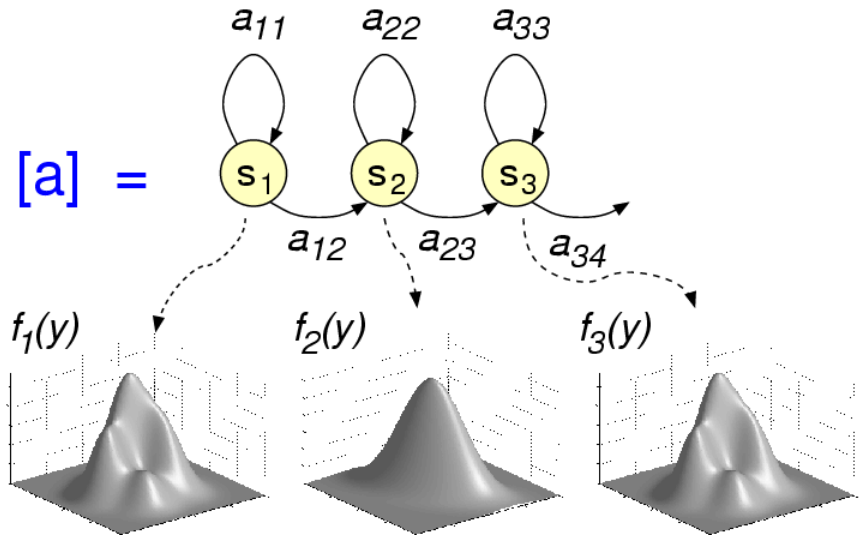
Ecole d'ingénieurs du canton de Vaud

Projet de diplôme

Reconnaissance vocale par chaîne de Markov cachée



Décembre 2005

Sébastien Dupertuis
TT-2005

Professeur responsable : Hervé Dedieu
Expert : Dr. Jean Hennebert

Cahier des charges

- Comprendre les techniques fonctionnelles utilisées dans chaque bloc d'un reconnaisseur évolué utilisant les Chaînes de Markov cachées.

- Implémenter en langage de haut-niveau (Matlab) le prétraitement (rehaussement, LPC, coefficient cepstraux).

- Implémenter en langage de haut-niveau (Matlab), la quantification vectorielle (k-means, LBG, etc.).

- Implémenter en langage de haut-niveau le reconnaisseur (algorithme de Viterbi).

- Implémenter en langage de haut-niveau les procédures d'entraînement des états Markoviens (réseau de neurones).

- Réaliser un cas test à l'aide d'une base de donnée déjà segmentée contenant des phonèmes en Français ou en Anglais.

- Une fois les étapes validées en langage de haut-niveau, passage à la programmation du processeur cible.

- On veillera à utiliser les validations en langage de haut-niveau pour cross-valider la programmation du processeur spécialisé.

Table des matières

1 Résumé

Le but de ce travail de diplôme est de réaliser une reconnaissance de la parole basée sur les chaînes de Markov cachées (HMM). La réalisation de cette reconnaissance va se dérouler en trois étapes distinctes :

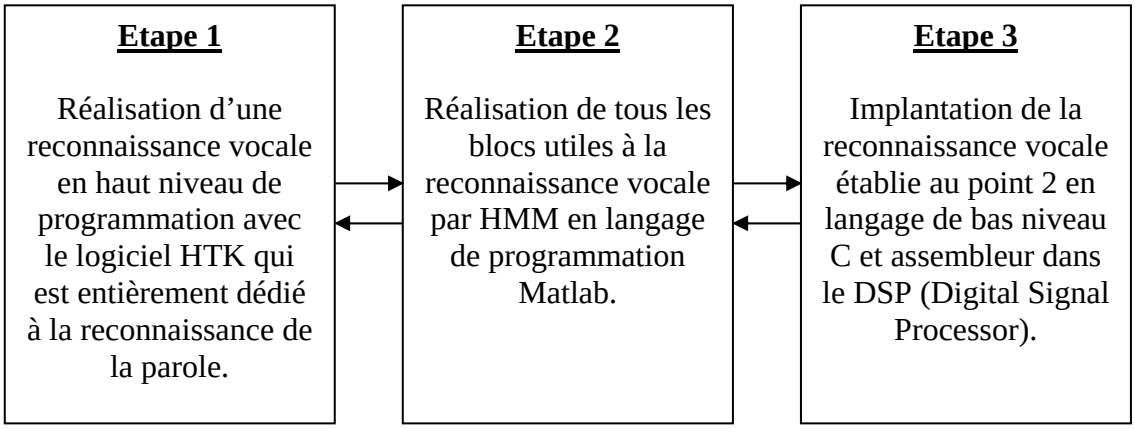


Figure 1 – Etude des différentes étapes de conception de la reconnaissance de la parole par HMM

La première phase permet de réaliser une reconnaissance vocale avec un grand niveau d'abstraction. En effet, il est possible avec HTK et seulement 20 lignes de code, de réaliser une reconnaissance de la parole basée sur les HMM. L'apprentissage de l'utilisation de ce logiciel a été réalisé lors du projet semestre en juin 2005.
La finalité de cette étape est de définir (lors de la phase d'entraînement) des modèles de chaînes de Markov robustes qui pourront être directement appliqués aux étapes 2 et 3.

La deuxième étape consiste à entrer dans la complexité, et à réaliser les divers blocs nécessaires au bon fonctionnement de la reconnaissance vocale, soit :

- l'extraction des paramètres cepstraux (voir au chapitre 5)
- la quantification vectorielle (voir au chapitre 6)
- les chaînes de Markov cachées (voir au chapitre 7, 8 et 9)
- l'algorithme de Viterbi (voir au chapitre 10)

Finalement, la troisième étape concerne l'implémentation de cette reconnaissance sur un DSP. Dans cette dernière étape, la complexité est maximale. La structure de la reconnaissance ne change pas de la phase 2, mais le contenu du code doit être plus strict et correspond à du code C et assembleur, ce qui correspond au niveau d'abstraction le plus faible.

Sur la figure 1, les flèches reliant les différentes étapes représentent la « cross validation ». En effet, cette technique de validation permet de vérifier le bon fonctionnement de chacun des blocs d'une étape par ceux réalisés lors de l'étape précédente (bien-sûr en partant de l'hypothèse que HTK fournit des données correctes).

2 Introduction

Le but de ce travail de diplôme est de réaliser une reconnaissance vocale basée sur l'utilisation des chaînes de Markov cachées.

Un schéma de modélisation bien précis de reconnaissance de la parole a été conçu. Ensuite, il est transcrit en une application logicielle haut niveau avec HTK, puis en langage de programmation Matlab.
La dernière étape consiste à implémenter cette reconnaissance vocale en langage bas niveau C et assembleur dans un DSP (Digital Signal Processor) à virgule flottante de chez Analog Devices et de type SHARC ADSP-21160M.

3 Schéma bloc du modèle choisi

L'acquisition et la détermination d'un son émis par le système vocal d'un être humain doit se faire en plusieurs étapes selon un archétype prédéfini. En se basant sur l'utilisation des chaînes de Markov cachées, voici le schéma standard choisi :

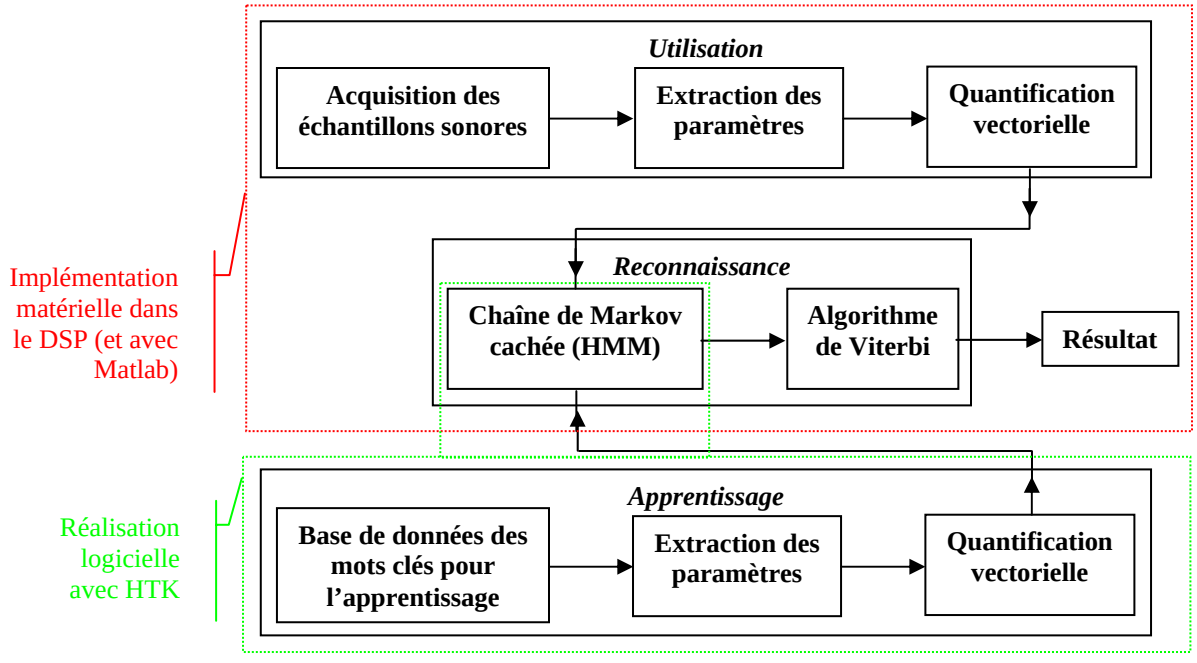


Figure 2 – Schéma bloc du système complet

Afin d'effectuer une reconnaissance vocale, deux phases sont nécessaires lors de la réalisation du système : une phase d'apprentissage et une phase d'utilisation.

La phase d'apprentissage consiste à entraîner les chaînes de Markov de manière à définir des modèles spécifiques. Ces modèles seront créés à partir de la base de données des mots utilisés pour la reconnaissance vocale. Cette étape sera réalisée de manière logicielle à l'aide de l'outil informatique HTK, qui a été spécialement conçu pour la reconnaissance vocale par HMM.
En effet, il serait trop compliqué d'implémenter la phase d'entraînement dans le DSP, c'est pour cela que je l'exécuterai directement avec HTK, et transférerai uniquement les résultats dans le code Matlab et le DSP.

La phase d'utilisation consiste à utiliser les chaînes de Markov entraînées lors de la phase d'apprentissage, afin de faire correspondre un signal reçu à l'entrée du système à un modèle d'entraînement enregistré. Elle sera entièrement implémentée dans le DSP, dont le code sera directement validé par Matlab.

Tous les blocs représentés à la figure 2 vont être détaillés dans les chapitres suivants.

4 Acquisition des échantillons sonores

Ce bloc permet l'acquisition et la quantification d'un son produit par le locuteur, afin de pouvoir traiter l'information à partir d'échantillon numérique.

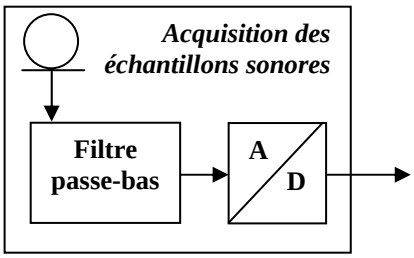


Figure 3 – Schéma de l'acquisition des échantillons sonores

Ce premier bloc est composé d'un :

- microphone unidirectionnel (associé à un préamplificateur) permettant de capter uniquement le son provenant du locuteur et évite ainsi la majeure partie du bruit environnant,
- filtre passe-bas qui permet de limiter la bande passante entre 0 et 8 [kHz], ce qui correspond à la largeur de bande pertinente du spectre de la voix,
- convertisseur analogique digital permettant d'échantillonner et de quantifier le signal vocal reçu.

Selon le théorème de Shannon, le convertisseur doit travailler au moins au double de la fréquence maximale de la largeur de bande. C'est pourquoi, on utilise une fréquence d'échantillonnage standardisée à 16 [kHz].

Le quantificateur ΣΔ (incorporé dans le convertisseur) travaillera avec une quantification non uniforme (suivant la loi μ logarithmique) sur 8 bits ; permettant ainsi d'obtenir une plage de quantification sur 256 valeurs, avec une meilleure résolution pour de faibles amplitudes.

Remarque : ce bloc d'acquisition des échantillons sonores et déjà réalisé sur la carte du DSP et ne nécessite pas de développement plus approfondi en code Matlab ou C.

5 Extraction des paramètres

Il existe deux modélisations distinctes du mécanisme de génération des sons par le système vocal ; ces deux modélisations se rapportent aux sons voisés et non voisés :

- Un signal voisé est modélisé par un filtre AR excité par un train d'impulsions, dont la fréquence (pitch) correspond à celle des vibrations des cordes vocales,
- Un signal non voisé est modélisé par un modèle AR excité par du bruit blanc.

Voici le modèle de production de sons :

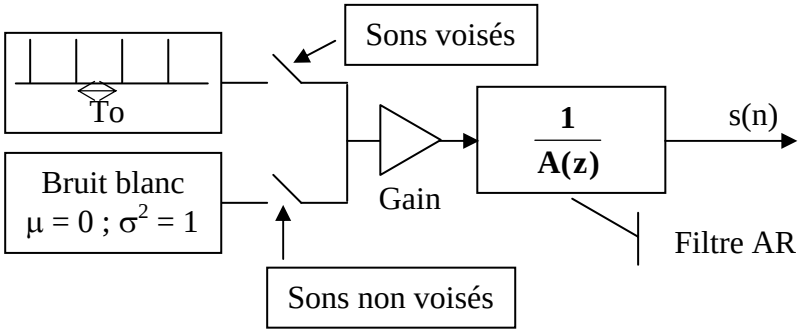


Figure 4 – Modèle AR pour le signal vocal

Ce modèle de codage par prédiction linéaire (LPC) simplifie le système phonatoire complet (conduit nasal, vocal et rayonnement des lèvres) en supposant que celui-ci ne comporte que des pôles. De ce fait, les paramètres pertinents sont les coefficients du filtre AR.

Le but de ce bloc d'extraction des paramètres est de retrouver ces coefficients LPC modélisant le système phonatoire lors de la prononciation de divers mots.
Il existe diverses méthodes amenant à des résultats similaires permettant de retrouver les paramètres LPC.

Lors du travail de semestre, deux méthodes ont été étudiées :

1. la méthode paramétrique modélisant le conduit vocal comme un système auto régressif (AR). Il existe divers procédés paramétriques, mais seule la méthode de l'autocorrélation a été étudiée, car elle est la plus utilisée en pratique.
2. la méthode par bloc (non paramétrique) utilisant les propriétés du signal sonore qui est composée de l'excitation des cordes vocales (pitch) convoluée avec la réponse impulsionnelle du conduit vocal.

Pour ce travail de diplôme, la méthode paramétrique a été choisie, car elle est plus aisée à implémenter sur un DSP (du point de vue du code). Cependant, l'inversion de la matrice de Toeplitz demande beaucoup de puissance de calcul et de ressource au DSP. De ce fait, il faudra implémenter l'algorithme de Levinson-Durbin qui évite l'inversion de la matrice. Ensuite, afin d'obtenir les coefficients cepstraux (utilisés par HTK), il faudra convertir les coefficients auto régressifs *a(k)* en coefficient cepstraux *c(k)*, à l'aide d'une formule de transformation.

L'autre méthode dite non paramétrique (composée de blocs distincts) à l'avantage de donner directement les coefficients MFCC (Mel Frequency Cepstral Coefficient). De plus, il n'y a pas de calcul matriciel. Cependant, implémenter une FFT et un filtre triangulaire en échelle de Mel dans un DSP serait un travail assez conséquent et complexe à réaliser. C'est pour cela que cette méthode n'a pas été retenue.

5.1 Développement théorique de la technique de l'autocorrélation

Le codage par prédiction linéaire consiste à estimer l'échantillon en cours à partir des échantillons du passé. La valeur estimée $\hat{y}[n]$ est calculée à partir des échantillons précédents pondérés par les coefficients a_k qui sont limités au nombre p de paramètres.

Calcul de l'identification d'un système AR par la méthode de l'erreur quadratique moyenne :

Observations : $y[n]$ signal de parole observé

Modèles AR : $y[n] = -\sum_{k=1}^{p} a_k y[n-k] + e[n]$ avec p = nombre de paramètres

Signal estimé : $\hat{y}[n] = -\sum_{k=1}^{p} a_k y[n-k] = \underline{w}^T \cdot \underline{\varphi n}$

avec $\underline{w}^T = -[a(1)\ a(2)\ a(3) \ldots a(p)]$

et $\underline{\varphi n}^T = [y(n-1)\ y(n-2)\ y(n-3) \ldots y(n-p)]$

Cherché : paramètres $a(k)$ avec $a_k = \underline{w}^T$

Critère de performance : La valeur des coefficients de prédiction a_k est obtenue par la minimisation de l'erreur quadratique moyenne (EQM) entre la valeur estimée et la valeur réelle

EQM :

$$EQM = E\left[(y[n]-\hat{y}[n])^2\right] = E\left[\left(y[n]-\sum_{k=1}^{p} a_k y[n-k]\right)^2\right]$$

$$= E\left[\left(y[n]-\underline{w}^T\cdot\underline{\varphi n}\right)^2\right]$$

$$= E\left[y[n]^2 - 2\underline{w}^T\cdot\underline{\varphi n}\cdot\underline{y[n]} + \underline{w}\cdot\underline{\varphi n}\cdot\underline{\varphi n}^T\cdot\underline{w}^T\right]$$

$$= E\left(y[n]^2\right) - 2\underline{w}^T\cdot E\left(\underline{\varphi n}\cdot\underline{y[n]}\right) + \underline{w}\cdot E\left(\underline{\varphi n}\cdot\underline{\varphi n}^T\right)\cdot\underline{w}^T$$

$$= r_{yy} - 2\underline{w}^T\cdot r_{\varphi y} + \underline{w}\cdot r_{\varphi\varphi}\cdot\underline{w}^T$$

Minimisation de l'EQM : Afin de minimiser l'erreur quadratique moyenne, on égale à zéro les dérivées partielles par rapport au vecteur w, soit les coefficients a_k.

$$\frac{\delta EQM}{\delta \underline{w}} = 0$$

$$\Rightarrow \begin{bmatrix} \frac{\delta EQM}{\delta w(0)} \\ \vdots \\ \frac{\delta EQM}{\delta w(p)} \end{bmatrix} = 0 - 2\cdot E\left(\underline{\varphi n}\cdot y[n]\right) + 2\cdot\underline{w}\cdot E\left(\underline{\varphi n}\cdot\underline{\varphi n}^T\right) = 0$$

$$\Rightarrow \underline{\boldsymbol{w}}_{\mathbf{opt}} = \boldsymbol{E\left(\underline{\varphi n}\cdot\underline{\varphi n}^T\right)^{-1}\cdot E\left(\underline{\varphi n}\cdot y[n]\right) = R_{\varphi\varphi}^{-1}\, r_{\varphi y}}$$

avec $R_{\varphi\varphi} = \underline{\varphi n}\cdot\underline{\varphi n}^T = \begin{bmatrix} r_{\varphi\varphi}(0) & r_{\varphi\varphi}(1) & \cdots r_{\varphi\varphi}(p-1) \\ r_{\varphi\varphi}(1) & r_{\varphi\varphi}(0) & \cdots r_{\varphi\varphi}(p-2) \\ \vdots & \vdots & \ddots \quad \vdots \\ r_{\varphi\varphi}(p-1) & r_{\varphi\varphi}(p-2) & \cdots \quad r_{\varphi\varphi}(0) \end{bmatrix}$

et $r_{\varphi y} = \underline{\varphi n}\cdot y[n] = [r_{xy}(0)\ r_{xy}(1)\ \ldots\ r_{xy}(p\text{-}1)]$

Remarque : Afin de simplifier le calcul de l'inversion de la matrice de Toeplitz symétrique R_{xx}, il est possible d'appliquer l'algorithme récursif de **Levinson-Durbin**, qui permet de diminuer le nombre de calculs.
Le développement théorique de cet algorithme est détaillé au chapitre 5.2.1, et l'algorithme est détaillé au chapitre 5. . .

5.2 Développement théorique de la méthode de Levinson-Durbin

5.2.1 La prédiction avant et arrière

Un des fondements utilisé dans l'algorithme de Levinson-Durbin [1] est l'utilisation des prédictions avant et arrière.
La prédiction avant permet de prédire la valeur du futur à partir des échantillons du passé, tandis que la prédiction arrière vérifie une valeur passée à partir de ces mêmes échantillons.

Pour le système AR d'ordre p défini précédemment, les prédictions sont définies comme suit :

Avant : $y_{PAv}(n) = -[a(1)\cdot y(n-1) + a(2)\cdot y(n-2) + ... + a(p)\cdot y(n-p)]$
Arrière : $y_{PAr}(n-p-1) = -[b(1)\cdot y(n-1) + b(2)\cdot y(n-2) + ... + b(p)\cdot y(n-p)]$

Pour le calcul de l'erreur de prédiction, il se calcule simplement en faisant la différence entre le signal original y et le signal prédit (avant ou arrière) y_{PAx} :

Erreur de prédiction avant :

$$e_{PAv}(n) = y(n) - y_{PAv}(n) = y(n) + \sum_{i=1}^{p} a(i)\cdot y(n-i) \Rightarrow \boldsymbol{e_{PAv}(n) = \sum_{i=0}^{p} a(i)\cdot y(n-i)}$$
$$\boldsymbol{avec\ a(0) = 1}$$

Erreur de prédiction arrière :

$$e_{PAr}(n) = y(n-p-1) - y_{PAr}(n-p-1) = y(n-p-1) + \sum_{j=1}^{p} b(j)\cdot y(n-j)$$
$$\Rightarrow \boldsymbol{e_{PAr}(n) = \sum_{j=1}^{p+1} b(j)\cdot y(n-j)}$$
$$\boldsymbol{avec\ b(p+1) = 1}$$

Dès lors, on peut définir les vecteurs de coefficients de prédiction avant et arrière :

Avant $\Rightarrow \underline{a} = [1 \;\; a(1) \;\; a(2) \;\; ... \;\; a(p)]^T$
Arrière $\Rightarrow \underline{b} = [b(1) \;\; b(2) \;\; ... \;\; b(p) \;\; 1]^T$

En faisant la transformée en Z des erreurs de prédiction, on peut déterminer les polynômes prédicteurs correspondants, ainsi que les erreurs de prédiction d'ordre *p* :

Erreur de prédiction avant :

$$E_{PAv}(z) = Z\left\{\sum_{i=0}^{p} a(i)\cdot y(n-i)\right\} = Y(z)\cdot\sum_{i=0}^{p} a(i)\cdot z^{-i}$$
$$= Y(z)\cdot\left\{1 + a(1)\cdot z^{-1} + a(2)\cdot z^{-2} + ... + a(p)\cdot z^{-p}\right\}$$

$$\Rightarrow \boldsymbol{E_{PAv}(z) = Y(z)\cdot A(z)}$$

$$avec\ A(z) = \left\{1 + a(1)\cdot z^{-1} + a(2)\cdot z^{-2} + ... + a(p)\cdot z^{-p}\right\}$$

Erreur de prédiction arrière :

$$E_{PAr}(z) = Z\left\{\sum_{j=1}^{p+1} b(j)\cdot y(n-j)\right\} = Y(z)\cdot\sum_{j=1}^{p+1} b(j)\cdot z^{-j}$$
$$= Y(z)\cdot\left\{b(1)\cdot z^{-1} + b(2)\cdot z^{-2} + ... + b(p)\cdot z^{-p} + z^{-(p+1)}\right\}$$

$$\Rightarrow \boldsymbol{E_{PAr}(z) = Y(z)\cdot B(z)}$$

$$avec\ B(z) = \left\{b(1)\cdot z^{-1} + b(2)\cdot z^{-2} + ... + b(p)\cdot z^{-p} + z^{-(p+1)}\right\}$$

5.2.2 Variance des erreurs de prédiction

La variance des erreurs de prédiction se calcule en prenant l'espérance de l'erreur de prédiction avant et arrière au carré :

Variance de l'erreur de prédiction avant :

$$\sigma^2{}_{PAv} = E\left[\left(\sum_{i=0}^{p} a(i)\cdot y(n-i)\right)\cdot\left(\sum_{j=0}^{p} a(j)\cdot y(n-j)\right)\right]$$
$$= E\left[\sum_{i,j=0}^{p} a(i)\cdot a(j)\cdot y(n-i)\cdot y(n-j)\right]$$
$$\sigma^2{}_{PAv} = \alpha = \sum_{i,j=0}^{p} a(i)\cdot a(j)\cdot r_{yy}(i-j) = \underline{\boldsymbol{a}}^{\boldsymbol{T}}\cdot\boldsymbol{R_{yy}}\cdot\underline{\boldsymbol{a}}$$

Variance de l'erreur de prédiction arrière :

$$\sigma^2{}_{PAr} = E\left[\left(\sum_{i=1}^{p+1} b(i)\cdot y(n-i)\right)\cdot\left(\sum_{j=1}^{p+1} b(j)\cdot y(n-j)\right)\right]$$
$$= E\left[\sum_{i,j=1}^{p+1} b(i)\cdot b(j)\cdot y(n-i)\cdot y(n-j)\right]$$
$$\sigma^2{}_{PAr} = \beta = \sum_{i,j=1}^{p+1} b(i)\cdot b(j)\cdot r_{yy}(i-j) = \underline{\boldsymbol{b}}^T\cdot\boldsymbol{R}_{yy}\cdot\underline{\boldsymbol{b}}$$

L'erreur de prédiction avant calculée précédemment est définie par :

$$e_{PAv}(n) = y(n) - y_{PAv}(n) = y(n) + \sum_{i=1}^{p} a(i)\cdot y(n-i)$$

De ce fait, la variance de l'erreur de prédiction avant vaut :

$$E\left[e_{PAv}(n)\cdot y(n)\right] = E\left[y(n)\cdot y(n)\right] + E\left[\sum_{i=1}^{p} a(i)\cdot y(n-i)\cdot y(n)\right] = r_{yy}(0) + \sum_{i=1}^{p} a(i)\cdot r_{yy}(i)$$

$$E\left[e_{PAv}(n)\cdot y(n)\right] = \sum_{i=0}^{p} a(i)\cdot r_{yy}(i) = E\left[e_{PAv}(n)\cdot\left\{\sum_{i=1}^{p} a(i)\cdot y(n-i) + e_{PAv}(n)\right\}\right]$$

Grâce au théorème d'orthogonalité : $\boldsymbol{E\left[e_{PAv}(n)\cdot y(n)\right] = E\left[e_{PAv}(n)^2\right]}$

car $e_{pav}(n)$ est orthogonal avec $y(n-i)$, si $i \geq 1$.

Par conséquent, on trouve : $\boldsymbol{E\left[e_{PAv}(n)^2\right] = \alpha = \sum_{i=0}^{p} r_{yy}(i)}$

avec α la variance de l'erreur de prédiction avant

On applique la même méthode pour la variance de l'erreur de prédiction arrière.

Ce qui nous donne le système suivant pour la prédiction avant :

$$\begin{bmatrix} r_{yy}(0) & r_{yy}(1) & \cdots & r_{yy}(p) \\ r_{yy}(1) & r_{yy}(0) & \cdots & r_{yy}(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_{yy}(p) & r_{yy}(p-1) & \cdots & r_{yy}(0) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ a(1) \\ \vdots \\ a(p) \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

et pour la prédiction arrière :

$$\begin{bmatrix} r_{yy}(0) & r_{yy}(1) & \cdots & r_{yy}(p) \\ r_{yy}(1) & r_{yy}(0) & \cdots & r_{yy}(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_{yy}(p) & r_{yy}(p-1) & \cdots & r_{yy}(0) \end{bmatrix} \cdot \begin{bmatrix} b(1) \\ \vdots \\ b(p) \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \beta \end{bmatrix}$$

Avec β la variance de l'erreur de prédiction arrière.

Relations entre prédiction avant et prédiction arrière

Si l'on définit A_{inv} étant la matrice dans laquelle on a inversé l'ordre des lignes et des colonnes de la matrice originale A, et si x_{inv} et y_{inv} sont respectivement les vecteurs x et y dans lesquels on a inversé l'ordre des composantes, le système :

$$\underline{x}_{inv} \cdot A_{inv} = \underline{y}_{inv} \equiv A \cdot \underline{x} = \underline{y}$$

La matrice symétrique R_{yy} de Toeplitz a la particularité qu'une telle transformation la laisse invariante. De ce fait, $\boldsymbol{R_{yy} = R_{yy\,inv}}$.

On peut donc écrire le système sous la forme inversée suivante :

$$\begin{bmatrix} r_{yy}(0) & r_{yy}(1) & \cdots & r_{yy}(p) \\ r_{yy}(1) & r_{yy}(0) & \cdots & r_{yy}(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_{yy}(p) & r_{yy}(p-1) & \cdots & r_{yy}(0) \end{bmatrix} \cdot \begin{bmatrix} a(p) \\ a(p-1) \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \alpha \end{bmatrix}$$

Si on le compare au système de la prédiction arrière, on observe que par identification :

$$\begin{bmatrix} a(p) \\ a(p-1) \\ \vdots \\ 1 \end{bmatrix} = \begin{bmatrix} b(1) \\ \vdots \\ b(p) \\ 1 \end{bmatrix} \quad \text{et} \quad \alpha = \beta$$

Ce résultat est tout à fait logique, car avec la prédiction avant on cherche la valeur de l'échantillon futur à partir de ceux du passé ; alors qu'avec la prédiction arrière on cherche à vérifier ce même échantillon à partir de ceux qui lui sont postérieurs.

C'est pour cela que les coefficients $a(1) = b(p)$, $a(2) = b(p-1)$ et ainsi de suite.

De ce fait, les polynômes prédicteurs sont liés par : $\boldsymbol{b(i) = a(p+1-i)}$
avec $i = 1, 2, \ldots, p+1$.

Par conséquent, si l'on fait la transformée en Z{}, on trouve :

$$Y(z)\cdot B(z) = Z\left\{\sum_{i=p+1}^{1} a(p+1-i)\cdot y(n-i)\right\} = Y(z)\cdot\sum_{i=p+1}^{1} a(p+1-i)\cdot z^{-i} \Rightarrow$$

$$B(z) = \sum_{i=p+1}^{1} z^{-(p+1)}\cdot a(i)\cdot z^{-i} = z^{-(p+1)}\cdot\left\{z^{-(p+1)}\cdot a(p+1) + z^{-p}\cdot a(p) + \ldots + z^{-1}\cdot a(1)\right\}\Rightarrow$$

$\boldsymbol{B(z) = z^{-(p+1)}\cdot A(z^{-1})}$ avec $A(z^{-1})$ est le même polynôme prédicteur que $A(z)$ avec des puissances positives pour chaque coefficient.

Finalement, les transformées des erreurs de prédiction E_{PAr} et E_{PAv} définies précédemment sont également reliées par la relation :

$$\boldsymbol{E_{PAr}(z) = z^{-(p+1)}\cdot E_{PAv}(z^{-1})}$$

5.2.3 Méthode de Levinson-Durbin

De façon à trouver une solution pour un système d'ordre $p+1$, on ajoute une composante nulle au dernier élément du vecteur $[1\ a(1)\ \ldots\ a(p)]$, ainsi qu'au premier élément du vecteur $[b(1)\ \ldots\ b(p)\ 1]$. A partir de cette transformation, on peut réécrire la matrice d'autocorrélation, ainsi que le vecteur du membre de droite que l'on avait obtenu lors du calcul des variances des erreurs de prédiction :

Prédiction avant :

$$\begin{bmatrix} r_{yy}(0) & r_{yy}(1) & \cdots & r_{yy}(p) & r_{yy}(p+1) \\ r_{yy}(1) & r_{yy}(0) & \cdots & r_{yy}(p-1) & r_{yy}(p) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ r_{yy}(p) & r_{yy}(p-1) & \cdots & \ddots & r_{yy}(1) \\ r_{yy}(p+1) & r_{yy}(p) & \cdots & r_{yy}(1) & r_{yy}(0) \end{bmatrix}\cdot\begin{bmatrix} 1 \\ a(1) \\ \vdots \\ a(p) \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha \\ 0 \\ \vdots \\ 0 \\ \mu \end{bmatrix}$$

Prédiction arrière :

$$\begin{bmatrix} r_{yy}(0) & r_{yy}(1) & \cdots & r_{yy}(p) & r_{yy}(p+1) \\ r_{yy}(1) & r_{yy}(0) & \cdots & r_{yy}(p-1) & r_{yy}(p) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ r_{yy}(p) & r_{yy}(p-1) & \cdots & \ddots & r_{yy}(1) \\ r_{yy}(p+1) & r_{yy}(p) & \cdots & r_{yy}(1) & r_{yy}(0) \end{bmatrix}\cdot\begin{bmatrix} 0 \\ b(1) \\ \vdots \\ b(p) \\ 1 \end{bmatrix} = \begin{bmatrix} \tau \\ 0 \\ \vdots \\ 0 \\ \beta \end{bmatrix}$$

avec μ et τ deux constantes non nulles.

Maintenant, nous pouvons poser les équations permettant de déterminer les constantes α, β, μ et τ :

Prédiction avant :
$$\begin{bmatrix} r_{yy}(0) & r_{yy}(1) & \cdots & r_{yy}(p) & r_{yy}(p+1) \\ r_{yy}(p+1) & r_{yy}(p) & \cdots & r_{yy}(1) & r_{yy}(0) \end{bmatrix} \cdot \begin{bmatrix} 1 \\ a(1) \\ \vdots \\ a(p) \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha \\ \mu \end{bmatrix}$$

Prédiction arrière :
$$\begin{bmatrix} r_{yy}(0) & r_{yy}(1) & \cdots & r_{yy}(p) & r_{yy}(p+1) \\ r_{yy}(p+1) & r_{yy}(p) & \cdots & r_{yy}(1) & r_{yy}(0) \end{bmatrix} \cdot \begin{bmatrix} 0 \\ b(1) \\ \vdots \\ b(p) \\ 1 \end{bmatrix} = \begin{bmatrix} \tau \\ \beta \end{bmatrix}$$

Avec ces quatre équations, on obtient le « système A » à quatre inconnues suivant :

$$\begin{cases} \alpha = r_{yy}(0) + \sum_{i=1}^{p} a(i) \cdot r_{yy}(i) & = \sum_{i=0}^{p} a(i) \cdot r_{yy}(i) \\ \beta = r_{yy}(0) + \sum_{j=p}^{1} b(j) \cdot r_{yy}(p-j+1) & = \sum_{j=p+1}^{1} b(j) \cdot r_{yy}(p-j+1) \\ \mu = r_{yy}(p+1) + \sum_{j=p}^{1} a(j) \cdot r_{yy}(p-j+1) & = \sum_{j=p}^{0} a(j) \cdot r_{yy}(p-j+1) \\ \tau = r_{yy}(p+1) + \sum_{i=1}^{p} b(i) \cdot r_{yy}(i) & = \sum_{i=1}^{p+1} b(i) \cdot r_{yy}(i) \end{cases}$$

SYSTEME A

Si l'on tient compte de la relation qui lie les coefficients *b(i)* aux *a(i)*, soit : $b(i) = a(p + 1 - i)$ *avec* $i = 1, 2, \ldots, p+1$, on observe que $\alpha = \beta$ *et* $\mu = \tau$.

A cause de la présence des constantes non nulles μ et τ, la solution d'ordre $p+1$ est une combinaison linéaire des deux solutions proposées pour a_p et b_p :

Solution d'ordre p + 1 : $a_{p+1}(i) = a_p(i) + k_{p+1} \cdot b_p(i)$

Transformée en Z{} : $A_{p+1}(z) = A_p(z) + k_{p+1} \cdot B_p(z)$

On a observé dans le système de quatre équations à quatre inconnues que $\alpha = \beta$ et $\mu = \tau$. De ce fait, deux équations sont redondantes et nous allons seulement résoudre le système pour α et μ.

Posons R_{yy}' la matrice partielle de R_{yy} qui donnera des valeurs non nulles :

$$R_{yy}' = \begin{bmatrix} r_{yy}(0) & \cdots & r_{yy}(p+1) \\ r_{yy}(p+1) & \cdots & r_{yy}(0) \end{bmatrix}$$

Selon la solution d'ordre $p+1$ précédente, on peut écrire :

$$R_{yy}' \cdot \begin{bmatrix} a_{p+1}(0) \\ a_{p+1}(1) \\ \vdots \\ a_{p+1}(p) \\ 0 \end{bmatrix} = R_{yy}' \cdot \begin{bmatrix} 1 \\ a_p(1) \\ \vdots \\ a_p(p) \\ 0 \end{bmatrix} + k_{p+1} \cdot R_{yy}' \cdot \begin{bmatrix} 0 \\ b_p(1) \\ \vdots \\ b_p(p) \\ 1 \end{bmatrix}$$

Après résolution de l'équation (en utilisant le système A précédent), on obtient le système suivant :

$$\begin{bmatrix} \alpha_{p+1} \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha_p \\ \mu_p \end{bmatrix} + k_{p+1} \cdot \begin{bmatrix} \tau_p \\ \beta_p \end{bmatrix}$$

puisque $\alpha = \beta$ et $\mu = \tau$, on obtient : $\begin{bmatrix} \alpha_{p+1} \\ 0 \end{bmatrix} = \begin{bmatrix} \alpha_p \\ \mu_p \end{bmatrix} + k_{p+1} \cdot \begin{bmatrix} \mu_p \\ \alpha_p \end{bmatrix}$

Avec ces deux équations, on obtient le « système B » à deux inconnues suivant :

$$\begin{cases} \alpha_{p+1} = \alpha_p + k_{p+1} \cdot \mu_p \\ 0 = \mu_p + k_{p+1} \cdot \alpha_p \end{cases}$$

SYSTEME B

En prenant la deuxième équation, on trouve : $\alpha_p = \beta_p = -\dfrac{\mu_p}{k_{p+1}}$

Dans le « SYSTEME A » précédent, on a pu calculer μ_p et α_p. De ce fait, on peut déterminer la valeur de k_{p+1} :

$$k_{p+1} = -\frac{\mu_p}{\alpha_p} = -\left[r_{yy}(p+1) + \sum_{j=p}^{1} a(j) \cdot r_{yy}(p-j+1) \right] \cdot \left(\frac{1}{\alpha_p} \right)$$

Finalement, en prenant la première équation, on trouve : $\alpha_{p+1} = \alpha_p \cdot \left(1 - k_{p+1}^{\,2}\right)$

De ce fait, il suffit de remplacer k_{p+1} par sa valeur dans cette équation, et on trouve α_{p+1}.

5.3 Réalisation de la méthode de l'autocorrélation

La méthode paramétrique appelée méthode de l'autocorrélation consiste à faire une autocorrélation du signal à traiter, puis de sélectionner le nombre de paramètres (nombre d'échantillons corrélés) souhaités et de les stocker dans un vecteur r_{xx}.
Ces paramètres sont ensuite disposés selon une matrice de Toeplitz symétrique (toutes les valeurs de sa diagonale sont identiques).
Finalement, on inverse cette matrice et on fait le produit matriciel avec le vecteur r_{xx} calculé précédemment décalé d'un échantillon sur la droite. Le résultat obtenu est un vecteur contenant les coefficients *a(k)* du modèle AR.

Cependant, puisque le calcul matriciel n'est pas économe du point de vue temps et des opérations pour un DSP, on implémente l'algorithme de Levinson-Durbin afin de calculer de manière récursive les coefficients *a(k)*.

Voici le schéma bloc du module permettant l'extraction des paramètres :

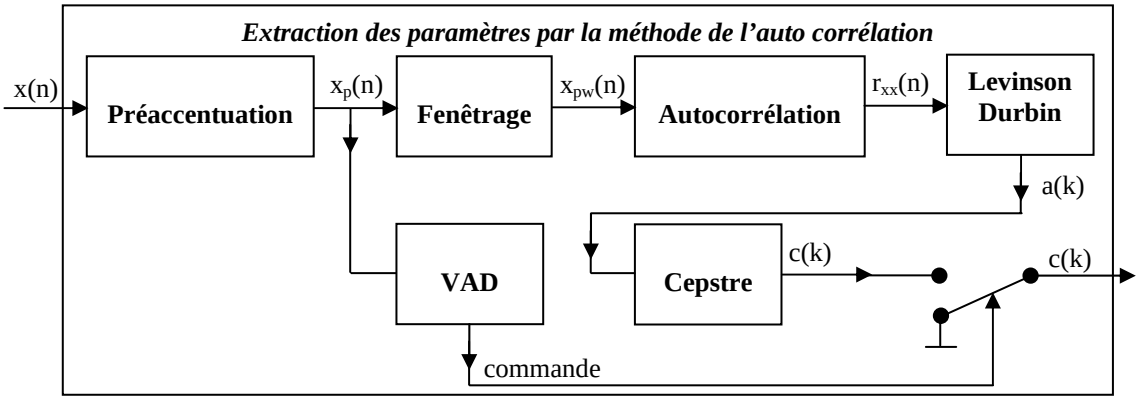


Figure 5 – Schéma bloc du module d'extraction des paramètres

5.3.1 Détail de chaque bloc

5.3.1.1 Préaccentuation

La **préaccentuation** consiste à augmenter l'amplitude des fréquences élevées par rapport aux basses fréquences, ou inversement, à diminuer l'amplitude des basses par rapport aux aigues.
Ceci se fait en général à l'aide d'un filtre FIR passe-haut du 1er ordre :

$$H(z) = 1 - a \cdot z^{-1}, \text{ avec } 0.9 \leq a \leq 1.0.$$

Par conséquent, pour un signal *x(n)* en entrée, le signal de sortie préaccentué $x_p(n)$ vaudra :

$$x_p(n) = x(n) - a \cdot x(n-1), \text{ avec } a = \frac{15}{16} = 0.9375$$

Cette valeur du coefficient *a* est la plus souvent utilisée afin d'effectuer un filtre de préaccentuation.

Voici la représentation en spectre bilatéral pour l'amplitude et la phase de ce filtre :

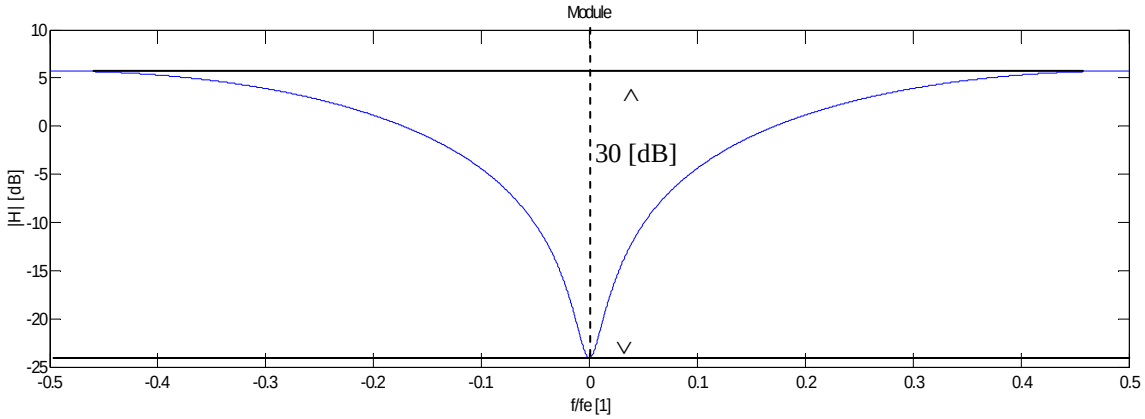


Figure 6 – Spectre bilatéral normalisé du module du filtre de préaccentuation (Matlab)

Ce filtre créé une augmentation de 30 [dB] pour f = fe/2 (8 [kHz]), par rapport à une fréquence nulle.

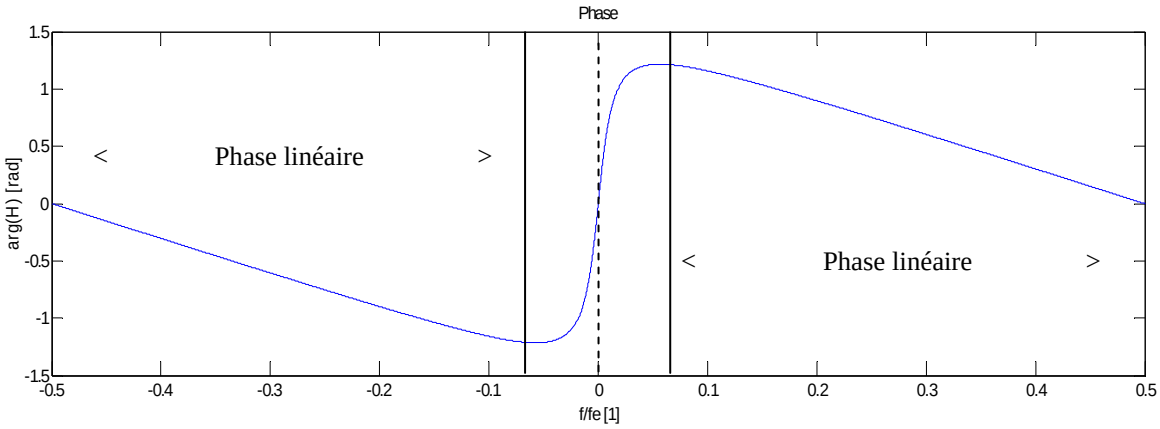


Figure 7 – Spectre bilatéral normalisé de l'argument du filtre de préaccentuation (Matlab)

La phase de ce filtre est linéaire entre 1 et 8 [kHz], soit sur le 87 % de la bande de fréquence comprise entre 0 et fe/2. Par conséquent, le retard introduit par ce filtre est presque constant sur toute la plage fréquentielle.

5.3.1.2 Fenêtrage

Le **fenêtrage** consiste à pondérer chacun des échantillons temporels reçus selon une fenêtre bien précise. Il existe différents types de fenêtre (Hamming, Hanning, Kaiser,…).Celle qui est la plus utilisée en reconnaissance de la parole est la fenêtre de Hamming qui fait moins confiance aux échantillons situés hors de l'intervalle de stationnarité délimité par les lignes rouge sur la figure 8 :

$$w(n) = 0.54 - 0.46 \cdot \cos\left(\frac{2 \cdot \pi \cdot (n-1)}{N-1}\right)$$, **avec** $0 \leq n \leq N\text{-}1$ (**N = longueur de la fenêtre**)

La longueur N de la fenêtre est caractérisée par la durée pendant laquelle on estime que le signal de la parole est stationnaire. Après plusieurs mesures, il a été établi que le signal de la parole est stationnaire pour une durée de 10 [ms].
Par conséquent, N doit être plus grand que la durée minimale d'une trame dite stationnaire, afin de ne pas éliminer des paramètres pertinents. Pour une trame stationnaire de durée égale à 10 [ms] et une fréquence d'échantillonnage de 16 [kHz], on a un nombre d'échantillons = $10 \cdot 10^{-3} \cdot 16 \cdot 10^{3}$ = **160**.
De ce fait, la longueur de la fenêtre de Hamming doit être plus grande que 160. La taille idéale de la fenêtre doit être supérieure au double de la durée de stationnarité du signal. Dès lors, on prendra une fenêtre de durée de 25 [ms], soit N = **400 échantillons**.

Voici la représentation de la réponse impulsionnelle de cette fenêtre :

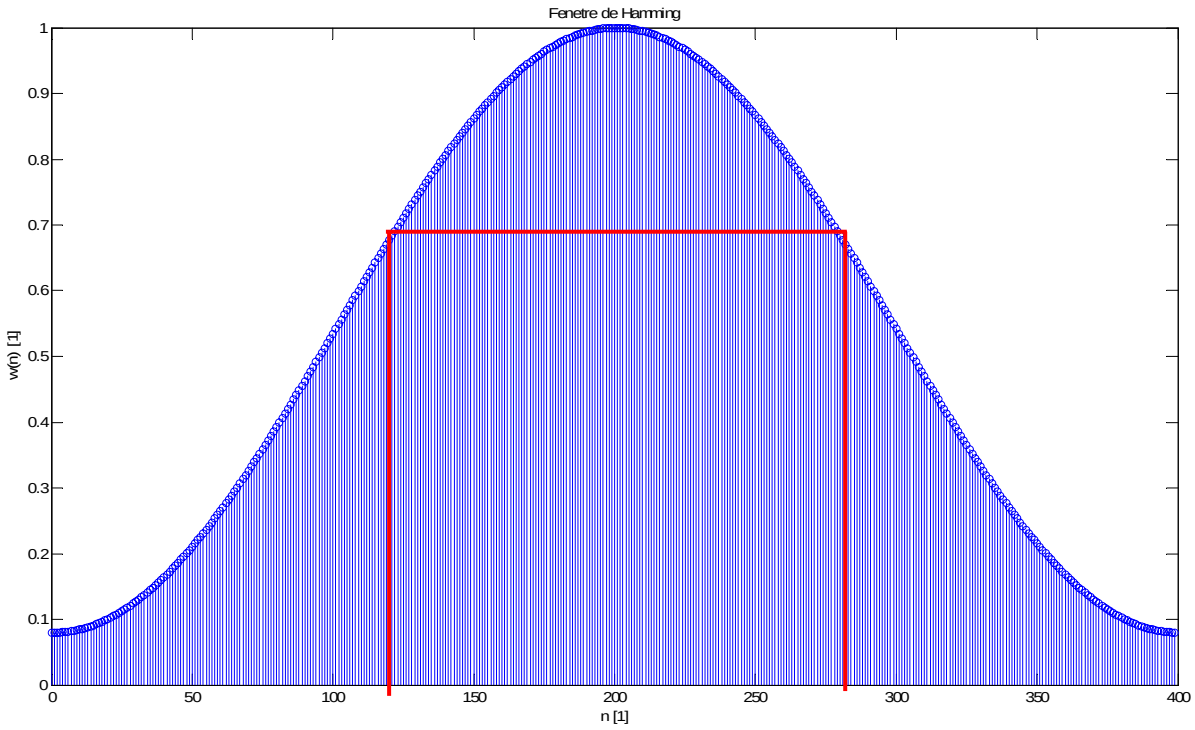


Figure 8 – Réponse impulsionnelle de la fenêtre de Hamming (Matlab)

Le signal obtenu après fenêtrage est le signal x_p pondéré par la fenêtre de Hamming, soit : $x_{pw}(n) = x_p(n) \cdot w(n)$.

Représentation de l'application des fenêtres de Hamming et de stationnarité :
Comme expliqué ci-dessus, la fenêtre de stationnarité est prise sur une durée de 10 [ms] (correspondant à 160 échantillons), et la fenêtre de Hamming est prise sur une durée de 25 [ms] (correspondant à 400 échantillons) avec fe = 16 [kHz].

Les trames d'échantillons utilisées pour faire l'autocorrélation sont traitées et sélectionnées de la manière suivante :

- On prend les 400 premiers échantillons que l'on pondère par la fenêtre de Hamming.
- On calcule l'autocorrélation sur ces 400 échantillons, afin de sortir 14 paramètres pertinents.
- Ensuite, on se décale de la durée de stationnarité, soit de 160 échantillons et l'on prend les 400 échantillons à partir du 160ème (échantillons 160 à 560), que l'on pondère par la fenêtre de Hamming.
- On calcule l'autocorrélation sur ces 400 nouveaux échantillons, afin de sortir 14 nouveaux paramètres pertinents. Et ainsi de suite jusqu'à la fin du signal.

Visualisation du déplacement des fenêtres :

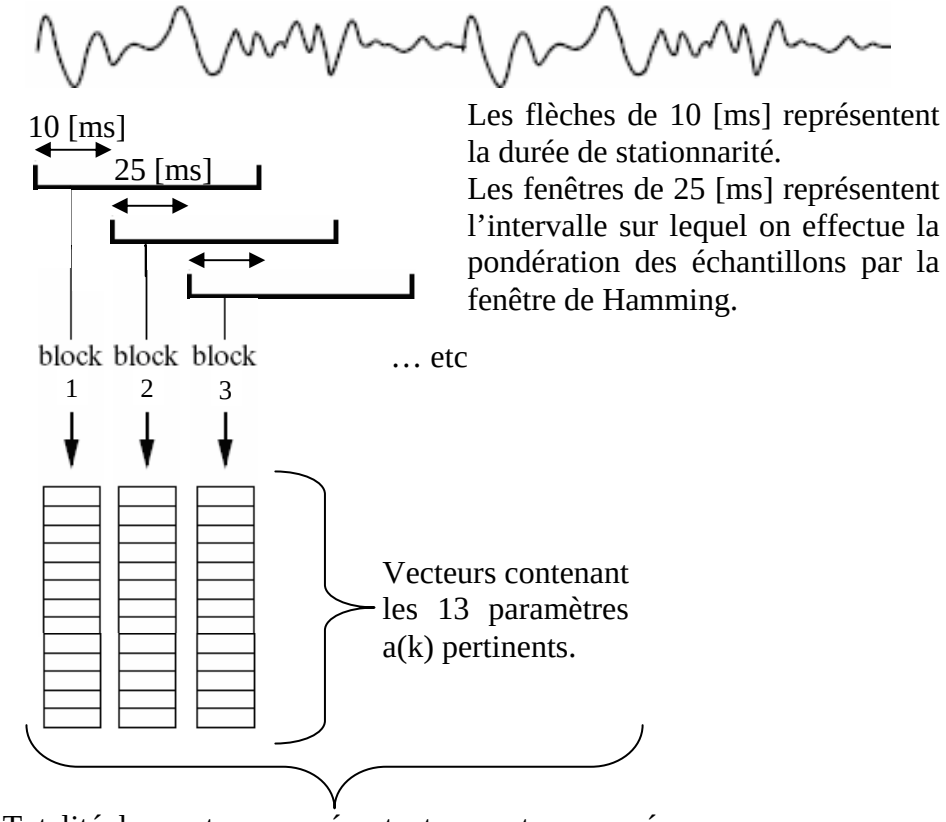


Figure 9 – Déplacement des fenêtres de stationnarité et de Hamming

5.3.1.3 Autocorrélation

L'**autocorrélation** du signal sur un nombre de paramètres *p* définis (14 pour ce travail) permet de sortir un vecteur de 14 composantes appelé r_{xx}.
Voici la formule de l'autocorrélation pour 14 paramètres :

$$r_{xx}[n] = \sum_{k=0}^{14} x[n] * x[n+k]$$

5.3.1.4 Algorithme de Levinson-Durbin

L'algorithme de **Levinson-Durbin** permet de calculer de manière récursive les coefficients *a(k)* à partir du vecteur d'autocorrélation r_{xx}. Dans le détail de l'algorithme, les indices indiquent si l'on est dans l'élément courant $_{p}$, précédent $_{p-1}$ ou suivant $_{p+1}$) :

Phase d'initialisation :

$a_p(0) = 1$; % Normalisation du 1er coefficient a(k) à 1.
$k_p(0) = 0$; % Initialisation du 1er coefficient du vecteur de réflexion k
% (si l'on souhaite utiliser ce vecteur par la suite).
$\alpha_0 = r_{xx}(0)$; % Initialisation de la variance de l'erreur de prédiction

Phase de récursion :

pour (*m = 0* jusqu'à *p-1*) boucle

$$k_{p+1} = -\left[\sum_{j=m-1}^{0} a_{p-1}(j) \cdot r_{yy}(m-j)\right] \cdot \left(\frac{1}{\alpha_p}\right)$$; % Calcul du nouveau k.

$k_{p+1}(m+1) = k_{p+1}$; % Affectation du k au vecteur de coefficients de
% réflexion.

pour (*i = 1* jusqu'à *m-1*) boucle

$a_p(i) = a_{p-1}(i) + k_p(m) \cdot a_{p-1}(m-i)$; % Calcul du a(k) courant.

$a_{p+1}(m) = k_{p+1}$; % Initialisation du prochain coefficient a(k).

$\alpha_{p+1} = \alpha_p \cdot \left(1 - k^2_{p+1}\right)$; % Calcul de la variance de l'erreur de
% prédiction.

fin

fin

On obtient comme vecteurs de sortie, les coefficients *a(k)* normalisés (« a_p » dans cet algorithme), ainsi que les coefficients de réflexion *k* (« k_p » dans cet algorithme).

Remarque : il y a 13 coefficients *a(k)* pertinents, car le coefficient a(0) = 1 n'apporte pas d'information.

5.3.1.5 Conversion en cepstres

Le bloc **Cepstre** permet de transformer les coefficients *a(k)* en *c(k)* par la formule de transformation suivante :

$$c[n] = -a[n] - \frac{1}{n} \cdot \sum_{i=1}^{n-i} (n-i) \cdot a[i] \cdot c[n-i]$$

Le cepstre fournit une méthode pour réduire la quantité d'informations dans la mesure où en conservant les premiers coefficients cepstraux, on peut lisser la représentation spectrale d'un signal afin de conserver l'enveloppe spectrale caractéristique du signal de la parole. De plus, les coefficients cepstraux ont l'avantage d'être décorrélé et cela permet d'utiliser uniquement la diagonale de la matrice de covariance (ceci est très utile afin de définir les modèles de HMM). Dans notre cas, le nombre de cepstres et de 13, car on a 13 coefficients *a(k)* pertinents.

Sous HTK, le vecteur de coefficients cepstraux subit une normalisation, car plus le nombre de cepstres est grand, plus leurs amplitudes sont petites, ce qui augmente leurs variances. On leurs applique un « liftrage » (filtrage dans le domaine temporel en échelle logarithmique) avec la formule suivante :

$$c'[n] = \left(1 + \frac{L}{2} \cdot \sin\frac{\pi \cdot n}{L}\right) \cdot c[n]$$

Avec L un coefficient en relation avec le nombre de cepstres. Sous HTK, un exemple montre que $L = 2 \cdot n$, soit dans notre cas où l'on utilise 13 cepstres, cela donne $L = 26$.

5.3.1.6 VAD (détecteur d'activité de voix)

Le **VAD** (Voice Activity Detection) est un bloc qui permet de détecter si le signal reçu contient une information pertinente de parole, ou s'il ne contient que du bruit.
Il donnera l'ordre au système de décoder le signal reçu s'il s'agit d'un mot prononcé, ou de ne pas le traiter s'il ne s'agit que de bruit.

Il existe divers types de VAD (voici les plus connus) :

- Méthode de seuillage par énergie donnant de bonnes performances pour des SNR (Signal to Noise Ratio) élevés, mais pas pour les faibles SNR.
- Mesure de deux énergies distinctes utilisant une durée à court terme et une à long terme et délimitées par deux seuils (parole et bruit).
- Utilisation de la mesure de la périodicité de la parole appelée LSPE (Least-Square Periodicity Estimator). Cependant la plus grande partie des sons de la vie courante ne sont pas périodiques, ce qui le rend inutilisable pour de faibles SNR.
- Technique se basant sur le nombre de passages par zéro pour différencier la parole du bruit. Cette méthode utilise l'hypothèse que le nombre de passages par zéro pour le bruit est considérablement plus élevé que pour la parole.
- Utilisation de modèles statistiques afin de différencier le bruit de la parole. Par exemple, l'utilisation d'une règle de décision bayesienne donne de bons résultats de segmentation parole/bruit.

Le fonctionnement de ce VAD consiste à calculer la valeur moyenne de l'énergie du signal pendant une longue, puis une courte durée. Ensuite, on compare cette valeur avec deux seuils distincts : un seuil de bruit et un seuil de signal de la parole. De ce fait, le système peut déterminer s'il s'agit d'une voix ou simplement du bruit ambiant. De plus, le VAD est adaptatif en ce sens que les niveaux des seuils s'adaptent en fonction de l'énergie moyenne de la portion de signal mesurée et en observant le nombre de passages par zéro.

Il est à noter que j'ai repris l'idée de la mesure des énergies et des deux seuils distincts [5], mais que je l'ai adaptée avec la technique se basant sur le nombre de passages par zéro [5].

L'algorithme est de ce fait un mélange entre deux méthodes, l'une basée sur le calcul des énergies moyennes pendant deux types de durées distinctes, soit t_1 = 100 [ms] et t_2 = 50 [ms] et l'autre sur la mesure du nombre de passages par zéro. En observant le nombre de passages par zéro sur une durée de 50 [ms], on obtient pour un bruit Gaussien avec un SNR de 20 [dB], environ 400 transitions. Pour un signal de parole, entre 50 et 250 transitions, sauf pour les sons non voisés ou l'on peut atteindre jusqu'à 380 transitions. De ce fait, il ne faut pas confondre un son non-voisé avec du bruit ambiant.

L'algorithme est décrit à la page suivante. Il est composé de deux phases, une d'initialisation (point 1 à 3) et une de fonctionnement (point 4 à 12). Les 12 étapes sont schématisées dans l'organigramme de la page 27. Les numéros cerclés sur l'organigramme renvoient aux numéros des étapes de l'algorithme.

Description de l'algorithme :

La variable **commande** est définie comme suit : - commande = 0 ⇒ du bruit est détecté
- commande = 1 ⇒ un mot est détecté

1. Mesure la valeur moyenne du logarithme de l'énergie du bruit pendant t_n (entre 0.5 à 1 seconde) ⇒ $\mathbf{E_n}$.
   Mesure du nombre de transitions par zéro du bruit ⇒ **nb-tr-b**.
   Calcul de E_x qui vaut (entre 0 à 6 [dB]) de plus que E_n ⇒ $\mathbf{E_x = E_n + offset}$.
   Après divers tests, j'ai défini l'offset à 5 [dB], car il donne de bons résultats.

2. Mesure la valeur moyenne du logarithme de l'énergie pendant t_1 ⇒ $\mathbf{E_1}$.
   Calcul du seuil de détection de la parole ⇒ $\mathbf{T_s = 10 \cdot log((10^{E1/10} + 10^{Ex/10})/2)}$.
   Calcul du seuil de détection du bruit ⇒ $\mathbf{T_n = 10 \cdot log((10^{E1/10} + 10^{En/10})/2)}$.

3. Mesure la valeur moyenne du logarithme de l'énergie pendant t_2 ⇒ $\mathbf{E_2}$.

4. Test du signal courant : **si $E_2 > T_s$ ⇒ commande = 1 ⇒ saut au point 9**
   sinon commande = 0 ⇒ suite.

5. Conserve la valeur de E_1 et calcule le niveau du seuil de bruit ⇒ $\mathbf{T_n = E_1 + E_n}$.

6. Test du signal courant : **si $E_2 > T_n$ ⇒ suite**
   sinon bruit détecté ⇒ saut au point 3.

7. Mise à jour du niveau du bruit ⇒ $\mathbf{E_n = 10 \cdot log((10^{(E2+En)/20})}$ et $\mathbf{E_x = E_n + offset}$.
   Calcul du nouveau seuil de détection de parole ⇒ $\mathbf{T_s = 10 \cdot log((10^{E1/10} + 10^{Ex/10})/2)}$.

8. Mesure la valeur moyenne du logarithme de l'énergie pendant t_2 ⇒ $\mathbf{E_2}$.
   Saut au point 4.

9. Mesure la valeur moyenne du logarithme de l'énergie pendant t_2 ⇒ $\mathbf{E_2}$.
   Mesure du nombre de passages par zéro du bruit ⇒ **nb-pp-z**.

10. Test du signal courant :
    si nb-pp-z > nb-tr-b ⇒ commande = 0
    sinon commande = 1.

11. Test s'il faut faire un lissage sur la commande :
    si $commande_{-2} = 1$ ET $commande_{-1} = 0$ ET commande = 1 ⇒ il s'agit du même mot ⇒ $\mathbf{commande_{-1} = 1}$.
    sinon
    si $commande_{-2} = 0$ ET $commande_{-1} = 1$ ET commande = 0 ⇒ il s'agit d'un parasite ⇒ $\mathbf{commande_{-1} = 0}$.

12. Saut au point 4.

Voici une représentation montrant l'évolution de la fenêtre de durée t_2 se déplaçant pour le calcul des diverses énergies :

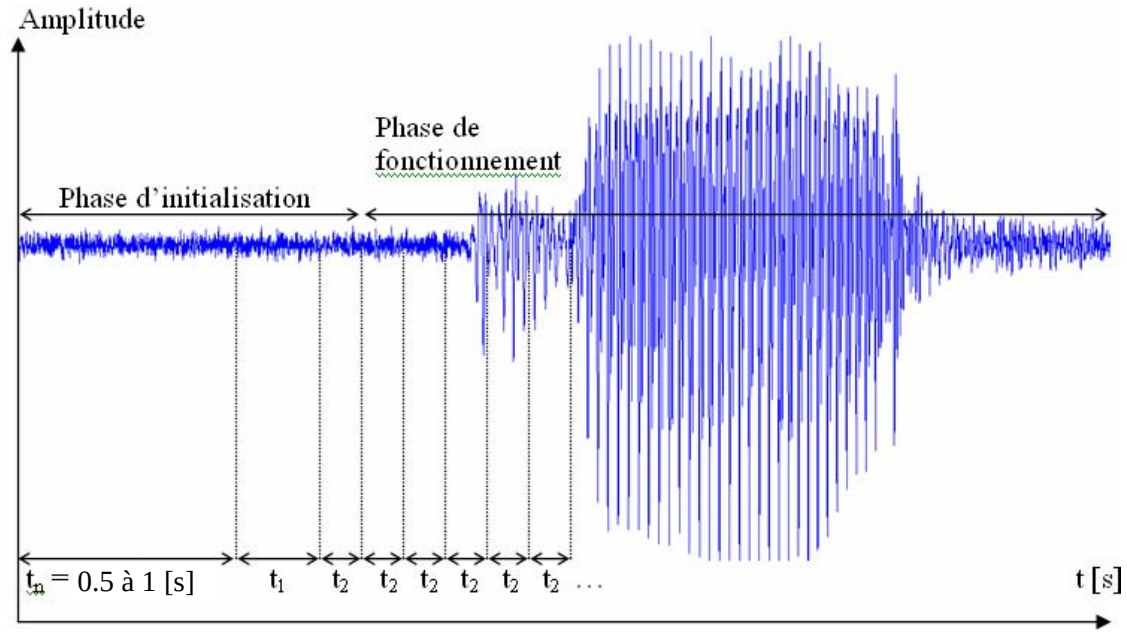


Figure 10 – Evolution de la fenêtre de durée t_2 pour le calcul des niveaux d'énergies

On observe bien les deux phases distinctes :

- la phase d'initialisation qui dure environ une seconde et pendant laquelle le signal ne doit être composé que du bruit ambiant,
- la phase de fonctionnement pendant laquelle on mesure à chaque itération dans la boucle de calcul l'énergie E2 pendant la durée t2, et l'on exécute les diverses opérations de l'algorithme.

A la page suivante, l'organigramme détaillé permet de visualiser les diverses opérations de l'algorithme, ainsi que la boucle de calcul.

Organigramme détaillé :

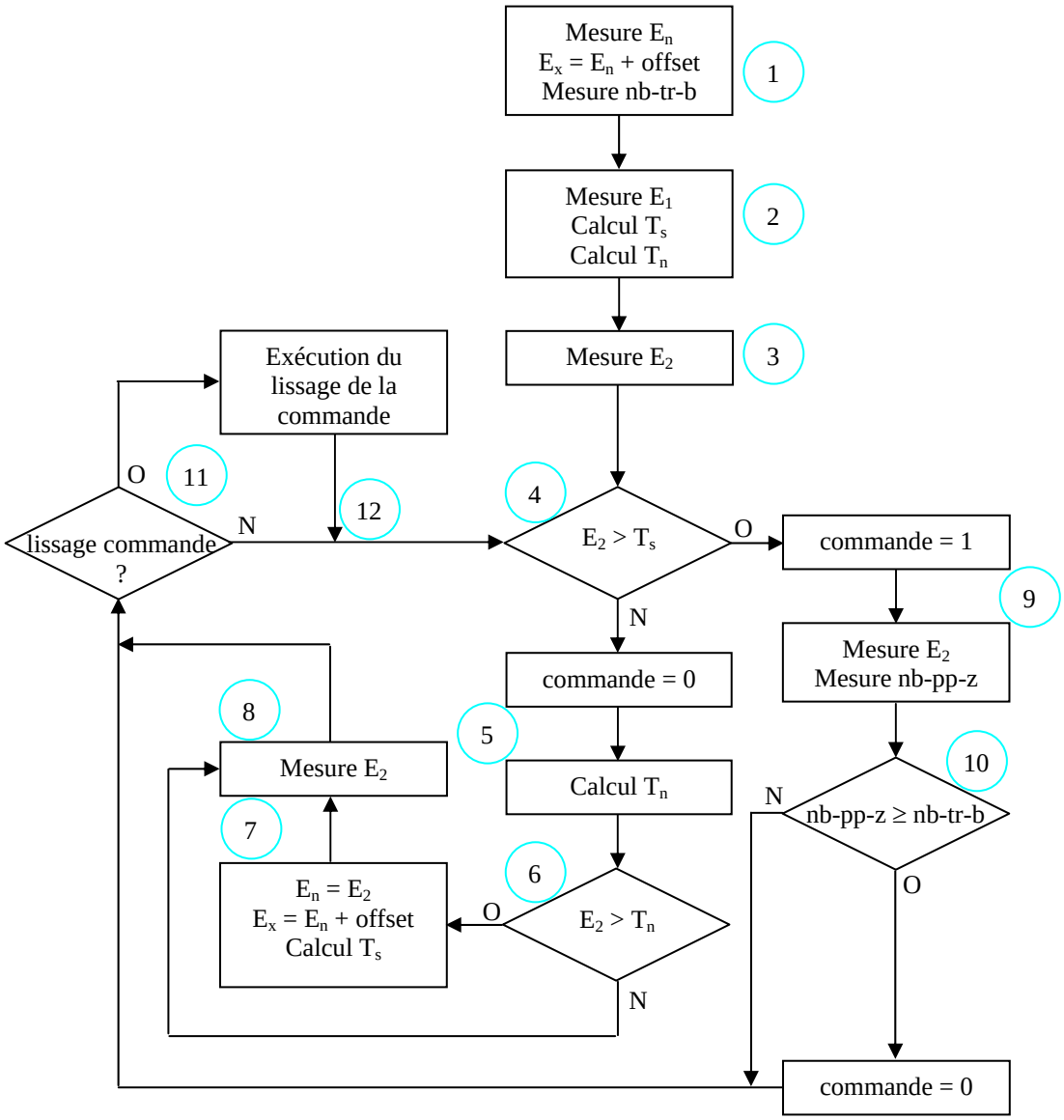


Figure 11 – Organigramme détaillé de l'algorithme du VAD

Comme expliqué précédemment, les points 1 à 3 représentent la phase d'initialisation et les points 4 à 12 la phase de fonctionnement avec la boucle de calcul.

5.3.2 Simulation et validation du système d'extraction des paramètres

Afin de déterminer le bon fonctionnement du système d'extraction des paramètres, on réalise une « cross-validation » entre les blocs réalisés avec Matlab et les résultats fournis par HTK.
Le signal test utilisé est le mot « ONE » d'une durée de 2 [s] avec une fréquence d'échantillonnage de 16 [kHz]. A partir de ce signal, on va pouvoir visualiser la distribution des pôles du système AR dans le cercle unité, la réponse fréquentielle du conduit vocal, et les coefficients cepstraux *c(k)* pour une trame de 400 échantillons.
Pour ce signal de 32000 échantillons, il y a 200 trames de 160 valeurs (chacune correspondant à une stationnarité de 10 [ms]). De ce fait, on visualisera les résultats obtenus avec la 100ème tranche de signal.

Voici la disposition des pôles du système AR modélisant le conduit vocal :

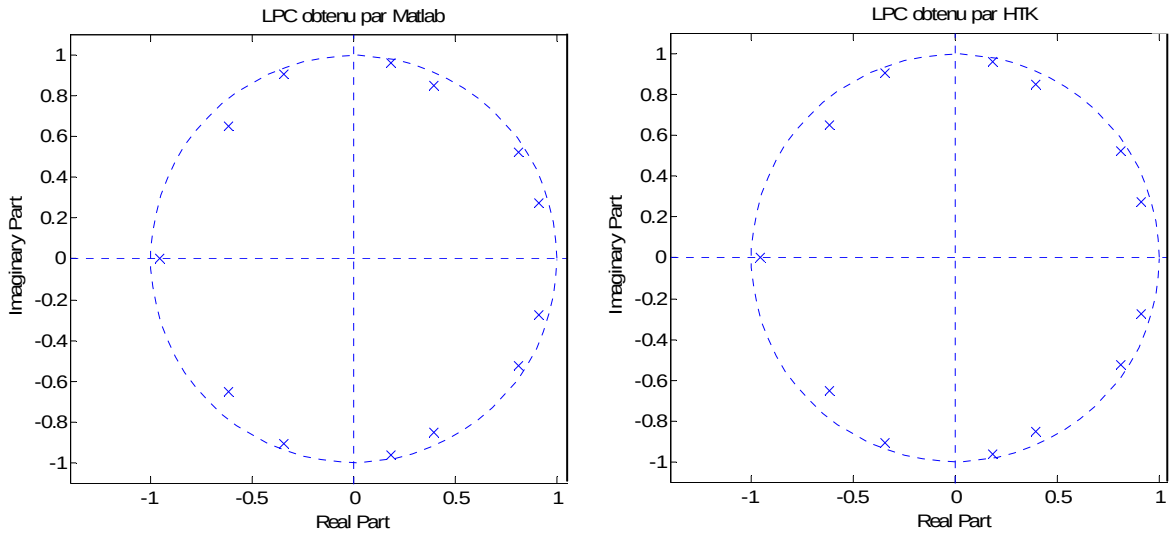


Figure 12 – Représentation de la comparaison des pôles du système AR de la 100ème tranche obtenu avec l'implémentation Matlab et fourni par le logiciel HTK (Matlab)

On observe que la disposition des pôles pour la 100ème tranche est identique entre le code Matlab et l'outil HTK.

Sur les graphiques de la page suivante, les traces bleues représentent la réponse fréquentielle et les cepstres obtenus par les blocs Matlab, et les traces rouges correspondent aux résultats donnés par HTK.

Voici la contribution du conduit vocal recouvrée :

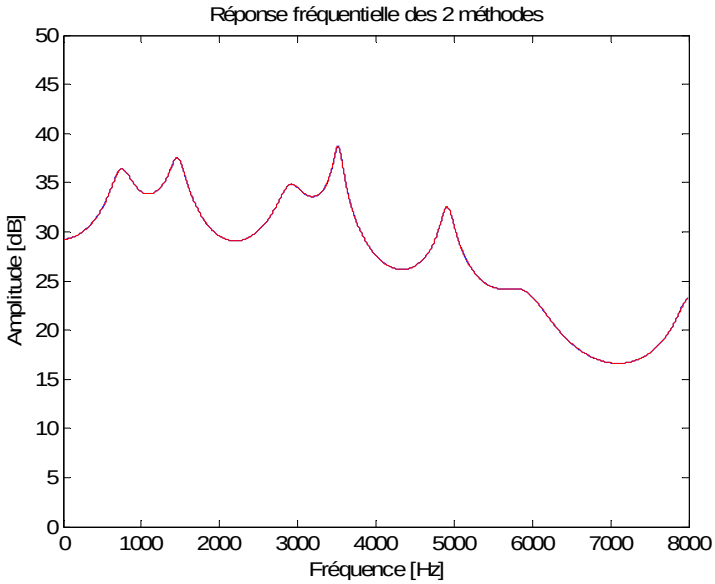


Figure 13 – Comparaison de la contribution du conduit vocal recouvrée à partir des coefficients LPC entre l'implémentation Matlab et les résultats fournis par HTK (Matlab)

On observe que les courbes se superposent de manière parfaite. De ce fait, on peut valider le bon fonctionnement du code Matlab calculant les coefficients LPC.
Voici les coefficients cepstraux *c(k)* obtenus par la transformation des *a(k)* :

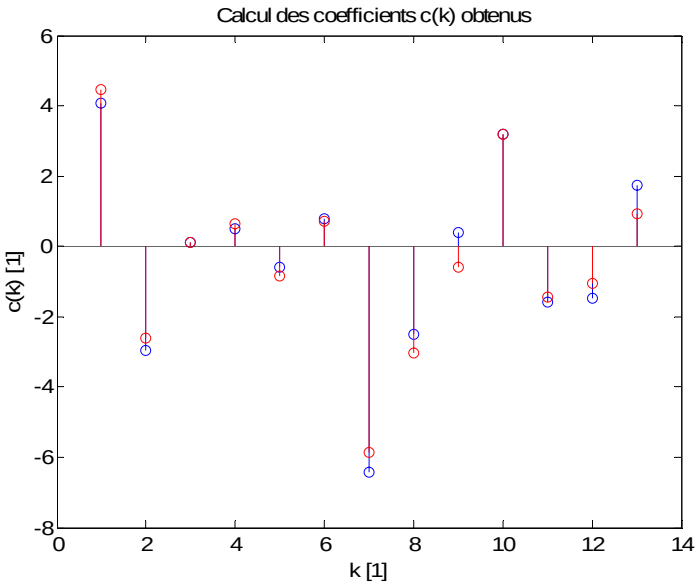


Figure 14 – Comparaison des cepstres c(k) obtenus à partir des coefficients LPC entre l'implémentation Matlab et les résultats fournis par HTK (Matlab)

Les *c(k)* des deux méthodes sont sensiblement identiques. Les faibles fluctuations d'amplitudes sont dues à un facteur de normalisation qui doit être légèrement différent sous HTK.

5.3.3 Simulation du VAD pour divers niveaux de bruits

Quand une personne parle normalement, sans chuchoter ou crier, le niveau sonore de sa voix se situe entre 35 et 45 [dB] environ. De ce fait, le but du test est de vérifier le bon fonctionnement du VAD en y ajoutant un bruit additif variant de 0 à 40 [dB], soit en faisant varier le rapport signal/bruit SNR entre 40 et 0 [dB].
Il est clair que pour un SNR de 0 [dB] (soit le niveau du bruit est égal au niveau du signal), le VAD ne peut fonctionner. De ce fait, on s'arrêtera à un SNR de 5 [dB].

Les tests sont réalisés sur un signal provenant de ma voix qui a été enregistrée dans la chambre anéchoïdale de l'EIVD afin d'obtenir un signal le plus pur possible.
Par la suite, on lui ajoute du bruit additif obtenu aléatoirement selon une distribution Gaussienne et dont l'amplitude varie en fonction du SNR.

Le signal est composé des dix mots de vocabulaire (prononcés en anglais et de manière aléatoire) que le système doit être capable de reconnaître

Ci-après, on observe les graphiques représentant les divers résultats obtenus. Avec en premier les mots prononcés ; en deuxième le niveau d'énergie de chaque mot, ainsi que le seuil de détection du bruit en vert et le seuil de détection de la parole en rouge ; en troisième le signal indiquant si de la voix a été détectée ou non.

SNR = 40 [dB] :

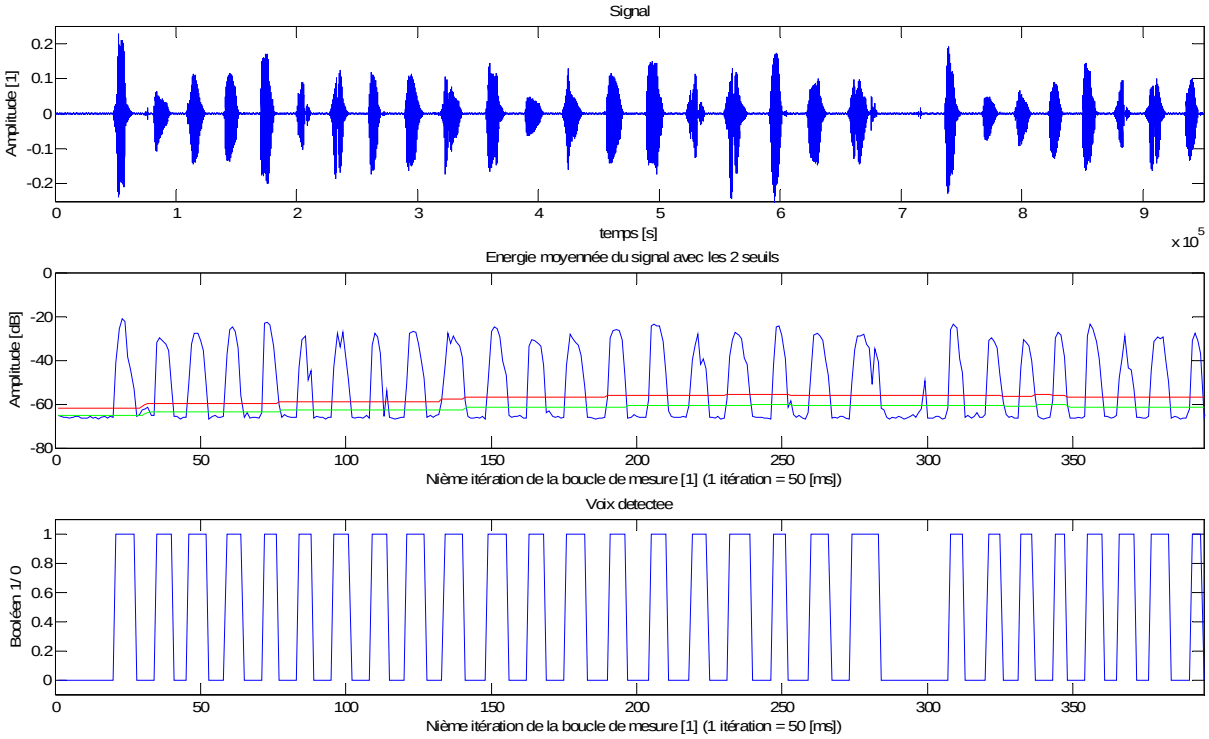


Figure 15 – Test du bon fonctionnement du VAD pour un SNR de 40 [dB] (Matlab)

SNR = 30 [dB] :

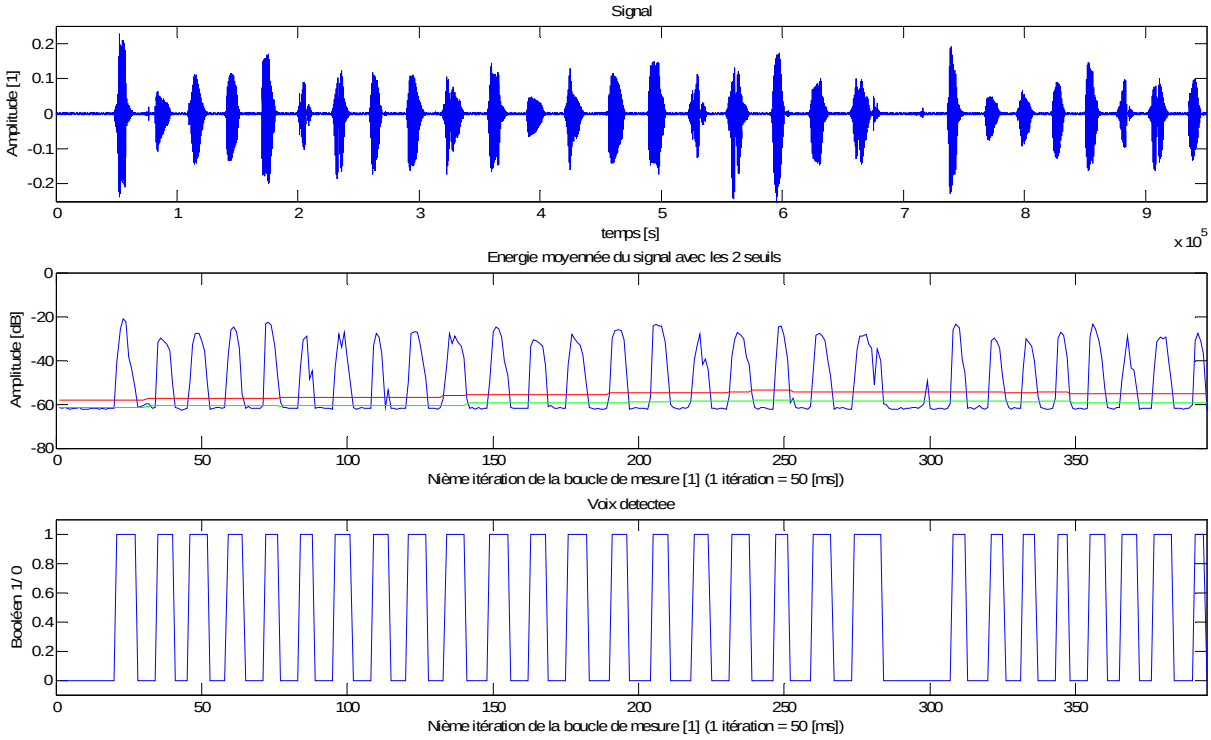


Figure 16 – Test du bon fonctionnement du VAD pour un SNR de 30 [dB] (Matlab)

SNR = 20 [dB] :

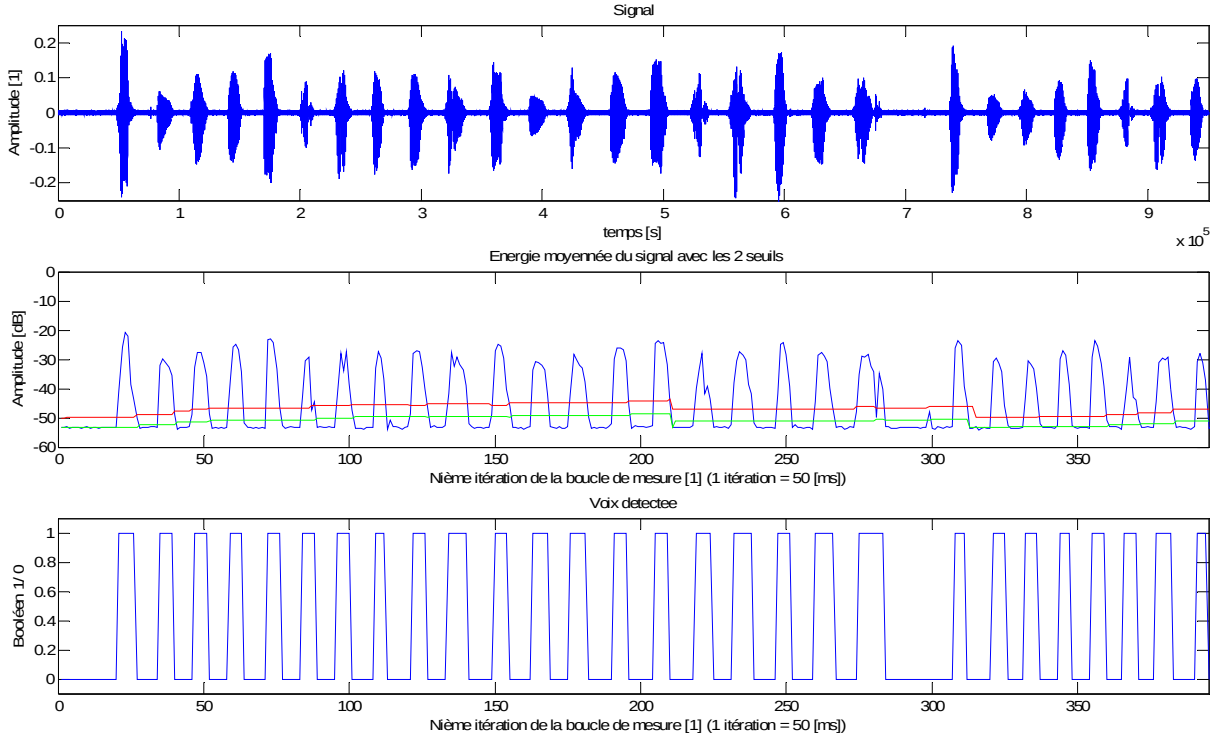


Figure 17 – Test du bon fonctionnement du VAD pour un SNR de 20 [dB] (Matlab)

SNR = 15 [dB] :

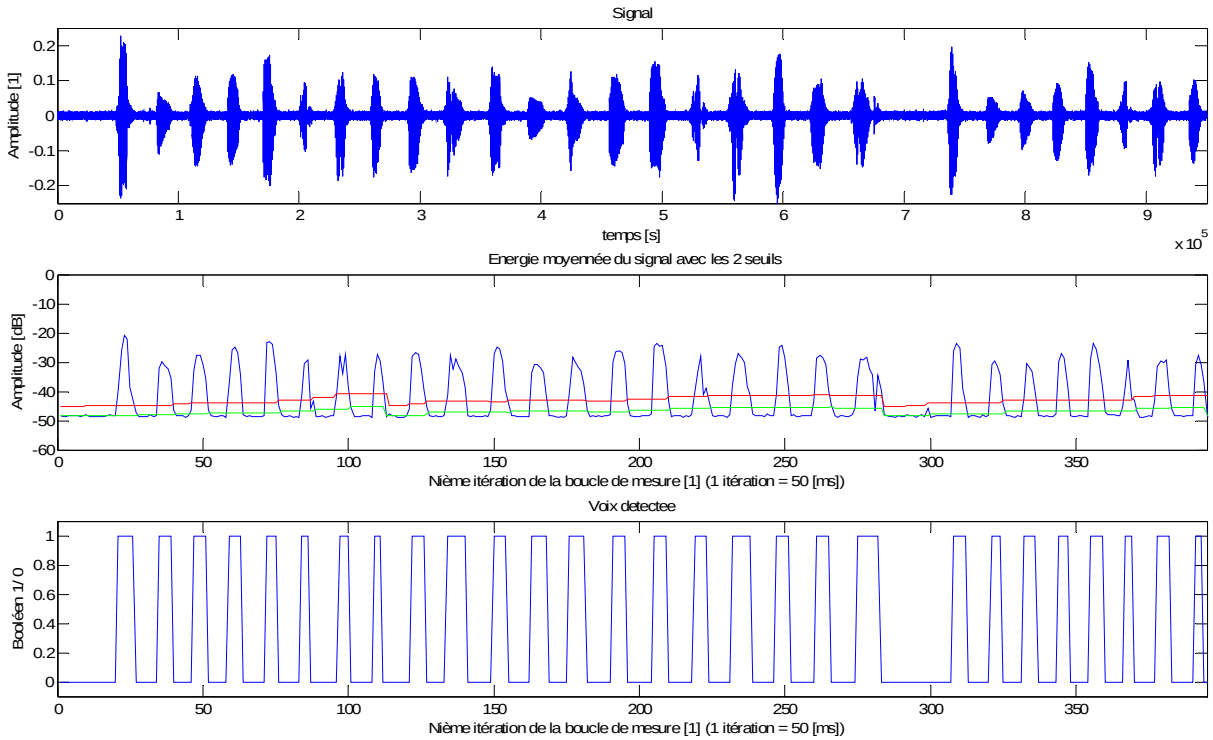


Figure 18 – Test du bon fonctionnement du VAD pour un SNR de 15 [dB] (Matlab)

SNR = 10 [dB] :

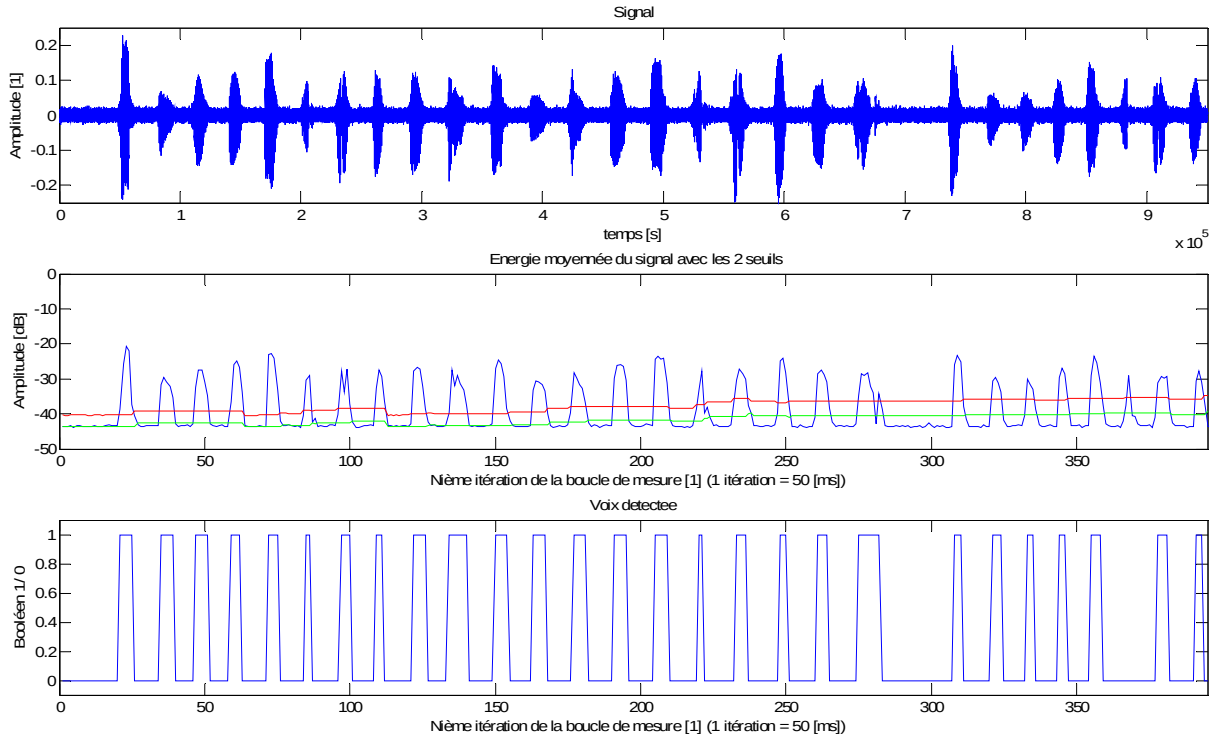


Figure 19 – Test du bon fonctionnement du VAD pour un SNR de 10 [dB] (Matlab)

SNR = 5 [dB] :

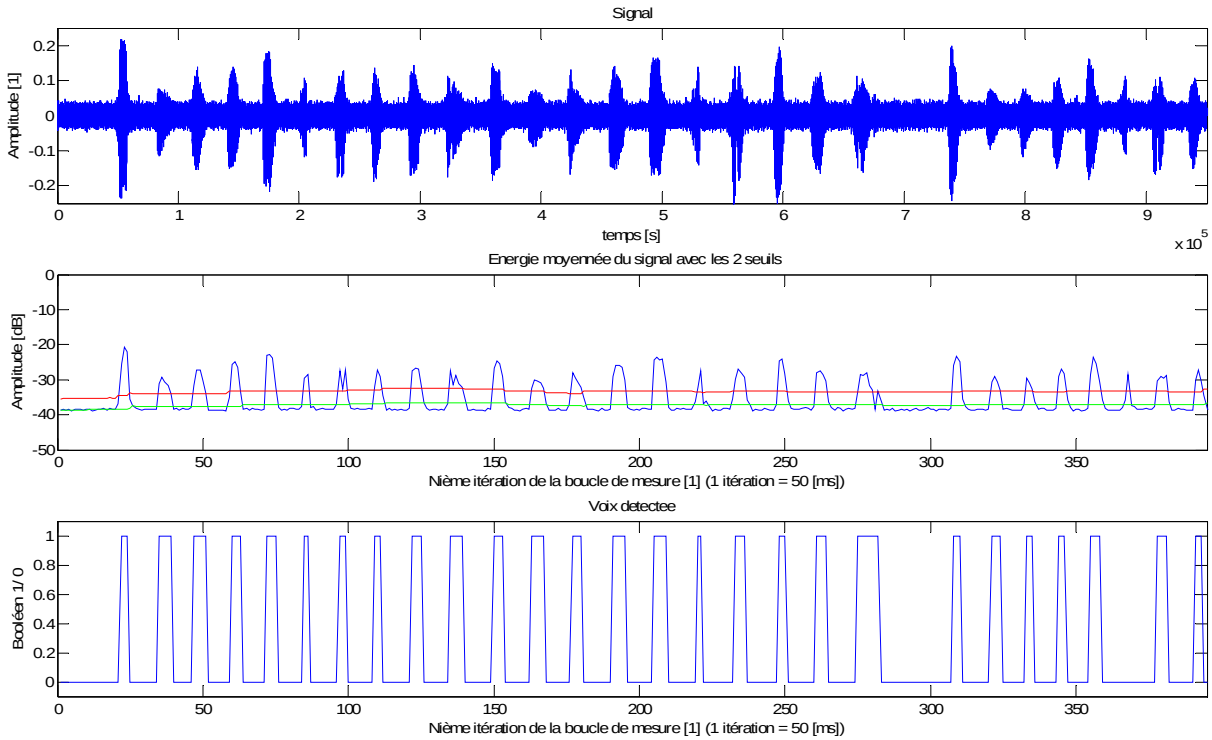


Figure 20 – Test du bon fonctionnement du VAD pour un SNR de 5 [dB] (Matlab)

Grâce à ces graphiques, on observe que plus le SNR diminue, plus les seuils de bruit et de parole augmentent, ce qui rend la détection de parole plus difficile pour le VAD. De ce fait, le VAD est plus lent à réagir au début de la prononciation d'un mot, car la phase d'apparition de ce dernier est noyée dans le bruit. Pour la phase d'extinction, le VAD réagit trop rapidement, car la fin du mot est également mélangée au bruit. Dès lors, la largeur des impulsions indiquant qu'un mot est détecté est plus courte.
Ce problème est perceptible à partir d'un SNR inférieur ou égal à 15 [dB]. On peut y remédier en indiquant au bloc calculant les coefficients cepstraux de tenir compte du même nombre d'échantillons situés avant et après la détection. Cependant, le bruit y sera très présent et le décodage risque de ne pas être objectif.

Dans les cas de faibles SNR (10 et 5 [dB]), le VAD n'a pas détecté le troisième mot avant la fin. Ceci est dû à deux facteurs. Le premier est le bruit qui est très présent à de faibles SNR et à une amplitude proche de celle du signal. Le deuxième est le lissage de la commande qui permet d'éliminer un éventuel parasite électrique. En effet, le niveau de ce mot dépasse pendant un instant trop court le seuil de détection de signal et est considéré comme un parasite. De ce fait, il est éliminé. Cependant, pour des SNR supérieurs ou égaux à 15 [dB], le lissage permet d'éliminer efficacement les parasites ne correspondant pas à du signal de parole, et par conséquent est indispensable à ce VAD.

Finalement, le VAD fonctionne correctement pour de forts et moyens SNR, mais il a plus de difficultés pour de faibles SNR.

6 Quantification vectorielle

Après avoir effectué l'extraction des paramètres *a(k)* (représentant les coefficients du filtre AR) par la méthode des coefficients LPC et les avoir convertis en cepstres *c(k*), on obtient pour chaque trame de 400 échantillons un vecteur contenant 13 composantes *c(k)* pertinentes ([*c(1) c(2) … c(13)*], sans *c(0)* qui est redondant et sans utilité).

Les coefficients cepstraux calculés subissent une classification non-supervisée, réalisée dans ce travail par une quantification vectorielle. Il existe diverses méthodes de recherche de classes en fonction de la proximité des vecteurs de *P* paramètres dans un espace à *P* dimensions.
Le but de ces méthodes est de réunir dans une même classe les vecteurs qui se ressemblent, et de mettre dans des classes différentes les vecteurs distincts. Dans le domaine mathématique, cela correspond à minimiser les distances intra-classe, tout en maximisant les distances entre les classes différentes.

La quantification vectorielle permet de résoudre ce problème. Les vecteurs sont représentés par un dictionnaire de *M* barycentres comportant chacun un numéro d'index : $\boldsymbol{Y} = \{\boldsymbol{y}i;\ 1 \leq i \leq M\ \}$, où *M* est le nombre de classes. Ce dictionnaire doit être fabriqué sur une base de données en minimisant une distance moyenne globale entre les vecteurs d'apprentissage et les mots du dictionnaire. Cet apprentissage est mis en œuvre par l'algorithme de Lloyd généralisé (K-means method) [1] en utilisant la distance Euclidienne entre les vecteurs.
L'ensemble des vecteurs d'apprentissage est constitué de coefficients cepstraux qui vont modifier et paramètriser les barycentres du dictionnaire initialisés aléatoirement.

Une fois le dictionnaire réalisé, on peut procéder à la quantification. Dans cette étape, on attribue à chaque vecteur de coefficients cepstraux le numéro (étiquette) de la classe qui lui est la plus proche.
De ce fait, seul l'index (position dans le dictionnaire) de ce vecteur défini sera à transmettre. Par conséquent, il y aura un gain sur le débit de transmission.

En d'autres termes, la quantification vectorielle est une opération qui permet de représenter un vecteur $\boldsymbol{c}$ formé de $\boldsymbol{k}$ composantes, par un vecteur $\boldsymbol{y}$ appartenant à un ensemble fini de $\boldsymbol{M}$ vecteurs (dictionnaire).

6.1 Méthode de Lloyd généralisée

Représentation graphique du principe de la quantification vectorielle :

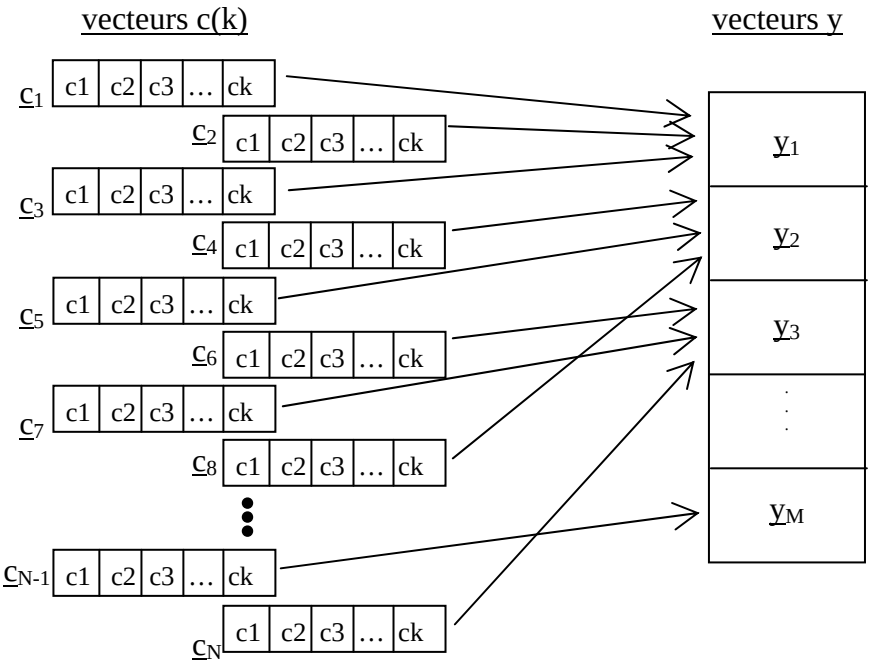


Figure 21 – Principe de la quantification vectorielle

On observe que les vecteurs $\underline{c_n}$ ayant des composantes ***c(k)*** proches correspondent à un vecteur $\underline{y_i}$ s'en rapprochant également et faisant parti du dictionnaire composé de ***M*** vecteurs.
Chaque vecteur $\underline{c_n}$ (contenu dans un même cluster ou classe) sera substitué par le vecteur $\underline{y_i}$ correspondant. Cette substitution va provoquer une distorsion locale notée ***d(c, y)***, aussi appelée distance euclidienne (erreur quadratique) :

$$d(c,y) = \frac{1}{k} \cdot \sum_{i=1}^{k} (c_i - y_i)^2$$

La distorsion D_i de la classe **i** due à la quantification vectorielle vaut :

$$D_i = \sum_j d(c_j^{(i)}, y_i)$$

Finalement, la distorsion ***D*** pour l'ensemble des classes vaut :

$$D = \sum_{i=1}^{M} D_i$$

6.2 Détail de l'algorithme K-means

Le centroïde (ou barycentre) d'un cluster sera calculé de manière à ce que cette distorsion totale **D** soit minimum. Voici une procédure itérative permettant de fixer le barycentre de chaque cluster de façon à minimiser la distorsion totale **D** :

1. On définit le nombre M de classes du système.

2. On fait un choix a priori de M centroïdes (chacun placé aléatoirement dans un cluster).

3. On affecte l'ensemble des vecteurs $\underline{\mathbf{c}}_\mathbf{n}$ aux diverses classes. Pour ce faire, on calcule toutes les distances $\mathbf{d(c_j, y_i)}$ avec $\mathbf{1 \leq j \leq N}$ et $\mathbf{1 \leq i \leq M}$, et on associe à $\mathbf{c_j}$ la classe numéro **i** du centroïde $\mathbf{y_i}$ le plus proche.

4. On recalcule la position de chaque barycentre $\mathbf{y_i}$ pour minimiser chaque distorsion $\mathbf{D_i}$.

5. On calcule la distorsion totale **D**.

6. On itère en revenant au point 3, jusqu'à ce que **D** varie de moins de $\boldsymbol{\varepsilon}$ [%] d'une itération à la suivante, avec $\boldsymbol{\varepsilon}$ = erreur quadratique.

6.3 Simulation et validation du bloc de quantification vectorielle

Il est plus que difficile de représenter graphiquement un espace à 13 dimensions quantifié vectoriellement. De ce fait, les résultats obtenus dans cette validation sont donnés par un système à 3 dimensions. Celui-ci vérifie également un système à 13 dimensions en ce sens que seul une variable (celle définissant le nombre de coefficients cepstraux) change dans le code qui est paramétrable.

Sur les graphiques qui suivent, les points bleus représentent des valeurs composées de 3 coefficients (X, Y, Z). Les cercles rouges représentent les barycentres permettant d'identifier le centre de gravité des diverses classes.
Il est à noter que pour cette validation, 14 barycentres ont été positionné aléatoirement dans l'espace selon une distribution Gaussienne. De plus, il n'y a que 10 classes à identifier. Le fait de prendre plus de barycentres que de classes, permet de s'assurer que toutes les classes seront représentées par un centroïde. Les barycentres qui ne sont pas rattachés à une classe lors de l'exécution de l'algorithme auront des coordonnées nulles (0, 0, 0) et seront éliminés à la fin de l'exécution de l'algorithme.

Répartition des divers points et barycentres initialisés dans l'espace :

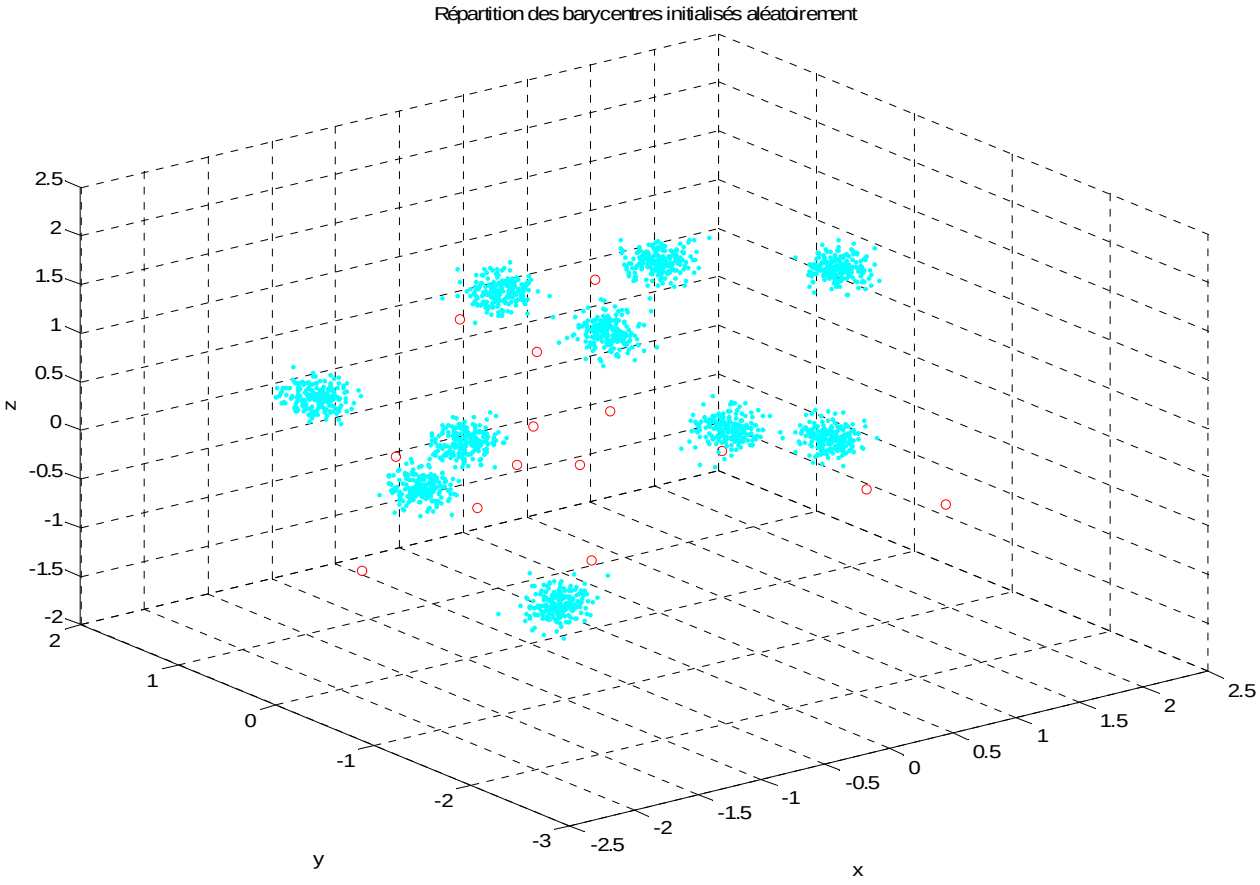


Figure 22 – Répartition aléatoire des barycentres dans l'espace lors de l'itération de départ (Matlab)

On observe bien qu'il n'y a aucun lien entre les divers nuages de points et tous les centroïdes.

Répartition des barycentres centrés dans les diverses classes :

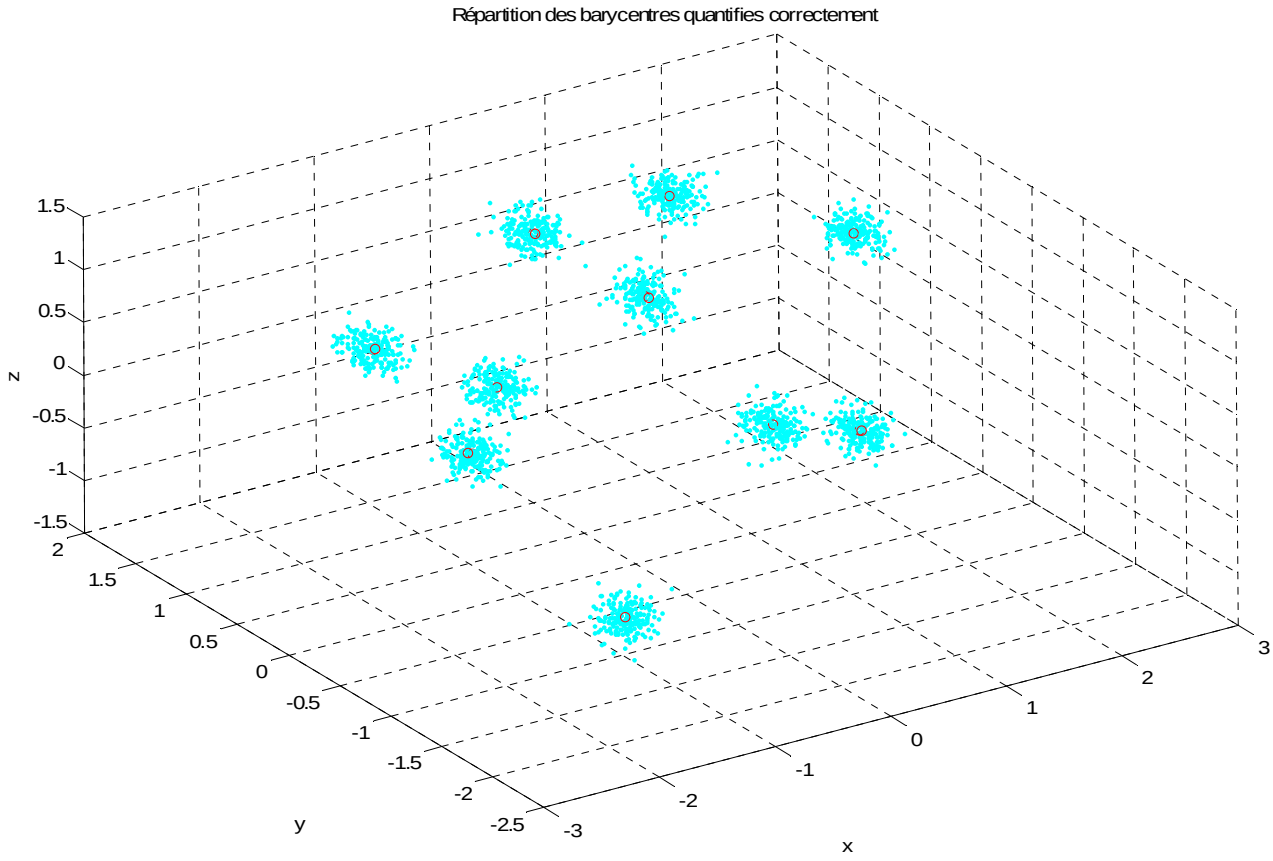


Figure 23 – Répartition finale des barycentres dans l'espace

On remarque que les barycentres se sont correctement positionnés au centre de chacun des nuages de points (clusters).
Ce résultat à été obtenu à partir d'un certain nombre d'itérations sur l'algorithme. Un graphique représentant la distorsion totale en fonction du nombre d'itération permet de savoir au bout de combien de passages les coordonnées des centroïdes ne seront plus modifiées.

Mesure de la distorsion totale, soit la somme de toutes les distorsions internes à chaque classe en fonction du nombre d'itérations de l'algorithme de quantification vectorielle :

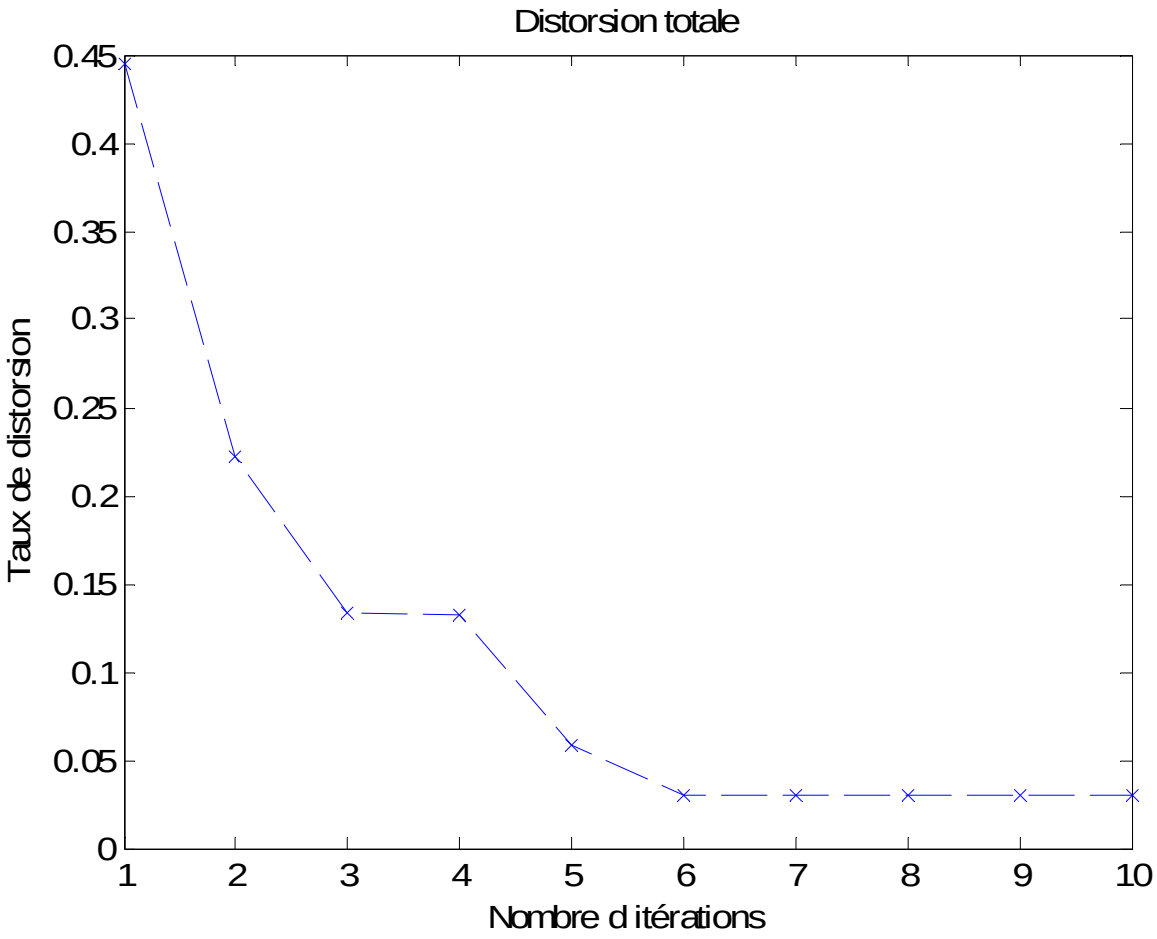


Figure 24 – Mesure de la distorsion totale

On observe qu'à partir de la 6ème itération, la position des centroïdes ne peut plus être améliorée, et on obtient un taux de distorsion de 0.03, ce qui est tout à fait satisfaisant.

6.4 Application de la quantification vectorielle à un espace à 13 dimensions

A partir de cet algorithme de quantification vectorielle, il va falloir établir le dictionnaire de M centroïdes $c(k)$ avec $1 \leq k \leq 13$. Pour ce faire, on sélectionne tous les vecteurs de cepstres générés par HTK lors de la phase d'entraînement, ce qui correspond à 16000 vecteurs pour 80 mots (sachant qu'il y a environ 200 vecteurs par mot). Puis, on les fait passer dans le bloc de quantification vectorielle.

Après une dizaine d'itérations, on obtient un dictionnaire de 724 vecteurs cepstraux minimisant la distorsion entre toutes les classes et représentant correctement les 16000 vecteurs initiaux.

7 Chaîne de Markov cachée (HMM)

Une chaîne de Markov [3] est une machine à états, la transition d'un état à un autre obéissant à une loi probabiliste. On parlera par la suite de matrice de transition d'états pour décrire la causalité probabiliste entre les états. A chaque état, est associée une loi probabiliste qui rend compte de la probabilité d'observer (dans cet état) un vecteur d'observation quelconque. A la fin de chaque tranche de temps de durée constante (horloge d'un système), le pointeur d'état de la chaîne change de position. La particularité d'une chaîne de Markov cachée et que l'on ne sait à aucun moment dans quel état on se trouve (sauf au temps $t = 0$ et $t = T$), de ce fait l'état est caché. On ne connaît l'état qu'à posteriori en utilisant une optimisation mathématique maximisant la vraisemblance de la succession d'états pour une séquence d'observations donnée.
Le temps actuel t correspondant à un moment bien précis dans la chaîne. Il nous permet de définir q_t qui est l'état actuel où l'on se situe au temps $t \{0, 1, 2, 3, \ldots, T\}$.

Les probabilités permettant de passer d'un état à un autre sont appelées les probabilités de transition et sont définies comme-suit : $\boldsymbol{a_{ij} = p(q_t = j \mid q_{t-1} = i)}$ avec a_{ij} la probabilité de transition d'un état vers tous les autres états.
Toutes les probabilités sortant du même état ont une somme totale de 1

$$\Rightarrow \sum_{j=1}^{N} aij = 1 .$$

La matrice $A = \{a_{ij}\}$ est la matrice de probabilité de transition entre les états avec la somme des probabilités d'une ligne égale à 1 :

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \ldots & a_{1N} \\ \ldots & \ldots & \ldots \\ a_{N1} & \ldots & a_{NN} \end{bmatrix}$$

Une succession d'états rend compte avec une certaine probabilité d'une suite d'événements observables (qui sont dans notre cas des vecteurs de coefficients cepstraux). On doit donc, en regard d'une suite d'observations, trouver la succession d'états qui a la plus grande probabilité (la meilleure vraisemblance) de s'être effectivement déroulée. De ce fait, la séquence d'observations O induit la suite des états. Par conséquent, calculer la vraisemblance qu'un modèle Markovien ait produit une certaine séquence d'observations O reviendra à trouver la meilleure séquence d'états qui maximise la probabilité d'une séquence d'observations donnée, ceci par rapport à un modèle de Markov défini lors d'une phase d'entraînement. On décrit cette probabilité par : ***p(O | Model Markovien)***. Lorsque l'on aura plusieurs mots à reconnaître, on aura par exemple un modèle Markovien par mot.

La reconnaissance d'un mot se fera en mettant en compétition tous les modèles Markoviens des différents mots. Le modèle markovien qui aura la plus grande probabilité ***p(O | Model Markovien)*** sera le vainqueur, et le mot sera reconnu par le modèle Markovien qui lui est associé.
On fait pour le moment l'hypothèse que le modèle de Markov est connu. Cependant, il va falloir l'entraîner lors de la phase d'apprentissage avec l'algorithme de Baum-Welch (ceci est expliqué au chapitre 9).

La probabilité de l'état initial par lequel on commence la chaîne est $\pi_i = p(q_1 = i)$ avec $1 \leq i \leq N$.
Pour les HMMs, l'observation est une fonction probabiliste de l'état. Il s'agit d'un processus stochastique embarqué dans un autre processus aléatoire et qui y est caché à l'intérieur. Ce processus peut être observé seulement à travers un autre ensemble de processus stochastiques qui produisent la séquence d'observations (la série de vecteurs cepstraux) reçue, soit : $O = (O_1, O_2, \ldots, O_T)$.

De ce fait, on peut représenter un automate de Markov comme suit :

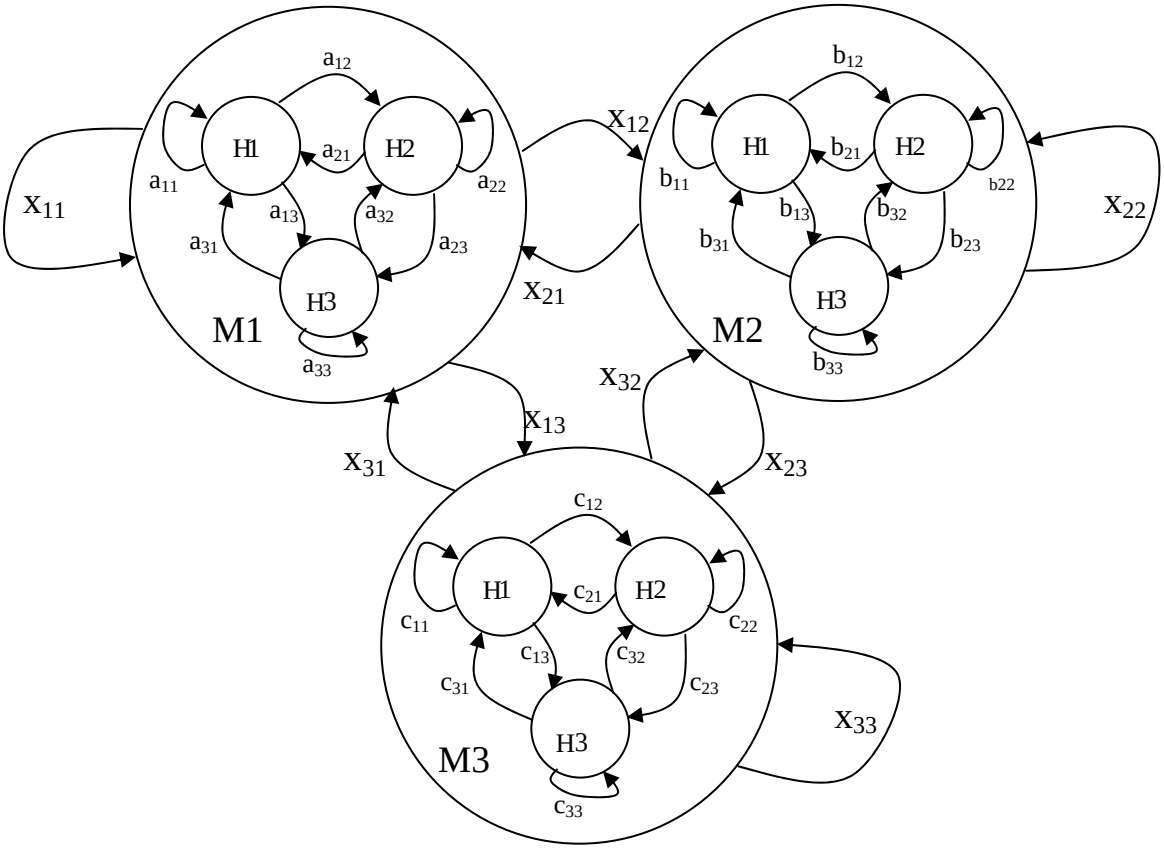


Figure 25 – Schéma d'une chaîne de Markov

Une HMM est caractérisée par :

1. Le nombre d'état N du model et l'état q_t au temps t.
2. Le nombre de symboles d'observations distinctes S par état, soit la sortie d'un modèle Markovien. Avec les symboles $V = \{v_1, v_2, \ldots, v_S\}$.
3. La distribution des probabilités de transition de la matrice A, avec $\boldsymbol{a_{ij} = p(q_{t+1} = j \mid q_t = i)}$ *et* $\boldsymbol{1 \leq i, j \leq N}$
4. La distribution de probabilité de symboles d'observations $B = \{b_j(k)\}$ avec $\boldsymbol{b_j(k) = p(O_t = v_k \mid q_t = j)}$ *et* $\boldsymbol{1 \leq k \leq S}$
5. La distribution des états initiaux $\pi = \{\pi_i\}$ avec $\boldsymbol{\pi_i = p(q_1 = i)}$

8 Application des HMM à la reconnaissance de la parole

8.1 Modèles de phonèmes

Chaque phonème (mono phonème) est représenté par une succession d'états (trois en principe). Chacun de ces phonèmes est composé d'un modèle de Markov défini (qui a la même structure pour tous les phonèmes). Le modèle de Markov caché adapté à la parole est le modèle « HMM à trois états de gauche à droite » :

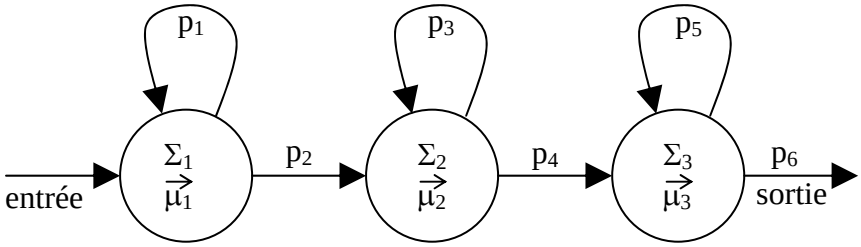


Figure 26 – HMM à trois états de gauche à droite

Les trois états modélisent le comportement de la prononciation d'un phonème, soit :

- le premier correspond à la phase d'attaque du phonème,
- le deuxième correspond à la phase de maintien du phonème,
- le troisième correspond à la phase d'extinction du phonème.

Si l'on reprend la figure 25 montrant le schéma d'une chaîne de Markov, on s'aperçoit que les grands états (M1, M2, M3) représentent les phonèmes, et que les états internes (H1, H2, H3) représentent le modèle HMM à trois états de gauche à droite.

Chacun des états internes H est caractérisé par une matrice de covariance Σ et un vecteur de moyenne $\vec{\mu}$, faisant référence aux probabilités d'observations d'un vecteur de cepstres reçu.

Les probabilités de passages entre états sont régies par la matrice de transition :

$$\begin{array}{l} \\ \text{états en entrée} \end{array} \begin{array}{c} \begin{array}{ccccc} \text{états en sortie : } 2 & 3 & 4 & \text{sortie} & \end{array} \\ \left\{\begin{array}{c} \text{entrée} \\ 2 \\ 3 \\ 4 \\ \end{array}\right. \begin{bmatrix} \boldsymbol{0} & \boldsymbol{1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{p_1} & \boldsymbol{p_2} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{p_3} & \boldsymbol{p_4} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{p_5} & \boldsymbol{p_6} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \end{array} = \boldsymbol{A}$$

Afin d'utiliser ce modèle pour réaliser une reconnaissance de la parole, un entraînement de l'automate sera nécessaire afin de pouvoir calculer la matrice de transition *A* et les vecteurs de variance et de moyenne. Ensuite, le modèle pourra être utilisé en phase de reconnaissance.
Afin de reconnaître les dix premiers chiffres en anglais, il faut définir 21 modèles de phonèmes distincts, soit :

- ONE w ah n
- TWO t uw
- THREE th r iy
- FOUR f ao
- FIVE f ay v
- SIX s ih k s
- SEVEN s eh v n
- EIGHT ey t
- NINE n ay n
- ZERO z ia r ow
- SILENCE sil

A droite, on observe la décomposition des dix mots (plus le modèle caractérisant un silence) par leurs phonèmes respectifs. Chacun de ces phonèmes est représenté par une HMM à trois états de gauche à droite.

Une fois que tous les modèles de phonèmes sont entraînés à partir d'une base de données, il va falloir concaténer certains phonèmes de manière à créer les mots du dictionnaire de reconnaissance.

Remarque : la base de données est composée de chaque mot prononcé huit fois par moi. Ce qui donne une petite base de données de 80 mots.

Voici la représentation du modèle obtenu pour le chiffre « FIVE » à partir de ces phonèmes respectifs :

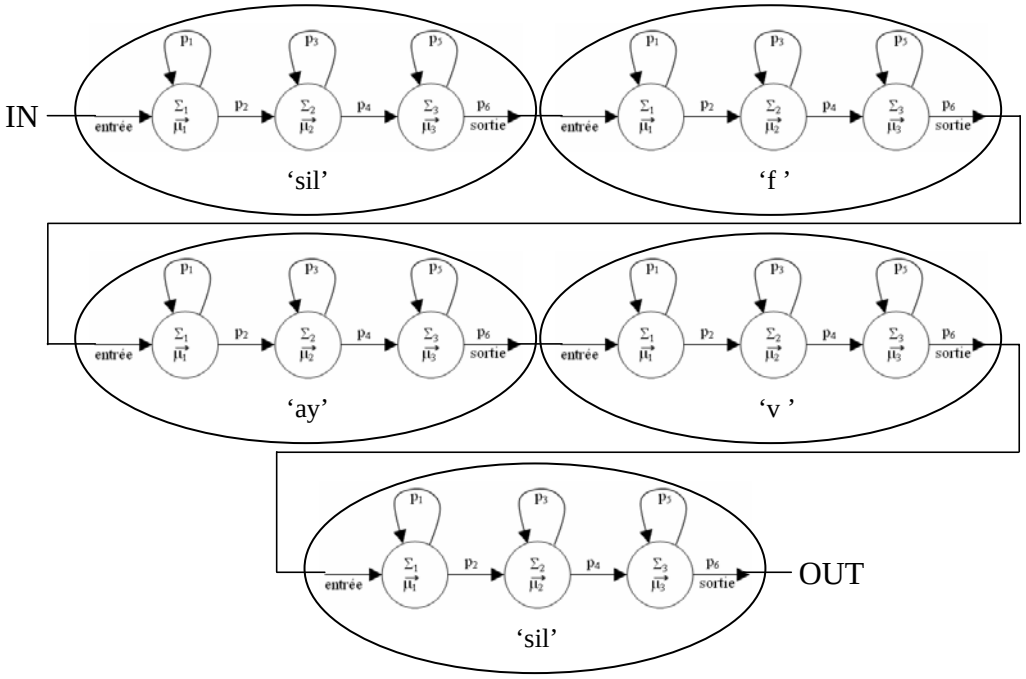


Figure 27 – Composition des phonèmes réalisant le modèle HMM pour le mot « FIVE »

Dans ce cas, le problème qui apparaît est de définir les valeurs des probabilités de transition entre deux modèles de phonèmes successifs.
Une des solutions consiste à fixer ces probabilités à 0.5. De ce fait, la probabilité de boucler sur l'état de sortie d'un modèle de phonème est également de 0.5, car n'oublions pas que la somme des probabilités sortant d'un état doit être égale à 1.

Les chiffres du dictionnaire identifiés par des modèles de phonèmes donnent des chaînes de Markov à longueurs variables. En effet, comme vu à la page précédente, on peut avoir 2, 3 ou 4 phonèmes permettant de caractériser les dix chiffres.
Cependant, ceci n'affecte en rien le décodage avec Viterbi (voir au chapitre 10).

8.2 Modèles de mots

Dans le cas des modèles de mots, on définit directement un modèle de Markov à *N* états, comme une entité unique représentant directement un mot. De ce fait, on ne fait plus la distinction entre les phonèmes, mais entre les mots.
Afin de reconnaître les dix premiers chiffres en anglais, il faut définir dix modèles de mots distincts, soit :

- ONE one
- TWO two
- THREE three
- FOUR four
- FIVE five
- SIX six
- SEVEN seven
- EIGHT eight
- NINE nine
- ZERO zero

Contrairement aux modèles de phonèmes, on n'a pas besoin pour les modèles de mots, de devoir concaténer des modèles ensemble afin de créer une entité, elle existe déjà.
Cependant, une HMM à trois états de gauche à droite n'est plus suffisante afin de caractériser directement un mot. De ce fait, il faut créer un modèle de mot avec plus de trois états.
En observant les mots que le système doit pouvoir reconnaître, on s'aperçoit qu'ils contiennent entre deux et quatre phonèmes chacun. Dès lors, la valeur moyenne est de trois phonèmes. Puisque chacun d'eux est composé de trois états, ceci nous amène à créer des modèles de mots composés de neuf états de gauche à droite.

Voici la représentation du modèle de mot obtenu pour le chiffre « FIVE » :

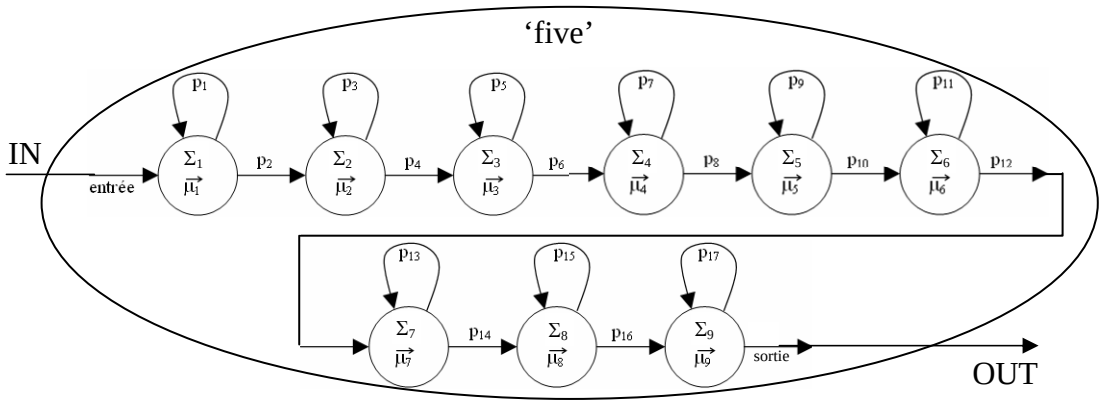


Figure 28 – Composition du modèle de mot HMM pour le mot « FIVE »

Dans ce cas, tous les modèles de mots ont une longueur fixe de neuf états.
Pour l'implémentation Matlab et DSP, on a choisi d'utiliser les modèles de mots.

9 Apprentissage avec HTK

9.1 Etiquetage

Afin de permettre l'estimation des probabilités d'observation, les paramètres de covariance Σ et de moyenne $\vec{\mu}$ de chaque état du modèle de Markov doivent être spécifiés (selon une loi de probabilité normale), ainsi que la matrice des probabilités de transitions entre états. Ces spécifications sont stockées dans un fichier contenant les divers modèles de Markov (de phonèmes ou de mots).

Il existe deux types d'entraînements distincts : **supervisé** (base d'entraînement étiquetée) et **non supervisé** (base d'entraînement non étiquetée).
L'entraînement supervisé permet d'associer le son d'un phonème ou d'un mot avec le label qui le représente. Cette technique est utilisée par HTK et est très pratique pour un dictionnaire contenant peu de mot.

Voici un exemple d'étiquetage des phonèmes composant le mot « zero » en anglais :

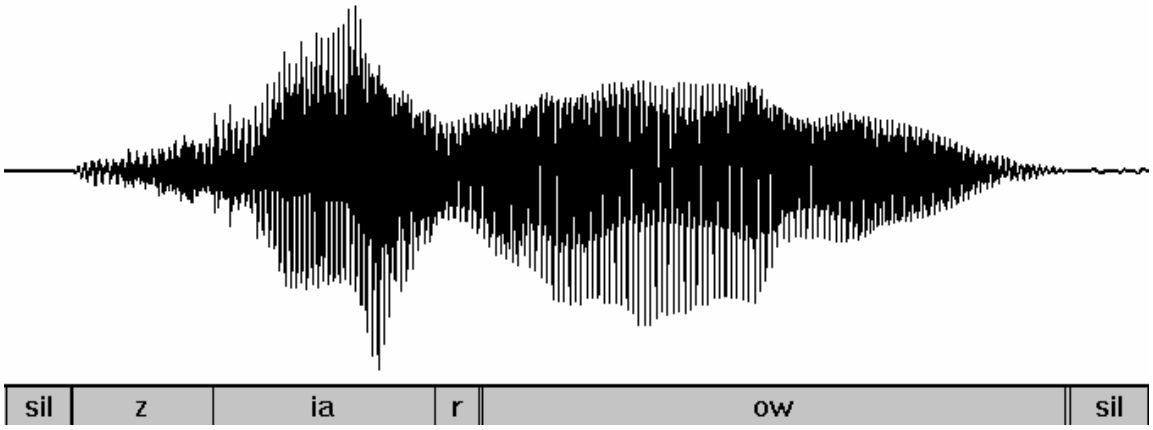


Figure 29 – Etiquetage phonétique du mot « zero » en anglais (HTK et Matlab)

Sur cette figure on observe bien les différents labels correspondants à chacun des phonèmes distincts.

Voici un exemple d'étiquetage de mot composant le mot « zero » en anglais :

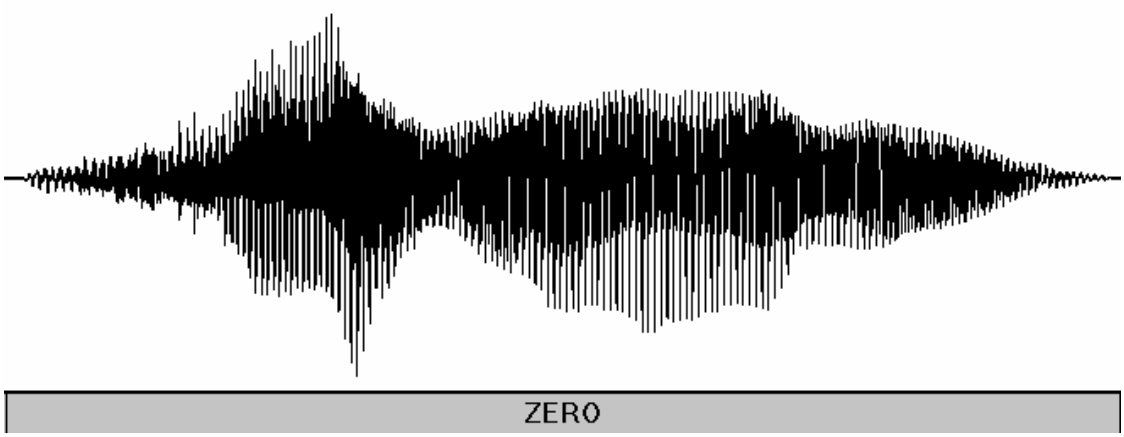


Figure 30 – Etiquetage du mot « zero » en anglais (HTK et Matlab)

Cette fois-ci, le mot « zero » est une entité à lui tout seul et n'est pas subdivisé en phonèmes.

On a décidé d'utiliser les modèles de mots, car ils sont plus appropriés à la reconnaissance de mots isolés d'un petit dictionnaire.

9.2 Algorithme de Baum-Welch

Dans le chapitre 8, le modèle « prototype » de HMM a été défini. Maintenant, il va falloir l'entraîner en fonction de l'étiquetage précédent des divers mots du dictionnaire, et avec l'algorithme d'entraînement de Baum-Welch [2] (également appelé entraînement avant-arrière) qui résulte d'un processus en trois étapes.

Voici les trois étapes simplifiées de la procédure d'entraînement :

1. **Etape d'estimation** : en partant d'un ensemble de paramètres Θ^t donnés à l'itération t, on applique les récurrences avant et arrière.
   La fonction **avant** $\boldsymbol{\alpha(t,q_i)}$ représente la probabilité d'observer les t premières observations et d'être à l'instant t dans l'état q_i.
   La fonction **arrière** $\boldsymbol{\beta(t,q_i)}$ représente la probabilité d'observer les T-t dernières observations sachant que l'on est à l'instant t dans l'état q_i.

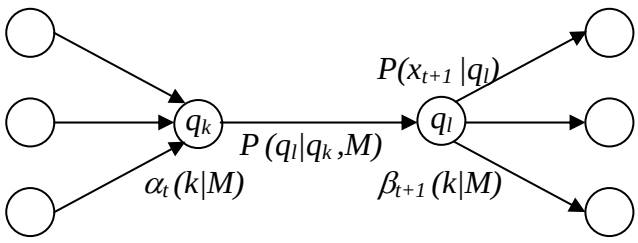


Figure 31 – Visualisation de la récurrence avant et arrière [1]

En multipliant ces deux fonctions, cela permet de calculer la probabilité que le modèle *M* ait généré la totalité de la séquence d'observation *O* tout en étant passé par l'état particulier q_l à l'instant *t* : $p\left(q_l^t, O \mid M, \Theta^t\right)$.

Cet estimateur peut alors être utilisé afin de calculer $p\left(q_l^t \mid O, M, \Theta^t\right)$, qui a son tour, donne un estimateur de l'espérance mathématique des fréquences relatives des transitions et émissions (probabilité à posteriori des variables cachées).

2. **Etape de maximisation** : mise à jour des paramètres de façon à maximiser la fonction de vraisemblance *Q*.

3. **Itération** : aussi longtemps que *Q* continue à augmenter, on passe à l'instant *t* suivant, soit *t+1*, et on retourne à l'étape d'estimation.

Le but de ce travail de diplôme n'étant pas la réalisation de la phase d'entraînement, l'algorithme de Baum-Welch ne sera pas plus développé. En effet, ceci prendrait beaucoup trop de temps. Dès lors, on utilise directement HTK afin d'entraîner les dix modèles de mots.

Pour la réalisation de l'apprentissage sur HTK, on initialise toutes les composantes de la matrice des covariances et des vecteurs de moyennes (paramètres de chacun des états du modèle) respectivement à 1 et 0. Suite à la séance d'apprentissage de Baum-Welch qui consiste à définir les probabilités de transitions entre états et les paramètres utiles à calculer les probabilités d'observations, il est possible de définir les dix **modèles Markoviens *M*** à utiliser.

9.3 Définition des modèles

Après l'apprentissage avec l'algorithme de Baum-Welch, les matrices des probabilités de transitions sont directement définies et peuvent être immédiatement utilisées.
Cependant, il faut construire les matrices des probabilités d'observations pour chaque modèle. Ceci est fait en fonction des vecteurs de moyennes et de la matrice de covariance de chaque état du modèle, et en fonction des 724 vecteurs cepstraux définissant les barycentres du dictionnaire de quantification vectorielle.

Sous HTK, les vecteurs de moyennes et les matrices de covariances de chaque modèle sont définis selon une loi de probabilité Gaussienne. Dès lors, il est possible de déterminer les matrices des probabilités d'observations par la formule suivante :

$$b_j(O_t) = \sum_{k=1}^{13} N(o_k; \mu_k, \Sigma_k)$$

Dans laquelle, la fonction $N(o_k; \mu_k, \Sigma_k)$ représente la loi de probabilité normale ou Gaussienne qui est définie comme suit :

$$N(o;\mu,\Sigma) = \frac{1}{\sqrt{(2\cdot\pi)^n \cdot |\Sigma|}} \cdot e^{-\frac{1}{2}(o-\mu)'\cdot\Sigma^{-1}\cdot(o-\mu)}$$

avec μ = vecteur de moyennes d'un état ; Σ = matrice de covariance d'un état ;
o = vecteur de cepstres observés ; $bj(Ot)$ = probabilité d'observation.

La fonction à implémenter sous Matlab qui permet de calculer ces probabilités d'observations n'est pas compliquée à réaliser, car la matrice de covariance a une particularité très intéressante.
En effet, seules les composantes de sa diagonale contiennent de l'information, soit les 13 variances correspondant aux 13 coefficients cepstraux. Toutes les autres valeurs de la matrice sont nulles. Dès lors, au lieu de faire des calculs redondants et inutiles avec cette matrice de covariance, on définit un vecteur de variances qui contient les valeurs de la diagonale de la matrice de covariance.
De plus, le facteur de multiplication n'est qu'une constante qui multiplie l'exponentielle et n'apporte pas d'informations pertinentes, et peut-être éliminé du calcul de la probabilité d'observation.

Maintenant que l'on possède les probabilités de transitions et d'observations, les dix modèles de HMM peuvent être entièrement définis. On obtient des modèles contenant une matrice 11x11 pour les probabilités de transitions entre états et une matrice 724x9 pour les probabilités d'observations correspondant aux 724 barycentres et aux neuf états d'un automate de Markov.

10 Reconnaissance (algorithme de Viterbi)

Le but de l'algorithme de Viterbi [2] est de trouver le meilleur chemin, soit la meilleure séquence d'état $q = (q_1\ q_2\ \ldots\ q_T)$ correspondant aux diverses séquences d'observations $O = (O_1, O_2, \ldots, O_T)$ au travers du modèle Markovien défini lors de l'entraînement.
De ce fait, cet algorithme dévoile la segmentation optimale (selon le modèle) en unités linguistiques élémentaires et calcule la vraisemblance d'un modèle par rapport à une séquence d'observation (vecteurs de cepstres).

10.1 Description de l'algorithme

L'algorithme est composé de deux phases distinctes :

- le calcul des probabilités de chaque état q_n
- le recouvrement du chemin total en partant du dernier état qui est connu q_T

Pour trouver le meilleur chemin afin d'accéder à l'état *j* à l'instant *t*, il faut déterminer quel est le chemin qui est le plus vraisemblable. La plus grande probabilité déterminant l'état par lequel on définit partiellement le chemin est :

$$\phi_j(t) = \max_i\{\phi_i(t-1)\cdot a_{ij}\}\cdot b_j(O_t)$$

Afin de retrouver la séquence idéale, on doit garder la trace des arguments calculés ci-dessus. Pour ce faire on les ordonne dans un tableau : $\psi_t(i)$. Voici la procédure permettant de trouver la séquence la plus probable :

1. Initialisation de la matrice de transition des états (définissant plusieurs chemins possibles) : $\phi_1(1) = 1$ et $\phi_j(1) = a_{1j}\cdot b_j(O_1)$, et du tableau permettant de retrouver la séquence d'état, soit $\psi_t(i) = 0$ pour tout i.

2. Log des diverses probabilités ($b_j(O_t)$, a_{ij}) pour éviter les multiplications lors de l'implémentation.

3. Récursion sur le calcul de l'état q le plus probable pour toutes les séquences d'observations (ceci est vrai pour les modèles continus) :
$$\phi_j(t) = \max_{1\le j\le N}\left\{\log\left(\phi_j(t-1)\right)+\log\left(a_{ij}\right)\right\}+\log\left(b_j(O_t)\right)$$

4. Insertion dans le tableau $\psi_j(t)$ de chaque valeur max calculée : $\psi_j(t) = \max_{1\le j\le N}\phi_j(t)$.

5. Détection de l'état final, soit du point d'arrivée du chemin et de sa probabilité : $q(T) = \psi_i(T)$ et $p = \max_{1\le i\le N}\left\{\log\left(\phi_j(T)\right)+\log\left(a_{ij}\right)\right\}$.

6. Recouvrement de la séquence d'état totale (back tracking), soit du chemin complet : $q(t) = \psi_{t+1}(q(t+1))$.

Si l'on définit une matrice ou les lignes représentent les états et les colonnes représentent les trames de parole, l'algorithme de Viterbi peut être visualisé comme l'outil permettant de trouver le meilleur chemin au travers de cette matrice.

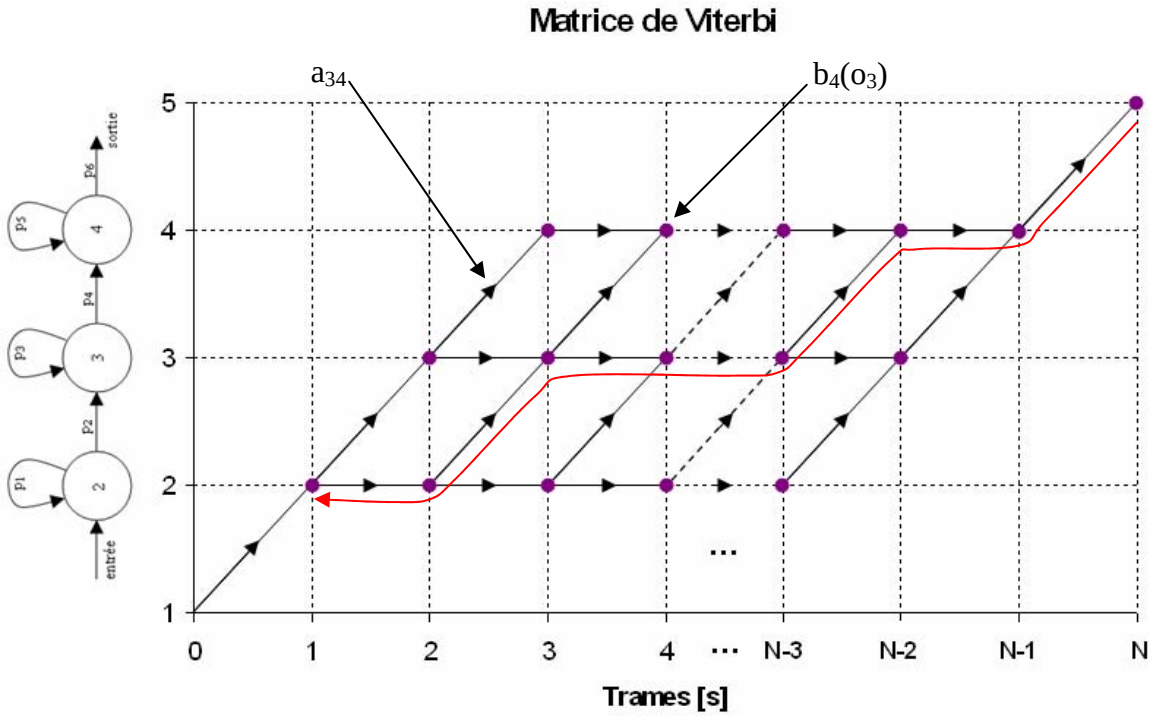


Figure 32 – Recouvrement du chemin optimal au travers de la matrice de Viterbi

Lors de la première phase de l'algorithme, on entre dans la matrice au point (1, 2) et on la parcourt de gauche à droite (flèches noires). A chaque fois que l'on arrive à un nouveau point, et que l'on peut accéder à ce dernier par plusieurs chemins, on élimine tous les chemins sous-optimaux permettant d'y arriver.
De ce fait, tous les chemins ayant une probabilité plus faible sont éliminés. Dès que l'on sort au point (N, 5), le dernier état *q* est déterminé, et la deuxième phase peut commencer.
A cet instant, on parcourt la matrice en sens inverse (flèche rouge) en repassant par tous les points qui ont permis de maximiser la probabilité du meilleur chemin. Dès lors, tous les états intermédiaires sont connus.

10.2 Validation de l'algorithme de Viterbi

Afin de vérifier le bon fonctionnement de la réalisation de l'algorithme de Viterbi, on a créé une routine qui permet de générer une série de cepstres synthétiques selon des lois de génération utilisant un Modèle Markovien avec une matrice de probabilités de transitions connue et une loi de probabilité d'une observation dans chaque état elle aussi connue. On observe qu'un mot (un des dix chiffres du dictionnaire) échantillonné à 16 [kHz] avec une fenêtre de stationnarité prise sur 10 [ms] contient environ 200 cepstres. Dès lors, la routine génère 200 cepstres synthétiques.

10.2.1 Principe de fonctionnement de la routine générant des cepstres synthétiques

Le but de cette fonction est de générer une série de cepstres en se basant sur les modèles de mots obtenus lors de la phase d'entraînement.

Détail de l'algorithme :

1. On sélectionne le modèle de mot que l'on souhaite utiliser afin de calculer les cepstres synthétiques.

2. On entre dans le système à l'état '1' au temps t = 0. On passe directement à l'état '2' au temps t = 1, car la probabilité de transition entre l'entrée (état 1) et le premier état du modèle (état 2) vaut 1.

3. On sélectionne les 13 moyennes et variances de l'état courant correspondant au modèle choisi. Ensuite, on génère 13 nombres aléatoires (correspondants aux 13 coefficients cepstraux d'un vecteur de cepstres) selon une distribution normale à moyenne = 0 et variance = 1. Puis, on modifie ces nombres, en leurs appliquant les moyennes et variances respectives de l'état courant.
   Dès lors, on peut enregistrer ce vecteur de cepstres dans le tableau de sortie.

Ensuite, il faut générer les autres vecteurs de cepstres en fonction de l'état dans lequel on se trouve, sachant que l'on peut boucler sur le même état ou passer à l'état suivant.

4. Pour ce faire, on va chercher les valeurs des probabilités de transition de l'état dans lequel on se trouve. Puis, on génère un nombre aléatoire selon une distribution uniforme entre 0 et 1, car les valeurs des probabilités de transition se situent également entre 0 et 1.

5. On regarde si ce nombre est compris dans la probabilité de boucler sur le même état. Si c'est le cas, on reste dans le même état, sinon on passe à l'état suivant. On s'arrange pour ne pas dépasser l'état dix, qui est le dernier état du modèle de mot.

6. Finalement, on fait la même opération qu'au point 3, mais pour l'état déterminé au point 5. Puis, on itère au point 4 jusqu'à obtenir les 200 vecteurs de cepstres.

10.2.2 Résultats obtenus

Le mot écrit en dessous de chaque graphique est le mot (cepstres synthétiques) à identifier. Les points verts indiquent le niveau de probabilité de décodage de chaque mot. Le niveau de probabilité est normalisé par rapport au meilleur score obtenu. Dès lors, ce dernier vaut '1'.

Voici les résultats obtenus :

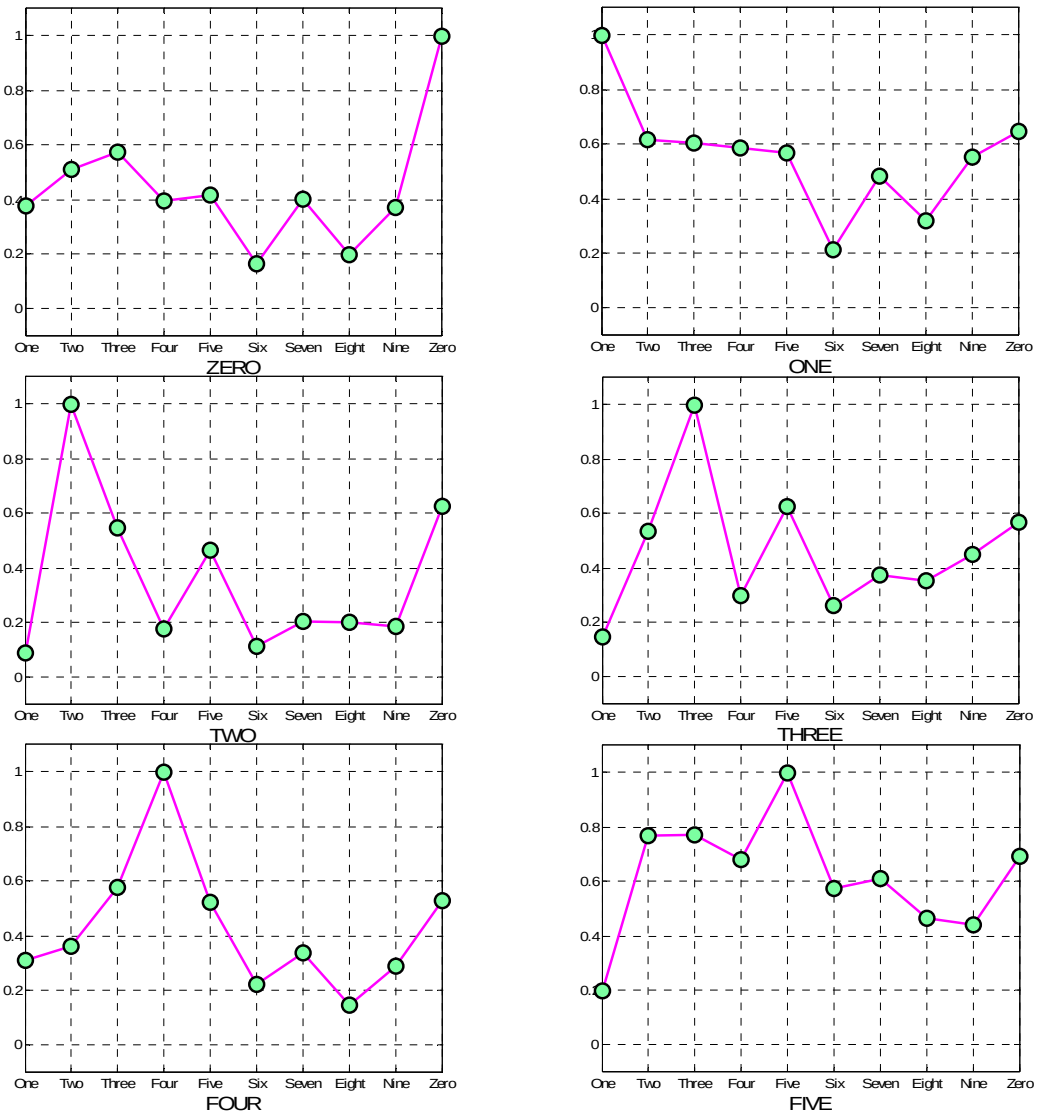


Figure 33 – Test de décodage des cepstres synthétiques représentant les mots 0 à 5 (Matlab)

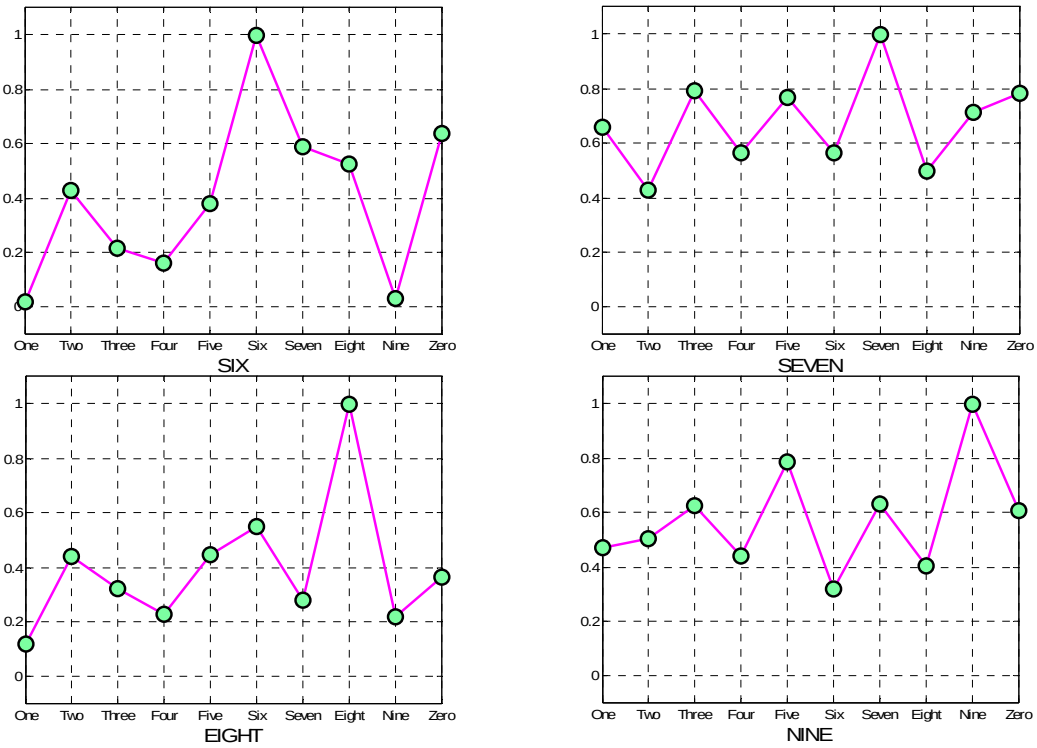


Figure 34 – Test de décodage des cepstres synthétiques représentant les mots 6 à 9 (Matlab)

On observe que l'algorithme de Viterbi fonctionne correctement, car lorsque l'on génère des vecteurs de cepstres synthétiques en fonction des modèles de mots, l'algorithme retrouve le modèle qui a été utilisé.
Sur 200 générations de mots synthétiques (20 fois pour chaque mot), il y a eu deux mots non reconnus, ce qui fait un taux de reconnaissance de 198/200 = 99 [%].

11 Test du fonctionnement de la reconnaissance sous Matlab

Afin de vérifier le bon fonctionnement de la reconnaissance vocale réalisée sous Matlab, on va réaliser deux séries de tests distincts.
La première consiste à tester tout le système en y appliquant les mots qui avaient été utilisés lors de la phase d'apprentissage des modèles de HMM.
La deuxième consiste à tester ce même système en utilisant des autres mots qui n'ont pas été utilisés lors de la phase d'entraînement.

11.1 Test avec des mots utilisés lors de la phase d'entraînement

Le mot écrit en dessous de chaque graphique est le mot à identifier. Les points verts indiquent le niveau de probabilité de décodage de chaque mot. Le niveau de probabilité est normalisé par rapport au meilleur score obtenu. Dès lors, ce dernier vaut '1'.

Voici les résultats obtenus :

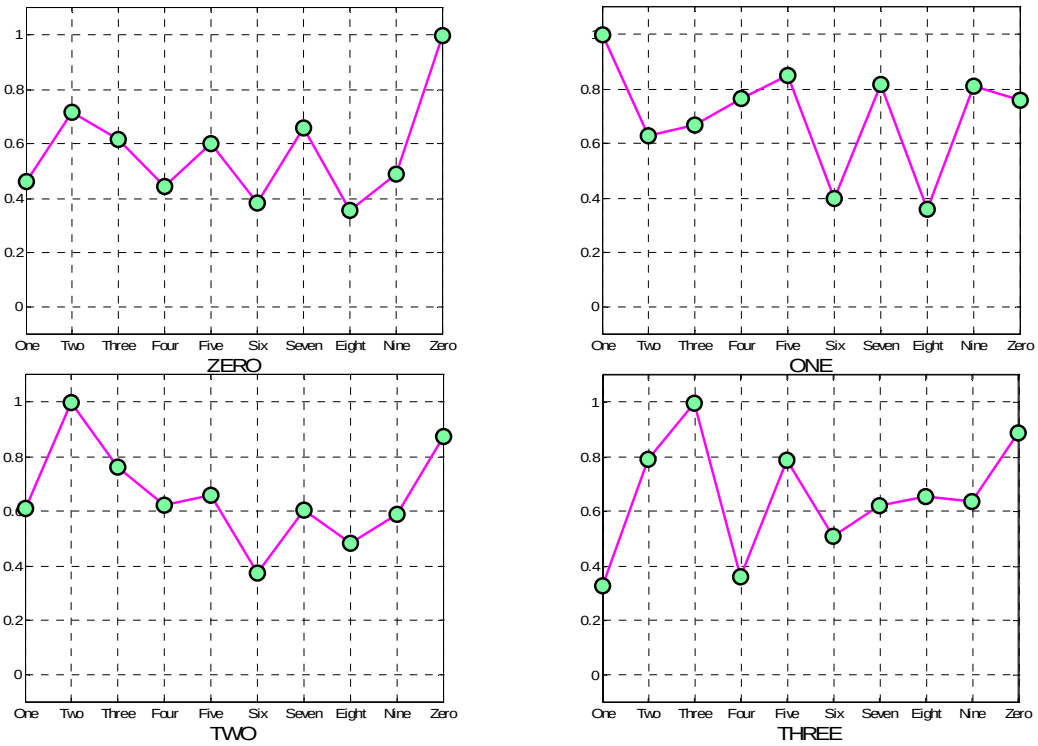


Figure 35 – Test de décodage des mots (0 à 3) utilisés lors de l'apprentissage (Matlab)

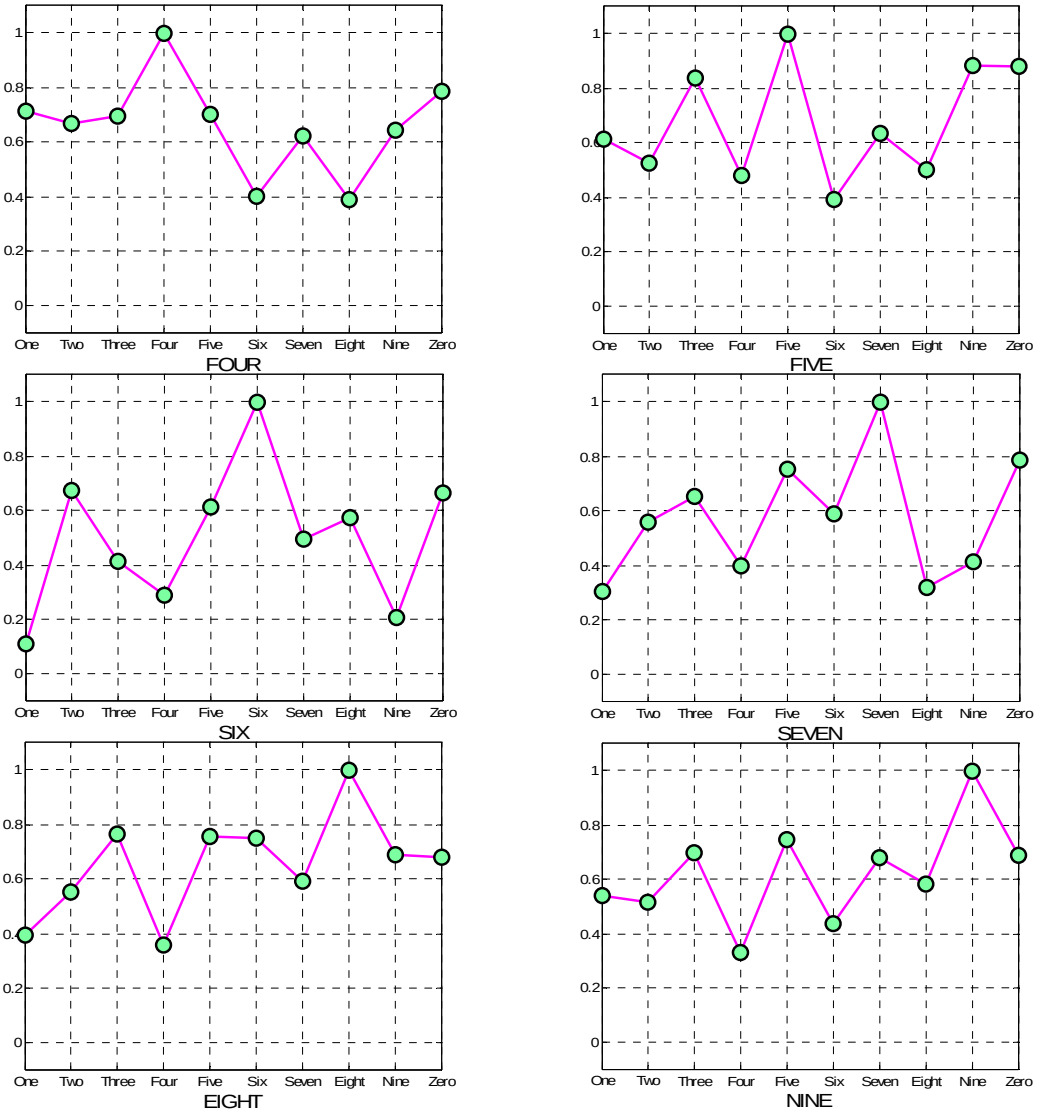


Figure 36 – Test de décodage des mots (4 à 9) utilisés lors de l'apprentissage (Matlab)

On observe que chaque mot est décodé correctement. Ceci prouve que le système fonctionne avec un taux de décodage de 100 [%], si on lui applique des mots utilisés lors de la phase d'entraînement. Ceci est logique puisque les modèles ont été paramétrés avec ces mêmes mots.

11.2 Test avec des mots non utilisés lors de la phase d'entraînement

Dans ce test, dix mots ont été enregistrés sous HTK et n'ont pas servi lors de la phase d'entraînement des chaînes de Markov. Voici les résultats obtenus :

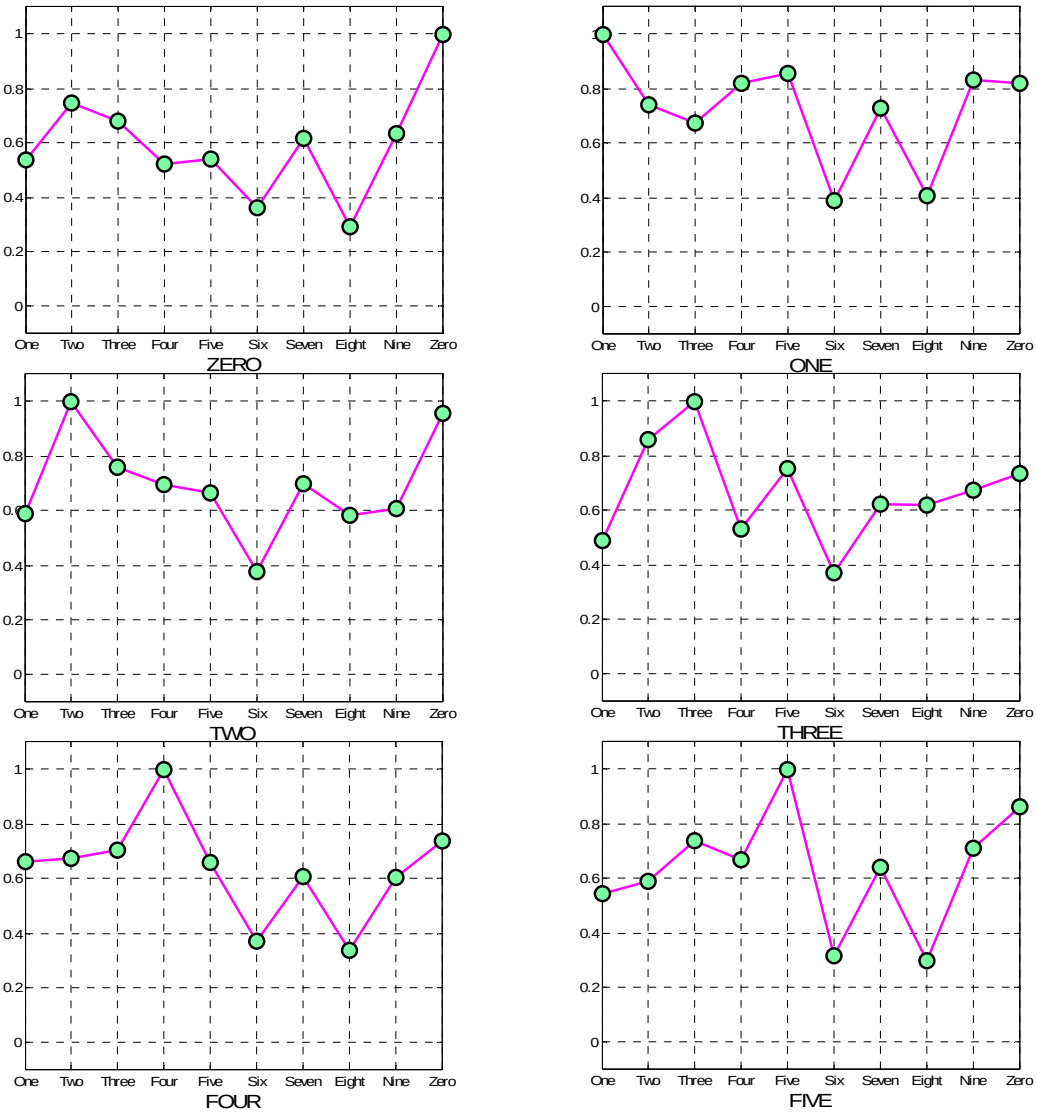


Figure 37 – Test de décodage des mots (0 à 5) non-utilisés lors de l'apprentissage (Matlab)

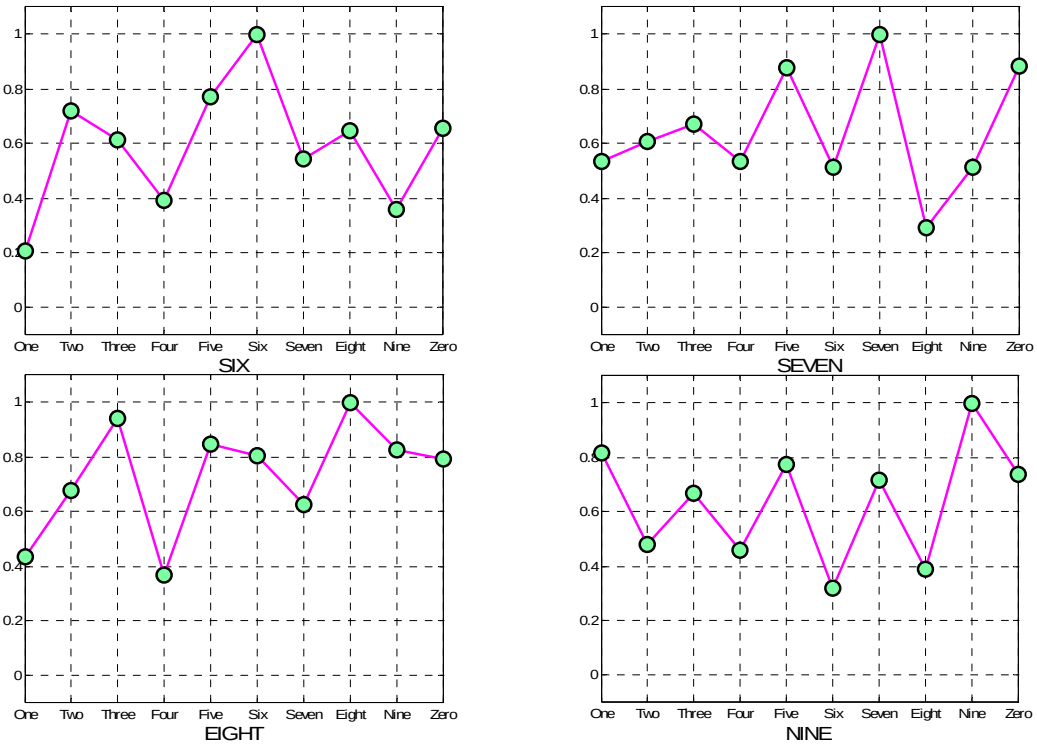


Figure 38 – Test de décodage des mots (5 à 9) non-utilisés lors de l'apprentissage (Matlab)

On observe que pour dix mots enregistrés sous HTK dans les mêmes conditions que ceux enregistrés pour entraîner les HMM, le système fonctionne correctement

11.3 Test en enregistrant 200 mots avec Matlab

Lors de ce test, chaque mot du dictionnaire a été prononcé 20 fois de manière distincte. Voici les résultats obtenus :

ONE	= 12/20	Dans 8 cas, le mot a été confondu avec SEVEN ou NINE
TWO	= 15/20	Dans 5 cas, le mot a été confondu avec ONE ou ZERO
THREE	= 15/20	Dans 5 cas, le mot a été confondu avec EIGHT
FOUR	= 20/20	
FIVE	= 14/20	Dans 6 cas, le mot a été confondu avec NINE
SIX	= 17/20	Dans 3 cas, le mot a été confondu avec EIGHT
SEVEN	= 20/20	
EIGHT	= 20/20	
NINE	= 20/20	
ZERO	= 20/20	

Ce qui donne un pourcentage total de reconnaissance = (173/200)·100 = **86.5 [%]**
Il est a noter que l'enregistrement des sons sous Matlab n'est pas identique à l'enregistrement sous HTK.

12 Implémentation en langage C pour le portage sur DSP

Le but ultime de ce travail est d'implémenter tout le système de reconnaissance décrit précédemment dans une unité portable équipé d'un DSP à virgule flottante (afin de ne pas avoir les contraintes imposées par un système à virgule fixe).

La programmation de la reconnaissance à implémenter dans le DSP doit être codée en langage C et/ou assembleur. Ces deux langages sont moins évolués que du code Matlab et n'offrent pas toutes les simplicités et subtilités de ce dernier. Dès lors, chaque routine transformée en C sera directement testée avec les fonctions écrites en Matlab, afin de valider leur bon fonctionnement.

12.1 La carte EZ-KIT Lite

Le DSP choisit pour l'implémentation est le SHARC ADSP-21160M de chez Analog Devices. Il est implanté dans la carte « EZ-KIT Lite » qui contient une entrée micro pour capter les échantillons sonores et une connexion USB afin d'insérer le programme depuis un ordinateur.

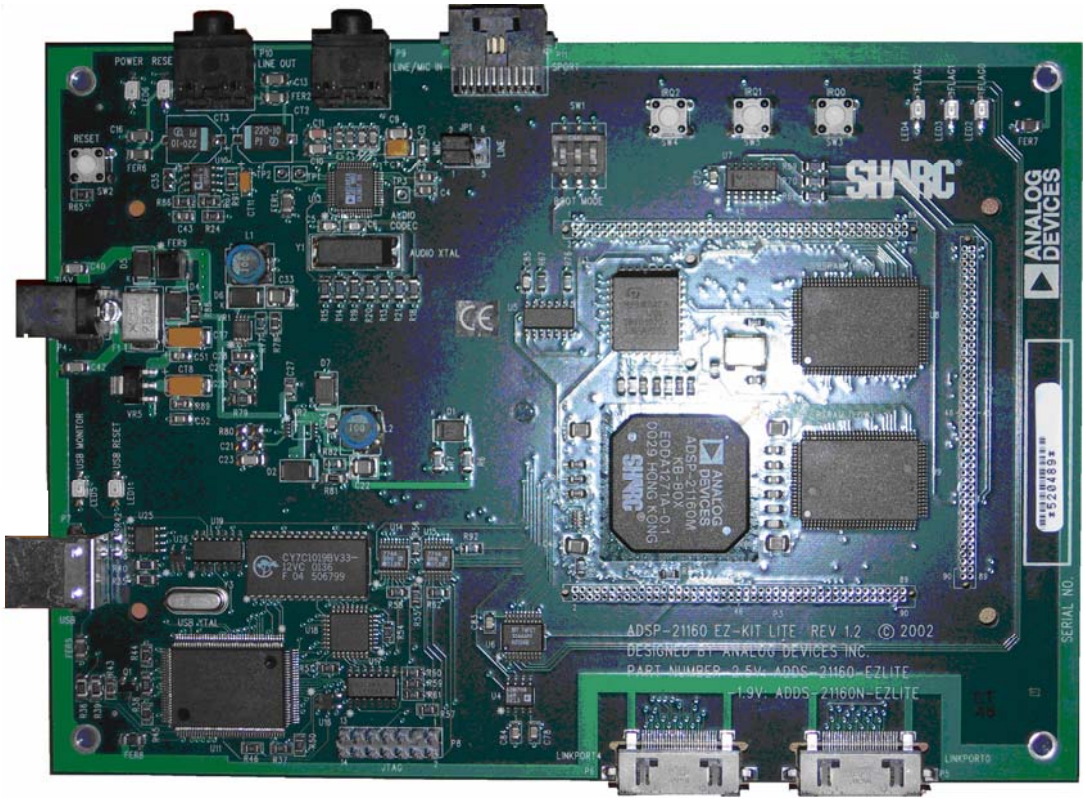


Figure 39 – Carte EZ-KIT Lite

12.2 Le « mapping » mémoire

Le mapping mémoire consiste à déterminer si l'architecture de la carte permet le stockage de l'algorithme à implémenter. En effet, lorsque l'algorithme tourne sur un ordinateur, la capacité mémorielle du système est « quasi infinie » et ce problème ne se pose pas. Dès que l'on souhaite implémenter l'algorithme dans une unité portable (comme cette carte avec son DSP), les mémoires ROM et RAM ont des capacités limitées.
Voici le schéma de l'architecture de la carte avec le DSP :

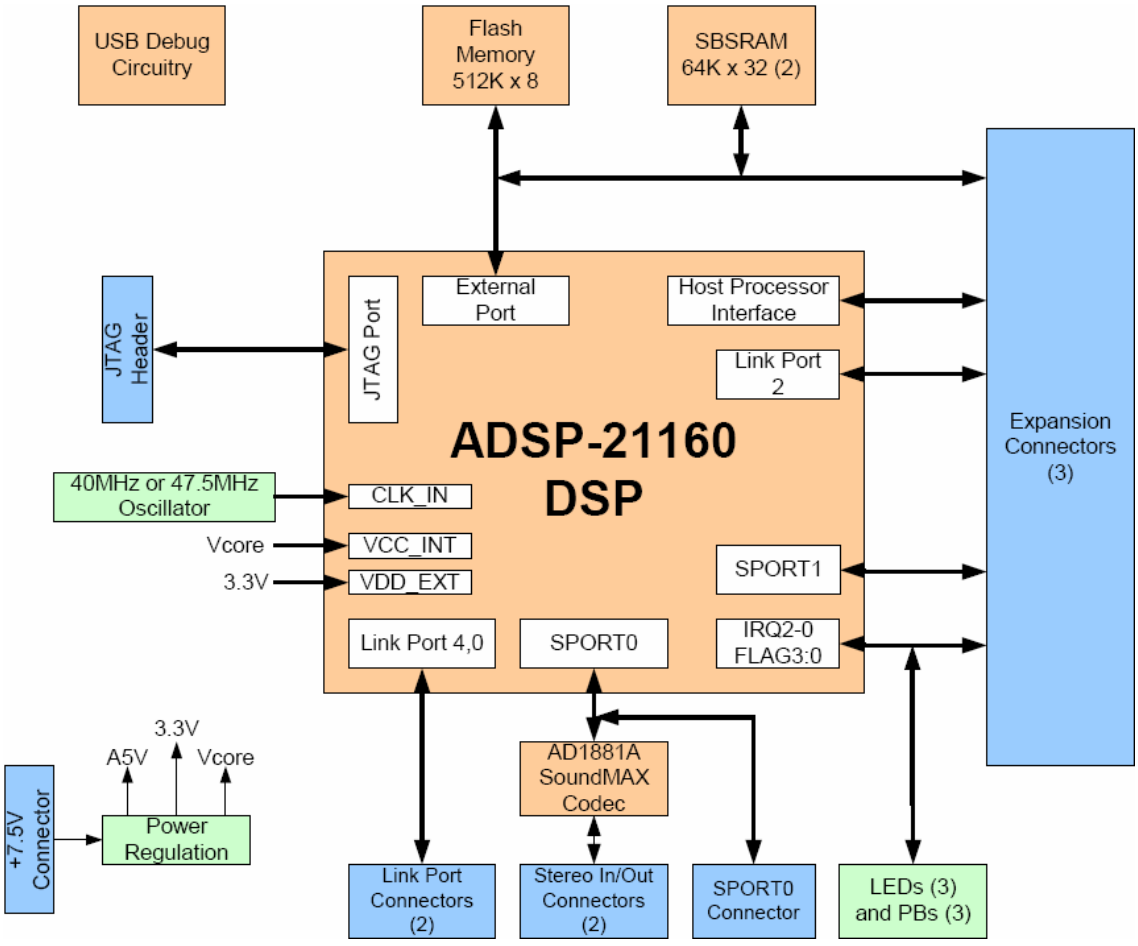


Figure 40 – Architecture de la carte et du DSP (ADSP-21160_EZ-KIT_Lite_Manual)

On observe que la carte contient trois mémoires, une mémoire Flash-ROM (512K x 8) et deux mémoires SBS-RAM (64K x 32).

Voici une représentation plus détaillée de la structure de ces mémoires :

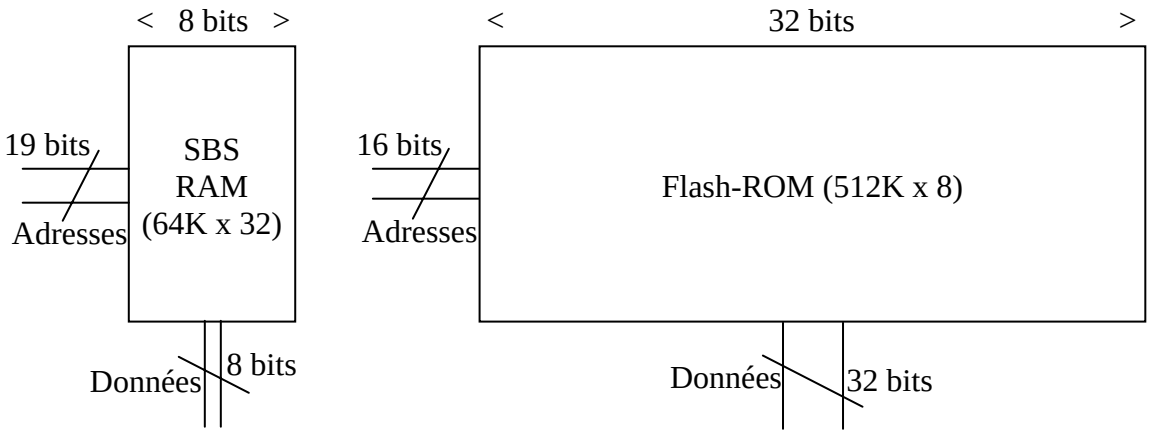


Figure 41 – Architecture des mémoires RAM et ROM

A partir de la figure précédente, on peut déterminer la capacité de stockage des mémoires de la carte.
Pour la mémoire RAM (64K x 32), on peut déterminer l'espace mémoire de la manière suivante :
64K = 64·1024 = ***65'536 adresses*** $\Rightarrow log_2(65'536)$ = 16 bits pour les adresses.
32 bits = ***4 Bytes*** $\Rightarrow$ une donnée est représentée par un mot de 4 Bytes.
Puisqu'il y a deux mémoires RAM, on peut déterminer la capacité totale de la mémoire RAM en octet ou Byte, qui vaut : *65'536·4·2 =* ***524.288 [ko]***

Pour la mémoire ROM (512K x 8) :
512K = 512·1024 = ***524'288 adresses*** $\Rightarrow log_2(524'288)$ = 19 bits pour les adresses.
8 bits = ***1 Byte*** $\Rightarrow$ une donnée est représentée par un mot de 1 Byte.
De ce fait, la capacité de la mémoire ROM vaut : *524'288·1 =* ***524.288 [ko]***

Dès lors, on obtient une capacité mémoire totale de : *2·524'288 =* ***1.048576 [Mo]***.
En déterminant la capacité de stockage totale des mémoires de la carte qui est de 1 [Mo], on connaît la limite de la taille du code que l'on peut y insérer ; le nombre de constantes et de variables ainsi que le nombre d'échantillons qu'elles peuvent stocker.

Plusieurs tableaux de constantes (chacune codées sur 4 octets) sont à stocker dans la mémoire ROM. Voici le dimensionnement de la mémoire à utiliser :
Le code comprend : - 1 tableau de 724 x 14 $\Rightarrow$ *724·14·4 = 40.544 [ko]*
- 10 tableaux de 724 x 9 $\Rightarrow$ *10·724·9·4 = 260.640 [ko]*
- 10 tableaux de 11 x 11 $\Rightarrow$ *10·11·11·4 = 4.840 [ko]*

Ceci donne un espace de stockage dans la ROM pour les constantes de :
(40.544 + 260.640 + 4.840)· k = ***305.668 [ko]***
A noter que la ROM va également contenir le code C de l'algorithme complet de reconnaissance de la parole qui est de 25 [ko].

On en conclut que la capacité de la mémoire ROM est largement suffisante afin de stocker les constantes du système, ainsi que le code nécessaire à la reconnaissance.

Concernant le stockage des échantillons d'entrée et des variables locales et globales, ceci est géré par les deux mémoires RAM. Si l'on considère des trames d'échantillons sur une durée de 5 [s] et que chaque échantillon est codé sur 4 octets, on obtient une taille de données à stocker de : $5 \cdot fe \cdot 4 = 5 \cdot 16'000 \cdot 4 =$ ***320 [ko]***.
Dès lors, il reste 200 [ko] afin de stocker les variables du système, ce qui est amplement suffisant.

12.3 Test du fonctionnement des routines implémentées en langage C

Toutes les fonctions écrites en Matlab sont réalisées en C, de manière à permettre la programmation du DSP. Chacune de ces fonctions est testée puis validée à partir des résultats obtenus lors de leur réalisation antérieure en code Matlab.

Il y a 12 fonctions à réaliser et à valider, plus le programme principal :

1. Préaccentuation
2. Fenêtre de Hamming
3. Autocorrélation
4. Levinson-Durbin
5. Conversion a(k) $\Rightarrow$ c(k)
6. Décodage LPC
7. Calcul de l'énergie moyenne en log
8. Calcul du nombre de passage par zéro
9. VAD
10. Dictionnaire de quantification vectorielle
11. Sélection des modèles de HMM
12. Viterbi
13. Programme principal d'exécution de la reconnaissance

Pour les fonctions 1, 3, 6, 7 et 8, un signal quelconque de parole a été choisi, puis les résultats donnés par Matlab et par les fonctions écrites en C sont comparés et doivent correspondre.
Dans les graphiques qui suivent, les valeurs des échantillons en bleu sont les résultats obtenus avec Matlab, et les échantillons en rouge ceux donnés par les code en C.

12.3.1 Fonction réalisant la préaccentuation

Vérification de l'application du filtre de préaccentuation à un signal de parole :

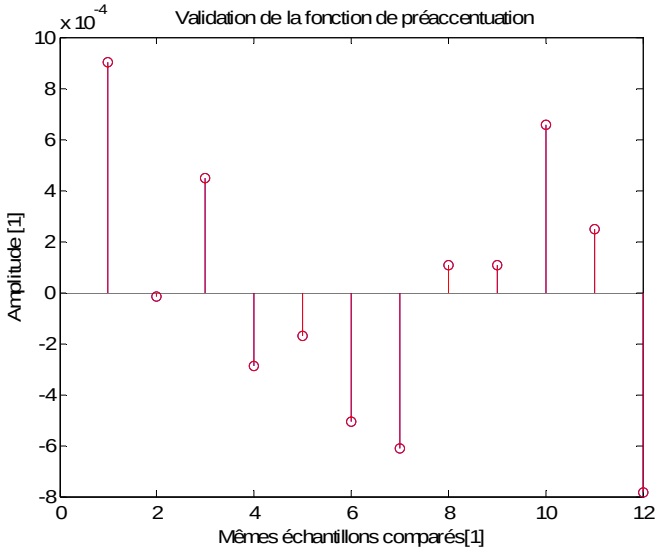


Figure 42 – Validation de la fonction de préaccentuation (C→Matlab)

On observe que les échantillons se superposent idéalement. Dès lors, on peut valider le bon fonctionnement de la fonction de préaccentuation.

12.3.2 Fonction réalisant la fenêtre de Hamming

Vérification de la forme de la fenêtre de Hamming (seul 11 échantillons sont représentés) :

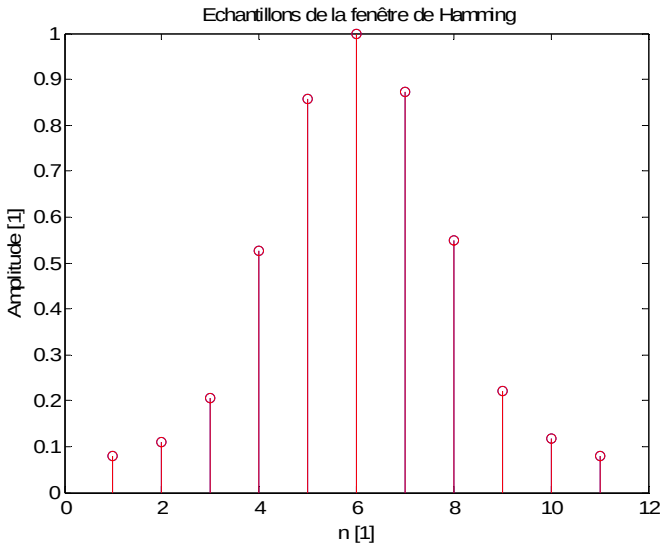


Figure 43 – Validation de la forme de la fenêtre de Hamming (C→Matlab)

On observe que les fenêtres de Hamming sont identiques. La fonction est donc validée.

12.3.3 Fonction réalisant une autocorrélation

Vérification du calcul de l'autocorrélation sur un signal de parole :

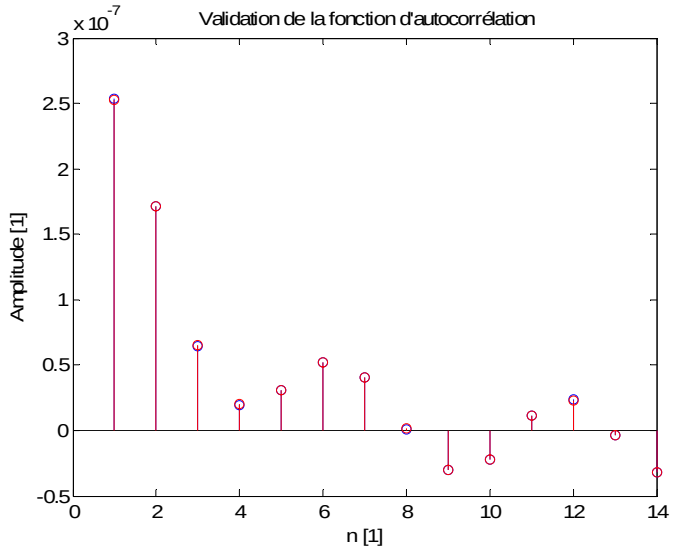


Figure 44 – Validation du calcul de l'autocorrélation (C→Matlab)

Le résultat de l'autocorrélation calculée pour 14 paramètres donne des résultats similaires. Par conséquent, la fonction d'autocorrélation est validée.

12.3.4 Fonction implémentant l'algorithme de Levinson-Durbin

Vérification du calcul des coefficients a(k), soit de la réponse fréquentielle du conduit vocal pour un signal de parole :

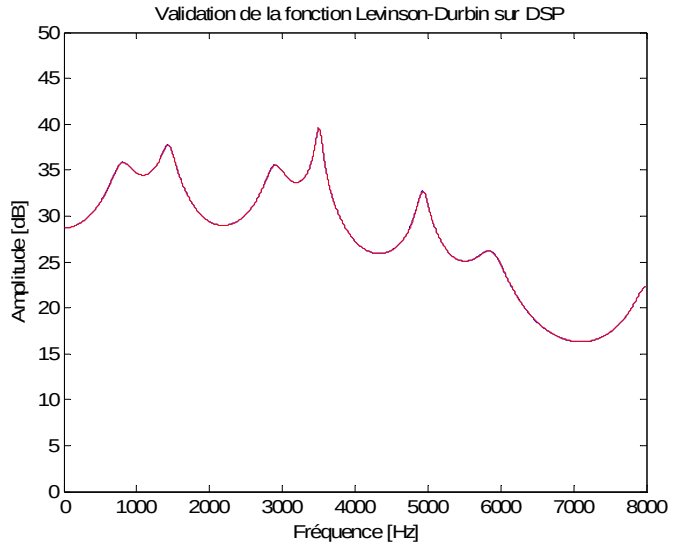


Figure 45 – Validation du calcul des a(k) par la méthode de Levinson-Durbin (C→Matlab)

On remarque que la réponse fréquentielle obtenue à partir des coefficients a(k) pour la fonction en C correspond à la réponse fréquentielle donnée par Matlab. De ce fait, le fonctionnement de la routine est validé.

12.3.5 Fonction réalisant la conversion a(k) ⇒ c(k)

Vérification du calcul des coefficients c(k) :

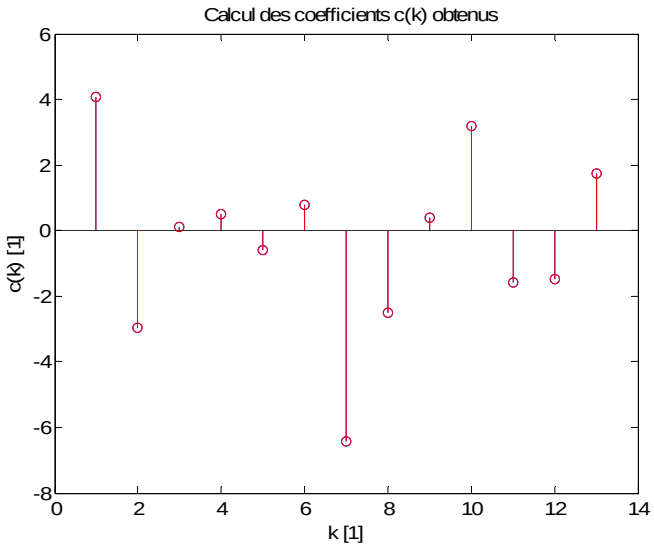


Figure 46 – Validation du calcul des c(k) à partir des a(k) (C→ Matlab)

On observe que les échantillons se superposent idéalement. Dès lors, on peut valider le bon fonctionnement de la fonction convertissant les a(k) en c(k).

12.3.6 Fonction réalisant le décodage LPC

Cette fonction réutilise toutes les routines précédentes afin de calculer tous les vecteurs de coefficients c(k) d'un signal de grande taille. Elle correspond à figure 5 de la page 18, sans le VAD.
Pour ce test, les coefficients c(k) affichés afin de comparer les valeurs ont été pris dans chacun des vecteurs calculés à divers positions (ce ne sont pas des coefficients consécutifs).

Voici les coefficients c(k) obtenus :

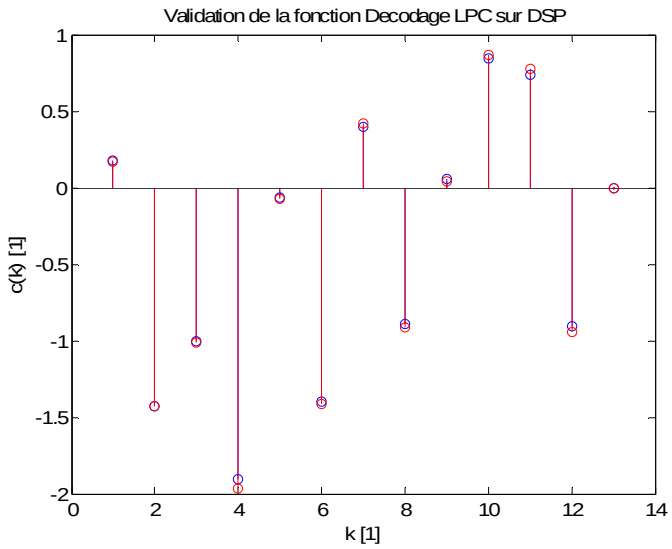


Figure 47 – Validation de la fonction du décodage LPC (C→ Matlab)

On observe que les coefficients c(k) ont des valeurs très similaires. Dès lors, on peut valider le bon fonctionnement du bloc de décodage LPC.

12.3.7 Fonctions calculant l'énergie moyenne et le nombre de passage par zéro

Ces deux fonctions sont utiles au VAD et permettent les mesures de l'énergie moyenne du signal, ainsi que son nombre de passage par zéro. Ceci afin de décider dans le VAD s'il s'agit de bruit ou d'un signal de parole.
Pour un signal de parole donné, le code Matlab a calculé une énergie moyenne de –65.9589 [dB], alors que la fonction en C a calculé une énergie moyenne –65.969 [dB].
Dans les deux cas, le nombre de passage par zéro a donné la valeur de 222.
Dès lors, on peut valider le bon fonctionnement de ces deux fonctions.

12.3.8 Fonction implémentant le VAD

Afin de tester cette fonction, on applique un mot pour le test (mot « FOUR ») qui contient une partie de bruit au début, puis le mot prononcé. Le but étant de décoder les vecteurs de cepstres dès que le système à détecter de la parole et non pas pendant qu'il y a uniquement du bruit.
Pour ce test, les coefficients c(k) affichés afin de comparer les valeurs ont été pris dans chacun des vecteurs calculés à divers positions (ce ne sont pas des coefficients consécutifs).

Voici les coefficients c(k) obtenus :

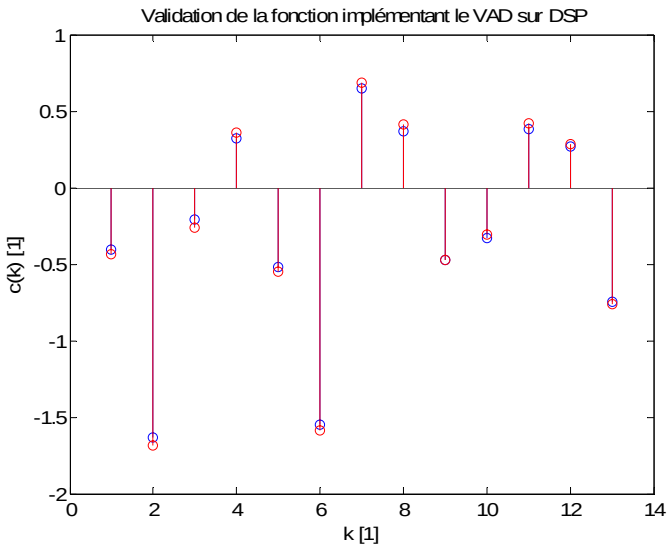


Figure 48 – Validation de la fonction VAD (C→Matlab)

On observe que les coefficients c(k) ont des valeurs très similaires. Ceci prouve que le système commence le décodage au bon moment, et ne le fait pas pendant des instants ne contenant pas d'informations pertinentes. Dès lors, on peut valider le bon fonctionnement du bloc implémentant le VAD.

12.3.9 Fonction réalisant la quantification vectorielle

En appliquant à cette fonction deux centroïdes faisant partis du dictionnaire, soit les barycentres 23 et 254, la fonction à retourner les deux numéros de classes 23 et 254.
Par conséquent, le fonctionnement de la routine est validé.

12.3.10 Fonction sélectionnant des modèles de HMM

La difficulté de cette fonction est de pouvoir retrouver n'importe quel élément des matrices de probabilités d'observation et de transition. En effet, en Matlab il est facile de sélectionner une valeur dans un tableau en deux dimensions, mais pas en C.
Dès lors, il faut faire passer le tableau à une dimension. A ce moment, il faut déplacer de manière correcte le pointeur sélectionnant une case.

Pour le test, le modèle HMM du mot « THREE » a été sélectionné. Le but étant de déplacer le pointeur afin de sélectionner la valeur souhaitée. Le calcul du pointeur s'effectue de la manière suivante :

$$Pointeur = numéro\ d'indexe\ de\ la\ ligne \cdot nombre\ de\ colonnes + numéro\ d'indexe\ de\ la\ colonne$$

A partir de cette formule, il est possible de déplacer le pointeur à l'endroit choisi. Les résultats obtenus ont validé cette formule, et de ce fait, le fonctionnement correct de cette routine.

12.3.11 Fonction implémentant l'algorithme de Viterbi

L'algorithme de Viterbi a été réalisé en C, mais n'a pu être validé. En effet, il s'agit de l'algorithme le plus difficile à transformer en code C. Le problème est dû à la gestion de ces multiples matrices à deux dimensions (matrices des probabilités d'observation, de transition et Phi) qui doivent être transformées en tableaux à une dimension. De ce fait, la position des divers pointeurs sur ces tableaux est sans doute incorrecte dans une certaine portion du code.
A cause du manque de temps, je n'ai pas eut la possibilité de cerner l'erreur et n'ai pas réussi à valider le bon fonctionnement de la routine en C.

13 Conclusion

Le but de ce travail de diplôme a été atteint. En effet, j'ai réalisé une reconnaissance vocale basée sur les chaînes de Markov cachées en langage de haut-niveau (HTK) et en langage Matlab. Les fonctions écrites en langage C ont toutes été validées, sauf la routine réalisant l'algorithme de Viterbi.

Au sujet de la partie traitant de l'extraction des paramètres LPC donnant les vecteurs cepstraux, les résultats obtenus sont corrects. En effet, le bloc d'extraction des paramètres donne des résultats similaires à tous les niveaux de programmation, soit HTK, Matlab et C.

Concernant le VAD, j'ai développé une méthode originale qui est hybride entre deux méthodes précédemment décrites dans la littérature. Cette version semble fonctionner très correctement, il resterait peut-être à comparer son efficacité avec d'autres méthodes connues ; ceci toutefois serait suffisant à remplir tout un travail de diplôme, et dans les limites du temps imparti, j'ai préféré terminer toute la chaîne de traitement en accord avec le cahier des charges défini. Comme pour le bloc d'extraction des paramètres LPC, le VAD fonctionne également à tous les niveaux de programmation.

À propos du bloc effectuant la quantification vectorielle, le dictionnaire composé des 724 barycentres représentent correctement les divers vecteurs cepstraux que l'on observe pour un vocabulaire de dix mots. L'avantage de ce bloc est de ressortir directement les numéros de classe des vecteurs de cepstres, au lieu de ces 13 composantes. Ces numéros sont ensuite transmis au modèle de Markov, qui retourne les probabilités d'observation et de transition correspondantes.

Concernant le bloc des chaînes de Markov, l'apprentissage des HMM réalisé à l'aide du logiciel HTK a permis de largement simplifier la création des modèles de mot.
Pour l'implémentation sur DSP, les modèles de HMM ont directement été enregistrés avec leur probabilité d'observation et de transition respectives. En effet, ceci évite énormément de calculs de fonctions Gaussiennes au niveau du processeur.

Finalement, l'algorithme de Viterbi a été réalisé et validé en Matlab, mais pas en C. Il s'agit de l'algorithme le plus complexe à porter sur le DSP, et je n'ai pas eut assez de temps afin de le faire fonctionner en C.

En définitive, ce projet aura été extrêmement intéressant. En effet, j'ai pu me rendre compte de la difficulté de réaliser tout un algorithme de reconnaissance vocale en Matlab, puis de porter ce dernier sur une plateforme portable, soit une carte DSP. Ce travail m'aura apporté beaucoup de nouvelles connaissances sur un domaine que je ne connaissais que très peu. De plus, il m'a permis de considérablement améliorer mes connaissances en C.

14 Bibliographie

[1] Traitement de la parole : auteur : René Boite ; année : 2000
édition : presses polytechniques et universitaires romandes.

[2] HTK Book (for HTK Version 3.2.1) : année : 2002 ; édition : Cambridge University Engineering Department.

[3] Fundamentals of Speech Recognition : auteur : Lawrence Rabiner ; année : 1993
édition : Prentice-Hall.

[4] Hidden Markov Models and artificial neural networks for speech and speaker recognition : auteur : Jean Hennebert ; année 1998 ; lieu : EPFL Lausanne.

[5] Détection et reconnaissance des sons pour la surveillance médicale :
auteur : Dan Mircea Istrate ; année 2003 ; lieu : INPG Grenoble.

15 Remerciements

Je tiens à remercier Messieurs Jean Hennebert de l'Université de Fribourg et Monsieur Robert Van Kommer de chez Swisscom, pour m'avoir reçu à Fribourg et pour m'avoir donné des conseils précieux.
Monsieur Jean Hennebert a pris sur son temps pour me guider lorsqu'un problème surgissait, qu'il en soit ici remercié chaleureusement.

Je remercie également les assistants de l'EIVD Jean-Pierre Miceli et Gilles Primault pour leur aide précieuse concernant la réalisation de la reconnaissance vocale en langage C. En effet, ils n'ont pas hésité à prendre sur leur temps de travail afin de m'expliquer quelques éléments de la syntaxe du langage C. De plus, ils m'ont aidé à débugger mon code C lorsqu'il y avait des problèmes dû au langage que je ne pouvais résoudre par moi-même.

16 Table des figures

17 Annexes

Ci-après, on trouve la liste des déclarations de toutes les routines utiles à l'implémentation sur DSP. Concernant la réalisation de la reconnaissance vocale en haut niveau sous HTK et tous les codes Matlab réalisés pour créer une reconnaissance de la parole détaillée, il faut ce référer au CD-ROM joint au rapport. Ce dernier contient également le corps des fonctions C pour l'implémentation sur le DSP.

17.1 Liste de toutes les routines réalisées pour l'implémentation sur DSP

Dans cette partie, on trouve toute les déclarations des fonctions utiles à l'implémentation de l'algorithme de reconnaissance de la parole dans le DSP.

17.1.1 Fonction réalisant la préaccentuation

Cette routine effectue la préaccentuation du signal sonore capté par le micro :

```
// Paramètres IN     : signal_x -> signal provenant du micro
// Paramètres IN/OUT : xconv    -> signal préaccentué
// Paramètres OUT    : -

void preaccentuation(float *signal_x, float *xconv);
```

17.1.2 Fonction réalisant le fenêtrage des échantillons

Cette méthode génère une fenêtre de Hamming permettant de pondérer les échantillons du signal sonore :

```
// Paramètres IN     : -
// Paramètres IN/OUT : w -> fenêtre de Hamming sur 400 échantillons
// Paramètres OUT    : -

void fen_hamming(float *w);
```

17.1.3 Fonction calculant une autocorrélation

Cette routine effectue l'autocorrélation du signal sonore sur le nombre de paramètre souhaité :

```
// Paramètres IN     : Nb_parametres -> correspond aux nombres de a(k) désirés
//                     signal_x      -> signal sonore
//                     longueur      -> longueur du signal à traité
// Paramètres IN/OUT : rxx           -> vecteur d'autocorrélation
// Paramètres OUT    : -

void LPC_autocorr(const int Nb_parametres, const int longueur, float *signal_x,
                  float *rxx);
```

17.1.4 Fonction calculant les coefficients a(k)

Cette méthode calcule les coefficients LPC par l'algorithme de Levinson-Durbin :

```
// Paramètres IN     : Nb_parametres -> correspond aux nombres de a(k) désirés
//                     rxx           -> vecteur d'autocorrélation
// Paramètres IN/OUT : a_k           -> vecteur des coefficients a(k)
// Paramètres OUT    : -

void Levinson_Durbin(int const Nb_parametres, float *rxx, float *a_k);
```

17.1.5 Fonction calculant les cepstres

Cette routine transforme les coefficients a(k) en cepstres c(k) :

```
// Paramètres IN     : a_k     -> vecteur des coefficients a(k)
// Paramètres IN/OUT : ck_norm -> vecteur des coefficients cepstraux c(k)
// Paramètres OUT    : -

void ak_to_ck(float *a_k, float *ck_norm);
```

17.1.6 Fonction effectuant le décodage LPC

Cette méthode utilise les fonctions précédentes afin de calculer les c(k) d'un mot :

```
// Paramètres IN     : signal_ bruite -> signal sonore
//                     borne_min      -> premier échantillon du signal à traiter
//                     borne_max      -> dernier échantillon du signal à traiter
//                     k              -> indique le kième vecteur de cepstres
// Paramètres IN/OUT : c_k -> vecteur des coefficients cepstraux c(k)
// Paramètres OUT    : retourne l'indice k global au programme

int Decodage_LPC(float *signal_bruite, int borne_min, int borne_max, float *c_k, int k);
```

17.1.7 Fonction calculant l'énergie moyenne

Cette fonction est utile au VAD et permet le calcul de l'énergie moyenne logarithmique d'une portion de signal :

```
// Paramètres IN     : borne_min -> premier échantillon du signal à traiter
//                     borne_max -> dernier échantillon du signal à traiter
//                     signal    -> signal sonore
// Paramètres IN/OUT :
// Paramètres OUT    : retourne l'énergie moyenne de la tranche calculée

float Energie_moyenne_log(int borne_min, int borne_max, float *signal)
```

17.1.8 Fonction calculant le nombre de passage par zéro

Cette fonction est utile au VAD et permet le calcul du nombre de passage par zéro d'une portion de signal :

```
// Paramètres IN     : borne_min -> premier échantillon du signal à traiter
//                     borne_max -> dernier échantillon du signal à traiter
//                     signal    -> signal sonore
// Paramètres IN/OUT :
// Paramètres OUT    : retourne le nombre de passage par '0' de la tranche calculée

int Energie_passage_zero(int borne_min, int borne_max, float *signal)
```

17.1.9 Fonction réalisant la détection de parole

Cette routine réalise l'algorithme du VAD afin de détecter la présence de voix :

```
// Paramètres IN     : fe              -> fréquence d'échantillonnage du système
//                     signal_bruite   -> signal sonore
//                     longueur_signal -> longueur du signal sonore
// Paramètres IN/OUT : c_k             -> matrice des coefficients c(k)
// Paramètres OUT    : matrice des vecteurs de cepstres correspondant au mot prononcé

float *VAD_LPC (int fe, float *signal_bruite, const int longueur_signal, float *c_k);
```

17.1.10 Fonction réalisant la quantification vectorielle

Cette méthode ressort les numéros de classe du dictionnaire correspondant aux cepstres reçus :

```
// Paramètres IN     : tableau_phonemes -> matrice des vecteurs de cepstres
//                     longueur         -> nombre de vecteurs de cepstres
//                     Nb_composantes   -> nombre de coefficients cepstraux par vecteur
// Paramètres IN/OUT : classes          -> vecteur contenant les numéros de classe
// Paramètres OUT    : -

void Dico_vectorielle(float *tableau_phonemes, const int longueur, const int
                      Nb_composantes, int *classes);
```

17.1.11 Fonction sélectionnant le modèle de HMM pour la reconnaissance

Cette routine permet de sélectionner le modèle de HMM que l'on va transmettre à l'algorithme de Viterbi :

```
// Paramètres IN     : variable -> indique le numéro du modèle de HMM correspondant
// Paramètres IN/OUT : -
// Paramètres OUT    : retourne le modèle de HMM sélectionné

pHMM_out selection_HMM_trans_obs(int variable);
```

17.1.12 Fonction effectuant le décodage du mot par l'algorithme de Viterbi

Cette méthode réalise le décodage du mot en appliquant l'algorithme de Viterbi au modèle de HMM passé en paramètres :

```
// Paramètres IN     : p_Observation -> probabilité d'observation du modèle de HMM
//                     p_Transition  -> probabilité de transition du modèle de HMM
// Paramètres IN/OUT : -
// Paramètres OUT    : retourne la probabilité maximum au travers du modèle

float Viterbi(float *p_Observation, float *p_Transition);
```