

**République Algérienne Démocratique et Populaire**  
**Ministère de l'Enseignement Supérieur et de la Recherche Scientifique**

Université Mohammed Seddik BenYahia -Jijel-

Faculté Des Sciences Exactes et Informatique

Département d'Informatique

Specialite : M1 ILM



Rapport de TP3 TNMI

---

Détection des contour utilisant les deux méthodes (gradient et laplacien)

---

- Birouk Mohammed Islam
- Chettab Mohcine

**L'Encadreur :**

- Zahir Mahrouk

---

---

## **Abstract :**

La détection des contours dans les images est une tâche essentielle en traitement d'images, mais la présence de bruit peut réduire considérablement sa précision. Ce programme répond à ce problème en mettant en œuvre une chaîne de traitement combinant l'ajout de bruit, l'utilisation de filtres de débruitage, et des techniques robustes de détection des contours comme Sobel . En simulant des environnements bruités et en appliquant des algorithmes résistants au bruit, cette solution améliore la fiabilité de la détection des contours dans des scénarios réels.

## **Programme et Description :**

Ce programme permet de détecter les contours d'une image en utilisant des algorithmes avancés de traitement d'image. Il offre des fonctionnalités pour charger une image, ajouter différents types de bruit (comme le bruit gaussien ou le bruit impulsionnel) et appliquer des techniques de filtrage et de détection de contours (par exemple, Sobel, Prewitt). Son objectif principal est de faciliter l'exploration et l'analyse des contours, même en présence de bruit, afin de mieux comprendre les caractéristiques des objets présents dans l'image.

## **Utilisation des Bibliothèques et Installation :**

Ce programme repose sur plusieurs bibliothèques Python pour effectuer le traitement d'images, la gestion de l'interface utilisateur, et d'autres fonctionnalités essentielles.

## **Pourquoi Python ?**

Python a été choisi pour ce projet grâce à sa simplicité et sa lisibilité, facilitant le développement de solutions complexes comme le traitement d'images. Sa compatibilité multiplateforme et son riche écosystème de bibliothèques spécialisées (comme OpenCV et NumPy) en font un outil performant pour manipuler des images et développer une interface graphique intuitive avec Tkinter et CustomTkinter. Enfin, sa grande communauté garantit un support fiable et de nombreuses ressources.

---

## Bibliothèques utilisées :

- **numpy**

Utilisé pour les calculs mathématiques et la manipulation des tableaux, essentiels pour les opérations de traitement d'images.

- **matplotlib**

Utilisé pour afficher et visualiser les images ou graphiques générés au cours du traitement.

- **PIL (Pillow)**

Permet de charger, manipuler et sauvegarder des images.

- **cv2 (OpenCV)**

Fournit des algorithmes avancés pour le traitement d'images et la détection des contours.

- **customtkinter**

Une bibliothèque pour créer des interfaces graphiques modernes et esthétiques avec Tkinter.

- **tkinter**

Utilisé pour l'interface utilisateur graphique, notamment pour les boîtes de dialogue et les messages (e.g., chargement de fichiers, messages d'alerte).

- **collections**

La classe **deque** est utilisée pour gérer des structures de données dynamiques (comme des files).

- **os**

Permet d'effectuer des opérations sur le système de fichiers, comme la gestion des chemins d'accès ou des fichiers.

---

## Installation des dépendances

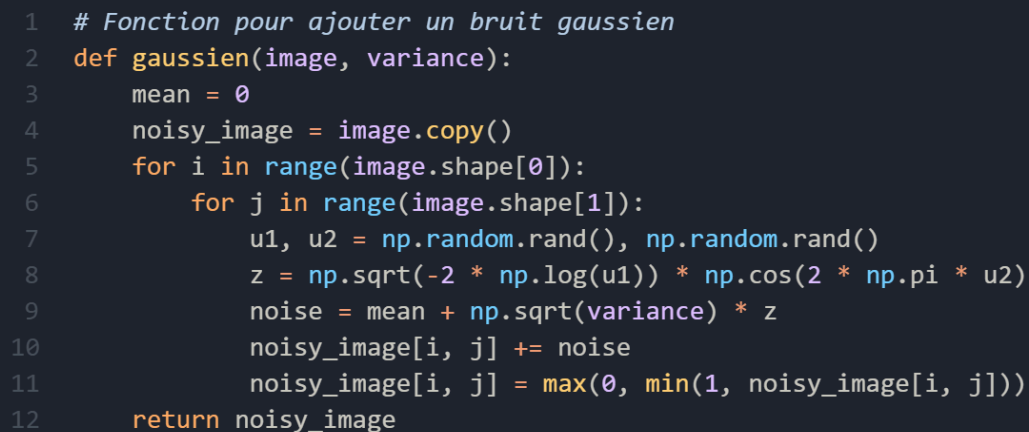
Pour simplifier le processus d'installation des dépendances, un fichier `requirements.txt` a été créé. Ce fichier liste toutes les bibliothèques nécessaires avec leurs versions.

Exécutez la commande suivante pour installer les dépendances :

```
pip install -r requirements.txt
```

## Explication des méthodes (fonction):

### 1-Gaussien



```
1  # Fonction pour ajouter un bruit gaussien
2  def gaussien(image, variance):
3      mean = 0
4      noisy_image = image.copy()
5      for i in range(image.shape[0]):
6          for j in range(image.shape[1]):
7              u1, u2 = np.random.rand(), np.random.rand()
8              z = np.sqrt(-2 * np.log(u1)) * np.cos(2 * np.pi * u2)
9              noise = mean + np.sqrt(variance) * z
10             noisy_image[i, j] += noise
11             noisy_image[i, j] = max(0, min(1, noisy_image[i, j]))
12     return noisy_image
```

Cette fonction, appelée **gaussien**, ajoute un bruit gaussien (normal) à une image. La méthode utilisée est la **méthode de Box-Muller**, qui transforme deux variables uniformes en une variable suivant une **loi normale**. La formule utilisée pour générer cette variable est :

$$Z_0 = R \cos(\Theta) = \sqrt{-2 \ln U_1} \cos(2\pi U_2)$$

---

## 2-poivre et sel

```
1 def poivre_sel(image, pourcentage):
2     noisy_image = image.copy()
3     num_pixels = int(pourcentage * image.size)
4     height, width = image.shape
5     for _ in range(num_pixels // 2):
6         i, j = np.random.randint(0, height), np.random.randint(0, width)
7         noisy_image[i, j] = 0
8     for _ in range(num_pixels // 2):
9         i, j = np.random.randint(0, height), np.random.randint(0, width)
10        noisy_image[i, j] = 1
11    return noisy_image
12
```

Cette fonction, appelée **poivre\_sel**, ajoute un bruit "poivre et sel" à une image. Elle remplace un certain pourcentage de pixels de l'image par des valeurs extrêmes (0 ou 1). Le bruit est ajouté de manière aléatoire : environ la moitié des pixels sont mis à 0 (noir) et l'autre moitié à 1 (blanc). Le pourcentage spécifié détermine la proportion totale de pixels affectés par ce bruit, calculée par rapport à la taille de l'image.

## 3-filtre de prewitt

```
1 # Filtre Prewitt
2 def prewitt(image):
3     kernel_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
4     kernel_y = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]])
5     grad_x = np.zeros_like(image)
6     grad_y = np.zeros_like(image)
7     for i in range(1, image.shape[0] - 1):
8         for j in range(1, image.shape[1] - 1):
9             grad_x[i, j] = np.sum(kernel_x * image[i - 1:i + 2, j - 1:j + 2])
10            grad_y[i, j] = np.sum(kernel_y * image[i - 1:i + 2, j - 1:j + 2])
11    grad_x = np.abs(grad_x) / np.max(np.abs(grad_x))
12    grad_y = np.abs(grad_y) / np.max(np.abs(grad_y))
13    return grad_x, grad_y
```

---

La fonction prewitt applique le filtre de Prewitt pour détecter les bords dans une image en calculant les gradients dans les directions horizontale et verticale.

1. Noyaux de Prewitt :

<b>-1</b>	<b>0</b>	<b>+1</b>
<b>-1</b>	<b>0</b>	<b>+1</b>
<b>-1</b>	<b>0</b>	<b>+1</b>

**G<sub>x</sub>**

<b>+1</b>	<b>+1</b>	<b>+1</b>
<b>0</b>	<b>0</b>	<b>0</b>
<b>-1</b>	<b>-1</b>	<b>-1</b>

**G<sub>y</sub>**

\*

On a 2 noyaux principales :

- Le noyau horizontal (kernel\_x) détecte les variations d'intensité horizontales
  - Le noyau vertical (kernel\_y) détecte les variations d'intensité verticales
2. Convolution : Pour chaque pixel de l'image (sauf les bords), la fonction applique les noyaux pour calculer les gradients horizontaux et verticaux en multipliant les pixels voisins par les éléments des noyaux et en les additionnant.
  3. Normalisation : Les gradients obtenus sont ensuite normalisés pour que leurs valeurs varient entre 0 et 1.
  4. Retour des résultats : La fonction retourne les deux gradients normalisés : un pour les bords horizontaux (grad\_x) et l'autre pour les bords verticaux (grad\_y).

---

## 4-filtre de sobel

```
1 # Filtre Sobel
2 def sobel(image):
3     kernel_x = np.array([[ -1,  0,  1], [-2,  0,  2], [-1,  0,  1]])
4     kernel_y = np.array([[ 1,  2,  1], [ 0,  0,  0], [-1, -2, -1]])
5     grad_x = np.zeros_like(image)
6     grad_y = np.zeros_like(image)
7     for i in range(1, image.shape[0] - 1):
8         for j in range(1, image.shape[1] - 1):
9             grad_x[i, j] = np.sum(kernel_x * image[i - 1:i + 2, j - 1:j + 2])
10            grad_y[i, j] = np.sum(kernel_y * image[i - 1:i + 2, j - 1:j + 2])
11     grad_x = np.abs(grad_x) / np.max(np.abs(grad_x))
12     grad_y = np.abs(grad_y) / np.max(np.abs(grad_y))
13     return grad_x, grad_y
```

La fonction sobel applique le filtre de Sobel, qui est similaire au filtre de Prewitt, pour détecter les bords dans une image en calculant les gradients dans les directions horizontale et verticale.

### 1. Utilisation de deux noyaux :

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

Gx

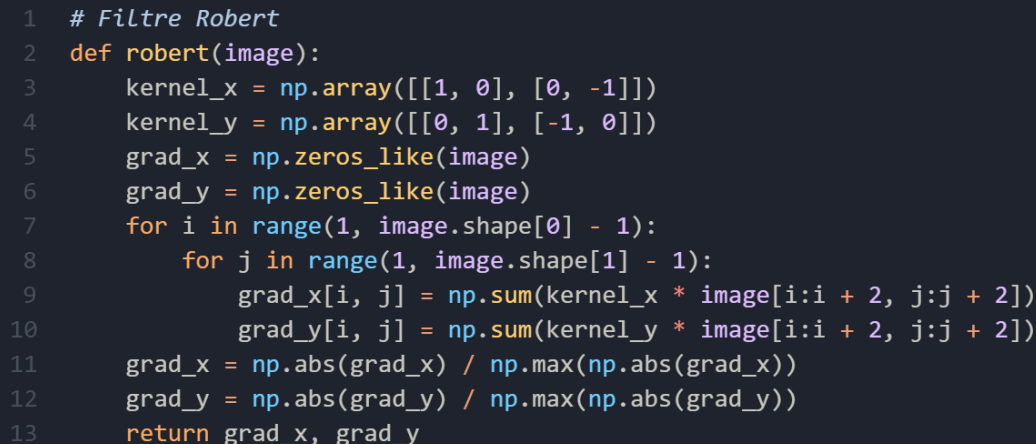
+1	+2	+1
0	0	0
-1	-2	-1

Gy

- Le premier noyau est utilisé pour détecter les variations d'intensité dans la direction horizontale
- Le deuxième noyau est utilisé pour détecter les variations d'intensité dans la direction verticale

- 
2. Convolution : La fonction applique ces noyaux à chaque pixel de l'image (en excluant les bords) pour calculer les gradients horizontaux et verticaux, en effectuant une multiplication et une somme des valeurs de l'image avec les noyaux correspondants.
  3. Normalisation : Les gradients calculés sont ensuite normalisés, ce qui signifie que leurs valeurs sont ajustées pour être comprises entre 0 et 1.
  4. Retour des résultats : La fonction retourne les deux gradients normalisés : un pour les bords horizontaux (grad\_x) et l'autre pour les bords verticaux (grad\_y)

## 5-Filtre de robert

A screenshot of a code editor with a dark background and light-colored text. The code defines a function named 'robert' that takes an 'image' as input. It initializes two 2x2 kernels: 'kernel\_x' for horizontal gradient detection and 'kernel\_y' for vertical gradient detection. It then iterates over the image pixels (excluding the borders) and calculates the gradients using the kernels. Finally, it normalizes the gradients by dividing by the maximum absolute value of the gradients and returns the two normalized gradient maps.

```
1  # Filtre Robert
2  def robert(image):
3      kernel_x = np.array([[1, 0], [0, -1]])
4      kernel_y = np.array([[0, 1], [-1, 0]])
5      grad_x = np.zeros_like(image)
6      grad_y = np.zeros_like(image)
7      for i in range(1, image.shape[0] - 1):
8          for j in range(1, image.shape[1] - 1):
9              grad_x[i, j] = np.sum(kernel_x * image[i:i + 2, j:j + 2])
10             grad_y[i, j] = np.sum(kernel_y * image[i:i + 2, j:j + 2])
11     grad_x = np.abs(grad_x) / np.max(np.abs(grad_x))
12     grad_y = np.abs(grad_y) / np.max(np.abs(grad_y))
13     return grad_x, grad_y
```

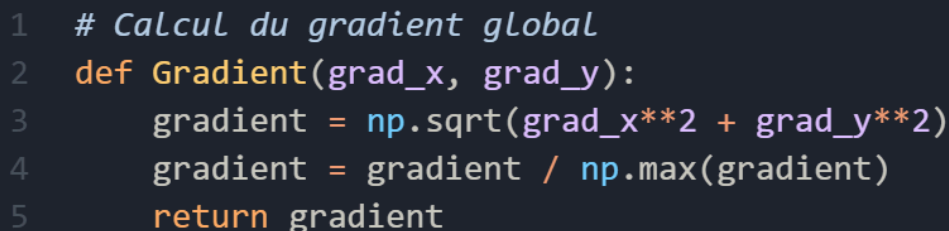
La fonction robert applique le filtre de Robert, qui est un opérateur de détection des bords très simple utilisant des noyaux de petite taille (2x2) pour détecter les gradients dans les directions horizontale et verticale.

1. Utilisation de deux noyaux :



- 
- Le premier noyau est utilisé pour détecter les variations d'intensité dans la direction horizontale
  - Le deuxième noyau est utilisé pour détecter les variations d'intensité dans la direction verticale
2. Convolution : La fonction applique ces noyaux à chaque pixel de l'image (en excluant les bords) pour calculer les gradients horizontaux et verticaux. Pour chaque pixel, une multiplication élément par élément est effectuée entre le noyau et une fenêtre 2x2 de l'image, et les résultats sont ensuite sommés pour obtenir les gradients.
  3. Normalisation : Les gradients calculés (grad\_x et grad\_y) sont normalisés pour que leurs valeurs soient comprises entre 0 et 1. Cela permet de standardiser les résultats indépendamment des variations d'intensité initiales de l'image.
  4. Retour des résultats : La fonction retourne les deux gradients normalisés : un pour les bords horizontaux (grad\_x) et l'autre pour les bords verticaux (grad\_y).

## 6-fonction du gradient



```
1  # Calcul du gradient global
2  def Gradient(grad_x, grad_y):
3      gradient = np.sqrt(grad_x**2 + grad_y**2)
4      gradient = gradient / np.max(gradient)
5      return gradient
```

La fonction Gradient combine les gradients calculés dans les directions horizontale (grad\_x) et verticale (grad\_y) pour obtenir l'amplitude du gradient global à chaque pixel, ce qui permet de détecter les bords plus efficacement.

- 
1. Calcul de l'amplitude du gradient : Le gradient global est obtenu en utilisant la formule de la norme du vecteur gradient :

$$\text{gradient} = \sqrt{\text{grad}_x^2 + \text{grad}_y^2}$$

Cette formule permet de combiner les informations des gradients horizontaux et verticaux. L'idée est que la somme quadratique des gradients horizontaux et verticaux donne l'intensité du changement total à un point donné.

2. Normalisation : Après avoir calculé l'amplitude du gradient pour chaque pixel, les valeurs du gradient sont normalisées pour que l'intensité du gradient soit comprise entre 0 et 1. Cela est effectué en divisant le gradient par sa valeur maximale :

$$\text{gradient} = \frac{\text{gradient}}{\max(\text{gradient})}$$

Cette étape permet de rendre les valeurs du gradient indépendantes de l'échelle de l'image, ce qui facilite leur comparaison.

3. Retour du résultat : La fonction retourne l'image du gradient normalisé, qui représente la force des bords à chaque pixel de l'image.

## 7-seuil-simple

```
1  # Fonction de seuillage simple
2  def SeuilSim(image, seuil):
3      if image.max() > 1:
4          image = image / 255.0
5      binary_image = np.zeros_like(image)
6      binary_image[image > seuil] = 1
7      return (binary_image * 255).astype(np.uint8)
```

---

La fonction `SeuilSim` effectue une binarisation de l'image en utilisant un seuil spécifique. Elle convertit l'image en une image binaire (noir et blanc), où les pixels dont l'intensité est supérieure à un seuil donné sont mis à 1 (blanc), et ceux inférieurs à ce seuil sont mis à 0 (noir).

1. Vérification de l'échelle de l'image :

- La fonction vérifie d'abord si les valeurs des pixels de l'image sont supérieures à 1. Si c'est le cas, cela signifie que l'image a des valeurs d'intensité comprises entre 0 et 255 (plage d'une image de 8 bits). Pour la binarisation, les valeurs doivent être normalisées entre 0 et 1, donc l'image est divisée par 255 pour ramener les intensités dans la plage `[0, 1]`.

2. Création de l'image binaire :

- Une image binaire vide (initialisée à 0) de la même taille que l'image d'origine est créée : `binary_image = np.zeros_like(image)`.
- Ensuite, la fonction applique un seuil. Tous les pixels de l'image dont l'intensité est supérieure au seuil spécifié sont définis à 1 (blanc), et les autres restent à 0 (noir) : `binary_image[image > seuil] = 1`.

3. Retour à l'échelle des 8 bits :

- Une fois la binarisation effectuée, l'image binaire est multipliée par 255 pour ramener les valeurs des pixels dans l'échelle des 8 bits (`[0, 255]`). Cela permet d'obtenir une image où les pixels sont soit 0 (noir) soit 255 (blanc).
- La fonction retourne l'image binaire sous forme de type entier non signé de 8 bits (`np.uint8`), ce qui est le format standard pour les images.

## 8-seuil-hystérisis

Une image binaire est initialisée à zéro (noir), et tous les pixels dont l'intensité est supérieure à un seuil élevé (`seuil_haut`) sont définis comme des arêtes fortes (1, blanc) dans l'image binaire :

```
binary_image[image > seuil_haut] = 1
```

```

1  def SeuilHys(image, seuil_bas, seuil_haut):
2      # Initialize binary image and visited matrix
3      binary_image = np.zeros_like(image, dtype=np.uint8)
4      binary_image[image > seuil_haut] = 1
5      visited = np.zeros_like(image, dtype=bool)
6      # Initialize queue
7      queue = deque()
8      # Enqueue all strong edges (above the high threshold)
9      for i in range(1, image.shape[0] - 1):
10         for j in range(1, image.shape[1] - 1):
11             if binary_image[i, j] == 1 and not visited[i, j]:
12                 queue.append((i, j))
13     # BFS for weak edge propagation
14     while queue:
15         x, y = queue.popleft()
16         if visited[x, y]:
17             continue
18         visited[x, y] = True
19         # Check if the current pixel qualifies as a weak edge
20         if seuil_bas < image[x, y] <= seuil_haut:
21             binary_image[x, y] = 1
22             # Propagate to neighbors
23             for dx, dy in [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)]:
24                 nx, ny = x + dx, y + dy
25                 if 0 <= nx < image.shape[0] and 0 <= ny < image.shape[1]:
26                     queue.append((nx, ny))
27     return binary_image

```

### 1. Propagation des arêtes faibles :

- Une matrice visitée est créée pour marquer les pixels qui ont déjà été analysés et ainsi éviter les répétitions.
- Un queue (file) est utilisée pour appliquer une recherche en largeur (BFS) sur les arêtes fortes détectées. Tous les pixels correspondant à des arêtes fortes sont ajoutés à la file pour commencer la propagation des arêtes faibles.
- Pour chaque pixel dans la file, la fonction vérifie s'il peut être classé comme arête faible (c'est-à-dire que son intensité est entre les seuils bas et haut). Si un pixel satisfait cette condition, il est marqué comme une arête forte et les voisins immédiats sont ajoutés à la file pour une propagation supplémentaire :

```
if seuil_bas < image[x, y] <= seuil_haut:
```

```
    binary_image[x, y] = 1
```

- 
- La propagation se fait pour les voisins immédiats dans les huit directions cardinales et diagonales.

## 2. Retour du résultat :

- Après avoir propagé les arêtes faibles et visité tous les pixels pertinents, l'image binaire est renvoyée avec les pixels fortement et faiblement détectés comme des bords.

## 9-log\_kernel(sigma) : Création du noyau Laplacien de Gaussien (LoG)

```
1 def log_kernel(sigma):
2     # Create the grid for x and y coordinates
3     x, y = np.meshgrid(np.arange(-1, 2), np.arange(-1, 2))
4     # Compute the Laplacian of Gaussian
5     norm = (x**2 + y**2) / (2 * sigma**2)
6     gaussian = np.exp(norm) * 4 / np.sqrt(2 * np.pi * sigma**2)
7     laplacian = (norm - 1) * gaussian
8     # Normalize the kernel to have a sum of zero
9     log_kernel = laplacian - laplacian.mean()
10    return log_kernel
```

Cette fonction crée un noyau LoG basé sur un paramètre de **sigma** (l'écart type de la distribution gaussienne).

- **Création d'une grille de coordonnées x et y** : La fonction commence par créer une grille de coordonnées x et y qui va de -1 à 1, formant une matrice 3x3.
- **Calcul de la norme** : La norme est calculée en fonction des coordonnées x et y, en utilisant la formule :

$$(x^2 + y^2)/(2\sigma^2)$$

---

qui est utilisée dans la fonction gaussienne.

- **Calcul de la fonction Gaussienne** : La fonction gaussienne est calculée en utilisant :

$$\exp(\text{norm}) \cdot \frac{4}{\sqrt{2\pi\sigma^2}}.$$

- **Calcul du Laplacien de Gaussien** : Le noyau de Laplacien est ensuite calculé avec la formule :  $(\text{norm}-1) \cdot \text{gaussian}$ , qui applique l'opérateur Laplacien à la gaussienne.
- **Normalisation** : Enfin, le noyau est normalisé pour que sa somme soit égale à zéro en soustrayant la moyenne de la matrice résultante.

Le noyau généré est retourné comme une matrice 3x3.

## 10-convolve(image, kernel) : Convolution d'une image avec un noyau

```
1  def convolve(image, kernel):
2      rows = image.shape[0]
3      cols = image.shape[1]
4      pad = kernel.shape[0] // 2
5      output = np.zeros_like(image)
6      for i in range(pad, rows-pad):
7          for j in range(pad, cols-pad):
8              if kernel.shape[0] % 2 == 0:
9                  area = image[i:i+pad+1, j:j+pad+1]
10             else:
11                 area = image[i-1:i+2, j-1:j+2]
12                 output[i, j] = np.sum(kernel * area)
13     return output
```

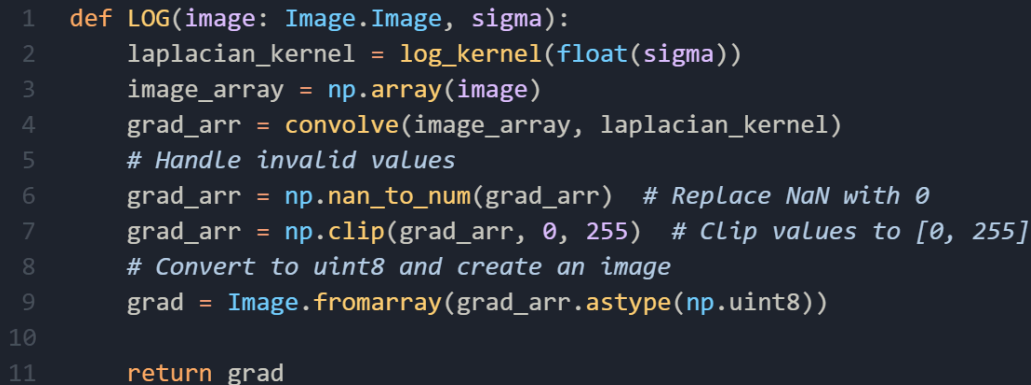
Cette fonction applique une convolution entre une image et un noyau (ici, le noyau LoG).

- 
- **Initialisation** : La fonction commence par initialiser les dimensions de l'image et du noyau. Elle crée aussi une matrice de sortie vide (**output**) de la même taille que l'image.
  - **Application de la convolution** : Ensuite, pour chaque pixel de l'image, un sous-ensemble de la matrice de l'image (en fonction de la taille du noyau) est pris autour du pixel courant. Ce sous-ensemble est multiplié par le noyau et la somme des produits est stockée dans l'image de sortie.
  - **Gestion des bordures** : La convolution est réalisée avec des bordures supplémentaires autour de l'image pour éviter des erreurs lors du calcul des pixels proches des bords.

Cette fonction retourne l'image après application de la convolution avec le noyau.

---

## 11- LOG(image, sigma) : Application du filtre Laplacien de Gaussien

A screenshot of a code editor with a dark background and light-colored text. The code defines a function LOG that takes an image and a sigma value. It calculates a Laplacian kernel, converts the image to a numpy array, performs a convolution, handles NaN values, clips the results to the range [0, 255], and finally converts the result back to an image format.

```
1 def LOG(image: Image.Image, sigma):
2     laplacian_kernel = log_kernel(float(sigma))
3     image_array = np.array(image)
4     grad_arr = convolve(image_array, laplacian_kernel)
5     # Handle invalid values
6     grad_arr = np.nan_to_num(grad_arr) # Replace NaN with 0
7     grad_arr = np.clip(grad_arr, 0, 255) # Clip values to [0, 255]
8     # Convert to uint8 and create an image
9     grad = Image.fromarray(grad_arr.astype(np.uint8))
10
11     return grad
```

La fonction **LOG** applique le filtre Laplacien de Gaussien à une image.

- **Création du noyau LoG** : Le noyau LoG est créé en appelant la fonction **log\_kernel(sigma)** avec un paramètre **sigma** donné.
- **Convolution de l'image** : L'image est convertie en un tableau numpy, puis convoluée avec le noyau LoG en appelant la fonction **convolve(image\_array, laplacian\_kernel)**.

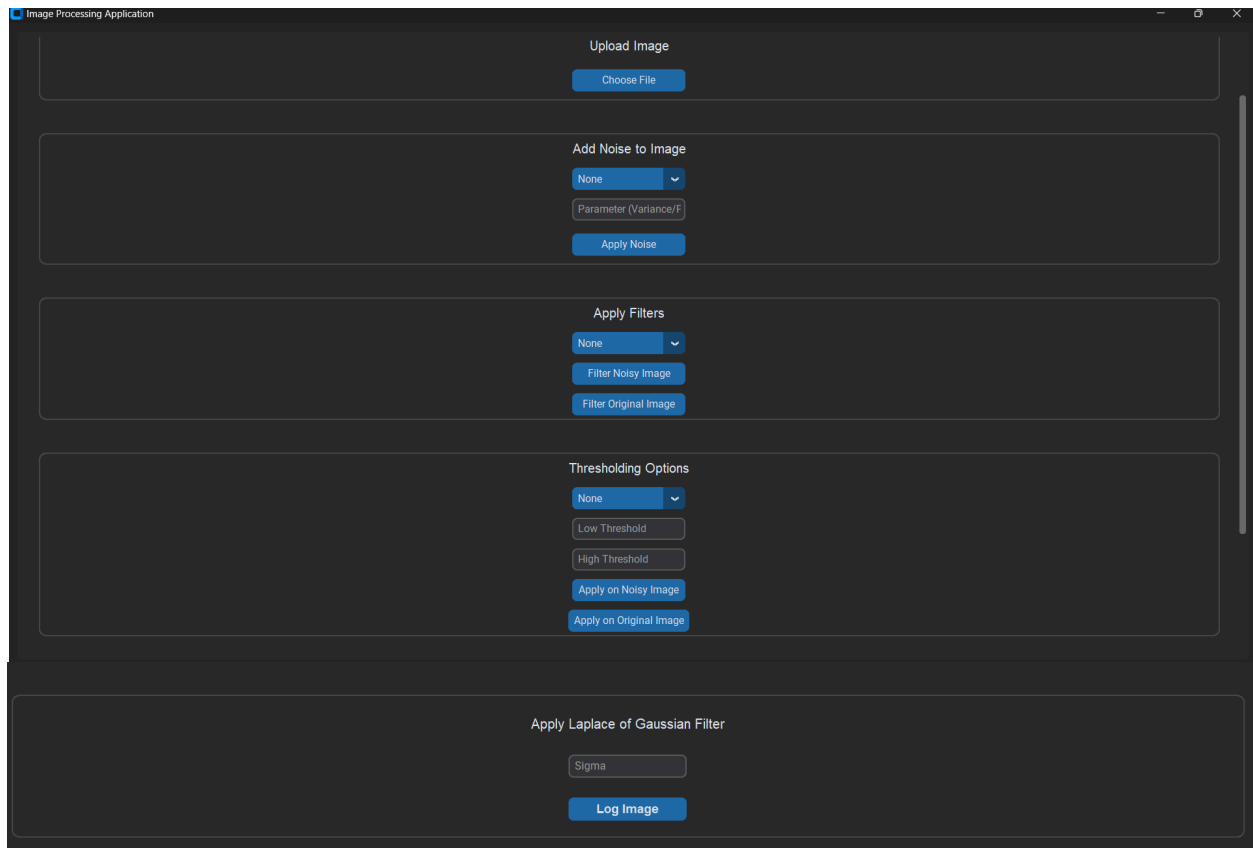
- **Gestion des valeurs invalides** : Après la convolution, les valeurs NaN sont remplacées par 0 et les valeurs en dehors de l'intervalle  $[0, 255]$  sont recadrées.
- **Conversion en image** : Enfin, les résultats de la convolution sont convertis en une image de type **uint8** et retournés sous forme d'image.

Le résultat final est une image où les bords sont accentués en utilisant un filtre **Laplacian of Gaussian**, qui détecte les zones de forte variation d'intensité dans l'image.

## Interface et execution :

### Interface :

Cette interface graphique (GUI) est conçue pour une application de **traitement d'images**, créée avec CustomTkinter. Elle permet aux utilisateurs de télécharger une image, d'ajouter du bruit, d'appliquer des filtres et des techniques de seuillage de manière structurée et intuitive.





---

Voici une description détaillée des différentes sections et fonctionnalités :

### **1. Section "Télécharger une image"**

- Un bouton intitulé "**Choose File**" permet à l'utilisateur de télécharger une image depuis son système local.
- Cette section est le point de départ pour effectuer le traitement de l'image.

### **2. Section "Ajouter du bruit à l'image"**

- Un menu déroulant intitulé "**None**" permet de sélectionner le type de bruit à ajouter (par exemple : bruit Gaussien, bruit de sel et poivre).
- Un champ de saisie intitulé "**Parameter (Variance/F)**" permet de configurer les paramètres du bruit (par exemple : la variance pour le bruit Gaussien).
- Un bouton intitulé "**Apply Noise**" applique le bruit sélectionné à l'image téléchargée.

### **3. Section "Appliquer des filtres"**

- Un menu déroulant intitulé "**None**" permet de choisir un filtre (par exemple : Sobel, Prewitt, etc.).
- Deux boutons sont disponibles :
  - "**Filter Noisy Image**" pour appliquer le filtre sur l'image bruitée.
  - "**Filter Original Image**" pour appliquer le filtre sur l'image d'origine.

### **4. Section "Options de seuillage"**

- Un menu déroulant intitulé "**None**" permet de choisir une méthode de seuillage (par exemple : seuillage par intervalles).
- Deux champs de saisie sont disponibles :
  - "**Low Threshold**" pour entrer la valeur minimale du seuil.
  - "**High Threshold**" pour entrer la valeur maximale du seuil.
- Deux boutons permettent d'appliquer le seuillage :
  - "**Apply on Noisy Image**" pour appliquer la méthode de seuillage sur l'image bruitée.

- **"Apply on Original Image"** pour appliquer la méthode sur l'image d'origine.

## 5. Section **"Applique le log (laplacien)":**

- **Bouton "Log"**

Applique une transformation logarithmique à l'image pour améliorer la visibilité des détails dans les zones sombres.

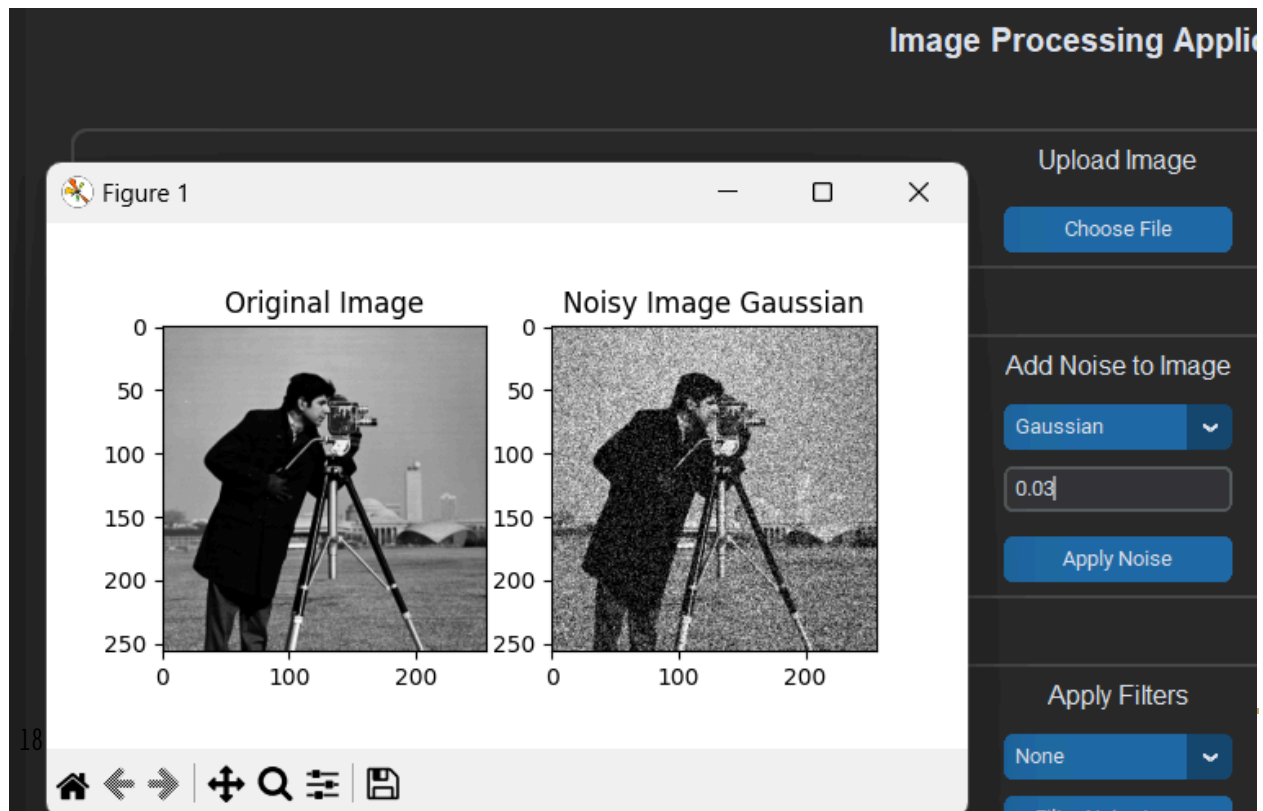
- **Entrée "Sigma"**

- Permet de définir l'écart-type (sigma) pour appliquer un filtrage, comme le flou gaussien, ou ajuster l'intensité du bruit dans l'image.

## Exécution et affichage des résultats :

### Chargement de l'image et ajout de bruit

Après avoir téléchargé l'image, sélectionnez le type de bruit que vous souhaitez ajouter (par exemple, bruit gaussien ou poivre sel ). Ensuite, entrez le paramètre approprié pour ce type de bruit (comme la variance pour le bruit gaussien ou la probabilité pour le bruit poivre\_sel). Une fois ces étapes terminées, l'image sera sauvegarder et affichée avec le bruit appliqué, permettant ainsi d'observer l'impact du bruit sur les détails de l'image.

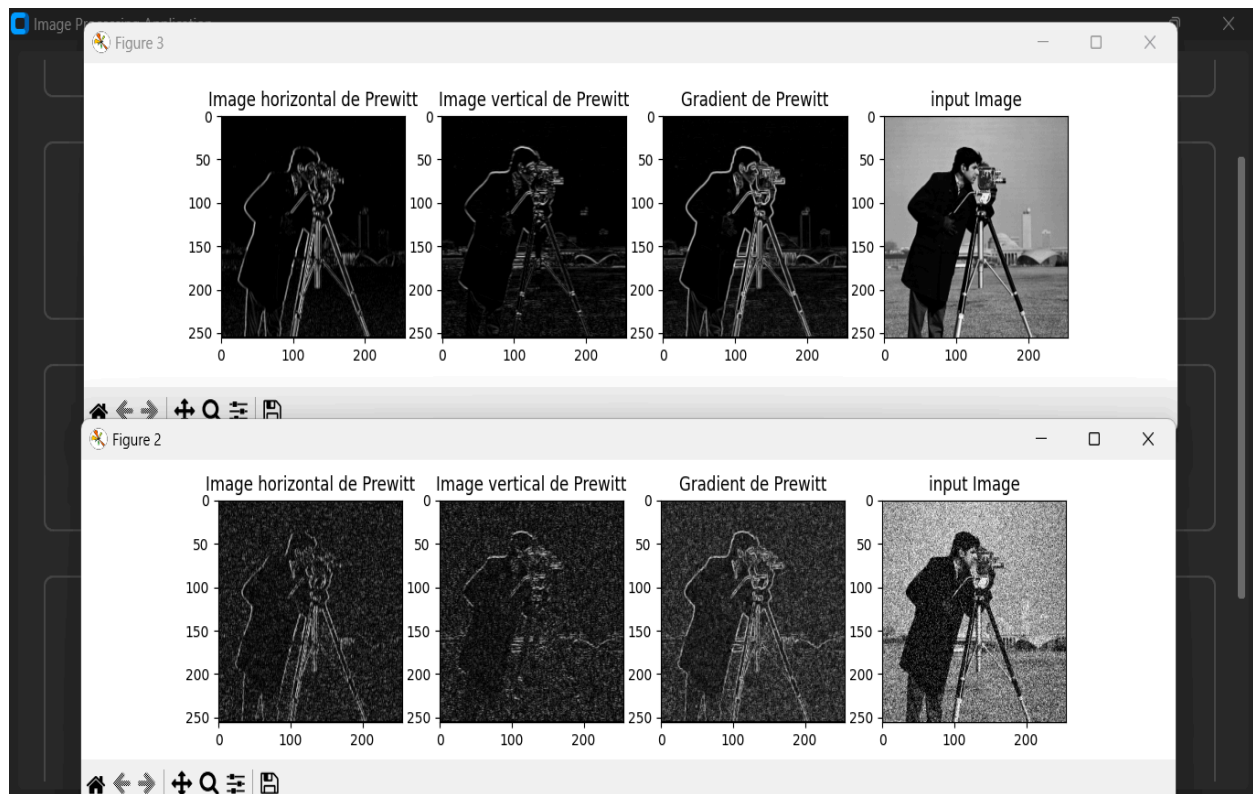


## Application du filtre Prewitt

En choisissant le filtre de Prewitt, l'image sera traitée pour détecter les contours. Cette opération peut être effectuée sur l'image originale ou sur l'image bruitée. (clique au button a choice) nous cliquant les deux button (bruit et original)

- **Figure 2** : Image avec bruit
- **Figure 3** : Image originale après application du filtre Prewitt.

Ces images permettent de comparer l'effet du filtrage sur l'image originale et bruitée.



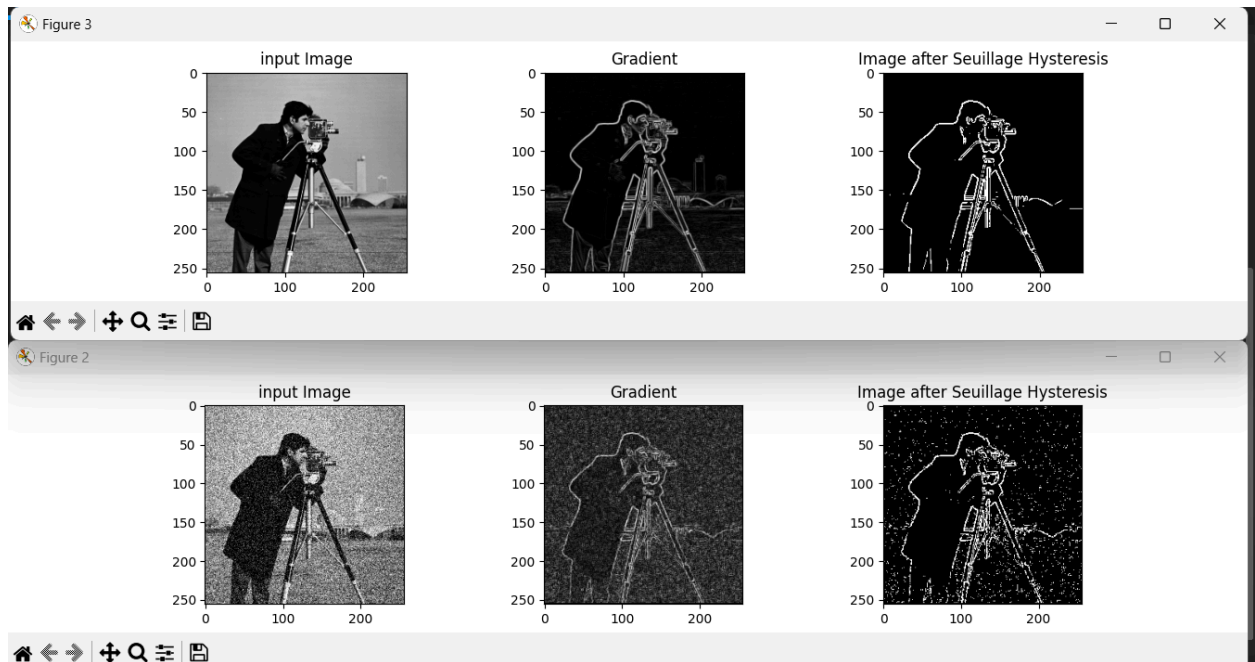
## Seuillage Hystérétique

L'application du seuillage hystérétique permet de définir des seuils pour distinguer les contours dans l'image. Par exemple, avec les paramètres suivants :

- **Seuil bas** : 0.6
- **Seuil haut** : 0.4

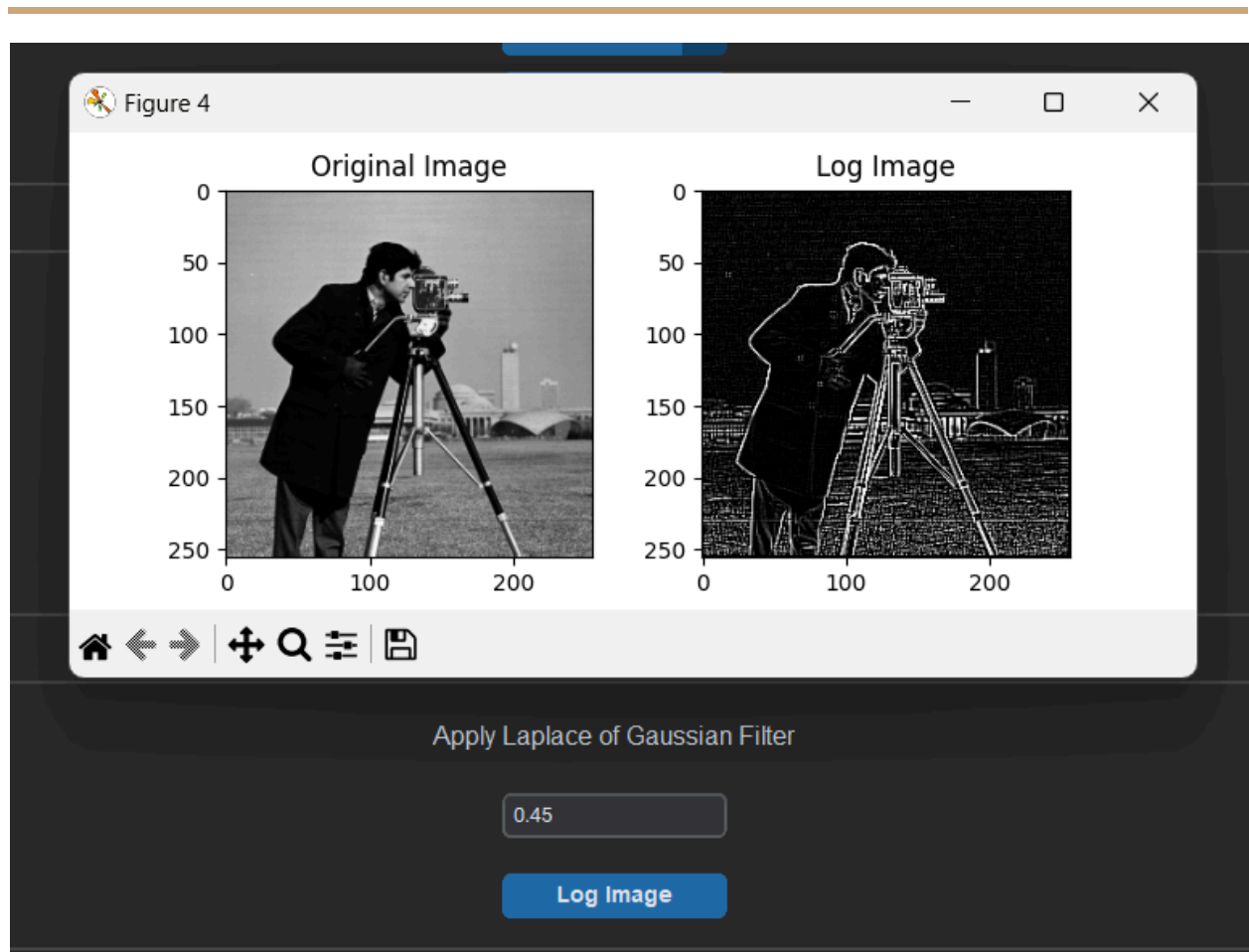
Vous pourrez observer les résultats sur les deux images suivantes :

- **Figure 2** : Image bruitée après application du seuillage hystérétique
- **Figure 3** : Image originale après seuillage hystérétique.



## LOG

Et maintenant détecter les contours en utilisant la méthode de laplacien of gaussien. avec sigma 0.45 (45)



## Conclusion :

Le choix du filtre le plus adapté dépend souvent du contexte et des exigences spécifiques de l'application. Toutefois, parmi les nombreux filtres disponibles, le filtre Sobel est souvent privilégié en raison de son équilibre entre sensibilité et robustesse. Il est particulièrement efficace pour détecter les contours dans des images où les détails sont essentiels, tout en restant résistant au bruit. De plus, sa simplicité et sa faible complexité le rendent facile à implémenter et à intégrer dans différents systèmes de traitement d'images. Malgré cela, pour certains cas d'utilisation, des filtres plus complexes peuvent être nécessaires pour répondre à des besoins spécifiques, mais le filtre Sobel reste une solution très performante dans la plupart des situations courantes.