République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Dr Yahia Farès de Médéa
Faculté des Sciences et de la Technologie
Département du Génie Électrique et de l'Informatique



Mémoire

Présenté en vue d'obtention du diplôme de Magister

Spécialité : Électronique

Option : Instrumentation et Microélectronique

Présenté par :
M^{er} MANSOURI Slimane

Thème

Traitement homomorphique de la parole par DSP

Soutenue publiquement le 27 Mai 2014

Jury

Pr. Salam ABUDURA	Professeur, U. Médéa	Président
Pr. Abderrezak GUESSOUM	Professeur, U. Blida	Examinateur
Dr. Mohamed OULDZMIRLI	MCA, U. Médéa	Examinateur
Dr. Noureddine BATEL	MCA, U. Médéa	Encadreur

2013-2014

Remerciements

À l'issue de cette fin d'étude, j'adresse mes sincères remerciements premièrement à dieu tout puissant qui m'a donné la santé, la volonté et la patience.

Ensuite, je tiens à adresser mes plus vifs remerciements à mon promoteur Mr. ***Noureddine BATEL****, Maître de conférence à l'université de Médéa pour m'avoir encadré, suivi et encouragé.*

*Mes profonds remerciements vont également à monsieur****: Abdelghani HARRAG****, Enseigner à l'université de M'sila pour ses conseils appréciables et ses critiques constructives.*

Je remercie également, tous les membres du jury de bien vouloir juger ce travail.

Ainsi que tous mes enseignants du Département d'Electronique de l'université de Médéa.

*Je n'exclue pas de ces remerciements toutes les personnes qui m'ont aidé de prés ou de loin dans la réalisation de ce mémoire, notamment ma promotion d'électronique et plus particulièrement****, Bensalahe.H.***

Je veux également remercier ma famille et mes amis pour leur soutien moral.

Enfin, je remercie tout particulièrement mes parents, pour leur soutien inconditionnel tout au long de mes longues années d'études.

ملخص :

توجد العديد من الأساليب التحليلية لإشارة الكلام من أجل استخراج معاملاته مثل التردد الأساسي و الترددات الرنينية. يعرض هذا العمل تنفيذا لترشيح هوموموفيك على (TMS320C6711 DSP) من شركة (Texas Instruments) ،ما سمح لنا بتقدير هذه المعاملات في الوقت الحقيقي. وقد رافق هذا العمل تنفيذ على (Matlab)، من أجل المقارنة و التحقق من النتائج.

Abstract:

Many analytical methods of speech signal exist to extract parameters such as fundamental frequency and formants. This work presents the implementation of a speech homomorphic filtering on TMS320C6711 DSP from Texas Instruments, which allows us to estimate the parameters in real time. This work is accompanied by the Matlab implementation, for comparison and results verification.

Résumé :

Plusieurs méthodes d'analyse du signal de la parole existent afin d'extraire ses paramètres tels que fréquence fondamentale et formants. Ce travail présente l'implémentation d'un filtrage homomorphique de la parole sur le DSP TMS320 C6711 de *Texas Instruments,* ce qui nous permet d'estimer les paramètres de la parole en temps réel. Le travail est accompagné d'une implémentation sur Matlab, à titre de comparaison et vérification des résultats.

Mots clés :

Signal parole – DSP – traitement homomorphique – cepstre – LPC – MFCC – pitch – formants.

SOMMAIRE

INTRODUCTION GENERALE 1

CHAPITRE I

MODELISATION ET ANALYSE DU SIGNAL DE LA PAROLE

I.1 Introduction 3

I.2 Le signal de la parole 3

I.3 Les paramètres de la parole 5

I.3.1 Le fondamental 5

I.3.2 Les formants 5

I.4 Traitement homomorphique de la parole 5

I.4 Systèmes de convolution homomorphique 5

I.5 Pré-traitement acoustique 10

I.5.1 La pré-accentuation du signal de la parole 10

I.5.2 Le fenêtrage 10

I.6 La transformée de Fourier rapide (FFT) 11

I.7 Analyse et paramétrisation du signal de la parole 12

I.7.1 Analyse cepstrale 12

I.7.2 Analyse par prédiction linéaire 16

I.7.3 Analyse par banc de filtres MEL 21

I.8 Le Spectrogramme 26

Conclusion 27

CHAPITRE II

CHOIX DU DSP ET PRESENTATION DE L'OUTIL DE DEVELOPPEMENT

II.1 Introduction 28

II.2 Présentation des DSP 28

II.2.1 Les avantages d'un DSP 29

II.2.2 Possibilités des DSP 30

II.2.3 Noyau du DSP 34

II.3 Classification des DSP 34

II.3.1 Virgule fixe ou flottante 34

II.3.2 Performance des DSP 37

II.3.3 Puissance de calcul d'un DSP 37

II.3.4 Mesure du temps d'exécution (Benchmark) 39

II.4 Architecture fonctionnelle 39
II.5 Les DSP de la famille TMS320 41
II.6 Choix du DSP pour notre application 43
II.6.1 La famille TMS320C54x 44
II.6.2 La famille TMS320C67x 46
Conclusion 48

CHAPITRE III

IMPLEMENTATION SUR DSP DU FILTRAGE HOMOMORPHIQUE

III.1. Analyse homomorphique sur Matlab 49
III.1.1 Données de traitement 49
III.1.2 Liftrage 50
III.2 Implémentaion du liftrage sur DSP 51
III.2.1 Fichiers d'application 51
III.2.2 Traitement par DSP 52
III.3 Utilisation du filtrage Mel 53
III.4 Analyse MFCC 56
III.5 Optimisation de l'application 60
III.5.1 Profileur du CCS 60
III.5.2 Optimisation du code 61
III.5.3 Démarches et précautions à prendre lors des relevés statistiques 61
III.5.4 Résultas du profileur 62
III.5.5 Remplacement de la bibliothèque standard 65
Conclusion 70
CONCLUSION GENERALE 71
REFERENCES 72
ANNEXE A 74
ANNEXE B 79

LISTE DES FIGURES

CHAPITRE 1

Figure I.1 : représentation d'un système de convolution homomorphique 6
Figure I.2 : Opération linéaire entre opération de déconvolution et son inverse 7
Figure I.3 : représentation d'un système de convolution homomorphique dans le domaine fréquentiel 7
Figure I.4 : Représentation du système de déconvolution homomorphique 8
Figure I.5 : Représentation du système inverse de déconvolution homomorphique 8
Figure I.6 : Pré-traitement et extraction des paramètres 10
Figure I.7 : Forme d'onde de la voyelle 'A' avec fenêtre Hamming 11
Figure I.8 : Fenêtre de pondération de Hamming 14
Figure I.9 : calcul du Cepstre 14
Figure I.10 : illustration des cepstres, réel et complexe 15
Figure I.11 : détection du pitch par le cepstre, et liftrage pour formants et excitation 15
Figure I.12 : équivalence entre modèle physique et modèle mathématique 17
Figure I.13 : Modèle simplifié de production de la parole, modèle tout-pôle 17
Figure I.14 : illustré de cepstre réel liftré et pondéré par les coefficients LPC 20
Figure I.15 : illustré de spectre de la parole et spectre lissé par LPC (bleu), LPCC n°1 voicebox (vert), et LPCC n°2 (noir) 21
Figure I.16 : Calcul des coefficients MFCC 22
Figure I.17 : Banc de filtres Mel 23
Figure I.18 : illustré du Banc de filtres (MEL-MFCC) et Banc de filtres (voicebox) 24
Figure I.19 : les coefficients MFCC 24
Figure I.20 : l'enveloppe spectrale lissée, FFT(rouge), LPC(bleu), liftrage(vert), notre_MFCC (noir) 25
Figure I.21 : l'enveloppe spectrale lissée par MFCC issu du banc de filtres Voicebox (noir) 26
Figure I.21 : chronogramme (en haut), et Spectrogramme (en bas) de la voyelle « A » 27

CHAPITRE 2

Figure II.1 : chaîne complète typique d'un système de traitement numérique du signal 29
Figure II.2 : Place du DSP vis-à-vis des autres processeurs 33
Figure II.3 : Evolution du temps d'exécution d'une opération MAC, selon Texas Instruments 38
Figure II.4 : Architecture de Harvard 41
Figure II.5 : Constitution interne du DSP TMS320C6711 47

CHAPITRE 3

Figure III.1 : phonème A (haut à gauche), segmentation et fenêtrage (haut à droite), module du spectre (bas à gauche), et son logarithme (bas à droite) 49
Figure III.2 : cepstre et fenêtrage passe-bas (haut à gauche) et passe-haut (haut à droite), liftrage des formants (bas à gauche), et liftrage de l'excitation (bas à droite) 50
Figure III.3 : segment d'entrée fenêtré (à gauche), module du spectre (à droite) 52
Figure III.4 : log du module du spectre (à gauche), cepstre réel (à droite) 52
Figure III.5 : liftrage passe-bas en quéfrence (à gauche), et en fréquence (à droite) 52
Figure III.6 : liftrage passe-haut en quéfrence (à gauche), et en fréquence (à droite) 53
Figure III.7 : espacement linéaire des filtres jusqu'à 1 kHz et logarithmique après 54
Figure III.8 : banc à 22 filtres MEL 55
Figure III.9 : Etalement fréquentiel des 22 filtres MEL, le dernier gabarit est représenté seul au tracé du bas 56
Figure III.10 : les 22 coefficients MFCC interpolés à l'aide de la fonction spline 57
Figure III.11 : spectre en fréquence (rouge) confronté aux traitements par liftrage (vert) et filtrage mel interpolé par spline (en noir) 57
Figure III.12 : isolation du spectre par filtrage mel (interpolé par spline) 58
Figure III.13 : MFCC (en haut), |FFT| en dB et spectre melcep en pointillé (au milieu), superposition |FFT| en dB et melcep (en bas) 59
Figure III.14 : fenêtre statistique du Profileur sans optimisation du compilateur ni l'équation ym 63
Figure III.15 : Profileur avec optimisation de l'équation ym mais pas le compilateur 63
Figure III.16 : Profileur avec optimisation du compilateur mais pas l'équation ym 64
Figure III.17 : Profileur avec optimisation du compilateur et de l'équation ym 64
Figure III.18 : Profileur sous fastrts67x.lib sans optimisation du compilateur ni l'équation ym 66
Figure III.19 : Profileur sous fastrts67x.lib avec optimisation de l'équation ym mais pas le compilateur 66
Figure III.20 : Profileur sous fastrts67x.lib avec optimisation du compilateur mais pas l'équation ym 67
Figure III.21 : Profileur sous fastrts67x.lib avec optimisation du compilateur et de l'équation ym 67
Figure III.22 : Environnement CCS montrant le dernier profilage d'exécution 79

Introduction générale

La communication verbale est la méthode la plus utilisée et la plus facile pour transmettre l'information et les connaissances entre humains. Elle est effective car dans un langage donné, on utilise les mêmes composants de base : phonèmes (sons), lexique (mots) et syntaxe (règles entre les mots). Les avantages de la parole sur d'autres moyens de communication sont divers; comme sa flexibilité par rapport à l'environnement, où l'on peut parler dans l'obscurité, à distance, ou pendant que l'on exécute d'autres tâches. La parole peut véhiculer des émotions, etc…

En dehors de la formulation et compréhension d'un message verbal dans un processus de communication, nous allons nous concentrer sur l'information acoustique transmise dans la voix (considérée comme un signal complexe).

Parler est la capacité d'effectuer de manière coordonnée et contrôlée, des mouvements d'organes du tracte vocal (glotte, cordes vocales, langue, lèvres). Ces organes génèrent et modifient l'air dans les passages respiratoires pour produire les sons de la parole. Les sons produits sont :

- les vibrations quasi-stationnaires à partir de mouvements périodiques des cordes vocales,
- les sons transitoires (similaires au bruit) produits par une semi-fermeture de la cavité orale,
- et les segments courts dont l'origine est une relâche subite d'air après une obstruction totale de la cavité orale.

La complexité de la structure des sons de la parole n'est pas due seulement à la position des articulateurs pendant sa génération, mais aussi aux caractéristiques du conduit vocal, dont la configuration (longueur, volume) change avec les mouvements de ses organes en fonction des sons que l'on veut produire. Le flux d'air produit est modulé alors dans la glotte et le conduit vocal détermine la qualité de la voix « son timbre ».

Un modèle simple du conduit vocal est une combinaison des tubes de différentes longueurs et volumes. Chaque tube subit différentes pressions d'air, ce qui produit un mouvement très complexe des molécules d'air. Dans ce cas, une description linéaire échoue pour décrire la dynamique du système. Les démarches non-linéaires semblent offrir de bonnes approximations au problème.

Le but de ce travail est l'implémentation sur un DSP, du filtrage homomorphique de la parole réalisant une déconvolution entre le signal d'excitation et la réponse impulsionnelle du conduit vocal, ce qui permet par la suite d'aboutir aux cepstres, complexe et réel, de la parole.

Une implémentation en parallèle sur MATLAB est à réaliser afin de vérifier les résultats obtenus par DSP. Ce dernier étant essentiellement utilisé pour assurer une exécution en temps réel des différents traitements à courts termes de la parole.

Ce mémoire regroupe trois chapitres, dont les contenus sont comme suit :

Le chapitre I présente une modélisation et analyse du signal de la parole, accompagnées de simulation des principales méthodes de paramétrisation, utilisées pour déterminer les coefficients cepstraux, formants, et pitch.

Dans le chapitre II, nous exposons notre choix du DSP et la présentation de l'outil de développement, en mettant l'accent sur les DSP de la famille de *Texas Instruments*, en particulier la famille TMS320C67x à virgule flottante.

Le chapitre III présente quant à lui, les résultats de notre travail et une étude comparative des résultats fournis par MATLAB et ceux fournis par le DSP.

CHAPITRE I

MODELISATION ET ANALYSE DU SIGNAL DE LA PAROLE

I.1 Introduction

Une fois produit, le signal de la parole est représenté de diverses manières dans les ordinateurs. Il est d'abord numérisé puis traité pour extraire ses principales caractéristiques. Les représentations paramétriques de la parole sont basées normalement sur des analyses de courte durée (segments de 10-30 ms), où le signal est considéré stationnaire [1].

Dans le domaine temporel, les analyses les plus courantes sont la détection de Pitch, le calcul d'énergie, le nombre de passages par zéro et la fonction d'autocorrélation. Dans le domaine spectral, on utilise un banc de filtres ou la transformée rapide de Fourier. Mais il y a aussi l'analyse homomorphique (cepstrum) de la parole qui essaie de séparer la source d'excitation de la configuration du conduit vocal. Un autre traitement possible est la prédiction linéaire, qui considère que la parole est la sortie d'un filtre variable dont la fonction de transfert approche les propriétés spectrales du conduit vocal et des impulsions glottales. Néanmoins, il y a une grande quantité de traitement qui ne sont pas considérés dans les paramètres décrits : la dynamique de la source transitoire pendant la production des consonnes, la variation des fréquences formantiques pendant la production des voyelles, la variation du débit de la parole, de l'énergie et de la fréquence fondamentale.

Le signal de la parole est donc un signal où plusieurs facteurs interviennent dans sa production. La représentation paramétrique que nous utilisons dans nos calculs est multidimensionnelle et elle représente de manière approximative des phénomènes non-linéaires complexes.

I.2 Le signal de la parole

Le signal de la parole appartient à la classe des signaux acoustiques produits par des vibrations des couches d'air. Les fluctuations de la pression de l'air produisent des variations de ce signal, en fonction du temps, qui peuvent être enregistrées de façon analogique ou digitale. Ceci constitue une représentation élémentaire du signal de la parole [2]. Ce signal est le résultat du passage du flux laryngé (l'air qui la passe par nos poumons à travers nos cordes vocales) à travers le conduit vocal. Le signal de la parole est donc une onde acoustique qui se propage dans un milieu donné (en général l'air) et qui est le résultat de la modulation par le conduit vocal d'une onde d'excitation.

Les phonèmes sont les éléments les plus brefs qui permettent de distinguer différents mots [3]. Un mot peut être considéré comme un ensemble de phonèmes. Plusieurs formes du conduit vocal peuvent produire le même phonème. La forme que le conduit vocal prend pour la production d'un phonème, dans un contexte donné, est assez variable et surtout dépendante de ce contexte. De ce fait, nous remarquons que les formes acoustiques associées à un phonème déterminé sont variables. Cette variabilité est double : d'une part une variabilité du contenu acoustique du phonème (dû à la variété des formes du conduit vocal donnant lieu à ce phonème) et d'autre part, une variabilité de la durée du phonème. Cette dernière variabilité résulte essentiellement du fait que le système articulatoire met en jeu des constantes mécaniques qui contrôlent les mouvements musculaires.

L'étude des mécanismes les plus répandus chez l'homme, ont montré que l'analyse des mouvements des différents paramètres articulatoires peut être représentée par un modèle de type source-filtre. Deux filtres placés en cascade constituent le modèle de chaque paramètre articulatoire.

Le premier est un filtre intégrateur, il est associé au système mécanique de la chaîne de production de la parole. Le deuxième est un intégrateur d'ordre 2 et il correspond au système physiologique. La source, quant à elle, est représentée par une séquence d'impulsions [4].

La parole est un signal quasi stationnaire. Elle est formée de phonèmes et de transitions entre ces phonèmes (bien que le phonème ne soit pas une entité acoustiquement fixe). Plusieurs types de phonèmes existent : les voyelles, les consonnes fricatives et les consonnes plosives, les nasales et les liquides. Les voyelles sont des phonèmes voisés (l'excitation se fait par la glotte), leur production se fait généralement avec un conduit vocal relativement ouvert et en absence de constrictions et leur prononciation peut être isolée et durable dans le temps. Les consonnes se caractérisent par une constriction (ou occlusion) dans le conduit vocal lors de leur production. Elles peuvent être voisées ou non voisées. Dans le cas des fricatives, la constriction génère un bruit local qui peut persister dans le temps et qui excite une partie du conduit vocal. Contrairement aux voyelles et aux fricatives, les plosives ne durent pas dans le temps, elles sont produites par un relâchement rapide d'une occlusion du conduit vocal, qui produit une perturbation locale se traduisant acoustiquement par un bruit impulsif de faible durée. Des transitions lient les phonèmes adjacents. D'une façon très simplifiée, les transitions acoustiques correspondent à des transitions dans l'appareil de production de l'état correspondant au premier phonème à l'état correspondant au suivant [5].

En conclusion, la parole est un signal quasi stationnaire formé de parties stationnaires et de transitions entre ces différentes parties. C'est un signal non déterministe, dans le sens où deux réalisations d'un même mot auront nécessairement deux formes acoustiques différentes, même si elles sont produites par un même locuteur. Il faut distinguer deux types de variations : la variation acoustique et la variation temporelle non linéaire du signal.

I.3 Les paramètres de la parole

I.3.1 Le fondamental

La vitesse à laquelle les cordes vocales s'ouvrent et se referment lors du processus de phonation, produit une vibration d'une hauteur variable appelée fréquence du fondamental dont la valeur est étroitement liée à la taille de l'appareil phonatoire de la personne, Cette fréquence est quasi stationnaire pour un signal de type voisé, elle varie de [6]:

. de 80 à 200 Hz pour une voix masculine,

. de 250 à 450 Hz pour une voix féminine,

. de 200 à 600 Hz pour une voix d'enfant.

Deux sons de même intensité et de même hauteur se distinguent par le timbre, qui est déterminé par les harmoniques du fondamental [5]. Un intérêt majeur pour la fréquence du fondamental se trouve dans les applications de la synthèse de parole.

I.3.2 Les formants

Le spectre du signal vocal résultant de l'action des sources de sons sur le conduit vocal présente des maximums et des minimums qui correspondent aux résonances et aux anti résonances du conduit vocal, appelés formants et anti-formants. Du point de vue perceptif, seul les trois premiers formants jouent un rôle essentiel pour caractériser le spectre vocal [5]. On peut caractériser toute voyelle en n'utilisant que ses trois premiers formants. En général la fréquence du premier formant varie de 200 à 900 Hz, celle du second de 500 à 2500 Hz et le troisième se situe entre 1500 et 3500 Hz. Des formants d'ordre supérieur existent même si leur rôle sur le plan perceptif est limité, ils contribuent à caractériser la voix.

I.4 Traitement homomorphique de la parole

I.4.1 Systèmes de convolution homomorphique

Les Systèmes de convolution homomorphique obéissent à un principe généralisé de superposition [7]. Le principe de superposition comme il est normalement exprimé par les systèmes conventionnels linéaires est donné par :

$$\begin{aligned} L\,[x(n)] &= L[x_1(n) + x_2(n)] \\ &= L[x_1(n)] + L[x_2(n)] \\ &= y_1(n) + y_2(n) = y(n) \end{aligned} \tag{I.1.a}$$

Et

$$L[ax(n)] = aL[x(n)] = ay(n) \tag{I.1.b}$$

Où L représente l'opérateur linéaire. Le principe de superposition stipule simplement que si un signal d'entrée est composé d'une combinaison linéaire de signaux élémentaires, alors la sortie est une combinaison linéaire des sorties correspondantes.

Une conséquence directe du principe de superposition est le fait que la sortie $y(n)$ d'un système linéaire $h(n)$ invariant dans le temps, peut être exprimée comme la somme de convolution

$$y(n) = \sum_{k=-\infty}^{\infty} h(n-k)x(k) = h(n) * x(n) \tag{I.2}$$

Nous pouvons définir une classe de systèmes qui obéissent à un principe généralisé de superposition, où l'addition est remplacée par la convolution, figure I.1.

$$\begin{aligned} D_*^{-1}\,[x(n)] &= D_*^{-1}\,[x_1(n) + x_2(n)] \\ &= D_*^{-1}\,[x_1] * D_*^{-1}\,[x_2] \\ &= y_1(n) * y_2(n) = y(n) \end{aligned} \tag{I.3}$$

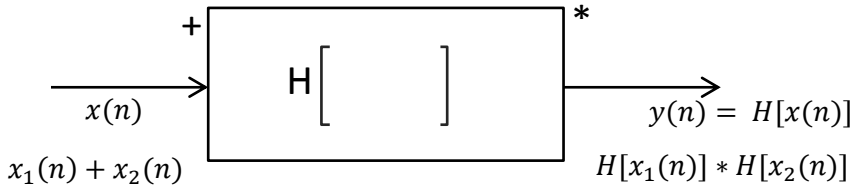


Figure I.1 : représentation d'un système de convolution homomorphique

Ou bien, l'opération inverse (déconvolution) :

$$D_*[x(n)] = D_*[x_1(n) * x_2(n)]$$
$$= D_*[x_1(n)] + D_*[x_2(n)] \quad \text{(I.4)}$$
$$= \hat{x}_1(n) + \hat{x}_2(n) = \hat{x}(n)$$

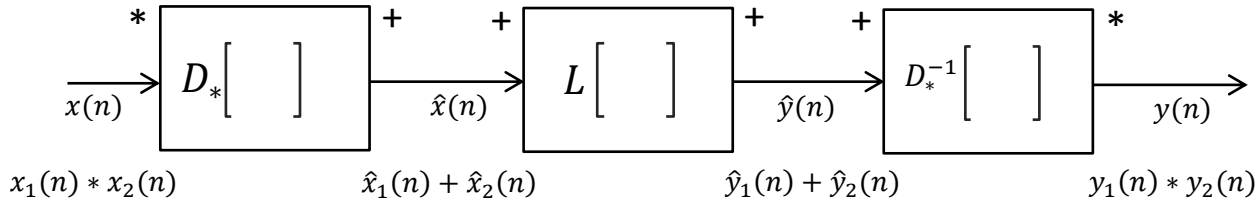


Figure I.2 : Opération linéaire entre opération de déconvolution et son inverse

La caractéristique de système inverse D_*^{-1} appliquée en figure I.2 donne :

$$D_*^{-1}[\hat{y}(n)] = D_*^{-1}[\hat{y}_1(n) + \hat{y}_2(n)]$$
$$= D_*^{-1}[\hat{y}_1(n)] * D_*^{-1}[\hat{y}_2(n)] \quad \text{(I.5)}$$
$$= y_1(n) * y_2(n) = y(n)$$

exemple :

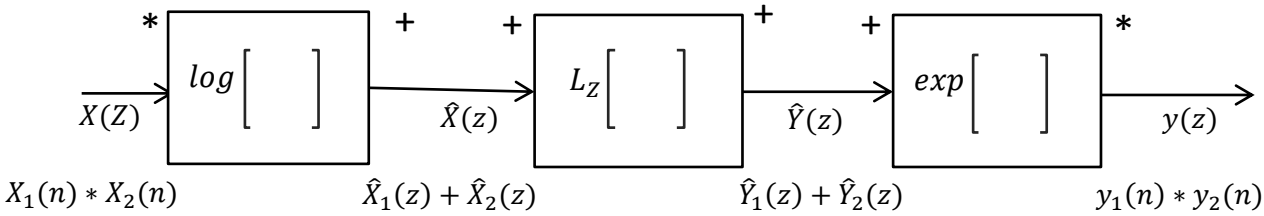


Figure I.3 : représentation d'un système de convolution homomorphique dans le domaine fréquentiel

Si l'entrée est une convolution de deux variables discretes :

$$x(n) = x_1(n) * x_2(n) \quad \text{(I.6)}$$

La transformée en z de l'entrée est le produit des transformées en z correspondants.

$$X(z) = X_1(z).X_2(z) \quad \text{(I.7)}$$

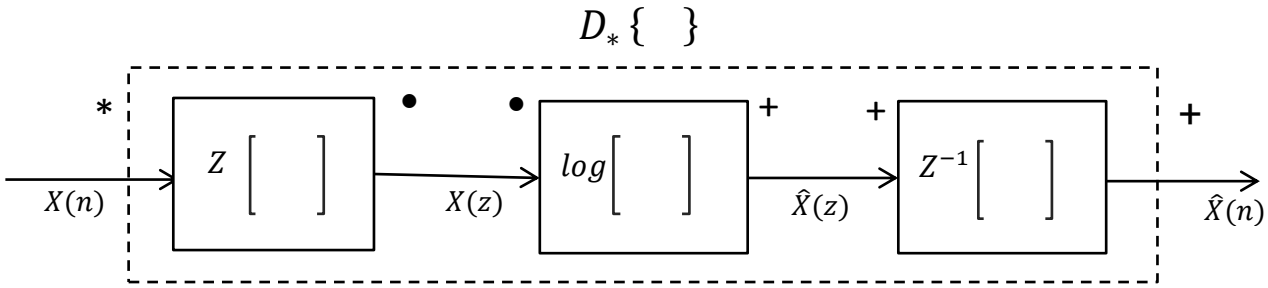


Figure I.4 : Représentation du système de déconvolution homomorphique

A Partir de l'équation (I.4), il est clair que la transformée en z de la sortie du système caractéristique doit être une combinaison additive de transformée z. Ainsi, le comportement dans le domaine fréquentiel du système de convolution doit avoir la propriété que, si un signal est représenté par un produit de transformée z à l'entrée, la sortie doit être une somme de sortie correspondant en transformée z.

Une approche pour la représentation d'un tel système est représentée sur la figure I.3. Cette approche est basée sur le fait que le logarithme d'un produit peut être défini de manière qu'il soit égal à la somme des logarithmes des termes composants. C'est-à-dire :

$$\hat{X}(z) = \log[X(z)] = \log[X_1(z).X_2(z)]$$

$$= \log[X_1(z)] + \log[X_2(z)] \qquad \text{(I.8)}$$

Si nous voulons représenter des signaux sous forme de séquences, plutôt que dans le domaine fréquentiel comme dans la figure I.3, alors le système peut être représenté comme le montre la figure I.4. De même, l'inverse de la caractéristique du système peut être représenté comme à la figure I.5.

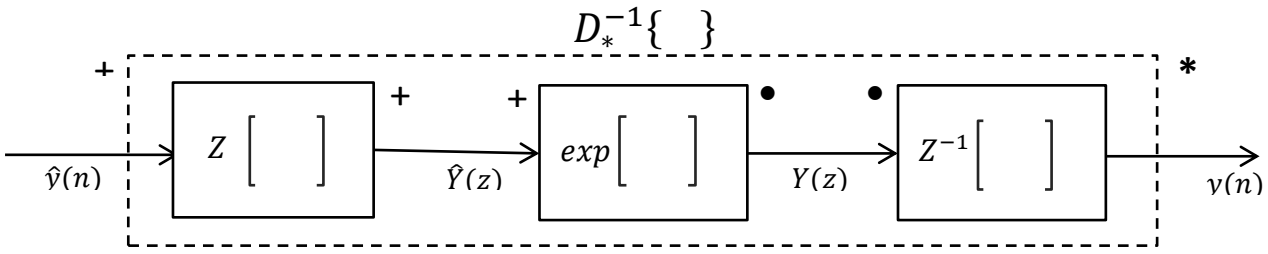


Figure I.5 : Représentation du système inverse de déconvolution homomorphique

La représentation du système et de son inverse comme représenté sur les figures I.4 et I.5, respectivement, dépend de la validité de l'équation (I.8). Autrement dit, le logarithme doit être défini afin qu'il ait la propriété que le logarithme d'un produit est égal à la somme des logarithmes. C'est vrai pour les quantités réelles positives. Cependant, la transformée en z est en général une quantité complexe et il existe d'importantes considérations de caractère unique lorsqu'il s'agit du logarithme d'un nombre complexe. L'équation (I.8) est valable lors de l'évaluation sur le cercle unité, c'est à dire, pour $z = e^{jw}$. Pour notre cas, il suffit d'indiquer qu'une définition appropriée du logarithme complexe est :

$$\hat{X}(e^{jw}) = log|X(e^{jw})| + j \arg[X(e^{jw})] \quad (I.9)$$

Dans cette équation, la partie réelle ($log|X(e^{jw})|$) ne provoque pas de difficulté particulière. Cependant, des problèmes d'unicité se posent dans la définition de la partie imaginaire ($\arg[X(e^{jw})]$), qui est simplement l'angle de phase de la transformée en z évaluée sur le cercle unité. Dans la référence [8], il est démontré qu'une approche pour faire face aux problèmes de l'unicité de l'angle de phase est d'exiger que l'angle de phase est une fonction continue impair de w. Avec cette condition, l'équation (I.8) est satisfaite.

Étant donné qu'il est possible de calculer le logarithme complexe de façon à satisfaire l'équation (I.8), la transformation inverse du logarithme complexe de la transformée de Fourier de l'entrée est la sortie du système de convolution, à savoir,

$$\hat{x}(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \hat{X}(e^{jw}) e^{jwn} dw \quad (I.10)$$

La sortie du système caractéristique, x (n), est appelé le "cepstre complexe» (Le terme «cepstre» a été introduit par Bogert et al. [9] et a fini par être acceptée comme terminologie de la transformée de Fourier inverse du logarithme du spectre de puissance d'un signal. Le terme «cepstre complexe» implique que le logarithme complexe est impliqué. Nous allons utiliser le terme «cepstre" pour la quantité :

$$c(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} log|X(e^{jw})| e^{jw}\, dw \quad (I.11)$$

(La séquence $c(n)$ peut être montrée qu'elle est égale à la partie réelle du cepstre complexe $\hat{x}(n)$.

Dans la discussion ci-dessus, nous avons défini la caractéristique de système de convolution homomorphique et donc défini la forme canonique pour tous les systèmes de convolution

homomorphique. Tous les systèmes de cette classe ne diffèrent que par la partie linéaire du système. Le choix du système linéaire dépend nécessairement des propriétés de ses signaux d'entrée.

I.5 Pré-traitement acoustique

Le calcul des paramètres acoustiques passe par une phase de pré-traitement contenant deux étapes, la pré-accentuation acoustique et le fenêtrage.

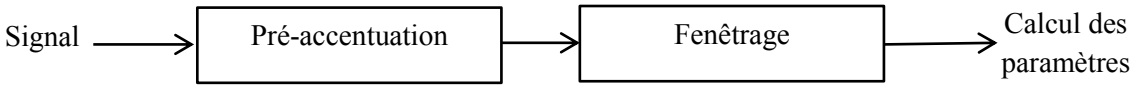


Figure I.6 : Pré-traitement et extraction des paramètres

I.5.1 La pré-accentuation du signal de la parole

Le signal est tout d'abord filtré puis échantillonné à une fréquence donnée. Une pré-accentuation est effectuée afin de relever les hautes fréquences, ensuite le signal est segmenté en trames. Chaque trame est constituée d'un nombre fixe N d'échantillons de parole. En général, N est fixé de telle manière que chaque trame corresponde à environ 30ms de parole (durée pendant laquelle la parole peut être considérée comme stationnaire). Enfin, le fait de traiter un petit morceau de signal amène des problèmes dans le filtrage (effets de bord). Pour éviter cela, nous utilisons des fenêtres de pondération.

I.5.2 Le fenêtrage

Ce sont des fonctions que l'on applique à l'ensemble des échantillons prélevés dans la fenêtre du signal original de façon à diminuer les effets de bords. Parmi les fenêtres les plus courantes, nous pouvons citer la fenêtre de Hamming.

La fenêtre de Hamming est donnée par l'expression :

$$w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right) \quad , \; 0 \le n \le N-1 \tag{I.12}$$

N : Le nombre d'échantillons dans une fenêtre.

En général, les fenêtres successives se recouvrent et elles doivent avoir une longueur suffisante. Ce traitement implique une hypothèse importante : Le signal vocal est supposé stationnaire sur une courte période.

La figure I.7 illustre les effets du fenêtrage sur une trame de parole.

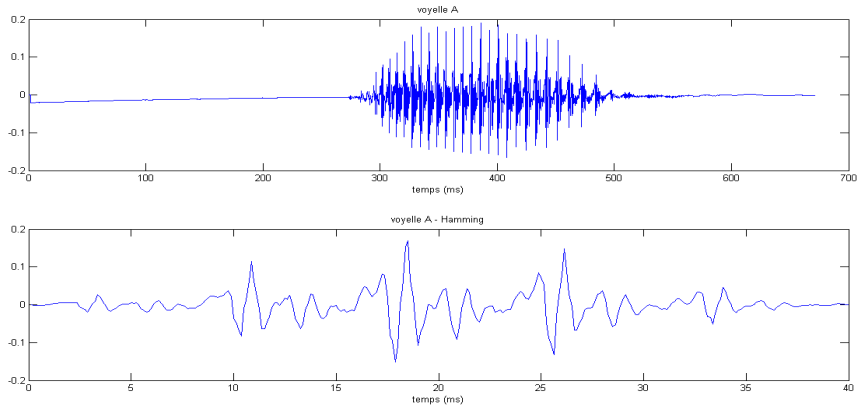


Figure I.7 : Forme d'onde de la voyelle 'A' avec fenêtre Hamming

Après cette mise en forme du signal (commune à la plupart des méthodes d'analyse de la parole), une transformée de Fourier discrète (DFT : Discret Fourier Transform), en particulier FFT (Transformée de Fourier Rapide), est appliquée pour passer dans le domaine fréquentiel et représentes le spectre du signal.

I.6 La transformée de Fourier rapide (FFT)

Au cours de cette étape chacune des trames, de N valeurs, est convertie du domaine temporel au domaine fréquentiel. La FFT est un algorithme rapide pour le calcul de la transformée de Fourier discrète (DFT) et est définie par la formule (I.13). Les valeurs obtenues sont appelées le spectre.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{\frac{-2j\pi}{N}kn} \quad , \quad 0 \leq k \leq N-1 \qquad \text{(I.13)}$$

En général, les valeurs $X[k]$ sont des nombres complexes et nous n'utilisons que leurs valeurs absolues dans le calcul d'énergie en fréquence.

I.7 Analyse et paramétrisation du signal de la parole

L'analyse acoustique du signal de la parole consiste à extraire l'information pertinente et à réduire au maximum la redondance. Généralement, on calcule un jeu de coefficients acoustiques à des intervalles de temps réguliers, sur des blocs de signal de longueur fixe. Ce jeu de coefficients constitue un vecteur acoustique. Les techniques de paramétrisation acoustique sont nombreuses. Néanmoins, on peut les regrouper en trois :

- Analyse cepstrale.
- Analyse par prédiction linéaire.
- Analyse par bancs de filtres.

L'objectif de notre travail de simulation est d'identifier les paramètres du signal de la parole tel que (pitch, formant…). Pour cela, on utilise des algorithmes sur MATLAB, en premier lieu et sur DSP en second lieu.

I.7.1 Analyse cepstrale

Le Cepstre d'un signal x[n] est une transformation de ce signal du domaine temporel vers un autre domaine temporel, celui des quéfrences. L'idée consiste à considérer un signal périodique donné comme la convolution d'un train d'impulsions par un filtre.

Dans le domaine des fréquences, les spectres sont multipliés: si on prend le logarithme du résultat, on obtient la somme des résultats indépendants. Ainsi, une convolution dans l'espace des temps correspond à une addition dans le domaine du cepstre: Si les deux spectres ont des caractéristiques différentes, il devient possible de les séparer. Selon que l'on prenne le logarithme réel ou complexe, on distingue [10]: le cepstre réel et le cepstre complexe.

I.7.1.1 Le cepstre réel

Le cepstre est obtenu en calculant le logarithme du module de la transformée de Fourier du signal. Puis en prenant la transformée de Fourier inverse du résultat.

$$c(n) = FFT^{-1}(log|FFT(x_n)|\,) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log(\left|X(e^{jw})\right|e^{jw}\,dw \qquad \text{(I.14)}$$

I.7.1.2 Le cepstre complexe

Le cepstre complexe est obtenu en calculant le logarithme complexe défini Comme :

$$\log(z) = \log(|z|) + j.\arg(z)$$

$$\hat{x}(n) = FFT^{-1}\,(\log(FFT(x_n))) = \frac{1}{2\pi}\int_{-\pi}^{\pi} \log(\hat{X}(e^{jw}))e^{jwn}dw \qquad \text{(I.15)}$$

I.7.1.3 Propriétés

Le Cepstre vérifie les propriétés suivantes:

1. Si le signal x possède une transformée en Z dont le domaine de convergence comprend le cercle unité, soit de la forme:

$$X(z) = \frac{A\prod_{k=1}^{N_0}(1 - a_k * z^{-1}) \times \prod_{k=1}^{N_1}(1 - b_k * z)}{\prod_{k=1}^{D_0}(1 - c_k * z^{-1}) \times \prod_{k=1}^{D_1}(1 - d_k * z)}$$

Avec : (I.16)

$$|a_i| < 1, |b_i| < 1, |c_i| < 1, |d_i| < 1$$

Alors son cepstre complexe s'écrit sous la forme suivante :

$$\hat{x}(n) = \begin{cases} \log(A) & n = 0 \\ \sum_{k=1}^{D_0}\left(\frac{c_k^n}{n}\right) - \sum_{k=1}^{N_0}\left(\frac{d_k^{-n}}{n}\right) & n > 0 \\ \sum_{k=1}^{D_1}\left(\frac{b_k^{-n}}{n}\right) - \sum_{k=1}^{N_1}\left(\frac{d_k^{-n}}{n}\right) & n < 0 \end{cases} \quad \text{(I.17)}$$

2. Le cepstre de la réponse impulsionnelle d'un filtre causal peu résonant est localisé autour de zéro.

3. En appliquant la propriété (2) avec $a_k = b_k = d_k = 0\ et\ c_k = ce^{2j\pi k/p}$ pour $0 \leq k < p$,le cepstre complexe de la réponse impulsionnelle d'un filtre en peigne $H(z) = 1/(1 - c^p z^p)$ est un peigne d'impulsions d'amplitudes localisées à des valeurs temporelles espacées de p. La quéfrence fondamentale du cepstre est donc égale à la période fondamentale du signal [9].

-Enregistrement des données sonores

La phase d'enregistrement concerne la conversion du signal continu de la parole en une suite ou succession d'impulsions normalisées, à une fréquence de 8KHz. Cette fréquence d'échantillonnage est suffisante pour prendre en compte la variabilité du signal vocal enregistré. Nous opérons pour chaque exemple à partir d'un signal de la parole enregistré sous format **.wav**, avec une fréquence d'échantillonnage de 8kHz et codage sur 16 bits.

L'analyse cepstrale suppose que le signal vocal s(n) est produit par un signal excitateur e(n) qui traverse un système linéaire passif (conduit vocal) de réponse impulsionnelle h(n).

Puisque le signal vocal est non stationnaire, on effectue un fenêtrage par la fenêtre de Hamming, figure.I.8.

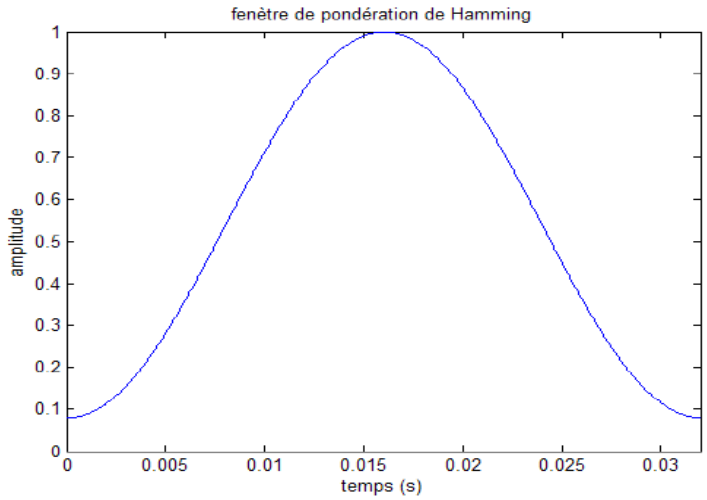


Figure I.8 : Fenêtre de pondération de Hamming

Pour extraire les coefficients cepstraux, on effectue les opérations de la figure.I.9.

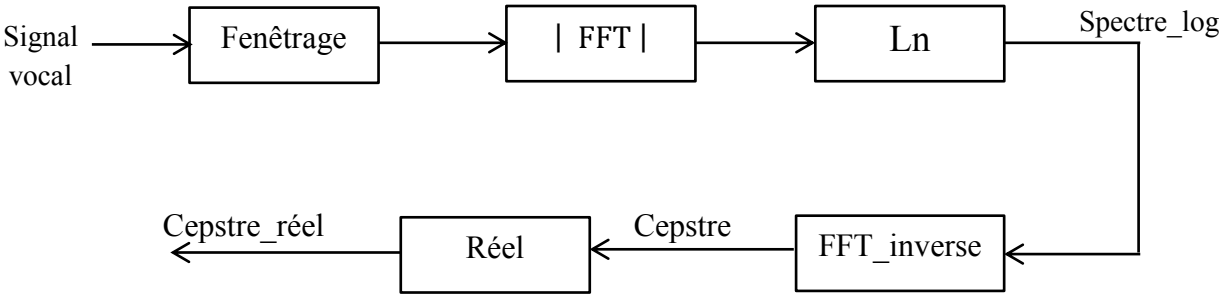


Figure I.9 : calcul du Cepstre

Ces opérations sont effectuées à l'aide des fonctions prédéfinies de logiciel MATLAB (**fft, log, ifft, abs**). Enfin, un ensemble de représentations graphiques permet de visualiser le cepstre réel, le cepstre complexe et l'enveloppe spectrale. De là, nous pouvons estimer les paramètres de la parole.

La figure I.10 présente les cepstres, réel et complexe, issus de l'analyse cepstrale d'un signal de la parole représentant la voyelle 'a'.

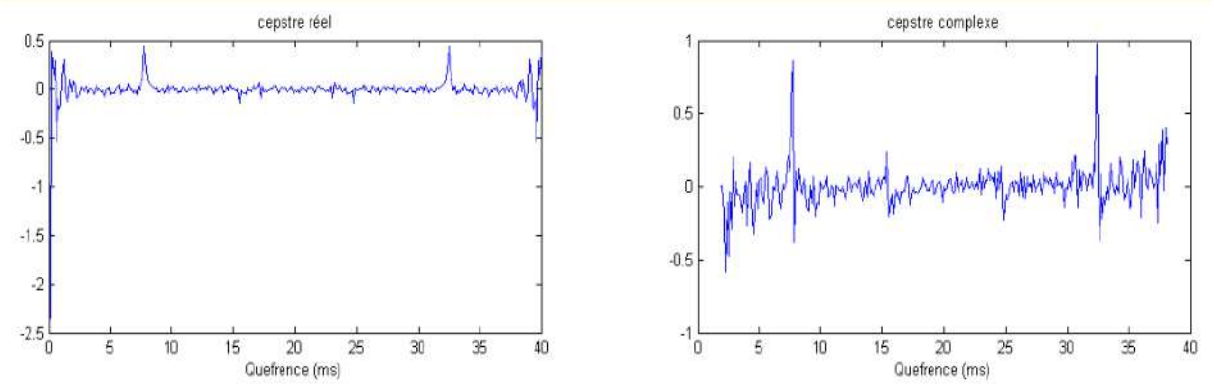


Figure I.10 : illustration des cepstres, réel et complexe

I.7.1.4 Liftrage

Pour avoir l'information sur les formants, on a filtré le cepstre par un filtre rectangulaire (opération dite liftrage). En prenant le filtre complémentaire, on peut extraire l'information sur l'excitation périodique de la voyelle. Le résultat de l'opération menée sur le fichier audio de la voyelle 'A' est présenté à la figure I.11. Pour obtenir la fréquence du fondamental, la distance du pic central en terme de temps (quéfrence), permet de déterminer la période fondamentale, soit la fréquence cherchée. On trouve donc une fréquence de 1/7.25 = 129 Hz à partir de la figure I.11.

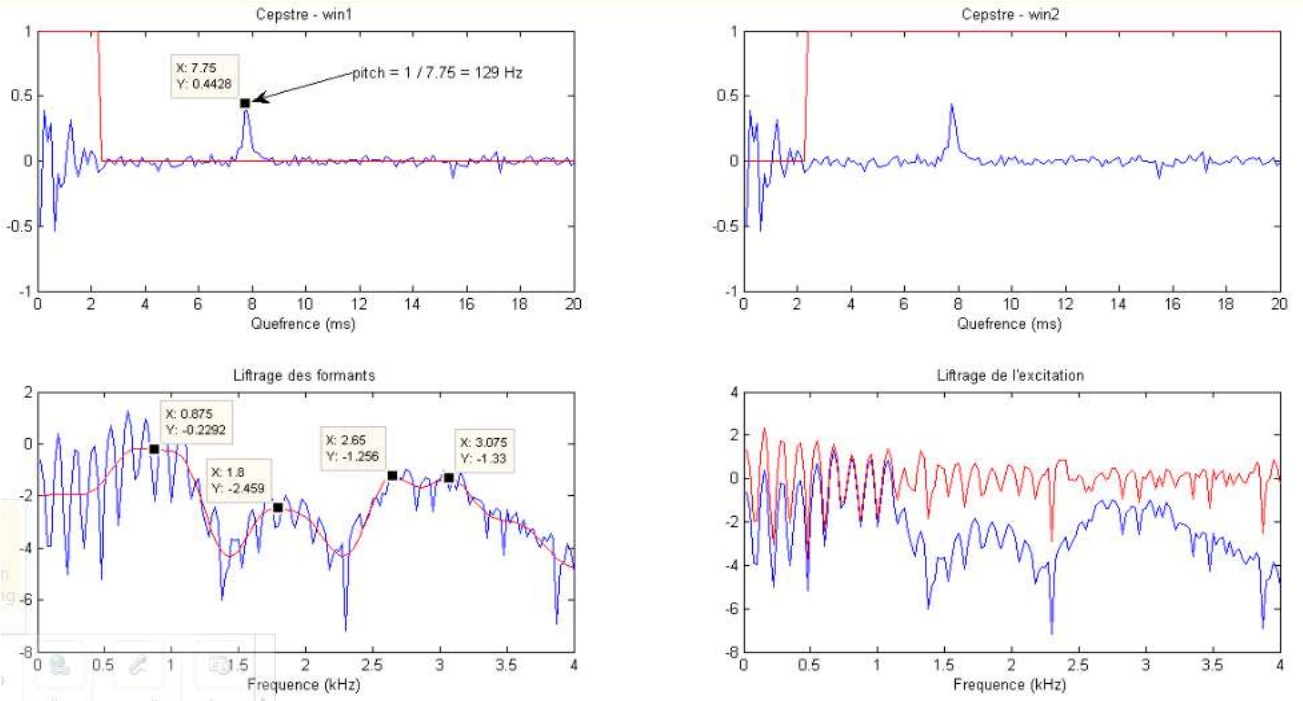


Figure I.11 : détection du pitch par le cepstre, et liftrage pour formants et excitation

La figure.I.11 illustre le liftrage d'une tranche voisée de la voyelle 'a'. Les parties voisées du signal apparaissant sous la forme de successions de pics spectraux, dont les fréquences centrales sont multiples de la fréquence fondamentale. La forme générale de spectre, appelée enveloppe spectrale, présente elle-même des pics et des creux qui correspondent aux résonances du conduit vocal et sont appelés formants. L'évolution temporelle de leur fréquence centrale et de leur largeur de bande détermine le timbre du son.

I.7.2 Analyse par prédiction linéaire

I.7.2.1 Modèle de production de la parole

Fant [11] a proposé en 1960 un modèle de production dont nous résumons ici la version numérique. Un signal voisé peut être modélisé par le passage d'un train d'impulsions *u(n)* à travers un filtre numérique récursif de type tout-pôles (*AR*). On montre que cette modélisation reste valable dans le cas de sons non voisés, à condition que *u(n)* soit cette fois un bruit blanc. Le modèle final est illustré à la Figure I.12 Il est souvent appelé modèle auto régressif, parce qu'il correspond dans le domaine temporel à une régression linéaire de la forme :

$$X(n) = G \cdot u(n) + \sum_{i=1}^{p} - a_i X(n-1) \qquad \text{(I.18)}$$

Et le filtre LPC : $$H(z) = \frac{G}{A(z)} \qquad \text{(I.19)}$$

Où *u(n)* est le signal d'excitation. Ce qui exprime que chaque échantillon est obtenu en ajoutant un terme d'excitation à une prédiction obtenue par combinaison linéaire de *p* échantillons précédents.

Les coefficients du filtre sont d'ailleurs appelés coefficients de prédiction et le modèle AR est souvent appelé modèle de prédiction linéaire.

Les paramètres du modèle AR sont : la période du train d'impulsions (sons voisés uniquement), la décision Voisé / Non Voisé (V/NV), le gain G , et les coefficients du filtre

1/A(z), appelé filtre de synthèse.

Les relations d'équivalence entre le modèle physique et le modèle mathématique sont :

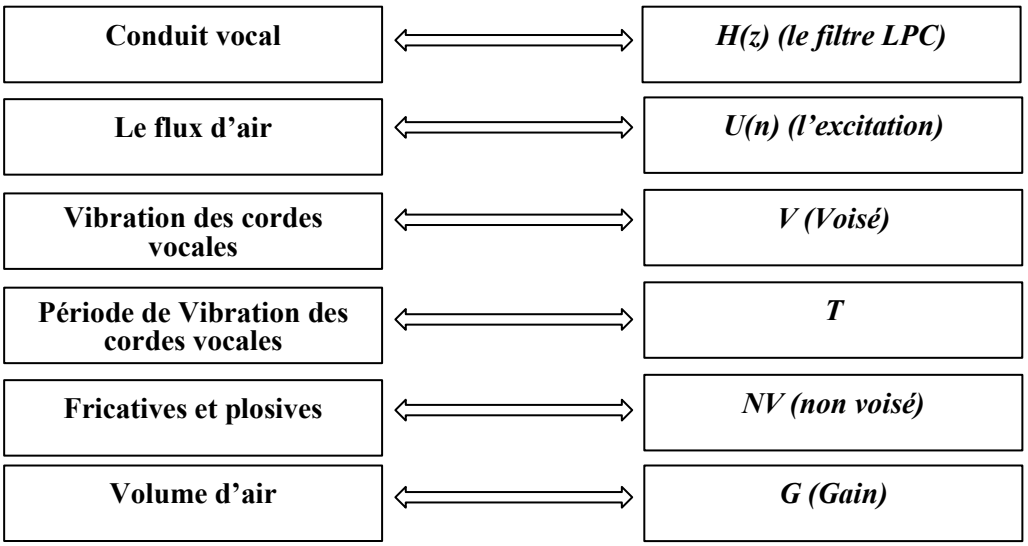


Figure I.12 : équivalence entre modèle physique et modèle mathématique

Le problème de l'estimation d'un modèle AR, souvent appelée analyse *LPC* revient à déterminer les coefficients d'un filtre tout pôles dont on connaît le signal de sortie, mais pas l'entrée. Il est par conséquent nécessaire d'adopter un critère, afin de faire un choix parmi l'infinité de solutions possibles. Le critère classiquement utilisé est celui de la minimisation de l'énergie de l'erreur de prédiction.

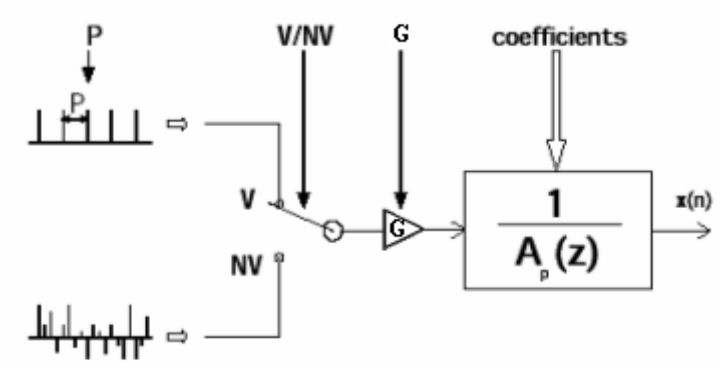


Figure I.13 : Modèle simplifié de production de la parole, modèle tout-pôle

I.7.2.2 Prédiction linéaire avec modèle Pôle-zéro

Le principe fondamental de la prédiction linéaire est qu'un échantillon du signal *S*(n) peut être modélisé comme la sortie d'un système Auto Régressif à Moyenne Ajustée (ARMA) avec une entrée u(n) [12] et [13] :

$$s(n)=\sum_{k=1}^{p} a_k s(n-k)+G\sum_{l=0}^{q} b_l u(n-l), \qquad b_0=1, \qquad \text{(I.20)}$$

Où $\{a_k\},(b_l)$ et le gain G sont les paramètres du système. L'Equation (1.20) prédit la sortie courante en utilisant une combinaison linéaire des sorties antérieures et des entrées courantes et antérieures.

Dans le domaine fréquentiel, la fonction de transfert du modèle de prédiction linéaire de la parole est de la forme :

$$H(z)=\frac{B(z)}{A(z)}=\frac{G[1+\sum_{l=1}^{q} b_l z^{-l}]}{1-\sum_{k=1}^{p} a_k z^{-k}} \qquad \text{(I.21)}$$

Les racines du dénominateur et numérateur sont, respectivement, les pôles et les zéros du système ou modèle pôle-zéro H(z).

Si $a_k=0$ pour $1\le k\le p$, $H(z)$ devient un modèle tout zéro ou modèle à moyenne ajustée (MA). Si $\{b_l=0\}$ pour $1\le l\le q$, $H(z)$ devient un modèle tout pôle ou modèle Auto Régressif (AR) :

$$H(z)=\frac{1}{A(z)} \qquad \text{(I.22)}$$

Dans l'analyse de la parole, les classes de phonèmes comme les fricatives et les nasales contiennent des vallées spectrales qui correspondent aux zéros dans H(z). Par contre, les voyelles contiennent des résonances qui peuvent être modélisées par le modèle tout-pôle. Pour des raisons de simplicité de calcul, ce modèle est préféré pour l'analyse par prédiction linéaire de la parole.

Ainsi, le signal prédit est égal à :

$$s(n) = \sum_{k=1}^{p} a_k s(n-k) \tag{I.23}$$

Et l'erreur de prédiction ou résiduel du signal est la sortie $e(n)$:

$$e(n) = s(n) - \sum_{k=1}^{p} a_k s(n-k) \tag{I.24}$$

Pour avoir les coefficients du filtre court-terme $\{a_i\}$ du processus AR, la méthode classique des moindres carrés peut être utilisée. La variance ou l'énergie, du signal erreur $e(n)$ est minimisée sur une trame de parole.

I.7.2.3 Coefficients cepstraux par prédiction linéaire (LPCC)

Les coefficients LPCC (c_n) sont dérivés directement des coefficients LPC à travers le système d'équations suivant :

$$\begin{cases} c_0 = \ln G \\ c_m = a_m + \sum_{k=1}^{m-1} \left(\frac{k}{m}\right) c_k a_{m-k} \qquad 1 \le m \le p \\ c_m = \sum_{k=1}^{m-1} \left(\frac{k}{m}\right) c_k a_{m-k} \qquad m > p \end{cases} \tag{I.25}$$

Où G est le coefficient de gain du modèle source-filtre.

La méthode utilisée pour l'extraction des formants est basée sur le calcul des pôles de la fonction de transfert du modèle LPC. En effet, La prédiction linéaire est une méthode de déconvolution directe qui sépare le signal périodique d'excitation de la fonction de transfert qui module ce signal. Cette fonction est un modèle tous pôles autorégressif (AR) où chaque pôle correspond aux caractéristiques d'un formant, c'est à dire à sa fréquence et à sa bande passante. Le nombre de pôles de cette fonction donne l'ordre du modèle.

Pour calculer les coefficients de ce modèle, nous avons utilisé des fonctions prédéfinies de la bibliothèque « Voicebox Toolbox » de MATLAB, L'algorithme utilisé est très simple. Il

consiste à déterminer d'abord par la commande **`lpc,`** les coefficients AR puis calculer le spectre LPC.

La figure I.15 illustre le cepstre réel liftré et pondéré par les coefficients LPCC.

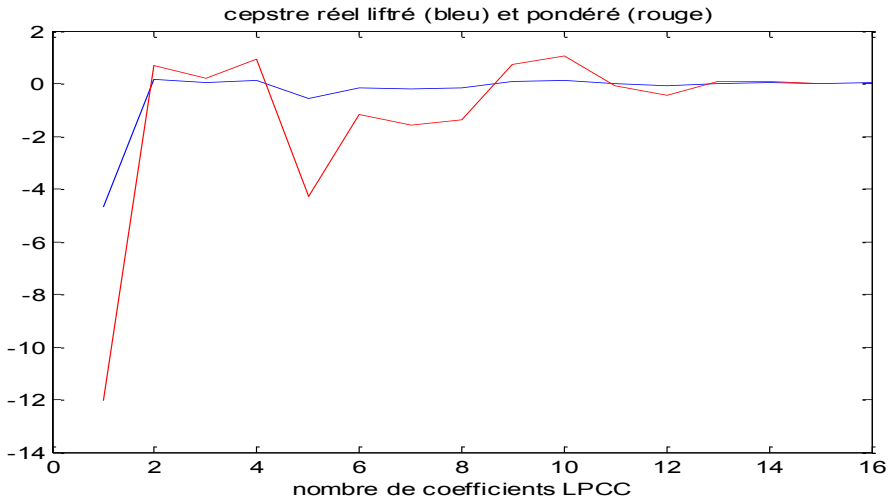


Figure I.14 : illustré de cepstre réel liftré et pondéré par les coefficients LPCC

A travers ces coefficients, nous avons extrait l'enveloppe spectrale LPCC par trois méthodes. La figure.I.14 montre les enveloppes spectrales lissées. en illustrant les différentes enveloppes estimées. Nous avons extrait les spectres par fft(rouge) et l'enveloppe spectrale lissée par LPC (bleu).

La fonction **`lpcar2cc`** de « Voicebox » nous a permis de calculer les coefficients AR, et à travers eux, nous avons extrait l'enveloppe LPCC n°1 (vert).

Nous avons extrait l'enveloppe LPCC n°2 (noir) par la conversion des coefficients LPC déterminés par réccurence de Durbin, en coefficients cepstraux, ces derniers sont décalés de 20 dB.

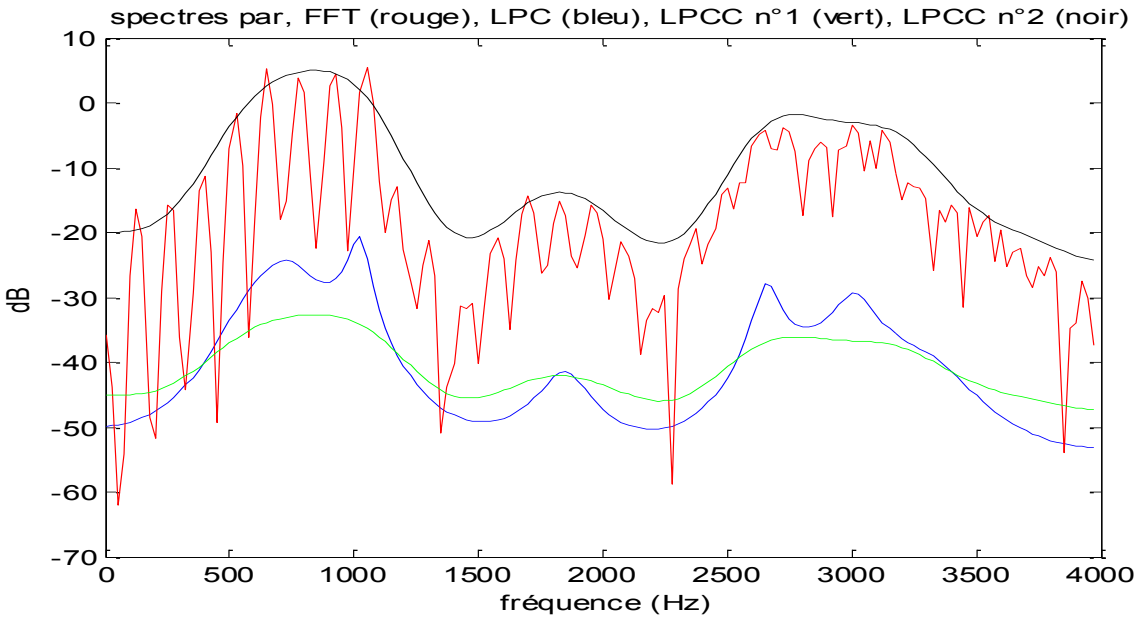


Figure I.15 : illustré de spectre de la parole et spectre lissé par LPC (bleu), LPCC n°1 voicebox (vert), et LPCC n°2 (noir)

I.7.3 Analyse par banc de filtres MEL

Les coefficients cepstraux les plus répandus sont les MFCC (Mel Frequency Cepstral Coefficients). Ils présentent l'avantage d'être faiblement corrélés entre eux, et qu'on peut donc approximer leur matrice de covariance par une matrice diagonale.

Pour simuler le fonctionnement du système auditif humain, les fréquences centrales du banc de filtres sont réparties uniformément sur une échelle perceptive. Plus la fréquence centrale d'un filtre est élevée, plus sa bande passante est large. Cela permet d'augmenter la résolution. Dans les basses fréquences, c'est la zone qui contient le plus d'information utile dans le signal de parole. Les échelles perceptives les plus utilisées sont l'échelle Mel et l'échelle Bark, fonction de la fréquence acoustique.

Echelle Mel :

$$F_{mel} = \frac{1000}{\log_{10} 2} \log_{10}\left(1 + \frac{f}{1000}\right) \qquad \text{(I.26)}$$

Echelle bark :

$$Bark(f) = 6\ Arc\ sinh\left(\frac{f}{1000}\right) \quad \text{(I.27)}$$

La procédure de calcul des coefficients MFCC est illustrée sur la figure I.16

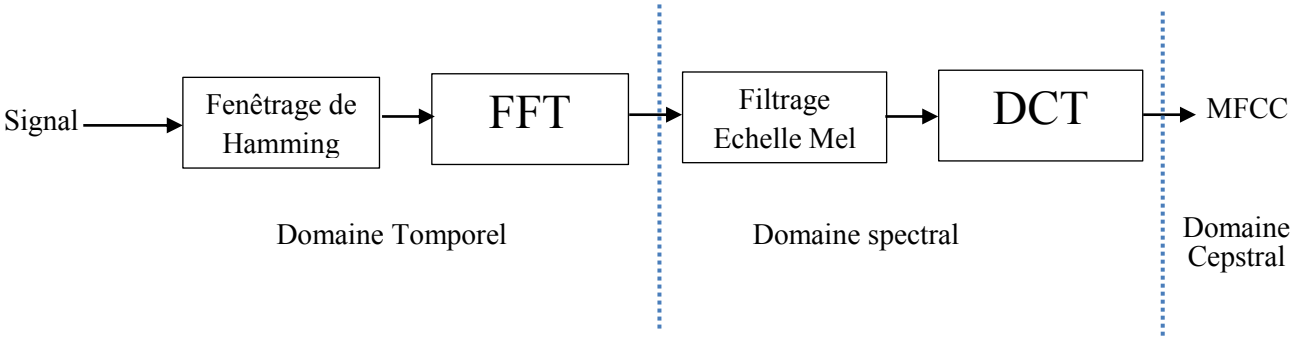


Figure I.16 : Calcul des coefficients MFCC

Soit un signal discret *s(n)* avec $0 \leq n \leq N - 1$, N est le nombre d'échantillons d'une fenêtre d'analyse, F_s est la fréquence d'échantillonnage, la transformée de Fourier discrète court terme *S(k)* est obtenue avec la formule :

$$S(k) = \sum_{n=0}^{N-1} s(n)\ exp\left(\frac{-j2\pi nk}{N}\right), 0 \leq k \leq N-1 \quad \text{(I.28)}$$

Le spectre du signal est filtré par un banc de filtres triangulaires, dont les bandes passantes sont de même largeur dans le domaine des fréquences Mel. Les points de frontières B_m des filtres en échelle de fréquence Mel sont calculés à partir de la formule :

$$B_m = B_b + m\frac{B_h - B_b}{M+1}\ ,0 \leq m \leq M+1 \quad \text{(I.29)}$$

Avec,

M : le nombre de filtres.

B_h : la fréquence la plus haute du signal.

B_b : la fréquence la plus basse du signal.

Dans le domaine fréquentiel, et d'après (I.29), les points f_m discrets correspondants sont calculés d'après :

$$f_m = B^{-1}\left(B_b + m\frac{B_h - B_b}{M+1}\right) \quad \text{(I.30)}$$

Où $B^{-1}(x)$ désigne la fréquence correspondante à la fréquence x sur l'échelle Mel,

$$B^{-1}(x) = 700\left(10^{\frac{x}{2595}} - 1\right) \quad \text{(I.31)}$$

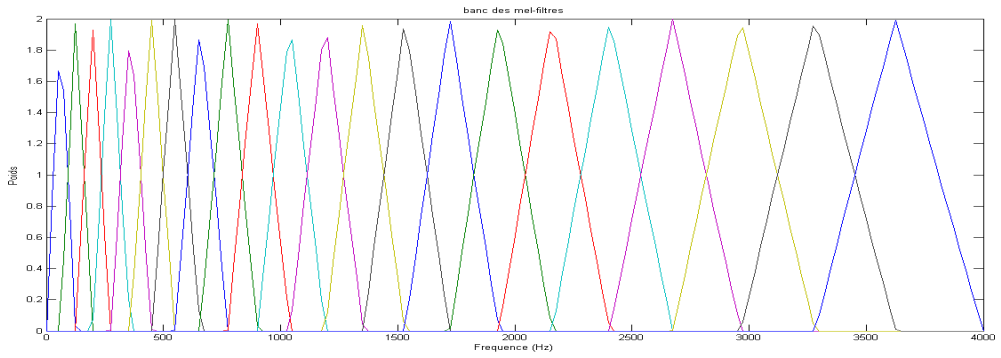


Figure I.17 : Banc de filtres Mel

Les coefficients cepstraux de fréquence en échelle Mel *(MFCC)* peuvent être obtenus par une transformée de Fourier inverse à partir des énergies d'un banc de filtres. En pratique, les d premiers coefficients cepstraux sont calculés directement à partir du logarithme des énergies E_i issues d'un banc de M filtres par la transformée en cosinus discrète définie comme :

$$c_k = \sum_{i=1}^{M} \log E_i \ cos\left[\frac{\pi k}{M}\left(i - \frac{1}{2}\right)\right] \quad ,1 \leq k \leq d \quad \text{(I.33)}$$

Et qui permet d'obtenir des coefficients peu corrélés.

Après avoir effectué une transformée de Fourier sur le signal de la parole, le spectre obtenu est traité par des filtres triangulaires en peigne. Nous avons travaillé avec deux banc de filtres triangulaires, le banc de filtres mel réalisé par la fonction **`melbankm`** et notre propre banc, appelé MEL-MFCC Mel figure I.18.

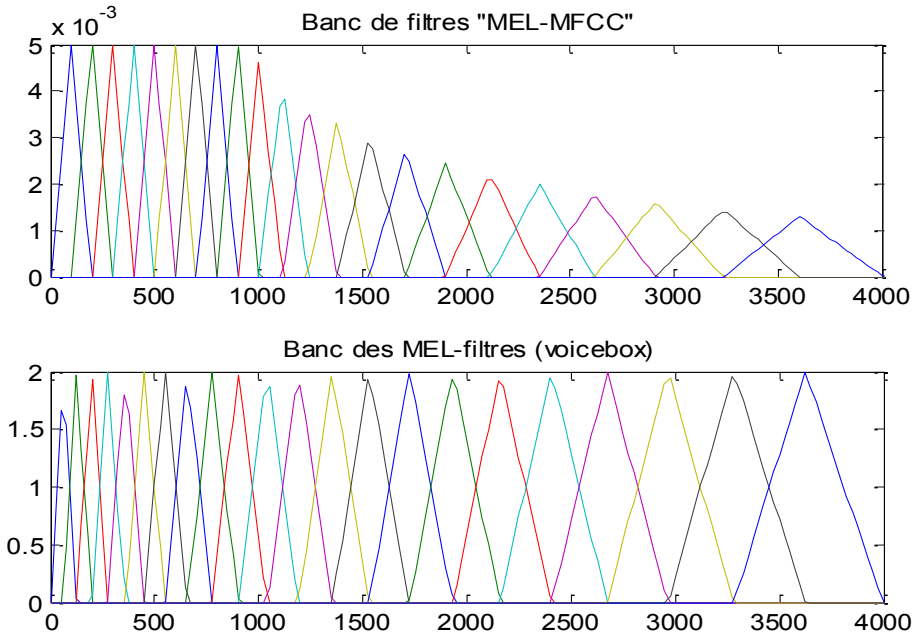


Figure I.18 : illustré du Banc de filtres (MEL-MFCC) et Banc de filtres (voicebox)

La Transformée en Cosinus Discrète est réalisée par la fonction **`rdct`** de MATLAB qui donne les coefficients MFCC, dans le cas où le filtrage est effectuée par la fonction **`melbankm()`** de voicebox. Par contre, les coefficients MFCC sont directement générés en employant la fonction **`melcepst()`** de voicebox, figure I.19.

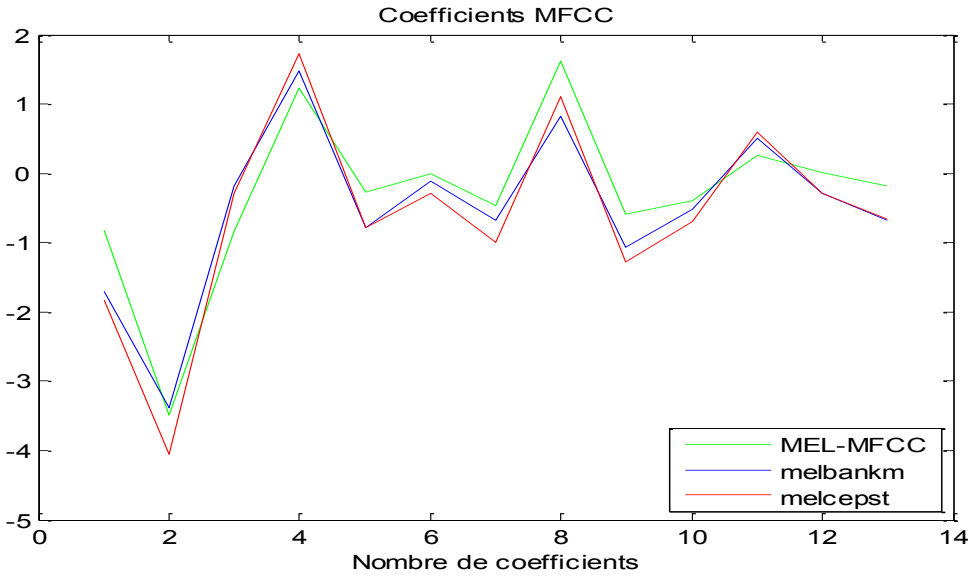


Figure I.19 : les coefficients MFCC

Cependant, nous avons constaté que le banc de filtres mel réalisé par la fonction **`melbankm`** n'assure pas une constance des pics des filtres triangulaires situés dans la zone linéaire (inférieur à 1KHz).

Nous avons donc réalisé notre propre banc, appelé MEL-MFCC, dont la source se trouve en annexe, tandis que sa présentation succinte est décrite dans le dernier chapitre. Les coefficients MFCC dérivés de notre banc sont reportés en figure I.18.

La figure I.20 illustre l'enveloppe spectrale estimée par les coefficients MFCC du banc de filtres (MEL-MFCC), qui suit bien la forme de spectre.

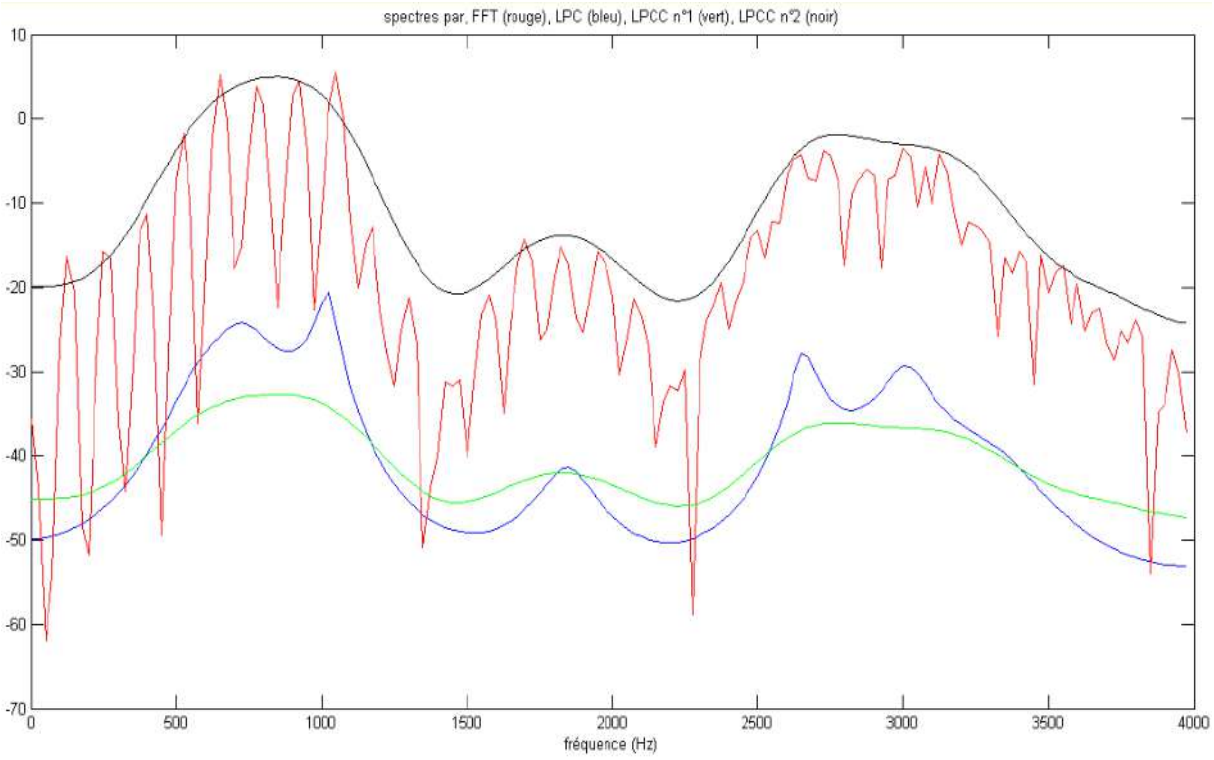


Figure I.20 : l'enveloppe spectrale lissée, FFT (rouge), LPC (bleu), liftrage(vert), notre_MFCC (noir)

La figure I.21 illustre l'enveloppe spectrale MFCC estimé par les coefficients de banc de filtres de « Voicebox ».

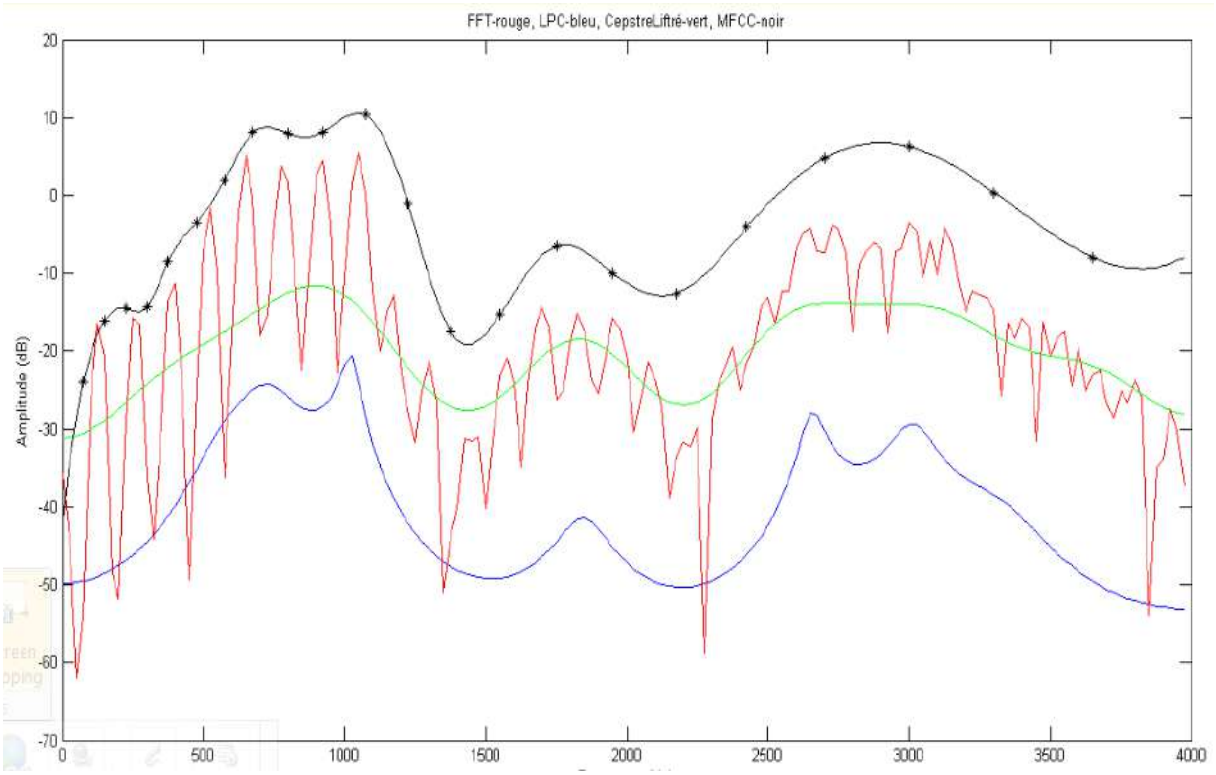


Figure I.21 : l'enveloppe spectrale lissée par MFCC issu du banc de filtres Voicebox (noir)

I.8 Le Spectrogramme

Le spectrogramme est un outil de visualisation utilisant la technique de la transformée de Fourier glissante ou à court terme. Il a commencé à être largement utilisé en 1947, à l'apparition du sonagraphe, et est devenu l'outil incontournable des études en phonétique pendant de nombreuses années.

L'apparition de l'informatique puis d'écrans graphiques de bonne qualité a permis d'abandonner tout matériel comme le sonagraphe, mais la technique du spectrogramme est encore aujourd'hui largement utilisée dans de nombreux domaines, du fait de sa simplicité de mise en œuvre et des résultats intéressants qu'elle procure.

On parle de spectrogramme *à larges bande***s** ou *à bandes étroites* selon la durée de la fenêtre de pondération. Les spectrogrammes à bandes larges sont obtenus avec des fenêtres de pondération de faible durée (typiquement 10 ms); ils mettent en évidence l'enveloppe spectrale du signal, et permettent par conséquent de visualiser l'évolution temporelle des

formants. Les périodes voisées y apparaissent sous la forme de bandes verticales plus sombres. Les spectrogrammes à bandes étroites sont moins utilisés. Ils mettent plutôt la structure fine du spectre en évidence : les harmoniques du signal dans les zones voisées y apparaissent sous la forme de bandes horizontales.

Le spectrogramme permet de mettre en évidence les différentes composantes fréquentielles du signal à tout instant, figure I.22.

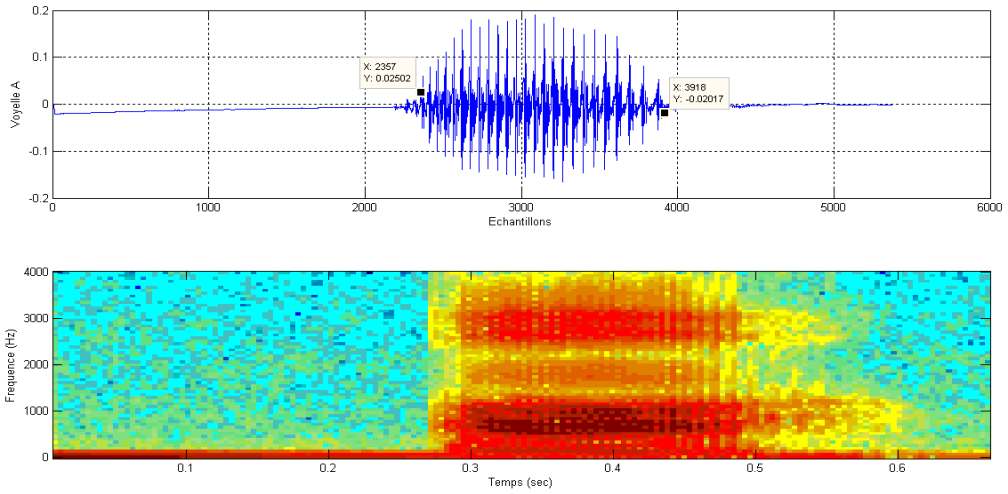


Figure I.22 : chronogramme (en haut), et Spectrogramme (en bas) de la voyelle « A »

Conclusion

Les simulations effectuées au sein de ce chapitre, couvrent les principales méthodes de paramétrisation de signal de la parole, telle que celles utilisées pour déterminer les coefficients cepstraux, formants et pitch. L'estimation du pitch est bien sûr liée à la localisation des tranches voisées. Cette tâche est difficile pour de nombreuses raisons, telle que la non-stationnarité du signal de la parole, On a déduit que la méthode cepstrale donne une meilleure estimation de la fréquence fondamentale de la parole .et les coefficients MFCC sont plus performants pour l'estimation des formants, puisqu'ils sont robustes au bruit qui est engendré par l'excitation glottale.

CHAPITRE II

CHOIX DU DSP ET PRESENTATION DE L'OUTIL DE DEVELOPPEMENT

II.1 Introduction

Depuis plusieurs années, le traitement numérique du signal est une technique en plein essor. Cette technique s'appuie sur plusieurs disciplines, citons les principales:

- L'électronique analogique et numérique (préparations, conditionnements des signaux, conversions numériques ↔ analogiques),
- Les microprocesseurs (classiques ou dédiés au traitement du signal),
- l'informatique (algorithmes, systèmes de développements, exploitations),
- Les mathématiques du signal (traitements du signal).

Dans ce chapitre nous exposons les principales caractéristiques des DSP et leurs applications, en mettant l'accent sur les DSP de la famille TMS320 de *Texas Instruments*.

II.2 Présentation des DSP

Lorsque les premiers microprocesseurs ont été conçus au début des années 1970, Il était hors de question de produire des circuits susceptibles de réaliser directement des opérations complexes comme la multiplication ou la division.

La solution utilisée alors, était de décomposer chaque instruction en une suite d'opérations simples, permise par le circuit interne du processeur. C'est ce que l'on appelle le microcodage. Avec ce système, chaque code machine correspond en fait à un micro programme, encodée dans le silicium. L'horloge du processeur détermine en général le nombre de cycles machines réalisé par second (par exemple, il y a des instructions comme la multiplication, qui peuvent demander 36 cycles machines pour certains processeurs).

Une telle approche permet d'améliorer le jeu d'instructions, donc les processeurs disposant d'un jeu d'instructions évolué au prix du nombre de cycles machines par instruction ont été regroupé sous le nom de CISC (*Complex Instruction Set Computer*).

Le problème principal qui se pose avec ces processeurs CISC est que le nombre de cycles nécessaire pour exécuter une instruction est variable dans une large proportion, et rend le calcul de la charge logicielle du processeur très difficile. Donc le but recherché était de définir un jeu d'instructions tel que, chacune d'entre elles s'exécute en un seul cycle d'horloge, ce qui conduit aux processeurs RISC (*Reduced Instruction Set Computer)* au début des années 1980.

Un DSP (*digital signal processor*) est un processeur dédié au traitement des signaux numériques. C'est un microprocesseur RISC dans lequel l'architecture et le jeu d'instructions sont optimisés afin d'obtenir un temps d'exécution le plus petit possible pour l'adapter aux algorithmes de traitement de signal [14].

Un DSP est un type particulier de microprocesseur. Il se caractérise par le fait qu'il intègre un ensemble de fonctions spéciales. Ces fonctions sont destinées à le rendre particulièrement performant dans le domaine du traitement numérique du signal.

Comme un microprocesseur classique, un DSP est mis en œuvre en lui associant des mémoires (RAM, ROM) et des périphériques figure II.1. Un DSP typique a plutôt vocation à servir dans des systèmes de traitements autonomes. Il se présente donc généralement sous la forme d'un microcontrôleur intégrant, selon les marques et les gammes des constructeurs, des mémoires, des timers, des ports séries synchrones rapides, des contrôleurs DMA, et divers ports d'E/S.

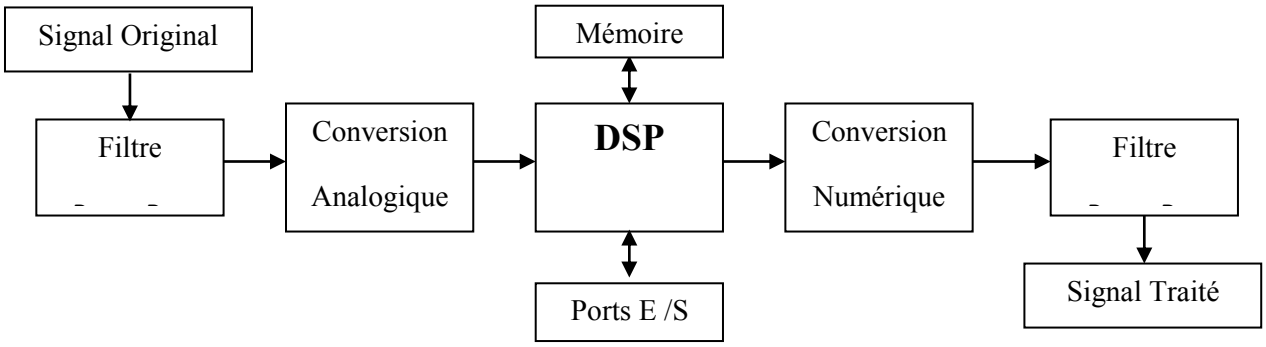


Figure II.1 : chaîne complète typique d'un système de traitement numérique du signal

II.2.1 Les avantages d'un DSP

Tous les systèmes à bases de DSP bénéficient des avantages suivants :

II.2.1.1 Souplesse de programmation

Un DSP est avant tout un processeur exécutant un programme de traitement du signal. Ceci signifie que le système bénéficie donc d'une grande souplesse de développement. De plus, les fonctions de traitements numériques peuvent évoluer en fonction des mises à jour des programmes, et cela pendant toute la durée de vie du produit incluant le système. Ainsi, modifier par exemple tel ou tel paramètre d'un filtre numérique ne nécessite pas un changement matériel.

II.2.1.2 Implémentation d'algorithmes adaptatifs

Une autre qualité issue de la souplesse des programmes. Il est possible d'adapter une fonction de traitement numérique en temps réel suivant certains critères d'évolutions du signal (exemple : les filtres adaptatifs).

Certaines fonctions de traitement du signal sont difficiles à implanter en analogique, voire irréalisables (exemple : un filtre à réponse en phase linéaire), elles se voient réalisés sur DSP.

II.2.1.3 Stabilité

En analogique, les composants sont toujours plus ou moins soumis à des variations de leurs caractéristiques en fonction de la température, de la tension d'alimentation, du vieillissement, etc. Une étude sérieuse doit tenir compte de ces phénomènes, ce qui complique et augmente le temps de développement. Ces inconvénients n'existent pas en numérique.

II.2.1.4 Répétabilité et reproductibilité

Les valeurs des composants analogiques sont définies avec une marge de précision plus ou moins grande. Dans ces conditions, aucun montage analogique n'est strictement reproductible à l'identique, il existe toujours des différences qu'il convient de maintenir dans des limites acceptables. Un programme réalisant un traitement numérique est par contre parfaitement reproductible.

Vu ces avantages, les domaines d'applications du traitement numérique du signal sont nombreux et variés (traitements du son, de l'image, synthèse et reconnaissance vocale, analyse, compression de données, télécommunications, automatisme, etc). Chacun de ces domaines nécessite un système de traitement numérique, dont le cœur est un ou parfois plusieurs DSP ayant une puissance de traitement adaptée, pour un coût économique approprié [15].

II.2.2 Possibilités des DSP

II.2.2.1 Numérisation du signal

Parallèlement aux microprocesseurs et aux microcontrôleurs, les processeurs de traitement numérique du signal, ou DSP *(Digital Signal Processor),* ont bénéficié d'énormes progrès en rapidité (grâce au faible temps de commutation) et en puissance de calculs (grâce au nombre de bits des bus internes).

D'une manière générale, tous les processeurs vérifient la loi de Moore, l'un des fondateurs de la société Intel qui annonçait, que la densité d'intégration des composants doublerait tous les

dix-huit mois. En 1995, on savait intégrer dix millions de transistors dans un composant unique.

Utilisés initialement pour gérer la carte son des micro-ordinateurs, les DSP ont vu leur utilisation s'accroître considérablement depuis 1985, tout d'abord grâce au développement des télécommunications (téléphonie numérique, puis téléphonie sans fil GSM ...), puis grâce à ses possibilités de traitement rapide de certaines commandes numériques faisant appel à des algorithmes complexes permettant ainsi le travail en « temps réel ».

II.2.2.2 Rôle du DSP

Le traitement numérique du signal, qu'il provienne du son ou d'une image vidéo, est rendu accessible par le DSP grâce à son unité de calcul spécifique *multiplicateur / additionneur / accumulateur de données.*

En effet, tout DSP est prévu pour effectuer le plus rapidement possible, en principe en un seul cycle d'horloge, l'opération multiplication/addition sur des grandeurs numériques :

$$MR=X\cdot Y+R \qquad \text{(II.1)}$$

Où X et Y sont soit des données, soit des constantes et R une donnée, une constante ou un résultat précédent. MR est alors le résultat de l'opération arithmétique.

Si le DSP fonctionne en virgule fixe avec des données sur 16 bits, le résultat MR est alors sur 32 bits (ou plus, selon l'architecture). Si l'utilisateur ne conserve que les 16 bits *de poids fort,* le calcul est alors effectué en *simple précision.* Si les 32 bits sont utilisés, on parle de *double précision :* le temps de calcul est alors plus long.

Si le DSP fonctionne en virgule flottante avec des données en 32 bits, le résultat MR est alors sur 40 bits (ou plus, selon l'architecture). L'utilisateur ne prend en compte que les données de 32 bits en ignorant les bits de poids faibles de la mantisse, en simple précision, mais travaille sur 64 bits en double précision.

Le premier DSP a été produit en 1982 par Texas Instruments. Depuis, cinq autres générations de DSP sont apparues. Les processeurs des générations 1, 2 et 5 sont en virgule fixe, les générations 3, 4, 6 en virgule flottante.

En 2014, les principaux fabricants de DSP sont :

− Texas Instruments
− Analog Devices
− Sony Semi-conductors
− Zilog
− Freescale

− NXP Semi-conductors
− Cirrus Logic
− Amptek Inc
_ Microchip Technology

Les trois premiers constructeurs sont les plus connus. Néanmoins, certains petits constructeurs développent des architectures originales. Microchip, fort du succès de ces microcontrôleurs PIC met sur le marché une famille de DSP 16 bits virgule fixe, les dsPIC.

II.2.2.3 Les diverses approches possibles

II.2.2.3.a Approche signal

Cette approche caractérise d'emblée le DSP et fait de lui un composant numérique pour les spécialistes du signal et les électroniciens. L'utilisateur retrouve les filtres, avec amplification, atténuation, mais aussi le traitement numérique du signal (convolution, corrélation, transformée de Fourier rapide (FFT), filtrage numérique, compression, codage et décodage des données ...), toutes ces applications faisant appel à des algorithmes.

Les signaux à traiter sont caractérisés par la fréquence maximale possible associée à un phénomène physique ou à un système donné. D'après le théorème de Shannon, la fréquence d'échantillonnage F_e, doit être telle que :

$$F_e \geq 2 f_{max} \qquad \text{(II.2)}$$

Si on ne respecte pas cette règle, il risque d'y avoir repliement spectral (aliasing). La fréquence d'échantillonnage Fe est en général imposée par le convertisseur analogique/numérique.

Le nombre de cycles de calcul à l'intérieur d'une période d'échantillonnage Te, donne la possibilité pour le DSP d'intervenir efficacement ou non en temps réel entre deux prises d'échantillons. Si ce n'est pas le cas, par exemple pour un signal vidéo, le DSP intervient selon un processus plus lent, tout en gérant des interruptions sur une période d'échantillonnage. La gestion des ports entrée/sortie devient alors très complexe.

Signal	***Fréquence d'échantillonnage***	***Période d'échantillonnage***	***Nombre de temps de cycle T_{cycle} entre deux échantillons avec T_{cycle} =50ns***
Signaux vibratoires	100 kHz	10 µs	200
Parole	8 kHz	125 µs	2 500
Audio	48 kHz	20.8 µs	417
vidéo	10 MHz	100 ns	2

Tableau II.1 : Nombre de cycles de calcul disponible en fonction de la fréquence d'échantillonnage

A l'examen du tableau.II.1, on comprend que les DSP sont couramment utilisés pour traiter en temps réel les signaux de la bande audio (0 à 20 kHz de largeur de bande), mais que le traitement des signaux vidéo en temps réel reste la plupart du temps du domaine des circuits intégrés dédiés ou d'architecture multi DSP [15]. Mais récemment, des DSP multicores sont apparus, et sont dédiés au traitement vidéo en temps réel.

II.2.2.3.b Approche technologique

La Figure II.2 montre la vitesse d'un composant en fonction de sa « performance », c'est-à dire de l'adaptation à des besoins spécifiques de l'électronique. On voit la place privilégiée du DSP par opposition à celle du microprocesseur, d'usage plus général.

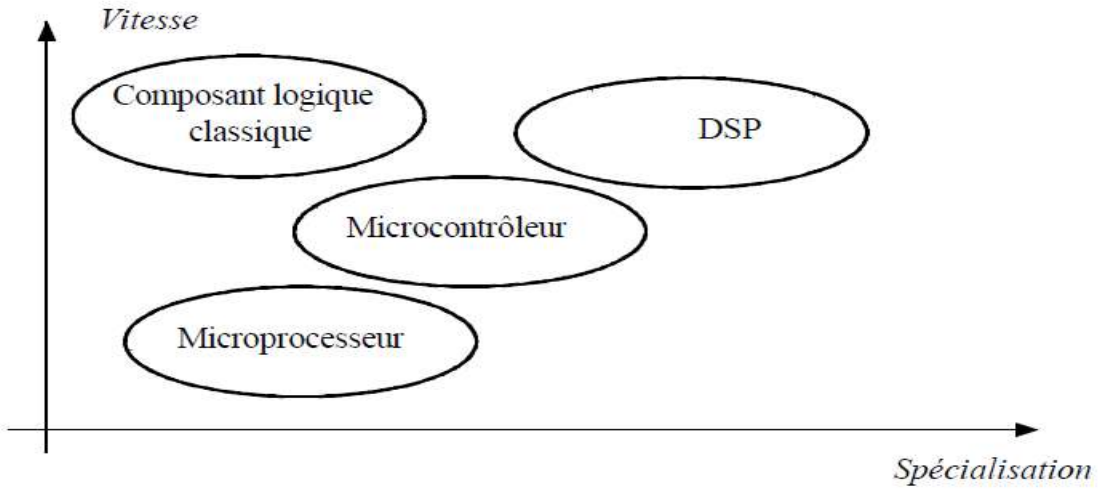


Figure II.2 : Place du DSP vis-à-vis des autres processeurs

II.2.3 Noyau du DSP

Soit un processeur assurant en un seul cycle d'horloge l'opération MAC élémentaire, expression (II.1).

Il s'agit par exemple d'effectuer des opérations arithmétiques du type produit de convolution

$$\sum x(n).y(n-k) \qquad \text{(II.3)}$$

Il est alors nécessaire que le processeur possède une architecture de Harvard avec deux mémoires vives au minimum, c'est-à-dire une mémoire de programme et une mémoire de données. Les données x(n) proviendront par exemple de la mémoire de données, et y (n – k) de la mémoire programme.

Ces mémoires communiquent avec quatre bus internes au moins, c'est-à-dire un bus d'adresse pour chacune des mémoires, un bus de données, et un bus de programme. Un système générateur d'adresses, appelé pointeur, s'impose pour activer les mémoires en lecture ou en écriture.

La rapidité de calcul s'obtient non seulement grâce à un cycle d'horloge très court, et donc une fréquence élevée, de l'ordre de 200MHz typique, mais aussi grâce à la gestion d'un pipeline.

II.3 Classification des DSP

Il est impossible d'effectuer une classification « définitive » des DSP, car chaque constructeur met sur le marché tous les ans un nouveau composant qui surclasse les anciens ou les concurrents par la puissance de calcul, la rapidité (gestion du pipeline et fréquence d'Horloge), le nombre de registres, de Timers, de ports série…

II.3.1 Virgule fixe ou flottante

Un point essentiel des DSP est la représentation des nombres (les données) qu'ils peuvent manipuler. Il est possible de distinguer deux familles :

− Les DSP à virgule fixe: les données sont représentés comme étant des nombres fractionnaires à virgule fixe, (exemple -1.0 à +1.0), ou comme des entiers classiques. La représentation de ces nombres fractionnaires s'appuie sur la méthode du « complément à deux».

− Les DSP à virgule flottante: les données sont représentés en utilisant une mantisse et un exposant. La représentation de ces nombres s'effectue selon la formule suivante :

n = *mantisse* x $\mathbf{2}^{\text{exposant}}$. Généralement, la mantisse est un nombre fractionnaire (-1.0 à +1.0), et l'exposant est un entier indiquant la place de la virgule en base 2.

II.3.1.1 Les DSP à virgule flottante

Les DSP à virgule flottante sont plus souples et plus faciles à programmer que les DSP à virgule fixe. Un DSP comme le TMS320C30 manipule des nombres formés avec une mantisse de 24 bits et un exposant de 8 bits (taille de la donnée en mémoire : 32 bits). Les valeurs intermédiaires des calculs sont mémorisées dans des registres avec un format de 32 bits de mantisse et un exposant de 8 bits (taille du registre : 32 + 8 bits supplémentaires). La dynamique disponible est très grande, elle va de -1 x 2^{128} à (1-2^{-23}) x 2^{127}, toutefois la résolution reste limitée à 24 bits au mieux. Outre les nombres fractionnaires, ce DSP sait également manipuler les entiers avec une précision de 32 bits.

La très grande dynamique proposée par les DSP à virgule flottante permet virtuellement de ne pas se soucier des limites des résultats calculés lors de la conception d'un programme. Cet avantage a cependant un prix, à savoir qu'un système basé sur un DSP à virgule flottante à un coût de fabrication supérieur par rapport à un système basé sur DSP à virgule fixe. La puce d'un DSP à virgule flottante nécessite à la fois une surface de silicium plus importante (cœur plus complexe), et un nombre de broches supérieur, car la mémoire externe est elle aussi au format 32 bits. Le système revient donc plus cher (exemple : 2 x 32 broches ne serait ce que pour les bus de données externes avec une architecture Harvard de base).

Un DSP à virgule flottante est plutôt adapté (sans être impératif) à des applications dans lesquelles :

− Les coefficients varient dans le temps (exemple: les filtres adaptatifs),

− Le signal et les coefficients ont besoin d'une grande dynamique,

− La structure mémoire est importante (exemple : traitement d'image),

− la précision est recherchée sur toute une gamme dynamique importante (exemple : traitements audiophoniques de qualité professionnelle).

A part leurs facilités de programmation, ils peuvent également se justifier dans des projets ou le temps et la facilité de développement sont des facteurs importants. On les trouve également dans des produits de faible volume de production, pour lesquels le prix du DSP n'est pas significatif.

II.3.1.2 Les DSP à virgules fixes

Un DSP à virgule fixe est un peu plus compliqué à programmer qu'un DSP à virgule flottante.

Dans un DSP à virgule fixe typique comme le TMS320C25, les nombre sont codés sur 16 bits (généralement, des entiers classiques ou des fractionnaires).

Toutefois, sur ce DSP, les calculs sont effectués avec des accumulateurs de 32 bits. Lorsque les résultats doivent être stockés en mémoire, les 16 bits les moins significatifs sont perdus.

Ceci permet de limiter les erreurs d'arrondis cumulatives. Il est toujours possible de stocker séparément en mémoire les 16 bits faibles puis les 16 bits fort s'il n'y a plus de registres libres lors d'une étape de calcul.

Cette particularité n'est pas toujours disponible sur tous les DSP. Dans ce cas, les calculs requérant absolument une double précision sont réalisés en chaînant deux à deux des instructions spéciales manipulant des données 16 bits en simple précision, au détriment du temps d'exécution.

La précision des calculs est un point critique des DSP à virgule fixe, car le concepteur de programmes doit rester vigilant à chaque étape d'un calcul. Il doit rechercher la plus grande dynamique possible (c.à.d. exploiter au mieux la gamme des nombres disponibles), pour conserver une bonne précision des calculs, tout en évitant autant que faire ce peu les débordements du ou des accumulateurs. Les bits supplémentaires des accumulateurs (les bits de garde) prévus à cet effet permettent de réduire cette contrainte.

Les programmeurs contournent les limites des DSP à virgule fixe en déterminant à l'avance, et avec soins, la précision et la dynamique nécessaire (par méthode analytique ou avec des outils de simulation) pour réaliser leurs projets.

Il est également possible d'effectuer des opérations en virgule flottante dans un DSP à virgule fixe par le biais de routines logicielles adéquates. Cette approche est néanmoins pénalisante en temps d'exécution, même sur un DSP à virgule fixe très rapide.

En termes de rapidité, les DSP à virgule fixe se placent d'ordinaire devant leurs homologues à virgule flottante, ce qui constitue un critère de choix important.

Les DSP à virgule fixe sont les plus utilisés, car ils sont moins chers que les DSP à virgule flottante. On les trouve dans tous les produits de grande diffusion ou le coût est un facteur important. Il peut cependant exister des exceptions, certains DSP à virgule fixe se présentant comme des microcontrôleurs perfectionnés plus chers qu'un DSP à virgule flottante de base.

II.3.2 Performance des DSP

Plus que pour un microprocesseur classique, les performances d'un DSP conditionnent son domaine d'application.

La plupart des DSP sont particulièrement destinés à des applications « temps réel » et spécialisées, c'est à dire des applications où le temps de traitement est bien sûr primordial, mais où la diversité des événements à traiter n'est pas notablement importante. De ce point de vue, l'approche DSP s'apparente plus à une étude « électronique » visant à réaliser une ou des fonctions de traitements de signal, que d'une approche informatique temps réel et/ou multitâche traditionnelle.

Dans tous les cas, les performances du DSP sont critiques. Le concepteur d'un système à base de DSP doit évaluer d'une part la « puissance » nécessaire pour réaliser les traitements numériques voulus, et d'autre part les performances des DSP disponibles pour réaliser son application.

II.3.3 Puissance de calcul d'un DSP

C'est un autre critère de classification des DSP. Cette puissance de calcul dépend de la rapidité de l'exécution des instructions, et donc de l'horloge. Dans un DSP, le MAC est pris comme une instruction de référence, pour laquelle on mesure le temps d'exécution, figure II.3

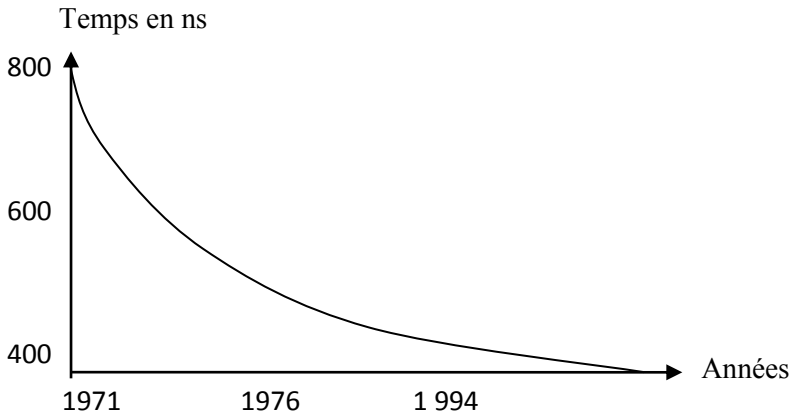


Figure II.3 : Evolution du temps d'exécution d'une opération *MAC*, selon Texas Instruments

Le Tableau II.2 résume les principales définitions en usage, concernant le performance du DSP en terme de vitesse d'exécution.

Acronyme Anglais	***Définition***
MFLOPS **M**illion **FL**oating-point **O**perations **P**er **S**econd.	Mesure le nombre d'opérations à virgule flottante (multiplications, additions, soustractions, etc.) que le DSP à virgule flottante peut réaliser en une seconde.
MOPS **M**illion **O**perations **P**er **S**econd	Mesure le nombre total d'opérations que le DSP peut effectuer en une seconde. Par opérations, il faut comprendre non seulement le traitement des données, mais également les accès DMA, les transferts de données, les opérations d'E/S, etc. Cette définition mesure donc les performances globales d'un DSP plutôt que ses seules capacités de calcul.
MIPS **M**illion **I**nstructions **P**er **S**econd.	Mesure le nombre de codes machines (instructions) que le DSP peut effectuer en une seconde. Bien que cette mesure s'applique à tous les types de DSP, le MFLOPS est préféré dans le cas d'un DSP à virgule flottante.
MBPS **M**ega-**B**ytes **P**er **S**econd.	Mesure la largeur de bande d'un bus particulier ou d'un dispositif d'E/S, c'est à dire son taux de transfert.

Tableau II.2 **:** Définitions des unités les plus courantes de mesures de vitesse des DSP

II.3.4 Mesure du temps d'exécution (Benchmark)

La vitesse de calcul pure d'un DSP n'est pas une indication universelle, les méthodes et les résultats diffèrent d'un DSP à l'autre. De plus elle ne rend pas compte d'un certain nombre de perfectionnement dont peuvent bénéficier tel ou tel DSP.

Certains DSP proposent en effet des modes d'adressages plus performants que d'autres. Ces modes sont spécialement adaptés à des algorithmes standards du traitement du signal (exemple : le mode d'adressage dit « bits reversing » pour accélérer les calculs des FFT), et ne devraient pas être ignorées.

Enfin, le temps d'accès à la mémoire est un autre paramètre incontournable. Certains DSP intègrent des blocs de mémoire vive rapide. Cette mémoire est placée dans l'espace d'adressage du DSP au même titre que de la mémoire vive externe, ce qui permet d'y ranger données et programmes sans avoir à effectuer des transferts permanents de ou vers l'extérieur.

Les éventuels cycles d'attente pouvant être nécessaires pour adresser une mémoire externe lente sont ainsi évités.

Pour toutes ces raisons, la mesure des performances par benchmark, complète avantageusement la mesure de vitesse pure. Elle consiste à mesurer le temps que met le DSP pour exécuter des programmes « standards » de traitement du signal.

II.4 Architecture fonctionnelle

Un processeur est un composant intégré (en technologie C-MOS) ou une partie de composant susceptible d'exécuter des instructions selon un programme d'instructions préétabli sur des données numériques.

L'architecture est ce qui détermine d'emblée et de manière définitive les principales caractéristiques du processeur, en particulier la rapidité d'exécution des instructions. Elles sont liées au nombre de bus internes, qui sont des liaisons parallèles à N bits, à la valeur même de N, aux opérations possibles sur les données transitant sur les bus et enfin à la puissance de calcul de l'unité centrale.

Les bus sont reliés à des registres temporaires pour stocker provisoirement les codes d'instructions, des adresses ou des données. D'autres types de registres existent :

- Les registres de contrôle ou d'état, qui ne sont pas stockés en mémoire car ils sont

adressés directement par une instruction, par mesure de sécurité. Ces registres sont essentiels car ils déterminent le déroulement précis d'une instruction. Certains d'entre eux sont en lecture seule, d'autres en écriture seule.

- les registres définissant les conditions de fonctionnement d'un port (série, DMA, hôte ...), des interfaces avec des mémoires de sortie ou du Timer, qui ont une adresse en mémoire de données.

Les instructions s'exécutant au cœur du processeur appelé unité centrale, comportant dans le cas le plus classique, une seule unité arithmétique de traitement des données. À l'entrée des unités de calcul constituant l'unité centrale, les données sont chargées dans des registres temporaires qui, lors de la réalisation de l'instruction de calcul, vont être reliés à l'unité de calcul ; le résultat de l'opération sera disponible dans un registre (temporaire) de sortie.

L'architecture est conçue pour que les instructions puissent :

- charger les données dans un registre temporaire d'entrée en lisant le contenu d'une mémoire vive à une adresse donnée, les données transitant par un bus de données et les adresses pour la lecture par un autre bus ;
- effectuer une opération logique ou arithmétique sur une donnée ou entre deux données dans l'unité centrale;
- effectuer des tests sur le résultat obtenu (dépassement, signe...) ;
- modifier certains registres ;
- aller écrire un résultat à une adresse de la mémoire prévue.

Toutes ces instructions doivent en principe être exécutées en un seul cycle d'horloge.

L'architecture contient donc un noyau DSP qui comporte une unité centrale, laquelle est composée d'une ou plusieurs unités de traitement, chacune comportant des unités de calcul, comme l'ALU (Arithmetic and Logic Unit) le MAC (Multiplier and ACcumulator), le décaleur à barillet (Barrel Shifter) et enfin des multiplexeurs d'aiguillage des données. En plus

de 1'unité centrale, le noyau DSP comporte un séquenceur, pour envoyer les adresses des instructions enregistrées dans la mémoire programme, et un ou plusieurs générateurs d'adresses (parfois appelés pointeurs) agissant sur les bus d'adresses.

Dans l'architecture dite de Harvard (car mise au point dans cette université américaine en 1930), on sépare systématiquement la mémoire de programme de la mémoire des données : l'adressage de ces mémoires est indépendant. La figure II.4 indique une architecture simple de Harvard, constituée d'un bus de données, d'un bus de programme et de deux bus d'adresse.

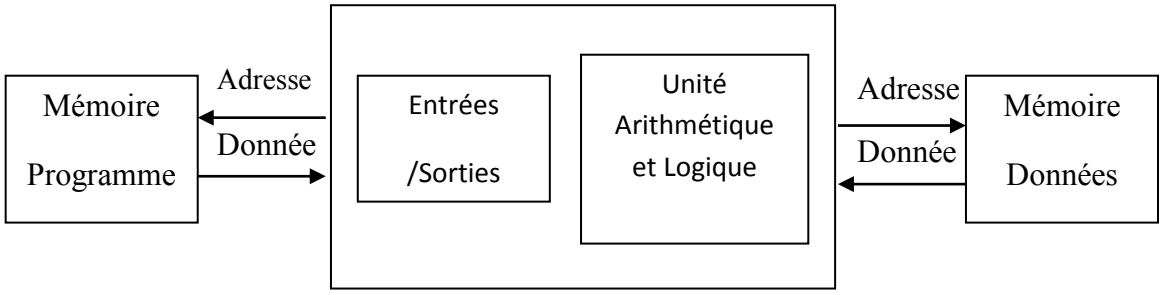


Figure II.4 : Architecture de Harvard

On voit que les échanges s'effectuent de manière double entre l'unité centrale et les deux mémoires, ce qui permet une grande souplesse pour l'enregistrement et l'utilisation des données. D'ailleurs, la mémoire de programme est également utilisée en partie comme mémoire de données pour obtenir encore plus de possibilités de traitement avec des algorithmes complexes. L'architecture Harvard est utilisée dans des DSP.

Pour réduire le coût de la structure Harvard, certains DSP utilisent l'architecture dite « Structure de Harvard modifiée ». À l'extérieur, le DSP ne propose qu'un bus de données et un bus d'adresse, comme la structure Von Neuman. Toutefois, à l'intérieur, la puce DSP dispose de deux bus distincts de données et de deux bus distincts d'adresses. Le transfert des données entre les bus externes et internes est effectué par multiplexage temporel.

II.5 Les DSP de la famille TMS320

La famille TMS320 se compose de processeurs à virgule fixe, à virgule flottante et de multiprocesseurs [14], [15], [16]. L'architecture des DSP TMS320 est conçue spécifiquement pour le traitement des signaux en temps réel. Les caractéristiques suivantes font de cette famille, le choix idéal pour une large gamme de traitement des applications :

- ensemble d'instructions très flexible ;
- flexibilité opérationnelle inhérente ;
- exécution à grande vitesse
- architecture parallèle innovatrice
- rentabilité

Les DSP TMS320 offrent des approches plus adaptables aux problèmes de traitement de signal classique tel que le filtrage, que des dispositifs de microprocesseur standard. Ils sont très utiles pour les applications complexes qui exigent souvent l'exécution simultanée de plusieurs opérations.

Les familles les plus récentes des DSP de *Texas instrument,* Tableau II.3, sont :

- TMS320C54x, DSP virgule fixe ;
- TMS320C55x, DSP virgule fixe
- TMS320C20x, DSP virgule fixe ;
- TMS320C24x, DSP virgule fixe ;
- TMS32OC62x, DSP virgule fixe à architecture VLIW (Very long instruction word);
- TMS32OC64x, DSP virgule fixe
- TMS32OC67x, DSP virgule flottante à architecture VLIW ;

Ces familles sont regroupées en 3 classes appelées plates-formes.

- TMS320C6000, formée des familles C62x et C67x et C64x ;
- TMS320C5000, formée de la famille C54x et C55x ;
- TMS320C2000, formée des familles C20x et C24x ;

Les Principales caractéristiques des 3 plateformes de TMS320 sont données dans le tableau II.3.

Applications		**Type de DSP**	**caractéristiques**
DSP hautes performances			
C62x	Applications exigeantes en vitesse : stations de base des réseaux de	16 bits virgule fixe.	1200-2400 MIPS

	communications mobiles, équipement de radiodiffusion, réseaux informatiques	Architecture VLIW.	
C67x	Applications exigeantes en précision, dynamique et vitesse : Antennes adaptatives des stations de base, imagerie médicale, reconnaissance de parole, graphisme ...	32 bits virgule flottante. architecture VLIW.	600MFLOPS-1GFLOPS
C5000 DSP optimisés en consommation			
C54x Et C55x	Applications de télécommunications exigeantes en coût, en consommation : voix sur IP.	16 bits virgule fixe.	0,54 mW/MIPS 30-200 MIPS
C2000 DSP optimisés pour les applications de contrôle			
C20x	Applications téléphonie, électronique grand public, appareils photos numériques ou contrôleurs de disques durs	16 bits virgule fixe	20-40MIPS
C24x	Applications de contrôle moteur, automatisation, robotique, contrôle d'appareils électroménagers... Bon compromis prix/performance	16 bits virgule fixe	20MIPS

Tableau II.3 : Principales caractéristiques des 3 familles de TMS320

II.6 Choix du DSP pour notre application

Notre choix s'est orienté au début vers la famille C54x, en particulier, le DSP type C5410. Cette famille est à format 16 bits, virgule fixe à grande économie d'énergie et suffisamment rapide pour des applications audio et traitement de la parole. Le DSP C5410 figure parmi les DSP les plus performants de la famille tout en étant simple à programmer en assembleur. Nous allons donné un bref aperçu sur cette famille, et plus particulièrement sur le DSP C5410.

II.6.1 La famille TMS320C54x

Les DSP C54x ont un degré élevé de flexibilité et de vitesse opérationnelle. Ils Combinent une architecture avancée de *Harvard* modifiée (avec un bus de mémoire programme, trois bus de mémoire de données, et quatre bus d'adresses), une CPU, une mémoire, des périphériques, et un jeu d'instructions fortement spécialisé [17].

II.6.1.1 Les avantages des DSP C54x

- Architecture de *Harvard* établie autour d'un bus de programme, trois bus de données, et quatre bus d'adresses pour accroître les performances et la polyvalence ;
- Conception avancée d'unité centrale (CPU), un degré élevé de parallélisme ;
- Un jeu d'instructions fortement spécialisé pour des algorithmes plus rapides et pour optimiser l'opération avec le langage de haut niveau ;
- Conception d'architecture modulaire pour le développement rapide des dispositifs ;
- Technologie avancée pour une faible consommation, et utilisation de nouvelles techniques de conception statiques [17].

II.6.1.2 Les composants de la famille TMS320C54x

La famille C54x est utilisée en particulier dans le domaine des télécommunications avec les mobiles. Elle possède des bonnes performances en consommation, en vitesse, et en prix. L'architecture se caractérise par [17] :

- **CPU :**

  - Quatre bus internes et deux générateurs d'adresse ;
  - Une ALU 40 bits pouvant travailler en double précision ou en mode dual ;
  - Deux accumulateurs 40 bits ;
  - Une unité de multiplication et d'addition, avec un multiplieur 17 × 17 bits ;
  - Unité de comparaison, sélection, sauvegarde (CSSU), appelée aussi accélérateur de *Viterbi* ;
  - Un calculateur d'exposant en un seul cycle pour faciliter les calculs en format flottant nécessaire dans certaines applications ;
  - Un registre à décalage 40 bits ;
  - Huit registres auxiliaires et une pile logicielle ;
  - Un jeu d'instructions contenant des instructions mono-cycle spécialisées dans le traitement de signal, comme le filtrage adaptatif ou les filtres symétriques.

- **Mémoire :**
  - 192 Kmots de 16 bits d'espace mémoire (64 Kmots pour la mémoire programme, 64 Kmots pour la mémoire données, et 64 Kmots pour les E/S), avec prolongement de la mémoire programme dans le C5410.
- **jeu d'instructions :**
  - opérations de répétition d'une Simple instruction et de répétition de bloc;
  - Instructions de déplacement de bloc mémoire pour une meilleure gestion de programme et les données ;
  - Instructions avec une longueur d'opérande de 32 bits ;
  - instructions avec 2 ou les 3-opérandes exécutés simultanément ;
  - Instructions arithmétiques avec enregistrement et chargement parallèle ;
  - Stockage des instructions conditionnelles ;
  - Rapidité du retour de l'interruption ;

Après avoir préparé le fichier de lien gérant la mémoire disponible du DSP cible C5410, puis avoir trouvé au sein du la bibliothèque téléchargée DSPlib, les fonctions en assembleur nécessaires à notre traitement : CFFT (fft complexe), CIFFT (ifft comlexe), LOGN, LOG_10, Power, SINE (cosinus à déduire), SQRTV (racine carrée sur 16 bits). Les fonctions importantes était là, sauf les opérations élémentaires telle la division entre réel, qui était au fait bornée aux types Q31 et Q15. Craignant le temps nous échapper en confectionnant des routines en assembleur pour l'opération division entre réels, ou d'autres fonctions parmi celles citées plus haut, et ce dans le cas où l'une d'elles montrait une insuffisance en précision de calcul, nous étions amené à utiliser en parallèle le langage C comme complément.

Malheureusement, le compilateur C du simulateur du DSP C5410 s'est avéré non opérationnel et n'accomplit que la vérification de la syntaxe du programme C.

La recherche de compilateurs C complets pour la famille C54x s'est vouée à l'échec.

En fin de compte nous nous sommes résolu à adopter le compilateur C de la famille C6x qui était disponible, et par là, opter sur la C6711 comme un DSP cible.

II.6.2 la famille TMS320 C67x [18]

Développée avec une architecture à haute performance et haute niveau de programmation, cette famille utilise la technique VLIW (très long mot d'instruction) afin de maximiser le parallélisme et minimiser les divers conflits. L'architecture est ajustée afin de mieux satisfaire le compilateur C.

Les outils de programmation facilitent le développement des programmes en C et réduisent le temps de développement en permettant la réutilisation du software sur d'autres plate-formes à DSP. L'optimiseur C est un outil d'optimisation du code source C vers l'assembleur, qui est hautement mis en parallèle dans le Pipline.

Outre le langage C, la programmation peut être effectuée en assembleur linéaire ou macro-assembleur.

II.6.2.1 Caractéristique de la famille C67x

- Unité centrale à huit unités fonctionnelles incluant deux multiplieurs et six UAL.
- Pipline à huit instructions par cycle.
- Technique VLIW avancée, où la taille du code est équivalente à huit instructions exécutée en parallèle.
- Tout le jeu d'instruction s'exécute conditionnellement, réduisant les temps de branchement.
- Exécution du code sur des unités fonctionnelles indépendantes, assurant une compilation C la plus efficace de l'optimiseur d'assemblage.
- Traitement de divers formats de donnés : 8/ 16/ 32 bit.
- Arithmétique d'extra-précision à 40 bits pour calculs intensifs.
- Manipulation de bits, utilisée dans la manipulation des données de contrôle.

II.6.2.2 Architecture et caractéristiques du C6711

Le C6711 est constitué de trois parties principales : CPU, mémoires, et périphériques,

figure II.5.

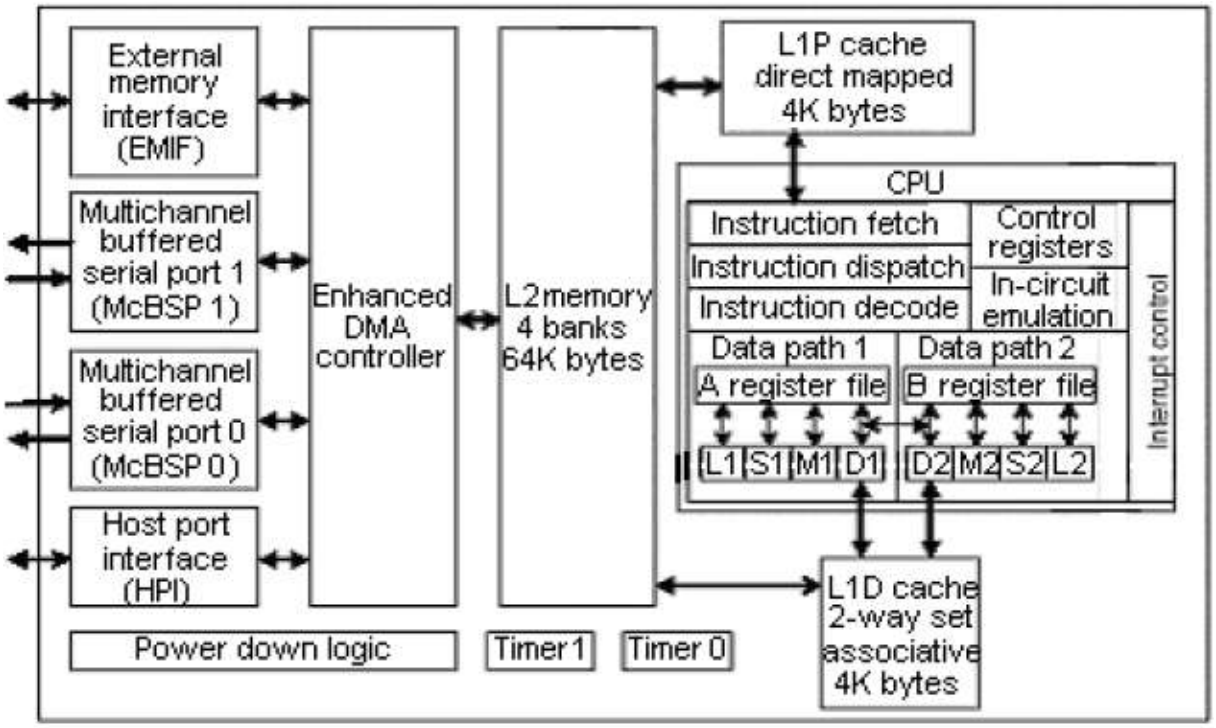


Figure II.5 : Constitution interne du DSP TMS320C6711

Les registres généraux sont au nombre de 32, tous à 32 bits. Il y a dix registres de contrôle (état, mode d'adressage, etc..). Trois registres additionnelles pour la gestion d'opération en virgule flottante.

En plus d'une RAM interne de 64ko, les périphériques internes les plus importants sont : deux ports série McBSP (multi-channel buffered serial port), deux Timers, et le DMA amélioré (EDMA).

Le C6711 communique avec le McBSP à travers dix registres de contrôle à 32 bits. Un timer est un compteur programmable à 32 bits, contrôlé par trois registres à 32 bits chacun.

L'EDMA assure le transfert d'un bloc de données ou celui des requêtes de périphériques. Il soulage ainsi le CPU des tâches fastidieuses. Seize canaux sont gérés individuellement par les événements (interruptions) correspondants, en déclenchant les transferts de données associés.

A part les registres de contrôle qui sont au nombre de huit, les paramètres de transfert sont stockés dans des tables en mémoire interne du EDMA sur 2 ko.

Il y a huit registres de contrôle pour gérer les interruptions. Le DSP supporte 16 vecteurs d'interruption à priorité configurable.

Le DSP dispose de deux jeux d'instructions, un pour les opérations en virgule fixe et l'autre pour les opérations en virgule flottante.

Conclusion

Nous avons montré l'intérêt de l'utilisation des DSP dans les différents domaines de traitement de signal, en exposant ses avantages.

D'autre part, autant les DSP sont d'une efficacité redoutable lorsqu'on les utilise en bon escient, autant leurs performances s'effondrent dès lors qu'on les sort de ce cadre. L'un des points critiques concerne le volume susceptible d'être traité. La capacité d'adressage des DSP est par nature limitée de quelques dizaines à quelques centaines de Kmots. Toute tentative pour leur faire gérer le plus grand espace mémoire, se traduit immanquablement par une diminution sensible du temps d'exécution, qui devient alors similaire ou inférieure à celle des processeurs CISC. Après avoir choisi d'utiliser le DSP C54x de Texas instruments, qui était un DSP à 16 bits en virgule fixe, le manque d'un compilateur C complet de son simulateur l'a fait écarter au profit d'un autre DSP de la famille C67x, à 32 bits en virgule flottante, cette fois-ci le compilateur C de son simulateur était disponible, en plus de la bibliothèque DSPLIB à fonctions spécialisées en traitement de signal.

CHAPITRE III

IMPLEMENTATION SUR DSP DU FILTRAGE HOMOMORPHIQUE

L'implémentation de l'analyse homomorphique en langage C sur DSP de type C6711 est accompagnée de l'implémentation sur Matlab, à titre de comparaison et vérification des résultats.

III.1 Analyse homomorphique sur Matlab

III.1.1 Données de traitement

Un fichier audio contenant le phonème de la voyelle A, a été enregistré sous le nom A8k16M.wav. L'enregistrement est sous format PCM 16 bits, avec l'attribut monophonique à échantillonnage 8 KHz. La zone utile entière sans silence, est située entre les échantillons n°2320 et n°4000. Soit 1680 échantillons sur 210 ms, figure.III.1.

Pour une segmentation de la parole par paquet de $2^8 = 256$ échantillons, cela équivaut à une durée de 32ms contenant 2 périodes du pitch minimal à 62.5Hz.

Le fichier A_Liftrage.m permet d'extraire un segment de 256 échantillons, situé approximativement au centre de la zone utile du fichier A8k16M.wav, entre les numéros 3088 et 3343. Ce segment est enregistré en mode ASCII dans le fichier son_256.dat, en utilisant l'instruction : save ('son_256.dat', 'e', '-ascii'). Ce qui permet de réaliser un fichier de données d'entrée pour le traitement sur DSP (le même segment traité par Matlab).

Un prétraitement du segment consiste à son fenêtrage Hamming sur la même durée de 32ms, figure.III.1. Le traitement quant à lui, vise à aboutir aux liftrages passe-bas et passe-haut du segment. Dans ce contexte, on passe par le calcul du module du spectre et son logarithme, figure.III.1.

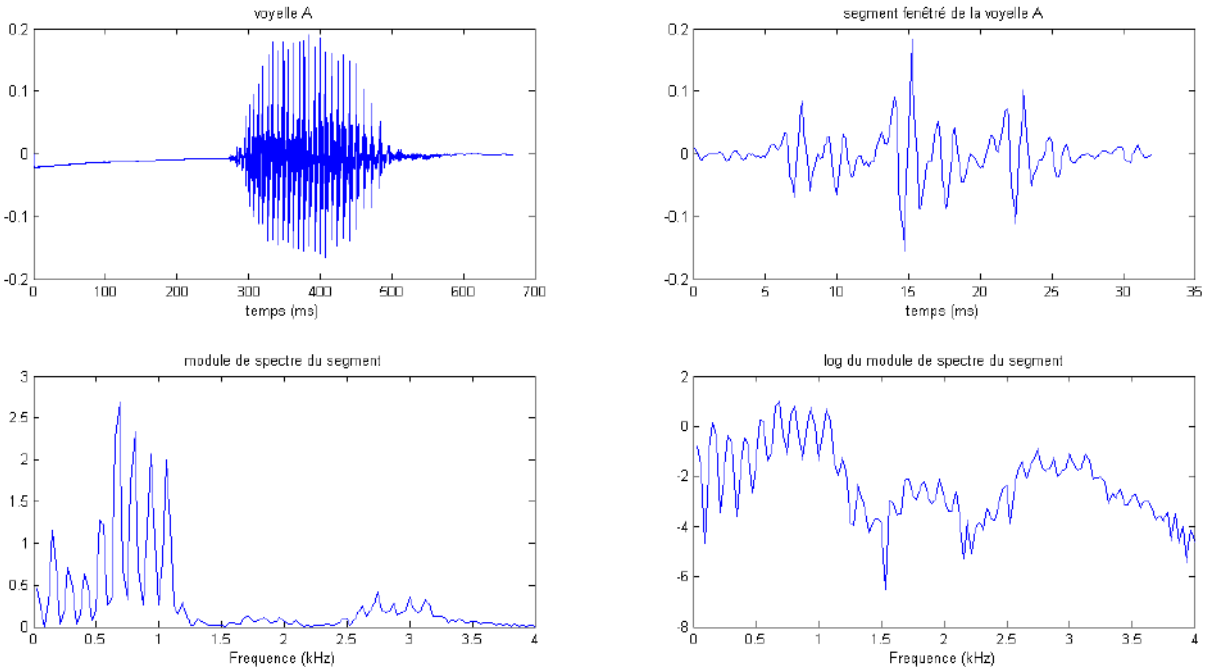


Figure.III.1 : phonème A (haut à gauche), segmentation et fenêtrage (haut à droite), module du spectre (bas à gauche), et son logarithme (bas à droite)

III.1.2 Liftrage :

Le cepstre réel étant la transformée ifft du log du module du spectre, il se compose dans le domaine des quéfrences, d'une partie relativement basse correspondant au signal dû à l'effet du conduit vocal et qui s'étend sur les 18 premiers échantillons, soit 2.25ms. Une deuxième partie relativement haute, correspond au signal glottique dont un pic est repéré à 7.75 ms, soit un pitch de 129 Hz du locuteur enregistré, figure.III.2.

La limitation du cepstre par une fenêtre rectangulaire de largeur optimale s'étendant sur 2.25ms et retenant ainsi uniquement l'effet du conduit vocal, constitue un liftrage passe-bas dont le spectre obtenu est lissé. Quatre formants peuvent être mesurés aisément sur le spectre lissé, figure.III.2 (courbe rouge). En utilisant une fenêtre rectangulaire complémentaire à la précédente, la limitation du cepstre retient cette fois-ci sa partie haute par liftrage passe-haut. Le spectre obtenu est propre au signal d'excitation dépourvu de l'effet du conduit vocal, figure.III.2 (courbe rouge).

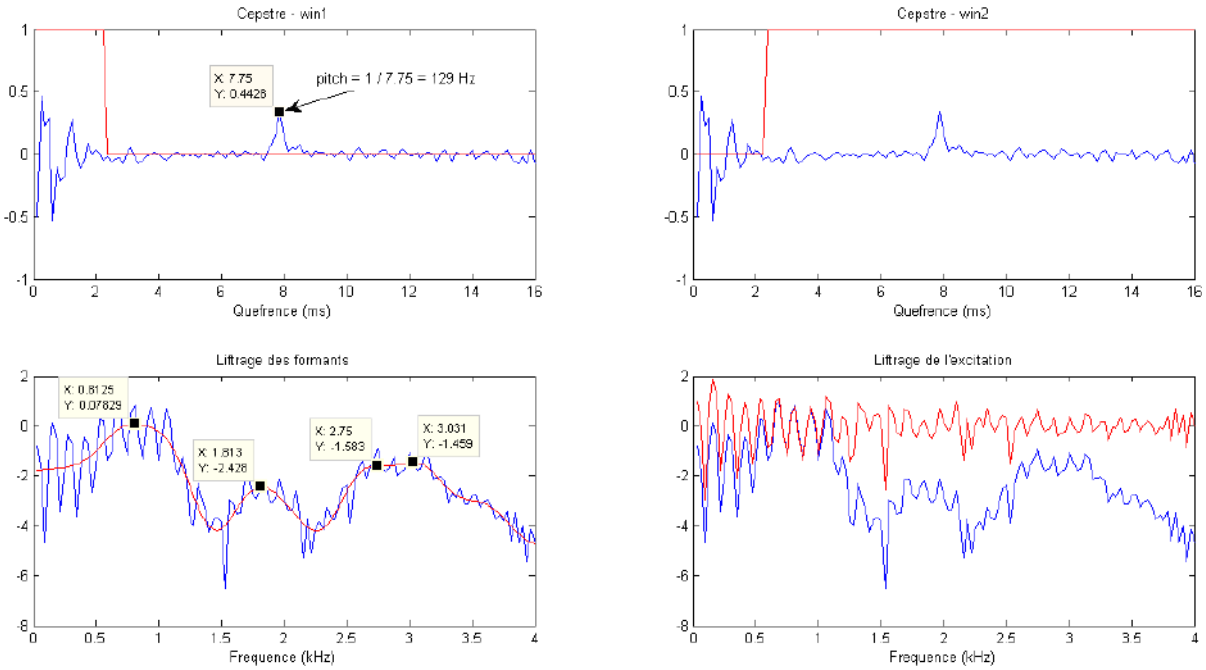


Figure.III.2 : cepstre et fenêtrage passe-bas (haut à gauche) et passe-haut (haut à droite), liftrage des formants (bas à gauche), et liftrage de l'excitation (bas à droite)

III.2 Implémentaion du liftrage sur DSP :

III.2.1 Fichiers d'application :

En plus d'un fichier en assembleur pour l'initialisation du point de départ, c_int00, au lancement du programme, et un fichier d'édition de lien déclarant une mémoire globale, SDRAM interne et externe de 16 Mo, et l'affectant aux différentes sections prédéfinies du compilateur C, contenant code, données, variables, tableaux, et constantes de notre programme, le projet 2_liftrage.pjt s'articule sur le fichier Principal.c qui inclut parmi les fonctions de la bibliothèque standard du C, la fonction math.h. Cette dernière fait partie de la bibliothèque rts6701.lib et ne renferme pas malheureusement, des algorithmes élaborés tel que la transformée fft et son inverse dont on a besoin. Ces transformées ont été dénichées par contre, à l'intérieur d'un fichier téléchargé, C67x_DSPLIB_sprc121.zip, où figure la bibliothèque dsp67x.lib réunissant des fonctions de traitement du signal. La compilation de cette dernière avec notre fichier d'application Principal.c n'ayant pas réussie, nous étions contraints d'extraire la partie qui nous intéresse, sous forme d'un fichier en assembleur linéaire, dsp67x.sa, contenant les fonctions, DSPF_sp_fftSPxSP pour la FFT complexe et DSPF_sp_ifftSPxSP pour la iFFT complexe.

Ces deux fonctions réclament à leur tour, l'utilisation de la fonction gen_twiddle qui génère les facteurs (twiddle) d'optimisation de l'algorithme FFT et son inverse, et ce, en permettant un accès linéaire à travers les données autorisant un niveau de parallélisme des données et des instructions (voir commentaire sur fichier dsp67x.sa). La fonction gen_twiddle est incluse dans le fichier d'accompagnement tw_fftSPxSP_C67.c. En outre, l'inversement de bit en papillon requis par la FFT rapide, est fourni dans le fichier d'entête brev_table.h.

Parlant d'entête, le fichier de données son_256.dat nécessite une entête afin qu'il puisse être lu par le DSP. L'entête est 1651 4 B648 1 100, ce qui signifie :

1651 : nombre magique choisi par la firme TI,
4 : désigne le format réel (float) des données du fichier,
B648 : adresse du buffer d'entrée, lieu d'aquisition des données du fichier,
1 : désigne l'espace mémoire de données (0 pour mémoire de programme),
100 : taille du segment de données en hexadécimal (256 échantillons).

III.2.2 Traitement par DSP :

La lecture des données du fichier son_256.dat se fait à l'aide de la sonde « probe point » vers le buffer ech[256]. Après fenêtrage de Hamming du segment de la parole, figure.III.3, les échantillons réels d'entrée sont rendus complexes en leur intercalant des zéros en guise de partie imaginaire. Après calcul en assembleur de la FFT complexe, sont déterminés, le module du spectre et son logarithme, figures 3 et 4, qui sont les mêmes que ceux obtenus par Matlab, figure.III.1.

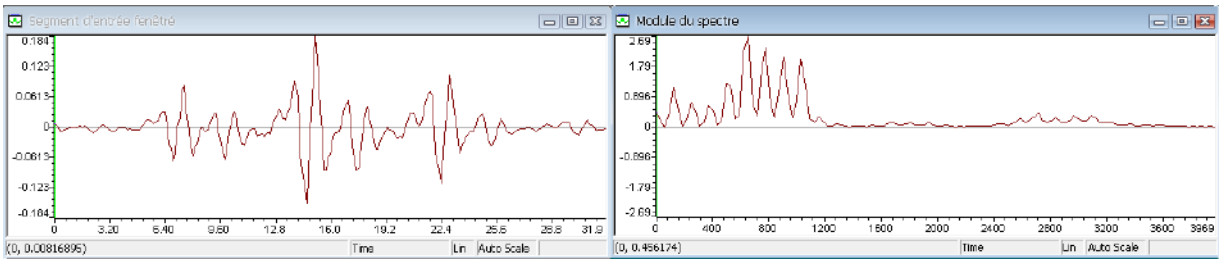


Figure.III.3 : segment d'entrée fenêtré (à gauche), module du spectre (à droite)

Le cepstre réel obtenu par la fonction DSPF_sp_ifftSPxSP, accuse un pic relatif au pitch à 7.75ms, figure 4, comme en figure.III.2 fournie par Matlab.

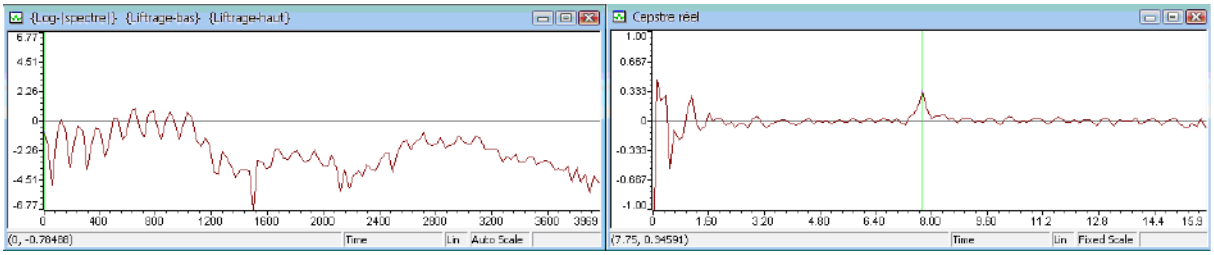


Figure.III.4 : log du module du spectre (à gauche), cepstre réel (à droite)

Le liftrage passe-bas dans le domaine des quéfrences et dans celui des fréquences, est donné en figure.III.5. Le spectre lissé dénonçant les formants du phonème, est identique à celui fourni par Matlab, figure.III.2.

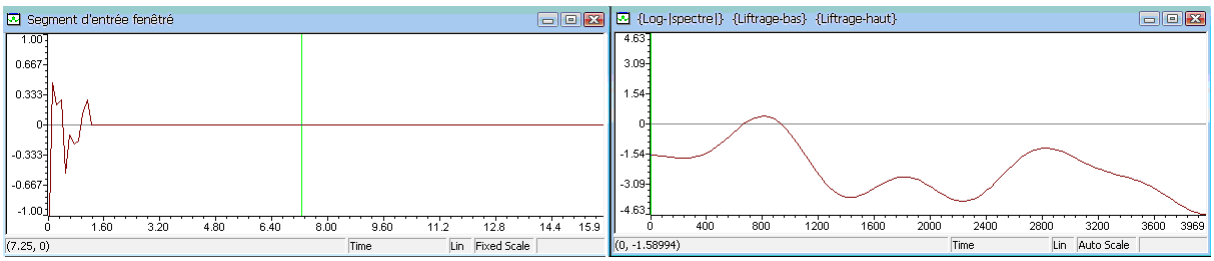


Figure.III.5 : liftrage passe-bas en quéfrence (à gauche), et en fréquence (à droite)

De même, le liftrage passe-haut dans le domaine des quéfrences et dans celui des fréquences, est donné en figure.III.6. Le spectre de l'excitation, est également identique à celui fourni par Matlab, figure.III.2.

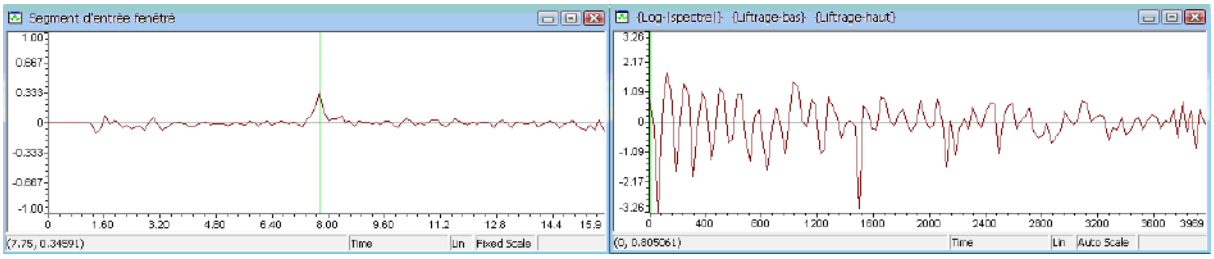


Figure.III.6 : liftrage passe-haut en quéfrence (à gauche), et en fréquence (à droite)

III.3 Utilisation du filtrage Mel :

La bande MEL est répartie suivant les fréquences [19] :

$$F_{mel} = \frac{1000}{\log_{10} 2} \log_{10}\left(1 + \frac{f}{1000}\right) \quad \text{.....(III.1)}$$

Le banc de filtre mel préconisé en pratique pour une bande de 4kHz de la parole [20], comporte 22 ou 24 filtres opérant sur l'énergie du signal d'entrée prétraité par fenêtrage Hamming et préaccentuation. L'énergie est tirée à partir du spectre du signal (module au carré du spectre X(k)), alors que le filtrage mel au rang r, MF(r), assure l'opération suivante [20] :

$$MF(r) = \log_{10}\left(\frac{\sum_{k=L_r}^{U_r} \left|V(k)\, X(k)\right|^2}{\sum_{k=L_r}^{U_r} \left|V(k)\right|^2} \right) \quad \text{.....(III.2)}$$

Où V(k) sont les paramètres de construction propre au filtre r, s'étendant d'une limite inférieure L_r (lower) vers une limite supérieure U_r (upper). Le dénominateur apporte une pondération du filtre qui assure une certaine normalisation en amplitude. Les coefficients cepstraux filtrés en mel, MFCC, sont alors déterminés en prenant la transformée en cosinus du signal filtré MF(r).

Ces coefficients sont une approximation du cepstre, et fournissent simplement une représentation orthogonale et compacte du log du module du spectre [21] [22].

Dans le banc de filtre mel, la détermination du nombre de filtre N_{Lin} opérant dans la partie fréquentielle linéaire (au dessous de 1kHz) et du nombre de filtre N_{Log} dans celle logarithmique (au-dessus de 1kHz), s'est effectuée de la manière approximative suivante :

Selon la figure.III.7, on peut écrire pour la partie à espacement linéaire :

$$f_1 + x_{Lin}.N_{Lin} = f_2 \quad \rightarrow \quad x_{Lin}.N_{Lin} = f_2 - f_1 = f_2$$

Pour x_{Lin} = 100Hz, nous obtenons : $N_{Lin} = f_2 / x_{Lin}$ = 1kHz / 0.1kHz = 10 filtres.

Pour la partie à espacement logarithmique :

$\log(f_2) + N_{Log} \log(x_{Log}) = \log(f_3) \rightarrow \log(f_2) + \log(x_{Log}^{NLog}) = \log(f_3) \rightarrow f_2.x_{Log}^{NLog} = f_3$

Pour $N_{Log} >= N_{Lin}$, et en fixant N_{Log} = 13 filtres, on obtient $x_{Log} = (f_3 / f_2)^{1/NLog} = 4^{1/13}$ = 1.1125. Mais en recalculant tout en tenant en compte cet espacement, on aboutit à f_3 = 4.45kHz au lieu de 4kHz fixé au départ. Le bon ajustage donne N_{Log} = 12, soit un nombre total des filtres du banc atteignant les 22.

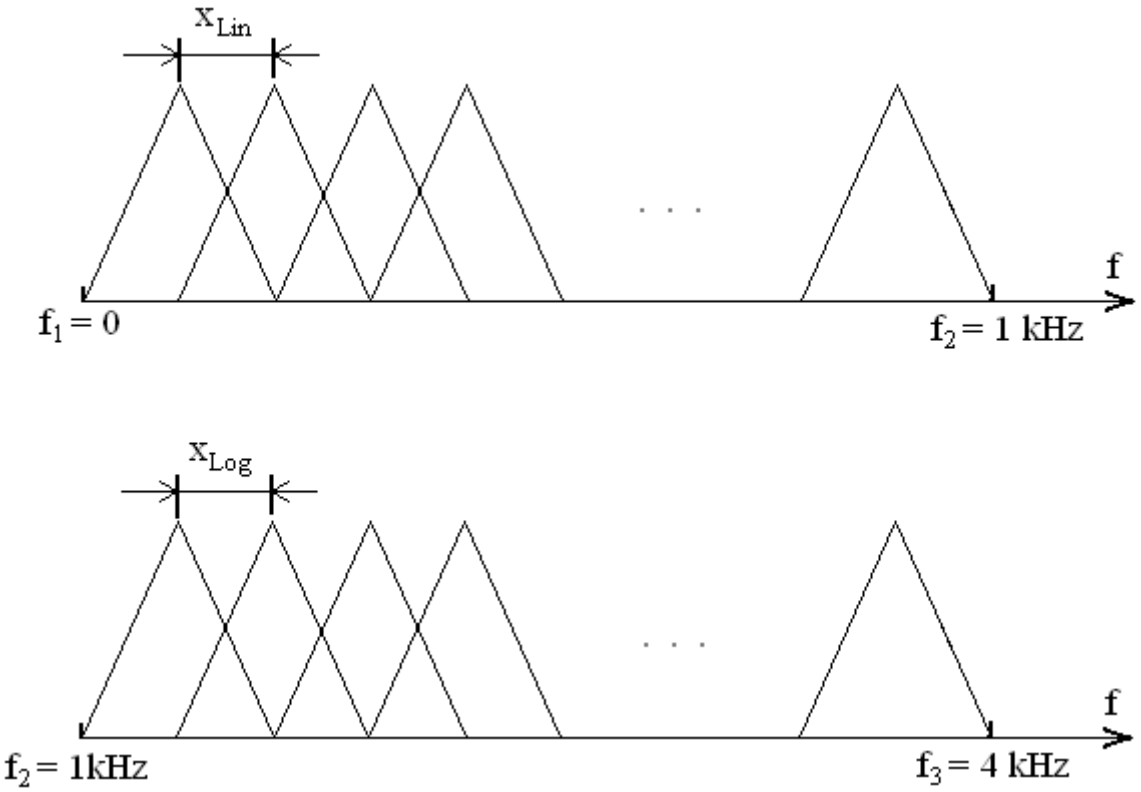


Figure.III.7 : espacement linéaire des filtres jusqu'à 1 kHz et logarithmique après

Dans la fonction Matlab banc.m, nous choisissons un banc de 22 filtres mel occupant une bande de 4 kHz. Nous prenons 10 filtres à espacement linéaire de 100, et 12 filtres à espacement logarithmique de 1.11253. Une normalisation (<1) du poids de chaque filtre est effectuée en fonction des limites de bande, basse et haute :

TailleTriangle = 1./(haut-bas)

Ce qui se manifeste par la décroissance des sommets de gabarit des filtres, notamment pour ceux opérant au-dessus de 1 kHz et dont les bandes s'élargissent en allant vers la fréquence

limite de 4 kHz. Le fichier D_MFCC.m exploite la fonction banc.m pour visualiser la construction du banc MEL, figure.III.8.

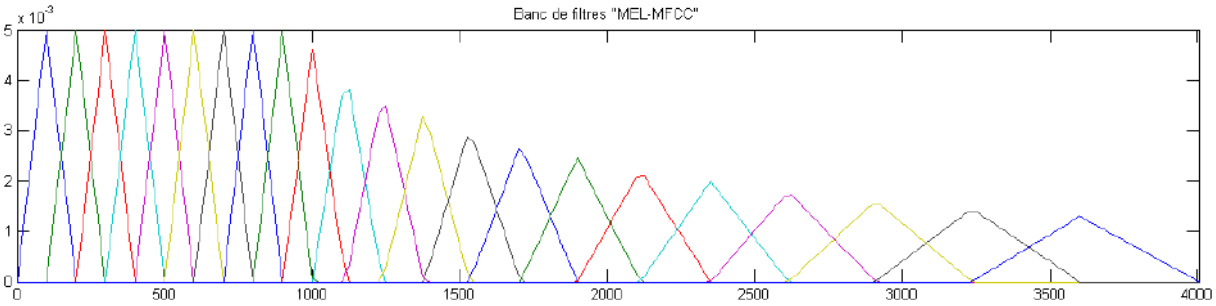


Figure.III.8 : banc à 22 filtres MEL

Le profil des différents gabarits du banc de filtres suit mieux la théorie que celui développé par la fonction melbankm() de voicebox dans Matlab.

Le même banc de filtre mel a été implémenté sur DSP, où le projet 4_BancFiltreMEL présente l'ensemble des 22 filtres du banc, étalés en fréquence, figure.III.9. L'outil « graphe » du CCS est configuré dans le domaine temporel pour représenter celui fréquentiel. Il y a deux solutions pour la représentation graphique des différents gabarits du banc. Une première représentation dans laquelle une même origine est gardée pour tous les gabarits de filtre, voir le tracé du bas de la figure.III.9. Mais dans ce cas, en dépit de l'exactitude des valeurs affichées sur l'axe fréquentiel, le graphe ne permet pas de sauvegarder des superpositions montrant le chevauchement des différents gabarits, comme en figure.III.8.
La deuxième représentation possible, touche simultanément l'ensemble des gabarits mais à condition qu'ils soient mis côte à côte, voir les trois tracés du haut de la figure.III.9 et dans ce cas, l'axe fréquentiel indique le cumul des différents espacements entre gabarits.

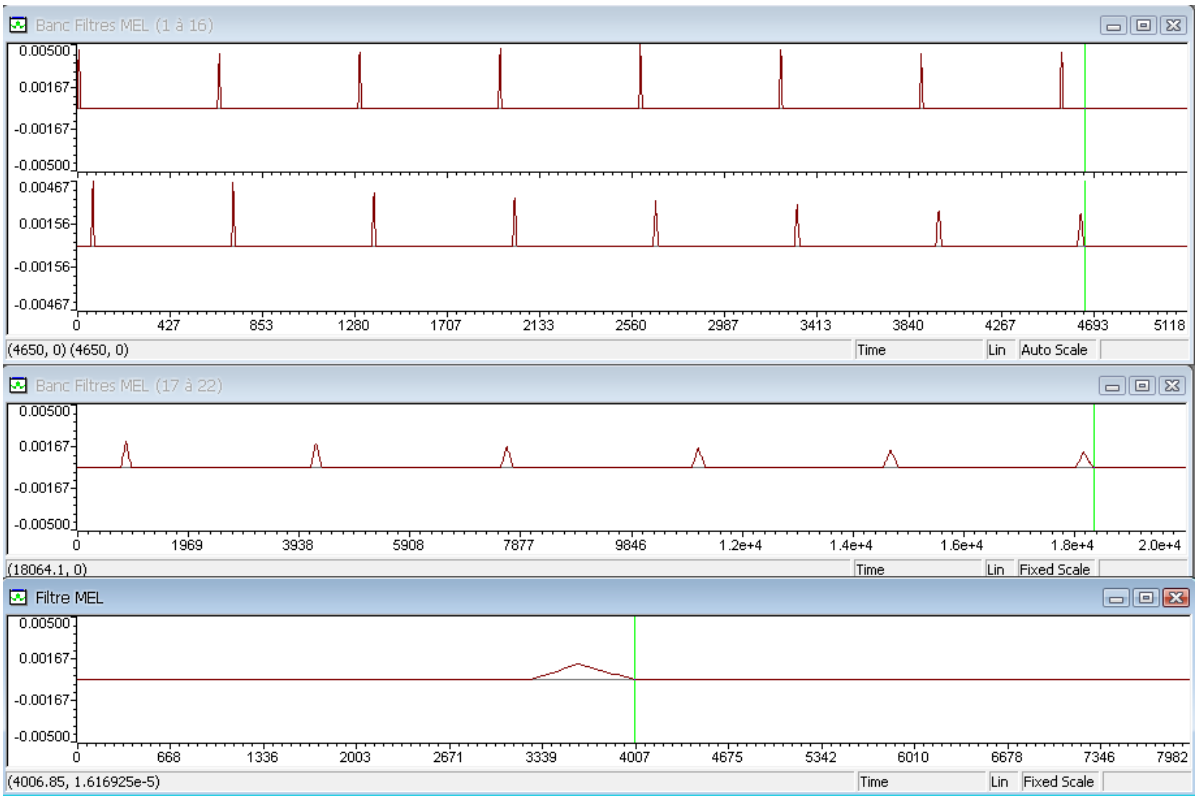


Figure.III.9 : Etalement fréquentiel des 22 filtres MEL, le dernier gabarit est représenté seul au tracé du bas

III.4 Analyse MFCC :

Dans le fichier E_Spectres_MelMfcc.m, une fois l'énergie spectrale du segment de la parole déterminée (carré du module du spectre), elle passe à travers le banc de filtres Mel en appelant la fonction banc.m.
Le spectre mel cepstral lissé (melcep) résultant est mis en échelle dB puis pondéré par l'énergie totale du segment. Cette dernière est calculée comme étant la somme des 256 raies de l'énergie spectrale puis rajoutée en dB, au spectre melcep exprimé lui aussi en dB.

En calculant la transformée en cosinus (rdct) du spectre melcep (pondéré et mis en échelle dB), vingt deux coefficients MFCC sont alors déduits. Une interpolation à l'aide de la fonction spline rassemble les 22 coefficients sous forme d'une courbe, figure.III.10.
La transformée en cosinus d'un vecteur x de longueur N, définie dans Matlab R2010a, s'écrit :

$$y(k) = w(k).\sum_{n=1}^{N} x(n).\cos\left[\frac{\pi(2n-1)(k-1)}{2N}\right]; \qquad k = 1,2,...,N \quad \text{.....(III.3)}$$

Où, $w(k) = \begin{cases} 1/\sqrt{N} & k = 1 \\ \sqrt{2/N} & 2 \le k \le N \end{cases}$

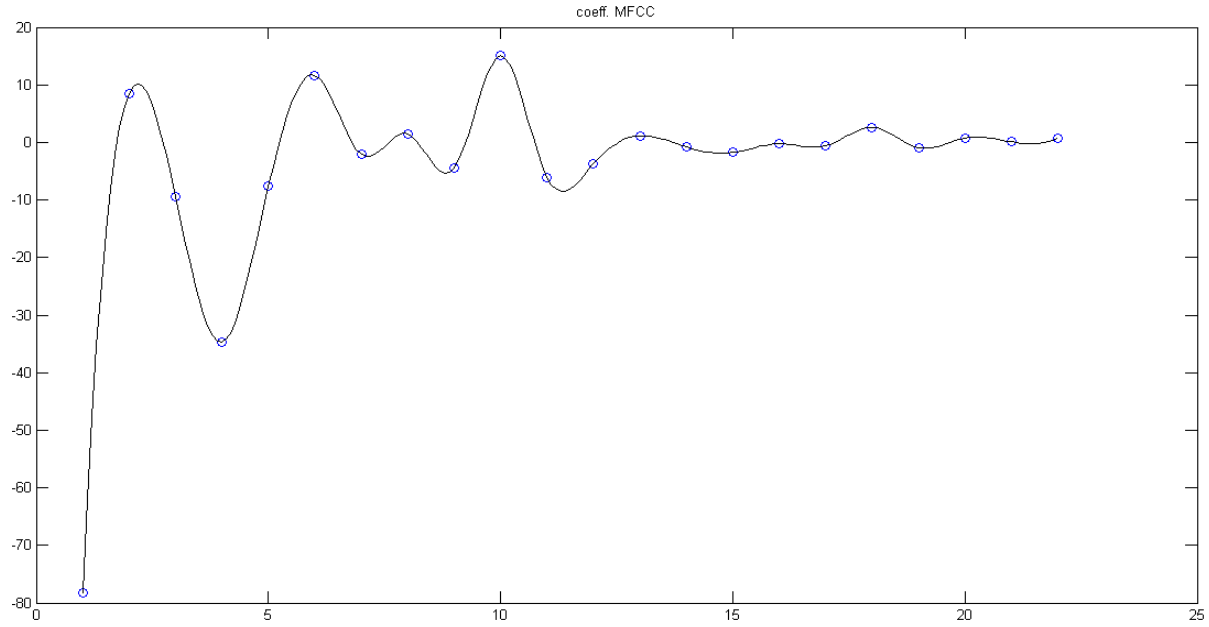


Figure.III.10 : les 22 coefficients MFCC interpolés à l'aide de la fonction spline

Le spectre mel cepstral lissé (melcep), est retrouvé par transformée inverse irdct des coefficients MFCC. En réunissant les 22 points du spectre melcep par interpolation spline, la figure.III.11 montre la trace résultante (en noir) en superposition au tracé en dB du spectre du segment de la parole (en rouge) et au tracé dû au liftage passe-bas du cepstre (en vert). On constate que le spectre melcep épouse mieux la forme du spectre |FFT| en dB, que celui issu du cepstre liftré.

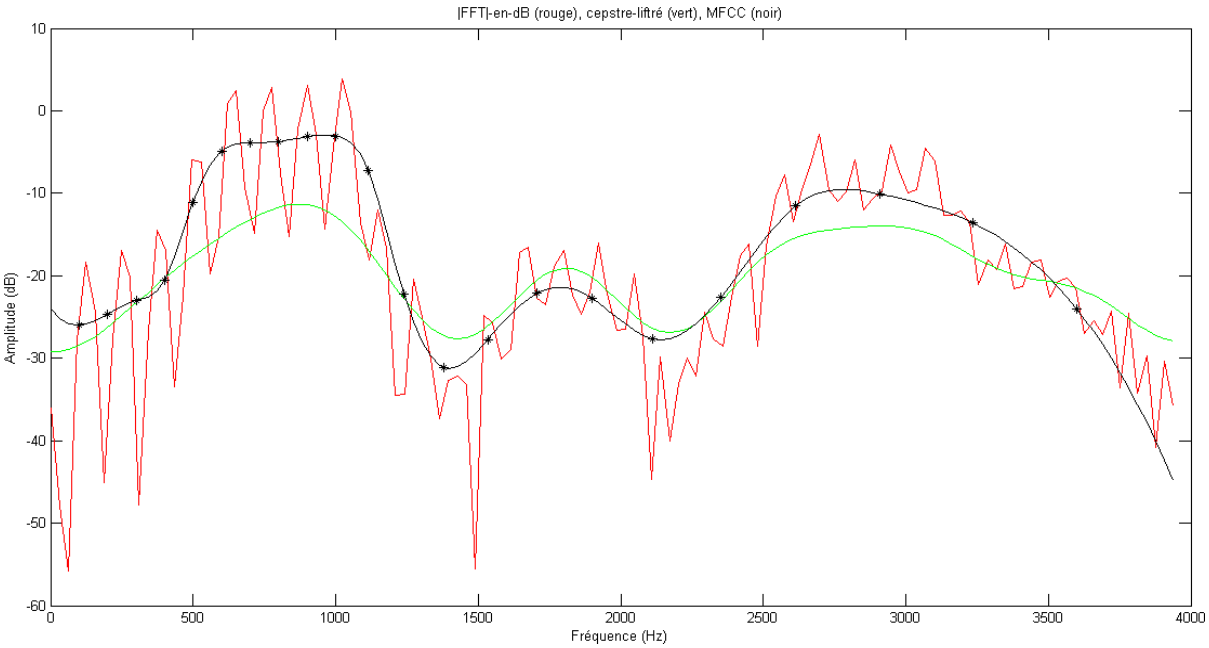


Figure.III.11 : spectre en dB (rouge) confronté aux traitements par liftrage (vert) et filtrage mel interpolé par spline (en noir)

La figure.III.12 montre le tracé melcep isolé des deux autres tracés de la figure.III.11, afin de mieux le comparer à celui fourni par calcul DSP. On remarque que les 22 points du spectre sont linéairement espacés jusqu'à 1 kHz, puis logarithmiquement espacés après. Ces espacements entre les points suivent au fait, ceux du banc de filtres mel, figures.III.8 et III.9.

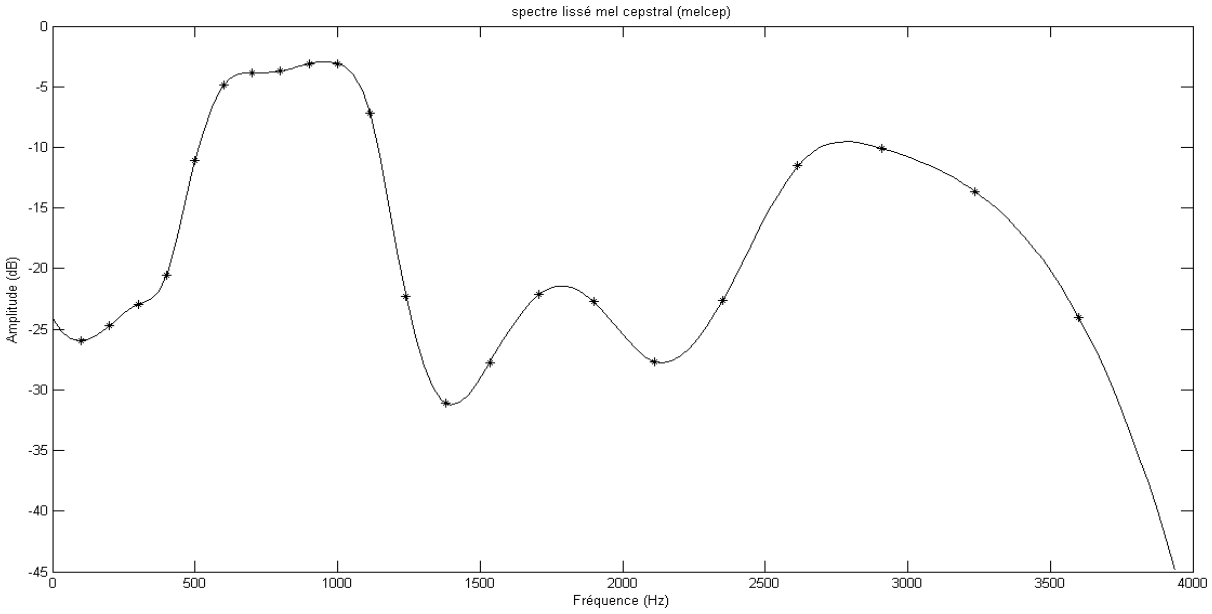


Figure.III.12 : isolation du spectre par filtrage mel (interpolé par spline)

L'implémentation en langage C du projet 5_mfcc sur DSP, traite le segment de la parole prétraité par préaccentuation et fenêtrage de Hamming en calculant le carré du module du spectre ou l'énergie spectrale durant les 32ms du segment. L'énergie spectrale est alors filtrée en mel et pondérée avec l'énergie totale du segment, puis remise à l'échelle dB. Le signal résultant est le spectre lissé (melcep), figure.III.13. La transformée en cosinus du spectre melcep permet de retrouver les MFCC, figure.III.13.

La définition en Matlab de la transformée en cosinus, expression (III.3), utilise des indices commençant à partir de 1. Par contre dans le langage C, les indices de boucles peuvent commencer à partir de 0. Ainsi, les indices n et k allant de 1 à N, seront remplacés respectivement par les indices i et j allant de 0 à N-1. Dans le projet 5_mfcc sur DSP, l'expression (III.3) est donc réécrite sous la forme :

$$y[j] \mathrel{+}= ((j==0)*sqrtf(1/N) + (j!=0)*sqrtf(2/N)) * x[i] * cosf(j * PI * (i+0.5)/N) \quad \text{.....(III.4)}$$

Où i, j = 0,1,…,N-1.

On remarque que l'instruction « if » nécessaire pour tenir compte des deux valeurs de pondération de w(k) dans (III.4), est remplacée par une écriture plus compacte faisant multiplier chaque valeur par sa condition booléenne.

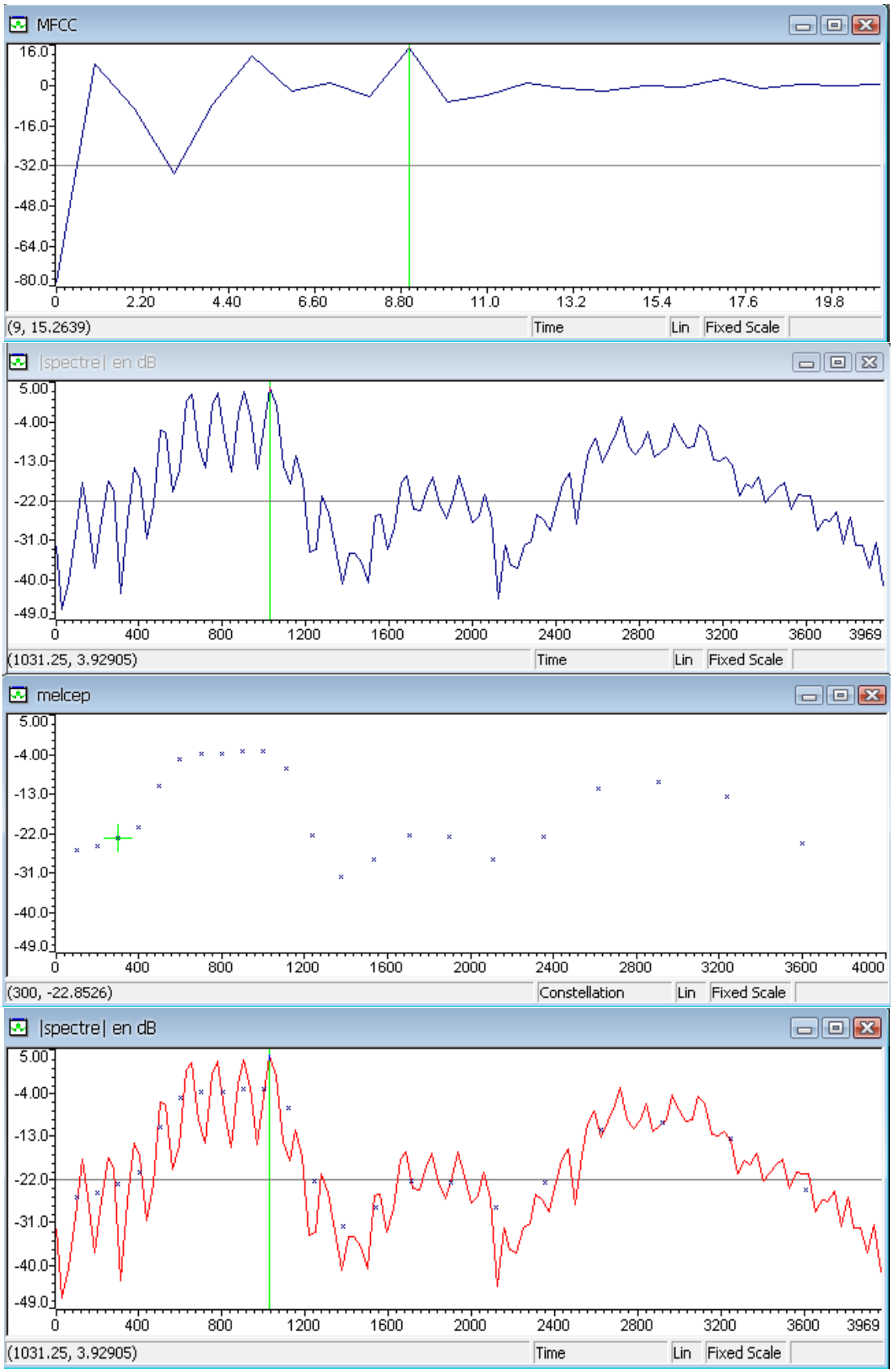


Figure.III.13 : MFCC (en haut), |FFT| en dB et spectre melcep en pointillé (au milieu), superposition |FFT| en dB et melcep (en bas)

Les valeurs des MFCC de la figure 13 ont toutes été vérifiées et sont quasiment les mêmes qu'en figure.III.10. Sauf que l'interpolation à l'aide de segments de droite du CCS en figure.III.13, est moins bonne que celle menée par la fonction spline de Matlab en figure.III.10.

Le spectre melcep en figure.III.13, est discrétisé en 22 points (impossible de les interpoler) en utilisant un graphe de type constellation. C'est le seul type dans CCS qui permet de représenter un axe quelconque pour les abscisses, pas forcément régulier comme l'axe temporel. Le spectre melcep discrétisé obtenu s'obtient également par transformée inverse en cosinus des coefficients MFCC, et il suit bien celui de la figure.III.12.

Un autre problème lié au CCS, réside dans l'incapacité de superposer le spectre melcep au spectre classique exprimé en dB, figure.III.13. Par conséquent, la superposition a été effectuée manuellement au bas de la figure.III.13, ce qui permet de conclure à son identité à la figure.III.11 de Matlab.

III.5 Optimisation de l'application :

III.5.1 Profileur du CCS :

Le composeur de code dans l'environnement de développement des applications sur DSP, permet la collecte des statistiques d'exécution de certaines régions du code [23]. Cela s'appelle profilage, et donne une mesure de la performance de l'application permettant ainsi d'optimiser son code. Principalement, les temps écoulés en terme de cycles machines consommés pendant l'exécution de tout le programme ou d'un certain nombre de ligne de code ou même certains sous-programmes, sont déterminés pendant le profilage. Cela permet la vérification de l'exécution en temps réel ou d'apporter des optimisations dans ce sens.

Au menu « Profiler » du CCS, il faudra activer l'horloge de mesure « Enable Clock ». Puis dans la fenêtre de dialogue du menu « Profiler\Clock Setup », choisir une durée de cycles de 6.667ns suite à la fréquence d'horloge 150MHz du C6711. En dernier, la session de profilage est appelée « tempo », comme temporisation. La fenêtre des statistiques du profil s'ouvre. Elle contient plusieurs champs pour chaque point de profil. Chaque point affiche le nombre de fois qu'il est rencontré et les mesures statistiques sur le nombre de cycles écoulés. Les statistiques incluent le minimum, maximum, total, et moyenne « average » du nombre de cycles qui sont mis à jour chaque fois que le point de profil est atteint. La mesure qui nous intéresse est la moyenne du fait que le nombre total est le produit entre les champs « average » et « count ».

III.5.2 Optimisation du code :

A travers l'outil PBC (profile based compilation) de CCS, on peut optimiser le compromis entre taille du code et nombre de cycles pour l'application [24]. Utilisant une variété de configurations de profilage, PBC collecte des données sur les fonctions individuelles et détermine la meilleure combinaison des options du compilateur. PBC fournit un tracé de ces configurations projetées, permettant de choisir graphiquement la configuration s'adaptant le mieux à nos besoins. Nous avons préféré procéder autrement, en essayant toutes les options offertes par le compilateur C du CCS.

Dans la fenêtre de dialogue du menu « Project\Build options », la catégorie « Basic » du compilateur offre deux boîtes déroulantes d'optimisation, « Opt level » et « Program level Opt », qui par défaut sont sur l'option « None » indiquant « pas d'optimisation » [25].

Les options retenues et qui ont donné une amélioration d'exécution du point de vue rapidité, sont : « Opt level = Local (-o1) » et « Program level = No external var refs (-op3) ».

L'option « Local (-o1) » permet d'allouer des registres internes du DSP aux variables, d'accomplir des rotations de boucles, de simplifier des expressions en éliminant celles locales communes, et d'éliminer des assignements non utilisés.

L'option « No external var refs (-op3) » indique qu'il y a des fonctions appelées à partir d'autres modules mais ne doit pas y avoir de variables globales qui soient modifiées dans d'autres modules.

Dans ce qui suit, nous retenons à titre de comparaison les deux configurations du compilateur, sans et avec optimisation (au deux options précédentes). D'autre part, une expression de calcul située dans une boucle, a fait l'objet de modification en guise d'accélérer son exécution. Il s'agit du tableau ym[] dans le fichier d'application du projet 2a_liftrage_optimisation, dans lequel on a :

$1^{ère}$ forme (initiale) : ym[i] = sqrtf(powf(y[2*i],2) + powf(y[2*i+1],2))

$2^{ème}$ forme (modifiée): ym[i] = sqrtf(y[2*i]*y[2*i] + y[2*i+1]*y[2*i+1]).

III.5.3 Démarches et précautions à prendre lors des relevés statistiques :

- Au menu Profiler, on active "Enable clock", puis "start new session" au nom "tempo".
- Au pop-menu de la fenêtre statistique du Profiler, on clique sur "Property page", pour taper "Instruction cycle times (ns)" = 6.6666667, et choisir l'option "Cycle" ou "Time (ns)" (si une option est déjà choisie mais n'est pas opérationnelle, on passe à l'autre puis on revient), ensuite on clique sur bouton appliquer.

- Les 2 configurations de compilation ci-dessous, seront indiquées respectivement par « sans optimisation » et « avec optimisation » :

  Project\Build Options\Compiler\Basic\Opt Speed vs Size = Speed Most Critical
  \Opt Level = None
  \Program Level Opt = None

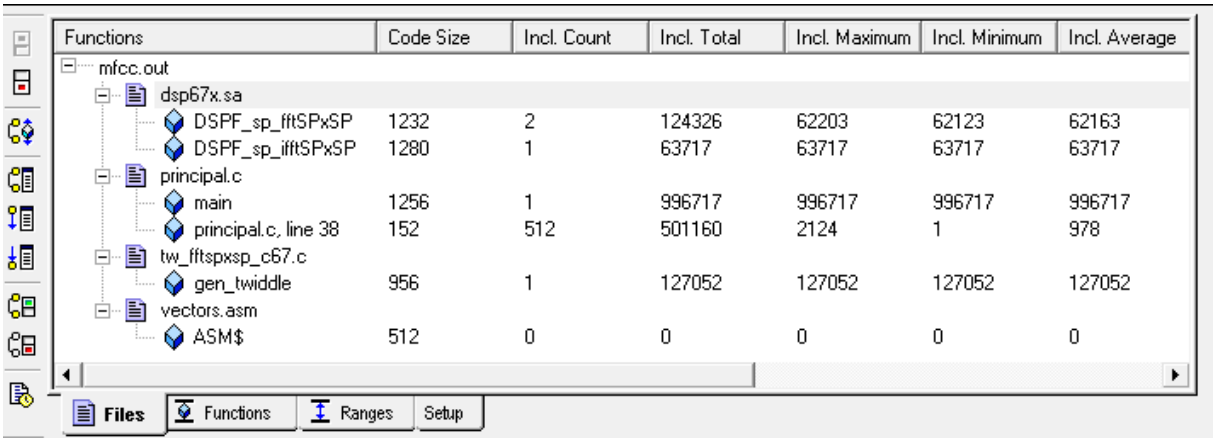
  Project\Build Options\Compiler\Basic\Opt Speed vs Size = Speed Most Critical
  \Opt Level = Local
  \Program Level Opt = No External Var Refs

- Les 2 écritures d'expression ym[] seront notées « 1ère forme » et « 2ème forme », voir ci-haut.
- En passant d'une configuration de compilation à l'autre, ou d'une écriture de la variable ym à l'autre, il faudra recopier la bonne adresse du buffer d'acquisition "ech[]" depuis la mémoire de données vers le fichier de lecture "son_256.dat".
- Il faudra aussi dans la fenêtre statistique de Profiler, rectifier le n° de ligne de l'expression de la variable ym choisie en positionnant d'abord le curseur sur la ligne de la variable puis sélectionner l'onglet "Ranges", cliquer le bouton "create profile area", puis choisir type "Range" et ok.
- Exécuter au moins 2 fois le programme, afin que les mesures statistiques se stabilisent.

III.5.4 Résultas du profileur :

Les données statistiques prises dans la fenêtre « Profiler », en terme de nombre moyen de cycles d'instruction "Incl. Average", sont affichées dans différentes figures selon les cas de compilation et d'optimisation de l'expression ym[]. La première forme de l'expression de la variable ym se trouve à la ligne 38 du programme d'application Principal.c, tandis que la seconde forme se trouve à la ligne 40. La fonction vectors.asm n'est pas profilée car elle est exécutée bien avant l'arrivée au programme principal « main ».

Compilation sans optimisation avec première forme d'expression ym[], figure.III.14.
On constate que le programme Principal.c (main) dans son entier réclame 996717 cycles, soit 6.645ms (il s'exécute donc à temps réel sous 8kHz d'échantillonnage mais ne l'est pas sous 44kHz). L'expression ym[] réclame quant à elle, 978 cycles ou 6.52us.

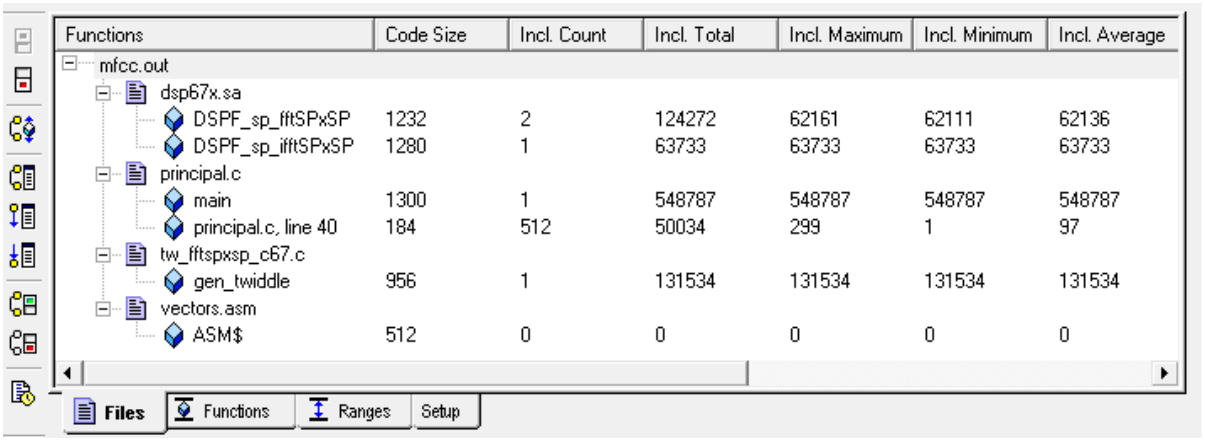


Functions	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
mfcc.out						
dsp67x.sa						
DSPF_sp_fftSPxSP	1232	2	124326	62203	62123	62163
DSPF_sp_ifftSPxSP	1280	1	63717	63717	63717	63717
principal.c						
main	1256	1	996717	996717	996717	996717
principal.c, line 38	152	512	501160	2124	1	978
tw_fftspxsp_c67.c						
gen_twiddle	956	1	127052	127052	127052	127052
vectors.asm						
ASM$	512	0	0	0	0	0

Files | Functions | Ranges | Setup

Figure.III.14 : fenêtre statistique du Profileur sans optimisation du compilateur ni l'équation ym

Compilation sans optimisation avec deuxième forme d'expression ym[], figure.III.15.
On constate que le programme Principal.c (main) dans son entier réclame 548787 cycles, soit 3.659ms (il s'exécute donc à temps réel sous 8kHz et même sous 44kHz). L'expression ym[] réclame quant à elle, 97 cycles ou 0.647us.



Functions	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
mfcc.out						
dsp67x.sa						
DSPF_sp_fftSPxSP	1232	2	124272	62161	62111	62136
DSPF_sp_ifftSPxSP	1280	1	63733	63733	63733	63733
principal.c						
main	1300	1	548787	548787	548787	548787
principal.c, line 40	184	512	50034	299	1	97
tw_fftspxsp_c67.c						
gen_twiddle	956	1	131534	131534	131534	131534
vectors.asm						
ASM$	512	0	0	0	0	0

Files | Functions | Ranges | Setup

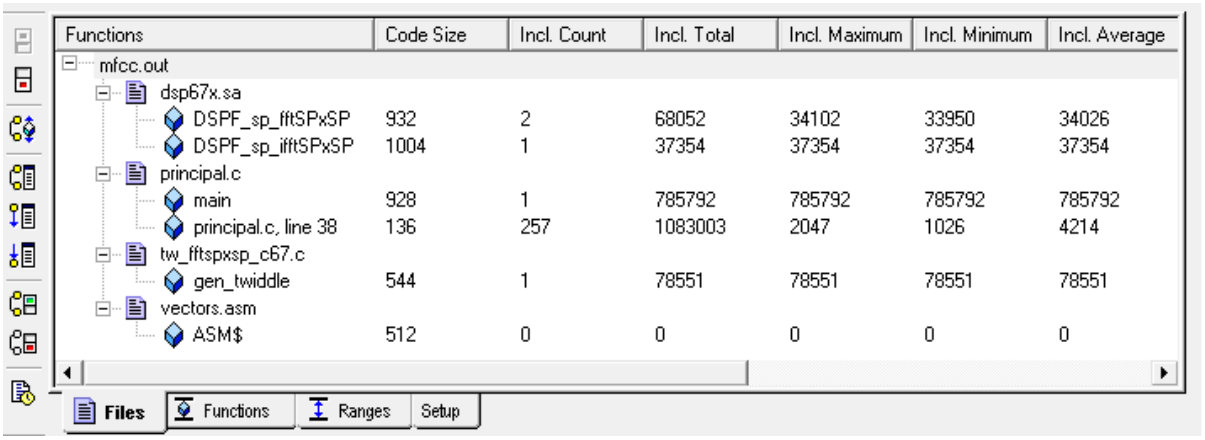
Figure.III.15 : Profileur avec optimisation de l'équation ym mais pas le compilateur

Normalement et au minimum, on devrait s'attendre à ce que la deuxième forme d'expression ym s'effectue au bout de 256 cycles (et non 97) pour le cas optimal d'un seul cycle par

instruction, soit 256 fois dans la boucle. Mais le parallélisme du pipeline réduit le nombre de cycles.

Compilation avec optimisation et première forme d'expression ym[], figure.III.16.

On constate que le programme Principal.c (main) dans son entier réclame 785792 cycles, soit 5.239ms (temps réel sous 44kHz d'échantillonnage). L'expression ym[] réclame quant à elle, 4214 cycles ou 28.093us.



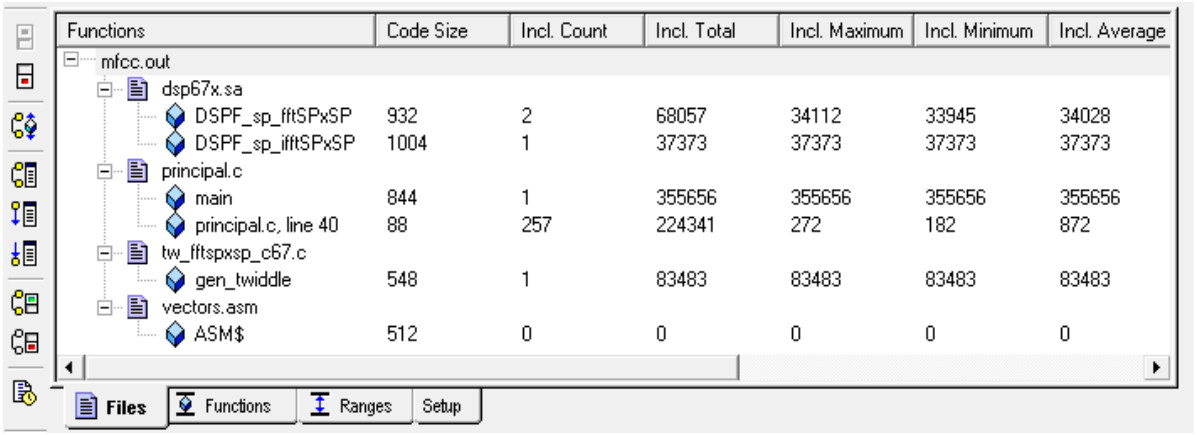
Functions	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
mfcc.out						
dsp67x.sa						
DSPF_sp_fftSPxSP	932	2	68052	34102	33950	34026
DSPF_sp_ifftSPxSP	1004	1	37354	37354	37354	37354
principal.c						
main	928	1	785792	785792	785792	785792
principal.c, line 38	136	257	1083003	2047	1026	4214
tw_fftspxsp_c67.c						
gen_twiddle	544	1	78551	78551	78551	78551
vectors.asm						
ASM$	512	0	0	0	0	0

Files | Functions | Ranges | Setup

Figure.III.16 : Profileur avec optimisation du compilateur mais pas l'équation ym

Compilation avec optimisation et deuxième forme d'expression ym[], figure.III.17.

On constate que le programme Principal.c (main) dans son entier réclame 355656 cycles, soit 2.371ms (temps réel sous 44kHz d'échantillonnage). L'expression ym[] réclame quant à elle, 872 cycles ou 5.813us.



Functions	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
mfcc.out						
dsp67x.sa						
DSPF_sp_fftSPxSP	932	2	68057	34112	33945	34028
DSPF_sp_ifftSPxSP	1004	1	37373	37373	37373	37373
principal.c						
main	844	1	355656	355656	355656	355656
principal.c, line 40	88	257	224341	272	182	872
tw_fftspxsp_c67.c						
gen_twiddle	548	1	83483	83483	83483	83483
vectors.asm						
ASM$	512	0	0	0	0	0

Files | Functions | Ranges | Setup

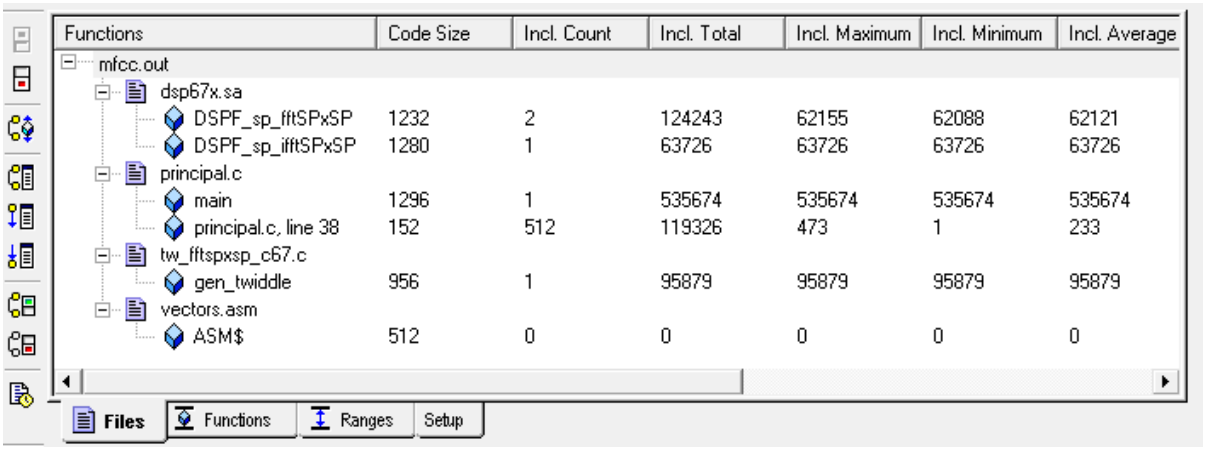
Figure.III.17 : Profileur avec optimisation du compilateur et de l'équation ym

On peut affirmer à ce stade d'optimisation, que la compilation sans optimisation permet quand même d'optimiser le calcul de la variable ym par rapport à la compilation avec optimisation, mais cette dernière optimise d'autres calculs dans « main » qui la réduisent davantage en temps d'exécution. Notamment, les fonctions en assembleur des transformées FFT et iFFT, et la fonction en c, gen_twiddle.

III.5.5 Remplacement de la bibliothèque standard :

La bibliothèque rts6701.lib se trouvant dans le dossier cgtools\lib renferme entre autres, les fonctions incluses dans notre application, telles que les standards <math.h> et <stdio.h>.
Nous l'avons remplacé par une bibliothèque plus rapide, fastrts67x.lib, mise dans le dossier mthlib\lib [26].
Il s'agit d'une bibliothèque de fonctions mathématiques optimisées en virgule flottante, pour les programmeurs en C sur la famille de DSP, C67x. Ces routines sont conseillées typiquement dans les calculs intensifs d'applications en temps réel, où la vitesse optimale d'exécution est critique. La bibliothèque fastrts67x.lib contient toutes les fonctions fournies par la bibliothèque standard rts6701.lib, en simple et double précision. Elles doivent être appelées par les anciens noms en leur rajoutant cependant, les suffixes sp pour le cas simple précision et dp pour double précision. Par exemple, la fonction exponentiel en double précision, exp() devient expdp(), et en simple précision, expf() devient expsp(). Pour l'utilisation dans une application, il faudra inclure l'entête <fastrts67x.h> dans le programme principal et tous ses sous-programmes.
On reprend à partir du projet 2b_liftrage_optimisation contenant la bibliothèque fastrts67x.lib, les résultats d'optimisation de notre application à l'aide du profileur, selon la procédure précédente.

Compilation sans optimisation avec première forme d'expression ym[], figure.III.18.
On constate que le programme Principal.c (main) dans son entier réclame 535674 cycles, soit 3.571ms (il s'exécute donc à temps réel sous 8kHz et 44kHz d'échantillonnage). L'expression ym[] réclame quant à elle, 233 cycles ou 1.553us.

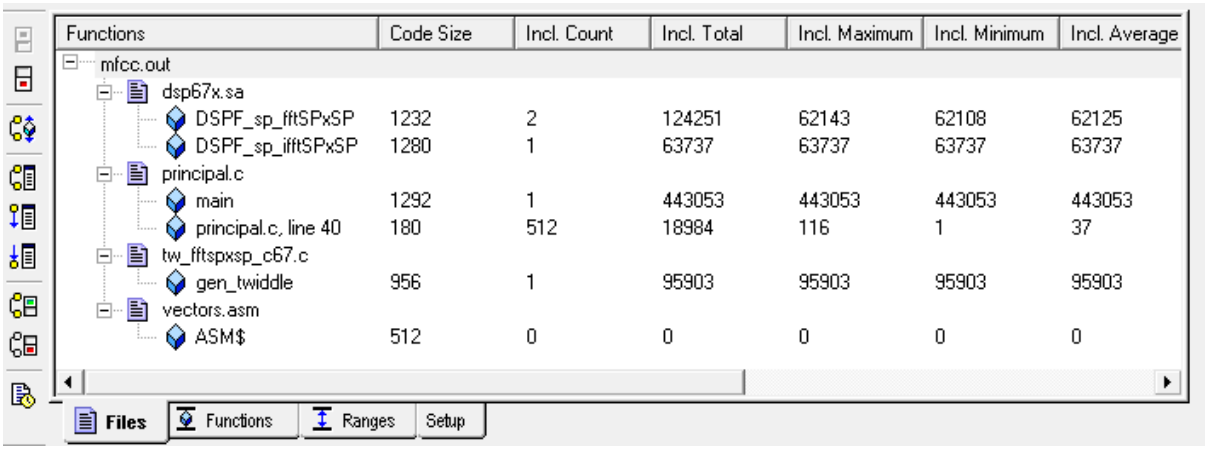


Functions	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
mfcc.out						
dsp67x.sa						
DSPF_sp_fftSPxSP	1232	2	124243	62155	62088	62121
DSPF_sp_ifftSPxSP	1280	1	63726	63726	63726	63726
principal.c						
main	1296	1	535674	535674	535674	535674
principal.c, line 38	152	512	119326	473	1	233
tw_fftspxsp_c67.c						
gen_twiddle	956	1	95879	95879	95879	95879
vectors.asm						
ASM$	512	0	0	0	0	0

Files | Functions | Ranges | Setup

Figure.III.18 : Profileur sous fastrts67x.lib sans optimisation du compilateur ni l'équation ym

Compilation sans optimisation avec deuxième forme d'expression ym[], figure.III.19.
On constate que le programme Principal.c (main) dans son entier réclame 443053 cycles, soit 2.954ms (temps réel à 44kHz, même pour des segments réduits jusqu'à 128 éch.). L'expression ym[] réclame quant à elle, 37 cycles ou 0.247us.



Functions	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
mfcc.out						
dsp67x.sa						
DSPF_sp_fftSPxSP	1232	2	124251	62143	62108	62125
DSPF_sp_ifftSPxSP	1280	1	63737	63737	63737	63737
principal.c						
main	1292	1	443053	443053	443053	443053
principal.c, line 40	180	512	18984	116	1	37
tw_fftspxsp_c67.c						
gen_twiddle	956	1	95903	95903	95903	95903
vectors.asm						
ASM$	512	0	0	0	0	0

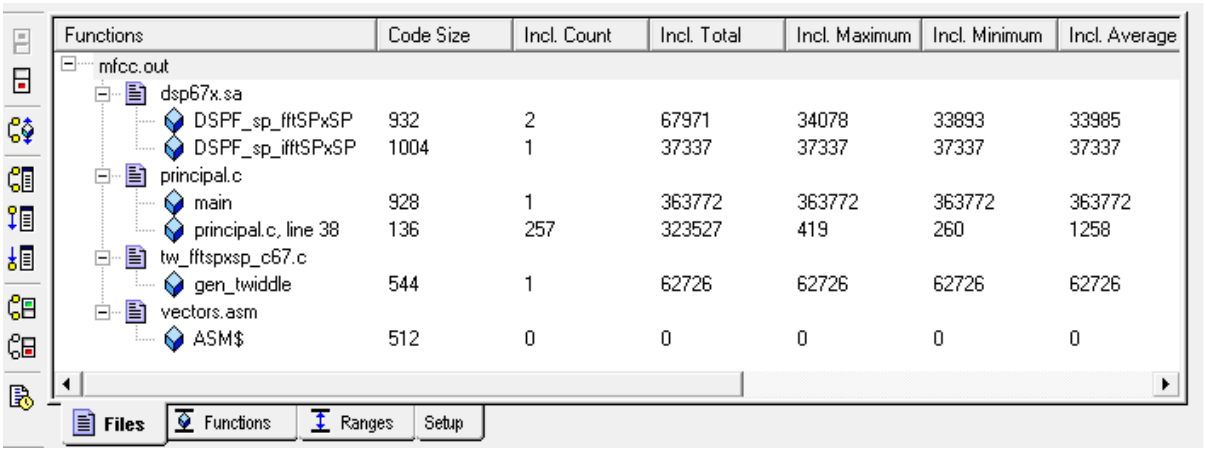
Files | Functions | Ranges | Setup

Figure.III.19 : Profileur sous fastrts67x.lib avec optimisation de l'équation ym mais pas le compilateur

Comme précédemment, l'expression de la ligne 40 profite du parallélisme du pipeline qui réduit le nombre de cycles jusqu'à 37 cycles.

Compilation avec optimisation et première forme d'expression ym[], figure.III.20.

On constate que le programme Principal.c (main) dans son entier réclame 363772 cycles, soit 2.425ms (temps réel sous 44kHz d'échantillonnage). L'expression ym[] réclame quant à elle, 1258 cycles ou 8.387us.

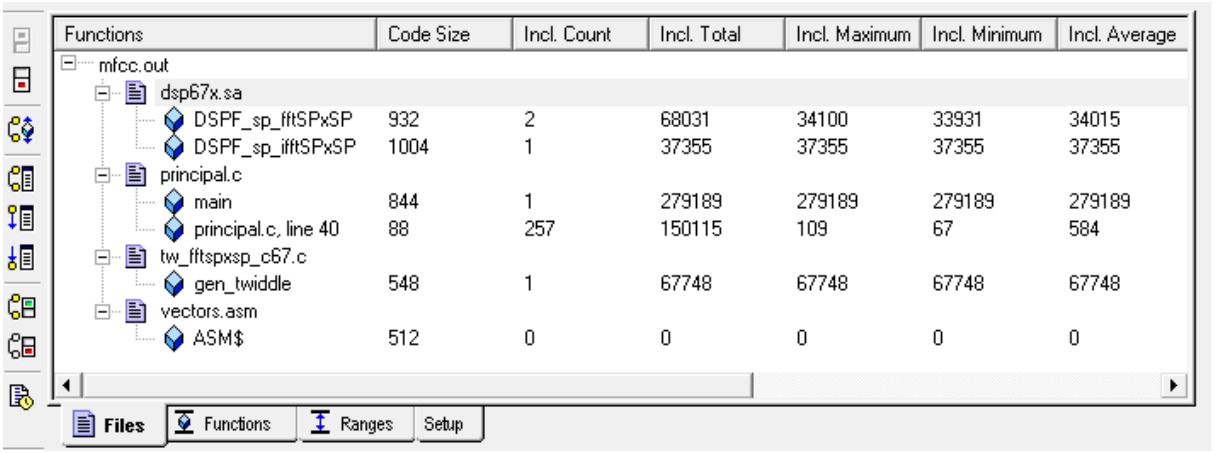


Functions	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
mfcc.out						
dsp67x.sa						
DSPF_sp_fftSPxSP	932	2	67971	34078	33893	33985
DSPF_sp_ifftSPxSP	1004	1	37337	37337	37337	37337
principal.c						
main	928	1	363772	363772	363772	363772
principal.c, line 38	136	257	323527	419	260	1258
tw_fftspxsp_c67.c						
gen_twiddle	544	1	62726	62726	62726	62726
vectors.asm						
ASM$	512	0	0	0	0	0

Files | Functions | Ranges | Setup

Figure.III.20 : Profileur sous fastrts67x.lib avec optimisation du compilateur mais l'équation ym

Compilation avec optimisation et deuxième forme d'expression ym[], figure.III.21.
On constate que le programme Principal.c (main) dans son entier réclame 279189 cycles, soit 1.861ms (temps réel à 44kHz, même pour segments réduits à 82 éch.). L'expression ym[] réclame quant à elle, 584 cycles ou 3.893us.



Functions	Code Size	Incl. Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average
mfcc.out						
dsp67x.sa						
DSPF_sp_fftSPxSP	932	2	68031	34100	33931	34015
DSPF_sp_ifftSPxSP	1004	1	37355	37355	37355	37355
principal.c						
main	844	1	279189	279189	279189	279189
principal.c, line 40	88	257	150115	109	67	584
tw_fftspxsp_c67.c						
gen_twiddle	548	1	67748	67748	67748	67748
vectors.asm						
ASM$	512	0	0	0	0	0

Files | Functions | Ranges | Setup

Figure.III.21 : Profileur sous fastrts67x.lib avec optimisation du compilateur et de l'équation ym

Comme précédemment, On peut dire que la compilation sans optimisation permet quand même d'optimiser le calcul de la variable ym par rapport à la compilation avec optimisation, mais celle-ci optimise d'autres calculs dans « main » qui la réduisent davantage en temps d'exécution.
Passant de la toute première combinaison d'options (compilateur sans optimisation avec première forme d'expression de ym) utilisant la bibliothèque standard rts6701.lib, vers la toute dernière combinaison d'options (compilateur avec optimisation et deuxième forme d'expression de ym) et utilisant la bibliothèque rapide fastrts67x.lib, la vitesse d'exécution se trouve améliorée de 3 fois et demie.

Une fois la mise au point du programme achevé, on peut éliminer les instructions d'affichage d'information appartenant à la bibliothèque <stdio.h> , telles que « puts() » et « printf() ».
On obtient alors pour la toute dernière combinaison d'options d'optimisation, figure.III.22, le profil statistique suivant :

Le programme Principal.c (main) dans son entier réclame 263087 cycles, soit 1.754ms (temps réel au standard maximal de 48 kHz et pouvant réduire les segments dans ce cas à 84 éch.). En définitif, la vitesse d'exécution globale se trouve améliorée de 3.8 fois par rapport au cas initial sans optimisation. L'expression ym[] réclame quant à elle, 567 cycles ou 3.78us.

On note cependant dans la figure.III.22, l'utilisation de l'option de menu « View\Mixed source/ASM » dans la fenêtre du programme principal « main ». Elle consiste à afficher après chaque ligne du code source, le code assembleur associé. L'optimisation effectuée du code montre l'utilisation intensive d'instructions en parallèle (pour le boucle « for » affichée, elle nécessite uniquement 2 exécutions séparées à chaque itération).

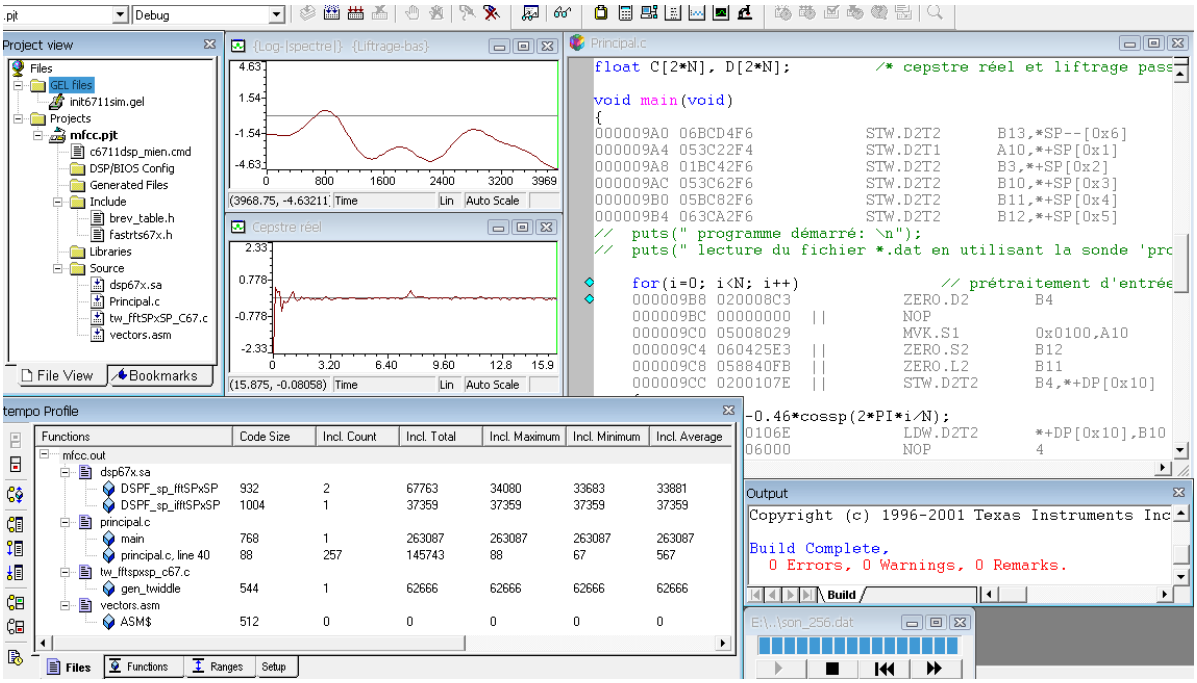


Figure.III.22 : Environnement CCS montrant le dernier profilage d'exécution

Conclusion

Dans ce chapitre, l'implémentation du projet a été faite en premier lieu sur MATLAB vu sa programmation aisée, mais aussi son utilité comme outil de validation des résultats obtenus à partir de l'implémentation finale du projet sur le DSP C6711, en langage C.

La constitution de notre projet sur DSP, est faite de deux fichiers en C (programme principal et routine de calcul des coefficients pour les transformées FFT et iFFT), d'un fichier en assembleur linéaire (pour accélérer le calcul des deux fonctions, FFT et iFFT), un fichier en assembleur d'initialisation du vecteur reset au point de départ du programme principal. Les résultats obtenus sont identiques aux résultats sur Matlab.

Nous avons fait une optimisation du programme. Ces changements d'optimisation ont abouti à une réduction du temps d'exécution à 1.75ms. Soit, plus rapide d'environ quatre fois de ce qu'il était.

Conclusion générale

L'implémentation de l'analyse homomorphique de la parole en langage C sur un DSP de type TMS320C6711 a été menée avec succès. Une vérification des résultats sur Matlab, a aussi était faite. Après prétraitement du segment choisi de la parole, le filtrage homomorphique consiste à effectuer une déconvolution entre l'excitation glottique et l'effet des cavités résonnantes du conduit vocal. Le cepstre réel résultant a été liftré en passe-bas pour l'obtention d'un spectre lissé montrant les formants du segment traité. Un liftrage en passe-haut isole par contre le spectre de l'excitation glottale.

Un banc de filtre mel le long de la bande acoustique de 4 kHz, a été utilisé afin de traiter l'énergie spectrale du signal, et permettre l'extraction de 22 coefficients cepstraux, MFCC. Ces coefficients sont une approximation du cepstre, et fournissent aussi un spectre lissé dit mel cepstral, compact sur 22 points. Le spectre mel cepstral et les coefficients MFCC forment une paire de la transformée en cosinus.

Les spectres lissés obtenus, par liftrage ou par filtrage mel, suivent bien l'allure du spectre du signal fourni par FFT et qui est affecté d'harmoniques du fondamental. On note cependant, que le spectre lissé par filtrage mel suit mieux le spectre FFT que celui lissé par liftrage du cepstre réel.

La constitution de notre projet sur DSP, est faite de deux fichiers en C (programme principal et routine de calcul de coefficients pour les transformées FFT et iFFT), d'un fichier en assembleur linéaire (pour accélérer les deux fonctions de calcul, FFT et iFFT), un fichier en assembleur (pour l'initialisation du vecteur reset au point de départ, c_int00, du programme principal), un à deux fichiers d'entête (dont l'un fournit la table d'inversion de bit requis par FFT et iFFT), et un fichier d'édition de lien (qui déclare une mémoire globale, SDRAM interne et externe de 16 Mo, en l'affectant aux différentes sections prédéfinies du compilateur C).

Une optimisation du programme de liftrage a été entamée par profilage, en comptant le nombre moyen de cycles d'horloge que consomme le programme entier, et une ligne particulière de code au sein d'une boucle du programme, dont l'expression a été réécrite afin de l'accélérer. La compilation du programme s'est faite de deux manières distinctes, sans et avec optimisation.

La bibliothèque standard au langage C a été remplacée par une autre plus rapide. Il s'agit d'une bibliothèque de fonctions mathématiques optimisées en virgule flottante.

Ces changements d'optimisation ont abouti à une réduction du temps d'exécution à 1.75ms. Soit, plus rapide d'environ quatre fois de ce qu'il était. Rendant ainsi, notre application exécutable en temps réel même sous un échantillonnage standard maximal de 48 kHz au lieu de 8 kHz de départ, et pouvant réduire la segmentation dans ce cas à 84 échantillons au lieu des 256 échantillons de départ.

Cette performance est due à la traditionnelle architecture VLIW, qui consiste en une multitude d'unités d'exécution opérant en parallèle, et traitant de multiples instructions durant un seul cycle d'horloge. Le parallélisme est la clé de l'extrême performance obtenue et rend les DSPs plus efficaces que les conceptions superscalaires traditionnelles [27].

L'architecture VelociTI de notre DSP cible (TMS320C6711) utilise une amélioration du traditionnel VLIW afin d'achever une haute performance à travers un parallélisme poussé au niveau d'instructions. Cette flexibilité architecturelle est la clé de l'efficacité du compilateur C6000 [27].

Références

[1] L. Rabiner et B. Juang, Fundamentals of Speech Recognition, Prentice-Hall, 1993.

[2] R. Boite et M. Kunt, Traitement de la Parole, Press Polytechniques Romandes, 1987.

[3] Calliope, Traitement automatique de la parole, Masson, 1989.

[4] M. George, P. Jospa, et A. Soquet, Articulatory trajectories generated by the control of the vocal tract by a neural network. In Proceedings of the International Conference on Spoken Language Processing, 1994.

[5] J. Koreman, B. Andreeva, et H. Strik, Acoustic parameters versus phonetic features in ASR, In Proceedings of the International Congress of Phonetic Sciences, 1999.

[6] R. Boite et M. Kunt, Traitement de la parole, Presses Polytechniques Romandes, Lausanne, l 987.

[7] A. V. Oppenheim, "Superposition in a Class of Nonlinear Systems," Tech. Report No. 432, Research Lab. of Electronics, MIT, Cambridge, March 1965.

[8] R. W. Schafer, "Echo Removal by Discrete Generalized Linear Filtering," Technical Report 466, Research Lab of Electronics, MIT, February 1969.

[9] B. Bogert, M. Healy, and J. Tukey, "The Quefrency Alanysis of Time Series for Echoes," Proc. Symp. on Time Series Analysis, M. Rosenblatt, Ed., Ch. 15, PP. 209-243, J. Wiley, New York, 1963.

[10] H. Amini et M. Dia, "Méthodes Cepstrales," Ecole Polytechnique, Enseignements d'Approfondissement, Traitement du signal, Palaiseau, 2005.

[11] J. H. Y. Loo, "Intraframe and interframe coding of speech spectral parameters", Departement of Electrical & Computer Engineering, McGill University, Montreal, Canada, Sep 1996.

[12] B; S. Atal, R. V. Cox, and P.Kroon, "Spectral quantization and interpolation for CELP coders," in Proc. IEEE Int. Conf. on Acoustics, Speech, and Sig. Processing, (Glasgow UK), May. 1989, pp. 69-72.

[13] W. Pereira,"Modifying LPC Parameter Dynamics to Improve Speech Coder Efficiency", Master Thesis, McGill University, Montreal, canada, Septembre 2001.

[14] B. Bouchez, Applications Audio Numériques des DSP, Publitronic, Paris, 4 déc. 2008.

[15] G. Baudoin et F. Virolieu, les DSP famille TMS320 C54x : Développement d'applications, Dunod, Paris, 2000.

[16] O. Sentieys, Processeur de Traitement Numérique de Signal (DSP), IRISA/ENSSAT Lannion, Université de Rennes I.

[17] Texas Instruments, TMS320 C54x DSP Reference Set, Volume1: CPU and Peripherals, (SPRU131), 2001.

[18] "Guide lins for software development efficiency on the TMS 320 C6000 velocity architects ", spra h, 1998.

[19] A. Drygajlo, « Traitement de la parole », Swiss federal institute of technology, Lausane (EPFL), 2011.

[20] L. R. Rabiner, R. W. Schafer, « Introduction to Digital Speech Processing », University of California, 2007.

[21] X. Huang, A. Acero, H.W. Hon, "*Spoken Language Processing - A Guide to Theory, Algorithm, and System Development"*, Prentice Hall, 2001.

[22] J. Deller, et. al., "*Discrete-Time Processing of Speech Signals"*, MacMillan Publishing, 2000.

[23] TI, « Code composer user's guide », spru296.pdf, 1999.

[24] TI, « Tutoriel du CCS », simulateur v2.0.

[25] TI, "TMS320C6000 optimizing compiler user's guide", SPRU187i, 2001.

[26] TI, "TMS320C67x fastRTS library programmer's reference", SPRU100A, 2002.

[27] TI, "TMS320C6000 Technical Brief", SPRU197D, 1999.

Annexe A

Listings des programmes sur MATLAB :

1-Programme A_Liftrage.m

```
clc; clear;
[y, fs, nbits] = wavread('E:\Slimane\A8k16M.wav');  % fs=8000Hz, nbits=16bits,
     (enregistrement sous format PCM, avec attributs : 8KHz, 16bits, mono)
sound(y,fs,nbits);
subplot(211); plot((1:length(y))/8,y)
xlabel('temps (ms)'); title('voyelle A');
duree = 256;                % fenêtre de Haming limitée à 2^8=256 échantillons
        (durée = 32ms),pour 2 périodes du pitch minimal (pitch=62.5Hz)
e=y(3088:3343).*hamming(duree);    % 256 échantillons de la zone utile
subplot(212); plot((1:length(e))/8,e);  % tronçon utile mesuré du son
xlabel('temps (ms)'); title('voyelle A - Hamming');
 save('son_256.dat', 'e', '-ascii');     % sauvegarde du tronçon fenêtré utile
                                    en mode ASCII
pause(3)                                % voir "A_son.bmp"


 C = log(abs(fft(e)));                   % log du module du spectre, C=exp(C)
                                pour module du spectre
D = ifft(C);                            % cepstre réel (partie réelle paire du
                                  cepstre complexe)
Quefrence = length(D)/2;                % 128 échantillons
for i=1:Quefrence
    if D(i)<-1
  DD(i)=-0.5;                    % bornage de D à +1 & -0.5, pour affichage
                                  optimal de D
    elseif D(i)>1
        DD(i)=1;
    else
        DD(i)=D(i);
    end
end
lwin = 18; % largeur relative optimale de la fenêtre rectangulaire de liftrage
win1 = [ones(1,lwin)'; zeros(1,size(D,1)-2*lwin)'; ones(1,lwin)'];
win2 = [zeros(1,lwin)'; ones(1,size(D,1)-2*lwin)'; zeros(1,lwin)'];
```

```
 E1 = D.*win1; E2 = D.*win2;                          % liftrage passe-bas E1 pour
                    formants et passe-haut E2 pour excitation
F1 = real(fft(E1)); F2 = real(fft(E2));      % spectres filtrés correspondants

subplot(221); plot((1:Quefrence)/8,DD,'b'); hold on;
plot((1:Quefrence)/8,win1(1:Quefrence),'r');
xlabel('Quefrence (ms)'); title('Cepstre - win1');
subplot(222); plot((1:length(DD))/8,DD,'b'); hold on;
plot((1:Quefrence)/8,win2(1:Quefrence),'r');
xlabel('Quefrence (ms)'); title('Cepstre - win2');
subplot(223); plot((1:length(C)/2)*8/320,C(1:length(C)/2)); hold on;
plot((1:length(F1)/2)*8/320,F1(1:length(F1)/2),'r');
xlabel('Frequence (kHz)'); title('Liftrage des formants');
subplot(224); plot((1:length(C)/2)*8/320,C(1:length(C)/2)); hold on;
plot((1:length(F2)/2)*8/320,F2(1:length(F2)/2),'r');
xlabel('Frequence (kHz)'); title('Liftrage de l''excitation');
pause(3)                    % voir "A_liftrage.bmp" avec pitch mesuré de 129 Hz
```

2-Programme D_MFCC.m

```
clc; clear;
 segment = 256;  fs=8000;              % segment en échantillons et fréquence
                              d'échantillonnage
[x,Filtres] = banc(segment,fs);        % x non utilisé
plot((0:segment-1)/segment*fs,Filtres'); title('Banc de filtres "MEL-MFCC"');
axis([0 4000 0 max(max(Filtres))])                  % [XMIN=0 XMAX=4kHz YMIN=0
                            YMAX=max_PicDeFiltres]
```

3-Fonction banc.m

```
% Fonction banc.m
function [Freqs,PoidsFiltres] = banc(duree,Fech)
 % paramètres du banc de filtres MEL-MFCC :
FreqBasse = 0; FiltresLIN = 10; EspaceLIN = 100; FiltresLOG = 12; EspaceLOG =
1.11253;
FiltresTotal = FiltresLIN + FiltresLOG;  % banc de 22 mel-filtres
```

```
% Localisation de toutes les bandes de fréquence: les fréquences sont espacées
linéairement, puis  logarithmiquement. Les limites de bande (bas, centre,
haut) sont consécutives pour tous les filtres.
Freqs = FreqBasse + (0:FiltresLIN-1)*EspaceLIN;     % freqs avec 1 ligne et 10
                                    colonnes
Freqs(FiltresLIN+1:FiltresTotal+2) = ...            % ajout de 14 colonnes, de
                            la 11ème à la 24ème colonne
      Freqs(FiltresLIN) * EspaceLOG.^(1:FiltresLOG+2);
bas = Freqs(1:FiltresTotal); centre = Freqs(2:FiltresTotal+1); haut =
Freqs(3:FiltresTotal+2);
TailleTriangle = 1./(haut-bas);                        % normalisation (<1) du
                              poids de chaque filtre
fftFreqs = (0:duree-1)/duree*Fech;
PoidsFiltres = zeros(FiltresTotal,duree);           % initialisation
for num=1:FiltresTotal           % réalisation de chaque gabarit trianglulaire
    PoidsFiltres(num,:) = ...
  (fftFreqs > bas(num) & fftFreqs <= centre(num)).* ...     % pente ascendante
 TailleTriangle(num).*(fftFreqs-bas(num))/(centre(num)-bas(num))+ ...        %
                              pondération des sommets
  (fftFreqs > centre(num) & fftFreqs < haut(num)).* ...    % pente descendante
   TailleTriangle(num).*(haut(num)-fftFreqs)/(haut(num)-centre(num));     %
                             pondération des minimums
end
```

4-Programme E_Spectres_MelMfcc.m)

```
% comparaison des spectres reconstruits, entre autres, à partir de notre
fonction "MEL-MFCC"
clc; clear; voiceboxpath='E:\Slimane\voicebox'; addpath(voiceboxpath);
 [y, fs, nbits] = wavread('E:\Slimane\A8k16M.wav');  % fs=8kHz, nbits=16bits,
                                    monophonique
segment = 256; e=y(3088:3087+segment);                      % fenêtrage à 320
                    échantillons de la parole (durée 40ms)
preAccentuation = filter([1 -.97], 1, e);
Data = preAccentuation.*hamming(segment);

%% spectre basé sur FFT et sur LPC
TF=(abs(fft(Data,segment)));                            % spectre par FFT
F= [1:31:3968]';
```

```
plot(F,20*log10(TF(1:segment/2)),'r'); hold on;          % mettre hold-on sous
                  commentaire pour isoler la trace du melcep

%% spectre basé sur cepstre liftré
AmpFFT = abs(fft(Data));
C = log(AmpFFT);                      % log de l'amplitude du spectre
D = real(ifft(C));                      % cepstre réel
D = D(1:15);                             % se limiter à 15 (donc liftrage passe-
                               bas pour formants)
cep = real(fft(D',floor(segment)));             % log-spectre avec cep allongé
             jusqu'à 'segment' par des zéros (suite du liftrage)
cep = exp(cep);                           % spectre
plot(F,20*log10(cep(1:segment/2)),'g'); % spectre en db

%% spectre basé sur mel-cepstre
AmpFFT = AmpFFT.^2;                    % énergie du segment sur la durée 32ms
[Freqs,Filtres] = banc(segment,fs);     % (calcul des poids du filtre mel) :

Energie = 0;
for num=1:segment;                              % calcul d'énergie du segment
    Energie = AmpFFT(num) + Energie;
end

spectre=10*log10(Energie * Filtres * AmpFFT );             % spectre mel lissé
                        (pondéré par l'énergie), en dB
MFCC = rdct(spectre);
FF= [1:0.1:length(MFCC)]';           % 210 points de résolution fréquentielle
MFCCInterpolee = spline(1:length(MFCC),MFCC,FF);              % interpole MFCC
                           suivant une résolution FF
figure(2); plot(1:length(MFCC),MFCC,'o',FF,MFCCInterpolee,'k');
title('coeff. MFCC');
% voir "E_MFCCmelmfcc.bmp"

% reconstruction du spectre lissé mel cepstral
melcep = irdct(MFCC);                              % melcep = spectre mel lissé
figure(1); plot(Freqs(2:23),melcep,'*k'); hold on;
```

```
MelCepInterpolee = spline(Freqs(2:23),melcep,F);        % interpole melcep aux
                   valeurs 'centre' suivant une résolution F
plot(F,MelCepInterpolee,'k');
xlabel('Fréquence (Hz)'); ylabel('Amplitude (dB)')
title('|FFT|-en-dB (rouge), cepstre-liftré (vert), MFCC (noir)');           %
voir "E_MFCCmelmfcc.bmp"
%title('spectre lissé mel cepstral (melcep)');                % pour melcep
isolé (voir commentaire du hold-on du haut)
```

Annexe B

Listings des programmes sur DSP :

1. Projet 2_Liftrage

a. Principal.c

```
#include <stdio.h>
#include <math.h>
#include "brev_table.h"
unsigned int i, j;                    /* compteurs de boucle */
 const int N=256;                      /* nombre d'éch. dans un segment traité
             (durée=32ms pour 2 périodes du pitch_min=62.5Hz)  */
 const int base=4, offset=0;              /* base=4 si N=x^4, base=2 si N=x^2,
                     et décalage d'indice d'éch. en FFT */
 const int bande=18;            /* largeur de bande pour liftrage passe-bas et
                                  passe-haut */
 float ech[256], ham[256], x[512];          /* éch. réels d'entrée, fenêtre de
                      Hamming, éch. complexes d'entrée */
 float w[512], y[512], ym[256];        /* facteurs complexes pour FFT, spectre
                           de sortie et son module */
float C[512], D[512];               /* liftrages (passe-bas et passe-haut) */

void main(void)
{
  puts(" programme démarré: \n");
  puts(" lecture du fichier *.dat en utilisant la sonde 'probe point': \n");
                  for(i=0; i<N; i++)             // prétraitement d'entrée
      {     ham[i]=0.54-0.46*cos(6.28*i/N);           // fenêtre de Hamming
            ech[i]*=ham[i];
            x[2 * i] = ech[i];                      // éch. complexes d'entrée
            x[2 * i + 1] = 0;
                  };
      gen_twiddle(w, N);
// Génère les facteurs w (twiddle) pour le calcul FFT
 DSPF_sp_fftSPxSP(N, x, w, y, brev, base, offset, N);     // calcule en asm la
                              FFT complexe y de x
      for(i=0; i<N; i++)                            // traitement du spectre
  {     ym[i] = sqrt(powf(y[2*i],2)+powf(y[2*i+1],2));              // module
                                  du spectre
            C[2*i] = log(ym[i]);           // log du module du spectre
```

```
            C[2*i+1] = 0;
      };
/*  la sortie IFFT en D est normalisée par division par N */
 DSPF_sp_ifftSPxSP(N, C, w, D, brev, base, offset, N);   // calcule en asm la
                                iFFT complexe de C
      for(i=0; i<2*N; i++)
  {     if (i>bande && i<2*N-bande) x[i]=0;          // liftrage passe-bas en
                                quefrence (temps)
            else x[i]=D[i];
      };
   DSPF_sp_fftSPxSP(N, x, w, C, brev, base, offset, N);    // liftrage C des
                                formants (fréquence)
      for(i=0; i<2*N; i++)
 {     if (i>bande && i<2*N-bande) x[i]=D[i];       // liftrage passe-haut en
                                quefrence (temps)
            else x[i]=0;
      };
    DSPF_sp_fftSPxSP(N, x, w, C, brev, base, offset, N);  // liftrage C de
                              l'excitation (fréquence)
      printf(" Programme terminé !\n");
}
```

b. tw_fftSPxSP_C67.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#ifndef PI
# ifdef M_PI
# define PI M_PI
# else
# define PI 3.14159265358979323846
# endif
#endif

int gen_twiddle(float *w, int n)
{   int i, j, k;
    for (j = 1, k = 0; j < n >> 2; j = j << 2)
    {   for (i = 0; i < n >> 2; i += j)
        {  w[k +  5] =  sin(6.0 * PI * i / n);
           w[k +  4] =  cos(6.0 * PI * i / n);
           w[k +  3] =  sin(4.0 * PI * i / n);
           w[k +  2] =  cos(4.0 * PI * i / n);
           w[k +  1] =  sin(2.0 * PI * i / n);
           w[k +  0] =  cos(2.0 * PI * i / n);
           k += 6;
        }
    }
```

```
    return k;
}
```

c. dsp67x.sa

```
;******************************************************************************
;* TEXAS INSTRUMENTS, INC.
;* NAME DSPF_sp_fftSPxSP -- Single Precision floating point mixed radix
;*      forwards FFT with complex input
;*
;* REVISION DATE     24-Sep-2003
;*
;* USAGE
;*       This routine is C-callable and can be called as:
;*       void DSPF_sp_fftSPxSP
*       (   int N,
*
*           float * ptr_x,
*
*           float * ptr_w,
*
*           float * ptr_y,
*
*           unsigned char * brev,
*
*           int n_min,
*
*           int offset,
*
*           int n_max
*       );

*       N      :  Length of fft in complex samples, power of 2 such that
*
*                    N >=8 and N<= 16385.
*
*       ptr_x  :  Pointer to complex data input.
*
*       ptr_w  :  Pointer to complex twiddle factor.
*
*       ptr_y  :  Pointer to complex output data.
*
*       brev   :  Pointer to bit reverse table containing 64 entries.
*
*       n_min  :  Smallest fft butterfly used in computation used for
*
*                    decomposing fft into subffts, see notes.
*
*       offset :  Index in complex samples of sub-fft from start of main fft.
*
*       n_max  :  size of main fft in complex samples.
* ============================================================================
        .text
        .global _DSPF_sp_fftSPxSP
_DSPF_sp_fftSPxSP   .cproc  N, ptr_x, ptr_w, ptr_y, brev, n_min, offset,
n_max
 ============================================================================

                    .reg    j, h2, l1, l2, predj, w_copy
                    .reg    Ltmp, ftmp1, ftmp2, ftmp3, ftmp4, ptr_x2, x, w
                    .reg    prod0, prod1, prod2, prod3
```

```
                    .reg    prod4, prod5, prod6, prod7
                    .reg    x_1:x_0, x_l1p1:x_l1, x_h2p1:x_h2, x_l2p1:x_l2
                    .reg    si1:co1, si2:co2, si3:co3
                    .reg    xh0, xh1, xl0, xl1, xh20, xh21, xl20, xl21
                    .reg    xt0, xt1, xt2, yt0, yt1, yt2, h2p1, l1p1, l2p1
                    .reg    h2_ind, l1_ind, l2_ind, ptr_x_h2, ptr_x_l2, p_x
                    .reg    whileFlag, radix, n_copy, x_copy, st_index
                    .reg    fft_jmp, tw_offset, stride, ptr_x_copy
                    .reg    k0, k1, y1, temp, j0, j1, ptr_x0, y0, l0, gtflag
                    .reg    x1:x0, x3:x2, x5:x4, x7:x6
                    .reg    xh0_0, xh1_0, xh0_1, xh1_1
                    .reg    xl0_0, xl1_0, xl0_1, xl1_1
                    .reg    yt3, yt4, yt5, yt6, yt7, eqflag, nmax, flag
                    .no_mdep

                    MV        ptr_x,          ptr_x_copy
                    MV        N,                stride
                    ;radix = n_min
                    MV        n_min,          radix
                    ZERO    tw_offset
whileLoop:    ZERO     j
                    ;fft_jmp = stride + stride>>1
                    SHRU    stride,         1,          Ltmp
                    ADD     stride,         Ltmp,       fft_jmp
                    ;Have Copies of ptr_x for load and store
                    MV      ptr_x,          x
                    MV      ptr_x_copy,     ptr_x2
                    ;Twiddle factor array accessed using 'tw_offset' index
                    ADDAW   ptr_w,          tw_offset,  w
                    MV      w,              w_copy
                    ;h2, l1, l2 are at 'Ltmp = srtide>>1' apart
                    MV        Ltmp,           st_index
                    ;Offset to be used with LDDW
                    SHR     Ltmp,           1,          Ltmp
                    MV        N,              n_copy
iLoop:              .trip   2
                    ;Load twiddle facotrs
                    ADDAW   w_copy,         j,          w
                    LDDW    *w,             si1:co1
                    LDDW    *+w[1],         si2:co2
                    LDDW    *+w[2],         si3:co3
                    ;Load input complex data, which are 'Ltmp' apart
                    MV        x,              x_copy
                    LDDW    *x_copy++[Ltmp],x_1:x_0
                    LDDW    *x_copy++[Ltmp],x_h2p1:x_h2
                    LDDW    *x_copy++[Ltmp],x_l1p1:x_l1
                    LDDW    *x_copy++[Ltmp],x_l2p1:x_l2
                    ADDSP   x_0,            x_l1,       xh0
                    ADDSP   x_1,            x_l1p1,     xh1
                    SUBSP   x_0,            x_l1,       xl0
                    SUBSP   x_1,            x_l1p1,     xl1
                    ADDSP   x_h2,           x_l2,       xh20
                    ADDSP   x_h2p1,         x_l2p1,     xh21
                    SUBSP   x_h2,           x_l2,       xl20
                    SUBSP   x_h2p1,         x_l2p1,     xl21
                    ADDSP   xh0,            xh20,       ftmp1
                    ADDSP   xh1,            xh21,       ftmp2
                    ; Store the values from butterfly calculation
                    MV        ptr_x2,           x2_copy
                    STW     ftmp2,          *+x2_copy[1]
                    STW     ftmp1,          *x2_copy++[st_index]
```

```
                ADD     x,              8,          x
                ADD     j,              6,          j
                SUB     j,              fft_jmp,    predj
        [!predj]    ADDAW   x,              fft_jmp,    x
        [!predj]    ZERO    j
                SUBSP   xh0,            xh20,       xt0
                SUBSP   xh1,            xh21,       yt0
                ADDSP   xl0,            xl21,       xt1
                ADDSP   xl1,            xl20,       yt2
                SUBSP   xl0,            xl21,       xt2
                SUBSP   xl1,            xl20,       yt1
                MPYSP   xt1,            co1,        prod0
                MPYSP   yt1,            si1,        prod1
                MPYSP   yt1,            co1,        prod2
                MPYSP   xt1,            si1,        prod3
                MPYSP   xt0,            co2,        prod4
                MPYSP   yt0,            si2,        prod5
                MPYSP   yt0,            co2,        prod6
                MPYSP   xt0,            si2,        prod7
                ADDSP   prod0,          prod1,      ftmp1
                SUBSP   prod2,          prod3,      ftmp2
                ADDSP   prod4,          prod5,      ftmp3
                SUBSP   prod6,          prod7,      ftmp4
                ;Store the values from butterfly calculation
                STW     ftmp4,          *+x2_copy[1]
                STW     ftmp3,          *x2_copy++[st_index]
                STW     ftmp2,          *+x2_copy[1]
                STW     ftmp1,          *x2_copy++[st_index]
                MPYSP   xt2,            co3,        prod0
                MPYSP   yt2,            si3,        prod1
                MPYSP   yt2,            co3,        prod2
                MPYSP   xt2,            si3,        prod3
                ADDSP   prod0,          prod1,      ftmp1
                SUBSP   prod2,          prod3,      ftmp2
                ;Store the values from butterfly calculation
                STW     ftmp2,          *+x2_copy[1]
                STW     ftmp1,          *x2_copy++[st_index]
                ADD     ptr_x2,         8,          ptr_x2
        [!predj]    ADDAW   ptr_x2,         fft_jmp,    ptr_x2
                SUB     n_copy,         4,          n_copy
        [n_copy]    B       iLoop
                ADD     tw_offset,      fft_jmp,    tw_offset
                SHR     stride,         2,          stride
                CMPGT   stride,         radix,      whileFlag
        [whileFlag] B       whileLoop
;--------------------End of While Loop-----------------------------------------

                SHR     offset,         2,          j
                MV      ptr_x,          ptr_x0
                MV      ptr_y,          y0
                ;get size of fft -> l0 = _norm(n_max) - 17
                NORM    n_max,          l0
                SUB     l0,             17,         l0
                ;find if radix is 2 or 4 and set the flag
                CMPEQ   n_min,          2,          eqflag
        [eqflag]    MVK     1,              flag
        [!eqflag]   ZERO    flag
                MV      n_max,          nmax
                CMPGT   n_min,          4,          gtflag
                ;Jump to end of Function if radix is neither 2 nor 4
        [gtflag]    B       retFunc
```

```
                    SHR     nmax,           1,          nmax
                    SUB     nmax,           1,          nmax
                    ;reversal computation
top:                .trip 2
                    ;j0 = j & 0x3F
                    EXTU    j,              26,         26,   j0
                    SHR     j,              6,          j1
                    ;Load bit reverse array using j0 and j1 as index
                    LDBU    *brev[j0],      k0
                    LDBU    *brev[j1],      k1
                    SHL     k0,             6,          k0
                    ADD     k0,             k1,         k0
                    SHRU    k0,             l0,         k0
                    ADDAW   y0,             k0,         y1
                    ADDK    1,              j
                    LDDW    *ptr_x0++,      x1:x0
                    LDDW    *ptr_x0++,      x3:x2
                    LDDW    *ptr_x0++,      x5:x4
                    LDDW    *ptr_x0++,      x7:x6
                    ADDSP   x0,x4,          xh0_0
                    ADDSP   x1,x5,          xh1_0
                    ADDSP   x2,x6,          xh0_1
                    ADDSP   x3,x7,          xh1_1
          [flag]    MV      x0,             xh0_0
          [flag]    MV      x1,             xh1_0
          [flag]    MV      x2,             xh0_1
          [flag]    MV      x3,             xh1_1
                    ADDSP   xh0_0,          xh0_1,      yt0
                    SUBSP   xh0_0,          xh0_1,      yt4
                    SUBSP   xh1_0,          xh1_1,      yt5
                    ADDSP   xh1_0,          xh1_1,      yt1
                    SUBSP   x0,x4,          xl0_0
                    SUBSP   x1,x5,          xl1_0
                    SUBSP   x2,x6,          xl0_1
                    SUBSP   x3,x7,          xl1_1
          [flag]    MV      x4,             xl0_0
          [flag]    MV      x5,             xl1_0
          [flag]    MV      x6,             xl1_1
          [flag]    MV      x7,             xl0_1
                    ADDSP   xl0_0,          xl1_1,      yt2
                    SUBSP   xl1_0,          xl0_1,      yt3
                    SUBSP   xl0_0,          xl1_1,      yt6
                    ADDSP   xl1_0,          xl0_1,      yt7
          [flag]    MV      yt3,            temp
          [flag]    MV      yt7,            yt3
          [flag]    MV      temp,           yt7
                   ;Store appropriate values at appropriate offsets in output
                    STW     yt0,            *y1++
                    STW     yt1,            *y1++[nmax]
                    STW     yt2,            *y1++
                    STW     yt3,            *y1++[nmax]
                    STW     yt4,            *y1++
                    STW     yt5,            *y1++[nmax]
                    STW     yt6,            *y1++
                    STW     yt7,            *y1
                    SUB   N,   4, N
            [N]     B       top
retFunc:
                    .return
                .endproc
 End of file: DSPF_sp_fftSPxSP.sa
```

```
 ============================================================================
 NAME DSPF_sp_ifftSPxSP -- Single Precision floating point mixed radix
       inverse FFT with complex input

 REVISION DATE    21-Oct-2003

   USAGE

       This routine is C-callable and can be called as:
       void DSPF_sp_ifftSPxSP
*       (   int N,
*
*           float * ptr_x,
*
*           float * ptr_w,
*
*           float * ptr_y,
*
*           unsigned char * brev,
*
*           int n_min,
*
*           int offset,
*
*           int n_max
*
*       );
*
*   same arguments and comments as above.
*
* ============================================================================
        .text
        .global _DSPF_sp_ifftSPxSP
_DSPF_sp_ifftSPxSP   .cproc  N, ptr_x, ptr_w, ptr_y, brev, n_min, offset,
n_max
* ============================================================================
               .reg    j, h2, l1, l2, predj, w_copy
               .reg    Ltmp, ftmp1, ftmp2, ftmp3, ftmp4, ptr_x2, x, w
               .reg    prod0, prod1, prod2, prod3, prod4, prod5, prod6, prod7
               .reg    x_1:x_0, x_l1p1:x_l1, x_h2p1:x_h2, x_l2p1:x_l2
               .reg    si1:co1, si2:co2, si3:co3
               .reg    xh0, xh1, xl0, xl1, xh20, xh21, xl20, xl21
               .reg    xt0, xt1, xt2, yt0, yt1, yt2, h2p1, l1p1, l2p1
               .reg    h2_ind, l1_ind, l2_ind, ptr_x_h2, ptr_x_l2, p_x
               .reg    whileFlag, radix, n_copy, x_copy, st_index, x2_copy
               .reg    fft_jmp, tw_offset, stride, ptr_x_copy
               .reg    k0, k1, y1, temp, j0, j1, ptr_x0, y0, l0, gtflag
               .reg    x1:x0, x3:x2, x5:x4, x7:x6
               .reg    xh0_0, xh1_0, xh0_1, xh1_1
               .reg    xl0_0, xl1_0, xl0_1, xl1_1
               .reg    yt3, yt4, yt5, yt6, yt7, eqflag, nmax, flag
               .reg    scale
               .no_mdep
                   MV        ptr_x,      ptr_x_copy
                   MV        N,            stride
                   ;radix = n_min
                   MV        n_min,      radix
                ZERO    tw_offset

whileLoop2:
```

```
                    ZERO    j
                    ;fft_jmp = stride + stride>>1
                    SHRU    stride,     1,            Ltmp
                    ADD       stride,     Ltmp,        fft_jmp
                    ;Have Copies of ptr_x for load and store
                    MV        ptr_x,      x
                    MV        ptr_x_copy,    ptr_x2
                    ;Twiddle factor array accessed using 'tw_offset' index
                    ADDAW    ptr_w,      tw_offset,  w
                    MV        w,           w_copy
                    ;h2, l1, l2 are at 'Ltmp = srtide>>1' apart
                    MV        Ltmp,        st_index
                    ;Offset to be used with LDDW
                    SHR        Ltmp,        1,        Ltmp
                    MV        N,           n_copy
iLoop2:              .trip    2
                    ;Load twiddle facotrs
                    ADDAW         w_copy,              j,      w
                    LDDW .2      *w,                  si1:co1
                    LDDW .2      *+w[1],              si2:co2
                    LDDW .D2T1   *+w[2],              si3:co3
                    ;Load input complex data, which are 'Ltmp' apart
                    MV          x,                     x_copy
                    LDDW .1   *x_copy++[Ltmp],        x_1:x_0
                    LDDW .1   *x_copy++[Ltmp],        x_h2p1:x_h2
                    LDDW .1   *x_copy++[Ltmp],        x_l1p1:x_l1
                    LDDW .1   *x_copy++[Ltmp],        x_l2p1:x_l2
                    ADDSP .1  x_0,        x_l1,       xh0
                    ADDSP .1  x_1,        x_l1p1,     xh1
                    SUBSP .1  x_0,        x_l1,       xl0
                    SUBSP .1  x_1,        x_l1p1,     xl1
                    ADDSP      x_h2,       x_l2,       xh20
                    ADDSP .2  x_h2p1,      x_l2p1,     xh21
                    SUBSP .2  x_h2,       x_l2,       xl20
                    SUBSP .2  x_h2p1,      x_l2p1,     xl21
                    ADDSP      xh0,        xh20,       ftmp1
                    ADDSP .1  xh1,        xh21,       ftmp2
                    ;Store the values from butterfly calculation
                    MV          ptr_x2,       x2_copy
                    STW          ftmp2,        *+x2_copy[1]
                    STW          ftmp1,        *x2_copy++[st_index]
                    ADD          x,          8,          x
                    ADD          j,          6,          j
                    SUB          j,          fft_jmp,    predj
   [!predj]     ADDAW     x,          fft_jmp,    x
   [!predj]     ZERO      j
                SUBSP .1  xh0,        xh20,       xt0
                SUBSP .2  xh1,        xh21,       yt0
                SUBSP .1  xl0,        xl21,       xt1
                SUBSP .2  xl1,        xl20,       yt2
                ADDSP .2  xl0,        xl21,       xt2
                ADDSP .1  xl1,        xl20,       yt1
                MPYSP .1  xt1,        co1,        prod0
                MPYSP .1  yt1,        si1,        prod1
                MPYSP .1  yt1,        co1,        prod2
                MPYSP .1  xt1,        si1,        prod3
                MPYSP .2  xt0,        co2,        prod4
                MPYSP .2  yt0,        si2,        prod5
                MPYSP .2  yt0,        co2,        prod6
                MPYSP .1  xt0,        si2,        prod7
                SUBSP .1  prod0,      prod1,      ftmp1
```

```
                ADDSP .1  prod2,      prod3,      ftmp2
                SUBSP .2  prod4,      prod5,      ftmp3
                ADDSP .1  prod6,      prod7,      ftmp4
                ;Store the values from butterfly calculation
                STW          ftmp4,        *+x2_copy[1]
                STW          ftmp3,        *x2_copy++[st_index]
                STW          ftmp2,        *+x2_copy[1]
                STW          ftmp1,        *x2_copy++[st_index]
                MPYSP      xt2,          co3,          prod0
                MPYSP      yt2,          si3,          prod1
                MPYSP      yt2,          co3,          prod2
                MPYSP .1  xt2,        si3,        prod3
                SUBSP      prod0,        prod1,      ftmp1
                ADDSP      prod2,        prod3,      ftmp2
                ;Store the values from butterfly calculation
                STW          ftmp2,        *+x2_copy[1]
                STW          ftmp1,        *x2_copy++[st_index]
                ADD          ptr_x2,       8,          ptr_x2
   [!predj]     ADDAW      ptr_x2,       fft_jmp,    ptr_x2
                SUB          n_copy,       4,          n_copy
   [n_copy]     B          iLoop2
                ADD          tw_offset,  fft_jmp,    tw_offset
                SHR          stride,       2,          stride
                CMPGT      stride,       radix,      whileFlag
   [whileFlag]  B          whileLoop2
;--------------------End of While Loop-------------------------------------

                SHR          offset,       2,          j
                MV          ptr_x,        ptr_x0
                MV          ptr_y,        y0
                ;get size of fft -> l0 = _norm(n_max) - 17
                NORM       n_max,        l0
                SUB          l0,           17,         l0
                ;find if radix is 2 or 4 and set the flag
                CMPEQ       n_min,        2,          eqflag
   [eqflag]     MVK        1,          flag
   [!eqflag]    ZERO       flag
                MV         n_max,      nmax
                CMPGT      n_min,      4,          gtflag
                ;Jump to end of Function if radix is neither 2 nor 4
   [gtflag]     B          retFunc2
                INTSP      nmax,       scale
                RCPSP      scale,      scale
                SHR          nmax,          1,          nmax
                SUB          nmax,          1,          nmax
                 ;reversal computation
top2:           .trip 2
                ;j0 = j & 0x3F
                EXTU .2   j,          26,          26,    j0
                SHR  .2   j,          6,          j1
                ;Load bit reverse array using j0 and j1 as index
                LDBU       *brev[j0],  k0
                LDBU       *brev[j1],  k1
                SHL          k0,           6,          k0
                ADD          k0,           k1,          k0
                SHRU       k0,          l0,          k0
                ADDAW       y0,          k0,          y1
                ADDK       1,          j
                LDDW       *ptr_x0++,  x1:x0
                LDDW       *ptr_x0++,  x3:x2
                LDDW       *ptr_x0++,  x5:x4
```

```
                LDDW      *ptr_x0++,  x7:x6
                ADDSP      x0,         x4,            xh0_0
                ADDSP      x1,         x5,            xh1_0
                ADDSP      x2,         x6,            xh0_1
                ADDSP      x3,         x7,            xh1_1
   [flag]       MV          x0,          xh0_0
   [flag]       MV          x1,          xh1_0
   [flag]       MV          x2,          xh0_1
   [flag]       MV          x3,          xh1_1
                ADDSP      xh0_0,       xh0_1,      yt0
                SUBSP      xh0_0,       xh0_1,      yt4
                SUBSP      xh1_0,       xh1_1,      yt5
                ADDSP      xh1_0,       xh1_1,      yt1
                SUBSP      x0,         x4,            xl0_0
                SUBSP      x1,         x5,            xl1_0
                SUBSP      x2,         x6,            xl0_1
                SUBSP      x7,         x3,            xl1_1
   [flag]       MV          x4,          xl0_0
   [flag]       MV          x5,          xl1_0
   [flag]       MV          x6,          xl1_1
   [flag]       MV          x7,          xl0_1
                ADDSP      xl0_0,       xl1_1,      yt2
                ADDSP     xl1_0,       xl0_1,      yt3
                SUBSP      xl0_0,       xl1_1,      yt6
                SUBSP      xl1_0,       xl0_1,      yt7
                ;Store appropriate values at appropriate offsets in output
                MPYSP     yt0,        scale,      yt0
                MPYSP     yt1,        scale,      yt1
                MPYSP     yt2,        scale,      yt2
                MPYSP     yt3,        scale,      yt3
                MPYSP     yt4,        scale,      yt4
                MPYSP     yt5,        scale,      yt5
                MPYSP     yt6,        scale,      yt6
                MPYSP     yt7,        scale,      yt7
                STW          yt0,          *y1++
                STW          yt1,          *y1++[nmax]
                STW          yt2,          *y1++
                STW          yt3,          *y1++[nmax]
                STW          yt4,          *y1++
                STW          yt5,          *y1++[nmax]
                STW          yt6,          *y1++
                STW          yt7,          *y1
                SUB          N,          4,        N
   [N]          B           top2
retFunc2:
                .return
                .endproc
 ;*****************************************************************************
 ; End of file: DSPF_sp_ifftSPxSP.sa
```

d. brev_table.h

```
;*****************************************************************************
;  TEXAS INSTRUMENTS, INC.
;  NAME  brev_table
;  USAGE This data is required to bit-reverse the output in the
;        sp_fftSPxSP routine. Paste this array onto your code
;        for use of the routine.
;*****************************************************************************
```

```
unsigned char brev[64] = {
0x0, 0x20, 0x10, 0x30, 0x8, 0x28, 0x18, 0x38,
0x4, 0x24, 0x14, 0x34, 0xc, 0x2c, 0x1c, 0x3c,
0x2, 0x22, 0x12, 0x32, 0xa, 0x2a, 0x1a, 0x3a,
0x6, 0x26, 0x16, 0x36, 0xe, 0x2e, 0x1e, 0x3e,
0x1, 0x21, 0x11, 0x31, 0x9, 0x29, 0x19, 0x39,
0x5, 0x25, 0x15, 0x35, 0xd, 0x2d, 0x1d, 0x3d,
0x3, 0x23, 0x13, 0x33, 0xb, 0x2b, 0x1b, 0x3b,
0x7, 0x27, 0x17, 0x37, 0xf, 0x2f, 0x1f, 0x3f
};
```

e. vectors.asm

```
======== vectors.asm ========
Plug in the entry point at RESET in the interrupt vector table

  ======== unused =========
plug inifinite loop -- with nested branches to
disable interrupts -- for all undefined vectors

unused  .macro id
        .global unused:id:
unused:id:
        b unused:id:                    ; nested branches to block interrupts
        nop 4
        b unused:id:
        nop
        nop
        nop
        nop
        nop
        .endm

        .sect ".vectors"
        .ref _c_int00               ; C entry point
        .align  32*8*4            ; must be aligned on 256 word boundary

RESET:                              ; reset vector
        mvkl _c_int00,b0       ; load destination function address to b0
        mvkh _c_int00,b0
        b b0                           ; start branch to destination function
        mvc PCE1,b0           ; address of interrupt vectors
        mvc b0,ISTP            ; set table to point here
        nop 3                        ; fill delay slot
        nop
        nop
        ;  plug unused interrupts with infinite loops to
        ;  catch stray interrupts
        unused 1
        unused 2
        unused 3
        unused 4
        unused 5
        unused 6
        unused 7
        unused 8
        unused 9
        unused 10
        unused 11
```

```
        unused 12
        unused 13
        unused 14
        unused 15
```

f. c6711dsp.cmd

```
/* c6711dsp.cmd */

MEMORY
{
   VECS:            o=0h,      l=200h              /* vecteurs d'interruption */
   SDRAM: o=0x200,  len =0x1000000             /* 16 Mo (toute mémoire confondue, interne et externe) */
}

SECTIONS
{
   .intvecs                  >      0h
   .text            >        SDRAM
   .stack           >        SDRAM
   .bss             >        SDRAM
   .const           >        SDRAM
   .data            >        SDRAM
}
```

2. Projet 4_BancFiltreMEL : bancFiltreMEL.c

```
#include <math.h>
unsigned int i, j, FiltresTotal;
const int duree=256, fs=8000, FiltresLIN=10, EspaceLIN=100, FiltresLOG=12;
const float FreqBasse=0, EspaceLOG=1.11253;
float freqs[24], bas[22], centre[22], haut[22], TailleTriangle[22],
fftFreqs[256];
float PoidsFiltres[22][256], Filtre[256];

void main(void)
{
/* paramètres du banc de filtres MEL-MFCC */
FiltresTotal = FiltresLIN + FiltresLOG;
// banc de 22 mel-filtres

// Localisation de toutes les bandes de fréquence. Les fréquences sont
espacées linéairement, puis  logarithmiquement. Les limites de bande (bas,
centre, haut) sont consécutives pour tous les filtres.
 for(i=0; i<FiltresLIN; i++) freqs[i] = FreqBasse + i*EspaceLIN;  //freqs avec
                                1 ligne et 10 colonnes

 for(i=FiltresLIN; i<FiltresTotal+2; i++)          // ajout de 14 colonnes, de
                             la 11ème à la 24ème colonne
   freqs[i] = freqs[FiltresLIN-1] * powf(EspaceLOG,i-9);
```

```
for(i=0; i<FiltresTotal; i++)
{  bas[i] = freqs[i];
   centre[i] = freqs[i+1];
   haut[i] = freqs[i+2];
  TailleTriangle[i] = 1/(haut[i]-bas[i]);          // normalisation (<1) du
                            poids de chaque filtre
};

for(i=0; i<duree; i++) fftFreqs[i] = fs*i/duree;    // abscisse fréquentiel
for(i=0; i<FiltresTotal; i++) for(j=0; j<duree-1; j++) PoidsFiltres[i][j] =
0;                                  // initialisation

  for(i=0; i<FiltresTotal; i++)            // réalisation de chaque gabarit
                                  trianglulaire
   for(j=0; j<duree-1; j++)
   {  PoidsFiltres[i][j] =
  (fftFreqs[j] > bas[i] && fftFreqs[j] <= centre[i]) *   // pente ascendante
    TailleTriangle[i]*(fftFreqs[j]-bas[i])/(centre[i]-bas[i]) +        //
                            pondération des sommets
 (fftFreqs[j] > centre[i] && fftFreqs[j] < haut[i]) *                // pente
                                  descendante
    TailleTriangle[i]*(haut[i]-fftFreqs[j])/(haut[i]-centre[i]);        //
                            pondération des minimums
      Filtre[j] = PoidsFiltres[i][j];
   };
}
```

3. Projet 5_mfcc : MFCC.c)

```
#include <stdio.h>
#include <math.h>
#include "brev_table.h"

#ifndef    PI
#define    PI    3.141592653
#endif

unsigned int i, j;                  /* compteurs de boucle */
const int N=256;   /* nombre d'éch. dans un segment traité (durée=32ms pour 2
                      périodes du pitch_min=62.5Hz)  */
```

```
const int base=4, offset=0; /* base=4 si N=x^4, base=2 si N=x^2, et décalage
                            d'indice d'éch. en FFT */
const int bande=18;           /* largeur de bande pour liftrage passe-bas et
                               passe-haut */
float ech[256], x[512];      /* éch. réels d'entrée, fenêtre de Hamming, éch.
                            complexes d'entrée */
 float w[512], y[512], ym[256], s[256];     /* facteurs complexes pour FFT,
                        spectre de sortie et son module */
  float freqs[24], fftFreqs[256], Filtre[256], Amp[22], melcep[22];    /*
                  param. du banc de filtre mel et sortie */
float MFCC[22];

void main(void)
{
     puts(" programme démarré: \n");
     puts(" lecture du fichier *.dat en utilisant la sonde 'probe point':
\n");
    s[0]=ech[0];                    // initialisation de la préaccentuation
    for(i=1; i<N; i++) {                // boucle de prétraitement d'entrée
         s[i]=ech[i]-0.97*ech[i-1];                  // préaccentuation
         s[i]*=0.54-0.46*cosf(6.28*i/N);             // fenêtrage de Hamming
         x[2 * i] = s[i];                          // éch. complexes d'entrée
         x[2 * i + 1] = 0;
         };
gen_twiddle(w, N);      // Génère les facteurs w (twiddle) pour le calcul FFT
 DSPF_sp_fftSPxSP(N, x, w, y, brev, base, offset, N);    // calcule en asm la
                            FFT complexe y de x
*ech = 0;                                    // initialisation de l'énergie
for(i=0; i<N; i++) {                         // traitement du spectre
    ym[i] = y[2*i]*y[2*i] + y[2*i+1]*y[2*i+1]; // module au carré du spectre
    s[i] = 10*log10f(ym[i]);                    // énergie spectrale en dB
    *ech += ym[i];                             // énergie du signal
    };
/* banc de 22 mel-filtres :
   Localisation de toutes les bandes de fréquence. Les fréquences sont
espacées linéairement, puis logarithmiquement. Les limites de bande (bas,
centre, haut) sont consécutives pour tous les filtres. */
for(i=0; i<10; i++) freqs[i] = i*100;     //freqs avec 1 ligne et 10 colonnes
for(i=10; i<24; i++)  // ajout de 14 colonnes, de la 11ème à la 24ème colonne
   freqs[i] = freqs[9] * powf(1.11253,i-9);
```

```
for(i=0; i<256; i++) fftFreqs[i] = 8000*i/256;          // axe fréquentiel : 0
à 8kHz en 256 points
 for(i=0; i<22; i++)      // réalisation de chaque gabarit trianglulaire puis
                           produit matriciel avec ym
{  Amp[i] = 0;                                        // initialisation
   for(j=0; j<256; j++)
   {  Filtre[j] = 0;                                   // initialisation
       Filtre[j] =
      (fftFreqs[j] > freqs[i] && fftFreqs[j] <= freqs[i+1]) *
// pente ascendante
    (fftFreqs[j]-freqs[i])/((freqs[i+1]-freqs[i])*(freqs[i+2]-freqs[i])) +
                           // pondération des sommets
      (fftFreqs[j] > freqs[i+1] && fftFreqs[j] < freqs[i+2]) *
// pente descendante
      (freqs[i+2]-fftFreqs[j])/((freqs[i+2]-freqs[i+1])*(freqs[i+2]-
freqs[i]));                  // pondération des minimums
      Filtre[j] *= ym[j];
      Amp[i] += Filtre[j];
   };
 melcep[i] = 10*log10f(Amp[i]**ech);                  // spectre mel cepstral
                        (pondéré par l'énergie), en dB
};
 for (j = 0; j < 22; ++j) {               // calcul des 22 MFCC (transformée
                             en cosinus de melcep)
     MFCC[j] = 0;                    // initialisation
  for (i = 0; i < 22; i++)                          // pondéré par 1/sqrt(22) si
                         j=0, et par sqrt(2/22) si j>0
         MFCC[j] += ((j==0)*0.2132 + (j!=0)*0.3015) * melcep[i] *
cosf(j*PI*(i+0.5)/22);
    };
printf(" Programme terminé !\n");
}
```