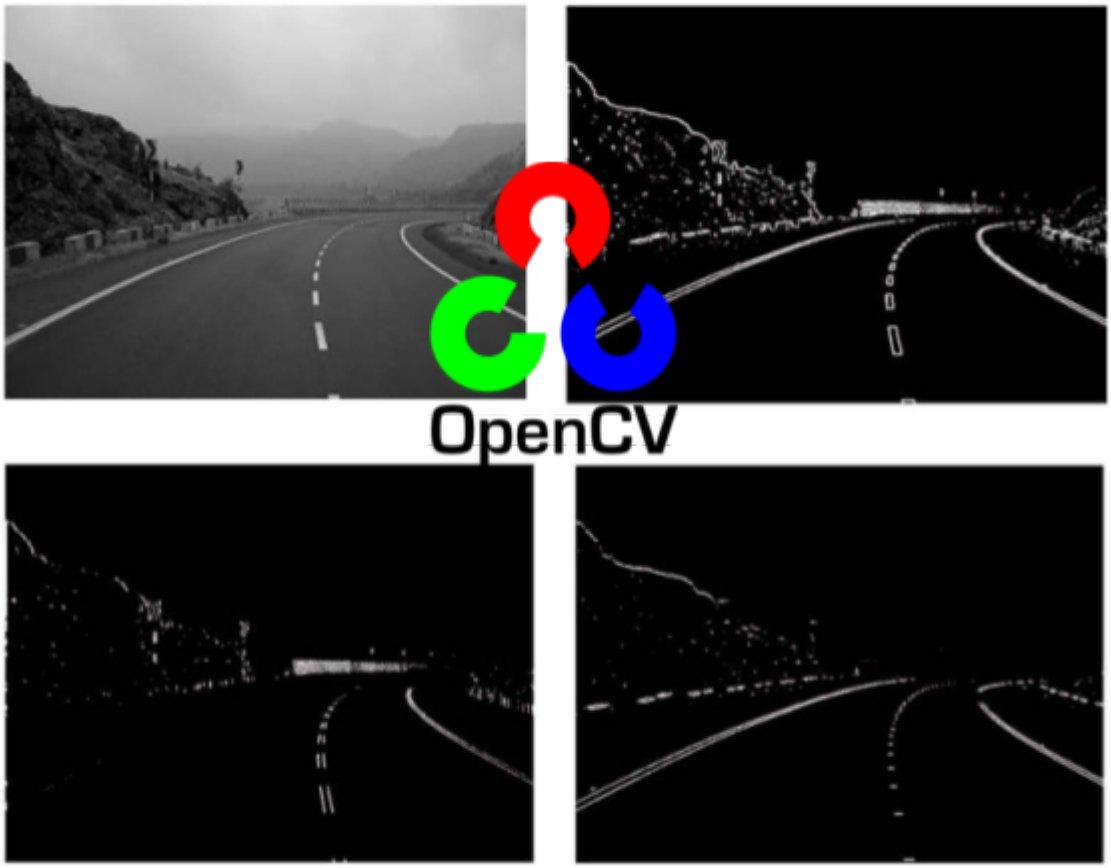
Vision et traitements d’images embarquées

Alex Didierlaurent & Oussama Jelliti E5FE



Introduction **2**
Filtre Sobel : 2
Présentation de la carte 2
Configuration de l'environnement pour le TP 2
Architecture de l'application 4

Comment ça marche ? **4**
Acquisition et format d'image 4
Pixel 4
Nombre de canaux 5
Format IplImage 5
Conversion en niveaux de gris 6
Filtre Médian 7
Filtre SOBEL 8

Développement & Optimisation **10**
Profiling 10
Automatisation 10
OpenCV 1 11
Version 1 11
Filtre Médian 11
Filtre Sobel 13
Résultat de la version 1 14
Version 2 15
Filtre Sobel 15
Filtre Médian 17
Temps d'exécution 18
Version 3 20
Résultat obtenu : 20
OpenCV 3 21
Modifications 21
Optimisations 22
Réduction de la résolution de l'image 22
Réduire les opération mathématique 23
Résultat pour une résolution de 432 x 240 24

Conclusion **26**

Introduction

L'objectif de ce TP est la mise en œuvre des techniques de traitement d'image vu durant le cours. Le but de ce mini projet est de réaliser un algorithme de détection de contours. Notre travaille utilisera la méthode SOBEL. Le traitement sera fait sur un flux d'image capturé avec une webcam. Enfin, durant la phase finale, nous allons essayer d'optimiser le temps d'exécution de notre programme.

Filtre 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.

Présentation de la carte

SABRE Lite :
Durant les séances de TP en présentiel, nous avons utilisé la carte SABRE Lite de chez element14. Parmi les caractéristiques de la carte on identifie qu'elle est basée sur :

- ARM Cortex A9 MPCore™ **4xCPU** Processor **@1GHz**
- RAM : 1GByte of 64-bit wide DDR3 **@ 532MHz**
- **Hardware accelerated Video Processing** Unit and Graphics Processing Unit
- **USB**, microSD, Ethernet, CAN interface

Configuration de l'environnement pour le TP

Notre configuration pour ce TP est basée sur une Raspberry PI 3. Elle dispose d'un processeur Broadcom BCM2837 64 bit à quatre cœurs ARM Cortex-A53 cadencé à 1,2

GHz. Ceci est conçu sur une architecture ARM de type RISC (Reduced instruction set computer).

Raspberry Pi 3 Model B	
Processor Chipset	Broadcom BCM2837 64Bit Quad Core Processor powered Single Board Computer running at 1.2GHz
Processor Speed	QUAD Core @1.2 GHz
RAM	1GB SDRAM @ 400 MHz
Storage	MicroSD
USB 2.0	4x USB Ports
Max Power Draw/voltage	2.5A @ 5V
GPIO	40 pin
Ethernet	Yes
WiFi	Built in
Bluetooth LE	Built in

On peut retrouver cette configuration en tappant la commande “lscpu” sur un terminal :

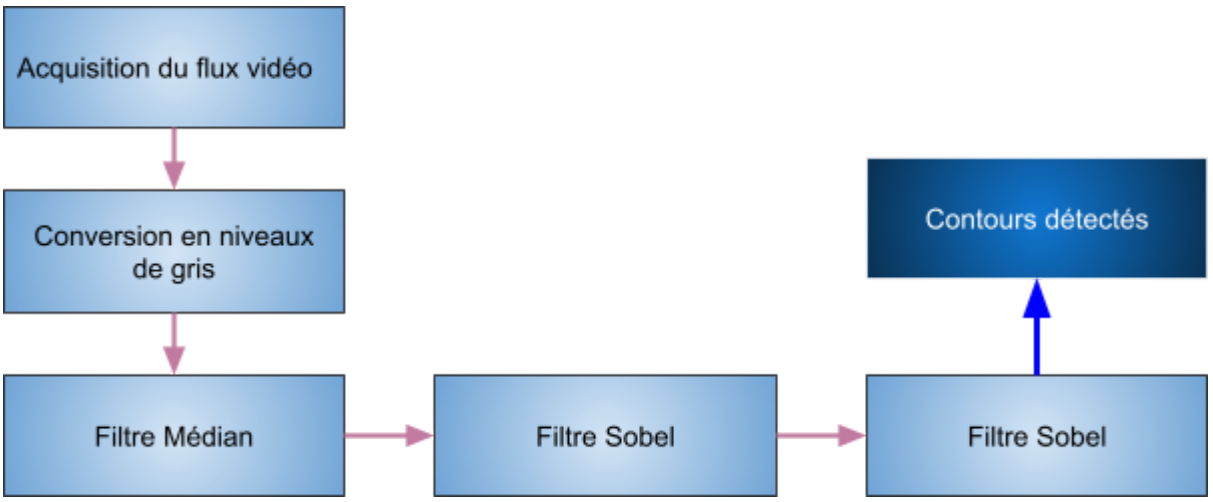
```
pi@raspberrypi:~/Desktop/imageprocess $ lscpu
Architecture:          armv7l
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list:   0-3
Thread(s) per core:    1
Core(s) per socket:    4
Socket(s):             1
Vendor ID:             ARM
Model:                 4
Model name:            Cortex-A53
Stepping:              r0p4
CPU max MHz:           1200.0000
CPU min MHz:           600.0000
```

Pour la réalisation de ce TP nous avons installé sur cette carte la bibliothèque graphique OpenCV. Ainsi, nous utiliserons une webcam comme entrée vidéo.

Architecture de l'application

Notre application effectue l'acquisition d'un flux vidéo venant de la webcam. Cette dernière sera capturée sous format d'une image RGB pour être envoyée vers un premier traitement pour la transformer en niveaux de gris. La sortie est envoyée sur un premier filtre appelé Media, l'objectif étant d'éliminer le bruit. Finalement nous effectuons notre traitement avec le filtre Sobel qui nous donnera une image noir et blanc avec les contours.

Le schéma ci-dessous décrit le parcours de l'application :



Comment ça marche ?

Acquisition et format d'image

La bibliothèque OpenCV nous permet de faire la capture d'un flux vidéo. La méthode utiliser dans un premier temps est :

```
capture = cvCreateCameraCapture(video_input);
```

Pour faire l'acquisition et récupérer l'image nous utiliserons la méthode suivant :

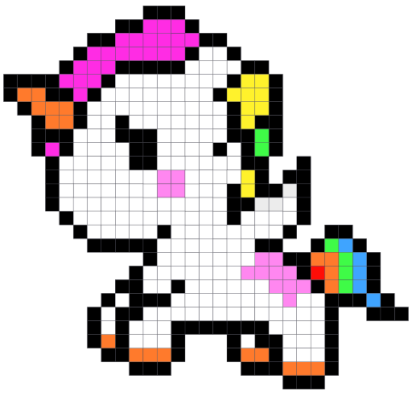
```
Image_IN = cvQueryFrame(capture);
```

Cette méthode renvoie une image sous le format IplImage.

Pixel

Voyons d'abord comment IplImage comprend les images sur notre ordinateur.
Dans un premier temps, les images sont définies par des carrés. Ces carrés délimités s'appellent des pixels.

En agrandissant une image, nous pouvons les voir.



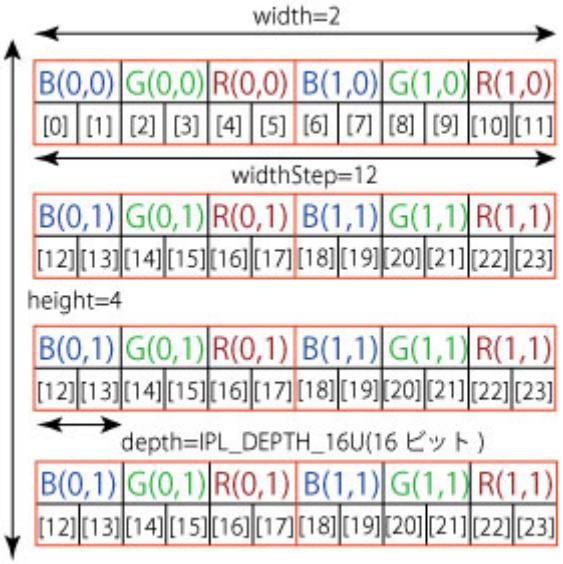
Le nombre de pixels dans la direction horizontale de l'image est la largeur. Le nombre de pixels dans la direction verticale est la hauteur. Le produit des deux définit la résolution d'une image.
Dans IplImage, la largeur est représentée par la largeur et la hauteur par la profondeur.

Nombre de canaux

Ce nombre de couleurs est appelé le nombre de canaux. Dans IplImage, il est spécifié par nChannels. Si c'est une image RGB le nombre de canaux sera de 3. S'il s'agit d'une échelle de gris, le nombre de canaux sera alors de 1.

Format IplImage

Ceci était très important à comprendre avant de passer à la programmation.
Voici le format IplImage :



Nous allons utiliser la profondeur IPL_DEPTH_8U: entier 8 bits non signé.

```
Image_GRAY =
      cvCreateImage(cvSize(Image_IN->width, Image_IN->height),
IPL_DEPTH_8U, 1);
```

Conversion en niveaux de gris

Pour trouver le niveau de gris, nous allons utiliser le modèle colorimétrique à 3 composantes:

$$\begin{bmatrix} Y' \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.595716 & -0.274453 & -0.321263 \\ 0.211456 & -0.522591 & 0.311135 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

- Y ' : la luminance
- I : la composante en phase
- Q : la composante en quadrature de la chrominance.

Le niveau de gris est obtenu en faisant la multiplication entre la luminance et chaque pixels de l'image.

Exemple :

```
  int i, j;  // indices des boucles
  for (i = 0; i < height; i++)
    for (j = 0; j < width; j++)
      Data_out[i * step_gray + j] =
          0.114 * Data_in[i * step + j * channels + 0] +
          0.587 * Data_in[i * step + j * channels + 1] +
          0.299 * Data_in[i * step + j * channels + 2];
```

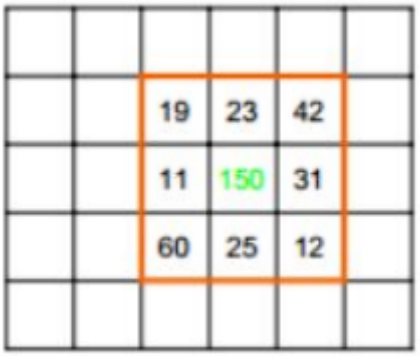
On peut observer le traitement de l'image en sortie :

Image en entrée Image niveaux de gris

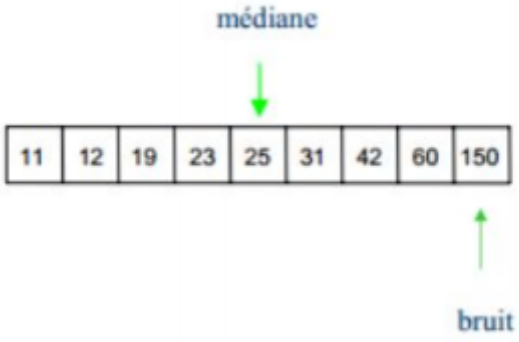
Filtre Médian

Ce filtrage non linéaire sera utilisé pour la réduction de bruit. Ce filtre est beaucoup utilisé dans les applications de traitement d'image car elle prépare le terrain pour un prochain traitement comme la détection de contours. Le principe de ce filtre est de remplacer un pixel par la moyenne des pixels autour.

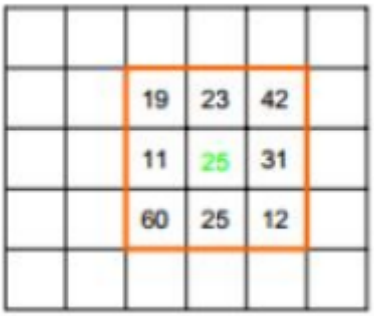
Exemple pour un cas de neuf pixel :



L'algorithme cherchera à déterminer la valeur médiane de cette matrice puis remplacera le pixel par cette dernière. Ici, on identifie une médiane de 25 :



La valeur aberrante sera donc remplacée par la médiane. Le résultat est le suivant :



Notre algorithme se décompose donc en deux partie :

- Trie et recherche de médiane de chaque pixel (différent algorithme de tri)
- Remplacer le pixel par la médiane

Le résultat n'est pas évident à l'oeil nu, surtout dans un environnement lumineu :

Grey

Médian

Filtre SOBEL

L'opérateur Sobel effectue une mesure de gradient spatial 2D sur une image et met ainsi l'accent sur les régions de haute fréquence spatiale qui correspondent aux bords. En règle générale, il est utilisé pour trouver l'amplitude absolue du gradient approximatif à chaque point d'une image en niveaux de gris d'entrée.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

Exemple :

```
  int G_x[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};//Matrice Vert.
  int G_y[3][3] = {{-1, -2, -1}, {0, 0, 0}, {1, 2, 1}};//Matrice Horiz.

  for (i = 1; i < height - 1; i++) {
    for (j = 1; j < width - 1; j++) {
      for (a = 0; a < 3; a++) {
        for (b = 0; b < 3; b++) {
          Gx_sum += Data_Median_out[(i - 1 + a) * step_gray + b + j -
1] * G_x[a][b];

          Gy_sum += Data_Median_out[(i - 1 + a) * step_gray + b + j -
1] * G_y[a][b];
        }
      }
      G = sqrt(abs(Gx_sum) * abs(Gx_sum) + abs(Gy_sum) * abs(Gy_sum));

      // Seuils White/Black
      G = G > 20 ? 255 : 0;
      G = G < 19 ? 0 : G;

      Sobel_out[i * step_gray + j] = (uchar)G;

      Gx_sum = 0;
      Gy_sum = 0;
    }
  }
```

Voici le résultat du traitement :

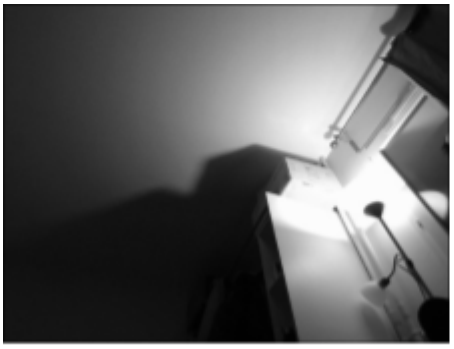


Image d'entrée

Image en sortie

Développement & Optimisation

Dans cette partie, nous allons rentrer dans le détail du programme, expliquer les choix algorithmique ainsi que les optimisation afin de réduire le temps d'exécution. Pour des raisons de compatibilité (et de persévérance), l'application a été réécrite sur la version OpenCV 3. La suite du rapport contiendra deux grandes parties : OpenCV 1 et OpenCV 3.

Profiling

Pour optimiser notre application, nous aurons besoin de statistiques sur le temps d'exécution de chaque application. Il existe plusieurs outils mais nous avons choisi Gprof. Afin d'avoir des résultats similaires et cohérents, nous limiterons le nombre de frames à 20.

Automatisation

Un fichier bash a été créé pour la compilation -> exécution -> profiling. Il s'occupera d'enregistrer chaque profil avec un nom correspondant à la version exécutée. La variable **video_input** permet de sélectionner l'entrée vidéo. La variable **res** permet de sélectionner la résolution de l'image capturée (uniquement pour OpenCV 3).

```
#!/bin/bash
echo "Compiling code with OpenCV1 ... "
g++ -pg `pkg-config opencv --cflags` Sabre_opencv1.cpp -o
Sabre_opencv1 `pkg-config opencv --libs`
video_input=0
count=3
for i in $(seq $count); do
    echo "Running the Application v$i"
    ./Sabre_opencv1 $video_input $i

    echo "Profiling ..."
    hash=$(date +%s)
    gprof Sabre_opencv1 > prof/resultat_profiling_${hash}_${i}.txt
    echo "Profiling file ->
prof/resultat_profiling_opencv1_${hash}_${i}.txt"
    sleep 3
done



echo "Compiling code with OpenCV3 ... "
g++ -pg `pkg-config opencv --cflags` Sabre_opencv3.cpp -o
Sabre_opencv3  `pkg-config opencv --libs`
```

```
echo "Running the Application"
res = 5
./Sabre_opencv3 0 $res

echo "Profiling ..."
hash=$(date +%s)
gprof Sabre_opencv3 > prof/resultat_profiling_$hash.txt

echo "Profiling file -> prof/resultat_profiling_$hash.txt"
```

Ce petit script shell nous a fait gagner énormément de temps. Surtout lorsque l'on a commencé à avoir plusieurs versions à gérer.

OpenCV 1

Le code est organisé sous des versions différentes. À chaque optimisation significative nous augmentons l'index de la version.

Version 1

Pour cette version, nous avons eu comme objectif de faire fonctionner l'application. Donc nous n'avons pas cherché à optimiser l'algorithme.

Les 3 fonctions principales sont :

```
      grey_filter();
      median_filter_v1();
      filtre_sobel_v1();
```

Le filtre gris a été proposé par le professeur et expliqué auparavant. Nous allons détailler les deux autres fonctions.

Filtre Médian

Dans un premier temps, nous allons parcourir pixel par pixel, à chaque fois nous allons chercher ces pixel voisins. Toutes ces valeurs sont groupées dans un tableau qui sera trié par la suite. Ce dernier à une taille de 9 dont sa valeur Médian se trouvera à l'index 4.

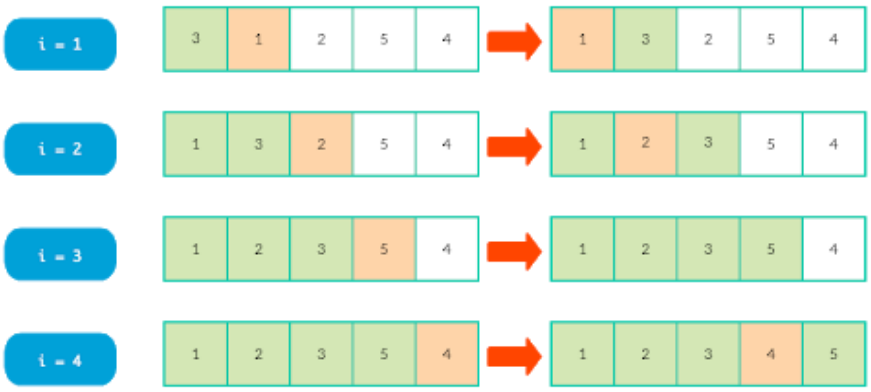
```
void median_filter_v1() {
  int a, b, k, i, j;  // indices des boucles

  int data_array[9];

  for (i = 1; i < height - 1; i++)
    for (j = 1; j < width - 1; j++) {
      k = 0;
```

```
      for (a = 0; a < 3; a++) {
        for (b = 0; b < 3; b++) {
          data_array[k] = Data_out[(i - 1 + a) * step_gray + b + j -
1];
          k++;
        }
      }
      // RUN SORT ALGO BY INSERTION
      insertionSort(data_array);
      // REPLACE BY MEDEAN
      Data_Median_out[i * step_gray + j] = data_array[4];
    }
}
```

Pour la recherche de la valeur Médian, nous avons utilisé un algorithme de tri par insertion. Voici le code :

```
void insertionSort(int array[]) {
  int temp, i, j;
  for (i = 0; i < 9; i++) {
    temp = array[i];
    for (j = i - 1; j >= 0 && temp < array[j]; j--) {
      array[j + 1] = array[j];
    }
    array[j + 1] = temp;
  }
}
```

Son principe est simple et efficace. Comme on peut le voir sur l'image ci-dessous :



Dans le cas d'un tri croissant, il va chercher la valeur la plus petite puis il va permuter les index.

Filtre Sobel

Le principe est le même, nous allons parcourir tous les pixels. Pour chaque pixel nous allons récupérer la matrice des pixels autour puis appliquer le produit de convolution par les matrices de Sobel.

La complexité ici était de récupérer les valeurs autour car elles sont rangées sous un format de vecteur où l'espacement entre deux pixel est définie par la variable step_grey. Elle est égale le rapport entre la largeur du pas de l'image niveau de gris et la taille d'un uchar :

```
step_gray = Image_GRAY->widthStep / sizeof(uchar);
```

Voici l'exemple :

```
void filtre_sobel_v1() {
  int Gx_sum, Gy_sum, G;  // Sobel var
  int i, j, a, b;  // indices des boucles

  for (i = 1; i < height - 1; i++) {
    for (j = 1; j < width - 1; j++) {
      for (a = 0; a < 3; a++) {
        for (b = 0; b < 3; b++) {
          Gx_sum += Data_Median_out[(i - 1 + a) * step_gray + b + j -
1] * G_x[a][b];

          Gy_sum += Data_Median_out[(i - 1 + a) * step_gray + b + j -
1] * G_y[a][b];
        }
      }
      G = sqrt(abs(Gx_sum) * abs(Gx_sum) + abs(Gy_sum) * abs(Gy_sum));

      //  White/Black treshold
      G = G > 20 ? 255 : 0;
      G = G < 19 ? 0 : G;

      Sobel_out[i * step_gray + j] = (uchar)G;

      Gx_sum = 0;
      Gy_sum = 0;
    }
  }
}
```

Nous récupérons une approximation de la norme du gradient qui sera la nouvelle valeur du pixel. Sur ce derniér nous appliquerons des seuil pour définir le niveaux de noir & blanc.

Résultat de la version 1

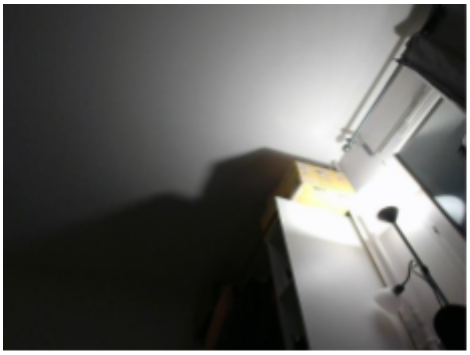
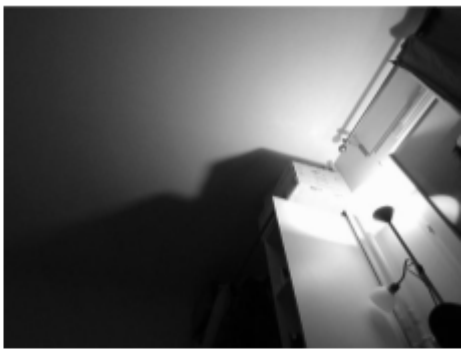
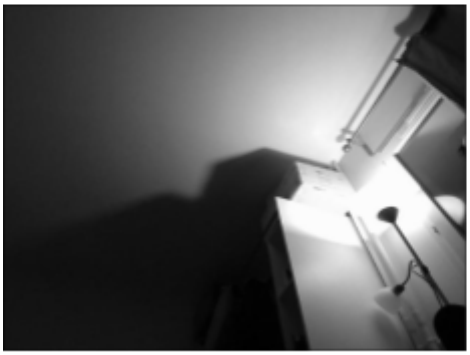


Image d'entrée



Niveaux de gris



Médian

Sortie du filtre So

Nous observons un résultat respectable en termes de rendu.

Nous allons maintenant nous intéresser au temps de traitement. L'outils Gprof nous renvois les résultats suivant :

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 38.35      5.04     5.04       20   252.13   252.13  filtre_sobel_v1()
 38.12     10.05     5.01  6099280     0.00     0.00  insertionSort(int*)
 18.03     12.43     2.37       20   118.56   369.18  median_filter_v1()
  5.55     13.16     0.73       20    36.52    36.52  grey_filter()
  0.00     13.16     0.00        4     0.00     0.00  file_name(char*)
  0.00     13.16     0.00        3     0.00     0.00  cvSize(int, int)
  0.00     13.16     0.00        3     0.00     0.00  CvSize::CvSize(int, int)
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.08% of 12.82 seconds

index % time    self  children    called     name
                                              <spontaneous>
[1]    100.0    0.00   12.82                 main [1]
                2.55    4.55      20/20          median_filter_v1() [2]
                4.98    0.00      20/20          filtre_sobel_v1() [3]
```

```
                0.74    0.00      20/20          grey_filter() [5]
                0.00    0.00       4/4           file_name(char*) [13]
                0.00    0.00       3/3           cvSize(int, int) [14]
-----------------------------------------------
                2.55    4.55      20/20          main [1]
[2]     55.4    2.55    4.55      20             median_filter_v1() [2]
                4.55    0.00 6099280/6099280     insertionSort(int*) [4]
-----------------------------------------------
                4.98    0.00      20/20          main [1]
[3]     38.9    4.98    0.00      20             filtre_sobel_v1() [3]
-----------------------------------------------
                4.55    0.00 6099280/6099280     median_filter_v1() [2]
[4]     35.5    4.55    0.00 6099280             insertionSort(int*) [4]
-----------------------------------------------
                0.74    0.00      20/20          main [1]
[5]      5.7    0.74    0.00      20             grey_filter() [5]
-----------------------------------------------
                0.00    0.00       4/4           main [1]
[13]     0.0    0.00    0.00       4             file_name(char*) [13]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[14]     0.0    0.00    0.00       3             cvSize(int, int) [14]
                0.00    0.00       3/3           CvSize::CvSize(int, int) [15]
-----------------------------------------------
                0.00    0.00       3/3           cvSize(int, int) [14]
[15]     0.0    0.00    0.00       3             CvSize::CvSize(int, int) [15]
-----------------------------------------------
```

Ce premier retour nous montre que la fonction filtre_sobel_v1 prend un temps significatif durant l'exécution. Elle est suivie par la fonction insertionSort().

Nous nous focalisons sur l'optimisation de ces deux fonctions pour but de réduire le temps d'exécution sans trop impacter la qualité du résultat.

Version 2

Filtre Sobel

Nous avons relevé deux potentielles optimisations :

- La première concerne l'utilisation des boucles **for**. Nous allons donc saisir manuellement les indexes des valeurs autour du pixel.
- La deuxième est la multiplication par 0 durant le produit avec les deux matrices de Sobel.

Pour ce faire, nous avons enlevé les lignes non essentielles.
Le code devient le suivant :

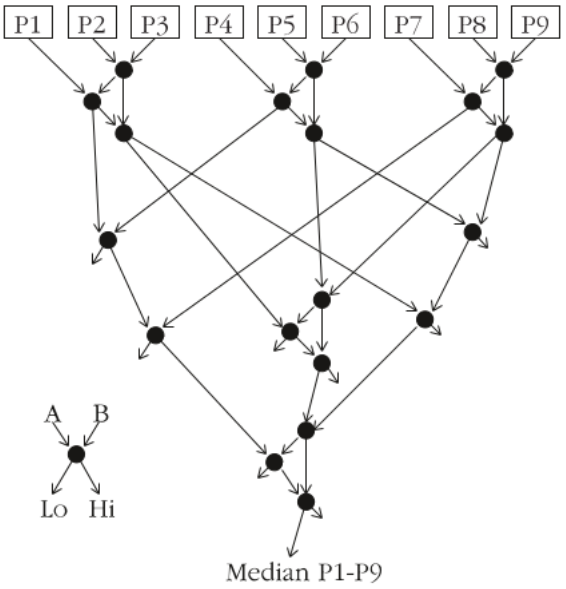
```
void filtre_sobel_v2() {
  int Gx_sum, Gy_sum, G;  // Sobel var
  int i, j, a, b;  // indices des boucles
  for (i = 1; i < height - 1; i++) {
    for (j = 1; j < width - 1; j++) {
      // Sobel Vertical Gy
      // L0
      Gy_sum += Data_Median_out[(i * step_gray) + j] * G_y[0][0];
      Gy_sum += Data_Median_out[(i * step_gray) + 1 + j] * G_y[0][1];
      Gy_sum += Data_Median_out[(i * step_gray) + 2 + j] * G_y[0][2];
      // L1
      Gy_sum += Data_Median_out[(i * step_gray + 2) + j] * G_y[2][0];
      Gy_sum += Data_Median_out[(i * step_gray + 2) + 1 + j] *
G_y[2][1];
      Gy_sum += Data_Median_out[(i * step_gray + 2) + 2 + j] *
G_y[2][2];

      // Sobel Horizental Gx
      Gx_sum += Data_Median_out[(i * step_gray) + 0 + j] * G_x[0][0];
      Gx_sum += Data_Median_out[(i * step_gray) + 2 + j] * G_x[0][2];

      Gx_sum += Data_Median_out[(i * step_gray + 1) + j] * G_x[1][0];
      Gx_sum += Data_Median_out[(i * step_gray + 1) + 2 + j] *
G_x[1][2];

      Gx_sum += Data_Median_out[(i * step_gray + 2) + j] * G_x[2][0];
      Gx_sum += Data_Median_out[(i * step_gray + 2) + 2 + j] *
G_x[2][2];

      G = sqrt(abs(Gx_sum) * abs(Gx_sum) + abs(Gy_sum) * abs(Gy_sum));

      // Seuils White/Black
      G = G > 128 ? 255 : 0;
      G = G < 0 ? 0 : G;

      Sobel_out[i * step_gray + j] = (uchar)G;

      Gx_sum = 0;
      Gy_sum = 0;
    }
```

```
  }
}
```

Filtre Médian

Nous avons opté pour changer la façon avec laquelle nous cherchons la valeur médiane. Après des recherches sur internet, nous avons trouvé un article par Torben Mogensen, expliquant une méthode plus rapide pour trouver cette valeur. Elle est valable pour des matrices 3x3,5x5,7x7,9x9 et 25x25. Le nombre d'itération est de O(log(n)).

Cet algorithme effectue une comparaison entre n et n+1 valeur et effectue une permutation si le type de tri n'est pas respecté (ascendant ou descendant). Voici un arbre décrivant la description de l'algorithme :



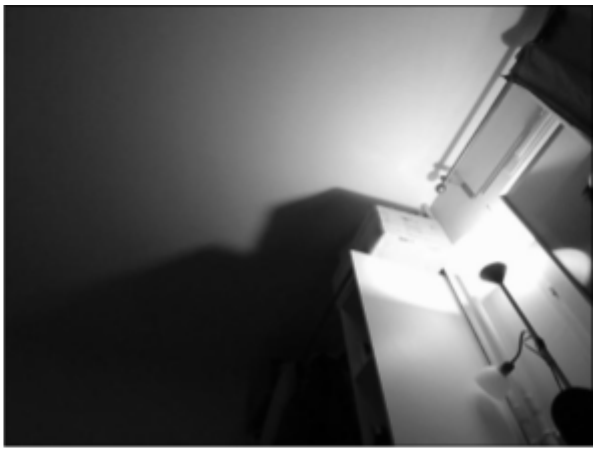
On met à jour notre code :

```
void median_filter_v2() {
  int a, b, k, i, j;  // indices des boucles

  int data_array[9];

  for (i = 1; i < height - 1; i++)
    for (j = 1; j < width - 1; j++) {
      k = 0;
      for (a = 0; a < 3; a++) {
        for (b = 0; b < 3; b++) {
          data_array[k++] = Data_out[(i - 1 + a) * step_gray + b + j -
1];
        }
      }
      median_opti9(data_array);
```

```
        // Get Median value
        Data_Median_out[i * step_gray + j] = data_array[4];
    }
}
```

On observe les résultats suivants en sorties :

Image en entrée



Après niveaux de gris

Après filtre Médiane

Filtre Sobel

Temps d'exécution

L'outils d'analyse Gprof nous renvoit les temps suivants:

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
31.05	2.30	2.30	20	115.05	115.05	filtre_sobel_v2()
29.97	4.52	2.22	20	111.05	219.60	median_filter_v2()
29.30	6.69	2.17	6099280	0.00	0.00	median_opti9(int*)
9.59	7.40	0.71	20	35.52	35.52	grey_filter()

```
  0.14      7.41     0.01        3     3.33     3.33  cvSize(int, int)
  0.00      7.41     0.00        4     0.00     0.00  file_name(char*)
  0.00      7.41     0.00        3     0.00     0.00  CvSize::CvSize(int, int)


granularity: each sample hit covers 2 byte(s) for 0.13% of 7.41 seconds

index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    7.41                 main [1]
                2.22    2.17      20/20          median_filter_v2() [2]
                2.30    0.00      20/20          filtre_sobel_v2() [3]
                0.71    0.00      20/20          grey_filter() [5]
                0.01    0.00       3/3           cvSize(int, int) [6]
                0.00    0.00       4/4           file_name(char*) [14]
-----------------------------------------------
                2.22    2.17      20/20          main [1]
[2]     59.2    2.22    2.17       20           median_filter_v2() [2]
                2.17    0.00 6099280/6099280     median_opti9(int*) [4]
-----------------------------------------------
                2.30    0.00      20/20          main [1]
[3]     31.0    2.30    0.00       20           filtre_sobel_v2() [3]
-----------------------------------------------
                2.17    0.00 6099280/6099280     median_filter_v2() [2]
[4]     29.3    2.17    0.00 6099280           median_opti9(int*) [4]
-----------------------------------------------
                0.71    0.00      20/20          main [1]
[5]      9.6    0.71    0.00       20           grey_filter() [5]
-----------------------------------------------
                0.01    0.00       3/3           main [1]
[6]      0.1    0.01    0.00       3           cvSize(int, int) [6]
                0.00    0.00       3/3           CvSize::CvSize(int, int) [15]
-----------------------------------------------
                0.00    0.00       4/4           main [1]
[14]     0.0    0.00    0.00       4           file_name(char*) [14]
-----------------------------------------------
                0.00    0.00       3/3           cvSize(int, int) [6]
[15]     0.0    0.00    0.00       3           CvSize::CvSize(int, int) [15]
-----------------------------------------------
```

Nous remarquons une baisse de temps d'exécution de la fonction filtre_sobel() par un facteur de 2 ce qui est significatif. Malgré ça, il reste quelques optimisations à effectuer comme le parallélisme des calculs et la réduction des opérateurs mathématiques.

Version 3

Dans cette troisième version, nous allons essayer d'implémenter directement l'algorithme de tri afin de réduire le temps d'exécution. Donc le contenu de la fonction median_opti9 sera implémenté directement dans la fonction médiane median_filter(). Cela évitera les saut d'instructions aussi appelé l'effet ping-pong JUMP.

Résultat obtenu :

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 46.45      2.62     2.62       20   131.00   131.00  median_filter_v3()
 40.78      4.92     2.30       20   115.00   115.00  filtre_sobel_v2()
 12.77      5.64     0.72       20    36.00    36.00  grey_filter()
  0.00      5.64     0.00        4     0.00     0.00  file_name(char*)
  0.00      5.64     0.00        3     0.00     0.00  cvSize(int, int)
  0.00      5.64     0.00        3     0.00     0.00  CvSize::CvSize(int, int)


granularity: each sample hit covers 4 byte(s) for 0.18% of 5.64 seconds

index % time    self  children    called     name
                                             <spontaneous>
[1]    100.0    0.00    5.64                 main [1]
                2.62    0.00      20/20        median_filter_v3() [2]
                2.30    0.00      20/20        filtre_sobel_v2() [3]
                0.72    0.00      20/20        grey_filter() [4]
                0.00    0.00       4/4         file_name(char*) [12]
                0.00    0.00       3/3         cvSize(int, int) [13]
-----------------------------------------------
                2.62    0.00      20/20        main [1]
[2]     46.5    2.62    0.00      20         median_filter_v3() [2]
-----------------------------------------------
                2.30    0.00      20/20        main [1]
[3]     40.8    2.30    0.00      20         filtre_sobel_v2() [3]
-----------------------------------------------
                0.72    0.00      20/20        main [1]
[4]     12.8    0.72    0.00      20         grey_filter() [4]
-----------------------------------------------
                0.00    0.00       4/4         main [1]
[12]     0.0    0.00    0.00       4         file_name(char*) [12]
-----------------------------------------------
                0.00    0.00       3/3         main [1]
[13]     0.0    0.00    0.00       3         cvSize(int, int) [13]
                0.00    0.00       3/3         CvSize::CvSize(int, int) [14]
-----------------------------------------------
                0.00    0.00       3/3         cvSize(int, int) [13]
[14]     0.0    0.00    0.00       3         CvSize::CvSize(int, int) [14]
-----------------------------------------------
```

Comme prévu, le temps d'exécution a baissé. Nous trouvons que cette optimisation permet d'avoir un temps 1.3 fois plus rapide. Cette mini modification permet d'obtenir des résultats significatifs.

-> Si on veut optimiser plus, nous pouvons regrouper toutes les fonctions dans le main.
D'autres optimisations devront être faites, mais nous avons rencontré un problème de compatibilité avec OpenCV 1. Donc nous décidons de passer sur une version plus récente (et que l'on maîtrise mieux).

OpenCV 3

Pour faire la migration jusqu'à cette version, nous avons dû modifier la majorité du code. Il est rangé sous des classes et la plupart des méthodes sont définies par des 'namesapce' et non sous forme de simple fonction. Ceci rendra la tâche de profiling plus compliquée mais nous avons tenté notre chance. Peut-être y arriverons nous du premier coup…

Modifications

La capture du flux vidéo se fait de la manière suivant :

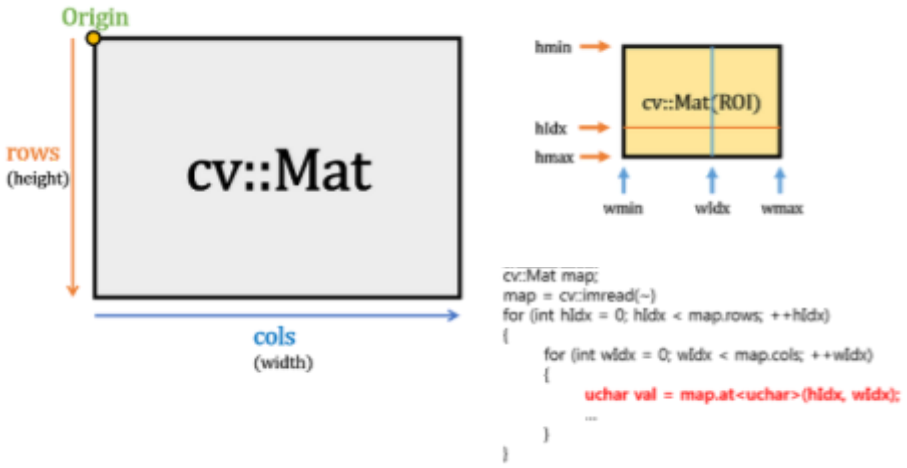
```
// Ouvrir le flux vidéo
  cv::VideoCapture capture(video_input);
```

Pour récupérer une image à partir de ceci, nous utiliserons l'opérateur >>

```
cv::Mat Image_IN;
capture >> Image_IN;
```

Le format d'une image est différent de celui étudié dans la première partie. Ici les pixels sont sauvegardés sous une forme matricielle. Ceci facilitera la manipulation des pixels.

Voici un schéma expliquant les caractéristiques de ce format :



Pour accéder à la valeur d'un pixel nous utiliserons la méthode .at :

```
(int)Image_MEDIAN.at<uchar>(j, i)
```

Nous nous intéressons maintenant à la modification de la résolution des images capturées. Ceci sera utilisé pour optimiser le temps d'exécution.
A l'aide de ces deux lignes nous fixons la hauteur et la largeur de l'image :

```
capture.set(CV_CAP_PROP_FRAME_WIDTH, width);
capture.set(CV_CAP_PROP_FRAME_HEIGHT, height);
```

Ces deux paramètres sont spécifiés lors de l'exécution de l'application. Ils dépendent aussi de la caméra utilisée.

Optimisations

Nous sommes reparties avec les algorithmes optimisés de la partie d'avant et nous avons essayé de gagner un peu plus en temps de traitement d'image.

Réduction de la résolution de l'image

La première solution que nous envisageons d'intégrer est de modifier la résolution de l'image capturée. Ceci nous permettra de réduire le temps de calcul vu qu'il y aura moins de pixel dans l'image. Pour ce faire, nous récupérons les résolutions supportés par notre webcam à l'aide de la commande Linux **v4l2-ctl --list-formats-ext** .

```
pi@raspberrypi:~ $ v4l2-ctl --list-formats-ext
ioctl: VIDIOC_ENUM_FMT
        Type: Video Capture

        [0]: 'MJPG' (Motion-JPEG, compressed)
                Size: Discrete 1280x720
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 640x480
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 544x288
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 320x240
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 432x240
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 160x120
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 800x600
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 864x480
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 960x720
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 1024x576
                        Interval: Discrete 0.033s (30.000 fps)
                Size: Discrete 1920x1080
                        Interval: Discrete 0.033s (30.000 fps)
```

Ces valeurs seront inscrites dans un tableau comme le montre le code ci-dessous :

```
void get_res(int res ,int *w,int *h){
  if(res < 10 ){
    int list_w[9] = {1280,640,544,320,432, 160,800,960,1024};
    int list_h[9] = {720,480,288,240,120,600,480,720,576};
    *w = list_w[res];
    *h = list_h[res];
  }else{
    *w= 432; *h = 240;
  }
}
```

Il suffit de spécifier la valeur de la variable “res” pour obtenir la valeur de la résolution correspondante. Ceci se fera lors de l'exécution de l’application.

Réduire les opération mathématique

Nous commençons par remplacer et regrouper les assignations d’addition dans une seule variable afin de réduire au maximum les opérations mathématiques.
Cela se résume avec l’exemple suivant :

Avant	`Gx_sum += Data_Median_out[(i * step_gray) + 0 + j] * G_x[0][0];` `Gx_sum += Data_Median_out[(i * step_gray) + 2 + j] * G_x[0][2];`

<table>
<tr><td></td><td>

```
      Gx_sum += Data_Median_out[(i * step_gray + 1) + j] *
G_x[1][0];
      Gx_sum += Data_Median_out[(i * step_gray + 1) + 2 +
j] * G_x[1][2];

      Gx_sum += Data_Median_out[(i * step_gray + 2) + j] *
G_x[2][0];
      Gx_sum += Data_Median_out[(i * step_gray + 2) + 2 +
j] * G_x[2][2];
```

</td></tr>
<tr><td>Après</td><td>

```
Gx_sum = (G_x[0][0] * (int)Image_MEDIAN.at<uchar>(j, i)) +
(G_x[0][2] *(int)Image_MEDIAN.at<uchar>(j + 2, i))
+(G_x[1][0] * (int)Image_MEDIAN.at<uchar>(j, i + 1)) + (
G_x[1][2] * (int)Image_MEDIAN.at<uchar>(j + 2, i + 1))
+(G_x[2][0] * (int)Image_MEDIAN.at<uchar>(j, i + 2)) +
(G_x[2][2] * (int)Image_MEDIAN.at<uchar>(j + 2, i + 2));
```

</td></tr>
</table>

Résultat pour une résolution de 432 x 240

Image en entrée

Après niveaux de gris

Après filtre Médiane

Filtre Sobel

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.92      1.17     1.17 49150000     0.00     0.00  unsigned char& cv::Mat::at<unsigned
char>(int, int)
 23.63      1.99     0.82  2046800     0.00     0.00  median_opti9(int*)
 15.08      2.51     0.52       20    26.01    91.13  median_filter()
 13.63      2.98     0.47       20    23.50    57.68  filtre_sobel_v2()
  6.38      3.20     0.22  6220800     0.00     0.00  cv::Vec<unsigned char, 3>&
cv::Mat::at<cv::Vec<unsigned char, 3> >(int, int)
  3.77      3.33     0.13       20     6.50    21.97  grey_filter()
  1.45      3.38     0.05  2046800     0.00     0.00
__gnu_cxx::__enable_if<std::__is_integer<int>::__value, double>::__type
std::sqrt<int>(int)
  1.16      3.42     0.04  6220800     0.00     0.00  cv::Vec<unsigned char, 3>::operator[](int)
  0.87      3.45     0.03
cvflann::anyimpl::big_any_policy<cvflann::anyimpl::empty_any>::~big_any_policy()

                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.29% of 3.45 seconds

index % time    self  children    called     name
                                  <spontaneous>
[1]     99.0    0.00    3.42                 main [1]
                0.52    1.30      20/20          median_filter() [2]
                0.47    0.68      20/20          filtre_sobel_v2() [4]
                0.13    0.31      20/20          grey_filter() [6]
                0.00    0.00      96/96          cv::String::String(char const*) [18]
                0.00    0.00      96/96          cv::String::~String() [19]
                0.00    0.00      84/84          cv::_InputArray::_InputArray(cv::Mat const&) [21]
                0.00    0.00      84/84          cv::_InputArray::~_InputArray() [22]
                0.00    0.00       4/4           std::vector<int, std::allocator<int> >::vector() [41]
                0.00    0.00       4/4           std::vector<int, std::allocator<int> >::~vector() [42]
                0.00    0.00       3/3           cv::MatSize::operator()() const [48]
                0.00    0.00       3/3           cv::Mat::operator=(cv::MatExpr const&) [45]
                0.00    0.00       3/3           cv::MatExpr::~MatExpr() [47]
                0.00    0.00       1/1           cv::Mat::type() const [60]
-----------------------------------------------
```

Le profiling nous montre un temps d'exécution (**3.45 secondes**) qui est beaucoup plus court que celui obtenu avec la dernière optimisation de la version OpenCV 1.

Conclusion

Ce TP nous a permis d'approfondir les connaissances vues en cours de vision et traitement d'images embarquées. Grâce à ce projet, nous avons pu mieux comprendre et exploiter les algorithmes de traitement d'image. Nous avons utilisé deux versions différentes d'OpenCV afin d'obtenir le résultat le plus optimal en sortie. Enfin, pour optimiser un maximum notre programme nous avons utilisé l'outil d'analyse Gprof afin de mesurer le temps d'exécution des fonctions pour les améliorer. Ce TP était très intéressant avec une réelle application du cours.

Nous n'avons pas pu utiliser d'autre technique d'optimisation tel que :

- Hardware acceleration
- Parallélisme (///)