SY32 – TD Vision 05 : Mise en correspondance stéréo

À partir d'une paire d'images stéréo calibrées et rectifiées, illustrées en Figure 1, l'objectif de ce TD est de calculer :

— une carte des disparités de la scène ;

— les profondeurs issues des disparités ;

— l'évaluation quantitative de cette carte des disparités par rapport à une vérité terrain.

Paire stéréo à traiter

(a) Vue gauche Ig. (b) Vue droite Id.

FIGURE 1 – Paire d'images à faire correspondre pour estimer les disparités et les profondeurs suivant le principe de la stéréovision. Issue à l'orignine du jeu de données Stereo Middlebury 2014 : `https://vision.middlebury.edu/stereo/data/scenes2014/`.

1. Charger la paire d'images stéréo *motorcycle* de scikit-image. Appellons Ig l'image gauche, Id l'image droite, et gt la vérité des disparités.
   Observer Ig et Id et en déduire la disparité maximale entre les deux (correspond au décalage maximal d'un même point entre les deux points de vue), que l'on nommera *maxdisp*. Connaître la disparité maximale est utile pour bien configurer l'algorithme de mise en correspondance et lui éviter de chercher à des positions trop éloignées.
   On pourra utiliser le zoom synchronisé par `sharex=True` et `sharey=True` :

```
import numpy as np
from skimage import data
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import cv2 as cv # librairie OpenCV

%matplotlib auto

# ouvrir les images en RGB et en niveaux de gris dans [[0 ;255]]
Ig, Id, gt = data.stereo_motorcycle()
Iggray = (rgb2gray(Ig)*255).astype('uint8')
Idgray = (rgb2gray(Id)*255).astype('uint8')

fig, axes = plt.subplots(1,2, sharex=True, sharey=True)
axes[0].imshow(Ig)
axes[0].set_title("Ig")
axes[1].imshow(Id)
axes[1].set_title("Id")
plt.show()
```

   Note : une autre méthode possible pour visualiser le décalage est de superposer les deux images en une seule, en moyennant leurs intensités.

Exercice 1 : Mise en correspondance stéréo et profondeurs

On considère deux caméras identiques, autrement dit avec les mêmes paramètres intrinsèques. Elles ont la même orientation (même base), et sont placées le long du même axe, confondu avec leur axe horizontal X. Ainsi, seule une translation le long de X les sépare. Cette séparation est l'entraxe, d'une

longueur b. Pour la suite, on travaille dans le repère centré sur le centre optique de la caméra gauche, caméra donnant l'image de référence. L'image droite est l'image de recherche pour le processus de mise en correspondance.

1. Montrer que dans cette configuration, un point apparaissant en (x, y) dans une des images est contraint à se projeter le long d'une ligne horizontale d'ordonnée y dans l'autre image. Illustrer la réponse par un schéma.
2. Rappeler le lien existant dans cette configuration entre la disparité d et la profondeur z.
3. Pour cet exercice, nous allons calculer les disparités avec les algorithmes disponibles dans OpenCV. Appliquer les algorithmes de mise en correspondance stéréo suivants, avec la prise en compte du paramètre *maxdisp*, et afficher la carte des disparités comme une image (les disparités obtenues sont données aux positions des points de l'image de référence qui est l'image gauche) :
   (a) `cv.StereoBM`

```
# attention avec ces implementations,
# numDisparities doit etre divisible par 16,
# les algos proposent une precision a 1/16 pixel ;
# en cherchant les disparites avec un pas de 1/16
# les disparites reelles s'en retrouvent *16
numdisp = maxdisp if maxdisp%16==0 else ((maxdisp//16)+1)*16

# Block Matching
# attention, cette implementation ne fonctionne que
# pour des images en niveaux de gris
stereoBM = cv.StereoBM.create(numDisparities=numdisp, \
                              blockSize=15) # blockSize impair
dispBM = stereoBM.compute(Iggray,Idgray).astype(np.float32) / 16.0
```

   (b) `cv.StereoSGBM`

```
# Semi Global Block Matching
# peut traiter des images RGB ou en niveaux de gris
stereoSGBM = cv.StereoSGBM.create(numDisparities=numdisp, \
                                  blockSize=15) # impair
dispSGBM = stereoSGBM.compute(Ig,Id).astype(np.float32) / 16.0
```

   (c) `cv.stereo_QuasiDenseStereo`

```
# Stereo Quasi Dense,
# eparse puis densification par propagation
# ! ! necessite OpenCV >= 4.5 et opencv-contrib
# travaille en niveau de gris, meme avec des images RGB
# donne directement les disparites relles
stereoQD = cv.stereo_QuasiDenseStereo.create(monoImgSize=Iggray.shape[::-1])
stereoQD.process(Iggray,Idgray)
dispQD = stereoQD.getDisparity()
```

4. Pour chaque carte des disparités, générer un masque M des points pour lesquels une disparité a été associée (les points valides : c'est-à-dire non occultés ou sans ambiguïté), et afficher le nombre de points sans disparité.
   Les points sans correspondant (sans disparité) valent `-1` avec `cv.StereoBM` et `cv.StereoSGBM`, et valent `np.nan` avec `cv.stereo_QuasiDenseStereo`.
   Note : éliminer aussi du masque les points de disparité nulle pour éviter de calculer des points projetés à l'infini ensuite, ce qui n'est pas plausible pour la scène traitée (en fait, nous pourrions aussi imposer une disparité minimale pour les correspondances).
5. À partir d'une carte des disparités, générer une carte des profondeurs en appliquant la formule $z = \frac{fb}{d}$. Veiller à ignorer les points sans disparité (utiliser le masque M généré à la question précédente).
   Pour cela, poser $f = 994,978$ et $b = 0,193001$ (données issues de la documentation *skimage* pour la paire d'images *motorcycle*). f représente la focale de la caméra en pixels, et b l'entraxe entre les deux points de vues en mètres.
   Afficher la carte des profondeurs en leur appliquant le logarithme `np.log` pour une bonne distinction des plans.
6. Afficher en 3D les points en associant les profondeurs aux coordonnées (x, y). Veiller encore à ignorer les points non appariés en utilisant le masque M.

```
xx, yy = np.meshgrid(np.arange(Ig.shape[1]), np.arange(Ig.shape[0]))

fig = plt.figure()
```

```
ax = fig.add_subplot(111, projection='3d')
ax.scatter(xx[maskBM].flatten(), yy[maskBM].flatten(), \
          depthBM[maskBM].flatten(), \
          s=1, c=Ig[maskBM].reshape((-1,3))/255., \
          depthshade=False)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_zlim([3.,20.]) # limite raisonnable de z pour la scene
plt.show()
```

Note : Si l'affichage 3D est trop long, afficher seulement un point sur 4.

Exercice 2 : Implémentation de la mise en correspondance par *block matching*

1. Implémenter une fonction calculant la disparité de l'image gauche (référence) à l'image droite, par méthode de *block matching*, avec la mesure de dissimilarité SAD (*Sum of Absolute Differences*) définie comme suit :

$$SAD(bloc1, bloc2) = \sum_{(u,v)\in N\times N} |bloc2(u,v) - bloc1(u,v)|$$

L'implémentation doit tenir compte des bornes de disparité qui sont $[-maxdisp; 0]$ (car les points de I_d sont décalés vers la gauche par rapport à leur position dans I_g), et d'une taille de fenêtre paramétrable $N \times N$. Pour cette question, fixer $N = 5$.
Conseil : Pour commencer, travailler avec les images converties en niveaux de gris. Ensuite, une évolution pour traiter les images en couleurs pourra être faite.

La Figure 2 illustre l'algorithme de *block matching* demandé.
Note : ignorer les pixels des bords de l'image pour une longueur de $N/2$ afin de ne pas perdre de temps de développement à gérer les cas où la fenêtre de traitement déborderait du canevas.

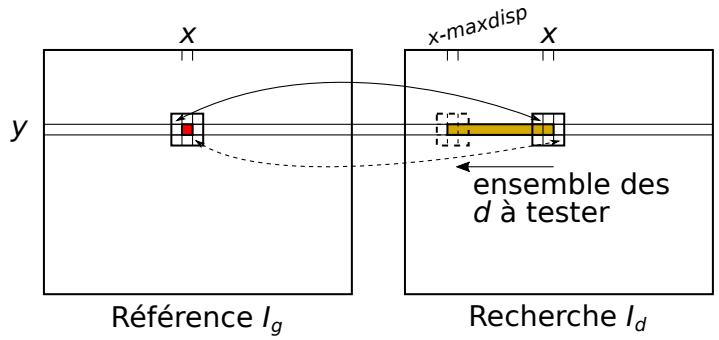

FIGURE 2 – Illustration de la correspondance de blocs le long des lignes épipolaires conjuguées (dans cet exercice pour la même ordonnée y), pour une fenêtre de taille 3×3, avec le point de référence en rouge, l'intervalle de recherche de la disparité en orange, et l'illustration des blocs à comparer. Cas où I_g est l'image de référence et I_d l'image de recherche.

2. Calculer aussi la disparité pour $N = 5$ en sens inverse, c'est-à-dire de l'image droite (référence) à l'image gauche. Dans ce cas la disparité devra être comprise dans l'intervalle $[0; maxdisp]$.
La Figure 3 montre un exemple des deux cartes de disparités obtenues en changeant d'image référence, pour une même implémentation de *block matching*, avec $N = 5$.
Note : En usage classique, on travaille presque toujours avec les disparités en valeur absolue. Convertir les cartes de disparités en valeurs absolues avant de les afficher.
3. Nettoyer chaque carte de disparités en appliquant un « filtre de mode » de taille 5×5. Ce filtre consiste à affecter au pixel considéré la valeur la plus présente localement, c'est-à-dire par rapport à la fenêtre 5×5 centrée sur ce pixel.
Commenter.
Dans ce contexte, pourquoi préférer ce filtre à un filtre médian ?
4. Tester l'algorithme de mise en correspondance développé pour différentes tailles de fenêtre (par exemple pour $N \in \{3, 5, 11, 21\}$). Faire ce test uniquement dans un sens pour éviter de faire trop de calculs.

FIGURE 3 – Cartes des disparités pour $N = 5$, à gauche l'image de référence est l'image gauche, à droite l'image de référence est l'image droite.

5. L'algorithme développé peut-il estimer des disparités avec une précision sous-pixellique ?
6. Que peut-on dire de la méthode développée au sujet des occultations ?
7. Réfléchir à une méthode de fusion bidirectionnelle des disparités.

Exercice 3 : Évaluation de cartes de disparités

1. Les disparités vérité sont données dans la matrice de flottants gt, par rapport aux points de l'image de référence : l'image gauche. Les occultations sont codées avec la valeur spécifique `np.inf`.
   Afficher une carte d'évaluation des occultations obtenues selon M par rapport à la vérité gt avec 4 valeurs possibles selon la table suivante (se rapporte à un problème d'évaluation de classification binaire) :

		Vérité gt	
		non-occulté ($< \infty$)	occulté (∞)
Masque M	valide / non-occulté (1)	1 (vrai négatif)	-1 (faux négatif)
	invalide / occulté (0)	-2 (faux positif)	2 (vrai positif)

   Pour cela, nous allons associer chaque valeur à une couleur, par l'utilisation d'une palette de couleurs créée sur mesure :

```
from matplotlib.colors import ListedColormap
# ordre des significations des couleurs : FP en rouge, FN en orange, "rien" en blanc, VN en bleu,
    VP en vert
occcmap = ListedColormap(["red", "orange", "white", "blue", "green"])
```

   Qu'observe-t-on ?
2. Le nombre d'occultations estimées qui sont des faux négatifs (points où une disparité a été estimée alors qu'ils n'auraient pas dû pouvoir être appariés) est un critère assez important pour l'évaluation des algorithmes de mise en correspondance. Afficher le nombre de ces points.
   Quel est l'intérêt de ce critère ?
3. Pour évaluer la qualité des disparités estimées, nous devons ignorer les points occultés, aussi bien de la vérité que du résultat. Calculer et afficher le masque des points à considérer pour les mesures d'erreur, défini par :
   $$Meval = M \ \cap \ (gt < \infty)$$
4. Afficher la carte des différences absolues entre les disparités calculées et les disparités vérité. Exclure et fixer à 0 les points hors du masque $Meval$. Fixer les bornes d'adaptation de l'échelle des pseudocouleurs de manière identique pour les différents algorithmes testés pour pouvoir comparer visuellement leurs erreurs, via les arguments `vmin` et `vmax` de la fonction `imshow`.
   Où se trouvent les erreurs les plus importantes ? Pourquoi ?
5. En ne considérant que les points admis par le masque $Meval$, calculer et afficher l'erreur moyenne, l'erreur médiane, l'erreur minimale, l'erreur maximale, et l'écart-type d'erreur de disparité.
6. Afficher un masque des points où l'erreur de disparité est supérieure à 1, en y incluant les points faux négatifs et faux positifs de l'évaluation des occultations (car ils sont erronés). Compter le nombre et la proportion de ces points (par rapport au total de points de l'image).

Liste de fonctions utiles :

- `skimage.io.imread`
- `numpy.zeros` et `numpy.zeros_like`
- `numpy.full` et `numpy.full_like`
- `numpy.shape`
- `numpy.copy`
- `numpy.sort`
- `numpy.count_nonzero`
- `numpy.nonzero`
- `numpy.meshgrid`
- `numpy.arange`
- `numpy.fabs`
- `numpy.flatten`
- `numpy.power` (ou opérateur `**`)
- `numpy.sum`
- `numpy.sqrt`
- `numpy.norm`
- `numpy.unique`
- `numpy.argmax`
- `numpy.logical_not`
- `numpy.logical_and` (ou opérateur `&`)
- `numpy.logical_or` (ou opérateur `|`)
- `numpy.pad`