Réception, décodage et affichage d'une image satellite METEOR-M2 - CORRECTION

PARTIE 2 : Décompression JPEG et affichage de l'image

Mots-clés : Images numériques - Compression JPEG.

Niveau concerné :	2ème année BTS Systèmes Numériques (option Informatique et Réseau)
Matière :	Physique
Partie du programme :	- Images numériques.
Capacités exigibles mobilisées :	- Énoncer qu'une image numérique est associée à un tableau de nombres. - Expliquer le principe de la compression d'une image fixe (JPEG).
Outils utilisés :	- Python 3.8.2 https://www.python.org/downloads/ - Jupyter Notebook : Ce TP est facilement réalisable en ouvrant les fichiers "ipynb" avec Jupyter.
Fichiers fournis :	- « viterbi.bin » et « viterbi2.bin » : données brutes décodées par Viterbi - « extract_viterb.py » : fichier contenant des fonctions utiles dans la deuxième partie
Contact :	thomas.lavarenne@ac-creteil.fr
Remerciements :	Cette activité n'aurait pas pu voir le jour sans les travaux passionnants et les explications toujours claires et précises de Jean-Michel Friedt : jmfriedt.free.fr (voir les références [1] et [2] pour approfondir les notions vues ici).

Réception, décodage et affichage d'une image satellite METEOR-M2 - CORRECTION

PARTIE 2 : Décompression JPEG et affichage de l'image

Dans la partie précédente nous avons réceptionné le signal provenant du satellite météorologique Meteor-M2, puis nous avons traité ce signal afin de pouvoir récupérer les données images compressées au format JPEG. L'objectif de cette partie est de décoder plusieurs imagettes de 8x8 pixels et de vérifier que le décodage correspond bien à une partie de l'image complète décodée par le programme "meteor-decoder".

I Extraction des données utiles

La fonction **extract(fichier)** contenue dans **"extract_viterbi.py"** effectue les actions suivantes :

- Ouvre le fichier binaire issue du décodage précédent (viterbi).
- Extrait les octets compris entre deux mots de synchronisation (1acffc1d), les convertit en décimal et place chaque ligne dans un array en forme de tableau.
- Pour chaque ligne, affiche l'endroit à partir duquel commencent les informations à exploiter.

Toutes ces étapes sont visibles dans le fichier "extract_viterbi.py" et ne présentent pas de difficultés particulières. Elles ont été réalisées en suivant les explications de [2] p6-8.

1. **Effectuer les opérations ci-dessus en important le fichier :**

```
>>> import extract_viterbi
```

Si vous souhaitez changer le fichier à traiter, ouvrir le fichier extract_viterbi.py et modifier l'instruction (autour de la ligne 100) :

```
extract_viterbi.extraction("viterbi2.bin")
```

```
>>> import importlib
>>> importlib.reload(extract_viterbi)      # A refaire apres chaque modification du fichier
```

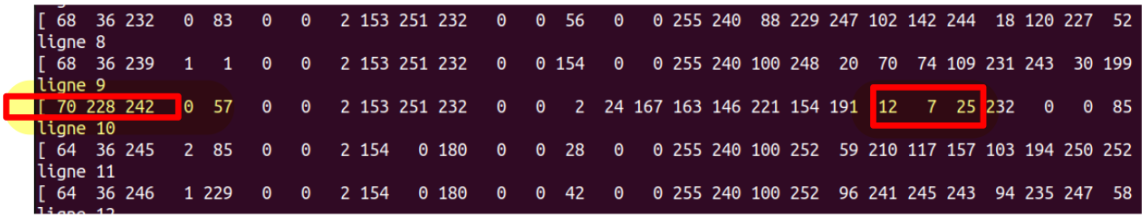
viterbi2.bin

*Le fichier qui correspond aux données de la partie précédente est **"viterbi.bin"**. Le fichier **"viterbi2.bin"** à été obtenu exactement de la même façon mais pour une zone un peu plus haute de l'image. Pour des raisons de reconnaissance de formes plus aisée dans cette partie d'image, nous utiliserons préférentiellement ce fichier dans la suite mais l'étude peut très bien se faire sur le fichier **"viterbi.bin"** (tout comme l'étude précédente peut aussi se faire sur le fichier **"binarymeteor2.s"** qui a servi à générer **"viterbi2.bin"**).*

Quasiment toutes les lignes commencent par la valeur **64, 65 ou 68**. Ce sont les APID (Packet identifier) de l'instrument à bord qui transmet les données (On rappelle qu'il y a bien trois gammes de longueurs d'onde). De temps en temps, une ligne commence par la valeur **70**. Il s'agit d'une information de télémétrie.

2. **Identifier une ligne de télémétrie et trouver l'heure envoyée par le satellite sur les octets 21, 22, 23 pour heure, minutes, secondes (en comptant l'octet 70 comme 0). Est-ce cohérent avec l'information donnée par meteor-decoder :** "Onboard time : 12 :07 :25.928". **Notre décodage est-il validé jusque-là ?**

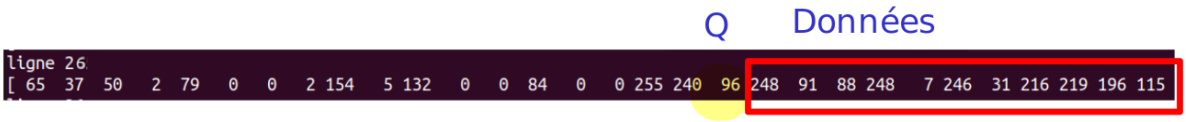
On trouve l'information à la ligne 9 (ou 10). L'image a été reçue à 12h07min25s. C'est donc bien cohérent avec le décodage par le programme meteor-decoder. Le décodage est validé.



Nous allons nous intéresser dans la suite à la **ligne 26**, concernant des données provenant de l'instrument ayant pour APID 65. Sur une telle ligne, l'**octet 18** correspond au **facteur de qualité Q** (qui nous servira plus tard) et les octets suivants concernent les données de l'image envoyée.

3. **Identifier la ligne en question, donner la valeur du facteur de qualité Q pour cette ligne ainsi que les huit premiers octets correspondants aux données de l'image.**

Q=96. Les données sont : 248, 91, 88 , 248, 7, 246, 31, 216 ...



II Décodage de Huffman

Commençons par revoir et approfondir un peu le principe de la compression JPEG :

Doc. 3 : Compression JPEG

Les étapes de la compression JPEG sont :

A. L'image est découpée en morceaux de 8x8 pixels, puis pour éviter de générer de trop gros nombre lors de l'étape suivante, on réalise une translation de chaque valeur des pixels dans la gamme [-128 ;127] :

B. On réalise la **transformation en cosinus discrete** (DCT) :

$$F_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} S_{yx} \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16}$$

Avec $C_u = 1/\sqrt{2}$ pour u=0 et $C_v = 1/\sqrt{2}$ pour v=0. $C_u = C_v = 1$ sinon.

Compression JPEG - suite

S

124	123	176	239	145	124	239	156
120	120	165	231	203	120	231	178
87	123	212	203	128	87	203	89
89	155	210	145	127	89	145	78
78	156	198	120	124	78	120	156
123	178	194	34	123	123	34	178
122	89	76	89	120	122	89	89
67	78	89	65	90	67	65	78

-128 →

-4	-5	48	111	17	-4	111	28
-8	-8	37	103	75	-8	103	50
-41	-5	84	75	0	-41	75	-39
-39	27	82	17	-1	-39	17	-50
-50	28	70	-8	-4	-50	-8	28
-5	50	66	-94	-5	-5	-94	50
-6	-39	-52	-39	-8	-6	-39	-39
-61	-50	-39	-63	-38	-61	-63	-50

DCT →

F

10	4	-60	-120	-35	26	-28	67
225	-53	-70	-63	0	139	-88	10
-20	-63	-2	66	61	55	-26	-31
61	-5	60	-5	15	-71	48	-14
-36	9	-8	-34	-23	18	-19	0
-22	28	-14	42	-23	39	-1	-15
-17	34	17	-40	-38	-20	33	6
-11	-20	-15	-8	3	53	-27	-9

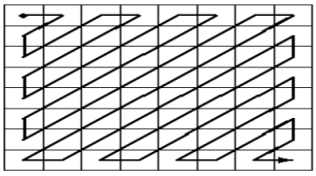
C. Les coefficients obtenus sont divisés par une matrice de quantification judicieusement choisie puis on lit les valeurs en "zig zag" en partant du coin en haut à gauche jusqu'au coin en bas à droite :

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

matrice de quantification Q

F[i] / Q[i] →

1	0	-6	-8	-1	1	-1	1
19	-4	-5	-3	0	2	-1	0
-1	-5	0	3	2	1	0	-1
4	0	3	0	0	-1	1	0
-2	0	0	-1	0	0	0	0
-1	1	0	1	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	1	0	0



matrice « zigzag »

[1, 0, 19, -1, -4, -6, -8, -5, -5, 4, -2, 0, 0, -3, -1, 1, 0, 3, 3, 0, -1, 0, 1, 0, 0, 2, 2, -1, 1, -1, 1, 0, -1, 0, 1, 0, 0, 0, 1, 0, -1, 0, 0, -1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, EOB]

EOB (End Of Byte) signifie qu'il n'y a plus que des zéros jusqu'à la fin. La première valeur est le coefficient DC (lié à la valeur moyenne), tous les autres sont les coefficients AC (ils donnent des informations sur les fréquences). Ils ne seront pas traités de la même façon par la suite.

D. Les valeurs décimales positives ou négatives sont encodées en binaire selon le tableau ci-dessous ([3]) :

Category	Symbols	Extra bits
0	0	—
1	-1, 1	0, 1
2	-3, –2, 2, 3	00, 01, 10, 11
3	-7, ..., –4, 4, ..., 7	000, ..., 011, 100, ..., 111
4	-15, ..., –8, 8, ..., 15	0000, ..., 0111, 1000, ..., 1111
⋮	⋮	⋮
15	-32767, ..., –16384, 16384, ..., 32767	0...00, ...,01 ... 1, 10 ... 0, ..., 11 ... 1
16	32768	

Exemple :

-6 → 001 (on peut le retrouver en effectuant la conversion "normale" 001 → 1 puis $1-(2^3-1) = -6$)

Compression JPEG - suite

E. Pour coder la valeur DC, on utilise la table de huffman suivante, qui indique sur combien de bits il faut lire la valeur encodée ([3]) :

Category	Code length	Codeword
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Exemple : **12** → 1100 sur 4 bits, on ajoute donc le code 101 devant : **1011100**

-102 → 0011001 ($= 25 - (2^7 - 1)$) sur 7 bits, on rajoute le code 11110 : **111100011001**

F. Pour les coefficients AC, on utilise la table suivante ([3]) :

R/S	L	Codeword
0/0	4	1010 (EOB)
0/1	2	00
0/2	2	01
0/3	3	100
0/4	4	1011
0/5	5	11010
0/6	7	1111000
0/7	8	11111000
0/8	10	1111110110
0/9	16	1111111110000010
0/A	16	1111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111110110
1/6	16	1111111110000100
1/7	16	1111111110000101
1/8	16	1111111110000110
1/9	16	1111111110000111
1/A	16	1111111110001000
2/1	5	11100
2/2	8	11111001
2/3	10	1111110111
2/4	12	111111110100
2/5	16	1111111110001001
2/6	16	1111111110001010
2/7	16	1111111110001011
2/8	16	1111111110001100
2/9	16	1111111110001101
2/A	16	1111111110001110
3/1	6	111010
3/2	9	111110111
3/3	12	111111110101
3/4	16	1111111110001111
3/5	16	1111111110010000
3/6	16	1111111110010001
3/7	16	1111111110010010
3/8	16	1111111110010011
3/9	16	1111111110010100
3/A	16	1111111110010101

Cette fois-ci on utilise un algorithme RLE ("Run Length Encoding") sur la valeur zéros. C'est à dire qu'à chaque code correspond deux valeurs (Run/Size). Run correspond au nombre de zéros répétés avant la première valeur non nulle suivante et Size correspond aux nombres de bits qu'il faut lire pour décoder la valeur suivante (sur le même principe que la valeur DC).

Exemples :

6 → 110 sur 3 bits (il y a aucun zéros devant) : code 0/3 : 100 on obtient donc : **100110**

0, 0, -4 → deux zéros devant -4 → 011 : code 2/3 : 1111110111 on obtient donc :

1111110111011

4. **Décoder les 7 premiers octets des données de la ligne 26 sachant qu'ils sont encodés selon l'algorithme de Huffman (commencer par les convertir en binaire *sur 8 bits*).**

   Les données sont : 248, 91, 88 , 248, 7, 246, 31, 216 ... Ce qui donne en binaire sur 8 bits :

   11111000 01011011 01011000 11111000 00000111 11110110 00011111...

   Premier code DC possible : 111110 → 8 bits suivants : 00010110 → $22 - (2^8 - 1) =$ **-233**

   On continue avec 11 01011000 11111000 00000111 11110110 00011111

   Premier code AC 11010 → code 0/5 : pas de zéro avant et on lit les 5 bits suivants : 11000 = **24**

   Il reste : 11111000 00000111 11110110 00011111

   Code AC : 11111000 → code 0/7 : pas de zéro et on lit les 7 bits suivants : 0000011 → $3 - (2^7 - 1) =$ **-124**

   Il reste 1 11110110 00011111

   Code AC : 111110110 → code 1/4 : un zéro et on lit les 4 bits suivants : 0001 → $1 - (2^4 - 1) =$ **-14**

   On n'oublie pas le 0 avant le -14 et pour l'instant nous obtenons :

   [-233, 24, -124, 0, -14...]

5. **Vérifier que votre décodage est correct en utilisant la fonction ci-dessous. Pourquoi a t-on ajouté 1010 à la fin ?**

```
>>> extract_viterbi.huffman("1111100001011011010110001111100000000111111101100001"
                            + "1010")
```

   On ajoute artificiellement 1010 à la fin (→ EOB) pour arrêter le décodage à cet endroit..

```
>>> extract_viterbi.huffman("1111100001011011010110001111100000000111111101100001"+"1010")
dct= [-233, 24, -124, 0, -14]
```

6. **Réaliser le décodage de la ligne 26 complète sachant qu'elle contient un MCU complet de 14 imagettes en utilisant la fonction :**

```
>>> extract_viterbi.huff(line,nbr)
```

   où line correspond à la ligne souhaitée et nbr au nombre d'imagettes à décoder.

```
[[], [-233, 24, -124, 0, -14, -201, 35, 7, -28, 6, -12, -8, 43, -35, 12, -4, 22, 2, 6, 6, -5, -2, 2, 8, -2, -1, -6, -4, 2, -5, 0, -2, -3, -3, 2, 1, 0, 1, 0, -2, -3, -2, 2, 4, 12, 2, 2, 1, -1, 0, -2, -3,
-2, -8, 0, -3, -1, -1, -1, 0, 4, -1, 3], [-226, 21, -131, 128, 81, -26, -71, -5, 73, 91, -4, 69, -22, 9, 7, 4, 11, 1, -21, 13, -12, 1, 5, -8, -11, 11, -2, -8, -4, -1, -3, -2, -2, 3, -5, 3, -1, -2, 0, 0,
3, -1, 7, 10, -4, -1, 2, -1, 1, -2, -1, 0, -1, -3, -3, 1, -2, 1, 0, 0, 3, -1, 1], [-337, -43, -102, -61, 50, -12, 3, -26, 27, -88, -29, -52, -5, -46, -1, 7, -37, 6, -5, -14, 19, 1, 3, 0, 10, 11, -4, 9, 4
, -2, 6, 9, -2, 5, -4, 1, -2, 6, 2, 1, -1, 4, -6, -2, -1, -3, -1, 3, 1, 0, 0, -1, -3, 2, 0, 0, -2, -1, -1, -2, 0, 1, 1], [-229, -37, -26, 15, -52, -10, 4, -34, 3, 16, 26, 42, -36, -11, -2, -1, 8, -23, 4,
 9, 2, -1, -2, -1, -2, -10, 3, 3, 3, 0, 2, -7, 3, 4, 1, 0, -1, 3, 2, 0, -4, 1, -4, 3, 0, 0, 0, 1, -1, 1, 0, 0, -2, 2, -3, 0, 0, 1, 1, 0, 1, 0, 1], [-184, 2, -6, -16, 27, 5, 23, 9, 5, -4, -2, 9, 2, 5, 6,
-11, 3, 1, 2, -5, -2, 2, 2, -1, 3, 4, -5, 3, -5, 0, 0, 2, 1, 2, 0, -2, 0, -2, -1, -1, 2, -1, -3, -3, 1, -1, 2, 1, 0, -1, 1, 0, 0, -2, -1, 0, -1, 0, 0, 0, 1], [-209, -31, 21, -26, -47, 33, 35, 21, 0, -9,
-7, -4, 31, 13, -5, 1, 4, 10, -10, -5, -5, -3, 2, 3, -1, 4, -1, 0, -1, -2, 0, 5, 2, 4, 1, -2, -1, 0, -2, 1, 1, -3, -4, 1, -1, 1, -2, -2, 1, 1, 1, 0, 2, -5, 2, 1, 1, -1, 0, 0, 1, 1, -1], [-244, 23, -25, -
25, 24, 16, -17, -10, 50, 74, 11, -2, 36, -4, -6, -1, 10, -5, -3, -3, 8, -1, 0, 4, -4, 1, 3, -4, -3, -1, -5, 3, 1, -7, -1, 1, 0, -3, -2, 0, 0, 1, 2, 1, -1, 0, 0, 2, -1, 2, 0, -1, 0, -2, -1, -1, 1, -1, 0,
 0, 0, 0, -1], [-267, 6, 50, -30, -65, 43, 23, -38, 58, 43, 27, 18, 17, -8, 22, -2, 16, -22, 1, 8, -3, 4, -5, 1, -2, -7, -2, -8, 0, 0, 6, -1, 1, 0, -4, -2, -2, -2, 4, 0, -2, 1, -1, 3, 3, 1, 3, 2, -1, -1,
 -2, -1, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 1], [-245, -24, 96, -83, -12, -7, 35, 37, 27, -12, 13, 33, -7, 4, 16, 7, 11, -15, -17, -10, 8, -4, 1, 1, -1, -6, 2, -4, -6, -3, -1, 4, -1, 3, -3, -3, 3, 1, 0, -2,
 -1, 4, -3, 3, -1, 0, 1, 4, -1, 1, 1, 2, -1, 0, 0, -1, 1, 0, 0, 0, 0, -1, 1], [-147, 104, 108, -13, 59, 2, -15, 11, -40, 10, 7, -42, -27, 8, 11, -8, -6, 6, -1, -14, 4, 0, 4, 7, -14, -9, 1, -1, -1, -1, -
1, -1, -3, -1, 1, -1, 2, 1, 1, 1, 0, 0, 0, -2, 2, 1, 5, 1, 0, 0, 2, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1], [-195, -56, -3, 7, 22, 11, 17, -6, -21, 52, 13, 19, -15, -7, -4, -3, -1, -6, 6, 7, 1, 2, 3, -2, -4, -
1, 2, -1, 1, 1, 0, -1, 3, -2, 2, 0, -1, 1, 2, 1, -1, 1, 0, 0, 0, -1, 0, 1, 1, 0, -1, 0, -1, 0, -1, 1, 1, 0, 1, 0, 0, -1], [-165, 37, 0, -20, -40, 8, 24, 18, 4, 14, -23, 16, -14, 13, 3, 1, 4, 16, 1, 2, -4,
 -1, -3, -2, -1, -4, -2, -2, 1, -1, 0, 1, 0, 0, -2, -1, 0, 1, 1, 0, -1, 0, 0, 1, 0, 1, 0, 0, -1, 1, 0, 1, 1, 0, 0, 0, 0, 0, -1, 0, 1], [-69, -103, -32, 10, 76, 36, 34, 8, -30, -15, -11, -11, -23, 1, 3, 4
, -4, 7, 6, -5, -4, 1, 1, 3, 18, -1, 2, 2, 2, -2, -1, -4, -5, 1, 3, 2, 0, -1, 0, -4, -4, 0, -1, 2, 3, 2, 1, 0, -2, -1, -1, 0, 0, -2, 0, 0, -1, 1, 0, 1, -1, 1], [4, 7, -9, 34, -12, 4, 1, -3, -11, -26, -27, 1
3, 6, -9, -5, -4, 0, 6, 14, 2, 6, 2, 2, -1, 4, 6, 0, 1, 3, 0, 0, 2, -3, -7, 0, -2, -2, 1, -1, -1, 0, -2, 1, 1, 1, 0, -1, 1, 1, -1, 0, -1, 0, 1, 0, -1, -2, 0, 2, 0, 0, 1, 1]]
>>> extract_viterbi.huff(25,14)
```

Lors du décodage manuel, on se rend compte que certains octets sont codés sur plus de huit bits, étrange pour une méthode de compression ? Pour l'exemple traité à la main plus haut, le gain en compression n'est pas flagrant... Qu'en est-il lorsqu'on considère un plus grand nombre d'octets ?

7. **La fonction précédente affiche dans la console le nombre d'octets compressés et le nombre d'octets décompressés. Calculer le pourcentage de compression gagné avec la méthode de Huffman. Commenter.**

   On remarque pour la ligne 26 : nbr total octets compressés= 740 et nbr total octets decompressés= 874. Il y a bien un gain de place : $(874 - 740)/874 \simeq 0,15$ donc environ 15% de gagné grâce à la compression de Huffman. En effet, statistiquement, il y a plus de code < 8 bits que de code > 8 bits. Le RLE sur la valeur 0 permet également de gagner beaucoup de place.

8. **Calculer le taux de compression associé à cette zone (on ne tient pas compte des tables à joindre au fichier ni du facteur de qualité à transmettre également). Commenter.**

   On a 14x8x8 = 896 pixels à afficher. Sur un canal, un pixel est codé par un octet donc une taille finale de 896 octets. On a vu précédemment qu'il y avait 740 octets compressés. taux compression = 896/740 = 1,21. Il s'agit d'une zone faiblement compressée. L'essentiel de la compression est ici effectué par le codage de Huffman.

III Des zig et des zag

9. **Compléter les 15 premières valeurs de la première matrice de la ligne 26 en utilisant la lecture en zigzag :**

-233							

-233	24	-201	35	12	...		
-124	-14	7	-35				
0	-28	43					
6	-8						
-12							

Puis vérifier avec la fonction :

```
>>> extract_viterbi.zigzag(ligne,nbr_total,matrice)
```

ligne : le numéro de la ligne

nbr_total : le nombre d'imagettes dans la ligne considérée

matrice : le numéro de la matrice à afficher

```
array([[-233.,   24., -201.,   35.,   12.,   -4.,   -4.,    2.],
       [-124.,  -14.,    7.,  -35.,   22.,   -6.,   -5.,    2.],
       [   0.,  -28.,   43.,    2.,   -1.,    0.,   -2.,    4.],
       [   6.,   -8.,    6.,   -2.,   -2.,   -3.,   12.,   -8.],
       [ -12.,    6.,    8.,   -3.,   -2.,    2.,   -2.,    0.],
       [  -5.,    2.,   -3.,    0.,    2.,   -3.,   -3.,    4.],
       [  -2.,    2.,    1.,    1.,   -2.,   -1.,    0.,   -1.],
       [   1.,    0.,   -1.,    0.,   -1.,   -1.,    3.,    0.]])
>>> extract_viterbi.zigzag(25,14,0)
```

10. **Faire afficher la matrice suivante. En vérifiant avec la liste affichée plus haut dans la console, que constate-t-on pour la valeur DC ? Prédire la valeur DC de la troisième matrice.**

```
array([[-226.,   21.,  -26.,  -71.,    7.,    4.,   -8.,   -4.],
       [-131.,   81.,   -5.,    9.,   11.,   -2.,   -1.,    7.],
       [ 128.,   73.,  -22.,    1.,   11.,   -3.,   -1.,   10.],
       [  91.,   69.,  -21.,  -11.,   -2.,    3.,   -4.,   -3.],
       [  -4.,   13.,   -8.,   -2.,    0.,   -1.,   -1.,   -3.],
       [ -12.,    5.,    3.,    0.,    2.,    0.,    1.,    3.],
       [   1.,   -5.,   -2.,   -1.,   -1.,   -2.,    0.,   -1.],
       [   3.,   -1.,    1.,   -2.,    1.,    0.,    1.,    0.]])
>>>
```

Dans la liste calculée plus haut, on a [-7, 21, -131, 128, 81 ...etc.]. La valeur DC est de -7 et pas -226. En effet, d'une imagette à l'autre la valeur DC indiquée est relative à celle précédente : -233 - (-7) = -226. Les valeurs AC sont inchangées.

Pour la troisième matrice, on a dans la liste DC=111 donc -226 - 111 = -337

```
array([[-337.,  -43.,  -12.,    3.,   -1.,    7.,    9.,    4.],
       [-102.,   50.,  -26.,  -46.,  -37.,   -4.,   -2.,   -6.],
       [ -61.,   27.,   -5.,    6.,   11.,    6.,    4.,   -2.],
       [ -88.,  -52.,   -5.,   10.,    9.,   -1.,   -1.,    2.],
       [ -29.,  -14.,    0.,   -2.,    1.,   -3.,   -3.,    0.],
       [  19.,    3.,    5.,    2.,   -1.,   -1.,    0.,    0.],
       [   1.,   -4.,    6.,    3.,    0.,   -2.,   -2.,    1.],
       [   1.,   -2.,    1.,    0.,   -1.,   -1.,    1.,    0.]])
```

IV Quantification inverse

Commencer par créér une variable DCT qui va stocker la valeur de la première matrice en zigzag :

```
>>> DCT=extract_viterbi.zigzag(26,14,0)
```

Pour la quantification on utilise une matrice classique de quantification que l'on nomme HTK (voir ci-dessous)

Cette matrice est modifiée en fonction du facteur de qualité Q selon les instructions suivantes :

— Si Q est compris entre 20 et 50 : $F = \frac{5000}{Q}$ et $F = 200 - 2 \times Q$ sinon.

— On en déduit une nouvelle matrice PTK dont les éléments sont ceux de HTK multiplié par F/100.

— La matrice originale avant quantification F0 est obtenue en multipliant élément par élément DCT et PTK.

11. **Ouvrir le fichier extract_viterbi.py et compléter la fonction quantif(DCT,Q) (vers les lignes 520) en tenant compte des instructions ci-dessus.**

```
def quantif(DCT,q):

    HTK=[
    [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]
    ]

    PTK=[
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0]
    ]


    #if ... Q .... :     ### COMPLÉTER ICI
    #   .......


    for u in range(0,8):
        for v in range(0,8):
    #       PTK[u][v]=round(.................)    ### COMPLÉTER ICI
            if PTK[u][v]<1:
                PTK[u][v]=1
    PTK=np.array(PTK)
    print("PTK=",PTK)

    F0=np.zeros((8,8), dtype=float)


    #for i in range(8):
    #   for j in range(8):
    #       F0[i,j]= ..............    ### COMPLÉTER ICI

    print("F0=",F0)
    return F0
```

```
    PTK=[
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0],
    [0,0,0,0,0,0,0,0]
    ]

    if 20<Q<50:          ### COMPLÉTER ICI
        F=5000/Q*1.0
    if 50<=Q<=100:
        F=200-2*Q*1.0


    for u in range(0,8):
        for v in range(0,8):
            PTK[u][v]=round(HTK[u][v]*F/100.0)
            if PTK[u][v]<1:
                PTK[u][v]=1
    PTK=np.array(PTK)
    print("PTK=",PTK)

    F0=np.zeros((8,8), dtype=float)


    for i in range(8):
        for j in range(8):
            F0[i,j]=DCT[i,j]*PTK[i,j]

    print("F0=",F0)
    return F0
```

12. **Executer la fonction pour afficher la matrice F0 et la stocker dans une variable F0 :**

```
>>> import importlib                #Faire qu'une fois
>>> importlib.reload(extract_viterbi)        #Refaire apres chaque modification
>>> F0 = extract_viterbi.quantif(DCT,Q)
```

En ayant bien pris soin de remplacer la valeur de Q par celle trouvée à la question 18.

Q= 96

```
PTK= [[ 1  1  1  1  2  3  4  5]
 [ 1  1  1  2  2  5  5  4]
 [ 1  1  1  2  3  5  6  4]
 [ 1  1  2  2  4  7  6  5]
 [ 1  2  3  4  5  9  8  6]
 [ 2  3  4  5  6  8  9  7]
 [ 4  5  6  7  8 10 10  8]
 [ 6  7  8  8  9  8  8  8]]
F0= [[-233.   24. -201.   35.   24.  -12.  -16.   10.]
 [-124.  -14.    7.  -70.   44.  -30.  -25.    8.]
 [   0.  -28.   43.    4.   -3.    0.  -12.   16.]
 [   6.   -8.   12.   -4.   -8.  -21.   72.  -40.]
 [ -12.   12.   24.  -12.  -10.   18.  -16.    0.]
 [ -10.    6.  -12.    0.   12.  -24.  -27.   28.]
 [  -8.   10.    6.    7.  -16.  -10.    0.   -8.]
 [   6.    0.   -8.    0.   -9.   -8.   24.    0.]]
>>>
```

V TRANSFORMÉE EN COSINUS DISCRÈTE INVERSE

La formule permettant de calculer la DCT inverse est donnée :

$$S_{yx} = \frac{1}{4}\sum_{u=0}^{7}\sum_{v=0}^{7} C_u C_v S_{vu} \cos\frac{(2x+1)u\pi}{16} \cos\frac{(2y+1)v\pi}{16}$$

Avec $C_u = 1/\sqrt{2}$ pour u=0 et $C_v = 1/\sqrt{2}$ pour v=0. $C_u = C_v = 1$ sinon.

13. **Toujours dans le même fichier, compléter la fonction suivante tcdi(F0) à l'aide de la formule ci-dessus (π = math.pi) :**

```
564 def tcdi(F0):
565     S=np.zeros((8,8), dtype=float)
566
567     for x in range(0,8):
568         for y in range(0,8):
569             somme=0
570             for u in range(0,8):
571                 for v in range(0,8):
572                     if u==0:
573                         Cu=1/math.sqrt(2)
574                     if v==0:
575                         Cv=1/math.sqrt(2)
576                     if u!=0:
577                         Cu=1
578                     if v!=0:
579                         Cv=1
580
581                     somme+=0.25*(..... )     ## COMPLETER ICI
582
583                     if somme >127:
584                         somme=127
585                     if somme<-128:
586                         somme=-128
587             S[x,y]=round(somme)
588     print("S=",S)
589     return S
590
591
```

Correction :

```
if v!=0:
    Cv=1

somme+=0.25*(Cu*Cv*F0[u,v]*math.cos((2*x+1)*u*math.pi/16)*math.cos((2*y+1)*v*math.pi/16))

if somme >127:
    somme=127
```

14. **Executer la fonction pour afficher la matrice S et la stocker dans une variable :**

```
>>> S = extract_viterbi.tcdi(F0)
```

```
S= [[-85. -52. -46. -36. -18. -64. -79. -41.]
 [-77. -66. -45.   1.   1. -54. -58. -51.]
 [-80. -42. -35.  -3.  32. -56. -60. -89.]
 [-63. -34. -37.  14.  29. -70. -44. -94.]
 [-40. -43.   4.  19. -10.  23. -65. -68.]
 [-32. -30.  -8.   9.   9.  45. -61. -63.]
 [-27. -33.  -4.   5.  53.  24. -27. -63.]
 [ -9. -17. -19.   4.  16.   4.  22. -78.]]
>>>
```

15. **Pourquoi a t-on des nombres négatifs ? Que reste t-il à faire ? Le réaliser.**

Il faut ajouter 128 pour revenir dans la gamme [0 ;255]

```
>>> S=S+128
>>> S
array([[ 43.,  76.,  82.,  92., 110.,  64.,  49.,  87.],
       [ 51.,  62.,  83., 129., 129.,  74.,  70.,  77.],
       [ 48.,  86.,  93., 125., 160.,  72.,  68.,  39.],
       [ 65.,  94.,  91., 142., 157.,  58.,  84.,  34.],
       [ 88.,  85., 132., 147., 118., 151.,  63.,  60.],
       [ 96.,  98., 120., 137., 137., 173.,  67.,  65.],
       [101.,  95., 124., 133., 181., 152., 101.,  65.],
       [119., 111., 109., 132., 144., 132., 150.,  50.]])
>>>
```

VI Enfin des images !

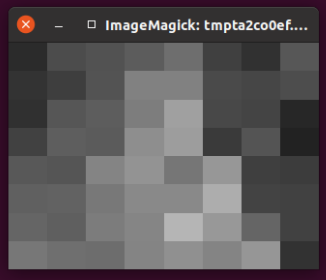
On utilise le module image de PIL :

```
>>> import PIL
>>> image = PIL.Image.fromarray(S)
>>> image.show()
```

16. **Qu'observez-vous ?**

On observe une imagette de 8x8 pixels... Pas facile à retrouver dans l'image originale (bon courage) !

```
>>> image = PIL.Image.fromarray(S)
>>> image.show()
>>> S
array([[ 43.,  76.,  82.,  92., 110.,  64.,  49.,  87.],
       [ 51.,  62.,  83., 129., 129.,  74.,  70.,  77.],
       [ 48.,  86.,  93., 125., 160.,  72.,  68.,  39.],
       [ 65.,  94.,  91., 142., 157.,  58.,  84.,  34.],
       [ 88.,  85., 132., 147., 118., 151.,  63.,  60.],
       [ 96.,  98., 120., 137., 137., 173.,  67.,  65.],
       [101.,  95., 124., 133., 181., 152., 101.,  65.],
       [119., 111., 109., 132., 144., 132., 150.,  50.]])
>>> import PIL
>>> from PIL import image
Traceback (most recent call last):
```



On va essayer de faire mieux en affichant les 14 imagettes de la ligne 26 :

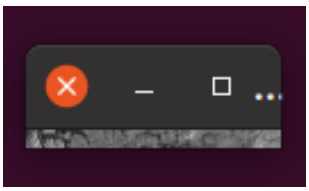
17. **En récapitulant toutes les étapes précédentes dans une boucle, remplir une liste s=[] contenant les 14 matrices S. Puis à l'aide de la fonction np.concatenate((a,b), axis=1) rassembler toutes les matrices en une seule ligne et afficher l'image.**

```
>>> s=[]
>>> for i in range(14):
>>>   DCT=...
>>>   F0=...
>>>   S= ...
>>>   S= ...
>>>   s.append(S)
```

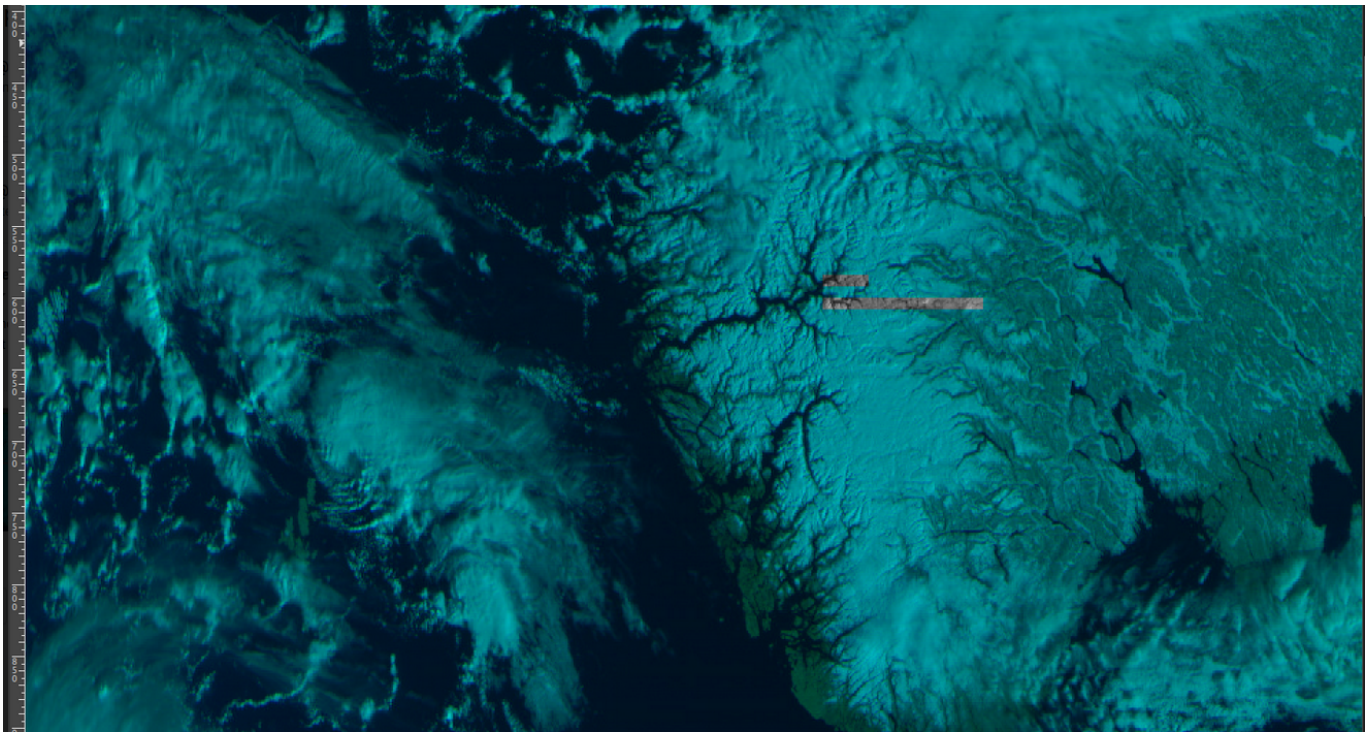
```
>>> S=s[0]
>>> for i in range(...):
>>>     S=np.concatenate((S,....), axis=1)
>>> image = PIL.Image.fromarray(S)
>>> image.show()
```

```
>>> s=[]
>>> for i in range(14):
...     DCT=extract_viterbi_CORR.zigzag(26,14,i)
...     F0 = extract_viterbi_CORR.quantif(DCT,96)
...     S=extract_viterbi_CORR.tcdi(F0)
...     S=S+128
...     s.append(S)
...
```

```
>>> S=s[0]
>>> import PIL
>>> S=s[0]
>>> for i in range(len(s)-1):
...     S=np.concatenate((S,s[i+1]), axis=1)
...
>>> image=PIL.Image.fromarray(S)
>>> image.show()
>>>
```

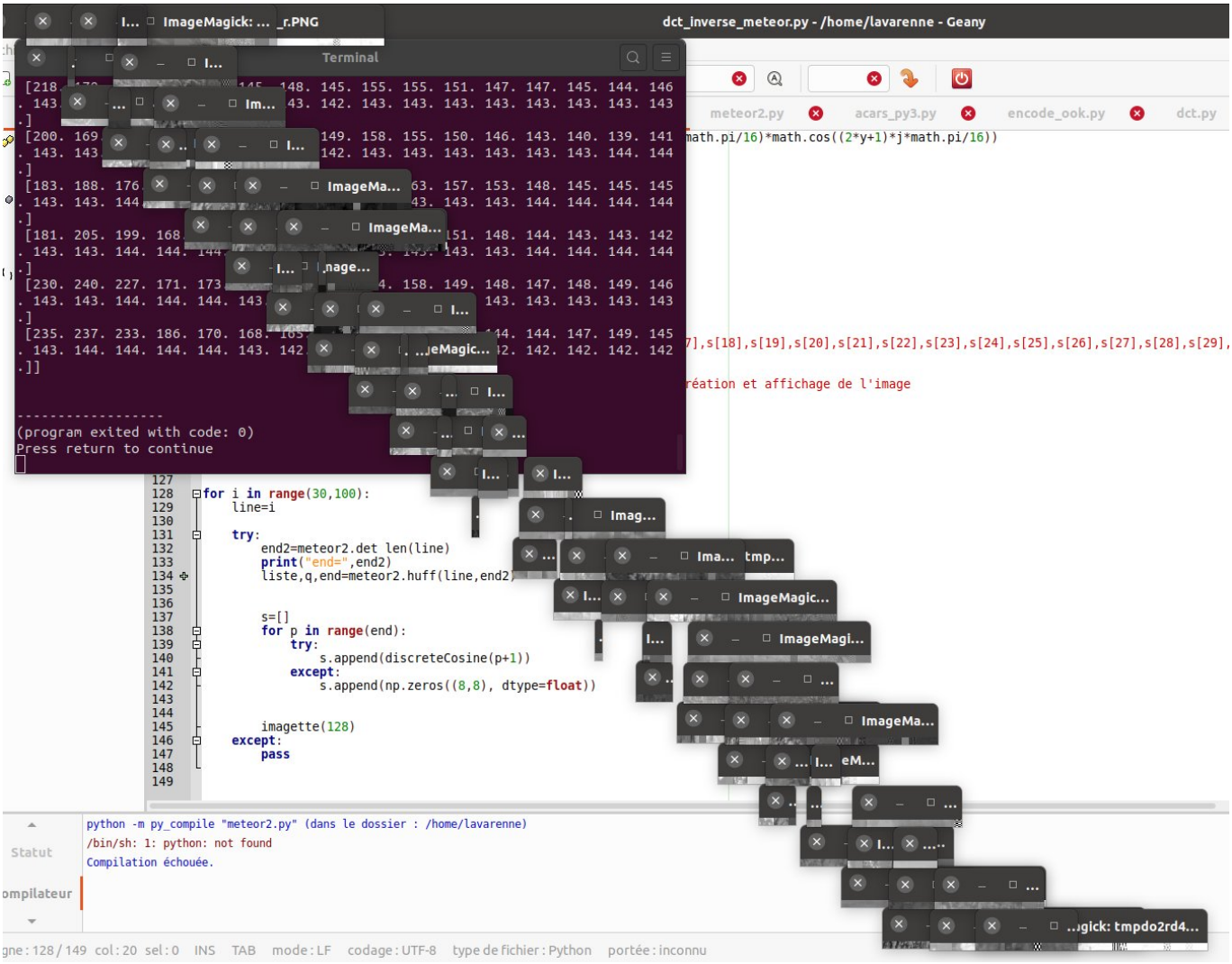


18. **Reconnaissez vous cet endroit sur l'image entière ? Enregistrer l'image au format bmp (clic - File - Save) et à l'aide de The Gimp par exemple, superposez cette imagette sur l'image originale pour montrer la correspondance (Indice : Regarder dans les pays nordiques..)**



19. **Bonus : Essayer avec d'autres lignes (attention, elles ne contiennent pas toutes 14 imagettes, commencer par essayer avec un nombre plus petit)**

En adaptant un peu la démarche, on peut créer un petit script qui décode et affiche une centaine de lignes à la suite. Il faudrait ensuite remettre tous ces morceaux dans l'ordre en utilisant le compteur présent au début de chaque ligne...



Références :

[1] J.-M Friedt, *Décodage d'images numériques issues de satellites météorologiques en orbite basse : le protocole LRPT de Meteor-M2 (partie 1/2)*, janvier 2019, http ://jmfriedt.free.fr/glmf_meteor1.pdf

[2] J.-M Friedt, *Décodage d'images numériques issues de satellites météorologiques en orbite basse : le protocole LRPT de Meteor-M2 (partie 2/2)*, mars 2019, http ://jmfriedt.free.fr/glmf_meteor1.pdf

[3] Memon Nasir, ... Ansari Rashid, in Handbook of Image and Video Processing (Second Edition), 2005, *The JPEG Lossless Image Compression Standards*, https ://www.sciencedirect.com/topics/engineering/huffm table

[4] Meteor-decoder, https ://github.com/artlav/meteor_decoder