat koje su imale nad programom:

A picture containing text, screenshot

Description automatically generated

Slika 6: Optimizovana vremena izvršavanja

Vidimo da se prosječno vrijeme izvršavanja u model1 smanjilo, te također i za neke druge funkcije. Iako su promjene u vremenu izvršavanja male, sa korisnikove strane mogu biti veoma primjetne.

# Grafičko korisničko sučelje

S ciljem lakšeg pristupa korisnika, omogućeno je direktno snimanje audio datoteke koja je podržana za konverziju u MP3, te ručno biranje audio datoteke za konverziju. Osnovni izgled aplikacije je sljedeći:

A screenshot of a computer

Description automatically generated

Slika 7: Izgled GUI

Kao što vidimo sa slike iznad, korisnik može da snimi, reproducira i sačuva audio datoteku pa je direktno konvertuje ili da odabere drugu audio datoteku. Parametri snimanja audio datoteke unutar samog programa su frekvencija uzorkovanja od 44.1 kHz kao 16 bitni PCM audio sačuvan sa .wav ekstenzijom. Ovo je implementirano pomoću PyAudio biblioteke koja omogućava lagan pristup ovim tipovima podataka. Odredišna datoteka kreirana ovim načinom će uvijek biti sačuvana u izvorni direktoriji programa i biće sačuvana kao output.wav. Dužina trajanja ove audio datoteke može biti maksimalno 5 sekundi.

Za odabir datoteke pri konvertovanju, potrebne su dvije stvari: da korisnik odabere izvornu datoteku koja mora biti PCM Wave audio formata i odredišnu datoteku koja može imenovati i imati ekstenziju .mp3 što omogućava da većina audio playera je mogu reprodukovati. U slučaju da korisnik unese neispravnu datoteku sa lošim parametrima, npr. loša frekvencija sempliranja, varijabilni način kodiranja PCM audio datoteke..., doći će do bacanja izuzetka pri čemu će korisnik biti obaviješten šta je tačno izazvalo taj izuzetak.

Pored toga, korisnik može da odabere bitrate sa kojim će audio datoteka biti konvertovana. Svi mogući bitrate-i koje aplikacija podržava su postavljeni unutar ComboBox-a. Kao što je već spomenuto, sa većim bitratom, kvalitet zvuka ostaje znatno veći no sa tim će i omjer kompresije znatno manji. U većini slučajeva najbolje je koristiti 320 kbit/s. U navedenim crtama možemo vidjeti najbolje koristi pojedinačnih bitrates:

* 32 kbit/s – općenito prihvatljivo samo za govor
* 96 kbit/s – obično se koristi za govor ili striming niske kvalitete
* 128 ili 160 kbit/s – kvalitet brzine prijenosa srednjeg opsega
* 192 kbit/s – brzina prijenosa srednjeg kvaliteta
* 256 kbit/s – često korištena brzina prijenosa visokog kvaliteta
* 320 kbit/s – najviši nivo podržan od strane MP3 standarda

Program također sadrži progress bar koji će reći korisniku u kojem je koraku pri konvertovanju audio datoteke. Ovo je izvedeno pomoću sljedećeg koda:

self.pb["value"] = input\_buffer.nprocessed\_samples / input\_buffer.nsamples \* 100

            self.parent.update()

Kako smo već ustanovili da ova implementacija nije najefikasnija, znamo da će vrijeme izvršavanja biti veliko čak i sa optimizacijama. S toga, da glavni prozor programa ne bi konstanto čekao funkciju da završi izvršavanje i bio blokiran, potrebno je pozvati funkciju od strane druge niti. Ovim je omogućeno da prozor i funkcija se ne izvršavaju na istoj niti gdje će čekati međusobno izvršavanje. Ovo je omogućeno sljedećim kodom:

def button\_callback(self):

        thread = threading.Thread(target=self.main(self.label\_choose\_input\_directory.cget("text"),

       self.label\_choose\_output\_directory.cget("text"),int(self.bitrate\_combobox.get())))

        thread.start()

# Rezultati

Sada možemo preći u rezultate pri korištenju programa za konverziju. Kao što je navedeno, potrebno je programu proslijediti PCM WAVE audio koji može biti 8, 16 ili 32 bitni bez varijabilnog kodiranja. Rezultati nekih primjera će se nalaziti unutar ./primjeri direktorije.

Kao primjer uzet ćemo birds.wav iz ./primjeri direktorija. Ova datoteka je signed 16 bit PCM Wave audio formata i originalna veličina iznosi 1.97 MB. Pri bitrate od 320 kbit/s, vidimo da se ukupna veličina datoteke smanjila na 458 KB, što je ukupan omjer kompresije od 2014.08/458 = **4.39**. Također ako stavimo da je bitrate 64 kbit/s, ukupna veličina kompresovane datoteke iznosi 91.6 KB, što predstavlja omjer kompresije **21.99**.

Može se primijetiti da pri korištenju bitrate od 320 kbit/s, dobivena audio datoteka je skoro pa ista kao izvorna datoteka, gdje imaju minimalni gubici, dok za bitrate od 64 kbit/s veoma je teško porediti dvije datoteke zbog velikog gubitka kvaliteta. Zbog ovoga, prikladan odabir bitrate je od velike važnosti.

U direktoriji se nalaze još nekoliko audio datoteka sa kojima se možete uvjeriti o rezultatima kompresije.

# Reference

Ambikairajah, E., Davis, A., & Wong, W. (August 1997). Auditory masking and MPEG-1 audio compression. *Electronics & Communication Engineering Journal*.

JTC, J. T. (1993). *ISO/IEC 11172-3: Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 3: Audio.* ISO/IEC.