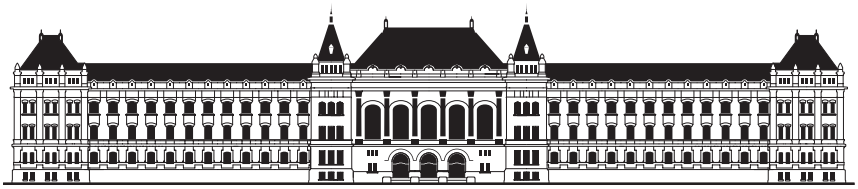
Diplomaterv

Galambos Róbert
2009



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

Time stretch és pitch shift algoritmusok megvalósítása

Diplomaterv

Galambos Róbert

Konzulens
dr. Sujbert László
egyetemi docens

2009

Kapott előlap ide jön

Ide jön a kapott előlap.

Time stretch és pitch shift algoritmusok megvalósítása

Az audio jelek digitális feldolgozása ma már elfogadott technika: nem csupán a hagyományos analóg feldolgozás alternatívája, hanem a korábbiaknál több lehetőséget nyújt. A felvételek utólagos korrigálásakor, vagy a műsorkészítés során gyakran változtatják a lejátszási sebességet vagy a hangmagasságot. A szűkebb szakterületen meghonosodott angol elnevezéssel time stretch és pitch shift algoritmusokat alkalmaznak.

A time stretch algoritmusok egy audio jel sebességét, időbeli lefutását befolyásolják (nyújtják, zsugorítják) úgy, hogy a jelek spektrális felépítése nem változik meg számottevően, azaz a gyorsított vagy lassított jel hangmagassága változatlan marad. A pitch shift algoritmusok ennek az ellenkezőjét teszik: az időbeli lefutás állandó marad, és a spektrális felépítés változik. Ez annyit jelent, hogy nem gyorsul és nem lassul a jel, de a hangmagassága megváltozik.

Az említett eljárások megvalósítására több módszer létezik, ezek tulajdonságai eltérők a jelek fajtája (periodikus és tranziens összetevők, beszéd- vagy zenei felvétel stb.), illetve a felhasznált algoritmusok bonyolultsága, számításigénye kapcsán. Különös jelentősége van azoknak az eljárásoknak, amelyek valós időben képesek a kívánt feldolgozást megvalósítani.

Az ismert algoritmusok között egy viszonylag új eljárás a wavelet-transzformáció alkalmazása. Ennek előnyei elsősorban a logaritmikus, éppen ezért a hangérzet szempontjából releváns frekvenciafelbontáshoz kapcsolódnak, de számítási nyereség is várható ezek alkalmazásától.

Fentiek alapján a diplomaterv keretében az alábbi konkrét feladatokat kell megoldani:

- Ismertessen és MATLAB szimulációk segítségével valósítson meg többféle time stretch és pitch shift algoritmust!
- Vizsgálja meg a wavelet-transzformáció alkalmazásának lehetőségét!
- Néhány – valós idejű megvalósításra alkalmas – algoritmust implementáljon ADDU-BF537-EZLITE jelfeldolgozó kártyán!
- Valós jelek, illetve felvételek segítségével végezzen vizsgálatokat az egyes algoritmusok alkalmazási körének meghatározására!

dr. Sujbert László
docens

Nyilatkozat

Alulírott, *Galambos Róbert*, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

.....
Galambos Róbert
hallgató

Tartalomjegyzék

Kivonat

A dolgozat témája a time stretch és pitch shift algoritmusok megvalósítása. A digitális audio jelfeldolgozás egy speciális ágát képezik azok az algoritmusok, eljárások, melyek segítségével egy mintavételezett és kvantált audio jelnek meg lehet úgy változtatni a hosszát – a sebességét –, hogy közben nem változnak meg a frekvenciakomponensei – hangmagassága. Ezeket nevezzük time stretch algoritmusoknak. Ha a jel hosszát – sebességét – változatlanul hagyva a frekvenciakomponenseit – hangmagasságát – módosítja az eljárás, akkor beszélünk pitch shift algoritmusról.

Diplomatervemben bemutatom, hogy ilyen jellegű jelfeldolgozásra gyakran van szükség, például stúdiótechnikában, ahol a rossz tempóban feljátszott sávokat lehet korrigálni time stretch segítségével, valamint a hamisan felvett, vagy felénekelt hanganyagot lehet hangokra igazítani pitch shift segítségével, vagy rádiós, illetve TV-s műsorszórásokban, egy-egy monológot felgyorsítva illeszteni az adásba.

Részletezem, milyen módszerek, algoritmusok léteznek a probléma megoldására, megemlítve az időtartománybeli szegmentálást, a fázis vokódolást, elemezve és részletezve működésüket, kitérve hibáikra. Megoldási lehetőségeket mutatok e hibák kiküszöbölésére, vagy csökkentésére. Felvetek egyéb lehetőségeket, melyek esetlegesen az algoritmusokban rejlenek, és más területeken is használhatóvá teszik őket, mint például a többsávos zajszűrés, a harmonizálás, vagy a diszharmonizálás.

Vizsgálataimat MATLAB-ban végzett szimulációkkal támasztom alá, melyek forráskódjai a mellékletben megtalálhatóak, valamint különböző hanganyagon végzett tesztekkel határozom meg az egyes alkalmazások hatáskörét. Ezenkívül bemutatom, hogyan implementáltam a lényegesebb algoritmusokat ADSP BF537 EZ-KIT Lite kártya segítségével, melynek forráskódja szintúgy megtalálható a mellékletben.

A jövőbeli terveim közé tartozik ezen algoritmusokat programbanként való implementálása, működésük finomítása, javítva használhatóságukat. Különböző jelfelismerő, és adaptív eljárásokkal kiegészítve egy univerzális szoftvereszközt létrehozni.

Abstract

My thesis work is about the implementation of time stretch and pitch shift algorithms. These algorithms belong to a special section of the field of digital audio signal processing, capable of modifying a sampled and quantized audio signal so that its length – speed – changes, but its frequency components – pitch – remains unchanged (time stretch algorithms), or the length — speed — remains unchanged and the frequency components – pitch -– are changed (pitch shift algorithms).

In my thesis I show that this type of signal processing is often needed. Time stretch can be used in a recording studio to correct tracks that were played with a wrong tempo, or correct false notes in an instrument play, or a vocal track. It is also capable of modifying the length of a monologue to fit in the playtime of a program on TV or radio broadcast.

I show in detail what type of methods, algorithms exist to solve this problem. I intorduce the time segmentation, the phase vocoder, analyzing how they work and what are their drawbacks, trying to solve or reduce the errors, and show some extra features hidden in the algorithms, for example multi-band noise cancellation, harmonizing or disharmonizing.

My investigation is based on MATLAB simulations. They can be found in the appendix with the audio tracks I have used to define the scope of each algorithm. I have implemented some of these on an ADSP BF537 EZ-KIT Lite DSP evaluation board. The source code can be found in the appendix too.

My plans for the future are to implement the algorithms as a software to refine and enhance usability. Furthermore, to integrate with signal recognizing and adaptive methods, to create an universal sotfwaretool.

Előszó

„Aki megkezdte – felét elvégezte a munkának."

Horatius

Feladatom többféle time stretch és pitch shift algoritmus elemzése és vizsgálata. Szimulációkat végeztem MATLAB-ban melyek bemutatják az egyes algoritmusok tulajdonságait, pontosabb képet kapva előnyeikről, és hátrányaikról. A wavelet-transzformációban rejlő lehetőségeket vizsgálva bemutatom, hogy az idő-frekvencia sík más jellegű felosztása milyen előnyöket rejt. A vizsgált algoritmusok közül néhányat DSP kártyán is megvalósítottam, melyeket szintén kifejtek a dolgozatomban. Mindezen eredményeket mintahanganyaggal teszem még szemléletesebbé, mely megtalálható a CD mellékleten.

Az 1. fejezetben kitérek a time stretch algoritmusoknak történelmi hátterére, és végigvezetem röviden, hogy különböző időszakokban milyen formában, és megoldással jelent meg ez a problémakör. Elérve napjainkig röviden felvázolom a használt szoftvereket, és bemutatom milyen nehézségek, és beállítási gondok jelentkeznek bennük. Ezután az algoritmusok minősítésére adok lehetőségeket, különböző szemszögből vizsgálva a jelet. Ezenkívül megemlítem a statisztika alapú minősítés lehetőségét is.

A 2. fejezetben a time stretch-re használható algoritmusokat veszem sorra. Többek között az időtartománybeli szegmentáció, a fázis vokódolás, és wavelet-transzformáció alapú algoritmus részletezésével. Bemutatom előnyeiket, hibáikat, felhasználási lehetőségeiket, és kitekintést nyújtok, hogy milyen más problémák megoldásai integrálhatóak az algoritmusba. Kitérek az egyes algoritmusok különböző speciális fajtáira, mint például a PSOLA, WSOLA, vagy egyes fázis vokóder módosításokra.

A 3. fejezetben rátérek a pitch shift algoritmusokra, kifejtem pontosan mit is takar ez a fogalom, részletezem hogyan származtathatóak time stretch algoritmusokból. Bemutatom, hogy használható néhány time stretch algoritmus direktben pitch shiftelésre, és kitekintést nyújtok más jelekre alkalmazott más kritériumhalmazt kielégítő pitch shift algoritmusokra, és létjogosultságukra, felhasználásukra.

Az 5. fejezetben az eredményeimről írok, melyeket úgy kaptam, hogy az algoritmusokat futtattam a hanganyagokon különböző beállítások mellett, és utána az 1. fejezetben említett összehasonlító módszerekkel vizsgáltam őket. A mérési eredményeket táblázatos formában közlöm, mellékelve a magyarázatot, a várt és a tapasztalt eredmények közti lehetséges eltérésre.

A 4. fejezetben az algoritmusok implementációi kerülnek leírásra, részletesen kitérve az alkalmazott hardver felépítésére, és tulajdonságaira, a szoftveres keretrend-

szer magyarázatára – mely a kártya beállítását és felkonfigurálást végzi –, valamint magukra az algoritmusokra. Kifejtem milyen problémák lépnek fel az egyes algoritmusok megvalósításnál, és ezen hibáknak mik az okai.

A 6. fejezetben összefoglalom munkám, és egy gyors áttekintést nyújtok az egész műről, melynek segítségével egy kerek egészet képez majd. Ezenkívül kitekintek az esetlegese fejlesztési lehetőségekre, és a jövőbeli terveimre.

Szeretnék köszönetet mondani azoknak embereknek akik segítették munkámat, és egyengették utamat. Dr. Sujbert Lászlónak, a konzulensi munkájáért, Balló Gábornak a LaTeX-es és MATLAB-os segítségekért, Dr. Gaál Józsefnek, dr. Elek Kálmánnak, és dr. Pfliegel Péternek, valamint a MIT-es DSP labornak szakmai segítségükért.

1. fejezet

Bevezetés

„A zene ott kezdődik, ahol a szó hatalma véget ér."
Claude Achille Debussy

A hallás az emberi érzékelés egyik legkifinomultabb formája, és a környezetünkről szerzett információink egyik legnagyobb forrása. Mi sem bizonyítja ezt jobban, mint, hogy képesek vagyunk a fülünkkel hangforrásokat lokalizálni a térben. Agyunk a visszhangok apró időkülönbségiből, és a környezet szűrőhatásaiból rekonstruálja a teret, és benne a hangforrást. Különböző hangokat meg tudunk különböztetni, és zajos környezetben is képesek vagyunk szelektíven egy adott hangra figyelni.

Mivel ilyen finom érzékszerv, így rengeteg inger érheti. A kellemesebbek közé tartozik a zene, mely egyidős az emberiséggel, és képes rengeteg állapotot, hangulatot, mondandót kifejezni. Ennek megfelelő komplexitás jellemzi, mind időtartományban, ahol ütemnek hívjuk a valamilyen szabály szerinti lüktetést, mind pedig frekvenciatartományban, ahol harmóniáknak az egyes hangmagasságok egymás közti viszonyát.

1.1. Sebességállítás a zenében

Ha a zenét gyorsítva, vagy lassítva szeretnénk meghallgatni, máris elérkeztünk az első alapproblémához. Változtassuk meg a hanganyag sebességét úgy, hogy más paramétere ne változzon meg.

A múltban, amikor még nem volt se digitális, se analóg hangrögzítés, ennek az egyetlen módja az volt, ha újra eljátszották a művészek a zenét. Ilyen esetben természetesen nem beszélhetünk szó szoros értelemben sebességállításról, hiszen a művészek minden egyes alkalommal újraalkották a zenét.

Később megjelentek az analóg hanghordozók, mint például a hanglemez, vagy a magnószalag. Elindult útján – többek között – a stúdiótechnika, és a rádiós műsorszórás. Ekkor merült fel legelőször a sebességállítás problémája, amikor a már rögzített hanganyagokat kellett volna utólag más sebességgel lejátszani. Az indokok különbözőek voltak. A stúdiótechnikában a rosszul felvett sávok ütemeit próbálták korrigálni, a rádióban értékes műsoridőt megspórolni. Ezt először analóg módon valósították meg. A lejátszók motorjának fordulatszámát változtatták, így lassítva a

lemez forgását, vagy a magnószalag haladását. A szórakoztatóiparban a lemezlovasok ezt a fajta sebességállítást még ma is használják, a zenék egymás után keverésére.

Ennek a módszernek problémája, hogy a sebességállítás közben a hanganyag frekvenciakomponensei is megváltoznak. Ha gyorsítunk a jelen, akkor a harmonikusok frekvenciái ennek megfelelően, egy egynél nagyobb számmal szorzódnak, míg lassítva egy egynél kisebbel. Ez látható a Fourier-transzformáció hasonlósági tételéből is.

$$\begin{array}{ccc} f(t) & \xrightarrow{t:=t/\alpha} & f(t/\alpha) \\ {\scriptstyle\mathcal{F}}\big\downarrow & & \big\downarrow{\scriptstyle\mathcal{F}} \\ F(\omega) & \longrightarrow & |\alpha|\, F(\alpha\omega) \end{array} \tag{1.1}$$

ahol $F(\omega) = \mathcal{F}\{f(t)\}$ és $|\alpha|\, F(\alpha\omega) = \mathcal{F}\{f(t/\alpha)\}$. ebből kifolyólag csak korlátozottan használható. Ha a sebességállítás következtében fél hanggal elcsúsznak a felharmonikusok, akkor az egy zeneműben már elfogadhatatlan, hiszen többé nem lesz összhangban magával a darabbal. Ha ezt számszerűsíteni akarjuk, akkor egy félhang távolság kifejezhető a következőképpen:

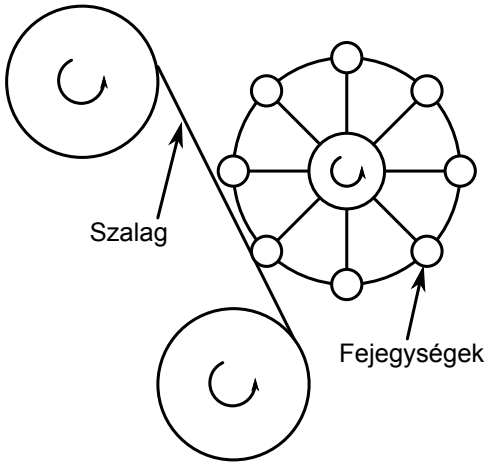
$$\begin{aligned} f_{\text{up}} &= \sqrt[12]{2} f \approx 1.05946 \cdot f \\ f_{\text{down}} &= \frac{1}{\sqrt[12]{2}} f \approx 0.94387 \cdot f \end{aligned} \tag{1.2}$$

amelyből az következik, hogy $\pm 5\%$-nál nagyobb állítás nem lehetséges anélkül, hogy elrontsa a harmóniákat. Persze ez csak közelítés, hiszen az abszolút hallással rendelkező emberek, vagy a vájt fülüek ennél jóval kisebb eltérést is már diszharmonikusnak, és élvezhetetlennek hallanak.

Az 1960-as években Pierre Schaeffer előállt egy analóg megoldással, melyet Phonogene-nek hívott. Õ egy szalagos lejátszót módosított úgy, hogy egy fej helyett többet épített bele, és ezek a fejek egy forgórészen voltak, és adott sebességgel forogtak [1]. Így ha a szalag sebességét állították, akkor a hosszát tudták változtatni a jelnek, míg a forgás sebességét állítva a hangmagasságok változtak. Ez látható az 1.1. ábrán.

A digitális technika megjelenésével a problémát új szemszögből lehetett megközelíteni, és a jelfeldolgozás adta lehetőségek, a DSP-k és számítógépek számítási kapacitása, valamint a Fourier-transzformáció segítségével, a jeleket a frekvencia- és időtartományban más, új módszerekkel lehetett befolyásolni.

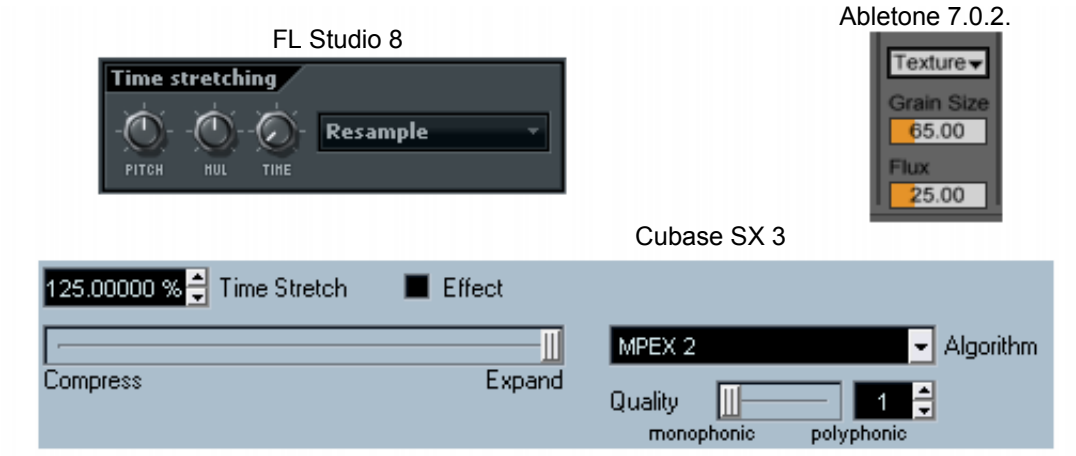
Így alakultak ki a time stretch és pitch shift algoritmus csoportok. Time stretch algoritmusoknak nevezzük azokat az algoritmusokat, melyek megváltoztatják egy digitális jel hosszát, tehát a sebességét, de változatlanul hagyják a frekvenciakomponenseit, vagyis a hangmagasságait. Ezzel ellentétben a pitch shift algoritmusok a jel sebességét, hosszát hagyják változatlanul, és a frekvenciakomponenseit, hangmagasságait változtatják meg.



1.1. ábra. A Phonogene működése

1.2. Napjaink szoftverei

Napjainkban több olyan szoftver van, mely képes time stretch és pitch shift algoritmusokra. A probléma általában a beállíthatóságukkal van. A legtöbb szoftverben csak néhány paraméter állítható, így a felhasználó nem tudja finomhangolni az adott jelhez az algoritmust. Erre láthatunk példát az 1.2. ábrán, ahol néhány szoftver felhasználófelületén figyelhető meg e szűk mozgástér.



1.2. ábra. Time stretch és pitch shift napjaink szoftvereiben

Többek között ez is biztosan oka, hogy az algoritmusok minősége is hagy kívánnivalókat maga után. Persze vannak a piacon elfogadható minőséget produkáló algoritmusok is, de ezeket általában a legnagyobb ipari titok övezi, és működésükről semmilyen információt nem hagynak kiszivárogni.

1.3. Algoritmusok minősítése

A minőséget definiálni és összehasonlítani nehéz feladat, hiszen az eredeti és a time stretchelt jel más sebességen van, míg az eredeti és a pitch shiftelt jel más hangmagasságon van. Így az időtartományban összehasonlítani és eltérési hibát számolni lehetetlen. Ezt még tetézi az, hogy a fül érzékelése szubjektív, és nehéz olyan objektív mértéket találni, amelynek segítségével eldönthetjük két jelről, hogy közülük melyiket halljuk jobb minőségűnek. A problémát még bonyolítja, hogy ahány ember annyiféle ízlés, így lehet, hogy ami az egyiknek jobban tetszik, az a másiknak kevésbé.

Mivel a szakirodalomban ilyen jellegű vizsgáló módszereket nem találtam, így kísérletet tettem olyan összehasonlító algoritmusok konstruálásra, melyek segítségével a szubjektív hallással is nagyrészt egyező módon hasonlítható egymáshoz az algoritmusok működése.

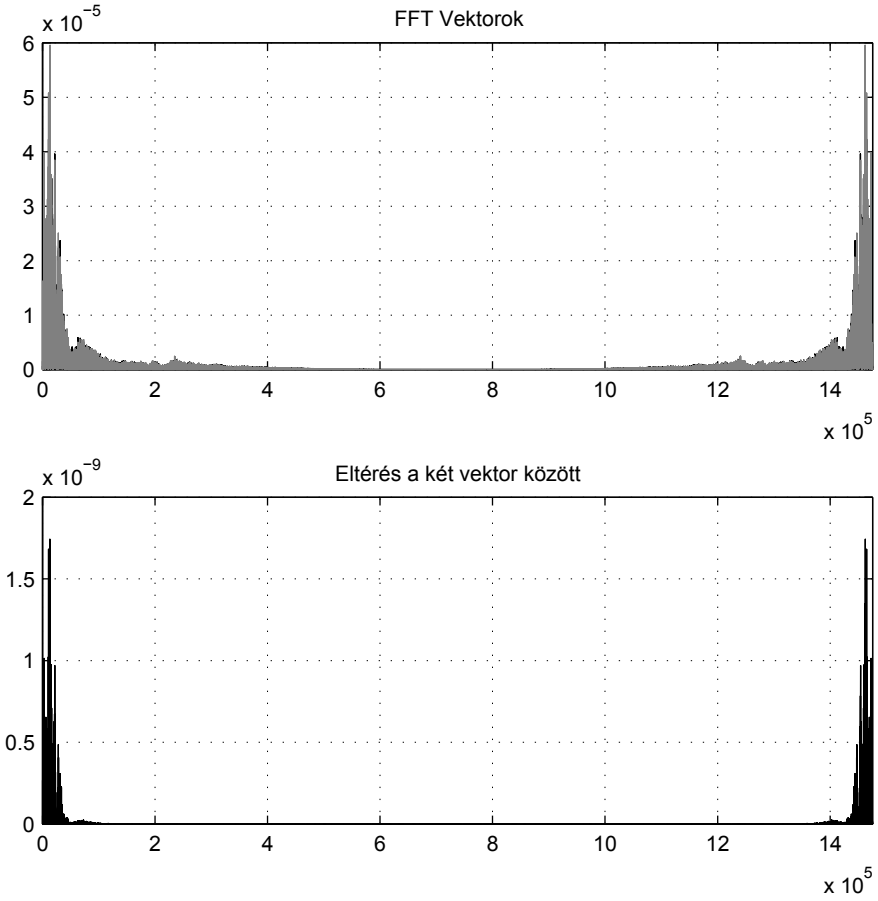
1.3.1. Frekvenciakomponensek összehasonlítása

Az első eljárás az eredeti és a sebességállított jel frekvenciakomponenseit hasonlítja össze. Ezt úgy teszi, hogy az eredeti és a feldolgozott jelek közül az időben rövidebb végét kipótolja nullákkal, ezzel ugyanolyan hosszúvá téve a két jelet. Ezután mind a két jelet transzformálja, ezzel megkapva a két jel frekvenciatartománybeli képét. A két jel viszont nem azonos energiával rendelkezik, így, bár a transzformált párok hasonlítanak egymásra, a skálázás eltérése miatt nem lehet hibát számolni. A két transzformáltat normálni kell. Először az amplitúdóspektrum maximumához normáltam őket, de ez nem vezetett eredményre, ugyanis a sebességállítás esetlegesen egy frekvenciakomponenst nagyon kiemel, ami, bár zavaró a hallgató számára, de nem annyira, mint azt a hiba kiszámításával kapjuk, ugyanis a kiugró komponens lesz az amplitúdókarakterisztika maximuma, és ehhez normálódik az egész transzformált. Így a frekvenciatengely minden pontján nagy eltérést, nagy hibát mutat a két transzformált, holott a hiba csak a kiugró komponens lenne. Ezért esett a választás az amplitúdó alatti területtel történő normálásra. Ez látható az 1.3. ábrán is. Az eljárás képlettel felírva:

$$e_{\text{freq}} = \frac{1}{N}\sum_{j=1}^{N}\left(\frac{|\mathcal{FFT}\left\{\hat{x}[n]\right\}|}{\sum_{i=1}^{N}|\mathcal{FFT}\left\{\hat{x}[n]\right\}|} - \frac{|\mathcal{FFT}\left\{\hat{x}_{\text{proc}}[n]\right\}|}{\sum_{i=1}^{N}|\mathcal{FFT}\left\{\hat{x}_{\text{proc}}[n]\right\}|}\right)^2 \tag{1.3}$$

ahol $x[n]$ és $x_{\text{proc}}[n]$ az eredeti és a feldolgozott jelet jelentik. A kalap jelölés arra utal, hogy nullákkal lett kipótolva a rövidebbik vége, hogy $x[n]$ és $x_{\text{proc}}[n]$ ugyanolyan hosszú legyen, és ezáltal az $\mathcal{FFT}$ ugyanolyan hosszú vektort adjon vissza. N az $\hat{x}[n]$ és az $\hat{x}_{\text{proc}}[n]$ vektorok hosszát jelöli. A képlet elején az N-nel történő leosztás függetlenné teszi a mérést az $\mathcal{FFT}$ méretétől

Az eljárás használható pitch shift algoritmusok minősítésére is. Ekkor a transzformáltakat interpolálni kell, hogy a megfelelő frekvenciakomponensek fedésbe kerül-



1.3. ábra. Frekvenciakomponensek összehasonlítása

jenek, és a normálás után hasonló módon lehet őket összehasonlítani. Mivel azonban a pitch shift algoritmusok általában a time stretch algoritmusokból származtathatóak, mint azt majd látni fogjuk a 3. fejezetből, így erre az eljárásra ritkábban van szükség.

A frekvenciakomponensek összehasonlításának az a legnagyobb hibája, hogy az időbeli felbontást elveszítjük, és ha az algoritmusok a frekvenciatartományban helyes eredményt is adnak, ez még nem jelenti azt, hogy az időtartományban is. Erre a következő eljárás ad megoldást.

1.3.2. Spectrogramok összehasonlítása

Mivel az is fontos, hogy melyik időpillanatban, hogyan változnak a frekvenciakomponsek értékei, így egy másik hibaszámítást kell alkalmazni. Vegyük az eredeti és feldolgozott jel spektrogramját (1.4), ami a short-time Fourier transzformáció abszolútértékének négyzetével közelíthető, akkor megkapjuk a jel időben változó frekvenciafelbontását. Az eredeti és a feldolgozott jel nem ugyanolyan hosszú, így az időben rövidebb jelnek a spektrogramját interpoláljuk az időtengely mentén úgy, hogy azonos számú egyenletes távolságú osztás keletkezzen. Így a két spektrogramban az összetartozó tranziens események „fedésbe” kerülnek, és az időben rövidebb

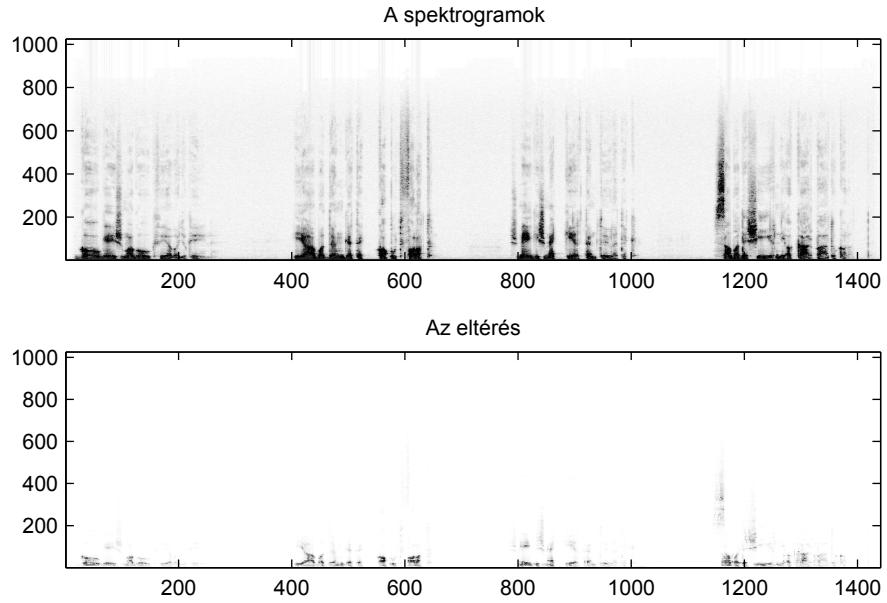
jel spektrogramját hozzá „nyújtottuk” az időben hosszabbéhoz.

$$\begin{aligned} \mathbf{S} &= \mathcal{STFT}\left\{x[n]\right\}^2 \\ \mathbf{S}_{\text{proc}} &= \mathcal{STFT}\left\{x_{\text{proc}}[n]\right\}^2 \end{aligned} \tag{1.4}$$

A $\hat{\mathbf{S}}$-t és a $\hat{\mathbf{S}}_{\text{proc}}$-t ismét súlyozva összegezzük, majd négyzetes hibát számolunk a kettő különbségéből. A kalap jelölés itt is arra utal, hogy a két spektrogram egymásra lett illesztve.

$$e_{\text{spec}} = \frac{1}{NM}\sum_{k=1}^{N}\sum_{l=1}^{M}\left(\frac{\hat{\mathbf{S}}}{\sum_{i=1}^{N}\sum_{j=1}^{M}\hat{\mathbf{S}}_{ij}} - \frac{\hat{\mathbf{S}}_{\text{proc}}}{\sum_{i=1}^{N}\sum_{j=1}^{M}\hat{\mathbf{S}}_{\text{proc}ij}}\right)^2 \tag{1.5}$$

ahol N és M az $\hat{\mathbf{S}}$ és $\hat{\mathbf{S}}_{\text{proc}}$ mátrix méreteit jelölik. Az NM-es leosztás azért kell, hogy a függetlenné tegyük az $\mathcal{STFT}$ méretétől. Az így kapott mérési eljárás adott tulajdonságok szerint hasonlítja össze a két jelet. A szimulációk során ez az eljárás is akkor adott nagyobb hibát, ha hallásra is rosszabb volt az eredmény.



1.4. ábra. Spectrogramok összehasonlítása

Ez is használható pitch shiftelt jelekre, csak a két spektrogram egymásra „nyújtása” nem az idő-, hanem a frekvenciatengely mentén történik.

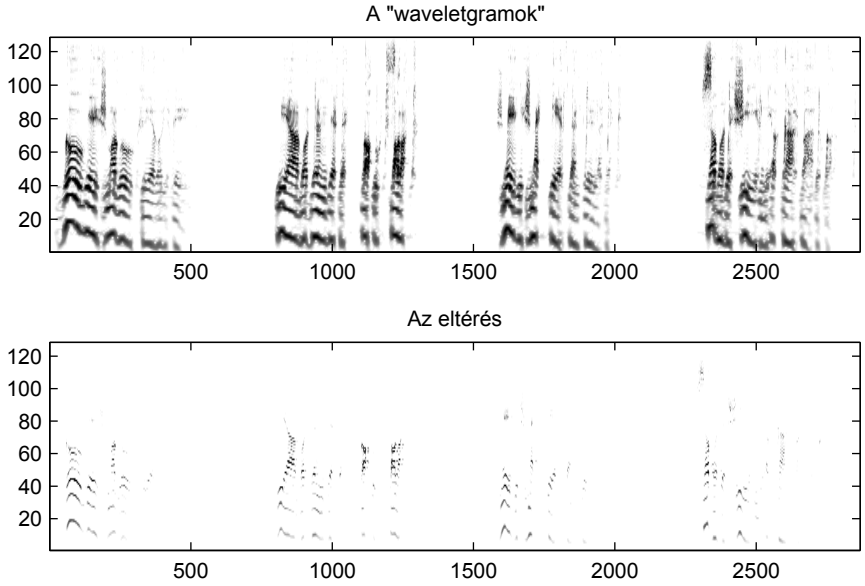
1.3.3. Többfelbontásos analízisek összehasonlítása

Mivel az $\mathcal{STFT}$ idő- és frekvenciafelbontása egyenletes rácsot alkot az idő- és frekvenciasíkon [2][3], és nem követi a zene logaritmikus, oktávos frekvenciafelbontását, ezért nem optimális ilyen jellegű analízisre. Sokkal alkalmasabb lenne egy olyan

felbontás, mely az alacsony frekvenciás hangokat nagyobb, míg a magas frekvenciás hangokat kisebb részletességgel elemzi a frekvenciatengely mentén. A probléma másik aspektusa, hogy a magas frekvenciás jelek rövidebb tranziens komponenseket tartalmaznak, így nagyobb időbeli lokalizáltság szükséges helyeik pontos meghatározásra, ez pedig csak úgy lehetséges a Heisenberg-tétel szerint, ha a frekvenciafelbontás csökken.

Erre alkalmas a többfelbontásos analízis, mely ezt a különböző felbontást biztosítja, az idő- és frekvenciasíkot különböző téglalapokra osztva. A téglalapok minimális területét a Heisenberg-tétel határozza meg. A többfelbontásos analízis egy speciális esete a wavelet-transzformáció is [3].

Az összehasonlítási eljárás lényegében ugyanaz, mint az 1.3.2. szakaszban látható. Az eltérés egyedül abban van, hogy nem spektrogrammal dolgozunk, hanem a többfelbontásos analízissel kapott „képpel”.



1.5. ábra. Többfelbontásos analízisek összehasonlítása

1.3.4. További lehetőségek

Lehetőség lenne a feldolgozott hangokat egy hallgatóságnak bemutatni, és az általuk egy vaktesztben rangsorolt mintákról statisztikát készítve, felállítani egy költségfüggvényt, hasonlóan az MP3 kódolás pszichoakusztikai modelljéhez. Ez figyelembe venné az emberi hallás különleges tulajdonságait. Hátránya lehetne, hogy nem az egyénhez, hanem a statisztikai átlaghoz viszonyítana, ami egy személyre nézve nem biztos, hogy optimális. Ilyen jellegű teszteket nem állt módomban folytatni, de mint lehetőséget említésre méltónak találtam.

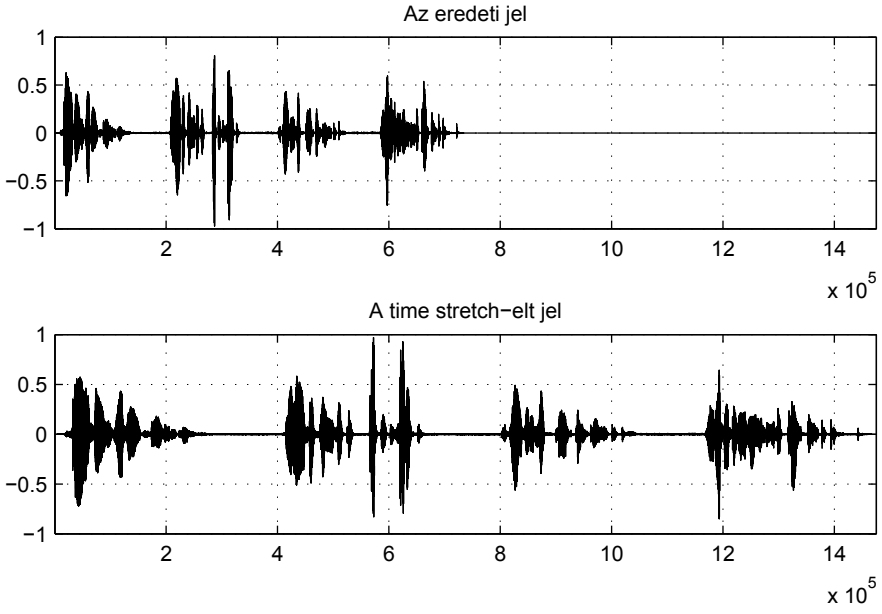
2. fejezet

Time stretch

„Az időnek egyetlen oka van: minden nem történhet egyszerre.”

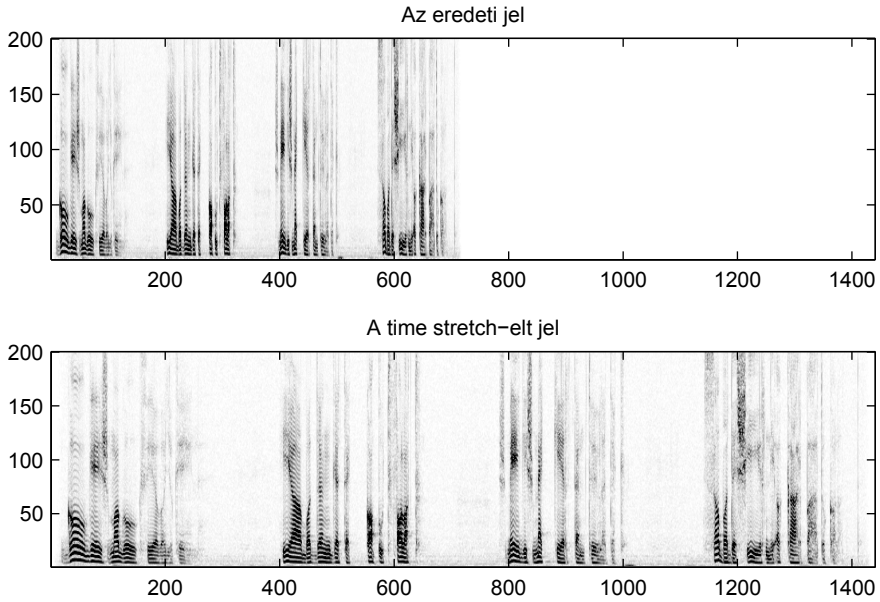
Albert Einstein

Time stretch az összefoglaló neve azon sebességállító algoritmusoknak, melyek képesek a jel sebességét úgy megváltoztatni, hogy közben a hangmagasságok változatlanok maradnak. Ez azt jelenti, hogy miközben gyorsabb, vagy lassabb lesz a jel, a frekvenciakomponensei nem fognak megváltozni, eltolódni.



2.1. ábra. A time stretch az időtartományban

Ahogy az a 2.1. ábrán látszik, a time stretch algoritmus az időtartományban megváltoztatta a jel hosszát. Viszont ha megnézzük a 2.2. ábrát, akkor a két spektrogramon láthatjuk, hogy a frekvenciakomponensek nem változtak.



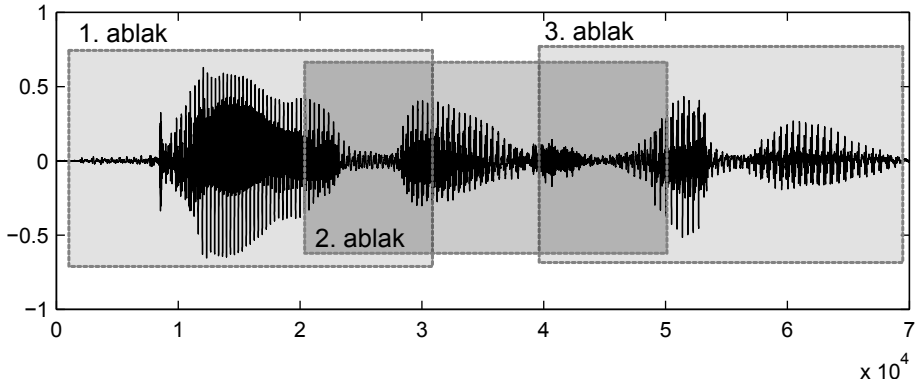
2.2. ábra. A time stretch a frekvenciatartományban

2.1. Időtartománybeli szegmentálás

Az első ilyen algoritmus az időtartománybeli szegmentálás, vagy angol nevén overlap and add, mely talán az egyik legegyszerűbb algoritmus az összes közül. Működési elve Pierre Schaeffer Phonogene-nek a digitális változata, és az algoritmus ötlete is valószínűleg innen ered.

2.1.1. Algoritmus működése

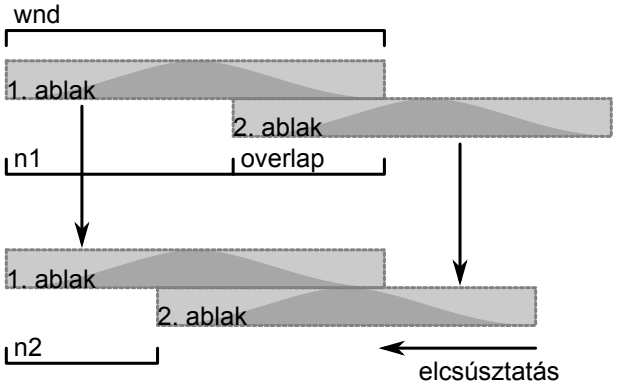
A Phonogene-szerű működésből következik, hogy a jelet időtartományban átlapolt ablakokra kell bontani, majd az ablakokat szét- illetve összecsúsztatni, és azokat újra összegezni. Ezt a technikát hívják „overlap and add”-nak (OLA).



2.3. ábra. Az időtartománybeli ablakokra bontás

A 2.3. ábrán látható, hogy a mintavételezett jelet az időtartományban hogy bontjuk ablakokra, majd csúsztatjuk össze, illetve szét. Az algoritmus több paraméterrel

rendelkezik, ezek láthatóak a 2.4. ábrán, ahol n_1 az eredeti jel lépésköze időtartományban, n_2 a feldolgozott jel lépésköze időtartományban, *wnd* az ablak mérete (később l_{wnd}),és *overlap* az átlapolódó részek hossza (később l_{overlap}).



2.4. ábra. Az algoritmus paraméterei

Az algoritmus egyenleteit is felírhatjuk. Ha a jel $x[n]$, az ablakfüggvény $w[n]$, és n_1 távolság van ablakok kezdete között, akkor az ablakok által kivágott részek felírhatók a következőképpen:

$$\begin{aligned} y_0 &= x[n] \cdot w[n] \\ y_1 &= x[n] \cdot w[n - n_1] \\ y_2 &= x[n] \cdot w[n - 2n_1] \\ ... &= ... \\ y_k &= x[n] \cdot w[n - kn_1] \end{aligned} \tag{2.1}$$

ahol y_k a k-adik kivágott ablakot jelenti. Ezeket az ablakokat kell most elmozgatnunk, méghozzá úgy, hogy az eddigi n_1 távolság helyett n_2 távolság legyen az egyes ablakok között.

$$\begin{aligned} y_k[n + (n_1 - n_2)k] &= x[n + (n_1 - n_2)k] \cdot w[n - kn_1 + (n_1 - n_2)k] \\ &= x[n + (n_1 - n_2)k] \cdot w[n - n_2k] \end{aligned} \tag{2.2}$$

Ezután ezeket a kivágott és elcsúsztatott ablakokat összegezni kell, hogy megkapjuk a kimeneti time stretchelt jelet.

$$\begin{aligned} z[n] &= \sum_{k=-\infty}^{\infty} y_k[n + (n_1 - n_2)k] \\ &= \sum_{k=-\infty}^{\infty} x[n + (n_1 - n_2)k] \cdot w[n - n_2k] \end{aligned} \tag{2.3}$$

ahol $z[n]$ a kimeneti jel. Mivel jelfolyamról beszélünk, így nincs eleje és vége, ezért a k index $\in [-\infty, \infty]$ tartománynak. A valóságban természetesen véges hosszú jelekre alkalmazzuk a képletet. Ilyenkor a jelek kezdeténél és végénél a $w[n]$ ablakfüggvénnyel

való szorzás miatt a jelalak torzul, egy „felhangosodó” rész keletkezik az elején, és egy „elhalkuló” rész a végén. Ha arra ügyelünk, hogy ne legyen ezeken a helyeken információtartalom, amit úgy érünk el, hogy a jel elé és mögé nullákat szúrunk be, akkor ezt a torzulást kiküszöbölhetjük.

Az ablakfüggvények illeszkedése nem feltétlen tökéletes, ezért a hangerő-ingadozás kiküszöbölése végett a $z[n]$ jelet normálni kell. A legegyszerűbb megoldás erre, ha a jel mellett számolunk egy $z^{'}[n]$-t, ami az ablakfüggvények összege.

$$z^{'}[n] = \sum_{k=-\infty}^{\infty} w[n - n_2 k] \tag{2.4}$$

Ha ezzel a $z^{'}[n]$-el leosztjuk az eredeti $z[n]$ kimeneti jelünket, akkor a jelszint-ingadozásokat korrigáltuk.

$$\begin{aligned} z[n] &= \frac{\sum_{k=-\infty}^{\infty} x[n + (n_1 - n_2)k] \cdot w[n - n_2 k]}{z^{'}[n]} \\ &= \frac{\sum_{k=-\infty}^{\infty} x[n + (n_1 - n_2)k] \cdot w[n - n_2 k]}{\sum_{k=-\infty}^{\infty} w[n - n_2 k]} \end{aligned} \tag{2.5}$$

2.1.2. Tempóingadozás

A 2.1.1. alszakaszban leírt algoritmus az ablakok kezdőpontját a sebességállításnak megfelelően áthelyezi, ellenben az ablakok időtartama alatt a hanganyagot még az eredeti sebesség jellemzi, hiszen pont ezért nem változnak a hangmagasságok a sebességállítás közben. Ebből az következik, hogy bár az ablakok elején a tempó a sebességállított jel tempója, az ablak végére ettől el fog térni. Az eltérés függ a sebességállítás mértékétől, és az ablak méretétől.

$$e_{\text{tempo}} = (1 - r_{\text{speed}}) \cdot l_{\text{wnd}} \tag{2.6}$$

ahol $r_{\text{speed}} = n_2/n_1$, az eredeti és a feldolgozott jelek sebességaránya, l_{wnd} az ablak hosszát, és e_{tempo} egyenlő a mintákban megadott tempóingadozás felső becslőjével.

A tempóingadozás olyan alkalmazásokban kritikus, ahol jeleket szinkronizálni kell valamihez. Ilyen felhasználási terület lehet például egy DJ keverőprogram, vagy CD lejátszó. Ha ilyen szemszögből vizsgáljuk a hibát, akkor át tudjuk számolni BPM (beat per minute) ingadozásba.

Egy négy negyedes ütemű zenét vizsgálva, a taktus és dobok egybeesnek. Ez általában igaz a mai elektronikus zenék nagy részére. Ilyenkor legrosszabb esetben két dob között e_{tempo} mintányi hiba lép fel. Így a BMP ingadozás a következőképpen számolható:

$$\Delta BPM = \frac{60 \cdot f_\mathrm{s}}{\frac{60 \cdot f_\mathrm{s}}{BPM} + e_\mathrm{tempo}} - BPM \tag{2.7}$$

A (2.7) összefüggést használva, ha $f_\mathrm{s} = 44100$ Hz, $BPM = 130$, $l_\mathrm{wnd} = 2048$ minta, és $r_\mathrm{speed} = 1.2$, akkor $e_\mathrm{tempo} = 409.6$, és $\Delta BPM = -2.56$, azaz viszonylag nagy lassítás hatására jelentős eltérés lehet egy rövid időre a BPMben. A becslésünket pontosíthatjuk, hisz az l_{wnd} nem minden esetben az ablak hosszával egyenlő. Ha két ablak átlapolódik, akkor csak az átlapolódásmentes részt kell nézni a pillanatnyi, és a következő ablak között. Ez alapján felírható:

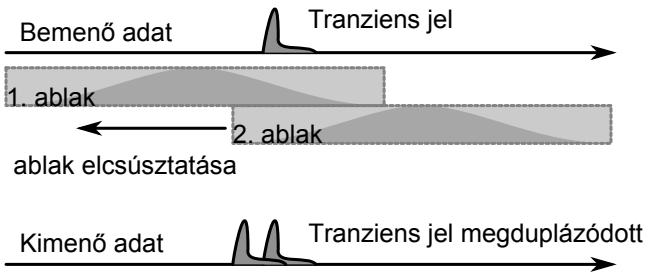
$$e_\mathrm{tempo} = (1 - r_\mathrm{speed}) \cdot n_1 \tag{2.8}$$

ahol r_speed az eredeti és a feldolgozott jelek sebességarányát, n_1 az ablak hosszát az átlapolódó rész nélkül, és e_tempo a mintákban megadott tempóingadozás felső becslőjét jelöli.

A kiszámoltakat figyelembe véve kedvezőbb ha minél rövidebb az átlapolódásmentes része az ablakoknak. Ezt vagy úgy érhetnénk el, hogy az ablakméretet csökkentjük, vagy úgy, hogy az átlapolódást növeljük. Természetesen egyik lehetőség sem megoldás, mert, mint látni fogjuk, mindkét esetben további hibák jelennek meg, melyek között kompromisszumot kell kötnünk.

2.1.3. Átlapolódó ablakok

Ha egy ablak átlapolódik a következővel, akkor az átlapolódott részen az információ duplán lesz jelen, hisz mindkét ablakban megtalálható, és amikor a sebességállítás miatt az ablakokat elcsúsztatjuk, a két ablak azonos információjú része nem lesz fedésben. Így azon a részen ismétlődés tapasztalható. Ez akkor jelentkezik a legjelentősebben, ha a jelfolyam tranziens része belefér az átlapolódásba, és ott nem tekinthető stacionáriusnak. Ezt szemlélteti a 2.5. ábra.



2.5. ábra. Információ ismétlődés

Ha az ismétlődés elkerülése érdekében úgy választanánk meg az ablakokat, hogy ne legyen átlapolódás, akkor kénytelenek lennénk olyan $w[n]$ ablakfüggvényt használni, amely, mivel nem kellően „sima”, lényegesen megváltoztatná a jel spektrális szerkezetét. Valamint, ha nem lapolódnának át az ablakok, akkor nem lehetne lassítani, hisz az ablakokat már jobban nem lehetne „széthúzni”.

2.1.4. Fésűszűrő hatás

Az átlapolódó ablakok okozta tranziensjel-kétszereződés is felfogható fésűszűrőnek, hiszen a jel időben eltolt önmagával adódik össze, de míg a 2.1.3. alszakaszban a tranziens jelek részéről vizsgáltuk a problémát, most a frekvenciatartomány felől közelítjük meg.

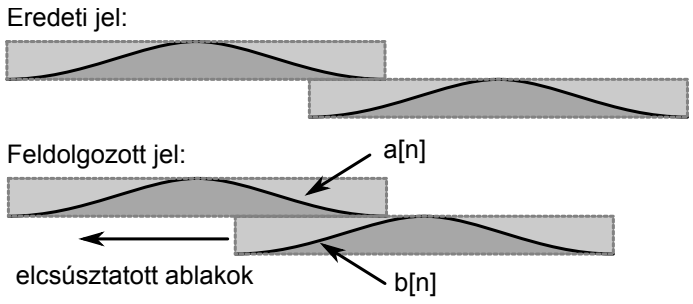
Ha n_1-et úgy választjuk meg, hogy az ablakok között legyen átlapolódás, és n_2-t úgy, hogy gyorsuljon a jel, akkor, mivel $n_2 < n_1$, $n_1 - n_2$ minta hosszú rész a feldolgozott jelen átlapolódást szenved, és egy időfüggő fésűszűrő alakul ki, melynek paraméterei az egymást követő ablakfüggvényektől függenek. Ha n_1-et úgy választjuk meg, hogy ne legyen átlapolódás, a fent említett hatás akkor is jelentkezik. Ha feltételezzük, hogy egyszerre csak két ablak lapolódik át, akkor a következőképpen írható fel a fésűszűrő egyenlete:

$$y[n] = a[n] \cdot x[n] + b[n] \cdot x[n + (n_1 - n_2)] \tag{2.9}$$

ahol $a[n]$ a pillanatnyi ablakfüggvény azon a része, mely az átlapolódásnál van, és $b[n]$ a következő ablakfüggvény átlapolódásnál lévő része. Hasonló egyenlet felírható több ablak átlapolódására is, csak ekkor nagyobb fokszámú lesz a szűrő, és a szűrési hatás is jobban jelentkezik:

$$y[n] = \sum_{i=0}^{N-1} a_i[n] \cdot x[n + i(n_1 - n_2)] \tag{2.10}$$

ahol N az átlapolt ablakok száma, és $a_i[n]$ az i-edik ablakfüggvény n helyen vett értéke. A 2.6. ábrán látható az átlapolódó ablakok egymásra hatása, és a fésűszűrő kialakulása.

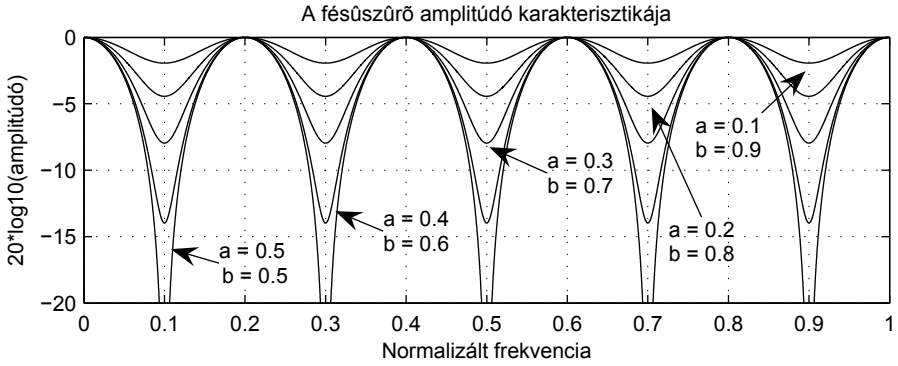


2.6. ábra. A fésűszűrő kialakulása

Ha a (2.9) egyenletben feltételezzük, hogy $a[n]$ és $b[n]$ lassan változó paraméterek, akkor a fésűszűrő felírható a Z-transzformáltjával, ahol $a[n]$ és $b[n]$ időben változó paraméterek:

$$
\begin{aligned}
Y(z) &= a[n] \cdot X(z) + b[n] \cdot X(z) z^{n_1 - n_2} \\
\frac{Y(z)}{X(z)} &= a[n] + b[n] \cdot z^{n_1 - n_2} \\
\frac{Y(\omega)}{X(\omega)} &= a[n] + b[n] \cdot e^{j\omega(n_1 - n_2)} \\
\left| \frac{Y(\omega)}{X(\omega)} \right| &= \sqrt{b[n]^2 + 2a[n]b[n]\cos((n_2 - n_1)\omega) + a[n]^2}
\end{aligned}
\tag{2.11}
$$

ahol X a bemeneti jel, és Y a sebességállítás utáni jel transzformáltja. A 2.7. ábrán látható különböző $a[n]$ és $b[n]$ értékek mellett a szűrő amplitúdókarakterisztikája. Ha megfigyeljük a (2.11) képletet, észrevehetjük, hogy a leszívások száma $n_2 - n_1$-től függ. Feltételezve, hogy $a[n] + b[n] = 1$ a hangerőnormálás miatt, kijelenthetjük, hogy akkor a legerősebbek a leszívások, ha $a[n]$ és $b[n]$ közel azonos értékűek.



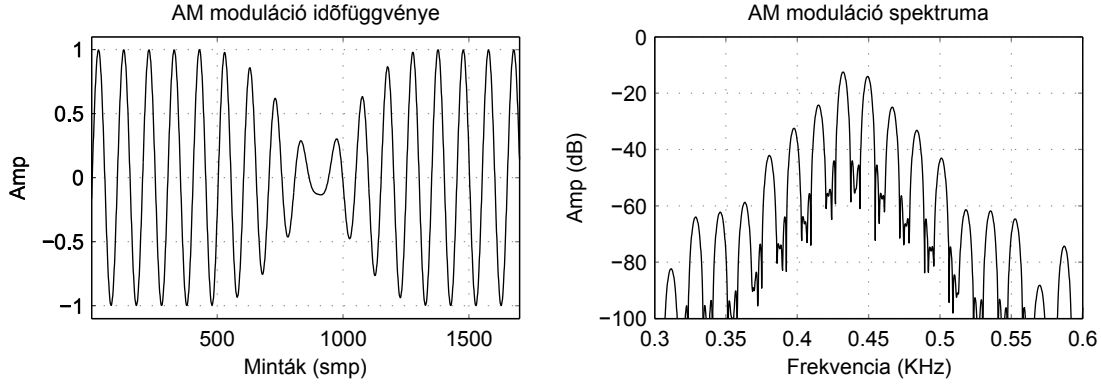
2.7. ábra. A fésűszűrő amplitúdókarakterisztikája

Ha ehhez figyelembe vesszük a 2.6. ábrát, akkor megfigyelhető, hogy $a[n]$ értéke csökken, míg $b[n]$ értéke folyamatosan nő az átmenet folyamán, így a leszívások nagysága először nő, majd elérve $a[n] = b[n]$-nél a maximális leszívást újra csökkenni kezd.

2.1.5. AM moduláció

Ha a 2.1.4. alszakaszban tárgyalt fésűszűrőt most időben vizsgáljuk, akkor láthatjuk, hogy az átlapolódó szakaszon kívül nem érvényesül a hatása. Ha egy megfelelő frekvenciájú szinuszjelet dolgozunk fel az algoritmusokkal, akkor azt tapasztaljuk, hogy változatlan marad az ablakok nem átlapolódó részénél, viszont az átlapolódásoknál a fésűszűrő leszívásai miatt csökkenhet a hangereje. Így egy frekvenciaszelektív modulációt kapunk, mely a leszívások környéki frekvenciákon a jelet amplitúdóban modulálja, megváltoztatva a frekvenciakomponenseit. Az OLA alapalgoritmusnál a synchronized overlap and add (SOLA) módosított algoritmus jobban teljesít, mert a SOLA algoritmusban az átlapolódó részek $a[n]$ és $b[n]$ paramétereit, tehát az aktuális ablak „elhalkulását”, és a következő ablak „felhangosodását”, pontosabban ezek helyét, mindig úgy választjuk meg, hogy az átlapolt részen megkeressük

a legnagyobb keresztkorrelációs értéket, és eszerint állítjuk be a két paramétert. Ez javít valamelyest a problémán, de teljesen nem szüntetni meg.



2.8. ábra. AM moduláció hatása

A 2.8. ábrán egy ablakátmenet látható, és a jobb oldalon a hozzá tartozó amplitúdóspektrum. A tesztjel egy 440 Hz-es szinuszjel, az algoritmus 4096 mintás Hanning ablakkal dolgozott, $n_1 = 2048$ és $n_2 = 3686$ paraméterek mellett egy 1.8-szoros lassítást végrehajtva.

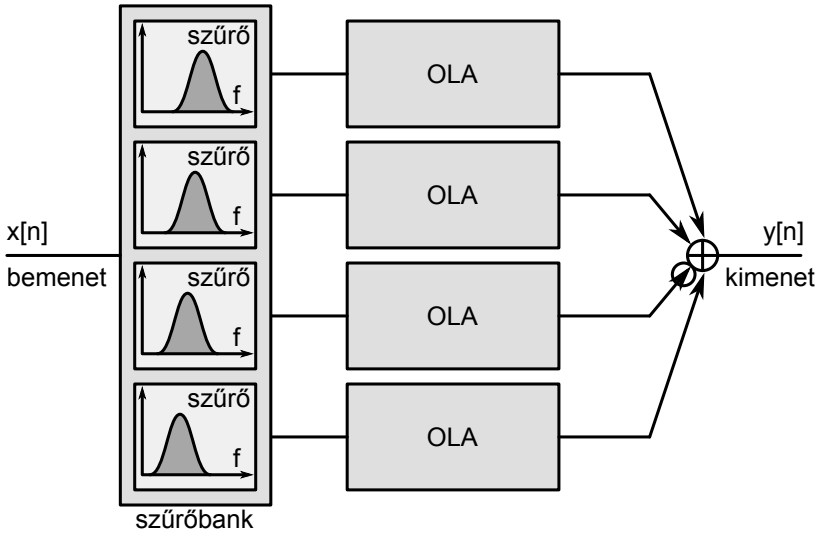
Az amplitúdómoduláció frekvenciája attól függ, mekkora ablakokat használunk. Ha nagyobbak az ablakok, akkor a moduláció frekvenciája is kisebb, ezáltal a spektrum torzulása is. Ezenkívül számít, hogy hány ablak lapolódik át, hisz ilyenkor a fésűszűrő hatás is megváltozik.

2.1.6. Többsávos szegmentálás

Mivel a jelfolyam magas és alacsony frekvenciás komponensei más sebességgel változnak, így feldolgozhatjuk a frekvenciasávokat külön-külön, sávra optimalizált paramétereket használva. Erre egy megoldás, ha a bejövő jelet egy szűrőbank segítségével szétbontjuk, majd ezeken a frekvenciasávokon hajtjuk végre a sávnak megfelelő paraméterekkel az OLA algoritmust, és végül a sávokat ismét egyesítjük.

A felbontás, feldolgozás, és összegzés a blokkvázlata a 2.9. ábrán látható. Ha összehasonlítjuk a sávokat, a magas frekvenciasávban rövidebb ideig tartanak a tranziens jelek, így ott kisebb ablakokat alkalmazhatunk, míg a mély frekvenciasávban nagyobb ablakokat kell használnunk, hogy az AM-moduláció ne jelentkezzen annyira intenzíven. Az eljárás hátránya, hogy a tempóingadozás minden sávban máshogy mutatkozik, és több frekvenciasávot átölelő jelet időben „delokalizál", „szétken" , mivel minden sávban más pillanatnyi értékkel tér el a kívánt tempótól.

Többsávos feldolgozásként kipróbáltam egy 3 sávra bontást FIR szűrőkkel, valamint a diszkrét wavelet-transzformáció „oktávszűrőbankját" [4][5]. Itt 8-16 sávra is felbontottam a jelet. Az eredmények a két feldolgozás között körülbelül hasonlóak voltak, a feldolgozott jel minősége némileg javult. A wavelet-es szűrőbank esetében a feldolgozás gyorsabb volt.



2.9. ábra. Több sávos szegmentálás

2.1.7. Összefoglaló

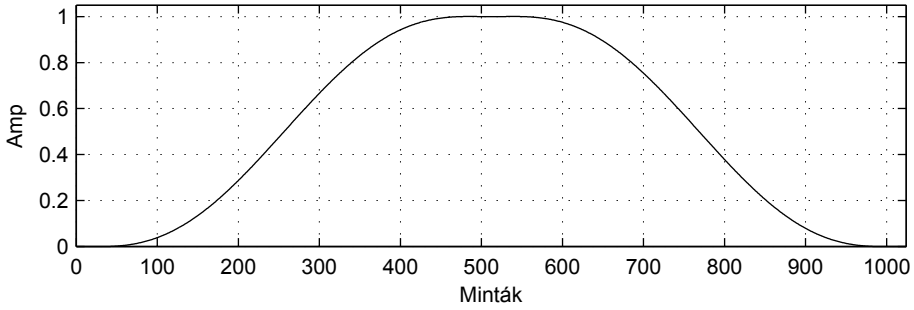
Az ablak paramétereinek megválasztásánál kompromisszumokra kényszerülünk, hiszen az egyes hatások teljes megszüntetése nem lehetséges. Az ablak méretét úgy kell megválasztani, hogy az AM moduláció ne legyen zavaró a jelre nézve, tehát az ablak hosszát elég nagyra kell választani, de túl nagy sem lehet, hisz akkor a tempó kezd el ingadozni. Az átlapolódó tartománynak elég kicsinek kell lenni, hogy a fésűszűrés időtartamát, valamint az információduplázódást csökkentsük, de túl kicsi sem lehet, mert akkor az ablakfüggvény lesz túl éles változású, és fogja befolyásolni a jelfolyamot.

A tipikusan megválasztott ablakméretek magas frekvenciasávokra $1000 - 2000$ minta körül, mély frekvenciasávokra $5000 - 10000$ minta körül mozognak $f_s = 44100$ Hz mintavételi frekvencia mellett. Ablakfüggvényként általában a Hanning ablakot használtam, de kipróbáltam egy ötödfokú polinom ablakot is, ahol az ablak deriváltja 25% és 75%-ánál nem 1 volt, mint a Hanning ablaknál, hanem 2, csökkentve ezzel a fésűszűrő hatást olyan módon, hogy $a[n]$ és $b[n]$ paraméterek még rövidebb ideig tartózkodnak 0.5 körül. Ez látható a 2.10. ábrán is.

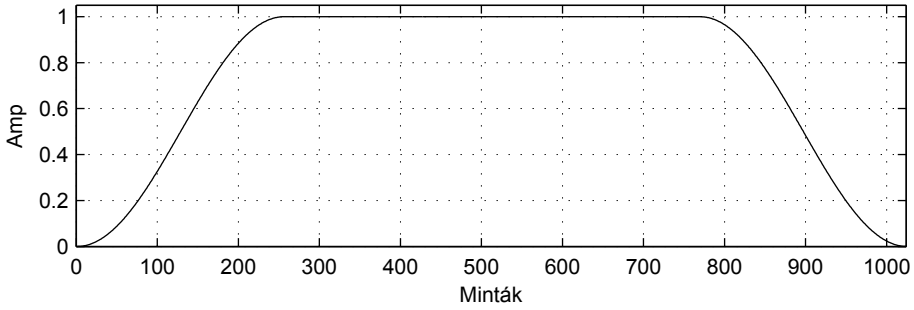
Ezen kívül Hanning szélű ablakokat is használtam, kis átlapolódású ablakok vizsgálatakor. Hanning szélű ablakok alatt olyan ablakokat értek, melyek egy középen kettévágott Hann ablakból állnak, melynek közepét kipótoltam egyesekkel. Ezt szemlélteti a 2.11. ábra.

2.1.8. Speciális esetek

A SOLA algoritmus gyorsítása a EM-TSM (Envelop-Matching for Time Scale Modification), mely a kereszt korreláció kiszámolását kerüli el, és helyette a frameek amplitúdó-burkológörbéit, pontosabban annak előjeleit használja, ezzel gyorsítva



2.10. ábra. Ötödfokú polinom ablak



2.11. ábra. Módosított Hanning ablak

az algoritmust, és csökkentve az igényelt számítási kapacitást [6].

A beszédfeldolgozásban használják a SOLA olyan változatát, mely a sebességállítási faktort változtatja a jelnek megfelelően úgy, hogy ahol tranziens részek jönnek, ott az eredeti tempóra állítja vissza ezt a faktort, ezzel az emberi fül számára érthetőbbé teszi a feldolgozott jelet [7].

Ha jelről van plusz információnk, akkor javíthatjuk a szegmentálás minőségét, és más kritériumokat is figyelembe vehetünk. Ilyen plusz információ, ha például a feldolgozott jel monofónikus. Ekkor alkalmazható a PSOLA, azaz pitch synchronous overlap-add technika [1][8]. Ennél a technikánál az ablakméret szinkronizálva van az alapharmonikus periódusával, és az „overlap-add" elvégzése után mind a jel hosszát, mind a frekvenciakomponenseit meg lehet változtatni. Ezen felül még mint plusz kritérium az algoritmus megtartja a jel formánsait – ezért is használják beszédjelre –, sőt képes ezeket megváltoztatni is. Ezzel ugyanolyan hangmagasságú, de más karakterisztikájú hangot lehet létrehozni. (Például egy női hangot férfivé alakítani.) Emiatt használják real-time vokál korrekcióra, és többszólamú kóruseffektus létrehozására [8].

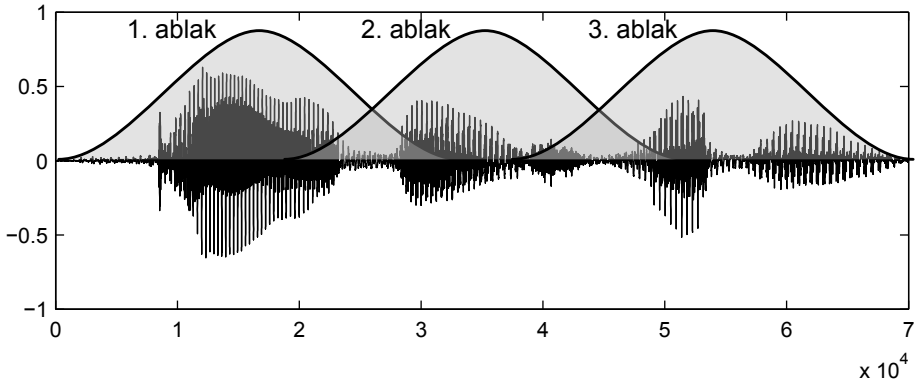
A WSOLA is nagyon hasonló a PSOLA-hoz, csak itt a nem az alapharmonikusra szinkronizálódik az algoritmus, hanem a maximális lokális hasonlóság alapján határozza meg, hová kerülnek az ablakok [9].

2.2. Fázis vokódolás

A következő algoritmus time stretch megvalósítására a fázis vokódolás [1]. A módszer a jelet nem az idő-, hanem a frekvenciatartományban változtatja meg.

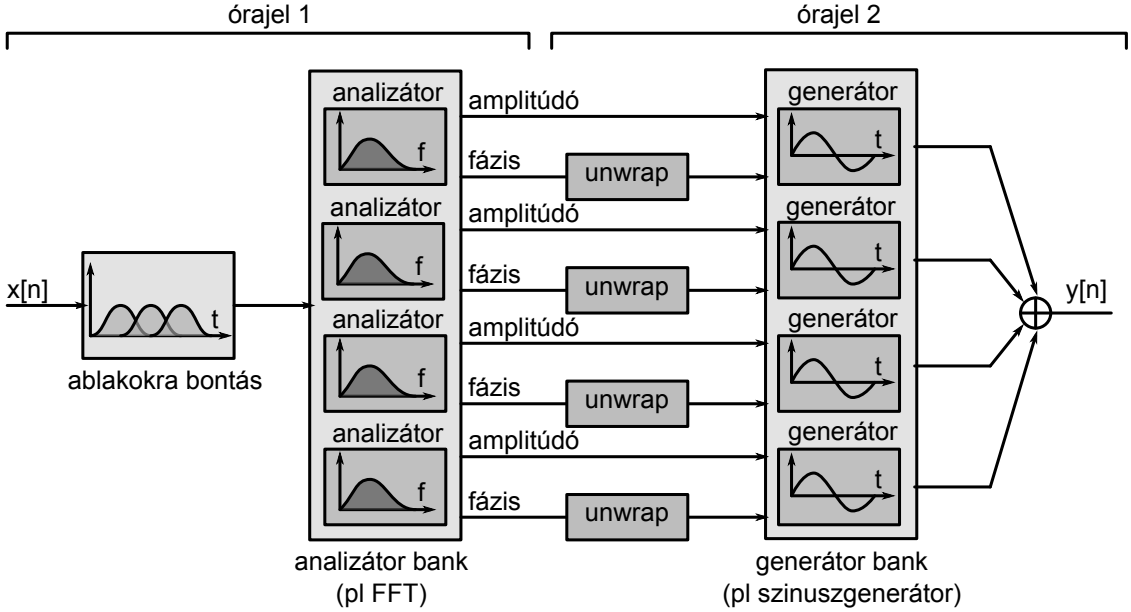
2.2.1. Algoritmus működése

Az algoritmus lényege, hogy $\mathcal{STFT}$ (Short-time Fourier Transform) segítésével a jel időbeli szegmenseiből kinyerje a különböző frekvenciákhoz tartozó fázis- és amplitúdóértékeket, majd ezen információkkal szintetizálja újra a jelet, például szinuszos generátorbank használatával. Az analízis és a szintézis más órajelre történik, és így el tudjuk nyújtani, vagy össze tudjuk nyomni a jelfolyamot, tehát meg tudjuk változtatni a sebességét.



2.12. ábra. A jelfolyam ablakokra bontása

A 2.12. ábrán látható a jelfolyam ablakokra bontása. Az ablakfüggvényt úgy kell megválasztani, hogy ne torzítsa el a jel frekvenciakomponenseit, ne okozzon nagy „spektrális szivárgást". Ehhez használható például Hanning ablak, vagy más $\mathcal{STFT}$-hez használt ablak. Az átlapolódást ugyanúgy változtattam, mint az időszegmens alapú feldolgozásnál, és általában 75-85%-os átlapolódást használtam, a fázisváltozás pontosabb követéséhez.



2.13. ábra. Analízis és szintézis

A 2.13. ábrán az analízis és szintézis blokkdiagramja látható. Az analízis rész felírható az $\mathcal{STFT}$ képlete segítségével:

$$\mathbf{G}_n = \mathcal{FFT}\left\{x[n] \cdot w[n - n_1 k]\right\} \tag{2.12}$$

ahol $\mathbf{G}_n$ az n-edik ablakhoz tartozó $\mathcal{STFT}$ vektor. Ebből a vektorból ki lehet számolni a generátorokhoz szükséges amplitúdó- és fázisinformációkat.

$$\begin{aligned} \mathbf{A}_n &= |G_n| \\ \mathbf{\Phi}_n &= \text{angle}(G_n) \end{aligned} \tag{2.13}$$

ahol $\mathbf{A}_n$ az amplitúdóvektor, $\mathbf{\Phi}_n$ pedig a fázisvektor, és az angle(.) függvény a komplex szám fázisát adja vissza. Mivel az analízis és a szintézis más órajelen működik, így az egymás utáni amplitúdóértékek között interpolálni kell, hogy pontosabb értéket kapjunk.

2.2.2. Amplitúdóinterpoláció

Az amplitúdó interpolálása egyszerű, hiszen csak az aktuálisan kiszámolt és a régi amplitúdók között kell valamilyen módon, például lineárisan interpolálni. Ezt szemlélteti a (2.14) egyenlet is.

$$\mathbf{A}_k = \frac{\mathbf{A}_{\text{now}} - \mathbf{A}_{\text{old}}}{n_1} \cdot k \tag{2.14}$$

ahol n_1 a két amplitúdóérték, $\mathbf{A}_{\text{now}}$ és $\mathbf{A}_{\text{old}}$, közötti mintaszám, tehát az időbeli lépés, és $k \in [0; n_1)$. Az így kapott $\mathbf{A}_k$ vektor amplitúdóértékei lesznek a generátorok amplitúdóértékei a k. mintánál.

2.2.3. Fázisinterpoláció

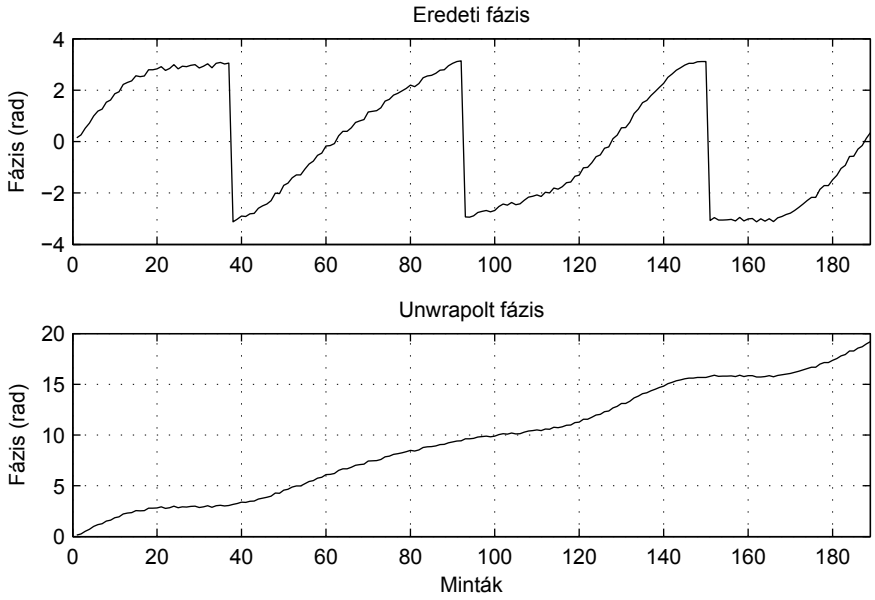
A fázis interpoláció komplikáltabb, hiszen figyelembe kell venni, hogy a szintézisnek eltérő órajele van az analízishez képest, emiatt a szintézis csak a fázisváltozást tudja reprodukálni, de az abszolút fázist nem képes követni. Ezért egy fázisinkoherencia lép fel a frekvenciasávok között, ami azt jelenti, hogy bár sávonként a fázisváltozást követni tudja az algoritmus, a sávok egymáshoz képest abszolút fázisban eltérnek, ha sebességállítást szenved a jel.

Ezenkívül még figyelni kell arra, hogy a fázis értékeit átlapolódáshelyesen kell kezelni az összeadásoknál és a kivonásoknál. Ez azt jelenti, hogy a fázist „unwrap”-elni kell, hogy biztosítsuk a helyes eredményeket. A 2.14. ábrán látható egy fázis-időfüggvény, eredeti és „unwrap”-elt változatban.

Két ablak között felírható a fázisra a következő egyenlet:

$$\phi_{\text{now}} = \text{wrap}\left(\phi_{\text{old}} + \phi_{\text{wnd}}(\omega) + \phi_{\text{diff}}\right) \tag{2.15}$$

ahol ϕ_{now} a mostani fázis, ϕ_{old} az előző fázis, ϕ_{wnd} az ablak hossza alatti fázisforgás, mely természetesen frekvenciafüggő. Ekkor ϕ_{diff} a fázisváltozás a két időpont között. A wrap(.) függvény pedig fázis „wrap”-elését oldja meg a következőképpen:



2.14. ábra. Fázis unwrap

$$\text{wrap}(x) \coloneqq x \mod 2\pi \tag{2.16}$$

Az implementációtól függ, hogy a wrap(.) függvény $[0; 2\pi)$ vagy $[-\pi; \pi)$ tartományba lapolja be a fázist. Ezekből (2.16) és (2.15) segítségével kifejezhetjük ϕ_{diff}-et a következőképpen:

$$\phi_{\text{diff}} = \text{wrap}\left(\phi_{\text{new}} - \phi_{\text{old}} - \phi_{\text{wnd}}\right) \tag{2.17}$$

Az így kiszámolt érték a fázis eltérése a normál állapottól, tehát azt mutatja meg, hogy az analizátor adott sávjának frekvenciájától mennyivel tér el az ablak hossza alatt. Ez könnyedén átszámítható frekvenciaoffsetté:

$$f_{\text{prec}} = f_{\text{bin}} + \frac{\phi_{\text{diff}}}{n_1 \cdot 2\pi} f_{\text{s}} \tag{2.18}$$

ahol f_{bin} az $\mathcal{STFT}$ megadott elemének frekvenciája, n_1 az $\mathcal{STFT}$ időtartománybeli lépése mintában, és f_{s} a mintavételi frekvencia. Ezzel a módszerrel pontosítható a jel $\mathcal{STFT}$ által mért frekvenciája, hiszen nem csak az amplitúdócsúcsot olvassuk le, hanem a fázisból kinyert plusz információ segítségével sokkal pontosab számításokra vagyunk képesek. Számszerűsítve ez azt jelenti, hogy 64 pontos $\mathcal{STFT}$-vel, $f_{\text{s}} = 44100$ Hz mintavételi frekvencia mellett, 0.1 Hz pontosan tudunk frekvenciát mérni, az $f_{\text{s}}/64 \approx 689$ Hz-es pontosság helyett. Természetesen, ha a mérendő jel frekvenciája pontosan két „bin” között van, akkor megtörténhet, hogy egy „bin”-nyi hiba lép fel, de ennek a kiküszöbölése lehetséges.

Visszatérve a sebességállításhoz, a kiszámolt ϕ_{diff} segítségével beállíthatjuk a szinuszos generátorbank fázisváltozásait:

$$\phi_{\text{gen}} = \frac{\phi_{\text{wnd}} + \phi_{\text{diff}}}{n_1} \tag{2.19}$$

ahol ϕ_{gen} a generátor egy minta alatt bekövetkezett, ϕ_{wnd} az ablak hosszából származó fázisváltozás, és ϕ_{diff} az ablak hossza alatt fellépő fáziseltérés. n_1 az $\mathcal{STFT}$ időtartománybeli lépése mintában.

2.2.4. Az algoritmus gyorsítása

Ha a generátorbank szinuszgenerátorokból áll, akkor nagyon nagy számítási kapacitást igényel az algoritmus. Hogy ezt elkerüljük, a szinuszgenerátorok helyett használhatunk $\mathcal{STFT}^{-1}$-et, így lényegesen csökkentve a számítási igényt. ϕ_{diff} számolása teljesen ugyanúgy történik, mint eddig, de ϕ_{gen} kiszámolása helyett most az $\mathcal{STFT}^{-1}$-hez szükséges fázisinformációt számítjuk ki a következőképpen:

$$\phi_{\text{pos}} = \text{wrap}\left(\phi_{\text{pos}} + \frac{\phi_{\text{wnd}} + \phi_{\text{diff}}}{n_1} n_2\right) \tag{2.20}$$

ahol ϕ_{pos} rekurzívan számolódik, és mindig az aktuális fázisértéket tárolja, biztosítva ezzel a fázis folytonosságát. Kezdeti értéke vagy nulla, vagy random értékkel töltjük fel. n_1 az analízis, és n_2 a szintézis időtartománybeli lépése.

Az így kiszámolt fázisértékek és az interpolált amplitúdók visszaírva derékszögű koordináta-rendszerbe, lehetővé teszi, hogy $\mathcal{STFT}^{-1}$-t számoljunk. Az időtartományban megkapott jelet megszorozzuk egy ablakfüggvénnyel, hogy elkerüljük az ablak szélén a pattanást, majd összegezzük a többi ablakkal:

$$y_n = \mathcal{STFT}^{-1}\left\{A_k e^{j\phi_{\text{pos}}}\right\} \cdot \text{W} \tag{2.21}$$

feltételezve, hogy A_k, és ϕ_{pos} a megfelelő értékekkel rendelkeznek, és W a pattanás elsimítására szolgáló ablakfüggvény, akkor a kimeneti jel felírható y_n-ek összegeként:

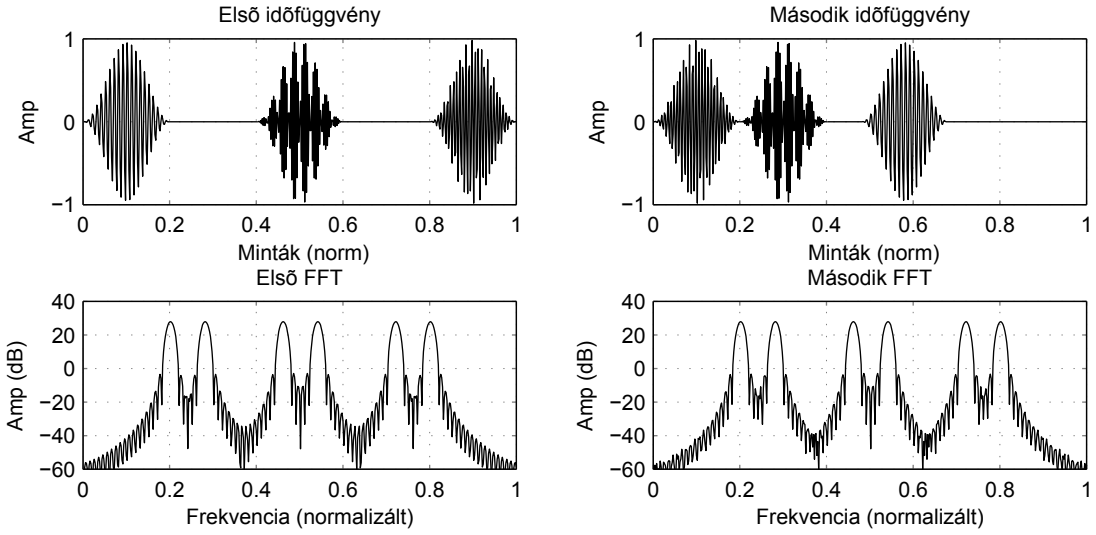
$$z[n] = \sum_{i=0}^{N} y_i[n - in_2] \tag{2.22}$$

ahol $y_i[n - in_2]$ azt jelenti, hogy az y_i-t a megfelelő helyre csúsztatva összegezzük $z[n]$ jellé.

2.2.5. Tranziens jelek

A tranziens jeleket az algoritmus „szétkeni”. Ennek az oka az említett frekvenciasávok közti inkoherencia, valamint az $\mathcal{STFT}$ felbontása. Minél nagyobb ablakot használunk az $\mathcal{STFT}$-nél, annál pontosabb lesz a frekvenciafelbontás, de annál rosszabb az időbeli felbontás. Hiszen az $\mathcal{STFT}$ egy ablakon belül csak azt határozza meg, milyen frekvenciakomponensek vannak, és mekkora a köztük lévő arány, azt nem tudjuk meg belőle, hogy az adott komponens az ablakon belül hol kezdődött és hol ért véget. Ha kisebb ablakot használunk, akkor ugyan jobb lesz az időtartományban a felbontásunk, de ezzel együtt a frekvenciatartományban csökken.

A 2.15. ábrán látható három jel összege. A három jelhez három különböző frekvenciájú szinuszjelet használtam, melyeket Hann ablakkal szoroztam, és elhelyeztem őket kétféleképpen az időtartományban, majd mindkét jelet transzformáltam.



2.15. ábra. Az $\mathcal{STFT}$ tranziensérzéketlensége

Az amplitúdókarakterisztikájuk az időfüggvények alatt látszik, és azt mutatja meg, hogy a két amplitúdóspektrum megegyezik.

2.2.6. További lehetőségek

Az A.2. függelékben mellékelt fázis vokóder implementációnak van néhány előnye, amit ki lehet használni. A jelet az $\mathcal{STFT}$ segítségével amplitúdó- és fázisértékekre bontjuk, melyek időben változnak. Minden amplitúdó- és fázisérték egy megadott frekvenciához tartozik, de nem kötelező a generátornak is ilyen frekvenciájúnak lenni. Így megtehetjük azt, hogy a generátorok frekvenciáját elhangolva elérhetjük azt, hogy a frekvenciakomponensek összekeveredjenek, eltolódjanak. Ezzel akár harmonizálni is lehet egy olyan jelet, melynek a felharmonikus frekvenciái valamilyen oknál fogva elcsúsztak [10]. Sőt, pitch shiftelni is lehet a jelet, miközben a sebességét állítjuk. Nincs szükség egy másik algoritmusra, ami ezt megvalósítja, és a számítási igénye sem lesz nagyobb, hiszen csak a bemeneti paramétereket változtatjuk meg.

A másik lehetőség az amplitúdók módosítása. Mivel minden frekvenciasávnak megvan az amplitúdóváltozása az idő függvényében, így megvalósíthatunk többsávos (ami jelen esetben több ezres nagyságrend) zajszűrőt. Ezt például úgy tehetjük meg, hogy egy másodfokú függvénnyel szorozzuk meg az amplitúdóértékeket, melyek a $[0; 1]$-es tartományban vannak, ettől a kis amplitúdók még kisebbek lesznek. Így a spektrumban szétterülő zajt hatékonyan el lehet nyomni, míg a jel kevés torzulást szenved. Automatikus hangszín-kiegyenlítő, vagy zene lekeverésénél használt többsávos dinamikakompresszor is ugyanígy megvalósítható minden nehézség nélkül.

Ezzel a két tulajdonsággal az algoritmus elég kompakt, és több funkció ellátásra alkalmas egymagában. Bár a zajelnyomással és a felharmonikusok összekeverésével ebben a dolgozatban nem foglalkozom, a függelékben található MATLAB kód kommentezett részébe írtam mintapéldát ezekre is.

2.2.7. Összefoglaló

A fázis vokódolás jobb eredményt adó algoritmus, mint az előző részben tárgyalt időtartománybeli szegmentálás. Természetesen a tranziens jelek, és nagy sebességállítás esetén itt is fellépnek nem kívánt hatások, mint a „szétkenődés". Hátránya még, hogy nagyobb számítási kapacitást igényel, mind az $\mathcal{STFT}$, mind a generátorbank megvalósítása. Ezen egy kicsit javít a generátorbank helyett használt $\mathcal{STFT}^{-1}$, de mégis lényegesen nagyobb számítási igénye van, mint a szegmentálásnak.

A szimulációim során a 2048 mintás analízis ablakot kielégítőnek találtam. Az 1024 mintás frekvenciafelbontása még kevésnek bizonyult. Ablakfüggvénynek Hanning ablakokat használtam, és általában 75%-os átlapolódást volt érdemes használni.

2.2.8. Speciális esetek

Ennek az algoritmusnak is van több változata, melyek különböző kisebb-nagyobb komplexitású javításokat tartalmaznak. Például van tranziensfelismeréses kiegészítése, mely a jelben megtalálja a tranzienseket, és nem változtatja meg őket [11]. Ezt úgy éri el, hogy ahol tranziens jelre akad, ott a fázisváltozást visszaállítja a tranziens jelek frekvenciakomponenseinél az eredeti jel fázisváltozásaira. Így eléri, hogy azok nem „kenődnek" szét, valamint, mivel csak a szükséges frekvenciákon hajtja ezt végre, így tranziens jel mellett szóló stacionárius jelek nem torzulnak az átállítástól. Ezzel a fázis vokóder minősége tovább javítható.

Több frekvenciasáv egymás közti fáziskoherenciájának megtartására törekvő algoritmus is van [12]. A fázisinkoherencia azért nem kívánatos jelenség, mert olyan hatást kelt mintha a jelforrás eltávolodott volna a mikrofontól, és „elveszett" volna a térben. Ezt az algoritmus úgy próbálja kiküszöbölni, hogy figyelembe veszi az egymáshoz közeli sávokat, és ezek fázisát befolyásolja.

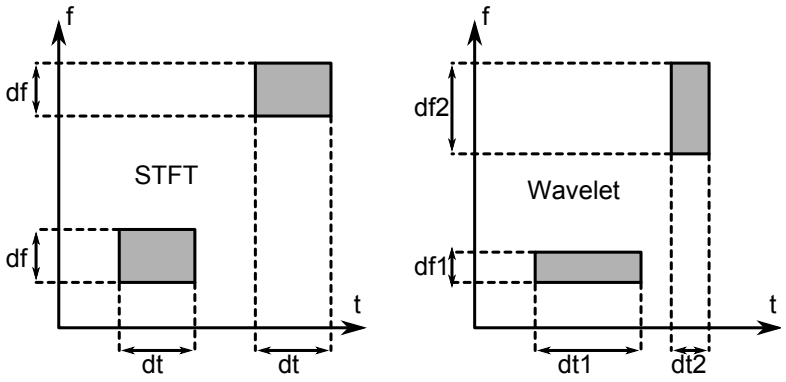
2.2.9. Idő- és frekvenciafelbontás

Mivel a hibák egyik oka az $\mathcal{STFT}$ felbontási problémája, ezért az algoritmus javításához ezt kellene megoldani. Egy megoldás a több párhuzamosan futó, más ablakméretet használó $\mathcal{STFT}$ alkalmazása [13]. Egy másik, de hasonló megoldás lehet a folytonos idejű wavelet-transzformáció használata, mely elvileg jobb felbontást biztosít, mint az $\mathcal{STFT}$ [14]. Az algoritmus nagy része megegyezik az eredeti fázis vokóderrel, csak a wavelet-transzformációt használja az $\mathcal{STFT}$ helyett, és ekkor a generátorbank sem feltétlen szinuszokból áll, hanem waveletekből.

2.3. Wavelet-transzformációs fázis vokódolás

A wavelet-transzoformáció az $\mathcal{STFT}$-vel ellentétben az idő-frekvencia síkot nem azonos téglalapokra bontja, hanem különböző frekvenciákon különböző időfelbontást alkalmaz [3]. Ezt szemlélteti a 2.16. ábra.

Az egyes téglalapok minimális területének a Heisenberg-féle határozatlansági elv szab korlátot, mely alapján teljes pontossággal nem tudjuk az időtartományban és a frekvenciatartományban egyszerre lokalizálni a jelet.



2.16. ábra. idő-frekvencia sík felbontása

Az alap függvényt ami segítségével a wavelet-transzformációt elvégezzük mother waveletnek, a belőle nyújtással, zsugorítással, és eltolással származtatott többi függvényt daughter waveletnek hívjuk. Ahhoz hogy waveletről beszélhessünk, a mother waveletnek teljesítenie kell pár feltételt. Így ha $\psi(t)$ a mother wavelet, akkor $\psi(t) \in \mathrm{L}^2(\mathbb{R})$, ami megfelel a következőnek:

$$\int_{-\infty}^{\infty} |\psi(t)|^2 \, \mathrm{d}t < \infty \tag{2.23}$$

Ezenkívül az integrálja nulla kell, hogy legyen:

$$\int_{-\infty}^{\infty} \psi(t) \mathrm{d}t = 0 \tag{2.24}$$

Valamint normalizálva kell, hogy legyen $|\psi(t)|^2 = 1$. Ekkor a mother waveletből származtatható daughter waveletek, időtartományban történő nyújtással és zsugorítással, valamint eltolással számíthatók:

$$\psi_{u,s}(t) = \frac{1}{\sqrt{s}} \psi \left(\frac{t-u}{s} \right) \tag{2.25}$$

ahol u az eltolás, és s a nyújtás, zsugorítás. Az $1/\sqrt{s}$ miatt a daughter waveletek is normalizáltak maradnak: $|\psi_{u,s}(t)|^2 = 1$. Így már felírható a folytonos wavelet-transzformáció képlete:

$$\mathcal{CWT} \left\{ f(t) \right\}_{u,s} = \int_{-\infty}^{\infty} f(t) \psi^*_{u,s}(t) \mathrm{d}t \tag{2.26}$$

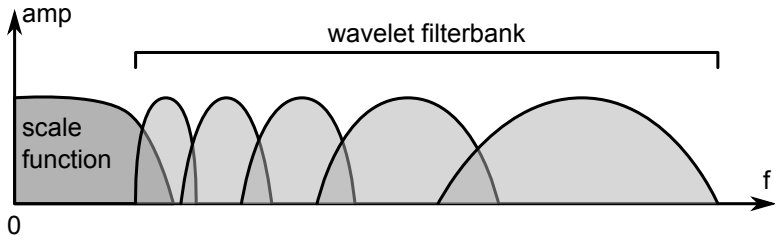
ahol a csillag a komplex konjugáltat jelöli. Az inverz transzformáció a következőképpen számítható:

$$f(t) = \frac{1}{C_\psi} \int_{-\infty}^{-\infty} \int_{-\infty}^{-\infty} \frac{1}{s^2} \mathcal{CWT} \left\{ f(t) \right\}_{u,s} \psi_{u,s}(t) \mathrm{d}s \mathrm{d}u \tag{2.27}$$

ahol $f(t) \in \mathrm{L}^2(\mathbb{R})$, és C_ψ egy konstans, mely a következő módon számolhat:

$$C_\psi \coloneqq \int_{-\infty}^{-\infty} \frac{|\hat{\psi}(\omega)|}{|\omega^2|} \mathrm{d}\omega < \infty \tag{2.28}$$

ahol $\hat{\psi}(\omega) = \mathcal{F}\{\psi(t)\}$, azaz $\psi(t)$ Fourier-transzformáltja. A wavelet-tranformáció felfogható mint egy szűrőbank, és az egyes waveletek szűrőket határoznak meg [2]. A waveletek által meghatározott szűrők sávátereszt˝o jellegűek, ez látszik abból is, hogy integráljuk nulla kell, hogy legyen, így, hogy a teljes jelet rekonstruálni lehessen, elvileg szükséges egy „scale function”-nak nevezett függvény, mely a waveletek által le nem fedhető 0 Hz körüli részt lefedi. Ezt szemlélteti a 2.17. ábra.



2.17. ábra. Wavelet szűrők és a scale function

Mivel az ember által hallható tartomány 20 Hz - 22 kHz között van, így az audio jeleknél ezzel a scale functionnel nem kell számolni, és elhagyható. Így ugyan nem lesz tökéletes a rekonstrukció, de csak a 0 Hz körüli frekvenciákat veszítjük el, melyek amúgy sem hordoznak érdemi információt az audiojelre nézve.

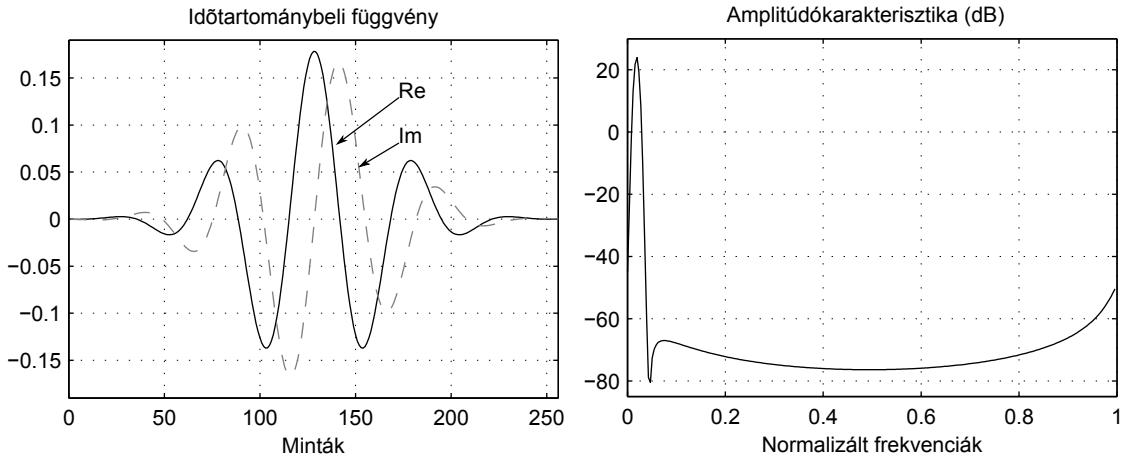
A folytonos wavelet-transzformációt diszkretizálni kell, hogy mintavételezett jeleken számolni tudjuk. Ezt úgy érhetjük el, hogy mind az eltolási mind a skálázási faktort diszkretizáljuk, a waveleteket pedig mintavételezzük, és így az integrálás helyett összegzést használhatunk.

2.3.1. Az algoritmus működése

Ha a wavelet-transzformációt alkalmazzuk a fázisvokodolás végrehajtására, akkor ez annyi különbséget jelent, hogy most nem valódi frekvencia- és idő-tengely szerint bontjuk fel a jelet, hanem s és u, skálázási és eltolási paraméterek terébe transzformáljuk át [15]. A fázisvokóder amplitúdó- és a fázisértékeket használ a jel megváltoztatására, hogy a fázisinformációnk is megmaradjon, így szükséges, hogy komplex waveletet használjunk mother waveletnek [3]. Az audio jeleknél további feltétel, hogy jól lokalizált legyen frekvenciában a wavelet által meghatározott szűrő, ezért leggyakrabban komplex morlet waveletet használnak [14][16], mely egy komplex exponenciálissal modulált Gauss-ablaknak felel meg, így a frekvenciatartományban könnyebb kézbentartani.

A 2.18. ábra szemlélteti a morlet waveletet. Az így kinyert amplitúdó- és fázisinformációkkal ugyanúgy eljárva, mint a fázisvokódernél (unwrap...) szintén meghajthatunk egy szinusz generátorbankot. Ezenkívül van lehetőség a scale faktorok átállításával, és inverz wavelet-transzformáció végrehajtásával arra, hogy azonnal pitch shifteljük a jelet [15].

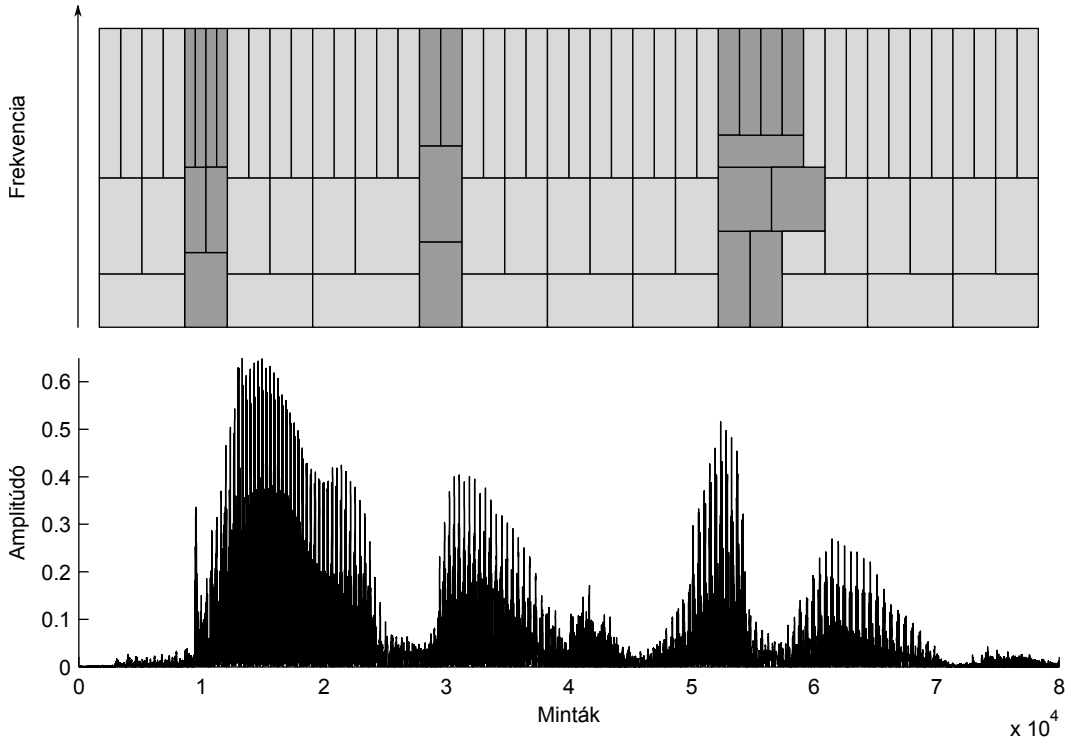
A szimulációimban én is ablakfüggvény-modulált waveletet használtam, de nagyobb szabadságot hagyva a wavelet beállításánál, a moduláció állítható a MATLAB függvényemben. A kinyert amplitúdó- és fázisinformációt ugyanúgy kezeltem, mint a fázisvokódernél, biztosítva a fázis a folytonosságát. Ezután egy szinuszgenerátorbankot hajtottam meg a módosított amplitúdó- és fázisinformációkkal.



2.18. ábra. Morlet wavelet

2.3.2. Adaptivitás

Mivel egy adott középponti frekvenciával rendelkező wavelet-szűrő idő- és frekvencia-felbontása állítható az algoritmus működése közben, így lehetőség van arra, hogy a jelhez alkalmazkodjon, és ha időben rövid lefutású tranziens részek vannak a jelben, a megfelelő frekvenciájú wavelet-szűrők időfelbontását növelve – így a frekvenciafelbontást csökkentve –, a periodikus részeknél pedig a frekvencia-felbontást növelve – az időfelbontást csökkentve –, a jelet pontosabban írja le. Ezt szemlélteti a 2.19. ábra.



2.19. ábra. Adaptív alkalmazkodás

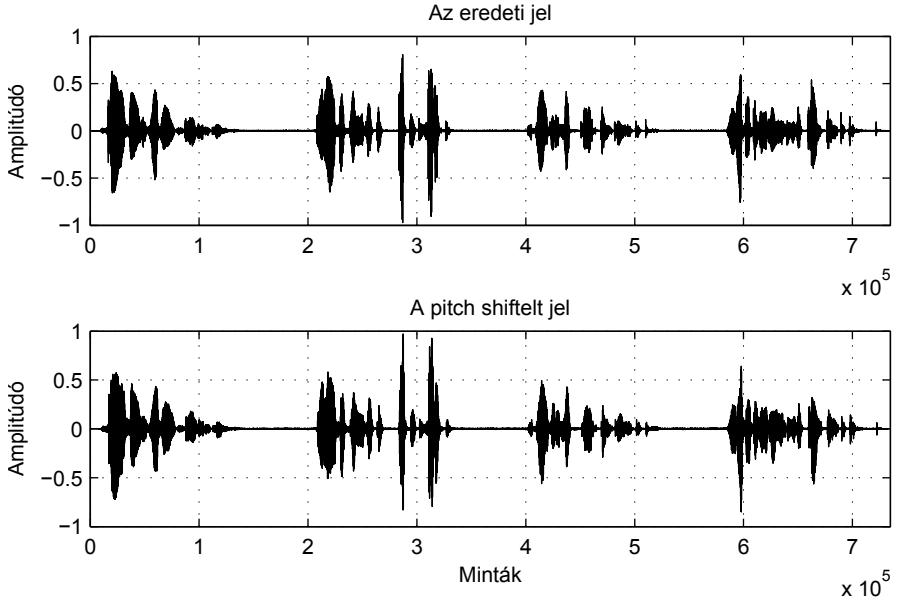
Persze azt biztosítani kell, hogy a spektrum a szűrők által teljesen le legyen fedve, és egy részt csak egy szűrő fedjen le, amit szűrőszélességek állításánál nem triviális kivitelezni. Az itt leírt elgondolás nagyon hasonló a 2.2.9. alszakaszban említett párhuzamos $\mathcal{FFT}$-alapú módszerhez [13].

3. fejezet

Pitch shift

„Hangok gyülevész csoportja,
mi rendezi őket sorba?”
Keresztury Dezső

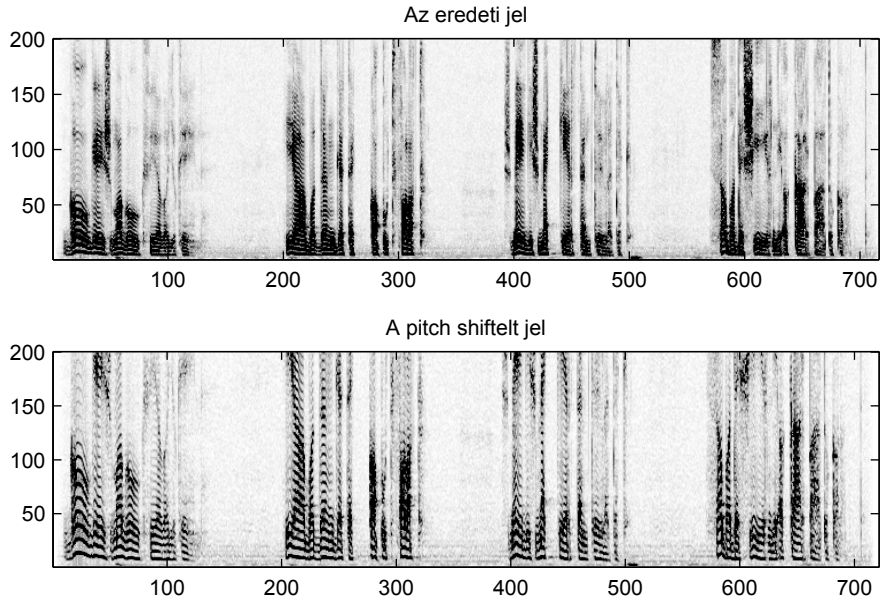
A time stretch algoritmus rokonalgoritmusa a pitch shift, mely működésében nagyon hasonló, de a pitch shift algoritmus a jelfolyam hosszát változatlanul hagyja, míg a frekvenciakomponenseit eltolja, másik hangra emeli. Felhasználási területe stúdiótechnikában főleg ének-, de akár más hangsáv korrigálása, egyszólamú éneksáv többszólamúvá alakítása.



3.1. ábra. A pitch shift az időtartományban

A 3.1. ábrán látható, hogy az eredeti és a pitch shiftelt jelnek ugyanaz az időbeli lefutása, nem változik meg a feldolgozás során az hossza.

A 3.2. ábrán pedig azt látjuk, hogy a frekvenciatartományban a feldolgozott jel frekvenciakomponensei el vannak csúsztatva, így más hangmagasságon szólal meg.



3.2. ábra. A pitch shift a frekvenciatartományban

3.1. Hangmagasság állítása

A pitch shift által eltolt frekvenciák nem úgy változnak meg, hogy minden frekvenciakomponens azonos Δf frekvenciával tolódik arrébb:

$$f_k^{'} \neq f_k + \Delta f \tag{3.1}$$

hiszen ez megegyezne a Fourier-transzformáció modulációs tételével:

$$\mathcal{F}\left\{e^{-j\omega_0 t} f(t)\right\} = F(\omega - \omega_0) \tag{3.2}$$

ez esetben nem is kellene hozzá a sebességállítás, hiszen Hilbert-szűrővel és szorzással meg lehetne oldani, oly módon, hogy a beérkező jelet Hilbert-szűrjük, ettől az kilencven fokos fázistolást szenved, majd az eredeti jelet megszorozzuk koszinusszal, a Hilbert-szűrtet pedig szinusszal, majd ezeket összeadjuk. A koszinusz és szinusz azonos ω frekvenciájú. Ez az ω frekvencia lesz az eltolás nagysága. A pitch shift által eltolt frekvenciáknak harmonikusoknak kell maradniuk. Tehát egy adott r_{pitch} értékszeresére kell minden frekvenciának változnia:

$$f_k^{'} = f_k \cdot r_{\text{pitch}} \tag{3.3}$$

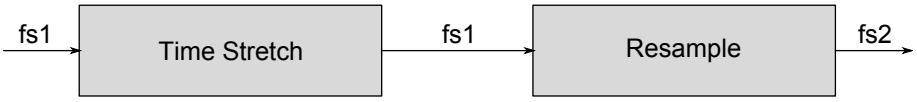
3.2. Származtatás time stretchből

Ha az analóg világ példájából indulunk ki, akkor egy lemezlejátszó lassítást kellene megvalósítani digitálisan. Ezzel az a gond, hogy a lemezlejátszónál a jel sebessége is lassul. De ha felhasználnánk hozzá a 2. fejezetben leírt time stretch algoritmusok valamelyikét, hogy ezt a lassulást ellensúlyozzuk, akkor elérhetnénk, hogy a jel hossza

ne változzon meg, de hangmagasságok igen. Már csak a lemezlejátszó lassulását kell reprodukálni, amit meg lehet valósítani például újramintavételezéssel.

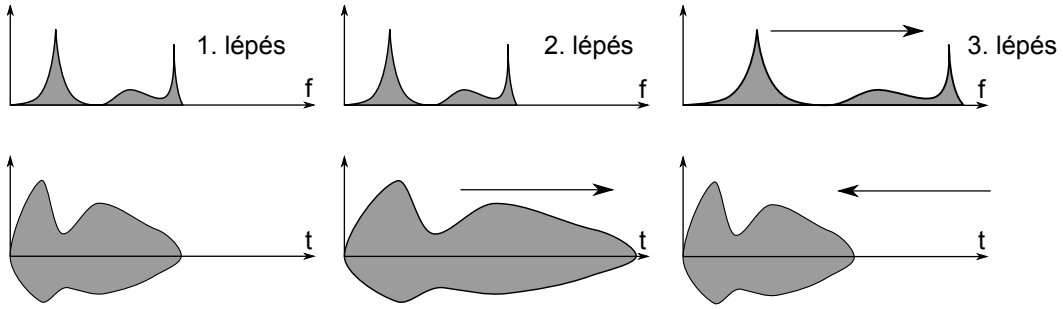
Az újramintavételezés felfogható úgy, mint analóg sebességállítás. Ha növeljük a minták számát a jelben, és az eredeti mintavételi frekvencián játsszuk le őket, akkor a jelünk mélyülni fog, ha csökkentjük a minták számát, és az eredeti mintavételi frekvencián játsszuk le őket, magasabb lesz.

Ha a time stretchelt jelünket úgy újramintavételezzük, hogy az éppen ellensúlyozza a time stretch hatását, például egy kétszer olyan hosszúra time stretch-elt jelet újramintavételezünk az eredeti mintavételi frekvenciája felével, akkor az eredeti jel hosszát kapjuk vissza, csak a frekvenciakomponensek máshol lesznek. Mindegyik feleakkora lesz, mint az eredeti jelben, tehát egy oktávval lejjebb toltuk a hangokat. A 3.3. ábrán látható a pitch shift blokk diagramja, benne a time stretch és az újramintavételezés elhelyezkedése. A kimeneten megjelenő f_{s2} jelet az eredeti f_{s1} mintavételi frekvenciával kell lejátszani.



3.3. ábra. A pitch shift blokkdiagramja

Az idő- és frekvenciatartományban végbement változásokat a 3.4. ábrán lehet megfigyelni. Az első lépés után a jel hossza változik, de a frekvenciakomponensei változatlanok maradnak, majd az újramintavételezés után megváltozik a hossza, és a frekvenciakomponensei.



3.4. ábra. Változások idő- és frekvenciatartományban

Az újramintavételezés és a time stretch sorrendje felcserélhető, hiszen mindegy, hogy előbb nyújtjuk meg a jelet, és utána változtatjuk meg a mintavételi frekvenciát, vagy fordítva. Ennek ellenére azért lehet minimális különbség a két elrendezés között, attól függően, milyen paraméterek mellett állítjuk a hangmagasságokat.

Nem ez az egyetlen módja a pitch shift számításának, hiszen ahogy azt a 2.1.8., a 2.2.6., és a 2.3.1. alszakaszokban is láttuk, vannak olyan algoritmusok, melyek a pitch shiftet azonnal, újramintavételezés nélkül végrehajtják.

3.3. Újramintavételezés

Az újramintavételezésre többféle algoritmust használhatunk. Létezik olyat, melyhez szükség van a jel összes mintájára, tehát csak offline módon, a times stretch befejeződése után tudjuk végrehajtani, de használhatunk online algoritmust is, melynek segítségével a time stretch algoritmussal egy időben tudjuk végrehajtani az újramintavételezést, és így a pitch shiftet is [17]. Az alábbiakban ezek közül ismertetek néhányat.

3.3.1. Offline újramintavételezés

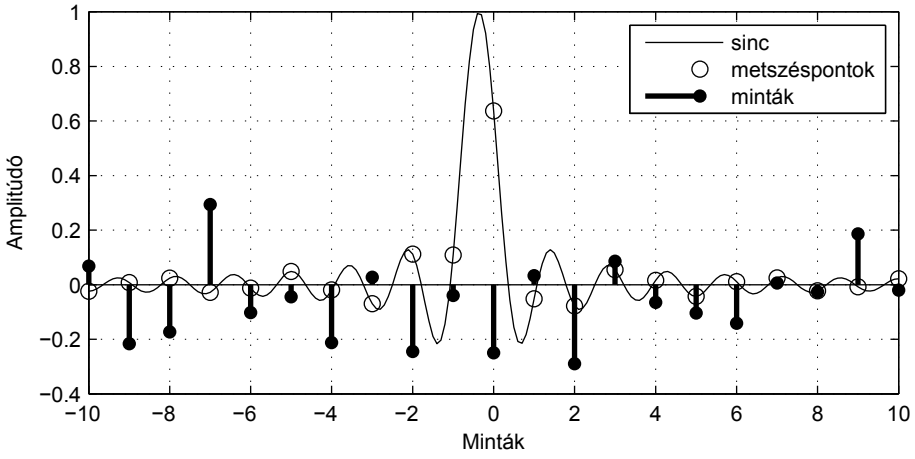
Az offline algoritmusok előnye, hogy jobb minőségű újramintavételezést produkálnak, hiszen nem időkritikus már a feladat, és rendelkezésre áll az összes minta a jelből.

Whitaker újramintavételezés

A sinc alapú interpoláció egy sinc alakú jellel súlyozza a mintákat, mely súlyozott összegből számítja ki az aktuális mintát:

$$x(t) = \sum_{n=-\infty}^{\infty} x[n] \cdot \mathrm{sinc}\left(\frac{t - nT}{T}\right) \tag{3.4}$$

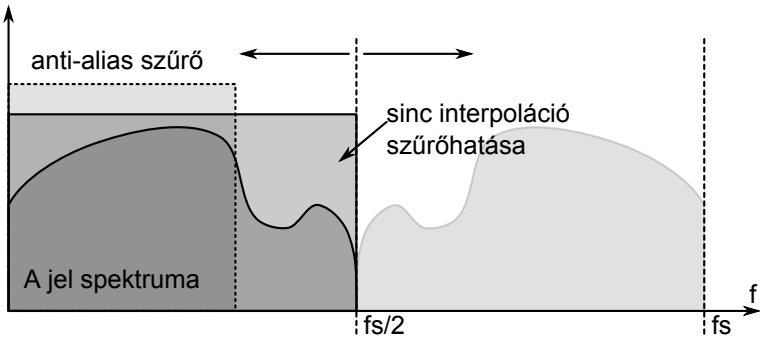
ahol $T = 1/f_\mathrm{s}$ a mintavételezési idő, valamint sinc a normalizált sinc függvény. Ha a sinc-et jól választjuk meg, akkor az új mintavételi sebességre való áttéréskor nem történik átlapolódás, mert a sinc a jelet egyúttal meg is szűri. Ehhez az kell, hogy az új mintavételi sebességre skálázott – ha szükséges a skálázás – sinc függvényt időtartományban arra helyre toljuk el, ahol a mintavételezett jel új mintáját ki szeretnénk számítani. Ekkor a sinc skálázása és az eltolás miatt, a jel mintái már nem csak egy helyen metszik zérustól különböző értékkel a sinc-et. A kiszámítandó minta ezen metszésponti minták sinc által súlyozott összege. Ezt szemlélteti a 3.5. ábra.



3.5. ábra. Sinc alapú újramintavételezés

Ha másik szemszögből nézzük, ez megfelel egy konvolúciónak. A jelet felfoghatjuk egy Dirac-delta sorozatnak, mely $x[n]$ konstansokkal van súlyozva, és a Dirac-delták között zérus értékű. Ekkor a sinc egy ideális aluláteresztő impulzusválasza, és e két jelet konvolváljuk adott pontban.

Ebből kiderül az is, hogy a sinc skálázásra csak akkor van szükség, ha a mintavételi frekvenciát csökkenteni akarjuk, mert ekkor az átskálázott sinc – ami a frekvenciatartományban egy másik ideális aluláteresztő szűrőnek felel meg – szűri ki azt a részt a spektrumból, ami átlapolódna az interpoláció hatására. Ha a mintavételi frekvencia növekszik, akkor a sincet nem kell átskálázni, és akkor éppen kiszűri a spektrum tükörképéből átkerülő részeket, mint ahogy a 3.6. ábra is mutatja.



3.6. ábra. Sinc szűrhatásai

Spline alapú újramintavételezés

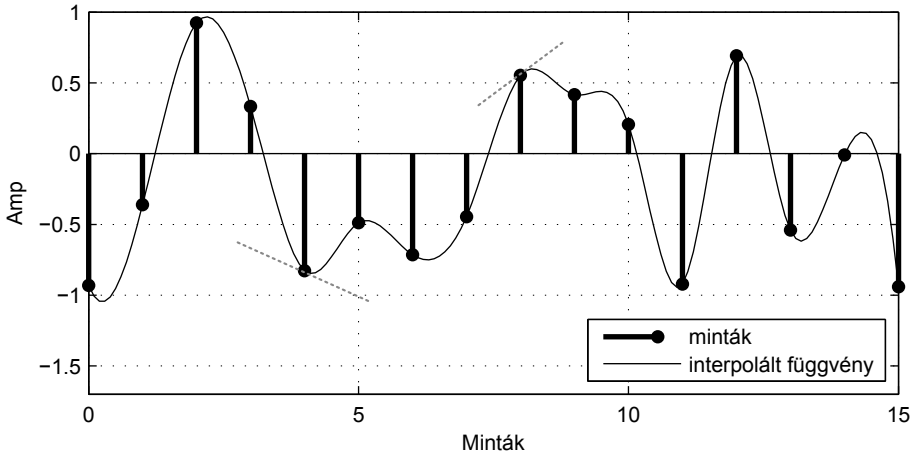
A spline alapú interpolációt használó újramintavételezés a megadott minták közé mintapáronként egy-egy – általában harmadfokú – polinom függvényt illeszt úgy, hogy a mintáknál az illesztett polinom folytonos legyen a függvény az n-edik deriváltjáig. Ha az új mintavételi frekvencia alacsonyabb a réginél, akkor szükséges átlapolódás gátló szűrő a spline interpoláció előtt, hogy a frekvenciatartományban ne legyen átlapolódás [18].

Ha $S(x)$ az interpoláló függvény, amit úgy definiálunk, hogy:

$$S(x) = \begin{cases} S_0(x), & x \in [x_0, x_1] \\ S_1(x), & x \in [x_1, x_2] \\ \dots \quad , & \dots \\ S_{n-1}(x), & x \in [x_{n-1}, x_n] \end{cases} \tag{3.5}$$

ahol $S_k(x)$ jelenti a k-adik szakaszra illesztett függvényt. Az interpoláció szükséges feltétele, hogy az ismert pontokat felvegye $S(x)$, így $S(x_i) = x_i$ $(i = 0, \dots, n)$ egyenletnek teljesülnie kell, valamint két szakasz illesztésénél nem lehet ugrás, ezért $S_{i-1}(x_i) = S_i(x_i)$ $(i = 1, ..., n-1)$, és a k-adik – általában első és második – deriváltjuknak is meg kell egyeznie: $S'_{i-1}(x_i) = S'_i(x_i)$, $S''_{i-1}(x_i) = S''_i(x_i)$ ...

Az így felírható egyenletrendszer megoldásához szükség van még feltételekre, melyeket a kezdeti és végpontokra felírt egyenletek adják. Ezen feltételek alapján van az egyes spline-oknak más neve. Például a harmadfokú spline-t akkor nevezzük természetes spline-nak, ha $S''_0 = S''_n = 0$. A 3.7. ábra a spline interpolációt szemlélteti.



3.7. ábra. Spline interpoláció

3.3.2. Online újramintavételezés

Az online algoritmusok előnye, hogy real-time alkalmazásokban is használhatóak, hátrányuk általában, hogy jobb minőségű újramintavételezés eléréséhez nagyobb számítási kapacitás szükséges.

Interpoláció alapú újramintavételezés

A fenti két offline újramintavételezés is felfogható mint interpoláció, de ezek azért nem valósíthatók meg online módon, mert mind a sinc-nél, mind a spline-nál egy minta kiszámolásához szükség van a jel összes mintájára, ezért ha online módon akarnánk implementálni, akauzális hálózatot kellene megvalósítanunk. Természetesen van lehetőség az akauzalitás megkerülésére, például a sinc csonkolásával, vagy a spline bizonyos paramétereit, más tapasztalati, vagy közelítő számítással meghatározni.

Az interpoláció alapú újramintavételezések az eddig ismert mintákra – vagy ezek részhalmazára – egy illesztést végeznek el és ezzel számolják ki az éppen szükséges új mintavételi frekvenciájú mintát. Ilyen illesztés lehet polinom illesztés, ahol nullad-, első-, másod- vagy n-ed fokú polinomokat illesztünk a már ismert mintákra, és ezek segítségével határozzuk meg az újat. Szokás a kimeneti mintát késleltetni, hogy az éppen számított új mintának ne csak az egyik oldaláról legyen ismert adatunk, ezzel növelve az illesztés pontosságát. Szükség van átlapolásgátló szűrőre, hogy megakadályozzuk a spektrum nem kívánt torzulását.

Az ilyen jellegű interpolációk illesztésfüggvényét átírhatjuk a frekvenciatartományba, megvizsgálva, hogy hogyan befolyásolja a jel spektrumát. Vegyük például a lineáris interpolációt:

$$y(n,k) = x[n-1] + \frac{x[n] - x[n-1]}{\Delta n} k \tag{3.6}$$

ahol $k \in [0;1]$, ami azt jelenti hogy az aktuális és az előző mintát egy egyenessel kötjük össze, így meghatározva a közöttük lévő pontokat. Ha ezt az egyenletet Z-

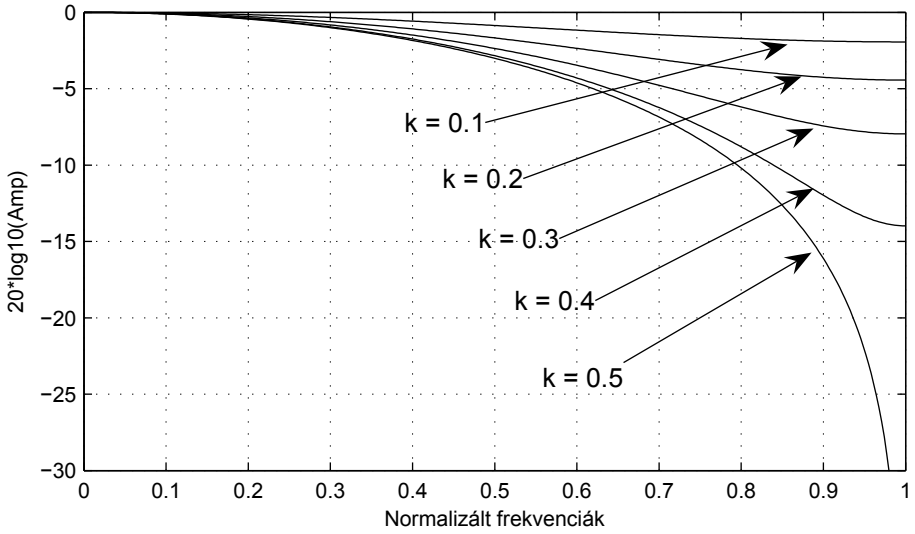
transzformáljuk, és feltételezzük, hogy a minták közti távolság 1, akkor a következőt kapjuk:

$$Y(z,k) = X(z)z^{-1} + (X(z) - X(z)z^{-1})k \tag{3.7}$$

ebből átrendezéssel kiszámolhatjuk az átviteli függvényt:

$$H(z,k) = \frac{Y(z,k)}{X(z)} = z^{-1} + (1 - z^{-1})k = \Big|_{z=e^{j\omega}} k + (1-k)e^{-j\omega} \tag{3.8}$$

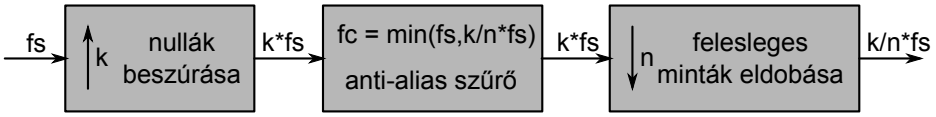
A 3.8. ábra mutatja az amplitúdóspektrumot, amiből látszik, hogy az átviteli függvény k-tól függ, és egy idővarián fésűszűrőnek felem meg, melynek egyetlen leszívása van. A legnagyobb spektrummegváltozás a két minta közötti időponthoz tartozik, ami nem meglepő, hiszen ilyenkor a két mintát átlagolja.



3.8. ábra. Lineáris interpoláció amplitúdóspektruma

Szűrő alapú újramintavételezés

A szűrő alapú újramintavételezés, a jel racionális arányú újramintavételezésére alkalmas. Ha k/n-ed részére akarjuk megváltoztatni a mintavételi sebességet, akkor először k szorosan túlmintavételezzük a jelet, ami annyit jelent, hogy minden minta után beszúrunk $k-1$ nullát, majd a jelet megszűrjük egy átlapolódásgátló (anti-alias) szűrővel, melynek töresponti frekvenciája $f_c = \min(f_s, k/n \cdot f_s)$, ezután csak minden n-edik mintát tartunk meg. A blokkdiagram a 3.9. ábrán látható.



3.9. ábra. Racionális arányú újramintavételezés

A szűrés és a minták eldobása összevonható egy polifázisos szűrővé, melynek következtében kevesebb számítási kapacitást igényel az eljárás.

4. fejezet

Megvalósítás

„A megismerés és megvalósítás között szakadék van: az győz, aki átugorja.”

Müller Péter

Feladatom része volt, hogy a 2. fejezetben említett algoritmusok DSP-n implementálható verzióit megvalósítsam ADSP-BF537 EZ-KIT Lite kártya segítségével. A választás azért erre a kártyára eset, mert az iparban is nagyon gyakran előforduló nem túl költséges Blackfin processzorcsalád tagját tartalmazza. A gyakorlatban is viszonylag sokszor ilyen processzorral találkozhatunk, a drágább, de nagyobb teljesítményű például lebegőpontos processzorok, például SHARC, vagy TigerSHARC helyett.

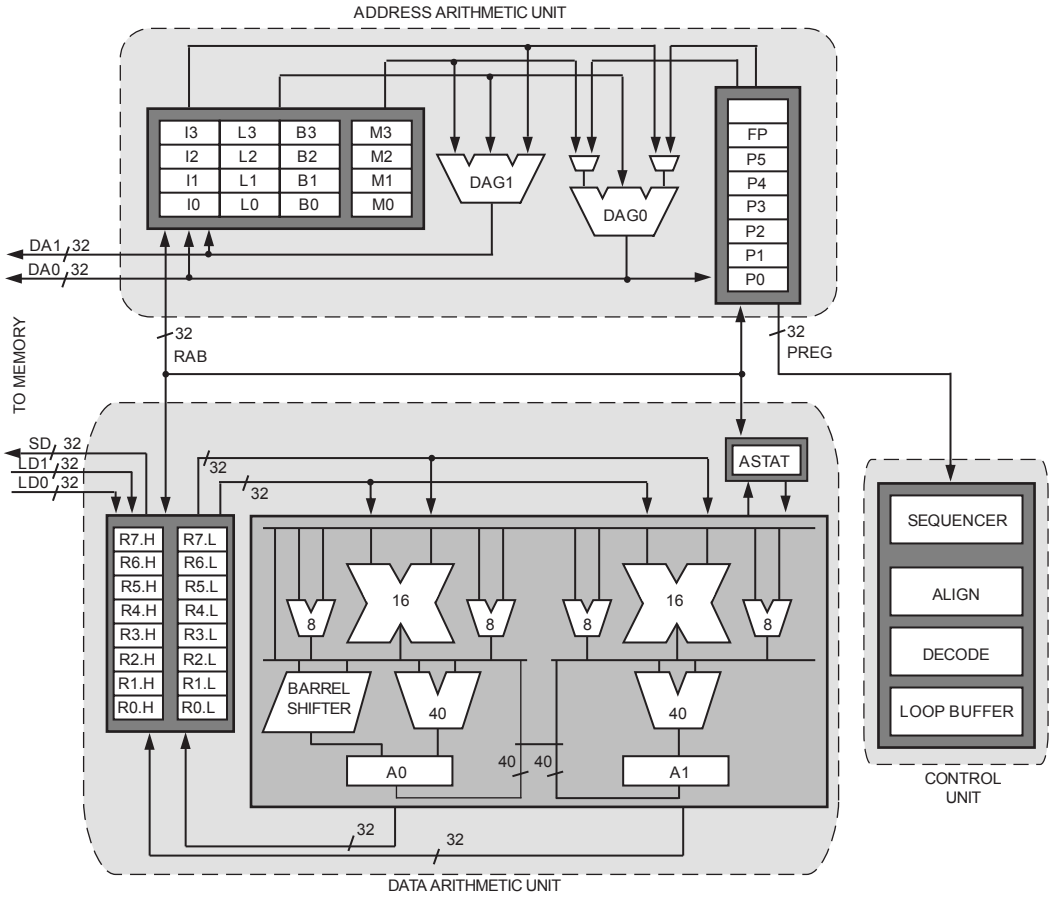
4.1. Az ADSP-BF537 EZ-KIT Lite kártya

Az ADSP-BF537 EZ-KIT Lite az Analog Devices cég által gyártott fejlesztőkártya. A kártya gyors és hatékony fejlesztéseket tesz lehetővé Blackfin processzorokra. Perifériái és kivezetései jól használhatóak, és debugolási lehetőségek is vannak rajta.

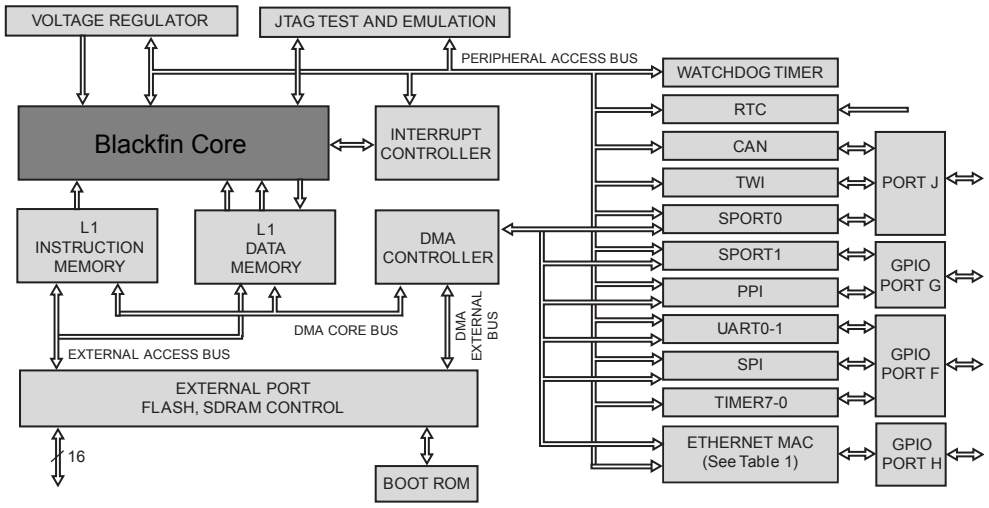
4.1.1. A processzor

A kártyán lévő processzor Blackfin BF-537-es. Akár 600 MHz-en is üzemelni képes fixpontos egymagos processzor, három 16 bites MAC, két 40 bites ALU, négy 8 bites video ALU, és egy 40 bites shifter található benne. RISC utasításkészletével és SIMD kapacitásával rugalmas és sokféle jelfeldolgozási feladatban használható architektúra. A processzormag architektúráját a 4.1. ábra szemlélteti.

A processzor sok perifériával rendelkezik, többek között 10/100 Ethernet MAC, CAN interface, két dual-channel full-duplex szinkron soros port (SPORT) – amivel 8 sztereó I^2S csatornát tud lekezelni –, PPI interface, 16 csatornás DMA, SPI interface, két UART – IrDA támogatással –, 2-wire interface (TWI) vezérlő, és 48 GPIO. A perifériák elhelyezkedését a 4.2. ábra szemlélteti.



4.1. ábra. BF-537 processzormag architektúrája (forrás: Analog)



4.2. ábra. Perifériák blokkvázlata (forrás: Analog)

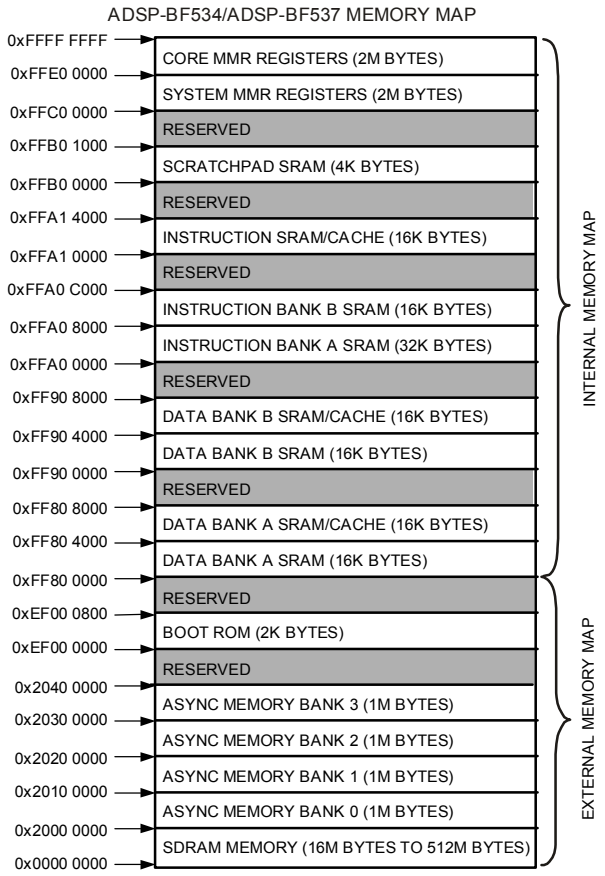
4.1.2. A memória

A processzor 320 kB belső memóriával rendelkezik, de a kártyán található még 64 MB külső memória is, 32 M szó 16 bites szószélességgel. Ezenkívül van még 4

MB flash memória, 2 M szó 16 bites szószélességgel.

4 GB-nyi címterületet kezel 32 bites címekkel. Ezen a területen keresztül éri el a belső, külső memóriát és az I/O vezérlő regisztereket. A memóriák hierarchikusan vannak felosztva. Ez azt jelenti, hogy az L1 memória a leggyorsabban elérhető, a külső SRAM pedig a leglassabban.

Az első belső memóriablokk az L1 programmemória, melynek 64 kB-ját a processzor teljes sebességgel el tudja érni. A következő on-chip blokk az L1 adatmemória, mely két 32 kB-os bankból áll, ezt a területet is eléri a processzor teljes sebességgel. Ezután következik egy 4 kB-os scratchpad rész, mely ugyanolyan gyors mint az L1 memóriák, de csak adatmemóriaként érhető el, és nem használható mint cache memória. A belső memóriák után a külső SDRAM és a flash memória következik. A memóriaterületek elhelyezkedését a 4.3. ábra mutatja be részletesen.



4.3. ábra. Memóriatérkép (forrás: Analog)

Megfigyelhető, hogy a belső memória szét van darabolva, és a legnagyobb egybefüggő blokk mérete 32kB. Ez azt jelenti, hogy 16 bites adatokból 16 K szó fér el, ami kicsit több memóriaigényű, de nagy gyorsaságot megkövetelő online feladatokhoz kevés lehet.

4.1.3. Audio átalakítók

A kártyán található egy AD1871-es sztereó Σ-Δ AD átalakító. A processzor 24 bites kvantálással, és 48 kHz mintavételi frekvencián használja, és SPORT-on keresztül kapja az adatot az átalakítótól. Maga a AD képes lenne 96 kHz-es mintavételi frekvenciára is, és az adatlapja alapján 100 dB körüli dinamikatartománnyal rendelkezik.

A DA-átalakító egy AD1854-es sztereó Σ-Δ átalakító. Szintén 24 bites kvantálás és 48 kHz-es mintavételi frekvencia mellett működik, és a processzor a SPORT másik csatornáján kommunikál vele. Ez az átalakító is képes 96 kHz-es mintavételi frekvenciára, és az adatlapja alapján kicsivel több mint 100 dB dinamikatartománnyal bír.

Az AD és DA kezelésére fel lehet használni a DMA két csatornáját, hogy az AD-ről bejövő adatot fogadja egy bufferbe, és IT-t generáljon mikor egy új minta megérkezik, majd az IT lekezelése után, egy kimeneti bufferből a DA-nak küldje a mintákat. Ezt az elrendezést használtam a programomnál is.

4.2. A fejlesztőkörnyezet

Fejlesztőkörnyezetnek az Analog Devices által forgalmazott VisualDSP++ 5.0-ás verzióját használtam. Ebben az integrált fejlesztőkörnyezetben (IDE) lehetőség nyílik mind C/C++, mind ASM nyelven programozni az Analog Devices különböző processzorcsaládjait, mint például az ADSP-21xx-es, a Blackfin, a SHARC és a TigerSHARC.

Az implementációkat C/C++ nyelven írtam, mert ez a nyelv kellően magas szintű, könnyebb átlátni, és fejleszteni benne, valamint a fordító kapacitása és a processzor felépítéséből kifolyólag, egy jól lekódolt program viszonylag kevés overheaddel rendelkezik egy ASM-ben megírt kódhoz képest. Figyelembe véve az assemblyben történő fejlesztés időigényességét a választás még jobban indokoltá válik. Nagyon időkritikus részeknél lehetőség van a kódot assembly nyelven megírni, és a C/C++-os keretrendszerbe integrálni.

A fordító extra nyelvi elemekkel egészíti ki a C/C++ standardot, elérve ezzel, hogy még közvetlenebb legyen a programozó kapcsolata a DSP-vel, és a magasszintű nyelv nyújtotta absztrakció mellett képes legyen átlátni, hogy milyen folyamatok játszódnak le a processzorban. Ilyen kiegészítő például a DSP-n használt számábrázolás, mely a 16 bites fixpontos számnál úgy helyezi el a kettedespontot, hogy a 16 biten ábrázolható számok a $[-1; 1)$ tartományba kerüljenek. Ez a `fract16` típus, van 32 bites verziója is a `fract32`, vagy a `segment("<memory_segment_name>")` parancsszót használva elérhetjük, hogy az egyes változók az általunk meghatározott memóriasegmensbe kerüljenek.

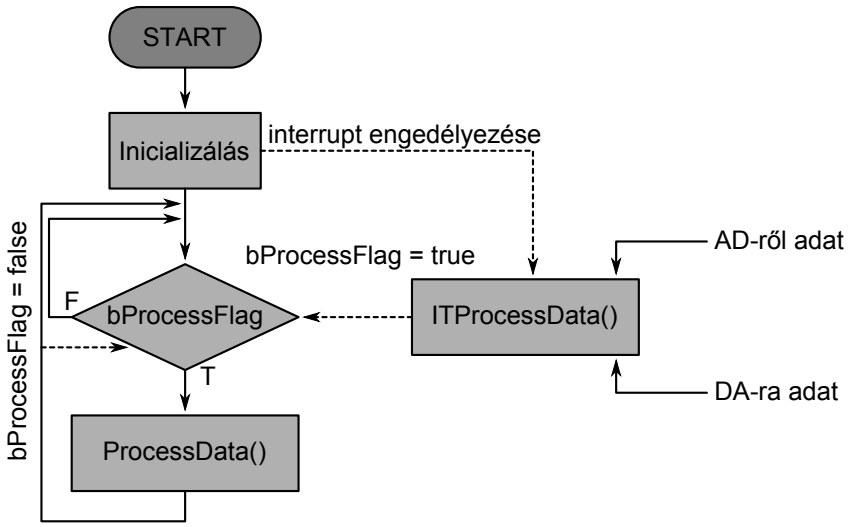
Előre megírt függvények segítenek gyakran előforduló problémák megoldásában, mint például a Fourier-, inverz Fourier-transzformáció, FIR szűrés, vektorműveletek, valamint `fract16`-os, és `fract32`-es beépített típusokra különböző műveletek. Ezenkívül alkalmas komplex számokkal számolásra, ahol a `complex_fract16` típus két `fract16`-ot tartalmaz. A `complex_fract16` típusra is vannak műveletek definiálva,

és ebből is van 32 bites verzió (`complex_fract32`).

Az IDE támogatja a debuggolást a kártya beépített JTAG interface-én keresztül. Így lehetőség van a kártyán az egyes regiszterek, memóriaterületek nyomonkövetésére, kiíratására, valamint kirajzolására. Képes a memóriaterületeket feltölteni a teszteléshez szükséges adatokkal, breakpointok elhelyezésére, lépésenkénti futtatásra, és mindennel fel van szerelve, ami egy modern IDE-től elvárható. Ezért is előnyösebb ezzel a programcsomaggal fejleszteni, mint a szabad forráskódú GCC-s fordítókörnyezettel.

4.3. A DSP program felépítése

A DSP-s feldolgozás általában két részre bontható. Az egyik a hardver inicializálása, majd az inicializálás és beállítás után az adatok feldolgozása történik, általában egy végtelen ciklusban. Ha az adatok egy aszinkron – tehát nem a processzor adott órajelenként bekövetkező – forrásból származnak, akkor általában a kiszolgálást IT rutin segítségével lehet megoldani. Ezt szemlélteti a 4.4. ábra.



4.4. ábra. DSP program általános felépítése

4.3.1. A keretrendszer

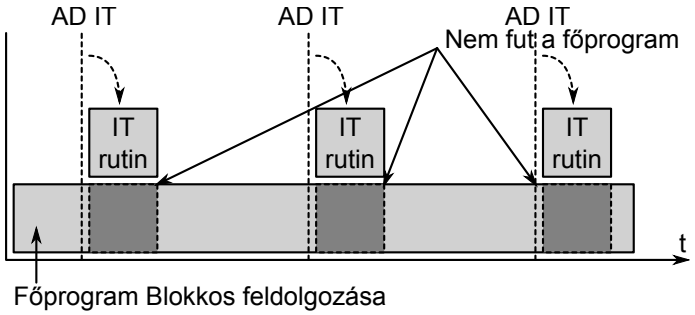
Az általam a DSP-re írt szoftveres keretrendszer feladat a DSP kártya beállítása, a különböző perifériák felkonfigurálása, hogy a kártyát úgy lehessen használni, hogy az AD-ről vett jelet feldolgozva online kitegye a DA-ra. Így az inicializálás a következő lépésekből áll:

- A DSP-n található PLL-t – ami az órajelet szorozza fel a külső 25 MHz-es kristályról –, az alap 10x-es szorzóról nagyobbra kell állítani, hogy az alap 250 MHz helyett 500 MHz-en üzemeljen a processzor.
- A megfelelő GPIO lábak ki-, és bemenetekké konfigurációja, hogy a kommunikáció az AD és DA-val megkezdődhessen.

- Ezután resetet kell küldeni az AD és DA-nak.
- Majd az SPORT0 transmit és receive paramétereit kell beállítani, hogy 24 bites sztereó audiót tudjon fogadni, és küldeni a megfelelő formában az AD felől és a DA-nak.
- Ezután a DMA következik. Be kell állítani, hogy az AD-ről automatikusan fogadja a jeleket, és egy interrupttal jelezze minden megérkezését. Ezenkívül a DMA-t arra is fel kell konfigurálni, hogy a DA-nak küldje a feldolgozott mintát.
- A következő lépés – ha van ilyen –, a feldolgozáshoz szükséges alapbeállítások elvégzése. Például $\mathcal{FFT}$ táblázatok létrehozása, a FIR szűrők inicializálása, vagy az ablakfüggvények generálása.
- Végül az interruptokat engedélyezzük, és bekapcsoljuk a DMA-t, hogy megkezdődhessen a feldolgozás.

4.3.2. Az algoritmus rutinjai

Az általános inicializálás után a főprogram egy végtelen ciklusban fut. Az algoritmusoknál a feldolgozás olyan jellegű, hogy azt nem mintánként, hanem blokkosan kell elvégezni, így a főprogramban van egy `ProcessData()` nevű függvény, ami akkor hívódik meg, ha kellő minta összegyűlt, hogy újabb blokkot dolgozzon fel az algoritmus. Ezt egy `bProcessFlag` nevű `bool` változón keresztül jelzi a `ItProcessData()` nevű függvény, amit az IT rutin hív meg minden egyes új minta beérkezésekor. Ez látható a 4.5. ábrán is.



4.5. ábra. Rutinok idődiagramja

Így az IT rutin, pontosabban az általa hívott `ItProcessData()` függvény csak mintákat gyűjt, és nem végez olyan nagy számításigényű műveleteket, amelyek miatt esetleg egy-egy minta kimaradna a feldolgozásból, abból kifolyólag, hogy a számolás ideje hosszabb, mint két minta között eltelt idő.

4.4. Időtartománybeli szegmentálás

Az első implementált algoritmus az időtartománybeli szegmentálás, melynek részletes leírása a 2.1. szakaszban olvasható. A számítási igény figyelembevételét

követően az algoritmus sztereó jel feldolgozására lett megvalósítva, ami azt jelenti, hogy minden bufferből két darab van, és minden feldolgozást kétszer végez el, egyszer a jobb, egyszer a bal csatornára.

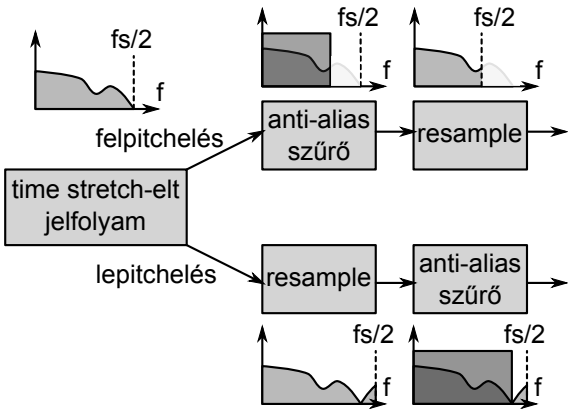
4.4.1. A működési elv

Az algoritmus `ItProcessData()` függvénye a bejövő mintákat az `iInBufferL` és az `iInBufferR` cirkuláris bufferekbe gyűjti, és ha beérkezett annyi, hogy egy újabb blokkot fel lehet dolgozni, akkor bebillenti a `bProcessFlag`-et.

Ennek hatására a `ProcessData()` elkezdi feldolgozni az új blokknyi adatot, miközben a `ItProcessData()` tovább gyűjti az új mintákat. `iInBufferL` és `iInBufferR` mérete nagyobb mint a blokkos feldolgozásé, biztosítva ezzel, hogy amíg a `ProcessData()` egy blokkon dolgozik, addig annak a blokknak a memóriacímére ne történjen írás az `ItProcessData()` által.

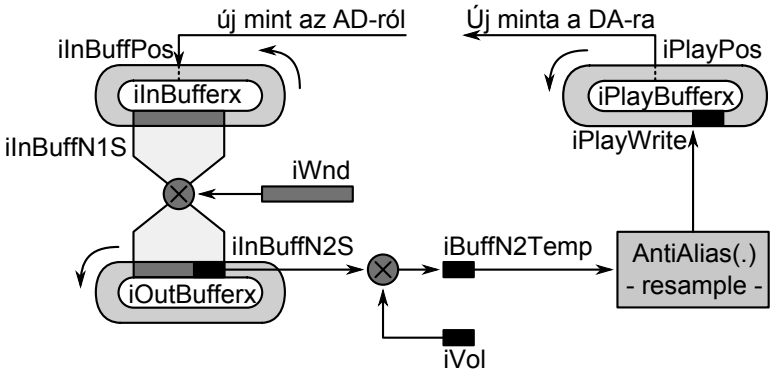
A `ProcessData()`-ban a feldolgozott blokk megszorzódik egy ablakfüggvénnyel, majd átmásolódik az `iOutBufferL` illetve `iOutBufferR` megfelelő részére. A jel itt már átesett a time stretchelésen. Mivel a feldolgozás online történik, így szükséges hogy ugyanannyi minta érkezzen, mint távozzon a DSP-ből, ezért a time stretch megvalósítása elvi korlátokba ütközne. A korlátok feloldása végett a time stretchből származtatott pitch shiftet hajtja végre az algoritmus.

Ez azt jelenti, hogy az `iOutBufferL` és `iOutBufferR`-ben lévő time stretchelt jel még átesik egy újramintavételezésen. Az újramintavételezés lineáris interpoláció, és a megfelelő anti-alias szűrők segítségével történik. Attól függően, hogy a pitch shift melyik irányba történik, az anti-alias szűrő vagy az interpoláció előtt vagy utána van. Ezt szemlélteti a 4.6. ábra.



4.6. ábra. Anti-alias szűrés

Az újramintavételezés után a jel az `iPlayBufferL` és `iPlayBufferR` cirkuláris lejátszási bufferbe kerül, valamint `iOutBufferL` és `iOutBufferR` bufferek megfelelő részei feltöltődnek nullákkal. Itt kanyarodunk vissza a folyamat elejére, mert a `ItProcessData()` nem csak minták gyűjtésére szolgál, hanem a lejátszási bufferekből folyamatosan vesz ki mintát, és küldi el a DA-ra. A program működését szemlélteti a 4.7. ábra.



4.7. ábra. Algoritmus blokkvázlata

4.4.2. Paraméterek beállítása

Több sebességállítástól függő paramétert is tartalmaz az eljárás, így ezek könnyű és egyszerű beállítására egy MATLAB szkriptet írtam, mely a felhasználótól pár paraméter bekérése után legenerál egy C/C++ header file-t amit azután már csak le kell fordítani a DSP-re, és feltölteni. Ilyen beállításfüggő paraméter például az ablakfüggvény, az ablakméretek, az átlapolódás, és az anti-alias szűrők paraméterei.

4.4.3. Megvalósítás korlátai

Mint az a 4.1.2. alszakaszban említésre került az egybefüggő belső memória területek mérete véges, ezért az ablakméret növelése nem lehetséges egy bizonyos méretnél nagyobbra. Ha a külső memóriában kerültek elhelyezésre a bufferek, akkor az elérési késleltetésből adódóan túl sokáig tartott a számolás, és kimaradtak blokkok, vagy blokk részletek, melynek hatására az audio jelben pattogások keletkeztek. Az ablak maximális mérete 4096 minta körül van, de az igazán jó beállításhoz 8192 mintányi ablakokat lenne érdemes használni. A kisebb méret okozta hiba úgy mutatkozik, hogy a fésűszűrő hatás – vagy másik szemszögből frekvenciaszelektív AM moduláció – sokkal erősebben jelentkezik, mint azt jól beállított paraméterek mellett tapasztalnánk.

4.5. Fázis vokódolás

A második algoritmus a fázis vokóder, melynek részletes leírása a 2.2. szakaszban olvasható. A számítási igény vizsgálata után itt a monó feldolgozás tűnt megvalósíthatónak – bár a bufferrendszer sztereó megvalósításra lett elkészítve –, mely kétféle módon is implementálva lett. Az elsőnél a generátorbank szinuszgenerátorokból áll, de ez az út nem járható ennél a kártyánál, mert a számítási kapacitás igénye annyira nagy, hogy a teljes generátorbank 20%-át sikerült csak bekapcsolni. Ezután inverz Fourier-transzformációval is megvalósításra került, mely már online számolható volt.

4.5.1. A működési elv

A működési elv nagyon hasonló a 4.4. részben leírthoz. Itt is az IT rutin álltal hívott `ItProcessData()` gyűjti a mintákat minden interruptban, és tárolja el `iInBufferL` és `iInBufferR` cirkuláris bufferekbe, majd ha a blokkos feldolgozáshoz elegendő minta összegyűlt, akkor `bProcessFlag`-et bebillentve lehetőséget ad a `ProcessData()` rutin futásának.

Itt is a `ProcessData()` végzi a lényeges számítást. Először megszorozza a cirkuláris `iInBufferL` bufferből származó jelet egy ablakfüggvénnyel, majd az `fftinput` temporális bufferbe tárolja, ahonnan az $\mathcal{FFT}$ rutin dolgozik. A Fourier-transzformáció után a kapott komplex Fourier-együtthatók az `fftcoeff`-be kerülnek, majd kinyerve az amplitúdó információt az `fftampNL` pointer által mutatott bufferbe, és a fázis információt az `fftphsNL` pointer által mutatott bufferbe, lezárul az analízis része a folyamatnak.

Ezután a mostani (`fftphsNL`), és az előző (`fftphsLL`) fázisinformáció segítségével kiszámolódik az aktuális fázisállapot (`iPhsL`). A kinyert amplitúdóval és a kiszámolt fázissal újra komplex együtthatókat képezünk, majd ezeket inverzi Fourier-transzformáljuk, és az eredményt `fftgen` ideglenes komplex bufferben tároljuk, majd egy ablakfüggvénnyel megszorozva átkerül az `iOutBufferL`-be, ahol már time stretchelt jelről beszélhetünk.

Itt is át kell esnie a jelnek egy újramintavételezésen – mint az előbb –, hogy online működőképes legyen az algoritmus. Lineáris interpolációt használva, pitch shiftelléstől függően vagy az interpolálás előtt vagy után anti-alias szűrő van beiktatva, mint ahogy azt a 4.6. ábrán már láthattuk.

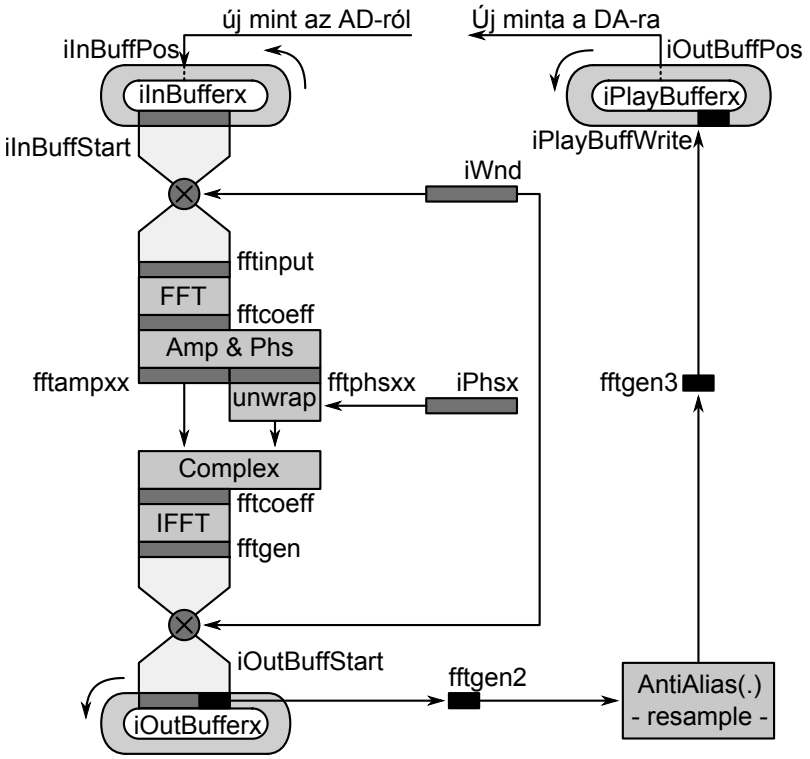
Az újramintavételezés után a jel az `iPlayBufferL`-be kerül, és az `iOutBufferL` megfelelő területe ki lesz nullázva. Az `ItProcessData()` a mintáit az `iPlayBufferL`-ből veszi folyamatosan, és küldi ki a DA-ra. Az implementáció könnyebb megértését segíti a 4.8. ábrán szereplő blokkdiagram.

4.5.2. Paraméterek beállítása

Mint az előbb a könnyebb használat érdekéért itt is egy MATLAB szkript, pár kérdés feltétele utána kigenerálja a C/C++ header fájlt, amivel együtt lefordítva a kódot, feltölthetjük a DSP-re. A paraméterekhez itt ilyenek tartoznak, hogy fázisléptetés, vagy anti-alias szűrőegyütthatók.

4.5.3. Megvalósítás korlátai

Az implementációnak memóriagondjai nem voltak, hisz kevesebb és kisebb bufferek kellettek, ezért minden elfért belső memóriában, de a számítási igénye annyival nagyobb, hogy ez megakadályozta az ablakok kellő átlapolódását, és az általában jól használható 75%-os átlapolódás helyett $50\% - 60\%$ körüli értékre lehet csak állítani, ebből következik, hogy hajlamosabb amplitúdómodulálni a jelet. A memóriaméretből származó korlát az $\mathcal{FFT}$ méretét limitálja 1024 pontra, ami már elfogadható minőséget adhatna. A legnagyobb probléma viszont magából a számábrázolásból származik. Mivel a processzor 16 bites, így az $\mathcal{FFT}$ és $\mathcal{FFT}^{-1}$ rutinok is 16 biten



4.8. ábra. Algoritmus blokkvázlata

számolnak, melyből következően az $\mathcal{FFT}$ $\mathcal{FFT}^{-1}$ páros on áteső hang erősen kvantálási zajjal van terhelve, így az audió minősége hagy kivetni valót maga után, de ezt a problémát ezen az architektúrán nem lehet kiküszöbölni. Megoldást az jelentene, ha lebegőpontos kártyára is implementálnánk az algoritmust, ahol már nem jelentkezhetnének ezek a zavaró hatások.

4.6. Wavelet-transzformációs fázis vokódolás

Mivel a wavelet-transzformációs algoritmust a kártyán csak konvolúciók segítségével lehetséges megvalósítani – hiszen nincs előre megírt fast wavelet-transform rutin –, a pár ezer konvolúciót a processzor már nem tudja időben végrehajtani, és ezért az algoritmus nem lenne képes online működni. De ha még képes is lenne, ugyanúgy elszenvedné a jel a kvantálási hibákat, mint a fázis vokódernél. Ezenfelül a sok szűrő impulzusválasza nem férne el a belső memóriában, és kétséges lenne, hogy a külső memóriában elég gyors eléréssel lehetne hozzáférni. Ilyen megfontolások alapján úgy döntöttem, hogy nincs értelme ezt az algoritmust megvalósítani ezen a DSP kártyán.

5. fejezet

Eredmények értékelése

„Nem vagyunk mások, csupán gondolataink eredménye.”

Buddha

A szimulációs vizsgálataim során különböző hanganyagokkal teszteltem az algoritmusokat, melyeket a feldolgozás eredményeivel, valamint a DSP kártyán keresztül felvett hanganyaggal megtalálhatók a CD mellékleten. A jobb eligazodás érdekében minden könyvtár tartalmaz egy szövegfájlt, melyben az algoritmus futtatási paraméteri vannak rögzítve, ezzel nyomonkövethetővé és reprodukálhatóvá téve az eredményeket. A hanganyagokat az 1.3 szakaszban részletezett algoritmusokkal összehasonlítottam, és az eredményeket a függelék táblázataiban közlöm.

5.1. Tranziensben gazdag jelek

Az olyan jeleknél, melyek tranziensekben nagyon gazdagok a fázis vokóder használata kevésbé ajánlott, ugyanis az algoritmus okozta tranziensszétkenődés már nagyon kis sebességállítási faktornál jelentkezik. Ez megfigyelhető például a `001_CosmicGate.wav`, a `001_Animal1.wav`, és `001_MustangNismo.wav` hangfájlokon, ahol a dobok elejét „elnyeli” az algoritmus. A hatás már 5%-os sebességállításnál is jelentkezik. Ilyen esetben érdemesebb az időtartományi szegmentálást használni, mert az jobban megtartja az tranziens jeleket, valamint kis sebességállításnál, még nem jelentkezik az információ duplázás. Erre a `001_Animal1.wav` példa, mely 0.95-ös sebességállítási faktor mellett jó szemléltetés. Ezenkívül a frekvenciaszelektív AM moduláció sem érzékelhető annyira, hiszen a tranziens részek miatt a jelben gyorsan változnak a frekvenciakomponensek, és a fülünk érzéketlenebb lesz a moduláció hullámzásával szemben. Erre példa 0.9-es faktor mellett a `005_VocalMale1.wav` hangfájl.

5.1.1. Zene

A szegmentálást használva, zenénél érdemes 8000 minta körüli ablakfüggvényt alkalmazni, 50%-os átlapolódással, mint ahogy ezt a 0.8-as sebességállítási faktornál

tettem. Egy-két százalékos sebességállítás esetén növelni kell a méretet, kiküszöbölve ezzel, az ablakok kevés mintával való eltolásából származó fésűszűrő hatást, ami ilyenkor erősebben, és zavaróbban jelentkezik, mint nagyobb állításoknál. Ez figyelhető meg a 0.95-ös és az 1.05-ös sebességállítási faktoroknál.

Természetesen ez minden zenére nem igaz, hiszen vannak olyan zenék, melyekben sok a hosszan kitartott énekhang, vagy a folytonosan szóló hangok. Ilyen a mellékelt hanganyagban például a `001_JulieLondon.wav` fájl. Ezeknél a 8000 mintás ablakméret kevés, mert a modulációs hatás sokkal erőteljesebben érzékelhető a kitartott hangokon. Itt 16000 minta körüli vagy akár még nagyobb ablakokat kell használnunk. Az ablakfüggvényt is érdemes változtatni. A módosított-Hanning ablak általában jobb eredményekhez vezet. Erre példa a 0.9-es sebességállítási faktorú `001_AgnesVanilla1.wav` fájl.

A nagy ablakmérettel persze vigyázni kell, mert még ha sok is a kitartott hang a zenében, azért inkább tranziens jelnek mondható. Így a meg kell találni a kompromisszumos ablakméretet, és ablakfüggvényt, amellyel minimalizálható a két nem kívánt hatás, ami nem mindig bizonyul egyszerű feladatnak.

Ha nagyobb sebességállítást akarunk végrehajtani, akkor az időtartománybeli szegmentálás már lehet, hogy túl nagy információduplázást eredményez, melynek következtében tapasztalható a tranzienseken visszhang jellegű hatás. Ez figyelhető meg a `001_CosmicGate.wav` fájlban 0.8-as és 1.2-es, illetve 1.3-as sebességállítási faktor mellett. Ilyen esetben a fázis vokóder jobb választásnak bizonyul, mert a tranzienseket némiképp csökkentő eljárás kellemesebb lehet a fülnek mint visszhangosodás.

Paraméterei legyenek körülbelül: 1000–2000 pontos $\mathcal{FFT}$, 50% átlapolódás. Kisebb pontszám esetén a mély frekvenciák felbontása nem kielégítő, nagyobb pontszámnál pedig a tranziensek „szétkenése” erősebben jelentkezik.

Ha a nagy sebességállításnál nagyon zavaró a tranziens jelek csökkenése, akkor érdemes lehet használni a wavelet alapú fázisvokódert, mely jobban tudja követni, és megkímélni a tranzienseket. Kis sebességállításnál is használható, de mivel jóval nagyobb a számítási kapacitása mint az időtartománybeli szegmentálásnak, vagy a fázis vokódernek, így csak olyan helyeken érdemes használni, ahol minőség mindennél fontosabb, ahol a másik két algoritmust nem lehet úgy beállítani, hogy az elfogadható eredményt adjon.

5.1.2. Ének és beszédjel

Ének és beszédjelek tartozhatnak a tranziensben gazdag jelekhez, ha kellően gyorsan változnak. Ilyen esetekben is igaz amit az előző részben említettem, miszerint kis sebességállításnál érdemesebb az időtartománybeli szegmentálást, nagyobb sebességállításnál a fázis vokódolást használni. Erra példa, a `001_Male_Ady1.wav` fájl 0.8-as sebességállítási faktor mellett, ami fázis vokóder algoritmussal hangzik jobban, mert az időtartománybeli szegmentációnál már zavaró modulációt lehet hallani, míg 0.95-ös sebességállítási faktor használva inkább az időtartománybeli szegmentációt preferálhatjuk, mert a fázis vokódernél a tranziens jelek jobban elvesznek.

Olyan ének és beszédjeleknél, ahol a frekvenciakomponensek nem változnak nagy tartományban, vagy ha változnak is, az csak rövid időre történik, akkor lehetőség

nyílik még az időtartománybeli szegmentálást olyan paraméterekkel beállítani, hogy az amplitúdó moduláció a jel komponenseit a lehető legkevésbé érje, így a zavaró hatás, ami fülünknek kellemetlen, kikerülhető, csökkenthető.

5.2. Tranziensben szegény jelek

Tranziensben szegény jeleknél a fázis vokódert érdemesebb használni, hiszen ezen jeleknél az időtartománybeli szegmentálás okozta frekvenciafüggő amplitúdó moduláció sokkal erőteljesebben érzékelhető. Erre példa a `001_Sin440Hz.wav` fájl 0.95-ös sebességállítási faktor mellett, ahol megfigyelhető, hogy az időtartománybeli szegmentálás a szinuszjelet zavaróan modulálja, míg a fázis vokódernél ez a hiba nem jelentkezik.

Mint említettem van ugyan lehetőség az időtartománybeli szegmentálást úgy beállítani, hogy annak nem kívánt hatásai a jel komponenseit elkerüljék, még ha nem is teljesen, de legalább kevésbé befolyásolják, azonban ez egy kellően komplex hangnál, vagy egy elég sok szólamú darabnál, a felharmonikusok száma, és a frekvencia komponensek elhelyezkedése miatt nem lehetséges ilyen beállítás.

6. fejezet

Összefoglaló

„Nem lehet egyszerre látni a dolgok kezdetét s végét.”

Quintus Horatius Flaccus

Diplomatervemben igyekeztem átfogó képet adni a time stretch és pitch shift algoritmusokról, kezdve egy rövid történelmi áttekintéssel, melyben az analóg hanglemez és magnószalag világától, eljutva a digitális jelfeldolgozó processzorokig és számítógépekig, végigvezettem, mikor és hol volt, illetve lehet szükség time stretch és pitch shift algoritmusokra, milyen módon tudja őket használni a stúdiótechnika vagy a szórakoztatóipar.

Ezután a sebességállító algoritmusok bemutatásáról volt szól részletesen kifejtve az időtartománybeli szegmentálás, a fázis vokódolás technikáját, betekintést adva a wavelet-transzformáció alkalmazási lehetőségeibe, valamint kitérve néhány érdekességre, és különleges felhasználási módokra

Majd a pitch shift és a time stretch algoritmusok rokonságára mutattam rá, és kifejtettem, hogyan lehet a time stretch algoritmusokat módosítani úgy, hogy azok pitch shift algoritmusokként működjenek.

Ezt követően a bemutatott módszerek MATLAB implementációin futtatott szimulációs eredményeim felsorolása következett, ahol összehasonlító eljárások segítségével igyekeztem rangsorolni az algoritmusokat. Megállapítottam, hogy az általam vizsgált összes algoritmus alkalmas time stretch és pitch shift megvalósítására. Minőség szempontjából ez a következők szerint részletezhető: Lassításra a leginkább a fázis vokóder alkalmazható – a wavelet-transzformáció alapú is –, gyorsítás esetén a legjobb módszer függ a feldolgozandó jeltől. Zenei anyagok esetén, amelyekben kevés tranziens szakasz fordul elő, ugyancsak a fázis vokódolás a legjobb megoldás, míg beszédjel, illetve sok tranziens részt tartalmazó jelek esetén, az időtartománybeli szegmentálás, és a wavelet-transzformáció alapú algoritmus alkalmazható. A módszerek jellegéből adódóan ugyanez a minősítés adható pitch shift algoritmusok esetén is.

A szimulációk után leírtam a ADSP-BF537 EZ-KIT Lite kártya felépítését, tulajdonságait, részleteztem az implementációimat, kifejtve mindegyiknél a működést,

kitérve előnyeikre, feltüntetve az algoritmusok korlátait, és az általuk produkált hibajelenségeket.

A jövőben szeretném ezen algoritmusokat programként implementálni, finomítani a működésüket, javítva használhatóságukat. Különböző jelfelismerő és adaptív eljárásokkal kiegészítve egy univerzális szoftvereszközt létrehozni.

Irodalomjegyzék

[1] Udo Zölzer. *Digital Audio Effects.* John Wiley & Sons Ltd, 2002. ISBN 0-471-49078-4.

[2] A.N. Akansu; M.J.T. Smith, editor. *Subband and Wavelet Transforms.* Kluwer Academic Publishers, 1996. ISBN 0-792-39645-6.

[3] Stéphane Mallat. *A Wavelet Tour of Signal Processing, Second Edition (Wavelet Analysis & Its Applications).* Academic Press, September 1999. ISBN 0-124-66606-X.

[4] Charles K. Chui. *An Introduction to Wavelets.* Academic Press, 1992. Library of Congress Catalog Card Number: 91-58831.

[5] Fritz Keinert. *Wavelets and Multiwavelets.* Chapman & HallCRC, 2004. ISBN 1-58488-304-9.

[6] O.C.; Wong P.H.W. Wong, J.W.C.; Au. „Fast time scale modification using envelope-matching technique(em-tsm).” In „Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium,” volume 5, pages 550 – 553. 31 May - 3 Jun 1998.

[7] O.C.; Wong J.W.C.; Lau W.H.B. Wong, P.H.W.; Au. „On improving the intelligibility of synchronized over-lap-and-add(sola) at low tsm factor.” In „Speech and Image Technologies for Computing and Telecommunications., Proceedings of IEEE,” volume 2, pages 487 – 490. 2-4 Dec 1997.

[8] Norbert Schnell; Geoffroy Peeters; Serge Lemouton; Philippe Manoury; Xavier Rodet. „Synthesizing a choir in real-time using pitch synchronous overlap add (psola).” 2009.

[9] M. Verhelst, W.; Roelands. „An overlap-add technique based on waveform similarity (wsola) for high quality time-scale modification of speech.” In „Acoustics, Speech, and Signal Processing 1993 IEEE International Conference,” volume 2, pages 554 – 557. 27-30 Apr 1993.

[10] Laroche J.; Dolson M. „New phase-vocoder techniques for pitch-shifting, harmonizing andother exotic effects.” In „Applications of Signal Processing to Audio and Acoustics, 1999 IEEE Workshop,” pages 91 – 94. 10.1109/ASPAA.1999.810857, 1999.

[11] Axel Röbel. „A new approach to transient processing in the phase vocoder." In „Proc. of the 6th Int. Conference on Digital Audio Effects (DAFx-03)," 8-11 Sept 2003.

[12] Jean Laroche; Mark Dolson. „Improved phase vocoder time-scale modification of audio." In „Speech and Audio Processing, IEEE Transactions," volume 7, pages 323 – 332. May 1999.

[13] Bonada J. „Automatic technique in frequency domain for near-lossless time-scale modification of audio." In „International Computer Music Conference," pages 396 – 399. 2000.

[14] De Gersem P.; De Moor B.; Moonen M. „Applications of the continuous wavelet transform in the processing of musical signals." In „Digital Signal Processing Proceedings," volume 2, pages 563 – 566. 2-4 Jul 1997.

[15] Geoff Shew. „A continuous wavelet transform based pitch shifting method for audio signals." Technical report.

[16] Anshul Saxena; Sanjay Kumar; Rajesh Meena. „Applications of the wavelet transform in the processing of musical signals." In „EE678 Wavelets Application Assignment," .

[17] Udo Zölzer. *Digital Audio Signal Processing.* John Wiley & Sons Ltd, 1997. ISBN 0-471-97226-6.

[18] C.; Mühlig H.; Szemengyajev K. A. Bronstejn, I. N.; Musiol. *Matematikai kézikönyv.* Typotex Kft., 1999. ISBN 978-963-9326-53-8.

A. függelék

MATLAB szimulációk

A.1. Időtartománybeli szegmentálás

A.1.1. A szegmentálás MATLAB függvénye

```
function Y = OLA(X,wnd,n1,n2)
%=========================================================%
% OLA Megvalositas                                        %
%                                                         %
% X      - Bemeneti vektor                                %
% wnd    - Ablak fuggveny                                 %
% n1     - Bemeneti ugras                                 %
% n2     - Kimeneti ugras                                 %
%                                                         %
% Y      - Kimeneti vektor                                %
%=========================================================%
Y = zeros(ceil(n2/n1*length(X)),1);
wndsize = length(wnd);
D = zeros(ceil(wndsize/n2)+1,2)-1;

disp(sprintf('Length Factor: %d',n2/n1));

wndcnt = 0;
for k=1:length(Y)-wndsize

    % Uj ablak kezdodik
    if (mod(k-1,n2)==0)
        l = 1;
        while(D(l,1)~=-1) l=l+1; end
        D(l,1) = wndcnt;
        D(l,2) = 0;
        wndcnt = wndcnt + 1;
    end

    % Osszegzes
    V = 0;
    for l=1:length(D)
        if (D(l,1)~=-1)
            Y(k)=Y(k)+X(D(l,1)*n1+D(l,2)+1)*wnd(D(l,2)+1);
            V = V + wnd(D(l,2)+1);
            D(l,2)=D(l,2)+1;
            % Ablak vege
            if (D(l,2)+2>wndsize)
                D(l,1) = -1;
                D(l,2) = -1;
            end
        end
    end
```

```
    % Volume Normalizalas
    if (V~=0) Y(k) = Y(k)./V; end
end
```

A.1.2. Egysávos szegmentálás MATLAB szkriptje

```
clc;
clear all;
format long;
format compact;

speed = 0.8;
[X,Fs,bit] = wavread('Input.wav');

wnd = hann(1024*1);
wnd = wnd(1:round(length(wnd)/2));
wnd = [wnd; ones(1024*5,1); 1-wnd];
n1  = round(length(wnd)*(1-0.7));
n2  = round(speed*n1);
Y   = OLA(X,wnd,n1,n2);

wavwrite(Y/max(Y),Fs,bit,'Output.wav');
```

A.1.3. Többsávos szegmentálás MATLAB szkriptje

```
clc;
clear all;
format long;
format compact;

speed = 1.5;
N     = 512;
win   = hann(N+1);
flag  = 'scale';
[X,Fs,bit] = wavread('Input.wav');

% Mely frekvenciasav
Fc   = 500;
filt1 = fir1(N, Fc/(Fs/2), 'low', win, flag);
X1   = filter(filt1,1,X);
wnd = hann(1024*5);
wnd = wnd(1:round(length(wnd)/2));
wnd = [wnd; ones(1024*10,1); 1-wnd];
n1  = round(length(wnd)*0.5);
n2  = round(speed*n1);
Y1  = OLA(X1,wnd,n1,n2);

% Kozep frekvenciasav
Fc1 = 500;
Fc2 = 5000;
filt2 = fir1(N, [Fc1 Fc2]/(Fs/2), 'bandpass', win, flag);
X2  = filter(filt2,1,X);
wnd = hann(1024*3);
wnd = wnd(1:round(length(wnd)/2));
wnd = [wnd; ones(1024*5,1); 1-wnd];
n1  = round(length(wnd)*0.5);
n2  = round(speed*n1);
Y2  = OLA(X2,wnd,n1,n2);

% Magas frekvenciasav
Fc   = 5000;
filt3 = fir1(N, Fc/(Fs/2), 'high', win, flag);
X3   = filter(filt3,1,X);
wnd = hann(1024*1);
wnd = wnd(1:round(length(wnd)/2));
wnd = [wnd; ones(1024*2,1); 1-wnd];
```

```
n1  = round(length(wnd)*0.5);
n2  = round(speed*n1);
Y3  = OLA(X3,wnd,n1,n2);

% Elteresek lehetnek a vektorok kozott, es ezt ki kell egyenliteni
Y = Y1(1:min([length(Y1) length(Y2) length(Y3)])) ...
  + Y2(1:min([length(Y1) length(Y2) length(Y3)])) ...
  + Y3(1:min([length(Y1) length(Y2) length(Y3)]));

wavwrite(Y/max(Y),Fs,bit,'Output.wav');
```

A.1.4. Wavelet szűrőbankos szegmentálás MATLAB szkriptje

```
clc;
clear all;
format long;
format compact;
%warning off;

% nyujtasi arany beallitasa
speed = 0.7;

[X,Fs,bit] = wavread('electro.wav');
Sum = zeros(2*length(X),1);

% 4 szintu struktura Daubechies wavelettel
[C,L] = wavedec(X,4,'db8');

wnd = hann(1024*5);
for k=1:length(L)-2
    n1 = round(length(wnd)*k./8);
    n2 = round(n1*speed);

    % Sav informacio kinyerese
    XX = upcoef('d',detcoef(C,L,k),'db8',k);
    YY = OLA(XX,wnd,n1,n2);
    Sum(1:length(YY)) = Sum(1:length(YY)) + YY;
end

XX = upcoef('a',appcoef(C,L,'db8',k),'db8',k);
YY = OLA(XX,wnd,n1,n2);
Sum(1:length(YY)) = Sum(1:length(YY)) + YY;

Sum = Sum./max(Sum);
wavwrite(Sum,Fs,bit,'electro_Wavelet_1.5.wav');
```

A.2. Fázis vokódolás szinusz generátorbankkal

A.2.1. A fázis vokóder amplitúdó interpoláló függvénye

```
function Ampd = PVIntAmp(Amp0,Amp,n2)
%=========================================================%
% Amplitudo interpolalas                                  %
%                                                         %
% Amp0  - Regi amplitudo ertek                            %
% Amp   - Mostani amplitudo ertek                         %
% n2    - Kimeneti ablak hossza                           %
%                                                         %
% Ampd  - Interpolalt amp vektor                          %
%=========================================================%
Ampd = (Amp-Amp0)/n2;
```

A.2.2. A fázis vokóder fázis interpoláló függvénye

```
function Phsd = PVIntPhs(Phs0,Phs,n1,n2,Freq)
%=========================================================%
% Fazis interpolalas                                      %
%                                                         %
% Phs0  - Regi fazis ertek                                %
% Phs   - Mostani fazis ertek                             %
% n1    - Bemeneti lepeskoz                               %
% n2    - Kimeneti lepeskoz                               %
% Freq  - Frekvencia vektor (f/fs)                        %
%                                                         %
% Phsd  - A generatorbank meghajtasara a fazis differencia %
%=========================================================%
WndPhs = 2*pi*Freq*n1;
Phsd = (WndPhs + PVPhsWrap(Phs - Phs0 - WndPhs))./n1;
```

A.2.3. A fázis vokóder fázis wrappoló függvénye

```
function Y = PVPhsWrap(X)
%=========================================================%
% Fazis-Wrap                                              %
%                                                         %
% X- Meneti fazis                                         %
%                                                         %
% Y- Kineti fazis (wrappolva van -pi es pi koze)          %
%=========================================================%
Y = mod(X + pi,2*pi)-pi;
```

A.2.4. A fázis vokóder fő függvénye

```
function Y = PhaseVocoder(X,wnd,n1,n2)
%=========================================================%
% PhaseVocoder - szinusz generatorbankos                  %
%                                                         %
% X     - Bemeneti vektor                                 %
% wnd   - Ablak fuggveny                                  %
% n1    - Bemeneti ugras                                  %
% n2    - Kimeneti ugras                                  %
%                                                         %
% Y     - Kimeneti sebessegallitott jel                   %
%=========================================================%
Y = zeros(ceil(n2/n1*length(X)),1);
wndsize = length(wnd);
disp(sprintf('Length Factor: %d',n2/n1));

Freq = [0:floor(wndsize/2)-1]'./wndsize;
% ================================================ %
% Frekvenci beavatkozasok (pl harmonizalas)        %
% Freq = 0.995*Freq+0.005;                         %
% ================================================ %

Amp0 = zeros(floor(wndsize/2),1);
Phs0 = zeros(floor(wndsize/2),1);
Phsh = zeros(floor(wndsize/2),1);
k = 1;
m = 1;
while (k<length(X)-length(wnd))
  F = fft(X(k:k+wndsize-1).*wnd);
  Amp = abs(F(1:floor(wndsize/2)));

  % ============================================================== %
  % Amplitudo beavatkozasok (pl zajszures)                         %
  % Amp = (Amp.*(1+3.*[1:floor(wndsize/2)]'./floor(wndsize/2))).^1.6; %
  % ============================================================== %

  Phs = angle(F(1:floor(wndsize/2)));

  Ampd = PVIntAmp(Amp0,Amp,n2);
```

```
    Amph = Amp0;

    Phsd = PVIntPhs(Phs0,Phs,n1,n2,Freq);
    for l=1:n2
        Amph = Amph+Ampd;
        Phsh = Phsh+Phsd;
        Gen(l) = Amph'*cos(Phsh);
    end
    Y(m:m+n2-1) = Gen';

    k = k+n1;
    m = m+n2;
    Amp0 = Amp;
    Phs0 = Phs;
    Phsh = PVPhsWrap(Phsh);
end
```

A.2.5. A fázis vokóder MATLAB szkriptje

```
clc;
clear all;
format long;
format compact;

[X,Fs,bit] = wavread('Input.wav');
X = X(1:100000);

speed = 2;
wnd = hann(1024*2);
n1  = 256;
n2  = round(n1*speed);

tic;
Y = PhaseVocoder(X,wnd,n1,n2);
toc;

wavwrite(Y./max(Y),Fs,bit,'Output.wav');
```

A.3. Fázis vokódolás $\mathcal{FFT}^{-1}$ generátorbankkal

A.3.1. A fázis vokóder fő függvénye

```
function Y = PhaseVocoderIFFT(X,wnd,n1,n2)
%=========================================================%
% Phase Vocoder IFFT felhasznalasaval                     %
%                                                         %
% X   - Bejovo mintak                                     %
% wnd - Ablak merete                                      %
% n1  - Bemeneti hopp                                     %
% n2  - Kimeneti hopp                                     %
%                                                         %
% Y   - Kiementi mintak                                   %
%=========================================================%
spd = n2/n1;
Y = zeros(round(length(X)*spd),1);

omega   = 2*pi*[0:wnd-1]'/wnd;
ffts    = zeros(wnd,1);
fftl    = zeros(wnd,1);
fftss   = zeros(wnd,1);
phs     = zeros(wnd,1);

k = 1;
while (k-1)*n1+wnd<length(X) && (k-1)*n2+wnd<length(Y)
    ffts = fft(fftshift(X((k-1)*n1+1:(k-1)*n1+wnd).*hann(wnd)));
```

```
        dphs = omega.*n1 + PVPhsWrap(angle(ffts)-angle(fftl)-n1.*omega);
        phs = PVPhsWrap(phs + dphs*(n2/n1));
        fftss = abs(ffts).*exp(j*phs);

        Y((k-1)*n2+1:(k-1)*n2+wnd) = Y((k-1)*n2+1:(k-1)*n2+wnd) + ...
                                     real(fftshift(ifft(fftss)).*hann(wnd));
        fftl = ffts;
        k = k+1;
    end

    Y = Y./max(Y);
```

A.3.2. A fázis vokóder MATLAB szkriptje

```
    clc;
    clear all;
    format long;
    format compact;

    wnd = 2048;
    spd = 1.3;
    n1 = wnd/4;
    n2 = round(spd*n1);

    [X Fs Bits] = wavread('In.wav');
    X = X(:,1);

    Y = PhaseVocoderIFFT(X,wnd,n1,n2);

    wavwrite(Y,Fs,Bits,'Out.wav');
```

A.4. Összehasonlító mérési eljárások

A.4.1. $\mathcal{FFT}$ alapú összehasonlítás függvénye

```
    function [FF F] = MeresFFT(X1,X2)
    %=========================================================%
    % FFT alapu hibaerteket szamol a ket jel kozott          %
    %                                                        %
    % X1 - Egyik jel (celszeru a rovidebbet)                 %
    % X2 - Masik jel                                         %
    %                                                        %
    % FF - Az elteres merteke                                %
    % F  - A normalt hibavektor                              %
    %=========================================================%
    if length(X2)>length(X1)
        X1 = [X1; zeros(length(X2)-length(X1),1)];
    else
        X2 = [X2; zeros(length(X1)-length(X2),1)];
    end

    F1 = abs(fft(X1));
    F2 = abs(fft(X2));

    F1 = F1./sum(F1);
    F2 = F2./sum(F2);

    F = (F1-F2).^2;
    FF = sum(F)./length(F1);
    F = F./max(F);
```

A.4.2. $\mathcal{STFT}$ alapú összehasonlítás függvénye

```
    function [SS S] = MeresSTFT(X1,X2,n,fs)
    %=========================================================%
```

```
% STFT alapu hibaerteket szamol a ket jel kozott %
%                                                %
% X1 - Egyik jel (celszeru a rovidebbet)         %
% X2 - Masik jel                                 %
% n  - Ablak hosszanak fele                      %
% fs - Mintaveteli frekvencia                    %
%                                                %
% SS - Az elteres merteke                        %
% S  - A normalt hibavektor                      %
%================================================%
[S1] = spectrogram(X1,2*n,n,2*n,fs);
[S2] = spectrogram(X2,2*n,n,2*n,fs);

ss1 = size(S1);
ss2 = size(S2);

xx1 = [0:1:ss1(2)-1]./(ss1(2)-1);
xx2 = [0:1:ss2(2)-1]./(ss2(2)-1);

S1 = spline(xx1,S1,xx2);

S1 = abs(S1./sum(sum(S1)));
S2 = abs(S2./sum(sum(S2)));

S = (S1-S2).^2;
SS = sum(sum(S))./(ss2(1)*ss2(2));

S = S./max(max(S));
```

A.4.3. wavelet-transzformációs összehasonlítás függvénye

```
function [WW W] = MeresWavelet(X1,X2)
%================================================%
% Wavelet alapu hibaerteket szamol a ket jel kozott %
%                                                %
% X1 - Egyik jel (celszeru a rovidebbet)         %
% X2 - Masik jel                                 %
%                                                %
% WW - Az elteres merteke                        %
% W  - A normalt hibavektor                      %
%================================================%
warning off;

S1 = WaveletGram(X1);
S2 = WaveletGram(X2);

ss1 = size(S1);
ss2 = size(S2);

xx1 = [0:1:ss1(2)-1]./(ss1(2)-1);
xx2 = [0:1:ss2(2)-1]./(ss2(2)-1);

S1 = spline(xx1,S1,xx2);

S1 = abs(S1./sum(sum(S1)));
S2 = abs(S2./sum(sum(S2)));

W = (S1-S2).^2;
WW = sum(sum(W))./(ss2(1)*ss2(2));

W = W./max(max(W));
```

A „wavletgram”-ot generáló függvény

```
function F = WaveletGram(X)
%================================================%
% WAVELET ATOM - Egy wavelet atomot keszit, megadott %
%                frekvencia es szelesseggel. Ha a %
```

```
% frekvencia es a szelesseg egymassal forditott viszonban  %
% van akkor "rendes" a wavelet.                             %
%                                                           %
% width - a wavelet szelessege idoben                       %
% freq  - a wavelet frekvenciaja                            %
%===========================================================%

% Ennel nagyobb felbontas is elerheto, de nagyon nagy a
% szamitasi igenye, ezert az ertekeleshez ezeket az adatokat
% hasznaltam.
width = 64;                         % Szurok szama
step  = round(64*width);            % Idobeli lepeskoz

% ==== LOG FREQ SCALE ======================================
freq = 2.*[0:1:width-1]'./(width-1)-1;
freq = 2.^(freq.*4);
freq = freq.*1000;

% ==== CALC BANDWITH =======================================
bandw = gradient(freq)./2;
bandw = 44100./bandw;
freq  = freq./44100;
for k=1:length(bandw)               % Limitalas
    if(bandw(k)>1024*8) bandw(k) = 1024*8; end
end

cnt = 1+floor((length(X)-max(bandw))/step);
F = zeros(width./2,cnt);
%wb = waitbar(0,'Analyze...');
for k=1:width
    Wavelet = WAtom(round(bandw(k)),freq(k));
    FF      = Analize(X,Wavelet,step,cnt);
    F(k,:)  = FF;
%   waitbar(k/width,wb);
end
%close(wb);
```

Egy adott wavelettel időtartományban vizsgálódó függvény

```
function F = Analize(X,W,step,cnt)
%===========================================================%
% ANALIZE - Egy adott wavelettel idoben mindenhol           %
%           analizalja a jelet, es vissza adja a komplex    %
%           vektorat.                                       %
%                                                           %
% X    - A bejovo jel.                                      %
% W    - A bejovo wavelet.                                  %
% step - Az x tengely menten a lepeskoz.                    %
% dist - Adott frekvenciahoz a lepeskoz fazisban.           %
%                                                           %
% F    - A kimeneti komplex vektor.                         %
%===========================================================%
F = zeros(cnt,1);

k = 1;
for z = 1:cnt
    F(z) = sum(X(k:k+length(W)-1).*W);
    k = k + step;
end
```

Wavelet generáló függvény

```
function Y = WAtom(width,freq)
%===========================================================%
% WAVELET ATOM - Egy wavelet atomot keszit, megadott        %
%                frekvencia es szelesseggel. Ha a           %
% frekvencia es a szelesseg egymassal forditott viszonban   %
% van akkor "rendes" a wavelet.                             %
```

```
%                                                                       %
% width - a wavelet szelessege idoben                                   %
% freq  - a wavelet frekvenciaja                                        %
%=======================================================================%

Y = [0:1:width-1]'-ceil(width./2);
Y = exp(i*2*pi*freq*Y);
Y = Y.*hann(width);
Y = 2*Y./sum(abs(Y));
%Y = Y-sum(Y)./width;     % OFFSET REMOVE
```

B. függelék

DSP C/C++ implementáció

B.1. Időtartománybeli szegmentálás megvalósítása

B.1.1. A `main(.)` függvény

```
/*****************************************************************************
* Frame Program for PS algorithms.
*
* Coded by: Robert Galambos
* Date: 2009.04.27.
*****************************************************************************/
#include "init.h"

int main(void)
{
  Init();
  while(1) if(bProcessFlag) ProcessData();
}
```

B.1.2. Az inicializálást végző header fájl

```
/*****************************************************************************
* Init header file.
*
* Coded by: Robert Galambos
* Date: 2009.04.27.
*****************************************************************************/
#ifndef INIT
#define INIT

#include <sys\exception.h>      // Interrup handling
#include <cdefBF537.h>          // MMR acces functions
#include <sysreg.h>             // Acces system registers [better than asm()]
#include <ccblkfn.h>            // Build in functions for system stuff
#include "Process.h"

/*****************************************************************************
* Function:     Init_Flags
* Description:  Configure PORTF flags to control ADC and DAC RESETs
*****************************************************************************/
void Init_Flags(void)
{
  *pPORTF_FER      = 0x0000;      // enables GPIO for PORTF
  *pPORTF_FER      = 0x0000;      // enables GPIO for PORTF (2nd time??)
  *pPORTFIO_DIR    = 0x1FC0;      // digital outputs (0001 1111 1100 0000)
  *pPORTFIO_INEN   = 0x003C;      // digital inputs  (0000 0000 0011 1100)
  *pPORTFIO_CLEAR  = 0x0FC0;      // clear the outputs
}
```

```
//#define delay 0xf00                  // It may need some delay
/*****************************************************************************
* Function:       Audio_Reset
* Description:    This function Resets the ADC and DAC.
*****************************************************************************/
void Audio_Reset(void)
{
//  for(int i=0;i<delay;i++){};    // wait for some time
  *pPORTFIO_SET = PF12;           // reset ADC & DAC
}

#define SLEN_24 0x0017  // SPORT0 word length
/*****************************************************************************
* Function:       Init_Sport0
* Description:    Configure Sport0 for I2S mode, to transmit/receive data
*                 to/from the ADC/DAC. Configure Sport for external clocks and
*                 frame syncs.
*****************************************************************************/
void Init_Sport0(void)
{
  // Sport0 receive configuration
  // External CLK, External Frame sync, MSB first, Active Low
  // 24-bit data, Secondary side enable, Stereo frame sync enable
  *pSPORT0_RCR1 = RFSR | RCKFE;
  *pSPORT0_RCR2 = SLEN_24 | RSFSE;

  // Sport0 transmit configuration
  // External CLK, External Frame sync, MSB first, Active Low
  // 24-bit data, Secondary side enable, Stereo frame sync enable
  *pSPORT0_TCR1 = TFSR | TCKFE;
  *pSPORT0_TCR2 = SLEN_24 | TSFSE;
}

#define FLOW_1 0x1000 // DMA flow mode
int iTxBuffer1[2];     // SPORT0 DMA transmit buffer
int iRxBuffer1[2];     // SPORT0 DMA receive buffer

/*****************************************************************************
* Function:       Init_DMA
* Description:    Initialize DMA3 in autobuffer mode to receive and DMA4 in
*                 autobuffer mode to transmit
*****************************************************************************/
void Init_DMA(void)
{
  // Configure DMA3 (default mapped to SPORT0 receive)
  // 32-bit transfers, Interrupt on completion, Autobuffer mode
  *pDMA3_CONFIG = WNR | WDSIZE_32 | DI_EN | FLOW_1;
  *pDMA3_START_ADDR = iRxBuffer1;          // Start address of data buffer
  *pDMA3_X_COUNT = 2;                      // DMA loop count
  *pDMA3_X_MODIFY = 4;                     // DMA loop address increment

  // Configure DMA4 (default mapped to SPORT0 transmit)
  // 32-bit transfers, Autobuffer mode
  *pDMA4_CONFIG = WDSIZE_32 | FLOW_1;
  *pDMA4_START_ADDR = iTxBuffer1;          // Start address of data buffer
  *pDMA4_X_COUNT = 2;                      // DMA loop count
  *pDMA4_X_MODIFY = 4;                     // DMA loop address increment
}


/*****************************************************************************
* Function:       Enable_DMA_Sport
* Description:    Enable DMA3, DMA4, Sport0 TX and Sport0 RX
*****************************************************************************/
void Enable_DMA_Sport0(void)
{
  *pDMA4_CONFIG = (*pDMA4_CONFIG | DMAEN);  // enable DMA4
  *pDMA3_CONFIG = (*pDMA3_CONFIG | DMAEN);  // enable DMA3
```

```
  *pSPORT0_TCR1 = (*pSPORT0_TCR1 | TSPEN);  // enable Sport0 TX
  *pSPORT0_RCR1 = (*pSPORT0_RCR1 | RSPEN);  // enable Sport0 RX
}

/*************************************************************************
* Function:     Sport0_RX_ISR
* Description:  This ISR is executed after a complete frame of input data
*               has been received. The new samples are stored in
*               iChannel0LeftIn and iChannel0RightIn.  Then the function
*               Process_Data() is called in which user code can be executed.
*               After that the processed values are copied from the
*               variables iChannel0LeftOut and iChannel0RightOut into the
*               DMA transmit buffer.
*************************************************************************/
EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
  *pDMA3_IRQ_STATUS = 0x0001;      // confirm interrupt handling

  int iLeft  = iRxBuffer1[0]<<8;
  int iRight = iRxBuffer1[1]<<8;

  ItProcessData(iLeft,iRight);

  iTxBuffer1[0] = iLeft>>8;
  iTxBuffer1[1] = iRight>>8;
}


/*************************************************************************
* Function:     Init_Interrupts
* Description:  Initialize Interrupt for Sport0 RX
*************************************************************************/
void Init_Interrupts(void)
{
  *pSIC_IAR0 = 0xFF2FFFFF;                  // Set Sport0 RX (DMA3)
                                            // interrupt priority to 2 (=IVG9)
  *pSIC_IAR1 = 0xFFFFFFFF;                  // Disable all other interrupts
  *pSIC_IAR2 = 0xFFFFFFFF;                  // Disable all other interrupts
  *pSIC_IAR3 = 0xFFFFFFFF;                  // Disable all other interrupts
  register_handler(ik_ivg9, Sport0_RX_ISR); // assign ISRs to interrupt vectors
  *pSIC_IMASK = 0x00000020;                 // Sport0 RX (DMA3) Enabled
}

/*************************************************************************
* Function:     Init_CoreClock
* Description:  Initialize the Core Clock to maximum speed
*************************************************************************/
void Init_CoreClock(void)
{
  int temp;

  *pPLL_CTL = 0x2800;                       // Set from 10x to 20x, for 500MHz
  *pPLL_DIV = 0x000A;                       // PLL_DIV need to be 2x

  // Needed to set the PLL correct
  temp = cli();
  idle();
  sti(temp);
}

void Init(void)
{
  Init_CoreClock();
  Init_Flags();
  Audio_Reset();
  Init_Sport0();
  Init_DMA();
  Init_Interrupts();
  Init_Filter();
```

```
165   Enable_DMA_Sport0();
}

#endif
```

B.1.3. A feldolgozást végző header fájl

```
0  /*****************************************************************************
   * Process Data header file.
   * Used, to code the algorithms
   *
   * Coded by: Robert Galambos
5  * Date: 2009.04.27.
   *****************************************************************************/
   #ifndef PROCESS
   #define PROCESS
   #include "MATLAB\Window.h"  // Window function in an ARRAY
10
   bool bProcessFlag = false;  // Flag to indicate ProcessData

   // INPUT BUFFER
   #define BUFF_SIZE (2*WND_SIZE)
15 /*segment("L1_data_a")*/ fract16 iInBufferL[BUFF_SIZE];
   /*segment("L1_data_b")*/ fract16 iInBufferR[BUFF_SIZE];
   int iInBuffPos = WND_SIZE;
   int iInBuffN1S = 0;
   int iInBuffN1C = 0;
20
   // OUTPUT BUFFER
   segment("L1_data_b") fract16 iOutBufferL[BUFF_SIZE];
   segment("L1_data_a") fract16 iOutBufferR[BUFF_SIZE];
   int iOutBuffN2S = 0;
25
   // PLAY BUFFER
   #define PLAY_BUFF_SIZE (2*BUFFN1)
   /*segment("L1_data_b")*/ fract16 iPlayBufferL[PLAY_BUFF_SIZE];
   /*segment("L1_data_b")*/ fract16 iPlayBufferR[PLAY_BUFF_SIZE];
30 int iPlayPos = 0;
   int iPlayWrite = 0;

   // ANTI-ALIAS FILTER
   #include<filter.h>
35 segment("L1_data_a") fir_state_fr16 FIRStateL;
   segment("L1_data_a") fir_state_fr16 FIRStateR;
   segment("L1_data_b") fract16 iFIRDelayL[FIR_CNT];
   segment("L1_data_b") fract16 iFIRDelayR[FIR_CNT];

40 /*****************************************************************************
   * Function:     ItProcessData
   * Description:  Gets data from ADC, and handles the buffer system.
   *****************************************************************************/
   void ItProcessData(int &iLeft,int &iRight)
45 {
     iInBufferL[iInBuffPos] = (iLeft>>16)/2;    // Save Data Left
     iInBufferR[iInBuffPos] = (iRight>>16)/2;   // Save Data Right
     iLeft  = iPlayBufferL[iPlayPos]<<16;       // Load Processed Data
     iRight = iPlayBufferR[iPlayPos]<<16;       // Load Processed Data
50   iPlayPos = (iPlayPos+1)%(PLAY_BUFF_SIZE); // Update BufferPos
     iInBuffPos = (iInBuffPos+1)%BUFF_SIZE;     // Update BufferPos
     iInBuffN1C++;                              // Update WindowStart
     if (iInBuffN1C>=BUFFN1)                    // If Enought New Samples
     {
55     iInBuffN1C = 0;                        // Restart Counting
       bProcessFlag = true;                   // Big Process
     }
   }

60 #include<math.h>
```

```
segment("L1_data_b") fract16 iBuffN2Temp[BUFFN2+1];
double dTemp;
int iTemp;
/*****************************************************************************
* Function:      AntiAlias
* Description:   AntiAlias Filter and ReSample.
*****************************************************************************/
void AntiAlias(fract16 iFrom[], int iFromL, fract16 iTo[], int iToL,
               fir_state_fr16 *FIRState)
{
  if(SPEED>1) fir_fr16(iFrom,iFrom,iFromL,FIRState);
  for(int i=0;i<iToL;i++)
  {
    dTemp = (i*(iFromL-1))/(iToL-1);
    iTemp = floor(dTemp);
    dTemp = dTemp-iTemp;
    iTo[i] = iFrom[iTemp] + (iFrom[iTemp+1]-iFrom[iTemp])*dTemp;
  }
  if(SPEED<1) fir_fr16(iTo,iTo,iToL,FIRState);
}

/*****************************************************************************
* Function:      Init_Filter
* Description:   Initialize FIR Filters.
*****************************************************************************/
void Init_Filter(void)
{
  fir_init(FIRStateL,iFIR,iFIRDelayL,FIR_CNT,0);
  fir_init(FIRStateR,iFIR,iFIRDelayR,FIR_CNT,0);
}

/*****************************************************************************
* Function:      ProcessData
* Description:   Process data in buffer system.
*****************************************************************************/
void ProcessData(void)
{
  // COPY to OUTPUTBUFFER
  for(int i=0;i<WND_SIZE;i++)
  {
    iOutBufferL[(iOutBuffN2S+i)%BUFF_SIZE] +=
      mult_fr1x16(iWnd[i],iInBufferL[(iInBuffN1S+i)%BUFF_SIZE]);
    iOutBufferR[(iOutBuffN2S+i)%BUFF_SIZE] +=
      mult_fr1x16(iWnd[i],iInBufferR[(iInBuffN1S+i)%BUFF_SIZE]);
  }

  for(int i=0;i<=BUFFN2;i++)
    iBuffN2Temp[i] = ((int)(iOutBufferL[(iOutBuffN2S+i)%BUFF_SIZE]*iVol[i]))>>15;
  AntiAlias(iBuffN2Temp,BUFFN2,&(iPlayBufferL[iPlayWrite]),BUFFN1,&FIRStateL);
  for(int i=0;i<=BUFFN2;i++)
    iBuffN2Temp[i] = ((int)(iOutBufferR[(iOutBuffN2S+i)%BUFF_SIZE]*iVol[i]))>>15;
  AntiAlias(iBuffN2Temp,BUFFN2,&(iPlayBufferR[iPlayWrite]),BUFFN1,&FIRStateR);

  // CLEAR BUFFERPART
  for(int i=0;i<BUFFN2;i++)
  {
    iOutBufferL[(iOutBuffN2S+i)%BUFF_SIZE] = 0;
    iOutBufferR[(iOutBuffN2S+i)%BUFF_SIZE] = 0;
  }

  iInBuffN1S = (iInBuffN1S+BUFFN1)%BUFF_SIZE;
  iOutBuffN2S = (iOutBuffN2S+BUFFN2)%BUFF_SIZE;
  iPlayWrite = (iPlayWrite+BUFFN1)%PLAY_BUFF_SIZE;

  bProcessFlag = false;
}

#endif
```

B.1.4. Paramétereket generáló MATLAB szkript

```
clc;
clear all;
format long;
format compact;
warning off;

disp('DSP_Frame_SOLA window.h generating enviroment.');
disp(' ');

wndsize = input('Select Window Size [4096]: ');
if isempty(wndsize)
    wndsize = 4096;
end

wndtype = menu( 'Choose Window Type', ...
                'Bartlett', ...
                'Barthannwin', ...
                'Blackman', ...
                'Blackmanharris', ...
                'Bohmanwin', ...
                'Chebwin', ...
                'Flattopwin', ...
                'Gausswin', ...
                'Hamming', ...
                'Hann', ...
                'Kaiser', ...
                'Nuttallwin', ...
                'Parzenwin', ...
                'Rectwin', ...
                'Tukeywin', ...
                'Triang',...
                'Home');

fnc = [@bartlett @barthannwin @blackman @blackmanharris @bohmanwin ...
       @chebwin @flattopwin @gausswin @hamming @hann @kaiser @nuttallwin ...
       @parzenwin @rectwin @tukeywin @triang];
if (wndtype<length(fnc))
    fun = fnc(wndtype);
    wndf = fun(wndsize);
else
    % ==== HOME WINDOW TYPE ====
    tt = 2;
    h = hann(wndsize/tt);
    wndf = [h(1:length(h)/2); ones(round(wndsize-length(h))-1,1); h(length(h)/2:end)];
end
wnd = round(((2^15)-1)*wndf);

n1 = input('Select Window N1 Overlap [0.5]: ');
if isempty(n1)
    n1 = 0.5;
end
n1 = round(n1*wndsize);

speed = input('Select Speed [0.6]: ');
if isempty(speed)
    speed = 0.6;
end
n2 = round(n1*speed);

disp(sprintf('\n\nWindow Size: %d\nSpeed: %d\nSize N1: %d\nSize N2: %d\n', ...
             wndsize,speed,n1,n2));

% ==== GENERATE ANTI-ALIAS FILTER ================================================
wbef = input('Before WPass freq [0.2] ');
if isempty(wbef)
    wbef = 0.2;
end
```

```
waft = input('After WPass freq [-0.1]');
if isempty(waft)
    waft = -0.1;
end

N = input('FIR Order [64]');
if isempty(N)
    N = 64;
end

if (speed>1)
    wfreq = 1/speed;
    bcoeff  = firpm(N, [0 (wfreq-wbef) (wfreq+waft) 1], [1 1 0 0]);
elseif (speed<1)
    wfreq = speed;
    bcoeff  = firpm(N, [0 (wfreq-wbef) (wfreq+waft) 1], [1 1 0 0]);
else
    bcoeff  = zeros(N+1,1);
end

% ==== SHOW WINDOW FUNCTION ================================================
figure(1);
plot(wnd);
title('Window Function');

% ==== CALC WINDOW OVERLAPPING =============================================
figure(2);
subplot(311)
wndsm = zeros(length(wndf)*4,1);
k = 1;
hold on;
while k+length(wndf)<length(wndsm)
    wndsm(k:k+length(wndf)-1) = wndsm(k:k+length(wndf)-1) + wndf;
    plot([k:k+length(wndf)-1],wndf);
    k = k+n2;
end
plot(wndsm);
hold off;
title('Sum of Window Functions');
subplot(312);
volcorr = wndsm(3*n2:4*n2);
plot(volcorr);
title('Overlap Volume');
volcorr = 1./volcorr;
subplot(313);
plot(volcorr);
title('Volume Correction');
volcorr = round(((2^15)-1)*volcorr);

% ==== SHOW ANTI ALIAS FILTER ==============================================
bcoeff = round(((2^15)-1)*bcoeff./sum(bcoeff));
fvtool(bcoeff,1);

% ==== GENERATE WINDOW.H FILE ==============================================
fid = fopen('Window.h','w');

fprintf(fid,'// MATLAB GENERATED FILE...\n');
fprintf(fid,'// Used for Window function\n');
fprintf(fid,'// Date:%s\n',date);
fprintf(fid,'#ifndef WINDOW\n');
fprintf(fid,'#define WINDOW\n\n');
fprintf(fid,'#define WND_SIZE %d\n',wndsize);

% EXPORT WINDOW ARRAY ------------------------------------------------------
fprintf(fid,'fract16 iWnd[WND_SIZE] = { ');
for k=1:length(wnd)-1
    fprintf(fid,'%d, ',wnd(k));
end
```

```
fprintf(fid,'%d };\n\n',wnd(length(wnd)));

% EXPORT OTHER DEFINES ------------------------------------------------------

fprintf(fid,'#define SPEED %f\n',speed);
fprintf(fid,'#define BUFFN1 %d\n',n1);
fprintf(fid,'#define BUFFN2 %d\n\n',n2);

% EXPORT VOLUME CORRECTION ARRAY --------------------------------------------
fprintf(fid,'int iVol[BUFFN2+1] = { ');
for k=1:length(volcorr)-1
    fprintf(fid,'%d, ',volcorr(k));
end
fprintf(fid,'%d };\n\n',volcorr(length(volcorr)));


% EXPORT ANTI-ALIAS FILTER ARRAY --------------------------------------------
% IMORTANT : segment("L1_data_a")
fprintf(fid,'#define FIR_CNT %d\n\n',length(bcoeff));
fprintf(fid,'segment("L1_data_a") fract16 iFIR[FIR_CNT] = { ');
for k=1:length(bcoeff)-1
    fprintf(fid,'%d, ',bcoeff(k));
end
fprintf(fid,'%d };\n\n',bcoeff(length(bcoeff)));

fprintf(fid,'#endif\n\n');
fclose(fid);

disp(' ');
disp('Data Saved...');
```

B.2. Fázis vokódolás megvalósítás $\mathcal{FFT}^{-1}$-vel

B.2.1. A `main(.)` függvény

```
/*****************************************************************************
* Frame Program for PS algorithms.
*
* Coded by: Robert Galambos
* Date: 2009.04.27.
*****************************************************************************/
#include "init.h"

int main(void)
{
  Init();
  while(1) if(bProcessFlag) ProcessData();
}
```

B.2.2. Az inicializálást végző header fájl

```
/*****************************************************************************
* Init header file.
*
* Coded by: Robert Galambos
* Date: 2009.04.27.
*****************************************************************************/
#ifndef INIT
#define INIT

#include <sys\exception.h>      // Interrup handling
#include <cdefBF537.h>          // MMR acces functions
#include <sysreg.h>             // Acces system registers [better than asm()]
#include <ccblkfn.h>            // Build in functions for system stuff
#include <window.h>
#include "Process.h"
```

```
15
/*****************************************************************************
* Function:       Init_Flags
* Description:    Configure PORTF flags to control ADC and DAC RESETs
*****************************************************************************/
20 void Init_Flags(void)
{
  *pPORTF_FER      = 0x0000;      // enables GPIO for PORTF
  *pPORTF_FER      = 0x0000;      // enables GPIO for PORTF (2nd time??)
  *pPORTFIO_DIR    = 0x1FC0;      // digital outputs (0001 1111 1100 0000)
25  *pPORTFIO_INEN   = 0x003C;      // digital inputs  (0000 0000 0011 1100)
  *pPORTFIO_CLEAR = 0x0FC0;      // clear the outputs
}

//#define delay 0xf00              // It may need some delay
30 /*****************************************************************************
* Function:       Audio_Reset
* Description:    This function Resets the ADC and DAC.
*****************************************************************************/
void Audio_Reset(void)
35 {
//  for(int i=0;i<delay;i++){};   // wait for some time?
  *pPORTFIO_SET = PF12;          // reset ADC & DAC
}

40 #define SLEN_24 0x0017  // SPORT0 word length
/*****************************************************************************
* Function:       Init_Sport0
* Description:    Configure Sport0 for I2S mode, to transmit/receive data
*                 to/from the ADC/DAC. Configure Sport for external clocks and
45 *                 frame syncs.
*****************************************************************************/
void Init_Sport0(void)
{
  // Sport0 receive configuration
50  // External CLK, External Frame sync, MSB first, Active Low
  // 24-bit data, Secondary side enable, Stereo frame sync enable
  *pSPORT0_RCR1 = RFSR | RCKFE;
  *pSPORT0_RCR2 = SLEN_24 | RSFSE;

55  // Sport0 transmit configuration
  // External CLK, External Frame sync, MSB first, Active Low
  // 24-bit data, Secondary side enable, Stereo frame sync enable
  *pSPORT0_TCR1 = TFSR | TCKFE;
  *pSPORT0_TCR2 = SLEN_24 | TSFSE;
60 }

#define FLOW_1 0x1000 // DMA flow mode
int iTxBuffer1[2];     // SPORT0 DMA transmit buffer
int iRxBuffer1[2];     // SPORT0 DMA receive buffer
65
/*****************************************************************************
* Function:       Init_DMA
* Description:    Initialize DMA3 in autobuffer mode to receive and DMA4 in
*                 autobuffer mode to transmit
70 *****************************************************************************/
void Init_DMA(void)
{
  // Configure DMA3 (default mapped to SPORT0 receive)
  // 32-bit transfers, Interrupt on completion, Autobuffer mode
75  *pDMA3_CONFIG = WNR | WDSIZE_32 | DI_EN | FLOW_1;
  *pDMA3_START_ADDR = iRxBuffer1;          // Start address of data buffer
  *pDMA3_X_COUNT = 2;                      // DMA loop count
  *pDMA3_X_MODIFY = 4;                     // DMA loop address increment

80  // Configure DMA4 (default mapped to SPORT0 transmit)
  // 32-bit transfers, Autobuffer mode
  *pDMA4_CONFIG = WDSIZE_32 | FLOW_1;
  *pDMA4_START_ADDR = iTxBuffer1;          // Start address of data buffer
```

```
    *pDMA4_X_COUNT = 2;                         // DMA loop count
    *pDMA4_X_MODIFY = 4;                        // DMA loop address increment
}


/*************************************************************************
* Function:       Enable_DMA_Sport
* Description:    Enable DMA3, DMA4, Sport0 TX and Sport0 RX
*************************************************************************/
void Enable_DMA_Sport0(void)
{
    *pDMA4_CONFIG = (*pDMA4_CONFIG | DMAEN);   // enable DMA4
    *pDMA3_CONFIG = (*pDMA3_CONFIG | DMAEN);   // enable DMA3
    *pSPORT0_TCR1 = (*pSPORT0_TCR1 | TSPEN);   // enable Sport0 TX
    *pSPORT0_RCR1 = (*pSPORT0_RCR1 | RSPEN);   // enable Sport0 RX
}

/*************************************************************************
* Function:       Sport0_RX_ISR
* Description:    This ISR is executed after a complete frame of input data
*                 has been received. The new samples are stored in
*                 iChannel0LeftIn and iChannel0RightIn.  Then the function
*                 Process_Data() is called in which user code can be executed.
*                 After that the processed values are copied from the
*                 variables iChannel0LeftOut and iChannel0RightOut into the
*                DMA transmit buffer.
*************************************************************************/
EX_INTERRUPT_HANDLER(Sport0_RX_ISR)
{
  int iLeft  = iRxBuffer1[0]<<8;
  int iRight = iRxBuffer1[1]<<8;

  ItProcessData(iLeft,iRight);

  iTxBuffer1[0] = iLeft>>8;
  iTxBuffer1[1] = iRight>>8;

  *pDMA3_IRQ_STATUS = 0x0001;         // confirm interrupt handling
}


/*************************************************************************
* Function:       Init_Interrupts
* Description:    Initialize Interrupt for Sport0 RX
*************************************************************************/
void Init_Interrupts(void)
{
  *pSIC_IAR0 = 0xFF2FFFFF;                    // Set Sport0 RX (DMA3)
                                              // interrupt priority to 2 (=IVG9)
  *pSIC_IAR1 = 0xFFFFFFFF;                    // Disable all other interrupts
  *pSIC_IAR2 = 0xFFFFFFFF;                    // Disable all other interrupts
  *pSIC_IAR3 = 0xFFFFFFFF;                    // Disable all other interrupts
  register_handler(ik_ivg9, Sport0_RX_ISR); // assign ISRs to interrupt vectors
  *pSIC_IMASK = 0x00000020;                   // Sport0 RX (DMA3) Enabled
}

/*************************************************************************
* Function:       Init_CoreClock
* Description:    Initialize the Core Clock to maximum speed
*************************************************************************/
void Init_CoreClock(void)
{
  int temp;

  *pPLL_CTL = 0x2800;                         // Set from 10x to 20x, for 500MHz
  *pPLL_DIV = 0x000A;                         // PLL_DIV need to be 2x

  // Needed to set the PLL correct
  temp = cli();
```

```
    idle();
    sti(temp);
}

/*****************************************************************************
* Function:     Init
* Description:  Initialize everything
*****************************************************************************/
void Init(void)
{
  Init_CoreClock();
  Init_Flags();
  Audio_Reset();
  Init_Sport0();
  Init_DMA();
  Init_Process();
  Init_Interrupts();
  Enable_DMA_Sport0();
}

#endif
```

B.2.3. A feldolgozást végző header fájl

```
/*****************************************************************************
* Process Data header file.
* Used, to code the algorithms
*
* Coded by: Robert Galambos
* Date: 2009.04.27.
*****************************************************************************/
#ifndef PROCESS
#define PROCESS
#include <filter.h>
#include <vector.h>
#include <stdlib.h>
#include <stdio.h>
#include "MATLAB\generator.h"

bool bProcessFlag = false;          // Flag to indicate ProcessData

// INPUT BUFFERING
#define BUFF_SIZE (2*WND_SIZE)

fract16 iInBufferL[BUFF_SIZE];
fract16 iInBufferR[BUFF_SIZE];
int iInBuffStart = 0;
int iInBuffPos = WND_SIZE;
int iInBuffData = 0;

// OUTPUT BUFFERING
#define OUT_BUFF_SIZE (2*WND_SIZE)
segment("L1_data_b") fract16 iOutBufferL[OUT_BUFF_SIZE];
segment("L1_data_b") fract16 iOutBufferR[OUT_BUFF_SIZE];
int iOutBuffStart = 0;
int iOutBuffPos = 1*WND_SIZE;

// PLAY BUFFER
#define PLAY_BUFF_SIZE (2*WND_SIZE)
/*segment("L1_data_a")*/ fract16 iPlayBufferL[PLAY_BUFF_SIZE];
/*segment("L1_data_a")*/ fract16 iPlayBufferR[PLAY_BUFF_SIZE];
int iPlayBuffPos = 0;
int iPlayBuffWrite = WND_SIZE;

// FFT DATA BUFFERS
segment("L1_data_b") fract16          fftinput[WND_SIZE];
segment("L1_data_b") complex_fract16  w[WND_SIZE];
segment("L1_data_a") complex_fract16  fftgen[WND_SIZE];
```

```
segment("L1_data_a") complex_fract16  fftcoeff[WND_SIZE];

segment("L1_data_b") fract16          fftgen2[N2];
segment("L1_data_a") fract16          fftgen3[N1];

fract16 wnd[WND_SIZE];
fract16 fftamp1L[WND_SIZE];
fract16 fftphs1L[WND_SIZE];
fract16 fftamp2L[WND_SIZE];
fract16 fftphs2L[WND_SIZE];
fract16 fftamp1R[WND_SIZE];
fract16 fftphs1R[WND_SIZE];
fract16 fftamp2R[WND_SIZE];
fract16 fftphs2R[WND_SIZE];

fract16 *fftampNL = fftamp1L;
fract16 *fftphsNL = fftphs1L;
fract16 *fftampLL = fftamp2L;
fract16 *fftphsLL = fftphs2L;
fract16 *fftampNR = fftamp1R;
fract16 *fftphsNR = fftphs1R;
fract16 *fftampLR = fftamp2R;
fract16 *fftphsLR = fftphs2R;

// FOLLOW THE PHASE
fract16 iPhsL[WND_SIZE];
fract16 iPhsR[WND_SIZE];

// ANTI-ALIAS FILTER
#include<filter.h>
fir_state_fr16 FIRStateL;
segment("L1_data_b") fract16 iFIRDelayL[FIR_CNT];
fir_state_fr16 FIRStateR;
segment("L1_data_b") fract16 iFIRDelayR[FIR_CNT];

/*************************************************************************
* Function:      ItProcessData
* Description:   Gets data from ADC, and handles the buffer system.
*************************************************************************/
void ItProcessData(int &iLeft,int &iRight)
{
  iInBufferL[iInBuffPos] = iLeft>>16;
  iInBufferR[iInBuffPos] = iRight>>16;
  iInBuffPos = (iInBuffPos+1)%BUFF_SIZE;
  iInBuffData++;
  if(iInBuffData >= N1)
  {
    iInBuffData = 0;
    bProcessFlag = true;
  }
  iLeft  = iPlayBufferL[iOutBuffPos]<<16;
  iRight = iLeft;//iPlayBufferR[iOutBuffPos]<<16;
  iOutBuffPos = (iOutBuffPos+1)%OUT_BUFF_SIZE;
}

/*************************************************************************
* Function:      Init_Process
* Description:   Inits the buffer for the process.
*************************************************************************/
void Init_Process(void)
{
  twidfftrad2_fr16(w,WND_SIZE);
  gen_hanning_fr16(wnd,1,WND_SIZE);

  for(int i=0;i<WND_SIZE;i++)
  {
    iPhsL[i] = rand()%PHASE_WRAP;
    iPhsR[i] = rand()%PHASE_WRAP;
  }
```

```
}

/*************************************************************************
* Function:      PhaseWrap
* Description:   Wraps the Phase of the FFT.
*************************************************************************/
fract16 PhaseWrap(int iPhs)
{
  return iPhs<0?(PHASE_MAX+iPhs)%PHASE_WRAP:(iPhs)%PHASE_WRAP;
}

/*************************************************************************
* Function:      GeneratePhs
* Description:   Makes The IFFT phase.
*************************************************************************/
void GeneratePhs(fract16 iPhsN[], fract16 iPhsL[], fract16 iPhs[])
{
  for(int i=0;i<WND_SIZE;i++)
  {
    iPhs[i]= PhaseWrap(iPhs[i] + (iWndPhs[i] +
                        PhaseWrap(iPhsN[i]-iPhsL[i]-iWndPhs[i]))*SPEED);
  }
}

/*************************************************************************
* Function:      Init_Filter
* Description:   AntiAlias Filter and ReSample init.
*************************************************************************/
void Init_Filter(void)
{
  fir_init(FIRStateL,iFIR,iFIRDelayL,FIR_CNT,0);
  fir_init(FIRStateR,iFIR,iFIRDelayR,FIR_CNT,0);
}

#include<math.h>
double dTemp;
int iTemp;
/*************************************************************************
* Function:      ReSample
* Description:   ReSamples the input data to the given format.
*************************************************************************/
void ReSample(fract16 iBIn[], fract16 iBOut[], fir_state_fr16 *FIRState)
{
  if(SPEED>1) fir_fr16(iBIn,iBIn,N2,FIRState);
  for(int i=0;i<N1;i++)
  {
    dTemp = i*SPEED;
    iTemp = floor(dTemp);
    dTemp = dTemp-iTemp;
    iBOut[i] = iBIn[iTemp] + (iBIn[iTemp+1]-iBIn[iTemp])*dTemp;
  }
  iBOut[N1-1] = iBIn[N2-1];
  if(SPEED<1) fir_fr16(iBOut,iBOut,N1,FIRState);
}

segment("L1_data_b") int blk_exp;
/*************************************************************************
* Function:      ProcessData
* Description:   Process data in buffer system.
*************************************************************************/
void ProcessData(void)
{
  for(int i=0;i<WND_SIZE;i++)
    fftinput[i] = mult_fr1x16(wnd[i],iInBufferL[(iInBuffStart+i)%BUFF_SIZE]);
  rfft_fr16(fftinput,fftcoeff,w,1,WND_SIZE,&blk_exp,1);
  for(int i=0;i<WND_SIZE;i++)
  {
    fftphsNL[i] = arg_fr16(fftcoeff[i]);
    fftampNL[i] = cabs_fr16(fftcoeff[i])<<1;
```

```
    }
    fftampNL[0] = 0;

    GeneratePhs(fftphsNL,fftphsLL,iPhsL);
    for(int i=0;i<WND_SIZE;i++) fftcoeff[i] = polar_fr16(fftampNL[i],iPhsL[i]);
    ifft_fr16(fftcoeff,fftgen,w,1,WND_SIZE,&blk_exp,1);

    for(int i=0;i<WND_SIZE;i++)
      iOutBufferL[(iOutBuffStart+i)%OUT_BUFF_SIZE] =
      iOutBufferL[(iOutBuffStart+i)%OUT_BUFF_SIZE] +
      mult_fr1x16(wnd[i],(fftgen[i]).re<<(WND_FRAC-1));
    for(int i=0;i<N2;i++) fftgen2[i] = iOutBufferL[(iOutBuffStart+i)%OUT_BUFF_SIZE];
    ReSample(fftgen2,fftgen3,&FIRStateL);
    for(int i=0;i<N1;i++) iPlayBufferL[(iPlayBuffWrite+i)%PLAY_BUFF_SIZE] = fftgen3[i];

    for(int i=0;i<N2;i++)
    {
      iOutBufferL[(iOutBuffStart+i)%OUT_BUFF_SIZE] = 0;
      iOutBufferR[(iOutBuffStart+i)%OUT_BUFF_SIZE] = 0;
    }


    // Update Buffer Pointers
    if(fftampLL==fftamp1L)
    {
      fftampLL = fftamp2L;
      fftampNL = fftamp1L;
      fftphsLL = fftphs2L;
      fftphsNL = fftphs1L;
      fftampLR = fftamp2R;
      fftampNR = fftamp1R;
      fftphsLR = fftphs2R;
      fftphsNR = fftphs1R;
    } else
    {
      fftampLL = fftamp1L;
      fftampNL = fftamp2L;
      fftphsLL = fftphs1L;
      fftphsNL = fftphs2L;
      fftampLR = fftamp1R;
      fftampNR = fftamp2R;
      fftphsLR = fftphs1R;
      fftphsNR = fftphs2R;
    }

    iInBuffStart = (iInBuffStart+N1)%BUFF_SIZE;
    iOutBuffStart = (iOutBuffStart+N2)%OUT_BUFF_SIZE;
    iPlayBuffWrite = (iPlayBuffWrite+N1)%PLAY_BUFF_SIZE;
    bProcessFlag = false;
}

#endif
```

B.2.4. Paramétereket generáló MATLAB szkript

```
clc;
clear all;
format long;
format compact;

disp('DSP_Frame_PV_gen.h_generating_enviroment.');
disp('_');

wnd = input('FFT_Window_Size_[1024]:_');
if isempty(wnd)
    wnd = 1024;
end
```

```
speed = input('Get Speed [1.0]: ');
if isempty(speed)
    speed = 1.0;
end

n1 = input('N1 step [400]: ');
if isempty(n1)
    n1 = 400;
end

disp(sprintf('FFT Window Size: %d',wnd));
disp(sprintf('Speed          : %d',speed));
disp(sprintf('N1 step        : %d',round(n1)));
disp(sprintf('N2 step        : %d',round(n1*speed)));

% ==== GENERATE ANTI-ALIAS FILTER ========================================
wbef = input('Before WPass freq [0.05] ');
if isempty(wbef)
    wbef = 0.05;
end

waft = input('After WPass freq [0.05] ');
if isempty(waft)
    waft = 0.05;
end

N = input('FIR Order [64] ');
if isempty(N)
    N = 64;
end

if (speed>1)
    wfreq = 1/speed;
    bcoeff  = firpm(N, [0 (wfreq-wbef) (wfreq+waft) 1], [1 1 0 0]);
elseif (speed<1)
    wfreq = speed;
    bcoeff  = firpm(N, [0 (wfreq-wbef) (wfreq+waft) 1], [1 1 0 0]);
else
    bcoeff  = zeros(N+1,1);
end


% ==== GENERATE GEN.H FILE ===============================================
fid = fopen('generator.h','w');

fprintf(fid,'// MATLAB GENERATED FILE...\n');
fprintf(fid,'// Used for PV sound generator\n');
fprintf(fid,'// Date:%s\n',date);
fprintf(fid,'#ifndef GENENRATOR\n');
fprintf(fid,'#define GENENRATOR\n\n');
fprintf(fid,'#define WND_SIZE %d\n',wnd);
fprintf(fid,'#define WND_FRAC %d\n',round(log2(wnd)));
fprintf(fid,'#define SPEED %f\n',speed);
fprintf(fid,'#define N1 %d\n',round(n1));
fprintf(fid,'#define N2 %d\n\n',round(n1*speed));
fprintf(fid,'#define PHASE_WRAP %d\n',2^15);
fprintf(fid,'#define PHASE_MAX %.0f\n',round(2^31));

fprintf(fid,'int iWndPhs[WND_SIZE] = {');
for k=1:wnd-1
    fprintf(fid,'%d, ',round(round(n1)*2^15*(k-1)/wnd));
end
fprintf(fid,'%d };\n\n',round(round(n1)*2^15*(wnd-1)/wnd));

% EXPORT ANTI-ALIAS FILTER ARRAY
fprintf(fid,'#define FIR_CNT %d\n\n',length(bcoeff));
fprintf(fid,'fract16 iFIR[FIR_CNT] = {');
for k=1:length(bcoeff)-1
    fprintf(fid,'%d, ',bcoeff(k));
```

```
    end
    fprintf(fid,'%d␣};\n\n',bcoeff(length(bcoeff)));

85  fprintf(fid,'#endif\n\n');
    fclose(fid);

    disp('␣');
    disp('Data␣Saved...');
```

C. függelék

Szimuláció kiértékelései

C.1. Vizsgálójelek

Hangfájl neve	Leírás
Tesztjelek	
001_Sin440Hz.wav	440 Hz-es szinuszjel
002_Saw440Hz.wav	440 Hz-es sávkorlátozott fűrészfogjel
003_SinExp.wav	Exponenciális frekvenciamenetű szinuszjel
004_Noise.wav	Fehérzaj
Beszédjelek	
001_Male_Ady1.wav	Ady Endre - Góg és magóg
002_Male_Ady2.wav	Ady Endre - Sem utódja, sem boldog őse
003_Female_KJanos1.wav	Komáromi János - Frekvenciák
004_Female_KJanos2.wav	Komáromi János - Valaki
Énekjelek	
001_VocalFemale1.wav	Steve Allen & Ben Alonzi Feat Tiff Lacey - Wildfire
002_VocalFemale2.wav	Morganpage - The Longest Road
005_VocalMale1.wav	David Guetta - The World Is Mine
006_VocalMale2.wav	David Latour - One Day
Zenei jelek	
001_Carmina1.wav	Carmina Burana
001_AgnesVanilla1.wav	Ágnes Vanilla - Talán eltűnök hirtelen (barokk)
001_JulieLondon.wav	Julie London - Cry Me A River
001_Animal1.wav	Animal Cannibals - Északon, Délen...
001_CosmicGate.wav	Cosmic Gate Featuring Tiff Lacey - Open Your Heart
001_MustangNismo.wav	Brian Tyler Feat. Slash - Mustang Nismo

C.1. táblázat: Vizsgáló jelek

C.2. Időtartománybeli szegmentálás eredményei

Állítási faktor = 0.6

Paraméter	Érték
Állítási faktor	0.6
Ablakméret	4096 minta (Hanning)
n1 lépésköz	2048 minta
n2 lépésköz	1229 minta

C.2. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	1.039982e-006	9.181939e-003	2.037199e-003
002_Saw440Hz	8.153753e-008	4.047159e-003	2.122639e-002
003_SinExp	7.012623e-013	3.668964e-003	6.325476e-004
004_Noise	2.442216e-012	1.418953e-004	6.165455e-005
001_Male_Ady1	7.809834e-012	8.211341e-005	1.558958e-004
002_Male_Ady2	1.825938e-011	5.697105e-004	2.235149e-004
003_Female_KJanos1	3.062948e-011	8.720398e-004	5.340661e-003
004_Female_KJanos2	3.075045e-011	6.075551e-003	2.603249e-003
001_VocalFemale1	5.458676e-011	7.101542e-004	3.113936e-003
002_VocalFemale2	4.265569e-012	1.389467e-004	1.748574e-003
005_VocalMale1	3.924789e-012	6.395069e-005	7.734510e-004
006_VocalMale2	7.250803e-012	8.311536e-004	1.694495e-004
001_Carmina1	1.042497e-010	7.270035e-005	4.972327e-004
001_AgnesVanilla1	1.158411e-011	5.128982e-004	1.428579e-004
001_JulieLondon	1.206627e-011	7.968007e-005	1.056982e-004
001_Animal1	1.023316e-011	1.952943e-004	5.648023e-005
001_CosmicGate	3.207460e-011	1.432075e-004	2.064629e-004
001_MustangNismo	8.029164e-012	8.914213e-005	1.712850e-004

C.3. táblázat: Mérési eredmények

Állítási faktor = 0.8

Paraméter	Érték
Állítási faktor	0.8
Ablakméret	8192 minta (Hanning)
n1 lépésköz	4096 minta
n2 lépésköz	3277 minta

C.4. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	1.048368e-006	5.267878e-002	1.214419e-003
002_Saw440Hz	8.151746e-008	1.705029e-003	8.067717e-002
003_SinExp	7.762060e-013	1.206089e-004	1.847303e-004
004_Noise	2.427833e-012	2.185859e-004	4.410836e-005
001_Male_Ady1	7.450025e-012	4.543945e-005	8.515705e-005
002_Male_Ady2	1.756772e-011	1.747204e-003	1.799959e-004
003_Female_KJanos1	3.001951e-011	4.233192e-004	1.264541e-003
004_Female_KJanos2	2.770846e-011	3.022814e-002	6.388986e-004
001_VocalFemale1	4.162435e-011	7.505793e-005	1.506006e-004
002_VocalFemale2	3.781610e-012	6.740486e-005	2.119916e-003
005_VocalMale1	3.654213e-012	1.001762e-004	2.510939e-003
006_VocalMale2	6.670793e-012	2.800288e-002	9.014947e-004
001_Carmina1	9.306073e-011	2.847103e-003	2.988794e-004
001_AgnesVanilla1	9.580802e-012	2.078038e-005	6.609111e-005
001_JulieLondon	9.589057e-012	6.398612e-005	1.864273e-005
001_Animal1	8.646890e-012	1.011619e-003	5.031365e-005
001_CosmicGate	2.915403e-011	4.968978e-005	3.867470e-005
001_MustangNismo	8.171446e-012	3.752748e-005	1.261139e-004

C.5. táblázat: Mérési eredmények

Állítási faktor = 0.9

Paraméter	Érték
Állítási faktor	0.9
Ablakméret	12289 minta (módosított-Hanning)
n1 lépésköz	6145 minta
n2 lépésköz	5531 minta

C.6. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	1.045993e-006	1.686904e-003	2.172124e-003
002_Saw440Hz	8.176444e-008	1.776211e-003	1.722026e-002
003_SinExp	1.748591e-012	6.497213e-005	1.223356e-004
004_Noise	2.856691e-012	3.013522e-005	4.564589e-005
001_Male_Ady1	8.722281e-012	4.190807e-005	9.528282e-005
002_Male_Ady2	2.171626e-011	2.376876e-003	1.354687e-004
003_Female_KJanos1	3.910500e-011	2.061387e-004	1.267187e-003
004_Female_KJanos2	3.118498e-011	2.065015e-002	6.571846e-004
001_VocalFemale1	5.601558e-011	8.896497e-005	1.164158e-004
002_VocalFemale2	4.648044e-012	8.903282e-005	1.389968e-003
005_VocalMale1	4.313579e-012	7.719077e-004	5.514969e-003
006_VocalMale2	7.262082e-012	3.230363e-002	5.507903e-003
001_Carmina1	8.971048e-011	3.516333e-003	4.077325e-004
001_AgnesVanilla1	1.026747e-011	1.410217e-005	4.904614e-005
001_JulieLondon	9.352646e-012	8.722971e-005	3.106765e-005
001_Animal1	9.857347e-012	6.990398e-004	1.574010e-005
001_CosmicGate	2.890315e-011	3.019096e-004	1.703328e-004
001_MustangNismo	8.239538e-012	2.788943e-005	8.008417e-005

C.7. táblázat: Mérési eredmények

Állítási faktor = 0.95

Paraméter	Érték
Állítási faktor	0.95
Ablakméret	20481 minta (módosított-Hanning)
n1 lépésköz	10241 minta
n2 lépésköz	9729 minta

C.8. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	1.044459e-006	1.650502e-003	5.056662e-003
002_Saw440Hz	8.135741e-008	7.766903e-004	1.224035e-002
003_SinExp	2.348600e-012	4.466351e-005	1.020310e-004
004_Noise	2.943194e-012	6.873661e-005	3.919832e-005
001_Male_Ady1	9.123945e-012	2.414074e-004	8.213824e-005
002_Male_Ady2	2.273757e-011	3.300618e-003	8.010133e-005
003_Female_KJanos1	3.691913e-011	3.076878e-004	9.722908e-004
004_Female_KJanos2	3.279662e-011	7.977972e-003	4.425734e-004
001_VocalFemale1	5.148604e-011	3.842802e-005	8.661923e-005
002_VocalFemale2	4.686906e-012	3.052022e-004	1.460716e-003
005_VocalMale1	4.653815e-012	2.469871e-005	2.121541e-003
006_VocalMale2	7.850194e-012	2.673097e-002	1.840502e-004
001_Carmina1	8.482870e-011	1.228399e-005	1.664552e-004
001_AgnesVanilla1	1.006859e-011	5.277875e-005	1.101823e-004
001_JulieLondon	8.298391e-012	1.864375e-005	3.410209e-004
001_Animal1	8.967042e-012	1.988000e-005	1.628611e-005
001_CosmicGate	2.450038e-011	2.962432e-005	3.172209e-005
001_MustangNismo	8.330686e-012	1.536119e-004	1.157505e-004

C.9. táblázat: Mérési eredmények

Állítási faktor = 1.05

Paraméter	Érték
Állítási faktor	1.05
Ablakméret	16384 minta (Hanning)
n1 lépésköz	8192 minta
n2 lépésköz	8602 minta

C.10. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	6.203208e-008	2.435650e-003	1.317705e-003
002_Saw440Hz	6.023360e-009	5.345102e-004	5.120808e-003
003_SinExp	7.648345e-013	5.160419e-005	7.873925e-005
004_Noise	2.176938e-012	6.037270e-003	7.202896e-005
001_Male_Ady1	6.313343e-012	3.573187e-004	1.201654e-004
002_Male_Ady2	1.594196e-011	4.630416e-003	2.208395e-005
003_Female_KJanos1	2.622722e-011	1.945714e-004	1.006482e-003
004_Female_KJanos2	2.634371e-011	7.522184e-003	3.622729e-004
001_VocalFemale1	3.674203e-011	1.853274e-005	3.330601e-005
002_VocalFemale2	3.266874e-012	4.468917e-005	9.433962e-004
005_VocalMale1	3.328303e-012	2.076612e-005	1.881614e-003
006_VocalMale2	5.491142e-012	1.461933e-002	1.603887e-005
001_Carmina1	7.094978e-011	1.647512e-004	1.294462e-004
001_AgnesVanilla1	7.860086e-012	3.827013e-005	2.748583e-005
001_JulieLondon	7.047972e-012	6.068025e-006	1.487282e-005
001_Animal1	7.733061e-012	4.329657e-005	1.278365e-005
001_CosmicGate	2.186685e-011	2.397047e-004	6.047252e-004
001_MustangNismo	7.182289e-012	1.149814e-005	1.040264e-004

C.11. táblázat: Mérési eredmények

Állítási faktor = 1.1

Paraméter	Érték
Állítási faktor	1.1
Ablakméret	16384 minta (Hanning)
n1 lépésköz	8192 minta
n2 lépésköz	9011 minta

C.12. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	4.838788e-008	9.585632e-004	6.990975e-003
002_Saw440Hz	4.930827e-009	2.614899e-004	1.537315e-003
003_SinExp	6.710456e-013	2.056921e-005	7.205060e-005
004_Noise	2.009246e-012	3.655141e-005	5.955684e-004
001_Male_Ady1	5.941678e-012	1.532540e-003	8.521283e-005
002_Male_Ady2	1.500068e-011	4.284067e-003	2.785528e-004
003_Female_KJanos1	2.497885e-011	8.741167e-005	5.183623e-004
004_Female_KJanos2	2.326739e-011	5.154535e-003	4.944150e-004
001_VocalFemale1	3.370247e-011	1.727614e-005	4.070321e-005
002_VocalFemale2	3.074072e-012	3.436574e-005	7.790385e-004
005_VocalMale1	3.061440e-012	1.146636e-005	1.431397e-003
006_VocalMale2	5.386025e-012	1.218590e-002	5.288562e-005
001_Carmina1	6.761758e-011	1.263025e-004	9.881380e-005
001_AgnesVanilla1	6.720413e-012	3.771395e-005	2.200514e-005
001_JulieLondon	6.596547e-012	9.975566e-006	6.741673e-005
001_Animal1	6.787784e-012	1.187704e-005	1.231617e-005
001_CosmicGate	2.242635e-011	3.262072e-005	3.203529e-005
001_MustangNismo	6.834836e-012	2.657435e-004	6.673723e-005

C.13. táblázat: Mérési eredmények

Állítási faktor = 1.2

Paraméter	Érték
Állítási faktor	1.2
Ablakméret	16384 minta (Hanning)
n1 lépésköz	8192 minta
n2 lépésköz	9830 minta

C.14. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	3.074363e-008	2.584284e-004	1.081277e-001
002_Saw440Hz	3.602680e-009	3.451021e-005	3.926092e-003
003_SinExp	5.267036e-013	1.679918e-004	3.040370e-005
004_Noise	1.716394e-012	2.863200e-005	2.390515e-005
001_Male_Ady1	5.155991e-012	9.580928e-006	5.953717e-005
002_Male_Ady2	1.279126e-011	4.967334e-003	1.321379e-004
003_Female_KJanos1	2.094866e-011	1.122005e-004	8.854451e-004
004_Female_KJanos2	1.959980e-011	5.782878e-003	3.304769e-004
001_VocalFemale1	2.737713e-011	3.167525e-005	5.434082e-005
002_VocalFemale2	2.613192e-012	4.277940e-005	7.087707e-004
005_VocalMale1	2.558551e-012	5.716332e-004	1.462059e-003
006_VocalMale2	4.256128e-012	1.034226e-002	1.659957e-004
001_Carmina1	5.534315e-011	2.441757e-004	9.883364e-004
001_AgnesVanilla1	5.688908e-012	7.372568e-006	1.938993e-005
001_JulieLondon	5.565143e-012	5.373828e-006	8.840832e-006
001_Animal1	5.876668e-012	3.388364e-006	1.482148e-005
001_CosmicGate	1.864493e-011	2.236368e-005	2.360058e-004
001_MustangNismo	5.509427e-012	1.003871e-005	2.880419e-003

C.15. táblázat: Mérési eredmények

Állítási faktor = 1.3

Paraméter	Érték
Állítási faktor	1.3
Ablakméret	10241 minta (módosított-Hanning)
n1 lépésköz	5121 minta
n2 lépésköz	6657 minta

C.16. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	4.590077e-008	1.183564e-003	6.460741e-003
002_Saw440Hz	4.406342e-009	5.689933e-004	1.755555e-003
003_SinExp	4.501443e-013	6.773835e-004	3.647962e-005
004_Noise	1.548895e-012	3.387411e-005	2.062359e-005
001_Male_Ady1	4.749263e-012	9.351157e-005	5.509517e-005
002_Male_Ady2	1.154883e-011	4.928081e-003	1.060251e-004
003_Female_KJanos1	1.932415e-011	5.667447e-004	4.573462e-004
004_Female_KJanos2	1.806765e-011	3.815022e-003	3.481114e-004
001_VocalFemale1	2.771769e-011	3.207236e-005	4.105513e-005
002_VocalFemale2	2.373292e-012	3.141148e-005	4.422137e-004
005_VocalMale1	2.391394e-012	4.270329e-005	1.274164e-003
006_VocalMale2	4.254524e-012	7.603559e-003	1.447741e-005
001_Carmina1	5.418089e-011	2.164247e-005	7.310220e-005
001_AgnesVanilla1	6.453767e-012	2.048756e-005	2.535737e-005
001_JulieLondon	5.676206e-012	8.647972e-005	1.732715e-005
001_Animal1	5.391262e-012	2.003483e-005	1.287217e-005
001_CosmicGate	1.592098e-011	1.610957e-005	1.663755e-005
001_MustangNismo	5.375851e-012	2.743637e-005	7.851794e-005

C.17. táblázat: Mérési eredmények

C.3. Fázis vokódolás eredményei

Állítási faktor = 0.6

Paraméter	Érték
Állítási faktor	0.6
Ablakméret	2048 minta
n1 lépésköz	512 minta
n2 lépésköz	307 minta

C.18. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	7.400820e-007	2.220125e-004	4.732996e-002
002_Saw440Hz	6.026025e-008	1.090200e-003	9.144097e-003
003_SinExp	1.243661e-012	5.584044e-005	6.305625e-004
004_Noise	2.398985e-012	4.842578e-005	7.464392e-005
001_Male_Ady1	7.430443e-012	3.156918e-005	1.663427e-004
002_Male_Ady2	2.182934e-011	6.349069e-004	1.238023e-004
003_Female_KJanos1	2.953313e-011	2.394178e-003	4.346194e-003
004_Female_KJanos2	2.922531e-011	5.187281e-003	5.078866e-004
001_VocalFemale1	3.398332e-011	1.806463e-004	9.704148e-005
002_VocalFemale2	3.863931e-012	1.108844e-004	1.869125e-003
005_VocalMale1	3.669696e-012	2.027178e-004	6.956166e-004
006_VocalMale2	6.162505e-012	4.422929e-004	6.092445e-005
001_Carmina1	6.552607e-011	2.779973e-005	1.100643e-003
001_AgnesVanilla1	6.382455e-012	2.528034e-005	1.890226e-004
001_JulieLondon	6.039804e-012	2.605923e-005	4.794591e-005
001_Animal1	5.913029e-012	4.658634e-005	3.537444e-005
001_CosmicGate	1.771261e-011	1.587029e-004	1.624406e-004
001_MustangNismo	6.472302e-012	6.217067e-005	1.155111e-004

C.19. táblázat: Mérési eredmények

Állítási faktor = 0.8

Paraméter	Érték
Állítási faktor	0.8
Ablakméret	2048 minta
n1 lépésköz	512 minta
n2 lépésköz	410 minta

C.20. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	6.324514e-007	8.682897e-005	8.258211e-003
002_Saw440Hz	5.003353e-008	1.811447e-001	6.571570e-002
003_SinExp	1.430817e-012	1.589026e-004	1.868490e-004
004_Noise	2.341832e-012	3.538685e-005	4.752897e-005
001_Male_Ady1	7.572568e-012	4.863728e-005	8.175472e-005
002_Male_Ady2	1.770304e-011	1.058276e-003	5.909553e-005
003_Female_KJanos1	3.047818e-011	2.710542e-004	1.551891e-003
004_Female_KJanos2	2.817806e-011	3.229213e-002	7.016949e-004
001_VocalFemale1	3.855060e-011	5.438958e-005	3.428223e-004
002_VocalFemale2	3.759412e-012	5.761563e-005	2.166571e-003
005_VocalMale1	3.596549e-012	6.390547e-005	2.368148e-003
006_VocalMale2	6.166251e-012	3.012844e-002	2.310557e-005
001_Carmina1	5.636765e-011	1.186798e-005	2.692126e-004
001_AgnesVanilla1	6.448693e-012	6.982069e-005	2.181987e-005
001_JulieLondon	6.023752e-012	1.302332e-004	1.470914e-005
001_Animal1	5.592619e-012	1.963978e-003	2.093463e-005
001_CosmicGate	1.704335e-011	8.288465e-004	1.466718e-003
001_MustangNismo	6.836056e-012	3.250188e-004	1.191920e-004

C.21. táblázat: Mérési eredmények

Állítási faktor = 0.9

Paraméter	Érték
Állítási faktor	0.9
Ablakméret	2048 minta
n1 lépésköz	512 minta
n2 lépésköz	461 minta

C.22. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	5.704017e-007	1.792805e-006	4.628199e-002
002_Saw440Hz	4.455567e-008	2.964396e-003	1.348450e-002
003_SinExp	9.342618e-013	8.403441e-004	1.711951e-004
004_Noise	2.288671e-012	7.493014e-005	4.073920e-005
001_Male_Ady1	6.952431e-012	6.697576e-004	6.782558e-005
002_Male_Ady2	1.890465e-011	4.175490e-003	6.383058e-005
003_Female_KJanos1	2.810208e-011	3.827181e-003	1.214301e-003
004_Female_KJanos2	2.859470e-011	1.986009e-002	2.231784e-004
001_VocalFemale1	3.181122e-011	2.937110e-005	1.339548e-004
002_VocalFemale2	3.497909e-012	5.977263e-005	1.058508e-003
005_VocalMale1	3.455060e-012	2.388452e-004	2.138792e-003
006_VocalMale2	5.769394e-012	3.054386e-002	2.204429e-003
001_Carmina1	5.926618e-011	1.524280e-004	2.549297e-004
001_AgnesVanilla1	5.959698e-012	6.012551e-004	5.602826e-005
001_JulieLondon	5.535742e-012	1.005707e-004	1.878833e-005
001_Animal1	5.632339e-012	1.707996e-004	3.008352e-004
001_CosmicGate	1.877601e-011	2.901776e-004	6.730866e-005
001_MustangNismo	6.342870e-012	2.251003e-004	1.108638e-004

C.23. táblázat: Mérési eredmények

Állítási faktor = 0.95

Paraméter	Érték
Állítási faktor	0.95
Ablakméret	2048 minta
n1 lépésköz	512 minta
n2 lépésköz	486 minta

C.24. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	5.245943e-007	7.024361e-004	3.439628e-002
002_Saw440Hz	4.082388e-008	1.269183e-003	2.492148e-003
003_SinExp	1.073703e-012	6.817479e-005	5.305206e-005
004_Noise	2.200372e-012	9.291790e-004	4.047227e-005
001_Male_Ady1	6.545181e-012	3.031120e-004	1.439461e-004
002_Male_Ady2	1.589979e-011	4.081001e-003	4.100624e-005
003_Female_KJanos1	2.594810e-011	2.648965e-004	1.178601e-003
004_Female_KJanos2	2.566407e-011	6.858082e-003	3.208174e-004
001_VocalFemale1	3.577648e-011	5.196009e-005	9.363188e-005
002_VocalFemale2	3.189394e-012	5.849009e-005	1.134675e-003
005_VocalMale1	3.284115e-012	1.035257e-003	1.847059e-003
006_VocalMale2	5.747511e-012	2.738706e-002	2.161527e-004
001_Carmina1	5.037137e-011	1.111208e-005	1.607287e-004
001_AgnesVanilla1	6.242342e-012	1.527722e-005	1.726790e-004
001_JulieLondon	5.600935e-012	4.647650e-006	3.009112e-005
001_Animal1	5.726232e-012	4.716286e-005	2.111873e-005
001_CosmicGate	1.956165e-011	2.883912e-005	2.976309e-005
001_MustangNismo	6.411479e-012	1.582303e-005	1.518932e-004

C.25. táblázat: Mérési eredmények

Állítási faktor = 1.05

Paraméter	Érték
Állítási faktor	1.05
Ablakméret	2048 minta
n1 lépésköz	512 minta
n2 lépésköz	538 minta

C.26. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	2.044128e-009	2.077778e-003	2.948539e-003
002_Saw440Hz	2.232292e-010	3.722101e-004	6.049919e-004
003_SinExp	9.195304e-013	2.108993e-005	4.235338e-005
004_Noise	2.000597e-012	3.723313e-005	1.820016e-004
001_Male_Ady1	5.990632e-012	1.297577e-005	7.745868e-005
002_Male_Ady2	1.444466e-011	4.469283e-003	4.728717e-005
003_Female_KJanos1	2.373020e-011	1.615436e-004	6.985431e-004
004_Female_KJanos2	2.253493e-011	8.641808e-003	3.438321e-004
001_VocalFemale1	3.248429e-011	2.774892e-005	7.341785e-005
002_VocalFemale2	2.923673e-012	4.657848e-005	6.097442e-004
005_VocalMale1	3.011577e-012	2.371278e-004	1.634104e-003
006_VocalMale2	5.165134e-012	2.124179e-002	1.889824e-005
001_Carmina1	4.896452e-011	7.080236e-005	1.143522e-004
001_AgnesVanilla1	5.630313e-012	1.178487e-004	2.183246e-005
001_JulieLondon	5.209721e-012	6.177580e-006	1.411472e-005
001_Animal1	5.093781e-012	2.097722e-005	1.532473e-005
001_CosmicGate	1.749567e-011	2.366639e-005	9.765394e-005
001_MustangNismo	5.991418e-012	2.863499e-004	1.201361e-004

C.27. táblázat: Mérési eredmények

Állítási faktor = 1.1

Paraméter	Érték
Állítási faktor	1.1
Ablakméret	2048 minta
n1 lépésköz	512 minta
n2 lépésköz	563 minta

C.28. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	4.835140e-009	2.023098e-003	1.127178e-003
002_Saw440Hz	2.882303e-010	3.752128e-004	8.325621e-004
003_SinExp	6.942030e-013	2.135155e-005	5.745207e-005
004_Noise	1.890067e-012	4.745685e-005	1.631745e-004
001_Male_Ady1	5.746861e-012	2.668035e-005	4.967005e-004
002_Male_Ady2	1.575673e-011	2.303395e-003	1.406910e-004
003_Female_KJanos1	2.376980e-011	1.275779e-004	8.468924e-004
004_Female_KJanos2	2.338422e-011	6.645320e-003	4.027468e-004
001_VocalFemale1	2.851124e-011	2.149011e-005	3.786261e-005
002_VocalFemale2	2.843694e-012	9.054675e-005	7.700410e-004
005_VocalMale1	2.885612e-012	1.201783e-005	1.429007e-003
006_VocalMale2	4.747567e-012	1.572978e-002	2.519790e-005
001_Carmina1	4.346563e-011	4.931112e-005	7.649764e-005
001_AgnesVanilla1	4.955505e-012	3.524979e-003	2.424054e-004
001_JulieLondon	4.563103e-012	2.458426e-005	5.579213e-004
001_Animal1	4.779878e-012	1.160431e-004	2.272718e-005
001_CosmicGate	1.477302e-011	3.062039e-005	7.114681e-005
001_MustangNismo	5.135961e-012	2.421743e-005	6.854857e-005

C.29. táblázat: Mérési eredmények

Állítási faktor = 1.2

Paraméter	Érték
Állítási faktor	1.2
Ablakméret	2048 minta
n1 lépésköz	512 minta
n2 lépésköz	614 minta

C.30. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	2.664649e-009	6.779399e-005	3.398810e+001
002_Saw440Hz	3.219202e-010	2.999393e-005	2.812479e-003
003_SinExp	8.195018e-013	8.727702e-005	6.805675e-005
004_Noise	1.620029e-012	2.499374e-005	9.921247e-005
001_Male_Ady1	5.159459e-012	1.668504e-004	6.195953e-005
002_Male_Ady2	1.229792e-011	3.926254e-003	2.935140e-005
003_Female_KJanos1	1.985691e-011	1.422808e-004	6.018886e-004
004_Female_KJanos2	1.891617e-011	5.491609e-003	2.812096e-004
001_VocalFemale1	2.628853e-011	6.052402e-005	2.298618e-003
002_VocalFemale2	2.572737e-012	5.908009e-005	7.818919e-004
005_VocalMale1	2.499117e-012	2.233814e-005	1.510054e-003
006_VocalMale2	4.190067e-012	1.287732e-002	7.465811e-005
001_Carmina1	3.838818e-011	9.588399e-006	7.144416e-005
001_AgnesVanilla1	4.412804e-012	4.858445e-005	2.236722e-005
001_JulieLondon	4.185962e-012	2.537679e-005	1.563877e-005
001_Animal1	3.965629e-012	2.401679e-004	9.082889e-006
001_CosmicGate	1.169821e-011	1.881426e-005	2.223853e-005
001_MustangNismo	4.510196e-012	3.094608e-005	7.239883e-005

C.31. táblázat: Mérési eredmények

Állítási faktor = 1.3

Paraméter	Érték
Állítási faktor	1.3
Ablakméret	2048 minta
n1 lépésköz	512 minta
n2 lépésköz	666 minta

C.32. táblázat: Paraméterek

Hangfájl neve	FFT Hiba	STFT Hiba	Wavelet Hiba
001_Sin440Hz	2.590846e-009	2.129824e-002	8.051233e-003
002_Saw440Hz	3.823587e-010	1.164166e-001	6.932081e-003
003_SinExp	5.280115e-013	1.061408e-004	3.942958e-005
004_Noise	1.386440e-012	4.526961e-004	2.270806e-005
001_Male_Ady1	4.156630e-012	8.614484e-005	4.684841e-005
002_Male_Ady2	1.009844e-011	4.388138e-003	9.684491e-004
003_Female_KJanos1	1.739842e-011	1.058470e-004	3.692446e-004
004_Female_KJanos2	1.596449e-011	3.656406e-003	3.075043e-004
001_VocalFemale1	2.337850e-011	1.696369e-003	6.292796e-005
002_VocalFemale2	2.166094e-012	2.663413e-004	5.192566e-004
005_VocalMale1	2.110361e-012	1.733529e-005	1.325396e-003
006_VocalMale2	3.539108e-012	8.702148e-003	1.928875e-004
001_Carmina1	3.629808e-011	1.904934e-005	6.646179e-005
001_AgnesVanilla1	3.468611e-012	5.855576e-005	1.545670e-005
001_JulieLondon	3.560172e-012	9.147506e-005	8.246002e-006
001_Animal1	3.427311e-012	5.990652e-003	3.254149e-005
001_CosmicGate	9.892201e-012	4.640305e-005	9.035540e-005
001_MustangNismo	3.929949e-012	8.570657e-005	9.901212e-005

C.33. táblázat: Mérési eredmények

D. függelék

A DSP szoftver használati utasítása

Ebben a mellékletben a DSP szoftver különböző paramétereinek a beállításáról lesz szó, valamint a kártyára történő fordítás menetéről. Mint az már a 4. fejezetben említve lett, a könnyebb kezelhetőség érdekében a MATLAB generálja ki azokat a paramétereket, melyek minden beállításkor megváltoznak. Például az anti-alias szűrőegyütthatók, vagy a speciális ablakfüggvények.

A fordítás úgy történik, hogy a MATLAB-ban a megfelelő szkript fájlt futtatva, az különböző kérdéseket tesz fel a felhasználónak, melyek megválaszolása után kigenerál egy C/C++ header fájlt.

Ezután a Visual DSP++ fejlesztőkörnyezetben a projektet újra kell fordítani, és fel kell tölteni a kártyára, ezáltal az új header fájl okozta változások érvényesülni fognak. Magában a kódban nem kell módosítást végezni.

Rosszul beállított paraméterek mellett megtörténhet, hogy a DSP számítási kapacitása nem elég. Ez úgy jelentkezik az audió jelben, hogy kimaradnak bufferek, vagy bufferrészek, ezáltal pattanást, kattanást okozva.

Az is előfordulhat, hogy a túl nagyra választott bufferméret nem fér el a DSP belső memóriájában, ekkor a fordítás sikertelen, és hibaüzenet tapasztalható.

D.1. Időtartománybeli szegmentálás

A MATLAB szkript paramétere az ablak mérete, amelyet az OLA használ a szegmensekre bontásnál. Itt a belső memória mérete szab korlátot, és 4096 a legnagyobb ablakméret, mely ilyen bufferelhelyezések mellett sikeresen lefordul. Jobb eredmény eléréséhez nagyobb bufferméretle lenne szükség, de ez a kártyán nem megvalósítható, mert a külső memória elérése túl lassú ahhoz hogy használjuk.

A második paraméter az ablak típusa. Itt a megszokott ablaktípusok közül lehet választani, valamint található egy „Home” nevezetű, mely egy módosított-Hanning ablakot takar.

Ezután az „N1” időtartománybeli lépésközt kell megadni. Ennyi mintánként kezdődik egy új ablak. Majd a pitch shift faktorára, ahol az egynél kisebb értékek a jel mélyítésének az egynél nagyobb értékek a magasításának felelnek meg.

A szkript kiszámolja, és grafikusan megjeleníti az ablakok illeszkedéséből származó hangerőkorrekciós vektort, ábrázolja az ablakfüggvényt, az ablakok átlapolódását,

valamint kiszámolja az anti-alias szűrő paramétereit, és ábrázolja a szűrő karakterisztikáját. Ezután elmenti a header fájlt, és már csak le kell fordítani és fel kell tölteni a DSP kártyára.

D.2. Fázis vokódolás

A fázis vokódolás MATLAB szkriptjénél az első paraméter az $\mathcal{FFT}$ mérete, melynek a memória elérés szab határt, és 1024 fölött hibajelenségek tapasztalhatóak, a többi paramétertől függően. Az ablakfüggvény itt fix Hanning ablak, melyet a DSP induláskor kigenerál magának.

Ezután a pitch shift faktorát kell megadnunk, majd az „N1" időtartománybeli lépésközt. Ez utóbbinál az ablakméret negyede lenne optimális, de kicsivel az ablakméret 50%-a alatt a számítási igénye túl nagy lesz az algoritmusnak, és bufferek, bufferrészletek kezdenek kimaradni.

A szkript itt is kiszámol egy anti-alias karakterisztikát, és a többi paraméterrel együtt elmenti a header fájlba, és itt is csak fordítani kell a kódot.

Rövidítések

AD	Analog-To-Digital Converter
ALU	Arithmetic Logic Unit
AM	Amplitúdó Moduláció
BPM	Beat Per Minute
CAN	Controller–Area Network
CWT	Continuous Wavelet Transform
DA	Digital-To-Analog Converter
DMA	Direct Memory Access
EM-TSM	Envelop-Matching for Time Scale Modification
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
GPIO	General-Purpose I/O
IDE	Integrated Development Environment
IrDA	Infrared Data Association
I^2S	Integrated Interchip Sound
IT	Interrupt
JTAG	Joint Test Action Group
MAC	Multiply-Accumulate
MAC	Media Access Control (Ethernet)
MRA	Multiresolution Analysis
OLA	Overlap and Add
PLL	Phase-Locked Loop
PPI	Parallel Peripheral Interface
PSOLA	Pitch-Synchronous Overlap and Add
RISC	Reduced Instruction Set Computer
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single-Instruction Multiple-Data
SOLA	Synchronized Overlap and Add
SPI	Serial Peripheral Interface
SPORT	Serial Peripheral Port
SRAM	Static Random Access Memory
STFT	Short-Time Fourier Transform
TWI	2-Wire Interface
UART	Universal Asynchronous Receiver/Transmitter
WSOLA	Waveform Similarity Overlap and Add