
VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
MATEMATIKOS IR MATEMATIKOS TAIKYMŲ STUDIJŲ PROGRAMA

# Garso signalų apdorojimas dažnių srityje

Praktinės užduoties nr. 5 ataskaita

Atliko: Ugnius Alekna
VU el. p.: ugnius.alekna@mif.stud.vu.lt

Vertino: Doc. dr. Gintautas Tamulevičius

Vilnius
2023

# 1. ĮVADAS

Garso signalų apdorojimas dažnių srityje yra procesas, kuris leidžia analizuoti ir modifikuoti signalus pagal jų dažnines komponentes. Apdorojant signalus dažnių srityje galima efektyviai atskirti ar pašalinti nepageidaujamus triukšmus, išryškinti reikalingus signalo aspektus ar kitaip modifikuoti signalo struktūrą. Šis procesas suteikia galimybę atlikti sudėtingą signalų analizę, kuri laiko srityje būtų sunkiai įmanoma ar neįgyvendinama.

## 1.1. Darbo tikslas

Šiame darbe nagrinėjamos trumpalaikės (200 ms) garso signalų atkarpos dažnių srityje, atliekama jų spektrinė analizė ir apdorojimas. Apdorojimo metu siekiama pridėti, sustiprinti ar pašalinti konkrečias dažnio dedamąsias, perstumti dedamąsias dažnio didėjimo (mažėjimo) kryptimi bei pašalinti triukšmingus intervalus.

## 1.2. Darbo priemonės

Darbas atliekamas Python programavimo kalba. Matematinėms operacijoms naudojama biblioteka *NumPy*, diagramų vaizdavimui pasitelkta *MatPlotLib* biblioteka, signalų nuskaitymui – *SciPy* ir *Wave* bibliotekos. Grafinė vartotojo sąsaja kuriama naudojant biblioteką *tkinter*. Darbe naudojami įvairios trukmės ir kilmės garso įrašai, išsaugoti PCM formatu **.wav* tipo failuose.

# 2. APDOROJIMAS DAŽNIŲ SRITYJE

Apdorojant signalus dažnių srityje, Furjė transformacija yra pagrindinė matematinė priemonė, skirta tolydžius laiko signalus atvaizduoti tolydžiu dažnių spektru. Atvaizduota funkcija yra kompleksinė, ji suteikia informacijos apie signalą aprašančių bazinių funkcijų amplitudžių bei fazių priklausomybę nuo dažnio. Fazės duomenys yra sunkiai interpretuojami, todėl atliekant signalų apdorojimą, modifikuojamos tik amplitudės reikšmės, o fazė panaudojama rekonstruojant signalą atgal į laiko sritį. Skaitmeninių signalų apdorojimui naudojama diskrečioji Furjė transformacija (DFT). Ji transformuoja diskrečią laiko srities duomenų seką į atitinkamą diskrečią dažnių komponenčių seką. DFT skaičiavimo operacijų skaičiaus sudėtingumas yra $\mathcal{O}(n^2)$, tai riboja jos tiesioginį taikymą ilgiems signalams. Greitoji Furjė transformacija (GFT) yra algoritmas, optimizuojantis DFT skaičiavimus ir sumažinantis skaičiavimo sudėtingumą nuo $\mathcal{O}(n^2)$ iki $\mathcal{O}(n \log n)$. Jis mums leidžia atlikti efektyvų ir greitą signalų apdorojimą dažnių srityje.

## 2.1. Apdorojimo procesas

Garso signalų apdorojimo procesas susideda iš kelių etapų – signalo nuskaitymo, amplitudės spektro apskaičiavimo, signalo apdorojimo dažnių srityje bei signalo atstatymo atgal į laiko sritį. Toliau trumpai aptarsime kiekvieną iš šių etapų.

### 2.1.1. Signalo nuskaitymas

Pasirinktas garso signalas ir jo diskretizavimo dažnis nuskaitomi bibliotekos *SciPy* moduliu `scipy.io.wavfile`. Signalo amplitudės reikšmės normuojamos į reikšmes iš intervalo [-1, 1]. Pagal naudotojo įvestą pradžios laiko reikšmę, iš pasirinkto signalo išskiriama 200 ms trukmės atkarpa, naudojama tolimesniam apdorojimui. Prieš išskiriant atkarpą, vartotojui nurodoma signalo trukmė bei suteikiama galimybė išklausyti pilną garso signalą.

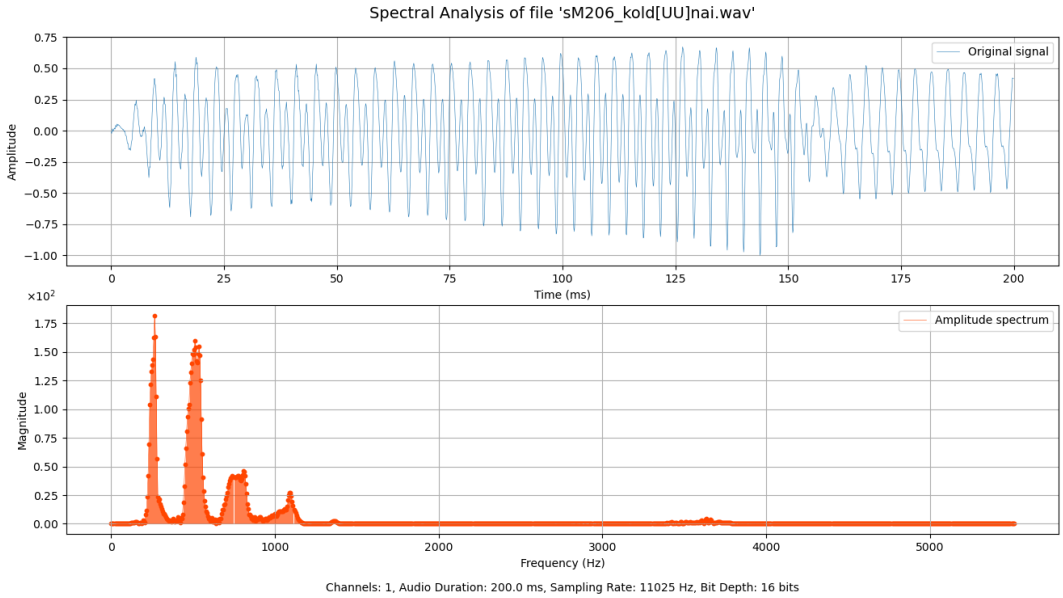
### 2.1.2. Amplitudės spektro apskaičiavimas

Gautoms signalo atkarpos reikšmėms pritaikoma lango funkcija, kuri palaipsniui sumažina signalo amplitudę atkarpos kraštuose, panaikindama trūkius signalo pradžioje bei pabaigoje. Šioje praktinėje užduotyje naudojama *hanning* lango funkcija, apibrėžiama formule

$$w(n) = 0.5 - 0.5\cos\left(\frac{2\pi n}{M-1}\right), \quad 0 \leq n \leq M-1,$$

kur *M* yra signalo reikšmių ilgis. Lango funkcija skaičiuojama naudojant *NumPy* bibliotekos funkciją `numpy.hanning()`. Signalo spektro reikšmės yra apskaičiuojamos panaudojant GFT algoritmą iš *NumPy* bibliotekos – `numpy.fft.fft()`. Signalo spektro reikšmės yra kompleksinės, todėl siekiant lengviau interpretuoti rezultatus, išskiriamas amplitudės spektras, panaudojant funkciją `numpy.abs()`, ir fazės spektras, panaudojant `numpy.angle()`. Furjė transformacija signalui, įgyjančiam tik realias reikšmes, sukuria simetrišką spektrą, todėl pakanka analizuoti tik pusę spektro reikšmių. Lyginio *M* ilgio sekos atveju paliekama (*M / 2 + 1*) pirmųjų reikšmių, nelyginio *M* atveju – (*(M + 1) / 2*) reikšmių. Šiai sutrumpintai Furjė transformacijos sekai reikia pritaikyti mastelio koeficientą, nes atmetus pusę spektro reikšmių dingsta informacija apie signalo energiją. Lyginio *M* atveju dvigubinamos reikšmės, pradedant nuo antros ir baigiant priešpaskutiniąja, nelyginio *M* atveju – dvigubinami elementai nuo antrojo iki paskutinio. Remiantis Naikvisto teorema, sugeneruojamos dažnio ašies reikšmės *[0, ..., Fs / 2]*, kur *Fs* –

signalo diskretizavimo dažnis. Vartotojui pateikiamos pasirinktos signalo atkarpos laiko ir apskaičiuoto amplitudės spektro diagramos. Pavyzdys pateikiamas 2.1 pav.




2.1 pav. Žmogaus šnekos laiko ir amplitudės spektro diagramos

### 2.1.3. Amplitudės spektro modifikavimas

Kai analizuojami realūs garso signalai, jų amplitudės spektras yra sutrumpinamas per pusę (kaip aprašyta 2.1.2 skyrelyje) siekiant pateikti paprastesnę vizualizaciją diagramose. Tačiau, norint modifikuoti amplitudės spektrą, tai galima atlikti dviem būdais: keisti neapdoroto amplitudės spektro reikšmes iš karto atlikus Furjė transformaciją, arba keisti reikšmes prieš tai jas sutrumpinant per pusę. Atsižvelgiant į tai, kad realaus signalo Furjė spektras visada yra simetriškas, modifikuojant nesutrumpintą spektrą labai svarbu išlaikyti šią simetriją. Paprastos operacijos, tokios kaip dažnio komponentų pridėjimas, sustiprinimas ar šalinimas sunkumų nesukelia – tereikia atlikti tą pačią operaciją du kartus, simetriškose amplitudės spektro vietose. Tačiau problemų kyla, kuomet atliekama dažnio komponenčių perstūmimo operacija. Perstūmus amplitudės spektro reikšmes, sugriaunama spektro reikšmių simetrija ir atstatytas signalas nebūna toks, kokio tikėtasi. Dėl to, prieš modifikuojant signalo amplitudės reikšmes, pasirinkta jas sutrumpinti ankščiau aprašytu būdu. Signalo amplitudės reikšmėms atliekami pakeitimai, aprašyti 2.2 skyrelyje. Norint atstatyti signalą, reikalingas pilnas amplitudės spektras, todėl prieš atliekant atvirkštinę Furjė transformaciją, reikia jį atkurti. Lyginio ilgio pradinio spektro sekos atveju, kiekviena dažnio komponentė, išskyrus nulinio dažnio ir Naikvisto (pusė diskretizavimo dažnio) komponentes, yra padalijama iš *2* ir atspindima simetriškai (Naikvisto komponentės atžvilgiu). Naikvisto komponentė nukopijuojama be pakeitimų, o nulinio dažnio komponentės kopijuoti nereikia. Nelyginio ilgio pradinio spektro atveju, dalijamos ir simetriškai atspindimos visos komponentės, išskyrus nulinį dažnį. Nulinio dažnio komponentė turi išlikti nepakitusi amplitudės spektro atkūrimo metu, kadangi ji nurodo vidutinę signalo vertę (pasvertą konstanta[1]). Taip amplitudės spektras yra atstatytas taisyklingai ir paruoštas atvirkštinei Furjė transformacijai.

[1] Tai tiesiogiai išplaukia iš DFT apibrėžimo. Jei DFT lygtis $X(k) = \sum_{n=0}^{M-1} x(n)e^{-2\pi ink/M}, 0 \leq k, n \leq M-1$, tai $X(0) = \sum_{n=0}^{M-1} x(n) = M \cdot \bar{x}, \ 0 \leq n \leq M-1$, kur $\bar{x}$ žymi vidutinę sekos $x(n), 0 \leq n \leq M-1$, reikšmę.

### 2.1.4. Signalo atstatymas

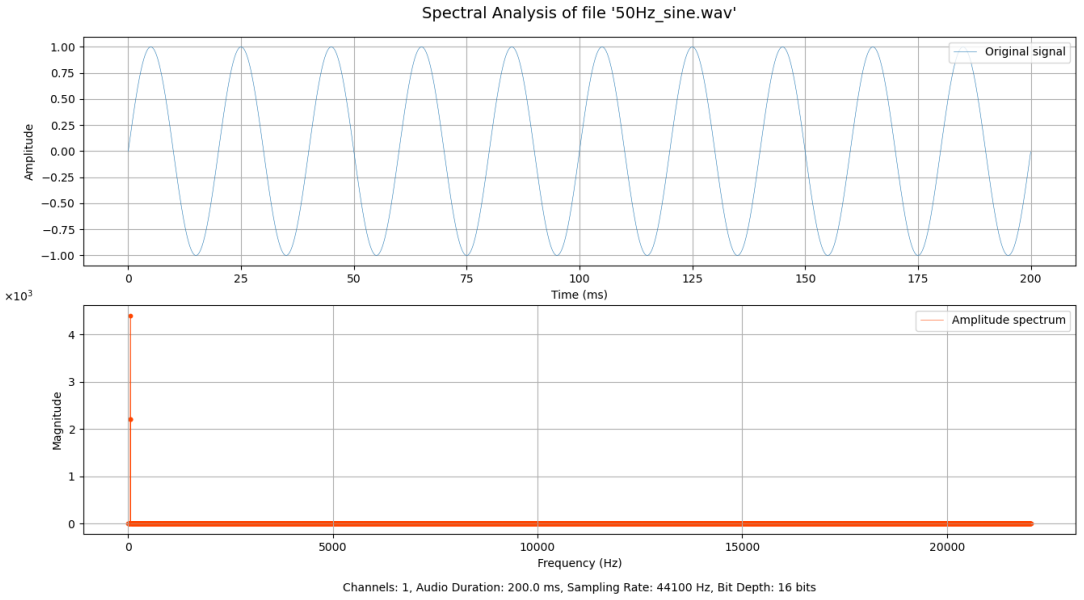
Signalas atstatomas panaudojant apdorotą amplitudės spektrą bei pradinio signalo fazės spektrą, kurie sujungiami į kompleksinę funkciją panaudojant polinę išraišką. Tada ši kompleksinė funkcija transformuojama į laiko sritį panaudojant atvirkštinę DFT su *NumPy* funkcija `numpy.fft.ifft()`. Gautas rezultatas yra apdorotas signalas, įgyjantis realias reikšmes, tačiau dėl *Python* slankiojo kablelio klaidos (floating-point error), kai kurios menamosios dalys tampa nenulinės, todėl jas reikia atmesti. Tam panaudojamas *NumPy* masyvo `.real` metodas, grąžinantis tik realias signalo reikšmes. Naudotojui pateikiamos apdoroto garso signalo laiko ir amplitudės spektro diagramos bei suteikiama galimybė išklausyti apdorotą garso signalo atkarpą.

## 2.2. Garso signalų apdorojimas

Garso signalo reikšmėms atlikus DFT, apskaičiavus amplitudės spektrą ir sutrumpinus spektro reikšmes, galime atlikti signalo apdorojimą dažnių srityje. Toliau apibūdinsime kelis nesudėtingus apdorojimo dažnių srityje metodus, pateiksime pavyzdžius bei subjektyviai įvertinsime matomus (girdimus) apdorotojo signalo garso pokyčius. Daugelyje pavyzdžių bus naudojami nesudėtingi dirbtiniai garso signalai, susidedantys iš vienos ar kelių dažnių komponenčių, kurie leis akivaizdžiai pastebėti skirtingų apdorojimo metodų veikimą.

### 2.2.1. Dažnio komponenčių pridėjimas

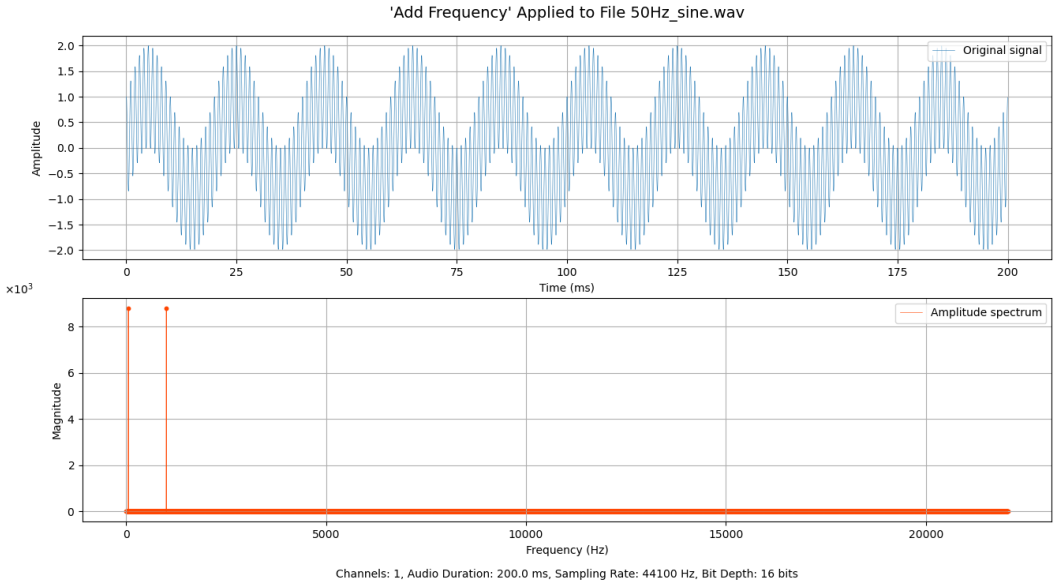
Signalo amplitudės spektro papildymas nauja dažnio komponente atliekamas pasirenkant dažnį $\tilde{f}$ bei jo stiprumą $\tilde{A}$, ir pridedant šią naują komponentę sutrumpintoje dažnių spektro sekoje. Jeigu $A(f)$ žymime signalo amplitudės spektrą, tai naujos dažnio komponentės pridėjimas gali būti aprašytas kaip $A(f) = A(f) + \tilde{A}$, kai $f = \tilde{f}$ ir $A(f) = A(f)$, kai $f \neq \tilde{f}$. Nagrinėkime 50 Hz dažnio garso signalą, pavaizduotą 2.2 pav. Signalo laiko diagramoje matome sinusoidę,




2.2 pav. 50 Hz dažnio sinusoidės garso signalas

atitinkančią periodą 1/50 s. Signalo amplitudės spektro diagramoje matomas vienas ryškus šuolis, žymintis atitinkamą 50 Hz dažnio komponentę. 2.3 pav. pateikiamos atitinkamos diagramos signalo, kuris buvo gautas prie pradinio 50 Hz signalo pridėjus vienodo stiprumo 1000 Hz dažnio

komponentę. Naujo signalo amplitudės spektro diagramoje matome pridėtą papildomą

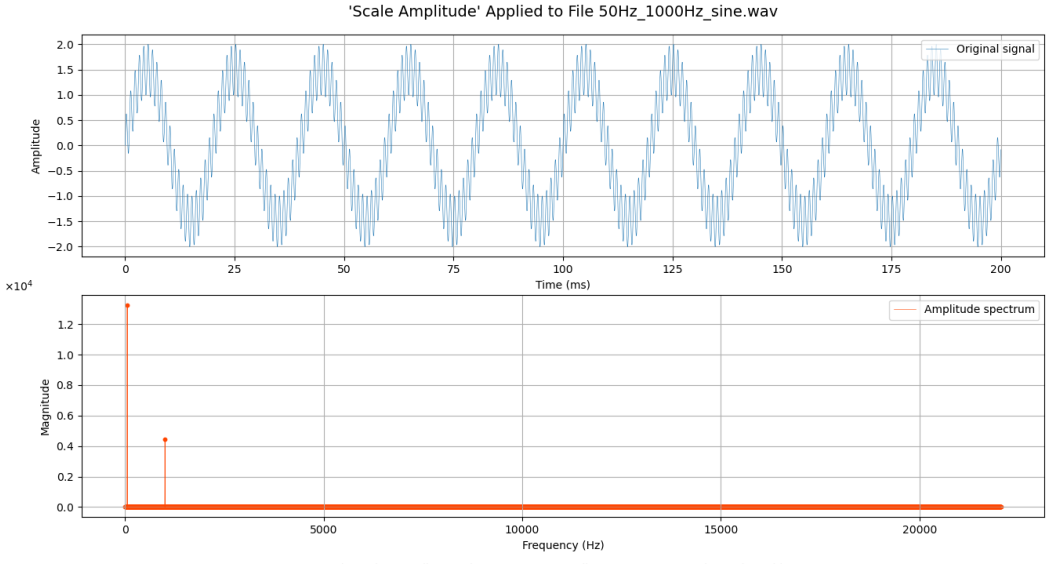



2.3 pav. 1000 Hz dažnio komponentė pridėta prie 50 Hz garso signalo.

šuolį, žymintį 1000 Hz. Signalo laiko diagramoje matoma aiški dviejų skirtingų sinusoidžių suma. Tai, jog buvo sudėti du vienodo stiprumo signalai, patvirtina 2 kartus didesnė apdoroto signalo amplitudė. Pradinis 50 Hz garso signalas girdimas kaip labai žemo dažnio tonas, sukeliantis vibravimo efektą, tačiau pridėjus 1000 Hz komponentę papildomai girdimas aukšto dažnio tonas, pasižymintis cypimu.

### 2.2.2. Dažnio komponenčių sustiprinimas

Konkretaus dažnio komponentės amplitudės sustiprinimas (susilpninimas) atliekamas pasirenkant dažnį $\tilde{f}$ ir pasveriant jį atitinkamu mastelio koeficientu $\alpha$. Jei $A(f)$ žymime signalo amplitudės spektrą, tai dažnio komponentės sustiprinimas (susilpninimas) gali būti aprašytas kaip $A(f) = \alpha \cdot A(f)$, kai $f = \tilde{f}$ ir $A(f) = A(f)$, kai $f \neq \tilde{f}$. Imkime garso signalą, gautą po apdorojimo 2.2.1 dalyje (prieš apdorojimą signalo amplitudės reikšmės normuojamos į reikšmes iš intervalo [-1,1]) ir sustiprinkime 50 Hz dažnio sinusoidės amplitudę. Rezultatas, kuomet pirmosios komponentės amplitudė padauginama iš 3, matomas 2.4 pav. Apdoroto signalo

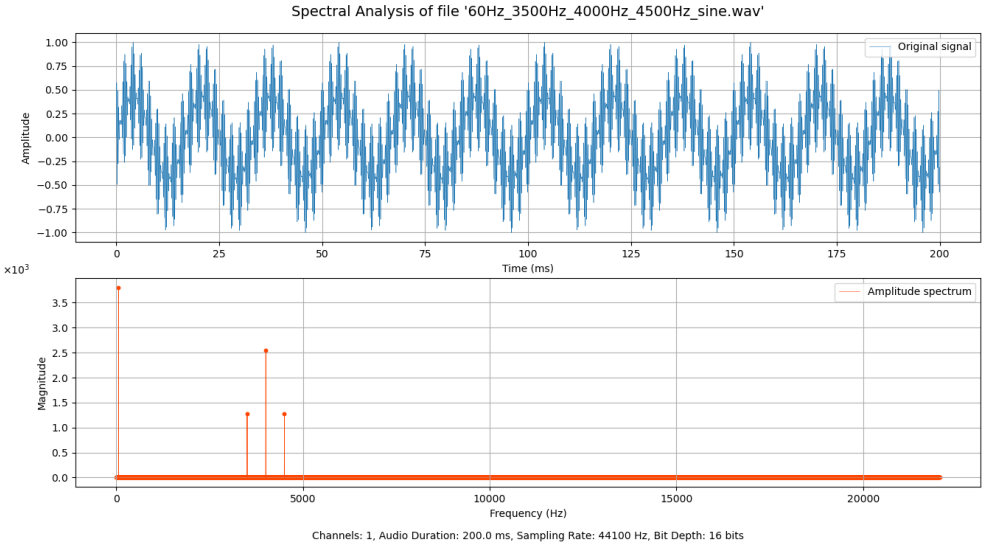



2.4 pav. 50 Hz dažnio komponentė sustiprinta 3 kartus lyginant su 1000 Hz dažnio komponente

amplitudės spektro diagramoje 50 Hz dažnį atitinkantis pikas tampa 3 kartus didesnis negu 1000 Hz dažnį atitinkantis pikas. Be to, signalo laiko diagramoje matoma išryškinta žemo dažnio sinusoidė. Pasvėrus žemo dažnio komponentę, 50 Hz tonas girdimas garsiau, žemo dažnio virpesiai yra stipresni, lyginant su pradiniu garso signalu.

### 2.2.3. Dažnio komponenčių pašalinimas

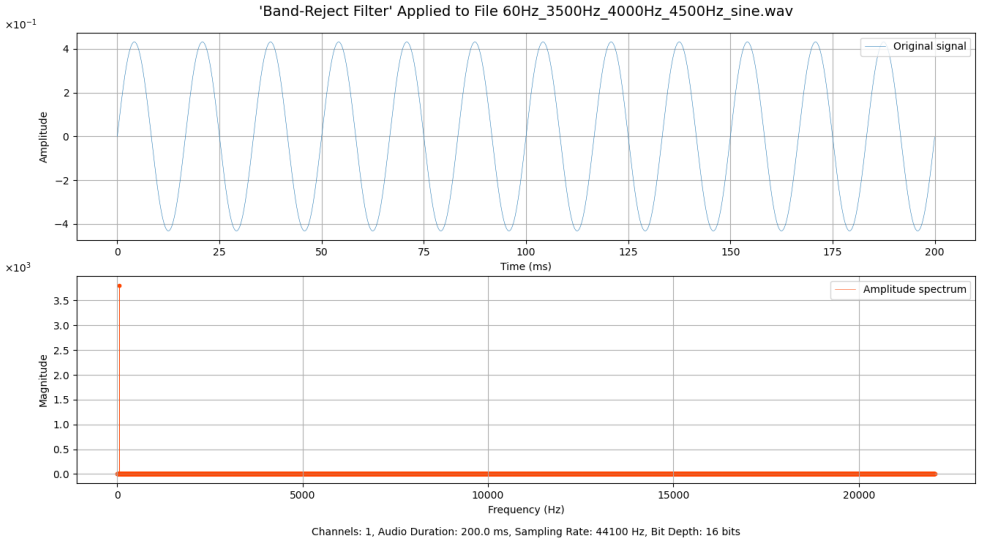
Pasirinkto dažnio komponentės pašalinimas gali būti atliekamas panaudojant 2.2.2 metodą, parenkant mastelio koeficientą $\alpha = 0$. Jeigu norime pašalinti ne vieną dažnio komponentę, o tam tikrą ruožą, galime pasinaudoti juostiniu filtru, kuris pasirinktai dažnių atkarpai $[f_{min}, f_{max}]$ atitinkamas amplitudės reikšmes $A(f_{\min}), \dots, A(f_{max})$ prilygina nuliui. Šiame darbe juostinis filtras realizuotas maskavimo (mask) pagalba. Jei $M(f)$ pažymime maskavimo funkciją, lygią $M(f) = 1$, kai $f \notin [f_{min}, f_{max}]$ ir $M(f) = 0$, kai $f \in [f_{min}, f_{max}]$, tai dažnio komponenčių pašalinimas gali būti aprašytas kaip $A(f) = M(f) \cdot A(f)$. Nagrinėkime garso signalą, sudarytą iš keturių sinusoidžių, pasvertų skirtingomis amplitudžių reikšmėmis, sumos, pavaizduotą 2.5 pav.




2.5 pav. 4 skirtingų sinusoidžių sumos laiko ir amplitudės spektro diagramos

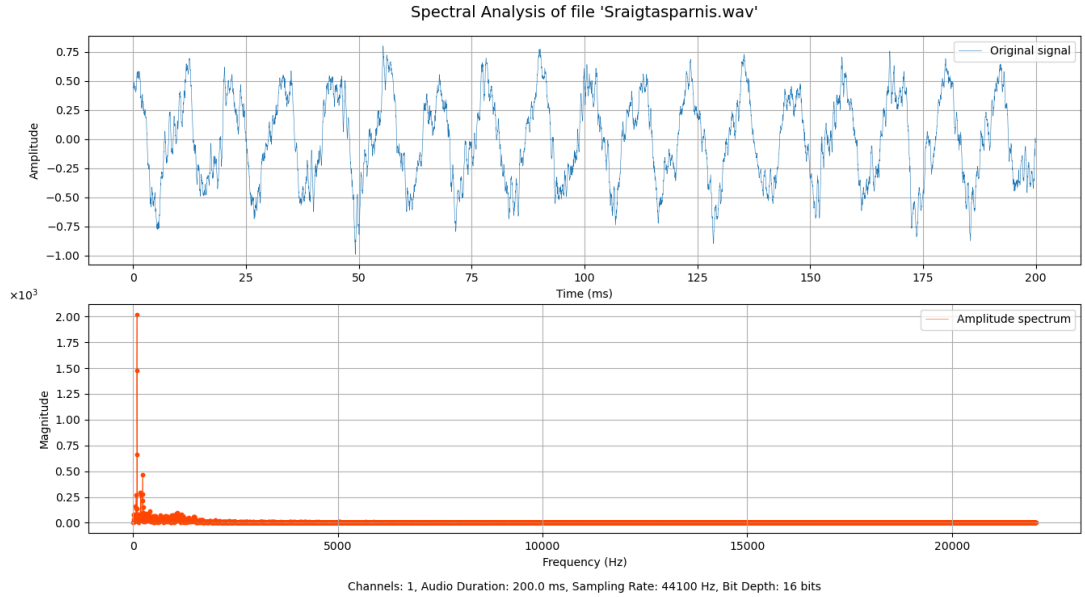
Pašalinkime iš signalo 3 dažnines komponentes – 3500 Hz, 4000 Hz ir 4500 Hz – pasirinkę atitinkamo pločio juostą. Apdorotą garso signalą turėtų sudaryti viena 60 Hz sinusoidė, nes aukšto

dažnio komponentės atmetamos. Rezultatas pateikiamas 2.6 pav. Gautame garso signale




2.6 pav. Signalo su pašalintomis dažninėmis komponentėmis laiko ir amplitudės spektro diagramos

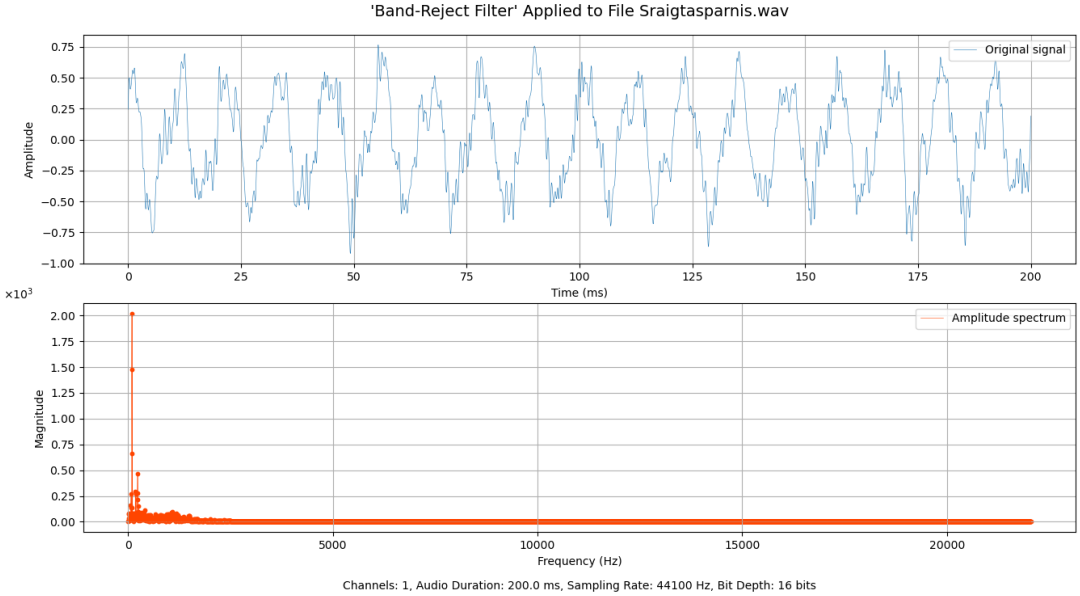
nebegirdimi aukšto dažnio cypimo efektą sukeliantys tonai, lieka vienas žemo tono garsas. Juostinis filtras gali būti tikslingai panaudotas, siekiant suglodinti signalą, panaikinant iš jo aukštadažnes komponentes. Pasirinkus $f_{max} = F_s/2$, kur $F_s$ – diskretizavimo dažnis, gauname filtrą, kuris „nepraleidžia“ visų komponenčių, aukštesnių negu $f_{min}$. Pavyzdžiui, nagrinėkime sraigtasparnio sukeliamo triukšmo garso signalą, pateiktą 2.7 pav. Pagrindinės signalo amplitudės




2.7 pav. Sraigtasparnio sukeliamo triukšmo signalo laiko ir amplitudės spektro diagramos

reikšmės sutelktos žemesnių dažnių intervale, todėl šiam signalui galime pritaikyti aukštų dažnių filtrą. Pasirinkę slenkstinę reikšmę 2500 Hz, nurodome, jog visos dažninės komponentės,

turinčios aukštesnį dažnį, būtų prilygintos 0. Apdorotas garso signalas pateikiamas 2.8 pav.



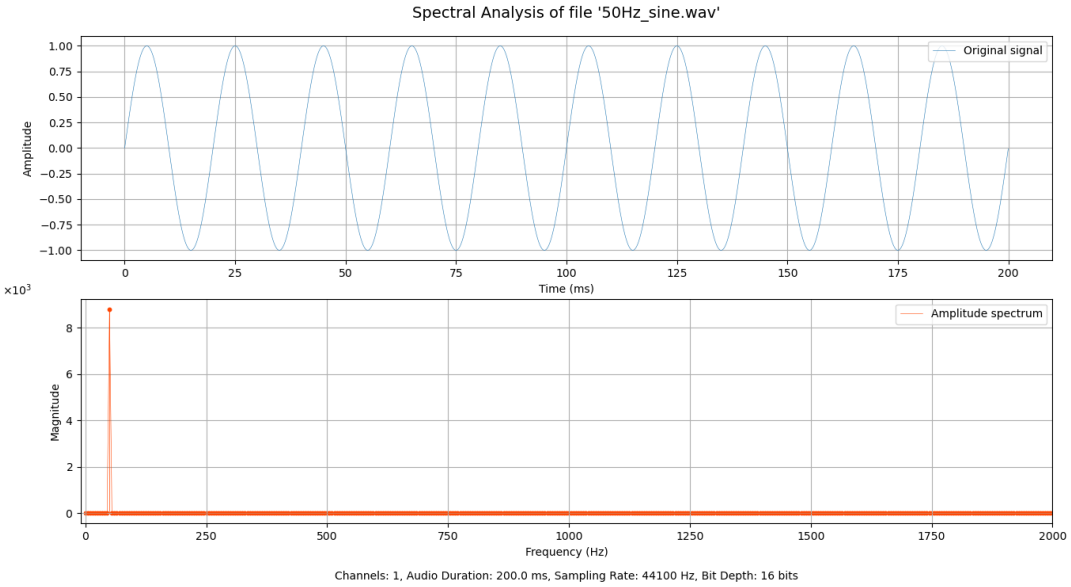

2.8 pav. Apdoroto sraigtasparnio sukeliamo triukšmo signalo laiko ir amplitudės spektro diagramos

Galime pastebėti, jog laiko diagramoje nebėra aštrių aukšto dažnio svyravimų, diagrama atrodo glodesnė. Amplitudės spektro diagrama atrodo nepakitusi, kadangi pradinio garso signalo aukšto dažnio komponenčių amplitudės buvo labai mažos. Išklausius apdorotą garso signalą galima teigti, jog nuo pradinio jis skiriasi kardinaliai. Nors ir amplitudės spektro pakitimai nežymūs, tačiau, aukštadažnės komponentės sudarė reikšmingą garso signalo informacijos dalį, be kurios signalas skamba dusliai ir nebeprimena sraigtasparnio sukeliamo triukšmo. Toks filtras galėtų būti panaudotas siekiant išskirti specifines signalo sudedamąsias, nekreipiant daug dėmesio į apdoroto signalo kokybę.

### 2.2.4. Dažnio komponenčių perstūmimas

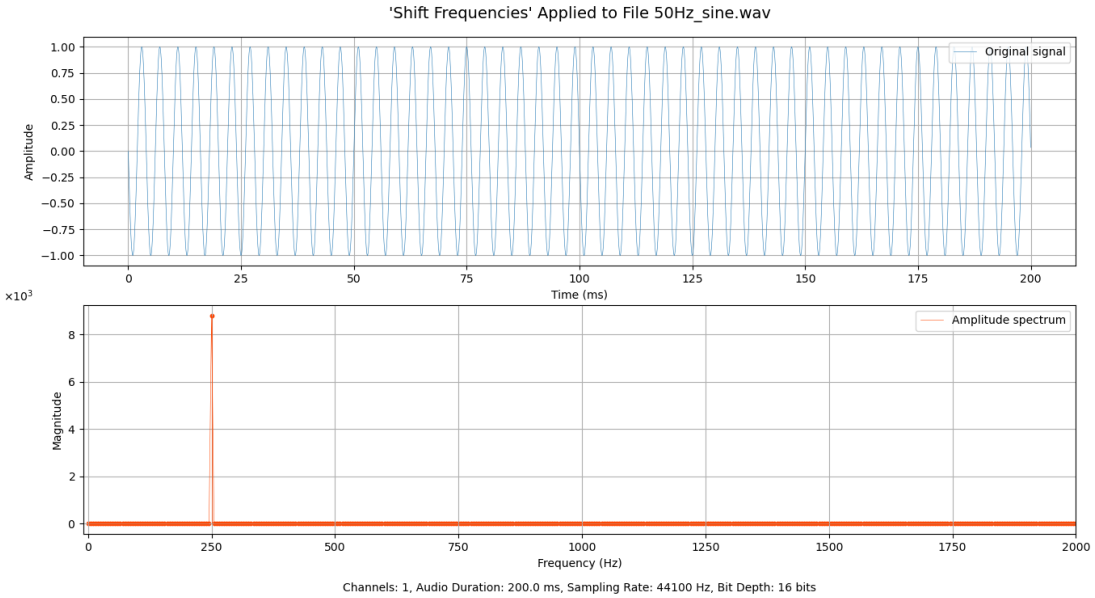
Dažnio komponenčių perstūmimas yra gaunamas paslinkus kiekvieną dedamąją dažnio didėjimo kryptimi. Jei $\Delta f$ pažymėsime poslinkį, $A'(f)$ – paslinktą amplitudės spektrą, tai sąryšį tarp pradinio ir paslinkto amplitudės spektro galime aprašyti kaip $A'(f) = A((f - \Delta f) \bmod M)$, kur $M$ – pusė diskretizavimo dažnio. Liekanos operacija mod užtikrina, kad amplitudės masyvo indeksas neįgytų neigiamų reikšmių ir ciklišk ai „grįžtų“ į masyvo ribas. Imkime 50 Hz dažnio

sinusoidę, matomą 2.9 pav. (amplitudės spektro diagrama priartinta), ir perstumkime šio signalo




2.9 pav. 50 Hz dažnio sinusoidės laiko ir amplitudės spektro diagramos

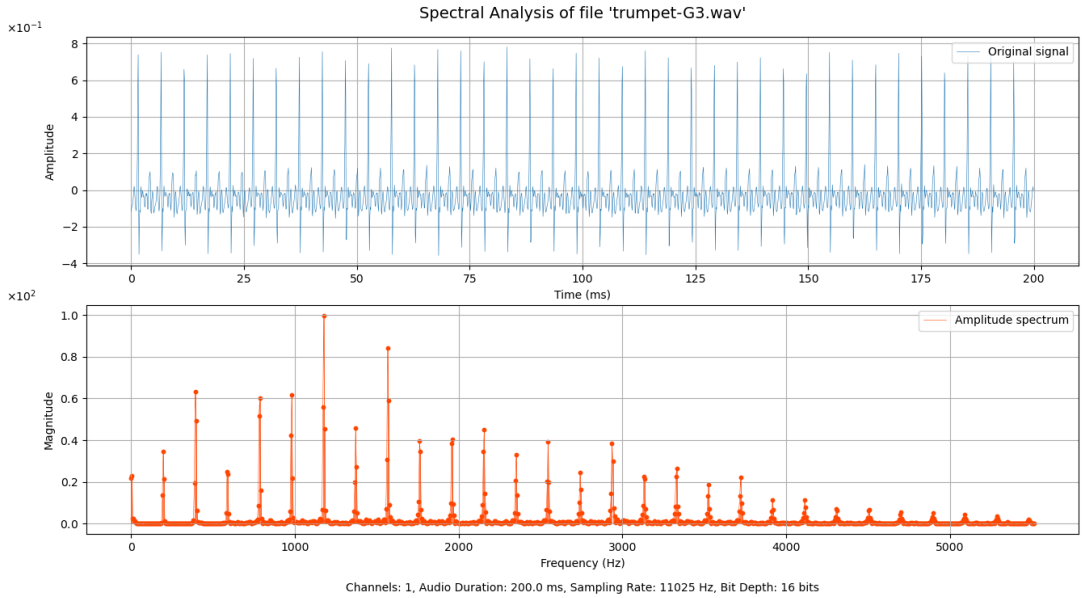
amplitudės spektro dedamąsias 200 Hz dažnio didėjimo kryptimi. Po paslinkimo, gautąjį signalą turėtų sudaryti viena 250 Hz dažnio sinusoidė. Rezultatas pavaizduotas 2.10 pav. Po apdorojimo,




2.10 pav. 50 Hz dažnio sinusoidės, paslinktos 200 Hz dažnio didėjimo kryptimi, laiko ir amplitudės spektro diagramos

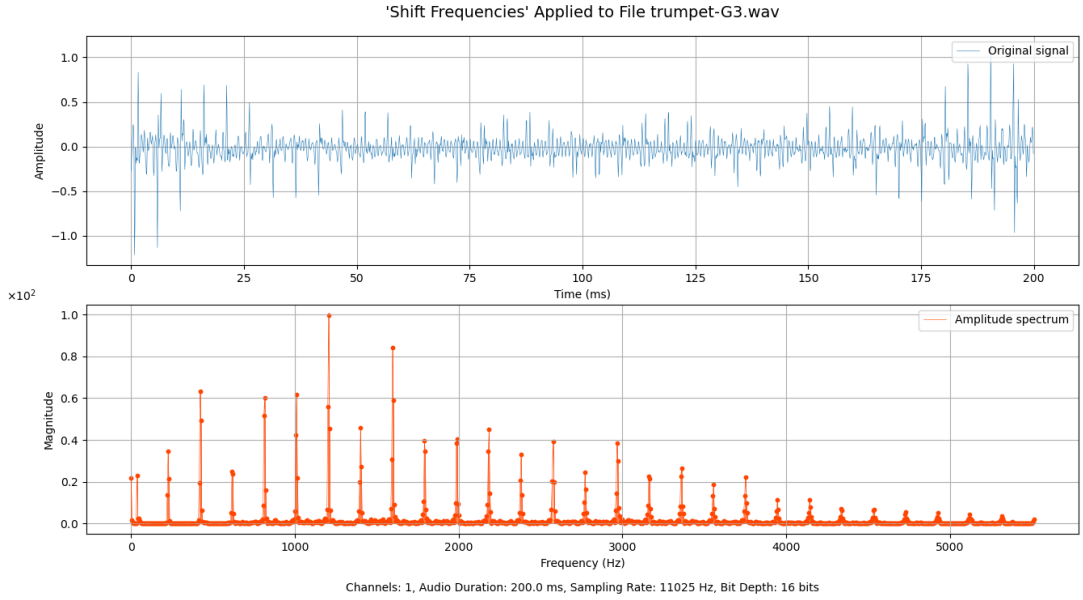
kaip ir tikėtasi, gautojo garso signalą sudaro viena aukštesnio dažnio sinusoidė, o jos paslinktas amplitudės spektras sudarytas iš vienos viršūnės, žyminčios 250 Hz dažnio dedamąją. Dažnių komponenčių perstūmimas gali būti įdomiai panaudotas, nagrinėjant įvairių instrumentų garso įrašus. Pavyzdžiui, nagrinėkime trimitu pučiamos G3 natos garso signalą. Jo laiko ir amplitudės

spektro diagramos pateikiamos 2.11 pav. Signalo amplitudės spektro diagramoje matomas




2.11 pav. Trimitu pučiamos G3 natos laiko ir amplitudės spektro diagramos

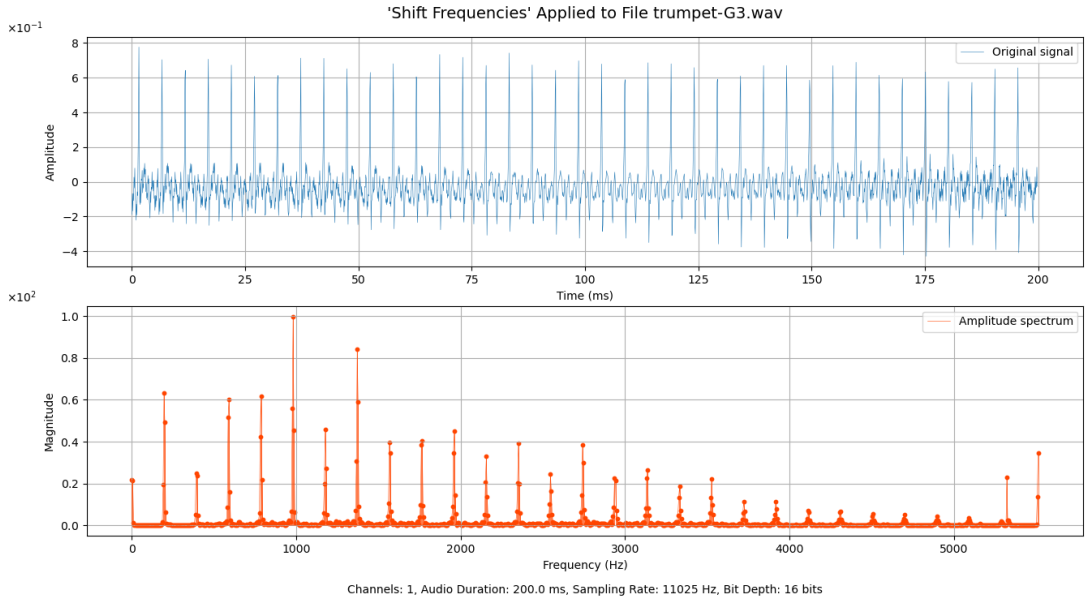
pagrindinis G3 natos dažnis lygus 195 Hz ir jo harmonikos. Paslinkus šio signalo amplitudės reikšmes 30 Hz į dešinę, pagrindinis dažnis persikelia į 225 Hz, o visos harmonikos taip pat pasislenka dešinėn. Natos garso signalas su paslinktu amplitudės spektru vaizduojamas 2.12 pav.




2.12 pav. Trimitu pučiamos G3 natos laiko ir paslinktos 30 Hz amplitudės spektro diagramos

Kadangi padidėja pagrindinis natos dažnis, nata skamba aukščiau. Paslinkus pradinio signalo harmonikas, pasikeičia signalo struktūra, kadangi šios paslinktos harmonikos nebėra pagrindinio dažnio (225 Hz) kartotiniai. Tai lemia nenatūraliai skambantį tembrą, dirbtinai skambantį garsą. Signalo laiko diagrama taip pat simbolizuoja nenatūraliai skambantį, neperiodišką garsą. Jeigu pradinio signalo amplitudės spektro reikšmes paslinksime per pagrindinį G3 natos dažnį, t.y. 195 Hz į kairę, turėtume gauti tos pačios natos, bet skirtingo tembro dažnių spektrą. Pagrindinis natos dažnis liks toks pats, pasikeis tik šio dažnio jo harmonikų amplitudės. Apdoroto garso signalo

laiko ir amplitudės spektro diagrama matoma 2.13 pav. Signalo harmonikos išlaiko



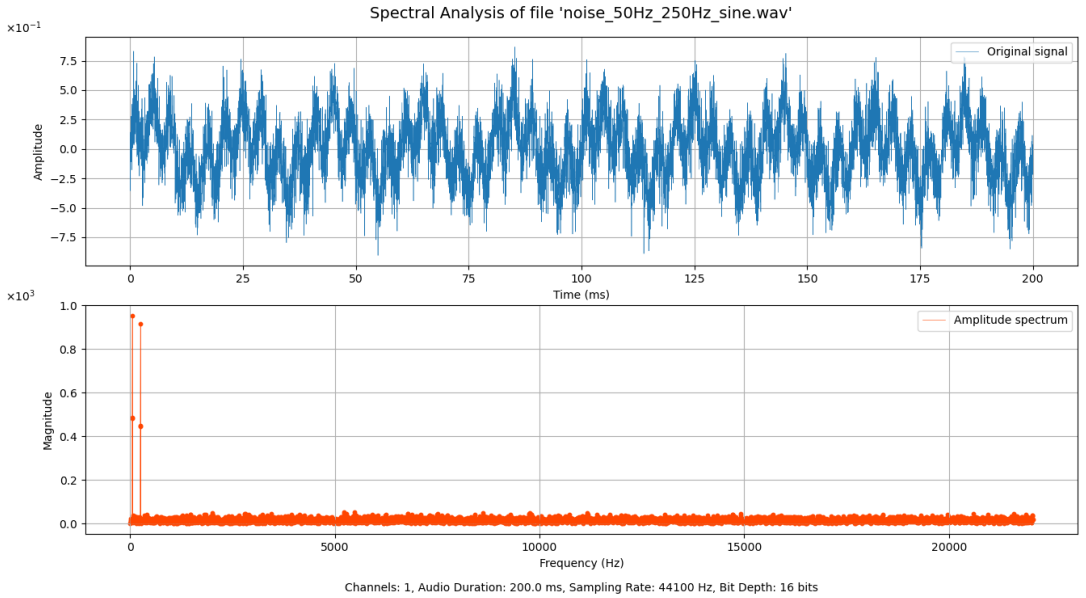

2.13 pav. Trimitu pučiamos G3 natos laiko ir paslinktos 195 Hz amplitudės spektro diagramos

taisyklingą pagrindinio dažnio kartotinumą. Signalo laiko diagrama atspindi periodiškumą, ji yra panaši į pradinio garso signalo laiko diagramą. Natos skambesys šiek tiek skiriasi nuo originalaus signalo, nes harmonikų amplitudės yra išsidėsčiusios kitaip, tačiau signalas skamba natūraliai. Nata turi skirtingą tembrą, tačiau kadangi pagrindinės dažninės komponentės išliko, skambesys turi trimito pūtimą primenantį garsą. Kadangi dažniai buvo paslinkti į kairę pusę, dvi harmonikos „apsivijo“ amplitudės spektrą, todėl garso įraše girdimas nežymus aukšto dažnio cypimas. Norint išgauti švarų garso signalą, šioms dažnių reikšmėms reikėtų pritaikyti 2.2.2 aprašytą komponenčių susilpninimo metodą.

### 2.2.5. Slenksčio metodas

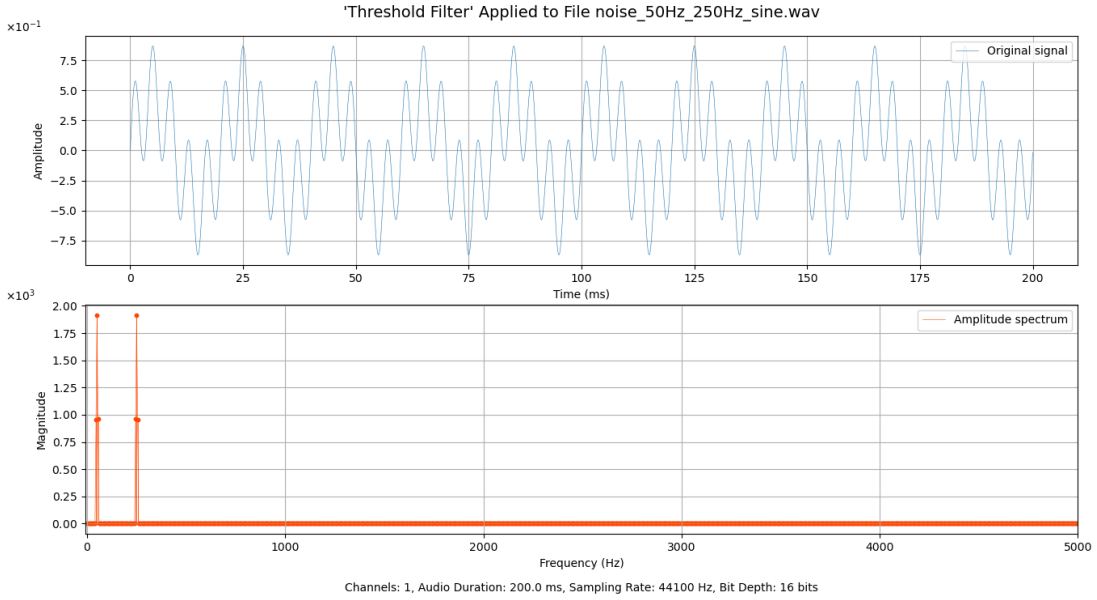
Tam tikros dažnio komponentės gali būti pašalinamos panaudojant slenksčio metodą. Šis metodas išskiria tik tas amplitudės spektro dalis, kurios viršija pasirinktą dažninių komponenčių amplitudės lygį $L$. Šiame darbe slenksčio metodas realizuotas maskavimo (mask) pagalba. Jei $M(f)$ pažymime maskavimo funkciją, lygią $M(f) = 1$, kai $A(f) \geq L$ ir $M(f) = 0$, kai $A(f) < L$, tai dažninis slenksčio metodas gali būti aprašytas kaip $A(f) = M(f) \cdot A(f)$. Šis metodas gali būti panaudotas efektyviai pašalinti nesudėtingą pridėtinį triukšmą. Nagrinėkime dirbtinį garso signalą, sudarytą iš dviejų sinusoidžių, kuris buvo sugadintas pridėjus atsitiktinį Gauso triukšmą,

pavaizduotą 2.14 pav. Signalo laiko diagramoje matoma daug triukšmingų svyravimų. Šie




2.14 pav. Triukšmingo dviejų sinusoidžių garso signalo laiko ir amplitudės spektro diagramos

svyravimai amplitudės spektro diagramoje vaizduojami kaip daugybė nedidelės amplitudės skirtingo dažnio dedamųjų. Pasirinkus atitinkamą slenksčio reikšmę, šias komponentes nesunku pašalinti. Apdorotas garso signalas pateiktas 2.15 pav. Garso signalo laiko diagramoje matoma




2.15 pav. Dviejų sinusoidžių garso signalo laiko ir amplitudės spektro diagramos pašalinus triukšmą

aiški dviejų sinusoidžių suma, „išvalytoje“ amplitudės spektro diagramoje matomi du pikai atitinkantys 50 Hz ir 250 Hz dažnių dedamąsias. Apdorotas garso signalas skamba švariai, nebegirdimas šnypštimo tonas. Nors pradinio garso signalo laiko diagrama atrodo stipriai paveikta triukšmo, tačiau kadangi triukšmas yra pridėtinis, jis yra visiškai pašalinamas apdorojant signalą dažnių srityje.

# 3. DARBO REZULTATŲ APIBENDRINIMAS IR IŠVADOS

Praktinės užduoties nr. 5 ataskaita apžvelgia garso signalų apdorojimo procesą dažnių srityje. Jis vykdomas modifikuojant signalų dažnines dedamąsias jų amplitudės spektre. Siekiant korektiškai atlikti signalų apdorojimą, pasirinkta keisti sutrumpinto amplitudės spektro reikšmes, o pilną amplitudės spektrą atstatyti remiantis Furjė spektro simetrija. Darbe buvo nagrinėjami penki apdorojimo metodai – dažninių komponenčių pridėjimas, pašalinimas, sustiprinimas (susilpninimas), dažnio srities perstūmimas ir slenksčio metodas. Analizė atskleidė:

- Į amplitudės spektrą pridėjus naują komponentę, pakinta signalo dažninis turinys bei garsinės savybės. Metodas gali būti naudojamas norint papildyti signalą naujomis dažnio charakteristikomis arba sukurti naujas garsines struktūras.
- Sustiprinant (susilpninant) tam tikrą dažnio komponentę, padidinama (sumažinama) jo amplitudė, koreguodama bendrą signalo garsą. Šis metodas gali būti panaudojamas reguliuojant tam tikrų dažnių garso lygį signale, pabrėžiant arba mažinant tam tikrus signalo aspektus.
- Šalinant pasirinktas dažnių komponentes, signale panaikinamos tam tikros nepageidaujamų dažnių sritys. Šis metodas yra naudingas triukšmų šalinimui iš signalo arba norint išryškinti svarbias signalo dalis, pašalinant mažiau reikšmingus dažnius.
- Perstumiant dažnio komponentes į kitą poziciją spektre, keičiama signalo harmoninė struktūra ir suvokiamas signalo aukštis. Metodas gali būti taikomas signalo tono ar aukščio korekcijai.
- Taikant slenksčio metodą, iš signalo amplitudės spektro pašalinami tie dažnio komponentai, kurių amplitudės neviršija nustatytos slenksčio vertės. Šis metodas gali būti panaudotas triukšmų mažinimui, paliekant tik svarbiausius signalo dažnius ir pašalinant mažesnės amplitudės komponentes, kurios dažniausiai siejamos su foniniais triukšmais ar nereikšmingais signalo pokyčiais.

Ši praktinė užduotis parodė dažnių srities apdorojimo veiksmingumą taikant skirtingus apdorojimo metodus. Ataskaitoje veiksmingai parodyta, kaip kiekvienas apdorojimo metodas yra pritaikomas, pasitelkiant dirbtinius bei tikrus garso signalus. Pateikiami aiškūs kiekvieno metodo poveikio signalo kokybei ir charakteristikoms pavyzdžiai ir rezultatai.

# 4. SPRENDIMO IŠEITIES KODAS

Praktinė užduotis buvo atlikta naudojant šį kodą. Visos ataskaitoje pateiktos diagramos buvo sugeneruotos manipuliuojant skirtingais nustatymais grafinės sąsajos lange.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
from scipy.io import wavfile
import wave
import tkinter as tk
from tkinter import filedialog, ttk
import winsound
import os

os.chdir(os.path.dirname(os.path.abspath(__file__)))

# Main GUI window
root = tk.Tk()
root.title('WAV File Reader')
root.resizable(False, False)
root.geometry('300x125')
def on_root_close():
    # Define root closing protocol
    plt.close('all')
    if os.path.exists('temp'):
        for filename in os.listdir('temp'):
            file_path = os.path.join('temp', filename)
            if os.path.isfile(file_path):
                os.remove(file_path)
        os.rmdir('temp')
    for window in root.winfo_children():
        if isinstance(window, tk.Toplevel):
            window.destroy()
    root.destroy()
root.protocol('WM_DELETE_WINDOW', on_root_close)

def open_file():
    # Dialog window to select wav files
    file_path = filedialog.askopenfilename(
        filetypes=(
            ("wav files", "*.wav"),
            ("all files", "*.*")))
    if file_path:
        # Folder for temp files
        if not os.path.exists('temp'):
            os.mkdir('temp')
        # Read WAV file and get attributes
        sample_rate, signal_data = wavfile.read(file_path, 'r')
        wave_object = wave.open(file_path, mode='r')
        num_of_channels = wave_object.getnchannels()
        bit_depth = wave_object.getsampwidth() * 8
```

```
        wave_object.close()
        file_name = os.path.basename(file_path)
        wav_info = [file_name, num_of_channels, bit_depth]
        signal_data = signal_data.reshape(-1, num_of_channels)
        signal_data = normalize(signal_data)
        # Ask the user for start time (end time is fixed to 200 ms)
        start_time, end_time, time_unit = ask_segment_time_input(file_path, sample_rate,
signal_data)
        signal_data = extract_segment(sample_rate, signal_data, start_time, end_time,
time_unit)
        # Apply window and truncate amplitude for plotting
        windowed_signal = apply_window(sample_rate, signal_data, num_of_channels,
window_function='hanning')
        truncated_amplitude = truncate_amplitude(windowed_signal)
        # Plot signal and its amplitude spectrum
        plot_signal(sample_rate, signal_data, truncated_amplitude, wav_info,
title=f"Spectral Analysis of file '{file_name}'")
        # Processing options window
        process_signal_options(sample_rate, signal_data, wav_info)

def play_audio(file_path):
    winsound.PlaySound(file_path, winsound.SND_ASYNC + winsound.SND_LOOP)

def pause_audio():
    winsound.PlaySound(None, 0)

def write_file(sample_rate, signal_data, output_filename):
    # Write the NumPy array as a WAV file
    output_file = 'temp\\' + output_filename + '.wav'
    wavfile.write(output_file, sample_rate, signal_data.astype(np.float32))
    # Get full path to the written temporary audio file
    script_path = os.path.dirname(os.path.abspath(__file__))
    file_path = script_path + '\\' + output_file
    return file_path

def get_time_scale(duration_sec):
    # Determine the appropriate time scale based on duration
    if duration_sec < 1:
        time_scale = 1000
        xlabel = 'ms'
    elif duration_sec < 60:
        time_scale = 1
        xlabel = 's'
    elif duration_sec < 60**2:
        time_scale = 1/60
        xlabel = 'min'
    else:
        time_scale = 1/60**2
        xlabel = 'h'
    return time_scale, xlabel
```

```
def get_duration_sec(sample_rate, signal_data):
    # Calculate signal duration in seconds
    return signal_data.shape[0] / sample_rate

def normalize(signal_data):
    # Return signal amplitude values normalized to [-1, 1]
    return signal_data/np.max(np.abs(signal_data))

def get_time_axis(sample_rate, signal_data, time_scale):
    # Calculate time linspace of the given signal
    return np.arange(0, signal_data.shape[0]) / sample_rate * time_scale

def extract_segment(sample_rate, signal_data, start_time, end_time, time_unit):
    # Slice signal array to extract only a section of it
    start_index = np.round(convert_to_seconds(start_time, time_unit) *
sample_rate).astype(int)
    end_index = np.round(convert_to_seconds(end_time, time_unit) * sample_rate).astype(int)
    segment = signal_data[start_index:end_index, :]
    return segment

def convert_to_seconds(numeric_value, time_unit):
    # Convert user input to seconds for further use
    match time_unit:
        case 'ms':
            return numeric_value / 1000
        case 's':
            return numeric_value
        case 'min':
            return numeric_value * 60
        case 'h':
            return numeric_value * 60**2

def apply_window(sample_rate, signal_data, num_channels, window_function):
    # Obtain time interval values
    duration = get_duration_sec(sample_rate, signal_data)
    time_scale, _ = get_time_scale(duration)
    time = get_time_axis(sample_rate, signal_data, time_scale)
    # Get values of window function in time interval
    match window_function:
        case 'hamming':
            window_function = np.hamming(len(time))
        case 'hanning':
            window_function = np.hanning(len(time))
        case 'blackman':
            window_function = np.blackman(len(time))
        case 'parzen':
            window_function = np.parzen(len(time))
        case 'bartlett':
            window_function = np.bartlett(len(time))
    windowed_audio = np.zeros_like(signal_data)
    for i in range(num_channels):
```

```
        windowed_audio[:, i] = window_function * signal_data[:, i]
    return windowed_audio

def get_freq_axis(sample_rate, signal_amplitude):
    # Calculate linspace of frequency values
    return np.linspace(0, sample_rate/2, signal_amplitude.shape[0])

def truncate_amplitude(signal_data):
    amplitude, _ = apply_fft(signal_data)
    num_channels = amplitude.shape[1]
    N = amplitude.shape[0]
    # Truncate half of the symmetric amplitude spectrum
    truncated = np.zeros((N//2 + 1 if N % 2 == 0 else (N+1)//2, num_channels))
    for i in range(num_channels):
        if N % 2 == 0:
            truncated_channel = amplitude[:N//2 + 1, i].copy()
            truncated_channel[1:-1] *= 2
        else:
            truncated_channel = amplitude[:(N+1)//2, i].copy()
            truncated_channel[1:] *= 2
        truncated[:, i] = truncated_channel
    return truncated

def reconstruct_amplitude(truncated, N):
    # Reconstruct the symmetry of the amplitude spectrum
    num_channels = truncated.shape[1]
    reconstructed = np.zeros((N, num_channels))
    for i in range(num_channels):
        if N % 2 == 0:
            reconstructed[1:N//2, i] = truncated[1:-1, i] / 2
            reconstructed[N//2, i] = truncated[-1, i]
            reconstructed[N//2 + 1:, i] = truncated[1:-1, i][::-1] / 2
        else:
            reconstructed[1:(N+1)//2, i] = truncated[1:, i] / 2
            reconstructed[(N+1)//2:, i] = truncated[1:, i][::-1] / 2
        reconstructed[0, i] = truncated[0, i]  # DC component
    return reconstructed

def modify_amplitude(sample_rate, signal_data, method,
                     cutoff_lower=None, cutoff_upper=None, threshold=None, freq_to_add=None,
                     amplitude_to_add=None, freq_to_scale=None, scale_factor=None,
shift=None):
    # Apply frequency domain processing
    match method:
        case 'Band-Reject Filter':
            modified_signal = band_reject_filter(sample_rate, signal_data, cutoff_lower,
cutoff_upper)
        case 'Threshold Filter':
            modified_signal = threshold_filter(sample_rate, signal_data, threshold)
        case 'Add Frequency':
```

```
            modified_signal = add_frequency(sample_rate, signal_data, freq_to_add,
amplitude_to_add)
        case 'Scale Amplitude':
            modified_signal = scale_amplitude(sample_rate, signal_data, freq_to_scale,
scale_factor)
        case 'Shift Frequencies':
            modified_signal = shift_frequencies(sample_rate, signal_data, shift)
    return modified_signal

def apply_fft(signal_data):
    # Extract amplitude and phase spectra
    signal_fft = np.fft.fft(signal_data, axis=0)
    amplitude = np.abs(signal_fft)
    phase = np.angle(signal_fft)
    return amplitude, phase

def inverse_fft(amplitude, phase):
    # Reconstruct signal from amplitude and phase spectra
    signal_fft = amplitude * np.exp(1j * phase)
    signal_data = np.fft.ifft(signal_fft, axis=0)
    return signal_data.real

def band_reject_filter(sample_rate, signal_data, cutoff_lower, cutoff_upper):
    amplitude, phase = apply_fft(signal_data)
    truncated_amplitude = truncate_amplitude(signal_data)
    freq_axis = get_freq_axis(sample_rate, truncated_amplitude)
    filtered_amplitude = np.copy(truncated_amplitude)
    # Apply band-reject filter
    if cutoff_lower is not None and cutoff_upper is not None:
        mask = (freq_axis >= cutoff_lower) & (freq_axis <= cutoff_upper)
    elif cutoff_lower is not None:
        mask = freq_axis >= cutoff_lower
    elif cutoff_upper is not None:
        mask = freq_axis <= cutoff_upper
    filtered_amplitude[mask, :] = 0
    reconstructed_amplitude = reconstruct_amplitude(filtered_amplitude, amplitude.shape[0])
    reconstructed_signal = inverse_fft(reconstructed_amplitude, phase)
    return reconstructed_signal

def threshold_filter(sample_rate, signal_data, threshold):
    amplitude, phase = apply_fft(signal_data)
    truncated_amplitude = truncate_amplitude(signal_data)
    filtered_amplitude = np.copy(truncated_amplitude)
    num_channels = signal_data.shape[1]
    # Apply threshold filter
    for i in range(num_channels):
        mask = (truncated_amplitude[:, i] <= threshold)
        filtered_amplitude[mask, i] = 0
    reconstructed_amplitude = reconstruct_amplitude(filtered_amplitude, amplitude.shape[0])
    reconstructed_signal = inverse_fft(reconstructed_amplitude, phase)
    return reconstructed_signal
```

```
def add_frequency(sample_rate, signal_data, freq_to_add, amplitude_to_add):
    amplitude, phase = apply_fft(signal_data)
    truncated_amplitude = truncate_amplitude(signal_data)
    freq_axis = get_freq_axis(sample_rate, truncated_amplitude)
    added_amplitude = np.copy(truncated_amplitude)
    # Add frequency
    idx = np.where((freq_axis > freq_to_add - 1) & (freq_axis < freq_to_add + 1))[0]
    added_amplitude[idx, :] += amplitude_to_add
    reconstructed_amplitude = reconstruct_amplitude(added_amplitude, amplitude.shape[0])
    reconstructed_signal = inverse_fft(reconstructed_amplitude, phase)
    return reconstructed_signal

def scale_amplitude(sample_rate, signal_data, freq_to_scale, scale_factor):
    amplitude, phase = apply_fft(signal_data)
    truncated_amplitude = truncate_amplitude(signal_data)
    freq_axis = get_freq_axis(sample_rate, truncated_amplitude)
    scaled_amplitude = np.copy(truncated_amplitude)
    # Scale frequency
    idx = np.where((freq_axis > freq_to_scale - 1) & (freq_axis < freq_to_scale + 1))[0]
    scaled_amplitude[idx, :] *= scale_factor
    reconstructed_amplitude = reconstruct_amplitude(scaled_amplitude, amplitude.shape[0])
    reconstructed_signal = inverse_fft(reconstructed_amplitude, phase)
    return reconstructed_signal

def shift_frequencies(sample_rate, signal_data, shift):
    amplitude, phase = apply_fft(signal_data)
    truncated_amplitude = truncate_amplitude(signal_data)
    # Convert Hz shift to sample shift
    shift_samples = np.round(shift / sample_rate * amplitude.shape[0]).astype(int)
    # Shift the frequency spectrum excluding DC component
    shifted_amplitude = np.concatenate([[truncated_amplitude[0, :]],
np.roll(truncated_amplitude[1:, :], shift_samples)])
    reconstructed_amplitude = reconstruct_amplitude(shifted_amplitude, amplitude.shape[0])
    reconstructed_signal = inverse_fft(reconstructed_amplitude, phase)
    return reconstructed_signal

def ask_segment_time_input(file_path, sample_rate, signal_data):
    duration_sec = get_duration_sec(sample_rate, signal_data)
    time_scale, xlabel = get_time_scale(duration_sec)
    duration_scaled = duration_sec * time_scale
    # Time input window
    segment_window = tk.Toplevel(root)
    segment_window.title("Time Input")
    segment_window.geometry('300x175')
    segment_window.resizable(False, False)
    def on_closing():
        pause_audio()  # Stop audio when the window is closed
        segment_window.destroy()
    segment_window.protocol("WM_DELETE_WINDOW", on_closing)  # Intercept window closing
event
```

```
    start_time = tk.StringVar()
    time_unit = tk.StringVar()
    ttk.Label(segment_window, text=f"File: {os.path.basename(file_path)}").grid(row=0,
column=0, columnspan=4, padx=10, pady=5)
    ttk.Label(segment_window, text=f"Duration: {duration_scaled:.2f} {xlabel}").grid(row=1,
column=0, columnspan=4, padx=10, pady=5)
    # Labels and Entries for the widgets
    ttk.Label(segment_window, text='Start Time:').grid(row=2, column=0, padx=10, pady=5,
sticky='e')
    ttk.Entry(segment_window, textvariable=start_time, width=6).grid(row=2, column=1,
padx=5, pady=5, sticky='w')
    tk.Button(segment_window, text="Play Audio", command=lambda: play_audio(file_path),
height=1, width=10).grid(row=2, column=2, padx=10, pady=5)
    ttk.Label(segment_window, text='Time Unit:').grid(row=3, column=0, padx=10, pady=5,
sticky='e')
    ttk.Combobox(segment_window, textvariable=time_unit, state='readonly', values=('ms',
's', 'min', 'h'),width=4).grid(row=3, column=1, padx=5, pady=5, sticky='w')
    tk.Button(segment_window, text="Pause Audio", command=lambda: pause_audio(), height=1,
width=10).grid(row=3, column=2, padx=10, pady=5)
    tk.Button(segment_window, text='OK', command=on_closing, height=1, width=10).grid(row=4,
column=2, padx=10, pady=5)
    # Configure columns to center the displayed widgets
    segment_window.grid_columnconfigure(0, weight=1)
    segment_window.grid_columnconfigure(1, weight=1)
    segment_window.grid_columnconfigure(2, weight=1)
    # Wait for the dialog window to close
    segment_window.wait_window()
    if start_time.get() and time_unit.get():
        # Implement checking if start_time + 200 ms < signal_duration
        if float(start_time.get()) + 0.2 * time_scale < duration_scaled:
            return float(start_time.get()), float(start_time.get()) + 0.2 * time_scale,
time_unit.get()
    else:
        print('Start time value not selected. Returning first 200 ms of the signal.')
        return 0, 200, 'ms'

def plot_signal(sample_rate, signal_data, signal_amplitude, wav_info, title):
    # Obtain and calculate relevant data
    duration_sec = get_duration_sec(sample_rate, signal_data)
    time_scale, xlabel = get_time_scale(duration_sec)
    duration_scaled = duration_sec * time_scale
    num_channels = wav_info[1]
    text_info = f'Channels: {wav_info[1]}, Audio Duration: {np.round(duration_scaled, 2)}
{xlabel}, Sampling Rate: {sample_rate} Hz, Bit Depth: {wav_info[2]} bits'
    # Create a figure with multiple subplots based on the number of channels
    fig, axs = plt.subplots(2, num_channels, figsize=(12, 9))
    axs = axs.reshape(-1, num_channels)
    time_axis_signal = get_time_axis(sample_rate, signal_data, time_scale)
    freq_axis_signal = get_freq_axis(sample_rate, signal_amplitude)
    for channel in range(num_channels):
        # Plot the signal and its amplitude spectrum for every channel
```

```
        axs[0, channel].plot(time_axis_signal, signal_data[:, channel], label='Original
signal', linewidth=0.4)
        if num_channels > 1:
            axs[0, channel].set_title(f'Channel {channel+1}', fontsize=12)
        axs[0, channel].set_ylabel('Amplitude')
        axs[0, channel].set_xlabel(f'Time ({xlabel})')
        axs[0, channel].set_xlim(-0.05 * duration_scaled, 1.05 * duration_scaled)
        axs[1, channel].plot(freq_axis_signal, signal_amplitude[:, channel],
label='Amplitude spectrum', c='#FF4500', linewidth=0.4)
        axs[1, channel].vlines(freq_axis_signal, [0], signal_amplitude[:, channel],
color='#FF4500', linewidth=0.5)
        axs[1, channel].scatter(freq_axis_signal, signal_amplitude[:, channel],
color='#FF4500', s=10)
        axs[1, channel].set_ylabel('Magnitude')
        axs[1, channel].set_xlabel('Frequency (Hz)')
        # Formatting options (scientific notation)
        for row in range(2):
            axs[row,
channel].yaxis.set_major_formatter(mticker.ScalarFormatter(useMathText=True))
            axs[row, channel].ticklabel_format(axis='y', style='sci', scilimits=(0,0))
            axs[row, channel].get_yaxis().get_offset_text().set_x(-0.08)
            axs[row, channel].grid(True)
            axs[row, channel].legend(loc='upper right')
    fig.suptitle(title, fontsize=14)
    fig.text(0.5, 0.03, text_info, fontsize=10, ha='center', va='top',
backgroundcolor='white')
    # Adjust the layout
    plt.subplots_adjust(left=0.1 if num_channels==1 else 0.085,\
                        right=0.9 if num_channels==1 else 0.98,\
                        top=0.93 if num_channels==1 else 0.9,\
                        bottom=0.11 if num_channels==1 else 0.12,\
                        hspace=0.17 if num_channels==1 else 0.25)
    plt.show(block=False)

def process_signal_options(sample_rate, signal_data, wav_info):
    def update_processing_options(*args):
        # Refresh window with new widgets
        for widget in option_widgets:
            widget.grid_forget()
        # Display widgets for each processing method
        processing_method = processing_method_var.get()
        match processing_method:
            case 'Band-Reject Filter':
                cutoff_lower_label.grid(row=2, column=0, padx=10, pady=5, sticky='e')
                cutoff_lower_entry.grid(row=2, column=1, padx=5, pady=5, sticky='w')
                cutoff_upper_label.grid(row=3, column=0, padx=10, pady=5, sticky='e')
                cutoff_upper_entry.grid(row=3, column=1, padx=5, pady=5, sticky='w')
            case 'Threshold Filter':
                threshold_label.grid(row=2, column=0, padx=10, pady=5, sticky='e')
                threshold_entry.grid(row=2, column=1, padx=5, pady=5, sticky='w')
            case 'Add Frequency':
```

```
                freq_to_add_label.grid(row=2, column=0, padx=10, pady=5, sticky='e')
                freq_to_add_entry.grid(row=2, column=1, padx=5, pady=5, sticky='w')
                amplitude_to_add_label.grid(row=3, column=0, padx=10, pady=5, sticky='e')
                amplitude_to_add_entry.grid(row=3, column=1, padx=5, pady=5, sticky='w')
            case 'Scale Amplitude':
                freq_to_scale_label.grid(row=2, column=0, padx=10, pady=5, sticky='e')
                freq_to_scale_entry.grid(row=2, column=1, padx=5, pady=5, sticky='w')
                scale_factor_label.grid(row=3, column=0, padx=10, pady=5, sticky='e')
                scale_factor_entry.grid(row=3, column=1, padx=5, pady=5, sticky='w')
            case 'Shift Frequencies':
                shift_label.grid(row=2, column=0, padx=10, pady=5, sticky='e')
                shift_entry.grid(row=2, column=1, padx=5, pady=5, sticky='w')
        apply_button.grid(row=4, column=1, padx=10, pady=15)
    def apply_processing():
        processing_method = processing_method_var.get()
        cutoff_lower = cutoff_lower_var.get()
        cutoff_upper = cutoff_upper_var.get()
        threshold = threshold_var.get()
        freq_to_add = freq_to_add_var.get()
        amplitude_to_add = amplitude_to_add_var.get()
        freq_to_scale = freq_to_scale_var.get()
        scale_factor = scale_factor_var.get()
        shift = shift_var.get()
        # Do original signal's frequency processing (for playing)
        modified_signal = modify_amplitude(sample_rate, signal_data, processing_method,
                                           cutoff_lower, cutoff_upper, threshold,
freq_to_add,
                                           amplitude_to_add, freq_to_scale, scale_factor,
shift)
        # Apply window to reduce spectral leakage
        windowed_signal = apply_window(sample_rate, signal_data, num_channels=wav_info[1],
window_function='hanning')
        # Do windowed signal's frequency processing (for displaying)
        modified_windowed_signal = modify_amplitude(sample_rate, windowed_signal,
processing_method,
                                                    cutoff_lower, cutoff_upper, threshold,
freq_to_add,
                                                    amplitude_to_add, freq_to_scale,
scale_factor, shift)
        # Truncate half of the amplitude (for displaying)
        truncated_amplitude_modified = truncate_amplitude(modified_windowed_signal)
        # Write and play temp files (for playing)
        file_path_original = write_file(sample_rate, signal_data,
output_filename="original_audio")
        file_path_processed = write_file(sample_rate, modified_signal,
output_filename="processed_audio")
        play_audio_files(wav_info[0], file_path_original, file_path_processed)
        # Plot processed signal and its amplitude spectrum
        plot_signal(sample_rate, modified_signal, truncated_amplitude_modified, wav_info,
title=f"'{processing_method}' Applied to File {wav_info[0]}")
    # Processing selection window
```

```
    processing_window = tk.Toplevel(root)
    processing_window.title('Processing Options')
    processing_window.geometry('300x175')
    processing_window.resizable(False, False)
    # Labels and Entries for the widgets
    processing_method_var = tk.StringVar()
    processing_method_var.trace('w', update_processing_options) # Trace variable to update
window with new widgets
    processing_method_label = ttk.Label(processing_window, text='Processing Method:')
    processing_method_dropdown = ttk.Combobox(processing_window,
textvariable=processing_method_var, state='readonly',
                                              values=['Band-Reject Filter', 'Threshold
Filter', 'Add Frequency', 'Scale Amplitude', 'Shift Frequencies'],
                                              width=17)
    processing_method_label.grid(row=0, column=0, padx=10, pady=15, sticky='e')
    processing_method_dropdown.grid(row=0, column=1, padx=10, pady=5, sticky='w')
    cutoff_lower_var = tk.DoubleVar()
    cutoff_upper_var = tk.DoubleVar()
    threshold_var = tk.DoubleVar()
    freq_to_add_var = tk.DoubleVar()
    amplitude_to_add_var = tk.DoubleVar()
    freq_to_scale_var = tk.DoubleVar()
    scale_factor_var = tk.DoubleVar()
    shift_var = tk.IntVar()
    cutoff_lower_label = ttk.Label(processing_window, text='Cutoff Lower (Hz):')
    cutoff_lower_entry = ttk.Entry(processing_window, textvariable=cutoff_lower_var,
width=6)
    cutoff_upper_label = ttk.Label(processing_window, text='Cutoff Upper (Hz):')
    cutoff_upper_entry = ttk.Entry(processing_window, textvariable=cutoff_upper_var,
width=6)
    threshold_label = ttk.Label(processing_window, text='Threshold:')
    threshold_entry = ttk.Entry(processing_window, textvariable=threshold_var, width=6)
    freq_to_add_label = ttk.Label(processing_window, text='Frequency to Add (Hz):')
    freq_to_add_entry = ttk.Entry(processing_window, textvariable=freq_to_add_var, width=6)
    amplitude_to_add_label = ttk.Label(processing_window, text='Amplitude to Add:')
    amplitude_to_add_entry = ttk.Entry(processing_window, textvariable=amplitude_to_add_var,
width=6)
    freq_to_scale_label = ttk.Label(processing_window, text='Frequency to Scale (Hz):')
    freq_to_scale_entry = ttk.Entry(processing_window, textvariable=freq_to_scale_var,
width=6)
    scale_factor_label = ttk.Label(processing_window, text='Scale Factor:')
    scale_factor_entry = ttk.Entry(processing_window, textvariable=scale_factor_var,
width=6)
    shift_label = ttk.Label(processing_window, text='Shift (Hz):')
    shift_entry = ttk.Entry(processing_window, textvariable=shift_var, width=6)
    option_widgets = [cutoff_lower_label, cutoff_lower_entry, cutoff_upper_label,
cutoff_upper_entry, threshold_label,
                      threshold_entry, freq_to_add_label, freq_to_add_entry,
amplitude_to_add_label, amplitude_to_add_entry,
                      freq_to_scale_label, freq_to_scale_entry, scale_factor_label,
scale_factor_entry, shift_label, shift_entry]
```

```
    apply_button = tk.Button(processing_window, text='Apply', command=apply_processing,
height=1, width=10)
    processing_window.mainloop()

def play_audio_files(file_name, path_original, path_processed):
    # Audio player window
    audio_window = tk.Toplevel(root)
    audio_window.title("Audio Player")
    audio_window.geometry('300x150')
    def on_closing():
        pause_audio()  # Stop audio when the window is closed
        file_paths = [path_original, path_processed]
        for file_path in file_paths:
            if os.path.isfile(file_path):
                os.remove(file_path)
        plt.close(plt.gcf())
        audio_window.destroy()
    audio_window.protocol("WM_DELETE_WINDOW", on_closing)  # Intercept window closing event
    # Display the audio file name
    audio_filename_label = tk.Label(audio_window, text=f"Selected Audio: {file_name}")
    audio_filename_label.pack()
    audio_filename_label.configure(anchor="center")
    # Display the duration of the audio
    signal_duration_label = tk.Label(audio_window, text=f"Signal Duration: 200 ms")
    signal_duration_label.pack()
    # Create widgets for playing/pausing original and processed audio
    play_original_button = tk.Button(audio_window, text="Play Original Audio",
command=lambda: play_audio(path_original), width=20)
    play_original_button.pack(pady=(5, 0), padx=10)
    modulate_button = tk.Button(audio_window, text="Play Processed Audio", command=lambda:
play_audio(path_processed), width=20)
    modulate_button.pack(pady=5, padx=10)
    pause_button = tk.Button(audio_window, text="Pause", command=lambda: pause_audio(),
width=20)
    pause_button.pack(pady=(0, 5), padx=10)

# Button for 'open_file'
open_button = tk.Button(root, text='Read WAV File', command=open_file, height=2, width=15)
open_button.pack(expand=True, pady=20)
# Tkinter mainloop
root.mainloop()
```