

UNIVERSITATEA DIN BUCUREŞTI  
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ  
SECȚIA INFORMATICĂ



UNIVERSITATEA  
DIN BUCUREŞTI



Facultatea de Matematică  
și Informatică  
Universitatea din Bucureşti

## PROIECT DE LICENȚĂ

Sistem de prelucrare digitală a sunetelor –  
Recunoașterea vocalelor

**Coordonator științific:**  
Conf. dr. Gramatovici Radu

**Absolvent:**  
Hasna Robert-Mădălin

BUCUREŞTI

2012

## **CUPRINS**

---

Cuprins .....	2
Introducere .....	3
Capitolul I : Reprezentarea informației audio.....	5
Unde audio.....	5
PCM și digitizare audio .....	9
Înregistrare.....	9
Redare.....	11
Tipuri de fișiere audio.....	11
Capitolul II : Prelucrarea audio digitală.....	14
Dithering audio .....	14
Formarea zgomotelor.....	18
Prelucrarea dinamică .....	21
Capitolul III : Analiza de frecvență .....	25
Seria Fourier .....	25
Transformata Fourier discretă (TFD) .....	30
Funcții fereastră .....	32
Cepstrumul și pitch-ul unui semnal audio .....	34
Capitolul IV : Arhitectura aplicației .....	36
Descrierea aplicației.....	36
Prezentarea utilității aplicației .....	39
Rezultatele obținute .....	43
Capitolul V : Implementare .....	46
Pachete și clase .....	46
Paralelism .....	53
Concluzii .....	54
Bibliografie .....	55
Anexă .....	56

## **INTRODUCERE**

---

Vorbirea este o parte a inteligenței și a exprimării umane. Poate fi considerată cea mai eficientă metodă prin care oamenii își comunică informații, fără a avea nevoie de dispozitive ajutătoare. Deși primim o cantitate mai mare de informații, de la stimuli exteriori, prin intermediul vederii, o comunicare mutuală vizuală este aproape total ineficientă, în comparație cu ceea ce este posibil folosind comunicarea prin intermediul vorbirii. Semnalul vocal conține atât informațiile care se doresc a fi transmise, cât și informații despre caracteristicile vocale ale vorbitorului și emoțiile acestuia, informațiile transmise prin vorbire având un rol extrem de important în viața noastră.

Imaginați-vă crearea unei legături de acest fel între om și mașină.

Comunicarea, prin intermediul vorbirii, cu un sistem de calcul deschide o gama largă de posibilități care ar simplifica foarte mult interfața dintre om și calculator. Acest tip de comunicare necesită un sistem de recunoaștere a vorbirii capabil să identifice orice rostire pronunțată de o persoană într-un microfon. Idealul unui astfel de sistem este recunoașterea vorbirii în timp real, cu acuratețe 100% a tuturor cuvintelor pronunțate inteligibil de orice persoană, într-o anumită limbă, independent de mărimea vocabularului, zgomot, caracteristicile și accentul vorbitorului sau condițiile canalului. Acest țel nu a fost încă atins, dar numeroasele cercetări în domeniu și dezvoltarea tehnologiei permit obținerea unor sisteme de recunoaștere din ce în ce mai performante.

Lucrarea este structurată în 5 capitole urmate de anexe și referințe bibliografice.

În **Capitolul I** este prezentat un scurt istoric al undelor sonore, formatul și forma sunetului salvat pe calculator și principalelor standarde.

În **Capitolul II** sunt descrise elemente de prelucrare audio digitală și nevoia de eliminare a zgomotelor.

**Capitolul III** descrie analiza de frecvență a sunetelor și cateva aplicații ale Transformatei Fourier și a împărțirii sunetului în mai multe ferestre folosite pentru a prelucra în timp frecvențele sunetelor.

**Capitolul IV** descrie aplicația, prin prezentarea etapelor prin care se trece pentru a ajunge la faza finală și cateva rezultate obținute.

**Capitolul V** prezintă în amănunt pachetele și clasele folosite în aplicație.

## **CAPITOLUL I : REPREZENTAREA INFORMATIEI AUDIO**

### **Unde audio**

Frecvența sunetului este măsurată în Hertz (Hz), însemnând cicluri pe secunda. Urechea umană percepse sunete cu frecvență cuprinsă între 20 Hz și 20 KHz în cazuri extreme.

Vocea umană în mod generic produce sunete cu frecvență cuprinsă între 80 Hz și 4 KHz. Conform unei vechi teorii a lui Henry Nyquist, emisă în anul 1917 și rămasă valabilă și astăzi, frecvența de eşantionare a semnalului audio trebuie să fie dublă față de frecvența maximă. Ca atare se folosește în mod constant o frecvență de eşantionare de 8 KHz. În anumite situații se poate folosi și o frecvență de eşantionare de 5 KHz deoarece frecvențele fundamentale în domeniul vocii umane se află sub 2,5 KHz. Telefonia în general utilizează o frecvență maximă de 3,1 KHz(300-3400 KHz).

Primele tehnici care au fost folosite la prelucrarea și înregistrarea sunetului au fost analogice: inițial cilindri mecanici și discuri, mai târziu bandă magnetică, piste de sunet pe film optic care depind de etajele de egalizare și amplificare, asigurate cu amplificatoare cu tuburi. Prelucrarea și înregistrarea au suferit un proces de rafinare graduală, dar a ajuns la limite prin fizica procesului, a apărut o nouă tehnologie și anume cea digitală. Semnalul analog este un semnal continuu variabil în timp, iar semnalul digital nu este altceva decât o succesiune de impulsuri (1 sau 0) care reprezintă un număr. Pentru a trece de la semnalul analog la cel digital trebuie că semnalul analog să fie înlocuit cu o succesiune de cifre, materializate sub formă unor impulsuri cu două nivele (0 și 1). În cazul semnalului digital se transmite o succesiune de numere, care reprezintă valoarea semnalului analog în anumite momente. Pentru ca semnalul digital să aproximeze cât mai bine semnalul analogic, reiese de la prima vedere că numărul de puncte în care se face citirea trebuie să fie cât mai mare, iar numerele care exprimă valoarea trebuie să aibă precizie cat mai mare. În tehnica digitală, trecerea de la semnalul analogic continuu la cel digital se numește discretizare, ceea ce presupune valori discrete - în număr finit - ale semnalului, corespunzătoare unor momente discrete.

Principalele probleme ce apar în domeniul procesării semnalelor audio sunt:

- zgomotul suprapus peste semnalul util atât în mediul de propagare acustic cât și pe calea de înregistrare și prelucrare electronică;
- reverberația cauzată de reflexia semnalului sonor pe diferite suprafețe care cauzează ecouri ce sunt înregistrate împreună cu semnalul util;
- reverberația cauzată de instrumentele electronice de înregistrare cum ar fi microfonul, amplificatorul, sintetizatorul.

Datorită complexității semnalului audio efectele perturbatoare enumerate mai sus nu pot fi eliminate printr-o banală filtrare deoarece un filtru ar scoate din semnal și componente spectrale utile.

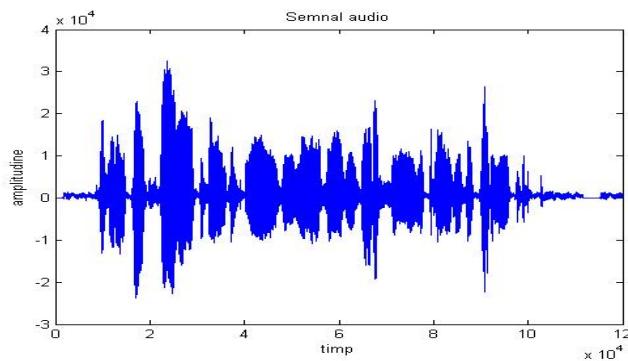


Fig 1.1. Semnal audio

Din punct de vedere tehnic, conversia semnalelor analogice în semnale digitale, adică discretizarea se realizează în două etape. În prima etapă se realizează discretizarea în timp, cunoscută sub numele de eșantionare și apoi se realizează cea de a doua etapă, discretizarea valorii, sau cuantificarea, ceea ce presupune atribuirea unei anumite valori numerice nivelului fiecărui eșantion. Valorile respective sunt în număr finit și sunt exprimate cu un anumit număr de cifre, în cazul acesta scrise în cod binar.

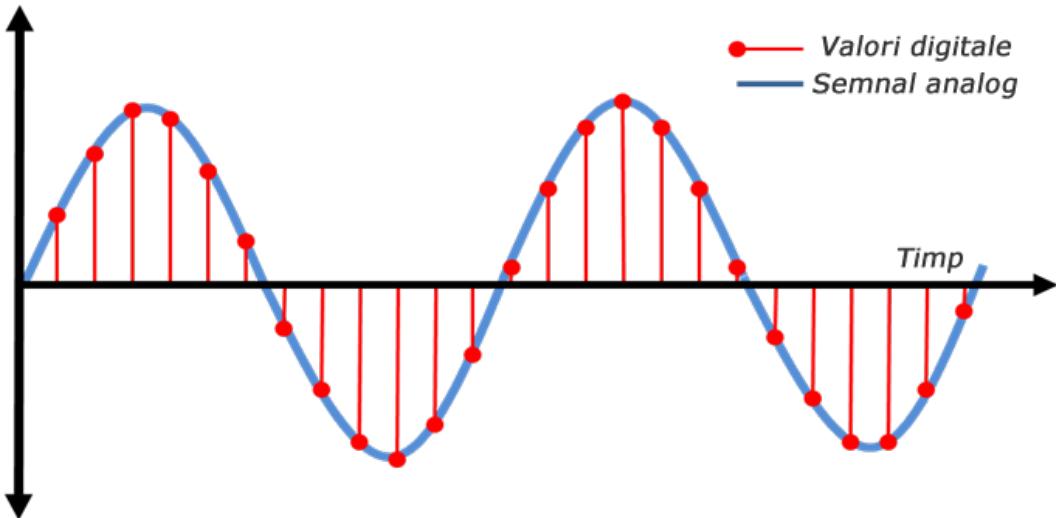


Fig 1.2. Conversie analogic-digital

În figura de mai sus, cu linie continuă este reprezentat semnalul analogic și rezultatul eșantionării. Aceasta a fost făcută în cele 8 momente marcate  $t_0, \dots, t_7$ . Valorile eșantioanelor (nivelele analogice) sunt cuprinse între 0 și 7 unități. Nivelul digital al fiecărui eșantion se exprimă cu ajutorul a 3 săgeți, deci pot fi scrise  $2^3 = 8$  valori. Fiecare eșantion își are asociată astfel un număr digital format din trei cifre care îl exprimă mărimea. Cuantificarea - are ca rezultat succesiunea de impulsuri, unde fiecare trei impulsuri reprezintă valoarea unui eșantion, iar succesiunea de impulsuri este expresia digitală a semnalului analog. O anumită valoare a unui eșantion se acordă valorilor cuprinse în plaja de  $\pm 0,5$  din jurul valoii fixe. De exemplu, se acordă valoarea zero pentru eșantioanele cuprinse între  $\pm 0,5$ ; valoarea 1 pentru eșantioanele cuprinse între 0,5 și 1,5 etc. Toate aceste „erori” de cuantizare constituie „eroarea de discretizare” care reprezintă una dintre problemele conversiei analogic-digitale. În figura 1.1 a fost trasată punctat caracteristica unui semnal analogic care are aceeași expresie digitală ca și caracteristica ideală, trasată cu linie continuă. În realitate, situația este mai complexă, valorile eșantioanelor nu sunt multipli întregi ai unității de eșantionare. Cu cât numărul de cifre este mai mare, cu atât precizia exprimării este mai mare, însă automat și numărul biților care-l exprimă trebuie să fie mai mare.

Într-un sistem analogic pentru înregistrare, distanța de-a lungul mediului de înregistrare este mai departe analogică cu timpul. Dacă semnalul este amplificat, mai multe detalii vor fi puse în evidență până la un anume punct, unde valoarea actuală este nesigură din cauza zgomotului.

Dacă viteza mediului de înregistrare nu este constantă, mărimea analoagă a originalului nu este cea reală.

Un avantaj al înregistrării audio digitale este independența de mediul de înregistrare.

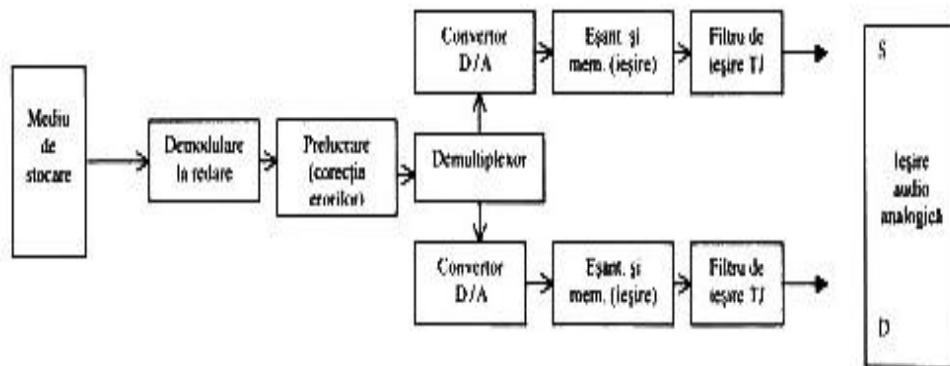
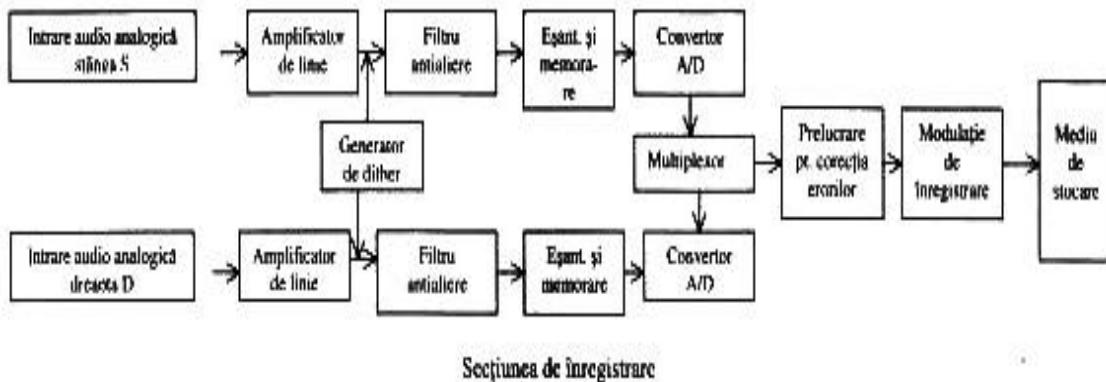
Răspunsul în frecvență, liniaritatea și zgomotul sunt determinate numai de calitatea procesului de conversie. Scara dinamică excepțională și linearitatea sunt obținute combinat cu interferența de zgomotul de diafonie și intermodulație. Recorderele vor suna la fel indiferent de marca benzii, dacă ea are o rată a erorilor acceptabilă.

O înregistrare digitală nu este alceva decât o serie de numere și deci poate fi copiată pe un număr indefinit de generații fară degradare. Viața înregistrării poate fi indefinită, deoarece chiar dacă mediul începe să cadă fizic, valorile eșantioanelor pot fi copiate pe un nou mediu fără pierderea informației.

## PCM și digitizare audio

### Înregistrare

În digitizarea PCM (PULSE-CODE MODULATION) stereo clasică semnalul audio este eșantionat, cuantizat, convertit în formă binară și codat pentru înregistrare sau transmisie.



**Secțiunea de redare**

Fig 1.3. Sistem PCM complet

Inversând procesul apare o replică a originalului.

Secțiunea de înregistrare constă în amplificatoare de intrare, un generator dither, filtre trece jos de intrare, circuite de eșantionare și memorare, convertoare A/D, multiplexor, procesare digitală, circuite de modulație și un mediu de stocare care poate fi bandă magnetică de transport cu cap magnetic rotativ, bandă magnetică de transport cu cap fix, disc optic sau magnetic. În cazul liniilor de întârziere ale reverberatoarelor digitale și a altor procesoare de semnal în timp real, mediul de înregistrare poate fi înlocuit cu un circuit digital de prelucrare, activ. Atunci nu mai este necesară modularea și demodularea și circuitul de protecție contra erorilor. Generatorul de dither este un circuit de zgomot controlat care să fie tipic zgomot alb. Semnalul analogic este trecut prin FTJ cu o frecvență de tăiere sub jumătatea frecvenței de eșantionare.

Un semnal de banda limitată poate fi recuperate din eșantioanele sale dacă acestea sunt luate cu o frecvență mai mare sau egală cu dublul celei mai mari frecvențe din spectrul semnalului initial.

Trecând semnalul eșantionat prin FTJ relational obținem:

$$f_{\Omega_0}(t) = h_{\Omega_0}(t) \otimes T \sum f_{\Omega_0}(nT) \delta(t - nT) = T \sum f_{\Omega_0}(nT) h_{\Omega_0}(t - nT)$$

$$\text{Dar } h_{\Omega_0}(t) = \frac{\sin \Omega_0 t}{\pi t} \text{ rezulta } f_{\Omega_0}(t) = T \sum f_{\Omega_0}(nT) \frac{\sin \Omega_0(t - nT)}{\pi(t - nT)}$$

relație adevarată dacă  $T \leq \frac{\pi}{\Omega_0}$  rezultând  $T \leq \frac{T_0}{2}$  sau  $\Omega \geq 2\Omega_0$  (1)

La recordările profesionale cu o frecvență de eșantionare de 48 kHz, frecvența de tăiere a filtrului este în jur de 22 kHz pentru a permite o maximă atenuare la jumătatea frecvenței de eșantionare. În procesul de eșantionare și memorare condensatorul și comutatorul sunt importante.

Convertorul A/D este cea mai critică și cea mai scumpă componentă în sistemul de digitizare. Totuși anumite operații trebuie realizate. Prima dată ieșirea convertorului A/D este cu date paralel și cele mai multe dispozitive de stocare permit numai date seriale. Astfel datele sunt

multiplexate, adică datele paralel sunt trecute serial. În al doilea rând, fluxul de date trebuie structurat pentru a se identifica cuvintele de date originale din fluxul de biți care rezultă. În înregistrarea analogică eroarea produsă în mediul de înregistrare este un eveniment de neschimbăt. În digital putem asigura detecția și corecția erorilor. Astfel fluxul de date este prevazut cu biți de paritate și controale redundante extradate, create din datele originale pentru a ajuta la detecția erorilor. În final datele sunt modulate și formatare înainte de a fi înregistrate pe bandă.

## **Redare**

Partea de hard la ieșirea sistemului de digitizare este compusă din circuitele de demodulare și prelucrare, un demultiplexor, convertoare digital-analogice, circuite de apertură, filtre trece jos de ieșire și amplificatorul de ieșire. La demultiplexare datele obținute din schema de modulație sunt puse din nou în forma paralel. Sunt folosite detecția erorilor și masurile de protecție în vederea corecției plasate în semnalul digital. Erorile datorită părții mecanice sunt eliminate. Datele vin la ieșire cu o viteză constantă dată de un ceas cu cuarț. Convertorul D/A este mai simplu decât cel A/D. Filtrul „trece jos” la ieșire este identic cu cel de la intrare. Poate fi folosit un filtru analogic sau un filtru digital cu tehnica de supraesantionare în care frecvența de esantionare este multiplicată și se obține un filtru mult mai elegant. Partea finală a sistemului este multiplicată și se obține un filtru mult mai elegant. Partea finală a sistemului este un amplificator analogic proiectat cu grijă.

## **Tipuri de fișiere audio**

In lucrul pe calculator cu semnale audio, după discretizare, valorile esantionate se codifică folosind diferiti algoritmi si se stocheaza sub forma unor fisiere audio. In continuare voi prezenta cateva dintre caracteristicile celor mai folosite tipuri de codificare audio.

Formatele de fișiere audio diferă printr-o serie de caracteristici:

- codificarea eșantioanelor:
  - cuantizare liniară;
  - cuantizare logaritmică;

- valori de eșantioane cu semn sau fără semn;
- formatul datelor:
  - existența unui header;
  - memorarea octetilor în ordinea little-endian sau big-endian;
  - întreținerea canalelor;
- tipul de informații memorate în fișiere:
  - constrângeri legate de rata de eșantionare și/sau adâncimea de biți;
  - constrângeri legate de numărul de canale;
  - informații suplimentare, ca „timestamps”, „loop points”;
- fișierul poate fi comprimat:
  - utilizând o procedură cu pierderi;
  - utilizând o procedură fără pierderi;
- format „open” sau proprietar.

Principalele formate audio sunt prezentate în tabelul urmator:

<b>Extensia de fisier</b>	<b>Tip de fisier sau codec</b>	<b>Caracteristici</b>
.aac	Advanced Audio Coding	Comprimare fără pierderi ca imbunătățire MP3, utilizat pentru iPods, cell phones și PlayStation portable.
.aif	Audio Interchange File Format, format wave standard Apple	Format PCM necomprimat, suportă mono/stereo, 8/16 biți, varietate de rate de eșantionare, ordine big-endian.
.au	NeXT/Sun (Java)	Potrivit să fie fără compresie sau cu compresie CCITT μ-law, A-law, G721; suportă mono/stereo, 8/16 biți, varietate de rate de eșantionare (fără compresie); utilizat pentru distribuție pe Internet și includere în appleturi și aplicații Java.
.flac	Free Lossless Audio Codec, Xiph.Org Foundation	Codec gratuit fără pierderi, compresie 1.5:1 sau 2:1; pentru compresie fără pierderi pentru web.
.mp3	MPEG-1 Layer 3 audio	Rata mare de compresie la o calitate bună, inclusiv pentru web.
.ogg	Ogg Vorbis, Xiph.Org Foundation	Codec gratuit, rata mare de compresie cu bună fidelitate, concurență cu .mp3.
.raw	Fisier Raw	Valori de eșantion fără header, la deschiderea fisierului se introduce rata de eșantionare, rezoluția și număr de canale.
.rm	Real Media	Suportă stream audio (permite ascultare înainte de descarcarea completă a fisierului).

.tta	True Audio	Codec gratuit fara pierderi, compresie 1.5:1 sau 2:1; pentru compresie fara pierderi pentru web, competitie cu .flac.
.wav A-law/μ-law	CCITT standard G.711	Fisiere cu esantioane pe 8 biti create din esantioane pe 16 biti folosind codificarea A-law/μ-law, cu scara dinamica de 13 biti (aprox. 78 dB).
.wav DVI/IMA	International Multimedia Association, versiune ADPCM format .wav cu compresie ADPCM	Utilizeaza o metoda diferita (mai rapida) decat Microsoft ADPCM, alternativa la MPEG cu decodificare rapida si compresie audio de buna calitate.
.wav Microsoft ADPCM	Microsoft	Utilizeaza ADPCM, furnizeaza 4 biti/canal date comprimate cu rata 4:1.
.wav Windows PCM	Microsoft	Standardul Windows pentru audio PCM necomprimat, suporta mono/stereo, varietate de adancime de biti si rate de esantionare, conform specificatiilor RIFF.
.wma/.ASF	Microsoft Windows Media (audio)	Codec Microsoft, permite alegerea calitatii, inclusiv rata de biti constanta (CBR – „constant bit rate”) sau variabila (VBR – „variable bit rate”), compresie cu/fara pierderi, suporta streaming audio.

Fig 1.4 Tabel cu principalele formate audio

## **CAPITOLUL II : PRELUCRAREA AUDIO DIGITALĂ**

### **Dithering audio**

Conversia analog-digitală este în mod obișnuit descompusă în două procese separate: eșantionarea în timp a intrării analogice și cuantizarea amplitudinii valorilor de semnal, pentru ca eșantioanele să poată fi reprezentate prin cuvinte binare de o lungime prescrisă. Operația de eșantionare nu implică pierdere de informație atât timp cât intrarea este de bandă de frecvență limitată, în conformitate cu teorema eșantionării. Natura aproximantă a operației de cuantizare determină în general o degradare a semnalului. O problemă similară apare la recuantizare, în care lungimea cuvintelor de date digitale este redusă după prelucrare pentru a satisface specificațiile pentru stocare sau transmisie. În Fig 2.1 (b) este reprezentată grafic caracteristica de transfer a unui cuantizor de tip treaptă la mijloc (midtread), iar în Fig. 2.1(c) cea a unui cuantizor cu front la mijloc (midriser). Treapta de cuantizare, cunoscută și ca bitul cel mai puțin semnificativ (LSB), este notată cu  $\Delta$ , iar cu  $w$  semnalul la intrarea cuantizorului.

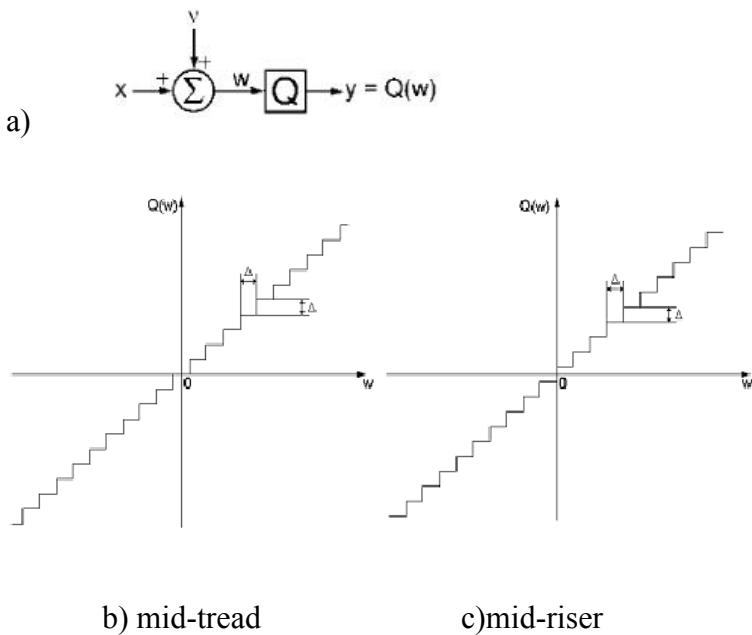


Fig 2.1

Figura 2.1 a) arată cum este adaugat semnalul cu dither  $v$  la semnalul audio  $x$  pentru a fi cuantizat.  $w$  este suma semnalelor  $v$  și  $x$ , care cunatizat va da semnalul de ieșire  $y=Q(w)$ .  $Q$  poate avea o caracteristică pas la mijloc (mid-tread)-figura 2.1(b) sau treapta la mijloc (mid-riser) - figura 2.1(c)

Un sistem analogic ideal, fără zgomot (practic nerealizabil) ar putea fi considerat să fie limită cu lungime infinită a unui sistem digital. Erorile de cuantizare sunt mereu prezente aproape de nivelul de zgomot al oricărui sistem digital. În situația cea mai bună putem spera să facem aceasta pentru a controla natura lor. Numarul de biți de precizie folosiți în cuantizare determină nivelul acestui prag de zgomot - numarul cel mai mare de biți determină nivelul de zgomot relativ cel mai mic pe întreaga scală, de aici cel mai mare raport  $S/Z$ .

Erorile de cuantizare pot fi inofensive, fie daunătoare, depinzând de nivelul și proprietățile semnalului care este cuantizat. Este o eroare dependentă de semnal. Pentru semnale complexe puternice, el poate suna ca un zgomot alb de fond de nivel mic și constant ce însoțește semnalul. Pentru nivele mici și semnale simple, el poate să se manifeste ca o distorsiune armonică semnificativă și ca o distorsiune de intermodulație, însoțită de o modulație severă a zgomotului de fond. Acest lucru este în mod clar nedorit. Idealul ar fi să avem apariția erorii de cuantizare ca un zgomot alb de nivel mic, al căruia nivel este independent de semnal. Acest lucru poate fi realizat prin adunarea unui semnal dither de zgomot dorit în timpul procesului de conversie  $A/D$ . Teoria din spatele funcționării unui astfel de dither fără scadere este complet dezvoltată.

Figura 2.1 arată cum un semnal de dither  $v$  ar trebui să fie adunat la un semnal audio și pentru a fi cuantizat înainte de a fi trimis la cuantizorul  $Q$ , semnalul suma este cuantizat și rezultă ieșirea cuantizată  $v$ . Alegând adecvat proprietățile semnalului de dither se poate controla natura erorilor de cuantizare rezultante. Figura arată de asemenea cele două funcții în scara de cuantizare uniformă - cuantizor cu pas la mijloc (sau rotunjire) și cuantizor cu treaptă la mijloc. Primul, cel cu pas la mijloc este cuantizorul folosit în mod uzual în sistemul multibit, în timp ce ultimul cu treaptă la mijloc, poate fi folosit în sistemele cu biți puțini și în special în sistemele cu 1 bit (care au numai 2 nivele). În astfel de sisteme nivelele disponibile pot fi aranjate simetric

deasupra și sub zero pentru a adapta în mod optim natura bipolară a semnalului audio și astfel să se maximizeze domeniul dinamic.

Simbolul  $\Delta$  semnifică marimea treptei de cuantizare, bitul cel mai puțin semnificativ (*LSB*). Fie  $b$  numarul de biți binari folosiți de cuantizor. El are astfel  $2^b$  nivele de cuantizare (*LSB-uri*). Fiecare creștere cu 1 în  $b$  are ca rezultat o îmbunătățire a raportului *S/Z* cu 6,02 dB. Pentru un cuantizor cu pas la mijloc fără dither cu  $b$  biti și *LSB*,  $\Delta$ , puterea erorii de cuantizare clasice este presupusă în mod uzual  $\Delta^2/12$ . Raportul *S/Z* pentru o formă de undă sinus cu scală întreagă este dat de relația:

$$S/Z = (6.02b + 1.76)[dB] \quad (2)$$

Fenomenul adițional de 1,76 dB justifică faptul că raportul *S/Z* este referit la o undă sinus cu scală întreagă a amplitudinii de vârf  $2^{b-1}$  *LSB-uri* ( $2^b$  *LSB-uri* vârf la vârf). Această formulă este suficient de precisă pentru un  $b$  de cel puțin 8, dar este prea optimistă pentru valori mai mici ale lui  $b$  datorită faptului că pentru un cuantizor, cu treaptă la mijloc, există coduri disponibile cu ieșire mai puțin pozitivă decât negativă. În astfel de cazuri s-ar putea folosi în loc o caracteristică de cuantizor cu treaptă la mijloc.

Pe această bază, un sistem cu 20 biți, de exemplu ar putea avea un raport *S/Z* de 122,2 dB. Problema este că eroarea de cuantizare (zgomot plus distorsiune) nu este constantă pentru un sistem fără dither și astfel termenul *S/Z* nu este în mod real cu semnificație deplină.

Presupunând că acestea reprezintă cuantizoare infinite funcțiile de transfer corespunzătoare pot fi exprimate astfel:

$$Q(w) = \Delta \left\lfloor \frac{w}{\Delta} + \frac{1}{2} \right\rfloor \quad (3)$$

pentru un cuantizor cu treaptă la mijloc sau:

$$Q(w) = \Delta \left\lfloor \frac{w}{\Delta} \right\rfloor + \frac{\Delta}{2} \quad (4)$$

pentru un cuantizor cu front la mijloc unde operatorul  $\lfloor \cdot \rfloor$  dă întregul cel mai mic sau egal cu argumentul său. Deși se va lucra cu cuantizor cu treaptă la mijloc rezultatele obținute sunt valabile pentru amândouă tipurile.

Cuantizarea sau recuantizarea introduce un semnal eroare  $q$  care este diferența între ieșirea cuantizorului  $Q(w)$  și intrarea sa  $w$ :

$$q(w) = Q(w) - w \quad (5)$$

Această eroare de cuantizare este arătată în Fig. 3.2 ca funcție de  $w$ :

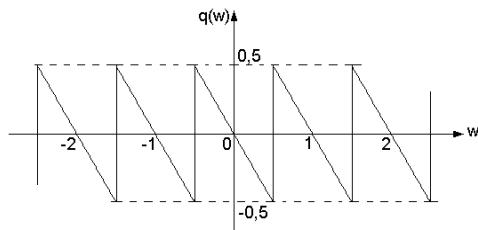


Fig 2.2.

Sistemele de cuantizare sunt de 3 tipuri:

- fără dither (Fig.3.3a);
- cu dither cu scădere (Fig.3.3b);
- cu dither fără scădere (Fig. 3c).

Dacă intrarea sistemului este  $x$ , ieșirea este  $y$ , atunci eroarea totală  $\varepsilon$  a sistemului este:

$$\varepsilon = y - x \quad (6)$$

Eroarea totală  $\varepsilon$  a sistemului este diferită de eroarea de cuantizare  $q$  definită de ecuația (5). În sistemele de cuantizare cu dither la semnalul de intrare al cuantizorului apare adunat un semnal aleator notat  $v$ , care este presupus a fi staționar și statistic independent de  $x$ .

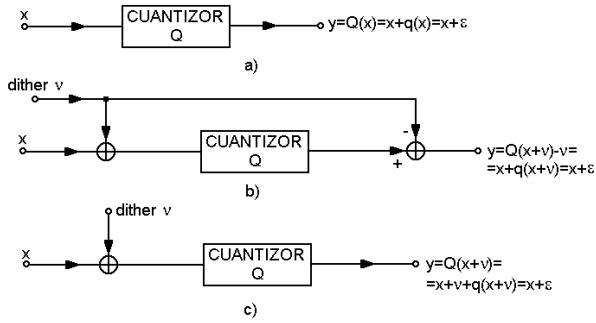


Fig 2.3

## Formarea zgomotelor

În lumea de azi suntem constant bombardăți de sunete. Multe dintre sunete sunt plăcute, dar din nefericire există situații în care sunetele devin zgomot.

Motivul pentru care există diferențe în ceea ce privește perceperea zgomotului este acela că unii oameni sunt mai sensibili decât alții. Dar suntem cu toții de acord ca zgomotul este un sunet nedorit. În termeni specifici, un zgomot puternic este un sunet care poate cauza o pierdere de auz.

Pentru persoanele care au o pierdere de auz, zgomotul de fond reprezintă o problemă particulară pentru că abilitatea lor de a înțelege vorbirea în medii zgomotoase este foarte limitată. Pe langă aceasta există o problema că unele apărate auditive amplifică nu numai vorbirea, dar și zgomotul de fond, facând vorbirea foarte dificilă în medii zgomotoase.

Într-o situație reală vocea umană, în cele mai multe cazuri nu este pură. Ea este frecvent însorită de zgomot. Zgomotul dacă depășește anumiți parametri devine supărător și poate chiar să disturbe în mod serios o conversație, ducând chiar la o lipsă de înțeligență dacă zgomotul se regăsește într-un alt domeniu de frecvențe, altul decât cel în care se află vocea umană, atunci el poate fi înălțat.

Detectia de voce este frecvent utilizată în sistemele de supraveghere și în spionaj, în

situării reale în care serviciile de spionaj încearcă să intercepteze o conversație între două sau mai multe persoane aflate în apropierea unor surse de zgomot. Interceptia se face cu ajutorul camerelor video și a microfoanelor dotate cu funcții de detecție de voce, plasate în apropiere. dacă în timpul discuției, zgomotul produs este puternic, poate disturba în mod serios discuția, ajungând chiar la situația de neînțelegere a conversației. Dar cu ajutorul algoritmilor de filtrare putem să înlăturăm o mare parte din zgomot și retine doar vocea.

În timpul înregistrărilor audio apar zgomote de fond (aparatul de aer conditionat, motorașul unui harddisk, vântul, etc). Pentru eliminarea acestor zgomote se pot utiliza trei tipuri de restaurare audio: blocarea zgomotului („noise gating”), reducerea zgomotului („noise reduction”) și înlăturarea click și pop („click and pop removal”).

Poarta de zgomot („noise gate”) blochează semnalele cu amplitudine sub un anumit prag. Cand eșantioanele sunt sub valoarea de prag, poarta se închide și eșantioanele nu sunt lăsate să treacă. Cand eșantioanele cresc deasupra valorii de prag poarta se deschide și lăsă să treacă aceste eșantioane. O poartă permite specificarea unor parametri:

- nivel de reducere („reduction level”): amplitudinea sub care sunt reduse eșantioanele
- timp de atac („attack time”): timpul de deschidere a porții când semnalul depășește valoarea de prag (timp scurt poate să prindă de exemplu un sunet de tobe)
- timpul de eliberare („release time”): timpul de închidere a porții din momentul în care semnalul scade sub prag (o bucată muzicală care scade încet în volum trebuie să aibă un timp de eliberare mare pentru a prinde efectul)
- control de deținere („hold control”): timpul minim cât poarta trebuie să stea deschisă
- control de histerezis („hysteresis control”): este util în cazul când semnalul fluctuează în jurul valorii de prag, ceea ce ar însemna închiderea și deschiderea continuă a porții (fenomen numit „chatter”), astfel se poate indica diferența dintre valoarea care deschide poarta și valoarea care o închide.



Fig 2.4. Interfața pentru poarta de zgomot (Logic Pro)

Interfețele pentru reducerea zgomotului afișează de obicei graficul analizei de frecvență: pe axa orizontală, se reprezintă frecvențele, iar pe axa verticală amplitudinile. Utilizând culori diferite se reprezintă semnalul original, cantitatea de reducere a zgomotului și zgomotul. Setarea principală se referă la reducerea zgomotului.

Profilul zgomotului se obține din analiza Fourier (se determină frecvențele la care apar zgomotele și amplitudinile corespunzătoare acestor frecvențe). După determinarea profilului zgomotului se compară spectrul de frecvențe al întregului semnal cu profilul, iar sectiunile care coincid sunt eliminate.

O astfel de interfață permite setarea unor parametrii de netezire:

- netezire de timp: timpii de atac și eliberare pentru reducerea zgomotului
- netezire de frecvență: gradul la care zgomotul identificat într-o bandă de frecvență afectează schimbările de amplitudine în benzile de frecvențe învecinate
- netezire de tranziție: setează o gama între amplitudini care sunt considerate zgomote și cele care nu sunt considerate zgomote

Software-ul pentru reducerea zgomotului poate fi setat pentru a căuta un anumit tip de zgomot:

- zgomot alb („white noise”): apare la amplitudini egale (sau aproape egale) pentru toate frecvențele, echivalent cu energii egale în benzi de frecvență de aceleași

- dimensiuni (graficul frecvență-amplitudine aproximativ orizontal, deci ariale de sub grafic sunt egale pentru benzi egale)
- zgomot roz („pink noise”): energii egale în benzi de octave (pentru o notă, trecerea de la o octavă la octava urmatoare înseamnă dublarea frecvenței)

Eliminatorul de click și pop detectează într-o porțiune selectată a fisierului audio o schimbare bruscă de amplitudine și elimină această schimbare prin interpolarea unei de sunet între punctele de start și sfârșit a click-ului sau pop-ului.

## Prelucrarea dinamică

Prelucrarea dinamică este procesul de ajustare a scării dinamice a unei selecțiuni audio fie pentru a reduce, fie pentru a crește diferența dintre pasajele cele mai puternice (tari) și cele mai reduse (silentoase). În producția de muzică, vocile și instrumentele sunt înregistrate separat, fiecare pe pista sa, iar fiecare pistă se poate ajusta dinamic în timp real sau după înregistrare.

Modificarea scării dinamice se poate realiza prin patru metode:

- compresie în jos („downward compression”): micșorează amplitudinile semnalului care sunt peste o anumită valoare de prag, fără să schimbe amplitudinile sub această valoare, reducând astfel scara dinamică
- compresie în sus („upward compression”): ridică amplitudinile semnalului care sunt sub o anumită valoare, fără modificarea amplitudinilor peste această valoare, reducând scara dinamică
- expansiune în sus („upward expansion”): ridică amplitudinile semnalului care sunt peste o valoare de prag, fără să schimbe amplitudinile sub această valoare, crescând scara dinamică
- expansiunea în jos („downward expansion”): micșorează amplitudinile semnalului sub o valoare de prag, fără modificarea amplitudinilor peste această valoare, crescând scara dinamică

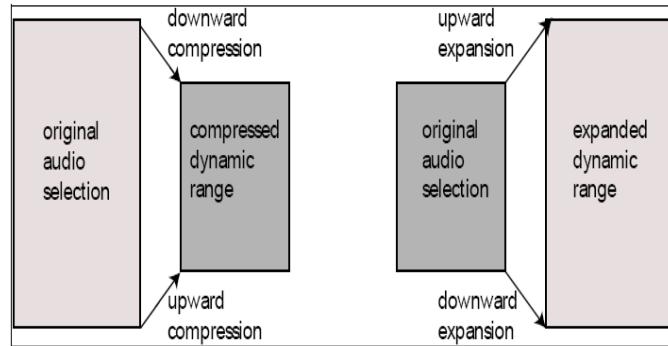


Fig 2.5

Limitarea audio limitează amplitudinea unui semnal audio la un nivel desemnat. Se poate realiza hardware, la înregistrare sau software, executându-se tăierea amplitudinilor mari („clipping”). Rezultă anumite distorsiuni ale formei de undă.

Normalizarea este un proces care crește amplitudinea unui semnal audio și deci și volumul percepției. Deoarece operează pe un întreg semnal audio trebuie aplicat după înregistrare.

Algoritmul de normalizare:

- 1) găsește amplitudinea cea mai mare din selecția audio;
- 2) găsește amplificarea necesară pentru a crește amplitudinea cea mai mare la valoarea maximă dorită (implicit 0 dBFS sau o anumită valoare setată de utilizator);
- 3) mărește toate amplitudinile din selectie cu această amplificare.

Varianta acestui algoritm normalizează amplitudinea RMS la o valoare de decibel specificată de utilizator, caci RMS furnizează o măsură mai bună pentru volumul percepției al sunetului. În unele programe se furnizează o setare predefinită împreună cu o explicație intuitivă (exemplu: “Normalize RMS to -10dB (speech)”).

Compresia și expansiunea se pot reprezenta matematic printr-o funcție de transfer sau grafic. O funcție de transfer corespunzătoare primei bisectoare (linie la  $45^\circ$ ) nu modifică

semnalul audio, însă o funcție liniară de transfer deasupra primei bisectoare crește amplitudinea, iar sub prima bisectoare scade amplitudinea.

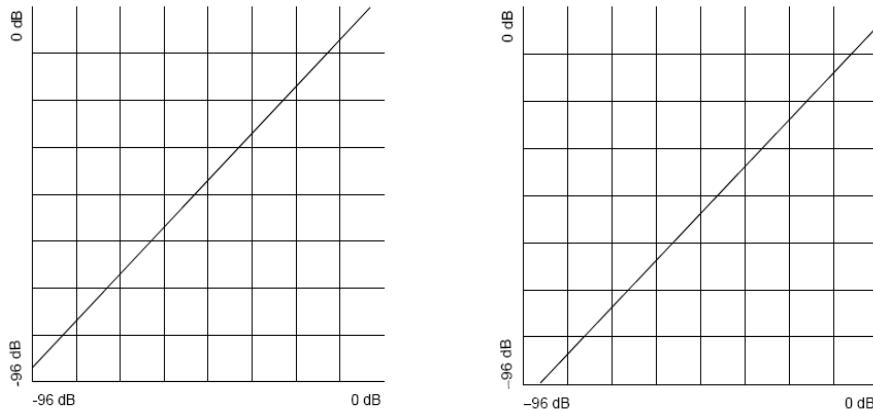


Fig 2.6

Compresia în jos necesită un prag (amplitudinile peste această valoare sunt micșorate).  
Exemplu de funcție de transfer:

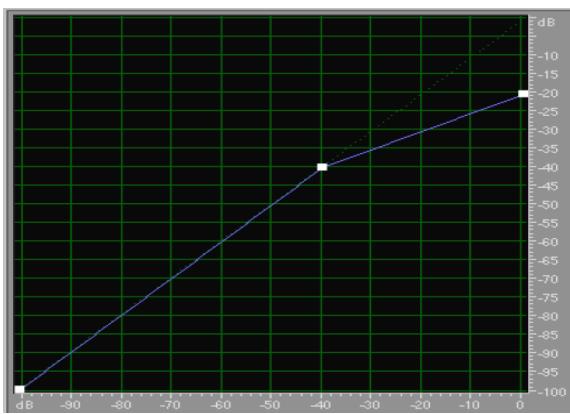


Fig 2.7

Amplitudinile mai mari decât -40 dB sunt micsorate cu un raport 2:1.

Compresia în sus crește amplitudinea semnalului mai mic decât o valoare de prag.

Exemplu de funcție de transfer:

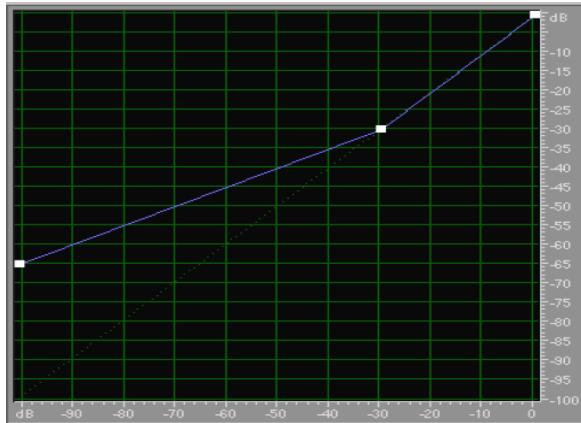


Fig 2.8

Amplitudinile mai mici de -30 dB sunt amplificate cu un raport 2:1.

Unele programe permit combinarea celor două operații într-o singură. În acest caz funcția de transfer poate arăta astfel:

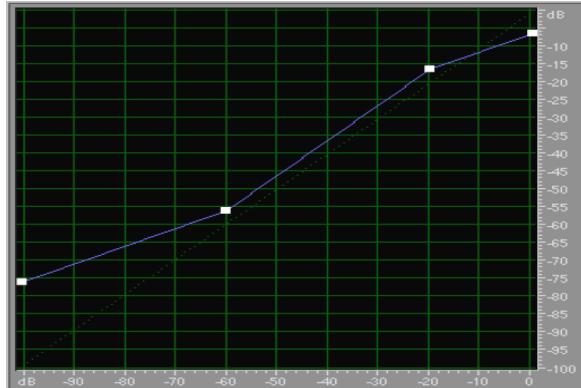


Fig 2.9

Se execută compresie în jos pentru amplitudinile mai mari decat -20 dB, fără compresie între -20 și -60 dB și compresie în sus sub -60 dB. Scara dinamică a fost redusă prin compresie în jos și compresie în sus.

## **CAPITOLUL III : ANALIZA DE FRECVENTĂ**

### **Seria Fourier**

Sistemele liniare invariante în timp sunt de departe cele mai studiate și utilizate sisteme în prelucrarea semnalelor. Un sistem se numește liniar dacă răspunsul acestuia la suma a două semnale este identic cu suma răspunsurilor la fiecare semnal în parte. Un sistem se numește invariant în timp dacă răspunsul său la un semnal este același indiferent de momentul când este aplicat semnalul respectiv la intrarea sistemului. Din teoria sistemelor, se știe că funcțiile proprii ale sistemelor liniare invariante în timp (pe scurt, SLIT) sunt (co)sinusoidale. Altfel spus, dacă la intrarea unui SLIT aplicăm o cosinusoidă pură de frecvență  $\omega_0$ , atunci la ieșire vom avea tot o cosinusoidă pură  $\omega_0$  (bineînțeles, având altă amplitudine și fază). Acest fapt permite studierea comportamentului sistemului la un semnal de intrare oarecare, cu condiția să putem scrie semnalul respectiv ca o sumă (fie și infinită) de sinusoide.

Semnalele periodice pot fi scrise ca o sumă numărabilă de componente sinusoidale ale căror amplitudini și faze pot fi calculate cu ușurință din semnalul respectiv, acestea fiind seriile Fourier. Transformata Fourier generalizează aceasta descompunere de semnal într-o sumă de sinusoide și pentru semnalele neperiodice.

Să începem prin a studia forma spectrului semnalului periodic în funcție de perioada sa . Se observă că pe măsură ce crește, componente din spectrul semnalului se “îndesesc”. Acest lucru este natural, întrucât creșterea lui este echivalentă cu scăderea frecvenței fundamentale  $\Omega = 2\pi/T$ , și deci, cu scăderea intervalului de frecvență între două componente succesive. Fig 3.1 ilustrează un exemplu. Evident, la limită, când  $T \rightarrow \infty$ , componentele frecvențiale se “contopesc”, iar spectrul semnalului devine de natură continuă.

Ajungem, deci, la definiția transformatei Fourier:

Fie  $x(t)$  un semnal de modul integrabil:  $\int_{-\infty}^{\infty} |x(t)| dt = M < \infty$  (1). Atunci, se definește transformata Fourier a semnalului  $x(t)$  ca fiind semnalul  $X(\omega)$  obținut după:

$$X(\omega) = F\{x(t)\}(\omega) = \int_{-\infty}^{\infty} x(t) e^{-i\omega t} dt \quad (2).$$

iar semnalul original  $x(t)$  poate fi recuperat din transformata sa prin aplicarea operatorului invers:

$$x(t) = F^{-1}\{X(\omega)\}(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} X(\omega) e^{i\omega t} d\omega \quad (3).$$

Este important, pentru înțelegerea noțiunilor, să observăm similitudinile și diferențele între relațiile (2) și (3) și cele care descriu descompunerea în serie Fourier complexă a unui semnal periodic:

$$A_{nc} = \frac{1}{T} \int_T x(t) e^{-in\Omega t} dt, \forall n \in \mathbb{Z}$$

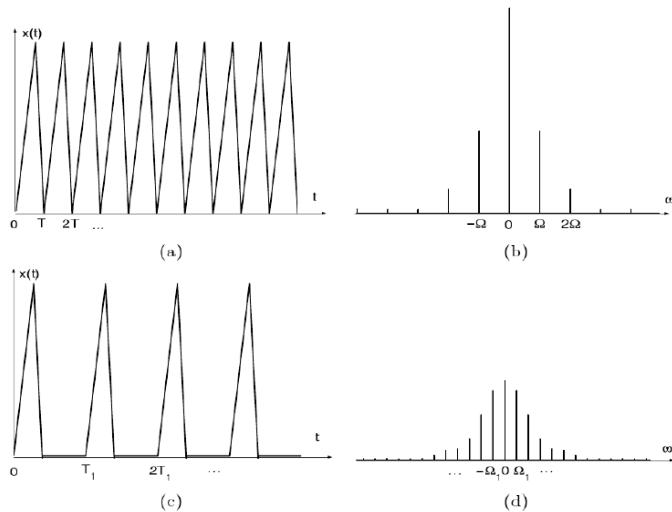


Fig 3.1: Forma spectrului unui semnal periodic în funcție de perioadă: (a)

Semnal periodic de perioadă T. (b) Modulul coeficienților  $A_{nc}$  pentru semnalul din figura (a). (c) Semnal periodic de perioada  $T_1 > T$ . (d) Modulul coeficienților  $A_{nc}$  pentru semnalul din figura (c).

Se observă că semnificația valorilor  $X(\omega)$  este similară cu cea a coeficienților  $A_{nc}$ , cu singura diferență că, în cazul transformatei Fourier, numărul de cosinusoide în care se descompune semnalul devine infinit nenumărabil. Modulul  $|X(\omega)|$ , și faza  $\phi(\omega)$  ale cantității complexe  $X(\omega) = |X(\omega)|e^{j\phi(\omega)}$  sunt amplitudinea, respectiv faza cosinusoidei de frecvență  $\omega$  ce intră în descompunerea spectrală a semnalului  $x(t)$ . Într-adevăr, observând că, în ipoteza unui semnal  $x(t)$  cu valori reale, valorile transformatei Fourier situate simetric față de 0 sunt complex conjugate:

$$x(t) \in \mathbb{R} \forall t \Leftrightarrow X(-\omega) = X^*(\omega) \Leftrightarrow \begin{cases} |X(-\omega)| = |X(\omega)| \\ \varphi(-\omega) = -\varphi(\omega) \end{cases} \quad (4)$$

Atunci (3) poate fi rescrisă ca:

$$\begin{aligned} x(t) &= \frac{1}{2\pi} \left( \int_{-\infty}^0 X(\omega) e^{i\omega t} d\omega + \int_0^\infty X(\omega) e^{i\omega t} d\omega \right) \xrightarrow{\omega = -\Omega} \\ x(t) &= \frac{1}{2\pi} \left( - \int_{-\infty}^0 X(-\Omega) e^{-i\Omega t} d\Omega + \int_0^\infty X(\omega) e^{i\omega t} d\omega \right) = \\ &\frac{1}{2\pi} \left( \int_0^\infty X^*(\Omega) e^{-i\Omega t} d\Omega + \int_0^\infty X(\omega) e^{i\omega t} d\omega \right) = \\ &\frac{1}{2\pi} \int_0^\infty X(\omega) e^{i\omega t} + [X(\omega) e^{i\omega t}]^* d\omega \end{aligned} \quad (5).$$

În continuare, folosind faptul că  $\forall z \in \mathbb{C}$ , avem  $z+z^* = 2 \Re(z)$  și că  $\Re(X(\omega)e^{i\omega t}) = \Re(|X(\omega)|e^{i(\omega t+\phi(\omega))}) = |X(\omega)| \cos(\omega t + \phi(\omega))$ , (5) devine:

$$x(t) = \frac{1}{\pi} \int_0^\infty |X(\omega)| \cos(\omega t + \varphi(\omega)) d\omega \quad (6).$$

Un filtru audio digital modifică amplitudinea sau faza uneia sau mai multor componente ale unui semnal audio. Filtrele audio digitale se pot clasifica în două categorii:

- filtre cu răspuns finit la impuls (FIR – “finite-inpulse response”)
- filtre cu răspuns infinit la impuls (IIR – “infinite-inpulse response”)

Filtrul FIR: Fie  $x(n)$  un semnal audio digital cu  $L$  eșanțioane,  $0 \leq n \leq L-1$  și  $y(n)$  semnalul audio filtrat. Fie  $h(n)$  o mască de convoluție operând ca filtru FIR de lungime  $N$ . Funcția de filtrare FIR este definită:

$$y(n) = h(n) \otimes x(n) = \sum_{k=0}^{N-1} h(k)x(n-k) \quad (7).$$

unde  $x(n-k) = 0$  dacă  $n-k < 0$ , iar  $\otimes$  reprezintă operatorul de convolutie.

Avantajul principal al filtrelor IIR este că permit o tăiere abruptă între frecvențele care sunt filtrate și cele care nu sunt filtrate. Filtrele FIR necesită măști mai largi și deci mai multe operații aritmetice pentru a avea aceeași eficiență de filtrare. De asemenea FIR necesită mai multă memorie și timp de prelucrare.

Alt avantaj al IIR este că aceste filtre se pot proiecta dinfiltrele corespunzătoare analogice. Filtrele FIR nu au corespondențe analogice.

Avantajul filtrelor FIR este că pot fi constrânsă să aibă un răspuns liniar de fază (deplasările de fază ale componentelor de frecvență sunt proporționale cu frecvența). Astfel frecvențele armonice sunt deplasate în aceeași proporție și relațiile dintre armonice nu sunt distorsionate (foarte important în muzică).

Alt avantaj al filtrelor FIR este că nu sunt aşa de sensitive la zgomot rezultat din adâncimea mică de biți și eroarea de rotunjire.

Spectrul de frecvență al unui semnal este reprezentarea sa în domeniul frecvență. Spectrul de frecvență al unui semnal este reprezentarea pe axa reală (axa frecvențelor) a coordonatelor semnalului măsurate într-un spațiu infinit dimensional în urma unei transformări numită transformata Fourier, spațiu unde fiecare dimensiune corespunde unei frecvențe, ca posibilă componentă a semnalului. Acest spatiu este un spațiu de funcții ortonormate și este același indiferent de semnalul analizat. În perspectivă, el este spațiul în care transformata Fourier a unui semnal este reprezentată ca un vector și reprezintă domeniul de frecvență al semnalelor (în general o transformată reprezentă o bijecție între axa reală sau orice spațiu finit dimensional și un spațiu infinit dimensional de funcții ortonormate).

Relația dintre convoluție și transformata Fourier:

Etapele operației de filtrare:

- calculează  $X(z)$  transformata Fourier a semnalului audio digital  $x(n)$ ;
- determină specificațiile filtrului;
- pe baza specificațiilor calculează  $H(z)$  astfel incat  $Y(z)=H(z)X(z)$ , iar  $Y(z)$  are componentele de frecvență dorite;
- execută înmulțirea  $H(z)X(z)$  pentru a obține  $Y(z)$ ;
- execută transformarea Fourier inversă a lui  $Y(z)$  pentru a obține  $y(n)$ , semnalul filtrat în domeniul timp.

Partea mai dificilă a algoritmului este gasirea lui  $H(z)$ . Calculul transformatei Fourier este echivalentă ca volum de calcul cu operația de convoluție în domeniul timp:

$$y(n) = h(n) \otimes x(n) = \sum_{k=0}^N h(k)x(n-k)$$

Echivalența celor două operații este dată de teorema convolutiei:

Fie  $H(z)$  transformata Fourier discretă a filtrului de convolutie  $h(n)$  și fie  $X(z)$  transformata Fourier discretă a semnalului audio digital  $x(n)$ . Atunci:  $y(n) = h(n) \otimes x(n)$  este echivalentă cu transformata Fourier discretă inversă  $Y(z) = H(z)X(z)$ .

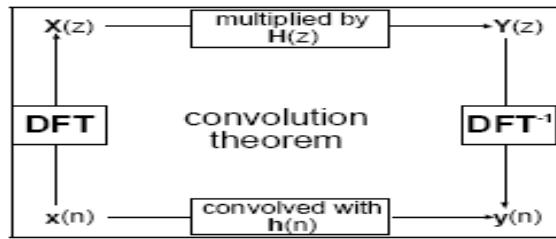


Fig 3.2

## Transformata Fourier discretă (TFD)

Principii:

- Se trece din timp discret în domeniu discret de frecvențe;
- Pentru N eșantioane ale unui semnal, TFD fixează domeniul de frecvențe:  $\{0, 1/N, 2/N, \dots, N-1/N\}$
- Ușor de interpretat și utilizat în practică;
- N dimensiuni  $\Rightarrow$  transformare inversabilă;
- Noi putem alege N astfel încât să avem rezoluția dorită;
- TFD diferă de DTFD, unde frecvențele sunt un domeniu continuu în  $[-, ]$ ;
- TFD cel mai des folosită în practică în varianta FFT (Fast Fourier Transform).

Este folosită în calculul spectrului unui semnal.

Transformarea Fourier a unui semnal permite analiza semnalului în raport cu frecvența, analiză extrem de importantă în studiul ulterior al modului în care semnalul se propagă prin diverse sisteme. Transformata Fourier Discretă (TFD) și Transformata Fourier Discretă Rapida

(TFDR) sunt instrumente care permit calculul facil al spectrului de frecvență al unei secvențe de date. Spectrul secvenței de date realizat pe baza relației:

$$U_{TFD[n]} = \sum_{k=0}^{N-2} u_{[k]} e^{-i \frac{2\pi n k}{N}}, \text{ unde } n = 0, 1, 2, \dots, N-1 \quad (8).$$

reprezintă un “alter ego” al acesteia, putând fi folosit la identificare, clasificare, comparare. Trebuie să vedem acum dacă spectrul secvenței de date este același cu spectrul semnalului din care acesta s-a prelevat. Prin analogie cu Teorema Fourier, care se referă la semnale discrete care au un spectru discret, trebuie menționat că, similar în cazul TFD, dacă dispunem de un spectru discret, înseamnă că secvența de date de la care acesta provine este periodică. Deci secvența de  $N$  date căreia îi se aplică TFD, este privită ca provenind dintr-un semnal periodic, având perioada egală cu  $N$  de unde  $T$  reprezintă perioada de eşantionare. Reciproc, dacă aplicăm TFD unei secvențe de  $N$  date, semnalul căruia îi va corespunde spectrul rezultat se obține multiplicand prin periodicitate această secvență.

Cele menționate au consecințe importante. Să analizăm exemplul urmator în care TFD este aplicată inițial unei secvențe ce conține un număr întreg de perioade, dintr-un semnal sinusoidal, situație reflectată fidel în spectrul său.

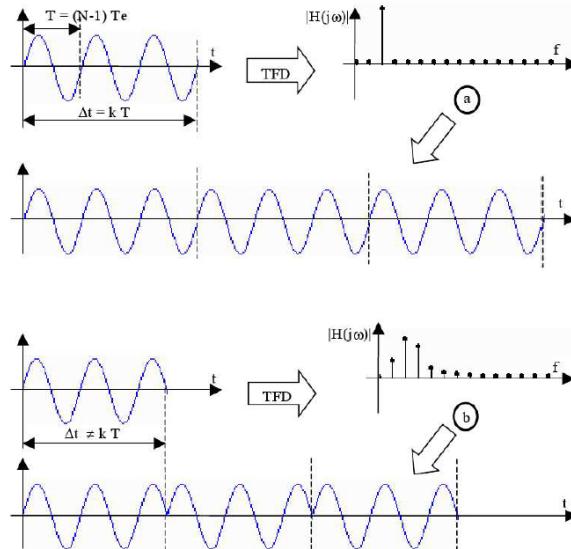


Fig 3.3 Spectrul dat de TFD pentru secvențele de date

Se observă că în al doilea caz, atunci când secvența nu conține un număr întreg de perioade, spectrul rezultat nu este cel corect fiindcă el este în fapt spectrul semnalului rezultat prin multiplimultiplicarea prin periodicitate a secvenței, reprezentat în figura 3.3.b, care nu este sinusoidă. În concluzie, dacă trebuie determinat spectrul unui semnal folosind TFD, atunci în cazul în care semnalul este periodic, secvența de  $N$  eșantioane “prelevată” din semnal trebuie să conțină un număr întreg de perioade.

## Functii fereastră

Atunci cand se preia o porțiune din  $N$  eșantioane dintr-un semnal, fără a le schimba valoarea, se zice că se preia o freastră dreptunghiulară. Am văzut că prin TFD putem obține spectrul corect numai dacă semnalul analizat este periodic și numai preluând o fereastră dreptunghiulară care conține un număr întreg de perioade.

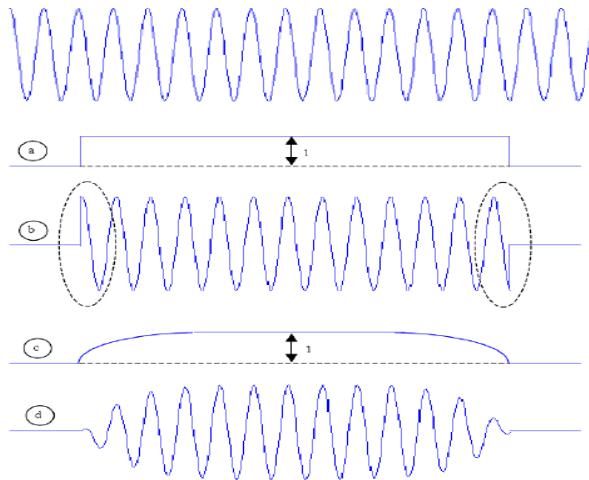


Fig 3.4 Portiune de semnal preluată cu și fără

În caz contrar, alterarea spectrului de frecvență se datorează cu prioritate zonelor de margine ale ferestrei. Acestea sunt privite ca ”făcând parte din semnal”, ori este evident că semnalul original nu are astfel de salturi, precum zonele încercuite din figura 3.3.a.

Soluția de înlăturare a variațiilor mari din zona de margine a unei porțiuni de semnal, o

constituie aplatizarea acestora. Acesta este obiectivul metodelor de "ferestruire". Algoritmul este cel descris de relația:

$$u_{w[kTe]} = w_{[k]} u_{[kTe]}, \text{ pentru } k = 0, 1, \dots, (N - 1) \quad (9).$$

Se observă că porțiunea prelevată se înmulțește cu funcția  $w_{[k]}$ , numită "funcție fereastră". Așa cum este arătat în figura 3.4, funcția fereastră trebuie să aibă amplitudinea unitară pe toată lungimea porțiunii de semnal prelevate, mai puțin în zonele de capete unde ea trebuie să descrească uniform către zero. Există mai multe funcții care au această proprietate, unele dintre ele consacrate deja în literatura de specialitate, ca de exemplu: fereastra triunghiulară (relația 10), fereastra Welch (relația 11), fereastra Hanning (relația 12).

$$w_{[k]} = 1 - \left| \frac{2k - (M - 1)}{M + 1} \right| \quad (10)$$

$$w_{[k]} = 1 - \left( \frac{2k - (M - 1)}{M + 1} \right)^2 \quad (11)$$

$$w_{[k]} = \frac{1}{2} \left[ 1 - \cos \left( \frac{2\pi k}{M - 1} \right) \right] \quad (12)$$

În exemplele date, M poate fi egal cu lungimea secvenței prelevate (N), sau poate fi mai mic decât N, și atunci algoritmii 10, 11, 12 se "împart în două", câte o jumătate pentru fiecare capăt al secvenței, așa cum este ilustrat în figura 3.4.

În concluzie, procedeul numeric de ferestruire se aplică secvenței numerice căreia urmează să-i fie aplicată transformarea TFD sau altă transformare. Ferestruirea are rolul de a reduce contribuția nefastă a porțiunilor de capăt ale secvenței prelevate, în spectrul de frecvență al semnalului. Tehnica de ferestruire mai este folosită pentru a ajusta și alți algoritmi de procesare numerică și anume pe cei cu rolul de filtru numeric.

## Cepstrumul și pitch-ul unui semnal audio

Cepstrumul este inversa transformatei fourier a logaritmului spectrului semnalului. Există un cepstul complex, un cepstrum real, un cepstum de putere (power cepstrum) și un cepstum de fază. Cepstrumul de putere în particular are aplicații în analiza vorbirii umane.

Cepstrumul de putere a fost definit în 1963 de către Bogert ca  $F(\log(|F(f(t)|^2))^2)$  (1) iar o analiză a cepstrumului pe timp scurt, propusă de Schroeder și Noll, poate determina pitchul vorbirii umane.

Cepstrumul poate fi văzut ca o informație despre gradul de schimbare a diferențelor benzi din spectru. El a fost inventat pentru a caracteriza undele seismice cauzate de cutremure și explziile bombelor dar de asemenea fost folosit pentru a determina frecvențele fundamentale ale vorbirii umane. Pitchul determinat cu ajutorul cepstrumului este în particular eficient din cauza că efectele excitării vocale și ale tractului vocal sunt sunt aditive în logaritmul cepstrumului de putere, notiuni cunoscute și ca "Pitch" și "Formant"-e.

Cepstrumul e o reprezentare folosită în procesarea semnalelor homomorfice pentru a converti semnale combinate prin convoluție în sumele lor cepstrele pentru o separare liniară. În particular cepstrumul de putere este des folosit pentru reprezentarea vocii umane și a semnalelor muzicale.

Pitchul este o proprietate perceptibilă care permite sortarea sunetelor în funcție de frecvență. El poate fi considerat ca fiind o frecvență dar nu este o proprietate fizică pură ci este un atribut psihacustic subiectiv al sunetului. Din punct de vedere istoric, studiul și percepția lui a fost o problema fundamentală în psihacustica și a fost folosit ca un instrument în formarea și testarea teoriilor despre reprezentarea, procesarea și percepția de către sistemul auditiv al sunetelor. Pitchul este o sensație auditivă în care ascultatorul asociază tonurile muzicale cu poziții relative pe o scara muzicală, asociere bazată primordial pe frecvența vibratiei. Există o relație strânsă între pitch și frecvență dar nu sunt echivalente. Undele sonore nu au pitch și oscilațiile lor pot fi măsurate pentru a obține frecvență, dar este nevoie de un creier uman pentru a asocia calitatea internă a pitchului. Pitchul este de obicei cuantificat ca frecvență în cicluri pe secundă, sau hertz, comparând sunete cu tonuri pure, care au forme sinusoidale periodice.

Intonatia cuprinde schimbari in pitch, intensitate si viteza in vorbire. In limbile tonale, in cele mai multe cazuri, tonul unei silabe este determinat de vocala componenta, ceea ce inseamna ca pitchul care marcheaza tonul este impus de catre vocala, ceea ce face recunoasterea silabelor cu ajutorul pitchului impractica dar este o metoda foarte eficienta pentru determinarea vocalei predominante dintr-o silaba.

## **CAPITOLUL IV : ARHITECTURA APlicațIEI**

### **Descrierea aplicației**

În lucrarea de față am abordat tema prelucrării digitale a sunetelor. Am ales această temă deoarece consider că interfețele om-mașină vor ajunge într-un viitor apropiat să se bazeze pe comunicarea verbală cu calculatorul. Pentru aceasta este nevoie de recunoaștere automată a sunetelor. Pentru limba română încă nu există un sistem de recunoaștere a vorbirii, motiv pentru care domeniul este foarte atractiv.

Tema mea s-a axat doar pe realizarea unui sistem care este capabil să recunoască vocalele mai multor utilizatori.

Sistemul de recunoaștere implementat în acest proiect funcționează în două etape: etapa de antrenare a sistemului și etapa de recunoaștere (etapa de testare). În faza de antrenare, cât și în etapa de recunoaștere propriu-zisă se face extragerea unui vector de valori specifice fiecarei vocale înregistrată. Aceste valori reprezintă pitch-ul semnalului audio (frecvența cu amplitudinea maximă a unei înregistrări) pe ferestre de diferite lungimi.

Aplicația nu stochează semnalul audio în nici un fisier ci folosește doar un buffer în care este păstrat semnalul audio în format PCM, cu 44100 sample-uri pe secundă iar fiecare sample este stocat pe 16 biți și este interpretat ca un întreg cu semn.

Aplicatia lucreaza doar cu intervale de sunet ce contin doar informatie utila, fara intervale de liniste. Acest lucru se realizeaza folosind un fir de executie ce inregistreaza semnalul primit de la microfon si il imparte in esantioane de lungimea unei secunde. Un alt thread foloseste aceste esantioane si concateneaza parti din ele care nu contin intervale de liniste.

Procesul prin care se face extractia vectorului de valori mentionat mai sus este urmatorul:

- 1) Intrarea în acest proces este un interval din semnalul captat de microfon care conține doar sunet util, fara intervale de liniste.
- 2) Acest interval este împărțit în ferestre de lungime  $2^k$  (unde  $k$  aparține multimii

{6,7,8,9,10}).

- 3) Pentru fiecare fereastra se calculeaza pitchul folosind metoda cepstrumului si apoi se face media lor, obtinand astfel 5 valori medii pentru cele 5 lungimi de fereastra.
- 4) Aceste 5 valori formeaza vectorul de valori care caracterizeaza intervalul de sunet.

O poza valoreaza cat o mie de cuvinte:

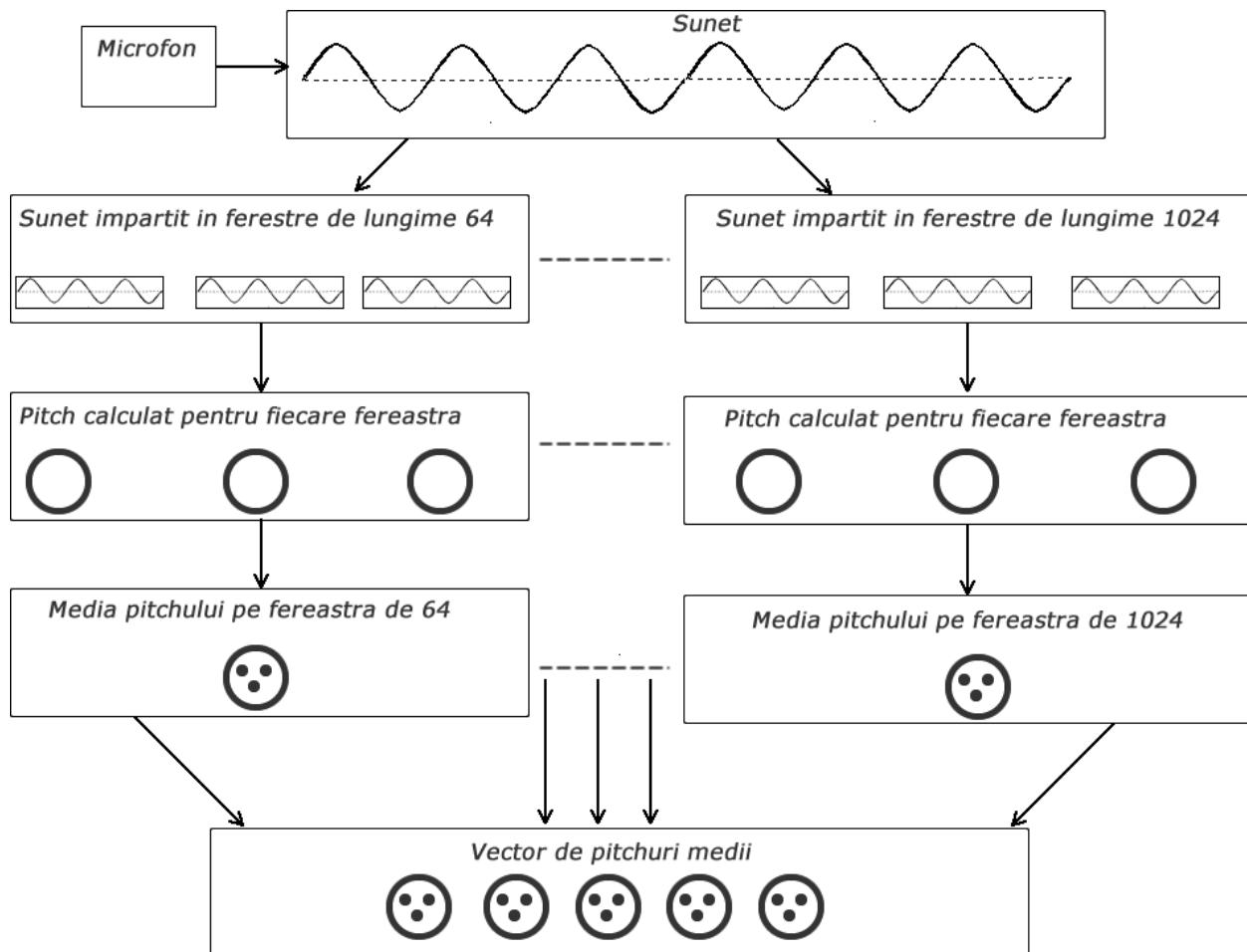


Fig 4.1 Diagrama ce ilustreaza modul in care este calculat vectorul de pitchuri pentru semnalul unui sunet

Etapa de antrenare constă în folosirea mai multor înregistrări a aceleiași vocale pentru a calcula o medie a vectorilor asociati fiecarei înregistrări.

Etapa de recunoastere constă în calcularea vectorului asociat sunetului ce urmează a fi recunoscut pentru a determina distanța făcătoare de vectorii medii calculați pentru fiecare vocală în etapa de antrenare. Distanța folosită în aplicație este calculată după formula:  
$$\sum_{i=0}^4 ((A[i] - B[i])^2)$$
, unde  $A[i]$  sunt valorile pitch-urilor salvate în baza de date pentru o vocală și  $B[i]$  sunt valorile pitch-urilor pentru sunetul care vrem să-l identificăm.

Se vor calcula erorile pentru fiecare vocală și valoarea sumei cea mai mică, adică cel mai parțial sunet de sunetul pe care îl vrem să-l identificăm este rezultatul întors de aplicație.

Interfața grafică a aplicatiei este una simplă și intuitivă ceea ce face ca aplicația să fie foarte ușor de folosit. Ea conține o singură fereastră în care este afișată vocala recunoscută, sau cea căreia trebuie rostita în etapa de antrenare.

În figura de jos este reprezentată fereastra aplicației în modul de recunoastere după ce a recunoscut vocala a :

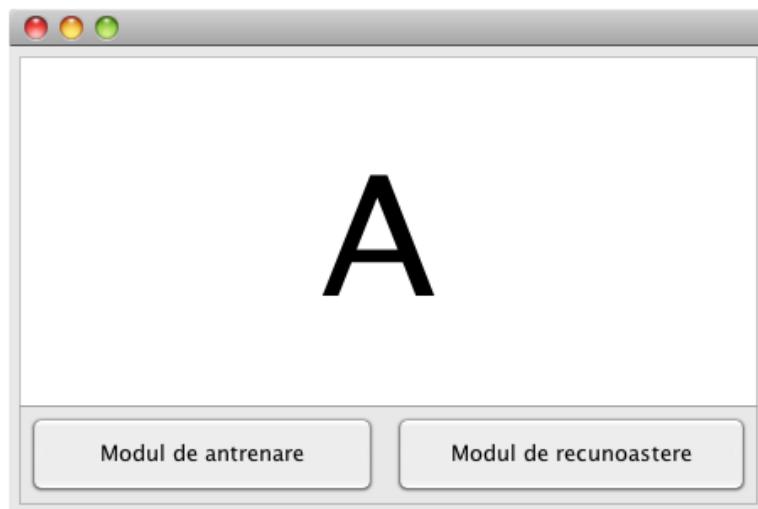


Fig 4.2 Fereastra aplicației în modul de recunoastere

In figura de mai jos se află fereastra aplicatiei în modul de antrenare după ce a înregistrat cu succes vocala „A” rostită de utilizator.

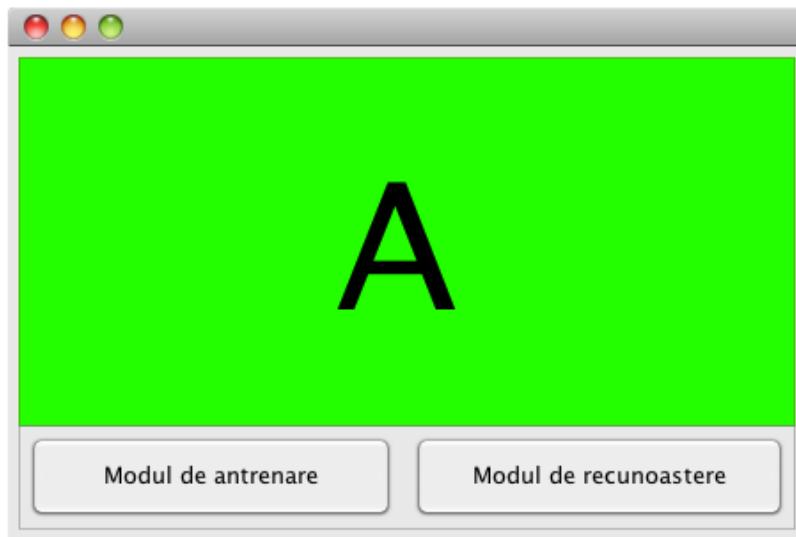


Fig 4.3 Fereastra aplicatiei in modul de antrenare

## Prezentarea utilității aplicației

Apliția poate ajuta un sistem mult mai complex de recunoștere a limbajului vorbit. Sunetele de bază ale vorbirii sunt împărțite în două categorii: vocale și consoane.

Propunem în continuare o caracterizare articulatorie a sunetelor limbii române; menționăm că această caracterizare a fost dedusă, în principiu, din literatura de specialitate (Beldescu, 1984; Ciompec, 1985; Goga, 2001; Ionescu, 2001; Vasiliu, 1965), dar setul de trăsături articulatorii a fost într-o anumită măsură unificat și apoi optimizat pentru a se preta la o prelucrare automată pe calculator.

Vocalele se disting între ele în primul rând prin înălțime (dată de frecvența fundamentală de Vibrație a coardelor vocale) și timbru (reprezentat prin conținutul de armonici care se alătură frecvenței fundamentale), însușiri determinate, la rândul lor, de volumul și forma rezonatorului bucal. Astfel, trăsăturile distinctive ale sistemului vocalic, care sunt în același timp și criterii de

identificare, se referă la gradul de deschidere, la localizare și la participarea (sau neparticiparea) buzelor la articulare.

În funcție de unghiul de deschidere a maxilarelor (apertura):

- vocale deschise: a
- vocale medii (semideschise): e, ă, o
- vocale închise: i, î (â), u

În funcție de localizarea lor, deci de locul de articulare

- vocale anterioare: e, i
- vocale centrale (neutre): a, ă, î (â)
- vocale posterioare (postpalatale): o, u

După modul în care sunt însotite sau nu de rotunjirea buzelor:

- vocale labializate (rotunjite): o, u
- vocale nelabializate: a, e, i, ă, î (â)

Natura diversă a procesării limbajului vorbit înglobează diverse ramuri ale științei: informatică, mathematică, inginerie electrică, gramatică și psihologie. Procesarea vorbirii este un subiect complex, care se bazează pe cunoștințe ale mai multor nivele, cum ar fi: acustic, fonetic, lingvistic, lexical, sintactic, semantic, pragmatic. Principalele domenii ale tehnologiei vorbirii sunt: recunoașterea automată, sinteza automată și codificarea vorbirii.

Recunoașterea automată a vorbirii este procesul de transformare a semnalului acustic continuu produs de organul fonator uman într-o reprezentare discretă căreia î se poate ataşa o semnificație și care, când e înțeleasă, poate fi folosită pentru a determina un răspuns. Există mai multe nivele de analiză a vorbirii în vederea recunoașterii/înțelegerei vorbirii:

- analiza acustica se referă la extragerea parametrilor acustici ai semnalului vocal
- analiza fonetica reprezintă evidențierea caracteristicilor sunetelor vorbirii
- fonologia se ocupă cu analiza variabilităților în pronunțare

- analiza prozodică se referă la utilizarea informației despre intonație, accent, ritm pentru identificarea unităților lingvistice mari
- căutarea lexicală se ocupă de compararea secvenței de parametri de test cu secvențele de parametri de referință stocați
- analiza sintactică realizează testarea consistenței sintactice (funcție de structura gramaticală a limbii) a unui cuvânt ipotetic, relativ la cuvintele deja recunoscute sau presupuse de a fi receptate în continuare
- analiza semantică se referă la testarea înțelesului secvenței de cuvinte ipotetice;
- analiza pragmatică face precizarea celor mai probabile variante de cuvinte receptate ținând seama de natura sarcinii de îndeplinit.

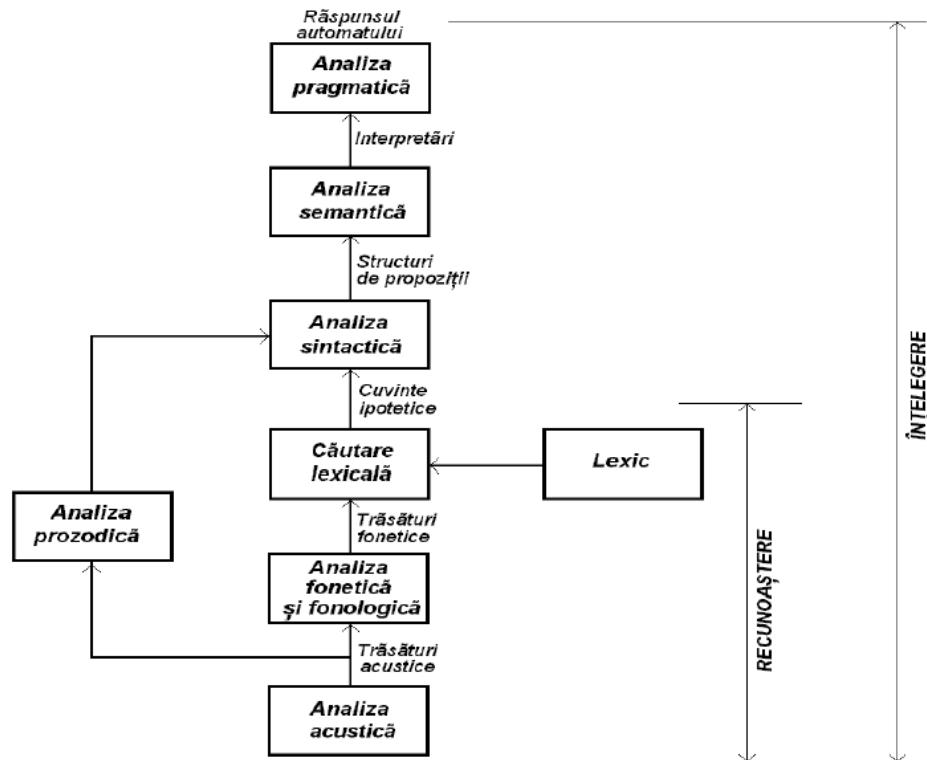


Fig 4.4 Modelul ierarhizat al înțelegerei vorbirii

Ținta tuturor cercetătorilor în acest domeniu este recunoașterea în timp real, cu acuratețe 100% a tuturor cuvintelor rostite inteligibil de orice persoană, independent de dimensiunea vocabularului, zgomot, caracteristicile vorbitorului, accent și condițiile canalului. În ciuda câtorva decenii de cercetare în acest domeniu, o acuratețe mai mare de 90% nu s-a putut obține decât în situația aplicării anumitor constrângeri. În funcție de constrângerile aplicate se pot obține diverse nivele de performanță; de exemplu recunoașterea de cifre, folosind un canal de microfon (vocabular mic, fără zgomot) poate fi mai mare de 99%. Dacă sistemul este antrenat să învețe vocea unui anumit individ, se pot folosi vocabulare mult mai mari, dar acuratețea scade undeva între 90% și 95% pentru sisteme disponibile comercial. Pentru recunoașterea vorbirii folosind vocabulare mari și diversi vorbitori, acuratețea nu crește mai mult de 87%, iar procesarea durează foarte mult timp.

O altă constrângere care se poate aplica sistemelor de recunoaștere a vorbirii este rostirea izolată a cuvintelor. Vorbitorul rosteste cuvintele cu mici pauze între ele. Acesta este cazul cel mai simplu, deoarece limitele între cuvinte sunt detectate foarte usor, iar cuvintele nu sunt puternic coarticulate. Sistemele de recunoaștere a vorbirii continue nu necesită rostirea cuvintelor cu pauză între ele. În acest caz cuvintele pot fi puternic coarticulate, aspect care face recunoașterea mult mai dificilă. Termenul "coarticulare" se referă la fenomenul în care poziția gurii pentru foneme individuale dintr-o rostire de sunete continue încorporează efectele poziției gurii pentru fonemele rostite imediat înainte și după. În mod tipic, aceste sisteme au lexic și sintaxă predefinite. O extensie a acestor sisteme este recunoașterea vorbirii spontane (naturale), pentru care vorbitorul nu este constrâns de dimensiunea vocabularului sau de o gramatică artificială. Această arie este încă în cercetare, deoarece, în acest caz sistemul trebuie să se confrunte cu ezitări în vorbire, propoziții care nu respectă regulile de gramatică, cuvinte din afara vocabularului, etc.

Recunoașterea vorbitorului este întrucâtva complementară recunoașterii vorbirii, deoarece ea are ca scop discriminarea subiecțiilor umani pe baza informației extrase din vocile acestora. Prin urmare, în sarcina recunoașterii vorbitorului nu interesează ce se spune, ci cine spune. Direct derivată din domeniul strict al recunoașterii vorbitorilor este abordarea ce încearcă segmentarea documentelor audio în raport cu un set de vorbitori, încercând un răspuns la întrebarea: cine spune și când spune; ceea ce se spune fiind irelevant.

Există două subclase de aplicații de recunoastere a vorbitorului: verificarea vorbitorului, care își propune să determine dacă un enunț aparține sau nu unui anumit subiect uman, și identificarea vorbitorului, care încearcă să realizeze o corespondență între o voce necunoscută și identitatea unui subiect uman dintr-un set dat. Întrucât în cazul verificării vorbitorului decizia este de tip binar, iar în cazul identificării este de tip N-ar (unde N este numărul de potențiali candidați, deci numărul de vorbitori ale căror identități sunt cunoscute), este de așteptat ca performanțele obținute în procesul de verificare să fie superioare celor obținute în procesul de identificare.

Din punctul de vedere al gradului de control asupra materialului vocal utilizat, sistemele de recunoastere a vorbitorului pot fi dependente de text, sau independente de text. În primele, semnalul vocal disponibil în etapa de antrenare și în cea de testare provine din același text. În această situație, metodele de programare pot fi aplicate pentru a realiza alinierea în timp a fragmentelor de semnal vocal. În cazul sistemelor independente de text, alinierea dinamică nu mai este utilizabilă, însă metodele statistice pot fi folosite cu succes.

## Rezultatele obținute

Am făcut pentru fiecare vocală câte 20 de îregistrări și am scos pitch-urile medii. Sunt afișate în tabele din figura 4.5 și figura 4.6.

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
<b>A</b>	95.12	103.81	108.72	109.68	108.81	105.88	107.66	110.00	112.88	104.56
<b>E</b>	91.6	102.81	97.54	96.7	103.9	96	105.32	103.9	100.37	104.75
<b>I</b>	105.9	109.59	112.54	113.05	104.58	117.87	104	106.57	109.4	106.85
<b>O</b>	95.82	115.02	106.38	94.55	101.65	106.69	96.17	112.55	92.25	101.48
<b>U</b>	117.7	114.66	107.93	102.14	107.81	106.39	108.86	110.17	96.53	107.26

Fig 4.5 Tabelul cu pitch-urile medii obținute

	<b>11</b>	<b>12</b>	<b>13</b>	<b>14</b>	<b>15</b>	<b>16</b>	<b>17</b>	<b>18</b>	<b>19</b>	<b>20</b>
<b>A</b>	105.33	106.25	105.1	113.45	102.13	103.2	97.85	103.1	100	97.4
<b>E</b>	100.9	105.98	102	89.9	97.26	97.11	93.92	92.25	95.45	95.21
<b>I</b>	106.38	104.12	109.46	112.53	105.08	106.36	117.68	109.45	102.65	111.38
<b>O</b>	101.63	102.21	105.32	116.89	90.144	103.97	108.25	108.78	91.52	99.86
<b>U</b>	109.01	118.67	112.38	105.82	102.09	99.26	105.03	98.3	104.85	108.45

Fig 4.6 Tabelul cu pitch-urile medii obținute

Am făcut câte 10 teste pentru fiecare vocală și am afisat pitch-urile în tabelul din figura 4.7.

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
<b>A</b>	98.6	99.7	99.41	99.04	90.7	98.53	90.73	97.81	95.81	86.06
<b>E</b>	97.64	94.80	96.99	99.25	109	109.44	98.43	88.36	92.81	97.52
<b>I</b>	110.7	111.54	101.38	121.58	116.114	118.18	136.114	145.81	133.97	116.23
<b>O</b>	93.06	87.55	89.12	88.51	82.06	80.31	83.85	78.21	81.02	88.78
<b>U</b>	123.92	99.28	107.63	131.19	111.47	114.94	109.28	98.67	99.46	104.72

Fig 4.7 Tabelul cu pitch-urile medii obținute din testare

Am obținut rezultate foarte bune, ținând cont de calitatea microfonul folosit și nivelul de zgomot prezent în București în timpul zilei.

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
<b>A</b>	A	A	A	A	A	E	A	A	A	E
<b>E</b>	E	E	E	A	E	E	E	A	E	E
<b>I</b>	I	I	I	I	I	I	I	I	I	I
<b>O</b>	O	O	O	O	O	O	O	O	O	O
<b>U</b>	U	U	U	U	U	U	U	U	U	U

Fig 4.8 Tabelul cu rezultatele obținute după testare

Din tabel se observă ca 46 din 50 de încercări s-au încheiat prin recunoașterea cu succes a vocaliei. Rata de recunoaștere este de **92%**.

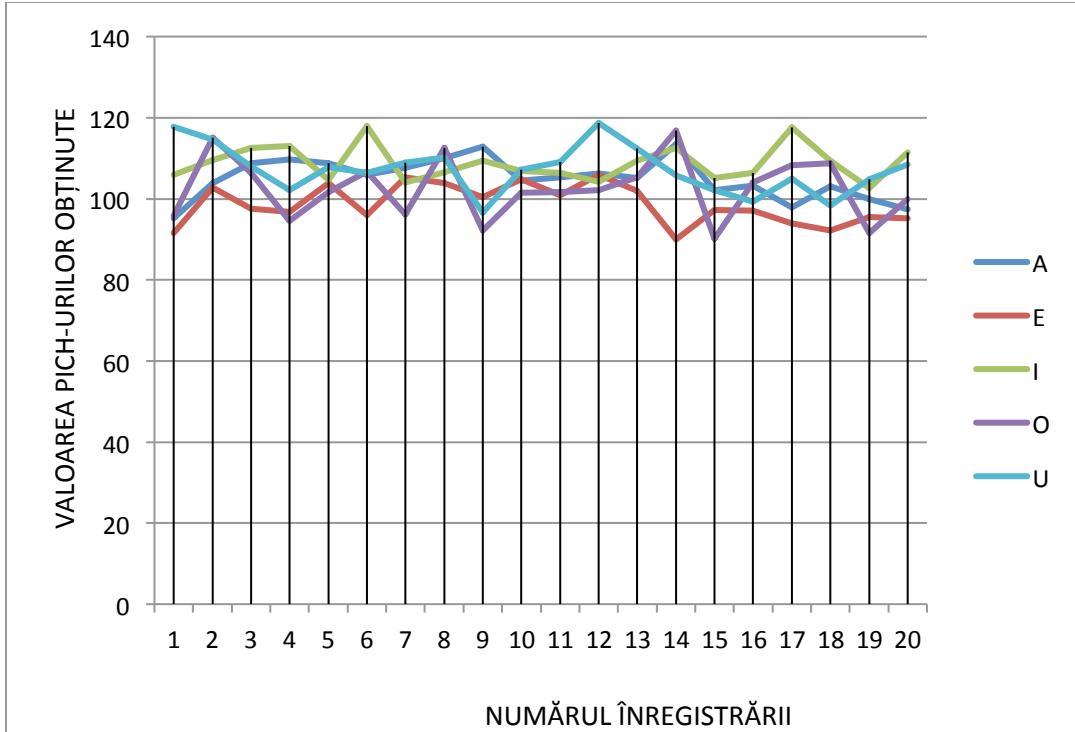


Fig 4.9 Graficul pitch-urilor medii pentru cele 20 de înregistrări

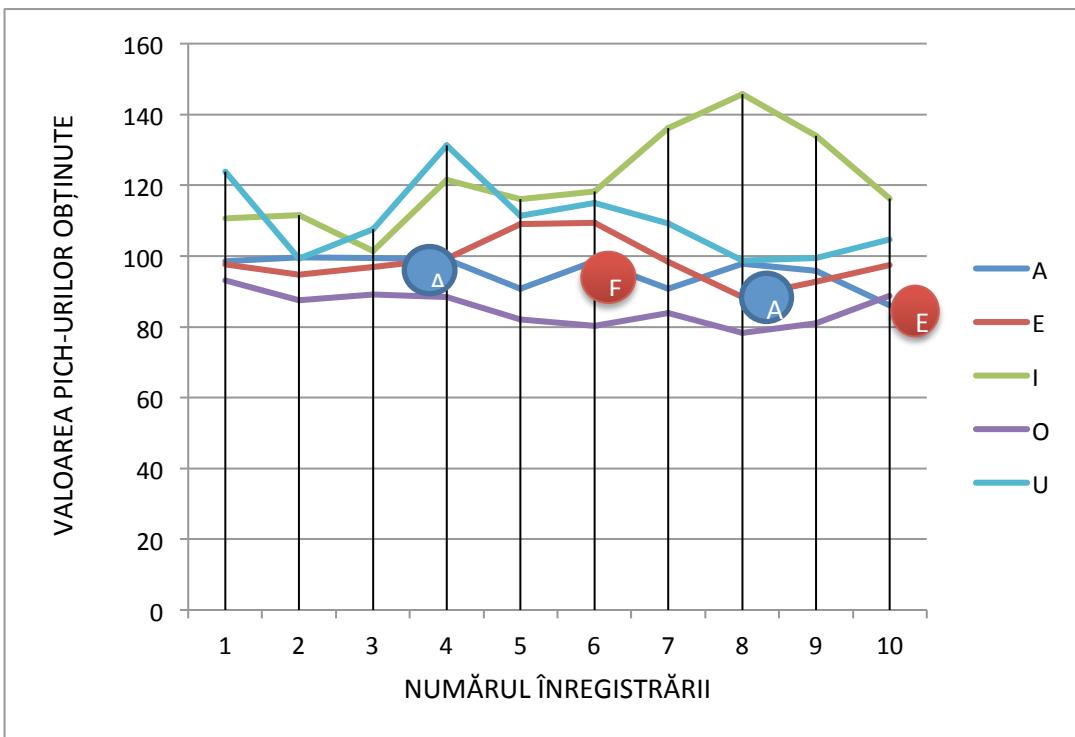


Fig 4.10 Graficul pitch-urilor medii pentru cele 10 teste

## **CAPITOLUL V : IMPLEMENTARE**

---

### **Pachete și clase**

Aplicația este structurată în 5 pachete. Fiecare pachet conține clase de obiecte care împreună modelează o parte bine delimitată din logica aplicației, pachetele fiind concepute să fie independente între ele.

Aplicația conține următoarele pachete:

- vowelrecognition
- vowelrecognition.core
- vowelrecognition.traineddata
- vowelrecognition.util
- vowelrecognition.gui

În continuare voi detalia structura și funcționarea fiecărui pachet.

- **Pachetul vowelrecognition.core:**

Pachetul conține clase care au scopul de a stoca și procesa informația audio cu care lucrează aplicația. Informația audio este stocată ca o listă de sample-uri, iar procesarea de care este capabil acest pachet este formată din determinarea cepstrumului, calcularea pitch-ului și recunoasterea vocaliei.

Pachetul conține următoarele clase :

- ComplexNumber
- SampledAudio
- AudioAnalyzerException
- AudioAnalyzer
- VowelMatcher

**Complex** este o clasă care modelează numerele complexe. Are două câmpuri private de tipul double “re” și “im” reprezentând partea reală și cea imaginată a numărului complex. Metodele clasei sunt cele uzuale pentru operații cu numere complexe : adunare și înmulțire cu numere reale și complexe, operația de conjugare și modul.

Scopul numerelor complexe în această aplicație este de a stoca rezultatul obținut prin aplicarea transformatei Fourier discrete asupra sirului initial. Rezultatul este păstrat într-o listă de numere complexe.

Clasa **SampledAudio** se ocupa cu stocarea informației audio. Aceasta este reprezentată ca vectori de numere complexe. Printre metodele clasei se găsesc mai mulți constructori care primesc vectori de numere în diferite reprezentări și salvează aceste informații într-un `ArrayList<Complex>`.

O metodă importantă din această clasă este `getSamplesInRange(int left, int right)` care returnează un `SampledAudio` care conține sample-urile aflate între cei 2 indici. Scopul acestei clase este de a fi folosită în procesare ca unică reprezentare a informației audio.

**AudioAnalyzerException** este o clasă moștenită din `Exception` și este folosită pentru a raporta cazuri particulare în prelucrarea semnalului audio.

**AudioAnalyzer** este clasa care se ocupă practice de prelucrarea audio digitală a sunetelor. Această conține numai metode statice:

- `fft` : este o metoda care primește ca parametru un obiect din clasa `SampledAudio` și returnează tot un `SampledAudio` care reprezintă transformata Fourier discretă a semnalului audio trimis ca parametru. Este folosit algoritmul Cooley-Tukey.
- `iff` : este o metoda care calculează transformata Fourier discretă inversă.
- `hammingWindow` : este o metodă care primește un semnal audio reprezentat de un obiect `SampledAudio` și returnează același semnal pe care s-a aplicat fereastra Hamming pentru a-l face periodic.

- cepstrum : este o metodă care calculează cepstrumul semnalului audio trimis ca parametru și îl returnează printr-un obiect din clasa SampledAudio. Cepstrumul este calculat folosind apeluri ale metodelor din aceeași clasă, metode descrise mai sus.
- pitch : este o metodă care calculează frecvența fundamentală a unei părți dintr-o vocală trimisă ca parametru. În procesul de calculare se apelează funcția care calculează cepstrumul și apoi se determină poziția pe care se află valoarea maximă. Căutarea nu se face în tot cepstrumul ci numai între pozițiile aflate între 10% din lungimea vectorului care stochează cepstrumul și 75% din aceeași lungime.
- averagePitch : este o metodă care primește ca parametru un obiect SampledAudio (adică tot sunetul înregistrat), și o dimensiune pentru fereastra Hamming pe care vrem să o folosim. Se calculează valoarea medie a pitch-ului din tot fișierul folosind bucăți de lungime dată. Bucățile nu sunt disjuncte (dacă  $B_i$  este bucata cu numărul  $i$  și  $|B_i|$  este lungimea bucății, atunci  $|B_i \cap B_{i+1}| = |B_i| / 2$ ).
- isSilence : este o metodă care decide dacă un obiect din clasa SampledAudio conține doar sunet care reprezintă zgomot de fundal. Este folosită pentru a detecta intervale de sunet util, care se presupune a fi voce umană.

Algoritmul Cooley-Tukey, este numit după J. W. Cooley și John Tukey și este cel mai utilizat algoritm pentru calcularea Transformantei Fourier discretă(DFT). El determină DFT într-o complexitate de  $O(N \log N)$ , împărțind sirul initial în siruri mai mici de lungime puteri ale lui 2.

Acest algoritm reduce substanțial volumul de calcul al transformantei Fourier, motiv pentru care analiza spectrală se utilizează și în cazul aplicațiilor reale, unde timpul de calcul se impune să fie cât mai mic.

În paralel cu algoritmii rapizi de calcul ai transformantei Fourier s-au dezvoltat și structuri hard de calcul care utilizează DSP-uri (Digital Signal Processor) ce dispun de instrumente de calcul paralel și o unitate de calcul în virgulă mobilă, cu posibilități de a realiza calcule în simplă, dublă precizie sau precizie extinsă.

**VowelMatcher**, această clasă are o singură metodă și anume “match”, metoda care primește un obiect din clasa SampledAudio și un String cu numele utilizatorului care a înregistrat semnalul audio și întoarce un String care reprezintă vocala cu cea mai mare probabilitate să fie aceeași cu cea care a fost înregistrată. În continuare voi detalia algoritmul de determinare a vocaliei.

Prima fază este aceea de a calcula pitch-ul mediu pentru fișierul trimis ca parametru pentru toate cele 5 dimensiuni ale ferestrei hamming folosite în aplicație : 64, 128, 256, 512, 1024. Apoi pe fiecare pitch astfel determinat aplic o transformare liniară care duce valoarea în intervalul [1,2].

Am definit distanța între 2 vectori A și B de pitchuri ca fiind:  $\sum_{i=0}^4 ((A[i] - B[i])^2)$ .

A doua fază este de a extrage din baza de date toate datele despre toate vocalele pentru utilizatorul care a înregistrat semnalul audio curent și de a construi 5 vectori de pitch-uri (cate unu pentru fiecare vocală), vectori construți în același mod cum a fost construit și cel din prima fază.

Faza a treia, și ultima constă în calcularea distanțelor dintre vectorul calculat în prima fază și fiecare vector calculat la faza a doua. Vocala a carei vector are cea mai mică distanță este cea care are cea mai mare probabilitate să fie cea înregistrată și numele ei este returnat ca un String.

Scopul principal al acestui pachet este acela de a calcula pitch-ul mediu dintr-un fișier audio.

- **Pachetul vowelmatcher.traineddata:**

Pachetul conține clase care au scopul de a modela și stoca informația în urma procesarii.

Informația este stocată structurat: baza de date conține informații despre utilizatori, fiecare utilizator conține informații despre cele 5 cadre, fiecare cadru conține informații despre cele 5 vocale, iar fiecare vocală conține informații despre pitch-urile medii calculate din diferite înregistrări audio.

Pachetul conține următoarele clase :

- VowelData
- FrameData
- UserData
- TrainedData
- TrainedDataHandler

**VowelData** este o clasă care modelează cunoștiințele despre o vocală. Cunoștiințele sunt reprezentate de mai multe valori ale pitch-ului mediu calculate din diferite înregistrări ale aceleiași vocale făcute de același utilizator. Clasa conține metode pentru a adăuga sau a șterge valoarea unui pitch mediu și o metodă care calculează din nou media tuturor pitchurilor medii stocate.

Clasa **FrameDate** este clasa care modelează cunoștiințele despre toate vocalele procesate cu o fereastră hamming de o anumită dimensiune. Clasa conține o metodă “getVowelData” care întoarce o referință la unul dintre obiectele VowelData pe care le conține.

Metoda “clearData” golește `HashMap<String, VowelData>()`-ul `vowels`; o altă metodă este “getSize” care returnează dimensiunea ferestrei folosite în procesare.

**UserData**, această clasă modelează cunoștiințele despre toate vocalele procesate cu toate cele 5 dimensiuni ale ferestrei pentru un utilizator. Datele sunt stocate ca un `HashMap<Integer, FrameData>`.

Metodele conținute au aceeași funcționalitate cu cele conținute de `FrameData`.

**TrainedData** este o clasă care modelează cunoștiințele despre toți utilizatorii. Datale sunt stocate într-un `HashMap<String, UserData>` care mapează numele utilizatorului cu un obiect `UserData`. Conține metode pentru adăugarea și ștergerea utilizatorilor, pentru obținerea informațiilor despre un utilizator anume și o metodă `getUsernames()` care întoarce toate numele utilizatorilor existenți în sistem.

**TrainedDataHandler**, scopul acestei clase este de a uni logic funcționalitatea pachetului database cu metoda care calculează pitch-ul mediu dintr-o înregistrare audio. Metoda cea mai importantă din această clasă este “train”, metoda care primește ca parametrii un obiect din clasa `SampledAudio` și 2 obiecte din clasa `String`, unul reprezentând numele vocii iar celălalt numele utilizatorului care a înregistrat acea vocală.

Din primul parametru se va calcula pitch-ul mediu pentru cele 5 dimensiuni ale ferestrei hamming folosite în aplicație, apoi adaugă toate aceste valori în baza de date a utilizatorului specificat ca al 3-lea parametru.

De asemenea clasa mai contine metode pentru a sterge informații din baza de date de pe orice nivel, și o metoda care întoarce o referință la obiectul din clasa `Database` asupra căruia executa modificările.

- **Pachetul vowelrecognizer.util:**

Pachetul vpower.engine contine clase care modeleaza functionalitatea aplicatiei :

- AudioRecorder
- SamplesProvider
- AudioSamplesProvider
- SpeechSamplesProvider

**AudioRecorder** este clasa care face înregistrarea audio propriu-zisă. Se folosește de API-ul audio pus la dispoziție de librăriile standard java. Clasa are două metode publice care controlează înregistrarea: start() și stop().

Semnalul audio înregistrat este stocat într-un buffer privat al clasei. Înregistrarea se recuperează cu ajutorul metodei getSampledAudio care întoarce un obiect SampledAudio asociat înregistrării curente.

**SamplesProvider** este o clasa abstractă care are ca scop implementarea funcționalității comune dintre AudioSamplesProvider și SpeechSamplesProvider. Ea are doar două metode publice, startThread() care porneste un fir de execuție intern ce procesează semnalul audio, și o metodă getSamples() care întoarce o referință la un obiect din clasa SampledAudio, obiect ce conține semnalul audio procesat.

**AudioSamplesProvider** este o clasa moștenită din SamplesProvider și se folosește de AudioRecorder pentru a pune la dispozitie referințe către un obiect din clasa SampledAudio ce conține semnal audio cu lungimea de o secundă.

**SpeechSamplesProvider** este o clasa moștenită din SamplesProvider și se folosește de AudioSamplesProvider pentru a pune la dispozitie referințe către un obiect din clasa SampledAudio ce conține doar semnal audio util, fără intervale de liniste.

Pachetele vowelrecognition.gui si vowelrecognition contin clase care implementeaza interfata grafica a aplicatiei respectiv o clasa Main cu o singura metoda statica “main” care porneste intreaga aplicatie.

## Paralelism

În această aplicație este folosit paralelisumul în trei locuri: la procesarea fișierelor audio, după ce se dă stop în aplicație și la etapa de recunoaștere.

La etapa de procesare a fișierelor audio există un thread pentru fiecare cadru. Fiecare thread calculază independent pitch-ul pentru câte o fereastră hamming.

Prelucrarea datelor durează mai mult de o secundă în care utilizatorul realizează ca interfața aplicației rămâne blocată. În acest caz este nevoie de un nou thread care să se ocupe interfață iar celelalte să continue cu prelucrarea datelor.

La etapa de procesare se va folosi un thread pentru fiecare vocală, el v-a trebui să calculeze suma erorilor.

## **CONCLUZII**

---

Rezultatele obținute demonstrează faptul că strategia de antrenare abordată funcționează și că dă rezultate din ce în ce mai bune pe măsură ce sunt efectuați mai mulți pași de antrenare/testare.

Pentru acest proiect s-au făcut 100 de înregistrări în baza de date în contul unui singur user și 50 de teste. Aplicația având o rata de recunoaștere de 92%.

O problemă ar fi cu diferențierea între A și E, rata de recunoaștere fiind de doar 80%. Însă în schimb rata de recunoaștere între celelalte vocale și A sau E este de 100%, pentru 20 de înregistrări pentru fiecare vocală și 10 teste.

Varianta neparalelezată a aplicației poate ocupa o mare parte din procesor dacă înregistrarea este foarte mare și poate chiar bloca aplicația. Pe procesor se calculează operații matematice intensive.

Rezultate bune se obțin de la 7 înregistrări pentru fiecare vocală.

Utilitatea aplicației este, din perspectiva mea, indiscutabilă. Variante ale aplicației, integrate într-un proiect complex, facilitează comunicarea dintre om și mașină.

Limitații lucrării mele sunt date de experiența insuficientă a autorului, de timpul în care a fost concepută și pusă în aplicare, precum și resurselor limitate.

Cred cu sărăcie că aplicația merită îmbunătățită în viitor.

## **BIBLIOGRAFIE**

---

1. Unde, Berkeley, volumul 3, 1983
2. Curs Prelucrarea Semnalelor, Prof. Dr. Ing. Șerban Petrescu
3. Burileanu, C., Tehnologia Vorbirii, note de curs, 2003.
4. The HTK Book, Steve Young, Gunnar Evermann, Mark Gales, Thomas Hain, Dan Kershaw
5. Gavat, I., Interfețe om-mașină, note de curs, 2009.
6. Classical and Quantum, Predrag Cvitanovic , Roberto Artuso, Per Dahlqvist, Ronnie Mainieri, Gregor Tanner, Gabor Vattay – Niall Whelan – Andreas Wirzba, printată pe 1 septembrie 2003
7. Distant Speech Recognition, Matthias Wolfel și John McDonough
8. <http://en.wikipedia.org/wiki/Dither>
9. [http://en.wikipedia.org/wiki/Fourier\\_transform](http://en.wikipedia.org/wiki/Fourier_transform)
10. [http://en.wikipedia.org/wiki/Pitch\\_\(music\)](http://en.wikipedia.org/wiki/Pitch_(music))
11. <http://en.wikipedia.org/wiki/Cepstrum>
12. Petrea, C.S., Buzo, A., Cucu, H. Pașca, M. Burileanu, C., Speech Recognition Experimental Results for Romanian Language, ECIT2010, 2010.
13. Huang, X.D., Acero, A., Hon, H.-W., Spoken Language Processing. A guide to Theory, Algorithm and System Development, Editura Prentice Hall, 2001

## **ANEXĂ**

---

Aceasta anexa contine codul sursa al backend-ului aplicatiei

```

package vowelrecognition.core;

public class AudioAnalyzer {

    public static SampledAudio fft(SampledAudio x)
        throws AudioAnalyzerException {
            int N = x.size();

            if (N == 1) {
                SampledAudio tmp = new SampledAudio();
                tmp.add(x.get(0));
                return tmp;
            }

            if (N % 2 != 0) {
                throw new AudioAnalyzerException(
                    "SampledAudio's size must be a power of 2");
            }

            SampledAudio even = new SampledAudio();
            for (int k = 0; k < N / 2; k++) {
                even.add(x.get(2 * k));
            }
            SampledAudio q = fft(even);

            SampledAudio odd = new SampledAudio();
            for (int k = 0; k < N / 2; k++) {
                odd.add(x.get(2 * k + 1));
            }
            SampledAudio r = fft(odd);

            ComplexNumber[] y = new ComplexNumber[N];
            for (int k = 0, k < N / 2; k++) {
                double kth = -2 * k * Math.PI / N;
                ComplexNumber wk = new ComplexNumber(Math.cos(kth), Math.sin(kth));
                y[k] = q.get(k).plus(wk.times(r.get(k)));
                y[k + N / 2] = q.get(k).minus(wk.times(r.get(k)));
            }
            return new SampledAudio(y);
        }

    public static SampledAudio ifft(SampledAudio x)
        throws AudioAnalyzerException {
            int N = x.size();
            SampledAudio y = new SampledAudio();

            for (int i = 0; i < N; i++) {
                y.add(x.get(i).conjugate());
            }

```

```

y = fft(y);

for (int i = 0; i < N; i++) {
    y.set(i, y.get(i).conjugate());
}

for (int i = 0; i < N; i++) {
    y.set(i, y.get(i).times(1.0 / N));
}

return y;
}

public static SampledAudio hammingWindow(SampledAudio samples) {
    SampledAudio tmp = new SampledAudio();
    for (int i = 0; i < samples.size(); ++i) {
        tmp.add(new ComplexNumber(samples.get(i).abs()
            * (0.54 - 0.46 * Math.cos(2 * Math.PI * i
            / (samples.size() - 1))), 0));
    }
    return tmp;
}

public static SampledAudio cepstrum(SampledAudio samples)
    throws AudioAnalyzerException {
    SampledAudio tmp = new SampledAudio(samples);
    tmp = AudioAnalyzer.hammingWindow(tmp);
    tmp = AudioAnalyzer.fft(tmp);

    SampledAudio tmp2 = new SampledAudio();
    for (ComplexNumber x : tmp) {
        tmp2.add(new ComplexNumber(Math.log(x.abs()), 0));
    }

    tmp = AudioAnalyzer.ifft(tmp2);

    return tmp;
}

public static int pitch(SampledAudio samples) throws AudioAnalyzerException {
    SampledAudio tmp = AudioAnalyzer.cepstrum(samples);
    int left = (int) (tmp.size() * 0.1);
    int right = (int) (tmp.size() * 0.75);
    int index = -1;
    double max = -1;

    for (int i = left; i <= right; ++i) {
        if (max < tmp.get(i).abs()) {
            max = tmp.get(i).abs();
            index = i;
        }
    }

    return index;
}

```

```

public static double averagePitch(SampledAudio samples, int windowSize)
    throws AudioAnalyzerException {

    int N = windowSize, S = 0;

    while (N != 1) {
        if (N % 2 == 1)
            throw new AudioAnalyzerException(
                "windowSize must be a power of 2");
        N >>= 1;
    }

    for (int i = 0; i + windowSize < samples.size(); i += windowSize >> 1, ++N) {
        S += AudioAnalyzer.pitch(samples.getSamplesInRange(i, i
            + windowSize));
    }

    return (double) S / N;
}

public static boolean isSilence(SampledAudio samples) {
    if (samples.size() == 0)
        return false;

    double silenceThreshold = 1000, maxim = samples.get(0).real();

    for (int i = 0; i < samples.size(); ++i) {
        if (samples.get(i).real() > maxim) {
            maxim = samples.get(i).real();
        }
    }

    return maxim < silenceThreshold;
}
}

public class AudioAnalyzerException extends Exception {
    private static final long serialVersionUID = -6787499657235357795L;

    public AudioAnalyzerException(String message) {
        super(message);
    }
}

public class ComplexNumber {
    private final double re;
    private double im;

    public ComplexNumber(double x) {
        re = x;
    }

    public ComplexNumber(double re, double im) {
        this(re);
        this.im = im;
    }
}

```

```

}

public ComplexNumber(ComplexNumber c) {
    this(c.re, c.im);
}

public double real() {
    return this.re;
}

public double imaginary() {
    return this.im;
}

public ComplexNumber plus(ComplexNumber c) {
    return new ComplexNumber(this.re + c.re, this.im + c.im);
}

public ComplexNumber plus(double x) {
    return this.plus(new ComplexNumber(x));
}

public ComplexNumber minus(ComplexNumber c) {
    return this.plus(new ComplexNumber(-c.re, -c.im));
}

public ComplexNumber minus(double x) {
    return this.minus(new ComplexNumber(x));
}

public ComplexNumber times(ComplexNumber c) {
    return new ComplexNumber(this.re * c.re - this.im * c.im, this.re * c.im
        + this.im * c.re);
}

public ComplexNumber times(double x) {
    return this.times(new ComplexNumber(x));
}

public ComplexNumber conjugate() {
    return new ComplexNumber(re, -im);
}

public double abs() {
    return Math.sqrt(re * re + im * im);
}

public static ComplexNumber[] getComplexArray(double[] v) {
    ComplexNumber[] tmp = new ComplexNumber[v.length];
    for (int i = 0; i < v.length; ++i) {
        tmp[i] = new ComplexNumber(v[i]);
    }
    return tmp;
}

public static List<ComplexNumber> getComplexArrayList(ComplexNumber[] v) {
}

```

```

List<ComplexNumber> tmp = new ArrayList<ComplexNumber>();
for (ComplexNumber c : v) {
    tmp.add(new ComplexNumber(c));
}
return tmp;
}

public static List<ComplexNumber> getComplexArrayList(double[] v) {
    return ComplexNumber.getComplexArrayList(ComplexNumber.getComplexArray(v));
}
}

public class SampledAudio extends ArrayList<ComplexNumber> {
    private static final long serialVersionUID = 414911875972606451L;

    public SampledAudio() {
    }

    public SampledAudio(AudioInputStream ais)
        throws UnsupportedAudioFileException, IOException {
        AudioFormat af = ais.getFormat();

        if (af.getChannels() != 1) {
            throw new UnsupportedAudioFileException("WavFile is not mono");
        }
        if (af.getSampleSizeInBits() != 16) {
            throw new UnsupportedAudioFileException(
                "WavFile must have 16bit sample size");
        }

        while (ais.available() >= 2) {
            byte[] buff = new byte[2];
            ais.read(buff);
            int value = ((buff[1] & 0xff) << 8) + (buff[0] & 0xff);
            if ((value & (1 << 15)) != 0) {
                value -= (1 << 15);
                value *= -1;
            }
            this.add(new ComplexNumber(value));
        }
    }

    public SampledAudio(List<ComplexNumber> samples) {
        this.addAll(samples);
    }

    public SampledAudio(ComplexNumber[] samples) {
        this.addAll(ComplexNumber.getComplexArrayList(samples));
    }

    public SampledAudio(double[] samples) {
        this.addAll(ComplexNumber.getComplexArrayList(samples));
    }

    public SampledAudio getSamplesInRange(int left, int right) {
        SampledAudio tmp = new SampledAudio();

```

```

        for (int i = left; i < right && i < this.size(); ++i) {
            tmp.add(this.get(i));
        }
        return tmp;
    }

}

public class VowelMatcher {
    private final TrainedData database;
    private final String[] vowels = new String[] { "A", "E", "I", "O", "U" };

    public VowelMatcher(TrainedData database) {
        this.database = database;
    }

    public String match(SampledAudio samples, String username)
        throws AudioAnalyzerException {
        class Counter {
            int count = 0;
            double[] pitch = new double[5];

            synchronized void increment() {
                count++;
            }

            synchronized int getCount() {
                return count;
            }

            synchronized void set(int index, double pitch_value) {
                pitch[index] = pitch_value;
            }

            synchronized double[] getPitch() {
                return pitch;
            }
        }
    }

    class Matcher extends Thread {
        SampledAudio samples;
        int window_size;
        Counter cnt;

        Matcher(Counter cnt, SampledAudio samples, int window_size) {
            this.cnt = cnt;
            this.samples = samples;
            this.window_size = window_size;
        }

        @Override
        public void run() {
            double a = window_size * 0.1, b = window_size * 0.75, temp;
            int i = 0, ww = window_size;
            while (ww != 1) {
                ++i;
                ww >= 1;

```

```

        }
        i -= 6;
        try {
            temp = AudioAnalyzer.averagePitch(samples, window_size);
            temp = (temp - a) / (b - a) + 1;
            cnt.set(i, temp);
        } catch (Exception ex) {
        }
        cnt.increment();
    }
}

Counter cnt = new Counter();
for (int i = 0; i < 5; ++i) {
    int windowSize = 1 << (i + 6);
    (new Matcher(cnt, samples, windowSize)).start();
}

while (cnt.getCount() != 5) {
    try {
        Thread.sleep(50);
    } catch (InterruptedException ex) {
    }
}

double[] diff = new double[vowels.length];
double[] pitch = cnt.getPitch();

for (int i = 0; i < vowels.length; ++i) {
    for (int j = 0; j < 5; ++j) {
        int windowSize = 1 << (j + 6);
        double a = windowSize * 0.1, b = windowSize * 0.75;
        double tmp = database.getUserData(username)

        .getFrameData(windowSize).getVowelData(vowels[i])
            .getAverage();
        tmp = (tmp - a) / (b - a) + 1;
        tmp -= pitch[j];
        tmp *= tmp;
        diff[i] += tmp;
    }
}

double min = -1;
int index = -1;
for (int i = 0; i < 5; ++i) {
    if (min == -1 || diff[i] < min) {
        min = diff[i];
        index = i;
    }
}

return vowels[index];
}
}

```

```

package vowelrecognition.traineddata;

public class FrameData implements Serializable {
    private static final long serialVersionUID = -4236663116953180602L;
    private final int size;
    private final Map<String, VowelData> vowels;

    public FrameData(int size) {
        this.size = size;
        vowels = new HashMap<String, VowelData>();
        vowels.put("A", new VowelData("A"));
        vowels.put("E", new VowelData("E"));
        vowels.put("I", new VowelData("I"));
        vowels.put("O", new VowelData("O"));
        vowels.put("U", new VowelData("U"));
    }

    public VowelData getVowelData(String name) {
        return vowels.get(name);
    }

    public int getSize() {
        return size;
    }

    public void clearData() {
        vowels.remove("A");
        vowels.remove("E");
        vowels.remove("I");
        vowels.remove("O");
        vowels.remove("U");
        vowels.put("A", new VowelData("A"));
        vowels.put("E", new VowelData("E"));
        vowels.put("I", new VowelData("I"));
        vowels.put("O", new VowelData("O"));
        vowels.put("U", new VowelData("U"));
    }
}

public class TrainedData implements Serializable {
    private static final long serialVersionUID = 4488896584494035470L;
    private final Map<String, UserData> users;

    public TrainedData() {
        users = new HashMap<String, UserData>();
    }

    public void addUser(String name) {
        users.put(name, new UserData(name));
    }

    public void removeUser(String name) {
        users.remove(name);
    }

    public UserData getUserData(String name) {

```

```

        return users.get(name);
    }

    public void clearData() {
        users.clear();
    }

    public String[] getUsernames() {
        String[] tmp = new String[users.keySet().size()];
        int poz = 0;
        for (String name : users.keySet()) {
            tmp[poz++] = name;
        }

        return tmp;
    }
}

public class TrainedDataHandler {
    private final TrainedData database;

    public TrainedDataHandler(TrainedData database) {
        this.database = database;
    }

    public void addUser(String name) {
        database.addUser(name);
    }

    public void removeUser(String name) {
        database.removeUser(name);
    }

    public void train(SampledAudio samples, String vowel, String username)
        throws AudioAnalyzerException {
        class WavAdder extends Thread {
            VowelData vowel;
            int window_size;
            SampledAudio samples;

            WavAdder(VowelData vowel, SampledAudio samples, int window_size) {
                this.vowel = vowel;
                this.window_size = window_size;
                this.samples = samples;
            }

            @Override
            public void run() {
                double pitch = 0;
                try {
                    pitch = AudioAnalyzer.averagePitch(samples, window_size);
                } catch (AudioAnalyzerException ex) {
                    System.err
                        .println("Error while computing the average
pitch :" + ex.getMessage());
                }
            }
        }
    }
}

```

```

        + ex);
    }
    vowel.add(pitch);
    // System.out.println("Window size " + window_size + " -> "
    // + pitch + " pitch");
}
}

UserData user = database.getUserData(username);

for (int i = 6; i < 11; ++i) {
    int windowSize = 1 << i;
    (new WavAdder(user.getFrameData(windowSize).getVowelData(vowel),
                  samples, windowSize)).start();
}
}

public void clearDatabase() {
    database.clearData();
}

public void clearUserData(String name) {
    database.getUserData(name).clearData();
}

public void clearFrameData(String username, int frameSize) {
    database.getUserData(username).getFrameData(frameSize).clearData();
}

public void clearVowelData(String username, int frameSize, String vowelname) {
    database.getUserData(username).getFrameData(frameSize)
        .getVowelData(vowelname).clearData();
}

public TrainedData getDatabase() {
    return database;
}
}

public class UserData implements Serializable {
    private static final long serialVersionUID = 3967168709865741376L;
    private final String name;
    private final Map<Integer, FrameData> frames;

    public UserData(String name) {
        this.name = name;
        frames = new HashMap<Integer, FrameData>();
        frames.put(64, new FrameData(64));
        frames.put(128, new FrameData(128));
        frames.put(256, new FrameData(256));
        frames.put(512, new FrameData(512));
        frames.put(1024, new FrameData(1024));
    }

    public FrameData getFrameData(int size) {
        return frames.get(size);
    }
}
```

```

    }

    public String getName() {
        return name;
    }

    public void clearData() {
        for (int i = 6; i < 11; ++i) {
            int windowSize = 1 << i;
            frames.remove(windowSize);
            frames.put(windowSize, new FrameData(windowSize));
        }
    }
}

public class VowelData implements Serializable {
    private static final long serialVersionUID = 6293024400132417137L;
    private double S;
    private final String name;
    private final ArrayList<Double> pitch;

    public VowelData(String name) {
        this.name = name;
        pitch = new ArrayList<Double>();
        S = 0;
    }

    public synchronized void add(double x) {
        pitch.add(x);
        S += x;
    }

    public synchronized void remove(double x) {
        for (int i = 0; i < pitch.size(); ++i) {
            if (pitch.get(i) == x) {
                S -= x;
                pitch.remove(i);
                break;
            }
        }
    }

    public synchronized double getAverage() {
        return S / pitch.size();
    }

    public String getName() {
        return name;
    }

    public synchronized void clearData() {
        pitch.clear();
        S = 0;
    }
}

```

```
package vowelrecognition.util;
```

```
public class AudioRecorder {
    private boolean recording;
    private final AudioFormat format;
    ByteArrayOutputStream baos;

    public AudioRecorder() {
        format = new AudioFormat(44100, 16, 1, true, false);
    }

    public void start() throws LineUnavailableException {
        startRecording();
        record();
    }

    public synchronized void stop() {
        recording = false;
    }

    private synchronized boolean isRecording() {
        return recording;
    }

    private synchronized void startRecording() {
        recording = true;
    }

    private void record() throws LineUnavailableException {
        DataLine.Info info = new DataLine.Info(TargetDataLine.class, format);

        final TargetDataLine target_data_line = (TargetDataLine) AudioSystem
            .getLine(info);
        target_data_line.open(format);
        target_data_line.start();

        Runnable runner = new Runnable() {
            byte buffer[] = new byte[1024];

            @Override
            public void run() {
                baos = new ByteArrayOutputStream();
                recording = true;
                try {
                    while (isRecording()) {
                        int count = target_data_line.read(buffer, 0,
                            buffer.length);
                        if (count > 0) {
                            baos.write(buffer, 0, count);
                        }
                    }
                    baos.close();
                } catch (IOException ex) {
                    System.err.println("I/O problems: " + ex);
                    System.exit(-1);
                }
            }
        };
        runner.run();
    }
}
```

```

        }
    }
};

Thread captureThread = new Thread(runner);
captureThread.start();
}

public SampledAudio getSamples() throws UnsupportedAudioFileException,
IOException {
    ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
    AudioInputStream ais = new AudioInputStream(bais, format,
        baos.toByteArray().length / format.getFrameSize());
    return new SampledAudio(ais);
}
}

public class AudioSamplesProvider extends SamplesProvider {

    @Override
    protected void action() {
        AudioRecorder ar = new AudioRecorder();
        try {
            ar.start();
            Thread.sleep(1000);
            ar.stop();
            setSamples(ar.getSamples());
        } catch (Exception e) {
            throw new RuntimeException("AudioSamplesProvider is dead", e);
        }
    }
}

import vowelrecognition.core.SampledAudio;

public abstract class SamplesProvider {
    private SampledAudio samples;
    private boolean stopFlag;

    public void startThread() {
        Runnable runnable = new Runnable() {
            @Override
            public void run() {
                while (shouldStop() == false) {
                    action();
                }
            }
        };
        (new Thread(runnable)).start();
    }

    public synchronized void stopThread() {
        stopFlag = true;
    }
}

```

```

public SampledAudio getSamples() {

    SampledAudio samples = null;
    while (true) {
        samples = privateGetSamples();
        if (samples != null || shouldStop())
            break;
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {
            System.err.println("Could not put SamplesProvider to sleep. "
                + "Busy waiting is performed.");
        }
    }
    return samples;
}

private synchronized SampledAudio privateGetSamples() {
    SampledAudio toReturn = samples;
    this.samples = null;
    return toReturn;
}

protected synchronized void setSamples(SampledAudio samples) {
    this.samples = samples;
}

private synchronized boolean shouldStop() {
    return stopFlag;
}

protected abstract void action();
}

public class SpeechSamplesProvider extends SamplesProvider {
    private final AudioSamplesProvider audioProvider;
    private SampledAudio currentSamples;
    private int left, right;

    public SpeechSamplesProvider() {
        audioProvider = new AudioSamplesProvider();
        currentSamples = null;
    }

    @Override
    public void startThread() {
        audioProvider.startThread();
        super.startThread();
    }

    @Override
    public void stopThread() {
        audioProvider.stopThread();
        super.stopThread();
    }
}

```

```

}

@Override
protected void action() {
    SampledAudio samplesToReturn = new SampledAudio();
    SampledAudio range;

    // skip over silence
    while (AudioAnalyzer.isSilence((range = getNextRangeOfSamples())))
        ;

    // save all speech until silence occurs
    samplesToReturn.addAll(range);
    while (AudioAnalyzer.isSilence((range = getNextRangeOfSamples())) == false) {
        samplesToReturn.addAll(range);
        if (range.size() == 0)
            break;
    }

    setSamples(samplesToReturn);
}

private SampledAudio getNextRangeOfSamples() {
    SampledAudio range;

    if (currentSamples == null) {
        currentSamples = audioProvider.getSamples();
        left = 0;
    }

    right = left + 1024;
    if (right > currentSamples.size()) {
        SampledAudio newSamples = audioProvider.getSamples();

        if (newSamples == null)
            return new SampledAudio();

        currentSamples.addAll(newSamples);
        currentSamples = currentSamples.getSamplesInRange(left,
                currentSamples.size());
        left = 0;
        right = 1024;
    }

    range = currentSamples.getSamplesInRange(left, right);
    left = right;

    return range;
}
}

```