

PROJET MSLD - INFORMATIONS COMPLÉMENTAIRES



SOMMAIRE



ORGANISATION DU CODE



Notebook et fichier Python séparés



Syntaxe Orient-Objet: Pourquoi?



Syntaxe Orient-Objet: en pratique...



Organigramme des Méthodes



IMPLÉMENTATION MSLD



Aperçu Général de l'implémentation



Basic Line Detector

→ QUESTION I.4



Création des masques de convolutions

→ QUESTION I.1 & I.2



Chargement de la base de données

→ QUESTION I.3 & II.14



Apprentissage du Seuil

→ QUESTION I.8 & I.10



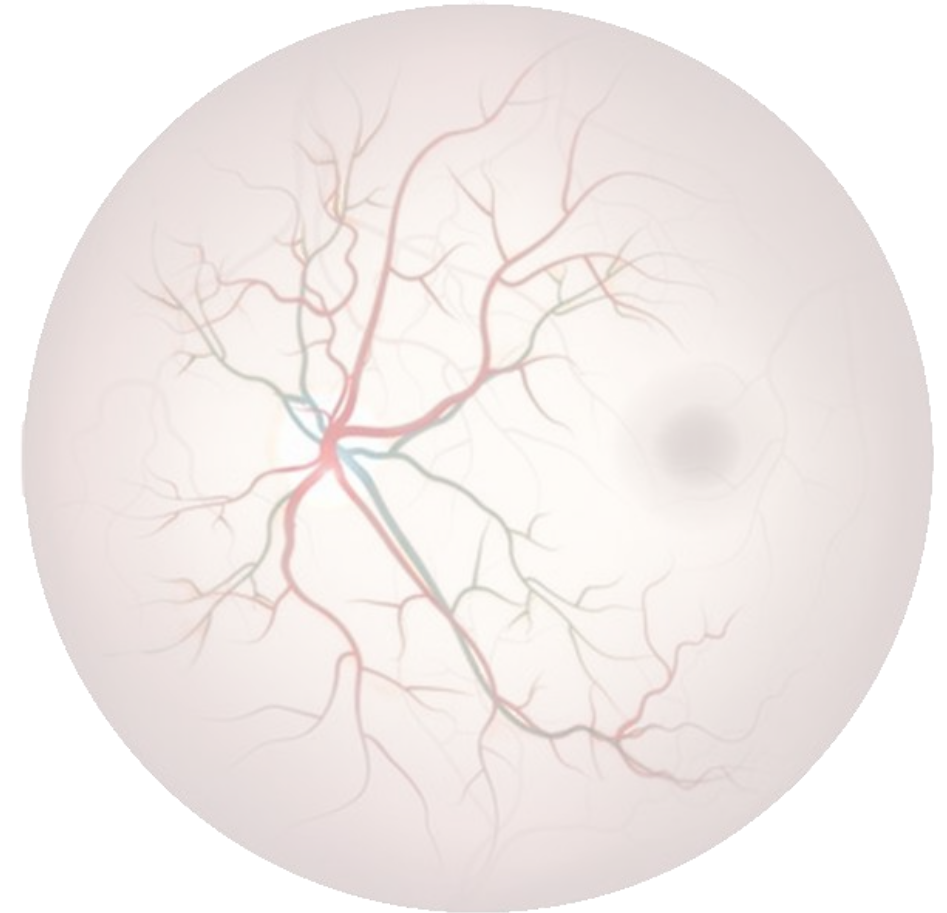
CORRECTIONS ET PRÉCISION DU SUJET



Installation des Packages



À corriger dans le code



NOTEBOOK ET FICHIER PYTHON SÉPARÉS

Contrairement aux TP, le code que vous à utiliser et compléter pour le projet est répartie entre 2 fichiers.

MSLD.py

- Un fichier de code python implémentant l'algorithme MSLD.

→ C'est un code *à trou*, Vous y complétez les fonctions implémentant l'algorithme et signalées par les commentaires:

```
### TODO: I.Q1
self.avg_mask = ...

### TODO: I.Q2
### line_detectors_masks est un dictionnaire contenant les masques
### de détection de ligne pour toutes les échelles contenues
```

- Ce fichier contient de nombreuse aide pour l'implémentation dans la documentation des fonctions ou en commentaire du code. Prenez le temps de les lire!
- Pour utiliser les fonctionnalités de ce fichier dans le Notebook il faut les importer. C'est le rôle de la seconde cellule du Notebook:

```
from MSLD import MSLD, load_dataset
```

RapportProjet.ipynb

- Un notebook ipython similaire à celui des TP.
- Vous y répondez aux questions théoriques et y affichez les images, graphiques...

! Pour que les changements apportés à MSLD.py soient pris en compte dans RapportProjet.ipynb il faut:

- sauvegarder MSLD.py,
- redémarrer le kernel de RapportProjet.ipynb,
- réexécuter toutes ses cellules.

→ Pour éviter les deux dernières étapes, ajoutez les 2 lignes suivantes dans la première cellule de RapportProjet.ipynb:

```
%load_ext autoreload
%autoreload 2

# Importe les modules qui seront utilisé dans le
laboratoire.
import numpy as np
import matplotlib.pyplot as plt
```

SYNTAXE ORIENTÉE-OBJET: POURQUOI?

Le fichier MSLD.py suit une syntaxe *orienté-objet*. Ces 2 slides présentent les bases de cette syntaxe.

→ La syntaxe Orienté-Objet permet de décrire la structure d'un modèle type (la classe) comme une variable possédant **des attributs** (des variables partagées au sein de l'objet) et **des méthodes** qui utilisent ces attributs.

Dans notre cas l'objet `MSLD` à ...

... les attributs:

- `L`: la liste des longueurs de ligne analysées
- `W`: la taille du filtre moyenneur
- `n_orientation`: le nombre d'orientation évaluées
- `threshold`: le seuil de binarisation de la réponse du MSLD
- `avg_mask`: le masque moyenneur de taille `W`
- `line_detectors_masks`: les masques moyenneur le long de ligne

... les méthodes:

- `__init__(W, L, n_orientation)`: le constructeur (*voir slide suivante*)
- `baseLineDetector(grey_lvl, L)`: l'algorithme base line detector
- `multiScaleLineDetector(image)`: le nombre d'orientation évaluées
- `learnThreshold(dataset)`: apprend le seuil optimal pour un dataset donné
- `segmentVessels(image)`: seuille la réponse du MSLD avec le seuil appris
- ...

→ Une fois défini, plusieurs objets peuvent être construit à partir d'une même classe, chacun avec des valeurs d'attributs propres. Puisque les méthodes utilisent la valeur de ces attributs dans leur code, le résultat d'une même méthode ne sera pas le même si elle est appelée depuis un objet ou un autre.

Le code :

```
msld1 = MSLD(W=11, L=[3,5,7], n_orientation=4)
msld2 = MSLD(W=9, L=[5,7], n_orientation=8)
```

instancie deux objet MSLD avec des valeurs différentes

d'attributs `W`, `L` et `n_orientation`. Ceux-ci sont des hyperparamètres de `multiScaleLineDetector(image)`, l'appel de cette méthode depuis l'une ou l'autre des instances ne produira donc pas le même résultat. Il sera donc facile de comparer l'effet des différents hyperparamètres:

```
R1 = msld1.multiScaleLineDetector(image)
R2 = msld2.multiScaleLineDetector(image)
```

SYNTAXE ORIENTÉE-OBJET: EN PRATIQUE...

Cette slide décortique la syntaxe Orienté-Objet de la classe MSLD.

MSLD.py

```

1 class MSLD:
    2 def __init__(self, W, L, n_orientation): 3
        self.W = W
        self.L = L
        self.n_orientation = n_orientation
        self.threshold = 0.56
        ...

    2 def basicLineDetector(self, grey_lvl, L):
        ...
        return R

    2 def multiScaleLineDetector(self, image):
        ...
        for l in self.L: 4
            R = self.basicLineDetector(lvl_grey, 1)
            ...
        return Rcombined
  
```

RapportProjet.ipynb

```

1 msld15 = MSLD(W=15, L=[3,5,7], n_orientation=12)
  msld25 = MSLD(W=25, L=[3,5,7], n_orientation=12)

R15 = msld15.basicLineDetector(grey_lvl, 3) 3
R25 = msld25.basicLineDetector(grey_lvl, 3)

print('msld15.W: ', msld15.W) 2
print('msld25.W: ', msld25.W)

msld15.W: 15
msld25.W: 25
  
```

Dans MSLD.py :

- ① Début de la définition de la classe
- ② Définition des méthodes de la classe
Notez que le premier paramètre est toujours `self`: c'est une référence vers l'instance de la classe MSLD qui a provoquée l'appel de la méthode (cf. ③).
- ③ Le constructeur de la classe MSLD
Le constructeur d'une classe est une méthode spéciale qui permet d'initialiser les attributs d'un objet lorsqu'il est instancié. Les attributs peuvent être initialisés par un paramètre du constructeur (`self.W`, `self.L`, `self.n_orientation`), par des variables calculées à partir de ces paramètres (`self.avg_mask`, `self.line_detectors_masks`) ou par une valeur par défaut (`self.threshold`).

Dans RapportProjet.ipynb :

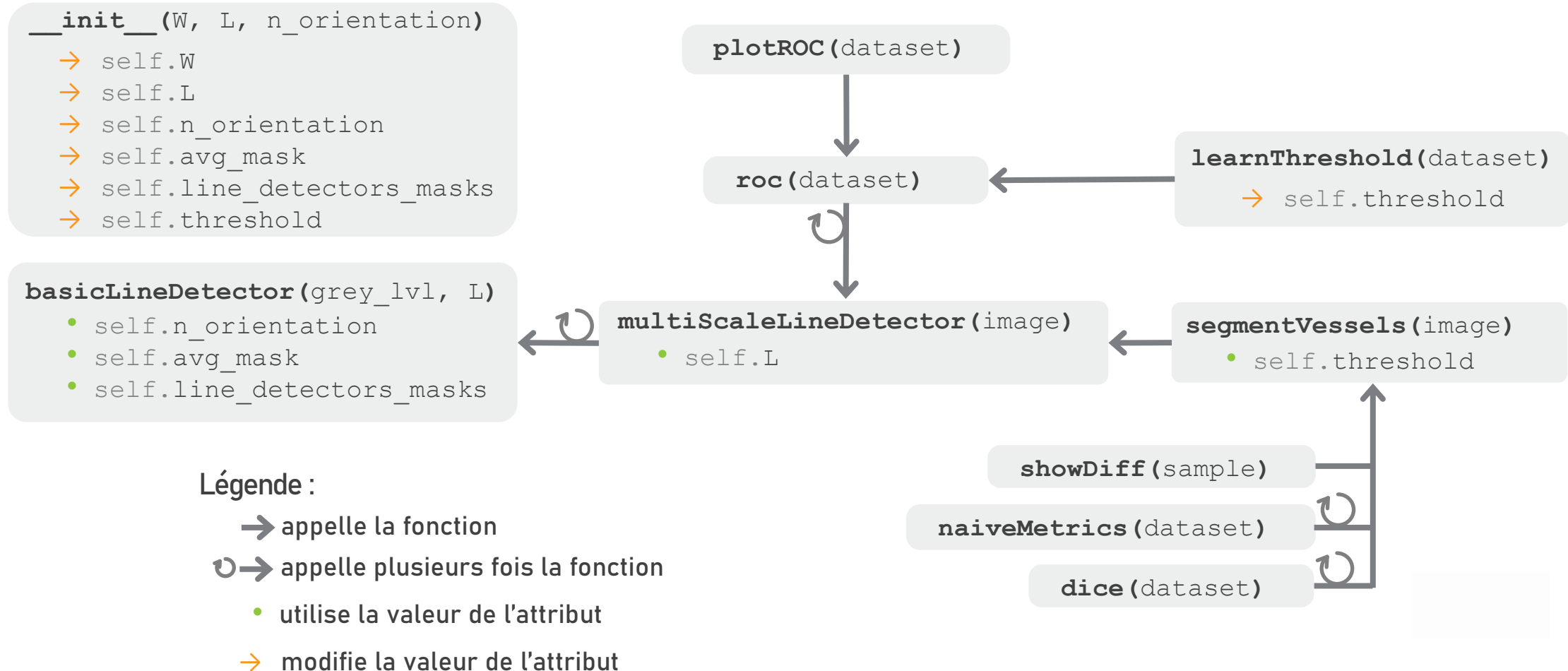
- ① Instanciation d'un objet
Le constructeur de la classe est appelé avec la syntaxe: `nomClasse(paramètres)`. Plusieurs objets peuvent être construit à partir de la même classe, chacun à une valeur propre pour ses attributs.
- ② Accès aux attributs d'un objet
Un fois instancié, on accède à un attribut d'un objet par: `nomObjet.attribut`.
- ③ Appel des méthodes d'un objet
Comme pour les attributs, on appelle une méthode par: `nomObjet.methode()`. Notez que le premier argument `self` est omis. Python le remplace automatiquement par l'objet à l'origine de l'appel (ici `msld15` et `msld25`).
- ④ Accès aux membres de la classe
À l'intérieur de la définition d'une méthode, il est possible d'accéder aux membres de la classe (ses attributs et ses méthodes) via le mot clé `self`. Les attributs peuvent être lu et modifié ce qui permet de partager une variable entre les différentes méthodes de l'objet. Notez que lorsqu'on appelle une méthode, on omet son premier paramètre `self` (python se charge de lui attribuer la bonne valeur).



ORGANIGRAMME DES MÉTHODES

L'une des forces de la syntaxe orienté-objet est de permettre aux différentes méthodes de la classe MSLD de s'appeler entre-elles et d'interagir avec des variables partagées entre-elles (celle préfixée de `self.`).

Ainsi, une fois toutes les implémentations complétées, l'organigramme des appels de méthodes devrait-être:



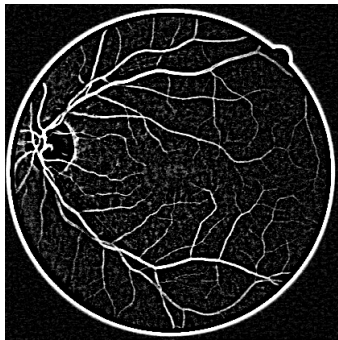
APERÇU GÉNÉRAL DE L'IMPLÉMENTATION

(Toutes les images sont affichées en fixant $v_{min}=0$ et $v_{max}=1$.)

1. Basic Line Detector

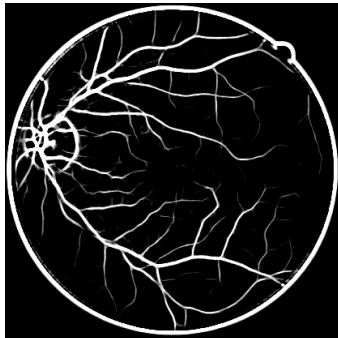
$$R_W^l = I_{max}^l - I_{avg}^l \quad (\text{Eq. 2})$$

$$R_W'^l = \frac{R_W^l - \text{mean}(R_W^l)}{\text{std}(R_W^l)} \quad (\text{Eq. 3})$$



$l=1$

⋮

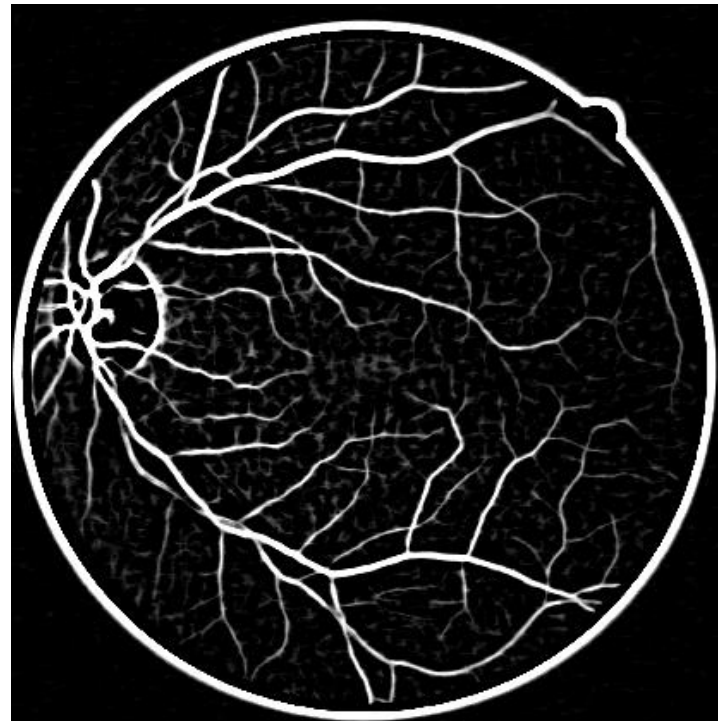


$l=15$

2. Multi-Scale Line Detector

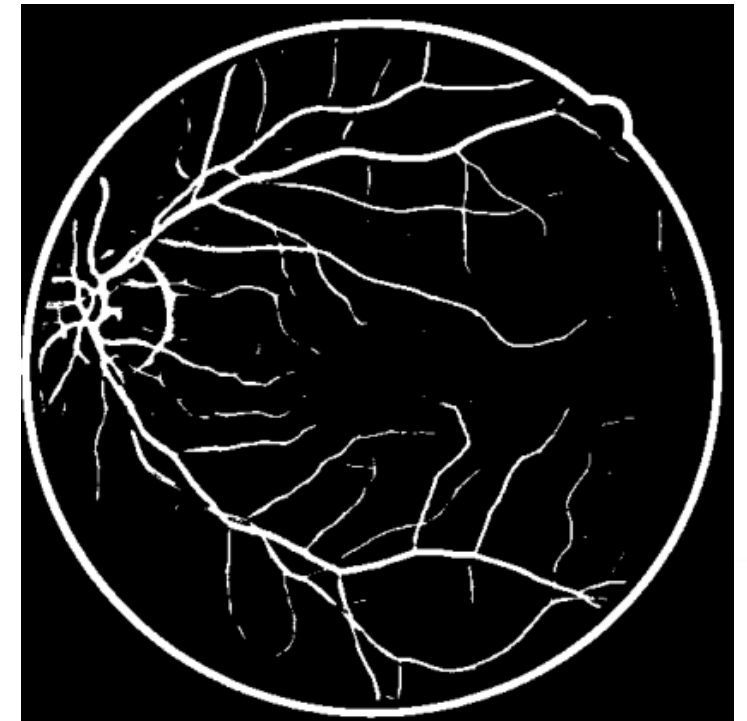
I_{igc} : Inverted Green Channel

$$R_{combined} = \frac{1}{n_L + 1} \left(I_{igc} + \sum_l R_W'^l \right) \quad (\text{Eq. 4})$$



3. Seuillage

$$R_{seuil} = R_{combined} > \text{seuil}$$



`basicLineDetector(grey_lv1, L)`

`multiScaleLineDetector(image)`

`segmentVessel(image)`

BASIC LINE DETECTOR


→ L'algorithme Basic Line Detector consiste à:

1. Calculer la valeur moyenne des intensités le long de lignes centrées sur le pixel (longueur fixe, orientation variable)
2. Parmi les orientations testées, sélectionner celle offrant la valeur moyenne maximale: I_{max}^W
3. Calculer la valeur moyenne des intensités d'un voisinage de taille W autour du pixel: I_{avg}^W
4. Calculer la réponse du BLD: $R_W = I_{max}^W - I_{avg}^W$.
5. (Pour améliorer la lisibilité je conseille de faire la normalisation à la fin du BLD: $R'_W = \frac{R_W - \text{mean}(R_W)}{\text{std}(R_W)}$).

Graphiquement avec $W=3$ et $n_orientation=4$, l'algorithme calcule pour chaque pixel:

$$R = \max \left(\begin{array}{c} \begin{array}{|c|c|c|} \hline \text{light blue} & \text{light blue} & \text{light blue} \\ \hline \text{red} & \text{red} & \text{red} \\ \hline \text{light blue} & \text{light blue} & \text{light blue} \\ \hline \end{array}, \begin{array}{|c|c|c|} \hline \text{red} & \text{light blue} & \text{light blue} \\ \hline \text{light blue} & \text{red} & \text{light blue} \\ \hline \text{light blue} & \text{light blue} & \text{red} \\ \hline \end{array}, \begin{array}{|c|c|c|} \hline \text{light blue} & \text{red} & \text{light blue} \\ \hline \text{light blue} & \text{red} & \text{light blue} \\ \hline \text{light blue} & \text{light blue} & \text{red} \\ \hline \end{array}, \begin{array}{|c|c|c|} \hline \text{light blue} & \text{light blue} & \text{red} \\ \hline \text{light blue} & \text{red} & \text{light blue} \\ \hline \text{red} & \text{light blue} & \text{light blue} \\ \hline \end{array} \right) - \begin{array}{|c|c|c|} \hline \text{red} & \text{red} & \text{red} \\ \hline \text{red} & \text{red} & \text{red} \\ \hline \text{red} & \text{red} & \text{red} \\ \hline \end{array}$$

Légende :



Intensité moyenne des pixels rouges autour du pixel central.

→ Pour éviter de devoir appliquer cette opération à chaque pixel avec des boucles for, on remplace les calculs d'intensité moyenne autour de chaque pixel par une convolution sur toute l'image. Ce qui revient au même en choisissant les bonnes valeurs pour les masques.

Ainsi, en notant I l'image sur laquelle on applique le BLD, $M_{W \times W}$ un masque moyennneur de taille W et $*$ la convolution, l'algorithme devient:

$$R = \max_{n_orientation} \left(I * \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 1/3 & 1/3 & 1/3 \\ \hline 0 & 0 & 0 \\ \hline \end{array}, I * \begin{array}{|c|c|c|} \hline 1/3 & 0 & 0 \\ \hline 0 & 1/3 & 0 \\ \hline 0 & 0 & 1/3 \\ \hline \end{array}, I * \begin{array}{|c|c|c|} \hline 0 & 1/3 & 0 \\ \hline 0 & 1/3 & 0 \\ \hline 0 & 1/3 & 0 \\ \hline \end{array}, I * \begin{array}{|c|c|c|} \hline 0 & 0 & 1/3 \\ \hline 0 & 1/3 & 0 \\ \hline 1/3 & 0 & 0 \\ \hline \end{array} \right) - I * M_{W \times W}$$

(Pour l'opération de maximum lire le rôle du paramètre `axis` dans la documentation de [np.max\(a, axis=None\)](#).)

CRÉATION DES MASQUES DE CONVOLUTION

→ Pour ne pas avoir à recalculer les masques de convolutions à chaque appel de la fonction `basicLineDetector` et toutes les méthodes qui l'utilisent, on va les calculer une seule fois lors de l'instanciation de l'objet MSLD et les stocker comme attributs:

- `self.avg_mask`: le masque moyenneur de taille `W` (à compléter à la question 1).
- `self.line_detectors_masks`: les masques moyenneurs le long de ligne pour chaque longueur et orientation (question 2).

→ Pour construire `self.line_detectors_masks` on suit la procédure suivante pour chaque valeur `l` contenue dans `self.L`:

1. Générer `line_detector`: un masque carré de taille $l \times l$ dont seule la ligne horizontale (ou verticale) est initialisée à $1/l$. (On peut supposer l impair.)
2. Effectuer `self.n_orientation - 1` rotations de ce masque à l'aide du code ci-dessous.
3. Empiler ses masques dans une matrice unique de taille $(l, l, n_orientation)$, en ayant normalisé chacun pour que sa somme soit égale à 1.
4. Empiler ces masques dans une matrice unique de taille $(l, l, n_orientation)$ puis la stocker dans `self.line_detectors_masks` à la clé `l`:

`self.line_detectors_masks[l] = np.stack(line_detector_masks, axis=2)`

→ Pour accéder au masque de longueur `L` et d'orientation `n`, on utilisera:

`self.line_detectors_masks[l][:, :, n]`

Par exemple, pour `l=3` et `5`, `n_orientation=3` et en notant `m = self.line_detectors_masks[l]`, on aura des masques similaires à :

| [l] | m.shape | m[:, :, 0] | m[:, :, 1] | m[:, :, 2] |
|-----|-----------------------|--|--|--|
| [3] | (3, 3, 4) | $\begin{bmatrix} 0 & 0 & 0 \\ .33 & .33 & .33 \\ 0 & 0 & 0 \end{bmatrix}$ | $\begin{bmatrix} .13 & .16 & 0 \\ .04 & .33 & .04 \\ 0 & .16 & .13 \end{bmatrix}$ | $\begin{bmatrix} 0 & .16 & .13 \\ .04 & .33 & .04 \\ .13 & .16 & 0 \end{bmatrix}$ |
| [5] | (3, 3, 4) | $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ .2 & .2 & .2 & .2 & .2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ | $\begin{bmatrix} .02 & .14 & 0 & 0 & 0 \\ 0 & .12 & .1 & 0 & 0 \\ 0 & .02 & .2 & .02 & 0 \\ 0 & 0 & .1 & .12 & 0 \\ 0 & 0 & 0 & .14 & .02 \end{bmatrix}$ | $\begin{bmatrix} 0 & 0 & 0 & .14 & .02 \\ 0 & 0 & .1 & .12 & 0 \\ 0 & .02 & .2 & .02 & 0 \\ 0 & .12 & .1 & 0 & 0 \\ .02 & .14 & 0 & 0 & 0 \end{bmatrix}$ |
| [l] | (l, l, n_orientation) | line_detector | rotated_mask | rotated_mask |
| | | | Rotation 180°/n_orientation | Rotation 180°/n_orientation |

Pour `n_orientation > 2`, certaines lignes chevaucheront plusieurs pixels sans parfaitement les traverser. Pour gérer ces orientations, on délègue la rotation à une librairie de traitement d'image qui interpolera et répartira l'intensité de la ligne sur les différents pixels qu'elle traverse.

Le code proposé dans `MSLD.py` pour la rotation produit des aberrations d'interpolation utilisez plutôt:

```
import cv2
r = cv2.getRotationMatrix2D((l//2, l//2), angle, 1)
rotated_mask = cv2.warpAffine(line_detector, r, (l, l))
```

avec:

- `l`: la longueur de la ligne à détecter;
- `angle`: l'angle de rotation en degrés
- `line_detector`: la ligne horizontale à pivoter.

(Voir [Installation des Packages](#) pour l'installation de `cv2`.)

CHARGEMENT DE LA BASE DE DONNÉES

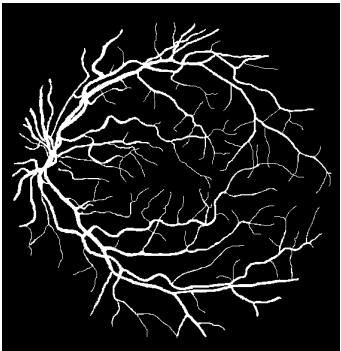
Chaque élément de la base de données contient 3 images:



image (type: image couleur)
L'image de fond d'œil.

Chemin des images

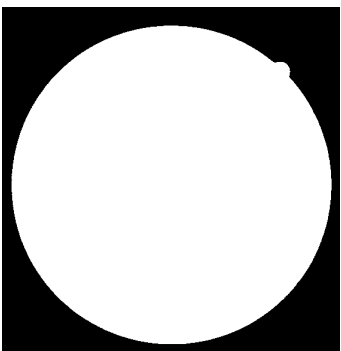
entraînement: 'DRIVE/data/training/'
test: 'DRIVE/data/test/'



label (type: masque binaire)
La segmentation désirée pour les vaisseaux. Elle va nous permettre d'apprendre le seuil optimal (entraînement) puis de tester l'algorithme (test).

Chemin des images

entraînement: 'DRIVE/label/training/'
test: 'DRIVE/label/test/'



mask (type: masque binaire)
La région d'intérêt de l'image. Les métriques ne seront calculées que sur cette région.

Chemin des images

entraînement: 'DRIVE/mask/training/'
test: 'DRIVE/mask/test/'

→ La structure de stockage des bases de données d'entraînement `train` et de test `test`, est une liste de dictionnaire: chaque dictionnaire contenant les entrées `'image'`, `'label'` et `'mask'`, et décrit un élément de la base. Ainsi pour accéder aux 3 images du 3^{ème} élément de la base de test, la syntaxe est:

```
elem3 = test[3]
image3 = elem3['image']
label3 = elem3['label']
mask3 = elem3['mask']
```

→ Pour charger les images, on va tirer profit du fait que les 3 fichiers appartenant au même échantillon ont le même nom:

1. Obtenir la liste des noms de fichiers dans le dossier contenant les images de fond d'oeil:
`files = sorted(os.listdir('DRIVE/data/training/'))`
Le mot clé `sorted` permet de lire les fichiers toujours dans le même ordre.

2. Pour chaque nom de fichier `file`, on crée un dictionnaire `sample` dans lequel on stocke le nom de l'image, l'image, le label et le masque (en ayant converti label et mask en binaire):

```
for file in files:
    sample = {}
    sample['name'] = file
    sample['image'] = imread('DRIVE/data/training/' + file)
    ...
```

→ Pour dupliquer une base, afin d'en modifier les éléments:
`trainCopy = deepcopy(train)`

→ Pour itérer sur tous les éléments d'une base, par exemple afin d'appliquer le même traitement à tous les masques, on utilisera la syntaxe:

```
for elem in train:
    elem['mask'] = ...
```

APPRENTISSAGE DU SEUIL

→ Pour identifier le seuil optimal, il faut tester tous les seuils possibles et identifier celui qui permet d'atteindre la meilleure précision (accuracy) sur l'ensemble d'entraînement.

Mais cette opération est longue. Pour l'optimiser on la délègue à la fonction `sklearn.metrics.roc_curve(label, prediction)` qui calcule pour une série de labels et de prédictions non-seuillées: le taux de faux positifs (fpr) et de vrais positifs (tpr) qui seraient obtenues pour chaque valeur de seuil (thresholds) possible.

→ Il souhaite donc pouvoir calculer la valeur de la précision à partir de fpr, tpr et de constantes indépendantes du seuil: le nombre de pixels positifs et négatifs du label (P et N).

Aidez-vous du formulaire ci-contre pour trouver cette formule.

→ Lors de l'implémentation de `learnThreshold(dataset)`, vous devrez utiliser `self.roc(dataset)` pour obtenir fpr et tpr, et vous devrez vous inspirer de son code pour calculer P, N et S en parcourant le dataset. Avec ces données vous pourrez finalement calculer la précision associée à chaque seuil et choisir celle maximum. N'oubliez pas de stocker le seuil optimal dans `self.threshold` pour qu'il puisse ensuite être utilisé par `segmentVessel(image)`.

Astuces:

- `label[mask]` retourne les pixels de `label` pour lesquels `mask` vaut `True`
- `np.sum(label)` compte le nombre de pixel positifs (égal à `True`) de `label`;
- `np.argmax(accuracies)` calcule l'index de l'élément maximum de `accuracies`.

Dans la documentation de la fonction `roc(dataset)`, les variables de retour fpr et tpr sont inversées.

Cette méthode s'utilise en fait:

```
fpr, tpr, thresholds = self.roc(dataset)
```

Si vous le désirez et pour que ce soit plus clair, vous pouvez modifier les 2 dernières lignes de `roc(dataset)` pour que les nomenclatures soient correctes:

```
fpr, tpr, thresholds = roc_curve(y_true, y_pred)
return fpr, tpr, thresholds
```

Formulaire Précision

(extrait de [la page ROC de Wikipedia](#))

- Nombre de pixel positif (`True`) du label: P
- Nombre de pixel négatifs (`False`) du label : N
- Nombre total de pixel: $S = N + P$
- Nombre de pixel vrais positifs: TP
- Nombre de pixel vrais négatifs: TN
- Taux de vrais positifs: $TPR = \frac{TP}{P}$
- Taux de vrais négatifs: $TNR = \frac{TN}{N} = 1 - FPR$
- Précision: $ACC = \frac{TP+TN}{P+N}$

INSTALLATION DES PACKAGES

Deux des paquets utilisés dans ce TP risque de ne pas être installé dans votre environnement Python: cv2 et sklearn.

Pour les installer, entrer les commandes suivantes dans votre terminale: Anaconda Prompt (celui utilisé au TP0).

- Si vous avez suivi à la lettre l'installation du TP0 et que votre environnement Python s'appelle tpGBM:

```
conda activate tpGBM
```

- Puis, pour installer cv2:

```
pip install opencv-python
```

- Et pour installer sklearn:

```
pip install scikit-learn
```

À CORRIGER DANS LE CODE

Il y a quelques coquilles dans le squelette de code. Elles sont regroupées ici (y compris celles déjà mentionnées précédemment).

La méthode `showDiff(sample)` fournie bug lors de son appel, corrigez les lignes:

```
# Applique le masque à la prédiction et au label
pred = pred & sample['mask']
label = sample['label'] & sample['mask']
```

Le code proposé dans le constructeur de `MSLD` pour la rotation de `line_detector` produit des aberrations d'interpolation utilisez plutôt:

```
import cv2
r = cv2.getRotationMatrix2D((l//2, l//2), angle, 1)
rotated_mask = cv2.warpAffine(line_detector, r, (l, l))
```

avec:

- `l`: la longueur de la ligne à détecter;
- `angle`: l'angle de rotation en degrés
- `line_detector`: la ligne horizontale à pivoter.

Dans la documentation de la fonction `roc(dataset)`, les variables de retour `fpr` et `tpr` sont inversées.

Cette méthode s'utilise en fait:

```
fpr, tpr, thresholds = self.roc(dataset)
```

Si vous le désirez et pour que ce soit plus clair, vous pouvez modifier les 2 dernières lignes de `roc(dataset)` pour que les nomenclatures soient correctes:

```
fpr, tpr, thresholds = roc_curve(y_true, y_pred)
return fpr, tpr, thresholds
```