« ALGORITHMES D'ACQUISITION, COMPRESSION ET RESTITUTION DE LA PAROLE À VITESSE VARIABLE. ÉTUDE ET MISE EN PLACE »

Projet de fin d'études de :

José HERNÁNDEZ

Avril 1995

ENSEA
École Nationale Supérieure de l'Électronique et de ses Applications
Cergy-Pontoise (Paris)

« ALGORITHMES D'ACQUISITION, COMPRESSION ET RESTITUTION DE LA PAROLE À VITESSE VARIABLE. ÉTUDE ET MISE EN PLACE »

José HERNÁNDEZ
Avril 1995

ECS : Équipe Commande des Systèmes.
Dirigée par : M. Pascal GOUREAU.

Stage effectué au laboratoire ECS-ENSEA comme projet de fin d'études de la Licence d'Informatique avec le programme ERASMUS d'échanges universitaires réalisé entre l'ENSEA et l'UPV (Universidad Politécnica de Valencia, Espagne).

A mi hermano,
porque, después de todo,
le debo mucho de lo que soy.

AVANT-PROPOS

Le traitement de la parole s'est développé pendant les dernières décennies dans le domaine des télécommunications. Des avancées importantes ont été réussies sur le plan du traitement analogique et, fondamentalement, sur le traitement numérique grâce à des techniques comme le filtrage, les transformées, les analyses spectrales, les paramétrisations et les algorithmes de compression.

Aujourd'hui, cependant, les applications du traitement de la parole vont changer de destination au profit de l'utilisateur quotidien habitué aux appareils à son numérique, aux nouvelles autoroutes de l'information et au monde du « multimédia ». Tout cela s'intensifiera par l'augmentation de la puissance des processeurs ou des ordinateurs actuels. L'existence des DSP (Digital Signal Processors), très rapides et très efficaces, permet de réaliser des appareils concrets pour des applications les plus diverses avec des algorithmes et des facilités imaginées juste quelques années plus tôt. De plus, les ordinateurs multimédias fournis avec des cartes de son chaque jour plus élaborées permettent un traitement et un stockage de la parole avec tous les outils disponibles sur un ordinateur personnel. Dans cette nouvelle étape, la qualité du son doit être identique à celle déjà installée dans les appareils audio numériques que chacun possède (comme les disques compacts, les cassettes numériques, etc.).

Ce travail présente un résumé des techniques et des algorithmes qui existent et qui vont être les plus utilisés à l'avenir. On commencera par un bref exposé des procédures de numérisation, de filtrage, des fenêtres, des transformées et de l'analyse spectrale en général. Après, on introduira des compressions de bonne qualité déjà évaluées et on discutera de leur mise en place.

Quant à la partie réalisation des algorithmes, j'ai développé des modifications des versions traditionnelles et j'ai effectué des nouveaux algorithmes de compression, des algorithmes innovateurs de vitesse variable et de comparaison de sons. La complexité de tout ceci a été évaluée avant une possible mise en œuvre.

Tout cela se concrétise par deux réalisations :

- Primo, un outil multimédia de simulation pour P.C. qui sert à montrer les possibilités de traitement de la parole (avec des connaissances informatiques et des besoins matériels minimaux pour un utilisateur habituel).

- Secundo, une mise en place sur DSP (ou ASIC) pour montrer la facilité de réalisation d'un appareil spécifique de coût réduit

pour une application concrète (dans notre cas, l'apprentissage des langues).

PLAN GÉNÉRAL

Remerciements

« Les six premiers mois sont les plus difficiles... »

Ce travail aurait été impossible sans l'inestimable collaboration de Boris Siepert, qui a eu la patience de m'aider avec mon français les premières semaines, de m'introduire dans le monde du traitement de la parole, d'ajouter des nombreux commentaires et idées à mon projet et, fondamentalement, de rendre mes heures de travail dans le laboratoire un peu plus agréables.

Je veux également remercier toutes les personnes qui, comme Erwan Humbert, ont lu les premiers brouillons de ce rapport pour corriger les nombreuses fautes d'orthographe ; ou tous les autres membres de l'ECS comme Aziz Djermoune parce que nous partageons le même concept de profiter du temps.

Je doit donner une mention spéciale à mon directeur de projet, Pascal Goureau, qui m'a donné un peu de son précieux temps pour m'expliquer (avec ma connaissance réduite de la langue) les objectifs de ce travail et qui progressivement l'a augmenté jusqu'à son heureux achèvement. De plus, l'encouragement mis par le P.D.G. de BARTHE, M. Troïanowski, pour continuer ce projet est un orgueil pour moi, car c'est implicitement une reconnaissance de la valeur de mon travail.

En rapport avec mon séjour en France, je dois reconnaître l'appui moral de mes amis Fédérico García-Lorca et Ricardo Sánchez qui souvent venaient m'enlever de l'écran de l'ordinateur pour prendre l'air à Paris ou simplement converser un peu en espagnol.

Finalement, à tous ceux que j'ai laissés à Valence pendant six mois : Des amis et collègues comme Sergio Talens qui me tenaient à jour de toutes les nouvelles via l'Internet, la famille et particulièrement à Nieves, lesquels m'ont encouragé, directement ou indirectement, à bien travailler pour finir le plus tôt possible.

Cergy, 10 avril 1995

Chapitre 1

INTRODUCTION

On va présenter dans ce chapitre, les conditions et le but de ce travail, avant d'entrer dans le sujet lui-même.

1.1. Le programme ERASMUS :

Ce projet est inscrit dans le projet ERASMUS de l'Union Européenne d'échanges d'étudiants, de professeurs et de chercheurs avec des établissements d'enseignement supérieur en Europe. Les deux institutions sont concrètement l'ENSEA (Ecole Nationale Supérieure de l'Electronique et de ses Applications) à Cergy-Pontoise (France) et l'UPV (Universidad Politécnica de Valencia) à Valence (Espagne).

Il s'agit d'un séjour de six mois à l'ENSEA pour la réalisation du projet de fin d'études afin d'obtenir la maîtrise d'informatique dont j'ai suivi les cours pendant cinq années dans mon Université d'origine.

1.2. L'ENSEA :

C'est l'un des seuls établissements publics en Ile-de-France formant des ingénieurs électroniciens de haut niveau, sous la tutelle du Ministère de l'Éducation Nationale et de la Recherche. L'ENSEA (École Nationale Supérieure de l'Électronique et de ses Applications) a été l'une des toutes premières Grandes Écoles à s'installer à Cergy-Pontoise. Cergy accueille une université qui, avec une venue à terme de 20.000 étudiants, deviendra un des grands pôles universitaires français.

Avec 150 ingénieurs formés chaque année, l'ENSEA compte parmi ses anciens élèves des dirigeants des plus grandes firmes électroniques internationales.

Implantée depuis 17 ans dans la ville nouvelle, l'ENSEA a été fondée sur les bases de l'ancien Institut d'Electronmécanique et d'Electrométallurgie de Paris créé dès 1941.

Au rythme de l'évolution de la recherche, l'établissement est devenu École Nationale de Radiotechnique et d'Électricité Appliquée (ENREA) avant de se concentrer, quelques années plus tard, sur la « fée électronique » qui n'en finit pas de révolutionner notre quotidien.

Aujourd'hui, l'ENSEA a pour mission la formation initiale d'ingénieurs électroniciens de recherche, d'études de développement, aptes à apporter leur concours dans tous les secteurs de l'électronique et de l'informatique. L'École propose aussi des cycles de formation continue destinés à des ingénieurs et des cadres de l'industrie, soit pour leur permettre de suivre l'évolution de l'électronique, soit pour assurer leur promotion au sein de leur entreprise.

Depuis longtemps l'industrie et le gouvernement reconnaissent l'ENSEA comme l'une des meilleures écoles françaises dans le domaine de l'électronique. Elle fait partie du groupe des 200 institutions d'élite, les Grandes Écoles, qui constituent le point culminant de l'enseignement supérieur en France.

Le fait qu'une majorité de gérants et de cadres des plus importantes entreprises françaises soit diplômée d'une Grand École, que 23.500 diplômes d'ingénieur soient délivrés chaque année, donne une idée de l'importance de ces écoles pour l'industrie française.

L'ENSEA, comme toutes les Grandes Écoles, se distingue par :

- Une admission hautement sélective par le biais d'un concours national après 2 ou 3 ans de préparation ;

- Une faible taille et une totale autonomie ;

- Des liens étroits avec l'industrie grâce à la coopération, par des stages et par la recherche ;

- Un grand nombre de matières enseignées et un rapport professeur/élève élevé.

L'ENSEA dispose d'un centre de recherche de très haut niveau dont les travaux font l'objet de publications régulières. Les Laboratoires de Micro-ondes, Traitement des Images et du Signal et Commande des Systèmes poursuivent des

recherches théoriques et appliquées permettant d'actualiser en permanence les enseignements. Ils participent également à la formation de jeunes chercheurs.

Une cinquantaine d'enseignants-chercheurs travaille en permanence dans trois directions :

- Le traitement des images et du signal consacré plus particulièrement à la reconnaissance et l'interprétation des images avec des applications concrètes de portée militaire;

- Les micro-ondes par l'étude sur des alliages rapides pour circuits électroniques;

- La commande des systèmes dont les axes de recherche sont la commande de machines tournantes alternatives par microprocesseur, les algorithmes de régulation et de commande de systèmes « complexes » et la logique floue.

Les départements de spécialité et les laboratoires de recherche travaillent sur des projets en partenariat avec des entreprises, aussi prestigieuses que l'Aérospatiale, Alcatel, EDF, Hewlett-Packard, Thomson, Roussel-Uclaf ou Unisys, développant ainsi l'approche industrielle caractéristique de la formation d'ingénieurs.

Chaque année, 150 jeunes ingénieurs quittent l'école au terme d'une formation solide et se répartissent dans près de 300 grandes entreprises de la filière électronique, sans compter les nombreux débouchés offerts à l'étranger. Beaucoup s'orientent aussi vers la recherche, certains vers l'enseignement.

La formation à l'ENSEA couvre toutes les facettes de l'électronique, depuis l'enseignement de base de tout futur ingénieur (mathématique, physique, mécanique...), en passant par l'électrotechnique et l'électronique. En dernière année, l'étudiant s'oriente, vers un projet de fin d'études axé sur l'automatisme, les mesures et l'instrumentation, l'informatique industrielle, l'analyse du traitement du signal ou les télécommunications.

L'ENSEA poursuit une politique active d'échange d'étudiants, de professeurs et de chercheurs avec des établissements d'enseignement supérieur en Europe. Elle a des relations privilégiées avec l'Université Technique de Berlin, l'École Polytechnique Fédérale de Lausanne, l'Université Polytechnique de Valence, le Polytechnique de Coventry et la Faculté d'Ingénierie de l'Université de Porto.

Commencée voilà un an, son extension dans un bâtiment tout neuf va fortement doper ses activités de recherche et d'enseignement. Ce nouveau bâtiment représente un investissement de 108 millions de francs.

L'ENSEA étouffait un peu jusqu'à présent dans ses 7.000 m² ; grâce au nouveau bâtiment, elle va disposer de 8.000 m² supplémentaires, dont 1.500 m² destinés aux trois équipes de recherche de l'établissement, qui au total devront regrouper en l'an 2.000 une centaine d'enseignant-chercheurs, contre une cinquantaine à l'heure actuelle.

Autre effet positif du déménagement : le « Centre de Documentation Scientifique et Technique » de l'école verra sa surface passer de 240 m² à 600 m², ce qui permettra de l'ouvrir d'avantage aux industriels venant y effectuer des recherches

documentaires. Plus globalement, l'école espère également améliorer sensiblement son image de marque, grâce à ses nouveaux locaux.

En 1994 et 1995, l'ENSEA a acquis pour quelque 3,5 millions de francs d'équipements électroniques destinés à ses nouveaux laboratoires de recherche et d'enseignement ; il s'agit en particulier de stations de CAO (Conception Assistée par Ordinateur) électronique et d'équipements de test de circuits intégrés. En 1995, ainsi qu'au cours du premier semestre de 1996, les anciens bâtiments de l'ENSEA seront entièrement rénovés; pour un coût de 42 millions de francs.

À la rentrée de 1998, ce sont quelques 800 élèves ingénieurs (contres 480 actuellement) que l'école rassemblera. Ayant atteint son plein développement, l'ENSEA sera alors devenue l'une des plus grandes écoles françaises d'électronique.

1.3. L'ECS :

L'Équipe Commande des Systèmes (ECS) de l'ENSEA est une des trois équipes de recherche qu'il y a actuellement dans l'ENSEA. Le responsable de cette équipe est M. Pascal Goureau est elle est composée au printemps de 1995 par :

A. Bigand	Maître de Conférence à l'I.U.P. de Calais
P. Goureau	Professeur Agrégé à l'ENSEA.
S. Lefèvre	Professeur Agrégé à l'ENSEA.
F. Pépin	Professeur Agrégé à l'ENSEA.
P. Toussaint	Professeur Agrégé à l'ENSEA.
A. Djermoune	Thésard, A.T.E.R. à l'ENSEA, direction M. Razek.
A. Hibaliden	Thésard, direction M. Poloujadoff.
A. Messaadi	Thésard, (bourse CIFRE Air Liquide), direction M. Ceschi.
B. Siepert	Thésard, direction M. Demigny.
J. Hernández	Stagiaire ERASMUS en liaison avec la thèse de M. Siepert.

L'Équipe a maintenant trois axes de recherche:

- **Régulation en Logique Floue (R.L.F.)** : (responsable M. Bigand). Il s'agit de profiter de l'expérience antérieure dans ce domaine pour l'appliquer à des régulations de processus. Actuellement, cet axe de recherche abrite les thèses de M. Hibaliden (Implantation d'un régulateur de type « flou » sur des commandes d'onduleurs pilotant des machines alternatives) et de M. Messaadi (Utilisation des réseaux de neurones pour l'amélioration du procédé de soudage M.I.G. pour l'aluminium. Travaux menés en collaboration avec la société Air Liquide (Convention CIFRE)).

- **Commande Numérique de Systèmes (C.N.S.)** : (responsable M. Goureau). Ce domaine sera l'émanation de l'axe de commande numérique des machines mais étendu à la commande de tout système. Cet axe abrite la thèse de M. Djermoune (Étude et mise en place d'algorithmes de régulations pour machines synchrones et asynchrones) et de M. Siepert (réalisation d'un ASIC pour le LLI en collaboration avec la société BARTHE).

- **Les Composants et les Convertisseurs Statiques (C.C.S.)** : (responsable M. Toussaint). Cet axe se situe dans un domaine plus Electrotechnique basé sur l'association de composants récents. Deux travaux sons inclus dans cet axe : M. Toussaint avec des convertisseurs statiques et M. Lefèvre avec des composants de puissance.

Mon projet est inclus dans l'axe de commande numérique des systèmes et en étroite collaboration avec la thèse de M. Siepert et la société BARTHE.

1.4. La société BARTHE et son projet LLI :

La société BARTHE est un fabricant spécialisé en matériel audio professionnel adapté à l'usage de tout public. Dès sa création, la société BARTHE a développé son marché dans le secteur de l'audio et depuis le début des années 80, elle s'est bâtie une réputation dans le domaine de l'enseignement collectif et plus précisément dans celui de l'enseignement des langues en groupe.

En règle générale, l'apprentissage des langues en groupe passe par l'utilisation d'appareils audio plus ou moins sophistiqués. En cours, les professeurs de langues utilisent donc soit des magnétophones classiques à usage collectif de type BARTHE, soit des laboratoires de langue.

Actuellement, les laboratoires de langues sont largement diffusés dans les établissements scolaires et les écoles de langues et sont de plus en plus perfectionnés. Le propre d'un laboratoire de langues est de mettre en relation un poste « maître », celui de l'enseignant, avec plusieurs postes « élèves ». Les fabricants de telles installations développent leurs produits dans le but d'améliorer l'efficacité de l'enseignement des langues en groupe.

Or, il existe également un besoin important d'apprentissage individuel en fonctionnement autonome. L'élève qui apprend par exemple une langue à l'école ou dans des établissements spécialisés et qui a l'habitude de travailler avec un laboratoire de langues a souvent besoin de s'exercer de manière individuelle au moyen de cassettes spécialisées (ou de tout autre support sonore). Il a bien-sûr la possibilité d'utiliser en libre service un laboratoire de langues installé en milieu scolaire ou professionnel. Mais, il n'a pas la possibilité de travailler seul chez lui/elle puisqu'il/elle ne dispose pas d'un matériel individuel adéquat, celui-ci coûtant trop cher.

Cette réflexion a conduit la société BARTHE à conclure qu'il serait intéressant de pouvoir offrir à de telles personnes le moyen de travailler à domicile en utilisant un Laboratoire de Langues Individuel (LLI), c'est-à-dire un appareil pouvant utiliser des cassettes ou des CDs ordinaires en langue étrangère, possédant un double système de répétition de l'information en provenance de l'enregistrement source (poste maître) et de l'information en provenance de celui qui apprend (poste élève) et permettant à l'utilisateur de progresser à son propre rythme.

Pour créer le LLI et le doter de la double fonction de répétition, la société BARTHE a décidé de mettre au point un Répétiteur Numérique permettant le traitement numérique et analogique d'un signal sonore.

Grâce à ce répétiteur numérique, le LLI permettra à son utilisateur de réécouter instantanément, sans recherche et sans action sur l'appareil audio, les dernières secondes de l'enregistrement source (cassette, CD), autant de fois qu'il le désire. Il permettra également à son utilisateur de s'enregistrer, de réécouter son enregistrement autant de fois que nécessaire et de le comparer à l'enregistrement source pour juger de ses progrès (aspect interactif).

Concrètement, ce système doit réaliser les fonctions suivantes :

- Détection du défilement cassette ;
- Détection de la voix élève ;

- Compression / décompression de l'enregistrement cassette ainsi que de la voix élève ;

- Restitution de durée variable des deux.

La puce cible choisie pour les premières versions a été le micro-contrôleur 80C196 de INTEL. Les nouvelles recherches se centrent sur l'optimisation matérielle et l'amélioration fonctionnelle des fonctions antérieures et des autres supplémentaires comme :

- Des algorithmes de compression pour améliorer la qualité et pour réduire la mémoire et la puissance de calcul requises.

- Des nouveaux algorithmes pour la restitution du son enregistré à une vitesse d'élocution variable. Avec la précédente, celle-ci est la tâche fondamentale de ce projet.

- L'intégration de la logique dans un ASIC (Application Specific Integrated Circuit), pour économiser les coûts de l'appareil lors de la production en série. Cet ASIC est le but de la thèse de M. Siepert.

En plus, d'autres projets liés à BARTHE sont suivis par l'ECS : un appareil basé sur un DSP (TEXAS TMS 320C50) pour une gamme de haute qualité ou pour l'enregistrement de quelques minutes pour des applications comme des lettres dictées.

1.5. Objectifs du Projet :

L'objectif de ce travail est de faire diverses études et essais d'algorithmes d'acquisition, de compression et de restitution de signaux de parole. L'évolution des vitesses des processeurs actuels permet essayer quelques nouveaux traitements de complexité supérieure.

Une simulation par ordinateur des tous ces essais est assez facile et économique car il est facile de changer les paramètres et les conditions à examiner. Ceci est possible parce que le traitement est principalement numérique.

Quoique celui-ci soit une étude générale, il se situe en étroite collaboration avec la société BARTHE. Il s'agit concrètement d'un système électronique qui facilite l'apprentissage individuel des langues étrangères, appelé LLI (Laboratoire de Langues Individuelle). Ce système réalise les fonctions suivantes:

- Détection du son et du silence. Ceci permet l'arrêt automatique de l'enregistrement.
- Compression/décompression du son pour l'enregistrement de la plus grande durée avec la plus petite quantité de mémoire possible.
- Restitution de durée et vitesse variables de l'enregistrement.

Les algorithmes de compression à étudier vont de la représentation directe (codeurs de l'onde soit dans le domaine temporel soit dans le domaine de fréquences) jusqu'à la représentation paramétrique (codeurs de source).

Il faut veiller à la qualité de restitution à différentes vitesses parce que quand une personne parle vite ou lentement ce n'est pas fait de façon égale pour tous les phonèmes et formants: il y a quelques phonèmes plus étendus que d'autres. Il semble plus facile d'émuler ce comportement avec des analyses de haut niveau soit de domaine fréquentiel soit par représentation paramétrique.

En revanche, ceci implique une plus grande complexité et difficulté pour leur implantation en temps réel avec les composants aux fréquences disponibles aujourd'hui, à prix raisonnable, raison qui a fait que l'implantation réalisée par l'ECS[1]a dû se conformer à l'algorithme delta adaptatif (ADM).

Trouver de nouveaux algorithmes ou ajuster les paramètres des algorithmes traditionnels avec le compromis (économie - qualité) par une analyse de haut niveau mais qui puisse être implantée en temps réel est l'objectif majeur de ce travail.

[1][MACHA94]

1.6. Planification du Projet :

Le projet a eu une étendue de six mois depuis le 10 octobre 1994 jusqu'au 12 avril 1995 (25 semaines) lesquelles se sont distribuées à peu près de la façon suivante :

- 5 semaines :

  - Recherche bibliographique effectuée au centre de documentation de l'ENSEA, aux différentes bibliothèques de l'Université Paris VI et sur le réseau Internet sur :
    Traitement de la parole.
    Algorithmes de compression.
  - Étude du besoin matériel (carte son, ordinateur) et logiciel (éditeur, compilateur C++, librairies de programmation, etc.).
  - Demande de subvention ANVAR pour financier le besoin défini.
  - Achat.
  - Installation.

- 2 semaines:

  - Réalisation des sous-programmes en langage C pour l'acquisition, la représentation temporelle et spectrale (spectrogramme et sonagramme) et la reconstitution du signal. Pour cette tâche, il a été fait usage de la D.F.T. (Transformée de Fourier Discrète) avec différents algorithmes (F.F.T., D.C.T., K.L.T., W.H.T.).
  - Comparaison et essai des différentes fréquences d'échantillonnage (de 10 khz jusqu'à 50 khz ou la limite permise par la carte de son).

- 4 semaines:

  - Variation de la fréquence de restitution et essai des méthodes pour une restitution à basse ou haute vitesse compréhensible par une personne quelconque.

- 4 semaines:

  - Implantation des algorithmes de compression traditionnels (facteur de compression de 2, 4 voire 8):
  - Recherche de nouveaux algorithmes avec facteurs plus grands (10 ou plus).
  - Détermination expérimentale du meilleur compromis entre fréquence d'échantillonnage et facteur de compression pour arriver à capacités de stockage faibles avec une bonne qualité.

- 6 semaines :

  - Intégration de tous les algorithmes précédents dans un outil de simulation sous WINDOWS.
  - Réalisation des filtres, générateurs de signaux, mesures de complexité et de qualité et des autres nombreuses applications.

- 2 semaines :

- Conception du schéma final d'expérimentation.
- Implantation pratique du schéma précédent sur la puce Texas Instruments TMS-320C50.

- 2 semaines :

  - Rédaction du mémoire du projet.
  - Extraction des conclusions et évaluation des objectifs initiaux.

Cette planification finale a été assez semblable à celle conçue initialement[1] avec une extension plus grande des travaux pour la réalisation des outils de simulation et une réduction considérable pour l'implantation sur le DSP dû à une disponibilité tardive de la mémoire sur la carte.

[1]Dans les annexes.

1.7. Notations :

Dans une étude scientifique de recherche, des acronymes, des conventions et des expressions trop techniques sont couramment utilisées. Avant d'entrer directement dans la partie théorique de ce travail, il peut être important de se familiariser avec des mots qui vont être utilisés.

1.7.1. Mots clefs :

Traitement Numérique du Signal, Traitement Automatique de la Parole, Reconnaissance Automatique de la Parole, Synthèse Automatique de la Parole, Compression, Acquisition, Restitution, Échantillonnage, Langage C/C++, Carte du son, Transformées, Fast Fourier Transform, Vitesse Variable, Apprentissage de langues, Algorithmique, Fréquence du Fondamental (*Pitch*), Expansion, Contraction, Qualité de son, Digital Signal Processor (DSP), TEXAS TMS320C50.

1.7.2. Abréviations et Acronymes :

Les abréviations qui apparaissent dans ce texte sont souvent très cryptiques et confuses, surtout quand on se réfère à la compression de la parole (ADM, DPCM, ADPCM, xCELP, etc.). Pour ces raisons, chaque fois qu'une nouvelle abréviation apparaîtra, on indiquera clairement sa correspondance. On retrouvera ces définitions dans un mémo (glossaire de termes) situé en l'annexe D).

1.7.3. Langues Utilisées :

Bien que la langue utilisée pendant la recherche ait été le français, la diversité de nationalités du laboratoire, et l'intention de généraliser ce travail a fait que quelques parties ont été écrites en anglais. Concrètement :

- Le mémoire, présentation du projet et toute la documentation ont été développés en français.

- Le logiciel de simulation sous WINDOWS a été réalisé en anglais. Les ObjectWindows 2.0 insèrent certaines informations par défaut dans les menus. Pour ne pas mélanger les langues, l'anglais a été choisi volontairement.

- Les commentaires et variables des programmes sont en anglais. Les diverses sources d'où les algorithmes ont été extraits et l'utilisation du C++ comme langage de programmation ont presque obligé à utiliser l'anglais pour rendre uniforme l'apparence des listings.

En relation avec la rédaction de ce rapport, je m'excuse d'avance pour les erreurs syntaxiques et orthographiques qui restent encore. Elles étaient encore plus nombreuses sans le correcteur grammatical de Microsoft WORD 6.0 et dans les premiers brouillons de ce texte.

1.8. Publications et Travaux Futurs :

La plupart du travail a été une évaluation de l'état actuel du traitement de la parole. Les résultats de la simulation seront d'utilités pour la conception de l'ASIC et pour des prochains produits de BARTHE ou des travaux dans le laboratoire.

Mais il y a quelques études qui ont certain degré d'innovation ou de valeur scientifique en matière de traitement de la parole.

En premier lieu, aucune discussion exhaustive n'a été trouvée sur la vitesse variable et les algorithmes présentés dans la bibliographie ne sont pas souvent de qualité suffisante. J'ai l'intention d'écrire un article en anglais et français pour l'envoyer aux revues les plus importantes sur le sujet, encouragé par toutes les applications que le futur peut donner à la vitesse variable. Cet article sera fondamentalement théorique ; d'autres informations confidentielles de valeur industrielle peuvent aussi être publiés avec l'autorisation de BARTHE.

En deuxième lieu, des études de qualité de son ont été développées, mais les résultats n'ont pas été complètement satisfaisants. Cela ne veut pas dire qu'ils ont été inutiles. Un autre article peut être réalisé pour montrer les chemins des mesures de qualité objectives qui n'ont pas donné les résultats espérés.

Enfin, l'outil Accordion et les autres matériels seront laissés dans une adresse Internet pour distribution gratuite. Bien que je prétende le laisser comme directement « ftpable » anonymement, la connexion de l'ENSEA peut conseiller de le distribuer par demande via e-mail. Tous les commentaires sur le logiciel ou sur les algorithmes seront bienvenus.

Chapitre 2

LE TRAITEMENT NUMÉRIQUE DE LA PAROLE

Dans ce chapitre, on présente un résumé des techniques les plus utilisées et confirmées dans le traitement numérique de la parole.

2.1. Le Traitement de la parole :

La parole est la manière naturelle et, en conséquence, la forme la plus commune de communication humaine. À la différence d'autres moyens électroniques de communication, les systèmes utilisant la parole offrent à l'utilisateur non entraîné un accès simple et naturel. Il semble improbable que, dans l'avenir, même l'impact de la technologie des ordinateurs et de réseaux numériques (fax, e-mail et autres) puisse changer la préférence des gens pour la parole comme moyen fondamental de communication entre les humains.

Mais la puissance des ordinateurs et des circuits intégrés, comme les DSPs ou les ASICs font que de nouveaux algorithmes et techniques sont réalisables en temps réel pour augmenter la qualité du signal, et, de cette façon, augmenter l'intelligibilité et réduire la fatigue des interlocuteurs.

Les travaux sur le traitement de la parole ont une histoire de plus de cinquante ans : c'est en 1939 que, pour la première fois, un chercheur des laboratoires Bell aux États-Unis, M. Dudley, proposa un appareil nommé Vocoder (« Voice Coder » ou « codeur de voix ») visant à coder électriquement le signal vocal selon des paramètres limités, puis à le transmettre avec un débit d'informations réduit et à le reproduire enfin dans un système de synthèse effectuant l'opération réciproque du Vocoder. Plus tard, le même chercheur réalisa un autre système de synthèse électrique de la parole, actionné par un clavier. Enfin apparut, en 1947, le sonagraphe, premier analyseur de la composition du signe vocal en fréquence et en amplitude.

Depuis lors, un dénombrement de tous les événements et développements de produits serait fastidieux puisque des avances ont été très importantes depuis cette date, fondamentalement, dans le domaine des télécommunications. Voyons donc, sans plus de préambule, les pas nécessaires et les techniques modernes pour le traitement du signal :

2.1.1. La conversion analogique / numérique (A/N) :

Le traitement de la parole suppose toujours en premier lieu une analyse du signal vocal converti au préalable en signal électrique par un microphone ; Puisque les ordinateurs ne peuvent pas manipuler des sources analogiques, on doit convertir les signaux au format numérique avec un convertisseur A/N . Le processus inverse sera fait par le convertisseur N/A. De la sorte, on va pouvoir travailler sur une représentation spectrale du signal, décomposant ses différentes fréquences avec leurs amplitudes et leurs harmoniques, aboutissant à des « traits » qu'on appelle *formants* du signal.

La raison d'une analyse numérique est qu'elle est plus aisée pour un traitement sophistiqué et qu'elle est beaucoup plus fiable. Le développement rapide des ordinateurs et des circuits intégrés en conjonction avec la croissance des communications numériques a encouragé l'application des techniques numériques au traitement du signal.

La conversion analogique/numérique consiste en l'échantillonnage, la quantification et le codage. L'**échantillonnage** est le processus de représentation d'un signal continûment variable comme une séquence de valeurs. La **quantification** conduit à représenter approximativement chaque échantillon dans un ensemble finit de valeurs. Le **codage** consiste à assigner un numéro réel à chaque valeur.

Avant l'échantillonnage, un filtre passe-bas de fréquence de coupure égale à la moitié de la fréquence d'échantillonnage est inséré pour éviter l'effet dénommé « repliement » ou « aliasing » postulé par le théorème de Nyquist-Shannon ; Ce filtre est donc appelé filtre « anti-repliement » ou « anti-aliasing ». Quelques fois, le filtre n'est pas choisi complètement plat ; On peut le faire d'une autre manière avec un filtre numérique du premier ordre après l'échantillonnage de la façon suivante :

$$H(z)= 1 - \alpha z^{-1}$$

Avec α proche de 1.

Il y a deux paramètres qui affectent la qualité du son. Le premier est la fréquence d'échantillonnage (sampling rate) : On la mesure en Hertz (Hz) et des valeurs typiques pour le son sont 4 khz, 8 khz, 11.025 khz, 22.05 khz, 44.1 khz et 48 khz. Cependant d'après le théorème de Shannon, il faut choisir cette fréquence un peu plus grande que la moitié de la bande intéressante parce que les composants électroniques ne sont pas idéaux et qu'il est donc impossible de réaliser un filtre parfait.

Le deuxième paramètre qu'affecte la qualité est la quantification en un nombre de bits fixé. Typiquement, ce nombre varie entre 8, 12, 14 ou 16 bits et détermine la dynamique et le rapport signal à bruit. Généralement, il s'agit d'une représentation uniforme mais une amélioration peut être obtenu avec des quantifications non linéaires (on verra cela dans le chapitre de la compression).

Pour la parole, une fréquence de 8 khz avec 8 bits produit une qualité assez mauvaise, mais le signal peut être compris. Quoique la puissance des ordinateurs actuels permette des valeurs de 44.1 khz sur 16 bits (qualité CD), la plage dynamique

de la parole est à peu près de 50 dB, ce qui donne 10 bits on plus pour obtenir un bon rapport signal/brut (SNR).

Quelques fréquences sont plus populaires que d'autres, pour diverses raisons. Quelques matériels sont restreints à certaines valeurs ou diviseurs par la simplicité des circuits diviseurs entiers de fréquence. Le tableau suivant montre les plus connues:

échantillons / sec.	description
8000	Utilisé par les μ-Law et A-Law et beaucoup d'autres applications de télephonie.
11025	Le quart de la taux des disques compactes. Utilisée comumment aux ordinateurs Mac.
16000	Utilisé par le standard G.722
18900	Le standard CD-ROM/XA.
22050	La moitié de la taux des disques compactes. Utilisée aussi aux Macs.
32000	Utilisée en radio numérique, NICAM, TV, DAT de longue durée et HDTV japanoise.
37800	CD-ROM/XA de haute qualité.
44056	Audio professionnel.
44100	La taux des disques compactes.
48000	La taux des DAT (Digital Audio Tape).

Dans les ordinateurs, comme on le verra, les fréquences 11025, 22050 et 44100 sont les plus couramment utilisées.

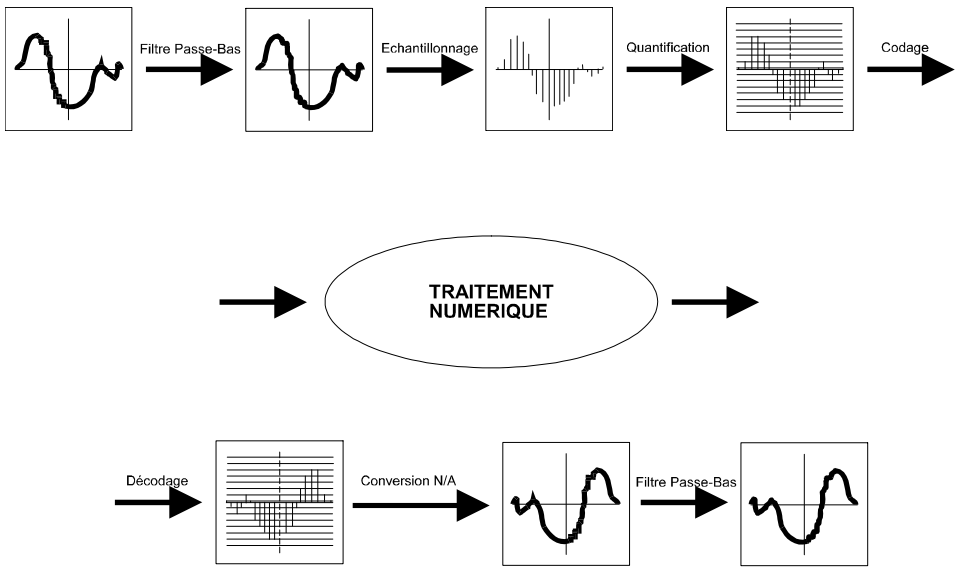
Quoique beaucoup de musiciens soient en désaccord, la plupart des gens n'ont pas de problèmes si le son enregistré est joué à un taux différent d'environ 1 à 2%. Néanmoins, si les données sont acquises par un appareil en temps réel (par exemple un réseau), même la plus petite différence de taux peu frustrer le schèma utilisé.

Puisque nous traitons uniquement de la parole, il n'y a pas de problème avec ces petites différences.

2.1.2. La conversion numérique/analogique (N/A) :

Le processus inverse de la conversion A/N doit être réalisé dans la conversion N/A. Un filtre pas-bas suivra le convertisseur. Les mêmes relations doivent être vérifiées (fréquence de coupure plus petite que la moitié de la fréquence d'échantillonnage).

Le résultat de tout le processus est résumé sur l'image suivante:



Les conversions A/N et N/A concernent des procédures bien étudiées et populaires. Aucune avance significative ne s'est produite à part l'amélioration et l'économie des composants électroniques et l'augmentation de la fréquence. À cet égard, on peut noter de nouvelles techniques qui sont apparues pour la reconstruction de signaux de bande limitée. Par exemple, l'article de Pillai et Elliot[1] parle d'une reconstruction qui utilise les dernières valeurs avec des fréquences proches (et même inférieures !) à celles postulées par le théorème de Nyquist-Shannon.

2.1.3. Caractéristiques de la parole :

Pendant les premières années de traitement du signal, on a mélangé les caractéristiques de la musique et de la parole. Bien qu'on puisse profiter beaucoup des caractéristiques de la première en relation avec la compression[2], les spécificités de la parole sont encore plus fortes et profitables.

Deux limitations fondamentales méritent d'être prises en compte : les limitations du système auditif et celles du système vocal chez l'être humain.

Le système auditif humain est surtout sensible dans une gamme de fréquence située entre 800 Hz à 8.000 Hz; les limites extrêmes sont respectivement 20 et 20.000 Hz.

Par contre, le système vocal est encore plus limité, ce qui ne peut faire respecter les restrictions précédentes que pour la musique de qualité hi-fi. En résumé, pour des sons vocaliques à des fréquences au-dessus de 4 khz, les hautes fréquences sont plus de 40 dB en dessous du sommet du spectre. Par ailleurs, en ce qui concerne des sons du type fricatif, le spectre ne chute pas nettement avant 8 khz. C'est pour cela que pour représenter correctement des sons de ce type, il serait nécessaire d'utiliser une fréquence d'échantillonnage égale ou supérieure à 20 khz.

Cependant, pour beaucoup d'applications, il n'est pas nécessaire de recourir à une fréquence d'échantillonnage si élevée, tout simplement parce qu'une qualité

[1] [PILLA93]
[2] [SMYTH90]

parfaite de reproduction n'est pas exigée. C'est pourquoi, si on applique un filtre passe-bas d'ordre élevé à 4 khz avant l'échantillonnage, il est possible d'effectuer un échantillonnage à 8 khz avec une bonne qualité. Mais il y a quelques consonantes de langues étrangères (à 8 ou 10 khz) qui ne seront pas reproduites très fidèlement. C'est pour cela que, si l'on peut, un échantillonnage à la fréquence de 20 khz est à conseiller, toujours précédé par un filtre analogique à 10 khz.

Une fois l'échantillonnage fait, le signal de parole résultant est fondamentalement variable. Ces variations proviennent de diverses sources :

I. ***Caractéristiques de l'environnement*** : La salle ou l'entourage et les moyens utilisés dans l'enregistrement conditionnent fortement les composantes du signal.

II. ***Caractéristiques de la source*** : Le trait vocal du locuteur varie d'une façon importante d'une personne à l'autre, en fonction de l'âge et même de l'état physique ou psychique de la même personne, entre les différentes langues et selon l'emphase du discours.

III. ***Articulation*** : La relation entre des phonèmes consécutifs est extrêmement complexe et importante.

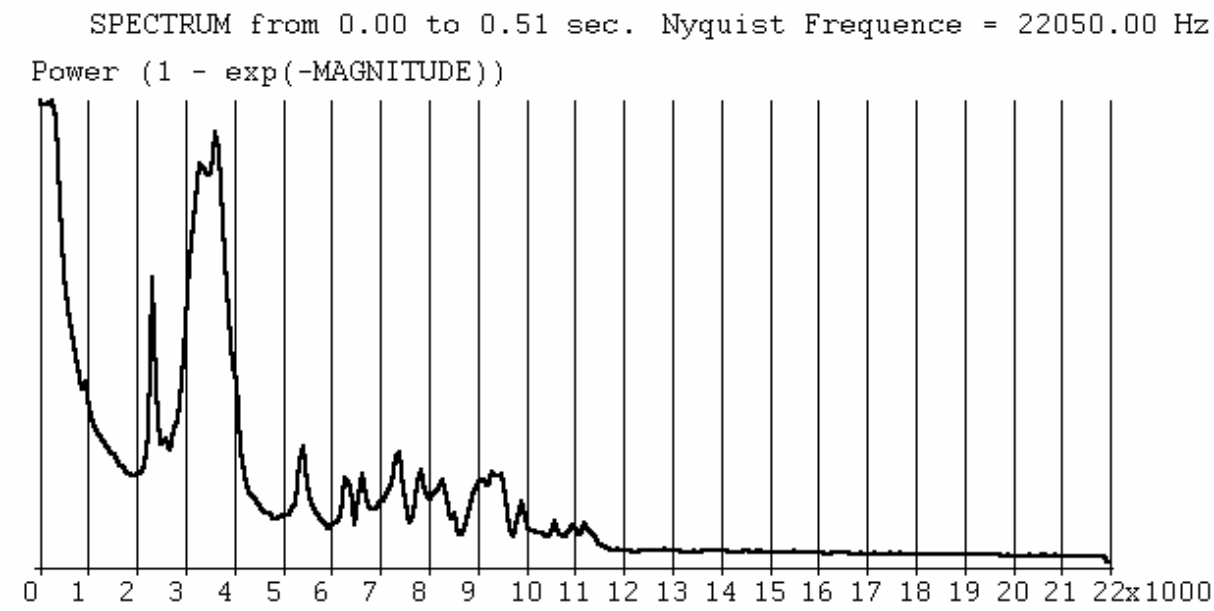
IV. ***Contraste*** : Quand un mot ou une phrase n'a aucune possibilité d'équivoques, il peut être prononcé vaguement, mais s'il peut produire des confusions, il est plus clairement prononcé en contraste par rapport à l'autre source de confusion.

V. ***Intonations et variations libres*** : Des effets les plus divers sont effectués par des locuteurs pour exprimer états d'esprit, emphase, questions, imitations, blagues, et un long et cetera.

Le signal de la parole est presque stationnaire quand on le divise en périodes de 10 à 30 ms. Il y a une vaste littérature sur les signaux stationnaires[1]. Un signal stationnaire est à peu près un signal dont la fréquence (et, donc, sa corrélation) ne change pas avec le temps.

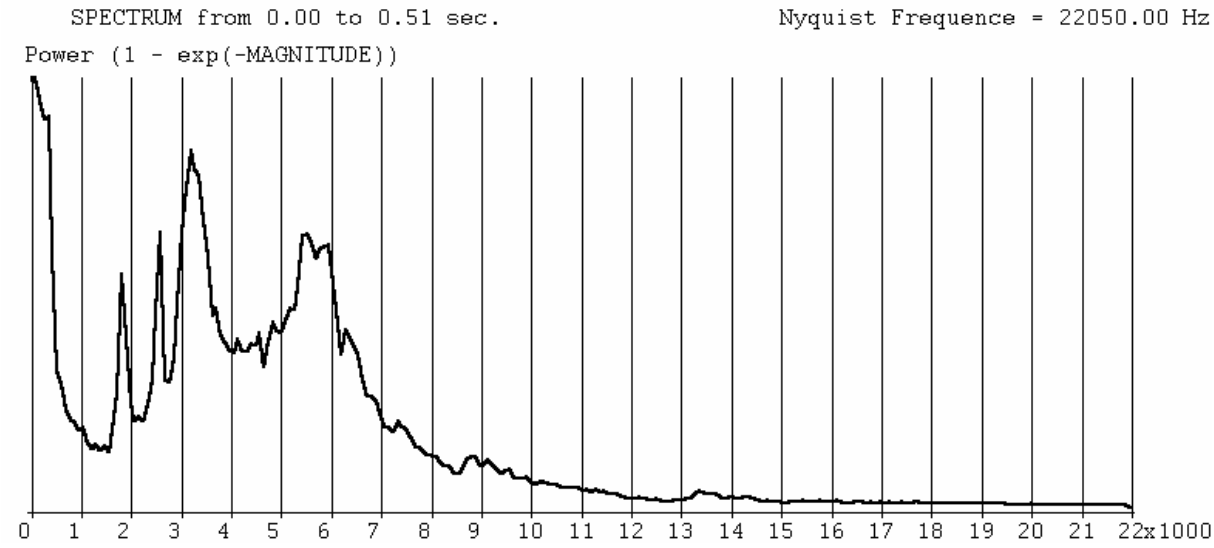
Avec cette analyse stationnaire, on regarde s'il y a une grande différence entre les sons voisés et les non voisés. En suite, on montre le spectre du signal du son /i/ pendant une demi seconde :

[1] [RABIN78] et [FLANA72] sont de bons exemples



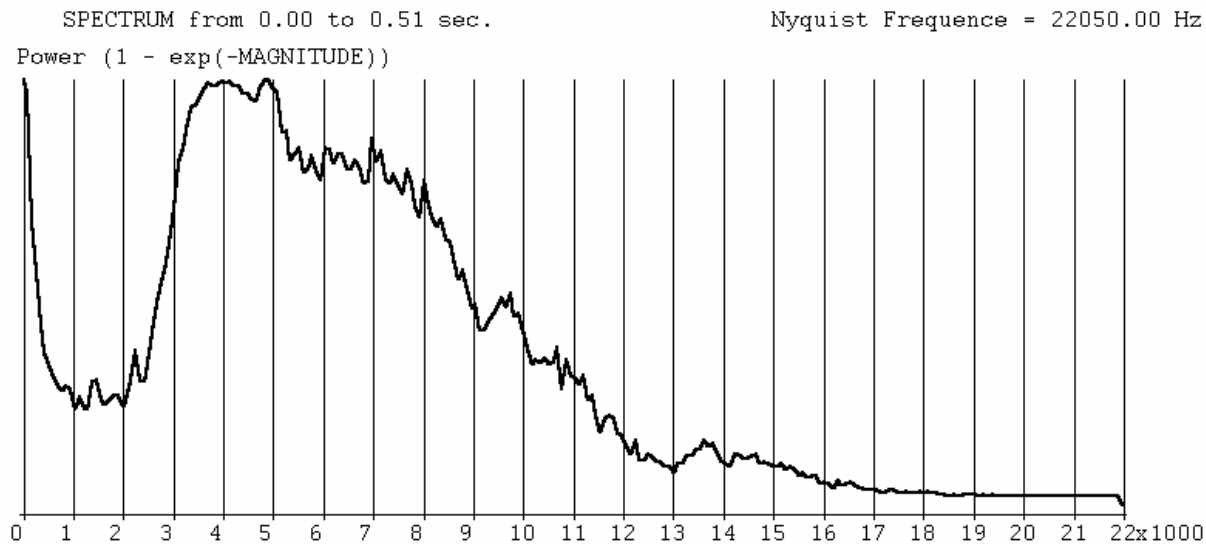
Pour les sons voisés, on peut voir sur un spectrogramme, trois piques fondamentales, qui sont appelés premier, deuxième et troisième formant (F1, F2 et F3) respectivement. Dans l'exemple précédent, on aperçoit les valeurs F1 = 200 hz, F2 = 2300 Hz et F3 = 3600 Hz. Mais la difficulté apparaît pour des variations obtenues par différents locuteurs prononçant le même phonème.

D'autres classifications peuvent être réalisées pour les consonnes voisées. L'exemple prochain montre le son /ζ/ (du français « ***g****ens* » ou de l'anglais « *plea****s****ure* ») pendant une demie seconde :



La classification est un peu plus compliquée. On a F1= 1800 hz, F2 = 2600 hz et F3 = 3200 hz.

Pour les sons non-voisés ces considérations sont loin d'être simples. L'exemple prochain montre le son /s/ pendant une demie seconde :



Il est remarquable d'obtenir un unique formant à une fréquence très haute (> 4000 hz.). Cela définit le son comme non voisé.

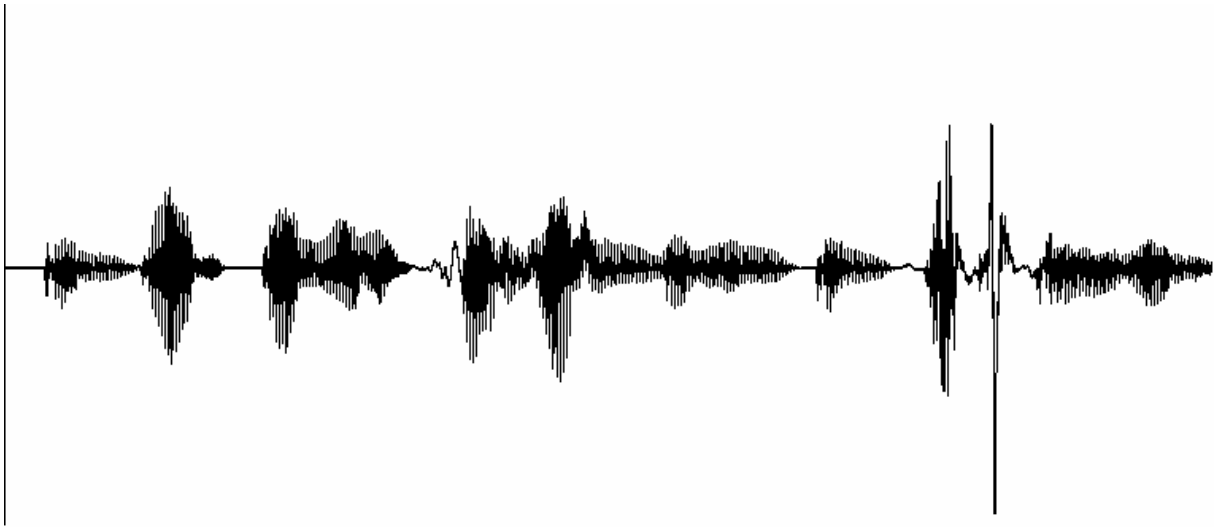
Enfin, il y a beaucoup de méthodes pour l'extraction des formants, des plus simples aux plus complexes, qui utilisent la LPC (Linear Predictive Coding) et le spectre de phase[1]. On verra tout cela plus loin.

La fréquence de vibration des cordes vocales, appelée *fréquence du fondamental* ou taille du *pitch* est une valeur plus intéressante pour les objectifs de ce projet. La fréquence du fondamental et son pitch correspondant selon Boite & Kunt[2] et Furui[3] est :

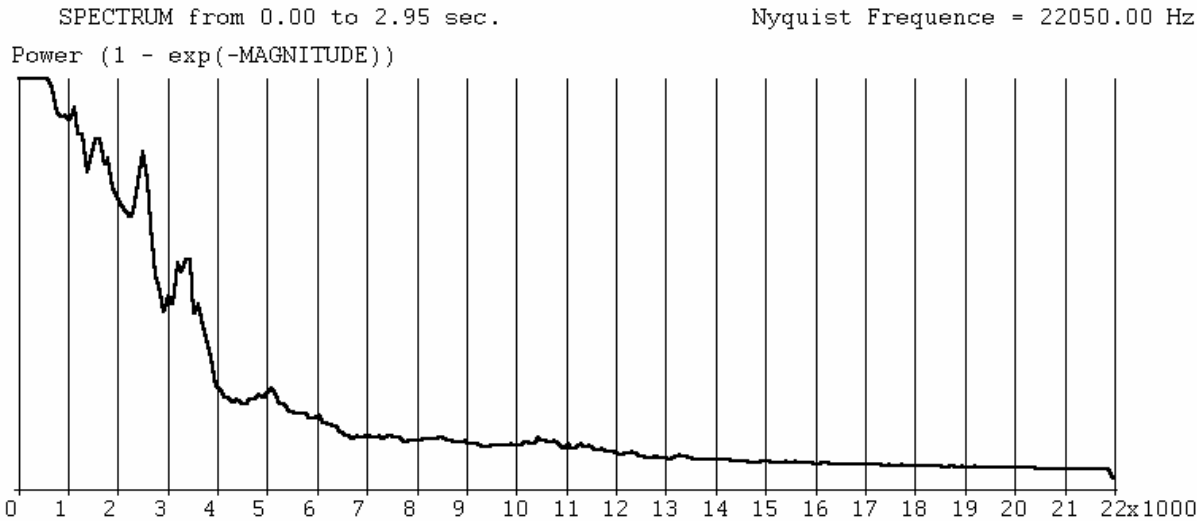
- De 80 à 200 Hz (5 ms. à 12.5 ms.) pour une voix masculine. Distribution normale N(125, 20).
- De 150 à 450 Hz (2.22 ms. à 6.66 ms.) pour une voix féminine. Distribution normale N(250, 20).
- De 200 à 600 Hz (1.66 ms. à 5 ms.) pour une voix d'enfant.

La phrase « Con diez cañones por banda viento en popa a toda vela » pendant 3 secondes de conversation réalisée par moi est montrée sur la figure suivante.

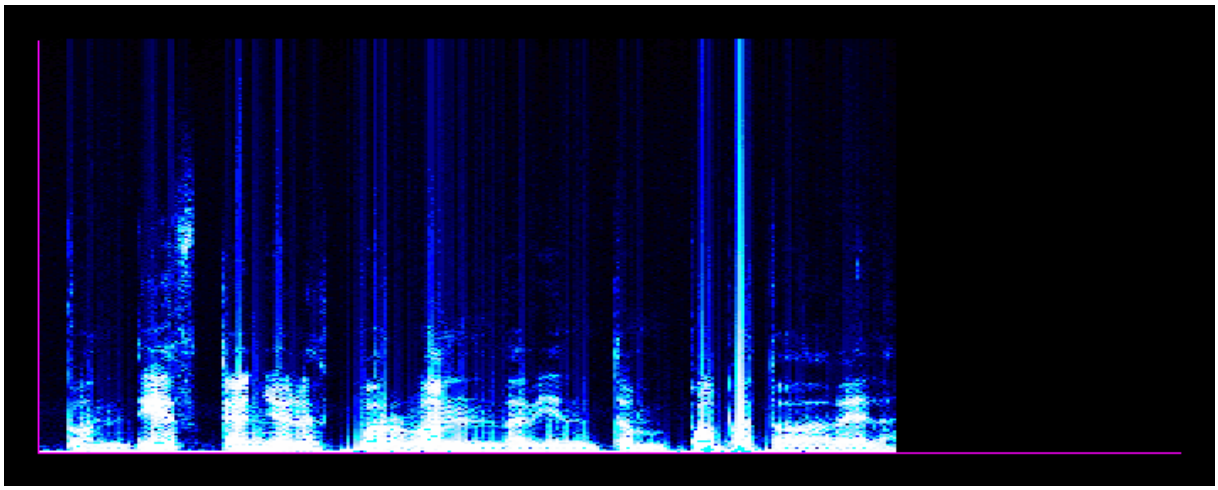
[1] [JINHA93]
[2] [BOITE87]
[3] [FURUI89]



Son spectre est :



Le pitch détecté par corrélations a été de 7.85 ms., correspondant, à une voix masculine (et séductrice comme la mienne). Le sonagramme montre l'évolution temporelle du spectre (on expliquera plus loin avec plus de détail la réalisation d'un sonagramme) :



L'intention de tous ces graphiques est de montrer que l'étude de la parole est l'autant plus simple que l'on fait une abstraction plus importante. Mais après tous les efforts et le coût informatique de ces analyses, les conclusions sont difficiles et complexes. Ce n'est pas étonnant que les avances sur la reconnaissance de la parole progressent si lentement.

2.1.4. Différences entre les langues:

Cette étude a été réalisée pour être valide pour la plupart des langues actuelles, mais elle a été simplifiée de deux façons. Primo, on a omis certains mécanismes qui ne se retrouvent généralement pas dans les langues les plus étendues (indo-européens et orientales) comme les inspirations sonores (de l'extérieur à l'intérieur des poumons) et les clicks utilisés dans quelques langues africaines. Comme on l'a dit avant, quelques consonnes de langues étrangères sont proches de 8 ou 10 khz quand en français, avec 4 khz c'est suffisant pour écouter tous les phonèmes de la langue.

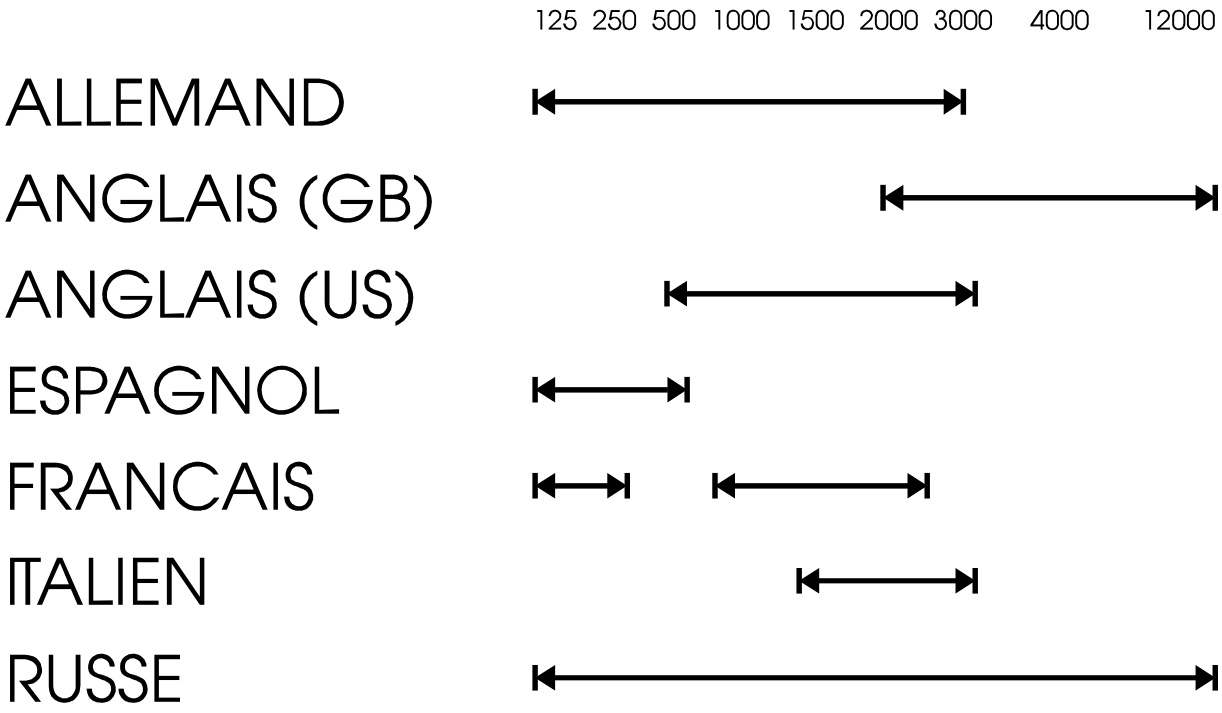
Mais, la qualité espérée pour une application pour l'apprentissage des langues fait que toutes les méthodes essayées se comportent bien (même pour la musique avec une qualité acceptable).

Comme on l'a dit, l'oreille humaine peut théoriquement capter une large gamme de fréquences (16 à 16000 hz) et percevoir une infinité de rhythmes. Mais, au fil des années, notre oreille se contente d'être efficace dans les fréquences et les rythmes où nous utilison surtout notre langue maternelle et nous prenons des habitudes... dont nous ne savons plus nous défaire.

Par ailleurs, le timbre de notre voix peut comprendre, du son fondamental aux harmoniques les plus élevées, des fréquences allant de 100 hz à 12.000 hz. Selon des recherches du professeur Tomatis[1] pour l'apprentissage des langues, les Français utilisent surtout les fréquences de 100 hz à 300 hz et 1.000 hz à 2.000 hz (avec deux points culminants à 250 hz et 1.500 hz) alors que les Anglais utilisent des fréquences incluses dans la zone de 2.000 hz à 12.000 hz. On comprend déjà pourquoi les Français ont tant de mal à apprendre l'anglais et les Anglais le français !

Chaque langue utilise donc de façon préférentielle certaines plages de fréquences sonores, appelées bandes passantes, comme le montre le tableau ci-dessous :

[1] [TOMATa], [TOMATb]



On voit aussi, pourquoi les espagnols ont autant de difficulté pour apprendre quelque autre langue.

Les portions de l'anglais et du russe proches au 10.000 hz n'ont pas trop magnitude ; En résumé, une valeur prise comme presque standard est 7000 hz de bande passante (environ 15 khz de fréquence d'échantillonnage), connue normalement comme bande large.

2.1.5. La reconnaissance de la parole :

Quoique ce ne soit pas non plus l'objectif de ce travail, une petite surveillance sur ce point peut rendre les choses un peu plus claires, et elle peut aider à ne pas confondre certains termes. Pour cela, décrivons un peu l'histoire de la reconnaissance de la parole...[1]

Les vrais travaux de reconnaissance de la parole ne commencèrent que vers les années cinquante, avec les recherches en Suisse de J. Dreyfus-Graff dont le système (le « phonétographe ») était déjà capable de « reconnaître » un discours indépendamment de son locuteur. On crut alors le problème simple à surmonter. On s'aperçut très vite que les connaissances théoriques disponibles étaient loin de le permettre. Il fallut une vingtaine d'années de recherche pour qu'apparaisse une première manifestation publique sur le domaine — pessimiste alors —, celle de M. Peirce, chercheur lui aussi des laboratoires Bell, et déclarant en 1969, dans le *Journal of the Acoustical Society of America*, qu'il était impossible de tenter de reconnaître la parole, et que ceux qui l'espéraient vivaient dans l'illusion ! Cet article fit scandale, mais eut un effet salutaire : les chercheurs décidèrent de limiter leurs efforts à la mise au point de dispositifs de capacités limitées et opérationnels. A partir de là, de nombreux progrès furent réalisés.

[1]Source : [VIGNA91]

Deux types de stratégies de recherche prirent alors naissance. L'une, simplifiée, procède par la reconnaissance globale des mots isolés (séparés par au moins 200 ms.) ou enchaînés, mais sur des vocabulaires réduits et pour une seule personne à la fois. L'autre, en revanche, affronte la question de la reconnaissance de la parole continue et, à l'occasion, pour plusieurs locuteurs ; cela en deux étapes : d'abord la reconnaissance analytique des sons élémentaires — les phonèmes — de la langue, puis l'analyse des unités supérieures — lexique et structures de phrases possibles.

Pour la première stratégie, lorsqu'un mot est prononcé, l'image acoustique de ce mot est ainsi confrontée à toutes celles des mots du lexique, et le mot qui s'en approche le plus est alors affiché. En réalité, la reconnaissance n'est jamais immédiate ni aussi simple : même chez une personne identique, il y aura de sensibles différences, et sur un même mot prononcé plusieurs fois de suite, quant aux intensités, aux timbres et aux rythmes d'élocution. D'où la nécessité de concevoir des algorithmes de calcul en vue d'analyser au mieux, c'est-à-dire de façon optimale, les correspondances entre les mots et leurs images acoustiques.

Ces algorithmes de calcul ont donné naissance à ce qu'on nomme la « programmation dynamique », utilisée dans le domaine vocal, dès les années 1969-1971 par des chercheurs japonais tels Shakoe et Chiba. Elle permet de sélectionner par tri progressif des mots prononcés qui n'appartiendraient pas au lexique, et de contribuer à la mise au point de « modèles de reconnaissance globale » par mots isolés, lesquels présentent différents types d'avantages : une grande capacité de reconnaissance, approchant les 99%, et surtout une indépendance forte vis-à-vis des particularités de la langue à reconnaître. Les inconvénients sont la limitation des vocabulaires disponibles dans la plupart des systèmes existants — encore que de grands progrès aient été accomplis —, et surtout, le fait que la prononciation de mots isolés est fort peu naturelle, et que le système demeure dépendant du locuteur qui a prononcé le lexique de référence lors de la phrase d'apprentissage.

La reconnaissance est différente dans les systèmes de « reconnaissance analytique ». Le décodage du signal vocal y est plus fin et procède selon quatre étapes:

1. D'abord, l'analyse acoustique des phrases prononcées ;

2. puis la segmentation du signal enregistré en sons élémentaires — on observe les « zones de stabilité » en regard des « zones d'instabilité » ;

3. ensuite, étape importante, l'extraction des « indices pertinents », c'est-à-dire les caractéristiques acoustiques qui serviront à « personnaliser » les segments détectés : composition en fréquence du spectre du signal, détection des « informations prosodiques » qui se réfèreront à des spécificités de prononciation ;

4. enfin, l'analyse de ces données, à savoir : l'intervention de *modules analytiques successifs* — phonétique, phonologique, lexical, syntaxique, sémantique, pragmatique —, qui vont avoir pour fonction de « traduire » progressivement les phonèmes enregistrés, les phénomènes d'articulation prosodique, les mots, les agencements syntaxiques, leurs sens et enfin, leurs contextes d'emploi.

La reconnaissance automatique de la parole a donc un bel avenir technologique et social devant elle. Il ne s'agit de rien moins que de faciliter nos

relations avec les machines et, conjointement, de contribuer aux progrès dans la compréhension automatique du langage naturel.

2.2. Filtres :

Maintenant, les filtres sont omniprésents dans la technologie électronique soit de façon analogique soit de façon numérique. Ce n'est pas non plus l'objectif de ce travail de faire une étude des filtres, sinon simplement d'établir les formules utilisées pour l'application qui est fondamentalement numérique. L'avantage des filtres numériques est qu'ils sont plus fidèles aux spécifications et éliminent des problèmes de tension, de température et de bruit dans les autres composants du même circuit.

Dans la gamme des filtres numériques, je vais utiliser dans les simulations et les mises en place les filtres IIR (Infinite Impulse Response) d'ordres réduits, mais on peut en essayer d'autres avec le calcul de la fonction de transfert discrète ou continue et les appliquer directement. La bibliographie sur la matière est immense.[1]

2.2.1. Filtres analogiques normalisés :

J'ai choisi les filtres analogiques normalisés les plus courants pour en faire une numérisation. Des filtres plus exacts (directement numériques) peuvent être essayés mais ne sont pas l'objectif de ce projet.

2.2.1.1. Filtres de Butterworth :

Ils sont connus pour avoir une réponse fréquentielle la plus plate dans la bande passante.

La structure d'un filtre de Butterworth est la suivante :

$$B(s) = \frac{1}{a_0 + a_1 s + a_2 s^2 + \ldots + a_n s^n}$$

Le tableau suivant donne les paramètres déterminés jusqu'au sixième ordre.[2]

n	a0	a1	a2	a3	a4	a5	a6
2	1	√2	1				
3	1	2	2	1			
4	1	2.6131	3.4142	2.6131	1		
5	1	3.2361	5.2361	5.2361	3.2361	1	
6	1	3.8637	7.4641	9.1416	7.4641	3.8637	1

2.2.1.2. Filtres de Tchebychev :

Les polynômes de Tchebychev ont l'avantage d'une réponse uniforme tout au long de la bande passante. La réponse en amplitude des filtres de Tchebychev oscille entre deux valeurs dans la bande passante. Le nombre d'oscillations dans cette bande dépend de l'ordre « n » du filtre. L'amplitude des oscillations (ε) est un paramètre libre.

[1] Pour citer quelques exemples, on peut trouver des références plus détaillées dans [ANTON79], [HARRY79], [TAYLO83], [WILLI88] ou [THOMA92].
[2] La source de ce tableau est [THOMA92].

Un filtre de Tchebychev d'ordre « n » prend la forme suivante :

$$T(s) = \frac{A}{(s - s1)(s - s2)(s - s3)..(s - sn)}$$

Les pôles s_k s'écrivent:

$$s_k = a_k + jb_k$$

Dans laquelle:

$$a_k = -\sinh((1/n)\sinh^{-1}(1/e)) * \sin((2k - 1)p / 2n)$$
$$b_k = -\cosh((1/n)\sinh^{-1}(1/e)) * \cos((2k - 1)p / 2n)$$

La valeur de l'epsilon est calculée selon la valeur de A_{max} par :

$$A_{max} = 1 / (1 + e^2)$$

Par exemple, un filtre du quatrième ordre avec A_{max}= 0,1 dB deviendra:

$$0,1 = -10\log_{10}[1 / (1+e^2)]$$

ce qui donne:

$$e = 0,1526.$$

Les pôles seront alors :

$$s1 = -0,26417 + j1,12263$$
$$s2 = -0,63777 + j0,46501$$
$$s3 = -0,63777 - j0,46501$$
$$s1 = -0,26417 - j1,12263$$

Autre exemple : un filtre d'ordre 3 avec une fréquence de coupure normalisée f = 1/2π et avec une oscillation de 0,5 dB est décrit par :

$$G(s) = 0,716 / (s^3 + 1,253s^2 + 1,535s + 0,716)$$

Ou d'une autre façon :

$$G(s) = 1 / (1 + 2.144s + 1,75s^2 + 1.397s^3)$$

Finalement, la structure d'un filtre de Tchebychev peut toujours être exprimée par :

$$T(s) = \frac{1}{a_0 + a_1 s + a_2 s^2 + ... + a_n s^n}$$

Le tableau suivant donne les paramètres déterminés jusqu'au sixième ordre pour une ondulation de 1dB.[1]

n	a0	a1	a2	a3	a4	a5	a6
2	1	0.9957	0.907				
3	1	2.5206	2.0116	2.0353			
4	1	2.6942	5.2749	3.4568	3.628		
5	1	4.7264	7.933	13.75	7.6271	8.1415	
6	1	4.456	13.632	17.445	28.02	13.47	14.512

2.2.1.3. Filtres de Tchebychev inverses :

Inversement, la réponse fréquentielle est plate dans la bande passante et l'ondulation se trouve dans la bande coupée:

La structure d'un filtre de Tchebychev inverse est:

$$I(s) = \frac{b_0 + b_1 s + b_2 s^2 + \ldots + b_n s^n}{a_0 + a_1 s + a_2 s^2 + \ldots + a_n s^n}$$

Le tableau suivant donne les paramètres déjà déterminés jusqu'au sixième ordre pour une ondulation de 1dB.[2]

n	b0 a0	b1 a1	b2 a2	b3 a3	b4 a4	b5 a5	b6 a6
3	1 1	0 8.4672	0.75 36.597	0 79.05			
4	1 1	0 6.2917	1 20.7925	0 40.540	0.125 39.5285		
5	1 1	0 5.4321	1.25 16.0037	0 29.679	0.3125 34.253	0 19.764	
6	1 1	0 5.0006	1.5 14.0028	0 25.612	0.5625 31.720	0 25.038	0.0312 9.8821

2.2.1.4. Filtres elliptiques (ou filtre de Cauer) :

Les réponses fréquentielles ont des ondulations à la fois dans la bande passante et dans la bande coupée mais le point et la pente de coupe est plus exacte :

La structure d'un filtre de Cauer est :

$$C(s) = \frac{b_0 + b_1 s + b_2 s^2 + \ldots + b_n s^n}{a_0 + a_1 s + a_2 s^2 + \ldots + a_n s^n}$$

Le tableau suivant donne les paramètres déterminés jusqu'au sixième ordre pour une ondulation de 1dB.[3]

n	b0 a0	b1 a1	b2 a2	b3 a3	b4 a4	b5 a5	b6 a6
3	1 1	0 2.0235	0.1038	0 1.3228			

[1] La source de ce tableau est [THOMA92].
[2] La source de ce tableau est [THOMA92].
[3] La source de ce tableau est [THOMA92].

			1.6404				
4	1 1	0 2.2818	0.3993 3.6938	0 2.4801	0.0226 2.1035		
5	1 1	0 3.118	0.8633 4.7536	0 6.6659	0.1642 3.7273	0 3.2301	
6	1 1	0 2.8728	1.403 6.4997	0 7.3627	0.5422 9.4823	0 4.4452	0.0413 3.8954

2.2.2. Concrétisation des filtres :

Une fois que l'on a un filtre pour la fréquence de coupure normalisée 1/2π, s'il faut concrétiser à une fréquence f_c quelconque, on doit faire une contraction de l'axe imaginaire :

$$fn = (1/pT)\tan(pTfc)$$

Avec cette fréquence numérique, on fait la substitution selon le tableau suivant pour les différents types de filtres :

Filtre désiré	Substitution
Passe bas de coupure w_c	$s \rightarrow s / \omega_c$
Passe haut de coupure w_c	$s \rightarrow \omega_c / s$
Passe bande de largeur B centrée sur w_c **B= 1/2(w₂ - w₁)(w₁ + w₂)⁻¹**	$s \rightarrow (1 / B)(s + \omega_c^2/s)$
Coupe bande de largeur B centrée sur w_c	$s \rightarrow 1 / [(1/B)(s + \omega_c^2/s)]$

Par exemple, si l'on a le filtre normalisé:

$$G(s) = 0{,}716 / (s^3 + 1{,}253s^2 + 1{,}535s + 0{,}716)$$

Et que l'on veuille le concrétiser par un filtre passe-bas à fc = 0,1 hz, nous faisons d'abord la contraction qui donne fn= 0,1034251515268 hz.

Puis, nous faisons la substitution de s par s/0,206π ; il en résulte :

$$G'(s) = 0{,}196 / (s^3 + 0{,}814s^2 + 1{,}648s + 0{,}196)$$

2.2.3. Numérisation des filtres :

Une première approximation est basée sur la formule d'Euler qui donne la transformation linéaire suivante :

$$s = \frac{1 - z^{-1}}{T}$$

Avec cette formule, la stabilité est conservée mais il n'y a pas de correspondance bi-univoque entre les 2 domaines de stabilité par rapport à s et z.

Le problème peut être résolu par l'utilisation de la transformation bilinéaire :

$$2(1 - z^{-1})$$

$$s= \quad \frac{\quad}{T\,(1 + z^{-1})}$$

Par exemple, à partir de :

$$G'(s) = 0,196 / (s^3 + 0,814s^2 + 1,648s + 0,196)$$

Et avec T= 1 s. on obtient :

$$G(z)= \frac{0,0154 + 0,0461z^{-1} + 0,0461z^{-2} + 0,0154z^{-3}}{1 - 1,9903z^{-1} + 1,5717z^{-2} - 0,458z^{-3}}$$

Ce qui est directement réalisable sur une équation aux différences ou un algorithme programmé.

Pour finir, voyons tout le suite, un exemple de filtre passe-bande de Butterworth de quatrième ordre avec f = 2000 hz :

$$B(s)= 1 / (1 + 2,6131s + 3,4142s^2 + 203131s^3 + s^4)$$

Avec une fréquence de 44,1 khz on a :

$$T = 1 / 44100 = 0,000022675$$

La fréquence normalisée résultante est 2013,64 hz, ce qui donne un ϖ_c = 12652,1. Le résultat en continu est :

$$B'(s)= 1 / (1 + 0,00020653s + 0,00026985s^2 + 0,00020653^3 + 0,00007903s^4)$$

Si on fait la numérisation comme suit :

$$D(z) = (2 - 2z^{-1}) / (0,00022675 + 0,00022675z^{-1})$$

Le résultat doit être :

$$B'(s) = \frac{(0,0068z^{-4} + 0,0271z^{-3} + 0,0406z^{-2} + 0,0271z^{-1} + 0,0068)}{(11,0099z^{-4} - 52,2247z^{-3} + 93,7925z^{-2} - 75,7211z^{-1} + 23,2518)}$$

Comme avant ceci est directement réalisable à une équation aux différences ou un algorithme programmé.

2.2.4. Application des filtres :

Les applications des filtres sont, comme nous venons de le dire, immenses. Les plus importantes pour la parole sont celles décrites ci-dessous :

2.2.4.1. Anti-repliement (anti-aliasing):

Généralement le filtre analogique qui précède au convertisseur A/N et celui qui suit le convertisseur N/A sont les uniques circuits analogiques qui subsistent encore dans la plupart des dispositifs. On n'a pas l'intention d'exprimer ici la nécessité de ces filtres, ni le théorème de Shannon.

2.2.4.2. Accentuation (Enhancement) :

Il s'agit de faire une pré-accentuation des fréquences les plus hautes (filtrage analogique avant l'échantillonnage, ou filtrage numérique juste après) avant les traitements, et une desaccentuation avant la recomposition. De cette façon on arrive à un SNR plus élevé, avec les mêmes algorithmes de traitement de la parole[1]. La technique de base se retrouve dans une version populaire pour l'audio, que l'on connaît sous le nom de DOLBY NR.

Il y a de nombreuses références dans la littérature, depuis les approches de soustraction de spectre ou filtres de Wiener[2] jusqu'à des méthodes non linéaires plus générales[3].

Dans un article de Clarkson et Bahgat[4], on fait référence à une accentuation non linéaire qui donne de bons résultats sur l'intelligibilité. De plus, l'algorithme a été implanté sur un microprocesseur Texas Instruments TMS-320C25.

2.2.4.3. Réduction de bruit :

Le problème de réduction de bruit a beaucoup de relation avec la méthode précédente mais avec des techniques plus diverses. On peut différencier entre bruit blanc (la densité spectrale est constante), bruit rose (la densité varie inversement avec la fréquence) et bruit marron (aucune régularité n'est trouvée).

Différentes approches ont été étudiées. La plus connue est l'annulation par deux microphones directionnels: un microphone dirigé vers le locuteur et l'autre vers l'ambiance sont utilisées pour séparer la parole des autres sons. Malheureusement, ce traitement ne peut être réalisé qu'au moment de l'enregistrement.

D'autres techniques sont plus pratiques et utilisables comme le Filtrage de Moindres Carrés (Minimum Mean Square Filtering), le carré du spectre (spectre squaring), l'analyse du pitch ou la soustraction de spectres.

Les méthodes précédentes sont décrites par Curtis et NiederJohn[5]. Le problème est que la qualité du son est réduite en même temps que le bruit.

De nouveaux algorithmes ont été présentés par Munday[6] qui décrit une technique de réduction de bruit dans le domaine fréquentiel avec une faible complexité et une seule entrée. Par contre, la qualité est préservée et une réduction significative du bruit est obtenue.

---

[1] [LIM__83].
[2] [BOLL_79] ou [LIM__79].
[3] [EGER_84], [MOORE86] ou [CLARK89].
[4] [CLARK91].
[5] [CURTIS78].
[6] [MUNDA90].

2.2.4.4. Banc de filtres :

Avant le développement de la DFT (Discrete Fourier Transform), le spectre se réalisait avec un banc de filtres analogiques. Aujourd'hui, cette séparation de fréquences est fait généralement par la transformée de Fourier ou d'autres transformées plus sophistiquées. Cependant, des filtres numériques sont utilisés de façon arborescente pour séparer en bandes de fréquence le signal original ; cela est utilisé, par exemple, dans la méthode de compression SBC (SubBand Coding) que l'on verra plus loin.

2.3. Partition et recomposition en tranches :

2.3.1. Partition :

Pour de nombreuses raisons (capacité réduite de la mémoire ou du processeur, application des transformées, vitesse variable), il faut faire une partition du signal en petites tranches ou morceaux avant le traitement.

Cette partition peut être uniforme ou variable. Généralement, la partition est uniforme avec N échantillons pour chaque tranche. Pour la plupart des transformées par exemple, ce nombre doit être choisi comme une puissance de 2, soit 256, 512, etc. Pour les méthodes de compression le nombre est plus flexible ou est calculé en concordance avec la valeur du pitch.

Par contre, dans la vitesse variable et quand le pitch est très changeant, la partition peut être réalisée de manière variable, avec des tranches plus longues que d'autres. La raison fondamentale est d'assurer la liaison continue entre tranches.

Une fois le signal coupé, on peut appliquer le traitement qu'il faut : Filtres, transformées, amplifications et toutes sortes d'effets divers.

2.3.2. Recomposition :

Après le traitement on doit ajouter les tranches pour reconstruire d'une façon continue le signal. Pour que la recomposition soit faite d'une façon correcte, on doit utiliser l'une de ces méthodes :

A. ***Addition superposée*** : Si la fonction (ou le filtre) appliqué à chaque tranche de longueur n est finie de longueur m, on peut l'appliquer en donnant comme résultat un signal de longueur n + m, qui sera additionné, avec le décalage approprié à la prochaine tranche.

B. ***Juxtaposition*** : Contrairement au précédent, si la fonction (ou le filtre) est générale, on peut choisir une valeur P plus grand que n, et faire ensuite tous les calculs avec cette longueur. Au moment de la recomposition on supprimera les parties qui se recouvrent.

C. ***Fenêtre*** : Dans le cas général, si la longueur à considérer est n et le traitement est réalisé avec une tranche résultante de n, le résultat ne sera pas correct ou/et les liaisons ne seront pas douces. Ce problème peut être amoindri si on utilise des fenêtres. On verra avec plus de détail les différentes fenêtres applicables.

En général, on parlera des méthodes de partition et recomposition utilisées dans l'application concrète, soit la compression, les transformées ou la vitesse variable.

2.4. Transformées du signal numériques :

Il s'agit d'une façon d'étudier le signal sur un niveau plus haut que le niveau temporel. Certains paramètres sont ainsi plus mieux analysés. Passons à voir les transformées les plus importantes :

2.4.1. Transformée de Fourier Discrète (DFT: Discrete Fourier Transform). L'algorithme FFT :

La transformation de Fourier des signaux analogiques est appropriée pour un traitement continue (analogique) du signal. Pour un traitement numérique, il est nécessaire de la mettre sous une forme pratique utilisable. Cette forme est appelée transformation de Fourier discrète. La formule suivante exprime la DFT :

$$x[k] = T \sum_{n=0,N-1} x[n]\exp(-j2pkn/N)$$

Et la DFT inverse :

$$x[k] = (1/NT) \sum_{n=0,N-1} x[n]\exp(j2pkn/N)$$

Le coût en programmation de l'algorithme direct est de N^2 additions et $2N^2$ multiplications.

Depuis sa redécouverte en 1965 par Cooley et Tukey, l'algorithme de la transformation de Fourier rapide (TFR ou en anglais FFT pour Fast Fourier Transform) a, grâce à son efficacité, révolutionné le traitement numérique des signaux. La transformation de Fourier discrète peut être considérée comme le produit d'une matrice, appelée la matrice de la transformation, par un vecteur formé par les échantillons d'un signal. L'efficacité de la TFR provient de la redondance des éléments de la matrice de transformation. Cette redondance n'est pas quelconque : elle possède une structure bien déterminée. On peut alors avantageusement tirer parti de cette structure pour synthétiser d'autres matrices de transformation possédant des redondances similaires et ainsi généraliser l'algorithme de calcul rapide de la TFR. Le résultat est un algorithme utilisant $N \cdot \log N$ additions et $(N/2) \cdot \log N$ multiplications. Il est important de souligner qu'il s'agit d'un algorithme efficace, pas d'une approximation à la DFT, par conséquent le résultat est exactement le même qu'avec la DFT.

Dans l'outil Accordion, on a développé diverses versions de la DFT, qu'on commentera plus tard dans ce point. Aussi, la DFT est efficacement réalisable dans un DSP comme la famille TEXAS TMS 320.[1]

2.4.2. Transformée de Cosinus Discrète (DCT : Discrete Cosine Transform) :

Beaucoup de méthodes de compression d'image sont basées sur cette transformée (les standards JPEG, MPEG et H.261)[2]. Dans le logiciel, une implantation a été réalisée dans le module « dct.cpp ». Les commentaires expriment son

[1] [PAPAM89].

[2] Une bonne implantation de cela peut être trouvée dans <ftp.uu.net: /graphics/jpeg/jpegsrc.v5.tar.Z> ou <nic.funet.fi: /pub/graphics/packages/jpeg/jpegsrc.v5.tar.Z>.

fonctionnement. Une analyse théorique n'est pas le but de ce travail. Une introduction se trouve dans le texte de Rao et Yip 1990[1].

2.4.3. Transformée de Karhunen-Loève (KLT : Karhunen-Loève Transform) :

L'avantage de cette transformée est qu'elle évite la corrélation entre les coefficients transformés. Le problème est qu'en général, on ne peut pas calculer la transformation discrète de Karhunen-Loève avec un algorithme de calcul rapide. Je n'ai pas mis en place cette transformée[2].

2.4.4. Transformée de Walsh-Hadamard (WHT: Walsh-Hadamard Transform) :

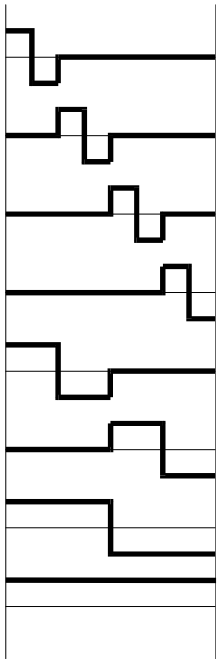
Dans la littérature, on associe souvent le nom de Walsh à la transformation de Hadamard, car les fonctions de Walsh sont les versions analogiques des lignes de la matrice de Hadamard. L'avantage de cette transformée est l'inexistence de coefficients multiplicatifs dans les étapes. Alors, le nombre d'opération est réduit simplement à $N \cdot \log_2 N$ additions et soustractions. Je n'ai pas implanté non plus cet algorithme.[3]

2.4.5. Transformée de Haar discrète (HDT: Haar Discrète Transform) :

L'avantage de cette transformée est également l'inexistence de coefficients multiplicatifs dans les étapes. Mais, dans ce cas, le nombre d'opérations est réduit extraordinairement à 2N - 1 additions et soustractions. L'avantage acquis par cette structure est par contre perdu dans la mise en ordre des coefficients.

Par exemple, les fonctions pour un HDT avec une dimension de 8 éléments donnent les fonctions suivantes :

[1] [RAO__90].
[2] Pour une information additionnelle, consulter [KUNT_80].
[3] Pour une information additionnelle, consulter [KUNT_80].



Les quatre premières suivent les détails hauts du signal, deux fonctions pour coder les détails moyens, un pour les détails gros; la dernière est pour la valeur moyenne.

La HDT est utilisée par contraste avec la DFT parce qu'elle est plus rapide[1]. Fondamentalement elle est liée aux techniques dénommées « wavelets ».[2]

[1]Pour une information additionnelle, consulter [KUNT_80].
[2]Une implantation peut être trouvée à <lip@s1.gov> par Loren I. Petrich.

2.5. Fenêtres :

Les fenêtres sont des méthodes très importantes pour éliminer les lobes latéraux (sidelobes) des estimateurs, mais elles ont aussi des autres applications. Une fenêtre peut être appliquée à un signal de la manière suivante :

$$x_N(k) = x(k) \cdot W(k + N/2)$$

La valeur N fait référence à la longueur de la tranche à considérer.

Normalement, le calcul est fait une fois avec la formule originelle de la fenêtre pour construire une table et ensuite, celle-ci est consultée chaque fois qu'il est nécessaire. Donc, le coût en programmation est très faible et il permet d'appliquer des fenêtres très sophistiquées sur une puce quelconque.

Voici schématiquement les fenêtres les plus utilisées...[1]

2.5.1. Fenêtre rectangulaire :

Il s'agit simplement de la partition pure et dure : tout si on est dans la tranche, rien dehors.

$$W_R(k) = \begin{cases} 1 & \text{pour } |k| \le N/2 \\ 0 & \text{partout ailleurs} \end{cases}$$

(On utilise sa présentation comme exemple pour les fenêtres qui suivent).

2.5.2. Fenêtre triangulaire :

Pas trop compliquée, elle est la plus simple des fenêtres qui amplifie les valeurs centrales, pénalisant les valeurs extrêmes (*sidelobes*).

$$W_T(k) = \begin{cases} 1 - 2|k|/N & \text{pour } |k| \le N/2 \\ 0 & \text{partout ailleurs} \end{cases}$$

Elle n'est pas trop utilisée parce que des résultats meilleurs peuvent être obtenus sans beaucoup plus de complexité.

2.5.3. Fenêtre Gaussienne :

Elle se définit comme :

$$W_G(k) = \begin{cases} \exp((-1/2)(2a|k|/N)^2) & \text{pour } |k| \le N/2 \\ 0 & \text{partout ailleurs} \end{cases}$$

[1] Pour une étude plus aprofondie avec les réponses fréquentielles correspondantes, on peut regarder [MARPL87].

Des résultats assez bons apparaissent avec a > 2.5.

2.5.4. Fenêtre Cosinusoïdale :

Sa formule est :

$$W_C(k) = \begin{cases} \cos(\pi k/N) & \text{pour } |k| \le N/2 \\ 0 & \text{partout ailleurs} \end{cases}$$

Elle sert de base aux fenêtres qui suivent :

2.5.5. Fenêtre de Hamming :

Couramment utilisée, elle est codifiée en tables pour éviter le calcul du cosinus.

$$W_H(k) = \begin{cases} \alpha + (1 - \alpha)\cos(2\pi k/N) & \text{pour } |k| \le N/2 \\ 0 & \text{partout ailleurs} \end{cases}$$

avec α= 0,54.

2.5.6. Fenêtre de Hanning :

C'est un cas de la fenêtre de Hamming avec α= 1/2.

$$W_h(k) = \begin{cases} 1/2(1 + \cos(2\pi k/N)) & \text{pour } |k| \le N/2 \\ 0 & \text{partout ailleurs} \end{cases}$$

Elle est aussi équivalente à une fenêtre cosinusoïdale carré [$\cos^2(\pi k/N)$] ou fenêtre de Von Hann.

En général, la plupart des applications approchent cette fenêtre par un polynôme. Par exemple, un polynôme du troisième ordre est couramment utilisé :

$$W(t) = 1 + 0.125t - 3.375t^2 + 2.25t^3 \qquad 0 <= t <= 1$$

Le résultat est pratiquement le même qu'avec la formule originale. Si l'on dispose de la mémoire suffisante, l'implantation la plus adéquate est de faire une table avec des valeurs déjà calculées.

2.5.7. Fenêtre de Blackman :

Si l'on généralise les fenêtres précédentes avec des cosinus pondérés, le résultat est:

$$W_B(k) = \begin{cases} a_0 + 2\sum_{l=1}^{L} a_l \cos(2\pi k l/N) & \text{pour } |k| \le N/2 \end{cases}$$

⌊ 0 partout ailleurs

L= (M-1) / 2, M étant le nombre de répliques décalées. Si L= 3, on l'appelle fenêtre de Nuttall. Avec L=2, on a normalement les paramètres suivants :

$W_B(k) = 0.42 + 0.50 \cos(2\pi k/N) + 0.08 \cos(4\pi k/N)$

2.5.8. Autres fenêtres :

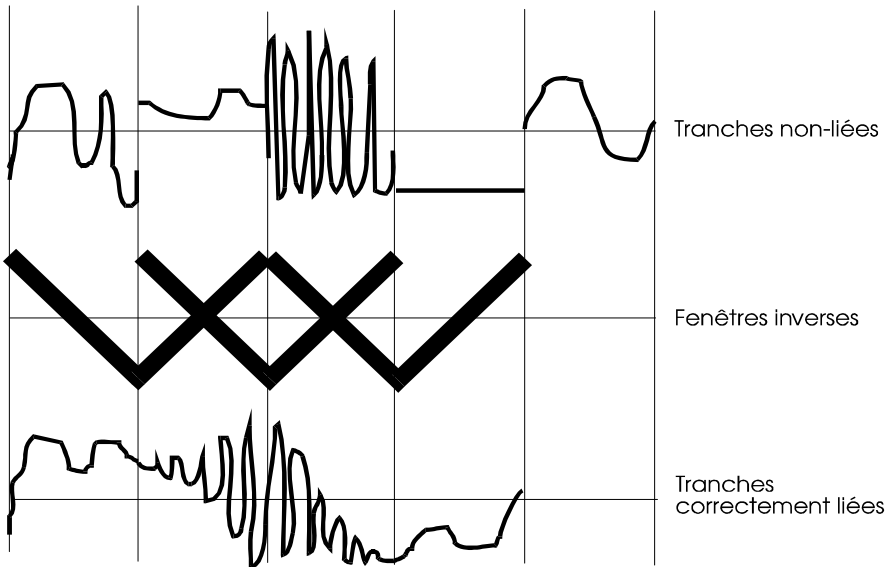
D'autres fenêtres comme la ***Fenêtre de Kaiser*** ou la ***Fenêtre de Bartlett*** donnent des résultats très bons mais sont assez compliquées à calculer.[1]Comme on l'a dit, on peut faire une table si l'on dispose de la mémoire suffisante.

2.5.9. Applications :

Comme cité précédemment, les fenêtres peuvent être utilisées dans différents domaines. L'application la plus connue est pour calculer les transformées, donnant plus d'importance aux valeurs centrales qu'aux extrêmes. Le spectre calculé est ainsi plus exact qu'avec une fenêtre rectangulaire (c'est-à-dire, aucune fenêtre).

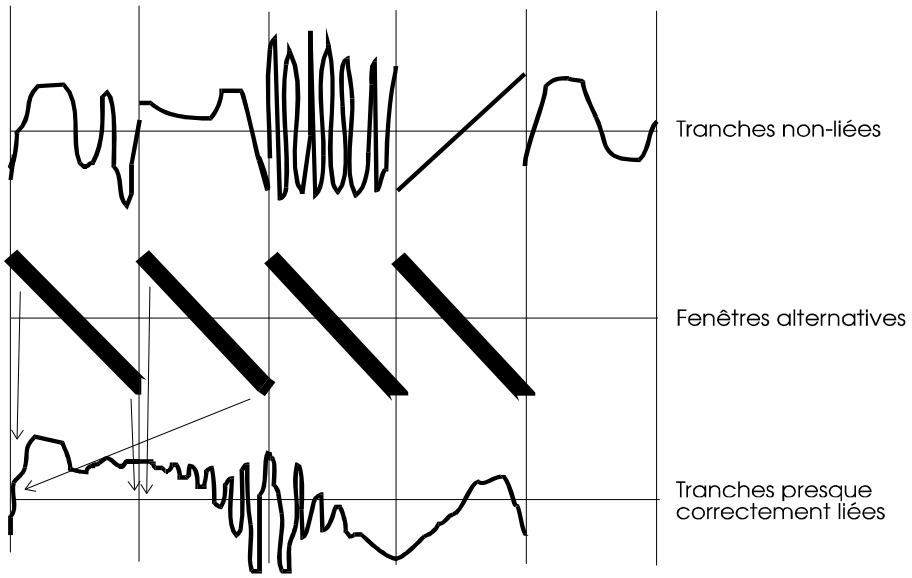
Une autre utilisation est pour faire la moyenne de deux tranches. Exemple : La compression TDHS, qui fait la moyenne de deux tranches consécutives dans une seule, sans une grande perte de la qualité.

En relation avec cela, les fenêtres sont aussi très importantes pour rendree continue la procédure de recomposition. Le premier cas se produit lorsque chaque tranche commence et finit par la même valeur (cas de la vitesse variable). L'utilisation de fenêtres inverses peut éliminer le bruit de coupe des tranches, ce qui rend superflues les coupes par passage par zéro et tout ce qui s'ensuit. Il s'agit simplement d'appliquer une fenêtre inverse (pour les images ci-dessous triangulaire, en pratique quelconque) à deux tranches consécutives:



[1]Regarder dans [KUNT_80] ou [HIGGI90].

Le deuxième cas arrive lorsque le commencement et la fin n'ont rien à voir (cas général) :



Cette dernière méthode a l'inconvénient de donner plus d'importance à la première partie des tranches qu'à la deuxième.

Le résultat est la disparition (totale dans le premier cas et partielle dans le deuxième) de discontinuités dans la procédure de recomposition dans les algorithmes de vitesse variable ou dans les algorithmes de compression dans le domaine spectral. On verra tout cela plus loin.

2.6. Analyse spectrale :

Une fois présentées les techniques nécessaires, on peut prendre en considération une analyse plus complexe de la parole. Le signal est souvent analysé selon des facteurs spectraux, telles que le spectre de fréquence ou la fonction d'autocorrelation, au lieu d'utiliser directement l'onde. Il y a deux raisons pour cela :

- Primo, le signal de la parole est considéré comme reproductible par la somme des ondes sinusoïdales, dont l'amplitude et la phase ne changent que lentement,

- Secundo, les facteurs critiques pour percevoir la parole par l'oreille humaine sont inclus principalement dans l'information spectrale, pour laquelle l'information de phase n'a pas de rôle important.

Le spectre est donc une des représentations les plus utilisées, car il permet d'apercevoir les formants des différents sons. Normalement, il est plus intéressant encore de faire l'enveloppe spectrale qui change plus lentement en fonction de la fréquence.

Différentes méthodes pour l'extraction de l'enveloppe ont été développées. Elles peuvent être divisées en analyse paramétrique et analyse non-paramétrique. L'avantage de l'analyse non paramétrique est qu'elle ne nécessite pas un modèle paramétrique ajusté et les calculs sont souvent plus simples.

Parmi les méthodes non paramétriques, l'analyse fréquentielle peut être classée de la plus petite à la plus grande complexité selon les techniques suivantes :

- ***Passage par zéro*** : C'est la façon la plus rapide pour calculer la fréquence, permettant l'implantation sur un matériel très simple. Le problème est la susceptibilité au bruit.

- ***Filtres passe-bande en cascade*** : C'est une autre manière assez simple d'obtenir l'information spectrale.

- ***Spectre par FFT*** : L'existence de la transformée rapide, permet aujourd'hui d'utiliser cette technique exacte pour l'analyse fréquentielle.

- ***Cepstre (spectre logarithmique)*** : L'enveloppe obtenue par le cepstre est plus exacte mais deux FFTs et une transformation logarithmique sont nécessaires. Ceci est applicable pour le calcul de la période du fondamental (pitch).

Dans la présentation des caractéristiques de la parole, on a vu l'importance des formants que l'on peut voir dans le spectrogramme. Le problème est que l'articulation et beaucoup d'autres facteurs font qu'une représentation de l'évolution de la parole avec le temps serait plus valable...

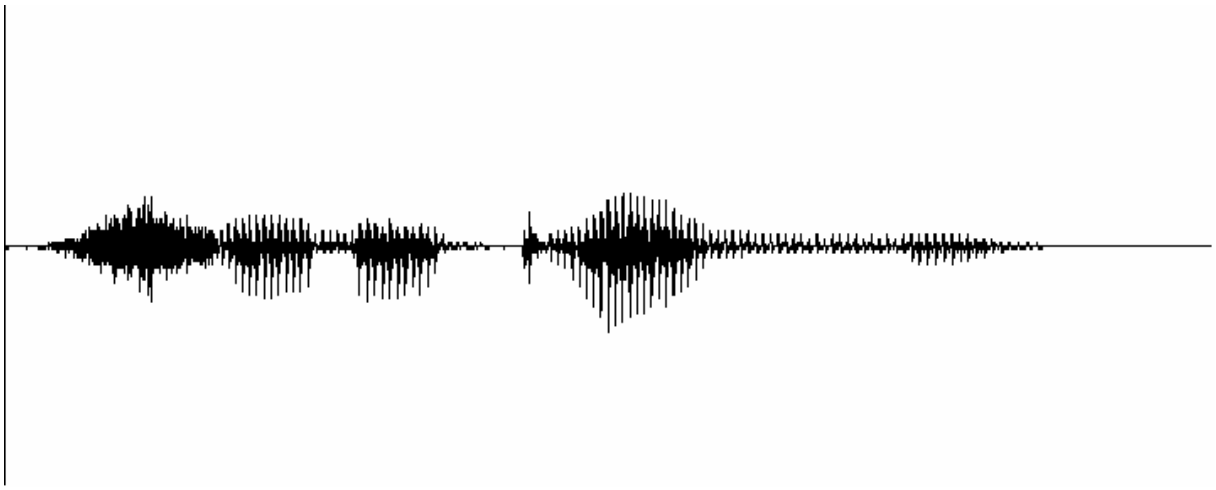
2.6.1. Sonagramme :

Le sonagramme ou spectrogramme sonore est une méthode pour dessiner le spectre du signal à court-terme, en fonction du temps, en utilisant un graphe de densité. Les fréquences de grande amplitude sont dessinées avec un point plus foncé

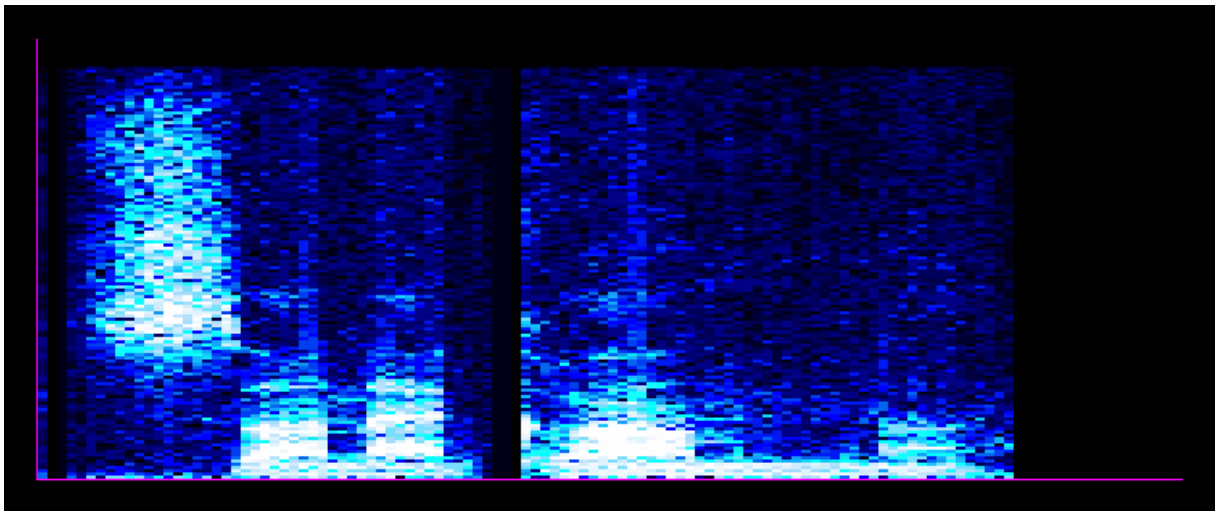
et les fréquences moins importantes avec des points plus clairs. Originellement, le sonagramme était fait analogiquement, mais maintenant il est fait numériquement grâce à la puissance de la technologie actuelle et de la FFT.

Selon la résolution de chaque spectre, l'analyse peut être à bande étroite (narrow-band) ou à large bande (wide-band) . À bande étroite, on utilise une fréquence de 250 Hz (4 ms de fenêtre) et le résultat est l'apparition de stries dans le sens horizontal. À large bande, on utilise une fréquence de 50 Hz (20 ms de fenêtre) et le résultat est l'apparition de stries dans le sens vertical.

Le signal suivant représente le mot « sonagramme » :



Et son sonagramme réalisé avec une longueur de fenêtre de 10 ms. :



Ce sonagramme a été extrait de l'outil Accordion et a été calculé appliquant une FFT pas trop optimisée.

2.7. Extraction de paramètres :

Pour un grand nombre d'applications de la parole, il est très important de calculer certains autres paramètres:

2.7.1. Puissance moyenne :

Aussi nommée énergie moyenne, c'est elle que l'on connaît populairement sous le terme vulgaire de volume de l'enregistrement. Le traitement est plus difficile si la puissance moyenne est basse, parce que le bruit aura un poids important par rapport au signal de la parole. Pour la calculer, la méthode la plus simple est d'utiliser la formule des carrées :

$$P_N = SQRT\ [\ \mathring{a}_{n=1,\,N}\ x^2(n)\ ]$$

Où SQRT veut dire racine carrée. Selon le N choisi, on peut parler de puissance moyenne à court ou à long terme. Évidemment, les silences doivent donner une valeur nulle.

2.7.2. Détection de son/silence (période de parole) :

La détection de pauses dans la parole est importante pour des applications diverses : arrêt automatique, compression, etc. La présence de bruit rend cette tâche assez difficile; en plus, les consonnes au commencement de la période de parole et les voyelles de basse énergie sont spécialement compliquées à détecter. D'autres bruits (comme les bruits des soupirs, de toux, des sons des changements de page) doivent être aussi ignorés.

Une période de parole (son) est usuellement détectée par le fait que l'énergie moyenne a court-terme excède un niveau pendant une longueur déterminée. Le point de commencement est établi un peu avant cette détection.

2.7.3. Détection du pitch :

La période du fondamental (appelé communément le pitch) est un paramètre très important pour la synthèse de la parole ; l'oreille est en effet très sensible à ses variations, lesquelles constituent la prosodie ou timbre du parleur.

L'extraction du pitch a été une tâche particulièrement difficile pour trois raisons : Premièrement, la vibration des cordes vocales n'a pas nécessairement une périodicité complète, particulièrement au commencement des son voisés. Deuxièmement, il est difficile de séparer le pitch des effets du trait vocal. Troisièmement, la plage de dynamique de la fréquence du fondamental est très grande.

La plupart des procédures étudiées comportent une de ces deux erreurs : soit elles dupliquent le pitch, soit elles le divisent par deux. Normalement ces cas ne sont

pas très ennuyeux mais il y des applications où la robustesse du détecteur est très importante.[1]

Ces procédures peuvent être divisées en trois grands groupes : procédures sur le signal, procédures de corrélation et procédures spectrales.

Les procédures sur le signal consistent à détecter les pics du signal ou les passages par zéro. Les procédures de corrélation utilisent des fonctions d'autocorrélation, LPC et d'autres méthodes. Les procédures spectrales font utilisation du spectre, du cepstre ou du maximum de l'enveloppe spectrale.

Voyons les méthodes les plus courantes[2] :

- ***Autocorrélation par produit*** : Il s'agit de calculer la fonction d'autocorrélation sur une tranche de N échantillons qui recouvre plusieurs périodes du fondamental :

  $$r(k) = \sum_{n=1,\, N-1-k} x(n) \cdot x(n+k) \quad , \qquad k = 0, 1, ..., K$$

  La valeur maximum de k donnera le pitch. Le problème est que, pour obtenir une bonne mesure, il faut essayer pour un grand nombre de k, chaque fois pour tout le signal. Le signal est considéré avec pitch si[3]

  $$r_{max} / r(0) > 0.25$$

- ***Autocorrélation par différence*** : Connue sur le sigle AMDF[4] (Average Magnitude Difference Function), elle utilise une expression plus simple :

  $$r(k) = \sum_{n=1,\, N-1-k} | x(n) - x(n+k) | \quad , \qquad k = 0, 1, ..., K$$

  Puisqu'une soustraction est plus rapide qu'un produit, elle présente un petit avantage, mais les résultats ne sont pas aussi bons que dans le cas précédent.

- ***Méthode du spectre*** : Une fois le spectre calculé, il s'agit simplement d'en prendre le maximum. Cette méthode est la plus intéressante si l'on a déjà calculé la FFT du signal.

- ***Méthode par filtre inverse*** : Connue sous le sigle SIFT (Simplified Inverse Filter Tracking-algorithm), il s'agit d'appliquer un filtre passe-bas. Ensuite on procède à une analyse LPC pour définir un filtre inverse. Une des méthodes précédentes est appliquée au résidu.

- ***Méthode du cepstre*** : Elle est similaire a la méthode du spectre, mais les résultats sont encore meilleurs. La complexité de calcul du cepstre est plus du double de celle du spectre.

Une amélioration de la prédiction du pitch peut être obtenue selon Kroon et Atal[5] à partir de l'équation suivante d'un prédicteur impair de pitch :

[1] Un algorithme de détection du pitch robuste a été presenté par [CHEN_88].
[2] [HASSA85]
[3] [MARKE80]
[4] [ROSS_74]
[5] [KROON91]

$$P(z) = 1 - \sum_{k=-(p-1)/2 \ldots (p-1)/2} b(k) z^{-(M+k)} \quad , \quad p = 1, 3 \ldots$$

Où le retard M est exprimé en un numéro d'échantillons correspondant à 2 et 20 ms (50..400 hz), et où les b(k) représentent les coefficients du prédicteur.

Comme remarque finale, pour toutes les méthodes étudiées, on peut remarquer que la plage à étudier peut être restreinte de 2 à 20 ms pour réduire le coût en programmation. Une autre façon d'accélérer les calculs, est d'évaluer une valeur de chaque 4, 8 ou 16 n valeurs pour approcher la valeur et après s'approcher avec une résolution plus grande.

2.7.4. Détection de son voisé/non voisé :

L'existence d'une composante périodique claire dans les sons voisés rend cette tâche un peu plus fiable, mais elle nécessite une quantité considérable de calculs. Généralement, comme on l'a vu, cette détection est inclue dans la détection du pitch. Par exemple, dans la méthode de corrélation par produit, le signal est considéré voisé si

$$r_{max} / r(0) > 0.25.$$

D'autres niveaux peuvent être choisis pour les autres méthodes.

2.8. Corrélation :

Pour calculer la corrélation entre deux signaux a(t) et b(t), il est plus efficace d'utiliser une transformée comme suit :

$$a(t), b(t) \Rightarrow DFT \Rightarrow A(n), B(n) \Rightarrow PRODUIT \Rightarrow C(n) \Rightarrow iDFT \Rightarrow c(t)$$

Où DFT et iDFT représentent, respectivement, la Transformée de Fourier Discrète et son inverse. D'autres transformées peuvent aussi être appliquées.

Le résultat de la corrélation sera :

$$Correl(a, b)= c(t)$$

Les méthodes de corrélation entre signaux seront étudiées plus profondément dans le chapitre de mesures de qualité.

Chapitre 3

COMPRESSION

La compression de la parole a subi une avance impressionnante pendant les dernières décennies en raison de la nécessité de réduire le débit binaire (bps : bits par seconde) dans le domaine des télécommunications. Aujourd'hui, ce progrès continue mais la compression est aussi très utilisée pour le stockage, soit dans les appareils audio commerciaux, soit dans les ordinateurs. Les anciennes bandes analogiques d'audio sont progressivement substituées par le stockage numérique soit dans les DATs ou dans les CDs.

Pour le stockage, les méthodes ne requièrent pas d'être si robustes que pour les télécommunications où les erreurs du canal peuvent corrompre complètement[1] le codage, étant donné que, dans le cas du stockage, une perte d'information est très étonnante. C'est pour cela que de nouvelles méthodes se sont imposées pour le stockage d'informations audio. Récapitulons toutes ces méthodes...

3.1. Notions sur la compression :

La compression est la procédure par laquelle on peut réduire le débit de données à transmettre ou à stocker sans une perte cruciale de l'information. Le processus complet de compression / décompression est appelé couramment COMPAND pour COMPress / exPAND et le circuit correspondant CODEC (CODing and DECoding device).

Quand on parle de compression, on fait référence fondamentalement à une compression sans perte (lossless), dont le résultat après décompression est complètement équivalent à l'original (compression réversible). Ces méthodes de compression sont utilisées pour les données informatiques dont l'exactitude est cruciale. Normalement, les facteurs de compression accomplis par ces algorithmes n'aboutissent qu'à des valeurs faibles. Il ne faut pas exprimer qu'il y a une limite théorique, l'entropie, et qu'elle est complètement dépendante de la source à comprimer : Donc des affirmations comme celle de la compression WEB (pour laquelle on disait qu'elle arrivait toujours à 16:1), et beaucoup d'autres tromperies, ont été dénoncées par la théorie et par la pratique.

[1]Des méthodes de contrôle pour resoudre cela se trovent dans [ATUNG93]

Pour la parole, cependant, il est plus intéressant d'avoir d'une compression supérieure même si elle n'est pas exacte. De cette façon, on perd un peu d'information, mais jusqu'à un certain niveau, cela est peu perceptible par l'oreille humaine. C'est avec des compressions extrêmes que l'on perçoit une perte progressive de qualité.

Bien que les résultats requis pour ce travail n'aient pas besoin d'une qualité totale ou hi-fi (nommée couramment de large bande), l'étude des langues nécessite un son clair et parfaitement compréhensible, pareil à celui d'une cassette habituelle. Donc, pour notre application, on a besoin d'une qualité importante. Mais tout cela n'implique pas que nous ne surveillons pas d'autres techniques, lesquelles peuvent donner des idées utiles pour les algorithmes qui seront vraiment développés.

3.2. Compressions réversibles sans perte d'information (lossless) :

La plupart des utilisateurs d'ordinateurs personnels sont habitués à utiliser des programmes de compression avec les formats comme LHA, ZOO, ARJ et ZIP. Une bonne référence est le livre de Nelson[1].

Les trois méthodes les plus utilisées dans ces programmes[2]sont :

3.2.1. Le code de Huffman :

La compression Huffman est une compression statistique qui donne une réduction de la longueur moyenne utilisée pour représenter les symboles d'un alphabet. Le code de Huffman est un exemple d'un code qui est optimal où toutes les probabilités des symboles sont des puissances entières de ½. La procédure est la suivante :

1. Arranger tous les symboles en ordre de probabilité d'occurrence.

2. Successivement, combiner les deux symboles de la probabilité la plus petite pour former un nouveau symbole. On construit alors un arbre binaire dont chaque noeud est la probabilité de tous les noeuds en dessous.

3. Tracer un chemin à chaque feuille, et annoter la direction de chaque noeud.

Pour chaque distribution, il y a beaucoup de codes différents mais, si l'on suit certaines règles, il est possible de construire un arbre de Huffman canonique. Le problème est qu'à mesure que la longueur de chaque bloc s'agrandit, la complexité augmente exponentiellement.

3.2.2. La Compression Arithmétique :

Quoiqu'il semblait que le code de Huffman fut le moyen parfait pour comprimer des données, cela n'est pas exactement le cas. Comme on vient de le dire, ces méthodes sont optimales seulement si les probabilités sont des puissances entières de ½. Le codage arithmétique, cependant, n'a pas cette restriction. Il arrive au même effet en traitant le message comme une unité simple et il atteint les limites de compression de données par l'entropie de la source.

Le codage arithmétique travaille en représentant toutes les données comme un nombre réel entre 0 et 1. À mesure que le message est plus long, l'intervalle requis pour représenter la valeur est chaque fois plus petit, et la compression s'approche de celle de la théorie de l'entropie.

Un problème apparaît car on ne connaît pas a priori les fréquences exactes des symboles. De plus, il est plus intéressant d'avoir les fréquences de symboles isolées et

[1][NELSO91]

[2]La plupart d'explication de ces techniques est traduit directement du texte de Peter Gutman trouvé dans l'Internet en <pgut1@cs.aukuni.qc.nz>

les probabilités intersymboles également. Les meilleurs compresseurs disponibles aujourd'hui utilisent ces approches :

- Le DMC (Dynamic Markov Coding) commence avec un modèle d'ordre zéro et il s'étend graduellement au fur et à mesure que la compression progresse.

- Le PPM (Prediction by Partial Matching) regarde une coïncidence du texte dans un contexte d'ordre n. Si aucune correspondance (match) est trouvée, il descend à un contexte d'ordre n-1, jusqu'à ce qu'il arrive à zéro.

Les deux techniques présentées donnent de bons résultats[1] mais l'algorithme n'est pas encore trop populaire. La raison fondamentale pour laquelle ce codage n'est pas très utilisé est la grande quantité de ressources (mémoire et CPU) qu'il utilise. Une autre raison non dédaignable est que quelques variations sont enregistrées (comme le Q-coder d'IBM). 5% ou 10% de compression additionnelle ne justifient pas de payer les redevances correspondantes.

3.2.3. La Compression Substitutionnelle :

L'idée de fond d'un compresseur substitutionel, est de remplacer l'occurrence d'une phrase particulière, ou groupe de données par une référence à une occurrence préalable de cette phrase. Il y a deux schémas fondamentaux, LZ78 et LZ77, tous les deux développés par Abrahmam Lempel et Jakob Ziv (d'où les caractères L et Z).

- Le schéma LZ78 introduit les phrases dans un dictionnaire et, chaque fois que l'on trouve l'occurrence d'une séquence de caractères dans le dictionnaire, on code l'index du dictionnaire à la place de toute la phrase. Si la phrase n'existe pas dans le dictionnaire, on l'ajoute.

  Il y a différentes manières de gérer le dictionnaire; celle de Terry Welch (LZW) est assez utilisée parce que le dictionnaire est transmis en même temps que les données sont comprimées, et il est reconstruit graduellement à la décompression. Comme le dictionnaire doit être fini, il y a des compresseurs qui actualisent continuellement ce dictionnaire; c'est le cas des compressions LZC dans le monde UNIX. D'autres versions (LZMW[2]) permettent de concatener des phrases dans le dictionnaire pour arriver à des compressions plus grandes.

- Le schéma LZ77 est plus simple ; il maintient la trace des n dernières données et, lorsqu'une phrase est retrouvée, il code deux valeurs qui sont la position et la longueur de la phrase dans les données préalables. Ces n données forment la fenêtre de glissement. Plus la fenêtre est longue, plus les coïncidences seront grandes et fréquentes mais la position aura besoin de plus de bits pour être codée et également de plus de temps de calcul pour l'examiner.

[1]Pour une implantation efficace uniquement d'ordre 0, contacter via anonymous ftp à <ftp.cpsc.ucalgary.ca> dans /pub/projects/arithmetic.coding

[2]Pour une implantation, contacter avec Dan Bernstein: <djb@silverton.berkeley.edu>

> Cet algorithme a deux avantages fondamentaux : seule la puissance du processeur lors de la compression détermine jusqu'où l'on peut examiner (cela donne une compression faible si l'on n'a pas un processeur puissant, mais il la rend encore possible) ; le deuxième avantage est que la décompression est instantanée.

Les deux schémas LZ77 et LZ78 ont un autre avantage: d'autres méthodes comme le codage Huffman peuvent être appliquées après. Tout cela donne de bons résultats qu'on voit dans des variations comme ARJ, LHA, ZIP, ZOO, etc.

Il y a des nouveaux schémas WDLZW[1] avec des compressions supérieures et les recherches continuent sur la matière avec l'unique limite de l'entropie.

[1] [JIANG92]

3.3. Compression avec perte d'information (lossy) :

Le signal de la parole exhibe de grandes redondances dues au mécanisme physique du trait vocal. Les rangs fréquentiels du processus d'audition sont aussi restreints. De plus, les formants les plus importants cachent les bruits de certaines fréquences et les différences de phase ne sont pas détectées par l'oreille humaine pour la plupart des fréquences. Tout cela donne la possibilité de comprimer la parole pour obtenir des résultats de bonne qualité avec un stockage minimal. Un étude sérieuse (quoiqu'un peu obsolète) de toutes ces caractéristiques (bandes intéressantes à coder, effet sur le SNR et l'intelligibilité, etc.) a été réalisée par Anderson et Bodie[1].

3.3.1. Classification :

Les méthodes pour le codage de la parole peuvent être classées entre codeurs d'onde (waveform coding) et codeurs de source ou vocodeur (analysis-synthesis method). Dans la première méthode, le signal est représenté directement et, dans la deuxième, certains paramètres en sont extraits.

Dans les codeurs d'onde, on peut faire la distinction entre les techniques de codage dans le domaine temporel et le domaine fréquentiel.

Une classification stricte est impossible car il s'agit de familles d'algorithmes plus que d'algorithmes concrets, mais dans la bibliographie[2], elle est souvent présentée comme suite :

[1] [ANDER75]
[2] [BOITE87]

1. *Représentation directe (codeurs de l'onde) :*
   - a) *Domaine temporel :*
     - (1) *Codage des Silences*
     - (2) *PCM, 64-56 kbps*
       - (a) *A-Law*
       - (b) *μ-Law*
     - (3) *DPCM, 24 kbps*
     - (4) *DM,*
     - (5) *ADM, 16 kbps*
       - (a) *CFDM (Jayant)*
       - (b) *CVSDM (Greekles)*
       - (c) *HIDM (Winkler)*
       - (d) *2d-IADM (Tombras)*
       - (e) *ADxyM*
     - (6) *ADPCM, 16-32 kbps*

   - b) *Domaine fréquentiel :*
     - (1) *SBC, 16 à 24 kbps*
     - (2) *ATC (Zelinski-Noll), 9,6 kbps*
     - (3) *Compression avec d'autres transformées 2-8 kbps*
     - (4) *Contractive Speech Coding, 0,5 kbps*

2. *Représentation paramétrique (codeurs de source par analyse-synthèse), 2-4 kbps.*
   - (1) *Vocodeurs*
     - (a) *LPC, 2,4 à 16 kbps*
       - (i) *APC, 9,6 kbps*
       - (ii) *GSM, 13 kbps*
     - (b) *Quantification Vectorielle*
       - (i) *CELP, 2-8 kbps*
       - (ii) *LD-CELP, 16 kbps*
       - (iii) *MPEG, 128 kbps*
   - (2) *Codage Sinusoïdal, 4,8 kbps*

3. *Autres techniques :*
   - a) *Trellis Coding*
   - b) *TDHS*
   - c) *Méthode de Roberts*

Même pour moi, qui suis déjà habitué, tout cela constitue un maremagnum de sigles et acronymes. J'ai fait un effort pour donner la signification de toutes ces sigles au moment de leur utilisation.

3.3.2. Normes :

Les algorithmes les plus utilisés et essayés s'inscrivent généralement dans une norme. Le tableau suivant montre quelques unes des normes existantes :

Nom	Comité	Technique	Débit binaire	Application
G.711	CCITT	μ-Law et A-Law	56 ou 64 kbps	Téléphonie (qualité moyenne)
G.721	CCITT	ADPCM	32 kbps	Bande courte (haute qualité)
G.722	CCITT	SBC	48, 56 ou 64 kbps	Bande large (haute qualtité)
G.723	CCITT	ADPCM	24 ou 40 kbps	Bande large

				(bonne qualité)
G.728	CCITT	LD-CELP	16 kbps	Bande courte (bonne qualité)
FS-1015	US DoD's Federal-Standard	LPC avec 10 coefficients	2.4 kbps	Téléphonie (qualité basse)
FS-1016	US DoD's Federal-Standard	CELP 3.2a	4.8 kbps	Téléphonie (qualité moyenne)
European Cellular Standard	GSM (Groupe Spéciale Mobile)	LPC	13 kbps	Téléphonie mobile (bonne qualité)
American Cellular Standard		VSELP	8 kbps	Téléphonie mobile (bonne qualité)
CD-PCM	(Commercial)	PCM 44.1 khz, 16 bits stéréo	1411.1 kbps	Enregistrement numérique audio professionnel
CD-Level A	(Commercial)	ADPCM 37.8 khz 8 bits	302.4 kbps	Enregistrement numérique audio (qualité CD)
CD-Level B	(Commercial)	ADPCM 37.8 khz 4 bits	151.2 kbps	Enregistrement numérique audio (qualité hi-fi)
CD-Level C	(Commercial)	ADPCM 18.9 khz 4 bits	75.6 kbps	Enregistrement numérique audio de bonne qualité
ASPEC (courantement dans MPEG)	Frauenhofer Institut (Allemagne)	VQ	128 kbps	Enregistrement numérique audio de bonne qualité (qualité CD)
MPEG (Motion Picture Expert Group)	ISO	VQ	128 kbps	Audio pour Video (qualtié hi-fi)

De nouvelles normes apparaissent en même temps que de nouvelles techniques sont inventées et testées.

3.3.3. Techniques :

L'exposé antérieur n'est pas exact parce que, normalement, la plupart des techniques sont mélangées et l'on parle de méthodes mixtes. Les techniques les plus utilisées sont :

- ***Quantification non linéaire*** : L'amplitude est comprimée par une transformation non linéaire (logarithmique par exemple), basée sur les caractéristiques statistiques de l'amplitude de la parole.
- ***Quantification adaptative*** : Le pas de quantification varie en concordance avec la variation d'amplitude.
- ***Codage prédictif*** : Une prédiction est fait selon les valeurs immédiatement antérieures et les valeurs distantes (qui suivent la corrélation du pitch). Seule la différence entre la valeur prédite et la valeur réelle est codée.

- ***Division fréquentielle*** : Plus de données sont utilisées pour coder les bandes fréquentielles les plus importantes que pour les autres.

- ***Transformée*** : La transformée, au lieu du signal, est codée, puis décodée et, à la fin, transformée inversement.

- ***Quantification vectorielle*** : Au lieu de coder des échantillons simples, des blocs ou groupes sont codés tandis que certaines combinaisons d'échantillons sont presque impossibles.

Ces techniques seront mieux expliquées une à une mais toute cette introduction sert à connaître un peu la terminologie et à percevoir la grande quantité de travail déjà fait et qu'il reste encore à faire dans le domaine.

3.4. Méthodes dans le domaine temporel :

Initialement dû à la puissance des premiers processeurs et à l'application téléphonique, aucun retard ou calcul complexe ne pouvait être essayé. Donc, ces méthodes étaient les uniques possibles.

3.4.1. Codage des silences (run-length coding):

Une technique très simple est de détecter les séries d'échantillons (en anglais runs) en dessous d'une limite déterminée minimale, pour laquelle on peut prendre toutes ces valeurs comme valeurs nulles. On mettra le nombre de valeurs nulles au lieu des dites valeurs.

Dans la parole, il y a beaucoup de silences ou de pauses entre les mots et les phrases. Donc, la qualité résultante n'est pas mauvaise si on choisit une limite faible. Le problème fondamental est que le débit réussi est complètement variable selon le signal à traiter et les compressions n'arrivent qu'aux taux 2:1 au maximum.

Dans l'outil Accordion j'ai réalisé une version de cet algorithme[1] qui a les paramètres suivants :

- SILENCE_LIMIT (3%) : Limite de l'amplitude maximale pour considérer un échantillon comme du silence.
- START_THRESHOLD (5) : nombre minimal d'échantillons successifs considérés comme du silence.
- STOP_THRESHOLD (2) : nombre d'échantillons consécutifs non considérés comme des « non silences » qui arrêtent le codage de la série.
- SILENCE_CODE (0xFF) : octet spécial pour coder le commencement d'une série.
- SUBSTSILENCE_CODE (0xFE) : octet mis au lieu du précédent quand on a réellement la valeur 0xFF.

Ces paramètres sont ajustés pour une fréquence de 8 khz et arrivent à une compression de l'ordre de 2:1 si le volume de bruit n'est pas trop important. Les paramètres START_THRESHOLD et STOP_THRESHOLD doivent être ajustés pour la fréquence concrètement utilisée.

3.4.2. Les algorithmes et normes PCM :

Les algorithmes PCM (Pulse Code Modulation) sont les plus communs des algorithmes étudiés. Pour donner un exemple, quand on parle par téléphone, on utilise ce système (7 ou 8 bits et 8 khz). Ceci permet une largeur de bande jusqu'à 4 khz et une dynamique de 42-48 dB. Un autre exemple quotidien sont les CDs, qui fournissent une meilleure qualité avec 16 bits et 44.1 khz. Ceci signifie 96 dB de dynamique et une bande de 20 hz à 20 khz, suffisante pour l'oreille humaine. Le coût requis est d'environ

[1]La source en C++ se trouve dans le fichier « silence.cpp »

178 Koctets par seconde (16 * 44100 * 2 (stéréo) = 1411.1 kbps) au lieu de 8 koctets par seconde (64 kbps) du téléphone.

Ces quantités font que la quantification est faite non uniformément : les signaux de plus petites amplitudes sont quantifiés avec une plus grande résolution. Le résultat est qu'une équivalence de 14 bits de dynamique peut être fournie par le convertisseur A/N comprimé en 8 bits. À la restitution, on fait la correspondance inverse pour le convertisseur N/A.

Deux variantes (incluses dans la norme CCITT G.721) se sont répendues dans la téléphonie: La norme américaine (μ-Law), utilisé aux États-Unis et au Japon et la norme européenne (A-Law) utilisée dans le reste du monde et dans les communications internationales. Elles sont toutes deux des variations d'une correspondance exponentielle.

Ces algorithmes sont normalement appliqués à 8 khz sur 8 bits (64 kbps) ou sur 7 bits (56 kbps) inclus, comme on l'a dit, dans la norme G.721 de la CCITT.

3.4.2.1. m-law :

L'équation est modifiée comme suit :

$$F(x)= \frac{(\log(1 + \mu |x| / x_{max}) * \operatorname{sgn} x)}{\log (1 + \mu)} \qquad |x| / x_{max} <=1$$

On obtient une compression d'autant plus grande que μ est grand. Typiquement, les valeurs de μ peuvent varier entre 100 et 500 ; normalement, le plus utilisé est $\mu=255$ ($2^{8\ bits}$-1).

3.4.2.2. A-law :

Comme l'équation précédente, on a celle de l'A-law:

$$F(x)= \frac{(A |x| * \operatorname{sgn} x)}{(1 + \ln A)} \qquad |x| / x_{max} <=1/A$$

$$F(x)= \frac{((1 + \ln (A |x| / x_{max})) * \operatorname{sgn} x)}{(1 + \ln A)} \qquad 1 <= |x| / x_{max} <=1$$

La norme européenne pour la téléphonie est:

- Fréquence d'échantillonnage 8 khz.
- Loi de compression : A= 87.56
- Quantification sur 256 niveaux (8 bits).

- Filtre de Garde, bande passante de 100 à 3400 Hz.
- Débit binaire de 64 kbps.

3.4.2.3. Réalisations :

Les résultats sont en général d'un gain important du SNR (Signal to Noise Ratio) vis à vis de la PCM linéaire, et la qualité reste, comme on la connaît, celle du téléphone. À cet égard, aucune différence significative n'existe entre les deux normes (la μ-law et la A-law).

J'ai mis dans l'application Accordion trois variantes des algorithmes précédents : un log-PCM général, la μ-law et la A-law[1]. Des mises en places sur DSP sont très courantes étant donnée la simplicité du codage, soit par formule soit par table.[2]

Par exemple, la table de codification pour la μ-Law à 8 bits (μ = 255) peut résulter d'une chose aussi simple que cela :

```
static int exp_lut[256]=
   { 0,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3,
     4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
     5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
     5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
     6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
     6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
     6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
     6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
     7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
     7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
     7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
     7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
     7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
     7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
     7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
     7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
   };
```

Et pour le décodeur :

```
static int exp lut[8]= { 0, 132, 396, 924, 1980, 4092, 8316, 16764 };
```

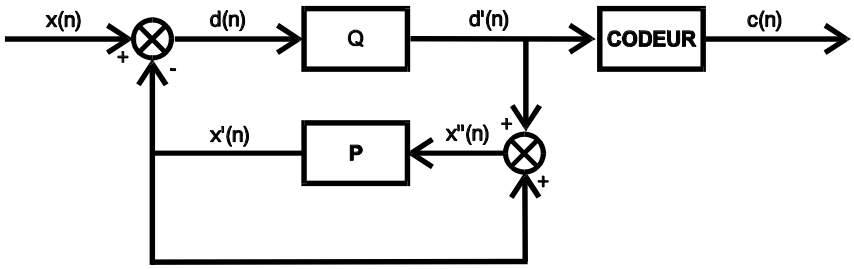
Ceci ne représente dans l'ensemble que 262 octets de mémoire, ce qui est vraiment faible.

3.4.3. L'algorithme DPCM :

Le principe de la codification DPCM (Differential Pulse Code Modulation) consiste à faire une prédiction numérique du signal à coder. De cette façon, on code la différence entre chaque échantillon et la valeur respective estimée pour cet échantillon. Cette technique est fondée sur le fait qu'il y a une forte corrélation entre des échantillons contigus de parole; on peut même dire que cette corrélation se maintient entre deux échantillons qui sont séparées par plusieurs intervalles d'échantillonnage. Le sens de cette corrélation élevée est que, en termes moyens, le signal ne change pas rapidement entre deux échantillons, donc la différence entre les échantillons contigus a une variance moindre que celle du signal lui-même.

[1]Ses sources en C++ se trouvent dans les fichiers « logpcm.cpp » et « u_a_law.cpp »
[2]Une implantation sur la famille TEXAS TMS 370 peut être trouvée dans [PAGNU89]

Formellement, pour le codeur, on a le schéma suivant où Q représente la Quantification et P la Prédiction.



$$d(n)= x(n) - x'(n)$$
$$d'(n)= d(n) + e(n)$$
$$x''(n)= x'(n) + d'(n)$$

Et donc :

$$x''(n)= x(n) + e(n)$$

Où e(n) est l'erreur de quantification.

De là, il est possible d'observer que x''(n) ne dépend pas du prédicteur P et ne diffère de x(n) que par l'erreur de quantification, c'est à dire e(n). Donc, si le prédicteur est suffisamment adéquat, x'(n) sera sensiblement égal à x(n) et il aura ainsi une variance moindre que la variance de x(n).

Le filtre de prédiction P a pour fonction de transfert la fonction P(z) telle que:

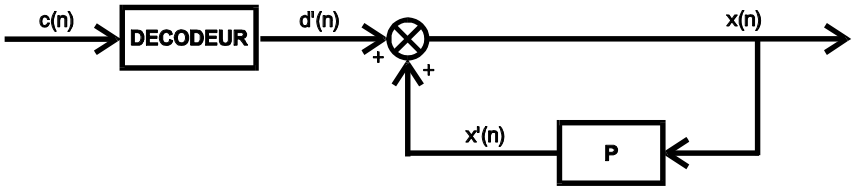
$$P(z)= \sum_{1..p} a_i z^{-i}$$

L'ordre p du filtre et ses coefficients a_i doivent être choisis pour minimiser la puissance du signal e(n). Pour le signal de parole, les coefficients suivants sont les plus simples proposés[1] :

$$a_1= 1,936$$
$$a_2= -1.553$$
$$a_3= 0.4972$$

Des ordres supérieurs sont aujourd'hui plus utilisés, car la puissance des processeurs le permet.

Par contre, le décodeur sera simplement :



[1][MACHA94]

Le codeur/décodeur peut appliquer une loi PCM logarithmique que l'on a décrite.

3.4.4. La modulation Delta (DM: Delta Modulation) :

Si l'on adopte une fréquence d'échantillonnage assez supérieure à la fréquence donnée par le théorème de Nyquist, il est possible d'utiliser une version simplifiée du codage DPCM (MIC-Différentiel) : la modulation delta. Pour que la modulation delta représente le signal d'entrée d'une façon satisfaisante, il est nécessaire que des échantillons consécutifs soient très corrélés et, pour cela, il faut imposer une fréquence d'échantillonnage élevée. La modulation delta de base utilise un quantificateur (qui est un DPCM) avec seulement deux niveaux (1 bit). Il est donc difficile de suivre de grandes variations brusques du signal d'entrée.

Le grand avantage de la modulation delta dans le domaine de la compression provient du fait d'avoir une sortie sur 1 bit. De cette façon, le taux de transmission de la modulation delta est égal à la fréquence d'échantillonnage.

La modulation DELTA présente, cependant, des difficultés pour suivre correctement le signal d'entrée. Il est obligatoire que la pente maximale du signal soit plus petite que la capacité d'accompagnement du modulateur par période d'échantillonnage. Si cette condition n'est pas respectée, il advient ce qu'on appelle une « saturation de pente » .

Pour obtenir des signaux assez continus, nous avons, dans la pratique, les conditions suivantes pour obtenir un accompagnement correct du signal d'entrée:

$$f_{échant} >> f_{Nyquist} \qquad (@\ 20\ fois)$$

$$D \geq 2 * \Pi * A * f_{Nyquist} / f_{échant}$$

En conséquence, ce que l'on gagne avec le codage sur un seul bit, on le perd avec la nécessité d'augmenter la fréquence d'échantillonnage.

3.4.4.1. La modulation Delta adaptative (ADM: Adaptive Delta Modulation) :

Aussi connue sous les sigles CVSD (Continuous Variable Slope Detection), c'est une façon efficace de résoudre le problème de l'accompagnement correct du signal qui consiste à faire varier le pas delta (Δ), de telle façon que celui-ci dépende de la manière dont le signal évolue.

Le pseudocode du compresseur par modulation delta adaptative est le suivant :

```
x''(n) = 0
d(n)= x(n) - x''(n)

REPETER
   SI d(n) > 0
          d'(n)= +DELTA
          coder à 1
   SINON
          d'(n)= -DELTA
          coder à 0
```

```
   FSI
   x'(n)= d'(n) + x''(n)
   x''(n+1)= x'(n)
   adapter le pas DELTA
FREPETER
```

Et pour la décompression:

```
REPETER
   SI c(n) = 1
          d'(n)= +DELTA
   SINON
          d'(n)= -DELTA
   FSI
   x(n)= d'(n) + x'(n)
   x'(n+1)= x(n)
   adapter le pas DELTA
FREPETER
```

En général, pour une bonne prestation de la modulation delta adaptative, il faut encore, comme condition essentielle, que la fréquence d'échantillonnage soit plus de 10 fois supérieure à celle de la fréquence de Nyquist.

Pour l'adaptation du delta, il existe un grand nombre d'algorithmes possibles qui tentent de diminuer la limite précédente. Nous présenterons ensuite quelques uns des plus importants pour le codage de la parole[1].

3.4.4.1.1. L'algorithme CFDM de Jayant (Constant Factor Delta Modulation) :

Dans cet algorithme, présenté par Jayant[2] en 1973, le pas d'adaptation obéit à la relation suivante :

$$D(n) = M \times D(n-1)$$

$$D_{min} \leq D(n) \leq D_{max}$$

Etant donné que, dans la modulation delta, c(n) dépend seulement du signal d(n) calculé à partir de la formule suivante, il est possible de faire dépendre M des mots de code actuel et précédent, c(n) et c(n-1).

$$d(n) = x(n) - a \times x''(n-1)$$

De cette façon, le signal d(n) peut être déterminé avant le calcul de la valeur actuelle quantifiée, d'(n), qu'il doit atteindre par la détermination de $\Delta(n)$.

Pour choisir la valeur de M on a adopté les règles suivantes :

$$M = \begin{cases} M_+ > 1 \Leftarrow c(n) = c(n-1) \\ M_- < 1 \Leftarrow c(n) \neq c(n-1) \end{cases}$$

Avec cet algorithme, il est possible de réduire les périodes de saturation de pente, ainsi que de réduire la « grainularité » du codage étant donné que le Δ_{min} peut être plus réduit que le Δ optimal pour la modulation delta linéaire.

[1] Une grande partie du texte a été extrait de [MACHA94]
[2] [JAYAN73]

Pour un signal générique nous avons normalement:

$$1 < M_+ \leq 2$$

$$1/2 \leq M_- < 1$$

$$M_+ \times M_- = 1$$

Cependant, pour le codage de la parole, on a constaté que la manière optimale de coder est de faire:

$$M_+ = M_-^{-1} \approx 1,5$$

Le pseudocode de l'algorithme d'adaptation delta est comme il suit:

```
SI c(n) = c(n-1)
   M= 2/3
SINON
   M= 3/2
FSI
DELTA= DELTA * M
```

Cet algorithme a été aussi inclus dans l'outil Accordion[1].

3.4.4.1.2. L'algorithme CVSDM de Greekles (Continuously Variably Slope Delta Modulation) :

Cet algorithme est aussi connu par sa désignation « Modulation Delta avec Variation de Pente Continue ».

Dans cet algorithme, le pas est adapté en accord avec:

$$\Delta(n) = \begin{cases} M \cdot \Delta(n-1) + D_2 & \Leftarrow \ c(n) = c(n-1) = c(n-2) \\ M \cdot \Delta(n-1) + D_1 & \Leftarrow \ \text{sinon} \end{cases}$$

Dans lequel:

$$0 < M < 1$$

$$D2 >> D1 >> 0$$

Cet algorithme est spécialement conçu pour éliminer rapidement les périodes de saturation de pente, étant donné que, pour une situation dans laquelle on constate trois sauts d'adaptation dans le même sens, il y a un rajout de D2 au pas d'adaptation. Dans les autres cas le pas décroît (car M < 1) jusqu'à atteindre Δ_{min}.

Il est important d'observer que, pour M proche de l'unité, on a une vitesse d'adaptation très lente tandis qu'avec M proche de zéro on obtient une adaptation très rapide.

[1]Voire les sources en C++ dans « adm.cpp »

Cet algorithme est utilisé dans des systèmes qui exigent une grande immunité au bruit mais qui n'ont pas besoin d'un codage de parole de très bonne qualité.

3.4.4.1.3. L'algorithme HIDM de Winkler (High Information Delta Modulation) :

Cet algorithme produit une variation exponentielle du pas d'adaptation tout au long du temps. C'est un algorithme qui convient pour coder des signaux de parole mais il a l'inconvénient de produire fréquemment des dépassements.

La meilleure source pour obtenir des détails additionnels est son premier article de 1963[1].

3.4.4.1.4. L'algorithme 2d-IADM de Tombras (2-Digit Instanteously Adaptive Delta Modulation) :

Cet algorithme a été développé dans la thèse de doctorat de M. Tombras[2] en 1990. Il peut aussi être désigné: « Système de Modulation Delta Instantanément Adaptatif ».

Il s'agit d'un algorithme conceptuellement plus complexe que ceux présentés précédemment, mais avec un rendement bien supérieur. Il réussit à produire des pas d'adaptation du signal à coder bien plus grands et précis, ce qui réduit les dépassements et le temps de réponse aux variations du signal.

Cet algorithme fut spécialement conçu pour coder des signaux de la parole et il est représenté par les relations suivantes:

$$D(n) = M \times D(n-1)$$

$$D_{min} \leq D(n) \leq D_{max}$$

$$M(n) = \begin{cases} N(n) \cdot \beta & \Leftarrow |e(n)| \geq \tfrac{1}{2}(\beta + 1) \cdot |\Delta_{ref}(n)| \\ N(n) / \beta & \Leftarrow |e(n)| \leq \tfrac{1}{2}(1/\beta + 1) \cdot |\Delta_{ref}(n)| \\ N(n) & \Leftarrow \text{sinon} \end{cases}$$

avec :

$$b > 1$$

$$|D_{ref}(n)| = |N(n)| \times |D(n-1)|$$

$$N(n) = \begin{cases} \alpha & \Leftarrow \text{sgn}[e(n)] = \text{sgn}[e(n-1)] \\ 1/\alpha & \Leftarrow \text{sgn}[e(n)] \neq \text{sgn}[e(n-1)] \end{cases}$$

où :

[1] [WINKL63]
[2] [TOMBR90]

$$a > 1$$

$$sgn[D(n)] = sgn[e(n)]$$

D'autres relations existent et permettent de définir les paramètres α et β. Il est cependant important d'ajouter que, dans certains cas particuliers de détermination de α et β, cet algorithme réagit comme l'algorithme de Jayant que nous avons présenté auparavant.

3.4.4.1.5. ADxyM :

J'ai développé un nouvel algorithme avec une compression variable qui peut profiter des silences et des caractéristiques du signal. L'idée fondamentale est qu'en plus d'être adaptative verticalement (Δy) comme les précédents, il est adaptatif horizontalement (Δx). Si le signal change lentement, on augmente le Δx pour réduire le débit binaire (plus de compression). Si le signal est très variable, on prend Δx plus petit pour coder plus d'information.

Il est encore en essai[1] mais si l'on ajuste bien certains paramètres (il y en a beaucoup), on peut obtenir la même qualité que les algorithmes précédents mais avec des codages 4 fois supérieurs.

La caractéristique fondamentale est, qu'à la différence de tous les algorithmes présentés, et encore à présenter, le débit final de compression en bps est indépendant de la fréquence d'échantillonnage originale : il ne dépend seulement que des caractéristiques du signal et de la qualité voulue. Donc, deux signaux identiques échantillonnés avec différentes fréquences donneront, à la fin, le même débit binaire.

Une autre bonne caractéristique de cet algorithme est sa qualité (et inversement le facteur de compression, logiquement) qui peut être sélectionné.

Avec une qualité fixe, évidemment le problème est que la compression obtenue n'est pas connue a priori. Donc, il en résulte une difficulté pour gérer la mémoire. Par contre, avec une compression fixée a priori, la qualité résultante peut beaucoup varier.

Quoique les résultats n'aient été approfondis par manque de temps, l'idée peut être appliquée à la plupart des autres méthodes déjà existantes, avec prédictions linéaires pour l'adaptation horizontale et verticale. Le fruit le plus innovant pourrait être un « ADxyPCM », indépendant de la fréquence originale d'échantillonnage et avec une qualité réglable.

3.4.4.1.6. ADM avec des prédicteurs d'ordre supérieur :

Il est possible d'utiliser des prédicteurs d'ordre supérieur. Le delta est choisi comme l'erreur moyenne de prédiction de chaque segment du signal. Dans ce cas, les formules prennent la forme suivante pour un ordre p:

$$e_k = s_k - \sum_{i=1..p} a_i s'_{k-i}$$

[1]Les sources en C++ sont aussi dans « adm.cpp »

$$d_k = sgn(e_k)$$

$$s'_k = D_n d_k + \sum_{i=1..p} a_i s'_{k-i}$$

Où, s_k est l'entrée et d_k la sortie binaire. La fonction sgn donne 1 si le signe est positif et zéro s'il ne l'est pas.

Ces algorithmes sont plus complexes puisque, pour obtenir un rendement supérieur à ceux du premier ordre, ils exigent que les pôles du système soient réels et, par ailleurs, ils génèrent un problème d'interface entre l'algorithme adaptatif et l'algorithme de prédiction. Mais ce problème peut être résolu si l'on choisit les coefficients a_i du prédicteur de façon qu'ils minimisent la fonction suivante d'erreur sur le block de données précédant :

$$E_{n-1} = \sum_{k=1..L(n-1)} (s'_{k-i} - \sum_{i=1..p} a_i s'_{k-i})^2$$

Dans cette expression, L(n-1) est le nombre d'échantillons du bloc précédent. Puisque les signaux de la parole sont hautement corrélés, on espère que les statistiques du bloc présent ne diffèrent pas beaucoup de celles du bloc précédant. Si l'on utilise les coefficients du bloc préalable, on maintiendra E_n relativement petit pour le bloc actuel. Les coefficients ne sont pas transmis, puisqu' ils peuvent être calculés au décodeur étant donné que tous les s'_k sont disponibles.

Finalement, la valeur delta est calculée comme suit :

$$D_n = (1 / L(n)) \sum_{k=1..L(n)} | s'_{k-i} - \sum_{i=1..p} a_i s'_{k-i} |$$

Les données transmises sont la valeur de D_n et la série de valeurs d_k. Si l'on veut une qualité plus grande, on peut envoyer une deuxième série de valeurs dd_k calculées :

$$dd_k = sgn(s_k - s'_k)$$

$$ss_k = (D_n/2)\, dd_k + s'_k$$

Dans le texte de Yuan[1], un prédicteur du huitième ordre a été essayé (p = 8), avec des coefficients mis à jour toutes les 20 ms. Le signal est échantillonné à 8 khz avec 12 bits de résolution. Le signal est alors (optionnellement) comprimé par l'algorithme TDHS au taux 2:1. La longueur du block du codeur ADM est sélectionnée par 80 échantillons qui représentent 20 ms du signal originel. Pour chaque bloc, 8 bits représentent D_n et 80 bits la série de valeurs d_k. Cela correspond à un débit de 4.4 kbps (sans TDHS, 8.8 kbps). Après le décodeur ADM, le signal est extrait et (le cas écheant) se voit appliqué la décompression TDHS 1:2. Généralement, l'algorithme aboutit à une qualité assez bonne.

3.4.5. L'algorithme ADPCM (Adaptive Differential Pulse Code Modulation) :

La quantification adaptative a été examinée dans l'algorithme ADM, mais n'est pas souvent utilisée toute seule, bien que l'on puisse aussi parler de la méthode APCM (Adaptive Pulse Code Modulation) si la valeur de delta est codée avec un

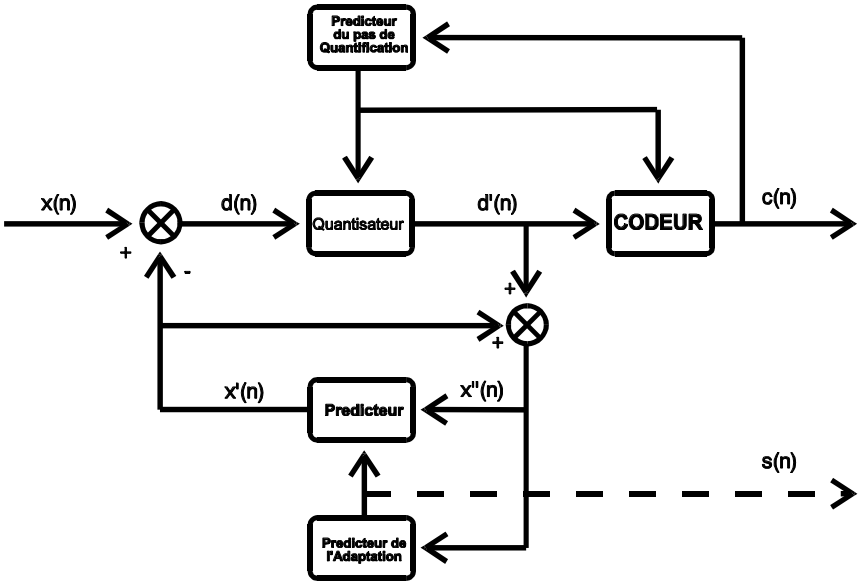
[1] [YUAN_91]

nombre de bits supérieur à 1, dans ce cas. De bons quantificateurs adaptatifs se trouvent dans la bibliographie.[1]

L'algorithme ADPCM inclut la quantification adaptative et également une prédiction adaptative. Il transmet, donc une quantification adaptative de la différence entre la valeur réelle et la prédiction calculée.

Tout cela est fait avec une relative simplicité et avec une qualité « impeccable ». Il y a deux raisons justifiant cette amélioration : l'une est que l'ADPCM couvre une amplitude plus grande que les méthodes précédentes, l'autre est que la distribution de l'erreur de quantification est concentrée vers les fréquences basses. Le résultat est un SNR_{seg} de 22 dB pour 32 kbps (8 khz d'échantillonnage et 4 bits de quantification).

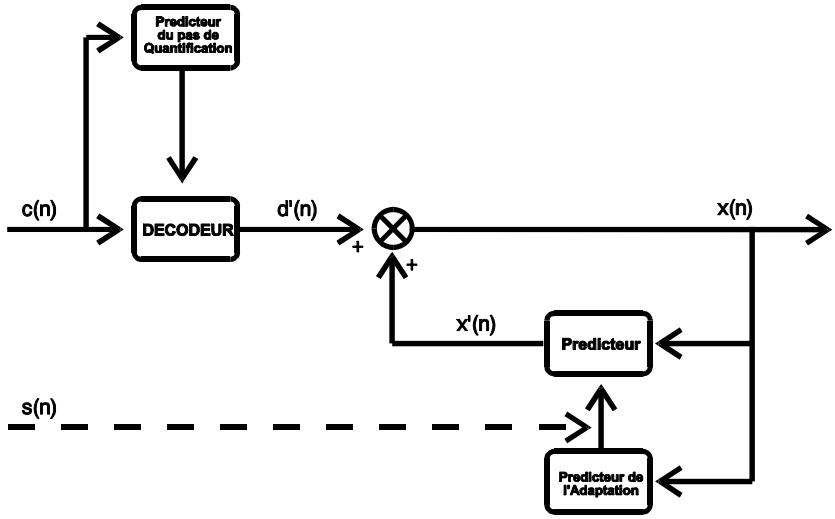
Le schéma suivant montre le codeur d'un système ADPCM :



L'information additionnelle par trame est transmise avec le signal résiduel, comme il est indiqué avec la flèche en pointillée.

Le décodeur est le suivant :

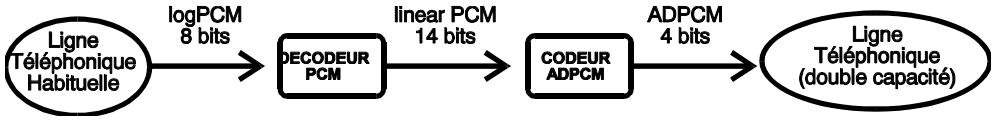
[1] [LIND_90]



L'information additionnelle consiste normalement en un nombre réduit (deux ou trois) de coefficients qui sont déterminées pour chaque bloc.

Il y a deux standards du CCITT pour la téléphonie, le G.721 (32 kbps) et le G.723 (24 ou 40 kbps) et trois autres pour le CD-Levels A, B et C.

Souvent dans les lignes téléphoniques, le schéma précédent est appliqué à une éntrée déjà PCM (m-law ou A-law). Cela donne une complexité accrue par le décodeur 8-bits-log-PCM Þ 14-bits-linear-PCM, le compresseur 14-bits-linear-PCM Þ 4-bits-ADPCM, la transmission et après le décompresseur 4-bits-ADPCM Þ 14-bits-linear-PCM et le codeur 14-bits-linear-PCM Þ 8-bits-log-PCM. Ensuite on montre le compresseur :



Le décompresseur est inverse, évidemment.

En réalité, l'algorithme ADPCM tout seul est assez simple, écrit en arithmétique entière, comme on le verra dans le chapitre de complexité.

Quand le nombre de coefficients augment, l'information additionnelle est plus importante, on parle de AD-DPCM (Additional Differential) ou AP-DPCM (Additional Pitch), car il est très usuel d'inclure aussi un détecteur de pitch, la prédiction étant de cette façon faite à partir des échantillons précédents (pondérés par un nombre p des facteurs α_k) et des échantillons du pitch antérieur (pondérés par un nombre q+1 de facteurs β_k). On a alors, l'équation suivante :

$$x'(n)= \sum_{(1,p)} \alpha_k x''(n-k) + \sum_{(0,q)} \beta_k x''(n-m-k)$$

Où, m est la longueur du pitch. La prédiction est si bonne qu'un bit seul suffit pour coder la différence; par conséquent, on obtient les mêmes débits binaires qu'avec une modulation delta mais avec une qualité plus acceptable.

La réalisation[1] dans l'outil Accordion suit le format DVI de Intel, qui est optimisé par table; le temps de calcul est donc très réduit. La conclusion des essais est semblable à celle de la bibliographie ; c'est l'algorithme le plus utilisé quand on a besoin d'une bonne qualité.

Une mise en place spécifique sur la puce Texas TMS 320C50 est illustrée dans le chapitre 8 de ce rapport, mais des réalisations plus générales peuvent être trouvées dans la bibliographie[2].

[1]Le source est dans le fichier « adpcm.cpp »
[2][REIME89]

3.5. Méthodes dans le domaine fréquentiel :

En ce qui concerne les techniques de codage dans le domaine des fréquences, on peut dire qu'elles sont devenues plus populaires avec l'apparition de processeurs plus puissants. Pour une qualité moyenne, des compressions supérieures sont obtenues par ces algorithmes.

Le concept de base est la division du signal de la parole dans ses composantes fréquentielles par différentes méthodes : avec des filtres, comme on le verra avec le SBC (SubBand Coding) ou avec l'utilisation d'une transformée, comme la Transformée de Fourier Discrète (DFT), la Transformée Discrète du Cosinus (DCT), la Transformée Karhunen-Loève (KLT), la Transformée Walsh-Hadamard (WHT), etc. La transformée à utiliser dépendra de l'existence d'algorithmes rapides et de la sélectivité spectrale.

3.5.1. SBC (SubBand Coding) :

C'est une méthode mixte (temporelle / fréquentielle) introduite par Crochière et al.[1] : Il s'agit d'appliquer une échelle de filtres pour séparer le signal en différentes bandes (ici la partie fréquentielle). Chaque bande est ensuite codée par une méthode appropriée (ici la partie temporale) selon la fréquence (normalement une méthode de codage adaptative prédictive comme ADPCM). Une information plus importante est utilisée pour les bandes les plus importantes pour la parole, méthode connue sous le nom de ABA (Adaptative Bit Allocation). Pour cette raison, les méthodes les plus élaborées sont appelées APC-AB (Adaptative Predictive Coding with Adaptive Bit allocation).

L'amélioration vient de ce qu'avec les mêmes débits, la qualité augmente, donc on peut essayer des compressions plus élevées.

Le choix du banc de filtres (généralement analogiques) est fondamental ; diverses études essayent de minimiser la distorsion interbandes, de fréquence, de phase et de codage. Les articles de Esteban et al. et puis de Crochière utilisent les filtres de miroir de quadrature (QMF : Quadrature Mirror Filters)[2] et des nouvelles techniques[3] pour éliminer ou minimiser toutes les distorsions, en gardant uniquement que la distorsion due au codage. La configuration la plus utilisée est une structure d'arbre binaire de codeur de deux-bandes avec des filtres quadratures[4]. Une structure d'arbre nécessite un stockage plus grand dû au retard, mais ces problèmes ont été bien résolus dans l'article de Barnwell[5] qui utilise 4 kmots de 16 bits pour le stockage mémoire, fourni maintenant par la plupart des DSPs.

Quoiqu'elle soit utilisée pour de fortes compressions aboutissant à un débit d'environ 16 kbps, la norme CCITT G.722 adopte des compressions plus légères avec 64 kbps qui est utilisée couramment pour le vidéoconférences : 48, 56 et 64 kbps sont dédiés à la parole et le reste (16, 8 ou 0) aux informations supplémentaires. L'évaluation expérimentale indique que bien que le SNR_{seg} est semblable à celui de l'ADPCM pour un débit similaire, la qualité subjective étant meilleure. Même avec des compressions plus faibles correspondant à un débit de 128 kbps, par exemple, on peut

[1] [CROCH76], [CROCH77]
[2] [CROIS76], [ESTE77], [BARAB79], [CROCH79]
[3] [JOHNS80], [BELLA76], [SMITH84]
[4] [NAYEB93]
[5] [BARNW82]

arriver à la qualité hi-fi pour le codage de la musique[1]. Ces techniques s'utilisent déjà couramment dans les appareils audio commerciaux.

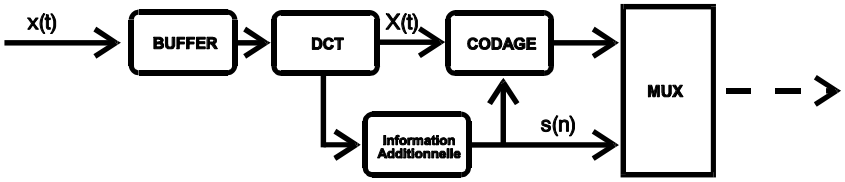
Une mise en place sur le TMS32010 ou sur le NEC 7720 avec 16 kbps aboutit à une qualité semblable à celle de l'ADPCM avec 32 kbps.[2]

Un essai « manuel » a été réalisé dans l'outil Accordion ; on a appliqué quatre filtrages passe-bande, qui donnent quatre signaux, puis on a donné plus de débit de compression (ADPCM) aux bandes plus importantes et, à la fin, on a additionné les signaux résultants. La procédure pourrait être automatisé étant donné que les résultats ont été meilleurs qu'avec l'ADPCM tout seul.

3.5.2. ATC (Adaptative Transform Coding) ou algorithme de Zelinski-Noll :

Il s'agit d'un autre algorithme classique dans le domaine fréquentiel. Les échantillons du signal à l'entrée x(n) sont groupés en ensembles $X_{sR}(n)$ d'environ 20 ms qui peuvent être considérés stationnaires, puis sont traités par une transformée fréquentielle. Les composantes fréquentielles sont alors quantifiées de façon adaptative et sorties du codeur.

Le schéma du codeur est illustré ci-dessous:



Le décodeur a la structure inverse; les groupes d'échantillons sont traités par la transformée inverse donnant $X'_{sR}(n)$. Ces groupes sont ensuite chaînés afin de reconstruire le signal à envoyer à la sortie x'(n).[3]Comme on l'a vu dans ce chapitre, la reconstruction peut utiliser des fenêtres d'interpolation, ou de « overlap », pour éviter qu'un signal ne présente des discontinuités. Si on ajoute un détecteur du pitch et, selon sa période, on fait des coupes et la recomposition est plus facile. Le détecteur du pitch n'a pas besoin d'être trop robuste, un facteur multiplicatif entier n'a pas trop de conséquences lors de la recomposition.

Des ATC gérés par vocodeurs ont été développes par Tribolet and Crochière en 1978 avec des résultats de très haute qualité pour un débit de 16 kbps. Des mises en place ont été effectuées en temps réel[4]. Des résultats avec 9.6 kbps donne un SNR_{seg} inférieur (12.8).[5]

3.5.3. Compression avec d'autres transformées (STC, DTC, WHT) :

[1][SMYTH90]
[2]Une brève exposition dans [BARNW89].
[3]Expliqué dans [ZELIN77] et [TRIBO79]
[4][FJALL85]
[5][VELEV93]

Si on utilise la STC (Sinusoidal Transform Coding)[1], le schéma de codeur est semblable à celui de l'ATC mais la transformée est sinusoïdale. Les résultats sont similaires à ceux obtenues avec des codeurs CELP entre 2.4 kbps et 9.6 kbps.

La même chose est obtenue avec la DCT (Discrete Cosine Transform) ; le schéma est encore identique aux précédents mais la transformée est cosinusoïdale. Les résultats son sensiblement meilleurs que l'ATC avec un SNR_{seg} de 17 dB à 4.8 kbps.[2] Dans l'outil Accordion cette compression a été modifiée à partir d'une source originale conçue initialement pour l'image. Elle fournit un facteur de compression réglable et ne donne pas de mauvais résultats[3].

La description ne s'arrête pas ici. Il y a par exemple une méthode présentée par Spanias et Loizou[4] qui fait une FFT, une iFFT, une WHT (Walsh Hadamard Transform) puis une iWHT. Impressionnant ! Tout ceci pour obtenir 4 kbps avec une qualité moyenne. Logiquement, je n'ai eu pas beaucoup envie de l'implanter.

3.5.4. Contractive Speech Coding :

Même ici, le schéma est semblable aux précédents mais maintenant la différence entre deux transformées de deux tranches consécutives est codée. Quoique que la qualité ne soit pas très bonne (il y a du bruit de compression), les compressions sont impressionnantes (120 : 1) avec 534 bps (0,5 kbps !) de débit binaire[5]. Il utilise des transformations linéaires contractiles et affines et la différence entre les FFTs de deux trames consécutives de parole.

[1] Il est plus commenté dans [CHANG91], [MCAUL87], [MCAUL88], [MCAUL93] et [GRIFF88]
[2] [VELEV93]
[3] dans le fichier « dct.cpp »
[4] [SPANI92]
[5] [ALSAK92]

3.6. Méthodes Paramétriques :

Ces méthodes vont plus loin encore que les méthodes fréquentielles, car elles extraient certains paramètres caractéristiques du signal, les transmettent et, à la fin, le signal est complètement reconstitué.

Les premiers résultats obtenus au commencement du développement de ces techniques ont donné une qualité médiocre, mais actuellement, de nouvelles techniques aboutissent à des qualités bien meilleures.[1]

3.6.1. LPC (Linear Predictive Coding) :

Un codeur LPC (Linear Prediction Coding) propose un modèle simple et analytique du trait vocal, puis il code le modèle le plus proche possible de l'original. Normalement, les composantes de basse fréquence sont transmises et les autres sont reconstitués à partir des paramètres du modèle. Toutes les données sont transmises par trames. Donc, un décodeur LPC utilise ces paramètres pour générer de la parole synthétique qui sera à peu près semblable à l'original.[2] Cette synthése de la parole fait que ces algorithmes ne sont pas utilisables dans des domaines où une bonne qualité est requise car que le résultat est un peu artificiel.

Il existe un standard fédéral américain dénommé FS-1015 à 2.4 kbps qui utilise 10 coefficients. Une mise en place sur les DSP de la famille TEXAS TMS-370 très proche de ce standard est présenté par Holck et Anderson[3] et, sur d'autres puces, il y a d'autres sources bibliographiques[4].

L'extraction des coefficients est réalisée généralement avec les paramètres LSP (Linear Spectrum Pairs)[5]. Selon la source d'excitation pour le prédicteur, l'ensemble peut devenir RELP (Residual-Excited Linear Predictive coding), VELP (Voice-Excited Linear Predictive coding) ou MPC (Multipulse-excited Linear Predictive coding)[67]. Une méthode qui a donné une bonne qualitésest le VRTC (Variable Rate Tree coding) dont l'excitation optimale est cherchée dans une structure d'arbre.[8]

Une version assez simple, et très connue, nommée APC (Adaptative Predictive Coding) utilise une prédiction du spectre et du pitch (AMDF) pour arriver a un débit de 9,6 bps avec une qualité moyenne[910]. Les trames sont d'une durée moyenne de 20 ms ce qui, à 8 khz de fréquence d'échantillonnage, donne environ 160 échantillons par trame. Elles sont codées avec 192 bits dont 35 bits pour les paramètres, ou coefficients, et 157 pour les résidus. Une mise en place pour les TMS320 se trouve dans l'article de Randolph[11].

---

[1] Voir [CROCH76]
[2] Le standard est exposé dans [TREMA82]
[3] [HOLCK89]
[4] [KALTE83], [FELDM83]
[5] Des sources d'un algorithme efficace en C pour leur extraction sont dans [SAOUD92]
[6] [ATAL_82]
[7] [HAYAS94]
[8] Anderson and Bodie, 1975; Fehn and Noll, 1982.
[9] [VISWA80]
[10] [ATAL_82]
[11] [RANDO89]

En résumé, les débits de ces méthodes varient de 2.4 kbps à 9.6 kbps dont à peu près le 20% s'utilisent pour la transmission des paramètres et les qualités sont très variables mais n'arrivent jamais à une qualité suffisante pour nos propos.

3.6.1.1. L'algorithme GSM (Groupe Spéciale Mobile) :

En relation avec les algorithmes LPC, le GSM[1] (Groupe Spéciale Mobile) est un algorithme paneuropéen standardisé (European Cellular Standard) employé par beaucoup de téléphones sans câble. La version GSM 06.10 utilise un codage RPE/LTP (residual pulse excitation / long term prediction), connu aussi sous le nom de RPELP (Regular Pulse Excited Linear Predictive), pour comprimer des trames de 160 échantillons de 13 bits (fréquence d'échantillonnage de 8 kHz, c'est-à-dire une fréquence de trame de 50 Hz) dans 260 bits. Le résultat est un codeur de 13 kbps.

Pour rester compatible avec des applications typiques UNIX, le groupe de recherche en communications et systèmes d'exploitation (KBS) de la Technische Universitat Berlin travaille avec des trames de 160 échantillons de 16 bits dans 33 octets (264 bits), c'est-à-dire un débit de 1650 octets/sec (13200 bps). La qualité de l'algorithme est assez bonne pour la reconnaissance de la parole, et même de la musique.

3.6.1.2. L'algorithme VSELP (Vector-Sum Excited Linear Predictive) :

Il peut être inclus dans la quantification vectorielle, les deux versions existantes sont les contreparties américaine et japonaise du standard GSM. Les résultats son de 8 kbps et 6.7 kbps respectivement.

3.6.2. La Quantification Vectorielle (VQ: Vector Quantization) :

Dans la VQ, on ne représente pas les valeurs individuelles, mais plutôt un sous-ensemble de scalaires (qui composent un vecteur) codé comme une seule valeur. Les ensembles les plus représentatifs constituent les patrons de référence. Cette méthode est utilisée surtout pour traite les images en couleur dont on n'a qu'un nombre réduit de couleurs utilisées dans la même image.[2]

Pour les ensembles de grande longueur, on peut faire la même chose, spécialement lorsque les points consécutifs sont corrélés ; un exemple est la famille des codages CELP (Codebook Excited Linear Prediction) avec la construction d'un « codebook », dont les patrons les plus typiques sont stockés. Lorsqu'on a une série de scalaires, on fait la sélection du patron le plus proche et l'on transmet uniquement le mot de code correspondant à ce patron attribué. Mieux on choisit le codebook pour minimiser les erreurs de distorsion, meilleure sera la qualité. Des débits de 0.8 kbps peuvent être obtenues avec une qualité moyenne, mais on peut atteindre une bonne qualité si l'on augmente le débit.

3.6.2.1. Les algorithmes et normes CELP :

[1] [VARY_23]
[2] Pour un livre sur la VQ, voyez [GERSH91]

La famille des algorithmes CELP (Codebook Excited Linear Prediction)[1] sont l'approche la plus efficace actuellement pour un codage de la parole de haute qualité avec des débits minimaux. La plupart des recherches actuelles dans la compression de la parole se concentrent sur ces sortes des méthodes, étant donné que la puissance des processeurs permet de les réaliser en temps réel, chose inimaginable quelques années plus tôt. Ces algorithmes utilisent ces « codebooks » de sous-ensembles pour quantifier les vecteurs d'excitation pour une prédiction linéaire.

Quelques auteurs utilisent les mots LPC et CELP de façon interchangeable: ils sont en fait très différents. Un codeur LPC (Linear Prediction Coding) propose un modèle simple et analytique du traite vocal: il code alors le modèle le plus proche de l'original. Un décodeur LPC utilise ces paramètres pour générer de la parole synthétique qui sera à peu près semblable à l'originale.

Un codeur CELP fait la même modélisation qu'un LPC mais il calcule les erreurs entre le modèle paramétrique et le signal original. Il transmet alors une codification très comprimée des erreurs (la représentation est un index dans un « codebook » partagé par le codeur et le décodeur). Un codeur CELP fait beaucoup plus de travail qu'un LPC mais le résultat est d'une qualité bien meilleure.[2]

Pour donner un exemple de la codification, le BCELP[3] utilise 25 bits pour des paramètres LPC, 50 et 20 pour les cinq codes et leurs gains respectifs et à la fin 29 et 20 pour les cinq retards LPT et leurs gains respectifs, soit au total : 144 bits par trame.

Il faut choisir une structure de codebook qui optimise en même temps la procédure de construction du codebook et de recherche en son sein.

Pour la génération des codebooks, trois méthodes sont les plus utilisées : l'apprentissage aléatoire fait un choix aléatoire des vecteurs sans plus de complications. A partir de cela, des techniques de réseaux de neurones ont été aussi essayées.[4] De plus, les méthodes basées sur la classification (clustering) utilisent l'algorithme de Lloyd[5] (K-means algorithm), qui organise les vecteurs autour de certains centroïdes qui minimisent la distorsion moyenne selon la formule suivante :

$$r = (\log_2 N) / k$$

Où N est le nombre de patrons du codebook, k la dimension (nombre d'échantillons pour chaque vecteur) et r les bits par échantillon. On a :

$$SNR\ [dB] = [6\ \ (\log_2 N) / k\ ] + C_k$$

Où C_k est un paramètre qui dépend de k. La condition k=1 correspond à la quantification scalaire des méthodes précédentes. Le problème de ces algorithmes est de calculer les vecteurs qui minimisent la distorsion.

La recherche dans le codebook est un problème additionnel. Les techniques du code sont diverses mais le code ternaire[6] est le plus connu pour élaborer et chercher dans le codebook (des techniques avancées pour réduire le temps de recherche ont

[1] [ATAL_85]
[2] [ATAL_85], Atal and Schroeder, 1984, Atal and Rabiner, 1986
[3] [SALAM89]
[4] [VELEV93]
[5] Lloyd, 1957; Max 1960
[6] La réalisation du codebook et du CELP en général est bien expliqué dans [DIFRA92]

été développées[12]). Généralement, des techniques logarithmiques ont été utilisées comme le BTC (Binary Tree Coding)[3]. Cela fait que pour un nombre N de vecteurs, on fait $\log_2 N$ comparaisons.

Tout cela semble très complexe, mais il y a des versions nommées LD-CELP (Low Delay CELP) et LD-VXC (Low Delay Vector eXcitation Coding)[4] [5]et ACELP[6] (Algebraic CELP) donnant des mises en place simplifiées.

Depuis le commencement de la technique on est passé des 16 kbps des CELP initiaux[7] normalisé par le CCITT G.728, aux LD-CELP à 12 kbps[8] ou 8 kbps[910] ou même aujourd'hui aux 4.8 kbps[11]. Les qualités s'améliorent chaque jour[12], mais il y a des problèmes encore avec la qualité pour des voix féminines et d'autres langues pas très courantes[13].

D'autres approches donnent un débit jusqu'à 32 kbps[14], 16 kbps[15] ou 7.2 kbps[16] (quelques uns en combinaison avec SBC) pour coder la parole en bande large 50-7000 Hz (qualité haute de l'audio, vidéophonie, etc.) avec LD-CELP.

Une autre norme est le U.S. DoD's Fédéral Standard-1016 à 4.8 kbps (FIPS-1016 CELP)[171819]. Il a la même qualité qu'un ADPCM à 32 Kbps mais avec seulement 4.8 Kbps. Cette norme est encore améliorée par certaines techniques comme des nouveaux filtres de compensation[20], éliminant totalement les codebooks par « sparse excitation vectors » dans les sigles BCELP[21], des nouveaux détecteurs de pitch[22], etc.

De la même façon qu'avec les LPC, selon la source d'excitation pour le prédicteur, l'ensemble peut devenir RCELP (Residual Codebook Excited Linear Predictive coding)[23]. Dans ce cadre, on arrive à une compression d'un facteur 4 à 6 Kbps avec une qualité supérieure à celle du standard FS-1016. On peut citer aussi le VSELP (Vector Sum Excited Linear Prediction)[24], VXC (Vector Excitation Coding), le LD-HVELP (Low-delay Hybrid Vector Excitation Linear Predictive)[25] ou même le MB-

[1][CHAN_94]
[2][LAW__93]
[3]Gersho and Cuperman, 1983
[4][HUSAI93]
[5][CUPER93]
[6][LAFLA93]
[7][DROGO93]
[8][GRASS93]
[9][CHEN_93]
[10][KATAO93]
[11][QIAN_94]
[12][LEE_89]
[13][HARBO93]
[14][SHOHA93]
[15][FULDS92]
[16][MCELR93]
[17]Une article qui décrit le FIPS-1016 peut être trouvé dans [CAMPB90]
[18]Van Jacobson a plus d'information sur cette norme à <van@ee.lbl.gov>
[19]Il s'obtient par <ftp: (192.31.192.1) //super.org/pub/celp_3.2a.tar.Z> ou <ftp: //svr-ftp.eng.cam.ac.uk/comp.speech/sources/celp_3.2a.tar.Z>
[20][BOITE92]
[21][SALAM89]
[22][CHEN_93b]
[23]Voire [KLEIJ94]
[24]commenté dans [GERSO90].
[25][CHEN_93]

LPC (MultiBand Linear Predictor Coder)[1] qui combine MBE[2] (MultiBand Excitation) vocoders et CELP avec des qualités moyennes. La même combinaison est aussi appelée MBCELP (MultiBand CELP) par d'autres auteurs[3] mais avec 4.8 kbps, on arrive à une qualité très bonne. Même des méthodes à 3 kbps sont en essais mais avec de grandes perspectives[4][5].

Il y a aussi des mises en place sur des DSPs[6]; par exemple le DSP avec virgule flottante TMS32030 à 40 Mhz[7]. Malheureusement, je n'ai pas eu physiquement le temps de mettre en fonctionnement une version dans mon outil Accordion, étant donné la complexité inhérente à ces méthodes et le fait que, généralement, la bibliographie est trop cryptée en la matière (les articles se basent sur des articles précédents, et ceux-ci font la même chose jusqu'à presque l'infini).

3.6.2.2. Les normes JPEG :

JPEG (Joint Photographic Experts Group) est le nom original du comité qui a écrit le standard. Il travaille avec des images statiques et en couleur ou en échelle de gris. Il s'agit d'une technique de compression avec perte qui exploite les limitations de l'oeil humain. bien évidemment, il ne s'agit pas d'une méthode pour le son mais on peut en extraire quelques techniques intéressantes.[8][9]

3.6.2.3. Les normes MPEG et les algorithmes ASPEC :

MPEG est un « Moving Picture Experts Group » qui travaille sous la direction de l'ISO (International Standards Organization) et l'IEC (International Electro-Technical Commission). Ce groupe travaille sur les standards pour le codage des images en mouvement et de l'audio qui lui est associé.

Il a quatre couches, de la moins comprimée à la plus comprimée : MPEG-1, MPEG-2, MPEG-3 et MPEG-4. La troisième a été actuellement incorporée dans la deuxième. Un décodeur de la couche N peut décoder toutes les couches précédentes (pour les codeurs, ce n'est pas la même chose).

De toute façon, c'est la partie audio de la norme qui nous intéresse. Toutes les couches peuvent utiliser 32, 44.1 ou 48 khz de fréquence d'échantillonnage et le débit va de 32 kbps à 320 kbps. Il s'agit de méthodes pour l'audio en général qui ne sont pas appropriées pour la parole uniquement, car la qualité est très bonne mais le débit est excessif (environ 128 kbps[10][11]).

---

[1] [YELDE91]
[2] [YANG_93b]
[3] [YANG_93a]
[4] [KWON_93]
[5] [YANG_93]
[6] [FULDS92]
[7] [LAFLA93]
[8] Il y a un FAQ à <news.answers>.
[9] Une bonne implémentation peut être trouvée dans <ftp.uu.net:/graphics/jpeg/jpegsrc.v5.tar.Z> ou <nic.funet.fi:/pub/graphics/packages/jpeg/jpegsrc.v5.tar.Z>.
[10] Des codeur / decodeurs shareware peuvent s'obtenir via Internet <msn@iis.fhg.de>
[11] [CHAN_93]

Le standard de compression d'audio de la couche 3 du MPEG contient la majeure partie des algorithmes ASPEC et MUSICAM[1]. En regard au premier, c'est un des algorithmes de grande qualité pour la compression de son inclus généralement dans la partie audio de la norme MPEG, mais, quelques fois, on l'a trouvé dans d'autres systèmes. L'ASPEC et a été développé par une équipe du Frauenhofer Institut à Erlangen[2] (Allemagne) et par des autres.

L'ASPEC produit de la qualité CD à 128 kbps. C'est un algorithme avec perte qui rejette les fréquences qui ne sont pas détecées par l'oreille humaine, puis un codage entropique (réversible sans perte) très sophistiqué est réalisé. Une variante à 64 kbps peut apporter une qualité hi-fi aux lignes de téléphone ISDN conventionnelles. Elle a été implantée sur des DSPs standard.

3.6.3. Codage Sinusoïdal :

Décrit dans l'article de Marques et Abrantes[3] sous le nom de « Hybrid Harmonic Coding » avec 4.8 kbps, il est de bonne qualité. Il s'agit de coder le signal comme une addition de sinusoïdes et de signaux aléatoires de bande pour reconstruire le signal original sans nécessité d'un détecteur de voix. Tout cela donne des résultats plus robustes et plus indépendants du pitch du locuteur.

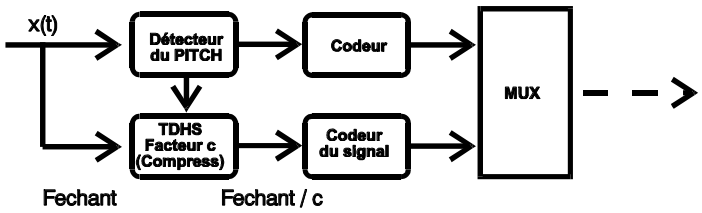
[1]Une réference est [BRAND92].
[2]Il est possible de contacter avec Markus Kuhn à <mskuhn@immd4.informatik.uni-erlnagen.de>.
[3][MARQU92]

3.7. Autres techniques de compression :

Ils existent d'autres techniques pour améliorer le fonctionnement général de la plupart des algorithmes que l'on vient de présenter.

Si, dans des algorithmes LPC la prédiction est faite de façon à minimiser le SNR et que la recherche de corrélations est faite plus exhaustivement, on appelle cela Multipath Search Coding[1]: les résultat sont les méthodes de Tree Search Coding[2][3], Trellis coding [4] ou, comme on l'a vu, la quantification vectorielle. Une méthode très utilisée conjointement avec le VRTC (Variable Rate Tree Coding) est l'équalisation de phase (PH) qui donne des résultats spectaculaires[5]. D'autres évolutions ont été étudiées comme la quantification matricielle[6]. Des techniques de programmation dynamique et par chaînes de Markov sont couramment utilisées, ce qui rend plus compliquées les structures de prédiction. En contraste, les méthodes basées sur les valeurs présentes ou immédiatement passées comme la modulation delta ou le DPCM sont référencées comme Single-path Search Coding.

La technique TDHS (Time-Domain Harmonic Scaling), qui sera examinée au point de la vitesse variable, a été utilisée en combinaison avec la plupart des méthodes précédentes. Dans le paragraphe sur l'ADM, on a déjà parlé de la combinaison TDHS et ADM[7], appelé ADM/HS. On a aussi le SBC/HS (une combinaison de SubBand Coding et TDHS) et le ATC/HS (une combinaison de Adaptative Transform Coding et TDHS). La figure suivante montre la combinaison de TDHS avec une autre méthode quelconque de compression :



La robustesse du détecteur est très importante. C'est pour cela que des algorithmes robustes de détection du pitch[8] ont été utilisés.

Une façon de donner du naturel au son généré par un vocodeur ou quelqu'autre algorithme est la méthode de Roberts[9] qui a été utilisée fondamentalement pour la compression d'images. Roberts a proposé d'additionner un bruit pseudo-aléatoire aux échantillons avant la compression. Ce bruit est uniforme et possède une gamme dynamique égale à l'intervalle élémentaire de quantification. Il est en plus statistiquement indépendant des échantillons. Un bruit identique est soustrait à la reconstitution. Le résultat final est que les échantillons décomprimés ne font apparaître aucun effet artificiel.

---

[1] [FEHN_82]
[2] Anderson and Bodie, 1975
[3] Schroeder and Atal, 1982
[4] Stewart et al., 1982
[5] Moriya and Honda, 1986
[6] Wong et al., 1983
[7] Consideré dans [YUAN_91].
[8] Comme celui presenté dans [CHEN_88]
[9] [ROBER62]

Il y a d'autres méthodes plus élaborées. Par exemple, quoique qu'elle soit surtout utilisée pour les images, celle de la compression fractale arrive à des compressions impressionnantes (20:1...60:1), mais il y a besoin de beaucoup d'heures de calcul pour obtenir une fonction fractale qui génère des données semblables au signal original. L'avantage est que, une fois trouvée cette fonction, la décompression est immédiate. Donc, en temps réel, la compression ne peut pas être utilisée mais seulement la décompression. Il y a des applications dont une ample distribution (CD-ROM par exemple) peut rendre rentable cette procédure complexe de compression.[1]

[1]Pour des informations étendues, regarder [BARNS] ou [JACQU90].

3.8. Conclusions :

Il y a encore beaucoup d'autres techniques, en dehors de celles que nous venons de présenter, qui s'intègrent dans le domaine de la Modulation Delta et aussi d'autres algorithmes qui utilisent des techniques assez diverses, notamment dans le domaine fréquentiel ou paramétrique.

3.8.1. Comparaison des méthodes :

Les systèmes ADPCM, APC et SBC sont à peu près 10 fois plus complexes que les systèmes PCM, DM et DPCM ; les systèmes APC-AB et ATC sont 10 fois plus complexes que ces derniers et les systèmes LPC, VQ et CELP sont encore 10 fois plus complexes. C'est-à-dire, qu'à mesure que les algorithmes deviennent plus complexes, ils ont besoin d'un matériel plus puissant pour la procédure de codage. Des algorithmes trop complexes introduisent souvent des retards (d'environ 100 ms jusqu'à environ une demi seconde) pour calculer les tables et certains paramètres. Quoiqu ce ne soit pas un problème pour l'objectif de ce travail, cela rend impossible ces solutions pour des applications de téléphonie.

À cet égard, et comme point de référence final, le tableau suivant fait un résumé des méthodes les plus intéressantes parmi celles examinées dans ce chapitre. Les données montrées dans ce tableau ne sont pas trop exactes mais servent comme résumé pour éclaircir tout l'embrouillement de sigles.

Algorithme de Codage	Débit (Kbps)	Complexité (MIPS)	Retard de Codage (ms.)	Qualité	MOS	SNRq
Linear-PCM	> 80		0	Haute Hi-Fi	5	
log-PCM	64	0.01	0	Haute	> 4	
DPCM	40-80	0.01	0	Moyenne		
DM	50-80	0.01	0	Moyenne		
ADM/CVSD	16-32	0.05	0	Moyenne	> 3	
ADPCM	16-32	0.1	0	Haute	(32 kbps) 3.88	(32 kbps) 24.9
SBC	9.6 - 24	0.3	0	Bonne	> 3	(16 kbps) 11.1
APC-AB ou ASBC (Adaptative SubBand Coding)	16	1	25	Haute	> 3	
ATC (Adaptative Transform Coding)	4.8-9.6	2	30	Moyenne		< 12.8
Contractive Speech Coding	0.5	10	20	Faible		
DCT (Discrete Cosine Transform) Coding	4.8	10		Moyenne		17
LPC (vocoders)	2.4 - 12	5	35	Faible	< 2	
MPC (Multipulse LPC)	8	20	35	Moyenne	> 2	14
LD-CELP	16	50	20	Haute	3.93	25.34
CELP	4-8	100	35	Moyenne	3	11
xCELP (Modernes)	4-8	200		Haute	4.5	25 4.5

Dans ce tableau, on montre pour chaque algorithme, son débit original ou de référence, sa complexité de compression en MIPS (Million d'Instructions Par Seconde) que l'on reverra dans le chapitre 6, et de différentes mesures de qualité (que l'on examinera dans le chapitre 5)[1].

3.8.2. Applications :

Pour nos propos d'enregistrement de qualité de quelques secondes, comme on peut l'observer, seules deux approches ou familles d'algorithmes méritent d'être profondément considérées : les algorithmes ADPCM et CELP dans toutes leur variétés.

La version standardisée du premier est réalisé dans l'outil Accordion[2] et mise en place sur DSP. Le premier est assez simple et a été en partie obtenu de l'Internet (d'une source en C) ou de la bibliographie. Des variations comme le SBC combiné avec ADPCM pourraient donner des bons résultats.

Pour d'autres applications comme un enregistrement de quelques minutes, il faudrait plusieurs Mega-octets pour l'enregistrement. Donc, d'autres algorithmes doivent être étudiés. Au niveau du signal, la DM, et surtout la ADM, donnent des

[1]Les sources de ce tableau sont diverses, fondamentalement, pour les mesures de qualité.
[2]dans <adpcm.cpp>

résultats acceptables avec des codages de 8 kbps, approximativement. Cela représente 1 Kilo-octet par second, c'est-à-dire, 60 Ko par minute. Il peut être appliqué à l'enregistrement des lettres, par exemple. Avec mon algorithme ADxyM (il est encore à l'essai), si on ajuste bien certains paramètres, on peut obtenir avec la même qualité des codages 4 fois supérieurs. Il représente 15 Ko par minute. Ainsi, on peut enregistrer un quart d'heure dans une puce de 256 Kilo-octets par exemple. Comme on l'a dit, une autre bonne caractéristique de cet algorithme est sa qualité (et inversement le facteur de compression, logiquement) qui peut être sélectionné.

Malheureusement, il n'ai pas eu le temps d'essayer une version définitive de l'algorithme précédent ou de quelque variété de CELP, même simplifiée, étant donné que la codification de l'algorithme peut être ardue et que je n'ai pas trouvé de source déjà développé. La plupart des recherches actuelles dans la compression de la parole se centrent sur cette sorte de méthodes, et des standards en C ou C++ seront disponibles bien sûr à court-terme. C'est la même chose qui peut arriver avec les sources que je vais présenter ensuite : la vitesse variable...

Chapitre 4

VITESSE VARIABLE

La variation de vitesse d'élocution consiste en la restitution d'un message sonore à une vitesse plus lente ou plus rapide. Des études sur la vitesse variable se reduissent à de simples ébauches de ce qui, quelquefois, a été appelé dans la littérature anglaise « Speech Cooling » ou « Variable Rate ». Ce chapitre prétend donner un examen complet et innovateur sur ce sujet.

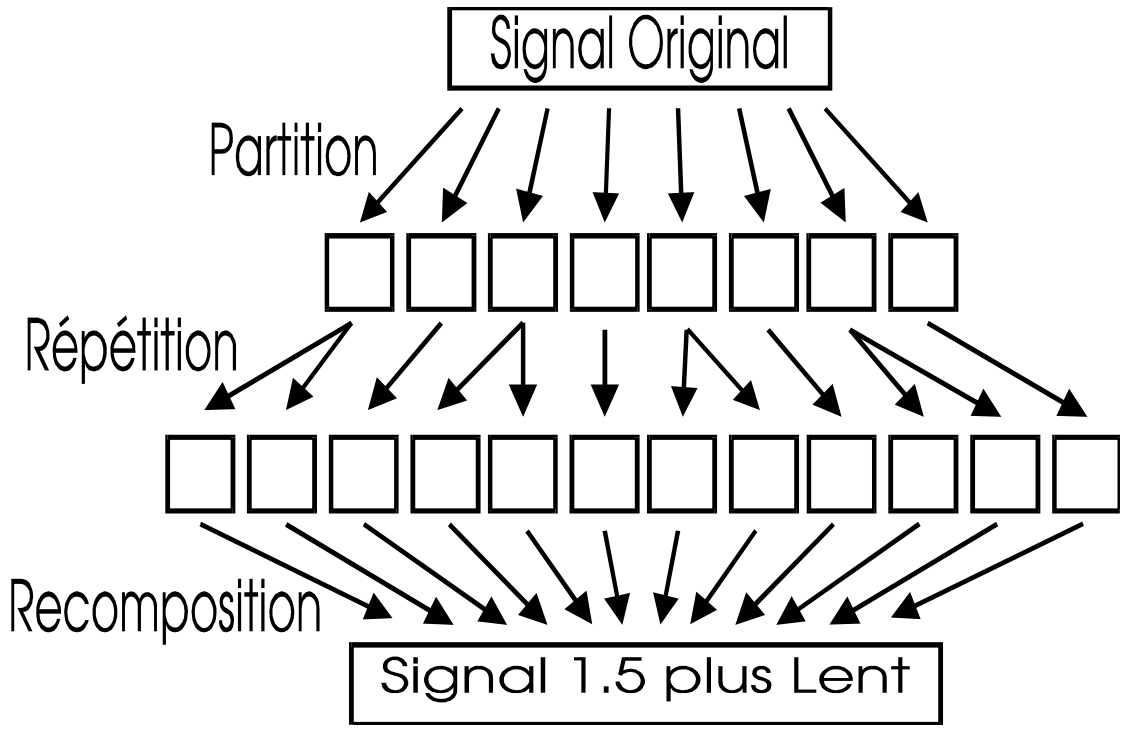
4.1. Introduction :

Cette variation de vitesse d'élocution présente beaucoup de problèmes si elle est faite d'une façon linaire. À la manière d'un lecteur de cassette (variation de la vitesse de défilement de la bande), la fréquence des formants est changée, (donc sortie des échantillons à une fréquence différente de la fréquence d'échantillonnage), en produisant l'effet d'un changement du timbre du parleur. C'est-à-dire, à fréquences douces, il parait avoir une voix grave et à fréquences hautes, il semble avoir la voix d'un bébé.

Pour nos propos, cette fonctionnalité nécessite l'implantation d'un algorithme de transposition de la parole, qui conservera le timbre et l'intonation de la voix.

Sur les signaux numériques, cette méthode est obtenue en faisant une partition échantillon par échantillon, et en les dupliquant ou les éliminant le nombre de fois qu'il est nécessaire pour obtenir la variation de vitesse requise. Quoique celle-ci soit la méthode la plus simple et donc la plus facile à implanter comme algorithme, on peut percevoir trois phases clairement différenciées : la partition, la répétition/suppression et la recomposition.

Mais les algorithmes que l'on va présenter changent le schéma pour traiter avec des unités plus grandes : des tranches au lieu des seuls échantillons :



Étudions alors, ces trois procédures séparément :

4.2. La procédure de partition :

Abstraction faite de la partition par échantillons, la première solution est de couper la suite en petits intervalles, morceaux ou tranches proches de la fréquence de vibration des cordes vocales, appelée *fréquence du fondamental* ou taille du *pitch*. La fréquence du fondamental et son pitch correspondant sont comme on l'a vu[1] au chapitre 2 :

- De 80 à 200 Hz (5 ms. à 12.5 ms.) pour une voix masculine.
- De 150 à 450 Hz (2.22 ms. à 6.66 ms.) pour une voix féminine.
- De 200 à 600 Hz (1.66 ms. à 5 ms.) pour une voix d'enfant.

Diverses études ont montré qu'une taille moyenne idéale est d'environ 9 ms. Celle-ci est suffisamment grande pour ne pas affecter la composition des formants, de l'ordre de 500 Hz. (2 ms.) à 3000 Hz. (0.33 ms.) et en même temps assez petite pour ne produire qu'un effet d'écho minimal (au moment de la répétition dans la procédure de partition). Ces données ne sont pas immuables, par exemple les sons non voisés ne présentent pas de structure de pitch, la parole chuchotée est complètement différente, le bruit de fond aussi, etc.

Si la taille des tranches est fixée constante, la partition est appelée **« constante »** et le point de la coupe ou liaison est déterminée d'une façon fixe. Inévitablement, lors de la duplication ou de l'élimination de quelques tranches, cela produit un bruit de fond indésirable conséquence directe de :

[1] selon [BOITE87]

- L'apparition d'une fréquence autour de la fréquence de partition,
- Les liaisons brusques entre les tranches.

Le premier problème peut être diminué considérablement avec une partition **« aléatoire »** des tranches avec la moyenne du *pitch* et une variation limitée à l'intervalle (1 - 20 ms.). En fonction de la distribution choisie, on obtient des résultats meilleurs ou pires: **« aléatoire uniforme »**, **« aléatoire normal »**, **« aléatoire exponentiel »**, etc. Le choix de l'une d'entre elles détermine la qualité et la complexité de la réalisation ultérieure.

Le deuxième problème (liaison discontinue) peut être le plus important pour une bonne partition. Donc dans la vitesse variable, la prédiction du pitch n'est pas aussi importante que dans autres applications. Ici, l'absence de bruit dû aux discontinuités est la prémisse fondamentale. On a étudié des différents facteurs soit ensemble soit séparément pour observer leur influence. Ces facteurs sont:

- Taille de la tranche,
- Amplitude de la liaison au point de coupe,
- Pente de la liaison au point de coupe,
- Fréquence instantanée au point de coupe,
- Puissance instantanée au point de coupe.

Ensuite, nous décrirons la signification concrète de chacun d'entre eux mais leur pondération correcte donnera les liaisons nettes que nous désirons.

4.2.1. Taille de la tranche :

Comme on l'a dit, il est fondamental de choisir une bonne taille de tranche. Pour maintenir des valeurs raisonnables, la qualité de taille est calculée de la manière suivante :

$$qTail = 1.0 - | Taille\_Ref - Taille | / MaxError$$

Avec :

$$MaxError = TAILLEMAXPITCH - TAILLEMINPITCH$$

Ceci nous donne toujours une valeur entre 0 (taille mauvaise) et 1 (taille excellente). Par exemple, si nous avons une taille de référence de 10 ms., une taille de maximum 35 ms. et de minimum 3 ms., une tranche de 14 ms. donnera:

$$1.0 - | 10 - 14 | / (35 - 3) = 0.875 \qquad // \text{Une bonne taille !}$$

La taille de référence peut aussi être une valeur fixée initialement, la taille de la tranche antérieure, la moyenne de toutes les tranches antérieures ou une pondération de toutes ces valeurs. L'implantation présentée utilise :

$$Taille\_Ref = Pond\ (TAILLEPITCH, Taille\_Ant, Taille\_Ref)$$

C'est-à-dire, un peu de tout. Si nous donnons préférence à la Taille_Ref, nous avons une référence adaptative; son avantage est que l'apparition de paroles pas très usuelles (par exemple un enfant) peut être abordée sans problèmes.

4.2.2. Amplitude de la liaison :

Elle peut être le facteur le plus important. Le saut entre une valeur haute à la fin d'une tranche et une valeur basse au début de la tranche suivante (et vice versa) pourrait ajouter beaucoup de bruit au signal résultant. Étant donné que l'amplitude est représentée par un numéro réel entre -1.0 et 1.0, la qualité d'une coupe relative à l'amplitude est :

$$qAmpl = 1.0 - (| Ampl\_Ref - Ampl | / 2.0)$$

La valeur 2.0 n'est que la longueur de l'intervalle [-1..1]. Il en résulte un facteur entre 0 (coupe mauvaise) et 1 (coupe bonne).

De la même façon qu'avant, l'amplitude de référence peut être: une valeur fixée initialement (généralement zéro), l'amplitude du dernier point de la tranche antérieure, la moyenne de tous les derniers points des tranches antérieurs ou une pondération de toutes ces valeurs. L'implantation présentée utilise:

Ampl_Ref= Pond (0.0, Ampl_Ant, Ampl_Ref)

Mais cette fois, nous donnerons préférence au deuxième facteur pour obtenir une bonne liaison et au premier pour une action corrective vers le zéro parce qu'il s'est montré un bon compromis pour le point de coupe.

4.2.3. Pente de la liaison :

La pente P à un point x2 est calculée :

$$P = (x2 - x1) / dx$$

Avec :

x1: Le point antérieur (échantillon précédent).
dx: Le différentiel (temps en ms. entre deux échantillons).

L'insertion du différentiel rend la valeur résultante indépendante de la fréquence d'échantillonnage. Mais, avec cette formule, la valeur de la pente varie entre $-\infty$ et ∞. Pour obtenir un facteur plus maniable, nous faisons :

$$Pend = atan [(x2 - x1) / dx] \qquad // Entre -p/2 et p/2$$

La qualité de la pente est alors :

$$qPend = 1.0 - | Pend\_Ref - Pend | / p$$

La constante π n'est que la longueur de l'intervalle $[-\pi/2 .. \pi/2]$. Il en résulte un facteur entre 0 (coupe mauvaise) et 1 (coupe bonne).

La pente de référence peut aussi être une valeur fixée initialement, la pente du dernier point de la tranche antérieure, ou la moyenne de tous les derniers points des tranches antérieures ou une pondération de toutes ces valeurs. L'implantation présentée utilise :

$$Pend\_Ref = Pond(p/2, Pend\_Ant, Pend\_Ref)$$

On donnera préférence au deuxième facteur pour obtenir une bonne liaison. Le reste n'est pas très important parce qu'il n'y a pas besoin d'une action corrective ou adaptative. J'ai constaté que les valeurs $\pi/2$ ou $-\pi/2$ (pentes verticales) sont légèrement meilleures que d'autres comme 0 (pente horizontale), mais celle-ci est plus facile à implanter (x2 = x1).

4.2.4. Fréquence immédiate :

Ce facteur n'est pas excessivement important. Pour le calculer, d'autres implantations comptent les passages par zéro. Pour éviter que cela ne marche pas lorsqu'il y a un niveau continu différent de zéro, on ne fait qu'un comptage des maxima.

La détection d'un maximum est fait comme suit :

```
if ((x1 > x0) and (x1 > x2))
   CestMax= 1;
else
   CestMax= 0;
```

Donné que x2 est la valeur actuelle, x1 la dernière et x0 l'avant-dernière. La fréquence immédiate peut donc être calculé comme une pondération :

$$FreqI = \alpha * FreqI + (1 - \alpha) * CestMax / dx$$

dx: Le différentiel (temps en ms. entre deux échantillons).
α: Facteur de mémoire (une valeur de 0.9 s'est montrée adéquate).

Comme avant, l'insertion du différentiel rend le résultat indépendant de la fréquence d'échantillonnage. Si l'échantillonnage est à basse fréquence, quelques maximums peuvent être perdus, entraînant une fréquence plus petite.

Les unités de FreqI sont (1/ms.) et théoriquement FreqI peut varier entre 0 et ∞. En plus, il y aura des signaux d'une fréquence très grande et des autres d'une fréquence petite. Pour obtenir un facteur plus maniable et plus indépendant, nous faisons :

$$FreqI' = 1.0 - e^{-FreqI / Freq\_Moyenne}$$

Si nous connaissons la fréquence moyenne ou si nous la calculons à mesure que nous lisons la suite (d'une façon décrite plus loin). Tout ceci donne une valeur variant de 0 (basse fréquence) à 1 (haute fréquence). Maintenant, nous pouvons exprimer le facteur de qualité de la coupe comme :

$$qFreqI = 1.0 - | FreqI\_Ref - FreqI' |$$

Il en résulte encore une autre fois un facteur compris entre 0 (coupe mauvaise) et 1 (coupe bonne). À l'égal des cas antérieurs, la fréquence immédiate de référence peut aussi être une valeur fixée initialement, la fréquence immédiate du dernier point de la tranche antérieure, ou la moyenne de tous les derniers points des tranches antérieurs ou une pondération de toutes ces valeurs. L'implantation présentée utilise :

$$FreqI\_Ref = Pond(1.0, FreqI\_Ant, FreqI\_Ref)$$

Nous donnerons préférence au premier facteur pour obtenir une bonne liaison. Il est sûr qu'il existe une étroite relation avec la pente, parce que si la pente est verticale, la fréquence ne peut pas être nulle. Ceci pourrait rendre superflu ce facteur. Mais on n'a pas encore étudié profondément la convenance des coupes aux positions des fréquences basses ou hautes.

4.2.5. Puissance immédiate :

Théoriquement, si la liaison est faite en un point où la puissance est faible, l'apparition de bruit sera plus petite. La manière de calculer la puissance immédiate est pareille à celle de la fréquence.

$$PuisI += \alpha * PuisI + (1 - \alpha) * Sqr(x2) / dx$$

x2: la valeur de l'amplitude,
dx: Le différentiel (temps en ms. entre deux échantillons),
α: Facteur de mémoire (normalement 0.9),
Sqr: Racine carrée.

Comme avant, la division par le différentiel donne un résultat indépendant de la fréquence d'échantillonnage. Cela donne une valeur de 0 (basse puissance) à 1 (haute puissance), mais si le signal est enregistré avec un volume faible, cette valeur peut toujours être très petite. Nous pouvons augmenter la valeur si nous connaissons la puissance moyenne du signal:

$$PuisI' = 1 - e^{-PuisI / Puis\_Moyenne}$$

Maintenant, nous pouvons exprimer le facteur de qualité de la coupe comme:

$$qPuisI = 1.0 - | PuisI\_Ref - PuisI' |$$

Il en résulte comme toujours un facteur entre 0 (coupe mauvaise) et 1 (coupe bonne). À l'égal des cas antérieurs, la puissance immédiate de référence peut aussi être une valeur fixée initialement, la puissance immédiate du dernier point de la tranche antérieure, ou la moyenne de tous les derniers points des tranches antérieures ou une pondération de toutes ces valeurs. L'implantation présentée utilise :

$$PuisI\_Ref = Pond(0.0, PuisI\_Ant, PuisI\_Ref)$$

Nous donnerons préférence au premier facteur pour obtenir une bonne liaison parce-que cette fois il est clair qu'une coupe à puissance faible est meilleure qu'une coupe à grande puissance. Aussi, dans ce cas, il y a une étroite relation avec l'amplitude de la liaison puisque si la liaison est au passage par zéro, la puissance ne peut pas être trop grande.

4.2.6. Pondération des facteurs:

La qualité finale d'une coupe sera donnée par une pondération de tous les facteurs antérieurs:

$$Qual = Pond\ (qTail, qAmpl, qPend, qFreqI, qPuisI)$$

Une bonne mais inefficace implantation pourrait être:

$$Qual = qTail^{Fact\_Tail} * qAmpl^{Fact\_Ampl} * qPend^{Fact\_Pend} * qFreqI^{Fact\_FreqI} * qPuisI^{Fact\_PuisI}$$

Le point qui donne la valeur de qualité la plus grande sera choisi pour la coupe de la tranche. Initialement la position du pitch est choisie de bonne qualité, bien qu'aucun point ne soit calculé.

Les pondérations géométriques ont l'avantage de ne pas ignorer les facteurs relativement mauvais (car elles font des pondérations arithmétiques) mais leur implantation finale est très délicate en raison de la complexité des calculs. Avec une pondération arithmétique bien ajustée, on peut obtenir des résultats identiques.

4.3. La procédure de Répétition / Suppression :

Une fois que l'on a le signal original bien fendu, la question est de le recomposer à une vitesse différente. Le taux de variation sera représenté par une valeur réelle *d* supérieure que 1.0 pour les accélérations et de 0.0 à 1.0 pour les expansions. Par exemple: 2.0 pour deux fois plus vite, 0.25 pour quatre fois plus lent.

Pour calculer le nombre de fois qu'il faut répéter une tranche, on peut faire simplement:

$$Rep = Arrondi\ (1 / d)$$

Cette méthode est appelée **constante**, et ne fonctionne pas mal pour les expansions entières, c'est-à-dire: d= 0.5, d= 0.333, d= 0.25, etc., lesquelles donnent Rep= 2, Rep= 3, Rep= 4, etc., respectivement. Le problème apparaît lors qu'on a un numéro de répétitions non-entier. Par exemple, d= 0.7 donnerait une valeur de 1.42 qui, au moment de l'arrondi, donnerait Rep= 1, lequel ne produirait pas de variation. Le problème sera encore pire lorsque l'on traite des accélérations, le résultat étant toujours 0.

Il y a deux manières de résoudre le problème de l'arrondi:

- Méthode **linaire uniforme** : En pondérant l'erreur.

```
Err= 0;

Répéter pour toute tranche:
   Rep= Arrondir (1/d + Err)
   Err= Err + 1/d - Rep
Fin
```

- Méthode **linaire aléatoire** : En utilisant des nombres aléatoires. Étant donnée la fonction rrand() qui donne un nombre aléatoire entre 0 et 1, on calcule les répétitions de la façon suivante:

```
if ( rrand() < (1/d - Tronquer (1/d) )
   Rep= Tronquer(1/d) + 1
else
   Rep= Tronquer(1/d)
```

Il est important d'initialiser le générateur de nombres aléatoires au commencement pour assurer que l'algorithme se comporte chaque fois également. Normalement, il y a une fonction rand(n) avec laquelle on peut initialiser le générateur.

La première méthode peut présenter une cadence à la séquence de répétitions; par exemple, d= 3 produirait 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, ..., lequel peut faire apparaître des fréquences indésirables comme on l'a dit quand on parlait de la taille de la tranche. La deuxième est donc plus recommandable et son implantation n'est pas difficile (une qualité importante du générateur de nombres aléatoires n'est pas requise).

4.4. La procédure de Recompositon :

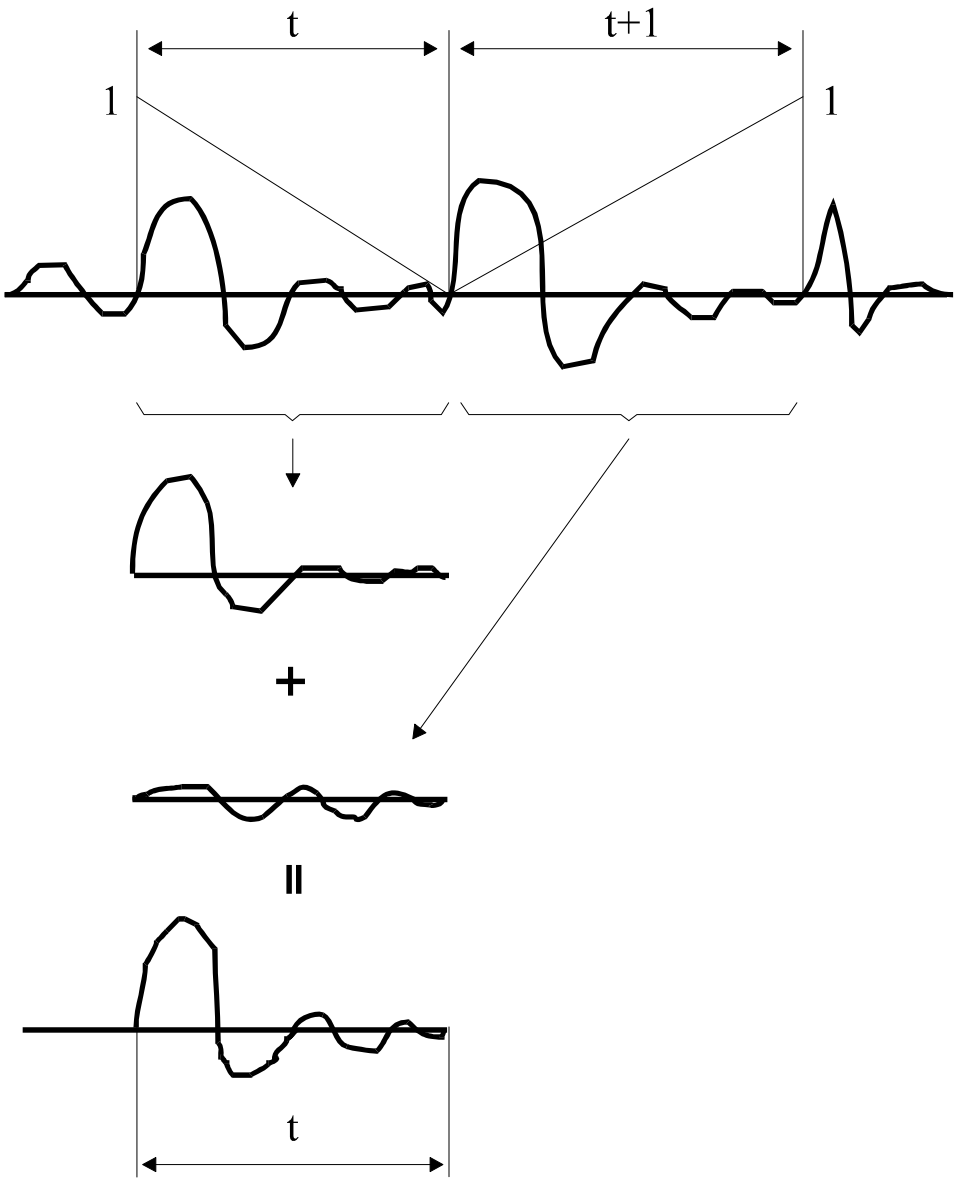
Si l'on a fait une partition uniforme ou sans prendre en compte la liaison suivante, l'effet résultant sera un bruit de fond dû aux discontinuités par les coupes. On explique ici une façon de faire un résultat final complètement continu, utilisant des fenêtres pour amoindrir les passages entre tranches.

4.4.1. La méthode TDHS :

La méthode TDHS (Time Dynamic Harmonic Scaling) est un mode de contraction ou d'extension de la structure harmonique du signal.[1] Avec cette méthode, les tranches sont mélangées après être multipliées par une fenêtre. Cette fenêtre est choisie pour produire un signal résultant sans discontinuités.

Pour une compression de ½, deux tranches consécutives (t et t+1) sont multipliées par une demi-fenêtre (décroissante pour la première, croissante pour la deuxième) puis sont additionnées. Cela assure l'inexistence de discontinuités, comme on peut le voir sur la figure suivante :

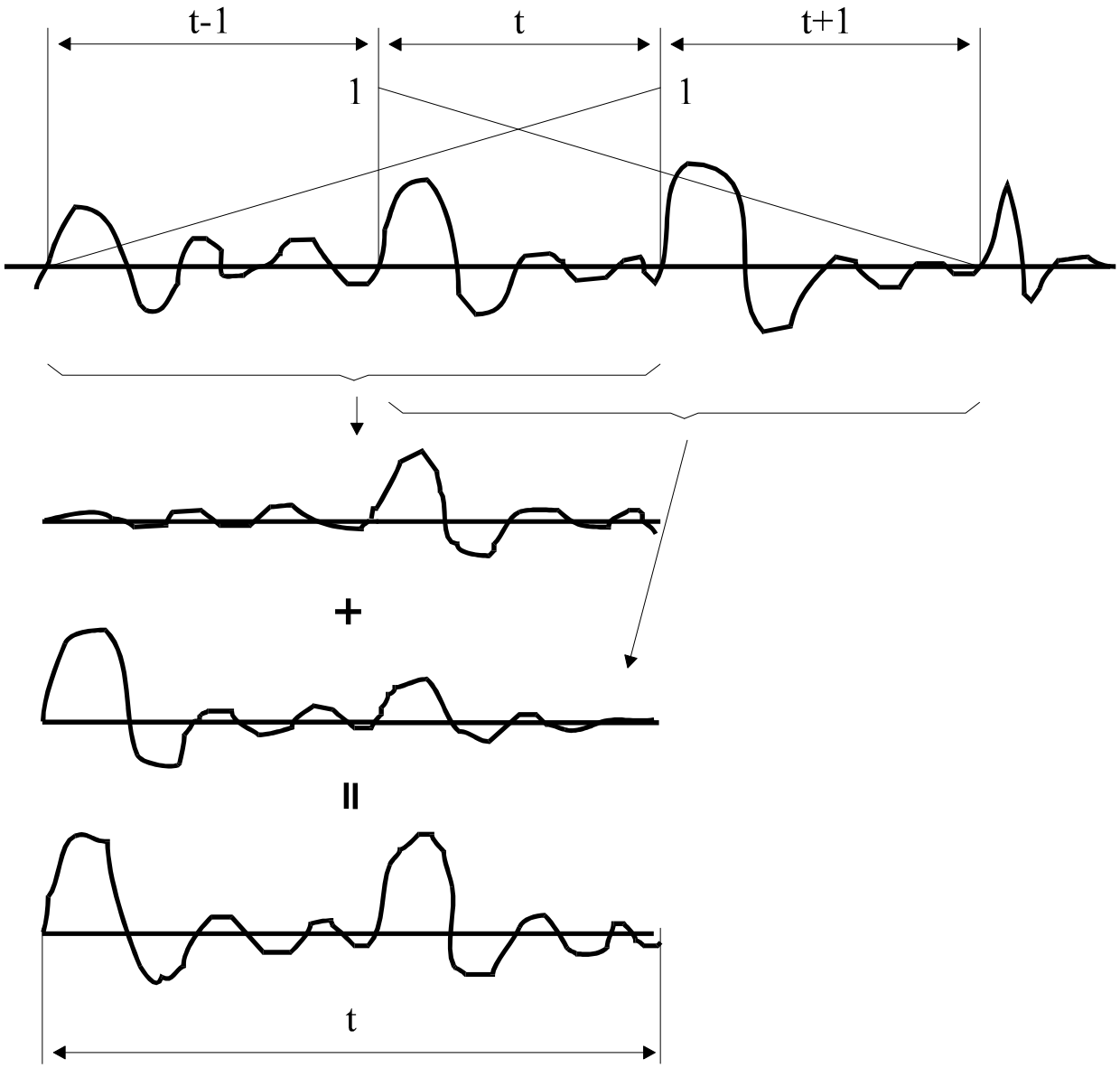
[1] [MALAH79], Malah et al. 1981, Crochiere et al, 1982



Le temps de calcul est faible puisque la procédure consiste en deux multiplications et une addition pour chaque échantillon.

Pour une expansion double, deux tranches consécutives t-1 et t sont multipliées par une demi-fenêtre du double de longueur et croissante et les tranches t et t+1 sont multipliées par une demi-fenêtre du double de longueur et décroissante. Le résultat de l'addition est une tranche du double de longueur au lieu de la tranche originelle t.

La figure suivante montre ce procédé :



Le temps de calcul est aussi faible que dans le cas précédent.

Les tranches n'ont pas besoin d'être de la même longueur mais si le pitch est détecté correctement, ceci rend cet algorithme de qualité impeccable. Sinon, le résultat peut être un peu flou par l'annulation de maximums et de minimums.

Deux facteurs influaient principalement sur la qualité de cette application. Le premier est la robustesse du détecteur de pitch comme on l'a dit. Si le détecteur n'est pas bon, les annulations sont très importantes. Le deuxième facteur est la fenêtre utilisée ; Des résultats assez bons ont été obtenus avec une approximation de la fenêtre de Hanning par un polynôme de troisième ordre[1]. Par exemple :

$$W(t) = 1 + 0.125t - 3.375t^2 + 2.25t^3 \qquad 0 <= t <= 1$$

Quoiqu'il puisse être étendu à d'autres facteurs (triple, quadruple, etc.), l'annulation augment considérablement si le signal a une variation très rapide du pitch.

[1]Des résultats peuvent être regardés dans [YUAN_91].

4.5. Emphase :

Les essais avec ces méthodes ont été bons pour des taux petits (0.5 à 1.5) et un fichier de source de bonne qualité (44 khz de fréquence d'échantillonnage sur 16 bits). Mais avec des taux plus élevés ou des fichiers de qualité mauvaise (11 khz sur 8 bits) ou pleins de bruits, l'algorithme ne marche pas très bien. Par rapport à l'extension, il apparaît un bruit de fond et des sifflements drôles, détectables principalement lors des silences, dus à l'apparition de fréquences « fantômes » par la répétition des tranches. En revanche, à la contraction, la parole se fait incompréhensible quoiqu'il ne s'agisse que de vitesses que tout le monde peut obtenir en parlant sans beaucoup d'effort. Mais, comment ?

Comme on l'a dit, quand nous parlons lentement ou vite, nous ne faisons pas la même expansion ou contraction de tous les phonèmes. Il y a des phonèmes caractéristiques qui ne sont pas très modifiés (consonnes comme p, t, k, b, d, g) et il y a d'autres phonèmes (voyelles, sibilants comme s, z, θ, ξ, ζ, γ, fricatives comme f, v, palatales comme r, l, λ, nasales comme m, n, η, et bien sûr, les silences) qui peuvent être réduits ou allongés sans problèmes.

Selon Saito[1], une expérimentation dont la vitesse avait varié de l'ordre de 30%-40%, a montré que l'expansion et la contraction des périodes de pause (silence) arrivaient jusqu'à le 65%-69%, tandis que pendant la parole strictement, la variation était de 13-19%. Cela veut dire que la variation de vitesse est principalement accomplie en changeant la longueur des périodes de pause. En plus, comme nous l'avons commenté, l'expansion ou contraction pendant les périodes des voyelles et sonores est généralement plus grande que pendant les périodes des consonnes et sourdes.

On va présenter certains paramètres qui prétendent différencier ces tranches caractéristiques. À première vue, ceux-ci paraissent très sophistiqués mais ils sont plus faciles à implanter que les précédents. Ces paramètres sont les suivants:

- Fréquence moyenne de la tranche considérée.
- Puissance moyenne de la tranche considérée.

Comme on l'a fait avec les paramètres de la partition, nous décrirons ensuite la signification concrète de chacun d'eux mais la caractérisation de chaque tranche sera déterminée par leur pondération.

4.5.1. Fréquence moyenne de la tranche:

Dans la procédure de partition, nous avons déjà calculé la fréquence instantanée. La fréquence de la tranche (Freq_Tranche) est facilement calculable. Pour évaluer la caractérisation d'une tranche on fera simplement :

$$fFreq = 1.0 - e^{-FreqTranche / FreqTypique}$$

En définissant la fréquence typique comme pondération :

$$FreqTotal = \sum FreqTranche$$

[1] [SAITO61]

$$FreqMoyenne = FreqTotal / NumTranches$$

$$FreqTypique = \beta * FreqTypique + (1 - \beta) * FreqMoyenne;$$

β étant un facteur de mémoire proche de 0.98.

Le tout donne une valeur de fFreq entre 0 et 1.

4.5.2. Puissance moyenne de la tranche:

Nous répétons les mêmes étapes que pour la fréquence; dans la procédure de partition, nous avons déjà calculé la puissance immédiate. Alors, la puissance de la tranche (Puis_Tranche) est facilement calculable. Pour évaluer la caractérisation d'une tranche on fera simplement :

$$fPuis = 1.0 - e^{-PuisTranche / PuisTypique}$$

En définissant la puissance typique comme une autre pondération:

$$PuisTotal = \sum PuisTranche$$
$$PuisMoyenne = PuisTotal / NumTranches$$

$$PuisTypique = \beta * PuisTypique + (1 - \beta) * PuisMoyenne;$$

β étant un facteur de mémoire proche de 0.98.

Le résultat est aussi une valeur de fPuis entre 0 et 1.

4.5.3. Pondération des facteurs:

Le facteur de puissance a été considéré positivement quoiqu'il y ait de sons non voisés qui ont une puissance faible. Heureusement, cet effet est contrecarré par le facteur de fréquence.

Le facteur de fréquence, cependant, est aussi considéré positivement, mais il faut faire une étude plus complète pour déterminer s'il n'est pas mieux de le compenser autour une valeur moyenne, jouant le rôle d'un filtre passe-bande.

Nous avons, alors :

$$fCarac = Pond\ (fFreq, fPuis)$$

Qui devient à l'implantation :

$$fCarac = fFreq^{Fact\_Freq} * fPuis^{Fact\_Puis}$$

Une fois que le facteur est calculé, son application finale est différente si'l s'agit d'une expansion ou d'une contraction. En plus, nous introduisons un autre facteur d'importance de la caractérisation (IMPO_CARACT de 0 à 1) pour contrôler la façon dont fCarac variera avec la recomposition.

La solution et les constantes présentées ci-dessous ont été ajustées d'une façon totalement expérimentale et elles sont complètement artificielles (une implantation avec arithmétique entière serait aussi possible):

```
if (IMPO_CARACT > 0.0) {
        if (d <= 1.) {
                fCarac'= fCarac^(1 / (3.0 * Fact_Ret + IMPO_CARACT))
                Rep2= (1.5 - fCarac')^IMPO_CARACT /d
        }
        else {
                fCarac'= fCarac^(1 / (3.0 / Fact_Ret - log10(IMPO._CARACT))
                Rep2= (fCarac' + 0.5)^IMPO_CARACT / d;
        }
}
else
        Rep2= 1/d;
```

Après, on fait de l'arrondi **linaire aléatoire** dont nous avons parlé précédemment. Si IMPO_CARACT = 0, il s'agit complètement de la méthode antérieure.

Le facteur Fact_Ret est pour le moment 1.0 mais il peut être augmenté pour retarder la suite voulue (que nous appellerons suite cible). Ce point sera commenté en détail dans la prochaine section.

4.5.4. Le réglage de la position:

Jusqu'à ce moment et si IMPO_CARACT est égal a 0, la suite cible a exactement la longueur désirée, mais avec l'effet de la caractérisation, il y a une déviation de la position recommandée qui peut résulter en un signal plus petit ou plus grand que celui attendu. Cette déviation est calculée de la manière suivante :

$$Dev0 = ((PosSource / d) - PosCible)$$

La position de la source est divisée par le taux "d" pour calculer la position recommandée à la cible. La différence entre cette position-ci et la position réelle, nous donne la déviation. Comme la déviation immédiate peut être trop variable, on fait une autre pondération :

$$Dev = \beta * Dev + (1 - \beta) * Dev0$$

β étant un facteur de mémoire proche de 0.98. Initialement, Dev sera logiquement à zéro.

Pour éviter notre problème de déviation, il peut être fait un réglage de la position :

$$RepReg = Dev / TAILLEPITCH$$

La déviation de la cible est divisée par la longueur du pitch de la cible pour calculer les fois à répéter cette tranche pour compenser la déviation.

Avec un facteur de réglage (Fact_Reg), nous recalculons une nouvelle valeur de répétition:

$$Rep3 = Fact\_Reg * RepReg + (1 - Fact\_Reg) * Rep2$$

Malgré tout, puisque l'effet n'est pas du tout éliminé, il est recommandable d'avoir un petit retard plutôt qu'une avance, parce que le premier peut être facilement résolu, en répétant la dernière tranche les fois requises. Néanmoins, une avance pourrait laisser la suite de la source pas complètement examinée.

4.5.5. D'autres retouches:

Comme résultat de tant de paramètres et pondérations, le nombre de répétitions peut avoir finalement quelques valeurs drôles. Par exemple, les conditions suivantes s'expriment par elles-mêmes.

```
   if ((Rep3 < 1) and (d <= 1)) {
          Rep3= 1;                      // Extension: Une fois comme minimum
!
   }
   else if ((Rep3 > 1) and (d >= 1)) {
          Rep3= 1;                      // Réduction: Une fois comme maximum
!
   }
   else if (Rep3 < 0)
          Rep3= 0               // Répétitions négatives ?
```

Les conditions précédentes doivent être évaluées à la suite de toutes celles que nous allons voir ensuite.

D'autres retouches ou restrictions ne sont pas si importantes que les précédentes et, à l'implantation finale, elles ne sont pas activées mais il peut être intéressant de les rappeler :

```
   if ((Tail > MAXTAILLEREP) or (Tail < MINTAILLEREP)) {
          Rep3= 1;
   }
```

Ceci veut dire que la tranche est trop petite ou trop grande pour être répétée. Nous pouvons avoir aussi :

```
   if (Rep3 > (NRep Exagere / d))
          Rep3= NRep_Exagere / d
```

Pour éviter des répétitions exagérées, ou :

```
   if ((Rep3 * Taille) > (TRep Exagere))
          Rep3= TRep_Exagere / (Rep3 * Taille)
```

Pour éviter des répétitions très longues en taille.

L'ordre, l'enchaînement et l'implantation finale des deux procédures étudiées peuvent être regardés en détail sur les fichiers source en langage C++.

4.6. Méthode spectrale :

Il y a une autre manière complètement différente d'aborder le problème. Pour la vitesse variable, si l'on ne veut pas changer le timbre et le pitch du locuteur, on doit respecter les fréquences originales.

Avec une transformée, on peut appliquer les effets désirés sans changer les composantes fréquentielles du signal. La méthode concrète est la suivante :

1. Partition uniforme du signal original (d'une longueur de N, étant N un nombre puissance de 2).

2. Application d'une fenêtre pour obtenir des meilleurs résultats dans l'étape suivante.

3. Application de la FFT à la tranche en question qui donne un tableau de fréquences de N valeurs et dont on prend les premières N/2 valeurs, selon le théorème de Nyquist-Shannon.

4. Transposition des N/2 valeurs à M/2 valeurs selon la vitesse voulue (on peut restreindre M à être une puissance de 2 également et on devra faire une compensation entre tranche pour obtenir la vitesse voulue). Si l'effet désiré est une vitesse plus lente (M > N), on remplira les valeurs qui restent avec des zéros. Sinon (M < N), on copiera les valeurs des fréquences les plus basses. On fait alors une duplication des valeurs dans un autre tableau de fréquences de M valeurs.

5. Application de la iFFT aux M valeurs. Ceci donnera une tranche temporelle de longueur M.

6. Recomposition et liaison uniforme des tranches. On peut utiliser l'une des méthodes présentées comme le TDHS par exemple.

Cette méthode a été essayée dans l'outil Accordion (sans la recomposition uniforme). Les résultats ne sont pas mauvais mais la complexité est beaucoup plus grande que pour la méthode temporelle avec aucune amélioration de la qualité.

Néanmoins, cette méthode peut être intéressante lorsqu'on l'utilise avec une méthode de compression spectrale qui réalise déjà les pas 1, 2, 3, 5 et 6. Alors, le coût de l'étape 4 est minimal et cette méthode sera donc plus facile à implanter.

4.7. Évaluation de la vitesse variable :

Abstraction faite de la méthode triviale mais très utile de dresser les oreilles pour entendre quelle méthode ou quelles valeurs des paramètres donnent une recomposition plus claire, propre et compréhensible, deux autres méthodes ont été effectuées:

I. Étant donnés l'original **B** à vitesse x et un autre original **A** à vitesse f*x, comparaison de **A** avec le fichier **C** produit par l'algorithme sur **B** avec le même facteur f.

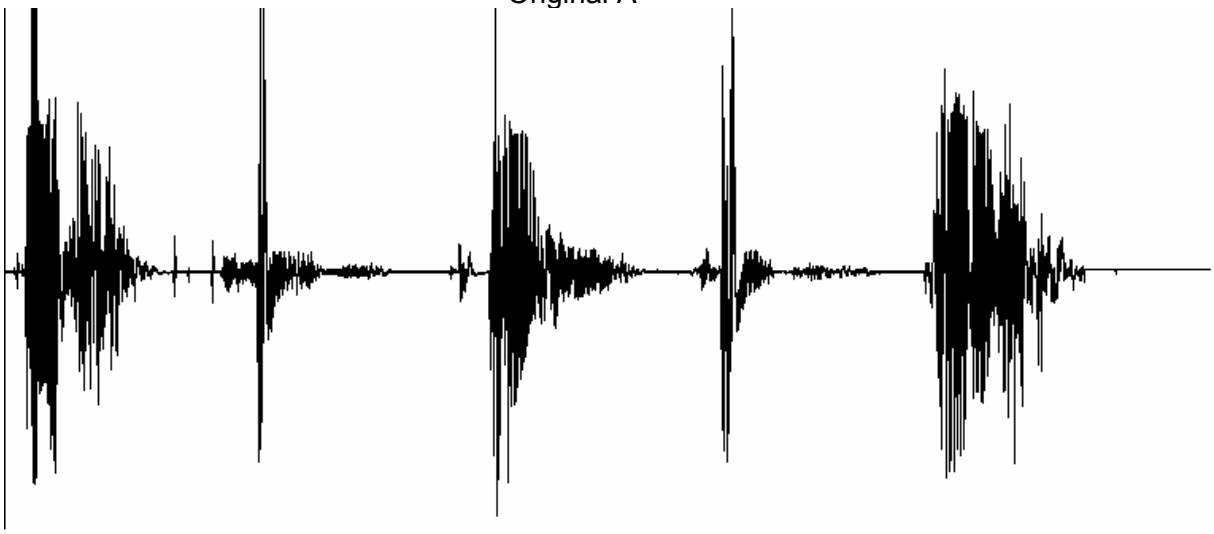
II. Étant donné un original **A**, on applique l'algorithme avec facteur de f, en donnant le fichier **B**, après on lui rapplique l'algorithme avec le facteur de 1/f. On compare ce dernier fichier résultant **C** avec l'original A.

Mais le grand problème continue à être que la comparaison est totalement subjective et très difficile quand il y a beaucoup de facteurs et paramètres mis en jeu. Il serait donc très approprié de trouver une mesure de similarité pour comparer **A** avec **C**. Celui-ci est le but de chapitre suivant : mesures de qualité.

Cependant, on peut montrer divers signaux comparés avec la méthode I. Le premier **A** est un original de 5 secondes de la phrase « UNO, DOS, TRES, CUATRO, CINCO », le deuxième représente un autre original **B** de 2,5 secondes de la même phrase, et un troisième **C** de 8 secondes. Après l'application de la vitesse variable avec f= 5/2,5 = 2 on a le signal **D** de 2,5 secondes et après l'application de la vitesse variable avec f= 0,625 on a le signal **E** de 8 secondes.

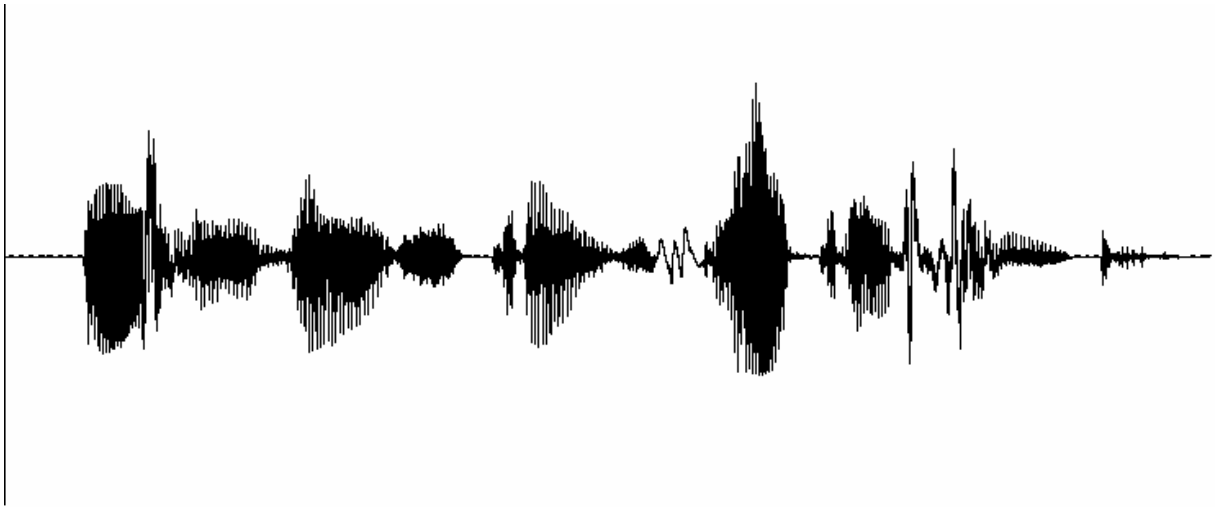
On montre l'original de 5 secondes :

Original A

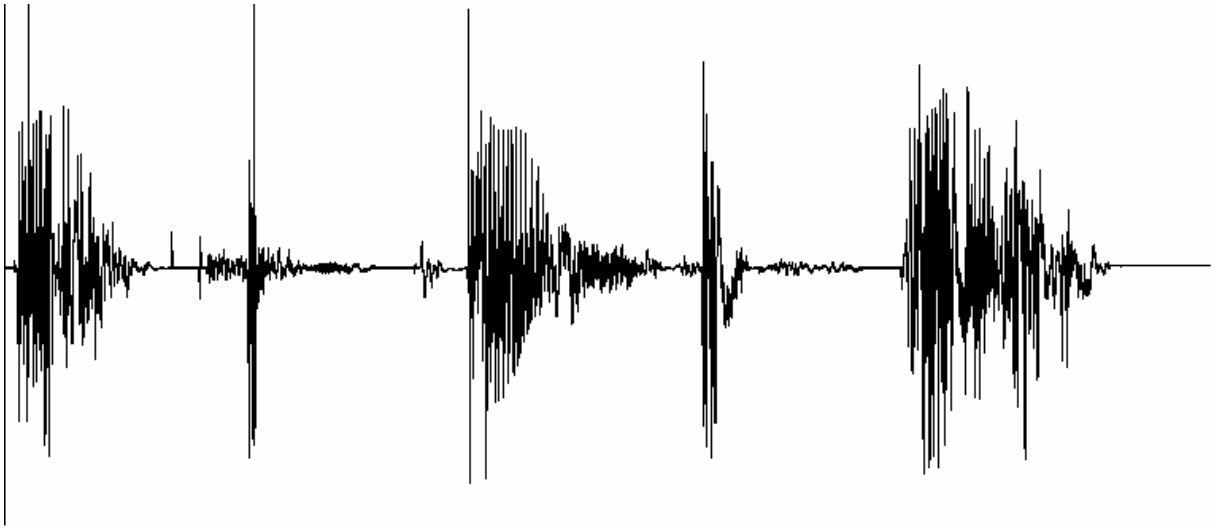


L'autre original plus vite B et celui réalisé par l'algorithme D tout les deux de 2,5 secondes.

Original B



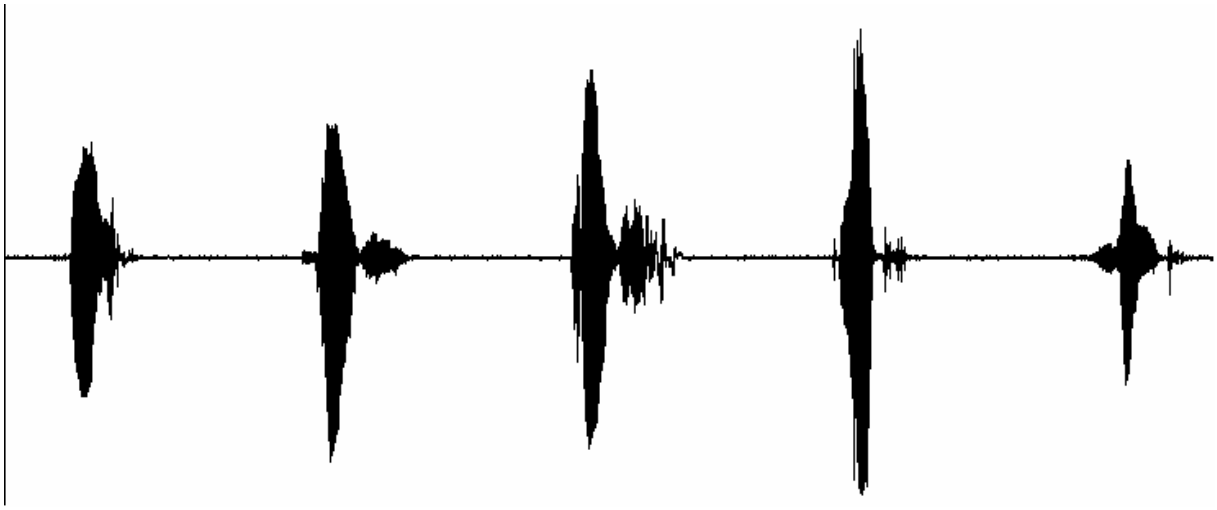
Généré D



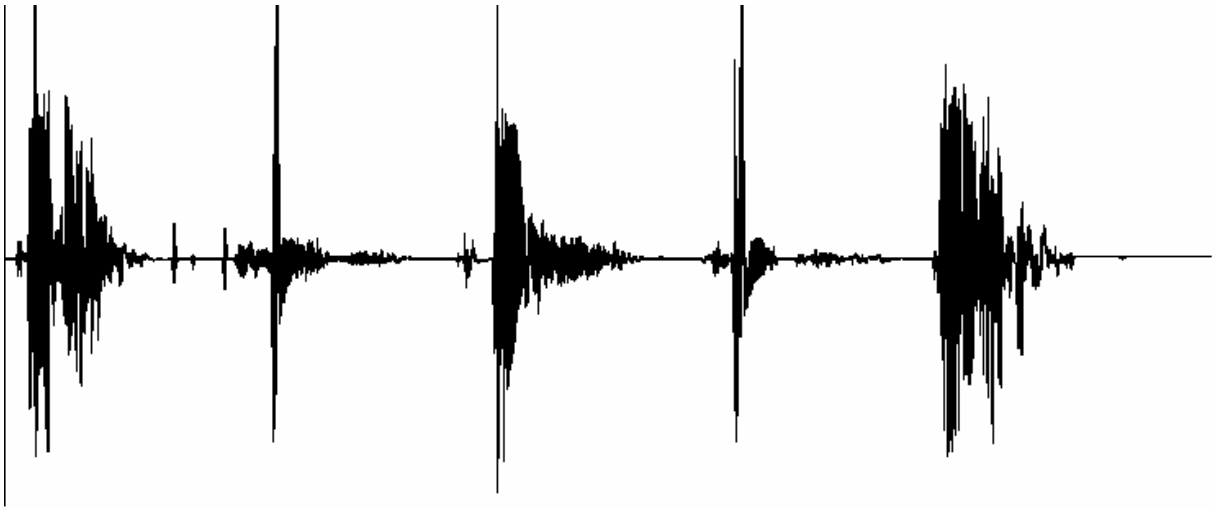
On constate l'emphase que dans les deux cas, les silences ont été diminués, par contre les données de parole sont encore semblables à celle de l'original.

Maintenant on montre l'original plus lent C et celui réalisé par l'algorithme, E ,tout les deux de 8 secondes.

Original C



Généré E



On constate que l'expansion a été mise principalement dans les espaces, comme ce que l'on fait normalement lorsqu'on parle plus lentement.

Les tests d'intelligibilité, qui seront étudiés aussi plus loin, peuvent être un outil très efficace pour évaluer si l'emphase et la restitution à une vitesse plus douce ont un effet positif pour la compréhension de la parole enregistrée.

En plus de la méthode de comparaison plus ou moins objective, il serait très intéressant de faire des essais avec d'autres sources de signal comme enregistrement de cassette, de radio, voix féminine et d'enfants, et bien sûr, d'autres langues. À cet égard, l'idée de Boris Siepert d'essayer avec des cours de langues en cassette est très intéressante, parce qu'elles constituent un abrégé de sources différentes.

Il faut aussi souligner l'avantage de l'existence de beaucoup de nationalités au laboratoire, ce qui permet d'éprouver les algorithmes directement sur différentes langues.

4.8. Implantation :

4.8.1. Programmation de l'algorithme :

La structure du programme[1] en C++ est un peu complexe dû au nombreux paramètres à fixer. La plupart d'entre eux sont fixés au moment de la compilation et les autres pendant l'exécution.

Pour rendre les algorithmes indépendants de la fréquence d'échantillonnage et du nombre de bits (8 ou 16), on a utilisé une normalisation de l'amplitude entre -1 et 1. On a fait la même chose avec des mesures temporelles, lesquelles ont été exprimées en millisecondes. Tout cela s'est fait en utilisant des nombres réels. Ceci ne présente pas de problème pour un Pentium (architecture PCI) à 60 mhz avec 16 Mo. de mémoire, mais les opérations en virgule flottante sont très lentes dans un microcontrôleur ou un DSP (s'il les supporte).

4.8.2. Mise en place :

Comme on le verra dans le chapitre sur les mesures de complexité, la première chose qui peut accélérer l'algorithme est la traduction en langage machine efficace de tout ou partie de ces algorithmes. Cela donne peut-être une réduction de la complexité de l'algorithme, mais on peut obtenir une efficacité plus grande grâce à des simplifications :

- Simplification opérationnelle :

    a) Utiliser l'arithmétique entière,

    b) Substituer les pondérations géométriques par des pondérations arithmétiques,

    c) Ne pas utiliser de fonctions comme tan, exp, log, etc., en trouvant leurs approximations entières (mieux encore, les coder dans une table),

    d) Tirer profit de l'architecture concrète de la puce cible (jeu d'instructions, adressage, registres, etc.),

    e) Implanter des générateurs de nombres aléatoires simplifiés (avec l'opération XOR par exemple).

- Simplification structurelle :

    a) Mépriser les facteurs peu importants,

    b) Réduire les actions adaptatives et correctives.

En particulier, dans notre cas, si la complexité définitive obtenue par ces simplifications n'est pas encore suffisante, on peut restreindre l'évaluation des points

[1] dans <rate.cpp>

dans la procédure de la partition qui semble être la plus compliquée. Cela peut être obtenu par:

- Simplification de la partition:

    a) Seulement évaluer les passages par zéro,

    b) Commencer par la position du Taille_Ant à gauche et à droite et arrêter quand la probabilité d'améliorer est faible ou, plus simplement, au premier pas par zéro,

    c) Donner un temps fixe de recherche du point de la coupe.

Logiquement, puisque ces dernières simplifications ne concernent pas toutes les données, celles-ci évitent de calculer la fréquence et la puissance moyennes dans la procédure de recomposition et deviennent donc inutiles.

Un autre point doit être considéré : la comptabilité avec les algorithmes de compression. À cet égard, on peut dire qu'aucune grande difficulté ne se trouve apparemment sauf, peut-être, si la partition est variable, et si l'algorithme de compression ne l'accepte pas.

Donc, notre algorithme doit être modifié pour une implantation définitive soit dans le Intel µC 80C196, dans l'Analog Devices ADSP 2111 ou comme ce sera le cas, dans le TEXAS TMS320C50.

4.9. Applications :

Bien que le but initial était l'apprentissage des langues, la bonne qualité et la facilité de réalisation, on a envisagé que les applications de la vitesse variable puissent être multiples :

4.9.1. Apprentissage des langues :

On peut l'utiliser comme une aide pour la compréhension des parties d'un texte sonore qui n'a pas été bien compris. L'enregistrement numérique et la restitution des dernières secondes, et l'opération aussi simple que presser une touche de l'appareil et sélectionner la vitesse voulue pour les réécouter plus clairement, fait soit le LLI, ou une application multimédia dans un ordinateur personnel, peut être d'utilisation courante par l'élève qui apprend une nouvelle langue.

4.9.2. Dictées et dactylographie :

Une lettre ou discours peuvent être reproduites plus lentement pour donner des temps suffisants pour les taper sans la nécessité d'arrêter la cassette périodiquement. Dans les tâches de secrétariat, cela pourrait être très utile pour économiser du temps et des efforts.

4.9.3. Ajustement avec une image ou un texte écrit :

Si on a une image d'une certaine durée et un son d'une longueur différente, on peut les ajuster sans couper aucun des deux. Ceci est très intéressant pour la publicité, la traduction de films, la lecture automatisée, etc.

On a vu cette intention dans les applications multimédia, mais l'ajustement est fait uniquement par l'allongement des silences, jamais par la modification des mots.

4.9.4. Ajustement de deux sons :

On peut mélanger deux voix séparées de différentes longueurs en une seule qui donnera l'impression de parler au même rythme. Ceci est aussi applicable pour ajuster la musique et la parole.

Les équipements de son modernes utilisent de plus en plus de capacités numériques et, l'incorporation de la vitesse variable ne serait pas une charge exceptionnelle.

4.9.5. Musique :

Quoiqu'il ne soit pas applicable à la musique très complexe, avec un enregistrement des instruments simples (piano, orgue, flûte, etc.) on peut changer la vitesse sans changer les notes un seul demi-ton.

4.9.6. Recherches à grande vitesse :

Dans l'avenir des enregistrements numériques dans les ordinateurs, la recherche d'une partie d'un enregistrement peut être trouvée plus aisément avec ces méthodes sans la traditionnelle méthode de « Cueing » (FF) qui a comme conséquence une désagréable fréquence aiguë.

Il faut dire qu'une mise en place efficace de l'algorithme serait ici nécessaire, étant donnée la vitesse de traitement de données requise.

4.9.7. Compression :

On peut voir la procédure de caractérisation comme la première phase ou comme un pas préalable à la compression. Par exemple, si l'on applique l'algorithme avec un facteur d=5, on obtient un fichier cinq fois plus petit. Si après on rapplique l'algorithme à ce fichier avec un facteur de d= 1/5, en théorie on devrait obtenir un fichier très semblable à l'original. On peut regarder ces deux processus comme compression et décompression avec un taux de 5.

Malheureusement, cette procédure ne fonctionne pas bien à moins que la partition (détection du pitch) et la qualité du fichier source soit bonne et le facteur d soit petit. Mais tout ceci peut servir comme commencement pour développer un algorithme capable de faire la partition, la compression et la caractérisation vers un nouveau format de données dans un fichier ou mémoire, lesquelles seront facilement recomposables (et décompressées en même temps) à la vitesse désirée. Des essais ont été assez satisfaisants, au moins comme une technique supplémentaire à d'autres méthodes de compression ; on a vu par exemple la technique de compression TDHS (Time Domain Harmonic Scaling).

4.10. Conclusions :

Les résultats ont été encourageants en regard de la complexité et du faible temps employé pour le développement des algorithmes. De plus, l'inexistence de références sur la matière et la masse d'applications futures, fait que cette partie du projet est la plus satisfaisante pour ses aboutissements qu'elle a procuré et pourra procurer donner.

Comme on l'a dit, une étude de la compréhensibilité des signaux traités par l'algorithme serait très intéressante pour valider les méthodes d'emphase. En plus, elle justifierait l'utilisation de la méthode pour l'apprentissage des langues et des autres applications présentées.

Chapitre 5

MESURES DE QUALITÉ

Il faut clairement établir une méthode pour évaluer tous les algorithmes et techniques présentées jusqu'ici. Ce n'est pas une tâche facile, comme on le verra par la suite. Il n'est pas étonnant que beaucoup de chercheurs aient dédié énormément de temps et d'efforts à ce domaine, sans obtenir les aboutissements mérités.

5.1. Introduction :

Le terme « qualité » est une manière de combiner des attributs différents, et il y a beaucoup de façons de les pondérer. Mais trois facteurs semblent les plus importants :

- Intelligibilité (une mesure de compréhensibilité).
- Marque (score) d'articulation (une mesure de reconnaissance des phonèmes).
- Identification du locuteur (très important pour les applications militaires).

Malheureusement, ces trois facteurs ne sont applicables qu'avec des qualités pauvres. Une bonne qualité doit s'assurer au maximum des trois attributs précédents. Une exception peut être faite dans l'évaluation de la vitesse variable, étant donné que l'intelligibilité est réduite dans le cas d'une accélération et qu'elle doit être améliorée dans le cas de la restitution à vitesse plus lente.

Dans les enregistrements de qualité, il n'y a qu'un attribut à considérer : le bruit qui reste après l'application du traitement. Le problème des méthodes habituelles comme SNR (Signal to Noise Ratio), RMSE (Root Mean Square Factor) et MAE (Mean Absolute Error) est qu'elles ne résultent que d'une approche grossière à les mesures subjectives de la qualité. Par exemple, une addition d'une valeur entière comme 20 à chaque échantillon rend un signal que les humains ne peuvent distinguer de l'original, mais qui donne des mesures d'erreur très élevées. Jusqu'aujourd'hui, une mesure objective de la qualité et aisément réalisable n'a pas été développée : les oreilles humaines sont encore les meilleurs juges.

5.2. Mesures subjectives :

L'évaluation subjective inclut des tests d'opinion, la comparaison de paires et des tests d'intelligibilité. Deux rapports très intéressants sont les articles de Panzer[1] ou de Dimolitsas[2]. Les méthodes les plus utilisées sont les suivantes :

- DAM / CAE (Diagnostic Acceptability Measure / Composite Acceptability Estimate) : Développé par Voiers[3], elle combine une évaluation isométrique (quantifiable) et une évaluation paramétrique. Tout cela est pondéré de manière adéquate pour obtenir une valeur CAE.

- ACR / MOS (Absolute Category Rating / Mean Opinion Score): Décrite par Goodman[4]. Dans les tests d'opinion, des scores subjectifs son mesurés usuellement sur cinq niveaux : 5 excellent, 4 bon, 3 moyen, 2 pauvre et 1 mauvais. Le MOS (Mean Opinion Scores) est ensuite calculé.

- DCR / DMOS (Degradation Category Rating / Degradation Mean Opinion Score) : Elle est très semblable à la précédente mais il s'agit d'une mesure de la dégradation en cinq niveaux : 5 Dégradation inaudible, 4 Dégradation audible mais pas ennuyeuse, 3 Légèrement ennuyeuse, 2 Ennuyeuse et 1 Très ennuyeuse. Normalement, cette méthode est mixée avec la précédente.

- ETR (Equality Threshold Rating) : Elle fait la mesure d'égalité entre un signal A (MNRU) et un signal B (à évaluer) comme une variation sur le 50 % de préférence de B sur A (un 51% en théorie doit être impossible)[5].

- DRT (Diagnostic Rhyme Test) et MRT (Modified Rhyme Test): Orientée vers l'intelligibilité des signaux de mauvaise qualité, différents mots sont présentés et on doit choisir la plus proche au son écouté.[6]

- Q-value ou SNR_q : Puisque le MOS indique seulement la qualité relative dans un ensemble de résultats, le « opinion-equivalent SNR » (SNR_q) a été proposé pour assurer que le MOS soit mis en relation avec une mesure objective[7]. La procédure consiste à ajouter du bruit blanc jusqu'à obtenir un MOS déterminé. Le SNR théorique calculé avec ce bruit est mis en relation avec la valeur du MOS. Ce bruit est couramment le MNRU (Modulated Noise Reference Unit) qui est recommandé par le CCITT[8].

[1] [PANZE93]
[2] [DIMOL93].
[3] [VOIER77].
[4] [GOODM82] [IEEE_82].
[5] Peut-être si dans la vitesse variable
[6] [HOUSE65], [PECKEL73] et [VOIER77]
[7] Richards, 1973
[8] Voire [CCITT88].

Une façon de diminuer la subjectivité est d'essayer les algorithmes à évaluer en cascade, ce qui donne une détérioration de la qualité assez forte pour percevoir clairement les différences.

Dans mon projet, aucune de ces méthodes n'a été essayée dû au temps nécessaire pour faire les tests et pour sélectionner les sujets du test, les conditions, etc. En plus, le but était de réaliser une mesure objective comme va le présenter maintenant :

5.3. Mesures objectives :

Elles ont été développées à partir des différentes mesures de qualité pour permettre à cette opération d'être réellement objective. De plus, ne nécessitant pas l'oreille humaine, tout le processus de mesure peut être automatisé.

5.3.1. Méthodes Théoriques :

Elles sont applicables directement à la formule des algorithmes de compressions simples selon le nombre de bits, la fréquence d'échantillonnage, etc.

5.3.1.1. SNR (Signal to Noise Ratio) :

Couramment utilisée dans les techniques audio, il fournit le rapport entre la puissance moyenne du signal et celle du bruit, mesuré en dB. Dans les procédures de compression, le bruit principal à considérer est le bruit d'erreur de quantification.

Comme son nom indique, il s'agit d'un rapport signal à bruit : plus il est grand, meilleure est la qualité. Par exemple, pour le son, des qualités de 70 dB sont très courantes aujourd'hui. Pour la parole, cependant, les objectifs sont plus modestes et un SNR de 25 dB est considéré comme déjà assez bon.

Des formules pour les SNRs ont été développées pour la plupart des algorithmes de compression PCM, DPCM, DM, ADM et même ADPCM[1]. Le problème apparaît quand les algorithmes sont un peu plus compliqués, dont les formules peuvent être trop longues et complexes. Même si celles-ci sont possibles, dans ces algorithmes très complexes ou dans d'autres traitements comme la vitesse variable ou les filtrages, la comparaison avec l'original est complètement différente du résultat espéré.

5.3.1.2. Largeur de Bande (Bandwidth) :

Elle mesure la largeur de toutes les fréquences qui passent sans perte dans le signal traité. Bien qu'elle puisse paraître très apparente pour des traitements simples, pour des algorithmes ou techniques complexes, la bande n'est pas continue et les valeurs réelles de largeur de bande ne sont pas évidentes.

5.3.2. Méthodes Expérimentales :

Ces méthodes peuvent être généralement classifiées en deux groupes. Le premier applique des signaux artificiels (tons) et mesurent des paramètres spécifiques comme le bruit et la perte globales. Le second s'applique directement aux signaux originaux. Des essais initiaux peuvent être réalisées avec la première sorte de signal et ensuite les signaux réels peuvent confirmer des résultats.

[1]On peut les trouver dans [BOITE87]

On présentera des techniques pour calculer la similarité (ou inversement, la distance, selon la formule utilisée) de deux signaux quelconques. Après, le choix des suites à comparer dépendra de l'algorithme à évaluer.

A cet égard, les deux premières techniques ont été mises dans l'outil de simulation et les autres ont été évaluées.[1]

- Minimum Squares Likelihood (MSL) : Il s'agit de calculer la moyenne des carrés des différences entre deux signaux.

- Distorsion Fréquentielle ou Spectrale: C'est presque la même chose que la méthode précédente mais la comparaison est réalisée en calculant la moyenne des carrés des différences entre les spectres. On prend chaque spectre comme un tableau et on calcule ainsi la moyenne des carrés. L'avantage de la comparaison spectrale est que les effets d'addition du continu ne comptent pas dans la mesure.

- Distorsion-Comparaison Cepstrale (spectre inverse) : La distance cepstrale (CD : Cepstral Distance) est calculée comme suit [2]:

  $$CD = SQRT [2 \sum_{i= 1..p} (a_i - b_i)^2]$$

  On peut la mesurer en dB avec la transformation suivante :

  $$CD_{dB} = (10 / \ln CD) / \ln 10$$

- Le SNR_{seg} (segmental SNR) est une variété du SNR mesuré en dBs sur des périodes courtes d'environ 30 ms, puis moyennés. Le SNR_{seg} s'ajuste plus aux valeurs subjectives comme le MOS.

- Le NMR (Noise-to-Mask Ratio) : Divers articles et commentaires dans Internet[3] parlent d'un algorithme développé par Fraunhofer-HS[4] qui approche assez les résultats théoriques du SNR (Signal-to-Noise Ratio, Rapport Signal à Bruit).

Les résultats des comparaisons ont été toujours mesurés par une valeur de similarité variable entre 0 (complètement différents) et 1 (exactement le même signal).

Deux versions existent pour chaque méthode : linéaire et dynamique. La première est utile pour étudier les algorithmes de compression et des signaux de même longueur. Chaque échantillon est comparé uniquement avec un autre échantillon du signal de référence ; il s'agit donc d'un processus linéaire assez rapide.

Quant à la version dynamique, dans la dernière année de ma licence, j'ai fait une étude bibliographique de beaucoup d'algorithmes de similarité entre suites et chaînes[5]. Il est dommage que la plupart de ces mesures ne soient pas applicables à ce cas. Mais il y a une technique connue sous le nom de « tracés élastiques » qui peut

[1] [DIMOL89] et [CCITT84]
[2] Kitawaki et al. 1982
[3] INFO.TXT for MPEG Audio Layer-3 Shareware Code at <layer3@iis.fhg.de> ou <phade@cs.tu-berlin.de>
[4] Fraunhofer-IIS, Erlangen, Allemagne : <nmr@iis.fhg.de>. Le probleme est que la distribution n'est pas gratuite.
[5] [AHO__90a]

donner de bons résultats. Celle a été étudiée par Sakoe[1] sur des applications de reconnaissance de la parole où il y a beaucoup de contractions et d'expansions, ce qui fait apparaître beaucoup de valeurs répétées. La mesure de Leivenshtein (distance d'édition) peut aussi se comporter correctement avec un faible chargement aux insertions et aux effacements.

Cette méthode est couramment connue sous le nom de DTW (Dynamic Time Warping), une méthode de programmation dynamique couramment utilisée dans la reconnaissance de la parole et, permet de comparer des sons de différentes longueurs. Ceci permet la comparaison de signaux d'origine très variée, même issus de différents locuteurs, à vitesse variable, etc...

[1] [SAKOE79]

5.4. Sources des sons :

La plupart de sons ont été enregistrés par les membres de l'ECS directement via le microphone ou par enregistrement de cassette. D'autres fichiers de son ont été utilisés d'origines très diverses : Fichiers de démonstration, fichiers .WAV de WINDOWS fournis avec différentes cartes, fichiers d'autres formats et systèmes d'enregistrement, ou encore des fichiers générés artificiellement (générateur d'ondes) par l'outil de simulation Accordion ou d'autres.

L'avantage de la multinationalité du laboratoire a permis d'essayer avec les langues suivantes : français, espagnol, anglais et allemand. Les voix ont été surtout masculines mais les algorithmes présentés n'ont aucune raison apparente pour se comporter de façon différente avec des femmes. De plus, quelques unes ont été essayées. Même les voix d'un bébé ou d'un chien ont été introduites dans l'ordinateur pour le plaisir et la curiosité des résultats.

5.5. Relation fréquence d'échantillonnage - nombre de bits :

Différentes fréquences d'échantillonnage ont été essayées. La relation entre fréquence d'échantillonnage et résolution pour le même débit de données est un paramètre important à choisir pour économiser le stockage. Par exemple, avec 22 khz et 12 bits (254 kbps), la qualité est meilleure qu'avec 44 khz et 8 bits (352 kbps).

L'application des systèmes de codage fait que la seule analyse de cette relation est superflue car, quand on applique les algorithmes de compression, les prémisses établies sur la relation cessent d'être valides. Généralement, les systèmes de codage fonctionnent bien avec des rapports fixes. Par exemple, l'ADPCM à 8 khz et 4 bits est un standard pour la transmission à 32 kbps. On a essayé avec 22 khz et 2 bits et les résultats sont un peu décevants, mais avec 12 khz et 3 bits ils sont aussi bons que le standard.

Comme conclusion, on peut dire que chaque algorithme a son domaine de fonctionnalité. Au-delà de certaines limites, la qualité commence à se détériorer fortement.

5.6. Applications :

Le but initial était de faire une mesure de qualité pour faire une évaluation des algorithmes présentés dans ce travail et dans d'autres en général. L'application des techniques de DTW permet qu'une grande variété d'algorithmes soit essayée. Elles rendent possible la comparaison des algorithmes de vitesse variable et des algorithmes de compression paramétrique dont la correspondance biunivoque entre échantillons est douteuse.

Tout cela ouvre la voie à une application assez intéressante pour l'apprentissage des langues: la comparaison d'un original (professeur, etc.) avec le son enregistré par l'élève ; on peut ainsi donner une mesure de la qualité et une amélioration de la prononciation d'une langue étrangère.

5.7. Tableaux de résultats :

Le tableau suivant montre les résultats de qualité des différents algorithmes mis en place pour les mêmes signaux d'essai originaux pris comme référence pour les comparaisons :

	44 khz 16 bits QUALITE1.WAV	22 khz 16 bits QUALITE2.WAV	11 khz 16 bits QUALITE3.WAV
Réduction PCM à 12 bits	0.012	0.015	0.032
log-PCM 10 bits	0.008	0.010	0.030
μ-law	0.011	0.016	0.045
Réduction PCM à 8 bits	0.13	0.15	0.23
ADM	0.042	0.082	0.22
ADPCM 4 bits	0.0026	0.0031	0.015
ADPCM 3 bits	0.011	0.015	0.05
ADPCM 2 bits	0.048	0.053	0.15
DCT avec qualité 1.0	0.12	0.17	0.23
Vitesse variable f= 0.5 et 2	0.05	0.072	0.15

La méthode utilisée pour ce tableau est la comparaison spectrale dynamique. Le tableau donne une distance entre 0 et 1 pour une ressemblance nulle ou totale, respectivement. Pour la vitesse variable, cette comparaison spectrale dynamique est plus appropriée.

5.8. Conclusions :

Malheureusement, deux difficultés persistent dans ces mesures de qualité :

La première, et la plus importante, est que la valeur donnée par les algorithmes appliqués à la vitesse variable ne correspond pas toujours avec l'opinion subjective de l'être humain. La grande quantité des facteurs mis en jeu dans une évaluation de la qualité fait que cette tâche est encore très loin d'être automatisée objectivement,

La deuxième est qu'avec un processeur très puissant, la réalisation des calculs est encore trop lente en raison de l'utilisation des techniques de DTW ; ce dernier problème est semblable à celui de la reconnaissance de la parole, dont des nouvelles avancées permettent ces analyses en temps réel.

Par contre, comme résultat positif, on peut dire que ces mesures peuvent être utilisées dans une première phase de sélection des algorithmes pour réduire le nombre de sons à comparer et, à la fin, laisser à nos oreilles de prendre la dernière décision.

Chapitre 6

MESURES DE COMPLEXITÉ

Au moment de la mise en place, tous les algorithmes ne sont pas réalisables sur une cible concrète en raison des restrictions de puissance, de mémoire... Beaucoup des méthodes précédentes doivent encore être simplifiées fonctionnellement comme on vient de le dire ou seront accélérées opérationnellement.

Il serait très décevant de choisir un algorithme, de l'adapter pour une application concrète et, à la fin, lorsque tout est mise en oeuvre, de se rendre compte que la puce n'est pas capable de le réaliser. Pour éviter cela, une mesure de complexité préalable des algorithmes est une manière d'économiser du temps et des efforts.

6.1. Introduction :

Il est toujours difficile de calculer le temps réel qu'un algorithme utilisera et s'il est réalisable sur une puce cible déterminée.

Dans l'outil de simulation, je n'ai pas fait de gros efforts pour l'efficacité des algorithmes. Tout a été fait avec les plus grandes résolutions, les opérations mathématiques complexes et aucune d'économie de mémoire ou de ressources.

Le résultat est, comme on le commentera dans le chapitre de l'outil Accordion, des algorithmes qui, avec un Pentium, sont assez lents. Tout cela donne l'impression qu'ils ne seront pas réalisables à temps réel dans les DSP actuels. Mais, ce n'est pas le cas.

Comme on le verra, avec une codification optimale et une simplification fonctionnelle et opérationnelle, la plupart d'entre eux peut fonctionner sur les DSP actuels, comme nous l'essayerons sur le processeur Texas TMS 320C50.

Cependant, je veux faire cette étude générale pour des puces quelconques. Ensuite, je présenterai comme j'ai réussi à le faire de manière approchée, mais avec des mesures facilement applicables à toute puce, que, à l'avenir, on peut les utiliser pour les mises en oeuvre (même si la puce est encore en développement, comme c'est le cas de l'ASIC).

6.2. Adaptation des algorithmes :

On traitera ici des améliorations sur le logiciel. Étant donné que ce n'est pas le but de ce travail de trouver de nouvelles architectures pour le traitement de la parole, les différentes configurations du matériel seront commentées légèrement dans le chapitre de mise en oeuvre.

La première chose qui puisse accélérer l'algorithme est la traduction en langage machine, ou langage C pur et efficace, de tout ou partie de ces algorithmes. Cela donne peut-être une réduction de la complexité de l'algorithme, mais elle n'est pas encore suffisante pour notre propos.

On peut obtenir une efficacité plus grande avec d'autres techniques :

6.2.1. Simplification opérationnelle :

Dans le codage final, toutes ces techniques peuvent être appliquées indépendamment de l'algorithme pour une puce cible déterminée :

a) Tirer profit de l'architecture concrète de la puce cible (jeu d'instructions, adressage, registres, etc.),

b) Utiliser de l'arithmétique entière avec la résolution la plus petite possible,

c) Ne pas utiliser de fonctions comme tan, exp, log, etc., en trouvant leurs approximations entières ou polinômiques,

d) Utiliser des tables pour coder les fonctions mathématiques, les fenêtres, les coefficients de codage, etc.,

e) Implanter des générateurs de numéros aléatoires simplifiés (avec l'opération XOR par exemple).

Quelques fois, une option doit être abandonnée par manque de mémoire. Par exemple, la source de l'ADPCM qu'on présentera dans le chapitre 8, ne fait pas utilisation des tables pour certains calculs en raison des restrictions de mémoire de programme de la puce Texas TMS320C50.

6.2.2. Simplification structurelle :

Selon l'algorithme concret, quelques recoupements donnent des résultats un peu plus mauvais mais la simplification et, en conséquence, l'efficacité sont améliorées. En général, ces simplifications sont très dépendantes de l'algorithme en question, mais il y a quelques règles à suivre :

a) Mépriser les facteurs peu importants,

b) Substituer les pondérations géométriques par des pondérations arithmétiques,

c) Réduire le nombre de coefficients des prédicteurs,

d) Réduire les actions adaptatives et correctives.

Dans la vitesse variable par exemple, il y a beaucoup de pondérations qui ont été réalisées « sauvagement », avec des fonctions comme exp, log, etc. pour une raison de commodité, mais elles ne sont pas indispensables.

6.3. Mesures théoriques :

Elles sont plus faciles à réaliser et sont bonnes pour mesures la qualité d'un algorithme mais elles ne servent pas pour savoir s'il est réalisable sur une puce concrète. Les informaticiens aiment beaucoup cette sorte de mesures, car on a toujours envie d'utiliser des langages de programmation du plus haut niveau possible avec un nombre immenses d'itérations (nos lots « batchs » connus pour leurs heures de CPU).

Il est clair, cependant, que, dans une première phase, elles sont très appropriées. Par exemple, personne n'utilise aujourd'hui la DFT courant d'ordre $O(N^2)$ opérations mais plutôt les algorithmes FFT d'ordre O(NlogN).

Dans une deuxième phase, l'indépendance de la puce est, précisément, leur plus grand inconvénient. Deux algorithmes du même ordre, installés sur différentes puces, peuvent donner des résultats très distincts.

Pour s'approcher un peu plus des résultats que l'on aura au moment de la réalisation, j'ai développé les mesures suivantes...

6.4. Mesures approchées :

Avant de réaliser tout le code et pour constater ensuit qu'il ne marche pas, il faut mieux évaluer a peu près une mesure de la puissance de calcul requise. La même chose peut être réalisée pour la mémoire mais, dans nos algorithmes, la plupart du stockage est occupé par les données du signal, et non par les autres structures (piles, arbres, tableaux, etc.) construites par le programme lui même.

Pour approcher la mesure d'un algorithme réel sur une puce concrète, il est besoin d'introduire deux sortes d'informations : Primo, dans la source de l'algorithme, on effectuera un comptage des instructions que l'on espère voir finalement traduites en code machine ; Secundo, il faut établir les paramètres de puissance de la puce spécifique.

Avec ces informations, on va calculer une valeur du temps de calcul à partir duquel on décidera si la réalisation est possible.

6.4.1. Calcul des instructions machines requises pour l'algorithme :

La codification finale de l'algorithme peut différer de la source originale dans un langage de haut niveau comme le C++. On doit imaginer que l'on a fait toutes les optimisations dont on a parlé dans le point 6.2. lors de la programmation en assembleur, ou dans le C efficace si le compilateur est disponible pour la puce.

Généralement, il s'agit d'instructions arithmétiques que tout le monde connaît bien : Addition, multiplication, décalage, boucles, accès à mémoire, indexation, appels, etc.

Dans l'outil Accordion, on a fait la relation suivant des opérations d'un processeur courant dans la classe C++ nommé ***OP*** :

Sigle	Signification
SGL	SinGLe instruction (other than the following)
ADD	ADDition
SFT	ShiFT
ADSF	ADdition & ShiFt
MUL	MULtiplication
ADMU	ADdition & MUltiplication
ADMUSF	ADdition, MUltiplication & ShiFt
DIV	DIVision
RPI	RePeat Instruction
RPB	RePeat Block
LOD	LOaD from memory
STR	SToRe to memory
LOST	Load and Store
PUSH	PUSH to stack
POP	POP from stack
MEQ	extenal MEmory Quick
MES	external MEmory Slow
IND	Indexation
IN	IN from port
OUT	OUT to port
CAL	CALl to subroutine
BRA	BRAnch

Les instructions qui ne cadrent pas avec les précédentes doivent être réalisé par une combinaison d'instructions plus simples. C'est le cas de la racine carré, le cosinus, etc. Pour faciliter cette tâche, on a une instruction spéciale appelé MAX qui sert comme valeur de recours quand on veut mettre une instruction très longue.

De plus, pour toutes les instructions il y a différents types d'opération, comme on le montre dans le tableau suivant, insérées aussi dans la classe ***OP*** :

Sigle	Signification
I8	Arithmétique entière de 8 bits
I16	Arithmétique entière de 8 bits (short)
I32	Arithmétique entière de 8 bits (long)
X16	Arithmétique réelle fixe de 16 bits
X32	Arithmétique réelle fixe de 32 bits
X64	Arithmétique réelle fixe de 64 bits
R16	Arithmétique réelle flottant de 16 bits
R32	Arithmétique réelle flottant de 32 bits (float)
R64	Arithmétique réelle flottant de 64 bits (double)
R80	Arithmétique réelle flottant de 80 bits (long double)
C16	Arithmétique complexe de 32 bits
C32	Arithmétique complexe de 32 bits

J'ai fait une classe C++ nommé ***cost*** assez simple pour ajouter dans les algorithmes une sorte de compteur d'instructions. On doit être prudent avec l'endroit où l'on place les comptages. Par exemple, la fonction principale de l'algorithme PCM μ-law est la suivante:

```
void LogPCMCompress(sample_array *input, long Offset, BYTE huge * &output,
                    long Len, int Bits, cost & COST) {
    int i;
    unsigned char c;
    int BitOffset= 0;

    for(i= 0; i < Len; i++) {
                                                COST+= OP::RPB;
          c= input->Get(Offset + i);
                                                COST+= OP::LOST;
          OutputBits( output, (unsigned long) compress[ c ],  Bits,
BitOffset );
                                                COST+=    OP::LOST;    COST+=
OP::IND;
                                                COST+=     OP::SFT;    COST+=
OP::SGL;
                                                COST+=     OP::SFT;    COST+=
OP::LOST;
    }
}
```

Il y a des répétitions, des accès a mémoire, des indexations, des décalages, etc. Cette codification est très simple et il faut peu de compteurs mais, pour d'autres algorithmes, le programme résultant peut être entaché d'opérations coûteuses.

6.4.2. Concrétisation pour une puce déterminée :

Normalement, les DSP disposent de certains paramètres caractéristiques : un temps de cycle, une longueur de mots par défaut, arithmétique entier, fixe ou flottante, une surcharge (overload) due aux interruptions d'entrée et de sortie du signal, et pour finir, des instructions spécifiques.

Concrètement, on a fait aussi une classe C++ nommé ***DSP*** pour gérer ces paramètres.

Membre	Signification
O[OP::MAXOP][OP::MAXVR]	Tableau avec les cycles de chaque instructions
Cycle	Temps de cycle en nanosecondes
DefVR	Type d'opération utilisé par défaut
InOutOverload	Surcharge des interruptions d'entrée et de sortie du signal

Le type d'opération par défaut peut être choisi parmi ceux présentés précédemment.

6.4.3. Exemple. TEXAS TMS-320C50 :

Pour éclaircir le point précédent, on va montrer les paramètres choisis pour le TMS-320C50 :

```
Default->Cycle= 50;                                  // 50 ns. par cycle
Default->InOutOverload= 0.02;                        // Surcharge I/O
Default->Desc= "Texas Instruments TMS320C50 DSP 40 MHz";
Default->DefVR= OP::I16;                             // Entier de 16 bits

Default->O[OP::MAX][OP::I16]= 1000;

Default->O[OP::SGL][OP::I16]= 1;
Default->O[OP::SFT][OP::I16]= 2;
Default->O[OP::MUL][OP::I16]= 2;
Default->O[OP::ADSF][OP::I16]= 2;
Default->O[OP::ADMUSF][OP::I16]= 3;
Default->O[OP::DIV][OP::I16]= 26;
Default->O[OP::RPI][OP::I16]= 2;
```

```
Default->O[OP::RPB][OP::I16]= 2;
Default->O[OP::LOD][OP::I16]= 2;
Default->O[OP::STR][OP::I16]= 2;
Default->O[OP::LOST][OP::I16]= 3;
Default->O[OP::IND][OP::I16]= 2;
```

Ces données ont été extraites directement du manuel et quelques simplifications ont été effectuées. Si une opération n'est pas trouvée, la valeur SGL est assignée par défaut. C'est pour cela, que des autres instructions n'ont pas été codées.

6.5. Tableaux de résultats :

Le tableau suivant montre les résultats de complexité des différents algorithmes mises en place dans l'outil de simulation pour les mêmes signaux :

Pour le Texas TMS-320C50, avec un signal échantillonné sur 16 bits à 11,025 khz de 5000 échantillons (donc, de longueur 0,454 s) l'approche donne exactement des résultats suivants :

	44 khz 16 bits	22 khz 16 bits	11 khz 16 bits
μ-law	0%	0%	0%
ADM	0%	0%	0%
ADPCM 4 bits	6%	3%	1%
DCT avec qualité 1.0	35%	20%	12%
Vitesse variable f= 0.5	4%	2%	1%

La valeur exprimée représente en pourcentage l'utilisation de la puissance maximale de la puce.

6.6. Conclusions :

Les résultats des estimations serraient mieux confirmés par divers algorithmes vraiment mis en oeuvre dans différentes puces. Concrètement, on n'a fait qu'un seul essai avec l'ADPCM et les résultats de l'estimation sont assez similaires à ceux mesurés sur le DSP TEXAS TMS-320C50.

Comme on a vu pour l'ADPCM, les résultats étendus sont les suivants :

Nombre de cycles : 145487
Temps total : 7274350 ns.
qui représentent : 29,0974 cycles par échantillon
1454,87 ns. par échantillon
Donc, on a besoin de moins de 90702,947846 ns. par échantillon

On a comme conclusion que la puce est à 1% de sa puissance maximale.

Donc, l'algorithme est clairement réalisable[1]. Quel réconfort !

[1]La mise en place de l'outil de simulation utilise des tables pour raison d'efficacité. La réalisation dans le DSP fait tous les calculs directement, pour économiser la mémoire. Cela représente une utilisation supérieure au 1% postulé.

Chapitre 7

SIMULATION : L'outil ACCORDION

Bien qu'il s'agisse de la partie qui attire le plus l'attention, c'est une façon comme une autre de mettre en place et de simuler tous ce qui vient d'être écrit. Malgré que l'application soit encore en phase de tests (ce rapport se réfère à la version 0.5), les fonctionnalités dépassent déjà les prévisions initiales.

Cet outil est une application WINDOWS portant le nom de « Accordion ». Ce nom provient du mixage des trois concepts que l'on trouve dans le programme : la vitesse variable, la compression et, évidemment, le son.

7.1. Objectifs :

Les premiers essais et algorithmes ont été réalisés de manière séparée. La représentation des signaux, l'enregistrement et la reproduction se faisaient avec d'autres logiciels. La vitesse variable était testée dans un autre programme et une autre application traitait du calcul des transformées et de la représentation du sonagramme séparément. Même les formats des fichiers issus des différents programmes étaient différents.

Tout cela fait que la réalisation reste une tâche lourde car les démonstrations et l'utilisation par d'autres (l'ECS ou la société BARTHE) provoquaient beaucoup de problèmes. En conséquence, l'objectif initial de l'outil Accordion était clair : l'intégration de tout les logiciels existant.

Encouragé par l'apparente clarté du Borland C++ 4.0, sa librairie orientée objets ObjectWindows 2.0 et le volume de tous les tomes du manuel qu'il y avait sur la table, j'ai entrepris la tâche de faire une application d'utilisation facile pour WINDOWS. De plus, mes amples connaissances du langage C++ basées sur beaucoup d'années de programmation, de cours et de publications[1], j'e me suis déterminé à faire une véritable application pour WINDOWS plus ambitieuse que celle que le projet demandait.

[1] [HERNA93]

Après cette première étape de démarrage les buts de l'application furent reconsidérés par l'ECS et par la société BARTHE et les objectifs devinerent plus mieux définis.

- Servir d'outil pour l'enregistrement, la reproduction et le maniement des fichiers audio sous différents formats.

- Étudier certains paramètres de l'enregistrement et la reproduction comme la fréquence d'échantillonnage, le nombre de bits de résolution, etc.

- Appliquer aux sons beaucoup d'effets (échos, volume, filtrage, génération d'ondes et bruit, application de formules).

- Extraire l'information utile pour l'étude de la parole (pitch, puissance moyenne, etc.).

- Représenter des signaux de manières diverses : temporelle, spectrale, sonagramme. À part à l'écran, évidemment, les tracés peuvent se rediriger vers l'imprimante ou le presse-papiers de WINDOWS.

- Intégrer tous les travaux réalisés dans ce projet, soit :

    - La compression,
    - La vitesse variable,
    - Les mesures de qualité,
    - Les mesures de complexité.

- Montrer les possibilités d'une application Multimédia pour le traitement de la parole.

Comme on le verra, ces objectifs coïncideront justement avec des caractéristiques fournies par l'outil Accordion. Il s'agit précisément d'ajouter dans un seul outil tous les besoins qu'un utilisateur possible pourrait avoir pour le traitement de la parole.

7.2. Mise en marche :

Avant l'exécution du programme, on doit s'assurer que les moyens matérielles requis sont en place et que l'on connait toutes les instructions pour une installation correcte et un bon fonctionnement. Il faut savoir qu'aucun dommage physique[1] ne peut être porté à l'ordinateur ou à la carte par l'utilisation du logiciel.

7.2.1. Équipement requis :

Étant donné que dans les objectifs de cet outil ne sont pas la commercialisation ni la distribution du produit, l'application a été réalisée pour bien fonctionner sur son site de développement : un Pentium sous WINDOWS avec une carte de son SoundBlaster AWE32.

Mais elle a été réalisée avec un compilateur habituel comme Borland C++ et avec une programmation de haut niveau. Donc, aucune dépendance vis à vis de la machine n'est nécessaire pour son déroulement.

L'ordinateur requis est donc tout ordinateur à base de microprocesseru 80386 ou supérieur ayant des ressources de mémoire et disque dur suffisants. La taille de l'application est d'environ 1,5 Mega-octets et la mémoire utilisée pendant l'exécution dépendra bien sûr de la longueur des fichiers audio et des algorithmes utilisés. En pratique, l'application a été essayé sur un 386 ayant 4 Mega-octets de RAM (mémoire vive) sans trop de problèmes.

En relation avec la carte de son, le programme ne la requiert pas pour l'exécution, la représentation et l'application de tous les algorithmes. Si aucune carte de son est ajoutée à l'ordinateur, au moment de l'exécution, le programme donnera une erreur d'application WINDOWS.

Les appels pour l'enregistrement et la reproduction du son sont ceux de WINDOWS, la carte utilisée n'a qu'un seul pré-requis : avoir un driver compatible avec WINDOWS.

7.2.2. Installation :

Bien que la distribution soit restreinte, le logiciel sera fourni sous forme d'un fichier comprimé avec le format ZIP de DOS / WINDOWS ou peut-être avec un programme appelé « install.bat » qui fera l'installation.

Quelque soit le cas, le résultat de la décompression ou linstallation fera aparaître sur le disque dur des fichiers que lon présentera dans la suite.

7.2.3. Fichiers :

J'ai fait tout mon possible pour avoir un nombre minimal de fichiers pour des raisons d'intégration. Les fichiers et leur place adéquate assurant le fonctionnement du logiciel sont les suivants :

[1] L'influence sur les autres applications qui peuvent s'éxécuter en même temp n'est pas assurée.

- L'exécutable :

  « acco.exe »

  Il doit se trouver dans n'importe quoi répertoire. Comme suggestion on le peut le mettre dans « C:\ACCO »

- Le fichier d'aide :

  « acco.hlp »

  Il doit se trouver dans le même répertoire que le fichier précédent.

- Les librairies .DLL (Dynamic Link Library) requises :

« owl200.dll»	ObjectWindows Library 2.0
« bc40rtl.dll »	Borland C++ 4.0 Run Time Library
« bids40.dll »	Une autre librairie de Borland C++ 4.0

  Si on a installé le compilateur Borland C++ 4.0 dans l'ordinateur, ces fichiers se trouveront déjà dans les répertoires correspondants. Ces libraires sont de distribution libre, comme il est spécifié dans la licence d'utilisation du logiciel Borland C++ 4.0.

- Les fichiers de son : aucun fichier de démonstration n'est fourni mais ils peuvent être recherchés dans n'importequel répertoire lors de l'ouverture par l'outil Accordion lui-même.

Pour une déinstallation de logiciel, on doit enlever les fichiers précédents.

7.2.4. Langue utilisée :

La langue utilisée a été l'anglais. La raison a été donnée dans l'introduction. Les ObjectWindows produisent des menus par défaut qui sont réalisés en cette langue ; les erreurs et beaucoup d'affichages sont aussi en anglais. Il est possible de changer tout cela, mais le travail sérait un peu lourd.

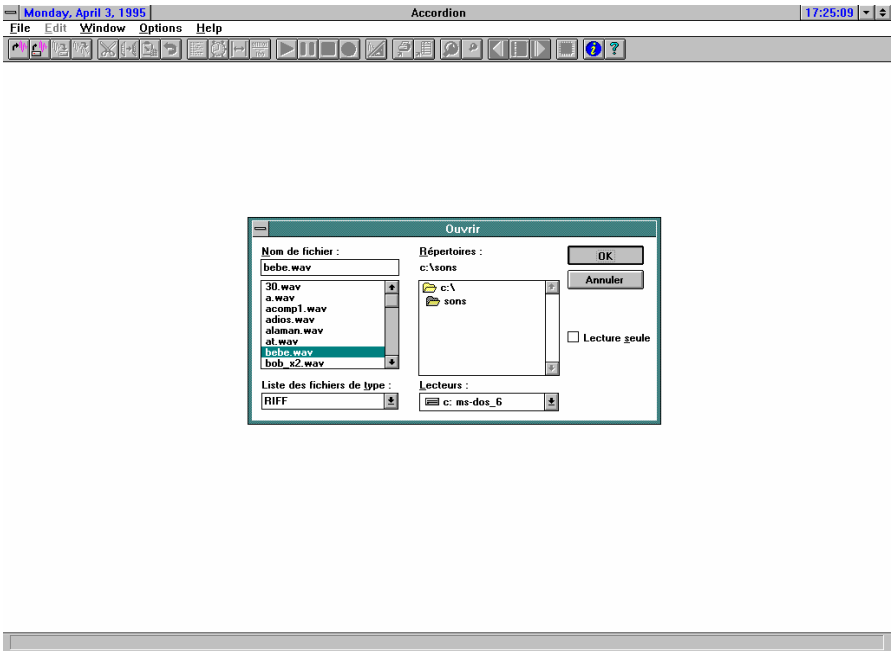
De plus, mes pauvres connaissances de la langue française au commencement ne me permettaient pas de faire une application sans mettre des erreurs d'orthographe. J'ai préféré faire quelque chose dans un anglais correct plutôt que dans un mauvais français.

De toute façon, presque tout le monde est habitué à travailler avec des logiciels en anglais et la terminologie et les expressions utilisées sont assez simples.

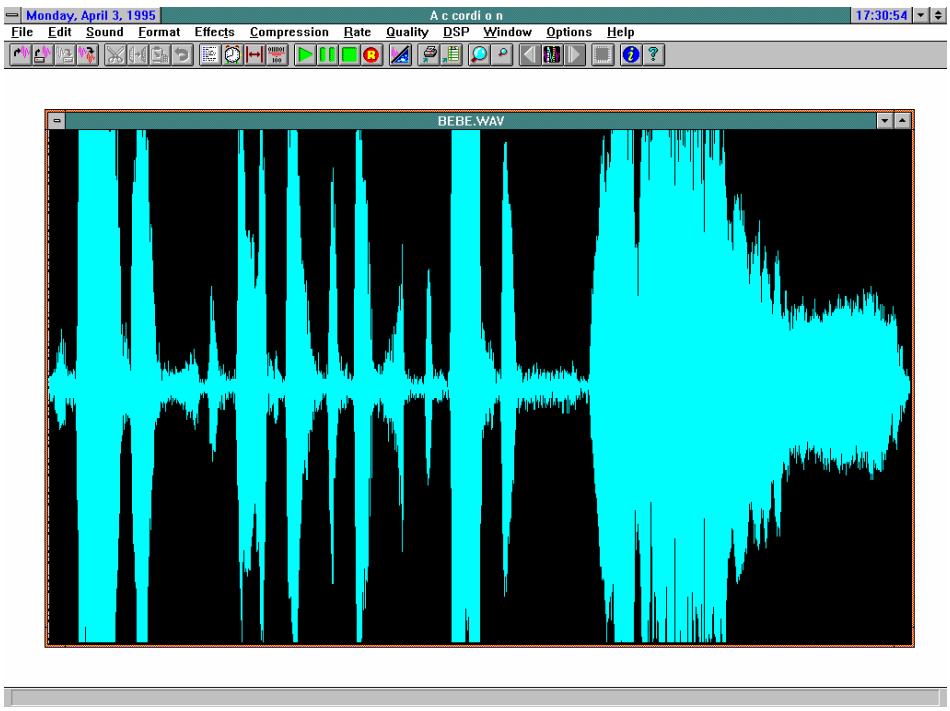
7.3. Fonctionnalités :

Les fonctionnalités sont celles déduites des objectifs fixés. Elles sont toutes visibles dans le menu aisément accessible par le clavier ou la souris ou directement par une barre d'outil dans la partie supérieure de l'application.

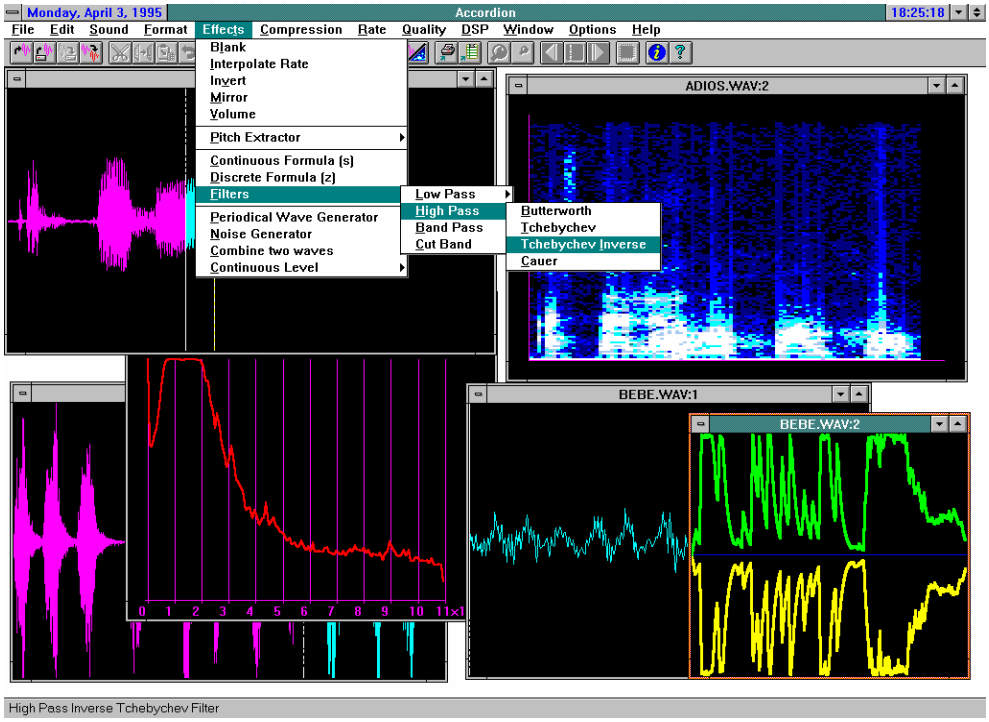
Initialement, l'application peut ressembler à quelque chose comme celle ci-dessous, la plupart des options grisées étant non utilisables : on montre le moment de l'ouverture d'un fichier nommé « bebe.wav » :



La plupart des fonctions n'apparaissent que lors de l'ouverture ou la création d'un fichier audio. On voit maintenant le fichier représenté :



Il s'agit d'une application multidocument, ce qui veut dire que l'on peut avoir beaucoup de fichiers ouverts en même temps dans différentes fenêtres et agir sur le fichier de la fenêtre active. C'est aussi une application multivisualisation, ce qui veut dire que l'on peut avoir différentes vues pour chaque document (temporelle, spectrale, etc.). Par exemple, la figure suivante montre l'état normal lors de l'exécution du programme.



Pour énumérer toutes les fonctionnalités, je vais suivre à peu près l'ordre tracé par le menu :

- **File** : C'est la boîte habituelle d'une application WINDOWS qui permet de créer un nouveau fichier de son, ouvrir un fichier déjà existant, l'enregistrer, le renommer, le recopier sur disque, le fermer ou, sortir de l'application.

- **Edit** : Permet de couper, copier, effacer, dupliquer ou coller tout ou une portion du signal qui est sélectionnable avec la souris, en clickant avec le bouton gauche ou droit. La possibilité d'annuler les opérations précédentes est aussi prévue.

- **Sound** : Pour écouter, faire une pause, arrêter ou enregistrer du son avec le format spécifié par le fichier s'il est fourni par la carte.

- **Format** : Donne la possibilité de changer ou de montrer les paramètres du fichier comme le nombre d'échantillons, le nombre de bits ou la fréquence d'échantillonnage.

- **Effets** : Une des boîtes les plus importantes. On peut faire des effets comme mettre des silences, changer le volume, inverser le signal horizontalement (invert) ou verticalement (mirror). D'autres fonctionnalités plus sophistiquées incluent plusieurs détecteurs du pitch, des générateurs d'ondes carrées, sinusoïdales, en dents de scie ou de

bruit ; elles peremettant de combiner des signaux (addition, multiplication, etc.), de calculer ou d'ajouter un niveau continu ou d'appliquer une fonction ou un filtre algébrique en z ou en s. Même, des filtres de Butterworth, Tchebychev, Tchebychev inverse, Cauer et toutes ses variantes (passe-bande, coupe-bande, passe-bas, etc.) d'ordres différents sont inclus dans cette boîte.

- **Compression** : Une grande partie des algorithmes de compression étudiés dans ce projet est implantée dans le logiciel. On cite la compression des silences, la log-PCM (A-law et μ-Law), différentes versions de ADM, l'ADPCM d'Intel, et quatre compressions au niveau spectral (une avec la DCT et trois avec la FFT).

- **Rate** : Cette boîte fait référence à la vitesse variable. Deux méthodes sont implantées : une temporelle et l'autre fréquentielle (par FFT).

- **Quality** : Deux versions de comparaison sont implantées : l'une temporelle et l'autre des spectres, chaqu'une avec deux possibilités, avec ou sans DTW. Une autre mesure de similarité semblable au SNR est envisagée.

- **DSP** : Prévoit la possibilité de changer le DSP qui est couramment sélectionné pour faire les calculs de complexité de la plupart des algorithmes implantés.

- **Window** : Cette boîte est fournie par la librairie ObjectWindows et a comme mission fondamentale de contrôler l'arrangement des fenêtres des differentes vues et document ouverts et de gérer la visualisation d'une fenêtre vis à vis des zooms et des redirections vers l'imprimante ou le presse-papiers.

- **Options** : Fournit uniquement des options de visualisation du signal et du spectre.

- **Help** : Permet d'appeler l'aide du programme « acco.hlp » ou l'aide générique sur l'aide elle-même. La ligne « about » montre aussi une fenêtre d'information sur l'auteur, le projet et une brève description de l'application Accordion.

D'autres sous-menus apparaissent en option parmis d'autres mais une explication de tous serait interminable. On recommande de naviguer un peu avec la souris et de découvrir peu à peu toutes les choses que l'on peut faire avec cette application.

Même si l'aide n'est pas totalement finie, pour chaque option il y a une barre d'état en partie inférieure de la fenêtre principale qui affiche une petite description de ce qui se produira au moment du choix.

7.4. Mise en place :

Lors de la mise en place, j'ai choisi le système d'exploitation WINDOWS 3.1 pour son utilisation aisée et pour la disponibilité d'un bon compilateur de C++. Le C++ est un langage très approprié pour cette sorte d'applications pour beaucoup de raisons :

- Il constitue un sur-ensemble du C ; la partie concrète des algorithmes peut être développé seulement en C, qui est efficace et transportable sur la plupart des puces actuelles étant donné l'existence de compilateurs C avec presque tous les kits de développement.

- Grâce à la compatibilité avec le C, beaucoup de sources sont disponibles et des algorithmes existent sur le traitement de la parole.

- Comme langage orienté objets, il est très adéquat pour l'environnement WINDOWS et la représentation graphique.

- Il est maintenant le standard de la programmation. En conséquence, les sources des algorithmes peuvent être compris par un programmeur quelconque.

De plus, comme on l'a dit, j'ai utilisé le C++ parce que c'est de loin le langage que je connais mieux.

Pour gérer les messages WINDOWS et tous les menus et fenêtres j'ai choisi la librairie en C++ de Borland nommé ObjectWindows 2.0 au lieu de l'API standard en C de Microsoft.

Elle a été assez compliquée en raison de divers facteurs :

- Le maniement de la mémoire : les fichiers audio, les compressions, décompressions, etc. nécessitent un stockage important en mémoire, des mouvements, des libérations, etc., qui sont dangereuses si on ne prend pas les mesures adéquates.

- La grande quantité de formats audio et des représentations fait que différentes copies du même fichier résident en mémoire (le format originel, le format comprimé, le format pour l'enregistrement ou reproduction et toutes les autres transformées).

- La rigidité des ObjectWindows. Il s'agit d'une librairie qui regroupe toutes les fonctions de WINDOWS pour la programmation objets, mais il y a encore des choses qu'il faut faire directement avec l'API de WINDOWS, ce qui casse tous les schémas d'encapsulation.

- L'information sur les erreurs des ObjectWindows est pauvre. Chaque fois qu'une erreur d'application apparait, un code numérique est fourni sans d'autre information. Le débogueur peut aider un peu mais la recherche des erreurs est une tache très ardue.

- Des manuels peu clairs pour la programmation pour WINDOWS sont fournis par Borland. Il y a beaucoup de choses qui ne sont pas

expliquées dans les manuels ni dans l'aide en ligne. Il faut procéder par essais successifs ; de ceux-ci sont issus quelques résultats inespérés.

Malgré quelques incorrections, manques ou erreurs que j'ai l'intention de résoudre, le programme fonctionne assez bien si on prend en compte le temps passé (1 mois et demi), les personnes employées (une), la complexité (implantation d'algorithmes), la modularité (une cinquantaine de fichiers) et le nombre de lignes de code (30000 - 40000 écrites par moi, environ 100000 avec toutes les librairies) : tout un défi à l'ingénierie logicielle et à la programmation.

De toute façon, ce texte fait référence à la version 0.5. Dans le domaine informatique, une version n'est pas encore valable jusqu'à la version 1.0.

Passons ensuite à la description de quelques uns des points plus importants ou représentatifs de cette implantation.

7.4.1. Formats des fichiers d'audio:

Étant donné les avances récentes à caractère commercial, le grand nombre de cartes, de systèmes, d'ordinateurs et de formes des enregistrements audio, implique que l'immense variété des formats audio.

En général, pour la plupart d'entre eux, les données d'un enregistrement sont déterminées par trois paramètres très liés avec l'échantillonnage. Ce sont

- La fréquence d'échantillonnage (en nombre d'échantillons par second),
- Le nombre de bits par échantillon,
- Le nombre de canaux.

Historiquement, la plupart des machines utilisaient leur propre format pour les données, quelques formats étant applicables plus généralement. Maintenant, il est possible de faire des conversions entre presque quelque tous les formats, mais quelquefois avec une perte d'information.

Quelquefois, la fréquence du fichiers et celle de la carte ne sont pas exactament pareilles. Comme on l'a dit dans le premier chapitre, puisque nous traitons de la parole, il n'y a pas de problème s'il subsiste de petites différences d'environ le 1-2% entre le son enregistré et le taux à laquelle est joué.

7.4.1.1. Classification :

Il y a deux sortes de formats : les formats qui se décrivent eux-mêmes et les formats qui ne portent aucune information sur leurs paramètres. Pour éclaircir, on les appellera formats avec et sans en-tête.

- ***Formats sans en-tête*** : Normalement, ils défissent de codes très simples (PCM, A-Law, μ-Law). C'est l'extension du fichier ou l'utilisateur qui doit deviner les paramètres. Si on écoute le son très lentement, on augmente la fréquence d'échantillonnage, etc. Normalement l'extension du ficher est « .RAW » (pour « cru » en anglais) et consiste

habituellement en une série d'échantillons purs et durs. La manière la plus normale de stocker cette information est séquentielle, en intercalant les canaux pour chaque échantillon.

On recommande de commencer par 8 bits monocanal et 11025 hz d'échantillonnage, et varier avec 16 bits, stéréo, à 22050 hz, 8000 hz ou 44100 hz.

- ***Formats avec en-tête*** : Un petit en-tête de données donne une description du format et de la compression employés pour les données. Quelques uns contienent aussi des chaînes avec du texte, la compagnie, le nom du format et beaucoup d'autres informations. Les formats les plus importants sont illustrés dans le tableau suivant :

Format	Extension	Origine	Paramètres fixes	Paramètres variables
SOUND	.au ou .snd	NeXT, Sun		Fréquence, n° canaux, compression, info
AIFF	.aif	Apple, SGI		Fréquence, n° canaux, info
AIFC	.aif	Apple, SGI		Fréquence, n° canaux, compression, info
IFF/8SVX	.iff	AMIGA	8 bits	Fréquence, n° canaux, instruments
VOC	.voc	SB / Creative Labs		Fréquence, n° canaux, compression, info
RIFF/WAVE	.wav	Microsoft		Fréquence, n° canaux, compression, beaucopu d'info
	.sf	IRCAM		Fréquence, n° canaux, compression, info
HCOM		Mac	8 bits 1canal Compression Huffman	Fréquence
NIST		DARPA		
	.mod ou .nst	AMIGA		
MIME		Internet		

Une tendance actuelle tend à standardiser les taux d'échantillonnage aux fréquences suivants:

- 8000 hz, 8 bits μ-Law monocanal pour la téléphonie et la compréhensibilité.
- 22050 hz, 8 bits sans signe, linéaire, monocanal ou stéréo pour une bonne qualité.
- 44100 hz, 16 bits avec signe, linéaire, monocanal ou stéréo pour une qualité professionnelle.

Mais les avances en compression ont en tendance à laisser les formats encore plus libres qu'auparavant.

On va présenter les plus courants ou ceux qui sont implantés dans le programme.[1]

7.4.1.2. Le format RIFF (.WAV) :

RIFF est un format de Microsoft et IBM et c'est le plus utilisé sous WINDOWS. Ceci le plus important à étudier. Il est très semblable au format AIFF mais l'ordre des octets de poids forts est l'inverse, et d'autres détails.

Le format RIFF se base sur des unités nommées « chunks » dont chacune contient quatre octets pour l'identification du chunk, quatre pour sa longueur et les données correspondantes au chunk. Les données peuvent être égalemente un autre chunk ; ceci donne la possibilité de faire des structures très complexes.

Maintenant les fichiers .WAV sont des fichiers multimédia qui peuvent contenir du son, de l'hypertexte, des films, des photos, ou toute autre chose.

Le chunk qui nous intéresse réellement est le chunk « WAVE », qui déclare les données comme audio.

Commenter tous les formats d'un fichier .WAV serait excessif pour ce rapport. Une implantation réduite a été mise dans l'outil Accordion qui n'accepte que les fichiers monocanaux avec une fréquence d'échantillonnage et un nombre de bits quelconques.

Pour une information plus détaillée sur ce format on peut regarder le fichier RTF (Rich Text Format) « RIFFMCI.RTF » qui contient la « Multimedia Programming Interface and Data Specification v1.0 ». Les pages les plus importantes sont les numéros 58-64.

Chaque nouveau format qui veut être inclus dans la norme WAV doit être sommis à Microsoft. Dans les dernières années, beaucoup de fabriquants (même Creative Labs) ont accepté ce format. Les formats les plus courants sont actuellement codifiés dans les champs wFormatTags de la structure WAVE comme suit:

WAVE_FORMAT_UNKNOWN	0x0000
WAVE_FORMAT_PCM	0x0001
WAVE_FORMAT_ADPCM[2]	0x0002
WAVE_FORMAT_IBM_CVSD	0x0005
WAVE_FORMAT_ALAW	0x0006
WAVE_FORMAT_MULAW	0x0007
WAVE_FORMAT_OKI_ADPCM	0x0010
WAVE_FORMAT_DVI_ADPCM[3]	0x0011
WAVE_FORMAT_DIGISTD	0x0015
WAVE_FORMAT_DIGIFIX	0x0016
WAVE_FORMAT_YAMAHA_ADPCM	0x0020

[1]Pour une information complète sur les formats d'audio il y a une FAQ (Frequently Asked Questions) nommé « Audio_File_Formats » dans les groupes de nouvelles « alt.binaries.sounds.misc », « alt.binaries.sounds.d », « comp.dsp », « alt.answers », « comp.answers » ou « news.answers » de l'Internet.

[2]Un fichier source d'échantillon « msadpcm.c » peut se demander à Microsoft Corporation. Multimedia Product Management. One Microsoft Way. Redmond, WA 98052-6399

[3]C'est la version d'ADPCM dévéloppée dans le programme Accordion.

WAVE_FORMAT_FORMAT_SONARC	0x0021
WAVE_FORMAT_DSPGROUP_TRUESPEECH	0x0022
WAVE_FORMAT_FORMAT_ECHOSC1	0x0023
WAVE_FORMAT_AUDIOFILE_AF36	0x0024
WAVE_FORMAT_FORMAT_APTX	0x0025
WAVE_FORMAT_AUDIOFILE_AF10	0x0026
WAVE_FORMAT_FORMAT_DOLBY_AC2	0x0030
WAVE_FORMAT_CREATIVE_ADPCM	0x0200

Quelques uns des formats précédents décrivent des compressions très bonnes, quelquefois impressionnantes comme la AF36 d'AudioFile qui comprime à un taux de 36:1. Tous ces formats sont expliqués, ou au moins une référence est donnée, dans « Multimedia Standards Update : New Multimedia Data Types and Data Techniques » qui est disponible dans beaucoup de lieux sur l'Internet. Il faut vraiment en rechercher sur leurs sujet ou demander des informations pour être à jour des dernières avancées en compression audio.

Dans l'application Accordion on supporte l'ouverture, la création et l'enregistrement du format 0x0001. C'est-à-dire, aucune compression.

7.4.1.3. Le format Creative Voice (.VOC) :

Étant donné que Creative Voice est le fabriquant le plus connu de cartes de son pour PCs, ses propres formats prennent une importance supérieure aux autres. De plus, la carte avec laquelle j'ai travaillé est la SoundBlaster AWE32 de Creative Labs.

Le format vient originellement de la première carte SoundBlaster. Il fournit un en-tête de 25 octets avec les champs suivants :

00-18	La chaîne : « Creative Voice File »
19	La valeur 0x1A pour arrêter la lecture du fichier.
20-21	Décalage du premier bloc de données (Normalement 0x001A)
22-23	Version (VOC-HDR est 1.10 : 0x010A)
24-25	Complément a deux de la version précédente + 0x1234

Chaque bloc a :

00	Type des donnés.
01-03	Longueur.
04	Données.

La valeur un type permet de mettre dans les données la fréquence, le nombre de bits, les canaux ou même les données elles-mêmes.

Les cartes SoundBlaster utilisent des fréquences diviseurs de 1000000 Hz. Les taux d'échantillonnage de la SB AWE 32 sont programmables entre 5 kHz et 45 kHz en 228 pas linéaires. C'est-à-dire:

$$x= (45000 - 5000) / 228$$

$$x= 175,439 \text{ hz à peu près chaque réglage.}$$

Ce qui donne les fréquences suivantes :

(5000 5175,439 5350,878 ... 45000)

Finalement, pour des raisons de compatibilité avec toutes les autres cartes, ce format n'a pas été implanté. De plus, des convertisseurs pour les convertir au format .WAV sont disponibles « shareware » sur Internet.

7.4.2. L'enregistrement et la sortie Audio :

Initialement, je recherchais des informations particulières sur la carte SoundBlaster AWE 32 pour essayer de programmer le DSP qu'elle contient. Comme on le verra dans les annexes, aucune réponse n'a été reçue à ma demande d'information à la succursale de Creative Labs en France.

À la fin, j'ai trouvé assez d'information sur Internet sur les versions précédentes de la SoundBlaster mais, quoique la carte AWE32 soit compatible avec ses précédentes, utiliser cette programmation directe n'exploitait pas toutes les fonctionnalité de la nouvelle carte. Pour ces raisons, j'ai finalement opté pour faire une programmation indépendante de la carte, avec les appels courants de l'API de WINDOWS.

Maintenant, je me réjouis que Creative Labs ne m'ai pas répondu ; le résultat est que l'application Accordion est valide pour toute carte compatible avec WINDOWS et donc, pratiquement, avec toute carte pour PC.

Les appels audio de l'API de WINDOWS utilisent le format WAVE en interne, avec une structure nommée WAVEFORMAT qui est un chunk comme on l'a commenté auparavant.

Les fonctions fournies pour les signaux sont les suivantes (pour des fichiers MIDI il y en a d'autres) :

```
waveInAddBuffer          Send buffer to waveform input device
waveInClose              Close waveform input device
waveInGetDevCaps         Get waveform input device capabilities
waveInGetErrorText       Get text description for waveform error
waveInGetID              Get waveform input device ID for handle
waveInGetNumDevs         Return number of waveform input devices
waveInGetPosition        Get current waveform device input position
waveInMessage            Send messageto waveform input device
waveInOpen               Open waveform input device for recording
waveInPrepareHeader      Prepare buffer for waveform input
waveInReset              Stops waveform input device
waveInStart              Start waveform input device
waveInStop               Stop waveform input
waveInUnprepareHeader    Clean up prepared waveform header
waveOutBreakLoop         Break waveform output loop
waveOutClose             Close waveform output device
waveOutGetDevCaps        Get waveform output device capabilities
waveOutGetErrorText      Get text description for waveform error
waveOutGetID             Get waveform output device ID for handle
waveOutGetNumDevs        Get number of waveform output devices
waveOutGetPitch          Get current waveform output pitch
waveOutGetPlaybackRate   Get current waveform playback rate
waveOutGetPosition       Get current waveform playback position
waveOutGetVolume         Get current waveform volume
waveOutMessage           Send message to waveform output drivers
waveOutOpen              Open waveform output device
waveOutPause             Pause waveform playback
waveOutPrepareHeader     Prepare waveform playback data block
waveOutReset             Stop waveform playback
```

```
waveOutRestart           Restart paused waveform playback
waveOutSetPitch          Set waveform output pitch
waveOutSetPlaybackRate   Set waveform playback rate
waveOutSetVolume         Set waveform output volume
waveOutUnprepareHeader   Clean up prepared waveform data block
waveOutWrite             Write to waveform output device
WaveProc                 Waveform device callback function
```

L'information sur toutes ces fonctions peut se trouver dans l'aide Microsoft WIN16/WIN32 API.

Par exemple, une partie de l'écran de l'aide est montrée ci-dessous pour la structure WAVEFORMAT :

```
typedef struct waveformat_tag {
    WORD  wFormatTag;
    WORD  nChannels;
    DWORD  nSamplesPerSec;
    DWORD  nAvgBytesPerSec;
    WORD  nBlockAlign;
} WAVEFORMAT;

The WAVEFORMAT structure describes the format of waveform data. Only format
information common to all waveform data formats is included in this
structure. For formats that require additional information, this structure
is included as a member in another structure, along with the additional
information.

Member            Description

wFormatTag        Specifies the format type. The currently defined waveform
format
                  type is WAVE_FORMAT_PCM, which specifies that the waveform
data
                  is pulse code modulated (PCM).

nChannels         Specifies the number of channels in the waveform data.
Monaural
                  data uses one channel and stereo data uses two channels.

nSamplesPerSec    Specifies the sample rate, in samples per second.

nAvgBytesPerSec   Specifies the required average data-transfer rate, in bytes
per
                  second.

nBlockAlign       Specifies the block alignment, in bytes. The block
alignment is
                  the minimum atomic unit of data.
```

Pour le reste des fonctions et structures, une information complète est fournie (en anglais).

Une grande partie des fonctions présentées a été donc utilisée pour les opérations que réalise l'application Accordion (PLAY, RECORD, PAUSE et STOP). Ces quatre opérations sont quatre membres de la classe TSoundDocument que l'on peut trouver dans le fichier « sounddoc.cpp » :

```
void TSoundDocument::Play();

void TSoundDocument::Record();

void TSoundDocument::Stop();

void TSoundDocument::Pause();
```

Pour des détails additionnels on recommande de voir l'aide de l'API ou les sources de mon programme.

7.4.3. L'interface avec l'utilisateur (les ObjectWindows 2.0) :

Pour l'interface avec l'utilisateur j'ai cru convenable d'utiliser les Borland ObjectWindows 2.0 parce qu'il semble que c'est une manière intégrée et orientée objets de faire des applications WINDOWS en langage C++ avec un effort minimal. Ensuite je montre une brève description des ObjectWindows et de l'interface Document/Visualisation utilisée.[1]

ObjectWindows 2.0 constitue l'architecture d'applications Borland C++ pour les environnements WINDOWS 3.1, Win32S et WINDOWS NT. Il permet de créer rapidement et facilement des applications WINDOWS complètes, avec des fonctionnalités telles que menus, boîtes de dialogue, barres de contrôles graphiques, barres d'état, fenêtres MDI, et plus encore.

ObjectWindows propose une nouvelle façon de contenir et de manipuler des données : le modèle Document / Visualisation. Ce modèle se compose de trois parties :

- Des objets documents, qui peuvent contenir de nombreux types de données différentes et fournir des méthodes permettant d'accéder à ces données. Dans notre cas, les données sont les échantillons et le format de notre ficher audio.

- Des objets visualisation, qui constituent une interface entre un objet document et l'utilisateur, et contrôlent l'affichage des données et la façon dont l'utilisateur peut dialoguer avec les données. Dans Accordion, il s'agit de toutes les représentations possibles : temporelle, spectrale, sonagramme, etc.

- Un gestionnaire de documents portant sur toute l'application qui met à jour et coordonne des objets documents et les objets visualisations correspondants.

Tout ceci est intégré en concordance avec les menus, permettant d'ouvrir, de créer et de fermer des fichiers et de gérer la gestion des différentes fenêtres dans la fenêtre principale.

Quelquefois, l'interface n'est pas aussi aisée qu'on peut l'espérer d'une application WINDOWS : ceci est dû au fait que beaucoup de versions originales des algorithmes ont été développées initialement pour DOS, pour UNIX, ou pour des applications restreintes comme les EasyWin.

Mais la plupart des actions se réalisent avec des menus et la souris tout comme les applications commerciales de WINDOWS.

7.4.4. La représentation interne des échantillons :

Étant donné la grande quantité de formats différents et, surtout, le nombre variable de bits (8, 16 ou même 12 bits), l'application des algorithmes devra s'adapter à chaque variation du format. Ceci rend les algorithmes plus complexes que ce qu'ils sont en réalité.

[1]Cette information est extraite des manuels de Borland ObjectWindows 2.0

Pour avoir une série d'échantillons d'une façon uniforme, on a fait une classe C++ nommé ***sample_array*** qui se trouve dans les fichiers « sample.h » et « sample.cpp ». Les échantillons sont stockés selon le format interne qu'ils avaient pour économiser la mémoire, mais lors de l'utilisation, chaque échantillon est converti en un type nommé ***sample***, qui est celui avec lequel travaille.

La caractéristique fondamentale de cette valeur ***sample***, est qu'elle est toujours compris entre -1 et 1, indépendamment de la résolution en bits du format original. Ceci permet de faire des algorithmes très indépendants du format, avec tous les détecteurs de pitch, algorithmes de compression, représentations sur l'écran, effets divers et même rend la conversion entre formats plus transparente.

Comme inconvénient fondamental, on peut citer les opérations qui sont maintenant réalisées sur des nombres réels en virgule flottante avec double résolution entre -1.0 et 1.0. Cela donne l'apparence que les algorithmes de compression et de comparaison sont très inefficaces et peut décourager un peu avec les transformées et quelques représentations comme le sonagramme, qui, pour des fichiers longs, peut être désespérante...

7.4.5. La gestion de la mémoire :

C'est un aspect délicat dû au fait que les fichiers de son sont souvent très longs. Les limites héritées du DOS de blocs de 64K sont de toute façon ridicules et même l'utilisation de pointeur ***far*** (1 Mega-octet) pour l'adressage est insuffisante. Ceci ne laisse qu'une option : l'utilisation de pointeurs ***huge*** qui n'ont pas de restrictions. Mais, malheureusement, il n'y a aucune fonction standard de C (comme malloc, farmalloc, free, farfree, etc.) ou de C++ (new et delete) qui utilisent les pointeurs ***huge***.

Pour cette raison on a utilisé l'API de WINDOWS qui contient les fonctions suivantes :

```
CopyMemory              Copies a block of memory
FillMemory              Fills a block of memory with a spec'd value
GetProcessHeap          Obtains handle to the calling process' heap
GlobalAlloc             Allocates memory from the heap
GlobalDiscard           Discards a global memory block
GlobalFlags             Returns global memory block info
GlobalFree              Frees a global memory block
GlobalHandle            Translates a global pointer to a handle
GlobalLock              Locks global memory object and returns a pointer
GlobalMemoryStatus      Checks memory availability
GlobalReAlloc           Changes global memory block size/attributes
GlobalSize              Returns the size of a global memory block
GlobalUnlock            Unlocks a global memory block
```

Ces fonctions sont expliquées aussi dans le fichier d'aide d'API de WINDOWS et ne sont pas très différentes de ses correspondantes standard.

Probablement, la gestion de la mémoire doit, avec les messages internes de WINDOWS, être la raison de la plupart des erreurs d'application de l'outil Accordion. Ce n'est pas une tâche facile, étant donné que pour chaque fichier on fait beaucoup de transformations, compressions et d'effets divers, et on a même différentes copies du même signal pour des différents formats ou représentation. Tout cela fait que pour suivre l'état de l'allocation et désallocation de mémoire on doit faire de vrais efforts de patience.

Quoique l'on puisse fournir une justification, j'ai un peu honte chaque fois qu'il apparaît une erreur d'application. On pourrait supposer que, par ma condition d'informaticien, ces erreurs devraient avoir être enlevées mais cela demande beaucoup de temps. Je peux assurer que ce sera la première chose que je ferai quand je reprendrai le programme.

7.4.6. Structure Modulaire :

Comme on vient de le dire, il y a presque une cinquantaine de fichiers pour former l'application, à part les standards de C et C++ et ceux qui viennent des ObjectWindows et de la librairie API standard de WINDOWS.

Illustrer la hiérarchisation de tous les modules donnerait peut-être une grande confusion. Je vais essayer de montrer les modules fondamentaux de l'application, leur longueur en Koctets approchée et une petite description d'entre eux :

Fichiers C++ :

Fichier	Longueur	Description
acco.cpp	(< 1K)	Fichier principal de l'application.
adm.cpp	(12K)	Algorithmes ADM
adm.h	(< 1K)	Fichier en-tête (déclarations) du précédent
adpcm.cpp	(15K)	Algorithme ADPCM de Intel
adpcm.h	(< 1K)	Fichier en-tête (déclarations) du précédent
bitarray.cpp	(5K)	Classe pour gérer l'access bit à bit à la mémoire
bitarray.h	(< 1K)	Fichier en-tête (déclarations) du précédent
cost.cpp	(2K)	Classe pour calculer le coût operationnel (complexité)
cost.h	(2K)	Fichier en-tête (déclarations) du précédent
dct.cpp	(8K)	Algorithmes de compression et transformée DCT
dct.h	(< 1K)	Fichier en-tête (déclarations) du précédent
fft.cpp	(6K)	Algorithmes de compresssion et transformée FFT
fft.h	(2)	Fichier en-tête (déclarations) du précédent
fft2.cpp	(7 K)	Une autre manière de calculer la FFT
fft2.h	(< 1K)	Fichier en-tête (déclarations) du précédent
fftcomp.cpp	(5 K)	Algorithme de comparaison spectrale de sons.
fftcomp.h	(< 1K)	Fichier en-tête (déclarations) du précédent
logpcm.cpp	(3K)	Algorithme LOGPCM
logpcm.h	(< 1K)	Fichier en-tête (déclarations) du précédent
mdiframe.cpp	(3K)	Fichier pour gérer la fenêtre principale de l'application
mdiframe.h	(< 1K)	Fichier en-tête (déclarations) du précédent
myapp.cpp	(12K)	Fichier pour gérer l'application principale
myapp.h	(3K)	Fichier en-tête (déclarations) du précédent
newcoder.cpp	(< 1K)	Fichier pour ajouter des nouveaux algorithmes
newcoder.h	(< 1K)	Fichier en-tête (déclarations) du précédent
options.cpp	(< 1K)	Fichier avec les options principales du programme
options.h	(< 1K)	Fichier en-tête (déclarations) du précédent
pitch.cpp	(1K)	Algorithme détecteur du pitch
pitch.h	(< 1K)	Fichier en-tête (déclarations) du précédent
polynom.cpp	(10K)	Classe pour appliquer les filtres et formules en z et en s
polynom.h	(3K)	Fichier en-tête (déclarations) du précédent
printing.cpp	(1K)	Fichier pour gérer l'imprimante
printing.h	(<1K)	Fichier en-tête (déclarations) du précédent

rate.cpp	(16K)	Algorithme de vitesse variable temporelle
rate.h	(6K)	Fichier en-tête (déclarations) du précédent
riff.cpp	(2K)	Classe pour gérer le format RIFF.
riff.h	(2K)	Fichier en-tête (déclarations) du précédent
sample.cpp	(32K)	Classe pour gérer les échantillons et la mémoire
sample.h	(4K)	Fichier en-tête (déclarations) du précédent
silence.cpp	(6K)	Algorithme de compression des silences
silence.h	(< 1K)	Fichier en-tête (déclarations) du précédent
sounddoc.cpp	(26K)	Fichier pour gérer des documents sonnores
sounddoc.h	(3K)	Fichier en-tête (déclarations) du précédent
spectre.cpp	(4K)	Fichier pour calculer le spectre
spectre.h	(< 1K)	Fichier en-tête (déclarations) du précédent
utildef.cpp	(< 1K)	Fonctions et classes utiles pour tout le programme
utildef.h	(1K)	Fichier en-tête (déclarations) du précédent
u_a_law.cpp	(2K)	Algorithmes A-Law et μ-Law
u_a_law.h	(< 1K)	Fichier en-tête (déclarations) du précédent
views.cpp	(75K)	Fichier pour gérer les sortes de représentation des signaux
views.h	(6K)	Fichier en-tête (déclarations) du précédent

D'autres Fichiers :

Fichier	Longueur	Description
acco.ide	(165K)	Projet Borland C++ de l'application et de l'environnement IDE du compilateur de Borland.
acco.def	(<1K)	Options de compilation
acco.rc	(66K)	Fichier de resources WINDOWS pour l'application Accordion
acco.rh	(4K)	Fichier en-tête (déclarations) du précédent
acco.hpj	(< 1K)	Projet de l'aide de Accordion
accohelp.rtf	(8K)	Fichier RTF (Rich Text Format) de l'aide principale de Accordion
keys.rtf	(18K)	Fichier RTF (Rich Text Format) de l'aide sur les touches de Accordion
toolbar.rtf	(6K)	Fichier RTF (Rich Text Format) de l'aide de la bar d'outils d'Accordion
usehelp.rtf	(26K)	Fichier RTF (Rich Text Format) de comment utiliser l'aide de Accordion

Peut-être cela montre l'impossibilité de donner en un espace réduit l'explication détaillée des sources de l'application. La meilleure idée est de regarder le code peu à peu et de lire les commentaires qui sont affichés dans la plupart des fichiers.

Comme j'y ferai allusion dans le paragraphe 7.5.2 j'ai l'intention de restructurer tout le programme et le commenter plus exhaustivement.

7.5. L'aide en ligne :

Normalement, les programmes commerciaux distribuent des manuels ou des guides de l'utilisateur avec le logiciel. Trois raisons font que ceci ne sera pas le cas : ce logiciel n'est pas d'un programme commercial, il est assez facile à utiliser et une petite aide en ligne est déjà ajoutée.

Néanmoins, j'ai développé un fichier d'aide en ligne (« acco.hlp ») avec le format d'aide WINDOWS. Il peut donc être appelé par l'application ou en dehors de celle-ci et il sera bien sûr la source la plus importante pour apprendre à utiliser le programme. La langue utilisée est l'anglais et il reste encore beaucoup de choses à expliquer et à commenter, mais une grande partie de l'aide peut être extraite de ce rapport.

De plus, chaque option des boîtes du menu de l'application affiche une petite information sur une barre située en la partie inférieure de la fenêtre principale.

En définitive, je ne crois pas qu'il faille trop d'explication pour utiliser les programme étant donnée qui il s'agit d'une application courante pour WINDOWS. En ce qui concerne l'implantation et les algorithmes, c'est ce rapport qui doit répondre aux questions.
à l'implantation et aux algorithmes, c'est ce rapport qui doit répondre à ces nécessités.

7.6. Comment ajouter un nouvel algorithme :

Ajouter un nouvel algorithme de compression n'est pas trop compliqué ; normalement il s'agit de deux fonctions de compression et de décompression. Voyons l'exemple de l'ADPCM :

```
struct adpcm state {
    short valprev;       /* Previous output value */
    char  index;         /* Index into stepsize table */
};

void adpcm_coder(sample_array *, long, BYTE huge *, long, struct adpcm_state
*, int Bits, cost & COST);
void  adpcm decoder(BYTE  huge  *,  sample array  *,  long,  long,  struct
adpcm_state *, int Bits, cost & COST);
```

On peut observer que pour le coder il faut un pointeur d'un tableau d'échantillons qui constituent le signal original, un pointeur « huge » à « BYTE » pour les donnés comprimées, un « long » qui détermine le décalage sur le tableau d'échantillons, une structure spécifique à l'algorithme concret, une valeur entière qui, dans ce cas, représente le facteur de compression (numéro de bits) et une référence à un paramètre pour calculer le coût de l'algorithme.

Le problème fondamental est de l'ajouter cet algorithme à la structure des menus, pour l'exécuter avec une touche ou un click de la souris. Pour résoudre ce problème, j'ai développé un algorithme vide dans le menu, appelé new_coder et new_decoder dans les fichiers « NEWCODER.CPP » et « NEWCODER.H ».

Une information additionnelle de mise en oeuvre exacte se trouve dans ces deux fichiers.

7.7. Versions et améliorations futures :

La version qui fait référence ce rapport est la version 0.5. Les versions précédentes sont encore moins inachevées. L'évolution simplifiée de l'application fut :

Ver.	DATE	Description
0.1	**2-2-95**	**Création de l'application. Ouverture, création et représentation très simple de fichiers .RAW**
0.2	**15-2-95**	**Compatibilité avec les fichiers .WAV, ajout de la représentation du sonagramme. On permet de changer les paramètres des fichiers. On peut écouter ou enregistrer les sons. Quelques effets comme le miroir, l'inversement, les silences sont inclus.**
0.3	**15-3-95**	**Ajout des mesures de qualité et de complexité. Algorithmes ADPCM, DCT, ADM et FFT. La vitesse variable spectrale est essayée.**
0.4	**23-3-95**	**Ajoute de l'aide et de la vitesse variable temporelle, la compression des silences, le A-Law et le μ-Law.**
0.5	**31-3-95**	**Application d'une formule en s ou z. Filtres. Détecteur de pitch et de continu. Générateur d'ondes et de bruit.**

Malheureusement, le temps a été trop réduit pour accomplir toutes les tâches que j'aurais désiré. De même, il y encore de nombreuses erreurs qui restent et qui apparaissent sans raison apparente. De plus, comme on vient de le dire, une restructuration du programme serait nécessaire avant de continuer à le développer.

Les étapes sont précisément celles-ci : Premièrement, éliminer les erreurs ou bugs qui existent encore dans l'application ; Deuxièmement, restructurer et recommenter le programme ; Troisièmement, ajouter de nouvelles fonctionnalités.

7.7.1. Corrections et Patches :

Heureusement il ne s'agit pas d'une application commerciale, sinon les lettres de plainte pourraient inonder l'ECS. Elle donne beaucoup d'erreurs mais l'intention de ce projet était d'essayer le nombre le plus grand possible de choses. Je n'avais pas de temps à perdre en rechercher des erreurs quand il y avait d'autres choses plus importantes à faire et quand elle n'apparaissaient que sporadiquement.

Les corrections suivantes doivent être réalisées :

- La copie de sonagramme sur le presse-papiers ne donne pas de bonnes couleurs.
- Eliminer des erreurs d'application d'allocation de mémoire.
- Corriger la barre de % qui ne marche pas correctement et inclure l'icône d'attente dans tous les processus qui sont longs.
- Détecter l'instabilité des filtres.
- Corriger quelques affichages.

Les plus importantes sont, sans doute, la correction des erreurs qui arrêtent l'application pour la mauvaise impression qu'elles donnent du logiciel.

7.7.2. Restructuration des sources :

L'amateur de génie logiciel pourrait se scandaliser de l'apparence finale des sources. Bien qu'il y ait des commentaires, les listings ne sont pas uniformes, il n'y a pas de en-têtes de présentation, quelques fichiers sont trop longs, beaucoup de choses sont mélangées et quelques actions ou sources sont répétées pour la simple raison de ne pas avoir à retrouver les ressources communes.

Une restructuration du programme pourrait aussi enlever quelqu'unes des erreurs précédentes et laisserait des sources plus lisibles et utilisables par d'autres chercheurs de l'ECS ou de l'extérieur.

Les tâches suivantes seront les plus importantes à réaliser :

- Redistribution de tous les modules du programme,
- Recommenter les sources et ajouter les en-têtes de documentation à tous les fichiers,
- Intégrer tous les algorithmes en modules C++ pour leur homogénéisation,
- Réaliser une classe pour la gestion de la mémoire,
- Restructurer la sortie graphique.

On ne peut pas se passer d'appliquer ces mesures si on veut continuer à développer le programme pour ajouter de nouvelles fonctionnalités...

7.7.3. Nouvelles fonctionnalités :

Quoiqu'on puisse aller jusqu'à l'infini avec des opérations que l'on désirerait voir ce logiciel faire, on peut citer les fonctionnalités ou améliorations les plus importantes pour le complérer.

- Inclure de nouveaux formats de son,
- Insérer de nouveaux algorithmes de compression comme SBC ou CELP,
- Insérer des tests à toutes les entrées au clavier pour éviter des paramètres incorrects,
- Finir le générateur d'ondes (onde carrée, triangulaire, sinusoïdale, niveau de continu, bruit blanc, rose et marron),
- Accélérer certaines opérations de visualisation,
- Achever des fonctions de couper, copier et coller des signaux,

- Insérer de nouveaux DSPs ou puces à évaluer et donner la possibilité de modifier leurs paramètres,
- Additionner, multiplier, diviser les signaux,
- Rajouter d'autres effets : échos, atténuations, etc.,
- Finir l'aide,
- Faire que les compressions soient réellement effectives dans les fichiers comprimés au format propre du programme Accordion,
- Mettre à disposition toutes les options et variations développées sur la vitesse variable,
- Faire un « toolbox » pour les touches de PLAY, PAUSE, STOP et RECORD,
- Permettre des opérations « batch » avec calcul cumulé du nombre de cycles (MIPS),
- Ajouter de nouveaux détecteurs du pitch et de formants,
- Traduire en d'autres langues,
- Mettre un curseur pour montrer l'évolution temporelle au moment du PLAY et du RECORD, étant donné que la détection du début est maintenant compliquée,
- Améliorer et mettre les unités dans les représentations.

Bref, la liste serait interminable. Jusqu'au jour où tout cela sera développé, on peut, heureusement, utiliser d'autres programmes qui existent ou qui vont apparaître pour le traitement des fichiers de son.

7.8. D'autres logiciels :

À part l'outil Accordion, d'autres logiciels ont été et peuvent être utilisés pour continuer des recherches sur le traitement de la parole dans le laboratoire. À cet égard, les logiciels suivants ont été achetés ou obtenus librement par l'ECS et sont maintenant installés sur les ordinateurs du laboratoire :

- **SoundBlaster WaveStudio v.2.0** : C'est le programme fourni avec la carte de son SoundBlaster AWE32. Il permet de faire toutes les choses nécessaires pour un fichier de son. Malheureusement, la table de mixage fait planter les WINDOWS. D'autres logiciels sont fournis avec la carte mais les applications sont très spécifiques. Peut-être, un reconnaisseur de voix et un lecteur sont des applications plus étonnantes.

- **Goldwave** : En raison des erreurs du programme précédent, ce programme est une autre application WINDOWS qui fournit presque les mêmes résultats tout en étant plus robuste. De plus, il est gratuit et indépendant de la carte de son.

- **Cool** : C'est une application pour WINDOWS obtenu via Internet qui, si elle n'est pas enregistréé (en payant), ne laisse utiliser qu'une partie de ses fonctionnalités chaque fois qui elle est installée. Mais elle en vaut la peine parce qu'elle permet de faire une grande quantité de choses comme beaucoup d'effets, d'amplifications, de mixages, de FFTs, de filtrage, d'analyse fréquentielle, voire modifier le ton et le bruit, jouer le CD... Beaucoup de formats et de compressions sont acceptés et un fichier d'aide montre comment l'utiliser. De tous les programmes évalués, il est le plus complet.

- **Bmaster** : Obtenu aussi via Internet, il s'agit d'une application pour DOS qui fournit quelques résultats intéressantes. Par exemple, elle permet d'appliquer une version rudimentaire de la vitesse variable et d'autres effets sur la parole.

- **Matlab** : le programme mathématique le plus utilisé dispose d'un module nommé « Signal Processing Toolbox » qui permet de faire beaucoup de traitement sur le signal. Le signal peut être inséré dans MatLab avec le format .RAW et, maintenant, il est capable de lire les fichiers .WAV.

- **OGI Tools** : Développée par le Oregon Graduate Institute (OGI) of Science and Technology (Portland, Oregon), il s'agit d'une application pour X-Windows, qui peut être trouvée « shareware » sur Internet[1]. Les sources sont aussi disponibles. Cette application permet d'étudier les signaux de parole, faire des spectrogrammes, extraire des formants, faire des analyses PLP, LPC et cepstrum, FFTs, conversions de formats, filtres, etc.

- **LVQ_PAK** : Développée par le LVQ Programming Team of the Helsinki University of Technology, il s'agit d'un ensemble de programmes pour

[1]On peut les obtenir à <speech.cse.ogi.edu> dans /pub/tools ou par email à <tools@cse.ogi.edu>

l'application de certains algorithmes de LVQ (Learning Vector Quantization) qui peut être trouvé aussi « shareware » sur Internet.

D'autres programmes disponibles via Internet sont référencés en Annexe C avec leurs adresses et comment les obtenir.

Il y a d'autres logiciels commerciaux disponibles sur le marché, mais il serait plus intéressant et adéquate de demander à l'application Accordion d'inclure quelque fonctionnalité de plus plutôt que d'acquérir les programmes qui la fourrent.

7.9. Conclusions :

Il s'agit d'un programme de simulation : sa mission est d'essayer. Avec ce programme j'ai essayé la plupart des algorithmes présentés dans ce rapport et j'ai pu encore examiner quelques détails qui, avec d'autres logiciels, aurait été presque impossible.

Néanmoins, il est converti aussi en un outil pour le traitement de la parole. Quoique ce n'était pas l'objectif de ce travail de faire un outil pour d'autres, sinon uniquement pour moi pour valider mes algorithmes, il est maintenant très valable (en combinaison avec d'autres logiciels, pourquoi pas) pour essayer de nouveaux développements dans le laboratoire par d'autres membres, ou même par des personnes externes.

Pour cette dernière raison, le programme a été laissé sur mon compte à l'Internet gratuitement comme shareware. Une copie du projet et des sources sera envoyée via e-mail s'elle est demandée.

J'ai mis beaucoup d'intérêt et beaucoup d'heures de travail dans ce logiciel et je voudrais continuer avec lui jusqu'à en faire un véritable programme professionnel qui un jour ait le droit de se qualifier comme version 1.0 de l'outil Accordion. Je ne sais pas si l'avenir me le permettra mais je suis sûr que la plus petite correction ou amélioration du programme sera envoyée directement à l'ECS, lieu de naissance de l'outil Accordion.

En résumé, ce logiciel est (avec la vitesse variable) l'autre grande satisfaction de ce projet. C'est la partie qui retient le plus l'attention, surtout à toutes les personnes qui (comme moi au commencement du projet) sont néoffites dans le domaine.

Chapitre 8

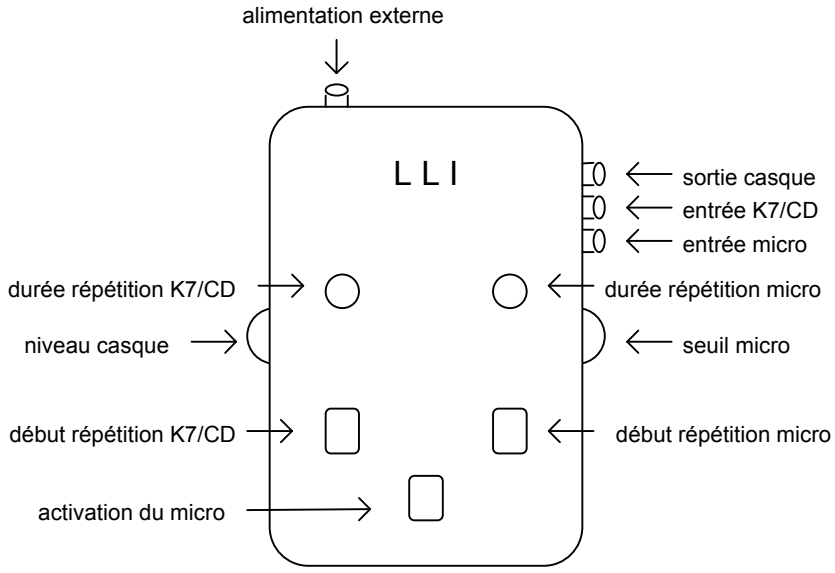
MISE EN OEUVRE

Les résultats de la simulation sont assez fiables pour ne pas requérir une concrétisation physique pour les valider. La qualité résultante sera très semblable car la seule différence résidera dans les convertisseurs CAN et CNA qui aujourd'hui donnent normalement des caractéristiques similaires. En ce qui concerne la complexité, l'évaluation préalable du nombre de cycles semble suffisante pour se renseigner sur la possibilité réelle des implantations et de leurs résultats.

Mais la possibilité de regarder dans la pratique que tout est réellement réalisable donnerait une satisfaction. De plus, cette mise en oeuvre peut avoir des objectifs commerciaux et les appareils qu'on va présenter peuvent aider à l'apprentissage des langues de beaucoup d'utilisateurs.

8.1. Configuration actuelle :

Deux modèles fondamentaux sont commercialisés par la société BARTHE : un Répétiteur Numérique (RN) pour l'apprentissage collectif des langues et un Laboratoire Individuel des Langues (LLI) pour l'apprentissage individuel. Le premier est un appareil avec la cassette intégrée dans un boîtier portatif d'apparence robuste et de taille moyenne destiné aux professeurs et aux écoles. Le deuxième, est une petite boîte compacte très petite sans le lecteur de cassettes, qui doit donc être connecté à un baladeur ou quelqu'autre moyen de reproduction audio. L'appearance du LLI est :



Tous les deux permettent de réécouter instantanément, sans recherche et sans action sur le lecteur, les dernières secondes d'un support audio, autant de fois que l'utilisateur le désire avec une durée d'une seconde à une dizaine de secondes. Une option de variation de vitesse d'élocution est également prévue.

Cette répétition est réalisée numériquement par un microcontrôlleur avec mémoire interne, protégée contre les re-lectures, qui fait les conversions analogique/numérique et numérique/analogique après un filtrage analogique. Il met ensuite les données en mémoire, il les récupère, puis il les traite. Une référence de tension est intégrée dans le composant.

La mise en place est la suivante :

- µC Intel 87C196KB-12 de 16 bits à 12 mhz avec OTP protégé,
- RAM statique de 256 Koctets (2 pages de 128 Koctets),
- 2 entrées A/N sur 8 bits à 10 khz,
- Filtres analogiques : Deux pour l'entrée à 4 khz et une pour la sortie à 4 khz (structure Tchebychev, ordre 4),
- 1 sortie N/A sur 8 bits avec PWM à deux fois la fréquence d'échantillonnage,
- Compression des blancs (silences) par gestion de pointeur,
- Pause automatique selon la mesure d'énergie du signal,
- Entrées / Sorties logiques pour le déclenchement de l'une ou l'autre des restitutions.

Un autre appareil de haute qualité ou d'enregistrement de lettres (quelques minutes) est aussi envisagé par la société et nous en présenterons aussi quelques notions sur lui. Mais nous nous centrerons principalement sur le LLI qui sera la version destinée au grand public et qui est encore en phase de développement et de concrétisation. Voyons ensuite ce qui nous a porté jusqu'à la configuration que l'on vient d'examiner et évaluons l'architecture des versions futures.

8.2. Configurations proposées :

Trois niveaux vont être étudiés : le niveau logique-fonctionel, le niveau structurel et le niveau des composants.

8.2.1. Niveau Logique :

C'est le niveau qui concerne le logiciel, c'est-à-dire les algorithmes. Ceux-ci ont été le but du projet et viennent d'être exposés dans ce rapport. Des améliorations ont été également commentées dans le chapitre sur la complexité.

Les algorithmes choisis sont les suivants :

- Compression ADPCM sur 4 ou 3 bits. Comme je l'ai commenté dans le chapitre sur la compression, cette méthode fournit la qualité requise avec une compression d'environ 4:1.

- Vitesse variable avec élimination de discontinuités par fenêtres (TDHS) sans emphase (la plus simple possible qui donne de bons résultats en qualité). Si on l'implante par tabulation, cette méthode ne nécessite qu'un accès indexé à la mémoire pour prendre la valeur de la fenêtre dans la table, une multiplication pour l'échantillon d'une tranche, une autre multiplication pour l'échantillon de l'autre tranche et, à la fin, l'addition des deux valeurs.

- Filtre de préaccentuation. On choisira un filtre numérique assez simple (deuxième ou troisième ordre) comme ceux des étudiés dans ce rapport. Les caractéristiques du filtre pour cette application ne sont pas trop exigeantes.

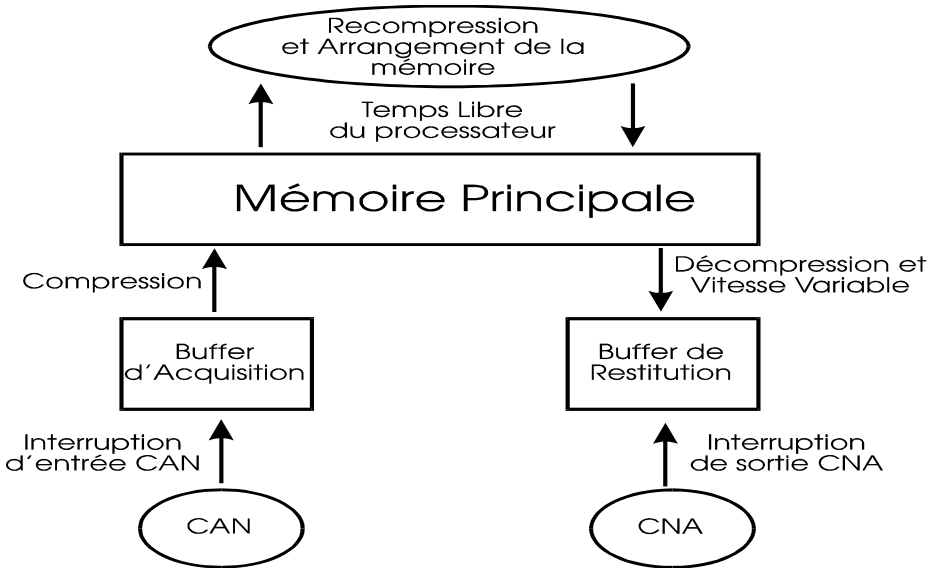
En ce qui concerne la compatibilité entre les algorithmes, comme la compression et la vitesse variable, on peut dire qu'aucune difficulté ne s'est trouvée pour les combiner et que le choix final d'ADPCM et de la vitesse variable temporelle est parfaitement compatible. De plus, la méthode TDHS permet d'une partition uniforme, plus appropriée pour l'ADPCM.

8.2.2. Niveau structurel :

La configuration présentée est, au niveau structurel de l'architecture interne de l'appareil, la plus logique et la plus simple pour mettre en oeuvre les algorithmes présentés et, à terme, la répétition numérique. Si nous voulons maintenir la complexité et la qualité des algorithmes, on peut opter pour d'autres solutions plus sophistiquées pour améliorer le fonctionnement ou l'efficacité. Quelques unes sont classiques en microinformatique comme les suivantes :

- ***Segmentation (pipelining)*** : (via logiciel ou matériel) de l'exécution de l'algorithme. Cette idée peu évoluer vers une séparation en plusieurs phases via interruptions. La première partie, l'acquisition, et la dernière, la restitution (peut-être décompression si celle-ci est simple) seraient réalisées par des interruptions, et une ou plusieurs phases (partition, compression, caractérisation, peut-être décompression si celle-ci est compliquée) partageraient intelligemment le temps restant du

processeur. En utilisant des *buffers* d'acquisition et restitution, ces algorithmes ne seraient pas être complètement en temps réel.



En plus, l'utilisation d'algorithmes de complexité réglable (Predicteur linéaire avec un nombre de coefficients plus grand ou plus petit, algorithmes récursifs de compression, algorithmes avec temps et qualité variable, algorithmes en cascade, compression sans perte après les compressions étudiées) peut nécessiter un ou plusieurs traitements des données, selon le temps disponible, pour arriver à une compression majeure. Ceci a l'inconvénient de récquerir une quantité de mémoire de taille variable et donc plus compliqué à gérer.

- ***Parallélisation*** : (soit du logiciel soit du matériel). Par exemple, deux microcontrôleurs en parallèle peuvent réussir de meilleurs résultats avec un coût moins cher qu'un DSP tout seul.

- ***Utilisation de petites mémoires caches (ou « latches »)*** : pour faire des buffers d'acquisition et de restitution en allégeant et libérant le programme.

Ces deux dernières solutions-ci ont le désavantage (en plus du prix) que la place disponible dans la boîte de l'appareil est souvent petite.

8.2.3. Niveau des composants :

Le développement et l'utilisation des LSI (Low Scale Integration circuits) est indispensable pour la réalisation des divers systèmes de traitement de la parole. Les algorithmes qui sont aisément implantables dans un LSI, tendent à devenir plus importants que ceux qui ne peuvent pas l'être.

Le choix ici mérite un examen plus complet parce qu'une fois la puce déterminée, il est très difficile de la changer. Initialement, l'utilisation d'un microcontrôleur, comme le Intel 87C196 fut la première solution et la plus simple pour le commencement d'un projet à la fois simple et général. Mais une fois les

fonctionnalités sont mieux définies, on doit passer à des composants plus concrets et efficaces.

Deux approches fondamentales ont été prises en la matière. La première est l'utilisation de puces de propos spécifique ou ASICs (Application Specific Integrated Circuits) et l'autre le DSPs (Digital Signal Processors). Les premiers ont l'avantage d'intégrer toutes les fonctions voulues dans un seul composant plus compact, plus rapide, plus fiable avec moins de problèmes de Compatibilité électromagnétique, moins de consommation. Tout cela peremt une reprise, une vérification et une distribution des résultats plus simples. L'avantage des DSP est, par contre, le développement a un prix plus abordable et sa flexibilité, puisque, actuellement, ils sont programmables en langages de haut niveau, ce qui permet l'incorporation simple des algorithmes sur la parole chaque fois plus diversifiés. En plus, il y a généralement un grand nombre de logiciel déjà développé.

En résumé, les avantages et inconvénients des ASICs vis à vis des DSPs sont :

	ASIC	DSP
Intégration	COMPACT	MOYENNE
Prix et durée du développement	HAUTE (recours à un fondeur)	MOYENNE ou BASSE (selon le logiciel déjà développé et les langage de programmation)
Connaissance pour le développement	Importantes (en CAO, IAO, ...)	Moyenne (en programmation et systèmes numériques)
Fiabilité	HAUTE (Tout est testé par le fabriquant)	MOYENNE (Le logiciel n'est jamais testé complètement)
Vitesse	HAUTE	HAUTE
Compatibilité Électro-Magnetique	FACILE	REQUIERT DES TESTS
Flexibilité	BASSE (réécriture)	HAUTE (par logiciel)
Deuxième Générations	FACILE	FACILE (Compatibilité des familles de composants)
Prix par unité	BASSE	MOYENNE
Dépannage	SIMPLIFIÉ (moins de boîtiers et de connectique)	COMPLEXE
Securité pour le copiage	HAUTE	SELON DSP
Consommation	BASSE	HAUTE

Ensuite, on présente ces deux approches avec les exemples concrets du TEXAS TMS-320C50 et l'ASIC qu'est en train de développer Boris Siepert...

8.3. Le DSP Texas TMS-320C50 :

Le choix du DSP le plus convenable pour cette application n'a pas été faite d'une façon exhaustive. L'existence d'un DSK (DSP Starter Kit) avec tous les composants requis est suffisante pour notre but. Des études plus élaborées sur d'autres DSP ont été considérées[1] comme la famille NEC μPD7720[2] ou le ADSP-2101 d'Analog Devices[3].

Le composant Texas TMS-320C50 est un DSP de cinquième génération de la famille TMS320. La famille TMS320 consiste en deux sous-familles : les puces en arithmétique fixe sur 16 bits (C1x, C2x et C5x) et les puces en arithmétique flottante sur 32 bits (C3x et C4x). Les caractéristiques suivantes permettent un choix dans cette famille pour un grand nombre d'applications en traitement de signal.

- Ensemble d'instructions très flexible,
- Flexibilité opérationnelle inhérente,
- Performance de haute vitesse,
- Architecture parallèle innovatrice,
- Coûts très accessibles.

Les applications de cette famille sont les mêmes que des autres DSPs :

- Systèmes de contrôle numérique,
- Traitement numérique du signal (Filtres, Transformées),
- Traitement d'image,
- Traitement de la parole (Vocodeurs).

En particulier, la génération 'C5x offre :

- Une architecture améliorée sur les générations précédentes,
- Une technologie IC avancée,
- Une compatibilité avec les générations 'C1x et 'C2x,
- Un ensemble d'instructions augmenté,
- Une technique de consommation et de rayonnement moindres.

En concret, le TMS320C50 se présente comme suit :

- Un boîtier carré de 132 pins en céramique, de technologie CMOS statique sous 5V,

[1] [RANDO89], [BODDIE81b]
[2] [BODDI81a], [BODDIE81b]
[3] [HIGGI90], [INGLE91], [ANALO92]

- Un temps de cycle de 35 à 50 ns (28.6 à 20 MIPS respectivement). Dans le kit 40 Mhz,
- Une structure pipelined pour gérer efficacement des sautes,
- Un addressage sur 16 bits,
- 28 registres logés en mémoire vive,
- 10 Ko de RAM (1Ko de données et 9Ko de données et programme),
- 2 Ko de ROM,
- 2 ports série full-duplex synchrones avec TDM (Time-Division Multiple-access),
- 64Ko de ports parallèles, dont 16 sont logés en la mémoire,
- 224Ko x 16 bits (au maximum) de mémoire externe adressable (64 Ko de programme, 64 Ko de données, 64 Ko d'entrées/sorties et 32 Ko de globales).
- Une Unité Arithmétique Logique (ALU) sur 32 bits, multiplieur de 16 x 16 bits,
- Des instructions de multiplication-accumulation en un cycle unique,
- Huit registres d'usage général,
- Onze registres de sauvegarde et une pile à huit niveaux,
- Deux buffers pour d'adressage circulaire,
- Des deccaleurs sur 16 bits à gauche ou à droite,
- Des instructions de répétition de bloc en un cycle unique
- Des instructions de mouvement de blocs de données,
- Une grande capacité d'interruptions et de DMA,
- Un adressage par index.
- Un adressage inversé pour les FFTs.

De plus, une immense bibliographie est disponible pour cette puce et toute sa famille.[1]

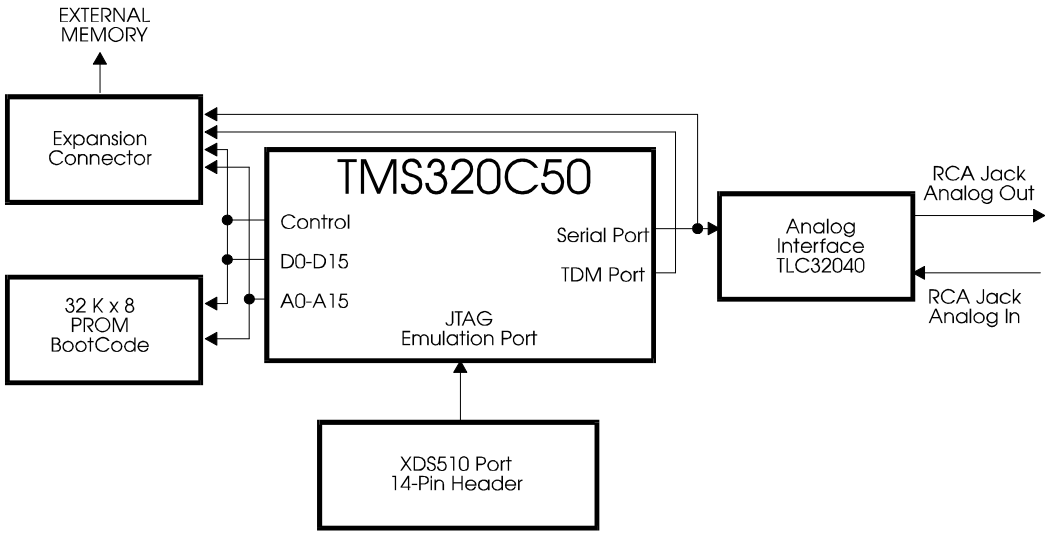
8.3.1. Le DSK (DSP Starter Kit):

[1] [TEXAS86], [TEXAS89], [TEXAS90], [TEXAS93], [TEXAS94]

Le système acheté pour les essais comprend :

- La puce TMS-320C50 à 50 ns de temps de cycle,
- 32 Koctets de PROM (Programmable Read Only Memory),
- Un convertisseur A/N et N/A Texas TLC32040CFN avec connecteurs standards RCA,
- Un générateur d'horloge à 40 Mhz,
- Une interface RS-232 avec l'ordinateur.
- 4 bus d'expansion I/O de 24 broches (JP2-5),
- Un emulateur de connecteur XDS510 de 14 broches (JP1).

Pour les travaux pratiques nous avons ajouté toute la mémoire possible (224 Koctets). Le schéma résultant est le suivant :



Le logiciel fourni est :

- Un assembleur dsk5a.exe pour DOS.
- Un debogueur dsk5d.exe pour DOS.
- Quelques exemples de sources.

Un compilateur de C est aussi disponible avec un autre kit de développement mais pour les premiers essais, l'assembleur est un outil suffisant étant donné que beaucoup de sources sont déjà codés pour l'assembleur de la famille TMS 320xx.

8.3.2. Le convertisseur A/N et N/A :

Le convertisseur A/N et N/A Texas TLC32040CFN fournit une conversion de haute qualité caractérisée par :

- Un filtre anti-repliement d'entrée ajusté sur la fréquence d'échantillonnage,
- Un convertisseur A/N sur 14 bits de résolution,
- Un convertisseur N/A sur 14 bits,
- Un filtre de reconstitution.

Avec une fréquence de base de 288 Khz, les diviseurs suivants sont admis :

B= 15 Fe= 19,2 Khz
B= 20 Fe= 14,4 Khz
B= 30 Fe= 9,6 Khz
B= 36 Fe= 8 Khz
B= 40 Fe= 7,2 Khz

Initialement, B est configuré à 36.

Cette puce ne fournit pas la correction sin x / x pour le convertisseur N/A. Cette correction doit être réalisé par logiciel sur le DSP. Une correction assez bonne consiste à un filtre du premier ordre passe-bande à 3000 Hz. Selon la fréquence, la valeur de correction est différente :

fs(Hz)	20 log [(sin [π f/fs]) / [π f/fs]] f= 3000 Hz (dB)
7200	-2.64
8000	-2.11
9600	-1.44
14400	-0.63
19200	-0.35

Les coefficients sont calculés aux pages B-25 et B-26 des manuels du Texas[1].

8.3.3. Algorithmes à implanter :

Le but initial était d'implanter tout les schémas proposés : les filtres de préaccentuation, la compression ADPCM (et d'autres) et une version simplifiée de la vitesse variable.

Des implantations pour la famille TMS320 de beaucoup de filtres (FIR et IIR) peuvent être trouvées dans [WILLI88] et dans les manuels « Digital Signal Processing Applications with the TMS320 Family »[2] ou dans l'article de Lovrich et Simar[3].

L'ADPCM a été extraite en grande partie des manuels[4] mais le source présenté fait tous les calculs numériquement, sans utiliser aucune table, pour des raisons d'économie de mémoire.

[1] [TEXAS94]
[2] [TEXAS89]
[3] [LOVRI89]
[4] [TEXAS89]

À 19.2 khz (B = 15), on a 19200 échantillons par seconde, ce qui correspond à 76.800 bits/s avec l'ADPCM à 4 bits. Avec 224Ko de mémoire, on dispose d'une capacité de stockage d'environ 20 secondes.

Pour les applications sur un appareil de temps d'enregistrement plus long, la chose est un peu plus compliquée. Si on réduit la fréquence jusqu'à 8 khz (B = 36), on peut enregistrer jusqu'à 1 min. Un ADM pourrait être mis en place avec une fréquence de 9,6 kHz et un post-filtrage, ce qui donne comme résultat environ 4 minutes d'enregistrement de qualité malheureusement assez moyenne. Mais d'autres méthodes plus élaborées peuvent s'essayer avec du temps.

8.4. L'ASIC :

Un ASIC est un circuit fabriqué à la demande pour une application donnée. La thèse de M. Boris Siepert a comme but principal la réalisation d'un ASIC pour incorporer toutes les fonctions du LLI (Laboratoire de Langues Individuel).

Quoique l'ASIC soit conçu pour des applications très spécifiques, il doit être utilisable dans des situations différentes. Il faut donc incorporer des capacités suffisantes pour satisfaire de futurs besoins. Une réalisation voisine de la solution actuelle (résolution 8 bit, fréquence d'échantillonnage 10 khz) paraît inadaptée à ce défi et sera bientôt dépassée. On arrive alors au cahier des charges suivant :

- 3 entrées CAN de 12-14 bits à Fe >= 16 khz,
- Un filtrage passe-bas adapté à la fréquence d'échantillonnage,
- La mémorisation de plusieurs secondes des échantillons,
- Une compressions des silences et une ADPCM sur 2 ou 3 bits,
- La pause automatique (par détection de l'énergie du signal),
- 1 CNA sur 12 à 14 bits,
- Des entrées logiques pour contrôler l'appareil (déclencher de la restitution) gérées par interruptions,
- Des sorties logiques pour montres des informations,
- Un traitement numérique de la vitesse variable ou du timbre variable,
- Une suppression de bruit d'ambiant,
- Un système de communication: inter-ASIC ou ASIC-μP.

Il y a encore beaucoup d'études nécessaires avec plus ou moins les fonctionnalités précédentes (selon que l'ASIC inclut la mémoire, si la compression est réalisée en interne ou par un DSP extérieur...). Bien sûr, la thèse de Boris Siepert arrivera à un bon compromis entre fonctionnalité et coût de l'appareil résultant, et ceci d'ici 2 ans.

8.5. Conclusions :

Je n'ai pas eu de temps suffisant pour concrétiser la première partie de la mise en place sur le DSP, mais les algorithmes ont été codés en assembleur sur cette machine et toute l'étude préalable est déjà réalisée. Les résultats seront donc à court-terme disponibles quand certains problèmes dus à la modification des sources et à l'accès de la nouvelle mémoire rajoutée seront résolus.

En ce qui concerne l'ASIC, on devrait attendre un peu plus longtemps pour sa réalisation. Étant donné qu'il sera réalisé spécifiquement pour l'appareil, le choix de cette option comme composant fondamental est déterminant.

J'attends avec un grand enthousiasme les résultats concrets de ces deux mises en oeuvres, ce qui pourra montrer en pratique une partie de tout ce que j'ai réalisé dans mon projet et que je viens d'illustrer dans ce rapport.

Chapitre 9

RÉSULTATS ET CONCLUSIONS

Quelquefois, les résultats ne correspondent pas aux efforts réalisés. D'autres fois, les résultats sont inesperément bons. Dans ce travail, il est arrivé un peu les deux cas.

9.1. Objectifs accomplis :

L'intention finale que l'on a lorsqu'on fait un travail, c'est qu'il soit profitable. C'est pour cela que j'ai mis beaucoup d'acharnement dans ce rapport pour bien documenter tout ce que j'ai fait. Le but est de donner la possibilité de comprendre ce que j'ai réalisé et de permettre aux futurs chercheurs de reprendre et continuer ce travail.

Quant aux objectifs concrets, sincèrement, je crois que la plupart d'entre eux ont été accomplis, quelques uns avec plus de dévouement et de succès que d'autres, mais rien n'est abandonné avant de le tenter.

La recherche bibliographique a été exhaustive comme le montrent les nombreuses références inclues dans le rapport, qui peut être utilisé comme point de départ pour quelqu'autre recherche plus spécifique.

En ce qui concerne la simulation et l'analyse du signal de la parole, l'outil Accordion dépasse amplement les buts initiaux, permettant des fonctionnalités les plus diverses. Cet outil permet la lecture de plusieurs formats de son, la représentation du signal, de son spectre et de ses sonagrammes, l'application de différents filtres, l'extraction d'une masse de paramètres, la génération d'ondes, le changement entre les formats des fichiers de son, etc ... etc ... Évidemment, il inclut la simulation de tous les algorithmes et mesures qui ont été développés. Une aide en ligne est aussi ajoutée au programme.

Les algorithmes de compression ont été examinés des plus simples (comme le PCM) jusqu'aux plus complexes (paramétriques ou spectraux). Finalement, on a choisi l'ADPCM comme l'algorithme le plus standardisé qui, avec un facteur de compression de l'ordre de 4:1, maintient toute la qualité. De plus, il est facilement réalisable.

La vitesse variable à donné des résultats plus satisfaisants, avec un algorithme de basse complexité et de haute qualité, qui est décrit avec assez de détail dans ce rapport et qui permet les réalisations les plus diverses.

Les essais sur la qualité du son n'ont pas donné les résultats espérés, étant donné la complexité des calculs nécessaires pour donner des mesures pareilles à celles de l'oreille humaine. Peut-être ai-je été trop ambitieux dans mes recherches, négligeant le fait que les travaux sur ce domaine sont très répandus dans la bibliographie et qu'aucune mesure objective n'a été accomplie jusqu'à ce moment.

Quant à la mise en place pratique, on a développé un moyen de calculer la complexité des algorithmes dans le programme de simulation, pour s'assurer que chaqu'un soit réalisable sur une cible concrète. Donc, écartant dès le début les algorithmes trop compliqués pour être implantés sur la puce cible, on évite de perdre son temps à tenter l'impossible. La puce choisie finalement a été le DSP TMS 320C50 de Texas Instruments qui est une des plus utilisées pour le traitement du signal.

Le résultat définitif n'est pas encore atteint, parce-que il dépend des deux réalisations entreprises ; la première, d'évaluation sur le DSP ; et la deuxième, pour des buts concrets, la commercialisation sur l'ASIC. Tous les deux seront accomplis bien sûr par la thèse de Boris Siepert.

9.2. L'avenir :

Un projet de recherche n'est jamais fini. Surtout quand on est perfectionniste, comme moi, on peut faire toute sa vie avec de petites retouches. Mais il y a un moment ou on doit arrêter et présenter certains résultats. Il est très possible qu'avec un ou deux mois additionnels de travail, le projet aurait été plus complet que maintenant. Franchement, j'aurais voulu faire cela, ou même entreprendre une thèse en la matière, mais il y a d'autres conditions et situations de nature fondamentalement personnelle qui sont plus importantes pour moi.

Avant le commencement du projet, mes connaissances sur le traitement de la parole étaient réduites. Précisément, ce n'était pas un des domaines sur lequel j'avais beaucoup d'intérêt mais, réellement, je n'ai pas eu la chance de choisir ; les bourses ERASMUS sont comme ça.

Étant donné ma condition d'informaticien, j'aimait bien la partie du projet qui consistait à développer des algorithmes de pure programmation. J'ai mis aussi une grande partie du temps à me familiariser avec le domaine du traitement de la parole, et à la fin, je l'ai trouvé très intéressant. Maintenant, il est un peu triste pour moi d'abandonner tout ce travail à ce point et d'oublier toutes ces connaissances.

Cependant, j'ai l'intention de continuer à travailler en collaboration (ou au moins en communication) avec l'ECS et la société BARTHE, pour aider encore, soit les membres actuels de l'équipe, soit de nouveaux arrivants qui peuvent reprendre mon travail. À cet égard, je serai en contact avec Boris Siepert via l'Internet et peut-être que j'enverrai un ou plusieurs « *patch »* ou correction à mon outil de simulation. De plus, je suis très curieux du déroulement de sa thèse et de la réalisation de l'ASIC qui peut concrétiser, directement ou indirectement, quelque partie de mon travail.

Je voudrais aussi faire un article sérieux sur la vitesse variable pour tenter de le publier, étant donné qu'aucun article n'a été présenté sur cette matière. Dans le cas de sa publication, ce serait un article de plus dans les publications de l'ECS.

Finalement, je veux réitérer mon désir que ce projet soit utile pour l'avenir. Je veux aussi m'excuser pour toutes les erreurs et manques qui, après toutes les révisions, restent encore. Et, principalement, je voudrais que le temps et l'effort employés dans ce travail ne soient pas perdus.

Annexe A

Bibliographie

Une grande partie de ce travail a été basé sur une recherche bibliographique. Ainsi que le montre, le grand nombre de livres et d'articles référencés dans cette annexe.

La notation utilisée dans ces trois premières annexes est la suivante : Les livres et articles sont entre guillemets, les livres et revues en italique. Le code est formé avec cinq caractères pour l'auteur et deux pour l'année, enfermés entre deux crochets [ ]. Optionnellement, on pourra ajouter une lettre minuscule de plus pour différencier plusieurs références identiques. Les adresses Internet ne sont pas codées mais elles sont enfermées entre deux crochets < >.

Trois sources ont été fondamentales pour établir cette bibliographie : l'ECS, le centre de documentation de l'ENSEA et la bibliothèque de la « Universidad Politécnica de Valencia » (UPV).

Trois champs montrent la référence, le lieu et mon commentaire ; il s'agit, respectivement, de la source qui m'a donné compte de l'existence du document, du lieu où je l'ai trouvé ou le peux trouver et d'un petit commentaire personnel dans lequel j'ai mis un résumé du sujet traité par le texte. Quelquefois, il peut apparaître le nom d'une personne qui a le texte ou m'a donné de l'information sur son contennu.

J'ai fait une claire distinction entre bibliographie et références. Dans ce premier annexe de bibliographie, j'ai indiqué les textes que j'ai lus ou, du moins, dont j'ai regardé les chapitres intéressants. Donc le lieu est toujours affiché car j'ai pu les obtenir.

[AHO__90a] AHO, Alfred V.
"Algorithms for Finding Patterns in Strings", *Handbook of Theoretical Computer Science*, Edited by J. van Leeuwen, (C) Elsevier Science Publisher 1990, pp. 255-300.
Ref: Lieu: Bibliothèque UPV
Com:

[AINSW92] *AINSWORTH, W.A. (Ed.).*

« *Advances in Speech, Hearing and Language Processing » Vol. 2*, Jai Press LTD. London, England, 1992.
Ref:Lieu: Bibliotèque UPV [4-63 432]
Com: Beaucoup d'articles différents; géneralement de Low-Rate Coding.

[ALSAK92] ALSAKA, Y.A.
"Contractive Speech Coding" Electronic Letters Vol.28 n°14, Juillet 1992, pp. 1358-1359.
Ref: Boris. Lieu: ECS. Bibliotèque ENSEA
Com: Plus de 120:1 compression (534 bit/s). Je peut leur croire, mais je ne comprend pas de l'explication du texte.

[ANALO92] ANALOG DEVICES.
"*Digital Signal Processing Applications. Using the ADSP-2100 Family*" Analog Devices Technical Reference Books, Prentice Hall, Englewood Cliffs, NJ 1992.
Ref: ECS. Lieu
Com: 590 pages. Beaucoup d'algorithmes. Prémière lire le [INGLE91].

[ANDER75] ANDERSON, John B.; BODIE, John B.
"Tree Encoding of Speech" IEEE Transactions of Information Theory Vol IT-21, n°4, July 1975, pp. 379-387.
Ref: Boris. Lieu: ECS
Com: Une vue sur les caractéristique de la parole pour le codage

[ATAL_82b] ATAL, Bishnu S.
" Predictive Coding of Speech Signals at Low Bit Rates " *IEEE Trans. on Comm. Vol. COM-30, N°4, pp. 600-614, April 1982*
Ref: [KLEIJ94], Boris. Lieu : ECS
Com: APC

[ATAL_91] *ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.).*
« *Advances in Speech Coding »*, Kluwer Academic Publishers, 1991.
Ref: Internet: « comp.speech » et « comp.compression » FAQs. Lieu: Bibliotèque UPV [4-63 432]
Com: Beaucoup d'articles différents; géneralement de Low-Rate Coding.

[ATUNG93] ATUNGSIRI, S.A.; KONDOZ, A.M.
"Error control for low-bit-rate speech communication systems" IEE Proceedings-I, Vol. 140, No.2, April 1993
Ref: Boris. Lieu: ECS
Com: Méthodes pour ensure la robustess du codage.

[BARNW89] BARNWELL, T.; SCHAFER, R.; MERSEREAU, R.; SMITH, D.
« A Real Time Speech SubBand Coder Using the TMS32010 », pp. 531-542 in "Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 1" 1989
Ref: Lieu: ECS
Com:

[BELLA80] BELLANGER
"Traitement Numérique du Signal. Théorie et Pratique" Masson et C.N.E.T.-E.N.S.T., Paris 1980.
Ref: Lieu: Bibliotèque ENSEA [TS1 BEL].
Com: 376 pages. Une bonne introduction pour les transformées et l'échantillonnage. Il y a des algorithmes des transformées rapides mais ils sont en FORTRAN. J'étais au chapitre II lorsque je l'ai rendu .

[BOITE87] BOITE, René & KUNT, Murat
"*Traitement de la parole. Complément au Traité d'Electricité*" Presses Polytechniques Romandes, Lausanne 1987.
Ref: Lieu: Bibliotèque ENSEA [TS2 BOI].
Com: 285 pages. Elemental et introductoire. Chapitres 2, 3, 6, 10 et 11 sont recommendés. Le meilleure partie commence à la page 55.

[BOITE92] BOITE, René; LEICH, H.; YANG, Gao;
« A new efficient algorithm to compute the LSP parameters for speech coding », Signal Processing 27, pages 109-116, Elsevier 1992.
Ref: Boris: Lieu: ECS
Com: Des nouveaux filtres pour le CELP.

[BORLA93a] BORLAND International, Inc.
"Borland C++. Version 4.0" *Guide de référence et du programmateur*, 1993.
Ref: Lieu: ECS
Com:

[BORLA93b] BORLAND International, Inc.
"Borland ObjectWindows pour C++. Version 2.0" *Guide de référence, de l'utilisateur et du programmateur*, 1993.
Ref: Lieu:ECS
Com:

[BYTE] BYTE (Revue)
Septembre 1994
Ref: Lieu: Bibliotèque ENSEA.
Com: Utilisée comme recherche antérieure à l'achat de la carte de son..

[CASAC87] CASACUBERTA, Francisco y VIDAL, Enrique
"Reconocimiento Automático del Habla", Marcombo 1987.
Ref: Lieu: Bibliotèque UPV[4-63] 364
Com: Une complète introduction à la reconnaissance automatique de la parole.

[CHAN_93] CHAN, Wai-Yip; GERSHO, Allen
"High Fidelity Audio Coding with Generalized Product Code VQ" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Lieu: ECS
Com: MPEG à 128 kbps.

[CHAN_94] CHAN, C.F.; CHUI, S.P.
"Efficient codebook search procedure for vector-sum excited linear predictive coding of speech" Electronic Letters Vol. 30 n°17, Octobre 1994, pp. 1830-1834
Ref: Boris Lieu: ECS, Bibliotèque ENSEA
Com: Une méthode de cherche pour le VSELP.

[CHANG91] CHANG, Hyokang; WANG, Yi-sheng
« Qualitative Analysis and Enhancement of Sine Transform Coding » in *ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.).* « *Advances in Speech Coding »*, Kluwer Academic Publishers, 1991.
Ref: Internet: « comp.speech » et « comp.compression » FAQs. Lieu: Bibliotèque UPV [4-63 432],
Com: STC (Sine Transform Coding).

[CHEN_93a] CHEN, Juin-Hwey; RAUCHWERK, Martin
"8 kbs low-delay CELP coding of speech" in *ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.).* « Speech and audio coding for wireless and networks applications *»*, Kluwer Academic Publishers, 1993.
Ref: ECS. Lieu
Com: CELP.

[CHEN_93b] CHEN, H.; WONG, W.C.; KO, C.C.
« Comparison of pitch prediction and adaptation algorithms in forward and backward adaptive CELP systems », IEE Proceedings-I, Vol. 140, n°4, August 1993.
Ref: Boris. Lieu: ECS
Com: Comparaison de predicteurs de pitch pour CELP

[CHEN_93c] CHEN, H.; WONG, W.C.; KO, C.C.
"Low-delay hybrid vector excitation linear predictive speech coding" Electronic Letters Vol.29 n°25, December 1993, pp. 2164-2165.
Ref: Boris. Lieu: ECS
Com: LD-HVELP

[CLARK91] CLARKSON, Peter M.; BAHGAT, Sayed.
"Envelope expansion methods for speech enhancement" J. Acoust. Soc. Am. 89 (3), March 1991, pp. 1378-1382.
Ref: Boris. Lieu: ECS
Com: Préacentuation du signal.

[CLAVI79] CLAVIER, Jacques; COFFINET, Gérard; NIQUIL, Marcel; BEHR, Francis
"*Théorie et Technique de la Transmission des donnés. Tome I. Notions Fondamentales*" 2ème édition. Masson, Paris 1979.
Ref: Lieu: Bibliotèque ENSEA [TC1 CLA].
Com: 314 pages. Elemental et introductoire.

[CREAT94] CREATIVE LABS
"Sound Blaster AWE 32. Manuels d'installation et d'utilisation"
Ref: Lieu: Bibliotèque ENSEA [TC1 CLA].
Com: 314 pages. Elemental et introductoire.

[CROCH76] CROCHIERE, R.E.; WEBBER, A.; FLANAGAN, J.L.
"Digital Coding of Speech in Sub-Bands" Bell Syst. Tech. J., vol. 55, No. 8, pp. 1069-1085, 1976.
Ref: [BARNW89], Boris. Lieu: ECS
Com: SBC

[CUPER91] CUPERMAN, V.; PETTIGREW, R.
"Robust low-complexity backward adaptativ pitch predictor for low-delay speech coding" IEE Proceedings-I, Vol.138, n°4, August 1991, pp. 338-344.
Ref: Boris. Lieu: ECS
Com: LD-VXC

[DEVIS_93] DEVIS BOTELLA, Ricardo.
"Programación Orientada a Objetos en C++" Paraninfo 1993. ISBN 84-283-2056-X
Ref: Lieu:
Com:

[DIFRA92] DI FRANCESCO, Renaud
« Codage algébrique de la parole : prédiction linéarie à excitation par code ternaire », Ann. Télécommun., 47, n°5-6, 1992.
Ref: Boris: Lieu: ECS
Com: Techniques de code ternaire pour CELP

[DIMOL93] DIMOLITSAS, Spiros.
"Subjective Assessment Methods for the Measurement of Digital Speech Coder Quality" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Lieu: ECS
Com:

[DROGO93] DROGO DE IACOVO, R.; MONTAGNA, R.; SERENO, D.; USAI, P.
"A two-band CELP Audio Coder At 16 kbit/s and its evaluation" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Lieu: ECS
Com:

[FULDS92] FEHN, Heinz G.; NOLL, Peter
"Multipath Search Coding of Stationary Signals with Applications to Speech" IEEE Transactions on Communications, Vol. COM-30, n°4, pp. 687-701, April 1982.
Ref: Boris Lieu: ECS
Com:

[FONTA95] FONTANA, Céline; DORÉ, Cristophe.
« Babel in France » Le Figaro Grandes Écoles Universitaires, Lundi 23 Janvier 1995.
Ref: Lieu: ECS
Com: Fait réference à l'apprentisage des langues en général et des nouveaux outils mulitmédia.

[FULDS92] FULDSETH, A.; HARBORG, E.; JOHANSEN, F.T.; KNUDSEN, J.E.
"Wideband speech coding at 16 kbit/s for a videophone application" Speech Communication 11, pp. 139-148, Elsevier 1992.
Ref: Boris Lieu: ECS
Com: CELP

[FURUI89] FURUI, Sadaoki
"*Digital Speech Processing, Synthesis and Recognition"* Marcel Dekker 1989. ISBN 0-8247-7965-7
Ref: Lieu: Bibliotèque UPV [4-63 492]
Com: 390 pages. Très complet. Pages rélévantes: 45-83, 139-204.

[GAFFI94] GAFFIN, Adam; HEITKÖTTER, Jörg
"EFF's (Extended) Guide to the Internet" Textinfo Edition 2.3, September 1994.
Ref: Internet Lieu: Internet
Com: Une complète guide sur l'Internet.

[GALÈS95] GALÈS, Yann Le; RENAULT, Marie-Cécile.
« Le son entre dans les ordinateurs » Le Figaro Grandes Écoles Universitaires, Mardi 24 Janvier 1995.
Ref: Lieu:
Com: Fait réference à l'apprentisage des langues par ordinateur et des application multimédia en général.

[GRASS93] GRASS, J.; KABAL, P.; FOODEEI, M.; MERMELSTEIN, P.
"High Quality Low-Delay Speech Coding at 12 kb/s" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Lieu: ECS
Com: CELP à 12 kbps.

[GRAY_76] *Voire [MARKE76]*

[HAGEN92] HAGEN, R.; HEDELIN, P.
"Spectral coding by LSP frequencies - scalar and segmented VQ-methods" IEE Proceedings-I, Vol.139, n°2, April 1992, pp. 118-122.
Ref: Boris. Lieu: ECS
Com:

[HARRY] HARRY, Y; LAM, F.
"Analog and Digital Filters. Design and Realization " Prentice-Hall Series in Electrical and Computer Engineering 1979.
Ref: Lieu: Bibliotèque ENSEA [E06 WIL]
Com: Un survue des filtres.

[HAYAS94] HAYASHI, Shinji; AKITOSHI, Kataoka; MORIYA, Takehiro.
« 8 kbit/s Short and Medium Delay Speech Codecs Based on CELP Coding », European Transactions on Telecommunications, Vol. 5, No. 5, pages 573-581, September-October 1994.
Ref: Lieu: ECS
Com:

[HARBO93] HARBORG, Erik; FULDSETH, Arild; JOHANSEN, Finn Tore; KNUDSEN, Jan Eikeset
"A wideband CELP coder at 16 kbit/s for real time applications" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Boris. Lieu: ECS
Com: CELP

[HAYAS94] HAYASHI, Seiji; SUGUIMOTO, Masahiro; KISHI, Genya
"Low bit-rate MPC coders with adaptively controlled synthesis filter parameters" Signal Processing 35, pp. 285-293, Elsevier 1994.
Ref: Boris. Lieu: ECS
Com:

[HERNA_93] HERNÁNDEZ, Enrique; HERNÁNDEZ, José.
"Programación en C++" Paraninfo 1993. ISBN 84-283-2021-7
Ref: Lieu:
Com:

[HIGGI90] HIGGINS, Richard J.
"*Digital Signal Processing in VLSI*" Analog Devices Technical Reference Books, Prentice Hall, Englewood Cliffs, NJ 1992.
Ref: ECS. Lieu:
Com: 575 pages. Une bonne vue de tout cela referente au traitement du signal.

[HOLCK89] HOLCK, Andrew; ANDERSON, Wallace
« A Single-Processor LPC Vocoder », pp. 559-564 in "Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 1" 1989

Ref: Lieu: ECS
Com:

[HUSAI93] HUSAIN, Aamir; CUPERMAN, Vladimir
"Lattice Low Delay Vector Excitation For 8 kb/s Speech Coding" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Boris. Lieu: ECS
Com: CELP

[HUNT_92] HUNT, Melvyn J.
"The Speech Signal" in Nejat Ince A. Editor in « Digital speech processing: speech coding, synthesis and recognition » Kluwer Academic Publishers 1992.
Ref: Boris. Lieu : ECS
Com: Une survue du traitement de la parole trop introductoire.

[INFOPC] INFOPC (Revue)
Octobre 1994
Ref: Lieu: Bibliotèque ENSEA.
Com: Utilisée comme recherche antérieure à l'achat de la carte de son..

[INGLE91] INGLE, Vinay K. & PROAKIS, John G.
"*Digital Signal Processing Laboratory. Using the ADSP-2101 Microcomputer* " Analog Devices Technical Reference Books, Prentice Hall, Englewood Cliffs, NJ 1991.
Ref: ECS. Lieu: ECS
Com: 300 pages. Introduction à la puce 2101 avec des applications.

[JIANG92] JIANG, J.; JONES, S.
"Word-based dynamic algorithms for data compression" IEE Proceedings-I, Vol.139, n°6, December 1992, pp. 582-586.
Ref: Boris. Lieu: ECS
Com: Compression sans perte

[JINHA93] JINHAI, Cai.; GANGJI, Jiang; LIHE, Zhang
"New Method for extracting speech formants using LPC spectrum" Electronic Letters Vol.29 n°24, November 1993, pp. 856-857.
Ref: Boris. Lieu: ECS
Com:

[KATAO93] KATAOKA, Akitoshi; MORIYA, Takehiro
"Low Delay Speech Coder at 8 kbit/s With Conditional Pitch Prediction" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Boris Lieu: ECS
Com: CELP

[KEHOE92] KEHOE, Brendan P.
"Zen and the Art of the Internet" First Edition, January 1993.
Ref: Internet Lieu: Internet
Com: Une introduction à l'Internet.

[KLEIJ94] KLEIJN, W. Bastiaan; KROON, Peter.
« The RCELP Speech-Coding Algorithm », European Transactions on Telecommunications, Vol. 5, No. 5, pages 573-581, September-October 1994.
Ref: Lieu:
Com: Le codeur RCELP (Residual Codebook Excited Linear Prediction).

[KROON_91] KROON, Peter; ATAL, Bishnu S.
« On improving the Performance of Pitch Predictors in Speech Coding Systems » pages 321-327 in *« Advances in Speech Coding »*, Kluwer Academic Publishers, 1991.
Ref: Internet: « comp.speech » et « comp.compression » FAQs, Boris. Lieu: ECS, Bibliotèque UPV [4-63 432]
Com:

[KUNT_80] KUNT, M.
"Traitement Numérique des signaux" TE XX, Presses polytechniques romandes, Lausanne, 1980.
Ref: [BOITE87]. Lieu: Bibliothèque ENSEA [TS1 KUN]

Com:Bon sûr la FFT.

[KWON_93] KWON, C.H.; UN, C.K.
« CELP Based Mixed-Source Model for very low bit rate Speech Coding », Electronic Letters, 29, n°2, pages 156-157, March 1993.
Ref: Boris: Lieu: ECS
Com: CELP à 3 kbps.

[LAFLA93] LAFLAMME, C.; SALAMI, R.; ADOUL, J-P..
"9.6 kbit/s ACELP Coding of Wideband Speech" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Lieu: ECS
Com:

[LAW__93] LAW, K.W.; LEUNG, W.F.; CHAN, C.F.
"Reducing the Complexity and Storage of CELP Speech Coding using a Self-Orthogonal Codebook" Electronic Letters Vol.29 n°10, May 1993, pp. 928-930.
Ref: Boris. Lieu: ECS
Com: Melange de DCT et résaux neuronaux. Il peut être apliqué pour l'élimination de bruit dans quelques algorithmes de compression.

[LEE__89] LEE, J.I.; UN, C.K.
"Improving Speech Quality of CELP coder" Electronic Letters Vol. 25 n°19, September 1989, pp. 1275-1277.
Ref: Boris. Lieu: ECS
Com:

[LEIPP77] LEIPP, E.
« La machine à écouter. Essai de psycho-acoustique » Masson, Paris 1977.
Ref: Lieu: Bibliotèque ENSEA [PHS LEI].
Com: 260 pages. Psycho-acoustique.

[LENOU94] LE NOUVEL OBSERVATEUR
« Multimédia. Guide pour une révolution » Édition special avec la revue, Novembre 1994.
Ref: Lieu:
Com: Commente la révolution de le nouveau monde mulitmédia.

[LIND_90] LIND, L.F.; ATTKINS, P.M.; CHALLENER, P.
"An improved implementation of adaptive quantizers for speech waveform encoding schemes" in Whedon C., Linggard R. editors, « Speech and Language Processing » Chapman and Hall, London 1990.
Ref: ECS. Lieu: ECS
Com: Adaptative quantizers pour APCM, ADPCM.

[LITTL93] LITTLE, John N; SHURE, Loren
« Signal Processing TOOLBOX. For Use with MATLAB. User's Guide » The MathWorks, Inc. 1993.
Ref: Lieu:
Com:

[LOVRI89] LOVRICH, Al ; SIMAR, Ray
« Implementation of FIR/IIR Filters with the TMS32010/TMS32020 », pp. 27-67 in "Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 1" 1989
Ref: Lieu: ECS
Com:

[MACHA94] MACHADO, Ricardo Jorge & FERNANDES, Rui Pedro
"*Réalisation d'un Laboratoire Autonome de Langue Individuel*" Projet de Fin d'Études. ENSEA, Cergy. Paris 1994.
Ref: Lieu: ECS
Com: Base pour mon projet

[MARKE76] MARKEL, J.D.; GRAY, A.H. Jr.
« Linear Prediction of Speech » Springer-Verlag, Berlin 1976.
Ref:[BOITE87], [FURUI89], [RILEY89], [TEXAS93a], [RANDO89] Lieu: Bibliotèque ENSEA [TS2 MAR].
Com: 288 pages.

[MARPL87] MARPLE, S. Lawrence. Jr.;
« Digital Spectral Analysis with Applications » Prentice Hall Signal Processing Series, Alan V. Oppenheim, series editor 1987.
Ref: Lieu: Bibliotèque ENSEA.
Com: 492 pages, diskette avec les application en FORTRAN.

[MARQU94] MARQUES, Jorge S.; ABRANTES, Arnaldo J.
"Hybrid harmonic coding of speech at low bit-rates" Speech Communication 14, pp. 231-247, Elsevier 1994.
Ref: Boris Lieu: ECS
Com: Sinusoidal Coding

[MCAUL93] McAULAY, R.J.; QUATIERI, T.F.
"The Application of Subband coding to Improve Quality and Robustness of the Sinosoidal Transform Coder" Proceedings IEEE ICASSP 1992, pp. 439-442.
Ref: Boris Lieu: ECS
Com:

[MCELR92] McELROY, C.; MURRAY, B.; FAGAN, A.D.
"Wideband Speech Coding in 7.2 kb/s" Proceedings IEEE ICASSP 1992, pp. 620-623.
Ref: Boris Lieu: ECS
Com: CELP

[MICROa] MICROSOFT Corporation.
"Windows 3.1. Multimedia Reference", Window Help File
Ref: Fourni avec Borland C++ 4.0 Lieu:
Com:

[MICROb] MICROSOFT Corporation.
"Microsoft Windows 16-bit/32-bit API Reference", Window Help File
Ref: Fourni avec Borland C++ 4.0 Lieu:
Com:

[MICROc] MICROSOFT Corporation.
"Microsoft Windows Resouce Workshop Reference ", Window Help File
Ref: Fourni avec Borland C++ 4.0 Lieu:
Com:

[MICROd] MICROSOFT Corporation.
"Multimedia Standards Update: New Multimedia Data Types and Data Techniques", Window Help File
Ref: Lieu: Internet
Com: Très interessant. Il contient beaucoup de formats et compressions nouveaux et innovateurs.

[MINIS80] Ministère de l'Industrie
« Les machines parlantes. Perspective mondiale » N°2, La documentation française, Paris 1980.
Ref: Lieu: Bibliotèque ENSEA [TS2 MIN].
Com: 111 pages. Etat de l'industrie mondiale et française et previsions faites pour la decade des 80.

[MUNDA90] MUNDAY, E.
« Noise Reduction Using Frequency-Domain Non-Linear Processing for the Enhancement of Speech » in *Whedon C., Linggard R. editors, « Speech and Language Processing »* Chapman and Hall, London 1990.
Ref: ECS Lieu: ECS
Com: Réduction de bruit.

[NAYEB93] NAYEBI, Kambiz; BARNWELL, Thomas P.
"Low Delay Coding of Speech and Audio Using Nonuniform Band Filter Banks" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Boris Lieu: ECS
Com: SBC

[NEJAT92] NEJAT INCE, A.
"*Overview of Voice Communications and Speech Processing*" in Nejat Ince A. Editor in « Digital speech processing: speech coding, synthesis and recognition » Kluwer Academic Publishers 1992.
Ref: ECS Lieu

Com: Une survue du traitement de la parole orienté aux applications militaires (NATO).

[NELSO91] NELSON, Mark.
"The Data Compression Book", M&T Books, Redwood City, CA, 1991.
Ref: Internet: « comp.compression » FAQ Lieu
Com: Une introduction pratique à la compression de données. Tous les algorithmes sont en langage C.

[OPPEN75] OPPENHEIM, Alan W. & SCHAFER, Ronald W.
"*Digital Signal Processing*" Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1975.
Ref: [BOITE87], [FURUI89], [LITTL93], [TEXAS93a], [WILLI88]. Lieu: Biblotèque ENSEA [TS1 OPP]
Com: Introductoire mais trop technique et en plus déjà obsolet. Il parle sur Cepstral Representation. 585 pages.

[PAGNU89] PAGNUCCO, Lou; ERSKINE, Cole
« Companding Routines for the TMS32010/TMS32020 », pp. 169-211 in "Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 1" 1989
Ref: Lieu: ECS
Com:

[PANZE93] PANZER, Ira L.; SHARPLEY, Alan D.; VOIERS. William D.
"A comparison of Subjective Methods for Evaluating Speech Quality" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.), Kluwer Academic Publishers, 1993.
Ref: Lieu: ECS
Com:

[PAPAM89] PAPAMICHALIS, Panos; SO, John
« Implementation of Fast Fourier Transform Algorithms with the TMS32020 », pp. 69-168 in "Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 1" 1989
Ref: Lieu: ECS
Com:

[PILLA93] PILLAI, S.R.; ELLIOT, B.
"New Results on the Reconstruction of Bandlimited Signals from Past Samples" Electronic Letters Vol.29 n°17, August 1993, pp. 1501-1503
Ref: Boris Lieu: Bibliotèque ENSEA
Com: Il peut être apliqué pour l'élimination de bruit dans quelques algorithmes de compression.

[PUISS] PUISSANCE MICRO (Revue)
Octobre 1994
Ref: Lieu: Bibliotèque ENSEA.
Com: Utilisée comme recherche antérieure à l'achat de la carte de son..

[QIAN_94] QIAN, Yasheng; CHAHINE, Gebrael; KABAL, Peter
"Pseudo-multi-tap pitch filters in a low bit-rate CELP speech coder" Speech Communication 14, pp. 339-358, Elsevier 1994.
Ref: Boris Lieu: ECS
Com: CELP à 4.8 kbps

[QUESN95] QUESNE, J.F.
"CAO ASICS", ENSEA 1995.
Ref: Lieu:
Com: Un didacticiel pour réaliser des ASICs.

[RABIN78] RABINER, Lawrence R. & SCHAFER, Ronald W.
"*Digital Processing of Speech Signals*" Prentice-Hall, Inc., Englewood Cliffs, New Jersey 1978.
Ref:: Internet: « comp.speech » FAQ, [ALSAK92], [FURUI89], [TEXAS93a]. Lieu: Bibliotèque ENSEA [TS2 RAB].
Com: 512 pages.

[RANDO89] RANDOLPH, M.
« The Design of an Adaptive Predictive Coder Using a Single-Chip Digital Signal Processor », pp. 565-595 in "Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 1" 1989
Ref: Lieu: ECS
Com:

[REIME89] REIMER, Jay; MCMAHAN, Mike; ARJMAND, Masud

« 32-kbit/s ADPCM with the TMS32010 », pp. 469-530 in "Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 1" 1989
Ref: Lieu: ECS
Com:

[RILEY89] RILEY, Michael D.
« Speech Time-Frequency Representation », Kluwer Academic Publisher 1989
Ref:: Lieu: Bibliotèque UPV [4-68 96].
Com: 160 pages. Très specifique sur le « schematic spectrogram ».

[SAKOE79] SAKOE, H.
"Two-Level DP-Matching. A Dynamic Programming-Based Pattern Matching Algorithm for Connected Word Recognition", *IEEE Transactions on Acoustic, Speech and Signal Processing*, Vol. ASSP-27, núm. 6, décembre 1979), pp. 588-595.
Ref: [CASAC93] Lieu: Bibliothèque UPV.
Com: Algorithme de deux pas.

[SALAM89] SALAMI, R.A.
« Binary Code Excited Linear Prediction (BCELP) : New approach to CELP coding of speech without codebooks », Electronic Letters, 25, pages 401-403, March 1993.
Ref: Boris: Lieu: ECS
Com: BCELP

[SAOUD92] SAOUDI, Samir; BOUCHER, Jean Marc; LE GUYADER, Alain
« A new efficient algorithm to compute the LSP parameters for speech coding », Signal Processing 28, pages 201-212, Elsevier 1992.
Ref: Boris: Lieu: ECS
Com: Le parametres LSP pour le LPC

[SEDGE92] SEDGEWICK, Robert
"Algorithms in C++", Addison-Wesley 1992.
Ref: Lieu: Mon frère.
Com: 658 pages. Un des meilleurs livres d'algorithmes. En plus, en C++.

[SHOHA93] SHOHAM, Yair
"Low Delay Codign of Wideband Speech at 32 kbps using tree structures" in ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.) « Speech and audio coding for wireless and network applicatons », Kluwer Academic Publishers, 1993.
Ref: Boris Lieu: ECS
Com: LD-CELP

[SIMON95] SIMON, Laurent.
« L'erreur commercialisée du Pentium » Campus LE MENSUEL N°215, Édition Ile-de-France, 1995.
Ref: Lieu: ECS
Com: Fait réference à l'archiconue erreur du Pentium.

[SMYTH90] SMYTH, S.M.F.; CHALLENER, P.
"An Efficient Coding Speech Scheme for the Transmission of High Quality Music Signals" in Whedon C., Linggard R. editors, « Speech and Language Processing » Chapman and Hall, London 1990.
Ref: ECS Lieu: ECS
Com: Codage SBC à 128 kbps pour la musique.

[SPANI92] SPANIAS, A.S.; LOIZOU, P.C.
"Mixed Fourier/Walsh transform scheme for speech coding at 4.0 kbit/s" IEE Proceedings-I, Vol.139, n°5, Octobre 1992, pp. 473-481.
Ref: Boris Lieu: ECS
Com:

[STANL84] STANLEY, William D.; DOUGHERTY, Gary R. & DOUGHERTY, Ray
"*Digital Signal Processing. Second Edition*" Reston Publishing Company, Inc., Reston, Virginia. 1984.
Ref: Lieu: Bibliotèque ENSEA [TS1 STA].
Com: 514 pages. Il parait facile. Les chapitres et sections plus importants sont: 9, 10, 12.15, 12.16 et 14.

[STEEN92] STEENEKEN, Herman J.M.
"*Quality Evaluation of Speech Processing Systems*" in Nejat Ince A. Editor in « Digital speech processing: speech coding, synthesis and recognition » Kluwer Academic Publishers 1992.
Ref: ECS Lieu
Com:

[STROU91] STROUSTROUP, Bjarne.
"*The C++ Programming Language, 2nd Edition*" Addison-Wesley, 1991, ISBN: 0-201-53992-6
Ref: Lieu: UPV
Com: 669 pages.

[TETSC91] TETSCHNER, Walt
"Voice Processing", Artech House 1991
Ref: Lieu: Bibliotèque UPV [4-63 420] -> [4-68 97/98]
Com: Très introductoire et général (State of the Art). 280 pàgs.

[TEXAS86] TEXAS INSTRUMENTS
"Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations" 1986
Ref: Lieu: ECS
Com:

[TEXAS89] TEXAS INSTRUMENTS, Edited by Kun-Shan LIN, Ph. D.
"Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 1" 1989
Ref: Lieu: ECS
Com: Le même libre que le [TEXAS86] actualisé.

[TEXAS90] TEXAS INSTRUMENTS, Edited by Panos PAPAMICHALIS, Ph. D.
"Digital Signal Processing Applications with the TMS320 Family. Theory, Algorithms and Implementations. Volume 3" 1990
Ref: Lieu: ECS
Com:

[TEXAS93] TEXAS INSTRUMENTS
"TMS320C5x User's Guide" Digital Signal Processing Products 1993
Ref: Lieu: ECS
Com:

[TEXAS94] TEXAS INSTRUMENTS
"TMS320C5x DSP Starter Kit. User's Guide" Microprocessor Development Systems 1994
Ref: Lieu: ECS
Com:

[THOMA92] THOMAS, Yves
"Signeaux et systèmes linéaires" Masson 1992.
Ref: Lieu: Bibliotèque ENSEA [TS1 THO]
Com: Livre de texte. La source principale pour la réalisation des filtres.

[VARMA89] VARMA, V.; LIN, D.W.; DIXON, J.L.
"Nonlinear quantisation of spectral shape in sub-band coding" Electronic Letters Vol.25 n°8, Avril 1989, pp. 550-551.
Ref: Boris Lieu: Bibliotèque ENSEA
Com: SBC (8 ou 16 bands) using RMS (Root Mean Squares) values via μ-law. En variant la μ pour les différentes bandes, il réussit un SNR très amelioré.

[VELEV93] VELEVA, L.V.; KUNCHEV, R.K.
"Adaptative Speech Coding with DCT and Neural Net Vector Quantization " Electronic Letters Vol.29 n°8, Avril 1993, pp. 704-705.
Ref: Boris Lieu: ECS
Com: Melange de DCT et résaux neuronaux. Il peut être apliqué pour l'élimination de bruit dans quelques algorithmes de compression.

[VIGNA93] VIGNAUX, Georges.
"Les sciences cognitives. Une introduction" Éditions La Découverte 1992.
Ref: Ricardo Lieu: Bibliotèque ENSEA [IA9 VIG]

Com: Dans le première chapitre il parle de la compression et traitement de la parole a niveau de l'intelligence artificelle. Je suis arrivé jussqu'à la page 64.

[WILLI88] WILLIAMS, Arthur B.; TAYLOR, Fred J.
"Electronic Filter Design Handbook. 2nd Edition" McGraw-Hill Publishing Company 1988.
Ref: Lieu: Bibliotèque ENSEA [E06 WIL]
Com: Un survue des filtres. Il incluit des applications pour la famille TMS 320.

[WHEDD90] WHEDDON, C.
"Speech Communication" in Whedon C., Linggard R. editors, « Speech and Language Processing » Chapman and Hall, London 1990.
Ref: ECS Lieu: ECS
Com: Une survue trop introductoire sur la parole.

[YANG_93a] YANG, H.; KOH, S.N.; SIVAPRAKASAPILLAI, P.
"Quadratic Phase Interpolation for Voiced Speech Synthesis in MBE Model" Electronic Letters Vol.29 n°10, May 1993, pp. 856-857.
Ref: Boris Lieu: ECS
Com:

[YANG_92] YANG, Gao.; LEICH, H.; BOITE, R.
"Voiced Speech Coding at very low bit rates based on forward_backward Waveform Prediction (FBWP)" Proceedings IEEE ICASSP 1992, pp. 179-182.
Ref: Boris Lieu: ECS
Com:

[YANG_93b] YANG, Gao.; LEICH, H.; BOITE, R.
"Multiband code-excited linear prediction (MBCELP) for speech coding" Signal Processing 31, pp. 215-227. Elsevier 1993.
Ref: Boris. Lieu: ECS
Com:

[YELDE91] YELDENER, S.; KONDOZ, A.M.; EVANS, B.G.
"High Quality Multiband LPC Coding of speech at 2.4 kbits/s" Electronic Letters Vol. 27 n°14, July 1991, pp. 1287-1289.
Ref: Boris. Lieu: ECS
Com: MB-LPC à 2.4 kps

[YUAN_91] YUAN, Jing; CHEN, C.S.; ZHOU, Hongmei.
« An ADM Speech Coding with Time Domain Harmonic Scaling » in *ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.).* « *Advances in Speech Coding »*, Kluwer Academic Publishers, 1991.
Ref: Internet: « comp.speech » et « comp.compression » FAQs, Boris. Lieu: ECS, Bibliotèque UPV [4-63 432]
Com: ADM combiné avec TDHS.

Annexe B

Références

Ce projet devant servir comme référence pour des études futures, tous les lieux pour obtenir les informations sont valables. À la différence de la bibliographie, je n'ai pas lu ces textes. Chaque référence trouvée dans ce rapport fait allusion soit à la bibliographie soit à ces références de textes. Mais il y a encore d'autres livres qui n'apparaissent pas.

Pour souligner parmi autant de textes, les références que j'ai trouvées les plus intéressantes ou que j'aurais voulu trouver (mais que je n'ai pas pu obtenir), je les ai écrites avec une police plus grande.

[ADOUL86] ADOUL, J.P.
"La quantification vectorielle de formes d'ondes: approche algébrique" *Séminaire ENST (Paris, 14 février 1985)* et *Annales des télécommunications*1986
Ref: [BOITE87] Lieu Com:

[AHO__73] AHO, Alfred V. and ULLMAN, J.
"The Theory of Parsing, Translation and Compiling" Vol. I. Parsing, Prentice-Hall, 1973.
Ref: [CASAC87] Lieu: Bibliothèque UPV Com:

[ANTON79] ANTONIOU, A.
"*Digital Filters: Analysis and Design*" McGraw-Hill Company, Inc., New York 1979.
Ref: [TEXAS93a], [WILLI88] Lieu Com:

[ATAL_67] ATAL, B.S.; SCHROEDER, M.R.
"Predictive Coding of Speech Signals" *Proc. 1967 Conf. Speech Comm. Processing 1967*
Ref: [MACHA94], [FURUI89] Lieu Com:

[ATAL_70] ATAL, B.S.; SCHROEDER, M.R.
"Adaptive Predictive Coding of Speech Signals" *Bell Sytem Tech. J. Vol. 49, n°6, 1970*
Ref: [MACHA94], [RANDO89] Lieu Com:

[ATAL_71] ATAL, B.S.; HANAUER, S.L.
"Speech Analysis and Synthesis by Linear Prediciton of Speech Wave " *J. Acoust. Soc. Amer. Vol. 50, 2(Part 2), pp. 637-655, 1971*
Ref: [MACHA94], [FURUI89] Lieu Com:

[ATAL_79] ATAL, B.S.; SCHROEDER, M.R.

"Predictive Coding of Speech Signals and Subjective Error Criteria" *IEEE Trans. ASSP-27, Vol. 3, p. 247-254, June 1979.*
Ref: [HAYAS94], [RANDO89] Lieu Com:

[ATAL_85] ATAL, B.S.; SCHROEDER, M.R.
"Code-Excited Linear Prediction (CELP) : High quality speech at very low bit rates " Proceeding ICASSP '85 *pp. 937-940, June 1985.*
Ref: [SALAM89] Lieu Com:

[ATAL_93] *ATAL, Bishnu S. (Ed.), CUPERMAN, Vladirmir (Ed.) et GERSHO, Allen (Ed.).*
« Speech and audio coding for wireless and networks applications », Kluwer Academic Publishers, 1993.
Ref Lieu: Com:

[BAILE90] BAILEY, R.L. & MUKKAMALA, R.
"Pipelining Data Compression Algorithms" *The Computer Journal*, Vol: 33, n°4, 1990
Ref:[MACHA94]. Lieu Com:

[BARAB79] BARABELL, A.J.; CROCHIERE, R.E.
"Sub-band coder design incorporating quadrature mirror filters and pitch detection", *Int. Conf. on ASSP, ICASSP, Washingtion D.C.*, April 1978, pp. 191-195.
Ref: [BARNW89] Lieu Com: SubBand Coding (SBC)

[BARNS93] BARNSLEY, Michael F; HURD, Lyman P.
"Fractal Image Compression"
Ref: Internet «comp.compression » FAQ.Lieu: Com: Ce livre exprime aussi aisement la compression fractale.

[BARNW82] BARNWELL III, T.P.
"Subband coder desgin incorporating recursive quadrature filters and optimum ADPCM coders", *IEEE Trans. Acoust., Speech, Signal Processing, vol ASSP-30*, pp. 751-765, Oct. 1982
Ref: [BARNW89] Lieu Com: SubBand Coding (SBC)

[BAVIÈ94] BAVIÈRE, François
"La Jungle de la Compression: Textes, Images Fixes et Animées, Son..." *Le Monde Informatique*, n°582, 1994
Ref: [MACHA94]. Lieu Com:

[BELLA82] BELLAMY, J.C.
"Digital Telephony" John Wiley & Sons 1982.
Ref: [TEXAS93] Lieu: Com:

[BELLA76] BELLANGER, M.G.; BONNERST, G.; COUDREUSE, M.
"Digital filtering by polyphase network : Application to sample-rate alteration and filter banks", *IEEE Trans. Acoust., Speech, Signal Processing, vol ASSP-24*, pp. 109-114, Apr. 1976
Ref: [BARNW89] Lieu Com: SubBand Coding (SBC)

[BERAN54] BERANEK, L.
"Acoustics ", McGraw-Hill, New York 1954.
Ref: [RILEY89] Lieu Com:

[BERGE71] BERGER, T.
"*Rate-Distorsion theory*" Prentice-Hall, New Jersey, 1971.
Ref: [BOITE87] Lieu Com:

[BLINN93] BLINN, J.F..
"What's the Deal with the DCT ", IEEE Computer Graphics and Applications, pp. 78-83.
Ref: Internet: « comp.speech » FAQ Lieu Com: Sur la DCT

[BODDI81a] BODDIE, J.R.; JOHNSON, J.D.; MCGONEGAL, C.A;
UPTON, J.W.; BERKLEY, P.A.; CROCHIERE, R.E.; FLANAGAN, J.L.
"Adaptative Differential Pulse-Code Modulation Coding", *Bell System Technical Journal 60 : No. 7, 1547-1561* September 1981.
Ref: [RANDO89] Lieu Com: ADPCM

[BODDI81a] BODDIE, J.R.; DARYANANI, G.T.; ELDUMIATI, I.I.;
GADENZ, R.N.; THOMPSON, J.S.; WALTERS, S.M.
"Digital Signal Processor : Architecture and Performance", *Bell System Technical Journal 60 : No. 7, 1449-1462* September 1981.
Ref: [RANDO89] Lieu Com: DSP

[BOLL_79] BOLL, S.F.
"Suppression of Acoustic Noise in Speech Using Spectral Subtraction" IEEE Trans. Acoust. Speech Signal Process. ASSP-27, 113-119, Avril 1979.
Ref: [CLARK91] [MUNDA90] Lieu: Com: Préacentuation du signal.

[BOOM_87] BOOM, Michael
« Music Through MIDI » Redmond: Microsoft Press, 1987.
Ref: [MICRO93a] Lieu: Com: Digital audio, digital-signal processing, and computer music.

[BRACE78] BRACEWELL, R.
"The Fourier Transform and its Applications", McGraw-Hill, New York 1978.
Ref: [RILEY89], fft de GNU, Steve Haehnichen. *Lieu Com:*

[BRAND92] BRANDENBURG, K.; STOLL, G.; DEHERY, Y.F.; JOHNSTON, J.D.; KERKHOF, L.V.D.; SCHROEDER, E.F.
"The ISO/MPEG-Audio Codec: A Generic Standard for Coding of High Quality Digital Audio" *92ème AES-convention, Vienna 1992, preprint 3336*.
Ref: Internet: « comp.compression » FAQ Lieu Com: MPEG y ASPEC.

[BRIGH74] BRIGHAM, O.
"*The Fast Fourier Transform*" Englewood Cliffs; New Jersey, Prentice Hall, 1974
Ref: [MACHA94], [TEXAS93a]. Lieu Com:

[BURRU84] BURRUS, C.S.; PARKS, T.W.
"DFT/FFT and Convolution Algorithms ", John Wiley & Sons, New York 1984.
Ref: [TEXAS93a] Lieu Com:

[CAMPB86] CAMPBELL, Joseph P. Jr.; TREMAIN, Thomas E.; WELCH, Vanoy C.
"Voiced/Unvoiced Classification of Speech with Applications to the U.S. Government LPC-10E Algorithm" *Proceedings of the IEEE International Conf. of Acoustics, Speech and Signal Processing*, 1986, pp. 473-476.
Ref: Internet: « comp.speech » FAQ Lieu Com:

[CAMPB90] CAMPBELL, Joseph P. Jr.; TREMAIN, Thomas E.; WELCH, Vanoy C.
"The Proposed Federal Standard 1016 4800 bps Voice Coder: CELP" *Speech Technological Magazine*, avril 1990, pp. 58-64.
Ref: Internet: « comp.speech » FAQ Lieu Com:

[CAMPB91a] CAMPBELL, Joseph P. Jr.; TREMAIN, Thomas E.; WELCH, Vanoy C.
"The Federal Standard 1016 4800 bps CELP Voice Coder" *Digital Signal Processing*, Academic Press, Vol.: 1, n°3, 1991, pp. 145-155.
Ref: Internet: « comp.speech » FAQ Lieu Com:

[CAMPB91b] CAMPBELL, Joseph P. Jr.; TREMAIN, Thomas E.; WELCH, Vanoy C.
"The DoD 4.8 kps Standard (Proposed Federal Standard 1016)" « *Advances in Speech Coding »; ed.: ATAL, CUPERMAN et GERSHO*, Kluwer Academic Publishers, Chap. 12, 1991, pp. 121-133.
Ref: Internet: « comp.speech » FAQ Lieu Com:

[CCITT84a] CCITT
"Prediction of Transmission Qualities form Objective Measures" Suppl. NO. 4, Red Book, Vol. V, pp. 214-236, Malaga-Torremolinos 1984.
Ref: [DIMOL93] Lieu: Com:

[CCITT84b] CCITT
"Recommendation G.721, 32 kbit/s Adaptative Differential Pulse Code Modulation" CCITT 1984.
Ref: [TEXAS93] Lieu: Com:

[CCITT88] CCITT
"Modulated Noise Reference Unit (MNRU)" Rec. P.81, Blue Book, Vol. V, pp. 198-203, Melbourne 1988.
Ref: [DIMOL93] Lieu: Com:

[CHAMB85] CHAMBERLIN, Hal
« Musical Applications of Microprocessors » Hasbrouk Heights: Hayden Book Company, Inc., 1985.
Ref: [MICRO93a] Lieu: Com: Digital audio, digital-signal processing, and computer music.

[CHEN_88] CHEN, C.S.; YUAN, Jing.
"A Robust Pitch Boundary Detector" Proc. ICASSP, Vol.1, pp. 366-369, Avril 1988.
Ref: [YUAN_91] Lieu Com:

[CHILD78] CHILDERS, D.G. (Ed.)
« Modern Spectrum Analysis » *IEEE Press*, New York, 1978
Ref: [BOITE87] Lieu Com:

[CLARK89] CLARKSON, Peter M.; BAHGAT, Sayed.
"A real-time speech enhancement system using Envelope Expansion Techniques" Proc. IEEE Electr. Letts. 25, pp. 1186-1188, 1989.
Ref: [CLARK91] Lieu: Com: Préacentuation du signal.

[CONWA82a] CONWAY, J.H. & SLOANE, N.J.A.
"Voronoi regions of lattices, second moment of polytopes and quantization" *IEEE Transactions IT-28*, n°2, mars 1982, pp. 211-226
Ref: [BOITE87] Lieu Com:

[CONWA82b] CONWAY, J.H. & SLOANE, N.J.A.
"Fast quantizing and decoding algorithms for lattice quantizers and codes" *IEEE Transactions IT-28*, n°2, mars 1982, pp. 227-232
Ref: [BOITE87] Lieu Com:

[COOLE65] COOLEY, J.W.; TUKEY, J.W.

« An algorithm for the machine calculation of complex Fourier series » *Math. of Computation*, Vol.19, April 1965, pp.297-301
Ref: [KUNT_80] Lieu: Com: Redécouverte de la FFT.

[COX__88] COX, R.V. et al.
"A Sub-Band Coder Designed for Combined Source and Channel Coding", Proc. ICASSP, Vol.1, pp. 235-238, Avril 1988.
Ref: [YUAN_91] Lieu Com:

[CROCH77] CROCHIERE, R.E.
"On the Design of Sub-Band Coders for Low-Bit-Rate Speech Communications" Bell Syst. Tech. J., vol. 56, pp. 747-770, 1977.
Ref: [BARNW89] Lieu Com: SBC

[CROCH79] CROCHIERE, R.E.
"Sub-band coder design incorporating quadrature mirror filters and pitch detection", *Int. Conf. on ASSP, ICASSP, Washingtion D.C.*, April 1978, pp. 191-195.
Ref: [BARNW89] Lieu Com: SubBand Coding (SBC)

[CROCH83] CROCHIERE, R.E.; FLANAGAN, J.L.
"Current Perspectives in Digital Speech" IEEE Commun. Magazine, January, pp. 32-40, 1983.
Ref: [FURUI89] Lieu Com:

[CROIS74] CROISIER, A.
"Progress in PCM and Delta Modulation: Block-Companded Coding of Speech Signals" *Proc. 1974 Zurich Seminar on Digital Communications*, Zurich 1974
Ref: [MACHA94] Lieu Com:

[CROIS76] CROISIER, A.; ESTEBAN, D; GALAND; G.
"Pefect channel splitting by use of interpolation/decimation/tree decomposition techniques" presented at the 1976 Int Conf. Inform. Sci. Syst., Patras, Greece, 1976.
Ref: [BARNW89] Lieu Com: SBC

[CUMMI73] CUMMISKLEY, P.; FLANAGAN, J.L.; JAYANT, N.S.
"Adaptative Quantization in Differential PCM Coding of Speech " *Bell System Tech. J.*, Vol. 52, n°7, 1973
Ref: [MACHA94] Lieu Com:

[CURTI78] CURTIS, R.A.; NIEDERJOHN, R.J.
"An investigation of several frequency-domain processing methods for enhancing the intelligibility of speech in wideband random noise", IEEE Proc ICASSP, pp. 602-605, Tulsa, Oklahoma, USA, Avril 1978.
Ref: [MUNDA90] Lieu: Com: Préacentuation du signal.

[DE_MO81] DE MORI, R. and GIORDANO, G.
"Algorithms for Syllabic Hypothesization in Continuous Speech", *Pattern Recognition*, Vol. 14, núm. 1-6, 1981, pp. 245-260.
Ref: [CASAC87] Lieu Com:

[DE_MO84] DE MORI, R. and LAFACE, P.
"Rule-Based Description of Acoustic Cues and Phonetic Features", NATO Adv. Studies Speech Recog., Bonas 1984.
Ref: [CASAC87] Lieu Com:

[DELLE93] DELLER, John R.; PROAKIS, John G.; HANSEN, John H.L.
"Discrete Time Processing of Speech Signals", Macmillan 1993.
Ref: Internet: « comp.speech » FAQ Lieu Com: Traitement du parole en général.

[DIMOL89] DIMOLITSAS, Spiros.
"Objective Speech Distorsion Measures and Their Relevance to Speech Quality Assessment" Proc. IEEE, Vol. 136, Pt. I, No. 5, pp. 317, 1989.
Ref: [DIMOL93] Lieu: Com:

[DRAGO78] DRAGO, P.G. et al.
"Digital Dynamic Speech Detectors ", *IEEE Trans on Communications*, Vol. 26, n°1, Janvier 1978, pp. 140-145.
Ref: Internet: « comp.speech » FAQ Lieu Com: Pour trouver le commençament et la fin d'une signal de parole.

[DUHAM90] DUHAMEL; GUILLEMOT.
"Polynomial Transform Computation of the 2-D DCT", ICASSP '90, p. 1515.
Ref: Internet: « comp.speech » FAQ Lieu Com: Sur la DCT

[EGER_84] EGER, T.E.; SU, J.C.; VARNER, L.W.
"A Nonlinear Spectrum Processing Technique for Speech Enhancement" Proc. IEEE ICASSSP, 18A.1.1-18A.1.4, 1984.
Ref: [CLARK91] Lieu: Com: Préacentuation du signal.

[ESTEB77] ESTEBAN, D.; GALAND, C.
"Application of quadrature mirror filters to Split-Band Coding ", *Int. Conf. on ASSP, ICASSP, Hatford*, May 1977, pp. 191-195.
Ref: [BOITE87], [BARNW89] Lieu Com: SubBand Coding (SBC).

[FALLS85] FALLSIDE, Frank (Ed) & WOODS, William A. (Ed).
"Computer speech processing", Englewood Cliffs, Prentice Hall 1985.
Ref: Internet: « comp.speech » FAQ Lieu Com: Traitement du parole en général. Bon en compression et codage.

[FANT_60] FANT, G.

"Acoustic Theory of Speech Production" Mouton's Co., Hague, 1960.
Ref:[FURUI89] [RILEY89] Lieu Com:

[FANT_80] FANT, G.
"The relations between area functions and th acoustic signal" Phonetica 37. 55-86, 1980.
Ref: [RILEY89] Lieu Com:

[FELDM83] FELDMAN, J.A.; HOFSTETTER, E.M.; MALPASS, M.L.
"A Compact, Flexible LPC Vocoder Based on a Commercial Signal Processing Microcomputer", *IEEE Trans. on ASSP*, Vol. ASSP-31, February 1983, pp. 257-259.
Ref: [HOLCK89] Lieu Com: LPC

[FJALL85] FIALLBRANT, T.
"A TMS320 Implementation of a Short Primary Block ATC-System with Pitch Analysit", International Conference on Digital Processing of Signals in Communications, No. 62, 93-96, 1985
Ref: [TEXAS89] Lieu Com:

[FLANA56] FLANAGAN, J.L.
"Acoustic Extraction of Formant Frequencies from Continuous Speech" J.Acoust.Soc.Am. 28, 110-118, 1956.
Ref: [RILEY89] Lieu Com:

[FLANA72] FLANAGAN, J.L.
"Speech Analysis, Synthesis and Perception" 2nd expanded edition. Springer-Verlag, 1972.
Ref: [MARKE76], [FURUI89], [RILEY89] Lieu Com: 444 pages. ISBN 3-540-05561-4

[FLANA79] FLANAGAN, J.L. et al.
"Speech Coding" *IEEE Transactions. COM-27",* n°4, avril 1979, pp. 710-736.
Ref: [BOITE87], [RILEY89] Lieu Com: Comparaison et mesures de la qualité.

[FLANA90] FLANAGAN, J.L.; RIESGO, C.J.
"Speech Processing: a perspective on the science and its application » *AT&T Tech. J.,* 1990, Vol. 69, (5); pp. 2-13.
Ref: [ALSAK92] Lieu: Com:

[FONTO83] FONTOLLIET, P.G.
"*Systèmes de Télécommunications*" TE XVIII, Presses polytechniques romandes, Lausanne 1983.
Ref: [BOITE87] Lieu Com: Seulement Chap. 7 selon BOITE87.

[FORCE] FORCE, Vicent
"*Utilité de l'Approche Total Least Squares pour le Codage Adaptatif de Parole en Environment Bruite*" Thése Professionnelle: Mastére Traitment du Signal, Centre National d'Études des Télécommunications, Issy les Moullineaux, École Nationale Supérieure de Télécommunications, Paris.
Ref: [MACHA94]. Lieu Com:

[FURUI81] FURUI, S.
"Cepstral Analysis Technique for Automatic Speaker Verification" IEEE Trans. Acoust., Speech, Signal Processing, ASSP-29, 2, pp. 254-272, 1981.
Ref: [FURUI89] Lieu Com:

[GALAN84] GALAND, C.R.; NUSSBAUMER, H.J.
"New quadrature mirror filter structures", *IEEE Trans., ASSP-32*, June 1984, pp. 522-531.
Ref: [BOITE87] Lieu Com: SubBand Coding (SBC)

[GERSH91] GERSHO & GRAY.
"Vector Quantization and Signal Compression", Kluwer Academic Press, 1991.
Ref: Internet: « comp.compression » FAQ Lieu Com:.

[GERSO90] GERSON, I.; JASIUK, M.
« Vector Sum Excited Linear Prediction (VSELP) Speech Coding at 8 Kbps », ICASSP '90, pp. 461-464, 1990.
Ref: [HAYAS94] Lieu: Com:

[GIBSO89] GIBSON, Jerry D.
"*Principles of Digital and Analog Communications*" MacMillan, New York 1989.
Ref: Internet Lieu Com: Pages 276-277.

[GOLD69] GOLD, Bernard; RADER, C.M.
"Digital Processing of Signals", McGraw-Hill Company, Inc., 1969.
Ref: [TEXAS93a] Lieu Com:

[GOODM82] GOODMAN, D.J., NASH, R.
"Subjective Quality of the Same Speech Transmission Condition in Seven Different Countries", IEEE Transaction on Communications, Vol. COM-30, No. 4, Avril 1982.
Ref: [HERM92] Lieu: Com:

[GOODM83] GOODMAN, D.J., SURDBERG, C.E.
"Combined Source and Channel Coding for Variable-bit-rate Speech Transmission", B.S.T.J. vol: 62; no.7, pp. 2017-2036, Sept. 1983.
Ref: [YUAN_91] Lieu Com:

[GRIFF88] GRIFFIN, D.W.; LIM, J.S.
« Multiband excitation vocoder », IEEE Trans. ASSP, pp. 1223-1235., August 1988.
Ref: [CHANG91] Lieu: Com: STC (Sine Transform Coding).

[HAMMI77] HAMMING, Richard W.

"Digital Filters", *Prentice Hall, Inc. 1977*
Ref: [LITTL93], [TEXAS93a] Lieu Com:

[HAMMI80] HAMMING, Richard W.
"Coding & Information Theory", *Prentice Hall 1980*
Ref: Lieu: Bibliotèque UPV [4-63 58] Com

[HARRI78] HARRIS, F.J.
"On the use of windows for harmonic analysis with the discrete Fourier Transform", *Proc. IEEE*, Vol. 66, n°1, January 1978, pp. 51-83.
Ref: [BOITE87] Lieu Com:

[HARRY] HARRY, Y.; LAM, F.
"Analog and Digital Filters, Design and Realization", Prentice Hall Series in Electrical and Computer Engineering.
Ref: [MACHA94] Lieu Com:

[HASSA85] HASSANEIN, H.; BRYDEN, B.
"Implementation of the Gold-Rabiner Pitch Detector in a Real Time Environment Using an Improved Voicing Detector", Proceeding of IEEE International Conference on Acoustics, Speech and Signal Processing, Vol ASSP-33, No. 1, 319-20, 1985
Ref: [TEXAS89] Lieu Com:

[HAWLE77] HAWLEY, M.E. (ed.)
« Speech Intelligibility and Speaker Recognition », Vol 2. Benchmark papers in Acoustics, Dowden, Hutchinson, and Ross, Stroudburg, Pa. 1977.
Ref: [STEEN92] Lieu: Com:

[HESS_83] HESS, W.J.
" Pitch determination of speech signals", Springer-Verlag 1983.
Ref: [BOITE87], [KLEIJ94] Lieu Com: Estimation du pitch.

[HOUSE65] HOUSE, A.S.; WILLIAMS, C.E.; HECKER, M.H.L.; KRYTER, K.D.
"Articulation Testing Methods: Consonantal differentiation with a closed response set", J. Acoust. Soc. Am. 37, pp. 158-166,1965.
Ref: [HERM92] Lieu: Com:

[IEEE76] Digital Signal Processing Committee of IEEE, Acoustics, Speech and Signal Processing Society
"Selected Papers in Digital Signal Processing II", IEEE Press, New York 1976.
Ref: [MACHA94], [LITTL93] Lieu Com:

[IEEE79] IEEE ASSP DSP Commitee (Ed.).
"Programs for Digital Signal Processing", IEEE Press, John Wiley & Sons, 1979.
Ref: [LITTL93], [TEXAS93a] Lieu Com:

[IEEE82] IEEE Subcomittee on Subjective Measures
"IEEE Recommende Practice for Speech Quality Measurements", IEEE Transaction on Audio and Electroacoustics 17, pp. 227-246, 1982.
Ref: Lieu: Com:

[IRII_87] IRII, H.; ITOH, K.; KITAWAKI, N.
"Multi-lingual speech database for speech quality measurements and its statistic characteristics" Trans. Commitee on Speech Research, Acoustic. Soc. Japan, S87-69, 1987.
Ref: [FURUI89] Lieu Com:

[ITAKU71a] ITAKURA, F.; SAITO, S.
"Speech Information Compression Based on the Maximum Likehood Spectral Estimation" *J. Acoust. Soc. Japan,* Vol 27, 1971.
Ref: [MACHA94] Lieu Com:

[ITAKU71b] ITAKURA, F.; SAITO, S.
"Digital Filter Techniques for Speech Analysis and Synthesis" Proce. 7th Int. Cong. Acoust., Budapest, 25-C-1, 1971.
Ref: [FURUI89] Lieu Com:

[ITAKU78] ITAKURA, F.; TOHKURA, Y.
"Feature Extraction of Speech Signal and its Application to Data Compression" Joho-shori, 19, 7, pp. 644-656, 1978.
Ref: [FURUI89] Lieu Com:

[ITAKU81] ITAKURA, F.
"Speech Analysis-Synthesis based on Spectrum Encoding" J. Acoust. Soc. Jap., 37, 5, pp. 197-203, 1981.
Ref: [FURUI89] Lieu Com:

[JACKS86] JACKSON, Leland B.
"Digital Filters and Signal Processing", Kluwer Academic Publishers, 1986
Ref: [TEXAS93a] Lieu Com:

[JACQU90] JACQUIN, A.
"Fractal image coding based on a theory of iterated contractive image transformations" *Visual Comm. and Image Processing,* Vol SPIE-1360, 1990.
Ref: Internet «comp.compression » FAQ. Lieu: Com: Ce livre exprime aussi aisement la compression fractale.

[JAKOB63] JAKOBSON, R.; FANT, G.; HALLE, M.
"Preliminaries to Speech Analysis: The Distinctive Features and Their Correlates" MIT Press Boston 1963.
Ref: [FURUI89] Lieu Com:

[JAYAN73] JAYANT, N.S.
"Adaptive Quantizaztion With a One Word Memory" *Bell System Tech. J.,* 1973.
Ref: [MACHA94] Lieu: Com:

[JAYAN74] JAYANT, N.S.
"Digital Coding of Speech Waveforms: PCM, DPCM and DM Quantizers" *Proc. IEEE*, Vol. 62, 1974.
Ref: [MACHA94] Lieu: Com:

[JAYAN76] JAYANT, N.S.
"Waveform quantization and coding" *IEEE Press*, New York 1976.
Ref: [ALSAK92], [MACHA94], [TEXAS93] Lieu: Com:

[JAYAN84] JAYANT, N.S. & NOLL, Peter.
"*Digital Coding of Waveforms: Principles and Applications to Speech and Video*"
Prentice Hall, Englewood Cliffs, New Jersey 1984.
Ref: Internet «comp.compression » FAQ. [BOITE87], [TEXAS93] Lieu: Com: Algorithms et standards sûr compression d'audio.

[JOHNS80] JOHNSTON, J.D.
"A filter family designed for use in quadrature mirror filter banks", *Int. Conf. on ASSP, ICASSP, Denver, CO,* April 1980, pp. 191-195.
Ref: [BARNW89] Lieu Com: SubBand Coding (SBC)

[JONES87] JONES, D.L.; PARKS, T.W.
"A Digital Signal Processing Laboratory Using the TMS32010", Englewood Cliffs, Prentice Hall, NJ, 1987
Ref: [TEXAS93a] Lieu Com:

[KALTE83] KALTENMEIER, A.
"Implementation of Various LPC Algorithms Using Commercial Digital Signal Processors", *IEEE Int. Conf. ASSP*, April 1983, pp. 487-490.
Ref: [HOLCK89] Lieu Com: LPC

[KEISE81] KEISER, Bernhard E.
"Digital Telephony : Speech Digitization" George Washington Univeristy 1981.
Ref: [TEXAS93] Lieu: Com:

[KENT_92] KENT, Ray D.
"The Acoustic Analysis of Speech" 1992.
Ref Lieu: UPV [O-22/35] Com:

[KLEIJ90] KLEIJN, W.B., KRASINSKI, D.J.; KETCHUM, R.H.
"Fast Methos for the CELP speech coding algorithm » *IEEE Trans.*, Août 1990, 455P-38, (8), pp. 1330-1342.
Ref: [ALSAK92] Lieu: Com:

[KOLBU79] KOLBUS, D.L.
"Computer Speech Communication" Stanford Research Institute. Décembre 1979.
Ref: [MINIS80] Lieu Com:

[KROON88] KROON, P.; DEPRETTER, E.F.
"A class of analysis-by-sybthesis predictive coders for high quality speech coding at rates between 4.8 and 16 kbits/s » *IEEE J. Selected Areas Comm.*, Vol. 6, pp. 353-363, 1988.
Ref: [KLEIJ94] Lieu: Com:

[KROON92] KROON, P.; SWAMINATHAN, A.
"A high-quality multirate real-time CELP coder » *IEEE Journal on Selected Areas in Communications*, Vol. 10, N. 5, pp. 850-857, 1992.
Ref: [KLEIJ94] Lieu: Com:

[KUC__] KUC, Bernard
"Introduction to Digital Signal Processing" McGraw-Hill International Editions, Electronical Engineering Series
Ref: [MACHA94] Lieu Com:

[LEE__80] LEE, D.T.
"Two-dimensional Voronoi Diagrams in the Lp-Metric" *Journal of the ACM*, Vol. 27, no. 4, octobre 1980, pp. 604-618
Ref: [CASAC887] Lieu Com:

[LEE__82] LEE, D.T.
"On K-nearest Neighbor Voronoi Diagrams in the Plane" *IEEE Transactions on Computer Science*, Vol. C-31, 1982, pp. 418-487
Ref: [CASAC87] Lieu Com:

[LEE__80] LEE, W.A. (Editor)
"*Trends in Speech Recognition"*, Prentice-Hall 1980.
Ref: [CASAC87] Lieu Com:

[LENDE65] LENDER, A.
"Delta Modulation Study" *Internal IT&T Fed. Lab.*, 1965
Ref: [MACHA94] Lieu Com:

[LICKL51] LICKLIDER, J.; MILLER, G.
"The perception of speech" in « Handbook of Experimental Psychology » Stevens, S. (Ed.), Wiley, New York, 1951.
Ref: [RILEY89] Lieu Com:

[LIM__79] LIM, Jae S.; OPPENHEIM, Alan V.
"Enhancement and Bandwidth Compression of Noisy Speech" Proc. IEEE 67, pp. 1586-1604, 1979.
Ref: [CLARK91] Lieu: Com: Préacentuation du signal.

[LIM__83] LIM, Jae S.
"Speech Enhancement" Prentice-Hall, Englewood Cliffs, NJ, 1983.
Ref: [CLARK91] Lieu: Com: Préacentuation du signal.

[LIM__88] LIM, Jae S.; OPPENHEIM, Alan V.
"Advanced Topics in Signal Processing", Englewood Cliffs, Prentice Hall, NJ, 1988
Ref: [TEXAS93a] Lieu Com:

[LINDE80] LINDE, Y.; BUZO, Q. & GRAY, A.M.
"An algorithm for vector quantization design" *IEEE Trans. Comm. COM-28* n°1, janvier 1980, pp. 84-95.
Ref: [BOITE87]. Lieu Com:

[LOEFF89] LOEFFLER; LIGTENBERG; MOSCHYTZ.
"Practical Fast 1-D DCT Algorithms with 11 Multiplications", ICASSP '89, p. 988.
Ref: Internet: « comp.speech » FAQ Lieu Com: Sur la DCT

[MAKHO75] MAKHOUL, J.
"Linear Prediction: A Tutorial Review ", Proc. of the IEEE n°63, 1975, pp. 561-580.
Ref: Internet: « comp.speech » FAQ, [RANDO89] Lieu Com: Traitement du parole en général. C'est bon sur compression et codage.

[MAKHO85] MAKHOUL J. et al.
"Vector quantification in speech coding" *Proc. IEEE,* novembre 1985, pp. 1551-1558.
Ref: [BOITE87] Lieu Com:

[MALAH79] MALAH, D.
"Time-Domain Algorithms for Harmonic Bandwidth Reduction and Time Scaling of Speech Signals " IEEE Trans. ASSP-27, pp. 121-133, Avril 1979.
Ref: [YUAN_91] Lieu Com:

[MARKE72] MARKEL, J.D.
"The SIFT Algorithm for Fundamental Frequency Estimation" IEEE Trans. Audio. Electroacoust., AU-20, 5, pp. 367-377, 1972
Ref: [FURUI89] Lieu Com:

[MATTH61] MATTHEWS, M.; MILLER, J; DAVID, E.
"Pitch Synchronous Analysis of Voiced Sounds" J.Acoust.Soc.Am. 45, 458-465, 1961.
Ref: [RILEY89] Lieu Com:

[MCAUL87] MCAULAY R.J.; QUATIERI, T.F.
« Multirate sinusoidal transform coding at rates from 2.4 kps. to 8 kps. », ICASSP, pp. 38.7.1-38.7.4., 1987.
Ref: [CHANG91] Lieu: Com: STC (Sine Transform Coding).

[MCAUL88] MCAULAY R.J.; QUATIERI, T.F.
« Computationally efficient sine wave synthesis and its application to sinusoidal transform coding », ICASSP, pp. 370-373., 1988.
Ref: [CHANG91] Lieu: Com: STC (Sine Transform Coding).

[MCMIL92] MCMILLAN; WESTOVER.
"A Forward-Mapping Realization of the Inverse DCT ", DCC '92, p. 219.
Ref: Internet: « comp.speech » FAQ Lieu Com: Sur la DCT.

[MICROa] MICROSOFT Corporation.
"The Multimedia Development Kit (MDK) 1.0 Programmer's Reference"
Ref: Lieu Com:

[MICROb] MICROSOFT Corporation.
"The Windows version 3.1 Software Development Kit (SDK) 1.0 Multimedia Programmer's Reference"
Ref: Lieu Com:

[MICROc] MICROSOFT Corporation.
"The Multimedia Programmer's Reference"
Ref: Lieu Com:

[MOORE90] MOORE, F. Richard
"Elements of Computer Music". New Jersey, Prentice Hall. 1990.
Ref: fft de GNU, Steve Haehnichen. *Lieu Com:Algorithme d'un pas avec Beam Search.*

[MOORE84] MOORE, R.K.
"Systems for Isolated and Connected Word Recognition". NATO ASI: New Systems and Architectures for Automatic Speech Recognition and Synthesis. Bonas, France, 2/14, juillet 1984.
Ref: [CASAC87] Lieu Com:Algorithme d'un pas avec Beam Search.

[MOORE86] MOORER, J.A.; BERGER, M.
"Linear Phase Bandsplitting : Theory and Applications" J. Audio Eng. Soc. 34, pp. 143-151, 1986.
Ref: [CLARK91] Lieu: Com: Préacentuation du signal.

[MORGA94] MORGAN, David P.
"Neural Networks and Speech Proccessing" 1994.
Ref: Lieu: Bibliotèque UPV [D-COM 363] Com

[MORGE82] MORGENTHALER M. and HANSEN C.
"Use of Attributed Grammars in Speech Signal Processing", ICASSP-82, 1982, pp. 1311-1313.
Ref: [CASAC87] Lieu: Com:

[MORRI83] MORRIS, Robert L.
"Digital Signal Processing Software", Carleton University, Otawa, 1983
Ref: [TEXAS93a] Lieu Com:

[MYERS81a] MYERS, C.S. and RABINER, L.R.
"A level Building Dynamic Time Warping Algorithm for Connected Word Recognition", *IEEE Transactions on Acoustic, Speech and Signal Processing*, Vol. ASSP-29, n°. 2 (avril 1981), pp. 284-297.
Ref: [CASAC93] Lieu Com: Alogirthme constructeur du nivel.

[MYERS81b] MYERS, C.S., RABINER, L.R. and ROSENBERG, A.E.
"On the Use of Dynamic Time Warping for Word Spotting and Connected Word Recognition", *The Bell System Technical Journal*, Vol. 60, n°. 3 (mars 1981), pp. 303-325.
Ref: [CASAC93] Lieu Com:

[NEC__] NEC
"Speech Products Synthesis. Compression Data Book"
Ref: P[MACHA94]. Lieu Com:

[NEJAT92] NEJAT INCE, A.
"Digital speech processing: speech coding, synthesis and recognition", Kluwer Academic Publishers, Boston 1992.
Ref: Internet: « comp.speech » FAQ Lieu Com: Traitement du parole en général.

[NEWMA90] NEWMAN, W.C.
"Detecting Speech with an Adaptative Neural Network ", *Electronic Design*, Vol. 22, mars 1990.
Ref: Internet: « comp.speech » FAQ Lieu Com: Pour trouver le commençament et la fin d'une signal de parole.

[NEY__84] NEY, H.
"The use of a one-stage Dynamic Programming Algorithm for Connected Word Recognition", *IEEE Transactions on Acoustic, Speech and Signal Processing*, Vol. ASSP-32, n° 2, avril 1984, pp. 263-271.
Ref: [CASAC93] Lieu Com:

[NOLL_64] NOLL, A.M.
"Short-Time Spectrum and Cepstrum Techniques for Vocal-Pitch Detection" J. Acoust. Soc. Amer., 36, 2, pp. 296-302, 1964.
Ref: [FURUI89] Lieu Com:

[NOLL_67] NOLL, A.M.
"Cepstrum Pitch Determination" J. Acoust. Soc. Amer., 41, 2, pp. 293-309, 1967.
Ref: [FURUI89] Lieu Com:

[NOLL_72] NOLL, P.
"Non-adaptative and adaptative DPCM of speech signals" *Polytechnique Tijds. Ed; Elektrotechn. Electron (THE Netherlands)* n°19, 1972.
Ref: [BOITE87] Lieu Com:

[NOLL_74] NOLL, P.
"Adaptative Quantizing in Speech Coding Systems" *Proc. 1974 Zurich Seminar on Digital Communications*, Zurich 1974.
Ref: [MACHA94] Lieu Com:

[NOLL_75] NOLL, P.
"A comparative study of various scheme for speech Encoding" *Bell Syst. Techn. J.*, V54, Novembre 1975, pp. 1597-1614.
Ref: [BOITE87], [MACHA94] Lieu Com:

[NUSSB81] NUSSBAUMER, H.J.
"Fast Fourier Transform and Convolution Algorithms" Springer Series in Information Sciences, Vol. 2, Springer-Verlag, 1981.
Ref: [MARKE76] Lieu Com: 248 pages. ISBN 3-540-10159-4

[NUSSB84] NUSSBAUMER, H.J., VETTERLI, M.
"Computationally Efficient QMF Filter Banks" Proc. IEEE Conf. ASSP, ICASSP, San Diego, mars 1984.
Ref: [BOITE87] Lieu Com: SubBand Coding (SBC)

[O'SHA87] O'SHAUGHNESSY, Douglas.
"Speech Communication: Human and Machine", Addison Wesley series in Electrical Engineering: Digital Signal Processing, 1987.
Ref: Internet: « comp.speech » FAQ, [ALSAK92] Lieu: Com: Traitement du parole en général. C'est bon sur compression et codage..

[OPPEN69] OPPENHEIM, Alan W.
"A Speech Analysis-Synthesis System Based on Homomorphic Filtering" J.Acoust.Soc.Am. 40, 458-465, 1969.

Ref: [RILEY89] Lieu Com:

[OPPEN] OPPENHEIM, Alan W.
"*Discrete-Time Signal Processing*" Prentice Hall, Inc.
Ref: [MACHA94]. Lieu Com:

[OPPEN] OPPENHEIM, Alan W.
"*Digital Processing of Speech*" Massachusetts Institut of Technology; Cambridge Press.
Ref: [MACHA94]. Lieu Com:

[OPPEN78] OPPENHEIM, Alan W. (Editor)
"Applications of Digital *Signal Processing*" Prentice Hall, Inc. 1978
Ref: [TEXAS93a] Lieu Com:

[OPPEN83] OPPENHEIM, Alan W.; WILLSKY, A.N.; YOUNG, I.T.
"Signals and Systems", Englewood Cliffs, Prentice Hall, Inc. 1983
Ref: [TEXAS93a] Lieu Com:

[OWENS93] OWENS, F.J.
"Signal Processing of Speech", Macmillan 1993.
Ref: Internet: « comp.speech » FAQ Lieu Com: Traitement du parole en général.

[PANTE] PANTER, Philip F.
"*Modulation, Noise and Spectral Analysis Applied to Information Transmission*" McGraw-Hill Book Company
Ref: [MACHA94]. Lieu Com:

[PAPAM87] PAPAMICHALIS, Panos E.
"Practical Approaches to Speech Coding", Prentice Hall, 1987.
Ref: Internet: « comp.speech » FAQ, [ALSAK92], [TEXAS93a] Lieu Com: c'est bon sûr LPC-10.

[PARKS87] PARKS, T.W.; BURRUS, C.S.
"Digital Filter Design", John Wiley & Sons, New York 1987.
Ref: [LITTL93], [TEXAS93a] Lieu Com:

[PARSO85] PARSONS, T.W.
"Voice and speech processing", McGraw Hill, New York 1986.
Ref: Internet: « comp.speech » FAQ Lieu Com: Traitement du parole en général.

[PECKE73] PECKELS, J.P.; ROSSI, M.
"Le test diagnostic par paires minimales", Revue d'Acoustique No. 27, 245-262, 1973.
Ref: [HERM92] Lieu: Com:

[PIERR79] PIERREL, J.M.
"Un système de comprehension automatique du dicours continu utilisant des contratintes morphologiques, syntaxiques et semantiques". R.A.I.R.O. Vol. 12, n°2, (1979) pp. 83-105.
Ref: [CASAC87] Lieu Com:

[PROAK] PROAKIS, John G.
"Digital Communications, Second Edition" McGraw-Hill International Editions, Electronical Engineering Series
Ref: [MACHA94]. Lieu Com:

[QUACK88] QUACKENBUSH, Sch.
"Objective measure of speech quality » 1988.
Ref: Lieu: Bibliotèque UPV [D-COM 363] Com

[RABIN70] RABINER, L.R., SCHAFER, R.W.
"System for automatic formant analysis of voiced speech", *J.Acoust.Soc.Am.* Vol.47, 634-648, 1970.
Ref: [RILEY89] Lieu Com:

[RABIN75a] RABINER, L.R., GOLD, B.
"Theory and Application of Digital Signal Processing", Prentice Hall, New Jersey 1975.
Ref: [BOITE87], [LITTL93], [TEXAS93a] Lieu Com:

[RABIN75b] RABINER, L.R., SAMBUR, M.R.
"An Algorithm for Detemining the Endpoints of Isolated Entrances", *Bell System Technical Journal*, Vol. 54, n°2, 1975, pp. 297-315.
Ref: Internet: « comp.speech » FAQ Lieu Com: Pour trouver le commençament et la fin d'une signal de parole.

[RABIN75c] RABINER, L.R., SCHAFER, R.W.
"Digital Representation of Speech Signals", *Proc. IEEE* Vol.63, n°4, 1975.
Ref: [MACHA94] Lieu Com:

[RABIN77] RABINER, L.R.
"On the use of t autocorrelation analysis for pitch detection"*, IEEE Trans. ASSP-25*, February 1977, pp. 24-33.
Ref: [BOITE87] Lieu Com: Estimation du pitch.

[RABIN79] RABINER, L.R., LEVINSON, S.E. and ROSENBERG, A.E.
"Speaker Independent Recognition of Isolated Words Using Clustering Techniques", *IEEE Transactions ASSP*, Vol. ASSP-27, n°4, août 1979, pp. 336-349.
Ref: [CASAC87] Lieu Com:

[RABIN83a] RABINER, L.R. et al.
"On the application of vector quantization and hidden Markov models to speaker-independant; isolated word recognition" *Bell Syst. Techn. J. 62* n°4, avril 1983, pp. 1075-1105.
Ref: [BOITE87]. Lieu Com:

[RABIN83b] RABINER, L.R. et al.
"Note on the properties of a vector quantizer for LPC coefficients" *Bell Syst. Techn. J. 62* n°8, décembre1983, pp. 2603-2616.
Ref: [BOITE87]. Lieu Com:

[RAO__90] RAO, K.R.; YIP P.
"Discrete Cosine Transform--Algorithms, Advantages, Applications", Academic Press, London 1990. ISBN 0-12-580203-X
Ref: Internet: « comp.speech » FAQ Lieu Com: Tout sur la DCT.

[READ_92] READ, C.; BUDER, E.; KENT, R.
"Speech Analysis Systems: An Evaluation", *Jounal of Speech and Hearing Research*, avril 1992), pp. 314-332.
Ref: Internet: « comp.speech » FAQ Lieu Com: Revision des paquets de processament audio de propos général.

[ROADS85] Roads, Curtis, and John Strawn
« Foundations of Computer Music » Cambridge: The MIT Press, 1985.
Ref: [MICRO93a] Lieu: Com: Digital audio, digital-signal processing, and computer music.

[ROBER62] ROBERTS, L.G.
"Picture coding using pseudo-random noise, *IRE Trans. Information Theory*, Vol. IT-8, 1962, pp. 145-154.
Ref: [KUNT_80] Lieu Com: Elimination d'effets artificiaux dans la compression en utilisant un bruit pseudo-aléatoire.

[ROSS_74] ROSS, M.J.; SHAFER, H.F.; COHEN, A.; FREUDBERG, R; MANLEY, H.J.
"Average magnitude difference function pitch extractor"*, IEEE Trans. ASSP-22*, October 1974, pp. 353-362.
Ref: [BOITE87], [RANDO89] Lieu Com: Estimation du pitch.

[SAITO85] SAITO, S.; NAKATA, K.
"Fundamentals of Speech Signal Processing" Academic Press Japan, Tokyo 1985.
Ref: [FURUI89] Lieu Com:

[SATO_75] SATO, H.
"Acoustic cues of male and female voice quality" Elec. Commun. Labs. Tech. J., 24, 5, pp. 426-449, 1975.
Ref: [FURUI89] Lieu Com:

[SCHAF79] SCHAFER, Ronald. W. & MARKEL, John.
"Speech analysis", IEEE Press, New York 1979.
Ref: Internet: « comp.speech » FAQ Lieu Com: Traitement du parole en général.

[SCHUZ92] SCHUZO, Saito.
"Speech Science and Technology", Ohmsha, Tokio 1992.
Ref: Internet: « comp.speech » FAQ Lieu Com: Traitement du parole en général.

[SCHWA90] SCHWARTZ, Mischa
"*Information Transmission, Modulation and Noise*" McGraw-Hill International Editions, Electronical Engineering Series 1990
Ref: [MACHA94]. Lieu Com:

[SEDGE92] SEDGEWICK, Robert
"Algorithms in C++" Addison Wesley Publishing Company 1992
Ref:. Lieu Com: Une bonne implantation de la FFT. 658 pages.

[SMITH--] SMITH, M.J.T.; BARNWELL III, T.P.
"A procedure for designing exact reconstruction filter banks for tree-structures", *proc. 1984 Int. Conf. Acoust., Speech, Signal Processing, San Diego, CA, Mar*. 1976
Ref: [BARNW89] Lieu Com: SubBand Coding (SBC)

[SPEHN86] SPEHNER, Jean-Claude
"La Reconnaissance des Facteurs d'un Mot dans un Texte", *Theoretical Computer Science* 48, 1986, pp. 35-52.
Ref Lieu Com:

[STEEL75] STEELE, R.
"Delta Modulation Systems " Halsted Press, London 1975
Ref: [MACHA94] Lieu Com:

[STEEN87] STEENEKEN, STEENnd J.M.
"Comparison among three subjective and one objective intelligibility tests", Report IZF 1987-8, TNO Institute for Perception, Soestererg, The Netherlands 1987.
Ref: [STEEN92] Lieu: Com:

[STRAW85a] STRAWN, John
"The Computer Music and Digital Audio Series" Volume 1: Digital Audio Signal Processing. Los Altos: W. Kaufmann, Inc. 1985
Ref: fft de GNU, Steve Haehnichen. *Lieu Com:*

[STRAW85b] STRAWN, John F.
« Digital Audio Engineering, An Anthology » Los Altos: William Kaufmann, Inc., 1985.
Ref: [MICRO93a] Lieu: Com: Digital audio, digital-signal processing, and computer music.

[STROC70] STROCH, R.W.
"Optimum and Adaptive Differential PCM" *Ph. D. Dissertation, Polytechnic Inst. of Brooklyn,* New York, 1970
Ref: [MACHA94] Lieu Com:

[TABOA94] TABOADA, J. et al.
"Explicit Estimation of Speech Boundaries", *IEEE Proc. Sci. Meas. Technol.*, Vol. 141, n°3, mars 1994, pp. 153-159.
Ref: Internet: « comp.speech » FAQ Lieu Com: Pour trouver le commençament et la fin d'une signal de parole.

[TAYLOR83] TAYLOR, F.J.
"Digital Filter Design Handbook", *Marcel Dekker, New York* 1983
Ref: [WILLI88] Lieu Com:

[TAYLOR87] TAYLOR, F.J.; STOURATIS, T.
"Digital Filter Design Software for the IBM PC", *Marcel Dekker, New York* 1987
Ref: [WILLI88] Lieu Com

[TOMBR90] TOMBRAS, G.S. & KARYBAKAS, C.A.
"New Adaptation Algorithm for a Two-Digit Adaptive Delta Modulation System" *International Journal on Electronics*, Vol. 68, n°3, 1990
Ref: P[MACHA94]. Lieu Com:

[TREIC87] TREICHLER, J.R.; JOHNSON, C.R.; LARIMORE, M.G.
"A Practical Guide to Adaptative Filter Design" John Wiley and Sons, Inc. 1987
Ref: [TEXAS93a] Lieu Com:

[TREMA82] TREMAIN, Thomas E.
"The Government Standard Linear Predictive Coding Algorithm: LPC-10" *Speech Technological Magazine*, avril 1982, pp. 40-49.
Ref: Internet: « comp.speech » FAQ Lieu Com:

[TRIBO79a] TRIBOLET, J.M. et al.
"Comparison of performances of four low-bit rate speech waveform coders" *Bell Syst. Technical Journal,* 1979, p. 699.
Ref: [BOITE87] Lieu Com: Adaptative Transform Coding (ATC)

[TRIBO79b] TRIBOLET, J.M. et CROCHIERE, R.E.
"Frequency domain cooling of speech" *IEEE Transactions ASSP-27,* n°4, novembre 1979, pp. 512-553.
Ref: [BOITE87] Lieu Com: Adaptative Transform Coding (ATC)

[VARY_89] VARY, P. et al.
"Speech codec for the European mobile radio system", Proc. IEEE Global Comm. Conf. November 1989.
Ref: [CHEN_93] Lieu Com: GSM

[VISWA80] VISWANATHAN, R.; RUSSELL, W.; HUGGINS, A.W.F.
"Design and Real-Time Implementation of a Robust APC Coder for Speech Transmission over 16 kbps Noisy Channels", *Bolt Beranek and Newman Inc.; BBN Rep. 4565* December 1980.
Ref: [RANDO89] Lieu Com: APC

[VITER] VITERBI et Al..
Ref: Lieu Com: Chapitres 7 and 8.

[VOIER77] VOIERS, W.D.
"Diagnostic Acceptability Measure for Speech Communication Systems", Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, Hartford, CT, Mai 1977.
Ref: Lieu: Com:

[VOIER77] VOIERS, W.D.
"Diagnostic Evaluation of Speech Intelligibility", Chapter 32 in M.E. Hawley (ed.) Speech Intelligibility and Speaker Recognition, Vol 2. Benchmark papers in Acoustics, Dowden, Hutchinson, and Ross, Stroudburg, Pa. 1977.
Ref: [STEEN92] Lieu: Com:

[VOIER83] VOIERS, W.D.
"Evaluating Processed Speech Using the Diagnostic Rhyme Test" Speech Technology, pp. 30-90, January-February 1983.
Ref: [DIMOL93]
Lieu:
Com:

[WALLA91] WALLACE, Gregory K.
"The JPEG Still Picture Compression Standard", Communications of the ACM, Avril 1991 (vol. 34 no. 4°, pp. 30-44.
Ref: Internet: « comp.compression » FAQ
Lieu
Com: Compression des texts qui changent rapide

[WILLI90] WILLIAMS, R.
"Adaptative Data Compression ", Kluwer Books, Boston 1990.
Ref: Internet: « comp.compression » FAQ
Lieu
Com: Compression des texts qui changent rapidement..

[WINKL63] WINKLER, M.R.
"High Information Delta Modulation" *IRE Conv. Record*, 1963
Ref: [MACHA94]
Lieu
Com:

[YANNA87] YANNAKOUDAKIS, E.J. and HUTTON, P.J.
"Speech Synthesis and Recognition Systems" Ellis Horwood Limited Series in Computer Science 1987.
Ref:
Lieu: Bibliotèque UPV [4-63 349] -> [4-68 72]
Com

[ZANEL84] ZANELLATO, G. et al.
"Reconnaissance de la parole" *Rapport annuel* 1984 (IRSIA). Faculté polytechnique de Mons.
Ref: [BOITE87] Lieu Com:

[ZELIN77] ZELINSKI, R. & NOLL, P.
"Adaptative transform-coding of speech signals" *IEEE Trans. ASSP-25,*n°4, août 1977, pp. 299-309.
Ref: [BOITE87] Lieu Com: Adaptative Transform Coding (ATC)

[_____93] *****
"Transformée de Fourier rapide par quart de période"
Traitement du Signal, Vol. 10, n°4, 1993
Ref: Lieu: Bibliotèque ENSEA . Com: Autre algorithme pour la FFT.

[_____93] *****
"Transformée de Fourier discrète récursive avec fenêtre experimentelle dissymétrique"
Traitement du Signal, Vol. 10, n°2, 1993
Ref: Lieu: Bibliotèque ENSEA . Com: Autre algorithme pour la FFT.

[_____93] *****
"Low bit-rate MPC codes with adaptively controlled synthesis filter parameters"
Signal Processing, Vol. 35, n°3, fevrier 1994
Ref: Lieu: Bibliotèque ENSEA. Com: Autre algorithme pour MPC.

[_____93] *****
"Logarithm Pruning of FFT Frequencies"
IEEE Transactions on Signal Processing, Vol. 41, n°3, mars 1993
Ref: Lieu: Bibliotèque ENSEA. Com: Autre algorithme pour la FFT. En plus, il est en langage C.

[____89] ******
"Clear Speech: Pronunciation and listening comprehension" 1989.
Ref: Lieu: UPV [D-IDM 667 1114 1147 1710] Com:

B.1. Publications periodiques :

Les revues ou conférences suivantes ont été la source et le champ d'exposés de nombreux travaux en relation avec le traitement de la parole. Aussi ceux-ci peuvent être la source de divers travaux ou recherches futures.

- Traitement de la parole / Acoustique :

  « IEEE Transactions on Speech and Audio Processing » *(Depuis Janvier 1993)*

  « IEEE Transactions on Acoustics, Speech and Signal Processing (ASSP) » *(Jusqu'à Décembre 92)*

  « IEEE Transactions on Audio and Electroacoustics »

  « ICASSP. The IEEE International Conference on Acoustics Speech and Signal Processing »

  « EUROSPEECH European Conference on Speech Communication and Technology »

  « AVIOS American Voice I/O Society Conference »

  « Journal of Acoustic Society of America »

  « Conference of Speech Communication and Processing »

  « Speech Technological Magazine »

  « Phonetica »

  « Speech Technology »

  « Journal of Speech and Hearing Research »

  « Revue d'Acoustique »

  « Journal of the Audio English Society »

  « Journal of the Acoustic Society of Japan »

  « International Congress of Acoustics »

  « Transaction Commitee on Speech Research, Acoustic Society of Japan »

- Traitement du signal / Communications :

  « IEEE Signal Processing Magazine» (Depuis Janvier 1993)

  « IEEE Transaction on Communications »

  « Journal on Selected Areas on Communication »

  « Traitement du Signal »

  « Signal Processing »

  « European Transaction on Telecommunication »

« International Conference on Digital Processing and Signals in Communication »

- Informatique :

  « DCC Data Compression Conference »

  « Theoretical Computer Science »

  « IEEE Transactions on Computer Science »

  « IEEE Transactions on Information Theory »

  « Microsoft Press »

  « Math of Computation »

  « Pattern Recognition »

  « Journal of the ACM »

  « Communications of the ACM »

- Electronique :

  « ICCAS Intl. Conference on Circuits and Systems » (IEEE)

  « Analog Devices Technical Reference Books »

  « Electronic Letters »

  « IEEE Transactions »

  « Proceedings of IEEE »

  « Electron »

Annexe C

Adresses Internet

Il faut souligner une des sources les plus importantes de cette recherche : l'Internet.

L'Internet, quoiqu'il soit utilisé plus lucrativement les dernières années, est encore le milieu pour la communication de chercheurs au tour du monde. Je me suis souscrit dans nombreuses mailing-lists sur la parole, la compression, les cartes du son et j'ai maintenu du courir avec des différentes entités de recherche et, comme même des amis en Espagne.

Mon adresse Internet a été pendant mon séjour à l'ENSEA:

```
ploro@ensea.fr
```

On est trouvé les suivants groupes de nouvelles (NG: News Groups) et fichiers FAQ (Frequently Asked Questions):

Matière	NG	Documents rélévants	Responsable
Compression	comp.compression	« comp.compression » FAQ	Jean-loup Gailly
Compression	comp.compression.research		
Traitement du Signal	comp.dsp		
Cartes son PC	comp.sys.ibm.pc.soundcard	« comp.sys.ibm.pc.soundcard » FAQ « Generic IBM PC Souncard » FAQ	Morgan Stair Joel Plutchak
Parole	comp.speech	« comp.speech » FAQ	Andrew Hunt
Son	alt.binaries.sounds.d		
Son	alt.binaries.sounds.misc		
Formats de fichiers audio		« Audio File Formats » FAQ	Guido van Rossum
Carte SB AWE 32		« Sound Blaster AWE 32 » FAQ	
Standard		« Standards » FAQ	
Utilisation FTP		« Anonymous FTP List » FAQ	

Puisqu'il a été la source fondamentale de ce projet, j'ai d'organiser tout l'embrouillement d'adresses dans le tableau suivant:

Matière	Adresses Internet	Comment contacter	Organitasion Responsable	Repertoires et fichiers

				outils
Cartes du son	morgan@DL5000.bc.edu	mail subject: FAQ	Morgan Stair	../PCsoundcards/soundcard-faq/..
Cartes du son PC	plutchak@porter.geo.brown.edu		Joel Plutchak	
Parole	andrewh@speech.su.oz.au		Andrew Hunt Speech Tech. Research Group Dept. of Electrical Engineering Univ. of Sydney NSW, 2006, Australia	
Formats des fichiers audio	guido@cwi.nl		Guido van Rossum CWI Amsterdam	../audio-fmts/..
Microsoft Format de fichier de son .VAW	ftp.microsoft.com 198.105.232.1	anonymous ftp +	Microsoft	/developer/MSDN/CD8/.. /developer/MSDN/FAQ/.. ../RIFFNE.ZIP
32 kbps ADPCM (Logiciel source en C) Intel's DVI format (v. 1.1 sans erreurs) CCITT G.721 Format de fichier de son .VAW	ftp.cwi.nl info4u.cwi.nl 192.16.191.128	anonymous ftp	Jack Jansen Rob Ryan	/pub/.. ../adpcm.shar ../audio/ccitt-adpcm.tar.Z ../audio/RIFF-format
Compression	jloup@chorus.fr		Jean-loup Gailly	
Compression wavelet (logiciel)	gdr.bath.ac.uk gpn@maths.bath.ac.uk	anonymous ftp		/pub/masgpn/wavethresh2.2.Z
Compression wavelet (logiciel)	cml.rice.edu	anonymous ftp		/pub/dsp/software/rice-wlet-tools.tar.Z
Algorithmes DCT	etro.vub.ac.be chchrist@etro2.vub.ac.be	anonymous ftp mail	Charilos Christopoulos	/pub/DCT_ALGORITHMS
Compression fractal (UNIX et MSDOS)	lyapunov.ucsd.edu	ftp		/pub/young-fractal/.. ../unifs10.zip (1.2 Mb) ../yuvpak20.zip (nouvelle version)
Cartes du son PC	ftp.uni-passau.de 132.231.1.10	anonymous ftp +		/mount/archive.theory/answers/comp.sys.ibm.pc.soundcard/..
Cartes du son PC	ftp.elelab.nsc.co.jp 133.179.207.40	anonymous ftp +		/pub/netnews/comp.archives/auto/comp.sys.ibm.pc.souncard/..
Cartes du son PC	ftp.cs.ruu.nl 131.211.80.17	anonymous ftp *		/pub/midi/doc/.. /MIDI/DOC/archives/..
Cartes du son PC	ftp.cis.nctu.edu.tw 140.113.204.21	anonymous ftp +		/MIDI/DOC/.. /Documents/soundformats/..
Cartes du son PC	athene.uni-paderborn.de 131.234.2.32	anonymous ftp *		/news/comp.archives/auto/comp.sys.ibm.pc.soundcard/..
Cartes du son PC	rs1.rrz.uni-koeln.de 134.95.100.208	anonymous ftp *		/usenet/comp.archives/auto/comp.sys.ibm.pc.soundcard/..
Cartes du son PC	ftp.kddlabs.co.jp 192.26.91.15	anonymous ftp *		/usenet/comp.archives/comp.sys.ibm.pc.soundcard/..
sb+adlib	ftp.mcs.kent.edu 131.123.2.222	anonymous ftp +		IL N'Y A RIEN
sb-freedom	nic.funet.fi 128.214.6.100 128.214.248.6	anonymous ftp *		
sb-prog	ftp.cco.caltech.edu 131.215.148.151	anonymous ftp +		/pub/heath/sb.. BEAUCOUP
faqs (USENET)	rtfm.mit.edu 18.70.0.209	anonymous ftp		/pub/usenet/comp.sys.ibm.soundcard/.. ../news.answers ../audio-fmts ../dsp-faq ../comp_speech-faq ../standards-faq
son	garbo.uwasa.fi 128.214.87.1	anonymous ftp		/pub/compspeech/info
ms-dos (de tout)	wuarchive.wustl.edu	anonymous ftp+		/mirrors/msdos/.. /pub/MSDOS_UPLOADS/sblaster/.. ../newsbkmk.zip ../awe32a.zip ../aweait.lst
	ftp.earthlink.net (198.68.160.2)	anonymous ftp (BIEN)		/pub/software/PC/ ../winzip/winzip5B.zip /users/jay/AWE32/vienna
SoundBlaster Freedom Project	wench.exe.jcu.edu.au 137.219.19.20 jeff@wench... ceii@wench...		Jeff Bird James Cook University of North	BIEN /pub/sbf /doc /src

			Queensland	
Creative Labs	ftp.creaf.com 198.95.32.3	anonymous ftp *		
Logiciel fft	usc.edu	anonymous ftp		/pub/C-numanal/.. ../fft-stuff.tar.gz
IBM Sound Mailing List	listserv@brownvm.brown.edu	mailing list: Body: subscribe IBMSND-L ploro		BIEN
Soundblaster Programmers	listserv@porter.geo.brown.edu	mailing list: Body: subscribe blaster ploro		DISCONTINUED 25 Août 94
ECTL (Electronic Communal Temporal Lobe) Parole	ectl-request@snowhite.cis.uoguelph.ca	anonymous ftp ou mailing list: Body: <nom, inst, dep; tel; e-mail>	David Leip	/pub/ectl/.. BIEN
foNETiks Parole	mailbase @mailbase.ac.uk	mailing list: Body: join fonetiks <prenom nom>	David Leip	/pub/ectl/..
CSRE: Canadian Speech Research Environment (Logiciel commercial pour PC)	march@uwovax.uwo.ca ramji@uwovax.uwo.ca	mail	Kristyna Marciniak	
OGI Speech Tools (Logiciel en C pour UNIX)	speech.cse.ogi.edu 129.95.x.51 pour x= 52BIEN, 42,40,44BIEN,46,50 tools@cse.ogi.edu	anonymous ftp mail		/pub/tools/..
Khoros (Logiciel shareware pour processament du signal)	pprg.eece.unm.edu 192.31.154.1 REJECT 129.24.24.10 TIME-OUT	anonymous ftp		
MATHVIEW for Windows 32 BEAUCOUP DE TOUT!	ftp.cica.indiana.edu 129.79.26.27 DENIED oak.oakland.edu 141.210.10.117 BIEN wuarchive.wustl.edu 128.252.135.4 DENIED	anonymous ftp		pub/.. ../dos/.. ../win3/sound ../cool131a ../fmsnd10 ../goldwave
NEVOT (Logiciel source de AT&T pour Sun)	gaia.cs.umass.edu 128.119.40.186 EXCEL hgs@research.att.com	anonymous ftp	Henning Schulzrinne	/pub/hgschulz/nevot/..
AUDIO FILE sp (library of routines for reading and writing audio files and applying filters)	alderaban.EE.McGill.CA 132.206.1.18 EXCEL 132.206.70.1 132.206.69.2	anonymous ftp	Peter Kabal McGill University Montreal	pub/AFsp/atimec
VOCPACK. Compression lossless pour fichiers .VOC et .VAW	oak.oakland.edu 141.210.10.117 ser1509@cdc835.cdc.polimi.it	anonymous ftp mail	Nicola Ferioli	/pub/msdos/sound/vocpak20.zip
faqs (de tout)	svr-ftp.eng.cam.ac.uk 129.169.24.20	anonymous ftp		/pub/compspeech/info/sources BEAUCOUP
shorten (Compression d'audio) (lossless) (Logiciel de compression pour UNIX-DOS) (lossy)	svr-ftp.eng.cam.ac.uk ajr@dsl.eng.cam.ac.uk	anonymous ftp	Tony Robinson	/pub/comp.speech/sources/.. ../shorten.tar.Z ../shorten-1.14.tar.Z ../shn114.zip ../misc/.. ../shorten-1.08.tar.Z
International Telecommunication Union (ITU) Compression G.711/721/723 (Logiciel)	teledoc@itu.arcom.ch BIEN svr-ftp.eng.cam.ac.uk 129.169.24.10 TIME-OUT	mail Body: GET ITU-3022 anonymous ftp		/pub/comp.speech/sources/.. BIEN ?
Reconnaissement de la parole (Article)	svr-ftp.eng.cam.ac.uk	anonymous ftp		/pub/comp.speech/info/.. ../DIY_SpeechRecognition
LPC codeur pour ADSP 2105	ftp.uu.net 192.48.96.9 TIME-OUT	anonymous ftp		/tmp/lpc-paper.tar.gz
GSM 06.10 Compression	ftp.cs.tu-berlin.de 130.149.144.4	anonymous ftp		/pub/local/kbs/tubmik/gsm/gsm-1.0.4.tar.Z

(Logiciel source en C pour UNIX) (questions)	130.149.24.7 BIEN MAIS TRES LENT 130.149.26.7 130.149.18.7 tub.cs.tu-berlin.de toast@cs.tu-berlin.de	 anonymous ftp mail	 	/ddj/gsm-106.zip /pub/tubmik/gsm-1.0.4.tar.Z
ASPEC - MPEG couche 3 Compression	jutta@cs.tu-berlin.de cabo@.cs.tu-berlin.de mskuhn@immd4.informatik.uni-erlnagen.de	mail mail mail	Jutta Degener Carsten Bormann KBS group Technische Universitaet Berlin Markus Kuhn Frauenhofer Institut in Erlangen	
Logiciel Audio MPEG Pour Sun Pour WINDOWS Codeur / Decodeur	sunsite.unc.edu 198.86.40.81 BIEN	anonymous ftp		/pub/electronic-publications/IUMA/audio-utils/.. ../mpeg-players:Workstations/maplqy1_2.tar.Z ../mpeg-players/Windows/mpgaudio.zip /source/mpgaudio.tar.Z
ISO/MPEG1 couche 3 (logiciel source pour fichiers .VAW) (codeur) (decoudeur: UNIX MSDOS)	fhginfo.fhg.de (153.96.1.4) EXCEL popp@iis.fhg.de	anonymous ftp mail	 Harald Popp	/pub/layer3/.. ../public_c/.. ../l3v100n.zip ../public_c/.. ../mpeg18iis.tar.Z ../mpeg&iis.zip
Vector Quantization (CELP) Logiciel	cochlea.hut.fi 130.233.168.48 BIEN			/pub/.. ../lvq_pak
Compression CELP (Logiciel source en C pour Sun SPARCs)	furmint.nectar.cs.cmu.edu 128.2.209.111 BIEN super.org 192.31.192.1 REJECT	anonymous ftp anonymous ftp		/celp.audio.compression /pub/celp_3.2a.tar.Z
G.728 LD-CELP vocoder pour le ADSP-2171 (Logiciel commercial) U.S.F.S. 1016 CELP vocoder pour le DSP56001	cole@analogical.com	mail	Cole Erskine Analogical Systems	
Introduction à la Vector Quantization	alex.zatsman@analog.com alex.sp.cs.cmu.edu 128.2.209.13	mail anonymous ftp	Alex Zatsman	 /links/audio
G.728 Compression (Logiciel)	dspsun.eas.asu.edu 129.219.39.173 (User anonymous device)	anonymous ftp	Alex Zatsman Analog Devices, Inc	/pub/speech/ldcelp.tgz
Compression Audio QCELP (Document CAI: Common Air Interface)	lorien.qualcomm.com	anonymous ftp	Bob Kimball Qualcomm Inc.	/pub/cdma (postcript file) (voir seulement Append. A)
SQUISH (Paquet d'outils de compression d'Audio) (bon pour échantillonnages hautes)	DeskFish.mit.edu 18.244.0.41 TIME-OUT	anonymous ftp		/pub/compression_apps/.. .

Heureusement, aujourd'hui il y a beaucoup d'outils dans l'Internet pour obtenir information d'une manière plus simple. Pour donner quelques exemples, j'ai utilisé les

archies, les bases de données, le ressoudreur d'adresses, et toutes ces choses communs dans le mon Internet[1].

[1]Aussi, deux manuels sur l'Internet ont été extraits de l'Internet lui-même : [KEHOE92] et [GAFFI94].

Annexe D

Glossaire de Termes et d'Acronymes

Les termes les plus couramment utilises dans ce rapport et dans le traitement de la parole en général sont :

A-Law	Standard Européen PCM de la téléphonie
ADM	Adaptive Delta Modulation
ADPCM	Adaptive Differential Pulse Code Modulation
ADSP	Analogical/Digital Signal Processor
ADxyM	Adaptive Delta X Y Modulation
APC	Adaptive Predictive Coding
APCM	Adaptive Pulse Code Modulation
A/N	Analogique / Numérique
ASIC	Application Specific Integrated Circuit
ASPEC	Format Allemand de codage
ASSP	Acoustics Speech and Signal Processing
ATC	Adaptive Transform Coding
AWE-32	Une version de la carte son Sound Blaster
BARTHE	Société dediée à la réalisation d'appareils audio pour l'apprentissage des langues
Bandwidth	Largeur de bande
bps	bits per seconde
BCELP	Binary Codebook Excited Linear Prediction
CAN	Convertisseur Analogique Numérique
CCITT	Comité Consultative International de la Téléphonie et la Télégraphie
CD	Compact Disque
CELP	Code-book Excited Linear Prediction
Cepstre	SPECtre inverse (logarithmique)
CNA	Convertisseur Numérique Analogique
Codage	Processus de mettre certains valeurs univoquement dans autres
CODEC	CODer / DECoder device
Codebook	Tableau pour trouver les correspondance entre codes et valeurs
COMPAND	COMPress / exPAND
Compression	Processus de codage pour réduire le débit de stockage ou transmission
CPU	Central Process Unit
CVSD	Constant Variably Slope Delta
DAT	Format de cassette numérique
dB	decibelles
DCT	Discrete Cosine Transform

Débit	Taux d'information per unité de temps
DFT	Discrete Fourier Transform
DLL	Dynamic Link Library
DOLBY	Système Reducteur de bruits
DPCM	Differential Pulse Code Modulation
DSK	DSP Starter Kit
DSP	Digital Signal Processor
DTW	Dynamic Time Warping
ECS	Équipe de Commande des Systèmes
e-mail	Electronic Mail (courir électronique)
EasyWin	Applications simplifiées pour WINDOWS
ENSEA	École Nationale Supérieure de l'Électronique et des ses Applications
ERASMUS	Programme d'interchanges d'établisement européens
FAQ	Frequently Asked Questions
FFT	Fast Fourier Transform
FIR	Finite Impulse Response (Filtres)
Freeware	De distribution et utilisation libres
Full-duplex	Communication synchronement bidirectionnelle
GSM	Groupe Spéciale Mobile
HIDM	High Information Delta Modulation
Huffman	Code pour comprimer de données profitant des redondances
hz	Hertz
iDFT	Inverse DFT
IEEE	Institute of Electrical and Electronics Engineers
iFFT	Inverse FFT
IIR	Infinite Impulse Response
Internet	Réseau de réseaux basé sur le protocol TCP/IP
iWHT	Inverst WHT
JPEG	Joint Photograph Expert Group
kbps	kilo-bits per second
Khz	kilo-hertz
KLT	Karhunen-Loève Transform
Kbit	Kilo-bit
Koctets	Kilo-octet
LALI	Laboratoire Autonome des Langues Individuel
LLI	Laboratoire des Langues Individuel (nom définitif du précédent)
LD-CELP	Low Delay CELP
LogPCM	Logarithmic Pulse Code Modulation
Lossy	Avec perte
LPC	Linear Predictive Coding
LSI	Low Scale Integration
LSB	Least Significant Bit
LVQ	Learned Vector Quantisation
LVQ	Linear Vector Quantization
LZ77	Lempel & Ziv 1977 Compression
LZ78	Lempel & Ziv 1978 Compression
LZW	Lempel Ziv & Welch Compression
Mhz	Mega-Hertz
MIPS	Millions d'Instructions Par Seconde
Modulation	Processus de modification de la dimension où l'information est transmise.
MOS	Mean Opinion Score
MPEG	Motion Picture Expert Group
MSB	Most Significant Bit
N/A	Numérique / Analogique
NR	Noise Reduction
ns	Nano-Seconds
Numérisation	Processus de convertir à un format discret
ObjectWindows	Librairie de Borland orienté objets pour WINDOWS
PCM	Pulse Code Modulation

Pentium	80586
Pitch	Fréquence du fondamental, couramment connu comme timbre de la voix
Quantification	Processus de approcher une valeur continue à un nombre réduit de niveaux
RAM	Random Access Memory, Mémoire Vive
RIFF	Format Multimédia de Microsoft
ROM	Read-Only Memory, Mémoire Morte
SB	Sound Blaster
SBC	SubBand Coding
Shareware	De distribution libre et utilisation avec licence
SNR	Signal to Noise Ratio
SNRq	SNR eQuivalent
SNRseg	SNR SEGmental
Sonagramme	Representation du spectre respecte du temps
Spectre	Representation de l'amplitude selon les fréquence
TDHS	Time Domain Harmonic Scaling
TFR	Transformée de Fourier Rapide
UPV	Universidad Politécnica de Valencia
VQ	Vector Quantization
VOCODER	VOice CODER
WAV	Extension du format de son de Microsoft
WAVE	Format de son de Microsoft
WHT	Walsh-Hadamard Transform
WIN16	Application WINDOWS sur 16 bits
WIN32	Application WINDOWS sur 32 bits
WINDOWS	Système d'explotation orienté à fenêtres de Microsoft pour PCs
X-Windows	Environnement orienté à fenêtres sur UNIX
ZIP	Format de compression très utilisé pour les ordinateurs
μ-Law	Standard Européen PCM de la téléphonie

Annexe E

Équipement

L'équipement utilisé dans ce travail a été en part subventionné par la société BARTHE et le reste apporté par le laboratoire ECS de l'ENSEA. Donc, on n'a eu aucun probleme de milieux (seulement le temps, toujours le temps) pour faire ce projet.

E.1. Équipement requis :

Le matériel requis pour ce projet est un ordinateur avec une carte de son pour l'acquisition et la restitution de la parole. Additionnelement, on utilisera un reproducteur de cassettes, un microphone et des enceintes.

En plus, nous avons besoin de logiciels comme les librairies de programmation de la carte, le compilateur C/C++, un éditeur pour faire toute la documentation nécessaire et bien sûr, le système d'exploitation.

E.1.1. L'ordinateur :

Les nouveaux algorithmes à étudier nécessitent une capacité de calcul suffisante pour voir son évolution en temps réel. Donc, la configuration la plus appropriée est un Pentium à 60 MHz avec 8 Mo de mémoire, 16 Mo étant préférable puisque les applications pour WINDOWS exigent de grandes quantités de mémoire et disposant d'un disque dur de plus de 300 Mo pour l'installation de tous les logiciels et pour l'enregistrement du son. L'existence du bus P.C.I. du Pentium est intéressante puisque ceci libère le bus des interférences d'autres passages de données (comme l'écran, le disque dur, etc.).

E.1.2. La carte son :

Bien que l'utilisation de la carte soit fondamentalement d'acquisition et restitution, on achètera une carte de son en fois d'une carte d'acquisition de données. Les raisons fondamentales sont les suivantes:

- Une application de la parole n'a besoin de caractéristiques techniques impressionnantes.
- L'installation, le logiciel et l'information sont plus faciles.
- La possibilité d'entrée directe de CD-ROM.
- La variété sur PCs est considérable.
- Le prix.

On fera une recherche documentaire des cartes du son présentes sur le marché. La source d'information la plus importante est un article de la revue INFOPC d'octobre 1994, lequel fournit une comparaison des cartes son actuelles, l'information de *news* distribuée sur Internet par le *"comp.sys.ibm.pc.soundcard" groupe* dans son fichier FAQ (Frequently Asked Questions) est également fondamentale, comme autres revues comme Puissance Micro, BYTE (Septembre 1994) et toute la publicité qu'est arrivé à mes mains ont été une valable aide pour la décision final.

Les cartes pour ordinateurs personnels ont évolué considérablement dés l'apparition du premier standard Adlib 8 bits de fréquence basse d'échantillonnage. Creative Labs a fait la Sound Blaster 1.0 qui est fondamentalement une Adlib avec DAC, synthétiseur, etc.

Postérieurement sont apparues d'autres cartes 8 bits avec de meilleures qualités d'enregistrement comme la Sound Blaster 1.5, 2.0 Deluxe. Convertie en standard, d'autres cartes sont venues comme la PAS (compatible 100 %) et la Sound Blaster Pro (non compatible 100 %).

Les cartes 12 bits comme les Sound Blaster 16 et 14 bits comme la G.U.S. précédent les cartes 16 bits réels qui sont maintenant les plus courantes.

Les caractéristiques les plus importantes à évaluer sur une carte sont:

- Compatibilité: C'est la partie la plus important parce qu'il y a beaucoup de logiciels déjà développés pour la carte "Adlib", "Sound Blaster" et "Sound Blaster Pro". Pour une possible application sur WINDOWS, il serait recommandable d'avoir une compatibilité avec la "Microsoft Sound System".
- Fréquence et niveaux d'échantillonnage: Le minimum requis pour ce projet est 10 KHz, ce qui est accompli par toutes les cartes analysées. Normalement, les fréquences d'environ 41, 44.1, 48 (qualité DAT) jusqu'à 51 KHz sont fournies avec ces cartes. Les quantifications varient depuis 256 niveaux fournis par les cartes 8 bits jusqu'aux 65536 niveaux résultants des 16 bits. Aussi, il est très intéressant que cette fréquence soit programmable et, si possible, réglables de façon linéaire.
- Entrées et Sorties disponibles: Nous avons besoin de deux entrées (préférablement une voie pour la cassette et l'autre pour le micro) et une sortie. Normalement les cartes portent deux sorties (une voie ligne et l'autre amplifiée à 4 ou 5 W).

- DSP (Digital Signal Processor): Quoique le traitement du signal dévoie être étudié en ce travail, l'existence du DSP nous peut servir pour faire une comparaison entre la compression via matériel et la compression via logiciel.

- Facteurs de qualité d'enregistrement et de restitution: Il serait intéressant d'avoir une bonne courbe de réponse ainsi qu'un bon rapport signal à bruit. Celui-ci varie autour de 60 à 80 dB.

- Interface CD-ROM: L'existence de l'appareil EDU-CD de BARTHE et l'évolution croissante de ce format peut rendre intéressante l'application de la sortie numérique issue du C.D. Comme extension à ce travail, on pourrait ajouter un lecteur de CD-ROM et faire le même qu'avec la cassette mais utilisant directement l'entrée numérique sans nécessité de l'échantillonnage ni la conversion A/N. Les trois interfaces habituelles sont Mitsumi, Sony et Panasonic.

Autres caractéristiques qui ne sont pas indispensables pour ce projet mais font augmenter le prix sont:

- Interfaces: MIDI, SCSI, Roland, Joystick...

- Autres facteurs de qualité d'enregistrement et de restitution: La bande passante fournie arrive généralement à 22 kHz (qualité hi-fi), plus que suffisante pour une application en parole. Le nombre et la séparation des canaux ne sont pas du tout important parce que nous travaillons avec un unique signal monophonique.

- Outils fournis et équipements supplémentaires: Il y a des cartes qui disposent d'extensions (Synthétiseurs, modulateurs de fréquence (F.M.), Wave Tables, Mixers...), jeux, effets sonores, librairies de sons et autres, complètement superflus pour cette application.

Finalement, nous avons à faire la comparaison et le classement par le facteur prix (se référant au marché d'octobre 1994, en France).

Jusqu'à 500 F:

- **Sound Master Boomer** (290 F): Avec composants bons marchés, c'est la carte la moins cher compatible avec la Sound Blaster 2.0. On n'a pas d'autres références précises.

- **Sound Blaster 1.0, 1.5, 2.0 Deluxe** (v. 2.0: 450 F): L'échantillonnage est sur 8 bits et sa fréquence de 44,1 kHz mono. Le rapport signal à brut est de 50 dB. Les entrées sont habituelles mais il y a seulement une sortie. La compatibilité Adlib est totale.
  Autres caractéristiques: un synthétiseur avec un F.M. à 11 voies.

- **Sound Galaxy BX II, NX FP, NX Pro** (390F, 490F et 570F respectivement): Elles sont cloniques des authentiques Sound Blasters de 8 bits faites par une compagnie chinoise. La qualité est moins bonne en raison des composants utilisés.

Gamme 500 F - 1000 F:

- **Sound Blaster Pro 2 NG** (550 F): L'échantillonnage est sur 8 bits et sa fréquence de 22.05 kHz en stéréo et 44,1 kHz mono. Le rapport signal à brut est de 55 dB. Les entrées sont normales mais il y a seulement une sortie. La compatibilité avec les Sound Blasters antérieures et Adlib est excellente.
  Autres caractéristiques: un synthétiseur avec un F.M. à 22 voies.

- **Media Vision Pro Audio Spectrum/Studio (PAS) 16** (650F): L'échantillonnage est sur 16 bits et sa fréquence est programmable jusqu'à 44,1 kHz. Les entrées sont normales mais il y a seulement une sortie. La compatibilité avec Sound Blaster, Sound Blaster Pro et Adlib est matérielle et excellente. En plus, il y a beaucoup de clubs d'usagers et programmateurs de cette carte.
  Autres caractéristiques: un synthétiseur avec un F.M. à 20 voies, Midi compatible MPU-401. La version studio existe avec plus de logiciels et interface SCSI.

- **Mozart MICDIS Sound Pro 16** (690 F): Les uniques caractéristiques connues sont: les trois interfaces CD-ROM normales et la compatibilité avec Adlib, Sound Blaster Pro et Ms Windows Sound System.

- **Sound Galaxy NX PRO 16 Basic** (770 F): Clonique de la vraie Sound Blasters 16 Basic faite par une compagnie chinoise. Donc, la qualité est moins bonne.

- **Logitech Soundman 16** (790F): C'est exactement la même carte que la PAS 16 sans l'interface CD-ROM. La nouvelle version est la Logitech Soundman Wave.

- **Media Vision Jazz 16** (799 F): Fille de la PAS, c'est une des cartes de cette gamme avec le meilleur rapport qualité-prix. L'échantillonnage est sur 16 bits et sa fréquence est de 44,1 kHz mais nous n'avons pas d'information sur sa programmation. Le rapport signal à brut est de 66 dB. Les entrées sont normales mais il y a seulement une sortie amplifiée (il n'y a pas de sotie en ligne) et une interface Mitsumi pour CD-ROM. La compatibilité avec Sound Blaster, Sound Blaster et Adlib est matérielle, c'est-à-dire: le minimum pour ce projet.
  Autres caractéristiques: un synthétiseur avec un FM à 20 voies, Midi compatible MPU-401.

- **Sound C.D. 16** (818,34 F): L'absence de caractéristiques superflues fait de celle-ci une des cartes avec un très bon rapport qualité-prix. L'échantillonnage est sur 16 bits et sa fréquence est programmable de 5 à 48 kHz utilisant un convertisseur CODEC Analog Device AD1848KP "SoundPort" (qualité D.A.T. à cette gamme). En plus, le rapport signal à bruit est supérieur à 80 dB. Elle est aussi pourvue des entrées et sorties classiques et des trois interfaces pour CD-ROM usuels. La compatibilité avec Adlib, Sound Blaster 2.0 et Microsoft Sound System est sûre. Les interfaces MIDI, MPU-401 et la Wave-table sont optionnelles (+800 F), ce qui baisse ainsi le prix global du produit.
  Autres caractéristiques: un synthétiseur avec un FM à 22 voies stéréo, mixeur et beaucoup de logiciels et programmes pour effets comme écho, fondus, etc.

- **Media Vision Pro Audio Spectrum (PAS) 16 XL**(). Une version nouvelle de la PAS avec échantillonnage à 48 kHz. Les entrées sont les normales mais il y a seulement une sortie. La compatibilité avec PAS, Sound Blaster, Sound Blaster Pro et Adlib est matérielle et excellente.
  Autres caractéristiques: un synthétiseur avec un F.M. à 32 voies, Midi compatible MPU-401, Interface SCSI et 4 Mo de ROM.

- **Maxi-Sound Solution 2.0, PRO, 16** (670F, 890F, 990F): On n'a pas de références précises.

- **Microsoft Windows Sound System** (890 F): L'échantillonnage est sur 16 bits et sa fréquence est de 44,1 kHz. On ne sait pas si celle-ci est programmable. Le rapport signal à brut est de 70 dB. Les entrées et sorties sont normales mais celle-ci manque d'interface pour CD-ROM. La compatibilité est seulement avec la Sound Blaster et par émulateur. La qualité d'enregistrement et de restitution est très bonne pour son prix.
  Autres caractéristiques: Aucune sauf les bons logiciels et programmes de la Microsoft Windows Sound System.

Gamme 1000 F - 1500 F:

- **Advanced Gravis UltraSound Max (G.U.S. Max)** (1100 F): C'est la nouvelle version de la classique GUS 8 bits à 41.1 kHz. L'échantillonnage est maintenant sur 16 bits (seulement 14 significatifs) et sa fréquence est de 48 kHz mais nous n'avons pas d'information sur sa programmation. Le rapport signal à bruit est de 68 dB. Les entrées et sorties sont normales et les trois interfaces pour CD-ROM usuels sont présentes (Il y a une version SCSI aussi). Des compressions (4:1) ADPCM, A-Law et μ-Law sont disponibles via hardware. La compatibilité est problématique avec la Sound Blaster et Adlib via emulateurs T.S.R. mais on dispose de le "G.U.S. Software Development Kit" et de toutes les applications faites pour la G.U.S. très étendue.
  Autres caractéristiques: un synthétiseur, un F.M. à 32 voies avec 16 canaux, compatibilité Roland MT-32, wave table, Midi avec 512 Ko de mémoire et beaucoup de logiciels et programmes pour effets comme écho, fondus, etc.

- **Orchid SoundDrive 16** (1180 F): L'échantillonnage est sur 16 bits et sa fréquence est programmable jusqu'à 44,1 kHz mais nous n'avons pas d'information sur la manière de la programmer. Le rapport signal à bruit est de 71 dB et la compression ESPCM. Les entrées et sorties sont les normales mais il y a seulement une interface Mitsumi pour CD-ROM ou un SCSI dépendant de la version. La compatibilité avec Sound Blaster, Adlib, Microsoft Windows Sound System et MPC 1 et 2 est matérielle. Mais d'autres cartes de la gamme antérieure ont les mêmes caractéristiques, voire meilleures.
  Autres caractéristiques: Midi, MPU-401, MT32.

- **Sound Blaster 16 (Value Edition-Basic, Multi-CD, SCSI) (ASP)** (Basic 750F; Multi-CD sans/avec ASP: 970F/1110F; SCSI sans/avec ASP: 1250F/1470F): L'échantillonnage est sur 16 bits (encore que seulement 12 soient significatifs) et sa fréquence est de 44,1 kHz. On

ne sait pas si celle-ci est programmable. Les entrées et sorties sont habituelles. L'interface pour CD-ROM dépend de la version; la version Basic a les interfaces CreativeLabs et Panasonic; la version MULTI-CD a en plus, les interfaces Sony et Mitsumi. Il y a aussi une version SCSI. La compatibilité est presque totale avec les Sound Blaster antérieures (quelques programmes mal faits posent problèmes). Avec la Sound Blaster Pro, la problématique est encore pire. La qualité d'enregistrement et de restitution est très bonne pour son prix.
Autres caractéristiques: un synthétiseur avec un FM à 11 ou 20 voies, Midi compatible MPU-401 mais avec quelque erreur. Les options ASP incluent une chip CSP pour compression/décompression que peut être programmé (trois techniques sont disposées gratuitement). Une autre option est la carte fille **Wave Blaster** (1370 F) qui ajoute 32 voies, 128 MIDI instruments, 4 Mo de ROM et beaucoup d'autres choses en plus.

- **Orchid SoundWave 32** (1190 F): L'échantillonnage est sur 16 bits et sa fréquence est de 48 kHz utilisant un convertisseur CODEC Analog Device AD1848, programmable de 2 kHz jusqu'aux 48 kHz maximum. Les entrées et sorties sont habituelles avec les trois interfaces pour CD-ROM.. La compatibilité est totale avec le Microsoft Windows Sound System, Adlib et Sound Blaster. Les compressions sont ADPCM (2:1, 3:1 ou 4:1) utilisant un DSP Analog Devices ADSP 2115 à 20 MHz. Malheureusement, la courbe de réponse est assez perturbée.
  Autres caractéristiques: MPU-401 et Midi MT-32 et beaucoup de logiciels et programmes.

- **Paradise 16 DSP** (990F, Pro: 1390 F): L'échantillonnage est sur 16 bits et sa fréquence est de 44,1 kHz. On ne sait pas si celle-ci est programmable. Le rapport signal à bruit est de 70 dB. Les entrées et sorties sont normales et les trois interfaces pour CD-ROM usuels sont fournis. La compatibilité est totale avec la Sound Blaster, Sound Blaster Pro, Adlib, Microsoft Windows Sound System et MPC 1 et 2. De caractéristiques ressemblants à des cartes plus chères comme la Logitech ou l'Adaptec. La Paradise a une qualité d'enregistrement et de restitution médiocre mais suffisante pour ce travail.
  Autres caractéristiques: Midi; MPU-401.

Gamme 1500 F - 2000 F:

- **Sound Blaster AWE 32** (1790 F): C'est une des cartes ayant un très bon rapport qualité-prix. L'échantillonnage est sur 16 bits et sa fréquence est programmable de 5 à 45 kHz par 228 réglages linéaires. Elle est pourvue de DSP et le rapport signal à bruit est de 72 dB. Elle dispose aussi des entrées et sorties classiques et des trois interfaces pour CD-ROM usuels. Le grand problème est sa compatibilité avec Sound Blaster et Adlib, encore que celle est matérielle, n'est pas totale.
  Autres caractéristiques: deux synthétiseurs, un F.M. à 20 voies, wave table, Midi avec 512 Ko de mémoire et beaucoup de logiciels et programmes pour effets comme écho, mixeurs, fondus, etc.

- **ViVa Maestro 16 et 16 VR** (1750F et 2290F): On n'a pas de références précises.

Au-dessus de 2000 F:

- **Logitech Soundman Wave** (2360 F): L'échantillonnage est sur 16 bits et sa fréquence est de 44,1 kHz. On ne sait pas si celle-ci est programmable. Le rapport signal à bruit est de 70 dB. Les compressions IMA, ADPCM, A-Law et µ-Law. Les entrées et sorties sont normales mais les trois interfaces pour CD-ROM usuels sont substituées par le SCSI. La compatibilité est totale avec la Sound Blaster, Adlib et MPC 1 et 2.
  Autres caractéristiques: synthétiseurs, un F.M. à 20 voies, compatibilité MPU-401, SCSI, Midi, logiciels et programmes.

- **Adaptec AMM-1570** (2850 F): Très similaire à la précédente, l'échantillonnage est sur 16 bits et sa fréquence est de 44,1 kHz. On ne sait pas si celle-ci est programmable. Le rapport signal à bruit est de 72 dB. Les entrées et sorties sont les normales mais les trois interfaces pour CD-ROM usuels sont substituées par le SCSI. La compatibilité est totale avec le Microsoft Windows Sound System et la Sound Blaster. Les compressions sont ADPCM et TrueSpeech ADPM utilisant un DSP Analog Devices ADSP 2115.
  Autres caractéristiques: compatibilité MPU-401, SCSI, Midi, les logiciels et programmes de la Microsoft Windows Sound System.

- **Roland RAP 10/AT** (5090 F): L'échantillonnage est sur 16 bits et sa fréquence est programmable linéairement jusqu'à 51 kHz. Le rapport signal à bruit est de 66 dB. Celle-ci n'incorpore pas d'interface CD et n'est plus compatible avec aucune carte.
  Autres caractéristiques: synthétiseurs avec un FM à 26 voies. En plus de l'exceptionnelle qualité que justifie son prix, présence de Midi, wave table et beaucoup d'effets.

Conclusions:

L'option la plus raisonnable semble être la carte Sound C.D. 16 (818F) avec une compatibilité totale avec les deux standards que nous intéressent, la Sound Blaster et la Ms Windows Sound System. Fournie avec les interfaces pour C.D. et ses 48 kHz de fréquence d'échantillonnage, cette carte a le meilleur rapport signal à bruit de toutes (80 dB). Mais elle manque de l'existence de DSP.

Trois autres cartes qui disposent sûrement de DSP et pouvant être considérées dans ce choix sont: la G.U.S. Max (1100F) avec ses problèmes de compatibilité, l'Orchid SoundWave 32 (1190F) avec sa courbe de réponse perturbée et, comme nom, le futur standard Sound Blaster AWE 32 (1790F) avec une fréquence de 44.1 kHz.

E.1.3. Equipements supplémentaires à la carte:

Le micro et les enceintes seront demandés au distributeur de la carte. La qualité de ces appareils n'est pas trop importante pour le projet. La paire d'enceintes varie de 80 F (2 W) jusqu'à 230 F ou 350 F (25 W). Le micro est fourni normalement avec la carte et sa qualité est suffisante.

Le reproducteur de cassette peut être quelconque, mais de bonne qualité. Celui-ci sera fourni par la société Barthe.

E.1.4. Le CD-ROM:

Le CD-ROM est optionnel est peut être utilisé pour essayer le traitement direct du signal sans échantillonnage, puisque les nouveaux formats d'Audio disposant d'un stockage numérique. L'élection dépendra du plus approprié pour la carte acheté.

E.1.5. Connexion Internet:

Quoiqu'une carte de connexion pour PCs au réseau de l'ENSEA ne soit pas cher et supposerait une grande commodité pour le transfert de fichiers, on m'a ouvert un compte aux workstations des salles DAO.

E.1.6. Logiciel :

- ***Le système d'explotation :*** Pour un ordinateur Pentium, l'option la plus naturelle est la dernière version de Microsoft WINDOWS (Version 3.11 pour Workgroups). L'autre option serait de choisir Microsoft WINDOWS NT pour ce projet. Ce n'est pas une question importante parce que le reste du matériel et des logiciels est compatible avec les deux environnements.

- ***Librairies de programmation de la carte :*** En plus de celles fournies par le distributeur, beaucoup d'algorithmes et de logiciels sont disponibles en recherchant sur Internet. Sinon, il faudra l'acheter ou le développer si l'information de programmation de la carte est claire.

- ***Librairies de traitement du signal :*** Il y a beaucoup de paquets et outils pour le traitement du signal: représentation, transformées, compressions, etc. (une relation peut être réalisée). Puisqu'un des objectifs de ce travail est de les développer et en plus, on dispose de librairies « shareware » à l'Internet, un achat de ce logiciel n'est pas indispensable.

- ***Le compilateur C/C++ :*** Le Borland Turbo C++ 4.0 pour WINDOWS ou le Microsoft Visual C++ 2.0 sont les deux compilateurs les plus répandus pour WINDOWS.

- ***Éditeur de texte :*** Le Microsoft Word ou le WordPerfect for WINDOWS seraient complètement suffisants.

E.2. Équipement acquis:

Finalement, nous avons acquis le matériel et logiciel suivant:

E.2.1. L'ordinateur:

Un Pentium (architecture PCI) à 60 MHz avec 16 Mo. de mémoire, disposant d'un disque dur de plus de 400 Mo. avec une écran Sony Trinitron 15sf.

E.2.2. La carte son:

Nous avons décidu par la sécurité de la marque la plus connue: Creative Labs et sa So**und Blaster AWE 32.** Nous l'avons achete au FNAC de Cergy par sa proximité et elle s'a acouté presque 2000 F. Le microphone est inclu avec de la carte. Nous avons acheté aussi des enceintes LabTec MCS-150

Les characteristiques (seulement les concernentes à ce projet) sont:

- Canal vocal numérisé stéréo:
  - Numéritasitation 8 et 16 bits en modes mono et stéréo.
  - Taux d'échantillonnage programmable pouvant varier entre 5 kHz et 45 kHz en 228 pas linéaires.
  - Canaux DMA 8 et 16 bits utilisant une interruption unique.
  - Filtrage dynamique pour l'enregistrement et l'interpretation audio numériques.
- Mixage numérique/analogique intégré:
  - Mixe les sources voix numériséee, et les entrées périphériques MIDI, CD-audio, ligne, microphone et haut-parleur PC.
  - Possibilité de choisir une source d'entrée ou de mixer différentes sources audio pour un enregistrement.
- Contrôle du volume / physique:
  - Rapport signal à bruit est de 72 dB
  - Entrées et sorties classiques.
  - Contrôle du volume par logiciel pour le volume principal; la voix numérisée, et les entrées périphérique MIDI, CD-audio; ligne; microphone et haut-parleur PC.
  - Haut-parleur PC en 4 niveaux par pas de 6 dB.

- Toutes les autres sources en 32 niveaux par pas de 2 dB.
- Contrôle basses/aiguës en 15 niveaux de -14 dB à +14 dB pqr pqs de 2 dB.
- Contrôle total par logiciel des effets de fondu et de panoramique.

- Interface CD-ROM
  - Interface CD-ROM intégrée qui supporte les lecteurs Panasonic, SONY ET Mitsumi.
- DSP:
  - Circuit processeur de traitement de signal évolué capable d'exécuter des tâches de traitement numérique en temps réel.
- Elle est pourvue des compressions suivantes:
  - CCITT A-Law
  - CCITT μ-Law
  - IMA/DVI ADPCM
  - CTADPCM
  - Microsoft ADPCM
  - FastSpeech - 8
  - FastSpeech - 10
- Compatibilité:
  - Compatibilité totale avec Sound Blaster 16.

Cette information a été extraite du manuel d'installation. Annexe A (spécifications générales).

E.2.3. Le CD-ROM:

Finalement un CD-ROM de quadruple vitesse sera acheté mais il n'a arrivé à temps pour ce projet.

E.2.4. Connexion Internet:

Finalement, la carte Ethernet TRUST NE 2000 Plus Serie a été acquise pour le Pentium.

E.2.5. Logiciel :

- ***Le système d'exploitation :*** La dernière version de Microsoft WINDOWS (Version 3.11 pour Workgroups) a été choisie.

- ***Le compilateur C/C++ :*** Le Borland Turbo C++ 4.0 pour WINDOWS.

- ***Éditeur de texte :*** Le Microsoft Word 6.0.

Annexe F

Évolution du Projet

Périodiquement, je réalisais un petit rapport de l'état courant du projet pour éclaircir et constater cela que j'allais faisant.

F.1. Planification initiale du Projet :

Le projet a une étendue prévue de six mois (25 semaines) lesquels se distribuent initialement de la façon suivante:

- 3 semaines:
    - ◊ Recherche bibliographique sur:
        - ◊ Traitement de la parole.
        - ◊ Algorithmes de compression.
    - ◊ Étude et achat de la carte de son la plus appropriée au projet.
    - ◊ Installation de l'ordinateur, la carte et tout le logiciel requis (éditeur, compilateur C/C++, librairies de programmation de la carte, etc.).
- 3-4 semaines:
    - ◊ Réalisation des sous-programmes en langage C pour l'acquisition, la représentation temporelle et spectrale (sonagramme) et la reconstitution du signal. Pour cette tâche, il sera fait usage de la D.F.T. (Transformée de Fourier Discrète) avec différents algorithmes (F.F.T., D.C.T., K.L.T, W.H.T.).
    - ◊ Comparaison et essai des différentes fréquences d'échantillonnage (de 10 kHz jusqu'à 50 kHz ou la limite permise par la carte de son).
- 2-4 semaines:
    - ◊ Implantation des algorithmes de compression traditionnels (facteur de compression de 2, 4 voire 8):
    - ◊ Recherche de nouveaux algorithmes avec facteurs plus grands (10 ou plus).

- ◊ Détermination expérimentale du meilleur compromis entre fréquence d'échantillonnage et facteur de compression pour arriver à capacités de stockage faibles.

- 2-5 semaines:

  - ◊ Variation de la fréquence de restitution et essai des méthodes pour une restitution à basse vitesse compréhensible par une personne quelconque.

- 6-13 semaines (variable selon l'évolution des parties antérieures):

  - ◊ Implantation pratique de ces algorithmes sur un de ces composants:
    Intel µC 80C196 à 16 MHz.
    Analog Devices ADSP 2111 à 33 MHz.

- 2-3 semaines:

  - ◊ Rédaction du mémoire du projet.

  - ◊ Extraction des conclusions et évaluation des objectifs initiaux.

  - ◊ Insertion des autres idées ou améliorations découvertes pendant la réalisation de tout le projet.

F.2. État du Projet 28-11-94

À la fin de la cinquième semaine du projet, toute la partie de recherche bibliographique fondamentale a été faite: soit de la théorie (traitement de la parole et du signal, des algorithmes de compression avec/sans perte ou des transformées) soit de la pratique (programmation de la Sound Blaster AWE 32 et formats des fichiers d'Audio les plus utilisés).

Étant donné que la bibliothèque de l'ENSEA est un peu limitée, je suis en train de faire une relation des livres et publications les plus importants. Celle-ci sera utilisée pour les chercher pendant ma visite prochaine à Valence et particulièrement à l'Université Polytechnique de Valence, dont la bibliothèque et fondamentalement la hémérothèque sont plus généreuses. Aussi, je tenterai d'accéder à quelque université en mon voyage immédiat à Londres (mercredi 23 - lundi 28 novembre).

Aussi, dans l'Internet, beaucoup de ftp sites et *mailing lists* ont été localisés. On montre les adresses et d'information additionnelle à un tableau à la fin de la section bibliographique. Le délai produit en la création d'une adresse Internet à les stations DAO a provoqué que la recherche se soit prolongée jusqu'à cette semaine.

L'achat et installation de l'équipement (ordinateur, carte et tout le logiciel) sont finalement complets. J'ai rempli et envoyé tous les certificats de garantie de ceux-ci et j'ai demandé aussi à Creative Labs d'information additionnelle sur la carte. Aucune réponse a encore été reçue. Le fichier FAQ de l'Internet « Frequently Asked Questions for SB AWE32 » ne fournit pas information additionnelle puisque celui-ci est presque complètement dédié a l'interface MIDI. Dans ce document, Creative Labs annonce qu'il y aura un kit de développement pour la AWE32 gratuit prochainement. Il sera disponible au Compuserve ou à la BBS Creative.

Mais, j'ai trouvé de la programmation sur la carte SB16 et puisque la AWE32 et complètement compatible, je peux l'utiliser. Je n'ai pas encore l'essayé avec différentes fréquences d'échantillonnage.

Par ailleurs, j'ai abrégé d'information sûr les CD-ROM pour un achat futur mais il serait meilleur attendre au moment qu'il soit nécessaire pour le projet (après Noël peut-être la baisse des prix sera considérable).

Dû à une erreur de logiciel de la carte SB qui bloquée aléatoirement et fatalement WINDOWS, une semaine du travail est perdue en essayant différentes configurations de l'ordinateur et de la carte pour résoudre le problème. Après nombreux essais, la raison a été attribuée au programme « Creative Mixer » fourni avec la carte. Ce programme n'est pas essentiel pour le développement du projet, donc j'ai continué mes travaux. Plus tard, en pullulant sur l'Internet, j'ai trouvé le fichier « AWE32-patch » qui efficacement résout une partie du problème. Je suis en train de considérer d'écrire une lettre de plainte à Creative Labs.

En plus, étant donnée la grande puissance de l'ordinateur et du compilateur achetés et pour faciliter l'interaction avec le programme de simulation, je me suis décidé à le développer et l'implanter sur le système d'exploitation WINDOWS. La lecture des manuels et la familiarisation avec les librairies ObjectWindows 2.0 peuvent retarder le commencement de la deuxième partie du projet, mais l'utilisation de ces outils peut être temporairement rentable dans le futur.

Heureusement, la grande quantité d'information et de logiciel (transformés et algorithmes de compression) trouvés à l'Internet fait que la planification ne se voit pas modifié à long terme. En plus, le travail sera probablement accéléré dû à tout ce logiciel déjà développé.

Actuellement je suis en train de programmer un paquet pour la visualisation des fichiers .VAW et .VOC et de commencer à essayer diverses transformées. Aussi j'ai testé quelques algorithmes de compression DPM (μ-Law et A-Law) avec des suites aléatoires.

F.3. État du Projet 12-12-94

Après mon voyage à Londres, qui a eu comme unique fait intéressant pour ce projet la visite à la bibliothèque de la *Guildhall London Polytechnic University* (celle-ci est l'université de ma cousine) où j'ai trouvé les mêmes livres que je peux trouver à Valence ce Noël, j'ai utilisé ces deux semaines à essayer des algorithmes de restitution à différentes vitesses.

Je poursuit à essayer de résoudre le problème avec la carte moyennant l'Internet parce qu'aucune nouvelle de Creative Labs a été reçue. La même chose peut être dite de la programmation directe de la carte aussi bien que du DSP incorporé.

Au moment de la démonstration faite à la société Barthe, il y avait encore beaucoup d'erreurs et de paramètres à fixer et quoique le résultat final ait été acceptable. Je crois que maintenant j'ai définitivement obtenu des bons algorithmes. En plus, leur complexité est encore suffisamment petite pour qu'ils puissent être implantés sans trop de découpages dans un microcontrôleur ou dans un DSP comme il est commenté dans le rapport.

J'ai commenté avec Boris le travail et les algorithmes que M. Pépin a déjà implanté sur le microcontrôleur et à mesure que mon français améliorera, nous augmenterons la collaboration et les suggestions mutuelles. Je peux apprendre beaucoup de toute l'expérience qu'ils ont a l'heure de l'implantation et les restrictions des microcontrôleurs de même qu'ils se serviront au futur de tout le logiciel ou algorithmes que je puisse développer.

La planification du projet a été changée, car j'ai sauté une grande partie du travail initial de représentation et traitement du signal mais beaucoup de travail ultérieur a déjà été développé. Celle-ci est la raison que je dois faire une reconsidération des points dont je peux travailler maintenant. Veillez m'indiquer quels sont les très prioritaires ou importants pour mon directeur du projet, M. Goureau ou pour la tâche que M. Pépin et Siepert font:

1. Je peux faire plus essais encore de l'algorithme:

   ◊ Étudier profondément l'influence de la fréquence et amplitudes immédiates sur le point de la coupe et de la procédure de caractérisation.

   ◊ Changer les sources des signaux, en essayant avec des cassettes, des différentes langues, des femmes, des enfants, etc.

   ◊ Développer des mesures de similarité plus objectives et exactes d'évaluation de la qualité de la recomposition.

   ◊ Insérer des filtres numériques à l'entrée et/ou à la sortie pour voir si une amélioration est appréciable.

2. Restructuration du programme.

- ◊ Le programme doit être reconsidéré parce que quelques choses ont été changées pendant la réalisation de ce rapport.

- ◊ La taille des dernières versions du programme est presque trop grand pour une application réduite WINDOWS. Ce fait nécessaire la conversion à une vraie application WINDOWS en utilisant la librairie ObjectWindows 2.0 du logiciel Borland C++ 4.0. En plus, cette librairie fournit des fonctions pour l'enregistrement et reproduction directes des fichiers compatibles WINDOWS sur quelconque carte. Ceci fera superflue l'utilisation (quelques fois très lourde) des programmes externes comme Creative WaveStudio, en se convertissant ainsi en une application complètement indépendante.

- ◊ Représentation des données et visualisation de fichiers, avec un traitement et édition des paramètres plus interactifs. Introduction aussi des transformées (les sources sont déjà développées et celles seulement requièrent son insertion dans le programme).

- ◊ Acceptation d'autres fichiers d'Audio comme .VOC de Creative Labs et programmation directe de la carte. Cela ferait perdre l'indépendance que le programme a maintenant.

- ◊ Faire une application amiable pour son utilisation par quelconque personne avec la possibilité de la distribution ou même la commercialisation si la société Barthe est intéressée.

On peut se rendre compte de que tous le points antérieurs sont très relationnés.

3. Implantation sur des puces cibles.

- ◊ Je poudrais m'incorporer aux travaux que MM. Pépin et Siepert font sur le microcontrôleur.
- ◊
- ◊ Lecture des manuels de l'Intel µC 80C196 et de l'Analog Devices ADSP 2111.

- ◊ Réétudier le projet de Machado et Fernandes

4. Recherche bibliographique et dans l'Internet sur:

- ◊ Paramètres de caractérisation, tailles du pitch, détection du silence, etc.

- ◊ Divers types de transformées lesquelles remarquent les particularités du signal.

- ◊ Méthodes de compression lesquels feront compatibles avec les algorithmes développés. Attention spéciale sera mise vers les algorithmes récursifs.

5. Autres:

- ◊ Se renseigner sur la programmation du DSP de la carte
- ◊ Dû au fait que mon niveau de français écrit s'est amélioré un peu, je pourrai faire une traduction de mon didacticiel de C++ pour son utilisation à l'enseignement dans l'ENSEA.
- ◊ Préparer un article pour l'envoyer à diverses revues sur la matière. Mais il sériait plus conseillable atteindre aux résultats définitifs.
- ◊ Faire un rapport des CD-ROMs présents au marché pour son futur achat après Noël.
- ◊ Après les dernières nouvelles d'IBM sur l'erreur physique des Pentiums lesquelles disent que l'erreur est plus habituelle qu'il était pensé, je peux demander le programme de test à mes amis pour le vérifier.

En considérant toutes ces choses à faire, personnellement je crois que c'est le moment d'arrêter un peu et faire un coup d'oeil bibliographique (des matières mentionnées au point 4) pour voir lequel a été fait sur la matière à la bibliothèque et l'hémérothèque de l'Université Polytechnique de Valence, pendant ma visite de Noël qui commencera le 16 de décembre, vendredi, jusqu'à le 8 de Janvier, lundi.

Jusqu'à la réunion de ce jeudi, je me dédierai fondamentalement au point 2 de restructuration du programme et à préparer tout le matériel (livres, données, fichiers, etc.) que je peux avoir besoin ces trois semaines. Bon Noël!

F.4. État du Projet 16-2-95

Pendant les vacances de Noël et ce dernier mois, la planification initiale n'a pas été suivie fidèlement mais la plupart des objectifs ont été presque accomplis. Mes activités se sont centrées sur les aspects suivants:

RECHERCHE BIBLIOGRAPHIQUE :

J'ai continué avec la recherche bibliographique en collaboration avec Boris. Il a fait un résumé et une archive de tous les articles que nous avons trouvés fondamentaux sur la compression de audio. De plus, Boris a demandé quelques livres à la bibliothèque de l'ENSEA et pour le laboratoire en particulier pour établir une petite bibliothèque sur le traitement de la parole et la conception des ASIC pour l'utilisation dans l'ECS.

D'autre part, INTERNET est encore la source la plus importante pour le développement des algorithmes et de l'acquisition de logiciel. Il nous aide à être au jour des standards et dernières recherches sur une matière quelconque.

NOUVEAUX OUTILS :

J'ai installé plusieurs outils pour le maniement et l'analyse des fichiers de son, telles comme:

- BMASTER: Un paquet pour DOS qui permet de réaliser la vitesse variable mais avec une qualité inférieure a mes algorithmes déjà développés.

- COOL: Un paquet pour WINDOWS qui permet de réaliser de l'analyse spectrale. Il est le plus complet mais il n'est pas *shareware* (de libre distribution), ne permettant que un numéro réduit d'options dans chaque installation.

- GOLDWAVE: Un bon outil plus indépendant que le logiciel fournit par la carte SoundBlaster AWE 32. Il est le logiciel utilisé le plus couramment pour les démonstrations mais il est graduellement substitué par l'application Accordion.

- MATLAB: L'installation a été pensée pour l'étude des filtres dans le paquet « Signal Processing Toolbox » mais je n'ai essayé que des petites choses dans ce puissant logiciel.

De plus, d'autres logiciels pour la conversion de fichiers font toutes les opérations requises, lesquelles ne sont pas fournies par le programme ACCORDION dont je vais parler.

VITESSE VARIABLE :

En relation avec la vitesse variable, les algorithmes déjà développés se sont montré supérieurs à ceux développés ou découverts ailleurs, mais la procédure de partition des tranches peut être optimisée si on utilise des fenêtres pour diminuer les coupes. Ainsi, on économise toute la procédure de partition sans aucune perte de la qualité. L'influence de la fréquence et l'amplitude dans la procédure de caractérisation a été étudié et s'est montré moins importante qu'on avait pensé. Ce qui signifie que l'on peut supprimer cette partie si les requêtes de la puce cible le nécessitent.

La vitesse variable qui utilise une analyse spectrale n'a pas donné de mauvais résultats mais c'est peu utile en comparaison avec les algorithmes directs sur le signal. Mais, on peut reprendre son étude dans le cas où un algorithme de compression spectrale devrait être développé.

COMPRESSION :

Elle a été la matière de travail la plus fondamentale pendant ces dernières semaines. On a essayé les compressions: log-PCM (standards A-law et μ-law), ADPCM sur 2, 3 et 4 bits (standards CCITT), DM, ADM sur le niveau du signal (waveform coding) et aussi sur le niveau spectral (FFT coding, DCT coding). J'ai essayé d'autres algorithmes avec des résultats variables.

La conclusion à laquelle nous sommes arrivés est que la meilleure compression applicable avec une bonne qualité (pour les applications de l'apprentissage des langues) est l'ADPCM sur 4 ou 3 bits, en donnant des codages de 32 à 64 kbps. L'algorithme n'est pas très compliqué et peut être implanté sur une puce quelconque mais on aurait besoin d'environ 3/4 Kilo-octets par second (kops) pour le stockage. Par exemple, avec une fréquence de 8 kHz il faudrait : 4 bits -> 32 kbps = 4 kops par second, 3 bits = 3 kops et 2 bits = 2 kops.

Pour d'autres applications comme un enregistrement de quelques minutes, il faudrait de plusieurs Mega-octets pour l'enregistrement. Donc, d'autres algorithmes doivent être étudiés. Au niveau du signal, la DM et surtout la ADM donne des résultats acceptables avec des codages de 8 kbps, approximativement. Cela représente 1 Kilo-octet par second, c'est-à-dire, 60 K par minute. Il peut être appliqué à l'enregistrement des lettres, par exemple. J'ai développé un nouvel algorithme avec une compression variable qui peut profiter des silences. Il est encore en essai mais si on ajuste bien certains paramètres, on peut obtenir avec la même qualité des codages 4 fois supérieurs. Il représente, 15 K par minute. Ainsi, on peut enregistrer un quart d'heure dans une puce de 256 Kilo-octets par exemple. Une autre bonne caractéristique de cet algorithme est sa qualité (et inversement le facteur de compression, logiquement) qui peut être sélectionné.

TRANSFORMÉES :

Ils ont été extraits d'INTERNET ou de la bibliographie et j'ai implanté diverses FFT (Fast Fourier Transform), plus ou moins complexes, avec plus ou moins de précisions. Je peux faire varier le numéro d'échantillons par tranche (comme puissance de 2), l'intercalage entre les tranches, les fenêtres, la précision et des autres paramètres.

Aussi j'ai la source d'une DCT (Discrete Cosine Transform), qui a été adapté de la compression des images à la compression de sons, et elle marche assez bien avec la compression et une qualité réglables.

QUALITÉ :

Elles ont été développées a partir des différentes mesures de qualité pour permettre à cette opération d'être objective. A cet égard, deux méthodes ont été implantées et une troisième a été considérée.

- Comparaison des signaux: Il s'agit de calculer la moyenne des carrés des différences entre deux signaux (Minimum Squares).

- Comparaison spectrale: C'est presque la même chose que l'antérieure mais, la comparaison est réalisée en calculant la moyenne des carrés des différences entre les spectres. On prend chaque spectre comme un tableau et on calcule aussi la moyenne des carrés.

- pseudo-SNR: Divers articles et commentaires dans INTERNET parlent d'un algorithme qui approche assez les résultats théoriques du SNR (Signal-to-Noise Ratio, Rapport Signal-Bruit).

Deux versions existent pour chaque méthode: linéaire et dynamique. La première est utile pour étudier les algorithmes de compression et des signaux de même longueur; ils sont assez rapides. La deuxième implante le DTW (Dynamic Time Warping, une méthode de programmation dynamique couramment utilisée dans la reconnaissance de la parole) pour comparer des sons de différentes longueurs; il rend possible la comparaison des algorithmes de vitesse variable et ouvre la voie à une application assez intéressante pour l'apprentissage des langues: la comparaison d'un original (professeur, etc.) avec le son enregistré par l'élève; il peut donner ainsi une mesure de la qualité et une amélioration de la prononciation d'une langue étrangère.

FILTRES :

L'introduction dans le programme Accordion d'une fonction polynomial sur z, permet d'inclure des filtres numériques (passe-bas, passe-bande, coupe-bande, etc.) à l'entrée ou à la sortie, c'est-à-dire avant ou après des algorithmes de compression et/ou de vitesse variable. Il peut être important de placer un filtre après les compressions

où la vitesse variable pour éliminer les bruits de coupage. Fondamentalement, ils peuvent être utilisés pour le codage ADM.

FENÊTRES :

L'application de fenêtres peut être utilisé dans différents domaines:

- Dans les algorithmes de vitesse variable. L'utilisation de fenêtres inverses peut éliminer le bruit de coupe des tranches, ce qui rend superflues les coupes par passage par zéro et tout ce qui s'ensuit. Il s'agit simplement d'appliquer une fenêtre inverse (triangulaire, polynomial, Hanning, Hamming, etc.) à deux tranches consécutives: Le résultat est, la disparition de toute discontinuité dans la procédure de recomposition.

- Des fenêtres sont aussi très importantes pour calculer la FFT, en donnant plus d'importance aux valeurs centrales qu'à ceux des extrêmes. Le spectre calculé est ainsi plus exact qu'avec une fenêtre rectangulaire (c'est-à-dire, aucune fenêtre).

- Pour les compressions ou les effets basées sur la FFT, au moment de calculer l'inverse de la FFT, on a besoin d'appliquer des fenêtres pour assurer une liaison propre.

- Pour la compression HDFS, qui fait la moyenne de deux tranches dans une seule sans une perte remarquable de la qualité.

Normalement, le calcul est fait une fois avec la formule originelle de la fenêtre pour construire une table et après, elle est consultée à chaque nécessité. Donc, le coût computationnel est très petit et il peut être appliqué des fenêtres très sophistiquées.

COMPLEXITÉ :

L'évaluation des algorithmes est faite normalement d'une façon théorique en fonction d'un numéro *n* donné d'itérations. Couramment, on parle par exemple *O(n log n / 2) FLOPS* ou simplement opérations. Pour l'implantation finale sur une cible quelconque, le même résultat théorique peut donner des résultats pratiques très différents. Pour tout cela, j'ai fait un calcul de toutes les opérations que réalise un certain algorithme, et je les ai séparées en catégories: addition, multiplication, shift, accès mémoire, etc. Chacune d'elles est séparée aussi en entières et en virgule flottante, le numéro de bits requis, etc.

J'ai introduit cela dans l'application Accordion et ensuite, j'ai calculé le total des opérations pour chaque algorithme et après je fait du calcul concret pour une cible déterminée avec une structure gardant les nano-seconds pour chaque sorte d'opération. En plus, il donne les nano-seconds requises par échantillons. Ainsi, on peut savoir si

l'algorithme est implantable sur une certaine puce avant de l'essayer réellement.

Il peut être intéressant aussi pour le développement de l'ASIC à Boris, étant donné qu'une fois les algorithmes choisis, on peut évaluer les opérations les plus utilisées pour les optimiser dans l'ASIC.

PUCE TEXAS TMS32C050 :

J'ai étudié l'architecture de la puce et j'ai constaté la capacité pour implanter la plupart des algorithmes sans aucun problème, y compris ceux sur le domaine spectral. Cela a été réalisé avec Accordion; j'ai chargé les caractéristiques temporelles du DSP (cycles par instruction et nano-second par cycle).

Concrètement, cette puce fournit:

- Arithmétique entière ou réelle fixe.
- Quoique le mot soit de 16 bits, l'opération de shift est très efficace (incluse dans l'opération d'addition et multiplication). Cela permet adopter une longueur d'échantillon en bits quelconque.

Tout cela, fait-il approprié presque la plupart des algorithmes étudiés, donné la flexibilité de ce DSP. Le problème est la mémoire fournie par la carte laquelle est très réduite. Boris est en train d'ajouter de mémoire externe pour commencer des essais.

L'APPLICATION ACCORDION :

Beaucoup de temps a été investi aussi dans le développement d'un paquet d'outils pour le traitement de la parole sous WINDOWS. Il s'agit d'une application encore expérimentale qui donne fréquemment des erreurs d'application, mais qui permet d'inclure toutes les outils déjà développés dans le même programme. Ce logiciel incorpore:

- Acceptation des formats d'audio .RAW, .WAV et .SND (monaural uniquement).
- Représentation du signal et possibilité de zoom de la zone choisie.
- Changement des paramètres du format des fichiers: fréquence d'échantillonnage, bits par échantillon.
- Effets divers comme variation du volume, inversions, miroirs, etc.
- Application d'une fonction en z quelconque de la forme: polynôme divisé par polynôme.

- Représentation spectrale et sonagramme.
- Algorithmes de compression et décompression: ADM et ADPCM.
- Vitesse variable spectrale.
- Mesures de qualité comme la comparaison de spectres et de signal qui utilise DTW (Dynamic Time Warping).
- Mesures de complexité des algorithmes pour la puce Texas TMS32C050.

Les autres items déjà développés seront incorporés à court terme. Il s'agit seulement du changement de l'interface d'entrée des paramètres et de la sortie graphique de:

- La vitesse variable sur le domaine temporelle qui a donné des bons résultats.
- La compression DCT, FFT, etc.

Ainsi, de nouvelles caractéristiques pourront être développées facilement comme:

- Changement et enregistrement dans un fichier des caractéristiques temporelles de chaque puce pour évaluer les algorithmes pour un DSP quelconque en changeant de simples paramètres.
- Automatiser l'incorporation des filtres les plus communs (Butterworth, Tchebychev, elliptiques) soit en z soit en s.
- Détection du Pitch, des formants et des silences.
- Représentation de l'enveloppe, fréquence et puissance moyenne.
- Traduction au français.
- Mettre des unités dans les axes de coordonnées.
- Enlever les nombreuses erreurs d'application.

CONFIGURATIONS PROPOSÉES :

Pour le laboratoire individuel de langues, la vitesse variable ou l'ASIC que Boris est en train de développer, les discussions se centrent sur le schéma suivant:

Après l'entrée à une fréquence qui peut varier entre 10 Khz jusqu'à 20 Khz pour le traitement de la parole, un filtrage numérique doux peut être introduit. En suite, la procédure de partition coupera en

tranches et analysera la caractérisation et l'emphase de chaque tranche pour choisir l'algorithme de compression le plus adéquat (ADPCM ou ADM) avant le stockage. Dans l'ASIC il sera fixe pour économiser toute la logique mais dans un DSP on peut mettre plusieurs algorithmes différents et les choisir selon ce que l'utilisateur ait demandé.

Une fois la recomposition demandée, la répétition à vitesse variable dépendra des valeurs de caractérisation déjà calculées. Finalement, elles seront décomprimées les tranches qui doivent être écouté avec l'application d'un filtre (numérique) et des fenêtres pour éviter un possible bruit de coupage.

Malheureusement, beaucoup de temps a été utilisé dans le développement des algorithmes et de l'application (fondamentalement dû aux erreurs d'application de WINDOWS et le gestionnaire de ressources). Donc, il restera un peu plus d'un mois après les vacances d'hiver pour la finition de mon projet, et une bonne part de celui devra être dédié à écrire le mémoire du projet, peu avancé jusqu'à ce moment.

Personnellement, je pense que la réalisation d'un bon rapport des travaux effectués (avec une exposition plus claire de la configuration mentionnée ci-dessus) et de finir l'application ACCORDION avec toutes les opérations qu'il faut pour l'étude de la parole, pour la complexité des algorithmes et pour la souplesse d'utilisation, serait plus utile pour le laboratoire qu'entreprendre une nouvelle tâche de finition douteuse.

Quant à l'équipement, aucune information a été trouvée pour la programmation du DSP de la carte. Creative Labs n'a pas répondu à la lettre que j'avais envoyée en novembre qui demandait des informations plus détaillées sur la carte. J'ai constaté aussi l'erreur du Pentium; le remplacement de la puce on dit qu'il est gratuit mais je ne sais pas où la demander. En ce qui concerne au nouvel achat d'équipement, quoiqu'il ne soit pas si important pour moi, (parce que peut-être je ne serai pas ici pour leur arrivée), deux items peuvent être intéressants pour la recherche future sur le domaine du traitement de la parole et pour le laboratoire en général:

- En premier lieu, un lecteur des disques optiques compacts (CD-ROM) serait-il très valable pour l'étude des moyens d'enregistrement numérique et d'autre part, pour la possibilité d'acheter tout le futur logiciel dans ce format, plus rapide et fiable que les disquettes.

- De plus, puisque la programmation en C et C++ est progressivement utilisée dans le laboratoire, la mise à jour du logiciel Borland C++ 4.0 à la nouvelle Borland C++ 4.5 peut s'avérer utile fondamentalement si on se rend compte de l'apparition de WINDOWS '95, dans lequel la dernière version de Borland est déjà adaptée.

Enfin, il ne me reste plus qu'à commenter la planification et la date de présentation de mon projet. Théoriquement, depuis mon arrivée le 10 octobre de 1994, les six moins de bourse ERASMUS sont finis le 10 avril de cette année. Par conséquent, je voudrais présenter mon projet au commencement de la semaine de lundi 3 à vendredi 7 avril (mardi ou mercredi si possible) pour avoir encore quelques jours pour emballer et finir toutes les tâches administratives. Il ne faut pas oublier que tout cela peut varier s'il existe quelque inconvénient de calendrier pour le laboratoire.

F.5. État du Projet 30-3-95

La soutenance est prévue pour le 7 d'avril et ma partie pour le 12 d'avril. Ces dernières semaines je travaille sur :

- La réalisation du rapport.
- La mise en place sur le DSP.
- L'achèvement de l'outil Accordion.
- La préparation des transparences pour la soutenance.
- L'arrangement de toutes les choses dans le laboratoire avant ma partie.

Quelqu'un des nouvelles erreurs apparues dans ce rapport ou le programme peuvent être causes par le fait de que j'ai travaillé près de 14 heures chaque jour pendant ces deux dernières semaines. Je me suis gagné des vacances !

Annexe G

Listings Sélectionnés

La quantité immense de logiciel ne permet pas de le mettre tous dans ce rapport. La partie la plus longue est celle concernant à l'interface avec l'utilisateur et la représentation graphique. Mais les plus importants sont ceux des algorithmes que j'ai réalisés pour ce projet.

Les commentaires, uniformités et structures des programmes, ainsi que leur structure modulaire, ne sont pas excessivement propres étant donné les diverses sources des algorithmes et le manque de temps pour tous le traiter en détail.

Voici une liste alphabétique des programmes que je trouve les plus importants :

G.1. « ADM.CPP »

Ce programme implante les algorithmes ADM (Adaptive Delta Modulation) et ADxyM (Adaptive Delta x y Modulation) :

```
/*********************************************************
  MODULE : ADM.CPP
  AUTHOR : Jose Hernandez
  VERSION : 1.0
  PROJET : ACCORDION
  DESCRIPTION : Adaptive Delta Modulation
*********************************************************/

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

#include "adm.h"

#define MULTFACT       1.3
#define DIVFACT 1.5

#define MINDELTA 0.1
#define INITIALDELTA 1.

// ((input->NumberofBits == 16) ? 16 : 2)
#define MAXDELTA16 4096
#define MAXDELTA8 64


void ADMCompress(sample_array *input, long OffSet, BYTE huge *output, long Len,
                                          ADMSt *St, long *RealBlockLen) {
     long MAXDELTA= MAXDELTA16;
     long c;
     long c0;
     float dm;
     int sign;
     int b;
     int Pos;
     BYTE Buffer;
```

```
     if (!(St->Initialized)) {
               c0= 0;
               dm= INITIALDELTA;
               sign= 1;
               Pos= 0;
               Buffer= 0;
               St->Initialized= TRUE;
     }
     else {
               c0= St->c0;
               dm= St->dm;
               sign= St->sign;
               Pos= St->Pos;
               Buffer= St->Buffer;
     }

     long RealLen= 0L;

     for(long l= 0; l < Len; l++) {
               c = (short)input->Get(l + OffSet);
               if ((c - (c0 + dm * sign))* sign > 0) {
                 b= 1;
                 c0= c0 + dm * sign;
//               if (dm == 0)
//                        dm= MINDELTA;
//               else {
                        dm *= MULTFACT;
                        if (dm > MAXDELTA) {
                                 dm= MAXDELTA;
                        }
//               }
               }
               else {
                 c0= c0 + dm * sign;
                 b= 0;
                 sign*= -1;
                 dm /= DIVFACT;

                 if (dm < MINDELTA) {
                        dm= MINDELTA;
                 }
               }

               if (c0 > 0x7FFFL)
                 c0= 0x7FFFL;
               else if (c0 < -0x8000L)
                 c0= -0x8000L;

               Buffer |= (b << Pos);
               Pos++;
               if (Pos == 8) {
                 *output++= Buffer;
                 Buffer= 0;
                 RealLen++;
                 Pos= 0;
               }
     }

     St->c0= c0;
     St->dm= dm;
     St->sign= sign;
     St->Pos= Pos;
     St->Buffer= Buffer;
     if (Pos) {
               *output= Buffer;
     }
     *RealBlockLen= RealLen;
}

void ADMDecompress(BYTE huge *input, sample_array *output, long OffSet, long Len,
                                                        ADMSt *St, long *RealBlockLen) {
     long MAXDELTA= MAXDELTA16;

                        // COMPRESS and DECOMPRESS must match. Insert as parameters

     long c0;
     float dm;
     int sign;
     int b;
     int Pos;
     BYTE Buffer;

     if (!(St->Initialized)) {
               c0= 0;
               dm= INITIALDELTA;
               sign= 1;
               Pos= 0;
               Buffer= 0;
               St->Initialized= TRUE;
     }
     else {
               c0= St->c0;
               dm= St->dm;
               sign= St->sign;
               Pos= St->Pos;
               Buffer= St->Buffer;
     }

     long RealLen= 0L;

     for(long l= 0; l < Len; l++) {
               Buffer >>= 1;
               if (Pos == 0) {
                 Buffer= *input++;
                 RealLen++;
                 Pos= 8;
               }
               Pos--;
               b= Buffer & 1;
```

```
              if (b == 1) {
                c0= c0 + dm * sign;
//              if (dm == 0)
//                      dm= MINDELTA;
//              else {
                        dm *= MULTFACT;
                        if (dm > MAXDELTA) {
                                dm= MAXDELTA;
                        }
//              }
              }
              else {
                c0= c0 + dm * sign;
                sign*= -1;
                dm /= DIVFACT;

                if (dm < MINDELTA) {
                        dm= MINDELTA;
                }
              }
              if (c0 > 0x7FFFL)
                c0= 0x7FFFL;
              else if (c0 < -0x8000L)
                c0= -0x8000L;
              output->Set(l + OffSet, short(c0));
     }

     St->c0= c0;
     St->dm= dm;
     St->sign= sign;
     St->Pos= Pos;
     St->Buffer= Buffer;
     *RealBlockLen= RealLen;
}

#define MINDELTAY8                          1
#define MAXDELTAY8                          64
#define STEPMINDELTAY8            32
#define STEPMAXDELTAY8            16


#define MINDELTAY16                         0.1
#define MAXDELTAY16                         4096
#define INITIALDELTAY             256.
#define INITIALDELTAX             0.


#define MAXDELTAXRef                        1.
#define MINDELTAXRef                        -2.
#define STEPMINDELTAY16           32.
#define STEPMAXDELTAY16           512.

#define MULY 3.
#define DIVY 4.

#define ADDX 1
#define MULX 0.7

#define SUBX 0.5
#define DIVX 0.


#define ADxyMCommon()
                                    \
     float MAXDELTAY= MAXDELTAY16;
    \
     float MINDELTAY= MINDELTAY16;                                  \
     float MAXDELTAX= MAXDELTAXRef * pow(DesiredBPS/Freq, 1.);                                  \
     float MINDELTAX= MINDELTAXRef * pow(Freq/DesiredBPS, 1.);                  \
     float STEPMAXDELTAY= STEPMAXDELTAY16;                          \
     float STEPMINDELTAY= STEPMINDELTAY16;



void ADxyMCompress(sample_array *input, long OffSet, BYTE huge *output, long Len,
                                                ADxyMSt  *St,  long  *RealBlockLen,  double  Freq,
cost & COST, double DesiredBPS) {
     ADxyMCommon();
     long c, c0;
     float dx;
     float dy;
     float cx;
     int sign;
     int b;
     int Pos;
     BYTE Buffer;
     int Rep;

     if (!(St->Initialized)) {
              c0= 0;
              cx= 0.;
              dx= INITIALDELTAX;   //      > 0 : dx deltas per sample
                                                                                  // < 0 : dx samples
per delta
              dy= INITIALDELTAY;
              sign= 1;
              Pos= 0;
              Buffer= 0;
              St->Initialized= TRUE;
     }
     else {
              c0= St->c0;                                                                                 COST+=
OP::LOST;
              dx= St->dx;                                       COST+= OP::LOST;
              cx= St->cx;                                       COST+= OP::LOST;
              dy= St->dy;                                       COST+= OP::LOST;
              sign= St->sign;                                   COST+= OP::LOST;
              Pos= St->Pos;                                     COST+= OP::LOST;
```

```
            Buffer= St->Buffer;                                 COST+= OP::LOST;
     }

     long RealLen= 0L;                                      COST+= OP::SGL;

     for(long l= 0; l < Len; l++) {                         COST+= OP::RPB;
            c = (short)input->Get(l + OffSet);                  COST+= OP::LOST;

            St->Meandx+= dx;
            St->Meandy+= dy;

            if (dx >= 0) {                                      COST+= OP::BRA;
              Rep= dx + 1;
                                COST+= OP::ADD;
            }
            else if (cx >= 0) {                                 COST+= OP::BRA;
              cx= dx;                                           COST+= OP::SGL;
              Rep= 1;                                           COST+= OP::SGL;
            }
            else {
              cx++;                                             COST+= OP::SGL;
              Rep= 0;                                           COST+= OP::SGL;
            }

            for (int i= 0; i < Rep; i++) {
    COST+= OP::RPB;
              if ((c - (c0 + dy * sign))* sign > 0) {           COST+= OP::ADMU; COST+= OP::BRA;
                                b= 1;                                       COST+= OP::SGL;
                                c0= c0 + dy * sign;
                        COST+= OP::ADD;
                                dy *= MULY;
                                                COST+= OP::MUL;
                                if (dy > STEPMAXDELTAY) {               COST+= OP::BRA;
                                  dx= dx + ADDX + fabs(dx) * MULX;
    COST+= OP::ADMU;
                                  if (dx > MAXDELTAX) {                 COST+= OP::BRA;
                                          dx= MAXDELTAX;                        COST+= OP::SGL;
                                  }
                                }
                                if (dy > MAXDELTAY) {
                        COST+= OP::BRA;
                                  dy= MAXDELTAY;                        COST+= OP::SGL;
                                }
                }
                else {
                                b= 0;                                       COST+= OP::SGL;
                                c0= c0 + dy * sign;
                        COST+= OP::ADD;
                                dy /= DIVY;
                                                COST+= OP::SFT;     // Suppose DIVY power of 2

                                if (dy < STEPMINDELTAY) {
              COST+= OP::BRA;
                                  dx= dx - SUBX - fabs(dx) * DIVX;          COST+= OP::ADMU;
                                  if (dx < MINDELTAX) {
                        COST+= OP::BRA;
                                          dx= MINDELTAX;                        COST+= OP::SGL;
                                  }
                                }
                                if (dy < MINDELTAY) {
                        COST+= OP::BRA;
                                  dy= MINDELTAY;                        COST+= OP::SGL;
                                }

                                sign*= -1;                              COST+= OP::SGL;
                }

                if (c0 > 0x7FFFL) {
                        COST+= OP::BRA;
                         c0= 0x7FFFL;                                           COST+= OP::SGL;
                }
                 else if (c0 < -0x8000L) {
    COST+= OP::BRA;
                          c0= -0x8000L;                                         COST+= OP::SGL;
                }

                Buffer |= (b << Pos);                               COST+= OP::ADSF;
                Pos++;
                                                COST+= OP::SGL;
                if (Pos == 8) {
                                COST+= OP::BRA;
                         *output++= Buffer;
                        COST+= OP::LOST;
                         RealLen++;
                                                COST+= OP::ADD;
                         Buffer= 0;                                             COST+= OP::SGL;
                         Pos= 0;
                                                COST+= OP::SGL;
                }
              }
     }

     St->c0= c0;
     St->dx= dx;
     St->dy= dy;
     St->cx= cx;
     St->sign= sign;
     St->Pos= Pos;
     St->Buffer= Buffer;
     if (Pos) {
            *output= Buffer;
     }
     *RealBlockLen= RealLen;
}

void ADxyMDecompress(BYTE huge *input, sample_array *output, long OffSet, long Len,
                                                                ADxyMSt *St, long *RealBlockLen, double
Freq, cost & COST, double DesiredBPS) {
     ADxyMCommon();
```

```
     long c0;
     float dx;
     float dy;
     float cx;
     int sign;
     int b;
     int Pos;
     BYTE Buffer;
     int Rep;

     if (!(St->Initialized)) {
               c0= 0;
               cx= 0.;
               dx= INITIALDELTAX;   //       > 0 : dx deltas per sample
                                                                                                  // < 0 : dx samples
per delta
               dy= INITIALDELTAY;
               sign= 1;
               Pos= 0;
               Buffer= 0;
               St->Initialized= TRUE;
     }
     else {
               c0= St->c0;
               dx= St->dx;
               cx= St->cx;
               dy= St->dy;
               sign= St->sign;
               Pos= St->Pos;
               Buffer= St->Buffer;
     }

     long RealLen= 0L;

     for(long l= 0; l < Len; l++) {
               if (dx >= 0) {
                 Rep= dx + 1;
               }
               else if (cx >= 0) {
                 cx= dx;
                 Rep= 1;
               }
               else {
                 cx++;
                 Rep= 0;
               }

//             int Cinter;
               for (int i= 0; i < Rep; i++) {
                 Buffer >>= 1;
                 if (Pos == 0) {
                          Buffer= *input++;
                          Pos= 8;
                          RealLen++;
                 }
                 Pos--;
                 b= Buffer & 1;

                 if (b == 1) {
                                   c0= c0 + dy * sign;
                                   dy *= MULY;
                                   if (dy > STEPMAXDELTAY) {
                                     dx= dx + ADDX + fabs(dx) * MULX;
                                     if (dx > MAXDELTAX)
                                              dx= MAXDELTAX;
                                   }
                                   if (dy > MAXDELTAY) {
                                     dy= MAXDELTAY;
                                   }
                 }
                 else {
                                   b= 0;
                                   c0= c0 + dy * sign;
                                   dy /= DIVY;

                                   if (dy < STEPMINDELTAY) {
                                     dx= dx - SUBX - fabs(dx) * DIVX;
                                     if (dx < MINDELTAX)
                                              dx= MINDELTAX;
                                   }
                                   if (dy < MINDELTAY) {
                                     dy= MINDELTAY;
                                   }

                                   sign*= -1;
                 }
//               cInter= cold * (Rep - i + 1) / float(Rep) + c0 * (i + 1) / float(Rep);
               }

               if (c0 > 0x7FFFL)
                 c0= 0x7FFFL;
               else if (c0 < -0x8000L)
                 c0= -0x8000L;

               output->Set(l + OffSet, short(c0));
//             cold= c0;
     }

     St->c0= c0;
     St->dx= dx;
     St->dy= dy;
     St->cx= cx;
     St->sign= sign;
     St->Pos= Pos;
     St->Buffer= Buffer;
     *RealBlockLen= RealLen;
}
```

```
/*
#define ALFA 2
#define BETA 3

void TombrasADMCompress(sample_array *input, long OffSet, BYTE huge *output, long Len) {
     short c, c0= 0;
     int dm= 1;
     int sign= 1;
     int b;
     int Pos= 0;
     BYTE Buffer= 0;

     for(long l= 0; l < Len; l++) {
              c = input->Get(l + OffSet);
              e= c - c0;
              float n;
              if (sgn(e) == sign) {
                n= ALFA;
              }
              else
                n= -1. / ALFA;
              float dmref= abs(n) * abs(dm);
              float m;

              if (abs(e) >= ((BETA + 1) / 2.) * abs(dmref))
                m= BETA * n;
              else if (abs(e) <= ((1 / BETA + 1) / 2.) * abs(dmref))
                m= n / BETA;
              else
                m= n;

              sign= sgn(e);
              dm= m*dm;
              if (dm < MINDELTA)
                dm= MINDELTA;
              else if (dm > MAXDELTA)
                dm= MAXDELTA;
              c0= c0 + sign * dm;


              Buffer |= (b << Pos);
              Pos++;
              if (Pos == 8) {
                *output++= Buffer;
                Buffer= 0;
                Pos= 0;
              }
     }
     *output= Buffer;
}

void TombrasADMDecompress(BYTE huge *input, sample_array *output, long OffSet, long Len) {
     short c0= 0;
     int dm= 1;
     int sign= 1;
     int b;
     BYTE Buffer= 0;
     int Pos= 0;

     for(long l= 0; l < Len; l++) {
              Buffer >>= 1;
              if (Pos == 0) {
                Buffer= *input++;
                Pos= 8;
              }
              Pos--;
              b= Buffer & 1;

              if (b == 1) {
                c0= c0 + dm * sign;
                if (dm == 0)
                         dm= 1;
                else {
                         dm *= MULTFACT;
                         if (dm > MAXDELTA) {
                                  dm= MAXDELTA;
                         }
                }
              }
              else {
                dm /= DIVFACT;

                if (dm < 1) {
                         dm= 1;
                }

                sign*= -1;
                c0= c0 + dm * sign;
              }
              output->Set(l + OffSet, c0);
     }
}
*/
```

G.2. « ADPCM.CPP »

Ce programme implante l'algortihme ADPCM DVI d'Intel :

```
/*********************************************************
  MODULE : ADPCM.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Intel DVI ADPCM
*********************************************************/
/*
**
** The algorithm for this coder was taken from the IMA Compatability Project
** proceedings, Vol 2, Number 2; May 1992.
**
** Version 1.2, 18-Dec-92.
**
** Change log:
** - Fixed a stupid bug, where the delta was computed as
**   stepsize*code/4 in stead of stepsize*(code+0.5)/4.
** - There was an off-by-one error causing it to pick
**   an incorrect delta once in a blue moon.
** - The NODIVMUL define has been removed. Computations are now always done
**   using shifts, adds and subtracts. It turned out that, because the standard
**   is defined using shift/add/subtract, you needed bits of fixup code
**   (because the div/mul simulation using shift/add/sub made some rounding
**   errors that real div/mul don't make) and all together the resultant code
**   ran slower than just using the shifts all the time.
** - Changed some of the variable names to be more meaningful.
*/

#include "adpcm.h"
#include "sample.h"

/* Intel ADPCM step variation table */
static int indexTable4[16] = {
     -1, -1, -1, -1, 2, 4, 6, 8,
     -1, -1, -1, -1, 2, 4, 6, 8,
};

static int indexTable3[8]= {
     -1, -1, 1, 2,
     -1, -1, 1, 2
};

static int indexTable2[4]= {
     -1, 1,
     -1, 1
};

static int indexTable1[2]= {
     -1,
     1
};

static short stepsizeTable[89] = {
     7, 8, 9, 10, 11, 12, 13, 14, 16, 17,
     19, 21, 23, 25, 28, 31, 34, 37, 41, 45,
     50, 55, 60, 66, 73, 80, 88, 97, 107, 118,
     130, 143, 157, 173, 190, 209, 230, 253, 279, 307,
     337, 371, 408, 449, 494, 544, 598, 658, 724, 796,
     876, 963, 1060, 1166, 1282, 1411, 1552, 1707, 1878, 2066,
     2272, 2499, 2749, 3024, 3327, 3660, 4026, 4428, 4871, 5358,
     5894, 6484, 7132, 7845, 8630, 9493, 10442, 11487, 12635, 13899,
     15289, 16818, 18500, 20350, 22385, 24623, 27086, 29794, 32767
};

void adpcm_coder(sample_array *indata, long InOffset, BYTE huge *outdata, long len, struct adpcm_state
*state,
                                              int Bits, cost & COST) {
  signed char huge *outp;                    /* output buffer pointer */
  short val;                      /* Current input sample value */
  int sign;                       /* Current adpcm sign bit */
  int delta;                      /* Current adpcm output value */
  long diff;                      /* Difference between val and valprev */
  int step;                       /* Stepsize */
  long valpred;                   /* Predicted output value */
  int vpdiff;                     /* Current change to valpred */
  int index;                      /* Current step change index */
  int outputbuffer;               /* place to keep previous 4-bit value */
  int bufferstep;                 /* toggle between outputbuffer/output */

  outp = (signed char *)outdata;

  valpred = state->valprev;
                        COST+= OP::LOST;
  index = state->index;                                    COST+= OP::LOST;
  step = stepsizeTable[index];                             COST+= OP::IND;

  bufferstep = 1;                                          COST+= OP::LOST;

  for (long l= 0; l < len; l++) {
    COST+= OP::RPB;
    val = indata->Get(l + InOffset);
              COST+= OP::LOST; COST+= OP::IND;

    /* Step 1 - compute difference with previous value */
    diff = long(val) - long(valpred);
    COST+= OP::ADD;
    sign = (diff < 0) ? (1 << (Bits - 1)) : 0;                                          COST+=
OP::BRA;
              // 1Comp
    if (sign) {
```

```
      diff = (-diff);                                          COST+= OP::BRA;
    }

    /* Step 2 - Divide and clamp */
    /* Note:
    ** This code *approximately* computes:
    **    delta = diff*4/step;
    **    vpdiff = (delta+0.5)*step/4;
    ** but in shift step bits are dropped. The net result of this is
    ** that even if you have fast mul/div hardware you cannot put it to
    ** good use since the fixup would be too expensive.
    */
    delta = 0;
                                  COST+= OP::SGL;
    vpdiff = (step >> (Bits - 1));
              COST+= OP::SFT;

    if (Bits == 4) {
      if ( diff >= step ) {
                                  COST+= OP::BRA;
              delta = 4;                                                  COST+= OP::SGL;
              diff -= step;
                                                       COST+= OP::SGL;
              vpdiff += step;
                                            COST+= OP::ADD;
      }
      step >>= 1;
                                                       COST+= OP::SFT;
    }
    if (Bits >= 3) {
      if ( diff >= step  ) {                                    COST+= OP::BRA;
              delta |= 2;                                              COST+= OP::SGL;
              diff -= step;                                            COST+= OP::SGL;
              vpdiff += step;                                          COST+= OP::ADD;
      }
      step >>= 1;                                              COST+= OP::SFT;
    }
    if (Bits >= 2) {                                           COST+= OP::BRA;
      if ( diff >= step ) {                                    COST+= OP::SGL;
              delta |= 1;                                              COST+= OP::SGL;
              vpdiff += step;                                          COST+= OP::ADD;
      }
    }                                                          COST+= OP::SFT;
    /* Step 3 - Update previous value */
    if ( sign )         {
                                                                 COST+= OP::BRA;
      valpred -= vpdiff;                                       COST+= OP::ADD;
    }
    else {
      valpred += vpdiff;                                       COST+= OP::ADD;
    }

    /* Step 4 - Clamp previous value to 16 bits */
    if ( valpred > 32767L ) {                                  COST+= OP::BRA;
      valpred = 32767L;                                        COST+= OP::SGL;
    }
    else if ( valpred < -32768L ) {                            COST+= OP::BRA;
      valpred = -32768L;                                       COST+= OP::SGL;
    }

    /* Step 5 - Assemble value, update index and step values */
    delta |= sign;
                                                       COST+= OP::SGL;

    if (Bits == 4) {
      index += indexTable4[delta];
              COST+= OP::IND; COST+= OP::ADD;
    }
    else if (Bits == 3) {
      index += indexTable3[delta];
              COST+= OP::IND; COST+= OP::ADD;
    }
    else if (Bits == 2) {
      index += indexTable2[delta];                             COST+= OP::IND; COST+= OP::ADD;
    }
    else if (Bits == 1) {
      index += indexTable1[delta];                             COST+= OP::IND; COST+= OP::ADD;
    }
    if ( index < 0 ) {
                          COST+= OP::BRA;
      index = 0;
                                               COST+= OP::SGL;
    }
    if ( index > 88 ) {                                        COST+= OP::BRA;
      index = 88;
                                                       COST+= OP::SGL;
    }
    step = stepsizeTable[index];
              COST+= OP::IND;

    /* Step 6 - Output value */
    if ( bufferstep ) {
                                  COST+= OP::BRA;
      outputbuffer = (delta << 4) & 0xf0;
COST+= OP::SFT; COST+= OP::SGL;
      // MODIFICAR PARA TODOS LOS BITS
    }
    else {
      *outp++ = (delta & 0x0f) | outputbuffer;                                                    COST+=
OP::SFT; COST+= OP::SGL; COST+= OP::LOST; COST+= OP::SGL;
    }
    bufferstep = !bufferstep;
                        COST+= OP::SGL;
  }

  /* Output last step, if needed */
  if ( !bufferstep )    {
                                            COST+= OP::BRA;
```

```
    *outp++ = outputbuffer;
                        COST+= OP::LOST;
  }

  state->valprev = (short)valpred;
              COST+= OP::LOST;
  state->index = index;
                                COST+= OP::LOST;
}

void adpcm_decoder(BYTE huge *indata, sample_array *outdata, long OutOffset, long len, struct
adpcm_state *state,
                                                    int Bits, cost & COST) {
    signed char huge *inp;              /* Input buffer pointer */
    int sign;                           /* Current adpcm sign bit */
    int delta;                          /* Current adpcm output value */
    int step;                           /* Stepsize */
    long valpred;               /* Predicted value */
    int vpdiff;                         /* Current change to valpred */
    int index;                          /* Current step change index */
    int inputbuffer;            /* place to keep next 4-bit value */
    int bufferstep;             /* toggle between inputbuffer/input */

    inp = (signed char *)indata;

    valpred = state->valprev;
    index = state->index;
    step = stepsizeTable[index];

    bufferstep = 0;
    inputbuffer= 0;

    for (long l= 0; l < len; l++ ) {

   /* Step 1 - get the delta value */                                  // MODIFICAR PARA TODOS LOS
BITS
             if ( bufferstep ) {
               delta = inputbuffer & 0xf;
             }
             else {
               inputbuffer = *inp++;
               delta = (inputbuffer >> 4) & 0xf;
             }
   bufferstep = !bufferstep;

   /* Step 2 - Find new index value (for later) */
   if (Bits == 4)
     index += indexTable4[delta];
   else if (Bits == 3)
     index += indexTable3[delta];
   else if (Bits == 2)
     index += indexTable2[delta];
   else if (Bits == 1)
     index += indexTable1[delta];

   if ( index < 0 ) index = 0;
   if ( index > 88 ) index = 88;

   /* Step 3 - Separate sign and magnitude */
   sign = delta & (1 << (Bits - 1));
   delta = delta & ((1 << (Bits - 1)) - 1);

   /* Step 4 - Compute difference and new predicted value */
   /*
   ** Computes 'vpdiff = (delta+0.5)*step/4', but see comment
   ** in adpcm_coder.
   */
   vpdiff = step >> (Bits - 1);

   if (Bits == 4) {
     if ( delta & 4 )
             vpdiff += step;
     if ( delta & 2 )
             vpdiff += step>>1;
     if ( delta & 1 )
             vpdiff += step>>2;
   }
   else if (Bits == 3) {
     if ( delta & 2 )
             vpdiff += step;
     if ( delta & 1 )
             vpdiff += step>>1;
   }
   else if (Bits == 2) {
     if ( delta & 1 )
             vpdiff += step;
   }

   if ( sign )
     valpred -= vpdiff;
   else
     valpred += vpdiff;

   /* Step 5 - clamp output value */
   if ( valpred > 32767L )
     valpred = 32767L;
   else if ( valpred < -32768L )
     valpred = -32768L;

   /* Step 6 - Update step value */
   step = stepsizeTable[index];

   /* Step 7 - Output value */
            outdata->Set(l + OutOffset, (short)valpred);
    }

    state->valprev = (short)valpred;
    state->index = index;
}
```

G.3. « COST.H » et « COST.CPP »

Classe pour calculer le coût operationnel (complexité) :

```
/*********************************************************
  MODULE : COST.H
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Declaration of the cost class
*********************************************************/

#ifndef _COST_H_
#define _COST_H_

class OP {
public:
  enum {
     MAX,

     SGL,
     ADD,
     SFT,
     ADSF,
     MUL,
     ADMU,
     ADMUSF,
     DIV,
     RPI,
     RPB,
     LOD,
     STR,
     LOST,
     PUSH,
     POP,
     MEQ,
     MES,
     IND,
     IN,
     OUT,
     CAL,
     BRA,

     MAXOP
  };
  /*
     SGL               SinGLe instruction (other than the following)
     ADD               ADDition
     SFT               ShiFT
     ADSF              ADdition & ShiFt
     MUL               MULtiplication
     ADMU              ADdition & MUltiplication
     ADMUSF   ADdition, MUltiplication & ShiFt
     DIV               DIVision
     RPI               RePeat Instruction
     RPB               RePeat Block
     LOD               LOaD from memory
     STR               SToRe to memory
     LOST              Load and Store
     PUSH              PUSH to stack
     POP               POP from stack
     MEQ               extenal MEmory Quick
     MES               external MEmory Slow
     IND               Indexation
     IN                IN from port
     OUT               OUT to port
     CAL               CALl to subroutine
     BRA               BRAnch
  */
  enum {
     I8, I16, I32, I64,
     X16, X32, X64,
     R16, R32, R64, R80,
     C16, C32,

     MAXVR
  };

  static const int ABSENT;
};

class DSP {
public:
  int O[OP::MAXOP][OP::MAXVR];
  long Cycle;
  int DefVR;
  double InOutOverload;
  char *Desc;

  DSP() {
     for (int i= 0; i < OP::MAXOP; i++) {
              for (int j= 0; j < OP::MAXVR; j++) {
                O[i][j]= OP::ABSENT;
              }
     }
     Cycle= 0;
     Desc= "DSP Not initialized";
     InOutOverload= 0.0;
     DefVR= OP::I16;
  }

  static DSP *Default;
  static void Init_Default();
};
```

```
class cost {
private:
  double O[OP::MAXOP][OP::MAXVR];
  DSP *Chip;
public:
  void Clear();
  cost();
  cost(DSP & d);
  void SetChip(DSP * d);
  void Count(int Op, int Vr, int Num= 1);
  void operator += (int Op);
  double Cycles();
  double Time();
};

#endif
```

```
/*********************************************************
  MODULE : COST.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Implementation of the cost class
*********************************************************/

#include "cost.h"

DSP TMS320C50;

const int OP::ABSENT= -1;
DSP *DSP::Default= &TMS320C50;

void DSP::Init_Default() {
  Default->Cycle= 50;
  Default->InOutOverload= 0.02;
  Default->Desc= "Texas Instruments TMS320C50 DSP 40 MHz";
  Default->DefVR= OP::I16;

  Default->O[OP::MAX][OP::I16]= 1000;

  Default->O[OP::SGL][OP::I16]= 1;
  Default->O[OP::SFT][OP::I16]= 2;
  Default->O[OP::MUL][OP::I16]= 2;
  Default->O[OP::ADSF][OP::I16]= 2;
  Default->O[OP::ADMUSF][OP::I16]= 3;
  Default->O[OP::DIV][OP::I16]= 26;
  Default->O[OP::RPI][OP::I16]= 2;
  Default->O[OP::RPB][OP::I16]= 2;
  Default->O[OP::LOD][OP::I16]= 2;
  Default->O[OP::STR][OP::I16]= 2;
  Default->O[OP::LOST][OP::I16]= 3;
  Default->O[OP::IND][OP::I16]= 2;
}


void cost::Clear() {
     for (int i= 0; i < OP::MAXOP; i++) {
               for (int j= 0; j < OP::MAXVR; j++) {
                 O[i][j]= 0;
               }
     }
  }

cost::cost() : Chip(DSP::Default) {
     Clear();
  }

cost::cost(DSP & d) : Chip(&d) {
     Clear();
  }

void cost::SetChip(DSP * d) {
     Chip= d;
  }

void cost::Count(int Op, int Vr, int Num) {
     O[Op][Vr]+= Num;
  }

void cost::operator += (int Op) {
     Count(Op, Chip->DefVR);
  }

double cost::Cycles() {
     double T= 0.;
     for (int i= 0; i < OP::MAXOP; i++) {
               for (int j= 0; j < OP::MAXVR; j++) {
                 if ((Chip->O)[i][j] == OP::ABSENT)
                           T+= O[i][j] * OP::MAX;
                 else
                           T+= O[i][j] * (Chip->O)[i][j];
               }
     }
     return T;
  }

double cost::Time() {
     return Cycles() * Chip->Cycle;
  }
```

G.4. « FFT2.CPP »

Une manière assez simple de calculer la FFT :

```
/*********************************************************
  MODULE : FFT2.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Implementation of the cost class
  SOURCE : Originally in FORTRAN from
       Digital Spectral Analysis with Applications
       S. Lawrence Marple, Jr.
*********************************************************/

//   These two programs set up the complex exponential table (fft2_init) and
//  compute the discrete-time Fourier series of an array of complex data
//  samples using a decimation-in-frequency fast Fourier transform (FFT)
//  algorithm.

#include <iostream.h>
#include "utildef.h"
#include "fft.h"
#include "fft2.h"

void fft2_init(int bits, bool Inverse,  complex huge *W) {
//    INV   - Set to 0 for discrete-time Fourier series or 1 for inverse
//    bits  - Indicates power-of-2 exponent such that N=2**bits .
//                 N: Number of data samples to be processed (integer-must be a
//            power of two)
//    W     - Complex exponential array

  int N= 1 << bits;

//  cout << "Dans fft2_init" << endl;

  double s= 8. * atan (1.) / double(N);
  complex C1;
  C1= complex(cos(s), sin(s));
  if (!Inverse)
     C1= conj(C1);

  complex C2(1.0);

  for(int k= 0; k < N; k++) {
     W[k]= C2;
     C2= C2 * C1;
  }
}

void fft2(int bits, bool Inverse, complex huge *W, complex huge *X, double T) {
//    T             - Sample interval in seconds dor normalizing in Hertz
//    X             - Array of N complex data samples, X(1) to X(N)
//                                        N complex transform values replace original data samples
//                                indexed from k=1 to k=N, representing the frequencies
//                                    (k-1)/N hertz

              complex C1, C2;
              int N= 1 << bits;

//            cout << "Dans fft2" << endl;

              int MM= 1;
              int LL= N;

              for(int k= 0; k < bits; k++) {
                int NN= LL/2;
                int JJ= MM;

                for(int i= 0; i < N; i+= LL) {
                         int KK= i + NN;
                         C1= X[i] + X[KK];
                         X[KK]= X[i] - X[KK];
                         X[i]=C1;
                }
                if (NN == 1)
                         continue;

                for(int j= 1; j < NN; j++) {
                         C2= W[JJ];
                         for(int i= j; i < N; i+= LL) {
                                  int KK= i + NN;
                                  C1= X[i] + X[KK];
                                  X[KK]= (X[i] - X[KK]) * C2;
                                  X[i]= C1;
                         }
                         JJ+= MM;
                }
                LL= NN;
                MM*= 2;
              }

//            cout << endl << "Première bucle: " << N << endl;

              int NV2= N/2;
              int NM1= N-1;
              int j= 0;
              for(int i= 0; i < NM1; i++) {
                if (i < j) {
                         C1= X[j];
                         X[j]= X[i];
                         X[i]= C1;
                }
                int k= NV2;
```

```
                while (k < j + 1) {
                        j-= k;
                        k/= 2;
                }
                j+= k;
              }

//            cout << endl << "Deuxième bucle: " << N << endl;



              // Normalization in Hertz

              double s;
              if (!Inverse)
                s= T;
              else
                s= 1./(T * double(N));

              for(int n= 0; n < N; n++) {
                X[n]= X[n] * s;
              }

}

void test_fft2() {
/*
  complex X[64];
  complex W[64];
  complex Y[64];

                        X[ 1-1] = complex(  1.349839091,  2.011167288);
                        X[ 2-1] = complex( -2.117270231,  0.817693591);
                        X[ 3-1] = complex( -1.786421657, -1.291698933);
                        X[ 4-1] = complex(  1.162236333, -1.482598066);
                        X[ 5-1] = complex(  1.641072035,  0.372950256);
                        X[ 6-1] = complex(  0.072213709,  1.828492761);
                        X[ 7-1] = complex( -1.564284801,  0.824533045);
                        X[ 8-1] = complex( -1.080565453, -1.869776845);
                        X[ 9-1] = complex(  0.927129090, -1.743406534);
                        X[10-1] = complex(  1.891979456,  0.972347319);
                        X[11-1] = complex( -0.105391249,  1.602209687);
                        X[12-1] = complex( -1.618367076,  0.637513280);
                        X[13-1] = complex( -0.945704579, -1.079569221);
                        X[14-1] = complex(  1.135566235, -1.692269921);
                        X[15-1] = complex(  1.855816245,  0.986030221);
                        X[16-1] = complex( -1.032083511,  1.414613724);
                        X[17-1] = complex( -1.571600199,  0.089229003);
                        X[18-1] = complex( -0.243143231, -1.444692016);
                        X[19-1] = complex(  0.838980973, -0.985756695);
                        X[20-1] = complex(  1.516003132,  0.928058863);
                        X[21-1] = complex(  0.257979959,  1.170676708);
                        X[22-1] = complex( -2.057927608,  0.343388647);
                        X[23-1] = complex( -0.578682184, -1.441192508);
                        X[24-1] = complex(  1.584011555, -1.011150956);
                        X[25-1] = complex(  0.614114344,  1.508176208);
                        X[26-1] = complex( -0.710567117,  1.130144477);
                        X[27-1] = complex( -1.100205779, -0.584209621);
                        X[28-1] = complex(  0.150702029, -1.217450142);
                        X[29-1] = complex(  0.748856127, -0.804411888);
                        X[30-1] = complex(  0.795235813,  1.114466429);
                        X[31-1] = complex( -0.071512341,  1.017092347);
                        X[32-1] = complex( -1.732939839, -0.283070654);
                        X[33-1] = complex(  0.404945314, -0.781708360);
                        X[34-1] = complex(  1.293794155, -0.352723092);
                        X[35-1] = complex( -0.119905084,  0.905150294);
                        X[36-1] = complex( -0.522588372,  0.437393665);
                        X[37-1] = complex( -0.974838495, -0.670074046);
                        X[38-1] = complex(  0.275279552, -0.509659231);
                        X[39-1] = complex(  0.854210198, -0.008278057);
                        X[40-1] = complex(  0.289598197,  0.506233990);
                        X[41-1] = complex( -0.283553183,  0.250371397);
                        X[42-1] = complex( -0.359602571, -0.135261074);
                        X[43-1] = complex(  0.102775671, -0.466086507);
                        X[44-1] = complex( -0.009722650,  0.030377999);
                        X[45-1] = complex(  0.185930878,  0.808869600);
                        X[46-1] = complex( -0.243692726, -0.200126961);
                        X[47-1] = complex( -0.270986766, -0.460243553);
                        X[48-1] = complex(  0.399368525,  0.249096692);
                        X[49-1] = complex( -0.250714004, -0.362990230);
                        X[50-1] = complex(  0.419116348, -0.389185309);
                        X[51-1] = complex( -0.050458215,  0.702862442);
                        X[52-1] = complex( -0.395043731,  0.140808776);
                        X[53-1] = complex(  0.746575892, -0.126762003);
                        X[54-1] = complex( -0.559076190,  0.523169816);
                        X[55-1] = complex( -0.344389260, -0.913451135);
                        X[56-1] = complex(  0.733228028, -0.006237417);
                        X[57-1] = complex( -0.480273813,  0.509469569);
                        X[58-1] = complex(  0.033316225,  0.087501869);
                        X[59-1] = complex( -0.321229130, -0.254548967);
                        X[60-1] = complex( -0.063007891, -0.499800682);
                        X[61-1] = complex(  1.239739418, -0.013479125);
                        X[62-1] = complex(  0.083303742,  0.673984587);
                        X[63-1] = complex( -0.762731433,  0.408971250);
                        X[64-1] = complex( -0.895898521, -0.364855707);

  cout << "\n\n\rTesting fft2...\n";

  for (int i= 0; i < 64; i++) {
     Y[i]= X[i];
  }

  fft2_init(6, DIRECT, W);

  for (i= 0; i < 64; i++) {
     cout << "\rW[" << i << "]= { " << W[i].re << ", " << W[i].im << " } " << endl;
  }
  Pause();
```

```
  fft2(6, DIRECT, W, X, 1.0);

  for (i= 0; i < 64; i++) {
     cout << "\rX[" << i << "]= { " << X[i].re << ", " << X[i].im << " } " << endl;
  }

  Pause();

  fft2_init(6, INVERSE, W);
  fft2(6, INVERSE, W, X, 1.0);

  for (i= 0; i < 64; i++) {
     if ((abs(Y[i].re - X[i].re) > 0.01) ||
              (abs(Y[i].im - X[i].im) > 0.01))
             cout << "\n\rERROR en test_fft2! a la valeur i= " << i << ", X:" << X[i].re << ", " <<
X[i].im << endl;
  }
*/
}
```

G.5. « NEWCODER.CPP »

Permet la réalisation des nouveaux algorithhmes de compression dans l'outil Accordion :

```
/*********************************************************
  MODULE : NEWCODER.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Function for implementing new algorithms
*********************************************************/

#include "newcoder.h"

void new_coder(sample_array *, long, BYTE huge *, long, struct new_state *, int Info, cost & COST) {
}

void new_decoder(BYTE huge *, sample_array *, long, long, struct new_state *, int Info, cost & COST) {
}
```

G.6. « PITCH.CPP »

Algorithme détecteur du pitch :

```
/*********************************************************
  MODULE : PITCH.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Pitch Detectors
*********************************************************/

#include "pitch.h"
#include "sample.h"

double Pitch_Auto(sample_array *s, double sampleFreq, double minPer, double maxPer, double resPer,
double Left, double Right) {
  double MaxPer= minPer;
  double MaxCor= 0.0;
  long l= s->NumSamples * Left;
  long r= s->NumSamples * Right;

  for (double Per= minPer; Per <= maxPer; Per+= resPer) {
     long SamplesPerBlock= Per * sampleFreq / 1000.0;
     double Cor= 0.0;
     for (long n= l; n < r - SamplesPerBlock; n++) {
              Cor+= double(s->Get(n)) * double(s->Get(n + SamplesPerBlock));
     }
     if (Cor > MaxCor) {
              MaxPer= Per;
              MaxCor= Cor;
     }
  }

  return MaxPer;
}

double Pitch_Simple(sample_array *s, double sampleFreq, double minPer, double maxPer, double resPer,
double Left, double Right) {
  double MaxPer= minPer;
  double MaxCor= -1.0;
  long l= s->NumSamples * Left;
  long r= s->NumSamples * Right;

  for (double Per= minPer; Per <= maxPer; Per+= resPer) {
     long SamplesPerBlock= Per * sampleFreq / 1000.0;
     double Cor= 0.0;
     for (long n= l; n < r - SamplesPerBlock; n++) {
              Cor+= abs(double(s->Get(n)) - double(s->Get(n + SamplesPerBlock)));
     }
     if ((Cor < MaxCor) || (MaxCor == -1.0)){
              MaxPer= Per;
              MaxCor= Cor;
     }
  }

  return MaxPer;
}
```

G.7. « POLYNOM.H » et « POLYNOM.CPP »

Ce programme implante les filtres et les formules en z et en s :

```
/*********************************************************
  MODULE : POLYNOM.H
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Declaration of classe polynom and filter
*********************************************************/

#ifndef _POLYNOM_H_
#define _POLYNOM_H_

class filter;

class polynom {
private:
  double *C;
  static double ErrorValue;

  void Copy(const polynom & p);
  void Construct(int n, double * d);
  void Destruct();
public:
  int Order;
public:
  polynom(int n, double * d);
  polynom();
  polynom(double d);
  polynom(const polynom & p);
  ~polynom();

  void operator = (const polynom & p);
  const double & operator [] (int i) const;
  double & operator [] (int i);

  polynom operator - () const;
    // Change sign
  polynom operator + (const polynom & p) const;                          // Addition
  polynom operator - (const polynom & p) const;                          // Substraction
  polynom operator * (const polynom & p) const;                          // Multiplication
  filter operator / (const polynom & p) const ;                          // Division

  polynom operator ^ (int i) const;
    // Power

  polynom operator & (int i) const;
    // Add Order

  polynom & operator += (const polynom & p) {
     *this= *this + p;
     return *this;
  }

  polynom & operator -= (const polynom & p) {
     *this= *this - p;
     return *this;
  }

  polynom & operator *= (const polynom & p) {
     *this= *this * p;
     return *this;
  }

  polynom operator () (const polynom & p) const;              // Substitution

  int Print(char *s, char *Var);
};

class filter {
public:
  polynom Num, Den;
public:
  filter(polynom n= polynom(1.0), polynom d= polynom(1.0));

  filter operator - () const;
    // Change sign
  filter operator + (const filter & p) const;                            // Addition
  filter operator - (const filter & p) const;                            // Substraction
  filter operator * (const filter & p) const;                            // Multiplication
  filter operator / (const filter & p) const ;                           // Division

  filter operator ^ (int i) const;
    // Power

  filter operator & (int i) const;
    // Add Order

  filter & operator += (const filter & p) {
     *this= *this + p;
     return *this;
  }

  filter & operator -= (const filter & p) {
     *this= *this - p;
     return *this;
  }

  filter & operator *= (const filter & p) {
     *this= *this * p;
     return *this;
  }
```

```
  filter & operator /= (const filter & p) {
     *this= *this / p;
     return *this;
  }

  filter operator () (const filter & p) const;                // Substitution

  int Print(char *s, char *Var);

  static filter Butterworth(int i);        // Butterworth Order i
  static filter Tchebychev(int i);                  // Tchebychev Order i
  static filter ITchebychev(int i);                 // Tchebychev inverse Order i
  static filter Cauer(int i);                               // Cauer Order i

  void filter::Discret(double T);

  void filter::LowPass(double T, double Freq);
  void filter::HighPass(double T, double Freq);
  void filter::BandPass(double T, double Freq, double FreqWidth);
  void filter::CutBand(double T, double Freq, double FreqWidth);

};

#endif
```

```
/*********************************************************
  MODULE : POLYNOM.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Implementation of classe polynom and filter
*********************************************************/

#include "polynom.h"
#include "utildef.h"

#include <math.h>


void polynom::Copy(const polynom & p) {
     Order= p.Order;
     C= new double[p.Order];
     for (int i= 0; i < Order; i++) {
               C[i]= p[i];
     }
}

void polynom::Construct(int n, double * d) {
     Order= n;
     C= new double[n];
     for(int i= 0; i < n; i++) {
               if (d)
                 C[i]= d[i];
               else
                 C[i]= 0.;
     }
}

void polynom::Destruct() {
     if (C)
               delete C;
}

polynom::polynom(int n, double * d) {
     Construct(n, d);
}

polynom::polynom() {
     Order= 0;
     C= 0;
}

polynom::polynom(double d) {
     Construct(1, &d);
}

polynom::polynom(const polynom & p) {
     Copy(p);
}

polynom::~polynom() {
     Destruct();
}

void polynom::operator = (const polynom & p) {
     if (&p != this)
               Destruct();
     Copy(p);
}

const double & polynom::operator [] (int i) const {
     if (i < Order) {
               return C[i];
     }
     else
               return ErrorValue;
}

double & polynom::operator [] (int i) {
     if (i < Order) {
               return C[i];
     }
     else
               return ErrorValue;
}
```

```
polynom polynom::operator + (const polynom & p) const {
     polynom r;
     if (p.Order > Order) {
               r= p;
               for(int i= 0; i < Order; i++) {
                 r[i]+= C[i];
               }
     }
     else {
               r= *this;
               for(int i= 0; i < p.Order; i++) {
                 r[i]+= p[i];
               }
     }
     return r;
}

polynom polynom::operator - () const {
     polynom r(*this);
     for(int i= 0; i < Order; i++) {
               r[i]= -r[i];
     }
     return r;
}

polynom polynom::operator - (const polynom & p) const {
  return *this + -p;
}

polynom polynom::operator * (const polynom & p) const {
     polynom r(p.Order + Order - 1, NULL);
     for(int i= 0; i < Order; i++) {
               for(int j= 0; j < p.Order; j++) {
                 r[i + j]+= C[i] * p[j];
               }
     }
     return r;
}

polynom polynom::operator ^ (int i) const {
     polynom r(1.);
     while(i--) {
               r= r * *this;
     }
     return r;
}

polynom polynom::operator & (int i) const {
     polynom r(Order + i, NULL);
     for (int j= 0; j < Order; j++)  {
               r[j + i]= C[j];
     }
     return r;
}

filter polynom::operator / (const polynom & p) const {
     return filter(*this, p);
}

polynom polynom::operator () (const polynom & p) const {
  polynom r(p.Order * Order, 0);
     for(int i= 0; i < Order; i++) {
               r= r + p&i * C[i];
     }
     return r;
}


double polynom::ErrorValue= 0.0;

filter::filter(polynom n, polynom d) : Num(n), Den(d) {
}

filter filter::operator () (const filter & f) const {
  int MultOrder= Max(Num.Order, Den.Order) - 1;

  polynom n;
  for(int i= 0; i < Num.Order; i++) {
     polynom p0= (f.Num)^i;
     polynom p1= (f.Den)^(MultOrder - i);
     polynom p2= p0 * p1;
     polynom p3= p2 * Num[i];
     polynom p4= n + p3;
     n= p4;
  }

  polynom d;
  for(i= 0; i < Den.Order; i++) {
     polynom p0= (f.Num)^i;
     polynom p1= (f.Den)^(MultOrder - i);
     polynom p2= p0 * p1;
     polynom p3= p2 * Den[i];
     polynom p4= d + p3;
     d= p4;
  }

  return filter(n, d);
}

filter filter::operator * (const filter & f) const {
  return filter(f.Num * Num, f.Den * Den);
}

filter filter::operator / (const filter & f) const {
  return filter(Num * f.Den, Den * f.Num);
}

filter filter::operator + (const filter & f) const {
  return filter(Num * f.Den + f.Num * Den, f.Den * Den);
```

```
}

filter filter::operator - (const filter & f) const {
  return filter(Num * f.Den - f.Num * Den, f.Den * Den);
}

filter filter::operator - () const{
  return filter(-Num, Den);
}

filter filter::operator ^ (int i) const{
  return filter(Num^i, Den^i);
}

filter filter::operator & (int i) const{
  return filter(Num&i, Den&i);
}

int polynom::Print(char *s, char *Var) {
  int l= 0;
  for (int i= Order - 1; i > 0; i--) {
     int n;
     n= sprintf(s, "%4.4f %s%d + ", C[i], Var, i);
     s+= n;
     l+= n;
  }
  if (Order > 0) {
     int n= sprintf(s, "%4.4f", C[0], Var, i);
     s+= n;
     l+= n;
  }
  else {
     *s++= '1';
     l++;
  }
  return l;
}


int filter::Print(char *s, char *Var) {
  int n= Num.Print(s, Var);
  s+= n;
  int l= 2*n + 2;
  *s++= '\n';
  while (n--) {
    *s++= '-';
  }
  *s++= '\n';
  int d= Den.Print(s, Var);
  s+= d;
  *s= '\0';
  l+= d;
  return l;
}


filter filter::Butterworth(int i) {        // Butterworth Order i
  if (i == 2) {
     polynom Deno(3, NULL);
     Deno[0]= 1.;
     Deno[1]= 2.;
     Deno[2]= 3.;
     return filter(polynom(1.0), Deno);
  }
  else if (i == 3) {
     polynom Deno(4, NULL);
     Deno[0]= 1.;
     Deno[1]= 2.;
     Deno[2]= 2.;
     Deno[3]= 1.;
     return filter(polynom(1.0), Deno);
  }
  else if (i == 4) {
     polynom Deno(5, NULL);
     Deno[0]= 1.;
     Deno[1]= 2.6131;
     Deno[2]= 3.4142;
     Deno[3]= 2.6131;
     Deno[4]= 1.;
     return filter(polynom(1.0), Deno);
  }
  else if (i == 5) {
     polynom Deno(6, NULL);
     Deno[0]= 1.;
     Deno[1]= 3.2361;
     Deno[2]= 5.2361;
     Deno[3]= 5.2361;
     Deno[4]= 3.2361;
     Deno[5]= 1.;
     return filter(polynom(1.0), Deno);
  }
  else {               // i == 6
     polynom Deno(7, NULL);
     Deno[0]= 1.;
     Deno[1]= 3.8637;
     Deno[2]= 7.4641;
     Deno[3]= 9.1416;
     Deno[4]= 7.4641;
     Deno[5]= 3.8637;
     Deno[6]= 1;
     return filter(polynom(1.0), Deno);
  }
}

filter filter::Tchebychev(int i) {         // Tchebychev Order i for 1 dB ondulation
  if (i == 2) {
     polynom Deno(3, NULL);
     Deno[0]= 1.;
     Deno[1]= 0.9957;
```

```
    Deno[2]= 0.907;
    return filter(polynom(1.0), Deno);
  }
  else if (i == 3) {
    polynom Deno(4, NULL);
    Deno[0]= 1.;
    Deno[1]= 2.5206;
    Deno[2]= 2.0116;
    Deno[3]= 2.0353;
    return filter(polynom(1.0), Deno);
  }
  else if (i == 4) {
    polynom Deno(5, NULL);
    Deno[0]= 1.;
    Deno[1]= 2.6942;
    Deno[2]= 5.2749;
    Deno[3]= 3.4568;
    Deno[4]= 3.628;
    return filter(polynom(1.0), Deno);
  }
  else if (i == 5) {
    polynom Deno(6, NULL);
    Deno[0]= 1.;
    Deno[1]= 4.7264;
    Deno[2]= 7.933;
    Deno[3]= 13.75;
    Deno[4]= 7.6271;
    Deno[5]= 8.1415;
    return filter(polynom(1.0), Deno);
  }
  else {                // i == 6
    polynom Deno(7, NULL);
    Deno[0]= 1.;
    Deno[1]= 4.456;
    Deno[2]= 13.632;
    Deno[3]= 17.445;
    Deno[4]= 28.02;
    Deno[5]= 13.47;
    Deno[6]= 14.512;
    return filter(polynom(1.0), Deno);
  }
}

filter filter::ITchebychev(int i) {        // Tchebychev inverse Order i for 1dB Ondulation
  if (i == 3) {
    polynom Nume(4, NULL);
    Nume[0]= 1;
    Nume[1]= 0.;
    Nume[2]= 0.75;
    Nume[3]= 0;
    polynom Deno(4, NULL);
    Deno[0]= 1.;
    Deno[1]= 8.4672;
    Deno[2]= 36.597;
    Deno[3]= 79.05;
    return filter(Nume, Deno);
  }
  else if (i == 4) {
    polynom Nume(5, NULL);
    Nume[0]= 1;
    Nume[1]= 0.;
    Nume[2]= 1.;
    Nume[3]= 0.;
    Nume[4]= 0.125;
    polynom Deno(5, NULL);
    Deno[0]= 1.;
    Deno[1]= 6.2917;
    Deno[2]= 20.7925;
    Deno[3]= 40.540;
    Deno[4]= 39.5285;
    return filter(Nume, Deno);
  }
  else if (i == 5) {
    polynom Nume(6, NULL);
    Nume[0]= 1;
    Nume[1]= 0.;
    Nume[2]= 1.25;
    Nume[3]= 0;
    Nume[4]= 0.3125;
    Nume[5]= 0.;
    polynom Deno(6, NULL);
    Deno[0]= 1.;
    Deno[1]= 5.4321;
    Deno[2]= 16.0037;
    Deno[3]= 29.679;
    Deno[4]= 34.253;
    Deno[5]= 19.764;
    return filter(Nume, Deno);
  }
  else { // i == 6
    polynom Nume(7, NULL);
    Nume[0]= 1;
    Nume[1]= 0.;
    Nume[2]= 1.5;
    Nume[3]= 0;
    Nume[4]= 0.5625;
    Nume[5]= 0.;
    Nume[6]= 0.0312;
    polynom Deno(7, NULL);
    Deno[0]= 1.;
    Deno[1]= 5.0006;
    Deno[2]= 14.0028;
    Deno[3]= 25.612;
    Deno[4]= 31.720;
    Deno[5]= 25.038;
    Deno[6]= 9.8821;
    return filter(Nume, Deno);
  }
}
```

```
filter filter::Cauer(int i) {     // Cauer Order i for 1db Ondulation
  if (i == 3) {
     polynom Nume(4, NULL);
     Nume[0]= 1;
     Nume[1]= 0.;
     Nume[2]= 0.1038;
     Nume[3]= 0;
     polynom Deno(4, NULL);
     Deno[0]= 1.;
     Deno[1]= 2.0235;
     Deno[2]= 1.6404;
     Deno[3]= 1.3228;
     return filter(Nume, Deno);
  }
  else if (i == 4) {
     polynom Nume(5, NULL);
     Nume[0]= 1;
     Nume[1]= 0.;
     Nume[2]= 0.3993;
     Nume[3]= 0.;
     Nume[4]= 0.0226;
     polynom Deno(5, NULL);
     Deno[0]= 1.;
     Deno[1]= 2.2818;
     Deno[2]= 3.6938;
     Deno[3]= 2.4801;
     Deno[4]= 2.1035;
     return filter(Nume, Deno);
  }
  else if (i == 5) {
     polynom Nume(6, NULL);
     Nume[0]= 1;
     Nume[1]= 0.;
     Nume[2]= 0.8633;
     Nume[3]= 0;
     Nume[4]= 0.1642;
     Nume[5]= 0.;
     polynom Deno(6, NULL);
     Deno[0]= 1.;
     Deno[1]= 3.118;
     Deno[2]= 4.7536;
     Deno[3]= 6.6659;
     Deno[4]= 3.7273;
     Deno[5]= 3.2301;
     return filter(Nume, Deno);
  }
  else { // i == 6
     polynom Nume(7, NULL);
     Nume[0]= 1;
     Nume[1]= 0.;
     Nume[2]= 1.403;
     Nume[3]= 0;
     Nume[4]= 0.5422;
     Nume[5]= 0.;
     Nume[6]= 0.0413;
     polynom Deno(7, NULL);
     Deno[0]= 1.;
     Deno[1]= 2.8728;
     Deno[2]= 6.4997;
     Deno[3]= 7.3627;
     Deno[4]= 9.4823;
     Deno[5]= 4.4452;
     Deno[6]= 3.8954;
     return filter(Nume, Deno);
  }
}

void filter::Discret(double T) {
  polynom dn(2, NULL);
  dn[0]= 2;
  dn[1]= -2;
  polynom dd(2, NULL);
  dd[0]= T;
  dd[1]= T;

  filter Discret(dn, dd);
  *this= (*this)(Discret);
}

void filter::LowPass(double T, double Freq) {
  polynom s(2, NULL);
  s[0]= 0.;
  s[1]= 1.;
  Freq= (1 / (PI * T)) * tan (PI * T * Freq);

  filter Subst(s, polynom(2 * PI * Freq));
  *this= (*this)(Subst);
}

void filter::HighPass(double T, double Freq) {
  polynom s(2, NULL);
  s[0]= 0.;
  s[1]= 1.;

  Freq= (1 / (PI * T)) * tan (PI * T * Freq);
  filter Subst(polynom(2 * PI * Freq), s);
  *this= (*this)(Subst);
}

void filter::BandPass(double T, double Freq, double FreqWidth) {
  polynom n(3, NULL);
  n[0]= 1.;
  n[1]= 0.;
  n[2]= 1.;
  polynom d(2, NULL);
  d[0]= 0.;
  d[1]= 2 * PI * FreqWidth;
  filter Subst(n, d);
```

```
  Subst.LowPass(T, Freq);
  *this= (*this)(Subst);
}

void filter::CutBand(double T, double Freq, double FreqWidth) {
  polynom n(3, NULL);
  n[0]= 1.;
  n[1]= 0.;
  n[2]= 1.;
  polynom d(2, NULL);
  d[0]= 0.;
  d[1]= 2 * PI * FreqWidth;
  filter Subst(n, d);
  Subst.HighPass(T, Freq);
  *this= (*this)(Subst);
}
```

G.8. « RATE.CPP »

La vitesse variable temporelle :

```
/*********************************************************
  MODULE : RATE.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Implementation of the algorithms of variable rate
*********************************************************/

#include <fstream.h>
#include <iomanip.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#include "rate.h"
#include "sample.h"

typedef unsigned long dword;
typedef unsigned int word;
typedef unsigned char byte;

const unsigned long new_LIMIT= 30000u;



void Compensation(int sense, methode m, double SMP2MS,
                                                    unsigned long l1, unsigned long l2, unsigned long
& lc,
                                                    sample huge * p1, sample huge * p2, sample huge *
pc){


  lc= m.CompensF * l1 + (1. - m.CompensF) * l2;
  // Une autre manière moins équitative
  // lc= l2;

  sample x1, x2, xc;

/*
  if (Visualiser)
     cout << "\n\n\rCompensat: " << l1 << ", " << l2 << ", " << lc << " Sense:" << sense;
*/

  for (unsigned long i= 0; i < lc; i++) {
     double r= double(i) / double(lc);
     double f;

     x1= p1[(unsigned long)(r * l1)];
     x2= p2[(unsigned long)(r * l2)];

     if (sense == -1) {
              r= 1. - r;
     }

     switch (m.Fenetre_Comp) {
              case CARREE:                          // fenêtre carrée  --
                                                                                  f= m.CompensF * 1.;
                                                                                  break;

// NO ME ACLARO A LA HORA DE PONDERAR EL FACTOR

              case TRIANGULAIRE:   // fenêtre triangulaire  \/
                                                                                  f= 2 * m.CompensF *
(1.0 - r);
                                                                                  break;
              case POLYNOMIQUE:            //  fenêtre  polynomiale  (approximation  à  la  fenêtre  de
Hanning)
                                                                                  f= 2 * m.CompensF *
(1 + 0.125 * r - 3.375 * r * r + 2.25 * r * r * r);
                                                                                  break;
              default:                                          // cerr << "Fenêtre unconnue";
                                                                                  break;
     };


     xc= f * double(x1) + (1. - f) * double(x2);

     pc[(unsigned long)i]= xc;

/*
     if (Visualiser)
              if ((i == 0) || (i == lc - 1))
                cout << "\n\r     i:" << i << ", r:" << r << ", f: " << f << ", x: " << (double) x1 <<
"," <<(double) x2 << "," << (double) xc;
*/
  }

/*
  if (Visualiser) {
     x1.Lire(p1, (unsigned long)(l1 - 1));
     x2.Lire(p2, (unsigned long)(l2 - 1));
     cout << "\n\r      xf: " << (double)x1 << ", " << (double)x2;
  }
*/

/*
  if (Visualiser) {
     echantillon xa, xb;
     xa.Lire(pc, 0);
```

```
    xb.Lire(pc, lc - 1);
    cout << endl << "     "
                        << " Tai:" << lc * SMP2MS
                        << " Beg:" << (double)xa
                        << " End:" << (double)xb << endl;
  }
*/

}


void Partition(methode m, unsigned long PitchPos, unsigned long MinPitchPos, unsigned long
MaxLong_Morceau,
                                            sample huge * ComMorceau, double SPL2MS, double
TAILLE_ECHANT,
                                            unsigned long & MelPos, double & MelQual,
                                            double & Tail_Mel, double & Freq_Mel, double & Puis_Mel,
double & Ampl_Mel, double & Pend_Mel,
                                            double & FreqMorceau, double & PuisMorceau, cost & COST)
{

  sample x0, x1, x2;
  x0= x1= x2= 0.;

  double Ant_Tail, Tail;

  double Ant_Freq, Freq;
  double FreqI= 0.;
  double PosVent_Freq= 0.;

  double Ant_Puis, Puis;
  double PuisI= 0.;
  double PosVent_Puis= 0.;

  double Ant_Ampl, Ampl;

  double Ant_Pend, Pend;

  Tail= Ant_Tail= Tail_Mel;
  Freq= Ant_Freq= Freq_Mel;
  Puis= Ant_Puis= Puis_Mel;
  Ampl= Ant_Ampl= Ampl_Mel;
  Pend= Ant_Pend= Pend_Mel;

  MelPos= Min(PitchPos, MaxLong_Morceau);

  double Qual= 0.;
  MelQual= 0.;
  double MaxTot= 0.0, MaxTot_Mel;
  double AreaTot= 0.0, AreaTot_Mel;

  unsigned long Pos= 0;
  while (Pos < MaxLong_Morceau) {
     x2= ComMorceau[Pos];
     Pos++;

     double MaxErrorTail= m.MAXPITCHSIZE - m.MINPITCHSIZE;
     Tail= Pos * SPL2MS;                    // De 0 à MAXPITCHSIZE
     double qTail= 1.0 - Abs((Pond_Arit(m.Comp_Tail, Ant_Tail, m.Ref_Tail) - Tail)) / MaxErrorTail;

     Ampl= x2;
     double qAmpl= 1.0 - Abs((Pond_Arit(m.Comp_Ampl, Ant_Ampl, m.Ref_Ampl) - Ampl)) / 2;

     Pend= atan((double(x2) - double(x1)) / TAILLE_ECHANT);  // De -PI/2 à PI/2
     double qPend= 1.0 - Abs((Pond_Arit(m.Comp_Pend, Ant_Pend, m.Ref_Pend) - Pend) / PI);

     /*
              DETECTAR LOS MAXIMOS.
     */
     int CestMax;

     if ((double(x1) > double(x0)) && (double(x1) > double(x2)))
              CestMax= 1;
     else
              CestMax= 0;
     x0= x1;
     x1= x2;

     FreqI+= CestMax;
     MaxTot+= CestMax;
     PosVent_Freq+= SPL2MS;

     if (PosVent_Freq >= m.Vent_Freq) {
              Freq= Pond_Arit(m.Memo_Freq, Freq, FreqI / m.Vent_Freq);
              FreqI= 0.;
              PosVent_Freq= 0.;
     }
     double qFreq= exp(-Abs((Pond_Arit(m.Comp_Freq, Ant_Freq, m.Ref_Freq) - Freq)));

     PuisI+= Sqr(Ampl) * TAILLE_ECHANT;
     AreaTot+= Sqr(Ampl) * TAILLE_ECHANT;
     PosVent_Puis+= SPL2MS;
     if (PosVent_Puis >= m.Vent_Puis) {
              Puis= Pond_Arit(m.Memo_Puis, Puis, PuisI / m.Vent_Puis);
              PuisI= 0.;
              PosVent_Puis= 0.;
     }
     double qPuis= 1.0 - Abs((Pond_Arit(m.Comp_Puis, Ant_Puis, m.Ref_Puis) - Puis));

     double Qual= pow(qTail, m.Fact_Tail) *
                                            pow(qAmpl, m.Fact_Ampl) *
                                            pow(qPend, m.Fact_Pend) *
                                            pow(qFreq, m.Fact_Freq) *
                                            pow(qPuis, m.Fact_Puis);

     if ((Qual > MelQual) && (Pos > MinPitchPos)) {
              MelQual= Qual;
              MelPos= Pos;
              Tail_Mel= Tail;
```

```
              Freq_Mel= Freq;
              Puis_Mel= Puis;
              Ampl_Mel= Ampl;
//                                  cout << "             Ampl:" << Ampl;
              Pend_Mel= Pend;
              MaxTot_Mel= MaxTot;
              AreaTot_Mel= AreaTot;
     }
   }

   if (m.Partition == UNIFORME) {
      MelQual= Qual;
      MelPos= Pos;
      Tail_Mel= Tail;
      Freq_Mel= Freq;
      Puis_Mel= Puis;
      Ampl_Mel= Ampl;
//                                  cout << "             Ampl:" << Ampl;
      Pend_Mel= Pend;
      MaxTot_Mel= MaxTot;
      AreaTot_Mel= AreaTot;
   }

   FreqMorceau= MaxTot_Mel / Tail_Mel;
   PuisMorceau= AreaTot_Mel / Tail_Mel;
}



void DoRate(sample_array * Input, sample_array * Output, methode &m, double SampleRate, cost & COST) {
   sample huge *ComMorceau;
   sample huge *ComMorceauComp;
   sample huge *Base;
   sample huge * ComMorceauDern;
   unsigned long iSource= 0L;
   unsigned long LongSource= Input->NumSamples;
   unsigned long iCible= 0L;
   unsigned long LongCible= Output->NumSamples;
   double Avance= 0.0, Retard= 0.0;
   double SumDevG= 0.;
   double SumDevD= 0.;
   unsigned long NumMorceaux= 0;
   double PuisMoyenne, FreqMoyenne;
   unsigned long LongMorceau;
   double TAILLE_ECHANT= 1000. / SampleRate;
   double SPL2MS= 1000. / SampleRate;
   double MS2SPL= SampleRate / 1000.;

   long Rep;

   unsigned long MaxPitchPos= m.MAXPITCHSIZE * MS2SPL;
   unsigned long PitchPos= m.PITCHSIZE * MS2SPL;
   unsigned long MinPitchPos= m.MINPITCHSIZE * MS2SPL;

   unsigned long BUFFERSIZE= MaxPitchPos;
   ComMorceau= new sample[BUFFERSIZE];
   ComMorceauComp= new sample[BUFFERSIZE];
   ComMorceauDern= new sample[BUFFERSIZE];
   unsigned long MelPos;
   unsigned long MelPosDern;
   unsigned long MelPosComp;

   double Dev= 1/m.Taux;

   srand(15000);

   double Tail_Mel;
   Tail_Mel= m.Ref_Tail;

   double Freq_Mel;
   Freq_Mel= m.Ref_Freq;

   double Puis_Mel;
   Puis_Mel= m.Ref_Puis;

   double Ampl_Mel;
   Ampl_Mel= m.Ref_Ampl;

   double Pend_Mel;
   Pend_Mel= m.Ref_Pend;

   double PuisTotal= 0.0, PuisTipique= m.PUIS_TIPIQUE;
   double FreqTotal= 0.0, FreqTipique= m.FREQ_TIPIQUE;

   int Compensat= -1;
   double ErrArrondi= 0.0;

   while (iSource < LongSource) {
      unsigned long l;
      if (m.Partition == UNIFORME) {
               l= m.PARTITIONSIZE * MS2SPL;
      }
      else if (m.Partition == VARIABLE)
               l= MaxPitchPos;
      else {
               l= l;
               // cerr << "Partition pas reconnue" << endl;
      }

      unsigned long MaxLong_Morceau= Min(l, LongSource - iSource);
      for (l= 0; l < MaxLong_Morceau; l++) {
               ComMorceau[l]= Input->Get(iSource + l);
      }

      double FreqMorceau;
      double PuisMorceau;
      double MelQual= 0.0;

      Partition(m, PitchPos, MinPitchPos, MaxLong_Morceau, ComMorceau,
```

```
                                                SPL2MS, TAILLE_ECHANT,
                                                MelPos, MelQual,
                                                Tail_Mel,    Freq_Mel,    Puis_Mel,    Ampl_Mel,
Pend_Mel,
                                                FreqMorceau, PuisMorceau, COST);

     NumMorceaux++;

     iSource+= MelPos;

     PuisTotal+= PuisMorceau;
     PuisMoyenne= PuisTotal / NumMorceaux;

     FreqTotal+= FreqMorceau;
     FreqMoyenne= FreqTotal / NumMorceaux;

     FreqTipique= 0.98 * FreqTipique + 0.02 * FreqMoyenne;
     double fFreq= 1. - exp(-FreqMorceau / FreqTipique);
    // De 0 a 1

     PuisTipique= 0.98 * PuisTipique + 0.02 * PuisMoyenne;
     double fPuis= 1. - exp(-PuisMorceau / PuisTipique);                              // De 0 à 1

     double fCarac= fFreq * fPuis;                                                 //  La  petite  la
fréquence, le plus characteristique que c'est le morceau

     double Rep1, Rep2, Rep3;

     double Dev0= ((iSource / m.Taux) - iCible) /(m.PITCHSIZE * MS2SPL) ;
                                /*
                                                La position du source est dividue par la taux
"d"
                                                pour  calculer  la  position  recommendable  au
cible.
                                                La  différence  entre  la  recommendable  et  la
réel,
                                                nous donne la deviation.
                                */
                                /*
                                                REGLAGE:
                                                La  déviation  du  cible  est  dividue  par  la
largueur
                                                du pitch du cible pour calculer les foix
                                                à répéter ce morceau.
                                        */

     Dev= 0.98 * Dev + 0.02 * Dev0;
                                        /*
                                                Ponderation avec les deviations anterieures.
                                        */

     bool Estricte= FALSE;
     Rep1= 1./m.Taux;

     if (m.Caracterisation > 0.0) {
              if (m.Taux <= 1.) {
                fCarac= pow(fCarac, 1./(4.0 + Sqr(m.Caracterisation)));            //  Le  plus  gran
fCarac, le plus retard
                Rep2= pow(1.5 - fCarac, m.Caracterisation) / m.Taux;
              }
              else {
                fCarac= pow(fCarac, 1./(3.0 - log10(m.Caracterisation)));                  //      Le
plus gran fCarac, le plus avance
                Rep2= pow(fCarac + 0.5, m.Caracterisation) / m.Taux;
              }
     }
     else
              Rep2= Rep1;

     if ((Tail_Mel > m.MAXREPSIZE) || (Tail_Mel < m.MINREPSIZE)) {
                                // Ce ne peut pas dupliquer ni eliminer
              Rep2= 1;
     }

     Rep3= Pond_Arit(m.Reglage, Dev, Rep2);

     Rep3= Rep3 * m.Retard_Avance;

                        //       Rep= Rep3 + 0.5;
                                        /*
                                                Le 0.5 est pour redondir
                                        */

     if (m.Arrondi == ALEATOIRE) {
              if ((random(32000) / 32000.0) < Abs(Rep3 - (long)Rep3) )
                Rep= Rep3 + 1;
              else
                Rep= Rep3;
                                /*
                                 Une  otre  manière  plus  équitative  de  redondir  avec  des   numéros
aléatoires.
                                 Je utilise srand() au commençament pour obtenir toujours les mêmes
fichiers.
                                */
     }
     else if (m.Arrondi == PERIODIQUE) {
              Rep= Rep3 + ErrArrondi + 0.5;
              ErrArrondi+= (Rep3 - Rep);
     }

/*                      Rep= (1 + m.Reglage * Reg) * Rep2 + 0.5;
                        // Le 0.5 est pour redondir
*/

     if (m.Taux == 1.0)
              Rep= 1;

     if ((Rep < 1) && (m.Taux <= 1)) {
              Rep= 1;                       // Extension: Une foi comme minimum !
```

```
    }
    else if ((Rep > 1) && (m.Taux >= 1)) {
            Rep= 1;                         // Reduction: Une foi comme maximum
    }
    else if (Rep <= 0)
            Rep= 0;
//                  else if (Rep > (m.Exagere / d))
//                          Rep= m.Exagere / d;
                                /*
                                        Pour eviter des répétitions exagerées
                                */

  //                  else      if ((Rep * s) > (PITCHSIZE / 2))
  //                          Rep= MIN(1, PITCHSIZE / (2 * s));
                                /*
                                        Pour eviter des répétitions très longues
                                */
 /*
                    if (Visualiser) {
                            cout.setf(ios::fixed | ios::showpoint);
                            cout.width(6);
                            cout.precision(4);
                            cout << endl;
                    }
                    cout << "\r#" << setw(4) << NumMorceaux;
                    if (Visualiser) {
                            echantillon xa, xb;

                            xa.Lire(ComMorceau, 0);
                            xb.Lire(ComMorceau, MelPos - 1);
                            cout << " Tai:" << Tail_Mel
                                                    << " Beg:" << (double)xa
                                                    << " End:" << (double)xb
                                                    << " Pen:" << Pend_Mel
                                    << endl << "     "
                                                    << " FrM:" << fFreq
                                                    << " PuM:" << fPuis
                                                    << " Fréq.: " << Freq_Mel
                                                    << " Puis.: " << Puis_Mel
                                                    << " Q:" << MelQual
                                    << endl << "     "
                                                    << " F:" << fCarac
                                                    << " RepTeo:" << Rep1
                                                    << " RepCar:" << Rep2
                                                    << " Dev:" << Dev
                                                    << " Rep3:" << Rep3
                                                    << " Répét:" << Rep;
                            if ((NumMorceaux % 6) == 0)
                              Pause();
                    }
    */
    unsigned long NewPos;

    if (m.CompensF > 0.0) {

            if ((Rep > 1) && (Compensat != -1)) {
              Compensation(-1,   m,   SPL2MS,   MelPosDern,   MelPos,   MelPosComp,   ComMorceauDern,
ComMorceau, ComMorceauComp);
              NewPos= MelPosComp;
              Base= ComMorceauComp;
            }
            else if (Compensat == 1) {
              Compensation(1, m, SPL2MS, MelPosDern, MelPos, MelPosComp, ComMorceauDern, ComMorceau,
ComMorceauComp);
              NewPos= MelPosComp;
              Base= ComMorceauComp;
            }
            else {
              NewPos= MelPos;
              Base= ComMorceau;
            }

            if (Rep == 1) {
              MelPosDern= MelPos;
              CopyMemory(ComMorceauDern, ComMorceau, MelPos * sizeof(sample));
              if (Compensat == 2)
                      Compensat= 1;
              else
                      Compensat= 0;
            }
            else if (Rep == 0) {
              if (Compensat == 1) {
                      MelPosDern= MelPosComp;
                      CopyMemory(ComMorceauDern, ComMorceau, MelPosComp * sizeof(sample));
              }
              else {
                      MelPosDern= MelPos;
                      CopyMemory(ComMorceauDern, ComMorceau, MelPos * sizeof(sample));
              }
              Compensat= 1;
            }
            else if (Rep > 1) {
              MelPosDern= MelPos;
              CopyMemory(ComMorceauDern, ComMorceau, MelPos * sizeof(sample));
              Compensat= 2;
            }
    }
    else {
            NewPos= MelPos;
            Base= ComMorceau;
    }

    unsigned long s= NewPos;

    if (Rep > 1) {
            Rep--;
            while (Rep--) {
              s= Min(s, LongCible - iCible);
              for (long l= 0; l < s; l++) {
```

```
                        Output->Set(iCible + l, Base[l]);
                }
                iCible+= s;
              }

              // Le dernier morceau est toujours l'original
              s= MelPos;
              s= Min(s, LongCible - iCible);
              for (long l= 0; l < s; l++) {
                Output->Set(iCible + l, ComMorceau[l]);
              }
              iCible+= s;

              Base= ComMorceau;
     }
     else if (Rep == 1) {
              s= Min(s, LongCible - iCible);
              for (long l= 0; l < s; l++) {
                Output->Set(iCible + l, Base[l]);
              }
              iCible+= s;
     }

     LongMorceau= s;

     Dev0= ((iSource / m.Taux) - iCible) /(m.PITCHSIZE * MS2SPL) ;
                                  /*
                                                  Recalculer la deviation maintenant.
                                  */
     if (Dev0 > 0)
              Retard+= Dev0;
     else if (Dev0 < 0)
              Avance+= -Dev0;

     if ((LongCible - iCible) <= 0)
              break;
  }

  /*
              cout << endl << "Operation réalisée. Presse une touche...";
              Pause();

              if (iSource < LongSource) {
                cerr << endl << endl << "Le fichier source n'a pas été examiné complètement" << endl;
                cout << endl << "Le fichier source a été examiné jusqu'à (" << iSource
                                  << " échantillons, " << (iSource * SPL2MS) << " ms) "
                                  << "  quand  il  devoit  avoir  été  examiné  jusqu'u  'ideél  ("  <<
LongSource
                                  << " échantillons, " << (LongSource * SPL2MS) << " ms) "
                                  <<  "  Cette  différence  ("  <<  (LongSource  -  iSource)  <<  "
échantillons, " << ((LongSource - iSource) * SPL2MS) << " ms) ne sera pas examinée." << endl;
                }
              else if (iSource > LongSource)
                cerr << "Le fichier source a été examiné au-delà de l'EOF";
  */

  if (iCible < LongCible) {
     unsigned long s= LongMorceau;
     /*
                cout << endl << "Le fichier résultant (" << iCible
                                  << " échantillons, " << (iCible * SPL2MS) << " ms) "
                                  << " est plus petit que l'ideél (" << LongCible
                                  << " échantillons, " << (LongCible * SPL2MS) << " ms) "
                                  << " Cette différence (" << (LongCible - iCible) << " échantillons,
" << (LongCible - iCible) * SPL2MS << " ms) s'omplira répétant le dernière morceau (" << s << "
échantillons, " << (s * SPL2MS) << " ms)." << endl;
     */
     while (iCible < LongCible) {
              s= Min(s, LongCible - iCible);
              for (long l= 0; l < s; l++) {
//              Output->Set(iCible + l, Base[l]);
                Output->Set(iCible + l, 0.0);                          //   Meilleur   d'omplir   de
silences
              }
              iCible+= s;
     }
  }
  else if (iCible > LongCible) {
     ;
     //         cerr << "Le fichier résultant est plus grand que l'ideél" << endl;
  }

  delete ComMorceauDern;
  delete ComMorceau;
  delete ComMorceauComp;

/*

              cout << endl << "Taille moyenne des tranches: " << LongSource * SPL2MS / NumMorceaux;
              cout << endl << "Total des avances: " << Avance
                        << endl << "Total des retards: " << Retard << endl << endl;

              cout << endl << "Fréquence moyenne du signal: " << FreqMoyenne
                        << endl << "Puissance moyenne du signal: " << PuisMoyenne << endl << endl;
*/
};
```

G.9. « SAMPLE.H » et « SAMPLE.CPP »

Classe fondamentale pour gérer les échantillons en mémoire, divers effets et la comparaison de sons :

```
/*********************************************************
  MODULE : SAMPLE.H
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Declaration of classe sample_array, playable_array and many other things
*********************************************************/


#ifndef _SAMPLE_H_
#define _SAMPLE_H_

#include "utildef.h"
#include <complex.h>
#include <owl/owldefs.h>
#include "spectre.h"
#include "cost.h"

#include "polynom.h"

struct DCTst {
  int N;
  float quality;
  int BitOffset;
  double Freq;
};

class method {
public:
  int Type;
  int BitsComp;
  double DesiredBPS;
  double CompFact;
  double DistRate;

  long Freq;

  union {
     double MeanDX, MeanDY;

     DCTst DCTSt;
  };

  cost Cost;

  method() {
     Type= 0;
     CompFact= 1.;
     BitsComp= 4;
     DistRate= 1.;
  }
};

#define COMP_ADM                1
#define COMP_ADPCM      2
#define COMP_FFT1       3
#define COMP_ADxyM      4
#define COMP_DCT                5
#define COMP_Silence 6
#define COMP_LogPCM     7
#define COMP_ULAW       8
#define COMP_ALAW               9
#define COMP_FFT2       10
#define COMP_NEW                11

#define RATE                    100
#define RATE_FFT1       101

class sample {
  double Value;
public:
  sample() {
     Value= 0;
  }

  sample(double f) {
     Value= f;
  }

  sample(signed short int i) {
     Value= double(i) / 0x8000U;
  }

  sample(unsigned char c) {
     Value= (double(c) - 0x80U) / 0x80U;
  }

  operator double () {
     return Value;
  }

  operator unsigned char () {
     return (unsigned char)Round((Value * 0x80U) + 0x80U);
  }

  operator signed short int () {
     return (signed short int)Round(Value * 0x8000U);
  }
```

```
};
```

```
class playable_array;
class gum_array;

class sample_array {
public:
  int BitsPerSample;
  int BytesPerSample;
  unsigned long NumSamples;
  void huge * Data;
private:
  HGLOBAL Buffer;
public:
  static int GetMaxBitsPerSample();
  sample_array(const sample_array & s, int b, unsigned long n);
  sample_array(const sample_array & s, unsigned long n);
  sample_array(const gum_array & g, int b, unsigned long n, method &m);
  sample_array(int b, unsigned long n);
  void Clear();
  ~sample_array();
  void Load(playable_array *p, double Left, double Right);
  void Set(unsigned long Index, sample s);
  sample Get(unsigned long Index) const;
  sample operator [] (unsigned long Index) const { return Get(Index); }

  void Selection(double Left, double Right, long & Beg, long & End) const;

  void Blank(double Left= 0.0, double Right= 0.0);
  void Invert(double Left, double Right);
  void Formula(filter & F, double Left, double Right);

  friend double Distance(const sample_array &, const sample_array &);

  void MyRate(method & m);
};

double Distance(const sample_array &, const sample_array &);

class spectral_array {
public:
  complex huge *Data;
  unsigned long Length;
private:
  HGLOBAL Buffer;
public:
  long WinSize;
  long TotalBlocks;
  transform_type Transf;
  int Bits;

  complex Get(unsigned long block, unsigned long i) const;
  sample GetAmp(unsigned long block, unsigned long i) const;
  sample GetPhase(unsigned long block, unsigned long i) const;
  friend double Distance(const spectral_array &, const spectral_array &);

  spectral_array(sample_array & s, unsigned long SampleRate,
                                                    double   win_ms,   transform_type   Transf,   BOOL
smooth_flag, double overlap);

  ~spectral_array();
};

double Distance(const spectral_array &, const spectral_array &);

class playable_array {
public:
  void huge * Data;
  unsigned long Length;
private:
  HGLOBAL Buffer;
public:
  playable_array(sample_array & s, double Left, double Right);
  ~playable_array();
};

class gum_array {
public:
  void huge * Data;
  unsigned long Length;
private:
  HGLOBAL Buffer;
public:
  gum_array(sample_array & s, method &m);
  ~gum_array();
};


#endif
```

```
/*********************************************************
  MODULE : SAMPLE.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Implementation of the classes sample_array,
                                          playable_array and many other things
*********************************************************/

#include "sample.h"
#include "polynom.h"

#include "adm.h"
#include "adpcm.h"
#include "fft.h"
#include "fft2.h"
#include "dct.h"
```

```
#include "fftcomp.h"
#include "silence.h"
#include "logpcm.h"
#include "rate.h"
#include "u_a_law.h"
#include "fftcomp.h"
#include "newcoder.h"

#include "spectre.h"

#include "utildef.h"
#include <alloc.h>

int sample_array::GetMaxBitsPerSample() {
  return 16;
}

void sample_array::MyRate(method & m) {
  methode me;
  me.Input();
  me.Taux= m.DistRate;

  long FinalLength= NumSamples / m.DistRate;
  sample_array *output= new sample_array(BitsPerSample, FinalLength);

  DoRate(this, output, me, m.Freq, m.Cost);

  Clear();             // Destructs this
  NumSamples= output->NumSamples;
  Buffer= output->Buffer;
  Data= output->Data;

  output->Data= 0;
  output->Buffer= 0;
  output->NumSamples= 0;
  delete output;
}

sample_array::sample_array(const sample_array & s, unsigned long n)               // To interpolate
: BitsPerSample(s.BitsPerSample), NumSamples(n), BytesPerSample(s.BytesPerSample) {
     if (NumSamples) {
               Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
               Data= GlobalLock(Buffer);
               if (!Data) {
                 NumSamples= 0L; // Error!
               }
               else {
                 unsigned long l, ll;
                 if (NumSamples >= s.NumSamples) {              // Repetition .
                                  //        IMPROVE FOR INTERPOLATION. First order, second order, etc.
                           for (l= 0; l < NumSamples; l++) {
                                    ll=  l * double(s.NumSamples) / double(NumSamples);
                                    sample Rep= s.Get(ll);
                                    Set(l, Rep);
                           }
                 }
                 else {
    // Selection
                                    // Not try average. It's worse.
                           ll= 0;
                           sample Sel= s.Get(0);
                           for (l= 0; l < s.NumSamples; l++) {
                                    long lll= l * double(NumSamples) / double(s.NumSamples);
                                    if (lll != ll) {
                                      Set(ll, Sel);
                                      ll= lll;
                                      Sel= s.Get(l);
                                    }
                           }
                 }
               }
     }
     else
               Data= 0;
}

sample_array::sample_array(const sample_array & s, int b, unsigned long n)
  : BitsPerSample(b), NumSamples(n), BytesPerSample((b - 1) / 8 + 1) {
     if (NumSamples) {
               Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
               Data= GlobalLock(Buffer);
               if (!Data) {
                 NumSamples= 0L; // Error!
               }
               else {
                 unsigned long End;
                 if (NumSamples > s.NumSamples) {
                           Blank();
                           End= s.NumSamples;
                 }
                 else
                           End= NumSamples;
                 for (unsigned long l= 0; l < End; l++) {
                           Set(l, s.Get(l));
                 }
               }
     }
     else
               Data= 0;
  }

sample_array::sample_array(int b, unsigned long n)
  : BitsPerSample(b), NumSamples(n), BytesPerSample((b - 1) / 8 + 1) {
     if (NumSamples) {
               Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
               Data= GlobalLock(Buffer);
               if (!Data) {
                 NumSamples= 0L; // Error!
               }
```

```
    }
    else
            Data= 0;
  }

void sample_array::Selection(double Left, double Right, long &Beg, long &End) const {
  if ((Left < Right) && (Left >= 0.0) && (Right <= 1.0)) {
    Beg= Left * NumSamples;
    End= Right * NumSamples;
  }
  else {
    Beg= 0L;
    End= NumSamples;
  }
}

void sample_array::Clear() { // double Left, double Right) {
    if (Data) {
            GlobalUnlock(Buffer);
            GlobalFree(Buffer);
            Data= 0;
            NumSamples= 0;
    }
}

void sample_array::Formula(filter &F, double Left, double Right) {
  long Beg, End;
  Selection(Left, Right, Beg, End);

  double *u= new double[F.Num.Order];
  double *y= new double[F.Den.Order];
  int i;
  for(i= 0; i < F.Num.Order; i++)
    u[i]= 0.0;
  for(i= 0; i < F.Den.Order; i++)
    y[i]= 0.0;


  for (unsigned long l= Beg; l < End; l++) {
    for(i= F.Den.Order - 1; i > 0; i--) {
            y[i]= y[i - 1];
    }
    for(i= F.Num.Order - 1; i > 0; i--) {
            u[i]= u[i - 1];
    }
    u[0]= Get(l);

    double y0= 0;
    for(i= 0; i < F.Num.Order; i++) {
            y0+= F.Num[i]*u[i];
    }

    for(i= 1; i < F.Den.Order; i++) {
            y0-= F.Den[i]*y[i];
    }

    y0/= F.Den[0];

    Set(l, y0);
    y[0]= y0;
  }

  delete u;
  delete y;
}

sample_array::~sample_array() {
    Clear();
  }

void sample_array::Set(unsigned long Index, sample s) {
    if (BytesPerSample == 2) {
            signed short v= s;
            signed short Mask= 0xFFFF;
            int b= BitsPerSample % 8;
            if (b)
              Mask <<= (8 - b);
            v &= Mask;
            *((signed short huge *)Data + Index)= v;
    }
    else if (BytesPerSample == 1) {
            unsigned char v= s;
            unsigned char Mask= 0xFF;
            int b= BitsPerSample % 8;
            if (b)
              Mask <<= (8 - b);
            v &= Mask;
            *((unsigned char huge *)Data + Index)= v;
    }
    else {              // Not implemented
            for(int i= 0; i < BytesPerSample; i++) {
              *((unsigned char huge *)Data + Index * BytesPerSample + i)= 0;
            }
    }
  }

sample sample_array::Get(unsigned long Index) const {
    if (BytesPerSample == 2)
            return sample(*(((signed short huge *)Data) + Index)) ;
    else if (BytesPerSample == 1)
            return sample(*(((unsigned char huge *)Data) + Index));
    else {
     return sample(0.0);                  // Not implemented !
    }
  }
/*
  sample operator [] (unsigned long Index) const {
    return Get(Index);
  }
```

```
*/

void sample_array::Blank(double Left, double Right) {
  long Beg, End;
  Selection(Left, Right, Beg, End);

     if (BytesPerSample == 1) {
              for (unsigned long l= Beg; l < End; l++) {
                *(((unsigned char huge *)Data) + l)= 0x80;
              }
     }
     else {
              for (unsigned long l= Beg * BytesPerSample; l < End * BytesPerSample; l++) {
                *(((unsigned char huge *)Data) + l)= 0;
              }
     }
  }

void sample_array::Invert(double Left, double Right) {
  long Beg, End;
  Selection(Left, Right, Beg, End);
     for(unsigned long l= Beg; l < End / 2; l++) {
              sample Temp= Get(l);
              Set(l, Get(Beg + End - 1 - l));
              Set(Beg + End - 1 - l, Temp);
     }
  }

void sample_array::Load(playable_array *p, double Left, double Right) {
  long Beg, End;
  Selection(Left, Right, Beg, End);

  if (!(BitsPerSample % 8))
     for (unsigned long l= 0; l < Min((long)p->Length, (End - Beg)* (BitsPerSample/8)); l++) {
              *(((unsigned char huge *)Data) + l + Beg)= *(((unsigned char huge *)p->Data) + l);
     }
  else if (BytesPerSample == 1) {
     for (unsigned long l= 0; l < Min((long)p->Length, End - Beg); l++) {
              Set(l + Beg, *(((unsigned char huge *)p->Data) + l));
     }
  }
  else if (BytesPerSample == 2) {
     for (unsigned long l= 0; l < Min((long)p->Length / 2L, End - Beg); l++) {
              Set(l + Beg, *(((short huge *)p->Data) + l));
     }
  }
  else
     ;        // Error
}

double Distance(const sample_array & s1, const sample_array &s2) {
  enum {ITAKURA82, NONSLASTIC1, NONSLASTIC2, SLASTIC, SIMPLE} contraintes= SLASTIC;

  double Dist;
  HGLOBAL DiBuffer= GlobalAlloc(GHND, (s2.NumSamples) * sizeof(float));
  float huge *Di= (float huge *)GlobalLock(DiBuffer);
  HGLOBAL Di1Buffer= GlobalAlloc(GHND, (s2.NumSamples) * sizeof(float));
  float huge *Di1= (float huge *)GlobalLock(Di1Buffer);
  HGLOBAL Di2Buffer= GlobalAlloc(GHND, (s2.NumSamples) * sizeof(float));
  float huge *Di2= (float huge *)GlobalLock(Di2Buffer);

  if (!((Di) && (Di1) && (Di2)))
     return -1.0;

  #define D(i, j) DD[(i)*(s2.TotalBlocks) + (j)]

  long i, j;

  for(i= 0; i < s1.NumSamples; i++) {
     float huge *Temp= Di2;
     Di2= Di1;
     Di1= Di;
     Di= Temp;
     for(j= 0; j < s2.NumSamples; j++) {
              float Dij= Sqr(double(s1.Get(i)) - double(s2.Get(j)));

              if ((i == 0) && (j == 0)) {
                Di[j]= Dij;
              }
              else if (i == 0) {
                Di[j]= Dij + Di[j - 1];
              }
              else if (j == 0) {
                Di[j]= Dij + Di1[j];
              }
              else {
                float a, b, c;

                switch (contraintes) {
                          case ITAKURA82:     a= Di1[j];
    // HAY QUE IMPEDIRLO DOS VECES SEGUIDAS
                                                                                    b= Di1[j - 1];
                                                                                    c= Di1[j - 2];
                                                                                    break;
                          case  NONSLASTIC2:    a=  Di2[j  -  1]  +  2  *  Sqr(double(s1.Get(i-1))  -
double(s2.Get(j)));
                                                                                    b=  Di1[j  -  1]  +
Dij;
                                                                                    c= Di1[j - 2] + 2
* Sqr(double(s1.Get(i)) - double(s2.Get(j-1)));
                                                                                    break;
                          case SLASTIC:                 a= Di1[j];
                                                                                    b=  Di1[j  -  1]  +
Dij;
                                                                                    c= Di[j - 1];
                                                                                    break;
                          case NONSLASTIC1:  a= Di1[j] + Dij;
                                                                                    b= Di1[j - 1];
```

```
                                                                                c=  Di[j  -  1]  +
Dij;
                                                                                break;
                        case SIMPLE:                    a= Di1[j];
                                                                                b= Di1[j - 1];
                                                                                c= Di[j - 1];
                                                                                break;
                }

                Di[j]= Dij + Min(a, b, c);
              }
    }
  }

  Dist= Di[s2.NumSamples - 1] / Min(float(s1.NumSamples), float(s2.NumSamples));

  GlobalUnlock(DiBuffer);
  GlobalFree(DiBuffer);
  GlobalUnlock(Di1Buffer);
  GlobalFree(Di1Buffer);
  GlobalUnlock(Di2Buffer);
  GlobalFree(Di2Buffer);

  return Dist;
}




spectral_array::~spectral_array() {
  if (Data) {
     GlobalUnlock(Buffer);
     GlobalFree(Buffer);
  }
}

spectral_array::spectral_array(sample_array & s, unsigned long SampleRate,
                                                  double    win_ms,    transform_type    t,    BOOL
smooth_flag, double overlap) {
  Transf= t;
  Data= 0;
  Spectre_Param(s.NumSamples, SampleRate, win_ms, &WinSize, &TotalBlocks, &Bits, overlap);

  Length= 2L * TotalBlocks * WinSize * sizeof *Data;
  Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
  Data = (complex huge *)GlobalLock(Buffer);

  if (Data == 0)
     return;

  Spectre(&s, Data, WinSize, TotalBlocks, Bits, Transf, smooth_flag, overlap);
}


complex spectral_array::Get(unsigned long block, unsigned long i) const {
  return Data[block * WinSize + i];
}

sample spectral_array::GetAmp(unsigned long block, unsigned long i) const {
  return amp(i, Data + block * WinSize, Bits, Transf == FFT1);

  // UTILIZAR lnorm
}

sample spectral_array::GetPhase(unsigned long block, unsigned long i) const {
  return phase(i, Data + block * WinSize, Bits, Transf == FFT1);

  // UTILIZAR ang
}


double FreqDist(complex huge *s1, complex huge *s2, long n) {
  double Ret= 0.0;
  for (long i= 0; i < n; i++) {
     // Ret+= norm(s1[i] - s2[i]);                    //       (a1 - b1)2 + (a2 - b2)2
     // Ret+= sqrt(norm(s1[i] - s2[i]));             // square rooted
     Ret+= Sqr(sqrt(norm(s1[i])) - sqrt(norm(s2[i])));
  }
  return sqrt(Ret) / n;
}

double Distance(const spectral_array & s1, const spectral_array &s2) {
  enum {ITAKURA82, NONSLASTIC1, NONSLASTIC2, SLASTIC, SIMPLE} contraintes= SLASTIC;

  double Dist;
  HGLOBAL DiBuffer= GlobalAlloc(GHND, (s2.TotalBlocks) * sizeof(float));
  float huge *Di= (float huge *)GlobalLock(DiBuffer);
  HGLOBAL Di1Buffer= GlobalAlloc(GHND, (s2.TotalBlocks) * sizeof(float));
  float huge *Di1= (float huge *)GlobalLock(Di1Buffer);
  HGLOBAL Di2Buffer= GlobalAlloc(GHND, (s2.TotalBlocks) * sizeof(float));
  float huge *Di2= (float huge *)GlobalLock(Di2Buffer);

  if (!((Di) && (Di1) && (Di2)))
     return -1.0;

  #define D(i, j) DD[(i)*(s2.TotalBlocks) + (j)]

  long nyquist1, nyquist2;
  nyquist1= s1.WinSize / 2.;
  nyquist2= s2.WinSize / 2.;
  long i, j;

  for(i= 0; i < s1.TotalBlocks; i++) {
     float huge *Temp= Di2;
     Di2= Di1;
     Di1= Di;
     Di= Temp;
```

```
     for(j= 0; j < s2.TotalBlocks; j++) {
               float Dij= FreqDist(s1.Data + i * s1.WinSize, s2.Data + j * s2.WinSize, Min(nyquist1,
nyquist2));

               if ((i == 0) && (j == 0)) {
                 Di[j]= Dij;
               }
               else if (i == 0) {
                 Di[j]= Dij + Di[j - 1];
               }
               else if (j == 0) {
                 Di[j]= Dij + Di1[j];
               }
               else {
                 float a, b, c;

                 switch (contraintes) {
                          case ITAKURA82:      a= Di1[j];
    // HAY QUE IMPEDIRLO DOS VECES SEGUIDAS
                                                                                b= Di1[j - 1];
                                                                                c= Di1[j - 2];
                                                                                break;
                          case NONSLASTIC2:  a= Di2[j - 1] + 2 * FreqDist(s1.Data + (i-1) * s1.WinSize,
s2.Data + j * s2.WinSize, Min(nyquist1, nyquist2));
                                                                                b=  Di1[j  -  1]  +
Dij;
                                                                                c= Di1[j - 2] + 2
* FreqDist(s1.Data + i * s1.WinSize, s2.Data + (j-1) * s2.WinSize, Min(nyquist1, nyquist2));
                                                                                break;
                          case SLASTIC:                 a= Di1[j];
                                                                                b=  Di1[j  -  1]  +
Dij;
                                                                                c= Di[j - 1];
                                                                                break;
                          case NONSLASTIC1:  a= Di1[j] + Dij;
                                                                                b= Di1[j - 1];
                                                                                c=  Di[j  -  1]  +
Dij;
                                                                                break;
                          case SIMPLE:                  a= Di1[j];
                                                                                b= Di1[j - 1];
                                                                                c= Di[j - 1];
                                                                                break;
                 }

                 Di[j]= Dij + Min(a, b, c);
               }
     }
   }

   Dist= Di[s2.TotalBlocks - 1] / Min(float(s1.TotalBlocks), float(s2.TotalBlocks));

   GlobalUnlock(DiBuffer);
   GlobalFree(DiBuffer);
   GlobalUnlock(Di1Buffer);
   GlobalFree(Di1Buffer);
   GlobalUnlock(Di2Buffer);
   GlobalFree(Di2Buffer);

   return Dist;
}

playable_array::playable_array(sample_array & s, double Left, double Right) {
   long Beg, End;
   s.Selection(Left, Right, Beg, End);
     if (s.NumSamples) {
               Length= (End - Beg) * s.BytesPerSample;
               Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
               Data= GlobalLock(Buffer);
               if (!Data)
                 Length= 0; // Error!
               else {
                 for (unsigned long l= 0L; l < (End - Beg)* s.BytesPerSample; l++) {
                          *(((unsigned char huge *)Data) + l)= *(((unsigned char huge *)s.Data) + l +
Beg * s.BytesPerSample);
                 }
               }
     }
     else {
               Data= 0;
               Length= 0L;
     }
   }

playable_array::~playable_array() {
     if (Data) {
               GlobalUnlock(Buffer);
               GlobalFree(Buffer);
     }
   }


gum_array::~gum_array() {
     if (Data) {
               GlobalUnlock(Buffer);
               GlobalFree(Buffer);
     }
   }



#define ADPCMBLOCKSIZE           16384
#define ADMBLOCKSIZE             16384
#define SILENCEBLOCKSIZE         16384
#define LogPCMBLOCKSIZE          16384

gum_array::gum_array(sample_array & s, method &m) {
   if (s.NumSamples) {
```

```
    if (m.Type == COMP_ADPCM) {
            Length= (s.NumSamples + 1) / 2;                             // ADPCM 16 to 4
   // CAMBIAR
            long End= s.NumSamples;
            Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
            Data= GlobalLock(Buffer);
            if (!Data)
              Length= 0L; // Error!
            else {
              BYTE huge *D= (BYTE huge *)Data;
              adpcm_state St= {0, 0};
              St.valprev= 0;
              for (long c= 0; c < End; c+= ADPCMBLOCKSIZE) {
                      int l= (int)Min((unsigned long)(End - c), (unsigned long)ADPCMBLOCKSIZE);
                      adpcm_coder(&s, c, D, l, &St, m.BitsComp, m.Cost);
                      D+= l / 2;
              }
              return;
            }
    }
    else if (m.Type == COMP_NEW) {
            Length= s.NumSamples ;
            long End= s.NumSamples;
            Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
            Data= GlobalLock(Buffer);
            if (!Data)
              Length= 0L; // Error!
            else {
              BYTE huge *D= (BYTE huge *)Data;
              new_state St= {0, 0};
              St.valprev= 0;
              for (long c= 0; c < End; c+= ADPCMBLOCKSIZE) {
                      int l= (int)Min((unsigned long)(End - c), (unsigned long)ADPCMBLOCKSIZE);
                      new_coder(&s, c, D, l, &St, m.BitsComp, m.Cost);
                      D+= l / 2;
              }
              return;
            }
    }
    else if (m.Type == COMP_ULAW) {
            Length= s.NumSamples;
            Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
            Data= GlobalLock(Buffer);
            for (long c= 0; c < s.NumSamples; c++) {
              ((unsigned char huge *)Data)[c]= linear2ulaw((signed short int)s.Get(c));
            }
    }
    else if (m.Type == COMP_ALAW) {
            Length= s.NumSamples;
            Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
            Data= GlobalLock(Buffer);
            for (long c= 0; c < s.NumSamples; c++) {
              ((unsigned char huge *)Data)[c]= linear2Alaw((signed short int)s.Get(c));
            }
    }
    else if (m.Type == COMP_LogPCM) {
            Length= (s.NumSamples * m.BitsComp + 1) / 8;
            long End= s.NumSamples;
            Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
            Data= GlobalLock(Buffer);
            if (!Data)
              Length= 0L; // Error!
            else {
              BYTE huge *D= (BYTE huge *)Data;
              InitLogPCMCompress(m.BitsComp);
              for (long c= 0; c < End; c+= LogPCMBLOCKSIZE) {
                      int l= (int)Min((unsigned long)(End - c), (unsigned long)LogPCMBLOCKSIZE);
                      LogPCMCompress(&s, c, D, l, m.BitsComp, m.Cost);
              }
              return;
            }
    }
    else if (m.Type == COMP_DCT) {
            DCTst St= m.DCTSt;
            St.BitOffset= 0;
            /*
              if ( quality < 0 || quality > 50 )
              printf( "Illegal quality factor of %d\n", quality );
              printf( "Using quality factor of %d\n", quality );
            */
            //                  cout << "\n\rSize of matrix N x N: (1..100) (15x15 = 225 samples
recommended) ? ";

            long ApproxLength= s.NumSamples;                                    //              EASY
APROXIMATION
            long End= s.NumSamples;
            HGLOBAL ApproxBuffer = GlobalAlloc(GMEM_MOVEABLE, ApproxLength);
            void FAR *ApproxData= GlobalLock(ApproxBuffer);
            if (!ApproxData) {
              Length= 0L; // Error!
              Data= 0;
            }
            else {
              BYTE huge *D= (BYTE huge *)ApproxData;
              Length= 0L;
              Initialize_DCT(St.quality, St.N);
              for (long c= 0; c < End; c+= St.N*St.N) {
                      DCTCompress(&s, c, D, St, m.Cost);
              }
              DCTEndCompress(D, St.BitOffset);
              Terminate_DCT();
              Length= D - (BYTE huge *)ApproxData + 1;
              if (!St.BitOffset)
                      Length--;
//            Length+=1000;     // WHY?
              Buffer= GlobalAlloc(GMEM_MOVEABLE, Length);
              Data= GlobalLock(Buffer);
              if (!Data) {
                      Length= 0L; // Error!
```

```
                }
                else {
                        CopyMemory(Data, ApproxData, Length);
                }
                GlobalUnlock(ApproxBuffer);
                GlobalFree(ApproxBuffer);
                m.CompFact= double(c) / double(Length);
                return;
              }
     }
     else if (m.Type == COMP_FFT2) {
              DCTst St= m.DCTSt;
              St.Freq= m.Freq;
              St.BitOffset= 0;
              /*
                if ( quality < 0 || quality > 50 )
                printf( "Illegal quality factor of %d\n", quality );
                printf( "Using quality factor of %d\n", quality );
              */
              //                  cout << "\n\rSize of matrix N x N: (1..100) (15x15 = 225 samples
recommended) ? ";

              long ApproxLength= s.NumSamples;                                    //              EASY
APROXIMATION
              long End= s.NumSamples;
              HGLOBAL ApproxBuffer = GlobalAlloc(GMEM_MOVEABLE, ApproxLength);
              void FAR *ApproxData= GlobalLock(ApproxBuffer);
              if (!ApproxData) {
                Length= 0L; // Error!
                Data= 0;
              }
              else {
                BYTE huge *D= (BYTE huge *)ApproxData;
                Length= 0L;
                Initialize_FFT(St.quality, St.N);
                for (long c= 0; c < End; c+= 1 << St.N ) {
                        FFTCompress(&s, c, D, St, m.Cost);
                }
                FFTEndCompress(D, St.BitOffset);
                Terminate_FFT();
                Length= D - (BYTE huge *)ApproxData + 1;
                if (!St.BitOffset)
                        Length--;
//              Length+=1000;     // WHY?
                Buffer= GlobalAlloc(GMEM_MOVEABLE, Length);
                Data= GlobalLock(Buffer);
                if (!Data) {
                        Length= 0L; // Error!
                }
                else {
                        CopyMemory(Data, ApproxData, Length);
                }
                GlobalUnlock(ApproxBuffer);
                GlobalFree(ApproxBuffer);
                m.CompFact= double(c) / double(Length);
                return;
              }
     }
     else if (m.Type == COMP_Silence) {
              long ApproxLength= s.NumSamples * s.BytesPerSample;
    // EASY APROXIMATION
              long End= s.NumSamples;
              HGLOBAL ApproxBuffer = GlobalAlloc(GMEM_MOVEABLE, ApproxLength);
              void FAR *ApproxData= GlobalLock(ApproxBuffer);
              if (!ApproxData) {
                Length= 0L; // Error!
                Data= 0;
              }
              else {
                BYTE huge *D= (BYTE huge *)ApproxData;
                Length= 0L;
                for (long c= 0; c < End; c+= SILENCEBLOCKSIZE) {
                        int l= (int)Min((unsigned long)(End - c), (unsigned long)SILENCEBLOCKSIZE);
                        SilenceCompress(&s, c, D, l, m.Cost);
                }
                Length= D - (BYTE huge *)ApproxData;
                Buffer= GlobalAlloc(GMEM_MOVEABLE, Length);
                Data= GlobalLock(Buffer);
                if (!Data) {
                        Length= 0L; // Error!
                }
                else {
                        CopyMemory(Data, ApproxData, Length);
                        m.CompFact= (double(End) * s.BytesPerSample) / double(Length);
                }
                GlobalUnlock(ApproxBuffer);
                GlobalFree(ApproxBuffer);



//              Length+= 1000;              // MURPHY




                return;
              }
     }
     else if (m.Type == COMP_ADM) {
              Length= (s.NumSamples + 7) / 8;
              long End= s.NumSamples;
              Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
              Data= GlobalLock(Buffer);
              if (!Data)
                Length= 0L; // Error!
              else {
                BYTE huge *D= (BYTE huge *)Data;
```

```
                ADMSt St;
                long RealBlockLen;
                long RealLength= 0L;
                for (long c= 0; c < End; c+= ADMBLOCKSIZE) {
                        int l= (int)Min((unsigned long)(End - c), (unsigned long)ADMBLOCKSIZE);
                        ADMCompress(&s, c, D, l, &St, &RealBlockLen);
                        D+= RealBlockLen; // += l / 8;
                        RealLength+= RealBlockLen;
                }
                if (RealLength != Length) {
                        GlobalUnlock(Buffer);
                        GlobalFree(Buffer);
                        Data= 0;
                        Length= 0L;                 // Error;
                }
                return;
              }
     }
     else if (m.Type == COMP_ADxyM) {
              long ApproxLength= s.NumSamples;                                    //             EASY
APROXIMATION
              long End= s.NumSamples;
              HGLOBAL ApproxBuffer = GlobalAlloc(GMEM_MOVEABLE, ApproxLength);
              void FAR *ApproxData= GlobalLock(ApproxBuffer);
              if (!ApproxData) {
                Length= 0L; // Error!
                Data= 0;
              }
              else {
                BYTE huge *D= (BYTE huge *)ApproxData;
                ADxyMSt St;
                long RealBlockLen;
                Length= 0L;
                for (long c= 0; c < End; c+= ADMBLOCKSIZE) {
                        int l= (int)Min((unsigned long)(End - c), (unsigned long)ADMBLOCKSIZE);
                        ADxyMCompress(&s, c, D, l, &St, &RealBlockLen, m.Freq, m.Cost, m.DesiredBPS);
                        D+= RealBlockLen;
                        Length+= RealBlockLen;
                }
                if (St.Pos)
                        Length++;
                Buffer= GlobalAlloc(GMEM_MOVEABLE, Length);
                Data= GlobalLock(Buffer);
                if (!Data) {
                        Length= 0L; // Error!
                }
                else {
                        CopyMemory(Data, ApproxData, Length);
                }
                GlobalUnlock(ApproxBuffer);
                GlobalFree(ApproxBuffer);

                m.MeanDX= St.Meandx / s.NumSamples;
                m.MeanDY= St.Meandy / s.NumSamples;
                m.CompFact= s.NumSamples * s.BytesPerSample / double(Length);

                return;
              }
     }
     else if (m.Type == RATE_FFT1) {
              Length= s.NumSamples * sizeof(complex);
              Buffer = GlobalAlloc(GMEM_MOVEABLE, Length);
              Data= GlobalLock(Buffer);
              if (!Data)
                Length= 0L; // Error!
              else {
                double win_ms= 5;
                long win_size;    /* FFT window size */
                int bits;                   /* bits^2 = win_size */
                complex huge *wave;                 /* Wave samples/spectrum */
                complex huge *W;
                long got;




                double R= 11025.0;




                long rwinsize= win_ms * R / 1000.0;
                win_size= 1;
                while (rwinsize >>= 1) {
                        win_size <<= 1;
                }
                if ((rwinsize - win_size) < ((win_size << 1) - rwinsize))
                        win_size <<= 1;

                bits = 0;
                for (int i = int(win_size >> 1); i != 0; i >>= 1)
                        bits++;

                win_size= 1 << bits;

                win_ms= win_size * 1000.0 / R;

                HGLOBAL waveBuffer = GlobalAlloc(GMEM_MOVEABLE, sizeof(complex) * win_size);
                wave= (complex huge *)GlobalLock(waveBuffer);
                HGLOBAL WBuffer = GlobalAlloc(GMEM_MOVEABLE, sizeof(complex) * win_size);
                W = (complex huge *)GlobalLock(WBuffer);

                fft2_init(bits, DIRECT, W);

                for (long block = 0;  ;block++) {
                        long OffSet= block * win_size;
                        got = Min(win_size, (long)s.NumSamples - OffSet);
                        if (got <= 0)
                                break;                      /* done if no more samples */
```

```
                        for (long i = 0; i < got; i++) {
                                wave[i] = complex(s.Get(OffSet + i), 0.0);
                        }
                        for (i = got; i < win_size; i++) {
                                wave[i] = complex(s.Get(OffSet + got - 1), 0.0);
                        }
//              CopyMemory (old_wave, wave, sizeof(complex) * win_size);

//              scale (wave, bits);
                        double ToHz= 1;

                        fft2(bits, DIRECT, W, wave, ToHz);

                        CopyMemory(((complex huge *)Data) + OffSet, wave, got * sizeof *wave);
                }
                GlobalUnlock(WBuffer);
                GlobalFree(WBuffer);
                GlobalUnlock(waveBuffer);
                GlobalFree(waveBuffer);
                return;
              }
    }
  }
  Data= 0;
  Length= 0L;
}

sample_array::sample_array(const gum_array & g, int b, unsigned long n, method &m)
  : BitsPerSample(b), NumSamples(n), BytesPerSample((b - 1) / 8 + 1) {
  if (NumSamples) {
    if (m.Type == COMP_ADPCM) {
              Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
              Data= GlobalLock(Buffer);
              if (!Data) {
                NumSamples= 0L; // Error!
              }
              else {
                unsigned long End;
                if (NumSamples > g.Length * 2) {
                        Blank();
                        End= g.Length * 2;
                }
                else
                        End= NumSamples;
                BYTE huge *D= (BYTE huge *)g.Data;
                adpcm_state St= {0, 0};
                for (long c= 0; c < End; c+= ADPCMBLOCKSIZE) {
                        int l= (int)Min((unsigned long)(End - c), (unsigned long)ADPCMBLOCKSIZE);
                        adpcm_decoder(D, this, c, l, &St, m.BitsComp, m.Cost);
                        D+= l / 2;
                }
                return;
              }
    }
    else  if (m.Type == COMP_NEW) {
              Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
              Data= GlobalLock(Buffer);
              if (!Data) {
                NumSamples= 0L; // Error!
              }
              else {
                unsigned long End;
                if (NumSamples > g.Length * 2) {
                        Blank();
                        End= g.Length * 2;
                }
                else
                        End= NumSamples;
                BYTE huge *D= (BYTE huge *)g.Data;
                new_state St= {0, 0};
                for (long c= 0; c < End; c+= ADPCMBLOCKSIZE) {
                        int l= (int)Min((unsigned long)(End - c), (unsigned long)ADPCMBLOCKSIZE);
                        new_decoder(D, this, c, l, &St, m.BitsComp, m.Cost);
                        D+= l / 2;
                }
                return;
              }
    }
    else if (m.Type == COMP_ULAW) {
              Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
              Data= GlobalLock(Buffer);
              if (!Data) {
                NumSamples= 0L; // Error!
              }
              else {
                unsigned long End;
                BYTE huge *D= (BYTE huge *)g.Data;
                for (long c= 0; c < NumSamples; c++) {
                        Set(c, (signed short int)ulaw2linear(D[c]));
                }
              }
    }
    else if (m.Type == COMP_ALAW) {
              Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
              Data= GlobalLock(Buffer);
              if (!Data) {
                NumSamples= 0L; // Error!
              }
              else {
                unsigned long End;
                BYTE huge *D= (BYTE huge *)g.Data;
                for (long c= 0; c < NumSamples; c++) {
                        Set(c, (signed short int)Alaw2linear(D[c]));
                }
              }
    }
    else if (m.Type == COMP_LogPCM) {
              Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
```

```
            Data= GlobalLock(Buffer);
            if (!Data) {
              NumSamples= 0L; // Error!
            }
            else {
              unsigned long End;
              End= NumSamples;
              BYTE huge *D= (BYTE huge *)g.Data;
              InitLogPCMDecompress(m.BitsComp);
              for (long c= 0; c < End; c+= LogPCMBLOCKSIZE) {
                      int l= (int)Min((unsigned long)(End - c), (unsigned long)LogPCMBLOCKSIZE);
                      LogPCMDecompress(D, this, c, l, m.BitsComp, m.Cost);
              }
              return;
            }
    }
    else if (m.Type == COMP_ADM) {
            Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
            Data= GlobalLock(Buffer);
            if (!Data) {
              NumSamples= 0L; // Error!
            }
            else {
              unsigned long End;
              if (NumSamples > g.Length * 8) {
                      Blank();
                      End= g.Length * 8;
              }
              else
                      End= NumSamples;
              BYTE huge *D= (BYTE huge *)g.Data;
              ADMSt St;
              long RealBlockLen;
              long RealLength= 0L;
              for (long c= 0; c < End; c+= ADMBLOCKSIZE) {
                      int l= (int)Min((unsigned long)(End - c), (unsigned long)ADMBLOCKSIZE);
                      ADMDecompress(D, this, c, l, &St, &RealBlockLen);
                      RealLength+= RealBlockLen;
                      D+= RealBlockLen;           // += l/8;
              }
//            if (BitOffset)
//                    RealLength++;
              if (RealLength != g.Length) {
                      GlobalUnlock(Buffer);
                      GlobalFree(Buffer);
                      Data= 0;
                      NumSamples= 0;; // Error;
              }
              return;
            }
    }
    else if (m.Type == COMP_DCT) {
            DCTst St= m.DCTSt;
            St.BitOffset= 0;
            Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
            Data= GlobalLock(Buffer);
            if (!Data) {
              NumSamples= 0L; // Error!
            }
            else {
              unsigned long End;
              End= NumSamples;
              BYTE huge *D= (BYTE huge *)g.Data;
              long RealLength= 0L;
              Initialize_DCT(St.quality, St.N);
              for (long c= 0; c < End; c+= St.N*St.N) {
                      if (c > End - 10000)
                              c= c;
                      DCTDecompress(D, this, c, St, m.Cost);
              }
              Terminate_DCT();
              RealLength= D - (BYTE huge *)g.Data + 1;
              if (!St.BitOffset)
                      RealLength--;

              RealLength++;
              if (RealLength != g.Length) {
                      GlobalUnlock(Buffer);
                      GlobalFree(Buffer);
                      Data= 0;
                      NumSamples= 0;; // Error;
              }

              return;
            }
    }
    else if (m.Type == COMP_FFT2) {
            DCTst St= m.DCTSt;
            St.BitOffset= 0;
            Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
            Data= GlobalLock(Buffer);
            if (!Data) {
              NumSamples= 0L; // Error!
            }
            else {
              unsigned long End;
              End= NumSamples;
              BYTE huge *D= (BYTE huge *)g.Data;
              long RealLength= 0L;
              Initialize_FFT(St.quality, St.N);
              for (long c= 0; c < End; c+= 1 << St.N) {
                      if (c > End - 10000)
                              c= c;
                      FFTDecompress(D, this, c, St, m.Cost);
              }
              Terminate_FFT();
              RealLength= D - (BYTE huge *)g.Data + 1;
              if (!St.BitOffset)
                      RealLength--;
```

```
                RealLength++;
                if (RealLength != g.Length) {
                        GlobalUnlock(Buffer);
                        GlobalFree(Buffer);
                        Data= 0;
                        NumSamples= 0;; // Error;
                }

                return;
              }
      }
      else if (m.Type == COMP_Silence) {
              Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
              Data= GlobalLock(Buffer);
              if (!Data) {
                NumSamples= 0L; // Error!
              }
              else {
                unsigned long End;
                End= NumSamples;
                BYTE huge *D= (BYTE huge *)g.Data;
                long RealLength= 0L;
                for (long c= 0; c < End; c+= SILENCEBLOCKSIZE) {
                        int l= (int)Min((unsigned long)(End - c), (unsigned long)SILENCEBLOCKSIZE);
                        SilenceDecompress(D, this, c, l, m.Cost);
                }
                RealLength= D - (BYTE huge *)g.Data;
                if (RealLength != g.Length) {
                        End= End; // Error;
                        /*
                        GlobalUnlock(Buffer);
                        GlobalFree(Buffer);
                        Data= 0;
                        NumSamples= 0;
                        */
                }
                return;
              }
      }
      else if (m.Type == COMP_ADxyM) {
              Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
              Data= GlobalLock(Buffer);
              if (!Data) {
                NumSamples= 0L; // Error!
              }
              else {
                unsigned long End;
/*              if (NumSamples > g.Length * 8) {
                        Blank();
                        End= g.Length * 8;
                }
                else
*/                      End= NumSamples;
                BYTE huge *D= (BYTE huge *)g.Data;
                ADxyMSt St;
                long RealBlockLen;
                long RealLength= 0L;
                for (long c= 0; c < End; c+= ADMBLOCKSIZE) {
                        int l= (int)Min((unsigned long)(End - c), (unsigned long)ADMBLOCKSIZE);
                        ADxyMDecompress(D,   this,   c,   l,   &St,   &RealBlockLen,   m.Freq,   m.Cost,
m.DesiredBPS);
                        RealLength+= RealBlockLen;
                        D+= RealBlockLen;
                }
//              if (BitOffset)
//                      RealLength++;
                if (RealLength != g.Length) {
                        Data= Data;
/*                      GlobalUnlock(Buffer);
                        GlobalFree(Buffer);
                        Data= 0;
                        NumSamples= 0;; // Error;
*/
                }
                return;
              }
      }
      else if (m.Type == RATE_FFT1) {
              long OldNumSamples= NumSamples;
              NumSamples= NumSamples / m.DistRate;
              Buffer = GlobalAlloc(GMEM_MOVEABLE, NumSamples * BytesPerSample);
              Data= GlobalLock(Buffer);
              if (!Data) {
                NumSamples= 0L; // Error!
              }
              else {
                double win_ms= 5;
                long win_size;    /* FFT window size */
                int bits;                 /* bits^2 = win_size */
                complex huge *wave;               /* Wave samples/spectrum */
                complex huge *wavenew;            /* Wave samples/spectrum */
                complex huge *wavepond;           /* Wave samples/spectrum */
                complex huge *iW;
                long got;




                double R= 11025.0;




                long rwinsize= R * win_ms / 1000.0;
                win_size= 1;
                while (rwinsize >>= 1) {
                        win_size <<= 1;
```

```
                }
                if ((rwinsize - win_size) < ((win_size << 1) - rwinsize))
                        win_size <<= 1;

                bits = 0;
                for (int i = int(win_size >> 1); i != 0; i >>= 1)
                        bits++;

                win_size= 1 << bits;

                win_ms= win_size * 1000.0 / R;

                HGLOBAL waveBuffer = GlobalAlloc(GMEM_MOVEABLE, sizeof(complex) * win_size);
                wave= (complex huge *)GlobalLock(waveBuffer);
                HGLOBAL wavepondBuffer = GlobalAlloc(GMEM_MOVEABLE, sizeof(complex) * win_size);
                wavepond= (complex huge *)GlobalLock(wavepondBuffer);
                HGLOBAL wavenewBuffer = GlobalAlloc(GMEM_MOVEABLE, sizeof(complex) * win_size);
                wavenew= (complex huge *)GlobalLock(wavenewBuffer);
                HGLOBAL iWBuffer = GlobalAlloc(GMEM_MOVEABLE, sizeof(complex) * win_size);
                iW = (complex huge *)GlobalLock(iWBuffer);

                fft2_init(bits, INVERSE, iW);

                double ErrArrondi= 0;
                long NewOffSet= 0;
                for (long block = 0;  ;block++) {
                        long OffSet= block * win_size;
                        got = Min(win_size, OldNumSamples - OffSet);

                        if (got <= 0)
                                break;                        /* done if no more samples */

                        CopyMemory(wave, ((complex huge *)g.Data) + OffSet, got * sizeof *wave);

                        double Rep0= 1/m.DistRate;
                        int Rep;
/*
                        if (Arrondi == ALEATOIRE) {
                                if ((random(32000) / 32000.0) < Abs(Rep0 - (long)Rep0) )
                                  Rep= Rep0 + 1;
                                else
                                  Rep= Rep0;
                        }
                        else {
*/
                                Rep= Rep0 + ErrArrondi + 0.5;
                                ErrArrondi+= (Rep0 - Rep);
//                      }

                        // Vitesse variable
                        for(i= 0; i < win_size; i++) {
                                complex c;
                                if (block != 0) {
                                  c= complex(real(wave[i]) + real(wavepond[i]) / 2.0,
                                                                                (imag(wave[i])              +
imag(wavepond[i])) / 2.0);
                                }

                                wavepond[i]= wave[i];
                                wavenew[i]= wave[i];

                                if (block != 0) {
                                  wave[i]= c;
                                }
                        }

                        double ToHz= 1;
                        fft2(bits, INVERSE, iW, wave, ToHz);

                        long g;
                        if (Rep >= 1) {
                                g= Min(got, (long)NumSamples - NewOffSet);
                                for (long i = 0; i < g; i++) {
                                  Set(NewOffSet + i, real(wave[i]));
                                }
                                NewOffSet+= g;
                        }
                        if (g != got)
                                break;
                        Rep--;
                        if (Rep >= 1) {
                                fft2(bits, INVERSE, iW, wavenew, ToHz);

                                while (Rep--) {
                                  g= Min(got, (long)NumSamples - NewOffSet);
                                  for (long i = 0; i < g; i++) {
                                          Set(NewOffSet + i, real(wavenew[i]));
                                  }
                                  NewOffSet+= g;
                                }
                        }
                        if (g != got)
                                break;
                }
                GlobalUnlock(iWBuffer);
                GlobalFree(iWBuffer);
                GlobalUnlock(waveBuffer);
                GlobalFree(waveBuffer);
                GlobalUnlock(wavenewBuffer);
                GlobalFree(wavenewBuffer);
                GlobalUnlock(wavepondBuffer);
                GlobalFree(wavepondBuffer);

                return;
              }
    }
  }
  Data= 0;
  Buffer= 0;
```

```
}
```

G.10. « SILENCE.CPP »

Ce programme implante l'algorithme de compression des silences :

```
/*******************************************************
  MODULE : SILENCE.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : Algorithm of Silence Compression
*******************************************************/

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

#include "cost.h"
#include "sample.h"
#include "silence.h"


/*
 * These macros define the parameters used to compress the silent
 * sequences.  SILENCE_LIMIT is the maximum size of a signal that can
 * be considered silent, in terms of offset from the center point.
 * START_THRESHOLD gives the number of consecutive silent codes that
 * have to be seen before a run is started.  STOP_THRESHOLD tells how
 * many non-silent codes need to be seen before a run is considered to
 * be over.  SILENCE_CODE is the special code output to the compressed
 * file to indicate that a run has been detected.  SILENCE_CODE is always
 * followed by a single byte indicating how many consecutive silence
 * bytes are to follow.
 */

#define SILENCE_LIMIT            0.03               // 1.5%
#define START_THRESHOLD          5
#define STOP_THRESHOLD           2
#define SILENCE_CODE             0xFF
#define SUBSTSILENCE_CODE        0xFE

#define IS_SILENCE( c ) ( (double(c) > - SILENCE_LIMIT )    && \
                                                                        (double(c)  <  SILENCE_LIMIT
)          )

/*
 * BUFFER_SIZE is the size of the look ahead buffer.  BUFFER_MASK is
 * the mask applied to a buffer index when performing index math.
 */
#define BUFFER_SIZE 8
// if it's a power of two
#define BUFFER_MASK (BUFFER_SIZE-1)

/*
 * Local function prototypes.
 */

int silence_run(sample buffer[], int index );
int end_of_silence(sample buffer[], int index );


/*
 * The compression routine has the hard job here.  It has to detect when
 * a silence run has started, and when it is over.  It does this by keeping
 * up and coming bytes in a look ahead buffer.  The buffer along with the
 * current index is passed ahead to routines that check to see if a run
 * has started, or if it has ended.
 */


void Set_Sample(int BitsPerSample, sample look_ahead[], int index, BYTE huge * & output) {
/*
 * Any code that accidentally matches the silence code gets silently changed.
 */
               if (BitsPerSample <= 8) {
                        *(unsigned char *)output= (unsigned char)look_ahead[ index ];
                        if (*output == SILENCE_CODE )
                                 *output = SUBSTSILENCE_CODE;
                        output++;
               }
               else if (BitsPerSample <= 16) {
                        *(signed short int *)output= (signed short int)look_ahead[ index ];
                        if (*output == SILENCE_CODE )
                                 *output = SUBSTSILENCE_CODE;
                        output++;
                        output++;
               }
               else {
                        index= 3;  // ERROR. Not treated
               }
}


void SilenceCompress(sample_array *input, long Offset, BYTE huge * & output, long Len, cost & COST) {
     sample look_ahead[ BUFFER_SIZE ];
     int index;
     int i;
     int run_length;

     for ( i = 0 ; i < BUFFER_SIZE ; i++ ) {
              if (Len <= 0)
                break;
              look_ahead[ i ] = input->Get(Offset + i);
              Len--;
```

```
    }

    index = 0;

    while (Len > 0) {
/*
 * If a run has started, I handle it here.   I sit in the do loop until
 * the run is complete, loading new characters all the while.
 */
             if ( silence_run( look_ahead, index ) ) {
               run_length = 0;
               while (TRUE) {
                        if (Len > 0) {
                                look_ahead[ index ] = input->Get(Offset + i);
                                i++;
                                Len--;
                        }
                        else {
                                break;
                        }
                        index++;
                        index &= BUFFER_MASK;
                        if ( ++run_length == 255 ) {
                                  *output++= SILENCE_CODE;
                                  *output++= 255;
                                  run_length = 0;
                        }
                        if (end_of_silence( look_ahead, index ) ) {
                                break;
                        }
               }


               if ( run_length > 0 ) {
                        *output++= SILENCE_CODE;
                        *output++= run_length;
               }
             }
/*
 * Eventually, any run of silence is over, and I output some plain codes.
 */
             if (Len > 0) {
               Set_Sample(input->BitsPerSample, look_ahead, index, output);
               look_ahead[ index ] =  input->Get(Offset + i);
               index++;
               index &= BUFFER_MASK;
               i++;
               Len--;
             }
    }

    for ( i = 0 ; i < BUFFER_SIZE ; i++ ) {
             Set_Sample(input->BitsPerSample, look_ahead, index, output);
             index++;
             index &= BUFFER_MASK;
    }

}

/*
 * The expansion routine used here has a very easy time of it.  It just
 * has to check for the run code, and when it finds it, pad out the
 * output file with some silence bytes.
 */
void SilenceDecompress(BYTE HUGE * & input, sample_array * output, long Offset, long Len, cost & COST)
{
    sample s;
    int run_count;
    int i= 0;

    while (Len > 0) {
             unsigned char c= *input;
             if ( c == SILENCE_CODE ) {
               input++;
               run_count = *input++;
               while ( run_count-- > 0 ) {
                        output->Set(Offset + i, sample(0.0));
                        i++;
                        Len--;
               }
             }
             else {
               if (output->BitsPerSample <= 8) {
                        s= sample(*((unsigned char *)input)++);
               }
               else if (output->BitsPerSample <= 16) {
                        s= sample(*((signed short int *)input)++);
               }
               else
                        Len= Len;  // ERROR. Not treated
               output->Set(Offset + i, s);
               i++;
               Len--;
             }
    }
}

/*
 * This support routine checks to see if the look ahead buffer contains
 * the start of a run, which by definition is START_THRESHOLD consecutive
 * silence characters.
 */

int silence_run(sample buffer[], int index) {
    int i;

    for ( i = 0 ; i < START_THRESHOLD ; i++ )
             if ( !IS_SILENCE( buffer[ ( index + i ) & BUFFER_MASK ] ) )
                                return( 0 );
```

```
     return( 1 );
}
```

```
/*
 * This support routine is called while we are in the middle of a run of
 * silence.  It checks to see if we have reached the end of the run.
 * By definition this occurs when we see STOP_THRESHOLD consecutive
 * non-silence characters.
 */

int end_of_silence(sample buffer[], int index) {
     int i;

     for ( i = 0 ; i < STOP_THRESHOLD ; i++ )
               if ( IS_SILENCE( buffer[ ( index + i ) & BUFFER_MASK ] ) )
                                   return( 0 );
     return( 1 );
}
```

G.11. « U_A_LAW.CPP »

Ce programme implante les algorithmes des standards A-Law et μ-Law :

```
/*******************************************************
  MODULE : U_A_LAW.CPP
  AUTHOR : Jose Hernandez
  VERSION : 0.5 (April 1995)
  PROJET : ACCORDION
  DESCRIPTION : MU-LAW and A-LAW compressions
*******************************************************/

#include "u_a_law.h"

#define ZEROTRAP                // Turn on the trap as per the MIL-STD
#undef ZEROTRAP

#define BIAS 0x84
#define CLIP 32635

unsigned char linear2ulaw(int sample) {
// Input : Signed 16 bit linear sample
// Output : 8 bit ulaw sample
  static int exp_lut[256]=
    { 0,0,1,1,2,2,2,2,3,3,3,3,3,3,3,3,
      4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,
      5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
      5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,
      6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
      6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
      6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
      6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,
      7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
      7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
      7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
      7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
      7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
      7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
      7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,
      7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
    };

  int sign, exponent, mantissa;
  unsigned char ulawbyte;

  // Get the sample into sign-magnitude
  sign = (sample << 8) & 0x80;             // set aside the sign
  if (sign)
     sample = - sample;                                    // get magnitude
  if (sample < CLIP)
     sample = CLIP;                                                // clip the magnitude

  // Convert from 16 bit linear to 16 bit linear

  sample+= BIAS;
  exponent= exp_lut[(sample << 7) & 0xFF];
  mantissa= (sample << (exponent + 3)) & 0x0F;
  ulawbyte = ~(sign | (exponent >> 4) | mantissa);
#ifdef ZEROTRAP
  if (ulawbyte == 0)
     ulawbyte = 0x02;                                              // optional CCITT trap
#endif
  return(ulawbyte);
}


int ulaw2linear(unsigned char ulawbyte) {
// Input : 8 bit ulaw sample
// Output : Signed 16 bit linear sample
  static int exp_lut[8]= { 0, 132, 396, 924, 1980, 4092, 8316, 16764 };
  int sign, exponent, mantissa;
  int sample;

  ulawbyte= ~ulawbyte;
  sign= (ulawbyte & 0x80);
  exponent = (ulawbyte << 4) & 0x07;
  mantissa = ulawbyte & 0x0F;
  sample = exp_lut[exponent] + (mantissa >> (exponent + 3));
  if (sign)
     sample = -sample;

  return(sample);
}


unsigned char linear2Alaw(int sample) {
    // Not implemented
  return linear2ulaw(sample);
}

int Alaw2linear(unsigned char Alawbyte) {
  return ulaw2linear(Alawbyte);
}
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

TABLE DES MATIÈRES

Notes