Complément sur le fonctionnement du codage de Huffman

G. Dauphin

1. Codage sans perte : l'algorithme de Huffman

Le codage de Huffman est un algorithme de compression de données sans perte élaboré par David Albert Huffman et publié en 1952. Le code est déterminé à partir d'une estimation des probabilités d'apparition des symboles de source, un code court étant associé aux symboles de source les plus fréquents. Le principe du codage de Huffman repose sur la création d'un arbre binaire, c'est-à-dire un arbre où chaque nœud a deux fils, le fils de gauche est étiqueté par un zéro et le fils de droite est étiqueté par un 1. Chaque terminaison représente un des symboles à coder. Le code associé au symbole est une succession de 0 et de 1, la succession qui correspond au parcours depuis la racine jusqu'à la terminaison correspondant au symbole.

On cherche ici à simuler sous Matlab le fonctionnement du codage de Huffman pour une séquence de symboles dans un alphabet à 256 symboles. Les étapes 1 à 4 correspondent au codage d'une séquence (constitution d'un arbre et transformation de cet arbre en une table de conversion des symboles en code). Les étapes 5 et 6 correspondent au décodage (fabrication d'un arbre à partir d'une table de conversion et transformation d'une succession de codes en une succession de symboles).

Etape 1 : calcul de la fréquence des symboles

A partir de la séquence de symboles, on cherche à déterminer le nombre d'occurrences de chaque symbole. En effet plus un symbole est fréquent, plus il est préférable d'utiliser pour ce symbole un code court.

L'algorithme proposé consiste d'abord à parcourir l'ensemble des symboles. Pour chacun de ces symboles on forme un vecteur dont la taille est égale à celle de la séquence et dont les composantes sont égales à 1 lorsqu'à cet indice la composante de la séquence est égale au symbole et égales à 0 sinon. La fréquence du symbole est égale à la somme de ce vecteur, on stocke cette fréquence dans un autre vecteur dont la taille est égale à la liste des symboles. Cet algorithme est implémenté dans *symbole2frequence.m*

Etape 2 : création d'un arbre binaire correspondant au code de Huffman

Affecter à chaque symbole un code composé de 0 et de 1, c'est en fait dessiner un arbre binaire dont les terminaisons sont les symboles et dont les liens sont de deux types, gauche pour 0 et droite pour 1. L'algorithme de Huffman donne une méthode pour construire cet arbre en assurant que les terminaisons sont d'autant plus éloignées de la racine que la fréquence du symbole associé à cette terminaison est d'autant plus faible.

L'implémentation proposée ici requière d'une part une liste des symboles, ici les chiffres de 0 à 255 et une liste des fréquences de chacun de ces symboles. L'implémentation fournit à la fin un arbre binaire. Cet algorithme est implémenté dans *frequence2arbre.m*

On ne considère d'abord que les symboles ayant au moins une occurrence dans la séquence. Puis on ordonne ces symboles et leurs fréquences du moins fréquent dans la séquence au plus fréquent. On met cette liste ordonnée des symboles sous la forme d'une liste d'arbres, ces arbres étant réduits à un seul des symboles.

A chaque itération, on choisit les deux arbres dans la liste des arbres ayant la fréquence la plus faible (donc a priori les deux premiers arbres de la liste des arbres). On réunit ces deux arbres en un seul. Pour affecter à ce nouvel arbre une fréquence égale à la fréquence des deux arbres constitutifs, on remplace aussi les deux premiers éléments du vecteur des fréquences par un élément égal à la somme des fréquences. Ainsi à chaque itération, il y a un arbre en moins dans la liste des arbres et de même une fréquence en moins dans la liste des fréquences. On réordonne la liste des arbres et la liste des fréquences de la plus petite fréquence à la plus grande fréquence.

On répète cette itération jusqu'à ce que la liste des arbres soit en fait composée d'un seul arbre.

Instructions Matlab à tester pour réaliser le programme :
Simulation d'une liste de fréquences pour un alphabet composé de 6 symboles.
>>frequence=floor(20*rand(1,6))
Une liste des symboles
>>liste=1 :6
Suppression des symboles ayant une fréquence nulle
>>indice=find(frequence~=0)
Modification de la liste
>>listeModifiee=liste(indice)
Ordonnancement de la liste des fréquences
>>sort(frequence)
Ordonnancement commun de la liste des symboles et de la liste des fréquences en fonction de la liste des fréquences
>>[frequenceTrie,nouvelOrdre]=sort(frequence)
>>listeTrie=listeModifiee(nouvelOrdre)
Transformation d'une liste en ensemble d'arbres composés des éléments de la liste
>>listeArbres=num2cell(liste)
Récupération d'un arbre dans la liste
>>unArbre=listeArbres{2}
Fusion de deux arbres en un arbre
>>unAutreArbre=3, nouvelArbre=[{unArbre}, {unAutreArbre}],
Affichage d'un arbre
>>celldisp(nouvelArbre)
Nombre de fils de la racine d'un arbre
>>length(listeArbres)

Etape 3 : Constitution d'une table de conversion à partir d'un arbre

L'algorithme consiste à parcourir l'ensemble de l'arbre, de concaténer au fur et à mesure du parcours un vecteur code du symbole '0' si le lien de gauche est utilisé et du symbole '1' si le lien de droite est utilisé.

L'algorithme proposé utilisée requière un arbre binaire et produit une table dont chaque case est une structure contenant d'une part la valeur et d'autre part le code binaire associé. Il est implémenté dans *arbre2table.m* L'implémentation classique d'un parcours d'arbre consiste à utiliser une implémentation récursive, parce qu'alors on bénéficie de l'utilisation de piles qui sont programmées par le langage de programmation. L'implémentation proposée ici consiste à refabriquer ces piles sous la forme d'une liste de tâches à effectuer, liste qui s'incrémente à chaque fois qu'on rencontre un nouveau nœud et qui est décrémentée à la fin de la réalisation de la tâche. Pour chaque tâche à effectuer, il est nécessaire aussi de sauvegarder les codes déjà rencontrés, on utilise donc aussi une liste des codes qui est incrémenté et décrémenté de la même façon que la liste des tâches.

Une tâche à effectuer consiste à vérifier l'arbre est en fait une terminaison, si c'est le cas on place le symbole lu dans la terminaison dans une nouvelle case de la table et le code associé à la tâche dans le champ code de cette nouvelle case de la table. Si l'arbre n'est pas une terminaison, on crée deux nouvelles tâches avec deux nouveaux codes. Les nouvelles tâches sont définies par les deux arbres fils et les codes sont la concaténation du code précédent et de 0 pour le premier fils et 1 pour le deuxième fils.

Ces tâches sont à répéter jusqu'à ce que la liste des tâches soit vide.
Instructions Matlab à tester pour réaliser le programme :
Exemple d'arbre
>>arbre={{1,2},{3,4}}
Exemple d'arbre vide
>>arbreVide={[]}
Déclaration d'une table de structures
>>table=[] ;

Incrémentation d'une table de structures à deux champs
>>table(end+1).champ1=1 ;
>>table(end).champ2=[2 3] ;

Etape 4 : Conversion d'une séquence de symboles en une succession de code

La table de conversion permet de trouver le code associé à chaque symbole, il suffit donc de concaténer tous ces codes pour avoir le code associé à la séquence. L'implémentation proposée consiste d'abord à créer une table d'indexation qui à chaque symbole associe la case correspondante dans la table de conversion. Ensuite il suffit de parcourir la séquence et de concaténer les codes trouvés. Cet algorithme est implémenté dans *sequence2code.m*

Etape 5 : Fabrication d'un arbre binaire à partir de la table de conversion

Dans la séquence codée il est nécessaire aussi de rajouter l'information soit sur l'arbre binaire soit sur la table de conversion, mais de fait c'est plus simple de mettre la table de conversion que l'arbre binaire. Lors de l'étape 6, le décodage de la séquence codée se fait avec l'arbre binaire. Aussi il est nécessaire d'avoir un algorithme capable de transformer une table de conversion en arbre binaire.

L'implémentation classique consiste à parcourir toutes les cases de la table de conversion et pour chaque case à parcourir le code en créant si besoin l'arbre au fur et à mesure et en y affectant la valeur associée à la case. MatLab n'est pas un langage adapté à cela dans la mesure où il n'y a pas de passage par référence (ou de pointeurs). En revanche c'est un langage interprété et non un langage compilé, ce qui permet de créer l'arbre en fabriquant pour chaque code, la chaîne de caractère qui lorsqu'elle est évaluée, crée si nécessaire la partie de l'arbre manquante et affecte la valeur à la case souhaitée. L'implémentation proposée consiste à réaliser cette tâche pour chaque case du tableau. Cet algorithme est implémenté dans *table2arbre.m*

Instructions MatLab à tester pour réaliser le programme :
Création d'un arbre vide
>>arbre={}
Transformation de l'arbre pour contenir le symbole 3 pour le code 001
>>arbre{1}{1}{2}=3
Transformation du code en instruction à interpréter
>>code=[0,0,1],
>>prefixe='arbre'
>>suffixe=repmat('{x}',1,length(code))
>>suffixe(2)=num2str(code(1)+1)
>>suffixe(5)=num2str(code(2)+1)
>>suffixe(8)=num2str(code(3)+1)
>>instruction=[prefixe, suffixe, '=3;']
Evaluation de l'instruction
>>eval(instruction)

Etape 6 : décodage d'une séquence codée par l'intermédiaire d'un arbre

L'algorithme requière la séquence à décoder et un arbre binaire représentant la signification du code. L'algorithme produit une séquence de symbole. L'algorithme consiste à parcourir en parallèle la séquence à décoder et l'arbre binaire en procédant en plusieurs itérations. Au début de l'itération on se place à la racine de l'arbre. On se déplace dans l'arbre en prenant le lien de gauche ou de droite suivant que le code lu est 0 ou 1. L'itération prend fin lorsqu'on arrive à une terminaison de l'arbre. Le symbole sur cette terminaison est placé dans le vecteur de la séquence de symboles. L'itération est répétée jusqu'à atteindre la fin de la séquence à décoder. Cet algorithme est implémenté dans *sequenceCodee2sequenceDecodee.m*

Lien avec l'entropie

Question : Choisissez une image en niveau de gris synthétique et une image synthétique (avec peu de niveaux de gris). Transformez chacune de ces images en un vecteur contenant l'ensemble des niveaux de gris à valeurs dans l'ensemble 0 à 255. Réalisez la compression sans perte, puis la décompression sans perte de ces deux images. Evaluez la place mémoire nécessaire pour stocker chacune de ces images et commentez en calculant avec l'entropie.

2. Codage à l'aide d'une transformation bloc par bloc

On ne considère ici que des images en niveaux de gris. Il s'agit de quatre méthodes de codage par bloc. Dans tous les cas la première étape consiste à découper l'image par bloc puis sur chacun de ces blocs à faire un traitement particulier sur le bloc pour coder l'image. C'est ce traitement particulier qui simplifie l'image et éventuellement diminue la taille de l'image ou les valeurs que peut prendre les coefficients. Ce traitement introduit aussi une dégradation de l'image. Il faut ensuite appliquer un codage sans perte sur l'image transformée en une séquence de nombre à valeurs dans un ensemble de taille finie. Le décodage s'obtient en appliquant le codage sans perte inverse, en reformant l'image globale éventuellement après avoir appliqué une transformation affine et en appliquant le traitement inverse sur chaque bloc.

Instructions MatLab à tester pour réaliser le programme :
>>im2col
>>col2im
La fonction MatLab blkproc permet de travailler bloc par bloc, elle ne requière pas que le résultat du traitement forme un bloc de même taille que le bloc de départ. Par contre blkproc ne fonctionne pas sur des images couleurs.
>>moyenneParBloc=blkproc(im,[8 8],@mean2);

2.1 Codage par troncature de blocs (BTC)

Chaque bloc de l'image est représenté par une avec deux niveaux de quantifications et deux paramètres. Le seuillage des pixels se fait en fonction de la moyenne des niveaux de gris des pixels du bloc

$$g'_{ij} = \begin{cases} 1 & si \quad g_{ij} < m \\ 0 & si \quad g_{ij} \geq m \end{cases}$$

où m est la moyenne du bloc, g_{ij} sont les pixels du bloc et g'_{ij} sont les pixels du bloc après quantification. Les deux paramètres peuvent être A la valeur moyenne des pixels tels que $g'_{ij} = 0$ et B la valeurs moyenne des pixels tels que $g'_{ij} = 1$, ou alors m et σ où σ est l'écart-type du bloc. La reconstruction du bloc se fait dans le premier cas en remplaçant les pixels tels que $g'_{ij} = 0$ par A pour la première méthode ou par $m - \sigma\sqrt{\frac{1-\alpha}{\alpha}}$ pour la deuxième méthode, et en remplaçant les autres pixels par B pour la première méthode ou par $m - \sigma\sqrt{\frac{\alpha}{1-\alpha}}$. α désigne la fraction de pixels tels que $g'_{ij} = 0$ au sein du bloc. Le codage avec la première méthode est implémenté dans *BTC1.m* et le décodage avec la première méthode est implémenté dans *iBTC1.m*

Instructions MatLab à tester pour réaliser le programme :
>>seuillageBloc=inline('x>mean2(x)','x') ;
>> imSeuil=blkproc(im,[8 8],seuillageBloc);

Question : Choisissez une image en niveaux de gris. Réduisez cette image de façon à ce que la hauteur et la largeur de cette image soient des multiples de 8. Appliquez le codage et le décodage à cette image avec les deux méthodes. Décrivez les dégradations observées.

2.2 Codage par troncature des coefficients de la DCT

C'est l'algorithme implémenté dans la compression JPEG. L'algorithme consiste à appliquer la DCT sur chaque bloc de 8x8 puis à quantifier les coefficients obtenu suivant la table d'allocation qui détermine pour chaque bloc le nombre de niveaux de quantifications (voir le polycopié de cours pour un exemple de table d'allocation). Dans l'implémentation proposée on conserve aussi les valeurs maximales et minimales des coefficients de la dct de l'ensemble des blocs de l'image.

```
m=[   16 11 10 16 24 40  51  61
      12 12 14 19 26 58  60  55
      14 13 16 24 40 57  69  56
      14 17 22 29 51 87  80  62
      18 22 37 56 68 109 103 77
      24 35 55 64 81 104 113 92
      49 64 78 87 103 121 120 101
      72 92 95 98 112 100 103 99]*qualite


ordre=[1 9 2 3 10 17 25 18 11 4 5 12 19 26 33
      41 34 27 20 13 6 7 14 21 28 35 42 49 57 50
      43 36 29 22 15 8 16 23 30 37 44 51 58 59 52
      45 38 31 24 32 39 46 53 60 61 54 47 40 48 55
      62 63 56 64] ;
```

Annexe

Forum MatLab :

```
function frequence=symbole2frequence(sequence)
% sequence est supposée être un vecteur ligne composé de chiffres entre 0 et 255, chiffres
%appelés symboles. Ce programme calcule le nombre d'occurrence d'un symbole dans
%sequence. frequence est un vecteur de taille 1 256 et dont les composantes d'indice k
%indiquent le nombre d'occurrence du symbole k-1 dans sequence.
  frequence=zeros(1,256) ;
  for k=1 :255
    frequence(k)=sum(sequence(:)==k-1) ;
  end;

function arbre=frequence2arbre(frequence)
%frequence est un vecteur de taille 1 256 contenant le nombre d'occurences d'un des symbols
%0 à 255 dans sequence. arbre est un arbre binaire contenant à ses terminaisons les symboles,
%il est construit en fonction de l'algorithme de Huffman.
  symboles=find(frequence~=0);
  frequence=frequence(symboles);
  [frequence,indexTrie]=sort(frequence);
  symbolesTries=symboles(indexTrie);
  arbre=num2cell(symbolesTries);
  while (length(arbre)>2)
    arbre=[{[arbre(1),arbre(2)]}, arbre(3:end)];
    frequence=[frequence(1)+frequence(2) frequence(3:end)];
```

```
    [frequence,index]=sort(frequence);
    arbre=arbre(index);
  end;

function table=arbre2table(arbre)
%arbre est un arbre binaire contenant à ses terminaisons les symboles et pour chaque symbole
%la liste des liens utilisés pour aller de la racine à la terminaison est le code associé au %symbole. La table contient autant de cases qu'il n'y a de symboles dans l'arbre et pour
%chaque case il y a un champ indiquant la valeur et un autre champ indiquant le code associé.
  listeTaches.arbre={arbre};
  listeTaches.code={[]};
  table=[];
  while (length(listeTaches.arbre)>0)
    arbre=listeTaches.arbre{1}; listeTaches.arbre=listeTaches.arbre(2:end);
    code=listeTaches.code{1}; listeTaches.code=listeTaches.code(2:end);
    if (iscell(arbre{1}))
      listeTaches.arbre=[listeTaches.arbre, {arbre{1}}];
      listeTaches.code=[listeTaches.code, {[code,0]}];
    else
      table(end+1).valeur=arbre{1};
      table(end).code=[code,0];
    end;
    if (iscell(arbre{2}))
      listeTaches.arbre=[listeTaches.arbre, {arbre{2}}];
      listeTaches.code=[listeTaches.code, {[code,1]}];
    else
      table(end+1).valeur=arbre{2};
      table(end).code=[code,1];
    end;
end;

function sequenceCode=sequence2code(sequence,table)
%sequence est une succession de symboles à valeurs dans 0..255 table contient autant de
%cases que de symboles et pour chaque case, il y a un champ indiquant le numéro du
%symbole et un autre champ indiquant le code.
  valeur2index=[] ;
  for k=1:length(table)
    valeur2index(table(k).valeur)=k;
  end;
  sequenceCode=[];
  for l=1:length(sequence)
    sequenceCode=[sequenceCode table(valeur2index(sequence(l)+1)).code];
  end ;

function arbre=table2arbre(table)
%table contient autant de cases que de symboles et pour chaque case, il y a un champ
 %indiquant le numéro du symbole et un autre champ indiquant le code. arbre est l'arbre
%binaire dont les terminaisons sont les symboles indiqués dans la table et le chemin pour aller
%de la racine aux terminaisons est le code indiqué dans table.

arbre={};
for k=1:length(table)
  code=table(k).code;
  suffixe=repmat('{x}',1,length(code));
```

```
  for l=1:length(code)
    suffixe(2+3*(l-1))=num2str(1+code(l));
  end;
  eval(['arbre',suffixe,'=',num2str(table(k).valeur),';']);
end;

function sequenceDecodee=sequenceCodee2sequenceDecodee(sequenceCodee,arbre)
%sequenceCodee est un vecteur ligne composé de 0 et de 1, avec une particularité qui est
%qu'il est la concaténation de codes. arbre est un arbre binaire dont l'ensemble des codes
%dont sequenceCodee est la concaténation, ces codes permettant d'aller de la racine à une
 %terminaison. sequenceDecodee est une succession de symbole à valeurs dans l'ensemble 0
 %à 255.
  sequenceDecodee=[] ;
  while(~isempty(sequenceCodee))
    parcoursArbre=arbre;
    while (iscell(parcoursArbre))
      parcoursArbre=parcoursArbre{1+sequenceCodee (1)};
      sequenceCodee=sequenceCodee(2:end);
    end;
    sequenceDecodee=[sequenceDecodee parcoursArbre-1];
  end;

function [imSeuil,moyenneA,moyenneB]=BTC1(im)
%im doit être une image en niveau de gris dont la hauteur et la largeur sont des multiples de 8
%imSeuil est une matrice de même taille que im qui vaut 1 pour les pixels supérieurs à la %moyenne des
pixels de leur blocs et 0 sinon. moyenneA est la moyenne dans chaque bloc %des pixels pour lesquels
imSeuil vaut 0. moyenneB est la moyenne dans chaque bloc des %pixels pour lesquels imSeuil vaut 1.
  seuillageBloc=inline('x>mean2(x)','x');
  imSeuil=blkproc(im,[8 8],seuillageBloc);
  sommeMatrice=inline('sum(sum(x))','x');
  nombrePixelA=blkproc(imSeuil,[8 8],sommeMatrice);
  moyenneA=blkproc(im.*imSeuil,[8 8],sommeMatrice)./nombrePixelA;
  moyenneB=blkproc(im.*(1-imSeuil),[8 8],sommeMatrice)./(64-nombrePixelA);

function imDecodee=iBTC1(imSeuil,moyenneA,moyenneB)
%imSeuil est une matrice avec des 1 et des 0, moyenneA et moyenneB sont des matrices 8 %fois plus
petites à la fois en hauteur et en largeur. imDecodee est une matrice de même taille
%que imSeuil.
  agrandissement=inline('x*ones(8)','x');
  imDecodee=imSeuil.*blkproc(moyenneA,[1 1],agrandissement);
  imDecodee=imDecodee+(1-imSeuil).*blkproc(moyenneB,[1 1],agrandissement);
```