Python et *Gimp*

Pour plus d'infos sur les objets et méthodes accessibles en *Python*, ainsi que sur le développement de plugins pour *Gimp* : `http://www.gimp.org/docs/python/index.html`.

1 Instructions en *Python* pour *Gimp*

A condition d'avoir installé correctement *Gimp* sous *Windows*[1], l'exécution de scripts est possible :

- soit via l'interface graphique (menu `Filtres / Python-fu` qui permet d'ouvrir une console), pour évaluer directement des codes en python;
- soit en créant des « plugins » qui seront chargés au démarrage de *Gimp* et exécutés à la demande via un menu.

Il est ainsi possible d'accèder à l'ensemble des images ouvertes, à leurs contenus, de les modifier et de mettre à jour leur affichage... et de leur appliquer tous les traitements existants dans le logiciel *Gimp*, comme si l'on cliquait sur le menu graphique. Ci-dessous une très brève présentation de l'API[2] :

- récupération d'une image (la première ouverte) parmi toutes les images ouvertes :

```
listeImg = gimp.image_list()  # listeImg est une liste python
img = listeImg[0]             # img est un alors objet image
```

- récupération du calque par défaut (le fond, ou arrière-plan, de l'image) :

```
fond = img.layers[0]          # fond est le calque d'indice 0 de img
```

- création et insertion d'un nouveau calque dans une image en niveau de gris :

```
neuf = gimp.Layer(img, "Mon_Filtre", img.width , img.height , GRAY_IMAGE, 100, NORMAL_MODE)
img.add_layer(neuf, 0)        # neuf est un nouveau calque en niveaux de gris ajouté à img
```

  d'autres « flags » existent pour d'autres modes : `RGB_IMAGE`, `INDEXED_IMAGE`...

- accès à la valeur d'un pixel :

```
calque.set_pixel(x, y, couleur)   # fixe la valeur du pixel de coordonnées (x, y)
couleur = calque.get_pixel(x, y)  # récupère la valeur du pixel (x, y)
```

  En RVB une couleur est un tuple (t-uplet) de 3 composantes. Un niveaux de gris est un tuple de 1 valeur :

```
couleur = tuple([128,128,128])    # du gris en RVB
couleur = tuple([128])            # du gris en niveaux de gris
```

  Ainsi, on peut accèder aux composantes de couleurs :

```
couleur = calque.get_pixel(x, y)  # renvoie un tuple décrivant la couleur
couleur[0]                        # renvoie la première valeur du tuple
```

- mise à jour de l'affichage de toute l'image :

```
calque.update(0,0,img.width,img.height)
```

De manière générale, en *Python*, un tableau de taille $2 \times n$ peut être vu comme une liste de n éléments qui sont tous des 2-uplets, comme dans l'exemple suivant :

```
une_serie_de_donnees = tuple([1, 2])
une_autre_serie_de_donnees = tuple([5, 6])
une_liste = [une_serie_de_donnees]
une_liste.append(une_autre_serie_de_donnees)
```

1. Le support de *Python* pour *Gimp* 2.6 sous *Windows* impose d'installer d'abord *Python* puis les 3 bibliothèques *Python-Cairo*, *Python-GObject* et *Python-GTK*, et enfin *The Gimp* en choisissant une « custom installation » pour activer ce support. Pour *Gimp* 2.8 l'installation est automatique, en revanche l'ajout de bibliothèque *Python* telles que *NumPy* est plus complexe - voir plus bas.
2. API (*Application Programming Interface*) : ensemble de fonctions mises à disposition pour développer des programmes.

2 Tutoriel

2.1 Recupération de la matrice image

1. Sur la plateforme `http://tice.agrocampus-ouest.fr`, récupérez l'image *tugs.png* et ouvrez-la dans *Gimp*.
2. S'il s'agit de la seule image ouverte, elle doit se trouver à l'indice 0 de la pile d'image. Par ailleurs, c'est une image en niveaux de gris qui ne comporte qu'un calque (donc lui aussi d'indice 0). Les instructions suivantes vous permettent normalement d'accéder à son contenu à l'aide d'un objet « calque » nommé ici `fond` :

```
listeImg = gimp.image_list()
img = listeImg[0]
fond = img.layers[0]
```

2.2 Colorier des pixels

Vous allez maintenant colorier les points de coordonnées (4,4) et (2,6) respectivement en blanc et en noir à l'aide des instructions suivantes :

1. construire des objets « couleurs » :

```
noir = tuple([0])
blanc = tuple([255])
```

2. fixer la couleur des pixels :

```
fond.set_pixel(4, 4, noir)
fond.set_pixel(2, 6, blanc)
```

3. mettre à jour l'affichage de toute l'image :

```
calque.update(0,0,img.width,img.height)
```

4. Vérifiez la bonne exécution en zoomant dans la partie concernée de l'image (en haut à gauche).

2.3 Parcourir le contenu de l'image

Le parcours complet d'un image s'effectue à l'aide d'une « double boucle », qui permet le parcours itératif imbriqué des deux coordonnées de l'image. Cela se fait de la manière suivante en *Python*, où « instructions » représente symboliquement ce que l'on veut faire pour chacun des pixels de l'image (attention l'indentation est très importante en *Python*, ne mélangez pas espaces et tabulations !) :

```
for x in range(0, img.width):
  for y in range(0, img.height):
    # instructions
```

Vous allez binariser l'image *tugs.png*. Pour cela, les pixels sont tous examinés un à un et comparés à un seuil, 127 par exemple. De manière très générale, le test de comparaison (avec `gris` et `bin` deux variables numériques entières), peut s'écrire de la manière suivante :

```
if gris > 127:
  bin = 255
else:
  bin = 0
```

Attention, les couleurs de l'image ne sont pas des entiers mais des tuples ! Pour obtenir la valeur numérique en niveau de gris et la modifier dans l'image :

```
gris = calque.get_pixel(x, y)  # renvoie un tuple
if gris[0] > 127:              # première et seule composante du tuple
  bin = tuple([255])
else:
  bin = tuple([0])
fond.set_pixel(x, y, bin)      # changer la valeur du pixel
```

1. Ecrivez le code à exécuter pour binariser l'image dans une éditeur de texte (*Notepad++*).
2. Vérifiez la bonne exécution en appliquant ce traitement à l'image, en effectuant un copier-coller dans la console *Python*. N'oubliez pas de demander la mise à jour de l'affichage...

2.4 Dessiner dans l'image

Pour finir, vous allez dessiner en gris un segment de droite de pente -1/3 entre les coordonnées (10,10) et (40,20) de l'image. Pour cela, à chaque « pas » sur y, 3 « pas » sur x sont couvert :

```
brosse = tuple([127])
```

```
for y in range(0, 10):
  for x in range(0, 2):
    fond.set_pixel(10+3*y+x, 10+y, brosse)
calque.update(10,10,40,20)
```

3 Créations de « plugins »

3.1 Fichier « plugin »

Un plugin est un script, stocké dans un fichier texte (extension `.py`), accessible à *Gimp* (il faut donc renseigner le chemin d'accès au plugins dans les préférences d'utilisation du logiciel via le menu *Édition / Préférences / Dossiers / Greffons*) et dont l'exécution peut être obtenue via son interface graphique. Le plugin « s'enregistre » au démarrage de *Gimp*. Ensuite, le script contenu est ré-évalué à chaque appel (inutile donc de redémarrer *Gimp* à chaque modification du code, sauf problème ou changement dans l'enregistrement aurpsè de *Gimp*).

Des exemples de plugins fonctionnels (outils de morphologie mathématique) vous sont donnés sur la plateforme TICE d'Agrocampus. Toutes les commandes présentes dans les menus de *Gimp* sont accessibles en *Python* via la *procedural database* c'est-à-dire grâce à l'objet `gimp.pdb` (voir menu *Aide / Navigateur de procédures*).

3.2 Structure du fichier « plugin »

Le fichier contenant le code a exécuter commence par une entête indiquant qu'il s'agit d'un script *Python* et par la précision sur le codage des caractères employé, par exemple ici en Iso :

```
#!/usr/bin/env python
# -*- coding: ISO-8859-15 -*-
```

Suivent alors les directives relatives aux bibliothèques *Python* employées :

```
from numpy import *   # calculs numériques et structures matricielles
from gimpfu import *  # API Gimp
```

Viennent ensuite les définition des fonctions utilisées, par exemple :

```
def trace_oblique(image, drawable):
    # mon code python
    # pour tracer une ligne
    # oblique dans l'image...
```

Puis la partie propre à l'enregistrement du plugin auprès de *Gimp* : nom de l'auteur, nom de la fonction a appeler, nom et position du menu à ajouter à l'interface graphique... (voir ci-dedssous) et enfin la dernière ligne :

```
main()
```

3.3 Enregistrement auprès de *Gimp*

la partie détaillée ici permet de faire connaitre le plugin à *Gimp* et à préciser les modes d'interactions (il ne faut pas utiliser de caractères spéciaux ou accentués dans les textes) :

```
register(
        "Eroder_et_dilater",          # nom du plugin
        "Erode et dilate",            # description du plugin (doc)
        "Erode et dilate",            # description du plugin (aide)
        "Louis Bonneau",              # nom de l'auteur
        "Agrocampus Ouest",           # nom de l'organisation
        "2012",                       # date
        "<Image>/Python/MorphoMath",  # position dans les menus de Gimp
        "GRAY*",                      # type d'images acceptées
        [],                           # variables a renseigner dans l'interface graphique
        [],                           # eventuelles valeurs de retour
        process_morphomath)           # nom de la fonction à appeler
```

Transformer votre code précédement ecrit pour en faire un plugin permettant de tracer automatiquement une diagonale rouge dans l'image en couleur éventuellement ouverte dans *Gimp*.

4 Raffinements

4.1 Accélération des traitements

Les procédures d'accès aux valeurs des pixels `drawable.get_pixel(x, y)` et `drawable.set_pixel(x, y, color)` sont peu efficaces. Afin de traiter des images de taille habituelle en un temps acceptable, il faut changer de méthode

d'accès à ces informations : on va ici récupérer l'image comme un flot de texte (une suite de caratcères hexadécimaux codant les couleurs par pixels), que l'on va remettre en forme grâce à la bibliothèque *NumPy*.

```
# la fonction get_pixel_rgn renvoie un tableau de caractères de type '\xhh'
flottexte = drawable.get_pixel_rgn(0, 0, w, h, False, False)
# reformatage du tableau complet sous forme de liste d'entiers 8bits
flotentier = fromstring(flottexte[:,:], dtype='B')
# reformatage sous forme d'une matrice numpy d'entiers 8bits
datas = array(flotentier.reshape(h, w), dtype='B')[:,:]
```

On accèdera alors au contenu en indexant de la manière suivante : `datas[ligne, colonne]` On pourra créer de nouvelles matrices emplies de zéro de la manière suivante :

```
matrice = zeros([h, w], dtype='int16') # exemple pour des entiers 16 bits
```

Enfin, lorsque les traitements seront finis, on re-copiera la matrice dans la structure `pixel_rgn` d'origine :

```
flottexte = datas.tostring() # retour sous forme de flot de texte
```

Modifier votre premier plugin afin de le rendre plus performant compte tenu des indications données.

4.2 Boites de dialogues

Au lancement d'un script *Python*, il est possible d'inter-agir avec l'utilisateur au travers d'une boite de dialogue. Le concepteur du plugin demande à *Gimp* d'afficher des « contrôles », les variables récupérées sont automatiquement transmises au programme. Au moment ou le plugin s'enregistre auprès de *Gimp* il fournira explicitement la liste des variables à renseigner, associées à leur type, leur nom, leur description et leur valeur par défaut :

```
[
   (PF_INT, "width", "Largeur de l'element", 3),
   (PF_INT, "height", "Hauteur de l'element", 3),
   (PF_INT, "xcentre", "Abscisse du centre", 1),
   (PF_INT, "ycentre", "Ordonnee du centre", 1),
],
```

Les types possibles sont nombreux :

- entier : `PF_INT`, `PF_INT8`, `PF_INT16`
- réel : `PF_FLOAT`
- texte : `PF_STRING`
- case à cocher, liste à choix : `PF_BOOL`, `PF_RADIO`
- couleur, à choisir dans une palette : `PF_COLOUR`, `PF_PALETTE`...

Dans l'exemple ci-dessous, la fonction `process_morphomath`, appelée par le plugin, prend 6 paramètres :

```
def process_morphomath(image, drawable, width, height, xcentre, ycentre):
```

Les deux premiers correspondent à l'image et au calque sur lesquels le plugin est appelé (toutes le fonctions sont construites avec ces deux paramètres au moins), les 4 suivants correspondent aux variables renseignées par l'utilisateur.

Modifier votre premier plugin afin de permettre à l'utisateur de choisir la couleur employée pour tracer la diagonale.

4.3 Retour en arrière

La possibilité d'annuler les traitments (`[Ctrl]+Z`) effectués dans le plugin nécessite la definition des points de départ et d'arrêt des blocs de code éventuellement annulables :

```
pdb.gimp_image_undo_group_start(image)    # regrouper les opérations pour annulation
# serie d'instruction python ...
pdb.gimp_image_undo_group_end(image)      # préciser la fin du bloc d'annulation
```

4.4 Patience

Dans le cas de longs traitements, il peut être souhaitable d'informer l'utilisateur sur la quantité de calculs dejà effectués (et restant à faire). L'interface de *Gimp* peut alors intégrer une patience :

```
# declaration et mise a zero du sablier
gimp.progress_init("Traitements en cours...")
# avancement du sablier (fraction comprise entre 0 et 1)
gimp.progress_update(fraction)
# fin du traitement
gimp.pdb.gimp_progress_end()
```

Terminez votre plugin en ajoutant une patience et la possibilité d'annuler la modification apportée.

5 Mini-projets

L'évaluation de ce cours repose sur un mini-projet individuel (ce qui ne veut pas dire que vous ne pouvez pas vous aider...) et sous deux modalités différentes.

Un retour sous forme électronique d'abord, c'est-à-dire une archive au format `.zip` envoyée à l'adresse (`louis.bonneau@agrocampus-ouest.fr`), et qui contiendra :

- un code informatique en *Python*, commenté (les blocs de codes et les variables utilisées sont décrites en quelques mots) et rédigé sous forme de texte non mis en forme : un fichier *txt* issu de *Notepad++* par exemple (et surtout pas un fichier issu d'un traitement de texte !). Il doit être possible d'obtenir l'exécution de ce code sans faire de modification.
- Le résultat du traitement obtenu (image au format png, tableau de valeurs numériques...)

Une présentation orale par ailleurs, de 15 minutes maximum, et avec un support graphique (mais uniquement des diapos au format `pdf`) dans laquelle vous présenterez l'algorithme, les difficultés rencontrées, et un résultat.

Les problématiques de projets ci-dessous sont présentées dans un ordre (subjectif) de difficulté conceptuelle croissante. Une difficulté générale que vous allez rencontrer est de toujours rester dans l'image : on ne peut travailler que dans des pixels qui existent bel et bien (« effet de bord »). Dans certain cas vous pourrez agrandir l'image avant traitement, et la recadrer avant de retourner un résultat... (voir exemple `Morpho.py`).

5.1 Ré-échantillonnage des couleurs d'une image

L'objectif est ici de réduire le nombre de couleurs utilisées pour représenter une scène, en passant d'une représentation RVB à des niveaux de gris sur 8bits. Plusieurs conversions sont envisageables :

- niveaux de gris obtenus par moyenne arithmetique : Gris = (R + V + B) / 3 ;
- proposition 709 de la Commision Internationale de l'Éclairage : Gris = 0.2125*R + 0.7154*V + 0.0721*B ;
- pellicule Kodak TMax 400 : Gris = 0.27*R + 0.36*V + 0.37*B ;

Le plugin devra permettre à l'utilisateur de choisir (la ou) les conversions préférées : s'il y en a plusieurs, il faudra les réaliser simultanément et les stocker dans des calques différents. L'illustration de la différence de rendu obtenu se fera à l'aide d'une image bien choisie.

5.2 Dessin d'un segment de droite

L'objectif est de dessiner un segment de droite automatiquement, entre deux points d'une image. La couleur du dessin ainsi que les coordonnées des points à relier seront demandé à l'utilisateur du plugin, lors du lancement de celui-ci. Il faudra vérifier que les coordonnées fournies sont bien à l'intérieur de l'image...

Raffinement : essayer de limiter l'aliasing.

5.3 Ré-échantillonnage géométrique d'une image

L'objectif est ici de changer la résolution spatiale d'une image, en demandant à l'utilisateur, dans une boite de dialogue, la nouvelle taille (votre algorithme la vérifira avant de lancer le calcul...). Vous mettrez en place trois interpolations différentes :

- aucune pour commencer (valeur du plus proche voisin).
- où les poids associés à chacun des voisins seront proportionnels à la distance euclidienne;
- où les poids associés à chacun des voisins seront proportionnels à la somme des distances sur chaques coordonnées.

Vous comparerez la qualité des résultats et le temps nécessaire à chaque calcul (utilisez la fonction `time.clock()`).

5.4 Amélioration de la dynamique d'une image

L'objectif est ici de corriger automatiquement la distribution (l'histogramme) des niveaux de gris d'une image.

1. Écrivez le code vous permettant de calculer automatiquement les trois paramètres qui vous seront utiles : niveaux de gris minimal, maximal et moyen, dans l'image traitée.
2. Écrivez la fonction mathématique vous permettant de modifier l'image pour ramener ces trois valeurs à (0, 127, 255).
3. Completez votre plugin pour que toute l'opération soit automatique.

Raffinement : améliorez votre plugin pour traiter maintenant des images en couleur.

5.5 Filtrage conditionnel

L'objectif est ici de filtrer des images en niveaux de gris avec des conditions fournies par l'utilisateur du filtre. L'idée est de ne pas tenir compte des pixels dont l'écart de couleur avec la valeur moyenne du voisinage est supérieur à un seuil.

1. Ecrivez l'algorithme de parcours et de filtrage d'une image en niveau de gris, pour le filtre donné ci-dessous.
2. Faites en sorte qu'une interface graphique demande à l'utilisateur de choisir un seuil.
3. Appliquez cet algorithme à l'image *tugs.png* fournie, en essayant différentes valeurs de seuil.

	1	1	1	
1	1	1	1	1
1	1		1	1
1	1	1	1	1
	1	1	1	

5.6 Filtrage de Nagao

Le filtrage de Nagao est un filtre réducteur de bruit sans perte de contraste. L'idée générale est de chercher le sous ensemble du filtre dans lequel la variance est minimale, et de conserver la moyenne correspondante. Il y a 9 voisinages distincts (les 3 ci-dessous et leur symétriques).

			1	1
		1	1	1

			1	1
		1	1	1
		1	1	

	1	1	1	
	1	1	1	
	1	1	1	

1. Construisez, dans un éditeur de texte, les codes en *Python* permettant de réaliser les calculs à effectuer pour chacun des 9 voisinages successivement, à partir des coordonnées arbitraites (x,y) du centre du filtre.
2. Il s'agit maintenant de conserver la moyenne arithmetique associée à la variance la plus faible : pour cela deux variables vont être utilisées, l'une stockant la plus faible variance rencontrée jusqu'ici, l'autre la moyenne qui lui était associée. À la fin de l'examen de chaque voisinage, ces variables seront éventuellement mises à jour.
3. Implémentez l'algorithme complet, comprenant les 2 boucles imbriquées de parcours de l'image et le bloc de calculs préparé aux étapes précédentes. Pour plus de facilité vous pourrez ignorer les bords de l'image sur 2 pixels d'épaisseur.
4. Appliquez cet algorithme à l'image *tugs.png* pour vérifier son bon fonctionnement.

5.7 Tramage d'une image en niveau de gris

Le tramage va consister à parcourir l'image non pas pixel par pixel, mais par blocs de la taille de la trame.

1. Calculez les seuils en niveaux de gris associés à la trame présentée ci-dessous.
2. Construisez la double boucle de parcours « image modulo trame » (s'il n'y a pas un nombre entier de trames dans l'image, vous pourrez ignorer la frange restante).
3. Ecrivez directement en *Python* la série des 16 tests à mettre en oeuvre pour seuiller les pixels de chaque trame.
4. Implémentez l'algorithme complet en python, avec pour objectif de créer automatiquement, à partir d'une image de départ en niveau de gris, une image binaire de même résolution (les éventuelles franges non traitées conserveront la couleur par défaut du calque).
5. Appliquez cet algorithme à l'image *tugs.png* fournie pour vérifier son bon fonctionnement.

12	5	6	13
4	0	1	7
11	3	2	8
15	10	9	14

5.8 Binarisation avec diffusion de l'erreur

L'objectif est ici de reproduire une binarisation de type Floyd-Steinberg, c'est à dire en diffusant l'erreur de binarisation sur les pixels voisins. La difficulté réside dans la contrainte de domaine : les valeurs des pixels sont comprises entre 0 et 255. La diffusion ne pourra donc pas toujours se faire comme prévue, et devra elle-même être reportée sur les voisins... Pour plus de facilité, vous pouvez dans un premier temps ne repporter l'erreur que sur un seul voisin. Par ailleurs, vous pouvez avoir intérêt à travailler temporairement dans une matrice d'entiers 16 bits .

1. Ecrivez l'algorithme de binarisation avec diffusion de l'erreur, c'est-à-dire l'écart absolu entre la valeur en niveau de gris d'origine et la valeur donnée au pixel examiné (soit 0 ou 255), additionné du reliquat éventuel issu des pixels précédents saturés.
2. Implémentez cet algorithme afin de créer automatiquement, à partir d'une image de départ en niveau de gris, une image binaire de même résolution.
3. Appliquez cet algorithme à l'image *tugs.png* fournie, pour vérifier son bon fonctionnement.

		$\frac{7}{16}$
$\frac{3}{16}$	$\frac{5}{16}$	$\frac{1}{16}$

5.9 Transformation tout-ou-rien

L'objectif est de mettre en oeuvre un filtrage dans lequel seuls les pixels de l'enveloppe convexe d'une forme, sont conservés.

	1	
0	1	1
0	0	

	1	
1	1	0
	0	0

	0	0
1	1	0
	1	

0	0	
0	1	1
	1	

Vous pourrez, si vous le souhaitez, travailler sur des images en niveaux de gris ne contenant que du noir et du blanc plutôt que sur des images en couleurs indexées.

1. En notant (x,y) les coordonnées centrales des filtres, construisez en *Python* les conditions à vérifier pour chacun des 4 cas de figures : tous les pixels marqués 1 dans le voisinage doivent appartenir à la forme, ceux marqués 0 doivent appartenir au fond.
2. Implémentez cet algorithme en python et appliquez-le à une image où cette transformation est pertinente, pour vérifier son bon fonctionnement.

Raffinements :

- faites en sorte qu'au lancement du filtre, une interface graphique demande à l'utilisateur la couleur de la forme;
- adaptez le code à des images en niveaux de gris, en demandant la valeur du seuil de binarisation à l'utilisateur.

5.10 Suppression des objets touchant le bord

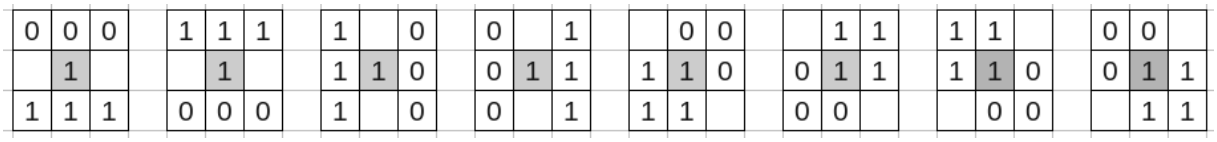
Vous allez créer automatiquement, à partir d'une image de départ binaire, une image binaire de même résolution dans laquelle il n'y aura plus d'objet en contact avec le bord de l'image. (vous pourrez si vous le souhaitez, travailler sur des images en niveaux de gris ne contenant que du noir et du blanc plutôt que sur des images en couleurs indexées).

1. Ecrivez le code permettant de parcourir et examiner le contenu des bords de l'image.
2. Ecrivez une fonction qui examine les éléments du voisinage d'un pixel de coordonnées quelconques et qui, si ceux-ci lui sont connectés, les supprime de la forme.
3. Complétez votre code en rendant votre fonction récursive (la fonction de nettoyage s'appelle elle-même sur son voisinage).

Veillez à tester votre plugin sur des petites formes (centaines de pixels) car la profondeur des appels recursifs est limitée... Raffinements : faites en sorte qu'au lancement du filtre, une interface graphique demande à l'utilisateur la couleur de la forme et du fond.

5.11 Squelettisation

L'objectif est de mettre en oeuvre une squelettisation par filtrage itératif des pixels non-essentiels. Les configurations dans lesquelles les pixels peuvent être enlevés sont les suivantes :



Vous allez créer automatiquement, à partir d'une image de départ binaire, une image binaire squelettisée de même résolution (vous pourrez si vous le souhaitez, travailler sur des images en niveaux de gris ne contenant que du noir et du blanc plutôt que sur des images en couleurs indexées).

1. En notant (x,y) les coordonnées centrales des filtres, construisez en *Python* les conditions à vérifier pour chacun des cas de figures : tous les pixels marqués 1 dans le voisinage doivent appartenir à la forme, ceux marqués 0 doivent appartenir au fond.
2. Implémentez cet algorithme en python pour une seule itération, comprenant les 2 boucles imbriquées de parcours de l'image et les 8 blocs de conditions préparés à l'étape précédente.
3. Rajoutez la structure d'itération « tant qu'on a modifié l'image à l'itération précédente » (pour cela il faut mémoriser le fait d'avoir ou non modifié l'image, et utiliser l'instruction `while` *condition*`:` où condition prend la forme d'un test vrai ou faux).
4. Appliquez cet algorithme aux contours de parcelles issues de l'image *chca.png* pour vérifier son bon fonctionnement.

5.12 Baguette magique

Vous allez rechercher à colorier, à partir d'un point de coordonnées fournies par l'urtilisateur, toute la composante connexe à laquelle ce point appartient. Deux pixels seront connectés s'ils sont peu différents.

1. Écrivez une fonction qui examine les éléments du voisinage d'un pixel de coordonnées (lig, col) et, si ceux-ci sont connectés, c'est-à-dire que l'écart entre leurs niveaux de gris est inférieur à (seuil), y substitue la couleur (coul).
2. Rendez cette fonction récursive, c'est-à-dire qu'elle s'appelle elle-même sur tous les voisins connectés.
3. Terminez votre plugin en faisant en sorte de récupérer les paramètres auprès de l'utilisateur (coordonnées, couleur de substitution et seuil de variation maximale).

Veillez à tester votre plugin sur des petites formes (centaines de pixels) car la profondeur des appels recursifs est limitée... Raffinements : faites en sorte de colorier la composante identifiée non pas avec la valeur d'un paramètre, mais avec la valeur moyenne de la composante.

5.13 Identification de composantes connexes

Vous allez créer automatiquement, à partir d'une image en niveau de gris, un calque de même dimension qui lui sera superposé, et qui identifiera certaines zones de l'image d'origine. L'objectif est d'identifier toutes les composantes connexes d'une même couleur (paramètre fourni par l'utilisateur) et de les numéroter. Ce projet abordera le problème avec un parcours récursif.

1. Écrivez le code permettant de créer un nouveau calque de même taille que l'image, et de le remplir avec des 0.
2. Faites en sorte de demander à l'utilisateur de renseigner le paramètre de couleur (i) à identifier, et de parcourir le contenu de l'image à la recherche de pixels valant (i). À chaque fois qu'un tel pixel sera trouvé, votre code incrémentera le numéro des composantes connexes déjà identifiées, et fera appel à la fonction ci-dessous.
3. Écrivez une fonction qui examine les éléments du voisinage d'un pixel de coordonnées (l, c) et, si ceux-ci lui sont connectés (c'est-à-dire de même couleur i), les marque dans le calque d'étiquettes.
4. Complétez votre code en rendant cette fonction récursive (la fonction de substitution s'appelle elle-même sur son voisinage).

Veillez à tester votre plugin sur des petites formes (centaines de pixels) car la profondeur des appels recursifs est limitée...

5.14 Composantes connexes

Vous allez mettre en oeuvre un étiquetage modifié des composantes connexes d'une image pour compter des cellules. La difficulté réside dans la manipulation de tableaux en *Python*.

1. Implémentez l'algorithme classique d'étiquetage en composante connexe, dont le fonctionnement normal est de donner le numéro de la composante à chaque pixel de celle-ci.
2. Ajoutez un tableau comptabilisant le nombre de pixels associés à chaque étiquette.
3. Récupérez et affichez l'image *cellules.png*, fixez un seuil de taille permettant de discriminer les cellules des noyaux.
4. Utilisez ce seuil pour calculer automatiquement le nombre de cellules dans l'image.