

**A**u cours de cette étude, je vous propose d'étudier le comportement de l'API OpenCV dans un programme Rust. Nous pourrons ainsi concevoir des applications de traitement d'images classiques, mais aussi de voir comment faire de la reconnaissance de forme pour identifier des objets particuliers (des voitures, des colis sur un chemin roulant, etc.), de pouvoir discriminer les lettres d'un texte sur différents type de support (plaqué minéralogique, devanture de magasin, etc.), ou reconnaître un visage, les yeux, la bouche, etc., plus d'autres traitements annexes.

Contrairement à beaucoup d'autres bibliothèques que nous avons eu l'occasion de rencontrer, l'API OpenCV est délivrée uniquement avec le code source que vous devez alors compiler dans le système d'exploitation de votre choix. La procédure d'installation complète est décrite ci-dessous.

## INSTALLATION DE L'API OPENCV

**T**out d'abord nous devons installer l'API OpenCV. Pour cela, nous devons récupérer les sources de l'API afin de les compiler après coup. Après avoir désarchivé le paquetage en format « zip », placez-vous dans le répertoire de base, créer un nouveau répertoire temporaire que vous nommerez par exemple « release ».

Vous pouvez dès lors compiler et installer votre nouvelle API. Mais attention, pour cela vous devez passer par un autre outil spécifique « CMAKE », que vous devez installer si cela n'est pas déjà fait.

Grâce cet outil « CMAKE » vous avez la possibilité de choisir votre mode de compilation et où se situera réellement votre librairie.

L'idéal est de placer cette librairie dans un répertoire souvent utilisé « /opt » par exemple. Du coup, si vous disposez de plusieurs comptes, vous trouverez toujours la bibliothèque « OpenCV » au même emplacement.

Voici ci-dessous les commandes à suivre pour l'installation complète :

```
~$ sudo apt install cmake
```

```
~$ cd opencv-4.4.0/
~/opencv-4.4.0$ mkdir release
~/opencv-4.4.0$ cd release/
```

```
~/opencv-4.4.0/release$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/opt/opencv ...
// N'oubliez pas les deux points.
~$ make
~$ sudo make install
```

Une fois que la bibliothèque OpenCV est définitivement compilée et installée, nous la retrouvons bien dans le répertoire « /opt ». Pour tous les projets que nous ferons ultérieurement, il est important de bien noter sa localisation.

## UTILISER OPENCV DANS UNE RASPBERRY

**L**orsque nous utilisons une caméra et que nous devons réaliser des analyses d'images particulières comme pour récupérer le numéro d'une plaque minéralogique d'une voiture qui vient de passer devant la caméra, nous pouvons fabriquer ce dispositif avec un système numérique de type Raspberry.

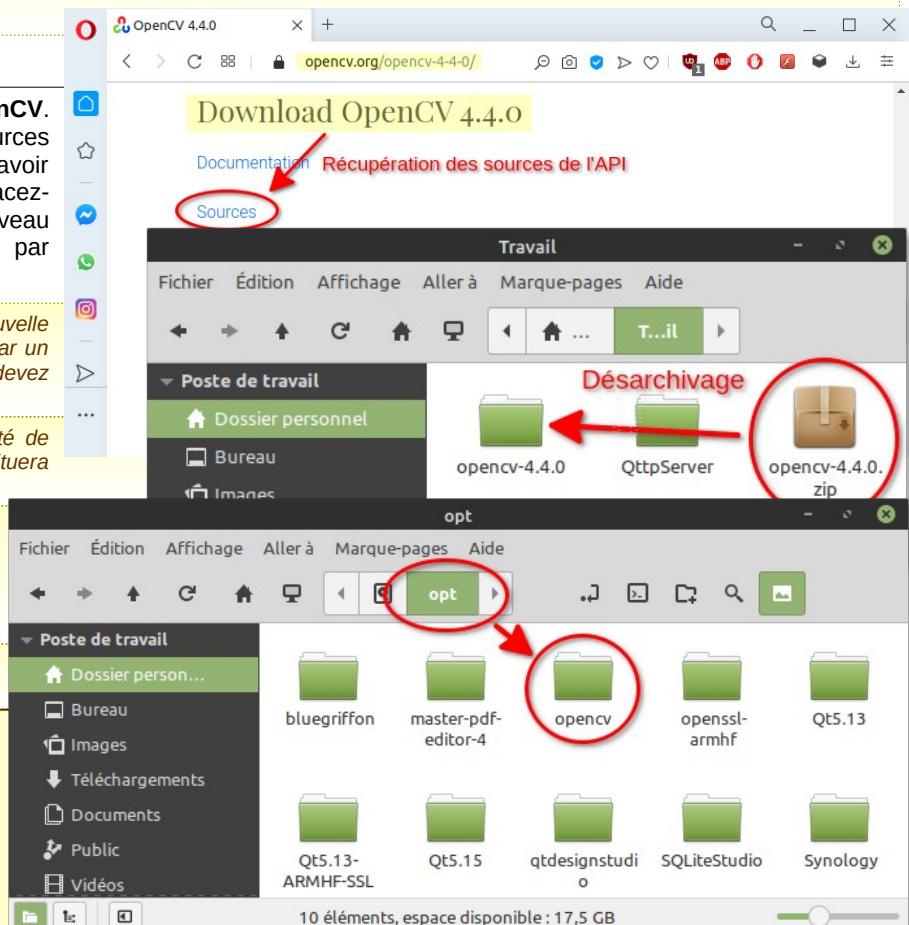
L'idéal, dans ce genre de système, c'est que la **Raspberry** soit juste un système numérique communiquant sans clavier ni écran, comme devrait l'être tout **nano-PC** dont l'objectif principal est généralement de capturer des informations à partir de capteurs câblés et piloter éventuellement des actionneurs pour par exemple démarrer le lave-linge à distance depuis son smartphone.

Le capteur qui nous intéresse ici, est une simple webcam qui va nous permettre d'avoir une vision à distance. En effet, nous pourrons nous servir de notre PC pour consulter à distance une caméra connecté à cette **Raspberry**. Pour que cela fonctionne correctement, la librairie **OpenCV** doit être installée directement dans la **Raspberry**.

Attention, dans ce cadre là, vu que la **Raspberry** ne peut être utilisée qu'à distance et qu'elle dispose d'un processeur ARM totalement différent de ceux qui existent dans un PC, il est plus judicieux de réaliser une suite de développement en compilation croisée.

C'est-à-dire que la conception du programme se fait sur un PC normal en utilisant un compilateur prévu pour un autre type de processeur en association avec une librairie OpenCV prévue également pour cet autre type de processeur. Le programme ainsi constitué sera ensuite déployé sur la **Raspberry** cible qui sera ensuite exécutée à distance pour remplir sa fonction.

Pour réaliser tout ce processus, nous devons construire deux nouvelles librairies OpenCV, une pour la **Raspberry** afin qu'elle soit adaptée au processeur **ARMv7**, et une autre placée dans le **PC** de développement qui nous permet de réaliser nos programmes avec les outils que nous connaissons bien.



**OpenCV pour ARMv7 côté PC de développement** : côté PC, pour réaliser cette compilation croisée, nous devons disposer du bon compilateur adapté aux processeurs de type **ARMv7**. Installez-le :

```
~$ sudo apt-get install crossbuild-essential-armhf
```

Ensuite, nous réutilisons les mêmes sources **OpenCV** qui nous ont servis à créer notre librairie normale directement pour le PC, mais cette fois-ci la configuration sera bien entendu différente pour prendre en compte la compilation croisée d'une part, et les instructions spécifiques de la Raspberry d'autre part.

En effet, les dernières **Raspberry** intègrent des processeurs plus performants qui possèdent des instructions spécifiques de type **NEON** et **VFPV3**. Voici ci-dessous les commandes à réaliser pour la mise en œuvre de tout le processus :

```
~$ cmake -D CMAKE_TOOLCHAIN_FILE=../platforms/linux/arm-gnueabi.toolchain.cmake
      -D ENABLE_VFPV3=ON -D ENABLE_NEON=ON -D CMAKE_INSTALL_PREFIX=/opt/opencv-armhf ..
~$ make -j4
~$ sudo make install
```

**OpenCV dans la Raspberry** : Pour la Raspberry, vous devez également récupérer les sources **OpenCV** et réaliser votre compilation afin d'obtenir les binaires respectifs directement à l'intérieur qui seront exploités lorsque les programmes de traitement d'images seront déployés. Là aussi, **OpenCV** doit pouvoir prendre en compte toutes les compétences du processeur utilisé.

Attention, vu que le développement des applications se fait par compilation croisée, vous devez préciser au système d'exploitation le chemin où se trouve les librairies **OpenCV** ou les déplacer dans un répertoire déjà connu, comme « **/usr/lib** ».

```
~$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D ENABLE_VFPV3=ON -D ENABLE_NEON=ON
      -D CMAKE_INSTALL_PREFIX=/opt/opencv-armhf ..
~$ make -j4
~$ sudo make install
~$ sudo cp /opt/opencv-armhf/libopencv* /usr/lib
```

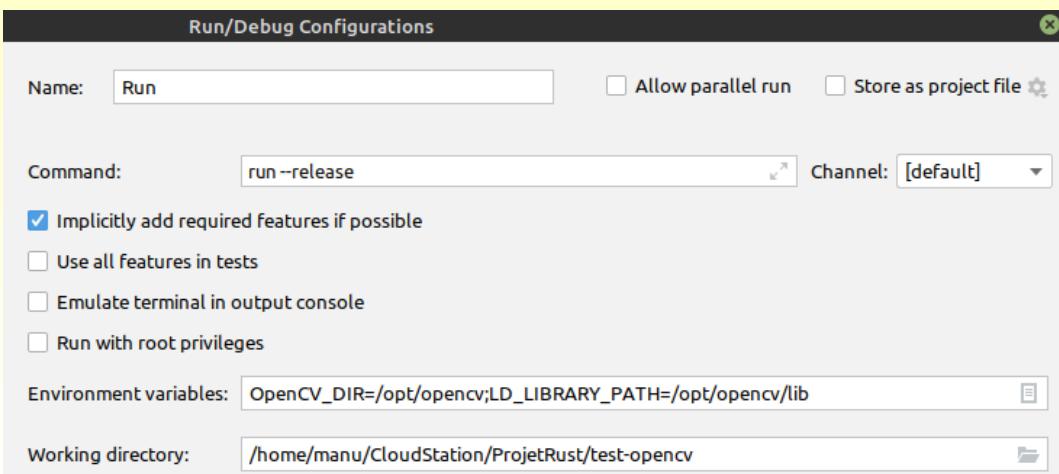
## FINALISATION DE L'INSTALLATION POUR RÉALISER DES APPLICATIONS AVEC RUST

Gâce aux installations précédentes, nous possédons maintenant la librairie **OpenCV** parfaitement opérationnelle. Toutefois, vu que nous sommes passé par « **make** » et « **make install** », l'API a été construite dans le langage **C++**. Afin de pouvoir exploiter les compétences de cette bibliothèque, il est nécessaire d'installer des outils supplémentaires qui permettront de transcrire les programmes **Rust** vers les modules **C++**.

Les deux outils à installer s'appellent **clang** et **libclang-dev** qui sont des compilateurs **C++**, ce qui veut dire que les programmes que nous réaliserons en relation avec **OpenCV** seront en réalité transcrit en langage **C++**.

```
~$ sudo apt install clang libclang-dev
```

Un dernier réglage à effectuer pour produire des programmes **Rust** avec analyses d'images consiste à spécifier où se situe exactement la librairie **OpenCV** à l'aide de deux variables d'environnement **OpenCV\_DIR** et **LD\_LIBRARY\_PATH**.



Nous allons maintenant pouvoir réaliser nos premiers programmes en **Rust**. Ils seront identiques à ceux que j'ai déjà proposé dans le langage **C++**. Vous remarquerez d'ailleurs que le nom des fonctions et des classes utilisées sont exactement les mêmes.

## Classe de base pour le traitement des images

Avant de nous lancer dans notre premier projet et afin de valider notre installation, nous devons prendre un peu de temps pour évoquer la classe indispensable à tout traitement d'image et de la visualisation des photos sur lesquelles nous allons agir. Cette classe s'appelle **Mat**. Elle factorise tout le contenu de l'ensemble de la photo, c'est-à-dire l'ensemble des octets formant chacun des pixels de l'image.

Cette classe **Mat** stocke la totalité de la photo à traiter quelque soit le format utilisé. Nous pouvons choisir le format d'image. Par défaut, chaque pixel peut être décomposé à l'aide des trois couleurs fondamentales « **rouge, vert, bleue** » (**RGB**) chacune codée sur un octet. Il est aussi possible d'intégrer un octet supplémentaire pour représenter la transparence, nommée couche « **alpha** ».

Nous pouvons choisir un autre type de format, là aussi codé sur trois octets, mais cette fois-ci exprimée sous la forme « **teinte, saturation, valeur** » (**HSV**). Il existe bien d'autres formats, comme (**YUV**) par exemple qui représente l'ancien format vidéo **PAL**. Enfin, nous pouvons aussi travailler en niveau de gris.

Cette classe est indispensable pour tous les traitements d'images que vous désirez réaliser avec **OpenCV**. Elle représente une « **matrice** » qui stocke l'ensemble des octets constituant votre photo. Par contre, contrairement à d'autres systèmes, elle ne dispose pas de méthodes spécifiques pour réaliser ces traitements. Pour cela, nous pouvons passer par des fonctions annexes qui prennent en compte une ou plusieurs de ces matrices, comme par exemple :

- **imread()** : pour récupérer l'image depuis un fichier.
- **imwrite()** : pour sauvegarder l'image dans un fichier.
- **blur()** : qui floute votre image.
- **cvtColor()** : qui permet de changer de format d'image.
- **split()** : qui permet de dissocier les types de couleurs et fournit des tableaux d'octets séparés.
- **merge()** : fait l'opération inverse et permet de fusionner des tableaux séparés en une seule matrice.
- **equalizeHist()** : qui rétablit un histogramme équilibré dans le cas où votre photo est sous-exposée ou sur-exposée. (et plein d'autres)

Attention, le format de stockage par défaut de cette classe **Mat** est **BGR** et non pas **RGB** comme c'est le cas dans beaucoup d'autres API. Nous devrons donc convertir systématiquement les formats pour passer d'une classe à l'autre. Les constantes de format d'image ont l'ossature suivante : **CV\_<profondeur><type>C<canaux>** :

- **<profondeur>** : peut être remplacé par une des valeurs **8, 16, 32 ou 64** qui représentent le nombre de bits pour chaque pixel et par canal.
- **<type>** : peut être remplacé par les valeurs **U, S ou F** qui représentent respectivement les types (**unsigned integer, signed integer et floating**).
- **<canaux>** : représente le nombre de canaux par pixel.

Ainsi, pour représenter une image au format standard **BGR**, la constante est : **CV\_8UC3**.

## AFFICHER UNE IMAGE ISSUE D'UN FICHIER

À près tout ce préambule, nous allons étudier les différentes fonctionnalités de cette librairie **OpenCV** au travers d'exemples toujours relativement simples afin de prendre le temps, et petit à petit de découvrir les fonctions et les classes utiles pour résoudre et discriminer les différentes tâches à implémenter pour arriver à la solution souhaité.

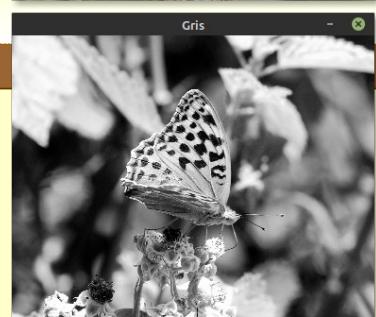
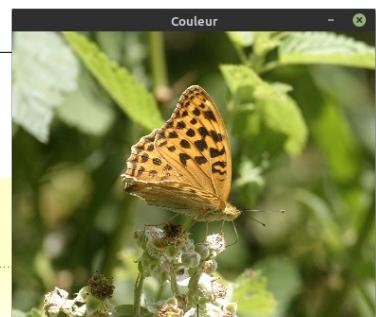
Pour l'instant, nous nous cantonnons sur du traitement d'une seule image, sachant que lorsque que nous aborderons la vidéo, les algorithmes proposés seront identiques, puisque une vidéo est tout simplement une suite d'images.

Ainsi, notre tout premier programme consiste à récupérer un fichier image et à l'afficher dans une fenêtre spécifique, en couleur d'une part et en niveau de gris d'autre part. Nous en profiterons pour vérifier le nombre d'octets constituant l'image dans ces deux cas de figure.

### Cargo.toml

```
[package]
name = "test-opencv"
version = "0.1.0"
edition = "2021"

[dependencies]
# sudo apt install cmake
# cd opencv-4.5.5, mkdir release, cd release
# cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/optopencv ...
# make -j4
# sudo make install
# OpenCV_DIR=/optopencv;LD_LIBRARY_PATH=/optopencv/lib
# sudo apt install clang libclang-dev
opencv = "0.63.0"
```



Après toute cette phase d'installation, pour réaliser vos programmes, le premier élément indispensable pour débuter consiste à bien intégrer le module associé à la librairie OpenCV.

### main.rs

```
use opencv::{
    highgui::*,
    imgcodecs::*, 
    prelude::*,
    Result
};

fn main() -> Result<()> {
    let couleur = imread("/home/manu/Images/Papillon.jpg", IMREAD_REDUCED_COLOR_2)?;
    let gris = imread("/home/manu/Images/Papillon.jpg", IMREAD_REDUCED_GRAYSCALE_2)?;
```

```

imshow("Couleur", &couleur)?;
imshow("Gris", &gris)?;
wait_key(0)?;

println!("Couleur : {} octets", couleur.total()*couleur.elem_size());
println!("Gris : {} octets", gris.total()*gris.elem_size());

Ok(())
}

```

### Résultats en mode console

Couleur : 983040 octets  
 Gris : 327680 octets

Pour récupérer une image à partir d'un fichier, vous utilisez la fonction `imread()` (pour image read) en spécifiant le chemin du fichier image et une constante qui permet de choisir si nous désirons une image en couleur ou en niveau de gris avec éventuellement un redimensionnement automatique par 2, 4 ou 8.

Cette fonction ne peut être utilisée que si nous intégrons le module « `imgcodecs` » qui permet ainsi de transformer une image compressée au format **JPEG** vers une image classique exploitable en mémoire de type **BMP**.

Cette fonction renvoie systématiquement une matrice au travers d'un objet de type **Mat**. À partir de là, il est possible d'utiliser des méthodes qui permettent de connaître le nombre de pixels de l'image récupérée grâce à la méthode `total()` et le nombre d'octets utilisé par pixel avec cette fois-ci la méthode `elem_size()`.

Dans le résultat obtenu en mode console, nous remarquons que la couleur prend trois fois plus d'octets qu'avec une image en niveau de gris, puisque chaque pixel exprime alors les trois couleurs fondamentales **BGR** avec un octet pour chaque couleur.

Pour afficher l'image récupérée, nous utilisons la fonction `imshow()` (pour image show) en spécifiant le nom de la fenêtre de visualisation et la référence de la matrice à visualiser. Il faut cette fois-ci intégrer le module « `highgui` ».

La fonction `waitkey()`, comme son nom l'indique permet d'attendre que l'utilisateur tape sur une touche du clavier et sorte ainsi du programme. Cela nous laisse le temps de contrôler le résultat des photos affichées. Dans cette fonction, il est possible de proposer une attente spécifique exprimée en millisecondes, la valeur **0** correspond à une attente infinie.

Afin d'éviter d'utiliser systématiquement l'appel aux méthodes `unwrap()` pour chacune des actions à réaliser lors de vos traitements spécifiques, je propose de déléguer la gestion des erreurs éventuelles à la fonction `main()` qui retourne un `Result<()>`. Pour cela, vous utilisez l'opérateur `?` à chaque fonction qui nécessite une vérification du résultat. Le code est ainsi plus concis.

## REDIMENSIONNER L'IMAGE ET AJUSTEMENT DE L'HISTOGRAMME

Dans le chapitre précédent, nous avons vu qu'il été possible de choisir la taille de l'image au moment de la récupération du fichier, sauf que les tailles proposées sont figées avec des valeurs en sous-multiple de **2**.

Il existe une fonction `resize()` qui nous permet de choisir la proportion souhaité (**dimension relative**) à partir du fichier image récupérée en taille normale ou alors de fixer précisément la taille de l'image en largeur et en hauteur (**dimension absolue**).

L'ajustement de l'histogramme permet de contraster l'image originale pour qu'elle soit moins fade. Pour cela, il s'agit d'élargir l'histogramme pour qu'il prenne le maximum de largeur possible afin que les ombres soient plus profondes et que les hautes lumières soient plus intenses, le tout en respectant les différentes nuances dans l'ensemble du spectre. Cela se fait au moyen de la fonction `equalize_hist()`.

### main.rs

```

use opencv::{ highgui::*, imgcodecs::*, core::*, imgproc::*, Result };

fn main() -> Result<()> {
    // Récupérer l'image en taille normale
    let mut gris = imread("/home/manu/Images/Papillon.jpg", IMREAD_GRAYSCALE)?;
    let mut redim = Mat::default();

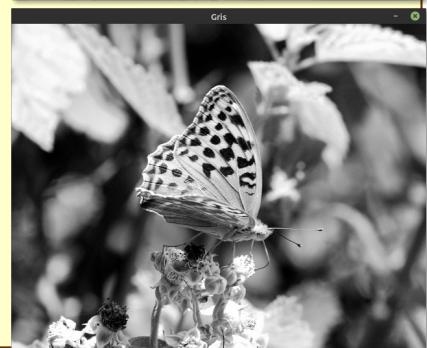
    // Redimensionner l'image à 70% de la version originale
    resize(&gris, &mut redim, Size{width: 0, height: 0}, 0.7, 0.7, INTER_CUBIC)?;
    imshow("Gris", &redim)?;
    wait_key(0)?;

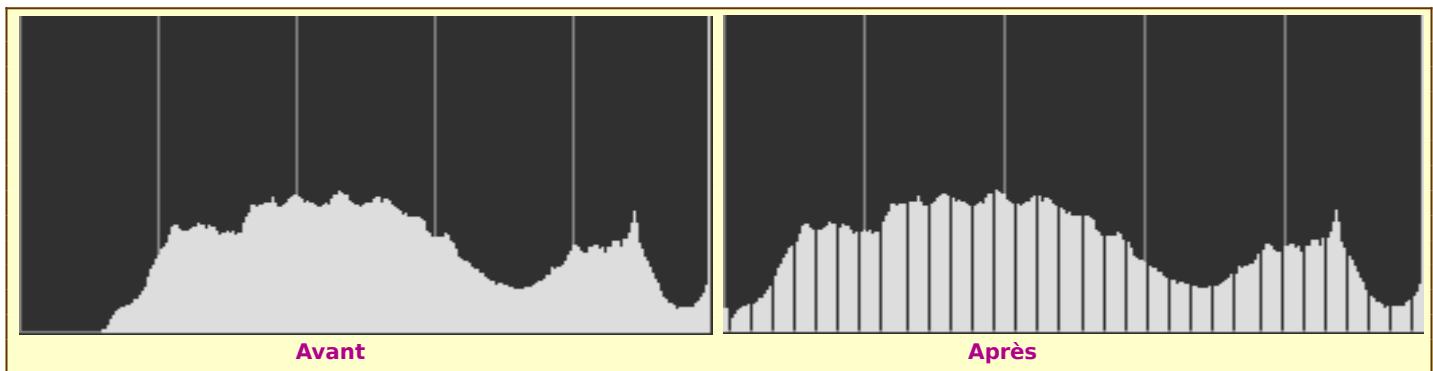
    // Ajuster l'histogramme pour renforcer le contraste
    equalize_hist(&redim, &mut gris)?;
    imshow("Gris", &gris)?;
    wait_key(0)?;

    // Redimensionner l'image en 640x480
    resize(&gris, &mut redim, Size{width: 640, height: 480}, 0., 0., INTER_CUBIC)?;
    imshow("Gris", &redim)?;
    wait_key(0)?;

    Ok(())
}

```





Beaucoup de fonctions de **OpenCV** prennent en arguments deux matrices, une pour lire les octets à transformer et une pour donner le résultat souhaité. Contrairement au langage **C++** qui autorise à prendre une seule matrice, **Rust** de par son fonctionnement de base interdit cette pratique.

Je rappelle que dans **Rust**, une seule et unique variable peut manipuler une donnée à un instant précis. Lorsqu'une matrice est empruntée pour un argument de la fonction, elle ne peut pas être également empruntée pour un autre argument. Il faut donc prévoir une autre matrice pour cela. Dans notre code, nous passons alternativement d'une matrice mutable à l'autre.

La fonction **resize()** peut être utilisée de deux façons différentes, soit nous spécifions un redimensionnement relatif en pourcentage, soit nous proposons des dimensions avec des valeurs absolues. Comme en **Rust** il n'existe pas de paramètre par défaut comme c'est le cas en **C++**, vous proposez des valeurs nulles sur les arguments à ne pas prendre en compte. Pour créer une nouvelle matrice vierge, vous utilisez la méthode statique **default()**.

Le redimensionnement effectue un échantillonnage de l'image originale. Afin d'obtenir un résultat, soit le plus parfait possible, soit le plus rapide possible, vous devez choisir une constante en conséquence. **INTER\_CUBIC** choisit un algorithme sophistiqué avec certainement le meilleur résultat d'échantillonnage de l'image finale.

Pour proposer un dimensionnement absolu, nous disposons d'une structure générique **Size{}** avec deux attributs **width** et **height** que vous renseignez en choisissant des valeurs avec les types de votre convenance.

La fonction **equalize\_hist()** est extrêmement performante, avec une utilisation des plus simples, sachant qu'elle s'occupe essentiellement de la répartition de la luminosité sur l'ensemble du spectre. Attention, elle ne fonctionne que sur un seul canal, ce qui convient parfaitement pour une image en niveau de gris.

Si vous souhaitez utiliser cette fonction **equalize\_hist()** avec une photo en couleur, elle n'est plus du tout adaptée pour les modes **BGR** ou **RGB** où la notion de luminosité n'apparaît pas. Il faut alors changer de mode de couleurs afin que cette notion de luminosité corresponde à un canal spécifique, ce que je vous propose de découvrir dans le chapitre qui suit.

## Choisir son mode de couleurs

Le mode de couleurs le plus utilisé en informatique est le mode **RGB** (Red, Green, Blue) qui spécifie que pour chaque pixel de l'image, nous proposons un niveau de valeur pour chacune des trois couleurs fondamentales, chaque couleur étant codée sur un octet. Finalement, chaque pixel, avec ce principe, peut représenter 16 millions de couleurs différentes.

Si le modèle **RGB** se révèle bien adapté à la représentation de la couleur en informatique, il est en revanche assez éloigné de la perception que nous avons des couleurs. En effet, nos yeux ne perçoivent pas les couleurs comme une somme de rouge, de vert et de bleu, mais plutôt comme une sensation de luminosité correspondant à l'intensité de la lumière ; on définit des objets plus ou moins clairs ou plus ou moins foncés.

A cette notion de **luminance**, il faut également rajouter une information de coloration, ce qu'on appelle la **chrominance**, définie à la fois par la **teinte** (la couleur) et la **saturation** (pureté de la teinte).

### Le modèle Teinte Saturation Luminosité

Le modèle **TSL** (ou **HSV** en anglais pour Hue, Saturation et Value) est un modèle colorimétrique perceptuel car il se rapproche fortement de la perception physiologique de la couleur par l'œil humain.

Dans ce système, les couleurs sont toujours caractérisées par trois dimensions mais qui ont une signification tout autre que dans le modèle **RGB**, puisqu'elles représentent ici la **teinte**, la **saturation** et la **luminosité**. Nous représentons généralement le modèle **TSL** à l'aide de deux cônes inversés placés l'un au-dessus de l'autre (figure page suivante).

La **teinte** qui correspond à la perception de la couleur est mesurée sur une échelle circulaire ( cercle de chromaticité de Newton) par un angle de 0° à 360°.

La **saturation** mesure le degré de pureté d'une couleur, c'est-à-dire la quantité de gris ajoutée ou pas à la couleur. Elle est représentée par le rayon d'une section circulaire du cône et varie de 1 (couleur pure ou saturée) à 0 (niveau de gris correspondant).

La **luminosité** représente le degré d'éclaircissement ou d'assombrissement d'une couleur. Elle est définie selon une échelle linéaire allant de 0 (noir) à 1 (blanc) en passant par tous les niveaux de gris. Sur la figure ci-dessus, la luminosité est représentée par la droite joignant les deux sommets des cônes et passant par le centre du cercle chromatique. Les teintes sur le cercle chromatique sont donc toutes au même niveau de luminosité de 50%.

### Choisir le modèle Teinte Saturation Luminosité pour sélectionner une couleur particulière

Si nous désirons manipuler une couleur particulière qui ne fait pas partie des couleurs fondamentales, il est plus judicieux de travailler directement sur la teinte en prenant ce modèle **HSV**.

Afin de pouvoir définir les conditions sur la teinte, nous nous servons du cercle chromatique. En effet, la teinte en HSV est codée en fonction de l'angle sur le cercle chromatique. Ainsi le rouge sera proche de l'angle  $0^\circ$  ou  $360^\circ$ , le vert aux alentours des  $120^\circ$  et le bleu vers  $240^\circ$ . Pour le jaune, c'est autour des  $60^\circ$ .

Voici ci-dessous les fourchettes à connaître :

\* **rouge** :  $0$  à  $20$  et  $330$  à  $360$ .

\* **orange** :  $20$  à  $40$ .

\* **jaune** :  $40$  à  $80$ .

\* **vert** :  $80$  à  $160$ .

\* **cyan** :  $160$  à  $200$ .

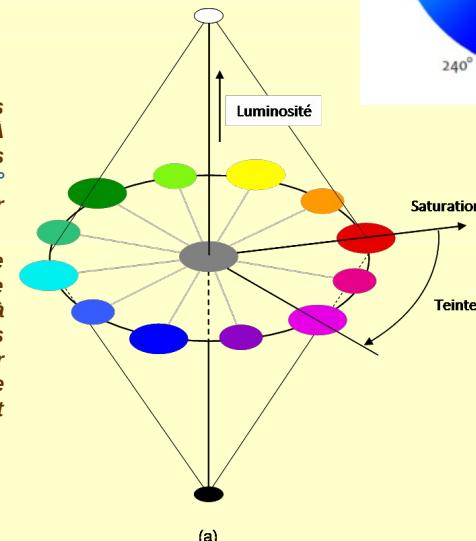
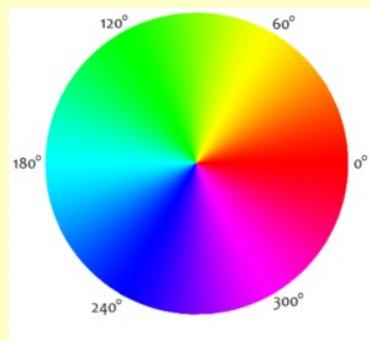
\* **bleu** :  $200$  à  $260$ .

\* **magenta** :  $260$  à  $330$ .

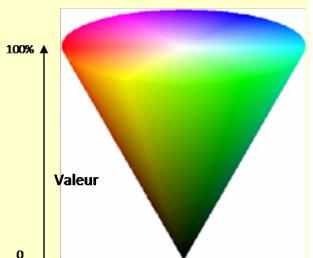
### HSV dans OpenCV

Attention, le modèle HSV est codé sur trois octets comme pour les autres modèles tel que BRG. À cause de ce principe là, pour la **teinte**, il n'est pas possible de proposer une valeur normale de  $360^\circ$  par exemple puisque nous dépasserions la valeur limite de  $255$ .

Pour que nous puissions respecter les limites de l'octet, il a été choisi de diviser la valeur normale par deux. Du coup, la fourchette établie va de  $0$  à  $180^\circ$ . Ainsi, la zone du jaune par exemple, est alors comprise entre les valeurs de **teinte**  $20$  et  $40$ . Pour la **saturation** et la **luminosité** les valeurs de réglage restent habituelles pour l'octet puisque elles sont comprises de  $0$  à  $255$ .



(a)



(b)

## AJUSTER L'HISTOGRAMME SUR UNE IMAGE EN COULEUR

**J**e vous propose de reprendre le sujet du projet précédent en proposant également un ajustement de l'histogramme mais cette fois-ci sur une image en couleur. Je rappelle que la fonctionnalité de l'ajustement de l'histogramme ne fonctionne que sur un seul canal, sachant qu'une image en couleur possède systématiquement trois canaux.

L'ajustement de l'histogramme n'a de sens en photographie que pour compenser une mauvaise exposition de la prise de vue. Soit la photo est sur-exposée, soit sous-exposée. L'exposition concerne uniquement la luminosité sur chacun des pixels de votre photo. Pour réaliser le traitement souhaité, nous devons donc changer de modèle de couleurs en basculant du modèle BGR vers le modèle HSV.

Nous profiterons de l'occasion pour savoir séparer chacun des canaux et les fusionner également par la suite. Attention, pour bien visualiser les différents réglages, nous devons toujours être dans le modèle BGR.

Dans le codage qui suit, je sépare les différents traitements au travers de méthodes spécifiques associées à une structure qui factorise les éléments importants de l'implémentation.

### main.rs

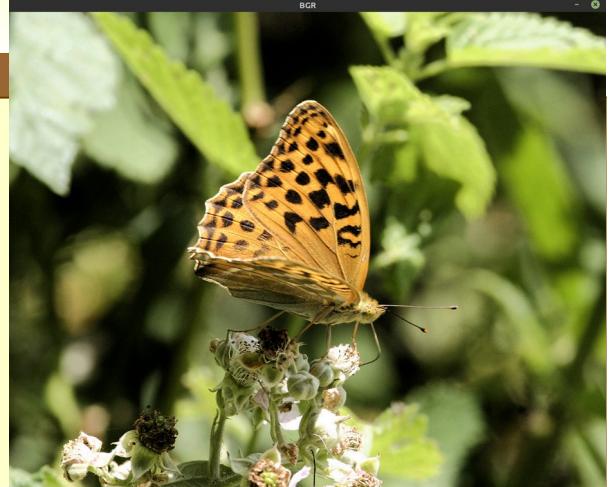
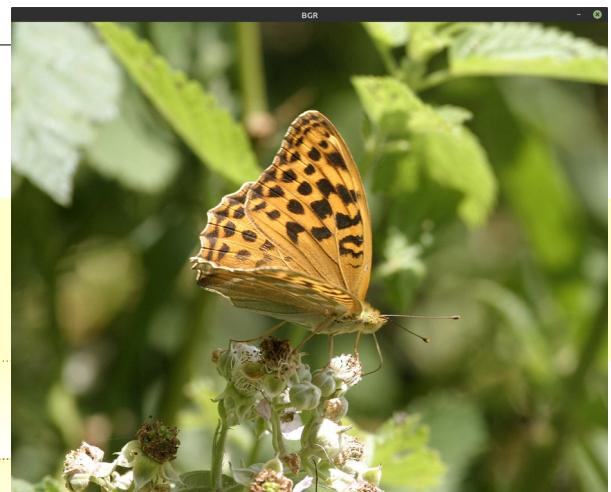
```
use opencv::highgui::*;
use opencv::imgcodecs::*;
use opencv::core::*;
use opencv::imgproc::*;
use opencv::Result;
use std::vec::VectorOfMat;

fn main() -> Result<()> {
    let mut matrices =
        Matrices::recuperation("/home/manu/Images/Papillon.jpg");
    matrices.gerer_hsv_canaux()?;
    matrices.ajuster_histogramme_luminosite()?;
    Ok(())
}

struct Matrices {
    bgr: Mat,
    hsv: Mat,
    canaux: VectorOfMat
}

impl Matrices {
    fn recuperation(&self) {
        self.bgr = Mat::load("Papillon.jpg").unwrap();
        self.hsv = self.bgr.cvtColor(COLOR_BGR2HSV);
    }

    fn gerer_hsv_canaux(&mut self) {
        self.canaux = VectorOfMat::new();
        self.canaux.push_back(self.hsv.get_chann
    }
}
```



```

fn recuperation(fichier: &str) -> Matrices {
    let original = imread(fichier, IMREAD_COLOR).unwrap();
    let mut bgr = Mat::default();
    resize(&original, &mut bgr, Size_{width:0, height:0}, 0.75, 0.75, INTER_CUBIC).unwrap();
    imshow("BGR", &bgr).unwrap();
    wait_key(0).unwrap();
    Matrices {
        hsv: Mat::default(),
        canaux: VectorOfMat::new(),
        bgr
    }
}

fn generer_hsv_canaux(&mut self) -> Result<()> {
    cvt_color(&self.bgr, &mut self.hsv, COLOR_BGR2HSV, 0)?;
    split(&self.hsv, &mut self.canaux)?;
    Ok(())
}

fn ajuster_histogramme_luminosite(&mut self) -> Result<()> {
    let mut luminosite = Mat::default();
    equalize_hist(&self.canaux.get(2)?, &mut luminosite)?;
    self.canaux.set(2, luminosite)?;
    merge(&self.canaux, &mut self.hsv)?;
    cvt_color(&self.hsv, &mut self.bgr, COLOR_HSV2BGR, 0)?;
    imshow("BGR", &self.bgr)?;
    wait_key(0)?;
    Ok(())
}
}

```

La structure **Matrices** factorise deux matrices représentant respectivement les deux modèles de coloration **BGR** et **HSV**. Cette structure possède également les trois canaux du modèle **HSV** qui sera utile par la suite pour le troisième canal afin de s'occuper de l'histogramme de la luminosité. L'ensemble de ces canaux est représentée par la structure **VectorOfMat** qui n'est autre que l'équivalent de **Vec<Mat>**.

La première méthode **recuperation()** est une méthode statique qui permet d'implémenter la structure en spécifiant le fichier image qui servira aux différents traitements. À l'issue de cette phase, nous obtenons notre première matrice **BGR** avec une image à 75 % de la dimension du format original.

Dans cette méthode, nous utilisons les fonctions que nous connaissons déjà. Pour chacune d'entre elles, nous faisons systématiquement appel à la méthode **unwrap()** sachant que normalement, nous n'aurons jamais aucun problème. Par contre, nous ne pouvons pas utiliser l'opérateur ? Puisqu'il faudrait que la méthode renvoie **Result<()>** ce qui n'est pas possible ici.

La deuxième méthode **generer\_hsv\_luminosite()** permet de changer de mode de couleurs et de séparer chacun des canaux obtenus. Pour changer de mode, vous passez par la fonction **cvt\_color()** qui prend en arguments la matrice source et la matrice destination. Vous devez également, au travers d'une constante, spécifier le changement de modèle de couleurs.

Pour la séparation des différents canaux, vous utilisez la fonction **split()** avec pour arguments, la matrice de couleurs choisie et le vecteur de plusieurs matrices, sachant que chaque canal représente effectivement une matrice particulière, donc ici, une matrice pour la **teinte**, une matrice pour la **saturation** et une matrice pour la **luminosité**.

La dernière méthode **ajuster\_histogramme\_luminosite()** permet d'utiliser la fonction **equalize\_hist()** sur le dernier canal. Une fois que cette procédure est réalisée, nous fusionnons les trois canaux pour avoir le modèle **HSV** modifié au travers de la fonction **merge()**. Nous transformons ensuite ce modèle pour retrouver le modèle **BGR** avec que la nouvelle image puisse être visualisée.

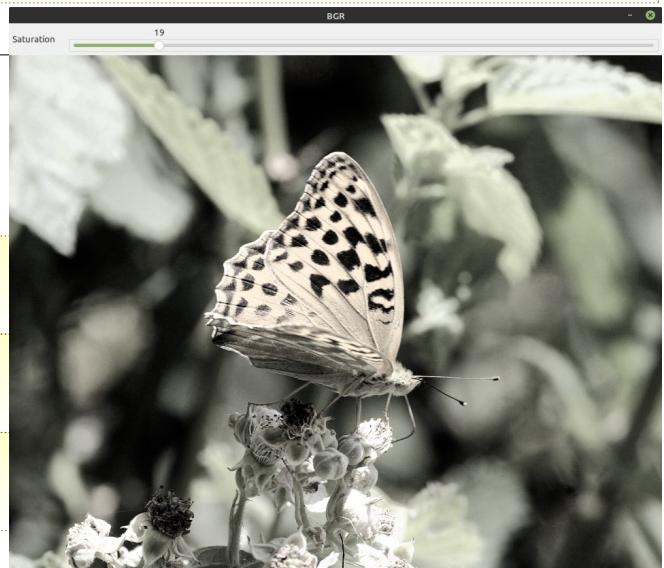
## AGIR SUR LA SATURATION DE L'IMAGE

Je vous propose de compléter le projet précédent afin que nous puissions découvrir ce qui se passe lorsque nous intervenons sur le canal de la saturation. Nous pouvons par exemple montrer que lorsque la saturation est nulle, votre photo est alors en niveau de gris. Nous pouvons aussi dans l'autre sens proposer de rehausser les couleurs pour avoir un résultat plus chatoyant.

Nous allons juste rajouter une nouvelle méthode dans le code précédent qui nous permettra de profiter de la séparation des canaux déjà réalisés.

Nous profitons également de ce sujet pour voir comment intégrer une barre de réglage qui permettra de choisir notre niveau de saturation de 0 à 130 %.

Grâce à ce nouveau dispositif, nous allons voir comment implémenter du calcul matriciel avec des relations simples, en utilisant les opérateurs de calculs arithmétiques classiques (\*, /, +, -).



## main.rs

```

use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*, Result };
use opencv::types::VectorOfMat;

fn main() -> Result<()> {
    let mut matrices = Matrices::recuperation("/home/manu/Images/Papillon.jpg");
    matrices.gerer_hsv_canaux()?;
    matrices.ajuster_histogramme_luminosite()?;
    matrices.reglage_saturation()?;
    Ok(())
}

struct Matrices {
    bgr: Mat,
    hsv: Mat,
    canaux: VectorOfMat
}

impl Matrices {
    fn recuperation(fichier: &str) -> Matrices {
        let original = imread(fichier, IMREAD_COLOR).unwrap();
        let mut bgr = Mat::default();
        resize(&original, &mut bgr, Size {width:0, height:0}, 0.75, 0.75, INTER_CUBIC).unwrap();
        imshow("BGR", &bgr).unwrap();
        wait_key(0).unwrap();
        Matrices {
            hsv: Mat::default(),
            canaux: VectorOfMat::new(),
            bgr
        }
    }

    fn gerer_hsv_canaux(&mut self) -> Result<()> {
        cvt_color(&self.bgr, &mut self.hsv, COLOR_BGR2HSV, 0)?;
        split(&self.hsv, &mut self.canaux)?;
        Ok(())
    }

    fn ajuster_histogramme_luminosite(&mut self) -> Result<()> {
        let mut luminosite = Mat::default();
        equalize_hist(&self.canaux.get(2)?, &mut luminosite)?;
        self.canaux.set(2, luminosite)?;
        merge(&self.canaux, &mut self.hsv)?;
        cvt_color(&self.hsv, &mut self.bgr, COLOR_HSV2BGR, 0)?;
        imshow("BGR", &self.bgr)?;
        wait_key(0)?;
        Ok(())
    }

    fn reglage_saturation(&mut self) -> Result<()> {
        let mut sat = 100;
        let mut original = Mat::default();
        self.canaux.get(1)?.copy_to(&mut original)?;
        create_trackbar("Saturation", "BGR", Some(&mut sat), 130, None)?;

        loop {
            let calcul = (&original * sat as f64 / 100.).into_result()?;
            let saturation = calcul.to_mat()?;
            self.canaux.set(1, saturation)?;

            merge(&self.canaux, &mut self.hsv)?;
            cvt_color(&self.hsv, &mut self.bgr, COLOR_HSV2BGR, 0)?;
            imshow("BGR", &self.bgr)?;
            if wait_key(30)? == 13 { break; }
        }
        Ok(())
    }
}

```

Dans ce chapitre, nous nous attardons bien entendu sur la dernière méthode **reglage\_saturation()** puisque le reste du code est similaire à l'étude précédente. L'objectif de cette méthode est de mettre en place une barre de réglage pour choisir l'intensité de la saturation. L'objectif est de choisir un pourcentage de la saturation originale que vous devez conserver pendant tout le processus de vérification, grâce à la matrice appelée **original**.

La génération de cette barre de réglage s'effectue au moyen de la fonction **create\_trackbar()** avec pour argument le titre associé à cette barre, dans quelle fenêtre elle doit apparaître, quelle est la référence à la variable entière qui prend en compte ce réglage

(ici **sat**), la valeur maxi à ne pas dépasser et enfin la fonction de rappel qui est automatiquement appelée lors d'un changement sur l'action de cette barre de réglage, ici réglée à **None**.

Une fois que nous gardons en mémoire la saturation originale et que nous avons mis en place cette barre de réglage, nous effectuons une boucle continue jusqu'à ce que l'utilisateur soit satisfait, c'est-à-dire lorsque qu'il appuie sur la touche « **Entrée** ».

La grande nouveauté dans cette méthode se situe au niveau de la variable **calcul** qui est de type **MatExpr**. Cette variable est issue d'un calcul effectué entre une matrice et de simples valeurs numériques. Ainsi, **chaque pixel** de la matrice peut effectuer un calcul avec une multiplication et une division pour arriver au pourcentage souhaité. L'expression est ainsi très concise et intuitive.

Si vous souhaitez récupérer la matrice issue de ce calcul, il suffit de faire appel à la méthode **to\_mat()**. À l'issu de ces traitements, nous recomposons notre modèle **HSV** par une fusion des différents canaux avec la nouvelle saturation à prendre en compte, que nous transformons ensuite en format **BGR** pour l'affichage du résultat.

## Tout est dans la matrice (code en C++)

Fondamentalement, une photo numérique est une matrice de pixels composant l'image. Profitons de ce chapitre pour analyser cette structure représentée par la structure **Mat**. Nous recenserons les fonctionnalités disponibles sur les traitements internes sur chacun de ces pixels de la structure globale de la matrice en passant par des fonctions spécifiques ou au moyen d'opérateurs redéfinis spécifiquement pour cette structure.

**La structure Mat est essentiellement composée de deux parties ; une en-tête et le bloc des pixels lui-même. L'en-tête contient toutes les informations relatives à la matrice des pixels (la taille, le nombre de canaux - couleurs, le modèle de couleur, etc.).**

Un élément vraiment important à souligner, c'est que l'en-tête comporte également **une variable interne qui est un pointeur** sur le bloc des données constitué par l'ensemble des pixels de l'image. Ce **pointeur** est très important, puisque lorsque nous faisons une copie d'une matrice vers une autre (objets de la structure **Mat**), nous copions uniquement l'en-tête de la matrice. Le bloc de données n'existe qu'en un seul exemplaire. Cela permet d'avoir une grande performance en terme de rapidité d'exécution. Si vous désirez vraiment une copie des données dans un autre bloc, il existe des méthodes spécifiques pour cela, **copyTo()** ou **clone()**.

Je vous propose de recenser quelques traitements spécifiques sur le bloc de données et sur l'en-tête en utilisant des méthodes ou des fonctions en relation avec cette structure **Mat**.

### Création d'une image

Il est possible de créer vos propres images durant la phase de génération de vos objets de type **Mat**. Il suffit de prendre le bon constructeur (il existe pas moins de 17 constructeurs). Durant cette phase de construction, vous devez préciser au minimum les dimensions de l'image, le nombre de canaux, et éventuellement une valeur de couleur par défaut.

Bien que la classe **Mat** propose beaucoup plus de solutions, généralement nous générerons de types d'image, soit en noir et blanc (niveau de gris) soit en couleur. Les deux constantes associées s'appellent respectivement **CV\_8U** et **CV\_8UC3** (8 pour octet, **U** pour non signé et **C** pour canaux). Il existe d'autres modèles de structure pour le pixel en prenant par exemple 16 bits par couleur pour l'imagerie médicale, des octets signés, etc.

```
$ Mat noire(240, 320, CV_U8); // image en noir et blanc (un seul canal) de 320x240 totalement noire au départ
$ Mat gris(240, 320, CV_U8, 100); // image gris foncé de 320x240
$ Mat couleur(240, 320, CV_U8C3); // image en couleur (trois canaux) de 320x240 totalement noire au départ
$ Mat rouge(240, 320, CV_U8C3, Scalar(0, 0, 255)); // image en couleur (trois canaux) de 320x240 totalement rouge au départ
```

### Retailler une image, insérer une image ou récupérer une partie de l'image

Lorsque vous récupérez des photos numériques, souvent les dimensions sont largement au-delà de la résolution de l'écran, il peut être judicieux de redimensionner l'image afin de réduire largement les temps de traitement (en divisant la largeur par deux par exemple, vous obtenez une surface divisée par quatre). La fonction **resize()** permet de réaliser cette opération en spécifiant soit la taille absolue désirée, soit la proportion de redimensionnement (coefficients multiplicateurs). Pour ce traitement, nous devons faire une interpolation. Par défaut, c'est une interpolation linéaire qui est proposée dont le traitement est très rapide à réaliser. Pour plus de finesse dans le résultat, au détriment du temps de réponse, vous pouvez choisir l'interpolation cubique.

```
$ Mat couleur = imread("Papillon.jpg"); // récupération de la photo
$ resize(couleur, couleur, Size(320, 240)); // redimensionnement de l'image en spécifiant directement les dimensions voulues
$ resize(couleur, couleur, Size(), 0.33, 0.33, INTER_CUBIC); // redimensionnement de l'image en divisant la largeur et la hauteur par 3
```

### Récupérer une partie de l'image et insérer une autre image

Nous pouvons insérer une image dans une partie d'une autre image. Pour réaliser cette manœuvre, vous devez d'abord récupérer la partie de l'image qui vous intéresse en générant une nouvelle matrice, grâce à la redéfinition de l'opérateur parenthèses en spécifiant les portions en **largeur** et en **hauteur** grâce à la classe **Range**. Attention, encore une fois, la portion d'image récupérée n'est pas indépendante de l'image originale. C'est d'ailleurs grâce à cela que nous pouvons modifier une partie de l'image originale.

Pour l'insertion, vous utilisez la méthode **copyTo()** associée à l'image que vous désirez intégrer et en argument la portion de l'image originale qui sert de cible. Il est possible de rajouter un deuxième argument à cette méthode qui sert alors de masque (qui doit être conçu en noir et blanc).

```
$ Mat photo = imread("Papillon.jpg"); // récupération de la photo
$ Mat rouge(120, 80, CV_U8C3, Scalar(0, 0, 255)); // création d'une image rouge de 120x80
$ Mat portion = photo(Range(10, 90), Range(10, 130)); // récupération d'une partie de la photo à 10 pixels du bord haut gauche
$ rouge.copyTo(portion); // L'image originale possède maintenant un plastron rouge de 120x80
```

### Fonctions de manipulation de pixels sur l'ensemble de l'image

Nous avons souvent besoin de fusionner deux images entre-elles, pour réaliser du photo montage. Lors de cette fusion, nous pouvons agir de façon arithmétique (addition, soustraction, multiplication, etc.) ou de façon logique (et, ou, ou exclusif, etc.). Nous pouvons également fusionner une partie de l'image en utilisant cette fois-ci un masque supplémentaire. Dans tous les cas, il est nécessaire que les deux images soient de même dimension et surtout de même type (sauf pour le masque bien sûr). Enfin, il est possible de proposer une proportion de chacune des images à fusionner (30% pour la première et 70% pour la seconde).

```
$ add(img1, img2, result); // Fusion de img1 avec img2 pour former result.
```

```
$ add(img1, Scalar(10), result); // Augmenter la luminosité de tous les pixels d'une même valeur
$ addWeighted(img1, 0.3, img2, 0.7, result); // Fusion avec 30% de img1 et 70% de img2 pour former result.
$ scaleAdd(img1, 0.3, img2, result); // Augmentation ou diminution de la luminosité sur la première image.
$ add(img1, img2, result, masque); // Fusion d'une partie de l'image résolue par le masque.
$ subtract(img1, img2, result); // Soustrait img2 de img1 pour former result.
$ absdiff(img1, img2, result); // Conserve les pixels qui sont différents entre img1 et img2 pour former result.
$ multiply(img1, img2, result); // Multiplie tous les pixels de img1 avec img2 pour former result.
$ divide(img1, img2, result); // Divise tous les pixels de img1 avec img2 pour former result.
$ bitwise_and(img1, img2, result); // Et logique sur l'ensemble des pixels de img1 avec img2 pour former result.
$ bitwise_or(img1, img2, result); // Ou logique sur l'ensemble des pixels de img1 avec img2 pour former result.
$ bitwise_xor(img1, img2, result); // Ou exclusif sur l'ensemble des pixels de img1 avec img2 pour former result.
$ bitwise_not(img1, img2, result); // Négation logique sur l'ensemble des pixels de img1 avec img2 pour former result.
```

La fonction `absdiff()` et très similaire à `subtract()` avec un avantage supplémentaire c'est que l'ordre des images n'a pas d'importance (notion de valeur absolue) alors que pour `subtract()`, il faut bien choisir dans quel ordre nous soumettons nos images pour avoir le résultat escompté. Ces fonctions peuvent être très utiles pour détecter un mouvement dans une vidéo, par exemple.

Comme pour la fonction `add()`, nous pouvons appliquer un masque avec les fonctions `subtract()`, `multiply()`, `divide()`, etc.

#### Opérateurs de manipulation de pixels sur l'ensemble de l'image

Toutes les fonctions que nous venons de découvrir peuvent être réalisées avec les opérateurs classiques d'arithmétique et de traitement binaire. Ils ont été spécialement redéfinis pour faire ces même traitements avec une syntaxe allégée.

```
$ result = img1 * 0,3 + img2 * 0,7; // Fusion avec 30% de img1 et 70% de img2 pour former result.
```

```
$ result = img1 * 1.3; // Augmentation ou diminution de la luminosité sur la première image.
```

## MANIPULATION DES PIXELS AVEC UNE COULEUR DE BASE

Dans le projet précédent, nous avons réalisé des traitements qui effectuaient des calculs matriciels avec des opérations de multiplication et de division. Nous continuons cette démarche en appliquant cette fois-ci d'autres opérations de base comme l'addition et la soustraction avec également des opérations binaires comme le **et** logique, le **ou** logique la **négation**, etc.

Pour ce type d'opérations, nous avons besoin de passer par la structure **Scalar** qui représente en réalité un vecteur de quatre **float** qui permet de spécifier pour chaque pixel la couleur souhaitée en adéquation du traitement à réaliser.

### main.rs

```
use std::ops::Add;
use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*};

fn main() -> Result<()> {
    let gris = imread("/home/manu/Images/Papillon.jpg",
IMREAD_REDUCED_GRAYSCALE_2)?;
    imshow("Fenêtre", &gris)?;
    wait_key(0)?;

    let negatif = (Scalar::all(255.) - &gris).into_result()?.to_mat()?;
    imshow("Fenêtre", &negatif)?;
    wait_key(0)?;

    let mut filtre = Mat::default();
    let masque = Mat::default();
    bitwise_not(&negatif, &mut filtre, &masque)?;
    imshow("Fenêtre", &filtre)?;
    wait_key(0)?;

    let mut gris_3canaux = Mat::default();
    cvt_color(&gris, &mut gris_3canaux, COLOR_GRAY2BGR, 0)?;
    let sepia = (&gris_3canaux / 2. + Scalar::new(120., 140., 169., 0.)).into_result()?;
    imshow("Fenêtre", &sepia.to_mat())?;
    wait_key(0)?;

    let rouge = Scalar::new(0., 0., 255., 0.);
    bitwise_and(&gris_3canaux, &rouge, &mut filtre, &masque)?;
    imshow("Fenêtre", &filtre)?;
    wait_key(0)?;

    bitwise_or(&gris_3canaux, &rouge, &mut filtre, &masque)?;
    imshow("Fenêtre", &filtre)?;
    wait_key(0)?;

    let gris_rouge = gris_3canaux.add(&rouge).into_result()?;
    imshow("Fenêtre", &gris_rouge.to_mat())?;
    wait_key(0)?;

    Ok(())
}
```



Cette structure **Scalar** possède deux méthodes statiques ***new()*** et ***all()*** qui permettent respectivement de spécifier les valeurs choisies pour chacun des canaux ou proposer la même valeur pour l'ensemble des canaux.

Cette structure **Scalar** ainsi que la structure **Mat**, contrairement au C++, manipulent des valeurs réelles. Rust ne permet pas de réaliser des calculs à la fois sur des entiers et des réels. Comme certains traitements proposent des proportions de valeurs sur chaque pixel, il a été choisi de prendre systématiquement des valeurs réelles.

Le projet ci-dessus recense les possibilités offertes afin de réaliser des traitements au niveau de chaque bit de l'image en utilisant des expressions de type **MatExpr**, des fonctions de traitement binaire de style ***bitwise\_xx()*** ou l'appel de méthodes sur une matrice particulière comme ***add()***, ***sub()***, ***mul()***, ***div()***, etc.

Le premier traitement réalisé permet de montrer la photo **négative** à partir d'une image en niveau de gris. L'idée de base est de retrancher chaque pixel de a valeur **255**. La deuxième solution pour obtenir le même résultat consiste à utiliser la fonction ***bitwise\_not()***.

Ensuite, nous désirons proposer une photo couleur sépia. Puisque nous manipulons une couleur, nous devons d'abord faire en sorte d'avoir les trois canaux de couleurs à partir d'une image en niveau de gris. D'où la conversion réalisée.

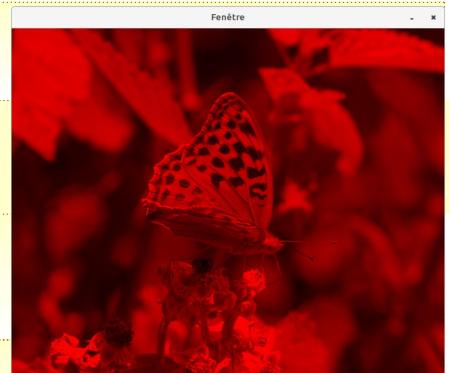
Nous créons ensuite la couleur sépia avec les valeurs nécessaires sur chacun des canaux **BGR** au moyen de cette structure adaptée **Scalar**. Le dernier canal n'est pas utilisé, puisque la transparence n'existe pas sur un format **JPEG**.

Toujours avec **Scalar**, nous réalisons un filtre rouge sur la photo originale au moyen de la fonction ***bitwise\_and()***.

Plutôt que de réaliser un filtre, nous pouvons rajouter une couleur, cette fois-ci avec la fonction ***bitwise\_or()*** ou tout simplement en additionnant la couleur avec la valeur en niveau de gris à l'aide de la méthode ***add()*** associée à la matrice grise en trois couches. Nous obtenons exactement le même résultat.

**Retenez bien que lorsque vous souhaitez rajouter une couleur sur une image noir et blanc, il faut nécessairement posséder les trois couches **BGR**, et donc faire une transformation en conséquence.**

Les fonctions ***bitwise\_xx()*** nécessitent un argument supplémentaire qui permet de proposer un masque si vous souhaitez réaliser le traitement que sur une partie de l'image. Pour que le traitement se fasse sur la totalité de l'image, vous proposez simplement une matrice vierge par défaut.



## ÉRODER OU DILATER UNE PHOTO

Pour des images bien particulières, notamment pour des caméras placées dans une chaîne de production, vous pouvez souhaiter éliminer les bruitages ou des bavures de taille relativement réduite grâce à la fonction ***erode()*** qui propose un algorithme où les parties claires de très petites tailles sont éliminées automatiquement, ou à l'inverse, la fonction ***dilate()*** qui élimine plutôt les parties sombres également de très petites tailles.

### Éroder ou dilater une image (version C++)

Les fonctions ***erode()*** et ***dilate()*** permettent de supprimer les pixels isolés qui correspondent plus à du « bruit » respectivement soit trop lumineux, soit trop sombre. Vous avez ci-dessous les deux écritures par défaut couramment utilisées

```
$ erode(original, resultat, Mat(); // Supprimer les pixels lumineux de toutes petites tailles 3x3
```

```
$ dilate(original, resultat, Mat(); // Supprimer les pixels les plus sombres de toutes petites tailles 3x3
```

Par défaut, lorsque nous utilisons un objet de type **Mat::default()** sans arguments, la suppression se fait pour des zones de pixel **3x3**. Il est possible bien sûr, de proposer une érosion plus poussée, en changeant la dimension de la zone de traitement.

```
$ erode(original, resultat, Mat(7, 7, CV_8U, Scalar(1))); // Supprimer les pixels lumineux de toutes petites tailles 7x7
```

```
$ dilate(original, resultat, Mat(7, 7, CV_8U, Scalar(1))); // Supprimer les pixels les plus sombres de toutes petites tailles 7x7
```

Une autre solution pour éroder de façon plus poussée, est de faire en sorte que l'algorithme de base soit automatiquement réalisé plusieurs fois :

```
$ erode(original, resultat, Mat(), Point(-1, -1), 3); // Éroder l'image trois fois
```

```
$ dilate(original, resultat, Mat(), Point(-1, -1), 3); // Dilater l'image trois fois
```

### main.rs

```
use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*};
use std::env;
fn main() -> Result<()> {
    let original = imread("/home/manu/Images/chien.jpg", IMREAD_COLOR)?;
    imshow("Résultat", &original)?;
    wait_key(0)?;

    let mut eroder = Mat::default();
    let mut dilater = Mat::default();
    let defaut = Mat::default();
    let point = Point::new(-1, -1);
    let scalaire = Scalar::all(1.0);
    erode(&original, &eroder, &defaut, &point, 3);
    dilate(&eroder, &resultat, &defaut, &point, 3);
    imshow("Résultat", &resultat)?;
    wait_key(0);
}
```

```

erode(&original, &mut eroder, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
imshow("Résultat", &eroder)?;
wait_key(0)?;

dilate(&original, &mut dilater, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
imshow("Résultat", &dilater)?;
wait_key(0)?;

let mut pixels = 1;
create_trackbar("Pixels", "Résultat", Some(&mut pixels), 11, None)?;

loop {
    let matrice = Mat::new_rows_cols_with_default(pixels, pixels, CV_8U, scalaire)?;
    erode(&original, &mut eroder, &matrice, point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Résultat", &eroder)?;
    if wait_key(30)? == 13 { break; }
}

let matrice = Mat::new_rows_cols_with_default(4, 4, CV_8U, scalaire)?;
erode(&original, &mut eroder, &matrice, point, 1, BORDER_DEFAULT, scalaire)?;
dilate(&eroder, &mut dilater, &matrice, point, 1, BORDER_DEFAULT, scalaire)?;
imshow("Résultat", &dilater)?;
wait_key(0)?;

Ok(())
}

```

Dans la version du langage **Rust**, les fonctions **erode()** et **dilate()** proposent plus d'arguments, puisque dans ce langage, je le rappelle, les paramètres par défaut n'existent pas.

Au delà des matrices source et résultat, nous avons besoin d'une matrice supplémentaire qui permet de choisir la dimension des bruits à éliminer. Je rappelle que la matrice par défaut est de dimension **3x3**.

Toujours dans ces fonctions, vous devez préciser le point de décalage. En prenant un point aux coordonnées **(-1,-1)**, nous spécifions que nous n'envisageons aucun décalage.

Le dernier argument permet d'établir la couleur de base à choisir au moyen de la structure **Scalar**. Il suffit de proposer la valeur unitaire sur chacun des canaux.

Lorsque nous utilisons la fonction **erode()**, les valeurs sombres grossissent légèrement suivant le choix de la matrice par défaut. De la même façon, la fonction **dilate()**, les valeurs claires se développent en relation avec cette même matrice.

Si vous souhaitez conserver les dimensions des éléments principaux, il est souvent nécessaire d'utiliser la fonction **dilate()** après avoir utilisé la fonction **erode()** et inversement. L'enchaînement de ces opérations permet de retrouver les dimensions originales.

Si vous devez changer la dimension de la matrice par défaut, nous devez utiliser la méthode statique **new\_rows\_cols\_width\_default()** avec les premiers arguments pour la dimension, le type de couleur pour chaque canal, ici **CV\_8U** et encore une fois la couleur par défaut toujours au travers de la structure **Scalar**.



## TRAITEMENTS MORPHOLOGIQUES

Dans le domaine de la détection, nous avons souvent besoin de repérer des formes géométriques particulières, des arêtes, des coins, des rectangles, des cercles, etc. La fonction **morphologyEx()** est particulièrement adaptée pour ce genre de recherche.

### Éroder et dilater en une seule opération

La fonction **morphologyEx()** possède des paramètres prédéfinis pour réaliser des opérations sophistiquées et adaptées au différentes situations. Nous avons souvent besoin d'éroder un masque suivi ensuite de la fonction inverse pour dilater avec les mêmes réglages de traitement, comme nous l'avons expérimenté dans le chapitre précédent. Dans ce cas de figure, la fonction **morphologyEx()** permet de faire la même opération en une seule passe en prenant le paramètre de réglage **MORPH\_OPEN**. Le paramètre **MORPH\_CLOSE** réalise l'opération inverse, la dilatation suivie de l'érosion.

```
$ morphologyEx(masque, resultat, MORPH_OPEN, Mat(7, 7, CV_8U, Scalar(1)));
$ morphologyEx(masque, resultat, MORPH_CLOSE, Mat(7, 7, CV_8U, Scalar(1)));
```

### Tous les réglages morphologiques

À partir de l'original ci-contre, voici tous les réglages morphologiques possibles :





MORPH\_BLACKHAT



MORPH\_CLOSE



MORPH\_CROSS



MORPH\_TOPHAT



MORPH\_DILATE



MORPH\_ELLIPSE



MORPH\_ERODE



MORPH\_GRADIENT



MORPH\_OPEN



MORPH\_RECT

**main.rs**

```
use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*, Result };

fn main() -> Result<()> {
    let original = imread("/home/manu/Images/chien.jpg", IMREAD_COLOR)?;
    imshow("Résultat", &original)?;
    wait_key(0)?;

    let mut eroder = Mat::default();
    let mut dilater = Mat::default();
    let mut morphologie = Mat::default();
    let defaut = Mat::default();
    let point = Point::new(-1, -1);
    let scalaire = Scalar::all(1.);

    erode(&original, &mut eroder, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
    dilate(&eroder, &mut dilater, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Résultat", &dilater)?;
    wait_key(0)?;

    morphology_ex(&original, &mut morphologie, MORPH_OPEN, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Résultat", &morphologie)?;
    wait_key(0)?;

    morphology_ex(&original, &mut morphologie, MORPH_ERODE, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Résultat", &morphologie)?;
    wait_key(0)?;

    morphology_ex(&original, &mut morphologie, MORPH_CLOSE, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Résultat", &morphologie)?;
    wait_key(0)?;

    morphology_ex(&original, &mut morphologie, MORPH_DILATE, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Résultat", &morphologie)?;
    wait_key(0)?;

    morphology_ex(&dilater, &mut morphologie, MORPH_GRADIENT, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Résultat", &morphologie)?;
    wait_key(0)?;
```



Dans la première partie du code, nous effectuons une érosion suivie d'une dilatation afin de supprimer les pixels les plus petits avec une matrice par défaut, comme nous l'avons déjà réalisé dans le projet précédent. Nous effectuons le même traitement avec un résultat totalement identique en exécutant une seule fonction **morphology\_ex()** en choisissant la constante **MORPH\_OPEN**.

Lorsque nous choisissons la constante **MORPH\_ERODE**, le résultat est similaire à la simple fonction **erode()**, de même, lorsque nous utilisons plutôt la constante **MORPH\_DILATE** cela revient également à utiliser directement la fonction **dilate()**.

La constante **MORPH\_CLOSE** permet d'avoir l'équivalent de l'utilisation successive des fonctions **dilate()** et **erode()** pour retrouver les formes originales de plus grosses parties. Enfin, le choix de la constante **MORPH\_GRADIENT** permet de souligner les contours des éléments principaux.

## SÉLECTIONNER UNE PARTIE DE L'IMAGE À L'AIDE D'UN MASQUE

Dans beaucoup de situations, nous avons besoin de sélectionner qu'une partie de l'image. Une des alternatives consiste à réaliser des masques par rapport à des seuils d'intensités lumineuses, au moyen de la fonction **threshold()**. Il existe une autre fonction **adaptiveThreshold()** dont l'objectif est différent puisqu'il s'intéresse plus au contours d'éléments contrastés. Nous réemployerons également les fonctions **erode()** et **dilate()** ou **morphology\_ex()** afin de supprimer les pixels intempestifs.

### Masque avec seuils

La fonction **threshold()** est spécialisée à la création de masque à partir de deux seuils d'intensité lumineuse, fourchette entre deux limites d'intensités. Attention, du coup, ce type de traitement impose de travailler avec des niveaux de gris et non avec des images en couleur.

Comme d'autres fonctions, plusieurs réglages sont possibles, comme nous le voyons ci-contre. Par exemple, dans le mode **BINARY**, lorsque le pixel possède une intensité en dehors des limites choisies, il devient totalement noir et à l'inverse, lorsque l'intensité correspond à la fourchette prévue, le pixel devient complètement blanc. Le mode **BINARY\_INV** propose de réaliser le masque inverse, etc.

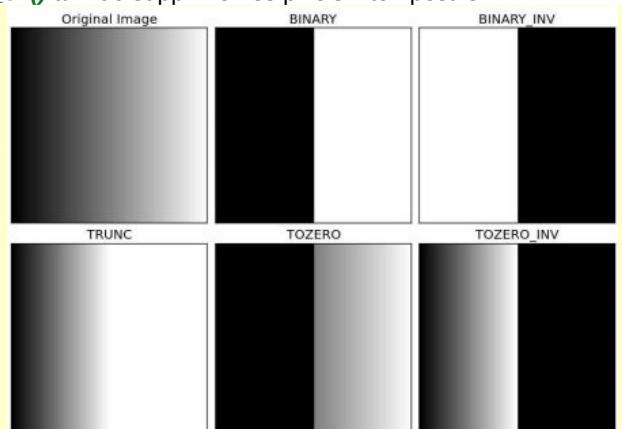
```
$ threshold(original, masque, seuilBas, seuilHaut, THRESH_BINARY);
```

// Génération d'un masque classique

La fonction **adaptiveThreshold()**, plus performante que la précédente, spécifie que nous devons maîtriser des seuils de luminosité avec en plus une certaine adaptation suivant le contexte. Attention, là aussi, ce type de traitement impose de travailler avec des niveaux de gris. Le résultat du traitement sera en noir et blanc, correspondant bien à la notion de masque, comme la fonction précédente d'ailleurs. Cette fonction possède pas mal de paramètres qui permettent ainsi d'ajuster le résultat afin de bien discriminer les bords des éléments principaux de l'image.

```
$ adaptiveThreshold(original, masque, 255, ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY, 9, 2);
```

// Génération d'un masque avec adaptation



Le premier paramètre de réglage correspond à la couleur de seuil souhaitée : 255 → blanc, 127 → gris.

Le deuxième paramètre de réglage spécifie le type de seuil que nous désirons prendre en compte :

Le troisième paramètre de réglage spécifie la finesse du traitement comme le montre les vues ci-contre :

Comme beaucoup, d'autres traitements, le paramètre qui suit correspond à la largeur du bloc de pixels à prendre en compte pour chaque pixel de l'image.

Enfin, le dernier paramètre permet de choisir la largeur en pixels des bordures discriminées à prendre en compte.

### main.rs

```
use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*, Result};

fn main() -> Result<()> {
    let stars = imread("/home/manu/Images/stars.jpg",
IMREAD_GRAYSCALE)?;
    imshow("Résultat", &stars)?;
    wait_key(0)?;

    let mut seuil = 130;
    let mut masque = Mat::default();
    named_window("Masque", WINDOW_AUTOSIZE)?;
    create_trackbar("Seuil", "Masque", Some(&mut seuil), 255,
None)?;
    loop {
        threshold(&stars, &mut masque, seuil as f64, 255., THRESH_BINARY)?;
        imshow("Masque", &masque)?;
        if wait_key(30)? == 13 { break; }

    }

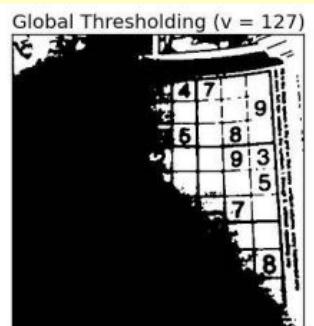
    let mut pixels = 5;
    let mut eroder = Mat::default();
    let point = Point::new(-1, -1);
    let scalaire = Scalar::all(1.);
    named_window("Eroder", WINDOW_AUTOSIZE)?;
    create_trackbar("Pixels", "Eroder", Some(&mut pixels), 15, None)?;
    loop {
        let defaut = Mat::new_rows_cols_with_default(pixels, pixels, CV_8U, scalaire)?;
        morphology_ex(&masque, &mut eroder, MORPH_OPEN, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
        imshow("Eroder", &eroder)?;
        if wait_key(30)? == 13 { break; }
    }

    let mut contour = Mat::default();
    adaptive_threshold(&eroder, &mut contour, 255., ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY, 5, 2.)?;
    imshow("Contour", &contour)?;
    wait_key(0)?;

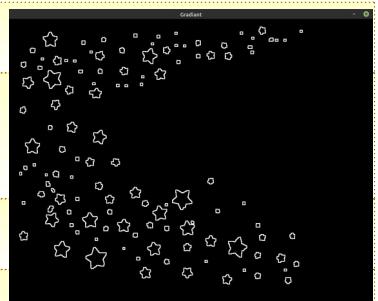
    morphology_ex(&eroder, &mut contour, MORPH_GRADIENT, &Mat::default(), point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Contour", &contour)?;
    wait_key(0)?;

    let mut inversion = Mat::default();
    bitwise_not(&contour, &mut inversion, &Mat::default())?;
    imshow("Contour", &inversion)?;
    wait_key(0)?;

    Ok(())
}
```



Le projet ci-dessus permet de discriminer le contour des étoiles principales sous la forme d'un masque avec donc deux valeurs possibles de couleur, noir et blanc.



Le premier traitement de ce projet consiste à faire une première ébauche du masque en éliminant les étoiles de teinte grise pour ne conserver que les plus brillantes au travers d'une barre de réglage qui permet de choisir le seuil de détection qui donne la limite entre les parties noires et les étoiles blanches.

Cela se fait au travers de la fonction **threshold()** qui prend une image en niveau de gris pour générer une nouvelle image en noir et blanc suivant le seuil de détection.

Afin d'éliminer dans le masque, les toutes petites étoiles et les traînées restantes, nous érodons l'image au travers de la fonction **morphology\_ex()**. Là aussi le réglage se fait au travers d'une nouvelle barre de réglage qui permet d'affiner notre résultat.

Il nous reste alors à créer le contour de chacune des étoiles restantes que nous pouvons réaliser au travers de la fonction **adaptive\_threshold()** sachant que le résultat donne un fond blanc et le tracé du contour en noir. Si vous souhaitez avoir plutôt un fond noir et le tracé en blanc, il suffit de choisir la constante **THRESH\_BINARY\_INV**.

Je rappelle que pour générer un contour, nous pouvons également utiliser de nouveau la fonction **morphology\_ex()** mais en prenant cette fois-ci la constante **MORPH\_GRADIENT** sachant que le fond sera alors noir et le tracé du contour en blanc. Si vous désirez retrouver le même contour que précédemment, il suffit d'inverser le masque au moyen de la fonction **bitwise\_not()**.

## DÉTECTOR LES CONTOURS

I existe plusieurs algorithmes pour détecter les contours d'une image pour discerner les éléments importants. L'algorithme **Canny** est très performant dans ce domaine et surtout très simple à utiliser puisqu'il existe une fonction spécifique pour cela. Dans le domaine de l'analyse des contours, nous pouvons également détecter des formes géométriques comme les cercles, grâce à la fonction **HoughCircles()**, des lignes, avec la fonction **HoughLinesP()**, et pour des formes quelconques la fonction **findContours()**.

### Discerner les contours

La fonction **canny()** est vraiment spécialisée dans la détection des contours et délivre une image directement en noir et blanc en utilisant un filtre **GRADIENT** du même type que la fonction **morphology\_ex()**. Vous devez juste proposer les seuils limites bas et haut de luminosité, à l'image d'un hystérisis, qui vont permettre de bien régler finement les contrastes à prendre en compte.

`$ canny(image, contours, seuilbas, seuilhaut);`

main.rs

```
use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*, Result };

fn main() -> Result<()> {
    let papillon = imread("/home/manu/Images/Papillon.jpg",
IMREAD_REDUCED_COLOR_2)?;
    imshow("Original", &papillon)?;
    wait_key(0)?;

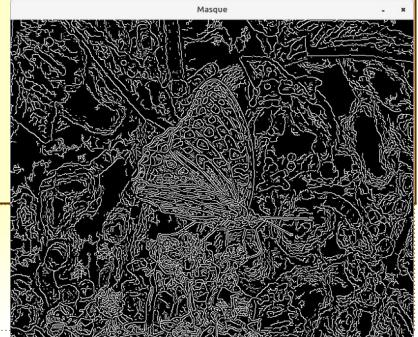
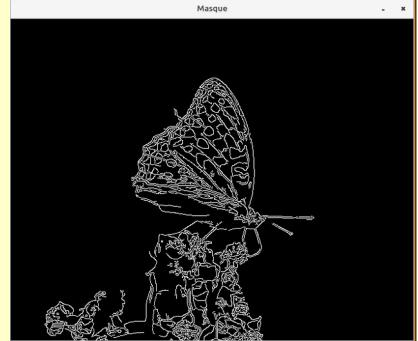
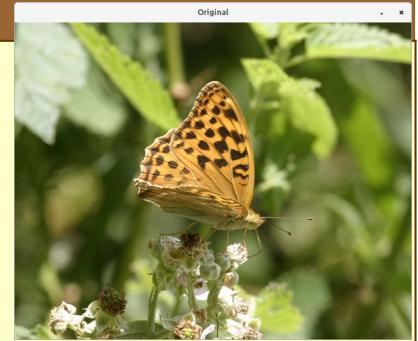
    let mut contour = Mat::default();
    canny(&papillon, &mut contour, 80., 300., 3, true)?;
    imshow("Masque", &contour)?;
    wait_key(0)?;

    canny(&papillon, &mut contour, 80., 300., 5, true)?;
    imshow("Masque", &contour)?;
    wait_key(0)?;

    let mut seuil_bas = 80;
    let mut seuil_haut = 300;
    create_trackbar("Seuil bas", "Masque", Some(&mut seuil_bas), 700, None)?;
    create_trackbar("Seuil haut", "Masque", Some(&mut seuil_haut), 700, None)?;
    loop {
        canny(&papillon, &mut contour, seuil_bas as f64, seuil_haut as f64, 3, true)?;
        imshow("Masque", &contour)?;
        if wait_key(30)? == 13 { break; }
    }

    let mut inversion = Mat::default();
    bitwise_not(&contour, &mut inversion, &Mat::default())?;
    imshow("Contour", &inversion)?;
    wait_key(0)?;

    Ok(())
}
```



La fonction **canny()**, comme pour toutes les autres fonctions de traitement d'image prend en arguments l'image initiale suivie de l'image traitée. Ensuite, comme expliqué plus haut, nous donnons les seuils bas et haut du filtre au format **f64**, sachant que la limite n'est pas fixée à **255** comme bien d'autres filtres.

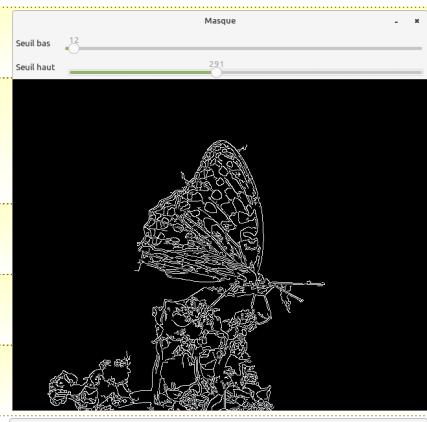
Les deux autres arguments permettent de régler l'ouverture et l'activation du mode **MORPH\_GRADIENT**.

Pour l'ouverture seules trois valeurs sont permises 3, 5 et 7. Dans la page précédente, vous pouvez bien voir le résultat entre une ouverture réglée à 3 où nous pouvons parfaitement discerner le contour du papillon, alors que la valeur 5 dessine pratiquement tous les contours riches de l'image.

Le gros avantage de cette fonction **canny()**, c'est qu'elle fabrique directement le masque et discrimine bien les contours importants de l'image, ceci avec finalement peu de réglages.

Afin que la discrimination corresponde bien à votre désir, il est préférable de prévoir les barres de réglage comme nous l'avons fait sur chacun des seuils.

Enfin, nous pouvons une fois de plus inverser le masque avec la fonction **bitwise\_not()** comme lors du projet précédent.



## DÉTECTOR DES FORMES GÉOMÉTRIQUES

Dans la lignée de recherche de contours, je vous propose maintenant de façon plus générale de retrouver des formes géométriques. Cette recherche particulière consiste à retrouver des polygones fermés dans l'ensemble des points délivrés dans la discrimination des contours, sachant que les formes circulaires peuvent être considérés comme des polygones avec un nombre d'arêtes conséquent.

### Retrouver des polygones et calculer leurs barycentres

La fonction **approx\_poly\_dp()** discrimine juste le nombre de points nécessaires à la construction de polygone, dans l'ensemble des points d'un contour retrouvé par la fonction **find\_contours()**. Cette fonction **approx\_poly\_dp()** prend en paramètres, la liste des points d'un contour et génère une nouvelle liste de points pour le tracé des polygones, le différentiel minimal pour chaque arête afin d'éviter d'avoir trop de points pour tracer les polygones (ou les ellipses), et enfin la validation d'une courbe fermée

La fonction **moments()** calcule automatiquement le barycentre d'un ensemble de points délivrés par un contour. Cette fonction génère un objet de type **Moments** qui dispose alors d'attributs pour calculer correctement le barycentre de la forme, les coordonnées étant  $x=m10/m00$  et  $y=m01/m00$ .

### main.rs

```
use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*, Result };
use opencv::types::{VectorOfMat};

fn main() -> Result<()> {
    let couleur = imread("/home/manu/Images/figures.png", IMREAD_COLOR)?;
    imshow("Original", &couleur)?;
    wait_key(0)?;

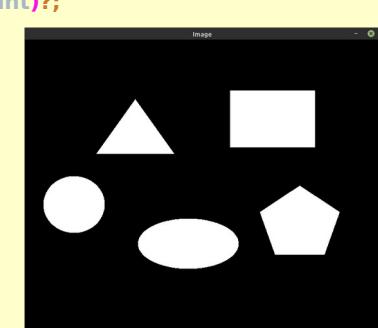
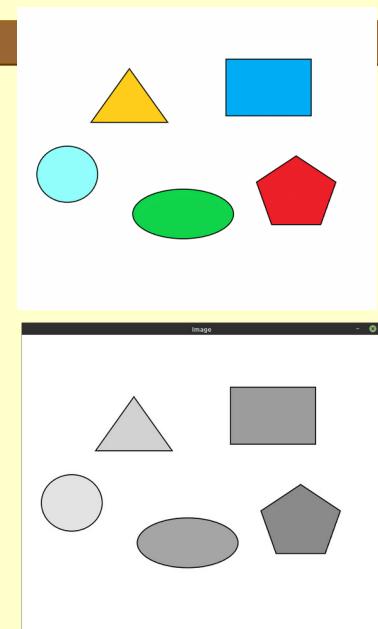
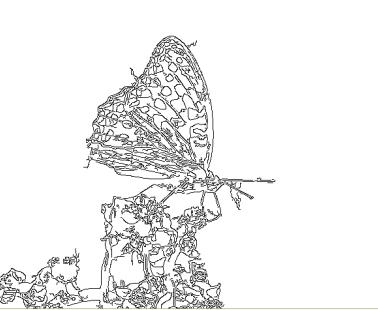
    let mut gris = Mat::default();
    cvt_color(&couleur, &mut gris, COLOR_BGR2GRAY, 0)?;
    imshow("Gris", &gris)?;
    wait_key(0)?;

    let mut masque = Mat::default();
    threshold(&gris, &mut masque, 220., 255., THRESH_BINARY_INV)?;
    imshow("Masque", &masque)?;
    wait_key(0)?;

    let point = Point::new(0, 0);
    let violet = Scalar::new(255., 0., 255., 0.);
    let bleu = Scalar::new(255., 0., 0., 0.);
    let blanc = Scalar::new(255., 255., 255., 0.);

    let mut rect = Rect::default();
    let mut contours = VectorOfMat::default();
    let mut resultat = Mat::new_rows_cols_with_default(600, 800, CV_8UC3, blanc)?;

    findContours(&masque, &mut contours, RETR_EXTERNAL, CHAIN_APPROX_NONE, point)?;
    for contour in contours {
        let mut polygone = Mat::default();
        approx_poly_dp(&contour, &mut polygone, 5., true)?;
        let octets = polygone.data_bytes()?;
        let nombre = polygone.size()?.height;
        println!("{} points : ", nombre);
        // let points = polygone.data_typed::<Point_i32>()?;
        let points = polygone.data_typed::<Point2i>()?;
        println!("{} ", match nombre {
            3 => "Triangle",
            4 => {
                rect = Rect::from_points(points[0], points[2]);
                "Rectangle"
            }
        });
    }
}
```



```

},  

5 => "Pentagone",  

8 => "Cercle",  

12 => "Ellipse",  

_ => "Forme inconnue"  

});  

println!("Octets = {:?}", &octets);  

println!("{:?}", points);  

polylines(&mut resultat, &contour, true, violet, 3, FILLED, 0)?;  
  

let barycentre = moments(&contour, true)?;  

let x = barycentre.m10/barycentre.m00;  

let y = barycentre.m01/barycentre.m00;  

let centre = Point::new(x as i32, y as i32);  

circle(&mut resultat, centre, 2, violet, 5, FILLED, 0)?;  

}  
  

println!("{:?}", rect);  

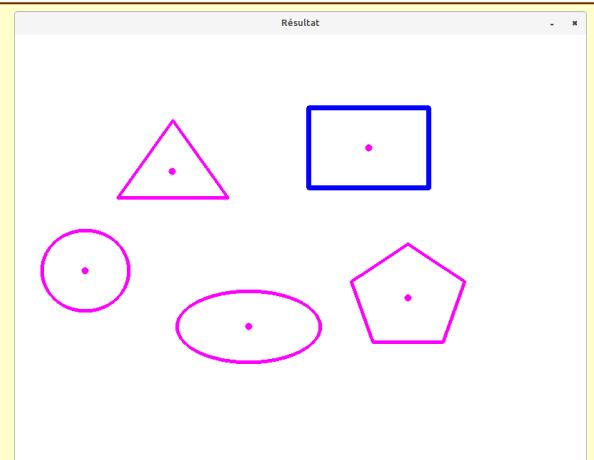
rectangle(&mut resultat, rect, bleu, 5, FILLED, 0)?;  

imshow("Résultat", &resultat)?;  

wait_key(0)?;  

Ok(())
}

```



## Résultat

12 points : Ellipse  
Octets = [227, 0, 0, 0, 147, 1, 0, 0, 232, 0, 0, 0, 169, 1, 0, 0, 247, 0, 0, 0, 183, 1, 0, 0, 54, 1, 0, 0, 202, 1, 0, 0, 133, 1, 0, 0, 192, 1, 0, 0, 161, 1, 0, 0, 175, 1, 0, 0, 171, 1, 0, 0, 158, 1, 0, 0, 166, 1, 0, 0, 136, 1, 0, 0, 151, 1, 0, 0, 122, 1, 0, 0, 88, 1, 0, 0, 103, 1, 0, 0, 9, 1, 0, 0, 113, 1, 0, 0, 237, 0, 0, 0, 130, 1, 0, 0]  
[Point { x: 227, y: 403 }, Point { x: 232, y: 425 }, Point { x: 247, y: 439 }, Point { x: 310, y: 458 }, Point { x: 389, y: 448 }, Point { x: 417, y: 431 }, Point { x: 427, y: 414 }, Point { x: 422, y: 392 }, Point { x: 407, y: 378 }, Point { x: 344, y: 359 }, Point { x: 265, y: 369 }, Point { x: 237, y: 386 }]  
5 points : Pentagone  
Octets = [215, 1, 0, 0, 89, 1, 0, 0, 245, 1, 0, 0, 174, 1, 0, 0, 87, 2, 0, 0, 174, 1, 0, 0, 117, 2, 0, 0, 91, 1, 0, 0, 39, 2, 0, 0, 37, 1, 0, 0]  
[Point { x: 471, y: 345 }, Point { x: 501, y: 430 }, Point { x: 599, y: 430 }, Point { x: 629, y: 347 }, Point { x: 551, y: 293 }]  
8 points : Cercle  
Octets = [89, 0, 0, 0, 18, 1, 0, 0, 48, 0, 0, 0, 42, 1, 0, 0, 38, 0, 0, 0, 82, 1, 0, 0, 60, 0, 0, 0, 118, 1, 0, 0, 108, 0, 0, 0, 130, 1, 0, 0, 149, 0, 0, 0, 106, 1, 0, 0, 159, 0, 0, 0, 66, 1, 0, 0, 137, 0, 0, 0, 30, 1, 0, 0]  
[Point { x: 89, y: 274 }, Point { x: 48, y: 298 }, Point { x: 38, y: 338 }, Point { x: 60, y: 374 }, Point { x: 108, y: 386 }, Point { x: 149, y: 362 }, Point { x: 159, y: 322 }, Point { x: 137, y: 286 }]  
3 points : Triangle  
Octets = [42, 1, 0, 0, 228, 0, 0, 0, 221, 0, 0, 0, 120, 0, 0, 0, 144, 0, 0, 0, 228, 0, 0, 0]  
[Point { x: 298, y: 228 }, Point { x: 221, y: 120 }, Point { x: 144, y: 228 }]  
4 points : Rectangle  
Octets = [155, 1, 0, 0, 102, 0, 0, 0, 155, 1, 0, 0, 215, 0, 0, 0, 68, 2, 0, 0, 215, 0, 0, 0, 68, 2, 0, 0, 102, 0, 0, 0]  
[Point { x: 411, y: 102 }, Point { x: 411, y: 215 }, Point { x: 580, y: 215 }, Point { x: 580, y: 102 }]  
Rect { x: 411, y: 102, width: 169, height: 113 }

La fonction `findContours()` est très souvent utilisée lorsque vous désirez retrouver des formes quelconques dans une zone particulière de l'image. Pour cela, vous devez systématiquement passer par la génération d'un masque qui permet de discriminer les zones utiles de l'ensemble de l'image.

Contrairement au C++, les contours retrouvés sont placés dans un vecteur de matrice. La composition de chaque matrice est alors adaptée à ce type de résultat. Les paramètres supplémentaires à fournir pour cette fonction permettent de spécifier l'algorithme de recherche par rapport à la délimitation du contour avec `RETR_EXTERNAL` et la discrimination de chaque point important avec `CHAIN_APPROX_NONE` (ici les formes sont parfaitement identifiées) ou `CHAIN_APPROX_SIMPLE` pour éliminer les points intermédiaires qui ne servent à rien.

Enfin, toujours dans cette fonction `findContours()`, nous devons spécifier si nous proposons un décalage en le désignant à l'aide d'un point. Vous verrez dans la suite des chapitres que nous avons très souvent besoin de cette fonction (elle est fondamentale pour la discrimination).

Pour chacun des contours, nous essayons de retrouver les formes géométriques grâce à la fonction `approxPolyDP()` qui donne le résultat, là aussi, dans une matrice qui enregistre les informations de façon spécifique afin qu'il soit facile par la suite de retrouver les points de chaque sommet. Une matrice est en réalité un moyen de stocker un ensemble de points, ce qui correspond bien à notre attente.

Puisque une matrice peut servir de stockage pour des images ou pour des points quelconques, il existe du coup beaucoup de méthodes qui permettent de résoudre les différents situations de recherche. Ainsi, à l'issue de la fonction `approxPolyDP()`, la méthode `size()` retourne la taille du résultat de la forme découverte avec une valeur `width` toujours égale à 1, et la valeur `height` qui indique le nombre de sommets trouvés.

Même si cela ne sert pas à grand-chose, vous pouvez retrouver les octets générés par la fonction `approxPolyDP()`. Plus utile, nous pouvons aussi récupérer les différents points constituant le polygone avec la méthode `dataTyped()` en spécifiant impérativement le type de point que vous souhaitez exploiter par la suite. Pour des points avec des coordonnées de type entier, vous pouvez choisir `Point2i` ou `Point<i32>`.

Pour le tracé de formes, vous pouvez passer par des fonctions de tracés spécifiques comme `circle()` ou `rectangle()` et pour les polygones plus généraux par la fonction de tracé `polylines()`.

Pour conclure, nous avons souvent besoin de travailler avec des zones rectangulaires. Pour cela, la structure `Rect` possède une méthode statique `fromPoints()` qui permet de retrouver les coordonnées du coin haut-gauche avec les dimensions de largeur et de hauteur à partir des deux points extrêmes trouvés par la fonction `approxPolyDP()`.

## GESTION DU FLOU DANS UNE IMAGE

Bien que cela paraisse bizarre de parler de flou pour une image alors que nous désirons la plupart du temps avoir une photo parfaitement nette, il existe des situations où le flou sera plutôt notre ami. C'est le cas notamment lorsque nous souhaitons sélectionner une partie de l'image sur un fond relativement uniforme. Dans ce cas de figure, nous faisons en sorte, grâce au flou, d'avoir un fond le plus homogène possible sans pixels intempestifs, en évitant les dégradés de couleur afin d'obtenir une teinte relativement homogène qu'il est facile ensuite de résoudre pour discriminer la partie importante de l'image.

### Trois fonctions pour gérer les flous :

La première fonction `blur()`, la plus rudimentaire, permet de réaliser un flou homogène sur l'ensemble de l'image. Elle prend en arguments deux objets de type `Mat`, l'origine de l'image et le résultat du traitement, suivi de la taille des pixels à prendre en compte pour réaliser le traitement pour chaque pixel examiné, par exemple `7x7`. Plus le pavé de pixels est grand, plus l'effet de flou est conséquent. Il faut par contre prendre systématiquement un nombre impair.

La fonction `gaussian_blur()` effectue un flou gaussien afin d'atténuer les micro imperfections dans un visage par exemple. Cette fonction, par rapport à la précédente, donne un résultat beaucoup plus esthétique puisqu'elle conserve globalement des contours plus nets. Le flou obtenu n'est donc pas homogène.

Elle aussi prend en arguments deux objets de type `Mat`, l'origine de l'image et le résultat du traitement, suivi de la taille des pixels à prendre en compte pour réaliser le traitement pour chaque pixel examiné, par exemple `7x7`, suivi enfin par le décalage éventuel de ces pavés de pixels par rapport au pixel d'origine. Plus le pavé de pixels est grand, plus l'effet de flou est conséquent. Même remarque pour le décalage. Il faut par contre prendre systématiquement un nombre impair. Un pavé de `3x3` peut largement suffire pour obtenir l'effet souhaité.

Enfin, la fonction `median_blur()` réalise un flou sur la photo, mais cette fois-ci de façon grossière. Elle est utilisée pour des traitements bien particulier, pour réduire globalement tous les petits détails d'une image pour ce concentrer sur les éléments fondamentaux qui ont une importance primordiale dans l'objectif final.

### Accentuation :

Ce sujet traite du flou, mais aussi de l'opération inverse qui consiste à rendre une image plus nette, ce que nous appelons l'accentuation. En réalité, pour avoir une image plus nette, nous partons du flou de l'image que nous retranchons à l'image originale qui donne en finalité une image beaucoup plus nette (voir le code ci-dessous).

#### main.rs

```
use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*, Result};

const GAUCHE: i32 = 81;
const HAUT: i32 = GAUCHE+1;
const DROITE: i32 = HAUT+1;
const BAS: i32 = DROITE+1;
const RETURN: i32 = 13;

fn main() -> Result<()> {
    let original = imread("/home/manu/Images/Papillon.jpg", IMREAD_COLOR)?;
    imshow("Original", &original)?;
    wait_key(0)?;

    let mut flou = Mat::default();
    let mut pixels = 3;
    let point = Point::new(-1, -1);
    loop {
        blur(&original, &mut flou, Size_{width:pixels, height:pixels}, point, BORDER_DEFAULT)?;
        imshow("Flou", &flou)?;
        match wait_key(50)? {
            DROITE => pixels+=2,
            GAUCHE => if pixels>3 { pixels-=2 },
            RETURN => break,
            _ => continue
        }
    }

    pixels=3;
    loop {
        gaussian_blur(&original, &mut flou, Size_{width:pixels, height:pixels}, 0., 0., BORDER_DEFAULT)?;
        imshow("Flou", &flou)?;
        let accentuer = (&original + 2.*(&original - &flou)).into_result()?;
        imshow("Accentuer", &accentuer)?;
        match wait_key(50)? {
            DROITE => pixels+=2,
            GAUCHE => if pixels>3 { pixels-=2 },
            RETURN => break,
            _ => continue
        }
    }

    pixels=5;
    loop {
        median_blur(&original, &mut flou, pixels)?;
        imshow("Flou", &flou)?;
    }
}
```

```

match wait_key(50)? {
    DROITE => pixels+=2,
    GAUCHE => if pixels>3 { pixels-=2 },
    RETURN => break,
    _ => continue
}

median_blur(&original, &mut flou, 11)?;
let mut largeur = 13;
let mut bord = 3.;
let mut gris = Mat::default();
cvt_color(&original, &mut gris, COLOR_BGR2GRAY, 0)?;
loop {
    let mut contour = Mat::default();
    let mut gris_3c = Mat::default();
    adaptive_threshold(&gris, &mut contour, 255., ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY, largeur, bord)?;
    cvt_color(&contour, &mut gris_3c, COLOR_GRAY2BGR, 0)?;
    imshow("Contour", &contour)?;
    let mut dessin = Mat::default();
    bitwise_and(&flou, &gris_3c, &mut dessin, &Mat::default())?;
    imshow("Flou", &dessin)?;
    match wait_key(50)? {
        DROITE => bord+=1.,
        GAUCHE => if bord>1. { bord-=1. },
        HAUT => largeur+=2,
        BAS => if largeur>3 { largeur+=2 },
        RETURN => break,
        _ => continue
    }
}
Ok(())
}

```



Pour effectuer les différents réglages de flou, nous ne passons plus par une barre de réglage, mais nous utilisons les touches du clavier qui gère les flèches d'orientation (Haut, Bas, Droite et Gauche). Pour l'accentuation, nous utilisons le flou gaussien pour rendre notre photo plus nette.

La fonction `median_blur()` est ici utile pour faire en sorte que notre photo devienne plutôt une planche de dessin, avec globalement l'atténuation des dégradés de couleur pour obtenir des aplats de couleur correspondant plus à ce que nous avons l'habitude de voir dans les bandes dessinées.

## DISCRIMINATION D'UNE PARTIE DE L'IMAGE

Lorsque nous possédons une image avec un fond relativement uniforme, il est possible de découper l'image originale pour ne conserver que la partie intéressante. C'est dans ce cas de figure que la notion de flou est généralement à exploiter pour que le fond soit encore plus homogène. Il suffit ensuite de choisir la bonne teinte du fond de l'image.

Je vous propose de réaliser cette expérience avec un livre que j'ai posé par terre sur un lino de teinte plutôt marron avec un motif relativement homogène qui possède juste des stries avec des couleurs qui suivent la teinte de base.

### main.rs

```

use opencv::{highgui::*, imgcodecs::*, core::*, imgproc::*, Result };
use opencv::types::VectorOfMat;

fn main() -> Result<()> {
    let original= imread("/home/manu/Images/livre.jpg", IMREAD_REDUCED_COLOR_4)?;
    imshow("Original", &original)?;
    wait_key(0)?;

    let (mut flou, mut hsv, mut masque) = (Mat::default(), Mat::default(), Mat::default());
    median_blur(&original, &mut flou, 11)?;
    imshow("Original", &flou)?;
    wait_key(0)?;

    cvt_color(&flou, &mut hsv, COLOR_BGR2HSV, 0)?;
    let (teinte, largeur) = (16., 16.);
    let seuil_bas = Scalar::new(teinte-largeur/2., 0., 0., 0.);
    let seuil_haut = Scalar::new(teinte+largeur/2., 255., 255., 0.);
    in_range(&hsv, &seuil_bas, &seuil_haut, &mut masque)?;
    imshow("Masque", &masque)?;
    wait_key(0)?;

    let (scalaire, point) = (Scalar::all(1.), Point::new(-1, -1));
    let (defaut, mut eroder) = (Mat::new_rows_cols_with_default(21, 17, CV_8U, scalaire)?, Mat::default());
    morphology_ex(&masque, &mut eroder, MORPH_CLOSE, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
    imshow("Masque", &eroder)?;
    wait_key(0)?;
}

```

```

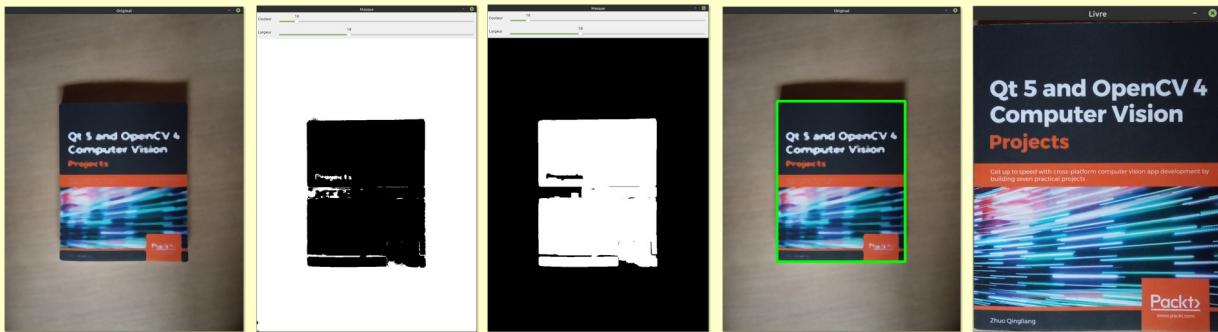
bitwise_not(&eroder, &mut masque, &Mat::default())?;
imshow("Masque", &masque)?;
wait_key(0)?;

let (mut contours, vert) = (VectorOfMat::new(), Scalar::new(0., 255., 0., 0.));
findContours(&masque, &mut contours, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, point)?;
let zone = bounding_rect(&contours.get(0)?);
rectangle(&mut flou, zone, vert, 3, FILLED, 0)?;
imshow("Original", &flou)?;
wait_key(0)?;

let livre = Mat::roi(&original, zone)?;
imshow("Original", &livre)?;
wait_key(0)?;

Ok(())
}

```



Le premier traitement consiste à discriminer le lino du reste de l'image puisque c'est la partie la plus homogène possible, le livre ayant une structure relativement complexe avec beaucoup d'informations et de couleurs différentes.

Dès le départ, il suffit d'utiliser la fonction **median blur()** pour faire en sorte que le lino possède juste une couleur moyenne sans toutes ces stries. Nous changeons ensuite de système de coloration pour basculer dans le modèle **HSV** qui nous permettra de choisir la teinte associée au lino quelque soit l'éclairage (remarquez bien l'ombre du livre).

Nous utilisons une nouvelle fonction **in\_range()** pour fabriquer notre masque en choisissant un seuil limite bas et un seuil limite haut, au travers des structures **Scalar**. Ainsi, vous pouvez sélectionner les limites de votre **teinte** en étant plus ou moins tolérant.

Pour prendre en compte les différentes **luminosités** du lino avec des parties très éclairées et d'autres complètement dans l'ombre, il suffit d'être cette fois-ci très tolérant, c'est-à-dire de prendre tout le spectre de 0 à 255. Nous pouvons appliquer le même spectre de tolérance pour la **saturation** (prendre toutes les couleurs plus ou moins prononcées).

Dans ce genre d'opération, il reste toujours de petits résidus que nous pouvons éliminer en érodant les pixels résiduels au travers de la fonction maintenant bien connue **morphology\_ex()**.

Il suffit ensuite d'inverser notre masque pour que le sujet principal ne soit plus le lino mais tout le reste, c'est-à-dire ici le livre que nous tentons finalement de discriminer, ceci avec la fonction **bitwise\_not()**.

Bien sûr, le masque lui-même ne sert à rien en tant que tel, si ce n'est pour retrouver sa délimitation, c'est-à-dire son contour, cela avec la fonction également bien connue **find\_contours()**.

Si la discrimination a bien été réalisée, nous devons retrouver un seul contour. Nous pouvons alors évaluer la zone rectangulaire qui nous intéresse, avec une nouvelle fonction **bounding\_rect()**, que nous visualisons en traçant un rectangle pour vérification.

L'objectif est atteint, nous pouvons découper alors notre image originale pour ne prendre que la partie du livre, avec la méthode statique **roi()** (*Region Of Interest*) de la structure **Mat**.

## ANNULER LA PERSPECTIVE D'UNE IMAGE

Dans l'exemple précédent, le livre était pris en photo vue de dessus afin qu'il corresponde facilement à une zone rectangulaire de l'image. La plupart du temps toutefois, la prise de vue ne sera pas aussi idyllique avec un cadrage qui montrera alors un livre avec des bords de fuite et une perspective. La délimitation du livre ressemble alors plus à un trapèze qu'à un rectangle parfait.

Nous allons reprendre le même sujet que précédemment avec les mêmes conditions d'éclairage, mais cette fois-ci, la prise de vue est réalisée avec un angle différent comme le montre la photo ci-contre.

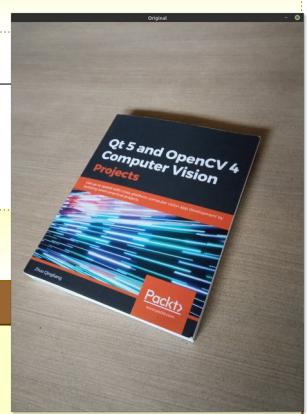
### main.rs

```

use opencv::{highgui:*, imgcodecs:*, core:*, imgproc:*, Result};
use opencv::types::VectorOfMat;

fn main() -> Result<()> {
    let original = imread("/home/manu/Images/livre-en-biais.jpg", IMREAD_REDUCED_COLOR_4)?;
    imshow("Original", &original)?;
}

```



```

wait_key(0)?;

let (mut flou, mut hsv, mut masque) = (Mat::default(), Mat::default(), Mat::default());
median_blur(&original, &mut flou, 11)?;
imshow("Original", &flou)?;
wait_key(0)?;

cvtColor(&flou, &mut hsv, COLOR_BGR2HSV, 0)?;
let (teinte, largeur) = (16., 16.);
let seuil_bas = Scalar::new(teinte-largeur/2., 0., 0., 0.);
let seuil_haut = Scalar::new(teinte+largeur/2., 255., 255., 0.);
inRange(&hsv, &seuil_bas, &seuil_haut, &mut masque)?;
imshow("Masque", &masque)?;
wait_key(0)?;

let (scalaire, point) = (Scalar::all(1.), Point::new(-1, -1));
let (defaut, mut eroder) = (Mat::new_rows_cols_with_default(21, 17, CV_8U, scalaire)?, Mat::default());
morphologyEx(&masque, &mut eroder, MORPH_CLOSE, &defaut, point, 1, BORDER_DEFAULT, scalaire)?;
imshow("Masque", &eroder)?;
wait_key(0)?;

bitwise_not(&eroder, &mut masque, &Mat::default())?;
imshow("Masque", &masque)?;
wait_key(0)?;

let (mut contours, mut trapeze, vert) = (VectorOfMat::new(), Mat::default(), Scalar::new(0., 255., 0., 0.));
findContours(&masque, &mut contours, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE, point)?;
let contour = contours.get(0)?;
approxPolyDP(&contour, &mut trapeze, 25., true)?;
polylines(&mut flou, &trapeze, true, vert, 3, FILLED, 0)?;
imshow("Original", &flou)?;
wait_key(0)?;

let points_originaux = conversion(trapeze.data_typed::<Point2i>()?)?;
let (hauteur, largeur) = trouver_rectangle(points_originaux.as_slice());
let points_ajustes = [Point{x: 0., y: 0.}, Point{x: 0., y: hauteur}, Point{x: largeur, y: hauteur}, Point{x: largeur, y: 0.}];
let (originaux, zone) = (Mat::from_slice(&points_originaux.as_slice())?, Mat::from_slice(&points_ajustes)?);

let perspective = getPerspectiveTransform(&originaux, &zone, DECOMP_LU)?;
let (mut livre, taille) = (Mat::default(), Size::new(largeur as i32, hauteur as i32));
warpPerspective(&original, &mut livre, &perspective, taille, INTER_CUBIC, BORDER_DEFAULT, scalaire)?;

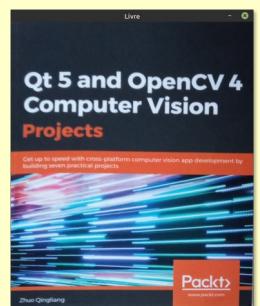
imshow("Livre", &livre)?;
wait_key(0)?;
Ok(())
}

fn conversion(points_2i: &[Point2i]) -> Vec<Point2f> {
    let mut points_2f = Vec::new();
    for point in points_2i {
        points_2f.push(Point2f::new(point.x as f32, point.y as f32));
    }
    points_2f
}

fn trouver_rectangle(points: &[Point2f]) -> (f32, f32) {
    let longueur_01 = hypotenuse(&points[1], &points[0]);
    let longueur_12 = hypotenuse(&points[2], &points[1]);
    let longueur_23 = hypotenuse(&points[3], &points[2]);
    let longueur_30 = hypotenuse(&points[0], &points[3]);
    ((longueur_01 + longueur_23)/2., (longueur_12 + longueur_30)/2.)
}

fn hypotenuse(p1: &Point2f, p2: &Point2f) -> f32 {
    let x2 = (p2.x - p1.x) * (p2.x - p1.x);
    let y2 = (p2.y - p1.y) * (p2.y - p1.y);
    f32::sqrt(x2 + y2)
}

```



Tout la première partie du code est totalement similaire au projet précédent. La différence intervient à partir du moment où nous récupérons le contour en forme de trapèze avec l'aide de la fonction `approxPolyDP()` en lieu et place de la fonction `boundingRect()`. Nous traçons alors la délimitation du contour avec la fonction `polylines()` au lieu de la fonction `rectangle()`.

À partir de là, l'objectif est de récupérer les points de ce **trapèze** issu de la **perspective** due à la prise de vue décalée et de faire en sorte que l'image interne devienne un **rectangle** parfait. Pour cela, nous devons recalculer les points en faisant en sorte de connaître la hauteur moyenne de cette image interne ainsi que la largeur moyenne. Nous calculons alors les **hypoténuses** respectives de chacun des côtés du **trapèze**. Cette **largeur** et cette **hauteur** estimée deviendra la référence de notre **rectangle**.

Une fois que nous avons les points du trapèze d'une part et les points du rectangle d'autre part, nous utilisons une fonction très puissante `get_perspective_transform()` qui, comme son nom l'indique, permet de fabriquer une matrice qui donne en interne l'algorithme permettant de transformer tous les pixels internes à cette zone en forme de trapèze pour qu'il soient recalibrer et devenir des pixels dans une zone rectangulaire équivalente.

La transformation effective se fait avec la fonction `warp_perspective()` qui prend en compte l'image originale, le résultat dans une nouvelle matrice, et surtout l'algorithme à suivre donné par la fonction précédente `get_perspective_transform()`. Par ailleurs, nous devons indiquer la taille du résultat. Nous pouvons alors prendre la **hauteur** et la **largeur** calculée précédemment. Nous pouvons aussi choisir d'autres dimensions puisque nous échantillonnons l'image interne avec le choix de l'interpolation.

Pour ces deux fonctions, comme nous l'avons déjà découvert précédemment, nous devons toujours passer par des matrices qui encapsulent les coordonnées des points (du trapèze et du rectangle) mais aussi qui encapsulent l'algorithme du traitement. **Rust** est vraiment très différent du C++, puisque pour ce dernier, nous passons plutôt par des tableaux de points.

**Dernière information importante :** pour la fonction `get_perspective_transform()`, afin que le résultat soit le plus fin possible, nous devons systématiquement travailler avec des points dont les coordonnées sont de type **f32**. C'est pour cette raison que les fonctions que j'ai fabriqué prennent en compte ce format là. Les matrices respectives encapsulent tous ces différents points dans ce nouveau format.

Pour la fonction personnelle `trouver_rectangle()` l'ordre des points du trapèze dans la matrice est proposée de la façon suivante : le premier point `p[0]` est toujours celui dont l'ordonnée est la plus petite, donc dans notre exemple, celui placé le plus en haut du trapèze. Les points suivants suivent alors le sens trigonométrique, `p[1]` le plus à gauche, `p[2]` le plus bas, `p[3]` le plus à droite.

## VIDÉO

Toute cette première partie concernait le traitement d'une image unique. Nous voici enfin sur la partie vidéo. Cette première partie était absolument indispensable pour comprendre tous les mécanismes de traitement. Nous n'avons pas perdu de temps, puisque de toute façon une vidéo est tout simplement une suite d'images. Tout ce que nous avons était capable de faire sur une image est bien entendu parfaitement valide pour une vidéo.

Pour une webcam la fréquence du flux des images est généralement de **30** par seconde. Ce qui nous laisse le temps pour faire des traitements spécifiques pour chacune des images. Je vous propose dans ce chapitre de faire uniquement le visuel normal d'une vidéo en choisissant un format de **640x480** en couleur.

### main.rs

```
use opencv::{highgui::*, core::*, videoio::*, Result};

fn main() -> Result<()> {
    let mut camera = VideoCapture::new(0, CAP_ANY)?;
    camera.set(CAP_PROP_FRAME_WIDTH, 640.)?;
    camera.set(CAP_PROP_FRAME_HEIGHT, 480.)?;
    let mut image = Mat::default();

    loop {
        camera.read(&mut image)?;
        imshow("Vidéo", &image)?;
        if wait_key(1)? == 13 { break; }
    }
    Ok(())
}
```

La classe qui gère la vidéo se nomme `VideoCapture` qui possède la méthode statique `new()` qui permet de préciser le choix de la caméra connectée sur un des ports **USB**. Si vous n'avez qu'une seule caméra, le numéro à proposer est tout simplement **0**, sinon vous proposer la valeur **1** pour la deuxième, etc. Vous pouvez régler votre caméra, notamment la résolution avec la méthode `set()` en proposant l'identification des bons paramètres comme par exemple `CAP_PROP_FRAME_WIDTH`.

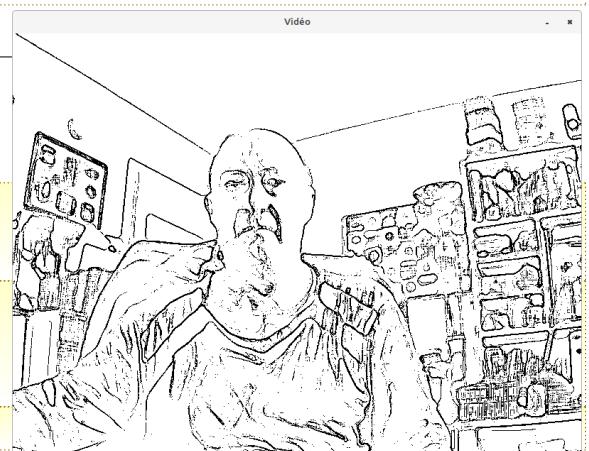
Enfin, pour récupérer chacune des images afin de pouvoir les afficher et éventuellement faire un traitement spécifique, il suffit d'appeler la méthode `read()`. Pour visualiser l'ensemble des images de la vidéo, il faut bien entendu proposer une boucle qui permet d'avoir une vidéo continue jusqu'à ce que l'utilisateur appuie sur la touche **Return**.

## MODE CONTOUR DANS UNE VIDÉO

Afin de bien démontrer qu'une vidéo est tout simplement qu'une suite d'images consécutives, je vous propose de faire une série de traitements sur chacune des images récupérées afin d'obtenir une vidéo où seul le contour des éléments est visualisé.

L'opération de base pour effectuer cet algorithme, comme nous l'avons découvert dans les chapitres précédents consiste à utiliser tout simplement la fonction `adaptive_threshold()`.

Pour cela, nous devons transformer au préalable notre image couleur en niveau de gris. Ensuite afin de limiter les contours qui n'ont pas d'intérêt et afin de garder uniquement les contours principaux, il est préférable de flouter l'image avec la fonction `median_blur()`.



## main.rs

```
use opencv::{highgui::*, core::*, videoio::*, imgproc::*, Result};

fn main() -> Result<()> {
    let mut camera = VideoCapture::new(0, CAP_ANY)?;
    camera.set(CAP_PROP_FRAME_WIDTH, 800.)?;
    camera.set(CAP_PROP_FRAME_HEIGHT, 600.)?;
    let (mut image, mut contours, mut gris) = (Mat::default(), Mat::default(), Mat::default());

    loop {
        camera.read(&mut image)?;
        cvt_color(&image, &mut gris, COLOR_BGR2GRAY, 0)?;
        median_blur(&gris, &mut image, 9)?;
        adaptive_threshold(&image, &mut contours, 255., ADAPTIVE_THRESH_GAUSSIAN_C, THRESH_BINARY, 5, 2.)?;
        imshow("Vidéo", &contours)?;
        if wait_key(1)? == 13 { break; }
    }

    Ok(())
}
```

Voyez que tous les traitements que nous avons appris s'appliquent parfaitement pour une vidéo. Rien de vraiment nouveau.

## DÉTECTION DE MOUVEMENTS – 1ÈRE PARTIE

Je vous propose maintenant de faire une application qui permet visualiser les parties qui sont en mouvement dans une vidéo. La détection des parties en mouvement sera visible avec des rectangles de couleurs. La première analyse que nous pouvons exploiter est de soustraire tout simplement deux images consécutives et de prendre en compte uniquement les zones qui sont différentes et d'éliminer les parties statiques.

*Le principe de fonctionnement est assez simple. Vous devez confronter deux images prises consécutivement dans le temps. Après avoir capturé la toute première image appelée **avant** qui sert de référence, dans la boucle, l'image qui suit appelée **après** est confrontée avec celle d'**avant** et le résultat sert d'analyse pour détecter les parties différentes de l'image.*

*Une fois que l'analyse et la détection réalisée, l'image d'**après** devient la nouvelle référence, elle est copiée dans l'image d'**avant** afin qu'elle soit confrontée à la nouvelle capture dans la boucle, et ainsi de suite.*

## main.rs

```
use opencv::{highgui::*, core::*, videoio::*, imgproc::*, Result};
use opencv::types::VectorOfMat;

fn main() -> Result<()> {
    let mut camera = VideoCapture::new(0, CAP_ANY)?;
    camera.set(CAP_PROP_FRAME_WIDTH, 800.)?;
    camera.set(CAP_PROP_FRAME_HEIGHT, 600.)?;

    let (mut avant, mut apres, mut gris) = (Mat::default(), Mat::default(), Mat::default());
    let (mut masque_av, mut masque_ap, mut diffusion) = (Mat::default(), Mat::default(), Mat::default());
    let point = Point::new(-1, -1);
    let scalaire = Scalar::all(1.);
    camera.read(&mut avant)?;

    loop {
        camera.read(&mut apres)?;
        let soustraction = ((&apres - &avant) * 7.).into_result()?.to_mat()?;
        cvt_color(&soustraction, &mut gris, COLOR_BGR2GRAY, 0)?;
        imshow("Zone", &gris)?;

        threshold(&gris, &mut masque_ap, 70., 255., THRESH_BINARY)?;
        let brosse = get_structuring_element(MORPH_RECT, Size::new(11, 11), point)?;
        erode(&masque_ap, &mut masque_av, &brosse, point, 1, BORDER_DEFAULT, scalaire)?;
        dilate(&masque_av, &mut masque_ap, &brosse, point, 7, BORDER_DEFAULT, scalaire)?;

        apres.copy_to(&mut diffusion)?;
        let mut contours = VectorOfMat::default();
        findContours(&masque_ap, &mut contours, RETR_TREE, CHAIN_APPROX_SIMPLE, Point::new(0, 0))?;
        for contour in contours {
            let zone = bounding_rect(&contour)?;
            rectangle(&mut diffusion, zone, Scalar::new(255., 0., 0., 0.), 5, FILLED, 0)?;
        }

        imshow("Détection", &diffusion)?;
        if wait_key(1)? == 13 { break; }
        apres.copy_to(&mut avant)?;
    }

    Ok(())
}
```

Le premier traitement important dans ce code consiste à faire une soustraction entre les deux matrices consécutives. Cette soustraction est en même temps amplifiée ( $x7$ ) afin que le résultat soit plus probant.

Je propose ensuite de réaliser un seuillage afin de garder uniquement les parties vraiment claires afin de bien discriminer les mouvements importants. Effectivement, les mouvements peuvent générer des changements d'éclairage sur d'autres parties de l'image.

Je rappelle que pour réaliser un seuillage, vous devez au préalable passer votre image d'analyse en niveau de gris.

Comme d'habitude, nous essayons d'éliminer les petits résidus qui n'ont aucun intérêt. Au lieu de prendre la fonction **morphology\_ex()**, nous passons successivement par la fonction **erode()** suivie de la fonction **dilate()** sauf que la dilatation est beaucoup plus importante que l'érosion.

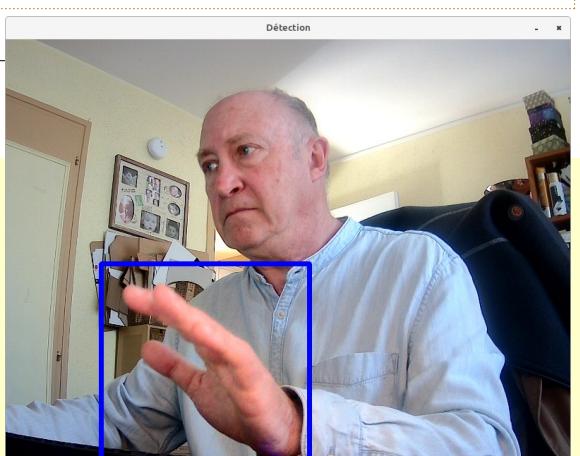
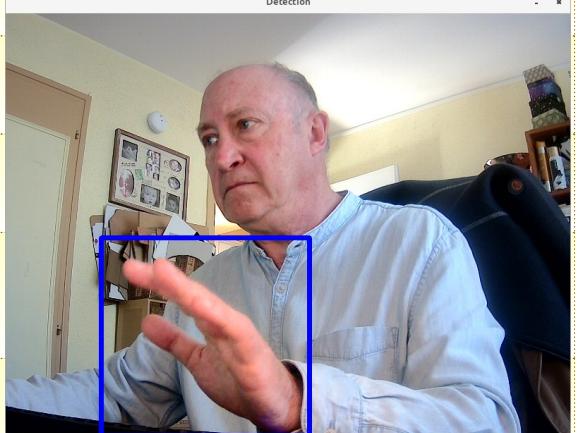
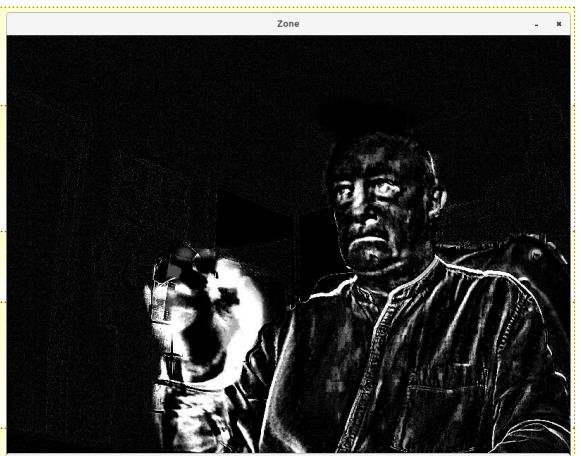
Le fait d'avoir une dilatation plus conséquente permet de bien entourer la zone en mouvement (zone légèrement plus grande).

La nouveauté ici concerne la fonction **get\_structuring\_element()**. Elle permet de fabriquer une matrice en précisant juste les dimensions ainsi que la forme globale à atteindre, ici une forme rectangulaire.

Nous aurions aussi pu fabriquer une matrice à l'aide de la fonction statique **new\_rows\_cols\_with\_default()** que nous avons déjà utilisé dans les chapitres précédents.

Quelque soit le choix que vous faites, cette matrice sert pour l'érosion et la dilatation. Seule l'occurrence est différente pour ces deux fonctions. Elle est de **1** (normale) pour l'érosion et **7** pour la dilatation (ce qui augmente son épaisseur).

Le reste du code n'appelle pas de commentaire particulier puisqu'il est maintenant bien connu. Nous pouvons effectuer différents réglages pour affiner votre résultat. Nous pouvons ainsi agir sur le **seuil mini** qui est actuellement de **70**. Nous pouvons aussi agir sur l'amplification de la soustraction qui est à **7**. Enfin, nous pouvons agir sur la largeur de la brosse qui est de **11x11** ou sur l'occurrence de la dilatation **x7**,



## DÉTECTION DE MOUVEMENTS – 2ÈME PARTIE

Je vous propose de réaliser de nouveau une détection en utilisant plus les compétences de OpenCV qui est capable de discerner le fond d'une vidéo du premier plan en mouvement.

Le principe utilisé par OpenCV pour la détection de mouvement est de séparer l'arrière plan du premier plan de votre vidéo. En effet, la plupart du temps, la « webcam » est placée une fois pour toute et ne bouge pas, ce qui fait que l'arrière plan est lui-même statique alors que les sujets du premier plan sont en mouvement.

C'est souvent le cas pour des « webcam » qui surveillent le trafic routier ou celles qui sont placées à proximité de tapis roulants dans des chaînes de production, par exemple. Dans la vidéo ci-contre, nous discriminons le mouvement de la main gauche.

Pour réaliser ce type de traitement, le but est d'arriver à discerner l'arrière plan du premier plan. Pour cela, il faut stocker un grand nombre d'images consécutives de votre vidéo pour évaluer les pixels qui restent statiques alors que d'autres parties de la vidéo n'ont jamais le même visuel. Même si le principe paraît simple, il faut souligner la difficulté de l'algorithme, notamment lorsque les lumières et les ombres changent au cours de la journée. Il faut pouvoir prendre en compte ce genre de critères.

Vous l'avez compris, la difficulté est de bien discriminer l'arrière plan. Pour cela OpenCV disposent de structures spécialisées hiérarchisées, avec la structure de base abstraite **BackgroundSubtractor**, et deux structures concrètes, respectivement : la structure **BackgroundSubtractorKNN** qui dispose d'un algorithme qui permet de prendre en compte les pixels connexes à chaque pixel analysé, et la structure **BackgroundSubtractorMOG2** qui propose un algorithme similaire en intégrant en plus le système gaussien qui permet généralement d'aboutir à de meilleurs résultats, mais avec un temps d'exécution plus long.

Il existe bien d'autres algorithmes qui font partie de cette hiérarchie qui possèdent des spécificités bien particulières, mais vous devez pour cela rajouter les modules adéquats. Dans la majorité des cas, les deux algorithmes de base proposés sont suffisants.

Pour que la discrimination de l'arrière plan puisse se faire correctement, nous devons stocker l'ensemble des images successives de la séquence vidéo en prenant en compte un nombre d'images relativement conséquent. Pour cela, nous utilisons un conteneur spécifique d'OpenCV nommé **Ptr**, qui comme beaucoup d'autres conteneurs est paramétré (utilisation de modèles).

**main.rs**

```
use opencv::{highgui::*, core::*, video::*, videoio::*, imgproc::*, Result};
use opencv::types::VectorOfMat;
```

```

fn main() -> Result<()> {
    let mut camera = VideoCapture::new(0, CAP_ANY)?;
    camera.set(CAP_PROP_FRAME_WIDTH, 800.)?;
    camera.set(CAP_PROP_FRAME_HEIGHT, 600.)?;
    let mut fond = create_background_subtractor_mog2(500, 16., true)?;
    let (mut image, mut masque, mut erosion) = (Mat::default(), Mat::default(), Mat::default());
    let mut contours = VectorOfMat::default();
    let point = Point::new(-1, -1);
    let brosse = get_structuring_element(MORPH_RECT, Size::new(15, 15), point)?;
    let scalaire = Scalar::all(1.);

    loop {
        camera.read(&mut image)?;
        BackgroundSubtractorMOG2::apply(&mut fond, &image, &mut masque, -1.)?;
        imshow("Masque", &masque)?;

        erode(&masque, &mut erosion, &brosse, point, 1, BORDER_DEFAULT, scalaire)?;
        dilate(&erosion, &mut masque, &brosse, point, 3, BORDER_DEFAULT, scalaire)?;

        findContours(&masque, &mut contours, RETR_TREE, CHAIN_APPROX_SIMPLE, Point::new(0, 0))?;
        for contour in &contours {
            let zone = boundingRect(&contour)?;
            rectangle(&mut image, zone, Scalar::new(255., 0., 0., 0.), 5, FILLED, 0)?;
        }
        imshow("Détection", &image)?;
        if waitKey(1)? == 13 { break; }
    }
    Ok(())
}

```

Notre code est relativement similaire au code précédent avec plus de simplification puisque nous n'avons plus besoin de changer de mode de couleur et de réaliser de seuillage. Du coup, nous avons également moins de matrices à prendre en compte.

Avant de rentrer dans la boucle de capture, vous devez précisez comment vous désirez discriminer l'arrière plan de votre vidéo du sujet principal. La fonction `create_background_subtractor_mog2()` permet de réaliser cette opération. Vous remarquez que cette fonction correspond au choix de la structure `BackgroundSubtractorMOG2`. L'objectif de cette structure, avec la collection `Ptr`, et de la fonction associée est de réaliser un masque permettant de filtrer le premier plan de l'arrière plan, comme cela vous est montré ci-dessous :

Cette fonction possède trois paramètres :

Le premier paramètre `history` qui, comme son nom l'indique, permet de préciser le nombre des dernières images de la vidéo à conserver pour réaliser la discrimination arrière plan / premier plan, par défaut réglé à 500 (historique des 500 dernières images). Bien entendu, plus le nombre est important, plus la discrimination est précise, mais il faut aussi beaucoup plus de mémoire et beaucoup plus de temps d'analyse.

Le deuxième paramètre, `var_threshold` correspond à la valeur du seuil à prendre en compte niveau bas pour fabriquer le masque, par défaut un seuil bas de 16.

Le dernier paramètre `detect_shadows` permet de prendre en compte ou non l'évolution de l'ombre dans la capture.

Lorsque vous rentrez dans la boucle des captures d'images de la vidéo, la première chose à faire est de réaliser une mise à jour de l'historique grâce à la méthode `apply()` de la classe `Ptr`, avec pour arguments, la nouvelle image, et le masque résultant de la nouvelle analyse.

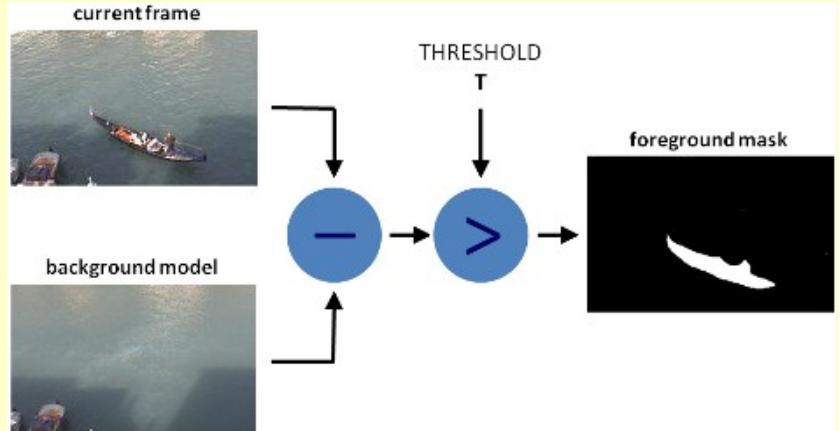
La séparation entre l'arrière plan et le sujet principal n'est jamais parfaitement nette. Il est préférables de réduire les petits pixels qui peuvent apparaître dans le masque, ce que nous appelons couramment le bruit (valeur des pixels faibles ou de trop petites tailles) avec les appels successifs de `erode()` et `dilate()`.

La première chose importante à réaliser est de mettre en place l'historique qui permet d'identifier le fond de la vidéo, avec le conteneur `Ptr<BackgroundSubtractorMOG2>` à l'aide de la fonction spécifique `create_background_subtractor_mog2()`. Vous devez spécifier le nombre d'images qui permet de faire une bonne analyse afin de bien discriminer le fond (par défaut 500), vous précisez également un seuil limite bas (par défaut à 16) et enfin vous précisez si vous désirez prendre en compte l'ombre.

Pour résumer, une fois que le conteneur d'historique est mis en place, la première chose à faire à chaque capture d'image de la vidéo est de réaliser une mise à jour constante de l'historique grâce à la méthode `apply()` de la structure `Ptr`, avec pour arguments, la nouvelle image à prendre en compte, et le masque résultant de la nouvelle analyse.

La particularité de cette méthode `apply()`, c'est qu'elle existe en plusieurs exemplaires, ce qui pose problème dans le monde de Rust. En C++, nous aurions écrit plutôt `fond.apply(image, masque)`. Afin de bien indiquer que nous désirons prendre la méthode `apply()` associée à la structure `BackgroundSubtractorMOG2`, nous devons préfixer l'appel par le type de cette structure.

Dans ce cas, vous devez donner alors le nom de la variable associée, ici `fond`, puisque vous savez que les méthodes commencent toujours par le premier paramètre `&self` ou `&mut self`. C'est ici une astuce du langage qui permet de spécifier le nom de l'objet correspondant à la manœuvre



Si vous expérimentez ces deux approches de détection de mouvements, vous remarquerez que la deuxième est quand même beaucoup plus performante puisqu'elle connaît bien le fond de la vidéo qui est statique par rapport à des passages intempestifs de nouveaux éléments en premier plan de la vidéo. Vous pouvez aussi changer le nombre d'image à prendre en compte dans votre historique, par exemple 1000 images consécutives.

Pour revenir à la fonction `create_background_subtractor_mog2()`, je vous montre les visuels des masques pour des valeurs de seuil respectivement de 3, de 16 et de 50 pour vous bien comment se construit le masque résultant. Vous pouvez ainsi décider de l'intensité du mouvement.



## DÉTECTION DES PERSONNES, DES VISAGES ET DES YEUX

Une des grandes fonctionnalités très à la mode actuellement est la reconnaissance automatique des visages, des yeux, du corps, etc. d'une ou plusieurs personnes. Beaucoup de systèmes, comme les appareils photos numériques notamment, utilisent les compétences d'OpenCV dans ce domaine particulier.

### La détection d'objets utilise le principe des cascades de Haar.

Le classificateur en cascade de Haar est l'une des premières et des plus connues des méthodes de détection d'objet. Publié en 2001 par Paul Viola et Michael Jones, l'article est devenu une référence dans le domaine de la détection d'objet.

Inventée à l'origine pour détecter des visages, elle est également utilisée pour détecter beaucoup d'autres types d'objets comme des voitures ou des avions, par exemple.

En tant que procédé d'apprentissage supervisé, la méthode de Viola et Jones nécessite de quelques centaines à plusieurs milliers d'exemples de l'objet que l'on souhaite détecter, pour entraîner un classificateur. Une fois son apprentissage réalisé, ce classificateur est utilisé pour détecter la présence éventuelle de l'objet dans une image en parcourant celle-ci de manière exhaustive, à toutes les positions, toutes les orientations et dans toutes les tailles possibles.

La méthode, en tant que méthode d'apprentissage supervisé (intelligence artificielle), est divisée en deux étapes : une étape d'apprentissage du classificateur basé sur un grand nombre d'exemples positifs (c'est-à-dire les objets d'intérêt, par exemple des visages) mais aussi d'exemples négatifs, avec une phase de détection par application de ce classificateur à des images inconnues.

### Fonctionnement des cascades de Haar

La méthode de Viola-Jones consiste à parcourir une image à l'aide d'une fenêtre glissante (de 24 x 24 pixels dans l'algorithme original) et de déterminer si un visage y est présent. Cette méthode consiste à parcourir l'ensemble de l'image en calculant un certain nombre de caractéristiques dans des zones se chevauchant. Elle a la particularité d'utiliser des caractéristiques très simples mais très nombreuses.

### Caractéristiques pseudo-Haar

Plutôt que de travailler directement sur les valeurs de pixels, Viola et Jones proposent d'utiliser des caractéristiques très simples : les caractéristiques pseudo-Haar, montrées ci-contre. Ces caractéristiques sont calculées par la différence des sommes de pixels de deux ou plusieurs zones rectangulaires adjacentes.

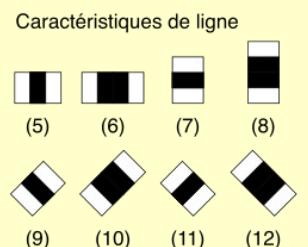
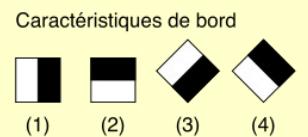
### Cascade de classificateur

Le terme cascade dans le nom du classificateur signifie qu'il est le résultat de plusieurs classificateurs plus simples (que l'on appelle stages) qui sont appliqués successivement sur une région d'intérêt jusqu'à ce qu'un des stages échoue où qu'ils soient tous validés.

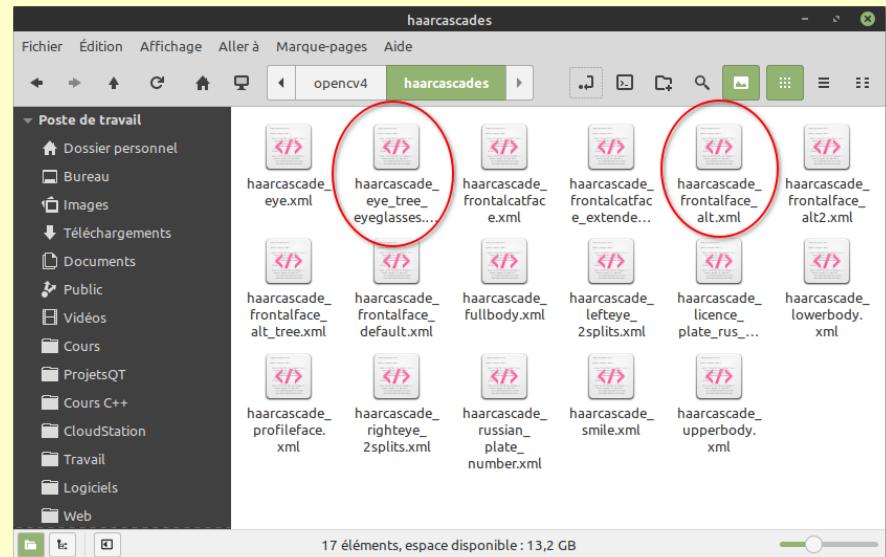
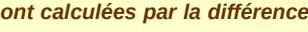
L'idée est de rejeter les zones ne contenant pas l'objet avec le moins possible de calculs. Le premier classificateur est donc le plus optimisé et permet de rejeter rapidement une zone si l'objet recherché ne s'y trouve pas. Si potentiellement l'objet s'y trouve, alors le deuxième classificateur est utilisé et ainsi de suite jusqu'au dernier.

### Apprentissage du classificateur

Une étape préliminaire et très importante est l'apprentissage du classificateur. Il s'agit d'entraîner le classificateur afin de le sensibiliser à ce que vous voulez détecter, ici des visages. Pour cela, il est placé dans deux situations. La première



### Caractéristiques centre-pourtour



où une énorme quantité de cas positifs lui sont présentés et la deuxième où, à l'inverse, une énorme quantité de cas négatifs lui sont également présentés. Concrètement, il s'agit d'une banque d'images contenant les visages de personnes est passée en revue afin d'entraîner le classificateur. Ensuite, une autre banque d'images ne contenant pas cette fois-ci de visages humains est également passée en revue.

### Les classificateurs

L'entraînement d'un classificateur est une étape longue. Il est nécessaire de réunir et d'annoter un grand nombre d'image contenant l'objet à détecter.

Dans le cas présent, Viola et Jones ont entraînés leur classificateur à l'aide d'une banque d'images du MIT. Il en résulte un classificateur sensibles aux visages humain. Il se présente sous la forme d'un fichier XML. Dans le cadre d'OpenCV, heureusement vous disposez de quelques classificateurs déjà entraînés. Vous les retrouvez dans le dossier nommé « haarcascades ». Pour notre projet, nous aurons besoin de deux classificateurs, un pour le visage et un pour les yeux.

main.rs

```
use opencv::highgui::*;
use opencv::core::*;
use opencv::videoio::*;
use opencv::imgproc::*;
use opencv::objdetect::*;
use opencv::Result;
use opencv::types::VectorOfRect;

fn main() -> Result<()> {
    let mut camera = VideoCapture::new(0, CAP_ANY)?;
    camera.set(CAP_PROP_FRAME_WIDTH, 800.)?;
    camera.set(CAP_PROP_FRAME_HEIGHT, 600.)?;

    let mut visages =
        CascadeClassifier::new("/opt/opencv/share/opencv4/haarcascades/haarcascade_frontalface_alt.xml")?;
    let mut yeux =
        CascadeClassifier::new("/opt/opencv/share/opencv4/haarcascades/haarcascade_eye_tree_eyeglasses.xml")?;

    let (mut image, mut gris) = (Mat::default(), Mat::default());
    let mut visages_detectes = VectorOfRect::new();
    let mut yeux_detectes = VectorOfRect::new();
    let taille = Size::new(0, 0);
    let (rouge, bleu) = (Scalar::new(0., 0., 255., 0.), Scalar::new(255., 0., 0., 0.));

    loop {
        camera.read(&mut image)?;
        cvt_color(&image, &mut gris, COLOR_BGR2GRAY, 0)?;

        visages.detect_multi_scale(&gris, &mut visages_detectes, 1.1, 3, 0, taille, taille)?;
        for visage in &visages_detectes {
            rectangle(&mut image, visage, rouge, 2, FILLED, 0)?;
            let image_visage = Mat::roi(&gris, visage)?;
            yeux.detect_multi_scale(&image_visage, &mut yeux_detectes, 1.1, 3, 0, taille, taille)?;
            for oeil in &yeux_detectes {
                let centre = Point::new(visage.x+oeil.x+oeil.width/2, visage.y+oeil.y+oeil.height/2);
                let rayon = (oeil.width+oeil.height) / 4;
                circle(&mut image, centre, rayon, bleu, 2, FILLED, 0)?;
            }
        }
        imshow("Visages", &image)?;
        if wait_key(1)? == 13 { break; }
    }
    Ok(())
}
```

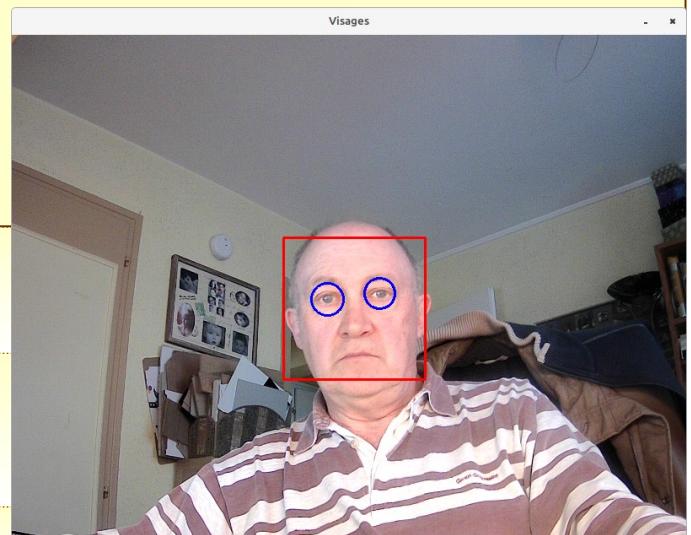
Pour résoudre notre projet, nous créons deux classificateurs associés à la détection souhaitée savoir, les visages et les yeux. Les objets sont générés à l'aide des fichiers XML correspondants dans le répertoire prévu à cet effet.

Nous réalisons la détection des visages et de leurs yeux de tous les personnages présents dans la vidéo de capture. La structure **CascadeClassifier** possède la méthode **detect\_multi\_scale()** qui permet de détecter tous les objets identifiés par les classificateurs décrits dans les documents XML choisis.

Vous devez au préalable changer votre mode de coloration pour basculer en niveau de gris.

L'appel de la première méthode **detect\_multi\_scale()** de l'objet **visages**, comme nous nous en doutons, s'occupe de détecter tous les visages présents dans l'ensemble de l'image capturée. Ensuite, pour chacun des visages, la deuxième méthode **detect\_multi\_scale()** de l'objet **yeux**, s'occupe de détecter les yeux, mais cette fois-ci, non pas sur toute l'image capturée, mais uniquement à partir de la zone rectangulaire de chacun des visages identifiés.

L'enregistrement des zones de capture, respectivement pour les visages et pour les yeux, se fait au travers d'une collection de rectangles avec la structure spécifique **VectorOfRect**.



## RETRouver un QRCode

Il est souvent intéressant de retrouver et d'analyser un **QrCode** dans une image ou dans une vidéo d'un produit alimentaire ou d'un colis qui défile devant une caméra afin de vérifier la validité du produit.

Pour cela, il existe dans le module **objdetect**, une structure spécifique **QRCodeDetector** qui permet de localiser l'endroit où se situe le **QrCode** dans votre image grâce à la méthode **detect()** et qui permet après cette localisation de le décoder au moyen d'une autre méthode **decode()**.

La méthode **detect()** renvoie une valeur booléenne suivant le résultat de l'analyse de l'image. Dans le cas de la bonne localisation du **QrCode**, la méthode retourne bien la valeur **true** pour indiquer que le **QrCode** a bien été identifié et donne également les points de cette localisation dans une matrice, qui va servir par la suite.

Nous n'utilisons la méthode **decode()** que si le **QrCode** a bien été identifié en nous servant de la matrice qui encapsule les points de localisation. Cette méthode retourne un vecteur contenant la liste des caractères représentant le message du **QrCode** si cette méthode arrive à bien l'interpréter, sinon elle renvoie un vecteur vide dans le cas où l'image de ce **QrCode** est trop flou.

Enfin, la particularité de cette méthode, c'est qu'elle fournit une image en modèle réduit, parfaitement nette et de très petite dimension pour un usage ultérieur. Plus l'image est petite, plus l'interprétation de ce **QrCode** se fera rapidement. Bien entendu, cette image est générée que dans le cas de la bonne interprétation du **QrCode** original.

main.rs

```
use opencv::{highgui::*, core::*, objdetect::*, imgcodecs::*, Result};

fn main() -> Result<()> {
    let image = imread("/home/manu/Images/multi-codes.png", IMREAD_GRAYSCALE)?;
    let mut qrcode = QRCodeDetector::default()?;
    let mut points = Mat::default();

    if qrcode.detect(&image, &mut points)? {
        println!("{}",&points.data_typed::<Point2f>());
        let mut resultat = Mat::default();
        let decodage = qrcode.decode(&image, &points, &mut resultat)?;
        imshow("Résultat", &resultat)?;
        println!("{}",&String::from_utf8(decodage.unwrap()));
    }

    imshow("Image", &image)?;
    wait_key(0)?;
    Ok(())
}
```

Résultat

```
[Point_ { x: 339.9465, y: 184.56378 }, Point_ { x: 535.9177, y: 212.55968 }, Point_ { x: 507.00748, y: 409.14926 },
Point_ { x: 311.1855, y: 380.13858 }]
http://remy-manu.no-ip.biz
```



Vous remarquez dans ce code que nous avons récupéré une image en mode **niveau de gris**. Ce programme fonctionne aussi parfaitement bien en mode de couleurs **BGR**. Voici ci-dessous un autre exemple de détection de **QrCode** au travers d'une vidéo.

main.rs

```
use opencv::{highgui::*, core::*, objdetect::*, videoio::*, Result};

fn main() -> Result<()> {
    let mut camera = VideoCapture::new(0, CAP_ANY)?;
    camera.set(CAP_PROP_FRAME_WIDTH, 1280.)?;
    camera.set(CAP_PROP_FRAME_HEIGHT, 720.)?;

    let mut qrcode = QRCodeDetector::default()?;
    let (mut image, mut points) = (Mat::default(), Mat::default());

    loop {
        camera.read(&mut image)?;
        if qrcode.detect(&image, &mut points)? {
            println!("{}",&points.data_typed::<Point2f>());
            let decodage = qrcode.decode(&image, &points, &mut Mat::default())?;
            if !decodage.is_empty() {
                println!("{}",&decodage);
                println!("{}",&String::from_utf8(decodage.unwrap()));
                break;
            }
        }
        imshow("Image", &image)?;
        wait_key(1)?;
    }
    Ok(())
}
```

**Résultat**

```
[Point { x: 442.0, y: 180.0 }, Point { x: 244.0, y: 79.0 }, Point { x: 267.40985, y: 10.0 }, Point { x: 564.0, y: 15.0 }]
[Point { x: 220.0, y: 81.0 }, Point { x: 115.0, y: 111.0 }, Point { x: -22.592953, y: -14.912727 }, Point { x: 194.0, y: 17.0 }]
[Point { x: 561.0, y: 442.0 }, Point { x: 545.0, y: 602.0 }, Point { x: 372.04654, y: 579.243 }, Point { x: 314.0, y: 461.0 }]
[Point { x: 628.9794, y: 312.06958 }, Point { x: 629.0, y: 509.0 }, Point { x: 423.8208, y: 502.0687 }, Point { x: 432.0, y: 298.0 }]
[Point { x: 616.0, y: 304.0 }, Point { x: 610.0, y: 509.0 }, Point { x: 392.76962, y: 504.6331 }, Point { x: 406.2725, y: 291.7028 }]
[104, 116, 116, 112, 58, 47, 47, 114, 101, 109, 121, 45, 109, 97, 110, 117, 46, 110, 111, 45, 105, 112, 46, 98, 105, 122]
http://remy-manu.no-ip.biz
```

Dans le code de la page précédente, nous essayons d'évaluer un **QrCode** dans une vidéo. Cette fois-ci, nous restons en mode **BGR**. Sur l'image ci-dessous vous remarquez qu'une partie du **QrCode** est cachée. Attention, même si un **QrCode** est identifié, il faut bien vérifier qu'il puisse être interprété.

Dans le résultat ci-dessus, vous remarquez que le **QrCode** est plusieurs fois identifié, par contre il n'est pas tout de suite interprétable. Il faut attendre le cinquième passage pour arriver à trouver le résultat final de l'interprétation complète. Une fois que nous avons trouvé le résultat l'application se termine automatiquement.

