



IP PARIS

ÉCOLE NATIONALE SUPÉRIEURE DE TECHNIQUES AVANCÉES

## **ANALYSE ET INDEXATION D'IMAGES**

Parcours Robotique

Année universitaire : 2020/2021

---

### **TP1 : Détection et Appariement de Points Caractéristiques**

---

**Auteurs :**

M. AHMED YASSINE HAMMAMI

MME. HANIN HAMDI

## Objectifs

L'objectif de ce TP est de se familiariser avec la bibliothèque de traitement d'images OpenCV sous Python, et d'expérimenter, critiquer et mettre en œuvre différentes techniques de représentation pour l'appariement de structures locales dans les images, en suivant les différentes étapes de la problématique :

- DéTECTeur : Comment réduire le support de la représentation.
- Descripteur : Quelle information attacher à chaque point du support.
- MéTRIQUE : Quelle mesure utiliser pour appairer des points.
- Recherche : Comment coder et parcourir l'espace des points d'un modèle.

## Format d'images et Convolutions

### Question 1 :

Dans cette question, on va analyser la différence entre la méthode du calcul direct par balayage du tableau 2d et la méthode du calcul qui utilise la fonction pré définie dans OpenCV *filter2d*.

Après avoir exécuter le code *Convolution.py*, on observe les deux images après filtrage par convolution :

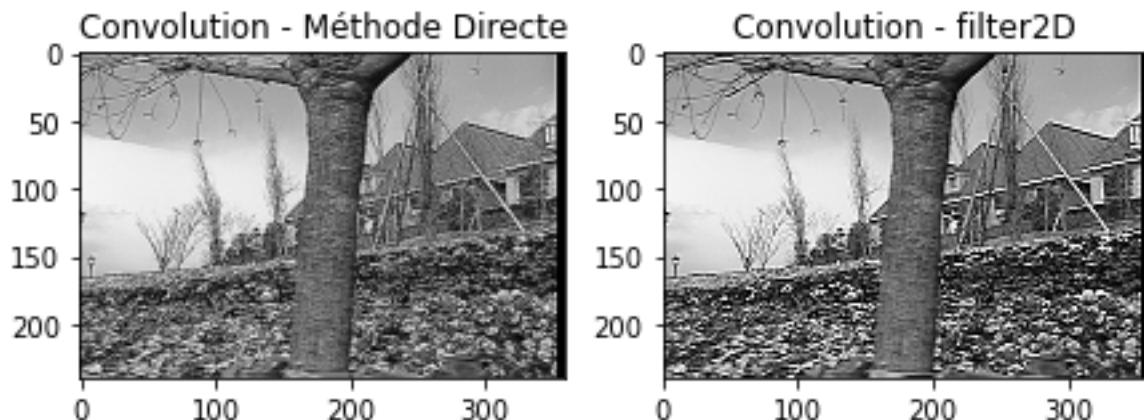


FIGURE 1 – Images filtrées

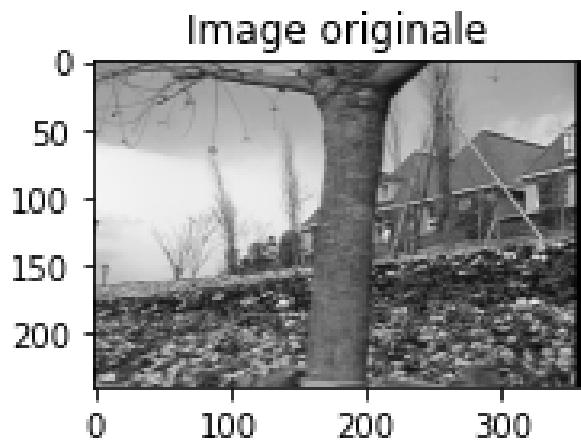


FIGURE 2 – Image originale

Ces deux filtres avec le noyau  $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$  représentent des filtres passe-haut, qui favorisent les hautes fréquences spatiales, comme les détails, et de ce fait, ils améliorent le contraste.

On remarque que la convolution par la méthode directe est moins nette que la convolution par filtre 2D.

#### Fonctions de OpenCV :

- Fonction de OpenCV utilisée pour la lecture de l'image : `cv2.imread(image)` : Cette fonction permet de convertir une image en niveau de gris en une matrice 2D. Chaque élément de la matrice  $I[x,y]$  représente un pixel de l'image et sa valeur représente le niveau de gris qui est entre 0 et 255.

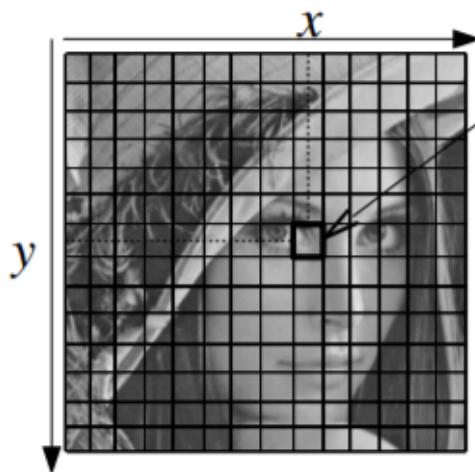


FIGURE 3 –

- La fonction `cv2.copyMakeBorder()` permet de copier une image mais en laissant l'utilisateur choisir l'épaisseur de la bordure. La méthode `BORDER_CONSTANT` applique

un remplissage d'une valeur constante pour toute la bordure, et dans la méthode *BORDER\_REPLICATE*, la bordure sera répliquée à partir des valeurs de pixels sur les bords de l'image d'origine.

- La méthode `plt.imshow()` permet de visualiser une matrice 2D en une image.

## Question 2

Le réhaussement de contraste permet de diminuer l'étendue de la zone de transition sans affecter l'intensité moyenne des régions situées de part et d'autre de cette transition.

Il permet aussi de limiter le risque de fusions intempestives de régions distinctes lors de la phase de segmentation en réduisant ainsi le bruit dans les zones stationnaires, et éviter les phénomènes de dépassement.

Le noyau du filtre de convolution se décompose selon cette formule :

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

La matrice  $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$  représente une approximation linéaire du Laplacien 4-connexe.

D'autre part, l'opération de réhaussement de contraste réalise une soustraction à l'image initiale d'une proportion de son Laplacien  $Rf[x,y] = f[x,y] - \gamma \Delta f[x, y]$ .

Cette formule affirme que le filtre par convolution avec le noyau  $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$  réalise un réhaussement de contraste.

## Question 3 :

Les approximations des dérivées directionnelles se font par différences finies calculées par convolution avec ces noyaux (filtre de Sobel) :

- $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$  pour  $\frac{\partial I}{\partial x}$
- $\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$  pour  $\frac{\partial I}{\partial y}$

Au niveau du code, la convolution approximative selon  $\frac{\partial I}{\partial x}$  est implémentée dans le calcul directe par l'ajout de cette ligne de code :

```
val = -img[y-1, x-1] - 2*img[y, x-1] - img[y+1, x-1] + img[y-1, x+1] + 2*img[y, x+1]+
      img[y+1, x+1]
```

et dans la fonction `filter2d` de OpenCV par la modification du noyau :

```
kernel = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
```

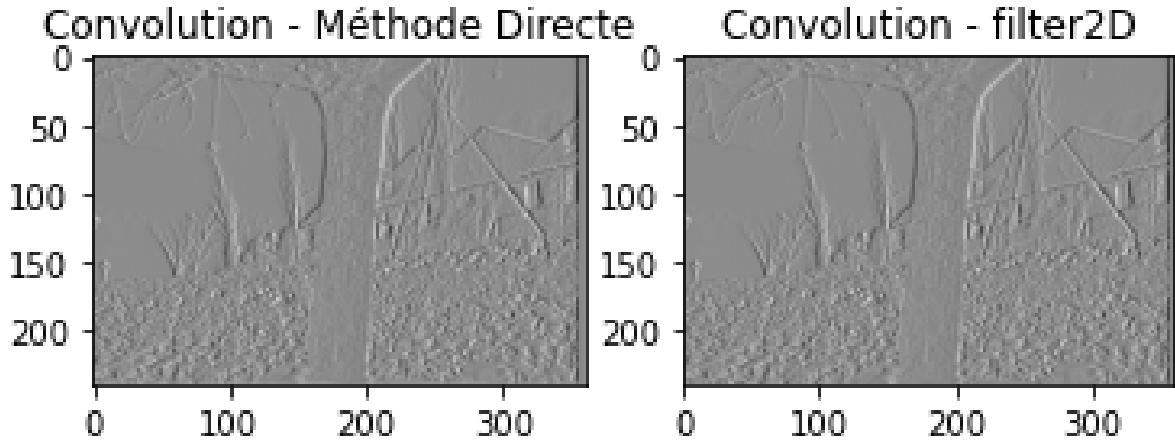


FIGURE 4 – Convolution approximative selon  $\frac{\partial I}{\partial x}$

Pour la convolution approximative selon  $\frac{\partial I}{\partial y}$ , on doit ajouter cette ligne pour le calcul direct

```
val = -img[y-1, x-1] - 2*img[y-1, x] - img[y-1, x+1] + img[y+1, x-1] + 2*img[y+1, x]+  
img[y+1, x+1]
```

et on doit modifier le noyau de la fonction filter2d :

```
kernel = np.array([[-1, -2, -1],[0, 0, 0],[1, 2, 1]])
```

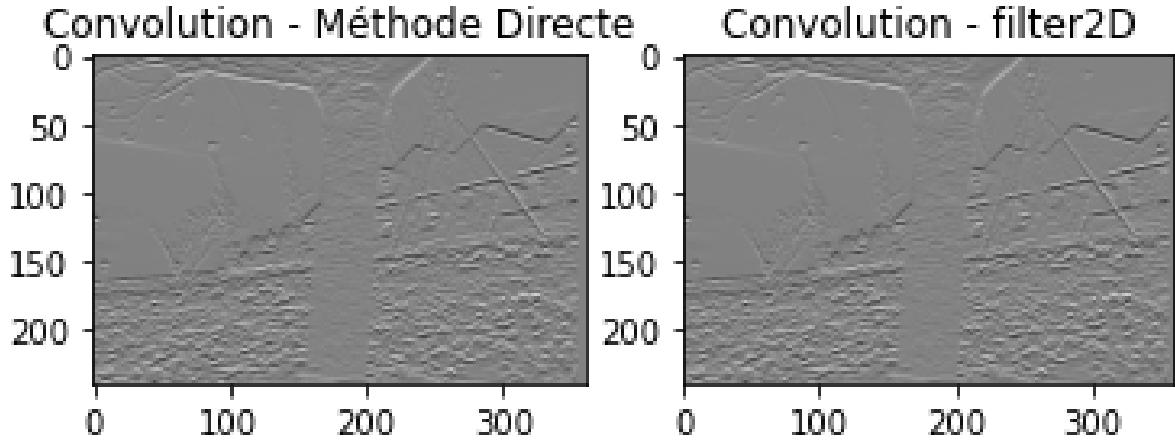


FIGURE 5 – Convolution approximative selon  $\frac{\partial I}{\partial y}$

Afin de pouvoir afficher le résultat, on doit remplacer cette ligne de code :

```
img2[y,x] = min(max(val,0),255) ==> img2[y,x] = val
```

Au niveau de l'affichage des matrices par Matplotlib, on doit supprimer les deux valeurs qui limitent le niveau de gris :

`plt.imshow(img3,cmap = 'gray',vmin = 0.0,vmax = 255.0)  $\Rightarrow$  plt.imshow(img3,cmap = 'gray')`

### Calcul de la norme euclidienne du gradient

La norme euclidienne du gradient est définie comme suit :

$$\|\nabla I\| = \sqrt{I_x^2 + I_y^2}$$

On doit donc additionner le carré des deux matrices obtenues après convolution approximative selon  $\frac{\partial I}{\partial x}$  et  $\frac{\partial I}{\partial y}$  :

*Méthode directe :*

```

1 for y in range(1,h-1):
2     for x in range(1,w-1):
3         #Convolution selon x
4         val = val = -img[y-1, x-1] - 2*img[y, x-1] - img[y+1, x-1] + img[y-1, x+1] + 2*img[y,
5             ↪ x+1]+ img[y+1, x+1]
6         #Convolution selon y
7         val1 = -img[y-1, x-1] - 2*img[y-1, x] - img[y-1, x+1] + img[y+1, x-1] + 2*img[y+1, x
8             ↪ ]+ img[y+1, x+1]
9         img2x[y,x] = val
10        img2y[y,x] = val1
11
12
13 img2_res=np.zeros((h, w))
14 img2_res=np.sqrt(img2x**2+img2y**2)

```

*Méthode filter2d*

```

1 #Convolution selon x
2 kernel = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
3 #Convolution selon y
4 kernel1 = np.array([[-1, -2, -1],[0, 0, 0],[1, 2, 1]])
5 img3x = cv2.filter2D(img,-1,kernel)
6 img3y = cv2.filter2D(img,-1,kernel1)
7
8 img3_res=np.zeros((h, w))
9 img3_res=np.sqrt(img3x**2+img3y**2)

```

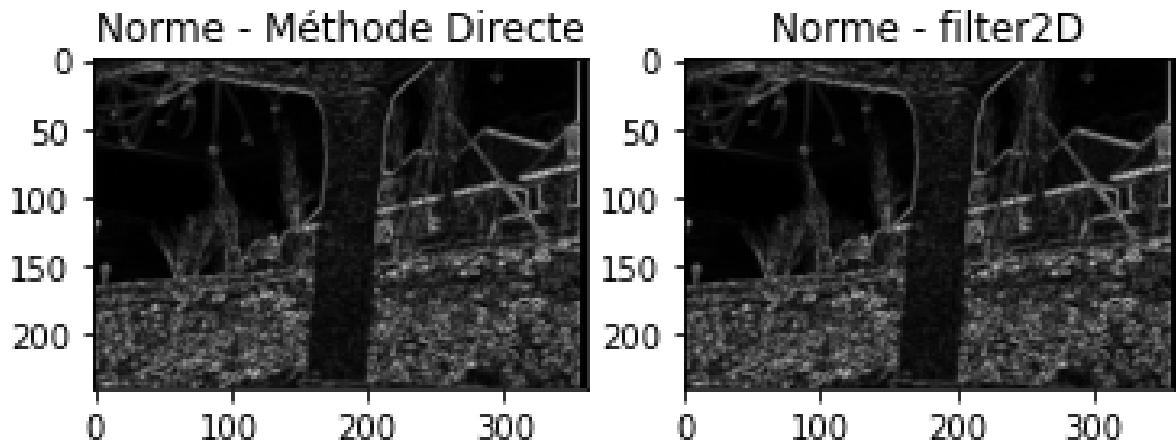


FIGURE 6 – Norme euclidienne du gradient

L'orientation du gradient est définie comme suit :

$$\arg(\nabla I) = \arctan\left(\frac{I_x}{I_y}\right)$$

Au niveau du code :

*Méthode directe*

```
1 #Calcul de l'orientataion mthode directe
2 orien_2 = cv2.phase(np.array(img2x, np.float32), np.array(img2y, dtype=np.float32),
   ↪ angleInDegrees=True)
```

*Méthode filter2d*

```
1 #Calcul de l'orientataion mthode filter2d
2 orien_3 = cv2.phase(np.array(img3x, np.float32), np.array(img3y, dtype=np.float32),
   ↪ angleInDegrees=True)
```

*Modification au niveau du code pour l'affichage*

Afin de bien visualiser le résultat, il faut définir des couleurs et attribuer à chaque orientation une couleur spécifique, on a attribué les couleurs suivants pour angles qui sont :

- 0 et 90° : rouge
- 90° et 180° : cyan
- 180° et 270° : vert
- 270° et 360° : jaune

```

1 image_map2 = np.zeros((orien_2.shape[0], orien_2.shape[1], 3), dtype=np.int16
2 # Define RGB colours
3 red = np.array([255, 0, 0])
4 cyan = np.array([0, 255, 255])
5 green = np.array([0, 255, 0])
6 yellow = np.array([255, 255, 0])
7
8 # Dfinir les couleurs correspondant aux angles
9
10 # Mthode directe
11 for i in range(0, image_map2.shape[0]):
12     for j in range(0, image_map2.shape[1]):
13         if orien_2[i][j] < 90.0:
14             image_map2[i, j, :] = red
15         elif orien_2[i][j] >= 90.0 and orien_2[i][j] < 180.0:
16             image_map2[i, j, :] = cyan
17         elif orien_2[i][j] >= 180.0 and orien_2[i][j] < 270.0:
18             image_map2[i, j, :] = green
19         elif orien_2[i][j] >= 270.0 and orien_2[i][j] < 360.0:
20             image_map2[i, j, :] = yellow
21
22 # Affichage de l'orientation du gradient
23 f, ax1 = plt.subplots(1, 1, figsize=(20,10))
24 plt.subplot(121)
25 plt.imshow(image_map2)
26 plt.title('Orientation du gradient – Mthode directe')
27
28 plt.subplot(122)
29 plt.imshow(image_map3)
30 plt.title('Orientataion du gradient – filter2D')
```

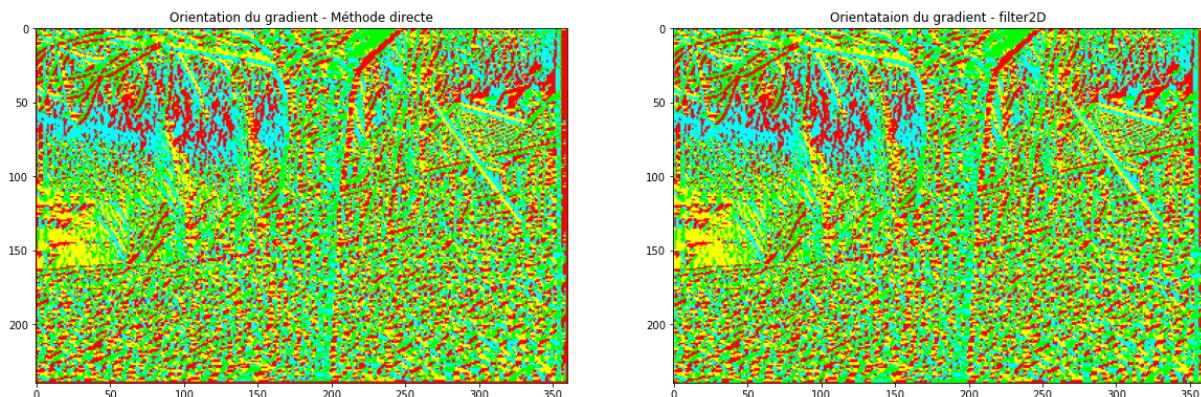


FIGURE 7 – Orientation du gradient

## Détecteurs

### Question 4

#### Implémentation de la fonction d'intérêt de Harris :

Pour calculer la fonction d'intérêt de Harris, on fait tout d'abords l'approximation des dérivées partielles  $I_x$  et  $I_y$  à partir du filtre Sobel en utilisant une convolution par rapport aux deux Kernels  $hx$  et  $hy$ . Ensuite, on calcule la matrice d'auto-corrélation  $C$  de l'image à partir de  $I_x * I_x$ ,  $I_y * I_y$ ,  $I_x * I_y$  et  $I_y * I_x$ . Finalement la fonction d'intérêt de Harris pour tout pixel  $(x, y)$  est :

$$\Theta(x, y) = \det(C) - \alpha * \text{trace}(C)^2$$

```
1 hx = np.array([[-1, 0, 1],  
2                 [-2, 0, 2],  
3                 [-1, 0, 1]])  
4 hy = np.array([[ -1, -2, -1],  
5                  [ 0, 0, 0],  
6                  [ 1, 2, 1]])  
7  
8 h_sum = np.array([[1, 1, 1],  
9                   [1, 1, 1],  
10                  [1, 1, 1]])  
11  
12 Ix = cv2.filter2D(Theta,-1,hx)  
13 Iy = cv2.filter2D(Theta,-1,hy)  
14  
15 Ixx = cv2.filter2D(Ix*Ix,-1,h_sum)  
16 Ixy = cv2.filter2D(Ix*Iy,-1,h_sum)  
17 Iyx = cv2.filter2D(Iy*Ix,-1,h_sum)  
18 Iyy = cv2.filter2D(Iy*Iy,-1,h_sum)  
19  
20 alpha = 0.04  
21 Theta = (Ixx*Iyy - Ixy*Iyx -alpha*(Ixx+Iyy)**2)
```

On obtient les figures suivantes pour la fonction d'intérêt de Harris et les points d'intérêt :

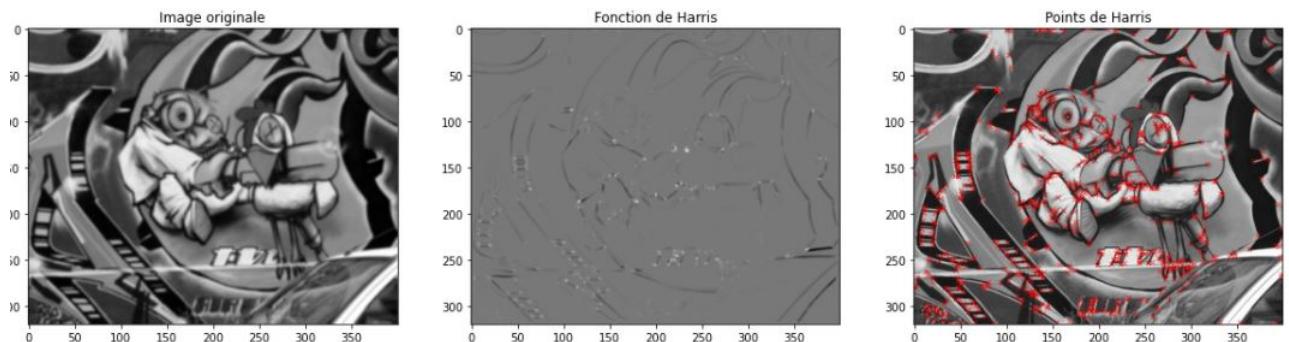


FIGURE 8 – La fonction de Harris et les points d'intérêt

*Question* : Comment le code fourni, qui utilise la dilatation morphologique, permet de calculer les maxima locaux de la fonction d'intérêt Theta ?

Les maxima locaux de la fonction d'intérêt sont calculés à partir de la dilatation de  $\Theta$  par rapport à  $se$  tel que :

```
1 se = np.array([[1, 1, 1],  
2     [1, 1, 1],  
3     [1, 1, 1]], dtype=uint8)
```

Cette opération permet d'augmenter le niveau de gris de chaque pixel ainsi que l'écart entre les maxima et les minima locaux de tel façon que la sélection des pixels dont le niveau de gris dépasse un seuil donné (dans notre cas 0.01) est plus facile. Ensuite, à partir de cette dilatation on supprime les maxima non locaux en annulant les valeurs des pixels qui sont inférieures que dans l'image dilatée :

```
1 Theta_maxloc[Theta < Theta_dil] = 0.0
```

## Question 5

**Effet de  $\alpha$  :** On fait varier  $\alpha$  pour une fenêtre de sommation de dimension égale à 3 et pour un seuil de décision = 0.01. On obtient les figures suivantes :

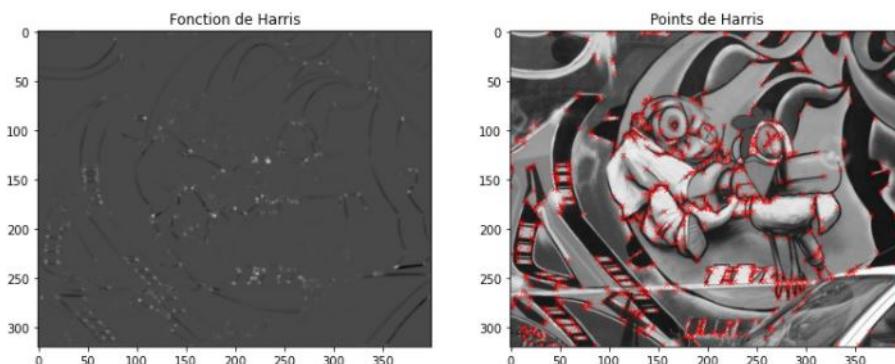


FIGURE 9 – La fonction de Harris et les points d'intérêt pour  $\alpha = 0.02$

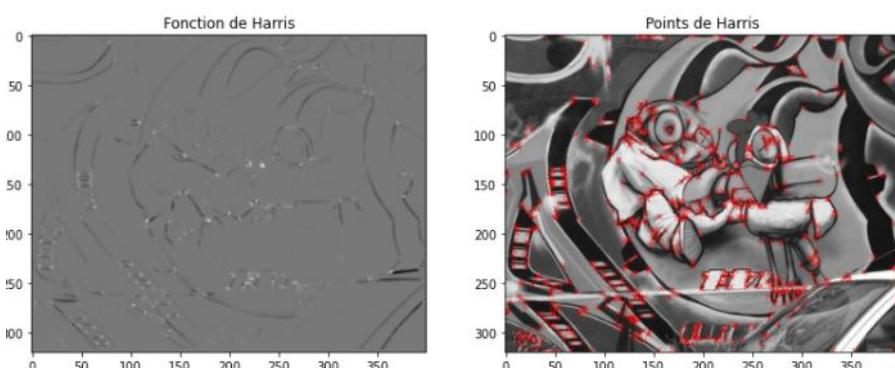


FIGURE 10 – La fonction de Harris et les points d'intérêt pour  $\alpha = 0.04$

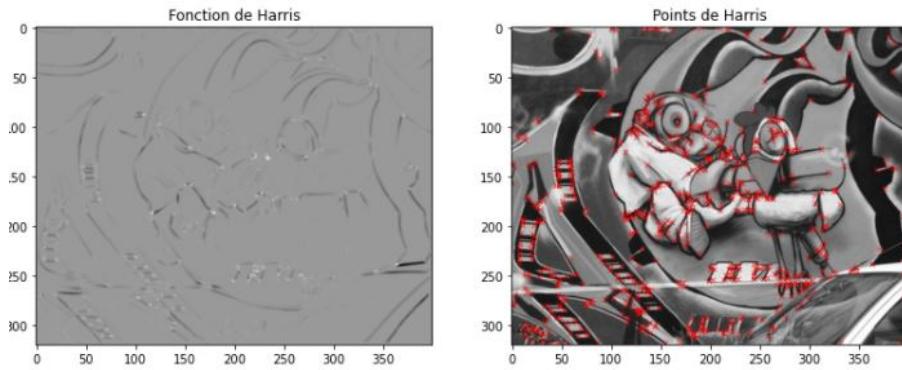


FIGURE 11 – La fonction de Harris et les points d'intérêt pour  $\alpha = 0.06$

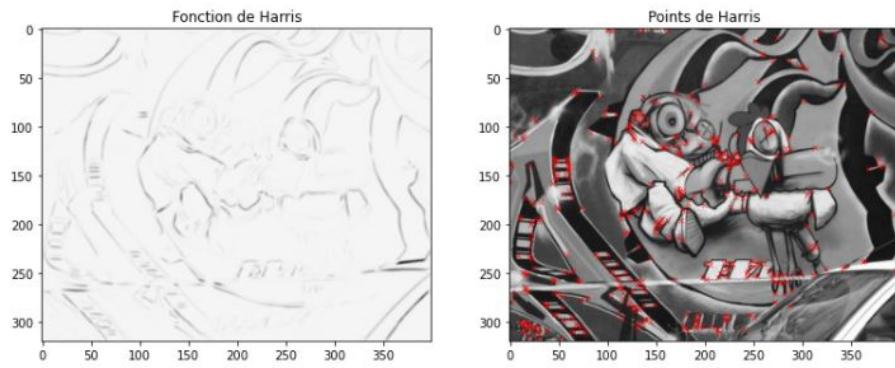


FIGURE 12 – La fonction de Harris et les points d'intérêt pour  $\alpha = 0.2$

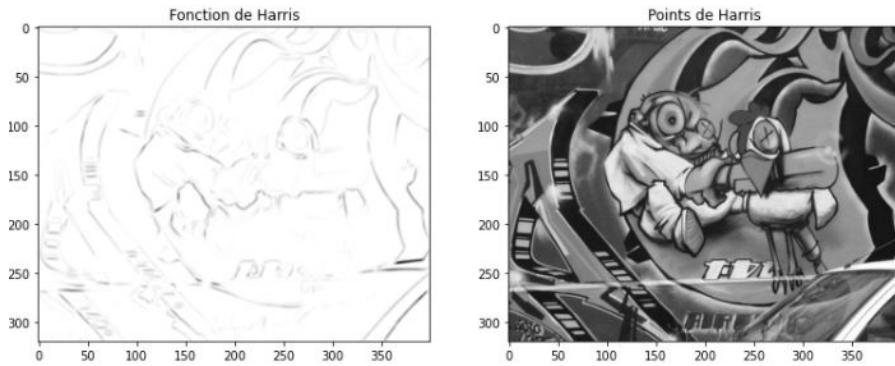


FIGURE 13 – La fonction de Harris et les points d'intérêt pour  $\alpha = 0.3$

Pour ces valeurs de  $\alpha$  on trouve que :

- Pour des valeurs de  $\alpha$  entre 0.02 et 0.06, la détection des points d'intérêt varie peu. Par contre, le contraste de la fonction de Harris diminue en augmentant  $\alpha$ .
- Pour des valeurs de  $\alpha$  entre 0.08 et 0.2, le nombre des coins détectés diminue : la détection devient moins précise.
- A partir de  $\alpha = 0.3$ , aucun point d'intérêt détecté.

*Conclusion :* Les meilleures valeurs de  $\alpha$  sont celles entre 0.04 et 0.06.

**Effet de la fenêtre de sommation :** On fait varier la fenêtre de sommation pour une valeur de  $\alpha$  égale à 0.06 et pour un seuil de décision = 0.01. On obtient les figures suivantes :

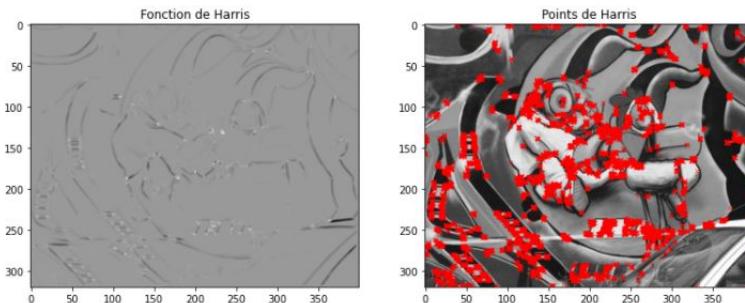


FIGURE 14 – La détection pour une fenêtre de sommation de dim = 1

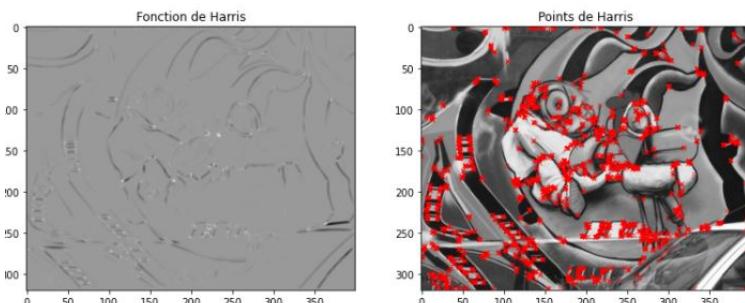


FIGURE 15 – La détection pour une fenêtre de sommation de dim = 2

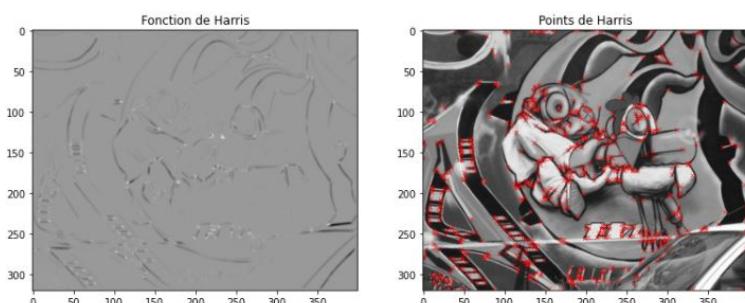


FIGURE 16 – La détection pour une fenêtre de sommation de dim = 3

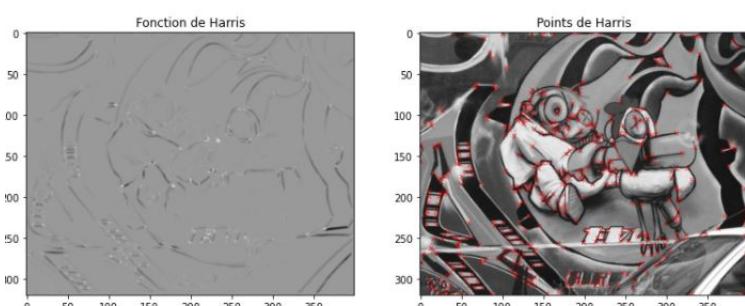


FIGURE 17 – La détection pour une fenêtre de sommation de dim = 6

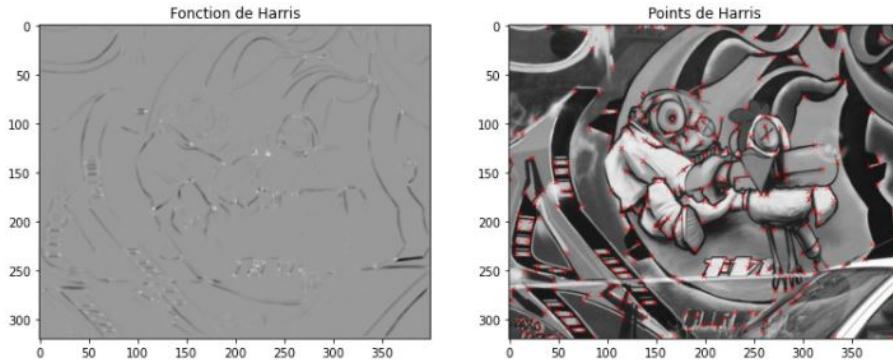


FIGURE 18 – La détection pour une fenêtre de sommation de dim = 10

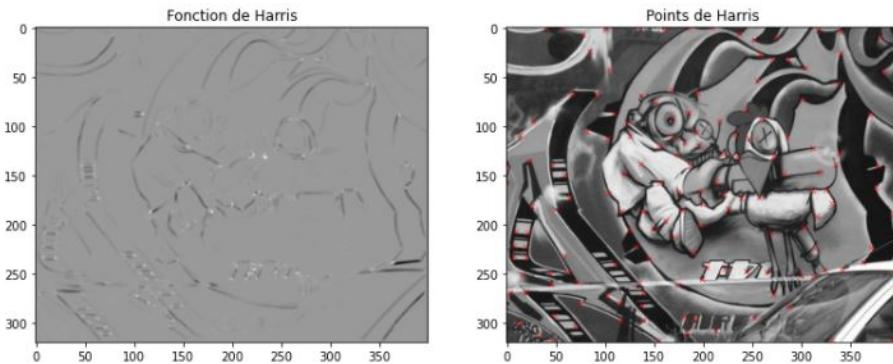


FIGURE 19 – La détection pour une fenêtre de sommation de dim = 20

*Conclusions :* En augmentant la dimension de la fenêtre de sommation le nombre des points d'intérêt détectés diminue. Pour des dimensions faibles (1 et 2), le nombre des caractéristiques détectées est grand et ne permet pas de caractériser correctement le contenu de l'image en entrée. Pour des dimensions de 3 à 8, le nombre des caractéristiques détectées est modéré et permet repérer les points les plus importants de l'image. Enfin, pour une dimension grande ( $> 10$ ), le nombre des points détectés devient faible et insuffisant pour caractériser l'image.

**Effet du seuil de décision :** On fait varier le seuil de décision pour une valeur de  $\alpha$  égale à 0.06 et pour une fenêtre de sommation d'une dimension égale à 3. On obtient les figures suivantes :

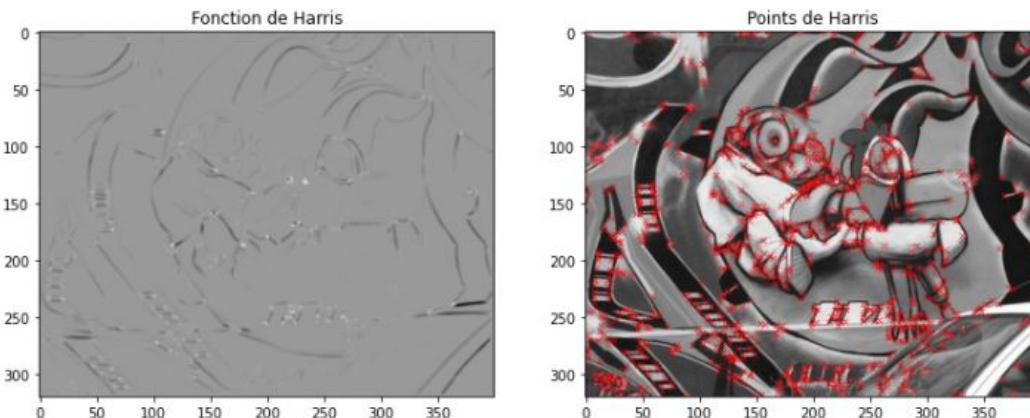


FIGURE 20 – La détection pour un seuil égal à 0.001

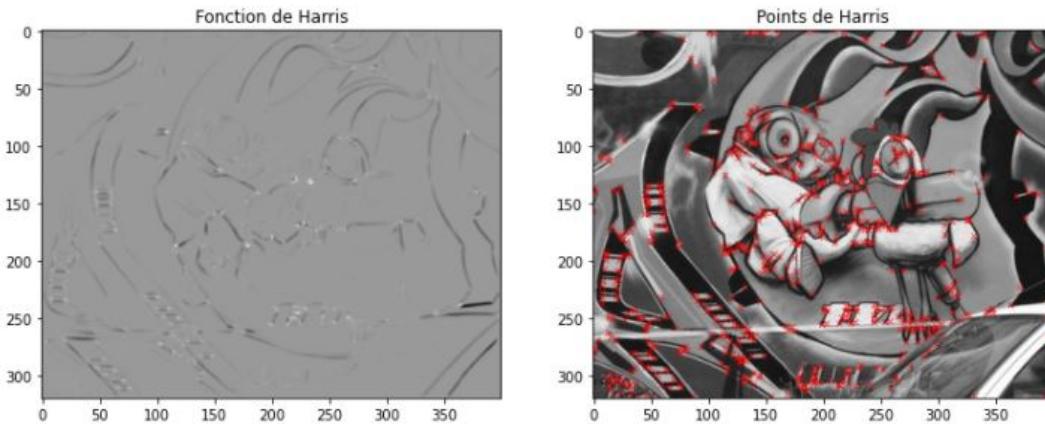


FIGURE 21 – La détection pour un seuil égal à 0.01

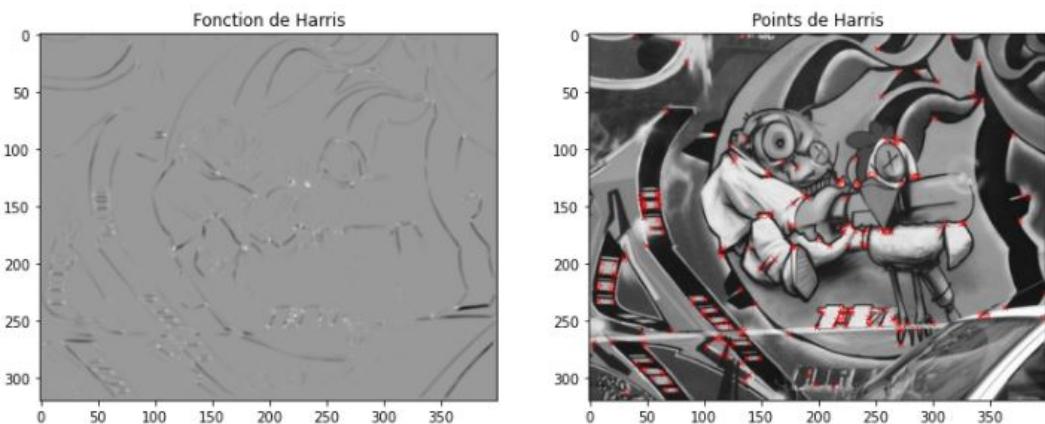


FIGURE 22 – La détection pour un seuil égal à 0.1

*Conclusions :* Il est clair que plus le seuil est grand plus le nombre des points d'intérêt détecté est petit.

*Question :* Comment peut-on réaliser ce calcul sur plusieurs échelles ?

Afin de déterminer les points d'intérêt sur plusieurs échelle, on les calcule à partir des dérivées partielles convoluées avec une dérivée de gaussienne d'écart-type  $\sigma$  (2, 3, 4, ..). Le code qui calcule la fonction d'intérêt de Harris devient alors :

```

1 hx = np.array([[-1, 0, 1],
2                 [-2, 0, 2],
3                 [-1, 0, 1]])
4 hy = np.array([[ -1, -2, -1],
5                   [ 0, 0, 0],
6                   [ 1, 2, 1]])
7
8 h_sum = np.array([[1, 1, 1],
9                   [1, 1, 1],
10                  [1, 1, 1]])
11
12 Ix = cv2.filter2D(Theta,-1,hx)

```

```

13 Iy = cv2.filter2D(Theta,-1,hy)
14
15 Ixx = cv2.filter2D(Ix*Ix,-1,h_sum)
16 Ixy = cv2.filter2D(Ix*Iy,-1,h_sum)
17 Iyx = cv2.filter2D(Iy*Ix,-1,h_sum)
18 Iyy = cv2.filter2D(Iy*Iy,-1,h_sum)
19
20 sigma = (3,3)
21 Ixx = cv2.GaussianBlur(Ixx,sigma,0)
22 Ixy = cv2.GaussianBlur(Ixy,sigma,0)
23 Iyx = cv2.GaussianBlur(Iyx,sigma,0)
24 Iyy = cv2.GaussianBlur(Iyy,sigma,0)
25
26 alpha = 0.06
27 Theta = (Ixx*Iyy - Ixy*Iyx -alpha*(Ixx+Iyy)**2)

```

On obtient alors les résultats suivants pour différentes échelles :

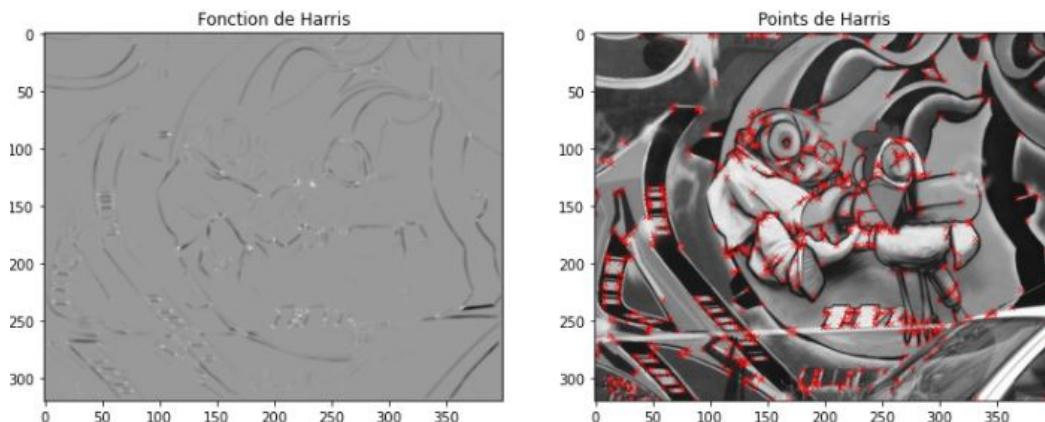


FIGURE 23 – La détection des points d'intérêt pour  $\sigma = 1$

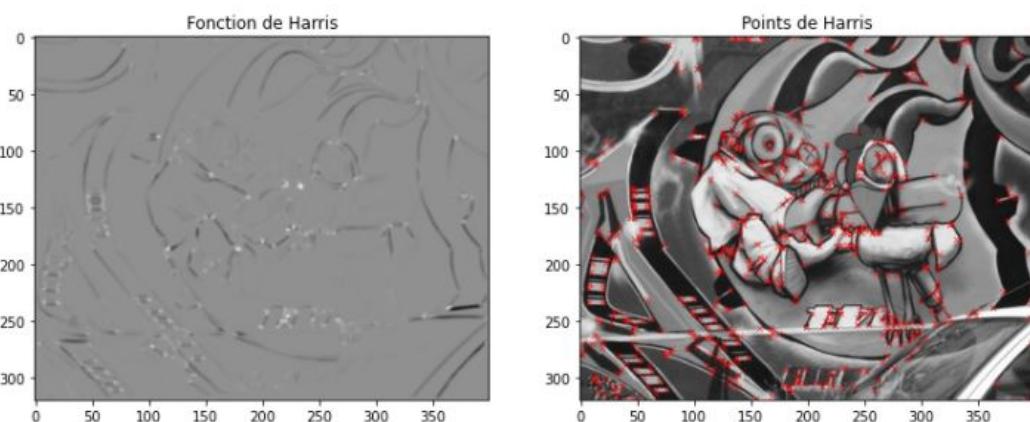


FIGURE 24 – La détection des points d'intérêt pour  $\sigma = 3$

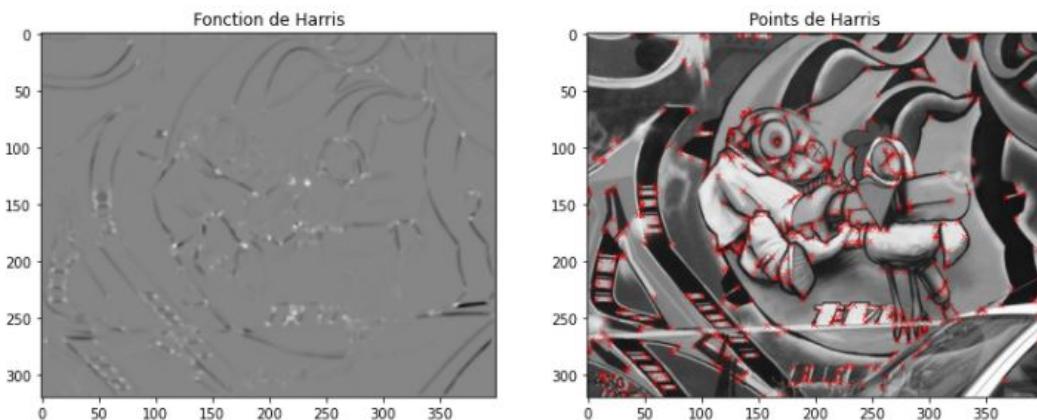


FIGURE 25 – La détection des points d'intérêt pour  $\sigma = 5$

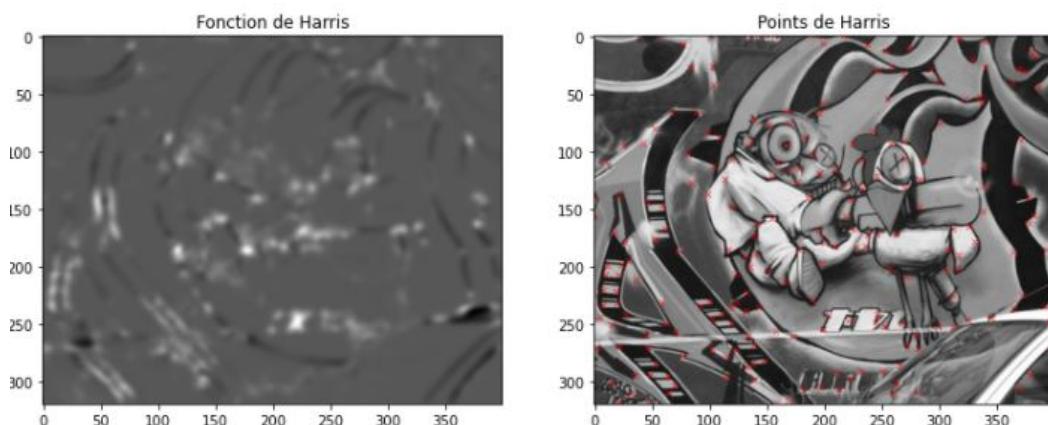


FIGURE 26 – La détection des points d'intérêt pour  $\sigma = 21$

*Conclusion :* On peut conclure que la détection à l'aide de la fonction de Harris à grandes échelles devient insuffisante.

*Question :* Comment étendre la notion de maxima locaux pour faire en sorte que deux points d'intérêt soient toujours distants d'au moins  $r$  pixels ?

Pour ce faire, on ne cherche pas les maxima locaux mais plutôt dans un ensemble plein de  $2r*2r$  pixels. C'est-à-dire :

La notion de maxima locaux consiste à chercher les points  $P$  auxquels la norme du gradient est maximale dans la direction locale du gradient :

Soit  $P$  un pixel et soient  $P_1$  et  $P_2$  les pixels situés de part et d'autre dans la direction  $\theta$  du gradient en  $P$ . On compare la norme du gradient en  $P$  avec celles du gradient en  $P_1$  et  $P_2$ . Si on trouve que cette norme en  $P$  est supérieure à celle en  $P_1$  et  $P_2$  on peut dire que  $P$  est un maximum local.

C'est la notion du maxima local. Maintenant si on veut que les points d'intérêt soient toujours distants d'au moins  $r$  pixels on, on les détermine non pas à partir de la comparaison de la norme du gradient par rapport aux deux pixels adjacents mais plutôt par rapport aux  $2r*2r$  pixels adjacents (centrés en  $P$ ).

## Question 6

Expérimentation du détecteur ORB :

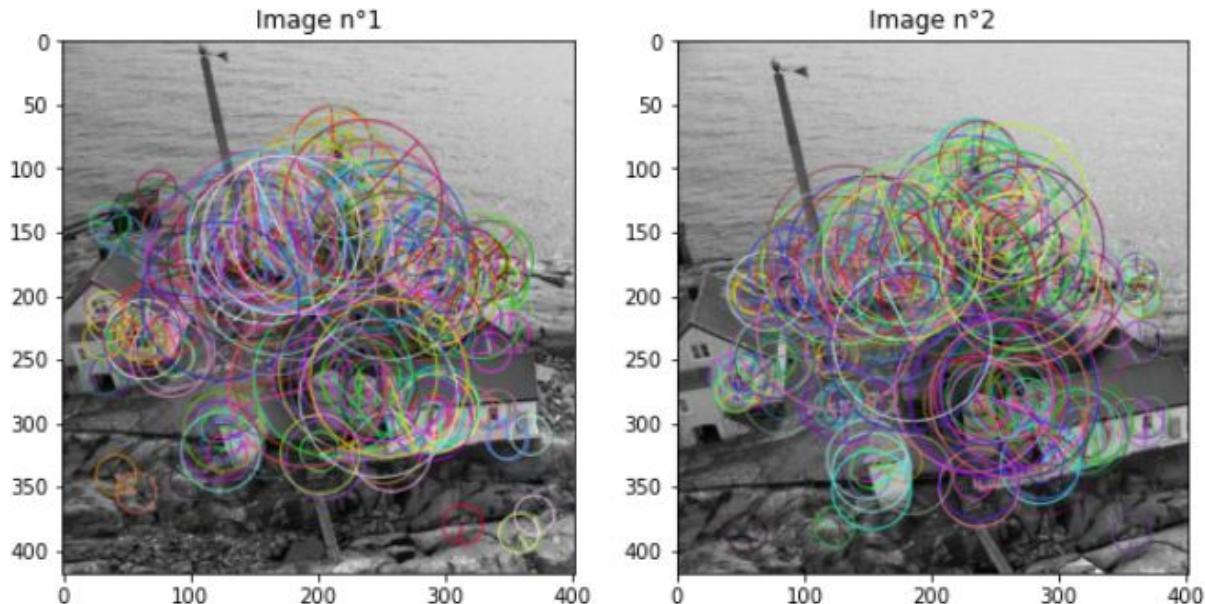


FIGURE 27 – La détection avec le détecteur ORB

**Principe du détecteur ORB :** Ce détecteur est une extension du détecteur FAST. Ses étapes sont :

- Extraction des caractéristiques à l'aide de FAST. Le détecteur FAST est calculée à plusieurs résolutions, donc chaque pixel de l'image possède une échelle caractéristique.
- On calcule ensuite pour chaque point d'intérêt P de FAST le centre de gravité O de l'imagette carrée qui circonscrit le cercle FAST
- Enfin, la direction (OP) est utilisée comme orientation caractéristique du point P.

Ainsi, les cercles dessinés dans les images de la figures 27 représentent les cercles FAST multi-échelles et les lignes dessinées à partir des centres de ces cercles sont les orientations caractéristiques de chaque point d'intérêt.

**Les paramètres du détecteur ORB :** Les paramètres de ce détecteur sont :

- nfeatures : le nombre maximal des points d'intérêt à considérer.
- scaleFactor : le ratio décimal de la pyramide utilisée pour la détection.
- nlevels : le nombre des niveaux de la pyramide utilisée pour la détection.

On constate que l'augmentation de ces paramètres entraîne les effets suivants sur la détection :

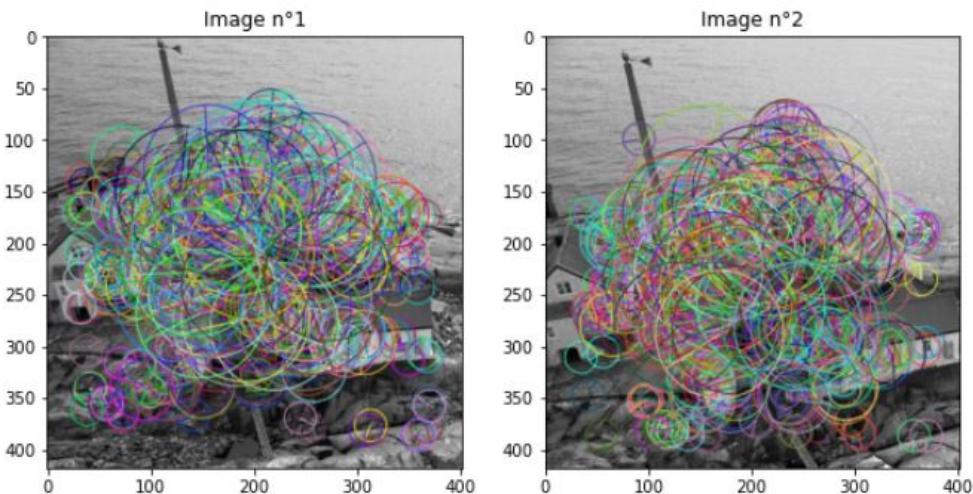


FIGURE 28 – La détection avec ORB pour nfeatures = 500, scaleFactor = 2 et nlevels = 3

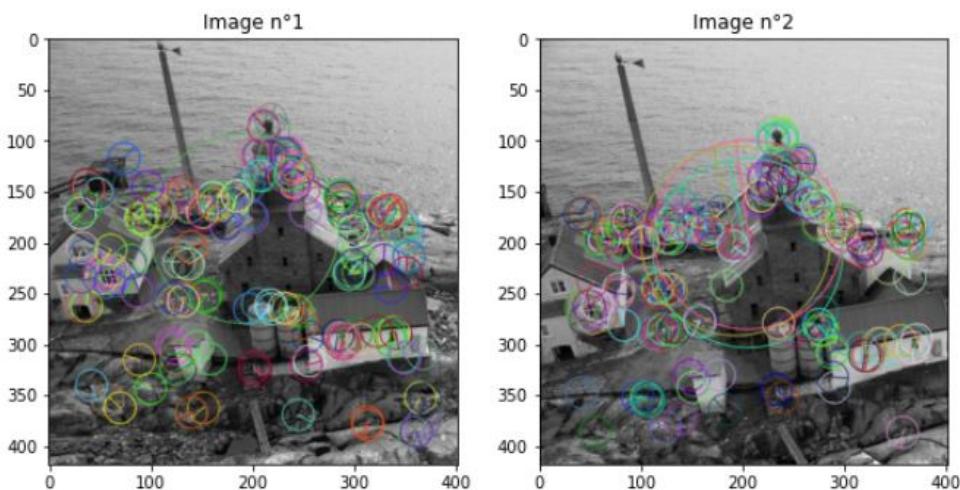


FIGURE 29 – La détection avec ORB pour nfeatures = 250, scaleFactor = 6 et nlevels = 3

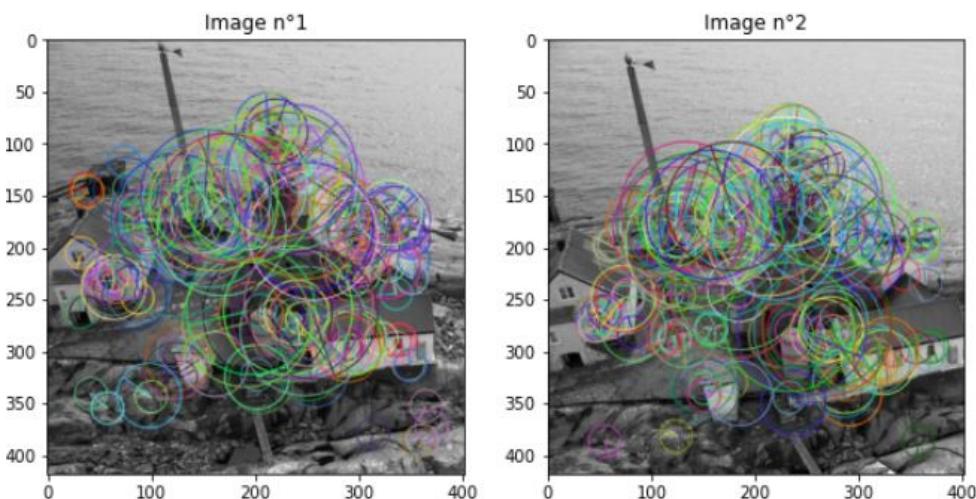


FIGURE 30 – La détection avec ORB pour nfeatures = 250, scaleFactor = 2 et nlevels = 10

### Expérimentation du détecteur KAZE :

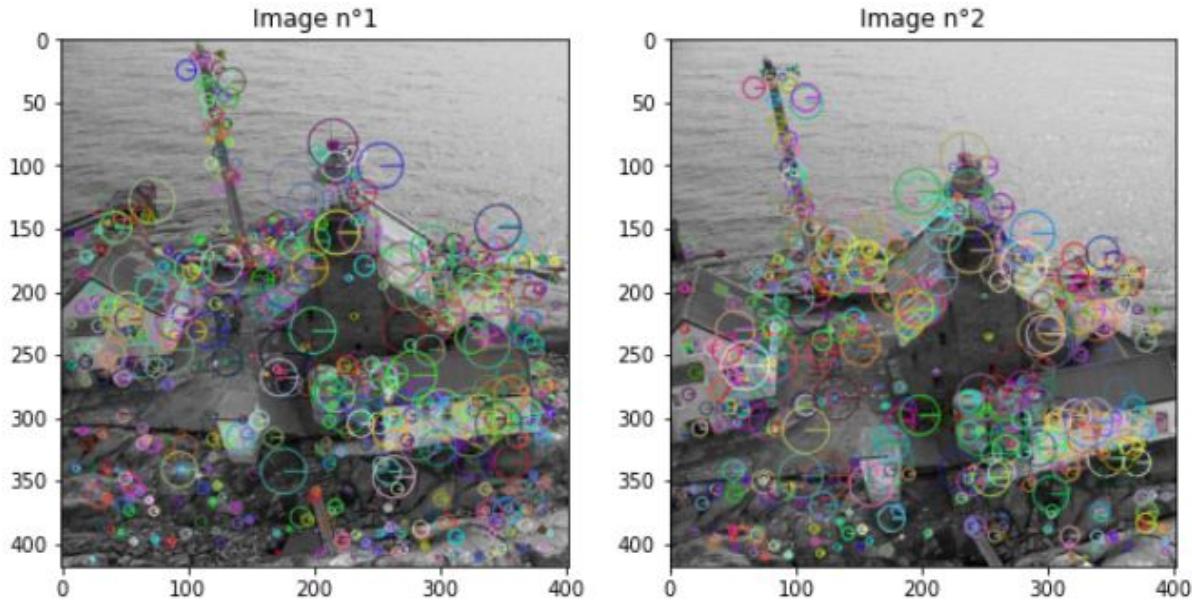


FIGURE 31 – La détection avec le détecteur KAZE

**Principe du détecteur KAZE :** Le descripteur KAZE permet de détecter et de décrire des caractéristiques 2D dans un extrema d'espace d'échelle non linéaire afin d'obtenir une meilleure précision de localisation.

Cet algorithme utilise un filtrage de diffusion non linéaire combiné à une fonction de conductivité. Le but est d'obtenir des fonctionnalités qui ont une répétabilité et un caractère distinctif plus élevés que SIFT.

L'image convoluée avec une gaussienne est donc la solution de l'équation de la chaleur (dans le cas où le facteur de conductance  $c$  est constant) :

$$\frac{\partial I}{\partial t} = \text{div}(c(x, y, t) \cdot \nabla I) \quad (3)$$

Ensuite, les points d'intérêt sont déterminés en calculant les maxima locaux du déterminant de la Hessienne.

**Les paramètres du détecteur KAZE :** Les paramètres de ce détecteur sont :

- upright
- threshold
- nOctaves
- nOctaveLayers
- diffusivity

On constate que l'augmentation de ces paramètres entraîne les effets suivants sur la détection :

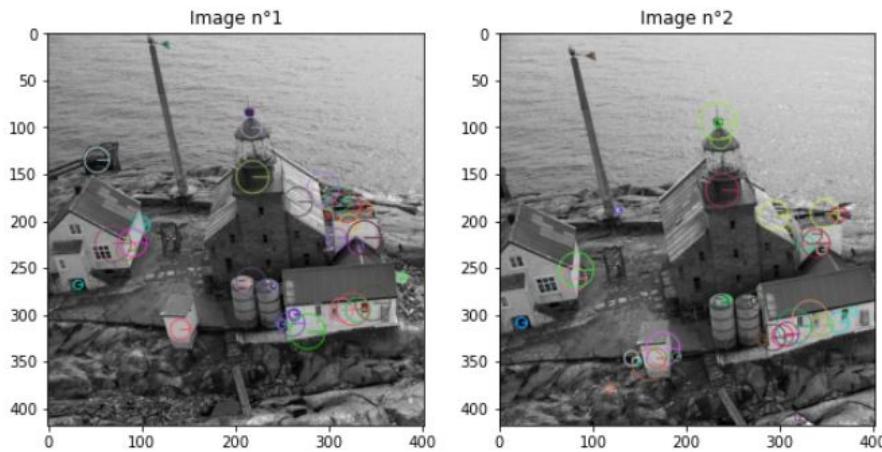


FIGURE 32 – La détection avec KAZE pour upright = False, threshold = 0.01 et nOctaves = 4, nOctaveLayers = 4 et diffusivity = 2

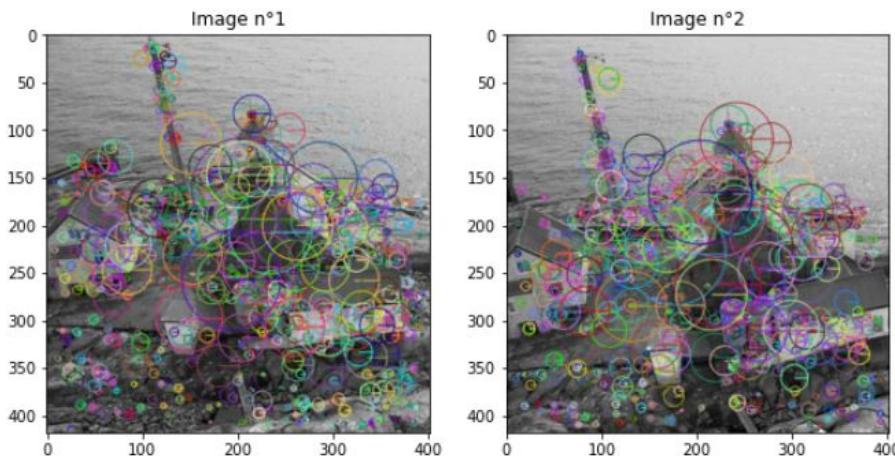


FIGURE 33 – La détection avec KAZE pour upright = False, threshold = 0.001 et nOctaves = 8, nOctaveLayers = 4 et diffusivity = 2

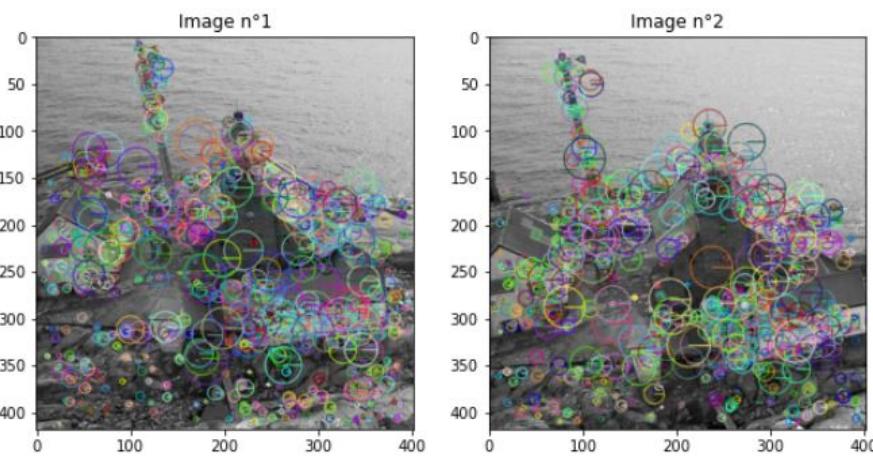


FIGURE 34 – La détection avec KAZE pour upright = False, threshold = 0.001 et nOctaves = 4, nOctaveLayers = 8 et diffusivity = 2

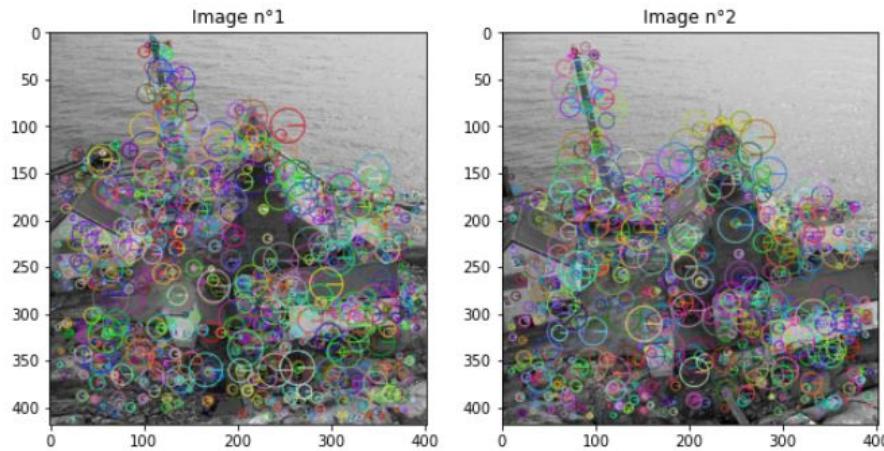


FIGURE 35 – La détection avec KAZE pour upright = False, threshold = 0.001 et nOctaves = 4, nOctaveLayers = 4 et diffusivity = 4

#### Question :

Comment peut-on visuellement évaluer la répétabilité de chaque détecteur appliqué sur une paire d’images ?

Pour pouvoir évaluer la répétabilité de chaque détecteur, on devra voir si l’on applique le détecteur à une image et à sa rotation d’un angle  $\theta$  (par exemple), l’emplacement des points d’intérêt reste le même dans les deux images (c’est-à-dire occupe le même pixel dans les deux images).

## Descripteurs et Appariement

### Question 7

#### Descripteurs attachés au points ORB

ORB est essentiellement une fusion du détecteur de point-clé FAST et du descripteur BRIEF avec de nombreuses modifications pour améliorer les performances. D’abord, il utilise FAST pour trouver des points clés, puis applique la mesure de coin de Harris pour trouver les N premiers points parmi eux. Il utilise également une pyramide pour produire des entités multi-échelles.

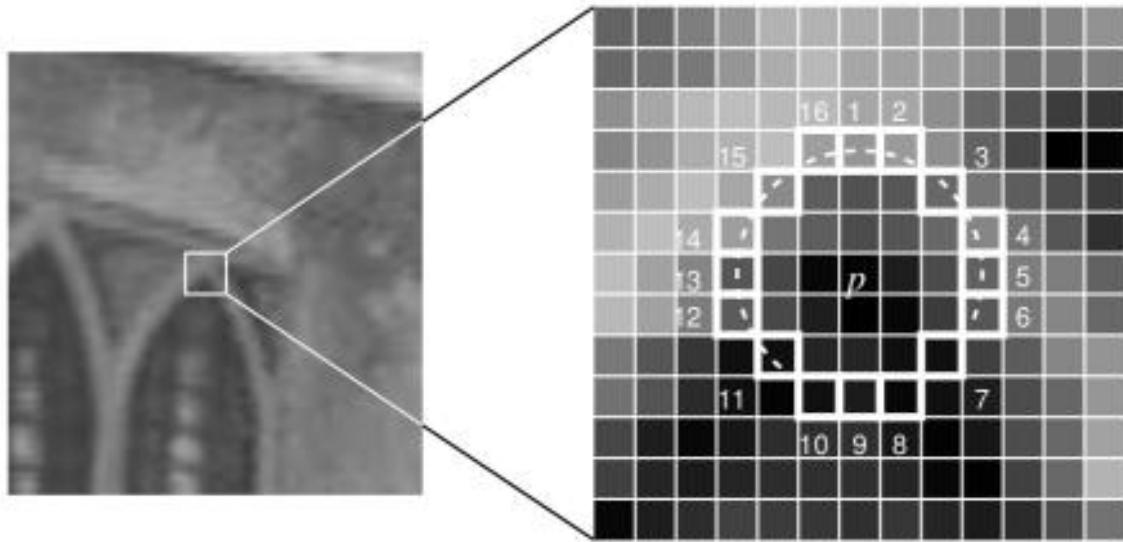


FIGURE 36 – Principe du détecteur ORB

### Descripteurs attachés au points KAZE

Le descripteur KAZE permet de détecter et de décrire des caractéristiques 2D dans un extrema d'espace d'échelle non linéaire afin d'obtenir une meilleure précision de localisation.

Cet algorithme utilise un filtrage de diffusion non linéaire combiné à une fonction de conductivité. Le but est d'obtenir des fonctionnalités qui ont une répétabilité et un caractère distinctif plus élevés que SIFT.

L'image convoluée avec une gaussienne et elle est donc la solution de l'équation de la chaleur (dans le cas où le facteur de conductance  $c$  est constant) :

$$\frac{\partial I}{\partial t} = \operatorname{div}(c(x, y, t) \cdot \nabla I) \quad (3)$$

### Propriétés des détecteurs qui permettent de rendre l'appariement invariant par changement d'échelles et par rotation :

- ORB : Il calcule le centre de gravité avec le coin situé au centre. La direction du vecteur de ce point d'angle au centre de gravité donne l'orientation. L'invariance par rotation est obtenue par le calcul des moments avec  $x$  et  $y$  qui devraient être dans une région circulaire de rayon  $r$ .
- KAZE : Le détecteur KAZE est basé sur un déterminant normalisé à l'échelle de la matrice Hessienne qui est calculée à plusieurs niveaux d'échelle. Les maximums de réponse du détecteur sont captés sous forme de points caractéristiques. La description de fonction introduit la propriété d'invariance de rotation en trouvant une orientation dominante dans un voisinage circulaire autour de chaque entité détectée, ainsi, les caractéristiques de KAZE sont invariantes à la rotation et à l'échelle.

**Propriétés des descripteurs qui permettent de rendre l'appariement invariant par changement d'échelles et par rotation :**

Invariants différentiels :

Il consiste à représenter les points d'intérêt par des indices qui soient invariants par rotation et par changement d'échelle. Le principe utilisé ici est basé sur l'utilisation des dérivées spatiales multi-échelle :

$$L_{i,j}^{\sigma} = I * G_{i,j}^{\sigma}$$

- ORB : Pour les descripteurs, ORB utilise des descripteurs BRIEF mais avec modification. ORB fait diriger le BRIEF en fonction de l'orientation des points clés. Pour tout ensemble de fonctionnalités de  $n$  tests binaires à l'emplacement  $(x_i, y_i)$ , on définit une matrice  $S$   $2 * n$  contenant les coordonnées de ces pixels. Ensuite, on trouve sa matrice de rotation  $\theta$  pour obtenir une version tournée.  $S_{\theta}$ .
- KAZE : Les descripteurs KAZE utilisent la même méthode que la méthode SIFT qui sont des histogrammes des orientations locales autour du point d'intérêt. Cette caractéristique le rend invariant par rotation.

## Question 8

Dans cette question, on va analyser des différentes méthodes d'appariement des points d'intérêts. On va comparer les performances des trois stratégies *Features\_Match\_CrossCheck.py*, *Features\_Match\_RatioTest.py* et *Features\_Match\_FLANN.py*

### Méthode d'appariement *Features\_Match\_CrossCheck.py*

Dans cette méthode, on a utilisé Brute-Force matcher. Il prend le descripteur d'une entité dans le premier ensemble et la met en correspondance avec toutes les autres entités dans le second ensemble en utilisant un calcul de distance. Le point le plus proche est retourné.

Ce matcher prend en paramètres la norme qu'on va utiliser pour le calcul des distances et une valeur booléenne *crossCheck*. Si *crossCheck=True*, il renvoie seulement les correspondances avec la valeur  $(i, j)$  telle que le i-ème descripteur de l'ensemble A a le j-ème descripteur de l'ensemble B comme meilleure correspondance et vice-versa. Autrement dit, les deux fonctionnalités des deux ensembles doivent correspondre. Si *crossCheck=False*, il va retourner les k meilleures correspondances où k est spécifié par l'utilisateur.

***BFMatcher(int normType=NORM\_L2, bool crossCheck=false)***

### *CrossCheck avec le descripteur ORB*

On a visualisé avec les deux descripteurs ORB et KAZE, les 200 meilleurs appariements. La différence entre ces deux descripteurs c'est que les meilleurs appariements ne sont pas les mêmes.

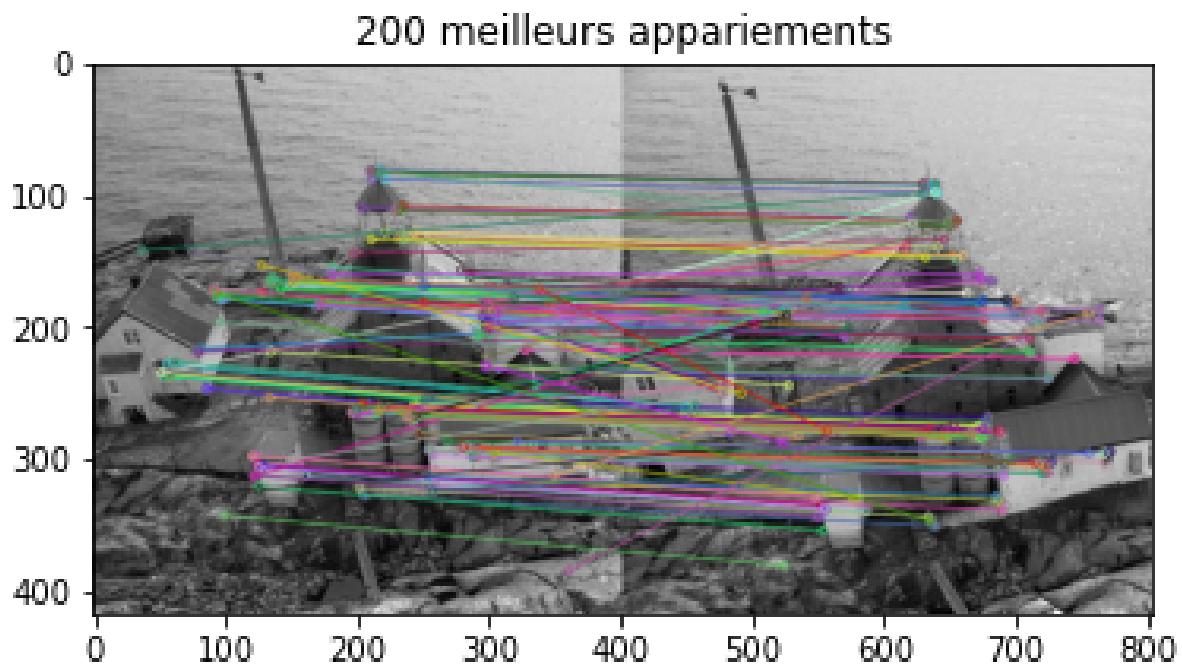


FIGURE 37 – CrossCheck avec ORB

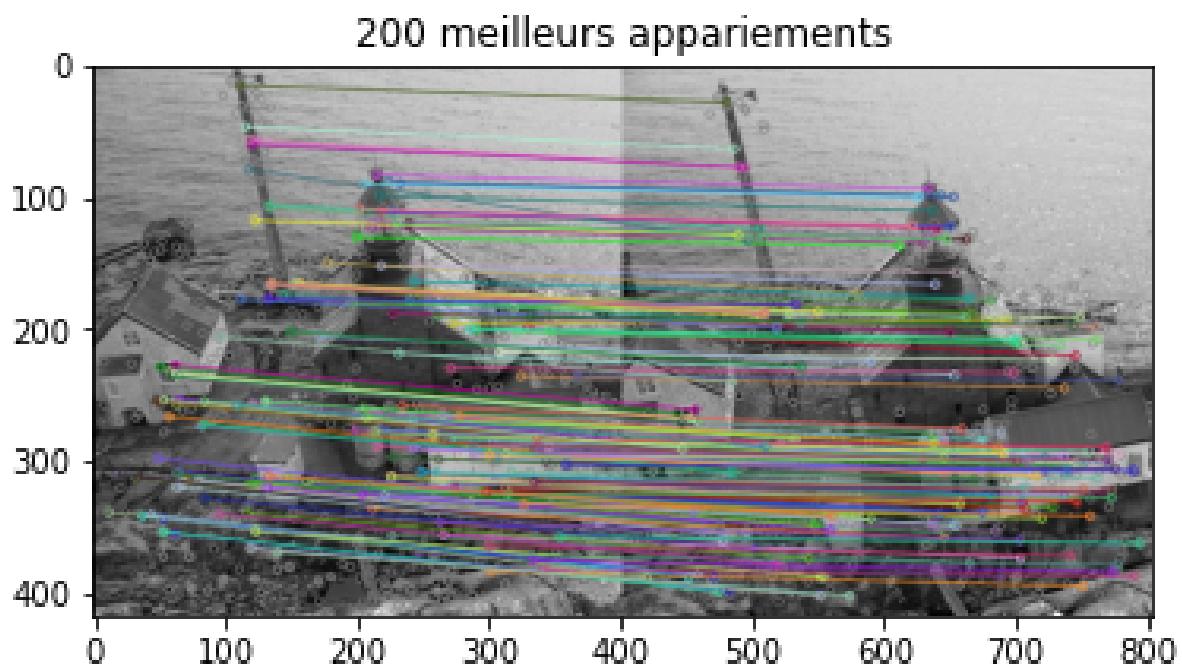


FIGURE 38 – CrossCheck avec KAZE

**Méthode d'appariement *Features\_Match\_RatioTest.py***

Cette méthode est basé aussi sur le BFMatcher mais en renvoyant les k meilleurs appariements (on va tester k=2 pour notre exemple). On remarque que le descripteur KAZE donne un résultat meilleur que ORB. On observe alors 175 appariements pour KAZE contre 64 pour ORB.

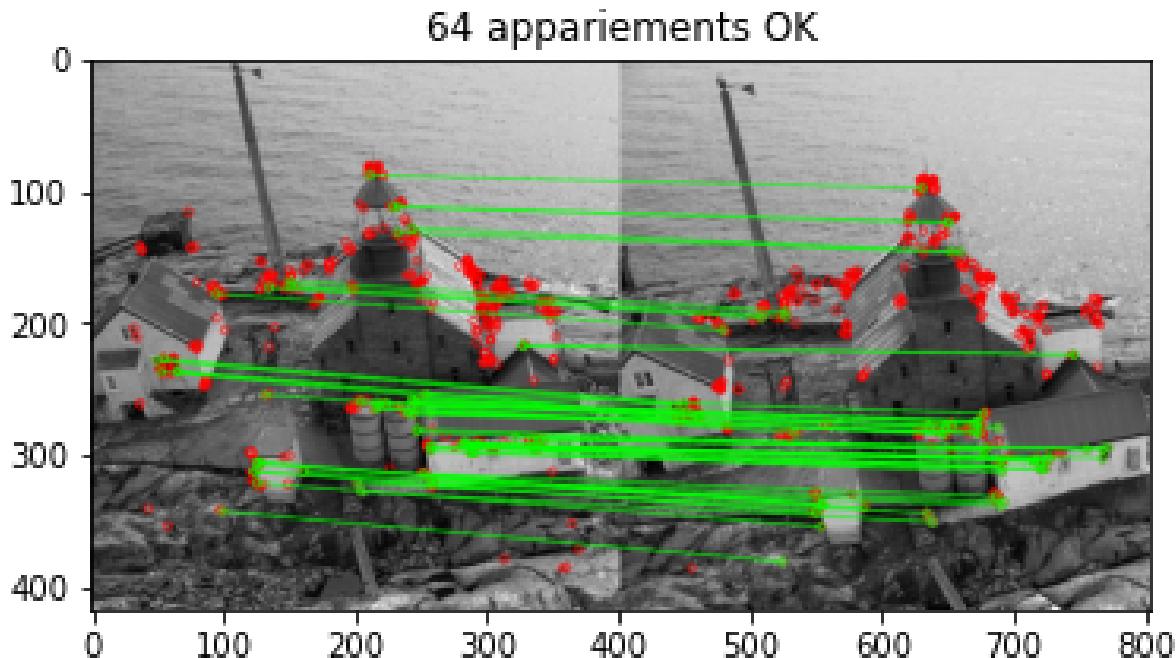


FIGURE 39 – RatioTest avec ORB

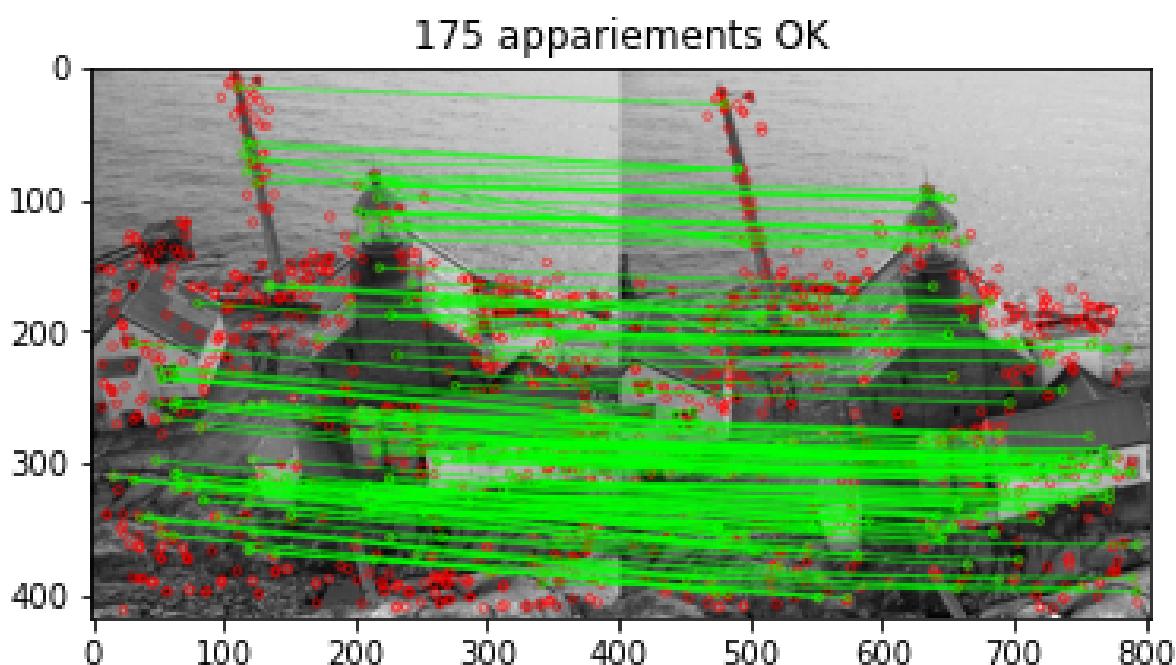


FIGURE 40 – RatioTest avec KAZE

### Méthode d'appariement *Features\_Match\_FLANN.py*

FLANN : Fast Library for Approximate Nearest Neighbours, contient une collection d'algorithmes optimisés pour la recherche rapide des plus proches voisins dans de grands ensembles de données et pour des entités de grande dimension. Il fonctionne plus rapidement que BFMatcher pour les grands ensembles de données.

Pour le matcher basé sur FLANN, nous devons passer deux dictionnaires qui spécifient l'algorithme à utiliser.

Avec le(descripteur ORB, on doit changer le *index\_params* pour que le code fonctionne :

```
1 # on doit changer index_params avec ORB  
2 index_params = dict(algorithm=6,table_number=6,key_size=12,multi_probe_level=2)
```

Comparaison des performances : On remarque que le(descripteur KAZE donne un résultat meilleur que ORB. On observe alors 175 appariements pour KAZE contre 66 pour ORB.

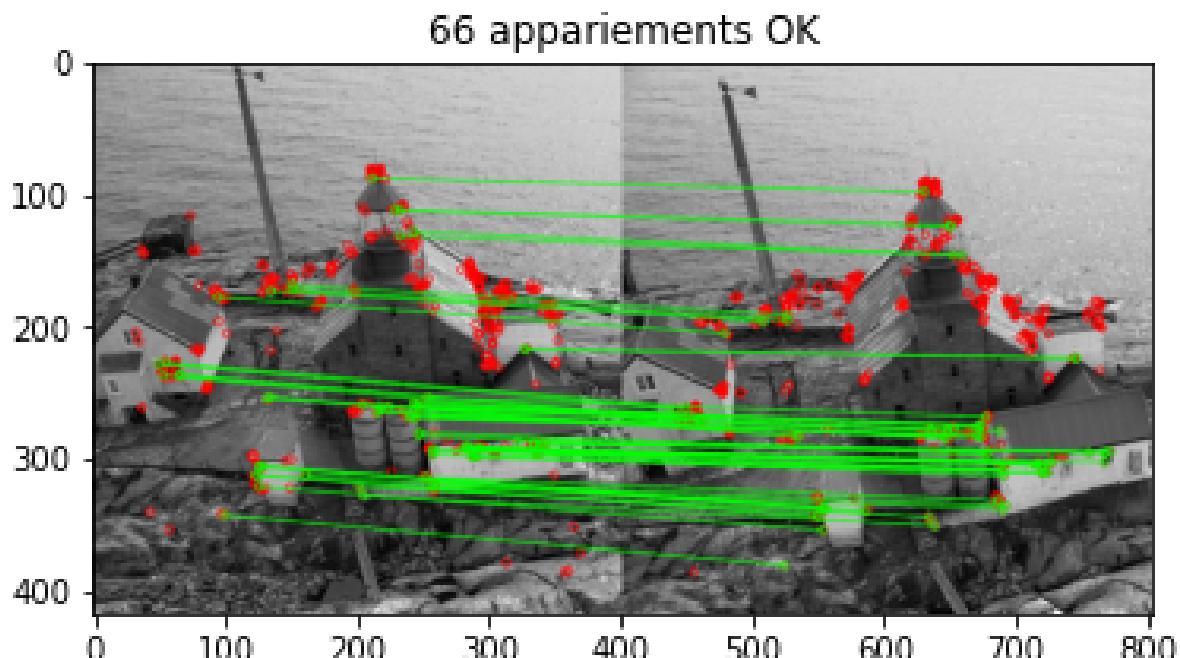


FIGURE 41 – FLANN avec ORB

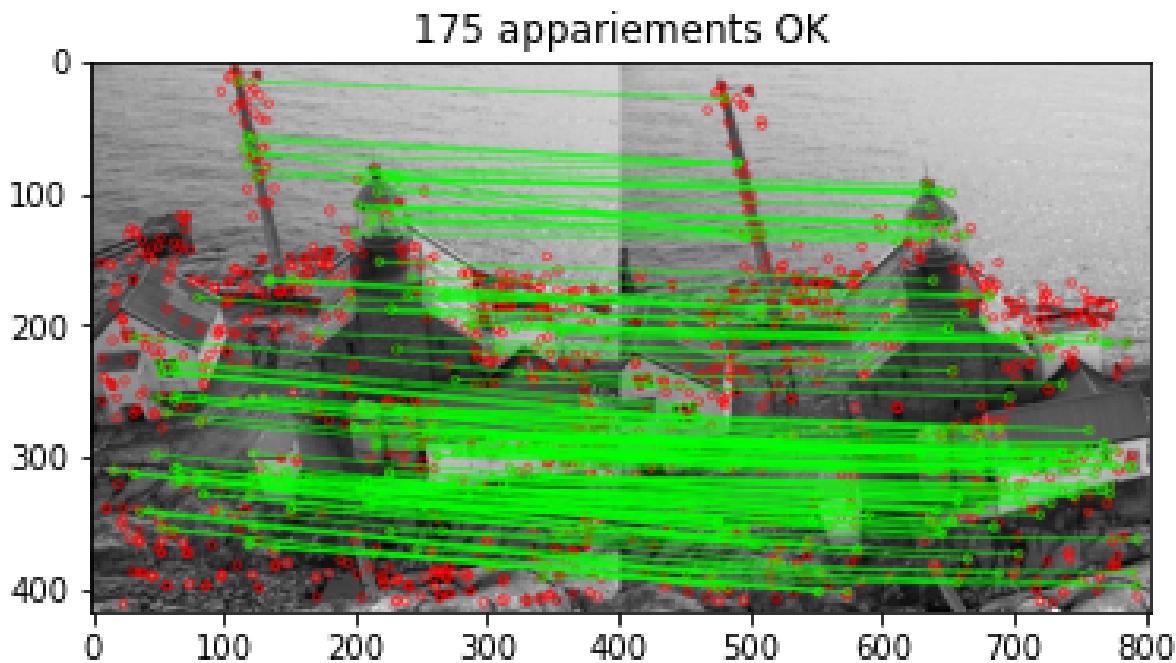


FIGURE 42 – FLANN avec KAZE

**Différence de distance :**

On remarque dans le code le descripteur ORB utilise la distance *Hamming* alors que KAZE utilise la norme L2

En fait, les descripteurs ORB sont des vecteurs de valeurs binaires. Si on applique la distance euclidienne à des vecteurs binaires, le résultat au carré d'une seule comparaison serait toujours 1 ou 0, ce qui n'est pas informatif lorsqu'il s'agit d'estimer la différence entre les éléments. La distance euclidienne globale serait la racine carrée de la somme de ces uns et des zéros, encore une fois ce n'est pas un bon estimateur de la différence entre les vecteurs.

De ce fait, la distance de Hamming est utilisée. Par contre pour le cas des descripteur KAZE, la distance L2 est bonne vu que qu'on est dans le cas classique.

**Question : Expliquer pourquoi la stratégie FLANN ne fonctionne pas bien avec les points ORB :**

La stratégie FLANN ne donne pas de bonne résultats pour le descripteur ORB. Ceci s'explique par le fait que cette stratégie utilise la méthode des plus proches voisins *knnMatch* pour trouver le meilleur appariement. (identique pour le BF avec RatioTest). En fait, dans cette méthode, on compare la distance du premier point avec la distance du deuxième point mais multiplié par 0.7. C'est à dire, on ne fait pas de tri des points mais seulement avec les deux points trouvés par l'algorithme du knn. Ceci ne donne pas de meilleur résultat pour ORB.

## Question 9

### Évaluation des performances de l'appariement :

Dans cette partie, on va proposer une stratégie qui permet d'analyser la performance des différentes méthodes d'appariements citées auparavant. On va comparer les résultats entre l'image 2 non-déformée et l'image 2 déformée.

On va se baser dans notre étude sur la déformation `cv2.warpAffine` qui permet d'effectuer une rotation de l'image avec un angle qu'on doit préciser.

On va tester les 3 méthodes pour différentes valeurs de l'angle de rotation et on va calculer la différence entre les points appariés avec l'image non tournée et les points appariés avec l'image tournée pour savoir le nombre des points qu'on a perdu en effectuant la rotation. Nous examinons aussi le temps d'exécution des détecteurs.

**Modification au niveau du code** Au niveau du code, on doit ajouter la fonction qui effectue la rotation ainsi que les différents étapes pour le calcul du descripteur et du points d'appariement :

```
1 #Rotation d'un angle 90
2 rows = img2.shape[0]
3 cols = img2.shape[1]
4 M = cv2.getRotationMatrix2D((cols/2,rows/2),90,1)
5 img3 = cv2.warpAffine(img2,M,(cols,rows))
```

Résultat pour l'algorithme ***Features\_Match\_CrossCheck.py*** avec le descripteur ORB :

On a testé la méthode de ***Features\_Match\_CrossCheck.py*** avec des différents angles (incrémentation de 15°) et on a obtenu ces résultats

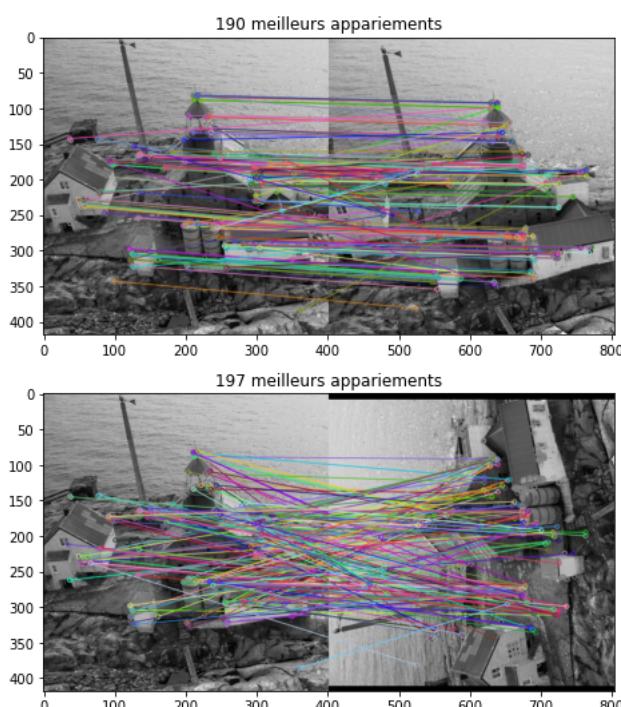


FIGURE 43 – CrossCheck avec ORB(angle=90°)

Angle de rotation	Appariements OK	Points non détectés
0°	190	-
15°	174	16
30°	180	10
45°	170	20
60°	175	15
75°	181	9
90°	197	-7
Moyenne	179,5	10,5

TABLE 1 – Résultat pour l'algorithme **Features\_Match\_CrossCheck.py** : ORB

On remarque pour le descripteur ORB avec l'angle 90°, on a obtenu de meilleur résultat lorsque l'image est tournée 197 points contre 197.

Résultat pour l'algorithme **Features\_Match\_CrossCheck.py** avec le descripteur KAZE :

On a testé la méthode de **Features\_Match\_CrossCheck.py** avec des différents angles (incrémentation de 15°) et on a obtenu ces résultats

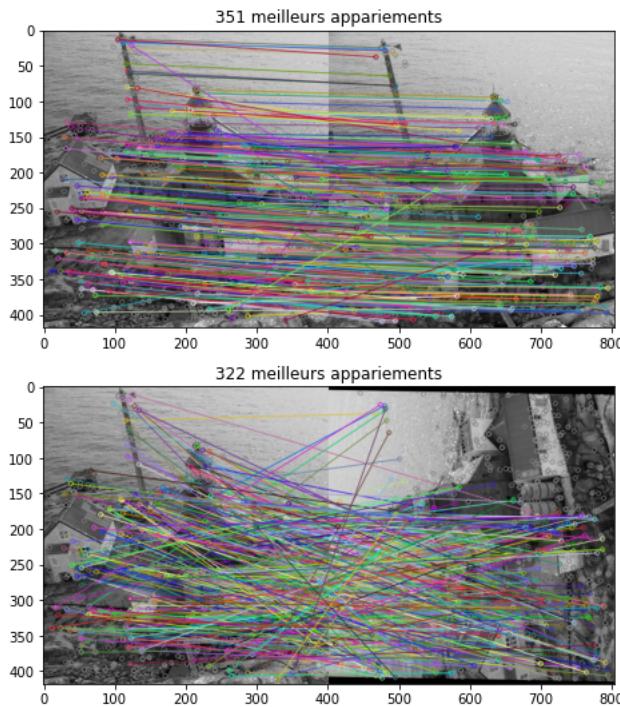


FIGURE 44 – CrossCheck avec KAZE(angle=89°)

Angle de rotation	Appariements OK	Points non détectés
0°	351	-
15°	292	59
30°	282	69
45°	273	78
60°	278	73
75°	282	69
89°	322	29
Moyenne	288,1	62,8

TABLE 2 – Résultat pour l'algorithme **Features\_Match\_CrossCheck.py** : KAZE

Résultat pour l'algorithme **Features\_Match\_RatioTest.py** avec le descripteur ORB :

On a testé la méthode de **Features\_Match\_RatioTest.py** avec des différents angles (incrémentation de 15°) et on a obtenu ces résultats

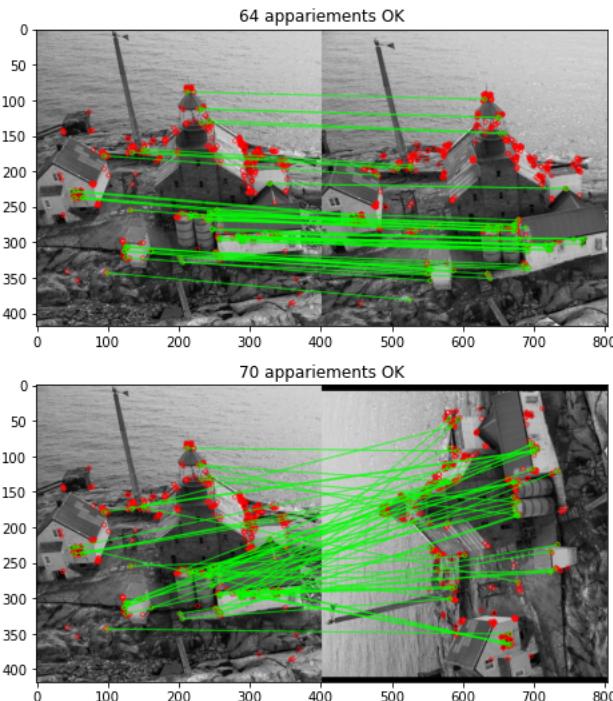


FIGURE 45 – RatioTest avec ORB(angle=90°)

On remarque pour le descripteur ORB avec l'angle 90°, on a obtenu de meilleur résultat lorsque l'image est tournée 70 poitns contre 64.

Angle de rotation	Appariements OK	Points non détectés	Temps de calcul du détecteur
0°	64	-	0.0114695s
15°	59	39	0.0161836 s
30°	45	19	0.0156707 s
45°	46	18	0.0185726 s
60°	41	23	0.0158558 s
75°	64	0	0.0155821 s
90°	70	-6	0.0212346 s
Moyenne	64,8	15,5	0.017s

TABLE 3 – Résultat pour l'algorithme **Features\_Match\_RatioTest.py** : ORB

Résultat pour l'algorithme **Features\_Match\_RatioTest.py** avec le descripteur KAZE :

On a testé la méthode de **Features\_Match\_RatioTest.py** avec des différents angles (incrémentation de 15°) et on a obtenu ces résultats

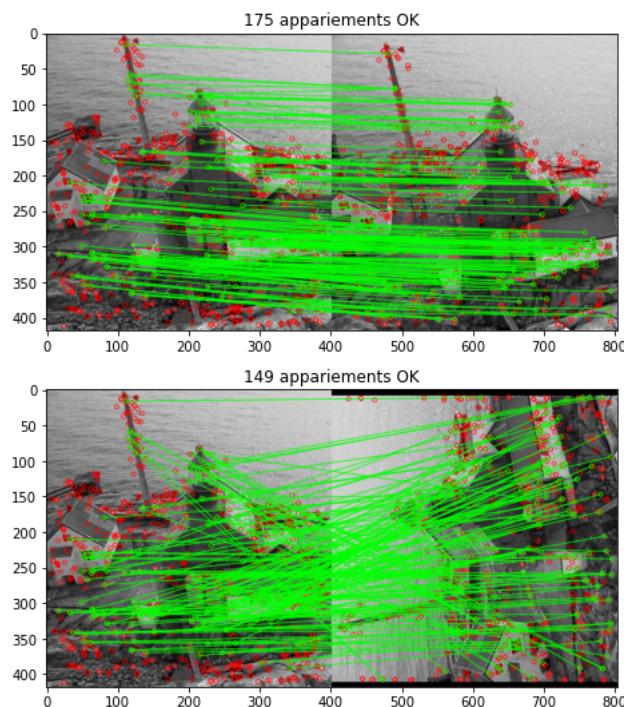


FIGURE 46 – RatioTest avec ORB(angle=90°)

Angle de rotation	Appariements OK	Points non détectés	Temps de calcul du détecteur
15°	136	39	0.4670533s
30°	129	46	0.5175992s
45°	124	51	0.4695532 s
60°	132	43	0.5094186 s
75°	135	38	0.4591551 s
90°	149	26	0.4228233 s
Moyenne	149	40,5	0.466666

TABLE 4 – Résultat pour l'algorithme **Features\_Match\_RatioTest.py** : KAZE

Résultat pour l'algorithme **Features\_Match\_FLANN.py** avec le descripteur ORB :

On a testé la méthode de **Features\_Match\_FLANN.py** avec des différents angles (incrémentation de 15°) et on a obtenu ces résultats

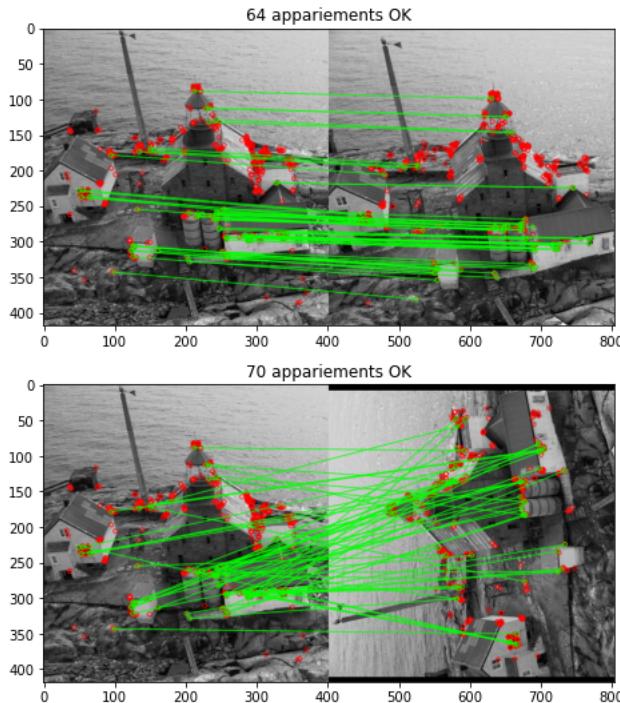


FIGURE 47 – FLANN avec ORB(angle=90°)

On remarque pour le descripteur ORB avec l'angle 90°, on a obtenu de meilleur résultat lorsque l'image est tournée 71 poitns contre 64.

Angle de rotation	Appariements OK	Points non détectés	Temps de calcul du détecteur
0°	64	-	0.0114695s
15°	59	5	0.0159769 s
30°	46	18	0.0186298 s
45°	46	18	0.0178073 s
60°	44	20	0.0156877 s
75°	66	-2	0.0160942 s
90°	71	-7	0.0165688 s
Moyenne	55,5	8,66	0.01616s

TABLE 5 – Résultat pour l'algorithme **Features\_Match\_FLANN.py** : ORB

Résultat pour l'algorithme **Features\_Match\_FLANN.py** avec le descripteur KAZE :

On a testé la méthode de **Features\_Match\_FLANN.py** avec des différents angles (incrémentation de 15°) et on a obtenu ces résultats

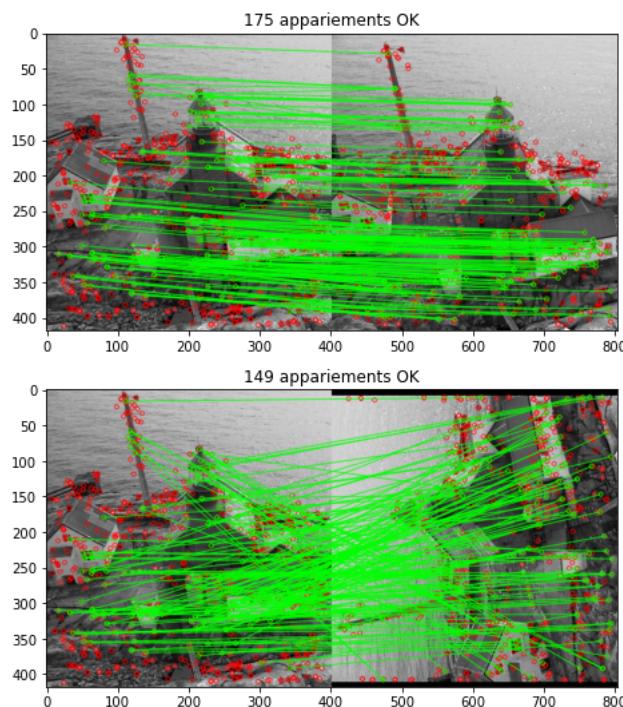


FIGURE 48 – FLANN avec KAZE(angle=90°)

Le meilleur résultat est obtenu pour une valeur de 90° avec 140 points contre 175 points pour l'image non tournée.

Angle de rotation	Appariements OK	Points non détectés	Temps de calcul du détecteur
0°	175	-	0.3181102 s
15°	136	39	0.4698299 s
30°	129	46	0.4338178 s
45°	124	51	0.4731365 s
60°	132	43	0.4662834 s
75°	135	40	0.4275428 s
90°	140	26	0.4146727 s
Moyenne	132,66	40,8	0,44s

TABLE 6 – Résultat pour l’algorithme *Features\_Match\_FLANN.py* : KAZE

## Conclusion et comparaison entre les méthodes :

On remarque que d’après les résultats que le(descripteur ORB possède un résultat meilleur que le(descripteur KAZE. En fait, les points non détectés par ce descripteur est inférieur aux points non détectés par le descripteur KAZE. On peut conclure que ORB est plus invariant à la rotation que KAZE. Par contre, ce descripteur n'affiche pas un grand nombre de points d'appariements ce qui le rend parfois inefficace.

### Descripteur ORB

Le nombre des points non détectés est similaires pour les 3 méthodes mais on peut remarquer que ORB avec *Features\_Match\_CrossCheck.py* détecte plus de points avec la variation de l’angle de rotation.

### Descripteur KAZE

Avec les méthodes *Features\_Match\_RatioTest.py* et *Features\_Match\_FLANN.py*, on a obtenu 40 points non détectés en moyenne, par contre pour *Features\_Match\_CrossCheck.py*, ce nombre atteint 62. On peut affirmer que *Features\_Match\_RatioTest.py* et *Features\_Match\_FLANN.py* sont plus meilleurs que *Features\_Match\_CrossCheck.py*.