COMPUTER VISION

# I. LIRE UNE IMAGE

### Contexte et objectif

En data science, quand vous travaillez sur des problématiques utilisant des images vous aurez besoin de savoir comment lire une image et comme pour des jeux de données classiques, comment faire des transformations en amont de l'utilisation d'un modèle.

L'objectif de ce notebook est de découvrir la librairie OpenCV, d'apprendre à lire une image et d'effectuer des modifications simples.

Commençons par la phase d'importation des packages.

* Importer les packages numpy, cv2 et pyplot.

import numpy as np

import matplotlib.pyplot as plt

import cv2

Nous allons commencer par apprendre à lire une image. La fonction imread de **cv2** permet de lire facilement une image que ce soit en couleur ou en noir et blanc. Il est important de savoir que cette fonction ne supporte que certains formats comme *.jpeg* ou *.png*. Vous trouverez une liste plus détaillée [ici](https://docs.opencv.org/4.1.2/d4/da8/group__imgcodecs.html)

Dans le premier argument de la fonction imread vous devez renseigner le chemin où se trouve votre image. Un deuxième argument est disponible. Il permet de déterminer si vous souhaitez lire l'image en couleur ou bien en noir et blanc. Pour cela il suffit de renseigner les arguments suivants :

* **cv2.IMREAD\_COLOR** : pour lire l'image en couleur
* **cv2.IMREAD\_GRAYSCALE** : pour lire l'image en noir et blanc

Exemple: utilisation de **imread**

img = cv2.imread("mon\_image.jpg") # pour lire mon\_image

* En utilisant la fonction imread de **cv2** lire, en couleur, l'image *street* qui est au format *.jpg* et la stocker dans une variable nommée img\_color
* En utilisant la fonction imread de **cv2** lire, en noir et blanc, l'image *street* qui est au format *.jpg* et la stocker dans une variable nommée img\_gray
* Observer les types de objets retournés

img\_gray = cv2.imread("street.jpg", cv2.IMREAD\_GRAYSCALE)

img\_color = cv2.imread("street.jpg", cv2.IMREAD\_COLOR)

print('le type de img\_gray est :', type(img\_gray))

print('le type de img\_color est :', type(img\_color))

Les objets retournés par la fonction imread de **cv2** sont des **array**.

* En utilisant la méthode shape afficher les dimensions de img\_color et img\_gray

print("Dimension de l'image en couleur : ", img\_color.shape)

print("Dimension de l'image en noir et blanc : ", img\_gray.shape)

Dimension de l'image en couleur : (150, 150, 3)

Dimension de l'image en noir et blanc : (150, 150)

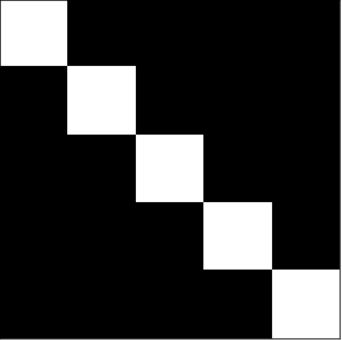
Comme vous l'avez remarqué avec la question précédente, une image est en fait un tableau de pixels. Selon que l'image est en couleur ou non la dimension des tableaux retournés est de 150x150 ou bien de 150x150x3. La raison vient de la manière dont est représentée l'image par la librairie cv2.

Chaque élément du tableau est un pixel. Dans une représentation en noir et blanc ce pixel est simplement un entier entre 0 et 255. Plus le nombre est proche de 0 plus le niveau d'activation du pixel est faible et donc de plus en plus noir. Inversement plus le nombre est proche de 255 plus le niveau d'activation est élevé et donc de plus en plus blanc.

Regardons un exemple simple pour mieux comprendre. Voici un tableau constitué de 25 pixels :

| **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- |
| 255 | 0 | 0 | 0 | 0 |
| 0 | 255 | 0 | 0 | 0 |
| 0 | 0 | 255 | 0 | 0 |
| 0 | 0 | 0 | 255 | 0 |
| 0 | 0 | 0 | 0 | 255 |

Voici le résultat après affichage :

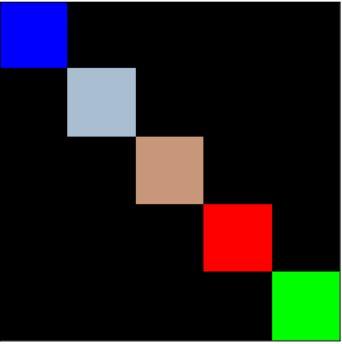


Dans le cas d'une représentation en couleur, chaque pixel est constitué d'un niveau de **bleu**, d'un niveau de **vert** et d'un niveau de **rouge**. Chacun des niveaux prend une valeur entre 0 et 255. Il s'agit d'un espace colorimétrique additif où les couleurs sont obtenues par une combinaison linéaire de valeurs de bleu, de vert et de rouge. Cet espace colorimétrique particulier est nommé **BGR** (Blue, Green, Red) et est utilisé par défaut par la librairie cv2. Il en existe d'autres comme **RGB** (Red, Green, Blue), qui fonctionne de manière similaire, ou encore **LAB**, **YCrCb**, **HSV** qui fonctionnent différemment.

Comme pour le cas en noir et blanc prenons un exemple simple pour mieux comprendre. Voici une représentation simple d'un tableau de dimensions 5 par 5 par 3 :

| **0** | **1** | **2** | **3** | **4** |
| --- | --- | --- | --- | --- |
| [0,0,255] | [0,0,0] | [0,0,0] | [0,0,0] | [0,0,0] |
| [0,0,0] | [170,190,210] | [0,0,0] | [0,0,0] | [0,0,0] |
| [0,0,0] | [0,0,0] | [200,150,120] | [0,0,0] | [0,0,0] |
| [0,0,0] | [0,0,0] | [0,0,0] | [255,0,0] | [0,0,0] |
| [0,0,0] | [0,0,0] | [0,0,0] | [0,0,0] | [0,255,0 |

Voici le résultat après affichage en utilisant l'espace de couleur **RGB**:



Pour afficher une image vous aurez besoin du package **pyplot**, de **matplotlib**, et de sa fonction imshow. Cette dernière vous permettra de visualiser l'image dans votre notebook.

Pour pouvoir lire une image en noir et blanc il est important d'utiliser l'argument **cmap**.

Exemple : Utilisation de **imshow**

plt.imshow(mon\_image, cmap = 'gray') #pour lire mon\_image en noir et blanc

plt.show();

plt.figure(figsize = (8,5))

plt.imshow(img\_gray, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();

* Mettre à 0 les pixels de la partie supérieure gauche de img\_gray et afficher

for i in range(50):

for j in range(50):

img\_gray[i,j] = 0

plt.figure(figsize = (8,5))

plt.imshow(img\_gray, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



Comme attendu et expliqué précédemment, la partie supérieure gauche est désormais en noir.

Egalement, il peut être utile de masquer l'image pour éliminer de l'information inutile dans l'image. Nous devons dans un premier temps créer un tableau mask composé uniquement de 0 et de même dimension que img\_gray.

Ensuite nous pouvons utiliser la fonction fillPoly pour déterminer un polygone à masquer dans l'image. Supposons que dans l'image img\_gray nous souhaitons garder un triangle qui commence dans le coin inférieur droit, se dirige vers le centre de l'image et suit un autre bord dans le coin inférieur gauche de l'image.

Un argument essentiel à renseigner dans l'utilisation de fillPoly est les sommets du polygone à masquer. Dans notre exemple, nous avons besoin des coordonnées du centre de l'image, du coin inférieur droit et du coin inférieur gauche.

Une fois que nous avons les coordonnées des sommets du polygone à masquer nous allons appliquer fillpoly sur mask pour garder à 0 les pixels du polygone et veiller à donner comme valeur 255 aux pixels que nous souhaitons conserver. Pour faire cela, nous devons utiliser l'argument **color** de fillpoly et le fixer à 255.

Exemple : utilisation de **fillpoly**

cv2.fillPoly(mon\_masque, sommets, color = 255)

Une fois que le masque est créé il ne nous reste plus qu'à "concaténer" en quelque sorte mask et img\_gray pour masquer la région contenant des informations inutiles. Nous allons utiliser la fonction bitwise\_and qui permet de sommer deux images d'une certaine manière : La fonction compare chacun des pixels et conservent le minimum des deux. Ainsi nous ne conservons que la partie intéressante de l'image.

Exemple : utilisation de **bitwise\_and**

cv2.bitwise\_and(mon\_image, mask) # pour masquer mon\_image

height, width = img\_gray.shape

sommets = [

(0, height),

(width / 2, height / 2),

(width, height),

]

sommets = np.array([sommets], np.int32)

mask = np.zeros\_like(img\_gray)

mask = cv2.fillPoly(mask, sommets, color = 255)

masked\_image = cv2.bitwise\_and(img\_gray, mask)

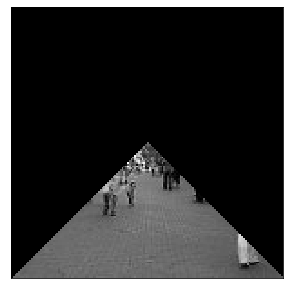
plt.figure(figsize = (8,5))

plt.imshow(masked\_image, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



* En utilisant imshow, afficher img\_color

plt.figure(figsize = (8,5))

plt.imshow(img\_color)

plt.xticks([])

plt.yticks([])

plt.show();



Vous remarquerez que les couleurs à l'affichage ne semblent pas naturelles. Le bâtiment à gauche apparaît en bleu ciel. La raison est simple : pyplot utilise l'espace colorimétrique RGB quand cv2 utilise l'espace colorimétrique BGR. Il est donc nécessaire avant d'utiliser imshow, de changer l'espace colorimétrique utilisé pour img\_color.

cv2, grâce à sa fonction cvtColor, permet de changer facilement l'espace colorimétrique utilisé pour représenter une image. L'attribut **code** permet de signifier le nouvel espace colorimétrique souhaité. Vous trouverez [ici](https://docs.opencv.org/master/d8/d01/group__imgproc__color__conversions.html#ga4e0972be5de079fed4e3a10e24ef5ef0) une liste des espaces de couleur disponibles. Les codes **cv2.COLOR\_BGR2GRAY** et **cv2.COLOR\_BGR2RGB** permettent respectivement de convertir l'image en noir et blanc et de convertir au format **RGB**.

Exemple : Utilisation de **cvtColor**

mon\_image = cv2.cvtColor(mon\_image, cv2.COLOR\_BGR2GRAY) #pour convertir mon\_image en noir et blanc

* Changer l'espace de couleur de **img\_color** de **BGR** en **RGB**
* Afficher de nouveau img\_color en utilisant imshow de pyplot

img\_color = cv2.cvtColor(img\_color, cv2.COLOR\_BGR2RGB)

plt.figure(figsize = (8,5))

plt.imshow(img\_color)

plt.xticks([])

plt.yticks([])

plt.show();



cv2 permet également de diminuer ou d'augmenter la taille de l'image si nécessaire. La fonction resize permet de le faire. L'attribut **dsize** de cette fonction permet de fixer la taille de l'image que vous souhaitez obtenir.

Cette fonction est utile si vous disposez d'images de tailles différentes et que vous souhaitez harmoniser ou bien si l'algorithme que vous utilisez est contraignant au niveau de la taille des images.

Exemple : Utilisation de **resize**

mon\_image = cv2.resize(mon\_image, dsize = (150,150))#pour modifier la taille de mon\_image en 150 par 150

fig = plt.figure(figsize = (12,12))

fig.add\_subplot(1,2,1)

plt.imshow(cv2.resize(img\_color, dsize = (75,75),interpolation = cv2.INTER\_AREA ))

plt.xticks([])

plt.yticks([])

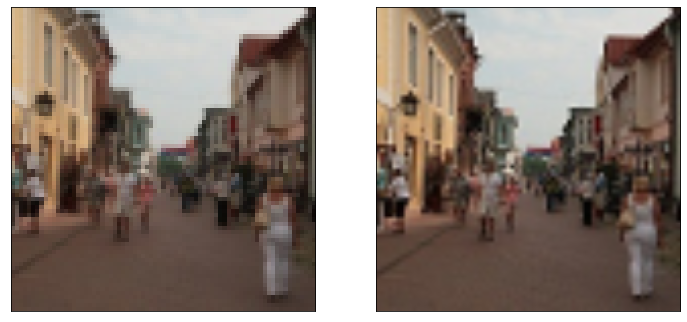
fig.add\_subplot(1,2,2)

plt.imshow(cv2.resize(img\_color, dsize = (1000,1000)))

plt.xticks([])

plt.yticks([])

plt.show();



Sur l'image de gauche nous remarquons que la netteté a fortement diminué. En effet, étant donné que nous avons divisé par 2 la taille de l'image, celle-ci est donc moins détaillée et apparaît plus floutée. Nous avons donc potentiellement perdu de l'information.

Sur l'image de droite nous avons fortement augmenté la taille de l'image. Nous nous apercevons que la netteté s'améliore un peu mais cela reste très peu relativement à la manière dont nous avons augmenté la taille et donc le poids de l'image. Travailler avec des images de taille aussi grande (1000x1000) peut très vite poser un problème de temps de calcul pour beaucoup d'algorithmes. Il est donc nécessaire de déterminer une taille d'image suffisamment importante pour distinguer de l'information pertinente mais sans pour autant en faire trop et prendre le risque de surcharger inutilement les calculs à venir.

### Conclusion

Dans ce notebook vous aurez appris à :

* Lire une image en utilisant la fonction imread de la librairie cv2.
* Comprendre comment l'image et les pixels associés sont représentés que ce soit en couleur ou en noir et blanc.
* Comprendre les notions d'espaces de couleur (RGB/BGR) qui ne sont pas les mêmes utilisés selon les librairies.
* Redimensionner une image à l'aide de la fonction resize de la librairie cv2.

# II. TRANSFORMATIONS SUR UNE IMAGE

### Contexte et objectif

En datascience, quand vous travaillez sur des problématiques utilisant des images vous aurez besoin de savoir comment faire des transformations sur vos images en amont de l'utilisation d'un modèle. Ces transformations ont pour but d'essentialiser l'information, de retravailler l'image pour obtenir une meilleure compréhension de la part de votre algorithme.

L'objectif de ce notebook est de découvrir des manipulations simples, des filtres et des techniques de seuil à utiliser avec la librairie OpenCV.

Commençons par la phase d'importation des packages.

* Importer les packages numpy, cv2, random et pyplot.

import numpy as np

import cv2

import matplotlib.pyplot as plt

import random

%matplotlib inline

* Lire, en couleur, l'image *street* qui est au format *.jpg*, la stocker dans une variable nommée **img\_color** et l'afficher.

img\_color = cv2.imread('street.jpg', cv2.IMREAD\_COLOR)

img\_color = cv2.cvtColor(img\_color, cv2.COLOR\_BGR2RGB)

plt.figure(figsize = (8, 5))

plt.imshow(img\_color)

plt.xticks([])

plt.yticks([])

plt.show();

Très souvent quand nous faisons face à des problématiques de classification d'images par exemple, nous avons besoin de faire ce que nous appelons de l'augmentation de données.

L'œil humain, que je lui donne un immeuble ou une rue à l'endroit ou à l'envers, sera toujours capable de faire la différence et de les reconnaître. Concernant des algorithmes complexes comme les réseaux de neurones ce n'est pas aussi simple. Donnez-lui 15000 images d'immeubles à l'endroit, il saura reconnaître un immeuble à l'endroit. Cependant donnez-lui une image d'immeuble tournée à l'envers ou bien à 90 degrés par exemple, il aura du mal à reconnaître l'immeuble. L'augmentation de données consiste, si on reprend notre exemple, à modifier parmi les 15000 images certaines, comme par exemple, les tourner à 90 degrés ou bien les retourner verticalement dans le but d'améliorer les performances d'un modèle complexe.

La librairie OpenCV (cv2) offre des fonctions simples d'utilisation pour effectuer des modifications de base comme par exemple effectuer une rotation de l'image.

Pour tourner une image selon un angle prédéfini vous aurez besoin de deux fonctions de cv2 :

* cv2.getRotationMatrix2D : permet de déterminer la matrice transformation nécessaire pour effectuer la rotation. Vous devez renseigner 3 éléments : les coordonnées du centre de l'image, l'angle de rotation et le niveau d'échelle.
* cv2.warpAffine : pour appliquer la transformation affine à l'image. Vous devez renseigner 3 éléments : l'image sur laquelle effectuer la transformation, la matrice de transformation et enfin la longueur et la largeur de l'image en sortie.

Pour effectuer une rotation sans changer l'échelle ou la taille de l'image il suffit de fixer à 1 le niveau d'échelle et dans la transformation affine renseigner les longueur et largeur d'origine.

Exemple : Rotation d'une image

long, large = mon\_image.shape[:2]

M = cv2.getRotationMatrix2D((int(long/2), int(large/2)), angle, 1)

mon\_image = cv2.warpAffine(mon\_image, M, (long, large)) #pour tourner à 45 degrés mon\_image

* Tourner **img\_color** à 90 degré, stocker l'image dans une variable nommée **img\_rotate** et l'afficher.

def rotation(img, angle):

long, large = img.shape[:2]

M = cv2.getRotationMatrix2D((int(long/2), int(large/2)), angle, 1)

img = cv2.warpAffine(img, M, (long, large))

return img

img\_rotate = rotation(img\_color,90)

plt.figure(figsize = (8,5))

plt.imshow(img\_rotate)

plt.xticks([])

plt.yticks([])

plt.show();



Une modification simple à réaliser est également ce que nous appelons un flip vertical ou horizontal. Il s'agit de retourner verticalement ou horizontalement l'image.

La fonction flip de cv2 permet de le faire très facilement. Il suffit de renseigner l'image que vous souhaitez flip et 0 (vertical) ou 1 (horizontal).

Exemple : flip d'une image

mon\_image = cv2.flip(mon\_image,1) #pour flip horizontalement mon\_image

* Flip verticalement img\_color et l'afficher

plt.figure(figsize = (8, 5))

plt.imshow(cv2.flip(img\_color, 0))

plt.xticks([])

plt.yticks([])

plt.show();

Vous pouvez également zoomer l'image. Il s'agit simplement d'isoler une partie de l'image comme par exemple un immeuble. Il suffit de redimensionner l'image en renseignant la zone à isoler.

Etant donné que nous allons redimensionner l'image nous nous servirons donc de resize qui est la fonction dédiée pour ça dans cv2.

Pour pouvoir zoomer il suffit de renseigner les coordonnées de la zone sur laquelle on souhaite effectuer un zoom. Ensuite il suffit de sous-sélectionner les pixels de la zone souhaitée. Voici un exemple si on souhaite isoler la partie en bas à gauche de l'image :

long, larg = img\_color.shape[:2]

long\_z = int(0.5\*long)

larg\_z = int(0.5\*larg)

long\_deb = 80

larg\_deb = 80

img\_zoom = img\_color[long\_deb:long\_deb + long\_z, larg\_deb:larg\_deb + larg\_z, :] #pour isoler le coin gauche de img\_color

Il est important d'utiliser la fonction resize pour redimensionner l'image aux dimensions de la partie zoomée

* Zoomer sur la partie supérieure gauche de l'image et afficher

long, larg = img\_color.shape[:2]

long\_z = int(0.5\*long)

larg\_z = int(0.5\*larg)

long\_deb = 0

larg\_deb = 0

img\_zoom = img\_color[long\_deb:long\_deb + long\_z, larg\_deb:larg\_deb + larg\_z, :]

img\_zoom = cv2.resize(img\_zoom, (long\_z, larg\_z))

plt.figure(figsize = (8,5))

plt.imshow(img\_zoom)

plt.xticks([])

plt.yticks([])

plt.show();



OpenCV (cv2) en plus de modifications de base permet également, à l'aide de différentes fonctions, d'effectuer des filtrages sur l'image. Les filtres d'image peuvent être utilisés pour réduire la quantité de bruit dans une image et pour améliorer les bords d'une image. Il s'agit de supprimer les bords de faible intensité. Il peut également être utilisé pour masquer les détails d'une image et donc à terme améliorer les performances d'un algorithme de classification par exemple.

Prenons un exemple simple de filtre : la moyenne. Pour ce faire, on effectue une convolution de l'image avec un filtre à boîte normalisée. Il prend simplement la moyenne de tous les pixels sous la zone du noyau et remplace l'élément central par cette moyenne.

Le produit de convolution est un produit particulier qui s'effectue avec des matrices. D'un côté nous avons une matrice d'entrée (notre image) de l'autre nous avons un noyau de convolution (notre filtre). Pour calculer le produit de convolution, il est indispensable que les deux matrices aient les mêmes dimensions. Les étapes du produit de convolution sont les suivantes :

* Produit terme à terme des deux matrices.
* Somme des produits.

Dans l'animation suivante nous illustrons le produit de convolution entre une matrice d'entrée et un noyau de convolution de dimensions 3x3.

* Pour interagir avec la figure, cliquez sur le bouton Next pour aller à l'étape suivante et Reset pour redémarrer l'animation.

from interaction\_cnn import show\_one\_operation\_conv

show\_one\_operation\_conv()

Comme nous avons pu le voir précédemment, une image est une matrice de pixels. Nous pouvons lui appliquer un produit de convolution et donc un filtre. Pour effectuer cette opération sur une image entière, il faut d'abord la découper en **plusieurs matrices de mêmes dimensions que le noyau de convolution que nous voulons utiliser**. Ensuite, nous pouvons effectuer le produit de convolution entre chaque matrice et le noyau.

Dans la figure suivante nous illustrons la convolution d'une image de dimensions 4x4 par un noyau de convolution de dimensions 3x3:

* Etape 1: Découpage de l'image en plusieurs matrices convolables. Chaque matrice a une couleur différente.
* Etape 2: Produits de convolutions entre chaque matrice et le noyau.
* Etape 3: Activation et concaténation des produits en une nouvelle matrice.
* Executer la cellule suivante pour afficher l'interaction

from interaction\_cnn import show\_operation\_conv

show\_operation\_conv()

Pour appliquer un filtre nommé averaging ou moyenne, on peut utiliser la fonction blur de **cv2**. Elle détermine directement le noyau de convolution nécessaire et renvoie l'image filtrée.

Pour utiliser la fonction blur correctement il est nécessaire de renseigner l'image que l'on souhaite filtrer et également la dimension du filtre souhaité. Plus la dimension est élevée, plus vous risquez de trop simplifier l'image et de perdre en information. L'objectif d'un filtre reste de se débarrasser des informations non nécessaires pour ne garder que le plus pertinent de l'image.

L'argument **ksize** permet de renseigner la dimension souhaitée du noyau de convolution utilisé. Vous trouverez des informations plus détaillées sur la manière dont est calculé le noyau de convolution utilisé dans la fonction blur [ici](https://docs.opencv.org/3.4/d4/d86/group__imgproc__filter.html#ga8c45db9afe636703801b0b2e440fce37).

Exemple : utilisation de **blur**

mon\_filtre = cv2.blur(mon\_image,ksize = (3,3))#pour filtrer mon\_image avec un noyau de convolution de dimension 3x3

plt.figure(figsize = (8, 5))

plt.imshow(cv2.blur(img\_color, ksize = (3, 3)))

plt.xticks([])

plt.yticks([])

plt.show();



Pour comprendre d'autant plus l'intérêt des filtres, nous allons créer une image **img\_sp\_noise** dans laquelle nous allons ajouter volontairement ce qu'on appelle du bruit poivre et sel. Il s'agit simplement de transformer aléatoirement certains pixels de couleur, en pixel noir ou blanc.

* Exécuter une fois la cellule de code suivante

salt\_value = 20

noise = np.random.randint(salt\_value+1, size=(150, 150))

img\_sp\_noise = img\_color.copy()

#---------- Poivre---#

indexe = np.where(noise == 0)

A = indexe[0]

B = indexe[1]

img\_sp\_noise[A,B,:] = 0

#---------- Sel---------#

indexe = np.where(noise == salt\_value)

A = indexe[0]

B = indexe[1]

img\_sp\_noise[A,B,:] = 255

plt.figure(figsize = (8,5))

plt.imshow(img\_sp\_noise)

plt.xticks([])

plt.yticks([])

plt.show();



Nous avons donc maintenant une image contenant aléatoirement des pixels blanc ou noir, ce qui nuit à la qualité de l'image et aux informations qu'elle contient. Il est important, si on souhaite par la suite avoir une bonne compréhension, d'utiliser des méthodes de filtrage pour essayer d'améliorer la qualité de l'image.

Une méthode simple et efficace consiste à utiliser la fonction medianBlur de cv2. Elle permet de corriger efficacement le bruit poivre et sel. Cette fonction calcule la médiane de tous les pixels sous la fenêtre du noyau et le pixel central est remplacé par cette valeur médiane. Vous trouverez plus d'informations sur la méthode de calcul du noyau de convolution associé à ce filtre [ici](https://docs.opencv.org/3.4/d4/d86/group__imgproc__filter.html#ga564869aa33e58769b4469101aac458f9)

L'argument **ksize** permet de préciser la dimension du noyau de convolution utilisé pour le filtre.

Exemple : utilisation de **medianBlur**

mon\_filtre = cv2.medianBlur(mon\_image, ksize = 3) #pour filtrer mon\_image avec un noyau de convolution de dimension 3x3

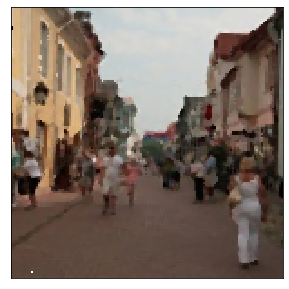
plt.figure(figsize = (8,5))

plt.imshow(cv2.medianBlur(img\_sp\_noise, ksize = 3))

plt.xticks([])

plt.yticks([])

plt.show();



Un autre filtre, GaussianBlur, est lui utilisé pour supprimer le bruit gaussien présent dans l'image. Comme pour **medianBlur** son utilisation est simple. Comme pour un filtre classique un noyau de convolution est préalablement calculé par la fonction aux dimensions demandées. Ensuite il s'effectue un produit de convolution entre l'image et le noyau de convolution calculé pour filtrer l'image.

L'argument **ksize** permet de préciser la dimension du noyau de convolution utilisé pour le filtre.

Exemple : utilisation de **GaussianBlur**

mon\_filtre = cv2.GaussianBlur(mon\_image, ksize = (3,3), sigmaX = 0) #pour filtrer et éliminer le bruit gaussien de mon\_image

* En utilisant GaussianBlur, filtrer **img\_color** et afficher

plt.figure(figsize = (8, 5))

plt.imshow(cv2.GaussianBlur(img\_color, ksize = (3, 3), sigmaX = 0))

plt.xticks([])

plt.yticks([])

plt.show();

En plus des filtres, il existe des techniques dites de seuil s'effectuant sur des images en niveaux de gris. Si la valeur d'un pixel est supérieure à une valeur seuil, on lui attribue une valeur (qui peut être blanche), sinon on lui attribue une autre valeur (qui peut être noire). La fonction utilisée est **threshold** de **cv2**. Le premier argument est l'image source, qui doit être une image en niveaux de gris. Le deuxième argument est la valeur de seuil qui est utilisée pour classer les valeurs des pixels. Le troisième argument est le maxVal qui représente la valeur à donner si la valeur des pixels est supérieure (parfois inférieure) à la valeur de seuil. OpenCV fournit différents styles de seuillage et il est décidé par le quatrième paramètre de la fonction. Les différents types sont :

* **cv2.THRESH\_BINARY** : Si l'intensité du pixel est supérieure au seuil fixé, la valeur est fixée à 255, sinon à 0 (noir).
* **cv2.THRESH\_BINARY\_INV** : Le même principe que cv2.THRESH\_BINARY mais inversé.
* **cv2.THRESH\_TRUNC** : Si la valeur de l'intensité du pixel est supérieure au seuil, elle est tronquée au seuil. Les valeurs des pixels sont fixées de manière à être identiques au seuil. Toutes les autres valeurs restent les mêmes.
* **cv2.THRESH\_TOZERO** : L'intensité des pixels est fixée à 0, pour toutes les intensités de pixels, inférieure à la valeur seuil.
* **cv2.THRESH\_TOZERO\_INV** : Le même principe que cv2.THRESH\_TOZERO mais inversé.

La fonction renvoie deux éléments : le seuil utilisé et l'image après utilisation de la technique de seuillage.

Exemple : utilisation de **threshold**

seuil,img\_seuillage = cv2.threshold(mon\_image,150,255,type = cv2.THRESH\_BINARY) #pour retravailler mon\_image avec une technique dite de seuillage

* Lire, en niveaux de gris, l'image building qui est au format .jpg, la stocker dans une variable nommée img\_gray
* En utilisant la fonction threshold et le style de seuillage **cv2.THRESH\_BINARY**, appliquer un seuillage à img\_gray en fixant le seuil à 115 et la **maxVal** à 255
* Afficher l'image après application du seuillage

img\_gray = cv2.imread('building.jpg', cv2.IMREAD\_GRAYSCALE)

filtre = cv2.GaussianBlur(img\_gray, ksize = (3,3), sigmaX = 0)

seuil, img\_seuil = cv2.threshold(img\_gray, 115, 255, type = cv2.THRESH\_BINARY)

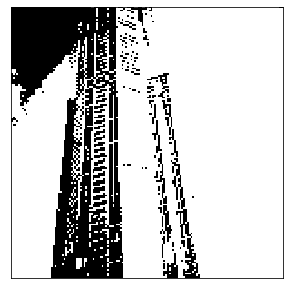
plt.figure(figsize = (8, 5))

plt.imshow(img\_seuil, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



Par l'utilisation de cette technique, nous avons réussi à mettre en avant la structure de l'immeuble qui apparaît plus nettement. Nous pouvons encore améliorer le résultat en utilisant une autre fonction.

Avec la fonction threshold, nous avons utilisé une valeur globale comme valeur seuil. Mais elle peut ne pas être bonne dans toutes les conditions où l'image présente des conditions d'éclairage différentes dans des zones différentes. Dans ce cas, nous avons opté pour un seuillage adaptatif. L'algorithme calcule le seuil pour de petites régions de l'image. Ainsi, nous obtenons différents seuils pour différentes régions de la même image et cela nous donne de meilleurs résultats pour des images avec un éclairage variable.

La fonction adaptiveThreshold de cv2 permet de faire cela facilement. Elle contient 3 paramètres supplémentaires :

* **adaptiveMethod** : La méthode adaptative détermine le mode de calcul de la valeur seuil. Deux choix sont possibles : cv2.ADAPTIVE\_THRESH\_MEAN\_C (Moyenne des valeurs de la zone de voisinage) ou cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C (Somme pondérée gaussienne des valeurs des pixels voisins).
* **blockSize** : décide de la taille de la zone de voisinage.
* **C** : Une valeur constante qui est soustraite de la moyenne ou de la somme pondérée des pixels du voisinage.

Exemple : utilisation de **adaptiveThreshold**

img\_seuillage = cv2.adaptiveThreshold(mon\_image,255,

adaptiveMethod = cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,

thresholdType = cv2.THRESH\_BINARY,

blockSize = 3,

C = 2)

* En utilisant la fonction adaptiveThreshold, appliquer un seuillage adaptable à **img\_gray** avec **adaptiveMethod**, **Block Size** et C respectivement égal à **cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C**, 3 et 2
* Afficher l'image après application du seuillage

filtre = cv2.GaussianBlur(img\_gray, ksize = (3,3), sigmaX = 0)

img\_seuillage\_2 = cv2.adaptiveThreshold(filtre,255,

adaptiveMethod = cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,

thresholdType = cv2.THRESH\_BINARY,

blockSize = 3,

C = 2)

plt.figure(figsize = (8, 5))

plt.imshow(filtre, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



Nous obtenons un meilleur rendu que lors de notre première tentative avec un seuil global. Ces modifications sont utiles si nous souhaitons déceler des structures particulières toujours dans le but d'améliorer par exemple les performances d'un algorithme de classification.

### Conclusion

Dans ce notebook vous aurez appris à :

* Effectuer des modifications simples sur des images à l'aide fonctions de la librairie cv2.
* Effectuer des filtrages simples à l'aide de fonctions de la librairie cv2 comme medianBlur pour corriger le bruit présent dans une image.
* Utiliser des méthodes de seuil pour déceler des structures particulières.

# III. TRANSFORMATIONS AVANCÉES SUR UNE IMAGE

### Contexte et objectif

Nous avons déjà vu dans les notebooks précédents comment lire, modifier et filtrer facilement une image avec la librairie cv2 (OpenCV). Nous pouvons aller encore plus loin en explorant des méthodes d'analyse d'image plus avancées permettant notamment d'améliorer potentiellement les performances d'un algorithme de classification.

L'objectif de ce notebook est de découvrir des algorithmes de détection de formes et des méthodes de transformation morphologique comme l'érosion et la dilatation.

Commençons par la phase d'importation des packages.

* Importer les packages numpy, cv2 et pyplot

import numpy as np

import cv2

import matplotlib.pyplot as plt

%matplotlib inline

Dans le notebook précédent nous avons vu qu'il était possible de supprimer du bruit dans une image comme les bruits poivre et sel ou gaussien, et également par des techniques de seuil de changer la perspective de l'image.

Nous pouvons aller plus loin, pour ne pas simplement corriger des bruits éventuels, mais aussi détecter des formes dans l'image. Nous allons utiliser les fonctions GaussianBlur et Canny de cv2. Il est important dans un premier temps d'éliminer le bruit gaussien qui peut être présent pour obtenir ensuite une meilleure détection de formes.

Comme nous l'avons vu, la fonction GaussianBlur fonctionne comme un filtre classique avec l'utilisation d'un noyau de convolution de dimension renseignée au préalable.

La fonction Canny de cv2 est l'implémentation de l'algorithme de John Canny développé en 1986. Il a été conçu pour détecter dans une image les contours caractéristiques. Pour chaque pixel ce dernier calcule l'intensité du contour selon une méthode spécifique. Ensuite, pour chaque pixel l'algorithme décide si oui ou non l'intensité est suffisamment forte pour considérer le pixel comme un contour important. Cette décision est prise à l'aide de deux seuils préalablement définis :

* Si l'intensité est inférieure au seuil minimal, le pixel est rejeté
* Si l'intensité est supérieure au seuil maximal, le pixel est accepté comme contour
* Si l'intensité est entre les deux seuils, le pixel est accepté comme contour s'il est connecté à un pixel déjà accepté

Cette procédure de sélection est ce qu'on appelle un seuillage à hystérésis. Vous trouverez des précisions sur l'algorithme Canny ici.

Exemple : utilisation de **GaussianBlur** et **Canny**

filtre = cv2.GaussianBlur(mon\_image,(3,3),0) #pour supprimer le bruit gaussien dans mon\_image

edges = cv2.Canny(filtre,100,200) #pour détecter les formes de mon\_image

* Dans un premier temps, lire en couleur, l'image *building* qui est au format *.jpg*, la stocker dans une variable nommée building\_color
* En utilisant la fonction GaussianBlur de cv2, éliminer le bruit gaussien dans building
* En utilisant la fonction Canny de cv2, détecter les formes de building en fixant les arguments **minVal** et **maxVal** respectivement à 100 et 200
* Afficher l'image filtrée

building\_color = cv2.imread('building.jpg', cv2.IMREAD\_COLOR)

building\_color = cv2.cvtColor(building\_color, cv2.COLOR\_BGR2RGB)

filtre = cv2.GaussianBlur(building\_color,(3,3),0)

edges = cv2.Canny(filtre,100,200)

plt.figure(figsize = (8,5))

plt.imshow(edges, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();

Une fois l'image filtrée à l'aide de l'algorithme de Canny, on peut utiliser la fonction HoughLinesP pour déterminer les coordonnées des lignes pertinentes de l'image. Cette fonction renvoie un array Si vous souhaitez avoir plus de détails sur la théorie mathématique derrière cette fonction, vous trouverez des informations [ici](https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html).

Différents arguments sont utilisés dans la fonction :

* **dst**: Sortie du détecteur de formes. Il devrait s'agir d'une image en niveaux de gris (bien qu'il s'agisse en fait d'une image binaire)
* **lignes** : Un vecteur qui va stocker les paramètres (x*{début}, y*{début}, x*{fin}, y*{fin}) des lignes détectées
* **rho** : La résolution du paramètre r en pixels.
* **theta** : La résolution du paramètre \theta en radians.
* **threshold**: Le nombre minimum d'intersections pour "détecter" une ligne
* **minLinLength**: Le nombre minimum de points qui peuvent former une ligne. Les lignes ayant moins que ce nombre de points ne sont pas prises en compte.
* **maxLineGap**: L'écart maximum entre deux points à considérer dans une même ligne.

Exemple : utilisation de **HoughLinesP**

lines = cv2.HoughLinesP(edges,rho=3,theta=np.pi / 20,threshold=100,minLineLength=0,maxLineGap= 20)

* Appliquer la fonction HoughLinesP à edges et stocker le résultat dans une variable lines.

lines = cv2.HoughLinesP(edges, rho=3, theta=np.pi/20, threshold = 100, minLineLength = 0, maxLineGap = 20)

Une fois que nous avons stocké les lignes pertinentes, nous allons les afficher en rouge dans l'image. Pour cela nous avons besoin dans un premier temps de créer un tableau rempli de 0 de même dimension que l'image building\_color que nous appellerons line\_img.

La fonction line de cv2 permet de tracer les lignes dans l'image line\_img. Il suffit de se resservir des coordonnées stockées dans la variable lines. Elle contient pour chaque ligne les coordonnées du point de départ et du point de fin. L'argument **color** permet de donner une couleur spécifique aux lignes que nous souhaitons mettre en valeur dans l'image. L'argument **thickness** permet de gérer l'épaisseur des lignes.

Exemple : utilisation de **lines**

for line in lines:

for x1, y1, x2, y2 in line:

cv2.line(mon\_image, (x1, y1), (x2, y2), color = [255, 0, 0], thickness=3)

Une fois que nous avons défini les lignes dans l'image line\_img il suffit de 'concaténer' les deux images line\_img et building\_color pour faire apparaître ces lignes dans l'image d'origine. Pour cela nous allons utiliser la fonction addWeighted. Elle permet de calculer la somme pondérée de deux images.

Pour utiliser la fonction il faut mettre dans l'ordre la première image, le poids associé, la seconde image, le poids associé et enfin un scalaire à ajouter en plus qu'on va mettre à 0.

Exemple : utilisation de **addWeighted**

cv2.addWeighted(mon\_image\_1, 0.8, mon\_image\_2, 1.0, 0.0)

* Créer line\_img un tableau composé de zeros avec les mêmes dimensions que building\_color.
* En utilisant la fonction **line**, déterminer en rouge les lignes détectées précedemment dans line\_img.
* En utilisant la fonction addWeighted, sommer les images line\_img et building\_color et stocker le résultat dans une variable nommée building\_lines.
* Afficher le résultat.

line\_img = np.zeros((building\_color.shape[0],building\_color.shape[1],3), dtype=np.uint8)

for line in lines:

for x1, y1, x2, y2 in line:

cv2.line(line\_img, (x1, y1), (x2, y2), color = [255, 0, 0], thickness=3)

building\_lines = cv2.addWeighted(building\_color, 0.8, line\_img, 1.0, 0.0)

plt.figure(figsize = (8,5))

plt.imshow(building\_lines)

plt.xticks([])

plt.yticks([])

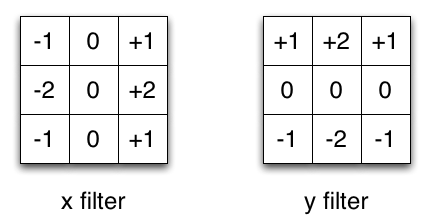
plt.show();



On peut également utiliser des filtres avancés pour détecter des formes dans l'image. Le filtre de Sobel permet notamment de détecter les bords verticaux ou horizontaux d'une image. Comme pour un filtre classique, le filtre Sobel utilise un noyau de convolution particulier qui lui permet de détecter les formes horizontales ou verticales dans l'image.

OpenCV met à disposition une fonction Sobel qui permet d'appliquer facilement le filtre à une image. Les arguments **dx** et **dy** permettent de déterminer si on souhaite mettre en évidence les bords verticaux ou horizontaux de l'image. Par exemple, la combinaison (dx, dy) = (1,0) mettra en évidence les bords verticaux de l'image. L'inverse vous donnera les bords horizontaux. L'argument **ddepth** doit être égal à **cv2.CV\_64F** pour une question technique liée aux types d'entiers utilisés.

Suivant les combinaisons **dx**, **dy** ((1,0) ou (0,1)) le noyau de convolution utilisé n'est pas le même. Voici les deux noyaux utilisés suivant les cas :



Le noyau de gauche est utilisé pour mettre en évidence les bords verticaux de l'image tandis que celui de droite est utilisé pour mettre en évidence les bords horizontaux.

Le filtre s'utilise sur des images en **niveaux de gris**.

Exemple : utilisation de **Sobel**

sobel = cv2.Sobel(mon\_image, ddepth = cv2.CV\_64F, dx = 1, dy = 0) #pour détecter les bords verticaux dans mon\_image

* Dans un premier temps, lire en niveaux de gris, l'image building qui est au format .jpg, la stocker dans une variable nommée street\_gray
* En utilisant Sobel, filtrer street\_gray de sorte à faire apparaître les bords verticaux à l'intérieur de l'image

street\_gray = cv2.imread('street.jpg', cv2.IMREAD\_GRAYSCALE)

sobel = cv2.Sobel(street\_gray, ddepth = cv2.CV\_64F, dx = 1, dy = 0)

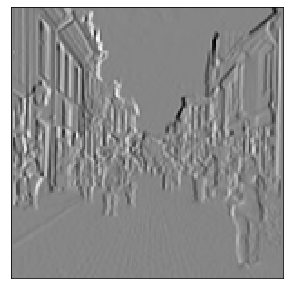
plt.figure(figsize = (8,5))

plt.imshow(sobel, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



On constate que le filtre a bien détecté les bords verticaux dans l'image. Ce type de

filtre est utile pour mieux comprendre les structures présentes dans une image comme une maison par exemple.

On peut aller un peu plus loin avec des filtres plus avancés comme le Laplacian. Il utilise la mise en évidence des discontinuités de niveaux de gris dans une image et tente de désaccentuer les régions dont les niveaux de gris varient lentement. Le résultat de cette opération est la production d'images dont les bords sont grisâtres et qui présentent d'autres discontinuités sur un fond sombre. Cela produit des bords intérieurs et extérieurs dans une image.

Une différence entre le Laplacian et les autres filtres comme Sobel est que, contrairement aux autres opérateurs, le laplacien ne prend pas de bords dans une direction particulière comme vertical ou horizontal.

L'argument **ddepth** doit être égal à **cv2.CV\_64F** pour une question technique liée aux types d'entiers utilisés.

Exemple : utilisation de **Laplacian**

laplacian = cv2.Laplacian(mon\_image, ddepth = cv2.CV\_64F)

* En utilisant la fonction Laplacian, filtrer street\_gray et observer le résultat

laplacian = cv2.Laplacian(street\_gray, ddepth = cv2.CV\_64F)

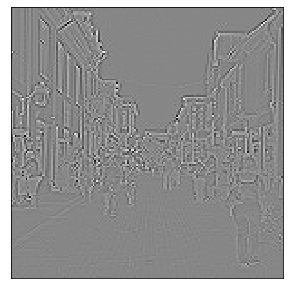
plt.figure(figsize = (8,5))

plt.imshow(laplacian, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



Le filtrage paraît plus affiné qu'avec **Sobel** et permet de distinguer des structures plus complexes au sein de l'image.

Ces filtres sont déterminants dans la lecture d'images pour déterminer des caractéristiques clés. Des algorithmes complexes comme les réseaux de neurones CNN utilisent un très grand nombre de filtres pour extraire au fur et à mesure des informations clés de plus en plus complexes leur permettant d'apprendre de manière efficace et par la suite faire de puissantes prédictions.

On peut également utiliser ce qu'on appelle des méthodes de transformations morphologiques pour détecter des structures pertinentes et éliminer celles non pertinentes pour la compréhension de l'image. Les deux méthodes de base très utilisées sont la dilatation et l'érosion.

Une première transformation morphologique simple à appliquer est l'érosion. Elle est utilisée pour faire disparaître les limites des objets de premier plan et également pour éliminer les petits bruits blancs des images. L'érosion peut également être utilisée pour détacher deux images reliées entre elles.

Elle s'utilise facilement avec la fonction erode de cv2. Il est nécessaire avant de définir un noyau de convolution qui peut avoir n'importe quelle forme ou taille. On définit le point d'ancrage du noyau comme le centre du noyau.

Lorsque le noyau balaie l'image, nous calculons la valeur minimale de la fenêtre recouverte par le noyau et nous remplaçons le point d'ancrage par cette valeur.

L'érosion a pour conséquence d'affiner l'image et notamment les zones claires. Pour comprendre et visualiser les effets de l'érosion nous allons utiliser erode sur une image d'araignée.

Exemple : utilisation de **erode**

kernel = np.ones((3,3),np.uint8) # création du noyau

erosion = cv2.erode(mon\_image, kernel) # erosion de mon\_image

* Lire, en niveaux de gris, l'image *spider* au format *.jpeg* et la stocker dans une variable nommée spider\_gray
* Redimensionner l'image en taille 150x150 à l'aide de resize
* Inverser les nuances de gris pour mettre en blanc les formes au premier plan
* Filtrer l'image avec un filtre gaussien et la stocker dans une variable nommée filtre
* Effectuer 3 érosions à l'aide de la fonction erode de cv2 en utilisant 3 noyaux de convolutions composés uniquement de 1 mais de dimensions respectives (3,3), (5,5) et (7,7)
* Afficher le résultat pour chaque érosion ainsi que l'image d'origine

spider\_gray = cv2.imread('spider.jpeg', cv2.IMREAD\_GRAYSCALE)

spider\_gray = cv2.resize(spider\_gray, (150,150))

spider\_gray = 255 - spider\_gray

filtre = cv2.GaussianBlur(spider\_gray, ksize = (3,3), sigmaX = 0)

fig = plt.figure(figsize = (12,12))

fig.add\_subplot(1,4,1)

plt.imshow(spider\_gray, cmap = 'gray')

plt.xticks([])

plt.yticks([])

fig.add\_subplot(1,4,2)

kernel\_1 = np.ones((3,3),np.uint8)

erosion\_1 = cv2.erode(filtre, kernel\_1)

plt.imshow(erosion\_1, cmap = 'gray')

plt.xticks([])

plt.yticks([])

fig.add\_subplot(1,4,3)

kernel\_2 = np.ones((5,5),np.uint8)

erosion\_2 = cv2.erode(filtre, kernel\_2)

plt.imshow(erosion\_2, cmap = 'gray')

plt.xticks([])

plt.yticks([])

fig.add\_subplot(1,4,4)

kernel\_3 = np.ones((7,7),np.uint8)

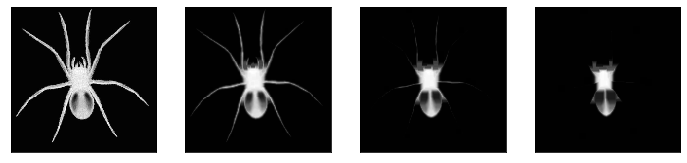
erosion\_3 = cv2.erode(filtre, kernel\_3)

plt.imshow(erosion\_3, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



L'érosion permet d'isoler le corps de l'araignée et d'éliminer la présence des pattes. C'est très utile pour isoler des parties spécifiques que l'on souhaite identifier et étudier.

Une autre transformation morphologique très connue est la dilatation. Elle produit l'effet inverse de l'érosion et fonctionne de manière identique.

Il est nécessaire avant de définir un noyau de convolution qui peut avoir n'importe quelle forme ou taille. On définit le point d'ancrage du noyau comme le centre du noyau. Lorsque le noyau balaie l'image, nous calculons la valeur maximale de la fenêtre recouverte par le noyau et nous remplaçons le point d'ancrage par cette valeur.

La dilatation est facilement utilisable avec la fonction dilate de cv2.

Exemple : utilisation de **dilate**

kernel = np.ones((3,3),np.uint8) # création du noyau

dilatation = cv2.dilate(mon\_image, kernel) # erosion de mon\_image

* Effectuer une dilatation de l'image erosion\_2 à l'aide d'un noyau de convolution composé de 1 de taille (5,5)
* Afficher l'image dilatée

fig = plt.figure(figsize = (12,12))

fig.add\_subplot(1,2,1)

kernel = np.ones((5,5),np.uint8)

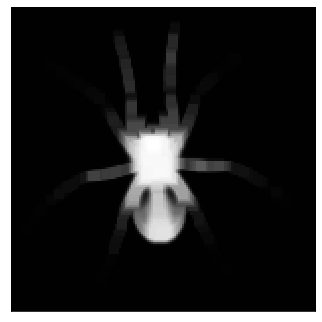
dilation = cv2.dilate(erosion\_2,kernel)

plt.imshow(dilation, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



* Effectuer la différence entre l'image d'origine spider\_gray et l'image dilatée **dilation**
* Afficher le résultat

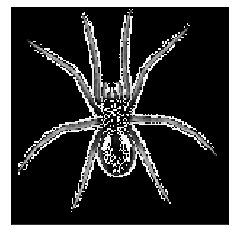
diff = spider\_gray - dilation

plt.imshow(diff, cmap = 'gray')

plt.xticks([])

plt.yticks([])

plt.show();



On peut voir qu'en dilatant le corps de l'araignée et en soustrayant à l'image d'origine, on obtient les contours de l'araignée. Ces techniques de transformations morphologiques permettent de détecter ou d'isoler des structures pertinentes au sein d'une image.

### Conclusion

Dans ce notebook vous aurez appris à :

* Détecter des formes complexes en utilisant des algorithmes avancés comme Canny.
* Effectuer des filtrages avancés comme Sobel ou Laplacian pour détecter des formes verticales ou horizontales dans une image.
* Effectuer des transformations morphologiques simples pour éroder ou dilater une image.