



**Année universitaire
2023 - 2024**

Majeure IMI — Partie 3 — 5ETI
**Acquisition Calibrage et
Reconstruction 3D**

TP d'acquisition



**Eric Van Reeth
Mehdi Ayadi**

Organisation du TP

Objectifs

Ce TP est une introduction à l'acquisition et au traitement d'un flux vidéo en temps réel via la librairie openCV. Les objectifs sont les suivants :

- interfacer une caméra avec l'ordinateur
- parcourir certaines fonctionnalités d'openCV déjà connues
- introduire certains concepts utiles pour la suite du module

Évaluation

Ce TP n'est pas noté, mais conservez votre code bien structuré et commenté. Il sera utilisé pour de futurs TP et projets.

Mise en route

Ce TP s'effectue en binôme par poste informatique sous LINUX, et sera codé en Python. Veillez bien à **utiliser l'interpréteur Python3** spécifié sur e-campus.

L'utilisation de l'IDE **VSCode** est fortement conseillée pour faciliter l'implémentation et le debug des codes. L'aide d'openCV pourra se trouver ici.

1 Capture vidéo

1.1 Script de base de la capture vidéo

Pour réaliser la capture, on crée un objet de la classe `VideoCapture`. La création de cet objet requiert un argument qui spécifie l'indice de l'appareil réalisant la capture vidéo, où le nom du fichier d'une vidéo déjà existante. Dans notre cas, une seule caméra étant connectée, on passera simplement 0.

La capture vidéo se fait dans une boucle `while`, dans laquelle la méthode `read()` de l'objet `cap` est appelée. Cette fonction retourne deux arguments : un booléen valant `True` si la lecture est effectuée correctement, et la frame acquise (en couleur par défaut).

La sortie de la boucle se fait lors de l'appui sur une touche du clavier (ici la lettre `q`). En sortie de boucle, la capture est arrêtée et l'affichage fermé. Le code d'acquisition et d'affichage sera donc :

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

cap = cv2.VideoCapture(0)
while(True):
    ret, frame = cap.read() #1 frame acquise à chaque iteration
    cv2.imshow('Capture_Video', frame) #affichage

    key = cv2.waitKey(1) #on évalue la touche pressée
    if key & 0xFF == ord('q'): #si appui sur 'q'
        break #sortie de la boucle while

cap.release()
cv2.destroyAllWindows()
```

À ce stade, vous devez être capable d'afficher le flux vidéo provenant de votre webcam. Pour éviter les plantages d'interface entre le PC et la webcam, pensez à bien **fermer le flux vidéo actif avec la touche `q`**, avant de démarrer la capture d'un nouveau flux.

1.2 Modification des paramètres de capture

Certains paramètres du flux vidéo capturé peuvent être modifiés dès l'acquisition via la méthode `set` associée à l'objet `cap` créé. De même, il est possible d'accéder à la valeur de ces paramètres via la méthode `get`. Voici quelques paramètres à tester qu'il peut être intéressant de modifier pour certaines applications.

Note : les paramètres modifiés seront conservés jusqu'au nouveau branchement d'un périphérique d'acquisition.

```
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 240) # Fixe la hauteur du flux acquis à
→ 240 pixels
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 480) # Fixe la largeur du flux acquis à
→ 480 pixels
cap.set(cv2.CAP_PROP_FPS, 20) # Fixe la fréquence du flux à 20 images/s
cap.set(cv2.CAP_PROP_SHARPNESS, 3) # Fixe la netteté du flux à 3
cap.set(cv2.CAP_PROP_AUTO_EXPOSURE, 1) # 1 : off - 3 : on. Active
→ l'exposition automatique.
cap.set(cv2.CAP_PROP_BRIGHTNESS, 100) # Gère l'exposition du flux
cap.set(cv2.CAP_PROP_SATURATION, 100) # Gère la saturation du flux
cap.set(cv2.CAP_PROP_CONTRAST, 100) # Gère le contraste du flux
```

1.3 Enregistrement d'images

Ajouter dans la boucle d'acquisition la possibilité d'enregistrer l'image courante du flux vidéo lors de l'appui sur la touche `s`.

- Utilisez la fonction `plt.imsave()` pour l'enregistrement des images

- Veillez à incrémenter le nom de l'image sauvegardée de sorte à ne pas écraser la précédente
- Attention à la cohérence des espaces couleur lors de la sauvegarde (convention BGR pour openCV, et RGB pour matplotlib). La fonction `cv2.cvtColor()` vous sera utile.

1.4 Traitements en temps réel

L'objectif de cette partie est d'appliquer des opérations sur chaque image acquise, et d'afficher le résultat en temps réel.

1.4.1 Affichage d'histogramme

Utiliser la fonction `cv2.calcHist()` pour le calcul de l'histogramme de chaque frame acquise **en niveau de gris**. Soyez attentifs au format des paramètres d'entrée (pensez à regarder l'aide). Un exemple d'utilisation est donné ci-dessous :

```
hist = cv2.calcHist([frame], [channel], None, [binsNb], [minHist, maxHist])
```

- `frame` est l'image dont on souhaite calculer l'histogramme
- `channel` le canal couleur d'intérêt
- `binsNb` le nombre d'éléments contenus dans l'histogramme
- `minHist` le minimum de l'intervalle d'intensité considéré
- `maxHist` le maximum de l'intervalle d'intensité considéré

Afin de mettre à jour le tracé de l'histogramme à chaque itération de la boucle d'acquisition, l'affichage se fera avec les commandes suivantes :

```
plt.plot(np.linspace(minHist,maxHist,bins),hist)
plt.title('Histogramme')
plt.draw()
plt.pause(0.0001) #delai nécessaire a l'affichage
plt.cla() #évite la superposition des courbes
```

1.4.2 Détection de lignes (transformée de Hough)

La transformée de Hough permet, entre autres, la détection de droites dans les images. Deux versions de cette transformée sont incluses dans openCV : `cv2.HoughLines()` et `cv2.HoughLinesP()`.

Nous utiliserons `cv2.HoughLinesP()` pour ce TP.

Après avoir effectué une détection de contours avec la méthode de votre choix (Canny, Sobel), détecter les droites sur chaque frame du flux vidéo acquis.

Pour l'affichage des lignes détectées, on utilisera le code suivant :

```
# On appelle lines la sortie de cv2.HoughLinesP
if not(lines is None): # si lines contient au moins une ligne
    for line in lines: # on itère sur les lignes détectées
        x1,y1,x2,y2 = line[0] # on récupère les coordonnées de la ligne
        ↪ courante
        res = cv2.line(frame, (x1,y1), (x2,y2), (0,255,0), 2) # on superpose la
        ↪ ligne courante à la frame courante pour l'affichage
```

Étudier l'influence des paramètres de la fonction `cv2.HoughLinesP()` à la fois sur la fluidité et sur la qualité de la détection.

1.4.3 Détection de points d'intérêts

La détection automatique de régions d'intérêts dans les images est une étape primordiale en vision par ordinateur (recalage d'image et reconstruction 3D notamment). Une région d'intérêt est une région circulaire de l'image, contenant des structures considérées comme importantes visuellement. Celles-ci sont caractérisées par certaines propriétés et correspondent en général à des éléments distincts et marquants de l'image (contours, intersections par exemple). Une description didactique peut être trouvée ici. Idéalement, la détection de points d'intérêts doit être robuste aux changements des paramètres d'acquisition (rotation, luminosité, changement d'échelle).

Plusieurs algorithmes de détection et d'appariement de régions/points d'intérêts ont été proposés (SIFT, ORB, SURF) dont certains sont protégés par des brevets toujours actifs. OpenCV propose plusieurs implémentations, et on s'intéressera ici à l'algorithme ORB.

Dans un premier temps, on initialise la détection de point d'intérêts (avant la boucle d'acquisition). Certaines options clés optionnelles peuvent être spécifiées (nombre de points d'intérêt détectés, nombre de résolution pris en compte notamment) :

```
orb = cv2.ORB_create(nfeatures=..., nlevels=...)
```

Pour chaque frame acquise, on exécutera la détection et l'affichage des points d'intérêt via les commandes suivantes :

```
kp = orb.detect(frame, None) # Détection des "keypoints" sur la frame
→ courante
res = cv2.drawKeypoints(frame, kp, None, color=(0,255,0), flags=0) #
→ L'image res contient les "keypoints" affichés sur la frame courante
```

En filmant différents types de scènes, repérer les structures qui sont systématiquement considérées comme points d'intérêts. Discuter de l'avantage en terme de sensibilité/robustesse de détection, par rapport à la détection de lignes de la section précédente.

2 Application d'une transformation affine

Il s'agit maintenant d'appliquer en temps réel des déformations géométriques sur l'image acquise. OpenCV permet à la fois l'implémentation de transformations linéaires, affines et non linéaires.

Pour le cas de transformations affines, on spécifiera la matrice M de dimension (2×3) , qui contient les paramètres de transformation :

$$\begin{pmatrix} x_d \\ y_d \end{pmatrix} = \begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad (1)$$

M contient à la fois les paramètres de translation, d'échelle, de rotation (angle et centre) :

$$M = \begin{pmatrix} \alpha & \beta & (1 - \alpha)c_x - \beta c_y + t_x \\ -\beta & \alpha & \beta c_x + (1 - \alpha)c_y + t_y \end{pmatrix} \quad (2)$$

avec (c_x, c_y) les coordonnées pixels du centre de rotation, (t_x, t_y) le vecteur translation, s le facteur d'échelle, et :

$$\begin{aligned} \alpha &= s \cdot \cos \theta \\ \beta &= s \cdot \sin \theta \end{aligned}$$

En utilisant la fonction `cv2.warpAffine`, implémenter les transformations permettant de réaliser sur chaque image du flux vidéo acquis :

- une translation de 10 pixels en abscisse et 15 pixels en ordonnées
- une rotation de 180 degrés autour du centre de l'image
- une diminution de moitié de la taille de l'image (dans les deux dimensions)

3 Application d'une transformation non linéaire

Par définition, l'application d'une transformation non linéaire ne peut se faire via une opération matricielle. On définit dans ce cas deux matrices de déformation, appelées D_x et D_y , de mêmes dimensions que l'image à déformer. Ces matrices contiennent les nouvelles positions (abscisses pour D_x et ordonnées pour D_y) des pixels de l'image déformée I_d (voir schéma ci-dessous) :

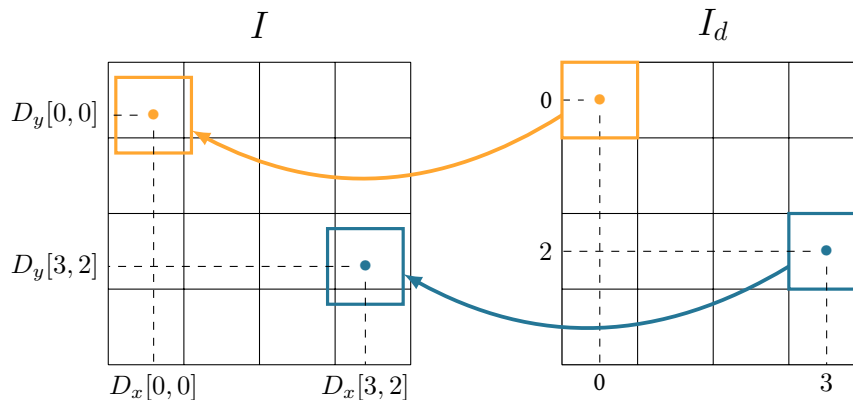
$$D_x[x, y] = x + \Delta x$$

$$D_y[x, y] = y + \Delta y$$

L'image déformée est donc obtenue à partir de l'image originale I , par la formule suivante :

$$I_d[x, y] = I[D_x[x, y], D_y[x, y]]$$

Les valeurs des pixels de I_d sont obtenues par interpolation des valeurs de I , aux emplacements indiqués par les matrices de déformation :



La fonction `cv2.remap`, permet d'appliquer une déformation non linéaire sur une image. Elle prend en argument les cartes de déformation (D_x et D_y), ainsi que la méthode d'interpolation à utiliser.

Tester quelques déformations de votre choix, ou inspirez-vous des propositions suivantes :

- Profil d'un filtre passe-bande :

$$\begin{cases} \Delta x = A \cos(\alpha) e^{-\left(\frac{\tilde{y}^2 + \tilde{x}^2 - \delta^2}{\sigma_x \sqrt{\tilde{x}^2 + \tilde{y}^2}}\right)^2} \\ \Delta y = A \sin(\alpha) e^{-\left(\frac{\tilde{y}^2 + \tilde{x}^2 - \delta^2}{\sigma_y \sqrt{\tilde{x}^2 + \tilde{y}^2}}\right)^2} \end{cases} \quad \text{avec } \tilde{x} = x - c_x, \tilde{y} = y - c_y, \text{ et } \alpha = \arg(\tilde{x} + i\tilde{y})$$

- Profil d'une Gaussienne 2D :

$$\begin{cases} \Delta x = A \cos(\alpha) e^{-\left(\frac{\tilde{x}^2}{\sigma_x} + \frac{\tilde{y}^2}{\sigma_y}\right)} \\ \Delta y = A \sin(\alpha) e^{-\left(\frac{\tilde{x}^2}{\sigma_x} + \frac{\tilde{y}^2}{\sigma_y}\right)} \end{cases}$$

- Déformation aléatoire, régénérée ou non à chaque frame

Note : la fonction `meshgrid` permet d'obtenir les cartographies de coordonnées.

Note : on prendra le centre de l'image comme point central de la déformation.

Les valeurs des paramètres A , σ_x , σ_y et δ doivent être adaptées au format de l'image acquise.