ministère éducation nationale

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

éduscol

Ressources pour le lycée général et technologique

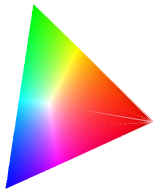
Ressources pour le cycle terminal général et technologique

Informatique et Sciences du Numérique

Algorithmes pour le traitement d'images - 2

Juin 2012

Présentation / Algorithmes pour le traitement d'images - 2



1 / Thème abordé

1.1 Problématique, situation d'accroche

À l'issue de la première partie, où le traitement de chaque pixel d'une image bitmap était indépendant des pixels voisins, il s'agit ici d'aborder des algorithmes dont le résultat est lié aux pixels environnants.

Un certain nombre de bases ayant été acquises, l'intérêt est maintenant de proposer le travail sous forme de mini-projets. Les filtres de type « matrice de convolution » constituent un thème riche, mais qui peut être abordé de façon simple, et donner lieu à plusieurs mini-projets. Un seul et même algorithme permet d'appliquer différents filtres à une image. Le passage d'un filtre à un autre se fait en ne modifiant qu'une seule ligne de calcul, dont l'essentiel est une somme de produits. L'utilisation conjointe du logiciel GIMP permet de se rendre compte des possibilités qu'offre ce type de filtres.

L'objectif final est de faire comprendre qu'un paramétrage adéquat du filtre permet de détecter les contours d'une image. Comme dans la première partie, les exemples annexés sont en langage Python et ils utilisent la bibliothèque PIL (Python Imaging Library).

À partir de l'image de gauche, comment obtenir celle du milieu ou celle de droite ? Comment détecter les contours d'une image ? Comment reproduire ce que fait GIMP à l'aide d'un programme d'une vingtaine de lignes tout au plus ? Il s'agit de comprendre l'algorithme utilisé, et en quoi la modification des paramètres du filtre donne des résultats très différents. On peut aussi montrer un petit dessin animé obtenu par détection de contours à partir d'une vidéo (nombreux exemples disponibles sur internet).

1.2 Frontières de l'étude et prolongements possibles

En fonction de l'intérêt des élèves, il est possible d'aller plus ou moins loin dans la compréhension du paramétrage des filtres. Outre la détection de contours, certains effets spectaculaires peuvent être facilement obtenus, comme l'embossage (en jouant sur les différentes directions possibles).

Par ailleurs, d'autres algorithmes de traitement faisant intervenir les pixels voisins du pixel traité peuvent également faire l'objet de mini-projets : la pixellisation d'une image par exemple, ou bien le tramage d'une image noir et blanc afin de lui donner un aspect « niveaux de gris », que l'on peut réaliser à l'aide d'un algorithme de diffusion d'erreur.

2 / Objectifs pédagogiques

2.1 Disciplines impliquées

Plusieurs disciplines sont concernées par l'image, mais le lien avec la Physique est tout à fait incontournable pour tout ce qui concerne la lumière. Des situations d'apprentissage en interdisciplinarité sont alors tout à fait envisageables. La détection de contours est d'un intérêt particulier en SVT par exemple (exploitation d'images satellitaires entre autres applications).

2.2 Prérequis

Essentiellement la représentation binaire de l'information et la numérisation. Voir la ressource « Algorithmes pour le traitement d'images numériques - 1 » qui précède logiquement celle-ci.

2.3 Éléments du programme

Contenus

- Représentation de l'information : numérisation.
- Algorithmes simples, programmation.

Compétences et capacités

Décrire et expliquer une situation, un système ou un programme :

- Comprendre un algorithme et expliquer ce qu'il fait.
- Modifier un algorithme existant pour obtenir un résultat différent.

Concevoir et réaliser une solution informatique en réponse à un problème :

- Concevoir, programmer un algorithme.
- Modifier format, taille, contraste, ou luminance d'images numériques.
- Filtrer et détecter des informations spécifiques.

Collaborer efficacement au sein d'une équipe dans le cadre d'un projet :

- Concevoir des programmes en autonomie.
- Gérer les étapes de l'avancement du projet en dialogue et en interaction avec le professeur.

3 / Modalités de mise en œuvre

3.1 Durée prévue pour la partie se déroulant en classe

2 séances au minimum semblent nécessaires.

3.2 Type de l'animation

Mini-projet à encadrer, de la remise d'un petit cahier des charges à la réalisation et la présentation orale.

3.3 Projet

Les travaux proposés dans ce document correspondent davantage à des mini-projets possibles qu'à de simples activités d'initiation au traitement d'images numériques.

3.4 Recherches documentaires

Les filtres de type « matrice de convolution » étant bien documentés et illustrés sur internet, on peut faire faire des recherches pour aider les élèves ayant des difficultés à visualiser le principe de l'algorithme. Ces recherches permettront également d'ajuster les coefficients du filtre pour obtenir d'autres effets (embossage par exemple).

3.5 Production des élèves

Algorithmes et programmes à concevoir ou à compléter dans le cadre d'un mini-projet à rendre et à présenter oralement.

3.6 Évaluation

Évaluation du mini-projet en distinguant la production et la présentation orale (sur le modèle des TPE par exemple).

4 / Outils

Il convient de choisir un langage de programmation simple, et qui permette de travailler aisément sur des images numériques. Python 2.7 associé à la bibliothèque de traitement d'images PIL est un bon exemple (voir les références à la fin du document).

5 / Auteur

François Passebon, professeur de Sciences Physiques, académie de Nantes

Algorithmes pour le traitement d'images - 2

1 / Réalisation de filtres

Il s'agit de donner les éléments nécessaires à la résolution du problème en fonction du niveau que l'on vise. Cela peut par exemple s'intégrer dans le cahier des charges d'un mini-projet, mais il n'est pas question de faire un cours spécifique.

Le type de filtre envisagé ici est un filtre à matrice de convolution, mais aucune connaissance sur les matrices n'est nécessaire, encore moins sur la convolution. Si on ne se laisse pas impressionner par l'intitulé, le principe est facile à comprendre sur un exemple.

Prenons le cas d'une matrice 3x3, donc un tableau de neuf nombres représentés ici par a, b, c, ..., i qui sont les paramètres du filtre et que l'on peut donc changer.

On veut appliquer ce filtre à une image en niveaux de gris (c'est plus simple), qui peut être obtenue facilement dans GIMP à partir d'une image couleur. On peut intégrer le passage de la couleur aux niveaux de gris dans le programme que l'on souhaite élaborer.

Les composantes rouge, verte, et bleue (RVB) d'un pixel quelconque de l'image sont donc identiques. Le pixel entouré en rouge, ci-dessous, correspond par exemple à R=V=B=115, c'est-à-dire à un niveau de gris égal à 115.

En quoi consiste l'application du filtre à ce pixel ?

La valeur 115 est remplacée par la somme de 9 produits

45.a + 60.b + 81.c + 82.d + 115.e + 133.f + 130.g + 154.h + 147.i

	0	1	2	3	4	5
0	48	45	60	81	83	65
1	58	82	115	133	104	55
2	99	130	154	147	96	37
3	136	160	163	138	86	39
4	156	158	157	139	89	42
5	154	154	156	145	98	45

Extrait de l'image en niveaux de gris. Lorsqu'on applique le filtre au pixel encadré en rouge, le calcul fait intervenir ce même pixel et ses 8 voisins (zone verte). La zone verte a la même taille que la matrice du filtre.

a	b	c
d	e	f
g	h	i

Matrice du filtre 3x3; les valeurs a, b, c, …, i peuvent être positives, négatives, ou nulles, entières ou non.

Qu'en est-il pour un pixel appartenant à la bordure de l'image, car il n'a pas tous ses voisins ?

La solution la plus simple est d'ignorer ces pixels : on balaye l'image de la deuxième ligne à l'avant-dernière, et pour chaque ligne, de la deuxième colonne à l'avant-dernière. Mais on peut aussi étendre l'image initiale en rajoutant une bordure d'un pixel sur laquelle on duplique les pixels voisins par exemple, ou ceux de la ligne ou colonne opposée. Pour une matrice de taille supérieure, 5x5 par exemple, le problème devient encore plus évident, c'est pourquoi il peut être intéressant d'envisager une fonction d'accès aux pixels, capable de créer un « miroir » de l'image là où tous les pixels voisins du pixel traité ne sont pas présents.

Que se passe-t-il si le résultat du calcul sort de l'intervalle [0;255] correspondant aux valeurs extrêmes des composantes RVB ?

Dans GIMP, lorsque l'option normalisée est cochée (voir annexe), le résultat, différent pour chaque pixel, est divisé par la somme des coefficients du filtre. Si cette somme est nulle, et c'est le cas des filtrages destinés à isoler les contours, le diviseur est 1, et on ajoute ensuite 128 au résultat de la division (décalage de 128), ce qui évite les valeurs négatives; c'est ainsi qu'est obtenue l'image du milieu en première page, avec la matrice ci-dessous à droite.

Autres solutions à tester (pas de diviseur ni de décalage) :

- écrêter : tout résultat supérieur à 255 est arrondi à 255, et tout résultat inférieur à 0 est arrondi à 0; c'est ainsi qu'est obtenue l'image de droite en première page, avec la matrice ci-dessous à droite là encore.
- ramener l'ensemble des résultats (obtenus pour tous les pixels) dans l'intervalle [0;255] à l'aide d'une règle de trois, ce qui nécessite de les parcourir pour en rechercher la valeur max et la valeur min; il s'agit d'une démarche de renormalisation.

Le projet consiste donc ici à programmer l'algorithme en Python, et à ajuster les paramètres du filtre pour obtenir tel ou tel résultat, avec in fine la détection des contours de l'image.

0	0	0
0	1	0
0	0	0

diviseur = Σcoef. = 1

décalage = 0

1	1	1
1	1	1
1	1	1

diviseur = Σcoef. = 9

décalage = 0

-1	-1	-1
-1	8	-1
-1	-1	-1

diviseur = 1 car Σcoef. = 0

décalage =128 donc

Le premier filtre donne sans surprise l'image de départ. Le coefficient central a-t-il une influence ?

Le second donne un flou : la valeur d'un pixel est moyennée avec les pixels environnants. On peut changer le coefficient central et voir l'effet sur le résultat.

À partir de là, on constate que le troisième est la différence entre le premier et le second (en ajustant le coefficient central à 9 pour le premier). Il permet la détection des contours. En conclusion, il vient la « formule » :

contours = | image initiale – flou |

On inverse pour finir tous les pixels (R=255-R, V=R, B=R) (il est aussi intéressant d'essayer sans inversion).

D'autres matrices sont envisageables. On peut aussi (pourquoi pas) augmenter la taille de la matrice (passer de 3×3 à 5×5 ...), en pensant au traitement spécifique des pixels n'ayant pas tous leurs voisins; voir remarque plus haut.

-1	-1	-1
-1	9	-1
-1	-1	-1

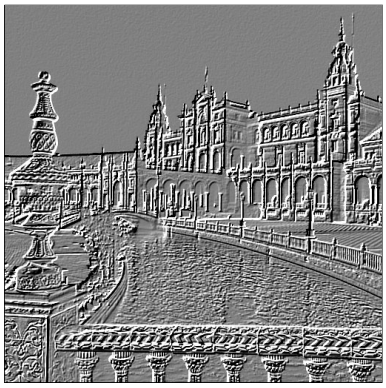
filtre réhausseur de contraste

-1/6	-2/3	-1/6
-2/3	13/3	-2/3
-1/6	-2/3	-1/6

autre filtre réhausseur

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

filtre d'embossage (effet de relief)



Les deux autres propositions de mini-projets (pixellisation d'une image, et tramage d'une image par diffusion d'erreur) sont détaillés en annexe.

2 / Références

2.1 Sitographie

Python 2.7 : http://www.python.org/download

Ressources Python : http://pythonfacile.free.fr/python/ressources.html

Opérateurs de base en Python (en anglais) :
http://www.tutorialspoint.com/python/python_basic_operators.htm

Python Imaging Library : http://www.pythonware.com/products/pil

Tutoriel PIL : http://www.pythonware.com/library/pil/handbook/image.htm

Filtres « matrice de convolution » dans GIMP : http://docs.gimp.org/fr/plug-in-convmatrix.html

Fundamentals of Image Processing : http://studenten.tudelft.nl/en/

(pdf d'une centaine de pages : rechercher « Fundamentals of Image Processing »)

Autres site sur les formats d'image : http://www.martinreddy.net/gfx

Traitement d'images : http://www.efg2.com/Lab/Library/ImageProcessing

ou http://www.ensta-paristech.fr/~manzaner/Support_Cours.html

2.2 Bibliographie

DUPRÉ X. *Programmation avec le langage PYTHON.* Éditions Ellipses, 2011

LAMBERT K., OSBORNE M. *Fundamentals of Python : first programs*. Cengage Learning, 2011

Les exemples ayant trait à l'image présentés dans cet ouvrage n'utilisent pas PIL, mais une bibliothèque créée par l'auteur. L'ouvrage est néanmoins très pédagogique quant à la présentation d'ensemble de Python.

FOLEY & VAN DAM. *Computer Graphics : Principles and Practice*. Addison-Wesley. Nouvelle édition (1995)

De BERG et.al. Computational Geometry : *Algorithms & Applications.* Springer (2008)

A. MARION. *Introduction aux techniques de traitement d'images.* Eyrolles, Paris, 1987. Nouvelle édition 1998

M. COSTER & J.L. CHERMANT. *Précis d'Analyse d'Images.* Ed. du CNRS, Paris

M. LAUG. *Traitement Optique du Signal et des Images.* Ed. Cepadues, Toulouse

A.K. JAIN. *Fundamentals of Digital Image* Processing. Prentice Hall, 1989

J.M. CHASSERY & A. MONTANVERT. *Géométrie Discrète en Analyse d'Images*. Hermes, Paris, 1991

R. HORAUD & O. MONGA. Vision par Ordinateur : *Outils Fondamentaux.* Hermes, Paris, 1993

M. KUNT, G. GRANLUND & M. KOCHER. *Traitement numérique des Images.* P.P.U.R, Lausanne, 1993

J.P. COCQUEREZ & S. PHILIPP. Analyse d'Images : *Filtrage et Segmentation.* Masson Ed., Paris, 1995

A. BOVIK. *Handbook of Image and Video Processing.* Academic Press, San Diego, 2000

2.3 Crédits photographiques

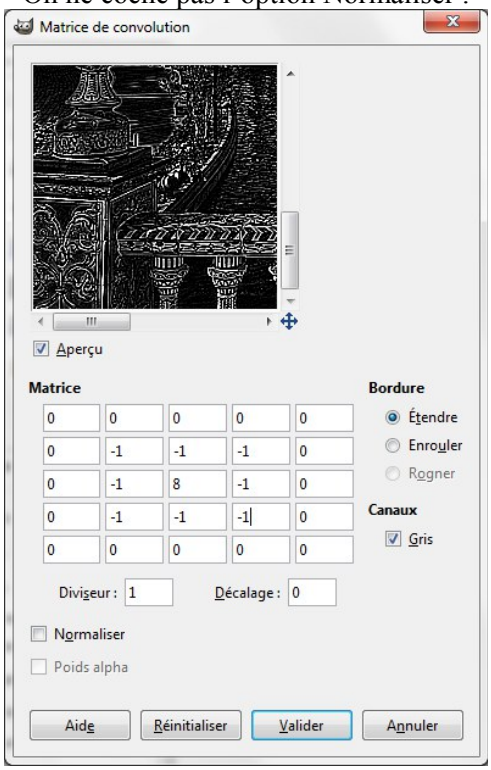
Les photographies incluses dans ce document sont la propriété de l'auteur.

Annexe – Exemples de mini-projets

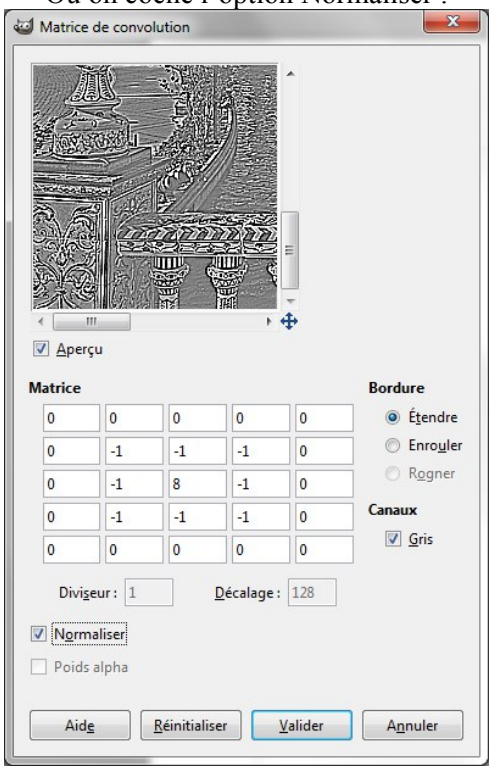
1 / Application d'un filtre permettant la détection de contours dans GIMP

L'image couleur de départ est chargée dans le logiciel.
Obtention d'une image en niveaux de gris : menu Image > Mode > Niveaux de gris (ou Menu Couleurs > Désaturer pour conserver les trois composantes afin de pouvoir les retravailler séparément).
Détection des contours : menu Filtres > Génériques > Matrice de convolution

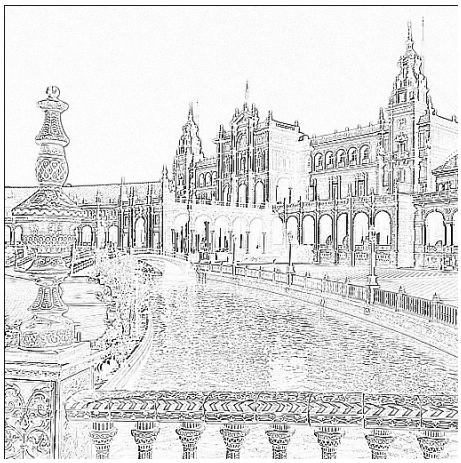
On ne coche pas l'option Normaliser :



Ou on coche l'option Normaliser :



Inversion de l'image : menu Couleurs > Inverser. Résultats obtenus :



Exemple commenté

Il s'agit d'une source en langage Python permettant d'obtenir l'image de droite ci-dessus.
Pour obtenir celle de gauche, il suffit de remplacer les deux lignes de code sur fond jaune par :

```
if r < 0: r = 0
if r > 255: r = 255
```

```
from PIL import Image                       # la bibliothèque PIL doit avoir été installée
im1 = Image.open("T:\Seville.png")          # ouverture du fichier image situé ici sous T:\ (image 512x512 pixels)
im2 = Image.new("RGB",(512,512))            # image destination pour les niveaux de gris (« vide » pour l'instant)
im3 = Image.new("RGB",(512,512))            # image destination pour les contours

for y in range(512):                        # on balaie toutes les lignes, de 0 à 511
    for x in range(512):                    # pour chaque ligne, on balaie toutes les colonnes
       p = im1.getpixel((x,y))              # en stockant le pixel (x,y) dans p, p[0] représente la composante
       r = (p[0]+p[1]+p[2])/3               rouge
       v = r                                # p[1] la composante verte, et p[2] la composante bleue
       b = r                                # on passe de la couleur aux niveaux de gris : r = int((r + v + b) / 3)
       im2.putpixel((x,y),(r,v,b))          # en niveaux de gris, les composantes RVB sont toutes égales
                                            # on écrit le pixel modifié sur l'image 2 où l'on va détecter les
for y in range(1,511):                      contours
    print y
    for x in range(1,511):                  # on balaie les lignes de la deuxième (1) à l'avant-dernière (510)
                                            # le traitement étant un peu long en Python, on affiche la ligne traitée
       pix0 = im2.getpixel((x,y))           # on balaie les colonnes de la deuxième (1) à l'avant-dernière (510)
       pix1 = im2.getpixel((x-1,y-1))       # on a besoin du pixel courant et de ses 8 voisins
       pix2 = im2.getpixel((x,y-1))         # stockage du pixel courant
       pix3 = im2.getpixel((x+1,y-1))       # de son voisin en haut à gauche
       pix4 = im2.getpixel((x-1,y))         # de celui du dessus
       pix5 = im2.getpixel((x+1,y))         # d'en haut à droite
       pix6 = im2.getpixel((x-1,y+1))       # de gauche
       pix7 = im2.getpixel((x,y+1))         # de droite
       pix8 = im2.getpixel((x+1,y+1))       # d'en bas à gauche
       r = 8*pix0[0]-pix1[0]-pix2[0]-       # d'en-dessous
pix3[0]-pix4[0]-pix5[0]-pix6[0]-            # d'en bas à droite
pix7[0]-pix8[0]                             # on applique le filtre sur la composante rouge du pixel courant,
       r = r/1                              # sachant que les verte et bleue lui sont égales
       r = r+128
       r = 255-r                            # la somme des coefficients du filtre étant nulle ici, le diviseur vaut 1
       v = r                                # pour la même raison, on décale de 128
       b = r                                # on réalise une inversion (« négatif »)
       im3.putpixel((x,y),(r,v,b))          # on est en niveaux de gris, donc R=V=B

im3.save("T:\Seville_contours.png")         # on écrit le nouveau pixel sur l'image de destination
im3.show()
                                            # on stocke l'image résultante (les contours), ici sous T:\
                                            # on montre l'image (il faut définir un programme par défaut)
```

2 / Pixellisation d'une image

La pixellisation est souvent utilisée pour flouter une image, en partie ou dans sa totalité. La zone pixelisée fait apparaître des blocs carrés contenant $n \times n$ pixels de couleur uniforme. On observe la même chose en agrandissant une image jusqu'au niveau du pixel, sauf que là, chaque carré représente 1 pixel.

image originale

pixellisation avec n = 4

pixellisation avec n = 8


pixellisation avec n = 16

Pour arriver à ce résultat, le principe de l'algorithme est donc de diviser l'image en blocs carrés de largeur n pixels. Les composantes R, V, et B de chacun des n^2 pixels d'un bloc sont remplacés par leur moyenne (une moyenne de n^2 valeurs par composante).

La largeur L et la hauteur H de l'image initiale doivent être divisibles par n (pour éviter les effets de bords, à moins de faire des traitements spécifiques sur les bords). On parcourt cette image non pas pixel par pixel mais par blocs de $n \times n$ pixels. Il y en a L/n en largeur et H/n en hauteur :

```
for b in range(H//n):                    # // correspond à la division entière en Python
     for a in range(L//n):
     ............
```

À l'intérieur de ces deux boucles, il faut à nouveau deux autres boucles pour balayer chaque bloc de n x n pixels, pixel par pixel :

```
for d in range(n):
     for c in range(n):
          p = im1.getpixel((a*n+c , b*n+d))
     .......
```

Pour le calcul des moyennes des composantes R, V, et B, on peut utiliser 3 variables moyR, moyV, et moyB, qu'on initialise à 0 avant ces deux boucles. On calcule la somme au fur et à mesure, et à la fin des deux boucles, on divise par $n \times n$ pour obtenir la moyenne (puisqu'il y a n^2 pixels par bloc), avant d'arrondir à l'entier le plus proche.

Rappel : les pixels sont numérotés à partir de 0 en largeur comme en hauteur. En Python, dans une boucle « `for p in range(q):` » p prend successivement toutes les valeur entières de 0 à q-1 soit q valeurs.

Exemple avec des niveaux de gris :

	0	1	2	3
0	172	127	103	109
1	134	85	65	67
2	120	78	56	41
3	120	61	49	40

extrait (16 pixels) d'une image en niveaux de gris

Traitement d'un premier bloc 2x2 (en vert) : on prend la moyenne des valeurs : (172 + 127 + 134 + 85)/4 = 130
On donne cette valeur aux 4 pixels de l'image initiale.
On passe ensuite au second bloc (en rouge) et on opère de même.
Les 4 pixels du bloc rouge prennent la valeur de la moyenne (86).
De même pour les blocs bleus et jaunes (dont les pixels prennent respectivement la valeur 95 et 47).
On obtient ainsi la nouvelle image ci-contre.

	0	1	2	3
0	130	130	86	86
1	130	130	86	86
2	95	95	47	47
3	95	95	47	47

Remarque : On peut adapter l'algorithme pour réduire la taille d'une image d'un facteur n en largeur comme en hauteur.

3 / Image noir et blanc ou couleur tramée - algorithme de tramage par diffusion d'erreur

Le principe est de produire une image en noir et blanc mais où l'erreur de quantification(1) est partiellement distribuée sur les pixels voisins non encore traités. L'avantage est que l'image obtenue ressemble à une image en niveaux de gris alors que chaque pixel est soit blanc (255) soit noir (0).

Il s'agit d'implémenter l'algorithme suivant, à partir d'une image en niveau de gris. Comme pour passer des niveaux de gris en noir et blanc, on se fixe un seuil, par exemple 128. Si le niveau est supérieur au seuil, le pixel devient blanc (255), sinon il devient noir (0).

	0	1	2	3
0	172	127	103	109
1	134	85	65	67
2	120	78	56	41
3	120	61	49	40
4	105	51	39	38

extrait (20 pixels) d'une image en niveaux de gris

Traitement du premier pixel : 172 > 128, donc le pixel devient blanc (255).
L'erreur de quantification vaut 172-255 = -83.
On reporte la moitié de cette erreur sur le pixel immédiatement à droite, qui prend donc la valeur 127+(-83/2)= 86 (on arrondit le résultat).
On reporte le quart de l'erreur sur le pixel en dessous : 134+(-83/4) = 113
et sur celui en dessous à droite : 85+(-83/4) = 64.
On traite le pixel suivant en prenant en compte les modifications : il vaut maintenant 86 et non 127. Etc.

(1) notion expliquée sur un exemple dans la suite du texte.

Il faut tenir compte du fait que les pixels de la dernière colonne de l'image n'ont pas de voisins de droite et que ceux de la dernière ligne n'ont pas de voisins de dessous. Ils sont néanmoins modifiés, notamment par le traitement de la colonne ou de la ligne précédente.

On peut envisager cet algorithme sur une image en couleur en l'appliquant successivement aux composantes R, V, B pour chaque pixel. On obtient alors une image dont les pixels ne peuvent prendre que 8 couleurs :

R	0	0	0	0	255	255	255	255
V	0	0	255	255	0	0	255	255
B	0	255	0	255	0	255	0	255

mais dont le tramage lui donne l'apparence d'une image comportant davantage de couleurs.

L'algorithme peut se résumer à l'aide de la notation suivante, où x représente le pixel traité : $\begin{pmatrix} x & 1/2 \\ 1/4 & 1/4 \end{pmatrix}$

Un traitement plus compliqué (l'algorithme de Floyd-Steinberg) est décrit par : $\begin{pmatrix} & x & 7/16 \\ 3/16 & 5/16 & 1/16 \end{pmatrix}$

Image noir et blanc « classique »

Image tramée en noir et blanc
obtenue par diffusion d'erreurs

Image tramée en 8 couleurs
obtenue par diffusion d'erreurs →