BE : Variations sur l'astuce des noyaux

Par groupe de 4 étudiants maximum, vous devez rédiger un rapport détaillé dans lequel vous répondrez aux questions du sujet et plus globalement au problème demandé. La qualité de rédaction ainsi que tous les compléments que vous apporterez (réflexions, analyses des réponses, etc...) seront fortement pris en compte. Toutes les fonctions programmées le seront en Python et seront abondamment commentées. Le rendu du projet se fera sous la forme d'un fichier .zip contenant le rapport ainsi que l'ensemble des programmes.

Exercice 1 : Reconstruction de surface avec les moindres carrées à noyaux

Dans cet exercice nous allons utiliser l'astuce des noyaux afin de reconstruire une surface (2D) à partir de la donnée de points de $\mathbb{R}^3$.

Supposons donné un ensemble (sous la forme d'une matrice) de points de m points de $\mathbb{R}^3$:

$$P := \begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \vdots & \vdots & \vdots \\ x_m & y_m & z_m \end{pmatrix}$$

1. Commençons par écrire la forme « linéaire » de la recherche d'une surface passant « au plus près de ces points ». On cherche donc un plan (le monde linéaire est ainsi fait...) :

$$\Pi : \quad \alpha x + \beta y + \gamma z + \delta = 0$$

où les (x, y, z) parcours $\mathbb{R}^3$. Ainsi déterminer un tel plan revient à déterminer les coefficients : α, β, γ et δ.

(a) On supposera dans tout cet exercice que $\gamma \neq 0$ [1]. Quels types de plan cette hypothèse supprime ? En déduire dans ce cas que l'équation du plan s'écrit sous la forme :

$$\Pi : \quad ax + by + d = z$$

avec a, b et c à déterminer. Ainsi le plan solution s'écrit sous la forme d'une fonction :

$$f(x, y) = ax + by + d$$

C'est cette fonction que nous allons essayer de déterminer.

(b) Montrer alors que le problème de la recherche des coefficients a, b, d s'écrit sous la forme d'un problème de moindre carrée de la forme :

$$X^* := \arg\min_{X \in \mathbb{R}^3} \|AX - Z\|_2^2$$

avec : $Z := (z_1, z_2, \cdots, z_m)^T \in \mathbb{R}^m$, les inconnues $X := (a, b, d)^T \in \mathbb{R}^3$ et A une matrice que vous déterminerez.

(c) Rédiger un programme Python qui résout au sens des moindres le problème de la recherche d'un plan passant au plus près des points P.

i. Vous créerez une fonction : **MCsurface(P)** qui prend en entrée un ensemble de points sous la forme d'une matrice P et qui renvoie en sortie la solution au sens des moindres carrées :

$$X^* := (a^*, b^*, d^*)^T$$

ainsi que l'erreur commise : $\|AX^* - Z\|$.

ii. Une fonction : **Visualisation(P,X,nb)** où P est la matrice des points, X est la solution au sens des moindres carrées et nb le nombre de points de discrétisation sur l'axe des x et des y (on supposera que l'axe des x et des y sont pareillement discrétisés) pour la visualisation du tracé. Cette fonction renverra en sortie le tracé 3D du plan (on pourra utiliser par exemple la fonction plot_surface) et pour la discrétisation des axes x et y on pourra utiliser les commandes :

1. En réalité il faudrait faire trois fois les calculs suivants adaptés à chaque cas : $\alpha \neq 0$, $\beta \neq 0$ et $\gamma \neq 0$. Puis prendre comme solution celle de norme minimale dans les moindres carrés.

```
discretex=np.linspace(np.min(P[:,0])-0.5,np.max(P[:,0])+0.5,nb)
discretey=np.linspace(np.min(P[:,1])-0.5,np.max(P[:,1])+0.5,nb)

discretex,discretey=np.meshgrid(discretex,discretey) #pour le tracé 3d
```

Vous testerez vos fonctions sur l'ensemble de points :

$$P := \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1.5 \\ 0 & 1 & 0.5 \\ 1 & 1 & 1 \\ 0.5 & 0.5 & 0 \end{pmatrix}$$

On tracera également sur le même graphique les points de P en utilisant la fonction *scatter* de Matplotlib. **Attention également penser à transposer votre matrice S que vous produirez à partir des discrétisations de x et de y et de X^*.**

2. Nous allons maintenant généraliser le procédé précédent et s'autoriser à avoir des surfaces non-linéaires grâce à l'astuce des noyaux.
   (a) Utilisons l'astuce des noyaux pour transformer notre problème :

$$X^* := \arg\min_{X\in\mathbb{R}^3} \|AX - Z\|_2^2$$

Supposons donné $k : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}$ un noyau. Justifier qu'il existe une fonction de redescription ϕ et un espace de Hilbert tel que le problème précédent se réécrive sous la forme :

$$c^* = \arg\min_R \sum_{i=1}^{m} \left(\langle R, \phi(w_i)\rangle - z_i\right)^2$$

avec : $R := \sum_{i=1}^{m} c_i k_{w_i}$ où les c_i sont des réels et $w_i := (x_i, y_i, 1)^T$, $c^* := (c_1^*, c_2^*, \cdots, c_m^*)^T$.

   (b) Montrer alors que le problème précédent s'écrit matriciellement sous la forme :

$$\|Kc - Z\|_2^2$$

avec K la matrice de Gram associée au w_i et $c = (c_1, c_2, \cdots, c_m)^T$. En déduire que si la matrice K est définie positive la solution existe (au sens classique) et est alors unique.

   (c) Justifier que la fonction solution s'écrit :

$$f_K(x,y) = \left\langle R^*, \phi\left(\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\right)\right\rangle = \sum_{i=1}^{m} c_i^* k\left(w_i, \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\right)$$

avec $R^* := \sum_{i=1}^{m} c_i^* k_{w_i}$ la solution des moindres carrés à noyau.

3. Rédiger un programme Python qui résout au sens des moindres avec noyau le problème de la recherche d'une plan surface au plus près des points P.
   (a) Vous créerez une fonction : **MCKsurface(P,kern)** qui prend en entrée un ensemble de points sous la forme d'une matrice P et une fonction noyau *kern* et qui renvoie en sortie la solution au sens des moindres carrées avec noyaux :

$$c^* := (c_1^*, c_2^*, \cdots, c_m^*)^T$$

ainsi que l'erreur commise : $\|Kc^* - Z\|$.

   (b) Une fonction : **VisualisationK(P,c*,nb, kern)** où P est la matrice des points, c^* est la solution au sens des moindres carrées et *nb* le nombre de points de discrétisation sur l'axe des x et des y (on supposera que l'axe des x et des y sont pareillement discrétisés) pour la visualisation du tracé. Cette fonction renverra en sortie le tracé 3D de la surface solution (on pourra utiliser par exemple la fonction plot_surface). Vous testerez vos

fonctions sur l'ensemble de points :

$$P := \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1.5 \\ 0 & 1 & 0.5 \\ 1 & 1 & 1 \\ 0.5 & 0.5 & 0 \end{pmatrix}$$

en utilisant les différents noyaux vu en TP (polynomiaux, exponentiels, gaussiens, etc...) On tracera également sur le même graphique les points de P en utilisant la fonction *scatter* de Matplotlib. **Attention également penser à transposer votre matrice S que vous produirez à partir des discrétisations de x et de y et de c^*.**

(c) **Question défi :** Chercher un noyau qui minimise le plus possible l'erreur commise pour l'ensemble de points contenu dans le fichier : « P_test.npy ».

4. Utiliser d'autres ensembles de points (relevés d'altitudes, autres...) P et tester les fonctions précédentes. Vous commenterez l'influence des différents noyaux utilisés.

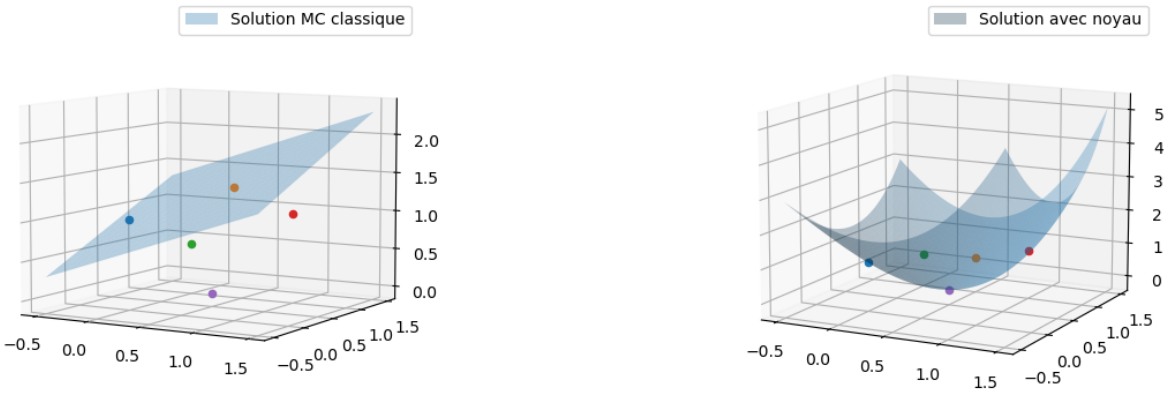

FIGURE 1 – Comparaison entre les deux méthodes.

Exercice 2 : Partitionnement d'image avec ACP et k-moyennes

Dans cet exercice nous allons créer un algorithme de partitionnement d'image (clustering) qui consistera dans ses grandes lignes à :

1. Choisir des points au hasard sur une image : position i, j en lignes et colonnes sur l'image.
2. Pour chaque pixel (i, j) on associera un vecteur (ligne) contenant la position i, j ainsi que l'intensité des trois couches de couleurs mais également d'autres paramètres que vous définirez par la suite.
3. L'étape précédente permet de créer une matrice où chaque ligne représente les pixels choisis. On réalisera ensuite une ACP (classique) sur ce tableau en ne considérant que les deux premières directions principales. On obtient une représentation 2D des pixels considérés avec leurs attributs (positions, couleurs, etc...).
4. Utiliser l'algorithme des k-moyennes pour créer des catégories/partitions entre les pixels. On associera à chaque catégorie une couleur afin de créer un « masque ponctuel » sur l'image.
5. On utilisera enfin l'algorithme d'inpainting (on programmera cela en Ma321) d'OpenCV. Cet algorithme permettra de donner une couleur de catégorie à tous les pixels de l'image (et non uniquement aux pixels choisis initialement). On crée ainsi un masque sur l'image totale.

On obtient ainsi un partitionnement de l'image. Avant de passer à l'algorithme global nous allons commencer par rappeler l'algorithme de l'analyse en composantes principales (ACP) puis dans une seconde partie l'algorithme des k-moyennes.

Partie 1 : L'ACP le retour du retour du retour...

Le but de notre future application de l'ACP est de le faire sur les pixels d'une image. Pour donner un ordre de grandeur, une petite image disons de 480 par 640 pixels contient : 307200 pixels... Comme présentée dans l'énoncé de l'exercice, nous allons devoir choisir (au hasard) un certain nombre de points. Néanmoins même si nous choisissons 10 000 pixels, réaliser une ACP en utilisant la décomposition SVD de $X_c = U\Sigma V^T$ (cf. notation de l'exercice 1 du TD2) est impossible. Dans le cas de 10 000 points la matrice U serait de taille 10 000 par 10 000.... grandeur qui dépasse l'utilisation mémoire autorisée d'un PC standard. Nous allons devoir donner une formulation moins gourmandes en espace mémoire de l'ACP.

Pour cela rappelons les notions de l'exerce 1 du TD2.

Considérons un tableau de données de la forme :

	X_1	X_2	$\cdots$	X_d
ind_1	x_{11}	x_{12}	$\cdots$	x_{1d}
ind_2	x_{21}	x_{22}	$\cdots$	x_{2d}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
ind_n	x_{n1}	x_{n2}	$\cdots$	x_{nd}

On notera la matrice $X \in \mathcal{M}_{nd}(\mathbb{R})$, la matrice associée à ces données. Chaque ligne (la ligne i sera notée ind_i) représente les données d'un individus et les colonnes (notée X_j) un type de donnée « mesurée » (i.e. les valeurs d'une variable aléatoire). Notons :

$$ind_g := \frac{1}{n}\sum_{i=1}^{n} ind_i$$

le vecteur ligne centre de gravité et posons la nouvelle matrice centrée t posons la nouvelle matrice centrée :

$$X_c := \begin{pmatrix} ind_1 - ind_g \\ ind_2 - ind_g \\ \vdots \\ ind_n - ind_g \end{pmatrix}$$

et l'on « réduit » la matrice en divisant chaque colonne de X_c par sa norme. Notons X_{rc} la forme centrée-réduite des données X.

On définit alors **l'estimateur de la matrice de covariance** par :

$$C := X_{rc}^T X_{rc} \in \mathcal{M}_{d,d}(\mathbb{R})$$

On définit **la première composante principale** comme le vecteur centré où :

$$v = arg \max_{\|v\|=1} Var(X_c v)$$

On appelle un tel v, **la direction principale**. Ainsi on rappelle qu'il nous faut résoudre le problème suivant :

$$arg \max_{\|v\|=1} Var(X_c v) \simeq arg \max_{\|v\|=1} v^T C v$$

où l'on remplace dans la dernière égalité la matrice de covariance par son estimateur.

1. Justifier que les directions principales sont données par les vecteurs propres de U dans la diagonalisation de la matrice de covariance :
$$C = UDU^T$$
si l'on range les valeurs propres de C par ordre décroissant. De plus justifier que la projection des données X sur les q-premiers axes principaux est données par :
$$X(q) := X_{rc} U_q$$
où U_q est la matrice de taille (d, q) formée des q-premiers vecteurs colonnes de U.
2. Démontrer que cette diagonalisation est obtenue en réalisant une SVD de C.
3. Rédiger une fonction Python : **ACP(X,q)** réalisant l'ACP d'un tableau de donnée X et renvoyant la matrice $X(q)$. On pourra également prendre la valeur de q donnée par la règle de Kaiser donnée sous la forme : q est le plus grand des indices tels que :
$$\lambda_q \geq \frac{1}{d}\sum_{\lambda \in Spec(C)} \lambda$$

4. Pour tester la robustesse de votre algorithme lancer-le sur les données déjà rencontrées en TDs : **iris.csv** et **mnist_train.csv**.
5. Certaines variables peuvent avoir une plus grande importance que d'autres, comme par exemple on peut vouloir que la position (i, j) des pixels soit plus valorisé que leurs couleurs. On peut alors introduire une matrice diagonale :

$$\sqrt{W} := \begin{pmatrix} \sqrt{w_1} & & & \\ & \sqrt{w_1} & & \\ & & \ddots & \\ & & & \sqrt{w_d} \end{pmatrix} \in \mathcal{M}_d(\mathbb{R})$$

et modifier la matrice de covariance par une version pondérée :

$$C_W = \frac{1}{\sum_{i=1}^{d} w_i} \left(X_{rc}\sqrt{W}\right)^T X_{rc}\sqrt{W}$$

Rédiger une fonction Python : **ACPpond(X,q,W)** réalisant l'ACP pondérée par une matrice $W = \sqrt{W}^2$. Tester en changeant la matrice W le résultat obtenue avec la base de donnée **iris.csv** et **mnist_train.csv**.

Partie 2 : L'algorithme des k-moyennes.

Dans cette partie nous allons voir (revoir si vous avez déjà réaliser le bonus du BE de Ma314) un des algorithmes les plus connus « résolvant » le problème du partitionnement de données : l'**algorithme des k-moyennes** (**k-means** en anglais). Nous allons commencer par décrire formellement le problème du partitionnement de données.

Soient $(a_i)_{i\in[\![1,m]\!]}$, m vecteurs (vu comme vecteurs colonnes) de $\mathbb{R}^n$. Fixons k un entier et notons $D = \{a_1, \cdots, a_m\}$ l'ensemble de ces vecteurs vu comme un ensemble à m éléments. On rappelle que $\mathcal{P}(D)$ note l'ensemble des parties de D. Le problème du partitionnement de données consiste à trouver une partition $S = \{S_1, S_2, \cdots, S_k\}$ de l'ensemble D en k parties minimisant la quantité suivante :

$$\min_{S\in\mathcal{P}(D)} \sum_{j=1}^{k} \sum_{a_i\in S_j} \|a_i - \mu_j\|_2^2$$

où μ_j est le barycentre de l'ensemble S_j i.e. :

$$\mu_j = \frac{1}{Card(S_j)} \sum_{a_i\in S_j} a_i$$

avec $Card(S_j)$ le cardinal (=nombre d'éléments) de l'ensemble S_j.

Nous allons voir l'**algorithme des k-moyennes** qui se veut une solution heuristique au problème de partitionnement des données.

Algorithme des k-moyennes classique : Soit $A \in \mathcal{M}_{mn}(\mathbb{R})$ une matrice que nous allons considérée par blocs de lignes :

$$A := \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix}$$

avec a_i le vecteur ligne : $a_i := (a_{i1}, a_{i2}, \cdots, a_{in})$.

Créer un programme/fonction $Kmoy(A, k, \epsilon)$ qui prend en entrée une matrice A, un entier $k \geq 2$ et un réel $\epsilon > 0$ et qui renvoie une liste de longueur k de matrices incarnant une partition de taille k de l'ensemble : $D = \{a_1, \cdots, a_m\}$ en suivant la procédure suivante :

1. **Étape 1 :** choisir k vecteurs a_l distincts aux hasards parmi les lignes de A :

$$\{a_{l_1}, a_{l_2}, \cdots, a_{l_k}\}$$

et définir : $\mu_j^{(0)} := a_{l_j}$ pour tout $j \in [\![1, k]\!]$.

2. **Étape 2 :** définir les partitions $S_j^{(1)}$ qui contiennent les a_i les plus proches de $\mu_j^{(0)}$ i.e. :

$$S_j^{(1)} := \left\{a_i \mid \left\|a_i - \mu_j^{(0)}\right\|_2 \leq \left\|a_i - \mu_p^{(0)}\right\|_2, \forall p \in [\![1, k]\!] \setminus \{j\}\right\}$$

3. **Étape 3 :** calculer les nouveaux barycentres des partitions $S_j^{(1)}$:

$$\mu_j^{(1)} := \frac{1}{Card(S_j^{(1)})} \sum_{a_i \in S_j^{(1)}} a_i$$

et définir la matrice (considérée par blocs lignes) :

$$\mu^{(1)} := \begin{pmatrix} \mu_1^{(1)} \\ \mu_2^{(1)} \\ \vdots \\ \mu_k^{(1)} \end{pmatrix}$$

4. **Étape 4 :** Itérer les étapes 2 et 3 jusqu'à ce qu'il existe un entier f :

$$\left\| \mu^{(f-1)} - \mu^{(f)} \right\| \leq \epsilon$$

5. **Étape 5 :** Renvoyer une matrice de la forme par blocs suivante :

$$S := \begin{pmatrix} & 1 \\ S_1 & \vdots \\ & 1 \\ & 2 \\ S_2 & \vdots \\ & 2 \\ \vdots & \vdots \\ & k \\ S_k & \vdots \\ & k \end{pmatrix}$$

où les S_j sont les matrices dont les vecteurs lignes sont données par les vecteurs de la partition $S_j^{(f)}$ et la dernière colonne le numéro j de la partition associée.

On prendra $\epsilon < 10^{-15}$ (l'algorithme converge rapidement donc on peut prendre des ϵ « très petit »).

Partie 3 : L'algorithme de partitionnement.

Pour cette partie il vous sera demandé de créer deux fichiers :

— un fichier : mabiblio.py : qui contiendra l'ensemble des fonctions programmées,

— un fichier : main.py : qui contiendra les applications du partitionnement sur les images.

Nous allons programmer sous forme de fonctions séparées chacun des points annoncés en début d'exercice. Néanmoins lorsque vous serez assurés du bon fonctionnement de vos fonctions vous pouvez créer (cela est même conseillé) une fonction globale réalisant la tache de partitionnement d'une image.

1. Créer une fonction **choixpts(img, nbpts)** qui à partir d'une image **img** (couleur) et d'un nombre de points **nbpts** renvoie une liste L de nbpts choisit au hasard en suivant une loi uniforme sur les lignes, sauf la première et la dernière, de même sur les colonnes sauf la première et la dernière.
2. Une fonction **data_pixels(img, L)** qui prend en entrée une image **img** et **L** la sortie de la fonction précédente et qui renvoie une matrice de taille nbpts×8 et on les lignes sont données par :

$$(c_{i1}, c_{i2}, c_{i3}, c_{i4}, c_{i5}, c_{i6}, c_{i7}, c_{i8})$$

avec :

— c_{i1} l'indice de ligne du pixel i, c_{i2} l'indice de colonne,

— c_{i3}, c_{i4}, c_5 les trois intensités de couleurs sur les trois couches de l'image,

— c_{i6}, c_{i7}, c_{i8} les trois moyennes des intensités de couleurs sur les 9 pixels autours du pixel i (lui compris).

On notera **data_img** la matrice de sortie précédente.

3. Une fonction **ACPimg(data_img)** qui réalise une ACP avec $q = 2$ des données **data_img**.
4. Cette étape comportera deux fonctions.
   (a) Une fonction **Kmoyimg(data_img,qkayser)** qui réalise un partitionnement à l'aide de l'algorithme des k-moyennes en **qkayser** parties où qkayser est donnée par la règle de Kayser lors de l'ACP précédente.
   (b) Une fonction **Masque(S)** qui prend en entrée la matrice de sortie de l'algorithme S et qui crée en sortie une image de même taille que **img**, l'image de départ, dont les pixels sont :
   — soit noir si le pixel (i, j) n'est pas dans les pixels choisis initialement,
   — soit d'une couleur différente pour chaque catégorie produite par l'algorithme des k-moyennes.
   On notera **imgmasqueponctuel** par la suite cette image de sortie.
5. On utilisera l'algorithme d'inpainting [2] (algorithme que nous programmerons dans un futur très proche... Ma321...). Cet algorithme issu de la mécanique des fluides permet de faire « diffuser » les couleurs des pixels choisis et coloriés du masque sur les pixels noirs (les non choisis). On utiliseras les commandes suivantes que l'on encapsulera dans une fonction : **RemplissageMasque(imgmasqueponctuel)** où **imgmasqueponctuel** est l'image représentant le masque ponctuel précédent :

```
mask=cv2.cvtColor(imgmasqueponctuel, cv2.COLOR_BGR2GRAY)
mask[mask>0]=255
mask=255*np.ones(np.shape(mask))-mask
imgmasque = cv2.inpaint(imgmasqueponctuel,np.uint8(mask),3,cv2.INPAINT_NS)
```

   Cette fonction renverra en sortie l'image **imgmasque**. On pourra afficher cette image grâce à la commande cv2.imshow() afin de voir apparaître les différentes zones trouvées par l'algorithme. On pourra également afficher les zones de l'image originelle dans des images séparées.

6. Modifier l'algorithme précédent en remplaçant l'ACP classique par une ACP pondérée. Commenter les résultats finaux en fonction des poids et des images utilisés.

En guise d'exemple, considérons l'image suivante :

Figure 2 – Image originelle

Après avoir choisi 10 000 pixels au hasard de manière uniforme puis réaliser l'ACP sur la matrice data_img, on trouve ici qkayser=2, on obtient alors le masque ponctuel et le masque global suivants :

2. cf. https://docs.opencv.org/3.4/df/d3d/tutorial_py_inpainting.html

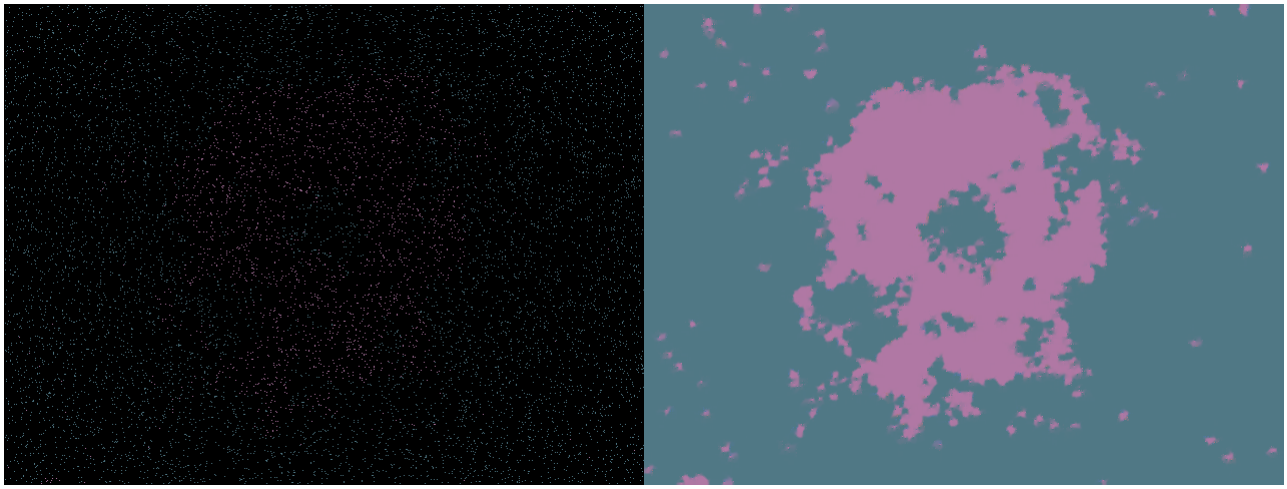

FIGURE 3 – Masque ponctuel (à gauche), masque après inpainting (à droite)

En utilisant le masque global on obtient enfin le résultat suivant sur l'image de base :

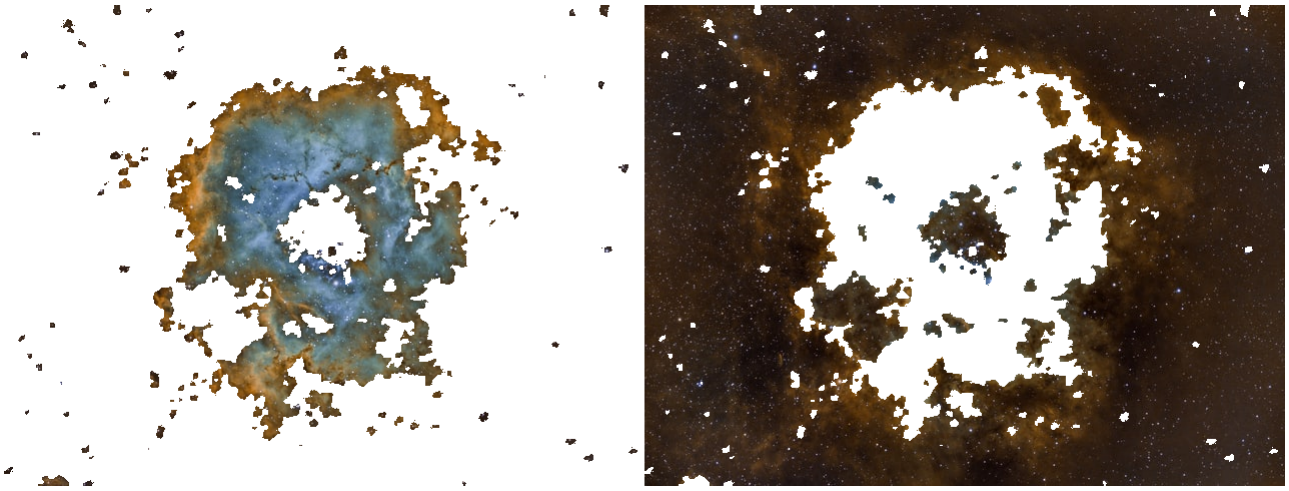

FIGURE 4 – Rendu final avec les couches séparées.