Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Énektudást fejlesztő alkalmazás

Készítette
Gungl Szilárd

Konzulens
Dr. Bank Balázs

2015

TARTALOMJEGYZÉK

HALLGATÓI NYILATKOZAT

Alulírott Gungl Szilárd, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2015. 12. 17.

.....

Gungl Szilárd

Összefoglaló

Napjainkban az énektanulást sokféle szoftver segíti. Ezek olcsó alternatívát jelentenek azok számára, akiknek nincs idejük, vagy pénzük, hogy nap mint nap énektanárhoz járjanak, de szeretnének a zuhany alatti éneklésnél tovább fejlődni.

Ezen programok általában a pontosságot igyekeznek fejleszteni, tehát, hogy az énekes képes legyen tisztán énekelni, azaz a kiénekelt hang minél közelebb legyen az ideálishoz. A programok egy csoportja vizuálisan is megjeleníti a felénekelt hangot. Ezáltal az énekes szinte azonnali visszajelzést kap a felénekelt hang magasságáról, és ezt látva rögtön tud korrigálni.

Az elérhető szoftverek referenciaként MIDI fájlokat dolgoznak fel, tehát a referenciajelet önálló tiszta zenei hangokból állítják össze. Ez hasonlít egy szintetizátoron lejátszott egyszólamú dallamhoz, mely a pontosság fejlesztéséhez elegendő ugyan, de nem ad információt a valós ének dinamikus tulajdonságairól. Megjeleníti a megcélzott hangokat, ahogyan azok a kottában le vannak írva, de nem tudjuk, mi történik a hangok között.

Ez alapvetően nem rossz, a kezdők számára remek gyakorlási lehetőséget biztosít. A haladóknak viszont igényük lehet arra, hogy a valós énekhang díszítéseit, például a hangok közti hajlításokat vagy a vibrato hullámzó hanghatását is minél pontosabban vissza tudják adni.

Ez a szakdolgozat ennek a megvalósítását tűzte ki célul. Egy olyan program elkészítése a feladat, mely referenciaként egy valós énekhangot dolgoz fel, majd a megállapított hangmagasságot kirajzolja, és lehetővé teszi, hogy az általunk felénekelt hangot összevessük a referenciával.

Dolgozatomban először a legismertebb hangmagasság-detektálási algoritmusokat vizsgálom. Ezeket egymással összevetem, majd a kapott eredményeket értékelve bemutatom az alkalmazott algoritmus kiválasztásának szempontjait.

Ezután a VST és a JUCE fejlesztőkörnyezet alapvető felépítését és működését ismertetem, kiemelve JUCE használatának legfőbb előnyeit. Majd a kiválasztott algoritmus C++ implementációját, valamint a teljes plugin felépítését és működését mutatom be. Végül az elkészült programot tesztelem és a kapott eredmények alapján összevetem a piacon elérhető többi hasonló alkalmazással.

A szakdolgozat zárásaként, az előállított szoftvert értékelem, és a jövőbeli továbbfejlesztési lehetőségeket is feltárom.

Abstract

Today there are many available software programs that help people to learn how to sing. They offer a favourable alternative for those who prefer not to take lessons from a singing teacher. These programs are cost effective and help participants save on time and money. Typically, this software uses a display to give visual feedback of the pitch sung. This grants the opportunity for the singer to correct the mistakes instantaneously.

These programs use a MIDI file as a reference, which suits the needs of the beginner who wishes to sing more accurately by focusing on simple, individual notes. However, an intermediate singer might want to perform like his/her favourite artist and further examine and practice all the nuances that a professional singer does; bending notes, belting and using vibrato, as is demonstrated in this program.

This app fulfils these performance expectations. The final product is a program in the form of a VST plugin that can load audio files and analyse them quickly. It is then able to compare the pitch of a live performance to the reference in real-time, making an easy-to-navigate tool for vocal performers. It further helps the vocalist practice singing in tune by loading a backing track.

By examining the most well-known pitch detection algorithms, this thesis explores the most common errors that occur within these programs. In order to find the most suitable algorithm for this software, comparison of known programs and errors is vital in order to create a successful program. Additionally, this thesis discusses the basics of VST and JUCE. It shows the fundamental structure of a VST plugin, which simulates a digital recording studio. The introduction of the JUCE development environment demonstrates how JUCE makes handling audio files and creating a graphical user interface simple. This thesis gives a brief explanation about the implementation of the chosen algorithm and the main components of the program: how it interacts with the user and the main functions.

In conclusion, this thesis tests the product with real and synthesizer generated test signals. It measures the accuracy and the dynamic properties of the created program and then compares this plugin to the available software.

1. Bevezetés

A zene története egyidős az emberiség történetével. Már az őskor embere is igyekezett megismerni az őt körülvevő világot: természeti jelenségeket, állatokat, növényeket. Az állathangokat utánozva kialakult egy hangzórendszer, melynek segítségével hívták egymást, elijesztették és űzték az ellenséget, óvták társukat a veszélytől. Minél több információt szerettek volna megosztani egymással, annál kifinomultabb jelrendszerre volt szükség.

Hosszas könyveket és tudományos értekezéseket találunk arról a vitás kérdésről, hogy melyik volt előbb: ének, vagy beszéd. Természetesen ma már úgy tekintünk az énekre, mint a beszéd egy dallamos változatára, de kialakulás szempontjából különbséget kell tennünk. Egyesek szerint őseink előbb zenéltek, mint hogy beszélni tudtak volna és az ének készítette őket fel a beszéden alapuló kommunikációra. Mások szerint viszont, a kialakult zene csak egy evolúciós véletlen [1].

Akárhogy is értelmezzük, a zene az emberiséget már a kezdetektől fogva elkíséri. Ezt bizonyítja a rengeteg feltárt hangszer is. Ám a hangszerek közül a legrégebbi minden bizonnyal maga az ember torka.

Különböző tanulmányok tárgyalják a zene, és kiváltképp az ének pozitív hatásait. Az éneklés megnöveli a tüdőkapacitást, javítja a közérzetet, csökkenti a stresszt és ezáltal erősíti az immunrendszert is [2]. Tehát az éneklés mindenki számára hasznos hobbivá válhat, még ha csak saját maga szórakoztatására zenél is valaki.

Napjainkban az énektanulást sokféle szoftver segíti. Ezek olcsó alternatívát jelentenek azok számára, akiknek nincs idejük, vagy pénzük, hogy nap mint nap énektanárhoz járjanak, de szeretnének a zuhany alatti éneklésnél tovább fejlődni.

Ezen programok általában a pontosságot igyekeznek fejleszteni, tehát, hogy az énekes képes legyen minél tisztábban énekelni. Más szavakkal, hogy a kiénekelt hang minél közelebb legyen az ideálishoz. A programok egy csoportja vizuálisan is megjeleníti a felénekelt hangot. Ezáltal az énekes szinte azonnali visszajelzést kap a felénekelt hang magasságáról, és ezt látva rögtön korrigálni tud.

Az elérhető szoftverek szinte kivétel nélkül fizetősek. Ezek közül talán az egyik leghíresebb a „Sing & See” [3], valamint a „Listening Ear Trainer” [4], mely ingyenes próbaverzióval is rendelkezik.

Ezen szoftverek alapötlete megegyezik a fent leírtakkal. Továbbá rendelkeznek még egyéb bővítményekkel, melyek egyrészt játékossá teszik az énektanulást, másrészt pedig a felénekelt hang követésén kívül egy kiválasztott referenciaminta gyakorlását is lehetővé teszik. Ez a referencia lehet egy skála, vagy akár egy egyszerű melódia is.

Ezek a programok az egyszólamú referenciajelet úgy állítják elő, mint ha azt, egy szintetizátoron játszottuk volna le. Ez, a diszkrét hangokból álló minta a tiszta éneklés gyakorlására kiváló lehetőséget nyújt, mivel csupán az egyes hangok minél pontosabb kiéneklésére összpontosít.

Az énekesek viszont sokféle eszközt használnak énekük díszítéséhez, mint például a hangok közti hajlítások vagy vibrato. A haladó énekesek számára fontos lehet, hogy ezen apró, ám mégis jelentős részleteket gyakorolni tudják. Ez a szakdolgozat egy olyan program megírását tűzte ki célul, mely ezt elérhetővé teszi.

A cél, hogy előre elkészített skálák használata helyett, a program lehetőséget biztosítson bármilyen egyszólamú hangfájl referenciaként történő felhasználására. Betöltés után a szoftver kiszámolná a hangmagassággörbét a fájl teljes terjedelmére, majd ezt vizuálisan is megjelenítené. Az elképzelés szerint, a gyakorlás során a mikrofon bemenőjelét folyamatosan elemezve kirajzolja az élő felvétel hangmagassággörbéjét is, a referencia görbe mellett. Ez azonnali visszajelzést ad, melynek segítségével az énekes rögtön tud korrigálni. A program természetesen eltárolná a görbéket, ezáltal lehetőséget biztosítva az utólagos elemzésre is.

Az élő felvétel kirajzolása során a pontosságot színekkel is szemléltetné. Az ideális hangtól való eltérés függvényében különböző színnel rajzolná a görbe adott szakaszát, mely lehetővé tenné az adott hang pontosságának azonnali megítélését.

Gyakorlás közben természetesen lehetőségünk nyílna a referencia felvétel ki- és bekapcsolására. Emellett a program kísérősávok (zenekari kíséret, vagy pl. egy kórusmű különböző szólamai) lejátszására is képes kell legyen. Ebben az esetben a tanuló a kísérősávot hallja, de továbbra is a referenciasávval történik az összehasonlítás.

A görgetési funkcióval szabadon mozoghatnánk a felvételben, mely lehetővé tenné például egy nehezebb szakasz folyamatos gyakorlását.

A dolgozatban először a különböző hangmagasság-detektálási módszereket tekintem át. Ismertetem ezen algoritmusok gyengeségeit és megpróbálok megoldást keresni a leggyakrabban előforduló problémákra. Az algoritmusok összevetése után kifejtem az alkalmazott algoritmus kiválasztásának fő szempontjait.

Ezt követi a program típusát jelentő VST szoftver interfész, és legfőbb előnyeinek bemutatása, melyek indokolttá tették a plugin forma kiválasztását. Majd megismerkedünk a JUCE fejlesztőkörnyezettel és osztálykönyvtárral, melynek segítségével a plugin és annak interfésze könnyedén elkészíthető.

A következő fejezetben szemléltetem a program felépítését és kifejtem a főbb funkciók megvalósítását. A szerkezet három nagyobb funkcionális egységre lett bontva a könnyebb érthetőség érdekében. Ezek az élő bemenőjel feldolgozását megvalósító-, a görbéket megjelenítő-, illetve a referenciafájlokat megnyitó és feldolgozó blokkok. Emellett kitérek az interfész megvalósítására és a különböző kijelző egységekre.

Végül az elkészült program tesztelése és értékelése következik. Különböző természetes énekhangokkal valamint szintetizátor által generált tesztjelekkel megvizsgálom a program tulajdonságait a pontosság és a dinamikus tulajdonságok szempontjából.

A programot nem ismerő felhasználók tesztjei után végrehajtott módosításokat is bemutatom. Ezek célja, hogy a plugin használata minél világosabb és egyszerűbb legyen.

Zárásként a programot összevetem a többi hasonló célú szoftverrel. Ezek közül alig néhány érhető el ingyenesen, így csak egyetlen próbaverzió tesztelésére nyílt lehetőség.

2. Az elmélet

Ezen fejezet célja az elméleti alapok áttekintése, a különböző, már létező hangdetektálási módszerek ismertetése, mind az idő, mind pedig a frekvenciatartományban, illetve azok bemutatása MATLAB környezetben.

2.1. Mi a hang?

A hang maga egy rugalmas közegben levő fizikai rezgés, mely hullámként terjed tovább az anyagban. Ennek a legegyszerűbb esete, ha a rezgés egy tiszta szinusz függvényt követ:

$$x = A\ cos(\omega t + \varphi) \tag{1}$$

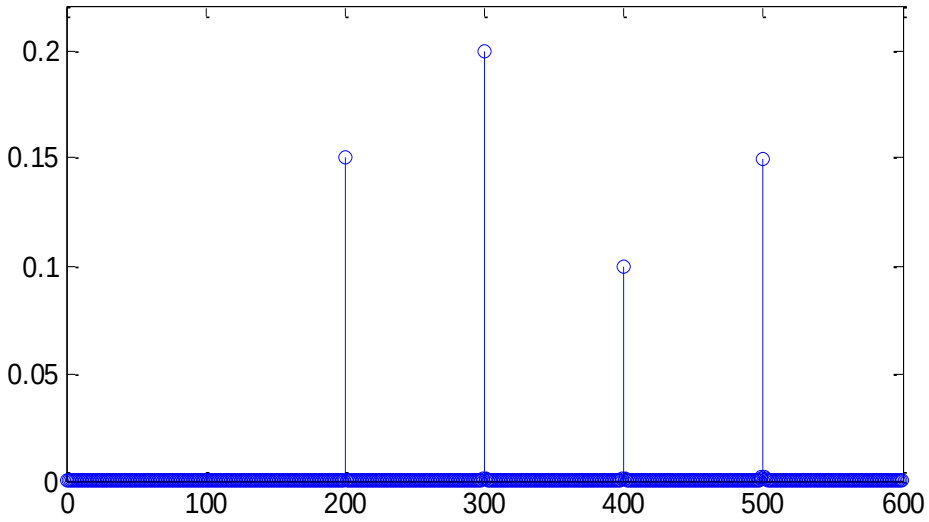
Egy valós hang azonban egy egyszerű szinusznál jóval összetettebb. Így például az énekhang és a hangszerek által keltett hang is sok-sok felharmonikust tartalmaz, ezek alakítják ki a jellegzetes hangszíneket. A felharmonikusok széles spektrumot alkotnak, mely akár tág frekvenciatartományon is elterülhet, viszont mindezek ellenére az emberi fül tisztán különbséget tud tenni az egyes, széles spektrummal rendelkező zenei hangok között. Ez felveti a következő érdekes kérdést, mi a hang magassága?

2.2. A hangmagasság (Pitch)

A hangmagasság (avagy az angol irodalomban „pitch”) egy érzékelt tulajdonság. Tudjuk, hogy a nagyobb frekvencia magasabb hangot eredményez. Ezen jelenség egy kézzelfogható szemléltetése a Seebeck-féle lyuksziréna. Az eszköz egy koncentrikus körök mentén egyenlő közű lyukakkal ellátott korong, amelyet egy rögzített tengely mentén megforgatunk. Ha az egyenletesen forgó korong valamelyik lyuksorára egy vékony csövön keresztül levegőáramot fújtatunk, a légáram periodikus megszakítása folytán a levegőben nyomáskülönbségek sora, azaz hanghullámok jönnek létre. A hanghullámok frekvenciája számértékben egyenlő a fúvóka előtt időegység alatt elhaladó lyukak számával [5].

Azt sejthetjük, hogy széles spektrumú hangok esetén is kapcsolatban áll a hangmagasság a hangot alkotó frekvenciákkal. A hangmagasság és a hang (Fourier felbontásban vett) alapharmonikusa gyakran meg is egyezik. Viszont léphet fel olyan eset is, amikor az alapharmonikus nincs is jelen (vagy csak nagyon kicsi amplitúdóval).

Ilyen például a virtuális hangmagasság (virtual pitch [6]) jelensége, melynél az alapfrekvenciát a felharmonikusokból tudjuk csak megállapítani, tudva, hogy ezek az alapharmonikus egész számú többszörösei (1. ábra).



1. ábra. „Virtuális hangmagasság”

Elterjedt értelmezés szerint a hangmagasság, vagyis a hang frekvenciája (f_0) az a frekvencia, melynek megfelelő tiszta szinusz jel azonos magasságú hang érzetét kelti, mint a vizsgált hang.

A különböző hangok sorba rendezésére zenei skálák szolgálnak. A skálák általában egy oktávot bontanak fel változó számú hangra. Egy oktáv az *f* frekvenciájú hang és *2f* közötti távolságot jelenti. Ebből tehát látszik, hogy a zenei hangok skálája frekvencia tekintetében logaritmikus. Ezen skálák nincsenek hangokhoz kötve, csupán a hangközöket szabják meg. A hangsor első hangja, az alaphang adja a skála nevét. Így például egy dúr skála, mely *B* hangról indul, *B*-dúrnak nevezünk. Ha egy félhanggal feljebb toljuk a skálát és *C*-ről indulunk, akkor *C*-dúrt kapunk. Ahhoz, hogy ilyen könnyedén tudjunk hangnemet váltani, meg kellett alkotni a „Temperált Kromatikus” (kiegyenlített) hangolást, mely az oktávot 12 egyenlő részre osztja fel [7].

Viszont így szemlélve a dolgokat minden hang relatívvá válik, kell egy alappont. Ez lett a normál zenei *A* hang (A4, A440), melyet a Nemzetközi Szabványügyi Szervezet 1955 novemberében jegyzett – be hosszas konferenciák és viták – után 440 hertzes frekvencián, majd 1975 januárjában újra megerősítette azt [8].

2.3. MIDI hangkódolás

A Musical Instrument Digital Interface (MIDI) egy szabvány, mely sok más egyéb tulajdonsága mellett a zenei hangok egy kézenfekvő digitális kódolási lehetőségét nyújtja.

A MIDI kódolásban 1 lépés a kromatikus skála 1 félhangjának felel meg. A frekvencia kódját az alábbi módon kaphatjuk meg (ahol *f* a kódolni kívánt frekvencia):

$$p = 69 + 12\ log_2\left(\frac{f}{440}\right) \quad (2)$$

Láthatjuk, hogy itt is megjelenik a 440 hertz, melyből nem nehéz kiszámolni, hogy a normál zenei hang MIDI kódja a 69.

Mivel ezen a logaritmikus skálán minden hang között a távolság 1 (tehát linearizált), így ez felhasználható egyfajta mérőszámként is, hogy a valós mért hang mennyire esik távol az ideálistól. Ennek kézenfekvő módja, hogy a kapott MIDI számból elvesszük a kerekített értékét és ezt megszorozzuk 100-zal, így egy – 50 és + 50 közötti értéket kapunk, ezt nevezzük centnek. Tehát tulajdonképpen 1 cent egy félhang 1/100-ad része.

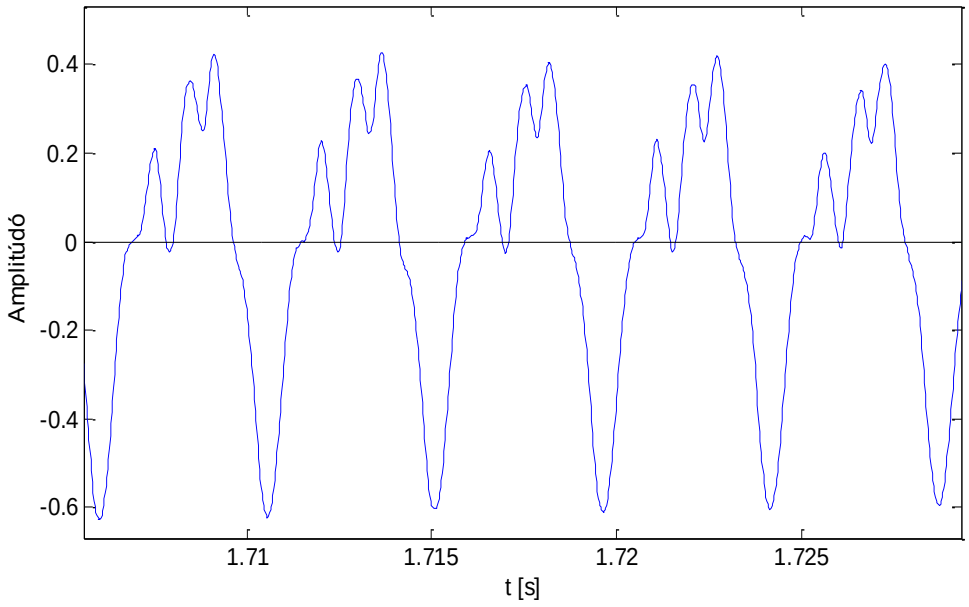
Két frekvencia közt az eltérés centben (1200 cent = 1 oktáv [12 félhang: temperált skála]):

$$n = 1200\ log_2\left(\frac{f_a}{f_b}\right) \quad (3)$$

Ez a megközelítés pusztán a matematikai leírást segíti meg. Az, hogy az emberi fül hány cent eltérést tud megkülönböztetni, nehéz megítélni. Egyrészt ez természetesen személyenként változik, másrészt viszont az is szerepet játszik, hogy egy önálló hangot (monofonikus) vagy harmóniát (akkord, polifonikus) hangot hallunk.
Önálló hang esetében a legkisebb, ember által megkülönböztethető különbséget 5-6 cent értékűnek tekintik [9], míg egy átlagos embernél ez 25 cent körül mozog. Ugyanakkor harmóniák esetében ez jelentősen javul.

2.4. Hangmagasság-detektálási módszerek

A virtuális hangmagasság (1. ábra) tárgyalásánál már láthattuk, hogy a hangmagasság megállapítása nem teljesen triviális feladat. További problémát jelent a felharmonikus tartalom. Ha egyszerűen csupán nyers jelen a nullátmenetek távolságából próbálnánk frekvenciát számolni, könnyen hibás eredményhez juthatunk.



2. ábra. Nullátmenet hiba

Vegyük a 2. ábra példáját, mely egy tiszta „A” gitárhang[1] jelét ábrázolja. A felharmonikus tartalom miatt egy perióduson belül 2 extra nullátmenet is látható. Felharmonikusról lévén szó, ez a detektáló algoritmusban pontosan oktávhibát fognak okozni.
Másrészről viszont frekvenciatartományban is könnyen problémákba ütközhetünk (1. ábra). Így tehát a hangmagasság detektálásához vagy az algoritmust magát kell robosztussá tenni, hogy kiküszöbölje az esetlegesen fellépő hibákat, mely természetesen a számítási igény növekedésével is jár, vagy magát az alkalmazási területet kell leszűkíteni, hogy az algoritmusnak enyhébb specifikációt adjunk.

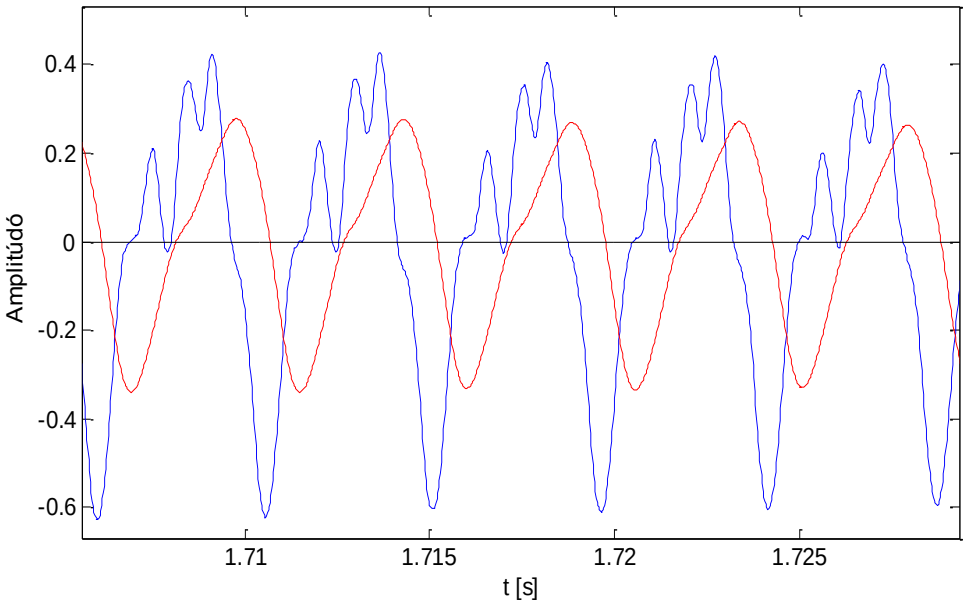
2.4.1. Időtartomány

Az időtartománybeli vizsgálat alkalmazása kézenfekvő a „real-time”, azaz a valós idejű felhasználáshoz: nem kell nagy adathalmazokkal dolgozni, és ezáltal a késleltetés sem lesz túlságosan nagy.

2.4.1.1. Nullátmenet vizsgálat (ZCR)

A bevezetőben említett módszer, a hangmagasság megállapításának talán egyik legtriviálisabb módja a nullátmenetek távolságának vizsgálata, avagy a Zero-Crossing Rate (ZRC). Ennek előnye, hogy gyors, akár már egy periódus után is szolgáltathat hasznos információt, viszont azt már láttuk, hogy a felharmonikus-tartalom miatt könnyen téves eredményt adhat. Ezt némileg kiküszöbölendő, először egy aluláteresztő szűrővel dolgozzuk fel a jelet, és csak utána próbálunk nullátmeneteket keresni.
Ezt a megoldást implementáltam MATLAB-ban is.

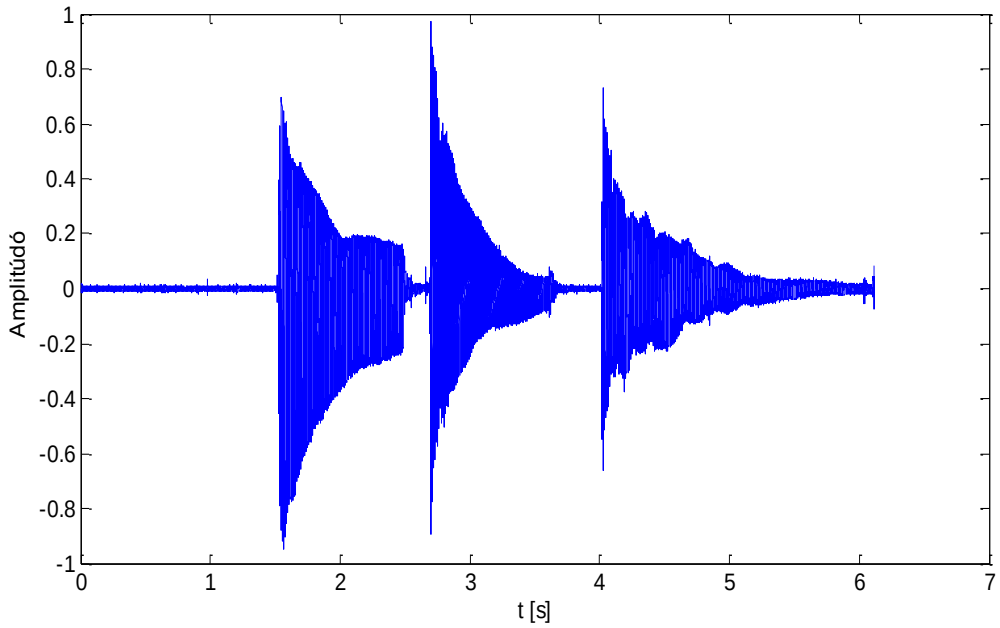
[1] A gitár könnyen hangolható, stabil jelet biztosít, kezdetben ezért esett erre a választás



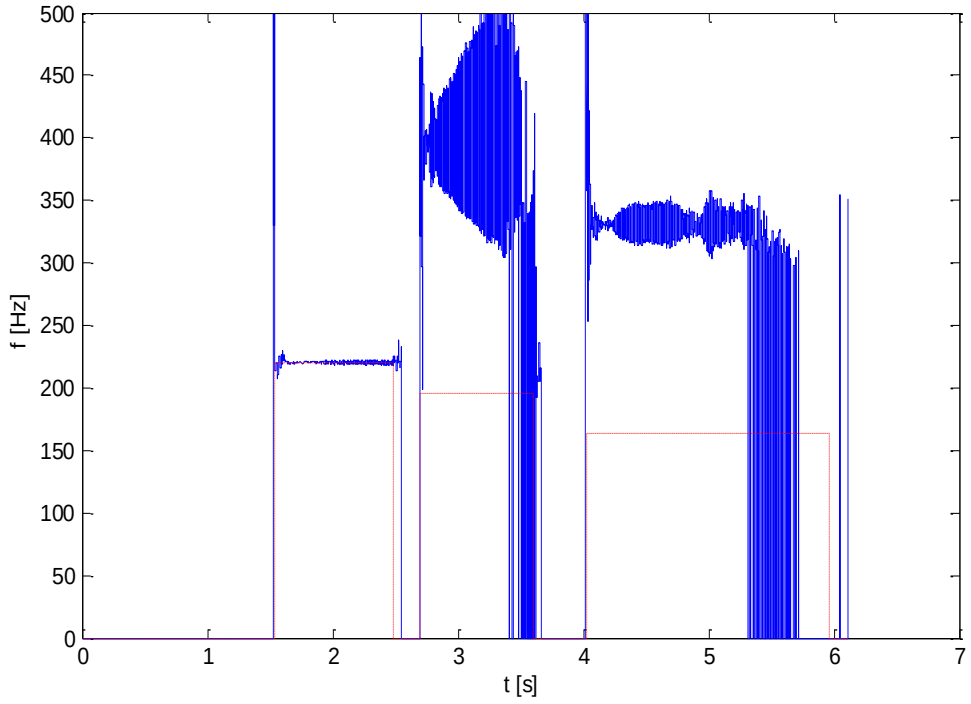
3. ábra. Eredeti (kék) és aluláteresztő szűrővel szűrt jel (piros)

Ahogy a fenti ábra mutatja, a szűrés eltűntette az extra nullátmeneteket és kissé kezelhetőbbé tette a jelet. Ezután megkeresve a negatívból pozitívba történő nullátmeneteket és ebből frekvenciát számolva egyszerűen megkapjuk a hangmagassággörbét (5. ábra).

A tesztjel három gitárhang, egymás után lejátszva: A (220 Hz), G (196 Hz) és E (164 Hz).



4. ábra. Az alkalmazott tesztjel



5. ábra. ZCR által megtalált hangmagasság
(piros vonallal a referencia)

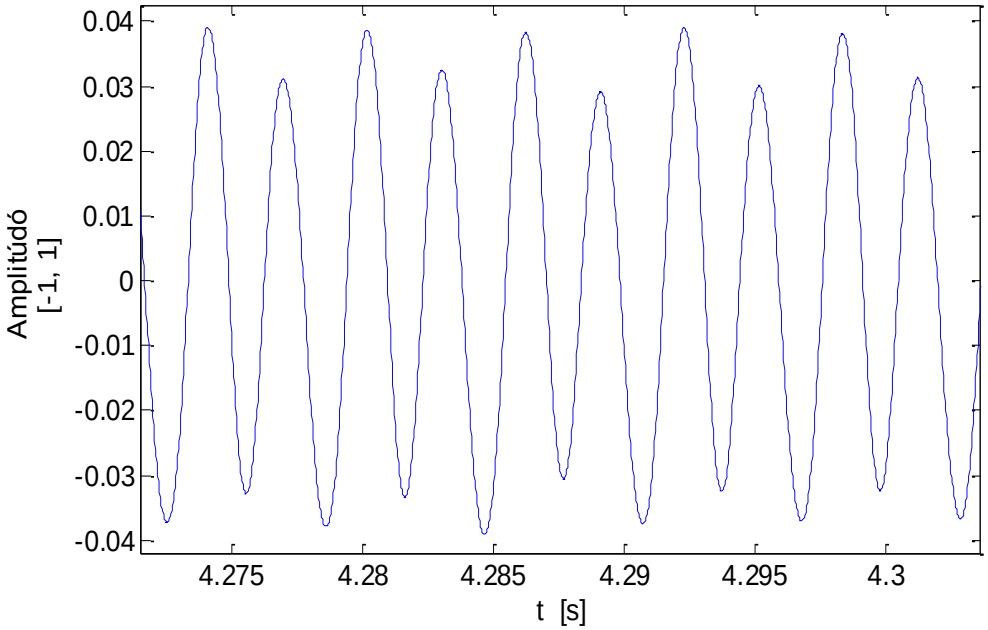
A kapott eredményt az 5. árán látható. Megfigyelhető, hogy ez az egyszerű módszer néhol eredményes, viszonylag stabil hangmagasságot talál, máshol viszont instabil, ám ami még rosszabb, hogy oktáv hibát is mutat.

Ennek oka, hogy egy egyszerű szűrő alkalmazása esetén továbbra is domináns marad a felharmonikus-tartalom. A legfőbb problémát az erős második harmonikus jelenti (6. ábra). Mivel azt szeretnék, hogy az algoritmus széles tartományon működjön – például az átlagos énekhangot lefedő 50-1000 Hz intervallumon – a szűrőt ekkora áteresztő-tartománnyal kell méretezni. Viszont ha például az A (220Hz) hangot vizsgáljuk, világossá válik, hogy a második harmonikusa még bőven az áteresztő tartományon belül van. Esetleg szűrhetnénk több sávon szűk áteresztő-tartományú szűrővel is. Ennek a módszernek a neve „Multirate Singal Processing”.

Alapötlete, hogy frekvenciasávokra bontja a jelet (felezi) és így teszi könnyebben feldolgozhatóvá. Ezzel az a probléma, hogy még negyedfokú szűrővel, 250 Hz vágási frekvenciával is jelentkezik az oktávhiba a harmadik, E hang esetében (6. ábra). A jel további sávokra bontása és a még magasabb fokú IIR vagy FIR szűrők alkalmazása viszont már jóval megnövelné a számítási teljesítményt, így elveszítenénk a ZCR egyszerűségét és gyorsaságát.

Léteznek egyéb módszerek is a ZCR pontosítására. Az egyik elképzelés az, hogy ne csak a nullátmeneteket tároljuk el, hanem az azokhoz tartozó meredekséget is, ezáltal szűrni tudunk köztük és könnyen megtalálhatjuk az ismétlődő mintát. Ezzel az a probléma, hogy egy egyszerű és tiszta gitár jelénél is ezek a meredekségek annyira közel esnek egymáshoz és annyira nagy a szórásuk, hogy nehéz különbséget tenni köztük.

Egy harmadik ötlet az, hogy egyszerűen csak tároljuk el a nullátmenetek helyét, és ezek közt keressünk ismétlődő mintát, így az ismétlődés frekvenciája lesz hangmagasság. Ez már lényegében egy korreláció alapú megoldás, melyet a következő fejezetben tárgyalunk.



6. ábra. Domináns felharmonikus tartalom

2.4.1.2. Korreláció család

Az hangmagasság időtartománybeli megállapításának egy széles családja a korreláción alapul. Rengeteg különböző nevű algoritmus létezik, mely kisebb trükkökkel próbálja gyorsabbá és robosztusabbá tenni ezt az algoritmust, de az alapötlet ugyanaz. A korreláció lényege, hogy a jelet eltolja és összehasonlítja eredeti önmagával, ezáltal a periodicitás és ismétlődő minták megtalálására tökéletes. Ha az eltolás mértékét változóként kezeljük, megkapjuk a korrelációfüggvényt.

$$r(\tau) = \sum_{j=-\infty}^{\infty} x_j \, x_{j+\tau} \tag{4}$$

Ennek a függvénynek maximuma a nullában van. Tegyük fel, hogy T periódusidejű szinuszról van szó, ekkor a függvénynek lokális maximumai vannak $\pm T$, $\pm 2T$, $\pm 3T$… eltolásoknál is.

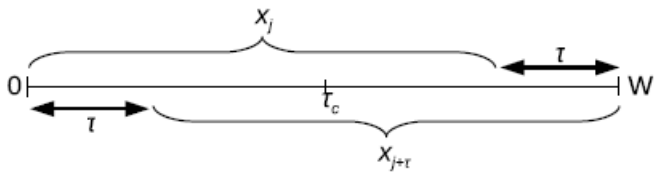
Valós idejű jelfeldolgozás esetén azonban nem az egész adathalmazon dolgozunk, csupán annak egy kiablakolt szakaszán. Így a korrelációfüggvény is némileg módosul.

$$r(\tau) = \sum_{j=0}^{W-1-\tau} x_j \, x_{j+\tau} \qquad 0 \le \tau < W \tag{5}$$

Ahol „W” az ablakolt szakasz hossza. Ennek az ablaknak legalább *2T* hosszúnak kell lennie, hogy az ismétlődés fellelhető legyen. Így viszont az algoritmus maga *T* késleltetést hoz be.

Az ablakhossz megválasztása is érdekes kérdés. Ahhoz, hogy tudjuk, mekkora ablakhosszal kell dolgoznunk, tudnunk kellene a mérni kívánt jel frekvenciáját. Így tehát egyetlen lehetőségünk, hogy az előforduló legkisebb frekvenciának megfelelő ablakméretet választjuk.

A korreláció ezen értelmezése változó ablakmérettel dolgozik, mely az értékek lineáris csökkenését eredményezi, ám nagy előnye, hogy az „effektív centruma” (τ_c) mindig állandó, az ablak közepén helyezkedik el. Tehát a τ-tól függő, egyre rövidülő szakaszok az ablakon belül mindig (középpontosan) szimmetrikusan helyezkednek el.



7. ábra. Effektív centrum értelmezése

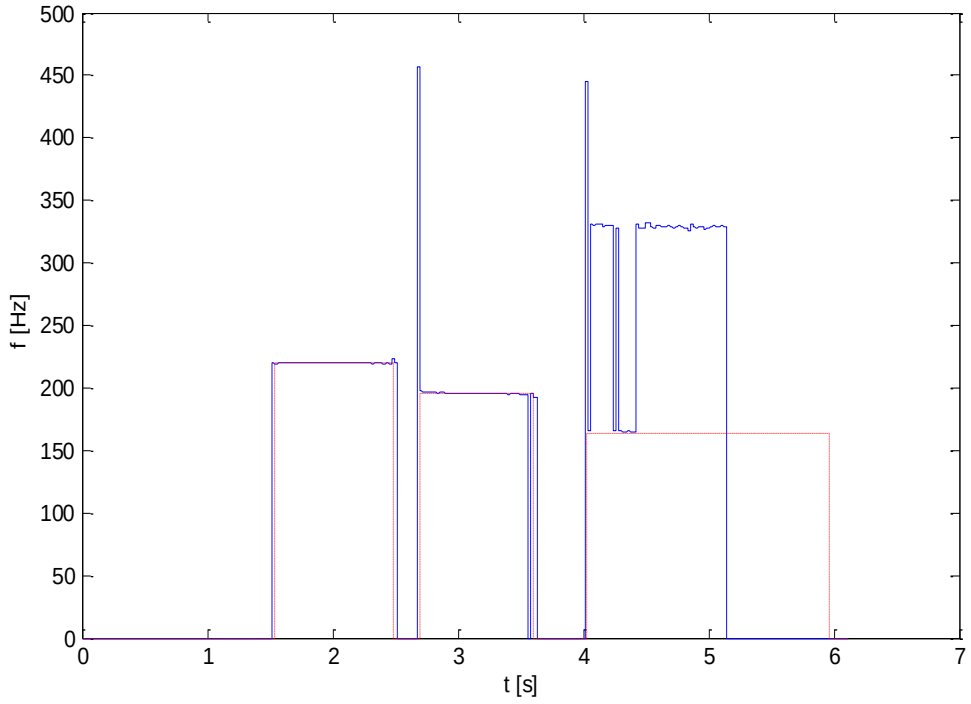
Ez a szimmetria lehetővé teszi, hogy mindig a teljes hosszát felhasználva számoljunk korrelációt, ezáltal kiegyenlítve változó frekvenciájú jel ablakon belüli különbségeit. Másrészről, ha esetleg interpolációval szeretnénk a maximum leolvasását pontosítani, értelmesebbnek tűnik ezt egy olyan korrelációs függvénynél megtenni, melynél az adatok időben ekvidisztáns pozíciókból származnak [10].

Az autokorreláció annak ellenére, hogy időtartománybeli módszer, meglehetősen nagy számításigénnyel jár. Ezt gyorsítani lehet FFT (Fast Fourier Transformation) alkalmazásával [11].

Ez tömören:

- Az eredeti adattömböt ki kell egészíteni nullákkal, hogy elkerüljük az átlapolódást (cirkuláris korrelációból lineáris korreláció). Ez *W*-db nullát jelent.
- Kiszámoljuk az FFT-t a *2W* hosszú tömbre.
- Vesszük a komplex elemek abszolútértékeinek négyzetét.
- Végül inverz FFT-vel megkapjuk a korrelációt (5)

FFT-vel gyorsított korreláció MATLAB implementációja:



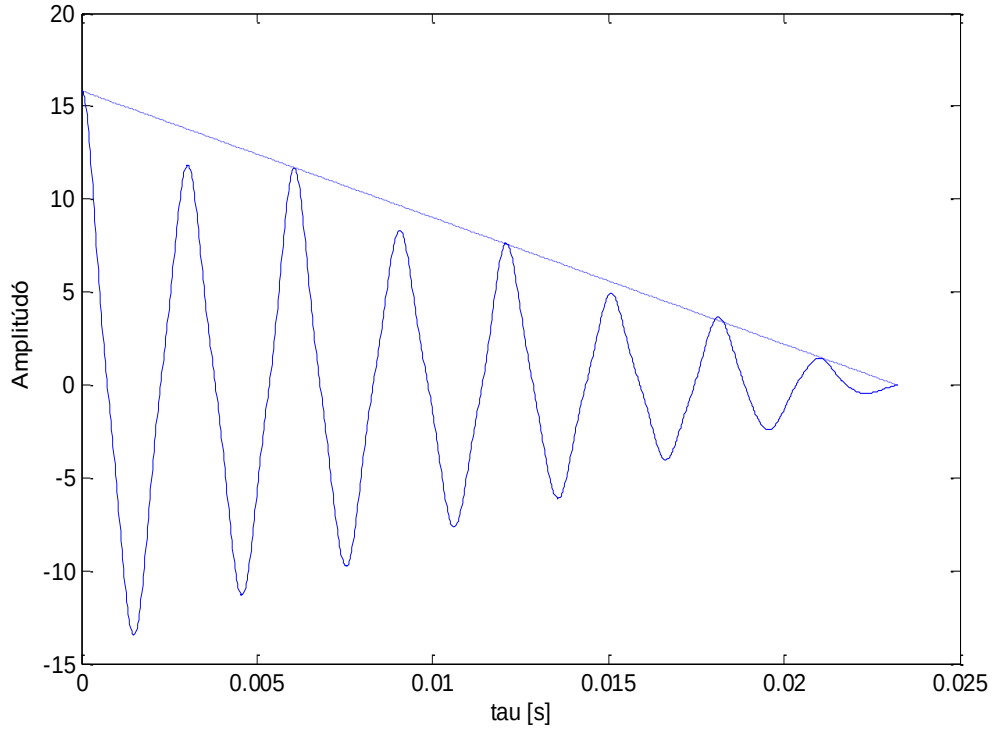
8. ábra. Korreláció segítségével kiszámolt hangmagasság

Mint látható (8. ábra), a korreláció egészen stabilan találja meg hangmagasságot, valamint a kitartott hang „határait” is élesen elválasztja. Ugyanakkor itt is megjelenik az oktávhiba a 4. másodperctől.

Ennek oka, hogy ez a fajta korreláció (az ablakolt korrelációt más módon is ki lehetne számolni) folyamatosan csökkenő ablakmérettel dolgozik. Vegyük szemügyre a (5) egyenletet. Eszerint folyamatosan csökkenő „τ” érték mellett a szumma mérete egyre csökken, tehát a korreláció függvény is ezáltal lineárisan csökken. Így tehát a jelen levő magasabb frekvenciakomponensek, melyek kisebb eltolást jelentenek, mintegy nagyobb súllyal számítanak. Egy egyszerű maximumkeresés ezért vezet oktáv hibához.

Ezt a lineáris csökkenést egy háromszög alakú normáló függvénnyel kompenzálhatnánk, ahogy az a 9. ábrán látható (a háromszög- és a korrelációfüggvény egyik fele van csak ábrázolva). Ez a módszer viszont nem veszi figyelembe azt, hogy a jel energiája nem egyenletesen oszlik el az ablakon belül.

A helyes maximumérték megtalálása korántsem triviális feladat. Kereshetjük a nulla utáni első lokális maximumot, ez sokszor helyes eredményhez vezet, viszont gyakran okoz oktávhibát (9. ábra). Kereshetnénk egy bizonyos küszöbszintet meghaladó első maximumot is, de még ha a nulla helyen vett értékével normálnánk is a korreláció függvényt, a küszöbszint jelfüggő lesz.



9. ábra. a korreláció hibája

A 7. ábrán láthatjuk, hogy ez első két lokális maximum amplitúdója egy picit közelebb van egymáshoz, ugyanígy a harmadik és a negyedik. Tehát úgy tűnik, jelen esetben mégsem lineárisan csökken a korreláció. Ennek oka, hogy az első lokális maximum nem a jel alapfrekvenciáját reprezentálja, hanem az első felharmonikust, ám mivel az ablakméret folyamatosan csökken, ezért a második – a hangmagasságnak megfelelő – lokális maximum alacsonyabb lesz, ezért hibás a maximumkeresés.

Ezt a hibát küszöböli ki a 2.4.1.4 fejezet megoldása.

2.4.1.3. Négyzetes eltérés függvény (SDF – Square Difference Function)

Az SDF függvény is tulajdonképpen autokorrelációt valósít meg, annyi különbséggel, hogy nem maximumot, hanem minimumot keres:

$$d(\tau) = \sum_{j=0}^{W-1-\tau} \left(x_j - x_{j+\tau}\right)^2 \qquad 0 \le \tau < W \tag{6}$$

A függvény a jelből elveszi az eltoltját és a különbségeket négyzetre emelve összeadja (abszolútértéket is lehetne venni, ez lenne az AMDF). Ha „τ” megegyezik a vizsgált jel periódusával, akkor ott a függvénynek lokális minimuma van.

Ez az algoritmus is sajnos egy eléggé erőforrásigényes megoldás. Szerencsére ennek is létezik FFT-vel gyorsított változata [10].

$$d(\tau) = \sum_{j=0}^{W-1-\tau} ({x_j}^2 + {x_{j+\tau}}^2) - 2 \sum_{j=0}^{W-1-\tau} x_j \, x_{j+\tau} \tag{7}$$

$$= m(\tau) - 2r(\tau)$$

A számítási igény csökkenthető, ha az első tagot az alábbi rekurzív módon számítjuk:

$$m(0) = \sum_{j=0}^{W-1} 2{x_j}^2 \tag{8}$$

$$m(\tau) = m(\tau - 1) - {x_{\tau-1}}^2 - {x_{W-\tau}}^2$$

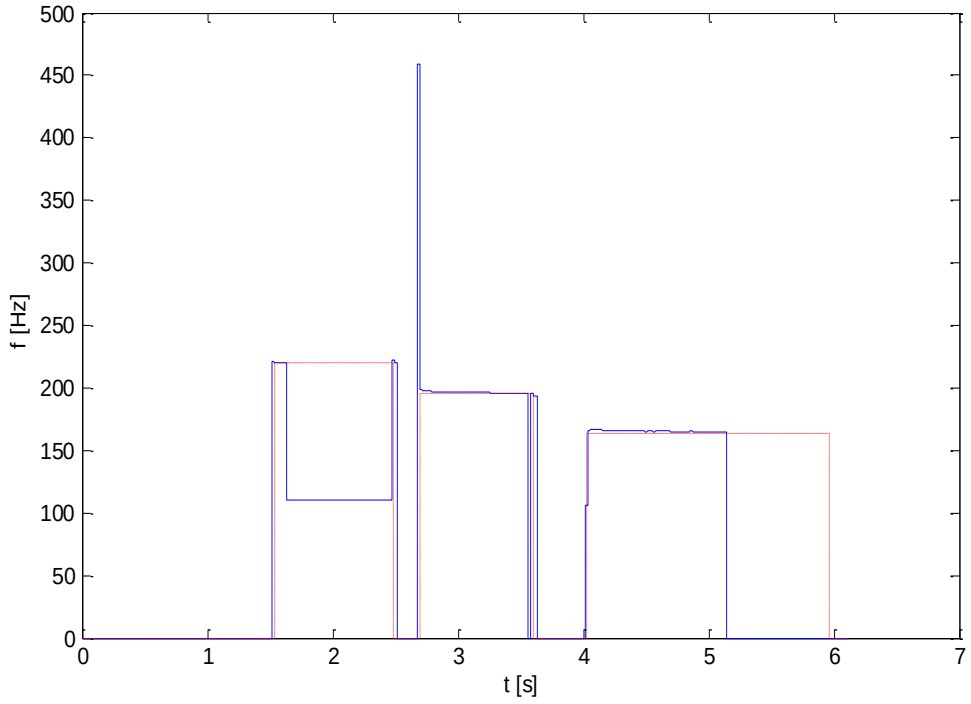
A második tag $r(\tau)$ pedig ugyanaz az FFT-vel segített korreláció, mint az (5) egyenlet.

Tehát az SDF függvény és a korreláció között látszólag annyi csupán a különbség, hogy míg egyik maximumot, addig a másik minimumot keres egészen hasonló elven. De létezik egy nagy előnye a korrelációval szemben, mely a jelből csak kevés periódust tartalmazó, kisebb méretű ablakoknál figyelhető meg. Ha az adott ablakban nem pont egész számú periódus található, akkor a korreláció maximumpontja – ugyanazon jel más kezdőfázisa esetén – eltolódik. Ugyanakkor az SDF függvény minimumpontja mindig ugyanoda esik [10].

Ez az eltolódás a sok periódust tartalmazó ablakoknál jelentéktelenül kicsi, ellenben ha csak néhány és nem egész számú periódust tartalmaz az ablak, akkor már számottevővé válik.

Az FFT-vel gyorsított SDF MATLAB implementációjának eredményét mutatja a 10. ábra.

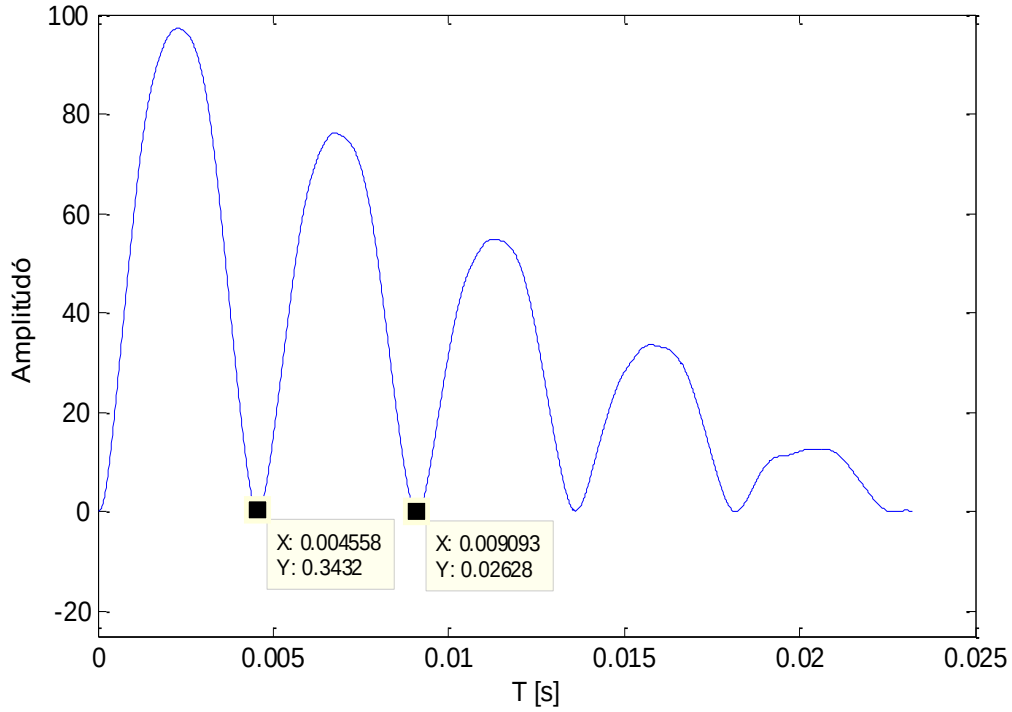
Mint látható, a korrelációs függvény gyengesége az SDF erőssége és fordítva. Itt is megjelenik az oktávhiba, viszont más helyen és más irányba (egy oktávval mélyebben). Ennek az oka ugyancsak a csökkenő ablakméretnél keresendő.



10. ábra. SDF algoritmus által kiszámolt hangmagasság

Mivel most a különbség minimalizálásáról van szó, így ha kevesebb adattal kell dolgozni, akkor fennáll a lehetősége annak, hogy ezek a rövidebb blokkok jobban „összepasszolnak”, tehát kevesebb köztük az eltérés valamint a frekvencia is állandóbbnak tekinthető egy rövidebb szakaszon, ezáltal kisebb lesz a minimum is. Ezt az eshetőséget mutatja a 11. ábra. Ugyanúgy, mint a korrelációnál a maximum- most a minimumkeresés lesz problémás.

Fel lehetne állítani mindenféle heurisztikus maximum- illetve minimumkeresési algoritmust, melyek megpróbálják kinyerni a megfelelő lokális maximumhoz tartozó eltolás τ értékét, de ennél létezik egy egyszerűbb módszer is. Ez a következő fejezetben ismertetett SNAC függvény.



11. ábra. SDF hibája

2.4.1.4. A SNAC függvény [10]

A SNAC függvény (Special Normalization of the Autocorrelation Function), avagy a „speciális módon” normált autokorrelációs függvény ötvözi az autokorreláció és a SDF legfőbb erényeit. Egyaránt magába foglalja az autokorreláció állandó „effektív centrum” és az SDF fázisintolerancia tulajdonságait.

A SNAC függvény definíciója az alábbi:

$$n(\tau) = \frac{2\sum_{j=0}^{W-1-\tau} x_j\, x_{j+\tau}}{\sum_{j=0}^{W-1-\tau}\left({x_j}^2 + {x_{j+\tau}}^2\right)} = \frac{2r(\tau)}{m(\tau)} \tag{9}$$

Eszerint venni kell a korrelációt és ezt elosztani egy normáló függvénnyel, $m(\tau)/2$-vel.

Korábbiakból viszont már láttuk, hogy mindkét függvény kiszámítását jelentősen tudjuk gyorsítani. A korreláció számításánál alkalmazhatunk FFT-t (17. oldal), az SDF függvényt pedig rekurzívan is kiszámíthatjuk (8) szerint.

Egy rövid levezetéssel bizonyítható, hogy ez a függvény pontosan ±1 közé normálja a korrelációt:

$$(x_a \pm x_b)^2 \geq 0$$

$${x_a}^2 + {x_b}^2 \geq \pm 2 x_a x_b$$

$$\sum_{j=0}^{W-1-\tau} \left({x_j}^2 + {x_{j+\tau}}^2\right) \geq \left| 2 \sum_{j=0}^{W-1-\tau} x_j \, x_{j+\tau} \right| \tag{10}$$

Ha teljes korrelációban van a jel és az eltoltja, akkor lesz a két tag egyenlő és a SNAC függvény 1. Léteznek más metódusok is a korreláció függvény normálására (legegyszerűbb az első értékkel történő osztás), ám ezek, még ha lineárisan csökkenő értékeket ki is küszöbölik, nem veszik figyelembe azt, hogy a jel energiája nem egyenletesen oszlik el az ablakban.

Azt, hogy a SNAC függvény a korrelációval rokonságot mutat, könnyű belátni, viszont egy kicsit közelebbről szemlélve azt is láthatjuk, hogy szorosan kapcsolódik az SDF függvényhez is. A (7) képletből:

$$2r(\tau) = m(\tau) - d(\tau)$$

Így a SNAC definíciója (9) alapján:

$$n(\tau) = \frac{m(\tau) - d(\tau)}{m(\tau)} = 1 - \frac{d(\tau)}{m(\tau)} \tag{11}$$

Eszerint a SNAC függvénynek pont ott lesz a maximuma, ahol az SDF függvény minimuma, ezáltal a normáló függvény egyfajta korrelációvá alakítja azt.

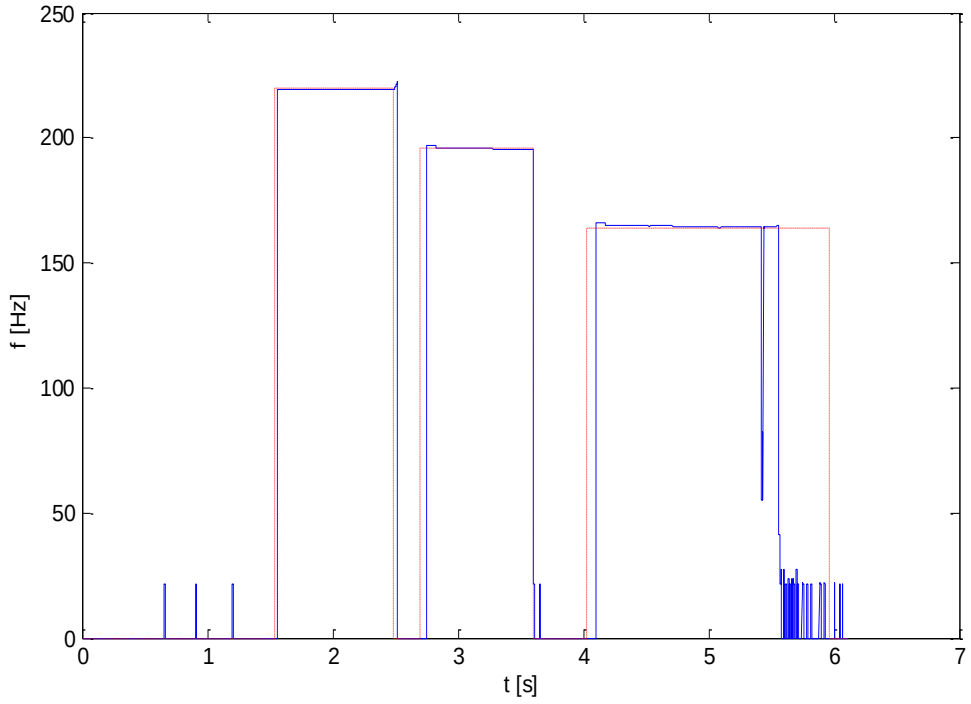
Tehát ezzel a korrelációt a jel tulajdonságainak figyelembevételével normáltuk. Ennek segítségével könnyebben, akár egy egyszerű küszöbszint megadásával is tudunk maximumhelyeket keresni. A diszkrét függvény maximumhelyének megtalálása után a még pontosabb mérés érdekében, a csúcs legfelső három pontjára parabolát illesztünk, és ennek keressük meg a maximumhelyét. A maximumhelyet megadó képlet, ha a felső három pont: $[x_1, y_1]$; $[x_2, y_2]$; $[x_3, y_3]$

$$x_{max} = \frac{{x_3}^2(y_2 - y_1) + {x_2}^2(y_1 - y_3) + {x_1}^2(y_3 - y_2)}{2[x_3(y_2 - y_1) + x_2(y_1 - y_3) + x_1(y_3 - y_2)]} \tag{12}$$

Mindezeket MATLAB-ban implementálva az eredmény a 12. ábra képén látható. A legjobb eredményt akkor kapjuk, ha a maximumkeresés küszöbszintjét 0.85 – 0.9 közé állítjuk be. Ez tapasztalati úton meghatározott érték [10].

Láthatjuk, hogy oktávhiba ekkor nem keletkezik, valamint az algoritmus egészen stabilan megtalálja a hangmagasságot. Ez különösen látványos a hosszasan lecsengő, elhalkuló harmadik hang esetén, melynél még 5.5 másodpercen túl is kapunk hasznos jelet.

Eddig ez az algoritmus tűnik a leginkább kecsegtetőnek, ám hátulütője, hogy nagy számításigénye van és megfelelő felbontáshoz nagy ablakméretet kell használnunk. Ugyan gyorsíthatjuk FFT alkalmazásával, de ekkor már felmerül a kérdés, hogy nem járnánk-e jobban egy frekvenciatartománybeli módszerrel. Ezt a kérdést járja körül a következő fejezet.



12. ábra. SNAC függvény által kiszámolt hangmagasság

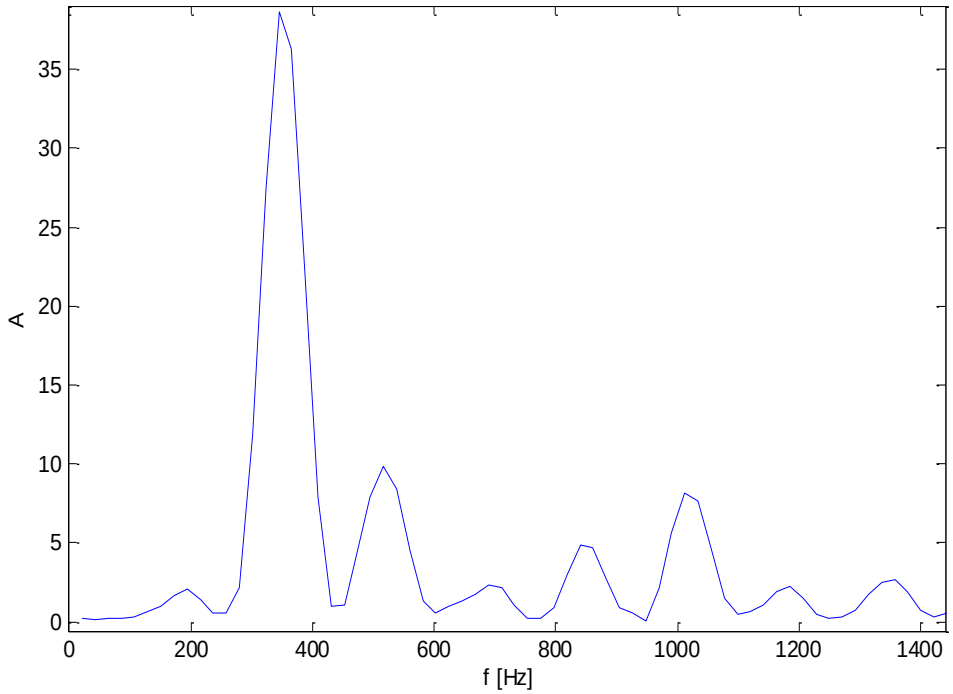
2.4.2. Frekvenciatartomány

A frekvenciatartománybeli módszerekről általánosságban elmondható, hogy a Fourier-transzformáció miatt nagy számításigényűek, valamint nagy ablakmérettel dolgoznak. Idő és pontosság fordított arányosságban állnak: ahhoz hogy pontosan tudjunk számolni, növelnünk kell az ablakméretet, ezáltal kevésbé lesz érzékeny, dinamikus az algoritmus, ellenben, ha gyorsítani szeretnénk, az a pontosság rovására megy.

Az időtartománybeli módszerek a jelből csupán 1-2 periódust igényelnek, míg egy pontos Fourier-transzformációhoz legjobb esetben is kell 4-5 periódus. Ez is további késleltetést jelent.

2.4.2.1. Spektrális csúcsok módszere

Ezen módszer alapötlete egészen egyszerű: végezzünk el Fourier-transzformációt a bejövő jelen, majd a kapott spektrumon keressük meg a hangmagassághoz tartozó csúcsot. A probléma a helyes maximum megtalálásában rejlik. Ha egyszerűen a legnagyobb csúcsot próbálnánk megkeresni, akkor erős felharmonikus tartalom esetén oktávhibát kaphatunk.



13. ábra. Énekhang (*G ~ 196 Hz*) spektruma

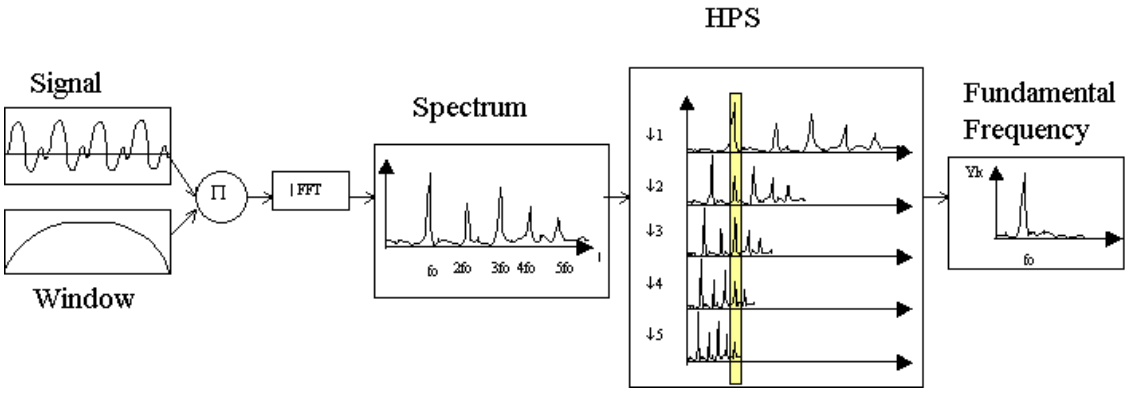
Mint látható, a második harmonikus sokkal nagyobb amplitúdóval van jelen, mint az alapharmonikus, ekkor eléggé nehézkes a helyes csúcs kiválasztása. Az esetek jelentős részére igaz, hogy nem az alapharmonikus hordozza a legnagyobb energiát.

Másrészt a megfelelő felbontású spektrumhoz a zenei jelfeldolgozásban használatos ablakméreteknél jóval több pontos FFT elvégzése szükséges.

2.4.2.2. Harmonic Product Spectrum (HPS)

A HPS módszer a megfelelő spektrális csúcs kiválasztására nyújt megoldást. Az algoritmus alapötlete a következő: legalább *N* felharmonikust tartalmazó hang spektrumát *N*-ed részére összenyomva (ami újramintavételezéssel egyszerűen megvalósítható) az *N*. felharmonikus épp az alapharmonikus eredeti helyére kerül. Ezután az eredeti és az összenyomott spektrumokat összeszorozva – ideális esetben – az alapharmonikuson kívüli összetevők elhanyagolhatóvá válnak és az alapharmonikus („fundamental frequency” 14. ábra) egyszerű maximumhely kereséssel meghatározható [12], [13].

A módszer előnye, hogy additív zajra érzéketlen. Hátránya azonban, hogy még így is gyakran oktávhibát mutat, valamint, hogy rövid, 1-2 periódust tartalmazó ablakok esetén pontatlan.



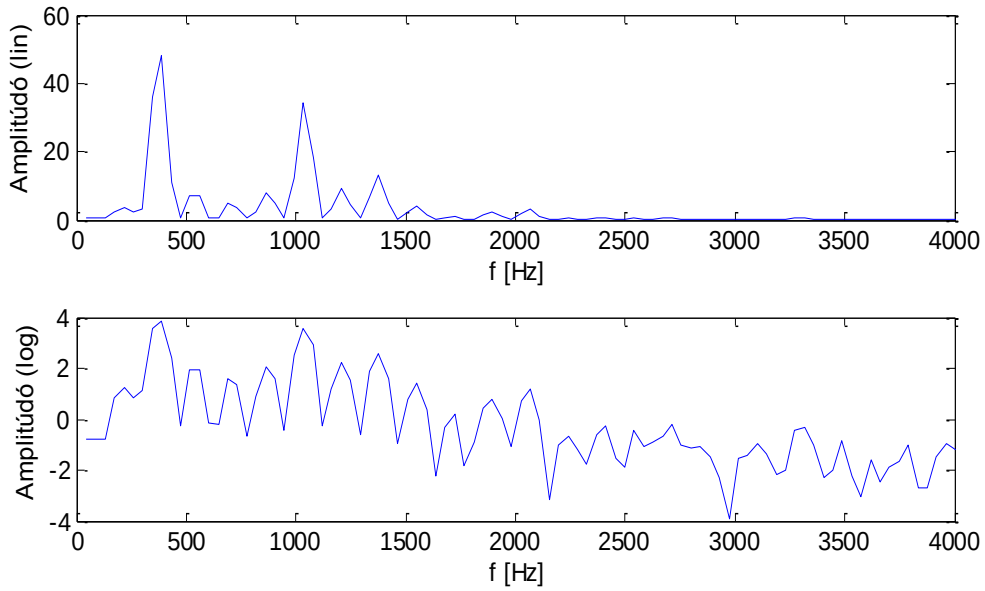
14. ábra. HPS algoritmus működése

2.4.3. Egyéb megközelítések

Ebben a fejezetben egy olyan elterjedt algoritmust tárgyalunk, melyet nem tekinthetünk tisztán időtartománybeli vagy tisztán frekvenciatartománybeli módszernek.

2.4.3.1. Cepstrum

A cepstrum („kepstrum”) módszer abból indul ki, hogy a vizsgált jel spektrumában egymástól egyenlő távolságra elhelyezkedő csúcsok vannak, melyek az alapfrekvencia periódusával ismétlődnek, ezek a felharmonikusoknak megfelelő csúcsok. Ha vesszük a spektrum logaritmusát, akkor a spektrumot összenyomjuk, csúcsok helyett inkább egy közel periodikus jelet kapunk.

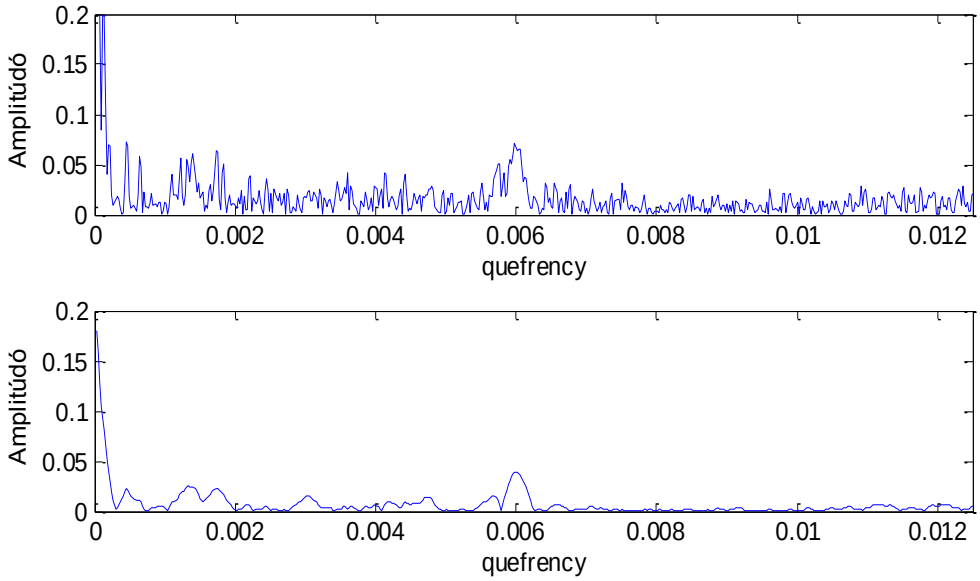


15. ábra. Jel spektruma (felül) és annak logaritmusa (alul)

Ha ezen a spektrumon végzünk el egy újabb Fourier-transzformációt (vagy inverz Fourier-transzformációt), akkor megkapjuk ennek a periodikus jelnek (a logaritmikus spektrumnak) a frekvenciakomponenseit. Ekkor tulajdonképpen a frekvencia frekvenciáját keressük meg, tehát visszatérünk időtartományba (ezt bizonyos irodalmak „quefrerncy”-nek is nevezik [10]). Ezután egy egyszerű maximumkereséssel megtalálhatjuk azt a csúcsot, mely pont az alapharmonikus periódusidejéhez tartozik.

Egyes tanulmányok azt fedezték fel, hogy a spektrum jobb oldalán, tehát olyan magas frekvenciákon, melyek már nem képzik a hasznos jel részét (zaj), a spektrum nulla közeli értékei – logaritmus elvégzése után nagy negatív értékek – a cepstrum váratlan viselkedését válthatják ki [10].

Ezt elkerülendő, a logaritmus számítása előtt egy konstanst adunk hozzá a spektrumhoz, így a nagy, esetenként végtelenbe tartó étékek helyett nulla közelieket kapunk. Ezt nevezzük módosított cepstrumnak [10].
Ez szemmel látható változást eredményez:



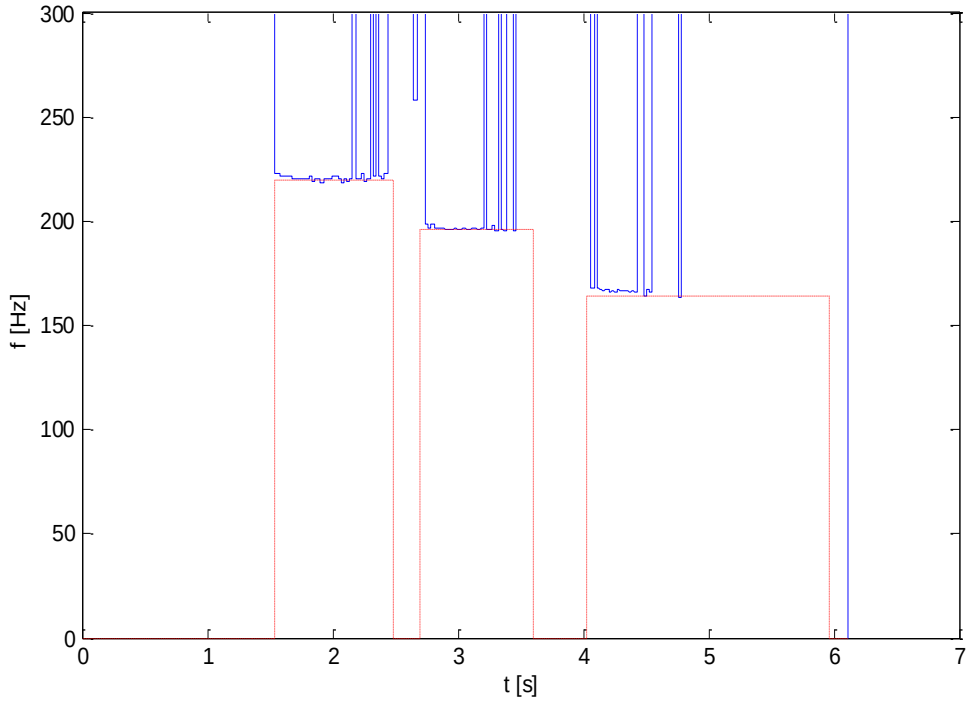
16. ábra. Eredeti (felül) és módosított cepstrum(alul)

Tehát a cepstrum kiszámításának módja:

$$|\mathcal{F}^{-1}\{\log(c + |\mathcal{F}\{x(t)\}|)\}| \tag{13}$$

Ezen algoritmus MATLAB implementációjának eredményét mutatja be a 17. ábra.

Ezen módszer előnye, hogy viszonylag stabilan találja meg az oktávot, valamint felharmonikusokban gazdag jel esetén pontos eredményt ad. Ugyanakkor hátránya a nagy számítási igény, két Fourier-transzformáció, két abszolútérték végzés és egy logaritmusszámítás, ez természetesen késleltetést is jelent a rendszerben.



17. ábra. Cepstrum módszerrel kiszámolt hangmagasság

2.5. Alkalmazott módszer

Ebben a fejezetben megismertünk számos hangmagasság-detektáló algoritmust. Ezek különböző eszközökkel igyekeznek a hangmagasságnak megfelelő frekvenciát kinyerni a jelből, valamint eltérő módokon próbálják kiküszöbölni a felharmonikus tartalom okozta oktávhiba jelenségét. Továbbá láttuk azt is, hogy a számítási pontosság és késleltetési idő között kompromisszumot kell kötni.

Az algoritmussal szemben támasztott általános elvárások a következők:

* Először is az algoritmus találja meg pontosan a hangmagasságot, legalább olyan pontossággal, amit az emberi fül érzékelni tud.
* Legyen robosztus, stabilan működjön széles frekvenciatartományon.
* Legyen érzékeny a hangmagasság időbeli változására (pl. vibrato effektus), dinamikusan kövesse azt
* Végül legyen „real-time” üzemeltethető.

Ezen szempontok alapján a választás a SNAC függvényre esett. Ugyan eléggé nagy számításigénye van, viszont nagyon pontosnak bizonyuló algoritmus.

3. Felhasznált eszközök

3.1. VST

A Virtual Studio Technology (VST) a német Steinberg zenei technológiával foglalkozó cég által fejlesztett interfész, mellyel hangszermodellek és effektek könnyedén integrálhatóak zenei szerkesztő és felvevő szoftverekbe. A VST digitális jelfeldolgozást (DSP) alkalmaz, hogy a hagyományos stúdiók hardver eszközeit szoftveresen szimulálni tudja.

A hangszerek modelljeit és az effekteket pluginok formájában készítik el, *.dll* kiterjesztéssel. Alapvetően háromfajta plugint különböztetünk meg:

* *VSTi (VST instrument), hangszermodellek*
  Ezek a plugin-ok hangot generálnak, általában valamely híres szintetizátor hardverét szimulálják.

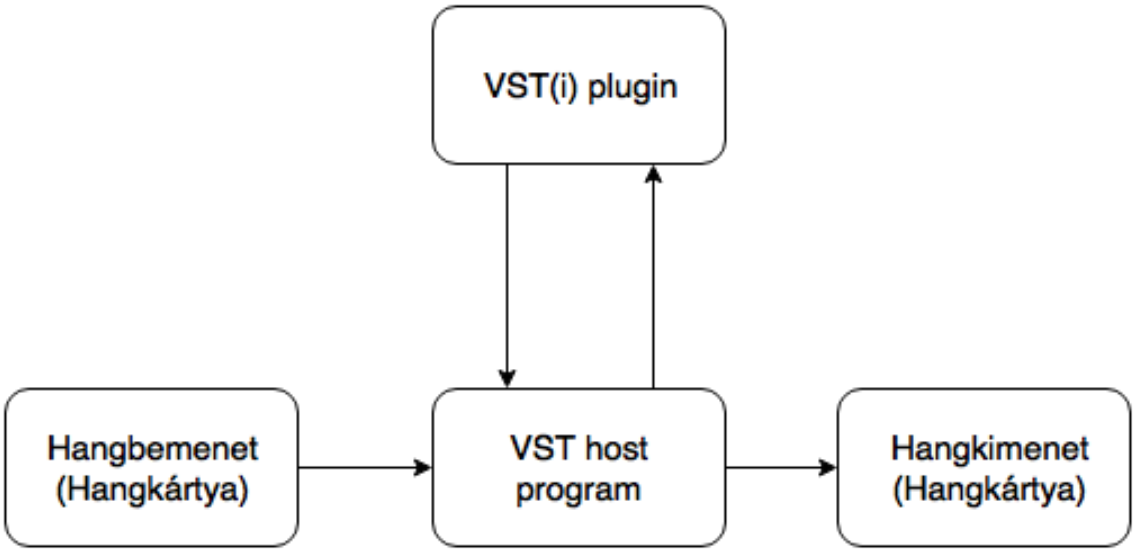
* *VST effekt*
  Feldolgozzák a bemenetükre adott jelet, különböző hardveres effekteket szimulálnak (reverb, phaser).
  Emellett elláthatnak monitorozási funkciót, vagyis a bemenőjelet vagy annak egy tulajdonságát megjelenítik, valamilyen vizuális visszajelzést adnak róla.

* *VST MIDI effekt*
  MIDI üzeneteket dolgoznak fel, vagy továbbítanak az előzőhöz hasonló módon.

Ezeket a pluginokat a VST gazdaprogramok, a hostok futtatják. A VST igen elterjedt és széles körben alkalmazott. Manapság számos ingyenes és fizetős VST-vel kompatibilis, azaz host-ként üzemeltethető Digital Audio Workstation (DAW, stúdiót szimuláló szoftver) és megannyi plugin áll rendelkezésre. Ezek felvételre és zenei utómunkára egyaránt alkalmasak.

A host keretprogramként funkcionál. Ő kommunikál az operációs rendszerrel, olvassa be az adatot a hangkártyából és küldi vissza a feldolgozott hangot. A host blokkosan szolgáltatja az adatot a plugin számára. Tehát a pluginokat értelmezhetjük egyfajta függvényként, vagy inkább függvények összességeként, melyeket a főprogram időnként meghív. Így a pluginnak nem kell törődnie a környezettel, csak a szolgáltatott adatok feldolgozásával.

Ezek a pluginok általában grafikus interfésszel is rendelkeznek, melyen keresztül megjelenítik a különböző vizuális információkat, valamint a működést befolyásoló főbb paramétereket is ennek segítségével tudjuk megváltoztatni.



18. ábra. VST működése

A VST programozása *C* vagy *C++* nyelven történik a Steinberg által szolgáltatott SDK (Software Development Kit) segítségével. Ez a VST pluginok elkészítéséhez keretrendszert és különböző fejlesztői eszközöket biztosít.

Én a szakdolgozatomban VST gazdaprogramként a „VSTHost" ingyenesen elérhető programot használtam fel, mivel a plugin működéséhez szükséges feltételeket biztosítja, valamint egyszerű a használata [14].

3.2. JUCE

A JUCE [15] egy nyílt forráskódú, multi-platform *C++* keretrendszer, mely segítséget nyújt asztali számítógép programok, és mobilalkalmazások létrehozásához. A JUCE alapvető célja, hogy ugyanazon forráskódú programot futtatni tudjunk Windows, Mac és Linux operációs rendszereken.

A JUCE sokféle fejlesztői környezetet és fordítót támogat, mint például GCC, Xcode, Code::Blocks, és amit én is felhasználtam: Microsoft Visual Studio.

Mint a többi hasonló keretrendszer, a JUCE is rengeteg osztályt kínál számunkra, melyekkel könnyedén megoldhatók a különböző feladatok. Az egyes osztály családok lehetőséget nyújtanak felhasználói interfész, grafikus elemek, audio alkalmazások, többszálú programozás, stb. megvalósítására.

A JUCE számunkra különösen előnyös sajátossága a többi keretrendszerhez képest, hogy rengeteg, direkt audio felhasználásra szánt függvénye és osztálya van. Támogatja a különböző audio eszközöket, mint az ASIO, CoreAudio, ALSA, JACK, WASAPI,

DirectSound és a legelterjedtebb hangfájl formátumokat: WAV, AIFF, FLAC, MP3. Külön keretrendszerrel rendelkezik a VST plugin-ok fejlesztésére, VSTi és VST effektek részére egyaránt [16].

JUCE részeként elérhető az Introjucer nevű fejlesztőkörnyezet, mellyel könnyedén tudunk JUCE projekteket létrehozni, módosítani. Új projekt létrehozása esetén, miután beállítottuk felhasznált fájlokat (például VST SDK) és kiválasztottuk a megfelelő beállításokat, az Introjucer automatikusan létrehozza a kívánt platformokon való működéshez szükséges projektfájlokat. Ezek tartalmazzák a preferált fordító vagy fejlesztőkörnyezet beállításait, a forrásfájlok vázát, szükséges könyvtárakat.

Introjucer ezen kívül kódszerkesztő felülettel is rendelkezik, ám ami ennél sokkal hasznosabb, grafikus interfész szerkesztővel is. Ebben lehetőségünk nyílik az alapvető megjelenés elkészítésén kívül feliratokat, gombokat, csúszkákat hozzáadnunk a GUI-hoz (Graphical User Interface). Ezekhez az objektumokhoz az Introjucer automatikusan generál például *Listener*eket, azaz olyan függvényeket melyek a gomb megnyomása, vagy a csúszka értékének megváltoztatása esetén hívódnak meg. Ezeket automatikusan beépíti a forráskódba is.

Annak érdekében, hogy a forráskód automatikusan generált részeit véletlenül se módosítsuk, világosan megjelöli, hogy melyek azok a részek, ahova a saját kiegészítéseinket elhelyezhetjük. Ezek html címkéhez hasonlító kommentek nyitó és záró elemmel, melyek közé szabadon írhatunk saját kódot:

```
//[MiscUserCode]
…
//[/MiscUserCode]
```

A JUCE weboldalán rengeteg dokumentációt, API-t, sőt még oktató jellegű leírásokat, alapvető mintaprogramokat is találhatunk [15].
Ilyen mintaprogram például egy egyszerű hangerőszabályzó plugin, mely csupán egyetlen csúszkát tartalmaz. Ezt én is felhasználtam a plugin elkészítése során [17].

Tehát összességében a JUCE a VST pluginok készítésének egyszerű és gyors módját jelenti. Segítségével könnyedén szerkeszthetünk grafikus interfészt, valamint a zenei jelfeldolgozáshoz is támogatást nyújt. Ezért esett erre a választás.

4. JUCE Implementáció

Mint láthattuk, a JUCE rengeteg osztályt kínál a különböző feladatok megvalósításához. Ezek az osztályok a programozás kiindulópontját jelentik. Saját osztályokat származtathatunk belőlük, felülírhatjuk a virtuális függvényeket, valamint a kódot további elemekkel bővíthetjük a felhasználási igények szerint. Ezeket az elkészített osztályokat fogja majd a host program példányosítani, a függvényeit meghívni.

Ez a fejezet az elkészített plugin szerkezetét, osztályait, függvényeit és azok funkcióit tárgyalja.

4.1. A plugin szerkezete

A plugin szerkezete forrásfájlok és működés szempontjából is két részre osztható. Az egyik a *PluginProcessor* (röviden processzor), a másik pedig a *PluginEditor* (röviden editor). Mindkét részhez tartozik külön – a JUCE által generált – header és forrásfájl.

4.1.1. *PluginProcessor*

A processzor tekinthető a plugin magjának. Ezt az elemet hívja meg az operációs rendszer, valamint rajta keresztül tarja a kapcsolatot a programmal, illetve itt történik a bejövő jel blokkonkénti feldolgozása is. Benne megtaláljuk a *FancyAudioProcessort*, mely az *AudioProcessor* nevű JUCE alap osztályból származik publikus módon.

Az *AudioProcessor* egy „bázisosztály”, mely a pluginok és szűrők elkészítéséhez nélkülözhetetlen vázat tartalmazza. Magába foglalja az összes olyan függvényt, mely információval szolgál a pluginról a host program számára. Ilyen például a program neve, csatornák száma, hogy kezel-e MIDI-t, van-e editora (GUI) stb.

Ha a plugin rendelkezik GUI-val, akkor az is ezen az osztályon keresztül hívódik meg.

Mindezek mellett megtaláljuk benne a tényleges jelfeldolgozást megvalósító *processBlock* nevű függvényt is.

```
void processBlock (AudioSampleBuffer&, MidiBuffer&);
```

Mint látható, ez a függvény egy *AudioSampleBuffer* osztályt kap referenciaként. Ezen keresztül tud adatot átadni a host és a plugin. Mérete a beállított mintavételi ablak nagyságától függ. A MIDI üzeneteket a mi esetünkben nem használjuk fel.

A buffer be- és kimenet egyaránt. Ha pluginnak van kimenete, akkor ide kell visszaírnia a bemenetéről feldolgozott adatokat, melyeket a host továbbít a hangkártya felé.

Az inicializálást az *AudioProcessor* konstruktorán kívül még egy *prepareToPlay* nevű függvény is segíti, mely paraméterként megkapja a beállított mintavételi frekvenciát és az ablakban levő minták számát.

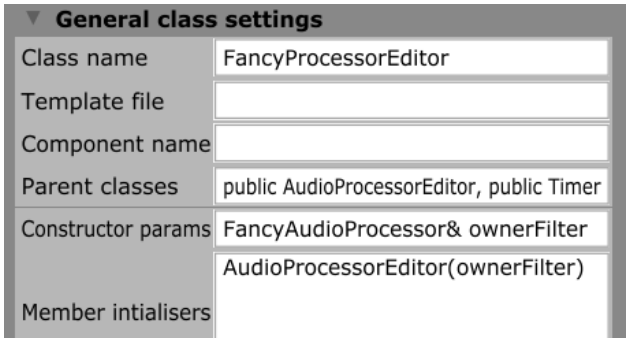
4.1.2. *PluginEditor*

Az editor teremti meg a kapcsolatot a felhasználó és a plugin közt. Ez lényegében egy grafikus interfész felület, melyen keresztül módosítani tudjuk a plugin paramétereit, valamint vizuális visszajelzést kapunk a jel vagy a plugin bizonyos tulajdonságairól.

Az általam megvalósított osztály neve *FancyProcessorEditor,* mely az *AudioProcessorEditor, Timer* és *ButtonListener* szülőosztályokból származik.

Az *AudioProcessorEditor* egy olyan „bázisosztály”, mely a felhasználói grafikus interfész (GUI) megírásához szükséges alapvető szerkezetet tartalmazza. Ennek az osztálynak természetesen kapcsolódnia kell egy *AudioProcessor*hoz.

Úgy tudjuk összekötni a kettőt, hogy az Introjucerben a Class fülnél a konstruktor paraméterei között megadjuk a processzort, mint referenciát.



19. ábra. Az editor osztályának beállításai

Az editorból a processzor adatait a *getProcessor()* segítségével tudjuk elérni. Ennek a függvénynek a visszatérési értéke egy pointer, mely a processzor osztály példányára mutat. Ha például megváltoztatjuk egy csúszka értékét, akkor ezen a pointeren keresztül módosíthatjuk a processzor megfelelő publikus változóját.

A *Timer* szülőosztály beállítható időközönként meghív egy *timerCallback()* nevű virtuális függvényt. Ennek segítségével frissíthetjük a megjelenített adatot a képernyőn.

A *SliderListener* és *ButtonListener* osztályok, ahogy a nevük is mutatja, a gombok és csúszkák változásait figyelik. Ha például megnyomunk egy gombot, akkor meghívódik a *buttonClicked* függvény, melyben megírhatjuk, hogy mi történjen a gombnyomás hatására.

Az editor az egyes grafikus elemeket különböző osztályokban kezeli. Az egyik ilyen a *Graphics*, mely a háttérgrafikát valósítja meg. Ezzel tudunk rajzolni egyszerű vonalakat, téglalapot, ellipszist, írhatunk ki vele szöveget, de akár elhelyezhetünk képet is a háttérben. Kiválaszthatjuk a színeket, használhatunk egy színt vagy több szín közti átmenetet is.

Ezeket a grafikákat a *paint()* függvény rajzolja ki, mely paraméterként kap egy *Graphics* példányt. Ezt a host hívja meg, viszont van rá lehetőségünk, hogy a *repaint()* segítségével jelezzünk a hostnak, hogy újra kell rajzolni az adott területet.

A másik nagy csoport a *Subcomponents* gyűjtőnévre hallgat. Ide tartoznak az interakciót megvalósító elemek: minden felirat, csúszka, szöveges gomb vagy olyan gomb, melynek kinézete egy betöltött kép. Emellett akár nyithatunk egy kisebb ablakot is az editoron belül. Látható, hogy a JUCE rengeteg lehetőséget kínál.

Az Introjuceren belül lehetőségünk nyílik a plugin végső kinézetét megtekinteni, valamint az elemeket szerkeszteni is tudjuk. Ez nagyon hasznos tulajdonság, mivel a program lefordítása nélkül képet kapunk a végtermékről. Ebben új elemeket tudunk hozzáadni, valamint az egyes objektumok minden lehetséges paraméterét módosíthatjuk egy eszköztár segítségével.

4.2. A plugin működése

Az elkészített plugin állapotgép szerkezetű, mely az interfészen található gombokkal vezérelhető. Ezen állapotok a specifikációban előírt funkciókat valósítják meg.

Az egyes állapotokat kiszolgáló erőforrásokat működés szempontjából 3 fő részre lehet osztani:

* Jelfeldolgozás
  Ez a rész az on-line, azaz a valós időben történő jelfeldolgozást valósítja meg. Egyrészt kiszámítja a bemenőjel hangmagassággörbéjét, valamint adatokat küld a kimenetre.

* Megjelenítés
  Ezen egység feladata a kiszámított hangmagassággörbe ábrázolása, valamint egyéb adatok szemléltetése a grafikus felületen.

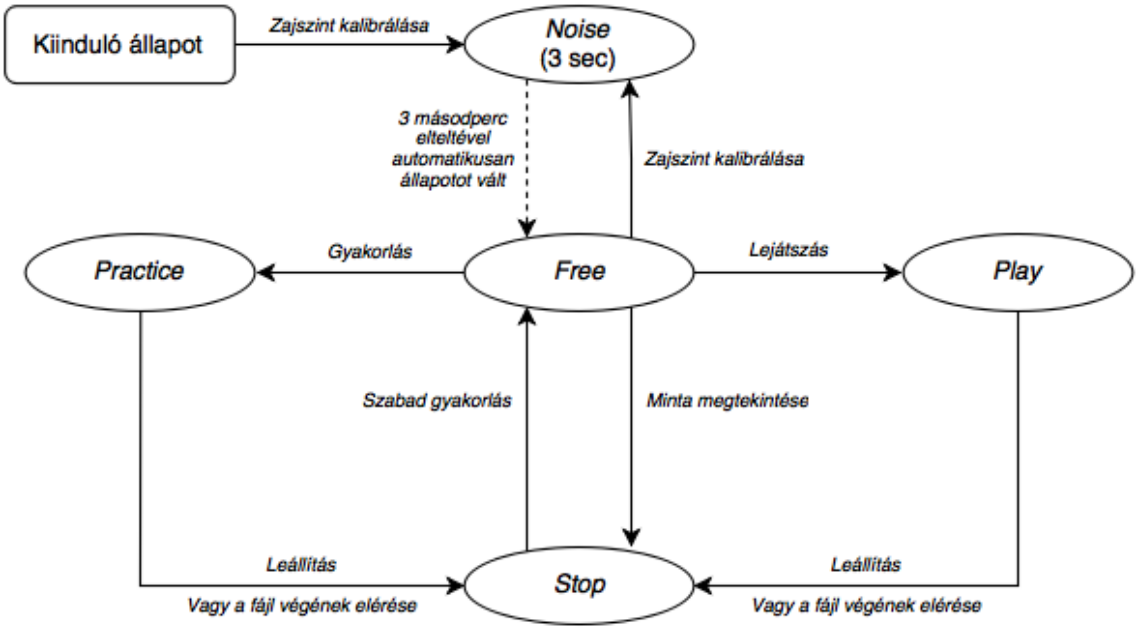
* Fájlok megnyitása
  Feladata a hangfájlok betöltése, kezelése és az off-line jelfeldolgozás. A fájl megnyitását követően kiszámítja a hangmagassággörbét az egész fájlra.

A következő fejezetek során először is megismerkedünk a program vázát jelentő állapotgép szerkezetével, majd a 3 fő funkcionális egység megvalósításával.

4.2.1. Az állapotgép

Annak érdekében, hogy a program a specifikációban előírt funkciókat végre tudja hajtani, egy állapotgépet készítettem, melyben az állapotátmeneteket a különböző gombnyomások váltják ki. A program működését, és ezáltal a vizuális- és audio kimenetet csupán az állapotok határozzák meg, tehát Moore modell szerint működik.

Az állapotgép blokkvázlata az alábbi ábrán látható.



20. ábra. A program blokkdiagramja
(a nyilakon az egyes gombok feliratai láthatóak)

Mint már említettem, az állapotátmeneteket egyrészt a gombnyomások váltják ki, másrészt pedig automatikusan is végbemehetnek, mint például a zajszint megállapítása, vagy a lejátszás befejezése esetén.

Annak érdekében, hogy ne lehessen értelmetlen állapotba irányítani a programot, bizonyos állapotokban egyes gombok inaktívvá válnak. Ha például még egyetlen fájl sincs megnyitva, a lejátszás gomb nem kattintható.

Az állapotokat *enum* típussal hoztam létre, az egész plugint vezérlő állapotváltozót pedig a processzorban tároltam el publikus változóként. Ezáltal a processzornak egyszerű hozzáférése van, illetve gombnyomás esetén az editor is könnyedén tudja módosítani.

Az állapotokat módosító függvény:

```
void changeState(FancyAudioProcessor::TransportState newState)
```

Ez paraméterként megkapja az új állapotot, és ha ez nem egyezik meg az állapotváltozó eddigi értékével, akkor át is írja azt a processzorban. Ezt követően az állapottól függően ki- és bekapcsolja (tehát megnyomhatóvá teszi, enable) a megfelelő gombokat, valamint beállít bizonyos változókat. Például a lejátszás helyét jelölő pointert visszaállítja nullára a Leállítás gomb megnyomásakor.

A nyomógombok a többi *Subcomponent* elemhez hasonlóan az Introjucer által automatikusan létrehozott objektumok. A deklarálásuk az editorban a *ScopedPointer<T>* JUCE template segítségével történik. Ezen keresztül az összes elem hasonlóan érhető el, például a szín és méret ugyanazon tagfüggvények segítségével módosítható.

Állapotváltást a processzor is kezdeményez. Ilyen automatikus állapotátmenet, amikor a lejátszás során elértük a fáj végét, és stop állapotba kerül a rendszert. Ekkor csak egy flaget billent be, majd az editor ezt észlelve hívja meg a *changeState()* függvényt.

4.2.2. Jelfeldolgozás

4.2.2.1. *processBlock* függvény

Ez a host által periodikusan, minden új mintavételi ablak esetén meghívott függvény. Itt történik a tényleges jelfeldolgozás, valamint a kimenet írása is. Attól függően, hogy mely állapotban van a plugin, eltérő feladatokat hajt végre. A függvény maga egy *switch – case* szerkezet, mely az állapotváltozó alapján választja ki a helyes műveletet.

Vegyük ezeket sorra:

* *Noise*:
  Ebben az állapotban folyamatosan olvassa a (mikrofon) bemenetről érkező adatokat és megkeresi ennek a maximumát. Ez a művelet könnyedén megvalósítható a buffer *getMagnitude()* tagfüggvényének segítségével. Az editor timere pár másodperc eltelte után állapotot vált, és az addig beérkezett adatok maximumát tekintjük zajküszöbnek. Ezt a *noiseLevel* nevű változóban tároljuk el, melynek értéke a „Zajszint kalibrálása” gomb lenyomása után nullázódik, így változó zajszint esetén újrakalibrálhatjuk a rendszert.

* *Play*:
  Ekkor a processzor feladata csupán a kimeneti buffer írása. Először is a kinullázza a buffert, majd egyenként bemásolja az adatokat attól függően, hogy a referencia valamint a kísérősáv közül melyek vannak kijelölve. Ha egyik sincs kijelölve, akkor a kimeneten nem fogunk hallani semmit.
  Az editor által megnyitott wav fájlból *AudioFormatReader* osztály *read()* tagfüggvénye segítségével könnyedén kiolvashatjuk és a bufferhez hozzáadhatjuk a soron következő blokkot. A lejátszás pozícióját a *playPos* nevű változóval követjük, úgy hogy minden beolvasás után hozzáadjuk a blokkméretet. Ha ezzel elértük a fájl végét, akkor bebillentünk egy flaget, mellyel jelezzük az editornak a *Stop* állapotba váltási szándékunkat.

* *Stop*:
  A függvény ekkor tulajdonképpen semmit sem csinál.

* *Free*:
  Ez a mód a szabad gyakorlást teszi lehetővé. Tehát ekkor nem viszonyítunk referenciához és nincs kísérősáv, csupán a mikrofon bemenőjelét elemezzük. Először is megvizsgáljuk, hogy a beolvasott blokknak van-e nagyobb amplitúdójú eleme, mint a zajküszöb (noiseLevel), tehát, hogy hasznos jel vagy zaj. Ha énekről van szó, akkor publikus tagváltozóként létrehozott *SNACclass* példány segítségével

kiszámoljuk az adott ablakra a hangmagasságot és eltároljuk a *freePitch* nevű standard tömb osztályban.
A megjelenítés során az ábrát mindig balra léptetjük, majd jobboldalra bekerül a legújabb adat. Így egy mozgó ábrát kapunk, mely folyamatosan frissül. Hogy ezt effektíven tudjuk megtenni, és ne kelljen mindig az egész tömböt másolni, cirkuláris tömbkezelést alkalmazunk. Egy pointerrel – *freePos* – megjelöljük a tömb kezdő / végpontját és ide szúrjuk be az új adatot, majd a pointert eggyel mozgatjuk. Ezt a pointert természetesen a megjelenítésnél is fel kell majd használnunk.

* *Practice*:
Ebben az állapotban az előzőhöz hasonlóan járunk el. Folyamatosan dolgozzuk fel az adatot, viszont ezt el is szeretnénk menteni, így a *recPitch* tömben tároljuk el azokat.
Emellett igény szerint a referenciafelvételt vagy a kísérősávot is le kell játszanunk. Ezt úgy oldottam meg, hogy a *switch – case* szerkezet ezen blokkjának végére nem írtam *break*-et, ezáltal ott folytatódik a programvégrehajtás a következő szakasszal, mely a *Play*-t valósítja meg. Ez viszont pontosan a kívánt feladatot végzi el.

A processzor ezeken kívül rendelkezik még néhány tagváltozóval. Ezek többségében flagek, melyekkel például a kísérősáv vagy a referencia lejátszásának szükségességét jelzi az editor.

4.2.2.2. SNACclass osztály

A hangmagasság megállapítását végző SNAC függvényt egy osztályként implementáltam. Ennek legfőbb oka az volt, hogy a számításhoz két extra tömbnek is helyet kell foglalnunk: az egyik a normálást megvalósító függvényt, a másik pedig a korrelációt tárolja el. Hogy ne kelljen minden egyes ablak számításánál ezeket a tömböket újrafoglalnunk, az osztály tagváltozóivá tettem őket. Így csak egyetlen példányt kell deklarálnunk, melylyel megvalósítható az összes számítás.

A hangmagasság kiolvasását további függvények is segítik. Ezek feladata az egyes formátumok – frekvencia, MIDI, zenei hang – közti konverzió elvégzése.

Az FFT kiszámítása a Cooley – Turkey algoritmus [18] nyilvánosan elérhető *C++* implementációjának segítségével történik [19].

A hangmagasságot kiszámító tagfüggvény tulajdonképpen a 2.4.1.4 szakaszban leírtak *C++* implementációja.

4.2.3. A megjelenítés

4.2.3.1. paint() függvény

A hangmagassággörbék kirajzolása háttérgrafikaként, azaz a *Graphics* osztály segítségével történik. Az *AudioProcessorEditor* bázisosztály virtuális tagfüggvénye, a *paint()* rajzolja fel a teljes grafikát. Ezt a host hívja meg az editor elindításakor, az ablak újraméretezésekor, stb. A *repaint()* függvény segítségével jelezhetjük a host számára, hogy az editor ablakát, vagy annak bizonyos részeit újra kell rajzolni.

A görbék felrajzolására és görgetésére két alapvető lehetőségünk van. Egyrészt eltárolhatnánk az egész ábrát például egy bitképként, és ezt minden időpillanatban, illetve görgetés esetén eltolnánk a megfelelő irányba. Ez rengeteg adatmásolással járna, valamint a bitkép kezelése is eléggé bonyolult lenne.

A másik módszer az, hogy csak a görbe elemeit tároljuk el, és ezt, mint egy függvényt, a megadott intervallumok között ábrázoljuk. Így minden időpillanatban, vagy görgetés esetén újra kell rajzolnunk az ablakot. Ez a megoldás sokkal egyszerűbb és gyorsabb is a bitkép alkalmazásánál.

A görbéket az editor felületének szinte egészét kitöltő ablakban ábrázoljuk. Az ablak két szélén zongorabillentyűzet részletet találunk. Ahelyett, hogy csupán a zenei hangokat írnánk ki táblázatszerűen, a billentyűk segítségével sokkal átláthatóbbá tesszük az ábrázolást. Természetesen a billentyűkön megtalálhatóak a hangok jelei is, ezzel segítve azokat, akik esetleg nem ismerik a zongora hangjait. A képeket az Introjucerben „cache-elhetjük" be. Ez azt jelenti, hogy a kép betöltése után bináris állományként bekerül a forráskódba (*pluginEditor.cpp*).

A két kép közötti fehér felületet szürke sávokkal bontjuk fel, annak érdekében, hogy az ablak közepén is tisztán látszódjon mely hangról van szó. A törzshangokat a fehér, a felemelt hangokat a szürke sávok jelzik. Erre a felületre rajzoljuk fel majd a hangmagassággörbéket.

Az ablak határait *w_left, w_right, w_top, w_bottom* szimbólumokkal definiáljuk az áttekinthetőség érdekében.

A *paint()* függvénnyel mindig csak az ablak közepét rajzoltatjuk újra, hiszen csak ez változik. A függvény része egyrészt a JUCE által automatikusan generált fehér háttér, valamint a szürke sávok, melyek *rectangle* objektumok.

Az általam hozzáadott kód tulajdonképpen egy, a *processBlock*-hoz hasonló *switch – case* szerkezet, mely a processzor állapotváltozója alapján dönt. A kirajzolás a későbbiekben ismertetett *drawGraph* függvény segítségével történik, melynek paraméterként többek között a kirajzoltatni kívánt tömböt és a tömbön belüli kezdő és végpontot kell megadni:

* *Free* állapot
  A *freePitch* tömböt kell kirajzolni. A tömböt inicializáláskor és a Szabad gyakorlás gomb megnyomásakor feltöltjük csupa -1 értékkel. Mivel a MIDI érték soha sem lehet -1 (ha a SNAC függvény 80 Hz-nél kisebb frekvenciát talál, akkor 0 értékkel tér vissza), ezáltal ha nincs teljesen feltöltve a tömb, tudjuk meddig kell

kirajzolni.
Tehát először megvizsgáljuk a *freePos* által mutatott elem értékét. Ha ez -1, akkor a tömböt 0-tól *freePos*-1 indexig ábrázoljuk. Ha nem, akkor *freePos*-tól *freePos*-1 indexig. (*freePos* mutat a legrégebben beírt elemre). Ez cirkuláris tömbkezelést igényel, melyet a *drawGraph* függvény valósít meg.

* *Stop* állapot
Ekkor a *wavPitch* tömböt ábrázoljuk, mely a referenciajel hangmagassággörbéjét tartalmazza. Ha a tömb kisebb, mint az ablak szélessége, akkor 0-tól a végéig ábrázoljuk, ha nem, akkor a *scrollPos*-tól kezdve, amennyi kifér az ablakra.

* *Practice* állapot
Mivel ez nagyon hasonlít a következő állapothoz, ezért csak egyetlen flag, az *isPractice* bebillentése a feladata. Ezen szakasz végén nincs *break* utasítás, tehát a programvégrehajtás folytatódik a következő, *Play* állapottal.

* *Play* állapot
Mivel ekkor szinkronban kell görgetnünk az ábrát a lejátszott hanggal, ezért a processzorban lejátszást vezérlő *playPos* segítségével irányítjuk a folyamatot. Mivel ez a wav fájlban jelöli a lejátszás helyét, és mi az adatokat 1024-es blokkokban dolgoztuk fel, ezért ezt a változót leosztjuk 1024-gyel, hogy címezni tudjuk a tömböt, ezt nevezzük *tempPos*-nak.
Ha a görbe kisebb, mint az ablak szélessége, akkor a *Stop* állapothoz hasonlóan ábrázoljuk. Azért hogy lássuk, hogy hol tarunk a görbén, egy piros vonalat futtatunk az ablakon balról jobbra. Ennek helyét jelen esetben egyszerűen a *tempPos* segítségével számítjuk ki.

Ha görbe hosszabb, mint az ablak szélessége, akkor folyamatosan léptetnünk kell az ábárát. A kezdőpontot a *scrollPos* értéke határozza meg. Az első fázisban, annyit ábrázolunk a görbéből, amennyi a kezdőponttól az ablakra felfér, és csak a vonalat futtatjuk az ablak közepéig.
A második fázisban a vonalat megállítjuk az ablak közepén, és az egész görbét léptetjük balra. Ezt egészen addig tesszük, míg jobb oldalon megjelenik a görbe utolsó pontja. Ezt követően az utolsó fázisban a vonalat kifuttatjuk a képernyő jobb oldalára.

Ezzel a gondolatmenettel meghatározhatjuk a tömb ábrázolni kívánt kezdő és végpontját, majd az *isPractice* flagtől függően a *wavGraph* tömb mellett megjelenítjük a *recPitch* tömböt is.

4.2.4. *drawGraph* függvény

Ennek a függvénynek a célja, hogy az editor ablakát a kapott tömb egy meghatározott szakaszának megfelelő görbével töltse ki.

A deklarációja az alábbi:

```
void drawGraph(Graphics& g, int start, int stop, std::valarray <float>& p,
               bool ifcolour = false, int offs = 0);
```

Mint láthatjuk, a paraméterek közt egy *Graphics* osztályt kap meg referenciaként. Ennek segítségével rajzolja fel a görbét.

Megkapja az ábrázolni kívánt tömböt, valamint a tömbben a kezdő és végpontot. Ezek alapján egy *for* ciklussal végigjárja a tömböt úgy, hogy az index érték kiolvasása előtt modulo műveletet végez a tömb hosszával. Így a tömb utolsó eleme után az első elemét olvassa ki, tehát cirkulárisan olvassa a tömböt. Egy JUCE vonal osztály példányt hozunk létre minden pont között. Ennek a vonalnak megvizsgáljuk a meredekségét, és ha ez olyan nagynak bizonyul, melyet valós emberi hanggal nem lehetne elérni, akkor azt a vonal darabot nem ábrázoljuk. Ha a vonal az ablak tartományain kívül esik (túl alacsony, magas), akkor sem ábrázoljuk.

Mivel a görbét ilyen vonaldarabkákból állítjuk össze, lehetőségünk van minden egyes szakasznak az ideális hangtól való eltérését színkóddal jelezni. Ez úgy történik, hogy kiszámítjuk a vonal két végpontjának megfelelő MIDI érték átlagát, és megvizsgáljuk a *MIDI_2_cents* függvény segítségével, hogy mekkora az eltérés centben. Ha ez kisebb mint 10, akkor zöld, ha kisebb mint 25 akkor sárga, egyébként piros színnel ábrázoljuk. Lehetőségünk van ezt a funkciót az *ifColour* paraméterrel kikapcsolni, ekkor a függvényhívás előtt beállított színnel történik az ábrázolás.

Az utolsó paraméter egy előjeles ofszet eltolást tesz lehetővé, hogy a feldolgozásból adódó késleltetéseket korrigálni tudjuk és szinkronban lássuk a felénekelt és a referenciajelet.

4.2.4.1. timerCallback függvény

Ez a timer osztály által, beállított időközönként meghívott virtuális függvény. Én a teljes editor képernyő folyamatos frissítésére használtam fel.

A *timerCallback* egyrészt frissíti a hangmagasságot, centeket, zajszintet megjelenítő feliratokat és slidereket. Ebben az esetben a slidert nem hagyományos módon használjuk, csupán a megjelenítés a feladata. A zajszint változásáról valamint a hangmagasság centben mért eltéréséről könnyen értelmezhető vizuális képet ad.

A további elvégzett feladatok már állapotfüggőek.

A zajszint kalibrálásánál megjeleníti a „Zajszint kalibrálása” szöveget a végén egy, kettő vagy három ponttal, mellyel a jelzi, hogy a művelet folyamatban van. Ezt az órajel leosztásával könnyen megtehetjük. Ha mind a három pontot kirajzolta, akkor a kalibrálás befejeződik és automatikusan átváltunk a *Free* – szabad gyakorlás állapotba.

A hangmagassággörbe kirajzolását igénylő állapotok esetén (*Free, Practice, Play*) meghívja a *repaint()* függvényt. Ezzel jelzi a host számára, hogy az editor képernyőt újra kell rajzolni. Ez nem eredményezi ugyan a *paint()* függvény azonnali meghívását, de a késleltetés elhanyagolhatóan kicsi lesz.

Végül figyeli a processzor *Stop* állapotba váltást jelző flagét. Ha ez be van billentve, akkor a plugint ebbe az állapotba vezérli, és kinullázza a flaget. A processzor akkor kér leállítást, amikor a lejátszás végére értünk, így nem időkritikus művelet, ezért valósítottam meg egyszerűen a *timerCallback* segítségével.

4.2.5. Fájlok megnyitása

Mint már említettem, a nyomógombok többsége az állapotátmeneteket irányítja. Emellett megnyomás esetén nullázhatnak egyes változókat, mint például a *noiseLevel*, vagy a görgetőgombok esetében növelik vagy csökkentik a lejátszás kezdetét jelentő *scrollPos* változót.

A két, hangfájl megnyitását kezdeményező gomb függvénye viszont jelentősen eltér a többitől. A gomb megnyomása után felugrik egy böngésző ablak, melyben kiválaszthatjuk a betölteni kívánt fájlt. Ezt beépített JUCE osztályok segítségével oldottam meg.

Első lépésként a programot *Stop* állapotba vezéreljük, illetve az összes gombot letiltjuk. Ezután létrehozunk egy *FileChooser* példányt, melynek paraméterként megadhatjuk a párbeszédablak címét, valamint azt, hogy milyen fájlok közül választhatunk a böngészőben. Ezt követően meghívjuk a *browseForFileToOpen()* tagfüggvényét, mely sikeres fájlmegnyitás esetén igaz értékkel tér vissza.

Ezután a *File* segítségével – mely a JUCE-ban a könyvtárak és fájlok reprezentációja – kiolvassuk a kiválasztott elemet. Majd tudván, hogy wavról van szó, ezt formátumot kezelő osztály, a *WavAudioFormat*-on keresztül egy olvasó pointert állítunk rá. Ezt a pointert adjuk oda processzor *AudioFormatReader* példányának, amely a zenefájlok olvasására alkalmas.

```
FileChooser chooser("Valasszon egy file-t a lejatszashoz",
                    File::nonexistent,
                    "*.wav");

if (chooser.browseForFileToOpen())
{
      File file(chooser.getResult());
      myproc->backingFormatReader =
      WavAudioFormat().createReaderFor(file.createInputStream(), false);
}
```

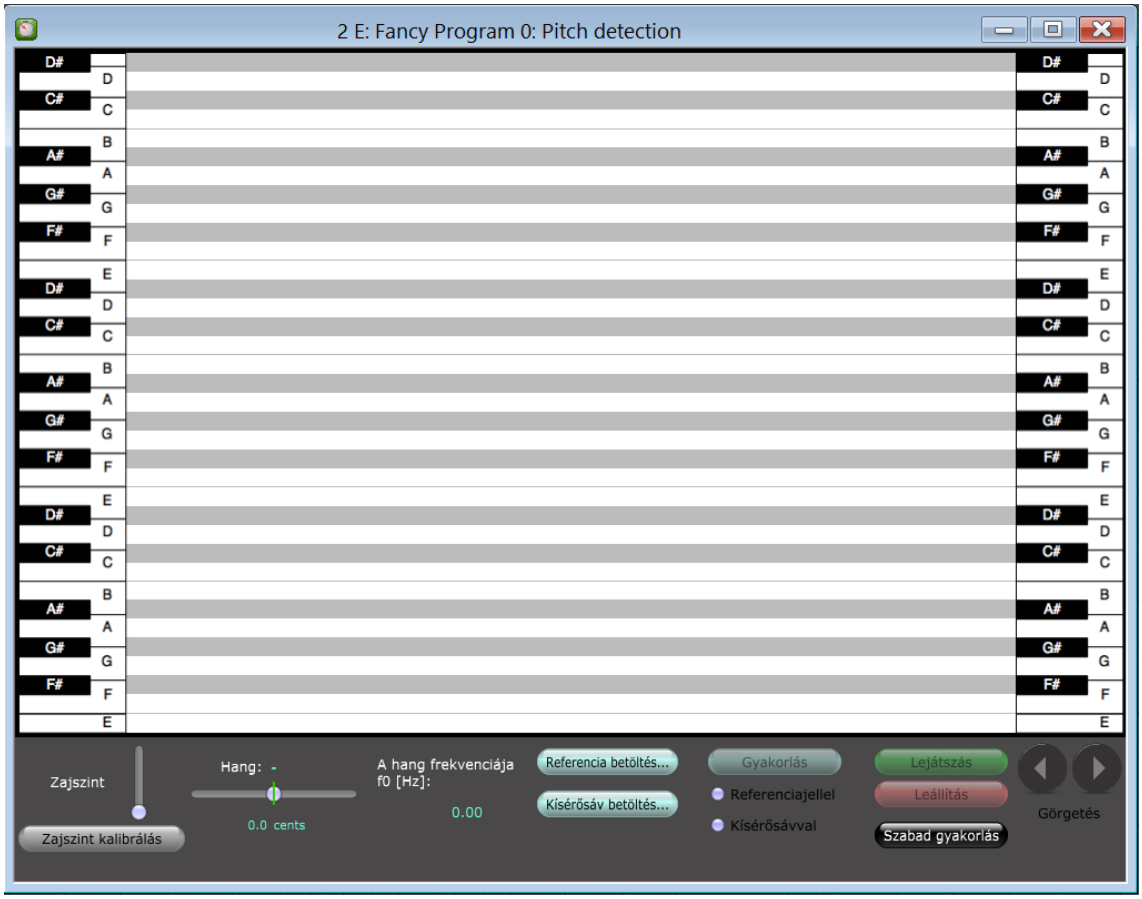
A processzorban a *read()* tagfüggvény segítségével lehetőségünk van a fájl egy szakaszát akár a bufferbe is beolvasni. Így elegendő csupán egy változó (*playPos*), melyben eltároljuk, hogy éppen hol járunk a fájlban és ezzel meg is oldottuk a lejátszás kérdését.

Itt is látszik, hogy a JUCE beépített függvényeivel mennyire egyszerűen tudjuk a felhasználóval való interakciót megvalósítani, illetve a fájlok (különösen a hangfájlok) kezelését megoldani.

A hangfájl megnyitása után a referenciaminta esetében még meg is kell állapítanunk annak hangmagassággörbéjét. Ez a *processBlock* működéséhez hasonlóan történik: 1024 minta hosszúságú szakaszokra bontjuk a teljes fájlt, majd ezekre egyenként kiszámoljuk a hangmagasságot a *SNACclass* példány segítségével. Az értékeket a *wavPitch* tömbben tároljuk el.

5. Az elkészült plugin értékelése, tesztelése

Az elkészült plugin felhasználói felülete az alábbi ábrán látható.



21. ábra. Az elkészült plugin felhasználói felülete

A plugin a VSTHost ingyenesen elérhető programban [14] került megnyitásra (az ábrán ebből csupán a plugin interfésze tekinthető meg).

Mint látható, a plugin legnagyobb részét a 3 oktávnyi ábrázoló felület tölti ki. Az egész ablak mérete akkora, hogy a manapság használatos monitorok könnyedén meg tudják jeleníteni, de legyen elég hely az ábrázoláshoz, és a kezelőfelület kényelmes használatához.

Jelen állapotban nincs megnyitva egyetlen referencia, vagy kísérősáv sem, így a Gyakorlás, Lejátszás és Leállítás gombok inaktívak (szürke színűek).

A Zajszint és a Hang címkék mellett található csúszkákat, mint már említettem, nem rendeltetésszerűen használom. Ezek az adatok bevitele helyett a kijelzést valósítják meg. Az ideális hangtól való eltérést kijelző csúszka csupán extra kényelmi szempont volt, viszont a zajszinthez tartozó a felhasználókkal való tesztelés során került bele a programba.

Gyakran előfordult, hogy zajkalibrálás esetén egy éles, hangos zaj hallatszott, ami miatt a program zajküszöbe nagyon magas lett, majd amikor az énekes megpróbált felénekelni

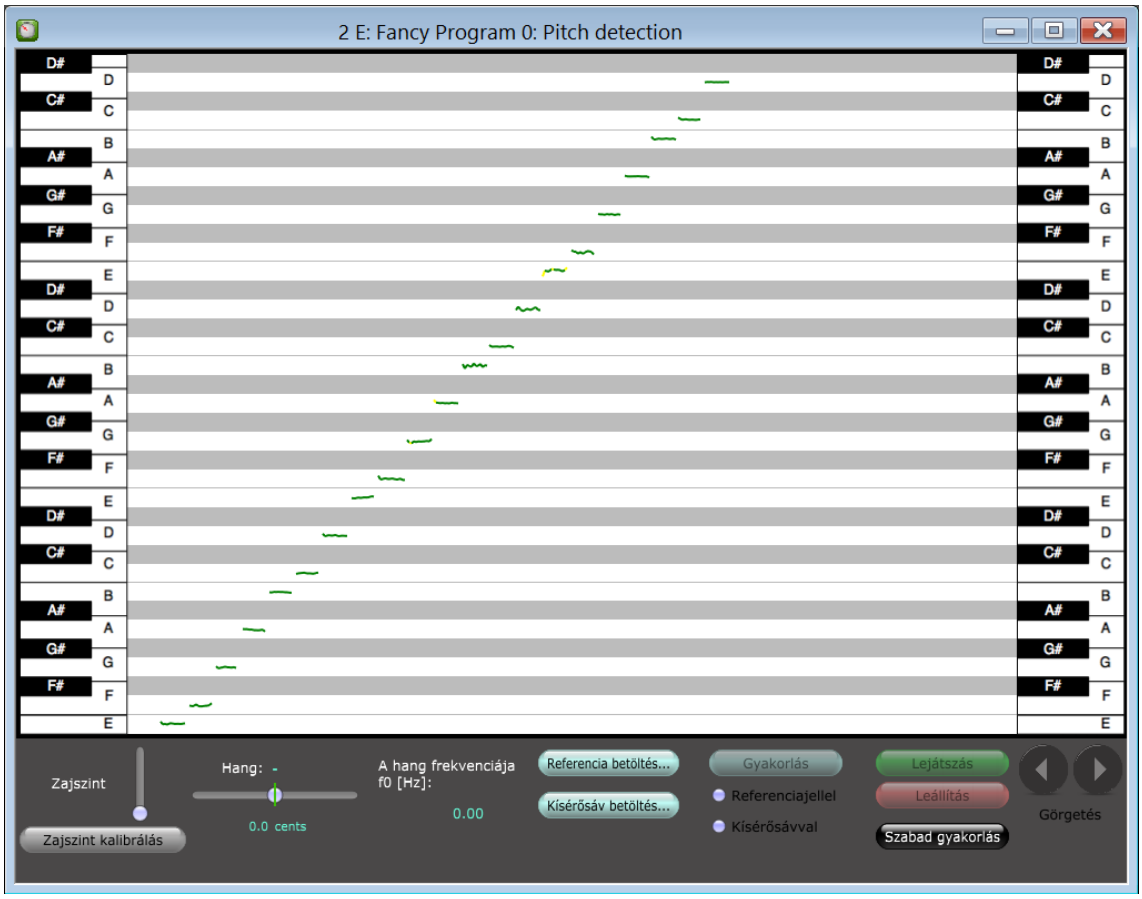
valamit, a program nem észlelt semmilyen valós hangmagasságot. Nem tudván a program működésének hátteréről, ennek a problémának a megoldása sokaknak gondot jelentett.

Ezért egy olyan vizuális elemet adtam hozzá, mely megkönnyíti a zajszint kalibrálásának nyomon követését. Ennek nem a precíz zajszint kijelzése a célja, csupán referenciát jelent, hogy valóban a zajszintet mértük-e meg, vagy csak egy hangos tranziens csúcsot.

5.1. Tesztjelek

A program működését valós- valamint szintetizátorral előállított tesztjelekkel is vizsgáltam.

Az első ilyen teszt eredménye látható az alábbi ábrán.



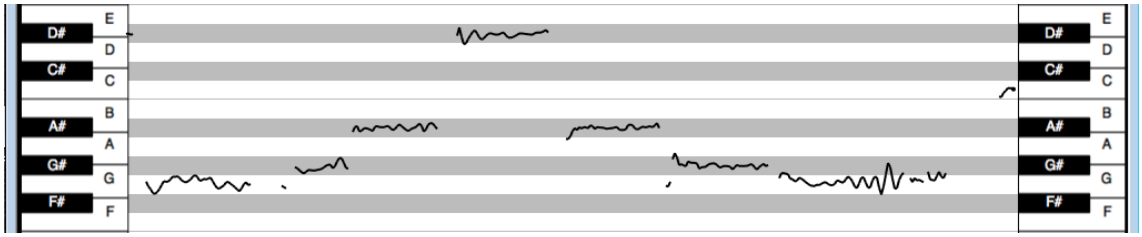
22. ábra. a plugin tesztelése törzshangokkal

Itt a tesztjel egy szintetizátorral lejátszott skála volt. Mint látható a skála hangjai csupa törzshangok voltak (tehát minden fehér billentyűnek megfelelő hang).

Ezzel a teljes, átlagos ember által kiénekelhető 3 oktávot szerettem volna tesztelni. Mint látható, a program a teljes skálán stabilan detektálja a hangmagasságot. A hangokat egymás után külön-külön meglehetősen gyorsan játszottam le, ennek ellenére a megállapított görbe szinte semmilyen tranzienst nem mutat az egyes hangok között.

A program késleltetése mindösszesen *10 ms* körül van, melyet csupán a kijelzőt szemlélve szinte észre sem veszünk. Ez a késleltetés az ablakméretből és a feldolgozási sebességből adódik.

A második tesztfájl a Himnusz szólóének előadásban. A fájl *.mp3* formátumban volt elérhető, ezt konvertáltam át egy *.wav* fájllá.

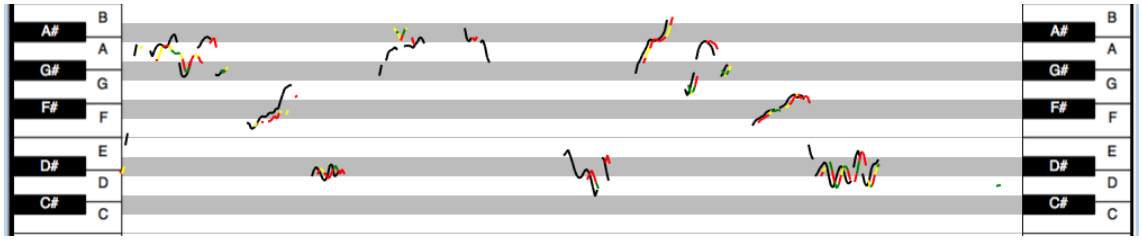


23. ábra. a referenciaként betöltött Himnusz első sora

Ezen tesztfájl egy hivatásos énekes stúdió felvétele. Lényegében egyszerű, hosszan kitartott tiszta hangokból áll. Mint ahogyan ez az ábrán is megfigyelhető, az egyes hangok határai jól elkülöníthetők.

Ezzel a teszttel kívántam szemléltetni, hogy a program alkalmas szinte bármilyen hangfelvétel referenciaként történő feldolgozására, legyen az általunk felvett, vagy stúdióban feldolgozott.

A harmadik teszt a program dinamikusabb tulajdonságait igyekezett tesztelni. Ebben egy sok hajlítást és vibratot tartalmazó referenciajelet próbált a tesztelő énekes lekövetni.



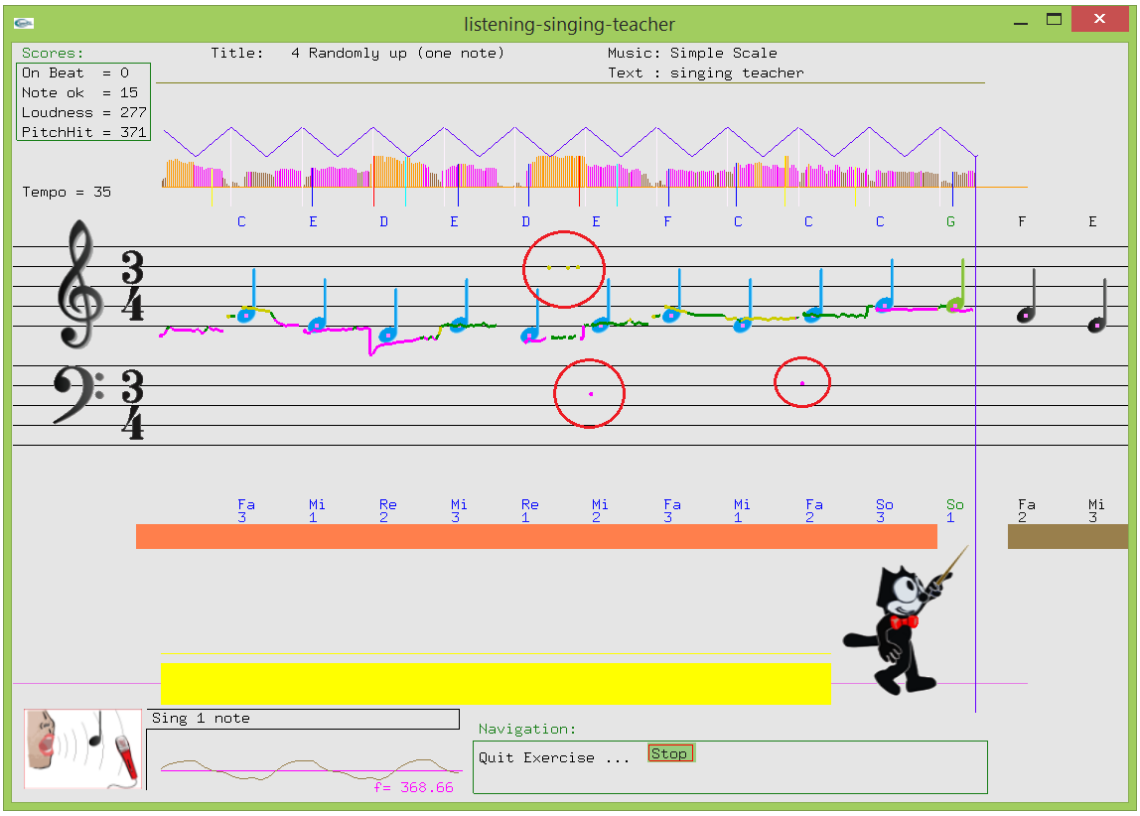
24. ábra. Dinamikus tulajdonságok vizsgálata
(referenciajel: fekete, élő ének: színes)

Elsőre talán értelmezhetetlennek tűnhet ez ábra. Ennek oka, hogy az énekes a dalszöveget minél inkább érthetően, artikulálva próbálta elénekelni, így az egymást váltó különböző magánhangzók egészen hullámzó hangmagasságot produkáltak.

Viszont a visszaénekelt hang többnyire pontosan követi a referenciát. Az ábra jobb oldalán láthatjuk a vibrato effektust. Ez mind a referencia, mind pedig a visszaénekelt hang esetében precízen visszaadja a hallottakat.

5.2. Összehasonlítás egy ingyenesen elérhető szoftverrel

A Listening Ear Trainer programnak találtam egyedül ingyenesen elérhető verzióját, így az általam elkészített plugint ezzel vetettem össze.



25. ábra. A Listening Ear Trainer ingyenesen elérhető próbaverziójának felülete

Ahogyan az a fenti ábrán látható, ez a szoftver a zenei alapok megtanítására törekszik. Rengeteg, a fentihez hasonló lecke található a program teljes verziójában, melyek a kotta olvasásától kezdve a különböző skálák kiénekléséig sok mindenre megtanítják a felhasználót.

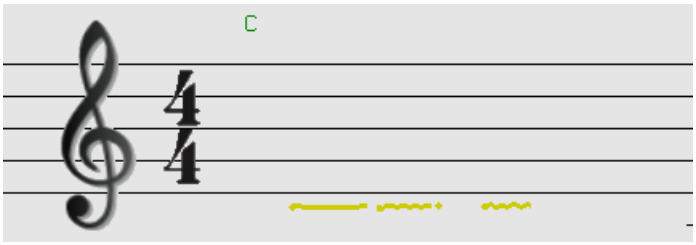
A program hangmagasság-detektáló algoritmusa természetesen nem tökéletes. Gyakran fellép az oktávhiba jelensége, mely az ábrán pirossal bekarikázott részeken figyelhető meg. Emellett a program használata során a késleltetés is feltűnő.

A két program között pontosság tekintetében nincs jelentős eltérés. A Listening Ear Trainer szoftvert szintetizátor jelével tesztelve, a kapott érték 261.81 Hz, az általam készített plugin esetében ez 261.82 Hz. A tesztjelet gitárhangolóval (analóg hardver eszköz) is megvizsgáltam, mellyel a mért érték 261.8 Hz volt.

Dinamikus tulajdonságok tekintetében viszont jelentős különbségeket fedezhetünk fel. Egyrészt, már a program első használatakor észrevehető a késleltetés, ám ez nem akkora,

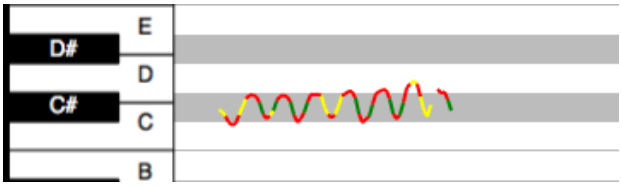
hogy zavaró lenne. Másrészt az is feltűnő, hogy a kereskedelmi program a hangmagasság gyors változásait nem tudja pontosan követni.

Annak érdekében, hogy ezt bemutassam, a két programot vibratoval díszített énekhanggal teszteltem.



26. ábra. Az ingyenesen elérhető szoftver dinamikus tulajdonságainak vizsgálata

A Listening Ear Trainer esetében a vibrato hullámzó hatása alig látható.



27. ábra. Az általam készített plugin dinamikus tulajdonságainak vizsgálata

Ahogyan ez a 27. ábrán megfigyelhető, az általam készített alkalmazás a dinamikus változásokat sokkal pontosabban követi.

6. Összefoglalás

A szakdolgozat célja egy hatékonyan működő és egyszerűen használható énektudást fejlesztő alkalmazás elkészítése volt. Véleményem szerint ezt sikerült megvalósítani. Egy olyan VST plugin készült el, mely egy igen stabil hangmagasság detektálási módszeren alapul, és az átlagos énekhangot jelentő 3 oktávnyi intervallumon belül tökéletesen működik.

A hangdetektálási algoritmus megalkotása során kompromisszumot kellett kötni a számítási sebesség és pontosság között. Egyaránt fontos volt, hogy a program kellően dinamikus legyen, ugyanakkor a hangmagasságot pontosan találja meg. A legáltalánosabb probléma a vizsgált algoritmusokkal az oktávhiba jelensége volt. Ezt a különböző módszerek eltérő eszközökkel próbálták kiküszöbölni, de általánosan igaz, hogy mind a számításigény jelentős növekedését eredményezte.

A választásom a SNAC függvényre esett, mivel a vizsgált módszerek közül ez bizonyult a legstabilabbnak. Ez az algoritmus volt képes az egyre halkuló tesztjel esetében a leghosszabb ideig detektálni a hangmagasságot, valamint oktávhibát sem mutatott. Igaz, a számításigénye meglehetősen nagy a többihez képest, ám nem akkora, hogy egy mai átlagos számítógép ne tudná futtatni.

Az alkalmazást VST plugin formájában implementáltam, a JUCE fejlesztőkörnyezet segítségével. Ez megkönnyítette a programozást, mivel nem kellett a hangkártya kezelésével foglalkozni, valamint a JUCE segítségével könnyedén tudtam a felhasználói interfészt létrehozni, hangfájlokat beolvasni.

Az elkészült plugin a vártaknak megfelelően működik. A tesztelés során apróbb változtatásokat hajtottam végre, hogy a használata még egyszerűbbé váljon. Ilyen volt például a zajszintet megjelenítő csúszka hozzáadása, valamint a zajkalibrálás automatikus befejezése.

Különböző teszteket végeztem el, melyekkel az implementált algoritmus pontosságát és dinamikusságát vizsgáltam. A program a teljes 3 oktávon stabilan megtalálja a beolvasott jel hangmagasságát, ugyanakkor a gyorsan változó frekvenciájú vibrato effektet is ábrázolni tudja.

Az általam készített plugin, a többi hasonló szoftverrel ellentétben, nem MIDI referenciával dolgozik. Egy wav formátumú egyszólamú hangfájlt, például egy énekhangot olvas be és ezt elemzi. Ezáltal lehetővé teszi a pontos és tiszta éneklés gyakorlásán túl a valós énekben fellelhető apró díszítések, hajlítások, vibrato kielemzését és gyakorlását is.

További fejlesztési cél a program optimalizálása. Ez elsősorban a folyamatosan futó hangmagasság-detektálási algoritmus gyorsítását jelenti. Az FFT megvalósítására léteznek gyorsabb, gazdaságosabb algoritmusok nyilvánosan elérhető könyvtárak formájában, melyek használata gyorsítaná a számítást. Emellett valószínűleg egy ügyes megoldással közvetlenül a frekvenciatartományban is képesek lehetnénk megállapítani a helyes hangmagasságot.

Esetleg több kis számításigényű algoritmus közös döntésével lecsökkenthetnénk a számításigényt, és ugyanakkor kiküszöbölhetnénk az oktávhibát is.

A plugin maga is további funkciókkal bővíthető, mint például egy kiválasztott szakasz folyamatos gyakorlása (loop alkalmazása). Annak érdekében, hogy a program ne csak az átlagos felhasználó hangterjedelmét fedje le, hanem a mélyebb és magasabb oktávokon is működjön (például operaénekesek számára is használható legyen), egy dinamikusan változó kijelzőablakot lehetne megvalósítani.

A referenciahang-nélküli gyakorlás megkönnyítésére metronóm funkciót lehetne beépíteni. Ehhez a kapott hangfájlon tempódetektálást kell elvégezni.

Összegzésül elmondható, hogy ez elkészült program előrelépést jelent az énektudást fejlesztő szoftverek körében. A valós énekfájlok elemzésével, valamint a könnyű használatával az amatőr-, a hivatásos énekeseket valamint az énekoktatást egyaránt segítheti.

Irodalomjegyzék

[1] D. Levitin, This is Your Brain on Music: The Science of a Human Obsession, New York: New York: Plume, 2006.

[2] S. Clift és G. Hancox, „The perceived benefits of singing” *The Journal of the Royal Society for the Promotion of Health,* 4. kötet, 12. szám, 248-256 oldal, 2001.

[3] © 2015 Cantovation Ltd., „Sing & See” [Online]. Available: http://www.singandsee.com/. [Hozzáférés dátuma: 3 október 2015].

[4] „Listening Ear Trainer” AlgorithmsAndDatastructures, [Online]. Available: http://www.listening-ear-trainer.com/. [Hozzáférés dátuma: 5 december 2015].

[5] A. J. M. Houtsma, „Pitch Perception” in *Hearing*, 8. fejezet, Academic Press, 1995, 267-291. oldal

[6] E. Terhardt, „Calculating virtual pitch” *Hearing Research,* 2. kötet, 1. szám, 155–182 oldal, March 1979.

[7] F. A. Kuttner, „Prince Chu Tsai-Yü's Life and Work: A Re-Evaluation of His Contribution to Equal Temperament Theory” *Ethnomusicology,* 2. kötet, 19. szám, 163–206. oldal, 1975.

[8] International Organization for Standardization, *ISO 16:1975 Acoustics -- Standard tuning frequency (Standard musical pitch),* 1975.

[9] D. Loeffler, „Instrument Timbres and Pitch Estimation in Polyphonic Music” in *MSc thesis,* Georgia Tech, 2006.

[10] P. McLeod, Fast, Accurate Pitch Detection, University of Otago: PhD thesis, 2008.

[11] R. W. S. Lawrence R. Rabiner, Digital Processing of Speech Signals, Prentice-Hall, 1978.

[12] F. Gergely, „Automatikus hangmagasság-korrekciós rendszer létrehozása” *Híradástechnika,* 2. kötet, 33-38. oldal, 2011.

[13] P. d. l. Cuadra, „PITCH DETECTION METHODS REVIEW” [Online]. Available: https://ccrma.stanford.edu/~pdelac/154/m154paper.htm. [Hozzáférés dátuma: 20 október 2015].

[14] H. Seib, „VSTHost” 25 június 2013. [Online]. Available: http://www.hermannseib.com/english/vsthost.htm. [Hozzáférés dátuma: 16 október 2015].

[15] „JUCE” software © 2015 ROLI Ltd., [Online]. Available: http://www.juce.com/. [Hozzáférés dátuma: 26 október 2015].

[16] Wikimedia Foundation, Inc., „Wikipedia” 20 November 2015. [Online]. Available: https://en.wikipedia.org/wiki/JUCE. [Hozzáférés dátuma: 8 december 2015].

[17] © 2015 ROLI Ltd., „Tutorial: Synthesis with level control” 2015. [Online]. Available: http://www.juce.com/doc/tutorial_synth_level_control. [Hozzáférés dátuma: 29 október 2015].

[18] J. W. Cooley és J. W. Tukey, „An algorithm for the machine calculation of complex Fourier series” *Mathematics of Computation,* 19. szám, 297–301. oldal, 1965.

[19] Rosetta Code, „Fast Fourier transform” [Online]. Available: http://rosettacode.org/wiki/Fast_Fourier_transform#C.2B.2B. [Hozzáférés dátuma: 3 november 2015].

[20] M. Rúzsa, „Himnusz” MTVA, 2013. [Online]. Available: https://www.youtube.com/watch?v=Ybz8x9rQrRo. [Hozzáférés dátuma: 5 december 2015].

Ábrajegyzék