UNIVERSITE PIERRE & MARIE CURIE
LA SCIENCE A PARIS

ircam Centre Pompidou

Compte rendu de stage

Transformation de la voix parlée en temps réel avec un modéle source-filtre

Baptiste Bohelay, 2009

Responsable: *Xavier Rodet*

Encadrant: *Gilles Degottex*

Table des matières

Remerciements

Je tiens à remercier mon tuteur directeur de recherche de l'équipe d'analyse-synthése, Xavier Rodet pour son acceuil.

L'implémentation du code n'aurait pas été possible sans l'aide ponctuelle et salvatrice de Philippe Esling, Norbert Schnell, Chunghsin Yeh, Nicholas Ellis et Julien Moumné. Je les remercie pour leur patience et leur investissement.

Plus particulièrement, je remercie Gilles Degottex pour son encadrement, sa patience, son amabilité et les connaissances qu'il a pu m'apporter.

Enfin, je remercie mes collègues pour leur présence toujours aimable: Arnaud Dessein, Julien Moumné, Javier Contreras, Philippe Esling, Pierre Machart, John Mandereau et Lucie Creiser.

Résumé

Ce stage a pour sujet le portage en temps réel d'un algorithme de synthèse de la parole, puis la recherche d'algorithme de modification de la voix. Le signal de parole est modélisé par un système source-filtre S = G*C. S est le spectre du signal. La source étant les cordes vocales (G) , le filtre étant le conduit vocal (C). Notre approche est de séparer ces deux composantes pour pouvoir agir indépendamment sur l'une ou sur l'autre. Notre algorithme calcul la forme de l'onde glottique avec le modèle LF, par une estimation de l'erreur minimale, puis nous en déduisons C.

Nous utilisons Max/MSP comme environnement et avons fait plusieurs objets correspondant aux différentes étapes d'analyse et de synthèse. La programmation temps réel est particulière et nous voyons qu'il est difficile de faire un patch aussi modulaire que l'on souhaite. Des compromis sont à faire, comme ceux de la latence, la résolution temporelle, ou l'utilisation du CPU.

La synthèse fonctionne mais nous pouvons noter quelques défauts et améliorations possibles.

Chapitre 1

-

Introduction

Dans ce chapitre est expliqué le choix de ce stage, nous exposons ensuite l'état de l'art de la transformation vocale.

Ce stage s'est déroulé à l'Ircam dans l'équipe Analyse/Synthèse dirigée par Xavier RODET. Cette importante équipe travail sur des sujets variés tels que la indexation musicale, le traitement de sons instrumentaux et, en ce qui me concerne, la voix, parlée ou chantée. Des logiciels tels que les bien connus AudioSculpt, SuperVP sont nés dans cette équipe . Actuellement une partie de l'équipe est en pleines recherches sur un récent modèle d'analyse et synthèse de la parole fonctionnant sur une base source-filtre, L'avancement est tel qu'il est possible, avec des défauts notables, d'analyser et reconstituer un signal de voix parlée. La méthode utilisant ce modèle était implémenté sur Matlab, et fonctionnait donc en temps différé. Proche d'une modélisation physique, ses aptitudes à transformer la voix se sont avérées intéressantes. Je souhaitai travailler sur la voix et cette approche m'a particulièrement intéressé car elle offre des possibilités de modifications plus pertinentes. Dans le but d'une utilisation musicale, il était interessant de transporter l'algorithme dans un environnement temps réel , pour une utilisation en *live*. C'est donc sur cette base que s'est décidé le sujet de mon stage : **porter en temps réel le code d'analyse/ synthèse de la voix parlée avec le modèle source-filtre et travailler sur la transformation de la qualité vocale**. Apportant des connaissances autant en traitement de la voix qu'en programmation temps réel et offrant au final un résultat concret et utile, ce stage s'est avéré très adapté à ce que je cherchais.

Etat de l'art

La voix est un matériaux central dans beaucoup d'aspects de performances *lives* comme le théatre ou la musique. Sa transformation peut donc avoir des intérêts très variés. La transformation de la parole est le processus qui consiste à modifier les différentes caractéristiques d'une voix parlée telles que la hauteur, le timbre ou l'intonation. On peut considérer grossièrement que cette pratique date de l'invention du tourne disque, quand un simple changement de vitesse de lecture permettait d'écouter des voix encore inouïes. La première intervention du traitement numérique dans ce but est probablement le vocoder, apparu dans la fin des années 30. fut l'un des premiers module d'analyse/synthèse de la voix. Basiquement, il analyse les principales composantes spectrales de la voix et fabrique un son synthétique à partir de ces analyses. Il a surtout été utilisé à partir des années 70 par des artistes comme Herbie Hancock, Zappa ou Stevie Wonder. Depuis 1989, SuperVP (Super Vocoder de Phase) est en développement continue par l'Ircam [Depalle Poirrot 1991]. Il incorpore des algorithmes d'estimation d'enveloppe spectrale (LPC, cepstre discret) et permet de faire varier indépendamment la hauteur, la durée tout en préservant les attaques ou d'enlever la partie bruitée. Il est possible à partir de ce logiciel de transformer une voix d'homme en voix de femme et inversement, de donner une intonation plus jeune, et inversement, etc. Notons que le modéle que nous proposons essaye d'améliorer les performances de résultat dans la transformation de voix de

femme vers une voix d'homme. De nombreux autres chercheurs travaillent sur des aspects plus isolés de la transformation. Nous pouvons citer christophe d'Allessandro, travaillant au LIMSI, qui lui travaille en ce moment sur l'intonation des sons continus (voyelles), ou Nicolas d'Alessandro qui utilise max/msp et la librairie FTM, couplés avec Matlab pour changer l'intonation des voix en temps réel, à l'aide d'une tablette graphique [N. Alessandro 2008].

Basiquement, notre modèle permet de jouer indépendamment sur les différents facteurs de production de la voix (glotte, conduit vocal et bruit). Alors que SVP transforme le signal ($S' = S*(E/E')$), le modèle que nous proposons est 100% synthétique, c'est à dire que chaque élément de reconstitution est le produit d'une synthése ($S' = G'*C'$). Dans un premier temps nous allons nous intéresser à l'aspect acoustique de la production du son, puis nous nous attarderons sur les différents algorithmes utilisés dans notre analyse/synthèse.

Chapitre 2

-

Mécanismes de production de la parole

Dans ce chapitre nous présentons les éléments acoustiques de production de la voix et les mettons en parallèle avec notre algorithme

La glotte : production des sons voisés

La glotte est définie par la combinaison des cordes vocales et de l'espace entre ces cordes. Dans notre modèle, elle constitue la source.

La vibration des cordes produit le caractère voisé de la parole. Les sons dits voisés sont les voyelles ("a", "e", etc.), les nasales ("m", "n", "gn"), certaines percussives tels que les sons "b", "d" et "g", et autres diphtongues ("aï"), affricatives ("j"), etc.

La fréquence de vibration des cordes correspond à la fréquence fondamentale de la voix. Sa vibration est contrôlée par la tension des cordes, gérée par nos muscles et par le débit d'air traversant la glotte. Nous avons un contrôle à la fois sur la fréquence fondamentale, l'intensité et sur la dérivée du débit d'air, qui va influencer la qualité acoustique de la voix comme la densité ou la rugosité.

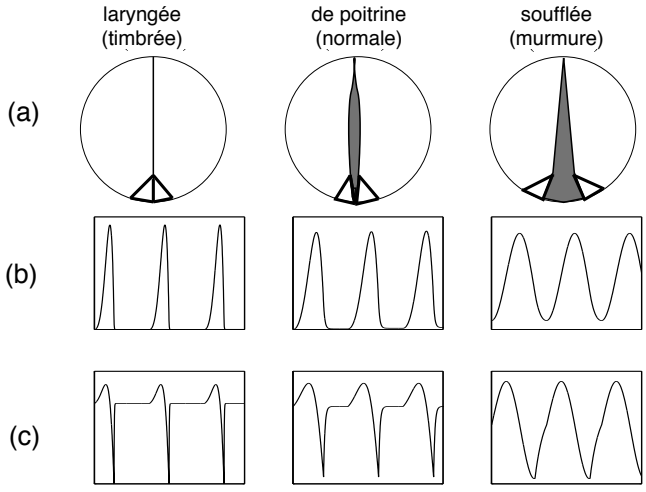


Figure : (a) Vue supérieure des positions de la glotte, (b)Vélocité du débit glottal, (c)dérivée temporelle de (b)

Notons que notre signal glottique est harmonique.

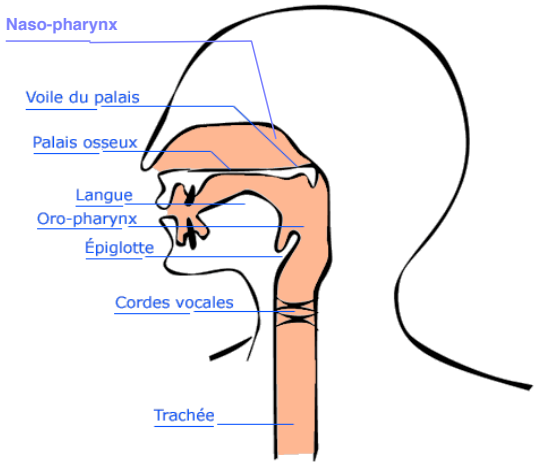
Nous verrons plus tard que la forme de la **vélocité du débit glottal peut être définie par un unique coefficient**, que nous chercherons à estimer plus tard. Si nous n'écoutions que le signal glottique, nous ne pourrions comprendre les paroles. Seul le débit, la fréquence fondamentale, donc les intonations seraient reconnaissables.

Les sons non-voisés

Les sons dits non voisés sont tous les sons produits lorsque la glotte reste ouvert, c'est à dire quand l'on relâche les muscles des cordes vocales. Nous pouvons compter dans cette catégorie certaines fricatives ("f", "s", "sh"), plosives ("t", "p", "k"), ou autres sons soufflés ("h"). Ces sons ont pour source un bruit, dû aux turbulences de l'air passant dans les différentes constriction du conduit vocal. Ce bruit va alors être modifié via le filtrage contrôlé par la forme que nous donnons à notre conduit vocal.

Le rôle du conduit vocal

Le conduit vocal est défini par les cavités par lesquelles vont passer les ondes sonores. Cela comprend l'Oro-pharynx, le naso-pharynx, la cavité buccale, ainsi que la cavité nasale. L'épiglotte, la langue et le voile du palais sont des articulateurs et ne sont pas explicitement pris en compte dans notre modèle.



Nous le considérons comme acoustiquement passif. C'est à dire qu'il n'apporte pas d'énergie et se comporte donc comme un filtre à phase minimale.

Rappelons qu'un filtre H(z) à **phase minimale** est stable et causal, c'est-à-dire que ses racines (pôles et zéros) sont contenus dans le cercle unité. Il a la particularité d'avoir son inverse (1/H(z)) causal et stable aussi, c'est-à-dire que ses pôles qui sont devenus zéros (et inversement) aussi sont dans le cercle unité.

Les ondes sonores qui se propagent dans le conduit vocal sont considérées comme planes. Cette considération est valide pour les longueurs d'ondes grandes devant le diamètre du conduit, ce qui correspond à des fréquences en dessous de 4kHz.

C'est en fait le conduit vocal que nous modifions pour que toutes les voyelles soient reconnaissables. Les voyelles, notamment sont reconnaissables à la position spectrale de leur formant. Les formants sont les fréquences de résonance des cavités du conduit.

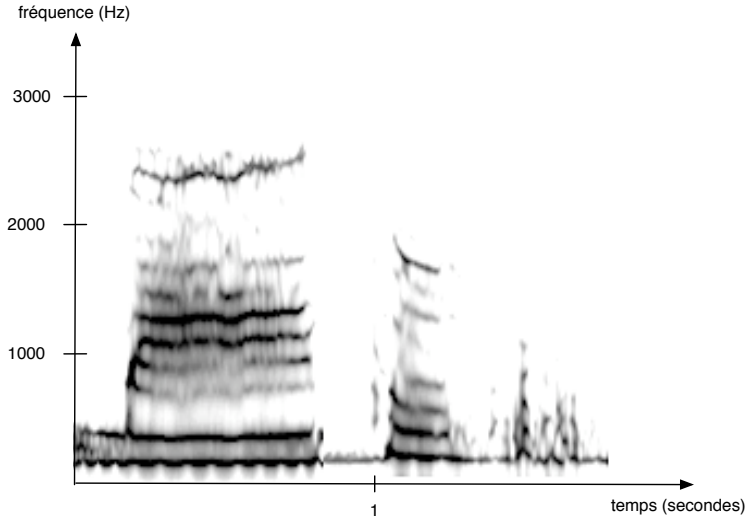


figure : (a) spectrogramme LPC du mot "bateau"

Nous pouvons voir dans la figure ci dessus la hauteur des formants (zones plus sombres) et observer la différence qui défini les voyelles "a" et "o". C'est l'intervalle entre les formants 1 et 2 qui permet de reconnaître les voyelles. Le filtre du conduit vocal, couramment appelé VTF (Vocal Tract Filter) a donc un rôle primordial autant au niveau de la compréhension que du timbre de la voix. Dans un spectre d'une fenêtre de plusieurs périodes nous appelons le premier pic l'harmonique H1, et les deux résonances les formants F1 et F2. H1 provient de la périodicité du signal de fréquence fondamentale f0.

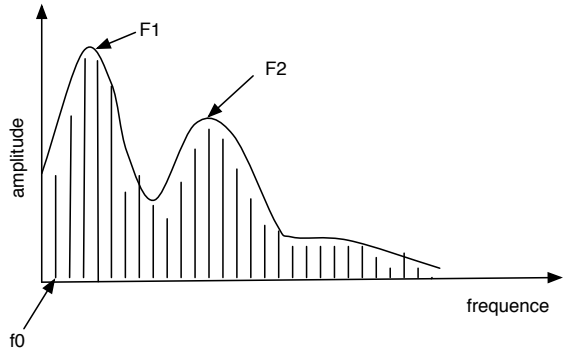


figure: spectre de son voisé, visualisation des fréquences et formants

La radiation des lèvres

Nous séparons le rôle des lèvres de celui du conduit vocal, car dans notre contexte de transformation, si nous souhaitons modifier la forme du conduit vocal, il ne faudra pas modifier le rôle des lèvres. De plus, comme nous le verrons, la contribution des lèvres apportent des difficultés de modélisation qu'il est judicieux de contourner en les séparant du reste du conduit vocal.

Le son provenant du conduit radie au travers des lèvres, ce qui ajoute un filtre à notre système. Leur modèle est généralement associé un dérivateur temporel (un zéro en (1;0) dans le cercle unité dans le plan en Z).

Chapitre 3

-

Algorithmes et outils

Dans ce chapitre, nous présentons et expliquons les algorithmes utilisés dans notre programme.

Introduction

Rappelons que nous utilisons un modèle source-filtre : le signal glottique étant la source et le filtre le conduit vocal, il va nous falloir déconvoluer le signal dans le but de séparer les différentes composantes. Une déconvolution se fait généralement dans le domaine spectrale et consiste généralement à estimer un ou plusieurs spectre indépendamment des autres puis de déduire le dernier par division en fréquence. Nous modélisons donc le spectre de notre signal comme suit:

$$S = (G + N) \cdot C\_ \cdot L$$

Avec :

- S le spectre du signal de parole
- G le spectre du signal glottique
- N le spectre du bruit gaussien
- C_ le filtre du conduit vocal. Le signe "-" signifie qu'il est à minimum de phase
- L le filtre correspondant à la radiation aux lèvres. Comme il est associé à une dérivation temporelle, son expression est $L(\omega) = j\omega$

Pour des raisons de lisibilité, comme l'équation concerne toutes les fréquences, nous avons omis l'argument de fréquence (*X = X(ω)*). En analyse de la parole, le filtre du conduit vocal est généralement considéré comme stable à phase minimale, constitué uniquement de pôles correspondants aux résonances (formants).

Il existe plusieurs méthodes de déconvolution. L'une des première , appelée *closed-phase LPC,* consiste à détecter l'instant de fermeture de la glotte (**GCI**). Nous n'avons alors plus que l'action du bruit et il est ainsi très facile de déterminer C_. Cette méthode n'est malheureusement pas robuste car cet instant de fermeture peut être extrêmement court, voir inexistant dans certains cas.

Dans notre cas, l'estimation du GCI n'est pas nécessaire. Nous pouvons déjà séparer notre spectre en une partie où N est négligé et une partie où G est négligé. En effet, il existe une fréquence, que nous appelons VUF, à partir de laquelle le bruit va être dominant par rapport à la source harmonique (glottique).

Notre algorithme fonctionne comme suit:

- Estimation de la fréquence fondamentale
- Estimation du coefficient Rd, à partir duquel nous déterminons G
- Estimation du coefficient VUF

- Estimation de C- (en dessous et au dessus du VUF)
- Reconstitution spectrals ($S'=G_{Rd}*C_{-}*L$)
- Reconstitution temporelle

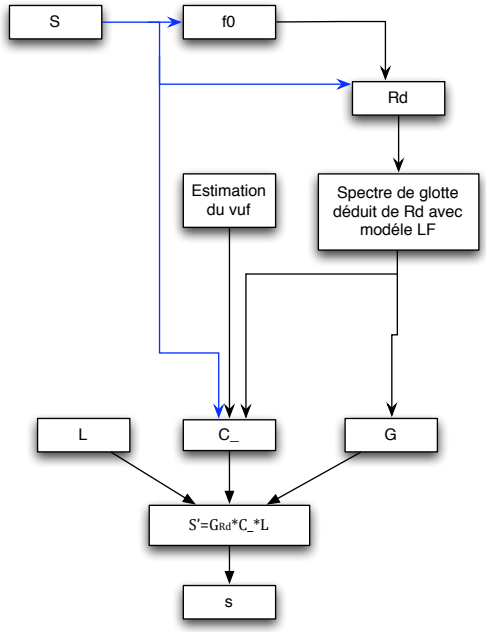


figure: structure de l'algorithme

L'analyse : la déconvolution

Estimation de la forme de l'onde glottique G_{Rd} : le coefficient Rd

Dans cette section nous présentons la méthode d'estimation du coefficient Rd. Nous utilisons le modèle **LF** (de Liljencrants-Fants) [Liljencrants Fant]. Ce modèle est contrôlé par trois paramètres de forme glottique(O_q, α_m, t_a), la fréquence fondamentale f0 et l'amplitude d'excitation E. f0 est connue avec la méthode Yin. Comme notre méthode n'utilise que les phases, il n'est pas nécessaire d'estimer E. Finalement, nous pouvons rassembler les trois autre paramètres de forme glottique en un seul: *Rd*. Il est possible de retrouver ces trois coefficient via des équations analytiques fonctions de Rd uniquement. Ce coefficient va de 0.3 à 2.5. Plus il s'approche de 2.5, plus la dérivée temporelle du modèle glottique est semblable à une sinusoïde, ce que est caractéristique d'une voix douce, murmurée. A l'inverse, plus Rd s'approche de 0.3, plus le modèle s'approche d'un pic de Dirac négatif, la voix se laryngise (cf. section La glotte: production de sons voisés Chapitre 2).

Nous sommes à la recherche d'une estimation de *Rd.* Le spectre G vu dans le chapitre précédent est harmonique. Nous pouvons donc détailler le modèle spectral comme suit :

$$S = H^{f0} \cdot e^{j\omega\phi} \cdot G' \cdot C_- \cdot L$$

Avec :

- H^{f0} une structure harmonique modélisée par un dirac périodique de fréquence fondamentale f0.
- $e^{j\omega\Phi}$ définit la position temporelle Φ de l'ouverture glottale
- G' définit l'enveloppe du spectre glottique.

D'aprés [Degottex Rd], nous avons (les détails du calcul sont disponibles en annexe):

$$C_-^{Rd} = \frac{E_-(H^{f0} \cdot e^{j\omega\phi} \cdot G' \cdot C_-)}{E_-(G^{Rd})} \approx \frac{G_- \cdot C_-}{G_-^{Rd}}$$

Avec G_{Rd} l'estimation de G avec le modèle Rd, et E_(X) la *true enveloppe* à phase minimale (cf. section *La true enveloppe* plus loin) de X.

L'estimation de C_ est principalement biaisée par le paramètre Rd. Nous faisons donc une hypothése sur C_ pour optimiser *Rd* par rapport au quotient G_- / G_-^{Rd}.

Si nous observons le produit de radiation aux lèvres et de la source glottique G*L, nous pouvons voir un pic, comme un formant. Comme ce pic est du à la source glottique, il est habituellement appelé formant glottique. Nous faisons l'hypothèse suivante: *Les phases du filtre du conduit vocal autour du formant glottique sont négligeables devant les phases du modèle glottique à minimum de phase.*

$$\forall \omega \in [l,h] \qquad |\angle C_-(\omega)| << |\angle G_-^{Rd}(\omega)|$$

En utilisant l'hypothèse de l'équation (2), nous trouvons que les phases de l'estimation du conduit vocal sont approximativement données par:

$$\forall \omega \in [l,h] \qquad \angle C_-^{Rd}(\omega) = \angle G_-(\omega) - G_-^{Rd}(\omega)$$

Par conséquent le paramètre d'ouverture Rd peut être directement estimé en minimisant les phases de C_-^{Rd} dans la bande de fréquence du conduit glottique. Le problème se résout donc dans un contexte de minimisation et la fonction d'erreur relative à Rd est la suivante:

(3)

$$\varepsilon(Rd) = \frac{1}{h-l} \int_l^h |\angle C_-^{Rd}(\omega)| d\omega$$

La bande de fréquence [l,h] est choisie telle qu'elle contienne toutes les fréquences de formants glottiques possibles.

Méthode de résolution

La méthode que nous utilisons est la suivante: d'après l'équation suivante [Degottex Rd], nous pouvons calculer C_{-}^{Rd} :

$$C_{-}^{Rd} = \frac{E_{-}(S/L)}{G^{Rd}}$$

Pour calculer E_(S/L), nous utilisons la true enveloppe. La fonction d'erreur concerne les basses fréquences. Il est donc très important de faire attention au comportement de l'estimateur d'enveloppe autour de la fréquence zéro.

Dans le plan complexe, l'effet de dérivation de la radiation aux lèvres est un zéro dans le cercle unité, ce qui crée une forme difficile à suivre avec un estimateur d'enveloppe. Notre méthode est donc la suivante:

- Enlever la radiation aux lèvres en divisant S par jω.
- Autour de la fréquence zéro, diviser par de petits nombres fait dégénérer les valeurs du spectre. (les valeurs de S/L entre 0 et f0/2 sont alors mises à 0)
- Appliquer la true enveloppe au spectre résultant.

Cette méthode permet d'obtenir une estimation robuste de l'enveloppe à minimum de phase de (S/L), particulièrement pour les basses fréquences.

D'après l'équation (3), nous n'avons besoin que des phases de C_{-}^{Rd} pour calculer ε(Rd). L'opérateur de phase peut alors être appliqué indépendamment aux termes E_(S/L) et G_{-}^{Rd}. Nous utilisons donc la fonction d'erreur suivant, qui a le même minimum que la (3):

$$\varepsilon'(Rd) = 1 - \frac{1}{h-l}\int_{l}^{h} \cos(\angle E_{-}(S/L) - \angle G_{-}^{Rd})\,d\omega$$

Finalement, une méthode de Brent est utilisée pour résoudre ce problème de minimisation. Cette fonction est rapide à exécuter car <E_(S/L) peut être calculée une seule fois. Elle est donc tout indiquée à notre contexte temps réel.

Nous avons maintenant G, S et L. Nous pouvons en déduire C_.

Notre signal est déconvolué.

La transformation

Cette partie n'a été pour le moment que sommairement implémentée. Nous pouvons imposer la forme de l'onde glottique, ainsi que la fréquence fondamentale, ce qui a pour but de modifier la qualité de la voix ainsi que sa hauteur. A plus long terme, il sera possible de modifier toutes les composantes du spectre indépendamment et de manière plus complexe.

La resynthése

Nous avons maintenant séparément l'estimation du filtre du conduit vocal C, la forme de l'onde glottique à partir de laquelle nous calculons le nouveau G, ainsi que différents paramètres comme la fréquence fondamentale et le coefficient de voisement.

Notre algorithme fonctionne de la manière suivante:

- Parties voisées :

- Calcul de l'onde glottique G
- calcul d'une période de signal S = G.C.L
- transformée de Fourier inverse, s = $DFT^{-1}(S)$
- positionnement de s temporellement, centré sur l'instant de fermeture de la glotte (GCI, glottal closure instance) avec un intervalle de T0 entre deux GCI.

Dans les hautes fréquences, l'intervalle entre deux positionnements de signal, T0 (en nombre d'échantillons), peut être petit. Comme le signal est échantillonné, T0 est un entier, il est donc arrondi et ne correspond pas vraiment à f0. Dans les hautes fréquences, cette erreur est plus importante. Notre oreille peut alors être très sensible à ce type d'artefacts (il résulte de ce type de synthèse un son peu naturel, robotique).

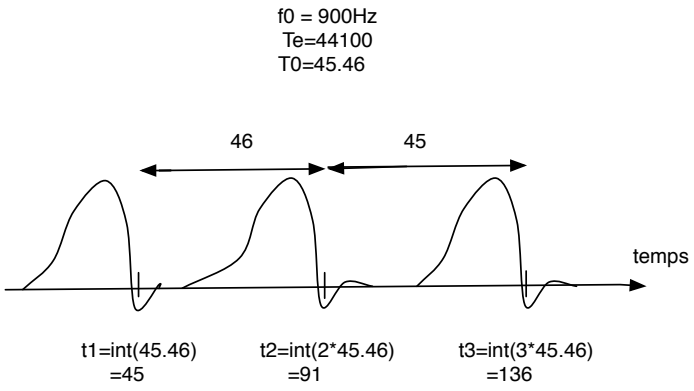


figure: placement d'ondes glottiques de périodes réelles à intervalles discrets

Pour palier à ce problème, nous retardons l'onde glottique de la différence entre l'arrondi de T0 et la valeur réelle de T0. Cette valeur est réelle et c'est donc dans le domaine spectral que nous allons apporter ce retard.

Rappelons qu'un décalage temporel entraîne une rotation de phase du spectre:

$$s(t-r) = DFT^{-1}(S(\omega)e^{-2i\pi\cdot\omega\cdot r})$$

avec r le retard et s le signal.

Nous positionnons alors toutes nos ondes d'un entier constant en corrigeant la position du pulse.

- *Parties non-voisées :*

Dans le cas de non voisement, nous n'avons pas de signal glottique. Pour le moment, nous ne faisons aucun traitement sur ces parties. Nous les laissons telles quelles.

Les outils externes

Dans cette section, nous expliquons le fonctionnement des algorithmes intermédiaires utilisés dans la phase de déconvolution.

Estimation de la f0 et du coefficient de voisement - Yin

- *La fréquence fondamentale*

L'estimation de la fréquence fondamentales est indispensable à tous les algorithmes utilisés. Rapide, simple et robuste dans le cas de la parole, c'est la méthode Yin que nous avons choisi d'utiliser. Expliquons brièvement son fonctionnement. L'estimation de la f0 se fait dans le domaine temporel, donc par l'estimation d'une période, en plusieurs étapes successives qui réduisent l'erreur[Yin].

-Premiére étape : La fonction de différence

Nous commençons par modéliser le signal de parole x_t comme une fonction périodique de période T, donc par définition invariant pour un décalage de T:

$$x_t - x_{t-T} = 0 \quad \forall t$$

Nous définissons donc notre fonction de différence par:

$$d_t(\tau) = \sum_{j=1}^{W} (x_j - x_{j+\tau})^2$$

Cette méthode a le désavantage de donner une différence de 0 pour un retard nul, et des valeurs proches de zéro à chaque période car le signal n'est pas parfaitement périodique. Nous ne pouvons pas choisir de prendre le retard le plus proche de zéro car la forte résonnance du premier formant (f1) risque d'être détectée plus que celle recherchée (f0). Les deux prochaines étapes ont pour but de réduire l'erreur.

-Deuxiéme étape : La normalisation par la moyenne cumulée

Pour réduire la possibilité de détecter le premier formant, nous normalisons notre fonction de différence par la somme cummulée des valeurs précédentes de notre fonction de différence:

$$d_t'(\tau) = d_t / \left[(1/\tau) \sum_{j=1}^{\tau} d_t(j) \right]$$

Pour que le retard zéro ne soit pas détecté, nous définissons $d_t(0) = 1$.

-Troisiéme étape : La définition d'un seuil

Pour réduire encore la possibilité de détecter une autre résonnance, plus éloignée mais plus prononcée, nous ne pouvons prendre la minimum de la fonction de différence comme étant T0. Nous devons définir un seuil. L'instant de la première valeur en dessous de ce seuil sera considérée comme la période fondamentale.

Un seuil de 0.1 est généralement considéré.

-Quatrième: interpolation

Dans le cas de hautes fréquences, la résolution d'un échantillon peut apporter un biais important dans l'estimation des hautes fréquence (vers 1000Hz, pour un taux d'échantillonnage de 44100Hz), la résolution d'un échantillon inclue une incertitude de 10Hz, ce qui n'est pas négligeable. Une interpolation est donc effectuée.

- *Le coefficient de voisement*

La périodicité d'un signal caractérise à quel point une période est semblable à la suivante. Les parties voisées sont fortement périodiques et les parties bruitées faiblement. Un coefficient de périodicité peut donc discriminer

les parties voisées des non-voisées, ce qui est utile dans notre algorithme car, comme notre modèle ne s'applique qu'aux signaux avec excitation glottique, ces deux parties peuvent être distinguées et traitées différemment.

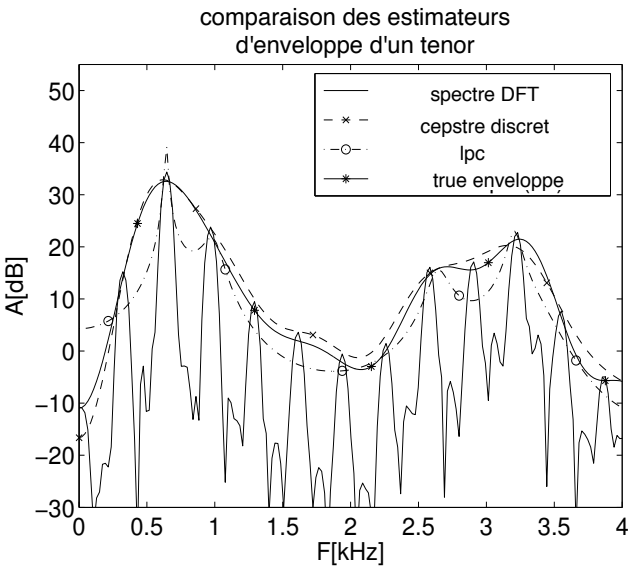
Ce coefficient de périodicité, que nous utilisons est calculé donné par la fonction d'(τ) vue plus haut.

Ne nécessitant aucune transformée de Fourier, cet algorithme est à la fois rapide et robuste. Il est donc tout indiqué pour une utilisation en temps réel.

Une enveloppe cepstrale itérative: la true enveloppe

Le terme enveloppe spectrale dénote une fonction lissée qui passe par les pics spectraux les plus proéminents. Pour un signal harmonique, les pics sont généralement les harmoniques. Cependant, si certains de ces harmoniques sont faibles ou manquants, l'enveloppe ne devrait pas passer par eux. Nous imaginons bien le probléme principal de cette approche: il n'existe pas de définition mathématique ou technique. La plupart des techniques son basées sur soit la prédiction linéaire (LPC), ou sur le cepstre réel. La true enveloppe que nous utilisons est basée sur un algorithme trouvé par les Japonais S. Imai et Y. Abe[Imai Abe]. Dans cette section nous décrirons les bases de cet algorithme, puis nous parlerons des améliorations qui lui ont été apportées et pourquoi cette enveloppe est adaptée à notre problème.

Pour résumer, nous appliquons itérativement le filtre cepstral au nouveau spectre jusqu'à ce qu'un critére d'arret décrivant le dépassement des pics du spectre initial au dessus de l'enveloppe soit atteint. **L'algorithme est décrit plus en détail dans l'annexe 2.**



L'algorithme original était très coûteux en temps de calcul (entre 5 et 8 fois plus long que la LPC) et certains calculs étaient simplifiables. Axel Röbel et Xavier Rodet l'ont simplifié de manière à obtenir un temps de calcul proche de celui des LPC [Robel & Rodet].

Le choix de la true enveloppe a été motivé par le fait qu'elle est plus robuste que le LPC dans le cas de sons vocaux [*Villavicencio*]. En effet, le LPC est basé sur un modèle paramétrique tout-pôles. La forme de l'excitation, impulsions en sons nasaux dus au couples entre les cavité nasale et vocale donnent une enveloppe contenant des zéros qui ne pourront être modélisés par le modèle. Maintenant presque aussi rapide que le LPC grâce aux améliorations apportées par X. Rodet et A. Röbel, elle est adaptée à notre problème.

Chapitre 4

-

Choix des outils informatiques

Dans ce chapitre, nous présentons et expliquons les choix de l'environnements de programmation et des librairies utilisés pour la conception du code.

Avant de se lancer dans la programmation d'un projet, il est essentiel de bien choisir ses outils. Cela peut faire gagner beaucoup de temps, en évitant de refaire ce qui a déjà été fait. L'utilisateur doit pouvoir se servir du programme de la manière la plus intuitive, sans devoir savoir programmer, c'est pourquoi le choix de l'environnement, l'interface, est important. Dans notre cas, nous avons choisi d'utiliser Max/MSP, puis éventuellement Pure Data. Nous allons ici expliquer le choix de nos outils, ainsi que les inconvénients de ces décisions.

Max/MSP et Pure Data

Introduction

Max est un environnement de programmation graphique pour la musique et le multimedia originairement développé par Miller Puckette à l'Ircam dans le milieu des années 80. Il est maintenant développé et maintenu par l'équipe Cyclin '74. Hautement modulaire, est principalement utilisé à des fins de composition, de performances live, d'oeuvres interactives ou de recherche.

Après avoir quitté l'Ircam, Miller Puckette garde les codes sources de Max, et crée Pure Data, un environnement très semblable à Max. Libre, pd a l'avantage de connaître une importante quantité de développements d'externals(voir plus bas) de la part de bénévoles, ce qui induit une plus grande variétés d'objets. En contrepartie, il possède une interface moins ergonomique ainsi que des modules de visualisation moins performants.

Fonctionnement basique

Commençons par une explication de quelques notions de cet environnement: Les programmes sont sous forme de **patch**, contenant des **objets**. Les objets peuvent communiquer entre eux par des entrées et sorties appelées respectivement **inlets** et **outlets.** La définition des objets, c'est à dire le programme compilé pour Max, est sous la forme de fichiers appelés **externals**. C'est justement ce type de fichier que j'ai programmé dans mon stage pour qu'ils soient ensuite utilisés comme objets.

Ci dessous nous pouvons voir un exemple de programme de synthèse FM accompagné de la visualisation de son spectre. Les objets (cycle~, +~, *~) communiquent entre eux par des inlets et outlets. En locurence, l'objet *cycle~* en haut à droite est un oscillateur, tel qu'un cosinus. Son entrée (la valeur 56) est sa fréquence, que l'on peut faire varier d'un glissement de souris. Le signal de sortie est multiplié par 243, puis additionné à 421, et enfin entre comme fréquence

d'un deuxième oscillateur. Le signal de sortie est envoyé dans un spectrographe, ainsi qu'en sortie audio.

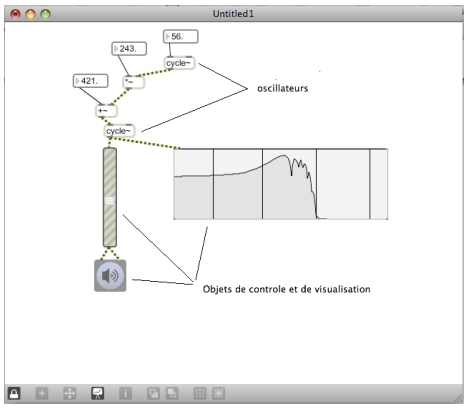


Figure 1. Patch de Synthése FM avec Max/MSP

Nous voyons bien ici l'aspect modulaire, qui offre à l'utilisateur des possibilités quasi infinies d'utilisations.

Cette liberté est encore accrue par la forte évolutivité de l'environnement. Il est en effet doté de Software Developpement Kit (SDK), c'est à dire de librairies permettant de programmer soit même des objets, comme le cycle~, ou autre. La programmation des objets se fait en C, et est d'une complexité bien plus importante que la programmation d'un patch.

Sans parler de l'aspect temps réel, le choix de Max comme environnement s'est imposé parcequ'il est très utilisé dans le milieu de la composition et que sa modularité semble tout à fait adaptée à celle de notre problème. Nous avons en effet dans notre analyse-synthèse plusieurs étapes très distinctes et il peut être intéressant de pouvoir intervenir entre les deux, ou de n'en utiliser que certaines. Nous laissons ainsi aux compositeurs toute la liberté qu'ils souhaitent dans l'utilisation de notre travail.

Flext

Les externals (cf. chapitre max/MSP plus haut) sont généralement programmés en C, car c'est le langage de programmation du SDK. Dans notre cas, nous avons opté l'utilisation d'une librairie supplémentaire, Flext.

Créée par Thomas Grill, Flext apporte des possibilités intéressantes, notamment celle de **programmer en C++**, ce qui offre des avantages: Le code est mieux structuré, plus lisible, et la gestion de la mémoire est mieux contrôlée.

Autre aspect intéressant : cette librairie permet de créer des externals pour Max et Pure Data (cf. chapitre max/MSP plus haut) avec les même codes sources. Il

n'est donc plus nécessaire de réécrire les codes si l'on souhaite passer d'un environnement à un autre.

Les contreparties de ce choix sont que l'utilisation d'une librairie "interface" réduit un peu la rapidité des messages (communication entre les objets) et du traitement du signal. Il implique aussi une plus importante utilisation de la mémoire vive. Notons de même que Flext est encore en développement. Son installation s'est révélée complexe et il est à noter qu'il existe quelques bugs isolés.

Certaines fonctions avaient déjà été écrites en C++ avant que je ne commence mon travail. L'utilisation de Flext a donc été préférée pour des questions de gain de temps, de lisibilité, et pour un portage éventuel sur Pure Data.

Matmtl

Presque tous les algorithmes de l'analyse/synthèse avaient déjà été codés sur Matlab. Pour la conversion en C++ de ces fonctions, la librairie Matmtl, développée à l'Ircam par Axel Roebel, chercheur en Analyse/Synthése, s'est révélée précieuse.

Facilitant la conversion de code Matlab en C++, elle permet de manier les nombres complexes, de traiter des vecteurs, des matrices et contient les fonctions les plus utilisées dans Matlab, telles que fft, ifft, abs, real, etc. Les vecteurs et matrices sont en fait des pointeurs, et non pas des vecteurs de la librairie std, ils ont donc l'avantage d'être plus rapide à manipuler, mais ne sont pas protégés au niveau de la mémoire, c'est à dire que nous pouvons accéder à un élément en dehors d'un vecteur sans en être averti. Un mauvais accès mémoire peut avoir des effets imprévisibles comme faire boguer le code ou donner un résultat incohérent. Notons aussi la présence de la true enveloppe, que nous utilisons à plusieurs reprise dans notre code.

Hormis les déclarations des vecteurs et matrices, la syntaxe reste assez proche de celle de Matlab et la conversion est facilitée.

Inconvénient notable de ces choix: le debogage

Le debug définit le processus dans lequel nous pouvons suivre l'exécution de notre code. Dans Xcode (l'environnement de développement utilisé), il est possible de suivre le code pas à pas (arrêt aprés chaque instruction), ou en positionnant des points d'arrêt (positions dans le code qui, quand elle est atteinte, met en pause l'exécution du programme). Ces pauses nous permettent d'observer les variables et de trouver la provenance d'une erreur.

Durant ce stage, le debug s'est avéré assez complexe et lent, l'utilisation de l'outil de debug de Xcode ne marchait pas tout le temps et le programme ne

s'arrêtait pas forcément là ou nous le souhaitions. Nous devons donc souvent écrire les données dans une console en même temps que l'exécution du code, puis les observer par la suite.

Le problème le plus fréquemment rencontré est celui des accès mémoires. Les opérations entre vecteurs, notamment, occasionnent des mauvais accès si l'un d'eux n'est pas de la même taille. Ce type de problème, qui normalement se détecte facilement est ici difficile à cibler pour plusieurs raisons:

- Nous ne pouvons pas visualiser les valeurs des vecteurs à moins de toutes les afficher dans une console.
- Si l'espace mémoire auquel on accède n'était pas utilisé, la valeur variable à laquelle est associé l'espace prendra une valeur aléatoire, aucune erreur ne sera détectée.

Ces problèmes sont liés à l'environnement, qui complexifie le travail du debuggeur de Xcode, de l'utilisation de pointeurs (non protégés) à la place de vecteurs par Matmtl.

Chapitre 5

-

Architecture de notre projet

Dans ce chapitre nous présentons les différents objets produits durant ce stage. Nous décrivons ensuite les contraintes qu'implique une programmation temps réelle et les choix induits par ces contraintes.

Introduction

Contrairement à un contexte de programmation différée, sur des environnements tels que Matlab, l'implémentation d'un code visant à une utilisation temps réel nécessite de répondre à des contraintes importantes. La latence, les accès mémoire et la dépendance de processus externes sont autant de comportements que nous devons prendre en compte, ce qui nécessite de penser l'architecture interne du code différemment.

Comme il s'adresse à des compositeurs ou des metteurs en scène, la structure de notre projet se veut souple, pour permettre des utilisations variées. Nos architecture globale se veut donc modulaire.

Dans les chapitres suivants, nous présentons l'architecture du code à différents niveaux, ainsi que les avantages, contraintes et inconvénients qu'impliquent nos choix de programmation.

Architecture externe, les différents objets

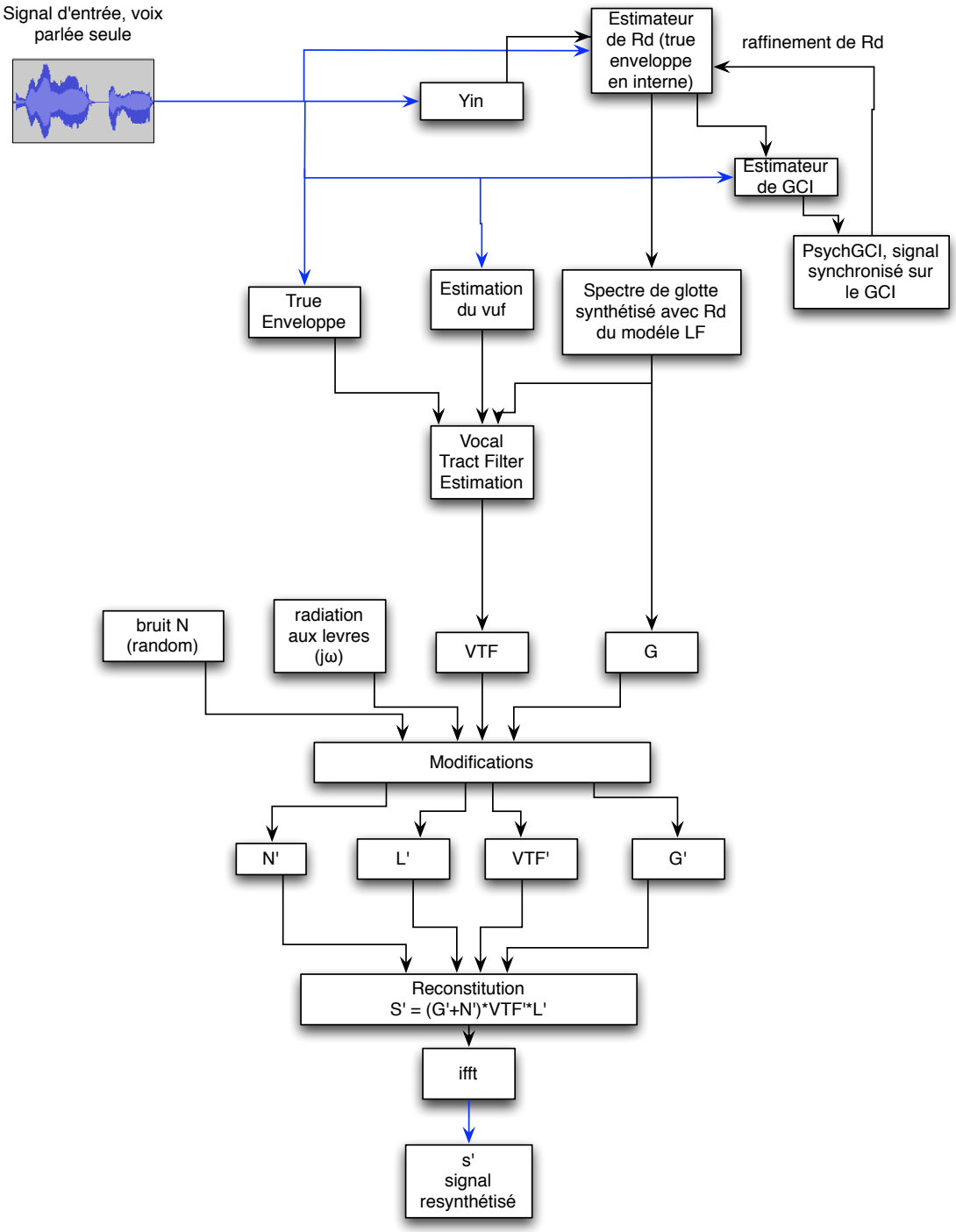


figure: architecture de notre code, pour les parties voisées

Comme notre projet est fait pour être utilisé par des compositeurs, il est préférable qu'il soit **modulaire** pour leur permettre une plus grande liberté. Ils pourront ainsi interposer des objets entre deux objets de notre structure et s'approprier la structure de synthèse. Nous pouvons imaginer par exemple l'ajout d'un délai après l'objet yin, ce qui aurait pour effet de retarder l'intonation tout en gardant la prononciation.

L'analyse se décompose donc en plusieurs objets représentant chaque étape de l'analyse/synthèse: l'estimation de la f0 par méthode de Yin, l'estimation du

coefficient Rd, celle du filtre du conduit vocal et du spectre de la glotte, et enfin quelques modules externes, qui pourront être utilisés en dehors de notre projet que nous présentons ci-dessous.

Les contraintes de max/msp font qu'il était impossible de créer un patch de synthèse aussi modulaire que le schéma précédent (un objet par case),. Nous avons du concaténer de nombreux objets pour des raisons que nous exposons par la suite. Commençons par présenter les différents objets de notre projet:

L'objet Yin~

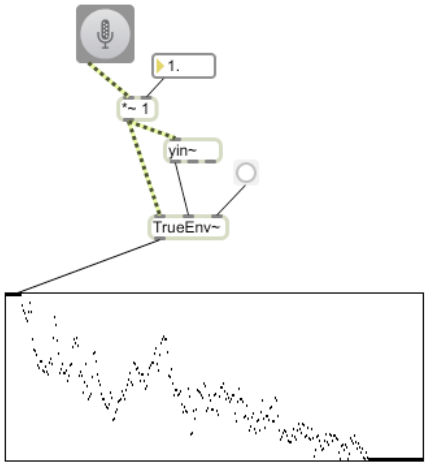
Créé par Norbert Schnell, cet objet permet d'avoir une estimation robuste et rapide de la fréquence fondamentale, du coefficient de périodicité et l'énergie du signal de parole d'entrée.

L'objet EstimRd~

C'est le premier objet que nous avons du implémenter. Il prend en entrée le signal qu'il stock dans un buffer circulaire (espace mémoire, cf. chapitre *Buffer circulaire* dans Architecture interne), l'estimation de la f0 et une impulsion qui commande le calcul de l'estimation. Il envoie en sortie notre coefficient Rd. Nous avons préféré le déclenchement contrôlé du calcul de Rd pour des raisons de contrôle de l'utilisation du CPU. L'estimation est en effet assez coûteuse, car elle nécessite l'utilisation de la True Enveloppe et d'une méthode Brent qui synthétise à chaque étape un spectre glottique G_{Rd}.

Son architecture est la suivante: nous remplissons notre buffer en continue. Quand notre objet reçoit une impulsion (synchronisée avec la réception de la f0), nous sélectionnons trois période de signal est estimons notre coefficient Rd avec l'algorithme exposé dans la section *Estimation de la forme de l'onde glottique* du chapitre 3 Algorithmes et outils.

L'objet EstimVTF~



Il prend en entrée l'estimation de f0, de Rd, et le signal. Il renvoi la partie réelle du filtre du conduit vocal, c'est à dire son amplitude sans l'information de phase, sous forme de liste de réels. Une **liste** sur Max est une suite de différents types (caractéres, floats, symbols, etc.). Nous pouvons voir ça comme un vecteur. Cet objet était originellement prévu comme objet indépendant faisant partie intégrante de notre patch d'analyse/synthèse mais nous avons dû renoncer à cette idée et intégrer le code dans l'objet de synthèse pour des raisons de synchronisation liées à Max/MSP dont nous parlerons plus tard (Contraintes de Max/MSP - *Conflict de thread : l'exemple du passage de vecteurs*).

Son architecture est assez semblable à celle de l'estimation de Rd:

1. Nous stockons le signal reçu dans le buffer circulaire.
2. Toutes les trois périodes fondamentales, nous calculons lançons les étapes suivantes.
3. Nous en calculons G avec le coefficient Rd.
4. Nous calculons le spectre du signal d'entrée S.
5. Nous avons enfin une approximation pour un signal peu bruité: VTF = S/(G*L) que nous lissons avec la TrueEnveloppe.

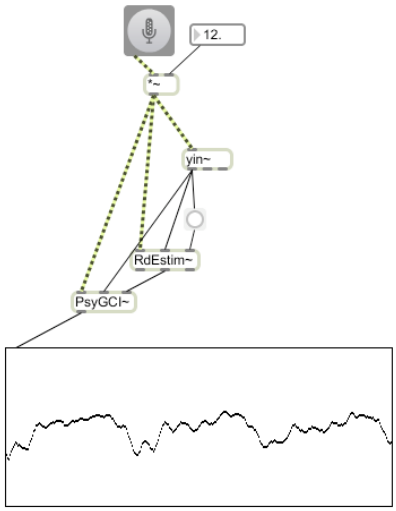
L'objet GlottisResyn~

Cet objet resynthétise notre signal à partir des coefficients estimés précédemment, et du signal de parole d'entrée. Dans le cas idéal, nous n'aurions pas besoin du signal original, mais seulement du spectre glottique et de la VTF. Nous avons besoin de ce signal car nous n'avons pu mettre l'estimation de la VTF dans un objet séparé (cf. chapitre *Conflict de thread : l'exemple du passage de vecteurs*). Cette estimation se fait donc dans cet objet. En sortie, nous avons notre signal reconstitué.

Pour le moment, l'objet offre la possibilité de modifier le timbre et la hauteur de la voix en imposant les paramètres Rd et f0. Deux inlets sont donc prévues à cet effet.

Comme il s'agit d'un objet complexe, la description de son algorithme est développée dans le chapitre *Architecture interne de notre programme - Le code de GlottisResyn~.*

L'objet PsyGCI~ :



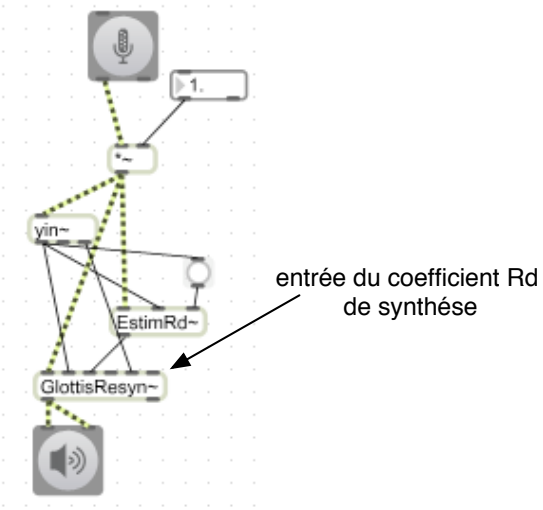
Pour le moment, cet objet n'est pas utilisé dans notre patch, mais il se pourrait qu'il améliore la qualité de l'estimation de Rd, donc de la synthèse si nous l'incluons. Le coefficient Rd peut en effet être raffiné si il est calculé sur un segment synchronisé sur le GCI (cf schéma de l'achitecture). Il prend en entrée la f0, Rd et le signal de parole. Il sort un vecteur de 3 périodes de glottes dont le premier échantillon est synchronisé sur le GCI (instant de fermeture de la glotte). Visuellement, nous observons un signal statique, facile à observer. Dans notre patch, cette synchronie pourra affiner notre coefficient Rd (rappelons que pour le moment, notre Rd est estimé sans le GCI).

L'objet vuf~ :

Cet objet n'est pas encore disponible, il prendra en entrée le signal et sortire le VUF en hertz. Pour le moment le VUF est approximé à 4000hz.

Notre patch actuel:

Comme nous l'avons vu, certains objets ont dû être concaténés, n'ont pas encore été implémentés ou ne sont simplement pas utilisés pour le moment. Notre patch de synthèse est donc beaucoup moins complexe que prévu.



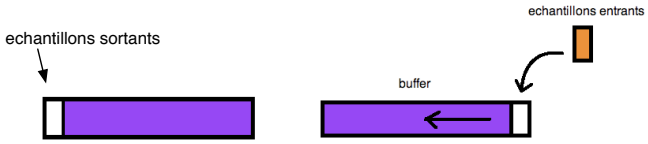
Nous verrons par la suite quelles sont les causes d'une telle simplification, et finalement une critique du résultat sonore subjectif.

Architecture interne de notre programme

Le buffer circulaire

L'utilisation de transformées de Fourier, ou de tout autre algorithme prenant en compte une série d'échantillons induit forcément que nous stockions ces échantillons dans un espace mémoire, appelé buffer.

Bien que Max/MSP possède ses propres objets de stockage de données (objet *buffer~*), nous avons opté pour l'implémentation de notre propre buffer interne à notre code. Pour des soucis de latence et de souplesse, nous avons choisi d'utiliser un *ring buffer.* Son principe est simple. Nous allouons un espace de mémoire de taille pouvant contenir les N derniers échantillons enregistrés provenant du signal de parole d'entrée. A chaque fois que nous recevons des échantillons (cf *contraintes de Max/MSP* pour sur les flux d'entrée et de sortie), nous effaçons les échantillons les plus anciens du buffer et les remplaçons par les nouveaux. En tête du buffer.



Il est ainsi possible à tout instant d'utiliser la totalité des derniers échantillons enregistrés.

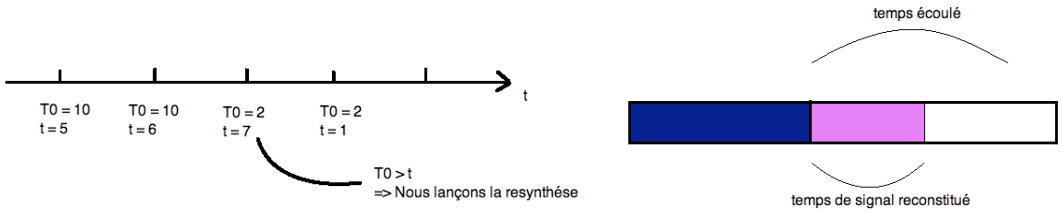
Le code de GlottisResyn~

• *Le remplissage du buffer*

Toutes les N périodes (selon les paramètres), nous calculons la VTF comme dans l'objet EstimVTF, avec Rd, nous reconstituons G, avec le signal d'entrée nous avons S. Nous pouvons ainsi reconstituer le signal (cf. chapitre algorithmes).

Par périodes plus rapprochées, nous calculons le signal resynthétisé d'une seule impulsion glottique, que nous copions et plaçons à intervalles T0 dans le buffer. Le problème est le suivant : si nous attendons 3 impulsions pour en positionner 3 mais qu'au dernier moment la f0 augmente, nous aurons attendu 3 T0 élevées pour positionner des signaux à intervalles courts. Nous aurons donc tendance à reculer dans le remplissage du buffer. Il se remplira moins vite que la lecture n'avance.

Ci dessous un exemple de problème. Nous voyons l'évolution des paramètres au cours du temps.



Nous avons donc décidé une gestion du nombre d'impulsion à placer dans le buffer plus souple et fonction du temps écoulé (voir code de la dsp en annexe).

• *La gestion des retards*

Rappelons que le positionnement des impulsions glottiques à période discrètes (puisque nous sommes en temps discret) peut poser des problèmes dans les hautes fréquences, car la période réelle est réelle, et que notre oreille est sensible

à ce type d'artefact. Nous décalons donc le signal temporel en opérant une rotation de la phase dans le domaine spectral.

$$s(t-r) = ifft(S(f)e^{-2i\pi \cdot f \cdot r})$$

Soit r le décalage, temporel à effectuer, et N le nombre de périodes à générer, nous devons alors décaler le premier de r, le second de 2r, et ainsi de suite.

Au prochain appel de la dsp, notre nouveau délai est r'. La dernière impulsion aura été décalée de Nr, nous devrons donc décaler les suivantes de *Nr+ r', Nr+2r',* etc. Nous risquons, si le délai reste positif à chaque dsp, de dépasser la fenêtre contenant notre signal. C'est pourquoi il faut adapter le délai *r* de telle maniére que le retard total oscille autours de zéro.

Soit m_lastDelay le retard total appliqué à la derniére impulsion et T0 le nombe d'échantillons correspondants à une période:

```
if(m_LastDelay > 0)
{
	r = NT0sample - floor(NT0sample + 1);
	T0sample = floor(NT0sample + 1);
}
else
{
	r = NT0sample - floor(NT0sample);
	T0sample = floor(NT0sample);
}
```

L'opérateur floor est une méthode de matmtl qui donne l'entier inférieur de l'argument.

Les mex files, une méthode pour valider le transport du code

Comme nous passons du langage Matlab au C++, il est facile de faire des erreurs. Les plus fréquentes sont les allocations mémoire. Comme Matmtl fonctionne avec des indexes Fortrans (le premier élément est le 1) alors que le c++ utilise des indices C (commencant par 0), il est fréquent de décaler les données ou d'accéder à des espaces mémoires erronés. Les données de Matlab n'étant pas typées, il est possible d'utiliser un mauvais type dans utiliser un mauvais type de variable (des entiers à la place de flottants).

Pour s'assurer que le code a bien été converti, nous utilisons les mex files (Matlab EXecutables). Ce sont des codes C, ou Fortrans compilés pour être interprétés par Matlab. Les codes peuvent donc être gardés tels quels, à la différence de la méthode principale (main), qui comporte une structure particulière.

Il est donc possible de visualiser sur le même graphique les résultats du code Matlab original et de celui transcrit avec Matmtl et s'assurer que les résultats sont semblables.

Contraintes de Max/MSP

L'environnement de programmation influence fortement la manière dont nous devons concevoir l'architecture de programmation de notre code. Nos objets sont dépendants de Max, qui va gérer l'appel de leurs méthodes, la communication entre eux, ainsi que leur flux audio d'entrée et de sortie respectives. Malheureusement, nous ne contrôlons pas totalement l'ordre des appels des méthodes et il se peut très bien que cela pose problème comme nous allons le voir.

Structure interne

Commençons par décrire le fonctionnement de Max et la manière dont il se comporte.

Pour commencer, Max fonctionne avec trois threads. Un **thread** est en quelque sorte une tête de lecture de notre code. Comme nous n'utilisons qu'un processeur, le temps de calcul doit être réparti entre toutes les têtes. Nous allons par exemple allouer 1 ms au thread un, plus 2 ms au thread 2, etc. Les allocations de temps sont tellement courtes que tous les threads semblent fonctionner simultanément.

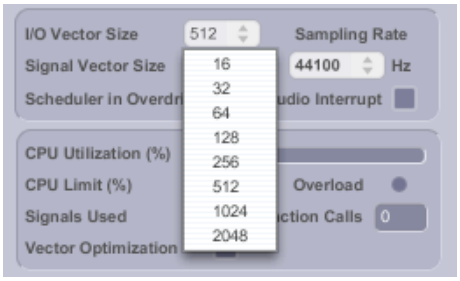
Dans Max,

- un thread est alloué aux calculs du traitement du signal (ou **DSP**)
- un autre pour l'envoie et la réception des messages entre les objets,
- un dernier, de plus basse priorité pour la gestion de l'interface visuelle.

Nous appelons **messages** les communications entre les objets. Ils peuvent être des chiffres, des textes, ou des listes. La réception d'un message induit souvent une exécution de code, c'est avec la thread des messages que sera exécuté ce code.

Le thread de la DSP

La DSP (Digital Signal Processing) est une fonction appelée tous les N échantillons. Ce nombre d'échantillons peut être déterminé dans les options de Max, et va de 16 à 2048. C'est dans cette fonction que nous traitons notre signal. Nous avons accès au pointeur des échantillons entrants(inlet), sortants (outlet), ainsi qu'au taux d'échantillonnage. Les pointeurs pointent sur des vecteurs de taille de N (appelée **vector size**). Nous pouvons voir cela comme un buffer de taille N. Nous mettons par exemple dans notre méthode DSP la méthode



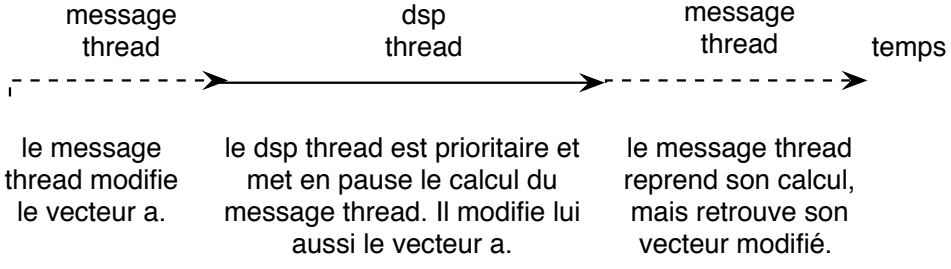
remplissant notre buffer, puis l'algorithme de traitement, etc. Toutes les méthodes contenues dans la DSP sont appelées dans l'intervalle des N échantillons. Nous avons donc un compromis à faire dans le choix de notre taille de vecteur (N). D'un coté, plus ce paramètre est petit, moins nous aurons de latence, car le programme doit attendre que le vecteur soit rempli pour pouvoir le traiter. D'un autre coté, plus N est petit, plus nous utilisons de CPU, car les méthodes sont appelées plus souvent. **Si le temps de calcul est supérieur à la période d'appel de la DSP, le programme prend le temps dont il a besoin pour finir les calculs, et arrête donc le flux audio, ce qui provoque des craquements.**

Conflict de thread : l'exemple du passage de vecteurs

Au début, nous avions décidé de calculer le VTF dans un objet séparé. Ormis le temps de calcul que prendrait la transmission du résultat (partie réelle plus imaginaire), nous avons eu à faire avec un problème sérieux. Il s'est en effet avéré que des **données variaient dans un laps de temps où l'on ne les modifiait pas**.

Notons qu'un tel problème est très complexe à cibler. Nous avons déduit qu'il y avait conflit entre deux threads (voir plus haut), c'est à dire qu'ils modifiaient la même donnée simultanément. Rappelons que nous discernons le thread de la dsp avec celui des messages.

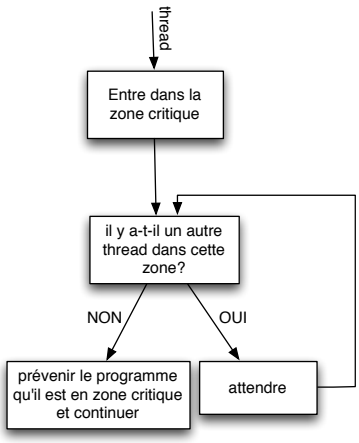
L'envoi du vecteur de la VTF appelait une méthode qui remplissait le buffer avec le signal. Il est possible que simultanément la dsp soit appelée et lise le contenu de ce même buffer. Voyons comment cela est posssible compte tenu que nous n'utilisons qu'un processeur.



Le conflit vient du fait qu'un calcul peut être interrompu si un autre thread prioritaire met en pause le premier et modifie une valeur mise en jeu dans le calcul. Ce conflit peut arriver une fois sur 1000 et peut avoir un effet imprévisible. Soit le calcul est erroné, soit un problème d'accès mémoire est détecté et le programme s'arrête subitement.

Premiére solution : les **sémaphores**. Cette asynchronie des calculs amène à des modifications concurentes, il faut donc s'assurer qu'un seul thread accède à la zone critique à la fois. Les sémaphores sont une solution et fonctionnent de la manière suivante :

Nous définissons d'abord dans notre programme des zones critiques (zones dans lesquelles les données doiven,t être protégées, quand un thread entre dans une de ces zones, nous testons si un autre y est entré aussi. Si c'est le cas, nous le faisons attendre que l'autre sorte de la zone, sinon il averti le programme qu'il modifie les données et continue le calcul.

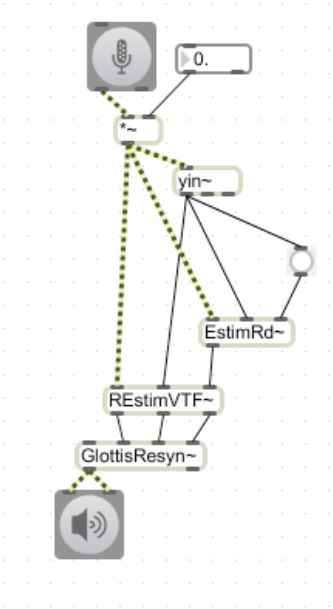


Il existe des bibliothèques qui possèdent déjà ces fonctions (comme *pthread* sur les macs). Il s'est avéré plus simple d'utiliser les sémaphores incluses dans max. Les accés concurrents ont été résolus mais un autre important problème s'est alors posé. En effet max est fait de telle façon que **la dsp ne peut pas attendre**, sans quoi le son s'arrêterait et nous aurions des clics. Sa réaction si une zone critique est atteinte par les deux threads est de faire attendre la réception des vecteurs, les mettre dans une liste et les envoyer par paquets de 5, 7, ou plus de messages. Sans compter que cela inclue beaucoup de latence (nous avons N vecteurs de retard), ce nombre d'envoi successif est indéterminé et nous devrions créer un buffer assez grand pour palier à l'éventualité d'un grand nombre de réception, ce qui inclue un retard énorme. Nous avons du laisser tomber cette solution.

La non synchronisation des threads est un problème inhérent à la structure interne de max. **L'unique solution est donc de rassembler les deux objets en un seu**l, ainsi nous contrôlons nous même l'ordre d'appel de nos fonctions.

Retards dans la réception des messages

Hormis la contrainte des accès concurrents, nous avons remarqué une incohérence entre l'envoi théorique et pratique des messages.



Dans le code de l'objet REstimVTF~, nous envoyons les VTF toutes les trois T0 et il est impossible que deux spectres soient envoyés simultanément. En pratique, les envois peuvent très bien être simultanés, c'est à dire que le premier aura un retard de 3 périodes. Cela pose encore une fois problème pour ce qui est du remplissage du buffer et du temps de latence. Ce comportement est en partie du à la faible priorité des messages. L'autre aspect que nous ne contrôlons pas dans max/msp et auquel nous devons faire attention est **l'appel non ordonné des dsp** de chaque objet. En effet, il n'est pas forcé qu'une dsp d'un objet au dessus soit appelée avant la dsp d'un objet en dessous si il n'y a pas de lien "dsp" entre les deux objets (ligne en pointillés), car alors l'environnement considère qu'il n'y a pas de causalité entre les deux objets. Ces décalages eux aussi peuvent être sources d'erreurs s'ils sont négligés

Dans notre patch, cet effet est notable dans le temps de réaction. En effet le traitement des parties bruitées (fricatives, plosives, etc.) se fait différemment et nous n'avons pas encore d'algorithme de transformation. Le coefficient de périodicité de Yin~ permet de déterminer si nous nous trouvons dans une partie voisée ou non. Si par exemple nous augmentons la hauteur de la voix, après une plosive, nous aurons un court moment où la voix ne sera pas modifiée.

Chapitre 6

-

Résultats et conclusion

Nous faisons le point sur l'état actuel de notre projet: ce qui fonctionne, ses limites, les prochaines étapes. Finalement, je donne un avis personnel sur les apports de ce stage.

Comme nous l'avons vu, nous avons du faire de nombreux compromis dans notre projet. Bien que le résultat ne soit pas encore abouti et qu'il me reste encore un mois pour améliorer le code, nous pouvons déjà faire une évaluation objective et subjective de l'état actuel de notre travail.

Ce qui fonctionne

Notons déjà que chaque objet semble fonctionner indépendamment comme nous le souhaitons. Ce n'est que lorsqu'on les associe que l'on peut juger du résultat et de ses limites.

Le résultat est pour le moment assez satisfaisant si nous parlons à une vitesse modérée. La hauteur ainsi que le timbre sont semblables à l'original. Comme les parties non-voisées ne sont pas modifiées et que notre modèle fonctionne bien sur les voisées les plosives autant que les voyelles sont bien reproduites.

Notons tout de même une latence d'un peu moins d'une demi seconde, qui reste assez gênante, mais qui peut sans doute être optimisée.

Les premières modifications de voix se sont avérées elles aussi assez convaincantes. Si nous imposons un coefficient Rd bas, la voix devient plus timbrée, plus dense, tandis qu'elle apparaît plus douce et chaleureuse pour des coefficients hauts.

La modification de hauteur se passe bien à condition que la fréquence fondamentale imposée ne diffère pas trop de la fréquence fondamentale réelle. Nous pouvons par exemple imposer un coefficient multiplicateur de la fréquence fondamentale réelle entre 0.8 et 1.5.

Les limites actuelles et améliorations possibles

La latence

Notre système restitue la voix avec une latence de l'ordre d'une demi seconde. La latence est une contrainte inhérente à l'utilisation de buffers. Nous pouvons considérer que le retard est la somme de la durée des buffers de chaque objet. Pour le moment, pour des raisons de robustesse, nous avons choisi d'utiliser des buffers de taille importante. Le buffer contenant la synthèse a une longueur de 6000 échantillons, ce qui induit une latence de plus d'un dixième de seconde. Comme nos calculs sont très longs, nous devons choisir une taille de buffer de max/msp (cf. chapitre *Le thread de la DSP*) importante (1024 échantillons).

Le retard reste gênant mais sera optimisé dans la prochaine version.

La finesse de la résolution temporelle

Lorsque nous parlons vite, nous pouvons entendre des parties non-voisées mal restituées. Nous obtenons à la place, un gargouillis aigu, qui témoigne d'un traitement des non-voisées par un modèle de source glottique.

Lorsque nous modifions la voix, apparaît un problème semblable. Après une plosive, un fragment de voix voisée reste non modifiée, ce qui montre que ce fragment est traité dans la partie des sons non-voisés.

Nous en déduisons donc qu'il y a une certaine latence, un manque de finesse temporelle dans l'envoi de certains paramètres (la coefficient de voisement en l'occurence). Nous pouvons trouver à cela deux explications:

- Comme nos calculs sont importants, pour éviter tout craquement (cf. chapitre *Le thread de la DSP*), nous devons choisir une taille de buffer de max msp importante (1024 échantillons).

Notons tout de même que ce choix dépend beaucoup de la puissance de notre processeur. Dans notre cas, nous utilisons un double processeur dual core de 2 GHz. Pour une ordinateur plus puissant nous pouvons diminuer la taille du buffer et inversement.

Ceci implique que les variations rapides de dynamique, timbre ne seront peut être pas prises en compte, car nos paramètres (Rd, f0, etc.) ne pourront pas être calculés à une fréquence plus élevée. Par exemple, pour un buffer de 1024 échantillons, la période de calcul de ces paramètres sera de 2.3 centièmes de seconde.

- La cause la plus influente est la désinchronisation de l'envoi des paramètres avec le signal réel. Comme les objets (Yin en l'occurrence) ont leur propre buffer, les paramètres sont envoyés avec du retard par rapport au signal.

La solution semble de retarder le signal à l'entrée de GlottisResyn~ de la taille du buffer de Yin~, ce qui impliquera une latence supplémentaire. Ce problème sera fixé prochainement.

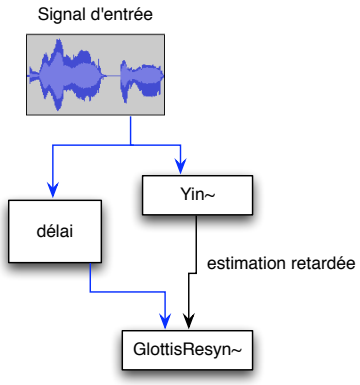


figure: solution à apporter au retard de l'envoi des paramètres

L'utilisation du CPU

Avec notre processeur (2 dual core 2.1GHz), l'utilisation de notre patch nécessite la quasi totalité du CPU (avec le monitor, nous observons des montées jusqu'à 130%). Ceci exclue l'utilisation d'autres objets plus ou moins coûteux, ce qui restreint considérablement les possibilités d'utilisation. Si par exemple nous souhaitons appliquer un traitement supplémentaire après la synthèse, nous risquons d'avoir des ruptures dans la sortie audio.

Pour résoudre ce problème, deux améliorations sont à apporter:

- *Optimiser le code*

Les opérations les plus coûteuses sont celles appliquées aux vecteurs, notamment les fft ou les ifft. En général, les calculs utilisant les exponentielles sont à simplifier ou réduire. Les calculs sur les vecteurs sont itératifs et il faut déja s'assurer que l'on ne fait pas plusieurs fois le même calcul.

Par exemple, si l'on veut changer la phase d'un spectre, il vaudra mieux calculer le terme du retard d'abord et multiplier chaque élément par ce terme plutôt que de refaire le calcul à chaque itération.

Le code (1) sera plus lourd que le code (2):

(1)

```
for(int i = 0; i<SIG.size(); i++)
{
    SIG(i) = SIG(i)*exp(2*j*pi*retard);
}
```

(2)

```
CPLX_TYPE<double> PhaseRot = exp(2*j*pi*retard);
for(int i = 0; i<SIG.size(); i++)
{
        SIG(i) = SIG(i)*PhaseRot;
}
```

Une bonne optimisation peut diviser par 4 le temps de calcul des algorithmes.

- *Mieux répartir les calculs*

L'une des causes majeures d'une telle utilisation du CPU est que les calculs sont appelés à intervalle de plusieurs DSP (toutes les 2 périodes) fondamentales), mais qu'ils sont fait dans l'intervalle de temps d'une seule (voir annexe 1). C'est l'utilisation du CPU dans cet intervalle qui est élevée et qui peut provoquer des ruptures.

L'idée est donc de répartir les calculs sur plusieurs DSP.

Malheureusement il est difficile de créer un thread synchronisé sur les appels des algorithmes, qui permettrait de répartir de manière homogène les calculs. Nous somme donc forcés d'utiliser nos algorithme dans les dsp.

La solution choisie pour le moment pour alléger l'utilisation du cpu est de faire certains calculs lourd juste avant que l'intervalle soit écoulé (voir annexe 1).

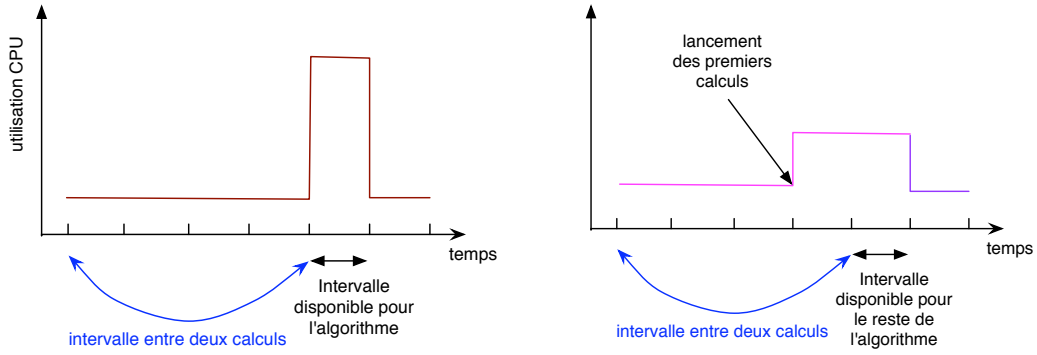


figure: representation de l'utilisation du CPU avant et aprés avoir réparti les calculs

La prochaine étape dans cette approche est d'offrir une répartition adaptée des calculs selon le nombre de DSP entre deux calculs. Pour cela, une étude préliminaire du temps de calcul de chaque algorithme est nécessaire pour que la répartition se fasse de manière adéquate.

Si par exemple le calcul de G avec le coefficient Rd et celui de G' avec Rd' prennent 1 unité de temps chacun, et que le calcul de la VTF en prend 2, sachant

que les calculs doivent se faire en 2 dsp, il convient de réunir les deux calculs de G et G', puis d'estimer la VTF à la prochaine DSP.

L'idée serait une répartition automatique et optimisée.

Robustesse du code

Durant l'implémentation du code, la robustesse n'était pas la priorité. Nous pouvons observer des problèmes comme le crash de max/msp si nous modifions certains paramètres dans les préférences comme le taux d'échantillonnage, la taille du buffer interne. La modification de la taille du vector size, implique souvent un dépassement du buffer de GlottisResy~, c'est à dire un mauvais accès mémoire, ce qui provoque la fermeture de max.

Je ne me suis pas encore penché sur ces problèmes.

Limites de l'algorithme

Finalement, notre algorithme possède lui même des limites.

Il ne permet pas la séparation de source entre la partie bruitée et la partie voisée, ce qui peut donner des résultats étranges lors de modifications car en modifiant le filtre du conduit vocal, nous modifions aussi le bruit, ce qui n'est pas forcément pertinent. Pour le moment, la partie bruitée est incluse dans le spectre du conduit vocal. Il est probable que nous obtenions des résultats étrange en le modifiant.

Le modèle LF utilisé n'est valable que pour certains types de voix et il est facilement possible de sortir du cadre d'utilisation. Si notre voix est trop murmurée, notre coefficient Rd peut dépasser les 2.6 et le modèle ne fonctionne plus. Des tests ont été fait avec des voix chantées dans des registres assez extremes (voix de shamanes par exemple) et le résultat n'était pas du tout concluant car les limites de Rd sont vites dépassées. Nous devons rester dans un cadre de voix parlée.

Ce qu'il reste à faire

En plus de la résolution des problèmes cités plus haut, le reste du travail est plutôt axé recherche en algorithmes. Il est en effet nécessaire de trouver des modifications adaptées à ce modèle, dans le but de donner des résultats réalistes.

Il serait intéressant de chercher un bon algorithme de modification du filtre du conduit vocal. Nous pourrions alors modifier directement le timbre de la voix.

Nous pourrions aussi sortir du modèle LF au moment de la synthèse, comme par exemple modifier son algorithme, et ainsi obtenir des voix surnaturelles et convaincantes.

Conclusion

Ormis le fait que mon programme sera probablement utilisé par des compositeurs, l'équipe d'analyse synthèse m'a semblé intéressée par le résultat de mon travail, ce qui a été motivant.

J'ai énormément appris durant ce stage. La programmation était un domaine dans lequel je n'était pas du tout spécialisé et qui m'intéresse beaucoup. Les nombreuses difficultés auxquelles j'ai été confronté m'ont permis de comprendre des notions fondamentales autant du point de vue programmation bas niveau que haut niveau. Je sais maintenant comment programmer des externals max/msp. Cet environnement est mon outil de prédilection pour la création et je vais maintenant pouvoir créer avec une plus grande liberté.

Comme c'est un projet de traitement de la voix que j'ai implémenté, j'ai acquis des notions tout aussi importantes en traitement de la parole et compris des algorithmes intéressants.

De par la forme du modèle, proche d'une modélisation physique, je pense que cette approche de la transformation a de l'avenir. En ne modifiant qu'une partie de la production de la parole, le résultat parait plus facilement naturel et les algorithmes de modifications plus évidents à trouver. Reste peut être à améliorer le modèle LF pour le rendre plus souple et à séparer la production du bruit de celle de la glotte, pour pouvoir ne modifier que la forme du filtre du conduit vocal.

Souhaitant travailler dans la voix, je suis certain que ces connaissances vont s'avérer utiles à l'avenir.

Bibliographie

[Rabiner] L. R. Rabiner/R. W. Schafer “Digital Processing of Speech Signals”, 1978

[Yin] Alain de Cheveigné, “Yin, a fondamental frequency estimator for speech and music”, 2002

[Liljencrants Fant] Gunnar Fant, Johan Liljencrants et Qi guang Lin, “A four-parameter model of glottal flow” STL-QPSR, vol.26, no. 4, pp. 1-13, 1985.

[Imai Abe] S. Imai et Y. Abe “Spectral enveloppe extraction by improved cepstral method” Electron and Commun vol. 62-A, no 4, 1979

[Robel & Rodet] A. Röbel et X. Rodet, “Efficient spectral enveloppe estimation and its application to pitch shifting and envelope preservation”, Dafx, 2005

[Degottex Rd] G. Degottex, A. Roebel, X. Rodet “Shape parameter estimate for a glottal model without time position”, Ircam, 2008.

[Villavicencio] Villavicencio, Röbel, Rodet, “Improving LPC spectral enveloppe extraction of voiced speech by true-envelope estimation”, Ircam, 2006

[Smits Yegnanarayana] R. Smits et B. Yegnanarayana, “Determination of instants of significant excitation in speech using group delay function” IEEE Trans. Speech and Audio Processing vol.3 pp 325-333. 1995

[Depalle Poirrot 1991] P. Depalle et G. Poirrot, “SVP : a modular system for analysis, processing and synthésis of sound signals” Proceedings of the International Computer Music Conference, 1991

[N. Alessandro 2008] N. d’alessandro, T. Drugman, T. Dubuisson, “Transvoice Table”, 2008.

[Peeters 2001] ”Modéles et modifications du signal sonore adaptés à ses caractéristiques locales » Thése de doctorat de l’Université Paris 6, 2001.

Annexe 1: code de la dsp de glottisresyn~

```
// methode appelée à chaque dsp
void GlottisResyn::CbSignal()
{
    // pointeur vers l'entrée audio
    float * ins = InSig(0);
    // pointeur vers la sortie audio
    float * out = OutSig(0);
    //vector size, taille du signal d'une dsp
    int n = Blocksize();
    // taux d'échantillonnage
    int sr = Samplerate();
    // nombres d'échantillons correspondant à une période
    float NT0sample = sr/m_staticf0;

    // Nombre d'appel de dsp à attendre avant de lancer le calcul de la synthése
    // (approximativement 2 périodes)
    int NBlockSizeToWait = floor(2.1*NT0sample/n) + 1;

    // On enregistre le signal d'entrée
    m_RingBuffer->RecordInput(ins, n);
    // On lit le buffer dans lequel on a mis la synthése en sortie
    m_RingBuffer->ReadOutput(out, n);

    // incrément du nombre de dsp appelé depuis dernier calcul
    m_TimeElapsed += 1;
    // une dsp avant le lancement du calcul de la synthése, nous calculons
    // les spectres glottiques estimé et de synthése (imposé)
    // Ces calculs sont couteux en temps de calcul. Les dissocier du reste
    // allége beaucoup le taux de CPU utilisé
    if(m_TimeElapsed == NBlockSizeToWait - 1)
    {
            GetGlottalSpectrums(sr, m_staticf0, m_Rd, m_RRd, m_G, m_RG);
    }
    // Les deux périodes sont écoulées, nous lançons le calcul de la synthése
    if(m_TimeElapsed >= NBlockSizeToWait)
    {
            float offset;
            // m_LastDelay est le dernier délai que nous avons appliqué à notre signal
            // par rotation de la phase du spectre

            // Si ce délai est positif, nous calculons un nouveau délai négatif
            if(m_LastDelay > 0)
            {
                  offset = NT0sample - floor(NT0sample + 1);
                  NT0sample = floor(NT0sample + 1);
```

```
		}
		// sinon nous calculons un délai positif
		else
		{
			offset = NT0sample - floor(NT0sample);
			NT0sample = floor(NT0sample);
		}
		// conteneur des impulsions de signal resynthétisé
		// C'est une matrice, chaque ligne contient une période de signal
		MatMTLMatrix<SREAL> sigs;
		// nous figeons les arguments pour ne pas su'ils soient modifiés durant le
		   calcul
		float f0 = m_staticf0; // fréquence fondamentale
		float Rd = m_Rd;     // Rd du signal d'entrée
		float RRd = m_RRd;         // Rd de synthése
		float Harmo = m_staticHarmo; // coefficient de périodicité

		// Calcul du nombre de période que l'on peut mettre dans le buffer
		int NbpulsInBuf = m_RingBuffer->GetNT0Free(NT0sample, f0, sr, n);
		if(NbpulsInBuf < 1)
		{
			m_TimeElapsed = 0;
			return;
		}
		// En l'absence du descripteur de vuf, nous le mettons à 4000
		double vuf = 4000;
		// On récupére le buffer contenant les échantillons entrants
		MatMTLVector<SREAL> Rbuffer = m_RingBuffer->GetBuffer();

		// Si signal voisé
		if(Harmo>0.55)
		{
			// On synthétise le signal, on rempli sigs du nombre de période
disponible dans le buffer
			Rd2sig(sr, f0, Rd, RRd, m_Sigma, vuf, offset, m_LastDelay, NbpulsInBuf,
Rbuffer, sigs, m_G, m_RG);
			//puis on rempli le buffer de synthése
			m_RingBuffer->AddSigToBuffer(sigs, NT0sample);
		}
		else
		{
			// sinon on rempli le buffer avec le signal d'origine
			int VectSize = NbpulsInBuf*NT0sample;
			BruitResynth(sr, f0, vuf, Rbuffer, sigs, VectSize);
			m_RingBuffer->AddSigToBufferUncut(sigs);
		}
		// On calcul le délai final appliqué
		//(Nouveau retard = Dernier retard + Nombre de période synthétisées* délai
		  calculé)
		m_LastDelay = m_LastDelay + NbpulsInBuf * offset;
```

```
            // On refige les nouveaux coeffs, on réinitialise le compteur de dsp, on
               efface les vecteurs
            m_staticf0 = m_f0;
            m_staticHarmo = m_Harmo;
            m_TimeElapsed = 0;
            m_G.clear();
            m_RG.clear();
    }
}
```

Annexe 2, algorithme de la true enveloppe

L'approche utilisée est un lissage cepstral. Cette méthode est basée sur une représentation dans le domaine de Fourier du log de l'amplitude du spectre du signal. Le cepstre réel (C(l)) d'un signal est la transformée inverse du log de l'amplitude spectrale d'un son. Si nous définissons X(k) pour représenter la DFT tu signal x(n), le cepstre réel du signal est:

$$C(l) = \sum_{k=0}^{K} \log(|X(k)|) e^{i\frac{2\pi kl}{K}}$$

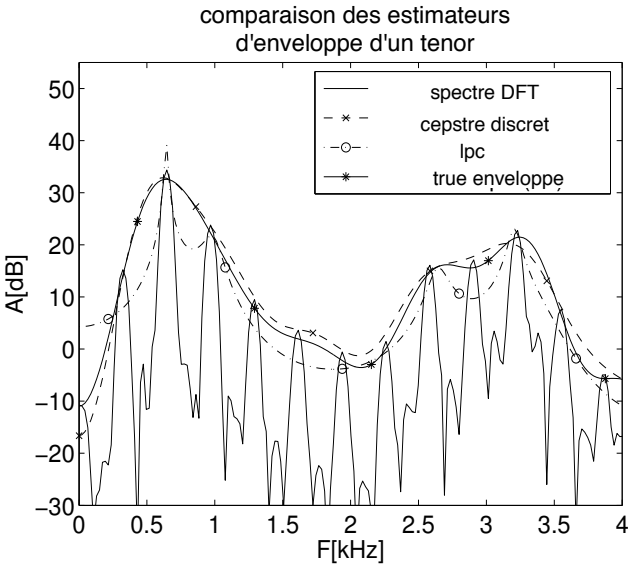
Parceque l'enveloppe spectrale est considérée comme étant une version lissée de l'amplitude du spectre, un moyen simple pour obtenir une estimation de l'enveloppe spectrale est de mettre toutes les quefrences d'ordre élevé à 0. Malheureusement, le cepstre filtré va créer une enveloppe qui va suivre la moyenne du spectre et donc pas le contour des pics spectraux. C'est pour palier à ce problème qu'a été développée la *true enveloppe.* Son algorithme est itératif et une implémentation robuste demande des calculs supplémentaires, notamment si la taille de fft est grande. Soit $V_i(k)$ la représentation cepstrale de l'enveloppe spectrale l'itération i, c'est à dire la transformée de Fourier du cepstre liftré. Nous initialisons les données comme tel:

$$A_0(k) = \log(|X(k)|) \text{ et } V_0(k) = -\infty$$

L'algorithme remplace alors itérativement la valeur de l'amplitude su spectre par la fonction suivante:

$$A_i(k) = \max(A_{i-1}(k), V_{i-1}(k))$$

Et nous appliquons itérativement le filtre cepstral au nouveau spectre A_i. Avec cette procédure, les creux entre les pics du spectre vont se "remplir" par le filtre cepstral et l'enveloppe estimée va progressivement augmenter jusqu'à ce que les pics soient couverts. Comme critère d'arrêt de la procédure le paramètre Δ est utilisé pour définir le dépassement autorisé du pic du spectre observé au dessus de l'enveloppe estimée.



L'algorithme original était très coûteux en temps de calcul (entre 5 et 8 fois plus long que la LPC) et certains calculs étaient simplifiables. Axel Röbel et Xavier Rodet l'ont simplifié de manière à obtenir un temps de calcul proche de celui des LPC.