

ROB317 - TD1

BRAMBILLA Davide Luigi - GOMES DA SILVA Rafael

26 September 2019

Les codes se trouvent dans notre dépôt *git* accessible au lien:
<https://gitlab.data-ensta.fr/brambilla/rob317.git>

1 Format d'images et Convolutions

Q1 - Comparaison entre les deux méthodes dans *Convolution.py*

En exécutant le code *Convolution.py* la première différence qui l'on note est donné par le temps d'exécution des deux méthodes: le premier prends environ *150 milliseconds* quand le deuxième prends moins de *1 millisecond*.

La raison est le fait que dans la première on est en train de faire deux boucles en effectuant des calculs pixel par pixel. Par contre, pour le deuxième, la fonction *filter2D* fait partie de la librairie *OpenCV* et est optimisé pour faire des calcul de convolution sur l'image.

Dans notre cas, on obtient un temps d'exécution plus petit car le *noyau* qui l'on utilise est petit par rapport à la taille de l'image.

En fait la fonction *filter2D* est implémenté de façon de pouvoir exploiter la propriété d'associativité de la convolution que dit que si on fait la convolution entre une image I et un noyau $(K+L)$ composé de deux *sous-noyaux* K et L a le même résultat que faire la somme entre la convolution entre l'image et le première *sous-noyau* et la convolution entre l'image et le deuxième *sous-noyau*:

$$I * (K + L) = I * K + I * L$$

Donc en exploitant cette propriété c'est possible de avoir un avantage i on prend, par exemple, une image de taille (H, W) et un *nøyau* de taille (X, Y) , avec l'approche d'itération sur les matrices on aura environ un nombre d'opérations à faire de l'ordre de $H \cdot W \cdot X \cdot Y$ mais en utilisant *filter2D* l'on en aura $H \cdot W \cdot (X + Y)$ car le *nøyau* sera divisé en deux *sous-noyaux* et l'opération sera divisé en deux étapes: le premier qui requis un nombre de opérations de l'ordre de $H \cdot W \cdot X$ et le deuxième qu'en requis $H \cdot W \cdot Y$.

Donc cette amélioration introduit par la fonction *filter2D* est de l'ordre de $\frac{X \cdot Y}{X + Y}$.

Dans notre cas le noyau de convolution est: $\begin{bmatrix} -1 \\ -1 & 5 & -1 \\ -1 \end{bmatrix}$ et peut être décomposé en trois *sous-noyaux*:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 5 & -1 \\ -1 \end{bmatrix} = [1] - \begin{bmatrix} 1 & 1 & 1 \\ 1 & -4 & 1 \\ 1 \end{bmatrix} = [1] - \left(\begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} + [1 \quad -2 \quad 1] \right)$$

de façon de exploiter, au moment de la convolution avec l'image, les avantages introduites par la propriété d'associativité.

Si l'on regarde le résultat, nous avons que les deux images obtenues avec les deux méthodes sont très similaires mais ils ne sont pas identiques(en faisant la différence entre les deux images nous n'obtenons pas une image totalement noir): cela est du au fait que on va créer des différents artefacts dans les deux images.

Les fonctions OpenCV et MatPlotLib-pyplot Les fonctions de la librairie *OpenCV* qui sont utilisés dans cette première partie sont:

- *cv2.imread(img)*: qui sert à lire l'image à partir du database.
- *cv2.getTickCount()*: qui sert pour avoir un information sur l'instant de temps actuel en mesurant les *ticks* actuels.
- *cv2.copyMakeBorder(img, 0, 0, 0, 0, cv2.BORDER_REPLICATE)*: qui permet de ajouter des contour de *padding* à notre image. Avec *BORDER_REPLICATE* on impose que la ligne ou la colonne au border est copié vers le contour ajouté.
- *cv2.getTickFrequency()*: qui divise la différence de temps entre les valeurs acquis avec *getTickCount* et il nous fait obtenir le temps mesuré en seconds.
- *cv2.filter2D(img, -1, kernel)*: qui effectue la corrélation entre l'image et le noyau de façon optimisé. -1 signifie que l'image en sortie aura la même dimension de la source.

Pour ce que concerne les fonctions de la librairie *MatPlotLib-pyplot* qui sont utilisées sont:

- `plt.subplot()`: qui sert pour pouvoir insérer dans un même plot plusieurs images.
- `plt.imshow(img3,cmap = 'gray',vmin = 0.0,vmax = 255.0)`: qui nous permet de faire des plots sur une échelle de gris avec des valeurs comprises entre 0 et 255.
- `plt.title('title')`: qui permet d'ajouter un titre à la seule image.
- `plt.show()`: qui permet de afficher à l'écran le plot préparé.

Q2 - Expliquer le rehaussement de contraste introduit par le noyau

Le noyau dont on dispose peut être décomposé comme l'on a vu dans la question Q1.

Dans le premier passage nous pouvons observer que le noyau est décomposé par la différence entre un noyau unitaire [1] et le noyau du *Laplacien* en 4-connexité $\begin{bmatrix} 1 \\ 1 & -4 & 1 \end{bmatrix}$.

En particulier le *Laplacien* est la somme des dérivés au deuxième ordre dans la direction verticale et dans la direction horizontale:

$$\Delta f = \frac{\delta^2 f}{\delta x^2} + \frac{\delta^2 f}{\delta y^2}$$

Quand nous allons appliquer ce noyau à l'image d'origine on aura comme résultat la convolution de l'image même avec ce noyau qui peut être vu, pour la propriété d'associativité de la convolution, comme la différence entre l'image même (car elle sera le résultat de la convolution avec un noyau unitaire) et le laplacien de l'image (le résultat de la convolution entre l'image et le *Laplacien* en 4 connexité).

Étant donné que le Laplacien met en évidence les zones de l'image où il y a un fort *gradient*, comme par exemple les contours, il est possible de remarquer que faire la différence entre l'image et son laplacien donne un rehaussement de contraste sur l'image d'origine.

Cela est mieux expliqué dans les graphiques suivants:

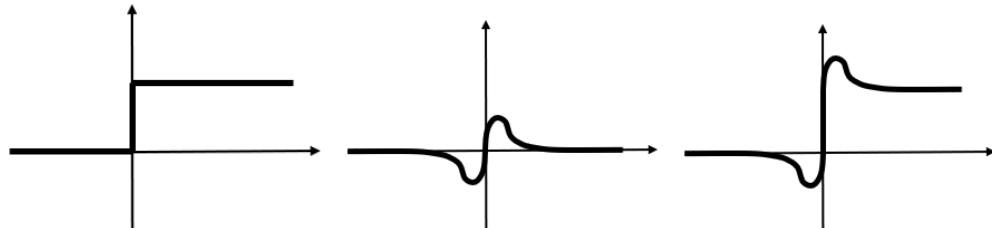


Figure 1: Échelon sur l'image, Laplacien de l'échelon, Différence entre l'échelon et son Laplacien

Dans le graphique de gauche nous avons reporté un *échelon* sur l'image qui représente un passage entre une zone sombre et une zone claire, dans lequel du milieu nous avons l'effet du gradient sur l'image qui s'occupe de détecter les points de contour et dans le graphique de droite nous avons le résultat de la différence des deux graphiques précédents: sur ce dernier nous pouvons voir comme l'image maintient ses propriétés mais qui présente un plus grand contraste car ses contours sont mieux mis en évidence.

Le résultat de cette opération est reporté dans les images suivantes.



Figure 2: Image d'origine

Q3 - La convolution qui approxime la dérivée partielle et le module du gradient

La détection des bords est un élément important du traitement des images. Pour ce faire, il est possible d'utiliser des filtres permettant d'affiner les transitions entre différentes régions. Cependant, il est important d'effectuer un bon pré-traitement de l'image avec un lissage gaussien avant d'appliquer ces types de filtres car ils accentuent également le bruit dans l'image.



Figure 3: Laplacien de l'image



Figure 4: Différence entre (2) et (3): **Rehaussement de Contraste**

Une façon de déterminer les bords dans une région d'une image consiste à utiliser des dérivés d'une image, car ils fournissent des opérateurs décrivant les bords en convolutionnant l'image avec un noyau spécifique. Ci-dessous, nous allons montrer les résultats de deux types de filtres utilisés pour effectuer la détection de bord d'une image à l'aide de dérivées partielles.

Dérivée partielle I_x

Pour la mise en œuvre de la détection de bord de convolution utilisant l'approximation de $I_x = \frac{\partial I}{\partial x}$, le noyau de convolution $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ a été utilisé comme référence, comme indiqué à la page 51 du cours d'introduction.

Le Code 1 présente la mise en œuvre de la convolution de l'image avec l'approximation de I_x en utilisant la méthode directe, alors que le Code 2 montre la mise en œuvre de la convolution de l'image avec l'approximation de I_x en utilisant la méthode du filtre 2D disponible en OpenCV. Le résultat obtenu pour chacune de ces méthodes est montré dans l'image 5.

Code 1: Détection de bords par convolution avec I_x en utilisant la méthode direct

```
val = 0*img[y, x] - img[y, x-1] + img[y, x+1]
img2[y,x] = min(max(val,0),255)
```

Code 2: Détection de bords par convolution avec I_x en utilisant la méthode filter2D

```
kernel = np.array([[0, 0, 0], [-1, 0, 1], [0, 0, 0]])
img3 = cv2.filter2D(img,-1,kernel)
```

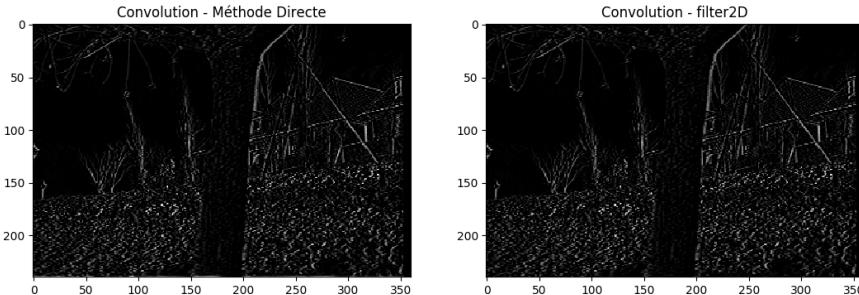


Figure 5: Résultats obtenus pour l'approximation de I_x en utilisant les deux méthodes proposées

Gradient

Pour la mise en œuvre de la détection de bord de convolution utilisant $\|\nabla I\|$, le noyau $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ pour I_x , et le noyau $\begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$ pour I_y ont été utilisés comme indiqué à la page 51 du cours d'introduction.

Le Code 3 présente la mise en œuvre de la convolution de l'image avec $\|\nabla I\|$ en utilisant la méthode directe, alors que le code le Code 4 montre la mise en œuvre de la convolution de l'image avec $\|\nabla I\|$ en utilisant la méthode du filtre 2D disponible en OpenCV. Le résultat obtenu pour chacune de ces méthodes est montré dans Image 6.

Code 3: Détection de bords par convolution avec $\|\nabla I\|$ en utilisant la méthode direct

```
val = 0*img[y, x] - img[y, x-1] + img[y, x+1]
img2[y,x] = min(max(val,0),255)
```

Code 4: Détection de bords par convolution avec $\|\nabla I\|$ en utilisant la méthode filter2D

```
kernel = np.array([[0, 0, 0], [-1, 0, 1], [0, 0, 0]])
img3 = cv2.filter2D(img,-1,kernel)
```

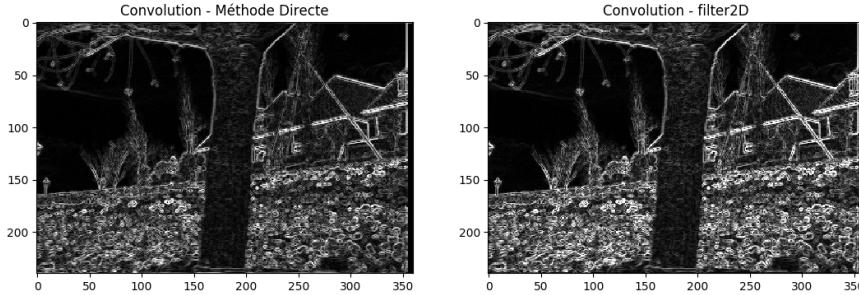


Figure 6: Résultats obtenus pour l'approximation de $\|\nabla I\|$ en utilisant les deux méthodes proposées

Conclusions

Dérivée partielle

En regardant les images de la Figure 5, on peut voir que les deux méthodes implémentées donnent le même résultat pour la détection de bord. Il est à noter que les arêtes présentes dans la direction de x sont mises en évidence comme prévu. Ainsi, on peut dire que, pour les filtres utilisant des dérivées du premier ordre, ils soulignent les contours présents dans une image, mais avec une magnitude inférieure à celle des filtres du second ordre.

Gradient

D'autre part, en regardant les images de la Figure 6 On peut aussi remarquer que les résultats sont similaires entre les deux méthodes mises en œuvre, mais comparés aux résultats de 5 présente une plus grande amélioration des bords de l'image et d'autres éléments que l'on peut considérer comme du bruit. Ainsi, on peut dire que les filtres qui utilisent le module à gradient améliorent davantage les détails de l'image, y compris le bruit.

De plus, il est possible de démontrer en observant la mise en œuvre des algorithmes pour les convolutions d'image avec le gradient que le calcul de la norme du gradient $\|\nabla I\| = \sqrt{I_x^2 + I_y^2}$ (Code 3) peut être simplifié à $\|\nabla I\| = |I_x| + |I_y|$ (Code 4).

Affichage correct

Enfin, il est important de noter que pour obtenir une affichage correcte de l'image, il est nécessaire de considérer qu'après les opérations de convolution, il est possible que le résultat de N obtenu pour $I(x, y)$ soit supérieur ou inférieur à celui de du nombre de niveaux de gris que peut prendre $I(x, y)$. Pour résoudre cette question, l'image finale obtenue est limitée dans l'intervalle $[0, 255]$.

2 DéTECTEURS

Q4 - la fonction d'intérêt de Harris

Pour calculer les points d'intérêt de *Harris*, nous allons calculer, tout d'abord, la fonction d'intérêt de *Harris*. Pour faire cela nous allons nous baser sur les opérations suivantes:

- Calcul des dérivées premières à partir des dérivées de gaussienne avec un écart-type σ_1 .
- Calcul des termes de la matrice d'auto corrélation en calculant une moyenne locale des dérivées sous la forme d'une deuxième gaussienne d'écart-type $\sigma_2 = 2\sigma_1$.
- Calcul de la fonction d'intérêt $\Theta = \det(\Lambda) - \alpha \cdot \text{trace}(\Lambda)^2$ où $\alpha = 0.06$ typiquement.
- Calcule les maxima locaux de Θ supérieurs à un seuil prédéfini: (typiquement 1% de Θ_{max}).

Le premier point de notre algorithme est de faire une convolution de l'image principale avec le filtre de *Sobel* en utilisant la fonction *cv2.Sobel* qui permet de combiner soit un lissage Gaussien avec l'application d'un première dérivé: l'on applique ce filtrage soit dans la direction horizontale soit dans la direction verticale en obtenant les deux dérivés dans les deux direction I_x et I_y .

En suite le but est de calculer la matrice d'autocorrelation Λ pour chaque pixel qui est défini à l'intérieur d'un voisinage comme:

$$\Lambda = \begin{bmatrix} \Sigma(I_x)^2 & \Sigma(I_x \cdot I_y) \\ \Sigma(I_x \cdot I_y) & \Sigma(I_y)^2 \end{bmatrix}$$

Pour calculer ces termes nous avons fait un deuxième filtrage respectivement sur I_x^2 , I_y^2 et $I_x \cdot I_y$ en utilisant un noyau Gaussien défini par la fonction *cv2.getGaussianKernel* qui nous permet d'effectuer un somme sur un voisinage du pixel considéré.

En suite, nous allons calculer la *fonction de Harris* en utilisant la formule défini précédemment. Le dernier point de la procédure est déjà implémenté et il est composé par un *seuillage* et le *calcul des maxima locaux*.

Pour trouver les maxima locaux l'algorithme effectue aussi une *dilatation* avec un *noyau* carré $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ qui permet d'étendre les zones des maxima locaux et de, en suite, supprimer les maxima non locaux avec la commande *Theta_maxloc[Theta < Theta_dil] = 0.0* qui va mettre à 0 toute les pixels sauf lequels qui présentent des maxima.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Finalement, l'application applique une deuxième *dilatation* qui utilise le *noyau* suivant:

permet de grossir les points d'intérêt de la fonction de *Harris* et de les mettre en évidence comme des *x*. Cet effet peut être visualisé dans les images suivantes: C'est évident que la *dilatation* consiste en placer le

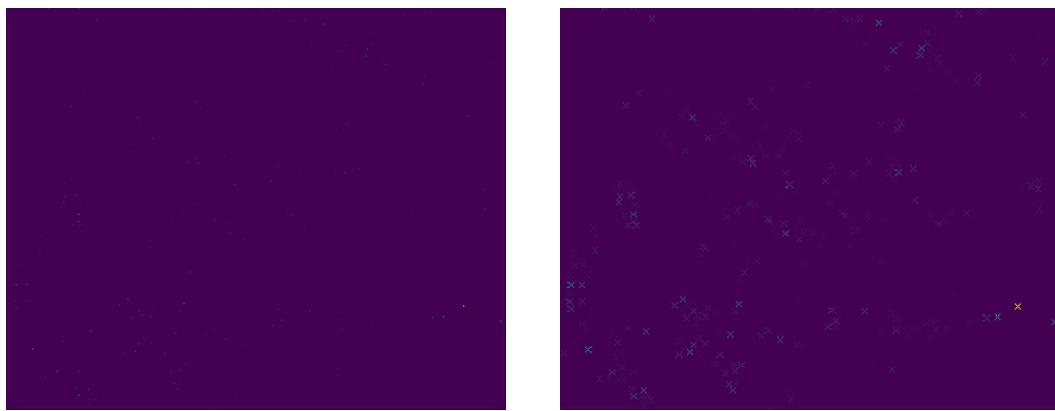


Figure 7: Points de *Harris* avant(à gauche) et après(à droite) la *dilatation*

noyau dans le points *claires* de l'image et substituer les points avec des croix qui permettent de les mieux visualiser et, en suite, le rendre rouge et les superposer à l'image finale.

Q5 - Les résultats du détecteur de Harris et l'effet des paramètres utilisés

La figure 8 montre les résultats obtenus avec la fonction d'intérêt de Harris décrite dans Q4. Comme on peut le constater, la sous-image du milieu montre les points d'intérêt identifiés en utilisant les paramètres $\alpha = 0.06$ et $\sigma = 1$, et on peut voir que les points de l'image qui varient dans les directions x et y sont bien représentés.

De plus, afin de réaliser une évaluation de la fonction implémentée, une comparaison visuelle a été effectuée entre les points d'intérêt trouvés avec la fonction implémentée et avec les points d'intérêt trouvés

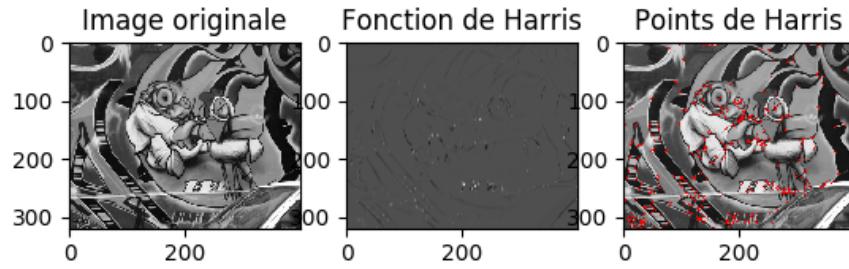


Figure 8: Résultats obtenus avec la fonction d'intérêt de Harris pour $\alpha = 0.06$ et $\sigma = 1$

avec la fonction `cv2.cornerHarris(img, blockSize, ksize, k)`. de OpenCV qui calcule la fonction de Harris, comme indiqué dans Code 5 ci-dessous. La figure 9 montre le résultat de la comparaison.

Code 5: Fonction d'intérêt de Harris avec Open CV

```
img = cv2.imread('..../Image_Pairs/Graffiti0.png')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray = np.float32(gray)
dst = cv2.cornerHarris(gray,2*sigma,3,alfa)
dst = cv2.dilate(dst,None)
img[dst>0.01*dst.max()]=[0,0,255]
cv2.imshow('dst',img)
```

En regardant Figure 9a et Figure 9b, nous voyons que la grande majorité des points trouvés avec Code 5 correspondent à des points trouvés avec la fonction `cv2.cornerHarris()`, montrant ainsi que les résultats obtenus sont conformes aux résultats attendus.



(a) Résultat de la fonction d'intérêt en utilisant le code créé



(b) Résultat de la fonction d'intérêt en utilisant OpenCV

Figure 9: Comparaison entre la fonction d'intérêt créée a et celle dans OpenCV

Analyse des paramètres

Pour calculer la fonction d'intérêt de Harris, les fonctions `cv2.Sobel(src, ddepth, xorder, yorder, ksize = sigma)` et `cv2.getGaussianKernel(ksize, sigma[, ktype])` ont été utilisées. La fonction `cv2.Sobel()` est une combinaison de la dérivée directionnelle de l'image avec le lissage gaussien, ayant comme paramètres l'image d'entrée (`src`), l'ordre des dérivées dans chaque direction (`xorder` ou `yorder`) et la taille du noyau de Sobel à utiliser (`ksize`), qui dans ce cas est égal à `sigma`.

La fonction `cv2.getGaussianKernel()` est utilisée pour générer un noyau gaussien afin de moyennier la fenêtre $W_{(x,y)}$, et a pour paramètres la taille du noyau à utiliser (`ksize`) et la variance gaussienne (`sigma`)

Le paramètre `ksize` doit être correctement sélectionné en fonction de la gaussienne à utiliser, car il se peut que le noyau ne couvre qu'une partie de la gaussienne, ce qui modifie l'effet souhaité. Le paramètre `sigma` est directement lié à l'échelle à utiliser pour l'analyse.

Taille de la fenêtre de sommation

La taille de la fenêtre de sommation doit être choisie pour détecter une quantité souhaitée de points d'intérêt. Pour les petites fenêtres, par exemple, il se peut que les points pouvant être considérés comme des points d'intérêt soient considérés comme des bords, en raison du choix fait par la fonction de Harris lors de la détermination du maximum local.

Valeur de α

Les points d'intérêt d'une image sont essentiellement identifiés dans les régions dans lesquelles il existe des variations d'intensité importantes des gradients dans toutes les dimensions et directions possibles. Dans la

fonction d'intérêt de Harris, $\Theta_I(x, y) = \det(\Xi_I(x, y)) - \alpha \cdot \text{trace}(\Xi_I(x, y))$, le paramètre α est un paramètre que sert à examiner les valeurs propres de la matrice pour détecter les points d'intérêt et pénaliser les points que sont considérés comme bords. Par exemple, les régions ayant $\lambda_1 \gg \lambda_2$ ou $\lambda_2 \gg \lambda_1$ peuvent être considérés comme points que appartient à des bords, alors que des points ayant λ_1 et λ_2 grands et $\lambda_1 \sim \lambda_2$, sont considérés comme des vrais points d'intérêt. Cela veut dire que avec une valeur de alpha plus grande, c'est possible de avoir moins de faux points d'intérêt, alors que avec une valeur de lambda petite les points que peuvent appartenir à des bords peuvent être détectés. Dans notre cas, la valeur de alpha utilisée était 0.06, et on a pu constater que pour une valeur de $\alpha > 0.25$, aucun point d'intérêt était détecté.

Le calcul sur plusieurs échelles

L'analyse multi-échelle a pour objectif d'obtenir une interprétation invariante de l'échelle. L'échelle d'une image est liée à la longueur focale de l'image et sa résolution spatiale.

Dans une situation idéale, lors du changement d'échelle d'une image, l'interprétation de ses caractéristiques importantes ne devrait pas changer, ce qui permettrait de relier l'information obtenue à une échelle donnée à l'information obtenue à une autre échelle, tout en maintenant la cohérence de l'information.

Dans le cas de la fonction Harris, le problème est basé sur le fait que son application n'est pas invariante à l'échelle et que, par conséquent, la répétabilité de la corrélation entre des points distincts peut varier en fonction de la modification de l'échelle de l'image. Pour effectuer une analyse multi-échelle de la fonction de Harris, il est nécessaire de s'appuyer sur la valeur du facteur gamma de lissage gaussien, car la fonction de Harris n'utilise que la première dérivée de l'image.

Par conséquent, pour calculer la fonction d'intérêt de Harris Θ_I^s sur une échelle s donnée, les premières dérivées de l'image (I_x^s et I_y^s) sont calculées par convolution avec un gaussien de variance de $2s$. Après cela, la matrice de covariance Ξ_x^s est calculée dans la fenêtre de sommation W pour un lissage gaussien de la variance de $2s'$.

Q6 - Les détecteurs ORB et KAZE

Les détecteurs sont des outils qui permettent d'identifier des parties significatives à l'intérieur d'une image qui peuvent être des *coins* ou des *blobs*. Les *coins* désignent un changement d'une caractéristique autour d'un pixel. Les *blobs* désignent une zone de pixels contigus qui partagent une même caractéristique.

Le détecteur ORB

Le détecteur *ORB* (*Oriented FAST and rotated BRIEF*) est une fusion entre le détecteur *FAST* et le descripteur *BRIEF*. Dans cette partie nous allons mettre en évidence ce que le détecteur *ORB* ajoute au détecteur *FAST*: la possibilité de calculer l'*orientation caractéristique* des points.

FAST est un détecteur qui sélectionne les points dont le voisinage circulaire présente des plages contiguës assez longues de points significativement plus clairs ou plus sombres. En particulier, pour un point donné, il va examiner les pixels qui ont une même distance de ce point et il va évaluer ses valeurs: si plusieurs pixels proches ont des valeurs similaires, le pixel sera détecté. Il s'agit d'un détecteur multi-échelle à haute efficacité qui est rapide dans le calcul des points d'intérêt.

Le détecteur *ORB*, en reprenant le principe de *FAST*, permet d'ajouter l'*orientation caractéristique* au point qui est donné par le vecteur \vec{PO} où P est le point détecté par *ORB* et O est le centre de gravité du noyau carré qui circonscrit le cercle utilisé par le *FAST*, qui est calculé de façon géométrique comme, par exemple, la moyenne des positions de pixels pondérée par le niveau de gris.

Le détecteur KAZE

Le détecteur *KAZE* est un détecteur qui combine le principe de *diffusion anisotrope* avec lequel de *maxima locaux du Hessien*: il est utilisé dans la détection des caractéristiques en deux dimensions (horizontale et verticale) dans une image en utilisant un échelle non linéaire.

Il se base essentiellement sur la résolution d'une équation aux dérivées partielles (EDP) du filtre à *diffusion anisotrope* qui signifie que la fonction chaleur va pas être diffusé uniformément sur l'image comme dans le cas de la gaussienne.

$$\frac{\delta I}{\delta t} = \text{div}(c \nabla I) = c \Delta I + \nabla c \cdot \nabla I$$

Il est basé sur un filtrage de diffusion non linéaire combiné à une fonction de conductivité.

Dans cette méthode, le *fonction de conductance* c , comme proposé par Perona et Malik, est variable et dépend de l'image:

$$c(x, y, t) = g(||\nabla I(x, y, t)||)$$

Cet opérateur donc n'utilise pas forcément un noyau Gaussien afin de pouvoir exploiter le principe de la *diffusion anisotropie*. Le but est d'obtenir des caractéristiques qui présentent une plus grande répétabilité.

Les principaux paramètres propres à chaque détecteur et leur effets

Nous allons analyser les fonctions qui sont présents dans le fichier *Features_Detect.py* pour la détection des points selon les méthodes *ORB* et *KAZE*.

Pour *ORB* le détecteur est défini avec le code:

```

kp1 = cv2.ORB_create(nfeatures = 250,          #Par dfaut : 500
                      scaleFactor = 2,    #Par dfaut : 1.2
                      nlevels = 3)       #Par dfaut : 8

```

où:

- **nfeatures** est le numéro de caractéristiques à maintenir.
- **scaleFactor** qui permet de travailler à plusieurs échelles et indique la quantité qui est utilisé dans la modification de la largeur du cercle (*schéma à pyramide*). En particulier, pour une *scaleFactor*=2, chaque niveau aura 4 fois moins de pixels par rapport au niveau précédent.
- **nlevels** est le nombre total de fois que on va modifier la largeur du cercle de façon de passer de la largeur minimale à laquelle maximale(descendre ou monter le schém à pyramide).

Pour *KAZE* le détecteur est défini avec le code:

```

kp1 = cv2.KAZE_create(upright = False,           #Par dfaut : false
                      threshold = 0.001,      #Par dfaut : 0.001
                      nOctaves = 4,           #Par dfaut : 4
                      nOctaveLayers = 4,      #Par dfaut : 4
                      diffusivity = 2)        #Par dfaut : 2

```

où:

- **upright** est une valeur *boolean* qui, si activé, habilite des descripteurs qui n'ont pas la propriété d'invariance à la rotation.
- **threshold** qui permet de choisir la seuil pour accepter où refuser un point.
- **nOctaves** représente la quantité des ensembles d'images de même échelle qui sera analysé par le descripteur. En particulier, un octave représente une échelle de l'image.
- **nOctavesLayers** qui est le numero de sous-niveaux pour chaque niveau de échelle.
- **diffusivity** qui permet de choisir le type de *diffusion anisotrope*: *DIFF_PM_G1*(0), *DIFF_PM_G2*(1), *DIFF_WEICKERT*(2) or *DIFF_CHARBONNIER*(3). Dans notre cas, nous avons une diffusivité de type *DIFF_WEICKERT*.

Enfin les points sont détectés avec l'instruction: *pts1 = kp1.detect(gray1, None)* et en suite affiché sur l'image avec l'instruction: *img1 = cv2.drawKeypoints(gray1, pts1, None, flags=4)*

La répétabilité des détecteurs

La **répétabilité** d'un détecteur est la propriété qui permet d'évaluer sa capacité de ne pas changer les points détectés quand l'image traitée subit des déformations.

Nous pouvons affirmer qu'un détecteur présente une bonne **répétabilité** si les points mis en évidence sur une image ne changent pas de position quand l'image résulte déformé.

Nous allons analyser l'effet des deux détecteurs reportés ci-dessus sur un image qui a subi une déformation.

Dans le cas de *ORB*, comme nous pouvons observer dans l'image 10, nous avons des difficultés à différentier un point d'un autre car l'application sur différentes échelles avec des différents rayons de courbure nous fait obtenir une image avec des superpositions qui nous empêchent de bien identifier les points d'intérêt.

Pour l'analyse de la répétabilité des détecteurs, nous allons nous concentrer sur la fenêtre et sur la porte de la maison de gauche: ici nous pouvons visualiser que les points détectés dans la première image ne sont plus sélectionnés dans l'image à coté: cela peut être dû au fait que, dans l'image de gauche, ces points sont devenus des points de contour pour lesquels n'est plus possible de définir un cercle suffisamment grande. Par ailleurs, si on regarde le petit bâtiment de premier plan les points d'intérêt ne changent pas de position après la rotation.

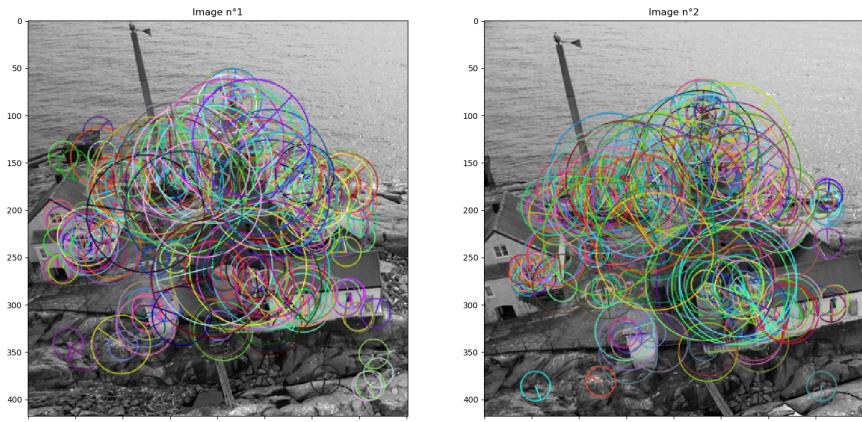


Figure 10: Répétabilité du détecteur *ORB*

Dans le cas de *KAZE*, reporté dans l'image 11, les points qui sont mis en évidence dans l'image de gauche comme par exemple les points sur la partie haute du poteau sur l'image de gauche correspondent à lesquels dur l'image de droite. De plus, les points qui indiquent les fenêtres du bâtiment au centre de l'image sont les mêmes et ne changent pas. Par rapport au cas précédent, c'est plus facile de différentier un point d'un autre car le cercle qui est lui associé est plus petite et il ne superpose pas avec des autres cercles.

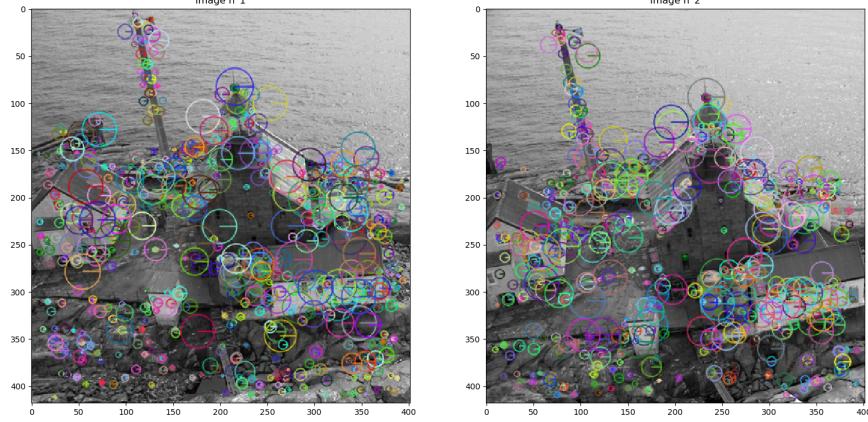


Figure 11: Répétabilité du détecteur *KAZE*

Nous pouvons donc conclure que, dans l'image proposée le comportement du détecteur *KAZE* permet de identifier les points d'intérêt plus facilement du détecteur *ORB* pour le fait que le noyau que on applique a la possibilité de changer selon la zone de l'image que on est en traine de traiter.

3 Descripteurs et Appariement

Q7 - Le principe des descripteurs attachés aux points ORB et KAZE

L'objectif des *descripteurs* est lequel de décrire les points d'intérêt mis en évidence par les *détecteurs*. Leur travail s'agit en analyser le voisinage proche de chaque point afin d'obtenir un vecteur descripteur pour chaque point d'intérêt.

Les descripteurs présentent deux caractéristiques principales: l'*invariance* et la *discriminance*. L'*invariance* nous permet de décrire un même point d'intérêt de la même manière dans deux images distinctes et indique aussi le fait que un détecteur n'est pas influencé par les transformations géométriques. La *discriminance* est la capacité à discerner deux points d'intérêt différents dans deux images distinctes.

Les descripteurs des points ORB

Les descripteurs associés aux points *ORB* sont les descripteurs *BRIEF* (*Binary Robust Independent Elementary Features*).

Le détecteur *BRIEF* consiste en considérer une série des N paires aléatoires de pixels sur la zone d'intérêt, soit le voisinage du point d'intérêt, et utiliser la *distance de Hamming*. Pour chacune de ces N paires, nous allons comparer leur valeurs et nous allons donner un bit de valeur 1 dans le cas où la valeur, en niveau de gris, associée au premier pixel $p(x)$ de la couple est inférieure à laquelle du deuxième $p(y)$ autrement nous allons donner un bit de valeur 0.

$$f(p, x, y) = \begin{cases} 1 & \text{si } p(x) < p(y) \\ 0 & \text{si } p(x) \geq p(y) \end{cases}$$

Le résultat sera un *vecteur descripteur* composé par N bits. Le problème plus grand dans le cas du descripteur *BRIEF*, est l'absence d'invariance aux rotations.

Nous arrivons donc au descripteur *ORB* qui permet d'ajouter l'orientation à l'information fourni par *BRIEF*: pour chaque ensemble de n échantillons il va définir une matrice qui contient leur coordonnées. Ensuite, il définit la matrice de rotation qui sert pour tourner la première matrice défini et il construit une table de conversion composé par des modèles *BRIEF* pré-calculés. Une fois que l'orientation est disponible, *ORB* applique une correction sur les points sélectionnés et qui seront utilisés pour calculer le *vecteur descripteur*.

Les descripteurs des points KAZE

Les descripteurs associés aux points *KAZE* sont les descripteurs *SURF* (*Speeded-Up Robust Features*).

Pour le calcul de l'orientation, *SURF* utilise les *ondelettes de Haar*: elles sont calculées en un voisinage circulaire de taille 6 fois l'échelle pour chaque point d'intérêt selon la direction horizontale et selon la direction verticale. Les réponses des *ondelettes* sont représentées en deux dimensions autour du point d'intérêt. Enfin, pour calculer l'orientation, il est nécessaire considérer une cône de 60 deg: le cône qui contient le plus grande nombre de réponse aux *ondelettes* donnera la direction associée au point.

En outre, pour constituer le *vecteur descripteur*, cette méthode considère un voisinage d'un point, le découpe en 64 sous-régions à l'intérieur de lesquelles il calcule le vecteur utilisant encore les *ondelettes de Haar*: son abscisse et son ordonnée est la somme des réponses selon la direction horizontale et laquelle verticale. On construit alors le vecteur v de chaque région à partir des 4 vecteurs de ses sous-régions : $v = (\Sigma dx, \Sigma |dx|, \Sigma dy, \Sigma |dy|)$.

Le *vecteur descripteur* est la concaténation des 16 vecteurs v et sa dimension est donc de 64. Vu que le calcul est fait dans les deux directions nous avons la possibilité d'avoir une robustesse plus grande par rapport au bruit.

Les propriétés qui rendent l'appariement invariant par changement d'échelle et par rotation

Pour *ORB*, le fait d'être invariant au changement d'échelle, est donné par le fait que la détection est faite multi-échelle et donc nous allons utiliser, pour trouver les points d'intérêt, des cercles avec de différents rayons. De plus, comme nous avons déjà dit avant, la possibilité pour chaque point d'intérêt de calculer son orientation déjà à niveau du détecteur permet de garantir l'invariance par rotation. De plus, le fait que deux points sont choisis aléatoirement donne une haut niveau de decorrélation qui permet d'éviter la redondance d'information et qui donne des différents informations au descripteur permettant, enfin, d'avoir une haut variance entre les données.

Pour *KAZE*, l'invariance au changement d'échelle est donné par le fait que, pour la détection des points d'intérêt, la fonction de conductance c est adapté à chaque image. La propriété d'être invariant à la rotation est donné par le fait que le descripteur *SURF*, en utilisant les réponses aux *ondelettes de Haar*, permet de reconstituer l'orientation.

Q8 - Les performances des trois stratégies d'appariement de points d'intérêt

Le but de l'*appariement* est de trouver les points correspondants dans une image et dans la même image qui a subit une transformation.

Le but est de décrire la performance des descripteurs.

Dans notre cas, nous allons analyser trois différentes méthodes: *Cross Check*, *Distance Ratio* et *FLANN*.

Stratégie Cross Check

La stratégie de *Cross Check*, aussi appelé *Reverse Matching*, consiste en effectuer les correspondances dans les deux sens et les conserver seulement dans le cas où elles correspondent.

Le but de ce double contrôle est de réduire les erreurs dans le cas où dans une des deux images le point n'est plus présent.

Dans les deux images suivantes, 12 et 13, nous allons analyser qualitativement le comportement des descripteurs.



Figure 12: Appariement *Cross Matching* dans le cas *ORB*

Dans l'image 12 nous pouvons observer que la major partie des points trouve un correspondant dans l'autre image: bien que la majorité des points est bien associé avec son correspondant, nous avons que pour certains points lequel associé n'est pas lequel correct. Ces points sont bien visibles pour le fait que la ligne de connexion a une direction différente par rapport à laquelle associée à la majorité des points.

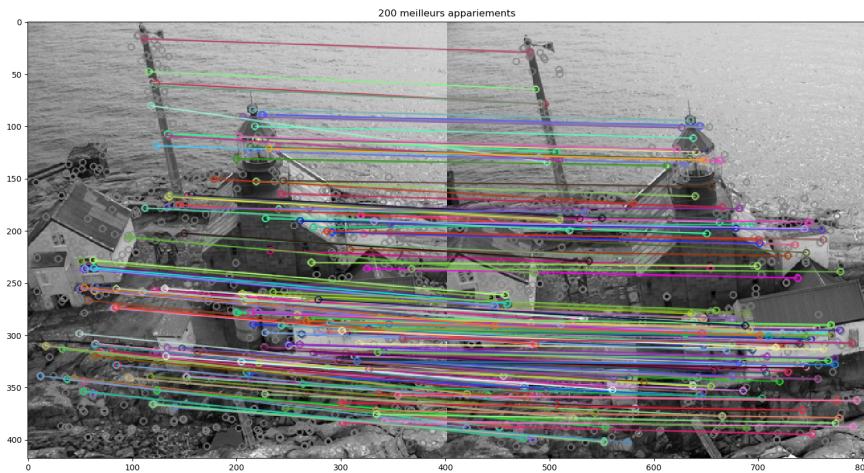


Figure 13: Appariement *Cross Matching* dans le cas *KAZE*

Dans l'image 13 nous observons que nous n'avons plus des directions différentes à l'intérieur de l'image: presque tous les points d'une image qui trouvent un équivalent dans l'autre image sont caractérisés par des lignes avec la même direction.

Cependant, dans ce cas, beaucoup plus de points ne trouvent pas leurs équivalents.

On peut conclure en disant que, avec *ORB*, presque tous les points trouvent le point équivalent dans l'autre image aussi si les associations ne sont pas forcément correctes. Par contre, avec *KAZE*, seulement aux points qui trouvent leur équivalent sont associés une ligne.

Stratégie Distance Ratio

La stratégie de *Distance Ratio* consiste en effectuer le rapport des distances vers les deux descripteurs les plus proches.

En particulier, il va considérer le rapport entre les distances entre le descripteur f_1 sur un image et les deux descripteurs les plus proches f_2 et f_2' sur la deuxième image comme:

$$r = \frac{d(f_1, f_2)}{d(f_1, f_2')}$$

Selon le cas, il faut choisir le bon valeur de r pour lequel on conservera les appariements. Normalement le valeur de ratio qui est considéré sera plus petit de 0.8. En fait il a été vérifié que, pour un ratio $r \geq 0.8$, la probabilité que les appariement soient incorrecte est plus élevé. Le plus le ratio se rapproche à l'unité, le plus l'ambiguïté sera haute et on aura une probabilité plus haute de faire une appariements incorrects.

Dans la figure 14, nous pouvons visualiser que peu de points trouvent un point équivalent sur l'image modifiée bien que les correspondances sont tous correctes. C'est un cas différent pas rapport à lequel visualisé dans l'image 12 où se produisait la situation opposée.

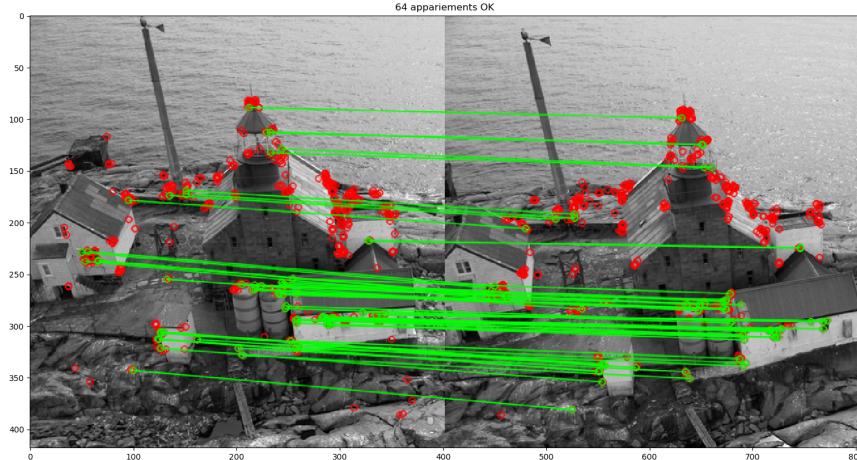


Figure 14: Appariement *Distance Ratio* dans le cas *ORB*

Dans la figure 15, nous avons un nombre de appariements plus élevés que dans le cas de *ORB*. Ici, comme dans le cas de la figure 13, la grande majorité des appariements ressemblent corrects et associent un point sur l'image de droite à le mémé points sur l'image transformée.

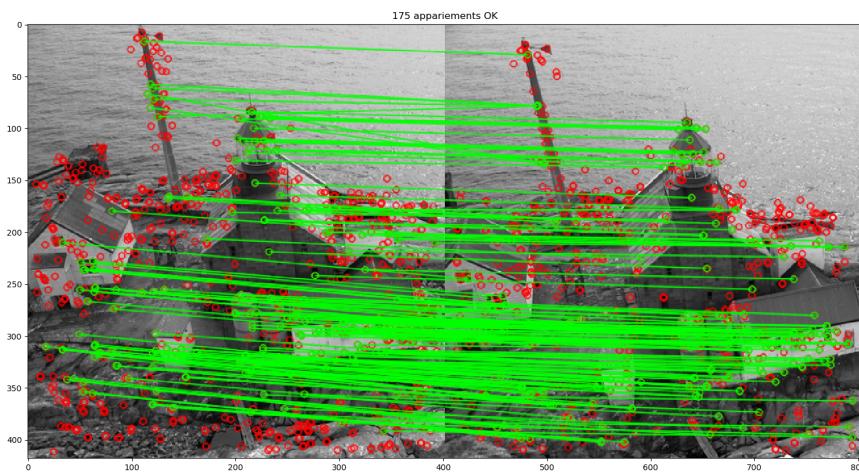


Figure 15: Appariement *Distance Ratio* dans le cas *KAZE*

Nous pouvons commenter que, dans ce cas, le comportement de l'appariement est beaucoup plus efficace dans *KAZE* car il arrive à trouver un nombre de appariements correctes qui est environ trois fois lequel dans le cas de *ORB*. Cela est dû au fait que l'évaluation en *ORB* est fait en utilisant le niveau de gris de deux points choisi aléatoirement autour du point d'intérêt et le résultat qui dérive de cette opération a une variance plus grande dans le temps par rapport au *KAZE* qui considère une grille tout autour du point d'intérêt et calcul les caractéristiques du point en utilisant les *ondelettes de Haar*.

Stratégie FLANN-KD_tree

La *FLANN* (*Fast Library for Approximate Nearest Neighbor*) permet de réaliser la méthode des arbre k-dimensionnelles.

Cette méthode permet de accélérer la recherche des plus proches voisins.

La première étape est laquelle de créer l'arbre en divisant les données récursivement en deux parties selon une caractéristique des points qui peut être le médiane, la direction de plus grande variance où encore une direction aléatoire parmi les plus grandes variances.

En suite, il y a la partie de recherche à l'intérieur de l'arbre qui consiste en descendre dans l'arbre pour

trouver le plus proche voisin dans la branche considéré.

Aussi dans ce cas nous avons une meilleure performance dans le cas de *KAZE* dans l'image 17 par rapport au cas *ORB*.

De plus, dans ce cas, la différence de performance est encore plus évidente.

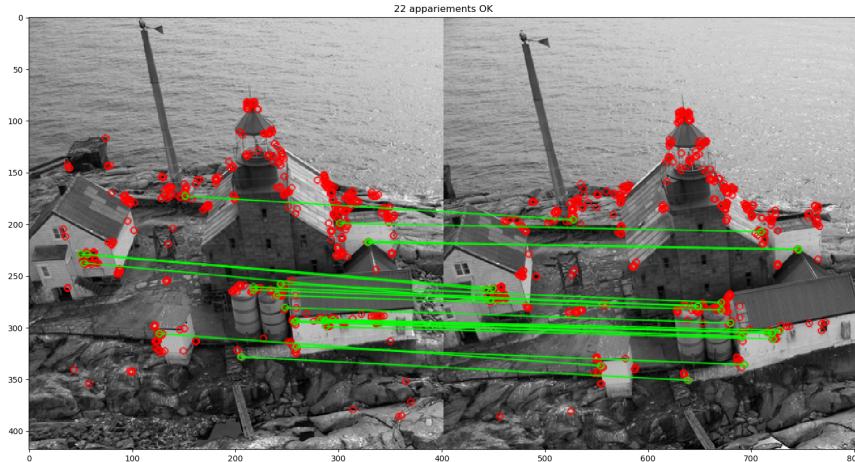


Figure 16: Appariement *FLANN* dans le cas *ORB*

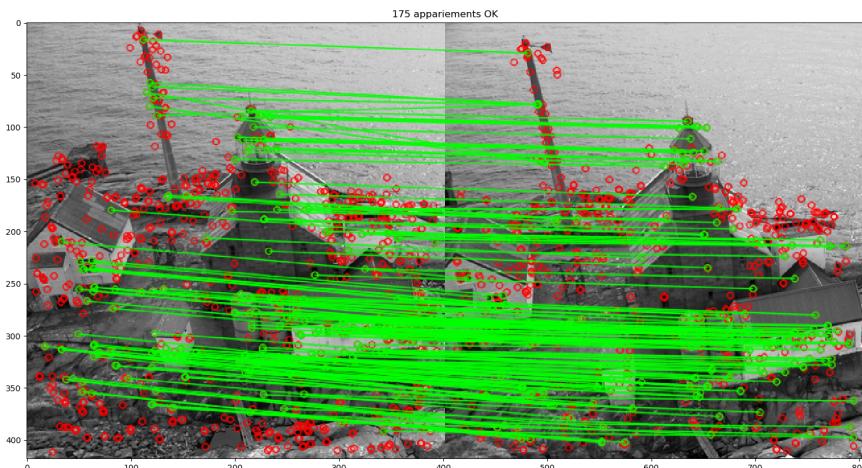


Figure 17: Appariement *FLANN* dans le cas *KAZE*

N.B.: Pour faire fonctionner le code dans le cas de *ORB* nous avons dû modifier le code de la fonction: en particulier, nous avons modifié la ligne 67 en `matches = flann.knnMatch(np.asarray(desc1, np.float32), np.asarray(desc2, np.float32), k=2)`.

Utilisation des différents distances pour les deux descripteurs

Le descripteur *ORB* est un descripteur binaire car il utilise le descripteur *BRIEF* et donne en sortie des vecteurs binaires descripteurs. En particulier, le calcul de distance sera fait au travers de la *distance de Hamming* étant donnée qu'elle est plus performante pour des vecteurs binaires.

Pour le descripteur *KAZE*, on utilise les dérivés directionnelles et nous allons donc avoir des valeurs réels: raison pour laquelle l'utilisation d'une distance différente est plus appropriée. La distance utilisées est laquelle *Euclidienne* ou de *norme-L2*.

FLANN et la mauvaise performance avec ORB

Dans le cas *FLANN* et en mineur quantité dans le cas de *Distance Ratio*, les performances obtenus avec le descripteur *ORB* sont inférieures par rapport à lesquelles obtenus avec *KAZE*.

Cela peut être expliqué par le fait que dans *ORB*, en se basant sur le descripteur *BRIEF*, nous allons choisir des points de façon aléatoire et les comparer avec la *distance de Hamming*. Un arbre-KD utilise la médiane des points pour différencier les groupes de données et avoir des informations en *distance de Hamming* n'est pas optimale pour la construction de cette arbre.

Q9 - Une évaluation quantitative de la qualité des appariements après avoir déformé l'image

Pour évaluer quantitativement la qualité des appariements nous nous sommes concentrés sur les codes des descripteurs en modifiant les codes des fichiers *Features_Match_CrossCheck.py*, *Features_Match_FLANN.py*

et *Features_Match_RatioTest.py* et en proposant trois nouveaux codes: un pour chaque descripteur.

Dans un premier moment, nous avons considéré la comparaison du comportement du descripteurs sur l'image d'origine afin de pouvoir obtenir le nombre total d'appariements corrects. En suite, nous avons ajouté une comparaison sur le comportement des descripteurs entre l'image d'origine et l'image qui a subi une transformation géométrique(codes *Q9_CrossCheck_old.py*, *Q9_RatioTest_old.py* et *Q9_FLANN_old.py*) en calculant le **nombre de appariements** qui sont bons dans le différents cas: image d'origine (où on aura la valeur maximale) et images transformées.

Dans un deuxième moment nous nous sommes dédiés à une vrai analyse **qualitative** sur les descripteurs liés aux points d'intérêt (codes *Q9_CrossCheck_new.py*, *Q9_RatioTest_new.py* et *Q9_FLANN_new.py*). Dans ces codes nous avons considéré les coordonnées des points d'intérêt calculé par la fonction *kp1.detectAndCompute(gray1, None)* qui se base sur l'objet *keypoints* précédemment créé. En particulier nous avons considéré les points détecté soit dans l'image d'origine soit dans l'image modifiée et étudié leur appariement grâce aux extensions *queryIdx* et *trainIdx*. Pour chaque appariement des points nous avons comparé leur coordonnées et considéré un appariement comme bon si les coordonnées étaient suffisamment proches les unes des autres.

Nous avons pris les coordonnées de l'image d'origine (x_1, y_1) , nous avons appliqué la même transformation appliquée à l'image en obtenant (qx_1, qy_1) et nous les avons comparés aux coordonnées des points obtenus dans l'image transformé (x_2, y_2) .

En particulier nous allons considérer un appariement comme bon dans le cas où la distance entre ce deux points est plus petit que une certaine valeur ϵ qui nous avons choisi comme 1.5.

$$d = \sqrt{(x_2 - qx_1)^2 + (y_2 - qy_1)^2} \leq \epsilon$$

Les différents transformations que nous avons appliqués sont:

- **rotation** de 10 dégrée et de 40 dégrée;
- **redimensionnement** avec un facteur de 1.5(grossissement) et un facteur de 0.5(réduction);

Pour changer la transformation il faut aller dans la section du code *Transformation de l'image originale* et choisir le valeur 2 pour une rotation et la valeur 3 pour une redimensionnement.

Nous allons reporter ci-dessous les résultats qui nous avons obtenus dans les différents cas:

Cross Check

Les fichiers auxquels on doit se référer sont *Q9_CrossCheck_old.py*(1) et *Q9_CrossCheck_new.py*(2).

Nous avons reporté ci-dessous les valeurs (le nombre de matches trouvés) obtenus dans les trois images soit pour *orb* que pour *kaze*:

Image	ORB	KAZE
d'origine	500	869
10 deg	325(65%)	476(54.77%)
40 deg	283(56.6%)	425(48.9%)

Table 1: Nombre des appariements dans les images

Nous pouvons observer ce que on attendait: le nombre de matches va baisser si on déforme de plus en plus l'image soit dans le cas de *orb* que dans le cas de *kaze* même si dans le cas de *orb* la perte en pourcentage est plus faible.

On va aussi reporter les trois figures obtenus pour les deux cas:

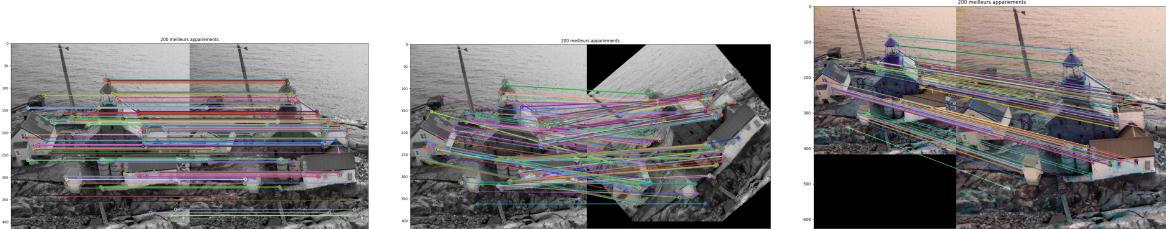


Figure 18: *Cross Check* avec *orb* pour: image d'origine, tourné de 40deg et avec un redimensionnement x1.5

En suite nous allons reporter les résultats obtenus dans l'analyse **quantitative** de la qualité des appariements: en particulier nous allons reporter la pourcentage des appariements que nous avons classifiés comme bon sur le nombre total des appariements. Nous pouvons voir que la performance est constante avec des différentes transformations.

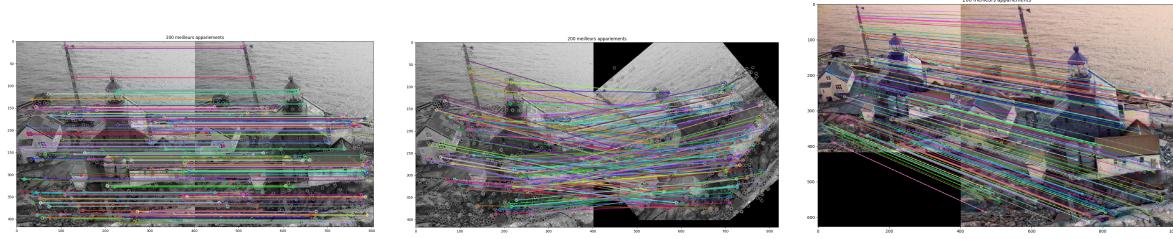


Figure 19: *Cross Check* avec *kaze* pour l'image d'origine, tourné de 40deg et avec le redimensionnement x1.5

Image	ORB	KAZE
Rotation 10 deg	76.27%	82.16%
Rotation 40 deg	74.19%	77.16%
Redimensionnement x1.5	61.60%	67.28%
Redimensionnement x0.5	75%	60.48%

Table 2: Pourcentage qui indique la qualité des appariements effectués

Ratio Test

Les fichiers auxquels on doit se référer sont les fichiers *Q9_RatioTest_old.py*(3) et *Q9_RatioTest_new.py*(4).

Nous avons reporté ci-dessous le nombre de appariements corrects et la pourcentage par rapport à l'image originelle obtenus dans les images soit pour *orb* que pour *kaze*:

Image	ORB	KAZE
d'origine	500	869
10 deg	310(62%)	426(49,02%)
40 deg	284(56,8%)	349(40,16%)

Table 3: Nombre des appariements dans les images

Dans ce cas aussi, le nombre de appariement et, de conséquence le ratio, est plus petit quand on applique une rotation plus accentuée.

En suite nous avons reporté les résultats de la pourcentage des appariements bons obtenus dans l'analyse **quantitative** de la qualité des appariements: Dans ce cas, nous pouvons observer que si on réduit la taille

Image	ORB	KAZE
Rotation 10 deg	62.20%	51.55%
Rotation 40 deg	55.40%	40.16%
Redimensionnement x1.5	38.20%	53.86%
Redimensionnement x0.5	37.20%	16.46%

Table 4: Pourcentage qui indique la qualité des appariements effectués

des images le *RatioTest* ne permet pas d'associer des bons points: cela peut être dû au fait que, en réduisant la taille des images, le rapport entre les distances va, de plus en plus, se rapprocher à l'unité en générant plus d'ambiguïté. On va aussi reporter les figures obtenues:

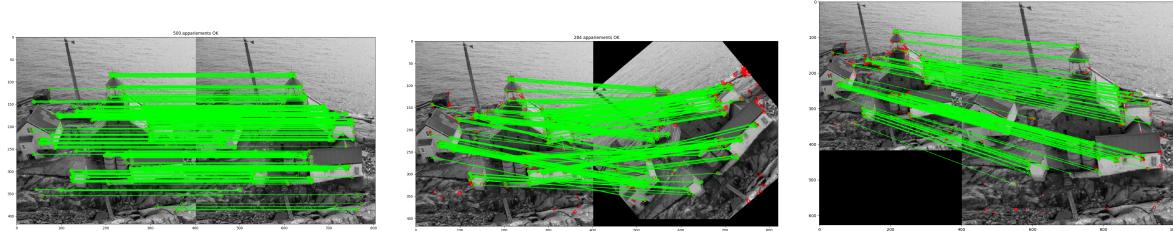


Figure 20: *Ratio Test* avec *orb* pour l'image d'origine, tourné de 40deg et avec le redimensionnement x1.5

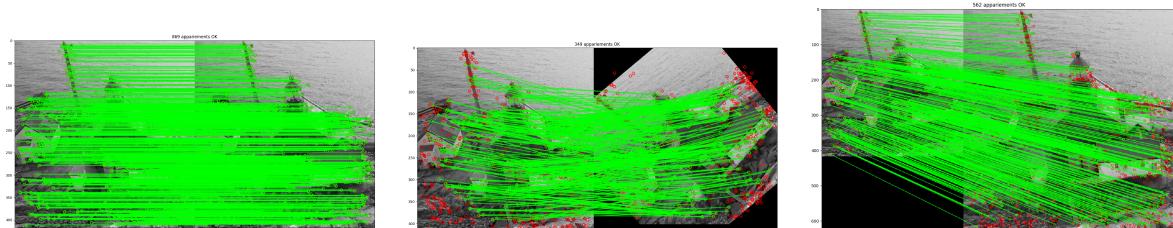


Figure 21: *Ratio Test* avec *kaze* pour l'image d'origine, tourné de 40deg et avec le redimensionnement x1.5

FLANN

Le fichiers auxquels on doit se référer sont *Q9_FLANN_old.py*(5) et *Q9_FLANN_new.py*(6).

Nous avons reporté ci-dessous les nombre des appariements obtenus dans les images:

Image	ORB	KAZE
d'origine	500	869
10 deg	155(31%)	426(49,02%)
40 deg	144(28,8%)	349(40,16%)

Table 5: Nombre des appariements dans les images

En suite nous allons reporter les résultats obtenus dans l'analyse **quantitative** de la qualité des appariements: en particulier nous allons reporter la pourcentage des appariements que nous avons classifiés comme bon.

Image	ORB	KAZE
Rotation 10 deg	55.80%	51.55%
Rotation 40 deg	51.20%	40.16%
Redimensionnement x1.5	33.60%	53.86%
Redimensionnement x0.5	30.60%	16.48%

Table 6: Pourcentage qui indique la qualité des appariements effectués

On a reporté aussi les images obtenues:

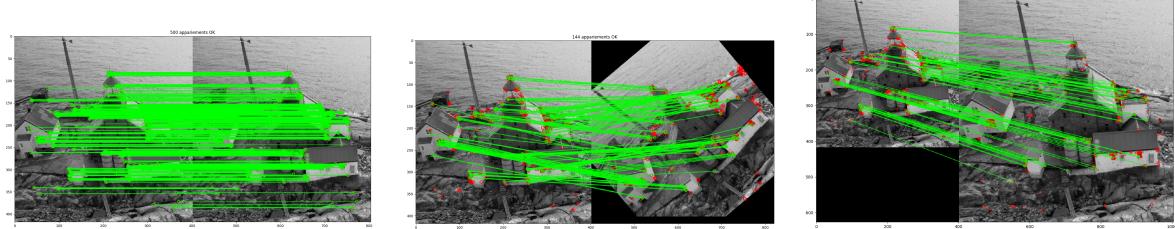


Figure 22: *FLANN* avec *orb* pour:image d'origine, tourné de 40deg et avec le redimensionnement x1.5

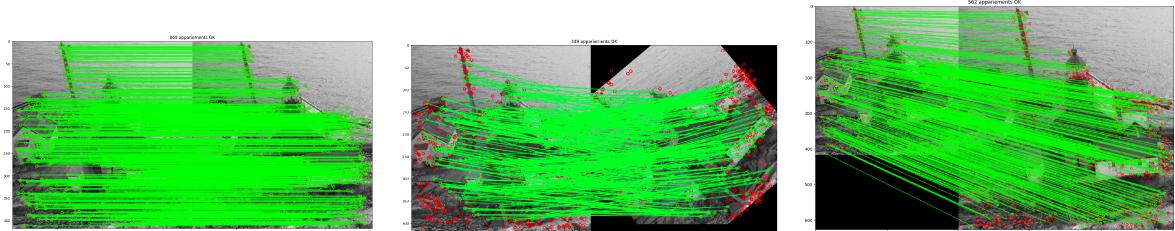


Figure 23: *FLANN* avec *kaze* pour:image d'origine, tourné de 40deg et avec le redimensionnement x1.5