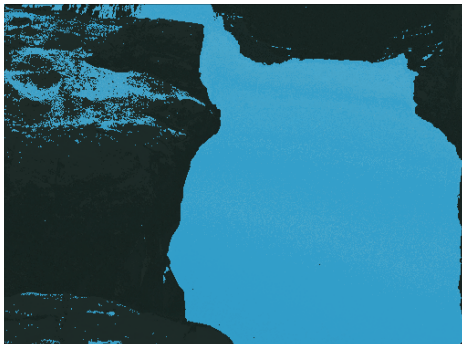
# Compression d'Image avec K-Means

LEMAIRE Théo

December 10, 2023



## Contents

# 1 Exactitude de la Compression

## 1.1 Taux de compresssion

Dans le cadre de la compression avec l'algorithme K-means, le taux de compression est variable, en effet, celui-ci dépendra du nombre de centroides que nous aurons défini au début de notre compression. Quand on parle de compression d'image, le nombre de centroides définis, va induire le nombre de clusters auquels vont se réferer les pixels de l'image, c'est le cas dans l'implémentation réalisée, si on éxecute notre programme avec 10 centroides, après N itérations, on se retrouveras avec au maximum 10 couleurs différentes dans l'image, dans le cas d'une image contenant 10000 pixels, on sera donc sur un taux de compression de 1000%.

Le facteur principal qui influera sur le taux de compression et/ou sur l'efficacité de notre algorithme, est la disparité des couleurs dans notre image, en effet, si on a une image constituée de 10000 pixels d'une meme couleurs, la compression sera à l'apogée de son efficacité pour une seule itération, à l'inverse, pour une image contenant des zones de couleurs avec une disparité élevée, pour une même itération, l'algorithme sera moins performant.

Le taux de compression peux également varier en fonction des facteurs suivants:

- Choix des calculs de distance, si on utilise RGB ou HSV par exemple.
- Nombre d'itérations de l'algorithme, si on atteint pas la convergence avec N couleurs pour N centroides définis, celà signifie dans la plupart des cas que notre nombre d'itérations est insuffisant.
- Seuil de tolérance, nous pouvons appliquer un seuil de tolérance à la disparité des couleurs dans notre compression.

# 2 Structure du projet

## 2.1 Le projet

Le projet est découpé en plusieurs parties:

- ima.h qui contient toutes les définitions des structures ainsi que les prototypes des fonctions.
- modif.c qui contient l'implémentation de k-means.
- main.c qui contient dans le cas 10, l'appel à k-means avec comme paramètre

```
Image * im, int nb_centroids, int min_distance_centroid, int nb_iteration, int dithering_enable
```

## 2.2 Choix des structures

- Color : Cette structure prends en paramètre trois entiers R, G, B, ainsi qu'une frequence, qui représente sa fréquence d'apparition dans l'image, l'attribut islast indique si un élément est le dernier élément d'une liste. Puis il y a 3 attributs qui définissent le centroide attribué pour la couleur (r,g,b, centroide)
- Image : cette structure prends la taille horizontale (sizeX) et la taille verticale de l'image (sizeY), ainsi qu'un GLubyte data qui représente la couleur de l'octet du RGB.

### 2.3 Fonctions

- k means : cette fonction englobe l'ensemble de l'algorithme de compression, elle va se charger d'effectuer la compression d'image en prenant en entrée une image, un nombre de centroides, une distance minimale entre chacuns des centroides (idéalement entre 50 et 150) ainsi que le nombre d'itérations maximal souhaité.
- build CLUT : Construit un tableau de Color représentatif de l'image, avec pour chaque élément un attribut RGB, ainsi que la fréquence d'apparition de celui-ci.
- getKcolorsfromCLUT : Construit un tableau de centroides à partir de la CLUT. Il prends en priorité les couleurs les plus fréquentes dans l'image.
- buildcolortablefromimage : Construit une représentation intermediaire de l'image sous forme d'un tableau de Color. Les seuls paramètres de la struct Color pris en compte dans ce tableau sont R,G et B.
- setcentroidsforimagecolours : Assigne à chacunes des couleurs de notre color table (représentation intermediaire) un centroide en utilisant les attributs r,g,b centroid.
- updatecoloursfromcentroid : Met à jour toutes les couleurs de la représentation intermediaire en faisant la moyenne de chacunes d'entre elles par rapport au centroid auquel elles sont rattachées (cf fonction mean).
- updatecentroidfromcolours : Met à jour l'ensemble des centroides en fonction des couleurs qui sont attribuées à chacun d'entre eux, on fait la moyenne pour chacun des centroides de toutes les couleurs qui lui sont attribués.
- updateimagewithcolorvector : Met à jour l'image à partir de la représentation intermediaire.
- dithering : Tentative de dithering sur l'image, on vient checker si chacunes des couleurs sont audessus ou non d'un certain seuil, puis on la défini en conséquence, le dithering permet "d'adoucir la transition" une fois la compression terminée.

## 3 Choix arbitraires

### 3.1 Arrêt de l'algorithme

J'ai décidé d'implémenter une verification d'évolution des centroides dans mon projet, en effet, si d'une itération à une autre, on se retrouve avec un tableau de centroides qui n'évolue pas, alors on arrête l'algorithme car on considère que l'on a atteint un point de convergence. De plus une autre condition d'arret à été implentée, si on se retrouve après une itération avec N couleurs différentes, si N est inférieur au nombre de centroides souhaité, alors on arrête l'algorithme, car on à atteint le niveau de compression souhaité.

### 3.2 Distance

Le calcul de la distance entre deux couleurs rgb à été d'une importance capitale dans la construction du projet, en effet, actuellement, j'ai laissé le choix entre deux méthodes implémentée:

- Entre 0 et 1,000 avec un simple calcul de distance.

```
int distance_RGB(Color color1, Color color2) {
    float deltaR = color2.r - color1.r;
    float deltaG = color2.g - color1.g;
```

```
        float deltaB = color2.b - color1.b;

        return fabs(sqrt(deltaR * deltaR + deltaG * deltaG + deltaB * deltaB));
    }
```

- Entre 0 et 1,000,000 (on manipule des entiers sur 24 (3*8) bits) avec la fonction ColorToInt qui m'a été fournie qui récupère bits de poids faible/fort.

```
int ColorToInt(Color couleur){ // convertie un RGB en un int
  GLubyte r = couleur.r;
  GLubyte g = couleur.g;
  GLubyte b = couleur.b;

  int bits = 8;
  int rgb = 0;

  int nb = 0;
  for(int i = bits ; i > 0; i --){
    rgb = rgb  | ((1&r)<<nb);
    r = r >> 1;
    nb++;

    rgb = rgb | ((1&g)<<nb) ;
    g = g >> 1;
    nb++;

    rgb = rgb | ((1&b)<<nb) ;
    b = b >> 1;
    nb++;
  }
  return rgb;
}
/* appel avec abs(ColorToInt(r,g,b)-ColorToInt(r',g',b'))*/
```

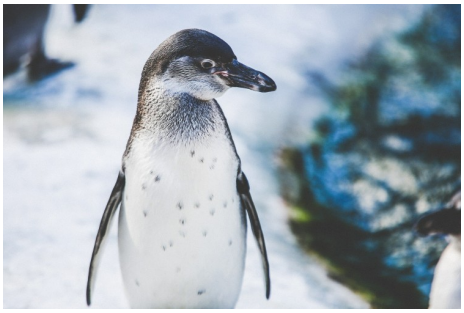
# 4 Fonctionnement général de l'algorithme

```
function k_means{
    CLUT = build_CLUT(image); // Création de la CLUT
    sort_CLUT(CLUT); // Tri de la CLUT
    centroid_list = get_K_colors_from_CLUT(); // Création du tableau de centroides
    color_table = build_color_table_from_image(image); // Création du tableau de couleur
    set_centroids_for_image_colours(); // On attribue à chaques couleurs un centroide.
    for each iteration{ // Pour chaque itération
        for each centroid { // Pour chaque centroide
            update_colours_from_centroid(); // On met à jour les couleurs.
            update_centroid_from_colours(); // On met à jour les centroides.
        }
        update_image_with_color_vector(); // On met à jour l'image à partir du tableau de couleur
```

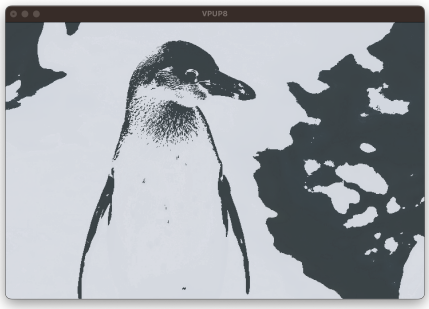
```
        set_centroids_for_image_colours(); // On attribue à chaques couleurs un centroide.

        if(nb_iteration_atteint?) break;
        if(nb_color<=nb_centroids?) break;

        Display(); // On affiche l'image
    }
}
```

# 5 Phase de test !

Test de l'algorithme pour une image de 51000 couleurs différentes. Dans mon implémentation, j'observe une convergence à 6 itérations pour k=100 et à 8 itérations pour k=10. On prendra une distance de 150 pour l'utilisation de la distance RGB classique.

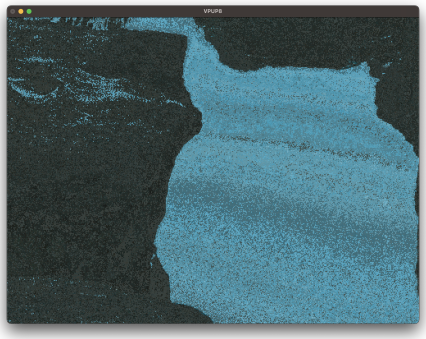


(a) Avec 100 centroides souhaités

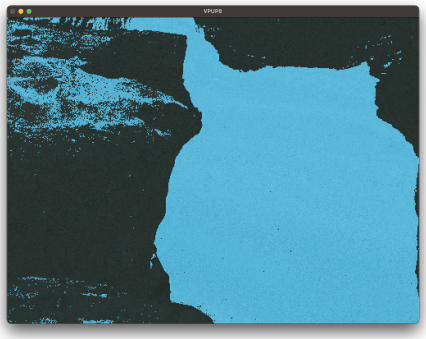


(b) Avec 10 centroides souhaités

Test de l'algorithme pour une image de 29000 couleurs différentes. Dans mon implémentation, j'observe une convergence à 4 itérations pour k=4000 et à 8 itérations pour k=10. On prendra une distance de 150 pour l'utilisation de la distance RGB classique.



(a) Avec 4000 centroides souhaités



(b) Avec 10 centroides souhaités

# 6 Sources

- https://iq.opengenus.org/image-compression-using-k-means/
- https://www.geeksforgeeks.org/image-compression-using-k-means-clustering/
- https://towardsdatascience.com/clear-and-visual-explanation-of-the-k-means-algorithm-applied-to-image-compression-b7fdc547e410