MODULE 5

Détection et analyse de contours

Objectifs de ce module :

- ✓ ***Amélioration des contours.***
- ✓ ***Détection des contours***
- ✓ ***Propriété mesurables des régions et descripteur de forme.***

Table des matières

Sujets ***Page***

Amélioration des contours

Les contours sont un aspect important de l'analyse de forme et de la reconnaissance de forme. En effet, si je vous demande de m'identifier les formes triangulaires dans un ensemble de figures géométriques, vous parviendrez en quelques secondes à identifier les triangles dans la scène. Le cerveau a identifié ce qui constituait le pourtour de la forme pour isoler les côtés et ainsi compter le nombre d'arrêtes de la figure.

C'est donc en analysant le contours des objets que l'on parvient à identifier les formes. OpenCV possède plusieurs fonctions et classes qui permettent d'isoler et d'identifier les contours. Nous verrons dans les prochaines sections comment y parvenir.

Renforcement des contours par la méthode de Sobel, Laplace et Canny

Filtre de Sobel

Le filtre de Sobel est un opérateur utilisé en traitement d'image pour la détection de contours. Il s'agit d'un des opérateurs les plus simples qui donne toutefois des résultats corrects.

Pour faire simple, l'opérateur calcule le gradient de l'intensité de chaque pixel. Ceci indique la direction de la plus forte variation du clair au sombre, ainsi que le taux de changement dans cette direction. On connaît alors les points de changement soudain de luminosité, correspondant probablement à des bords, ainsi que l'orientation de ces bords.

En termes mathématiques, le gradient d'une fonction de deux variables (ici l'intensité en fonction des coordonnées de l'image) est un vecteur de dimension 2 dont les coordonnées sont les dérivées selon les directions horizontale et verticale. En chaque point, le gradient pointe dans la direction du plus fort changement d'intensité, et sa longueur représente le taux de variation dans cette direction. Le gradient dans une zone d'intensité constante est donc nul. Au niveau d'un contour, le gradient traverse le contour, des intensités les plus sombres aux intensités les plus claires.

Voici les matrices :

$$\begin{matrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{matrix}$$

Matrice de Sobel pour les contours horizontaux

$$\begin{matrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{matrix}$$

Matrice de Sobel pour contours verticaux

Exemple d'application de Sobel

```
#include <opencv2\core\core.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <opencv2\imgproc\imgproc.hpp>

using namespace cv;

int main()
{
	Mat ImgSource;
	Mat ImgGris;
	Mat ImgResultat;

	ImgSource = imread("sign.jpg", CV_LOAD_IMAGE_COLOR);

	 if (!ImgSource.data)
	{
			cout << "Erreur dans ouverture du fichier source" << endl;
	    return -1;
	}

	//blur(ImgSource, ImgSource, Size(5, 5));  // Applique une matrice de convolution de 3 x 3 pour le blur

	cvtColor(ImgSource, ImgGris, CV_BGR2GRAY);

	Sobel(ImgGris, ImgResultat, CV_16S, 0, 1, 3); // On cherche les contours horizontaux y = 1
	// On utilise la fonction suivante pour les contours verticaux x = 1
	//Sobel(ImgGris, ImgResultat, CV_16S, 1, 0, 3); // On cherche les contours verticaux x = 1

	convertScaleAbs( ImgResultat, ImgResultat);
	blur(ImgResultat, ImgResultat, Size(3, 3));  // Pas absolument nécessaire mais peut être utile


	namedWindow("Image Source", CV_WINDOW_AUTOSIZE);
	namedWindow("Image Resultat", CV_WINDOW_AUTOSIZE);


	imshow("Image Source", ImgSource);
	imshow("Image Resultat", ImgResultat);

	waitKey(10000);

	return 0;
```

Résultat de l'application de Sobel

Image originale

Image après Sobel horizontal

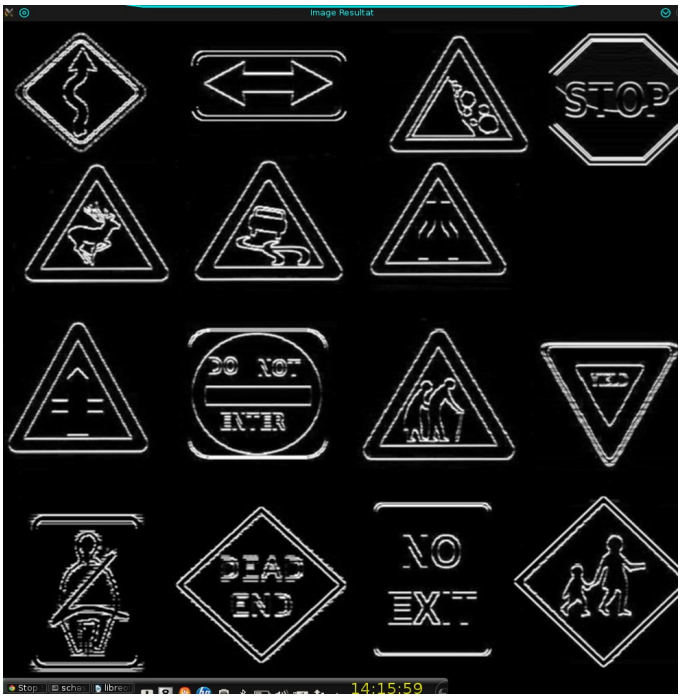
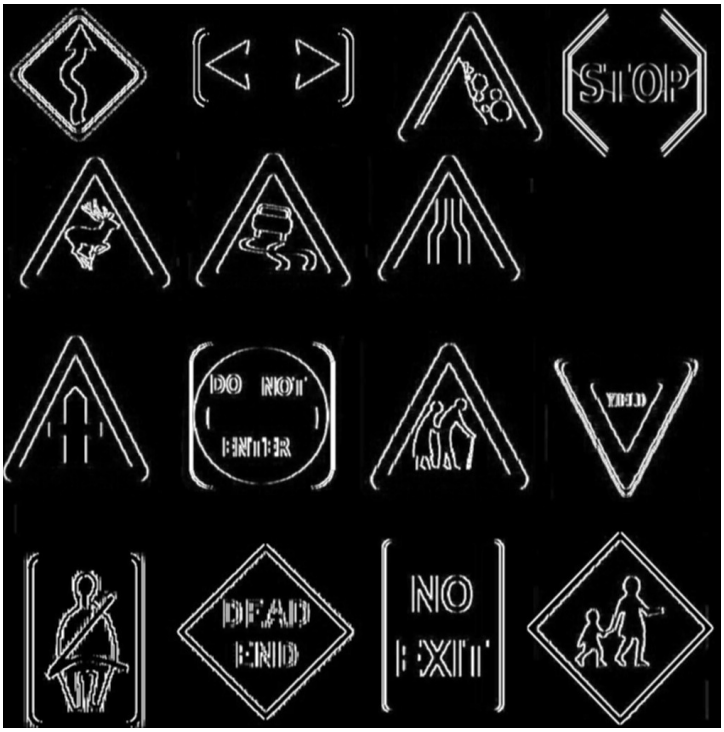


Image après Sobel vertical



Matrice de convolution de Laplace

Le filtre Laplacien est un filtre de convolution particulier utilisé pour mettre en valeur les détails qui ont une variation rapide de luminosité. Il est donc idéal pour rendre visible les contours des objets. D'un point de vue mathématique, le Laplacien est une dérivée d'ordre 2, à deux dimensions et se note $\triangle I(x,y)$ ou $\nabla 2I(x,y)$.

Dans le cas du traitement d'image, l'image de départ f(i,j) n'est pas une fonction continue, mais une fonction discrète à cause de la numérisation effectuée. Mais on peut tout de même obtenir la dérivée seconde (soit le laplacien) avec une bonne approximation grâce aux noyaux de convolution suivants (entre autres).

Matrice de Laplace :

$$\begin{matrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{matrix}$$

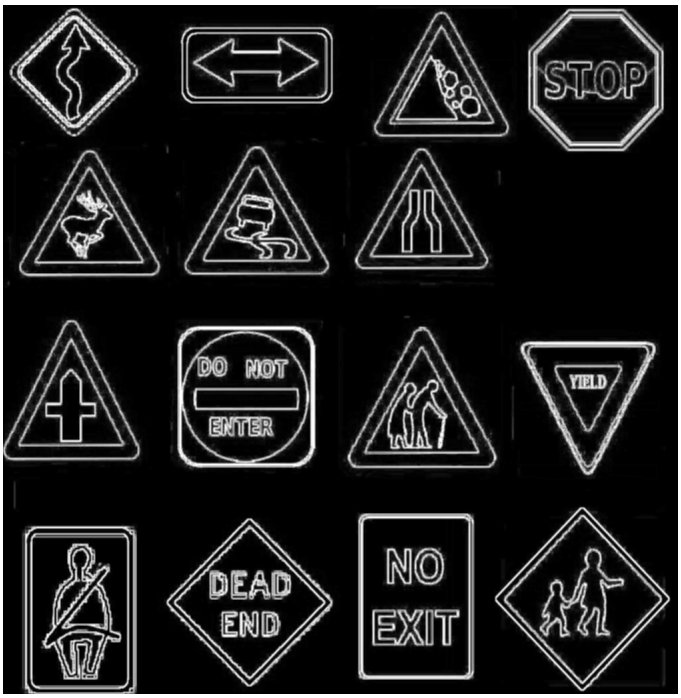
Exemple d'application de Laplace

```
#include <opencv2\core\core.hpp>
#include <opencv2\highgui\highgui.hpp>
#include <opencv2\imgproc\imgproc.hpp>

using namespace cv;

int main()
{
	Mat ImgSource;
	Mat ImgGris;
	Mat ImgResultat;

	ImgSource = imread("sign.jpg", CV_LOAD_IMAGE_COLOR);

	if (!ImgSource.data)
	{
		cout << "Erreur dans ouverture du fichier source" << endl;
		return -1;
	}

	//blur(ImgSource, ImgSource, Size(5, 5));  // Applique une matrice de convolution de 3 x 3 pour le blur

	cvtColor(ImgSource, ImgGris, CV_BGR2GRAY);

	Laplacian(ImgGris, ImgResultat, CV_16S, 3);

	convertScaleAbs( ImgResultat, ImgResultat);

	blur(ImgResultat, ImgResultat, Size(3, 3));  // Pas absolument nécessaire mais peut être utile


	namedWindow("Image Source", CV_WINDOW_AUTOSIZE);
	namedWindow("Image Resultat", CV_WINDOW_AUTOSIZE);


	imshow("Image Source", ImgSource);
	imshow("Image Resultat", ImgResultat);

	waitKey(10000);

	return 0;
}
```

Résultat de l'application de Laplace

Image originale

Image après Laplace



Renforcement des contours par la méthode de Canny

- Ouvrir l'image à traiter
- Convertir l'image originale en ton de gris (si nécessaire)
- Appliquer, au besoin, une binarisation (threshold)
- Appliquer l'algorithme de Canny (Canny)

Exemple :

```
int main()
{
	Mat ImgSource;
	Mat ImgGris;
	Mat ImgResultat;

	ImgSource = imread("sign.jpg", CV_LOAD_IMAGE_COLOR);

	if (!ImgSource.data)
	{
		cout << "Erreur dans ouverture du fichier source" << endl;
		return -1;
	}

	//blur(ImgSource, ImgSource, Size(5, 5));  // Applique une matrice de convolution de 3 x 3 pour le blur

	cvtColor(ImgSource, ImgGris, CV_BGR2GRAY);

	blur(ImgGris, ImgGris, Size(3, 3));  // Pas absolument nécessaire mais peut être utile

	threshold(ImgGris, ImgGris, 140, 255, CV_THRESH_BINARY_INV);
	Canny(ImgGris, ImgResultat, 10, 20);

	namedWindow("Image Source", CV_WINDOW_AUTOSIZE);
	namedWindow("Image Resultat", CV_WINDOW_AUTOSIZE);


	imshow("Image Source", ImgSource);
	imshow("Image Resultat", ImgResultat);

	waitKey(10000);

	return 0;
}
```

Analyse des contours

Après avoir procéder à améliorer les contours et les isoler adéquatement du fond de l'image, nous sommes maintenant prêt à trouver les contours des objets dans l'image.

Trouver les contours des objets (findContours)

Cette fonction permet de trouver les contours en remplissant un vecteur de point qui correspond aux points qui sont définies sur le contour extérieur de l'objet. En fait, plusieurs constantes sont définies dans OpenCV et permettent de trouver les points extérieurs de l'objet mais aussi ceux qui font parties des « trous » internes de l'objet. On peut aussi vouloir créer une hiérarchie d'objet en passant au travers une liste des contours extérieurs jusqu'au contours intérieurs.

Dans le module précédent, nous avons appliqué un filtre de Canny pour ressortir les pixels faisant partie du contour d'un objet. On peut ensuite, avec la fonction « findContours » trouver les points sur ce contour. Cette séquence de point est essentielle dans la reconnaissance de forme. Nous allons donc appliquer cette fonction aux objets de façon à déterminer leur forme.

Syntaxe de la fonction findContours

void findContours(InputOutputArray **image**, OutputArrayOfArrays **contours**, OutputArray **hierarchy**, int **mode**, int **method**, Point **offset**=Point())

Paramètres:

- **image** – C'est l'image source sur 8 bits et un canal. L'image est traitée en format binaire, ce qui veut dire qu'elle doit avoir subi une binarisation. Vous devrez donc utiliser une fonction parmi les suivantes : inRange(), threshold(), Canny().
- **contours** – Le vecteur de points qui constitue le contour.
- Hierarchy – C'est le vecteurs de contours qui permet d'obtenir la hiérarchie des contours d'un objet. Ainsi, pour un contour i, hierarchy[i][0] est le contours extérieurs, hierarchy[i][1] est le deuxième contours intérieurs de l'objet et ainsi de suite. Si les objets n'ont pas de contours intérieurs, ce paramètre n'est pas utilisé.
- **mode** – Le mode de récupération du contour. Une constante parmi les suivantes :
    - **CV_RETR_EXTERNAL** récupère le contour extérieur seulement.

- **CV_RETR_LIST** récupère tous les contours sans établir de ralation hiérarchie.

- **CV_RETR_CCOMP** récupère tous les contours et les organise dans une hiérarchie à deux niveaux. Le premier niveau constitue le contour extérieur alors que le deuxième niveau constitue le contour des trous présents dans l'objet.

- **methode** – La méthode d'approximation du contour. Utilisez l'une des constantes suivantes :
  - **CV_CHAIN_APPROX_NONE**
  - **CV_CHAIN_APPROX_SIMPLE**
  - **CV_CHAIN_APPROX_TC89_L1,CV_CHAIN_APPROX_TC89_KCOS**.
- **offset** – Le point de départ qui constitue le premier point du contour. Par défaut, c'est le point de coordonnée (0,0). Peut-être utile lorsqu'on travaille avec des régions d'intérêt.

Étapes pour trouver les contours

Voici les étapes habituellement réalisées pour trouver les contours des objets :

Étape 1:

Définir un vecteur de vecteur de point.

```
vector<vector<Point>> Contours;
```

Étape 2 :

Charger l'image en ton de gris (très suggéré) ou la transformer en ton de gris par la suite si vous l'avez chargé initialement en couleur.

Étape 3:

Appliquer un « blurring » pour enlever le bruit.

Binariser l'image en filtrant uniquement les contours. Il s'agit habituellement d'appliquer la fonction Canny.

Étape 4:

Appliquer la fonction « findContours ».

```
findContours(ImgCanny, Contours, CV_RETR_EXTERNAL,
                                        CV_CHAIN_APPROX_SIMPLE);
```

Étape 5:

Trouver les caractéristiques des objets.

Par exemple, le nombre d'objet total trouvé correspond au nombre d'élément dans le vecteur. Ainsi, « Contours.size() » donnera le nombre d'objet total dans l'image.

Exemple :

```
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>

using namespace cv;

int main(int argc, char *argv[])
{
            Mat ImgSource;
            Mat ImgGris;

            Mat ImgResultat;

  vector<vector<Point> > Contours;

  ImgSource = imread("chessboard.jpg", CV_LOAD_IMAGE_COLOR);
  if (!ImgSource.data)
  {
      cout << "Erreur dans ouverture du fichier source" << endl;
      return -1;
  }

  blur(ImgSource, ImgSource, Size(5, 5));  // Applique une matrice de convolution de 3 x 3
                                           //pour le blur

  cvtColor(ImgSource, ImgGris, CV_BGR2GRAY);

  threshold(ImgGris, ImgGris, 230, 255, CV_THRESH_BINARY_INV);
  Canny(ImgGris, ImgResultat, 10,20);

  findContours(ImgResultat, Contours, CV_RETR_LIST ,CV_CHAIN_APPROX_SIMPLE);
```

La fonction « findContours » permet de trouver les points qui constituent le contour des objets. En fait, le deuxième paramètre « contours » est un vecteur de vecteur de point qui contient les points des contours de chacun des objets.

Ainsi,

Contours[0] représente la séquence de points de l'objet « 0 ».

Contours[1] représente la séquence de points de l'objet « 1 » et ainsi de suite.

Après l'appel de findContours, nous avons déjà certains résultats qui sont accessibles par le vecteur de vecteur. Le nombre d'objet trouvé dans l'image est facilement disponible. En effet, le nombre d'élément que contient le vecteur correspond directement au nombre d'objet trouvé dans l'image.

Caractéristiques et descripteurs de forme

Nombre d'objet trouvé

Le bout de code suivant nous permet de déterminer le nombre d'objet présent dans l'image après y avoir appliquer la détection « canny ».

```
cout << "Nombre d'objet : " << Contours.size() << endl;
```

Dessiner le contour des objets trouvés

La fonction « drawContours » permet de tracer les contours qui ont été trouvés. On parcours le vecteur de contours et pour chaque objet trouvé, on dessine la séquence de points de l'objet comme montré ci-dessous :

```
findContours(ImgResultat, Contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE);

Mat ImgResultat2 = Mat::zeros( ImgResultat.size(), CV_8UC3 );

for( int i = 0; i< Contours.size(); i++ )
{
   Scalar color = Scalar(0, 255, 255);
   drawContours( ImgResultat2, Contours, i, color, 2, 8 );
}

namedWindow("Image Source", CV_WINDOW_AUTOSIZE);
namedWindow("Image Resultat", CV_WINDOW_AUTOSIZE);

imshow("Image Source", ImgSource);
imshow("Image Resultat", ImgResultat2);

waitKey(0);
```

Paramètre de « drawContours » :

- Image qui recevra les résultats : ImgResultat
- Le vecteur de contour : Contours
- L'indice de l'objet contenu dans le vecteur de contours. Ainsi, la valeur 0, signifie Contours[0].
- La couleur utilisée pour dessiner le contour : color. Ce paramètre est de type « Scalar » qui correspond à un triplet qui représente la couleur (B,G,R).
- Les deux derniers paramètres correspondent à l'épaisseur du trait et au type de ligne respectivement. Leur valeur par défaut est de 1 pour l'épaisseur et de 8 pour le type de ligne.

On créé également une matrice couleur d'une dimension égale à la dimension de l'image qui contient les contours trouvés par la fonction findContours.

Mat ImgResultat2 = Mat::zeros(ImgResultat.size(), CV_8UC3);

Déterminer le type de forme des figures géométriques simples

Nous allons nous attarder à trouver le nombre de segment qui compose une figure géométrique simple (rectangle, cercle, triangle). Pour ce faire, nous utiliserons à nouveau la fonction findContours pour trouver les contours et nous approximerons le nombre de point qui forme les contours. La fonction findContours trouve beaucoup trop de point pour déterminer les segments minimaux qui compose la figure. Il devient donc difficile de déterminer vraiment le nombre de segment qui nous permettra de conclure sur le type de figure.

Nous devons donc utiliser une fonction qui nous permettra de réduire le nombre de point utilisé sur le pourtour de la figure.

Fonction: approxPolyDP – Approximation du nombre de point sur le contour d'un objet

void approxPolyDP(InputArray **curve**, OutputArray **approxCurve**, double **epsilon**, bool **closed**)

Paramètres:

- **curve :** Le vecteur 2D des points qui sont sous forme d'objet Mat ou contenu dans un vecteur (vector).
- **approxCurve :** Un vecteur de point qui recevra les points approximés. Il peut être défini comme suit : vector <point> approx;
- double **epsilon : Une valeur qui représente un pourcentage de point à prendre pour l'approximation. De façon générale, une valeur de 2% à 5% du périmètre donne de bons résultats. Elle peut être calculée de la façon suivante : arcLength(Contours[i], true) * 0.02**
- **bool closed : Booléen qui indique si la figure est fermée ou non.**

La boucle de traitement de contour devient :

```
int NbRect = 0, NbTriangle = 0;

  for( int i = 0; i< Contours.size(); i++ )
  {
      approxPolyDP(Contours[i], approx, arcLength(Contours[i], true) * 0.05, true);

     if (approx.size() == 4) // Si il y a 4 points, c'est alors une forme rectangulaire
        NbRect++;

     if (approx.size() == 3) // Ici, c'est nécessairement une forme triangulaire
        NbTriangle++;

  }

  cout << "Il y a " << NbRect << "forme rectangulaire et " << NbTriangle << " forme triangulaire" << endl;
```

Le nombre d'élément dans le vecteur « approx » correspond à l'approximation du nombre de point qui a été utilisé pour l'identification des segments. Or le nombre de points, qui est en fait le nombre d'éléments du vecteur (méthode size() , correspond au nombre de segments et donc au nombre de côté qui constitue la forme.

Ainsi :

- Si approx.size() retourne 4, nous pouvons affirmer que la forme est de type rectangulaire
- Si approx.size() retourne 3, nous pouvons affirmer qu'il s'agit d'une forme triangulaire.
- Tout autre valeur correspondra à la forme recherché (par exemple, si le vecteur contient 5 points, il s,agit probablement d'un pentagone). Il existe cependant des cas où nous ne pourrons pas déterminer avec certitude la forme de l'objet géométrique. Il faudra recourir à d'autres critères de calcul que nous allons voir dans les prochaines sections.

Autres caractéristiques (aire, périmètre, centre de masse, rectangularité et circularité)

Périmètre

Il s'agit de trouver l'ensemble des pixels qui font partie de l'objet et qui ont au moins un voisin qui appartient au fond (background). Il faut cependant déterminer si les pixels sont connectés sous une forme de 4-adjacence ou de 8-adjacence.

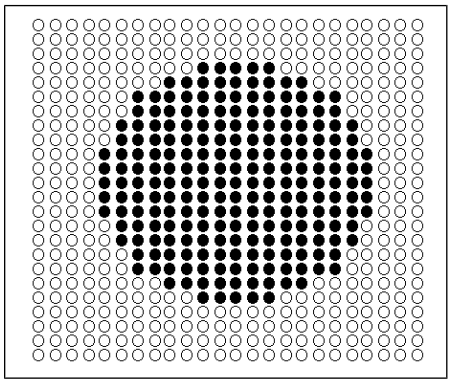
Algorithme:

- Identifier les pixels sur la région.
- Chaque pixel situé sur le périmètre et ses voisins sont comparés à un "template" pour connaître la distance (poids) qui doit être assignée à ce pixel.
- Faire la somme de toutes les distances (poids) trouvées précédemment. Ceci donne le périmètre approximé.

Technique utilisée pour trouver la distance (poid) associée à un pixel

L'algorithme calculant le périmètre d'un objet, de même que plusieurs autres algorithmes que nous verrons plus loin, utilise une technique qui permet d'associer un poids à un pixel. Ce poids est donné en relation avec la position du pixel analysé et de ses pixels voisins. Utilisons un exemple pour fixer les idées.

Soit le cercle suivant:



L'image ci-contre représente un objet circulaire ayant un diamètre de 17 pixels. Le périmètre mathématique de cet objet devrait donner:

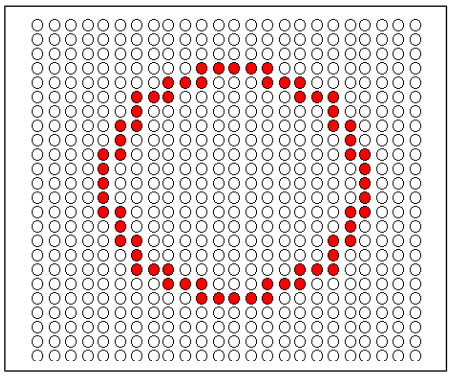
$$Périmètre = 2\pi r$$

où r : le rayon du cercle.

Or le diamètre étant de 17 pixels, on obtient:

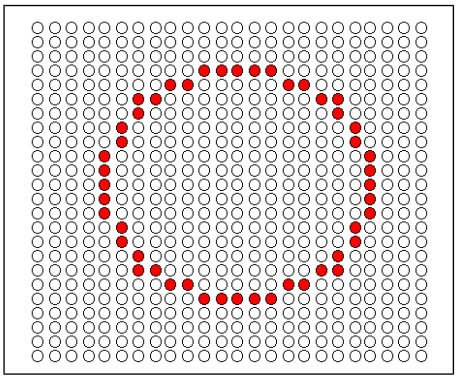
$$Périmètre = 2 * \pi * 8.5 = 53.407$$

Vous savez que les pixels peuvent être connectés sous une forme 4-adjacente ou 8-adjacente. Les deux figures suivantes montrent le résultat du périmètre précédent en utilisant la 4-adjacence et la 8-adjacence:



4 adjacence

Périmètre = 64



8 adjacence

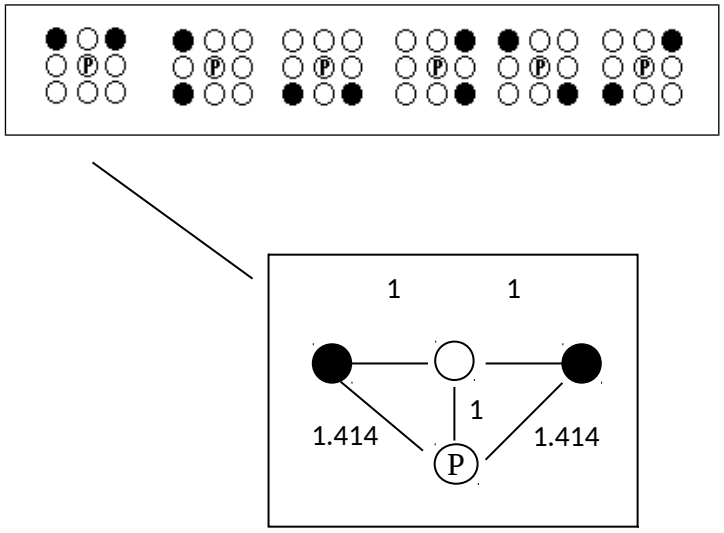
Périmètre = 48

On voit clairement qu'il y a une grande erreur de propagée par rapport à la valeur réelle mathématique. En plus, les valeurs diffèrent entre la 8 et la 4 adjacence!! La raison en est fort simple, c'est qu'un pixel représente non pas une distance linéaire mais plutôt une surface. Une hypothèse de base que l'on peut accepter concernant les pixels, c'est que ceux-ci représentent une région rectangulaire et il y a, en conséquence, plusieurs façons qu'un périmètre d'une figure puisse passer au travers d'un pixel.

Nous sommes tentés de compter le nombre de pixel tout simplement et c'est là l'erreur. La solution consiste à donner un poids au pixel considéré selon le "pattern" observé avec ses voisins. Ainsi, dépendamment du "pattern", un pixel aura tantôt la valeur 1, tantôt la valeur 0.5, etc.

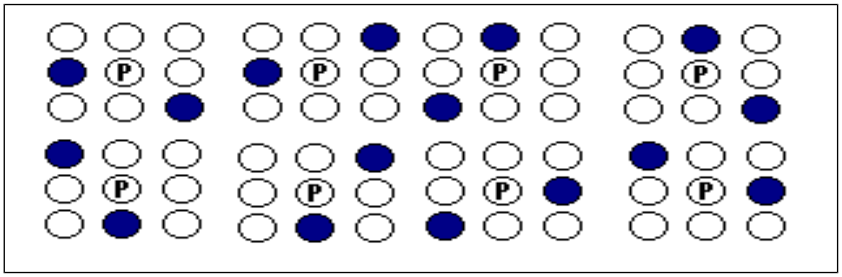
C'est donc plus logique d'essayer de trouver quelle est la distance représentée par un pixel et de donner un poids à ce dernier. Il en résulte les cas suivants:

- Pixel ayant des voisins en diagonale



Les pixels marqués P ont une "longueur" effective de 1.414. En effet, comme le montre la figure ci-contre, le pixel P obtient un poids qui représente une distance de 1 unité par 1 unité.

- Pixel ayant des voisins en diagonale et sur la verticale ou l'horizontale



Le pixel P obtient un poids équivalent à ½ fois la longueur de 1 pixel d'unité plus ½ fois la longueur d'une diagonale de 1.414 unités. Ainsi, le pixel P obtient un poids de:

$$P_{poids} = ½ + 1.414/2 = 0.5 + 0.707 = 1.207$$

Reprenons l'exemple du cercle ayant un diamètre de 17 pixels. En utilisant la 8 adjacence et en donnant des poids au pixel selon les techniques précédentes, on obtient un périmètre de 54.6 dont ce dernier est à 2% prêt de la valeur réelle.

Avec OpenCV, le périmètre de chaque objet peut être trouvé en utilisant la fonction « arcLength ».

Fonction: arcLength

Paramètres:

- **Contours** – Le vecteur 2D des points qui sont sous forme d'objet Mat ou contenu dans un vecteur (vector).
- **closed** – Booléen indiquant si le contour a considéré est une figure fermée ou ouverte.

Exemple :

La boucle de traitement sur les contours ressemble à celle-ci :

```
findContours(ImgResultat, Contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE);

for( int i = 0; i< Contours.size(); i++ )
{
    Perimetre = arcLength(Contours[i], true);

    cout << "L'objet #" << i << " a un perimetre de " << Perimetre << endl;

}
```

Aire

L'aire d'une région est exprimée par le nombre de pixels appartenant à une région déterminée. L'aire physique est obtenue en multipliant le nombre de pixels par l'aire réelle de ce pixel. Par exemple, une région contenant 10 pixels, chacun correspondant à 1.2 cm2 aura une aire physique réelle de 10 X 1.2 = 12 cm2. Cependant, il arrive souvent que l'aire est exprimée en nombre de pixel seulement.

Algorithme :

- Une région est marquée d'une valeur.
- Le nombre de pixel ayant cette valeur est ensuite compté et donne l'aire.

En OpenCV, il existe une fonction toute simple qui se nomme « contourArea » pour réaliser ce calcul.

La boucle de traitement des contours ressemble à celle-ci :

```
findContours(ImgResultat, Contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE);

for( int i = 0; i< Contours.size(); i++ )
{
      Aire = contourArea(Contours[i]);

      cout << "L'objet #" << i << " a une aire de " << Aire << endl;

}
```

Fonction: contourArea

Paramètres:

- **Contours** – Le vecteur 2D des points qui sont sous forme d'objet Mat ou contenu dans un vecteur (vector).
- **closed** – Booléen indiquant le sens de rotation (horaire ou anti-horaire). Par défaut, la valeur est a « false » et ceci correspond à la valeur absolue de l'aire.

Centre de masse

Le centre de masse d'un objet représente le point milieu de l'objet étudié. Il correspond au point qui permettrait de faire tenir physiquement en équilibre l'objet en question. Ce n'est pas toujours le point central car le centre de masse dépend de la forme de l'objet mais aussi de sa densité au niveau physique. Cependant, comme c'est une image, la densité n'est évidemment pas un facteur.

Pour les formes usuelles comme des cercles ou des rectangles ou triangles, le point central est relativement simple à trouver et correspond habituellement au centre réel de l'objet. Il en est autrement pour des formes plus complexes.

Nous utiliserons la classe « moments » qui permet de calculer les moments d'ordre 3 sur des polynômes ou des figures.

Voici cette classe :

```
class Moments
{
public:
   Moments();
   Moments(double m00, double m10, double m01, double m20, double m11,
           double m02, double m30, double m21, double m12, double m03 );
   Moments( const CvMoments& moments );
   operator CvMoments() const;

   // spatial moments
   double  m00, m10, m01, m20, m11, m02, m30, m21, m12, m03;
   // central moments
   double  mu20, mu11, mu02, mu30, mu21, mu12, mu03;
   // central normalized moments
   double  nu20, nu11, nu02, nu30, nu21, nu12, nu03;
}
```

Fonction: moments

Paramètres:

- **Contours** – Le vecteur 2D des points qui sont sous forme d'objet Mat ou contenu dans un vecteur (vector).
- **binaryImage** – Booléen qui spécifie si l'image est une image binaire ou non, donc sur laquelle nous avons effectué un seuil de filtrage (threshold).

La fonction retourne les descripteurs de moments dans une structure qui contient les éléments recherchés.

Dans le cas d'une image, les moments spatiaux $\texttt{Moments::m}_{ji}$ sont calculés comme suit :

$$\texttt{m}_{ji} = \sum_{x,y} \left(\texttt{array}(x,y) \cdot x^j \cdot y^i \right)$$

où array(x,y) représente la valeur du pixel dans l'image traitée. Ainsi, dans le cas d'une image en ton de gris, la valeur de array(x,y) sera une valeur située entre 0 et 255 inclusivement.

Les centres de masse sont calculés comme suit :

$$\mathrm{mu}_{ji} = \sum_{x,y} \left(\mathrm{array}(x,y) \cdot (x - \bar{x})^j \cdot (y - \bar{y})^i \right)$$

où $(\bar{x}, \bar{y})$ est la coordonée du centre de masse:

$$\bar{x} = \frac{m_{10}}{m_{00}},\ \bar{y} = \frac{m_{01}}{m_{00}}$$

Ainsi, la valeur m00 représente l'aire d'un objet et le centre de masse est calculé comme suit :

x = m10 / m00 et y = m01 / m00

où les valeurs de x et de y représente le point milieu de l'objet.

La boucle de traitement des contours ressemble à ceci :

```
Mat ImgGris, ImgSource;
vector <vector <Point>> Contours;

cvtColor(ImgSource, ImgGris, CV_BGR2GRAY);

threshold(ImgGris, ImgGris, 220*, 255, CV_THRESH_BINARY_INV);
Canny(ImgGris, ImgResultat, 10,20);

findContours(ImgResultat, Contours, CV_RETR_EXTERNAL , CV_CHAIN_APPROX_SIMPLE);

for( int i = 0; i< Contours.size(); i++ )
{
   Moments lesMoments = moments(Contours[i], true );

   double Aire = lesMoments.m00;

   double Xcentre = lesMoments.m10 / Aire;
   double Ycentre = lesMoments.m01 / Aire;

   cout << "L'objet #" << i << " a son centre de masse situé a (" << Xcentre << "," << Ycentre << ") "<< endl;
   cout << "L'objet #" << i << " a une aire de " << Aire << endl;
}
```

* Il est à noter que la valeur du seuil de 220 dans l'appel de la fonction « threshold » devra sûrement être modifiée selon vos besoins et l'image que vous traitez.

Rectangularité

La rectangularité est une mesure qui nous permet de déterminer à quel point une forme est de type rectangulaire. Cette mesure permet d'obtenir une approximation de la forme rectangulaire d'un objet. On peut donc savoir si un objet est de type rectangulaire ou s'il s'en éloigne.

La rectangularité est définie comme le rapport de l'aire de l'objet sur l'aire du plus petit rectangle qui englobe l'objet. Mathématiquement :

$$R = A_{objet} / A_{min}$$

Où : A_{objet} est l'aire de l'objet et A_{min} est l'aire du plus petit rectangle qui englobe l'objet à analyser.

A_{min} , c'est-à-dire, l'aire du rectangle minimal peut être calculé grâce à la classe « RotatedRect » et à la fonction « minAreaRect » de OpenCV. La classe « RotatedRect » ressemble à ceci :

```
class CV_EXPORTS RotatedRect
{
public:
    //! various constructors
    RotatedRect();
    RotatedRect(const Point2f& center, const Size2f& size, float angle);
    RotatedRect(const CvBox2D& box);

    //retourne les 4 points qui forment le rectangle
    void points(Point2f pts[]) const;
    //Retourne les coordonnées du rectangle minimum sans angle
    Rect boundingRect() const;
    //Compatibilité avec les anciennes structures de OpenCV < 2.0
    operator CvBox2D() const;

    Point2f center; //Le centre de masse
    Size2f size;    //Longueur et largeur du rectangle (avec rotation)
    float angle;    //l'angle de rotation.  Les multiples de 90 degrés signifient que le rectangle est droit
                      et n'est pas en angle
};
```

Fonction: RotatedRect `minAreaRect`(InputArray **points**) – Calcule le plus petit rectangle (avec rotation) qui englobe un objet

Paramètres:

- **points** – Le vecteur 2D des points qui sont sous forme d'objet Mat ou contenu dans un vecteur (vector).

Fonction: Rect `boundingRect`(InputArray **points**) – Calcule le plus petit rectangle (sans rotation) qui englobe un objet

Paramètres:

- **points** – Le vecteur 2D des points qui sont sous forme d'objet Mat ou contenu dans un vecteur (vector).

Exemple :

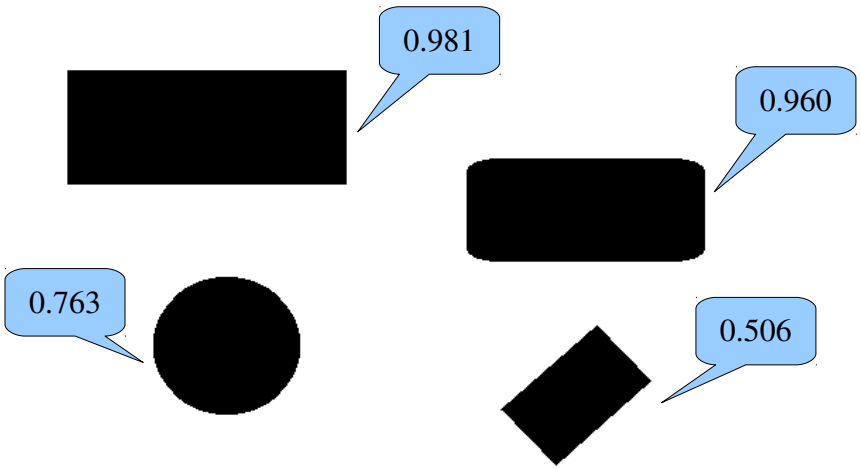
Voici l'image analysée :

En utilisant « boundingRect », on calcule le plus petit rectangle qui englobe chacun des objets ci-dessus. On applique ensuite la méthode des rapports d'aire.

Voici le bout de code :

```
findContours(ImgResultat, Contours, CV_RETR_EXTERNAL , CV_CHAIN_APPROX_SIMPLE);

for( int i = 0; i< Contours.size(); i++ )
{
  // Test de la rectangularité
  Rect leRect = boundingRect(Contours[i]);

  double Aire_leRect = leRect.width * leRect.height;

  double Aire_Rect_Contour = contourArea(Contours[i]);

  double Rectangularite = Aire_Rect_Contour / Aire_leRect;

  cout << "Rectangularite de l'objet # : " << i << " : " << Rectangularite << endl;

 }
```

Nous obtenons les valeurs suivantes :



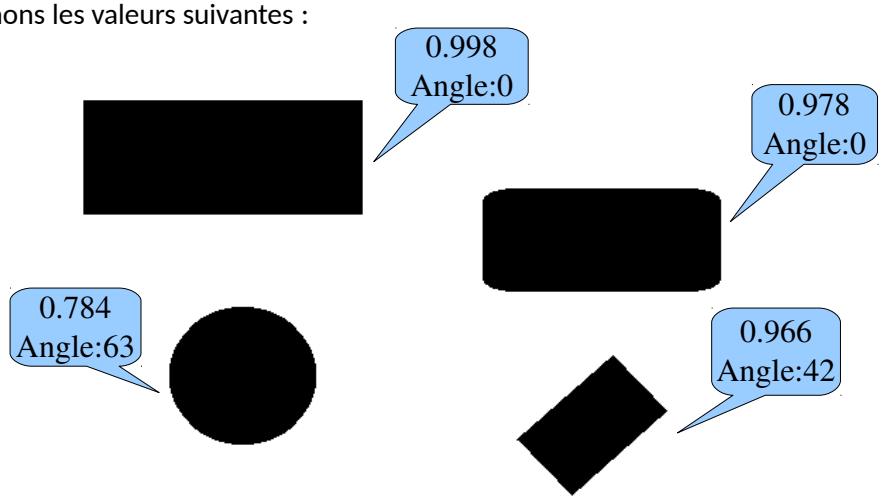
Les valeurs sont excellentes pour les rectangles qui ont leur base parallèle à l'axe des X de l'image. Nous obtenons effectivement des valeurs de 98% et 96% pour les 2 figures du haut même si celle du haut à droite possède des coin arrondis. Le cercle est définitivement à exclure avec 76%. La dernière forme en bas à droite est somme toute assez rectangulaire mais possède un angle. La fonction « boundingRect » n'en tient pas compte et on calcule un taux de l'ordre de 50% soit plus faible encore que la forme circulaire! On devrait obtenir des valeurs non loin de celles obtenues pour les deux formes rectangulaires du haut.

Pour tenir compte de l'angle, on utilise alors « minAreaRect ». En fait, on peut même utiliser cette fonction sur les formes qui n'ont pas d'angle à la base.

Reprenons les calculs avec « minAreaRect »

```
findContours(ImgResultat, Contours, CV_RETR_EXTERNAL , CV_CHAIN_APPROX_SIMPLE);

vector<Point> approx;
int NbRect = 0, NbTriangle = 0;
Mat ImgResultat2 = Mat::zeros( ImgResultat.size(), CV_8UC3 );

for( int i = 0; i< Contours.size(); i++ )
{
  // Test de la rectangularité
  RotatedRect leRect = minAreaRect(Contours[i]);

  double Aire_leRect = leRect.size.width * leRect.size.height;

  double Aire_Rect_Contour = contourArea(Contours[i]);

  double Rectangularite = Aire_Rect_Contour / Aire_leRect;

  cout << "Rectangularite de l'objet # : " << i << " : " << Rectangularite << endl;

  cout << "Angle de la forme # " << i << ": " << leRect.angle;

}
```

Nous obtenons les valeurs suivantes :



Les valeurs sont nettement plus satisfaisantes:D. On peut dire sans trop se tromper qu'il y 3 formes rectangulaires dans cette image. En utilisant la fonction « minAreaRect », non seulement nous obtenons de meilleurs valeurs mais nous connaissons en plus l'angle de rotation de chaque figure. Ainsi, le rectangle en bas à droite a effectué une rotation de 42 degrés environ.

Circularité

Cette méthode permet de trouver le degré de **circularité** d'une pièce. En fait, c'est le calcul du rapport de l'aire du plus petit cercle qui englobe l'objet circulaire sur l'aire de l'objet circulaire original. Dans le cas d'un cercle parfait, ce rapport devrait s'approcher d'une valeur égale à 1.

Voici le bout de code qui nous aide à réaliser ce calcul :

```
findContours(ImgResultat, Contours, CV_RETR_EXTERNAL , CV_CHAIN_APPROX_SIMPLE);


  Point2f CentreCercle;
  float RayonCercle;

  for( int i = 0; i< Contours.size(); i++ )
  {
    // Test de la circularité

    minEnclosingCircle(Contours[i], CentreCercle, RayonCercle );

    double Aire_Cercle = 3.1415 * RayonCercle * RayonCercle;

    double Aire_Cercle2 = contourArea(Contours[i]);

    double Circularite = Aire_Cercle2 / Aire_Cercle;

    cout << "Circularité de l'objet # : " << i << " : " << Circularite << endl;


  }
```

Point2f CentreCercle : Représente le point milieu (centre de masse) de plus petit cercle qui sera trouvée. Cette variable est une structure de type « Point2f »

float RayonCercle : Représente le rayon du cercle de centre « CentreCercle ».

L'appel à « minEnclosingCircle » nous permet d'entourer l'objet à analyser d'un cercle qui englobe totalement cet objet. Le centre du cercle et son rayon est alors assigné par le biais de l'appel de la fonction et les deux paramètres passées par référence.

double Aire_Cercle = 3.1415 * RayonCercle * RayonCercle;

Ceci calcule l'aire du cercle retournée par la fonction « minEnclosingCircle ».

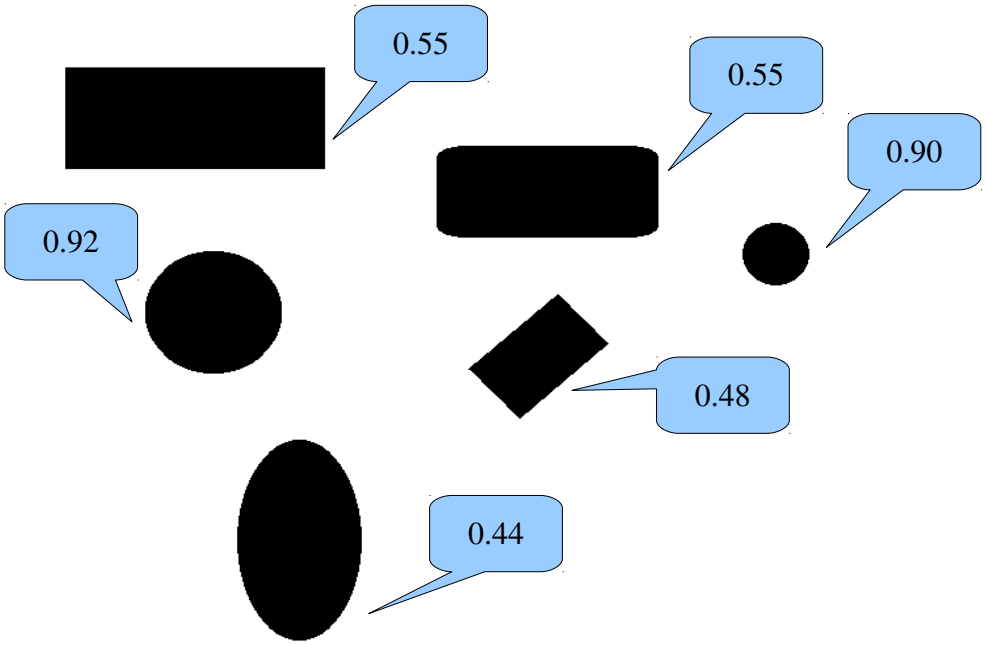
double Aire_Cercle2 = contourArea(Contours[i]);

Ceci nous permet de calculer l'aire de l'objet analysé avec la fonction « contourArea ».

Le rapport des aires est donc calculé comme ci-dessous :

double Circularite = Aire_Cercle2 / Aire_Cercle;

Voici les valeurs de circularité en appliquant l'algorithme aux formes suivantes :



Les formes non circulaires nous donne des valeurs de l'ordre de 50%. On peut sans contredit affirmer qu'elle ne sont pas circulaires. Les formes circulaires, quant à elles, obtiennent des valeurs de l'ordre de 90%. On pourrait ici utiliser d'autres techniques comme l'analyse du contour extérieure avec la fonction « convexHull » ou « isContourConvex ». Cette dernière permet de savoir si l'objet a un contour convexe ou non. Cette fonction jumelée au calcul de circularité pourrait aider davantages à conclure sur l'aspect définitif d'une circularité.