Université de Toulouse III, FSI – Département de Mathématiques
Licence 2, module "Calcul scientifique", 2019–2020

Projet: "Traitement d'images, détection de contours"

Février 2020

Détecter les contours d'une image constitue une étape préliminaire à de nombreuses applications pratiques en traitement d'images : analyse d'images médicales, reconnaissance de formes, restauration... Le but de ce projet est d'aborder quelques méthodes simples de détection de contours.

1 Fichiers fournis et installation

Avant tout, il faut installer la bibliothèque `image` de Python grâce à cette commande :

```
pip install image --user
```

L'ensemble des fichiers de travail peut être téléchargé à cette adresse :

https://www.math.univ-toulouse.fr/~vfeuvrie/l2spe/image.zip

Vous trouverez dans cette archive la bibliothèque `imtools` (voir par exemple le listing 1) qui simplifie les tâches courantes de gestion des images, ainsi que quelques images pour vos expériences numériques. Pour les utiliser, une fois `image` installée comme expliqué plus haut, il suffit de recopier tous les fichiers dans votre répertoire de travail.

2 Images digitales

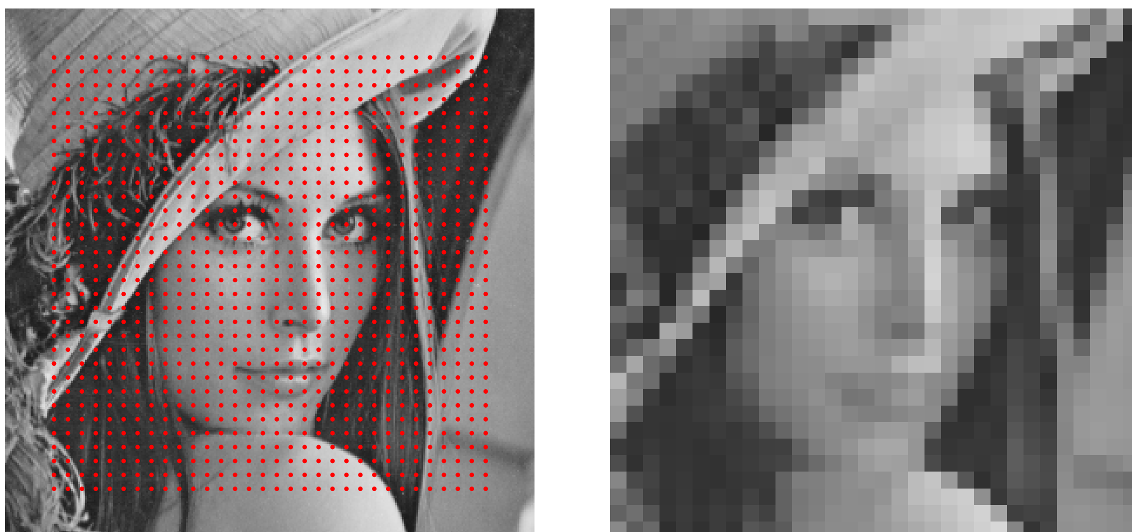

FIGURE 1 – Échantillonnage d'une image sur une grille de taille 32×32.

Une manière de définir une *image digitale* u de taille $M \times N$ est de la voir comme une matrice

$$\Big(u_{m,n}\Big)_{\substack{0 \leq m < M \\ 0 \leq n < N}} \in \mathcal{M}_{M,N}(\mathbb{R}). \tag{1}$$

On notera qu'à la différence des conventions habituelles en mathématiques, on a numéroté les lignes et les colonnes à partir de zéro, afin de respecter l'indexation des tableaux `numpy` de Python. Pour simplifier, on ne considérera que des images en niveaux de gris. Chaque case de la matrice est un *pixel* (abréviation de *picture element*). Habituellement une image digitale prend des valeurs comprises entre zéro (pixel noir) et 255 (pixel blanc), même si d'autres conventions sont possibles.

Une image digitale correspond à l'échantillonnage d'une *image analogique* f, vue comme une application $f\colon \mathbb{R}^2 \to \mathbb{R}$ représentant le flux lumineux théorique reçu au niveau d'un plan, en fonction de la position sur ce plan. Un récepteur placé derrière ce plan ne perçoit qu'une portion bornée du flux, sur le pavé $[a, b] \times [c, d]$. Sur ce pavé on construit une grille de points (x_m, y_m) uniformément répartis (pour simplifier, on va supposer que la grille d'échantillonnage utilisée a un pas égal à 1), et on considère que l'image échantillonnée est obtenue tout simplement en posant (voir figure 1) :

$$u_{m,n} = f(x_m, y_n). \tag{2}$$

Dans les calculs qu'on sera amené à traiter, on fera parfois référence à des pixels d'indices situés à l'extérieur du pavé $\{0, \ldots, M-1\} \times \{0, \ldots, N-1\}$. Dans ce cas, plusieurs conventions sont possibles (voir figure 2) :

- considérer que l'image est constante (par exemple égale à zéro) à l'extérieur de la zone d'échantillonnage ;
- prolonger l'image en attribuant la même valeur que le pixel du bord situé à distance minimale du point considéré ;
- prolonger l'image par parité et périodicité ;
- prolonger l'image par périodicité, en posant $u_{m,n} = u_{m',n'}$ où m' est le reste de la division euclidienne de m par M et n' celui de la division euclidienne de n par N.

On préférera l'utilisation de cette dernière convention, car elle est facile à implémenter en Python : il suffit par exemple d'écrire `u[m % M, u % N]` pour effectuer cette périodisation.

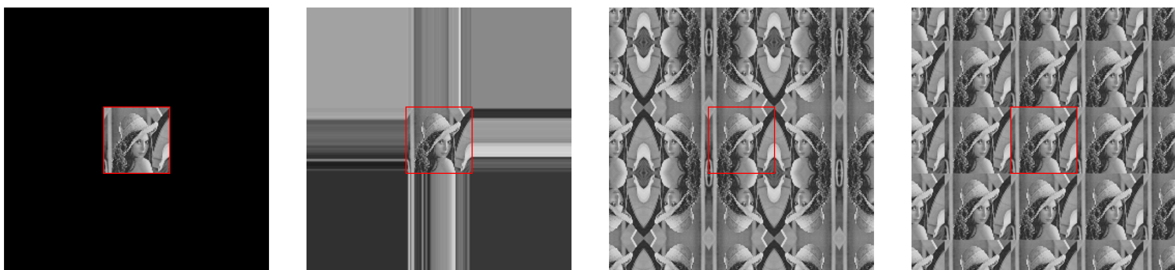

FIGURE 2 – Prolongements possibles d'une image hors de son support (en rouge).

On fournit une petite bibliothèque `imtools.py` qui introduit quelques fonctions permettant de manipuler des images en Python :

- `open_image(filename)` permet d'ouvrir un fichier image en niveaux de gris (habituellement au format PNG) et d'obtenir les données sous forme d'un tableau `numpy` à deux dimensions ;

- `save_image(u, filename)` permet d'enregistrer un tableau `numpy` dans un fichier PNG. Les données sont converties en nombres entiers compris entre 0 et 255 : il peut donc être nécessaire de les ajuster avant d'enregistrer (si par exemple on manipule des tableaux avec des valeurs négatives) ;

- `display_image(u)` affiche une image à l'écran. On notera que par défaut, Python va automatiquement interpréter les données de manière à ce que la valeur la plus basse du tableau corresponde à du noir et la valeur la plus haute à du blanc. Ce n'est pas toujours souhaitable. On pourra, si nécessaire, utiliser par exemple

  ```
  display_image(u, vmin = 0, vmax = 255)
  ```

  pour ajuster l'échelle (ici : le noir à zéro et le blanc à 255). Il peut être également nécessaire d'ajuster la taille d'affichage, car par défaut les notebook affichent les figures avec une petite taille, ce qui empêche de percevoir les détails. On pourra utiliser par exemple :

  ```
  plt.figure(figsize = (10, 10))
  ```

  avant l'utilisation de `display_image` pour augmenter la taille d'affichage.

- `display_images(l)` affiche une liste d'images en une seule fois.

3 Produit de convolution

3.1 Filtrage des images

Si u et h sont deux images de même taille $M \times N$, on définit le produit de convolution $u * h$ par

$$(u * h)_{m,n} = \sum_{\substack{0 \leq k < M \\ 0 \leq l < N}} u_{m-k,n-l} h_{k,l}. \tag{3}$$

On peut vérifier que le produit de convolution ainsi défini est bilinéaire, commutatif, associatif, et que l'élément neutre est la masse de Dirac $\delta_{0,0}$ définie par

$$\delta_{0,0} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix}. \tag{4}$$

De manière similaire, on notera $\delta_{m,n}$ la masse de Dirac en (m, n), qui vaut zéro partout sauf à l'indice (m, n) où elle vaut 1.

Mathématiquement, u et h jouent des rôles symétriques dans la définition (3). Toutefois dans le contexte du traitement d'image, on considère souvent que l'un des opérandes (noté en général u) est une image tandis que l'autre (noté en général h) est un *filtre*. Avec ces conventions, le calcul de $u * h$ est appelé un *filtrage* de l'image.

3.2 Calcul en pratique lorsque le support du filtre est petit

Lorsque h a peu de coefficients non nuls, la façon la plus efficace de calculer le produit de convolution $u * h$ en Python repose sur l'observation suivante. Si on note $\tau_{k,l}$ l'opérateur linéaire de décalage (avec la convention de périodisation des indices comme expliqué précédemment)

$$\tau_{k,l} : \begin{cases} \mathcal{M}_{M,N}(\mathbb{R}) \to \mathcal{M}_{M,N}(\mathbb{R}) \\ (u_{m,n}) \mapsto (u_{m-k,n-l}), \end{cases} \tag{5}$$

alors on peut écrire :

$$u * h = \sum_{(k,l) \in \operatorname{Spt} h} h_{k,l} \tau_{k,l}(u). \tag{6}$$

Ici, $\operatorname{Spt} h$ désigne le *support* de h, c'est à dire l'ensemble des indices (k, l) tels que $h_{k,l} \neq 0$.

La bibliothèque `imtools` fournit une fonction pour calculer $\tau_{k,l}(u)$: il suffit d'appeler

```
shift_image(u, k, l)
```

pour obtenir le tableau correspondant. Cette fonction est beaucoup plus rapide que le calcul direct avec des boucles `for`, notoirement lentes en Python.

Pour simplifier les notations, lorsqu'un filtre a un support petit, on se contentera de le noter sous la forme d'un tableau réduit à la taille de son support. Pour lever d'éventuelles ambiguïtés, on encadrera le coefficient situé à l'indice $(0, 0)$. Ainsi, le filtre $\delta_{0,0}$ peut se noter sous forme abrégée :

$$\delta_{0,0} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & \boxed{1} & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} \boxed{1} \end{pmatrix}. \tag{7}$$

Le listing 1 donne un exemple d'implémentation efficace en Python du filtre

$$h = -3\delta_{1,-1} + 5\delta_{0,0} - 2\delta_{-1,1} = \begin{pmatrix} 0 & 0 & -2 \\ 0 & \boxed{5} & 0 \\ -3 & 0 & 0 \end{pmatrix}. \tag{8}$$

```python
from imtools import *

def filtrage(u):
    return -3 * shift_image(u, 1, -1) + 5 * u - 2 * shift_image(u, -1, 1)

u = open_image('lena.png')
v = filtrage(u)
plt.figure(figsize = (10, 5))
display_images([u, v])
```

Listing 1 – Implémentation efficace du filtre $-3\delta_{1,-1} + 5\delta_{0,0} - 2\delta_{-1,1}$

4 Discrétisation de quelques opérateurs différentiels

La plupart des méthodes de détection de contours sont basées sur l'étude des dérivées de l'image. Mais il reste à définir ce qu'on entend par là, puisque les images digitales sont des objets de nature discrète. Prenons l'exemple de la première dérivée selon la première variable. Si on regarde l'équation (2), en supposant que f est suffisamment régulière, il s'agit donc de $\frac{\partial f}{\partial x}$. On se pose alors la question d'estimer l'échantillonnage de la dérivée $\frac{\partial f}{\partial x}(x_m, y_n)$ connaissant seulement l'échantillonnage u de f.

4.1 Schémas de différences finies d'ordre 1

Plusieurs schémas numériques ont été proposés pour résoudre ce problème. L'une des possibilités consiste à supposer que sur un voisinage U de (x_k, y_k) suffisamment grand, l'image analogique f est affine :

$$\forall (x,y) \in U\colon f(x,y) = f(x_m, y_n) + \alpha(x - x_m) + \beta(y - y_n). \tag{9}$$

Dans ce cas, en supposant que U contient les pixels voisins, on a :

$$\frac{\partial f}{\partial x}(x_m, y_n) = \alpha = f(x_m + 1, y_n) - f(x_m, y_n) = u_{m+1,n} - u_{m,n} \tag{10}$$

$$\frac{\partial f}{\partial y}(x_m, y_n) = \beta = f(x_m, y_n + 1) - f(x_m, y_n) = u_{m,n+1} - u_{m,n}, \tag{11}$$

et dans ces conditions le calcul des dérivées de u peut s'obtenir en effectuant le produit de convolution de u respectivement par les deux filtres :

$$\delta_{-1,0} - \delta_{0,0} = \begin{pmatrix} 1 & \boxed{-1} \end{pmatrix} \qquad \delta_{0,-1} - \delta_{0,0} = \begin{pmatrix} 1 \\ \boxed{-1} \end{pmatrix}. \tag{12}$$

On parle alors de *dérivée par différences finies décentrées à droite*. Mais d'autres choix sont possibles, comme par exemple :

$$\delta_{0,0} - \delta_{1,0} = \begin{pmatrix} \boxed{1} & -1 \end{pmatrix} \qquad \delta_{0,0} - \delta_{0,1} = \begin{pmatrix} \boxed{1} \\ -1 \end{pmatrix}. \tag{13}$$

On parle cette fois de dérivées décentrées à gauche. On peut également obtenir des dérivées centrées en considérant la moyenne des deux versions :

$$\frac{\partial}{\partial x}\colon \frac{1}{2}(\delta_{-1,0} - \delta_{1,0}) = \frac{1}{2}\begin{pmatrix} 1 & \boxed{0} & -1 \end{pmatrix} \qquad \frac{\partial}{\partial y}\colon \frac{1}{2}(\delta_{0,-1} - \delta_{0,1}) = \frac{1}{2}\begin{pmatrix} 1 \\ \boxed{0} \\ -1 \end{pmatrix}. \tag{14}$$

4.2 Schémas de différences finies d'ordre 2

Il est possible d'itérer les schémas d'ordre 1 pour calculer les dérivées d'ordre supérieur. Étant donnée l'associativité du produit de convolution, le filtre correspondant à une double différentiation est obtenu en convoluant les deux filtres de différentiation. Voici ce qu'on obtient à l'ordre 2 :

$$\frac{\partial^2}{\partial x^2}\colon (\delta_{-1,0} - \delta_{0,0}) * (\delta_{0,0} - \delta_{1,0}) = \begin{pmatrix} 1 & \boxed{-2} & 1 \end{pmatrix} \tag{15}$$

$$\frac{\partial^2}{\partial y^2}\colon (\delta_{0,-1} - \delta_{0,0}) * (\delta_{0,0} - \delta_{0,1}) = \begin{pmatrix} 1 \\ \boxed{-2} \\ 1 \end{pmatrix} \tag{16}$$

$$\frac{\partial^2}{\partial x \partial y}\colon \frac{1}{4}(\delta_{0,-1} - \delta_{0,1}) * (\delta_{-1,0} - \delta_{1,0}) = \frac{1}{4}\begin{pmatrix} 1 & 0 & -1 \\ 0 & \boxed{0} & 0 \\ -1 & 0 & 1 \end{pmatrix}. \tag{17}$$

Un cas particulier est celui de l'opérateur laplacien. En faisant comme précédemment une moyenne entre différents décalages possibles, on peut donner cette version qui a un comportement un peu plus anisotrope que la simple addition des filtres de $\frac{\partial^2}{\partial x^2}$ et $\frac{\partial^2}{\partial y^2}$:

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\colon \frac{1}{8}\begin{pmatrix} 1 & 1 & 1 \\ 1 & \boxed{-8} & 1 \\ 1 & 1 & 1 \end{pmatrix}. \tag{18}$$

On en trouvera un exemple d'implémentation dans le listing 2

```
def laplacian(u):
    v = 0.0 * u
    for x in range(-1, 2):
        for y in range(-1, 2):
            if x == 0 and y == 0:
              r = -1
            else:
              r = 1 / 8
            v += r * shift_image(u, x, y)
    return v
```

Listing 2 – Calcul du laplacien

4.3 Sensibilité au bruit

En pratique, l'échantillonnage décrit dans l'égalité (2) fait souvent intervenir un bruit, qui peut être dû à différents facteurs (imprécision des capteurs, mouvement lors de la mesure, etc...). Pour simplifier, on supposera que le bruit est additif et décrit par une famille de variables aléatoires $b_{m,n}$ indépendantes, gaussiennes, centrées de lois $\mathcal{N}(0, \sigma^2)$. La formule d'échantillonnage devient alors

$$u_{m,n} = f(x_m, y_n) + b_{m,n}, \tag{19}$$

et en pratique on considère que l'image obtenue correspond à une réalisation des variables $b_{m,n}$. On pourra simuler l'ajout d'un bruit à une image avec la fonction donnée dans le listing 3.

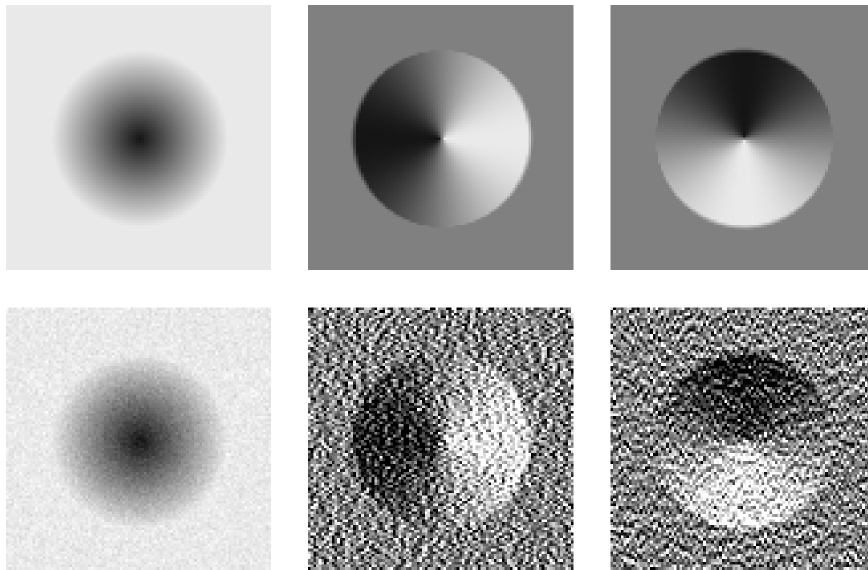

FIGURE 3 – Effet de l'ajout d'un bruit sur le calcul des dérivées de l'image

```
def noise(u, sigma):
    return u + np.random.normal(0, sigma, size = u.shape)

u = open_image('peppers.png')
v = noise(u, 30)
plt.figure(figsize = (15, 10))
display_images([u, v]
```

Listing 3 – Exemple de simulation du bruitage d'une image

Comme on peut le constater sur la figure 3, les opérateurs de dérivation sont très fortement sensibles au bruit, ce qui va nuire à la détection des contours. Il est donc préférable d'effectuer un traitement préalable de l'image pour réduire le bruit.

4.4 Réduction du bruit

Une méthode classique de réduction du bruit en traitement d'images est l'utilisation de l'EDP (équation aux dérivées partielles) dite de la chaleur :

$$\begin{cases} \dfrac{\partial v}{\partial t} - \Delta v = 0 \\ v(0, x) = f(x). \end{cases} \tag{20}$$

Ici $v \colon \mathbb{R}_+ \times \mathbb{R}^2 \to \mathbb{R}$ est une fonction des deux variables $t \geq 0$ et $x \in \mathbb{R}^2$. En partant de la condition initiale $v(0, \cdot) = f$ on fait évoluer v au cours du temps de la même manière que la chaleur se répartit au sein d'un solide homogène. Il se produit alors une régularisation de la fonction initiale f qui estompe peu à peu le bruit.

On peut montrer par exemple que sous certaines conditions, notamment de régularité de f, alors $v(t, \cdot)$ converge uniformément vers la fonction constante égale à la valeur moyenne de f lorsque $t \to \infty$. Il n'est pas question d'aller jusque là dans les cas qui nous intéressent, on se contentera d'aller jusqu'à $t = T$, où T est une constante à ajuster selon le degré d'atténuation du bruit désiré (voir figure 4). Il faut trouver le bon compromis entre le floutage des contours et la réduction du bruit.

Pour résoudre l'équation (20) avec des images digitales, on pourra utiliser le schéma numérique discret suivant :

$$\begin{cases} u(0) = u \\ u(t_{k+1}) = u(t_k) + \delta t \Delta u(t_k), \end{cases} \tag{21}$$

où Δ désigne la convolution par le laplacien discrétisé donné en (18).

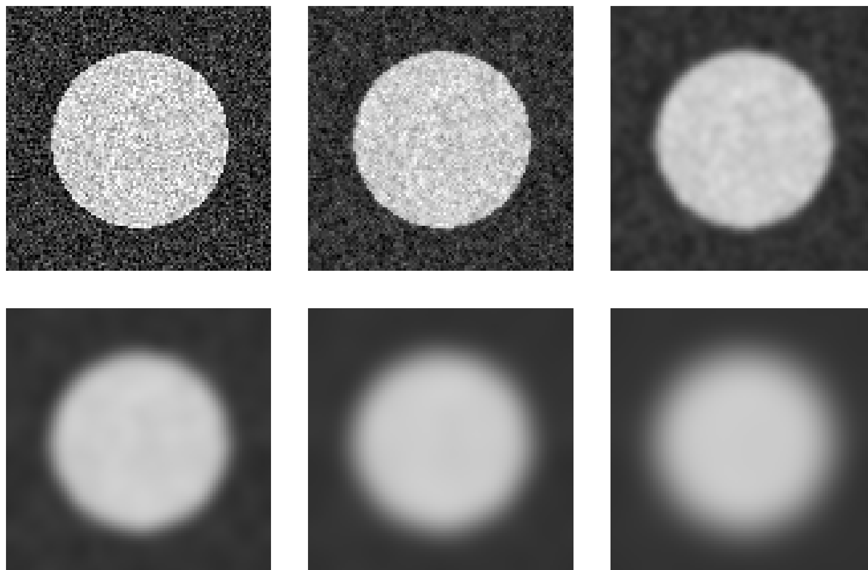

FIGURE 4 – Réduction du bruit par l'équation de la chaleur pour différentes valeurs de T

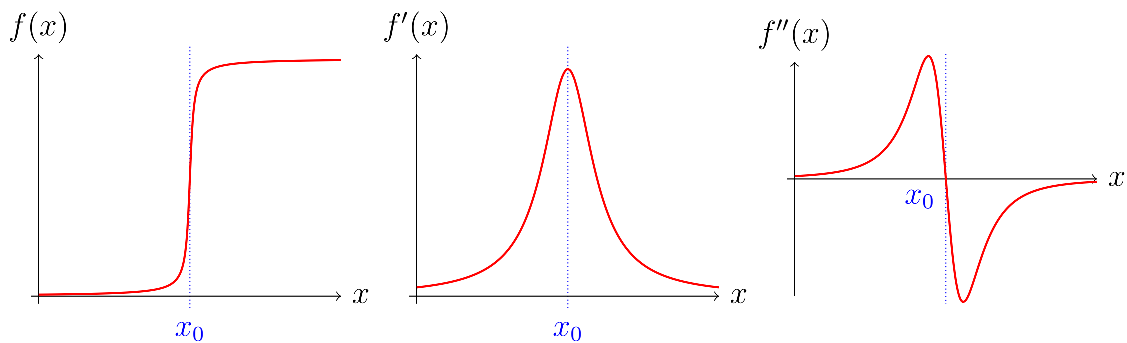

FIGURE 5 – Exemple d'un contour en dimension 1 localisé en $x = x_0$ avec le comportement des deux premières dérivées

5 Méthodes de détection des contours

Avant d'aller plus loin, il faut définir un peu plus précisément ce qu'on entend par la notion de « contour ». Intuitivement, l'œil percevra un contour là où la luminosité de l'image varie brusquement. En dimension 1, on peut considérer qu'un contour correspond à un extremum local de la dérivée première, ou encore à une annulation de la dérivée seconde (voir figure 5).

5.1 Seuillage du gradient

Pa analogie avec la dimension 1, on peut considérer que les contours d'une image sont localisés là où la norme du gradient présente un maximum local.

```
def threshold(u, s):
    return (u > s).astype(float)
```

Listing 4 – Implémentation du seuillage : la fonction renvoie un tableau contenant des zéros et des uns, qui correspondent à la fonction caractéristique de l'ensemble considéré.

FIGURE 6 – Seuillage de la norme du gradient de Lena pour différentes valeurs de s

Une première idée pour trouver leur localisation consiste à *seuiller* la norme du gradient, c'est à dire à considérer que les pixels qui font partie des contours de l'image sont ceux pour lesquels la norme du gradient est plus grande que s, où $s > 0$ est un seuil fixé arbitrairement. Formellement, on peut écrire :

$$(m, n) \text{ est sur un contour de l'image } \iff \|\nabla u_{m,n}\| > s. \tag{22}$$

Une fonction de seuillage peut se définir très simplement en Python (voir listing 4). On peut voir quelques exemples de seuillage du gradient sur la figure 6.

On rappelle qu'avant d'utiliser la méthode, il peut être nécessaire de débruiter l'image comme expliqué dans la section 4.4, car le calcul approché du gradient par différences finies est très sensible au bruit.

5.2 Annulations du laplacien

La méthode précédente est facile à implémenter, mais présente plusieurs défauts :

- il est difficile de savoir quel seuil s choisir a priori ;
- certains contours rectilignes sont mal localisés : on trouve des lignes épaisses là où on voudrait une ligne ponctuelle.

On peut améliorer les résultats en cherchant les annulations du laplacien de l'image.

Pour simplifier, on considère que le laplacien s'annule entre deux pixels voisins si les valeurs qu'il prend sur chaque pixel sont de signes opposés. Toutefois, cette méthode risque de détecter les petites oscillations locales dues au bruit, même après débruitage (qui atténue le bruit sans le faire disparaître totalement). Pour éviter cela, il faut appliquer un double seuillage du laplacien. On fixe un seuil $\epsilon > 0$ et on définit :

$$v_{m,n} = \begin{cases} +1 & \text{si } \Delta u_{m,n} > \epsilon \\ 0 & \text{si } -\epsilon \leq \Delta u_{m,n} \leq \epsilon \\ -1 & \text{si } \Delta u_{m,n} < -\epsilon. \end{cases} \tag{23}$$

Une fois que v a été calculée, il ne reste plus qu'à détecter les changements de signe comme sur la figure 7. On fournit (voir listing 5) l'implémentation d'une fonction en Python qui utilise ce procédé pour détecter les annulations d'une image.

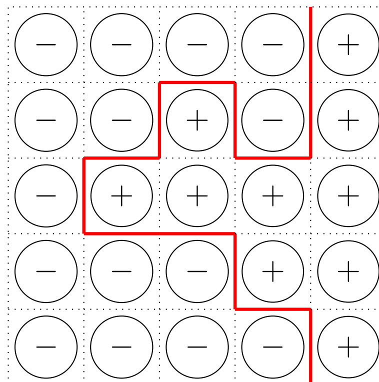

FIGURE 7 – Détection des changements de signe d'une image (en rouge).

```
def detect_vanish(u, epsilon):
    v1 = u < -epsilon
    v2 = u > epsilon
    w = np.zeros(u.shape, bool)
    for (x, y) in ((0, 1), (1, 0), (1, 1)):
        w += v1 * shift_image(v2, x, y) + v2 * shift_image(v1, x, y)
    return w.astype(float)
```

Listing 5 – Dt́ection des annulations d'une image.

5.3 Annulation de la dérivée seconde dans la direction du gradient

Il est possible d'améliorer encore la méthode précédente. On rappelle que la matrice hessienne d'une fonction réelle f de deux variables et de classe $\mathcal{C}^2$ est définie par :

$$\mathrm{Hess_f}(a,b) = \begin{pmatrix} \dfrac{\partial^2 f}{\partial x^2}(a,b) & \dfrac{\partial^2 f}{\partial x \partial y}(a,b) \\ \dfrac{\partial^2 f}{\partial x \partial y}(a,b) & \dfrac{\partial^2 f}{\partial y^2}(a,b) \end{pmatrix}. \tag{24}$$

Il s'agit de la matrice d'une forme bilinéaire (qui est la différentielle seconde de f au point (a,b) considéré). On va alors chercher les changements de signe non plus du Laplacien de l'image, mais de la dérivée seconde dans le sens du gradient, qui est donnée (dans le cas d'une fonction) par :

$$^{\top}\nabla f \, \mathrm{Hess_f} \, \nabla f. \tag{25}$$