Université Paris Cité
UFR de Mathématiques et Informatique

TP n°3 – Analyse d'images

Convolution et Transformations géométriques

Master 1 Vision et Machine Intelligente

Enseignant : C. Kurtz

Auteur : Alexandre Marseloo

Date : Février 2026

Introduction

L'objectif de ce TP était de mettre en œuvre des opérations de traitement d'image à l'aide de OpenCV. Deux types de traitement ont été utilisés :

- **Les convolutions** : elles permettent d'appliquer des filtres pour le lissage ou le renforcement de contours.
- **Les transformations géométriques** : elles permettent de modifier la taille, la forme ou la position d'une image, des méthodes d'interpolation nous permettent d'estimer les valeurs des pixels pour agrandir une image.

Le code est disponible sur GitHub.

Méthodologie

Convolution

Pour la première partie consacrée aux convolutions, j'ai codé mes propres fonctions et exécutables. J'ai implémenté une fonction `convolution` capable de calculer la convolution d'une image par un noyau, en gérant les bords de l'image avec un effet mirroir. Cette fonction m'a permis d'appliquer différents filtres sur des images, notamment un filtre moyenneur pour lisser l'image et un filtre de Sobel pour calculer la norme du gradient et mettre en évidence les contours.

Par ailleurs, j'ai également développé une fonction spécifique pour appliquer un filtre médian, permettant de réduire le bruit tout en préservant les contours de l'image.

L'utilisation détaillée de ces fonctions est fournie dans le fichier `readme.txt` du dossier `tp_convolution`.

Je n'ai pas implémenté le filtrage bilatéral, car il est incompatible avec la manière dont ma fonction `convolution` est codée. En effet, ce filtrage nécessite de modifier le noyau pour chaque pixel, alors que ma fonction calcule la convolution pour l'ensemble de l'image d'un seul coup. Pour l'ajouter, il faudrait modifier la fonction afin de calculer et renvoyer les valeurs pixel par pixel.

Transformations géométriques

Pour la deuxième partie consacrée aux transformations géométriques, j'ai complété le fichier `tpGeometry.cpp` du projet fourni. Pour chaque exercice, j'ai inclus des traces d'exécution illustrant les résultats des tests réalisés.

Convolution

Ex 2 : Produit de convolution

La fonction `convolution`, codée dans `tpConvolution.cpp`, calcule la convolution d'une image f par un noyau g selon la formule :

$$(f * g)(x, y) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i, j)\, g(x - i, y - j), \quad \forall (x, y) \in \mathbb{Z}^2.$$

Le résultat est ensuite normalisé si la somme des éléments du noyau est différente de zéro.

Ex 1 : Filtre moyenneur

L'exécutable `moyenneur` applique la fonction `convolution` sur une image en utilisant un noyau moyenneur de taille 11×11, défini par :

$$g = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}_{11 \times 11}.$$

Ex 3 : Détecteur de contours de Sobel

L'exécutable `grad_sobel` applique la fonction `convolution` sur une image en utilisant deux noyaux de Sobel, l'un pour la détection des gradients horizontaux (G_x) et l'autre pour les gradients verticaux (G_y). Les noyaux sont définis par :

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

Après avoir calculé les convolutions avec ces noyaux, l'exécutable calcule la norme du gradient de Sobel pour chaque pixel (x, y) selon la formule :

$$\|\nabla f(x, y)\| = \sqrt{\big((f * G_x)(x, y)\big)^2 + \big((f * G_y)(x, y)\big)^2}.$$

Cette opération permet de mettre en évidence les contours de l'image en combinant les variations horizontales et verticales.

Ex 5 : Filtre médian

L'exécutable `median` applique un filtre médian sur une image afin de réduire le bruit tout en préservant les contours.

Pour chaque pixel de l'image, on extrait les valeurs des pixels voisins définis par le noyau, on les place dans une liste qu'on trie pour récupérer la valeur médiane. Cette valeur devient celle du pixel dans l'image filtrée.

Ainsi, contrairement au filtre moyenneur, le filtre médian n'effectue pas de moyenne et est moins sensible aux valeurs extrêmes. Il permet ainsi de réduire efficacement le bruit impulsionnel tout en conservant la netteté des contours.

Transformations géométriques

Ex 1 : Transposée d'une image

La fonction `transpose` calcule la transposée d'une image. Pour chaque pixel (x, y), sa valeur est échangée avec celle du pixel (y, x) :

$$g(x, y) = f(y, x).$$

Cette opération échange les lignes et les colonnes de l'image, produisant une image miroir par rapport à la diagonale principale.

Ex 2 : Agrandisssement d'image et interpolation par plus proche voisin

L'agrandissement d'image est réalisé à l'aide de deux fonctions principales : `interpolate_nearest` et `expand`.

- La fonction `interpolate_nearest` retourne la valeur d'un pixel en utilisant la méthode du plus proche voisin. Pour un point de coordonnées réelles (x, y) dans l'image agrandie, la valeur est donnée par :

  $$f_{\text{interpolated}}(x, y) = f\big(\text{round}(x), \text{round}(y)\big),$$

  où round désigne l'arrondi à l'entier le plus proche.

- La fonction `expand` calcule l'image agrandie pixel par pixel. Si le facteur d'agrandissement est $s > 0$, chaque pixel (x, y) de l'image agrandie correspond à une position (x', y') dans l'image originale donnée par :

  $$x' = \frac{x}{s}, \quad y' = \frac{y}{s}.$$

  La valeur du pixel agrandi est ensuite obtenue par interpolation via la fonction passée en argument (ici `interpolate_nearest`) :

  $$g(x, y) = \text{interpolate}(f, x', y').$$

Ex 3 : Agrandisssement d'image et interpolation bilinéaire

Interpolation bilinéaire (formule générale)

Pour un point (x', y') situé dans le carré défini par les quatre pixels (x_1, y_1), (x_2, y_1), (x_1, y_2), (x_2, y_2), la valeur interpolée est donnée par :

$$f(x', y') = (1-\alpha)(1-\beta)f(x_1, y_1)+\alpha(1-\beta)f(x_2, y_1)+(1-\alpha)\beta f(x_1, y_2)+\alpha\beta f(x_2, y_2),$$

avec

$$\alpha = \frac{x' - x_1}{x_2 - x_1} \in [0, 1], \quad \beta = \frac{y' - y_1}{y_2 - y_1} \in [0, 1].$$

Cette formule combine une interpolation linéaire horizontale (α) et verticale (β) pour obtenir la valeur finale du pixel, ce qui produit une image lissée et continue. Dans ce TP, cette interpolation bilinéaire est utilisée avec la fonction `expand` pour calculer pixel par pixel l'image agrandie selon un facteur réel.

Ex 4 : Rotation

La fonction `rotate` effectue la rotation d'une image d'un angle θ (en degrés) autour de son centre en utilisant une fonction d'interpolation passée en argument.

Le principe mathématique est le suivant : pour chaque pixel $(x_{\text{out}}, y_{\text{out}})$ de l'image de sortie, on calcule les coordonnées correspondantes $(x_{\text{orig}}, y_{\text{orig}})$ dans l'image originale à l'aide de la transformation inverse de rotation centrée :

$$\begin{cases} x_{\text{orig}} = (x_{\text{out}} - c_x^{\text{new}}) \cos\theta + (y_{\text{out}} - c_y^{\text{new}}) \sin\theta + c_x \\ y_{\text{orig}} = -(x_{\text{out}} - c_x^{\text{new}}) \sin\theta + (y_{\text{out}} - c_y^{\text{new}}) \cos\theta + c_y \end{cases}$$

où (c_x, c_y) et $(c_x^{\text{new}}, c_y^{\text{new}})$ sont les coordonnées des centres des images originale et agrandie, respectivement.

Avant de parcourir les pixels, on détermine la taille de l'image de sortie en calculant la position des coins de l'image originale après rotation : si (x_i, y_i) sont les coordonnées des quatre coins centrées sur l'origine, leurs nouvelles coordonnées sont :

$$\begin{cases} x_i^{\text{rot}} = x_i \cos\theta - y_i \sin\theta \\ y_i^{\text{rot}} = x_i \sin\theta + y_i \cos\theta \end{cases}$$

On en déduit alors les dimensions de l'image de sortie :

$$\text{new\_width} = \max(y_i^{\text{rot}}) - \min(y_i^{\text{rot}}), \quad \text{new\_height} = \max(x_i^{\text{rot}}) - \min(x_i^{\text{rot}})$$

Enfin, pour chaque pixel de l'image de sortie, la valeur est calculée via l'interpolation :

$$\text{res}(x_{\text{out}}, y_{\text{out}}) = \text{interpolationFunction}(\text{image}, x_{\text{orig}}, y_{\text{orig}}).$$

Cette approche garantit que chaque pixel de l'image résultante correspond exactement à une position dans l'image originale, tout en conservant le centre de rotation et en utilisant l'interpolation choisie pour estimer les valeurs intermédiaires.

Tests

Convolution

Je n'ai pas implémenté de fonctions de test dédiées pour la partie convolution. Néanmoins, l'application de ces fonctions aux images du dossier `data` permet de vérifier facilement leur bon fonctionnement, les effets des différents filtres étant visuellement observables.

Transformations géométriques

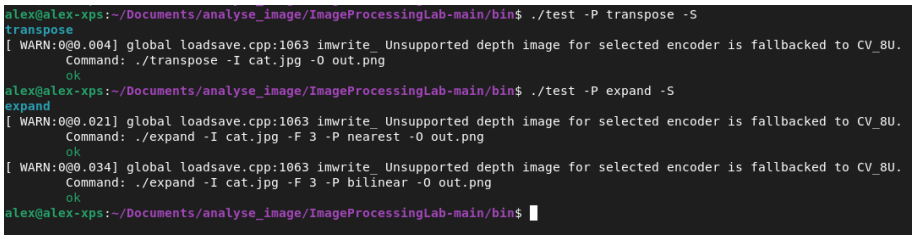

Figure 1: Tests transformation géométriques.

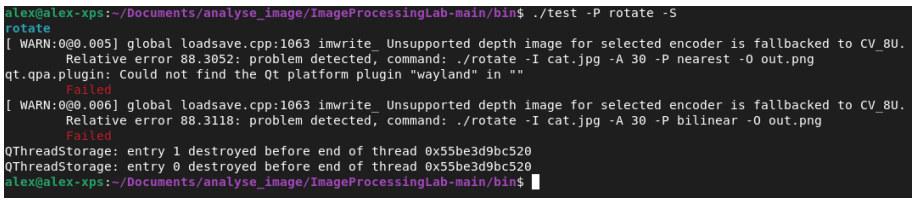

Figure 2: Test rotate.

Voici le résultat quand la fonction test du projet fourni lance "./rotate -I cat.jpg -A 30 -P nearest -O out.png" : Figure 3
Pourtant, si je lance à la main les mêmes commandes (Figure 4), j'obtiens : Figure 5 et Figure 6

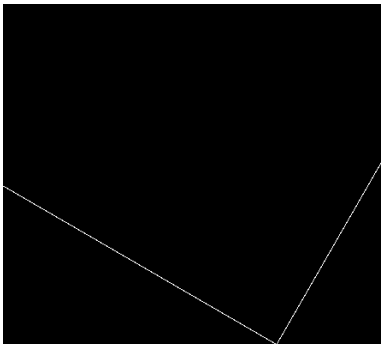

Figure 3: Result ./test.

```
alex@alex-xps:~/Documents/analyse_image/ImageProcessingLab-main/bin$ ./rotate -I cat.jpg -A 30 -P bilinear -O out.png
[ WARN:0@0.006] global loadsave.cpp:1063 imwrite_ Unsupported depth image for selected encoder is fallbacked to CV_8U.
alex@alex-xps:~/Documents/analyse_image/ImageProcessingLab-main/bin$ ./rotate -I cat.jpg -A 30 -P nearest -O out.png
[ WARN:0@0.004] global loadsave.cpp:1063 imwrite_ Unsupported depth image for selected encoder is fallbacked to CV_8U.
alex@alex-xps:~/Documents/analyse_image/ImageProcessingLab-main/bin$
```

Figure 4: My commands.

Figure 5: My out for rotate nearest.

Figure 6: My out for rotate bilinear.