

# DIIC - LSI, 3<sup>e</sup> année

—

## Module CN

### TP 1 - Codeur d'image prédictif :

Gaël Sourimant, Luce Morin

17 Novembre 2006

## 1 Introduction

L'objet de ce travaux pratique est de se familiariser avec un codeur/décodeur d'image simple, le MICD (*Modulation par impulsion et codage différentiel*). Dans le souci de se concentrer sur les aspects du cours, des ressources ci-dessous sont votre à disposition dans les sous-répertoires de `/share/diic3/lsi_tpcmv`. Copiez-les dans votre répertoire de travail. Créez également un sous-répertoire `output`.

- Images de tests au format PGM `/share/diic3/lsi_tpcmv/images/`  
Nous utiliserons les images `bjork`, `sati` et `cornouaille`.
  
- La librairie `libit`  
Pensez à exécuter les lignes suivantes pour pouvoir utiliser la librairie et les pages de manuels qui s'y rapportent, respectivement.  

```
export PATH=$PATH:/share/diic3/local/bin
export MANPATH=$MANPATH:/share/diic3/local/man
```

*Ces variables ne sont valables que dans la console où elles sont déclarées.*
  
- Un squelette de programme `/share/diic3/lsi_tpcmv/tp1/tp1.c`  
Dans l'état actuel, ce programme quantifie grossièrement l'image entrée en paramètre dans le fichier `tp1.param` en ne gardant que les bits les plus significatifs. Elle reconstruit alors et écrit l'image des différences.
  
- Le fichier de paramètre `/share/diic3/lsi_tpcmv/tp1/tp1.param`  
Ce fichier permet de passer des paramètres au programme. Vous pouvez ainsi modifier simplement les paramètres sans recompiler le code.
  
- Le fichier de compilation `/share/diic3/lsi_tpcmv/tp1/Makefile`  
Un appel de ce fichier par la commande `make` compilera votre `tp` sans que vous ayez besoin de vous soucier des différents paramètres de compilation.

Ce travail donnera lieu à un compte-rendu qui sera noté. Ce compte-rendu est à rendre au plus tard pour le 24 novembre. Les éventuels retards seront pris en compte dans la note. Vous pouvez rendre votre travail sous l'une des formes suivantes :

**Compte-rendu au format papier** avec en-tête.

**Compte-rendu sous forme électronique** : *obligatoirement* au format PDF ou OpenOffice (pas ce Word). Ce compte-rendu peut être envoyé à tout moment (avant la limite fatidique) à l'adresse `gael.sourimant@irisa.fr`. Le nom du fichier doit inclure les noms des deux binômes (`nom1_nom2.pdf`).

Ce compte-rendu inclura d'une part les réponses aux questions que vous aurez traitées, d'autre part des commentaires sur les algorithmes et les résultats de simulation. Le code source devra également être inclu *dans le compte-rendu* en annexe. La qualité du code (justesse, clarté, commentaires) sera prise en compte, mais de manière marginale : ce n'est pas un TP de C. Il est important que vous preniez soin de montrer une réflexion par rapport aux questions posées. En effet, c'est la pertinence de vos réponses sera jugée et non la quantité. Ainsi, une observation uniquement factuelle d'un résultat sans aucune interprétation n'est pas considérée comme une réponse valable.

## 2 Présentation du codeur MICD/MICDA

Le codeur MICD (en Anglais DPCM pour *Differential Pulse Code Modulation*) est un système simple, appartenant à la famille des codeurs prédictifs. Il en existe deux versions :

- MICD : MIC Différentiel,
  - MICDA : MICD Adaptatif (celui auquel nous nous intéressons dépend du contenu des données image).
- Les diagrammes en blocs du codeur et du décodeur sont présentés sur la figure 1, avec les notations suivantes :

- $X(i, j)$  : signal d'entrée
- $X_p(i, j)$  : prédiction du signal d'entrée
- $E(i, j)$  : erreur de prédiction
- $\hat{E}(i, j)$  : erreur de prédiction quantifiée
- $\hat{X}(i, j)$  : signal codé - décodé (signal reconstruit qui sert à la prédiction de l'étape suivante)

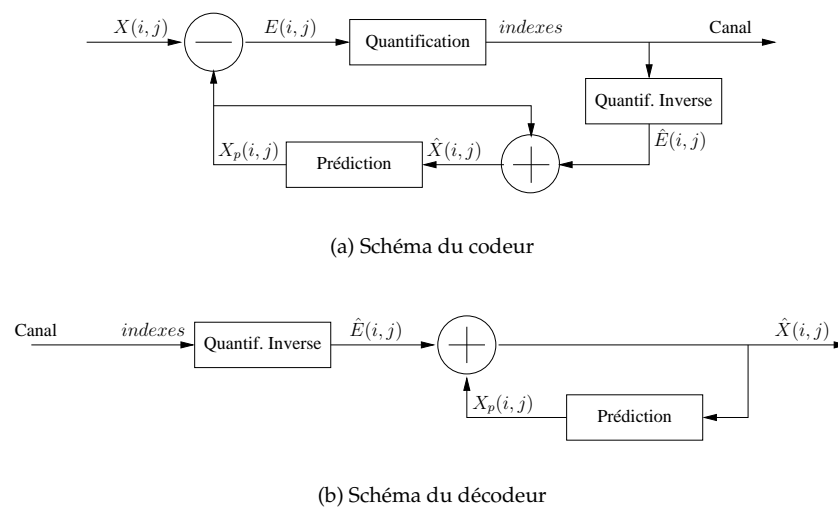


FIG. 1 – Diagrammes du schéma de codage / décodage

La motivation de ces codeurs vient du fait que les pixels voisins (spatialement) ont une intensité lumineuse voisine (numériquement). Dans le schéma de codage, il s'agit de quantifier la différence  $E(i, j)$  entre le signal  $X(i, j)$  et une prédiction  $X_p(i, j)$  de ce signal. Cette prédiction est réalisée à partir des valeurs "passées" de  $\hat{E}$  et de  $X_p$ . Si l'on note  $Q$  le bruit de quantification, on a :

$$\hat{E}(i, j) = E(i, j) + Q(i, j)$$

En suivant la chaîne de codage, il vient :

$$\begin{aligned} \hat{X}(i, j) &= X_p(i, j) + \hat{E}(i, j) \\ &= X_p(i, j) + E(i, j) + Q(i, j) \\ &= X(i, j) + Q(i, j) \end{aligned}$$

Aux erreurs de quantification près, les intensités reconstruites sont donc égales aux intensités initiales.

## 2.1 Prédiction

Les pixels sont des valeurs d'intensité lumineuse et sont ici comprises entre 0 et 255. Pour les deux codeurs (MICD et MICDA), les pixels voisins du pixel à coder (noté  $X$ ) sont notés comme suit :

ligne précédente	B	C	D
ligne courante	A	X	

Vous allez étudier trois modes de prédiction : deux prédictions fixes pour le MICD et une prédiction adaptative pour le MICDA.

Prédictions MICD :

$$\begin{aligned} P(X) &= A \quad (\text{mono-dimensionnelle}), \\ P(X) &= \frac{A+C}{2} \quad (\text{bi-dimensionnelle}). \end{aligned}$$

Prédiction MICDA :

$$P(X) = \begin{cases} A & \text{si } |C - B| < |A - B| \\ C & \text{sinon} \end{cases}$$

Conditions aux bords :

Pour la prédiction mono-dimensionnelle, vous prendrez  $P(X) = 0$  sur la première colonne.

Pour les deux autres prédictions, vous prendrez également  $P(X) = 0$  sur la première colonne et  $P(X) = A$  sur la première ligne.

## 2.2 Quantification

Le quantificateur qui vous est proposé par l'intermédiaire des deux fonctions `q_index` et `q_reconst` est un quantificateur fixe à 16 niveaux de reconstruction (i.e. 4 bits sont nécessaires à la transmission de l'erreur de prédiction quantifiée, si un code à longueur fixe est utilisé). Contrairement à l'histogramme de niveau de gris qui est très variable d'une image à l'autre, l'histogramme d'erreur de prédiction peut être modélisé par une gaussienne généralisée (gaussienne, laplacienne, etc.) de moyenne nulle.

Les valeurs de seuil de décision suivantes ont été extraits de tables :

$$\{ -255 \quad -113 \quad -89 \quad -66 \quad -46 \quad -28 \quad -13 \quad -4 \quad 2 \quad 5 \quad 14 \quad 29 \quad 47 \quad 67 \quad 90 \quad 114 \quad 255 \}$$

Les niveaux de reconstruction correspondants sont les suivants :

$$\{ -126 \quad -101 \quad -79 \quad -55 \quad -37 \quad -19 \quad -7 \quad -2 \quad 3 \quad 8 \quad 20 \quad 38 \quad 56 \quad 78 \quad 102 \quad 127 \}$$

### 3 Travail à effectuer : codage / décodage

Dans la suite, vous travaillerez sur les images `bjork.pgm`, `sati.pgm` et `cornouaille.pgm`. Pour ce TP et le suivant, vous allez utiliser la librairie `libit`<sup>1</sup>. N'hésitez pas à consulter les pages de manuel ou la documentation fournie à l'url <http://libit.sourceforge.net/doc>.

#### 3.1 Présentation de librairie `libit`

Lors des TP, vous allez être amené à utiliser la librairie `libit`. Elle est écrite en C et permet de manipuler des objets et structures couramment utilisés en codage. Elle permet également de manipuler des images, qui sont représentées comme des matrices (ou des tableaux à deux dimensions, selon l'usage qui en est fait). Vous trouverez une documentation succincte à l'url <http://libit.sourceforge.net/doc>. Parcourez éventuellement les sections 2.2 (Vectors) à 3.4 (Parser) de cette documentation. Une liste des fonctions qui vous intéresseront directement est donnée en annexe. Cette librairie est installée dans le répertoire `/share/diic3/local`. Afin de pouvoir l'utiliser, suivez les étapes suivantes.

1. Ajouter le répertoire `/share/diic3/local/bin` dans votre variable d'environnement `PATH`.  
`export PATH=$PATH:/share/diic3/local/bin`
2. Ajouter le répertoire `/share/diic3/local/man` dans la variable d'environnement `MANPATH` (si vous voulez la doc sous forme de man pages ; pas nécessaire sinon).  
`export MANPATH=$MANPATH:/share/diic3/local/man`

#### 3.2 Travail à effectuer

**Question 1.** Lisez le source du programme `tp1`. Une image est lue puis est codée/décodée (avec respectivement les fonctions `dpcm_encode` et `dpcm_decode`). En l'état actuel, l'encodage consiste à ne garder que les bits les plus significatifs de l'image, puis à reconstruire celle-ci. Combien de bits significatifs sont conservés ?

Dans le but de fournir une mesure objective de la distorsion introduite par la compression, on utilisera le rapport signal sur bruit (PSNR, *Peak Signal to Noise Ratio*).

Soient respectivement  $I_o$  et  $I_r$  les images originale et reconstruite, de même taille  $M \times N$ . Le calcul du PSNR fait intervenir le calcul de l'erreur quadratique moyenne, EQM (ou *MSE* pour *Mean Square Error*) :

$$EQM(I_o, I_r) = \frac{1}{M \times N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I_o(i, j) - I_r(i, j)]^2$$

où  $I(i, j)$  représente l'intensité lumineuse au pixel de coordonnées  $(i, j)$ . Puis :

$$PSNR(I_o, I_r) = 10 \log_{10} \frac{D^2}{EQM(I_o, I_r)}$$

où  $D$  est la dynamique couverte par les variations de l'intensité lumineuse (les pixels des images à traiter sont codés sur 8 bits, donc  $D = 2^8 - 1 = 255$ ). L'unité du PSNR est le décibel (*dB*) : plus celui-ci est important, plus les images comparées sont semblables.

**Question 2.** Modifiez et exécutez le programme `tp1` pour les trois images, afin de modifier le nombre de bits significatifs transmis entre l'encodage et le décodage, et calculez alors le PSNR obtenu entre les images originales et les images ainsi reconstruites. Interprétez les résultats.

---

<sup>1</sup><http://libit.sourceforge.net>

**Question 3.** Utilisez le squelette fourni (tp1.c) pour compléter les fonctions suivantes permettant de programmer l'encodeur et le décodeur (les fonctions de quantification à utiliser sont décrites en annexe ; le code précédent permettant de conserver les bits les plus significatifs n'a plus lieu d'être et doit être supprimé) :

- int dpcm\_predict( imat  $\hat{X}$ , int i, int j, int mode ) (cf sec. 2.1)
- imat dpcm\_encode( imat  $X$ , int mode ) (cf fig. 1(a))
- imat dpcm\_decode( imat  $\hat{E}_I$ , int mode ) (cf fig. 1(b))

La fonction dpcm\_predict implantera les trois types de prédiction (choisis par la variable mode), en respectant les conditions aux bords. La méthode dpcm\_encode utilise dpcm\_predict pour coder l'image en utilisant les fonctions de quantification. Elle renvoie donc les valeurs d'index associées à l'erreur de prédiction quantifiée.

## 4 Résultats, mesures, questions

Dans cette partie, n'hésitez pas à utiliser les fonctions fournies dans le squelette du programme.

**Question 4.** Pourquoi, dans ce schéma, la quantification est-elle insérée dans la boucle, et non pas placée en aval (i.e. : avant l'indexation) ? Appuyez votre argumentation sur un exemple simple.

**Question 5.** Générez l'image codée-décodée  $\hat{X}$ . Les erreurs de codage sont-elles perceptibles sur les trois images de test ?

**Question 6. (Bonus)** Il se peut que vous voyiez apparaître un certain nombre de points blancs sur l'image reconstruite. Pourquoi ? Remédiez au problème.

**Question 7.** Pour les trois modes de prédiction, générez l'image d'erreur de codage  $Q = \hat{X} - X$ . En quels endroits ces erreurs apparaissent-elles le plus ? En quoi est-ce conforme à la construction du codeur ?  
Note : Pour rendre visibles les erreurs, il faut les recentrer sur 128 et les multiplier par un facteur <sup>2</sup>.

**Question 8.** Calculez l'histogramme<sup>3</sup> des erreurs de prédiction  $\hat{E}_I$ , et tracez-le pour les trois images de test. Comparez-les entre eux, puis avec ceux des trois images originales de test. La motivation initiale du codage DPCM vous paraît-elle pertinente ?

**Question 9.** Pour chaque mode de prédiction et chaque erreur, calculez l'entropie de l'image de l'erreur de prédiction :

$$H = - \sum_i p_i \log_2 p_i$$

où  $p_i$  est la probabilité d'apparition d'un pixel d'intensité  $i$  dans l'image<sup>4</sup>. Pourquoi est-il intéressant de connaître cette entropie ? Comparez les entropies des images originales et des images d'erreur de prédiction.

Rappel :  $\lim_{x \rightarrow 0} x \log(x) = 0$ .

**Question 10.** Calculez le PSNR obtenu pour chacun des modes de prédiction et pour les trois images.

**Question 11.** Concluez sur le meilleur mode de codage en interprétant les différents résultats.

⊗ Fin de la partie "recommandée" ⊗

⊗ Début de la partie "Bonus" ⊗

<sup>2</sup>Une fonction vous est fournie pour cela. Voir en annexe.

<sup>3</sup>Une fonction vous est fournie pour cela. Voir en annexe.

<sup>4</sup>Une fonction vous est fournie dans libit pour cela, voir en annexe.

## 5 Analyse des effets des erreurs de transmission

Vous allez maintenant vous intéresser aux conséquences d'une erreur ponctuelle de transmission.

**Question 12.** Introduisez un faible nombre d'erreurs dans images (expliquez comment vous aurez procédé) puis décodez l'image, pour chaque mode de prédiction. Évaluez visuellement le type et l'importance des dégradations occasionnées.

**Question 13.** Introduisez un facteur de fuite  $\alpha$  dans le système de prédiction, afin de réduire l'influence du passé :

$$E = X - (1 - \alpha) \times X_p - \alpha \times 128 \text{ avec } 0 < \alpha < 1$$

Évaluez l'effet de ce facteur de fuite sur les conséquences d'erreurs de transmission (i.e. : refaire les expériences ci-dessus en prenant  $\alpha = 0.1$  puis  $\alpha = 0.05$ ). Concluez sur l'intérêt de ce facteur de fuite afin de pallier les erreurs ponctuelles de transmission.

**Question 14. (Ultra-Bonus)** - En supposant qu'il n'y a pas de quantification, calculez l'effet d'une erreur simple sur les deux modes non adaptatifs. Vous pouvez éventuellement effectuer d'autres hypothèses simplificatrices. Le résultat analytique que vous obtenez se vérifie-t-il en simulation ?

⊗ Fin du TP ⊗

---

## 6 Annexes

### 6.1 Quelques mots sur la librairie *libit*

#### 6.1.1 Types

*Libit* vous permet de définir différents types pour manipuler des vecteurs ou des matrices. Par exemple, une matrice d'entiers sera définie comme étant de type `imat`. De même, un vecteur d'entiers sera de type `ivec`, alors qu'un vecteur contenant des nombres flottants sera de type `vec`. Les différents types que l'on peut utiliser pour les matrices et vecteurs sont :

- `vec` : Vecteur de flottants
- `ivec` : Vecteur d'entiers
- `bvec` : Vecteur de booléens
- `cvec` : Vecteur de nombres complexes
- `mat` : Matrice de flottants
- `imat` : Matrice d'entiers
- `bmat` : Matrice de booléens
- `cmat` : Matrice de nombres complexes

A priori seuls les types contenant des nombres entiers voire flottants sont susceptibles de vous intéresser pour ce tp.

#### 6.1.2 Opérations de bases

Ci-dessous sont rappelées quelques fonctions permettant une manipulation basique de variables matricielles. Les mêmes fonctions sont disponibles pour les types vecteurs, et évidemment plus d'information est disponible à l'adresse <http://libit.sourceforge.net/doc>.

– Création d'une matrice d'entiers de taille 12x8 :

```
imat m = imat_new( 12, 8 );
```

- Accès à l'élément (ligne *i*, colonne *j*) de la matrice *m* :  
`m[i][j] = ... ;`
- Création d'une référence à la matrice *m* :  
`imat m1 = m ;`
- Création d'une copie de la matrice *m* :  
`imat m2 = imat_clone( m ) ;`
- Création d'une matrice de taille 2x3 en initialisant les éléments à 0 :  
`imat m3 = imat_new_zeros( 2, 3 ) ;`
- Création d'une matrice de taille 2x3 en initialisant les éléments à 128 :  
`imat m3 = imat_new_set( 128, 2, 3 ) ;`
- Récupérer le nombre de lignes ou de colonnes d'une matrice :  
`int nb_rows = imat_height( m ) ;`  
`int nb_cols = imat_width( m ) ;`

### 6.1.3 Images

Les images dans ce tp sont des images de type PGM<sup>5</sup>. On les stockera donc dans des matrices d'entiers (*imat*). Les fonctions suivantes sont à votre disposition pour lire une image depuis le disque ou au contraire enregistrer une matrice en tant qu'image sur le disque :

- Lecture de l'image de nom *filename*, que l'on stocke dans la matrice d'entiers *m*.  
`imat m = imat_pgm_read( const char * filename ) ;`
- Ecriture de la matrice *m* dans le fichier de nom *filename*.  
`imat_pgm_write( const char * filename, imat m ) ;`

## 6.2 Fonctions de quantification

Deux fonctions (respectivement d'indexation et de reconstruction) vous sont fournies dans le fichier `tp1.c`.

- Indexation du pixel *val* en utilisant le vecteur *seuils* contenant les seuils de décision.  
`int q_index(int val) ;`
- Reconstruction de l'index *ind* en utilisant le vecteur *reconst* contenant les niveaux de reconstruction.  
`int q_reconst(int ind) ;`

## 6.3 Calcul et dessin d'un histogramme

Dans le squelette du programme, vous trouverez une fonction permettant de calculer et d'enregistrer dans un fichier PGM l'historgramme d'une image.

- Calcul de l'historgramme de l'image *m* et sauvegarde de celui-ci dans le fichier *filename*, qui aura une largeur de 512 pixels et une hauteur de *h* pixels.  
`void imat_pgm_histo( const char * filename, imat m, int h ) ;`

## 6.4 Calcul et dessin d'une image d'erreur

Dans le squelette du programme, vous trouverez une fonction permettant de calculer et d'enregistrer dans un fichier PGM l'image d'erreur entre deux images.

- Calcul de l'erreur (différence) entre les images *m1* et *m2*, et sauvegarde de celle-ci dans le fichier *filename*.  
`void imat_pgm_save_error( const char * filename, imat m1, imat m2, int lambda ) ;`
- Les deux images ayant des valeurs entre 0 et 255, l'erreur a des valeurs entre -255 et 255. Pour que celle-ci soit à valeurs dans 0-255, l'erreur est divisée par deux puis recentrée non plus sur 0 mais sur 128. De plus, de faibles erreurs étant peu visibles, on ajoute à la fonction un paramètre *lambda* qui multiplie cette

<sup>5</sup>Le format PGM (Portable GreyMap) est l'un des formats PNM. Pour une description succincte de format, voir par exemple <http://astronomy.swin.edu.au/~pbourke/dataformats/ppm>

erreur. Après multiplication par  $\lambda$ , si des valeurs se retrouvent hors de la plage 0-255, on les coupe pour correspondre à ces valeurs (une valeur inférieure à 0 vaudra 0 et une valeur supérieure à 255 vaudra 255). Prenez donc soin de ne pas utiliser un  $\lambda$  trop important si vous voulez pouvoir interpréter correctement l'image d'erreur (une valeur de 10 pour  $\lambda$  est en général suffisante).

## 6.5 Calcul de la Pdf d'une image (Probability density function)

Dans la librairie `libit`, une fonction est disponible pour calculer la p.d.f. d'un *vecteur*. Pour calculer la probabilité d'apparition de chaque pixel d'une image, il faut donc d'abord transformer la matrice la représentant en un vecteur. Si on note `pdf` le vecteur de 256 valeurs contenant la probabilité d'apparition de chaque pixel de l'image `m`, on a alors :

```
- ivec tmp = imat_to_ivec( m ); /* transformation de la matrice en un vecteur */  
- vec pdf = histogram_normalized( 256, tmp ); /* calcul de la pdf */  
- ivec_delete( tmp ); /* libération de mémoire */
```

`pdf` est de type `vec` et non `ivec` car il contient des `double`. La valeur `pdf[i]` est donc la probabilité d'apparition dans l'image `m` du pixel de valeur `i`.

## 6.6 Tirer parti du fonctionnement du parser de *libit*

Dans sa forme actuelle, le passage des paramètres au programme `tp1` se fait *par défaut* via le fichier `tp1.param`. Vous pouvez cependant écraser ces paramètres par défaut en les passant à la ligne de commande. Par exemple, si le fichier de paramètres contient la ligne `-mp=1`; et que vous tapez la commande `./tp1 -mp=3`, c'est effectivement le 3<sup>e</sup> mode de prédiction qui sera utilisé.

Vous pouvez donc gagner du temps en lançant le programme avec différents paramètres sans redéfinir le contenu du fichier `tp2.param`. Exemples :

```
- Lancer automatiquement le programme tp1, sur l'image définie dans le fichier tp1.param, pour tous les modes de prédiction :  
  for i in 1 2 3; do ./tp1 -mp=$i; done  
- Lancer automatiquement le programme tp1, sur les 3 images, pour le mode de prédiction défini dans le fichier tp1.param:  
  for n in bjork cornouaille sati; do ./tp1 -i="$n.pgm"; done
```

Vous pouvez même si vous le souhaitez faire une double boucle à la ligne de commande, pour effectuer le codage de toutes les images aux trois modes de prédiction, en une seule fois.