Rapport de BE
Reconnaissance automatique de la parole

Bataillou Grégory
Perrin Victor

12 janvier 2017

Table des matières

1 Lecture des fichiers audio et affichage de forme d'onde **3**
1.1 lecture . . . 3
1.2 Affichage des formes d'onde . . . 3

2 Calcul des coefficients LPC **4**
2.1 La fonction calcul_lpc.m . . . 5
2.1.1 Déclaration/calcul des variables initiales . . . 5
2.1.2 Boucle sur les fenêtres temporelles . . . 5
2.1.3 Calcul de la matrice de covariance . . . 6
2.1.4 Détermination des coefficients LPC . . . 6
2.2 Insertion dans le programme principal . . . 7

3 Calcul et affichage de la matrice des distances entre les signaux audio **7**
3.1 Théorie sur le recalage de deux signaux et le calcul de la distance élastique . . . 7
3.2 Code dans le programme principal . . . 7
3.3 la fonction distance_elastique.m . . . 8
3.3.1 initialisation des variables . . . 8
3.3.2 Initialisation/cas particuliers de la matrice g . . . 8
3.3.3 Cas général . . . 9
3.3.4 retour de la fonction . . . 10
3.4 Affichage des distances . . . 10

4 Classification **10**
4.1 Algorithme de classification . . . 10
4.2 Classifier les signaux au préalable . . . 11
4.3 Codage de l'algorithme . . . 12
4.3.1 Tri des distances . . . 12
4.3.2 Récupération de la classe majoritaire . . . 12
4.3.3 Mise à jour du nombre de bonnes estimations . . . 13

5 Améliorations réalisées **13**
5.1 Troncage du signal utile et normalisation de la forme d'onde . . . 13
5.1.1 La fonction rogner . . . 14
5.1.2 Normalisation de la forme d'onde . . . 15
5.1.3 Insertion dans le programme principal . . . 15
5.2 Fenêtre de Hamming . . . 16

6 Résultats obtenus **17**
6.1 Résultats sur notre série 17
6.2 Tests avec une autre série 17
6.3 Limites du programme 18

1 Lecture des fichiers audio et affichage de forme d'onde

Cette section a pour objectif de présenter le code qui permet de lire des fichiers audio et d'afficher les signaux.

1.1 lecture

Pour lire les fichiers audio, on liste tout d'abord les fichiers au format .wav se trouvant dans le répertoire 'oui-non/' . On créé alors une boucle for pour récupérer à chaque nouvelle itération le nom du fichier audio suivant dans un vecteur 'nomFichier', et on récupère le signal numérique grâce à la fonction wavread

```
clear
close all;
rep = 'oui-non/';
nomFichier = {};
nbFichiers = length(flist);
%Listing des fichiers au format .wav
flist = dir(strcat(rep,'*.wav'));

for i = 1:length(flist)
        n = flist(i).name;
        nomFichier{length(nomFichier) + 1} = n;
        [s Fe nbits] = wavread(strcat(rep,n));
        s = s(:,1);
        ...
end
```

1.2 Affichage des formes d'onde

Dans notre boucle de lecture, rajoutons cette ligne permettant de récupérer tous les signaux dans une seule matrice ss.

```
ss{length(nomFichier)} = s;
```

Il nous suffit alors de générer une figure après la boucle de lecture et récupération des signaux

```
%Calcul du nombre de fenêtres à afficher par ligne (5 max)
    et par colonne
nLi     = floor(nbFichiers/5)+ (mod(nbFichiers,5)>0);
```

```
nCol     = 5;
%affichage des signaux
for l = 1:length(flist)
        subplot(nLi,nCol,l);
        plot(ss{l});
        title(['Signal : ',nomFichier{l}]);
end
```

On obtient la figure 1.

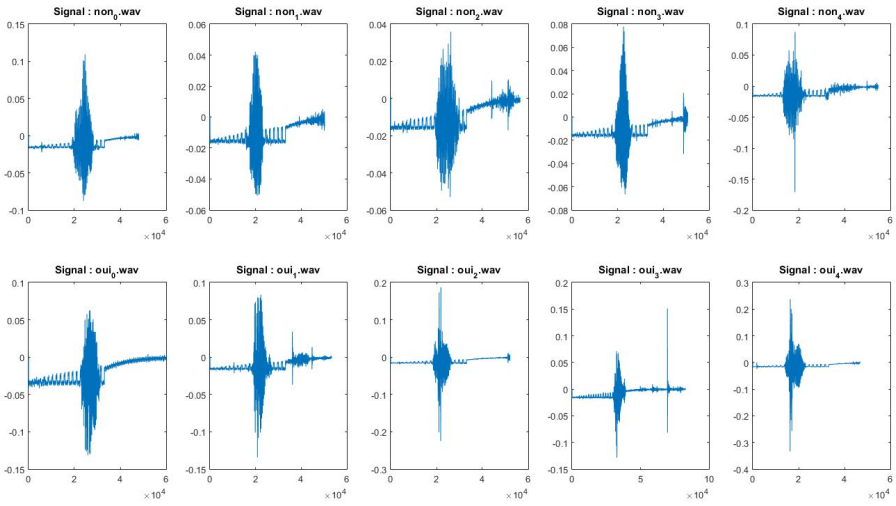

FIGURE 1 – Affichage des formes d'onde des signaux

2 Calcul des coefficients LPC

Dans cette section, nous sommes au coeur de l'algorithme. Nous expliciterons les principales étapes de cet algorithme.

Rappelons à toute fin utile que le principe premier est de définir un échantillon du signal $x_n = x(n)$ comme une combinaison linéaire des p échantillons précédents :

$$x_n = x(n) = a_1 * x(n-1) + a_2 * x(n-2) + ...a_p * x(n-p) = \sum_{i=1}^{p} a_i * x(n-i) \quad (1)$$

L'objectif alors est de connaître $a_1, a_2...a_p$, qu'on appelle coefficients LPC (p est l'ordre du signal)

2.1 La fonction calcul_lpc.m

L'objectif de la fonction calcul_lpc est, comme précisé dans les commentaires en préambule, de renvoyer une série de coefficients LPC, correspondant à chaque fenêtre temporelle du signal que l'on décale chaque fois que l'on a fini de calculer les coefficients LPC du signal en question dans cette fenêtre. On peut à notre guise entrer en paramètre de la fonction l'ordre des coefficients LPC 'N', le recouvrement des fenêtres temporelles 'rec' :

```
1 function lpc = calcul_lpc(s, Fe, N, rec)
```

2.1.1 Déclaration/calcul des variables initiales

Ici nous avons besoin :

- Du nombre d'échantillons du signal s : Ns
- Du nombre d'échantillons par fenêtre de 20 ms : $Nechantillon = floor(0.02 * F_e)$
- Du nombre d'échantillons dont on se décale pour prendre une nouvelle fenêtre : $offset = floor(Nechantillon * rec)$
- Du nombre de fenêtres $Nfenetre = floor(Ns/offset) - floor(1/rec)$: (- les dernières pour éviter les dépassements d'index)
- D'initialiser la matrice des coeffcients LPC : $lpc = zeros(N, Nfenetre)$
- D'initialiser notre vecteur qui sera multiplié par la matrice de covariance : $vecteurInv = zeros(N + 1, 1)$ et $vecteurInv(1) = 1$

2.1.2 Boucle sur les fenêtres temporelles

Notre programme réalise les mêmes opérations sur chaque fenêtre temporelle. Nous créons donc une boucle pour automatiser ces opérations :

```
9  for fen = 1: Nfenetre % Pour chaque fenetre
10 debEchantillon = (fen - 1) * offset;
11 end
```

Nous nous plaçons maintenant à l'intérieur de la boucle. les indices d'échantillons débutent donc au début de la fenêtre temporelle en question.

2.1.3 Calcul de la matrice de covariance

Pour obtenir les valeurs de la matrice de covariance, nous avons d'abord calculé la matrice S_{mat} définie ci-dessous :

$$S_{mat} = \begin{bmatrix} s(1) & s(2) & \dots & s(n-1) & s(n) \\ s(2) & s(3) & \dots & s(n) & 0 \\ \vdots & \vdots & \cdots & 0 & 0 \\ s(n) & 0 & \dots & 0 & 0 \end{bmatrix} \tag{2}$$

Cette matrice nous permet d'obtenir tous les coefficients de la matrice de covariance grâce à une simple multiplication :

$$\begin{aligned} s * S_{mat} &= \begin{bmatrix} s(1) & s(2) & \dots & s(n-1) & s(n) \end{bmatrix} * S_{mat} \\ &= \begin{bmatrix} \sum_{i=1}^{n} s(i)^2 & \sum_{i=1}^{n-1} s(i) * s(i+1) & \dots & \sum_{i=1}^{n-k} s(i) * s(i+k) & \dots & s(n)s(1) \end{bmatrix} \\ &= \begin{bmatrix} R(0) & R(1) & \dots & R(k) & \dots & R(n) \end{bmatrix} * n \\ &= R_i * n \end{aligned}$$

On obtient la matrice de covariance R à partir du vecteur R_i en ajoutant ce code :

```
%Creation de la matrice R
R = zeros(N+1);
for diago = 0:N
        R = R + diag(Ri(diago+1).*ones(N + 1 - diago, 1),
            diago );
        if diago > 0
                R = R + diag(Ri(diago+1).*ones(N + 1 -
                    diago, 1), - diago);
        end
end
```

2.1.4 Détermination des coefficients LPC

Une fois la matrice de covariance déterminée, le principale est fait : Il suffit maintenant de l'inverser et de multiplier cette inverse au vecteur $\begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix}$ initialisé au début. en divisant notre résultat par σ^2, soit le premier terme du vecteur obtenu, on tombe sur les coefficients LPC :

```
lpci = R\vecteurInv;
lpci = lpci./(lpci(1));
lpci = lpci(2:end);
```

On ajoute cette série de coefficients LPC à notre future matrice de retour de la fonction et on recommence avec la fenêtre suivante !

```
38 lpc(:,fen) = lpci;
```

2.2 Insertion dans le programme principal

Dans notre programme principal, on définit au début des constantes qui définiront les valeurs des paramètres à rentrer dans la fonction calcul_lpc puis on l'annonce dans la boucle principale.

```
11 CONST.ORDERLPC = 40;
12 CONST.RECLPC = 1/3;
13 ...

58 ss{length(nomFichier)} = s;
59 %Calcul des coefficients lpc sur le signal utile
60 lpc = calcul_lpc(s,Fe,CONST.ORDERLPC,CONST.RECLPC);
61 %ajout d'une matrice de coefficients lpc dans la matric de
       matrice lpcs
62 lpcs{length(nomFichier)} = lpc;
```

3 Calcul et affichage de la matrice des distances entre les signaux audio

3.1 Théorie sur le recalage de deux signaux et le calcul de la distance élastique

Afin de classer les signaux dans l'une ou l'autre des 2 classes, il est nécessaire de calculer une distance entre un signal et un autre. Cependant, comme les signaux ne sont pas de même taille, il faut utiliser une technique de recalage des signaux basés sur un algorithme de programmation dynamique, réalisé dans notre cas par la fonction distance_élastique.m . Cette fonction repose sur l'algorithme d'Alignement Temporel Dynamique (DTW).

3.2 Code dans le programme principal

Le code ci-dessous est inséré dans le programme principal. Cela permet d'obtenir une matrice NxN (N étant le nombre de fichiers étudiés) contenant les distances élastiques entre signaux. La fonction distance_elastique est explicitée plus tard.

```
for i = 1: length(lpcs)
    for j = 1: length(lpcs)
        %Calcul de la distance élastique entre les
            coefficients lpcs
        distanceElast(i,j) = distance_elastique(lpcs{i},
            lpcs{j});
    end
end
```

3.3 la fonction distance_elastique.m

Ainsi, la fonction distance_elastique est chargée de calculer la distance. prend en paramètre 2 vecteurs, chacun correspondant aux coefficients LPC d'un signal en particulier, et retourne un nombre caractéristique qui quantifie la différence entre les signaux.

```
function D = distance_elastique(u1, u2)
```

3.3.1 initialisation des variables

```
%Coefficients pondérateurs
wi = 1;
wd = 1;
ws = 1;

[N, L1] = size(u1); %L1 fenêtres temporelles, ordre LPC : N
[N, L2] = size(u2);

d = zeros(L1, L2); % future matrice de produits scalaires
g = zeros(L1, L2); %future matrice du chemin parcouru
```

3.3.2 Initialisation/cas particuliers de la matrice g

Cette partie du code concerne l'initialisation de la récurrence sur la matrice g qui se construit de proche en proche, avec la première ligne et première colonne de la matrice devant être programmée spécialement. la distance d, elle, est toujours calculée par le produit scalaire des deux vecteurs en question, soit le produit matriciel de la transposée de la différence des deux vecteurs avec cette même différence.

```
%calcul de g(1,1)
d(1,1) = sqrt(transpose(u1(:,1)-u2(:,1))*(u1(:,1)-u2(:,1)))
    ;
```

```
14 g(1,1) = d(1,1);
15
16 %On fait la premiere ligne
17 for j = 2: L2
18     d(1,j) = sqrt(transpose(u1(:,1)-u2(:,j))*(u1(:,1)-u2(:,
           j)));
19     g(1,j) = g(1,j-1) + wi * d(1,j);
20 end
21
22 %On fait la premiere colonne
23 for i = 2:L1
24     d(i,1) = sqrt(transpose(u1(:,i)-u2(:,1))*(u1(:,i)-u2
           (:,1)));
25     g(i,1) = g(i - 1,1) + wd * d(i,1);
26 end
```

3.3.3 Cas général

Le code ici reprend l'algorithme de création de la matrice g, à savoir :

$$g(i,j) = min \begin{cases} g(i-1,j) & +\omega_v * d(i,j) \\ g(i-1,j-1) & +\omega_d * d(i,j) \\ g(i,j-1) & +\omega_h * d(i,j) \end{cases} \tag{3}$$

```
27 %Cas général
28 cas = zeros(1,3);
29 for i = 2: L1
30     for j = 2:L2
31         d(i,j) = sqrt(transpose(u1(:,i)-u2(:,j))*(u1(:,i)-
               u2(:,j)));
32
33         cas(1) = abs(g(i,j-1) + wi * d(i,j));
34         cas(2) = abs(g(i - 1,j) + wd * d(i,j));
35         cas(3) = abs(g(i - 1,j - 1) + ws * d(i,j));
36
37         g(i,j) = min(cas);
38     end
39 end
```

3.3.4 retour de la fonction

Il suffit alors de renvoyer $D = \frac{g(L1,L2)}{L1+L2}$:

```
40 D = g(L1, L2)/(L1 + L2);
```

3.4 Affichage des distances

Comme pour l'affichage des figures (sous-section 1.2), nous avons pu afficher les distances élastiques relatives à chaque élément, via le code ci-dessous inséré dans le programme principal, afin d'avoir un rendu visuel de ce que nous venions de calculer.

```
1 for k = 1:nbFichiers
2         subplot(5,5,k);
3         bar(distanceElast(:,k));
4         title(['Distance de ', nomFichier{k}]);
5 end
```

On obtient alors la figure 2.

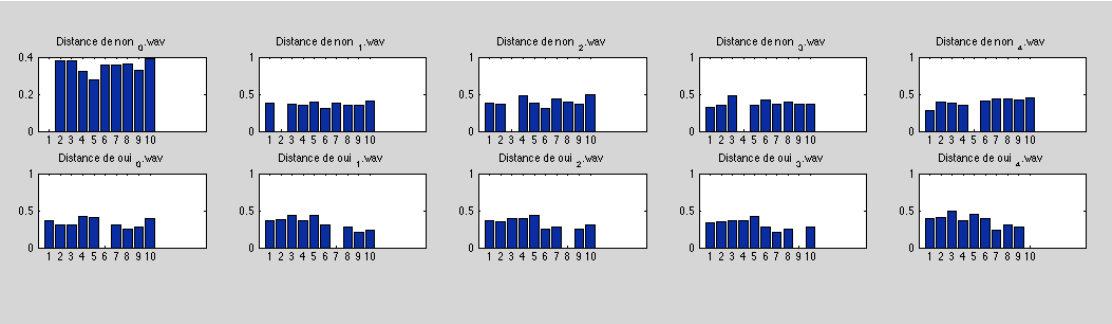

FIGURE 2 – Figure récapitulant les distances entre les signaux

4 Classification

4.1 Algorithme de classification

l'algorithme ci-dessous montre le principe de la classification que nous avons utilisé selon la méthode des k-éléments plus proches (avec comme exemple k = 5). Dans notre cas, nous ferons attention à bien utiliser un k impair pour éviter les égalités du nombre d'éléments dans chaque classe.

Input: $D(i,:)$: L'ensemble de distances élastiques entre le signal considéré i et les autres signaux

$classePrediteSgn \leftarrow 0$;
$nombreComparaisons \leftarrow 5$;
for $k \leftarrow 1$ **to** $nombreComparaisons$ **do**
  if *Classe du k-ieme element plus proche vaut 'oui'* **then**
    $classePrediteSgn \leftarrow classePrediteSgn + 1$
  else
    $classePrediteSgn \leftarrow classePrediteSgn - 1$
  end
end
if $classePrediteSgn > 0$ **then**
  $classeEltEstimee \leftarrow'$ *oui*$'$
else
  $classeEltEstimee \leftarrow'$ *non*$'$
end
return $classeEltEstimee$;

Algorithm 1: Classification d'un signal

4.2 Classifier les signaux au préalable

Nous devons, avant de pouvoir comparer la classe de notre signal, faire manuellement la classification des signaux. Pour cela, on insère une ligne de code dans notre première boucle for (celle du calcul LPC). On remarquera que pour cela, il a fallu écrire une fonction contains.m qui renvoie vrai si le premier argument contient le second, faux sinon.

```
for i = 1:nbFichiers
    n = flist(i).name;
    nomFichier{length(nomFichier) + 1} = n;
    %Si le nom du fichier contient 'non', la classe est '
       non' et
    %pareil avec 'oui'
    if (contains(n, 'non'))
        classOuiNon{length(nomFichier)} = 'non';
    else if (contains(n, 'oui'))
            classOuiNon{length(nomFichier)} = 'oui';
        else classOuiNon{length(nomFichier)} = 'und'; %
           ligne de debug
        end
    end
```

```
13      [s Fe nbits] = wavread(strcat(rep,n));
14      ...
15  end
```

4.3 Codage de l'algorithme

Tout d'abord, au début du programme, nous anonçons le nombre de comparaisons dont nous aurons besoin pour établir une classification :

```
13  CONST.KELEMENTS = 5;
```

Ensuite, nous nous plaçons dans la boucle du calcul de distance élastique par ligne, après ledit calcul :

```
90  for i = 1: nbFichiers
91      for j = 1: nbFichiers
92          distanceElast(i,j) = distance_elastique(lpcs{i},
                lpcs{j});
93      end
94      [on se place ici]
```

4.3.1 Tri des distances

Nous avons besoin de l'index des k plus petites distances de l'élément i :

```
94  %Tri des distances
95  [C I] = sort(distanceElast(i,:));
96  indexTri(i,:) = I;
```

4.3.2 Récupération de la classe majoritaire

```
97  %Classification des signaux
98  classPredite = 0;
99  for k = 2:CONST.KELEMENTS+1
100         if (classOuiNon{indexTri(i,k)} == 'oui')
101                 classPredite = classPredite + 1;
102                 else    if (classOuiNon{indexTri(i,k)} == '
                        non')
103                                 classPredite = classPredite
                                        - 1;
104                 end
105         end
```

```
106 end
107
108         %Mémorisation de la prédiction
109         if classPredite > 0
110                 classOuiNonEstimee{length(
                        classOuiNonEstimee) + 1 } = 'oui';
111                 else     if classPredite < 0
112                         classOuiNonEstimee{length(
                                classOuiNonEstimee) + 1 } = 'non
                                ';
113                 else classOuiNonEstimee{length(
                        classOuiNonEstimee) + 1 } = 'und';
114                         end
115         end
```

4.3.3 Mise à jour du nombre de bonnes estimations

```
116 %Comparaison avec la vérité
117 if classOuiNonEstimee{end} == classOuiNon{i}
118         estimCorrecte = estimCorrecte + 1;
119 end
```

5 Améliorations réalisées

Les résultats que nous obtenons en premier lieu ne sont pas fameux. En effet, beaucoup de parasites viennent interférer sur les performances. Parmi ces parasites, on pourra citer :

— Les effets de bords lorsque l'on fenêtre notre signal
— Le bruit
— Les biais, qui décalent la moyenne du signal

Pour diminuer au plus ces influences sur le calcul de nos coefficients LPC, nous avons mis en place plusieurs techniques.

5.1 Troncage du signal utile et normalisation de la forme d'onde

En affichant les signaux (voir fig. 1), on remarque la qualité des signaux est très médiocre, et que leurs amplitudes ne sont pas normalisées. cela altère énormément les résultats. L'objectif est donc de tronquer les signaux pour ne conserver que la partie utile, puis de normaliser l'amplitude des différents signaux. On créé alors, dans un premier temps, la fonction rogner, permettant de tronquer la partie utile.

5.1.1 La fonction rogner

Nous sommes partis des principes suivants :

— Le signal est très brouillé pour les premiers et derniers échantillons, il ne faut pas les prendre en compte.

— Un signal devient intéressant dès que son énergie dépasse un certain seuil, qu'on choisira empiriquement.

La fonction rogner balaye un signal représentatif de l'énergie du signal, du début (un peu décalé) jusqu'à arriver à l'endroit où le seuil est dépassé. cet endroit est mis en mémoire. Ensuite, le signal est balayé en partant de la fin (encore une fois un peu décalée), jusqu'à dépasser un certain seuil, ou bien que ce balayage ne se rapproche pas trop de l'endroit mis en mémoire précédemment. L'endroit trouvé est de nouveau mis en mémoire.

Les deux endroits mis en mémoire correspondent alors respectivement au début et à la fin signal tronqué.

```
1  function sModif = rogner(s)
2  %Définition du rapport limite
3  CONST.SEUIL = 0.03; % Constante seuil choisie empiriquement
4  CONST.SINUTILE = 10000; % nb d'échantillons à partir d'où
       faire le balayage
5  CONST.LONGSINMIN = 15000;% nb d'échantillons du signal
       tronqué au min
6  %On calcule le signal Carré du signal moins sa moyenne,
       récupère le max
7  sMoy = s - mean(s);
8  sCarre = sMoy.*sMoy;
9  maxSCarre = max(sCarre);
10
11 %initialisation des variables d'échantillon du début et de
       la fin du signal
12 %utile
13 debut = CONST.SINUTILE;
14 fin = length(s) - CONST.SINUTILE;
15 debutNonTrouve = 1; finNonTrouvee = 1;
16
17 % On crée une boucle simulant le balayage du signal
18 while(debutNonTrouve)
19     %Tant que le signal au carre ne dépasse pas un certain
           seuil, on
20     %continue de balayer le signal
21         if (sCarre(debut)/maxSCarre) < CONST.SEUIL
```

```
22                      debut = debut + 1;
23              else
24                      debutNonTrouve = 0;
25              end
26  end
27
28  while(finNonTrouvee)
29              if (((sCarre(fin)/maxSCarre) < CONST.SEUIL) ||((fin
                    - debut) > CONST.LONGSINMIN))
30                      fin = fin - 1;
31              else
32              finNonTrouvee = 0;
33              end
34  end
35  %Retourner le signal rogné
36  sModif = s(debut:fin);
```

Les résultats sur la figure 3 montrent la performance du rognage.

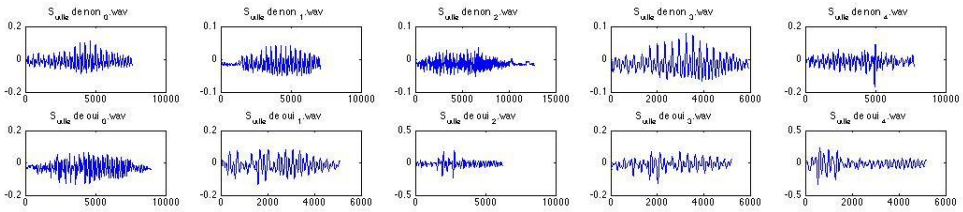

FIGURE 3 – Signaux rognés avec la fonction rogner.m

5.1.2 Normalisation de la forme d'onde

Puis, on divise notre signal utile obtenu par l'amplitude maximale du signal en question :

```
1  sUtile = sUtile/(max(abs(sUtile)));
```

5.1.3 Insertion dans le programme principal

Il nous suffit alors de mettre en paramètre de la fonction calcul_lpc non pas les signaux originaux mais leur troncage :

```
...
s = s(:,1);
%Troncage du signal pour ne conserver que la partie utile
sUtile = rogner(s);
%Normalisation de la forme d'onde
sUtile = sUtile/(max(abs(sUtile)));
%Mise en mémoire du signal tronqué
ss{length(nomFichier)} = sUtile;
%Calcul des coefficients lpc sur le signal utile
lpc = calcul_lpc(sUtile,Fe);
...
```

5.2 Fenêtre de Hamming

Une amélioration supplémentaire consiste à diminuer fortement les effets de bords dû au fenêtrage lors du calcul des coefficients LPC en effectuant le produit simple des frames temporelles par une fenêtre de Hamming qui permet de limiter le bruit numérique. Cette fenêtre prenant le nombre de points que l?on définit grâce à la fonction Hamming préexistante de Matlab (cf.fig. 4).

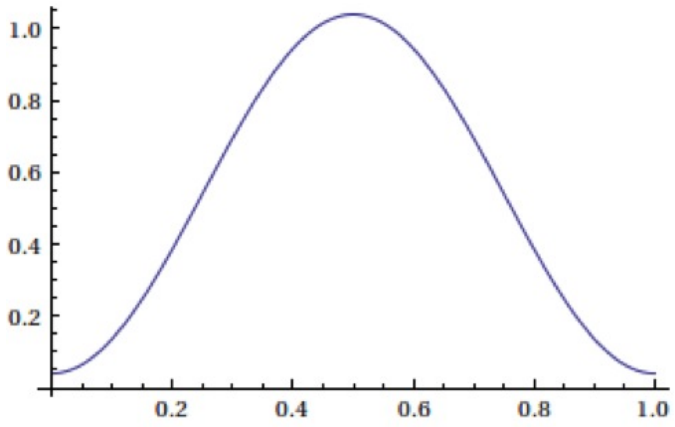

FIGURE 4 – Exemple de fenêtre de Hamming sous matlab

6 Résultats obtenus

6.1 Résultats sur notre série

Après améliorations, nos résultats deviennent très performants : pour une classification à 5 éléments plus proches voisins, un ordre de LPC de 40, et un recouvrement de un tiers, on obtient un taux de 10 /10 éléments bien classifiés (cf. fig. 5) ! !

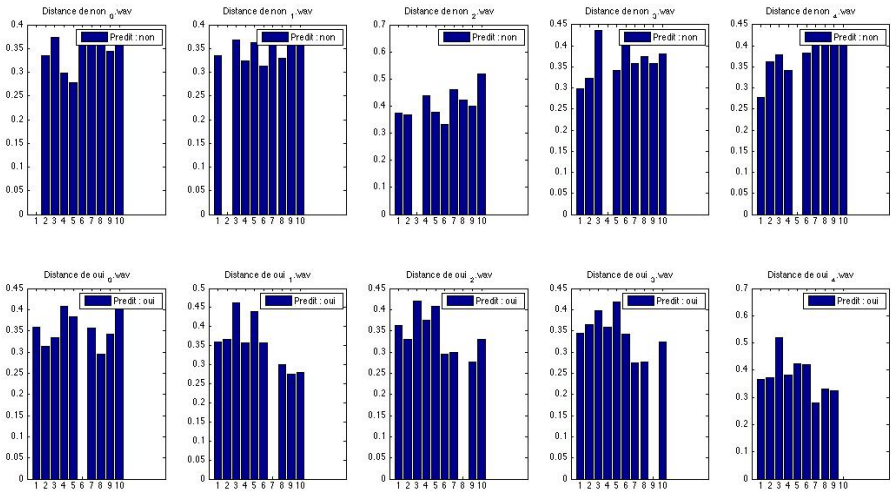

FIGURE 5 – Résultats obtenus sur la liste donnée

6.2 Tests avec une autre série

Nous avons voulu tester la robustesse de nos résultats et avons enregistré une série de oui/non avec notre voix à partir d'un dictaphone (les fichiers se nomment sous la forme 'oui_test3.wav' et sont dans le dossier 'oui-non-tests', pour les essayer, il faut les déplacer dans le dossier 'oui-non'). Malheureusement, les résultats sont décevants : la distance élastique entre nos signaux test et la série de signaux déjà donnés est extrêmement importante relativement à la distance élastique entre 2 signaux. Pourtant, la fréquence d'échantillonnage est la même, et le nombre de bits de codage est aussi identique.

6.3 Limites du programme

Ainsi, il ne faut pas se réjouir trop vite : Pour la série en question, les résultats étaient performants, mais dès que l'on change la prise du son, la personne parlant, ou même l'ordre du LPC, on tombe rapidement sur des résultats non satisfaisants. D'autre part, il faut mentionner que notre programme n'utilise que 2 mots, oui ou non, et en forçant la classification, si un programme n'est pas performant, le signal a toujours environ une chance sur 2 de se retrouver dans la bonne classe. Pour tester une robustesse d'un tel programme, mieux vaudrait réaliser au moins 5 classes (et donc 5 mots différents), et avoir plus d'échantillons.

Conclusion

Nous avons pu mettre en pratique la théorie de la distance élastique entre deux séries de coefficients LPC. Cela nous a mené à réaliser un petit programme dans lequel est implantée une classification des signaux par une méthode connue. Nous avons poussé un peu le programme plus loin en testant nos propres voix, sans succès. Le calcul des coefficients LPC marche pour une série d'enregistrements dans les mêmes conditions, mais devient alors très peu robuste dès qu'on modifie ses conditions. Il faut donc trouver des modifications pour normaliser au mieux le signal traité afin de pouvoir mieux les comparer. Cela dit, en admettant une application pour la reconnaissance vocale lié à un seul micro et à une seule personne (comme par exemple la reconnaissance de la parole d'un téléphone), il semble qu'après une période d'apprentissage, l'algorithme soit robuste.

Dans un esprit d'ouverture, on peut imaginer améliorer notre programme par les possibilités suivantes :

- améliorer la fonction rogner
- trouver des critères supplémentaires pour normaliser les signaux pour pouvoir mieux évaluer la distance élastique en cas de changements de conditions d'enregistrements.
- rajouter des phonèmes et donc des classes
- Passer sur un autre modèle acoustique : les modèles de Markov cachés. Ces modèles sont d'autant plus performants qu'ils peuvent apprendre, décoder ou évaluer en fonction de probabilités. Cela rend les choses plus complexes, il faudrait d'abord utiliser un modèle du premier ordre (l'état actuel ne dépend que de l'état précédent), puis l'améliorer en augmentant l'ordre.
- Instituer un modèle de langage afin de pouvoir interpréter des phrases !