Transfert de Couleurs

Zohra BENALI
Eliot GUEZ

19 Novembre 2023

Contents

Le but de ce projet est de transférer les données couleurs d'une image vers une autre.On représentera les images comme des tableaux à 3 dimensions [x,y [R;G;B]]. On considère ensuite la dernière composante de l'image et la représenterons comme un nuage de points dans le cube de coordonnées RGB (cf. Image 1) et on transfert le nuage de l'image a vers l'image b. Cette première vision a déjà sucité des questions comme la perte de l'information de la position du pixel (en effet le nuage ne considère que les données R, G et B de l'image) et l'indépendance des canaux. On traitera ces questions au fur et à mesure en suivant le TP proposé.

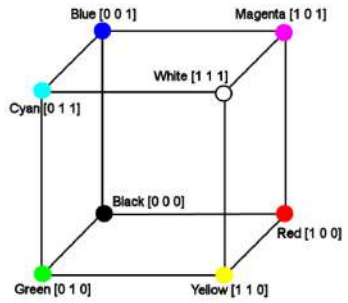


Figure 1: Image 1 : Cube RGB

Pour ce projet, on travaillera avec les images suivantes:

Image A: Le déjeuner des canotiers, Renoir

Image B : Jour de Dieu, Gauguin

1 Prescription et Egalisation d'histogramme

Une première idée pour effectuer un transfert de couleurs est de le réaliser en 1D sur les histogrammes de chaque canal de couleur en les supposant indépendants. Dans un premier temps, on effectuera une transformation affine de chaque histogramme puis on appliquera les algorithme de prescription d'histogramme vu en cours aux trois canaux de couleurs

1.1 Transfert affine

Pour cette méthode, on essaie d'appliquer à l'image A, la moyenne et la variance de B. En notant u et v les tableaux de nombres (np.array) respectivement associés à A et B et w une copie de u, on cherche alors m et p tels que :

$$w = m * u + p$$

$$E(w) = E(v)$$

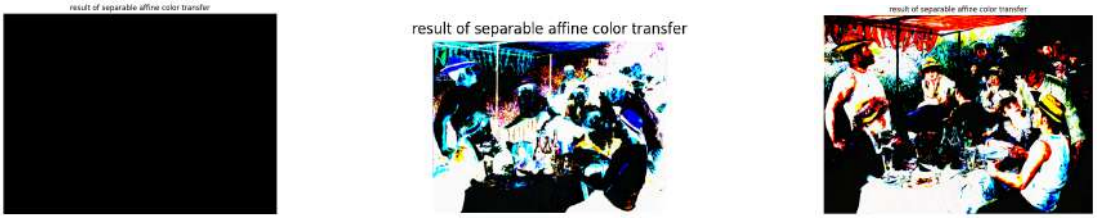
et

$$Var(w) = Var(v)$$

En résolvant ce système on trouve une expression de w qu'on implémente donc la fonction suivante :

```
def affine_transfer(u,v):
    w = np.zeros(u.shape)
    for i in range(0,3):
        w[:,:,i] = (u[:,:,i] -np.mean(u[:,:,i]))/
                   np.std(u[:,:,i])*np.std(v[:,:,i])
                   + np.mean(v[:,:,i])
    return w
```

Les premières implémentations donnent lieu a des images surprenantes comme celles-ci:



Premiers résultats

Il y avait en fait dans nos premières fonctions des inversions de coefficients. La fonction appliquée est alors décroissante et donne cet effet de négatif.

En corrigeant cette erreur et en repassant en base uint8 sur Python, nous avons finalement réussi à avoir des images concluantes:

Figure 2: Résultat 4: Premier transfert de couleur

On voit des sortes de saturations en turquoises sur l'image dues à un problème de clipping et de lecture dans la bonne base.

Finalement, on trouve cette image:

Figure 3: Résolution du phénomène de saturation

Les couleurs ont en effet été modifiées mais on ne retrouve pas la palette de couleur du tableau de Gauguin.

Pour s'en rendre mieux compte, on a aussi testé avec un tableau aux couleurs plus marquantes comme Nuit étoilée de Van Gogh:

(a) La nuit étoilée, Van Gogh

(b) Résultat du transfert affine

Figure 4: Test avec La Nuit étoilée

On a l'impression qu'on applique un filtre à l'image plus qu'on ne lui transfère des couleurs.

1.2 Prescription d'histogrammes

Une autre solution vu au TP 1 consiste à faire une spécification d'histogramme sur chaque canal de couleurs. Pour cela, on implémente les fonctions suivantes:

```
def todo_specification_1d(u,v):
    ind=np.unravel_index(np.argsort(u, axis=None), u.shape)
    unew=np.zeros(u.shape,u.dtype)
    unew[ind]=np.sort(v, axis=None)
    return unew
```

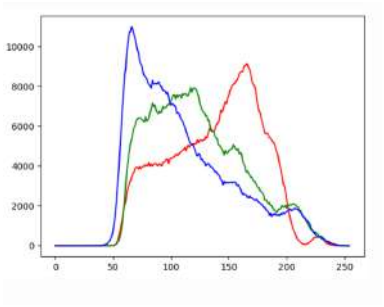
```
def todo_specification_separate_channels(u,v):
    w=np.zeros(u.shape,u.dtype)
    for k in range(3):
        w[:,:,k]=todo_specification_1d(u[:,:,k],v[:,:,k])
    return w
```

On obtient l'image suivante:

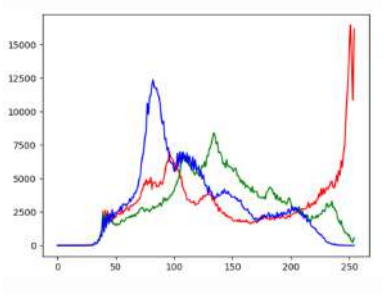
Figure 5: Color Separable transfer result

L'image est devenue plus claire et les couleurs y sont plus vives mais on ne retrouve pas du tout l'image espérée avec les couleurs de l'image B. C'est pourquoi on va chercher une autre méthode plus efficace pour obtenir l'effet souhaité.

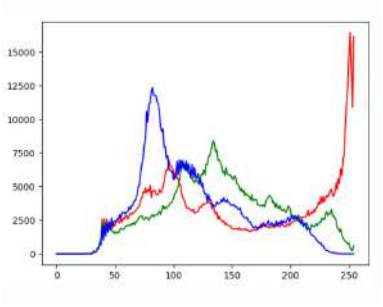
Ce qu'il s'est passé ici, c'est que les histogrammes par canaux ont bien été réalisés comme le montrent les graphes ci-dessous. Mais les canaux ne sont pas indépendants. En effet, il y a des points à l'intérieur de la projection de l'image dans l'espace associé au cube RGB ne sont pas pris en compte par l'égalisation d'histogramme (celle-ci n'opère en fait que sur les faces d'où les résultats observés).



Histogramme par canal : Le déjeuner des canotiers, Renoir



Histogramme par canal : Jour de Dieu, Gauguin



Histogramme par canal : Le déjeuner des canotiers après égalisation

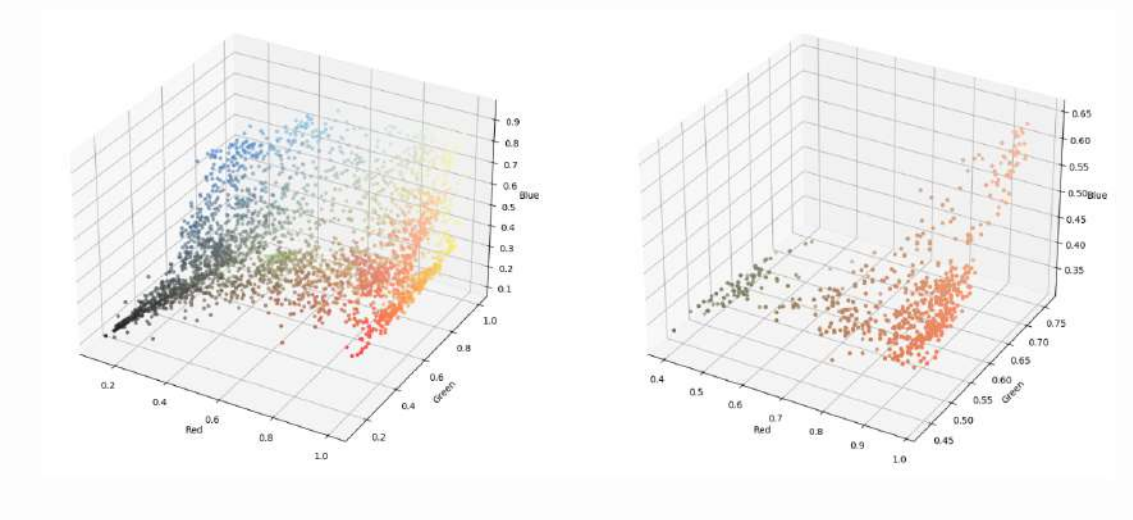


Figure 6: A gauche : nuage de points souhaité; A droite ; nuage obtenus par égalisation d'histogramme par canal

1.3 Conclusions

Ces premières méthodes de transfert de couleurs donnent des résultats satisfaisants dans le sens où les couleurs de l'image sont modifiées mais améliorable car on n'assiste pas à un transfert de couleurs mais plus à l'application d'un filtre par dessus l'image. De plus, on constate une perte de détails sur les images traitées (par exemple, des plis de la nappe disparaissent) ce qui pose la question de méthodes plus élaborées.

2 Problème du transport optimal

Dans la section précédente, on s'intéressait aux canaux de couleurs séparéments et on déplaçait les valeurs des couleurs sans prendre en compte leur répartition dans le cube RGB. En effet, si on se réfère à celui-ci, ce qu'on a fait dans la partie 1 était de faire du transfert de couleur sur la projection du nuage de couleurs sur les faces du cube ne prenant donc pas en compte l'intérieur de celui-ci.

Le problème ici est donc de réaliser un transport optimal en 3D puisqu'on veut déplacer le nuages de points en 3 dimensions RGB de l'image a à l'image b. Cependant, le calcul ayant une complexité très élevée, on va faire le transport à une dimension sur une projection non plus sur les faces mais sur la boule unité. On sait en fait résoudre ce problème en 1D.
On définit σ_X et σ_Y tels que

$$X_{\sigma_X(1)} \leq X_{\sigma_X(2)} \leq \cdots \leq X_{\sigma_X(n)},$$

$$Y_{\sigma_Y(1)} \leq Y_{\sigma_Y(2)} \leq \cdots \leq Y_{\sigma_Y(n)},$$

La permutation qui minimise le coût de transport de X à Y est $\sigma = \sigma_Y \circ \sigma_X^{-1}$ Cette fonction est réalisée avec le code suivant:

```
def transport1D(X,Y):
    sx = np.argsort(X)
    sy = np.argsort(Y)
    return((sx,sy))
```

Pour passer au cas 3D, on va générer des matrice orthonormales et on va effectuer une descente de gradient sur une projection du nuage de point dans un espace 1D : Si X et Y sont deux nuages de points, on introduit Z qui est au départ égale à X.

On calcule à chaque itération avec u_i (on projete sur une des couleurs) et ϵ un pas, une descente de gradient :

$$Z_{\sigma_Z} = Z_{\sigma_Z} + \varepsilon((Y,u_i)_{\sigma_Y} - (Z,u_i)_{\sigma_Z})u_i,$$

On observe donc le transport des points de X vers Y avec le programme suivant;

```
def todo_transport3D(X,Y,N,e):
    Z=np.copy(X)
    for _ in range(N):
        u=np.random.randn(3,3)
        q=np.linalg.qr(u)[0]  #matrice orthonormale
        for i in range(3):
            projectedY=np.dot(Y,q[:,i])
            projectedZ= np.dot(Z,q[:,i])
            [sy, sz]= transport1D(projectedY,projectedZ)
            diff=(projectedY[sy] - projectedZ[sz])
            Z[sz,:]+= e * diff[:,None] * q[:, i][None,:]
    return ((Z,sy,sz))
```

On observe ainsi le résultat sur deux nuages de points X et Y généré aléatoirement.

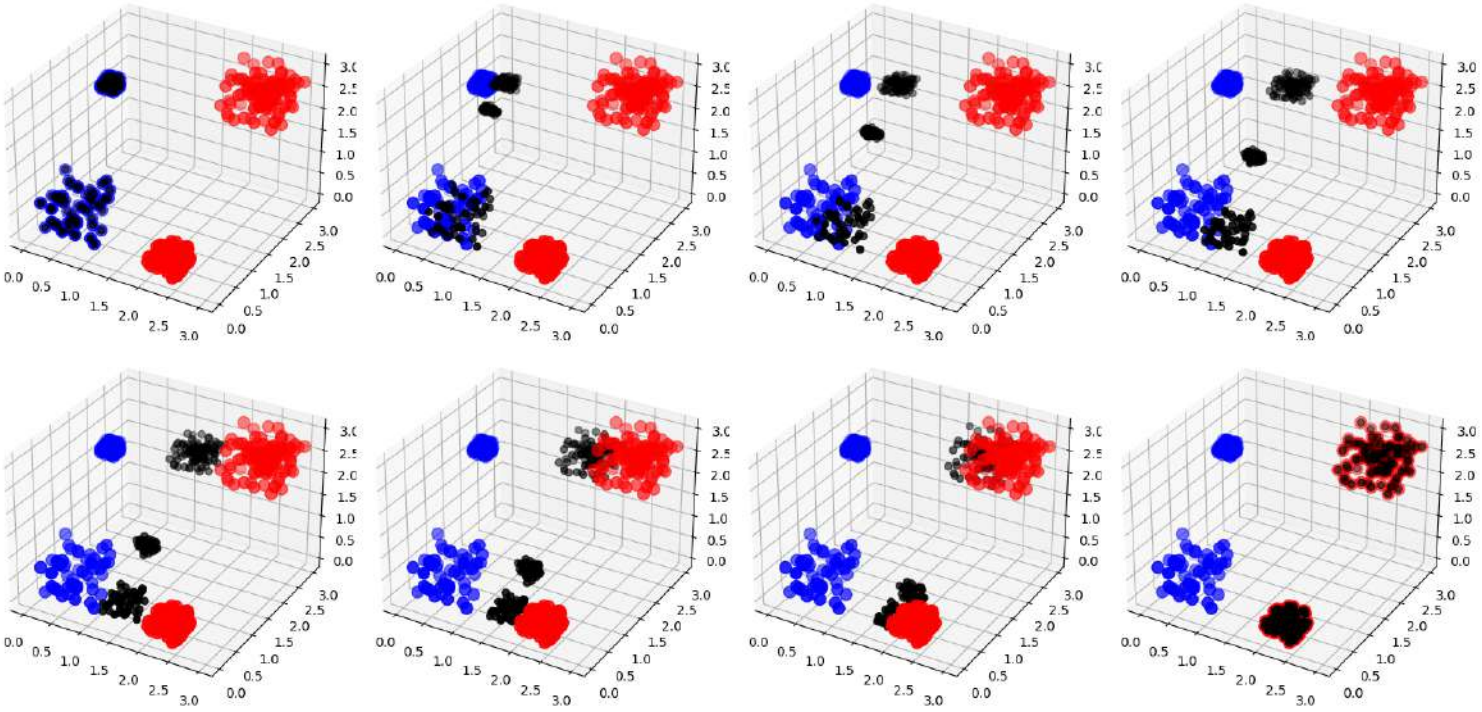


Figure 7: Transport du nuage bleu au nuage rouge

Maintenant, on va chercher à appliquer ceci aux deux images A et B du départ. Etant donné que le temps de calcul reste très élevé, on va sous-échantillonner les images de départ et observer comment l'image est transformée par notre programme précédent.

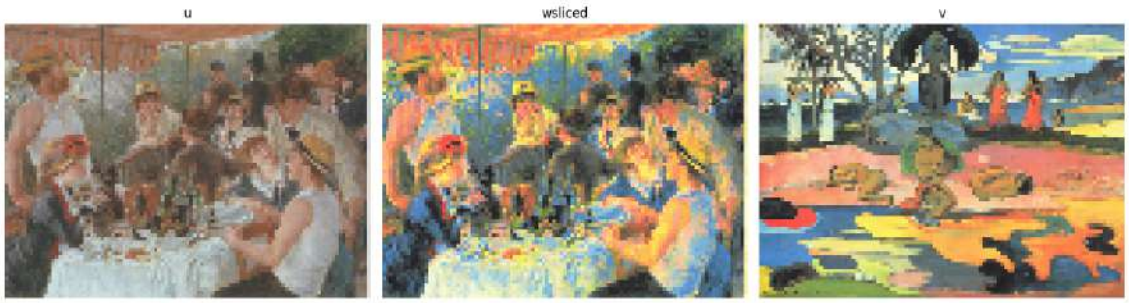


Figure 8: wsliced subsample by a factor a=10

Figure 9: wsliced subsample by a factor a=5

Figure 10: wsliced subsample by a factor a=3

On voit ici que, malgré l'échantillonnage, l'effet attendu est celui souhaité. On commence à retrouver la palette de couleur de l'image B sur l'image A. Cependant, on observe aussi des artefacts et du bruit sur l'image. C'est pourquoi nous allons filtrer l'image afin de supprimer ces effets sur notre image.

3 Postprocessing regularization

On remarque sur les différents exemples que les problèmes qu'on peut retrouver sont dûes à des irrégularités spatiales de la carte des transports définies par $M(u) = T(u) - u$ avec T(u) le transport de l'image de départ u sur l'image voulu v. Ansi l'image reconstruite sera de la forme $T(g(u)) = Y_u(g(u) - u) + u$ ou encore $T(g(u)) = Y_u(g(u) + u - Y_u(u)$ avec $Y_u(g(u))$ l'image filtré de T(u) et $u - Y_u(u)$ les détails de l'image u.

3.1 Simple Averaging Filter

On propose donc en premier de prendre pour Y_u un simple filtre moyenne : en utilisant un calcul intégrale que deux boucles for.

```
def average_filter(ima,r):
    (nrow, ncol) = ima.shape
    big_u = np.zeros((nrow+2*r+1,ncol+2*r+1))
    big_u[r+1:nrow+r+1,r+1:ncol+r+1] = ima
    big_u = np.cumsum(np.cumsum(big_u,0),1)  # integral image

    out = big_u[2*r+1:nrow+2*r+1,2*r+1:ncol+2*r+1] + big_u[0:nrow,0:ncol]

        - big_u[0:nrow,2*r+1:ncol+2*r+1] - big_u[2*r+1:nrow+2*r+1,0:ncol]
    out = out/(2*r+1)**2
    return out
```

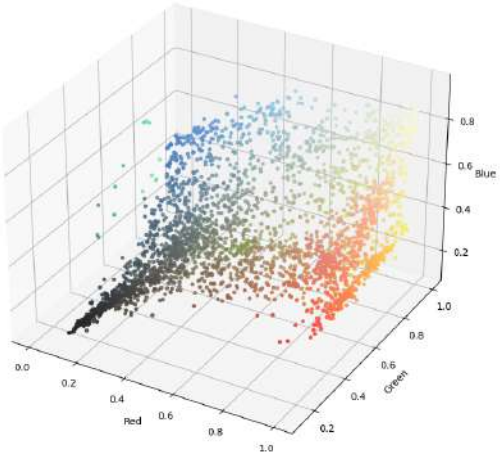
Figure 11: image filtré vs non filtré pour un échantillonnage par 3

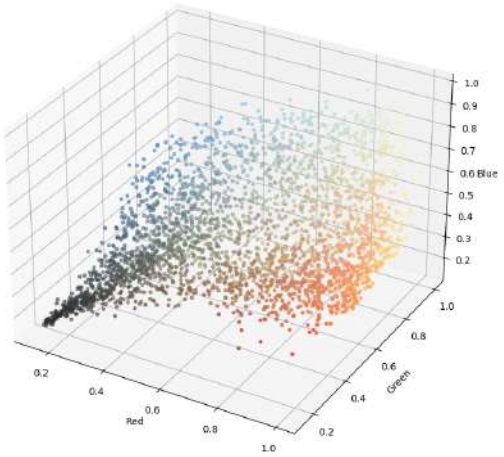
On voit ici qu'on a perdu les couleurs qu'on voulait transférer dans l'image. Cela est du au fait que le rayon choisi dans le filtre moyen est trop élevé (5). On choisit donc un rayon égal à 2 :

Image A: image filtré avec r=2

Image B : Le déjeuner des canotiers, Renoir

On peut également regarder le nuage de points associé à l'image filtré :





On observe effectivement que le nuage de point a été modifié.

Le résultat est déjà assez satisfaisant mais on retrouve tout de même du flous dû au fait qu'on a pas essayé de suivre les irrégularités de l'image de base. Une idée beaucoup plus intéressante et qui va être à la base des deux prochaines méthodes est de faire en sorte que M(u) suivent la régularité de u.

3.2 Guided filter

On va donc choisir u comme image guide. On veut que le gradient de M(u) approxime le plus possible le gradient de u. Pour tout les pixels k de u. On va prendre une fenetre autour de ce point ω_k, on cherche les coefficients a_k et b_k qui minimise la distance euclidienne entre u et $q = a_k guide + b_k$. On obtient par régression linéaire :

$$a_k = \frac{\frac{1}{|\omega_k|}\sum_{i\in\omega_k} guide(i)u(i) - \text{mean}(u)_k \text{mean}(guide)_k}{\sigma(guide)_k^2}$$

$$b_k = \text{mean}(u)_k - a_k \text{mean}(guide)_k$$

avec :

$$\sigma(guide)_k^2 = \frac{1}{|\omega_k|}\left(\sum_{i\in\omega_k} guide(i)^2 - \text{mean}(guide)_k^2\right).$$

et mean la valeur moyenne de l'image u sur la fenêtre carré ω_k

Voici le code de cette fonction:

```
def todo_guided_filter(u,guide,r,eps):
    C= average_filter(np.ones(u.shape),r)
    mean_u       = average_filter(u, r)/C
    mean_guide   = average_filter(guide, r)/C
    corr_guide   = average_filter(guide*guide, r)/C
    corr_uguide  = average_filter(u*guide, r)/C
    var_guide    = corr_guide - mean_guide **2
    cov_uguide   = corr_uguide - mean_u * mean_guide

    alph = cov_uguide / (var_guide + eps)
    beta = mean_u - alph * mean_guide

    mean_alph = average_filter(alph, r)/C
    mean_beta = average_filter(beta, r)/C

    q = mean_alph * guide + mean_beta
    return q

diff = wsliced-usubsample
out = np.zeros_like(usubsample)
for i in range(3):
    out[:,:,i] = todo_guided_filter(diff[:,:,i], usubsample[:,:,i], 3,1e-4 )
```

On peut remarquer qu'on applique le filtre sur chaque channel indépendamment. Etant donné que l'average filter crée un déséquilibre sur les bords puisque on divise par la taille de la fenêtre alors qu'on prend moins de points. On divise par C pour compenser qui est un filtre moyen sur une matrice de 1.

Figure 12: image filtré avec guided filter vs non filtré

De cette manière, on a du mal à voir en quoi cette méthode de régularisation a de meilleurs résultats que la première: on va donc afficher la différence entre ces deux images et multiplier celle-ci par un facteur multiplicatif pour la faire ressortir: on obtient l'image suivante en multipliant par 100 :

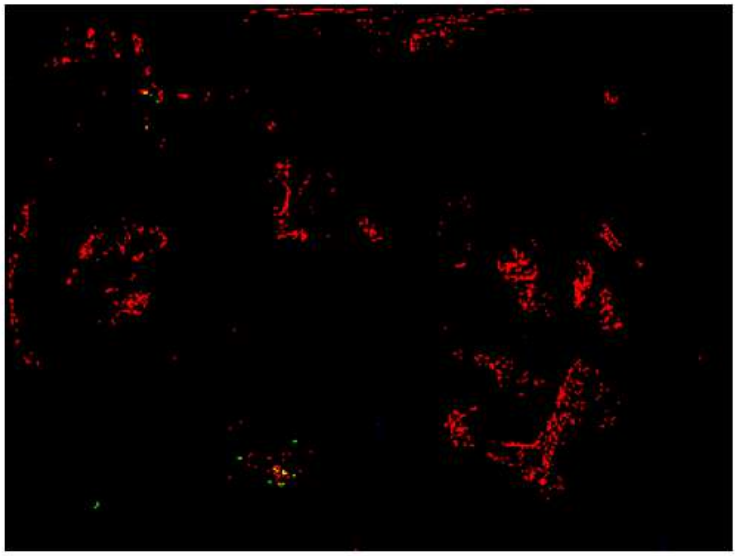


Figure 13: différence entre image filtré avec le guided filter

On voit que les différences principales se font au niveau du bras de la personne au premier plan.

3.3 Transportation Map Regularization

Le principe est assez similaire au filtre précédent : on a l'image reconstruite : $TMR_u(T(u)) = Y_u(T(u)) + u - Y_u(u)$
avec $Y_u(v)(x_k) = \frac{1}{C(x_k)} \int_{y \in N(x_k)} v(y) \cdot w_u(x_k, y) dy$
Ici, on a les poids $w_u(x_k, y) = e^{\frac{-||u(x_k)-u(y)||^2}{\sigma^2}}$ et la constante de normalisation $C(x_k) = \int_{y \in N(x_k)} w_u(x_k, y) dy$
σ un paramètre et N(x) un voisinage du point x

On peut simplifier cette expression pour avoir : $\frac{1}{C(x)} \int_{y \in N(x)} (T(u)(y) - u(y)) \cdot e^{\frac{-||u(x)-u(y)||^2}{\sigma^2}} dy - u$

On prendra ici un $\sigma = 10$ un rayon de 4 puisqu'il est recommandé de prendre comme un voisinage un disque de diamètre 10. Pour que ce filtre soit optimale, il faudrait itérer plusieurs fois cette opération. On va donc regarder ce qu'on obtient avec la première itération. Le code suivant permet de réaliser une tel transformation :

```
def transportation_Map_Regularization(u,T,r,sigma):
    a,b,_=u.shape

    wu=np.ones((a,b,3))

    for x in range (a):
        for y in range(b):
            int,C=0,0
            for i in range(max(0,x-r),min(x+r,a)):
                for j in range(max(0,y-r),min(y+r,b)):

                    int+= (T[i,j]-u[i,j])*np.exp(- np.linalg.norm((u[x,y]-u[i,j])**2)/s
                    C+=np.exp(-np.linalg.norm(u[i,j]-u[x,y])/sigma**2)
            wu[i,j] = int/C
    return wu
```

Ainsi pour chaque élément x_k de l'image : on choisit les points qui sont inclus dans un voisinage de celui-ci: la valeur de l'intégrale augmente de $(T(u)(y) - y) * w_u(x_k, y)$ avec i,j les coordonnées de y dans l'image. On a également la constante de normalisation qui est calculé. Il ne manque plus qu'a ajouté l'image initiale pour obtenir le résultat souhaité.

Avec une itération on obtient curieusement :

Figure 14: TMR 1 itération

L'image est assez différente de l'image de départn, il y a un fond bleu qui apparaît sur l'image. Avec 5 itérations on obtient :

Figure 15: TMR 5 itération

On observe que plus le nombre d'itération augmente, moins le résultat est bon. Une image blanche au centre est assez perturbant et laisse à penser qu'il y a un réelle problème dans l'algorithme.

4 Gaussian Mixture Model

4.1 Considérations théoriques : distance de Wasserstein dans les GMM

Dans cette partie, nous nous sommes intéressés au Transport Optimal pour un modèle Gaussien et comment cela pouvait s'appliquer au transfert de couleur.

Cette idée vient du fait que l'on sait exprimer la distance de Wasserstein entre deux gaussiennes multivariées. Celle-ci s'écrit :

$$W_2^2(G_0, G_1) = \|m_0 - m_1\|^2 + tr(\Sigma_0 + \Sigma_1 2(\Sigma_0^{\frac{1}{2}}\Sigma_1\Sigma_0^{\frac{1}{2}})^{\frac{1}{2}})$$

On a alors que la fonction de transport est :

$$\forall x \in R^d, T(x) = m_1 + \Sigma_0^{-\frac{1}{2}}(\Sigma_0^{\frac{1}{2}}\Sigma_1\Sigma_0^{\frac{1}{2}})^{\frac{1}{2}}\Sigma_0^{-\frac{1}{2}}(x - m_0)$$

Le tout est alors de savoir comment assigner les gaussiennes entre elles. Pour cela, on va générér toutes les maps possibles (matrice T) puis faire la moyenne de ces maps pondérées par le produit de leurs poids (probabilité qu'un pixel appartienne à une composante gaussienne) et de cette dite gaussienne. Il s'agit du Tmean dans nos fonctions défini par :

$$T_{mean}(x) = \frac{\sum_{k,l} w_{k,l} G_{m_k,\sigma_k} T_{kl}}{\sum_{k,l} w_{k,l} G_{m_k,\sigma_k}}$$

On cherche alors à approximer nos images par des GMMs pour ensuite y appliquer notre fonction de transport.

4.2 Notre approche et implémentation de l'algorithme

Pour générer ces GMMs, dont nous choisirons le nombres de composantes Gaussiennes, nous allons utiliser l'algorithme EM (Expectation, Maximisation). Cet algorithme va alors générer à partir des données les moyennes, covariances et poids de chaque Gaussiennes ainsi générées. Au travers de cet algorithme, on cherche à minimiser la distance entre notre distribution de couleurs et la distribution gaussienne. Pour mesurer cela, on se propose de minimiser la log-vraissemblance de notre nuage de points (c'est la valeur de log-likelihood dans le code ci-dessous).

Nous avons essayé d'implémenter nous-mêmes les fonctions de gmm et de transport avant d'utiliser les librairies python. Voici le code associé à la fonction gmm :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal


def gmm(data, nbr_components, max_iters=10000, seuil=0.5):

    nb_pixels, nb_color = data.shape
    prev_log_likelihood = 0
```

```
    # On initialise les moyennes , covariances et poids

    means = data[np.random.choice(nb_pixels, nbr_components,
    replace=False)]
    covariances = np.array([np.eye(nb_color)
    for _ in range(nbr_components)])
    weights = np.ones(nbr_components) / nbr_components

    #On utilise l'algorithme EM

    for _ in range(max_iters):
# E-step: Calcule proba qu'un point appartienne a une compo gaussienne

        pdfs = np.array([weights[i] * multivariate_normal.pdf(data,
        mean=means[i],cov=covariances[i])
        for i in range(nbr_components)])
        responsibilities = pdfs / pdfs.sum(0)

     #respo[i][j] est la proba que le pixel i soit dans la gaussienne j


        # M-step: On fit les variances et moyennes au set de points

        Nk = responsibilities.sum(1) #nb de points dans la composante i

        means = np.dot(responsibilities, data) / Nk[:, None]

        covariances = np.array([np.dot((responsibilities[i, :, None]
        * (data - means[i])).T , (data - means[i])) / Nk[i]
        for i in range(nbr_components)])

        weights = Nk / nb_pixels

        label = np.argmax(responsibilities, axis=0)

        log_likelihood = np.sum(-np.log(np.sum(pdfs, axis=0)))

        delta_log_likelihood = log_likelihood - prev_log_likelihood

        if np.abs(delta_log_likelihood) < seuil:
            return means, covariances, weights, label,
            responsibilities.T

        prev_log_likelihood = log_likelihood

    return means, covariances, weights, label, responsibilities.T
```

4.3 Résultats et Analyse

En essayant d'appliquer ce code directement aux images, nous nous sommes retrouvés face à des résultats surprenants. Les images étaient ternes et on ne retrouvait pas les couleurs du tableau source.

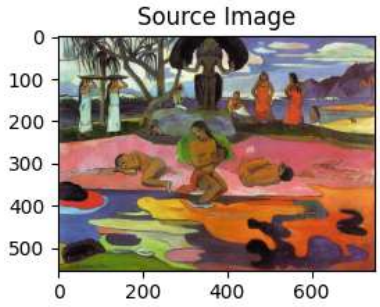


Image A: Jour de Dieu, Gauguin

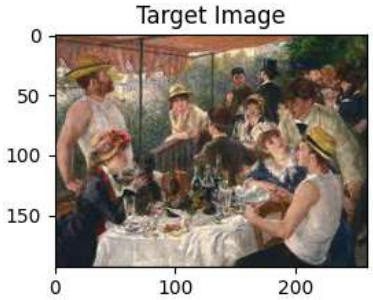


Image B : Le déjeuner des canotiers, Renoir

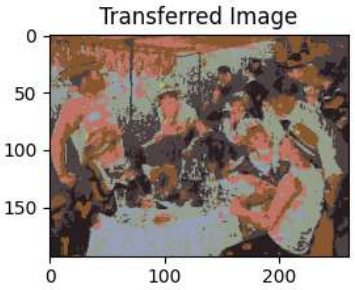


Image C : Résultats du modèle GMM

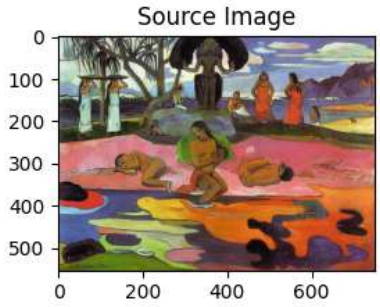


Image A: Jour de Dieu, Gauguin

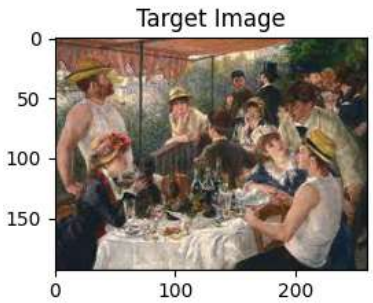


Image B : Le déjeuner des canotiers, Renoir

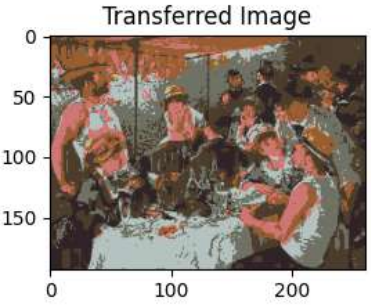


Image C : Résultats du modèle GMM

Face à ses résultats, nous avons afficher les nuages de points générés par le code pour mieux comprendre la répartition :

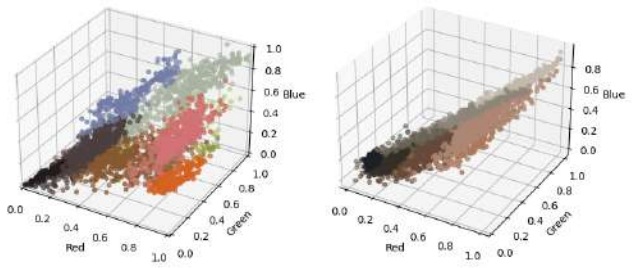


Figure 16: Nuages de points des 2 images après GMM et avant transfert

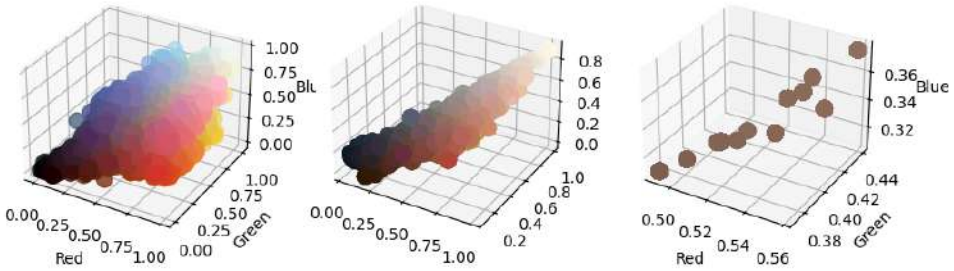
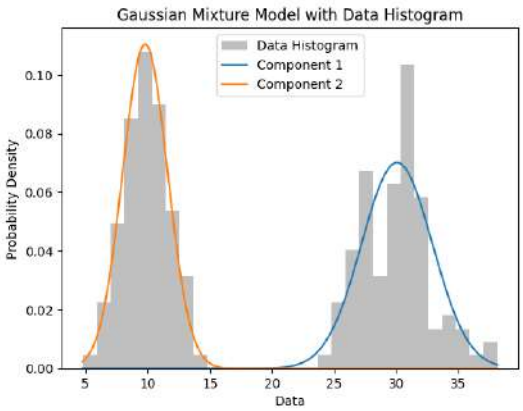
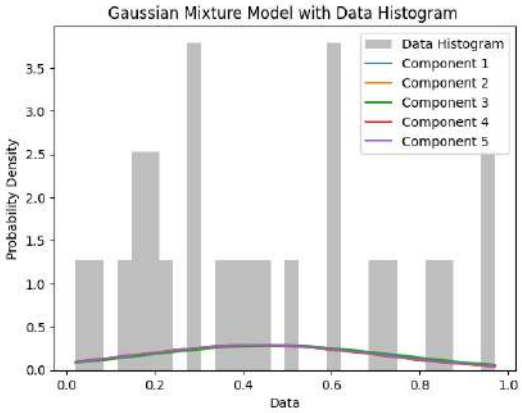


Figure 17: Nuages de points pour 11 composantes gaussiennes

On remarque que bien que les moyennes obtenues ne donnent des couleurs un peu ternes, on a une image résultante avec 1 point brunâtre par composantes ce qui n'est pas du tout ce à quoi nous nous attendions.
On applique ce code alors à des données 1D pour tester d'abord son efficacité

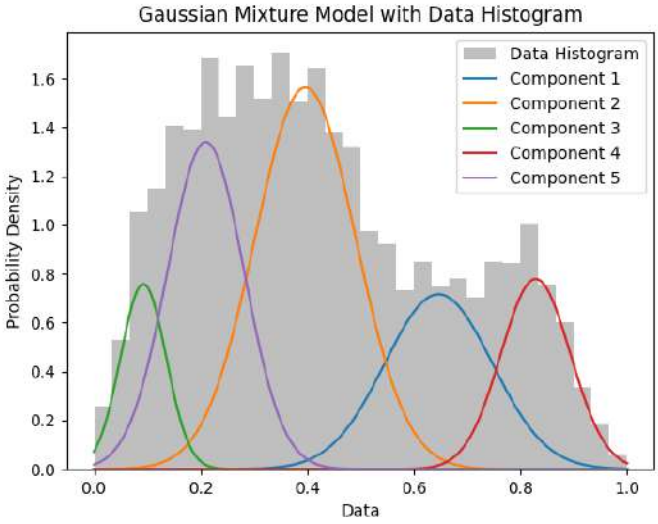


Données aléatoires répartition gaussienne



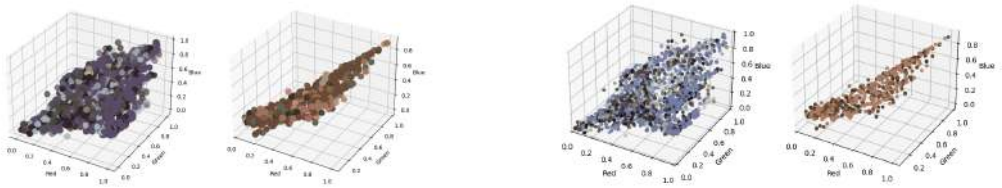
Données aléatoires, répartition aléatoire

Le modèle semblait alors avoir du mal à estimer les mixture si le modèle était très loin d'un modèle gaussien. Le problème venait en fait de la façon dont nous actualisions la matrice r des proba d'appartenir à une gaussienne dans le step M de l'algorithme.
Une fois ceci résolu, nous avons testé l'algorithme 1D sur les images :



Données du tableau de gauguin

Le modèle semblait alors bien fonctionner sur des tableaux 1D. Mais nous obtenions alors des nuages comme ceux-ci après transport :



Nuages tests GMM

En cherchant nos erreurs, il y avait une coquille avec la manière dont on faisait notre transport. En effet, nous n'avions pas implémenter correctement le Transport et celui-ci ressortait en fait une moyenne pour chaque pixel d'un voisinnage de celui-ci d'où l'aspect très brunâtre de nos images.

Nous avons alors corriger cette erreur et nous avions avec le code suivant les images :

4.4 Utilisation de scikit-learn

Enfin, en recherchant dans les travaux de Julie Delon et les librairies Python, on peut assez bien implémenter ce modèle à l'aide du module Gaussian Mixture de scikit- learn.

5 Conclusions

Finalement, nous avons pu au cours de ce projet étudier différentes méthodes de transfert de couleurs plus ou moins satisfaisantes.

L'approche que nous avons retenu est cetlle d'un transfert optimal entre les nuages de points(section 1).

Le premier transport étudié fut de se ramener au problème du transport optimal de Monge 1D et de l'appliquer à chaque canaux ce qui nous a donné des résultats très satisfaisants malgré des artefacts de transports que nous avons pu corriger à l'aide de filtres (section 2).
On observe que nos résultats concernant le traitement à postérioiri du transfert de couleur sont assez loin de ceux donné dans le papier de Julien Rabbin, Julie Delon, et Yann Gousseau. En effet, l'algorithme qui a un coût minimal et de résultats qui n'ont rien a envié aux autres méthodes est l'average filter. Cependant, peut-être que cela est du à un problème d'algorithme. Ainsi, cette piste reste à creuser.

L'approche par GMM (section 3) est assez intéressante dans le sens où l'on simplifie le problème en utilisant les connaissances sur les Gaussiennes et une expression du problème du transport plus simple. La partie difficile de cet algorithme fut plus de générer les gaussiennes que de réaliser le transport. Cette application fut donc facilitée par l'utilisation de la librairie sklearn. Il subisistait néanmoins des artefacts dû aux transformations que l'on peut corriger par l'application de filtres (section 2.). Nous avons notamment vu dans le papier de Julie Delon et Agnès Desolneux à ce sujet que les GMMs ont d'autres propriétés notamment dans la synthèse de textures.

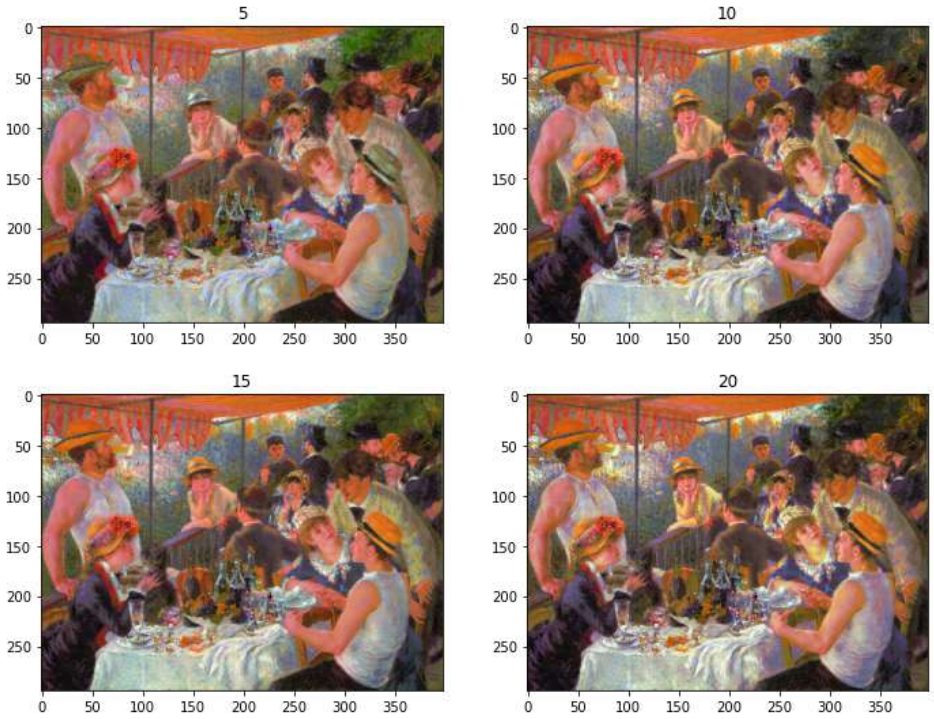


Table 1: Résultats GMM pour K = 5,10,15 et 20

Dans le cadre du transfert de couleurs, nous avons vu que différentes approches étaient possibles (nous n'en avons exploré que 2 dans ce projet). Nous avions au début de ce projet l'ambition d'appliquer cela à des séquences de films. Le fait est que la difficulté d'implémentation sur des images nous a assez occupés pour cette période. On pourrait imaginer dans la suite de réaliser cette première idée et pourquoi pas de s'intéresser à la colorisation de films qui est un tout autre procédé.

6 Annexes et documentations

1. Julie Delon, Agnès Desolneux. A Wasserstein-type distance in the space of Gaussian Mixture Models. SIAM Journal on Imaging Sciences, 2020, 13 (2), pp.936-970. ff10.1137/19M1301047ff. ffhal-02178204v4f

2. Rabin Julien, Gabriel Peyré, Julie Delon, Bernot Marc. Wasserstein Barycenter and its Application to Texture Mixing. SSVM'11, 2011, Israel. pp.435-446. ffhal-00476064

3. Julien Rabin, Julie Delon, Yann Gousseau - REGULARIZATION OF TRANSPORTATION MAPS FOR COLOR AND CONTRAST TRANSFER