

Imagerie numérique - Introduction à OpenCV

Feuille de TD N°2

ESIEA

L. Beaudoin

1 Compléments sur IplImage

1.1 ROI ou Region Of Interest

Pour certains traitements, on peut être amené à ne travailler que sur un extrait d'une image (comme dans l'exercice 353 du TD1 par exemple). OpenCV offre la possibilité de restreindre une `IplImage` à une zone (forcément rectangulaire) d'intérêt (ROI Region Of Interest). L'intérêt est qu'à chaque fois que vous travaillerez sur l'`IplImage`, elle ne sera réduite qu'à la zone d'intérêt. Les principales fonctions utiles sont :

- `void cvSetImageROI(IplImage* image, CvRect rect)`
qui permet de définir la ROI de l'image `image` délimitée par le rectangle `rect`.
- `CvRect cvGetImageROI(const IplImage* image)`
qui permet de récupérer le rectangle définissant la ROI de l'image `image`.
- `void cvResetImageROI(IplImage* image)`
qui permet de remettre la ROI sur l'ensemble de l'image `image`.

1.2 Accès au tableau des pixels

Lorsque vous définirez vos propres algorithmes de traitement d'image, vous serez amené à travailler sur le tableau des valeurs directement. Bien que l'on est déjà vu que la fonction `cvGet2D` permettait de récupérer les valeurs BGR d'un pixel, ce n'est pas cette approche, beaucoup trop lente, qu'il vous faut utiliser pour modifier les pixels d'une image. Il vous faut manipuler les pointeurs sur les données `imageData`.

En mémoire, une image est stockée en une seule ligne. Dit autrement, les lignes, du haut vers le bas, sont mises les unes à la suite des autres. Ainsi, si vous voulez aller au début de la ligne `row`, il vous suffit de partir du début de la première et d'avancer de `row` fois la taille en octets d'une ligne de l'image (pour rappel, c'est le paramètre `widthStep` de la structure `IplImage`). Si maintenant, c'est le pixel qui est situé à la colonne `col` qui vous intéresse, il vous suffit alors d'avancer depuis le début de la ligne de `col` fois la taille en octets occupée par chaque pixel, soit `nChannels` fois `sizeof(XXX)` où `XXX` est le type de chaque plan (`unsigned char`, `float`...). Enfin, le dernier écueil à éviter est qu'il est indispensable de retyper les données car par défaut `imageData` est de type `char*`, ce qui n'est pas forcément le cas de vos données !

Un exemple pour fixer les idées. Supposons que nous ayons 2 images `imgC` et `imgG` composées de 3 plans couleur d'ordre BGR et chaque plan codé en `unsigned char`. `imgG` est la conversion de l'image couleur `imgC` en noir et blanc par la formule :

$$G = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

Un code correspondant pourrait être :

```
unsigned char *pRowC, *pRowG;
for(row=0; row<imgC->height; row++){
    pRowC = (unsigned char*)(imgC->imageData + row * imgC->widthStep);
    pRowG = (unsigned char*)(imgG->imageData + row * imgG->widthStep);
    for(col=0; col<imgC->width; col++){
        /* pour le canal B de imgG */
        *(pRowG + col*imgG->nChannels*sizeof(unsigned char)) =
            (unsigned char) (*(pRowC + col*imgC->nChannels*sizeof(unsigned char)) * 0.114 +
                *(pRowC + col*imgC->nChannels*sizeof(unsigned char)+1) * 0.587 +
                *(pRowC + col*imgC->nChannels*sizeof(unsigned char)+2) * 0.299);
        /* pour le canal G de imgG */
        *(pRowG + col*imgG->nChannels*sizeof(unsigned char) + 1) =
            (unsigned char) (*(pRowC + col*imgC->nChannels*sizeof(unsigned char)) * 0.114 +
                *(pRowC + col*imgC->nChannels*sizeof(unsigned char)+1) * 0.587 +
                *(pRowC + col*imgC->nChannels*sizeof(unsigned char)+2) * 0.299);
        /* pour le canal R de imgG */
        *(pRowG + col*imgG->nChannels*sizeof(unsigned char) + 2) =
            (unsigned char) (*(pRowC + col*imgC->nChannels*sizeof(unsigned char)) * 0.114 +
                *(pRowC + col*imgC->nChannels*sizeof(unsigned char)+1) * 0.587 +
                *(pRowC + col*imgC->nChannels*sizeof(unsigned char)+2) * 0.299);
    }
}
```

1.3 Exercices

1.3.1 Conversion en Noir et Blanc

Reprendre l'exercice 3.5.3 du TD1 en effectuant la conversion en noir et blanc par la méthode précédente.

1.3.2 Poivre et Sel

- charger une image et convertissez la en noir et blanc par la fonction `cvCvtColor`,
- soient `NoisePourcentage` la variable représentant le pourcentage de pixels de l'image en noir et blanc que vous allez bruitez et `NbPixels` le nombre total de pixels de l'image. Mettre aléatoirement $(\text{NoisePourcentage} \times \text{NbPixels})/2$ pixels de l'image en niveaux de gris à 255 et $(\text{NoisePourcentage} \times \text{NbPixels})/2$ autres pixels de l'image en niveau de gris à 0. Pour générer le hasard, vous utiliserez les fonctions `srandom`, `random` et `time`. Pour un premier test, on prendra `NoisePourcentage` égal à 20%.
- déplacez vous ensuite de pixel en pixel sur l'image noir et blanc. Pour chaque pixel, prenez les valeurs des voisins de gauche et des deux voisins de droite que vous mettrez dans un tableau. Triez le tableau et prenez la valeur médiane (i.e. la case numéro de ce tableau à 5 case). Prenez ce résultat et affectez le au pixel. Pour ne pas avoir de problèmes avec les bords de l'image, vous débuterez à la colonne 3 et vous arrêterez à la colonne `imgG->width-3`.
- affichez simultanément l'image originale en niveaux de gris, l'image bruitée et l'image filtrée par la médiane. Magique ! Faites plusieurs tests avec différentes valeurs de `NoisePourcentage`.

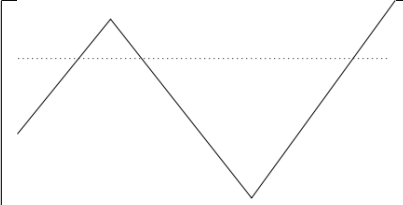
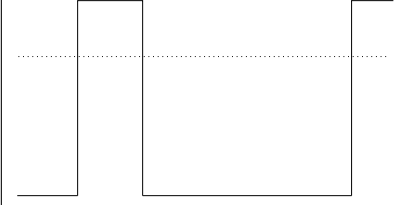
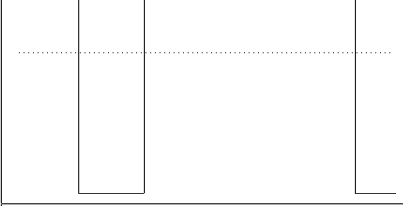
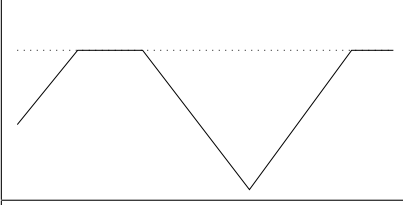
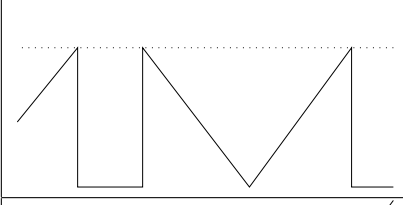
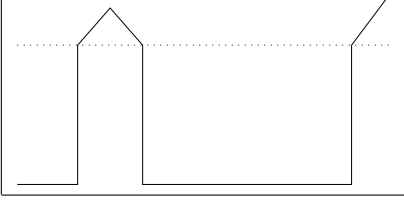
2 Seuillages

2.1 Principe

Un des traitements les plus simples sur une image consiste à effectuer un seuillage sur les valeurs des pixels. Dit autrement, le résultat d'un seuillage dépendra de la comparaison de la valeur du pixel à un seuil donné. Le seuillage est indispensable lorsque l'on souhaite binariser un résultat (i.e. décider pour chaque pixel s'il fait partie de la classe recherchée ou non). La fonction à utiliser est :

```
double cvThreshold(const CvArr* src, CvArr* dst, double threshold, double maxValue, int thresholdType)
```

où l'image `dst` est le résultat du seuillage de l'image `src`, `threshold` est la valeur du seuil, `maxValue` est la valeur qui sera affectée selon la méthode de seuillage `thresholdType` utilisée. Le tableau suivant illustre ces différentes méthodes de seuillage.

	profil d'une ligne de l'image (trait plein) et seuil (pointillés)
	CV_TRESH_BINARY
	CV_TRESH_BINARY_INV
	CV_TRESH_TRUNC
	CV_TRESH_TOZERO_INV
	CV_TRESH_TOZERO

L'utilisation des fonctions de seuillage précédentes montre une grande sensibilité aux variations des conditions d'éclairage d'une image. Pour adapter localement le seuillage, il existe une autre fonction :

```
void cvAdaptiveThreshold(const CvArr* src, CvArr* dst, double maxValue, int adaptive_method=
CV_ADAPTIVE_THRESH_MEAN_C, int thresholdType=CV_THRESH_BINARY, int blockSize=3, double param1=5)
```

où `src` est l'image à seuiller et `dst` l'image seuillée (2 images distinctes nécessairement), `maxValue` la valeur haute affectée et `adaptive_method` est la méthode de seuillage locale. `adaptive_method` vaut soit :

- `CV_ADAPTIVE_THRESH_MEAN_C` : le seuil local est calculé comme étant la moyenne des `blockSize×blockSize` voisins à laquelle on retranche la valeur `param1`. Si la valeur du pixel central est supérieure à ce seuil local, on applique le seuillage `thresholdType` (qui ne peut valoir que `CV_TRESH_BINARY` ou `CV_TRESH_BINARY_INV`),
- `CV_ADAPTIVE_THRESH_GAUSSIAN_C` : le seuil local est calculé comme étant la moyenne pondérée en fonction de la distance au centre des `blockSize×blockSize` voisins à laquelle on retranche la valeur `param1`. Si la valeur

du pixel central est supérieure à ce seuil local, on applique le seuillage `thresholdType` (qui ne peut valoir que `CV_TRESH_BINARY` ou `CV_TRESH_BINARY_INV`),

2.2 Exercice

2.2.1

Charger un image et afficher le résultat du seuillage par toutes les méthodes précédentes.