TD5 – Généralisation à un modèle générique (3 séances)

L'objectif est de reprendre l'étude précédente dans le cas d'un modèle plus général qu'un simple damier.

On notera que deux nouvelles fonctions sont données dans le `helper.py` :

— `keypoints2numpyarray`

— `drawMatches`.

1 Résumé de l'approche

Introduction Au lieu de détecter les coins des cases d'un damier, nous allons considérer des points caractéristiques détectés dans l'image de référence ("keypoints" - voir Figure 1) et trouver les points correspondants dans chaque nouvelle image (i.e. avec les mêmes caractéristiques).

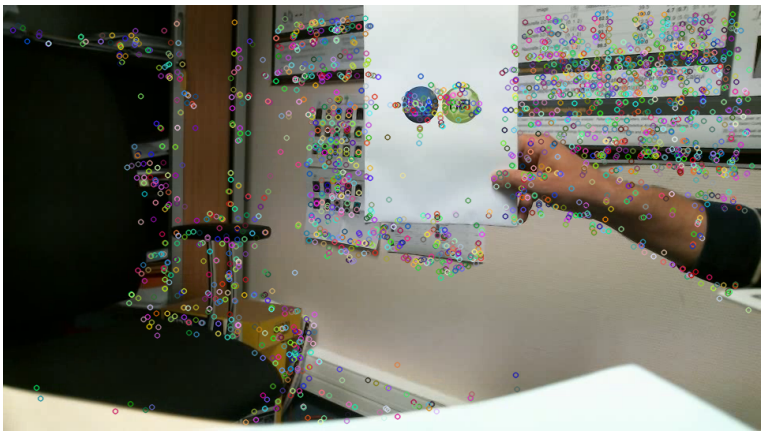

FIGURE 1 – Une image avec ses "keypoints" superposés (algorithme SURF). De plus, chaque keypoint est muni d'une description de son voisinage sous forme d'un vecteur numérique.

Un keypoint est caractérisé par des coordonnées (un point particulier de l'image) et est muni d'un descripteur (vecteur de valeurs numériques caractérisant le voisinage de ce point, e.g. les intensités). L'algorithme choisi pour détecter les keypoints et leurs vecteurs numériques associés est le *SURF* («speed up robust features»). Cet algorithme est couramment utilisé en vision par ordinateur, et pas seulement pour du recalage et de la réalité augmentée (e.g. reconnaissance, suivi,...).

Application à notre cas Afin d'estimer la matrice extrinsèque, il est impératif d'appareiller les keypoints similaires entre deux images (comme pour le damier, entre les cases dans l'image de référence et celles de l'image courante). Deux keypoints (de deux images différentes) sont appareillés (i.e. considérés comme similaires) si leurs vecteurs caractéristiques respectifs sont

similaires. Cette similarité entre deux vecteurs caractéristiques peut être mesurée en calculant simplement la distance qui les sépare.

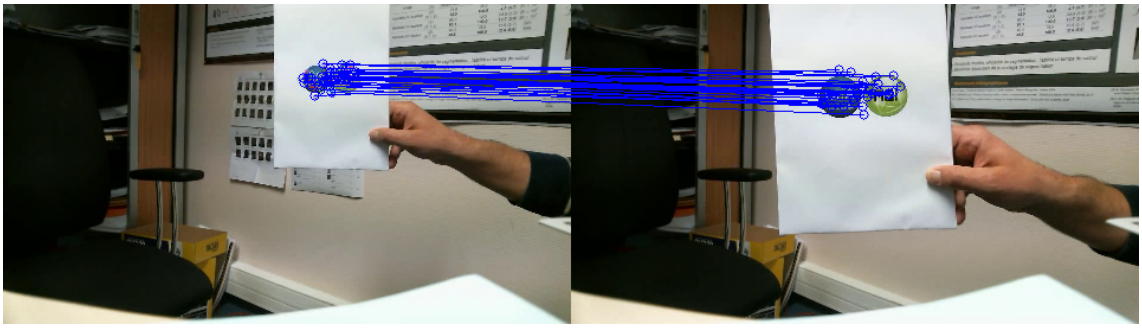

FIGURE 2 – Les “keypoints” de l’image courante (droite) sont mis en correspondance avec ceux de l’image de référence (gauche).

A partir de cette correspondance entre les “keypoints”, il est possible d’estimer la matrice extrinsèque. A partir de cette matrice extrinsèque et comme dans le TP précédent, on pourra correctement positionner un objet virtuel. Cette possibilité est illustrée par la figure 3

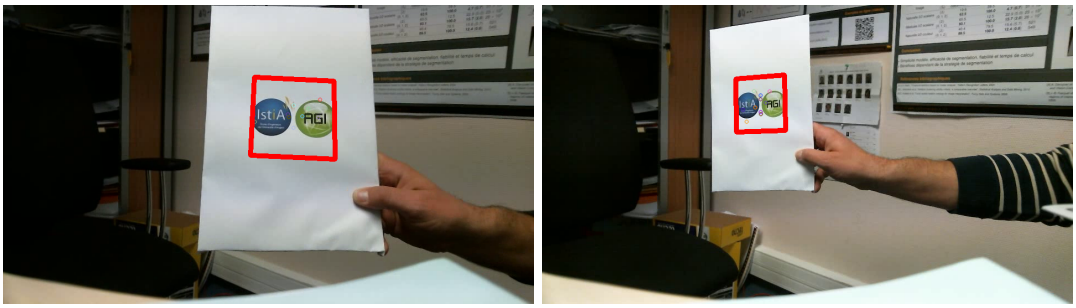

FIGURE 3 – Utilisation de la matrice extrinsèque pour correctement positionner l’objet virtuel.

2 Travaux pratiques

Exercice 1 (Mise en correspondance)
L’objectif de cet exercice est de découvrir

- comment calculer les keypoints et leurs caractéristiques sur une image donnée,
- comment, à partir de deux familles de keypoints obtenues sur deux images, les mettre en correspondance.

1. Le code suivant donne la manière de calculer les keypoints et leurs caractéristiques :

```
image1 = cv2.imread('reference_image_logo.png')
detector=cv2.xfeatures2d.SURF_create()
keypoints1, descriptors1 = detector.detectAndCompute(image1,None)
```

   Calculez les keypoints de l’image de référence `reference_image_logo.png`.

2. Faites de même pour les points de l’image `pose_logo_1.png`.

3. Afin de mettre en correspondance les deux familles de keypoints obtenues dans les deux questions précédentes, on utilisera la classe opencv `BFMatcher` :

```
bf=cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
```

Cette classe permet de trouver les correspondances entre les descripteurs (méthode `match`). Ces correspondances sont retournées sous forme d'une liste de `matches` (objets de la classe opencv `Match`). Un objet `Match` stocke l'indice de deux descripteurs similaires (attributs `trainIdx` et `queryIdx`), permettant ainsi d'avoir les indices des points clés associés.

Calculez cette liste de correspondances et utilisez la fonction d'affichage fournie par opencv pour afficher une image comme celle illustrée par la figure 4.

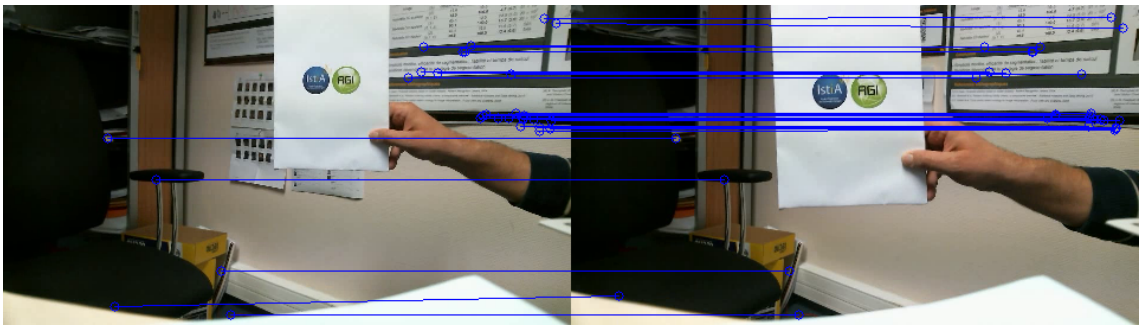

FIGURE 4 – Les traits relient les keypoints dont les caractéristiques sont considérées comme similaires.

4. Comme vous pouvez le constater, votre correspondance est en quelque sorte trop riche. Afin d'afficher uniquement les correspondances les plus significatives, on pourra les trier avec une instruction du type :

```
matches = sorted(matches, key = lambda x:x.distance)
```

Conservez uniquement que les 30 premiers `matches` (e.g. de 0 à 30), i.e. correspondant aux points les plus similaires, et affichez le résultat.

Indications : Les `matches` sont des instances de la classe opencv [DMatch](voir documentation en ligne). Les attributs publics sont les suivants
- `distance` est la distance entre les deux descripteurs associés au `match`.
- `queryIdx` et et `trainIdx` sont respectivement les indices permettant de retrouver les points clés et descripteurs associés à cette correspondance de type `match`.

5. Comme seuls les keypoints de la feuille blanche nous importent, calculez les keypoints uniquement dans cette zone de l'image.

Indication : Il est possible d'appliquer un masque à une image afin de filtrer une région d'intéret. On pourra consulter la documentation de la classe `SURF` afin d'adapter votre code en utilisant l'image `reference_image_logo_mask.png` fournie.
Indication : On chargera l'image masque en niveau de gris avec une instruction comme

```
mask=cv2.imread('reference_image_logo_mask.png',0) #0: pour un
    channel
```

6. Calculez et affichez ensuite les correspondances entre l'image de référence masqués et chaque image fournie.

Exercice 2 (Estimation de la matrice extrinsèque)
Cet exercice se décompose en deux parties :

— placement du carré virtuel sur l'image de référence.

— estimation de la matrice extrinsèque afin de correctement placer l'objet virtuel.

placement

1. Calculez les keypoints sur l'image de référence.
2. A partir de ces keypoints, superposez le carré virtuel grâce à la fonction `create_square`.
   Indication : il s'agit de la même approche qu'avec les coins du damier : connaissant la configuration de la prise de vue de l'image de référence ($Z = 500$ mm) et la matrice intrinsèque, on peut estimer la position des points clés en 3D (repère caméra) et ainsi la position initiale du carré dans le repère caméra.
   Indication : Pour récupérer les coordonnées (x,y) des points clés sous forme d'un numpy array, pensez à utiliser la fonction `keypoints2numpyarray` fournie dans le helper.
3. En vous inspirant du travail fait avec le damier, estimez la matrice extrinsèque grâce aux points clés mis en correspondance, et appliquer la transformation adéquate à l'objet virtuel. Vous devriez obtenir des représentations comme celles données sur la figure 5

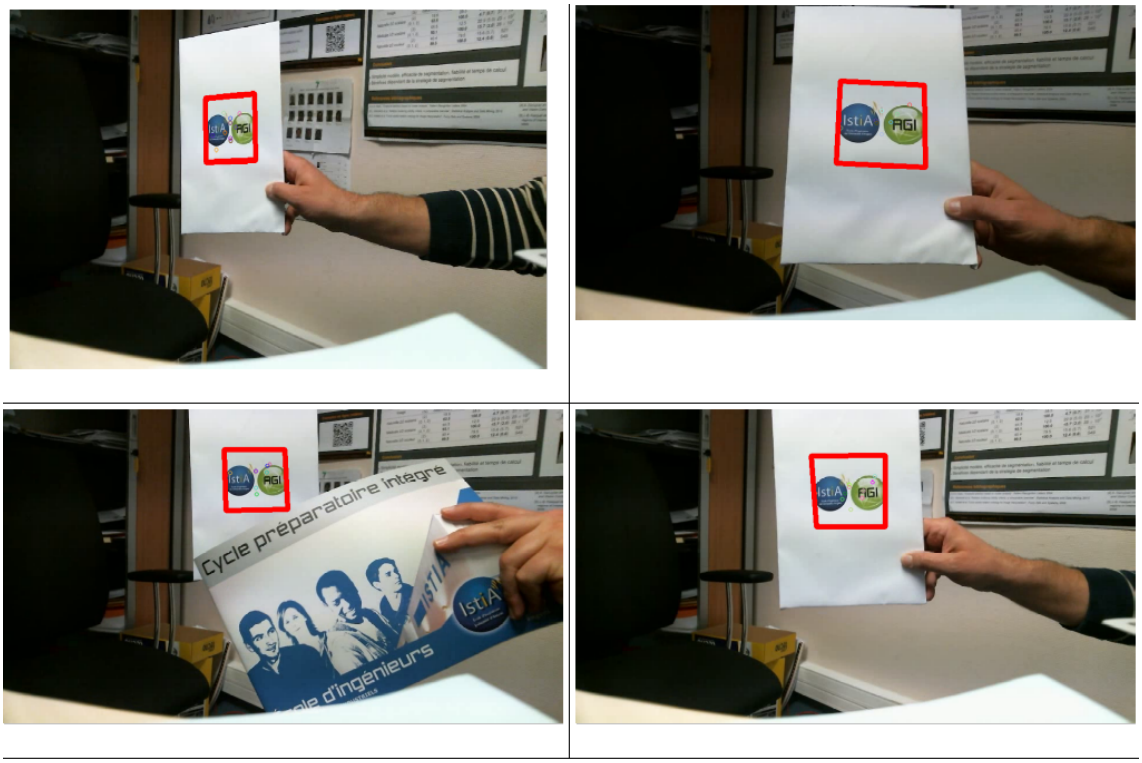

FIGURE 5 – Placements de l'objet virtuel.

Exercice 3
Dans le cas de l'exercice précédent, comme constaté pour certaines poses, le recalage peut s'avérer parfois de très mauvaise qualité (voir carrés rouges déformés sur la figure 6).

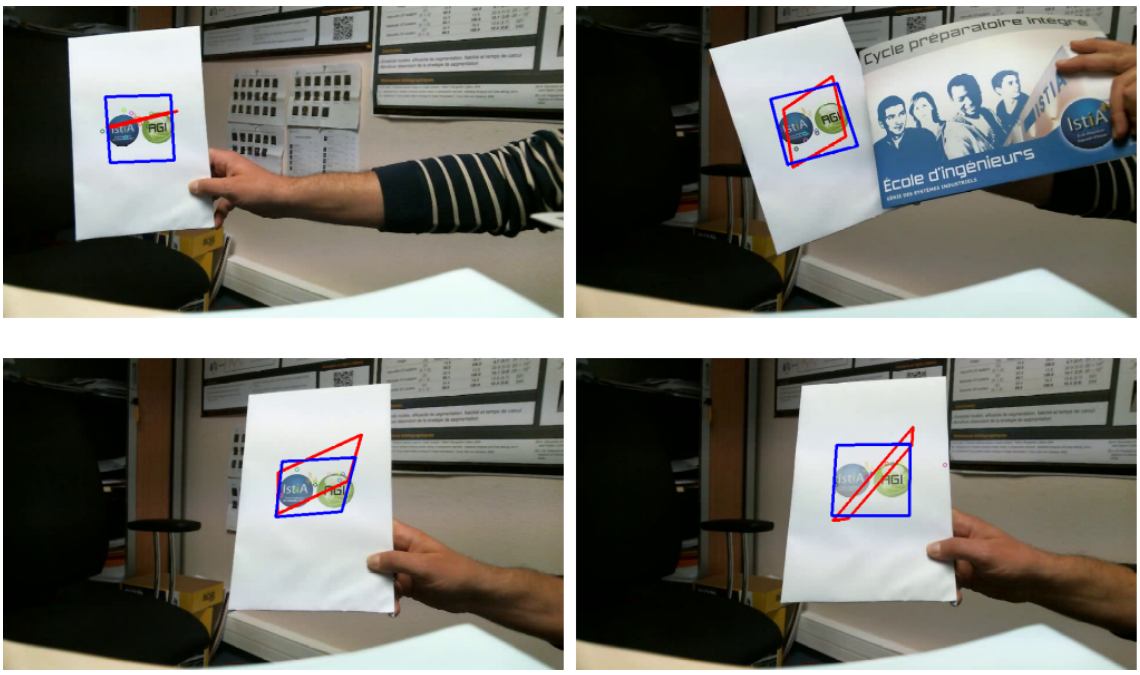

FIGURE 6 – Mauvais placements de l'objet virtuel.

Ceci est du à une mauvaise estimation de la matrice extrinsèque. Cette estimation erronée provient de mauvais appareillement entre certains points (des «outliers» qui faussent l'estimation).

Par contourner ce problème, nous proposons d'utiliser la procédure d'estimation RANSAC («RANdom SAmple Consensus») : l'idée est d'estimer plusieurs matrices, pour différents jeux de points (tirés aléatoirement), et de conserver le jeu de point minimisant l'erreur (i.e. entre points détectés et points projetés, en connaissant les appareillements).

1. Utilisez la fonction `solvePnPRansac` d'opencv, et estimez la matrice extrinsèque en couplant `SolvePnP` et *RANSAC*. Pour cela, vous pourrez créer une nouvelle fonction `helper.compute_extrinsic_matrix_ransac`, qui invoquera `opencv`.
2. Appliquez cette méthode afin d'affichez simultanément les carrés bleus (RANSAC) et les carrés rouges (sans RANSAC) comme sur la figure 6.