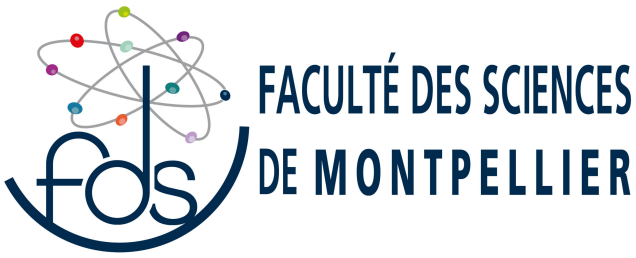
Université de Montpellier

M1 - Imagine
Projet Image-Compression
Compression basée super-pixels

Etudiants :
Valentin LEFEBVRE
Nicolas LUCIANI

Encadrants :
William Puech
Bianca Jansen Van Rensburg

Année 2022-2023



Sommaire

1. État de l'art

(a) Image originale (b) Image segmentée en super-pixels

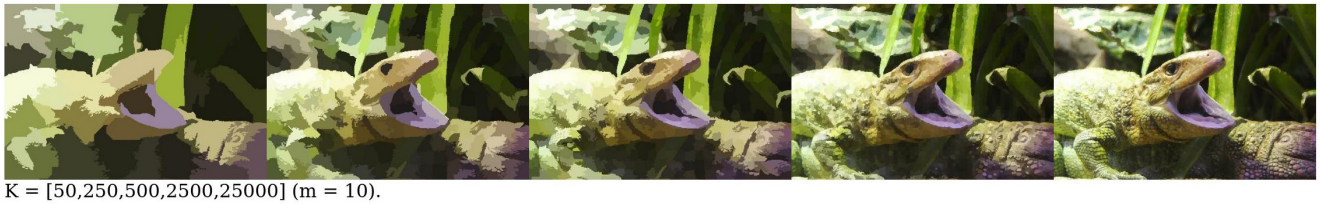
FIGURE 1.1 – État de l'art : segmentation en super-pixels

K = [50,500,5000,25000] (m = 5).

FIGURE 1.2 – État de l'art : compression en super-pixels

K = [50,500,5000,25000] (m = 20).

FIGURE 1.3 – État de l'art : compression en super-pixels



K = [50,250,500,2500,25000] (m = 10).

FIGURE 1.4 – État de l'art : compression en super-pixels

2. Segmentation

2.1 Conversion des pixels de l'espace RGB à l'espace LAB

Afin de pouvoir appliquer notre algorithme de segmentation SLIC pour super-pixels, nous avons tout d'abord besoin de passer notre image de l'espace **RGB** à l'espace **LAB**. L'espace LAB correspond à des couleurs définies par 3 valeurs : Luminosité (luminance) codée en pourcentages ainsi que a et b qui correspondent à l'information colorée (chrominance) où la couleur est définie à partir d'un mélange de vert à magenta (a) et un mélange de bleu à jaune (b).

2.1.1 Conversion de l'espace RGB à l'espace XYZ

```
1  float M[3][3];
2  M[0][0] = 0.4887180; M[0][1] = 0.3106803; M[0][2] = 0.2006017;
3  M[1][0] = 0.1762044; M[1][1] = 0.8129847; M[1][2] = 0.0108109;
4  M[2][0] = 0.0000000; M[2][1] = 0.0102048; M[2][2] = 0.9897952;
5     for (int i=0; i < nTaille3; i+=3)
6       {
7         ImgOut[i]=(M[0][0]*ImgIn[i]+M[0][1]*ImgIn[i+1]+M[0][2]*ImgIn[i+2]);
8         ImgOut[i+1]=(M[1][0]*ImgIn[i]+M[1][1]*ImgIn[i+1]+M[1][2]*ImgIn[i+2]);
9         ImgOut[i+2]=(M[2][0]*ImgIn[i]+M[2][1]*ImgIn[i+1]+M[2][2]*ImgIn[i+2]);
10      }
```

FIGURE 2.1 – Image originale *sheep.ppm*

FIGURE 2.2 – Image convertie dans l'espace XYZ

2.1.2 Conversion de l'espace XYZ à l'espace LAB

```
float delta = 6/29;
float ft(float t){
  if(t>pow(delta,3)) return pow(t, 1/3.);
  else return t/3*pow(delta,2)+4/29;
}
float Xn = 95.0489;
float Yn = 100;
float Zn = 108.8840;
for (int i=0; i < nTaille3; i+=3)
    {
        ImgOut[i]=116*ft(ImgIn[i+1]/Yn)-16.;
        ImgOut[i+1]=500*(ft(ImgIn[i]/Xn)-ft(ImgIn[i+1]/Yn));
        ImgOut[i+2]=200*(ft(ImgIn[i+1]/Yn)-ft(ImgIn[i+2]/Zn));
    }
```

FIGURE 2.3 – Image convertie dans l'espace LAB

2.2 Conversion des pixels de l'espace LAB à l'espace RGB

2.2.1 Conversion de l'espace LAB à l'espace XYZ

```
float Xn = 76.04;
  float Yn = 80;
  float Zn = 87.12;
   for (int i=0; i < nTaille3; i+=3)
     {
      float P = (ImgIn[i]+16)/116;
       ImgOut[i]=Xn*pow((P+ImgIn[i+1]/500),3);
       ImgOut[i+1]=Yn*pow(P,3);
       ImgOut[i+2]=Zn*pow((P-ImgIn[i+2]/200),3);
    }
```

FIGURE 2.4 – Image convertie dans l'espace XYZ

2.2.2 Conversion de l'espace XYZ à l'espace RGB

```
M[0][0] = 3.2404542; M[0][1] = -1.5371385; M[0][2] = -0.4985314;
M[1][0] = -0.9692660; M[1][1] = 1.8760108; M[1][2] = 0.0415560;
M[2][0] = 0.0556434; M[2][1] = -0.2040259; M[2][2] = 1.0572252;
   for (int i=0; i < nTaille3; i+=3)
     {
       ImgOut[i]=(M[0][0]*ImgIn[i]+M[0][1]*ImgIn[i+1]+M[0][2]*ImgIn[i+2]);
       ImgOut[i+1]=(M[1][0]*ImgIn[i]+M[1][1]*ImgIn[i+1]+M[1][2]*ImgIn[i+2]);
       ImgOut[i+2]=(M[2][0]*ImgIn[i]+M[2][1]*ImgIn[i+1]+M[2][2]*ImgIn[i+2]);
     }
```

FIGURE 2.5 – Image convertie dans l'espace RGB

2.2.3 Conversion à l'aide d'une libraire externe

Malheureusement, nous n'avons pas réussi à implémenter ces conversions de manière fonctionnelle et nous avons donc utilisé une librairie externe, appelée **ColorSpace**.

```
/*Converts RGB to CIELAB*/
ColorSpace::Lab convert(OCTET pR, OCTET pG, OCTET pB)
{
```

```
4           ColorSpace::Rgb rgb((double)pR, (double)pG, (double)pB);
5           ColorSpace::Lab lab;
6           rgb.To<ColorSpace::Lab>(&lab);
7           return lab;
8   }
9
10  /*Converts CIELAB to RGB*/
11  ColorSpace::Rgb convert(double l, double a, double b)
12  {
13          ColorSpace::Lab lab(l,a,b);
14          ColorSpace::Rgb rgb;
15          lab.To<ColorSpace::Rgb>(&rgb);
16          return rgb;
17  }
18
19  /*
20  Converts OCTET* Img (in RGB color space) using ColorSpace lib onto array of structs CIELAB to ease next computat
21  */
22  void convert(OCTET* ImgIn, std::vector<CIELAB> &out, int nW, int nH)
23  {
24      for (int i = 0; i<(nW*nH*3); i+=3)
25      {
26          int indexI = floor((i/3)/nW); int indexJ = (i/3) - indexI*nW;
27          ColorSpace::Lab p = convert(ImgIn[i], ImgIn[i+1], ImgIn[i+2]);
28          CIELAB c = {p.l, p.a, p.b, indexI , indexJ};
29          out[indexI * nW + indexJ] = c;
30      }
31  }
32
33  /*
34  Converts CIELAB Image into OCTET* Image (RGB space) to display
35  */
36  void reverseConvert(std::vector<CIELAB> &in, OCTET* ImgOut)
37  {
38          int n = (int) in.size();
39          int index = 0;
40          for (int i = 0; i < n; ++i)
41          {
42                  index = i*3;
43                  ColorSpace::Rgb p = convert(in[i].l, in[i].a, in[i].b);
44                  ImgOut[index] = (unsigned char) p.r;
45                  ImgOut[index+1] = (unsigned char) p.g;
46                  ImgOut[index+2] = (unsigned char) p.b;
47          }
48
49  }
```

La librairie définit une structure pour chacun des espaces couleurs, et implémente les conversions les uns vers les autres. Les formules utilisées sont similaires aux notres. Nous avons préféré utiliser cette librairie car nous l'avons comprise et étudiée. Voici comment nous la faisons fonctionner

2.3 Segmentation en super-pixels

2.3.1 Simple linear iterative clustering (SLIC)

2.3.1.1 SLIC sur une image couleur

```
1  //Segmentation naïve
2  int t = 0; //colonnes rouges
3     int s = 0; //lignes rouges
4     for (int i=0; i < nTaille3; i+=3)
5       {
6         if(s==9*nW*3) s=0;
7         if (t%8==0 || s>8*nW*3) ImgOut[i]=255;
8         else ImgOut[i]=ImgIn[i];
9         ImgOut[i+1]=ImgIn[i+1];
10        ImgOut[i+2]=ImgIn[i+2];
11        t++;s++;
12      }
13
```

FIGURE 2.6 – Progression actuelle (segmentation naïve) de l'implémentation de SLIC

FIGURE 2.7 – Résultat attendu de la segmentation SLIC (cf ***Site externe*** s. d.)

2.3.1.2 SLIC en niveaux de gris

```
1  //Segmentation naïve
2      for(int i = 8 ; i < nH-8 ; i++){
3        for(int j = 8 ; j < nW-8 ; j++){
4          if(i%8==0 && j%8==0){
5            ImgOut[i*nW+j]=255;
6            for(int k = -4 ; k < 4 ; k++){
7              ImgOut[(i-8)*nW+j+k]=255;
8              ImgOut[(i+k)*nW+j+8]=255;
9              ImgOut[(i+8)*nW+j+k]=255;
10             ImgOut[(i+k)*nW+j-8]=255;
11           }
12         }
13         else ImgOut[i*nW+j]=ImgIn[i*nW+j];
14       }
15     }
```

FIGURE 2.8 – Affichage des clusters initiaux (en niveaux de gris)

2.3.1.3 Implémentation de l'algorithme pour des images couleurs

Nous avons défini une structure permettant de stocker plus facilement un pixel dans l'espace LAB.

```
1  struct CIELAB
2  {
3          /*pixel in CIELAB*/
4          int l,a,b,x,y;
5
6          void print(){printf("{%d, %d, %d, %d, %d}\n", l,a,b,x,y);}
7  };
8
9
10 /*
11 Converts OCTET* Img (in CIELAB color space) onto array of structs CIELAB to ease next computations
12 */
13 void convert(OCTET* ImgIn, std::vector<CIELAB> &out, int nW, int nH)
14 {
15         int n = nW * nH;
16         int index;
17         for (int i = 0; i < nH; ++i)
18         {
19                 for (int j = 0; j < nW; ++j)
20                 {
21                         index = i*nW+j;
22                         CIELAB c = {ImgIn[index], ImgIn[index+1], ImgIn[index+2], i, j};
```

```
                        out[index] = c;
                }
        }
}

/*
Converts CIELAB Image into OCTET* Image to display
*/
void reverseConvert(std::vector<CIELAB> &in, OCTET* ImgOut)
{
        int n = (int) in.size();
        int index;
        for (int i = 0; i < n; ++i)
        {
                index = i*3;
                ImgOut[index] = in[i].l;
                ImgOut[index+1] = in[i].a;
                ImgOut[index+2] = in[i].b;
        }
}
```

Nous avons séparé chaque étape de l'algorithme en plusieurs méthodes pour gagner en lisibilité et pouvoir plus facilement gérer les problèmes techniques. En se basant sur les explications présentes dans notre bibliographie, nous avons donc définit ces différentes étapes.

```
/*
Create and retrieves clusters C (vector of 5-uples size k) from Img
*/
void initClusters(OCTET* ImgIn, std::vector<CIELAB> &C, int k, int nW, int nH)
{
        int nTaille = nW * nH;
        int nTaille3 = nTaille * 3;
        for(int i = 0 ; i < nTaille/k; i++) C[i] = {0,0,0,0,0};

        int o = 0;
        for (int i=0; i < nTaille3; i+=3)
        {
                if(i%k==0 && i!=0)
                {
                        C[o].l=(int)ImgIn[i];
                        C[o].a=(int)ImgIn[i+1];
                        C[o].b=(int)ImgIn[i+2];
                        //y = i - x*nW; x = i/nW ;
                        C[o].x = floor((i/3)/nW);
                        C[o].y = (i/3) - C[o].x*nW;
                        //C[o][4]=(i/3)%/nW;
                        //C[o][3]= ImgIn[i] - C[o][4] * nH;
                        o++;
                }
        }
```

```
}
void updateClusters(std::vector<CIELAB> &Img, std::vector<CIELAB> &C, int* labels, int k)
{
        int n = Img.size();
        int total, avgL, avgA, avgB, avgX, avgY;
        for (int i = 0; i < k; ++i)
        {
                total = avgL = avgA = avgB = avgX = avgY = 0;
                for (int j = 0; j < n; ++j)
                {
                        if(labels[j] != i) continue;
                        CIELAB p = Img[j];
                        avgL += p.l; avgA += p.a; avgB += p.b; avgX += p.x; avgY += p.y;
                        total ++;
                }
                if(total != 0)
                {
                        avgL/=total; avgA/=total; avgB/=total; avgX/=total; avgY/=total;
                        CIELAB average = {avgL, avgA, avgB, avgX, avgY};
                        C[i] = average;

                        /*
                        TODO error E calculation
                        */
                }
        }
}
```

Le calcul de l'erreur résiduelle notée E n'a pas encore été implémentée. L'objectif de ce calcul est de limiter le nombre d'itérations de l'algorithme. Cependant, comme décrit dans l'article il est possible de limiter simplement à 10 itérations, et les résultats devraient être intéressants. De plus, nous avons décrit la méthode permettant de calculer la distance d'un pixel au centre du super-pixel auquel il appartient. On donne en paramètre S représentant la taille d'un côté d'une cellule de la grille (un super-pixel sera de taille $S \times S$ au maximum). Le paramètre m représente lui une constante représentant la distance maximale de couleur entre deux pixels de l'espace LAB. La distance finale est donc la combinaison des distances euclidiennes spatiale et couleur entre deux pixels de l'espace LAB.

```
float distance(CIELAB &i, CIELAB &j, int S, int m)
{
        float dc, ds, D;
        /*color euclidian distance*/
        float squared_dc = pow((j.l - i.l), 2) + pow((j.a - i.a), 2) + pow((j.b - i.b), 2);
        dc = sqrt(squared_dc);

        /*spatial euclidian distance */
        float squared_ds = pow((j.x - i.x), 2) + pow((j.y - i.y), 2);
        ds = sqrt(squared_ds);

        float squared_D = squared_dc + pow(ds/(float)S, 2) * m*m;
```

```
        D = sqrt(squared_D);
        return D;
}
```

L'algorithme final ressemblerait donc à la fonction suivante.

```
/*Initialisation*/
std::vector<CIELAB> imgLAB(nTaille);
int k = 8;
std::vector<CIELAB> C(nTaille/k);
initClusters(ImgIn, C, k, nW, nH);
convert(ImgIn, imgLAB, nW, nH);

int* labels;
labels = new int[nTaille];
float* distances;
distances = new float[nTaille];
int S = (int) sqrt(nTaille/k);
int c_x, c_y, index;

int m = 10;

int iteration = 0;
while(iteration <10)
{
    /* for each cluster C[i] */
    for (int c_i = 0; c_i < nTaille/k; ++c_i)
    {
        c_x = C[c_i].x; c_y = C[c_i].y;
        for(int i = c_x-2*S; i < c_x+2*S; i++)
        {
            for (int j = c_y-2*S; j < c_y+2*S; ++j)
            {
                index = i*nW+j;
                float D = distance(C[c_i], imgLAB[index], S, m);
                if(D < distances[index])
                {
                    distances[index] = D;
                    labels[index] = c_i;
                }
            }
        }
    }
    updateClusters(imgLAB, C, labels, k);
    iteration++;
}
for (int i = 0; i < nTaille; ++i) imgLAB[i] = C[labels[i]];
reverseConvert(imgLAB, ImgOut);
ecrire_image_ppm(cNomImgEcrite, ImgOut,  nH, nW);
/*Desallocation memoire*/
```

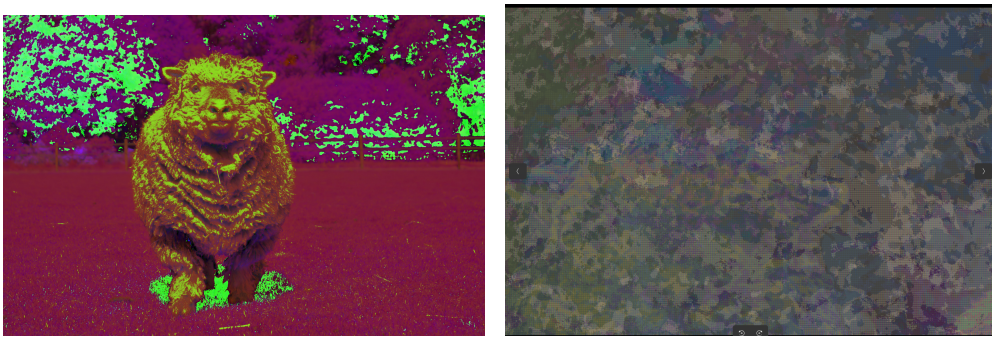
Cependant, un problème se pose lors du calcul des centres des clusters. Il est expliqué sur l'article qu'il faut déplacer les centres des clusters à la valeur du gradient la plus basse dans le 8 voisinage.

2.3.1.4 Déplacement des centres des clusters

Nous avons donc implémenté le déplacement des centres des clusters en ajoutant deux méthodes. La première calcule la norme du gradient dans le 8 voisinage du pixel. La seconde implémente l'algorithme de déplacement des centres ces clusters comme indiqué dans l'article : le minimum de la valeur des normes dans un voisinage 3×3 (c'est à dire le 8 voisinage).

```
1
2  float gradientNorm(OCTET* ImgIn, int i, int j, int nW, int nH)
3  {
4          int x,y;
5          if(i<nH-1) x = ImgIn[(i+1)*nW + j] - ImgIn[i*nW+j];
6          else x = 0;
7          if(j<nW-1) y = ImgIn[i*nW+(j+1)] - ImgIn[i*nW+j];
8          else y = 0;
9          return sqrt(x*x + y*y);
10 }
11
12 void moveClusters(OCTET* ImgIn, std::vector<CIELAB> &C, int S, int nW, int nH)
13 {
14         int o = 0;
15         for (int i = 0; i < nH; i+=S)
16         {
17                 for (int j = 0; j < nW; j+=S)
18                 {
19                         float minG = FLT_MAX;
20                         int minX,minY;
21                         for (int k = i; k < i+S; ++k)
22                         {
23                                 for (int l = j; l < j+S; ++l)
24                                 {
25                                         float norm = gradientNorm(ImgIn, k,l,nW,nH);
26                                         if(norm <= minG)
27                                         {
28                                                 minG = norm;
29                                                 minX = k; minY = l;
30                                         }
31                                 }
32                         }
33                         C[o].x = minX; C[o].y = minY;
34                         o++;
35                 }
36         }
37 }
```

On obtient alors des premiers résultats qui ne sont malheureusement pas concluants.



(a) Image d'entrée (b) Segmentation : 10 itérations

FIGURE 2.9 – Premiers résultats

Cela peut être dû à différents facteurs. Cependant, le plus probable reste le calcul de l'erreur (notée E plus tôt) qui est censé garantir la terminaison de l'algorithme. En effet, nous avons limité notre exécution à 10 itérations, il est possible que ce ne soit pas adéquat. Autrement, il semble parfois il y avoir des problèmes d'allocation mémoire, ou des erreurs d'accès mémoire. Nous tenons donc à reprendre notre implémentation en détail pour tenter de cibler le problème.

Nous avons corrigé nos problèmes de conversion de l'espace RGB à l'espace XYZ et de l'espace XYZ à l'espace LAB. Nous pouvons recommencer l'algorithme de SLIC sur notre nouvelle image LAB et voilà le résultat obtenu :

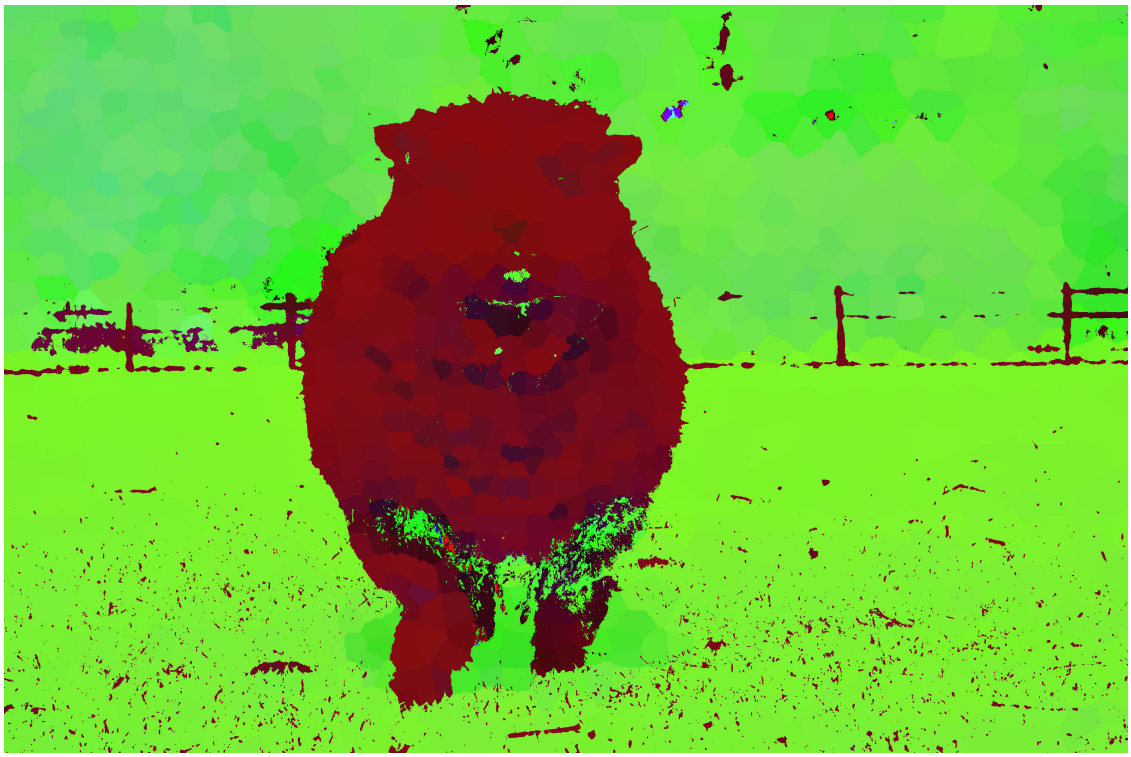


FIGURE 2.10 – Nouvelle Segmentation : 10 itérations

Une fois le code repris, nous avons donc ajouté l'implémentation du calcul de l'erreur E.

```
1
2  void updateClusters(std::vector<CIELAB> &Img, std::vector<CIELAB> &C, int* labels, int k, float &E)
3  {
4          int n = Img.size();
5          double total, avgL, avgA, avgB, avgX, avgY;
6          for (int i = 0; i < k; ++i)
7          {
8                  total = avgL = avgA = avgB = avgX = avgY = 0;
9                  for (int j = 0; j < n; ++j)
10                 {
11                         if(labels[j] != i) continue;
12                         CIELAB p = Img[j];
13                         avgL += p.l;
14                         avgA += p.a;
15                         avgB += p.b;
16                         avgX += p.x;
17                         avgY += p.y;
18                         total ++;
19                 }
20                 if(total != 0)
21                 {
```

```
                        avgL/=total; avgA/=total; avgB/=total; avgX/=total; avgY/=total;
                        CIELAB average = {avgL, avgA, avgB, avgX, avgY};
                        int prevX = C[i].x; int prevY = C[i].y;
                        C[i] = average;

                        E = sqrt((avgX - prevX) * (avgX - prevX) + (avgY - prevY) * (avgY - prevY));
                }
        }
}
```

Cependant, des problèmes persistent. Nous avons compris que ces problèmes provenait de nos changement d'espace couleur. Pour pouvoir avancer convenablement sur le projet, nous avons donc décidé d'utiliser une libraire de changement d'espace couleur. Nous avons repris de cette librairie les fichiers qui nous intéressaient, le lien vers le dépôt git de la librairie est en annexe. Les classes qui nous intéressaient étaient donc les classes définissant un pixel dans un espace couleur, et l'entête implémentant les conversions. Nous avons donc modifié notre fichier impélemntant l'algorithme SLIC pour utiliser correctement ces conversions. Il ne semblait pas convenable de tenter de visualiser les images dans les espace CIELAB ou CIEXYZ. Ainsi, seuls les calculs sont réalisés dans l'espace CIELAB, l'affichage se fait dans l'espace RGB standard. Voici les résultats obtenus et beaucoup plus satisfaisants.

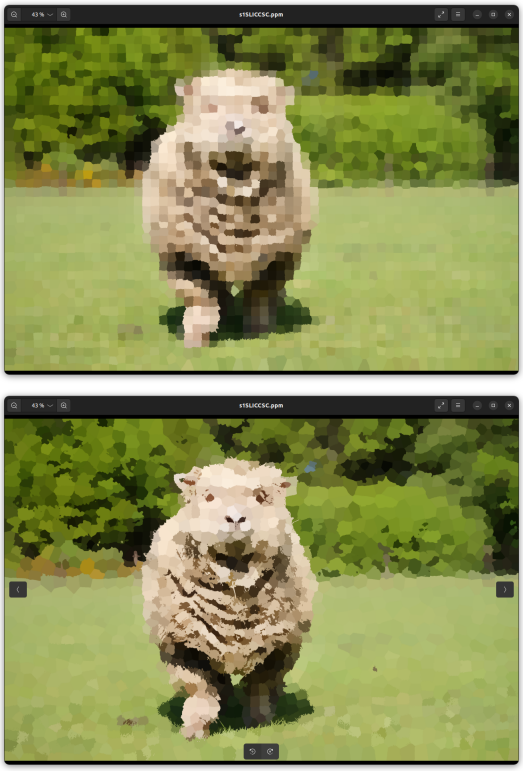


FIGURE 2.11 – Segmentation obtenue après 10 itérations avec des paramètres différents

L'objectif sera donc maintenant d'observer l'effet des paramètres sur la segmentation obtenue, ainsi que de correctement utiliser l'erreur calculée plus tôt comme condition d'arrêt de l'algorithme.

```
float gradientNorm(OCTET* ImgIn, int i, int j, int nW, int nH)
{

        ColorSpace::Lab p,pX,pY;
        int x,y;
        if(i<nH-1)
        {
                p = convert(ImgIn[i*nW*3], ImgIn[i*nW*3+1], ImgIn[i*nW*3+2]);
                pX = convert(ImgIn[(i-nW)*nW*3], ImgIn[(i-nW)*nW*3 + 1], ImgIn[(i-nW)*nW*3 + 2]);

                x = (p.l - pX.l) +
                (p.a - pX.a) +
                (p.b - pX.b);
        }
        else x = 0;
        if(j<nW-1)
        {
                p = convert(ImgIn[i*nW*3], ImgIn[i*nW*3+1], ImgIn[i*nW*3+2]);
                pY = convert(ImgIn[(i+3)*nW*3], ImgIn[(i+3)*nW*3 + 1], ImgIn[(i+3)*nW*3 + 2]);
                y = (p.l - pY.l) +
                (p.a - pY.a) +
                (p.b - pY.b);
        }
        else y = 0;
        return sqrt(x*x + y*y);
}
/*
Create and retrieves clusters C (vector of 5-uples size k) from Img
*/
void initClusters(OCTET* ImgIn, std::vector<CIELAB> &C, int k, int nW, int nH)
{
        int nTaille = nW * nH;
        int nTaille3 = nTaille * 3;

        moveClusters(ImgIn, C, (int) sqrt(nTaille/k), nW, nH);
        for (int i = 0; i < k; ++i)
        {
                int index = C[i].x * nW + C[i].y;
                ColorSpace::Lab p = convert(ImgIn[index], ImgIn[index+1], ImgIn[index+2]);
                C[i].l = p.l;
                C[i].a = p.a;
                C[i].b = p.b;
        }

```

Après avoir ajouté la librairie ColorSpace à notre programme, il a fallu apporter quelques modifications aux fonctions déjà présentes pour pouvoir l'utiliser correctement.

2.3.2 Turbo-Pixels

L'algorithme Turbo-Pixel utilise la même base que l'algorithme SLIC : initialiser des centres de façon régulière sur l'image. Une fois les centres des super-pixels initialisés, la méthode consiste à faire grandir ces super-pixels grâce au calcul d'une vitesse d'évolution. Cette vitesse est calculée à partir de différents paramètres : l'intensité du gradient en niveau de gris, une fonction de courbure permettant de s'assurer que la vitesse d'évolution diminue à l'approche des bords de l'image ou des autres super-pixels. On se servira également de la vitesse calculée à l'itération précédente. L'article présente une formule donnée à partir des contours actifs géodésiques de l'image.

Pour faire grandir les super-pixels, nous avons besoin également de calculer un squelette des régions encore non-assignées à des super-pixels. Le calcul de ce squelette est basé sur un algorithme d'affinement qui n'est pas détaillé explicitement dans l'article. La complexité asymptotique de l'algorithme est présentée comme linéaire en nombre de pixels dans l'image.

3. Méthode de compression

3.1 Compression sans pertes

3.1.1 256-Mean

On peut utiliser une palette de couleur pour stocker l'image selon les indices de couleur de la palette comme vu lors du travail sur l'algorithme de K-means. Le taux de compression est de 3.

FIGURE 3.1 – Compression sans perte du résultat de SLIC

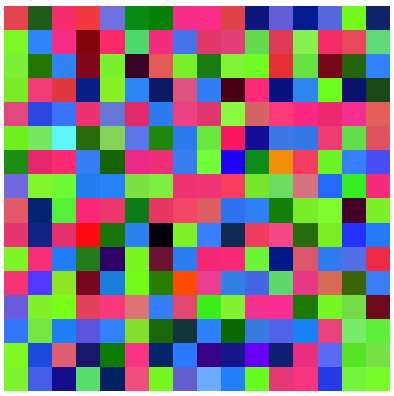


FIGURE 3.2 – Palette des couleurs de l'image SLIC

3.1.2 Compression par plage

```
1
2  std::vector<std::pair<OCTET, int>> RLEComponent(OCTET* Comp, int nW, int nH)
3  {
4          std::vector<std::pair<OCTET, int>> compressedComp;
5          int j,k, count;
6          for (int i = 0; i < nH; ++i)
7          {
8                  j = 0;
9                  while(j<nW)
10                 {
11                         OCTET val = Comp[i*nW+j];
12                         count = 1;
13                         k = j+1;
14                         while(k<nW && Comp[i*nW+k] == val)
15                         {
16                                 count++; k++;
17                         }
18                         compressedComp.push_back(std::make_pair(val, count));
19                         j=k;
20                 }
21         }
22         return compressedComp;
23 }
24
25 int main(int argc, char const *argv[])
26 {
27         char cNomImgLue[250], cNomImgEcrite[250];
28
29         if (argc != 3)
30         {
31                 printf("Usage: ImageInRGB.ppm downscaledImage.ppm\n");
32                 exit (1) ;
33         }
```

```
34
35          sscanf (argv[1],"%s",cNomImgLue) ;
36          sscanf (argv[2],"%s",cNomImgEcrite);
37
38          int nH, nW, nTaille;
39
40          OCTET *ImgIn;
41
42          lire_nb_lignes_colonnes_image_ppm(cNomImgLue, &nH, &nW);
43          nTaille = nH * nW;
44
45          int nTaille3 = nTaille * 3;
46          allocation_tableau(ImgIn, OCTET, nTaille3);
47          lire_image_ppm(cNomImgLue, ImgIn, nH * nW);
48
49          OCTET* R, *G, *B;
50          allocation_tableau(R, OCTET, nTaille);
51          allocation_tableau(G, OCTET, nTaille);
52          allocation_tableau(B, OCTET, nTaille);
53          planR(R, ImgIn, nTaille);
54          planV(G, ImgIn, nTaille);
55          planB(B, ImgIn, nTaille);
56
57          std::vector<std::pair<OCTET,int>> compressedR = RLEComponent(R, nW, nH);
58          std::vector<std::pair<OCTET,int>> compressedG = RLEComponent(G, nW, nH);
59          std::vector<std::pair<OCTET,int>> compressedB = RLEComponent(B, nW, nH);
60
61          std::vector<std::pair<OCTET,int>> compressedImg;
62          compressedImg.insert(compressedImg.end(), compressedR.begin(), compressedR.end());
63          compressedImg.insert(compressedImg.end(), compressedG.begin(), compressedG.end());
64          compressedImg.insert(compressedImg.end(), compressedB.begin(), compressedB.end());
65
66          std::string delimiter = ",";
67          std::string compressedImgString = "";
68
69          // Open a binary file for writing
70          std::ofstream outFile(cNomImgEcrite, std::ios::binary);
71
72          // Write the compressed image data to the file
73          for (const auto& pair : compressedImg) {
74              outFile.write(reinterpret_cast<const char*>(&pair.first), sizeof(pair.first));
75              outFile.write(reinterpret_cast<const char*>(&pair.second), sizeof(pair.second));
76          }
77
78          // Close the file
79          outFile.close();
80          return 0;
81      }
82
```

La compression par plage utilise les séquence répétitives de pixels d'une image pour la

compresser. On affecte un code à chaque suite de pixel, que l'on écrit ensuite dans un fichier binaire. Le taux de compression obtenu par cette méthode varie beaucoup en fonction du paramètre m. En effet, ce paramètre influe sur le poids de la distance spatiale des pixels au centre de leur clusters, ces derniers se voient être plus ou moins uniforme une fois la segmentation faite. L'uniformité de ces clusters garantit des séquences répétitives de pixels, parfaites pour le codage par plage. Ainsi, les meilleurs taux de compression sont atteint pour m=40, environ égaux à 1.5

3.2 Compression avec pertes

On peut également choisir d'utiliser une compression avec pertes en ne gardant qu'une ligne et colonne sur deux. Le taux de compression est de 4.

```
1  for(int i = 0 ; i < nTaille3; i+=3){
2          ImgOut[i/4]=ImgIn[i];
3          ImgOut[i/4+1]=ImgIn[i+1];
4          ImgOut[i/4+2]=ImgIn[i+2];
5   }
```

FIGURE 3.3 – Image compressée avec pertes

4. Résultats

(a) Image de base

(b) Image segmentée, paramètres k=200 m=40

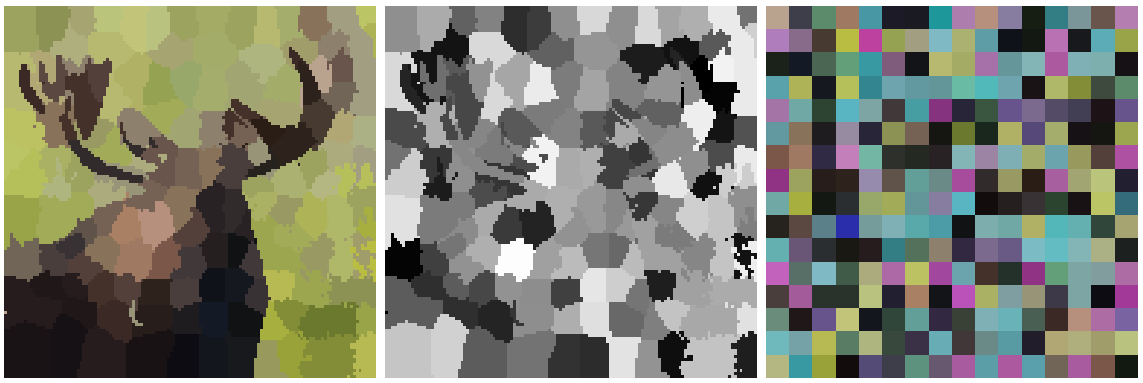


FIGURE 4.1 – Image segementée puis compressée par palette : Taux de compression 3, PSNR 20.08

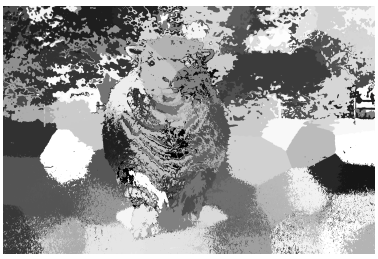
(a) Image de base

(b) Image segmentée, paramètres k=1000 m=10

FIGURE 4.2 – Image segementée puis compressée par palette : Taux de compression 3, PSNR 26.13

(a) Image de base

(b) Image segmentée, paramètres k=200 m=15



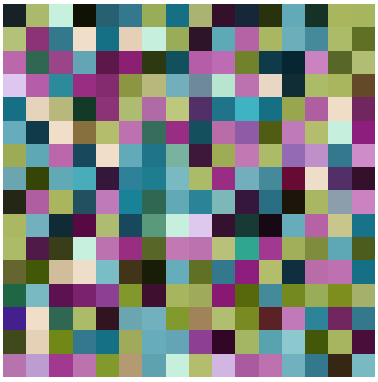


FIGURE 4.3 – Image segementée puis compressée par palette : Taux de compression 3, PSNR 30.1

5. Interface Utilisateur

On crée une interface en python à l'aide de *tkinter* permettant à l'utilisateur d'ouvrir une image pour pouvoir lui appliquer directement l'algorithme de SLIC. L'image va ensuite est compressée à l'aide d'un algorithme de codage par plage. L'interface va enfin afficher l'image des indexs résultat. Si les images sont trop grandes pour la résolution de l'écran, on réduit leur taille par un facteur 2.

```
filepath = askopenfilename(title="Ouvrir une image",
filetypes=[('png files','.png'),('all files','.*')])
file_nom = filepath.rsplit("/",1)[-1]
new_name = file_nom.rsplit(".",1)[0] + "SLIC.pgm"
os.system("slic " + file_nom + " " + new_name)

new_path = filepath.rsplit(".",1)[0] + "SLIC.pgm"
image2 = Image.open(new_path)
if image2.width>=1900 or image2.height>=1000 :
resize_image2 = image2.resize((int(image2.width*0.5),int(image2.height*0.5)))
else : resize_image2 = image2
test2 = ImageTk.PhotoImage(resize_image2)

label2 = Label(fenetre,image=test2)
label2.image = test2

label2.place(x=0, y=0)
if image2.width>=1900 or image2.height>=1000 :
   w = int(image2.width*0.5)
   h = int(image2.height*0.5)
else :
   w = image2.width
   h = image2.height
fenetre.geometry(f"{w}x{h}")
fenetre.mainloop()
```

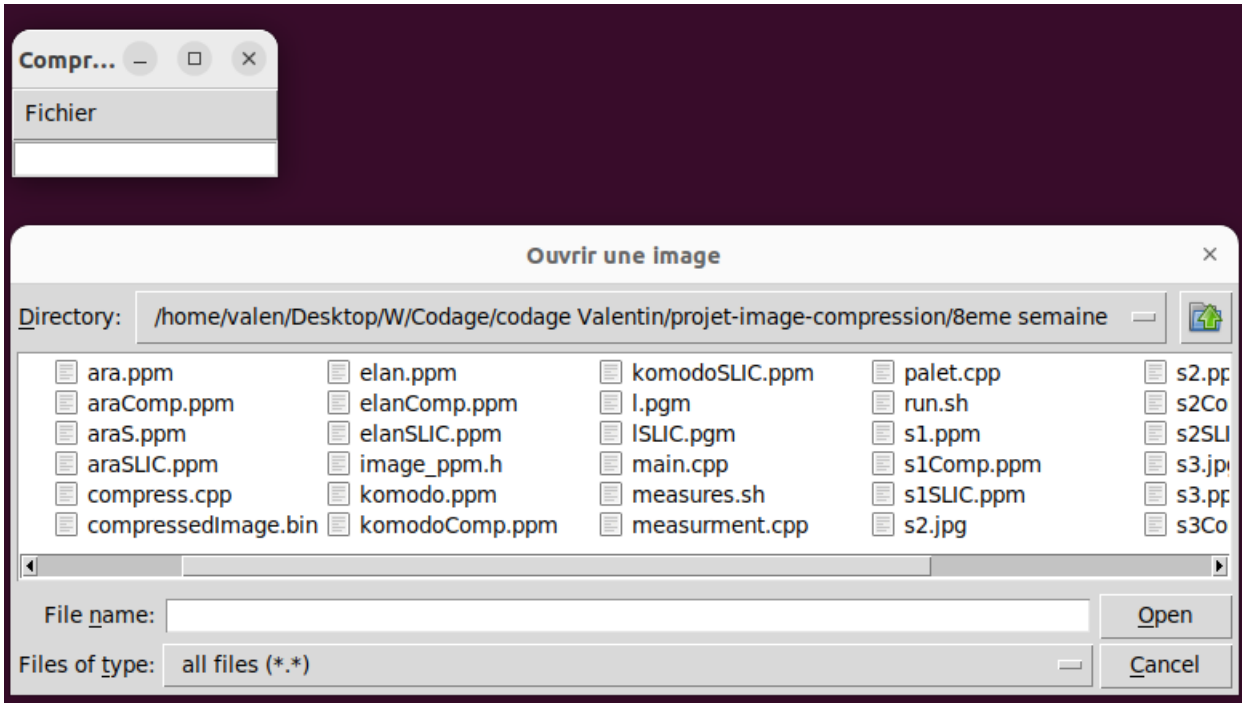


FIGURE 5.1 – Ouverture de l'interface

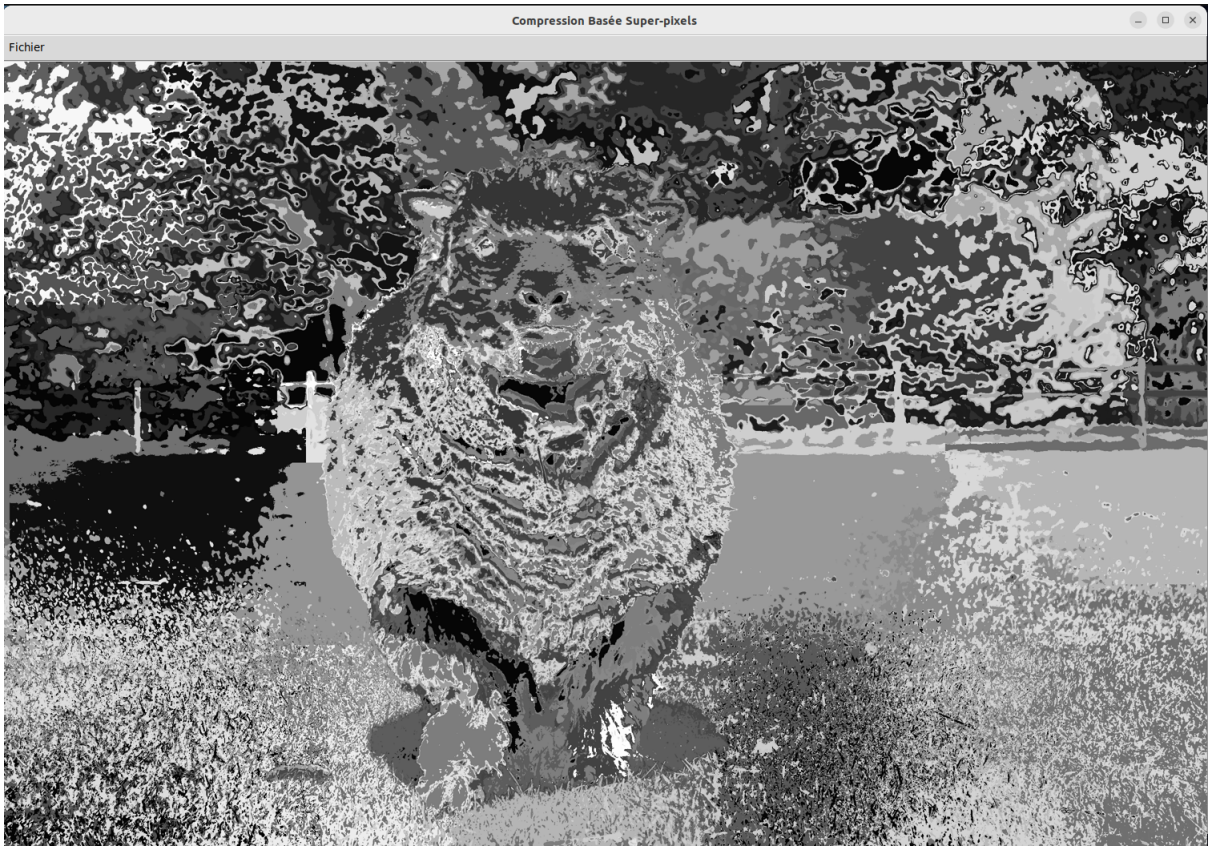


FIGURE 5.2 – Image résultat de l'appel de *SLIC* et de la compression

6. Conclusion

Ainsi, nous avons réussi à créer une application qui permet à l'utilisateur de segmenter et compresser ses images personnelles à l'aide d'une segmentation en super-pixels. Lors de ce projet, nous avons appris à utiliser une nouvelle forme de segmentation tout en respectant des délais assez courts car il nous fallait faire rapport de nos avancées chaque semaine.

Pour conclure, ce projet a été une grande expérience d'apprentissage pour nous. Nous avons acquis une compréhension approfondie des algorithmes de segmentation de superpixels, des techniques de compression d'image sans perte et de la façon de développer des interfaces utilisateur graphiques en utilisant tkinter en Python. De plus, nous avons compris l'importance de l'ajustement minutieux des paramètres et de son impact sur la qualité des résultats. Avec plus de temps, nous aurions aimé explorer des techniques de compression plus avancées, telles que la compression basée sur les ondelettes ou améliorer encore les performances de l'algorithme de segmentation de superpixels.

Annexe & Bibliographie

Lien vers notre page Gitlab : **Gitlab Projet Image-Compression**

Les articles de recherche utilisés dans le rapport sont les suivants :
- *Segmentations d'image basée super-pixels* s. d.
- *Measuring and Evaluating the Compactness of Superpixels* s. d.
- *SLIC Superpixels Compared to State-of-the-Art Superpixel Methods* s. d.
- *TurboPixels: Fast Superpixels Using Geometric Flows* s. d.

Voici le lien de la librairie externe de conversion d'espaces couleurs utilisée dans la partie Segmentation : *Colorspace* s. d.

Voici le lien de la documentation utilisée pour la création de l'interface graphique : *Python Tkinter* s. d.