



MASTER MVA: REMOTE SENSING DATA: FROM SENSOR LARGE-SCALE
GEOSPATIAL DATA EXPLOITATION.

PROJECT REPORT

**Construction d'une carte de hauteur avec
l'algorithme PatchMatch.**

Valery **DEWIL**

Juliette **RENGOT**

Etudiants

Gabriele **FACIOLO**

Enric **MEINHARDT**

Carlo **DE FRANCIS**

Superviseurs

Mars 2019

Table des matières

I	Introduction et état de l'art	2
II	Présentation de la méthode proposée	3
II.1	Le principe général	3
II.2	Les résultats	5
II.2.1	Une première estimation d'une carte de hauteurs sur une zone urbaine de Jacksonville	5
II.2.2	Une analyse de l'influence de la taille des patchs utilisés.	6
II.2.3	Une étude de l'influence du choix de la zone d'intérêt considérée sur les performances	7
III	Quelques extensions possibles	8
III.1	Lisser et débruiter la carte de hauteurs	8
III.2	Rectifier les images	12
III.3	Prendre en compte un plus grand nombre d'images	13
III.4	Définir la grille de hauteurs dans le système <i>easting/northing</i>	14
IV	Présentation des détails d'implémentation	15
IV.1	Description des fonctions	15
IV.2	Complexité de l'algorithme	18
V	Discussion	19
VI	Références	21
	Appendices	23

I Introduction et état de l'art

L'objectif de ce projet est d'estimer la géométrie terrain d'un espace. Il s'agit de calculer la hauteur d'éléments remarquables dans l'environnement (bâtiments, arbres, monuments, reliefs...). Cela permet ensuite d'établir des cartes de hauteurs de la ville ou du paysage considéré.

L'estimation de cartes de hauteurs est un problème de recherche difficile et novateur. Plusieurs approches ont déjà été proposées. Par exemple, Jeon et al. [2015] ont utilisé une caméra "lenslet light field" pour estimer des correspondances stéréos multi-vues. D'autres outils de capture de données peuvent être utilisés. Les satellites offrent des images particulièrement adaptées à ce type de recherche. Ainsi, Qayyum et al. [2015] se sont intéressés au problème d'interférence des arbres et de la végétation avec les lignes à haute tension. Pour éviter ce problème, ils ont conçu un algorithme de mise en correspondances stéréo pour estimer des cartes de disparités à partir d'images satellites. Ils ont ensuite calculé les hauteurs des arbres proches des lignes à surveiller en construisant une relation inversement proportionnelle entre la disparité et la hauteur. Pour cette dernière étape, ils utilisent les algorithmes de programmation dynamique ou de mise en correspondances de blocs par minimisation d'énergie.

Wang et al. [2016] proposent un algorithme efficace pour estimer la géométrie du terrain survolé par plusieurs satellites et y associer un label. Leur approche se base sur l'algorithme *PatchMatch*. Cet algorithme a été conçu pour l'édition d'images (Barnes et al. [2009]). Il permet d'établir des correspondances entre deux images en alternant une étape d'estimation aléatoire des disparités entre deux patchs de l'image et une étape de correction des résultats. Cette approche a été largement adaptée pour les problèmes de stéréo : Besse et al. [2014] et Heise et al. [2013] montrent son efficacité à l'échelle de précision sous-pixel. Heise et al. [2013] intègrent cette approche dans un lissage variationnel, ce qui leur permet d'obtenir une carte de disparités régularisée. Zheng et al. [2014] ont aussi basé leur approche sur *PatchMatch*. Leur projet se focalise sur le choix des images sources à utiliser pour obtenir la meilleure carte de hauteurs possible. Ils obtiennent une méthode robuste contre les systèmes d'acquisition d'image non-structurées et hétérogènes. Par ailleurs, Bleyer et al. [2011] proposent une extension de l'algorithme *PatchMatch* pour se libérer de l'hypothèse selon laquelle les pixels au sein d'une même région ont une même disparité. Cette supposition n'est pas vérifiée pour les surfaces inclinées. Elle crée parfois des erreurs. La solution proposée consiste à estimer, pour chaque pixel, un plan 3D sur lequel la région est projetée. On utilise l'approximation des plus proches voisins selon ce plan. La propagation est ainsi modifiée.

Enfin, Fanello et al. [2017] mettent en avant une autre difficulté de l'estimation des cartes de hauteurs: leur construction peut être coûteuse. Leur modèle "HashMatch" est conçu

pour résoudre des problèmes de vision artificielle en général, de manière parallélisable et peu coûteuse en puissance de calcul. La complexité de ce modèle est moindre par rapport à une approche basée sur les réseaux de neurones (comme, par exemple, Liu et al. [2015], Godard et al. [2017] ou Darabi and Maldague [2002]).

Dans le cadre de ce projet, nous proposons une nouvelle approche pour l'estimation des cartes de hauteurs. L'objectif est d'adapter la méthode *PatchMatch* pour s'affranchir de l'étape de triangulation, obligatoire dans l'approche classique. Dans ce rapport, nous commencerons par présenter la méthode proposée dans sa version la plus simple. Ensuite, nous présenterons différentes extensions possibles pour améliorer et généraliser les résultats. Nous détaillerons ensuite les détails de l'implémentation. Enfin, nous conclurons et discuterons des possibles travaux futurs.

II Présentation de la méthode proposée

II.1 Le principe général

On considère deux images I_1 et I_2 , non rectifiées. A chacune de ces images est associée une fonction de localisation L , qui associe à chaque pixel de l'image ses coordonnées (dans le système longitude/latitude) dans l'environnement réel. On note P la fonction de projection, qui effectue la transformation inverse.

On définit ensuite une grille pour la carte de hauteurs à estimer. Le pas de la grille détermine la résolution. A chaque point de la grille, correspond un patch dans chacune des images. La taille du patch est un paramètre que l'utilisateur peut choisir et qui influence beaucoup les résultats (Section II.2.2).

Lors de la phase d'initialisation, on assigne à chaque point de la grille une hauteur aléatoire. Il est évident que la carte de hauteur ainsi obtenue est très éloignée de la vérité-terrain. Cependant, il est très peu probable qu'aucun point n'ait reçu une hauteur aléatoire cohérente avec la vraie valeur. En effet, si la carte de hauteurs comporte M pixels, la probabilité qu'un pixel soit mis à jour par une itération de *PatchMatch* vaut $\frac{1}{M}$. Par conséquent, la probabilité qu'un pixel ne soit pas mis à jour est égale à $\left(1 - \frac{1}{M}\right)$. Dès lors, la probabilité qu'aucun pixel ne soit mis à jour est de $\left(1 - \frac{1}{M}\right)^M$. Cette quantité tend (par valeur inférieure) vers $\frac{1}{e}$. Finalement, la probabilité qu'au moins un pixel soit mis à jour est $1 - \left(1 - \frac{1}{M}\right)^M$. Cette quantité reste toujours plus grande que $1 - \frac{1}{e} \approx 0.632$. Ceci nous donne donc une borne inférieure pour la probabilité d'améliorer la carte : si on arrondi, un peu *grossièrement*, cette valeur, on a presque qu'au moins deux chances sur trois d'améliorer la carte à chaque recherche aléatoire! A partir de cette constatation, nous allons chercher à repérer les bonnes prédictions et à propager l'information pour améliorer la qualité globale de la carte de hauteurs.

Pour cela, on projette chaque points de la grille sur les deux images (en utilisant les fonctions de projection) et on compare les couleurs des deux patchs obtenus. Notre fonction de coût est donc basée sur la ressemblance des patchs. On compte le nombre de bytes qui diffèrent. Plus la valeur de la fonction de coût est faible, plus la hauteur attribuée peut être considérée comme correcte par rapport à la vérité-terrain.

On essaye alors de propager cette valeur aux points voisins dans la grille de hauteurs. Un voisin prend cette nouvelle valeur si cela lui permet de diminuer sa valeur de fonction de coût. C'est l'étape de propagation. Pour accélérer les calculs, nous n'avons pas considéré les huit voisins qui entourent un point de la grille. Nous ne nous intéressons pas aux voisins situés sur les diagonales car si la hauteur ne se propage pas aux voisins directs (haut, bas, gauche, droite), elle n'a que très peu de chance de se propager en diagonale. En fait, nous regardons seulement les deux voisins directs qui ont déjà été mis à jour (Figure 1) et qui ont donc une hauteur déjà améliorée. Pour ne pas favoriser une direction particulière de propagation, l'algorithme alterne entre une propagation en sens direct (de haut en bas et de gauche à droite) et une propagation en sens indirect (de bas en haut et de droite à gauche).

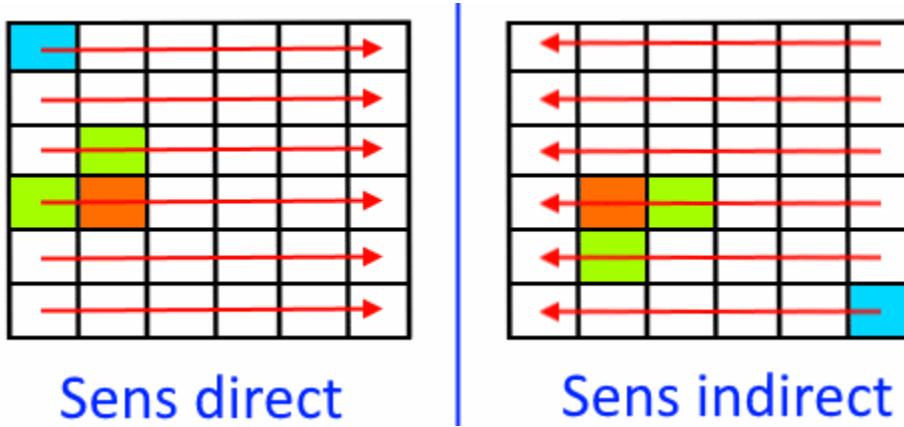


Figure 1: Schématisation des deux stratégies de parcours du voisinage pendant l'étape de propagation (*sens direct* à droite et *sens indirect* à gauche). On part de la case bleue et on parcourt la grille en suivant les flèches rouges. Pour un point donné (case orange), on regarde les deux voisins directs qui ont déjà été mis à jour (cases vertes).

Ensuite, l'étape de mise à jour aléatoire consiste à choisir aléatoirement une valeur de hauteur, dans l'intervalle plausible fixé au préalable. Nous gardons alors la meilleure des prédictions testées à cette itération et nous recommençons le processus. Le critère d'arrêt peut être un nombre d'itérations fixé et/ou un nombre minimal de modifications appliquées à la carte.

II.2 Les résultats

II.2.1 Une première estimation d'une carte de hauteurs sur une zone urbaine de Jacksonville

Nous avons testé cette version simple de l'algorithme sur deux images satellites, représentant la même zone urbaine de Jacksonville (Figure 2). Ces images sont rognées de façon à être de taille 400×400 . Nous avons choisi une taille de grille aussi égale à 400×400 , ce qui correspond à une résolution d'environ 30cm par pixel.

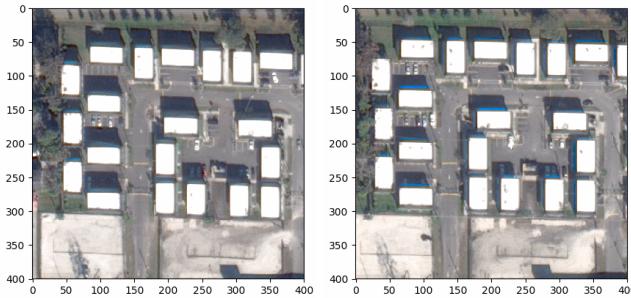


Figure 2: Les images satellites étudiées.

L'algorithme implémenté n'impose pas de contrainte sur le type d'image. Nous avons d'abord travaillé avec des images RGB, puis nous avons utilisé des images panchromatiques afin d'accéder à plus de détails dans la structure représentée. Les deux approches donnent des résultats très similaires.

Nous initialisons la carte de hauteur aléatoirement (Figure 3). Ensuite, nous itérons *PatchMatch* tant que les hauteurs de plus de 1000 points de la grille sont modifiées (dans un maximum de 20 itérations). Nous avons restreint la recherche des altitudes à l'intervalle $[-30, -10]$. Pour trouver un intervalle raisonnable de hauteurs, nous avons centré l'intervalle sur la hauteur de référence, donnée par la fonction "srtm4" du module "srtm4", pour les coordonnées longitude/latitude de la zone étudiée. Nous pourrions imaginer, dans un travail futur, une méthode adaptative qui corrigerait cet intervalle si cela permet d'améliorer les résultats. Dans une première expérience, nous considérons des patchs de taille 15×15 .

Une fois l'algorithme terminé, nous avons obtenu une carte de coûts relativement faibles (Figure 4). L'échelle de couleurs varie du rouge (coûts élevés) au bleu foncé (coûts faibles). Nous pouvons remarquer que le sol a tendance à avoir des coûts plus élevés que les bâtiments. L'estimation des hauteurs sera donc moins bonne dans ces zones.

La Figure 5 permet de visualiser la carte de hauteurs obtenue. Sur les visualisations tridimensionnelles, les pixels clairs correspondent aux zones les plus hautes de l'image

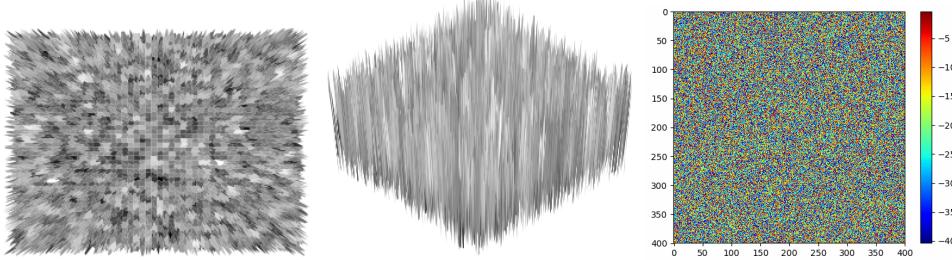


Figure 3: L'initialisation aléatoire de la carte de hauteurs. Visualisations tridimensionnelles (à gauche et au centre) et bidimensionnelle (à droite).

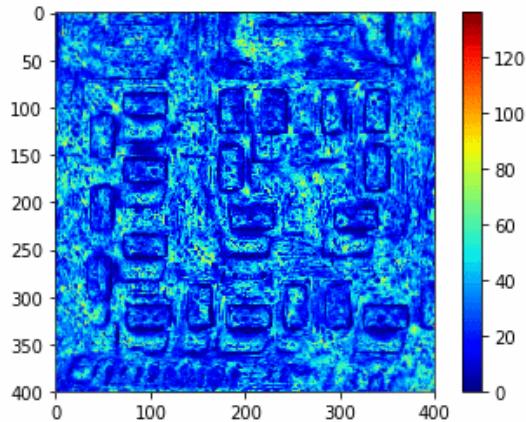
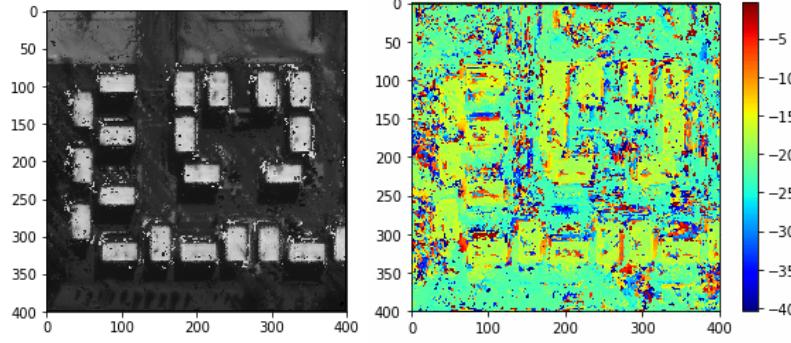


Figure 4: La carte des coûts obtenue avec la version simple de l'algorithme

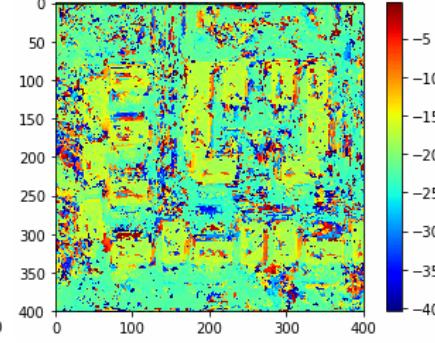
alors que les pixels sombres correspondent aux zones les moins élevées. Sur la visualisation bidimensionnelle, l'échelle de couleurs varie du rouge (zones hautes) au bleu (zones basses). Nous percevons nettement la forme des bâtiments mais le résultat reste bruité. Certaines zones possèdent de fortes variations d'altitude sans que cela ne soit expliqué par la réalité-terrain. Ceci est un problème car nous créons alors une fausse information. Il pourrait être préférable de ne pas avoir de renseignement sur les zones où la détection de hauteurs est trop difficile.

II.2.2 Une analyse de l'influence de la taille des patchs utilisés.

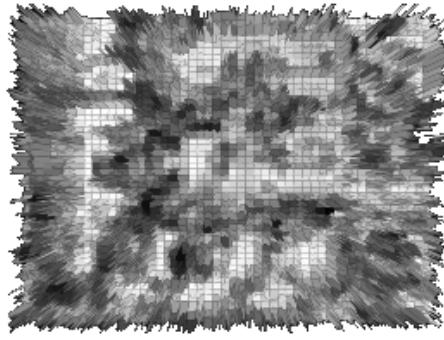
La mise en correspondances des différentes prises de vue est réalisée selon la ressemblance des patchs de l'image. Le patch associé à un pixel donné est défini comme un carré de taille $2 * \text{patch_size} + 1 \times 2 * \text{patch_size} + 1$ centré sur ce pixel. Le paramètre patch_size doit être choisi consciencieusement car il impacte fortement les résultats. Les coûts deviennent plus petits lorsque la valeur de patch_size est petite (Figure 6). Plus patch_size est grand, plus la carte de hauteurs présente de grandes zones homogènes (Figure 7). La carte est plus régulière et présente moins de bruit. Cependant, si nous choisissons une valeur trop élevée, les objets auront tendance à se dilater. La précision diminue. Ainsi, nous



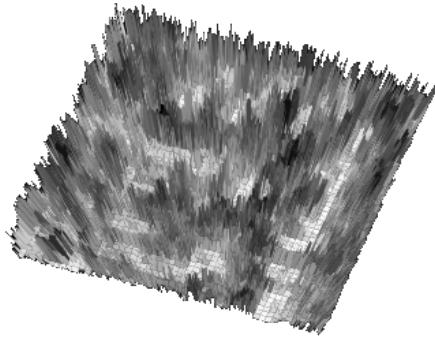
(a) Carte orthorectifiée avec les couleurs d'origine



(b) Visualisation bidimensionnelle



(c) Visualisation tridimensionnelle - Vue 1



(d) Visualisation tridimensionnelle - Vue 2

Figure 5: L'estimation de la carte de hauteurs obtenue avec la version simple de l'algorithme.

pouvons voir que lorsque $patch_size = 10$, nous perdons la séparation entre les différents bâtiments (Figure 7d).

II.2.3 Une étude de l'influence du choix de la zone d'intérêt considérée sur les performances

Nous remarquons que la qualité de l'estimation varie dans l'image. Certains endroits semblent plus difficiles à traiter que d'autres. Pour mettre en avant ce phénomène, nous avons testé l'algorithme sur plusieurs zones d'intérêt aux caractéristiques différentes (Figure 8).

L'algorithme n'est pas efficace dans un certain nombre de situations. Ainsi, les objets mobiles (comme des voitures dans la figure 8a) altèrent les performances. Comme nous cherchons des correspondances dans les deux images, l'approche n'est plus pertinente si trop de points dans une image ne peuvent pas être retrouvés dans la seconde image. De plus, les images doivent présenter suffisamment de détails et de textures pour permettre à l'algorithme d'avoir de bons résultats (Figures 8b et 8c). Nous remarquons aussi que sur

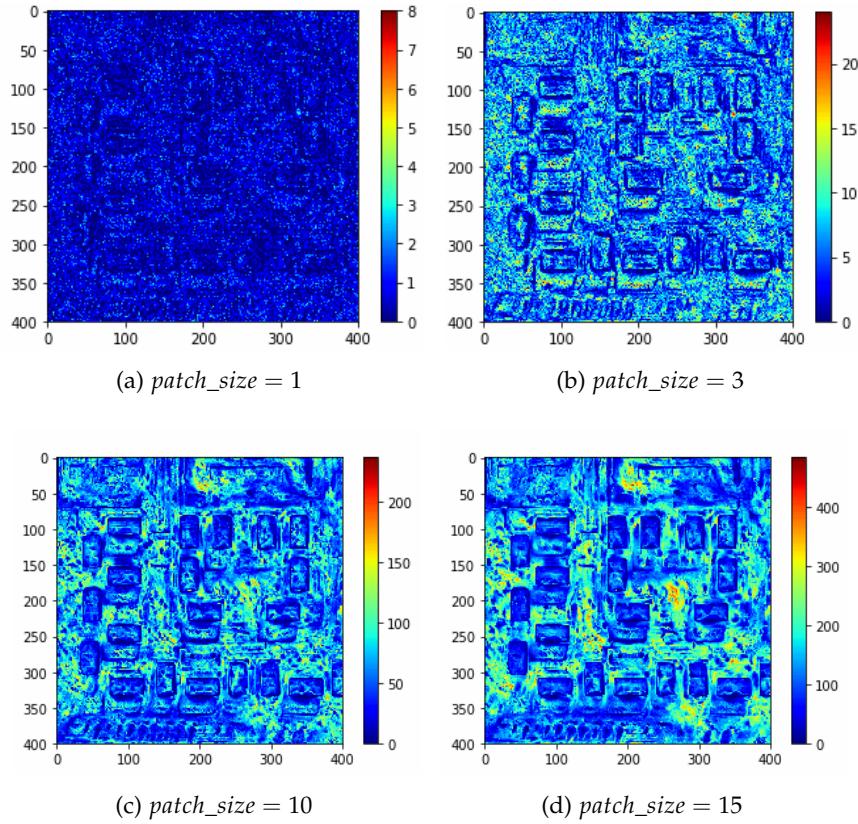


Figure 6: Cartes de coûts obtenues avec la version simple de l'algorithme, pour différentes tailles de patch.

des zones très uniformes et sans variations de hauteur suffisantes (comme au milieu d'un lac dans la figure 8d), l'algorithme est moins performant.

Toutes ces limitations mériteraient d'être étudiées plus en détail dans un travail futur afin de pouvoir proposer des améliorations et une généralisation de l'algorithme.

III Quelques extensions possibles

III.1 Lisser et débruiter la carte de hauteurs

Les cartes de hauteurs obtenues avec la version basique de l'algorithme ne sont pas tout à fait satisfaisantes. Les résultats sont trop fortement bruités. Nous pouvons noter que certains points n'ont pas de correspondance dans les deux images choisies. Il y des occlusions. La hauteur associée à ces points n'est donc pas pertinente. Pour les supprimer de la carte finale (et ainsi de pas générer de fausses informations), nous pouvons sélectionner tous les points dont le coût associé reste inférieur à un certain seuil. Nous obtenons ainsi une carte de hauteurs moins perturbée par des variations de hauteur non-pertinentes (Figure 9).

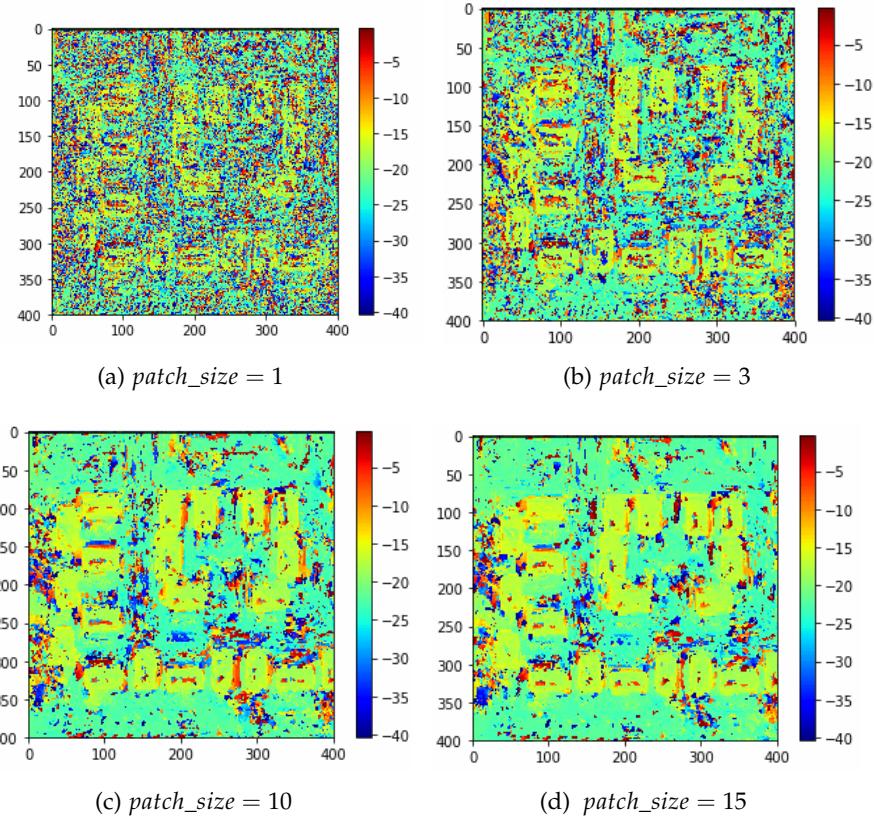
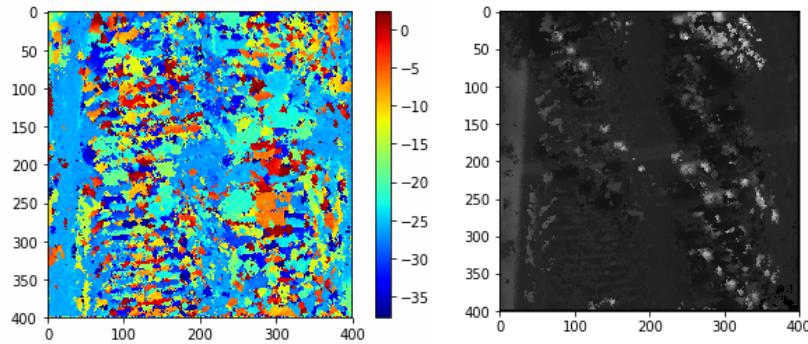


Figure 7: Carte de hauteurs obtenues avec la version simple de l’algorithme, pour différentes tailles de patch.

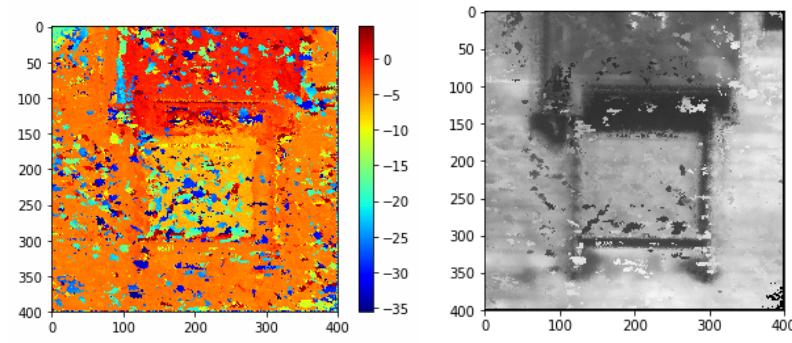
Le seuil est à choisir avec attention (Figure 10). Pour une valeur élevée, nous créons une carte plus fiable car il y a moins de pics isolés. Cependant, la carte générée contient aussi moins d’information. Elle devient parcimonieuse : il y a plus de zones vides. Il faut donc trouver un bon compromis. En pratique, nous avons modifié ce paramètre de sorte à obtenir la meilleure carte de hauteurs possible par expérimentations successives. La figure Fig. 11 montre une carte de hauteurs satisfaisante, obtenue avec une valeur de seuil égale à 150.

Nous avons aussi essayé d’appliquer d’autres approches de débruitage sur cette carte lissée. Tout d’abord nous avons pensé à convoluer la carte obtenue avec un noyau Gaussien de taille 3×3 . Cette méthode n’a pas permis d’améliorer les résultats et nous l’avons donc abandonnée.

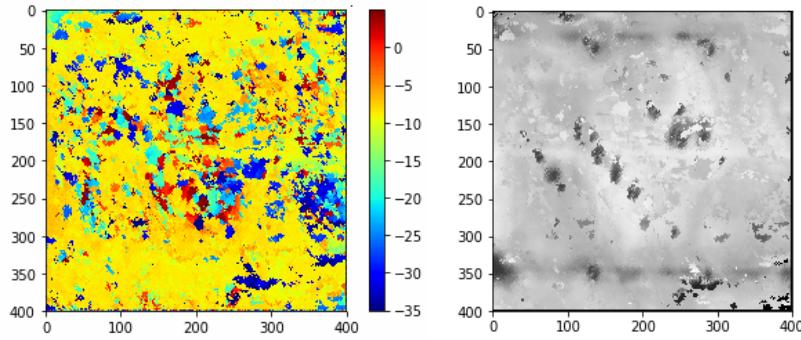
Nous nous sommes ensuite intéressés au filtrage de chatoiement (*speckle filtering*). Le bruit de chatoiement est caractéristique des images radars et provient des diffusions multiples des ondes. Il est généralement admis qu’il suit une loi de Rayleigh dont la fonction



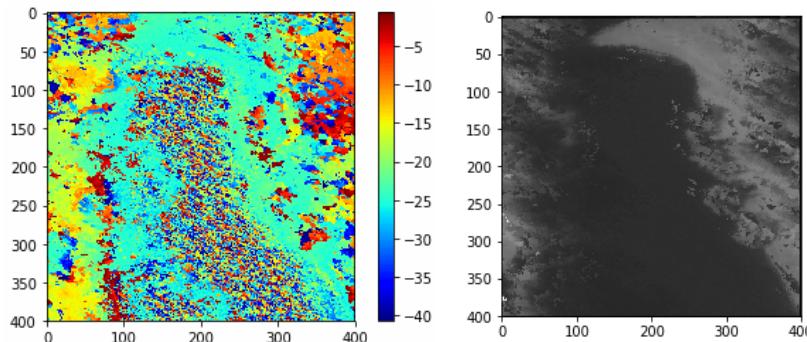
(a) Zone de parking.



(b) Bordure de toits.



(c) Zone de faible contraste au milieu d'un toit.



(d) Zone plane au milieu d'un lac.

Figure 8: Définition de différentes zones d'intérêt (Carte de hauteur à gauche et carte orthorectifiée avec les couleurs originales à droite).

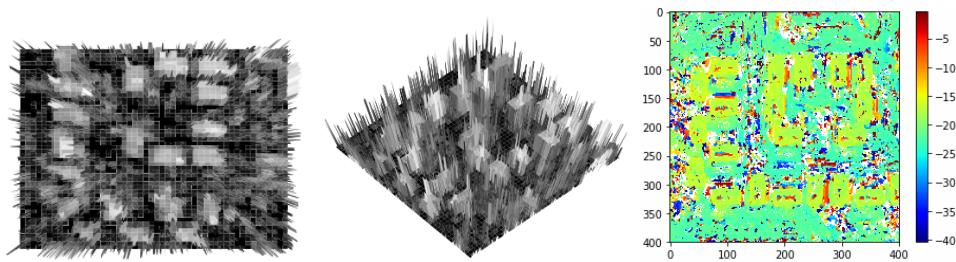


Figure 9: Suppression des points dont le coût dépasser un seuil fixé à 100

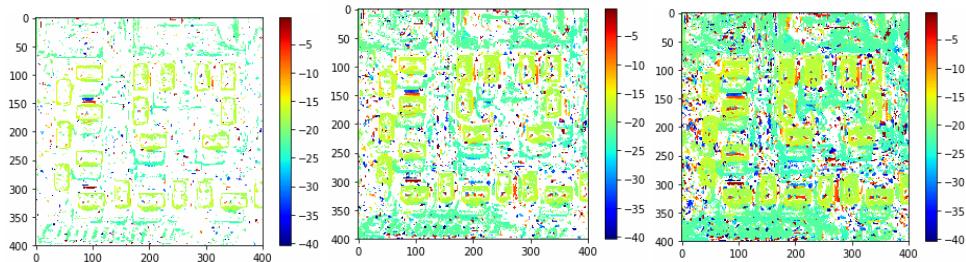


Figure 10: Suppression des points dont le coût dépasser un seuil fixé à 25 (gauche), 50 (milieu) ou 75 (droite)

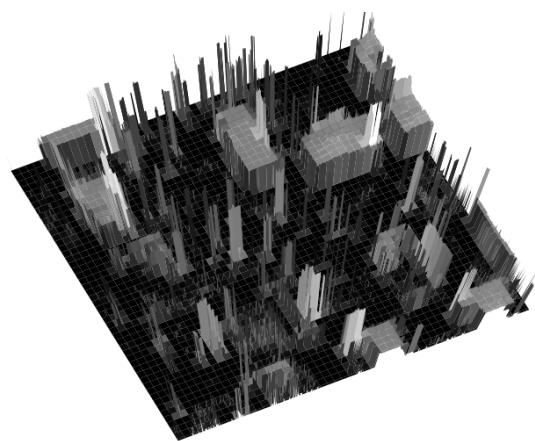


Figure 11: Visualisation 3D d'une carte filtrée avec seuil de coût égal à 150

de densité est donnée par :

$$f(x, \sigma) = \frac{x}{\sigma^2} \cdot e^{-\frac{x^2}{2\sigma^2}} \text{ pour } x \geq 0$$

Il s'agit d'un bruit multiplicatif : $y = x \cdot b$ avec y l'image bruitée, x l'image sans bruit et b le bruit. Pour appliquer un filtrage de chatoiement aux cartes de hauteurs, nous avons utilisé la fonction "specklefilter" qui possède un paramètre "area" qui influence fortement les résultats (Figure 12). Plus on choisit une grande valeur de paramètre, plus l'information conservée est éparsé. Là aussi, un bon compromis à trouver.

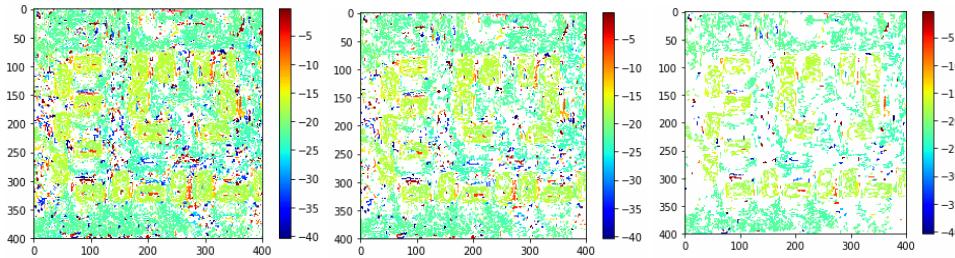


Figure 12: Filtrage de chatoiement pour une valeur du paramètre "area" égal à 3 (gauche), 5 (milieu) ou 10 (droite), sur un carte filtrée au seuil 100.

Trouver une méthode de paramétrage automatique pourrait être intéressant et apporter beaucoup à la méthode proposée. Cette recherche est laissée en travail futur.

III.2 Rectifier les images

Jusque là, nous avons utilisé des images non-rectifiées. Cela ne posait pas de problème car les images choisies avaient une géométrie suffisamment proche. Cela n'est pas toujours le cas. Pour généraliser notre méthode à toute paire d'images, nous avons rajouté une étape de rectification.

La rectification permet de comparer des patchs de même orientation dans les différentes images disponibles. Une transformation affine est appliquée aux images pour obtenir des lignes épi-polaires horizontales et parallèles (Figure 13).

Cette étape intervient au tout début de l'algorithme. Nous appliquons la transformation *Census* sur les images rectifiées. Pour la rectification, nous avons utilisé la fonction "rectify_aoi" du fichier "rectification.py". Nous obtenons ainsi deux images rectifiées et rognées sur la zone d'intérêt, deux matrices de projection $P1$ et $P2$ dans le repère original et deux matrices de rectification $S1$ et $S2$ qui permettent de passer de la géométrie originale à la géométrie rectifiée (Figure 14). Pour passer du monde réel 3D à l'image rectifiée, nous calculons donc simplement les produits matriciels $S1 \cdot P1$ et $S2 \cdot P2$.

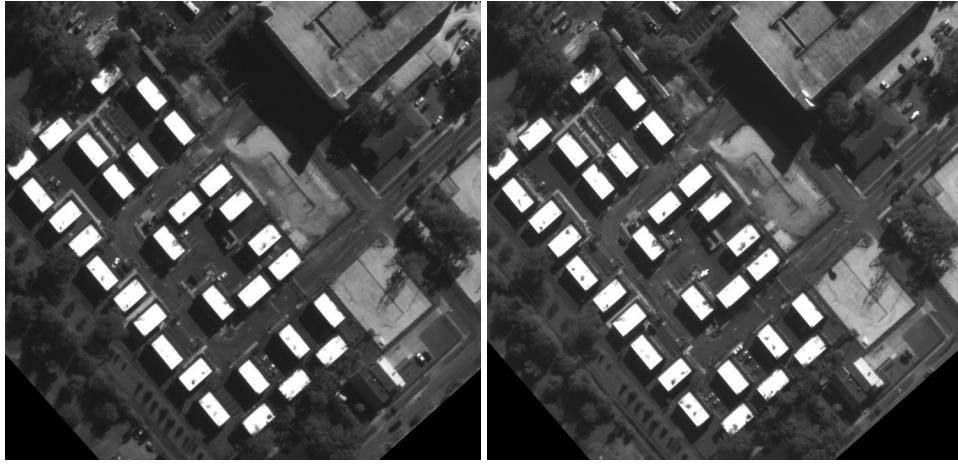


Figure 13: Un exemple de paire d'images rectifiées

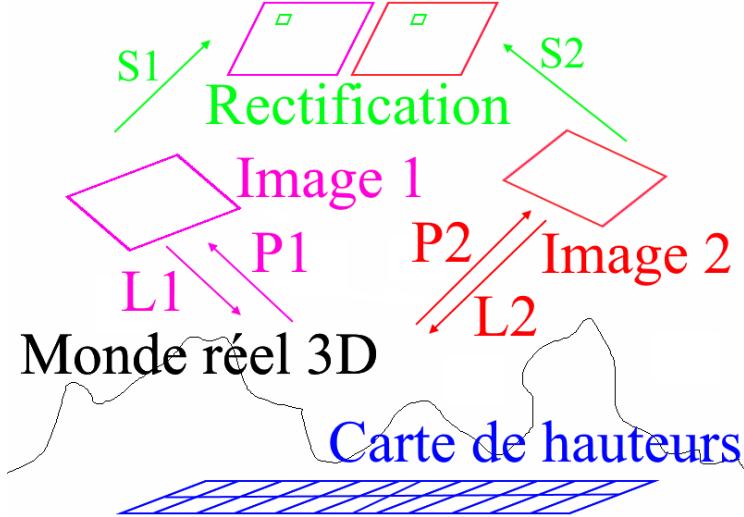


Figure 14: L'ajout de la rectification dans le modèle

Les cartes de coûts (Figure 15b) et de hauteurs (Figure 15) sont cohérentes. On peut maintenant appliquer l'algorithme à des images sans qu'elles se ressemblent initialement. L'algorithme a été testé sur différentes zones d'intérêts pour vérifier la pertinence des résultats (Figures 16 et 17). Nous constatons que les zones d'intérêt les plus difficiles (comme le parking) restent imparfaitement traitées.

III.3 Prendre en compte un plus grand nombre d'images

Sur certaines paires d'images, des occlusions, dues à l'orientation et à la position des caméras, empêchent *PatchMatch* de fonctionner correctement. Pour pallier ce problème, nous pouvons multiplier les points de vue et travailler avec un nombre arbitraire d'images.

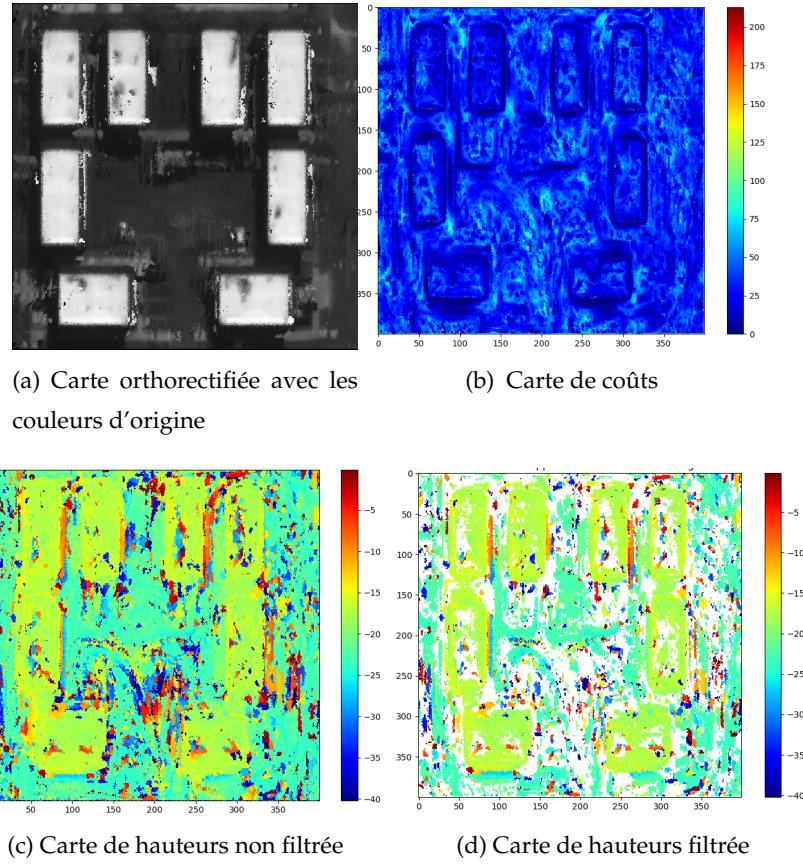


Figure 15: Exemple 1 - Résultat obtenu en utilisant des images rectifiées

Toutes les images disponibles sont regroupées paire par paire. Le choix des meilleures stratégies pour créer ces paires d'images est un sujet à explorer dans un travail futur. Nous avons choisi d'associer des images dont les dates d'acquisition étaient les plus proches possible. D'autres choix auraient pu être fait : même heure (pour avoir la même activité urbaine par exemple), même saison (pour les zones naturelles par exemple), même conditions météorologiques...

Nous appliquons ensuite notre algorithme sur chaque paire d'images. Nous sélectionnons les hauteurs associées aux plus faibles coûts de toutes les estimations obtenues (Figure 18). Plus de détails sont capturés, ce qui était l'objectif.

III.4 Définir la grille de hauteurs dans le système *easting/northing*

Il existe plusieurs conventions dans le système de coordonnées: *longitude/latitude* ou *easting/northing*. Nous proposons les deux options d'affichage sans que cela n'impacte les performances.

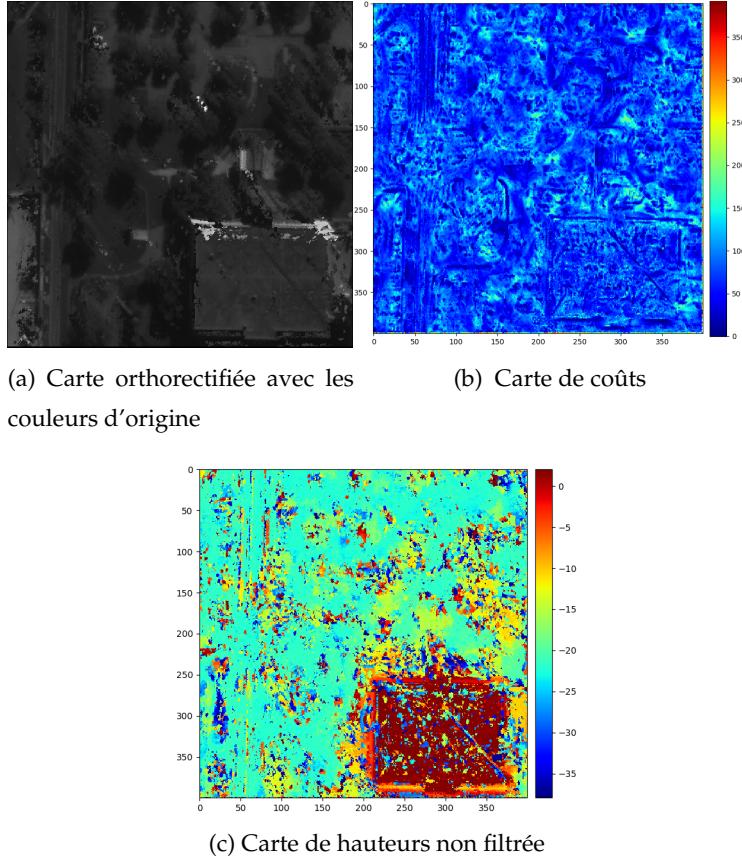


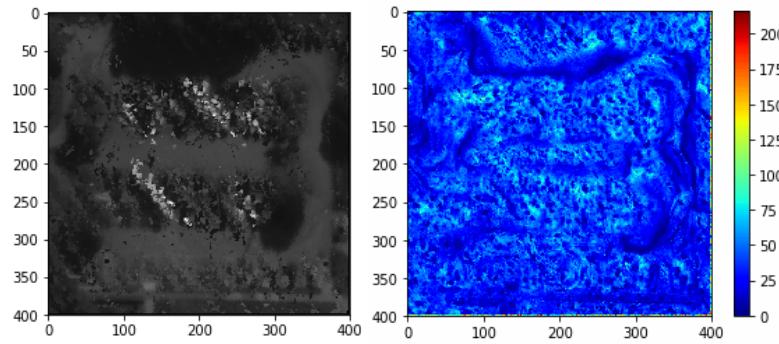
Figure 16: Exemple 2 - Résultat obtenu en utilisant des images rectifiées

IV Présentation des détails d'implémentation

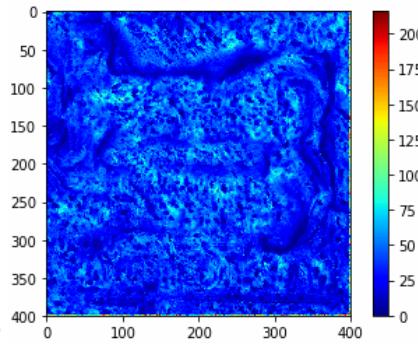
IV.1 Description des fonctions

La méthode proposée a été implémentée intégralement en *python*. Les fonctions principales sont disponibles, sous forme de pseudo-code, en annexes (section VI).

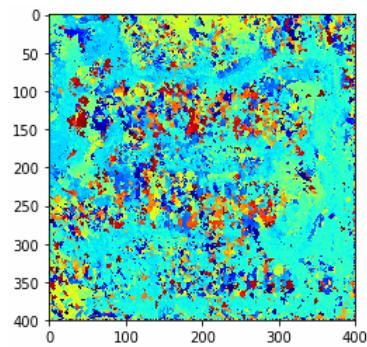
La fonction principale (Algorithme 3) prend en arguments deux images *brutes* (rectifiées ou non), une taille de grille $n \times m$, des bornes de hauteurs minimales et maximales spécifique à la zone étudiée (typiquement déterminées par l'utilisateur par exemple avec les données de la mission **SRTM Shuttle Radar Topography Mission** fournies par la *NASA* et la *NGA*) et une zone d'intérêt (*aoi*). Cette fonction calcule une grille de points, en *Eastings*, *Northings* ou en *Longitudes*, *Latitudes*, ciblée sur l'*aoi*, de taille $n \times m$ (donnée en argument). Les images d'entrée sont alors rectifiées. A l'issue de la rectification, nous récupérons les images rectifiées ainsi que les matrices d'homographie servant à passer de l'image non-rectifiée à l'image rectifiée et les matrices de l'approximation affine de la projection RPC. On applique ensuite la transformation *Census* sur chacune des images.



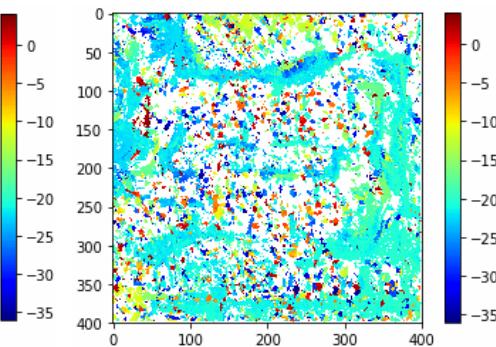
(a) Carte orthorectifiée avec les couleurs d'origine



(b) Carte de coûts



(c) Carte de hauteurs non filtrée



(d) Carte de hauteurs filtrée (seuil = 37)

Figure 17: Exemple 3 - Résultat obtenu en utilisant des images rectifiées sur une zone de parking

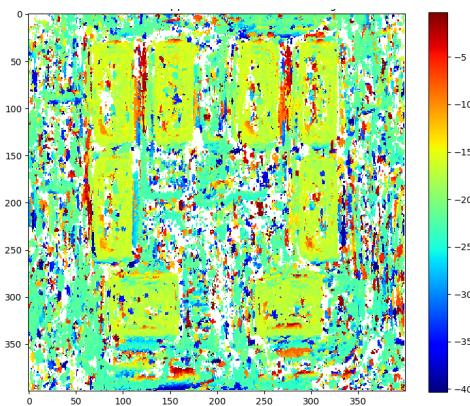


Figure 18: L'estimation de la carte de hauteurs, filtrée, obtenue à partir de six paires d'images rectifiées

On initialise alors une carte de hauteurs. Pour l'initialisation aléatoire, nous avons testé une distribution uniforme et une distribution Gaussienne centrée sur la hauteur de référence donnée par *SRTM*, sans que cela n'impacte significativement les résultats. On lui associe une carte de coûts (carte en échelle de gris dont la valeur de chaque pixel est le coût associée à la hauteur de ce pixel dans la carte de hauteurs).

La mise à jour de la carte de hauteurs (et donc de la carte de coûts également) se fait par alternances itératives de passe de l'algorithme *PatchMatch* dans le sens *direct* et dans le sens *indirect* comme détaillé précédemment (Figure 1). L'algorithme s'arrête lorsque le nombre de points de la grille mis à jour est suffisamment faible (en pratique, on peut prendre un pourcentage de la taille de la grille, entre 0.1% et 1% par exemple). Un nombre maximal d'itérations peut aussi être fixé.

A la différence de l'algorithme *PatchMatch* initial (comme décrit dans Barnes et al. [2009]), nous procédons à la mise à jour par propagation des voisins et à la recherche aléatoire en une seule étape. En effet, l'article original de 2009 alterne une recherche aléatoire et une propagation de l'information des voisins. Cette approche facilite la compréhension de la méthode, mais elle implique de parcourir deux fois la grille pour mettre à jour l'information des cartes. Nous avons jugé plus judicieux de sélectionner la mise à jour directement comme la meilleure candidate (c'est-à-dire celle qui engendre le coût le plus faible) parmi celles des voisins et une valeur aléatoire. Cela nous exempte d'un parcourt de grille sur deux et divise ainsi par un facteur 2 notre temps de calcul. La complexité de notre algorithme est détaillée dans la Section IV.2. Partant de cette constatation, nous avons aussi essayé de faire plusieurs recherches aléatoires en même temps, ce qui permettait encore d'améliorer la vitesse de convergence de la méthode, surtout lorsque la recherche aléatoire devient prédominante sur la propagation et que la gamme de hauteur de recherche est large (i.e. $h_{max} - h_{min}$ grand).

La fonction *PatchMatch* (Algorithmes 1 et 2) effectue les mises à jour. Elle prend en arguments les deux images rectifiées, les matrices de projection affine (comme les images d'entrées sont rectifiées, ce sont les produits des matrices de rectification et de projection dans les images non-rectifiées respectives), les cartes de hauteurs et de coûts à mettre à jour, une carte de hauteurs aléatoire modifiée à chaque appel de *PatchMatch*, le maillage *Eastings/Northings* et un sens de propagation.

Selon le sens de propagation, on parcourt la grille de haut en bas et de gauche à droite ou inversement. A chaque point de la grille, on récupère les coordonnées que l'on convertit dans le système *Longitude/Latitude*. On compare le coût actuel (associé à la hauteur actuelle) avec celui associé aux hauteurs des voisins précédents (comme illustré en Figure 1), et une hauteur aléatoire (de la carte de hauteurs aléatoire donnée en argument). Si une de ces hauteurs contribue à diminuer le coût actuel, la carte de coûts est mise à jour en ce point de

la grille en la changeant par la valeur de ce coût plus faible. La carte de hauteur est aussi réactualisée par la hauteur qui a occasionné le coût minimal parmi ceux testés.

Au fil des itérations, ces deux cartes sont progressivement révisées et améliorées. A chaque fois qu'une mise à jour s'est effectuée, nous incrémentons un compteur. La fonction retourne le nouveaux couple cartes de hauteurs / carte de coûts ainsi que la valeur du compteur. Compter ainsi le nombre de mises à jour nous fournit un critère d'arrêt des itérations de la fonction *PatchMatch* : si le nombre de points modifiés n'est plus significatifs, nous arrêtons les itérations.

Toutes ces fonctions reposent sur le calcul du coût d'une hauteur donnée en un point de la grille. Nous avons utilisé la distance de *Hamming* entre deux transformée de *Census* des images.

IV.2 Complexité de l'algorithme

Le cœur de l'algorithme est composée de la fonction *PatchMatch* pour laquelle nous parcourons entièrement la grille ($O(nm)$). Les autres fonctions intermédiaires ne nécessite pas de parcourir entièrement la grille est son en temps de calcul négligeable devant $O(nm)$.

Toutefois la fonction de projection (issue des métadonnées RPC des images enregistrées au format *tiff*) avait un temps d'exécution assez long (50 secondes d'exécution en moyenne sur 5 millions d'itérations _ soit $10\mu\text{s}$ par appel). Nos images représentant une zone terrestre suffisamment petite, nous avons utilisé l'approximation de *Taylor* au premier ordre de cette fonction au point central de notre grille. Pour des grilles jusqu'à trois fois l'ordre de grandeur de nos images, nous obtenons sur l'ensemble de la grille une erreur engendrée d'au plus de l'ordre de 10^{-3} par l'approximation affine et la fonction de projection. De plus, le développement limité étant fait au centre de la grille, l'erreur est typiquement beaucoup plus faible dans la grande partie centrale de la grille $\approx 10^{-5}$.

Nous ne perdons donc pas d'informations *capitales* à utiliser cette projection affine et l'intérêt est qu'elle est beaucoup plus facile à calculer : elle est calculée *assez* rapidement une bonne fois pour toute et chaque appel est en moyenne 15 fois plus rapide (de l'ordre de 600ns sur une moyenne de 5 millions d'itérations).

L'initialisation des itérations est assez longue. Le calcul d'une carte de hauteur et sa carte de coûts associés sont très rapide, mais la transformation *Census* est très longue. D'autant plus que la taille des patchs est grandes (si les patchs sont de taille α^2 , il faut parcourir (en négligeant les bords) $\alpha^2 \times ntimesm$ et appliquer à chaque fois la transformation *Census*. Mais ce calcul est effectué une seule fois, une fois et n'a pas lieu d'être recalculé à moins que l'on change la taille des patchs.

Nous avons essayé d'optimiser la plupart des implémentations (recherche du minimum, *etc.*). Nos fonctions sont aussi optimisés pour être évaluée de manière performante avec la

bibliothèque **numba**, ce qui nous a fait gagné un facteur de temps considérable. Pour nos tests, nous avons coupé nos images pour avoir des tailles de l'ordre de 500×500 . L'appel de la fonction *PatchMatch* prend en moyenne un quart de seconde (chaque couple de passage sens *direct* / *indirect* prend donc une demi seconde). Compte tenu de la visualisation de la carte de coûts au fur et à mesure des itérations, nous pouvons conclure que le nombre de passage dans un sens puis l'autre de propagation doit être de l'ordre de la dizaine. En pratique, le nombre de pixels mis à jour décroît très vite au cours des premières itérations, mais la décroissance est plus faible au delà d'une dizaine d'itérations. Le comportement de décroissance du nombre de hauteurs améliorées à chaque itérations de *PatchMatch* est illustrer en figure Fig. 20, obtenu sur une image de taille 500×500 . En réalité, nous pensons que la propagation des voisins est très rapide et qu'après cette dizaine d'itération, les points mis à jour sont des points qui n'ont pas la possibilité d'être améliorés par leur voisins. La propagation est donc très rapidement fini et seul la recherche aléatoire peut améliorer, ce qui explique cette perte de vitesse de décroissance du nombre de points mis à jour. La figure Fig. ?? (obtenue sur une image de taille 400×400) témoigne de cette dernière constatation que nous avons faite. En outre, après cette dizaine d'itérations, le nombre de points encore modifiés est au plus de l'ordre de 1%, voire plus bas encore. Sur l'exemple de la figure Fig. ??, on voit clairement qu'il est inutile de faire plus de 10 itérations. Arrivé à 10 itérations, moins de 0.5% des hauteurs sont améliorées et continuer plus loin (cette figure illustre une expérience où nous avons continué jusque 30 itérations) n'améliore pas grandement la carte de hauteur. En effet, au delà de 10 itérations, le nombre de hauteurs mises à jour reste globalement décroissant mais atteint presque un plateau de l'ordre de 0.5% de la taille de la grille. D'ailleurs, visuellement, la carte de coûts ou de hauteur ne nous paraît plus évoluer à partir de ce moment. C'est pourquoi dans la pratique, nous arrêtons les itérations dès que le nombre de points modifiés n'excédait pas une seuil proche d'un centième de la taille de la grille.

Finalement, en conclusion, il faut de l'ordre de la minute pour l'exécution totale de notre algorithme pour 6 ou 7 paires d'images.

V Discussion

Pour conclure, nous avons proposé une nouvelle méthode d'estimation de cartes de hauteurs à partir d'images satellitaires. Notre approche donne des résultats prometteurs dans de nombreuses situations et a l'avantage d'être très rapide.

Cependant, des limitations nuisent encore aux performances de notre algorithme. Notre méthode est moins efficace sur des zones avec peu de détails ou de contraste ou sur des endroits contenant beaucoup d'objets mobiles. Pour ne pas créer de fausse information, nous avons appliqué des techniques de filtrage mais cela donnent des cartes où nous n'avons pas d'information sur tous les points. Les données sont éparses.

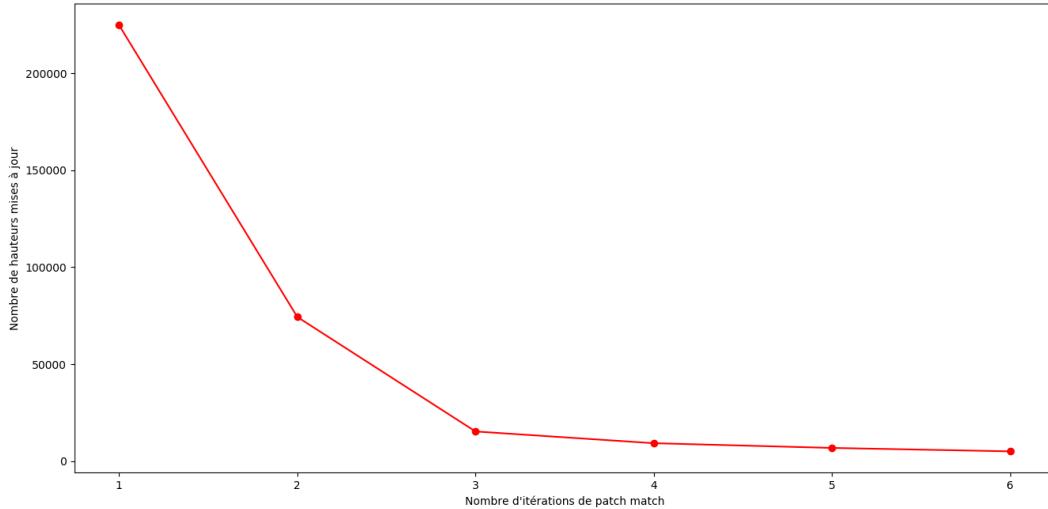


Figure 19: Décroissance *typique* du nombre de hauteurs modifiées au cours des itérations

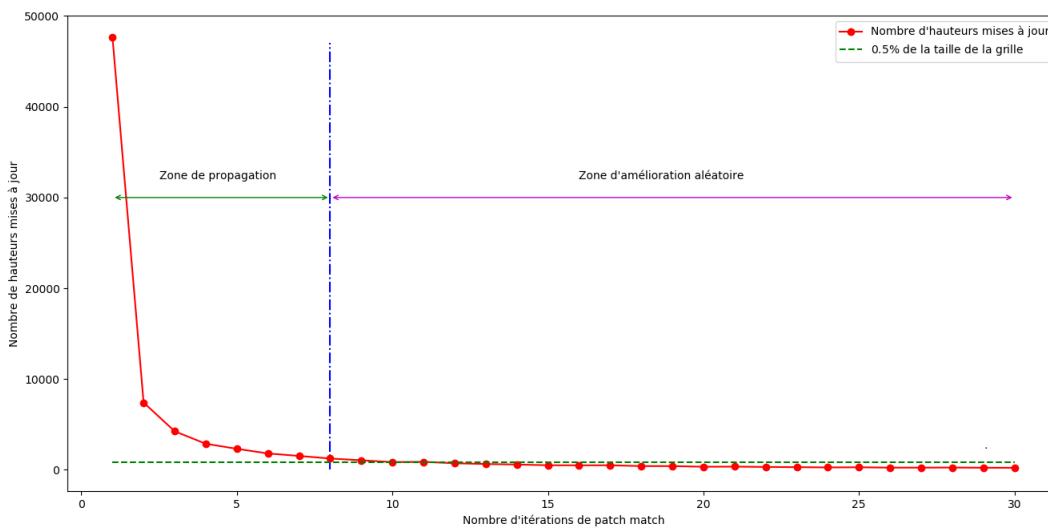


Figure 20: Illustration de la vitesse de propagation et de l'inutilité de poursuivre trop d'itérations.

Des améliorations pourraient être proposées dans un travail futur. Nous pourrions par exemple imaginer adapter la méthode de consistance droite/gauche pour mieux repérer les hauteurs aberrantes dues aux occlusions. Enfin, des cartes de vérité-terrains devraient être collectées afin de pouvoir évaluer quantitativement les erreurs commises dans nos estimations.

VI Références

- Hae-Gon Jeon, Jaesik Park, Gyeongmin Choe, Jinsun Park, Yunsu Bok, Yu-Wing Tai, and In So Kweon. Accurate depth map estimation from a lenslet light field camera. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.
- A. Qayyum, A. S. Malik, M. N. B. M. Saad, F. Abdullah, and M. Iqbal. Disparity map estimation based on optimization algorithms using satellite stereo imagery. In *2015 IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, pages 127–132, Oct 2015. doi: 10.1109/ICSIPA.2015.7412176.
- K. Wang, C. Stutts, E. Dunn, and J. Frahm. Efficient joint stereo estimation and land usage classification for multiview satellite data. In *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1–9, March 2016. doi: 10.1109/WACV.2016.7477657.
- Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. Patchmatch: A randomized correspondence algorithm for structural image editing. In *ACM Transactions on Graphics (ToG)*, volume 28, page 24. ACM, 2009.
- Frederic Besse, Carsten Rother, Andrew Fitzgibbon, and Jan Kautz. Pmbp: Patchmatch belief propagation for correspondence field estimation. *International Journal of Computer Vision*, 110(1):2–13, 2014.
- Philipp Heise, Sebastian Klose, Brian Jensen, and Alois Knoll. Pm-huber: Patchmatch with huber regularization for stereo matching. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2013.
- Enliang Zheng, Enrique Dunn, Vladimir Jojic, and Jan-Michael Frahm. Patchmatch based joint view selection and depthmap estimation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.
- Michael Bleyer, Christoph Rhemann, and Carsten Rother. Patchmatch stereo - stereo matching with slanted support windows. In *Bmvc*, volume 11, pages 1–11, January 2011. URL <https://www.microsoft.com/en-us/research/publication/patchmatch-stereo-stereo-matching-with-slanted-support-windows/>.
- Sean Ryan Fanello, Julien Valentin, Adarsh Kowdle, Christoph Rhemann, Vladimir Tankovich, Carlo Ciliberto, Philip Davidson, and Shahram Izadi. Low compute and fully parallel computer vision with hashmatch. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 3894–3903. IEEE, 2017.
- Fayao Liu, Chunhua Shen, and Guosheng Lin. Deep convolutional neural fields for depth estimation from a single image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5162–5170, 2015.

Clément Godard, Oisin Mac Aodha, and Gabriel J Brostow. Unsupervised monocular depth estimation with left-right consistency. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 270–279, 2017.

Akbar Darabi and Xavier Maldaque. Neural network based defect detection and depth estimation in tnde. *Ndt & E International*, 35(3):165–175, 2002.

Appendices

Algorithme 1 : Mise à jour par *PatchMatch* dans le sens direct

Entrées : $im_1, RPC_1, im_2, RPC_2, carte_hauteurs, carte_couts, carte_haut_alea,$
 $maillage(Eastings, Northings)$

Sorties : $compteur, carte_hauteurs, carte_couts$

for $i = 1 \dots n$ **do**

$east \leftarrow Eastings[i]$

for $j = 1 \dots m$ **do**

$north \leftarrow Northings[j]$

// Passage en coordonnées longitude / latitude

$(lat, lon) \leftarrow \text{utm.to_latlon}(east, north)$

// Recherche d'une meilleure hauteur

$haut_alea = carte_haut_alea[i, j]$

$haut_voisin_gauche = carte_haut[i - 1, j]$

$haut_voisin_dessus = carte_haut[i, j - 1]$

// Calcul des coûts associés

$cout_alea \leftarrow cout(im_1, RPC_1, im_2, RPC_2, haut_alea, lon, lat)$

$cout_voisin_gauche \leftarrow cout(im_1, RPC_1, im_2, RPC_2, haut_voisin_gauche, lon, lat)$

$cout_voisin_dessus \leftarrow cout(im_1, RPC_1, im_2, RPC_2, haut_voisin_dessus, lon, lat)$

// Recherche de la hauteur qui minimise le coût

$cout_min \leftarrow \min\{cout_alea, cout_voisin_gauche, cout_voisin_dessus\}$

$haut_min \leftarrow \{haut_alea, haut_voisin_gauche, haut_voisin_dessus\}$

// Mise à jour des cartes (hauteurs et coûts) si $cout_min$ est inférieur au coût actuel

if $cout_min < carte_couts[i, j]$ **then**

$carte_couts \leftarrow cout_min$

$carte_hauteurs \leftarrow haut_min$

end

end

end

Algorithme 2 : Mise à jour par *PatchMatch* dans le sens indirect

Entrées : $im_1, RPC_1, im_2, RPC_2, carte_hauteurs, carte_couts, carte_haut_alea,$
 $maillage(Eastings, Northings)$

Sorties : $compteur, carte_hauteurs, carte_couts$

for $i = n - 1, n - 2 \dots 1$ **do**

$east \leftarrow Eastings[i]$

for $j = m - 1, m - 2 \dots 1$ **do**

$north \leftarrow Northings[j]$

//Passage en coordonnées longitude / latitude

$(lat, lon) \leftarrow \text{utm.to_latlon}(east, north)$

//Recherche d'une meilleure hauteur

$haut_alea = carte_haut_alea[i, j]$

$haut_voisin_droite = carte_haut[i + 1, j]$

$haut_voisin_dessous = carte_haut[i, j + 1]$

// Calcul des coûts associés

$cout_alea \leftarrow cout(im_1, RPC_1, im_2, RPC_2, haut_alea, lon, lat)$

$cout_voisin_droite \leftarrow cout(im_1, RPC_1, im_2, RPC_2, haut_voisin_droite, lon, lat)$

$cout_voisin_dessous \leftarrow cout(im_1, RPC_1, im_2, RPC_2, haut_voisin_dessous, lon, lat)$

// Recherche de la hauteur qui minimise le coût

$cout_min \leftarrow \min\{cout_alea, cout_voisin_droite, cout_voisin_dessous\}$

$haut_min \leftarrow \{haut_alea, haut_voisin_droite, haut_voisin_dessous\}$

// Mise à jour des cartes (hauteurs et coûts) si $cout_min$ est inférieur au coût actuel

if $cout_min < carte_couts[i, j]$ **then**

$carte_couts \leftarrow cout_min$

$carte_hauteurs \leftarrow haut_min$

end

end

end

Algorithme 3 : Estimation d'une carte de hauteur avec *PatchMatch*

Entrées : $image_1, image_2, nb_iter, seuil_arret_PM, h_{min}, h_{max}, n, m$

Sorties : $carte_de_hauteurs$

// Rectification

$(im_1, im_2, S_1, S_2) \leftarrow rectification(image_1, image_2)$

// Fonctions RPC

$(P_1, P_2) \leftarrow projection_affine_rpc(image_1, image_2)$

$RPC_1 = S_1.P_1$

$RPC_2 = S_2.P_2$

// Définition de la grille

$Eastings, Northings \leftarrow meshgrid(n, m)$

// Census transform

$im_1 \leftarrow Census_Transform(im_1)$

$im_2 \leftarrow Census_Transform(im_2)$

// Initialisation

$carte_hauteurs \leftarrow random(n, m)$

$carte_couts \leftarrow \text{calcule_carte_couts}(carte_hauteurs)$

// Pour pouvoir rentrer dans la boucle...

$compteur \leftarrow seuil_arret_PM + 1$

$iter \leftarrow 0$

while $iter < nb_iter$ AND $compteur > seuil_compteur_max$ **do**

//Pour la recherche aléatoire...

$carte_hauteurs_alea \leftarrow random(n, m)$

//Application de *PatchMatch* dans le sens direct puis indirect

$(carte_hauteurs, carte_couts, compteur) \leftarrow \text{PM_direct}(im_1, RPC_1, im_2, RPC_2,$
 $carte_hauteurs, carte_couts, carte_hauteurs_alea, Eastings, Northings)$

$(carte_hauteurs, carte_couts, compteur) \leftarrow \text{PM_indirect}(im_1, RPC_1, im_2, RPC_2,$
 $carte_hauteurs, carte_couts, carte_hauteurss_alea, Eastings, Northings)$

$iter \leftarrow iter + 1$

end
