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

Valós idejű akkordfelismerés Android platformon

Szakdolgozat

Készítette
Rozmán Péter

Konzulens
Dr. Bank Balázs

2018. december 10.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Rozmán Péter*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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 autentikált 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.

Budapest, 2018. december 10.

Rozmán Péter
hallgató

Kivonat

Az okostelefonok tömeges elterjedése olyan lehetőségeket adott a felhasználók kezébe, amilyeneket korábban el sem tudtak képzelni. A mai, modern készülékek kameráik, mikrofonjuk és számos különféle érzékelőjük segítségével képesek megmérni a minket körülvevő világ megannyi tulajdonságát, egyre növekvő számítási kapacitásuk segítségével pedig összetett módon képesek feldolgozni a kinyert információkat.

Szakdolgozatomban egy olyan alkalmazás fejlesztését tűztem ki célul, mely az okostelefonok eme képességét kezdő és hobbizenészek segítésére fordítja. A program digitális jelfeldolgozási módszereket felhasználva képes valós időben felismerni a nyugati zene két leggyakoribb akkordtípusát, a dúr és a moll hármashangzatokat. Az alkalmazás az egyik legelterjedtebb mobil operációs rendszerre, az Androidra készült, az akkordfelismeréshez pedig a készülék beépített mikrofonját használja.

Megvalósítás előtt utánajártam, hogy milyen módszereket alkalmaznak az automatikus akkordfelismerés témakörében, majd az alkalmasnak tűnő módszereket MATLAB segítségével tettem próbára. A tesztelt technikákból egy olyan eljárás összeállítására törekedtem, amely nagyobb késleltetés nélkül, megbízhatóan felismeri a hallott akkordot. Az eljárás az egyszerre egy hangszeren játszott akkordok felismerését teszi lehetővé, tesztelését klasszikus gitárral végeztem.

A választott eljárás az akkordot tartalmazó hangminta Pitch Class Profile (PCP) vektorának elemzésén alapul. Ezen vektor meghatározásához a hangmintát frekvenciatartományon ábrázolom, majd spektrumának segítségével meghatározom a kromatikus skála hangjainak intenzitását, azok frekvenciáinak ismeretében. A különböző oktávokon mért, azonos zenei hanghoz tartozó intenzitásokat ezután összesítem, ezzel egy 12 elemű vektort nyerve, melyet PCP vektornak neveznek. Ezután mintaillesztés segítségével meghatározom, hogy a PCP vektor melyik akkord ideális vektorára hasonlít leginkább.

Az eljárás Androidos megvalósítása előtt megvizsgáltam rendszer képességeit, és az általa nyújtott lehetőségeket egy ilyen jellegű program létrehozásához. Az implementáció során törekedtem az Android rendszerek különböző verzióinak széles körében kompatibilis szoftveres eszközök használatára, hogy az alkalmazás régebbi készülékeken is működjön.

Az elkészült alkalmazást felismerési gyorsaság és pontosság szempontjából összevetettem a MATLAB-ban összeállított programmal, illetve néhány, a Google Play-en megtalálható, ingyenes applikációval. Megállapítottam, hogy a saját alkalmazásom kevesebb funkciót tartalmaz ezeknél, ugyanakkor mind felismerési gyorsaság, mind pontosság szempontjából jobban teljesített a kipróbált programoknál.

Abstract

The widespread accessibility to smartphones has provided the users with possibilities that were unimaginable before. Through their cameras, microphones and various sensors, devices today are able to measure numerous features and qualities of the world surrounding us, and with their ever-improving computational power they can turn the extracted data into valuable information.

The goal of this thesis is to develop an application that uses this capability of smartphones to aid music enthusiasts and beginners at playing an instrument. The program uses digital signal processing (DSP) methods to recognize two of the most common chord types in western music, the major and minor triads, in real time. The application is made for Android, one of the most popular smartphone platforms; and uses the built-in microphone of the device to recognize the chord being played.

Prior to the implementation I have researched the DSP methods used in the field of chord recognition. I have selected those that seemed the most promising in terms of real-time application, then tested them using MATLAB. Using these techniques, I aspired to compose a method that is able to consistently recognize the aforementioned chord types, without significant delay.

The method I chose is based on the analysis of the Pitch Class Profile (PCP) of the sound sample that contains the chord. The PCP is obtained by representing the sample in frequency domain using a Fast Fourier Transform, then measuring the intensity of each note in the chromatic scale throughout several octaves of range. The intensity values that belong to the same note are then added up, thus creating a 12-ary vector, which is called a PCP vector or a chroma vector. Then pattern matching techniques are used to determine which chord's ideal representation resembles the PCP vector the most.

I have also researched the capabilities of the Android operating system and the means it provides to create such an application. Throughout the implementation I strived to make the program compatible with a wide range of Android versions, so that it can be used on older devices as well.

After implementation, I have compared the Android application to the method implemented in MATLAB with regard to recognition time and accuracy. I have also made a comparison with some of the free applications available on Google Play. I have come to the conclusion that my application has fewer features than those I have tried, but it offers lower latency and greater accuracy.

Bevezető

> A zene híd. (...) Összeköti az embereket valami feljebbvalóval, a zene kimondja azt, ami kimondhatatlan.
>
> *Emily Murdoch*

A zene szerves részét képezi mindnyájunk életének, még akkor is, ha nem műveljük, vagy keressük tudatosan. Olykor csak szórakoztat bennünket, máskor elgondolkodtat, vagy épp emlékeket ébreszt bennünk. Megannyi, számunkra kedves tulajdonsága hatására sokakban felébred a vágy, hogy kedvenc számaikat el tudják játszani, kísérni valamilyen hangszeren.

Szakdolgozatom keretei közt egy olyan akkordfelismerő alkalmazás fejlesztésére törekedtem, mely a zene iránt érdeklődők, elsősorban kezdő és hobbizenészek számára nyújt hasznos információt.

Az első fejezetben a dolgozat megértéséhez szükséges zenei alapfogalmakat ismertetem, kitérve a zenei hangok fizikai tulajdonságaira, ezen tulajdonságok és az emberi hangérzékelés viszonyára, és természetesen az akkord fogalmára.

A második fejezetben az automatikus zenelejegyzés, és vele az akkordfelismerés múltjával foglalkozom. Leírom a technológia fejlődésével párhuzamos, fokozatos fejlődésüket, említést téve néhány fontosabb mérföldkőről. Emellett röviden ismertetek néhány, már a piacon lévő programot, melyek ezen kutatási területek módszereit, vívmányait alkalmazzák.

A harmadik fejezet részletezi az akkordfelismerés technológiai lehetőségeit, a különböző digitális jelfeldogozási eljárásokat és azok matematikai hátterét, kitérve előnyeikre és esetleges hátrányaikra. Az itt ismertetett algoritmusok az akkordok Pitch Class Profile (PCP) tulajdonságai alapján történő felismerését veszik alapul, hiszen ez az eljárás meglehetősen általános, és egymástól nagyrészt független lépésekből épül fel.

Ahhoz, hogy a különböző módszerek közül a megfelelőket ki tudjam választani, teszteltem őket MATLAB segítségével. A tesztelés módját, tapasztalatait, tanulságait a negyedik fejezetben részletezem, egyúttal megadom, hogy a módszerek közül melyeket használtam fel az Android alkalmazásban.

Az ötödik fejezet az Android platform bemutatásával, rövid történetével és technikai jellemzőivel foglalkozik, valamint bemutatja az Android-alkalmazások főbb összetevőit, a feladat megértéséhez szükséges részletességgel.

Az elkészült alkalmazás felépítését és működését a hatodik fejezetben tárgyalom. Szintén itt térek ki a programozói döntésekre, melyek befolyásolták a fejlesztés irányát, illetve az

implementáció során felmerült nehézségekre és hibalehetőségekre, melyekbe belefutottam.

Az Androidos alkalmazást a hetedik fejezetben leírtak alapján összevetettem a MATLAB-os megvalósítással, valamint néhány ingyenes, Google Play-en elérhető alkalmazással.

Az nyolcadik, és egyben utolsó fejezetben összefoglalom a dolgozat számomra fontos tanulságait, valamint leírom, hogy a továbbiakban milyen irányba mehet tovább az alkalmazás fejlesztése annak érdekében, hogy szélesebb körben is alkalmazható legyen.

1. fejezet

Zenei alapok

> A hang a zenének ősanyaga;
> ugyanaz a muzsikus számára,
> mint a festő számára a festék és
> a vászon.
>
> *Dr. Kesztler Lőrinc*

Ez a fejezet elméleti bevezetőként szolgál, amely ismerteti a legfontosabb, megértéshez szükséges zenei fogalmakat.

1.1. A hang, mint fizikai jelenség

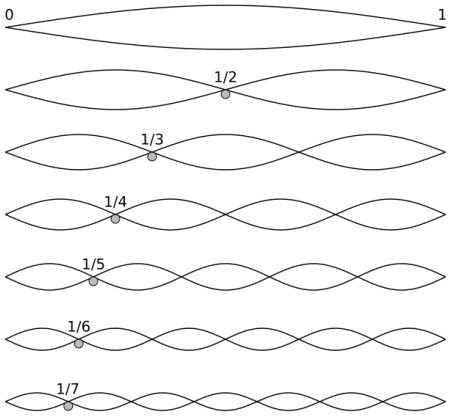
A hang fizikai értelemben nem más, mint egy mechanikai hullám, melyet a levegő, mint közvetítő közeg juttat a fülünkbe. Forrása bármilyen rezgésre képes test lehet, mely a rezgésével a vele érintkező levegőrészekben hullámokat kelt.

Zenei hangról akkor beszélünk, ha a hanghullám periodikus, azaz egyenlő időközönként egyenlően ismétlődik [13]. A szabálytalan, nem periodikus hangokat zörejnek nevezzük. Jelen dolgozatban csak az előbbiekkel foglalkozom.

1.2. Zenei hangok

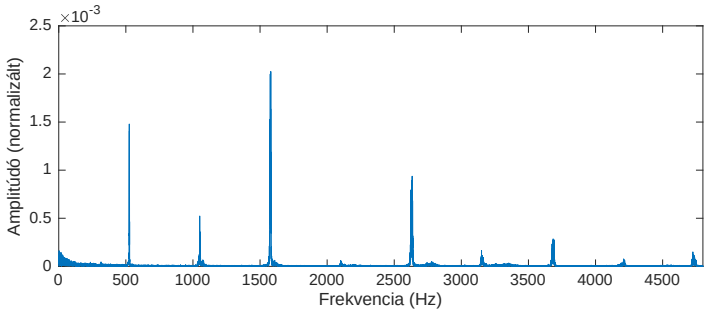
A zenei hang lényegében a levegő periodikus hullámzása, így rendelkezik a hullámok két, számunkra legfontosabb tulajdonságával: amplitúdóval és frekvenciával. Az amplitúdó a hang intenzitását, azaz a hangerőt határozza meg, míg a frekvencia a hangmagasságot.

Egy zenei jel jellemzően nem egyetlen szinuszos jelből áll. Ugyanaz a hang különböző hangszereken megszólaltatva egész más hangérzetet kelt bennünk. Ennek oka, hogy a hangszer hangot keltő része nem csak az alapfrekvenciáján végez rezgést, hanem annak egész számú többszörösein is, melyeket felharmonikusoknak, vagy *felhang*oknak nevezünk. Az alaphangot és a felhangokat együtt *részhang*oknak nevezzük.

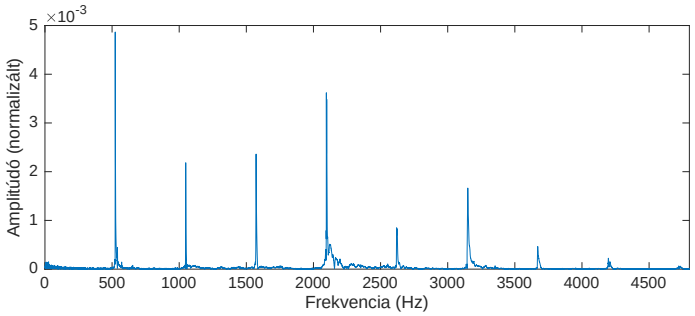


1.1. ábra. *Az alaphang és az első hat felhang rezgése egy húron, a hozzájuk tartozó hullámhosszokkal [28].*

Egy zenei hang így az alaphang és a felhangok összegeként áll elő. A hallható felharmonikusok száma, illetve azok egymáshoz és az alaphanghoz viszonyított intenzitása adja a hangszínt. Ennek köszönhetően halljuk más jellegűnek különböző hangszerek hangját, illetve az emberek hangját is hangszínük alapján tudjuk megkülönböztetni.



(a) furulya



(b) elektromos gitár

1.2. ábra. *A 523.25 Hz-es c" hang spektruma fafurulyán és elektromos gitáron lejátszva (mikrofonos felvétel alapján)*

Az 1.2. ábra szemlélteti ugyanazon hang spektrumát elektromos gitáron és fafurulyán. A gitár hangja láthatóan gazdagabb felharmonikusokban. Bár a hangerőt tipikusan decibelben szokás ábrázolni, mivel az emberi érzékeléshez közelebb áll, jelen dolgozatban a

számításokhoz nem decibelt használok, ezért ábrázolás terén is maradok az intenzitásnál.

A zenei hangok másik fontos tulajdonsága a hangmagasság, mely elsősorban az alaprezgés frekvenciájától függ: nagyobb frekvencia esetén magasabb hangot hallunk. Ha azonban az alaphang az emberi hallástartományon (20 Hz - 20000 Hz) kívül esik, vagy a felharmonikusok intenzitása elnyomja azt, az az általunk érzékelt hangmagasságot is megmásíthatja. Elmondhatjuk tehát, hogy a hangszínhez hasonlóan ez a jellemző is szubjektív, inkább érzeti tulajdonsága a hangnak.

1.3. Hangrendszerek és hangközök

A legtöbb zene elsősorban meghatározott számú kitüntetett szerepű zenei hangból építkezik, melyek hangrendszert alkotnak. A nyugati kultúrkörben, és a világ számos más táján uralkodóvá vált hangrendszer hét ilyen hangot jelöl ki, melyeknek külön elnevezése van. Ezeket törzshangoknak nevezzük, elnevezésük területenként eltérő [25].
Az angolszász országokban:

C - D - E - F - G - A - B (- C - ...)

Más germán, nyugati és déli szláv nyelvű országokban, illetve Magyarországon is:

C - D - E - F - G - A - H (- C - ...)

A könnyűzenében általánosabb az angolszász elnevezések használata, így a továbbiakban én is ezeket használom.

Az azonos nevű hangok között kétszeres frekvenciakülönbség van, az A hangok frekvenciáját pedig 1955-ben kanonizálták (440 Hz, 880 Hz, stb.), a többi hang frekvenciája hangolásonként[1] kisebb mértékben eltér.

A hét törzshangból további öt hangot származtathatunk, melyek frekvenciában a törzshangok között helyezkednek el[2]:

C - **C♯** - D - **E♭** - E - F - **F♯** - G - **G♯** - A - **B♭** - B (- C - ...)

Az így kapott 12 hang alkotja az úgynevezett kromatikus hangsor fokait, melyben minden egymással szomszédos fok frekvenciaaránya azonos, a köztük lévő távolságot félhangnak is nevezik. Ebből látható, hogy az egymást követő fokok frekvenciája nem lineárisan, hanem exponenciálisan változik. A hangsor két fokának frekvenciaaránya határozza meg a két fok hangközét.

A hangköz elnevezése attól függ, hogy hány félhang különbséget ölel fel. A jelentősebb hangközöket a 1.1. táblázat foglalja magába.

[1] A dolgozat a könnyűzenében leggyakoribb hangolást, a *temperált* (kiegyenlített) hangolást veszi alapul, így a hangközökre vonatkozó megállapítások is erre a hangolási módra vonatkoznak. A temperáltan hangolt hangszeren bármilyen hangnemben lehet játszani anélkül, hogy át kellene hangolni.

[2] Ezeknek a hangoknak a nevei kontextustól függően a félhanggal mélyebb és magasabb hang nevéből is származtathatók, így például az F♯ (F-sharp, magyarul Fisz) és a G♭ (G-flat, magyarul Gesz) ugyanazt a hangot jelöli. Az egységesség kedvéért a továbbikban a „♯" alakot használom.

1.1. táblázat. *A kromatikus skála hangközei*

Félhangok száma	Név	Példa
0	tiszta prím	C-C
1	kis szekund	C-C♯
2	nagy szekund	C-D
3	kis terc	C-E♭
4	nagy terc	C-E
5	tiszta kvart	C-F
6	tritónusz	C-F♯
7	tiszta kvint	C-G
8	kis szext	C-G♯
9	nagy szext	C-A
10	kis szeptim	C-B♭
11	nagy szeptim	C-B
12	tiszta oktáv	C-c

A hangközök egy része fizikai értelemben is különleges kapcsolatot jelent két hang között. A tiszta oktáv 2:1 frekvenciaarányt ír le, ami azt is jelenti, hogy bármely hang tiszta oktávja az első felharmonikusa is egyben. Sőt, egy hangtól az n-edik tiszta oktávra lévő hang az eredeti hang $2^n - 1$-edik felharmonikusa. Ez az erős fizikai kapcsolat az oka annak, hogy az egymástól oktávnyira lévő hangokat - más hangmagasságuk ellenére - ugyanannak a zenei hangfoknak érezzük.

A felhangsorban továbbá megjelennek más hangközök is, például a tiszta kvint (2. és 5. felhang), vagy a nagy terc (4. felhang), melyek az emberi fül számára igencsak harmonikusan csengenek.

1.4. Akkordok

A hangsor több fokának együttes megszólalását hívjuk hangzásnak, vagy akkordnak. Az akkordok fontos szerepet töltenek be a nyugati zenében, elsősorban a kíséretet játszó hangszerek esetén, mint például a gitár, vagy a zongora.

Minden akkord tartalmaz egy úgynevezett alaphangot[3], mely általában a legmélyebb hang az akkordban. A többi hang alaphanggal vett hangköze határozza meg az akkord típusát. Már két fok megszólalása is alkothat akkordot, ilyen például az *üres akkord*, mely az alaphang mellett csupán a tiszta kvintet tartalmazza. Gyakrabban azonban 3, 4, esetenként még több hang összecsengésével jönnek létre, ezeket rendre hármashangzatoknak, négyeshangzatoknak, stb. nevezzük.

A hármashangzatok legfontosabb típusai a következők:

- **dúr:** alaphang, nagy terc, tiszta kvint
- **moll:** alaphang, kis terc, tiszta kvint
- **szűkített:** alaphang, kis terc, tritónusz
- **bővített:** alaphang, nagy terc, kis szext

[3]Nem összekeverendő a felhangoknál tárgyalt alaphanggal

A dúr vidám, derült, erőteljes hangzású; míg a moll inkább szomorkás, lágy, drámai. Ez a két akkordtípus kétségtelenül a leggyakoribb a nyugati zenében, így ez a dolgozat ennek a két típusnak a felismerésére szorítkozik.

Egy akkordot alaphangja és típusa alapján tudunk azonosítani, nevük is ebből a két tulajdonságukból áll össze (pl C-dúr, A-moll). Kottában a dúr akkordokat egyszerűen az alaphang nagybetűs nevével szokás jelölni (pl. C), a moll akkordokat pedig az alaphang nevével és egy kis „m" betűvel, vagy az alaphang kisbetűs nevével (pl. Am, vagy a).

1.5. A félhangok felosztása

A hangközök leírására korábban a két hang frekvenciájának hányadosát használtam, ez azonban nem feltétlenül a legkényelmesebb megoldás a gyakorlatban. A zenei hangok frekvenciájának exponenciális skálázódása miatt ugyanaz a hangköz más és más frekvenciakülönbséget jelent attól függően, hogy az azt közrefogó hangok hol helyezkednek el a skálán. A frekvenciaaránnyal való leírás bonyolultabbá teszi a hangok közti távolságok számolását, ráadásul az emberi érzékeléssel sincs összhangban.

Ez igényt teremt egy új mértékegység bevezetésére, mellyel a hangközök egyszerű számtani különbségként írhatók le. Ez a mértékegység a *cent* [26], mely az exponenciális skála logaritmikus linearizálásán alapul. Egy tiszta oktávot 1200 fokú lineáris skálává alakít, melyben a kromatikus skála fokai egységesen 100 centre helyezkednek el egymástól. Így például egy tiszta kvint 700 cent, egy nagy terc pedig 400 cent különbséget jelent a skálán mindenhol.

Két hang frekvenciaarányát ($Q = \frac{f_2}{f_1}$) az alábbi képlettel lehet centben (C) kifejezni:

$$C = 1200 \cdot \log_2 Q \tag{1.1}$$

A centérték frekvenciaaránnyá alakításához pedig az alábbi képlet alkalmazható:

$$Q = (\sqrt[1200]{2})^C \tag{1.2}$$

A centtel tehát arányokat lehet leírni, így ahhoz, hogy hangmagasság mérésére lehessen használni, ki kell jelölni egy frekvenciértéket referenciának. Ebben a dolgozatban ez az állandó érték a C_2 hang, azaz a 65.41 Hz-es frekvencia lesz.

2. fejezet

Automatikus zenelejegyzés és akkordfelismerés

> Tudom, mennyi zene marad lejegyzetlenül, ami csak az improvizáció pillanataiban létezik.
>
> *Ljudmila Ulickaja*

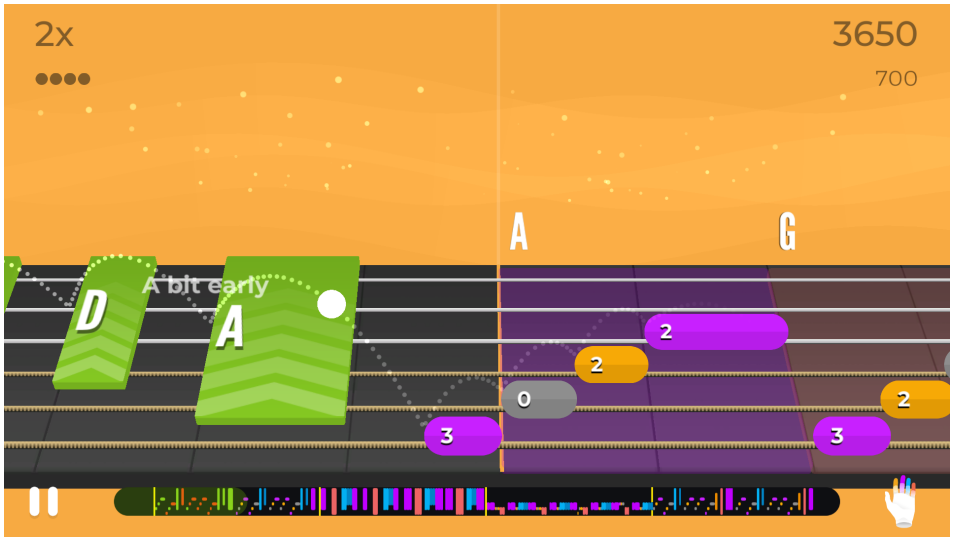
A dallamok lejegyzése, a dalok, zenék megörökítése már egészen korán, az ókorban is foglalkoztatta az embereket. Azonban egészen a XX. századig erre csak hallás alapján volt lehetőség, melyet általában képzett zenészek végeztek, hiszen jó zenei hallásra, és a kotta (vagy más zenei jelrendszer) ismeretére volt hozzá szükség. A XX. század során a számítógépek megjelenése azonban teret adott egy új kutatási területnek, az *automatikus zenelejegyzésnek*. Az „automatic music transcription" kifejezést feltehetőleg először 1977-ben használták [20], melynek lényege a zene emberi olvasásra alkalmas formában történő lejegyzése hangfelvétel alapján. Ebbe a témakörbe tartozik például a hangnem-, a tempó- és az *akkordfelismerés* is.

Előbb a személyi számítógép, majd később az okostelefonok, tabletek elterjedése lehetővé tette, hogy az automatikus zenelejegyzés megoldásait használó szoftverek széles körben elterjedjenek, hiszen ezek az eszközök már kielégítő számítási kapacitással és beépített mikrofonnal is rendelkeznek. Az alkalmazások közt találhatók hangolók, játékok, oktató szoftverek és ezek ötvözetei is.

Ebben a fejezetben a már elérhető programok közül ismertetek néhányat, a terület technológiai hátterét és lehetőségeit pedig a 3. fejezetben részletezem.

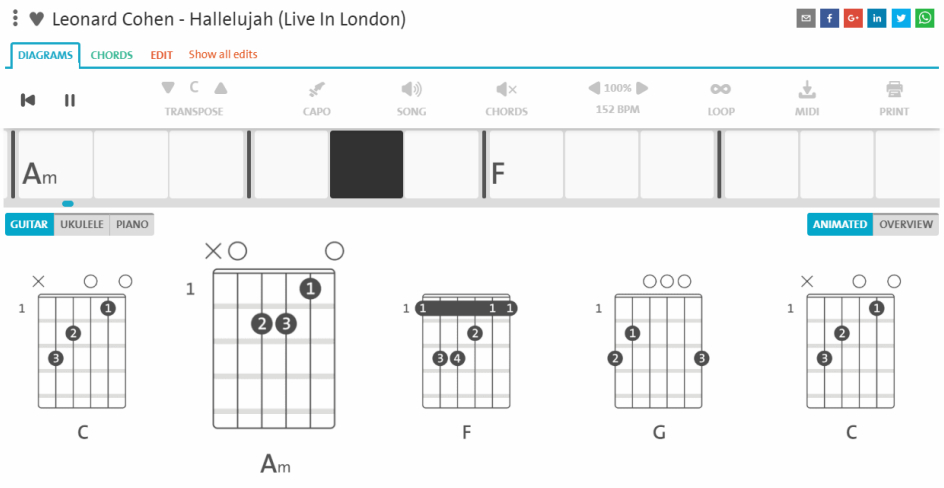
A hangszeres oktatóprogramok közé tartozik a hang- és akkordfelismerésen alapuló Yousician, mely Windows, iOS, macOS és Android platformokon érhető el, és jelenleg gitárt, zongorát, ukulelét, basszusgitárt és éneket támogat. Különlegessége, hogy zenei alapra lehet neki játszani a kiválasztott hangszeren, kotta vagy gitártab alapján, a program pedig mikrofonon keresztül figyeli a felhasználó játékát, és azonnali visszajelzést ad a lejátszott hangok

helyességéről és időbeli pontosságáról. Nagyon hasonló célt szolgál a népszerű Rocksmith videojáték sorozat, mely elektromos gitárt és basszusgitárt támogat.



2.1. ábra. *Yousician az egyik lecke játszása közben*

Az akkordfelismerés témájához még közelebb álló példa a Chordify nevű program, mely elérhető okostelefonokra és webes alkalmazásként is. Képes hangfájlokat és videoklipek hangsávját elemezni, majd teljes akkordmenetet felírni a zenéhez, figyelembe véve azok tempóját is. A lejátszójával elindítva a zenét folyamatosan mutatja, hogy milyen akkordot kell épp lefognunk, és hogy mikor és milyen akkord fog következni.



2.2. ábra. *Chordify, Leonard Cohen „Hallelujah" című számának akkordmenetével*

Kifejezetten valósidejű akkordfelismerésre is találhatók példák, ilyen például a 2014-es születésű Uberchord, mely jelenleg csak iOS-re érhető el, és akkordok széles skálájának felismerését ígéri, szintén mikrofonon keresztül.

A kiforrott programok mellett számos kisebb akkordfelismerő alkalmazás is található a Google Play-en és az Apple App Store-ban egyaránt.

Jelen dolgozat célja egy olyan Android alkalmazás kifejlesztése, mely képes minden dúr és moll akkordot valós időben felismerni az eszköz mikrofonján keresztül. A felismerésért felelős algoritmust a klasszikus gitár hangjához igazítom, más hangszerek és teljes zeneszámok akkordjainak felismerése egy hosszabb távú cél részét képezik.

3. fejezet

Akkordfelismerési módszerek

Az akkordfelismerés a felvett hangrészletben felcsendülő zenei hangoknak, azok frekvenciániak felismerésével lehetséges. Ezért a feladat elsősorban a digitális jelfeldolgozás (*digital signal processing*, DSP) tudományterület módszereire támaszkodik.

Amikor hangfelvételt készítünk, a mikorfon először elektromos jellé alakítja a hanghullámot, a hullámot jelentő légnyomás-váltakozást feszültség-váltakozásként ábrázolva. Digitalizálás során a felvevőeszköz ennek az elektromos jelnek méri meg a feszültségét adott időközönként [1], *mintavételezi* azt. A mintavételek gyakorisága határozza meg a *mintavételi frekvenciát*, mely az egyik legfontosabb leíró tulajdonsága a digitális jeleknek.

A felvett hang így feszültségértékek sokaságaként tárolódik az adathordozón, melyeknek időköze azonos, az időköz ismeretében pedig értelmezhető, lejátszható a hang. Ekkor a hangfelvétel *időtartományon* van ábrázolva.

Bár léteznek időtartománybeli módszerek, melyek egy zenei hang felvételéről meg tudják határozni a hangmagasságot [9], ezek az eljárások nem alkalmasak összetettebb hangok, több zenei hangot tartalmazó minták elemzésére, ami akkordfelismerésnél kizáró ok.

Ahhoz, hogy az akkordot alkotó hangok frekvenciáit el tudjuk különíteni egymástól, frekvenciatartományon kell ábrázolnunk a jelet. Ezt a jel diszkrét Fourier-transzformációjával (DFT) kaphatjuk meg, mely képes felbontani a periodikus jelet különböző frekvenciájú és amplitúdójú szinusz és koszinusz jelek összegére. A DFT elvégzéséhez használható valamely gyors Fourier-transzformáció (fast Fourier transform, FFT) algoritmus, mely kedvezőbb futási idővel rendelkezik.

A transzformáció eredményeként megkapjuk a jelrészlet *spektrumát*, mely a frekvencia szerint ábrázolja a szinuszos és koszinuszos összetevők amplitúdóját. Ennek a vizsgálatával foglalkozik a fejezet.

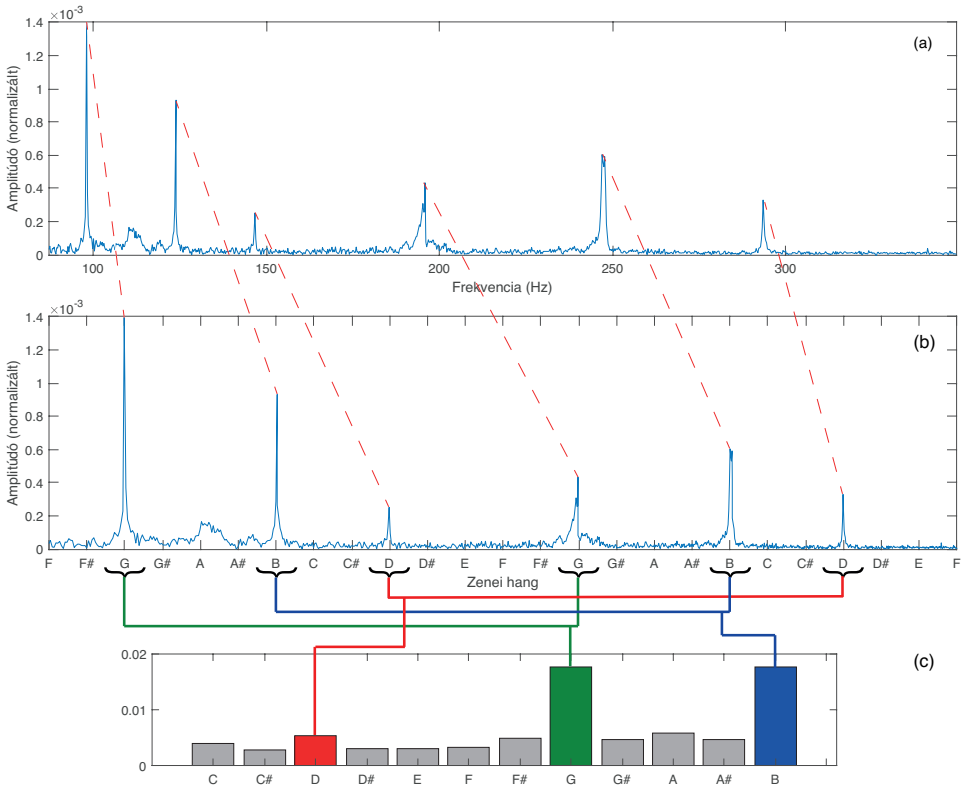
3.1. Pitch Class Profile (PCP)

Az eljárást, melynek meghatározó szerepe lett az akkordfelismerés kutatásában, Fujishima [10] írta fel 1999-ben. Alapelve a következő: Először DFT segítségével meghatározza

a hangminta spektrumát, majd a spektrumban meghatározza minden zenei hang intenzitását, azok frekvenciái ismeretében. Ezután a spektrum kiválasztott, több oktávot felölelő tartományban összeadja az azonos zenei hanghoz tartozó intenzitásértékeket, ezzel egy 12 elemű vektort kapva, melyet *PCP vektor*nak, *chromagram*nak, vagy *chroma vektor*nak neveznek.

A chroma vektort ezután összehasonlítja minden egyes akkord „ideális vektorával", vagyis azzal a 12 elemű vektorral, melyben az akkordhangok helyén 1, az akkordban nem szereplő hangok helyén pedig 0 áll. Az észlelt akkord az lesz, amelyikkel a chroma vektor a legjobb egyezést mutatja.

Fujishima a spektrum minden mintáját hozzáadja a chroma vektor egy eleméhez. Ahhoz, hogy el lehessen dönteni, hogy a minta melyik hanghoz áll a legközelebb, minták frekvenciaértékeit először a hangmagasság szerint lineáris formában kell kifejezni, amit a (1.1) képlethez hasonló módon, egy referenciaérték kijelölésével és logaritmus alkalmazásával ér el, az oktávnyi hangterjedelmeket a kromatikus skála fokainak megfelelően 12 egyenlő részre osztva. Ezt a folyamatot a 3.1. ábra illusztrálja.



3.1. ábra. *G-dúr triád spektrumának azonos részlete (a) frekvencia és (b) hangmagasság szerint ábrázolva, illetve a spektrumból létrejövő chroma vektor (c) (csak az akkordhangok vannak kiemelve)*

Az eljárás formálisabban megfogalmazva: Az $x(n)$ bemenő jelnek DFT segítségével meghatározza a spektrumát ($X(k)$), majd a spektrum minden amplitúdóértékéhez (mintájához) a hangmagassága szerint hozzárendel egy egész számot (p) a $\{0..11\}$ intervallumról, ezzel megfeleltetve őket a kromatikus skála 12 foka közül valamelyiknek (C = 0; B = 11). Ezután a hozzárendelt számok alapján csoportosítva összesíti a spektrum értékeit, ezzel

megalkotva a chroma vektort. Más szóval, a chroma vektor egy indexéhez a kromatikus skála egyetlen foka tartozik, melynek különböző oktávokon mért intenzitása adja az indexhez tartozó értéket. A hozzárendeléseket az $M(l)$ függvénnyel kaphatjuk meg, mely az l-edik spektrum mintához rendelt chroma vektor indexet adja meg. Az $M(l)$-t a következő képlet definiálja:

$$M(l) = \begin{cases} -1, & \text{ha } l = 0 \\ \left[12 \log_2 \left(\frac{f_s\left(\frac{l}{N}\right)}{f_{ref}} \right)\right] \mod 12, & \text{ha } l = 1, 2, ..., N/2 - 1 \end{cases} \tag{3.1}$$

A chroma vektor a következő képlet alapján áll össze:

$$PCP(p) = \sum_{M(l)=p} ||X(l)||^2 \tag{3.2}$$

A képletekben N a jel Fourier-transzformáltjának mintaszáma, melyeken $l = 0, 1, ..., N/2 - 1$ segítségével iterálunk végig[1]; f_{ref} az alsó referencia-frekvenciaérték, mely a legmélyebb figyelembe vett hangot jelöli; az $f_s\left(\frac{l}{N}\right)$ kifejezés pedig az $X(l)$ spektrumminta frekvenciáját jelenti. A szögletes zárójel a legközelebbi egészre való kerekítés operátora. A (3.2) képlet alapján látható, hogy nem közvetlenül a spektrum mintáit, hanem azok (tagonkénti) abszolútérték-négyzeteit összegzi, melyet *energiaspektrum-sűrűségnek* nevezünk.

Az előállított chroma vektor a 12 félhang intenzitását tartalmazza. Az akkord meghatározásához a chroma vektor tartalmát valamilyen módon össze kell vetnünk az általunk ismert akkordok „képével", erre a célra mintaillesztést használunk. Az akkordtípusokat szintén 12 elemű vektorokkal, bitmaszkokkal ábrázoljuk, melyeket Fujishima Chord Type Template-nek (CTT) nevez. A $CTT_a(b)$ ($b \in \{0..11\}$) értéke akkor 1, ha az a típusú, C alaphangú akkord tartalmazza a b-vel jelölt hangot. Minden más esetben az értéke nulla. A dúr akkord CTT-je például (1,0,0,0,1,0,0,1,0,0,0,0).

Kihasználva azt, hogy az azonos típusú akkordok hangjai ugyanannyi félhang távolságra helyeztkednek el egymástól, elegendő egyetlen bitmaszkot eltárolni típusonként[2]. A CTT elemeinek d-vel jobbra léptetésével[3] ugyanis megkapjuk az akkordtípus bitmaszkját d félhanggal magasabban.

A chroma vektor és a bitmaszkok összevetésére Fujishima kétféle mintaillesztési eljárást hasonlít össze:

- *a legközelebbi szomszéd módszerét* (Nearest Neighbor Method), és
- *a súlyozott összeg módszerét* (Weighted Sum Method)

[1] valós bemenet esetén a DFT eredményvektorának első fele az egész spektrumot tartalmazza
[2] a Chord Type Template elnevezés önmagában is erre utal
[3] a léptetés során kicsordult érték a vektor elejére kerül

Legközelebbi szomszéd módszer: A kiszámított PCP vektor és az a akkord bitmaszkjának ($T_a(p)$) tagonkénti különbségének négyzeteit összegezzük. Az észlelt akkord az, amelyiknek a bitmaszkja a legkisebb pontszámot éri el. Képlettel:

$$P_{NN,a} = \sum_{p=0}^{11} (T_a(p) - PCP(p))^2 \tag{3.3}$$

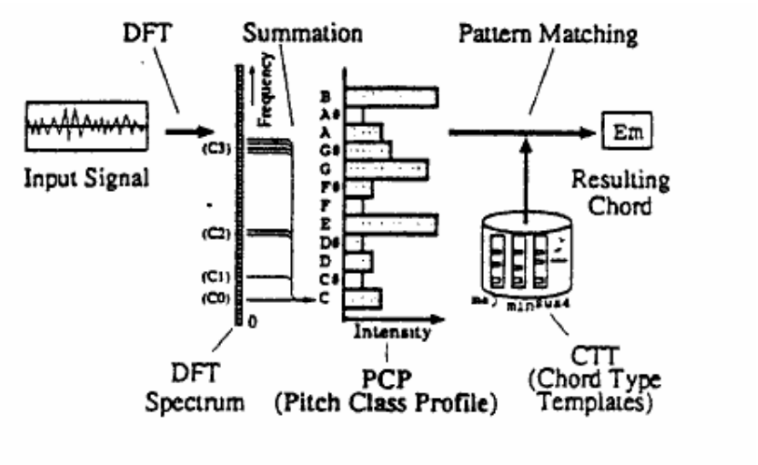
ahol $P_{NN,a}$ az a akkord pontszáma a módszer szerint.

Súlyozott összeg módszer: A PCP vektor és az a akkord súlyozott bitmaszkjának ($W_a(p)$) skaláris szorzatát vesszük. A súlyok meghatározása tapasztalat útján történik, melyben figyelembe lehet venni olyan heurisztikákat, mint az akkord előfordulási gyakoriságát, vagy az akkordot alkotó hangok számát. Az észlelt akkord az, amelyiknek a bitmaszkja a legnagyobb pontszámot éri el. Képlettel:

$$P_{WS,a} = \sum_{p=0}^{11} W_a(p) \cdot PCP(p) \tag{3.4}$$

ahol $P_{WS,a}$ az a akkord pontszáma a módszer szerint.

Az imént ismertetett PCP algoritmus vázlatát a 3.2. ábra foglalja össze.



3.2. ábra. *Fujishima PCP algoritmusának folyamatábrája [10]*

A PCP nagy népszerűségnek örvend az akkord- és hangnemfelismerés témakörében. Ennek oka, hogy viszonylag egyszerű, számításigénye nem nagy, és könnyen felbontható egymástól többnyire független részekre. Ezek a részek pedig továbbfejleszthetők, finomhangolhatók az épp aktuális feladat igényeihez igazítva. Ennek lehetőségeit ismertetem a következő pontban.

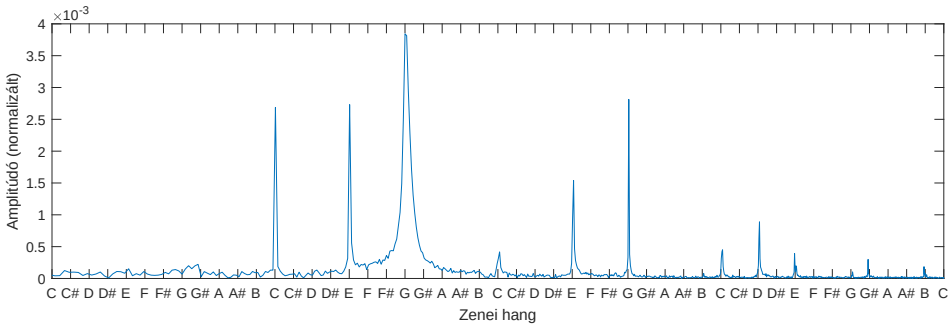
3.2. A chroma vektor meghatározása

A PCP eljárás első fázisa a chroma vektor előállítása a spektrum alapján. Olyan chroma vektor előállítására érdemes törekedni, amely az akkord hangjait a lehető legjobban kiemeli, ezzel megkönnyítve az akkord felismerését. Ebben a pontban ennek a lehetőségeit ismertetem.

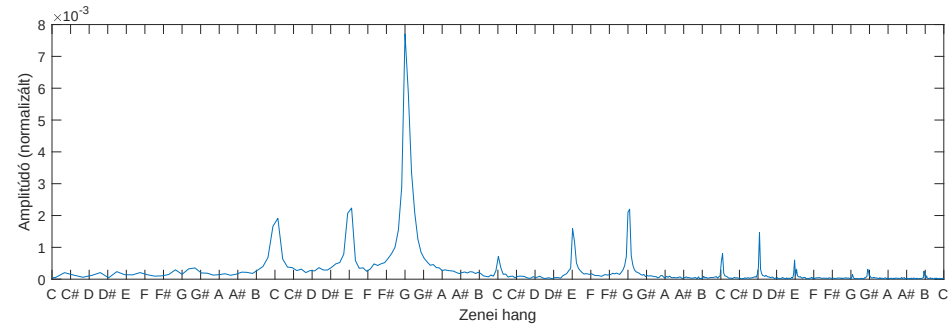
3.2.1. Ablakméret megválasztása

A felismerés pontossága nagyban függ az ablakmérettől, azaz hogy milyen hosszú, hány mintából álló hangfelvételt elemzünk. Az FFT algoritmus egy N mintából álló ablakra N adatpontból álló amplitúdóspektrumot ad vissza, melyben az adatpontok frekvencia szerint lineárisan helyezkednek el, vagyis két tetszőleges szomszédos adatpont frekvenciakülönbsége ugyanakkora.

Ahhoz, hogy a zenei hangoknak megfelelően centben ábrázoljuk a hangok magasságát, logaritmizálnunk kell a frekvenciatengelyt. Ennek az az eredménye, hogy a mélyebb hangoknak megfelelő tartományokba kevesebb adatpont kerül, a magasabb hangokéba pedig több. Ez önmagában nem lenne gond, viszont így előfordulhat megfelelően kicsi ablakméretnél, hogy mélyebb hangok esetén a spektrális csúcsok nem jelennek meg az alacsony adatfelbontás miatt. Sőt, ez a probléma a magasabb hangoknál is előfordulhat, elsősorban ott, ahol az amplitúdó csak egy szűk frekvenciatartományon kiugró.
A jelenséget a 3.3. ábra szemlélteti:



(a) *1 mp*



(b) *0,5 mp*

3.3. ábra. *C-dúr triád spektruma ugyanabból a felvételből kivágott 1 mp hosszú és 0,5 mp mintával*

Az ábrán jól látható, hogy az egyes csúcsok nem azonos mértékben vesztenek a mért intenzitásukból az ablakméret csökkentésével: a legnagyobb intenzitású G hanghoz képest a mélyebb C és E hangok, és az egy oktávval magasabban lévő E és G hangok csúcsai is jóval kisebbek lettek.

Az ablakméret növelésével viszont nő a kockázat, hogy az ablakba akkordváltás kerül (akár több is), ez pedig használhatatlanná is teheti az egész ablakot[4]. Emellett nő a rendszer késleltetése is.

Az adatfelbontás egyenetlenségére megoldást jelenthet a Lee [12] által is használt Constant-Q transzformáció [2], mely a DFT egy módosított változata. Az általa számított spektrumban a minták száma a hangmagasság szerint lineáris, nem pedig a frekvencia szerint, mint a sima FFT esetén, ezért zenei jelek spektrumának elemzéséhez kitűnő. Ennek azonban komplexitása és számításigénye jóval nagyobb az FFT-hez képest, ami miatt valósidejű használatra kevésbé alkalmas. Bár léteznek megoldások, melyekkel hatákonyabbá lehet tenni a számítását, ezek jelentősen bonyolultabbá tennék az implementációt, a számítási idő pedig továbbra is kérdéses lenne. Ezeknek a módszereknek a tárgyalása ezért nem képezi jelen dolgozat részét.

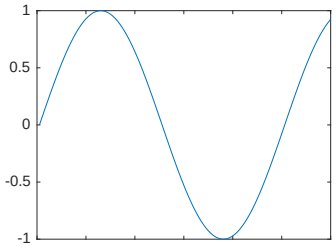
Az akkordváltások miatti ablakvesztésekre megoldást jelenthet, ha megengedjuk az ablakok átlapolódását, úgynevezett csúszóablakot alkalmzunk. Az átlapolódás lehetővé teszi, hogy jobban megközelítsük az akkordváltás pillanatát, vagyis legyen legalább egy olyan ablakunk is, melynek eleje nagyjából az akkordváltás pillanatában van. A módszer hátránya, hogy használatával a teljes akkordfelismerő algoritmust gyakrabban kell lefuttatni, mint az átlapolódás nélküli esetben: 50%-os átlapolódásnál kétszer, 75%-osnál már négyszer olyan gyakran. Ez arányosan nagyobb számításigénnyel jár. Emellett nem jelent megoldást az olyan esetekre, ahol az ablakméretnél rövidebb időközönként történnek akkordváltások. Ilyen esetben csak az segíthetne, ha tudnánk detektálni az akkordváltásokat, ez azonban jelentős mértékben növelné az algoritmus komplexitását. Ezzel az *onset detection* kutatási területe foglalkozik, melynek tárgyalása a dolgozat keretein túlmutat.

3.2.2. Ablakozás [19]

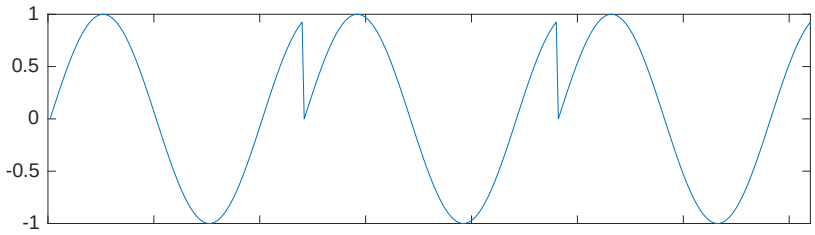
Mielőtt az ablakba foglalt hangmintát alávetnénk az FFT-nek, figyelembe kell vennünk a spektrumszivárgás (más néven spektrális szóródás) jelenségét. A diszkrét Fourier-transzformáció feltételezi, hogy a transzformálandó hangminta a zenei jel egész számú periódusa, azonban ez a valóságban a legtöbb esetben nincs így. A jelsorozat végessége csonkított hullámformát eredményez, ahol a periódusok határán „szakadások", éles átmenetek fordulhatnak elő.

A nem egész számú periódusból álló mintát, és a belőle képzett periodikus függvényt a 3.4. ábra illusztrálja.

[4]Ez persze kis ablakméretnél is előfordul, de olyankor az adatveszteség is kisebb.



(a) mintavételezett szakasz, a szinusz függvény 1,2-szeres periódusa



(b) A DFT által észlelt, időtartománybeli jel

3.4. ábra. *Szakadások időtartományban*

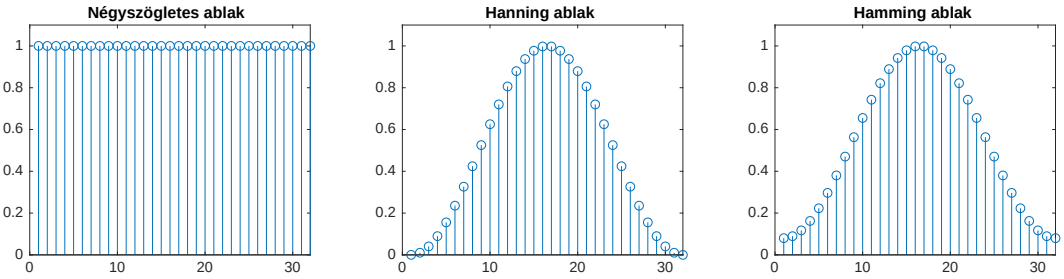
Ezek a szakadások oly módon torzítják a Fourier-transzformációval kiszámított spektrumot, mintha egy frekvenciakomponens energiája szétterjedne a szomszédos frekvencia vonalakon vagy sávokon, így a csúcsok kevésbé lesznek kiemelkedőek. Ezt a jelenséget nevezik spektrumszivárgásnak.

Ennek csillapítására alkalmazzák az *ablakozás* módszerét, mely a szakadások amplitúdóját hivatott csökkenteni a periódusok határán. Az ablakozás művelete az eredeti ablak értékeinek tagonkénti szorzását jelenti egy úgynevezett ablakfüggvény értékeivel, még a spektrum számítása előtt, időtartományban[5]. A jelfeldolgozás területe többféle ablakfüggvényt megkülönböztet, ezek közül a téma szempontjából fontosabbak:

- *Négyszög ablak*
- *Hanning ablak*
- *Hamming ablak*

Ezek közül a négyszög ablak jelenti az ablakozatlan formát, azaz ha semmilyen explicit simító ablakot nem alkalmazunk. A felsorolt ablaktípusokat a 3.5. ábra szemlélteti. Ugyanazon hangminta különböző ablaktípusokkal vett spektrumára a 4.1 pontban látható példa.

[5]Ezzel egyenértékű a konvolúció művelete frekvenciatartományban.



3.5. ábra. *32 minta szélességű négyszög, Hanning és Hamming ablak*

Egyes irodalmi források [17, 15, 16] Hamming ablakot használnak a PCP algoritmusban, míg a BME MOGI [19] az ehhez hasonló Hanning ablakot javasolja általános célra és szinuszos jelek kombinációjából álló jelekre. A következő fejezetben megvizsgálom, hogy ezekkel az ablaktípusokkal érhető-e el javulás az implicit négyszög ablakhoz képest.

3.2.3. Spektrális csúcsok kiválasztása

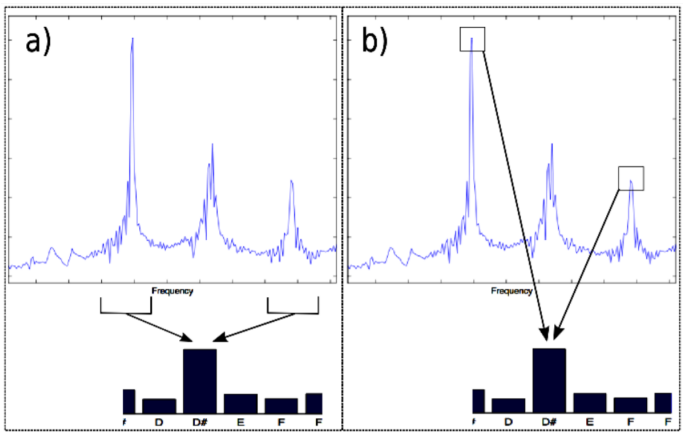
Miután megkaptuk a hangminta spektrumát, ki kell nyernünk belőle azokat a frekvenciaértékeket, melyeket a zenei hangoknak feleltetünk meg. A (3.1) képlettel felvázolt megoldás a spektrum minden mintáját besorolja valamelyik zenei hanghoz, melynek hátránya, hogy sok, zenei hangoz nem köthető zaj kerül a PCP vektorba. Felvétel során zaj keletkezhet környezetből, magából a hangszerből, vagy a hangfelvevő eszközből. Tekintve, hogy a vizsgált hangminták a telefon mikrofonjával lesznek felvéve, a zaj nem hagyható figyelmen kívül. Erről a problémáról Fujishima [10] is említést tesz.

A zenei hangok frekvenciája rögzített, ezért egy lehetséges megoldás, hogy csak az adott zenei hang körüli frekvenciatartományban összegezzük az amplitúdóértékeket, ezzel beleszámítva a tartomány összes energiát. A tartomány a hangszer kis mértékű elhangolódásának, illetve a mérési pontatlanságok tűrésére szolgál, mérete megválasztható; annak érdekében, hogy a mély és magas hangokra is egységes szélességű legyen, ezt centben érdemes meghatározni.

A tartományok bevezetése még nem zárja ki a zaj lehetőségét, mert az összegzéssel továbbra is sok felesleges energia kerülhet a chromagramba, ami miatt a figyelembe vett tartományok szélességét nagyon körültekintően kell megválasztani, mely a hibatűrés rovására mehet.

A egyik megoldás valamilyen zajszűrő eljárás alkalmazása. Ma [14] az úgynevezett Soft Thresholding eljárást alkalmazza PCP alapú akkordfelismerő algoritmusában, mellyel sikerült javítania a felismerés pontosságán. A zajszűrés azonban általánosságban egy számításigényes folyamat, jelen dolgozatban pedig az erőforrásigény minimalizására törekszem.

Egy másik megközelítés, hogy az említett tartományokban összegzés helyett maximumkiválasztást végzünk, így minden hanghoz a spektrumnak egyetlen csúcsát rendeljük [17]. Ezáltal minden hanghoz csak a releváns, kiugró amplitúdó értéket vesszük számításba, ugyanakkor megengedjük a hang kis mértékű hamisságát. Ezzel a megközelítéssel megkerülhető az erőforrásigényes, előzetes zajszűrés. A módszert a 3.6. ábra szemlélteti.



3.6. ábra. *a) minták összesítése a tartományokban, b) maximumkiválasztás [17]*

3.2.4. Harmonic Product Spectrum (HPS)

A standard PCP alapú módszer egy hiányossága, hogy az előállított chromavektor - az akkordban szereplő hangok felhangjai miatt - gyakran zajos, a vektor mind a 12 eleme nullától különböző értéket tartalmaz. Ez a zaj tévesztést okozhat a felismerésben, főleg olyan akkordoknál, melyek közös hango(ka)t tartalmaznak.

A legtöbb tévesztés a párhuzamos és az azonos alaphangú[6] dúr-moll párok között lép fel [12]. Egy dúrhoz tartozó párhuzamos moll nála 3 félhanggal lejjebb (kis terc távolságra) helyezkedik el (pl. C-dúr és A-moll), az azonos alaphangú pedig tiszta prímre, így értelemszerűen ugyanarról a hangról kapja a nevét (pl. C-dúr és C-moll). Ezek az akkordpárok két-két közös hangot is tartalmaznak.

Erre a problémára megoldást jelenthet, ha meg tudjuk határozni az akkord alaphangját. Lee [12] és Ma [14] a Harmonic Product Spectrum (HPS) eljárásnak egy variánsát használja erre a célra. A HPS egy egyszerű algoritmus, melyet korábban monofón hangok alaphangjának detektálására használta. Lényege, hogy a spektrum minden amplitúdóértékét megszorozza a frekvencia szerinti egész számú többszöröseivel, azaz minden hang intenzitását a felhangjai intenzitásával súlyozza. Működését a következő képlet írja le [12]:

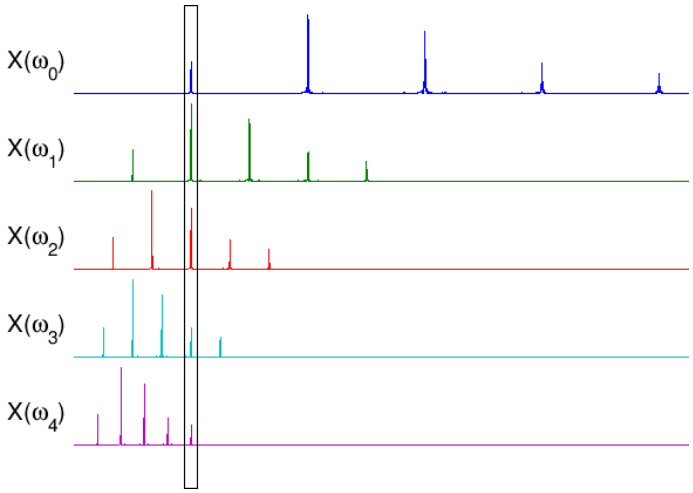
$$HPS(\omega) = \prod_{m=0}^{M-1} |X(m\omega)| \tag{3.5}$$

$$f_0 = \arg\max_{\omega_i} HPS(\omega_i) \tag{3.6}$$

ahol $HPS(\omega)$ a Harmonic Product Spectrum, $X(m\omega)$ a jel DFT-vel nyert spektruma, M azoknak a felhangoknak a száma, melyeket figyelembe szeretnénk venni, f_0 pedig a feltételezett alaphang frekvenciája.

A HPS tehát azokat a hangokat emeli ki, melyeknek sok felharmonikusa szerepel a vizsgált tartományon. Ez monofón hangoknál megfelelő, akkordok esetén azonban az akkordot

[6] Az itt párhuzamosnak nevezett viszonyt angol nyelvterületen *relative*-nek hívják, és az azonos alaphangút nevezik *parallel*nek. Ezek a megnevezések Magyarországon kevéssé ismertek, ezért a továbbiakban kerülni fogom a használatukat.



3.7. ábra. *A HPS működésének illusztrációja [22]*

alkotó hangok gyakran egymás felharmonikusai (ez a dúrra és a mollra is igaz), így pedig a HPS a közös részhangokat több hanghoz is beszámítaná. Az említett források [12, 14] ennek kiküszöbölésére a HPS-nek egy olyan változatát haszálják, amely minden hanghoz csak a 2^m-edik felhangokat veszi számításba, vagyis az adott hang oktávjait:

$$HPS(\omega) = \prod_{m=1}^{M} |X(2^m \omega)| \tag{3.7}$$

Tekintve, hogy zenei hangokkal foglalkozunk, melyeknek rögzített a frekvenciája, a HPS-t elenegdő lehet a kiválasztott spektrális csúcsokra elvégezni a teljes spektrum helyett, ezzel csökkentve a számítási időt.

A HPS-sel történő alaphang keresés feltételezi, hogy az alaphang dominál az akkordhangok közül. Gond akkor adódhat, ha egy akkord valamelyik *fordítása* formájában fordul elő, azaz nem az alaphang a legmélyebb az akkordot alkotó hangok közül.

3.2.5. Spektrum súlyozása

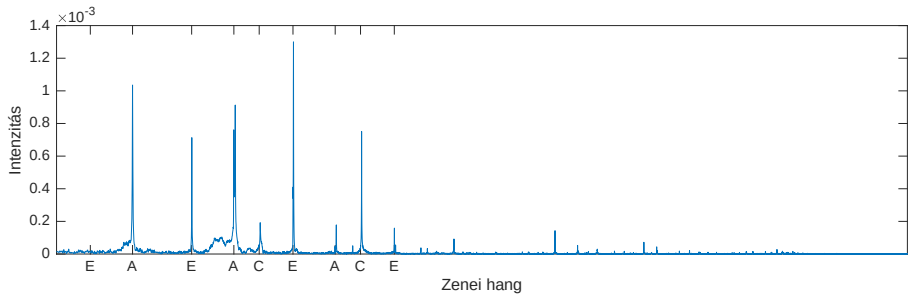
Az alaphang keresésének egy egyszerű módja lehet a spektrum értékeinek súlyozása a hangmagasságuk reciprokával. Ehhez meg kell határozni a spektrum minden értékének a hangmagasságát a frekvenciája szerint, egy alsó referenciapont segítségével (esetünkben ez a C_2), majd az amplitúdóértéket egyszerűen leosztani vele. Alkalmazható egy kijelölt tartományra is, például egy magasabb frekvenciatartományra, melyet kisebb súllyal szeretnénk figyelembe venni.

Ha a centben vett hangmagassággal osztunk le, akkor a súlyozás tartományában az intenzitások nagyságrendekkel kisebb értékeket vesznek fel, mint a tartományon kívül. Ennek kiküszöbölésére a kísérletek során a centérték $\frac{1}{1000}$ részével osztok le. Az így kapott súlyozott spektrumot a cent (1.1) képletének felhasználásával a következő képlettel írhatjuk le:

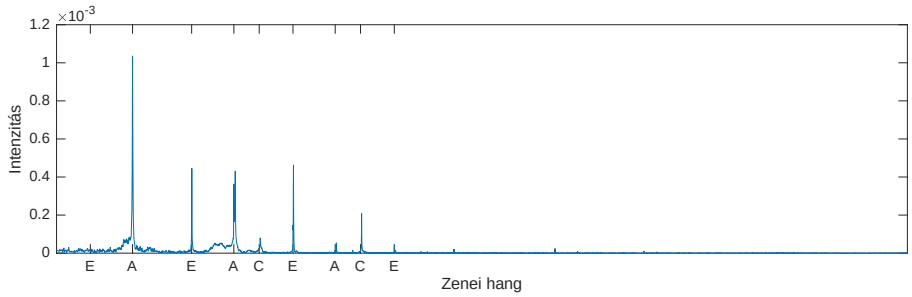
$$X_w(f) = \frac{X(f)}{1{,}2 \cdot \log_2\left(\frac{f}{f_{ref}}\right)} \tag{3.8}$$

ahol $X_m(f)$ az f frekvenciához tartozó súlyozott spektrumérték, az $X(f)$ a hozzá tartozó eredeti spektrumérték, az f_{ref} pedig az alsó referencia-frekvenciaérték, mely esetünkben a már korábban is említett C_2 hang (65.41 Hz).

Az így kapott spektrum eltérését a 3.8. ábra mutatja.



(a) *súlyozás nélkül*



(b) *súlyozás a legmélyebb oktáv kivételével*

3.8. ábra. *Intenzitások változása az A-moll spektrumában a súlyozás hatására*

3.3. Akkordok azonosítása a PCP vektor alapján

A chroma vektor meghatározása és normalizálása után az akkordot valamilyen módon azonosítanunk kell az összetevő hangok alapján.

3.3.1. Mintaillesztés

Fujishima [10] a korábban ismertetett legközelebbi szomszéd és súlyozott összeg módszerek összehasonlításánál arra jutott, hogy a legközelebbi szomszéd módszer a szintetizátoron játszott akkordokat kitűnően felismerte (100%-ban), valódi zeneszámok esetén viszont nem teljesített jól. A szintetizátoros hangokat a súlyozott összeg módszer is felismerte kevés tévesztéssel (96%-os pontosság), ugyanakkor valódi zene esetén is 94%-os teljesítményt nyújtott.

Stark és Plumbley [17] más megközelítést alkalmaz, mely - a legközelebbi szomszéddal szemben - nem érzékeny arra a problémára, hogy a PCP vektor esetenként igen eltérő

értékeket tartalmaz az akkordban szereplő hangok helyén, azok eltérő intenzitása miatt. Egy-egy összehasonlításnál nem azt vizsgálja, hogy a bitmaszk hangjai (ahol 1 értéket vesz fel) milyen intenzitással szerepelnek a chroma vektorban, hanem inkább a maszkban nem szereplő hangok minimális jelenlétét keresi.

Ennek érdekében a PCP és a bitmaszk „komplemensének" skaláris szorzatát számolja ki, melyek közül a legkisebb érték lesz a nyertes választás. Képlete a következő:

$$P_{SP,a} = \frac{\sqrt{\sum_{p=0}^{11} \bar{T}_a(p)(PCP(p))^2}}{(12 - N_a)} \tag{3.9}$$

ahol $\bar{T}_a$ az a akkordhoz tartozó bitmaszk komplemense, azaz $\bar{T}_a(p) = 1 - T_a(p)$ igaz $\forall p \in \{0..11\}$-re; N_a az akkord hangjainak száma, a $12 - N_a$ kifejezés a különböző számú hangot tartalmazó akkordok „esélyeinek" kiegyenlítésére szolgál, esetünkben ez az osztás elhagyható. A többi kifejezés a (3.3) és a (3.4) képleteknek megfelelő.

Ez az eljárás egyúttal a súlyozott összeg módszernél is praktikusabb megoldást nyújt a különböző számú hangot tartalmazó akkordok esélyeinek kiegyenlítésére, hiszen itt nincs szükség a súlyok hosszadalmas, kísérleti alapon történő beállítására.

3.3.2. Rejtett Markov-modellek

A PCP vektor felismerésének másik megközelítése a digitális jelfeldolgozási módszereket statisztikai módszerekkel ötvözi. Sheh és P. W. Ellis [21] publikált egy cikket, melyben az akkordfelismerést rejtett Markov-modellek (Hidden Markov Models, HMM) segítségével végezték.

A HMM egy stochasztikus, véges automata, melynek minden állapotához tartozik egy megfigyelés. Ebben az esetben az állapotok az akkordok, a megfigyelések pedig a PCP vektorok. Az eljárás első fázisa a tanítási folyamat, mely először Fujishima módszeréhez hasonló módon egy PCP vektort alkot a hangmintából, majd a vektort egy HMM paramétereinek beállítására, tanítására használja. A tanítás az úgynevezett *elvárás-maximalizáció* (expectation maximization, EM) algoritmusával történik.

Jelen dolgozatban egy általánosan használható, rövid futásidejű eljárás összeállítására törekszem, a rejtett Markov-modellek módszerét pedig bonyolultsága miatt nem használom.

Létezik neurális hálót alkalmazó módszer [11] is a PCP vektor azonosítására, ezzel azonban hasonló okokból nem foglalkozom a dolgozat keretei között.

4. fejezet

Módszerek tesztelése MATLAB-ban

Az előző fejezetben ismertetett módszerek hatékonyságát MATLAB segítségével teszem próbára, mielőtt felhasználnám őket az Android alkalmazásban. Szándékom egy olyan, a Pitch Class Profile módszerén alapuló megoldás készítése, mely az eljárás különböző fázisaiból egy megfelelően pontos, és relatíve alacsony számításigényű algoritmust alkot.

A kipróbált módszerek mindegyikéhez végrehajtottam egy akkordfelismerési tesztet, minden módszer esetén ugyanazt a hangmintát elemztem. Ezt a hangmintát klasszikus (nylon húros) gitárral, és egy Lenovo Y50 laptop beépített mikrofonjával rögzítettem. A minta 48 akkord megszólaltatását tartalmazza, azaz minden egyes dúr és moll akkord kétszer, két különböző fogással szerepel benne. Minden akkordhoz tartozó minta egységesen 0,3 mp hosszú, köztük 0,3 mp hosszú szünetekkel, a szerkesztés során a minták véletlen egymásra csúszásának elkerülése végett. Ezek a szünetek nem képezik az elemzés részét.

A mintát kétféle ablakmérettel teszteltem, 0,3 mp és 0,1 mp hosszúságúval. A gyakorlatban a 0,3 mp-es ablakméret már elegendő adatfelbontást biztosít az akkordok konzisztens felismeréséhez, az általa okozott késleltetés pedig még tolerálható. A tesztelés során viszont arra is kíváncsi voltam, hogy kisebb ablakméret esetén hogyan változik az egyes módszerek teljesítménye. A 0,3 mp-es ablakméretnél 48; 0,1 mp-esnél pedig 144 az elemzett mintarészletek száma.

A tesztek során mindenhol 44.1 kHz-es mintavételi frekvenciát használtam, a spektrum számításba vett része pedig 4 oktáv terjedelmű, $C\sharp_2$-től C_6-ig terjed, ami a 69.30 Hz és 1046.50 Hz közötti tartománynak felel meg. A számításokat szintén az említett laptopon végeztem (Intel Core i7-4710HQ, 8 GB DDR3L RAM).

Mivel a PCP alapú algoritmus szakaszai egymás eredményét dolgozzák fel, felmerül a kérdés, hogy az egy szakaszhoz tartozó módszerek tesztelésénél a többi szakasz milyen módszer alapján működjön. Igyekeztem ezeket a módszereket úgy megválasztani, hogy a felismerést minél kevésbé befolyásolják, azaz, hogy a vizsgált módszer hatása a felismerés pontosságára minél egyértelműbben látszódjon.

A választott módszerek a következők:

1. *Spektrum súlyozásának szakasza:* nincs súlyozás
2. *A csúcsok kiválasztásnak szakasza:* a zenei hangokat közrefogó, mindkét irányban 20 cent széles tartományokban való összegzés
3. *Ablakozás szakasza:* nincs explicit ablakozás (implicit négyszög ablak)
4. *Mintaillesztés szakasza:* Legközelebbi szomszéd módszere

4.1. Ablakozás

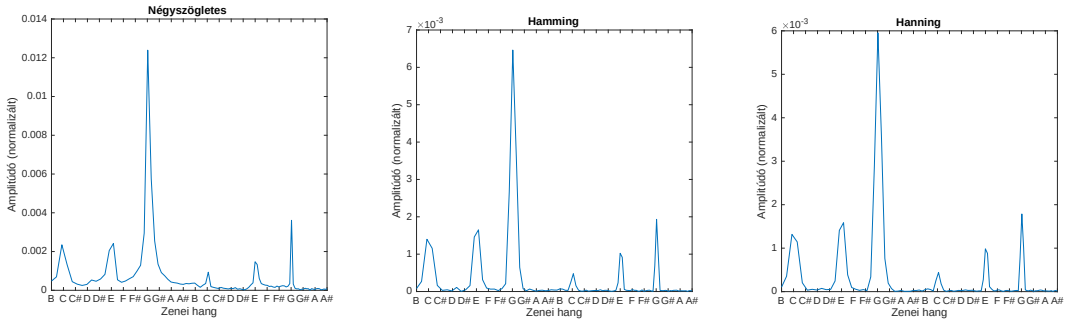
A módszerek tesztelését az ablakozással kezdem, hiszen az ismertetett algoritmus lépései közül ez az egyetlen, mely még időtartományban történik. Tesztelésnél az ablakozást mindenhol átlapolódás nélkül végzem.

A következő ablaktípusokkal vizsgálom a spektrumszivárgást:

1. négyzetes ablak,
2. Hanning ablak,
3. Hamming ablak

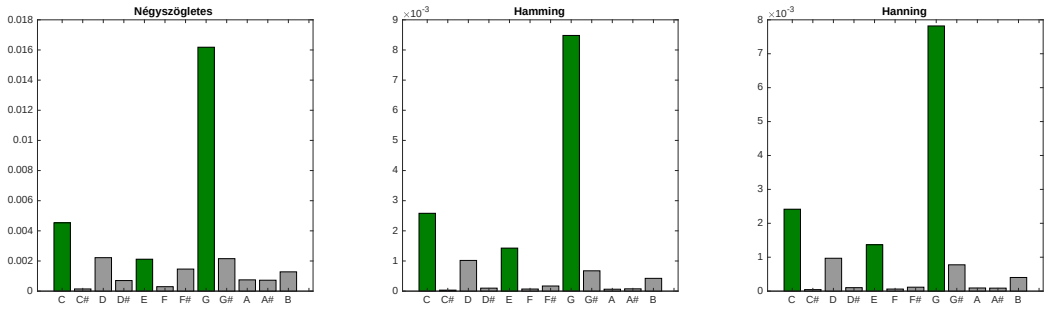
A tesztek során arra a megállapításra jutottam, hogy az ablakfüggvény használata annál jelentősebb, minél kisebb az ablakméret. A spektrumszivárgás szélesebb ablakok esetén is jelentkezik, de ott a jel DFT által érzékelt periódusa hosszabb, így az abból képzett periodikus jelben ritkábbak a szakadások, az akkordhangok pedig ezért általában jobban kiemelkednek a zajból.

Kisebb ablakméret esetén látványosabb a különbség, melyet a 4.1. ábra szemléltet.



4.1. ábra. *C-dúr triád spektruma különböző ablakozási módszerekkel, 0,3 mp hosszú minta esetén*

Az ábráról leolvasható, hogy Hanning és Hamming ablakok használatával a spektrum kisebb értéket vesz fel az akkordhangok csúcsai között. Ez a különbség a spektrumokból előállított chroma vektorokon is észrevehető (4.2. ábra).



4.2. ábra. *C-dúr triád chroma vektorai különböző ablakozási módszerekkel, 0,3 mp hosszú minta esetén*

Az akkordhangok felhangjai persze továbbra is jelen vannak a chroma vektorban (pl. a D, ami a G akkordhang tiszta kvintje, azaz második felhangja); ahogy a hangolási, vagy mérési pontatlanságból származó értékek is (pl. G♯), hiszen ezek nem a spektrumszivárgás következményei. A tényleges akkordhangok ugyanakkor láthatóan könnyebben elválaszthatók a megfelelő ablakozással.

Számításigényük is kedvező, egy 0,3 mp hosszú (13230 mintát tartalmazó) ablak létrehozása és szorzás elvégzése átlagosan 0,29 ms-ot vesz igénybe MATLAB-ban. Bár a dúr és moll akkordok az implicit négyzetes ablakkal is felismerhetők voltak, több felismerendő akkordtípus esetén fontos lehet, hogy a chroma vektor minél pontosabban ábrázolja az akkord hangjait.

A Hanning és Hamming ablakok igen hasonló eredményt értek el, a Hamming ablak kicsivel jobban teljesített, ezért a továbbiakban ezt használom. A számítási igény csökkentése érdekében nem alkalmazok átlapolódást.

4.1. táblázat. *A ablakozási módszerek teszteredményei*

Módszer	Ablakméret (mp)	Találatok száma	Találati arány
négyszög ablak	0,3	46	95,83%
	0,1	75	52,08%
Hamming ablak	0,3	48	100%
	0,1	92	63,89%
Hanning ablak	0,3	48	100%
	0,1	90	62,5%

A 4.1. táblázatból látható, hogy a Hanning és Hamming ablakok használata különösen kisebb ablakméret esetén jelentős javulást ér el a felismerésben.

4.2. Spektrum súlyozása

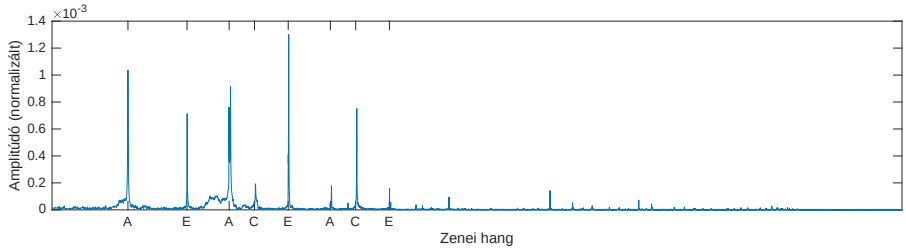
A következő eljárásokat próbálom ki az alsó frekvenciatartomány kiemelésére:

1. Harmonic Product Spectrum,
2. Harmonic Product Spectrum csak a kiválasztott csúcsokra,
3. spektrum súlyozása a hangmagasság reciprokával, magasabb tartományokra,

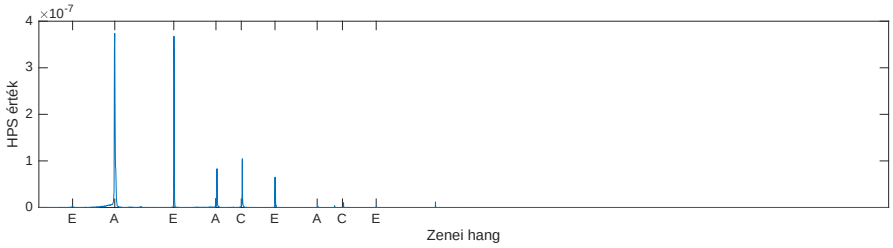
4. energiaspektrum-sűrűség használata

A HPS összességében hasznosnak bizonyult az akkordhangok kiemelésére, az akkordban nem szereplő hangok HPS értékei minimálisak a spektrumértékekhez képest. Ez mindenképp előnyös, mert bár a dúr és moll akkordok azonosítása a HPS alkalmazása nélkül is sikeres volt a tesztelt hangmintákra, egy későbbi bővítés esetén, melyben az alkalmazás többféle akkordot is fel tud majd ismerni, ezeknek a hangoknak a jelenléte tévesztést okozhat.

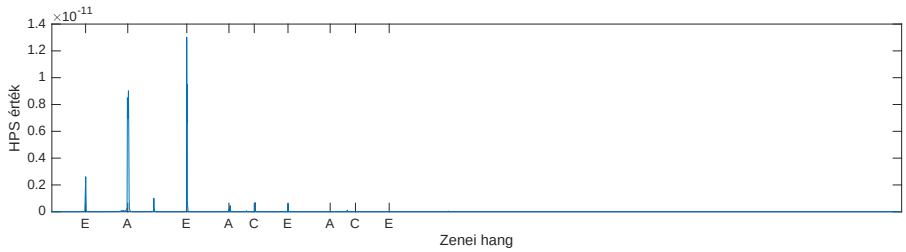
A HPS tesztelése során ugyanakkor arra a megállapításra jutottam, hogy gitár esetén nem számíthatunk arra, hogy az akkord alaphangjának intenzitása dominál, még akkor sem, ha az oktávjait is figyelembe vesszük. Ahogy az 1.2. ábrán is láthattuk, a gitár húrjai egyenként is felharmonikusokban gazdag hangot keltenek, az akkordok pedig ezeket a felhangokat tovább erősítik - például a dúr akkordokban szereplő nagy terc és tiszta kvint a felhangsorban is elöl szerepel. Ráadásul gitáron egy dúr, vagy moll triádot gyakran öt, vagy hat húr megpengetésével szólaltatnak meg, ami azt jelenti, hogy egy-egy hang tiszta oktávos különbséggel több húron is megszólal, ezzel nagyobb intenzitáshoz jutva a spektrumban. A nyitott E-dúr akkord például 6 húron szólal meg, az E alaphang 3 húron is megcsendül, míg a G♯ csak 1 húron. A HPS az ilyen helyzetben lévő hangokat aránytalanul kiemeli.



(a) spektrum, HPS nélkül



(b) HPS, $M = 1$



(c) HPS, $M = 2$

4.3. ábra. *A-moll spektruma, és Harmonic Product Spectruma* $M = 1$ *és* $M = 2$ *esetekre*

A 4.3. ábrán látható, ahogy az A-moll esetén a tiszta kvintet jelentő E hang kapja a legmagasabb HPS értéket, míg a kis tercet jelentő C hang szinte teljesen eltűnik az ábráról. Az ablakméret - és így az adadfelbontás - csökkentésével pedig egyre nő a kockázat, hogy valamelyik akkordhang HPS értéke jelentéktelenre csökken egyetlen kisebb csúcs miatt. A HPS összességében nem befolyásolta az akkordfelismerés pontosságát, a vele kapcsolatban felmerülő aggályok miatt pedig nem alkalmazom.

A másik, HPS gondolatmenetén alapuló megoldás, hogy a HPS műveletét a csúcsok kiválasztása után hajtja végre, ezzel - a pontosság csökkentésének fejében - csökkentve a számítási időt. A HPS problémái azonban itt is jelentkeznek, a pontatlanság pedig csak ront a chroma vektor felismerhetőségén, így ezt a módszert sem alkalmazom.

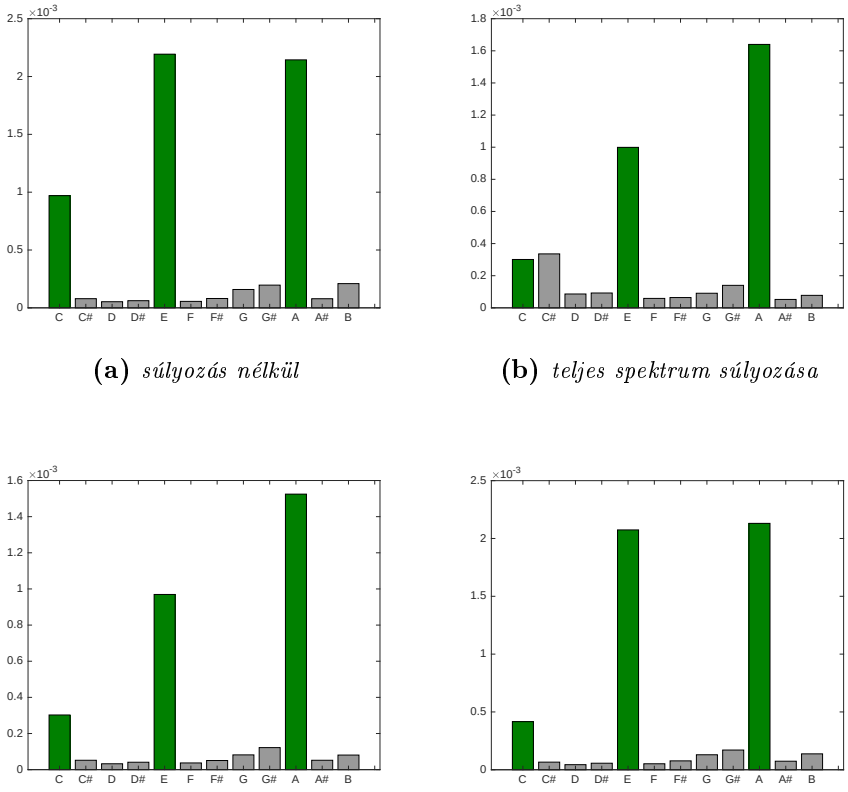
A spektrum amplitúdóértékeinek (3.2.5 pont szerinti) leosztása is hasonló problémát okoz, mint a HPS. Vannak esetek, amikor valóban kiemeli az alaphangot, ehhez azonban a súlyozás tartományának optimális beállítására lenne szükség, mely minden akkord esetén különböző - ez a hallott akkord felismerése előtt nyilván nem alkalmazható stratégia. Emellett arra figyeltem fel, hogy a terc jelenlétét csökkenti a chroma vektorban, szinte minden esetben. Ennek oka lehet, hogy gitáron a terc dúr és moll akkordok esetén is gyakran egy oktávval feljebb csendül fel, mint az alaphang és a kvint (azaz az alaphang tiszta oktávjának a kis, vagy nagy terce szólal meg). Emiatt ez a módszer óhatatlanul a terchez tartozó értéket fogja csökkenteni a chroma vektorban, ami probléma, hiszen ez a hang különbözteti meg az azonos alaphangú dúr és moll akkordokat. A jelenséget a 4.4. ábra szemlélteti, ahol A az alaphang, C a kis terc és E a tiszta kvint.

A spektrum súlyozására a legegyszerűbb módszer a bemutatottak közül az energiaspektrum-sűrűség használata a spektrum helyett. Ez a módszer már Fujishima eljárásában is szerepel, a kiszámításához a spektrum minden egyes mintáját az abszolútértékének négyzetére cserélünk. Ezzel csak a kisebb és nagyobb intenzitások közti eltérést növeljük.

A tesztelés során született eredményeket a 4.2. táblázat szemlélteti. A táblázatból kiolvasható, hogy a súlyozatlan állapothoz képest egyedül az energiaspektrum-sűrűség használata tudott kisebb javulást elérni, 0,1 mp-es ablak esetén. A többi eljárás inkább csak rontott a felismerés pontosságán.

4.2. táblázat. *A spektrumot súlyozó módszerek teszteredményei*

Módszer	Ablakméret (mp)	Találatok száma	Találati arány
súlyozás nélkül	0,3	46	95,83%
	0,1	75	52,08%
HPS (m=2)	0,3	42	87,5%
	0,1	42	29,17%
HPS a kiválasztott csúcsokon (m=2)	0,3	44	91,67%
	0,1	73	50,69%
súlyozás a hangmagasság reciprokával (csak a legmagasabb oktávra)	0,3	39	81,25 %
	0,1	35	24,3%
energiaspektrum-sűrűség	0,3	45	93,75%
	0,1	83	57,64%



(a) *súlyozás nélkül*

(b) *teljes spektrum súlyozása*

(c) *súlyozás a legmélyebb oktáv kivételével*

(d) *súlyozás csak a legmagasabb oktávra*

4.4. ábra. *A-moll chroma vektora különböző súlyozásokkal*

4.3. A csúcsok kiválasztása

A korábban 3.2.3 pont alatt ismertett megoldások közül hármat teszteltem:

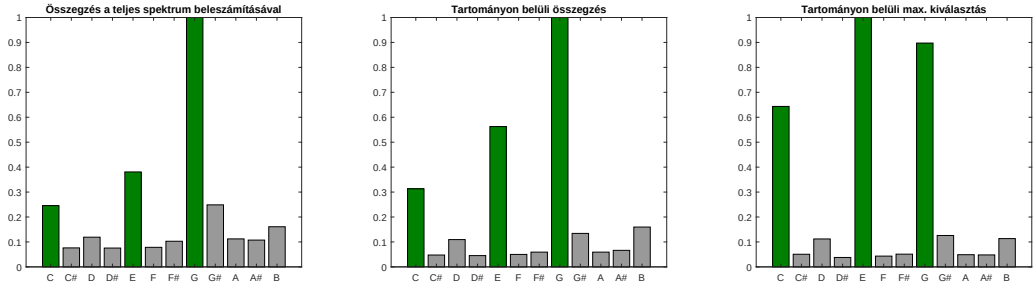
1. a teljes spektrumot figyelembe vevő összegzést,
2. a tartományokban való összegzést,
3. és a tartományokban való maximumkiválasztást

Ezek közül végül Stark és Plumbley maximumkiválasztásos módszere mellett döntöttem. A 4.5. ábrán láthatók a három módszer által létrehozott chroma vektorok.

Az akkord hangjait zöld szín jelzi, a használt minta pedig 2 mp hosszú. A tartományok minden irányban 20 cent szélesek, mely a gyakorlatban megfelelőnek bizonyult.

Az ábrákról leolvasható, hogy a tartományok bevezetése csökkentette a zajt, azaz az akkordban nem szereplő hangok értékeit. A 2. és 3. módszerek között elsősőrban az akkordhangok egymáshoz mért intenzitásában figyeltem meg különbséget.

Csendes körülmények között a két módszer nagyon hasonlóan teljesít, különösen kisebb (0,2-0,3 mp hosszú) ablakméretnél, ahol az egyes tartományok kevesebb amplitúdóértéket tartalmaznak. A felismerés pontosságát ezért jelen feladatban nem befolyásolja, hogy a



4.5. ábra. *C-dúr triád chroma vektorai különböző csúcsválasztásos módszerekkel (2 mp hosszú minta esetén)*

kettő közül melyiket választom. A döntést a későbbi továbbfejleszthetőségi lehetőségeket szem előtt tartva hozom meg.

A 3. megoldás kisebb esélyt hagy a zaj bekerülésének, és nem részesíti előnyben azokat a hangokat, melyeknél a spektrum - akár hangolási, akár mérési pontatlanságok miatt - egymás mellett több magas amplitúdóértéket tartalmaz, azaz ahol a csúcs „szoknyája" szélesebb (lásd pl. a 3.3. ábra esetén a legmagasabb amplitúdójú, G hanghoz tartozó csúcsot). Ezért választom ezt a megoldást.

A teszt eredményeit a 4.3. táblázat mutatja. A vele kapcsolatban felmerült aggályok ellenére a teljes spektrum figyelembevétele összességében jobban teljesített 0,1 mp-es ablakméretre. A nagyobb toleranciája kisebb felbontású FFT esetén kedvező is lehet.

4.3. táblázat. *A csúcskiválasztásos módszerek teszteredményei*

Módszer	Ablakméret (mp)	Találatok száma	Találati arány
összegzés a teljes spektrumon	0,3	45	93,75%
	0,1	97	67,36%
összegzés tartományokban	0,3	46	95,83%
	0,1	75	52,08%
max. kiválasztás tartományokban	0,3	46	95,83%
	0,1	76	52,77%

4.4. Mintaillesztés

A korábban ismertetett mintaillesztő megoldások közül a következőket próbálom ki:

1. a legközelebbi szomszéd módszerét,
2. a skaláris szorzatot*,
3. a 3.3.1 pontban ismertetett skaláris szorzat változatot

*A 3.1 pontban leírt súlyozott összeg módszere a súlyok beállításához az akkord hangjainak számát, és előfordulási gyakoriságát veszi figyelembe. Mivel csak a két leggyakoribb akkordtípussal foglalkozunk, melyek mindketten hármashangzatok, nincs értelme ilyesfajta súlyozást alkalmaznunk, így ez a módszer maximális skaláris szorzat kereséssé redukálódik.

A 3.3.1 pontban tárgyalt, Stark és Plumbley által megfogalmazott problémák a legközelebbi szomszéd és a súlyozott összeg módszereket illetően akkor állnak fent, ha különböző számú hangot tartalmazó akkordokat kell felismernünk. A három módszer teljesítményében - dúr és moll triádok felismerését illetően - nem tapasztaltam különbséget, sem számítási idő, sem felismerési pontosság szempontjából.

Ahogy a 4.4. táblázat is mutatja, a három eljárás pontossága között igen csekély az eltérés, ha csak dúr és moll akkordokat kívánunk felismerni.

4.4. táblázat. *A mintailleszési módszerek teszteredményei*

Módszer	Ablakméret (mp)	Találatok száma	Találati arány
legközelebbi szomszéd módszere	0,3	46	95,83%
	0,1	75	52,08%
skaláris szorzat	0,3	46	95,83%
	0,1	75	52,08%
Stark és Plumbley módszere	0,3	46	95,83%
	0,1	74	51,39%

A program későbbi bővítését szem előtt tartva Stark és Plumbley módszerét használom a végleges programban, hiszen a négyeshangzatok bevonása után ez nagyobb robosztusságot biztosít.

4.5. A kiválasztott módszerek

Az előző pontokban kiválasztott módszereket a 4.5. táblázat foglalja össze. A végső döntéshez a tesztek eredményein kívül a a felhasznált források megállapításait is figyelembe vettem, melyek a program későbbi bővítése során játszhatnak nagyobb szerepet.

4.5. táblázat. *Az Android-alkalmazásban megvalósítandó módszerek*

Szakasz	Módszer
Ablakozás	Hamming ablak átlapolódás nélkül
Spektrum súlyozása	energiaspektrum-sűrűség
Csúcsok kiválasztása	maximumkiválasztás tartományokban
Mintaillesztés	Stark és Plumbley módszere

A fenti szakaszokból összeállított eljárás felismerési pontossága 0,3 mp-es ablakméret esetén **100%**, 0,1 mp-es ablak esetén pedig **62,5%**.

5. fejezet

Az Android bemutatása

Az Androidot bemutató részek a [18] forrás alapján készültek, az egyéb forrásból származó gondolatokat külön jelöltem.

Napjaink egyik legelterjedtebb operációs rendszere a Google gondozásában készült Android. A Google 2005-ben felvásárolta az Android Incorporated nevű céget, majd megkezdte saját, mobiltelefonokra szánt operációs rendszere fejlesztését, melynek első verziója 2008-ban került a piacra.

Az Androiddal elsősorban okostelefonokon találkozhatunk, de számos más eszközön is megjelenik, például táblagépeken, televíziókon, háztartási eszközökön. Kényelmesen használható olyan eszközökön, ahol korlátozott erőforrás áll rendelkezésre, és a felhasználói interakció elsősorban nem egérrel és billentyűzettel, hanem érintőképernyőn keresztül történik.

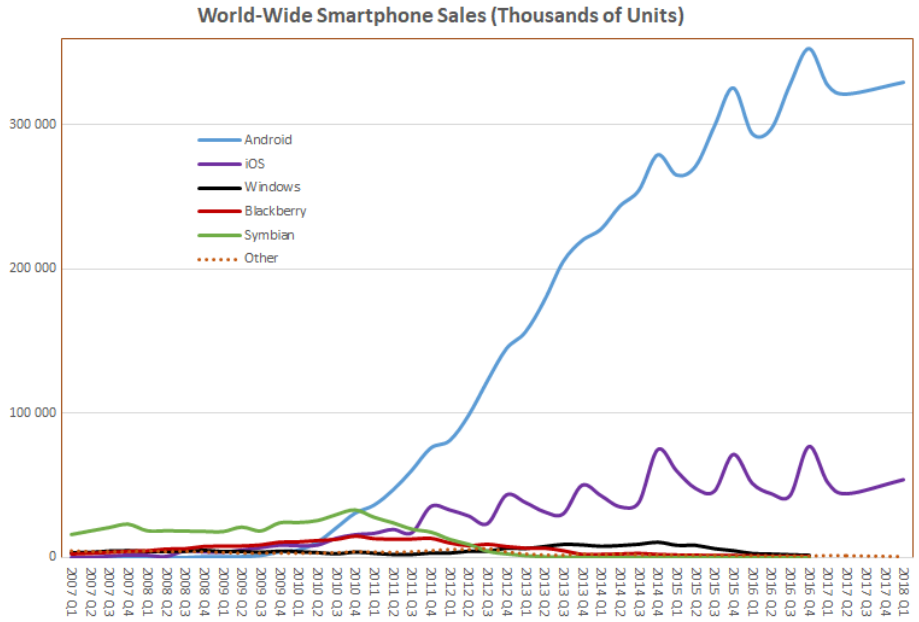
Az Android sikerének okai közé sorolható, hogy ingyenes és nyílt forráskódú, melynek köszönhetően rengeteg gyártó árusítja vele az eszközeit. Legnagyobb konkurensével, az Apple iOS-szel ketten jelenleg uralják a mobil operációs rendszerek piacát. Az iOS-szel szemben az Android az okostelefonok minden árkategóriájában megtalálható, az egyszerű alapmodellektől kezdve egészen a csúcskategóriáig. Emiatt jóval több felhasználóhoz jut el világszerte versenytársánál (5.1. ábra).

Az Androidot futtató készülékek sokfélesége egyben hátrányt is jelent, hiszen így nem lehet garantálni, hogy ugyanaz az alkalmazás megfelelően fog működni egy korlátozottabb hardverrel rendelkező eszközön is.

5.1. Platform verziók

Az Android fejlődése megjelenése óta töretlen, ami az okostelefonok gyors fejlődése miatt szükséges is. Első kiadása óta számos frissítést megért, a verziókat a 2009-ben megjelent 1.5 (Cupcake) verzióval kezdődően édességekről nevezik el, melyek ábécérendben követik egymást (5.1. táblázat). A jelenleg legfrissebb verzió az Android 9 (Pie).

Az újabb és újabb verziók egyre fejlettebb API-t (application programming interface) bocsátanak a fejlesztők rendelkezésére, melyek az újításaikkal megkönnyítik a fejlesztést,



5.1. ábra. *Okostelefon eladások világszerte a Gartner adatai alapján [27]*

esetleg új fejlesztési ajánlásokat vezetnek be. A verziók visszafelé kompatibilisek egymással, így egy korábban megírt alkalmazás a frissebb Android rendszereken is garantáltan futni fog.

A gyors léptékű fejlődés következménye, hogy az egyidejűleg forgalomban lévő készülékek esetenként nagyon eltérő Android verziót használnak. Az 5.1. táblázatban látható, hogy az Oreo verzió az Androidos készülékek még mindig alig több, mint 20%-án van jelen, pedig már az eggyel újabb Android Pie is folyamatosan elérhetővé válik a különböző készülékekre. Az alkalmazásfejlesztés során ezért szempont lehet, hogy egy régebbi, széles körben támogatott API funkcióit használjuk a legújabb helyett, hiszen így több felhasználóhoz juthat el az alkalmazásunk.

5.1. táblázat. *A különböző Android verziók eloszlása (2018. december) [7]*

Verziószám	Kódnév	API	Eloszlás
2.2	Froyo	8	<0.1%
2.3.3 - 2.3.7	Gingerbread	10	0.2%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.1%
4.2.x		17	1.5%
4.3		18	0.4%
4.4	KitKat	19	7.6%
5.0	Lollipop	21	3.5%
5.1		22	14.4%
6.0	Marshmallow	23	21.3%
7.0	Nougat	24	18.1%
7.1		25	10.1%
8.0	Oreo	26	14.0%
8.1		27	7.5%
9.0	Pie	28	<0.1%

5.2. A platform főbb technikai jellemzői

Az Android rendszer alapja egy monolitikus Linux-kernel, a memóriakezelésért, a feladatok ütemezéséért az alacsony fogyasztást elősegítő teljesítménykezelésért felel. A Linux miatt Androidra az alkalmazásfejlesztés elsősorban Java nyelven történik. Az alacsonyabb szintű funkciók eléréséhez azonban natív C++ kód használatára is lehetőség van, 2017 októbere óta pedig a Google a Kotlin nyelvet is támogatja az alkalmazások fejlesztéséhez. Az alkalmazások futtatása a Javánál megszokott módon kezelt környezetben, virtuális gépen történik[1].

5.3. Az alkalmazások felépítése

Az Android alkalamzások építőelemeit komponenseknek nevezik. A komponensek külön felülettel rendelkeznek, melyen keresztül a rendszer el tudja indítani őket, feladatuk pedig általában jól elkülöníthető. Egymáshoz lazán kapcsolódnak, egy alkalmazásnak elég lehet akár egyet is tartalmaznia belőlük. A komponenseknek négy különböző típusa van, melyek a következők:

- *Activity*
- *Service*
- *ContentProvider*
- *BroadcastReceiver*

Az *Activity* a leggyakrabban használt komponenstípus. Saját grafikus felhasználói felülete van, általában az alkalmazás egy jól elhatárolható funkciójáért felel. Némileg hasonló szerepet tölt be, mint az asztali operációs rendszerek esetén az ablakok.

A *Service* komponensek a háttérben (huzamosabb ideig) futó szolgáltatásokat jelképezik. Önmagukban nem rendelkeznek felhasználói felülettel, de jelezhetnek a felhasználónak például értesítés küldésével, vagy egy Activity indításával.

A *ContentProvider* komponensek adatforrások kezeléséért felelnek. Egyfajta interfészt nyújt, melyen keresztül a folyamatok hozzáférhetnek a legkülönbözőbb adatforrásokhoz, például egy, a telefon tárhelyén lévő adatbázishoz, a telefonkönyvhöz, a naptárhoz, vagy akár internetes adatforráshoz.

A *BroadcastReceiver* egyfajta eseménykezelő, mely az Android rendszer *broadcast*-tal (üzenetszórással, sugárzással) jelzett eseményei hatásásra képes aktiválódni, és végrehajtani a neki megadott feladatot. Ilyen broadcast üzenet például az alacsony akkumulátortöltöttség, vagy a beérkező telefonhívás.

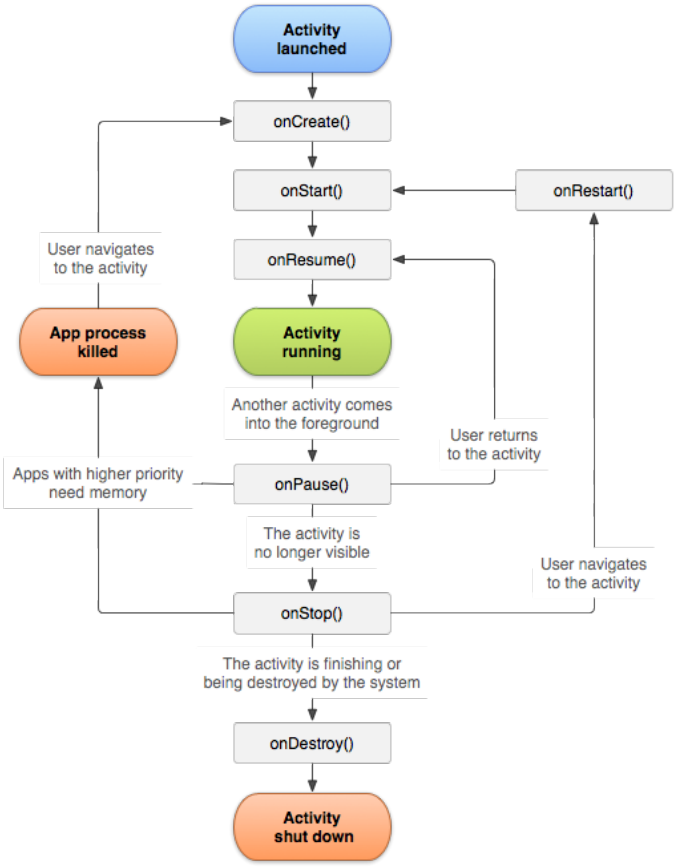
A Java - és így az Android API is - erősen objektumorinetált, így nem meglepő, hogy a komponensek mindegyike osztályként jelenik meg az API-ban. Ahhoz, hogy egy komponenst a saját igényeink szerint meg tudjunk tervezni, örökléssel le kell származtatnunk a

[1] Ez a Dalvik virtuális gép, mely a Java Virtual Machine jelentősen átdolgozott, optimalizált verziója

saját komponensünk osztályát az eredetiből. Ezzel számos metódust is öröklünk, melyeket természetesen felülbírálhatunk (override).

A fenti komponenstípusok közül csak az Activity-t használom az alkalmazásban, a többi részletezésével ezért nem foglalkozom.

Az Activity komponens saját életciklussal rendelkezik, ami meghatározza, hogy mi történik vele bizonyos felhasználói interakciók esetén, mint például az Activity indítása, átnavigálás egy másik Activity-re, visszatérés az Activity-re, stb. Az Activity életciklusát az 5.2. ábra szemlélteti.



5.2. ábra. *Az Activity életciklus-modell [3]*

Ahogy azt az ábra is sejteti, az Activity életciklusának változtatása az `Activity` osztály metódusain keresztül történik. Például az Activity indítása sorban az `onCreate()`, `onStart()` és `onResume()` metódusok hívását jelenti, míg egy másik Activity előtérbe kerülése az `onPause()` metódus lefutásával jár. Ezek a metódusok határozzák meg az Activity viselkedését az adott eseményekre. Jelentőségük abban rejlik, hogy felülbírálhatók, ezáltal az alkalmazásfejlesztő testreszabhatja ezt a viselkedést[2]. Ezeket a függvényeket a rendszer automatikusan meghívja, amikor a kiváltó esemény bekövetkezik, a fejlesztőnek nem kell explicit hívást intéznie.

[2]A felülbírált függvénynek kötelessége meghívnia az eredeti, őstől örökölt metódust, így az ős viselkedését kibővítheti, de nem írhatja felül.

5.4. Hang rögzítése az Android API segítségével

Az Android API alapvetően kétféle lehetőséget is kínál hang rögzítésére egy alkalmazásban:

- `MediaRecorder` osztály
- `AudioRecord` osztály

A MediaRecorder [8] elsősorban alapszintű használtra való. Hang- és videófelvételt is lehet vele igen egyszerűen készíteni, melyet a kiválasztott tömörített formátumban tárol el, a készülék tárhelyén. Ez az osztály elsősorban akkor hasznos, ha a felvételt lejátszható formában szeretnénk hosszabb távon tárolni. Jelfeldolgozási célokra viszont nem célszerű a használata a felesleges tömörítés és fájl írás műveletek miatt.

Az AudioRecord [5] jóval alacsonyabb szintű műveleteket kínál. Segítségével nyers audió adatot olvashatunk be a mikrofonról, a beolvasott adatok nem kerülnek fájlba, hanem a memóriában tárolódnak tömb, vagy Java `Buffer` formájában. Az AudioRecord az alkalmas választás, ha az adatokat azonnal, valós időben fel szeretnénk dolgozni. Használata ugyanakkor nagyobb odafigyelést is igényel a MediaRecordernél, mert a fejlesztő feladata, hogy az alkalmazás megfelelő gyakorisággal meghívja beolvasó függvényt, mely az AudioRecord példánytól lekéri az adatokat. Ha ez nincs biztosítva és a buffer megtelik, adatvesztés történhet, a felvételbe szakadások kerülhetnek. Az implementáció során az AudioRecord osztályt fogom használni.

5.5. Késleltetési problémák

Az Android rendszer egy régóta ismert hiányossága az audiórendszer magas késleltetése. Ahhoz, hogy egy készülék a hangtechnikában a professzionális („pro audio") követelményeknek megfeleljen, a rendszer körülfordulási késleltetése (round-trip latency) nagyjából 10 ms lehet, vagy kevesebb. Ez az audiójel „átfutási" ideje a rendszeren, vagyis a jel mikrofonon keresztül való beérkezése, és a hangszórón való lejátszása között eltelt idő.

Az iOS a 10 ms-ra vonatkozó feltételt már régóta teljesíti, Androidon azonban sokáig nem volt lehetőség a késleltetés elfogadható mértékű csökkentésére. Az Android 5.0 (Lollipop) rendszerrel érkező HTC Nexus 9 a megjelenésekor erősnek számító hardvere ellenére is csak 35 ms késleltetéssel tudott működni, egyes készülékek esetén ez az idő a tizedmásodperces nagyságendet is elérte [23]. Emiatt sok, az iOS-en jól működő, alacsony audiókésleltetést igénylő alkalmazás meg sem jelent Androidra. A jelenséggel és a technikai okaival a Superpowered [24] foglalkozott részletesen, ezzel felhívva mind a felhasználók, mind a Google figyelmét a problémára.

Az Android 6.0 (Marshmallow) már jelentős javulást ért el, ezzel a rendszerrel a Nexus 9 késleltetése 15 ms-ra csökkent a korábbi 35 ms-ról. A rendszer hardverkezelésének újragondolásán túl a Google azóta több lehetőséget is kínál a fejlesztőknek, mellyel az alkalmazásaik késleltetését csökkenthetik [4].

6. fejezet

Implementáció Androidon

A különböző módszerek tesztelése után a kiválasztott eljárásokat egy Android alkalmazásba ültettem. A fejlesztés során célom az volt, hogy a program az Androidos készülékek széles választékával kompatibilis legyen, megfelelő teljesítménnyel működjön, emellett letisztult, könnyen kezelhető felülettel rendelkezzen.

Az alkalmazás tesztelését egy Xiaomi Mi A2 Lite készüléken végeztem, melyre *tesztesz-köz*ként fogok hivatkozni a továbbiakban. Fontosabb specifikációi a következők:

- *Chipset:* Qualcomm Snapdragon 625
- *Processzor:* 8 x 2 GHz, Cortex-A53
- *RAM mérete:* 4 GB
- *Rendszer verzió:* Android 8.1 (Oreo)

6.1. Az alkalmazás felépítése

Az alkalmazás egyetlen Activity-ből épül fel, melyet `MainActivity`-nek nevezek. A program indítása után egy „Start” gomb és egy „Press start” felirat fogadja a felhasználót. A gomb megnyomásával indítható el a mikrofonon figyelése, a felismert akkord neve pedig a képernyő tetején, a „Press start” felirat helyén jelenik meg.

Ha épp nem észlelhető semmilyen akkord, az utolsó észlelt akkord neve elhalványodik a képernyőn. Amennyiben az akkordok pontozása közben a második és harmadik helyet elérő akkordok pontszáma csak 10%-ban tér el az előttük lévőtől, úgy ezek az akkordok is megjelennek a képernyőn, pontszámuk szerinti sorrendben.

A mikrofon figyelése a gomb újbóli megnyomásával állítható le.

Az alkalmazás a reszponzív működés érdekében összesen három külön szálat használ:

1. A grafikus felületet (graphical user interface, GUI) kezelő szál (főszál)
2. A hangfelvételt végző szál
3. A hangminta feldolgozását végző szál

A grafikus felület eseményeinek kezelése mindenképpen más szálon kell, hogy történjen, mint a mikrofon figyelése, hiszen a mikrofon használatát a gomb segítségével csak így lehet leállítani.

Annak érdekében, hogy az alkalmazás gyengébb hardver esetén se okozzon túlfutást a mikrofon bufferében, a jelfeldolgozást és akkordfelismerést egy külön szálnak delegáltam.

A programkódban ezek a szálak külön osztályokként jelennek meg, melyek a Java `Runnable` interfészét valósítják meg. Emiatt a továbbiakban a szálak említésével egyúttal a hozzájuk tartozó osztályokra is utalok.

6.1.1. Főszál

Az alkalmazás indítása a főszál indítását jelenti, amely betölti az alkalmazást jelentő Activity-t és a rajta lévő GUI elemeket. Egy Activity tartalmát, felületét XML fájlok írják le, a kódban pedig az XML fájlokban megadott azonosítókkal hivatkozhatunk az elemekre. Ez a függény eltárolja ezeket a hivatkozásokat. Szintén ez a szál rendeli hozzá a GUI elemekhez az eseménykezelőket, jelen esetben ez a Start/Stop gomb eseménykezelőjét jelenti.

Fontosabb metódusok:

- `void onCreate()`:
  Az alkalmazás belépő pontja a `MainActivity` `onCreate()` metódusa. Ebben a függvényben töltődnek be az XML-ben definiált GUI elemek, azaz a Start/Stop gomb és az akkordok nevét tartalmazó `TextView` objektumok. Ezek közül induláskor még csak az elsőnek van szöveges tartalma, a többi üres sztringet tartalmaz.

  Szintén ebben a metódusban kerül sor az alkalmazás működéséhez szükséges engedélyek kérésére is a felhasználótól. Ez jelen esetben csak a hangrögzítéshez való jogot jelenti. Amennyiben a felhasználó nem engedélyezi a mikrofonhoz való hozzáférést, az alkalmazás bezárul, hiszen ez az engedély elengedhetetlen a működéséhez.

- `void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults)`:
  Ez szintén az `Activity` osztály egy felülbírált metódusa. Ez a függvény automatikusan meghívódik, amikor a felhasználó döntést hoz az engedélyek megadásáról. Jelen esetben ez a függvény bezárja az Activity-t, ha a felhasználó megtagadja a szükséges engedélyt.

- `void View.OnClickListener.onClick()`:
  Egy grafikus elemhez úgy rendelhetünk kattintás-eseménykezelő függvényt, hogy meghívjuk a `setOnClickListener()` függvényét egy olyan objektumot átadva paraméterként, ami megvalósítja az Android API beépített `View.OnClickListener` interfészét. Az eseménykezelőt ezen interfész `onClick()` metódusa definiálja, ez a függvény fog lefutni minden alkalommal, amikor az esemény bekövetkezik. Esetünkben a gomb viselkedése attól függ, hogy épp folyamatban van-e a mikrofon hallgatása.

Ha nem nincs folyamatban a mikrofon figyelése:

- Elindítja a hangrögzítő szálat
- Letiltja a képernyő automatikus kikapcsolását
- A gomb feliratát átállítja „Stop”-ra

Ha folyamatban van a figyelés:

- Leállítja a mikrofont figyelő szálat
- Engedélyezi a képernyő automatikus kikapcsolását
- A gomb feliratát visszaállítja „Start”-ra

A képernyő kikapcsolásának engedélyezése és letiltása az Android API `FLAG_KEEP_SCREEN_ON` nevű allapotjelzőjének beállításával lehetséges.

```
//Képernyő kikapcsolásának tiltása:
getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

//Kikapcsolás engedélyezése:
getWindow().clearFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
```

6.1.2. Hangrögzítő szál

Ez a szál felelős az `AudioRecord` objektum létrehozásáért és a pufferek kezeléséért. Végtelen ciklusban lekérdezi az audióadatokat az `AudioRecord` objektmtól. Annak érdekében, hogy a mikrofon belső puffere ne csorduljon túl, ez a szál nem dolgozza fel a kinyert audióadatokat, amint egy alblakméretre való adat áll rendelkezésre, új szálat indít az ablak feldolgozására.

A hangminta beolvasása az `AudioRecord.read()` függvényeivel lehetséges, melynek többfajta túlterhelése (overload) is létezik [5]. Ezek a függvények lehetővé teszik a hangrészlet adatainak eltárolását egy, a bitmélységnek megfelelő típusú tömbben (8 bit esetén ez a `byte`, 16 bit esetén ez a `short`, stb.), vagy egy `ByteBuffer` típusú objektumban, mely a Java API részét képezi.

A `ByteBuffer` előnye, hogy lehetővé teszi a natív memóriaműveleteket a JVM megkerülésével[1], ezáltal gyors I/O műveleteket biztosítva. Hátránya azonban, hogy használata jóval több hibalehetőséget rejt az egyszerű tömbökénél, mert olyan döntéseket bíz a feljesztőre, melyekre a Java használatakor általában nem kell ügyelnie. Ezekkel a hibalehetőségekkel bővebben a 6.2 pontban foglalkozom. A hibák elkerülése végett kerülöm a `ByteBuffer` használatát.

[1]Csak abban az esetben, ha *direct* buffert használunk. Részletesebben lásd: [6])

Az `AudioRecord` inicializálásához a konstruktorban meg kell adni a felvétel forrásával és formátumával kapcsolatos paramétereket. Az Android API definiál konstansokat erre a célra, hogy ne kelljen „mágikus számokat[2]" használni a kódban. A következő paramétereket használom:

- Felvétel forrása: `MediaRecorder.AudioSource.MIC` (beépített mikrofon)
- Mintavételezési frekvencia: `44100` Hz
- Csatornák száma: `AudioFormat.CHANNEL_IN_MONO` (mono)
- Bitmélység: `AudioFormat.ENCODING_PCM_16BIT` (16 bit)

A minimálisan megengedett pufferméret az `AudioRecord.getMinBufferSize()` függvénnyel kérdezhető le, a valódi pufferméret az alkalmazásban egy konstans érték beállításával szabályozható, melyet másodpercben kell megadni. A program az itt megadott értéket automatikusan felkerekíti a minimális pufferméret legközelebbi többszörösére.

A fent megadott 44.1 kHz mintavételezési frekvencia és a 16 bites PCM bitmélység nemcsak gyakori értékek digitális hangfelvételek esetén, de az egyetlen formátumok, melyeket garantáltan támogat minden Androidos eszköz.

Fontosabb metódusok:

- `void run()`:
  Ez a szál fő metódusa, a szál indításával ez a függvény hívódik meg. Feladata az `AudioRecord.read()` folyamatos hívása, majd egy feldolgozó szál indítása az ablakot jelentő puffer átadásával.

6.1.3. Feldolgozó szál

Ez a szál felelős a hangminta elemzéséért, majd az akkord felismeréséért - minden akkordfelismerési eljárás ezen a szálon hajtódik végre.

A feldolgozás a következő lépésekben történik minden egyes ablakra:

1. jel szorzása Hamming ablakkal
2. spektrum számítása FFT-vel
3. energiaspektrum-sűrűség számítása a spektrumból
4. frekvencia számítása a spektrum minden értékére
5. hangmagasság számítása a frekvenciából minden értékre
6. binek létrehozása minden zenei hang számára, majd a releváns spektrumértékek összegyűjtése (nem összegzése)

[2]azaz tisztázatlan jelentésű számokat, melyekről a kód alapján nem lehet tudni, hogy miért épp annyi az értékük, amennyi

7. zenei hangok intenzitásának meghatározása a binekben való maximumkiválasztással
8. chroma vektor létrehozása az azonos hanghoz tartozó binek értékeinek összegzésével
9. chroma vektor normalizálása
10. pontszám meghatározása minden akkordra mintaillesztéssel
11. pontszámok rendezése növekvő sorrendbe
12. nyertes akkord megjelenítése a képernyőn
13. második és harmadik helyezettek megjelenítése (ha pontszámuk engedi)

Az FFT számításához az Apache Commons Math 3.4 API `FastFourierTransformer` osztályát használtam.

Fontosabb metódusok:

- `double[] linspace(double first, double last, int total)`:
  Egy lineárisan skálázott, `total` elemszámú vektort hoz létre két meghatározott szám között (`first` és `last`), zárt intervallumon. A MATLAB azonos nevű függvénye mintájára készítettem. Ennek segítségével határozom meg a spektrum mintáinak frekvenciáját. Az intervallum határolói ebben az esetben a 0 és a mintavételi frekvencia fele, azaz a Nyquist-frekvencia; a mintaszám pedig a spektrum mintaszámának fele.

```
private double[] linspace(double first, double last, int total) {
  double[] vec = new double[total];
  for (int i = 0; i < total; ++i) {
    vec[i] = first + i * (last - first) / (total - 1);
  }
  return vec;
}
```

- `double[] log2space(double first, double last, int total)`:
  A `linspace` függvény által létrehozott lineáris skálát az (1.1) képlet alapján logaritmizálja, ezzel centben kifejezve a hangmagasságokat.

- `void run()`:
  A fentiekben leírt műveletsort hajtja végre az akkordfelismeréshez, a korábbi fejezetekben ismertetett eljárások segítségével. Miután meghatározta az egyes akkordokhoz tartozó pontszámokat, az Activity `runOnUiThread()` metódusának segítségével íratja ki az akkordok neveit a képernyőre. A `runOnUiThread()` metódus - ahogy a neve is mutatja - visszaadja a futás jogát a GUI-t kezelő szálnak. Ez azért szükséges, mert az Android más szálat nem enged hozzáférni a GUI-hoz.

6.1.4. Akkordok adatainak tárolása

Az akkordtípusok szükséges adatait a `Chords` nevű enumerátor típus segítségével tárolom. A Java lehetővé teszi, hogy az `enum` típusok különböző értékeihez több adatot is eltároljunk

tagváltozókban, csakúgy mint egy osztály esetén. Sőt, a metódusok írása is megengedett. Ugyanakkor egy jellegzetes különbség az osztály és az `enum` között, hogy az `enum` konstruktora mindenképp `private` láthatóságú, hiszen a belőlük példányosított válozók csak előre meghatározott értékeket vehetnek fel.

A `Chords` kétféle lehetséges értéket tartalmaz, egyet a dúr, és egyet a moll akkordtípusokhoz. Minden típushoz három tulajdonságot tárolok el tagváltozókban:

- a bitmaszkot C alaphanggal
- az akkordtípus jelölését
- az akkordtípus nevét (későbbi felhasználás céljából)

Az értékek száma tetszőlegesen bővíthető más akkordtípusokkal, melyhez a program többi részét nem szükséges módosítani[3].

Fontosabb metódusok:

- `Double[] shift(Chords chordType, int distance)`:
  Ez a statikus metódus körkörösen lépteti az akkordípushoz tartozó bitmaszk elemeit `distance` lépéssel jobbra, ezzel ugyanazt az akkordot `distance` félhanggal magasabban megadva. Ennek a függvénynek köszönhetően nem kell minden akkordípushoz egyenként megadni mind a 12 bitmaszkot (12 különböző alaphangra), elég csak egyet, a 3.1 pontban leírtaknak megfelelően.
- `Double[] getInverse(Chords chordType, int shiftBy)`:
  Ez a statikus függvény a Stark és Plumbley [17] által használt mintaillesztés miatt szükséges, ez adja meg a `chordType` akkordtípus C-nél `shiftBy` félhanggal magasabb alaphangú bitmaszkjának komplemensét.

6.1.5. Konstansok

Az alkalmazásban használt konstansokat statikus változókként egy külön osztályba helyeztem. Ebben az osztályban van meghatározva például a mintavételezési frekvencia, a bitmélység, a buffer mérete bájtban és másodpercben, sőt, a legmélyebb hang frekvenciája is itt van megadva. Amennyiben módosítani kellene valamelyik adatot, így elég ezt egy helyen megtenni.

6.2. Hibalehetőségek az implementáció során

Az implementáció során beleütköztem néhány olyan hibába, melyek véleményem szerint könnyen elkövethetők, a fejlesztés menetét pedig igencsak megakaszthatják.

Az egyik ilyen hibába akkor futottam bele, amikor `ByteBuffer`-t használtam a hangminta tárolására. Ezek az objektumok lehetővé teszik a tárolt adatok bájt szintű kezelését.

[3] A felismerés pontosságát persze befolyásolja az itt megadott akkordok típusa és száma.

Több bájt méretű típusok tárolását is lehetővé teszik, kiolvasásnál azonban a fejlesztő felelőssége megadni, hogy hogyan szeretné értelmezni a pufferben lévő adatokat, azaz, hogy hány bájt tesz ki egy-egy értéket, és hogy milyen bájtsorrendben olvasandók ki a pufferből.

A `ByteBuffer` objektumok alapértelmezett bájtsorrendje csökkenő (big-endian), az `AudioRecord read()` függvényei azonban a készülék natív bájtsorrendjében írnak a pufferbe, ami a processzor típusától, gyártójától függ. Az általam használt teszteszköz esetén ez növekvő (little-endian) bájtsorrendet jelent. Az `AudioRecord` nem jelez hibát, és nem is figyelmeztet, ha az általa használt puffer bájtsorrendje eltér a natívtól.

Esetemben a hibakeresést nehezítette, hogy a puffer tartalmának helyességét úgy próbáltam ellenőrizni, hogy kiírattam azt egy bináris fájlba, majd a nyers adatfájlt WAV formátumra konvertálás után lejátszottam. A WAV alapértelmezett bájtsorrendje azonban szintén növekvő, így a kapott hangfájlok a helyes formátumban kerültek kiírásra.

A hiba a kiváltó ok felfedezése után könnyen javítható, a `ByteBuffer` objektumot a létrehozását követően natív bájtsorrendre kell állítani:

```
ByteBuffer byteBuffer = ByteBuffer.allocateDirect(size);
byteBuffer.order(ByteOrder.nativeOrder());
```

Egy másik hibalehetőség - melyet némileg könnyebb észrevenni - a bitmaszkok léptetésénél, a `Chords.shift(Chords chordType, int distance)` függvényben ütötte fel a fejét. A bitmaszkokat jelentő láncolt listákon a „shift” műveletet a `Collections.rotate()` függvényével végzem, mely a léptetést az eredeti objektumon hajtja végre, ezért mindenképp a lista „mély másolatát” (deep copy-ját) kell átadni neki, hogy az eredeti bitmaszk érintetlen maradjon.

```
List<Double> bitmaskCopy = new ArrayList<>(bitmask);
```

7. fejezet

Értékelés

Az Android-alkalmazás elkészültével ellenőriznem kellett a működésének helyességét. Először összehasonlítottam az azonos hangmintákra adott eredményeit a MATLAB-ban megírt programéval, majd gyakorlati alkalmazhatóság szempontjából összevetettem egy hasonló, ingyenesen letölthető alkalmazással.

7.1. A MATLAB-ban és az Androidon megvalósított eljárások összehasonlítása

A két implementáció összehasonlítását a következőképp végeztem: Az Android-alkalmazásnak klasszikus gitáron játszottam különböző akkordmeneteket, és kiírattam vele mind az audiópuffer tartalmát, mind az ablakra vonatkozó felismert akkordot fájlba. A nyers audiófájlokat ezután (tömörítés nélkül) WAV fájlokká konvertáltam, majd ezen futtattam a MATLAB-ban megvalósított eljárást. A két algoritmus így azonos a hangmintákon, azonos pozíciójú ablakokkal futott.

Az összehasonlítás során mindkét program esetén azonos paraméterekkel futtattam az eljárást. Az ablakok 14336 mintát tartalmaznak, mely 0,3 másodpercél kicsivel hosszabb. Ennek oka, hogy Android esetén a programnak másodpercben megadott ablakméretet felkerekítem a rendszer által meghatározott, minimális ablakméret legközelebbi többszörösére, az implemetáció egyszerűstése miatt. Mindkét eljárás akkor ír csak ki akkordnevet, ha nyertes akkord „pontszáma”, azaz a mintaillesztés során hozzárendelt érték nem haladja meg a 0.7-et. Ez a kritérium a gyakorlatban megfelelőnek bizonyult az olyan téves észlelések kiszűrésére, ahol az ablak nem tartalmaz zenei hangot.

A felvételekben a leggyorsabb akkordváltások megközelítőleg 100 bpm-es tempóban történtek, azaz 0,6 másodpercenként. A mintákban mind a 24 felismerni kívánt akkord szerepel. A tesztelés során mindösszesen 197 ablaknyi minta elemzését hasonlítottam össze.

Mindkét megvalósításról elmondható, hogy az összes lejátszott akkordot felismerték, azaz minden megszólaló akkordhoz tartozott legalább egy ablak, melyben a helyes, hozzá tartozó akkordnév szerepelt. Androidon az alkalmazás működésének megfelelően rögzítettem a második és a harmadik helyezett akkordot is, amennyiben a pontszámuk az előttük lévő helyezettől legfeljebb 10%-ban különbözött.

Összesen 33 ablak esetén tért el az első helyezett akkord a két eljárásban (16,8%), ezek közül

- 13 ablak esetén a MATLAB által észlelt akkord szerepelt az Android első három tippjében,
- és 13 ablak esetén az egyik eljárás nem észlelt akkordot, vagyis nem volt akkord amely elérte volna a szükséges pontszámot

Ezek alapján 7 olyan ablak volt, ahol az Android és a MATLAB teljesen más akkordot észlelt, ezek ez ablakok akkordváltást tartalmaztak. A két megvalósítás közti eltérés okát eddig nem sikerült felderítenem. A teszteredmények szerint ezekben az esetekben a MATLAB-os program gyorsabban felismerte az akkordváltást.

7.2. Összehasonlítás hasonló programokkal

A Google Play áruházban keresgélve három olyan akkordfelismerő alkalmazást találtam, melynek van ingyenesen letölthető verziója, ezek a következő neveket viselik:

- Chord detector
- Chord Detector ZAX
- Music Translator LITE

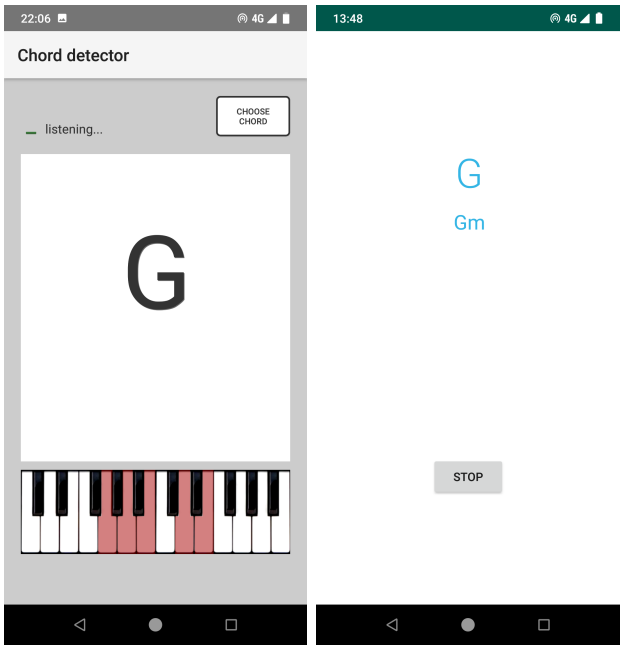
A *Chord Detector ZAX* jelenleg még alpha verziójú, az én appomhoz hasonlóan dúr és moll akkordok felismerésére törekszik, emellett rendelkezik pár extra funkcióval, mint például az alapvető tempó érzékelés, és a spektrum kirajzolása. Próbálkozásaim során azonban az akkordokat elvétve ismerte fel.

A *Music Translator LITE* egy hangrögzítő és zenelejegyző program, mely a hangfelvétel mellé kottát és akkordmenetet is rögzít. Az hangokat egyenként többnyire felismerte, akkordfelismerés terén azonban itt sem jártam sikerrel. Ez az alkalmazás a Google Play-en 3,8/5 csillagos értékeléssel rendelkezik, ezért lehetséges, hogy a kompatibilitással vannak problémák.

A teszteszközömön a három alkalmazás közül egyedül a *Chord detector* volt képes valódi akkordfelismerésre, így ezzel a programmal hasonlítom össze a sajátomat.

Ez az alkalmazás hármashangzatok felismerésére képes, a dúr és moll akkordok mellett szűkített, bővített és késleltetett akkordok is megtalálhatók benne. A több akkordtípus megkülönböztetése persze kifinomultabb akkordfelismerő eljárást igényel. Emellett a megszólaló hangokat egy zongora billentyűzeten is ábrázolja. Az én alkalmazásommal szemben ugyanakkor késletetési időben alulmaradt, egy-egy akkord felismerése olykor másodperces nagyságrendű késést okoz, míg az én programom esetén ez néhány tizedmásodpercet jelent. A késleltetett akkordok (*sus2* és *sus4*) felismerésében gyakran alaphangot téveszt.

Ezt a programot a 24 dúr és moll akkord lejátszásával teszteltem, melyeket - a 4. fejezetben végzett tesztekhez képest - hosszan, 1,5-2 másodpercig kitartottam. Ezekből az akkordokból 18-at sikerült elsőre eltalálnia (75%), újabb próbálkozással további hármat



7.1. ábra. *Képernyőkép a kipróbált Chord detector nevű alkalmazásról, és a saját alkalmazásomról*

felismert (87,5%). A magasabb fekvésben (9. bundon és afelett) lejátszott F♯-moll, G-moll és G♯-moll akkordokat a próbálkozásaim során egyáltalán nem ismerte fel, lehetséges, hogy ezek a frekvenciák már nincsenek benne az általa elemzett tartományban.

Ugyanezeket az akkordokat a saját alkalmazásom az akkord megszólalásától számított második ablakban mind felismerte, így a késleltetés legfeljebb az ablakméret kétszerese lehet (kb. 0.65 mp), a gyakorlatban az átlagos késleltetés azonban ennél kevesebb.

8. fejezet

Összefoglalás

Szakdolgozatom célja egy valós időben működő Android-alkalmazás elkészítése volt, mely képes felismerni a dúr és moll hármashangzatokat.

A dolgozat készítése során megismerkedtem az automatikus akkordfelismerés terén használt digitális jelfeldolgozási módszerekkel, melyeket kipróbáltam a gyakrolatban. Ezek közül a tesztek eredményei és a felhasznált irodalmi források megfontolásai alapján választottam ki azokat, melyek a legalkalmasabbnak bizonyultak a feladatra.

Áttekintettem az Android rendszer és a Java nyelv által kínált alkalmazásfejlesztési lehetőségeket, majd ezek, és a kiválasztott jelfeldolgozási módszerek felhasználásával megvalósítottam az alkalmazást.

Az elkészült program megbízható, kiegyensúlyozott teljesítményt nyújt a felismerés terén, késleltetése a gyakorlati alkalmazás során nem zavaró mértékű, használata pedig igen egyszerű. Az automatikus akkordfelismerés témaköre ugyanakkor számos lehetőséget tartogat, melyekkel a program tovább bővíthető.

8.1. Kitekintés, továbbfejlesztési lehetőségek

A bővítési lehetőségek közül talán a legkézenfekvőbb az akkordfelismerés kiterjesztése más akkordtípusokra; szűkített, bővített, késleltetett hármashangzatokra, üres akkordokra, négyeshangzatokra. Ehhez persze az eljárás további pontosítására, finomítására van szükség. Az imént felsorolt akkordokkal az általam használt források is foglalkoznak.

Egy viszonylag egyszerű bővítési lehetőség más hangolások támogatásának hozzáadása, melyek nem a 440 Hz-es kamarahangból számítják a többi hang frekvenciáját, hanem például a 432, vagy 446 Hz-ből. Ehhez csupán a spektrum centes skálázását kellene egy, a hangolás szerinti referencia-frekvenciaérték alapján számítani.

A kezdő gitárosok számára hasznos bővítés lehet a *kápó* (kapodaszter, capo) használatának figyelembevétele a programban. A kápó a gitár nyakára rögzíthető eszköz, mely a rögzítés helyén „lefogja” az összes húrt. Gyakran használt eszköz a transzponáláshoz, segítségével egy dal ugyanazokkal az akkordfogásokkal eljátszható tetszőleges számú félhanggal magasabban. Az akkordmenet jelölésében kápó használatakor gyakran azokat az akkordokat tüntetik fel, melyeket a gitárosnak le kell fognia, nem pedig azokat, melyek valóban

elhangzanak. A bővítés így annyiból állna, hogy az n pozícióban lévő kápó esetén az észlelt akkordokat n félhanggal mélyebb alaphanggal jelezné ki a program.

A mintaillesztés hiányosságait kiküszöbölve a chroma vektor azonostáshoz használható valamilyen statisztikai alapú megoldás, mint például a *rejtett Markov-modellek*.

A felismerés pontosságán javíthat, ha az alkalmazás az akkordok megszólalásának idejét az *onset detection* kutatási terület módszereivel érzékelné. Ezzel az ablakméret az akkordok időtartamához igazodva dinamikusan változhatna. A programot tempófelismeréssel kombinálva zenelejegyző („akkordmenet lejegyző") alkalmazás fejleszthető.

A felsorolt bővítési alternatívák megvalósításával olyan alkalmazás hozható létre, mely nemcsak a kezdő, de a haladó és profi zenészek számára is érdekes, hasznos információkat nyújthat.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani konzulensemnek, *Dr. Bank Balázsnak*, aki hasznos tanácsaival, alaposságával és készségességével nagyban segítette dolgozatom elkészítését, szakmai útmutatása révén pedig rengeteg új ismeretre tettem szert a digitális jelfeldolgozás témakörében.

Köszönettel tartozom az egész családomnak, különösen *édesanyámnak* aki szüntelenül támogatott egyetemi pályafutásom során.

Szeretném megköszönni a *barátaim* támogatását is, akik segítettek átlendülni a nehézségeken, és szebbé tették számomra az egyetemi éveket.

Irodalomjegyzék

[1] Szigetvári Andrea, Horváth Balázs. *Bevezetés a zenei informatikába*, page 142. Typotex Kiadó, Budapest, 2014.

[2] Judith C. Brown. Calculation of a constant Q spectral transform. *Journal of the Acoustical Society of America*, 89(1):425–434, January 1991.

[3] Android Developers. Activity. `https://developer.android.com/reference/android/app/Activity`, 2018. [Elérés ideje: 2018. december 04.].

[4] Android Developers. Audio Latency. `https://developer.android.com/ndk/guides/audio/audio-latency`, 2018. [Elérés ideje: 2018. december 04.].

[5] Android Developers. AudioRecord. `https://developer.android.com/reference/android/media/AudioRecord`, 2018. [Elérés ideje: 2018. december 04.].

[6] Android Developers. ByteBuffer. `https://developer.android.com/reference/java/nio/ByteBuffer`, 2018. [Elérés ideje: 2018. december 04.].

[7] Android Developers. Distribution dashboard. `https://developer.android.com/about/dashboards/`, 2018. [Elérés ideje: 2018. december 04.].

[8] Android Developers. MediaRecorder. `https://developer.android.com/guide/topics/media/mediarecorder`, 2018. [Elérés ideje: 2018. december 04.].

[9] Pituk Dávid. Hangoló megvalósítása Android platformon. Budapesti Műszaki és Gazdaságtudományi Egyetem, 2016. május.

[10] Takuya Fujishima. Realtime Chord Recognition of Musical Sound: a System Using Common Lisp Music. In *Proceedings of the 1999 International Computer Music Conference (ICMC 1999)*, pages 464–467, Beijing, China, October 1999.

[11] S. Piérard J. Osmalskyj, J-J. Embrechts and M. Van Droogenbroeck. Neural networks for musical chords recognition. In *Journées d'informatique musicale*, pages 39–46, Mons, Belgium, May 2012.

[12] Kyogu Lee. Automatic Chord Recognition From Audio Using Enhanced Pitch Class Profile. In *Proceedings of the 2006 International Computer Music Conference (ICMC 2006)*, pages 306–313, New Orleans, USA, November 2006.

[13] Dr. Kesztler Lőrinc. *Összhangzattan*, page 1. Editio Musica Budapest, Budapest, 1952.

[14] Ke Ma. *Automatic Chord Recognition*, 2016. URL: https://pdfs.semanticscholar.org/2a1f/46984aefde976ba99301847394311b0a3c17.pdf. [Elérés ideje: 2018. december 5.].

[15] Kurnia Muludi, Aristoteles, and Abe Frank SFB Loupatty. Chord Identification Using Pitch Class Profile Method With Fast Fourier Transform Feature Extraction. *IJCSI International Journal of Computer Science Issues*, 11(3):139–144, 2014.

[16] Steffen Pauws. Musical Key Extraction From Audio. In *Processing of 5th International Society for Music Information Retrieval*, pages 96–99, Barcelona, Spain, October 2004.

[17] Adam M. Stark, Mark D. Plumbley. Real-time Chord Recognition for Live Performance. In *Proceedings of the 2009 International Computer Music Conference (ICMC 2009)*, pages 85–88, Montreal, Canada, August 2009.

[18] Ekler Péter, Fehér Marcell, Forstner Bertalan, and Kelényi Imre. *Android-alapú szoftverfejlesztés*. SZAK Kiadó, Budapest, 2012.

[19] Budapesti Műszaki és Gazdaságtudományi Egyetem Mechatronika, Optika és Gépészeti Informatika Tanszék. Méréselmélet 12. fejezet - Simító ablakok (2018. november 22.). http://www.mogi.bme.hu/TAMOP/mereselmelet/ch12.html.

[20] Eric David Scheirer. *Music Perception Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, USA, October 1998.

[21] Alexander Sheh and Daniel P.W. Ellis. Chord Segmentation and Recognition using EM-Trained Hidden Markov Models. In *Processing of 4th International Society for Music Information Retrieval*, pages 183–189, Baltimore, Maryland, USA, October 2003.

[22] Tamara Smyth. *Music 270a: Signal Analysis*. University of California, San Diego (UCSD), November 2015. URL: http://musicweb.ucsd.edu/~trsmyth/analysis/.

[23] Superpowered. iOS and Android Audio Latency Test App. https://superpowered.com/latency. [Elérés ideje: 2018. december 04.].

[24] Superpowered. Android Audio's 10 Millisecond Problem: The Android Audio Path Latency Explainer. https://superpowered.com/androidaudiopathlatency, 2015. [Elérés ideje: 2018. december 04.].

[25] Wikipédia. Zenei hang – Wikipédia. https://hu.wikipedia.org/w/index.php?title=Zenei_hang&oldid=19164320, 2017. [Elérés ideje: 2018. november 25.].

[26] Wikipédia. Cent (zene) – Wikipédia. https://hu.wikipedia.org/w/index.php?title=Cent_(zene)&oldid=20340435, 2018. [Elérés ideje: 2018. novermber 25.].

[27] Wikipedia contributors. Mobile operating system — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Mobile_operating_system&oldid=871761037, 2018. [Elérés ideje: 2018. december 5.].

[28] Wikipedia contributors. Overtone — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Overtone&oldid=861113852, 2018. [Elérés ideje: 2018. november 25.].