****LUFAU Killian

TROESCH Guillaume

**Traitement d’images**

UTT Automne 2016

**Sommaire**

1. **Introduction**

**3**

1. **Avant tout**

**5**

1. **Fonctions**
2. **Miroir 9**
3. **Inversion des couleurs 9**
4. **Retoucher les couleurs 10**
5. **Nuances de gris 10**
6. **Flouter en pixélisé 11**
7. **Détecter les contours 12**
8. **Rotation de l’image 13**
9. **Changer la luminosité 15**
10. **Flouter en modifiant le bruit 16**
11. **Conclusion**

**17**

1. **Introduction**

De nos jours les logiciels de traitement d’image sont indispensables, du simple Paint au plus complexe Gimp ou Photoshop. Nous en avons tous besoin à un moment ou un autre. Pour par exemple redimensionner une image ou changer ses couleurs…

C’est dans cette optique d’utilité qui a suscité notre intérêt que nous avons décidé de réaliser le projet de traitement d’image. L’écriture de fichiers et la communication avec l’ordinateur était aussi un aspect intéressant que nous retrouvions dans ce sujet.

Notre projet est donc réalisé en langage C, il utilise uniquement les bibliothèques standard *stdio* et *stdlib* ainsi que la bibliothèque *math.h.*  Il a pour but la prise en charge et la modification des images PPM et PGM (format bitmap assez simple à prendre en charge).

Les fonctions que nous avons implémentées sont des modifications utiles que tout le monde utilise et qui restent les fonctions essentielles à la modification basique d’images (rotation, inversion de couleurs, flou, luminosité…).

Il existe deux types d’images que notre programme prend en charge, les images PPM et les PGM, ces deux formats sont distingués par une différence simple : les PGM sont en nuances de gris (8 bits par pixel) contrairement aux PPM qui sont en couleurs (8bits par composantes et 3 composantes par pixel, donc 24 bits par pixel). Une image PGM composé de 100 000 pixels aura donc 100 000 caractères dans son fichier PGM, mais une image PPM en aura 300 000.

Un fichier PPM ou PGM est donc tout d’abord composé du « magic number » qui indique le type de l’image, P5 pour PGM ou P6 pour PPM.

Il est ensuite suivi de la taille de l’image (Largeur\*Hauteur)

Et enfin nous avons le nombre de valeurs maximales que prend une composante d’un pixel. Dans la plupart des cas 255, soit 8bits d’intensité.

Voici donc un exemple d’en-tête d’une image PPM sur 24bits/pixel de taille 1920\*1080

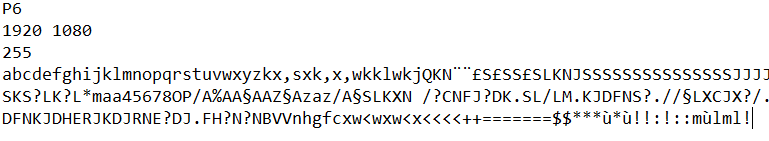
P6

1920 1080

255

Toutes les données qui se situent après l’intensité maximale d’une composante constituent les données de l’image, ce sont les valeurs des pixels. Pour ce projet nous les avons visualisés en caractère et nous les avons manipulés en binaire.

Dans l’exemple ci-après « abc » correspond au tout premier pixel de l’image (de coordonnées [0 ;0]). Le « e» correspond à la composante verte du deuxième pixel (de coordonnée [1 ;0] ).



On obtient donc un fichier à la structure assez simple et avec des pixels rangés dans l’ordre (de gauche à droite et de haut en bas) avec des caractères regroupés ou non par 3 (selon le type de l’image qui va permettre une lecture assez facile de toutes les informations de l’image.

1. **Avant tout**

**Principes généraux de notre programme**

Nous avons choisi quelques conventions pour simplifier le travail sur notre programme, tout d’abord nous avons décidé d’utiliser les mêmes compteurs dans tout notre code : i (pour parcourir la largeur), j (pour la hauteur), et k (pour les composantes RVB).

Nous parcourons donc notre tableau de cette manière :

Pour j de 0 à hauteur de l’image par pas de 1

{

Pour i de 0 à largeur de l’image par pas de 1

{

Pour k de 0 à 3ou1(selon le type de l’image) par pas de 1

{

//MODIFICATIONS

}

}

}

La seule variable de cette convention est pour le compteur k qui varie soit de 0 à 1 (Image PGM) soit de 0 à 3 (Image PPM).

Pour remédier à cela nous utilisons une fonction simple k\_max()qui prend en argument le type de notre image et qui renvoie soit 3 soit 1. Nous avons donc pour toutes nos boucles de ce type, k qui varie de 0 à k\_max(type de notre image). Nous verrons tout de suite après comment on obtient le type de notre image.

Ce code est basé sur l’utilisation de structures, nous avons donc créé notre structure image qui contient :

La variable entière type (6 pour une image PPM ou 5 pour une image PGM)

La variable entière largeur

La variable entière hauteur

La variable entière max\_couleur

La variable pixels, un triple pointeur sur un caractère non signé

Le pointeur triple *pixels* constitue notre matrice 3D de pixels, il est de non signé car l’on souhaitait manipuler des valeurs de caractères allant de 0 à 256 et non de -127 à 127.

**Quelques fonctions de base**

**Allocation dynamique**

alloc\_dyna()

La fonction demande les entiers largeur, hauteur et type pour retourner le pointeur triple alloué dynamiquement.

Nous utilisons la fonction de l’allocation dynamique pour créer le pointeur triple pixels dans la structure de l’image. Elle se fait en 3 étapes : l’allocation de la largeur grâce à la fonctionmalloc*,* qui alloue la taille d’un caractère non signé sur la largeur :

pixels = (unsigned char\*\*\*)malloc(largeur \* sizeof(unsigned char\*\*));

Puis similairement, une allocation de la hauteur pour toutes les cases de la largeur:

for(i=0 ; i < largeur ; i++)

{ pixels[i] = (unsigned char\*\*)malloc(hauteur \* sizeof(unsigned char\*)); }

Et également une allocation de la 3e dimension du tableau qui sert pour les 3 couleurs Rouge, Vert et Bleu, pour toutes les cases de la hauteur sur toute la largeur grâce à 2 boucles.

for(i=0 ; i < largeur ; i++)

{

for(j=0 ; j < hauteur ; j++)

{

pixels[i][j] = (unsigned char\*)malloc(k\_max(type) \* sizeof(unsigned char));

}

}

Nous avons ajouté un cas où une erreur serait produite lors de l’allocation dynamique avec la fonction malloc, le programme se fermerait aussitôt :

exit(EXIT\_FAILURE);

**Lecture de notre fichier**

**Lecture()**

La fonction permet de lire et de retourner la structure image.

Notre image étant un fichier composé d’un en-tête comme nous l’avons vu dans notre introduction, nous devons tout d’abord extraire ces informations de ce fichier : type, largeur, hauteur, intensité des pixels.

Pour cela nous avons bien entendu créé un pointeur sur fichier qui pointe sur le fichier portant le nom précisé par l’utilisateur. Cette simple commande permet de récupérer toutes les informations de l’en-tête :

fscanf(PointeurSurFichier, ‘‘P%d %d %d %d %d ’’, &type, &largeur, &hauteur, &max\_couleur) ;

Connaissant ainsi notre largeur et hauteur d’image, on peut alors assigner à chaque case de notre matrice une composante de pixel en utilisant notre boucle usuelle des triples for.

Au sein de cette boucle nous lisons et stockons chaque caractère dans notre tableau à l’aide de cette ligne :

fscanf (PointeurSurFichier,"%c", &image1.pixels[i] [j] [k]);

**Ecriture dans un fichier**

Copie()

La fonction demande la structure de l’image pour la copier et ne retourne rien.

Cette fonction est appelée à chaque début de la boucle dans la fonction *main*. Elle permet de créer un fichier, écrire dedans, et afficher l’image.

Le fichier créé est de type ppm ou pgm selon la variable *type* de la structure de l’image, il sera nommé test.ppm ou test.pgm

L’écriture se fait similairement à la lecture, en commençant par l’en-tête :

fprintf(new\_fichier, "P%d\n%d %d\n%d\n", type, largeur, hauteur, max\_couleur);

Puis nous écrivons les caractères de notre tableau à la suite, de la même manière que la lecture :

fputc(image1.pixels[i][j][k], new\_fichier);

Puisque cette fonction est appelée à chaque début de la boucle principale, nous en profitons pour afficher l’image avec Irfanview grâce à l’instruction suivante :

system("Irfanview.exe test.ppm"); (respectivement .pgm selon le type évidemment)

Notons qu’Irfanview.exe doit être dans le dossier du main.c dans ce cas.

L’instruction fclose(new\_fichier); a été ajoutée pour afficher correctement avec Irfanview.

**Principe de notre main**

main()

Après la lecture de l’image, la fonction main propose une boucle qui ne s’arrête que lorsque le choix de l’utilisateur est 10 (soit « quitter »). Dans cette boucle, on crée un fichier grâce à la fonction *Copie* énoncée précédemment, puis on demande simplement à l’utilisateur quelles modifications il veut apporter à l’image, numérotées de 1 à 9, puis 10 pour terminer.

Les modifications possibles sont donc les suivantes :

- Faire un miroir de l’image, c’est-à-dire la retourner (Page 9)

- Inverser les couleurs (Page 9)

- Apporter des modifications sur les couleurs (Page 10)

- Nuancer en gris (Page 10)

- Rendre l’image floue (Page 11)

***-*** Détecter les contours et les retracer en noir et blanc (Page 12)

- Faire une rotation de l’image (Page 13)

- Changer la luminosité de l’image (Page 15)

*-* Rendre l’image floue en modifiant le bruit (Page 16)

Ces fonctions demandent toutes en argument la structure de l’imageet renvoie ainsi l’image modifiée.

Elles sont énoncées dans l’ordre du programme.

1. **Fonctions**

**Miroir**

Miroir()

Dans cette fonction, le but est d’intervertir chaque case [i][j][k] par la case [largeur-1-i] [j][k] dans le cas d’un miroir horizontal. Pour un miroir vertical, on intervertit chaque case [i][j][k] avec la case [i][hauteur-1-j][k]. On soustrait le 1 car le pixel inférieur droit est représenté dans notre matrice par la case [largeur-1][hauteur-1]

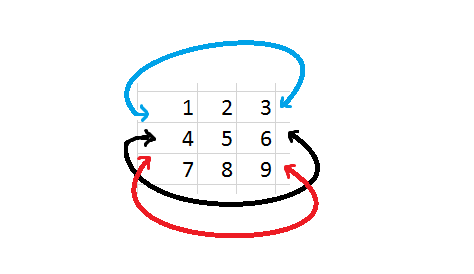
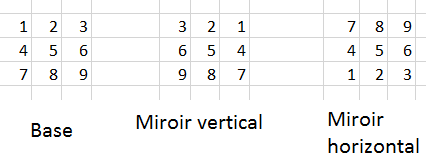


Schéma de fonctionnement pour un miroir horizontal



Résultat Usuel

**Inversion de couleurs**

Inverser\_Couleurs()

Cette fonction fut la plus simple à implémenter, il suffit de parcourir notre matrice de pixels dans l’ordre habituel et de remplacer chaque valeur de pixels par 255-sa valeur. Ici le problème que l’on rencontre parfois du pixel qui dépasse la valeur 255 ou qui devient négatif ne se présente pas car la valeur initiale de chaque pixel est comprise entre 0 et 255.

**Retouche des couleurs**

Retoucher\_Couleurs()

Cette fonction permet de modifier les couleurs : on peut ajouter une valeur à une composante Rouge, Vert ou Bleu ; soustraire par une valeur ; multiplier par une valeur ; ou remplacer par une valeur.

Cette fonction commence donc par demander le choix de l’utilisateur.

Si on a une image au format ppm, on demande ensuite à quelle couleur il veut affecter sa modification. Si c’est au format pgm, on n’a qu’une seule couleur, donc on fait directement la modification.

Pour la modification, dans le programme on change la valeur sur la bonne couleur donc seule 2 boucles sont nécessaire : la boucle parcourue par le k de la convention servait à parcourir toutes les couleurs. On n’utilise que 2 boucles pour parcourir l’image en largeur et en hauteur.

Un problème s’est posé lorsque les valeurs dépassaient l’intervalle [0 ; 255], on a donc ajouté des conditions pour ne pas dépasser cet intervalle.

**Nuances de gris**

Nuance\_gris()

Cette fonction est assez simple, on donne le choix de la méthode à l’utilisateur pour nuancer en gris son image PPM. Soit la transformer en une image PPM, soit la transformer en image PGM.

Dans le cas d’une image PPM, il faut affecter à chaque composante d’un pixel la même valeur.

Dans le cas d’une image PGM, on affecte simplement la valeur à la seule case disponible par pixel.

Dans les deux cas, la valeur à reporter pour un pixel donné se trouve par 4 méthodes différentes que l’utilisateur choisit : soit on reporte directement la valeur de la composante R de l’image de base, soit on reporte la valeur de la composante V, soit la valeur de la composante B, la dernière méthode consiste à faire la moyenne des trois composantes.

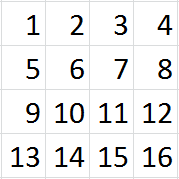
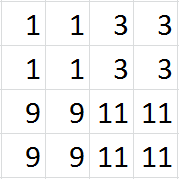
Après avoir obtenu la valeur, on l’injecte dans la nouvelle matrice.

**Flou pixélisé**

Flou\_pixelise()

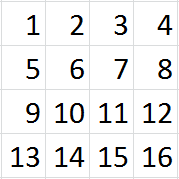
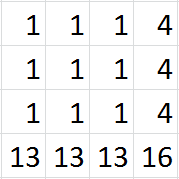
Cette fonction crée une pixellisation de l’image. Le fonctionnement n’est pas le même que celui du flou avec une modification du bruit.

Ici, la fonction permet de changer les pixels de la manière suivante :

C’est-à-dire que les cases en haut à gauche ont été décuplées sur un carré de 2 sur 2, par exemple la case n°1 a été copiée. La définition de case en haut à gauche a été choisie arbitrairement, nous aurions pu copier la case n°2, soit celle en haut à droite.

Nous avons donc un carré de pixels simulant un gros pixel, d’une taille de 2 sur 2 pixels. Avant d’appliquer ce flou pixélisé, la fonction *Flou\_pixelise* demande la taille du nouveau gros pixel. Ici, c’était juste pour une taille de 2x2. Pour une taille de 3x3, on aurait la modification suivante :

Pareil pour une taille de 4x4, et au-delà. Cette demande permet de rendre la pixellisation plus marquée.

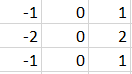
Dans le programme, cette fonction commence donc par demander la valeur, puis à l’aide de différentes boucles, demande de modifier les valeurs en carré de ValeurxValeur pour qu’elles deviennent celle du pixel en haut à gauche. On a donc besoin de 5 compteurs : i, j, k pour parcourir le tableau comme le veut la convention imposée dans l’introduction, puis l et m pour créer le carré.

Une dernière condition a été ajoutée, pour ne pas dépasser la taille de l’image : si les compteurs sortent de la taille de l’image, on ne modifie rien sous peine d’une erreur du programme.

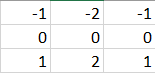
**Détection de contours : SOBEL**

Contours()

Cette fonction utilise l’opérateur de Sobel pour calculer le gradient des composantes de chaque pixel. On souhaite obtenir une image composée que de noir et blanc, le blanc représente les contours.



Opérateur pour calculer le gradient selon x



Opérateur pour calculer le gradient selon y

Ces gradients donnent une idée de la place d’un pixel par rapport à son voisinage, un gradient élevé indique une forte différence par rapport à ses voisins.

Une fois que l’on a les deux gradients on trouve le gradient total en calculant la racine de gradx²+grady².

La méthode utilisée ici est de parcourir une première fois le tableau afin de trouver le max des gradients de tous les pixels (l’image est désormais en PGM donc une seule composante par pixel)

Ensuite on parcourt une deuxième fois l’image en mesurant à chaque fois le gradient, si le gradient est proche du gradient max, cela veut dire que l’on est sur un pixel plutôt différent de ses voisins, on colore alors le pixel en blanc (on remplace sa valeur par 255), s’il n’est pas proche, on le colore en noir (on remplace sa valeur par 0).

On voit alors apparaître un problème : il faut utiliser un deuxième tableau pour pouvoir modifier la valeur des pixels sans influencer le traitement des pixels environnants. On crée donc un deuxième tableau image2 strictement égal à image1, on parcoure image2 pour lire les gradients et on écrit dans image1 255 ou 0. Avant de retourner la valeur de image1 on remplace tous les pixels des bords par des pixels noirs.

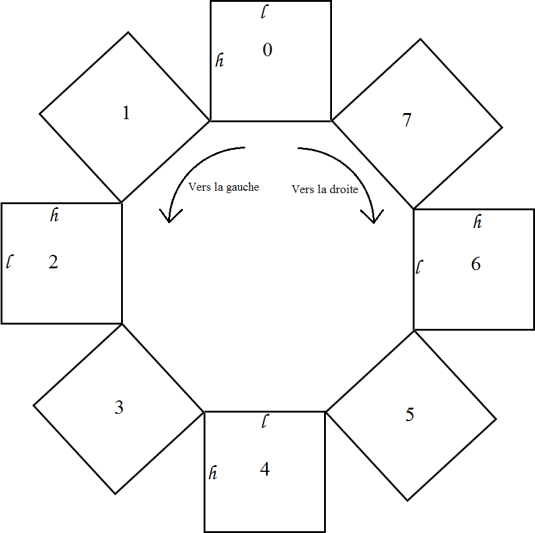
**Rotation**

Tourner()

Cette fonction permet de tourner l’image à 90° ou à 45°. Elle commence donc par demander à l’utilisateur l’angle de rotation et le sens (droite ou gauche).

On utilise un compteur n qui permet de savoir dans quelle position est l’image.

Voici un schéma des rotations possibles grâce à cette fonction, n est au centre de chaque rectangle qui illustre l’image avec la largeur l et la hauteur h :



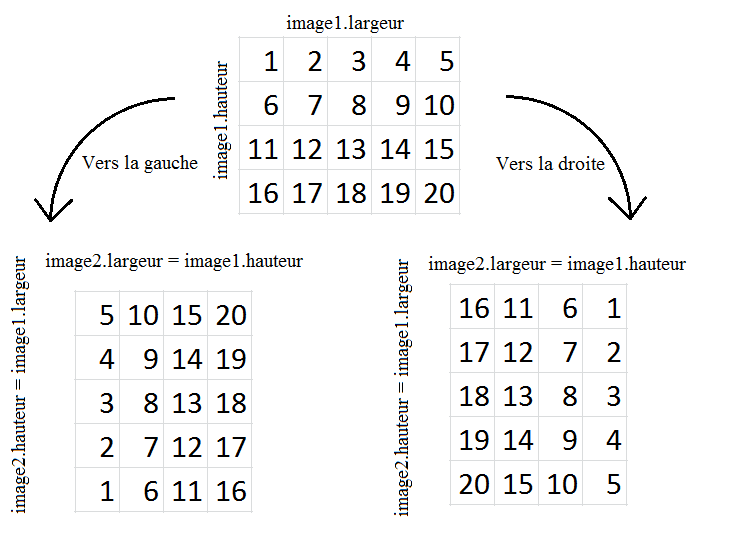
n varie donc entre 0 et 7 selon la position de l’image.

Nous avions commencé par faire les rotations à 90°. La fonction crée donc une 2e image, avec les mêmes caractéristiques que la 1ère image, sauf que la largeur et la longueur était inversées. Ensuite, on remplit simplement les cases du tableau de l’image 2 [i] [j] [k] avec les cases appropriées de l’image 1

[image1.largeur-1-j] [i] [k] pour une rotation vers la gauche,

[j] [image1.hauteur-1-i] [k] pour une rotation vers la droite.

Le schéma suivant résume ce fonctionnement :



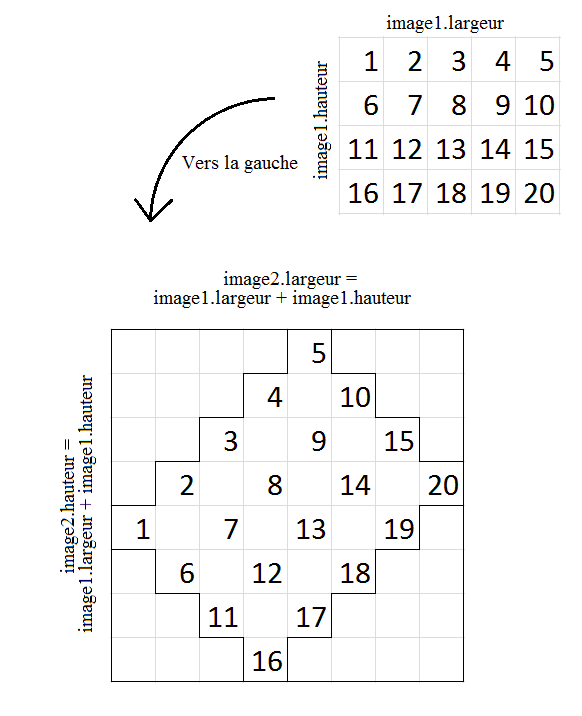
Pour les rotations à 45°, nous avons dû utiliser une image plus grande de taille (largeur+hauteur)x(largeur+hauteur), déclarée et initialisée grâce à la fonction pour allouer dynamiquement, puis rempli le tableau de valeur 0 qui correspond à une couleur noire, pour de la même façon remplir les cases appropriées selon le sens de rotation.

Cette fois on prend les cases de l’image 1 [i] [j] [k] pour les placer correctement sur l’image 2.

[i + j] [image1.largeur – i + j] [k] pour une rotation vers la gauche,

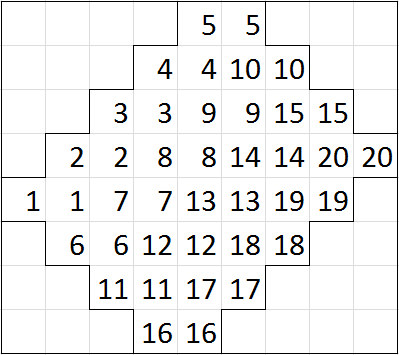
[image1.hauteur + i –j] [i+j] [k] pour une rotation vers la droite.7

Le schéma suivant résume ce fonctionnement :



Les cases non remplie sont des cases noires puisqu’on a rempli le tableau de noir précédemment.

On a donc des cases noires au milieu de l’image, nous avons donc choisi arbitrairement de les remplir de la même valeur que la case à gauche (nous aurions pu choisir celle du dessus par exemple), le résultat final d’une rotation à 45° à gauche donne ainsi :



Notons que la largeur a dû être augmenté de 1 pour avoir ce résultat.

Un dernier cas a été ajouté lorsque l’on passe d’un n impair à un n pair, nous aurions pu garder le fonctionnement précédent, mais l’image déjà tournée de 45° a été agrandie, et aurait été encore agrandie pour refaire une rotation de 45°. Nous avons donc décidé de garder les premières largeurs et hauteurs avant l’entrée dans la boucle principale dans le main, et de les réutiliser lorsqu’on arrive à un n pair.

Le fonctionnement des placements des cases dans le tableau est l’inverse du précédent, et on réobtient une image sans le dédoublement de cases dans le cas d’un n pair.

A la fin de cette fonction, on n’oublie pas de modifier la valeur de n, de la remettre dans l’intervalle [0 ; 7] et de libérer l’espace mémoire allouée par l’ancienne image 1, et on retourne l’image 2.

**Luminosité**

Luminosite()

Avec cette fonction on donne la possibilité à l’utilisateur de choisir la valeur de l’augmentation ou la diminution de la luminosité. On lui demande une valeur entre -255 et 255.

On parcourt ensuite l’image pixel par pixel et composante par composante et on ajoute la valeur entrée par l’utilisateur à la valeur du pixel, si l’addition des deux est supérieure à 255, on remplace directement la valeur du pixel par 255, si en revanche l’addition est inférieure à 0 (valeur entrée négative), on remplace la valeur du pixel par 0.

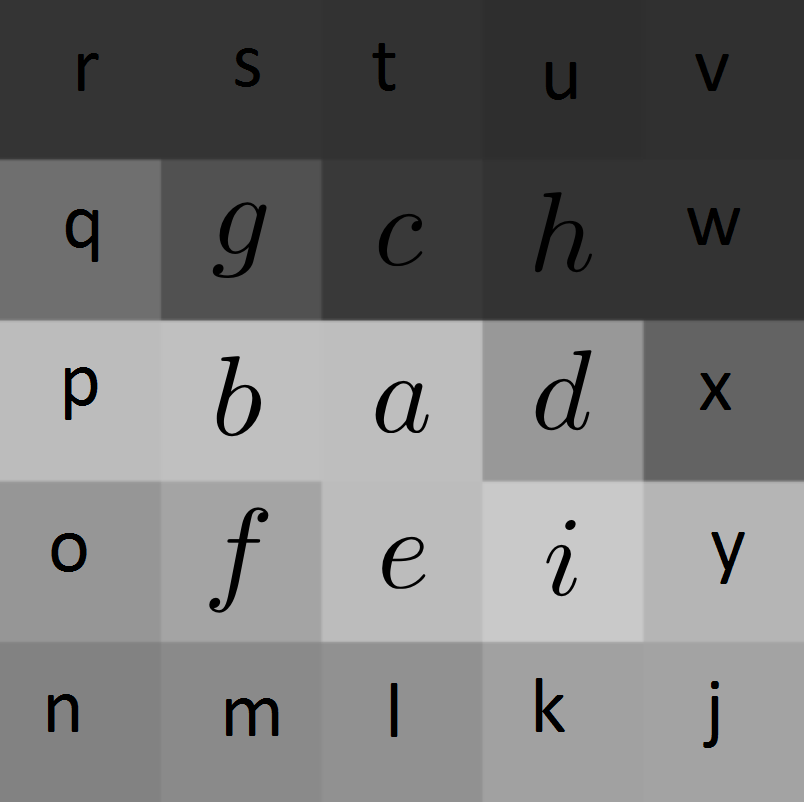
Comme on modifie toutes les composantes de chaque pixel, les couleurs restent fidèles et tendent vers le noir dans le cas où on augmente la valeur des pixels ou tendent vers le blanc dans le cas où on diminue leur valeur.

**Flou**

Flou\_bruit()

Dans cette explication on ne prend pas en compte les composantes RVB de chaque pixel, on prendra juste l’entité du pixel comme un tout, un membre de coordonnée (i,j) de la matrice (largeur,hauteur). On néglige donc le k et l’on considère en quelque sorte que l’on travaille sur une image PGM.

La fonction flou a un principe simple de fonctionnement : pour chaque pixel considéré de coordonnée (i, j) , on le remplace par une moyenne de ses voisins. Cela va ainsi diminuer la précision de l’image, et au passage réduire le bruit de l’image. C’est donc pour cela que l’on va utiliser cette fonction avant la détection de contours (pour limiter la détection de faux contours locaux).



L’image de gauche correspond à une intensité 1, et l’image de droite correspond à une intensité 2 autour du pixel a.

Intensité 1 : a=(a+b+c+d+e+f+g+h+i )/9

Intensité 2 : a= (a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+x+y+z )/ 25

On effectue donc la moyenne de toutes les valeurs environnantes avec une formule qui se dégage pour une intensité i : pixel(i,j)=voisins/ (2\*i+1)² (carré de côté 2i+1 pixel).

Le problème de cette manipulation est qu’il faut penser aux pixels proches des bords de l’image.

Par exemple si l’on voulait effectuer une moyenne d’intensité 1 sur le pixels r de l’image de droite, on ne pourrait prendre en compte que les pixels r, s, g et q, et pas les autres qui sortent de la grille.

Le principe de l’algorithme repose alors sur des boucles for qui parcourent l’image pixel par pixels. Et à l’intérieur de ces boucles et ce pour chaque pixel, on intègre deux compteurs supplémentaires compt1 et compt2 qui vont servir à parcourir le carré regroupant les pixels voisins. Compt1 et compt2 varient de -intensité à +intensité et pour chaque valeur de compt1 et compt2, on doit vérifier si le pixel voisin que l’on souhaite additionner à notre pixel (i, j) n’est pas à l’extérieur de notre matrice. Pour cela on vérifie simplement que i+compt1 est supérieur à 0 et inférieur à la largeur de l’image. Et que j+compt2 est supérieur à 0 et inférieur à la hauteur de l’image.

Nous avons dû créé un tableau temporaire image2 afin de stocker les pixels moyennés de image1, car si l’on le stockait à la place du même pixel, cela changerait la valeur des pixels voisins pour les pixels suivants.

Chaque moyenne calculée pour un pixel [i, j] de image1 est alors stockée dans la case [i, j] de image2.

On renvoie la valeur de la structure image2 à la fin de cette fonction.

\*A FINIR\* : Conclusion