

# Rapport de soutenance

Samy ABOU AL TOUT

6 avril 2022



## Table des matières

<b>1</b>	<b>Le format BITMAP</b>	<b>3</b>
<b>2</b>	<b>Filtre de convolutions</b>	<b>5</b>
<b>3</b>	<b>Images RGB</b>	<b>8</b>
<b>4</b>	<b>Autres éléments utilisés</b>	<b>10</b>
<b>5</b>	<b>Conclusion</b>	<b>11</b>

# 1 Le format BITMAP

La plupart d'entre nous connaissent diverses techniques de traitement d'images, sur différentes plateformes telles que MATLAB, SciLAB, etc. Ici, dans cette série d'articles de blog, je vais me concentrer uniquement sur le traitement d'images en utilisant le langage C. Parfois, les plates-formes mentionnées ci-dessus ne sont pas disponibles pour nous, et il peut y avoir un besoin de traiter des images dans le langage le plus basique, comme le C.

Il existe plusieurs formats d'images largement disponibles tels que JPEG, PNG, TIFF, BMP, etc. Comme le BMP est plus simple à comprendre et à décoder que les formats compressés comme le JPEG, le BMP sera privilégié. Comme le BMP est moins complexe à comprendre et à décoder que les formats compressés comme le JPEG, ce sera le BMP qui sera utilisé pour le projet. Comme je n'ai pas l'intention d'utiliser une bibliothèque, il faudra traiter tous les en-têtes de l'image et le moindre octet d'information par du code, le BMP est donc très pratique à utiliser.

En général, toute image est structurée de la manière suivante :

\*En-tête de l'image

\*Table des couleurs (si elle existe)

\*Données de l'image.

De plus, l'image BMP est structurée de la même manière. Elle comporte un en-tête d'image de 54 octets, une table des couleurs de 1024 octets si elle est présente, et le reste constitue les données de l'image.

offset	size	description
0	2	signature, must be 4D42 hex
2	4	size of BMP file in bytes (unreliable)
6	2	reserved, must be zero
8	2	reserved, must be zero
10	4	offset to start of image data in bytes
14	4	size of BITMAPINFOHEADER structure, must be 40
18	4	image width in pixels
22	4	image height in pixels
26	2	number of planes in the image, must be 1
28	2	number of bits per pixel (1, 4, 8, or 24)
30	4	compression type (0=none, 1=RLE-8, 2=RLE-4)
34	4	size of image data in bytes (including padding)
38	4	horizontal resolution in pixels per meter (unreliable)
42	4	vertical resolution in pixels per meter (unreliable)
46	4	number of colors in image, or zero
50	4	number of important colors, or zero

La table des couleurs est un bloc d'octets (une table) listant les couleurs utilisées par l'image. Chaque pixel dans une image couleur indexée est décrit par un nombre de bits (1, 4 ou 8) qui est un index d'une seule couleur décrite par cette table. L'objectif de la palette de couleurs dans les bitmaps en couleurs indexées est d'informer l'application de la couleur réelle à laquelle correspond chacune de ces valeurs d'index. L'objectif de la table des couleurs dans les bitmaps non indexés (non palettisés) est de répertorier les couleurs utilisées par le bitmap à des fins d'optimisation sur les dispositifs ayant une capacité d'affichage des couleurs limitée et pour faciliter la conversion future à différents formats de pixels et la palétisation.

En termes simples, la colorTable nous aide à identifier la nuance de la couleur en fonction de sa valeur. Elle sert de table de consultation des couleurs pour cette image particulière.

Le reste des octets, une fois que nous avons lu les 54 octets de l'en-tête de l'image et les 1024 octets de colorTable, sont toutes les données de l'image, c'est-à-dire les informations sur les pixels.

```
// extract image height, width and bitDepth from imageHeader
int width = *(int*)&byte[18];
int height = *(int*)&byte[22];
int bitDepth = *(int*)&byte[28];
```

Après avoir lu les 54 octets, on va extraire de ces 54 octets (comme dans le code ci dessus) les données qui nous intéressent comme la largeur, la longueur et le bitDepth.

## 2 Filtre de convolutions






Les filtres de convolution (également connus sous le nom de noyaux) sont utilisés avec les images pour le flou, la netteté, le gaufrage, la détection des bords, etc. Ceci est accompli en effectuant une convolution entre un noyau et une image. Les noyaux sont généralement des matrices 3x3 et le processus de convolution peut être exprimé mathématiquement comme cela :

$$g(x, y) = \omega * f(x, y) = \sum_{dx=-a}^a \sum_{dy=-b}^b \omega(dx, dy) f(x + dx, y + dy)$$

Où  $g(x,y)$  est l'image filtrée de sortie,  $f(x,y)$  est l'image d'entrée et  $w$  est le filtre kernel.

Selon les valeurs des filtres, la convolution peut avoir une variété d'effets. Certains de ces filtres ont également un nom.

Ci dessous les plus connus :

Name	Kernel	Image Result
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Mean Blur	$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$	
Laplacian	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
Gaussian Blur	$\begin{bmatrix} 1/16 & 2/16 & 1/16 \\ 2/16 & 4/16 & 2/16 \\ 1/16 & 2/16 & 1/16 \end{bmatrix}$	

Pour l'implémentation, voici le pseudo-code (source Wikipedia) utilisé pour mettre en place la convolution pour le traitement d'image :

```

for each image row in input image:
  for each pixel in image row:

    set accumulator to zero

    for each kernel row in kernel:
      for each element in kernel row:

        if element position corresponding* to pixel position then
          multiply element value corresponding* to pixel value
          add result to accumulator
        endif

    set output image pixel to accumulator

```

Voici le code que j'ai écrit en C pour le filtre de flou gaussien :

```

float v=1.0 / 9.0;                                //blurring kernel
float kernel[3][3]={v,v,v},
                  {v,v,v},
                  {v,v,v}};

for(x=1;x<height-1;x++)
{
  for(y=1;y<width-1;y++)
  {
    float sum0= 0.0;
    float sum1= 0.0;
    float sum2= 0.0;
    for(i=-1;i<=1;++i)
    {
      for(j=-1;j<=1;++j)
      {
        // matrix multiplication with kernel with every color plane
        sum0=sum0+(float)kernel[i+1][j+1]*buffer[(x+i)*width+(y+j)][0];
        sum1=sum1+(float)kernel[i+1][j+1]*buffer[(x+i)*width+(y+j)][1];
        sum2=sum2+(float)kernel[i+1][j+1]*buffer[(x+i)*width+(y+j)][2];
      }
    }
    out[(x)*width+(y)][0]=sum0;
    out[(x)*width+(y)][1]=sum1;
    out[(x)*width+(y)][2]=sum2;
  }
}
}

```

Pour le code que j'ai écrit, j'ai implémenté trois variables sum pour chaque "color plane" (vu qu'on est sur du RGB, il faut traiter chaque "color plane").

On applique le pseudo code cité avant et cela permet d'obtenir un filtre de convolution sur l'image.

Pour voir comment cela fonctionne, on commence par inspecter une image en noir et blanc. La matrice à gauche contient des nombres, entre 0 et 255, qui correspondent chacun à la luminosité d'un pixel dans l'image d'un visage. La grande image pixelisée à droite est celle qu'on va traiter par exemple.

```
255 255 247 245 244 253 247 245 136 151 255 255 255 255 255 255 254 257 231 255 254 254 255 255 254 255 252 255 255 254 255 247
244 161 137 244 254 255 254 255 118 103 259 228 155 153 236 193 74 52 66 173 255 254 254 255 255 255 254 255 254 253 244 164
192 154 75 250 249 255 255 255 110 96 84 61 35 44 89 53 44 45 43 54 140 213 253 255 255 255 255 245 167 186 176 223
30 159 96 143 223 255 255 252 117 75 41 35 31 24 25 36 45 44 44 46 81 118 148 234 252 254 255 248 231 248 255 254
67 69 107 195 236 255 255 255 104 25 34 35 29 20 25 34 32 30 32 34 53 85 100 142 231 242 247 249 255 255 255 255
55 51 45 134 218 251 255 252 51 12 26 33 24 24 46 75 82 76 71 66 58 53 67 90 136 238 258 158 253 245 249 255
79 58 56 75 224 255 255 118 11 27 74 99 91 106 140 162 173 173 173 172 158 137 92 46 79 167 217 206 254 232 233 255
38 43 47 52 147 255 259 56 41 81 129 145 160 169 169 172 176 179 179 179 177 177 172 110 31 82 259 238 255 244 249 255
40 40 33 36 30 245 171 32 65 110 139 145 151 162 171 174 179 179 182 184 187 183 173 162 71 45 167 255 254 255 254 255
37 44 44 31 69 250 158 36 70 129 143 142 153 162 171 175 177 178 182 191 194 188 180 170 120 51 137 255 254 255 254 255
34 45 51 64 116 237 181 53 116 138 140 143 154 164 176 176 177 183 186 185 183 176 140 66 141 254 252 235 249 255
34 36 52 74 71 188 156 63 131 134 144 155 160 161 173 179 179 189 193 190 185 187 182 156 93 148 250 254 214 247 255
32 38 52 54 159 250 126 57 129 138 138 140 151 156 166 168 171 178 180 187 186 185 183 180 162 136 242 255 255 254 254
36 32 72 129 212 228 115 65 121 104 102 104 94 103 134 158 170 162 125 108 121 143 155 160 191 104 134 230 253 253 255 251
61 82 116 167 179 247 124 60 101 90 111 119 103 81 94 147 191 178 126 98 123 153 147 161 200 92 100 222 207 167 227 215
144 178 167 231 210 232 170 67 115 88 76 62 83 85 88 139 162 160 135 80 53 99 141 165 201 97 79 162 245 235 248 249
127 148 149 195 254 213 197 95 133 122 117 133 126 108 110 139 161 167 167 129 127 148 147 171 168 110 121 228 233 180 215 212
97 112 100 79 85 82 65 75 142 148 151 153 138 125 120 149 191 190 193 175 174 193 198 190 208 127 163 239 219 149 198 195
63 83 109 134 129 106 39 78 132 148 155 159 139 111 124 164 195 200 186 192 191 195 200 202 200 143 217 253 249 242 238 234
69 78 79 113 97 74 43 106 127 142 152 155 125 97 112 150 185 194 174 183 196 198 202 208 209 186 247 254 255 254 254 254
72 44 63 59 46 52 49 74 127 137 146 149 152 103 79 90 134 141 168 165 199 207 204 203 216 193 236 244 221 242 236 243
55 20 69 73 59 80 46 74 117 127 144 161 148 124 105 120 156 187 193 182 189 206 201 203 214 194 174 185 197 188 193 193
65 49 77 89 50 68 43 61 159 127 141 147 113 100 121 145 148 169 181 176 181 201 201 205 202 174 166 169 178 183 188 184
82 76 92 79 54 58 37 47 90 121 132 116 89 79 111 148 163 149 122 124 180 197 197 188 178 149 146 152 155 157 159 168
104 107 122 133 105 79 27 33 66 111 122 120 114 114 147 175 190 196 163 151 170 200 187 185 156 145 146 139 127 141 140 145
117 124 127 133 105 21 28 37 88 115 121 126 126 141 142 168 202 212 153 164 186 180 168 154 146 144 149 151 151 147 144
119 118 118 125 128 111 21 29 28 58 100 118 131 140 151 159 186 201 205 192 180 168 149 169 119 144 147 143 140 141 144 148
117 119 120 130 139 106 18 29 44 58 70 102 133 147 168 197 212 215 210 195 177 152 133 195 57 59 126 151 145 143 142 141
115 108 126 134 145 102 27 54 52 38 45 69 103 135 175 189 193 216 206 186 139 111 164 203 74 5 121 151 142 143 142 141
101 108 123 121 132 105 44 40 31 35 57 44 58 101 147 144 138 163 145 94 90 145 196 187 84 48 105 160 142 144 142 145
98 97 97 96 104 76 34 33 30 48 41 49 51 58 74 53 55 65 63 89 150 188 209 156 62 108 140 149 125 133 131 131
102 102 97 88 73 35 30 23 42 50 65 41 90 60 59 51 57 62 123 157 187 205 169 62 96 151 155 151 154 135 135 135
```



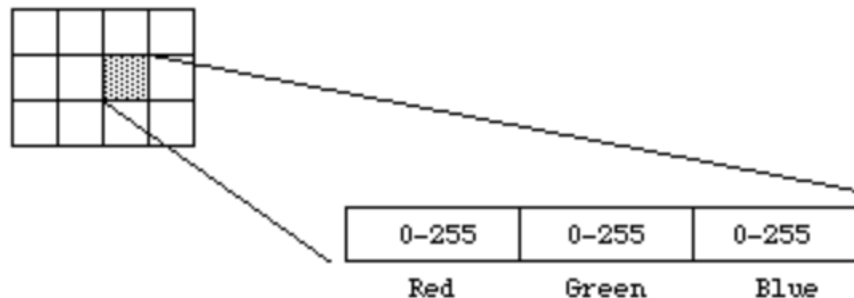
Vu qu'on utilisera des matrices 3\*3 de filtre kernel, pour chaque bloc 3x3 de pixels dans l'image de gauche, nous multiplions chaque pixel par la matrice 3\*3 (filtre kernel), puis on fait la somme. Cette somme devient un nouveau pixel dans l'image de droite. On effectue cela sur toute l'image ce qui nous donnera le résultat que l'on souhaite.

### 3 Images RGB

L'image RVB se différencie du grayscale par la composante couleur. Les niveaux de gris ne contiennent aucune information de couleur. Le principal facteur de différenciation avec les images grayscale est le Bitdepth. Ce dernier nous aide à identifier les composants de couleur présents ou non dans l'image.

Pour les images grayscale, cette valeur sera inférieure ou égale à 8, et pour les images RVB, elle sera supérieure à 8. Il y a une exception pour les images RVB 8 bits également, nous en parlerons plus tard.

Dans la plupart des cas, le Bitdepth du RGB est de 24 bits. Cela signifie que chaque pixel est de 24 bits et que, en le subdivisant, chaque composante de couleur est de 8 bits. Nous avons donc 8 bits de données pour le rouge, 8 bits pour le vert et 8 bits pour le bleu. Dans chaque composante, 0 signifie qu'il n'y a pas de contribution de cette couleur, et 255 signifie qu'il y a une contribution complète de cette couleur. Comme chaque composante a 256 états différents, il y a un total de 16777216 couleurs possibles.



RGB image 24 bits

Il peut même y avoir une image couleur RGB indexée sur 8 bits. Il est plus économique de stocker des bitmaps couleur sans utiliser 3 octets par pixel. Comme pour les bitmaps gris 8 bits, un octet est associé à chaque pixel. Toutefois, cet octet ne contient pas l'information sur la couleur mais un index dans une table de couleurs, appelée palette ou table de couleurs (colorTable).

Il existe aussi des images RGB 32 bits, c'est la même chose que la couleur 24 bits, mais avec une iavec 8 bits appelée canal alpha. Ce canal peut être utilisé pour créer des zones masquées ou représenter la transparence.

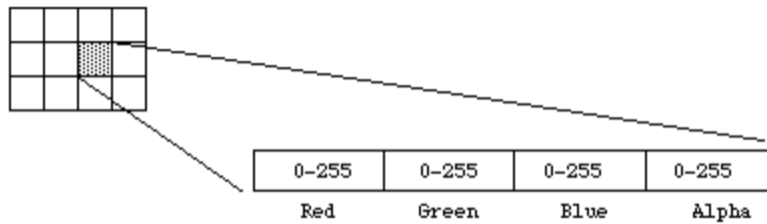
Une autre chose à garder à l'esprit est le stockage des pixels dans le format de fichier BMP. Lorsque les informations sur les pixels commencent, les données sont disposées de la manière suivante,



‘ B0 G0 R0 B1 G1 R1 B2 G2 R2 .... Bn Gn Rn ‘

La composante de couleur est stockée au format BGR et non au format RGB. Par conséquent, la composante bleue du pixel 0 vient en premier, suivie de la composante verte et de la composante rouge. Il faut donc faire très attention lors de la lecture de l'image.

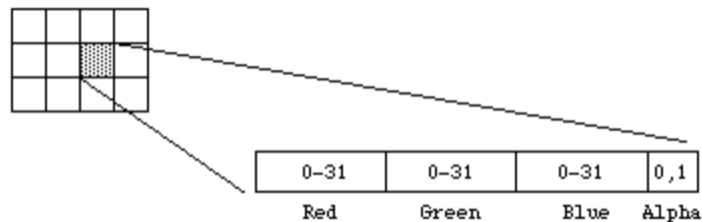
Un autre point intéressant à noter est la disposition des pixels au format BMP. Le format BMP stocke les données des pixels à l'envers ! Cela signifie que le premier pixel que l'on trouve en lisant le fichier représente le dernier pixel (pixel inférieur droit) de l'image que l'on voit visuellement. Par conséquent, si on a besoin de certaines opérations pour une implémentation basée sur la région, il faut vérifier deux fois les pixels que l'on adresse.



RGB image 32 bits

Index	Red	Green	Blue
0			
1			
2	0-255	0-255	0-255
3			
4			
254			
255			

RGB image 8 bits



RGB image 16 bits

## 4 Autres éléments utilisés

J'ai utilisé plusieurs fonctions de base du C pour pouvoir lire les données de l'image correctement et de manière optimisée.

```
for(i=0;i<54;i++) // read the 54 byte header from fIn
{
    byte[i] = getc(fIn);
}
```

Par exemple, dans le code ci dessus, on utilise la fonction "getc" afin de pouvoir récupérer les données du header présent dans le fichier Bitmap.

La fonction de la bibliothèque C int getc(FILE \*stream) récupère le caractère suivant (un caractère non signé) du stream spécifié. Cette fonction renvoie le caractère lu sous la forme d'un char non signé converti en un int ou EOF en cas de fin de fichier ou d'erreur.

```
for(i=0;i<size;i++)
{
    putc(out[i][2],fOut);
    putc(out[i][1],fOut);
    putc(out[i][0],fOut);
}
```

Pour mettre les données de la nouvelle image dans un nouveau fichier bitmap, il faut utiliser la fonction putc (sur le code ci dessus, on le fait pour 3 arrays car on est sur une image RGB).

La fonction de la bibliothèque C int putc(int char, FILE \*stream) écrit un caractère (un caractère non signé) spécifié par l'argument char dans le stream spécifié. Cette fonction renvoie le caractère écrit sous la forme d'un char non signé converti en un int ou EOF en cas d'erreur.

On utilise les fonctions "fopen" et "fclose" pour pouvoir ouvrir et fermer le fichier sans erreur.

## 5 Conclusion

Pour conclure, cela fut assez riche comme projet car il y avait l'aspect algorithmique avec l'effet de challenge de n'utiliser aucune librairie pour le traitement d'image. On avait d'un côté l'aspect algorithmique et mathématique avec la convolution et les matrices  $3 \times 3$ , puis d'un autre côté l'aspect visuel car l'on manipule des images et c'est assez beau comme résultat.