

Rapport SAE Image

A.0

La première partie du fichier est l'en tête du fichier :

- les 2 premier octets 42 4D correspondent au format du fichier 42 qui fait B en Ascii et 4D qui fait M.

0000:0000 424D9973 0C000000 00001A00 00000C00
0000:0010 00008002 A9010100 1800FFFF FFFFFFFF
- Ensuite il y a 4 octets : 99 73 0C 00 qui donnent la taille du fichier avec cela on peut avoir la taille du fichier en octets. Cependant il faut inverser tout les octets pour convertir car c'est du little endian : $9 + 9 \times 16^1 + 3 \times 16^2 + 7 \times 16^3 + 12 \times 16^4 = 816025$ L'erreur avec display vient du fait que la taille du fichier en octet sur okteta ne correspond pas à la taille réelle en octet.



Ici on voit que l'image fait 816026 octets or sur le fichier il est indiqué 816025 octets il faut donc rajouter 1 bit dans le fichier pour qu'il n'y ai plus d'erreur avec display. Il faut donc changer 99 en 9A pour avoir : $10 + 9 \times 16^1 + 3 \times 16^2 + 7 \times 16^3 + 12 \times 16^4 = 816026$

0000:0000 424D9A73 0C000000 00001A00 00000C00
0000:0010 00008002 A9010100 1800FFFF FFFFFFFF

A.1

Avec ce code :

```
42 4D 4A 00 00 00 00 00 00 00 1A 00 00 00 0C 00
00 00 04 00 04 00 01 00 18 00 00 00 FF FF FF FF
00 00 FF FF FF FF FF FF FF 00 00 FF FF FF FF 00
00 FF 00 00 FF FF FF FF 00 00 FF FF FF FF FF FF
FF 00 00 FF FF FF FF 00 00 FF
```

on obtient cette image



Les suite FF FF FF sont les pixels de couleur blanche et les 00 00 FF sont pour le rouge (B = 0, G = 0, R = 255 car on es en little endian) 4A est la taille du fichier en octets (sur 4 octets), 1A est l'adresse à laquelle commence l'image (le début des pixels) le 0C donne la taille de la deuxième partie de l'entête. Enfin le 18 vient du fait que l'on utilise 3 octets pour coder la couleur d'un pixel (RGB), on utilise donc $8 \times 3 = 24$ bits pour coder un pixel et en hexadecimal $24 = 18$.

A.2

En convertissant l'image en bmp3, on obtient bien une nouvelle taille de 102 octets Avec ce code :

```
42 4D 4A 00 00 00 00 00 00 00 1A 00 00 00 0C 00
00 00 04 00 04 00 01 00 18 00 FF FF 00 FF 00 FF
E8 9D 0F FF FF FF 00 00 FF 00 00 FF 00 00 FF FF
00 00 00 00 FF 00 00 FF 00 00 FF 00 00 FF 00 00
FF 00 FF 00 00 00 FF 00 00 FF
```

on obtient cette image



Ici on a juste changé les pixels pour avoir d'autres couleurs par exemple 00 FF 00 pour avoir du vert. Pour les couleurs non primaire il a fallu convertir leur code rgb décimal en hexadécimal et le mettre en little endian sur le code. Par exemple pour le bleu céruleen, on a le code 15, 157, 232 qui donne en hexadécimal : 0F, 9D, E8 qui donne en little endian E8 9D 0F.

A.3

En enlevant 12 octets à la taille de l'image et en en rajoutant 40 , on obtient une nouvelle taille de $72 - 12 + 40 = 102$ octets

Voici le code de l'image convertie

```
SAE > Image > ≡ Image1.bmp
⚙️      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  42 4D 66 00 00 00 00 00 00 00 00 36 00 00 00 28 00
00000010  00 00 04 00 00 00 04 00 00 00 00 01 00 18 00 00 00
00000020  00 00 30 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030  00 00 00 00 00 00 00 00 00 FF FF FF FF 00 00 FF FF
00000040  FF FF FF FF FF 00 00 FF FF FF FF 00 00 FF 00 00
00000050  FF FF FF FF 00 00 FF FF FF FF FF FF FF FF 00 00 FF
00000060  FF FF FF 00 00 FF +
```

Sur ce code nous pouvons voir que la taille de l'image est toujours à l'adresse 0x02 sur 4 octets. Ici la valeur à cette adresse est de 66 soit 102 en décimal. On retrouve bien la taille calculée précédemment.

On remarque aussi qu'un autre en tête a été ajouté à partir de 18 00 à l'adresse 0x1C.

1. Il y a toujours 24 bits par pixels (18 = 24 en décimal)
2. La taille des données pixels est de 48 octets. Cette valeur s'obtient grâce à la valeur 30 à l'adresse 0x22 qui est la taille des données pixels de l'image. 30 = 48 en décimal soit 48 octets de pixels.
3. La compression de l'image est indiquée à l'adresse 0x1E, cette compression est codée sur 4 octets. Dans notre cas à l'adresse 0x1E il n'y a que des zéros, aucune compression n'est donc utilisée.
4. Non le codage des pixels n'a pas changé, on code toujours les pixels sur 3 octets en little endian.

A.4

En convertissant l'image on obtient ce code :

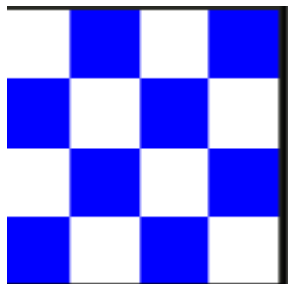
```
SAE > Image > ≡ Image2.bmp
⚙️      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  42 4D 4E 00 00 00 00 00 00 00 00 3E 00 00 00 28 00
00000010  00 00 04 00 00 00 04 00 00 00 01 00 01 00 00 00
00000020  00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 02 00
00000030  00 00 02 00 00 00 00 00 00 FF 00 FF FF FF 00 50 00
00000040  00 00 A0 00 00 00 50 00 00 00 00 A0 00 00 00 +
```

1. Il y a 1 bit par pixel. ce nombre se trouve à l'adresse 0x1C.

2. La taille des données pixels se trouve à l'adresse 0x22. A cette adresse on trouve le nombre 10 qui en décimal nous donne 16. La taille des données pixels est donc de 16 octets.
3. La compression utilisée est indiquée à l'adresse 0x1E. A cette adresse on ne trouve que des zéros donc il n'y a pas de compression utilisée pour cette image.
4. Les couleurs de la palette sont codées en BGR (RGB mais en little endian) sur 4 octets avec un octet réservé. Pour le blanc on a FF FF FF et pour le rouge on a 00 00 FF
5. la palette de couleur est composée de deux couleurs. On trouve cette information à l'adresse 0x2E ou l'on trouve 2 en hexadécimal soit 2 en décimal donc 2 couleurs utilisées pour la palette.
6. Oui le codage des pixels a changé. Dans ce code, chaque pixel est représenté par 1 seul bit alors que dans les images précédentes on utilisait 24 bits pour représenter un pixel.
7. Avec ce code :

```
SAE > Image > Image3.bmp
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 42 4D 4E 00 00 00 00 00 00 00 3E 00 00 00 28 00
00000010 00 00 04 00 00 00 04 00 00 00 01 00 01 00 00 00
00000020 00 00 10 00 00 00 00 00 00 00 00 00 00 00 02 00
00000030 00 00 02 00 00 00 FF 00 00 00 FF FF FF 00 50 00
00000040 00 00 A0 00 00 00 50 00 00 00 A0 00 00 00 +
```

on obtient cette image :



Pour obtenir cette image il faut modifier la valeur d'une des deux couleurs de notre palette, ici c'est le rouge qu'il fallait changer. Il fallait donc remplacer le 00 00 FF 00 (qui correspond au rouge avec un octet réservé à la fin) à l'adresse 0x36 et mettre FF 00 00 00 pour avoir du bleu.

8. Avec ce code :

```
SAE > Image > ≡ ImageBleueinverse.bmp
⚙️      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  42 4D 4E 00 00 00 00 00 00 00 3E 00 00 00 28 00
00000010  00 00 04 00 00 00 04 00 00 00 01 00 01 00 00 00
00000020  00 00 10 00 00 00 00 00 00 00 00 00 00 00 02 00
00000030  00 00 02 00 00 00 FF 00 00 00 FF FF FF 00 A0 00
00000040  00 00 50 00 00 00 A0 00 00 00 50 00 00 00 00 +
```

on obtient cette image :

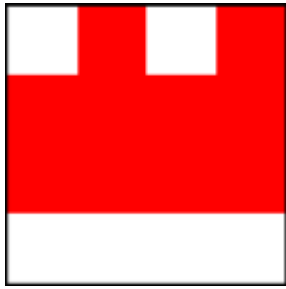


Dans ce code comme les pixels sont codés sur 1 seul bit et qu'il n'y a que deux couleurs dans la palette, on peut en déduire que le 0 correspond au bleu et le 1 au blanc. Si on regarde pour les A0 dans les données pixels, ils donnent en binaire 1010 0000 soit en big endian : 0000 1010 ce qui nous donne blanc, bleu, blanc, bleu. Pour les 50, on a en binaire 0101 0000 qui donne en big endian 0000 0101 : bleu, blanc, bleu, blanc. En faisant cela 4fois on retrouve le patern du damier précédent. Pour l'inverser, il suffisait de seulement changer les 50 en A0 et les A0 en 50.

9. Avec ce code :

```
SAE > Image > ≡ Image3.bmp
⚙️      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  42 4D 4E 00 00 00 00 00 00 00 3E 00 00 00 28 00
00000010  00 00 04 00 00 00 04 00 00 00 01 00 01 00 00 00
00000020  00 00 10 00 00 00 00 00 00 00 00 00 00 00 02 00
00000030  00 00 02 00 00 00 00 00 00 FF 00 FF FF FF 00 F0 00
00000040  00 00 00 00 00 00 00 00 00 00 00 00 A0 00 00 00 +
```

on obtient cette image :



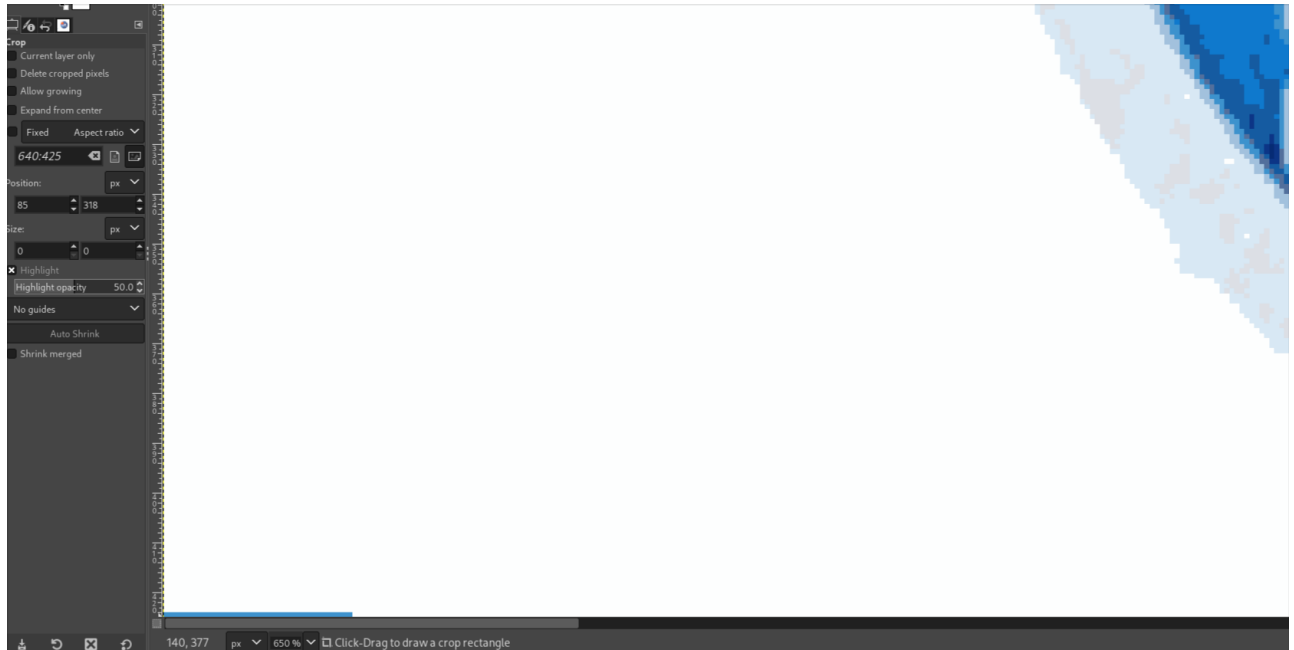
Pour ce code, il a fallu remettre le rouge dans la palette de couleur avec 00 00 FF 00 à l'adresse 0x36. Pour la ligne de pixel blancs, il a fallu mettre F0 en hexa. En effet une ligne de pixels blanc correspond à une suite de 1 en binaire : 1111 qui en little endian donne 1111 0000 et en hexa donne F0. Ensuite, le 0 correspondant au rouge en binaire, il a fallu laisser les octets à 0 sur deux lignes pour avoir les deux lignes de pixels rouges sur l'image. Enfin pour la dernière ligne, il fallait alterner entre deux couleur (1 sur 2) comme dans la question précédente. Il fallait donc coder cette suite en binaire : 1010 soit blanc, rouge, blanc, rouge. En little endian cela donne 0000 1010 qui donne A0 en hexa.

L'en tête de l'image convertie :

```
SAE > Image > ≡ ImageExempleIndexBMP3_16.bmp
⚙️      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 42 4D B6 13 02 00 00 00 00 00 76 00 00 00 28 00
00000010 00 00 80 02 00 00 A9 01 00 00 01 00 04 00 00 00
00000020 00 00 40 13 02 00 00 00 00 00 00 00 00 00 10 00
00000030 00 00 10 00 00 00 74 2B 06 00 5C 23 0A 00 70 50
00000040 38 00 0C 66 FA 00 20 67 E8 00 51 6F AB 00 86 35
00000050 0C 00 A1 5B 15 00 CD 79 0F 00 9C 6F 52 00 CD 92
00000060 3D 00 F4 E8 D8 00 FE FE FD 00 E5 DF DC 00 DD C1
00000070 A2 00 5C 91 DD 00 CC CC CC CC CC CC CC CC CC
```

10. On trouve le nombre de couleur dans la palette à l'adresse 0x2E. ici on trouve 10 en hexa soit 16 en décimal donc 16 couleurs dans la palette.
11. La couleur blanche pure en RGB est codée en hexa par FF FF FF. Il faut donc trouver la couleur dans la palette dont le code se rapproche le plus de celui du blanc pur. On trouve FE FE FD à l'adresse 0x66 qui est une couleur à dominante blanc.
12. Le tableau de pixel commence à l'adresse 0x76. On trouve cette information à l'adresse 0x0A qui donne l'adresse de la zone de définition de l'image ici c'est 76.

13. L'image avec les pixels bleus rajoutés :



Le code de cette nouvelle image :

```
SAE> Image > ImageExempleIndexBMP3_16PixelsBleus.bmp
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 42 4D B6 13 02 00 00 00 00 00 76 00 00 00 28 00
00000010 00 00 80 02 00 00 A9 01 00 00 01 00 04 00 00 00
00000020 00 00 40 13 02 00 00 00 00 00 00 00 00 00 10 00
00000030 00 00 10 00 00 00 74 2B 06 00 5C 23 0A 00 70 50
00000040 38 00 0C 66 FA 00 20 67 E8 00 51 6F AB 00 86 35
00000050 0C 00 A1 5B 15 00 CD 79 0F 00 9C 6F 52 00 CD 92
00000060 3D 00 F4 E8 D8 00 FE FE FD 00 E5 DF DC 00 DD C1
00000070 A2 00 5C 91 DD 00 AA AA AA AA AA AA AA AA AA
00000080 AA AA AA AA AA AA AA AA AA CC CC CC CC CC CC
00000090 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
000000A0 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
000000B0 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
```

Pour rajouter ces pixels bleus en bas à gauche, il a fallu rajouter des A pour chaque octets au début de la zone de définition de l'image. Mais pourquoi ? On sait que l'image a 16 couleurs grace au 10 à l'adresse 0x2E qui nous donne 16 en décimal. Ensuite, on sait que la palette de couleur commence à l'adresse 0x36 et que la zone de définition de l'image commence à l'adresse 0x76. On peut donc en déduire que la palette de couleur va de l'adresse 0x36 à l'adresse 0x76. Il faut maintenant trouver une couleur qui donne bleu parmi les 16 que contient la palette. Cependant, nous sommes en little endian il ne faut donc pas oublier d'inverser le rouge et le bleu pour passer de BGR à RGB. J'ai choisi la couleur CD 92 3D (3D 92

CD en big endian) qui donne le bleu des pixels qu'on voit sur la première image. Après avoir choisi la couleur, il faut la coder de sorte à ce que les pixels soit en bleus. Pour cela il faut d'abord regarder la position de la couleur dans la palette, ici elle se trouve à l'adresse 0x5E et en comptant à partir du début de la palette on trouve que c'est la 10ème couleur de la palette. Enfin, on sait qu'un pixel est codé sur 4bits et que par conséquent un octet permet de coder 2 pixels. En convertissant 10 en binaire on a 1010 (ce qui tient bien sur 4 bits) et en hexa cela nous donne A. En remplaçant les CC par des AA dans les premiers octets, on modifie pour chaque octet la couleur de 2 pixels en bleus


14. L'image avec seulement 4 couleurs :



Visuellement, on remarque qu'il manque des couleurs comme le orange (remplacé par du gris) et que des pixels qui étaient avant des nuances de certaines couleurs comme le bleu on été remplacés par du blanc.

Le code de l'image :

SAE > Image > ≡ ImageExempleIndexBMP3_4.bmp



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	42	4D	B6	13	02	00	00	00	00	00	76	00	00	00	28	00
00000010	00	00	80	02	00	00	A9	01	00	00	01	00	04	00	00	00
00000020	00	00	40	13	02	00	00	00	00	00	00	00	00	00	10	00
00000030	00	00	10	00	00	00	A2	59	17	00	FC	FB	F9	00	DD	C1
00000040	A2	00	6A	54	5A	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	11	11	11	11	11	11	11	11	11	11
00000080	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
00000090	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11


Au niveau de l'hexa on peut voir qu'une bonne partie de la palette de couleur a été remplacée par des 0 puisqu'il n'y a plus que 4 couleurs. Les pixels ont changé puisque qu'il n'y a plus les mêmes couleurs qu'avant. Cependant on peut voir que selon l'hexa le nombre de couleurs n'a pas changé puisqu'à l'adresse 0x2E on retrouve toujours 10 en hexa soit 16 en décimal donc 16 couleurs alors qu'il n'y en a que 4.

A.5

- En changeant la hauteur de 04 00 00 00 à FC FF FF FF, on passe la hauteur de 4 à -4. Pour avoir ce nombre négatif, il fallait passer par le C2 : 4 en binaire donne 0000 0000 0000 0100 et en C2 cela donne 1111 1111 1111 1100 qui donne en hexa : FF FF FF FC.

Dans le code cela donne :

SAE > Image > ≡ Image3.bmp



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	42	4D	4E	00	00	00	00	00	00	00	3E	00	00	00	28	00
00000010	00	00	04	00	00	00	FC	FF	FF	FF	01	00	01	00	00	00
00000020	00	00	10	00	00	00	00	00	00	00	00	00	00	00	02	00
00000030	00	00	02	00	00	00	00	00	FF	00	FF	FF	FF	00	F0	00
00000040	00	00	00	00	00	00	00	00	00	00	A0	00	00	00		+

On peut voir qu'on a remplacé la hauteur de l'image par FC FF FF FF

cela donne l'image :



On peut voir qu'en passant la hauteur de l'image en négatif, l'image c'est retournée.

3. Pour cette image, il suffit de faire le même calcul que précédemment : on converti la hauteur de l'image en binaire : 00 00 01 A9 -> 0000 0001 1010 1001. Puis on inverse les bits à gauche du premier 1 rencontré dans notre chiffre : 1111 1110 0101 0111. Ce qui donne en hexa FF FF FE 57 et en little endian 57 FE FF FF

Dans le code cela donne ceci :

```
SAE > Image > ImageExempleIndexBMP3_16.bmp
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 42 4D B6 13 02 00 00 00 00 00 76 00 00 00 28 00
00000010 00 00 80 02 00 00 57 FE FF FF 01 00 04 00 00 00
00000020 00 00 40 13 02 00 00 00 00 00 00 00 00 00 10 00
```

On retrouve bien le 57 FE FF FF à l'adresse 0x16

Avec ce code on obtient cette image :



A.6

1. Le poids du nouveau fichier est de 1120 octets alors qu'avant la compression il n'en faisait que 102. On a donc multiplié par 10 la taille du fichier. Cela peut s'expliquer en regardant le nouveau code de cette image :

SAE > Image > ≡ Image4.bmp

⚙	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	42	4D	60	04	00	00	00	00	00	00	36	04	00	00	28	00
00000010	00	00	04	00	00	00	04	00	00	00	01	00	08	00	01	00
00000020	00	00	2A	00	00	00	00	00	00	00	00	00	00	00	00	01
00000030	00	00	00	01	00	00	00	00	FF	00	FF	FF	FF	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

On peut remarquer que l'en tête du fichier a beaucoup changé. En effet on peut voir que la palette de couleur est composée de 256 couleurs (dont toutes sont du noirs sauf les deux premières qui sont rouge et blanc) car on a 8 bits par pixels soit $2^8 = 256$ couleurs. De plus à l'adresse du nombre de couleur utilisées on trouve qu'il y en a 0 soit le maximum. Enfin on peut voir que le nombre de bits par pixels a diminué, on est passé de 24 bits par pixels à 8 bits par pixels.

2. L'offset du début des pixels se trouve à l'adresse 0x0A. Cet offset nous donne une adresse :0x436

Le codage des pixels :

00000430	00	00	00	00	00	00	01	00	01	01	01	00	01	01	00	00
00000440	01	01	01	00	01	01	01	00	00	00	01	00	01	01	01	00
00000450	01	01	00	00	01	01	01	00	01	01	01	00	00	00	00	01
00000460	+															

3. On sait que le codage des pixels commence à l'adresse 0x436. Il faut maintenant comprendre comment fonctionne la compression RLE : la compression RLE code nos pixels grâce à deux octets. Le premier octet nous donne le nombre de pixels à coder et le deuxième nous donne la couleur de ce pixel (sa position dans la palette de couleur). Par exemple le premier couple d'octet que l'on rencontre est : 01 00. Le premier octet dit que l'on va coder 1 fois un pixel et le deuxième octet dit que c'est la première couleur

de la palette que l'on va coder (ici le rouge) : en effet, si on regarde la palette de couleur le rouge est bien la première couleur de la palette et on la code bien une seule fois (on commence en bas à gauche de l'image). ce processus se répète pour chaque pixels (01 01 signifie 1 pixel de couleur blanche)



A.7

1. L'entête de l'image convertie :

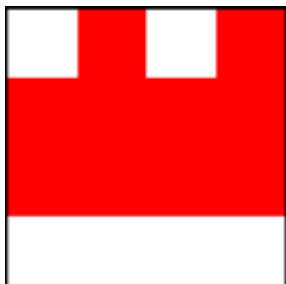
SAE > Image > ≡ Image5.bmp																	
⚙	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	42	4D	4E	04	00	00	00	00	00	00	36	04	00	00	28	00	
00000010	00	00	04	00	00	00	04	00	00	00	01	00	08	00	01	00	
00000020	00	00	18	00	00	00	00	00	00	00	00	00	00	00	00	01	
00000030	00	00	00	01	00	00	00	00	FF	00	FF	FF	FF	00	00	00	
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Le codage des pixels de cette image :

00000420	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000430	00	00	00	00	00	00	04	01	00	00	04	00	00	00	04	00	
00000440	00	00	01	01	01	00	01	01	01	00	00	00	00	01	+		

Le fichier fait 1102 octets. C'est 18 octets de moins que le fichier précédent car si on regarde le codage des pixels, on remarque qu'il y a moins de données pixels qu'avant.

Voici l'image obtenue :



- La compression étant la même, la manière de coder les pixels n'a pas changé. Prenons la première ligne de pixels blanc en bas à gauche de l'image. Ces pixels sont codés comme ceci : 04 01. Le premier octet (04) signifie que l'on va coder 4 pixels et le deuxième octet (01) signifie qu'on va coder la deuxième couleur de la palette (On commence à 0 donc c'est bien la deuxième), ici cette couleur correspond au blanc. C'est la même chose pour les 2 lignes rouges qui suivent : 04 00, 4 pixels codés en rouge (00

correspondant à la première couleur de la palette soit le rouge ici). Pour la dernière ligne, c'est la même façon qu'à la question précédente 01 01 un pixel de couleur blanche, 01 00 un pixel de couleur rouge. A noter qu'on sépare chaque ligne de pixels par deux octets à zéro.

A.8

Le codage des pixels de l'image obtenue :

```
00000430  00 00 00 00 00 00 00 02 01 01 00 01 01 00 00 04 00
00000440  00 00 04 00 00 00 00 01 01 01 00 01 01 01 00 00 00
00000450  00 01 +
```

Par rapport à l'image d'avant, on a modifié la première ligne de pixels. On a remplacé 04 00 00 00 par 02 01 01 00 01 01 00 00. 02 01 pour avoir les deux premier pixels blancs, 01 00 pour ensuite avoir 1 pixel rouge et enfin 01 01 pour avoir 1 pixel blanc.

Cela donne l'image :



L'entête de l'image :

```
SAE > Image > ≡ Image6.bmp
⚙️          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000    42 4D 52 04 00 00 00 00 00 00 36 04 00 00 28 00
00000010    00 00 04 00 00 00 04 00 00 00 01 00 08 00 01 00
00000020    00 00 18 00 00 00 00 00 00 00 00 00 00 00 00 01
00000030    00 00 00 01 00 00 00 00 FF 00 FF FF FF 00 00 00
```

La seule chose qu'il a fallu changer par rapport à l'image5 est la taille de l'image car on a rajouté des données pixels, donc la taille n'était plus la même. Ici la nouvelle image fait 1106 octets soit 452 en hexa et 52 04 00 00 en little endian.

A.9

L'image obtenue :



L'en tête de l'image :

```
SAE > Image > ≡ Image7.bmp
⚙️      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  42 4D 52 04 00 00 00 00 00 00 36 04 00 00 28 00
00000010  00 00 04 00 00 00 04 00 00 00 01 00 08 00 01 00
00000020  00 00 18 00 00 00 00 00 00 00 00 00 00 00 00 01
00000030  00 00 00 01 00 00 00 00 FF 00 FF FF FF 00 FF 00
00000040  00 00 00 FF 00 00 00 00 00 00 00 00 00 00 00 00
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

J'ai remplacé deux couleurs noires de la palette par les couleurs vert et bleu dans la palette, ici on trouve le bleu à l'adresse 0x3E (FF 00 00 00) et le vert à l'adresse 0x41 (00 FF 00 00).

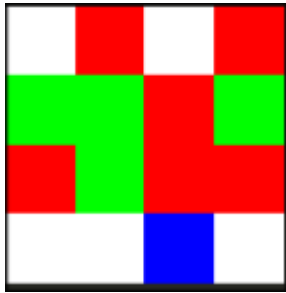
Le codage des pixels :

```
00000430  00 00 00 00 00 00 02 01 01 02 01 01 00 00 04 00
00000440  00 00 04 03 00 00 01 01 01 00 01 01 01 00 00 00
00000450  00 01 +
```

Pour la première ligne (en bas), il fallait changer le pixel qui était rouge en bleu. Pour cela j'ai changé le deuxième couple d'octets à l'adresse 0x438. J'ai donc remplacé le couple 01 00 par le couple 01 02. 01 pour 1 fois un pixel et 02 pour la deuxième couleur de la palette, ici c'est le bleu. Enfin j'ai changé le couple 04 00 à l'adresse 0x442 en 04 03. 04 pour quatre pixels et 03 pour la 3ème couleur de la palette ici le vert.

A.10

- 1. L'image obtenue :



L'entête de l'image:



Dans cette entête, seule la taille a été modifiée par rapport à l'image précédente car des données pixels on été ajoutées cela a donc fait passé la taille de 1106 octets à 1114 octets soit 45A en hexa et 5A 04 00 00 en little endian sur 32 bits.

Le codage des pixels :

```
00000430  00 00 00 00 00 00 00 02 01 01 02 01 01 00 00 01 00
00000440  01 03 02 00 00 00 00 02 03 01 00 01 03 00 00 01 01
00000450  01 00 01 01 01 00 00 00 00 00 01 +
```

Par rapport à l'image précédente, ce sont les lignes 2 et 3 qui on été modifiées. Pour la deuxième ligne (en partant du bas), on a 01 00 01 03 02 00. Le premier couple d'octet 01 00 veut dire qu'on code un seul pixel avec la première couleur de la palette (le rouge ici). Le deuxième couple d'octet 01 03 veut dire qu'on code un pixel avec la 3ème couleur de la palette (ici le vert). Enfin le dernier couple d'octet 02 00 veut dire que l'on code deux pixels avec la première couleur de la palette (le rouge). Pour la 3ème ligne on a 02 03 01 00 01 03. Le premier couple d'octet 02 03 veut dire que l'on code deux pixels avec la 3ème couleur de la palette (ici le vert). Le deuxième couple d'octet 01 00 veut dire que l'on code un pixel avec la première couleur de la palette (le rouge) et le dernier couple d'octet 01 03 veut dire que l'on code un pixel avec la 3ème couleur de la palette.

2. Je n'ai pas puait text à faire la deuxième partie de cette question par manque de temps.

B.1

Pour créer la transposée d'une image il faut inverser les lignes et les colonnes. Voici le programme qui permet de faire cela :

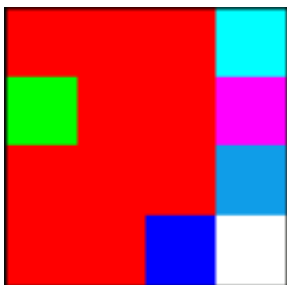

```

1  from PIL import Image
2
3  image = Image.open("SAE/Image/images/ImageTest.bmp")
4  image_transpose = image.copy()
5
6  def transpose_image(image):
7      for ligne in range(image_transpose.size[1]):
8          for colone in range(image_transpose.size[0]):
9              pixel = image.getpixel((colone, ligne))
10             image_transpose.putpixel((ligne, colone), (pixel))
11     return image_transpose.save("SAE/Image/images/Imageout0.bmp")
12 transpose_image(image)

```

Tout d'abord, on ouvre l'image à transposer et on en fait une copie (lignes 3 et 4). Ensuite, on définit une fonction qui va parcourir l'image ligne par ligne en partant en haut à droite. c'est ce que font les deux boucles for. La ligne 9 va s'occuper de récupérer la valeur de chaque pixel que l'on rencontre. C'est-à-dire qu'elle va stocker un tuple qui va stocker la couleur comme ceci : (R,G,B). Quand on a récupéré un pixel, on le met sur l'image copiée et on inverse ligne et colonne dans "putpixel". Quand on a fini de parcourir l'image, on enregistre l'image transposée (ligne 11).

cela donne l'image :



B.2

```

1  from PIL import Image
2
3  coloneInverse = 0
4
5  image = Image.open("SAE/Image/images/hall-mod_0.bmp")
6  image_inverse = image.copy()
7
8  def inverse_image(image):
9      for ligne in range(image.size[1]):
10         for colone in range(image.size[0]):
11             coloneInverse = image.size[0]-(colone+1)
12             pixel = image.getpixel((coloneInverse, ligne))
13             image_inverse.putpixel((colone, ligne), (pixel))
14     return image_inverse.save("SAE/Image/images/Imageout1.bmp")
15 inverse_image(image)

```

Pour inverser une image comme dans un miroir il faut que le dernier pixel d'une ligne, devienne le premier pixel de cette ligne et inversement (le premier doit devenir le dernier). Il faut faire cela pour chaque ligne de l'image. On va donc commencer par définir une variable "colonneInverse" qui stockera l'indice de la colonne opposée du pixel qu'on est entrain de lire. Comme précédemment, on commence par ouvrir l'image à inverser et on en fait une copie (lignes 5 et 6). Ensuite, on parcourt l'image par lignes et par colonnes avec les deux boucles for. La ligne 11 permet de parcourir l'image de droite à gauche. En effet, on prend la longueur de l'image à laquelle on enlève la colonne où on se trouve en lui ajoutant 1 car on commence à 0. Cela permet d'aller récupérer le pixel opposé à celui que l'on regarde. Quand on a cet indice, on va aller chercher la valeur du pixel à cet indice et on le met sur l'image copiée (ligne 12 et 13). Enfin on enregistre l'image copiée.

cela donne l'image :



B.3

```

1  from PIL import Image
2
3  image = Image.open("SAE/Image/images/ImageExemple.bmp")
4  image_grise = image.copy()
5  pixel_gray = ()
6
7  def niveau_de_gris(image):
8      for ligne in range(image.size[1]):
9          for colone in range(image.size[0]):
10             pixel = image.getpixel((colone, ligne))
11             pixel_gray = ((pixel[0]+pixel[1]+pixel[2])//3, (pixel[0]+pixel[1]+pixel[2])//3, (pixel[0]+pixel[1]+pixel[2])//3)
12             image_grise.putpixel((colone, ligne), (pixel_gray))
13         return image_grise.save("SAE/Image/images/Imageout2.bmp")
14     niveau_de_gris(image)

```

les lignes 3 et 4 permettent d'ouvrir l'image que l'on veut passer en niveau de gris et d'en faire une copie. La variable pixel_gray à la ligne 5 va permettre de stocker la valeur d'un pixel passé en niveau de gris. Ensuite dans la fonction, on parcourt l'image de gauche à droite en partant d'en haut à gauche. Pour chaque indice (colone, ligne) on récupère la valeur du pixel grâce à la ligne 10 puis on passe ce pixel en niveau de gris grâce à la formule donnée : $\text{Gris} = (R + V + B) / 3$. On fait cela pour le rouge, le vert et le bleu (ligne 11). Après avoir passé le pixel en gris on le met sur la copie de l'image originale (ligne 12). Enfin quand on a fini de parcourir l'image originale, on enregistre l'image copiée.

cela donne l'image :



B.4

```

1  from PIL import Image
2
3  image = Image.open("SAE/Image/images/IUT-Orleans.bmp")
4  image_noir_blanc = image.copy()
5  ValeurMoyenne = (255*255*3)/2
6
7  def image_noir_et_blanc(image):
8      for ligne in range(image.size[1]):
9          for colone in range(image.size[0]):
10             pixel = image.getpixel((colone, ligne))
11             BlackOrWhite = ((pixel[0]*pixel[0]+pixel[1]*pixel[1]+pixel[2]*pixel[2]))
12             if BlackOrWhite > ValeurMoyenne:
13                 image_noir_blanc.putpixel((colone,ligne), (255,255,255))
14             else:
15                 image_noir_blanc.putpixel((colone,ligne), (0,0,0))
16         return image_noir_blanc.save("SAE/Image/images/Imageout3.bmp")
17     image_noir_et_blanc(image)

```

Pour passer l'image en noir en blanc, on doit utiliser cette formule : $(R * R + V * V + B * B) > 255 * 255 * 3/2$. A la ligne 5 on définit la deuxième partie de la formule qui nous servira de comparaison pour savoir si l'on met un pixel noir ou blanc. Quand on récupère la valeur d'un pixel (ligne 10), on calcule la partie gauche de la formule. Ensuite on compare les deux parties entre elles, si la partie gauche est supérieure à la partie droite, on met un pixel blanc sinon, on met un pixel noir.

cela donne l'image :



B.5

```
1  from PIL import Image
2
3  def trouver(i):
4      return i%2
5
6  def cacher(i,b):
7      return i-(i%2)+b
8
9  #-----
10
11 image_base = Image.open("SAE/Image/images/hall-mod_0.bmp")
12 image_hote = image_base.copy()
13
14 def valeur_rouge_pair(image):
15     for ligne in range(image.size[1]):
16         for colone in range(image.size[0]):
17             pixel = image.getpixel((colone, ligne))
18             Rouge_pair = pixel[0]-(pixel[0]%2)
19             image.putpixel((colone,ligne), (Rouge_pair, pixel[1], pixel[2]))
20     return image.save("SAE/Image/images/Imageout_steg_0.bmp")
21 valeur_rouge_pair(image_hote)
```

la fonction `valeur_rouge_pair` va mettre toutes les valeurs rouges des pixels à une valeur paire. Ici on utilise la formule: $\text{valeur_R} = \text{valeur_R} - (\text{valeur_R} \% 2)$. Cette fonction va donc parcourir l'image hote, et pour chacun de ses pixels, elle va passer la valeur rouge du pixel à une valeur paire. c'est ce que font les lignes 17 et 18. après avoir transformé les valeurs, elle va mettre les nouveaux pixels dans la copie de l'image faite à la ligne 12. Enfin, elle va enregistrer la nouvelle image.

cela donne l'image :



Maintenant que chaque pixel de la nouvelle image réserve une place dans ses unités pour un bit de l'image à cacher, il faut cacher cette image.

```

23 #-----
24
25 image_a_cacher = Image.open("SAE/Image/images/Imageout3.bmp")
26 def cacher_image(image_hote, image_a_cacher):
27     for ligne in range(image_a_cacher.size[1]):
28         for colone in range(image_a_cacher.size[0]):
29             pixel_a_cacher = image_a_cacher.getpixel((colone, ligne))
30             pixel_hote = image_hote.getpixel((colone, ligne))
31             if pixel_a_cacher == (0,0,0):
32                 image_hote.putpixel((colone, ligne), (cacher(pixel_hote[0], 1), pixel_hote[1], pixel_hote[2]))
33             else:
34                 image_hote.putpixel((colone, ligne), image_hote.getpixel((colone, ligne)))
35     return image_hote.save("SAE/Image/images/Imageout_steg_1.bmp")
36 cacher_image(Image.open("SAE/Image/images/Imageout_steg_0.bmp"), image_a_cacher)
37
38 #-----

```

La fonction `cacher_image` va prendre deux arguments, une image hôte et l'image à cacher dans l'image hôte. Elle va parcourir l'image à cacher et récupérer la valeur des pixels de l'image à cacher mais aussi ceux de l'image hôte (lignes 29 et 30). Une fois les valeur récupérées, elle va tester si le pixel de l'image à cachée est un pixel noir c'est-à-dire qu'il est égal à (0,0,0). Si il est noir, on utilise la méthode `putpixel` sur l'image hôte en utilisant la fonction `cacher` pour mettre la valeur rouge du pixel de l'image hôte a une valeur impaire. Si le pixel n'est pas noir, on met tout simplement le pixel de l'image hôte que l'on est entrain de lire (lignes 31 à 34). L'image est donc cachée de cette manière : si la valeur rouge d'un pixel de l'image hôte est impaire, c'est un pixel noir, sinon c'est un pixel blanc.

cela donne l'image :



Mais comment savoir que l'image est bien cachée et au bon endroit ? Pour la révéler, il suffit d'augmenter la valeur rouge quand on cache un pixel noir dans l'image hôte. Cela se fait à la ligne 32.

```
image_hote.putpixel((colone, ligne), (cacher(pixel_hote[0], 1),
pixel_hote[1], pixel_hote[2]))
```

En mettant 60 à la place du 1 ici : `(cacher(pixel_hote[0], 60), 1)` On obtient cette image:



Sur cette image on voit bien apparaître le logo de l'iut ce qui montre que l'image est bien cachée.

```
40 image_hote = Image.open("SAE/Image/images/Imageout_steg_1.bmp")
41 image_cachee = image_hote.copy()
42
43 def trouver_image(image_hote):
44     for ligne in range(image_hote.size[1]):
45         for colone in range(image_hote.size[0]):
46             pixel = image_hote.getpixel((colone, ligne))
47             if trouver(pixel[0]) == 0:
48                 image_cachee.putpixel((colone, ligne), (255, 255, 255))
49             else:
50                 image_cachee.putpixel((colone, ligne), (0, 0, 0))
51     return image_cachee.save("SAE/Image/images/Imageout3trouve.bmp")
52 trouver_image(image_hote)
```

La dernière fonction `trouver_image` va permettre de retrouver et recréer l'image cachée. La fonction va donc parcourir l'image hôte, récupérer la valeur des pixels qu'elle rencontre et va tester grâce à la fonction `trouver`, si

la valeur rouge du pixel est paire ou non (si la valeur modulo 2 est égale à 0). Dans notre cas, si la valeur est paire, c'est un pixel blanc et si elle est impaire c'est un pixel noir.

Cela donne cette image :

