Szegedi Tudományegyetem

Informatikai Tanszékcsoport

SZAKDOLGOZAT

**Krausz Róbert Zsolt**

2013

Szegedi Tudományegyetem

Informatikai Tanszékcsoport

Izolált szavas, személyfüggő, kötött szótáras beszédfelismerő

Szakdolgozat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Készítette: |  | Témavezető: |  |
|  | Krausz Róbert Zsolt |  | Dr. Pletl Szilveszer |  |
|  | informatika szakos hallgató |  | főiskolai tanár |  |

Szeged

2013

## Feladatkiírás

Az automatizált rendszerek esetében egyre nagyobb teret kapnak a hangvezérlések. A hallgató feladata egy hangvezérlő rendszer készítése. A rendszernek fel kell ismernie az előre megadott vezényszavakat, valamint a hozzájuk tartozó utasításokat egy OPC szerver felé kell továbbítania. A feladat az alábbi részfeladatokra bontható: áttekinteni azon jelfeldolgozó algoritmusokat, amelyek alkalmazásával megbízhatóan működő hangvezérlő rendszer építhető, elkészíteni a rendszer szoftver tervét, elkészíteni a programot és végül tesztelni a megvalósított alkalmazást. A dolgozatnak tartalmaznia kell minden részfeladat részletes dokumentációját.

## Tartalmi összefoglaló

* ***A téma megnevezése:***

Izolált szavas, személyfüggő, kötött szótáras beszédfelismerő

* ***A megadott feladat megfogalmazása:***

Képessé kell tenni a számítógépet arra, hogy a felhasználó által betanított, majd kiejtett parancsszavakat felismerje, és végrehajtsa a parancsokhoz tartozó utasítást.

* ***A megoldási mód:***

C++ nyelven

* ***Alkalmazott eszközök, módszerek:***

A programot C++ nyelven valósítottam meg, programfejlesztő környezetnek a  Qt Creator-t választottam. Külső librarit az FMOD-ot használtam a hangfájlok megnyitására, és lejátszására, de ezt csak a teszteléshez használja fel a program.

* ***Elért eredmények:***

A program átlagos körülmények között, kevés zajterhelés mellet kiválóan működik, nagy pontossággal felismeri a betanított szavakat, és elküldi a parancsszavakat egy teszt szervernek.

* ***Kulcsszavak:***

Izolált szavas, személyfüggő, kötött szótáras beszédfelismerő, dinamikus idővetemítés

## Tartalomjegyzék

[Feladatkiírás 2](#_Toc374017460)

[Tartalmi összefoglaló 3](#_Toc374017461)

[Tartalomjegyzék 4](#_Toc374017462)

[1. Bevezetés 6](#_Toc374017463)

[2. A hang 7](#_Toc374017464)

[2.1. A hang digitalizálása 7](#_Toc374017465)

[2.2. Digitalizálás a programban 8](#_Toc374017466)

[3. Fourier transzformáció [4] 9](#_Toc374017467)

[3.1. Gyors Fourier-transzformáció 10](#_Toc374017468)

[4. Jelsorozat ablakozása [3] 11](#_Toc374017469)

[4.1. A téglalap ablak 11](#_Toc374017470)

[4.2. Hann ablak 11](#_Toc374017471)

[4.3. Ablakozás a programban 12](#_Toc374017472)

[5. Mel-skála [2] 12](#_Toc374017473)

[6. Kezdő- és végpont detektálás 14](#_Toc374017474)

[6.1. A mel-spektrum normalizálása 15](#_Toc374017475)

[7. Dinamikus idővetemítés 15](#_Toc374017476)

[7.1. A DTW keresési tér csökkentése 18](#_Toc374017477)

[7.2. A DTW használata a programban 18](#_Toc374017478)

[7.3. A DTW gyorsítása a mel spektrumon 19](#_Toc374017479)

[7.4. A keresés optimalizálása szó szinten 20](#_Toc374017480)

[7.5. DTW algoritmus tesztelése 21](#_Toc374017481)

[7.6. A szószintű idővetemítés tesztelése 24](#_Toc374017482)

[7.7. Keresési tesztek értékelése 27](#_Toc374017483)

[8. Program terv 28](#_Toc374017484)

[9. A beszédfelismerő rendszer felépítése 28](#_Toc374017485)

[9.1. Analóg digitális átalakító 29](#_Toc374017486)

[9.1.1. Az analóg átalakító kódja 30](#_Toc374017487)

[9.1.2. Paraméter vektor előállítása 31](#_Toc374017488)

[9.1.3. processSamples forráskódja 31](#_Toc374017489)

[9.2. Paraméter vektorok feldolgozása 35](#_Toc374017490)

[9.2.1. A paraméter feldolgozó forráskódja 35](#_Toc374017491)

[9.3. Minták megtanulása 36](#_Toc374017492)

[9.3.1. Szó paraméterek elmentésének forráskódja 37](#_Toc374017493)

[9.4. Paraméter vektorok felismerése 38](#_Toc374017494)

[9.4.1. A recognise forráskódja 39](#_Toc374017495)

[9.5. Dinamikus idővetemítés 44](#_Toc374017496)

[9.5.1. DTW forráskódja 44](#_Toc374017497)

[9.6. Mel-szűrő 47](#_Toc374017498)

[9.6.1. MelFilter forráskódja 47](#_Toc374017499)

[10. A program 50](#_Toc374017500)

[10.1. Felhasználói interfész 50](#_Toc374017501)

[11. A program tesztelése 53](#_Toc374017502)

[11.1. A parancs felismerő tesztelése 53](#_Toc374017503)

[11.2. MySQL adatbázisba írás tesztelése 55](#_Toc374017504)

[12. Irodalomjegyzék 57](#_Toc374017505)

# Bevezetés

Mindennapi életünk során egyre gyakrabban kerülünk kapcsolatba számítógépekkel és számítógéppel vezérelt készülékekkel. Ezeket az eszközöket vezérelnünk kell, és azok visszajelzéseket adnak. Vezérlésük hatékonyságát a kommunikációs interfészek határozzák meg. A számítógépeknél sokáig a billentyűzet és az egér dominált, mint input beviteli eszközök, azonban napjainkban a fizikai nyomógombos eszközöket egyre jobban háttérbe szorítják az érintő képernyős készülékek, melyeken a képernyőbe van integrálva a billentyűzet és egér funkcióit betöltő eszközök. Az érintő képernyős eszközökön az érintés pozíciójával és mozdulat gesztusokkal adhatunk ki utasításokat a készülékünknek.

A számítógépekkel való kommunikációt új szintre emelhetnénk, ha az ember által legtermészetesebb módon, beszéddel tudnánk kommunikálni velük. Tehát meg kell tanítanunk a gépeket, hogy megértsék az emberi beszédet.

Szakdolgozatomban egy izolált szavas, személyfüggő, kötött szótáras beszéd felismerőt valósítok meg. A beszéd felismerés e formájánál a számítógép kezelője tanítja meg a használni kívánt parancsszavakat a programnak. A beszéd felismerés során pedig szüneteket kell tartani a parancsszavak között, hogy a program meg tudja határozni a beszéd kezdő- és végpontját.

# A hang

Amit az ember hangnak érzékel az a levegő nyomásának periodikus változása. A hang analóg jel, azaz minden időpillanatban definiált, és végtelen sok értéke lehet. A számítógép azonban nem képes folytonos jeleket kezelni, ezért digitalizálni kell a jeleket.

## A hang digitalizálása

Digitalizálás során először a hangnyomás változás által hordozott információt a számítógép által értelmezhetővé kell tenni. Erre szolgáló eszköz a mikrofon, amely a levegőben terjedő rezgéseket átalakítja elektromos jelekké, amelyekből a számítógép hangkártyája előállít egy digitális számsorozatot, amikkel már fel tud dolgozni a számítógép. A hang digitalizálása során három fontos tényezőt kell figyelembe venni, amelyek befolyásolják a felvett hang minőségét [5]:

* a mintavételi frekvencia
* a kvantálási szám
* a hangcsatornák száma (mono vagy sztereo)

Mintavételezés során szabályos időközönként mintát veszünk a jelből. A mintavételezési frekvencia értéke a rögzíteni kívánt hangfrekvenciás jel frekvenciatartománya határozza meg. Ha ez az érték az analóg jelben előforduló legnagyobb frekvencia kétszerese, akkor a visszaállított hanganyag hangfrekvenciák szempontjából hibamentes lesz. Ezt a Shannon féle mintavételezési törvény fogalmazza meg:

Egy analóg jel mintavételezésénél adott a jelben előforduló maximális frekvencia *fmax*, és a mintavételező egység mintavételi frekvenciája *fminta*. A kettő között szoros kapcsolat található, ugyanis:

Négy szabályos mintavételezési frekvencia terjedt el, a 8000Hz, 11025 Hz, 22050 Hz és 44100Hz.

Kvantálásnak nevezzük azt a folyamatot, amikor mintavételezett impulzussorozat amplitúdó értékeit meghatározott számú bit felhasználásával bináris számokká alakítják át. A rendelkezésre álló bitek száma a kvantálási hossz. Minél több amplitúdó értéket lehet megkülönböztetni, annál pontosabban lehet visszajátszáskor a digitális jelsorozatból visszaállítani az eredeti analóg jelet. A kvantálás általában 16 biten történik, mely eljárás ebben az esetben 65 536 különböző kvantálási szintjével Hi-Fi minőséget eredményez. C++ programban egy mintát short típusú változóban tárolhatunk el melynek értéktartománya –32 768-tól 32,767-ig terjed.

Hangcsatornák száma szerint megkülönböztetünk mono és stereo hangcsatornás digitalizálást. Mono felvétel esetén egy mikrofonnal vesszük fel a hangot, ami egy csatornás felvételt eredményez. Két csatornás, sztereó felvételhez két mikrofonra van szükség, melyek eltérő pozícióból rögzítik a hangokat.

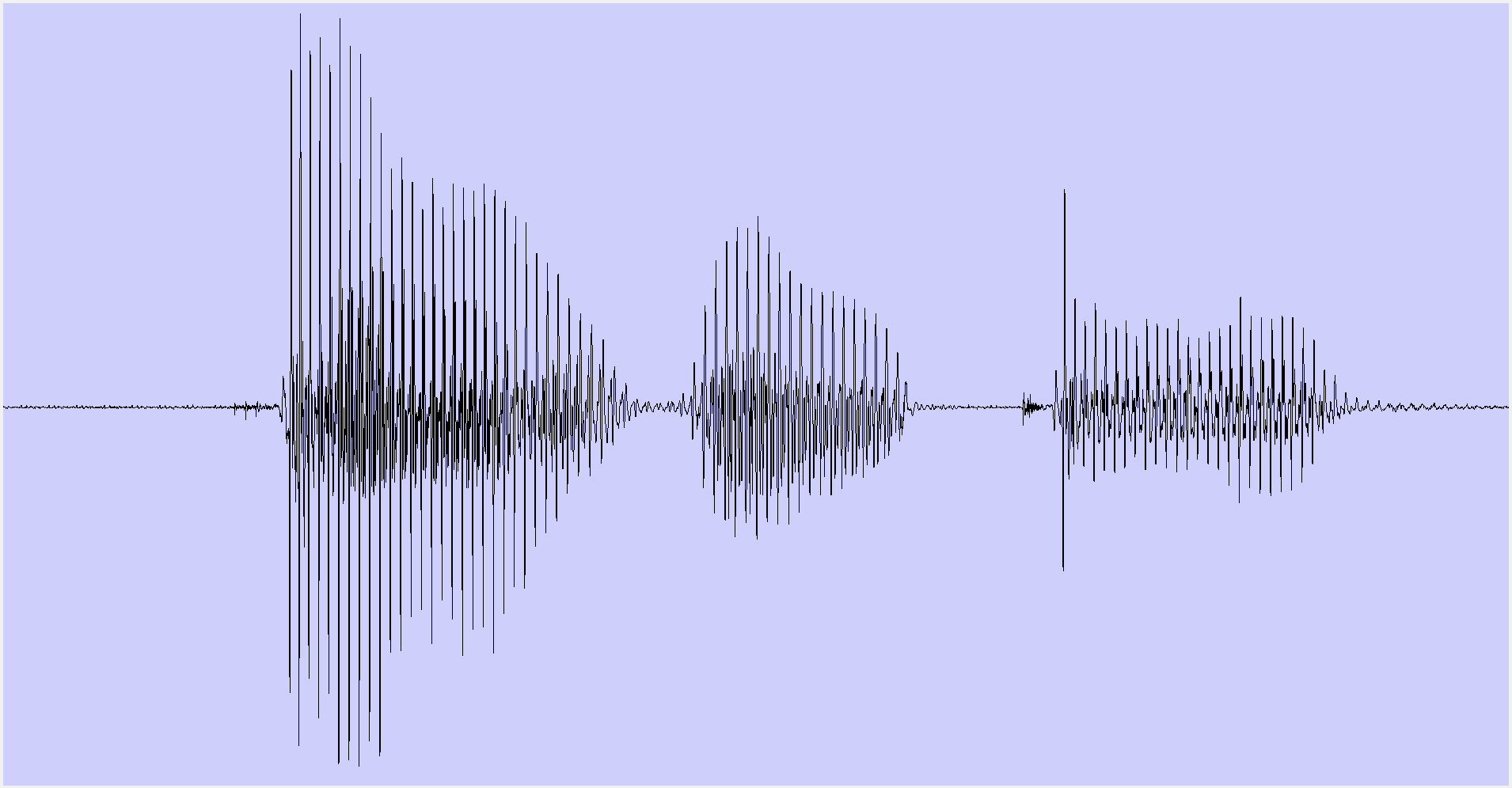
## Digitalizálás a programban

Az átlagos beszédben a beszéd tulajdonságait hordozó frekvencia spektrum felső határa körülbelül 5000hz-ig tart. Annak érdekében, hogy ne veszítsünk érdemi információt, a Shannon mintavételezési tételnek megfelelően 5000hz kétszeresénél magasabb mintavételezési frekvenciát kell választani. Azonban nem választhatunk számunkra tetsző mintavételezési frekvenciát, mivel a választható frekvenciákat a hangkártya paraméterei szabják meg. A legelterjedtebb mintavételezési frekvenciák közül, ezért a 11025 Hz-et választottam, amely megfelel a mintavételezési törvénynek, és nem terheli a hardvert, extra minták feldolgozásával.

Kvantálási méretnek a 2 bájtot választottam, ami elég nagy felbontás biztosít a mintáknak, és egyszerűen kezelhető, c++-ban short változóval.

Mivel otthoni környezetben legegyszerűbben mono felvételt tudunk készíteni, ezért az egycsatornás megoldást választottam, ami megfelel a minták feldolgozásához. A két csatornás felvételnek annyi előnye lenne a mono felvétellel szemben, hogy két csatorna esetén a zajszűrés megoldható lenne időtartományban, a két csatornához érkező beszéd időbeli távolságának kiszámítása után kivonva a két csatornát egymásból, megkapnánk a zajt, amit aztán ki tudnánk vonni az eredeti jelből, és sokkal tisztább hangokkal dolgozhatna a beszédfelismerő. Azonban zajszűrés után egy csatornás módszerrel folytatnánk a hangfeldolgozást. Csendes környezetben, statikus zajszűrővel dolgozva a mono felvétel is megfelel a programnak.

A 2.2.1. ábrán a „felvétel” szó digitalizálásának eredménye látszik. A hangminták értéke reprezentálja a légnyomás aktuális állapotát. Ábrázolva a mintákat a nagyobb amplitúdójú jeleknek nagyobb volt a beszéd során a hangerejük.



2.2.1. ábra*: A „felvétel” szó digitalizálás után*

# Fourier transzformáció [4]

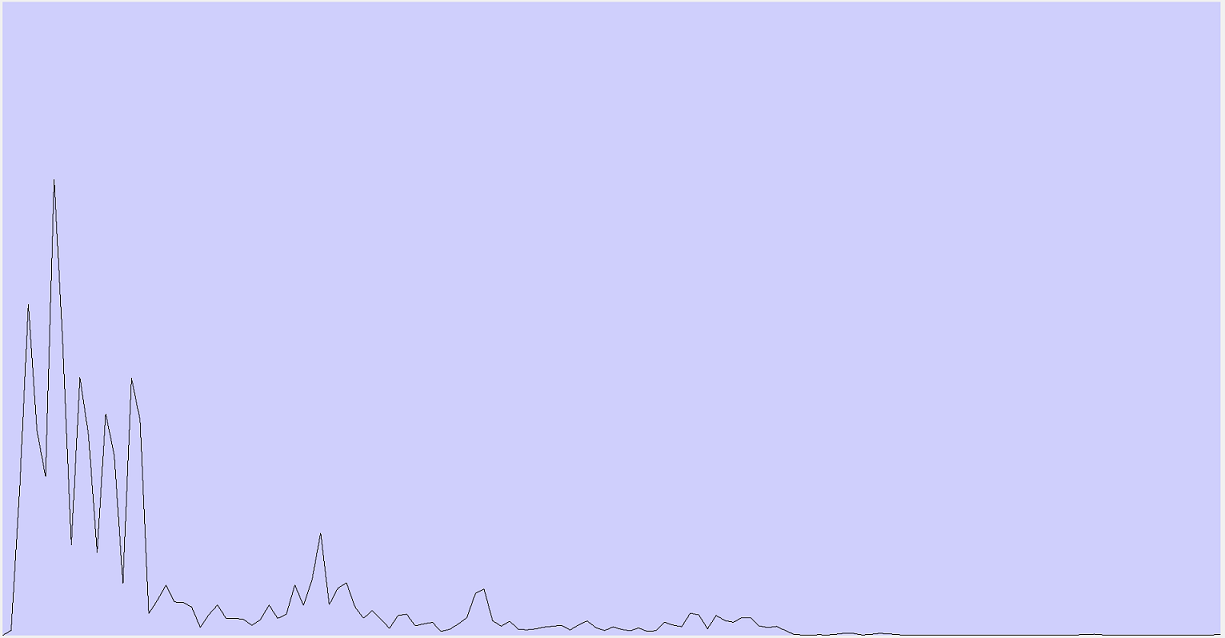
A beszéd egy időben változó jel, ami előállítható különböző frekvenciájú, fázisú és amplitúdójú jelek összegeként. A Fourier-transzformáció az a művelet, amely egy adott jelhez megadja ezt a felbontást.

Gyakorlati szempontból fontos eset, amikor egy véges hosszúságú, véges tartójú jel spektrumára vagyunk kíváncsiak. A Diszkrét Fourier-transzformáció végrehajtása véges hosszúságú jeleken nem okoz problémát, hiszen e jeleket tekinthetjük olyan végtelen hosszúságú jeleknek, amelyek számunkra érdektelen részei azonosan 0 értékűek. Tehát, tekintsük egy véges N mintából álló diszkrét x jelet, a Diszkrét Fourier transzformáció képlete:

Ahol Xj a j-edik spektrum elemének amplitúdóját és fázisát tárolja el egy komplex számban. Így Xj felírható:

Re(Xj) a szám valós része Im(Xj) pedig a szám képzetes része. Beszéd feldolgozásban a fázisnak nincs szerepe ezért csak a komplex szám amplitúdójára van szükségünk, amit megkaphatunk a következő képlettel:

A képletből megkapott A vektor tárolja a komplex számok amplitúdóját, ezt megjelenítve láthatóvá válik az aktuális spektrum, ennek megjelenítése látszódik a 3.1. ábrán.



3.1. ábra: Kiszámított spektrum ábrázolva

## Gyors Fourier-transzformáció

A gyors Fourier-transzformáció (Fast Fourier Transform) eredménye egyezik a fenti diszkrét Fourier transzformáció eredményével, azonban a művelethez szükséges számítási teljesítmény igénye sokkal kisebb. Azaz:

* DFT esetén a számítás lépésszáma: N2 (tmul + tadd)
* FFT esetén ugyanez a számítás N\*log2(N) \* (tmul/2 + tadd) lépésszámból teljesíthető.

Ami például egy 1024 pontos transzformációnál nagyságrendileg század akkora processzorteljesítmény felhasználását teszi szükségessé.

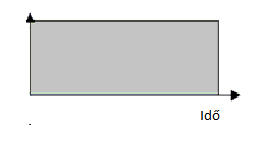
Az FFT egyetlen korlátja, hogy a pontszám nem lehet tetszőleges, például nem lehet 1000 pontos, csak 2 valamely hatványa lehet.

# Jelsorozat ablakozása [3]

Mivel beszédfeldolgozásnál nem periodikus jellel dolgozunk, csak kvázi periodikussal, a FFT transzformáció előtt ablakozni kell a jelet. Az ablak csak a jel egy kisebb időintervallumát engedi át, a többit nullára állítja. A jel ablakozása során ezt az ablakot toljuk mindig arrébb, és az ablakozott jelnek számoljuk ki a spektrumát. Az ablak eltolása általában az ablak szélességének a felével történik.

## A téglalap ablak

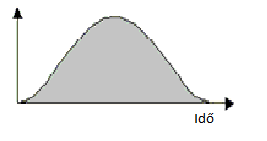
A legegyszerűbb ablak a téglalap ablak, ez egy meghatározott intervallumon belül egyes értéket vesz fel azon kívül mindenhol nulla. Hátránya, hogy ha a jel nem tökéletesen periodikus, akkor a széleinél a jel torzul, és így a spektrumban is torzulások léphetnek fel.



* + 1. ábra: A téglalap ablak

## Hann ablak

A Hann ablaknál nem lép fel a spektrum torzulás, mint a négyszög ablaknál, mivel a Hann ablak fokozatosan 0-hoz tart az ablak széleihez haladva. Képlettel felírva az N széles ablak:



**4.2.1. ábra: A Hann ablak**

## Ablakozás a programban

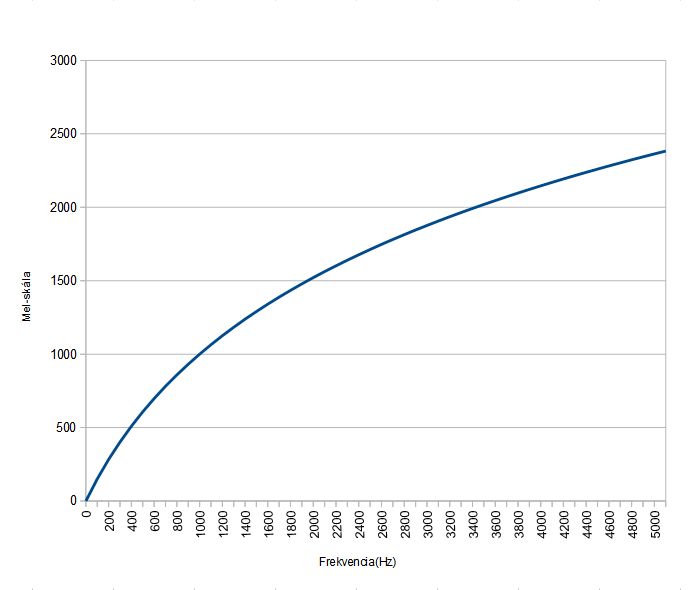
Mivel a beszéd során nagyon gyorsan változik a spektrum, ezért kis ablak szélességet kell választani, hogy az egyes hangok spektruma ne olvadjon össze. A program mintavételezési frekvenciája 11025Hz, az ablakméretet 512 mintának az ablak eltolását 256-nak választva másodpercenként körülbelül 43 spektrumot kapunk, ami elégségesnek bizonyult a szavak felismeréséhez. Ablakozó függvénynek a Hann ablakot választottam.

# Mel-skála [2]

Az emberi hangérzékelés a frekvencia függvényében változik, ez a fülünk felépítése miatt van, ugyanis a csigában a szőrsejtek eloszlása a magasabb frekvenciák felé ritkul. Míg 1000Hz-alatt az érzékelésünk nagyjából lineáris, efölött a frekvencia és az észlelés helye közötti összefüggés logaritmikus. Ennek megfelelően az alacsonyabb frekvenciák észlelése jobb felbontású, mint a magasabbaké. Kísérletek során tesztelték, milyen frekvenciájú tiszta szinuszos hangokat érzékelünk egymástól azonos távolságra lévőnek. A kapott értékeken alapszik a mel-skála. Mivel az ember hallása az emberi beszéd megértésére kiváló, ezért célszerű ezt utánozni. Ez a skála tulajdonképpen azt mutatja, hogy a tartalomra jellemző információ a felismerés szempontjából milyen sűrűséggel helyezkedik el a frekvenciatengely mentén.

A frekvencia és mel-skála közti összefüggés képlete:

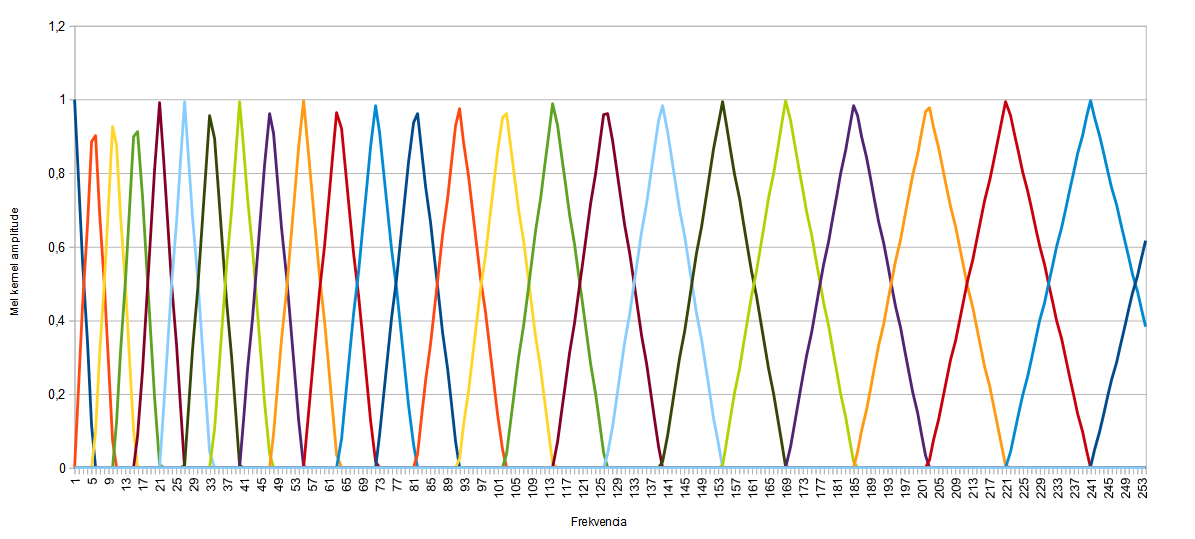
A képlet megmutatja, hogy az f frekvencia hol helyezkedik el a mel-skálán.



5.1. ábra Frekvencia transzformáció mel tartományba ábrázolva

A frekvencia mel-skála transzformációt a gyakorlatban egymást félig átfedő sávszűrőkkel lehet megvalósítani, melyek egyenlő szárú háromszög alakúak, és a frekvencia növekedésével a mel-skálának megfelelően szélesednek, ezeket mel-skála szerinti sávszűrőknek hívják. A sávszűrők száma meghatározza a mel-spektrum felbontását. A mel-spektrum elemeit úgy számoljuk ki, hogy összegezzük az egyes sávszűrők alatti spektrum értékeket.

Lejjebb látható a programban használt mel-sávszűrők ábrázolva. A megvalósításhoz 30 sávszűrőt használtam, ezzel 30-re csökkent a spektrumhoz tartozó paraméter vektor, az eredeti spektrum 256-os méretéhez képest.



5.2. ábra: A programban használt mel-szűrő keretek

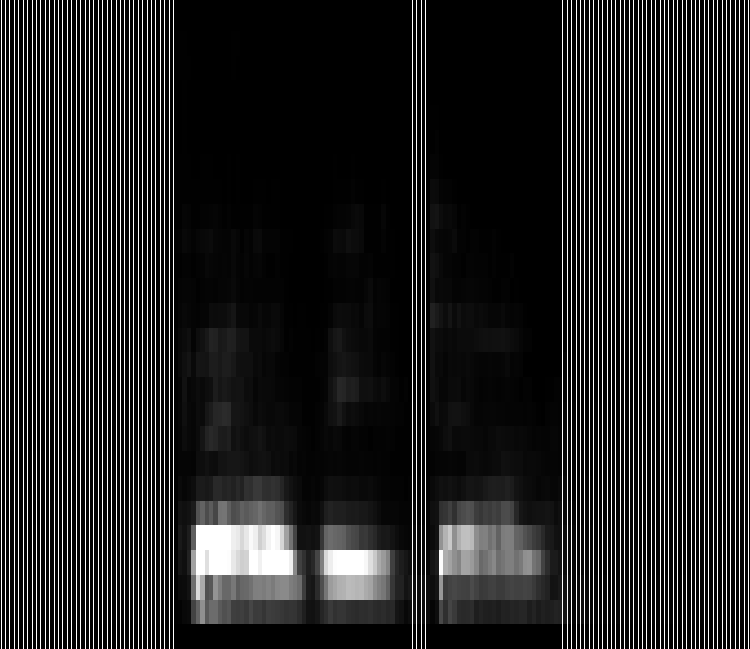
# Kezdő- és végpont detektálás

A beszéd kezdő- és végpont detektálása azon idő intervallumok meghatározását jelenti, amik alatt beszéd hangzik el. A beszédhang detektálását a mel-spektrum átlag amplitúdójának vizsgálatával határozom meg. A küszöbértéket a program futása során dinamikusan határozza meg. Ha a küszöbértéknél kisebb a jelenlegi amplitúdó, akkor a küszöbérték felveszi ezt a kisebb amplitúdót, ha nagyobb, akkor lassan elkezd nőni a nagyobb érték felé. Ezzel az eljárással a küszöbérték beáll a statikus háttérzaj értékére. A beszéd detektálása során a küszöbérték párszoros többszörösét veszi a program figyelembe, hogy a nagyon kis kiugrásokra ne figyeljen oda.

Zavaró tényező lehet a rövid ideig tartó nagy amplitúdójú zajok. Ezeket a zavarokat, ki lehet szűrni azzal, ha a szavaknak megadunk egy minimális időt, amíg a küszöbérték felett kell tartaniuk az amplitúdót, hogy a kezdőpont detektálódjon.

Ennek a zavarnak a párja, amikor egy háttérzaj túl hangos és túl sokáig tart, ilyen zajt kelthet az utcán elmenő autó. Ezeket a zavarokat figyelmen kívül hagyhatjuk, ha megadunk a szavaknak egy maximális hosszt, amibe be kell férnie.

A 6.1. ábrán látszik a kezdő és végpont detektálás a "felvétel" szón, a fehér vonallal jelölt részeket veszi a program csendnek. Látszik az is hogy a szón belül van egy csendes intervallum, ez a 't' betű előtti csend miatt van. Ezt a problémát azzal lehet megoldani, hogy a szavak végén várni kell adott hosszúságú csend intervallumot, hogy a szó végét detektálhassuk, ha ezen intervallumon belül megint a küszöbérték felé kerül az amplitúdó, akkor folytatjuk a szóvég keresését.



6.1. ábra: A "felvétel" szón a kezdő és végpont detektálás

## A mel-spektrum normalizálása

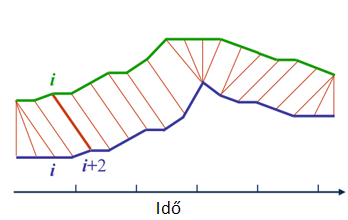
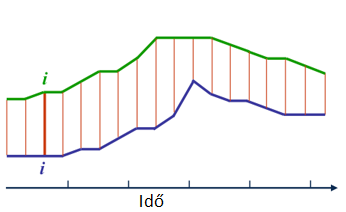
Miután meghatározta a program a szavak kezdő és végpontját, a szót reprezentáló paramétereket normalizálni kell. Erre azért van szükség, mivel nem mindig ugyanakkora hangerővel ejtjük ki a szavakat, más lehet a mikrofon helyzete és így nem ugyanolyan intenzitású paraméter vektorok jönnek létre. A szó normalizálása során a program leosztja az összes paramétert a paraméterek összegével, ezzel előállítva a hang intenzitás független paraméter vektorokat.

# Dinamikus idővetemítés

Az emberi beszédnek van egy olyan tulajdonsága, hogy soha nem mondjuk ki ugyanazt a szót kétszer tökéletesen megegyezően. Mindig van egy kis eltérés az ejtésünkben, nem egyforma ideig ejtjük ugyanazt a magánhangzót, nem ugyanazon az alaphangon beszélünk. Ezért amikor egy szót a jelleg vektorából fel szeretnénk ismerni, számolni kell az időbeni és a spektrumbeli eltérésekkel. Ezeknek az eltéréseknek a számítására szolgál a dinamikus idővetemítés algoritmusa, rövid nevén DTW (Dynamic Time Warping).

A DTW[1] eljárás bemenetén két vektor kap, kimenetén pedig a lehető legjobb idővetemítés mellett a két vektor közötti távolságok összege. A 8.1. ábra egy példát mutat két vektor távolságának kiszámítására. Az ábra zölddel és kékkel jelöli a bemeneti minta sorozatokat, a piros vonalak az összehasonlított minta párokat. Baloldalon látható, hogy végeznénk el az összehasonlítást elemenként, ezzel a módszerrel a megegyező indexű elemekből számítanánk a két vektor távolságát.

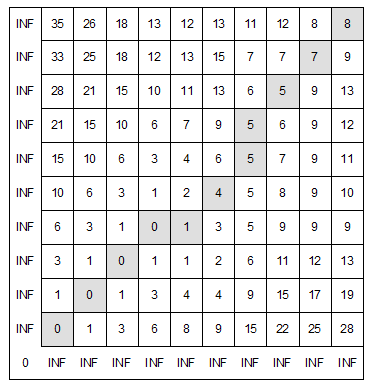
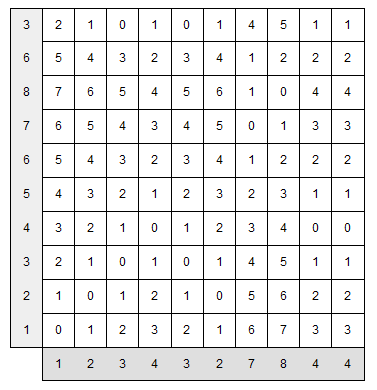
Jobboldalon az idővetemítés módszer látható, ahol a két vektor elemeit a másik vektor összes elemével összehasonlítva kapjuk meg a legoptimálisabb vetemítés melletti távolságot. Az ábrán látható, hogy a piros vonalak nem metszik egymást, és minden elemhez tartozik a másik mintahalmazból egy elem.



8.1. ábra: Baloldalt az elemenkénti távolság,  
jobb oldalon pedig az idővetemítéses távolságra látható példa.

Az idővetemítés problémáját dinamikus programozással lehet megoldani. Legyenek a függvény bemenetei *n* és *m* hosszú mintasorozatok. A megoldáshoz használjunk egy -es D (distance) mátrixot, amelyben az egyes minták távolságát tároljuk el, ezután a D mátrixban keressük meg a minimális költségű utat D[1,1] és D[n,m] között. Az út keresése során fontos szabályok, hogy minden lépésnél csak előre felé haladhatunk, tehát ha [i, j]-ről lépünk [] -re akkor:

a másik szabály, hogy az út folytonos, mindig csak a szomszédos cellákra léphetünk.



8.2. ábra: Baloldalon a DTW távolság tömb, jobb oldalon a DTW útkeresés eredménye.

Az 8.2. ábra a DTW algoritmus futását szemlélteti, baloldalon a távolság tömb, szürke mezőkben vízszintesen, és függőlegesen a bemeneti sorok, jelöljük ezeket a-val és b-vel. A cellákban pedig a bemenetek távolságának abszolút értéke:

A jobboldali ábrán a legkisebb út meghatározás algoritmusának eredmény látszik, a   
[0, 0] pontban nullára, és az első sort és oszlop elemeit végtelenre inicializáljuk. Ezután bejárjuk a tömböt elemről elemre, és meghatározzuk minden indexbe vezető legkisebb költségű út súlyát. Ha W-vel jelöljük az út számító tömböt, akkor

ahol

D[i, j] Az adott pontban lévő távolság értéke.

W[i-1, j-1] Az egyezés költsége, ha az összehasonlításban előre lépünk.

W[i-1, j] A beillesztés költsége, ha az előző mintát újra beillesztjük az idő sorba.

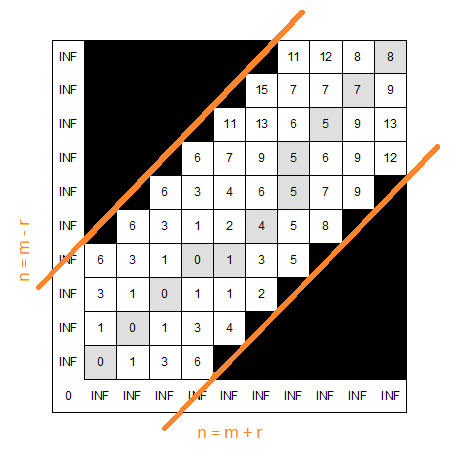
W[i-1, j] A törlés költsége, ha az adott elemet kitöröljük.

Az algoritmus futás végén a W[n, m]-edik tömbelemben található a legkisebb költségű út értéke.

Hátránya az algoritmusnak, hogy n és m méretével exponenciálisan nő a futási idő, mivel az *a* tömb elemét összehasonlítja *b* tömb minden elemével.

## A DTW keresési tér csökkentése

A DTW exponenciális futási idő csökkenthető, ha korlátozzuk a keresési teret, ezt az n/m meredekségű egyeneshez képest r (radius) eltérés megadásával adhatjuk meg. Ezzel a n\*m-es összehasonlítások száma n\*(2\*r)-re csökkent. Ez látható a 8.1.1. ábrán, a fekete cellák jelölik a keresésből kihagyott elemeket, amik a keresési térből kiesnek.



8.1.1. ábra: a keresési tér korlátozása

## A DTW használata a programban

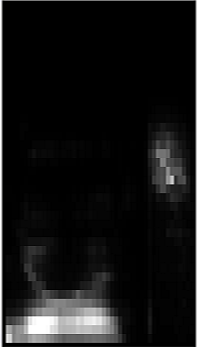
A program a szavakat mel-spektrumok sorozataként tárolja, ezért két dimenziós tömbként ábrázolható, és időben és spektrumban is lehetnek eltérések a két spektrum sorozat között. A kétdimenziós tömbök közötti összehasonlítást két egymásba ágyazott DTW algoritmussal valósítottam meg. A belső DTW a mel-spektrumok távolságát határozza meg, míg a külső, a belső DTW által megadott távolságokon számolja ki az optimális vetemítést, és adja meg a minimális költségű utat.

Mind kettő DTW-nek külön állítottam be egy-egy rádiuszt, amely korlátozza a szavak és spektrumok maximális vetemítési mértékét.

## A DTW gyorsítása a mel spektrumon

A spektrum ábrázolása során megfigyelhető, hogy az emberi beszédre jellemző paraméterek nagyrészt a spektrum alján helyezkednek el. A spektrumból kivonva a statikus zaj spektrumát, és a spektrum statikus zajnál kisebb részeit nullának vehetjük, ezek úgysem hordoznak lényegi információt.

Ezzel a spektrum felső részei nulla értéket vesznek fel. Ha ezeket a nulla értékeket levágjuk, és a spektrum hosszát az utolsó nem nulla érték indexénél határozzuk meg, jelentősen csökkenthetjük a keresési teret, amit az útkereső algoritmus bejár, ezzel gyorsítva az algoritmus futását.



8.3 1. ábra: a "nyolc" szó, paraméter vektora

Az 8.3.1. ábrán a "nyolc" szó szín spektruma látható, az ábra jobb szélén a spektruma 'c' betű paraméterei, ami a spektrum felső felén helyezkednek el. Látható hogy felesleges processzor időt fecsérelnénk el, ha a spektrum felső részeit is kielemeznénk.



8.3.2. ábra A programban két mel-spektrum összehasonlításának eredménye

Az 8.3.2. ábrán két mel-spektrum a legkisebb út keresése látható, a spektrumok felső részét nullák alkotják, ez látható a kép jobb felső sarkában lévő szürke téglalapban, ugyanis itt az utak költsége már nem nő a 0 távolságok miatt, ezért helyes eredményt ad, ha a nulla sorozatok kezdő indexeken lévő út értékével határozzuk meg a legkisebb utat.

## A keresés optimalizálása szó szinten

A program szavak paraméter vektorainak listáit tárolja el, minden egyes intervallumon, amit megfelelő ideig átlépi a beszédküszöböt, és szónak minősül, le kell futtatni a keresési algoritmust, amely összehasonlítja a hallott szó paraméter vektorát a tárolt paraméter vektorokkal. A kielégítő válaszidő szempontjából fontos hogy ezt a keresést a lehető leggyorsabban hajtódjon végre.

A keresést gyorsíthatjuk, ha kizárunk bizonyos tárolt paraméter vektorokat, amelyekre nem végezzük el az összehasonlítást. Kizáró feltétel lehet:

* A szavak hosszának abszolút különbsége túl távol van.
* Az átlag amplitúdók távolsága átlép egy bizonyos küszöb számot.
* A szavak távolsága összehasonlítás alatt túl távol került.
* A szavak hossz szerinti keresése (Logaritmikus kereséssel)

Az első és leggyorsabban végrehajtható kizárási feltétel, ha a szavak hosszát hasonlítjuk össze. Ebben az esetben megadunk egy küszöbszámot, amelyet ha a szavak abszolút távolsága meghalad, akkor nem végezzük el az összehasonlítást.

A paramétervektorokat egyszerűsíthetjük, az egyes paraméterekhez tartozó átlag amplitúdóval. Ez az amplitúdó minta hossza megegyezik a szót reprezentáló paraméter vektor hosszával, de csak egy értéket tartalmaz. Az amplitúdó vektor a paraméter vektor egy egyszerűsített, kisebb felbontású alakja, mellyel gyorsan végezhető egy összehasonlítás, a teljes paraméterek összehasonlítása előtt. Ha az amplitúdók idővetemített távolságát lekorlátozzuk egy felső határral, aminél nagyobb a kiszámított érték, akkor megszakíthatjuk az időigényes teljes távolság kiszámítását.

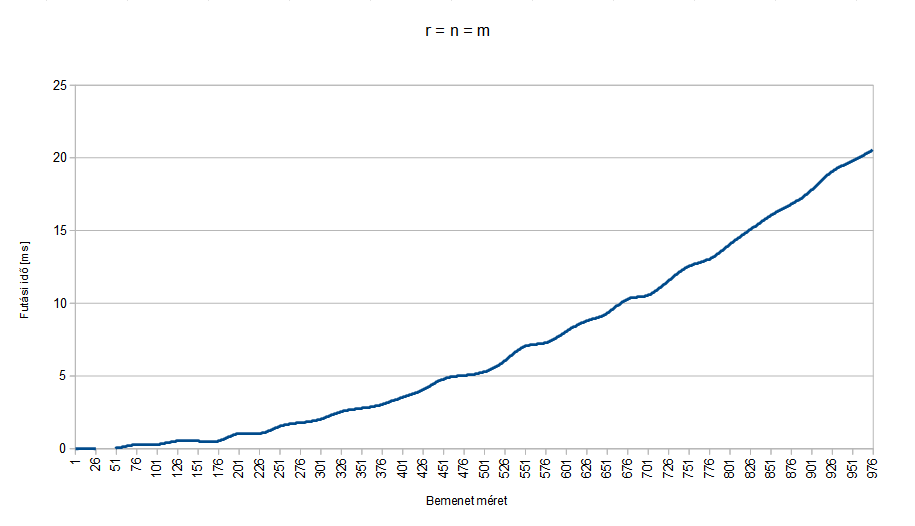
A harmadik lehetőség, amit kihasznál a program, hogy idővetemítés során folyamatosan számon tarthatjuk a legkisebb költségű úthoz tartozó értéket, amit az egyforma úthosszakhoz tartozó minimum értékekből kaphatunk meg, és tudjuk, hogy a következő lépésben ehhez az értékhez képest a következő távolságra nagyobb vagy egyenlő lehet az utak költsége. Ha keresésen belül eltároljuk a legkisebb szó távolságát, és ha keresés közben a legkisebb út költsége e távolság felé ér, akkor megszakítjuk az adott szó feldolgozását.

Némi javulás érhető el a keresésben, ha a szavakat hosszuk szerint sorba rendezzük így a rövid parancsszavakat gyorsabban átnézi a program, és hamarabb ad eredményt.

Az utolsó optimalizálás, ami a rendezett szavakra épül, az a logaritmikus keresés algoritmusát használja ki. Ezzel az algoritmussal logaritmikus időn belül kereshetjük meg azt a szót, aminek a hossza a megengedett szótávolság minimumán helyezkedik el. Ezzel megkapva a szavakat tároló tömb kezdő indexét, aminél az összehasonlításokat kezdeni kell. Az összehasonlításokat addig végezzük, amíg a tárolt szavak hossza a megengedett felső szóhossz határ fölé nem megy.

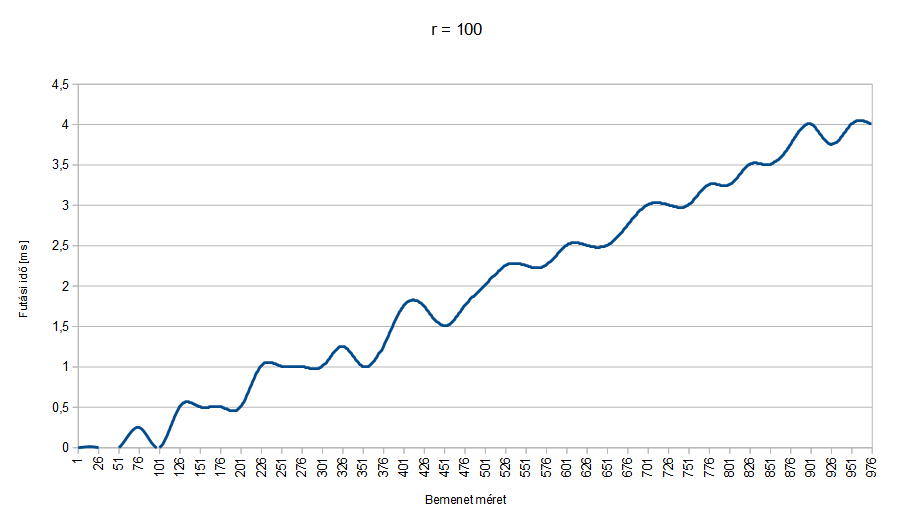
## DTW algoritmus tesztelése

A hangfelismerő rendszer leginkább processzor igényesebb része a keresési algoritmus, azon belül is a dinamikus idővetemítést megvalósító rész. Ezért fontos tesztelni ezt a modult, hogy ismerjük a rendszer határait, amin belül képes értelmes válaszidőn belül eredményeket szolgáltatni.



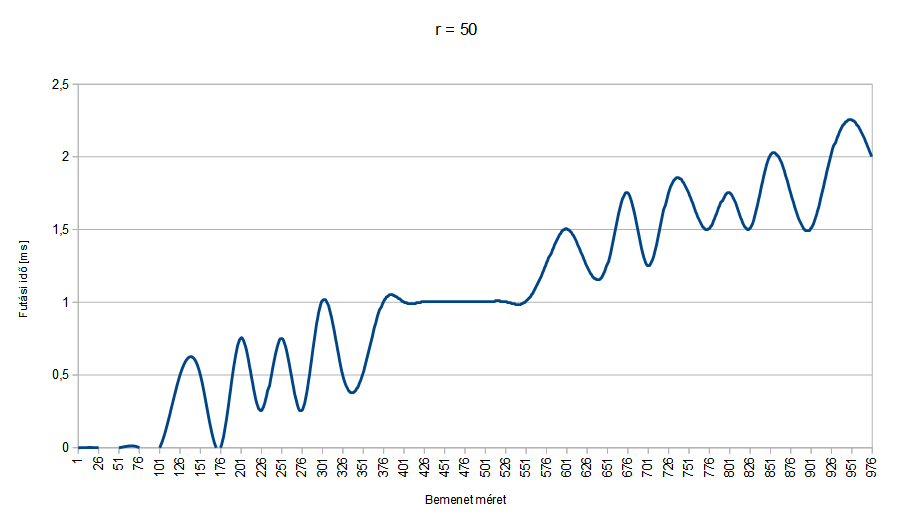
8.5.1. ábra: DTW futás idő teszt, ha a rádiusz megegyezik a bemenő minták hosszával.

Az első DTW teszt látható az 8.5.1. ábrán, vízszintes tengelyen a bemenő minták mérete, függőlegesen a futási idő milliszekundumban. Látható, hogy a bemenő mintákhoz képest a számítási idő exponenciálisan növekszik. Ez azért van, mert *a* bemenő minta sorozat elemét összehasonlítja *b* mintasorozat minden elemével. A legnagyobb bemenő mintahalmaz körülbelül 1000 mintából áll, ami 1 millió lépésben járható be, ez körülbelül 20ms-ig tart.



8.5.2. ábra A rádiuszt lecsökkentve 100-ra a futási idő gyorsul.

A 8.5.2. ábrán annak eredmény látszódik a futás időn, hogy lecsökkentettem a rádiusz értékét 100-ra, ezáltal a futási idő lineárisra vált. Látható az első ábrához képest hogy a maximális futás idő ötödére csökkent, ez azért van, mert az 1000 nagyságú minta sorozatot csak 2\*r = 200 szélességben hasonlítja össze így 1000 / 200 = 5 -ből megkapjuk a csökkenés mértékét. A kisebb hullámok a több feladatos operációs rendszer megszakításai miatt jönnek létre.

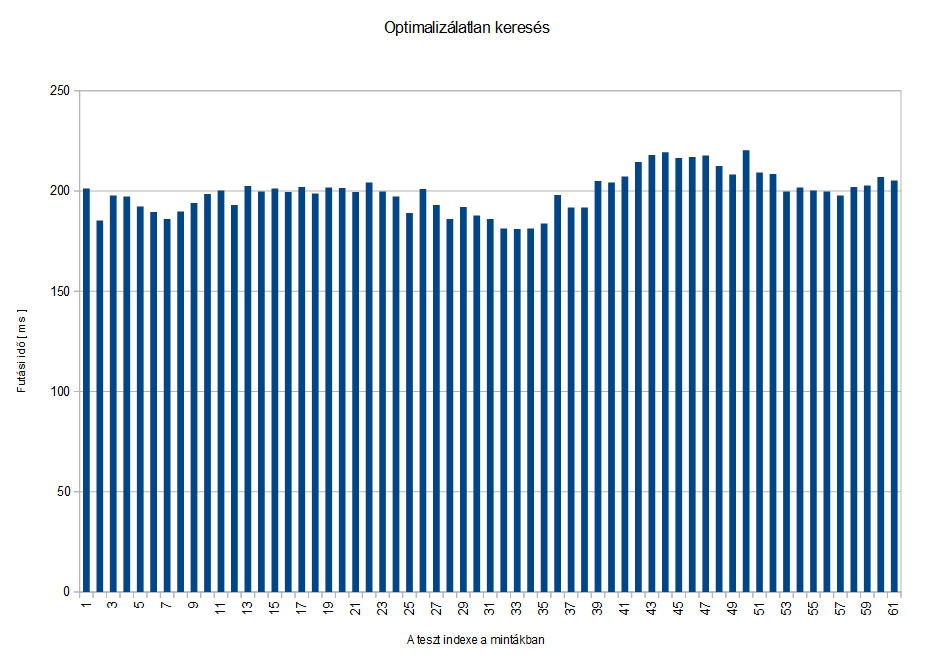


8.5.3. ábra: A rádiuszt lecsökkentése 50-re

Ha tovább csökkentjük a rádiuszt egyre gyorsabban fog lefutni az összehasonlítás, 50 rádiusz esetén már két ezer hosszú minta összehasonlítása 2 milliszekundumig tart. Ez látható az 8.5.3. ábrán.

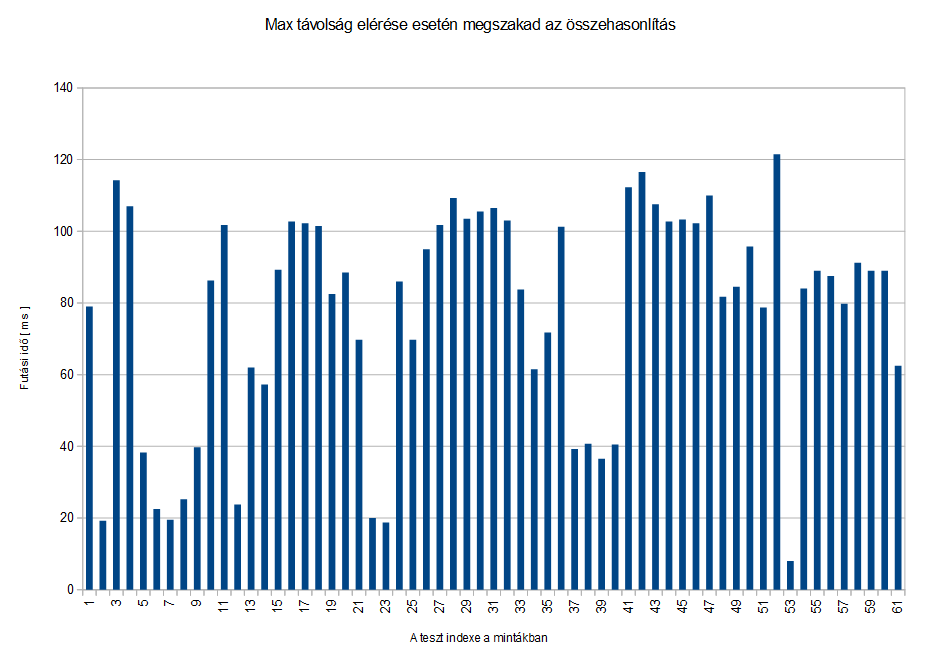
## A szószintű idővetemítés tesztelése

Ebben a részben a bejövő paramétervektorok összehasonlítását a tárolt paraméter vektorokkal végző modult tesztelem futásidőre. Tesztnek megfelel, ha a tárolt paraméter vektor elemeire lefuttatom az összehasonlítást. Az optimalizálás fázisait különböző ábrákon jelenítem meg.



8.6.1. ábra: optimalizálatlan keresés

Az 8.6.1. ábrán a teljesen optimalizálatlan keresés látható, a bemenő paraméter vektort összehasonlít minden tárolt paraméter vektorral. Az ábrán látható, minden összehasonlítás nagyjából konstans (200ms) idő alatt fut le.

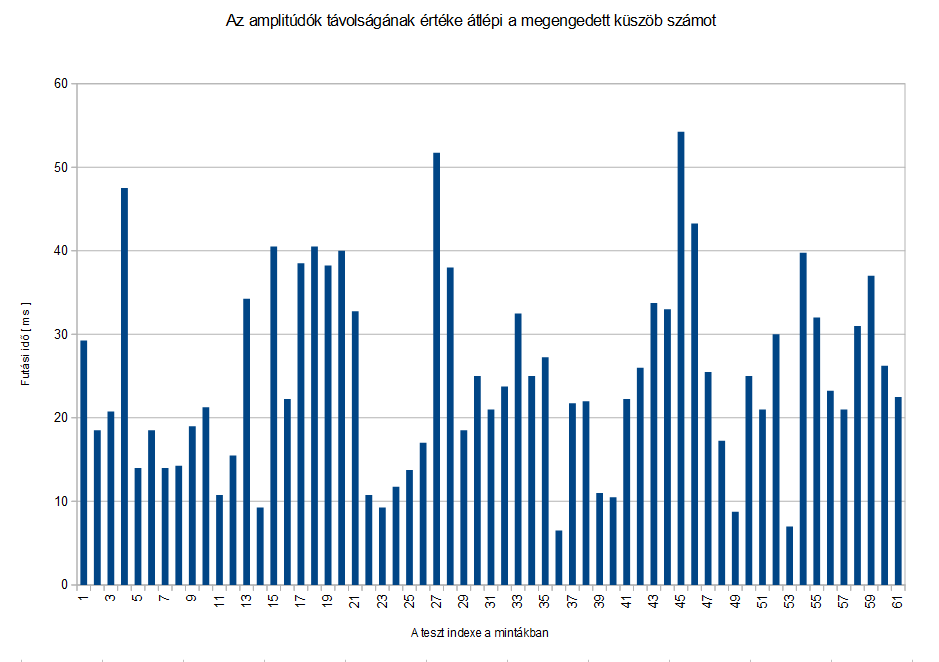


8.6.2. ábra: A maximum távolság esetén megszakad az összehasonlítás.

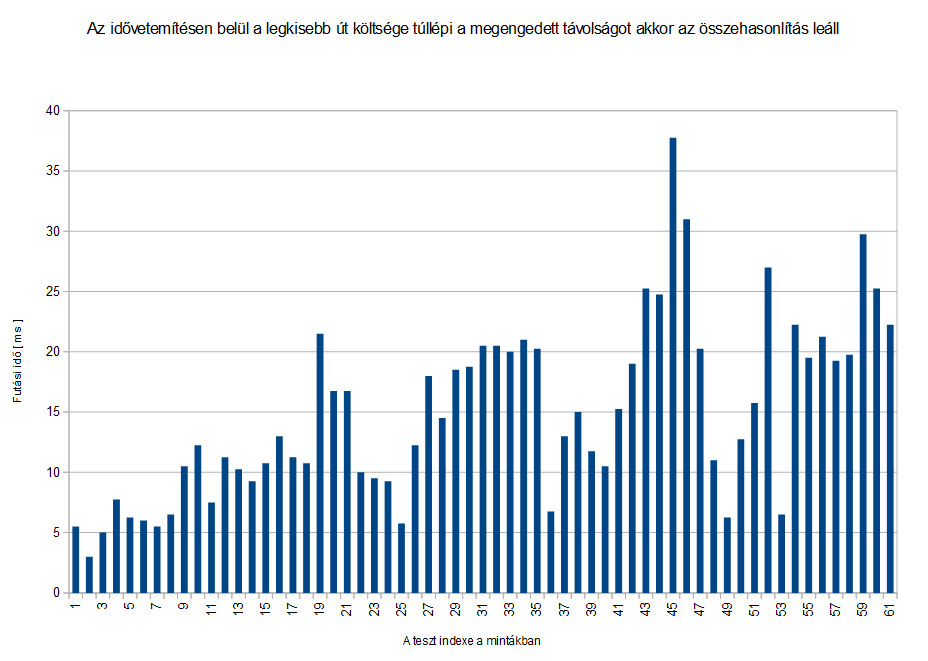
Az 8.6.2. ábrán a maximum szóhossz szerinti szűrés eredménye látható. például az 53-as szó hiába van a lista végén, egyedi hossza miatt kevés szóval hasonlítódik össze ezért gyorsan eredményt kapunk. Látható az is, hogy a többi esetben is jóval a 200ms alá csökkent a futási idő. Átlagosan 77ms-ig futottak a keresések. A legtovább tartó keresés 125ms-ig tartott.

Az 8.6.3. ábrán az látszik, hogy az amplitúdók távolságának túl mesze kerülése miatt még több szó kiesik az összehasonlításból. Ennek eredményeként az átlag összehasonlítási idő 25ms alá csökken. A legtovább tartó összehasonlítás 55ms-ig tartott.

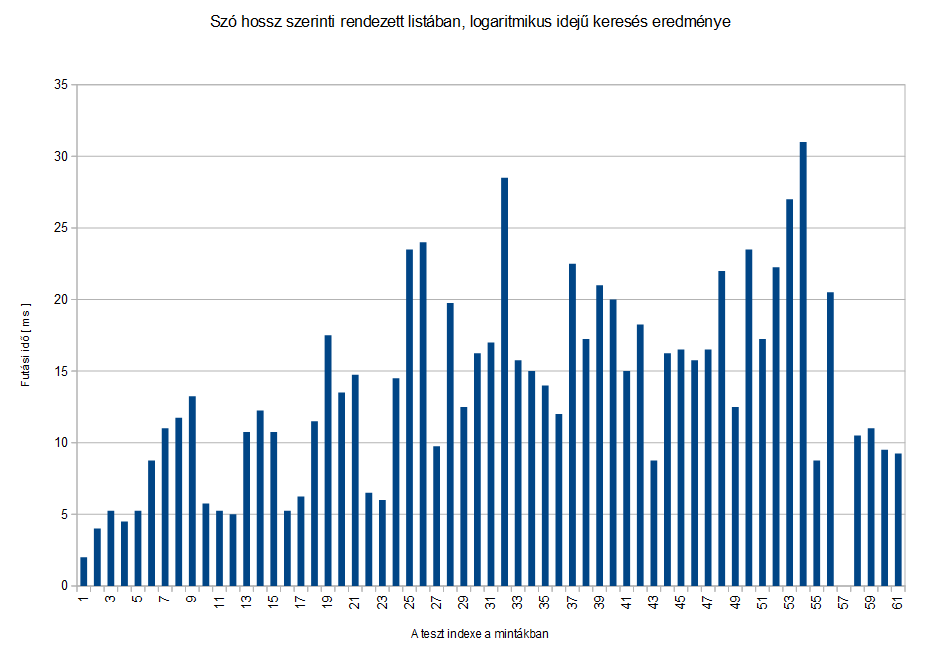
Az 8.6.4. ábrán az látszik, hogy az idővetemítés közben legkisebb költségvetésű út értéke nagyobb lesz a globális minimum költségnél, ekkor az összehasonlítás megszakad, és az aktuális legkisebb költségű út értékével tér vissza. Látszik az ábrán, hogy ez lineárisan befolyásolja a futási időt, a megtalált minta pozíciójának megfelelően. Mivel ha a keresés során az első összehasonlítással egy kis távolságot kapunk, akkor a későbbi összehasonlítások nem fognak végig futni, csak amíg át nem lépi a legkisebb út távolságuk a minimális út távolságát. Ezzel a módszerrel az összehasonlítás átlagos időtartama 15ms-ot vett igénybe. A legtovább tartó összehasonlítás 37ms-ig tartott.



8.6.3. ábra: az amplitúdó összehasonlításos optimalizálás eredménye



8.6.4. ábra: Az idővetemítés közben legkisebb költsége vetésű út értéke nagyobb lesz a globális minimum távolságnál, ekkor az összehasonlítás megszakad



8.6.5. ábra: Szó hossz szerinti rendezett listában, logaritmikus idejű keresés eredmény

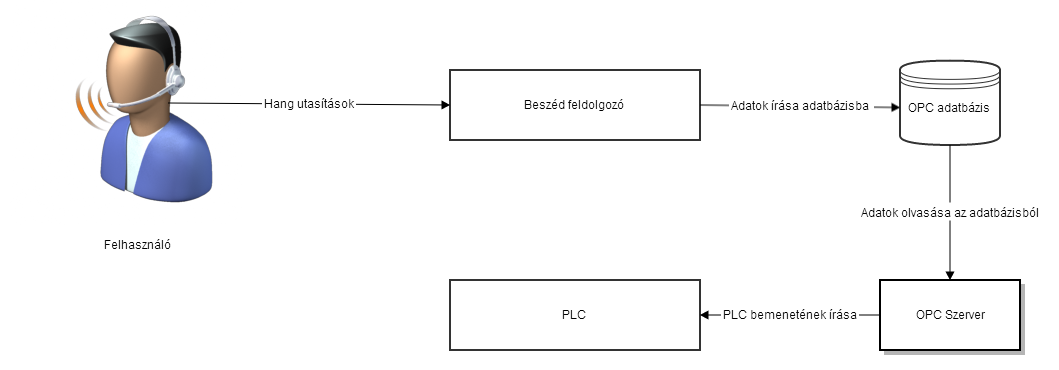
Ha a lineáris szó keresést átváltjuk logaritmikus idejű algoritmusra, sokkal gyorsabban megtalálhatjuk a megfelelő hosszúságú szavakat, és értékes processzor időt nyerhetünk. Ez látható az 8.6.5. ábrán. Az átlagos futási idő 13ms-ra csökkent. A maximális futás idő 31ms-let.

## Keresési tesztek értékelése

A keresési teszteket a laptopon végeztem el, amely energiatakarékos módban futott 1,18GHz órajel sebességen. A keresési teszt során egy 40 szóból álló tesztadatbázist használtam, melyben egyes szavakhoz több paramétervektor is tartozott a felismerés pontosságának javítása céljából, így állt a mintaadatbázis 61 elemből. Sikerült a programnak ezen adatbázison, átlagosan 13ms-on belül eredményt hoznia. Ez elhanyagolható idő ahhoz képest, hogy körülbelül 400ms-omot kell várni a szó kimondása után, hogy biztosan detektálhassuk a szó végét, és ne vegyük a „kettő” szót két különálló szónak. De még ennyi késleltetéssel is tekinthető a program valós idejű alkalmazásnak, mivel a felhasználó szinte azonnali válasznak érzékeli a gép reakcióját.

# Program terv

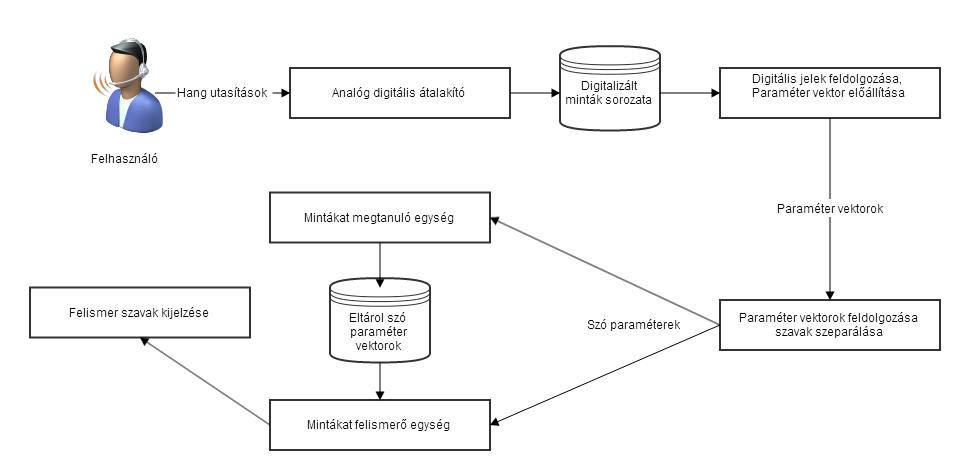
A végcél egy PLC vezérlésének lehetővé tétele a programból. A számítógép és a PLC között az OPC szerverrel teremt kapcsolatot, mely adatokat ír az PLC bemenetére. Mivel az OPC szerver és az irányítani kívánt PLC típusa és paraméterei változhatnak, ezért a programot úgy valósítottam meg, hogy univerzálisan lehessen használni, bármilyen OPC szerverrel. Ezt az univerzitás egy MySQL adatbázis közbeiktatásával lehet elérni, amelybe a program adatokat tud beírni, és az OPC szerver kiolvasva azokat, beállítja a PLC bemenetét. A beszédfelismerő programmal hozzá lehet csatlakozni a MySQL adatbázishoz, a vezényszavakhoz SQL utasításokat lehet rendelni, amelyek lefutnak a beállított szó felismerésekor. Az SQL utasítások tetszőlegesen módosíthatók, ezért bármilyen struktúrájú adatokat be lehet írni az adatbázisba. A 8.1. ábrán a program működésének sematikus képe látható. A felhasználó hang utasításokat ad, melyeket a beszéd feldolgozó egység feldolgoz, és végre hajtja az adatbázison az SQL parancsokat, majd a beírt adatokat az OPC szerver kiolvassa, és beállítja a PLC bemenetét.



8.1. ábra: A program működésének terve.

# A beszédfelismerő rendszer felépítése

A beszédfelismerő rendszer, egymásra épülő adatfeldolgozó modulokból épül fel. Minden modul az előtte lévő modultól kap, és az utána lévő modulnak adja tovább a feldolgozott adatokat. A 9.1. ábrán láthatóak a program fő moduljai. Az első modul a hang digitalizálásért felelős, amely egy globális tömbben tárolja el a digitalizált mintákat, egy FIFO sorban. Amikor a digitalizáló beállítja a tömb utolsó elemét, átugrik a tömb elejére és ott folytatja a minták mentését. A következő modul a paraméter vektorok előállítását végzi. Kiolvassa a digitalizált mintákat a FIFO tömbből, figyelve, hogy ne előzze meg a kiolvasott minta az utoljára beírt mintát. A harmadik modul a paraméter vektorokat feldolgozva, előállítja az egy szóhoz tartozó paraméter vektor listát, amit a rendszer állapotától függően, hogy tanuló vagy felismerő állapotban van e, továbbküldi a tanuló vagy felismerő modulnak.



9.1. ábra Az adatfeldolgozó modulok

A tanuló modul elmenti a felismert szó paraméter vektorait egy globális listában. A felismerő modul pedig a megtanult minták közül megkeresi a legközelebbit és kiírja a megtalált szót.

## Analóg digitális átalakító

A Qt Creator szerencsére magas szintű programozói környezetet biztosít a programozóknak, ezért a hangfelvételt egy osztállyal a QAudioRecorder-rel valósíthatjuk meg. A Qt objektumok signalok-kal tudják a kapcsolatot egymással tartani. Egyik objektum signal-ját össze lehet csatolni egy másik osztály slot-jával, ami egy függvény, a signal-nak megfelelő paraméterekkel. Amikor az egyik objektum emit-tálja a signal-ját bizonyos paraméterekkel, a másik objektum slotja képes azt feldolgozni. Hangrögzítés során is ez történik, a QAudioRecorder osztály példányát megfelelően inicializálva és a felvételt elindítva, szabályos időközönként emit-tálja az audioBufferProbed (QAudioBuffer) signal-ját amit egy másik objektummal fel tudunk dolgozni. A programban erre a célra egy ˝Recorder˝ nevű osztályt hoztam létre, amely egy interfész felületet valósít meg a QAudioRecorder és a főprogram között, és teljesen lefedi a 9.1. ábrán lévő analóg digitális átalakító funkcióit. Ez az osztály inicializálja a QAudioRecorder-t és rögzíti a felvett mintákat. A felvett mintákat egy globális konstans hosszú FIFO sorba írja be, egy másik globális változóban elmenti, hogy melyik elemet írta be utoljára, hogy a digitalizált elemeket használó következő réteg nehogy túl olvasson rajta.

### Az analóg átalakító kódja

void Recorder::processBuffer(const QAudioBuffer& buffer) {

// Digitalizált mintákat átalakítása short tömbbe

const short\* data = buffer.constData<short>();

// A minták hosszának lekérdezése és eltárolása

uint length = buffer.sampleCount();

// A g\_recordedSamples globális tömbben, hozzáfűzi a mintákat.

// A g\_recordedSamplesLastSetted-ben eltárolja az utoljára

// beállított minta indexét.

uint dc = 0; // data counter

while (dc != length){

g\_recordedSamples[g\_recordedSamplesLastSetted] = data[dc];

g\_recordedSamplesLastSetted ++;

if (g\_recordedSamplesLastSetted == RECORD\_SAMPLE\_SIZE) {

g\_recordedSamplesLastSetted = 0;

}

dc++;

}

// Emittál egy signal-t, ami jelzést küld a főprogramnak, hogy

// rendelkezésre állnak újabb digitalizált minták.

emit recording();

}

A programkódból kiderül, hogy a felvett minták a g\_recordedSamples tömbbe kerülnek, ami a 9.1. ábrán a ˝Digitalizált minták sorozata” nevű adat tárolónak felel meg. Ennek a tárolónak a méretét globálisan a RECORD\_SAMPLE\_SIZE konstans határozza meg.

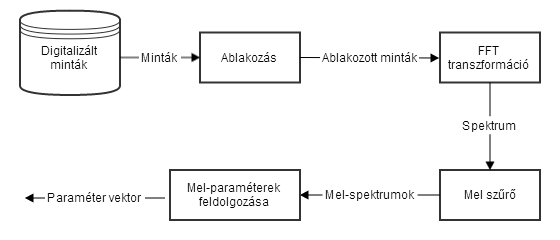
### Paraméter vektor előállítása

A paraméter vektor a következőkből áll:

* a minták mel-spektrumából
* a mel-spektrum átlag amplitúdójából
* a mel-spektrum hosszából
* egy bool értékből, ami tárolja, hogy a spektrum átlag amplitúdója, átlépte-e a beszédhez tartozó hangerősség küszöböt.

Ahhoz, hogy a paraméter vektort előállíthassuk a digitalizált mintákból módosításokat kell végrehajtani az adatokon. A digitalizált minták feldolgozásának fázisai:

* minták ablakozása
* ablakozott minták spektrumának meghatározása
* a spektrum mel-szűrővel feldolgozva, előállítani a mel-spektrumot
* mel-spektrum elemzése, beszéd küszöb meghatározása



9.1.2.1. ábra: Paraméter vektor előállításának lépései

A paraméter vektorok előállítását a VoiceProcessor osztály végzi, a főprogram a processSamples metódusát hívja meg, amely a következő feldolgozandó ablakozott jel paraméter vektorát adja vissza.

### processSamples forráskódja

Parameter\* VoiceProcessor::processSamples() {

// Az fftBuffer tárolja az ablakozott jelet, amit a getFFTBuffer()

// állít elő

float \*fftBuffer = getFFTBuffer();

// Ha nem áll rendelkezésre elég minta akkor NULL-al tér vissza a

// függvény

if (fftBuffer == NULL) {

return NULL;

}

// A spektrum kiszámolása és eltárolása

float \*spektrum = getSpektrum(fftBuffer);

// Spektrumból a mel-spektrum kiszámítása, ami már egy paraméter

// vektorral tér vissza

Parameter \*parameter = getMelSpektrum(spektrum);

// Paraméter vektor elemzése, beszéd küszöb megállapítása

bool inWord = checkInWord(parameter);

// visszatérés a paraméter vektorral

return parameter;

}

* getFFTBuffer() függvény

A globális tömbből ablakozza a mintákat, és feltölti a fft buffert. Miután az fft buffer megtelt az ablakot fél ablakmérettel tolja el, biztosítva az ablakok átfedését.

float\* VoiceProcessor::getFFTBuffer() {

// statikus FFT buffer

static float fftBuffer[WINDOW\_SIZE] = {};

// utoljára beállított FFT buffer indexe

static uint lastSettedFFTBufferIndex = 0;

// Utoljára olvasott globális minta buffer indexe

static uint lastProcessedSampleIndex = 0;

// Ha az utoljára feldolgozott minta indexe megegyezik az utoljára

// kiolvasott minta indexével, akkor nem került új adat a felvett

// minták tömbjébe, a függvény NULL-al tér vissza

if (lastProcessedSampleIndex == (\*m\_recordedSamplesLastSetted)) {

return NULL;

}

// Amíg nincs feltöltve a FFT buffer, addig olvassa a mintákat.

while (lastSettedFFTBufferIndex != WINDOW\_SIZE ) {

// Az ablakozás végrehajtása.

fftBuffer[ lastSettedFFTBufferIndex ] =

m\_recordedSamples[lastProcessedSampleIndex]

\* g\_window[lastSettedFFTBufferIndex];

// A feldolgozás indexeinek növelése

lastSettedFFTBufferIndex ++;

lastProcessedSampleIndex ++;

// Ha elérjük a record buffer végét, akkor átugrik a nulladik

// indexre

if (lastProcessedSampleIndex == m\_recordedLength ) {

lastProcessedSampleIndex = 0;

}

// Ha elér a felvett minták végére NULL-al visszatér. A

// metódus következő meghívásánál a statikus változók miatt

// itt folytatódik a futást

if (lastProcessedSampleIndex

== (\*m\_recordedSamplesLastSetted))

{

if (!m\_loopActive) {

lastProcessedSampleIndex = 0;

}

return NULL;

}

}

// FFT buffer megtelt az utoljára beállított indexét 0-ra állítja,

// hogy következő meghívásnál elölről kezdje feltölteni az FFT

// buffert.

lastSettedFFTBufferIndex = 0;

// Az ablak eltolása fél ablak mérettel

if(((int)lastProcessedSampleIndex - WINDOW\_SHIFT) < 0) {

int shift = (int)lastProcessedSampleIndex - WINDOW\_SHIFT;

lastProcessedSampleIndex = m\_recordedLength + shift;

} else {

lastProcessedSampleIndex -= WINDOW\_SHIFT;

}

return fftBuffer;

}

* float\* getSpektrum(float \*fftBuffer)

Előállítja az fft buffer spektrumát és visszatér vele.

float\* VoiceProcessor::getSpektrum(float \*fftBuffer) {

static float spektrum[SPEKTRUM\_SIZE]; // Spektrumot tároló tömb

static FFT fft = FFT(WINDOW\_SIZE); // FFT-t megvalósító

// osztály

// FFT algoritmus lefuttatása

fft.RealFFTf(fftBuffer);

// Spektrum tömb beállítása

for(int i = 1; i < ((WINDOW\_SIZE/2)); i++) {

float real =

fftBuffer[fft.getHFFT()->BitReversed[i] ] / 10000;

float imag =

fftBuffer[fft.getHFFT()->BitReversed[i]+1] / 10000;

spektrum[i-1] = sqrt(real\*real + imag\*imag) ;

}

return spektrum;

}

* Parameter\* getMelSpektrum(float \*spektrum)

Előállítja a mel-spektrumot és eltárolja a paraméterben.

Parameter\* VoiceProcessor::getMelSpektrum(float \*spektrum) {

static float melSpektrum[MEL\_SPEKTRUM\_SIZE];

static Parameter parameter;

// Spektrum mel-szűrése, a mel-spektrum a melSpektrum tömbben

// tárolódik el.

m\_melFilter.melFiltering(spektrum, melSpektrum);

float ampl = 0;

// Kiszámolja az amplitúdót

for(int i = 0; i < MEL\_SPEKTRUM\_SIZE; i++) {

ampl += melSpektrum[i];

}

ampl /= MEL\_SPEKTRUM\_SIZE;

// A paraméter vektor beállítása

parameter.setParameters(melSpektrum, MEL\_SPEKTRUM\_SIZE-1, ampl);

return &parameter;

}

* bool checkInWord(Parameter\* parameter);

Feldolgozza a paramétereket, kiszámolja a zajküszöböt és megállapítja, hogy a paraméter amplitúdója nagyobb-e a zajküszöbnél.

bool VoiceProcessor::checkInWord(Parameter\* parameter) {

// A beszéd küszöb értéke

static float noiseBorder = 1.0f;

if (noiseBorder == 0) {

noiseBorder = 1.0;

}

// A paraméter amplitúdójának összehasonlítása a

// zajküszöbbel, és eltárolás a paraméter vektorban,

// hogy szóhoz tartozik e vagy nem. Amennyiben nagyobb a

// paraméter a zajküszöb elkezd lassan nőni, ezzel beállva a

// statikus háttérzajra.

if (noiseBorder > parameter->amplitude) {

noiseBorder = parameter->amplitude;

parameter->inWord = false;

} else {

if (NOISE\_MULT \* noiseBorder < parameter->amplitude) {

parameter->inWord = true;

}

noiseBorder \*= (NOISE\_CONVERG);

}

return parameter->inWord;

}

## Paraméter vektorok feldolgozása

A paraméterek feldolgozását a voiceLearner osztály valósítja meg, a processParameter metódusával. A metódus a paraméter vektoroknak a beszéd küszöbszint feletti tulajdonságát figyeli, hogy az szóhoz tartozik-e vagy sem. Számon tartja a szóhoz tartozó paraméter vektorok számát, a szó utáni küszöbszint alatti paraméter vektorokat. Ha a szó utáni szünet átlépi a globális konstansban meghatározott értéket, és a szó hossza megfelel, a minimális szó hossznak akkor a visszatérési paraméterben jelzi a főprogramnak, hogy elérte a szó végét. A főprogram fogja a további utasításokat adni, hogy az osztály mit csináljon a paramétervektorokkal. Ez a metódus játszik szerepet a rövid idejű háttér zajok kiszűrésében, mivel a túl rövid szavakat kiszűri a feldolgozási láncból.

### A paraméter feldolgozó forráskódja

int voiceLearner::processParameter(Parameter\* parameter) {

// Ha a paraméter nagyobb, mint a beszéd alsó küszöbe,

// vagyis az inWord tulajdonság true,

// akkor hozzáadódik a paramétereket tároló tömbhöz,

// a beszéd szint alatti paraméterek számát lenullázza

if (parameter->inWord) {

saveParameter(parameter);

m\_noiseCount = 0;

// Ha új szó kezdődik, a szóhoz tartozó

// paraméter számlálót le nullázza

if(!m\_inWord) {

m\_inWord = true;

m\_parameterCount = 0;

}

} else { // Másik eset, amikor nem szóhoz tartozik a paraméter

// Ha a szó véget ér, de a szó hossza nem éri el a

// szó minimum hosszát, akkor nem veszi szónak, és nem

// dolgozza fel. Ezzel a rövid idejű háttérzajokat

// figyelmen kívül hagyja a program.

if ( m\_inWord

&& m\_parameterCount > 0

&& m\_parameterCount < WORD\_MIN\_LENGTH)

{

m\_inWord = false;

// Másik eset mikor a szó hossza nagyobb, mint a globális

// minimum konstans, ekkor a csendes intervallumok is fontos

// részei lehetnek a szónak.

} else if(m\_inWord && m\_parameterCount > WORD\_MIN\_LENGTH){

// elmenti a paramétert

saveParameter(parameter);

// növeli a zaj számlálót

m\_noiseCount++;

// Ha a csendes intervallum meghaladja a

// szó utáni maximum csend hosszát

// a program visszatér a WORD\_END értékkel,

// jelezve, hogy a szó végére ért.

if(m\_noiseCount > NOISE\_AFTER\_WORD) {

m\_inWord = false;

m\_parameterCount -= m\_noiseCount;

return WORD\_END;

}

}

}

// Visszatérési érték jelzi, hogy a metódus feldolgozása

// rendben lezajlott.

return LISTENING;

}

## Minták megtanulása

Miután rendelkezésre állnak a szóhoz tartozó paraméter vektorok, a főprogram aktuális állapotától függően eltárolásra vagy felismerésre kerül a sor. Ahhoz, hogy az összehasonlítást el tudjuk végezni először mintákat, kell megtanítani a programnak, ebben a részben a minták eltárolását ismertetem. Az azonos minták tárolása, rontja a keresés hatékonyságot, ezért mielőtt elmenteném a mintát, lefuttatok egy keresést a már megtanult minták között, ami visszaadja, hogy milyen távol van a legközelebbi tárolt szótól. Ha a távolság nagyon kicsi, felesleges még egyszer eltárolni a szót. A minimális távolságot, megfigyelés és tapasztalati úton határoztam meg, nem érdemes túl kicsire állítani, mert akkor mindent megtanul, de túl nagyra se, mert előfordulhat, hogy két szó nagyon hasonlít egymásra, és ha nem engedjük a programnak, hogy megtanulja a szó elég változatát, nem fogja tudni felismerni később. Ilyen szavak például a ˝négy˝ és a ˝hét˝, amik paraméter vektorai nagyon hasonlóak, ha a program megtudja különbözetni ezt a két szót, és pár minta tanítása után leáll a többi minta elmentésével, valószínűleg jó értéket választottunk. A paraméterek elmentését a voiceLearner osztály saveWord metódusa végzi, mely elvégzi a fent leírt műveleteket, és ha kell, elmenti a paraméter vektorokat az azokat tároló globális listában.

### Szó paraméterek elmentésének forráskódja

SearchResult\* voiceLearner::saveWord(QString word) {

reducedParameterVector();

// Globális paraméter vektor rendezése hossz szerint.

// a keresés rendezett listában gyorsabb

qSort( g\_parameterVectors.begin(),

g\_parameterVectors.end(),

parameterVectorCompare

);

// Az aktuális szó paraméterének összehasonlítása,

// a már eltárolt mintákkal. A SearchResult

// eltárolja a legközelebbi szót, és a hozzá tartozó

// távolságot.

SearchResult\* sr =

m\_recogniser.recognise(m\_parameters, m\_parameterCount);

// Ha a távolság kisebb mint a LEARN\_MAX\_DIST

// akkor a program nem tárolja el a paraméter vektort.

if (sr->distance < LEARN\_MAX\_DIST) {

cout << "Nem szükséges menteni" << endl;

return sr;

}

// Paraméter vektor eltárolása

ParameterVektor\* npv = new ParameterVektor();

npv->setParameters(m\_parameters, m\_parameterCount);

WordParameters\* pw = getWordParameter(word);

npv->m\_wordParameter = pw;

pw->parameterVectors << npv;

g\_parameterVectors << npv;

m\_parameterCount = 0;

m\_noiseCount = 0;

return sr;

}

* reducedParameterVector()

A reducedParameterVector függvény eltávolítja a tárolt paraméterekből az információt nem hordozó elemeket. Ezt úgy éri el, hogy átlagolja a paraméter-vektorok elemeit és az átlagnál 10%-ánál kisebb elemeket nullára állítja, újra számolja a paraméterek hosszát, ezzel csökkentve a számítási igényt.

void voiceLearner::reducedParameterVector() {

float sumParameters = 0;

float avg = 0;

// Paraméterek összeadása

for (int i = 0; i < m\_parameterCount; i++) {

for (int j = 0; j < MEL\_SPEKTRUM\_SIZE; j++) {

sumParameters += m\_parameters[i].parameters[j];

}

}

// átlag kiszámítása

avg = sumParameters / (m\_parameterCount\*MEL\_SPEKTRUM\_SIZE);

// Az átlag alatti elemek 0-ra állítása

// és az új parméter hosszak kiszámítása.

for (int i = 0; i < m\_parameterCount; i++) {

float ampl = 0;

for (int j = 0; j < MEL\_SPEKTRUM\_SIZE; j++) {

if ( m\_parameters[i].parameters[j] < avg\*0.10 ) {

m\_parameters[i].parameters[j] = 0.0f;

} else {

m\_parameters[i].length = j;

ampl += m\_parameters[i].parameters[j];

}

}

m\_parameters[i].amplitude = ampl / MEL\_SPEKTRUM\_SIZE;

}

}

## Paraméter vektorok felismerése

Mivel a voiceLearner osztályban meg van valósítva a szó paramétereket feldolgozó algoritmus, és létrehozza a szóhoz tartozó paraméter vektorokat, a keresést is ebből az osztályból hívom meg. A keresési metódusokat kiemeltem a wordRecogniser osztályban, ami a szó paraméter vektorait hasonlítja össze az eltárolt szavak paraméter vektoraival. Az algoritmus megvalósításánál törekedtem arra, hogy ne fusson le felesleges összehasonlítás, ezért minden módszer megvalósítottam, amit az 5.4-es fejezetben ismertettem.

* A szavak hosszának abszolút különbsége túl távol van.
* Az átlag amplitúdók távolsága átlép egy bizonyos küszöb számot.
* A szavak távolsága összehasonlítás alatt túl távol került.
* A szavak hosszuk szerint sorba rendezése
* A szavak hossz szerinti keresése (Logaritmikus kereséssel)

A szó szintű összehasonlító algoritmus belépési pontját a wordRecogniser recognise metódusa adja.

### A recognise forráskódja

SearchResult\* wordRecogniser::recognise(Parameter\* parameters,

int length)

{

// A szóhoz tartozó paraméterekből, létrehoz egy

// paraméter vektort.

ParameterVektor pv;

pv.parameters = parameters;

pv.parameterLength = length;

// Notmalizálja a paraméter vektort.

pv.normalize();

g\_learnParameters->addNormalisedParameters(

pv.parameters,

pv.parameterLength

);

// Logaritmikus kereséssel megkeressük a hossz szerint

// rendezet listában a paraméter vektor hosszánal

// legmegfelelőbb tárolt paraméter vektort.

int begin = 0;

int end = g\_words.size()-1;

int minBorder = length - DTW\_MAX\_LENGTH\_DISTANCE;

int maxBorder = length + DTW\_MAX\_LENGTH\_DISTANCE;

while(begin < end) {

int midle = (begin+end) / 2;

if (g\_parameterVectors[midle]->parameterLength < length) {

begin = midle+1;

} else {

end = midle;

}

}

int searchLeftIndex = begin;

int searchRightIndex = begin+1;

// Inicializálja egy megfelelően nagy értékre

// a távolságot.

m\_searchResult.distance = FLT\_MAX;

m\_searchResult.word = "";

// Inicializálja a minimum amplitúdó távolságot.

float minAmplitudeDist = FLT\_MAX;

bool searchingRun = true;

/\*

\* A megfigyelt paraméter vektor összehasonlítását, a hosszban

\* legközelebbi tárolt paraméter vektorral kezdi, majd hosszban

\* egyre távolabbi paraméter vektorokkal fojtatom, a kezdeti

\* pontól lépkedve jobbra és balra.

\*

\*/

while (searchingRun) {

bool leftIsRuning = true;

// Elvégzi a baloldali paraméter vektor összehasonlítását

// és beállítja a legkisebb távolságot ha szükséges.

if (g\_parameterVectors.size() > 0

&& searchLeftIndex >= 0

&& g\_parameterVectors[searchLeftIndex]->parameterLength

>= minBorder )

{

float ampDist = minAmplitudeDist;

float dist = DTWDistance(&pv,

g\_parameterVectors[searchLeftIndex],

m\_searchResult.distance,

minAmplitudeDist,

&ampDist

);

if (dist < m\_searchResult.distance) {

minAmplitudeDist = ampDist;

m\_searchResult.distance = dist;

m\_searchResult.word =

g\_parameterVectors[searchLeftIndex]

->m\_wordParameter->word;

m\_searchResult.wordPointer =

g\_parameterVectors[searchLeftIndex]

->m\_wordParameter;

}

searchLeftIndex--;

} else {

leftIsRuning = false;

}

bool rightIsRuning = true;

// Elvégzi a jobboldali paraméter vektor összehasonlítását

// és beállítja a legkisebb távolságot ha szükséges.

if (g\_parameterVectors.size() > 0

&& searchRightIndex < g\_parameterVectors.size()

&& g\_parameterVectors[searchRightIndex]

->parameterLength < maxBorder+1 )

{

float ampDist = minAmplitudeDist;

float dist = DTWDistance(&pv,

g\_parameterVectors[searchRightIndex],

m\_searchResult.distance,

minAmplitudeDist,

&ampDist

);

if (dist < m\_searchResult.distance) {

minAmplitudeDist = ampDist;

m\_searchResult.distance = dist;

m\_searchResult.word =

g\_parameterVectors[searchRightIndex]

->m\_wordParameter->word;

m\_searchResult.wordPointer =

g\_parameterVectors[searchRightIndex]

->m\_wordParameter;

}

searchRightIndex++;

} else {

rightIsRuning = false;

}

if (!rightIsRuning && !leftIsRuning) {

searchingRun = false;

}

}

return &m\_searchResult;

}

* DTWDistance függvény forráskódja

A DTWDistance függvény végzi két szó paraméter vektorainak összehasonlítását.

Paraméterei:

* ParameterVektor\* a, Az első szó paraméter vektorai
* ParameterVektor\* b, A második szó paraméter vektorai
* float max\_distance, A legközelebbi szó távolsága, ha keresés alatt ennél nagyobb értékre ér az algoritmus leáll, és visszatér a nagyobb értékkel.
* float max\_amplitude\_dist A legközelebbi amplitúdó értéke, ha az összehasonlítás ennél nagyobb értéket talál, megszakítja a keresést.
* float\* ret\_amplitude\_dist A kiszámított amplitúdó visszatérési értéke

float wordRecogniser::DTWDistance(ParameterVektor\* a,

ParameterVektor\* b,

float max\_distance,

float max\_amplitude\_dist,

float\* ret\_amplitude\_dist

)

{

int a\_length = a->parameterLength;

int b\_length = b->parameterLength;

// Az első és második paraméter vektor hossz

// különbségének kiszámítása

float length\_distance = qAbs(a\_length - b\_length );

// Ha hossz eltérése nagyobb, mint a globális konstans

// DTW\_MAX\_LENGTH\_DISTANCE meghatározott érték

// a függvény visszatér egy nagy értékkel.

if ( length\_distance > DTW\_MAX\_LENGTH\_DISTANCE) {

return 1000;

}

// Az amplitúdók távolságának kiszámítása

(\*ret\_amplitude\_dist) = getAmplitudeDistance(a,b, max\_amplitude\_dist);

// Ha az amplitúdók eltérése nagyobb, mint a legkisebb eltérés

// akkor a paraméter vektorok távolsága is nagyobb lesz, ezért

// az összehasonlítás megszakad, a függvény visszatér egy

// nagy értékkel.

if ((\*ret\_amplitude\_dist) > max\_amplitude\_dist) {

return 1001;

}

// Radius a keresési tér korlátozásának mértékét

// tárolja le.

float radius = DTW\_WORD\_RADIUS;

if (radius < 0) {

radius = b\_length;

}

// A két paraméter vektor hosszából kiszámolja

// a keresési tér átlójának meredekségét.

float m = ((float)b\_length / a\_length);

// eltárolja az előző összehasonlítási térszelet

// kezdő és végpontját.

int pre\_bottom = 0;

int pre\_top = radius;

// Végigmegy az a szó paraméter vektorain,

// és összehasonlítja b vektor megfelelő paramétereivel

for (int i = 1; i < a\_length; i++) {

int bottom = (i\*m) - radius;

int top = (i\*m) + radius;

if (bottom < 1)

bottom = 1;

if (top > b\_length)

top = b\_length;

int j = bottom;

float minDistance = DEFAULT\_COST;

for (; j < top; j++) {

// A két szó paraméter vektorának összehasonlítása

m\_distanceArray[i][j] =

m\_parameterDTW.runDTW(a->parameters[i].parameters,

a->parameters[i].length,

b->parameters[j].parameters,

b->parameters[j].length,

DTW\_PARAMETER\_RADIUS

); //cost

float insert = DEFAULT\_COST;

float delet = DEFAULT\_COST;

float match = DEFAULT\_COST;

// Beillesztés költsége

if (i > 0 && j < pre\_top ){

insert = m\_distanceArray[i-1][j];

}

// Törlés költsége

if ( (j-1) >= bottom ){

delet = m\_distanceArray[i][j-1];

}

// Az egyezés költsége

if (i==0 && j==0){

match = 0;

} else if ( i>0 && j > 0

&& (j-1) >= pre\_bottom

&& (j-1) < pre\_top

)

{

match = m\_distanceArray[i-1][j-1];

}

// A költségek közül a legkisebb kiválasztása

float min = insert;

if (min > delet) {

min = delet;

}

if (min > match) {

min = match;

}

// A legkisebb költség és az adott pontban lévő

// paraméterek távolságának összegzése.

m\_distanceArray[i][j] += min;

// Az aktuális összehasonlítási térszelet minimumának

// meghatározása

if(m\_distanceArray[i][j] < minDistance) {

minDistance = m\_distanceArray[i][j];

}

}

// A keresési tér szelet kezdő és végpontjának

// eltárolása

pre\_bottom = bottom;

pre\_top = top;

// Ha a szelet legkisebb távolsága nagyobb mint a

// maximálisan megengedett távolság, akkor az összehasonlítás

// megszakad, és az aktuális minimum távolsággal tér vissza.

if(minDistance > max\_distance) {

return minDistance+1.0;

}

}

// Ha az összehasonlítás lefutott a két vektor távolságát

// a távolság tömb első vektor hossza és a második vektor

// hosszának indexéből lehet kiolvasni.

return m\_distanceArray[a\_length-1][b\_length-1];

}

## Dinamikus idővetemítés

A dinamikus idővetemítést a DTW osztályban valósítottam meg. A DTW osztály lefoglal egy két dimenziós tömböt, amelyben eltárolja az idővetemítés alatt kiszámított távolságokat. A tömböt a createDistanceArray tagfüggvénnyel hozhatjuk létre, melynek meg kell adni a tömb méreteit. Fontos, hogy a tömb méret paramétereinek meg kell egyeznie a később összehasonlított paramétervektorok hosszával. Futás során nem változik a tömb mérete, ezzel nem terheli a processzort felesleges memória lefoglalásokkal, és felszabadításokkal.

Az idővetemítő algoritmust a DTW osztály runDTW metódusa valósítja meg. Öt paramétert vár:

* Első paraméter tömböt és a hosszát.
* Második paraméter tömböt és a hosszát.
* A keresési tér rádiuszát.

### DTW forráskódja

Az algoritmus megvalósítja a dinamikus idő-vetemítést. Lekorlátozza a keresési teret, és megkeresi a legkisebb költségű út költségét, vagyis a két paraméter vektor közti távolságot.

Paraméterei:

* float\* a Az első paraméter vektor
* int a\_length Az első paraméter vektor hossza
* float\* b A második paraméter vektor
* int b\_length A második paraméter vektor hossza
* int radius A keresési tér rádiusza

float DTW::runDTW(float\* a,

int a\_length,

float\* b,

int b\_length,

int radius)

{

// Ha a radius kisebb, mint nulla, a keresés

// az egész keresési teret bejárja.

if (radius < 0) {

radius = b\_length;

}

// Ha az első vektor hossza nulla,

// a távolság a második eleminek összegével

// lesz egyenlő.

if(a\_length == 0 ) {

float sum = 0;

for(int i = 0; i < b\_length; i++) {

sum += b[i];

}

return sum;

}

// Ha a második vektor hossza nulla,

// a távolság az első vektor elemeinek összegével

// lesz egyenlő

if(b\_length == 0 ) {

float sum = 0;

for(int i = 0; i < a\_length; i++) {

sum += a[i];

}

return sum;

}

// A keresési tér átlójának, meredekségének

// kiszámítása a vektorok hosszából.

float m = ((float)b\_length / a\_length);

// ellenőrzés, ha véletlen a paraméter vektorok

// hossza nagyobb lenne, mint a távolság tömb mérete

// ne fagyjon le a program.

if(a\_length >= m\_width || b\_length >= m\_height)

return 10000;

// Az előző keresési tér szelet, kezdő és

// végpontjának eltárolása.

int pre\_bottom = 0;

int pre\_top = radius;

// Az első vektor hosszán végigmegy a keresés,

// kiszámolja a keresési tér adott szelethez

// tartozó kezdő és végpontját (bottom, top)

// és elvégzi az összehasonlítást és útkeresést.

for (int i = 0; i < a\_length; i++) {

int bottom = (i\*m) - radius;

int top = (i\*m) + radius;

if (bottom < 0)

bottom = 0;

if (top >= b\_length)

top = b\_length;

int j = bottom;

for (; j < top; j++) {

// Adott pontban lévő költség kiszámítása, a két

// elem különbségének abszolút értékével

m\_distanceArray[i][j] = qAbs(a[i] - b[j]); //cost

float insert = DEFAULT\_COST;

float delet = DEFAULT\_COST;

float match = DEFAULT\_COST;

// A beillesztés költsége, az a vektor előző

// elemének ismétlése

if (i > 0 && j < pre\_top ){

insert = m\_distanceArray[i-1][j];

}

// a törlés költsége, a b vektor előző elemének

// ismétlése.

if ( (j-1) >= bottom ){

delet = m\_distanceArray[i][j-1];

}

// Az egyezés költsége, ha mindkét vektoron előre lépünk

if (i==0 && j==0){

match = 0;

} else if (i>0 && j > 0

&& (j-1) >= pre\_bottom

&& (j-1) < pre\_top)

{

match = m\_distanceArray[i-1][j-1];

}

// A legkisebb költségű út kiválasztása.

// A legkisebb költség megkeresésével.

float min = insert;

if (min > delet) {

min = delet;

}

if (min > match) {

min = match;

}

// Összegzi a legkisebb út költségét és

// a adott pontban lévő távolságot.

m\_distanceArray[i][j] += min;

}

pre\_bottom = bottom;

pre\_top = top;

}

if(m==0) {

m = 1;

}

// A két vektor közti minimális út költségét a

// távolság tömb a\_length-1 és b\_length-1-edik indexen

// tárolja a futás végén az algoritmus.

return m\_distanceArray[a\_length-1][b\_length-1];

}

## Mel-szűrő

A mel-szűrőt a melFilter osztályban implementáltam le. Mivel a programban globálisan vannak deklarálva a mel-szűrő paraméterei:

* A szűrő mérete
* A szűrő tartomány alsó frekvenciája
* A szűrő tartomány felső frekvenciája

Az osztály konstruktora létrehozza a szűrésnél használt kereteket és azok paramétereit. A szűrő méretét 30-nak választottam, ami elég nagy felbontást ad még a paramétereknek, és megfelelő felismerési hatékonyságot nyújt. Az emberi beszéd alsó frekvenciája 300Hz, a felső frekvenciája 3300Hz. Ezeknek az értékeknek megfelelően állítottam be a mel-szűrő frekvenciatartományait. A szűrő kereteket egy struktúrában (filter) tárolja az osztály, mely három értéket tárol:

* A szűrő kezdő indexét
* A szűrő hosszát
* A szűrő súly vektorát

A frekvencia növekedésével egyre szélesedik a szűrők hossza, egyre több frekvenciát fognak át. Szűrés közben a program végig megy a tárolt szűrő kereteken, megszorozza a spektrum megfelelő elemeit a szűrő keret megfelelő súlyával, és összeadja a kapott értékeket minden keretre. Az egyes keretekkel való szűrés állítja elő a mel-spektrumot.

### MelFilter forráskódja

* MelFilter konstruktora

Kiszámítja a szűrő kereteket és eltárolja egy Filter struktúrát tároló tömbben.

melFilter::melFilter()

{

// A mel-skála növekményét kiszámítja a melDelta

// változóban, a melScale függvény visszaadja a

// paraméterben kapott frekvencia mel-skálabeli értékét.

float melDelta =

(melScale(MEL\_MAX\_FREQUENCY) – melScale(MEL\_MIN\_FREQUENCY))

/ (MEL\_SPEKTRUM\_SIZE + 1);

// Ami a mel-skálán lineárisan nő, az a frekvencia

// skálán logaritmikus. A m\_filterCentersInHZ tömbben

// eltárolja a keretek frekvencia tartománybeli

// közép pontját.

for (int i = 0; i < MEL\_SPEKTRUM\_SIZE; i++) {

m\_filterCentersInHZ[i] = melScaleInverse(melDelta \* i);

}

// melFilter a keret súlyait ideiglenesen tároló tömb.

float melFilter[1024];

int begin = 0;

int end = 0;

int mfcount = 0; // mel-filter count

// Minden mel-spektrum elméhez létrehoz egy

// keretet, mellyel a spektrum keretezését,

// és súlyozott összegének, előállítását

// végezzük.

for(int i = 0; i < MEL\_SPEKTRUM\_SIZE; i++) {

begin = -1;

end = -1;

int c = 0;

// Végig megy a spektrum frekvenciáin,

// és meghatározza, az egyes frekvenciákhoz

// tartozó súlyokat.

for(int k = 0; k < SPEKTRUM\_SIZE-1; k++) {

// Adott frekvenciához tartozó súly kiszámítása.

float filterWeight =

m\_filter( m\_filterCentersInHZ,

i,

k \* (1.0f /

((float)WINDOW\_SIZE/(SAMPLE\_RATE/2))));

// A keret kezdő és végpontjának meghatározása

if (filterWeight > 0) {

if(begin < 0) {

begin = k;

melFilter[c] = filterWeight;

} else {

melFilter[c] = filterWeight;

}

end = k;

c++;

}

}

// a keret hossza

int length = end-begin + 1;

// a keret súlyainak elmentése a filter struktúrákat

// tartalmazó tömbbe.

if(begin > 0) {

m\_mellFilters[mfcount].filterWeights = new float[length];

m\_mellFilters[mfcount].filterLength = length;

m\_mellFilters[mfcount].startIndex = begin;

memcpy(m\_mellFilters[mfcount].filterWeights,

melFilter,

length \* sizeof(float)

);

} else {

m\_mellFilters[mfcount].filterLength = 0;

}

mfcount++;

}

}

Az osztály m\_mellFilters tömbjében eltárolja a szűrő kertek paramétereit, mellyel elvégezhetjük a szűrést. A mel-szűrést végrehajtó metódus neve melFiltering, amely paraméterben várja a spektrumot tartalmazó tömböt, és egy tömböt, amelyben az eredményt írhatja. A metódus visszaadhatna egy tömböt, de ezzel, hogy tömbre mutatót adunk át neki, elkerülhetővé válik számos felesleges tömbmásoló utasítás, ezzel gyorsítva a programot.

A melFiltering forráskódja:

void melFilter::melFiltering(float\* spectrum,

float p\_melSpektrum[MEL\_SPEKTRUM\_SIZE])

{

// Bejár minden mel-szűrőt és a súlyokkal beszorozza és összegzi

// a spektrum megfelelő indexű elemeit, előállítva a mel-

// spektrumot.

for (int i = 0; i < MEL\_SPEKTRUM\_SIZE; i++) {

// Nullára Inicializálja az eredmény tömböt.

p\_melSpektrum[i] = 0;

for ( int k = 0; k < m\_mellFilters[i].filterLength; k++)

{

// Hozzáadja az eredmény tömbhöz a súlyozott spektrum

// összeget.

p\_melSpektrum[i] +=

m\_mellFilters[i].filterWeights[k]

\* spectrum[ m\_mellFilters[i].startIndex + k];

}

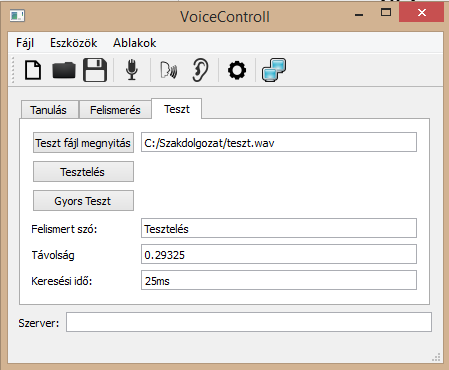
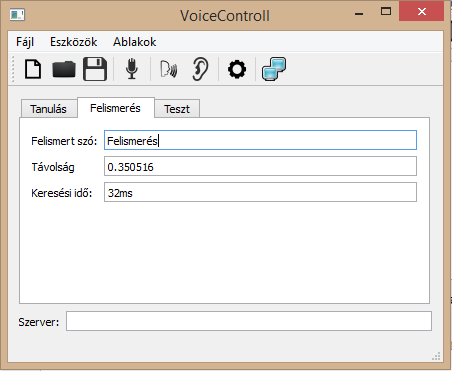
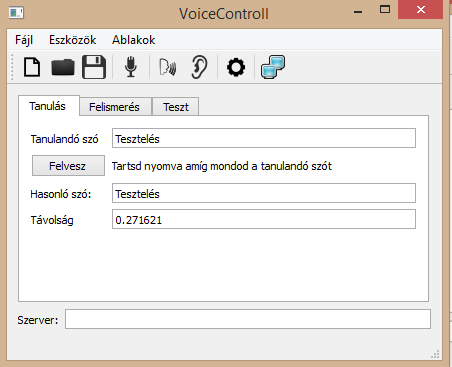
}

}

# A program

A program némi bővítés után számos területen alkalmazhatóvá válhat. Igyekeztem a program felhasználói felületét könnyen kezelhetővé tenni, ezáltal egyszerű felhasználók is könnyen megtudják tanítani a parancsszavakat a programnak, némi gyakorlás után. A program bővíthetőségét szem előtt tartva létrehoztam egy modult, amely tcp/ip protokollon keresztül más programokhoz tud csatlakozni, és kommunikálni velük, átküldeni a parancsszavakhoz tartozó információkat, utasításokat. Így a kiegészítő programokat akár más nyelveken is létrehozhatjuk, például java-ban.

## Felhasználói interfész



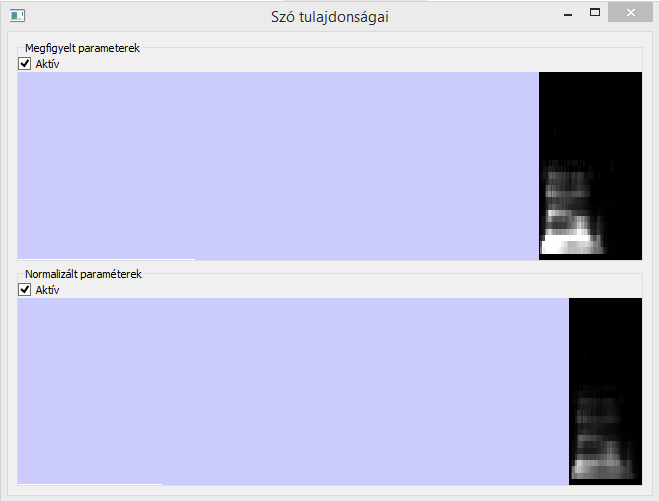
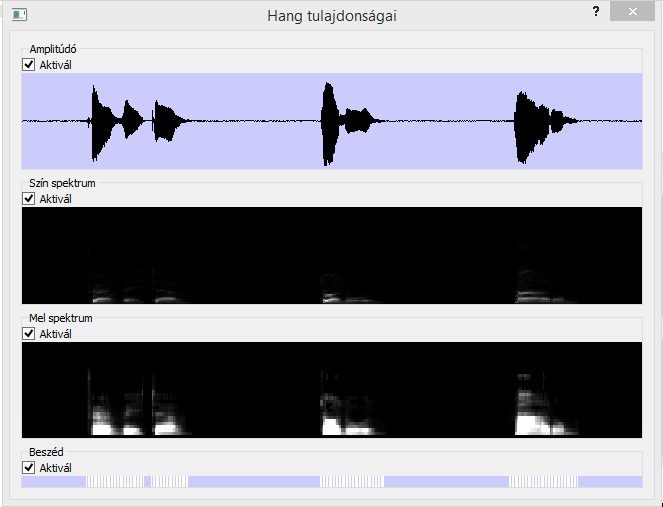
* + 1. A program fő felhasználói felületei.

A 10.1.1. ábrán a főprogram látható. A felső ikonok:

* Új szótár: Kiüríti a megtanult szavakat.
* Szótár megnyitása: Betölthető az előzőleg elmentet szótár
* Szótár mentése: Elmenti a megtanult szavakat.
* Mikrofon be- és kikapcsolása
* A felvett hang paramétereinek megtekintésére szolgáló ablak megnyitása
* A szavak paraméterinek megtekintése.
* A szavak szerkesztése menü.
* Kapcsolódás szerverhez dialóg megnyitása

Alatta három darab fül közül választhatunk. Az első fület választva miután bekapcsoltuk a felvételt, taníthatjuk be az új szavakat. A program addig tanulja a szavakat, amíg nyomva tartjuk a ˝Felvesz˝ gombot. Tanítás közben megfigyelhető, hogy a megtanult szavak közül, melyik szó és milyen távol volt a tanított szótól, így esetleg ha két szó nagyon hasonlít egymásra többször rámondva növelhető a felismerés hatékonyságát. A második felismerés fülre kattintva a program átvált a felismerésre, ebben a módban, ha a mikrofon be van kapcsolva, a program figyeli a kiejtett szavakat, ha felismer egyet, akkor kiírja a szót, a szavak távolságát, majd végrehajtja a szóhoz tartozó utasítást, esetleg elküldi a szervernek a parancsot, ha csatlakozunk egyhez. A harmadik fül a tesztelést szolgálja. Itt betölthetünk egy megfelelő formátumú ( Sample rate: 11025, 1 csatornás), wav fájlt, amit a ˝Tesztelés˝ gombbal lejátszhatunk, hallgathatjuk a fájl tartalmát, és közben fut a szó felismerés, ha a felismerő talál egy ismert kifejezést, akkor kiírja. Második gomb a ˝Gyors teszt˝, amire kattintva a program nem játssza le a fájlt, csak feldolgozza és kiíratja a felismert szavakat pontosvesszővel elválasztva.

A hang feldolgozása során lehetőséget biztosít a program, hogy figyelemmel kövessük a feldolgozás különböző fázisait, ez látható a 10.1.2. ábrán.



10.1.2. ábra: A hang tulajdonságai megjelenítő ablakok

A 10.1.2. ábra bal oldalán a minták feldolgozásának különböző fázisai láthatóak, melyek felülről lefelé:

* A digitalizált minták, a beszéd rezgésének légnyomás változtatása
* A digitalizált minták spektrumai, egymás után fűzve alkotják a színspektrumot.
* A spektrumok mel-szűrésével kapott spektrum.
* A beszéd küszöb átlépésének vizualizálása

A baloldali ábrán a legalsó vékony megjelenítő, fehér részei jelzik ahol a beszéd hangereje, átlépte a beszéd küszöböt, ezt az adatsorozatot dolgozza fel a szavakat egymástól elkülönítő modul.

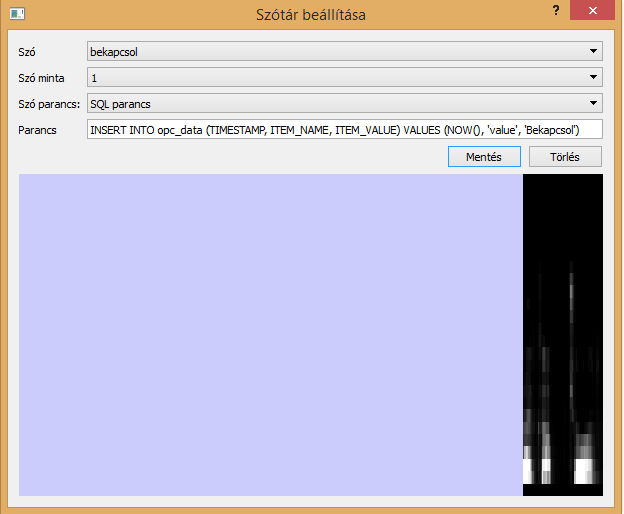
10.1.2. ábra baloldalán két megjelenítő látható:

* A megfigyelt paraméterek
* Normalizált paraméterek

Miközben a felismerő modul feldolgozza a beszédet egymás után fűzi a paraméter vektorokat, ennek eredménye látható a felső megjelenítőn. Ha a megfigyelt és eltárolt paraméter vektorok hosszra megfelelnek egy lehetséges szó hosszának, akkor a program levágja a paraméter vektor végéről a csendet tartalmazó részt, amit a program kivárt, hogy biztos vége legyen a szónak. Ezeket a paraméter vektorokat normalizálja, és tovább küldi a felismerő modulnak, amely visszaadja a megfigyelt paraméterekhez tartozó legközelebbi szót.

10.1.3. ábrán újabb két dialóg ablak látható. A baloldali szótár beállításai lehetőségeit láthatjuk, ezek:

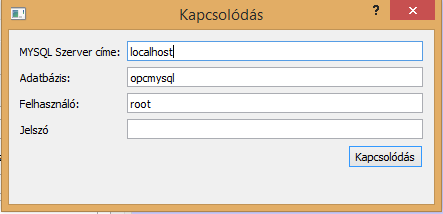
* Az egyes szó minták törlése, ha egy szóból minden mintát kitörlünk, a szó is törlődik.
* A szóhoz tartozó parancs és SQL utasítás beállítása.



10.1.3. ábra: Szótár beállításai és Kapcsolódás a szerverhez dialógjai

Szóhoz tartozó parancs lehet:

* SQL parancs
* nop, ebben az esetben nem tartozik hozzá utasítás.



10.1.4. ábra: Kapcsolódás a MySQL szerverhez

A 10.1.4. ábra bal oldalán a szerverhez kapcsolódás dialógja látszik. Ezen az interfészen keresztül lehet csatlakozni a kiválasztott MySQL szerverhez, melyhez a felismert parancsszavakhoz tartozó SQL utasításokat elküldődi a program.

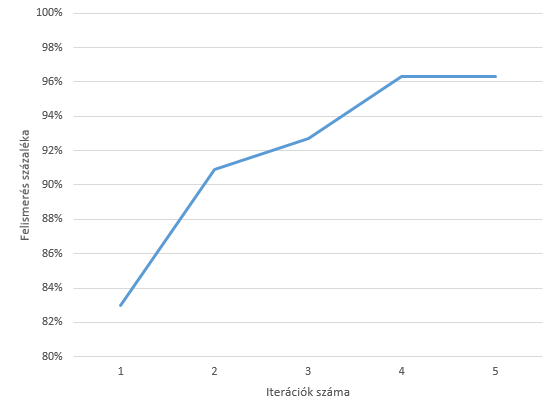
# A program tesztelése

## A parancs felismerő tesztelése

Először a program szó felismerő rendszerét tesztelem. Iterálva a betanítást és a tesztelést, első körben egyszeri rámondással megtanítom 0-tól 10-ig a számokat a programnak, majd ötször elszámolok 0-tól 10-ig feljegyezve a program tévesztéseit, és a keresések időtartamát. Megismételve az eljárást várhatóan javulni fog a felismerések mértéke.

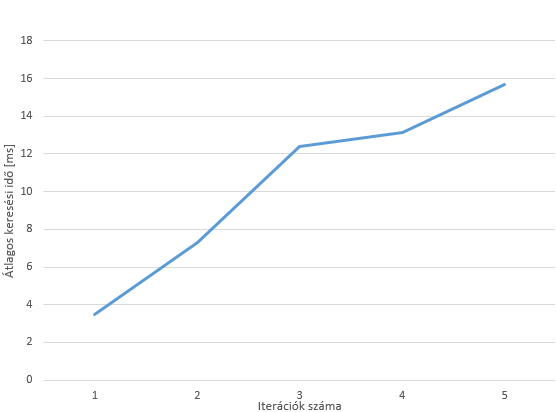
A tesztet ötször ismételtem meg, minden tesztnél újabb mintákat mondtam fel a programnak, majd letesztelve százalékos arányt kaptam, hogy a kiadott parancsszavakat milyen arányban ismerte fel, ez látszik a 11.1. ábrán. A felismerés százaléka a negyedik és ötödik iteráció között nem nőtt, 96%-nál megállt, ez azt jelenti, hogy 55 kiejtett szóból körülbelül 2-3 szót hibásan ismert fel a program.

A program futásának másik kritikus pontja a futási idő, mialatt a program megkeresi a kiejtett szóhoz legközelebb eső tárolt szó paramétereit. Ezért teszt alatt a keresések időtartamát lejegyeztem és iterációnként átlagoltam, ez látható a 11.2. ábrán. A tesztelés alatt a futási idő lineárisan nőtt minden iterációban.



11.1. ábra: A felismerés mértéke minden betanítási iteráció után.

A legutolsó iterációban, amikor minden számról 5 mintája volt a programnak, azaz 55 paraméter vektort tárolt összesen, ekkor a futási idők átlaga 16ms körül volt. A programnak még jó néhány parancsszó megtanítható anélkül, hogy érzékelhetően lassan határozná meg a program a legkisebb távolságot, és elvesztené a program valósidejű működését.



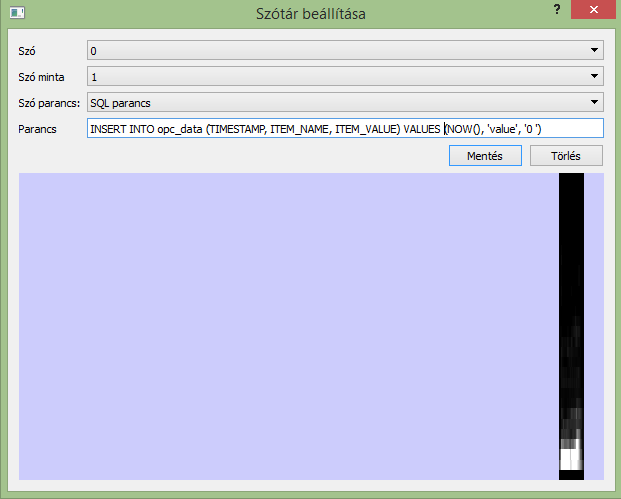
11.2. ábra: Az összehasonlítások átlagos futásideje iterációnként.

## MySQL adatbázisba írás tesztelése

Ebben a részben letesztelem a program és a MySQL adatbázis közti kapcsolatot, erre a localhostra feltelepített wamp szerver MySQL adatbázisát használtam fel. Létrehoztam benne egy opcmysql nevű adatbázist, az adatbázisban pedig egy opc\_data nevű táblát mely tárolja az OPC szervernek szóló utasításokat. A tábla struktúrája:

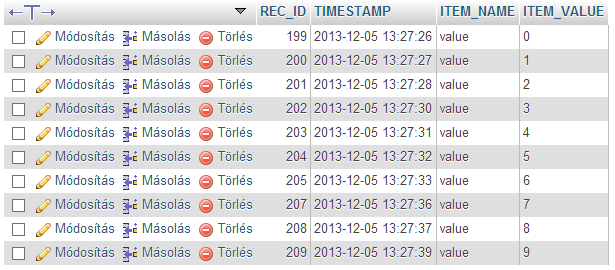
CREATE TABLE `opc\_data` (  
  `REC\_ID` int(11) NOT NULL AUTO\_INCREMENT,  
  `TIMESTAMP` datetime DEFAULT NULL,  
  `ITEM\_NAME` varchar(25) NOT NULL,  
  `ITEM\_VALUE` varchar(255) DEFAULT NULL,  
 PRIMARY KEY (`REC\_ID`)  
)

Eltárolja a rekord id-jét, az időbélyeget, az item nevét, és értékét. A beszédfelismerő programban a szavak beállításánál az SQL utasításokat beállítottam, hogy a táblába szúrjanak be egy új elemet a jelenlegi idővel, item neve ’value’ és az item értéke a szám értékének feleljen meg, ez látható a 11.2.1. ábrán.



11.2.1. ábra: Az SQL utasítás beállítása.

A tesztben elszámolta 0-tól 9-ig. A MySQL adatbázist meg nézve a program beírta a kiejtett értékeket, ez látható a 11.2.2. ábrán.



11.2.2. ábra: Az adatbázis tartalma a teszt után.

# Irodalomjegyzék

[1] http://en.wikipedia.org/wiki/Dynamic\_time\_warping

[2] http://en.wikipedia.org/wiki/Mel\_scale

[3] http://en.wikipedia.org/wiki/Window\_function

[4] http://hu.wikipedia.org/wiki/Fourier-transzform%C3%A1ci%C3%B3

[5] http://molnarimre.atw.hu/Hangtechnika.html

Alulírott Krausz Róbert Zsolt, mérnök informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport Tanszékén készítettem, BSC diploma megszerzése érdekében. Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

Dátum, Aláírás