Elektrotehnički fakultet
Univerzitet u Banjoj Luci

IZVJEŠTAJ PROJEKTNOG ZADATKA

iz predmeta

SISTEMI ZA DIGITALNU OBRADU SIGNALA

Student:
Tanja Popović 1218/17
,

Mentori:
prof. dr Mladen Knežić
prof. dr Mitar Simić
ma Vedran Jovanović
dipl. inž. Damjan Prerad

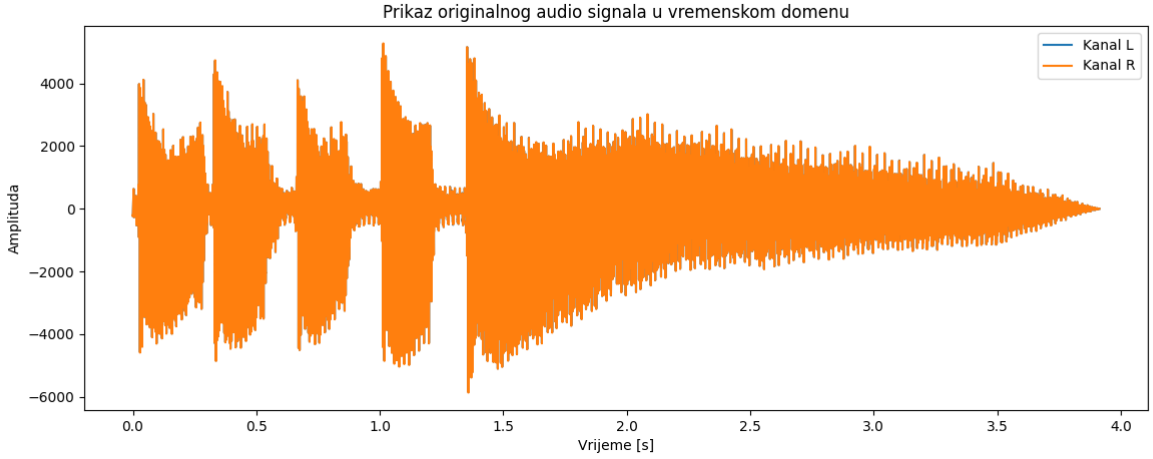
Februar 2024. godine

1. Opis projektnog zadatka

U sklopu projektnog zadatka potrebno je realizovati sistem za dodavanje muzičkih efekata u audio signal korištenjem ADSP-21489 razvojnog okruženja. Ulazni signal, muzički efekti, kao i svi izlazni signali su prvenstveno generisani unutar Python programskog jezika (program effects.py) i služe kao referenca signalima implementiranim u okviru ciljnog programskog paketa CrossCore Embedded Studio. Nakon primjene muzičkog efekta, odmjerci izlaznog signala su upisani u odgovarajuće tekstualne datoteke koje se iz Pythona čitaju, prikazuju i eksportuju kao *wav* fajlovi tako da se primijenjeni efekti mogu čuti. Pored implementacije efekata, u sklopu CCESa izvršeno je i profilisanje koda, a signali su međusobno upoređeni pomoću signala greške. Implementirano je ukupno 11 muzičkih efekata: Delay, Echo, Compressor, Noise Gate, Envelope Filter, Volume Pedal, Tape Saturation, Octave Up, Tremolo, Override i Distorsion.

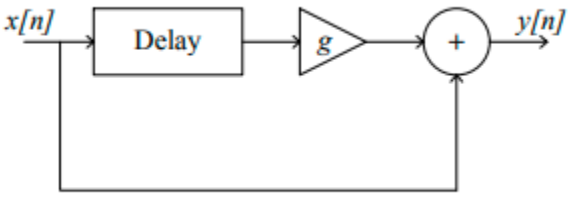
2. Izrada projektnog zadatka

Za potrebe izrade projektnog zadatka, kao ulazni signal iskorišten je konkretan zvuk gitare – audio fajl pod nazivom *guitar_sound.wav*. Radi se o stereo signalu, frekvencije odmjeravanja 44,1 kHz i trajanja 3,9 sekundi prikazan na Slici 2.1. Iako se sa Slike 2.1 ne primjeti velika razlika između lijevog i desnog kanala, radi lakše implementacije zadatka, stereo signal je konvertovan u mono signal (*source_audio_mono.wav*) i kao takav predstavlja „sirovi“ ulazni signal na koji se primjenjuju audio efekti. Softverska implementacija audio efekata zasnovana je na jednačinama diferencija odgovarajućih sistema. Za smještanje izlaznog signala alocirano je 700 kB memorije na *heapu* (172 617 odmjeraka * 4 B = 690 468 B), pri čemu je korištena eksterna memorija.



Slika 2.1 – Originalni, stereo signal

Delay je audio efekat koji vrši obradu signala zasnovanu na kašnjenju. Ukoliko se na originalni signal superponira njegova zakašnjena verzija, skalirana faktorom $g \in [-1, 1]$ koji predstavlja faktor refleksije originalnog i zakašnjenog signala dobijamo Delay efekat. Blok šema ovakvog sistema prikazan je na Slici 2.2, a softverska implementacija je zasnovana na jednačini sistema 2.1. Parametar n_0 predstavlja zakašnjene odmjerke, a može se izračunati kao umnožak kašnjenja u vremenu i frekvencije odmjeravanja. S obzirom to, unutar CCESa definisani su makroi *sample_rate*, *delay_time*, *delay_samples* i *gain*. Signali nakon primjene Delay efekte implementiran u Pythonu, zatim u CCESu i signal greške izračunat kao razlika ova dva signala prikazani su na Slikama 2.3, 2.4 i 2.5 respektivno.

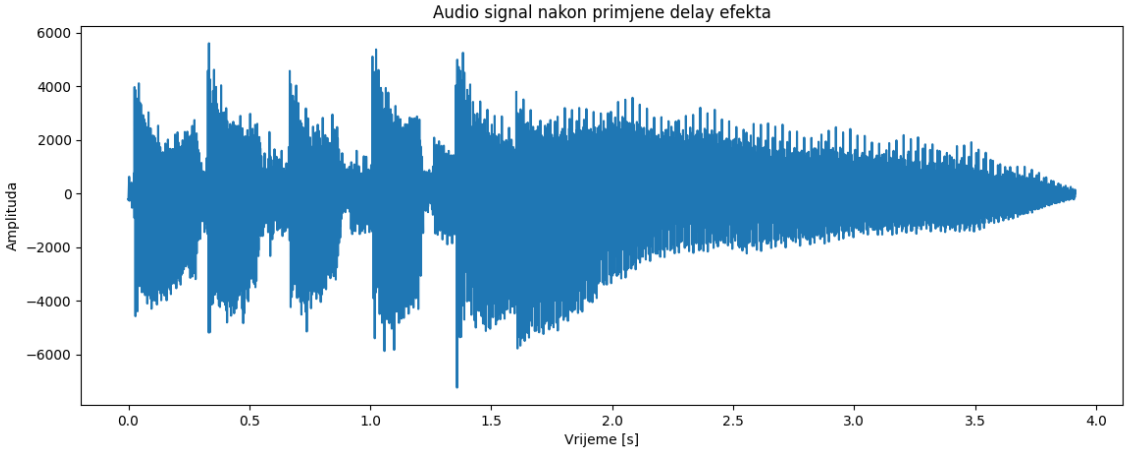


Slika 2.2 – Blok šema sistema za primjenu Delay efekta

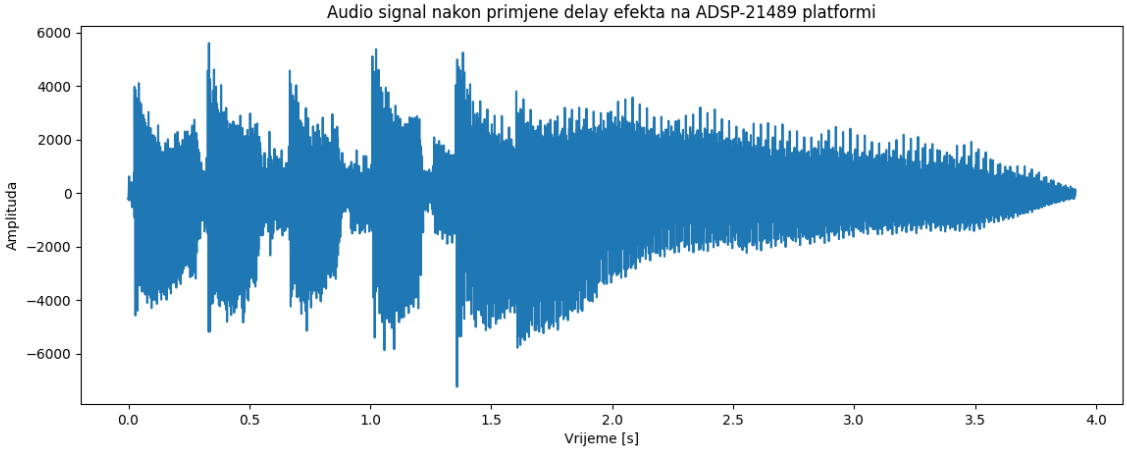
$$y[n] = x[n] + gx[n - n0] \tag{2.1}$$

Ukoliko se umjesto jedne zakašnjene replike originalnog signala, superponira više zakašnjenih replika tada se radi o višestrukom Delay efektu ili Echo efektu. Echo efekat je implementiran kao IIR filtar, na osnovu jednačine 2.2.

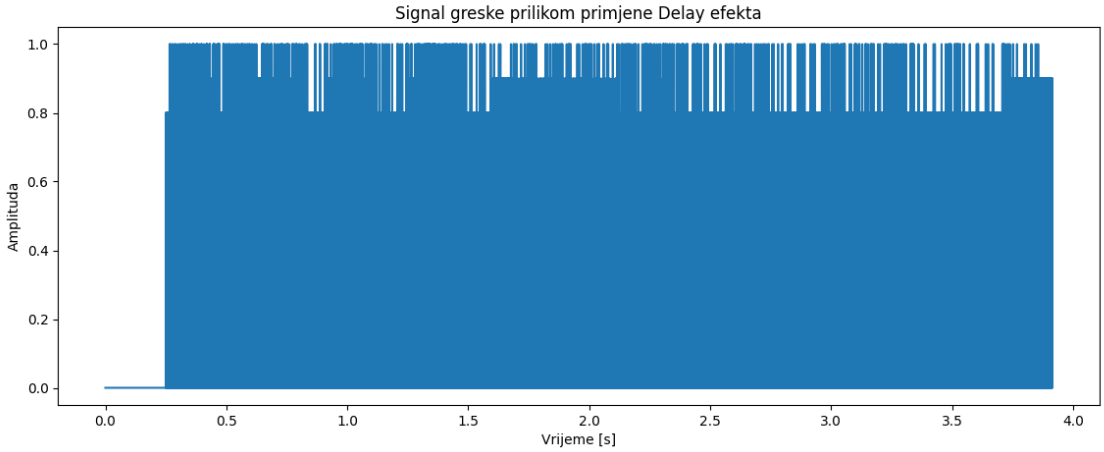
$$y[n] = x[n] - gy[n - n0] \tag{2.2}$$



Slika 2.3 – Primjena Delay efekta na originalni signal, generisan unutar Pythona



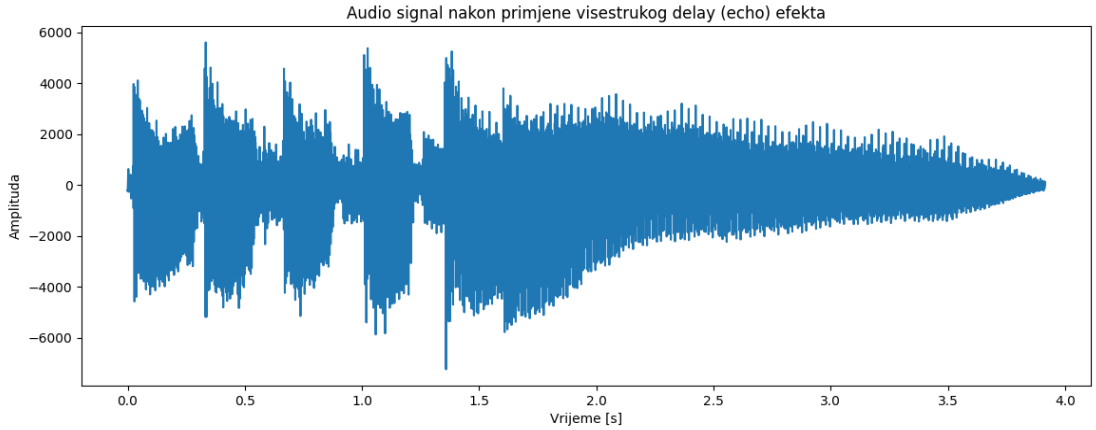
Slika 2.4 - Primjena Delay efekta na originalni signal, generisan unutar CCESa



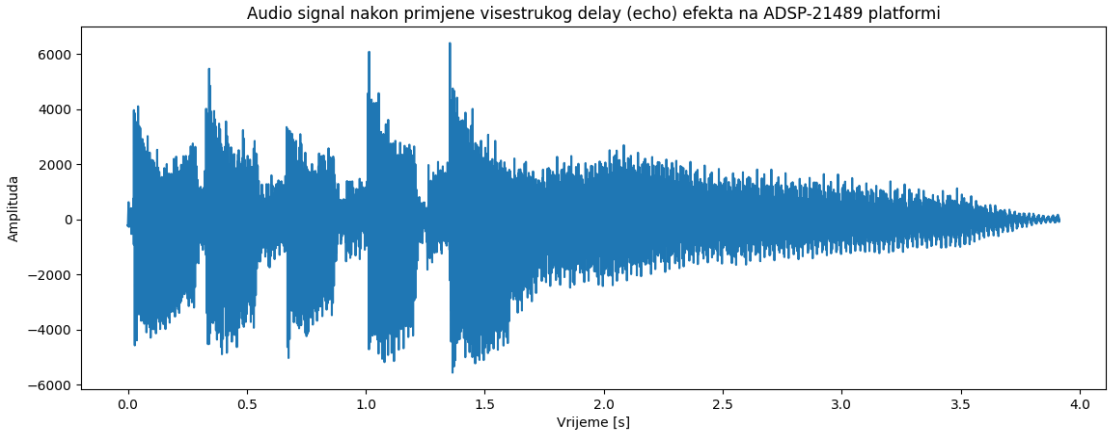
Slika 2.5 – Signal greške kod Delay efekta

Signali nakon primjene Echo efekta generisanog u Pythonu, CCESu, kao i signal greške prikazani su na Slikama 2.6, 2.7 i 2.8 respektivno. Zbog ograničenja u veličini izvještaja, detaljnija implementacija ostalih efekata (prikaz signala, signal greške) biće prikazana u dodatnom poglavlju Dodaci.

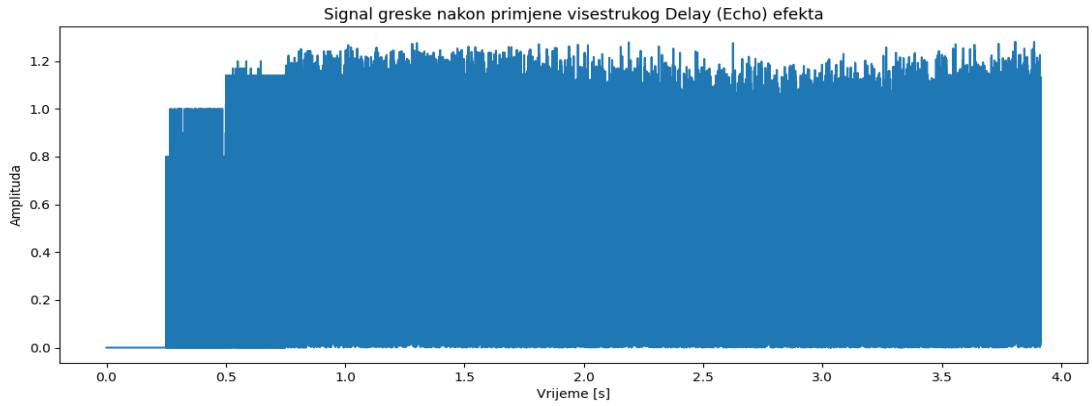
Na osnovu signala greške prikazanih na Slikama 2.5 i 2.8, vidimo da postoji razlika između signala generisanig unutar Pythona i unutar CCESa. Ukoliko se vrijednosti signala greške uporede sa vrijednostima amplitude signala, vidimo da je ta razlika zanemarivo mala, kako i treba biti. Kada razmatramo signal greške treba težiti njegovoj minimalizaciji. Razlog postojanja signala greške kod primjene ovih efekata je u različitim tipovima podataka sa kojima je rađeno, u C program jeziku to su integeri, a poznato je da Python sam dodjeljuje tip podatka na osnovu dodijeljene vrijednosti. Pored toga, neki od parametara filatara su *float* vrijednosti, te se i tu unosi greška kastovanja, tj. cjelobrojnog zaokruživanja.



Slika 2.6 – Signal nakon primjene Echo efekta, genernisan u Pythonu



Slika 2.7 –Signal nakon primjene Echo efekta, generisan u CCESu



Slika 2.8 – Signal greške kod Echo efekta

Proces profilisanja koda je obavljen pomoću dva makroa koje sadrži zaglavlje *cycle_count.h*, a prikazuju broj procesorskih ciklusa koji se potroše na izvršavanje određenog dijela koda. Broj procesorskih ciklusa koji se potroše na izvršavanje pojedinačnih funkcija koje predstavljaju muzičke efekte prikazan je u Tabeli 2.1. Profilisanje je izvršeno bez ikakve optimizacije, a kao što se može vidjeti najviše procesorskih ciklusa treba za izvršavanje funkcije *tremolo*, koja će u ovom slučaju predstavljati fokus optimizacije – na ovaj način je detektovano usko grlo (eng. *bottle neck*) u kodu.

Tabela 2.1 – Muzički efekti sa odgovarajućim brojem procesorskih ciklusa (bez optimizacije)

Delay	Echo	Compressor	Noise Gate	Volume Pedal	Tape Saturation
20 283 523	18 158 689	13 967 836	10 489 237	8 713 719	14 174 859
Octave Up	Envelope Filter	**Tremolo**	Distorsion	Override	
9 324 359	17 490 248	**25 134 797**	12 695 583	21 415 226	

Prvi korak pri optimizaciji je uključivanje kompajlerske optimizacije i dodavanje identifikatora *restrict* pored parametara funkcije. Kako se sa Slike 2.9 može vidjeti da se *tremolo* funkcija sastoji od *for* petlje sa velikim brojem iteracija, prilikom čega se u svakoj iteraciji poziva *m_function* funkcija, ista je proglašena kao *inline*. Ovakav vid optimizacije smanjuje broj procesorskih ciklusa na 18 465 873.

```
203 inline float m_function(int n)
204 {
205     return 1 + DEPTH * cosf(2.0 * M_PI * n * FLFO / SAMPLE_RATE);
206 }
207
208 void tremolo(const pm int *  restrict input_signal, pm int * restrict output_signal, dm unsigned int signal_length)
209 {
210     for(int i = 0; i < signal_length; i++)
211         output_signal[i] = input_signal[i] * m_function(i);
212 }
```

Slika 2.9 – Implementacija Tremolo funkcije

Sljedeći pokušaj optimizacije jeste ručno odmotavanje petlje, tako da imamo manje iteracija ali je blok izvršavanja proširen. Primjer ručnog odmotavanja petlje unutar *tremolo* funkcije prikazan je na Slici 2.10. Broj ciklusa nakon ovakvog načina implementacije funkcije je smanjen na 16 731 696, a pored toga posmatrani su rezultati dodavanja *pragma* direktiva koje se odnose na odmotvanje petlji kao što je *loop_unroll* sa parametrima 1000, 100 i 10. To je dalo sljedeće rezultate: 21 726 042, 18 310 214 i 17 690 884 za nabrojene parametre, respektivno. Ovakvi rezultati predstavljaju pogoršanje u odnosu na kompajlersku optimizaciju, a s obzirom na uputstvo u specifikaciji kompajlera da nije potrebno ručno odmotavati petlje već to prepustiti kompajleru, vratićemo se prvobitnoj implementaciji funkcije *tremolo* kao na Slici 2.9.

```
203 inline float m_function(int n)
204 {
205     return 1 + DEPTH * cosf(2.0 * M_PI * n * FLFO / SAMPLE_RATE);
206 }
207
208 void tremolo(const pm int *  restrict input_signal, pm int * restrict output_signal, dm unsigned int signal_length)
209 {
210     for(int i = 0; i < signal_length; i+=2)
211     {
212         output_signal[i] = input_signal[i] * m_function(i);
213         output_signal[i+1] = input_signal[i+1] * m_function(i+1);
214     }
215 }
216
```

Slika 2.10 – Ručno odmotavanje petlje unutar Tremolo funkcije

Kako su unutar CCESa parametri svih muzičkih efekata definisani kao makroi, uporedićemo brzinu izvršavanja funkcije ukoliko se parametri definišu kao globalne promjenjive. Pred toga, unutar *tremolo* funkcije je dodana druga pragma direktiva za optimizaciju petlji, koja se odnosi na vektorizaciju – SIMD_for. Primjer implementacije ovakve funkcije prikazan je na Slici 2.11. Profilisanjem ovakve funkcije, dobili smo poboljšanje u odnosu na kompajlersku optimizaciju te sada broj ciklusa iznosi 11 994 354. Ukoliko želimo da uporedimo da li je izvršavanje funkcije brže sa globalnim promjenjivim ili makroima, uz vektorizaciju, dolazimo do sljedećeg rezultata – 9 598 584 ciklusa što predstavlja značajno manje procesorskih ciklusa u odnosu na početnu vrijednost, i najbolji rezultat optimizacije. Prikaz svih rezultata optimizacije funkcije *tremolo* prikazan je u Tabeli 2.2.

```
202 float depth = 0.9;
203 int flfo = 5;
204 int samplerate = 44100;
205 inline float m_function(int n)
206 {
207     return 1 + depth * cosf(2.0 * M_PI * n * flfo / samplerate);
208 }
209
210 void tremolo(const pm int *  restrict input_signal, pm int * restrict output_signal, dm unsigned int signal_length)
211 {
212     #pragma SIMD_for
213     for(int i = 0; i < signal_length; i++)
214     {
215         output_signal[i] = input_signal[i] * m_function(i);
216     }
217 }
```

Slika 2.11 – Vektorizacija petlje i dodavanje globalnih promjenjivih umjesto makroa

Tabela 2.2 – Optimizacioni koraci za funkciju Tremolo i dobijeni rezultati

Bez optimizacije	Kompajlerska optimizacija, inline, restrict	Ručno odmotavanje petlje	#pragma no_vectorization	#pragma loop_unroll 1000, 100, 10	#pragma SIMD_for, globalne promjenjive	#pragma SIMD_for, makroi
25 134 797	18 465 873	16 731 696	16 726 042	21 726 042 18 310 214 17 690 884	11 994 354	9 598 584

S obzirom da je prvenstveno kompajlerska optimizacija značajno smanjila broj ciklusa potrebnih za izvršavanje *tremolo* funkcije, a zatim i vektorizacija, posmatraćemo kako kompajlerska optimizacija, a kako vektorizacija utiče na ostale funkcije. Upoređivanjem rezultata u Tabelama 2.3 i 2.4, vidimo da vektorizacija ipak nije uvijek najbolja tehnika za optimizaciju. Broj ciklusa izvršavanja pojedinih funkcija je ostao isti, odnosno kompajler je vid optimizacije za te funkcije prepoznao bez vektorizacije (kao što postoji pragma direktiva no_vectorization). Pored toga, forsiranje vektorizacije može dovesti do povećanja broja ciklusa prilikom izvršavanja funkcija kao što je to primjer sa Volume Pedal i Octave Up funkcijama.

Tabela 2.3 – Broj ciklusa nakon kompajlerske optimizacije

Delay	Echo	Compressor	Noise Gate	Volume Pedal	Tape Saturation
14 156 568	14 690 624	4 844 718	5 328 974	4 844 724	10 783 200
Octave Up	Envelope Filter	Tremolo	Distorsion	Override	
4 844 704	7 438 216	18 465 978	7 344 552	15 929 756	

Tabela 2.4 – Broj ciklusa nakon kompajlerske optimizacije i vektorizacije

Delay	Echo	Compressor	Noise Gate	Volume Pedal	Tape Saturation
14 156 392	14 690 624	4 844 686	5 291 502	5 080 568	10 782 736
Octave Up	Envelope Filter	Tremolo	Distorsion	Override	
5 080 134	7 438 200	9 598 584	7 344 572	15 929 376	

U specifikaciji kompajlera se dodatno razmatraju slučajevi kako uslovna grananja unutar petlji takođe značajno utiču na brzinu izvršavanja programa. Savjetuje se da se petlje nalaze unutar grananja, a ne obrnuto, ukoliko je to dozvoljeno. U konkretnom slučaju, imamo primjer implementacije funkcija *distorsion* i *override*, gdje se unutar petlje nalaze uslovna grananja koja se zbog zavisnosti od iteratora ne mogu izbaciti izvan petlje, kao što je prikazano na Slici 2.12. Ukoliko se uslovna grananja ne mogu izbaciti, savjetuje se korištenje funkcija *expected_false* i *expected_true*, kojima se vrši predviđanje rezultata uslova i na taj način ubrzava njihovo izvršavanje. Pored toga, unutar uslovnog bloka za poređenje se savjetuje poređenje neke vrijednosti sa nulom, umjesto sa nekom drugom vrijednošću. Primjer izmjene implementacije funkcije *distorsion* prikazana je na Slici 2.13.

Kao što je prikazano u Tabeli 2.4, vektorizacija nije doprinijela poboljšanju vremena izvršavanja funkcije *distorsion*. Ukoliko se uslovi unutar grananja napišu na drugačiji način, kao što je prikazano na Slici 2.12, broj ciklusa potebnih za izvršavanje *distorsion* funkcije će se povećati na 7 370 468. S obzirom na to, implementacija će ostati kao na Slici 2.13, a uz dodavanje prethodno pomenutih funkcija

rezultat izvršavanja iznosi 7 108 828 ciklusa. Ovim primjerom je pokazano da drugačiji načini implementacije mogu uticati na brzinu izvršavanja programa. Poboljšanje vremena izvršavanja *distorsion* funkcije prikazano je u Tabeli 2.5.

```
248 void distorsion(const pm int * restrict input_signal, int * restrict output_signal, unsigned int signal_length)
249 {
250
251     for(int i = 0; i < signal_length; i++)
252     {
253         int x = input_signal[i];
254         if(GI * x + 1 <= 0)
255             output_signal[i] = - GO;
256         else if(GI * x + 1 >= 0 && GI * x - 1 < 0)
257             output_signal[i] =(int)(GI * GO * x);
258         else
259             output_signal[i] = GO;
260     }
261 }
```

Slika 2.12 – Implementacija distorsion efekta

```
232 void distorsion(const pm int * restrict input_signal, int * restrict output_signal, unsigned int signal_length)
233 {
234
235     for(int i = 0; i < signal_length; i++)
236     {
237         int x = input_signal[i];
238         if(expected_false(GI * x <= -1))
239         //if(expected_false(GI * x + 1 <= 0))
240             output_signal[i] = - GO;
241         else if (expected_true(GI * x >= -1 && GI * x < 1))
242         //else if(expected_true(GI * x + 1 >= 0 && GI * x - 1 < 0))
243             output_signal[i] =(int)(GI * GO * x);
244         else
245             output_signal[i] = GO;
246     }
247 }
```

Slika 2.13 – Implementacija distorsion efekta dodavanjem prediktivnih funkcija

Tabela 2.5 – Rezultati izvršavanja *distorsion* funkcije

Kompajlerska optimizacija	Uslovno poređenje sa nultom vrijednošću	Prediktivne funkcije, poređenje sa nenultom vrijednošću
7 344 572	7 370 468	7 108 828

3. Zaključak

Razlog velikog broja odmjeraka ulaznog signala (172 617) jeste taj što se za odmjeravanje koristila velika frekvencija odmjeravanja od 44,1 kHz. Razlog korištenja ovakve frekvencije odmjeravanja je dodatna mogućnost eksportovanja modifikovanih signala u audio fajlove, tako da se pored njihovog prikaza mogu i poslušati primijenjeni efekti. Ukoliko se odlučimo za manju frekvenciju

odmjeravanja, ili neki drugi ulazni signal generisan unutar Pythona (uz, naravno poštovanje Nikvistovog kriterijuma), to će rezultovati manjim brojem odmjeraka, većom brzinom izvršavanja te manjim zauzećem memorije. To predstavlja vrlinu i manu ovakvog prostupa implementacije muzičkih efekata. Ukoliko želimo brži sistem, sa manje odmjeraka i manjim zauzećem u memoriji, jedan od načina realizacije ovakvog pristupa jeste primjena procesa decimacije originalnog signala, uz obavezno filtriranje istog.

Današnji kompajleri su veoma dobro razvijeni, te se u procesu optimizacije lako može uočiti da li se ide dobrim ili lošim putem, praćenjem upozorenja koje nam kompajler daje i vremena izvršavanja određenog koda.

Ukoliko bismo željeli primjenu vise ulančanih efekata, to je moguće postići na sljedeći način, implementiranjem pseudo koda 3.1. Pored toga, potrebno je obratiti pažnju na dodatno alociranje memorije za izlazne signale svakog pojedinačnog sistema (efekta).

```
echo(input_signal, output_echo_signal)

tremolo(output_echo_signal, output_tremolo_echo_signal)

override(output_tremolo_echo_signal, output_signal)
```

Pseudo kod 3.1 – Primjer ulančane primjene audio efekata

Kao dodatna stavka implelentiran je loading bar.

4. Literatura

[1] Materijali sa predavanja i laboratorijskih vježbi iz predmeta Sistemi za digitalnu obradu signala, Elektrotehnički fakultet, Univerzitet u Banja Luci.

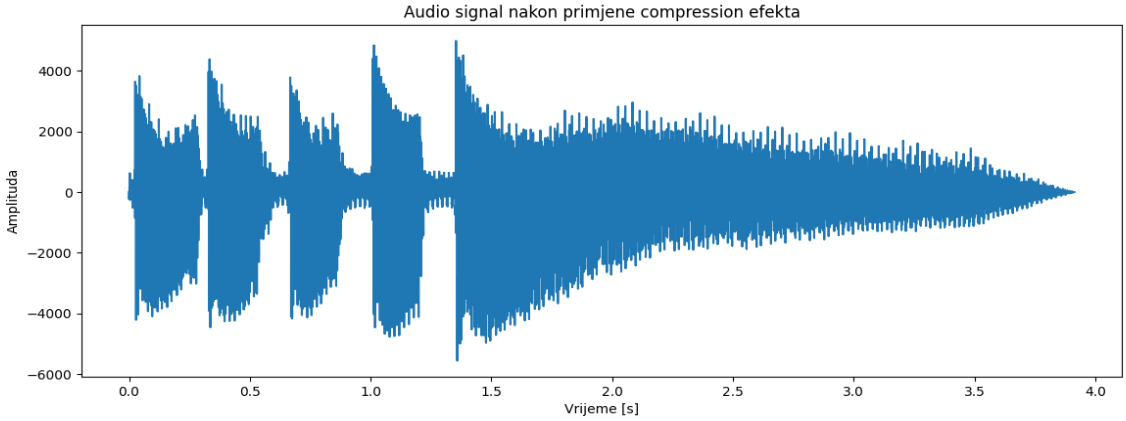
[2] CrossCore Embedded Studio, *CCES 2.9.0 C/C++ Complier Manual for SHARC Processors*, Revision 2.2, Analog Devices, May 2019.

[3] CrossCore Embedded Studio, *CCES 2.9.0 C/C++ Library Manual for SHARC Processors*, Revision 2.2, Analog Devices, May 2019.

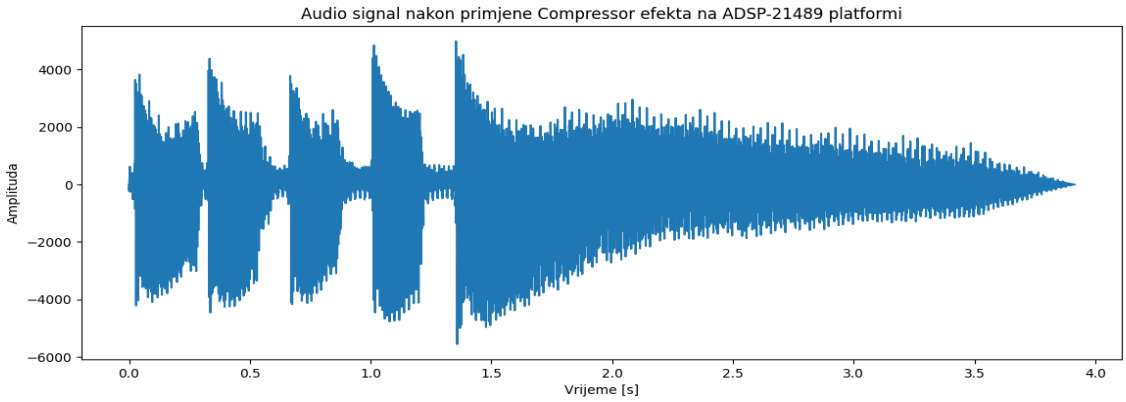
[4] Joshua D. Reiss, Andrew P. McPherson, „Audio effects. Theory, Implementation and Application“, CRC Press Taylor and Francis Group, US, 2015.

5. Dodaci

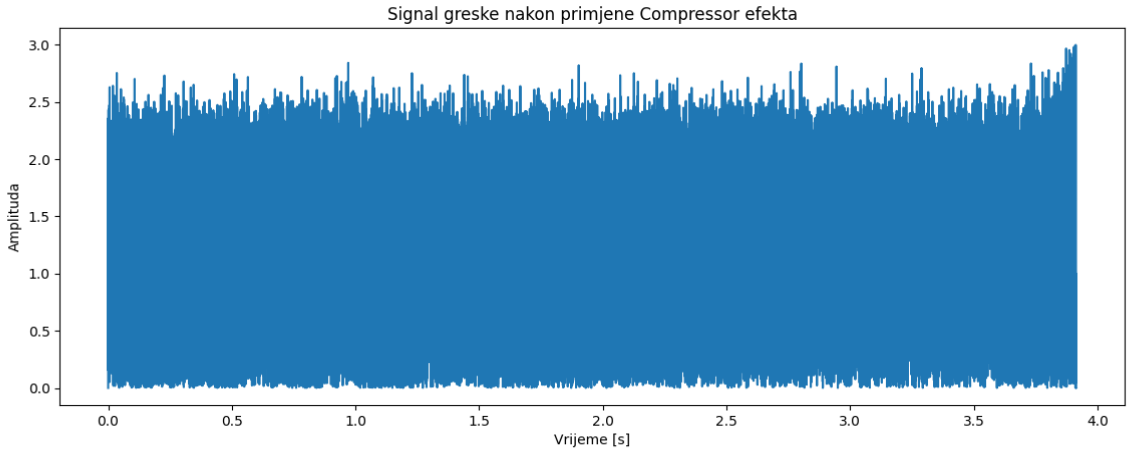
Svi implementirani efekti se mogu poslušati, a u zavisno od toga gdje su izgenerisani imaju sufiks *pygen* ili *cgen*. Tako naprimjer, Compressor efekat primijenjen na *source_audio_mono.wav* se može poslušati u sklopu fajlova *compressor_effect_pygen.wav* i *compressor_effect_cgen.wav*. Isto važi i za ostale muzičke efekte. U prilogu se nalaze prikazi svih ostalih signala nakon primjene odgovarajućeg efekta unutar Pythona i CCESa, kao i odgovarajući signal greške.



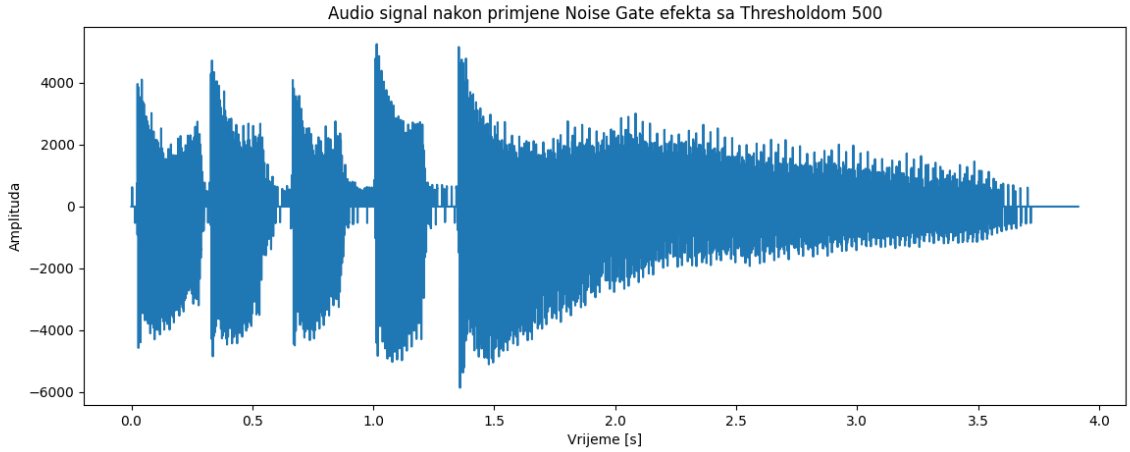
Slika 5.1 – Signal nakon primjene Compressor efekta, generisan u Pythonu



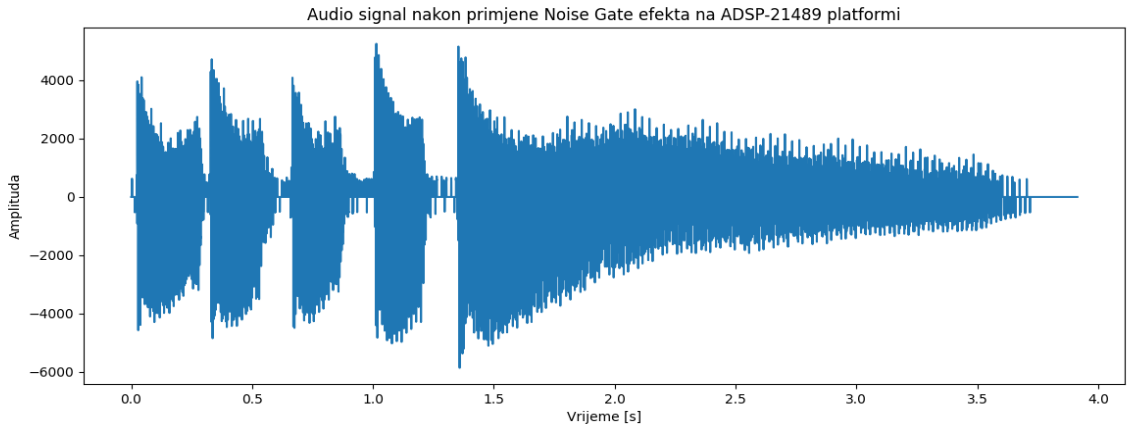
Slika 5.2 – Compressor efekat, generisan unutar CCESa



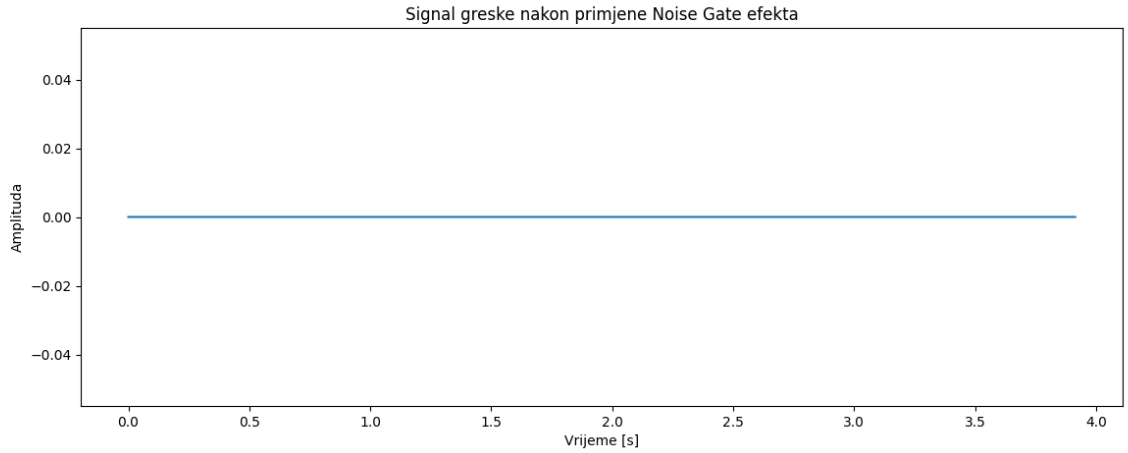
Slika 5.3 – Signal greške kod Compressor efekta



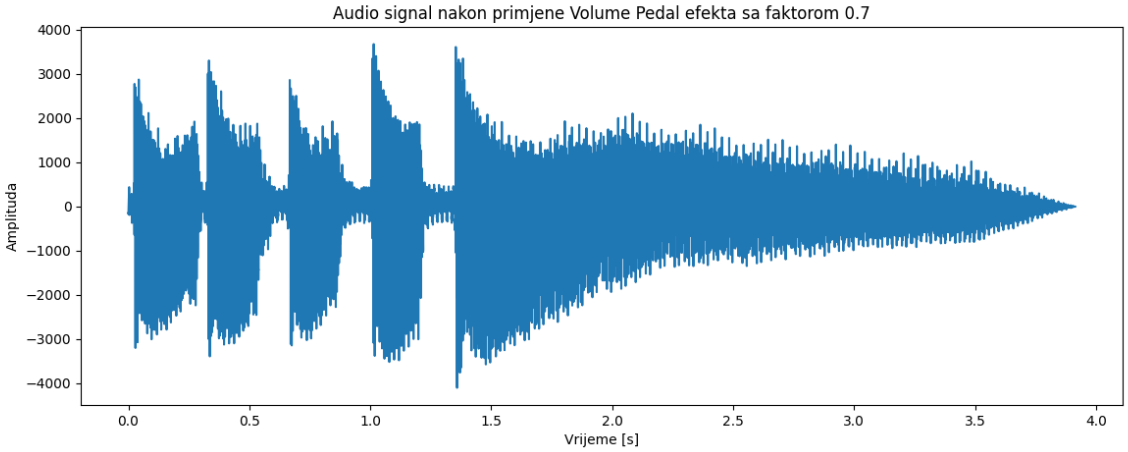
Slika 5.4 – Signal nakon primjene Noise Gate efekta, generisan u Pythonu



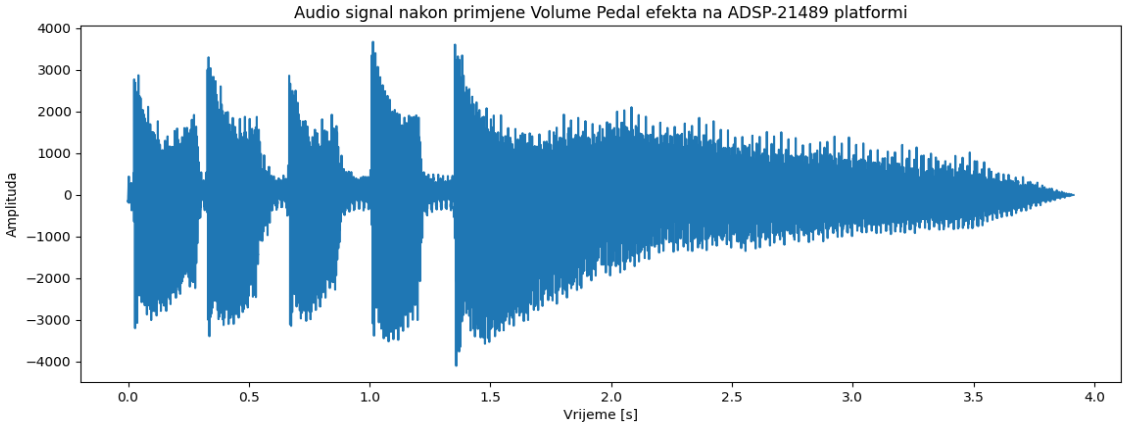
Slika 5.5 – Signal nakon primjene Noise Gate efekta u CCESu



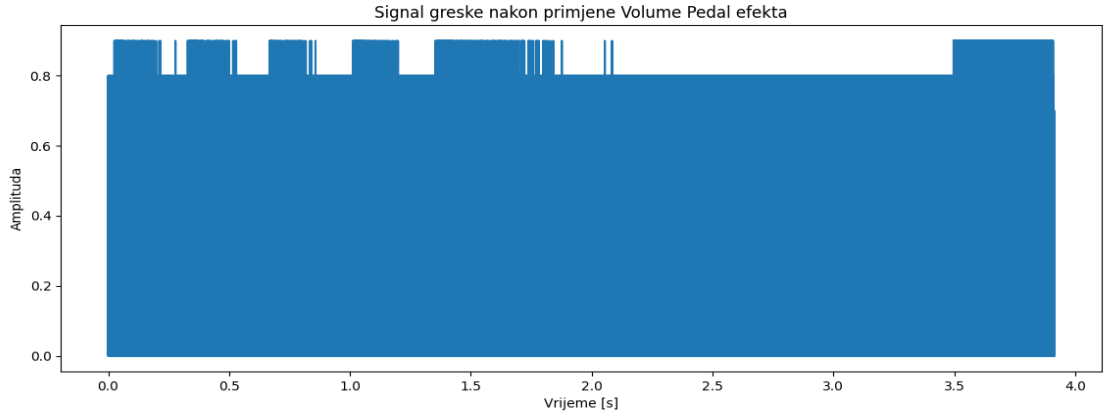
Slika 5.6 – Signal greške kod Noise Gate efekta



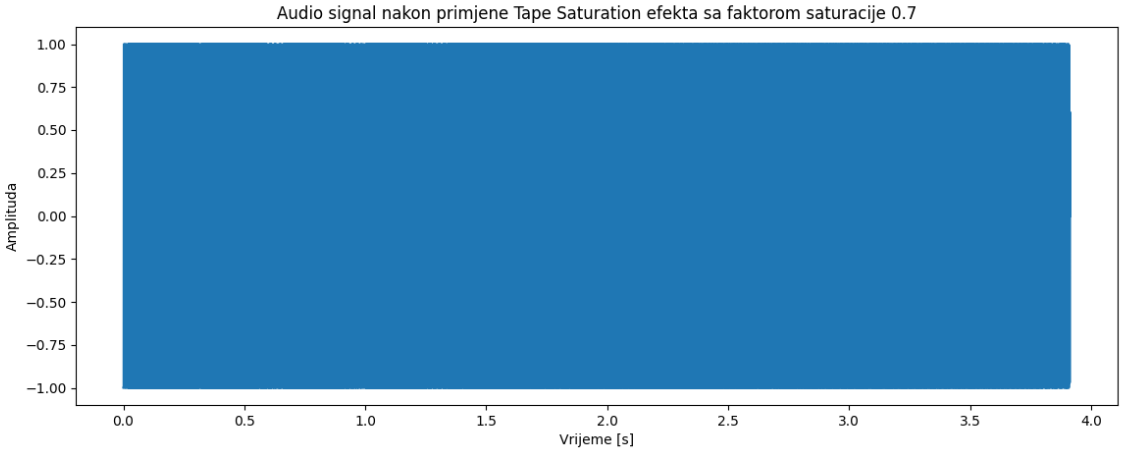
Slika 5.7 – Signal nakon primjene Volume Pedal efekta



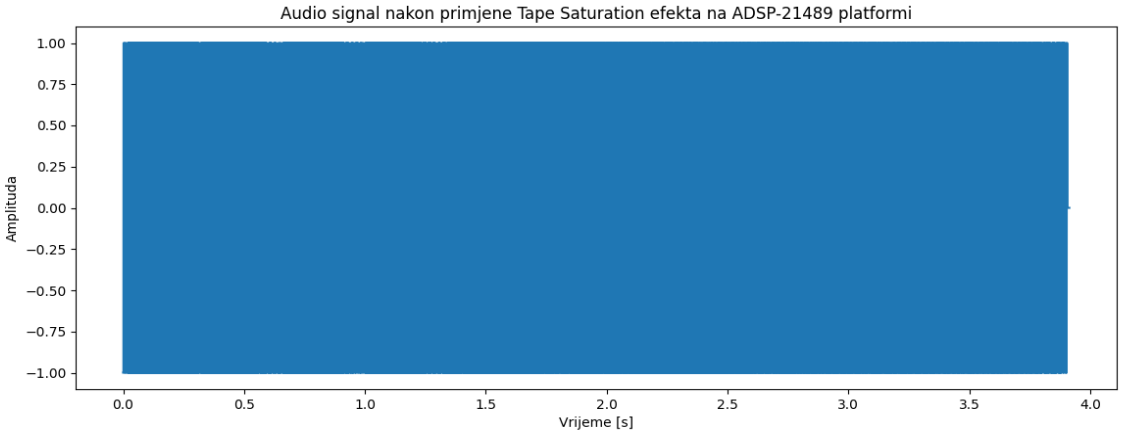
Slika 5.8 – Volume Pedal efekat implementiran unutar CCESa



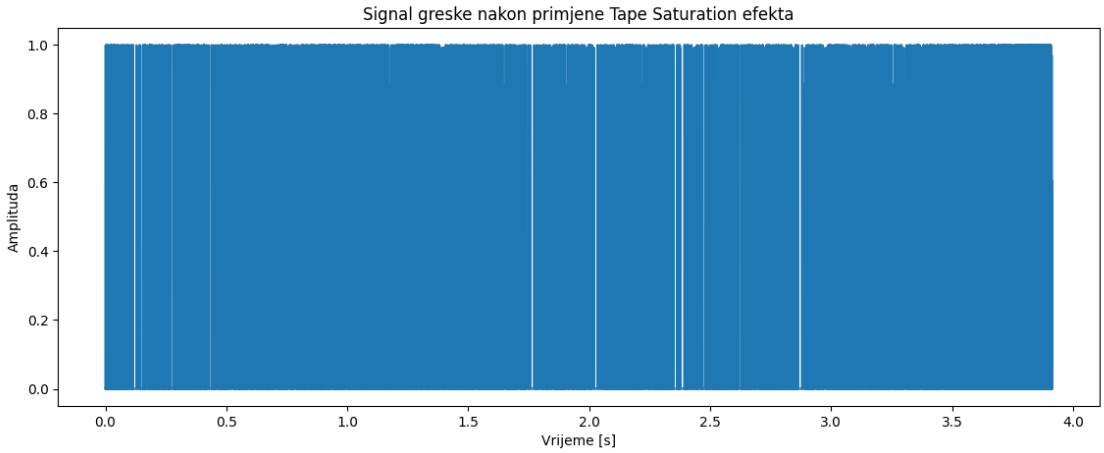
Slika 5.9 – Signal greške kod Volume Pedal efekta



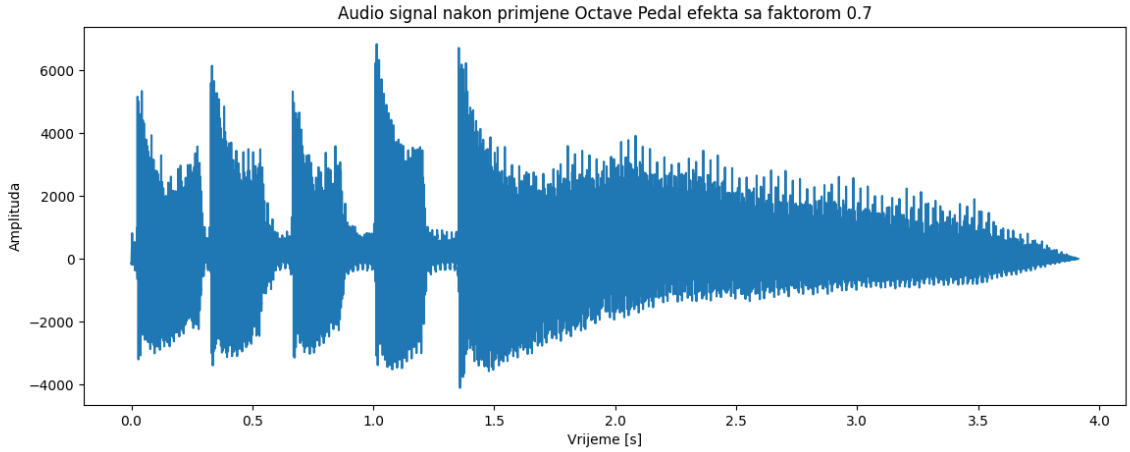
Slika 5.10 – Signal nakon primjene Tape Saturation efekta, generisan u Pythonu



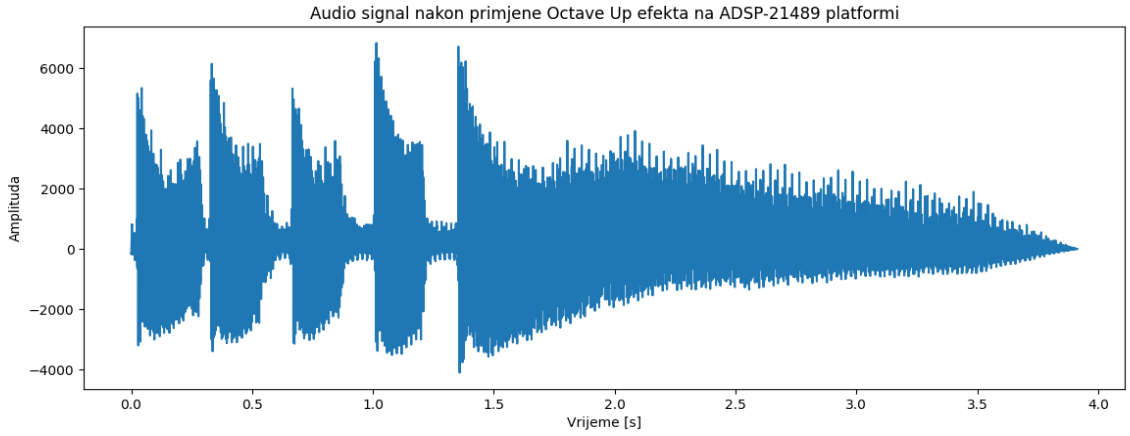
Slika 5.11 – Tape Saturation efekat, generisan unutar CCESa



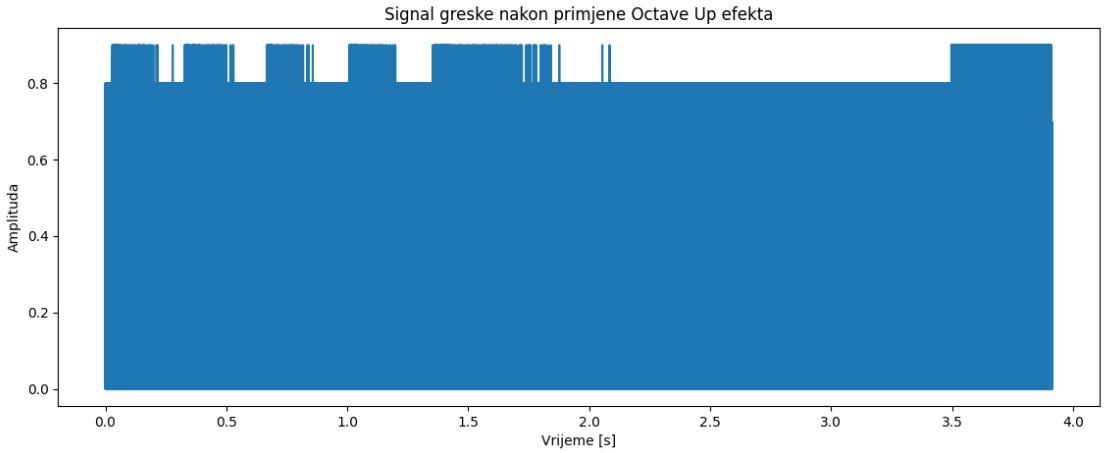
Slika 5.12 – Signal greške kod Tape Saturation efekta



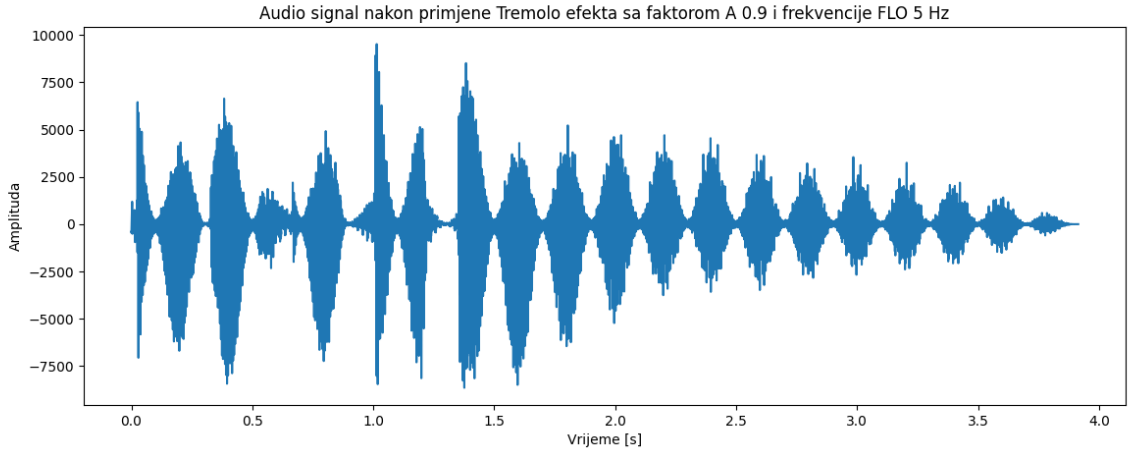
Slika 5.13 –Signal nakon primjene Octave Up efekta, generisan u Pythonu



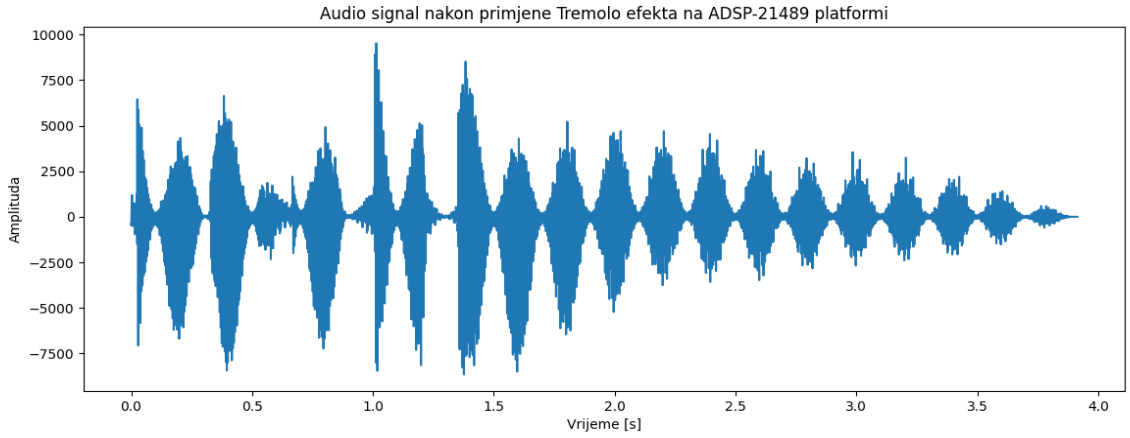
Slika 5.14 – Signal nakon primjene Octave Up efekta, generisan u CCESu



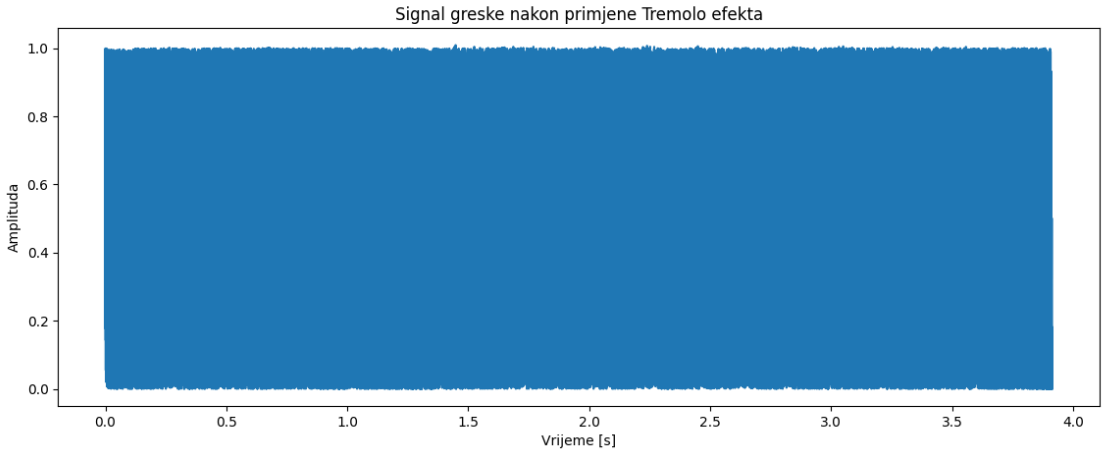
Slika 5.15 – Signal greške kod Octave Up efekta



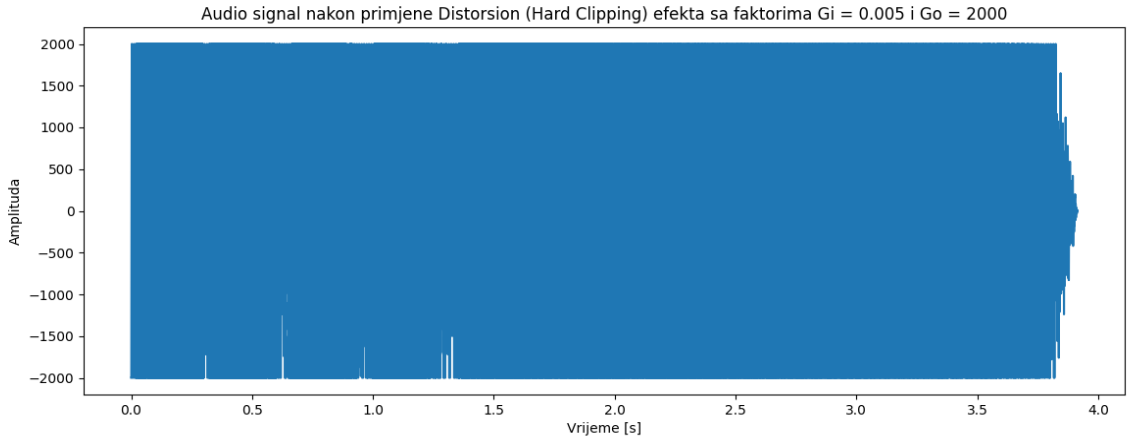
Slika 5.16 – Signal nakon primjene Tremolo efekta, generisan u Pythonu



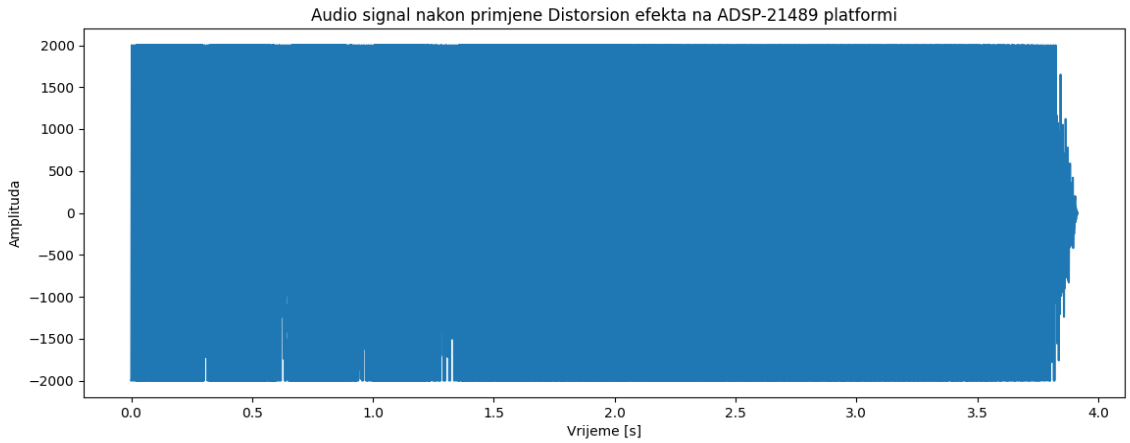
Slika 5.17 – Signal nakon primjene Tremolo efekta, generisan u CCESu



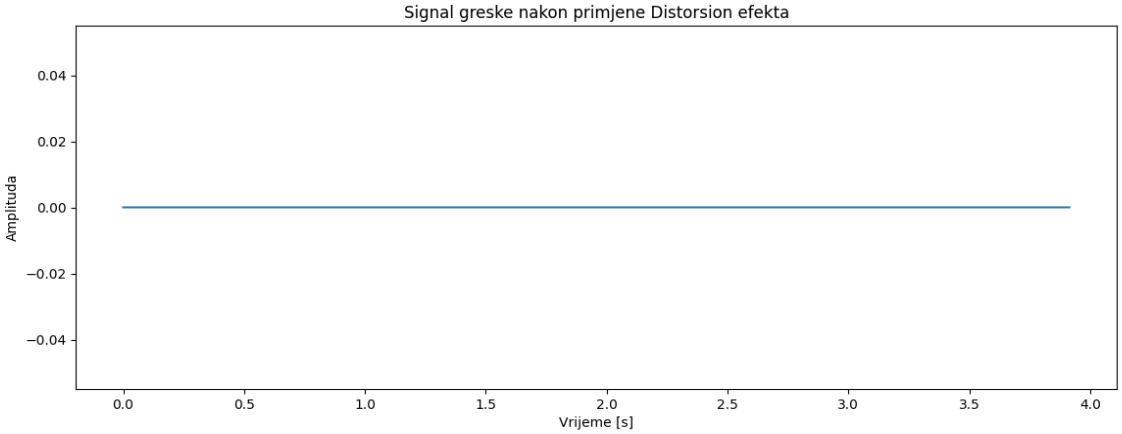
Slika 5.18 – Signal greške kod Tremolo efekta



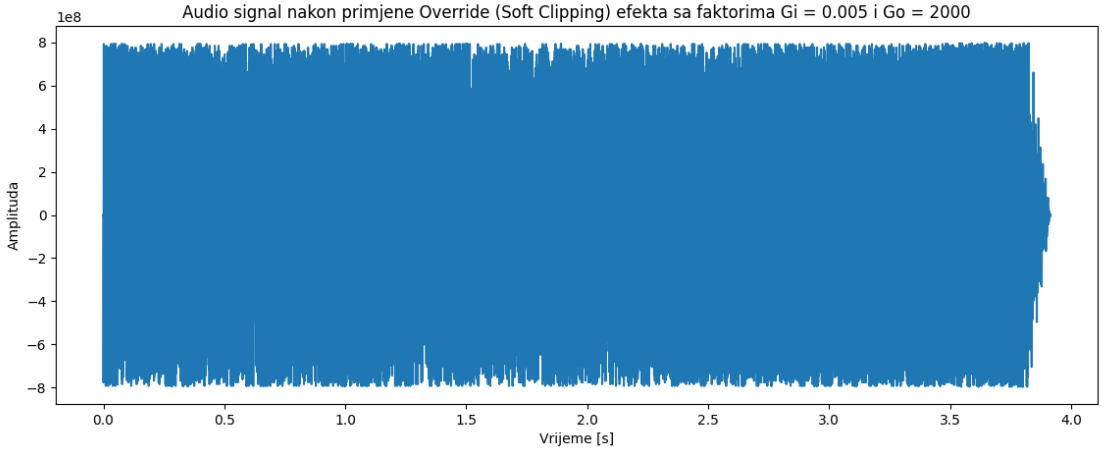
Slika 5.19 – Signal nakon primjene Distorsion efekta, generisan u Pythonu



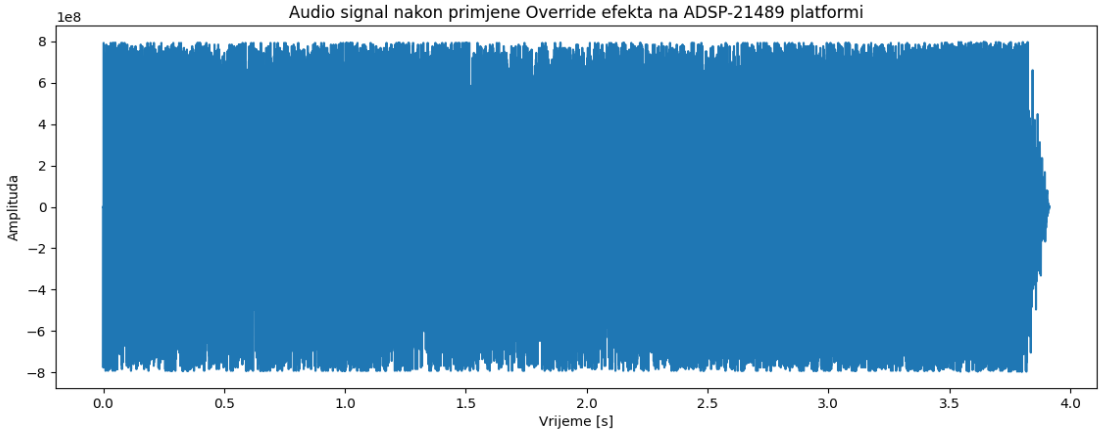
Slika 5.20 – Signal nakon primjene Distorsion efekta, generisan u CCESu



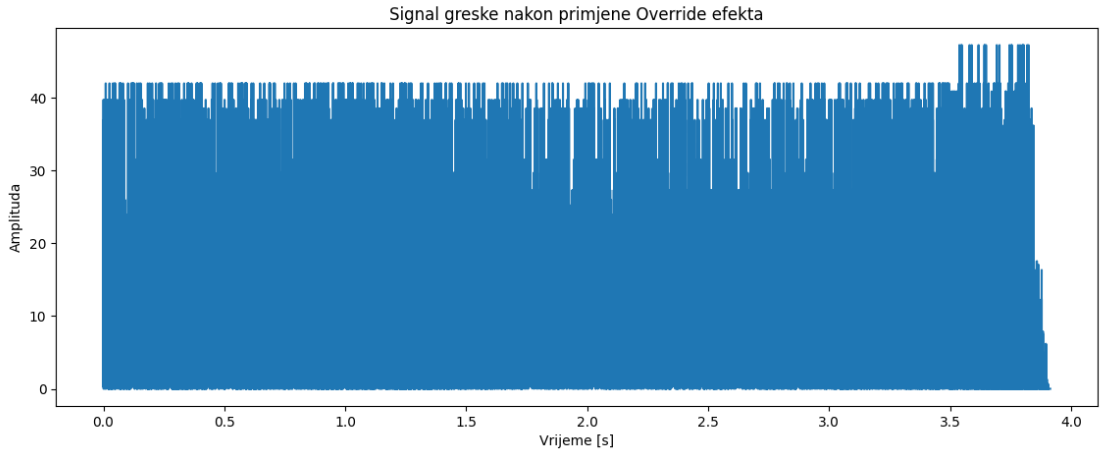
Slika 5.21 – Signal greške kod Distorsion efekta



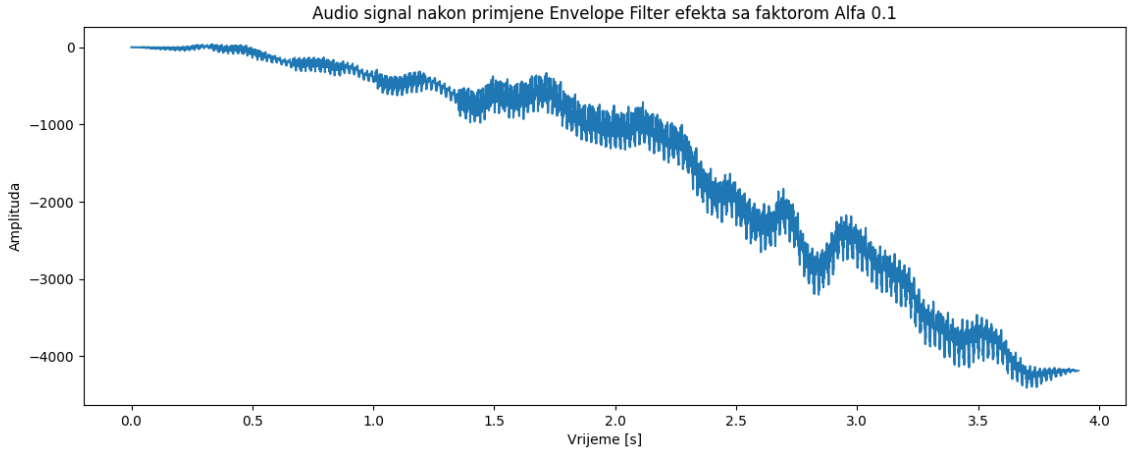
Slika 5.22 – Signal nakon primjene Override efekta, generisan u Pythonu



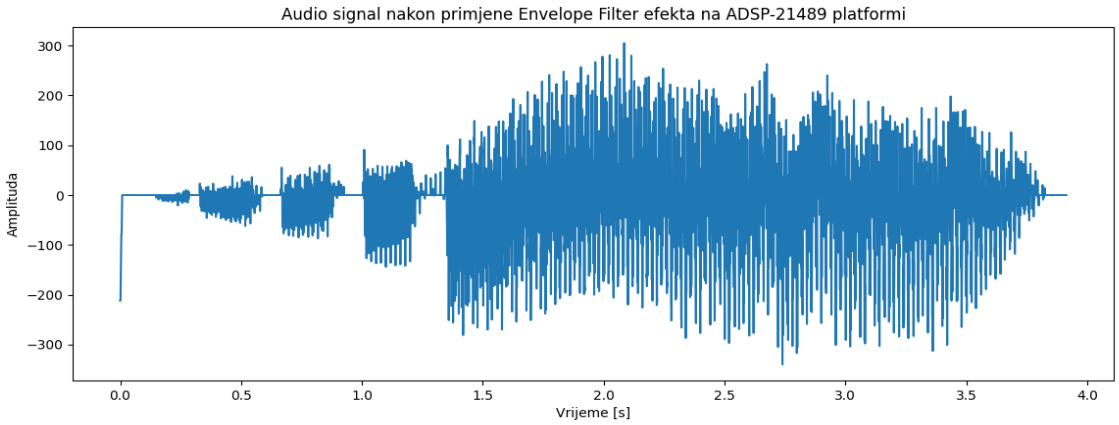
Slika 5.23 – Signal nakon primjene Override efekta, generisan u CCESu



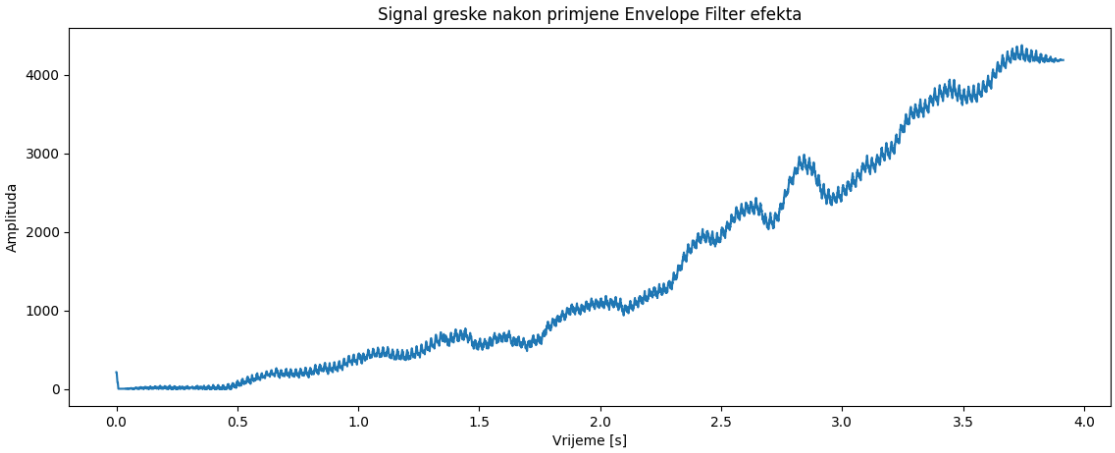
Slika 5.24 – Signal greške kod Override efekta



Slika 5.25 – Signal nakon primjene Envelope filtra, generisan u Pythonu



Slika 5.26 – Signal nakon primjene Envelope filtra, generisan unutar CCESa



Slika 5.27 – Signal greške kod Envelope filtra