

Elektrotehnički fakultet

Univerzitet u Banjoj Luci

**IZVJEŠTAJ PROJEKTNOG ZADATKA**

iz predmeta

**SISTEMI ZA DIGITALNU OBRADU SIGNALA**

Student: Mentori:

Vidić Luka 11102/21 prof. dr Mladen Knežić

prof. dr Mitar Simić

ma Vedran Jovanović

dipl. inž. Damjan Prerad

Februar 2024. godine

# Opis projektnog zadatka

Cilj projektnog zadatka je implementacija sistema za dodavanje muzičkih audio efekata korištenjem razvojnog okruženja ADSP-21489 i na personalnom računaru pomoću Python jezika i njegovi biblioteka NumPy i SciPy. Pored same implementacije potrebno je izvršiti i odgovajuće profilisanje koda i njegovu optimizaciju. Audio efekti su osnovi muzičke produkcije i mogu se, prema načinu obrade signala, podijeliti na sledeće grupe :

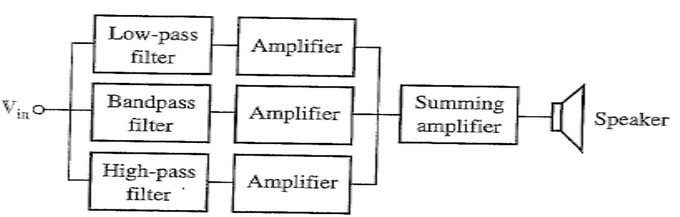
* filtriranje – niskopropusni filtri, equalizer
* vremenski promjenljivi filtri – wah-wah, phaser
* kašnjenje – vibrato, flanger, chorus, echo, delay
* nelinearne obrade – kompresija, limiter, distorzija, noise gate
* specijalni efekti – panning, reverb, surround, pitch shifter, rotary speaker...

Ovi audio efekti su podijeljeni u grupe po težini implementacije i zahtjevana je implementacija bar tri efekta.

# Izrada projektnog zadatka

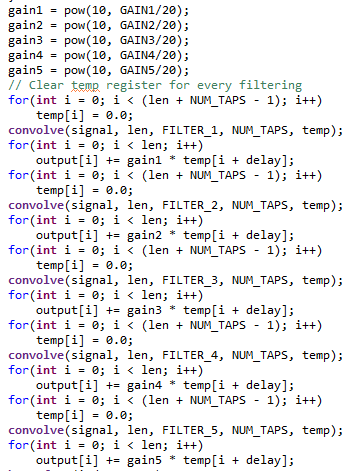
U konrektnom rješenju projektnog zadatka implementirana su četiri audio efekta: equalizer, wah-wah efekat, flanger i tremolo. Daćemo kratku teorijsku osnovu svakog efekta nakon čega ćemo pojasniti i implementaciju. Prije nego što pređemo na implementacije efekata, pomenućemo da je signal nad kojim je vršeno testiranje efekata složenoperiodični signal trajanja od 3 sekunde koji se sastoji od zbira sinusoida sledećih frekvencija (u Hz): {50, 120, 260, 440, 520, 720, 1020, 1250, 1400}. Ovaj signal generisan je u Pythonu i eksportovan u vidu header fajla *compound\_signal.h* u CrossCore projekat gdje je implementiran program za ADSP procesor. Frekvencija odmjeravanja koja je uzeta za generisanje signala je 10 kHz.

**Equalizer** je primjer primjene selektivnog filtriranja audio signala koji omogućuje da izdvojimo tonove iz određenih frekvencijskih opsega, nakon čega ih možemo utišavati ili pojačavati množenjem sa odgovarajućim faktorima pojačanja koji predstavljaju parametre equalizer-a. Equalizeri se realizuju pomoću filtara, počevši sa niskopropusnim filtrom na koji se nadovezuje niz filtara propusnika opsega, a završava se visokopropusnim filtrom. Na svaki od filtara dovodi se ulazni signal nakon čega se filtriranjem izdvajaju ciljni opsezi, koji se potom množe sa odgovarajućim faktorima pojačanja i na kraju sabiraju čime dobijamo izlazni audio signal. Blok šema equalizator-a sa tri opsega koja ilustruje prethodno opisani proces data je na sledećoj slici.



Slika 2.1 – *Blok šema equalizer-a sa tri opsega*

Napredniji equalizeri mogu imati i promjenljive parametre graničnih frekvencija koje izdvajaju opsege, ali u našem slučaju dizajniran je equalizer sa pet opsega definisani sledećim fiksnim graničnim frekvencijama (u Hz): (0,299) - (300,599) - (600, 899) - (900, 1199) - (1200,1500). Granične frekvencije su odabrane tako da vrše jasno izdvajanje komponenata složenoperiodičnog testnog signala. Filtri su projektovani u Pythonu koristeći se Parks-McClellan algoritmom koji se može koristiti iz biblioteke SciPy pozivom funkcije *remez*. Potrebni argumenti koje proslijeđujemo funkciji su dužina filtra, granične frekvencije, sekvenca pojačanja koja opisuje tip filtra te opciono relativne težine svakog od opsega filtra. Nakon projektovanja filtara i primjene nad ulaznim signalom uz pojačanja u Pythonu, koeficijenti filtara su izvezeni u header fajl *equ\_filters.h* da bi se mogli iskoristiti u CrossCore projektu. Blok koda koji obavlja funkciju equalizer-a na ADSP procesoru dat je na sledećoj slici.



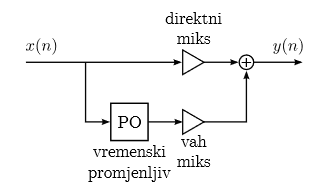
Slika 2.2 – *Blok koda equalizer-a*

Prvo je potrebno definisane parametre pojačanja koji su u decibelima, pretvoriti u pojačanja bez dimenzije prema formuli:

|  |  |
| --- | --- |
|  | (2.1) |

Za operaciju stepenovanja koristimo funkciju *pow* iz zaglavlja *math.h*. Prethodno u funkciji smo alocirali memoriju za pomoćni niz *temp* koji će da skladišti međurezultate filtriranja ulaznog signala sa odgovarajućim filtrima equalizer-a da bi potom mogli te međurezultate da pomnožimo sa prethodno proračunatim pojačanjima, te akumulišemo u izlazni signal. Operacija filtriranja realizovana je konvolucijom ulaznog signala sa filtrima pozivom funkcije *convolve* iz zaglavlja *filter.h*. Pošto je rezultat konvolucije niz dužine koja je jednaka zbiru dužina ulaznog signala i filtra – 1, onda niz *temp* mora imati toliko alocirane memorije. Koeficijenti koji se uzimaju iz tog niza za formiranje izlaza, uzimaju se sa kašnjenjem koje je fiksno i koje unose FIR filtri (kao što su filtri dobijeni *remez* funkcijom). Ovo kašnjenje se može proračunati kao (dužina filtra – 1) / 2 i ono je smješteno u varijablu *delay* koja se nalazi u funkciji equalizer-a. Dakle, operacija koju obavlja naš equalizer je filtriranje ulaznog signala sa 5 različitih filtara pomoću konvolucije, izdvajanje odgovarajućih odmjeraka i njihovo moženje sa faktorom pojačanja za taj opseg, te akumulacija u izlazni signal.

**Wah-wah** efekat spada u grupu efekata dobijenih vremenski promjenljivim filtrima. Ovaj efekat se postiže filtriranjem signala filtrom propusnikom opsega koji ima promjenljivu centralnu frekvenciju i uzak propusni opseg. Filtrirani signal se miksa sa originalnim čime se dobija izlazni signal. Blok šema koja opisuje ovaj proces data je na sledećoj slici.



Slika 2.3 – *Blok dijagram sistema za realizaciju vah-vah efekta*

Centralna frekvencija filtra se mora mijenjati tokom iteracija algoritma, što je postignuto pomoću trougaonog signala koji linearno povećava i spušta frekvencije od donje do gornje granice i nazad. Parametri ovog efekta su: vah frekvencija, donja granična frekvencij, gornja granična frekvencija i faktor prigušenja (NAPOMENA: Ove granične frekvencije nisu direktno vezane za granične frekvencije filtra, nego za opseg frekvencija kroz koji će centralna frekvencija „šetati“ tokom izvršavanja algoritma). Jednačine koje implementiraju algoritam date su u nastavku:

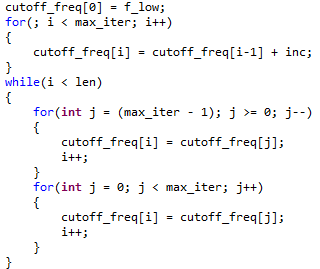
|  |  |
| --- | --- |
| ,  ,  . | (2.2) |

Gdje se Q1 i F1 računaju u zavisnosti od parametra faktora prigušenja i centralne frekvencije PO filtra respektivno, prema formulama:

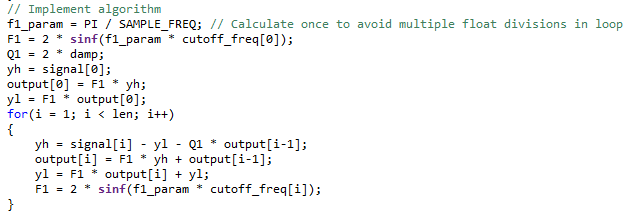
|  |  |
| --- | --- |
| ,  . | (2.3) |

U formulama d je parametar prigušenja, Fs je frekvencija odmjeravanja i Fc je centralna frekvencija filtra. Izlazni signal na koji je primjenjen wah-wah efekat je signal yb iz jednačina (2.2).

U nastavku su dati blokovi C koda koji implementiraju efekat. Prvo je prikazan način generisanja trougaonog signala koji mijenja centralne frekvencije, a nakon toga i implementacija samog algoritma (koji nije ništa drugo nego implementacija jednačina (2.2)).



Slika 2.4 – *Generator trougaonog signala*



Slika 2.5 – *Implementacija wah-wah algoritma*

Sa slike 2.4 vidimo da se niz koji sadrži centralne frekvencije popunjava linearno trougaonim zakonom u granicama od donje granične frekvencije (f\_low) do gornje (f\_high) i nazad sa korakom koji je definisan varijablom *inc*. Ova varijabla se računa u odnosu na wah frekvenciju kao količnik te frekvencije i frekvencije odmjeravanja. Nakon generisanja trougaonog signala prelazimo na implementaciju algoritma. Prvu iteraciju zbog nultih početnih uslova izvršavamo van petlje kao što vidimo sa slike 2.5, nakon čega se izvršava petlja prema jedinačinama (2.2) i (2.3). Radi čuvanja memorije nismo kreirali nove nizove yl, yb i yh kao u jedinačini (2.2) već smo koristili samo po jednu varijablu (jer nam trebaju odmjerci sa jediničnim kašnjenjem kroz iteracije), a odmjerke yb smo stavljali direktno u izlazni niz.

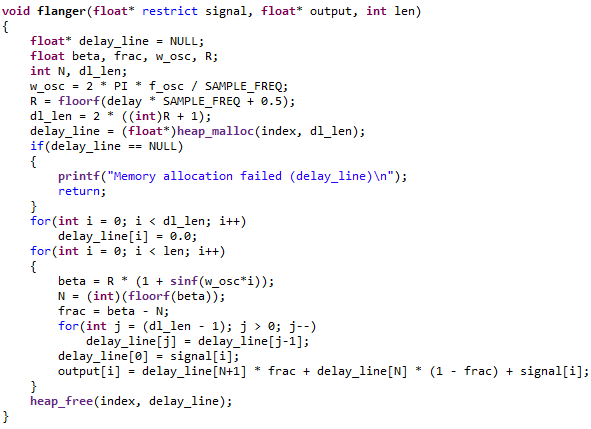
**Flanger** efekat je efekat dobijen kašnjenjem ulaznog signala koji se zasniva na vibrato efektu. Osnovna ideja je da se promjena relativne udaljenosti između slušaoca i zvuka opaža kao promjena visine zvuka što se u digitalnom domenu realizuje kao sistem sa promjenljivim kašnjenjem. Jedačina diferencija koja opisuje ovaj proces je sledeća:

|  |  |
| --- | --- |
|  | (2.4) |

Za realizaciju efekata kao što su vibrato i flanger, kašnjenje je periodična funkcija niskofrekventnog oscilatora:

|  |  |
| --- | --- |
|  | (2.5) |

U formuli R je maksimalno željeno kašnjenje izraženo u odmjercima, R = round(T\*FS). FS je frekvencija odmjeravanja, a T je parametar maksimalnog željenog kašnjenja u vremenskoj dimenziji koji korisnik unosi. Učestanost oscilovanja zavisi od frekvencije oscilatora. Za efekat vibrata tipična kašnjena su od 0.1ms do 10ms, a frekvencije oscilovanja su od 5Hz do 14Hz. U slučaju vibrata izlazni signal je potpuno određen jednačinom (2.4). Sa druge strane kod flanger efekta frekvencija oscilovanja je manja i tipično je 1Hz, a izlazni signal se dobija tako što se zakašnjeli signal iz jednačine (2.4) kombinuje sa originalnim signalom. Implementacija u C kodu data je na sledećoj slici.



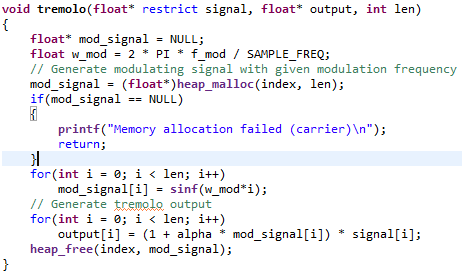
Slika 2.6 – *Implementacija flanger efekta*

Za implementaciju kašnjenja alocirali smo novi niz *delay\_line* u koji se smještaju odmjerci ulaznog signala shodno kašnjenju.Vidimo da se za implementaciju algoritam blago modifikovao tako da se umjesto jednog zakašnjelog odmjerka uzmu dva, ali skalirana odgovarajućim *frac* faktorom. Razlog ovoga je mnogo bolji kvalitet zvuka, jer onaj koji je implementiran originalnom jednačinom (2.4) bude veoma grub za slušanje.

**Tremolo** je efekat baziran na konceptu amplitudne modulacije signala. Jednačina diferencije koja opisuje amplitudnu modulaciju ulaznog signala je:

|  |  |
| --- | --- |
|  | (2.6) |

Koeficijent se naziva dubina modulacije i predstavlja jedan od parametara efekata baziranih na amplitudnoj modulaciji i on se kreće u granicama [0,1] u zavisnosti od toga koliko želimo da izrazimo modulaciju. Signal je modulišući sporopromjenljivi prostoperiodični signal čija frekvencija predstavlja parametar algoritma i za tremolo efekat se kreće u granicama od 0.1 Hz do 20 Hz. Implementacija algoritma u C kodu data je na sledećoj slici.



Slika 2.7 – *Implementacija tremolo efekta*

Vidimo sa slike da je efekat implementiran baš po jednačini (2.6) s tim što smo prvo izgenerisali modulišući signal u novom nizu, pa potom računali izlaz čime smo otvorili put boljoj optimizaciji.

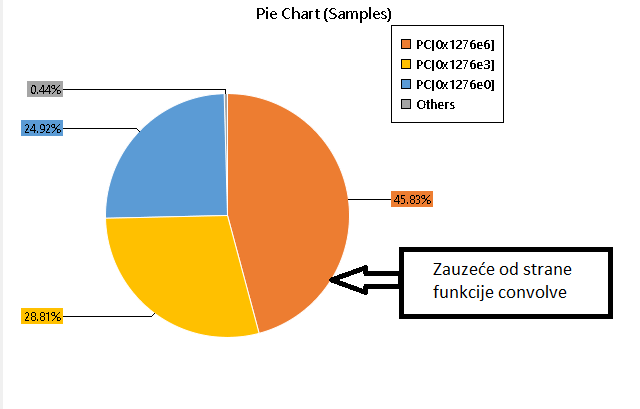
Nakon prezentacije implementiranih efekata osvrnućemo se na profilisanje koda i optimizacije. Profilisanje koda izvršili smo koristeći se mehanizmima zaglavlja *cycle\_count.h* koje nam daje metode *START\_CYCLE\_COUNT*, *STOP\_CYCLE\_COUNT* i *PRINT\_CYCLES*. Ove metode služe za dobijanje i ispisivanje informacija o broju ciklusa koje DSP utroši za izračunavanje dijelova koda izmedju poziva *START\_CYCLE\_COUNT* i *STOP\_CYCLE\_COUNT*. Za svaki od algoritama ćemo izvršiti ovakvo profilisanje. Prvo ćemo dobiti rezultate bez ikakvih optimizacija, nakon toga ćemo uključiti kompajlerske optimizacije (optimizacije za maksimalnu brzinu –Ov100) i na kraju dodati i *#pragma SIMD\_for* za optimizacije petlji. Dobijeni rezultati (brojevi ciklusa) za pojedine algoritme smješteni su u sledećoj tabeli.

Tabela 2.1 – *Brojevi ciklusa ADSP procesora za različite algoritme i nivoe optimizacije*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Bez optimizacija | Kompajlerske optimizacije | Vektorizacija petlji |
| Equalizer | 3615472733 | 3608166794 | 3603352042 |
| Wah-wah | 8996954 | 5397518 | (neuspješno) |
| Flanger | 40441333 | 28200953 | (neuspješno) |
| Tremolo | 5400273 | 3960261 | 1890333 |

NAPOMENA: Za polja koja spadaju pod kolone vektorizacije petlje koja su naznačena kao neuspješna, kompajler nije mogao da postigne paralelizaciju usljed zavisnosti podataka unutar petlji.

Kao što vidimo iz prethodne tabele, kompajlerske optimizacije postižu veoma dobra ubrzanja na algoritmima wah-wah (ubrzanje 40%), flanger (30%) i tremolo (27%). Problem dolazi kod algoritma equalizera gdje kompajlerske optimizacije praktično ostvaruju zanemarljiv učinak. Razlog zbog koga se ovo dešava jeste zbog korišćenja funkcije *convolve* u kojoj se troši ogroman broj ciklusa što možemo vidjeti kada pokrenemo Profiling prozor CCES razvojnog okruženja (naredna slika).



Slika 2.8 – *Grafik dobijen Profiling alatom CCES okruženja za slučaj funkcije equalizera*

Rješenje koje poboljšava drastično ovaj problem jeste da filtriranje implementiramo koristeći funkciju *fir* koja se takođe nalazi u zaglavlju *filter.h*. Sada umjesto da mi vršimo konvolucije signala i koeficijenata filtara, jednostavno pozovemo funkciju koja će u pozadini to da odradi mnogo efikasnije. Rezultati koje dobijamo su dati u narednoj tabeli.

Tabela 2.2 - *Brojevi ciklusa ADSP procesora za unaprijeđeni algoritam equalizera*

|  |  |  |  |
| --- | --- | --- | --- |
|  | Bez optimizacija | Kompajlerske optimizacije | Vektorizacija petlji |
| Equalizer | 63717536 | 57079453 | 52278612 |

Kao što vidimo, poboljšanja su drastična, a ako proračunamo ubrzanje dobijeno u odnosu na dvije implementacije sa kompajlerskim optimizacijama ono iznosi čak 98% !

Prije nego što pređemo na zaključak, razmotrićemo značajan problem koji se javlja kada probamo vektorizacije petlji. Nakon izvršavanja algoritama, dobijeni rezultati kada su petlje vektorizovane ne budu korektni. Preciznije, svaki drugi odmjerak izlaznog signala bude jednak tačno 0,000. Ovaj problem je pokušano riješiti u više navrata. Ono što je glavna sumnja zbog čega se petlje ne mogu korektno vektorizovati SIMD instrukcijama jesu problemi sa memorijom. S obzirom da je ulazni signal kao i izlazni (i dosta pomoćnih signala) smješten u niz dužine 30000 floating point tipa oni ne mogu da se smjeste u internu memoriju uređaja, zbog čega su svi nizovi alocirani i smještani u SRAM eksternu memoriju. Kada pročitamo dokumentaciju ADSP procesora, vidimo da SIMD dobavljanje podataka nije podržano za eksterne memorije kod serija 211xx,212xx i 213xx, međutim za serije 214xx (u koju spada i naša platforma ADSP-21489) SIMD i eksterna memorija nisu u problemu jer su magistrale dovoljno široke da dobave po dva podatka. Sledeća sumnja je bila memorijsko poravnanje korištenih nizova, jer korištenje SIMD instrukcija zahtjeva da podaci budu poravnani (4B poravnanje) da bi se moglo izvršiti ispravno dvostruko dohvatanje. Pošto je u rješenju kreiran jedan niz veličine 500KB koji je predstavljao heap, i sve memorije su alocirane iz njega dinamički, ovaj problem je uistinu postojao. Kada se ispišu početne adrese korištenih nizova (npr. iz prvog equalizer algoritma testirali bi početne adrese nizova signal i temp) neke stvarno ne bi bile poravnane (djeljive sa 4). Da bismo riješili ovo, pokušaj je bio izbacivanje heap memorije i statičko alociranje nizova kao globalnih sa potrebnim veličinama, a po dokumentaciji SHARC kompajlera ovakvi nizovi su podrazumijevano poravnani što se i pokaže kada se ispitaju adrese početnih elemenata. Međutim, kada riješimo problem poravnanja rezultati i dalje ostaju pogrešni. Ovo je naravno problem jer korišćenje SIMD instrukcija može da dovede do odličnih ubrzanja (npr. kao kod analize ciklusa tremolo efekta gdje se uz vektorizaciju petlje postiže ubrzanje od 65%), ali nema smisla ako rezultati nisu korektni.

# Zaključak

Primjene kompajlerskih optimizacija kao i vektorizacija petlji mogu da imaju veoma dobar uticaj na poboljšanje performansi kao što smo se mogli uvjeriti razmatrajući rezultate iz tabele (2.1). Sa druge strane, poznavanje i čitanje dokumentacije kao i razmatranje raznih funkcionalnosti koje nam nude već postojeće biblioteke može da se pokaže kao krucijalno za poboljšanje nekog algoritma. U ovo smo se jasno uvjerili kada smo pokazali poboljšanje algoritma equalizer-a, kako je samo pronalaženje i upoteba druge funkcije iz istog zaglavlja koja tačno odgovara problemu dovela do ogromnih poboljšanja performansi. Naravno sa druge strane vidjeli smo da neki algoritmi audio efekata nisu baš „prijateljski nastrojeni“ prema metodama optimizacija i preporukama. Primjeri ovih algoritama su algoritam wah-wah efekta i flanger efekta. Kod njih uočavamo postojanje ugnježdenih petlji koje imaju znatno manje iteracija nego spoljašnje, što je suprotno preporukama za efikasno izvršavanje petlji. Takođe, vidjeli smo da nad petljama koje realizuju ove efekte nije moguće iskoristiti vektorizacije petlji. Razlog je postanje zavisnosti izmedju podataka u petlji (*true dependencies*). Drugim riječima za izvršavanje jedne linije koda u petlji potreban je rezultat neke od prethodnih unutar iste iteracije (čisti primjer je petlja wah-wah efekta). Ove probleme nameće sama priroda algoritma i način implementacije, tako da se s tim moramo pomiriti ili osmisliti neki drugi način za implementaciju koji bi nam otvorio put ka rješenju ovih problema. Ovo može biti poprilično zahtjevno/nemoguće, memorijski neefikasno/neizvodivo ili da na kraju i ne donese neke značajne rezultate pogotovo za primjere dva prethnodno spomenuta algoritma.

Još nam je ostalo da prokomentarišemo i uporedimo rezultate dobijene u Pythonu i na ADSP procesoru. Zbog ograničenja maksimalnog broja stranica izvještaja, slike dobijenih signala i signala greške nećemo ovde kačiti, nego ćemo se pozivati na *Jupyter Notebook* fajl koji je kreiran u sklopu projekta. Unutar ovog fajla se nalaze python implementacije svih algoritama i njihovo reprodukovanje na audio izlaz, kao i čitanje fajlova koje generiše ADSP koji predstavljaju izlazne signala na koji su primjenjeni efekti. Nakon čitanja ovih fajlova oni se prevode u *NumPy* nizove i takođe reprodukuju nakon čega poredimo rezultate. Kao što možemo vidjeti sa slika iz datog fajla, svi algoritmi se izvšavaju identično na obe platforme a dobijene apsolutne greške su dimenzije 10-5 što je zanemarivo. Takođe kada slušamo reprodukovane signale apsolutno nema nikakvih razlika pa možemo zaključiti da su svi efekti ispravno implementirani (u odnosu na Python kod).

Predlog poboljšanja i dodatne implementacije jeste proširenje aplikacije tako da se implementiraju metode kao što su *Overlap-Add* ili *Overlap-Save* čime otvaramo put ka obradi stvarnih audio zapisa koji će da imaju mnogo veći broj odmjeraka od testnog signala koji smo mi koristili. Tada bismo mogli da primjenjujemo efekte na stvarne audio zapise (kao npr. zapis *acoustic.wav* koji je bio korišten u *notebook* fajlu za prikaz djelovanja efekta nad stvarnim audio zapisom isječka odsvirane melodije na gitari). Takođe, za stvarne primjene bismo iskoristili druge filtre odnosno projektovali na druge načine tako da relaksiramo neke kriterijume, ili uportijebimo IIR filtre da bismo smanjili broj koeficijenata, jer je 500 koeficijenata izuzetno velika dužina filtra pogotovo za real-time aplikacije (praktično neupotrebljiva).

# Literatura

[1] Vladimir Risojević, *Multimedijalni sistemi*, Univerzitet u Banjoj Luci, Elektrotehnički fakultet, 2018.

[2] Udo Zölzer, *DAFX: Digital Audio Effects, Second Edition*, John Wiley & Sons, 2011.

[3] Marion, Jean Guy Bruno, *Wah Wah - DAS 2014 Lab Report 2*, University of Sydney, 2014.

[4] Oficijalna CCES dokumentacija za SHARC procesore.

[5] Materijali sa predmeta Sistemi za digitalnu obradu signala, Osnovi digitalne obrade signala.