MARIN Nicolas Info4A RéVA

Analyse d’image : Compte Rendu TP3
Détection de contours

Table des matières

POLYTECH MARSEILLE
Aix*Marseille Université

I. Introduction

Analyse d'image

Détection de contours

L'objectif de ce TP est de comparer 3 détecteurs de gradient (Sobel, Laplace, Canny) avec une série d'images. Dans ces méthodes, il vous faudra fixer des seuils ou paramètres. Ces valeurs ne seront certainement pas adaptées à toutes vos images. L'ajout de « trackbar » à vos fenêtres vous ferra certainement gagner un peu de temps.

Vous devrez rédiger un rapport dans lequel vous présenterez une évaluation quantitative des détecteurs. Pour cela, il vous faudra comparer les images de contours obtenues à partir de votre programme avec des images de contours de référence (des images où les contours ont été tracés «à la main»). Vous trouverez ce type de données à l'adresse suivante : http://www.cs.rug.nl/~imaging/databases/contour_database/contour_database.html

Pour comparer deux images, vous calculerez 5 grandeurs :

- **contours_détectés** : nombre de pixels contours dans l'image calculée
- **contours_référence** : nombre de pixels contours dans l'image de référence
- **contours_corrects** : nombre de pixels contours correctement détectés dans l'image calculée selon l'image de référence (aussi égal à contours_détectés ∩ contours_référence)
- **faux_positifs** : nombre de pixels détectés comme contours, mais non contours dans l'image de référence (aussi égal à contours_détectés - contours_corrects)
- **faux_négatifs** : nombre de pixels contours non détectés dans l'image calculée, mais contour dans l'image de référence (aussi égal à contours_référence - contours_corrects)

Attention, pour le calcul de «contours_corrects», il est plus parfois plus judicieux de déterminer la présence d'un pixel dans l'image de référence en utilisant un voisinage (3x3 est suffisant) plutôt qu'uniquement sa position dans l'image dans l'image des contours.

A partir de ces 5 grandeurs, vous calculerez 3 mesures pour évaluer les détecteurs :

- **La performance** :
  P = contours_corrects / (contours_corrects + faux_positifs + faux_négatifs)
- **Le taux de faux positifs** :
  TFP = faux_positifs / (contours_corrects + faux_positifs + faux_négatifs)
- **Le taux de faux négatifs** :
  TFN = faux_ négatifs / (contours_corrects + faux_positifs + faux_négatifs)

Enfin, vous commenterez vos résultats (bonne détection, bonne localisation, «meilleure méthode», variation en fonction des paramètres d'entrée, méthode utilisée pour la comparaison [atout/faiblesse ...], ... etc …).

A RENDRE Un dossier **TP_contours** contenant :

- Votre code source qui permet de calculer les mesures présentées ci-dessus sur un lot d'image.
- Un rapport au format PDF contenant au moins les éléments suivants :
  - Présentation théorique des détecteurs utilisés
  - Explications des fonctions d'OpenCV utilisées. En particulier cette explication devra faire le lien entre la présentation des détecteurs et les paramètres des fonctions.
  - Tableau résumant les valeurs des mesures calculées.
  - Interprétation des résultats obtenus et comparaison des trois détecteurs.

L’objectif de ce TP est d’utiliser les différentes méthodes d’analyse d’image afin de déterminer les contours d’une image. On utilisera pour cela trois méthodes différentes : Sobel, Laplace et Canny. Ces 3 méthodes possèdent un fonctionnement différent et obtiendront des résultats différents selon les images. En les comparant avec des images sources, nous donnant les contours références, nous allons pouvoir analyser les différents résultats des algorithmes en fonction de valeurs comparatives.

II. Méthodes utilisées

1. Détection des contours

La détection de contours est un procédé de repérer les points d'une image matricielle qui correspondent à un changement brutal de luminosité. Ces changements indiquent la présence d'éléments appartenant à une structure, dans notre cas les contours d'une forme. Dans une image en niveau de gris, un changement brutal de la valeur caractérise un contour. Le but est de modifier l'image afin de ne laisser que paraître ces contours.

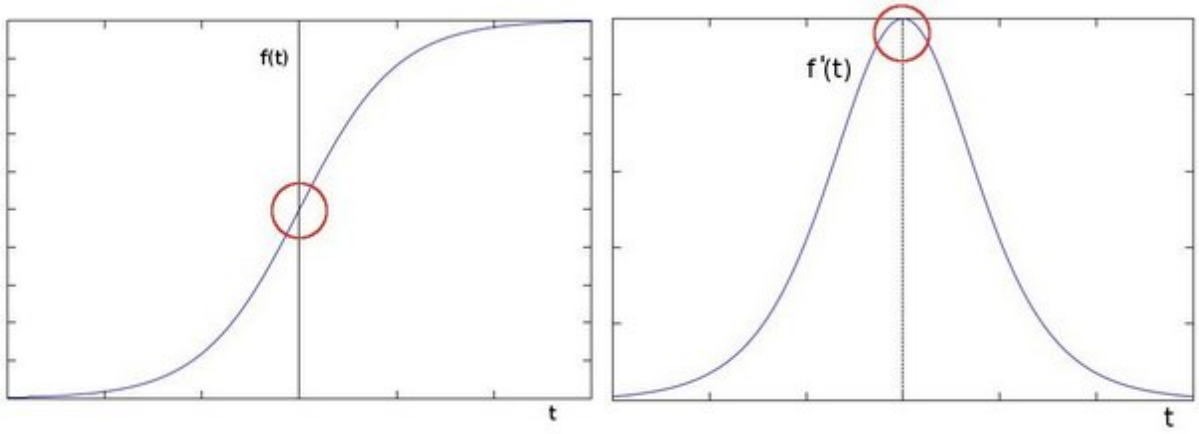
On utilisera alors 3 méthodes qui décrit précédemment Sobel, Laplace et Conny.

2. Sobel

Sobel est un des opérateurs les plus simples donnant des résultats corrects. Ca simplicité et sa rapidité sont les principales qualités de son utilisation. Mais l'utilisation d'approximation ainsi que sa méthode font que les résultats ne sont pas forcement très précis sur des images bruités (ex : ours).

On calcul le gradient de l'intensité de chaque pixel. Celui-ci indique la direction de la plus forte variation du clair au sombre ainsi que le taux de changement. Cela permet d'indiquer les points changeant de luminosité correspondant fréquemment à des bords.

Pour cela Sobel recherche les extremums de la dérivée première, lorsque celle-ci atteindra un maximum alors celui-ci marquera un changement d'intensité.



Soit $\boldsymbol{I}$ l'image.

Les changements Horizontaux sont calculés en faisant le produit de I avec la matrice Gx (taille impair) soit :

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

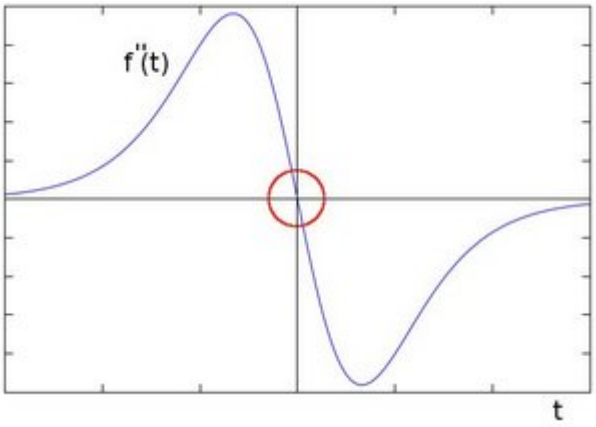
Les changements verticaux sont calculés en faisant le produit de I avec la matrice Gy (taille impair) soit :

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

On somme ensuite les résultats afin d'obtenir $G = |G_x| + |G_y|$

3. Laplace

Laplace est une méthode fonctionnement similairement à Sobel mais celle-ci utilise la dérivée seconde au lieu de la première. On ne cherchera plus alors un extremum mais un moment ou la dérivée seconde est nulle.



Similairement à Sobel Laplace possède les mêmes problèmes d'approximation et de sensibilité au bruit mais elles reste toujours un méthode simple au calcul et rapide.

4. Canny

Canny fonctionne similairement à Sobel mais cherche à réduire l'impact du bruit de celle-ci. Dans un premier temps on effectue des calculs similaires à Sobel avec les matriices Gx et Gy puis leurs combinaisons.

Canny va ensuite effectuer un seuillage des contours avec l’aide d’un seuil bas et haut. On compara alors le gradient de chaque pixel avec ces seuils. Si le gradient est inférieur au seuil bas, on rejette le pixel. Dans le cas supérieur au seuil haut, le pixel est accepté comme contour. Si le gradient est compris entre les deux seuils alors on l’accepte si il est en contact avec un autre pixel déjà accepté. Avec l’acceptation des tous ces pixels, on obtiendra comme résultat l’image binaire des contours. On obtient alors avec cette méthode de meilleurs résultats qu’avec Sobel dans la majorité des cas.

III. Application des méthodes

1. Sobel

Comme les Tps précédent, nous utiliserons les librairies OpenCV qui comprend déjà des méthodes pour calculer Sobel. Nous allons donc l'utiliser pour l'implémentation dans notre code.

```
// algorithme de sobel
Mat sobel_image(Mat image) {

    Mat grad, newimage;
    int ddepth = CV_16S;

    if(image.empty()) {
        return image;
    }
    cvtColor( image, newimage, COLOR_BGR2GRAY );

    Mat grad_x, grad_y;
    Mat abs_grad_x, abs_grad_y;

    Sobel( newimage, grad_x, ddepth, 1, 0, 3, scaleSobel, delta, BORDER_DEFAULT );
    Sobel( newimage, grad_y, ddepth, 0, 1, 3, scaleSobel, delta, BORDER_DEFAULT );

    convertScaleAbs( grad_x, abs_grad_x );
    convertScaleAbs( grad_y, abs_grad_y );

    addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad );

    grad = seuil_Img_Binaire(grad, seuilSobel);
    return grad;
}
```

Notre fonction prend en entrée l'image sur laquelle elle doit effectuer son filtrage. On effectue dessus deux fois la fonction de Sobel, la première fois pour obtenir le gradient horizontal Gx puis pour obtenir le gradient vertical Gy.
La fonction de Sobel prend comme entrée :

newimage : la matrice de l'image source

grad_y / grad_y : matrice des gradients de destination

ddepth : la profondeur de l'image. Dans notre cas nous avons initialisé celle-ci à CV_16S comme la documentation de la fonction utilisées

1, 0 / 0, 1: correspondent à l'ordre des dérivées en x, y. En fonction du gradient calculer on utilisera soit 1, 0 pour x ou 0, 1 pour y

3 : taille de la matrice utilisée par Sobel, nous allons utiliser comme dans la documentation une matrice 3x3

scaleSobel : facteur utilisée pour faire varier les valeurs des dérivées

delta : valeur ajoutée aux résultats permettant de choisir les niveaux de gris de l'image de sortie. On la placera à 255 pour obtenir une images à points noirs sur fond blanc.

BORDER_DEFAULT : méthode d'extrapolation des pixels, on utilisera celle par défaut.

Nous allons ensuite grâce à la fonction convertScaleAbs convertir les images en formats Unsigned 8bits. Une fois cela effectuer, nous allons utilliser la fonction addWeighted pour sommer les deux matrices. La fonction prendra en entrée

src1 : 1ère image source

alpha : coefficient de celle-ci

src2 : 2nd image source

beta : coefficient de celle-ci

gamma : un scalaire ajouter à chaque somme

dst : la matrice servant d'outpout

L'opération appliquée par celle-ci sere :

$$\mathtt{dst}(I) = \mathtt{saturate}(\mathtt{src1}(I) * \mathtt{alpha} + \mathtt{src2}(I) * \mathtt{beta} + \mathtt{gamma})$$

Nous obtenons alors une image sur laquelle Sobel est bien appliquée mais Sobel nous retourne alors une image en différent niveau de gris alors que nous souhaitons une image binaire. Nous utiliserons alors une fonction que nous créons transformant notre image en image binaire en fonction d'un seuil définit.

```
Mat seuil_Img_Binaire(Mat img, int seuil) {

    for (int i = 0; i < img.rows; i++) {
        for (int j = 0; j < img.cols; j++) {
            if (img.at<uchar>(i, j)<seuil) {
                img.at<uchar>(i, j)=0;
            } else {
                img.at<uchar>(i, j)=255;
            }
        }
    }
}
```

2. Laplace

Similairement à Sobel, il existe une fonction Laplace dans les librairies OpenCV, nous l'utiliserons aussi dans notre code que voici.

```
// algorithme de laplace
Mat laplace_image(Mat image) {

    Mat dst, newimage;
    int kernel_size = 3;
    int ddepth = CV_16S;

    if( image.empty() ) {
        return image;
    }

    cvtColor( image, newimage, COLOR_BGR2GRAY ); // Convert the image to grayscale
    Mat abs_dst;
    Laplacian( newimage, dst, ddepth, kernel_size, scaleLaplace, delta, BORDER_DEFAULT );

    convertScaleAbs( dst, abs_dst );
    abs_dst = seuil_Img_Binaire(abs_dst, seuilLaplace);

    return abs_dst;
}
```

Comme précédemment la fonction Laplace prendre plusieurs paramètres en entrée que nous expliciterons :

newimage : la matrice de l'image source
dst : la matrice ou le résultat sera stocké
ddepth : similaire au filtrage de Sobel cf : II.1
kernel_size : similaire au filtrage de Sobel cf : II.1
scaleLaplace : similaire au filtrage de Sobel cf : II.1
delta : similaire au filtrage de Sobel cf : II.1
BORDER_DEFAULT : similaire au filtrage de Sobel cf : II.1

Ce coup-ci il n'est pas nécessaire d'effectuer deux fois la fonction exécutant la méthode comme pour sobel. Nous n'avons plus qu'à la convertir en Unsigned 8 bits puis effectuer un seuillage pour la transformer en image binaire comme précédemment.

3. Canny

Canny est aussi une fonction dans OpenCV que nous utiliserons dans le code implémenté ci-dessous :

La fonction de Canny prend en entrée :

la matrice de l'image source
la matrice de l'image résultat

```
// algorithme de canny
Mat canny_image(Mat image) {

    Mat newimage, dst;

    int kernel_size = 3;

    cvtColor( image, newimage, COLOR_BGR2GRAY );
    Canny( newimage, newimage, lowThreshold, lowThreshold*ratioCanny, kernel_size );

    newimage = negative_image(newimage);

    return newimage;
}
```

seuil bas pour l'hystérésis de Canny
seuil multiplié par le ratio pour avoir le seuil haut de canny
kernel_size : taille de la matrice utilisée par Canny

Canny nous renvoie une image à points blancs sur fond noir donc effectue le négatif de cette image à l'aide d'une fonction que nous créons.

```
Mat negative_image(Mat img) {

    for (int i = 0; i < img.rows; i++) {
        for (int j = 0; j < img.cols; j++) {
            img.at<uchar>(i, j) = 255 - img.at<uchar>(i, j);
        }
    }

    return img;
}
```

IV. Résultats et calcul des grandeurs

1. Obtention des résultats

Comme écrit précédemment nous allons chercher à obtenir des valeurs permettant de comparer les 3 méthodes sur un panel d'image que nous allons parcourir à l'aide de donc de la performance, le taux de faux positifs et le taux de faux négatifs. Nous utiliserons pour ça plusieurs variables qui nous permettront de les calculer comme ci-dessous dans l'énoncé.

Pour comparer deux images, vous calculerez 5 grandeurs :

- **contours_détectés** : nombre de pixels contours dans l'image calculée
- **contours_référence** : nombre de pixels contours dans l'image de référence
- **contours_corrects** : nombre de pixels contours correctement détectés dans l'image calculée selon l'image de référence (aussi égal à contours_détectés ∩ contours_référence)
- **faux_positifs** : nombre de pixels détectés comme contours, mais non contours dans l'image de référence (aussi égal à contours_détectés - contours_corrects)
- **faux_négatifs** : nombre de pixels contours non détectés dans l'image calculée, mais contour dans l'image de référence (aussi égal à contours_référence - contours_corrects)

Attention, pour le calcul de «contours_corrects», il est plus parfois plus judicieux de déterminer la présence d'un pixel dans l'image de référence en utilisant un voisinage (3x3 est suffisant) plutôt qu'uniquement sa position dans l'image dans l'image des contours.

A partir de ces 5 grandeurs, vous calculerez 3 mesures pour évaluer les détecteurs :

- **La performance** :
  P = contours_corrects / (contours_corrects + faux_positifs + faux_négatifs)
- **Le taux de faux positifs** :
  TFP = faux_positifs / (contours_corrects + faux_positifs + faux_négatifs)
- **Le taux de faux négatifs** :
  TFN = faux_ négatifs / (contours_corrects + faux_positifs + faux_négatifs)

Nous allons alors dans un premier créer une fonction permettant d'obtenir le nombre de contours détectés. Ceci est très simple, on parcourt l'image pixel par pixel et on incrémente un compteur si la valeur d'un pixel est noir.

```
// détection des contours de l'image
int contours_detect(Mat image) {

    int res=0;
    for (int i = 0; i < image.rows; i++) {
        for (int j = 0; j < image.cols; j++) {
            if (image.at<uchar>(i, j)==0) {
                res++;
            }
        }
    }
    return res;
}
```

Cette méthode permettra d'obtenir les contours_détectés et les contours_références car il s'agit juste de trouver les pixels noirs mais sur l'image source ou celle de référence.

Une fois cette valeur obtenue nous allons chercher à obtenir la valeur de contours_corrects. Nous allons créer aussi une méthode pour cela comparant l'image obtenue par notre méthode de détection de contours et notre image de référence. Mais nous ne comparerons pas pixel par pixel mais au voisinage de chaque pixel. Nous analyserons les pixels sur une matrice 3*3 autour de celui-ci.

```
// détection des contours corrects de l'image
int contours_corrects(Mat imageRef, Mat image) {

    int res=0;
    bool haveRes=false;

    for (int i = 1; i < imageRef.rows-1; i++) {
        for (int j = 1; j < imageRef.cols-1; j++) {
            haveRes=false;
            if (imageRef.at<uchar>(i,j) == 0) {
                for(int k=-1; k<=1; k++) {
                    for(int l=-1; l<=1; l++) {
                        if(!haveRes && image.at<uchar>(i+k,j+l) == 0) {
                            res++;
                            haveRes=true;
                        }
                    }
                }
            }
        }
    }
    return res;
}
```

Une fois ces 3 valeurs obtenues nous pouvons facilement obtenir les autres pour effecteur les comparaisons entre les méthodes.

Pour effectuer cette comparaison entre les différentes méthodes, nous allons faire varier les différents arguments de chaque méthode afin de trouver la meilleur performance de chaque image pour chaque méthode. Cette comparaison sera très lourde en calcul et prendra un certain temps car il s'agit de comparer et de ré-effectuer un nombre important de fois les différentes méthodes.

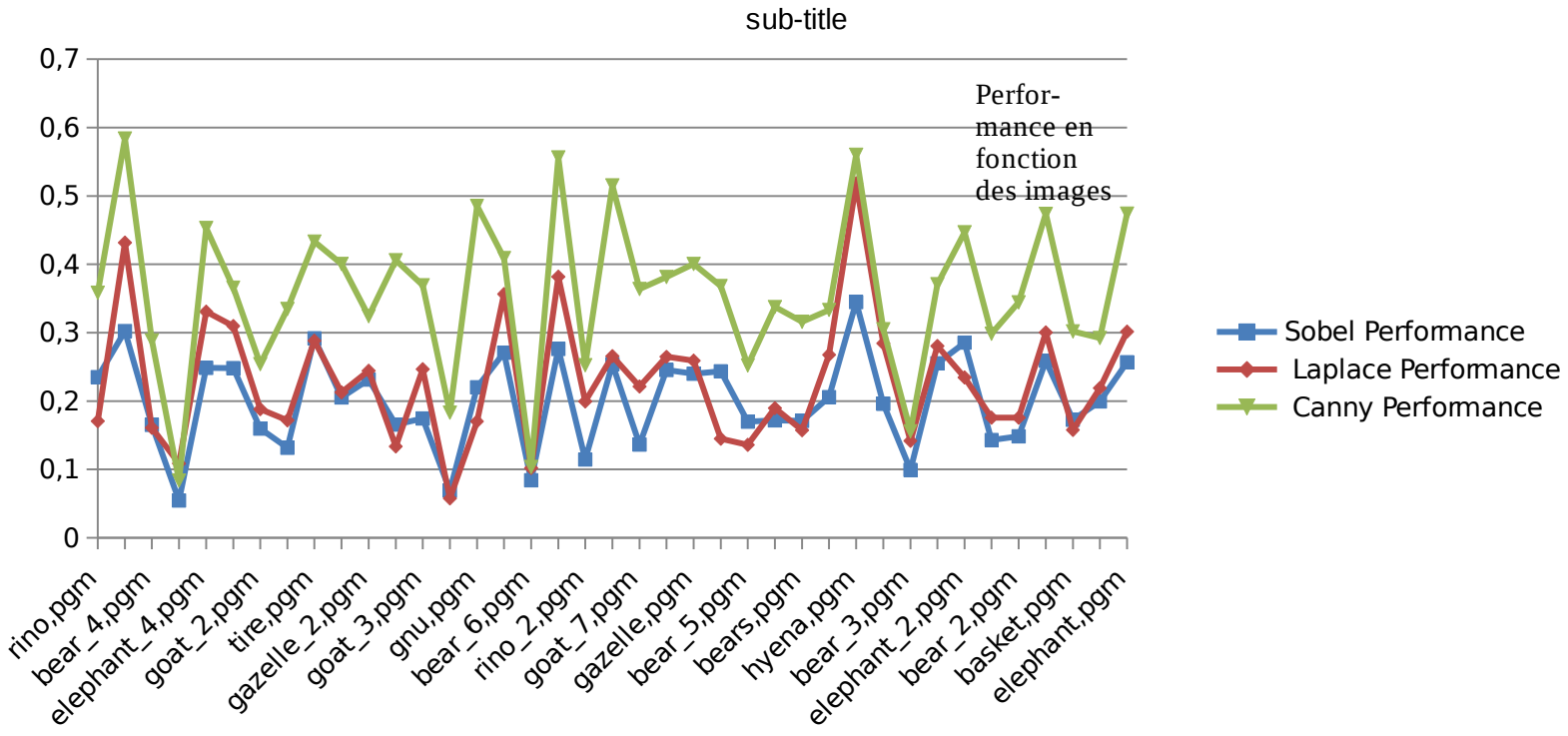
Voici un exemple d'implémentation d'une des méthodes de calcul de la meilleur performance pour Sobel. Les autres étant similaires pour les autres méthodes.

```
// calcul du meilleur sobel
void sobelOptim(Mat image,Mat imageRef, vector<float> &res) {

    int bestScale=0;
    int bestSeuil=0;
    float bestPerf=0;
    vector<float> tampon;

    for (int i=0; i<=scale_max; i++) {
        for (int j=0; j<=255; j++) {
            scaleSobel=i;
            seuilSobel=j;
            compare(sobel_image(image),imageRef,tampon);
            if(tampon[0]>bestPerf){
                bestPerf=tampon[0];
                bestScale=i;
                bestSeuil=j;
            }
        }
    }
//    cout<<"bestscale="<<bestScale<<endl;
//     cout<<"bestseuil="<<bestSeuil<<endl;
    scaleSobel=bestScale;
    seuilSobel=bestSeuil;
    compare(sobel_image(image),imageRef,res);
}
```

Ce calcul permet de stocker les différentes valeurs dans un tableau tampon dont les valeurs seront ensuite réécrite dans un fichier texte. Ce fichier permettra de ressortir les valeurs dans un tableur afin de pouvoir les visualiser plus facilement.

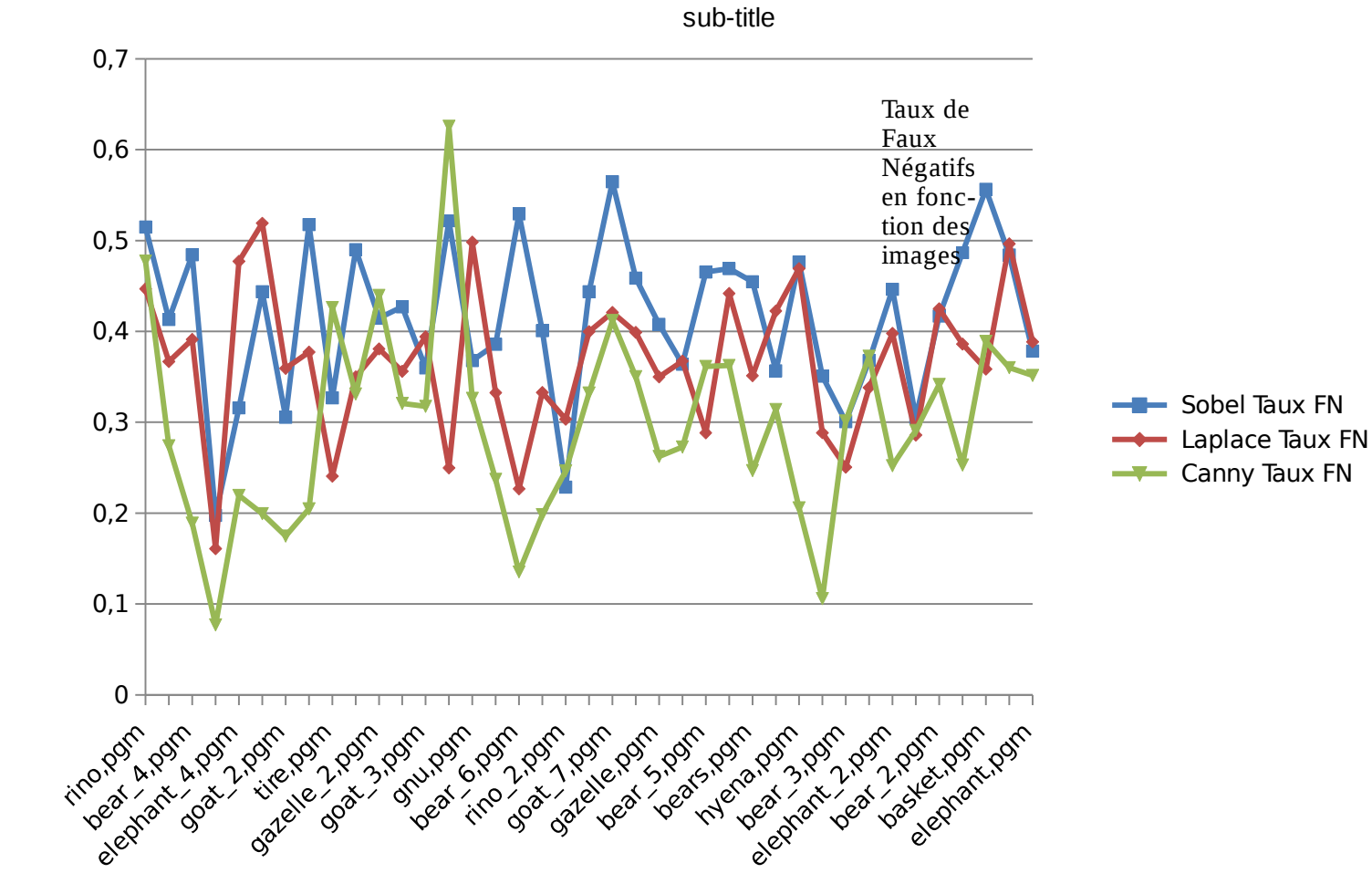
2. Analyse des résultats

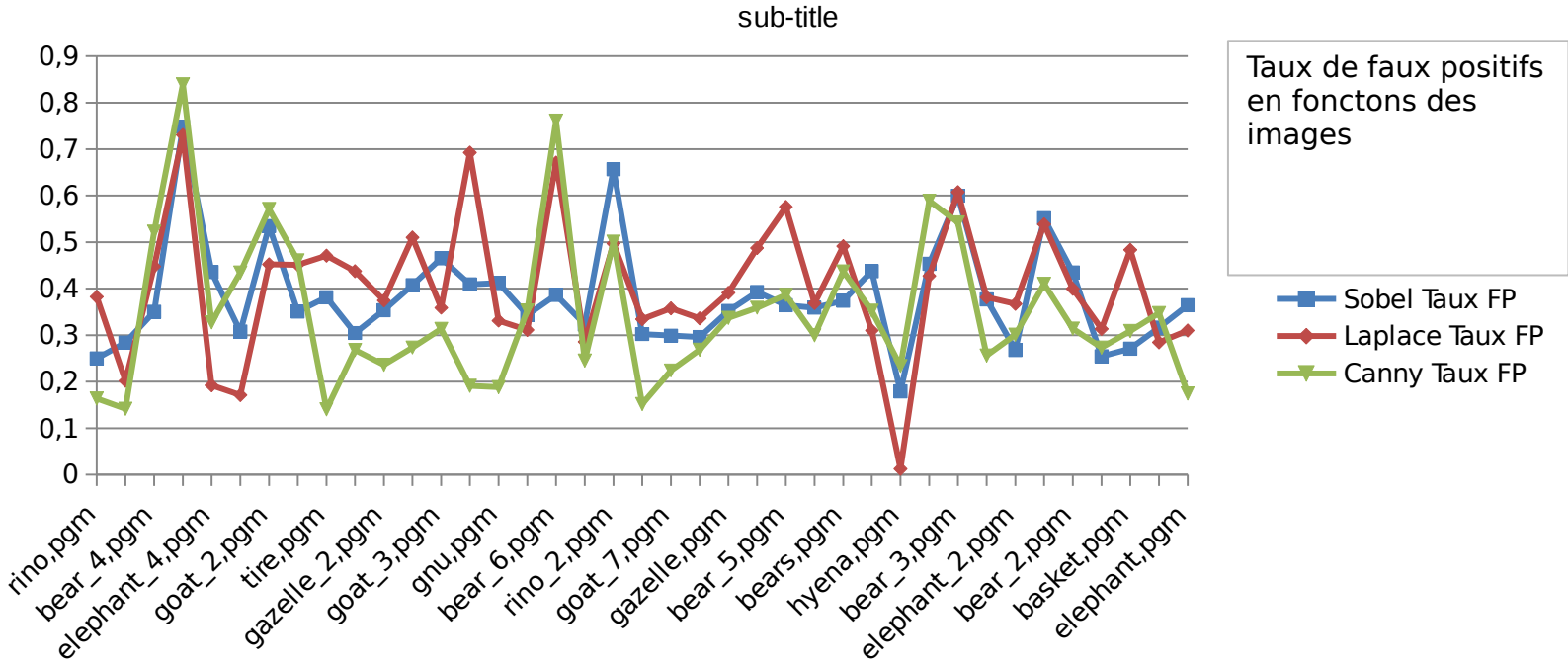
Suite aux calculs (approximativement 30min) et à la génération de tous les résultats sur notre tableur nous pouvons extraire les résultats suivant :



On observe que dans la majorité des cas Canny obtient les meilleurs résultats des 3 méthodes. Sobel semble être la moins efficace mais Laplace est à peine meilleur.
On obtient sur l'image golfcart.pgm notre meilleure performance avec Sobel pour un résultat de :
0,584098

On va maintenant analyser les différents taux obtenues :


sub-title
Taux de Faux Négatifs en fonc-tion des images
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
rino,pgm
bear_4,pgm
elephant_4,pgm
goat_2,pgm
tire,pgm
gazelle_2,pgm
goat_3,pgm
gnu,pgm
bear_6,pgm
rino_2,pgm
goat_7,pgm
gazelle,pgm
bear_5,pgm
bears,pgm
hyena,pgm
bear_3,pgm
elephant_2,pgm
bear_2,pgm
basket,pgm
elephant,pgm
Sobel Taux FN
Laplace Taux FN
Canny Taux FN


sub-title
Taux de faux positifs en fonctons des images
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
rino,pgm
bear_4,pgm
elephant_4,pgm
goat_2,pgm
tire,pgm
gazelle_2,pgm
goat_3,pgm
gnu,pgm
bear_6,pgm
rino_2,pgm
goat_7,pgm
gazelle,pgm
bear_5,pgm
bears,pgm
hyena,pgm
bear_3,pgm
elephant_2,pgm
bear_2,pgm
basket,pgm
elephant,pgm
Sobel Taux FP
Laplace Taux FP
Canny Taux FP

On observe que canny possède des taux plus bas que ses deux camarades dans l'ensemble. Les taux de Laplace et Sobel sont très variants. Cela est dut que ces 2 méthodes sont très sensibles aux bruits sur les images chargés tel que l'herbe ou les poils d'animaux.

Canny semble être le meilleur rapport puissance, temps et résultat. Il obtient les meilleurs résultats sur presque toutes les images. Il serait possible d'augmenter les valeurs de performance de Laplace ou Sobel en effectuant un Flou plus important sur les images afin de réduire l'impact du bruit sur celle-ci. Mais cela aura un impact fort sur le temps de calcul qui augmentera fortement. On notera que les résultats sont les plus performants sur des structures ne possédant que peu de bruit ce qui correspond souvent à des objets « humains ». Les méthodes ont du mal sur les objets noyés dans le décor.

Laplace semble être légèrement plus performant que Sobel sur les différentes images mais les résultats sont proches donc on peut difficilement statuer de la meilleur méthode entre les 2.

V. Conclusion

Que peut on donc retenir de ce TP sur la détection de contours ?

On retient que la méthode semblant le plus efficace rapport performance/rapidité est la méthode de Canny. Celle-ci obtient des résultats souvent corrects et n'est pas trop lourde coté calcul. On obtient aussi très rapidement la meilleure performance sans avoir besoin de faire varier beaucoup de paramètres.

Mais malgré ses performances honorables, Canny a du mal sur les images bruitées dont les objets ne sont pas forcément clairement identifiables et donc peut être encore améliorable. Les méthodes peuvent tous être améliorable probablement en jouant sur différentes valeurs que je n'ai pas utilisé durant ce TP tel que le flou ou la taille des matrices de méthodes. En cherchant ses valeurs on aurait pu obtenir de meilleurs résultat sur les 3 méthodes mais cela augmenterait grandement le temps de calcul pour chaque image. Nous nous situons actuellement dans un cas ou les résultats sont corrects et le temps de calcul lui aussi reste correcte.