



RAPPORT DE PROJET

NOVEMBRE 2024

4IM01

Object Removal by Exemplar-Based Inpainting

CLÉMENT GILLI
JEAN GROËNINGER

ENCADRANT : ARTHUR LECLAIRE

Table des matières

1	Introduction	2
2	Formalisation du problème	2
3	Implémentation et choix faits	3
3.1	Structure de données	3
3.2	Calcul de ∇I_p sur la frontière	4
3.3	Calcul de n_p	5
3.4	Recherche du plus proche voisin	5
3.5	Zone de recherche	7
4	Utilisation	8
5	Résultats	9
6	Améliorations possibles	12
7	Conclusion	14
	Références	15

1 Introduction

Le projet consiste à faire de l'inpainting d'image, c'est-à-dire enlever des éléments d'une image (comme un objet masquant une partie d'un paysage) de manière automatique.

Il existe dans la littérature différentes méthodes pour accomplir cette tâche. L'approche utilisée dans le papier de recherche étudié [1] repose sur une combinaison de deux approches différentes pour en tirer les avantages des deux.

L'algorithme est assez simple à comprendre :

- on choisit un patch à remplir sur la frontière entre la zone connue et la zone à remplir
- on cherche un patch dans toute l'image qui se rapproche le plus du patch choisi
- on colle le patch trouvé au bon endroit

Le point clé du papier repose sur l'ordre de choix des patch à remplir. En effet, on pourrait appliquer l'algorithme décrit ci-dessus en choisissant les patchs aléatoirement, ou en tournant dans un sens. Mais tout le papier tend à montrer que leur méthode pour choisir les patchs est plus efficace.

2 Formalisation du problème

L'algorithme repose sur 3 étapes que nous allons détailler juste après les notations.

Notation :

- Φ : la région d'origine
- Ω : la région masquée à compléter
- $\delta\Omega$: la frontière de la région masquée
- Ψ_p : un patch de taille $k \times k$ pixels, centré en p

1. Calcul des priorités

D'abord il faut calculer la priorité $P(p) = C(p)D(p)$ de chaque patch Ψ_p de l'image, où $C(p)$ est le terme dit de *confidence* et $D(p)$ est le terme dit *data*. Durant l'initialisation, on attribue les valeurs suivantes :

$$C(p) = \begin{cases} 0, & \forall p \in \Omega \\ 1, & \forall p \in \bar{\Omega} \end{cases}$$

Ensuite, on peut calculer les 2 termes pour chaque patch de l'image avec

$$C(p) = \frac{\sum_{q \in \Psi_p \cap \bar{\Omega}} C(q)}{|\Psi_p|} \quad D(p) = \frac{|\nabla I_p^\perp \cdot n_p|}{\alpha}$$

où $|\Psi_p|$ est l'aire de Ψ_p , α est un facteur de normalisation, n_p est un vecteur orthogonal unitaire à $\delta\Omega$ au point p et ∇I_p^\perp est l'isophote (direction et intensité) au point p .

Interprétation :

Le terme de *confidence* est le ratio de valeurs de l'image déjà connu sur la taille du patch. Plus $D(p)$ est grand, moins il y a de nouvelles valeurs à déterminer dans le patch p .

Le terme *data* quantifie la situation où un contour franchit la frontière. On cherche à conserver les contours, donc on assigne une priorité accrue aux zones avec une grande valeur de $D(p)$.

2. Propagation des structures et de la texture

Une fois toutes les priorités calculées, on choisit le patch qui maximise P :

$$\Psi_{\hat{\mathbf{p}}} \mid \hat{\mathbf{p}} = \arg \max_{\mathbf{p} \in \delta\Omega} P(\mathbf{p})$$

On cherche ensuite le patch qui se rapproche le plus de celui-ci dans la région d'origine Φ :

$$\Psi_{\hat{\mathbf{q}}} = \arg \min_{\Psi_{\mathbf{q}} \in \Phi} d(\Psi_{\hat{\mathbf{p}}}, \Psi_{\mathbf{q}})$$

où d est la distance entre deux patchs définies par la somme des carrés des différences des pixels déjà remplis dans les 2 patchs.

Enfin, on copie chaque valeur de pixel du patch trouvé $\Psi_{\hat{\mathbf{q}}}$ sur le patch cible $\Psi_{\hat{\mathbf{p}}}$ (uniquement dans la zone manquante du patch cible).

3. Mise à jour des termes de confidence

Une fois cela fait, il ne reste plus qu'à mettre à jour les priorités des patchs. Le terme *data* dépend du gradient et du vecteur normal de la frontière, il doit donc être recalculer. De même, on met à jour le terme *confidence*.

$$\forall \mathbf{q} \in \Psi_{\hat{\mathbf{p}}} \cap \Omega, C(\mathbf{q}) = C(\hat{\mathbf{p}})$$

Ainsi, on répète ces 3 étapes qui forment une itération de notre algorithme d'inpainting, et cela jusqu'à ce que l'image soit complètement remplie, ie. $\Omega = \emptyset$

3 Implémentation et choix faits

L'algorithme est simple mais lorsqu'on essaye de l'implémenter, on est vite confronté à des choix et à des problèmes.

Dès le début, nous savions que l'algorithme était très coûteux, c'est pourquoi nous avons cherché à optimiser directement chaque étape de notre code pour ne pas revenir dessus par la suite.

3.1 Structure de données

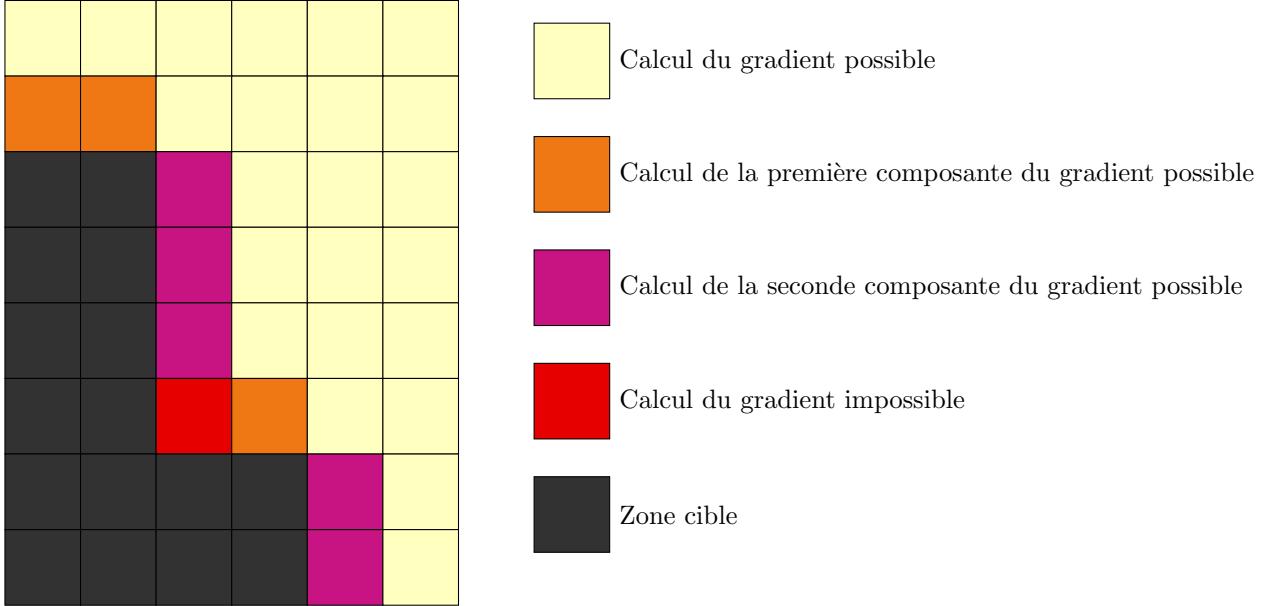
Le tout premier choix était celui de la structure de donnée : comment allons-nous stocker l'image de manière efficace pour l'utilisation de notre algorithme. Cet algorithme se repose sur une méthode par patch, c'est pourquoi nous avons décidé de créer un objet *PatchedImage* qui permet d'accéder facilement aux patchs. Les patchs sont des carrés de taille $2k + 1$, ce qui leur permet d'être centrés. De plus, on définit deux matrices de la taille de l'image où on stocke directement les priorités et la zone de chaque patch (Φ , Ω ou $\delta\Omega$).

À partir de cela, nous avons pu commencer à implémenter différentes fonctions nécessaires pour l'algorithme, notamment le calcul des priorités de chaque patch. Le calcul du terme de *confidence* était assez facile à mettre en place une fois qu'on avait bien posé le problème dans le code. Le terme de *data* était quant à lui plus compliqué à implémenter :

Pour calculer $D(\mathbf{p})$ il faut préalablement déterminer $\nabla I_{\mathbf{p}}$ et $\mathbf{n}_{\mathbf{p}}$.

3.2 Calcul de ∇I_p sur la frontière

Sur la frontière, le manque d'information sur la région cible rend le calcul du gradient impossible.



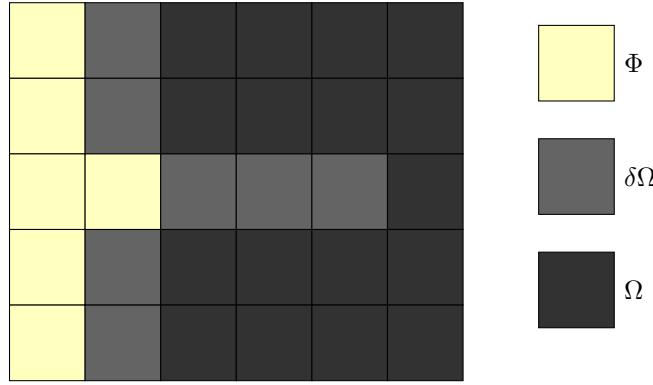
En effet, par définition de la frontière, si $(a, b) \in \delta\Omega$ alors $(a - 1, b) \in \Omega$, ou $(a + 1, b) \in \Omega$, ou bien $(a, b - 1) \in \Omega$, ou encore $(a, b + 1) \in \Omega$ donc $I(a - 1, b) = NaN$, ou $I(a + 1, b) = NaN$, ou $I(a, b - 1) = NaN$, ou enfin $I(a, b + 1) = NaN$. Cela rend le calcul du gradient en (a, b) impossible.

Il faut donc attribuer une valeur au gradient à la frontière. L'objectif de l'indicateur $D(\mathbf{p})$ est de prioriser les remplissage dans les zones où les contours croisent la frontière. Ainsi, le choix des valeurs à la frontière doit être fait en respectant cet objectif. Par exemple, attribuer la valeur 0 aux gradients à déterminer n'est pas envisageable. Nous avons fait le choix suivant :

$$\nabla I_{(\mathbf{a}, \mathbf{b})} = \begin{cases} \nabla I_{(\mathbf{a}, \mathbf{b})}, & \text{si } (a, b) \in \Phi, \\ (\frac{\partial}{\partial a} I_1(a, b), \frac{\partial}{\partial a} I_2(a, b + \varepsilon))^T, & \text{si } (a, b) \in \delta\Omega \text{ et } \exists \varepsilon \in \{-1, 1\}, (a, b + \varepsilon) \in \Phi, \\ (\frac{\partial}{\partial a} I_1(a + \varepsilon, b), \frac{\partial}{\partial a} I_2(a, b))^T, & \text{si } (a, b) \in \delta\Omega \text{ et } \exists \varepsilon \in \{-1, 1\}, (a + \varepsilon, b) \in \Phi, \\ \nabla I_{(\mathbf{a} + \varepsilon_1, \mathbf{b} + \varepsilon_2)} & \text{si } (a, b) \in \delta\Omega \text{ et } \exists \varepsilon_1, \varepsilon_2 \in \{-1, 1\}, (a + \varepsilon_1, b + \varepsilon_2) \in \Phi \\ 0 & \text{sinon} \end{cases}$$

Ainsi, en complétant le gradient avec des valeurs de gradients voisins, on peut penser que l'information sur les contours est préservée.

Dans le cas, où aucun gradient voisins n'est calculable, on attribue la valeur 0, le patch n'est pas prioritaire. On se trouve en fait dans une situation similaire à la suivante :

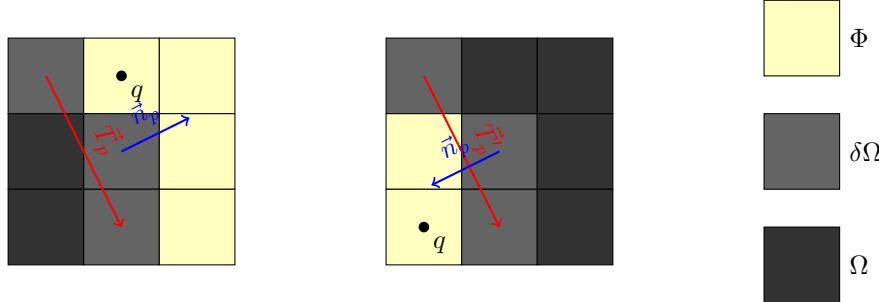


3.3 Calcul de n_p

Soit $p \in \delta\Omega$, soit $A_p = \{p' \in \mathbb{N}^2, \|p - p'\|_2 \leq 1\}$. On peut montrer que $A_p \cap \Phi \neq \emptyset$ et $A_p \cap \delta\Omega \neq \emptyset$.

Alors on peut définir $p_{min} = \min(A_p \cap \delta\Omega)$ et $p_{max} = \max(A_p \cap \delta\Omega)$ en utilisant l'ordre lexicographique. Puis $\vec{T}_p = \overrightarrow{p_{min}p_{max}}$ et $q \in A_p \cap \Phi$.

Alors $\vec{n}_p = \varepsilon \frac{\vec{T}_p^\perp}{\|\vec{T}_p\|_2}$ où $\varepsilon = \begin{cases} 1 & \text{si } q \text{ est au dessus de la droite passant par } p_{min} \text{ et } p_{max}, \\ -1 & \text{sinon} \end{cases}$.



On peut donc calculer $D(\mathbf{p}) = \frac{|\nabla I_p^\perp \cdot \mathbf{n}_p|}{\alpha}$ puis $P(\mathbf{p}) = C(\mathbf{p})D(\mathbf{p})$

Une fois le calcul des priorités effectuées, il faut déterminer un patch qui maximise P . Nous utilisons de la même manière la bibliothèque *numpy* pour optimiser le temps d'exécution en évitant l'usage des boucles de *python*.

Il ne reste plus qu'à trouver le patch le plus proche de celui avec la priorité maximale. C'est l'étape la plus coûteuse de l'algorithme et elle ne doit donc pas être négligée.

3.4 Recherche du plus proche voisin

L'idée naïve pour trouver le patch optimal est de calculer la distance entre le patch cible et tous les patchs de l'image. Ce n'est pas faisable en pratique : si on a une image de taille $N \times N$ et une taille de patch de $M \times M$, on doit alors réaliser $O(N^2 M^2)$ opérations à chaque fois qu'on cherche un patch optimal (sans compter les canaux RGB). Nous avons tout de même implémenté cette méthode naïve de manière à comparer les temps d'exécution avec la méthode efficace suivante.

L'article [2] répond à cette question. Il compare les différentes méthodes de plus proche voisin qui existent, plus particulièrement les algorithmes qui utilisent des arbres de recherche. D'après ce papier, les arbres les plus adaptés sont les *Ball Tree* et *vp-Tree*. Dans notre cas, ces deux méthodes ont des performances similaires car nous avons besoin de créer l'arbre une seule et unique fois (au début du programme). Or les Ball Tree sont déjà implémentés dans la bibliothèque *scikit-learn*, nous avons donc choisi d'implémenter les Ball Tree pour notre recherche optimale de patch le plus proche.

Pour utiliser l'algorithme, il faut aplatisir les patchs de taille $M \times M$ en vecteurs de taille M^2 , puis on stocke ces vecteurs dans un tableau. Il faut également préciser une fonction de distance spécifique pour la recherche. En effet, les valeurs de la zone cible ne doivent pas apparaître dans le calcul des distances. Il faut donc une fonction de distance entre 2 patchs qui prenne seulement en compte les pixels non-masqués.

L'algorithme de Ball Tree renvoie l'indice du patch le plus proche dans l'image aplatie. On calcule les coordonnées du centre du patch dans l'image d'origine. Puis, il suffit de coller sur la partie masquée du patch cible, la partie correspondante du patch renvoyé par le Ball Tree.

À noter que nous avons commencé par implémenter notre algorithme pour les images en niveau de gris, mais notre choix d'implémentation à rendu très facile le passage à la couleur : rien ne changeait à part le calcul de coordonnées pour retrouver le patch à partir de l'index renvoyé par le Ball Tree.

Method	Construction Performance	ϵ -NN Search Performance	k -NN Search Performance
<i>k</i> d-Tree	Excellent	Poor	Poor
PCA Tree	Poor	Fair	Fair
Ball Tree	Fair	Excellent	Excellent
<i>k</i> -Means	Poor	Good	Good
<i>vp</i> -Tree	Excellent	Excellent	Excellent

FIGURE 1 – Tableau récapitulatif des méthodes de recherche [2]

Nous avons tout de même implémenter notre propre version de KNN basique pour comparer le temps d'exécution (2).

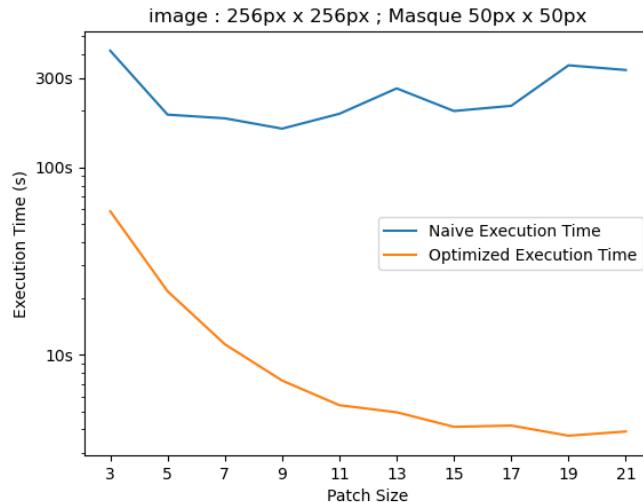


FIGURE 2 – Temps d'exécution des deux méthodes en fonction de la taille des patchs

Une amélioration notable des performances est observée entre les deux approches étudiées. La méthode naïve présente un temps d'exécution moyen d'environ 300 secondes pour le cas testé, et ce temps ne diminue pas significativement avec l'augmentation de la taille des patchs.

En revanche, la méthode optimisée montre une réduction drastique du temps d'exécution lorsque la taille des patchs augmente. Par exemple, le temps d'exécution passe d'environ 100 secondes pour des patchs de dimension $3\text{ px} \times 3\text{ px}$ à l'ordre de la seconde pour des patchs de dimension supérieure à $13\text{ px} \times 13\text{ px}$. Cela représente une amélioration d'environ deux ordres de grandeur (10^2) en termes de temps d'exécution.

Cette optimisation souligne l'efficacité de la nouvelle méthode, particulièrement pour des patchs de grande taille, ce qui en fait une solution plus adaptée pour des applications à grande échelle.

3.5 Zone de recherche

Jusqu'ici notre algorithme fonctionne bien et rapidement pour des images relativement petite (256x256 pixels, voire 512x512). Cependant pour des images plus grandes, le programme renvoyait une erreur : cela est dû à la création du Ball Tree car la taille du Ball Tree est proportionnelle à la taille de l'image et à la taille des patchs.

L'idée est donc de limiter la zone de recherche des patchs dans une région autour de la zone masquée. On peut supposer que les informations les plus pertinentes pour combler la zone cible sont proches de cette zone. Nous avons implémenter cette méthode. Le temps d'execution est plus faible et les résultats restent similaires. On peut tout de même changer la taille de la zone de recherche si on s'aperçoit que, pour une image spécifique, on ait besoin de patchs éloignés de la zone masquée.

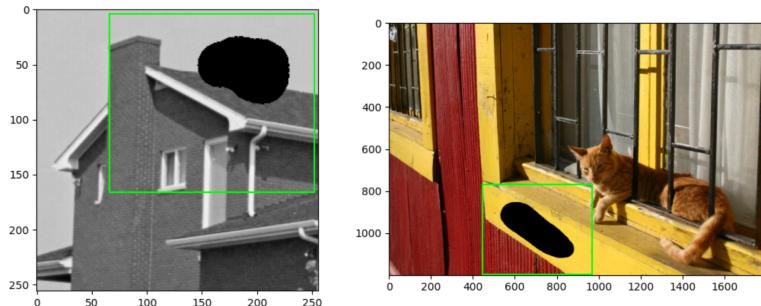


FIGURE 3 – En vert la zone de recherche

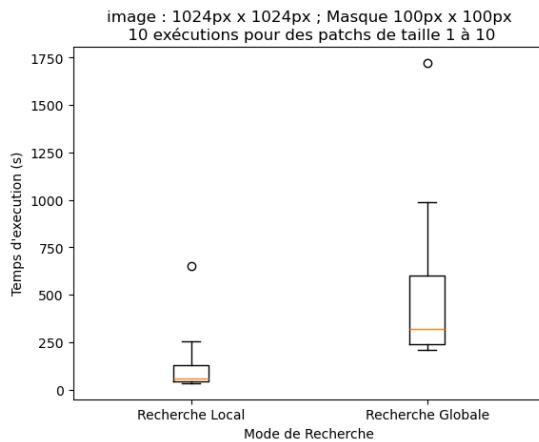


FIGURE 4 – BoxPlot du temps d'exécution pour la Recherche Locale et Globale

Les temps d'exécution pour la recherche locale sont nettement inférieurs, concentrés entre 150 et 300 secondes environ, tandis que ceux de la recherche globale présentent une médiane avoisinant 500 secondes. La recherche Locale est nettement plus rapide que la recherche Globale, avec une amélioration d'un facteur significatif (plus de 2 fois en médiane). À noter que dans le cas de très grandes images, nous ne pouvions même pas exécuter le programme, alors qu'avec la recherche Locale, nous pouvons tout à fait le faire (tant que la zone masquée n'est pas trop grande également).

Enfin, nous avons utilisé la bibliothèque python *numba* qui permet de compiler en C certaines fonctions, ce qui rend l'exécution encore plus rapide.

4 Utilisation

Notre but était de constituer un outil accessible pour faire de l'inpainting. C'est pourquoi, en plus du notebook pour faire le développement et nos tests, nous avons un fichier principal qui permet de lancer le programme facilement sur l'image désirée et en choisissant les options.

Voici la commande pour lancer le programme :

```
python3 main.py <path_image> <patch_size> <options>
```

Pour plus de détails :

```
python3 main.py --help
```

Généralement, on utilise un patch size égal à 4 (ce qui correspond à des patchs de taille 9x9) pour les petites images, et égal à 10 (soit du 21x21) pour les plus grandes. Le mode de recherche est par défaut en *Local* mais peut être changé en *Global*.

Pour faciliter l'utilisation de notre outil d'inpainting, nous avons rajouté une fonctionnalité permettant de dessiner directement avec la souris la zone à supprimer, ce qui est plus pratique pour une utilisation réelle (sur un éditeur d'image comme GIMP par exemple). Il y a également la possibilité de sauvegarder le masque dessiné, ce qui permet de le réutiliser ultérieurement.

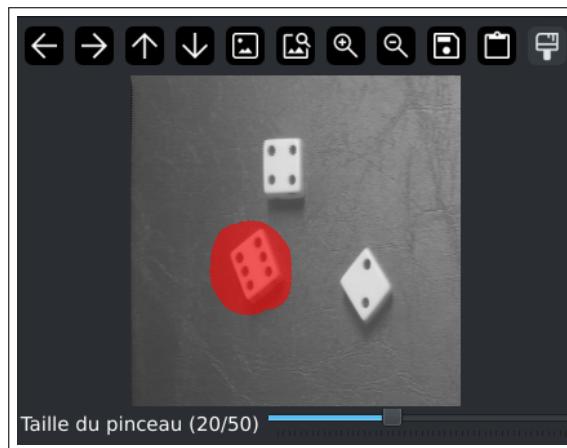


FIGURE 5 – Outil pour dessiner la zone à supprimer (ici en rouge)

De plus, nous affichons en temps réel la reconstruction de l'image. Cela n'a aucun impact sur les performances et permet de voir dans quel ordre est reconstruite l'image. Cela est pratique pour nos tests, mais également pour voir l'avancement du programme et de pouvoir interrompre l'algorithme en cas d'erreur sur le choix d'un paramètre.

5 Résultats

Les résultats de l'algorithme dépendent beaucoup des hyper-paramètres, ce qui rend difficile son analyse. En effet, lorsqu'on obtient des mauvais résultats, peut-être qu'en modifiant certains paramètres on peut en obtenir des biens meilleurs. Le paramètre qui influe le plus le résultat est la taille du patch. Pour l'illustrer, nous avons utiliser l'algorithme sur la même image (ainsi que le même masque) pour différentes tailles de patch (6).



Masque en rouge



9x9



13x13



17x17



21x21

FIGURE 6 – Résultat en fonction de la taille des patchs : le 9x9 est trop petit (on perd la cohérence entre les patchs) et le 21x21 trop grand (on voit distinctement les patchs)

Il est donc important de bien choisir la taille de patch utilisée pour une image donnée. Le deuxième paramètre qui influe beaucoup le résultat est le masque : si on prends deux masques quasiment identiques, on peut obtenir deux résultats distincts. Les résultats suivants ont été obtenus en choisissant la taille de patch qui marchait le mieux.

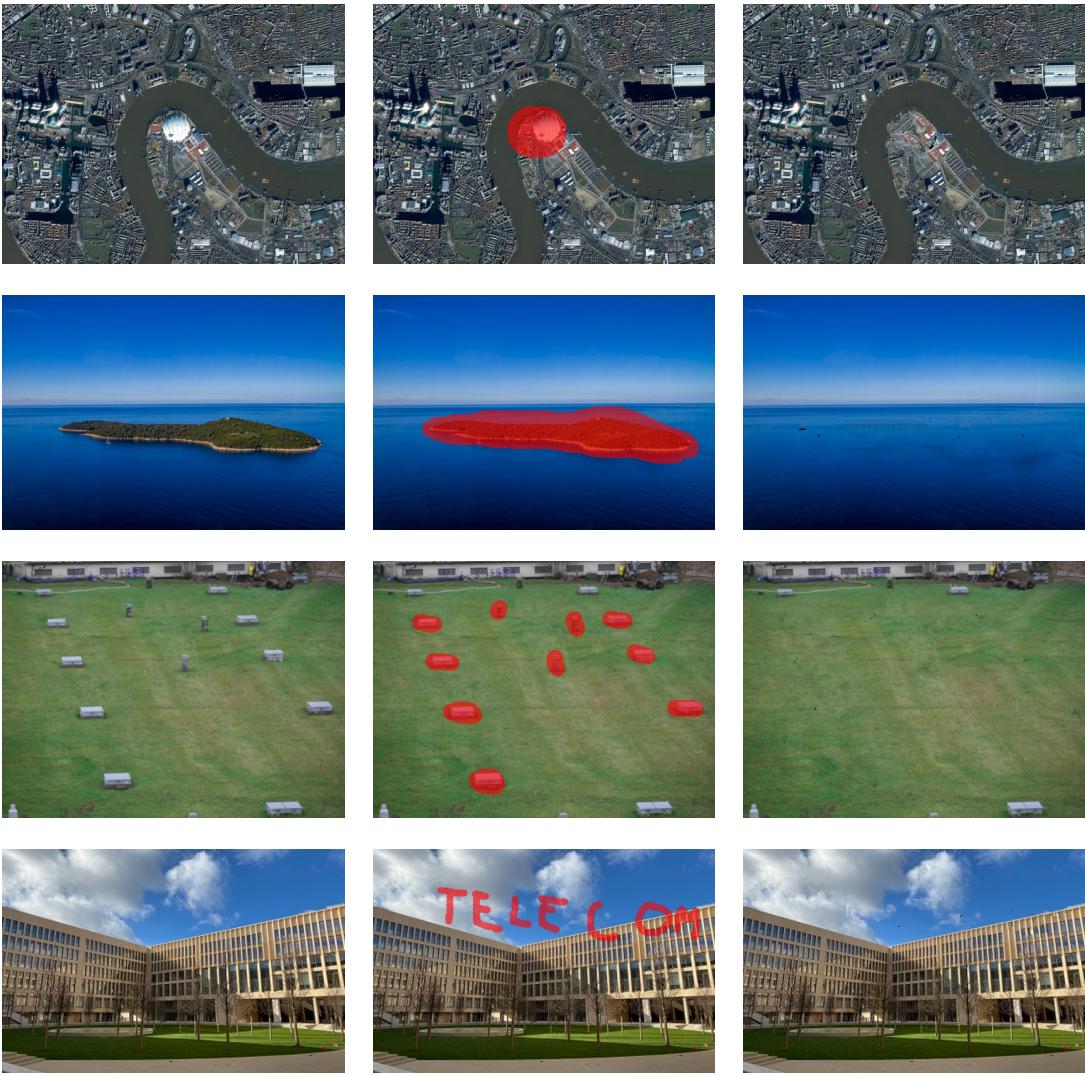


FIGURE 7 – De gauche à droite : l'image originale, le masque en rouge, le résultat

Les résultats sont très plaisants visuellement, les différentes parties de l'image sont respectées. On remarque notamment sur l'image de Télécom que les structures des fenêtres sont assez bien propagées. Le résultat le plus impressionnant est certainement celui du panneau (8). En effet, le poteau du panneau est bien prolongé tout le long afin d'obtenir un poteau connexe. Ici sans l'utilisation des priorités, nous n'aurions jamais obtenu ce résultat.



FIGURE 8 – Propagation des structures parfaite

Nous nous sommes également intéressés au cas des textures. Nous avons pu identifier 3 cas :

- la texture est un motif répété (9)
- la texture est gaussienne (10)
- la texture est non-gaussienne (11)

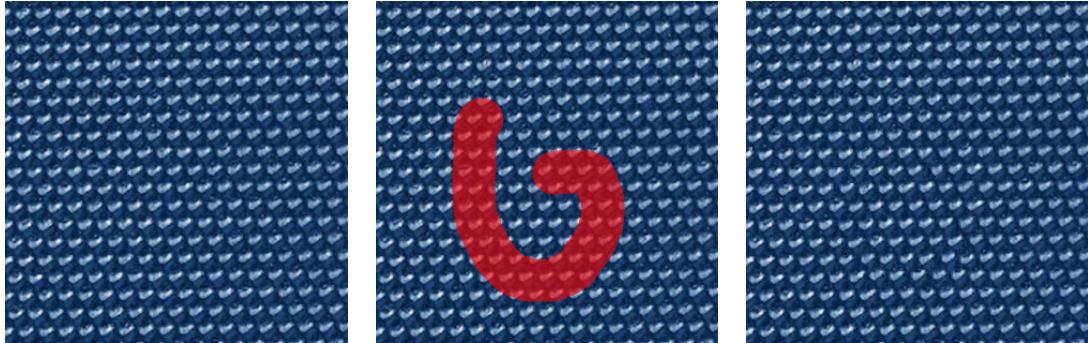


FIGURE 9 – Motif répété

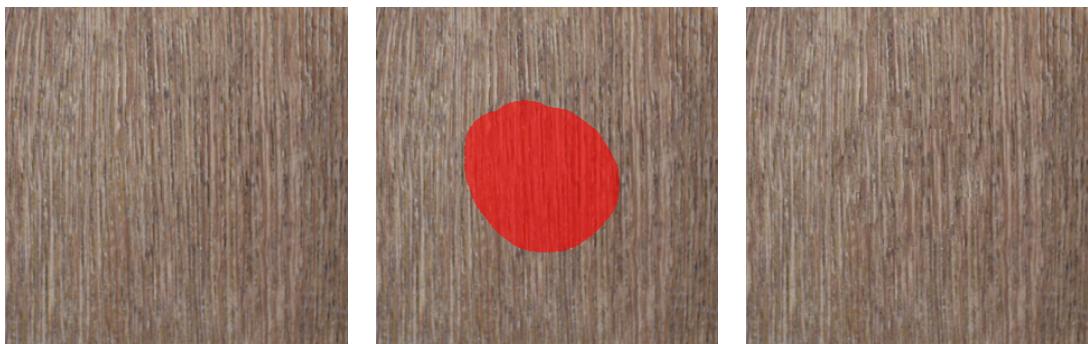


FIGURE 10 – Gaussien

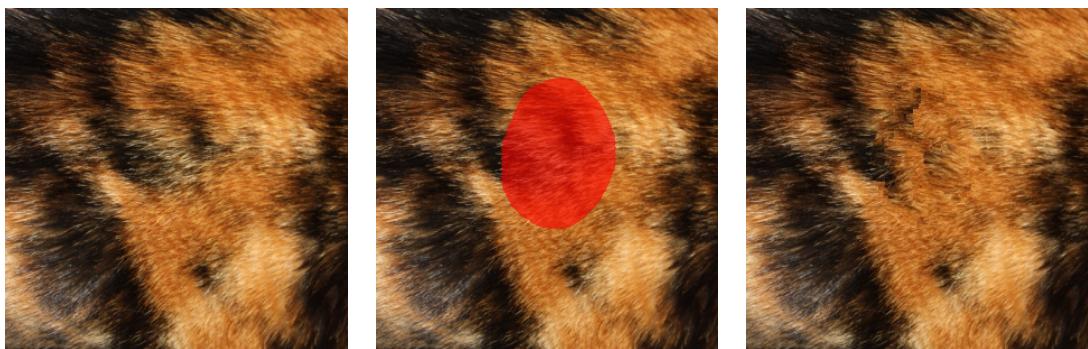


FIGURE 11 – Non-gaussien

On observe que dans le cas du motif, le résultat obtenu est parfait. Concernant le cas gaussien, le résultat est globalement satisfaisant mais il y a quelques problèmes au niveau du centre du masque car un des points faibles de la méthode présentée dans le papier est la jointure des différents morceaux reconstruits : il n'y a rien dans le papier qui permet de lisser les discontinuités au niveau des frontières entre les patchs. Enfin, le cas non-gaussien ne fonctionne pas vraiment : la méthode par patch n'est pas très adaptée ici.

Vous pouvez retrouver sur ce lien différents gifs qui permettent de visualiser l'ordre de remplissage (typiquement pour la propagation des structures) : [lien](#).

6 Améliorations possibles

Bien que nous ayons des résultats satisfaisants, plusieurs pistes d'amélioration sont possibles : nous pouvons essayer d'améliorer le rendu visuel, ou encore améliorer la complexité du programme (qui reste tout de même élevé pour des grandes images).

Concernant le rendu visuel, nous avons trouvé un papier qui reprend le papier de Criminisi en essayant de l'améliorer [3]. Le papier propose trois changements : changer le calcul des priorités, changer la fonction de distance (pour la recherche de plus proche voisin) et changer la synthèse du patch. Pour le changement du calcul des priorités, ils donnent plus de poids au terme de data qu'au terme de confidence :

$$\hat{P}(\mathbf{p}) = C(\mathbf{p}) \exp\left(\frac{D(\mathbf{p})}{2\sigma^2}\right)$$

Cela permet de prioriser encore plus la continuité des structures, car il est vrai que dans certains cas, on voudrait donner beaucoup plus de poids au terme de data (par exemple pour l'image du panneau 8). Pour la distance, ils combinent la distance SSD (sum of square distance, celle qu'on utilise déjà) et la distance de Hellinger. Cette distance est différente en fonction du type du patch, si c'est plutôt un patch de texture ou bien de structure (on ne détaille pas plus ici car c'est assez complexe). Enfin, le dernier changement consiste à synthétiser les patches à coller sur la zone masquée à partir de combinaisons linéaires des k patches les plus proches trouvés (au lieu de juste coller le patch le plus proche) : le poids de chaque patch est calculé en résolvant un problème d'optimisation. Nous avons essayé d'implémenter tout cela, mais le papier n'est pas très clair sur certains points et nous n'avons pas eu le temps d'approfondir dessus. Cela reste néanmoins une grande piste d'amélioration.

La deuxième chose à améliorer est la *jointure* entre les patches collés : comme on l'a observé dans le cas des textures gaussiennes (mais aussi dans le cas des autres images si on zoom), il n'y a rien qui limite les discontinuités au niveau des frontières entre les différents patchs. Il faudrait utiliser des procédures de recollement, mais malheureusement ces méthodes sont très coûteuses.

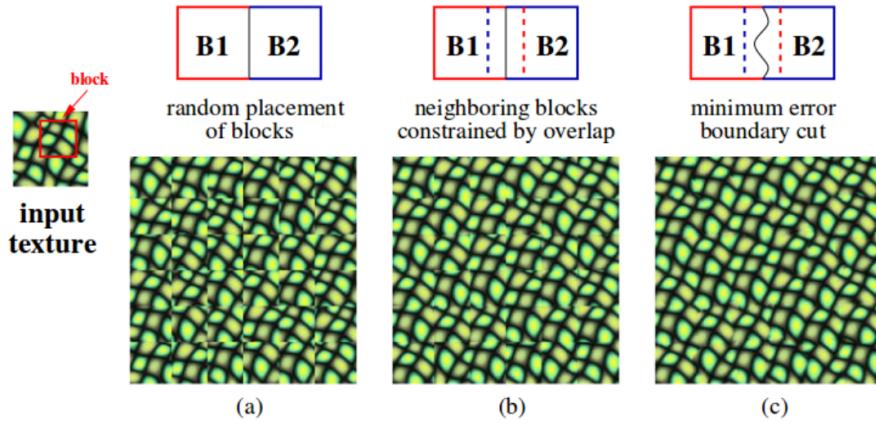


FIGURE 12 – Méthodes de recollement (Efros-Freeman 2001)

En ce qui concerne la complexité de l'algorithme, nous avons discuté d'une méthode avec M. Leclaire. Elle consiste à tout d'abord sous-échantillonner notre image. On applique ensuite l'algorithme d'inpainting sur cette image mais en conservant les positions des patchs collés (on stocke donc sous forme de dictionnaire les positions des patchs remplis (clés) ainsi que celles des patchs les plus proches associés (valeurs)). Enfin, on peut remplir l'image d'origine (en plus grande résolution) en collant les patchs à partir des positions de ceux trouvés et stockés en basse résolution. Il ne reste plus qu'à appliquer une étape de lissage pour enlever les artefacts dûs au ré-échantillonnage.

Cette méthode permet de gagner beaucoup de temps de calcul car on sait que le temps d'exécution est proportionnelle à la taille de l'image. On peut donc appliquer plusieurs fois cette méthode sur une même image pour la reconstruire petit à petit : par exemple, on part d'une image 1024x1024 pixels, on sous-échantillonne en 512x512, puis 128x128, on reconstruit en 128x128 et on passe à l'échelle supérieure deux fois pour obtenir l'image d'origine avec la zone masquée remplie.

Nous avons commencé à implémenter cette méthode mais cela reste expérimental : vous pouvez tout de même l'essayer en rajoutant une option lors de l'exécution du programme :

```
python3 main.py <path_image> <patch_size> <optional:resampling>
```

où *resampling* est un entier qui précise le coefficient de sous-échantillonnage.

Voici ci-dessous un résultat obtenu :

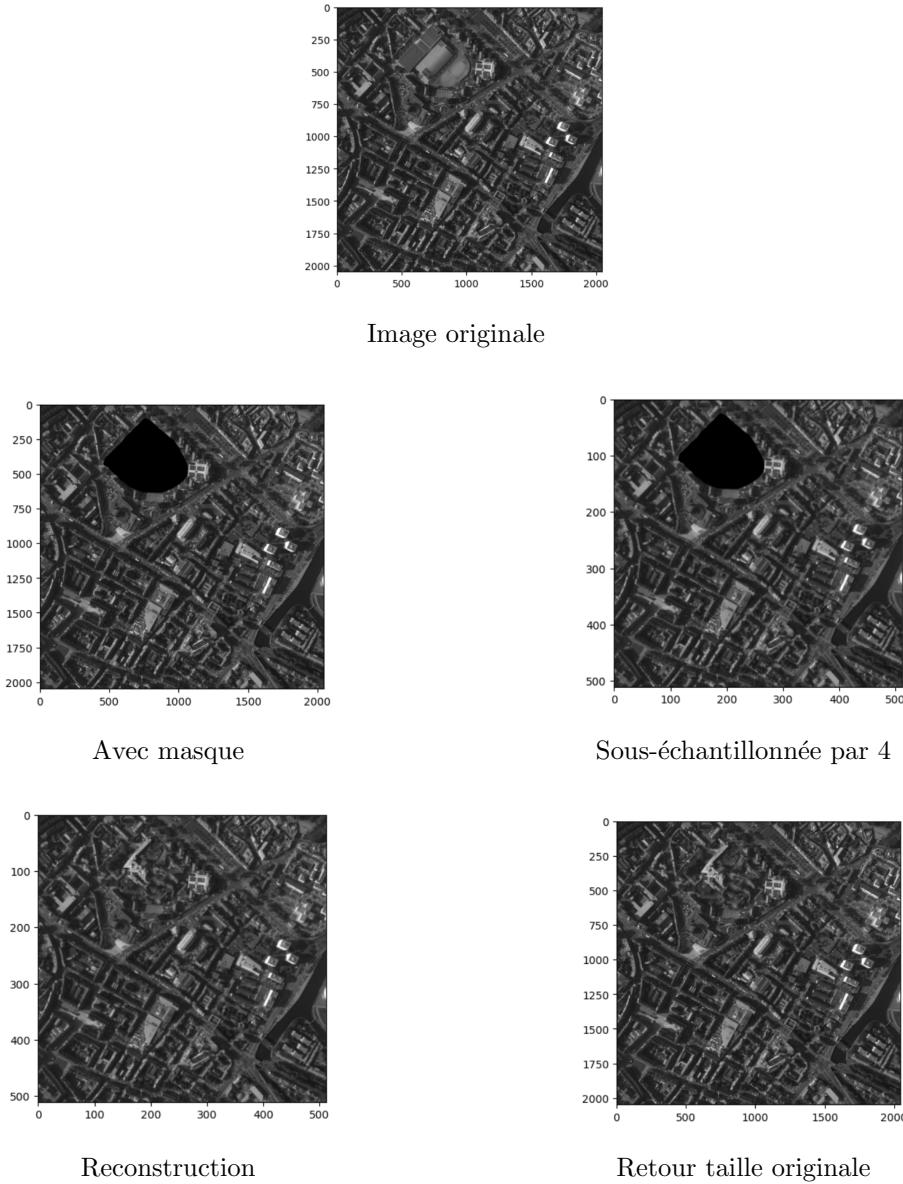


FIGURE 13 – Les différentes étapes de la méthode de ré-échantillonnage : avec la taille de l'image et du masque d'origine, nous n'aurions jamais pu obtenir de résultat sans utiliser cette méthode.

7 Conclusion

En conclusion, ce projet d'inpainting d'image a permis de mettre en œuvre un algorithme efficace basé sur des priorités pour remplir automatiquement des zones masquées, tout en préservant les structures et textures. Malgré la complexité et les défis rencontrés lors de l'implémentation, notamment pour optimiser les performances et ajuster les paramètres, les résultats obtenus sont globalement satisfaisants, avec des reconstructions visuellement cohérentes dans de nombreux cas. Cependant, les résultats sont fortement influencés par le choix des paramètres. Par exemple, une simple modification de la taille des patchs ou du masque peut transformer un résultat médiocre en une reconstruction de grande qualité, ce qui montre une certaine instabilité de l'algorithme.

Nous avons démontré que l'optimisation des recherches avec des structures comme les Ball Trees et la limitation de la zone de recherche, permet de gérer la complexité de l'algorithme, en particulier pour des images de grande taille.

Cependant, il reste des pistes d'amélioration importantes. Sur le plan visuel, des travaux supplémentaires pourraient intégrer des techniques de synthèse avancées pour améliorer la qualité des textures et réduire les artefacts aux jointures. Sur le plan des performances, des méthodes d'inpainting multi-résolution basées sur le sous-échantillonnage peuvent réduire davantage les temps de calcul, comme nous avons pu le voir.

Références

- [1] A. CRIMINISI, P. PÉREZ et K. TOYAMA. “Object Removal by Exemplar-Based Inpainting”. In : *Microsoft Research*. 2003.
- [2] Neeraj KUMAR, Li ZHANG et Shree NAYAR. “What is a Good Nearest Neighbors Algorithm for Finding Similar Patches in Images ?” In : *Columbia University*. 2008.
- [3] Raul Martinez NORIEGA, Aline ROUMY et Gilles BLANCHARD. “Exemplar-based image inpainting : fast priority and coherent Nearest Neighbor search”. In : *IEEE Workshop on Machine Learning for Signal Processing*. 2012.

Lien du git : <https://github.com/clementgilli/inpainting>