TRAITEMENT D'IMAGE
Débruitage

Adrien MARQUÈS - Alexis HELSON

Septembre 2015

Prélude

1 Introduction

De nombreuses images numériques sont produites aujourd'hui : imagerie médicales, images grand public (appareils photos numériques), numérisation de films anciens, etc... Une problématique qui se pose est le débruitage des images obtenues. Nous proposons dans ce dossier d'aborder certaines techniques utilisées.

Image numérique

Une image numérique de taille $n_x \times n_y$ est composée de pixels (le nombre de pixels est $n = n_x.n_y$), et pour chaque pixel on dispose (selon les cas) :
- une valeur entière comprise entre 0 et 255 (image en niveaux de gris codée en 8 bits)
- trois valeurs entières comprises entre 0 et 255, correspondant aux trois canaux de couleur (RGB)
- une valeur entière codée sur 12 ou 16 bits, ou même sur 1 seul, mais ces cas ne seront pas traités ici.

Bruit

Dans une image numérique, les processus de dégradation peuvent provenir de plusieurs sources, et donner du bruit qui peut avoir différentes expressions mathématiques. Notons u l'image d'origine (l'image «réelle») et u^{OBS} l'image mesurée. Les exemples traités dans ce dossiers sont décrits ci-dessous.

- l'image u^{OBS} est mesurée par des appareils électroniques qui sont perturbés par des fluctuations aléatoires, et ces perturbations viennent s'ajouter au signal mesuré. On parle de *bruit addifit*, et on a $u^{OBS} = u + n$, où n est le bruit ajouté.

- l'image u^{OBS} comporte certains pixels où la valeur est aberrante (par exemple à cause d'un capteur défectueux, ou de défaut de transmission de l'information). Pour fixer les idées, on parle de *bruit «Salt and Pepper»* lorsque certains pixels (au hasard) ont un niveau 0, et certains ont un niveau 255.

- lorsque le processus de formation de l'image comporte des compteurs de photons ou des photomultiplicateurs, l'erreur commise peut être proportionnelle au signal mesuré. On parle alors de *bruit multiplicatif* et on a $u^{OBS} = u.(1 + n)$.

Cette étude vise à améliorer les méthodes de *nettoyage d'image*, et donc de développer un programme universel de débruitage, c'est-à-dire qu'il réduirait convenablement tout type de bruit, dépendant toutefois de paramètres adaptés à chacun.

Avant d'essayer d'améliorer quelque chose, il faut d'abord le comprendre.

Nous avons donc commencé par lire des études sur le débruitage, la détection de contours, de formes, et la vectorisation d'images. Pour cela, nous avont étudié:

- Le débruitage standard, c'est à dire les méthodes déjà existantes.

- Le filtrage par convolution, afin de mettre en valeur les contours et formes et d'effectuer des "lissages".

- La détection de formes, afin d'effectuer un débruitage sans «effet escalier».

- La détection de contours, afin d'optimiser la détection de formes.

Le programme a été développé en Python2 pour la simplicité et l'efficacité du langage. Le code est disponible sur Github à l'adresse suivante:

https://github.com/xdrm-brackets/Denoising.py

2 Architecture et structure du programme

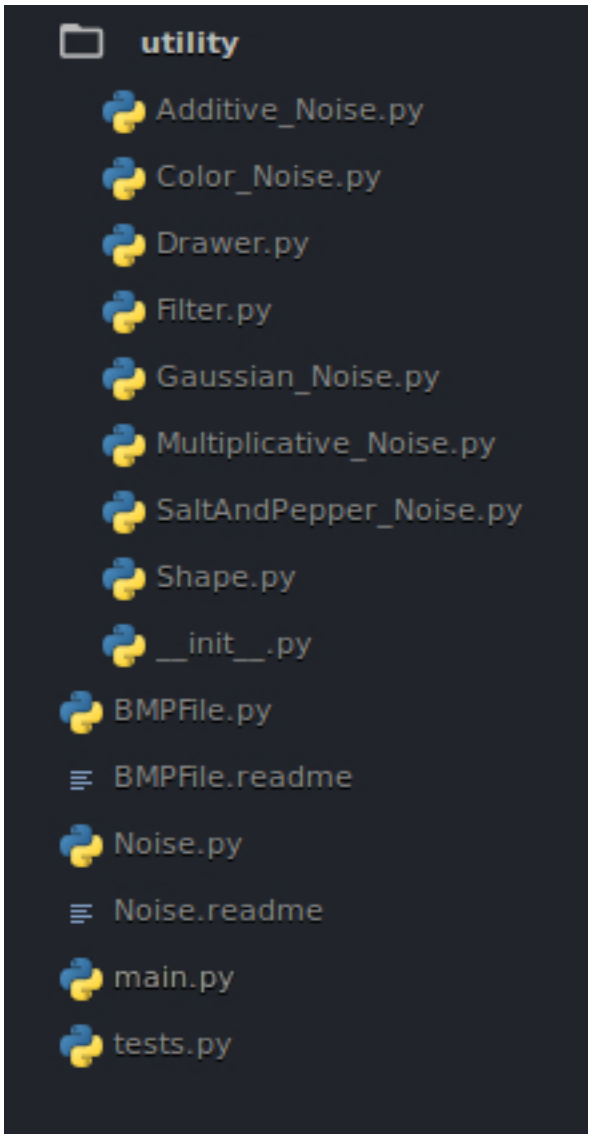


Figure 1. Structure de fichiers

main.py
Programme principal, il affiche une interface (en console) permettant d'éxécuter une des opérations disponibles.

tests.py
Contient les corps de toutes le opérations pouvant être choisies dans le programme principal.

BMPFile.py
Contient les structures *BMPFile*, *BMPContent*, *BMPHeader*, permettant d'encoder et de décoder un fichier BMP. Ainsi que la strucure *RGBPixel* permettant de coder un pixel.

Noise.py
Contient tout les traitements tels que les bruitages et débruitages, les filtres, et les détections de formes et contours. (il importe tout le dossier *utility*)

utility/

SaltAndPepper_Noise.py
Contient les méthodes de bruitage et débruitage du bruit de type *Poivre & Sel*.

Additive_Noise.py
Contient les méthodes de bruitage et débruitage du bruit de type *additif*.

Gaussian_Noise.py
Contient les méthodes de bruitage et débruitage du bruit de type *gaussien*.

Multiplicative_Noise.py
Contient les méthodes de bruitage et débruitage du bruit de type *multiplicatif*.

Color_Noise.py
Contient les méthodes de bruitage et débruitage du bruit de type *multicolore*.

Filter.py
Contient les méthodes de filtrage (standard et par convolution).

Shape.py
Contient les méthodes de détection de formes et de contours

Le fichier **tests.py** utilise la structure *BMPFile* afin de lire/écrire les fichiers ainsi que la structure *Noise* qui contient toutes les méthodes de bruitage et de débruitage, les filtres et les détections de formes et de contours. Le fichier ***Noise.py*** (qui contient la structure de même nom) appelle les différents fichiers contenus dans le dossier **utility**. Les fichiers contenus dans **utility** peuvent être:
- Un fichier qui contient toutes les méthodes pour un type de bruit (bruitage et débruitage)
- Le fichier *Filters.py* qui contient toutes les méthodes de filtrage
- Le fichier *Shapes.py* qui contient toutes les méthodes de détection de formes et contours

3 Variables globales

Ces valeurs seront utilisées pour la majeure partie des calculs et correspondent à des notations générales à l'image ou aux autres notations utilisées.

Soit $(n_x, n_y) \in \mathbb{N}^2$, respectivement la largeur et la hauteur de l'image en pixels.
Soit la matrice I_{n_y,n_x} correspondant à la répartition des pixels dans l'image.
Soit le couple $(x, y) \in \mathbb{N}^2$, les coordonnés d'un pixel dans I, tel que $x \in [0; n_x[,\ et\ y \in [0; n_y[$.
Soit $\vec{u}$ le vecteur associé à une unité de contour, tel que $||\vec{u}|| = 1,\ et\ sa\ direction\ d = \frac{n\pi}{2}[2\pi]\ rad,\ avec\ n \in \mathbb{N}$.
Soit K_n, la matrice carré d'ordre n. Par exemple $K_3 = \begin{pmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{pmatrix}$

Afin de comparer les images nous avons implémenté le SNR, qui permet de donner calculer la qualité du débruitage.

```python
# import des libs internes
from utility import SaltAndPepper_Noise, Additive_Noise, Multiplicative_Noise, Color_Noise, Gaussian_Noise
from utility import Filter, Shape

from BMPFile import RGBPixel




class Noise:

    # instanciation des classes qui sont dans utility/
    def __init__(self):
        self.SaltAndPepper  = SaltAndPepper_Noise.SaltAndPepper_Noise();
        self.Additive       = Additive_Noise.Additive_Noise();
        self.Multiplicative = Multiplicative_Noise.Multiplicative_Noise();
        self.Gaussian       = Gaussian_Noise.Gaussian_Noise();
        self.Color          = Color_Noise.Color_Noise();

        self.Filter         = Filter.Filter();
        self.Shape          = Shape.Shape();

    # retourne le SNR d'une image bruitée par rapport à sa référence
    # @param uRef                   matrice de pixels de l'image d'origine
    # @param uNoisy                 matrice de pixels de l'image bruitée
    #
    # @return SNR                   retourne le SNR associé
    #
    def SNR(self, uRef, uNoisy, grayscale=True): ...




    # retourne la puissance d'une image (somme du carré de ses valeurs) teinte de gris
    # @param pixelMap               matrice de pixels de l'image de laquelle on veut la puissance
    #
    # @return power                 puissance de l'image
    #
    def Power_grayscale(self, pixelMap): ...


    # retourne la puissance d'une image (somme du carré de ses valeurs) teinte de gris
    # @param pixelMap               matrice de pixels de l'image de laquelle on veut la puissance
    #
    # @return power                 puissance de l'image
    #
    def Power(self, pixelMap): ...;
```

Figure 2. Implémentation du SNR.

Encodage et décodage de fichiers

4 Fonctionnement global

La lecture et l'écriture des fichiers se fait octet par octet, la structure *BMPFile* permet de transformer les octets d'un fichier BMP en matrice de pixels, et inversement, de transformer une matrice de pixels en fichier BMP. Les types d'encodage des couleurs pris en charge sont 8 (nuance de gris), et 24 (rouge, vert, bleu) bits par pixels. Les pixels sont du type *RGBPixel*, ce type contient les coordonnées du pixel (x, y), ainsi que sa couleur au format RGB (rouge, vert, bleu) chaque nuance prend sa valeur entre 0 et 255.

Le seul format pris en charge est le format BMP. Pour que le programme fonctionne avec d'autres formats (jpeg, png, gif, etc), il suffirait de récupérer une matrice de *RGBPixel* à partir d'une image. Toutes les méthodes de traitement d'image du programme agissent uniquement sur cette matrice. Il serait ainsi possible de convertir une image vers un autre format. Le fichier BMP peut se décomposer en 3 parties:

- Le *header* donne des informations sur le fichier et son contenu.
  Sa taille est de 54 octets.

- La *palette* donne des informations sur l'encodage des couleurs.
  Sa taille est variable, elle est donnée dans le header.

- Le *bitmap* est la matrice de pixels, c'est le «corps» du fichier.
  Sa taille est variable, elle est donnée dans le header.

4.1 Le header

Le header fournit des informations sur le fichier, sur l'image, sur la palette et sur les tailles et positions des différentes parties. Il est

Position (octets)	**Taille**	**Libellé**
0	2 octets	signature, toujours 4D42
2	4 octets	taille du fichier (octets)
6	2 octets	réservé (défaut: 0)
8	2 octets	réservé (défaut: 0)
10	4 octets	offset de la bitmap (octets)
10	4 octets	taille du BITMAPINFOHEADER (octets)
14	4 octets	largeur de l'image (pixels)
18	4 octets	hauteur de l'image (pixels)
22	2 octets	nombre de plans (défaut: 0001)
26	2 octets	nombre de bits par pixels (1, 4, 8, ou 24)
30	4 octets	type de compression (0=rien, 1=RLE-8, 2=RLE-4)
34	4 octets	taille de l'image padding inclus (octets)
38	4 octets	résolution horizontale (pixels)
42	4 octets	résolution verticale (pixels)
46	4 octets	nombre de couleurs ou 0
50	4 octets	nombre de couleurs importantes ou 0

Figure 1 - BMP Header

4.2 La palette

La palette permet de définir les différentes couleurs d'une image. Elle correspond à une liste de couleurs ou bien à un «code» associé à un encodage spécifique. Elle dépend en premier lieu du nombre de bits par pixels. Cette valeur nous informe sur combien de bits est codée chaque couleur, les valeurs possibles sont 1, 4, 8, et 24. Le nombre de couleurs possibles dépend directement du *bpp* (bits par pixels), plus il est grand, plus il y a de nuances. $nbcouleurs = 2^{bpp}$. Les différents types de palette correspondant à un *bpp* sont décrites ci-dessous.

4.2.1 Encodage en 8 bits par pixels

Pour une image encodée en 8 bits par pixels, chaque pixel est codé sur 1 octet. Il y a donc une possibilité de $2^8 = 256$ couleurs. Ces 256 couleurs sont définies dans la *palette* sur 4 octets chacunes. Les 3 premiers octets contiennent les couleurs au format RGB, et le dernier octet est à "0", c'est un octet de bourrage car la taille du fichier doit être multiple de 4. Par exemple si l'on veut que la couleur "0" corresponde à du rouge (R=255,G=0,B=0), et la couleur 1 à du rose (R=255,G=0,B=255), la palette commencera par:

rang	**couleur**	**octet1**	**octet2**	**octet3**	**octet4**
0	rouge	255	0	0	0
1	rose	255	0	255	0
..	..	..	..	..	..
255	..	..	..	..	0

Par défaut, nous utilisons uniquement le codage en 8 bits par pixels pour des nuances de gris, notre palette a donc comme valeurs:

rang	**couleur**	**octetS**	**octet2**	**octet3**	**octet4**
0	noir	0	0	0	0
1	noir	1	1	1	0
2	noir	2	2	2	0
..	..	..	..	..	..
254	blanc	254	254	254	0
255	blanc	255	255	255	0

* $4 \times 256 = 1024$ *octets* au total

4.2.2 Encodage en 24 bits par pixels

Pour une image encodée en 24 bits par pixels, chaque pixel est codé sur 3 octets au format BGR, ce format est équivalent au format RGB, mis à part l'ordre des octets qui est inversé, celà correspond à $2^24 = 16.777.216$ couleurs par pixels. La *palette* associée est définie par défaut comme suit:

Position (octets)	**Taille (octets)**	**Valeur entière**	**Valeur ASCII**
0	1	66	"B"
1	1	71	"G"
2	1	82	"R"
3	1	115	"s"
4	48	0	inconnu
52	1	2	inconnu
53	15	0	inconnu

* 68 octets au total

4.3 Le bitmap

Le *bitmap* du fichier (matrice de pixels) peut avoir plusieurs formats qui sont propres au nombre de bits par pixels et dépendent de la *palette*. De plus la matrice est dans l'ordre inverse, c'est à dire que le premier pixel est en fait le dernier, il convient donc d'inverser les lignes et les colonnes. De manière générale les pixels sont enregistrés en une succession de lignes. Le nombre d'octets de chaque ligne doit être un multiple de 4, est donc ajouté en fin de ligne un «padding» composé de 0 à 3 octets ayant pour valeur zéro.

```
5   ##############################################
6   # classe qui parse le header (binaire) en objet #
7   ##############################################
8   class BMPHeader:
9
10      # CONSTRUCTEUR: initialise les variables
11      def __init__(self): ...
33
34
35      # parse le header au format bin en objet
36      def parse(self, binHeader=""): ...
70
71
72      # fonction qui créer <self.binData> à partir des attributs
73      def unparse(self): ...
102
103     # Retourne au format humain, toutes les infos du header
104     def info(self, type=0): # 0 = int, 1 = hex ...
141
142
143     # Affiche au format humain, toutes les infos du header
144     def printInfo(self, type=0): # 0 = int, 1 = hex ...
177
178
179     # convertit les octets <bytes> en entier
180     def toInt(self, bytes): ...
185
186     # écrit le valeur entière <value> en octet bourrés jusqu'à la taille <size> dans le tableau <intData>
187     def fromInt(self, value, size): ...
196
197
198
199
200 ################################################
201 # classe qui parse le content (binaire) en matrice #
202 ################################################
203 class BMPContent:
204
205     # CONSTRUCTEUR: instancie les attribus
206     def __init__(self): ...
210
211     # parse le content (bin) <binContent> avec les informations:
212     #    <header>    BMPHeader de l'image en question
213     def parse(self, binContent, header): ...
252
253
254     # unparse une map de pixels en binaire
255     def unparse(self, pixelMap, palettes, headerHandler=None): ...
300
301
302 ################################
303 # classe contenant un pixel RGB #
304 ################################
305 class RGBPixel:
306     def __init__(self, r=0, g=0, b=0, x=-1, y=-1, bpp=24): ...
335
336
337     def setRGB(self, r=0, g=0, b=0, x=None, y=None, bpp=24):
338         self.__init__(r=r, g=g, b=b, x=x, y=y, bpp=bpp);
339
340     def setBin(self, binData, width, padding, index, bpp=24): ...
399
400     # retourne la moyenne des 3 canaux RGB (c'est à dire en nuances de gris)
401     def grayscale(self):
402         return ( self.r + self.g + self.b ) / 3
403
404
405
406
407
408 ################################################
409 # classe qui parse un fichier BMP complet en objet #
410 ################################################
411 class BMPFile:
412
413     # CONSTRUCTEUR: instancie les attributs
414     def __init__(self, drawerBool=False): ...
425
426     # parse à partir de <binFile> en objets <BMPHeader> et <BMPContent>
427     def parse(self, binFile=""): ...
462
463     # unparse à partir d'un <BMPHeader> et d'un <BMPContent>
464     def unparse(self, newBpp=None): ...
497
498
499     # écrit l'image dans un fichier
500     def write(self, filename):
501         with open(filename,"w") as file:
502             file.write( self.binData );
503
504
```

Figure 3. Structure du fichier BMPFile.py

Bruits

5 Le bruit de type «Poivre & Sel»

Définition graphique: L'image est parsemée de pixels ayant une teinte extrême (blancs ou noirs) sans rapport avec leur contexte (voisinage).

Bruitage: Le bruitage se fait en fonction du paramètre réel $seuil \in [0;1]$. Il y a donc $n = seuil \times n_x \times n_y$ couples (x, y) aléatoires qui sont en noir ou blanc (choix aléatoire), ce qui correspond à $(100 \times seuil)$ % de la totalité des pixels.

Débruitage: L'image est débruitée en fonction de deux paramètres *seuil* et *borne*, tout deux compris dans $[0; 255]$. Chaque pixel est traité si et seulement si il est proche du noir ou du blanc, c'est-à-dire compris dans $[0; borne]$ pour le noir et dans $[255 - borne; 255]$ pour le blanc. Chaque pixel traité est comparé à la couleur moyenne de ses 3 (haut gauche) voisins, si la différence avec sa couleur est supérieure à *seuil*, le pixel se voit attribuer la couleur moyenne de ses voisins. Il est à noter que si borne vaut 255, tous les pixels seront traités, de même que si seuil vaut 255, tous les pixels traités se verront affecter la couleur moyenne de leurs voisins.

Image originale | Bruitée, seuil de 0.5 | Débruitée sans lissage | Débruitée avec lissage

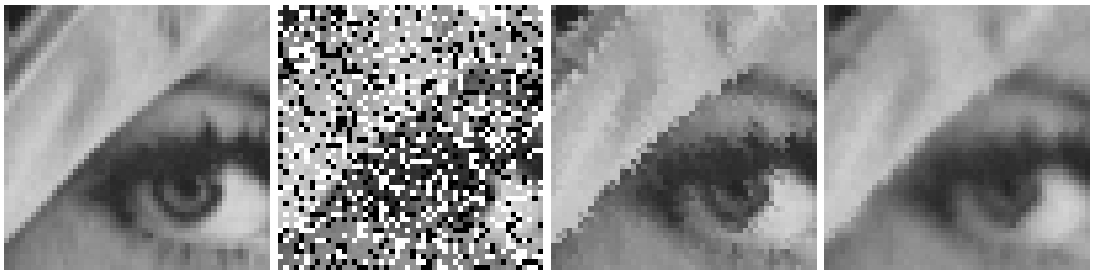


Image originale (zoom) | Bruitée, seuil de 0.5 (zoom) | Débruitée sans lissage (zoom) | Débruitée avec lissage (zoom)

On remarque bien ici que les contours subissent un «effet escalier» ce qui dégrade en majeure partie le rendu. Nous avons réglé partiellement ce problème en ajoutant un lissage au processus de débruitage, mais le lissage réduit aussi la netteté de l'image. Par contre, les couleurs sont convenablement restituées, de même pour les teintes et contrastes. Il suffirait de traiter l'image en prenant en compte les contours, ce qui permettrait de gagner en qualité.

```
    # Applique le bruitage de type "Poivre & Sel" sur la matrice de pixels #
    ######################################################################
    # @param pixelMap       Matrice de pixel à traiter (modifier)
    # @param seuil          pourcentage de l'image à bruiter (50% <=> 1 pixel sur 2 est bruité)
    #
    def set(self, drawer, pixelMap, seuil=10):
        seuil = float(seuil);

        while seuil >= 1:
            seuil /= 100.0

        nbPixel = int( len(pixelMap) * len(pixelMap[0]) * seuil )

        for bruit in range(0, nbPixel):
            x = random.randint(0, len(pixelMap[0]) - 1 )
            y = random.randint(0, len(pixelMap)    - 1 )

            if random.randint(0,1) == 1:
                pixelMap[y][x].setRGB(r=255,g=255,b=255, x=x, y=y, bpp=pixelMap[y][x].bpp);
            else:
                pixelMap[y][x].setRGB(r=0,g=0,b=0, x=x, y=y, bpp=pixelMap[y][x].bpp);

        drawer.fill(pixelMap);
```

Figure 4. Création et application du bruit «Poivre & Sel»

```
    # Applique le débruitage de type "Poivre & Sel" sur la matrice de pixels #
    ##########################################################################
    # @param pixelMap       Matrice de pixel à traiter (modifier)
    # @param seuil          Seuil à partir duquel on doit traiter les pixels (écart entre la moyenne des pixels avoisinant et le pixel concerné)
    # @param borne          0 = Noir pur et blanc pur sont enlevés; 255 ou + = tout les pixels sont traités
    #
    def unset(self, drawer, pixelMap, seuil=5, borne=5):
        width  = len( pixelMap[0] )
        height = len( pixelMap    )

        if seuil < 0 or seuil > 255: # si le seuil est incohérent  => valeur par défaut (5)
            seuil = 5;

        if borne < 0 or borne > 255: # si la borne est incohérente => valeur par défaut (5)
            borne = 5;


        # on parcourt tout les pixels
        for y in range(0, len(pixelMap)):
            for x in range(0, len(pixelMap[y])):

                # on calcule la moyenne des valeurs R G B du pixel courant
                pMoy = ( pixelMap[y][x].r + pixelMap[y][x].g + pixelMap[y][x].b ) / 3

                # si couleur proche du blanc ou noir (en fonction de la borne)
                if pMoy >= 255-borne or pMoy <= borne:
                    xmin, ymin, xmax, ymax = x, y, x, y;                    # les bornes ducarré 3x3 autour du pixel
                    rMoy, gMoy, bMoy, count = 0.0, 0.0, 0.0, 0              # initialisation des variables de moyennes et de total
                    rInterval, gInterval, bInterval, rgbInterval = 0, 0, 0, 0  # initialisation des variables d'intervalles entre les couleurs


                    # GESTION DES ANGLES

                    # ordonnées: borne inférieure
                    if y-1 > -1:
                        ymin = y-1
                    # ordonnées: borne supérieure
                    if y+1 < height:
                        ymax = y+1
                    # abscisses: borne inférieure
                    if x-1 > -1:
                        xmin = x-1
                    # abscisses: borne supérieure
                    if x+1 < width:
                        xmax = x+1


                    # pixels = [ pixelMap[y][xmin], pixelMap[y][xmax], pixelMap[ymin][x], pixelMap[ymax][x] ];
                    # for p in pixels:
                    #   if p != pixelMap[y][x]:
                    #       rMoy += p.r;
                    #       gMoy += p.g;
                    #       bMoy += p.b;
                    #       count += 1

                    # on parcourt le carré de 3x3
                    for j in pixelMap[ymin:ymax]:
                        for pix in j[xmin:xmax]:

                            # si le pixel n'est pas le pixel courant (mais ceux autour)
                            if pix != pixelMap[y][x]:
                                # calcul de la moyenne autour du pixel
                                rMoy  += pix.r;
                                gMoy  += pix.g;
                                bMoy  += pix.b;
                                count += 1

                    # si il y a au moins un pixel autour (normalement tjs mais évite l'erreur div par zéro)
                    if count > 0:
                        # on calcule les moyennes somme(xi) / n
                        rMoy = int( rMoy / count )
                        gMoy = int( gMoy / count )
                        bMoy = int( bMoy / count )

                        # calcul de la différence entre les couleurs du pixel et la moyenne des couleurs des pixels autour
                        rInterval = abs( pixelMap[y][x].r - rMoy )
                        gInterval = abs( pixelMap[y][x].g - gMoy )
                        bInterval = abs( pixelMap[y][x].b - bMoy )

                        # calcul de la différence en nuance de gris (moyenne des couleurs)
                        rgbInterval = ( rInterval + gInterval + bInterval ) / 3

                        # si la couleur est trop "différente" (dépend du seuil) alors on remplace sa couleur par la moyenne des couleurs alentours
                        if rgbInterval > seuil:
                            pixelMap[y][x].setRGB(r=rMoy, g=gMoy, b=bMoy, x=x, y=y, bpp=pixelMap[y][x].bpp);

                            drawer.setPixel( pixelMap[y][x] );

        drawer.refresh();
```

Figure 5. Calcul et suppression du bruit «Poivre & Sel»

6 Le bruit additif de Bernouilli

Définition graphique: L'image est parsemée de pixels ayant une teinte plus ou moins incohérente avec leur contexte (voisinage). Ceci ressemble au bruit «Poivre et Sel» mais les pixels ne sont pas uniquement blancs ou noirs, mais parfois plus proches de la teinte d'origine.

Bruitage: Le bruitage se fait en fonction du paramètre réel $seuil \in [0;1]$. Il y a donc $n = seuil \times n_x \times n_y$ couples (x, y) aléatoires qui voient leur couleur changée (choix aléatoire), ce qui correspond à $(100 \times seuil)$ % de la totalité des pixels. Soit un pixel aléatoirement choisi de couleur $c \in [0; 255]$. Soit:
- sa teinte reste inchangée, mais il est éclairci (plus ou moins)
- sa teinte reste inchangée, mais il est foncé (plus ou moins)

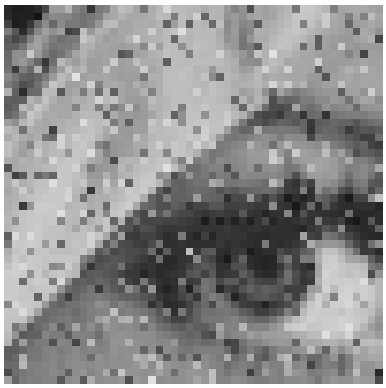
Débruitage n°1: L'image est débruitée en fonction du paramètre $seuil \in [0; 255]$. Chaque pixel traité est comparé à la couleur moyenne de ses 8 voisins, si la différence avec sa couleur est supérieure à *seuil*, le pixel se voit attribuer la couleur moyenne de ses voisins. Il est à noter que si seuil vaut 255, tout les pixels traités se verront affecter la couleur moyenne de leurs voisins.

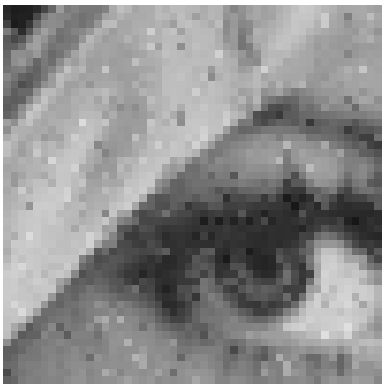

Image originale


Bruitée, seuil de 0.3


Débruitée, seuil de .35


Image originale (zoom)


Bruitée, seuil de 0.3 (zoom)


Débruitée, seuil de .35 (zoom)

On remarque bien ici qu'une majorité des pixels ajoutés lors du bruitage a été proprement retirée, mais une partie reste bien visible. Celà est dû au fait que les pixels considérés comme bruit se voient appliquer un filtre moyen (couleur moyenne des voisins), mais que certains voisins peuvent être eux-aussi corrompus. De plus, certains pixels corrompus ne sont pas traités du fait qu'ils ont une teinte trop proche de la moyenne voisine. Il faut noter que lors du débruitage «Poivre et Sel» le paramètre *borne* permettait d'éviter ce problème car nous avions des références absolues : le noir et le blanc. Nous avons donc décidé d'améliorer cet algorithme de débruitage (cf. *Débruitage n°2*).

Débruitage n°2: L'image est débruitée en fonction du paramètre *seuil* $\in \mathbb{N}^+$, tel que *seuil* $\in [0; \frac{n^2-1}{2}]$. Pour chaque pixel, on établit la matrice carré M_n d'ordre n correspondant à son voisinage. Soit $P_{2,2}$ le pixel en cours, et

$$M_3 = \begin{pmatrix} P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,1} & P_{3,2} & P_{3,3} \end{pmatrix}$$

On calcule ensuite M'_n la matrice carré d'ordre n correspondant à la différence absolue à $P_{2,2}$.

$$M'_3 = \begin{pmatrix} |P_{1,1} - P_{2,2}| & |P_{1,2} - P_{2,2}| & |P_{1,3} - P_{2,2}| \\ |P_{2,1} - P_{2,2}| & |P_{2,2} - P_{2,2}| & |P_{2,3} - P_{2,2}| \\ |P_{3,1} - P_{2,2}| & |P_{3,2} - P_{2,2}| & |P_{3,3} - P_{2,2}| \end{pmatrix} = \begin{pmatrix} |P_{1,1} - P_{2,2}| & |P_{1,2} - P_{2,2}| & |P_{1,3} - P_{2,2}| \\ |P_{2,1} - P_{2,2}| & 0 & |P_{2,3} - P_{2,2}| \\ |P_{3,1} - P_{2,2}| & |P_{3,2} - P_{2,2}| & |P_{3,3} - P_{2,2}| \end{pmatrix}$$

Soit m les valeurs de M'_n excepté $P_{2,2}$ telles que $m_i \leq m_{i+1}$, ce qui nous permet de calculer le **poids statistique** du pixel.

$$PS(P_{2,2}) = \sum_{i=0}^{\frac{n^2-1}{2}} m_i$$

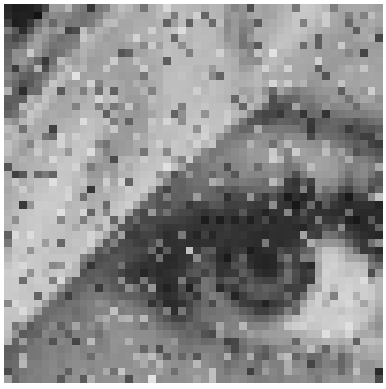
Si le poids statistique d'un pixel est supérieur à *seuil*, alors un filtre moyen lui est appliqué.


Image originale


Bruitée, seuil de 0.3


Débruitée, seuil de .05


Image originale (zoom)


Bruitée, seuil de 0.3 (zoom)


Débruitée, seuil de .05 (zoom)

On remarque qu'avec cette nouvelle méthode, les contours sont moins dégradés et le bruit et d'autant plus diminué. Cet algorithme permet de différencier un contour d'un pixel bruité car la différence entre les poids statistiques écarte les deux cas.

```
    # Applique le bruitage de type "Additif de Bernouilli" sur la matrice de pixels #
    ################################################################################
    # @param pixelMap       Matrice de pixel à traiter (modifier)
    # @param seuil          pourcentage de l'image à bruiter (50% <=> 1 pixel sur 2 est bruité)
    #
    def setBernouilli(self, drawer, pixelMap, seuil=10):
        seuil = float(seuil);

        while seuil >= 1:
            seuil /= 100.0

        nbPixel = int( len(pixelMap) * len(pixelMap[0]) * seuil )

        for bruit in range(0, nbPixel ):
            x = random.randint(0, len(pixelMap[0]) - 1 )
            y = random.randint(0, len(pixelMap)    - 1 )



            if random.randint(0,1) == 1:
                maxColor = max(pixelMap[y][x].r, pixelMap[y][x].g, pixelMap[y][x].b)
                randomAdd = random.randint(0, (255-maxColor) / 2 )
            else:
                minColor = min(pixelMap[y][x].r, pixelMap[y][x].g, pixelMap[y][x].b)
                randomAdd = - random.randint(0, minColor / 2 )

            pixelMap[y][x].setRGB(
                r=pixelMap[y][x].r + randomAdd,
                g=pixelMap[y][x].g + randomAdd,
                b=pixelMap[y][x].b + randomAdd,
                x=x,
                y=y
            );

        drawer.fill( pixelMap );
```

Figure 6. Création et application du bruit de Bernouilli

```
    # Applique le débruitage de type "Additif" sur la matrice de pixels #
    ######################################################################
    # @param pixelMap       Matrice de pixel à traiter (modifier)
    # @param seuil          Seuil de "poids statistique" à partir duquel on doit traiter les pixels compris entre 0 et 100
    #
    # @return cleanMatrix   matrice propre qui est retournée
    def unset2(self, drawer, pixelMap, seuil=10):
        width  = len( pixelMap[0] )
        height = len( pixelMap    )
        ordre  = 3 # ordre matrice carré
        ordreN = (ordre**2 - 1)/2

        # matrice qui sera retournée
        cleanMatrix = []

        while seuil >= 1: # si le seuil n'est pas un pourcentage, on le met en pourcentage
            seuil /= 100.0;

        seuil *= 256 * ordreN



        # on parcourt tout les pixels
        for y in range(0, len(pixelMap)):
            cleanMatrix.append( [] );
            for x in range(0, len(pixelMap[y])):

                # on ajoute le pixel à la matrice "propre"
                cleanMatrix[y].append( RGBPixel(
                    r = pixelMap[y][x].r,
                    g = pixelMap[y][x].g,
                    b = pixelMap[y][x].b,
                    x = pixelMap[y][x].x,
                    y = pixelMap[y][x].y,
                    bpp = pixelMap[y][x].bpp,
                ));
                drawer.setPixel( cleanMatrix[y][x] );

                # on calcule la moyenne des valeurs R G B du pixel courant
                pMoy = ( pixelMap[y][x].r + pixelMap[y][x].g + pixelMap[y][x].b ) / 3

                xmin, ymin, xmax, ymax = x, y, x, y;                    # les bornes ducarré 3x3 autour du pixel
                rMoy, gMoy, bMoy, count = 0.0, 0.0, 0.0, 0              # initialisation des variables de moyennes et de total
                rInterval, gInterval, bInterval, rgbInterval = 0, 0, 0, 0  # initialisation des variables d'intervalles entre les couleurs


                # GESTION DES ANGLES

                # ordonnées: borne inférieure
                if y-1 > -1:
                    ymin = y-1
                # ordonnées: borne supérieure
                if y+1 < height:
                    ymax = y+1
                # abscisses: borne inférieure
                if x-1 > -1:
                    xmin = x-1
                # abscisses: borne supérieure
                if x+1 < width:
                    xmax = x+1



                # contiendra la matrice M' des poids statistiques
                neighboursAbsoluteDiff = []

                # on parcourt le carré de 3x3
                for j in pixelMap[ymin:ymax+1]:
                    for pix in j[xmin:xmax+1]:
                        # si le pixel n'est pas le pixel courant (mais ses voisins) et que sa couleur n'est pas trop éloignée des autres
                        if pix != pixelMap[y][x]:
                            # calcul de la moyenne autour du pixel
                            rMoy  += pix.r;
                            gMoy  += pix.g;
                            bMoy  += pix.b;
                            count += 1
                            # ajout aux poids statistiques
                            neighboursAbsoluteDiff.append( abs( (pix.r+pix.g+pix.b)/3 - pMoy ) );

                # on garde que la moitié la plus petite
                statisticWeight = 0;

                neighboursAbsoluteDiff.sort() # on trie la liste

                # on récupère la somme de la moitié des éléments les plus petits (car triée)
                for infVal in range(0, ordreN):
                    if infVal >= len(neighboursAbsoluteDiff): # si liste vide on arrête
                        break;
                    statisticWeight += neighboursAbsoluteDiff[infVal] # on effectue la somme


                # si il y a au moins un pixel autour (normalement tjs mais évite l'erreur div par zéro)
                if count > 0:
                    # on calcule les moyennes somme(xi) / n
                    rMoy = int( rMoy / count )
                    gMoy = int( gMoy / count )
                    bMoy = int( bMoy / count )

                    # si la couleur est trop "différente" (dépend du seuil) alors on remplace sa couleur par la moyenne des couleurs alentours
                    if statisticWeight > seuil:
                        cleanMatrix[y][x].setRGB(r=rMoy, g=gMoy, b=bMoy, x=x, y=y);
                        drawer.setPixel( cleanMatrix[y][x] );

        return cleanMatrix;
```

Figure 7. Calcul et suppression du bruit de Bernouilli.

7 Le bruit additif gaussien

Définition graphique: Chaque pixel de l'image est plus ou moins bruité, l'image semble avoir moins de contraste et chaque pixel est éclairci ou foncé.

Bruitage: Le bruitage se fait en fonction du paramètre réel $\sigma \in [0; 255]$. Soit u l'image d'origine, u^{OBS} l'image observée et X le bruit, on a :

$$u^{OBS} = u + X$$

Afin de bruiter l'image, pour chaque pixel, on récupère un nombre réel $r \in [0; 1]$ aléatoire, puis on applique la formule suivante:

$$u_{i,j}^{OBS} = u_{i,j} + r\sigma$$

Débruitage: L'image est débruitée en utilisant le **Débruitage n°2** ainsi conçu pour le bruit additif de Bernouilli.

Image originale

Bruitée, 15%

Débruitée, seuil 1%

Image originale (zoom)

Bruitée, 15% (zoom)

Débruitée, seuil 1% (zoom)

On remarque bien ici que les contours subissent un lissage ce qui dégrade en majeure partie le rendu. Mais pour ce qui est du bruit additif Gaussien, le débruitage n'a jamais un très bon résultat sans l'utilisation de techniques avancées, nous remarquons donc un bon résultat comparé aux exemples vus sur internet.

```
    # Applique le bruitage de type "Additif Gaussien" sur la matrice de pixels #
    ############################################################################
    # @param pixelMap       Matrice de pixel à traiter (modifier)
    # @param seuil          pourcentage de l'image à bruiter (50% <=> 1 pixel sur 2 est bruité)
    #
    def setGaussian(self, drawer, pixelMap, sigma=10):
        width  = len( pixelMap[0] )
        height = len( pixelMap    )

        sigma = float(sigma);

        # vérification de la cohérence de sigma
        if 0 > sigma or sigma > 255:
            print "sigma have incoherent value"
            exit();


        from numpy import random as npRand # random.rand(height,width) renvoie une matrice de flottants entre 0 et 1
        factors = npRand.rand(height, width)


        # on parcourt en même temps les facteurs aléatoires et la matrice de pixels
        for lineP, lineF in zip(pixelMap, factors):
            for pixel, fact in zip(lineP, lineF):

                # ajout ou suppression (choix aléatoire)
                if random.randint(0,1) == 1:
                    fact *= -1

                r = int( pixel.r + sigma * fact )
                g = int( pixel.g + sigma * fact )
                b = int( pixel.b + sigma * fact )

                # on attribue les valeurs aux pixels
                pixel.setRGB(
                    r = 0 if r<0 else ( 255 if r > 255 else r),
                    g = 0 if g<0 else ( 255 if g > 255 else g),
                    b = 0 if b<0 else ( 255 if b > 255 else b),
                    x = pixel.x,
                    y = pixel.y
                );
                drawer.setPixel( pixel );

```

Figure 8. Création et application du bruit gaussien.

Optimisation du débruitage par la détection de contours

Problème: Nous avons remarqué précedemment que de manière générale après un débruitage, les contours de l'image deviennent irréguliers. Soit l'image subit un «effet escalier», c'est à dire que certains pixels débruités ont une couleur trop éloignée de leur contexte ce qui donne l'impression que l'image n'est pas bien nettoyée. Soit l'image subit un effet de «lissage» ce qui dégrade les délimitations et contours présents.
textbfSolution: La solution retenue s'appuie sur la **détection de contours**, cette partie vise à «régulariser» les contours.

Processus détaillé Nous avons décidé d'opter pour un algorithme en 6 étapes:

Etape 1: Premier nettoyage de l'image permettant d'optimiser la détection de contours.
Etape 2: Détection de contour afin de les indépendentialiser des formes pleines.
Etape 3: Débruitage de l'image d'origine.
Etape 4: Débruitage des contours de l'image d'origine à partir des contours.
Etape 5: Superposition des contours débruités sur l'image débruitée.
Etape 6: Lissage minimal transitionnel de la superposition.

Etape 1: En premier lieu, il faut réduire suffisamment le bruit pour pouvoir correctement détecter les contours. Nous utilisons donc un débruitage standard adapté (méthodes vues précedemment).
Etape 2: En second lieu, il faut isoler les contours, il existe plusieurs moyens de le faire:
- soustraire l'image de base à l'image lissée (par filtre moyen)
- utiliser un algorithme non-récursif d'indépendentisation de contour.
Nous avons mis au point les 2 méthodes vues ci-dessus afin de comparer leur temps d'exécution et leur efficacité. Ceci nous permettra de plus de les combiner.

Méthode 1 - soustraction de l'image lissée

L'image est lissée avec un filtre moyen. Nous effectuons un produit de convolution (noté $\otimes$) avec l'image I et le noyau tel que défini :

$$I_{lisse} = I \otimes \frac{1}{8}\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

On effectue ensuite la soustraction matricielle afin de mettre les contours en valeur

```
    # applique le filtre de "convolution" sur l'image #
    ###################################################
    # @param pixelMap       la matrice de pixels à modifier
    #
    # @history
    #           applique le filtre
    def Convolution(self, drawer, pixelMap, kernel=None):
        width  = len( pixelMap[0] )
        height = len( pixelMap    )

        if kernel == None:
            print "no kernel given!"
            exit()

        # nb total résultant du kernel des valeurs positives
        kernelFactor = 0;
        for b in kernel:
            for a in b:
                if a > 0:
                    kernelFactor += a;

        kMidWidth  = len( kernel[0] ) // 2
        kMidHeight = len( kernel    ) // 2

        # map de résultat (filtrée)
        convolvedMap = []

        # on parcourt tout les pixels
        for y in range(0, height):

            # on rajoute une ligne à la map filtrée
            convolvedMap.append( [] )

            for x in range(0, width):

                pixel = pixelMap[y][x];

                # on définit la matrice de même taille que le kernel mais correspondant aux pixels superposés
                pixelM = [];

                for b in range(-kMidHeight, 1+kMidHeight):
                    pixelM.append( [] );
                    for a in range(-kMidWidth, 1+kMidWidth):
                        # on met les valeurs entre 0 et longueur ou largeur pour pas récupérer les valeurs de l'autre côté (lissage bizarre)
                        ruledX = 0 if x+a<0 else width-1  if x+a>=width  else x+a;
                        ruledY = 0 if y+b<0 else height-1 if y+b>=height else y+b;

                        # if x+a<0 or x+a>=width or y+b<0 or y+b>=height: # si pixel dépasse de l'image, on l'ajoute pas

                        # on ajoute le pixel courant
                        pixelM[len(pixelM)-1].append( pixelMap[ruledY][ruledX] );

                r,g,b = 0,0,0

                # on effectue la convolution
                for linePixelM, lineKernel in zip(pixelM, kernel):
                    for pix, fact in zip(linePixelM, lineKernel):
                        # pour chacun des filtres
                        r += pix.r * fact
                        g += pix.g * fact
                        b += pix.b * fact

                r = int(r/kernelFactor) % 256
                g = int(g/kernelFactor) % 256
                b = int(b/kernelFactor) % 256


                # définition des couleurs sur la map filtrée
                convolvedMap[y].append( RGBPixel(
                    r   = r,
                    g   = g,
                    b   = b,
                    x   = x,
                    y   = y,
                    bpp = pixel.bpp
                ) )
                drawer.setPixel( convolvedMap[y][x] );

        drawer.refresh();

        return convolvedMap
```

Figure 9. Implémentation du produit de convolution.

Méthode 2 - indépendentisation de contour

Un algorithme lance sur chaque pixel de l'image un "envahisseur", c'est-à-dire un algorithme qui référencie chaque pixel considéré dans la même forme que le pixel de départ. par exemple, on obtient le résultat ci-dessous en essayant de récupérer les formes noires uniquement :

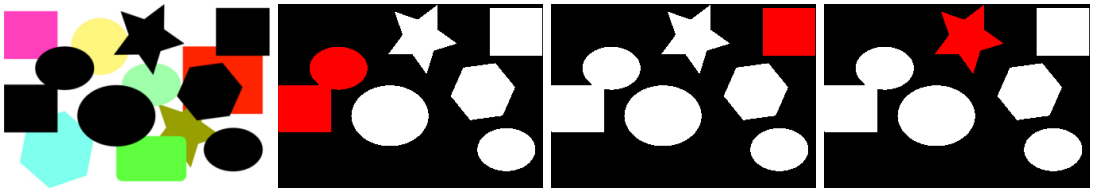


Image originale Détection sur forme 1 Détection sur forme 2 Détection sur forme 3

Les résultats sont corrects mais le traitement devient vite très long, le temps d'éxécution moyen pour ces résultats est de 20 secondes. Nous avons donc décidé de créer un nouvel algorithme qui lui récupérerait uniquement les contours.

Etape 3: Consiste à appliquer le débruitage standard à l'image d'origine.

Etape 4: Consiste à appliquer le débruitage standard à l'image d'origine, mais uniquement sur les zones des contours récupérées à la deuxième étape.

Etape 5: Correspond à la superposition des contours débruités sur l'image débruitée.

Etape 6: Correspond au lissage du contours des contours.

8 Mise en valeur par produit de convolution

Afin de faire ressortir les contours d'une image nous avons utilisé le produit de convolution, qui consiste à appliquer un noyau (de convolution) à l'image d'origine ce qui résulte en une image nettoyée de ses zones pleines. Nous avons notamment utilisé les filtres de *Prewitt*, *Laplace*, *Roberts*, et s*Sobel*. Le problème est que la détection de contours par cette méthode fait ressortir le bruit. Nous avons décidé d'appliquer un débruitage sommaire afin d'avoir un meilleur résultat.

9 Processus détaillé

ç *Méthode 2 - indépendentisation de contour*

References

[1] microsoft.com, Official Bitmap Header Format

[2] Mohammed M. Siddeq, Dr. Sadar Pirkhider Yaba, Using Discrete Wavelet Transform and Wiener filter for Image De-nosing, Wasit Journal for Science & Medicine, 2009

[3] Roger L. Easton, Jr., Fundamentals of Digital Image Processing, 22 November 2010

[4] Antoine Manzanera, ENSTA, Cours Master2 IAD, TERI : Traitement et reconnaissance d'images

[5] UFRIMA, Détection de contours

[6] Image processing for space's pictures, Chapter 4

[7] Ecole polytechnique de Montréal, Mathématiques Appliquées et Génie Industriel, Applications des mathématiques : Détection des contours d'une image, exercice 6

[8] Roman Garnett, Timothy Huegerich, Charles Chui, Fellow, IEEE, and Wenjie He*, Member, IEEE, A Universal Noise Removal Algorithm with an Impulse Detector

[9] IN_GBM_04_CONTOURS, Détection de contours

[10] Peter Selinger, Potrace: a polygon-based tracing algorithm, September 20, 2003