République Algérienne Démocratique et populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Ecole Nationale Polytechnique

Département d'Electronique
Laboratoire de Signal et Communication

Mémoire de MAGISTER

En

Signal et Communications

Présenté par
AHMED SAID Aziz

Ingénieur d'Etat en Electronique

Thème :

Implémentation d'un codeur de parole multipulse à 8 Kb/s en temps réel sur TMS 320C30

Soutenu le : 29 Novembre 2003

Devant le jury composé de :

R. AKSAS	Professeur. ENP	Président du jury
D. BERKANI	Professeur. ENP	Rapporteur
M. GUERTI	Maître de conférence. ENP	Examinateur
L. HAMANI	Dr . ENP	Examinateur
R. SADOUN	Chargé de cours. ENP	Examinateur

IMPLEMENTATION D'UN CODEUR DE PAROLE MULTIPULSE

A 8 kb/s EN TEMPS REEL SUR TMS320C30

Réalisée par :

AHMED SAID Aziz

Ingénieur d'Etat en Electronique

Septembre 2003

TABLE DES MATIERES

INTRODUCTION

Chapitre I. **Caractéristique de la parole** 5

I.1 Le modèle prédictif linéaire du signal vocal 13
I.2 Codage par LPC 15

Chapitre.II **Codage de la parole** 17

II.1 Vue générale sur les codeurs de forme d'onde 17
II.2 Codage par prédiction linéaire et analyse par synthèse 20

Chapitre.III **CODEUR MULTI-PULSE**

III.1 Introduction 23

III.2 Le predicteur court terme 24

III.3 Les LSP 28
III.3.1 Calcul des LSP 29
III.3.2 Conversion des LSP en paramètres LPC 30

III.4 Le predicteur long terme 31

III.5 Le filtre de pondération de l'erreur 32

III.6 L'Excitation
III.6.1 Position du problème 34
III.6.2 L'approche Multi-Puls 37
III.6.3 Modification de l'algorithme MPE 40

III.7 La quantification 42
III.7.1 Quantification des LSP 43
III.7.2 Quantification de l'excitation 44

Chapitre.IV **L'implémentation**

IV.1 Le matériel 45
IV.1.1 L'EVM 45
IV.1.2 Le TMS320C30 46
IV.1.3 L'AIC 47
IV.2 Le SOFT 48

IV.3 La Programmation 49
IV.3.1 Organisation des trames de parole 49
IV.3.2 Aperçu du programme principal 53
IV.3.3 Les routines de codage 56
IV.3.3-1 Les variables globales 56

IV.3.3.2 Fonction d'initialisation des variables 59
IV.3.3.3 Fonction de fenêtrage 60
IV.3.3.4 Fonction de calcul des coefficients LSP 60
IV.3.3.5 Fonction de calcul du résidu 62
IV.3.3.6 Fonction de calcul du signal Pondéré 63
IV.3.3.7 Fonction de recherche de maximum 64
IV.3.3.8 Fonction de calcul de la réponse implusionelle du filtre de synthèse 65
IV.3.3.9 Fonction de calcul des autocorrelations de h(n) 68
IV.3.3.10 Fonction de calcul de l'excitation 70
IV.3.3.11 Fonction d'arrangement des impulsions 73
IV.3.3.12 Fonction de synthèse de la parole 73
IV.3.3.13 Fonction de filtrage de sortie 74
IV.3.3.14 Fonction de calcul des LSP 76
IV.3.3.15 Fonction des conversion des LSP en paramètres LPC 78
IV.3.3.16 Fonction de quantification d'une valeur réelle 79
IV.3.3.17 Fonction de quantification des LSP 80
IV.3.3.18 Fonction de Quantification de l'excitation 81
IV.3.3.19 Fonction de décodage des LSP 83
IV.3.3.20 Fonction de décodage des impulsions d'excitations 84
IV.3.4 Le programme principal 86
IV.3.5 Test 88

CONCLUSION

ANNEXE
BIBLIOGRAPHIE

DEDICACE

A mes parents je dédie ce travail qui est le résultat des mes efforts, mais avant tout de leurs énormes sacrifices pour m'assurer la quiétude nécessaire à la réflexion et à la recherche.

Aziz,

REMERCIEMENTS

J'exprime ici ma gratitude et mon respect à mon directeur de thèse M. D. BERKANI, pour m'avoir suivi avec patience et égards tout au long de ce travail

J'adresse aussi, mes remerciements aux membres du jury qui ont bien voulu me faire l'honneur d'examiner et d'évaluer ce travail.

Enfin, je remercie les membres de ma famille et en particulier ma sœur pour son aide précieuse et tous les amis qui m'ont patiemment soutenu.

ملخص :

هذا العمل يعرض كيفية تزويد جهاز DSP TMS 320 C30 التابع لشركة _(تكساس أسترومنت) (TEXAS INSTRUMENT) برامزة الكلمة المتعدّدة النبضات لـــ 8 ك. ب/س . (CODEUR DE PAROLE MULTI-PULSE ă KB/S) .

بعد تلخيص موجز عن تقنيات ترميز الكلمة ، عرضنا نظرية الرامزات المتعددة النبضات متبوعة بتفاصيل إدخال الرامزة في بطاقة التقييم لـ C30 ثم الإدماج بإستعمال لغة C بالمؤلف © (COMPILATEUR C) متطابق ANSI لـ TI الخاص بـ (DSP) من عائلات C3X و C4X .الرامزة المتحصل عليها ،تنتج الكلمة بأحسن نوعية و في وقتها الطبيعي .

الكلمات المفتاحية : رمز ،كلمة ،متعدد النبضات ، DSP ، TMS 320 C30.

Résumé

Ce travail présente l'implémentation d'un codeur de parole multi-pulse à 8 kb/s en temps réel sur le DSP TMS320C30. Après un bref résumé des techniques de codage de la parole, la théorie des codeurs multi-pulse est exposée suivi des détails de l'implémentation du codeur sur la carte d'évaluation du C30. L'implémentation a été faite en langage C avec le compilateur C compatible ANSI de TI pour les DSP des familles C3X et C4X. Le codeur obtenu, produit une bonne qualité de parole et en temps réel.
Mots clés : Codage, Parole, Multi-Pulse, DSP, TMS320C30.

Summary

This work presents the implementation on the DSP TMS320C30 of a real time multi-pulse speech coder at 8kb/s. After a brief summary of speech coding techniques, the theory of the multi-pulse coders is presented followed by the implantation detail on the C30 evaluation board. The implementation has been carried out in C language using the Texas Instrument C30x/C4X ANSI
Compiler. The obtained coder produces a good qualty speech and in real time.
Key word : Multi-pulse, Speech, Coding, DSP, TMSC320.

INTRODUCTION

Introduction

Le traitement de la parole est aujourd'hui une composante fondamentale des sciences de l'ingénieur. Située au croisement du traitement du signal numérique et du traitement du langage (c'est-à-dire du traitement de données symboliques), cette discipline scientifique a connu depuis les années 60 une expansion fulgurante, liée au développement des moyens et des techniques de télécommunications.
L'importance particulière du traitement de la parole dans ce cadre plus général s'explique par la position privilégiée de la parole comme vecteur d'information dans notre société humaine.

L'extraordinaire singularité de cette science, qui la différencie fondamentalement des autres composantes du traitement de l'information, tient sans aucun doute au rôle fascinant que joue le cerveau humain à la fois dans la production et dans la compréhension de la parole et à l'étendue des fonctions qu'il met en œuvre pour y parvenir de façon pratiquement instantanée.

La parole la vue et le mouvement sont en effet les seuls à être à la fois *produit* et *perçu* instantanément par le cerveau. La parole est produite par le conduit vocal, contrôlé en permanence par le cortex moteur. L'étude des mécanismes de phonation permettra donc de déterminer, dans une certaine mesure, ce qui est parole et ce qui n'en est pas. De même, l'étude des mécanismes d'audition et des propriétés perceptuelles qui s'y rattachent permettra de dire ce qui, dans le signal de parole, est réellement perçu. Mais l'essence même du signal de parole ne peut être cernée de façon réaliste que dans la mesure où l'on imagine, bien au-delà de la simple mise en commun des propriétés de production et de perception de la parole, les propriétés du signal dues à la mise en boucle de ces deux fonctions. Mieux encore, c'est non seulement la perception de la parole qui vient influer sur sa production par le biais de ce bouclage, mais aussi et surtout sa compréhension.

On ne parle que dans la mesure ou l'on s'entend et où l'on se comprend soi-même; la complexité du signal qui en résulte s'en resent forcément.

S'il n'est pas en principe de parole sans cerveau humain pour la produire, l'entendre, et la comprendre, les techniques modernes de traitement de la parole tendent cependant à produire des systèmes automatiques qui se substituent à l'une ou l'autre de ces fonctions :

• Les *reconnaisseurs* ont pour mission de décoder l'information portée par le signal vocal à partir des données fournies par l'analyse.

• Les *synthétiseurs* ont quant à eux la fonction inverse de celle des reconnaisseurs de parole : ils produisent de la parole artificielle.

• Enfin, le rôle des *codeurs* est de permettre la transmission ou le stockage de parole avec un débit réduit, ce qui passe tout naturellement par une prise en compte judicieuse des propriétés de production et de perception de la parole.

Ce travail présente l'implémentation d'un codeur de parole multi-pulse à 8 kb/s en temps réel sur le DSP TMS320C30.

Dans le premier chapitre les caractéristiques de la parole sont brièvement exposées. Le deuxième chapitre, donne un bref résumé des techniques de codage de la parole. Le troisième chapitre, expose toute la théorie des codeurs multi-pulse. Et le quatrième chapitre, aborde tous les détails de l'implémentation du codeur sur la carte d'évaluation du C30 et les explications du programme entier. A la fin, l'annexe donne le code source complet.

Par codeur en temps réel, il est entendu un codeur qui traite la parole en même temps que celle-ci est échantillonnée et non après son enregistrement. Le codage ne se fait pas instantanément mais avec un certain temps de retard, qui comme on le verra dans le chapitre II est de 45 ms pour le codeur présenté.

Le matériel utilisé pour l'implémentation est le module d'évaluation du processeur de traitement du signal TMS320C30 de Texas Instrument. L'implémentation est faite en

langage C avec le compilateur C compatible ANSI de TI pour les DSP des familles C3X et C4X.

Les objectifs de ce travail sont l'implémentation d'un codeur en temps réel, une bonne qualité de parole et la flexibilité du code.

Chapitre I

Caractéristique de La Parole

La parole apparaît physiquement comme une variation de la pression de l'air causée et émise par le système articulatoire. La *phonétique acoustique* étudie ce signal en le transformant dans un premier temps en signal électrique grâce au transducteur approprié : le microphone (lui-même associé à un préamplificateur).

De nos jours, le signal électrique résultant est le plus souvent numérisé. Il peut alors être soumis à un ensemble de traitements statistiques qui visent à en mettre en évidence les *traits acoustiques* : sa *fréquence fondamentale*, son *énergie*, et son *spectre*. Chaque trait acoustique est lui-même intimement lié à une grandeur perceptuelle : *pitch*, *intensité*, et *timbre*.

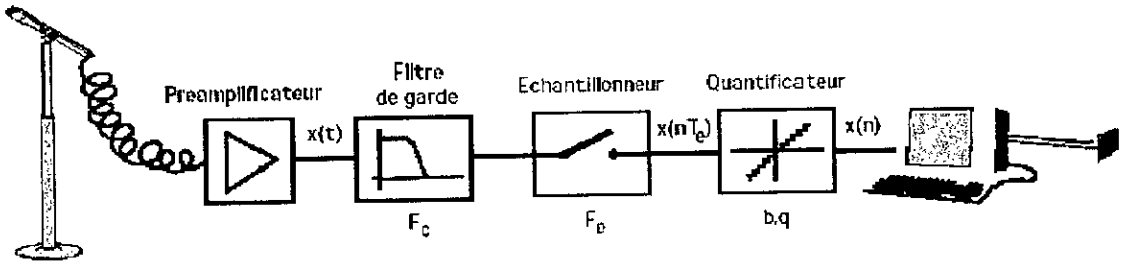


Figure. I.1 Enregistrement numérique d'un signal acoustique. La fréquence de coupure du filtre de garde, la fréquence d'échantillonnage, le nombre de bits et le pas de quantification sont respectivement notés fc , fe , b, et q.

L'échantillonnage (Figure. I.1) transforme le signal à temps continu *x(t)* en signal à temps discret *x(n)* défini aux instants d'échantillonnage, multiples entiers de la période d'échantillonnage; celle-ci est elle-même l'inverse de la fréquence d'échantillonnage. Pour ce qui concerne le signal vocal, le choix de cette fréquence d'échantillonnage résulte d'un compromis. Son spectre peut s'étendre jusque 12 kHz. Il faut donc en principe choisir une fréquence égale à 24 kHz au moins pour satisfaire raisonnablement au théorème de Shannon. Cependant, le coût d'un traitement numérique, filtrage, transmission, ou simplement enregistrement peut être réduit d'une façon notable si l'on

Chapitre I

CARACTERISTIQUE DE LA PAROLE

accepte une limitation du spectre par un filtrage préalable. Pour la téléphonie, on estime que le signal garde une qualité suffisante lorsque son spectre est limité à 3400 Hz et l'on choisit une

fréquence d'échantillonnage égale à 8000 Hz. Pour les techniques d'analyse, de ou de reconnaissance de la parole, la fréquence peut varier de 6000 à 16000 Hz. Par pour le signal audio (parole et musique), on exige une bonne représentation jusque 20 kHz et l'on utilise des fréquences d'échantillonnage de 44.1 ou 48 kHz.

Parmi le continuum des valeurs possibles pour les échantillons *x(n)*, la quantification retient qu'un nombre fini *2b* de valeurs (*b* étant le nombre de bits de la quantification), espacées du pas de quantification *q*. Le signal numérique résultant est noté *x(*quantification de bonne qualité requiert en général 16 bits.

Une caractéristique essentielle qui résulte du mode de représentation est le debit exprimé en bits par seconde (b/s), nécessaire pour une transmission ou un enregistrement du signal vocal. La transmission téléphonique classique sur une ligne RNIS exige de 8 kHz x 8 bits = 64 kb/s; la transmission ou l'enregistrement d'un signal audio principe un débit de l'ordre de 48 kHz x 16 bits = 768 kb/s (à multiplier par deux signal stéréophonique).

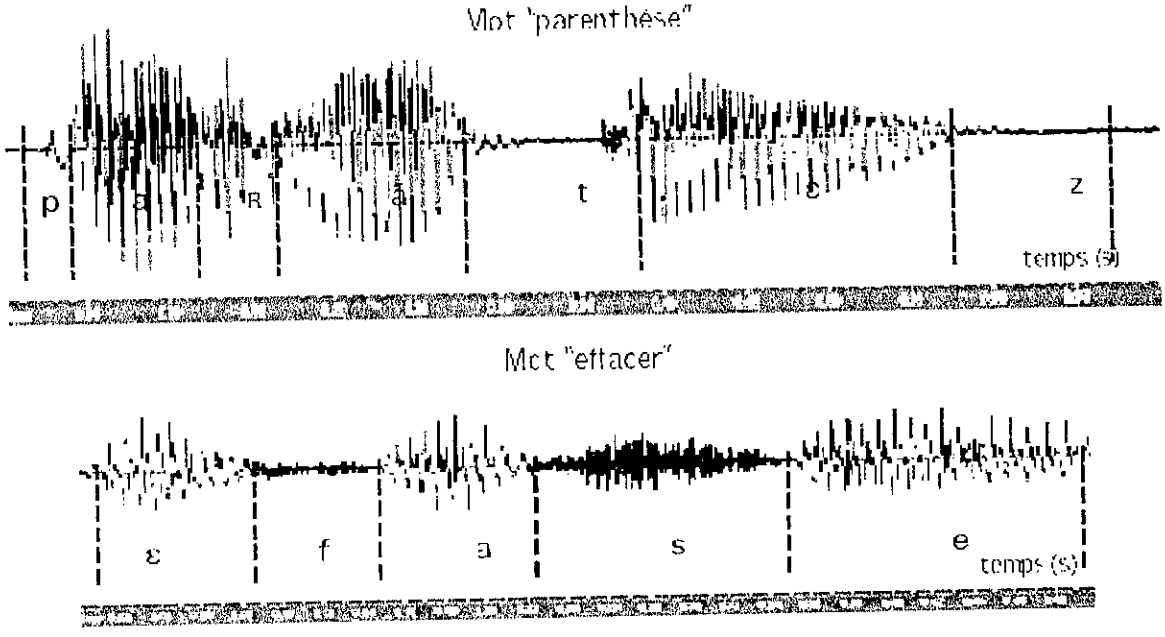


Figure. I.2 Audiogramme de signaux de parole.

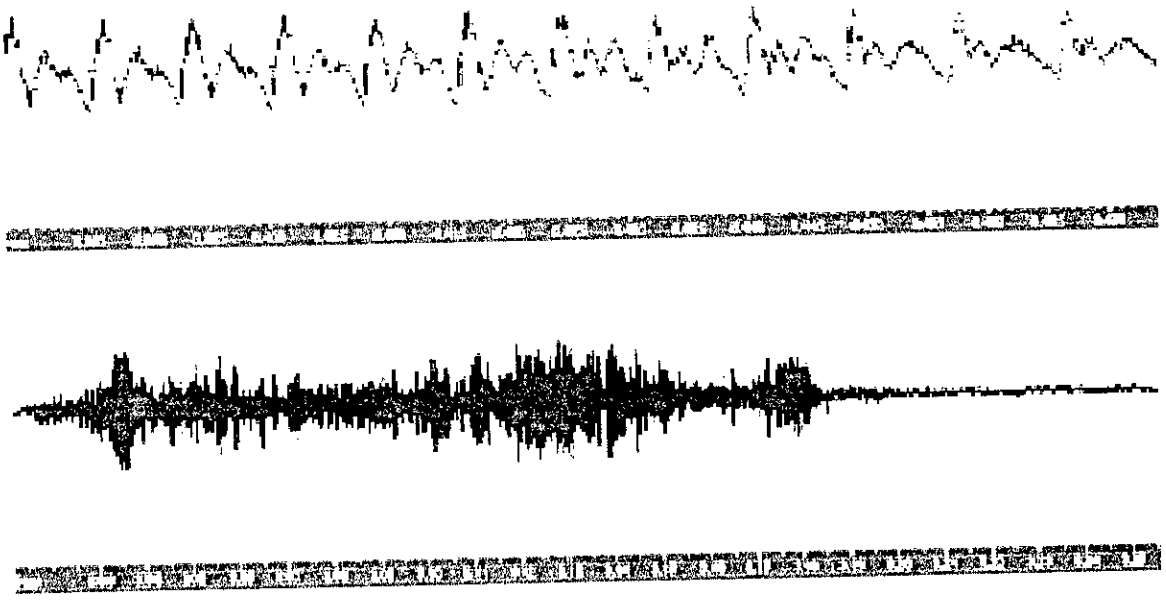


Figure. I.3 Exemples de son voisé (haut) et non-voisé (bas).

La figure I.2 représente l'évolution temporelle, ou *audiogramme*, du signal vocal pour les mots 'parenthèse', et 'effacer'. On y constate une alternance de zones assez périodiques et de zones bruitées, appelées zones *voisées* et *nonvoisées*.
La figure I.3 donne une représentation plus fine de tranches de signaux voisés et non voisés. L'évolution temporelle ne fournit cependant pas directement les traits acoustiques du signal. Il est nécessaire, pour les obtenir, de mener à bien un ensemble de calculs ad-hoc.

La transformée de Fourier à court terme est obtenue en extrayant de l'audiogramme une trentaine de millisecondes de signal vocal et en effectuant un transformée de Fourier sur ces échantillons. Le résultat de cette transformation mathématique est souvent présenté dans un graphique qui donne, en fonction de la fréquence, l'amplitude des composantes présentes dans le signal analysé.

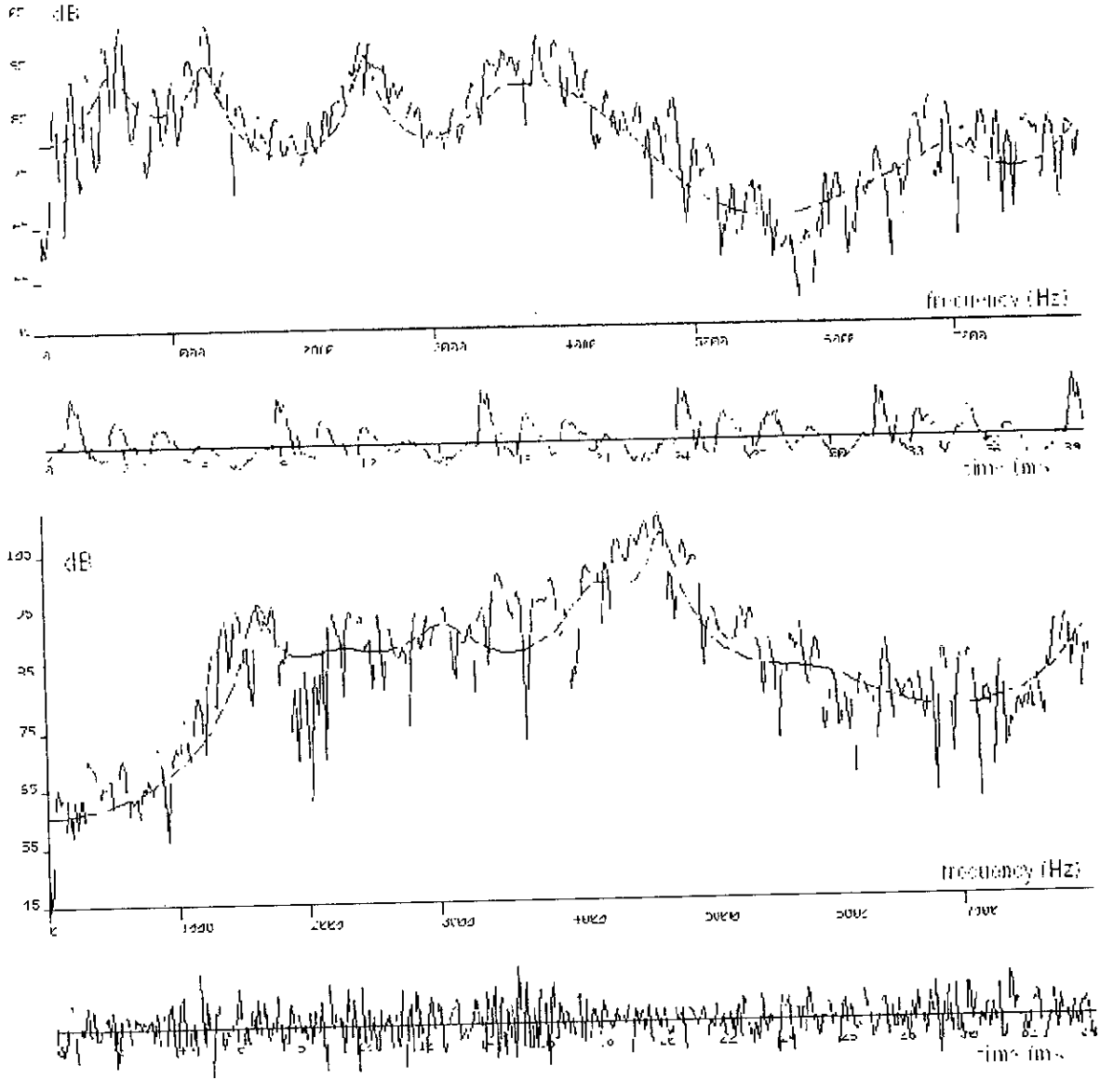


Figure. I.4 Evolution temporelle (en haut) et transformée de Fourier discrète (en bas) du [a] et du [] de *'baluchon'* (tranche de 30 ms).

La figure I.4 illustre la transformée de Fourier d'une tranche voisée et celle d'une tranche non-voisée. Les parties voisées du signal apparaissant sous la forme de successions de pics spectraux marqués, dont les fréquences centrales sont multiples de la fréquence fondamentale. La forme générale de ces spectres, appelée *enveloppe spectrale*, présente elle-même des pics et des creux qui correspondent aux résonances et aux anti-résonances du conduit vocal et sont appelés *formants* et *anti-formants*. L'évolution temporelle de leur fréquence centrale et de leur largeur de bande détermine

le timbre du son. Le spectre d'un signal de type voisé possède en général plus de composantes en basse fréquence qu'en haute fréquence. Par contre, le spectre d'un signal non voisé présente une amplitude plus importante en hante fréquence (ce qui correspond à la perception que nous en avons : les fricatives non-voisées [f,s,p] sont des sons plus « aigus » que les voyelles).

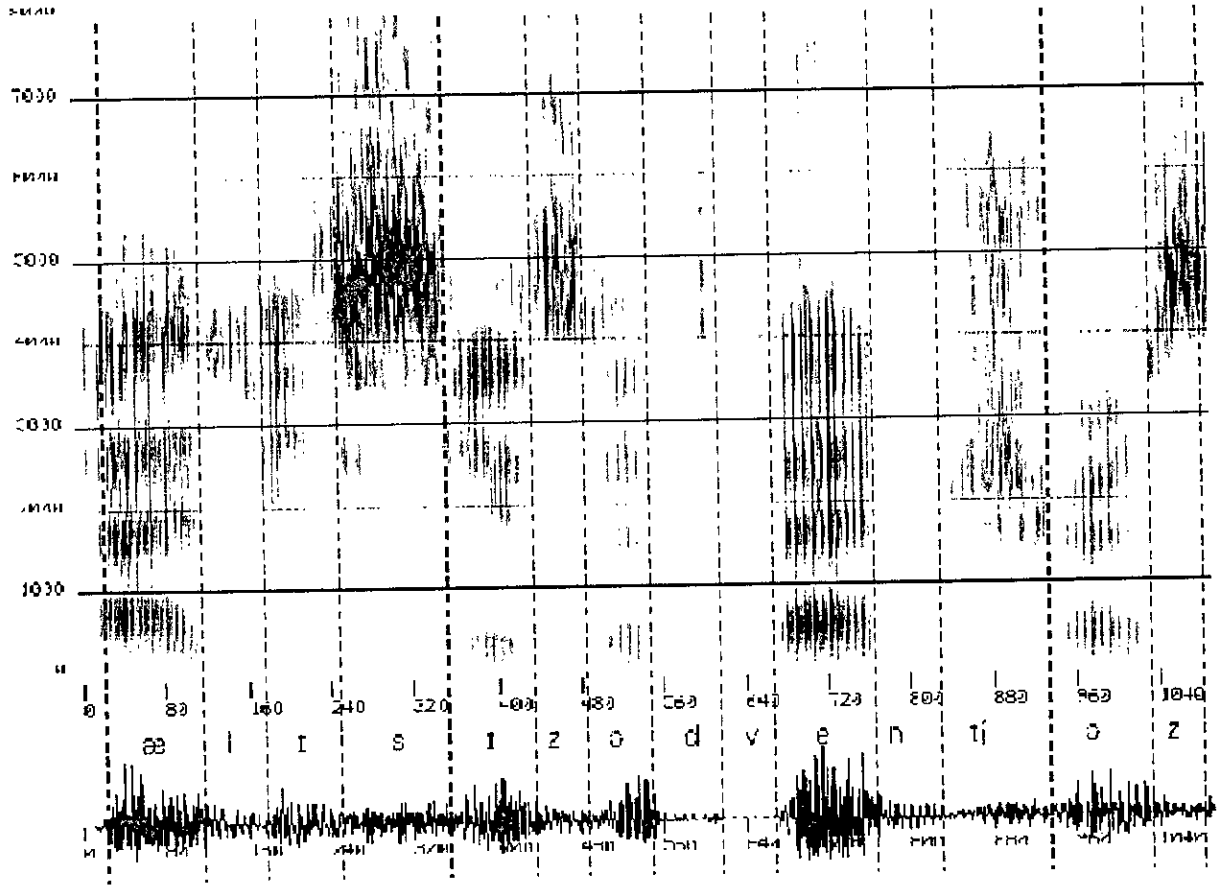


Figure. I.5 Spectrogramme et audiogramme de la phrase anglaise *'Alice's adventures'*, échantillonnée à 11.25 kHz.

Il est souvent intéressant de représenter l'évolution temporelle du spectre à court terme d'un signal, sous la forme d'un *spectrogramme*. L'amplitude du spectre y apparaît sous la forme de niveaux de gris dans un diagramme en deux dimensions temps-fréquence. Ils mettent en évidence l'enveloppe spectrale du signal, et permettent par conséquent de visualiser l'évolution temporelle des formants. On reconnaît sans peine, à la figure 1.5, les formants caractéristiques des voyelles, et les composantes à plus hautes

fréquences caractéristiques des consonnes. La position et l'évolution des formants est caractéristique des sons produits. La seule lecture d'un spectrogramme (sans l'écoute du signal correspondant) permet d'ailleurs à l'œil expérimenté de certains phonéticiens de retrouver le contenu du message parlé. Cette propriété n'est évidemment pas vraie pour l'audiogramme, qui renseigne peu sur le timbre des sequences sonores produites. C'est donc bien que le spectrogramme présente sous une forme simple l'essentiel de l'information portée par le signal vocal.

Notons pour terminer qu'une analyse d'un signal de parole n'est pas complete tant qu'on n'a pas mesuré l'évolution temporelle de la fréquence fondamentale ou *pitch*. La figure I.6 donne l 'évolution temporelle de la fréquence fondamentale de la phrase *"les techniques de traitement de la parole"*. On constate qu'à l'intérieur des zones voisées la fréquence fondamentale évolue lentement dans le temps. Elle s'étend approximativement de 70 à 250 Hz chez les hommes, de 150 à 400 Hz chez les femmes, et de 200 à 600 Hz chez les enfants.

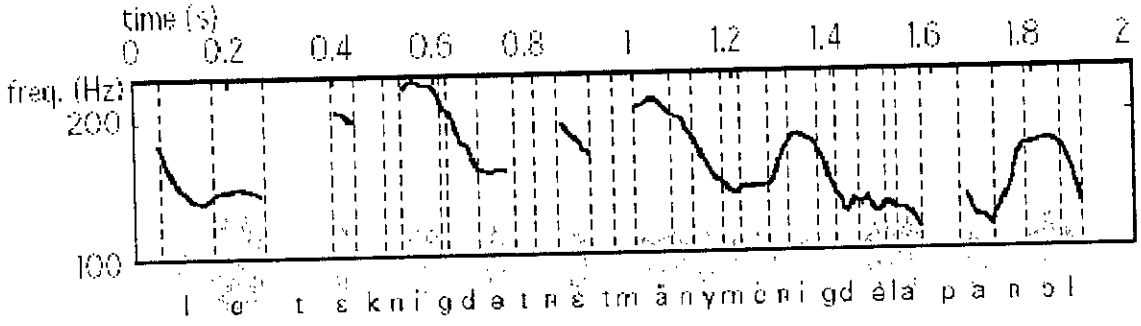


Figure. I.6 Evolution de la fréquence de vibration des cordes vocales dans la phrase "*les techniques de traitement numérique de la parole*". La fréquence est donnée sur une échelle logarithmique; les sons non-voisés sont associés à une fréquence nulle..

Les traits acoustiques du signal de parole sont évidemment liés à sa production. L'intensité du son est liée à la pression de l'air en amont du larynx. Sa fréquence, qui n'est rien d'autre que la fréquence du cycle d'ouverture/fermeture des cordes vocales, est déterminée par la tension de muscles qui les contrôlent. Son spectre résulte du filtrage dynamique du signal glottique (impulsions, bruit, ou combinaison des deux) par le conduit vocal, qui peut être considéré comme une succession de tubes ou de cavités acoustiques de sections diverses. Ainsi, par

exemple, on peut approximativement représenter les voyelles dans le plan des deux premiers formants (Figure. I.7).

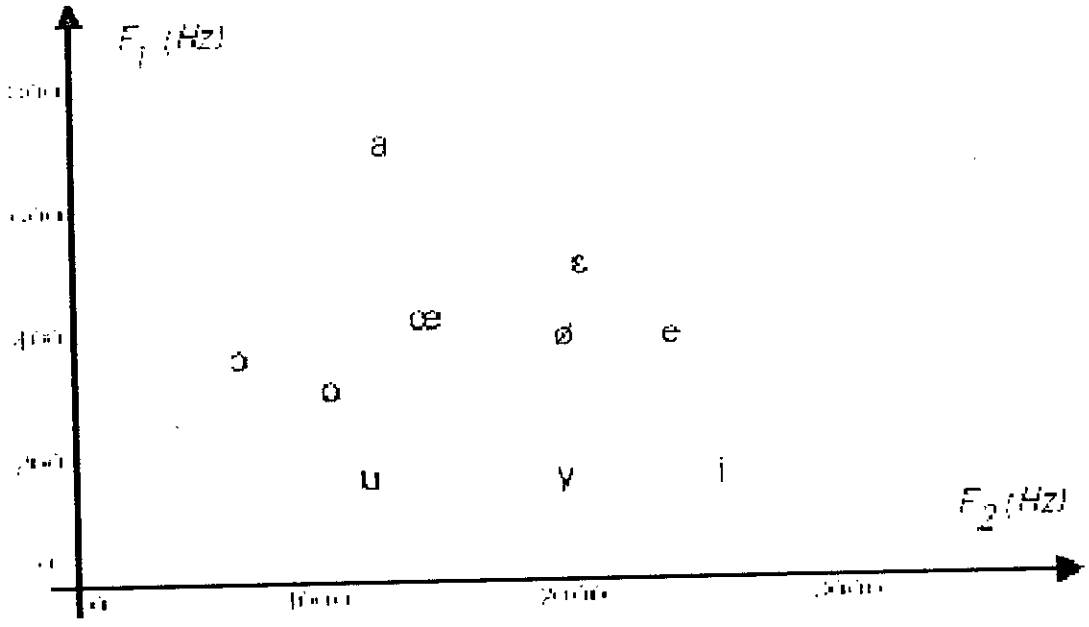


Figure. I.7 Représentation des voyelles dans le plan F1-F2

On observe en pratique un certain recouvrement dans les zones ormantiques correspondant à chaque voyelle. D'une manière plus générale, d'ailleurs, voyelles et consonnes apparaissent sous une multitude de formes articulatoires, appelées *allophones* (ou *variantes*). Celles-ci résultent soit d'un changement volontaire dans l'articulation d'un son de base comme cela arrive souvent dans les prononciations régionales (ex : les différentes prononciations régionales du [ô] en français). De telles variations ne portent aucune information sémantique. Les variantes phoniques sont également causées, et ce de façon beaucoup plus systématique, par l'influence des phones environnants sur la dynamique du conduit vocal. Les mouvements articulatoires peuvent en effet être modifiés de façon à minimiser l'effort à produire pour les réaliser à partir d'une position articulatoire donnée, ou pour anticiper une position à venir. Ces effets sont connus sous le nom de *coarticulation.* Les phénomènes coarticulatoires sont dus au fait que chaque articulateur évolue de façon continue entre les positions articulatoires. Ils apparaissent même dans le parlé le plus soigné (on en trouve un exemple frappant à la figure I.8). Ces phénomènes de coarticulation sont en

grande partie responsables de la complexité des traitements realizes sur les signaux de parole pour en obtenir l'analyse, la reconnaissance, ou la synthèse.

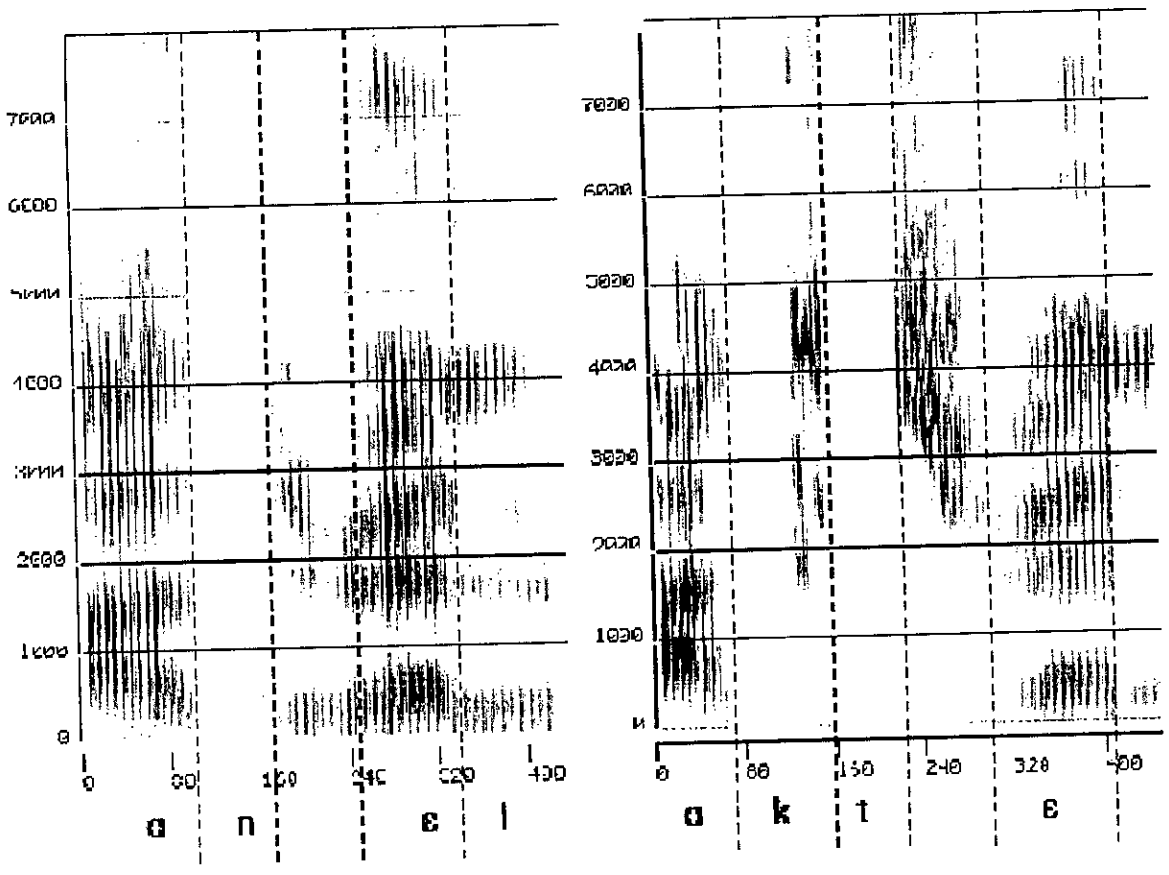


Figure. I.8 Un cas d'assimilation de sonorité (coarticulation affectant le voisement d'une sonorc). A gauchc, le début du mot *'annuellement'*, dans lequel [ ] est placé dans un contexte voisé. A droite, le début de *'actuellement'* : [ ] est totalement dévoisé à cause de la plosive sourdc qui précède

I.1 LE MODÈLE PRÉDICTIF LINÉAIRE DU SIGNAL VOCAL

L'analyse de la parole est une étape indispensable à toute application de synthèse, de codage, ou de reconnaissance. Elle repose en général sur un *modèle*. Celui-ci possède un ensemble de *paramètres* numériques, dont les plages de variation définissent l'ensemble des signaux couverts par le modèle.

Pour un signal et un modèle donné, l'*analyse* consiste en l'*estimation* des paramètres du modèle dans le but de lui faire correspondre le signal analysé. Pour ce faire, on met en oeuvre un *algorithme d'analyse*, qui cherche généralement à minimiser la différence, appelée *erreur de modélisation*, entre le signal original et celui qui serait produit par le modèle s'il était utilisé en tant que synthétiseur (Figure. I.9).

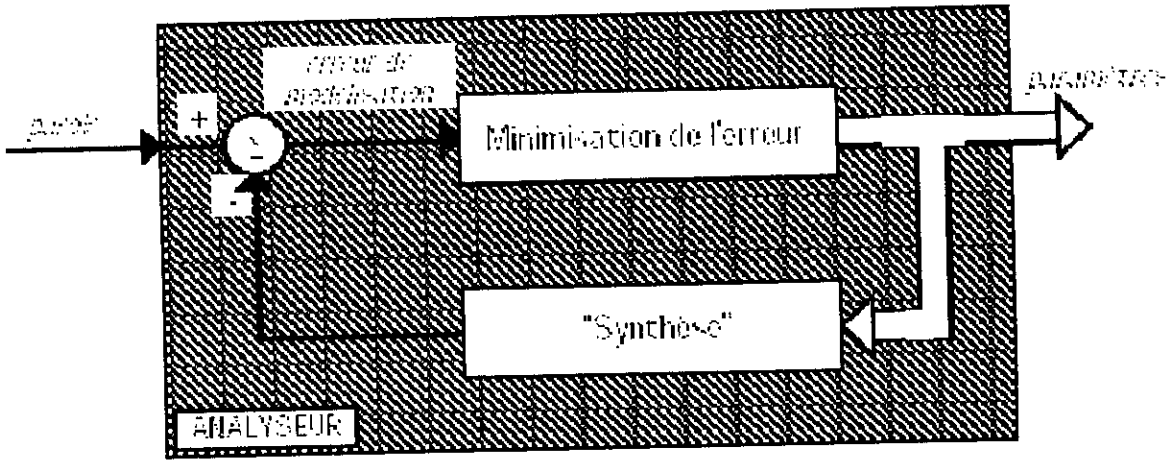


Figure. I.9 Schéma de principe d'un analyseur de parole. En pratique, l'étape de synthèse peut être implicite.

S'il existe de nombreux modèles de parole, il en est un que l'on retrouve partout, et dans un nombre croissant d'appareils « grand public » : le modèle prédictif linéaire (LPC : Linear Predictive Coding).

signal d'excitation numérique à travers un filtre numérique récursif. Le signal d'excitation sera tantôt une suite d'impulsions numériques (qui serviront à simuler les impulsions de débit créées par les cordes vocales) tantôt du bruit numérique (qui reproduira le souffle poussé par les poumons).

Ce modèle est appelé « prédictif linéaire » en raison du fait qu'il correspond à une régression linéaire très simple entre le signal d'excitation et le signal vocale produit. Les coefficients de cette régression linéaire sont les coefficients du filter numérique récursif.

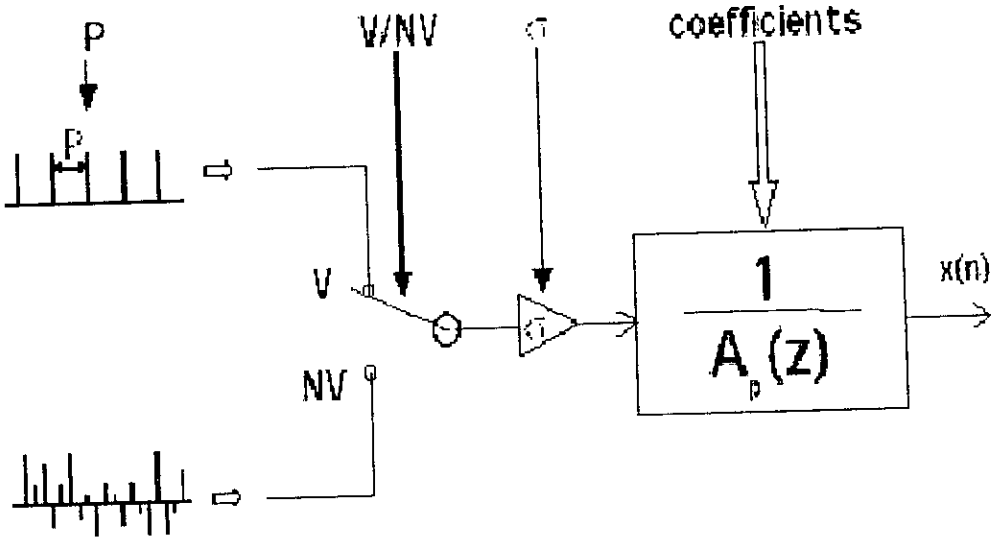


Figure. I.10 Le modèle auto-régressif.

Les paramètres du modèle LPC sont tout simplement : la période du train d'impulsions (sons voisés uniquement), la position de l'interrupteur Voisé/NonVoisé (V/NV), le gain de l'amplificateur σ, et les coefficients du filter numérique de synthèse.

Le problème de l'estimation d'un modèle LPC, souvent appelée *analyse LPC* revient à déterminer les coefficients du filtre connaissant 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*. La mise en équation de ce problème conduit aux équations dites de *Yule-Walke*. Les inconnues sont les coefficients du filtre de synthèse,

dont le nombre, p, est typiquement de l'ordre de 10. Le signal de parole étant fortement non-stationnaire, ce type de modélisation ne reste guère valable plus d'une dizaine de millisecondes. On retiendra donc que l'analyse LPC d'un signal de parole implique la résolution d'un système de 10 d'équations à 10 inconnues toutes les 10 ms.

I.2 CODAGE LPC

La figure ci-dessous donne le schéma de principe d'un codeur LPC, tel qu'il peut être utilisé pour les transmissions de voix par satellite (ex : voix d'un journaliste en mission dans un pays lointain) ou plus communément dans un GSM. Le signal vocal mesuré par le micro est découpé en trames, analysé par l'algorithme de Schur et par un algorithme d'analyse de la fréquence des cordes vocales. Les paramètres qui en résultent sont *quantifiés*, c.-à-d. qu'ils sont codés sur un ensemble fini de nombres entiers (ce qui permet d'associer à chaque paramètre un nombre fini de *bits* par trame).

En d'autres termes, lors d'un appel par GSM, le GSM émetteur (qui n'est rien d'autre qu'un ordinateur de poche spécialisé dans l'analyse, le codage, le décodage, et la synthèse LPC) enregistre la parole à transmettre, en réalise toutes les 10 millisecondes une analyse LPC (par laquelle il trouve les coefficients de prédiction qui « collent » le mieux au conduit vocal de l'appelant, pour la tranche de parole considérée), et transmet ces coefficients (et non la voix originale de l'appelant). Le GSM récepteur reçoit quant à lui les paramètres du conduit vocal de l'appelant, produit un signal de synthèse simulant ce conduit vocal, et le fait entendre au correspondant, qui croit entendre l'appelant. Il s'agit pourtant bien de parole de synthèse.

Chapitre II

CODAGE DE LA PAROLE

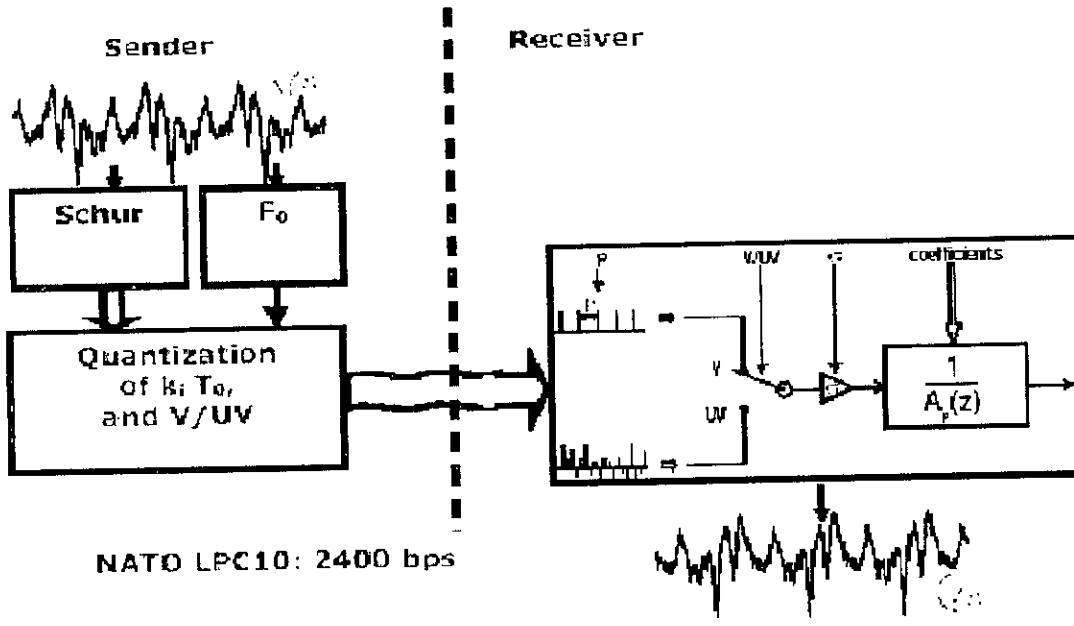


Figure. I.11 Transmission de parole basée sur le codage LPC.

Les débits de transmission obtenus avec ce type de modèle (et ses perfectionnements) sont respectivement de 2400 bits par seconde pour la transmission de voix par satellite, et de 13400 bits par seconde pour la transmission par GSM. Ces chiffres sont à comparer aux 64000 bits par secondes pour la téléphonie numérique. Le GSM n'aurait tout simplement pas pu voir le jour sans les efforts consentis pour parvenir à ces débits. (voir référence [8] pour plus de détail)

En conclusion, la parole est caractérisée par le timbre, l'intensité et le pitch. Pour la téléphonie, le spectre de la parole est limité à 3400Hz, une fréquence d'échantillonnage de 8000Hz est utilisée est une quantification sur 8 a 16 bits est suffisante

Chapitre II

Codage de la parole

Le codage de la parole a pour but la réduction du nombre de bits (le débit) requis pour la transmission de la parole tout en préservant une bonne qualité.

C'est un thème très important dans la téléphonie mobile, la réduction du débit d'un codeur se traduit directement par l'augmentation du nombre d'abonnés et donc des bénéfices. Mais cette réduction du débit ne doit pas conduire à des codeurs trop complexes qui ne seraient pas implémentables sur une seule puce monolithique.

Le compromis entre la réduction du débit et la complexité a été réalisé [1] pour des débit de 16 à 8 kb/s par des codeurs comme le RPE (*regular pulse exited linear predictive codec*), le MPE (*multi-pulse exited linear predictive codec*) et un nombre de codeur en sous bandes (SBC).

Les codeurs de parole peuvent être classer en deux catégories, les codeurs de forme d'onde et les codeurs d'analyse par synthèse.

II.1 VUE GENERALE SUR LES CODEURS DE FORME D'ONDE

Les codeurs de forme d'onde sont des codeurs qui reçoivent les échantillons 'analogiques' et les numérisent avant la transmission. A la réception le décodeur inverse le processus de codage pour recouvrir le signal de parole. En l'absence d'erreurs de transmission, la forme d'onde de la parole reconstituée ressemble beaucoup à la forme d'onde de la parole originale.

Le codeur de forme d'onde le plus simple est le PCM (*pulse code modulation*) où la parole est échantillonnée à la fréquence de Nyquist, et chaque échantillon est représenté par un nombre binaire [1]. La PCM logarithmique a été standardisée par le CCITT à 64 kb/s [1]. Les échantillons sont compressés logarithmiquement avant d'être quantifiés et à la réception ils sont expansés logarithmiquement après le décodage (figure 1).

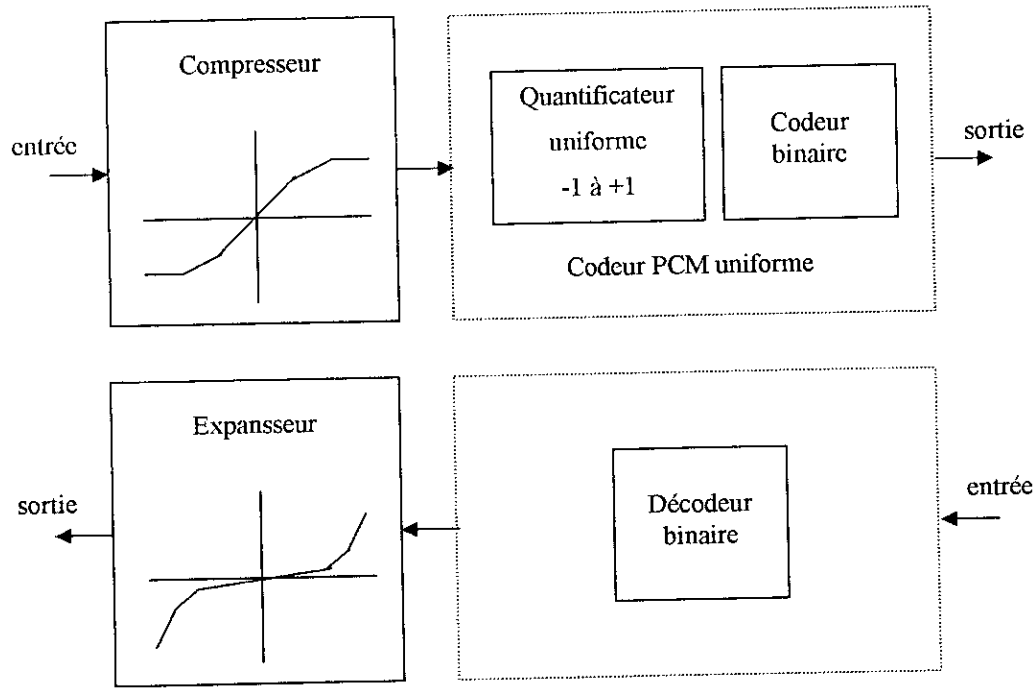


Figure 1 : Codeur/Décodeur
PCM logarithmique

Un autre codeur de forme d'onde supérieur au PCM [1] est le DPCM (*differential pulse code modulation*). Dans ce type de codeur la corrélation du signal de parole est exploitée par la prédiction des séquences de parole et en formant un signal d'erreur entre la parole effective et la parole prédite qui pourra être codé à un moindre débit (figure 2 a et b).

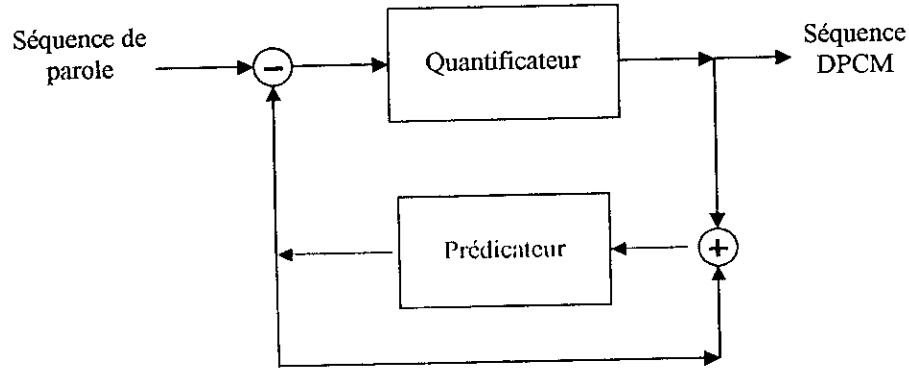


Figure 2 a : Codeur DPCM

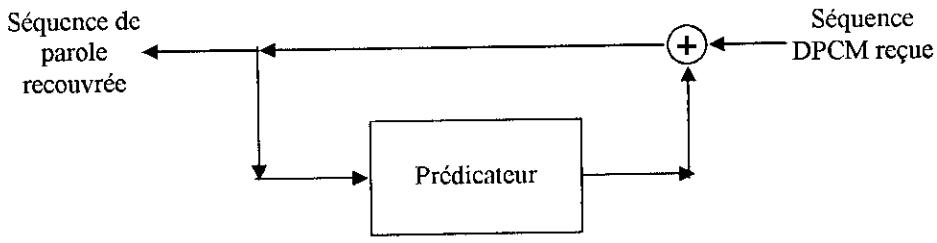


Figure 2 b : Décodeur DPCM

Le quantificateur utilisé dans le codeur DPCM peut être uniforme, non uniforme ou à pas adaptatif changeant selon les échantillons précédents [1]. Le quantificateur peut opérer sur un seul échantillon quand il est scalaire ou sur une séquence d'échantillons quand il est vectoriel.

Le prédicteur peut avoir des coefficients fixes ou adaptatifs. Quand le quantificateur ou le prédicteur est adaptatif le codeur est appelé ADPCM (*adaptative* PCM). Les codeurs ADPCM peuvent travailler à 24 kb/s avec une qualité de parole acceptable [1].

Le débit peut être réduit à 16 kb/s pour la même qualité de parole en utilisant le codage par transformé (*transform coding* TC). Dans le TC les échantillons de parole sont organisés en blocs et transformés dans le domaine fréquence [1].

Beaucoup de transformés peuvent être utilisées, mais la plus populaire et la DCT (*discret cosine transform*). Chaque coefficient de la transformé est codé avec un nombre de bits qui dépend de son importance perceptuelle. Certains coefficients considérés inconséquents sont négligés. Le nombre de bits assignés à chaque coefficient et la décision de négliger un coefficient peuvent dépendre de la statistique du bloc, dans ce cas le schéma est appelé ADCT (*adaptive DCT*).

Le codage en sous bandes SBC [1] (*sub band coding*) est un cas particulier de ADCT. La, le signal de parole est filtré en bandes puis un codeur adaptatif (tel que le APCM ou le ADPCM) est utilisé pour coder chaque bande. Les signaux codés en sous bandes sont multiplexés puis transmis. A la réception le démultiplexage des signaux codés en sous bandes est effectué suivi du décodage. Les signaux en sous bandes sont combinés pour donner le signal de parole.

II.2 CODAGE PAR PREDICTION LINEAIRE ET ANALYSE PAR SYNTHESE

Le codage de forme d'onde comme le SBC échoue dans la production de parole de haute qualité en dessous de 16 kb/s [1]. D'un autre coté, les vocodeurs LPC (*linear predictive coding*) peuvent travailler à 2 kb/s, mais la qualité de la parole synthétique n'est pas approprié pour la téléphonie commerciale.
La nécessité de produire une bonne qualité de parole à des débits inférieurs à 10 kb/s pour des applications où la largeur de bande des canaux est limitée, a conduit les chercheurs à penser à des algorithmes plus efficaces pour le codage prédictif.

La principale limitation des vocodeurs LPC est l'hypothèse que le signal de parole est voisé ou non voisé [1]. Alors, la source d'excitation du filtre tout pôle de synthèse est ou bien un train d'impulsions (pour la parole voisée) ou bien un bruit aléatoire (pour la parole non voisée). En fait, il y a plus que deux modes d'excitation du conduit vocale, et souvent ces modes sont mélangés. Même quand la forme d'onde de parole est voisée, c'est une grande simplification de considérer qu'il n'y a qu'un point d'excitation dans le *pitch*.

En 1982, Atal a proposé un nouveau modèle (figure 3.a) pour l'excitation appelé *multi-pulse excitation* MPE. Dans ce modèle, il n'est question ni de signal voisé/non-voisé ni de *pitch*. L'excitation est modélisée par un nombre d'impulsion (généralement 4 par 5 ms) dont les amplitudes et les position sont déterminées par la minimisation de l'erreur pondérée (selon un critère de perception) entre la parole originale et la parole synthétique.

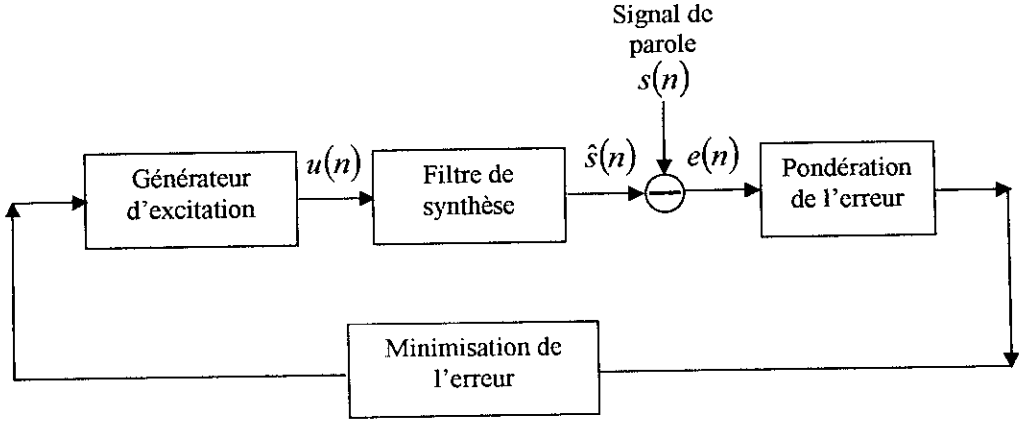


Figure 3.a : codage a
Analyse par synthèse

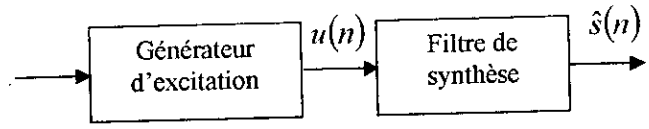


Figure 3 b : décodage

L'introduction de ce modèle a suscité un grand intérêt. Ce fut le premier d'une nouvelle génération de codeurs d'analyse par synthèse capables de produire de la parole de haute qualité à des débits autour de 10 kb/s.
Cette nouvelle génération de codeurs utilise le même filtre de synthèse tout pôle comme dans les codeurs LPC. Cependant le signal d'excitation est soigneusement optimisé et efficacement codé par les techniques de codage de forme d'onde.

Tous les codeurs d'analyse par synthèse partage la même structure (figure 3.a) de base où l'excitation est déterminée en minimisant l'erreur pondérée (selon un critère de perception) entre la parole originale et la parole synthétique [1]. Ils diffèrent dans la façon dont l'excitation est modélisée.

L'approche MPE suppose qu'initialement, ni les positions, ni les amplitudes des impulsions ne sont connues. Elles sont déterminées à l'intérieur de la boucle de minimisation impulsion par impulsion (une à la fois).

L'approche RPE (*regular pulse excitation*) suppose que les impulsions sont régulièrement espacées et ne calcule que les amplitudes en résolvant M ensembles de $M \times M$ équations (où M est le nombre d'impulsions).

Dans l'approche CELP (*code excited linear prediction*), le signal d'excitation est une entrée dans un très large *codebook* stochastique. La complexité de ces codeurs augmente avec la diminution du débit. Par exemple, CELP peut produire une bonne qualité de parole à des débits aussi bas que 4.8 kb/s au prix d'une très grande demande en calcul due à la recherche exhaustive dans le très grand codebook (généralement 1024 vecteurs) pour déterminer la séquence d'innovation optimale.

En conclusion les codeurs de parole se divisent en deux catégories : Les codeurs de forme d'onde comme le PCM, TC, SBC ... qui reproduisent fidélement la forme d'onde du signale de parole. Ces codeurs ont une bonne qualité de parole jusqu'à 16 kb/s. La deuxième catègorie est celle des codeurs par prédiction linéaire et analyse par synthése comme le MPE, RPE, CELP ...etc. Dans ces codeurs la production de la parole se fait selon un critére perceptuel et peut être de bonne qualité à bas débit (CELP 4,8 Kb/s).

Chapitre III

CODEUR MULTI-PULSE

Chapitre III

Codeur Multi Pulse

III.1 INTRODUCTION

Après avoir passé en revue quelques types de codeurs de parole, nous allons maintenant décrire en détail le schéma des codeurs d'analyse par synthèse (figure 3.a et 3.b) et le calcul de l'excitation dans le cas de l'algorithme MPE.

Le modèle consiste en trois parties essentielles [1] :

La première est le filtre de synthèse qui est un filtre tout pôle, variant dans le temps, utilisé pour la modélisation de l'enveloppe spectrale dans un intervalle de temps court de la forme d'onde de la parole. Il est souvent appelé 'filtre de corrélation de court terme' *(short term predictor)* parce que ces coefficients sont calculés par la prédiction d'un échantillon de parole à partir de quelques échantillons précédents (8 à 16).
Le filtre de synthèse peut aussi inclure un 'filtre de corrélation de long terme' LTP *(long term predictor)* en cascade avec le filtre de corrélation de court terme. Ce filtre modélise la structure fine du spectre de parole.

La seconde partie du modèle est le générateur d'excitation. Ce générateur produit la séquence d'excitation qui doit être injectée au filtre de synthèse pour reproduire la parole à la réception. L'excitation est optimisée en minimisant l'erreur (pondérée selon un critère de perception) entre la parole synthétique et la parole originale. Comme on le voit dans la figure (3.a) un décodeur est présent dans le codeur. La méthode d'analyse pour l'optimisation de l'excitation choisit la séquence d'excitation qui minimise l'erreur pondérée.

L'efficacité de la méthode d'analyse par synthèse vient de la procédure d'optimisation en boucle fermée, qui permet la représentation du résidu de prédiction avec un très bas débit tout en maintenant une haute qualité de parole.

Ceci explique la supériorité du codage par prédiction linéaire et analyse par synthèse par rapport à d'autres codeurs par prédiction qui ont une structure en boucle ouverte comme le *Residual Excited Linear Prediction coder* (RELP). Le point clé dans la structure en boucle ouverte est que le résidu est quantifié en minimisant l'erreur entre la parole synthétique et l'originale au lieu de minimiser l'erreur entre le résidu et sa version quantifiée comme dans la boucle ouverte.

La troisième partie du modèle est le critère de minimisation utilisé dans la minimisation de l'erreur. Le critère de minimisation le plus courant est l'erreur au sens des moindres carrés. Dans ce modèle, un critère subjectif de minimisation d'erreur est utilisé ou l'erreur *e(n)* est passé à travers un filtre de pondération qui va modifier le spectre du bruit de façon à concentrer la puissance sur les fréquences des formants du spectre de la parole ce qui a pour effet le masquage du bruit par le signal de parole.

La procédure de codage inclut deux étapes [1] :

Premièrement, les paramètres du filtre de synthèse sont déterminés (10 à 30 ms de parole) à l'extérieur de la boucle d'optimisation.
Deuxièmement, la séquence d'excitation optimale pour ce filtre est déterminée en minimisant le critère d'erreur. L'intervalle d'optimisation de l'excitation est de l'ordre de 4 à 7.5 ms qui est inférieur à la trame LPC (de calcul des paramètres du filtre de synthèse). C'est pour cela que la trame de parole est divisée en sous blocs ou sous trames, où l'excitation est déterminée individuellement pour chaque sous-trame.

Les paramètres quantifiés du filtre et l'excitation quantifiée sont envoyés au récepteur. La procédure de décodage est faite en passant l'excitation décodée à travers le filtre de synthèse pour reconstituer la parole.

III.2 LE PREDICTEUR COURT TERME

Le prédicteur court terme modélise l'enveloppe spectrale sur un court intervalle de temps. L'enveloppe spectrale d'un segment de parole de longueur L (L = nombre

d'échantillons d'une trame de parole) peut être approchée par la fonction de transfert d'un filtre tout pôle de la forme :

$$H(z)=\frac{1}{1-P(z)}=\frac{1}{1-\sum_{k=1}^{p}a_k z^{-k}} \qquad (1)$$

ou :

$$P(z)=\sum_{k=1}^{p}a_k z^{-k} \qquad (2)$$

Est le prédicteur court terme.

Les coefficients a_k sont calculés en utilisant la méthode de la prédiction linéaire (LP). L'ensemble des coefficients $\{a_k\}$ est appelé 'paramètres LPC' ou 'coefficients de prédiction' ou encore 'coefficients LP'. Le nombre des coefficients p est appelé ordre de prédiction.

L'idée dans l'analyse par prédiction linéaire [1] est qu'un échantillon de parole peut être approché par une combinaison linéaire d'échantillons passés (8 à16) :

$$\tilde{s}(n)=\sum_{k=1}^{p}a_k s(n-k) \qquad (3)$$

où *s(n)* est l'échantillon de parole et $\tilde{s}(n)$ l'échantillon prédit à un instant *n*. l'erreur de prédiction *e(n)* (ou résidu de prédiction) est définie par :

$$e(n)=s(n)-\tilde{s}(n)=s(n)-\sum_{k=1}^{p}a_k s(n-k) \qquad (4)$$

en prenant la transformation en *z* de l'équation (4) on obtient :

$$E(z)=S(z)\cdot A(z) \qquad (5)$$

où :

$$A(z)=1-\sum_{k=1}^{p}a_k z^{-k} \qquad (6)$$

A(z) est l'inverse de H(z) dans l'équation (1), c'est pourquoi on l'appel le filtre inverse.

A cause de la nature variante dans le temps de la parole, les coefficients de prédiction doivent être estimés sur de courts segments de paroles (10 à 20 ms). L'approche de base est de trouver un ensemble de coefficients qui minimise l'erreur de prédiction (au sens des moindres carrés) sur un court segment de parole. L'erreur moyenne de prédiction est définie par :

$$E = \sum_n e^2(n) = \sum_n \left[s(n) - \sum_{k=1}^{p} a_k s(n-k) \right]^2 \qquad (7)$$

Les coefficients qui minimisent E sont calculés par la méthode des autocorrélations grâce à l'algorithme de Wienner Levinson Durbin :

E (0) = R (0)

Pour i=1 à p faire

$$k_i = \left[R(i) - \sum_{j=1}^{i-1} a_j^{(i-1)} R(i-j) \right] / E(i-1)$$

$$a_i^{(i)} = k_i$$

Pour j = 1 à i-1 faire

$$a_j^{(i)} = a_j^{(i-1)} - k_i a_{i-j}^{(i-1)}$$

fin

$$E(i) = (1 - k_i^2) E(i-1) \qquad (8)$$

fin

avec :

$$R(i) = \sum_{n=i}^{L_a - 1} s(n) s(n-i) \qquad (i=0 \text{ à } p) \qquad (9)$$

$R(i)$ sont les autocorrélations du signal de parole $s(n)$, et L_a le nombre d'échantillons dans une trame d'analyse LPC.

la solution finale est donnée par :

$$a_j = a_j^{(p)} \qquad (j=1 \text{ à } p)$$

Les coefficients k_i sont appelés 'coefficients de réflexion', leurs valeurs sont comprises entre -1 et 1. C'est d'ailleurs une condition nécessaire et suffisante pour la stabilité du filtre $H(z)$.

A l'extérieur de l'intervalle $0 \le n \le L_a - 1$ les échantillons de parole sont considérés nuls, cependant cette troncation est équivalente à la création d'une grande augmentation de l'erreur de prédiction au début et à la fin du segment analysé. Ce problème est évité en utilisant une fenêtre telle que la fenêtre de Hamming :

$$w(n) = 0.54 - 0.46\cos(2\pi n/(L_a - 1)) \qquad 0 \le n \le L_a - 1 \qquad (10)$$

où L_a est la longueur de la trame LPC.

La longueur L_a de la fenêtre de Hamming est généralement choisie plus grande que la longueur de la trame de parole L. ceci donne un effet de lissage dans l'analyse LPC qui atténue les changements abrupts des coefficients LPC entre blocs d'analyse.

Les paramètres LPC représentent les coefficients du filtre LPC de synthèse $H(z) = 1/A(z)$. Leur quantification doit assurer la stabilité du filtre de synthèse. En d'autres termes, les pôles du filtre de synthèse doivent rester à l'intérieur du cercle unité ce qui est difficile si les $\{a_k\}$ sont quantifiés directement. C'est pourquoi il est nécessaire de transformer les paramètres LPC en un autre ensemble de paramètres dont la quantification assure la stabilité du filtre de synthèse. Les paramètres k_i ont été utilisés [1] à la place des a_k, en les transformant en coefficients qu'on appelle LAR_i (*log area ratios*) :

$$LAR_i = \log\frac{1-k_i}{1+k_i} \qquad (11)$$

III.3 LES LSP

Une autre transformation importante des paramètres LPC est l'ensemble des LSP (*line spectrum paires*). Les LSP sont les solutions non triviales des polynômes P et Q suivant :

$$P(z) = A(z) - z^{-(p+1)}A(z^{-1}) \tag{12}$$

$$Q(z) = A(z) + z^{-(p+1)}A(z^{-1}) \tag{13}$$

Les polynômes P et Q sont respectivement symétrique et antisymétrique et ont les propriétés suivantes :

1. Toutes leurs racines sont sur le cercle unité.
2. Les racines de P et Q alterne entre elles sur le cercle unité.
3. La stabilité du filtre $H(z)$ est facilement préservée [1] après quantification des racines de P et Q.

Comme les racines de P et Q sont sur le cercle unité elles sont données par $e^{j\omega_i}$ et pour un ordre de prédiction p paire on a :

$$P(z) = (1 - z^{-1}) \prod_{i=2,4...,p} (1 - 2\cos(2\pi f_i)z^{-1} + z^{-2}) \tag{14}$$

et

$$Q(z) = (1 + z^{-1}) \prod_{i=1,3...,p-1} (1 - 2\cos(2\pi f_i)z^{-1} + z^{-2}) \tag{15}$$

Les f_i $(0 < f_i < 1/2)$ ou bien les ω_i $(0 < \omega_i < \pi)$ sont appelées les LSF/LSP (*line spectrum frequencies/ line spectrum pairs*).

Les LSF ont une propriété importante qui est l'ordonnancement :

$$0 < f_1 < f_2 < \cdots < f_{p-1} < f_p < 1/2 \tag{16}$$

En plus de cette propriété les LSP/LSF possèdent d'autres caractéristiques [1] qui leur permettent d'être quantifiés plus efficacement par rapport au LAR_i :

1- Les LSF ont de meilleures propriétés statistiques.

2- La stabilité du filtre de synthèse est préservée plus facilement.

3- Entre trames adjacentes, Les paramètres LSP sont très corrélés. Ceci peut être exploité pour réduire le débit en utilisant une quantification prédictive. Ca permet aussi de mettre en œuvre des procédures d'interpolation très efficaces où les LSF ne sont transmis qu'à chaque trame impaire ce qui réduit considérablement leur débit.

III.3.1 CALCUL DES LSP

Dans la plupart des cas l'ordre de prédiction p est égal à 10. Dans ce qui suit les calculs [2] sont faits pour p=10.

Soit les deux polynômes F_1 et F_2 définis par :

$$F_1(z) = Q(z)/(1+z^{-1}) \tag{17}$$

$$F_2(z) = P(z)/(1-z^{-1}) \tag{18}$$

Chaque polynôme a 5 racines q_i conjuguées sur le cercle unité $e^{\pm j\omega_i}$:

$$F_1(z) = \prod_{i=1,3,\ldots,9} (1 - 2q_i z^{-1} + z^{-2}) \tag{19}$$

$$F_2(z) = \prod_{i=2,4,\ldots,10} (1 - 2q_i z^{-1} + z^{-2}) \tag{20}$$

où $qi = \cos(\omega_i)$.

Comme les polynômes F_1 et F_2 sont symétriques, seul cinq coefficients de chaque polynôme ont besoin d'être calculés. Les coefficients de ces polynômes sont calculés grâce à la relation récursive suivante :

$$f_1(i+1) = -(a_{i+1} + a_{10-i} - f_1(i)) \qquad i = 0,\ldots,4 \tag{21}$$

$$f_2(i+1) = -a_{i+1} + a_{10-i} + f_2(i) \qquad i = 0,\ldots,4 \tag{22}$$

Avec $f_1(0) = f_2(0) = 1.0$.

Les LSP sont obtenus en évaluant les polynômes F_1 et F_2 sur 60 points également espacés entre 0 et π, et en recherchant les changements de signe. Un changement de signe signifie la présence d'une racine, celle-ci est ensuite raffinée par quelques bipartitions (4 ou 5).

Une autre méthode consiste à utiliser les polynômes de Chebyshev pour évaluer F_1 et F_2. Dans ce cas les racines sont trouvées directement dans le domaine cosinus. F_1 et F_2 évalués à $z = e^{j\omega}$, peuvent être écrit comme suit :

$$F(\theta) = 2e^{-j5\omega}C(x) \tag{23}$$

Avec :

$$C(x) = T_5(x) + f(1)T_4(x) + f(2)T_3(x) + f(3)T_2(x) + f(4)T_1(x) + f(5)/2 \tag{24}$$

où $T_m(x) = \cos(m\omega)$ est le polynôme de Chebyshev d'ordre m.

Et $f(i), i = 1,...,5$ sont les coefficients de F_1 ou de F_2 des équations (21) et (22).

Le polynôme $C(x)$ est évalué à un certain point x par la relation récursive :

Pour k = 4 à 1 faire

$$b_k = 2xb_{k+1} - b_{k+2} + f(5-k)$$

fin

$$C(x) = xb_1 - b_2 + f(5)/2 \tag{25}$$

avec :

$b_5 = 1$ et $b_6 = 0$.

III.3.2 CONVERSION DES LSP EN PARAMETRES LPC

La transformation des LSP en paramètres LPC se fait [2] en recalculant les coefficients de $F_1(z)$ et de $F_2(z)$ à partir des paramètres q_i des équations (19) et (20) selon la relation récursive suivante :

Pour i = 1 à 5 faire

$$f_1(i) = -2q_{2i-1}f_1(i-1) + 2f_1(i-2)$$

pour j = i-1 à 1 faire

$$f_1(j) = f_1(j) - 2q_{2i-1}f_1(j-1) + f_1(j-2) \qquad (26)$$

fin

fin

avec :

$f_1(0) = 1$ et $f_1(-1) = 0$. Les coefficients $f_2(i)$ sont calculés de la même façon en remplaçant q_{2i-1} par q_{2i}.

Après avoir calculé les coefficients $f_1(i)$ et $f_2(i)$, les polynômes $F_1(z)$ et de $F_2(z)$ sont multipliés par $1+z^{-1}$ et $1-z^{-1}$ respectivement pour obtenir $Q(z)$ et $P(z)$:

$$f_1'(i) = f_1(i) + f_1(i-1) \qquad i = 1,...,5 \qquad (27)$$

$$f_2'(i) = f_2(i) - f_2(i-1) \qquad i = 1,...,5 \qquad (28)$$

Enfin les coefficients LP sont calculés à partir de $f_1'(i)$ et $f_2'(i)$:

$$a_i = \begin{cases} 0.5 \cdot (f_1'(i) + f_2'(i)) & pour \quad i = 1,...,5 \\ 0.5 \cdot (f_1'(11-i) - f_2'(11-i)) & pour \quad i = 6,...,10 \end{cases} \qquad (29)$$

III.4 LE PREDICTEUR LONG TERME

Le prédicteur long terme LTP (appelé plus communément *pitch predictor*) est utilisé [1] pour modéliser la structure fine de l'enveloppe spectrale du signal de parole. Le filtre inverse $A(z)$ permet d'enlever une partie de la redondance du signal de parole. Mais le résidu $r(n)$ de cette prédiction court terme sur p (généralement p=10) échantillons passés contient toujours un peu de périodicité. Cette périodicité est de l'ordre de 20 à 160 échantillons (50 à 400 Hz) (d'où le nom de prédicteur long terme).

L'utilisation du LTP va permettre d'enlever cette périodicité pour donner un résidu qui est presque un bruit blanc.

Le filtre de corrélation long terme est de la forme :

$$\frac{1}{P(z)} = \frac{1}{1 - P_l(z)} = \frac{1}{1 - \sum_{k=-m_1}^{m_2} G_k z^{-(\alpha+k)}}$$

où :

$P_l(z)$ est le LTP.

α la période du pitch.

En général, $m_1 = m_2 = 0$, dans ce cas le résidu LTP est donné par :

$$e(n) = r(n) - Gr(n-\alpha) \tag{30}$$

Le filtre $1/P(z)$ devient :

$$\frac{1}{P(z)} = \frac{1}{1 - Gz^{-\alpha}} \tag{31}$$

Le LTP n'est pas essentiel pour les codeurs à débit moyen (8 à 16 kb/s) comme le MPE-LPC et le PRE-LPC, mais il devient indispensable pour les codeurs de très faible débit comme le CELP.

III.5 LE FILTRE DE PONDERATION DE L'ERREUR

Les études faites sur l'audition [1] nous apprennent que le bruit dans les régions des formants serait partiellement ou totalement masqué par le signal de parole. D'où, une grande partie du bruit perçue dans un codeur vient des régions fréquentielles où le niveau du signal est faible. C'est pourquoi, pour réduire le bruit perçu, le spectre du bruit est modifié de sorte à diminuer l'énergie des composantes fréquentielles du bruit dans les régions inter-formants par rapport à celles des régions des formants.

Le filtre de pondération de l'erreur utilisé est de la forme :

$$W'(z) = A(z) \cdot W(z) = \frac{A(z)}{A(z/\gamma)} = \frac{1 - \sum_{k=1}^{p} a_k z^{-k}}{1 - \sum_{k=1}^{p} a_k \gamma^k z^{-k}} \qquad (32)$$

où : $0 < \gamma < 1$

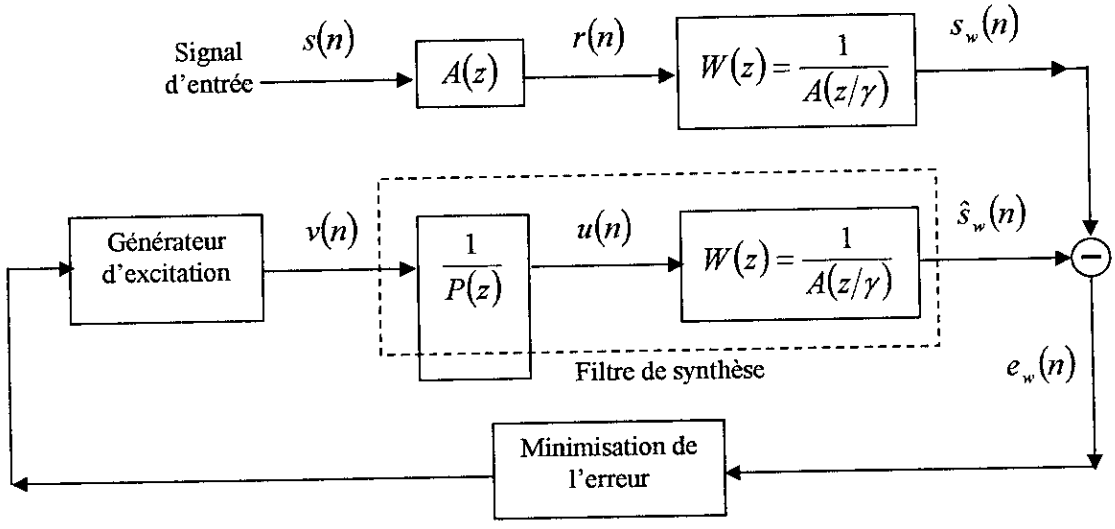


Figure 4 : Structure de base des codeurs par prédiction linéaire et analyse par synthèse.

γ est généralement choisi proche de 0.8.

$W(z) = \frac{1}{A(z/\gamma)}$ est appelé filtre de synthèse pondéré.

En utilisant le filtre de pondération de l'équation (32), et en pondérant le signal original et le signal synthétique avant leur soustraction, le schéma de la figure (3.a) peut être donné sous la
forme de la figure (4).

Dans ce qui suit, c'est ce modèle que nous allons utiliser.

III.6 L'EXCITATION

III.6.1 POSITION DU PROBLEME

L'excitation du filtre de synthèse est définie par un nombre M d'impulsions dont il faut calculer les amplitudes β_k et les positions m_k sur une trame d'excitation de N échantillons.

Comme nous l'avons mentionné auparavant, la trame de parole de L échantillons est divisée en trames d'excitation (généralement 4 sous trames) de N échantillons.
Le signal d'excitation $v(n)$ est défini par :

$$v(n) = \sum_{k=0}^{M-1} \beta_k \delta(n - m_k) \qquad n = 0,\ldots,N-1 \qquad (33)$$

avec :

$$\delta(n) = \begin{cases} 1 & si \quad n = 0 \\ 0 & si \quad n \neq 0 \end{cases}$$

Les amplitudes et les positions des impulsions sont calculées en minimisant l'erreur quadratique moyenne pondérée E_w entre le signal original et signal synthétisé.

$$E_w = \sum_{n=0}^{N-1} e_w^2(n) \qquad (34)$$

avec :
$e_w(n) = s_w(n) - \hat{s}_w(n)$ (Voir figure 4).

Si $h(n)$ est la réponse impulsionnelle du filtre de synthèse alors :

$$\hat{s}_w(n) = v(n) * h(n) = \sum_{i=0}^{n} v(i) h(n-i) \qquad (35)$$

En remplaçant $v(n)$ par sa valeur dans l'équation (35) on obtient :

$$\hat{s}_w(n) = \sum_{i=0}^{n} h(n-i) \sum_{k=0}^{M-1} \beta_k \delta(i - m_k)$$
$$= \sum_{k=0}^{M-1} \beta_k \sum_{i=0}^{n} h(n-i) \delta(i - m_k)$$
$$= \sum_{k=0}^{M-1} \beta_k h(n - m_k)$$

$$\hat{s}_w(n) = \sum_{k=0}^{M-1} \beta_k h(n - m_k) \qquad (36)$$

La formule (36) est très importante puisque c'est elle qu'on va utiliser pour la synthèse de la parole.

Reste maintenant à trouver la réponse impulsionnelle h du filtre de synthèse. Ceci ce fait aisément à partir de l'équation (32) qui nous donne la réponse fréquentielle du filtre de synthèse pondéré $W(z)$:

$$W(z) = \frac{1}{1 - \sum_{k=1}^{p} a_k^w z^{-k}} \Rightarrow W(z) - \sum_{k=1}^{p} a_k^w W(z) z^{-k} = 1 \Rightarrow W(z) = 1 + \sum_{k=1}^{p} a_k^w W(z) z^{-k}$$

où :

$$a_k^w = a_k \gamma^k \qquad (37)$$

Ce qui nous donne enfin :

$$h(n) = \delta(n) + \sum_{k=1}^{p} a_k^w h(n-k) \qquad (38)$$

En cas d'utilisation du filtre long terme $1/P(z)$ (pour le *pitch*) on trouve par le même calcul :

$$h(n) = \delta(n) + Gh(n-\alpha) + \sum_{k=1}^{p} a_k^w [h(n-k) - Gh(n-k-\alpha)] \qquad (39)$$

L'erreur E_w devient :

$$E_w = \sum_{n=0}^{N-1}\left(s(n) - \sum_{k=0}^{M-1} \beta_k h(n-m_k) \right)^2 \qquad (40)$$

Le problème maintenant et de trouver les amplitudes β_k et les positions m_k qui minimisent l'erreur E_w.

En posant $\partial E_w / \partial \beta_i = 0$ pour $i = 0,...,M-1$ on a :

$$\partial E_w / \partial \beta_i = 0 \Rightarrow -2\sum_{n=0}^{N-1}\left[s(n) - \sum_{k=0}^{M-1} \beta_k h(n-m_k) \right] h(n-m_i) = 0$$

$$\Rightarrow \sum_{n=0}^{N-1} s(n)h(n-m_i) = \sum_{n=0}^{N-1} h(n-m_i)\sum_{k=0}^{M-1} \beta_k (n-m_k)$$

$$\Rightarrow \sum_{n=0}^{N-1} s(n)h(n-m_i) = \sum_{k=0}^{M-1} \beta_k \sum_{n=0}^{N-1} h(n-m_k)h(n-m_i)$$

En posant :

$$\begin{cases} \phi(i,j) = \sum_{n=0}^{N-1} h(n-m_i)h(n-m_j) \\ \psi(i) = \sum_{n=0}^{N-1} s(n)h(n-i) \end{cases} \qquad (41)$$

on obtient :

$$\sum_{k=0}^{M-1} \beta_k \phi(m_i, m_k) = \psi(m_i) \qquad i = 0,...,M-1 \qquad (42)$$

Donc pour trouver l'excitation optimale il nous faut résoudre ce système de M équations à $2 \times M$ variables.

La solution du système d'équations (42) n'est pas évidente puisqu'on doit la trouver pour toutes les combinaisons possibles des positions des impulsions ($0 \le m_i < N \; i=0,...,M-1$) pour choisir celle qui minimise l'erreur.

Or le nombre de combinaisons possibles pour des valeurs typiques de $N=40$ et $M=4$ est de 91390. Ceci montre la très grande complexité de la recherche d'une solution optimale.

Dans ce qui suit nous allons montrer une méthode [1] sous optimale de solution du système d'équations (42). Cette méthode s'appelle l'approche *multi-pulse*.

III.6.2 L'APPROCHE MULTI PULSE

Nous allons d'abord, trouver une expression pour l'erreur minimale à partir des équations (40) (41) et (42) :

$$
\begin{aligned}
E_w &= \sum_{n=0}^{N-1}\left(s(n) - \sum_{k=0}^{M-1} \beta_k h(n-m_k) \right)^2 \\
&= \sum_{n=0}^{N-1} s^2(n) - 2\sum_{n=0}^{N-1} s(n) \sum_{k=0}^{M-1} \beta_k h(n-m_k) + \sum_{n=0}^{N-1}\left[\sum_{k=0}^{M-1} \beta_k h(n-m_k) \right]^2 \\
&= \sum_{n=0}^{N-1} s^2(n) - 2\sum_{k=0}^{M-1} \beta_k \sum_{n=0}^{N-1} s(n) h(n-m_k) + \sum_{n=0}^{N-1}\sum_{i=0}^{M-1}\sum_{k=0}^{M-1} \beta_i \beta_k h(n-m_i) h(n-m_k) \\
&= \sum_{n=0}^{N-1} s^2(n) - 2\sum_{k=0}^{M-1} \beta_k \psi(m_k) + \sum_{i=0}^{M-1}\sum_{k=0}^{M-1} \beta_i \beta_k \sum_{n=0}^{N-1} h(n-m_i) h(n-m_k) \\
&= \sum_{n=0}^{N-1} s^2(n) - 2\sum_{k=0}^{M-1} \beta_k \psi(m_k) + \sum_{i=0}^{M-1}\sum_{k=0}^{M-1} \beta_i \beta_k \phi(m_i, m_k)
\end{aligned}
$$

or selon l'équation (42) :

$$
\sum_{k=0}^{M-1} \beta_k \psi(m_k) = \sum_{k=0}^{M-1} \beta_k \sum_{i=0}^{M-1} \beta_i \phi(m_k, m_i) = \sum_{k=0}^{M-1}\sum_{i=0}^{M-1} \beta_i \beta_k \phi(m_k, m_i)
$$

D'où l'expression de l'erreur minimale :

$$
E_{\min} = \sum_{n=0}^{N-1} s^2(n) - \sum_{k=0}^{M-1} \beta_k \psi(m_k) \qquad (43)
$$

L'approche multi-pulse détermine une seule impulsion à la fois dans un processus à M étapes. A chaque étape j, une nouvelle impulsion est déterminée de façon à diminuer l'erreur d'une excitation à j-1 impulsions.

Au début, nous supposant qu'il n'y à qu'une seule impulsion. L'équation (42) devient :

$$\beta_0 = \frac{\psi(m_0)}{\phi(m_0, m_0)} \tag{44}$$

Et l'expression de l'erreur minimale devient :

$$E_{\min} = \sum_{n=0}^{N-1} s^2(n) - \frac{\psi^2(m_0)}{\phi(m_0, m_0)} \tag{45}$$

Donc pour trouver la première impulsion il faut trouver la valeur de m_0 qui minimise l'erreur dans (45), ce qui est équivalent à maximiser le second terme du second membre de l'équation (45) :

$$m_0 = \max_m \left(\frac{\psi(m)}{\phi(m,m)} \right)^2 \qquad pour \qquad m = 0,\ldots,N-1 \tag{46}$$

Et β_0 est donné par l'équation (44).

Introduisant maintenant, une nouvelle impulsion. L'erreur (équation 40) est donnée par :

$$E_w^{(1)} = \sum_{n=0}^{N-1} \left[s(n) - \beta_0 h(n-m_0) - \beta_1 h(n-m_1) \right]^2$$

$$= \sum_{n=0}^{N-1} \left[s^{(1)}(n) - \beta_1 h(n-m_1) \right]^2$$

où : $$s^{(1)}(n) = s(n) - \beta_0 h(n-m_0)$$

La minimisation de $E_w^{(1)}$ par rapport à β_1 ($\partial E_w^{(1)}/\partial\beta_1 = 0$) nous donne des équations similaires à (44) et (45) :

$$\beta_1 = \frac{\psi^{(1)}(m_1)}{\phi(m_1, m_1)} \tag{47}$$

Et :

$$E_{\min}^{(1)} = \sum_{n=0}^{N-1}\left[s^{(1)}(n)\right]^2 - \left[\frac{\psi^{(1)}(m_1)}{\phi(m_1, m_1)}\right]^2 \tag{48}$$

Avec :

$$\begin{aligned}\psi^{(1)}(i) &= \sum_{n=0}^{N-1} s^{(1)}(n)h(n-i) = \sum_{n=0}^{N-1}\left[s(n) - \beta_0 h(n-m_0)\right]h(n-i) \\ &= \psi(i) - \beta_0\phi(m_0, i)\end{aligned} \tag{49}$$

On en déduit alors que :

$$m_1 = \max_m\left(\frac{\psi^{(1)}(m)}{\phi(m,m)}\right)^2 \qquad pour \qquad m = 0,\ldots,N-1 \tag{50}$$

On voit donc que la deuxième impulsion se calcule comme la première sauf qu'il faut mettre à jours ψ en enlevant l'effet de l'impulsion précédente selon l'équation (49). Et ainsi de suite, pour le reste des impulsions.

En résumé, l'algorithme MPE se déroule comme suit :

$$\psi^{(0)}(i) = \psi(i) \qquad pour\ i = 0,...,N-1$$

Pour j = 0 à M-1 faire :

$$\psi^{(j)}(m_i) = 0 \qquad pour\ i = 0,...,j-1$$

$$m_j = \max_m \left(\frac{\psi^{(j)}(m)}{\phi(m,m)} \right)^2 \qquad pour \qquad m = 0,...,N-1$$

$$\beta_j = \frac{\psi^{(j)}(m_j)}{\phi(m_j,m_j)}$$

$$\psi^{(j+1)}(i) = \psi^{(j)}(i) - \beta_j \phi(m_j,i) \qquad pour\ i = 0,...,N-1$$

fin (51)

La troisième ligne est nécessaire pour ne pas avoir plusieurs impulsions dans la même position.

III.6.3 MODIFICATION DE L'ALGORITHME MPE

La limitation de la recherche des impulsions sur une sous trame induit une distorsion plus grande à la fin que dans le reste de la sous trame [1]. Pour palier à cela, la recherche se fait sur une sous trame rallongée de quelques échantillons (généralement 10), mais la synthèse se fait juste sur une sous trame. Cette nouvelle sous trame s'appelle la 'trame d'excitation'. Le choix de prendre une trame d'excitation plus grande que la sous trame de parole va dépendre de la qualité de parole voulue et du débit souhaité.

Malgré la simplification qu'apporte l'algorithme MPE, la complexité reste assez grande du fait de la grande demande en calcul qu'exige le calcul des $\phi(i,j)$. Une autre simplification [1] de l'algorithme MPE consiste à utiliser la méthode des autocorrélations :

$$\phi(i,j) = \phi(|i-j|) = \sum_{n=|i-j|}^{N-1} h(n)h(n-|i-j|) \qquad (52)$$

En utilisant la méthode des autocorrélations les termes à maximiser dans l'algorithme (51) deviennent :

$$m_j = \max_m \left(\frac{\psi^{(j)}(m)}{\phi(0)} \right)^2 \tag{53}$$

Ce qui est équivalent à :

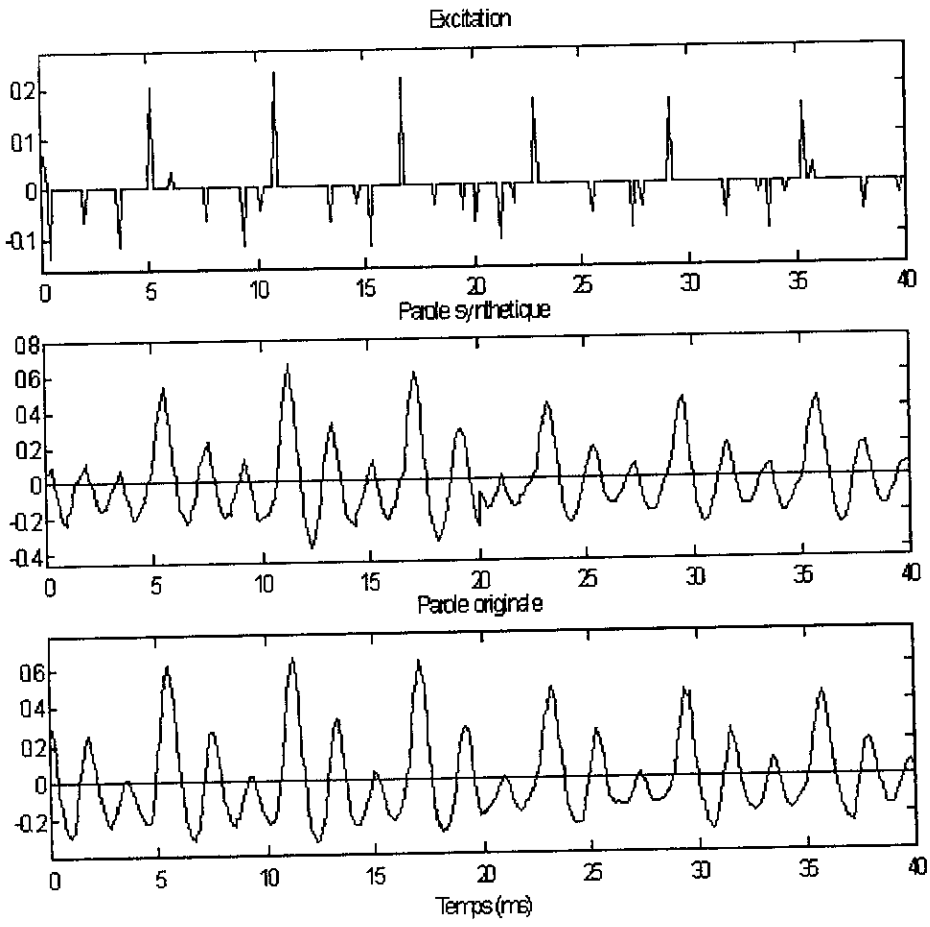
$$m_j = \max_m \left| \psi^{|j|}(m) \right| \tag{54}$$

Et les β_j deviennent :

$$\beta_j = \frac{\psi^{(j)}(m_j)}{\phi(0)} \tag{55}$$

Ainsi le nombre de $\phi(i,j)$ à calculer se limite à N (N longueur de la trame d'excitation) au lieu de $N \times N/2$. Les $\phi(i,j)$ ne sont calculés qu'une seule fois par trame de parole et il n'est besoin de faire qu'une seule division par trame de parole ($1/\phi(0)$) au lieu de $N \times M \times SF$ (SF est le nombre de sous trames). Sachant que les DSP de TI n'ont pas d'instructions de divisions [4] [7], elles sont implémentées par programme [6], et dans le cas du TMS320C30, une division en virgule flottante prend 40 cycles machine au lieu d'un comme pour les autres instructions (multiplications, additions, soustractions,...).

La figure ci-dessous montre le signal d'excitation calculé par l'algorithme MPE modifié pour deux trames de parole. On voit que la parole synthétisée ressemble à la parole originale dans ses variations.



Calcul de l'excitation par l'algorithme MPE

(Cette figure a été obtenue par simulation sur PC.)

III.7 LA QUANTIFICATION

Après le calcul des LSP et de l'excitation, la dernière étape est la quantification.

Pour faire la quantification nous avons utilisé un programme (de notre conception) de quantification scalaire basé sur l'algorithme de Loyd.

Le programme commence par un *codebook* distribué uniformément sur l'intervalle d'étalement des données qu'il améliore à chaque itération en minimisant l'erreur totale de quantification des données.

La base de données a été créée à partir du codage de 10732 trames de parole (plus de 3 mn et demi de parole).

III.7.1 QUANTIFICATION DES LSP

Nous avons quantifié les LSP sur 32 bits en codant la différence pour certains LSP, comme le montre le tableau suivant :

Après le décodage, la stabilité du filtre de synthèse est préservée [2] par les précautions suivantes :

-1 si $LSP_1 > 0.999987$ alors $LSP_1 = 0.999987$.

-2 si $LSP_i > LSP_{i-1} - 7.5 \cdot 10^{-4}$ alors $LSP_i = LSP_{i-1} - 7.5 \cdot 10^{-4}$.

-3 si $LSP_{10} < -0.999987$ alors $LSP_{10} = -0.999987$.

Paramètres	Nombre de bits	Erreur moyenne
LSP1	3	0.21%
LSP1-LSP2	3	0.44%
LSP2-LSP3	3	0.84%
LSP3-LSP4	3	1.12%
LSP4-LSP5	4	0.82%
LSP5-LSP6	4	0.71%
LSP7	3	1.33%
LSP8	3	1.11%
LSP9	3	0.83%
LSP10	3	0.53%

Quantification des LSP

III.7.2 QUANTIFICATION DE L'EXCITATION

Pour une trame d'excitation de 40 échantillons, et à partir de quatre impulsions par sous trame, la différence entre les positions des impulsions n'excède pratiquement jamais 31 à condition de les ordonner par ordre croissant de la position. Ce qui nous permet de les coder par 5 bits.
Les amplitudes des impulsions sont codées sur 3 bits (le codebook a été obtenu avec 0.78 % d'erreur relative moyenne). Ce qui fait en tous 8 bits par impulsion.

Avec une trame de parole de 20 ms (160 échantillons) et des sous trames de 5 ms (40 échantillons) le débit du codeur par trame est de 160 bits ($32+4\times4\times8$). Le débit total du codeur est de 8kb/s.

En conclusion, Le modèle multi-pulse consiste en trois parties : Le filtre de synthèse qui est utilise pour donner forme au signal d'excitation, le générateur d'excitation qui calcul la séquence d'excitation optimale et le critère de minimisation qui permet de se concentrer que sur l'élimination du bruit perceptible et ainsi obtenir une meilleur qualité a bas débit.

Chapitre IV

L'IMPLEMENTATION

Chapitre IV

L'implémentation

IV.1 LE MATERIEL

Le matériel dont nous disposons est un module d'évaluation (EVM) de TMS320C30 de TI (Texas Instruments). Ce module est un système de développement qui, connecté à une entrée et une sortie analogique (comme un microphone et un haut-parleur) devient un outil de traitement du signal.

IV.1.1 L'EVM

L'EVM est une carte 8 bits compatible IBM PC/AT qui comporte le DSP TMS320C30 de TI cadencé à 30 MHZ, un convertisseur AN/NA TLC32044 (AIC) et 16K X 32 de SRAM (figure 5).

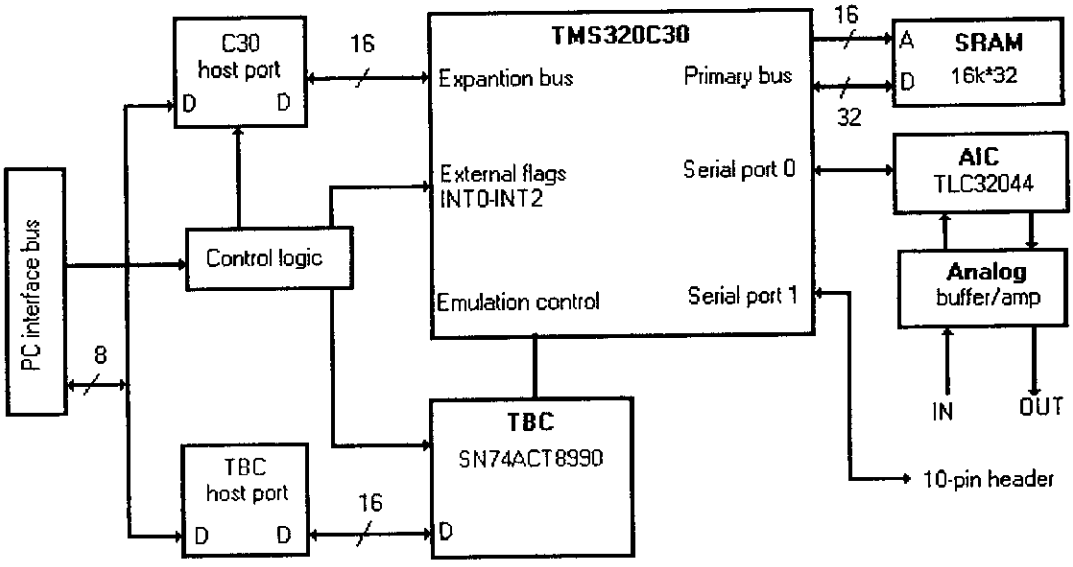


Figure 5 : schéma bloc de l'EVM

(Pour une description détaillée voir [3].)

IV.1.2 LE TMS320C30

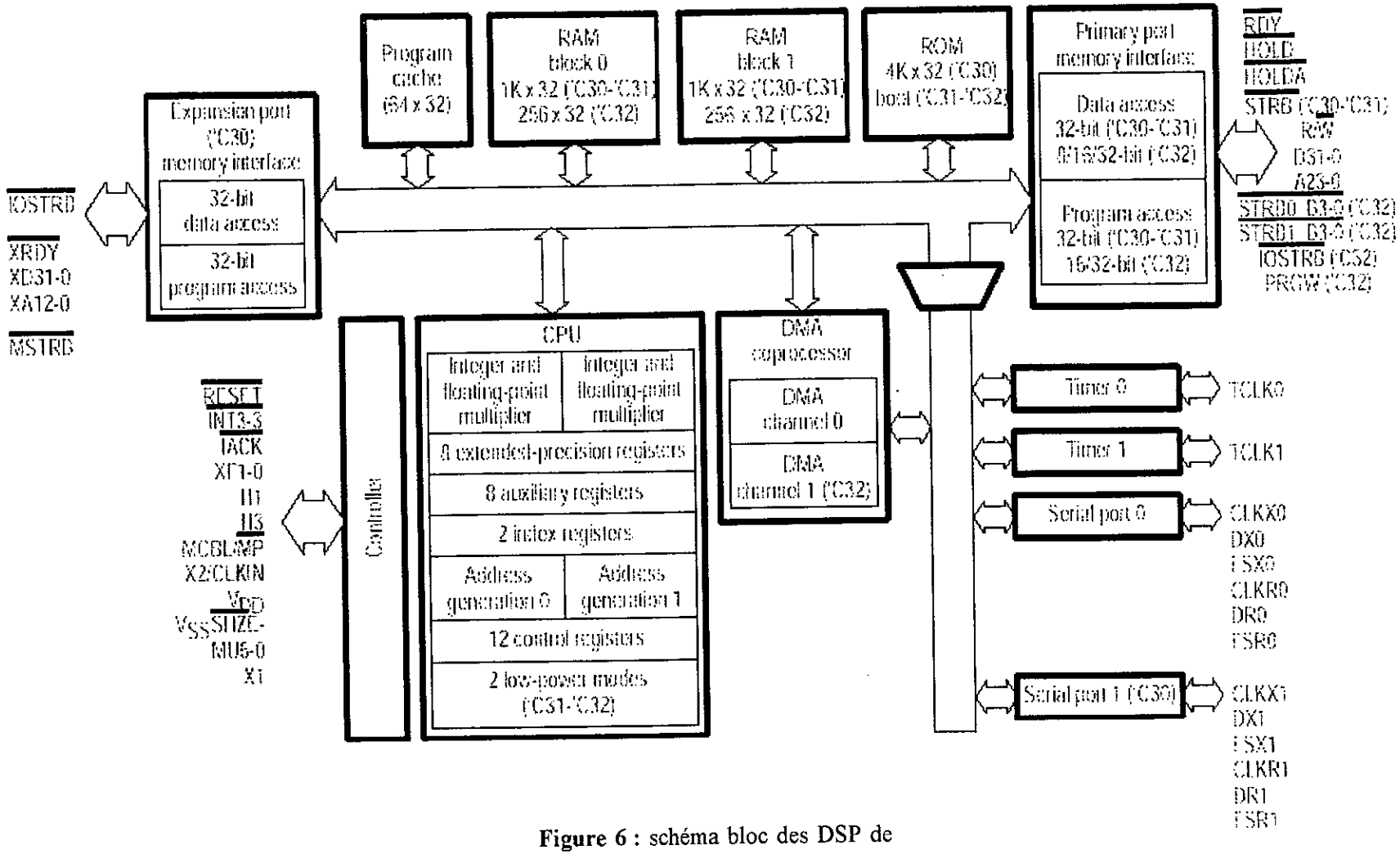


Figure 6 : schéma bloc des DSP de la famille TMS320C3X

Le TMS320C30 est un DSP à virgule flottante (32 ou 40 bits) avec une large mémoire interne (64 de cache, 2k de RAM et 4k de ROM). Il est doté de plusieurs périphériques qui le rendent indépendant de toute logique externe (*zero glue logic*). Ces périphériques sont le contrôleur DMA qui peut nous servir pour le transfert de données entre le PC haut est le DSP, deux ports série que nous allons utiliser (port 0) pour le transfert d'échantillons entre le DSP et l'AIC (figure 5) et deux *timers* dont l'un (*timer* 0) va fournir une base de temps pour l'AIC.

(Pour une description détaillée voir [4].)

IV-1.3 L'AIC

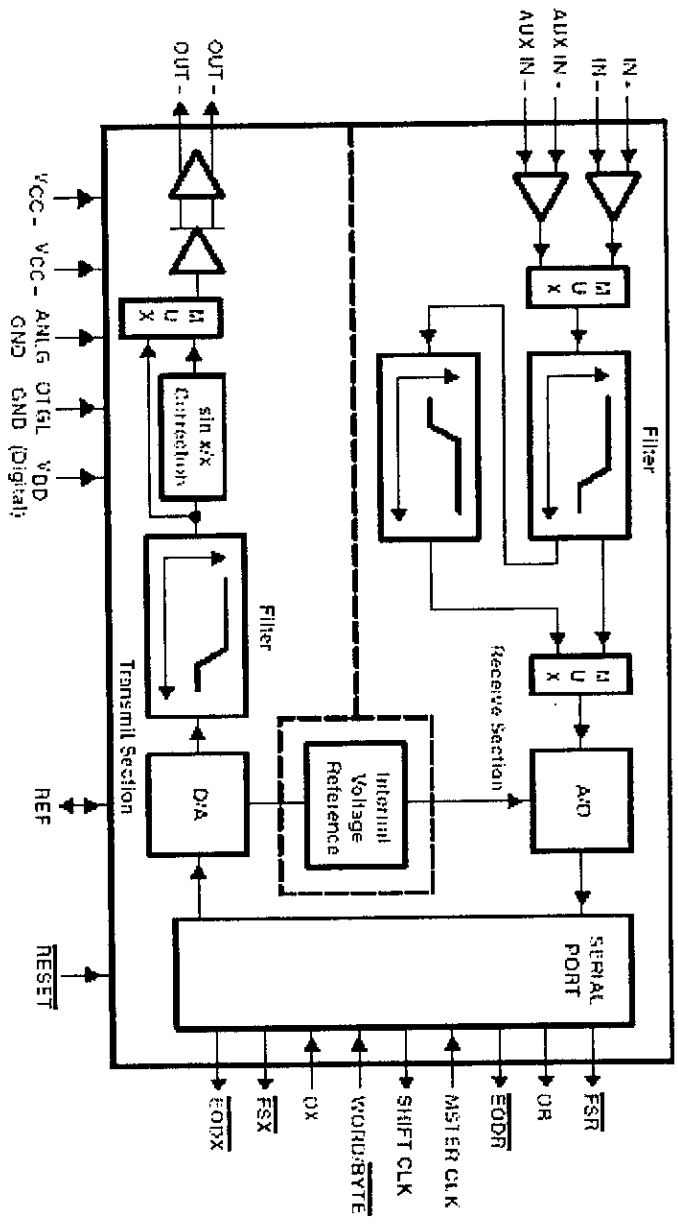


Figure 7 : schéma bloc du TLC32044

Le TLC32044 est un *voice-band analogue interface controller* (AIC) avec un filtre passe-bande anti-recouvrement en entrée et un filtre de sortie de reconstitution, tous les deux programmables, 14 bits de résolution en conversion AN et NA, format en complément à deux, fréquence d'échantillonnage variable jusqu'à 19200 HZ. Il est doté d'un port série qui permet la liaison avec les DSP de la série TMS320.

(Pour une description détaillée voir [5].)

IV.2 LE SOFT

Le langage de programmation utilisé est le C. Pour cela nous disposons du compilateur C compatible ANSI de TI pour les DSP des familles C3X et C4X.
Le compilateur est livré avec un programme shell 'cl30.exe' qui permet l'exécution de toutes les étapes de génération d'un fichier exécutable en une seule ligne de commande.

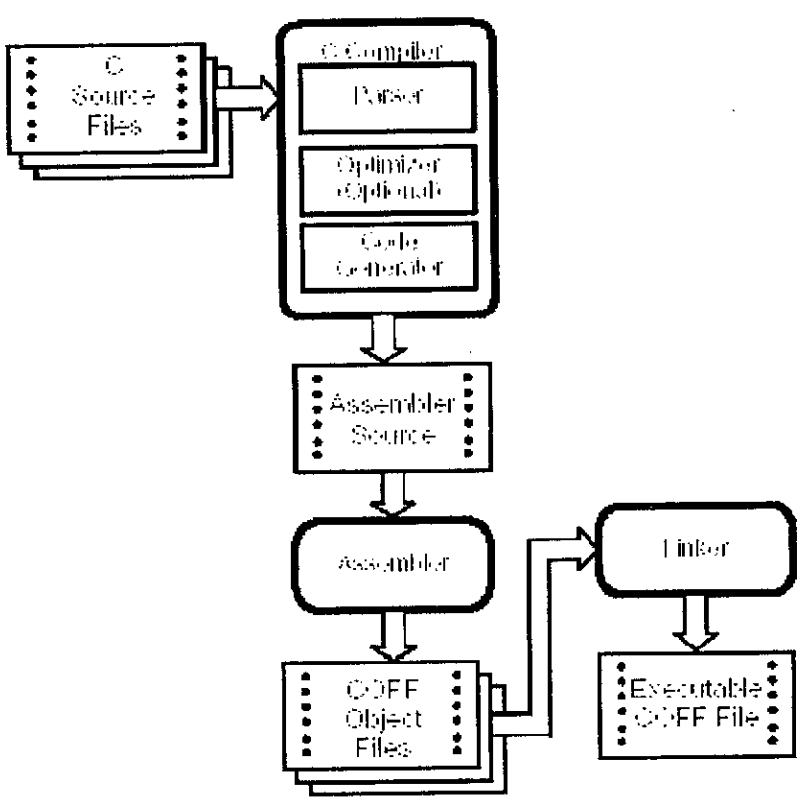


Figure 8 : étapes pour la génération d'un fichier exécutable

La compilation se fait par une ligne de commande de la forme :

cl30 [–options] [noms fichiers] [**–z** [options_link] [fichier objets]]

- option : pour spécifier la façon dont les fichiers d'entrées doivent être compilés.

noms fichiers : les noms des fichiers C, assembleur ou objet à compiler.

-z : est l'option pour l'édition de lien (*link*).

options_link : les options pour le contrôle de l'édition de lien.

fichiers objets : noms des fichiers objets que le compilateur créera.

Ex : cl30 –o –g source.c

Compilation du fichier 'source.c' avec optimisation (-o) et génération de directives (-g) pour le debuggage.

(Pour une description détaillée voir [6].)

IV.3 LA PROGRAMMATION

Dans ce qui suit nous allons entrer dans la partie pratique du travail où nous verrons concrètement la réalisation de toute la théorie exposée précédemment.

IV.3.1 ORGANISATION DES TRAMES DE PAROLE

Nous allons tout d'abord, voir le processus d'acquisition et de restitution de la parole.

Les échantillons sont obtenus à partir de l'AIC qui les transmet au DSP via le port série 0 (figure 9). La réception d'un échantillon par le port série déclenche une interruption, l'exécution du code associé à cette interruption permet de stocker le nouvel échantillon dans le buffer d'entrée ('aic_in') et la sortie sur l'AIC d'un échantillon du buffer de sortie ('aic_out')

Des que le buffer 'aic_in' est rempli (trame complète), la variable d'index 'sample' est remise à zéro pour remplir une nouvelle trame de parole et sortir la nouvelle trame de parole reconstituée.

(voir programme en annexe)

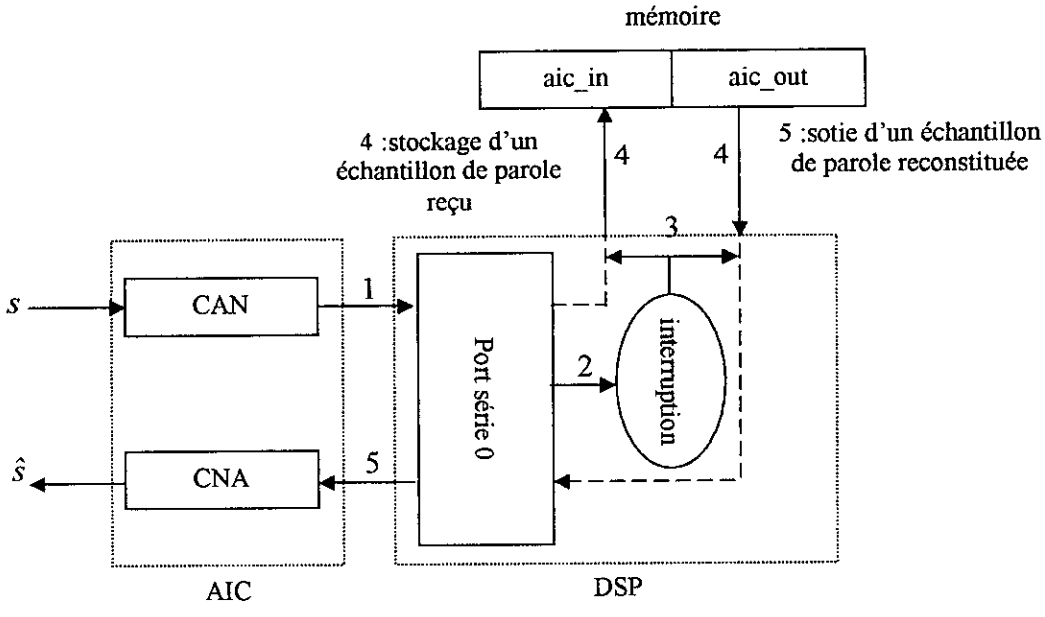


Figure 9 : schéma bloc de l'acquisition et de la restitution des échantillons de parole

Une fois la nouvelle trame remplie, le contenu de 'aic_in' est transféré dans le buffer de traitement d'entrée 's_in', et le contenu du buffer de traitement de sortie 's_out' dans le buffer de sortie 'aic_out' pour la restitution. Ceci ce fait par simple rotation de pointeur que nous verrons plus bas (figure 10 et 14).

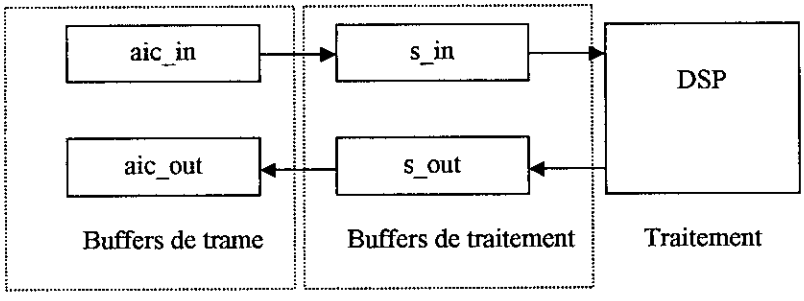


Figure 10 : rotation des buffers

Comme nous l'avons vu au chapitre II, les échantillons de parole sont organisés en trames pour le traitement. Il y a la trame d'analyse LPC pour le calcul des coefficients de prédiction et la trame de parole qui est divisée en quatre sous trames.

A chaque sous trame nous calculons le signal d'excitation qu'il faut pour la reproduction de la parole (figure 11) .

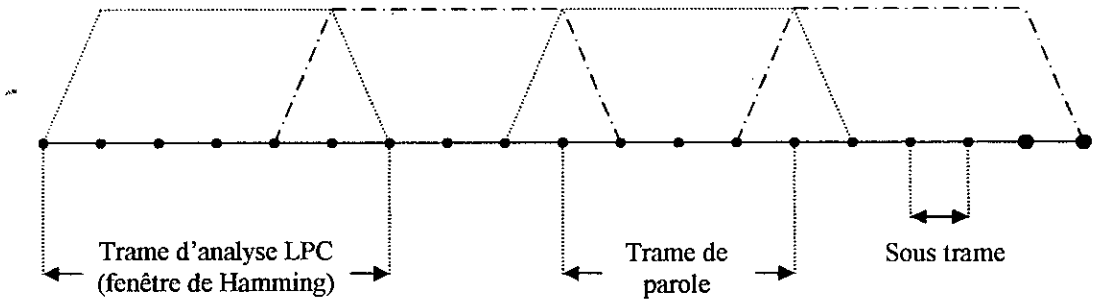


Figure 11 : organisation des différentes trames

A la figure (12) on voit la progression d'une trame tout au long du processus de codage. Au début, une trame 'i' est en entrée (remplissage de 'aic_in') tandis que le DSP est en train de coder la trame précédente 'i-1'. Une fois le buffer d'entrée complètement remplit, la trame 'i' passe à la phase de codage alors que la trame 'i-1' (reconstituée précédemment à partir des ses paramètres de codage calculés par le DSP) est sortie sur l'interface analogique (AIC).

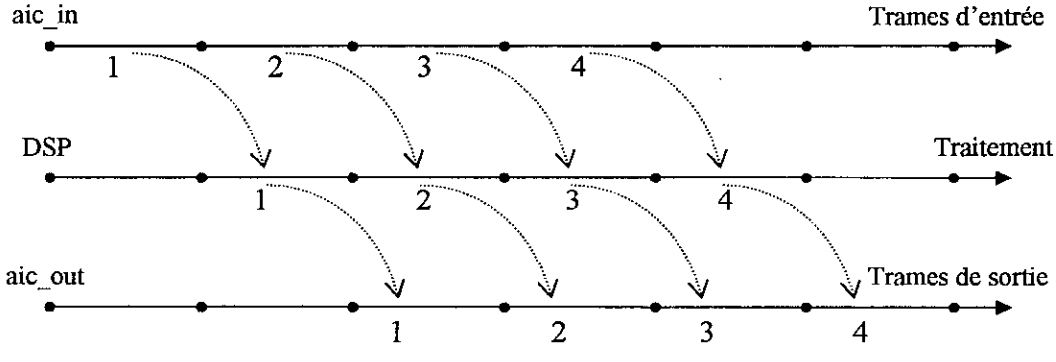


Figure 12 : enchaînement des trames

Le buffer 's_in' (trame de traitement d'entrée) est de longueur égale à celle de la trame d'analyse LPC, or cette dernière est composée d'une trame de parole plus une sous trame future et une sous trame passée !.

En réalité on évite le problème de la sous trame future en décalant la nouvelle trame d'entrée d'une sous trame en avant, ce qui fait que la trame de parole devient composée d'une sous trame passée et de trois sous trames actuelles (figure 13).

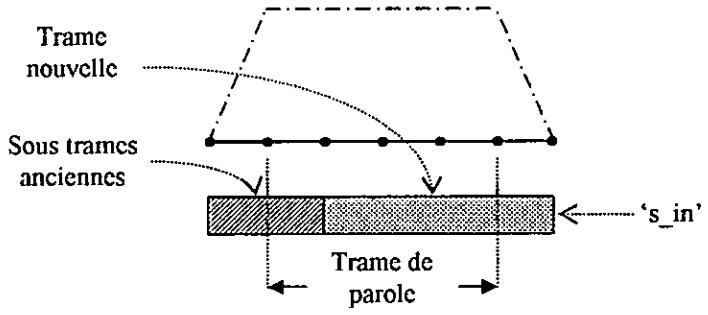


Figure 13 : structure du buffer d'entrée

La figure (14) résume tout ce qui concerne l'organisation des différentes trames, le recouvrement des fenêtres d'analyses et la rotation des buffers.

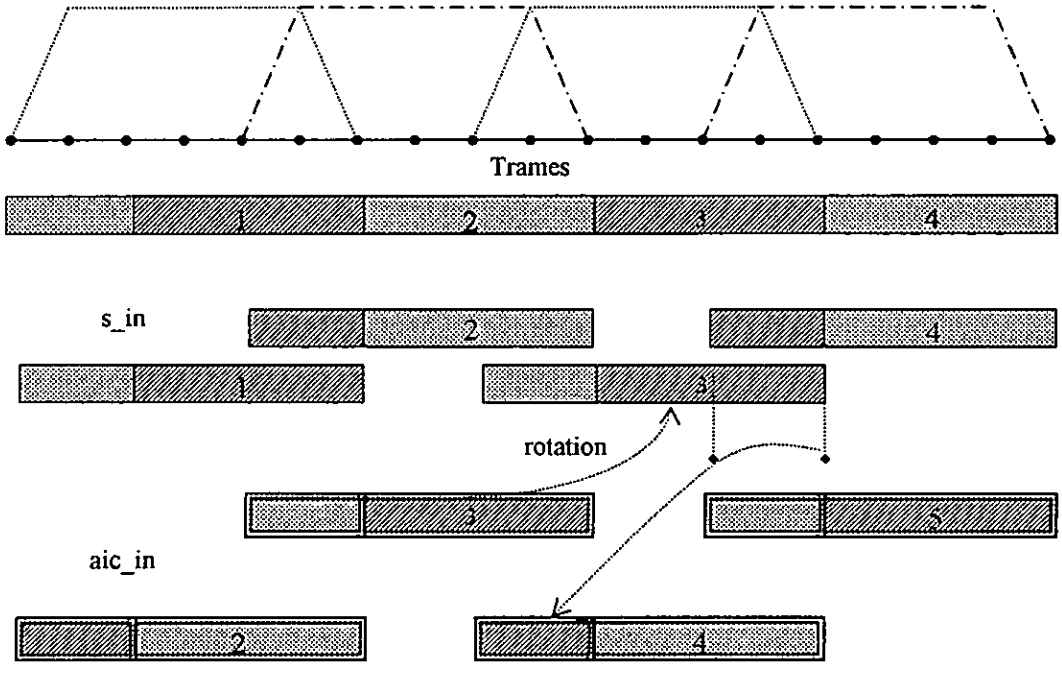


Figure 14 : recouvrement des fenêtres d'analyse et rotation des buffers d'entrée

IV.3.2 APERÇU DU PROGRAMME PRINCIPAL

```
void main(void)
{
        int n,sf,g,i,j;
        double c,f;

   init_evm();          /* INITIALIZE TMS320C30 AND EVM                */
   init_arrays();       /* INITIALIZE MPE ARRAYS                       */
   init_aic();          /* INITIALIZE TLC32044 AIC COMMUNICATIONS */

   for(;;)
   {
        wait_frame();        /* WAIT FOR FULL FRAME OF DATA          */
        /*================================================*/
        /*                                                */
        /*                  Speech processing             */
        /*                                                */
        /*================================================*/
   }
}
```

Figure 15 : squelette du programme principal

Le programme commence par la déclaration des variables qui vont être utilisées dans les différentes boucles.

Il se poursuit par l'appel de la fonction d'initialisation du module d'évaluation 'init_evm', de la fonction d'initialisation des variables globales utilisées pour le codage 'init_arrays', et de la fonction d'initialisation de l'AIC 'init_aic'. L'AIC est programmé pour fonctionner à une fréquence de 8KHZ.

Après l'initialisation, le traitement de la parole peut commencer. A l'intérieur de la boucle sans fin, on attend à chaque fois que la nouvelle trame soit complètement remplie pour ensuite exécuter les routines de traitement.

Ce squelette de programme est général et peut être utilisé comme squelette pour n'importe qu'elle application de traitement de la parole (ou autre), si celle-ci organise les échantillons par blocs.

Les fonctions 'init_aic' et 'init_evm' sont fournies par TI. Elles n'ont pas besoin d'être modifiées.

```
/*=====================================================*/
/*      WAIT_FRAME()                                   */
/*      WAIT FOR A NEW FRAME OF DATA SWAP ACQUISITION  */
/*      AND PROCESSING ARRAYS                          */
/*=====================================================*/
void wait_frame(void)
{
   double *ptr;                    /* SWAPPING VARIABLE          */

   while(sample);                  /* WAIT FOR A NEW FRAME       */

   ptr    = s_out;                 /* SWAP OUTPUT ARRAYS         */
   s_out  = aic_out;
   aic_out = ptr;

   ptr    = aic_in;                /* SWAP INPUT ARRAYS          */
   aic_in  = s_in;
   s_in    = ptr;

/* COPY OVERLAP BETWEEN WINDOWS                                 */
   memcpy(aic_in ,s_in + F , OVERLAP);
}
```

Figure 16 : fonction 'wait_frame'

La fonction 'wait_frame' (figure 16) réalise la temporisation pour le remplissage de la trame (remplissage de 'aic_in'). Pour cela, elle teste la variable d'index 'sample' qui à son passage à zéro indique la disponibilité d'une nouvelle trame. Puis elle fait la rotation (figure 10) des buffers de sortie et des buffers d'entrée. La rotation des buffers d'entrée nécessite la copie des deux dernières sous trames (de 's_in') dans 'aic_in' (figure 13 et 14). Ce qui est fait par la fonction C 'memcpy'.

La synchronisation du programme se fait par l'interruption du port série 0. C'est elle qui nous informe de la disponibilité d'un nouvel échantillon, de l'instant où il faut sortir un échantillon de parole reconstituée sur l'AIC et quand une nouvelle trame est prête. L'utilisation du mode interruptible est indispensable pour le travail en temps réel, car elle permet d'éviter le sondage pour faire l'acquisition des données. En effet, de la technique du sondage à la technique par interruption, le

temps de traitement d'une trame passe de T_e (0.125 ms) à la durée d'une trame de parole (20 ms) ! .

```
/*=========================================================*/
/* C_INT05()                                               */
/* SERIAL PORT 0 TRANSMIT INTERRUPT SERVICE ROUTINE        */
/* 1. IF SECONDARY TRANSMISSION SEND AIC COMMAND WORD      */
/* 2. OTHERWISE IF COMMAND SEND REQUESTED SETUP FOR        */
/*    SECONDARY                                            */
/* TRANSMISSION ON NEXT INTERRUPT                          */
/* 3. OTHERWISE WRITE OUT OUTPUT DATA AND READ IN INPUT    */
/*    DATA                                                 */
/* 4. RESET SAMPLE INDEX IF FRAME IS FULL                  */
/*=========================================================*/
void c_int05(void)
{
   if (secondary_transmit)
   {
        serial_port[0][X_DATA] = aic_secondary;
        secondary_transmit     = OFF;
   }
   else if (send_command)
   {
        serial_port[0][X_DATA] = 3;
        secondary_transmit     = ON;
        send_command           = OFF;
   }
   else
   {
        serial_port[0][X_DATA]    = (int) aic_out[sample]  << 2;
        aic_in[sample+OVERLAP]=serial_port[0][R_DATA]<<16>>18;
        if (++sample == F) sample = 0;
   }
}
```

Figure 17 : routine d'interruption du port série0

La routine d'interruption (*interrupt service routine*) est également fournie par TI. Il n'est nécessaire de modifier que le dernier bloc 'else' selon les besoins de l'application (figure 17). Dans notre cas, nous avons changé la variable représentant le nombre d'échantillons par trame par la notre ('F').

Cette routine d'interruption sert aussi à la programmation de l'AIC en initiant une séquence de transmission (bloc 'if') pour ensuite envoyer le mot de commande de

programmation de l'AIC (bloc 'else if'). Sans trop rentrer dans les détails (voir [5] et programme en annexe), il faut juste savoir que 'serial_port[0][X_DATA]' représente l'adresse du registre de transmission du port série 0 et 'serial_port[0][R_DATA]' l'adresse de son registre de réception.

IV.3.3 LES ROUTINES DE CODAGE

Notre méthodologie de programmation répond à des impératifs relatifs à l'application. Nous avons privilégié en premier lieu l'augmentation de la vitesse afin de pouvoir faire du temps réel. Deuxièmement, nous avons veillé à garantir la plus grande flexibilité possible pour permettre la modification facile du programme. Enfin, nous avons essayé de clarifier le code au maximum par une programmation structurée et par l'insertion de commentaire.

Avant de présenter les routines de codage, nous commençons par spécifier toutes les variables globales utilisées.

IV.3.3.1 LES VARIABLES GLOBALES DE CODAGE

Variables	Type	Taille	Valeur	Fonction
p	int	1	10	Ordre de prédiction
hp	int	1	5	P/2
L	int	1	240	Nb échantillons trame LPC
F	int	1	160	Nb échantillons trame de parole
Fv	int	1	160	F+Nv-N
N	int	1	40	Nb échantillons sous trame
Nv	int	1	4	Nb échantillons trame d'excitation
SF	int	1	4	Nb sous trame
M	int	1	4	Nb impulsions
tpi	double	1	6.28318	2 * PI

tw	double	L		Fenêtre de Hamming
w	double	1	0.7	Coefficient de pondération
pw	double	p		Stockage des puissances de w
a	double	p		Coefficients de prédiction
aw	double	p		Coefficients de prédiction pondérés
q	double	p		LSP
f1	double	hp+1		Utilisée dans le calcul des LSP
f2	double	hp+1		Utilisée dans le calcul des LSP
l1	double	hp+1		Utilisée dans le calcul des LSP
l2	double	hp+1		Utilisée dans le calcul des LSP
u	double	Nv		Stockage de h(n)
st	BOOL	Nv		Indique si u[n] est calculé
phii	double	Nv		Autocorrélations de h(n)
inv0	double	1		Sauvegarde de 1/phii[0]
s_in	double	L		Trame LPC
s_out	double	F		Trame de sortie
xa	double *	1		Pointeur sur s_in
xf	double *	1		Pointeur sur s_out
r	double	Fv		Le résidu
pitch	int	1		La période du pitch
gain	double	1		Gain du LTP
b	double	M		Amplitudes des impulsions
norm	double	1	1.221E-04	Pour normaliser les amplitudes
scale	double	1	8190.01	Pour restaurer l'échelle des b[n]
mp	int	M		Positions des impulsions
dq	double	1	7.5E-04	Distance min entre LSP adjacents
qmax	double	1	0.999987	Valeur maximale des LSP
qmin	double	1	0.999987	Valeur minimale des LSP

lsp1_codes	double	8		Codes LSP1
lsp2_codes	double	8		Codes LSP2
lsp3_codes	double	8		Codes LSP3
lsp4_codes	double	8		Codes LSP4
lsp5_codes	double	16		Codes LSP5
lsp6_codes	double	16		Codes LSP6
lsp7_codes	double	8		Codes LSP7
lsp8_codes	double	8		Codes LSP8
lsp9_codes	double	8		Codes LSP9
lsp10_codes	double	8		Codes LSP10
b_codes	double	8		Codes des amplitudes
code	struct	1		Structure de bits de codage

Il faut noter que le type de donnée 'int' représente des entiers sur 32 bits, les types 'double' et 'float' sont équivalents et représentent des valeurs réelles sur 32 bits. Le type BOOL n'existe pas en C, nous l'avons déclaré comme étant un type 'int'.
Certaines variables de stockage sont utilisées pour l'accélération du programme, elles permettent d'éviter de recalculer la même chose plusieurs fois. En effet, notre plus grand souci est le temps de calcul (qui est limité) que nous allons essayer de réduire en augmentant notre utilisation de mémoire (les 16K de l'EVM le permettent).

(Pour plus de détail sur l'utilité de chaque variable voir programme en annexe)

IV.3.3.2 FONCTION D'INITIALISATION DES VARIABLES

```
/*=========================================================*/
/* INIT_ARRAYS ()                                          */
/*          1. SETUP HAMMING WINDOW                        */
/*          2. CLEAR DATA ARRAYS                           */
/*=========================================================*/
BOOL init_arrays(void)
{
        int n,i,j;/* general vars */
        double f;

        f=tpi/(L-1);
        for(n=0;n<L;n++)
        {
                tw[n]=0.54-0.46*cos(f*n);
        }

        for(pw[0]=w,i=0,j=1;j<p;j++)
        {
                pw[j]=pw[i]*w;
                i=j;
        }

        u[0]=1;
        st[0]=1;

        for(n=1;n<Nv;n++)
        {
                st[n]=0;
        }

        for(n=0;n<L;n++)
        {
                s_in[n]  =0;
                aic_in[n]=0;
        }

        for(n=0;n<F;n++)
        {
                s_out[n]  =0;
                aic_out[n]=0;
        }

        return 0;/*no error*/
}
```

Figure 18 : fonction 'init_arrays'

La fonction 'init_arrays' (figure 18) calcule les coefficients de la fenêtre de Hamming et les sauvegarde dans le vecteur 'tw', calcule les puissances de 'w' et les sauvegarde dans le vecteur 'pw'. Elle initialise 'u[0]' et les vecteurs 'st', 's_in' et 'aic_in', 's_out' et 'aic_out'.

IV.3.3.3 FONCTION DE FENETRAGE

La fonction 'window' (figure 19) réalise le fenêtrage d'un vecteur 'x' de dimension 'Ls' en utilisant la fenêtre de Hamming 'tw'.

```
/*=====================================================*/
/*  window (double * x, int Ls)                        */
/*  WINDOWING BUFFER x OF LENGTH Ls                    */
/*=====================================================*/
BOOL window(double * x,int Ls)
{
        int n;

        for(n=0;n<Ls;n++)
        {
                x[n]*=tw[n];
        }

        return 0;/*no error*/
}
```

Figure 19 : fonction 'window'

IV.3.3.4 FONCTION DE CALCUL DES COEFFICIENTS LP

La fonction 'wld' (figures 20) calcule les coefficients de prédiction selon l'algorithme Wienner Levinson Durbin (8). Nous avons utilisé exceptionnellement dans cette fonction un type de données de plus grande précision 'long double' (40 bits) pour assurer la stabilité du filtre de synthèse. En effet, nous avons constaté qu'avec le type 'double' l'accumulation des erreurs (de précision) conduit à l'instabilité du filtre de synthèse (les écarts entre les valeurs des coefficients de prédiction calculés par le DSP et ceux calculés par simulation sont trop importants). La fonction commence par le calcul des autocorrélations, puis elle fait 'p' itérations pour le calcul des coefficients lp. La solution finale est copiée dans le vecteur 'a' (et 'aw').

```
/*=========================================================*/
/*  wld(double * x,int Ls)                                 */
/* COMPUTATION OF LP COEFFICIENTS USING WLD ALGORITHME */
/*                    FROM VECTOR X OF LENGTH LS           */
/*=========================================================*/
BOOL wld(double * x,int Ls)
{
        int n,i,j,m,u,g;/* general vars */
        long double c;/* general vars */
        long double rs[p+1];/* signal autocor */
        long double  e[p+1];
        long double ai[p][p];/* a(i,j)=ai[i][j] */
        long double    k[p];/* reflection coefficients */
        m=p+1;


        /*1- computation of autocor */

        for(j=0;j<m;j++)
        {
                c=0;
                for(n=j;n<Ls;n++)
                {
                        c+=x[n]*x[n-j];
                }
                rs[j]=c;
        }

        /*2- start computing lp coefs */

        for(e[0]=rs[0],i=1;i<m;i++)
        {
                c=0;
                n=i-1;
                g=i-2;


                for(j=n,u=0;j>0;j--,u++)
                {
                        c+=ai[g][u]*rs[j];

                }

                k[n]=(rs[i]-c)/e[n];
```

Figure 20.a : fonction 'wld'

```
            for(u=0,j=1;j<i;j++)
            {
                ai[n][u]=ai[g][u]-k[n]*ai[g][n-j];
                u=j;
            }

            c=k[n];

            e[i]=(1-c*c)*e[n];
        }

        u=p-1;

        /*3- copy of final solution */
        for(j=0;j<p;j++)
        {
            a[j]=(double) ai[u][j];
            aw[j]=pw[j]*a[j];
        }

        return 0;/*no error*/
}
```

Figure 20.b : fonction 'wld'

IV.3.3.5 FONCTION DE CALCUL DU RESIDU

La fonction 'residual' (figure 21) calcule le résidu de prédiction 'r' selon l'équation (4) à partir du signal 'xf' de longueur 'Ls'.

```
/*=========================================================*/
/*  residual(int Ls)                                       */
/*  COMPUTATION OF THE RESIDUAL SIGNAL r FROM VECTOR xf    */
/*  OF LENGTH Ls                                           */
/*=========================================================*/
BOOL residual(int Ls)
{
        int n,i,j,g;/* general vars */
        double c;


        for(n=0;n<Ls;n++)
        {
                c=xf[n];

                if(n<p) g=n+1;
                else g=p+1;

                for(i=0,j=1;j < g;j++)
                {
                        c-=a[i]*xf[n-j];
                        i=j;
                }

                r[n]=c;

        }

        return 0;/*no error*/

}
```

Figure 21 : fonction 'residual'

IV.3.3.6 FONCTION DE CALCUL DU SIGNAL PONDERE

La fonction 'wsfilter' (figure 22) calcule le signal de parole pondéré selon l'équation (4) à partir du résidu 'r' de longueur 'Ls' mais en utilisant les 'aw'. Le signal de parole pondéré est sauvegardé dans le vecteur 'xf' qui sera le signal d'entrée pour le calcul de l'excitation.

```
/*=======================================================*/
/* wsfilter(int Ls)                                      */
/*      COMPUTATION OF THE WEIGHTED SPEECH FROM THE      */
/*                    RESIDUAL r OF LENGTH Ls            */
/*=======================================================*/
BOOL wsfilter(int Ls)
{
        int n,i,j,g;/* general vars */
        double c;

        for(n=0;n<Ls;n++)
        {
                c=r[n];

                if(n<p) g=n+1;
                else g=p+1;

                for(i=0,j=1;j<g;j++)
                {
                        c+=aw[i]*xf[n-j];
                        i=j;
                }

                xf[n]=c;
        }


        return 0;/*no error*/

}
```

Figure 22 : fonction 'wsfilter'

IV.3.3.7 FONCTION DE RECHERCHE DE MAXIMUM

La fonction 'mxm' (figure 23) donne la position 'pos' et l'amplitude 'max' du maximum du vecteur 'x' de longueur 'Ls'.

```
/*=====================================================*/
/*  mxm(double * x,int Ls,int * pos,double * max)      */
/*       GIVE THE AMPLITUDE (max) AND THE INDEX (pos)  */
/*        OF THE MAXIMUM OF VECTOR X OF LENGTH Ls      */
/*=====================================================*/
BOOL mxm(double * x,int Ls,int * pos,double * max)
{
        int n;/* general vars */
        double c;/* general vars */
        int ps;
        double mx;

        mx=x[0];
        ps=0;

        for(n=0;n<Ls;n++)
        {
                c=x[n];
                if(mx<c)
                {
                        mx=c;
                        ps=n;
                }
        }

        *pos=ps;
        *max=mx;

        return 0;/*no error*/

}
```

Figure 23 : fonction 'mxm'

IV.3.3.8 FONCTION DE CALCUL DE LA REPONSE IMPULSIONELLE DU FILTRE DE SYNTHESE

La fonction 'h' (figures 24) calcule la réponse impulsionelle du filtre de synthèse selon l'équation (39). Chaque coefficient *h(n)* calculé est sauvegardé dans 'u[n]' et 'st[n]' est mis à 1 pour indiquer cela. Si 'st[n]=1' la fonction renvoi directement la valeur de 'u[n]'.

Comme *h(n)=0* si *n* est négative alors on peut distinguer plusieurs cas selon la valeur du pitch (α). Ainsi la fonction 'h' opère conformément aux équations (56) (voir plus bas).

```
/*=======================================================*/
/* h(int n)                                              */
/* RETURN THE nTH SAMPLE OF THE GLOBAL SYNTHESIS FILTER  */
/*                    PULSE RESPONSE                     */
/*=======================================================*/
double h(int n)
{
        int i,j;/* general vars */
        double v;

        if(n<0)
        {
                v=0;
                goto end;
        }
        if(n<pitch)
        {
                if(n==0) v=1;
                else
                {
                        if(st[n])
                                v=u[n];
                        else
                        {
                                v=0;
                                i=0;
                                for(j=1;j<=p;j++)
                                {
                                        v+=aw[i]*h(n-j);
                                        i=j;
                                }
                                u[n]=v;
                                st[n]=1;
                        }
                }

        }
        else
        {
                if(st[n])
                        v=u[n];
                else
```

Figure 24.a : fonction 'h'

```
                {
                        v=gain*h(n-pitch);
                        i=0;
                        for(j=1;j<=p;j++)
                        {
                                v+=aw[i]*(h(n-j)-gain*h(n-j-pitch));
                                i=j;
                        }
                        u[n]=v;
                        st[n]=1;
                }
        }
end:
        ;

        return v;/*no error*/

}
```

Figure 24.a : fonction 'h'

L'expression de *h(n)* dans le cas général est :

$$h(n) = \delta(n) + Gh(n-\alpha) + \sum_{k=1}^{p} a_k^w \left[h(n-k) - Gh(n-k-\alpha)\right]$$

$$\Leftrightarrow h(n) = \begin{cases} 0 \quad si \quad n < 0 \\ 1 \quad si \quad n = 0 \\ \sum_{k=1}^{p} a_k^w \left[h(n-k)\right] \quad si \quad 1 \le n < \alpha \\ Gh(n-\alpha) + \sum_{k=1}^{p} a_k^w \left[h(n-k) - Gh(n-k-\alpha)\right] \quad si \quad \alpha \le n \end{cases} \qquad (56)$$

Dans notre codeur nous n'utilisons pas de LTP, alors nous avons posé 'pitch=1000' et 'gain=0' (voir programme).

IV.3.3.9 FONCTION DE CALCUL DES AUTOCORRELATIONS DE H(N)

```
/*=======================================================*/
/*  phi(int i,int j,int Ls)                              */
/*   COMPUTATION OF THE AUTOCORRELATION OF THE GLOBAL    */
/*     SYNTHESIS FILTER PULSE RESPONSE AT INDEXES i & j   */
/*                    Ls IS THE NB OF SAMPLES            */
/*=======================================================*/
double phi(int i,int j,int Ls)
{
        int n,k;/* general vars */
        double c;/* general vars */
        c=0;

                k=(i-j);
                if(k<0) k=-k;
                for(n=k;n<Ls;n++)
                {
                        c+=h(n)*h(n-k);
                }

        return c;/*no error*/

}
double phi2(int i,int j,int Ls)
{
        int n,k;/* general vars */
        double c;/* general vars */
        c=0;

                k=(i-j);
                if(k<0) k=-k;
                for(n=k;n<Ls;n++)
                {
                        c+=u[n]*u[n-k];
                }

        return c;/*no error*/

}
```

Figure 25 : fonction 'phi'

La fonction 'phi' (figure 25) calcule les autocorrélations de la réponse impulsionelle du filtre de synthèse (selon l'équation (41)). Ces autocorrélations seront sauvegardées dans le vecteur 'phii'.

La fonction 'phi2' est une version plus rapide de la fonction 'phi'. La différence entre les deux est que l'une fait appelle à la fonction 'h' alors que la deuxième utilise les valeurs stockées dans 'u'. Cette simple différence fait que 'phi2' est plus rapide que 'phi' car l'appel de 'h' fait perdre beaucoup de cycles machine comme le montre la figure (26).

```
_h:
;*************************************************************
;* début de la fonction 'h'                                  *
;*************************************************************
        push    fp
        ldiu    sp,fp
        push    r4
        push    r5
        pushf   f6
        pushf   f7
        push    ar4
        push    ar5
        push    ar6
        push    ar7
        ldiu    *-fp(2),ar6         ; |346|
;*************************************************************
;* corps de la fonction 'h'                                  *
;*************************************************************
;*
;*************************************************************
;*  fin de la fonction 'h'                                   *
;*************************************************************
;** 372         ----------------------  return v;
        ldiu    1,r0                ; |372|
        stf     f6,*+ar0(ir0)       ; |371|
        sti     r0,*ar4             ; |372|
L98:
        pop     ar7                 ; |372|
        ldiu    *-fp(1),bk          ; |372|
        pop     ar6                 ; |372|
        ldfu    f6,f0               ; |372|
        pop     ar5                 ; |372|
        pop     ar4                 ; |372|
        ldiu    *fp,fp              ; |372|
        popf    f7                  ; |372|
        popf    f6                  ; |372|
        bud     bk                  ; |372|
        pop     r5                  ; |372|
        pop     r4                  ; |372|
        subi    2,sp                ; |372| Unsigned
```

Figure 26 : extrait du code assembleur de la fonction 'h'

On voit que 'h' consomme au moins 11 cycles machine au début de son exécution (chaque instruction = 1 cycle machine, voir [7]) et au moins 13 cycles à la fin (ce qui fait 24 cycles machine). Comme 'phi' doit être appelé 'Nv' fois et que $phi(i,j,Nv)$ fait $2\times(Nv-|i-j|)$ appelle à 'h' on perd donc (si $Nv=40$ et $|i-j|=0,\ldots,Nv-1$) plus de 39360 $(24\times\frac{40}{2}(2+80))$ cycles machine (soit plus de 1.3 ms).

Seulement pour utiliser 'phi2' il faut que 'u' soit totalement rempli, alors on calculera juste 'phii[0]' par 'phi' pour être sur du remplissage de 'u' puis on utilisera 'phi2' pour le reste.

En fait, la plus grande charge de calcul est due aux $\phi(i,j)$. L'utilisation de la méthode des autocorrélations, de la fonction 'phi2' et la sauvegarde des $\phi(i,j)$ dans le vecteur 'phii' nous a permis de gagner énormément de temps et de réaliser un codeur en temps réel.

IV.3.3.10 FONCTION DE CALCUL DE L'EXCITATION

La fonction 'mpe' (figures 27) calcule les amplitudes et les positions des impulsions de l'excitation selon l'algorithme (51) et en utilisant les équations (53) et (55). La fonction commence d'abord par calculer la première impulsion ('mp[0]' et 'b[0]') puis continue avec le reste des impulsions. Comme pour les 'phii' le calcul des 'psi' (équation 41) se fait en utilisant directement 'u' au lieu de faire appelle à la fonction 'h'.

A la fin, la fonction 'order' ordonne les impulsions trouvées par ordre croissant de la position. Ceci est nécessaire pour le codage des positions des impulsions (voir quantification de l'excitation chapitre II).

```
/*=======================================================*/
/*  mpe(int offset,int Ls)                               */
/*    COMPUTATION OF THE EXITATION PULSES AMPLITUDES     */
/*    AND POSITIONS                                      */
/*              Ls IS THE EXCITATION FRAME LENGTH        */
/*              offset IS THE START INDEX OF THE EXCITATION */
/*              FRAME                                    */
/*=======================================================*/
BOOL mpe(int offset,int Ls)
{
        int n,i,j,m,g,k;/* general vars */
        double c;/* general var */
        double psi[M][Nv];
        double d[Nv];
        double * x;

        x=xf+offset;


        /*1- computation of b[0] & mp[0] */

        /* computation of psi[0] */
        for(i=0;i<Ls;i++)
        {
                c=0;
                for(n=i;n<Ls;n++)
                {
                        c+=x[n]*u[n-i];/* h(n-i) */
                }
                psi[0][i]=c;/* j=0 */
        }

        /* computation of square(psi[0]) */
        for(i=0;i<Ls;i++)
        {
                c=psi[0][i];
                d[i]=c*c;/**/
        }

        /* find max square(psi[0])=mp[0] */
        mxm(d,Ls,mp,b);
        /* computation of b[0] */
        b[0]=psi[0][mp[0]]*inv0;
```

Figure 27.a : fonction 'mpe'

```
        /*2- step 1 to M-1 : computation of b[j] & mp[j] */

        for(n=0,j=1;j<M;j++)
        {
                /* update psi[j] */
                for(i=0;i<Ls;i++)
                {
                        k=mp[n]-i;
                        if(k<0) k*=-1;
                        psi[j][i]=psi[n][i]-b[n]*phii[k];
                }

                /* clear psi[j] where there is already a pulse */
                for(i=0;i<j;i++)
                {
                        psi[j][mp[i]]=0;
                }

                /* computation of square(psi[j]) */
                for(i=0;i<Ls;i++)
                {
                        c=psi[j][i];
                        d[i]=c*c;
                }

                /* find max square(psi[j])=mp[j] */
                mxm(d,Ls,&mp[j],&b[j]);
                /* computation of b[j] */
                b[j]=psi[j][mp[j]]*inv0;
                n=j;

        }

        /*3- Put  pulses in ascending position order */
        order();

        return 0;/* no error */
}
```

Figure 27.b : fonction 'mpe'

IV.3.3.11 FONCTION D'ARRANGEMENT DES IMPULSIONS

La fonction 'order' (figure 28) ordonne les impulsions trouvées par ordre croissant de la position.

```
/*=====================================================*/
/*  BOOL order()                                       */
/*            Put  pulses in ascending position order  */
/*=====================================================*/
BOOL order()
{
        int n,i,rp,m,index;
        double rb;

        for(i=0;i<M;i++) /* for each position */
        {
                m=Nv;

                for(n=i;n<M;n++) /* find the max pulse index */
                {
                        if(mp[n]<m)
                        {
                                index=n;
                                m=mp[n];
                        }
                }
                /* rotation */
                rp=mp[i];
                mp[i]=mp[index];
                mp[index]=rp;
                rb=b[i];
                b[i]=b[index];
                b[index]=rb;
        }
        return 0;
}
```

Figure 28 : fonction 'order'

IV.3.3.12 FONCTION DE SYNTHESE DE LA PAROLE

La fonction 'synt' (figure 29) synthétise la parole à partir des impulsions d'excitation calculées par la fonction 'mpe' selon l'équation (36). La fonction reconstitue une sous trame de parole de longueur 'Ls' qu'elle met dans le vecteur 'xf' à partir de la position indiquée par 'offset'.

```
/*=========================================================*/
/* synt(int offset,int Ls)                                 */
/*     SYNTHESIS OF SPEECH USING THE EXCITATION PULSES     */
/*                  Ls IS THE SUB FRAME LENGTH             */
/*                offset IS THE START INDEX OF THE SUB FRAME*/
/*=========================================================*/
BOOL synt(int offset,int Ls)
{
        int n,i,j;/* general vars */
        double * x,c;

        x=xf+offset;

        for(n=0;n<Ls;n++)
        {
                c=0;
                for(j=0;j<M;j++)
                {
                        i=n-mp[j];
                        c+=b[j]*h(i);
                }
                x[n]=c;
        }

        return 0;/*no error*/
}
```

Figure 29 : fonction 'synt'

IV.3.3.13 FONCTION DE FILTRAGE DE SORTIE

La parole synthétisée par la fonction 'synt' est la reconstitution de la parole originale passée à travers le filtre de pondération (fonction 'wsfilter'). Pour retrouver la parole non pondérée, on fait passer la parole synthétisée par le filtre inverse (figure 30). C'est ce que fait la fonction 'sfilter' (figure 31).

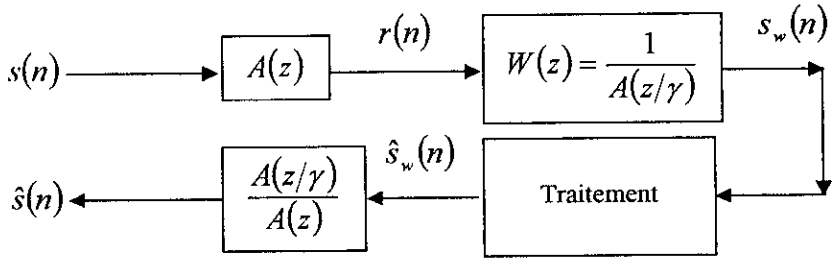


Figure 30 : schéma de production de la parole

```
/*=======================================================*/
/*   sfilter(int Ls)                                     */
/* FILTER THE WEIGHTED SPEECH TO GIVE THE OUTPUT SPEECH */
/*       Ls IS THE SPEECHE FRAME LENGTH                  */
/*=======================================================*/
BOOL sfilter(int Ls)
{
        int n,i,j,g;/* general vars */
        double c;


        for(n=0;n<Ls;n++)
        {
                c=xf[n];
                i=0;
                if(n<p) g=n+1;
                else g=p+1;
                for(j=1;j < g;j++)
                {
                        c-=aw[i]*xf[n-j];
                        i=j;
                }
                r[n]=c;

        }

        for(n=0;n<Ls;n++)
        {
                c=r[n];
                i=0;
                if(n<p) g=n+1;
                else g=p+1;
                for(j=1;j<g;j++)
                {
                        c+=a[i]*xf[n-j];
                        i=j;
                }
                xf[n]=c;
        }


        return 0;/*no error*/

}
```

Figure 31 : fonction 'sfilter'

IV.3.3.14 FONCTION DE CALCUL DES LSP

```
/*=======================================================*/
/* BOOL lsp()                                            */
/*        computation of lsps from lp coefficients       */
/*=======================================================*/
BOOL lsp()
{
        double h=0.02,nb=5,x,y,xi,al,bh;
        double c1,c2,*f;
        int i,j;
        BOOL odd=0;

        for(f1[0]=f2[0]=1,i=0;i<hp;i++)
        {
                c1=a[i];
                c2=a[p-i-1];
                f1[i+1]=-c1+c2+f1[i];
                f2[i+1]=-(c1+c2+f2[i]);
        }

        for(xi=1,i=0;i<p;odd= odd ^ 1,xi=q[i],i++)
        {
                if(odd) f=f1;
                else    f=f2;
                c1=c(xi,f);

                for(x=xi-h;x>-1;x-=h)
                {
                        c2=c(x,f);
                        if(c2==0) /* lsp found */
                        {
                                q[i]=x;
                                break;
                        }
                        if(c1*c2<0) /* x<lsp<x+h */
                        {
                        /* Do nb bisections to refine the lsp */
                                for(al=x,bh=x+h,c1=c(bh,f),j=0;j<nb;j++)
                                {
                                        y=0.5*(al+bh);
                                        c2=c(y,f);
                                        if(c2==0) {q[i]=y;goto next;}
                                        if(c1*c2<0) al=y; /* y<qi<bh */
                                        else /* al<qi<y */
```

Figure 32.a : fonction 'lsp'

```
                                        {
                                                bh=y;
                                                c1=c(bh,f);
                                        }
                                }
                                q[i]=0.5*(al+bh);goto next;/* error < h/2^(nb+1) */
                        }
                }
next: ;
        }


        if(i==p) return 0; /* all lsp have been found */
        else return 1;

}
double c(double x,double * f)
{
        int k;

        for(l1[5]=0,l1[4]=1,k=4;k>0;k--)
        {
                l1[k-1]=2*x*l1[k]-l1[k+1]+f[hp-k];
        }
        return (x*l1[0]-l1[1]+f[5]*0.5);
}
```

Figure 32.b : fonction 'lsp'

La fonction 'lsp' (figures 32) calcule les LSP 'q' à partir des coefficients de prédiction 'a'. La fonction commence par calculer 'f1' et 'f2' selon les équations (21) (22), puis elle entame la recherche des LSP en partant de 1 et en descendant

vers –1 avec un pas 'h=0.02'. Le polynôme dont les racines sont les LSP est évalué grâce à la fonction 'c' selon l'algorithme (25). La fonction recherche alternativement un LSP d'indice paire puis un LSP d'indice impaire. Dès qu'il y a un changement de signe (présence d'une racine), l'intervalle est raffiné par 'nb' bissections.

IV.3.3.15 FONCTION DE CONVERSION DES LSP EN PARAMETRES LPC

La fonction 'LsptoLp' (figures 33) recalcule les paramètres LPC ('a') à partir des paramètres LSP ('q'). La fonction commence par calculer 'l1' et 'l2' selon l'algorithme (26). Puis elle calcule 'f1' et 'f2' selon les équations (27) et (28) et enfin elle obtient les paramètres LPC en appliquant la relation (29).

```
/*=======================================================*/
/* BOOL LsptoLP()                                        */
/*        transform lsps to lp coefficients              */
/*=======================================================*/
BOOL LsptoLP()
{
        double d1,d2;
        int i,j,i1,i2,j1,j2,ti;



        l1[0]=1;
        l1[1]=-2*(q[1]);

        l2[0]=1;
        l2[1]=-2*(q[0]);

        for(i=2;i<6;i++)
        {
                i1=i-1;
                i2=i-2;
                ti=2*i1;
                d1=-2*(q[ti+1]);
                d2=-2*(q[ti]);
                l1[i]=d1*l1[i1]+2*l1[i2];
                l2[i]=d2*l2[i1]+2*l2[i2];
```

Figure 33.a : fonction 'LsptoLP'

```
            for(j=i1;j>1;j--)
            {
                j1=j-1;
                j2=j-2;
                l1[j]=l1[j]+d1*l1[j1]+l1[j2];
                l2[j]=l2[j]+d2*l2[j1]+l2[j2];
            }
            l1[1]=l1[1]+d1*l1[0];
            l2[1]=l2[1]+d2*l2[0];
        }
        for(i=1;i<6;i++)
        {
            i1=i-1;
            f1[i]=l1[i]-l1[i1];
            f2[i]=l2[i]+l2[i1];
        }
        for(i=1;i<6;i++)
        {
            a[i-1]=-(f1[i]+f2[i])*0.5;
            a[i+4]=(f1[6-i]-f2[6-i])*0.5;
        }
        return 0;

}
```

Figure 33.b : fonction 'LsptoLP'

IV.3.3.16 FONCTION DE QUANTIFICATION D'UNE VALEUR REELLE

La fonction 'QSV' (figure 34) réalise la quantification d'une valeur 'v' en utilisant le codebook 'cv' de dimension 'Mv'. La fonction calcule la distance euclidienne entre 'v' et tous les éléments du vecteur 'cv' puis choisit la composante de 'cv' qui donne l'erreur minimale. La valeur de cette composante est affectée à 'v', et son indice est transmis par la fonction comme valeur de retour.

```
/*=======================================================*/
/*  int QSV(int Mv,double * v,const double * cv)         */
/*        Quantize v using code vector cv                */
/*        Mv = cv length                                 */
/*=======================================================*/
int QSV(int Mv,double * v,const double * cv)
{
        int i,m;
        double d,dm;

        for(dm=1e+10,i=0;i<Mv;i++)
        {
                d=*v-cv[i];
                /*if(d<0) d*=-1;*/
                d*=d;
                if(d<dm)
                {
                        dm=d;
                        m=i;
                }
        }
        *v=cv[m];
        return m;
}
```

Figure 34 : fonction 'QSV'

IV.3.3.17 FONCTION DE QUANTIFICATION DES LSP

La fonction 'Qlsp' (figure 35) réalise la quantification et le codage des LSP. La fonction commence par calculer les différences pour les LSP 'q[1]' à q[5] (LSP2 à LSP6 sont codés par la différence, voir quantification au chapitre II). Puis, chaque LSP est quantifié individuellement grâce à la fonction 'QSV'. Les indices fournis par la fonction 'QSV' sont enregistrés dans la structure de codage 'code'. On réalise ainsi le codage des LSP sur 32 bits.

```
/*=======================================================*/
/*  BOOL Qlsp()                                          */
/*        Quantify & code LSPs                           */
/*=======================================================*/
BOOL Qlsp()
{
        double f1,f2;

        f1=q[1];
        q[1]=q[0]-f1;
        f2=q[2];
        q[2]=f1-f2;
        f1=q[3];
        q[3]=f2-f1;
        f2=q[4];
        q[4]=f1-f2;
        q[5]=f2-q[5];

        code.cw1.lsp1 = QSV(8 ,q+0,lsp1_codes);
        code.cw1.lsp2 = QSV(8 ,q+1,lsp2_codes);
        code.cw1.lsp3 = QSV(8 ,q+2,lsp3_codes);
        code.cw1.lsp4 = QSV(8 ,q+3,lsp4_codes);
        code.cw1.lsp5 = QSV(16,q+4,lsp5_codes);
        code.cw2.lsp6 = QSV(16,q+5,lsp6_codes);
        code.cw2.lsp7 = QSV(8 ,q+6,lsp7_codes);
        code.cw2.lsp8 = QSV(8 ,q+7,lsp8_codes);
        code.cw2.lsp9 = QSV(8 ,q+8,lsp9_codes);
        code.cw2.lsp10= QSV(8,q+9,lsp10_codes);

        return 0;
}
```

Figure 35 : fonction 'Qlsp'

IV.3.3.18 FONCTION DE QUANTIFICATION DE L'EXCITATION

La fonction 'Qpulses' (figure 36) réalise la quantification et le codage des positions et des amplitudes des impulsions d'excitation. La fonction commence par calculer les différences entres les positions des impulsions (elles sont codées par la différence) tout en s'assurant qu'elles ne dépassent pas 31 (voir quantification au chapitre II). Puis elle enregistre les nouvelles valeurs dans la structure de codage. Le codage des amplitudes se fait comme pour les LSP en utilisant la fonction 'QSV'. La variable 'n' est l'indice de la sous trame. La structure 'code' contient 4 structures 'cw3', c.-à-d. une pour chaque sous trame.

```
/*=========================================================*/
/*  BOOL Qpulses(int n)                                    */
/*        Quantify & code Pulses                           */
/*        n is the sub frame index                         */
/*=========================================================*/
BOOL Qpulses(int n)
{
        int m;


        if(mp[0]>31) mp[0]=31;

        mp[1]-=mp[0];
        if(mp[1]>31) mp[1]=31;

        m=mp[1]+mp[0]; /* m= new mp[1]*/
        mp[2]-=m;
        if(mp[2]>31) mp[2]=31;

        m+=mp[2]; /* m= new mp[2]*/
        mp[3]-=m;
        if(mp[3]>31) mp[3]=31;


        code.cw3[n].P12.m1 = mp[0];
        code.cw3[n].P12.m2 = mp[1];
        code.cw3[n].P34.m3 = mp[2];
        code.cw3[n].P34.m4 = mp[3];

        /* normalization of the amplitudes before coding */
        for(m=0;m<M;m++)
        {
                b[m]*=norm;
        }

        code.cw3[n].P12.b1 = QSV(CBS ,b+0,b_codes);
        code.cw3[n].P12.b2 = QSV(CBS ,b+1,b_codes);
        code.cw3[n].P34.b3 = QSV(CBS ,b+2,b_codes);
        code.cw3[n].P34.b4 = QSV(CBS ,b+3,b_codes);

        return 0;
}
```

Figure 36 : fonction 'Qpulses'

IV.3.3.19 FONCTION DE DECODAGE DES LSP

```
/*=========================================================*/
/*  BOOL DECODE_lsp()                                      */
/*        Decode LSPs                                      */
/*=========================================================*/
BOOL DECODE_lsp()
{
        double d;


        /* decode LSPs */

        q[0]  =lsp1_codes[code.cw1.lsp1];
        q[1]  =lsp2_codes[code.cw1.lsp2];
        q[2]  =lsp3_codes[code.cw1.lsp3];
        q[3]  =lsp4_codes[code.cw1.lsp4];
        q[4]  =lsp5_codes[code.cw1.lsp5];
        q[5]  =lsp6_codes[code.cw2.lsp6];
        q[6]  =lsp7_codes[code.cw2.lsp7];
        q[7]  =lsp8_codes[code.cw2.lsp8];
        q[8]  =lsp9_codes[code.cw2.lsp9];
        q[9] =lsp10_codes[code.cw2.lsp10];

        /* check for stability */

        if(q[0]>qmax) q[0]=qmax;

        q[1]=q[0]-q[1];
        d=q[0]-dq;
        if(q[1]>d) q[1]=d;

        q[2]=q[1]-q[2];
        d=q[1]-dq;
        if(q[2]>d) q[2]=d;

        q[3]=q[2]-q[3];
        d=q[2]-dq;
        if(q[3]>d) q[3]=d;

        q[4]=q[3]-q[4];
        d=q[3]-dq;
        if(q[4]>d) q[4]=d;
```

Figure 37.a : fonction 'DECODE_lsp'

```
        q[5]=q[4]-q[5];
        d=q[4]-dq;
        if(q[5]>d) q[5]=d;

        d=q[5]-dq;
        if(q[6]>d) q[6]=d;

        d=q[6]-dq;
        if(q[7]>d) q[7]=d;

        d=q[7]-dq;
        if(q[8]>d) q[8]=d;

        d=q[8]-dq;
        if(q[9]>d) q[9]=d;

        if(q[9]<qmin) q[9]=qmin;


        return 0;
}
```

Figure 37.b : fonction 'DECODE lsp'

La fonction 'DECODE_lsp' (figures 37) réalise le décodage des LSP, recalcule les LSP codés par la différence et vérifie la stabilité du filtre de synthèse (voir quantification des LSP chapitre II).

IV.3.3.20 FONCTION DE DECODAGE DES IMPULSIONS D'EXCITATION

La fonction 'DECODE_pulses' (figure 38) réalise le décodage des amplitudes et des positions des impulsions, puis recalcule les positions codées par la différence.

```
/*=========================================================*/
/*  BOOL DECODE_pulses(int n)                              */
/*        Decode pulses amplitudes & positions             */
/*=========================================================*/
BOOL DECODE_pulses(int n)
{
        int i;


        mp[0] = code.cw3[n].P12.m1;
        mp[1] = code.cw3[n].P12.m2;
        mp[2] = code.cw3[n].P34.m3;
        mp[3] = code.cw3[n].P34.m4;


        mp[1]+=mp[0];
        mp[2]+=mp[1];
        mp[3]+=mp[2];



        b[0]=b_codes[code.cw3[n].P12.b1];
        b[1]=b_codes[code.cw3[n].P12.b2];
        b[2]=b_codes[code.cw3[n].P34.b3];
        b[3]=b_codes[code.cw3[n].P34.b4];

        /* restoration of the scale of the amplitudes */
        for(i=0;i<M;i++)
        {
                b[i]*=scale;
        }

        return 0;
}
```

Figure 38 : fonction 'DECODE_pulses

Maintenant que nous avons vu toutes les routines de codage, nous pouvons enfin donner le programme principal complet.

IV.3.4 LE PROGRAMME PRINCIPAL

```
void main(void)
{
        int n,sf,g,i,j;
        double c,f;

   init_evm();          /* INITIALIZE TMS320C30 AND EVM                 */
   init_arrays();       /* INITIALIZE MPE ARRAYS                        */
   init_aic();          /* INITIALIZE TLC32044 AIC COMMUNICATIONS */

   for(;;)
   {
        wait_frame();        /* WAIT FOR FULL FRAME OF DATA          */
        /*=============================================*/
                        xa=s_in;/* xa= LPC frame */
                        xf=s_out;/* xf= output speech */
                        memcpy(xf ,xa + N , Fv); /*1- copy speech frame */
                        window(xa,L);/*2- xa*window */
                        wld(xa,L);/*3- computation of lp coefficients */
                        lsp();/*4- computation of lsp */
                        Qlsp();/*5- quantification & encoding of lsp */
                        DECODE_lsp();/*6- decoding of lsp */
                        LsptoLP();/*7- compute lp coefs from quantified lsp*/
                        residual(Fv);/*8- computation of the residual signal */
                        wsfilter(Fv);/*9- computation of the weighted speech */
                        /*10- computation of phii */
                        phii[0]=phi(0,0,Nv);
                        inv0=1/phii[0];
                        for(i=1;i<Nv;i++)
                        {
                                phii[i]=phi2(i,0,Nv);
                        }
                        /*11----------------------*/
                        for(g=0,sf=0;sf<SF;sf++,g+=N)
                        {
                                mpe(g,Nv); /*12- computation of excitation */
                                /*13- quantification & encoding of pulses */
                                Qpulses(sf);
                                DECODE_pulses(sf);/*14- Decoding of pulses*/
                                synt(g,N);/*15- synthesis of the sub frame*/
                        }
                        /*----------------------*/
                sfilter(F);/*16-Compute output speech*/
        /*=============================================*/
   }
}
```

Figure 39 : le programme principal

Ce programme est la réalisation du schéma de codage de la figure (4) et (31) :

1- Copie de la trame de parole dans le vecteur 'xf'(xf=s). La trame de parole est située à un offset d'une sous trame à partir du début de la trame LPC (voir figures 11 et 13).
2- Fenêtrage de la trame LPC.
3- Calcul des coefficients de prédiction 'a'.
4- Calcul des LSP 'q' à partir de 'a'.
5- Quantification est codage des LSP.
6- Décodage des LSP. $(q = \hat{q})$
7- Calcul des coefficients de prédiction à partir des LSP quantifiés. $(a = \hat{a})$ Nous faisons cela pour calculer l'excitation pour un filtre de synthèse quantifié. Ca permet de minimiser l'effet de la quantification des LSP.
8- Calcul du résidu. Dans notre codeur nous n'avons pas besoin de faire le calcul du résidu puisque nous ne calculons pas de *pitch*. Mais nous avons fait cela au cas où nous voudrions rajouter le LTP.
9- Calcul de la parole pondérée en utilisant les coefficients de prédiction pondérés 'aw' (xf=s_w).
10- Calcul des 'phii'. 'phii[0]' est calculé par la fonction 'phi', puis la fonction 'phi2' calcule le reste des 'phii'. La variable 'inv0' est utilisée dans la fonction 'mpe', elle n'est calculée qu'une seule fois par trame.
11- Pour chaque sous trame, calcul de l'excitation et reconstitution de la parole.
12- Calcul des positions et des amplitudes des impulsions de l'excitation.
13- Quantification et codage des impulsions.
14- Décodage des impulsions.
15- Synthèse de la sous trame à partir des impulsions d'excitation quantifiées (xf=$\hat{s}_w$).
16- Calcul de la parole non pondérée par le filtre inverse au filtre de pondération (xf=$\hat{s}$) (figure 31).

Après cela, la fonction 'wait_frame' attend que la nouvelle trame de parole soit prête pour faire la rotation des buffers. Le contenu de 'xf' sera transféré dans 'aic_out' pour être sorti sur l'interface analogique. Et le programme continue ainsi.

Bien sûr dans le cas pratique, la structure de bits 'code' est transmise chaque trame au récepteur.

IV.3.5 TEST.

Le programme marche correctement, la parole synthétique est comparable à l'originale.

Nous avons testé le programme avec et sans quantification des LSP, avec et sans quantification des amplitudes des impulsions. L'effet de la quantification des LSP n'est pas perceptible (à l'audition) par contre la quantification des amplitudes induit du bruit. Cela est dû au fait que nous n'avons utilisé que 3 bits pour la quantification de chaque amplitude. La qualité peut être améliorée avec une quantification sur 4 bits (8,8 kb/s).

Nous avons fait varier le nombre d'impulsions par sous trame, et nous avons constaté que pour 8 impulsions (14.4 kb/s) la parole synthétique était pratiquement identique à l'originale.

En conclusion, un codeur de parole complet a été implémenté sur TMS320C30. Le codeur travail en temps réel et produit une parole de bonne qualité a un débit de 8kb/s.

CONCLUSION

CONCLUSION

Le codage de parole est aujourd'hui plus que jamais indispensable dans la transmission de la parole. En une décennie, les techniques de traitement de la parole ont connu une révolution.

Parmi les applications de ces techniques, celle qui touche pour le moment le plus d'utilisateurs, est celle de la téléphonie mobile.

En effet, une proportion grandissante de la population transporte, souvent sans le savoir, un ordinateur de poche spécialisé dans l'analyse synthèse LPC.

L'objectif du travail présenté dans ce document à savoir : l'implémentation d'un codeur de parole multi- pulse de bonne qualité et en temps réel est atteint.

Ce travail présente donc, l'implémentation d'un codeur de parole multi pulse de 8 kb/s en temps réel sur un DSP TMS 320C30. Une attention particulière, a été observée à toutes les étapes de codage ce qui s'est traduit par l'obtention d'une excellente qualité de la parole.

En outre, dans ce codeur la vitesse a été optimisée au maximum pour permettre le codage en temps réel tout en préservant la structuration et la clarté du code.

Le résultat de cette optimisation de la vitesse a été un codeur performant, flexible et enfin facilement modifiable, car les paramètres du codeur peuvent être changés facilement pour modifier le débit et la qualité de la parole.

De plus, le codeur peut être amélioré par l'utilisation de la quantification vectorielle (QV), pour les LSP et les amplitudes des impulsions. Dans la référence [2], les LSP sont quantifiés par QV à deux étages sur 18 bits.

Pour ce travail il a été utilisé le module d'évaluation du processeur de traitement de signal de TEXAS INSTRUMENT, l'implémentation en langage C avec le compilateur C compatible.

Au terme des tests effectués sur ce travail d'implémentation
Il a été noté avec satisfaction le fonctionnement correct du programme puisque la parole synthétique est comparable à l'original.

L'effet de la quantification des LSP n'est pas perceptible à l'audition alors que la quantification des amplitudes induit, quant à elle, du bruit (lorsque il est utilisé seulement 3 bits pour la quantification). La qualité peut être améliorée avec 4 bits (8,8 kb/s).

Le fait de varier le nombre d'impulsions par sous trame, donne pour 8 impulsions (14,4 kb/s) à la fin du test une parole synthétique pratiquement identique à l'originale.

BIBLIOGRAPHIE

Bibliographie

[1] R.A.Salami, L. Hanzo, R. Steele, K.H.J. Wong et I. Wassell. « speech coding » : chapitre 3 du livre « Mobile radio communications ». Pentech Press, London, 1991.

[2] « Coding of speech at 8 kbit/s using Conjugate-Structure-Algebraic-Code-Excited-Linear-Prediction (CS-ACELP) ». ITU-T Recommendation G.729, 03/1996.

[3] TMS320C30 evaluation module technical reference (spru069).

[4] TMS320C3x user's guide (spru031).

[5] TLC32044 data sheet (slas017f).

[6] TMS320C3X/4X optimizing C compiler user's guide (spru034h).

[7] TMS320C3X/4X floating point DSP assembler language tools user's guide (spru035b).

[8] ***Traitement de la Parole***, R. Boite, H. Bourlard, T. Dutoit, J. Hancq et H. Leich, Presses Polytechniques Universitaires Romandes, Lausanne, **2000.**

[9] B.S. Atal, S.L. Hanauer. Speech Analysis and Synthesis by linear Prediction of the speech Wave. *J. Acoust.Soc. Amerr., Vol 50 No2* PP. 637-657, **1971.**

[10] B. S. Atal. Efficient coding of LPC parameters by temporal decomposition. In *Proc. IEEE ICASSP 83*, pages 1-84, **1983.**

[11] M.R. Schroeder, B. Atal. Code-Excited Linear Prediction(CELP): High Quality Speech at Very Low Bit Rates.*Proc. IEEE ICASSP-85*, pp. 937-940, Tamp, **1985.**

[12] A. Gersho. Advances in speech and audio compression. *Proc. IEEE*, 82(6):900-918, June **1994.**

[13] T.E. Tremain. the government standard Linear Predictive Coding Algorithm: LPC10. *Speech Technology, Vol.1, No2*, pp. 40-49, Apr. **1982.**

[14] L.M. Supplee, R.P. Cohn, J.S. Collura, and A.V. McCree, "MELP : The new federal standard at 2400 bits/s",*IEEE International Conference on Acoustics, Speech, and Signal Processing*, Munich, April 1997, pp. 1591-1594.

Annexe

Cette annexe donne le code source complet du codeur multi-pulse que nous avons présenté.

Les fichiers 'C30_1.H', 'C30_2.H'
sont fournis par TI pour l'utilisation de l'EVM.
Le fichier 'MPE_C30.H' contient les prototypes des fonctions et la déclaration de toutes les variables globales.
Le fichier 'MPE_C30.C' contient le programme principal et le code de toutes les fonctions
Le fichier 'VECS.ASM' contient la section assembleur d'initialisation du vecteur d'interruptions. Pour spécifier quelle routine (d'interruption), correspond à quelle interruption du DSP.
Le fichier 'MPE.CMD' est le fichier de commande pour l'édition de lien (link).

Génération du programme exécutable :

-Assemblage du fichier 'VECS.ASM' :

```
ASM30 vecs.asm
```

-Compilation des fichiers 'C30.C' et 'MPE_C30.C' :

```
CL30 -s -v30 -mmnci -x2  c30 mpe_c30
```

-Lien des fichiers objet:

```
LNK30 mpe.cmd
```

```
/************************************************************************/
/*   C30_1.H                                                            */
/*                                                                      */
/*   TMS320C30 EVALUATION MODULE TMS320C30 SUPPORT FILE 1               */
/*      : FUNCTION PROTOTYPES, MACROS, STRUCTURES                       */
/*                                                                      */
/*   (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                */
/************************************************************************/
/* FUNCTION PROTOTYPES*=================================================*/
/*======================================================================*/
void c_int11(void);
void c_int99(void);
void init_evm(void);
void init_aic(void);
void init_host(void);
void configure_aic(int i);
void recording(int *source, int rec_cmd);
void playing(int *dest, int play_cmd);

/*======================================================================*/
/* MACROS *=============================================================*/
/*======================================================================*/
#define OFF    0x00
#define ON     0x01
#define TOGGLE 0x01            /* BIT-MASK TO TOGGLE BETWEEN ON AND OFF */
#define NONE   0x00
/*======================================================================*
/*          TMS320C30 MEMORY-MAPPED CONTROL REGISTER INDICES            */
/*======================================================================*/
/*----------------------------------------------------------------------*/
/* BASE GLOBAL CONTROL REGISTER                                         */
/*----------------------------------------------------------------------*/
#define GLOBAL 0
/*----------------------------------------------------------------------*/
/* EXTERNAL BUS CONTROL REGISTERS                                       */
/*----------------------------------------------------------------------*/
#define EXPANSION 0                                  /* EXPANSION BUS */
#define PRIMARY   4                                  /* PRIMARY BUS   */
/*----------------------------------------------------------------------*/
/* SERIAL PORT CONTROL REGISTERS                                        */
/*----------------------------------------------------------------------*/
#define X_PORT  2                                /* TRANSMIT CONTROL */
#define R_PORT  3                                /* RECEIVE  CONTROL */
#define X_DATA  8                                /* TRANSMIT DATA    */
#define R_DATA 12                                /* RECEIVE  DATA    */
/*----------------------------------------------------------------------*/
/* TIMER CONTROL REGISTERS                                              */
/*----------------------------------------------------------------------*/
#define PERIOD  8                                 /* PERIOD REGISTER */
/*----------------------------------------------------------------------*/
/* DMA CONTROL REGISTERS                                                */
/*----------------------------------------------------------------------*/
#define SOURCE   4                        /* SOURCE ADDRESS REGISTER      */
#define DEST     6                        /* DESTINATION ADDRESS REGISTER */
#define TRANSFER 8                        /* TRANSFER COUNTER REGISTER    */
/*----------------------------------------------------------------------*/
/* AIC VOLTAGE INPUT CONTROL MACROS                                     */
/*----------------------------------------------------------------------*/
#define THREE_V    1
#define LINE_V     2

/*======================================================================*/
/* STRUCTURES*==========================================================*/
/*======================================================================*/
```

```
/*   AIC COMMAND WORD BITFIELD ENCODING STRUCTURES                          */
/*=========================================================================*/
typedef struct
{
    unsigned int command :2;     /* COMMAND BITS --- SHOULD BE SET TO 00*/
    unsigned int ra      :5;     /* RECEIVE COUNTER A LOAD VALUE        */
    unsigned int d_78    :2;     /* UNUSED - DON'T CARES                */
    unsigned int ta      :5;     /* TRANSMIT COUNTER A LOAD VALUE       */
    unsigned int d_ef    :2;     /* UNUSED - DON'T CARES                */
} AIC_COMMAND_0;

typedef struct
{
    unsigned int command  :2;    /* COMMAND BITS --- SHOULD BE SET TO 01*/
      signed int ra_prime :6;    /* RECEIVE COUNTER DELTA A' LOAD VALUE */
    unsigned int d_8      :1;    /* UNUSED - DON'T CARES                */
      signed int ta_prime :6;    /* RECEIVE COUNTER DELTA A' LOAD VALUE */
    unsigned int d_f      :1;    /* UNUSED - DON'T CARES                */
} AIC_COMMAND_1;

typedef struct
{
    unsigned int command :2;     /* COMMAND BITS --- SHOULD BE SET TO 10*/
    unsigned int rb      :6;     /* RECEIVE COUNTER B LOAD VALUE        */
    unsigned int d_8     :1;     /* UNUSED - DON'T CARES                */
    unsigned int tb      :6;     /* TRANSMIT COUNTER B LOAD VALUE       */
    unsigned int d_f     :1;     /* UNUSED - DON'T CARES                */
} AIC_COMMAND_2;

typedef struct
{
    unsigned int command  :2;    /* COMMAND BITS --- SHOULD BE SET TO 11*/
    unsigned int highpass :1;    /* HIGHPASS FILTER ENABLE              */
    unsigned int loopback :1;    /* LOOPBACK TEST ENABLE                */
    unsigned int aux      :1;    /* AUX INPUT ENABLE                    */
    unsigned int sync     :1;    /* SYNCHRONOUS TRANSMIT/RECEIVE ENABLE */
    unsigned int gain     :2;    /* GAIN SELECTION BITS                 */
    unsigned int d_8      :1;    /* UNUSED - DON'T CARES                */
    unsigned int sinx     :1;    /* SINX/X CORRECTION FILTER ENABLE     */
    unsigned int d_abcdef :1;    /* UNUSED - DON'T CARES                */
} AIC_COMMAND_3;
```

```
/************************************************************************/
/*   C30_2.H                                                            */
/*                                                                      */
/*   TMS320C30 EVALUATION MODULE TMS320C30 SUPPORT FILE 2:              */
/*      : GLOBAL VARIABLES                                              */
/*                                                                      */
/*   (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                */
/************************************************************************/
/* GLOBAL VARIABLES                                                     */
/*======================================================================*/
int buffer;                          /* block size for PC data transfer */
/*----------------------------------------------------------------------*/
/* 32-BIT DMA INTERRUPT ENABLE MASKS                                    */
/*----------------------------------------------------------------------*/
int dma_int0 = 0x00010000;                                    /* INT0   */
int dma_int1 = 0x00020000;                                    /* INT1   */
int dma_int2 = 0x00040000;                                    /* INT2   */
/*----------------------------------------------------------------------*/
/* AIC CONTROL VARIABLES                                                */
/*----------------------------------------------------------------------*/
AIC_COMMAND_0  aic_command_0;                    /* AIC COMMAND WORD 0 */
AIC_COMMAND_1  aic_command_1;                    /* AIC COMMAND WORD 1 */
AIC_COMMAND_2  aic_command_2;                    /* AIC COMMAND WORD 2 */
AIC_COMMAND_3  aic_command_3;                    /* AIC COMMAND WORD 3 */
volatile int send_command       = OFF;  /* FLAG TO SEND AIC COMMAND WORD */
volatile int secondary_transmit = OFF; /* FLAG TO SENT SECONDARY TRANSMIT*/
int aic_secondary = 0;          /* COMMAND TO SENT ON SECONDARY TRANSMIT */
/************************************************************************/
/*         TMS320C30 CONTROL LOCATIONS                                  */
/************************************************************************/
/*----------------------------------------------------------------------*/
/* SERIAL PORT BASE LOCATION                                            */
/*----------------------------------------------------------------------*/
volatile int (*serial_port)[16] = (volatile int (*)[16]) 0x808040;
/*----------------------------------------------------------------------*/
/* TIMER BASE LOCATION                                                  */
/*----------------------------------------------------------------------*/
volatile int (*timer)[16] = (volatile int (*)[16]) 0x808020;
/*----------------------------------------------------------------------*/
/* BUS BASE LOCATION                                                    */
/*----------------------------------------------------------------------*/
volatile int *bus = (volatile int *) 0x808060;
/*----------------------------------------------------------------------*/
/* DMA BASE LOCATION                                                    */
/*----------------------------------------------------------------------*/
volatile int *dma = (volatile int *) 0x808000;
/*----------------------------------------------------------------------*/
/* HOST COMMUNICATION REGISTER LOCATION                                 */
/*      NOTE: ONLY THE PC HOST CAN READ WHAT THE TMS320C30 WRITES.      */
/*            CONVERSELY ONLY THE TMS320C30 CAN READ WHAT THE HOST PC   */
/*            WRITES.                                                   */
/*----------------------------------------------------------------------*/
volatile int *host = (volatile int *) 0x804000;
```

```
/*****************************************************************************

                c30.c

                staff

                10-09-90

          (C) Texas Instruments Inc., 1992

          Refer to the file 'license.txt' included with this
          this package for usage and license information.

*****************************************************************************/
/*****************************************************************************/
/*   C30.C                                                                   */
/*                                                                           */
/*   TMS320C30 EVALUATION MODULE TMS320C30 SUPPORT PROGRAMS                  */
/*                                                                           */
/*   (C) 1990 TEXAS INSTRUMENTS, HOUSTON                                     */
/*****************************************************************************/

/*****************************************************************************/
/*   BUG FIXES:            10/09/90  JR                                      */
/*                                                                           */
/*  1. A fast host would start sending commands to the EVM before it had     */
/*     completed its reset initialization.  Fix is to signal the host on     */
/*     reset initialization complete.  This is done with a function          */
/*     init_host().  This also requires a fix in the host code, pc.c,        */
/*     to poll for this signal.                                              */
/*****************************************************************************/

#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "c30_1.h"              /* FUNCTION PROTOTYPES, MACROS, STRUCTURES */
#include "c30_2.h"              /* GLOBAL VARIABLES                        */
/*===========================================================================*/
/* C_INT11(): DMA INTERRUPT SERVICE ROUTINE                                  */
/*===========================================================================*/
void c_int11(void)
{
    while(*host & 0xFF);            /* WAIT FOR HOST TO WITHDRAW REQUEST */
    *host       = NONE;             /* ACKNOWLEDGE WITHDRAWL OF REQUEST  */
    dma[GLOBAL] = 0;                /* TURN OFF DMA                      */

    asm(" AND 0FFFFh,IE");          /* TURN OFF DMA INTERRUPTS           */
}
/*===========================================================================*/
/* C_INT99(): ERRONEOUS INTERRUPT SERVICE ROUTINE                            */
/*            THIS ROUTINE IDLES AFTER RECEIVING AN UNEXPECTED INTERRUPT     */
/*===========================================================================*/
void c_int99(void)
{
    for(;;);

}
/*===========================================================================*/
/* INIT_EVM(): INITIALIZE TMS320C30 EVALUATION MODULE                        */
/*===========================================================================*/
void init_evm(void)
{
    bus[EXPANSION] = 0x0;           /* ZERO WAIT STATES ON EXPANSION BUS */
    bus[PRIMARY]   = 0x0;           /* ZERO WAIT STATES ON PRIMARY BUS   */
```

```
    asm(" OR 800h,ST");              /* TURN ON CACHE                          */
}
/*=========================================================================*/
/* INIT_AIC(): INITIALIZE COMMUNICATIONS TO AIC                            */
/*             NOTE: i IS A VOLATILE TO FORCE TIME DELAYS AND TO FORCE     */
/*                   READS OF SERIAL PORT DATA RECEIVE REGISTER TO CLEAR   */
/*                   THE RECEIVE INTERRUPT FLAG                            */
/*=========================================================================*/
void init_aic(void)
{
    volatile int i;
    /*---------------------------------------------------------------------*/
    /* SET AIC CONFIGURATION CHIP                                          */
    /*  1. ALLOW 8 KHZ SAMPLING RATE AND 3.6 KHZ ANTIALIASING FILTER       */
    /*     GIVEN A 7.5 MHZ MCLK TO THE AIC FROM A 30 MHZ TMS320C30         */
    /*  2. ENABLE A/D HIGHPASS FILTER                                      */
    /*  3. SET SYNCHRONOUS TRANSMIT AND RECEIVE                            */
    /*  4. ENABLE SINX/X D/A CORRECTION FILTER                             */
    /*  5. SET AIC FOR +/- 1.5 V INPUT                                     */
    /*---------------------------------------------------------------------*/
    aic_command_0.command  = 0;      /* SETUP AIC COMMAND WORD ZERO       */
    aic_command_0.ra       = 13;     /* ADJUST SAMPLING RATE TO 8 kHz     */
    aic_command_0.ta       = 13;     /*    AND 3.6 kHz ANTIALIAS FILTER   */
    aic_command_1.command  = 1;      /* SETUP DEFAULT AIC COMMAND WORD 1  */
    aic_command_1.ra_prime = 1;
    aic_command_1.ta_prime = 1;
    aic_command_1.d_f      = 0;
    aic_command_2.command  = 2;      /* SETUP DEFAULT AIC COMMAND WORD 2  */
    aic_command_2.rb       = 36;
    aic_command_2.tb       = 36;
    aic_command_3.command  = 3;
    aic_command_3.highpass = ON;     /* TURN ON INPUT HIGHPASS FILTER     */
    aic_command_3.loopback = OFF;    /* DISABLE AIC LOOPBACK              */
    aic_command_3.aux      = OFF;    /* DISABLE AUX INPUT                 */
    aic_command_3.sync     = ON;     /* ENABLE SYNCHRONOUS A/D AND D/A    */
    aic_command_3.gain     = LINE_V; /* SET FOR LINE-LEVEL INPUT          */
    aic_command_3.sinx     = ON;     /* ENABLE SIN x/x CORRECTION FILTER  */
    /*---------------------------------------------------------------------*/
    /* CONFIGURE TIMER 0 TO ACT AS AIC MCLK                                */
    /* THE TIMER IS CONFIGURED IN THE FOLLOWING WAY                        */
    /*         1. THE TIMER'S VALUE DRIVES AN ACTIVE-HIGH TCLK 0 PIN       */
    /*         2. THE TIMER IS RESET AND ENABLED                           */
    /*         3. THE TIMER'S IS IN PULSE MODE                             */
    /*         4. THE TIMER HAS A PERIOD OF TWO INSTRUCTION CYCLES         */
    /*---------------------------------------------------------------------*/
    timer[0][PERIOD] = 0x1;
    timer[0][GLOBAL] = 0x2C1;
    /*---------------------------------------------------------------------*/
    /* CONFIGURE SERIAL PORT 0                                             */
    /*            1. EXTERNAL FSX, FSR, CLKX, CLKR                         */
    /*            2. VARIABLE DATA RATE TRANSMIT AND RECEIVE               */
    /*            3. HANDSHAKE DISABLED                                    */
    /*            4. ACTIVE HIGH DATA AND CLK                              */
    /*            5. ACTIVE LOW FSX,FSR                                    */
    /*            6. 16 BIT TRANSMIT AND RECEIVE WORD                      */
    /*            7. TRANSMIT INTERRUPT                                    */
    /*            8. RECEIVE INTERRUPT ENABLED/RECEIVE                     */
    /*            9. FSX, FSR, CLKX, CLKR, DX, DR CONFIGURED AS SERIAL     */
    /*               PORT PINS                                             */
    /*---------------------------------------------------------------------*/
    serial_port[0][X_PORT] = 0x111;
    serial_port[0][R_PORT] = 0x111;
```

```
    asm(" LDI 2,IOF");          /* RESET AIC BY PULLING XF0 ACTIVE-LOW    */

    for(i = 0; i < 50; i++);    /* KEEP RESET LOW FOR SOME PERIOD OF TIME */
    serial_port[0][GLOBAL] = 0x0e970300; /* WRITE SERIAL PORT CONTROL     */
    serial_port[0][X_DATA] = 0x0;        /* CLEAR SERIAL TRANSMIT DATA    */

    asm(" LDI 6,IOF");                   /* PULL AIC OUT OF RESET         */

    asm(" LDI     0,IF");/* CLEAR ANY INTERRUPT FLAGS                     */
    asm(" LDI  410h,IE");/* ENABLE DMA & SERIAL PORT0 TRANSMIT INTERRUPTS*/
    asm(" OR  2000h,ST");/* SET GLOBAL INTERRUPT ENABLE BIT               */
    /*-------------------------------------------------------------------*/
    /* MODIFY AIC CONFIGURATION                                          */
    /*-------------------------------------------------------------------*/
    configure_aic(*((int *) &aic_command_0));
    configure_aic(*((int *) &aic_command_3));
}
/*************************************************************************/
/* The following function fixes an initialization timing bug. It does so */
/* by signalling the host that initialization is complete.               */
/*************************************************************************/

/*=======================================================================*/
/* INIT_HOST(): INITIALIZE HOST INTERFACE                                */
/*=======================================================================*/
void init_host(void)
{
    *host = NONE;                       /* SIGNAL INITIALIZATION COMPLETE   */
}
/*=======================================================================*/
/* CONFIGURE_AIC(): INITIATE AIC CONFIGURATION WORD TRANSMISSION ON NEXT */
/*                  INTERRUPT AFTER ALL PREVIOUS COMMANDS ARE SENT       */
/*=======================================================================*/
void configure_aic(int i)
{
    while(send_command || secondary_transmit);
    aic_secondary = i;
    send_command = ON;
}
/*=======================================================================*/
/* RECORDING(): SEND DATA FOR STORAGE ON PC                              */
/*=======================================================================*/
void recording(int *source,int rec_cmd)
{
    asm(" AND     0FFF8h,IF");    /* CLEAR ANY PENDING HOST INTERRUPTS*/
    asm(" OR  @_dma_int2,IE");    /* SYNCHRONIZE DMA WITH PC READ INT */

    dma[SOURCE]   = (int) source;       /* READ FROM CODEWORD          */
    dma[DEST]     = (int) host;         /* WRITE TO HOST               */
    dma[TRANSFER] = buffer;             /* TRANSFER ENTIRE STRUCTURE   */
    /*-----------------------------------------------------------------*/
    /* SETUP DMA FOR RECORDING                                         */
    /*       1. INCREMENT SOURCE ADDRESS ONLY ON EACH TRANSFER         */
    /*       2. SYNCHRONIZE WRITES WITH DATA READ INTERRUPT FROM PC     */
    /*          FROM PC                                                */
    /*       3. TRANSFER COUNTER STOPS UPON REACHING ZERO              */
    /*       4. THE DMA INTERRUPT SOURCE IS ENABLED                    */
    /*-----------------------------------------------------------------*/
    dma[GLOBAL] = 0x0E13;
    *host       = rec_cmd;              /* ACKNOWLEDGE REQUEST         */
}
/*=======================================================================*/
/* PLAYING(): RECEIVE RECORDED DATA FROM PC                              */
/*=======================================================================*/
```

```
void playing(int *dest,int play_cmd)
{
    asm(" AND     0FFF8h,IF");     /* CLEAR ANY PENDING HOST INTERRUPTS*/
    asm(" OR  @_dma_int1,IE");     /* SYNCHRONIZE DMA WITH PC WRITE INT*/

    dma[SOURCE]   = (int) host;         /* READ FROM HOST              */
    dma[DEST]     = (int) dest;         /* WRITE TO CODEWORD           */
    dma[TRANSFER] = buffer;             /* TRANSFER ENTIRE STRUCTURE   */
    /*-------------------------------------------------------------------*/
    /* SETUP DMA FOR PLAYING                                             */
    /*     1. INCREMENT DESTINATION ADDRESS ONLY ON EACH TRANSFER        */
    /*     2. SYNCHRONIZE WRITES WITH DATA WRITE INTERRUPT FROM PC       */
    /*     3. TRANSFER COUNTER STOPS UPON REACHING ZERO                  */
    /*     4. THE DMA INTERRUPT SOURCE IS ENABLED                        */
    /*-------------------------------------------------------------------*/
    dma[GLOBAL] = 0x0D43;
    *host       = play_cmd;             /* ACKNOWLEDGE REQUEST         */
}
```

```
 # define Nv N/* surch frame length*/
 # define M 4/* Number of excitation pulses */
 # define SF 4/* Number of sub frames */
 # define F 160/* SF*N speech frame length */
 # define Fv F /* F+Nv-N speech frame length with overlap */
 # define L F+2*N/* F+2*N lp analysis frame length */
 double a[p];/* lp coefficients */
 double q[p];/* lsp coefficients */
 double f1[hp+1];/* used in lsp computation */
 double f2[hp+1];/* used in lsp computation */
 double l1[hp+1];/* used in lsp computation */
 double l2[hp+1];/* used in lsp computation */

 double  u[Nv];/* used to store the synthesis filter pulse response */
 BOOL st[Nv];/* st[n] indicate if u[n] has already been computed */
 double phii[Nv];/* pulse response autocorrelation */
 double inv0;/* used to store 1/phii[0] */
 int pitch=1000;/* pitch */
 double gain=0;/* LTP gain */
 double b[M];/* pulses amplitudes */
 #define norm 1.221E-4 /* to normalise pulses amplitudes */
 #define scale 8190.01 /* to restore pulses amplitudes scale */
 int mp[M];/* pulses positions */
 double w=0.7;/* weighting parameter */
 double pw[p];/* used to store pow(w,j) */
 double aw[p];/* used to store aw=a(j)*pow(w,j),updated each frame */
 double tw[L];/*tapered window, must be computed at program start &
parameters change.*/
 # define tpi 6.28318530717959

 double r_b[Fv];/* lpc residual */
 double * r=r_b;/* lpc residual */
 double * xa;/* pointer to lpc analysis frame */
 double * xf;/* pointer to speech frame */


/*-------------------------------------------------------------------------*/
 #define OVERLAP L - F       /* OVERLAP BETWEEN FRAMES                      */
/*=========================================================================*/
/*          DATA ACQUISITION VARIABLES                                     */
/*=========================================================================*/

 double out1[Fv];           /* OUTPUT SAMPLE ARRAYS                        */
 double out2[Fv];
 double in1[L];             /* INPUT SAMPLE ARRAYS                         */
 double in2[L];

 double *aic_in  = in1;    /* POINTER TO INPUT ARRAY FOR INPUT FROM AIC  */
 double *s_in    = in2;    /* POINTER TO INPUT ARRAY FOR ANALYSIS        */
 double *aic_out = out1;   /* POINTER TO OUTPUT ARRAY FOR OUTPUT TO AIC  */
 double *s_out   = out2;   /* POINTER TO OUTPUT ARRAY FOR SYNTHESIS      */

 volatile int sample = 0;  /* INDEX FOR AIC I/O ARRAYS                    */

/*=========================================================================*/
/* LSP codes in cosinus domain                                             */
/*=========================================================================*/

 const double dq=7.5e-04;/* min lsp distance */
 const double qmin=-0.999978;/* lsp min */
 const double qmax= 0.999978;/* lsp max */
```

```
/*************************************************************************/
/*   AZIZ  AHMED SAID     09/2002                                        */
/*   MPE_C30.H                                                           */
/*                                                                       */
/*   LINEAR PREDICTIVE CODING: PROTOTYPES, MACROS, GLOBALS, STRUCTURES   */
/*                                                                       */
/*************************************************************************/
/* FUNCTION PROTOTYPES                                                   */
/*=======================================================================*/
#define BOOL int

void main(void);
void c_int05(void);
void wait_frame(void);
BOOL init_arrays(void);
BOOL wld(double * x,int Ls);
BOOL ltp(double * r,int Ls);
BOOL window(double * x,int Ls);
BOOL residual(int Ls);
BOOL wsfilter(int Ls);
BOOL sfilter(int Ls);
BOOL mxm(double * x,int Ls,int *pos,double *max);
double phi(int i,int j,int Ls);
double phi2(int i,int j,int Ls);
double h(int n);
BOOL mpe(int offset,int Ls);
BOOL order();
BOOL synt(int offset,int Ls);
BOOL lsp();
double c(double x,double * f);
BOOL LsptoLP();
BOOL Qlsp();
BOOL Qpulses(int);
BOOL Q(int N,int M,double * v,double * cv);
int QSV(int Mv,double * v,const double * cv);
BOOL DECODE_lsp();
BOOL DECODE_pulses(int n);
/*=======================================================================*/
/* EXTERNAL VARIABLES                                                    */
/*=======================================================================*/
/*-----------------------------------------------------------------------*/
/* AIC CONTROL VARIABLES                                                 */
/*-----------------------------------------------------------------------*/
extern AIC_COMMAND_0  aic_command_0;              /* AIC COMMAND WORD 0 */
extern AIC_COMMAND_1  aic_command_1;              /* AIC COMMAND WORD 1 */
extern AIC_COMMAND_2  aic_command_2;              /* AIC COMMAND WORD 2 */
extern AIC_COMMAND_3  aic_command_3;              /* AIC COMMAND WORD 3 */
extern volatile int send_command;       /* FLAG TO SEND AIC COMMAND WORD  */
extern volatile int secondary_transmit;/* FLAG TO SENT SECONDARY TRANSMIT*/
extern int aic_secondary;       /* COMMAND TO SENT ON SECONDARY TRANSMIT */
/*-----------------------------------------------------------------------*/
/* SERIAL PORT BASE LOCATION                                             */
/*-----------------------------------------------------------------------*/
extern volatile int (*serial_port)[16];
/*-----------------------------------------------------------------------*/
/* DMA BASE LOCATION                                                     */
/*-----------------------------------------------------------------------*/
extern volatile int *dma;
/*-----------------------------------------------------------------------*/

/*---------Global vars for MPELPC-----------*/
 # define p 10/* prediction order */
 # define hp p/2/* half prediction order*/
 # define N 40/* sub frame length*/
```

```
/*--------------------------*/
/*         LSP1             */
/*--------------------------*/
const double lsp1_codes[8]=
{
     9.2268e-001,  9.4579e-001,  9.5979e-001,  9.7014e-001,
     9.7784e-001,  9.8412e-001,  9.9003e-001,  9.9815e-001
};
/*--------------------------*/
/*      DLSP2=LSP1-LSP2     */
/*--------------------------*/
const double lsp2_codes[8]=
{
     9.9093e-003,  2.0654e-002,  3.3680e-002,  4.9885e-002,
     6.8662e-002,  9.3436e-002,  1.2320e-001,  1.6388e-001
};
/*--------------------------*/
/*      DLSP3=LSP2-LSP3     */
/*--------------------------*/
const double lsp3_codes[8]=
{
     2.6999e-002,  4.9183e-002,  7.3754e-002,  1.0478e-001,
     1.3935e-001,  1.7733e-001,  2.2419e-001,  3.1016e-001
};
/*--------------------------*/
/*      DLSP4=LSP3-LSP4     */
/*--------------------------*/
const double lsp4_codes[8]=
{
     6.5002e-002,  1.1363e-001,  1.5808e-001,  1.9798e-001,
     2.3672e-001,  2.8249e-001,  3.4588e-001,  4.3979e-001
};
/*--------------------------*/
/*      DLSP5=LSP4-LSP5     */
/*--------------------------*/
const double lsp5_codes[16]=
{
     6.8199e-002,  1.1979e-001,  1.6227e-001,  2.0021e-001,
     2.3358e-001,  2.6686e-001,  3.0145e-001,  3.3257e-001,
     3.6576e-001,  4.0004e-001,  4.3859e-001,  4.8312e-001,
     5.2813e-001,  5.9080e-001,  6.7308e-001,  7.8962e-001
};
/*--------------------------*/
/*      DLSP6=LSP5-LSP6     */
/*--------------------------*/
const double lsp6_codes[16]=
{
     6.3107e-002,  9.6916e-002,  1.2767e-001,  1.6380e-001,
     1.9883e-001,  2.3076e-001,  2.5802e-001,  2.8393e-001,
     3.0992e-001,  3.3600e-001,  3.6777e-001,  4.0450e-001,
     4.4904e-001,  5.0804e-001,  6.0151e-001,  7.4834e-001
};
/*--------------------------*/
/*         DLSP7            */
/*--------------------------*/
const double lsp7_codes[8]=
{
     -5.4739e-001, -4.4617e-001, -3.8667e-001, -3.3824e-001,
     -2.8900e-001, -2.3217e-001, -1.6234e-001, -4.7741e-002
};

/*--------------------------*/
/*         LSP8             */
/*--------------------------*/
```

```
const double lsp8_codes[8]=
{
     -7.2373e-001, -6.5204e-001, -6.0729e-001, -5.7291e-001,
     -5.3552e-001, -4.9020e-001, -4.2544e-001, -3.0691e-001
};
/*-------------------------*/
/*         LSP9            */
/*-------------------------*/
const double lsp9_codes[8]=
{
     -8.9355e-001, -8.4981e-001, -8.2308e-001, -7.9907e-001,
     -7.7390e-001, -7.4107e-001, -6.9535e-001, -6.1245e-001
};
/*-------------------------*/
/*         LSP10           */
/*-------------------------*/
const double lsp10_codes[8]=
{
     -9.5920e-001, -9.4399e-001, -9.3154e-001, -9.1826e-001,
     -9.0191e-001, -8.7565e-001, -8.3765e-001, -7.8011e-001
};

/*-------------------------------------------------------------------------*/
/*                      Pulses Amplitudes codes                            */
/*=========================================================================*/
const int CBS=8;
double b_codes[8]=
{
     -3.3690e-001, -1.7011e-001, -8.4252e-002, -2.9704e-002,
     -4.0209e-004,  3.5581e-002,  1.0061e-001,  2.1059e-001
};

/*=========================================================================*/
/* MPE_LPC BIT-FIELD ENCODING STRUCTURES                                   */
/*=========================================================================*/
/* BIT STRUCTURES HAVE BEEN ARRANGED TO OCCUPY 16 BITS PER CODEWORD        */
/*=========================================================================*/

typedef struct
{
        unsigned int lsp1:3;
        unsigned int lsp2:3;
        unsigned int lsp3:3;
        unsigned int lsp4:3;
        unsigned int lsp5:4;
} CODEWORD1;

typedef struct
{
        unsigned int lsp6:4;
        unsigned int lsp7:3;
        unsigned int lsp8:3;
        unsigned int lsp9:3;
        unsigned int lsp10:3;
} CODEWORD2;

typedef struct
{
        unsigned int b1:3;
        unsigned int b2:3;
        unsigned int m1:5;
        unsigned int m2:5;
} PULSES12;
```

```
typedef struct
{
        unsigned int b3:3;
        unsigned int b4:3;
        unsigned int m3:5;
        unsigned int m4:5;
} PULSES34;
typedef struct
{
      PULSES12 P12;
      PULSES34 P34;

} CODEWORD3;
typedef struct
{
      CODEWORD1 cw1;
      CODEWORD2 cw2;
      CODEWORD3 cw3[SF];/* 'SF' = for each sub frame */
} CODE;
CODE code;
```

```
/*************************************************************************/
/*   AZIZ  AHMED SAID    09/2002                                         */
/*                                                                       */
/*   MPE_C30.C                                                           */
/*                                                                       */
/*   LINEAR PREDICTIVE CODING: MPE-LPC ON THE TMS320C30                  */
/*                                                                       */
/*   DESIGNED TO RUN ON THE TMS320C30 EVALUATION MODULE (EVM)            */
/*                                                                       */
/*************************************************************************/

#include <math.h>
#include <stdlib.h>
#include "c30_1.h"  /* GENERAL EVM MACROS,STRUCTURES,FUNCTION PROTORYPES */
#include "mpe_c30.h" /* PROTOTYPES,GLOBALS,STRUCTURES,MACROS for MPE_LPC */

#define C_ONLY ON  /* USE C-CODED FUNCTIONS ONLY */
/*=======================================================================*/
/* MAIN()                                                                */
/*      MAIN PROGRAM LOOP                                                */
/*      FULL DUPLEX OPERATION                                            */
/*=======================================================================*/
void main(void)
{
      int n,sf,g,i,j;
      double c,f;
    init_evm();               /* INITIALIZE TMS320C30 AND EVM           */
    init_arrays();            /* INITIALIZE MPE ARRAYS                  */
    init_aic();               /* INITIALIZE TLC32044 AIC COMMUNICATIONS */
    for(;;)
    {
      wait_frame();           /* WAIT FOR FULL FRAME OF DATA            */
      /*=====================================================================*/
                  xa=s_in;
                  xf=s_out;
                  /* copy speech frame from lpc frame */
                  memcpy(xf ,xa + N , Fv);
                  window(xa,L);/* xa*window */
                  wld(xa,L);/* computation of lp coefficients */
                  lsp();/* computation of lsp */
                  Qlsp();/* quantization & encoding of lsp */
                  DECODE_lsp();/* decoding of lsp */
                  LsptoLP();/* compute lp coefs from quantized lsp */
                  residual(Fv);/* computation of the residual signal */
                  wsfilter(Fv);/* computation of the weighted speech */
                  /*- computation of phii */
                  phii[0]=phi(0,0,Nv);
                  inv0=1/phii[0];
                  for(i=1;i<Nv;i++)
                  {
                        phii[i]=phi2(i,0,Nv);
                  }
                  /*-------------------------*/
                  for(g=0,sf=0;sf<SF;sf++,g+=N)
                  {
                        mpe(g,Nv);/* computation of excitation pulses */
                        Qpulses(sf);/* quantization & encoding of pulses */
                        DECODE_pulses(sf);/* Decoding of pulses */
                        synt(g,N);/* synthesis of the sub frame*/
                  }
                  /*-------------------------*/
                  sfilter(F);/*Compute output speech from weighted speech*/

      /*=====================================================================*/
```

```
    }
}
/*=========================================================================*/
/* C_INT05()                                                               */
/*        SERIAL PORT 0 TRANSMIT INTERRUPT SERVICE ROUTINE                 */
/*        1. IF SECONDARY TRANSMISSION SEND AIC COMMAND WORD               */
/*        2. OTHERWISE IF COMMAND SEND REQUESTED SETUP FOR SECONDARY       */
/*           TRANSMISSION ON NEXT INTERRUPT                                */
/*        3. OTHERWISE WRITE OUT OUTPUT DATA AND READ IN INPUT DATA        */
/*        4. RESET SAMPLE INDEX IF FRAME IS FULL                           */
/*=========================================================================*/
void c_int05(void)
{
    if (secondary_transmit)
    {
      serial_port[0][X_DATA] = aic_secondary;
      secondary_transmit     = OFF;
    }
    else if (send_command)
    {
      serial_port[0][X_DATA] = 3;
      secondary_transmit     = ON;
      send_command           = OFF;
    }
    else
    {
      serial_port[0][X_DATA]    = (int) aic_out[sample]  << 2;
      aic_in[sample + OVERLAP]  = serial_port[0][R_DATA] << 16 >> 18;
      if (++sample == F) sample = 0;
    }
}
/*=========================================================================*/
/* INIT_ARRAYS()                                                           */
/*               1. SETUP HAMMING WINDOW                                   */
/*               2. CLEAR DATA ARRAYS                                      */
/*=========================================================================*/
BOOL init_arrays(void)
{
      int n,i,j;/* general vars */
      double f;/* general vars */



      f=tpi/(L-1);
      for(n=0;n<L;n++)
      {
            tw[n]=0.54-0.46*cos(f*n);
      }



      for(pw[0]=w,i=0,j=1;j<p;j++)
      {
            pw[j]=pw[i]*w;
            i=j;
      }

      u[0]=1;
      st[0]=1;

      for(n=1;n<Nv;n++)
      {
            st[n]=0;
```

```
        }
        for(n=0;n<L;n++)
        {
              s_in[n]  =0;
              aic_in[n]=0;
        }
        for(n=0;n<F;n++)
        {
              s_out[n]  =0;
              aic_out[n]=0;
        }
        return 0;/*no error*/
}
/*=========================================================================*/
/* WAIT_FRAME()                                                            */
/*              WAIT FOR A NEW FRAME OF DATA SWAP ACQUISITION              */
/*              AND PROCESSING ARRAYS                                      */
/*=========================================================================*/
void wait_frame(void)
{
    double *ptr;                                    /* SWAPPING VARIABLE   */

    while(sample);                                 /* WAIT FOR A NEW FRAME */
    ptr     = s_out;                               /* SWAP OUTPUT ARRAYS   */
    s_out   = aic_out;
    aic_out = ptr;
    ptr     = aic_in;                                 /* SWAP INPUT ARRAYS */
    aic_in  = s_in;
    s_in    = ptr;
    memcpy(aic_in ,s_in + F , OVERLAP);  /* COPY OVERLAP BETWEEN WINDOWS */
}
/*=========================================================================*/
/* window(double * x,int Ls)                                               */
/*               WINDOWS BUFFER X OF LENGTH LS                             */
/*=========================================================================*/
BOOL window(double * x,int Ls)
{
      int n;/* general vars */

      for(n=0;n<Ls;n++)
      {
            x[n]*=tw[n];
      }

      return 0;/*no error*/

}
/*=========================================================================*/
/* wld(double * x,int Ls)                                                  */
/*            COMPUTATION OF LP COEFFICIENTS USING WLD ALGORITHME          */
/*                    FROM VECTOR X OF LENGTH LS                           */
/*=========================================================================*/
BOOL wld(double * x,int Ls)
{

      int n,i,j,m,u,g;/* general vars */
      long double c;/* general vars */
      long double rs[p+1];/* signal autocor */
      long double  e[p+1];
      long double ai[p][p];/* a(i,j)=ai[i][j] */
      long double    k[p];/* reflection coefficients */
      m=p+1;
```

```
        /*1- computation of autocor */

        for(j=0;j<m;j++)
        {
                c=0;
                for(n=j;n<Ls;n++)
                {
                        c+=x[n]*x[n-j];
                }
                rs[j]=c;
        }

        /*2- start computing lp coefs */

        for(e[0]=rs[0],i=1;i<m;i++)
        {
                c=0;
                n=i-1;
                g=i-2;


                for(j=n,u=0;j>0;j--,u++)
                {
                        c+=ai[g][u]*rs[j];

                }

                k[n]=(rs[i]-c)/e[n];


                ai[n][n]=k[n];

                for(u=0,j=1;j<i;j++)
                {
                        ai[n][u]=ai[g][u]-k[n]*ai[g][n-j];
                        u=j;
                }

                c=k[n];

                e[i]=(1-c*c)*e[n];
        }

        u=p-1;

        /*3- copy of final solution */
        for(j=0;j<p;j++)
        {
                a[j]=(double) ai[u][j];
                aw[j]=pw[j]*a[j];
        }


        return 0;/*no error*/
}
/*=========================================================================*/
/* residual(int Ls)                                                        */
/*      COMPUTATION OF THE RESIDUAL SIGNAL r FROM VECTOR xf OF LENGTH Ls */
/*=========================================================================*/
BOOL residual(int Ls)
{
        int n,i,j,g;/* general vars */
        double c;
```

```
	for(n=0;n<Ls;n++)
	{
		c=xf[n];

		if(n<p) g=n+1;
		else g=p+1;

		for(i=0,j=1;j < g;j++)
		{
			c-=a[i]*xf[n-j];
			i=j;
		}
		r[n]=c;

	}

	return 0;/*no error*/

}

/*=========================================================================*/
/* wsfilter(int Ls)                                                        */
/*  COMPUTATION OF THE WEIGHTED SPEECH FROM THE RESIDUAL r OF LENGTH Ls    */
/*=========================================================================*/
BOOL wsfilter(int Ls)
{
	int n,i,j,g;/* general vars */
	double c;


	for(n=0;n<Ls;n++)
	{
		c=r[n];

		if(n<p) g=n+1;
		else g=p+1;

		for(i=0,j=1;j<g;j++)
		{
			c+=aw[i]*xf[n-j];
			i=j;
		}
		xf[n]=c;
	}


	return 0;/*no error*/

}
/*=========================================================================*/
/* mxm(double * x,int Ls,int * pos,double * max)                           */
/*          GIVE THE AMPLITUDE (max) AND THE INDEX (pos)                   */
/*            OF THE MAXIMUM OF VECTOR X OF LENGTH Ls                      */
/*=========================================================================*/
BOOL mxm(double * x,int Ls,int * pos,double * max)
{
	int n;/* general vars */
	double c;/* general vars */
	int ps;
	double mx;

	mx=x[0];
```

```
        ps=0;

        for(n=0;n<Ls;n++)
        {
                c=x[n];
                if(mx<c)
                {
                        mx=c;
                        ps=n;
                }
        }

        *pos=ps;
        *max=mx;

        return 0;/*no error*/

}
/*=========================================================================*/
/* h(int n)                                                                */
/*              RETURN THE nTH SAMPLE OF THE GLOBAL SYNTHESIS FILTER       */
/*                                  PULSE RESPONSE                         */
/*=========================================================================*/
double h(int n)
{
        int i,j;/* general vars */
        double v;

        if(n<0)
        {
                v=0;
                goto end;
        }
        if(n<pitch)
        {
                if(n==0) v=1;
                else
                {
                        if(st[n])
                                v=u[n];
                        else
                        {
                                v=0;
                                i=0;
                                for(j=1;j<=p;j++)
                                {
                                        v+=aw[i]*h(n-j);
                                        i=j;
                                }
                                u[n]=v;
                                st[n]=1;
                        }
                }

        }
        else
        {
                if(st[n])
                        v=u[n];
                else
                {
                        v=gain*h(n-pitch);
                        i=0;
                        for(j=1;j<=p;j++)
```

```
                {
                        v+=aw[i]*(h(n-j)-gain*h(n-j-pitch));
                        i=j;
                }
                u[n]=v;
                st[n]=1;
        }
    }
end:
    ;

    return v;/*no error*/

}
/*=========================================================================*/
/* phi(int i,int j,int Ls)                                                 */
/*     COMPUTATION OF THE AUTOCORRELATION OF THE GLOBAL SYNTHESIS FILTER   */
/*                          PULSE RESPONSE AT INDEXES i & j                */
/*                          Ls IS THE NB OF SAMPLES                        */
/*=========================================================================*/
double phi(int i,int j,int Ls)
{
    int n,k;/* general vars */
    double c;/* general vars */
    c=0;

        k=(i-j);
        if(k<0) k=-k;
        for(n=k;n<Ls;n++)
        {
                c+=h(n)*h(n-k);
        }

    return c;/*no error*/

}
double phi2(int i,int j,int Ls)
{
    int n,k;/* general vars */
    double c;/* general vars */
    c=0;

        k=(i-j);
        if(k<0) k=-k;
        for(n=k;n<Ls;n++)
        {
                c+=u[n]*u[n-k];
        }

    return c;/*no error*/

}
/*=========================================================================*/
/* mpe(int offset,int Ls)                                                  */
/*         COMPUTATION OF THE EXITATION PULSES AMPLITUDES AND POSITIONS    */
/*                          Ls IS THE EXCITATION FRAME LENGTH              */
/*                          offset IS THE START INDEX OF THE EXCITATION    */
/*                          FRAME                                          */
/*=========================================================================*/
BOOL mpe(int offset,int Ls)
{
    int n,i,j,m,g,k;/* general vars */
    double c;/* general vars */
```

```
        double psi[M][Nv];
        double d[Nv];
        double * x;

        x=xf+offset;

        /*1- computation of b[0] & mp[0] */

        /* computation of psi[0] */
        for(i=0;i<Ls;i++)
        {
              c=0;
              for(n=i;n<Ls;n++)
              {
                    c+=x[n]*u[n-i];/* h(n-i) */
              }
              psi[0][i]=c;/* j=0 */
        }
        /* computation of square(psi[0]) */
        for(i=0;i<Ls;i++)
        {
              c=psi[0][i];
              d[i]=c*c;/**/
        }
        /* find max square(psi[0])=mp[0] */
        mxm(d,Ls,mp,b);
        /* computation of b[0] */
        b[0]=psi[0][mp[0]]*inv0;

        /*2- step 1 to M-1 : computation of b[j] & mp[j] */

        for(n=0,j=1;j<M;j++)
        {
              /* update psi[j] */
              for(i=0;i<Ls;i++)
              {
                    k=mp[n]-i;
                    if(k<0) k*=-1;
                    psi[j][i]=psi[n][i]-b[n]*phii[k];
              }
              /* clear psi[j] where there is already a pulse */
              for(i=0;i<j;i++)
              {
                    psi[j][mp[i]]=0;
              }
              /* computation of square(psi[j]) */
              for(i=0;i<Ls;i++)
              {
                    c=psi[j][i];
                    d[i]=c*c;
              }
              /* find max square(psi[j])=mp[j] */
              mxm(d,Ls,&mp[j],&b[j]);
              /* computation of b[j] */
              b[j]=psi[j][mp[j]]*inv0;
              n=j;

        }

        /*3- Put  pulses in ascending position order */
        order();

        return 0;/* no error */
}
```

```
/*=============================================================================*/
/* BOOL order()                                                                */
/*                  Put  pulses in ascending position order                    */
/*=============================================================================*/
BOOL order()
{
      int n,i,rp,m,index;
      double rb;

      for(i=0;i<M;i++) /* for each position */
      {
            m=Nv;

            for(n=i;n<M;n++) /* find the max pulse index */
            {
                  if(mp[n]<m)
                  {
                        index=n;
                        m=mp[n];
                  }
            }
            /* rotation */
            rp=mp[i];
            mp[i]=mp[index];
            mp[index]=rp;
            rb=b[i];
            b[i]=b[index];
            b[index]=rb;
      }
      return 0;
}

/*=============================================================================*/
/* synt(int offset,int Ls)                                                     */
/*        SYNTHESIS OF SPEECH USING THE EXCITATION PULSES                      */
/*                          Ls IS THE SUB FRAME LENGTH                         */
/*                          offset IS THE START INDEX OF THE SUB FRAME         */
/*=============================================================================*/
BOOL synt(int offset,int Ls)
{
      int n,i,j;/* general vars */
      double * x,c;

      x=xf+offset;


      for(n=0;n<Ls;n++)
      {
            c=0;
            for(j=0;j<M;j++)
            {
                  i=n-mp[j];
                  c+=b[j]*h(i);
            }
            x[n]=c;
      }

      return 0;/*no error*/
}
/*=============================================================================*/
/* sfilter(int Ls)                                                             */
/*        FILTER THE WEIGHTED SPEECH TO GIVE THE OUTPUT SPEECH                 */
/*        Ls IS THE SPEECHE FRAME LENGTH                                       */
/*=============================================================================*/
```

```
BOOL sfilter(int Ls)
{
      int n,i,j,g;/* general vars */
      double c;


      for(n=0;n<Ls;n++)
      {
            c=xf[n];
            i=0;
            if(n<p) g=n+1;
            else g=p+1;
            for(j=1;j < g;j++)
            {
                  c-=aw[i]*xf[n-j];
                  i=j;
            }
            r[n]=c;

      }

      for(n=0;n<Ls;n++)
      {
            c=r[n];
            i=0;
            if(n<p) g=n+1;
            else g=p+1;
            for(j=1;j<g;j++)
            {
                  c+=a[i]*xf[n-j];
                  i=j;
            }
            xf[n]=c;
      }


      return 0;/*no error*/

}
/*=========================================================================*/
/* BOOL lsp()                                                              */
/*           computation of lsps from lp coefficients                      */
/*=========================================================================*/
BOOL lsp()
{
      double h=0.02,nb=5,x,y,xi,al,bh;
      double c1,c2,*f;
      int i,j;
      BOOL odd=0;

      for(f1[0]=f2[0]=1,i=0;i<hp;i++)
      {
            c1=a[i];
            c2=a[p-i-1];
            f1[i+1]=-c1+c2+f1[i];
            f2[i+1]=-(c1+c2+f2[i]);
      }

      for(xi=1,i=0;i<p;odd= odd ^ 1,xi=q[i],i++)
      {

            if(odd) f=f1;
            else  f=f2;
            c1=c(xi,f);
```

```
            for(x=xi-h;x>-1;x-=h)
            {
                  c2=c(x,f);
                  if(c2==0) /* lsp found */
                  {
                        q[i]=x;
                        break;
                  }
                  if(c1*c2<0) /* x<lsp<x+h */
                  {
                        /* Do nb bisections to refine the lsp */
                        for(al=x,bh=x+h,c1=c(bh,f),j=0;j<nb;j++)
                        {
                              y=0.5*(al+bh);
                              c2=c(y,f);
                              if(c2==0) {q[i]=y;goto next;}
                              if(c1*c2<0) al=y; /* y<qi<bh */
                              else /* al<qi<y */
                              {
                                    bh=y;
                                    c1=c(bh,f);
                              }
                        }
                        q[i]=0.5*(al+bh);goto next;/* error < h/2^(nb+1) */
                  }
            }
next: ;
      }

      if(i==p) return 0; /* all lsp have been found */
      else return 1;

}
double c(double x,double * f)
{
      int k;

      for(l1[5]=0,l1[4]=1,k=4;k>0;k--)
      {
            l1[k-1]=2*x*l1[k]-l1[k+1]+f[hp-k];
      }
      return (x*l1[0]-l1[1]+f[5]*0.5);
}
/*=========================================================================*/
/* BOOL LsptoLP()                                                          */
/*          transform lsps to lp coefficients                              */
/*=========================================================================*/
BOOL LsptoLP()
{
      double d1,d2;
      int i,j,i1,i2,j1,j2,ti;


      l1[0]=1;
      l1[1]=-2*(q[1]);

      l2[0]=1;
      l2[1]=-2*(q[0]);

      for(i=2;i<6;i++)
      {
```

```
            i1=i-1;
            i2=i-2;
            ti=2*i1;
            d1=-2*(q[ti+1]);
            d2=-2*(q[ti]);
            l1[i]=d1*l1[i1]+2*l1[i2];
            l2[i]=d2*l2[i1]+2*l2[i2];

            for(j=i1;j>1;j--)
            {
                  j1=j-1;
                  j2=j-2;
                  l1[j]=l1[j]+d1*l1[j1]+l1[j2];
                  l2[j]=l2[j]+d2*l2[j1]+l2[j2];
            }
            l1[1]=l1[1]+d1*l1[0];
            l2[1]=l2[1]+d2*l2[0];
      }
      for(i=1;i<6;i++)
      {
            i1=i-1;
            f1[i]=l1[i]-l1[i1];
            f2[i]=l2[i]+l2[i1];
      }
      for(i=1;i<6;i++)
      {
            a[i-1]=-(f1[i]+f2[i])*0.5;
            a[i+4]=(f1[6-i]-f2[6-i])*0.5;
      }
      return 0;

}
/*=========================================================================*/
/* BOOL Q(int N1,int N2,double * v,double * cv)                            */
/*           Quantize vector v using code vector cv                        */
/*           N1 = v length                                                 */
/*           N2 = cv length                                                */
/*=========================================================================*/
BOOL Q(int N1,int N2,double * v,double * cv)
{
      int n,i,m;
      double d,dm;

      for(n=0;n<N1;n++)
      {
            for(dm=10,i=0;i<N2;i++)
            {
                  d=v[n]-cv[i];
                  if(d<0) d*=-1;
                  if(d<dm)
                  {
                        dm=d;
                        m=i;
                  }
            }
            v[n]=cv[m];
      }
      return 0;
}
/*=========================================================================*/
/* int QSV(int Mv,double * v,const double * cv)                            */
/*           Quantize v using code vector cv                               */
/*           Mv = cv length                                                */
/*=========================================================================*/
```

```
int QSV(int Mv,double * v,const double * cv)
{
      int i,m;
      double d,dm;

      for(dm=1e+10,i=0;i<Mv;i++)
      {
            d=*v-cv[i];
            /*if(d<0) d*=-1;*/
            d*=d;
            if(d<dm)
            {
                  dm=d;
                  m=i;
            }
      }
      *v=cv[m];
      return m;
}
/*=============================================================================*/
/*                                                                             */
/* BOOL Qlsp()                                                                 */
/*           Quantize & code LSPs                                              */
/*=============================================================================*/
BOOL Qlsp()
{
      double f1,f2;

      f1=q[1];
      q[1]=q[0]-f1;
      f2=q[2];
      q[2]=f1-f2;
      f1=q[3];
      q[3]=f2-f1;
      f2=q[4];
      q[4]=f1-f2;
      q[5]=f2-q[5];

      code.cw1.lsp1 = QSV(8 ,q+0,lsp1_codes);
      code.cw1.lsp2 = QSV(8 ,q+1,lsp2_codes);
      code.cw1.lsp3 = QSV(8 ,q+2,lsp3_codes);
      code.cw1.lsp4 = QSV(8 ,q+3,lsp4_codes);
      code.cw1.lsp5 = QSV(16,q+4,lsp5_codes);
      code.cw2.lsp6 = QSV(16,q+5,lsp6_codes);
      code.cw2.lsp7 = QSV(8 ,q+6,lsp7_codes);
      code.cw2.lsp8 = QSV(8 ,q+7,lsp8_codes);
      code.cw2.lsp9 = QSV(8 ,q+8,lsp9_codes);
      code.cw2.lsp10= QSV(8,q+9,lsp10_codes);

      return 0;
}
/*=============================================================================*/
/*                                                                             */
/* BOOL Qpulses(int n)                                                         */
/*           Quantize & code Pulses                                            */
/*           n is the sub frame index                                          */
/*=============================================================================*/
BOOL Qpulses(int n)
{
      int m;


      if(mp[0]>31) mp[0]=31;

      mp[1]-=mp[0];
      if(mp[1]>31) mp[1]=31;
```

```
        m=mp[1]+mp[0]; /* m= new mp[1]*/
        mp[2]-=m;
        if(mp[2]>31) mp[2]=31;

        m+=mp[2]; /* m= new mp[2]*/
        mp[3]-=m;
        if(mp[3]>31) mp[3]=31;


        code.cw3[n].P12.m1 = mp[0];
        code.cw3[n].P12.m2 = mp[1];
        code.cw3[n].P34.m3 = mp[2];
        code.cw3[n].P34.m4 = mp[3];

        for(m=0;m<M;m++) /* normalisation of the amplitudes before coding */
        {
              b[m]*=norm;
        }

        code.cw3[n].P12.b1 = QSV(CBS ,b+0,b_codes);
        code.cw3[n].P12.b2 = QSV(CBS ,b+1,b_codes);
        code.cw3[n].P34.b3 = QSV(CBS ,b+2,b_codes);
        code.cw3[n].P34.b4 = QSV(CBS ,b+3,b_codes);

        return 0;
}
/*=============================================================================*/
/* BOOL DECODE_lsp()                                                           */
/*            Decode LSPs                                                      */
/*=============================================================================*/
BOOL DECODE_lsp()
{
        double d;


        /* decode LSPS */

        q[0]   =lsp1_codes[code.cw1.lsp1];
        q[1]   =lsp2_codes[code.cw1.lsp2];
        q[2]   =lsp3_codes[code.cw1.lsp3];
        q[3]   =lsp4_codes[code.cw1.lsp4];
        q[4]   =lsp5_codes[code.cw1.lsp5];
        q[5]   =lsp6_codes[code.cw2.lsp6];
        q[6]   =lsp7_codes[code.cw2.lsp7];
        q[7]   =lsp8_codes[code.cw2.lsp8];
        q[8]   =lsp9_codes[code.cw2.lsp9];
        q[9] =lsp10_codes[code.cw2.lsp10];

        /* check for stability */

        if(q[0]>qmax) q[0]=qmax;

        q[1]=q[0]-q[1];
        d=q[0]-dq;
        if(q[1]>d) q[1]=d;

        q[2]=q[1]-q[2];
        d=q[1]-dq;
        if(q[2]>d) q[2]=d;

        q[3]=q[2]-q[3];
        d=q[2]-dq;
        if(q[3]>d) q[3]=d;
```

```
      q[4]=q[3]-q[4];
      d=q[3]-dq;
      if(q[4]>d) q[4]=d;

      q[5]=q[4]-q[5];
      d=q[4]-dq;
      if(q[5]>d) q[5]=d;

      d=q[5]-dq;
      if(q[6]>d) q[6]=d;

      d=q[6]-dq;
      if(q[7]>d) q[7]=d;

      d=q[7]-dq;
      if(q[8]>d) q[8]=d;

      d=q[8]-dq;
      if(q[9]>d) q[9]=d;

      if(q[9]<qmin) q[9]=qmin;


      return 0;
}
/*=============================================================================*/
/*                                                                             */
/* BOOL DECODE_pulses(int n)                                                   */
/*           Decode pulses amplitudes & positions                              */
/*=============================================================================*/
BOOL DECODE_pulses(int n)
{
      int i;


      mp[0] = code.cw3[n].P12.m1;
      mp[1] = code.cw3[n].P12.m2;
      mp[2] = code.cw3[n].P34.m3;
      mp[3] = code.cw3[n].P34.m4;


      mp[1]+=mp[0];
      mp[2]+=mp[1];
      mp[3]+=mp[2];



      b[0]=b_codes[code.cw3[n].P12.b1];
      b[1]=b_codes[code.cw3[n].P12.b2];
      b[2]=b_codes[code.cw3[n].P34.b3];
      b[3]=b_codes[code.cw3[n].P34.b4];

      /* restoration of the scale of the amplitudes */
      for(i=0;i<M;i++)
      {
            b[i]*=scale;
      }

      return 0;
}
```

```
;****************************************************************
;
;                   vecs.asm
;
;                   staff
;
;                   07-30-90
;
;            (C) Texas Instruments Inc., 1992
;
;            Refer to the file 'license.txt' included with this
;            this package for usage and license information.
;
;****************************************************************
*************************************************************
*   VECS.H                                                  *
*                                                           *
*   LINEAR PREDICTIVE CODING: LPC-10 ON THE TMS320C30       *
*                                                           *
*   (C) 1990 TEXAS INSTRUMENTS, HOUSTON                     *

*************************************************************
*   INTERRUPT AND RESET VECTORS                             *
*************************************************************
        .sect ".vecs"     ; interrupt and reset vectors

        .ref _c_int00     ; compiler defined C initialization reset
        .ref _c_int05     ; serial port transmit interrupt routine
        .ref _c_int11     ; DMA counter interrupt
        .ref _c_int99     ; unexpected interrupt handler

reset:  .word   _c_int00
int0:   .word   _c_int99
int1:   .word   _c_int99
int2:   .word   _c_int99
int3:   .word   _c_int99
xint0:  .word   _c_int05
rint0:  .word   _c_int99
xint1:  .word   _c_int99
rint1:  .word   _c_int99
tint0:  .word   _c_int99
tint1:  .word   _c_int99
dint:   .word   _c_int11
```

```
/*************************************************************************/
/*                                                                       */
/*   mpe.CMD                                                             */
/*                                                                       */
/*************************************************************************/
/*   LINK COMMAND FILE FOR MPE PROGRAM                                   */
/*************************************************************************/
-c
vecs.obj
c30.obj
mpe_c30.obj
-o MPE.OUT
-l rts30.lib
-m mike.map
-stack 0x200
-heap 0x1000
-x
-w


MEMORY
{
   VECS:   org = 0         len = 0x40       /* RESERVED VECTOR LOCATIONS
*/
   SRAM:   org = 0x40      len = 0x3FC0     /* PRIMARY BUS SRAM (16K)
*/
   RAM :   org = 0x809800 len = 0x800       /* INTERNAL RAM (2K)
*/
}

SECTIONS
{
   .vecs:     > VECS                        /* RESET/INTERRUPT VECTORS   */
   .text:     > SRAM                        /* CODE                      */
   .cinit:    > SRAM                        /* C INITIALIZATION TABLES   */
   .data:     > SRAM                        /* ASSEMBLY CODE CONSTANTS   */
   .const:    > SRAM
   .sysmem:   > SRAM
   .stack:    >  RAM                        /* STACK */
   .bss:      > SRAM                        /* C VARIABLES */
}
```