BORDEAUX INP Enseirb-Matmeca

Rapport

TS225 – Projet Image
Lecture de code-barres par lancers aléatoires de rayons

Rédigé par :
Mehdi Khabouze
Mohammed Arif
Salma Alouah
Mohamed Louriz

Encadrant :
M. Donias

Décembre 2024

Table des matières

1 Introduction

Dans un contexte où les codes-barres sont omniprésents sur les produits de consommation et les documents administratifs, leur lecture et décodage automatique représentent un enjeu majeur. Ce projet porte sur le développement d'une solution de lecture de codes-barres EAN-13 à partir d'images numériques, utilisant une approche originale par lancers aléatoires de rayons. Les objectifs principaux sont d'implémenter une méthode de lecture robuste capable de décoder les codes-barres à partir d'images de qualité variable, de développer une approche automatique basée sur des lancers aléatoires de rayons simulant le comportement d'un scanner laser, d'optimiser le processus en identifiant les régions d'intérêt pour réduire le nombre de tirages nécessaires, et de valider la méthode sur une base de données diversifiée de codes-barres. Le projet s'articule autour d'une démarche progressive, partant d'une initialisation manuelle pour aboutir à une solution entièrement automatisée. L'implémentation sera réalisée en Python, avec une interface graphique pour faciliter l'utilisation du système.

2 Algorithme

2.1 Programme principal

Cet algorithme décrit le fonctionnement général du programme principal permettant la détection et le décodage d'un code-barres EAN-13 en utilisant des lancers aléatoires de rayons.

Algorithme 1 : Algorithme du programme principal

Input : Image d'entrée I (chemin du fichier), Paramètres : nombre maximal d'essais N_{max}

Output : Code-barres décodé C ou message d'échec

1. Chargement et validation de l'image :
- Demander à l'utilisateur le chemin vers l'image.
- Vérifier l'existence et la validité du fichier.

2. Détection de la région d'intérêt :
- Appliquer la fonction `segmentation` pour localiser la zone potentielle contenant le code-barres.
- Récupérer les bornes de la région (p_1, p_2, p_3, p_4).

3. Lancers de rayons aléatoires et extraction :
- Initialiser un compteur $k = 0$.
- Tant que $k < N_{max}$ et qu'aucun code-barres valide n'est trouvé :
  - Générer un rayon aléatoire avec `lancer_aleatoire`.
  - Extraire la signature binaire avec `extract_signature`.
  - Tenter de décoder la signature avec `decode_ean13_signature`.
  - Si le décodage réussit, retourner le code-barres détecté.
  - Sinon, incrémenter k.

4. Résultats :
- Si un code-barres valide est détecté, l'afficher.
- Sinon, afficher un message indiquant l'échec après N_{max} tentatives.

2.2 Détection des régions d'intérêt

2.2.1 Segmentation

L'objectif de cette étape est d'identifier la région d'intérêt contenant potentiellement un code-barres. L'approche repose sur des techniques de traitement d'image, combinant des filtres gaussiens, des calculs de gradients et des mesures de cohérence structurale. Ces étapes permettent de repérer des motifs parallèles caractéristiques des codes-barres et d'en isoler la zone.

Algorithme 2 : Algorithme de segmentation

Input : Image d'entrée I, paramètres : σ_G, σ_T, seuil τ
Output : Masque binaire M_{final}

1. Chargement et prétraitement :
Charger l'image I et la convertir en niveaux de gris.
Ajouter un bruit gaussien pour simuler des perturbations.
2. Calcul des gradients :
Générer des filtres dérivés gaussiens G_x et G_y à l'aide du paramètre σ_G.
Appliquer ces filtres pour calculer les dérivées partielles I_x et I_y.
Normaliser les gradients afin de limiter les variations extrêmes.
3. Tenseur de structure :
Lisser les gradients avec un filtre gaussien supplémentaire basé sur σ_T.
Calculer les composantes du tenseur de structure :

$$T_{xx}, T_{xy}, T_{yy}$$

Ces valeurs permettent de modéliser la distribution locale des directions dans l'image.
4. Analyse de cohérence :
Estimer la cohérence directionnelle $D1$ pour détecter les zones ayant une orientation marquée.
Seuiler cette mesure avec τ afin d'obtenir un masque binaire M :

$$M(i,j) = \begin{cases} 1 & \text{si } D1(i,j) > \tau \\ 0 & \text{sinon.} \end{cases}$$

5. Nettoyage morphologique :
Appliquer des opérations d'ouverture et de fermeture pour :
- Connecter les barres fragmentées.
- Éliminer le bruit résiduel.

6. Sélection de la région la plus pertinente :
Détecter toutes les composantes connexes dans le masque binaire.
Identifier la plus grande région et en extraire les contours.
Générer un masque final M_{final} basé sur cette région dominante.

2.2.2 Détermination des Bornes du code-barre

L'objectif de cette étape est d'encadrer précisément la région détectée par un rectangle orienté. Cette procédure repose sur une analyse géométrique utilisant l'Analyse en Composantes Principales (PCA) pour identifier l'orientation et les dimensions principales de la région d'intérêt.

Algorithme 3 : Algorithme de détermination des bornes du code-barre

Input : Masque binaire M_{final}
Output : Quatre coins $(C1, C2, C3, C4)$ définissant un rectangle orienté

1. Calcul du barycentre :
    Estimer le barycentre (O_x, O_y) en utilisant la moyenne pondérée des positions des pixels actifs dans M_{final}.
2. Analyse des directions principales (PCA) :
    Centrer les coordonnées autour du barycentre.
    Calculer la matrice de covariance des positions des pixels.
    Extraire les vecteurs propres pour déterminer les directions principales de la région.
3. Projection et délimitation :
    Projeter les coordonnées sur les axes définis par les vecteurs propres.
    Déterminer les valeurs minimales et maximales des projections pour encadrer la région.
4. Construction du rectangle englobant :
    Calculer les coordonnées des quatre coins $(C1, C2, C3, C4)$ en utilisant les vecteurs propres et les extrêmes des projections.

2.3 Lancers des rayons

2.3.1 Lancer manuel du rayon

Cet algorithme décrit la méthode permettant à l'utilisateur de définir manuellement un rayon sur une image pour extraire des informations à partir d'un code-barres potentiel.

Algorithme 4 : Algorithme du lancer manuel du rayon

Input : Image d'entrée I (chemin du fichier)
Output : Coordonnées des points p_1 et p_2 définissant le rayon

1. Chargement de l'image :
Demander à l'utilisateur de fournir un chemin d'accès valide pour l'image.
Charger l'image et vérifier son intégrité.
Afficher l'image dans une interface graphique.
2. Sélection des points :
Afficher un message guidant l'utilisateur à sélectionner deux points.
Premier clic : Enregistrer les coordonnées du point de départ p_1.
Deuxième clic : Enregistrer les coordonnées du point d'arrivée p_2.
Tracer une ligne reliant les deux points sur l'image affichée.
3. Calcul de la longueur du rayon :
Calculer la distance euclidienne entre p_1 et p_2 :

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Afficher la longueur du rayon calculé.
4. Échantillonnage des points sur le rayon :
Interpoler N points équidistants entre p_1 et p_2.
Calculer les coordonnées (x_i, y_i) de chaque point :

$$x_i = x_1 + t \cdot (x_2 - x_1), \quad y_i = y_1 + t \cdot (y_2 - y_1)$$

où t varie de 0 à 1.

2.3.2 Lancers aléatoires des rayons

Approche 1 : Horizontalement et selon les deux diagonales

Cet algorithme génère des rayons orientés horizontalement et suivant les deux diagonales dans une région définie par quatre coins. L'objectif est de balayer différentes directions dans la zone d'intérêt afin d'augmenter les chances d'intercepter un code-barres.

Algorithme 5 : Algorithme des lancers aléatoires des rayons - Approche 1

Input : Coordonnées des coins C_1, C_2, C_3, C_4

Output : Coordonnées des points (p_1, p_2) définissant un rayon

1 **1. Vérifier les entrées :**

2 S'assurer que chaque coin C_i est défini par deux coordonnées (x, y).

3 **2. Calculer le centre de la région :**

4 Déterminer les coordonnées du centre en faisant la moyenne des positions des coins :

$$center_x = \frac{x_1 + x_2 + x_3 + x_4}{4}, \quad center_y = \frac{y_1 + y_2 + y_3 + y_4}{4}$$

5 **3. Calculer le rayon maximal :**

6 Mesurer la distance maximale entre le centre et les coins opposés :

$$radius = \min(\|C_1 - center\|, \|C_3 - center\|)$$

7 **4. Choisir une direction aléatoire :**

8 Générer un angle θ compris entre 0° et 360°.

9 Convertir cet angle en radians :

$$\theta_{rad} = \frac{\pi \cdot \theta}{180}$$

10 **5. Calculer les points de départ et d'arrivée :**

11 Définir les extrémités du rayon en utilisant l'angle et la longueur calculée :

$$p_1 = (center_x + radius \cdot \cos(\theta_{rad}), center_y + radius \cdot \sin(\theta_{rad}))$$

$$p_2 = (center_x - radius \cdot \cos(\theta_{rad}), center_y - radius \cdot \sin(\theta_{rad}))$$

Approche 2 : Lancers avec angles aléatoires contraints

Cet algorithme génère des rayons dont les angles sont créés de manière aléatoire via la sélection aléatoire de points sur des segments opposés du rectangle. Les angles formés par ces rayons avec la direction principale sont naturellement aléatoires dû à la position aléatoire des points. Une contrainte de ±30 degrés est appliquée pour sélectionner uniquement les rayons dont l'angle respecte cette limite, augmentant ainsi la probabilité d'une lecture valide du code-barres. Cette approche combine l'aléatoire dans la génération des angles avec un contrôle sur leur plage accep-

Algorithme 6 : Algorithme de génération d'angles aléatoires contraints pour les rayons

Input : Coins du rectangle orienté C_1, C_2, C_3, C_4
Output : Points définissant le rayon (p_1, p_2)

1. Initialisation :
  Calculer la direction principale :

$$\text{direction} = \frac{C_2 - C_1}{\|C_2 - C_1\|}$$

  Définir l'angle maximal autorisé : $\theta_{\max} = \pi/6$ (30 degrés)
  Initialiser le compteur d'essais : essais $= 0$
2. Boucle de génération :
  while essais ≤ 100 **do**
    Incrémenter essais
    if random() < 0.5 **then**
      Générer p_1 aléatoirement sur $[C_1, C_4]$
      Générer p_2 aléatoirement sur $[C_2, C_3]$
    else
      Générer p_1 aléatoirement sur $[C_2, C_3]$
      Générer p_2 aléatoirement sur $[C_1, C_4]$
    end if
    Calculer le vecteur rayon :

$$\text{rayon} = \frac{p_2 - p_1}{\|p_2 - p_1\|}$$

    Calculer l'angle avec la direction principale :

$$\text{angle} = \arccos(|\langle \text{rayon}, \text{direction} \rangle|)$$

    if angle $< \theta_{\max}$ **then**
      return (p_1, p_2)
  end while
  return Dernière paire (p_1, p_2) générée
Où génération d'un point aléatoire sur un segment $[P_1, P_2]$ **:**
  1. Générer $t \in [0, 1]$ aléatoirement
  2. Calculer le point : $P = P_1 + t(P_2 - P_1)$

L'algorithme produit des angles aléatoires de manière naturelle grâce à la sélection de points au hasard sur des segments opposés du rectangle. Cette sélection aléatoire des points définit un vecteur dont la direction, et donc l'angle formé avec la direction principale, est par construction aléatoire. Le filtrage appliqué par la contrainte de ± 30 degrés préserve cet aspect aléatoire tout en ne retenant que les rayons susceptibles de permettre une lecture fiable du code-barres. En effet, les rayons trop déviés de la direction principale sont écartés, tandis que ceux retenus, bien qu'aléatoires, restent suffisamment alignés pour traverser efficacement les barres du code. Cette méthode assure ainsi un équilibre optimal entre l'exploration aléatoire de la zone d'intérêt et la fiabilité de la lecture.

2.4 Extraction

Objectif : L'extraction vise à récupérer une signature numérique le long d'un rayon défini dans la région détectée. Cette signature est ensuite binarisée à l'aide du critère d'Otsu pour

générer une séquence binaire de 95 bits, qui sera utilisée lors de l'étape de décodage.

Algorithme 7 : Algorithme d'extraction avec binarisation

Input : Image I, Points de départ et d'arrivée du rayon (p_1, p_2)
Output : Signature binaire S_{bin} (95 bits)

1. Calcul de la longueur du rayon :

Mesurer la distance entre les deux points p_1 et p_2 :

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

2. Générer les points échantillonnés le long du rayon :

Diviser le rayon en $N = \max(L, 95)$ points pour garantir une résolution suffisante.

Interpoler les coordonnées (x_i, y_i) entre p_1 et p_2 :

$$x_i = x_1 + t_i(x_2 - x_1), \quad y_i = y_1 + t_i(y_2 - y_1)$$

avec :

$$t_i = \frac{i}{N-1}, \quad i \in [0, N-1]$$

3. Extraire l'intensité des pixels :

Échantillonner les valeurs d'intensité dans l'image I aux coordonnées interpolées (x_i, y_i).

Former la signature S :

$$S = [I(x_0, y_0), I(x_1, y_1), \ldots, I(x_{N-1}, y_{N-1})]$$

4. Calcul du seuil optimal par le critère d'Otsu :

Construire l'histogramme des intensités H pour S.

Pour chaque seuil k, calculer :

— Probabilités cumulées des classes :

$$w_b = \sum_{i=0}^{k-1} H[i], \quad w_f = \sum_{i=k}^{255} H[i]$$

— Moyennes des intensités :

$$\mu_b = \frac{\sum_{i=0}^{k-1} i \cdot H[i]}{w_b}, \quad \mu_f = \frac{\sum_{i=k}^{255} i \cdot H[i]}{w_f}$$

— Critère d'Otsu :

$$\sigma^2(k) = w_b \cdot w_f \cdot (\mu_b - \mu_f)^2$$

Choisir k^* maximisant $\sigma^2(k)$.

5. Binarisation de la signature :

Transformer la signature S en une signature binaire :

$$S_{bin}(i) = \begin{cases} 1 & \text{si } S[i] > k^* \\ 0 & \text{sinon} \end{cases}$$

6. Ajustement aux 95 bits :

Rééchantillonner ou tronquer S_{bin} pour obtenir exactement 95 bits.
Conserver uniquement les valeurs pertinentes après binarisation.

2.5 Décodage

Objectif : L'objectif de cette étape est de décoder une signature binaire extraite en un code-barres **EAN-13** valide. Le processus utilise des motifs de garde, des tables de codage et des vérifications de parité pour interpréter correctement la signature.

Input : Signature binaire extraite S (longueur 95 bits)
Output : Code-barres EAN-13 décodé ou erreur

1. Vérification de la longueur :
S'assurer que la signature contient exactement **95 bits**.
if $|S| \neq 95$ **then**
  Retourner une erreur : "Longueur incorrecte".
end if
2. Vérification des motifs de garde :
Vérifier les motifs :
— Début ($S[0:3] = 101$)
— Centre ($S[45:50] = 01010$)
— Fin ($S[92:95] = 101$)
if Motif incorrect **then**
  Retourner une erreur : "Motif de garde invalide".
end if
3. Décodage des 6 chiffres de gauche :
Initialiser $L = []$ et motif de parité $P = ''$.
for $i = 0$ **to** 5 **do**
  Extraire les 7 bits suivants $S[3 + 7i : 3 + 7(i + 1)]$.
  if Motif dans table L **then**
    Ajouter chiffre à L et ajouter 'L' à P.
  else if Motif dans table G **then**
    Ajouter chiffre à L et ajouter 'G' à P.
  else
    Retourner une erreur : "Motif inconnu".
  end if
end for

Suite sur la page suivante...

Algorithme 8 : Algorithme de décodage EAN-13 (suite)

4. Déterminer le premier chiffre :
Utiliser la table de parité pour traduire P en premier chiffre.
if P invalide **then**
  Retourner une erreur : "Parité incorrecte".
end if
5. Décodage des 6 chiffres de droite :
Initialiser $R = []$.
for $i = 0$ **to** 5 **do**
  Extraire les 7 bits suivants $S[50 + 7i : 50 + 7(i + 1)]$.
  if Motif dans table R **then**
    Ajouter chiffre à R.
  end if
  else
    Retourner une erreur : "Motif inconnu".
  end if
end for
6. Assemblage du code-barres complet :
Concaténer le premier chiffre, les chiffres de gauche (L) et les chiffres de droite (R) :

$$EAN\_13 = C_0 + L_1...L_6 + R_7...R_{12}$$

7. Vérification de la clé de contrôle :
Calculer la clé de contrôle :
1. Somme des chiffres en positions paires (2, 4, ..., 12) multipliée par 3 :

$$S_{pair} = 3 \sum_{i \in \text{pairs}} EAN[i]$$

2. Somme des chiffres en positions impaires (1, 3, ..., 11) :

$$S_{impair} = \sum_{i \in \text{impairs}} EAN[i]$$

3. Calcul du total :

$$Total = S_{pair} + S_{impair}$$

4. Calculer la clé :

$$C = (10 - (Total \mod 10)) \mod 10$$

Comparer avec le dernier chiffre du code-barres.
if $C \neq EAN[12]$ **then**
  Retourner une erreur : "Clé de contrôle invalide".
end if
8. Retourner le code-barres décodé :
Si toutes les vérifications sont réussies, retourner EAN_13.
Sinon, signaler une erreur.

Explications :

— **Motifs de garde :** Délimitent le début, le centre et la fin du code pour structurer la lecture.

— **Encodages L, G et R :** Différents types de codage utilisés pour garantir l'intégrité des données.
— **Clé de contrôle :** Assure la validité des 12 premiers chiffres en détectant les erreurs potentielles.

2.6 Base de données de codes-barres

Pour évaluer et valider nos algorithmes, nous avons constitué une base de données variée de codes-barres. Cette base joue un rôle essentiel dans l'analyse des performances et de la robustesse des méthodes que nous avons développées.

2.6.1 Objectifs et contenu

L'objectif de cette base de données est d'offrir un ensemble d'images suffisamment diversifié pour :
— Tester les algorithmes dans des conditions réalistes et variées.
— Reproduire des scénarios complexes, comme des orientations aléatoires ou des variations d'éclairage.
— Vérifier que les résultats respectent la norme `EAN-13`.
Les images utilisées présentent plusieurs caractéristiques intéressantes :
— **Sources :** Elles proviennent de smartphones, de scanners ou encore de bases de données en ligne.
— **Normes :** La majorité des codes-barres sont au format `EAN-13`.
— **Diversité :** Les images incluent des variations en termes de qualité, d'orientation, de luminosité et de bruit.

2.6.2 Collecte des données

Pour constituer cette base, nous avons adopté plusieurs approches :

1. Téléchargement d'images depuis des bases de données en ligne existantes.
2. Captures réalisées à l'aide de smartphones et de scanners dans différents environnements.
3. Création de situations simulées, comme des images prises sous différents angles et avec des conditions d'éclairage variées.

2.6.3 Utilisation et perspectives

Cette base de données nous a permis de :
— Tester chaque étape de l'algorithme, de la segmentation à la lecture du code-barres.
— Ajuster des paramètres critiques comme les seuils de binarisation et l'échantillonnage.
— Identifier les faiblesses des méthodes implémentées et les améliorer.
Même si la base actuelle couvre une large variété de cas, des améliorations sont envisageables. Par exemple, on pourrait l'enrichir avec des situations encore plus complexes, comme des codes-barres flous, partiellement masqués ou en mouvement. Ces ajouts permettraient d'évaluer davantage la robustesse et l'efficacité des algorithmes face à des défis plus poussés.

La figure 1 montre un aperçu des images utilisées dans notre base de données. Ces exemples incluent différentes orientations, résolutions et conditions d'éclairage pour évaluer les performances des algorithmes sur des cas variés.

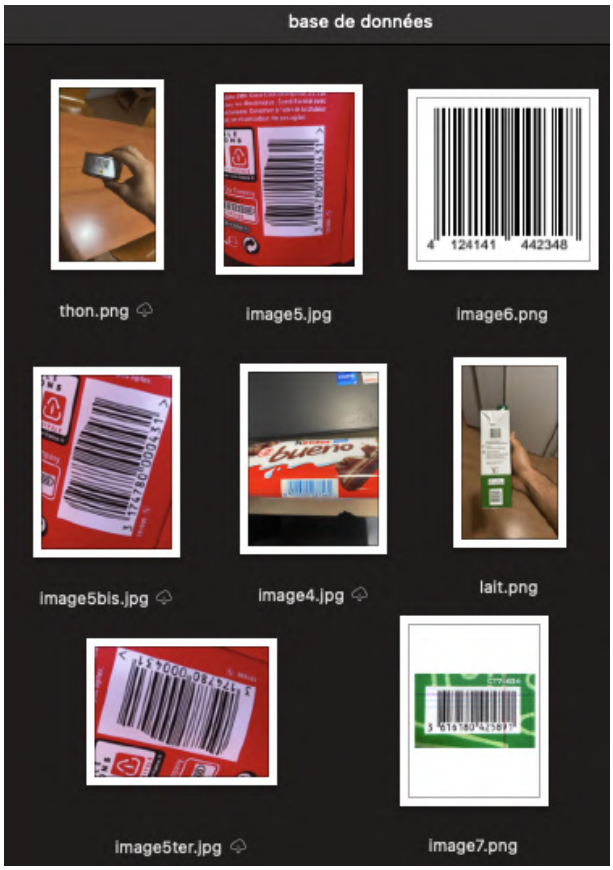


FIGURE 1 – Aperçu de la base de données de codes-barres utilisée pour les tests.

3 Implémentation

3.1 Segmentation

L'implémentation de la segmentation repose sur des méthodes de traitement d'images et d'analyse géométrique afin de repérer les zones potentiellement associées à un code-barres. Cette section détaille les différentes étapes mises en œuvre pour atteindre cet objectif.

3.1.1 Chargement et prétraitement

L'image est chargée avec la bibliothèque `scikit-image`. Si un canal alpha (transparence) est présent, il est supprimé. L'image est ensuite convertie en niveaux de gris afin de simplifier les calculs tout en conservant l'information essentielle sur les contrastes.

Un bruit gaussien est ajouté à l'image pour casser d'éventuels alignements parfaits des pixels. Cela permet d'améliorer la détection en simulant des perturbations réalistes.

3.1.2 Calcul des gradients

Des filtres gaussiens dérivés sont appliqués pour calculer les gradients horizontaux et verticaux. Ces gradients mettent en évidence les variations d'intensité, typiques des lignes parallèles présentes dans un code-barres.

Les gradients sont ensuite normalisés afin d'éviter des distorsions dues à des valeurs extrêmes et de garantir une analyse cohérente sur l'ensemble de l'image.

3.1.3 Analyse par tenseur de structure

Pour analyser la structure locale des gradients, un tenseur de structure est calculé. Un filtre gaussien supplémentaire est appliqué pour lisser les gradients et réduire le bruit résiduel.

Ainsi on obtient une carte de cohérence directionnelle, qui met en évidence les endroits potentiels du code-barre et des zones de bruit.

3.1.4 Segmentation et nettoyage

Un seuil est appliqué sur la carte de cohérence pour créer un masque binaire, pour isoler les régions potentiellement intéressantes et rejeter les zones de bruit.

Afin d'affiner ce masque, nous avons appliqué des techniques issues du cours de traitement d'images, en particulier des opérations morphologiques :

— **Fermeture :** pour relier les segments de lignes discontinues.

— **Ouverture :** pour éliminer les petites zones indésirables.

Les régions connectées dans le masque ont été identifiées à l'aide d'un étiquetage des composants connexes. Ensuite, nous avons conservé uniquement les régions présentant une taille significative, en éliminant les zones plus petites susceptibles de correspondre à du bruit ou à des motifs non pertinents. La région la plus étendue est supposée contenir le code-barres et est sélectionnée pour un traitement plus approfondi.

3.1.5 Délimitation des bornes

Pour déterminer précisément l'orientation et les limites de la région sélectionnée, une analyse en composantes principales (PCA) a été appliquée. Cette méthode permet d'ajuster un rectangle orienté autour de la zone détectée. Les quatre coins de ce rectangle sont ensuite calculés et définissent avec précision la zone d'intérêt pour les étapes suivantes. (Voir la sous-section *Détermination des Bornes du code-barre*).

Résultats obtenus : Les résultats de la segmentation sont présentés dans la figure 2. Ces exemples montrent que l'algorithme est capable d'isoler efficacement des régions contenant des motifs similaires à des codes-barres, même dans des contextes visuels variés.

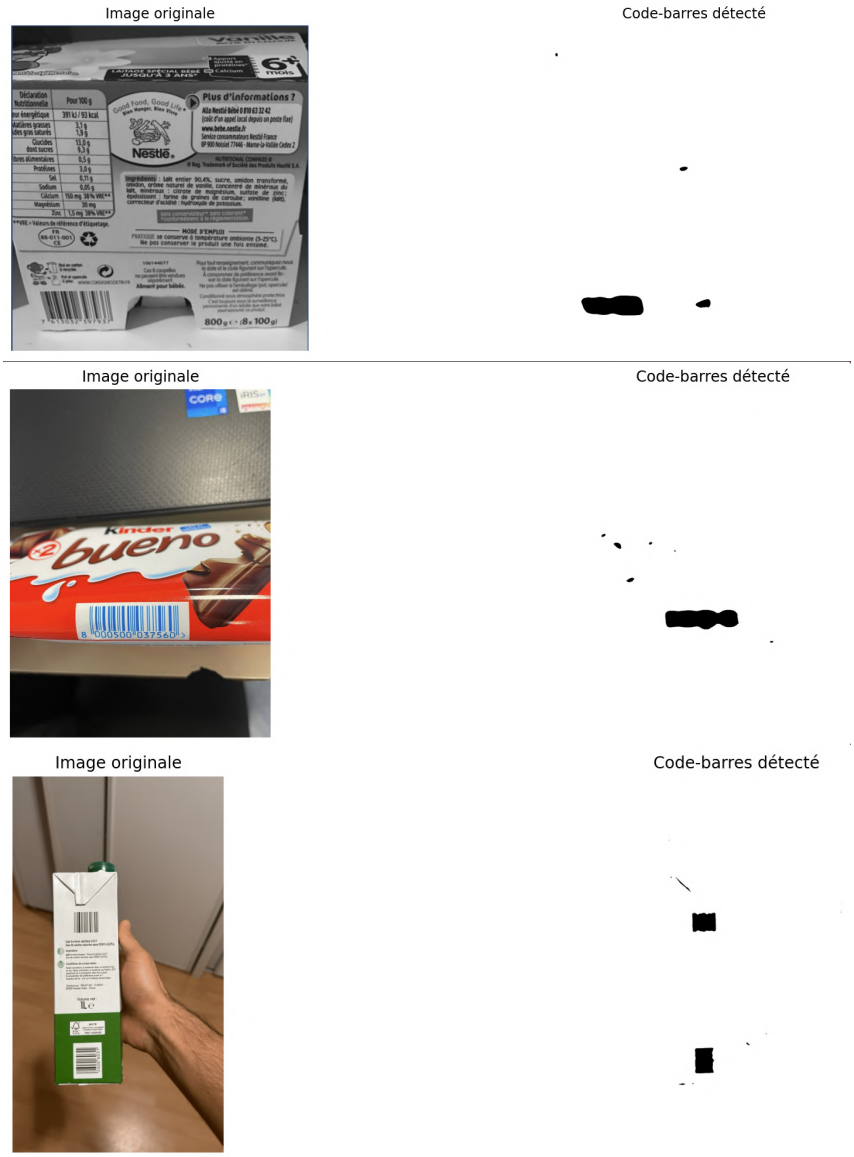


FIGURE 2 – Exemples de résultats obtenus après segmentation sur différentes images.

Limites et perspectives : Bien que cette approche fonctionne correctement sur des images claires et bien définies, elle présente certaines limites :

— Sensibilité au flou ou aux obstructions partielles dans l'image.

— Difficulté face à des variations extrêmes d'éclairage ou à des ombres marquées.

Pour rendre l'algorithme plus solide, on pourrait ajouter des approches basées sur l'intelligence artificielle, comme les réseaux de neurones convolutifs (CNN). Ce genre de méthode serait pratique pour gérer des cas un peu délicats, comme des images floues, des zones masquées ou des variations extrêmes d'éclairage.

3.2 Détermination des Bornes du code-barre

Pour implémenter la détermination des bornes du code-barres, nous avons fait plusieurs choix techniques importants :

Bibliothèques utilisées

Notre implémentation s'appuie principalement sur NumPy pour sa performance dans les cal-

culs matriciels, notamment pour la manipulation efficace des grilles de coordonnées et le calcul des vecteurs propres. scikit-image est utilisé pour les opérations morphologiques qui permettent de nettoyer les régions détectées.

Calcul du barycentre et matrice de covariance

Le calcul du barycentre utilise une moyenne pondérée par le masque final, car cela permet de prendre en compte uniquement les pixels appartenant effectivement au code-barres, en ignorant naturellement le bruit et les artefacts environnants. La matrice de covariance est ensuite construite avec une pondération par les gradients :

$$C = \begin{bmatrix} \frac{\sum(x-\bar{x})^2 w}{\sum w} & \frac{\sum(x-\bar{x})(y-\bar{y})w}{\sum w} \\ \frac{\sum(x-\bar{x})(y-\bar{y})w}{\sum w} & \frac{\sum(y-\bar{y})^2 w}{\sum w} \end{bmatrix} \tag{1}$$

où w représente les poids basés sur les gradients. Cette pondération par les gradients permet d'accorder plus d'importance aux zones de transition entre les barres noires et blanches, caractéristiques essentielles du code-barres, plutôt qu'aux régions uniformes. Ainsi, les vecteurs propres de cette matrice s'alignent naturellement avec la structure alternée du code-barres.

Optimisations

Les optimisations clés incluent :

- Utilisation de masques de Sobel simplifiés $[-1, 0, 1]$ au lieu des masques 3x3 classiques. Ce choix réduit le nombre d'opérations tout en conservant une détection efficace des contours horizontaux et verticaux du code-barres.
- Vectorisation des calculs de projection via NumPy : les opérations sur les coordonnées (X_c, Y_c) et les vecteurs propres sont effectuées en une seule instruction matricielle plutôt qu'en boucles itératives, accélérant significativement les calculs.
- Gestion optimisée de la mémoire en réutilisant les tableaux existants pour les calculs intermédiaires (gradients, projections) et en évitant la création de copies temporaires lors des opérations de transposition et de reshape.

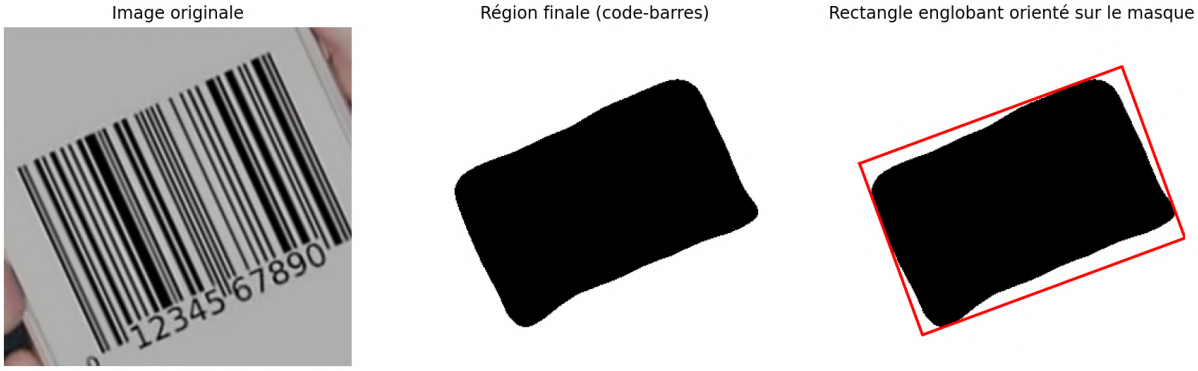
Gestion des situations complexes

Notre implémentation prend en charge :

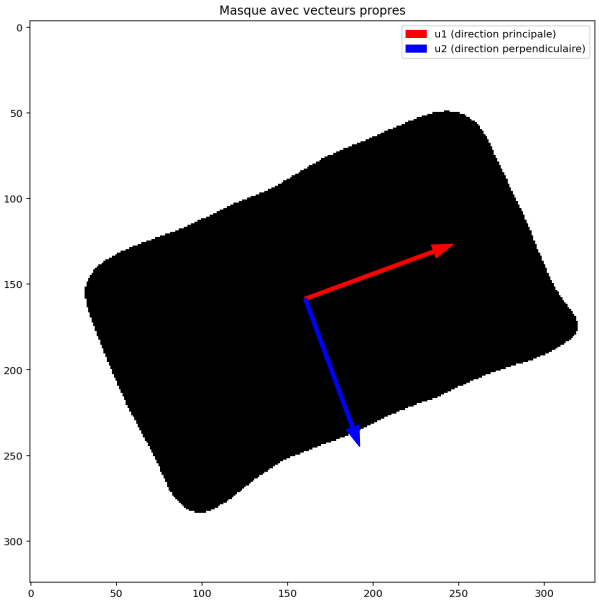
- Les codes-barres fortement inclinés grâce à l'analyse en composantes principales : les vecteurs propres de la matrice de covariance fournissent naturellement les directions principales du code-barres, permettant de construire un rectangle orienté précis quelle que soit l'inclinaison.
- Les cas de détection multiple : nous sélectionnons la plus grande région connexe, qui correspond généralement au code-barres principal dans l'image. Cette approche simple mais efficace évite les faux positifs.
- L'absence de code-barres : notre algorithme vérifie que la somme des pixels du masque final est non nulle et que les dimensions du rectangle détecté correspondent aux proportions attendues d'un code-barres. Cette validation permet d'identifier rapidement l'absence de code-barres dans l'image.

Cette approche permet une détermination précise des bornes même dans des conditions difficiles, tout en maintenant des performances de calcul optimales.

Résultats et Validation Les résultats de l'implémentation sont illustrés dans les figures suivantes :



(a) De gauche à droite : Image originale, Région finale (code-barres), Rectangle englobant orienté sur le masque



(b) Vecteurs propres de la région détectée (rouge : direction principale, bleu : direction perpendiculaire)

FIGURE 3 – Validation visuelle des bornes détectées et orientation principale

3.3 Lancers des rayons et Extraction

3.3.1 Lancer manuel du rayon

Pour tester et visualiser l'extraction des signatures sur un code-barres, nous avons développé une interface graphique interactive. Cette interface permet à l'utilisateur de sélectionner manuellement deux points sur l'image pour définir un rayon à analyser.

Fonctionnement : L'utilisateur commence par cliquer sur un premier point de l'image, marquant ainsi le début du rayon (voir figure 4). Ensuite, un deuxième clic définit la fin du rayon. Une fois les deux points sélectionnés, le traitement est automatiquement lancé pour extraire la signature le long de ce rayon (voir figure 5).

L'application affiche les résultats sous forme de graphiques (voir figure 6) :

— La première signature extraite.
— La version binarisée de cette signature après application du seuil d'Otsu.
— Une signature finale ajustée et binarisée sur 95 bits, prête pour l'étape de décodage.

— Le critère d'Otsu en fonction des seuils, montrant le choix optimal.
— La signature binaire finale affichée sous forme numérique (figure 7).

Résultats obtenus : Les figures suivantes illustrent les étapes du processus : La figure 4 montre la sélection des points. La figure 5 affiche le rayon tracé sur l'image. La figure 6 présente les courbes issues du traitement, et la figure 7 montre la signature finale binarisée sous forme de tableau.

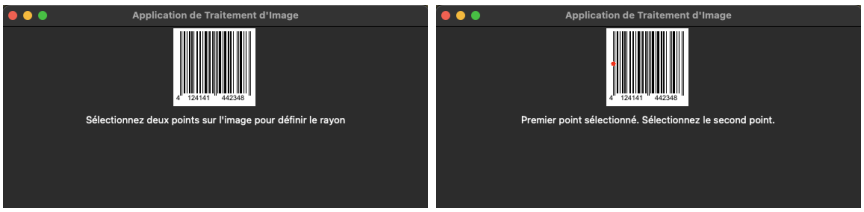


FIGURE 4 – Sélection des points pour définir un rayon manuel.

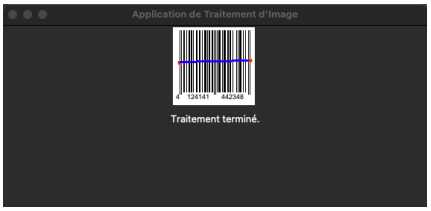


FIGURE 5 – Rayon manuel tracé après sélection des points.

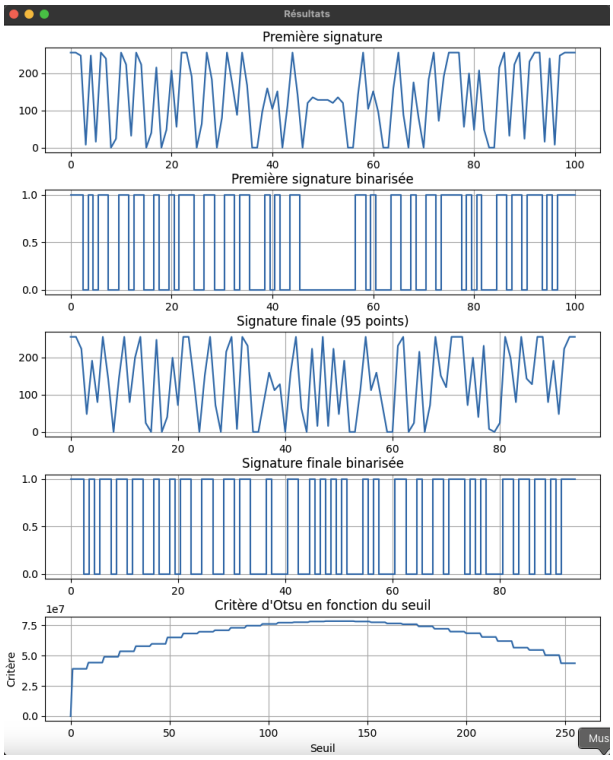


FIGURE 6 – Résultats graphiques de l'extraction et de la binarisation de la signature.

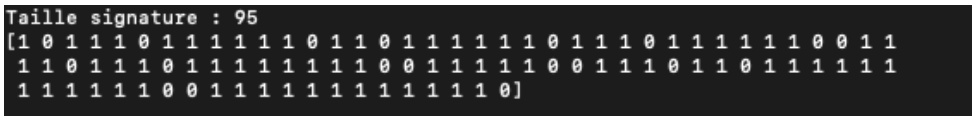


FIGURE 7 – Signature finale binarisée de 95 bits affichée en sortie.

Limites et améliorations : Le lancer manuel est utile pour valider visuellement le processus, mais il nécessite une intervention humaine. Pour automatiser davantage, la section suivante détaille l'approche basée sur des lancers aléatoires de rayons.

3.3.2 Lancers aléatoires des rayons

Implémentation des lancers avec angles aléatoires contraints

L'implémentation de notre système de génération de rayons contraints repose sur des choix techniques spécifiques pour garantir efficacité et robustesse.

Contrôle de l'angle

Le contrôle de l'angle est réalisé par :

- Le calcul de la direction principale du rectangle par normalisation du vecteur $\vec{C_2C_1}$(où C_1 et C_2 sont les coins inférieurs du rectangle englobant le code-barres), car cette direction correspond à l'orientation naturelle du code-barres. Le vecteur est normalisé pour simplifier les calculs d'angles ultérieurs.
- Une contrainte angulaire de ± 30 degrés ($\pi/6$ radians), car les tests montrent que les rayons au-delà de cet angle traversent trop peu de barres pour permettre une lecture fiable.
- La vérification via le produit scalaire plutôt que par calcul trigonométrique direct, car le produit scalaire entre vecteurs normalisés donne directement le cosinus de l'angle, évitant ainsi des calculs supplémentaires.

Bibliothèques et calculs

Notre implémentation utilise NumPy pour ses avantages :

- Opérations vectorisées évitant les boucles Python, ce qui accélère significativement le traitement des coordonnées et des angles
- Génération aléatoire uniforme optimisée qui assure une bonne distribution spatiale des rayons
- Gestion efficace de la mémoire pour le traitement des grands tableaux de coordonnées

Génération et validation des rayons

La génération des points utilise une interpolation linéaire simple :

$$P = P_1 + t(P_2 - P_1), \quad t \in [0, 1] \tag{2}$$

Cette approche est choisie car :

- Le paramètre t entre 0 et 1 assure que les points générés sont bien sur les segments du rectangle, évitant tout point en dehors
- L'utilisation du produit scalaire pour calculer l'angle nécessite moins d'opérations que les formules trigonométriques classiques
- La limite de 100 essais est un compromis : elle laisse suffisamment de tentatives tout en évitant une boucle infinie

Les tests montrent que notre approche de génération de rayons aléatoires permet une exploration de la zone d'intérêt comme illustré sur la figure 8.

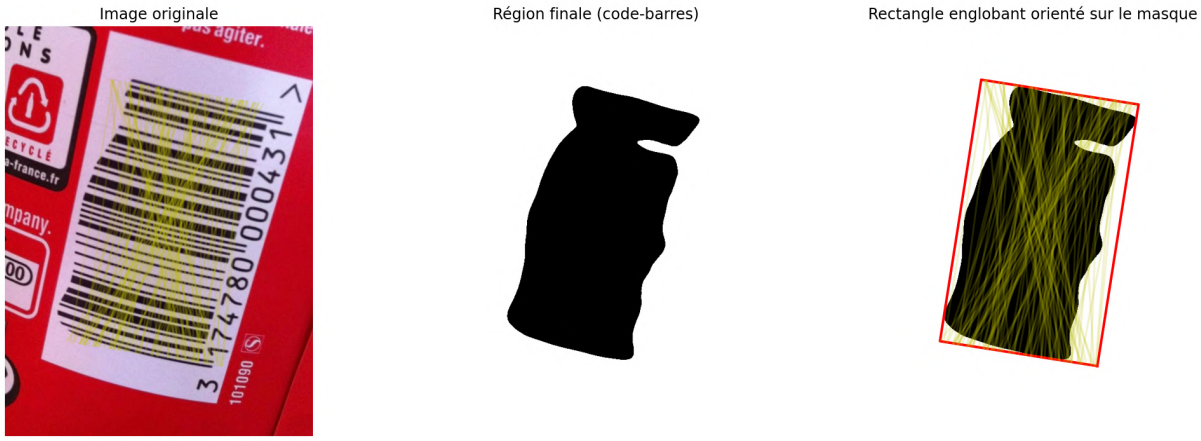


FIGURE 8 – Visualisation des rayons testés (jaune) montrant une bonne couverture de la zone

3.4 Décodage

L'étape de décodage permet de convertir la signature binaire extraite et binarisée en un code-barres au format EAN-13. Ce processus repose sur l'analyse de motifs binaires spécifiques et la vérification d'une clé de contrôle.

Principe du décodage : Le format EAN-13 suit une structure standard avec des motifs de garde et des séquences codées pour représenter 13 chiffres. La signature binaire de 95 bits est divisée en plusieurs segments :

- Trois bits pour la **garde gauche**.
- Six groupes de 7 bits pour les **chiffres de gauche**.
- Cinq bits pour la **garde central**.
- Six groupes de 7 bits pour les **chiffres de droite**.
- Trois bits pour la **garde droit**.

Chaque chiffre est encodé selon trois tables :

- Table **L** pour les chiffres de gauche avec parité paire.
- Table **G** pour les chiffres de gauche avec parité impaire.
- Table **R** pour les chiffres de droite (complémentaire).

L'implémentation du décodage se déroule en plusieurs étapes :

1. Vérification de la longueur de la signature (95 bits).
2. Validation des motifs de garde (gauche, central, droit).
3. Analyse des 6 premiers chiffres (zone gauche) :
   - Identification des motifs binaires selon les tables **L** et **G**.
   - Détermination du premier chiffre grâce à un motif de parité.
4. Décodage des 6 chiffres restants (zone droite) en utilisant la table **R**.
5. Reconstruction du code complet en combinant les chiffres décodés.
6. Vérification de la clé de contrôle :
   - Calcul de la somme pondérée des chiffres (pairs et impairs).
   - Validation avec le dernier chiffre comme clé de contrôle.

Exemple d'exécution : Un exemple de signature binaire est fourni pour un code-barres donné (figure 9). Le programme valide d'abord les motifs de garde, puis décode les chiffres en utilisant les tables de correspondance. En sortie, il affiche le code-barres décodé ou signale une erreur si la signature est invalide.

```
Code EAN-13 à décoder : 4006381333931
Signature EAN-13 à décoder : 10100011010100111010111011111010001001011001101010100001010000101000010111010010000101100110101
Code-barres EAN-13 décodé : 4006381333931
```

FIGURE 9 – Exemple de signature binaire extraite et binarisée.

Le décodage fonctionne correctement pour des signatures bien extraites et binarisées. Cependant, en présence de bruit ou d'imperfections dans la segmentation, des erreurs peuvent survenir.

Limites et perspectives : L'algorithme actuel repose fortement sur la qualité de l'extraction. Il fonctionne efficacement pour des images claires et bien alignées. Pour des cas plus complexes (flou, angles inclinés), des techniques d'apprentissage automatique pourraient être intégrées pour améliorer la fiabilité.

4 Interface graphique (Application)

Dans cette section, nous présentons l'interface graphique développée pour la mise en œuvre et l'intégration des différentes fonctionnalités de traitement d'images liées à la lecture et au décodage des codes-barres.

L'objectif principal de cette interface graphique est de regrouper, au sein d'une seule plateforme, l'ensemble des traitements nécessaires au décodage d'un code-barres, tout en offrant une visualisation en temps réel des résultats intermédiaires. Elle permet notamment :

- Le chargement d'une image contenant un code-barres.
- La segmentation automatique de la région d'intérêt.
- Le choix entre des lancers de rayons aléatoires ou manuels pour l'extraction de la signature.
- L'affichage et la visualisation des courbes correspondant aux différentes étapes du traitement.
- Le décodage et l'affichage du code-barres extrait.
- La vérification de la présence du code-barres dans une base de données chargée par l'utilisateur.
- Des fonctionnalités interactives telles que le zoom, le déplacement et la réinitialisation.

4.1 Structure et organisation

L'interface a été développée en utilisant la bibliothèque `Tkinter` de Python pour une intégration native sur la plupart des systèmes d'exploitation. Son architecture est divisée en plusieurs sections :

1. **Zone de commande :** Située en haut de l'interface, elle contient des boutons et des options pour effectuer les différentes étapes du traitement (segmentation, extraction, décodage, etc.).
2. **Zone d'affichage de l'image :** Placée au centre, cette zone permet de visualiser l'image chargée ainsi que les résultats intermédiaires des traitements appliqués.
3. **Zone de feedback :** Située en bas, elle affiche des messages indiquant l'état des traitements en cours, offrant un retour visuel pour suivre l'avancement des opérations.

FIGURE 10 – Page d'accueil de l'app dévloppée.

4.2 Fonctionnalités développées

Pour répondre aux objectifs du projet, nous avons intégré plusieurs fonctionnalités interactives :

1. Chargement d'image : L'utilisateur peut sélectionner une image via un explorateur de fichiers. Une fois l'image chargée, elle est affichée dans la zone centrale pour faciliter la visualisation.

2. Segmentation : Cette étape applique un algorithme de segmentation basé sur les gradients et la cohérence directionnelle. La région contenant potentiellement un code-barres est mise en évidence, et son contour est affiché sur l'image. Une courbe représentant les résultats intermédiaires est également affichée.

3. Extraction des signatures : L'utilisateur peut choisir entre des rayons manuels ou aléatoires pour extraire la signature binaire. Les signatures extraites sont affichées pour faciliter l'analyse visuelle.

4. Décodage : La signature extraite est traitée et décodée en un code-barres conforme au format `EAN-13`. Le résultat est affiché dans la zone de feedback.

5. Vérification dans la base de données : L'interface permet de charger une base de données (fichier .txt) contenant des codes-barres valides. Le code-barres décodé est comparé à ceux présents dans cette base, et l'utilisateur est informé du résultat (présence ou absence).

6. Réinitialisation : Un bouton de réinitialisation permet de rétablir l'état initial de l'application, effaçant toutes les données précédemment traitées.

7. Quitter : Un bouton 'quitter' qui permet de fermer l'application.

L'interface graphique a été conçue avec un design simple et professionnel, comprenant un dégradé de fond pour un rendu visuellement agréable. Les boutons et options sont disposés de manière logique pour simplifier l'utilisation, même pour un utilisateur non technique.

Des messages de feedback dynamiques guident l'utilisateur tout au long des traitements, fournissant des informations sur l'état des calculs et sur les erreurs éventuelles rencontrées.

4.3 Défis rencontrés et solutions

Le développement de cette interface a impliqué plusieurs défis, notamment :

- **Gestion des erreurs :** Assurer un retour clair et rapide en cas d'erreurs de chargement d'images ou d'échec de traitement.
- **Affichage interactif :** Intégrer des graphiques dynamiques pour visualiser les résultats intermédiaires.
- **Intégration modulaire :** Connecter efficacement les différentes fonctionnalités tout en maintenant un code propre et réutilisable.
- **Réactivité :** Maintenir une interface réactive même lors de l'exécution de traitements complexes grâce à la mise à jour dynamique des éléments d'affichage.

Pour résoudre ces problèmes, nous avons adopté une approche modulaire, où chaque fonctionnalité est implémentée sous forme de fonctions indépendantes mais connectées. Cela facilite la maintenance et les futures améliorations. (fichier fonctions.py)

4.4 Perspectives d'amélioration

Bien que l'interface actuelle réponde aux besoins définis, plusieurs améliorations peuvent être envisagées :

- Ajouter des fonctionnalités de traitement avancé basées sur l'ia.
- Ajouter une acquisition en temps réel depuis un appareil photo.
- Permettre l'enregistrement des résultats et des graphiques générés.
- Intégrer un mode de traitement par lots pour analyser plusieurs images en une seule session.
- Optimiser davantage les performances pour gérer des images plus lourdes ou des bases de données étendues.

4.5 Conclusion

Cette interface graphique constitue une solution simple et interactive pour l'analyse et le décodage des codes-barres. Elle met en œuvre toutes les étapes clés du traitement d'image, de la segmentation à la vérification dans une base de données. Grâce à son ergonomie et ses fonctionnalités riches, elle répond aux exigences du projet tout en offrant une base pour des extensions futures.

5 Conclusion

Ce projet avait pour objectif de développer un système capable de lire et décoder des codes-barres **EAN-13** à partir d'images en utilisant une approche basée sur des lancers de rayons. Au fil des étapes, nous avons mis en place une logique allant de la segmentation des régions d'intérêt jusqu'au décodage final du code-barres.

Même si les résultats finaux sont globalement satisfaisants, le chemin pour y parvenir n'a pas été simple. Nous avons rencontré plusieurs difficultés, aussi bien techniques qu'organisationnelles. Sur le plan technique, il a fallu gérer des cas complexes comme des images avec des orientations variées, des tailles différentes, des éclairages inégaux ou encore des motifs parasites qui rendaient la détection moins évidente. Pour surmonter ces défis, nous avons dû affiner nos algorithmes et expérimenter différentes méthodes, en particulier pour la segmentation et la binarisation des signatures.

Sur le plan humain, l'une des principales difficultés a été la gestion des contributions au sein du groupe. Tout le monde n'a pas participé de manière équilibrée, ce qui a obligé certains membres à assumer davantage de responsabilités pour maintenir le projet sur la bonne voie. Ce déséquilibre a parfois ralenti notre progression et ajouté un peu de pression, surtout à l'approche des échéances. Malgré cela, nous avons réussi à tenir nos engagements et à fournir une solution fonctionnelle.

Ce projet nous a aussi permis d'apprendre énormément, que ce soit sur le traitement d'images, la gestion d'algorithmes complexes ou encore l'organisation d'un travail d'équipe. Il a mis en évidence l'importance d'être adaptable face aux imprévus et de savoir ajuster ses méthodes en fonction des résultats obtenus.

Pour l'avenir, plusieurs pistes d'amélioration pourraient être envisagées :

— Intégrer des modèles d'apprentissage automatique pour gérer les cas plus complexes comme des codes-barres flous ou partiellement masqués.
— Optimiser les performances pour traiter des images plus rapidement tout en conservant une bonne précision.
— Enrichir la base de données avec des scénarios encore plus variés pour rendre le système plus robuste.

En résumé, ce projet a été un vrai défi, mais il nous a permis de développer des compétences techniques solides tout en mettant en lumière l'importance de la collaboration et de la gestion de projet dans un cadre pratique.

6 Implication de chacun des eleves

Le tableau ci-dessous résume les contributions spécifiques de chaque membre de l'équipe au projet. Chaque tâche a été répartie en fonction des compétences et des disponibilités des participants.

Tâche	**Responsable(s)**
Détection des régions d'intérêt	Mehdi
Détection des bornes du code-barres	Arif
Lancers aléatoires des rayons	Mehdi (50%), Arif (50%)
Lancer manuel des rayons	Mehdi (50%) , Arif (50%)
Extraction des signatures	Mehdi (60%) , Arif (40%)
Décodage du code-barre	Mehdi
Base de données	Salma (75%), Mehdi (25%)
Interface graphique	Mehdi (60%), Salma (60%)

TABLE 1 – Répartition des tâches entre les membres de l'équipe.

Remarque : La répartition des tâches a dû être ajustée en cours de projet en raison d'un déséquilibre dans les contributions individuelles. Certains membres ont dû assumer des responsabilités supplémentaires pour compenser ce manque, ce qui a rendu la gestion du projet plus complexe et a nécessité un effort supplémentaire pour respecter les délais.

7 Annexes

7.1 Code du programme principal

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 13 10:21:31 2024

@author: Mehdi

main program
"""

################################################################################
#                                IMPORTATIONS                                  #
################################################################################

import os
from fonctions import segmentation, lancer_aleatoire, extraction,
    decode_ean13_signature

################################################################################
#                                  MAIN                                        #
################################################################################

def main():
    """
    Programme principal :
```

```
    1. Charge une image fournie par l'utilisateur.
    2. Effectue la segmentation pour detecter la region d'interet.
    3. Lance des rayons aleatoires pour extraire des signatures.
    4. Tente de decoder le code-barres EAN-13 jusqu'a reussir ou atteindre la
       limite.
    """
    # Demande a l'utilisateur de fournir un chemin d'image valide
    image_path = input("Veuillez entrer le chemin du fichier image : ")
    while not os.path.exists(image_path):
        print("Fichier introuvable. Veuillez reessayer.")
        image_path = input("Veuillez entrer un chemin valide : ")

    # etape 1 : Segmentation pour detecter la region d'interet
    try:
        min_row, min_col, max_row, max_col = segmentation(image_path)
        print("Segmentation reussie. Region detectee.")
    except Exception as e:
        print(f"Erreur lors de la segmentation : {e}")
        return

    # Definir les coins de la region detectee
    p1 = (min_col, min_row)
    p2 = (max_col, min_row)
    p3 = (max_col, max_row)
    p4 = (min_col, max_row)

    # etape 2 : Recherche d'un code-barres en lancant des rayons aleatoires
    max_attempts = 20  # Limite d'essais
    attempt = 0
    code_barres = None

    while attempt < max_attempts:
        print(f"Tentative {attempt + 1}/{max_attempts}...")

        # Generer un rayon aleatoire
        point1, point2 = lancer_aleatoire(p1, p2, p3, p4)

        # Extraire la signature le long du rayon genere
        signature_95bits = extract_signature(image_path, point1, point2)

        if signature_95bits is not None:
            try:
                # Tenter de decoder la signature
                code_barres = decode_ean13_signature(signature_95bits)
                print(f"Code-barres detecte : {code_barres}")
                break  # Arreter la boucle si un code valide est trouve
            except ValueError as e:
                print(f"Erreur de decodage : {e}")
        else:
            print("Signature non valide ou non extraite correctement.")

        attempt += 1

    # etape 3 : Resultat final
    if code_barres:
        print(f"Code-barres final detecte : {code_barres}")
    else:
        print("echec de la detection apres plusieurs tentatives.")


################################################################################
#                              EXECUTION DU PROGRAMME                          #
```

```
################################################################################

if __name__ == "__main__":
    main()
```

Listing 1 – main.py

7.2 Code des fonctions

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Jan  3 18:11:26 2025

@author: Mehdi

Diverses Fonctions

"""


################################################################################
#                                IMPORTATIONS                                  #
################################################################################

# Modules standards
import os
import random

# Modules scientifiques
import numpy as np
from scipy.signal import convolve2d
from skimage import color, io
from skimage.morphology import closing, opening, square
from skimage.measure import label, regionprops
from skimage.filters import threshold_otsu

# Modules pour l'interface graphique
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from scipy.ndimage import map_coordinates

################################################################################
#                                SEGMENTATION                                  #
################################################################################

def segmentation(image_path):
    # Chargement de l'image
    img = io.imread(image_path)
    if img.shape[-1] == 4:
        img = img[..., :3]
    I = color.rgb2gray(img)

    # Parametres
    sigma_noise = 0.02
    sigma_G = 1.8
    sigma_T = 18
    seuil_coherence = 0.3
```

```
    # Ajout de bruit
    bruit = np.random.normal(0, sigma_noise, I.shape)
    I_bruite = np.clip(I + bruit, 0, 1)

    # Calcul des gradients
    size = int(3 * sigma_G)
    x, y = np.meshgrid(np.arange(-size, size+1), np.arange(-size, size+1))

    G_x = -(x / (2 * np.pi * sigma_G**4)) * np.exp(-(x**2 + y**2) / (2*sigma_G
        **2))
    G_y = -(y / (2 * np.pi * sigma_G**4)) * np.exp(-(x**2 + y**2) / (2*sigma_G
        **2))

    I_x = convolve2d(I_bruite, G_x, mode='same', boundary='symm')
    I_y = convolve2d(I_bruite, G_y, mode='same', boundary='symm')

    norme = np.sqrt(I_x**2 + I_y**2) + 1e-8
    I_x /= norme
    I_y /= norme

    # Tenseur de structure
    size_T = int(2 * sigma_T)
    G = (1 / (2 * np.pi * sigma_T**2)) * np.exp(-(x**2 + y**2) / (2 * sigma_T
        **2))

    T_xx = convolve2d(I_x**2, G, mode='same', boundary='symm')
    T_xy = convolve2d(I_x * I_y, G, mode='same', boundary='symm')
    T_yy = convolve2d(I_y**2, G, mode='same', boundary='symm')

    D1 = 1 - np.sqrt((T_xx - T_yy)**2 + 4 * (T_xy**2)) / (T_xx + T_yy + 1e-15)

    # Segmentation
    M = (D1 > seuil_coherence).astype(int)
    M_clean = closing(M, square(3))
    M_clean = opening(M_clean, square(2))

    labels = label(M_clean)
    if labels.max() > 0:
        regions = regionprops(labels)
        largest_region = max(regions, key=lambda r: r.area)
        return largest_region.bbox
    else:
        raise ValueError("Aucune region coherente detectee.")


##############################################################################
#                            RAYON ALEATOIRE                                 #
##############################################################################
def lancer_aleatoire(C1, C2, C3, C4):

    """
    Genere un rayon aleatoire ou oriente dans une zone delimitee par 4 coins.

    Parametres:
        C1, C2, C3, C4 (tuple): Coordonnees des coins de la region (x, y).

    Retourne:
        point1, point2 (tuple): Coordonnees des points de depart et d'arrivee du
            rayon.

    Remarque:
        - Les rayons peuvent etre orientes selon un angle aleatoire autour du
```

```
            centre de la zone.
        - Cela permet d'explorer differentes directions pour couvrir plus de
            possibilites de detection.
    """

    # Verification des coordonnees
    if not all(len(point) == 2 for point in [C1, C2, C3, C4]):
        raise ValueError("Tous les points doivent avoir deux coordonnees (x, y).
            ")

    # Generer un angle aleatoire
    angle = np.random.uniform(0, 360)
    angle_rad = np.radians(angle)

    # Calculer le centre de la region
    center_x = (C1[0] + C2[0] + C3[0] + C4[0]) / 4
    center_y = (C1[1] + C2[1] + C3[1] + C4[1]) / 4

    # Rayon depuis le centre dans la direction donnee
    radius = min(np.linalg.norm(np.array(C1) - np.array([center_x, center_y])),
                 np.linalg.norm(np.array(C3) - np.array([center_x, center_y])))

    # Calcul des points
    x_start = center_x + radius * np.cos(angle_rad)
    y_start = center_y + radius * np.sin(angle_rad)
    x_end = center_x - radius * np.cos(angle_rad)
    y_end = center_y - radius * np.sin(angle_rad)

    return (x_start, y_start), (x_end, y_end)



################################################################################
#                                  EXTRACTION                                  #
################################################################################


def extraction(image, p1, p2):
    """
    Extrait une signature binaire de 95 bits le long d'un rayon defini par deux
        points.

    Parametres:
        - image (numpy.ndarray): Image en niveaux de gris.
        - p1 (tuple): Point de depart du rayon (x, y).
        - p2 (tuple): Point d'arrivee du rayon (x, y).

    Retourne:
        - signature_95bits (list): Liste de 95 bits representant la signature
            extraite.
    """

    # etape 1 : Calcul de la longueur du rayon
    longueur_rayon = int(np.sqrt((p2[0] - p1[0])**2 + (p2[1] - p1[1])**2))
    print(f"Longueur du rayon: {longueur_rayon} pixels")

    # etape 2 : Extraction initiale de la signature
    nb_points = max(longueur_rayon, 95)  # Nombre de points a echantillonner
    t = np.linspace(0, 1, nb_points)    # Parametre d'interpolation

    # Calcul des coordonnees du rayon
    x = p1[0] + (p2[0] - p1[0]) * t
```

```
    y = p1[1] + (p2[1] - p1[1]) * t

    # Extraction des intensites sur le rayon avec interpolation bilineaire
    intensities = map_coordinates(image, [y, x], order=1, mode='reflect')

    # etape 3 : Application du seuil d'Otsu
    threshold = threshold_otsu(intensities)  # Calcul du seuil d'Otsu
    binary_signature = (intensities > threshold).astype(int)  # Binarisation

    # etape 4 : Trouver les limites utiles
    non_zero = np.nonzero(binary_signature)[0]
    if len(non_zero) == 0:
        print("Aucune region utile trouvee dans la signature.")
        return None  # Retourne None si aucune donnee utile

    # etape 5 : Reduction aux indices utiles
    start_idx = non_zero[0]
    end_idx = non_zero[-1]

    # Coordonnees des points utiles
    t_start = start_idx / nb_points
    t_end = end_idx / nb_points

    useful_p1 = (
        p1[0] + (p2[0] - p1[0]) * t_start,
        p1[1] + (p2[1] - p1[1]) * t_start
    )
    useful_p2 = (
        p1[0] + (p2[0] - p1[0]) * t_end,
        p1[1] + (p2[1] - p1[1]) * t_end
    )

    # etape 6 : Ajustement et extraction finale
    useful_length = np.sqrt((useful_p2[0] - useful_p1[0])**2 + (useful_p2[1] -
        useful_p1[1])**2)
    u = max(1, int(useful_length / 95))  # Calcul de l'unite de base
    print(f"Unite de base u calculee : {u}")

    # etape 7 : Extraction finale sur 95 * u points
    nb_points_final = 95 * u
    t = np.linspace(0, 1, nb_points_final)
    x = useful_p1[0] + (useful_p2[0] - useful_p1[0]) * t
    y = useful_p1[1] + (useful_p2[1] - useful_p1[1]) * t

    # Extraction finale avec interpolation
    final_signature = map_coordinates(image, [y, x], order=1, mode='reflect')

    # Binarisation finale avec Otsu
    final_threshold = threshold_otsu(final_signature)
    final_binary_signature = (final_signature > final_threshold).astype(int)

    # etape 8 : Selection des 95 premiers bits
    if len(final_binary_signature) >= 95:
        signature_95bits = final_binary_signature[:95]
        return signature_95bits  # Retourne la signature binaire
    else:
        print("La signature extraite est trop courte.")
        return None


################################################################################
#                                 DECODAGE                                     #
```

```
##############################################################################

def decode_ean13_signature(binary_signature):
    """
    Decoder un code-barres EAN-13 a partir de sa signature binaire.

    Parametres:
    - binary_signature: liste d'entiers (0 ou 1), representant la signature
        binaire du code-barres (95 bits)

    Retourne:
    - code_barres: chaine de caracteres representant le code EAN-13 decode
    """
    # Verifier la longueur de la signature
    if len(binary_signature) != 95:
        raise ValueError("La signature binaire doit contenir exactement 95 bits.
            ")

    # Motifs de garde
    guard_left = [1, 0, 1]
    guard_center = [0, 1, 0, 1, 0]
    guard_right = [1, 0, 1]

    # Verifier les motifs de garde
    if binary_signature[0:3] != guard_left:
        raise ValueError("Motif de garde gauche incorrect.")
    if binary_signature[45:50] != guard_center:
        raise ValueError("Motif de garde central incorrect.")
    if binary_signature[92:95] != guard_right:
        raise ValueError("Motif de garde droit incorrect.")

    # Tables de codage pour les chiffres
    code_L = {
        '0001101': '0',
        '0011001': '1',
        '0010011': '2',
        '0111101': '3',
        '0100011': '4',
        '0110001': '5',
        '0101111': '6',
        '0111011': '7',
        '0110111': '8',
        '0001011': '9',
    }

    code_G = {
        '0100111': '0',
        '0110011': '1',
        '0011011': '2',
        '0100001': '3',
        '0011101': '4',
        '0111001': '5',
        '0000101': '6',
        '0010001': '7',
        '0001001': '8',
        '0010111': '9',
    }

    code_R = {
        '1110010': '0',
        '1100110': '1',
        '1101100': '2',
```

```
    '1000010': '3',
    '1011100': '4',
    '1001110': '5',
    '1010000': '6',
    '1000100': '7',
    '1001000': '8',
    '1110100': '9',
}

# Table de parite pour determiner le premier chiffre
parity_table = {
    'LLLLLL': '0',
    'LLGLGG': '1',
    'LLGGLG': '2',
    'LLGGGL': '3',
    'LGLLGG': '4',
    'LGGLLG': '5',
    'LGGGLL': '6',
    'LGLGLG': '7',
    'LGLGGL': '8',
    'LGGLGL': '9',
}

# Decodage des 6 chiffres de gauche
left_digits = []
parity_pattern = ''
for i in range(6):
    start = 3 + i * 7
    end = start + 7
    pattern_bits = binary_signature[start:end]
    pattern = ''.join(map(str, pattern_bits))

    if pattern in code_L:
        digit = code_L[pattern]
        parity_pattern += 'L'
    elif pattern in code_G:
        digit = code_G[pattern]
        parity_pattern += 'G'
    else:
        raise ValueError(f"Motif inconnu dans la partie gauche : {pattern}")
    left_digits.append(digit)

# Determiner le premier chiffre a partir du motif de parite
if parity_pattern in parity_table:
    first_digit = parity_table[parity_pattern]
else:
    raise ValueError(f"Motif de parite inconnu : {parity_pattern}")

# Decodage des 6 chiffres de droite
right_digits = []
for i in range(6):
    start = 50 + i * 7
    end = start + 7
    pattern_bits = binary_signature[start:end]
    pattern = ''.join(map(str, pattern_bits))

    if pattern in code_R:
        digit = code_R[pattern]
    else:
        raise ValueError(f"Motif inconnu dans la partie droite : {pattern}")
    right_digits.append(digit)
```

```
    # Construire le code-barres complet
    code_barres = first_digit + ''.join(left_digits) + ''.join(right_digits)

    # Verifier la cle de contr le
    digits = list(map(int, code_barres))
    if len(digits) != 13:
        raise ValueError("Le code-barres doit contenir 13 chiffres.")

    # Calcul de la cle de contr le selon la norme EAN-13
    # etape 1 : Somme des chiffres en positions paires (2e, 4e, ..., 12e)
    sum_even = sum(digits[i] for i in range(1, 12, 2))

    # etape 2 : Multiplier la somme par 3
    sum_even *= 3

    # etape 3 : Somme des chiffres en positions impaires (1ere, 3e, ..., 11e)
    sum_odd = sum(digits[i] for i in range(0, 12, 2))

    # etape 4 : Calculer le total
    total = sum_even + sum_odd

    # etape 5 : Calculer la cle de contr le
    check_digit = (10 - (total % 10)) % 10

    # Verifier si la cle de contr le est correcte
    if check_digit != digits[-1]:
        raise ValueError(f"Cle de contr le invalide : attendu {check_digit},
            obtenu {digits[-1]}")

    return code_barres
```

Listing 2 – Code des fonctions

7.3 Code de l'interface graphique

```
import tkinter as tk
from tkinter import filedialog, messagebox
from PIL import Image, ImageTk
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import os
from fonctions import segmentation, extraction, lancer_aleatoire,
    decode_ean13_signature  # Import des fonctions


class BarcodeApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Lecteur de Code-Barres")
        self.geometry("1200x800")
        self.minsize(800, 600)
        self.resizable(True, True)


        # Degrade de fond
        self.canvas = tk.Canvas(self, width=1200, height=800)
        self.canvas.pack(fill="both", expand=True)
        self.create_gradient()

        # Variables
```

```
        self.image = None
        self.image_path = ""
        self.binary_signature = None
        self.decoded_barcode = None

        # Creer l'interface
        self.setup_ui()


    def create_gradient(self):
        """Creer un degrade de fond."""
        for i in range(100):
            color = f"#{i:02x}{i:02x}{255-i:02x}"
            self.canvas.create_rectangle(0, i * 9, 1200, (i + 1) * 3, fill=color
                , outline="")

    def setup_ui(self):

        """Creer les widgets de l'interface."""
        self.frame = tk.Frame(self.canvas)
        self.frame.place(relx=0.5, rely=0.05, anchor="n")

        # Bouton pour charger une image
        self.load_button = tk.Button(self.frame, text="Charger une image",
            command=self.load_image,  fg="black", font=("Arial", 12))
        self.load_button.grid(row=0, column=0, padx=10, pady=5)

        # Boutons pour selectionner le mode
        self.mode_var = tk.StringVar(value="manuel")
        tk.Radiobutton(self.frame, text="Rayon Manuel", variable=self.mode_var,
            value="manuel", font=("Arial", 12)).grid(row=0, column=1, padx=10)
        tk.Radiobutton(self.frame, text="Rayons Aleatoires", variable=self.
            mode_var, value="aleatoire", font=("Arial", 12)).grid(row=0, column
            =2, padx=10)

        # Boutons pour les actions
        self.segment_button = tk.Button(self.frame, text="Segmentation", command
            =self.segment_image,  fg="black", font=("Arial", 12))
        self.segment_button.grid(row=0, column=3, padx=10, pady=5)

        self.extract_button = tk.Button(self.frame, text="Extraction", command=
            self.extract_signature,  fg="black", font=("Arial", 12))
        self.extract_button.grid(row=0, column=4, padx=10, pady=5)

        self.decode_button = tk.Button(self.frame, text="Decoder", command=self.
            decode_barcode,  fg="black", font=("Arial", 12))
        self.decode_button.grid(row=0, column=5, padx=10, pady=5)

        self.verify_button = tk.Button(self.frame, text="Verifier dans la base",
             command=self.verify_database,  fg="black", font=("Arial", 12))
        self.verify_button.grid(row=0, column=6, padx=10, pady=5)

        # Bouton de reinitialisation
        self.reset_button = tk.Button(self.frame, text="Reinitialiser", command=
            self.reset_app,  fg="black", font=("Arial", 12))
        self.reset_button.grid(row=0, column=7, padx=10, pady=5)

        # Zone d'affichage d'image
        self.image_label = tk.Label(self.canvas, bg="#ffffff", bd=2, relief="
            sunken")
        self.image_label.place(relx=0.5, rely=0.6, anchor="center", width=800,
            height=500)
```

```
        # Zone de feedback
        self.feedback = tk.Label(self, text="Pret", font=("Arial", 12), bg="#
            e9ecef", fg="black")
        self.feedback.place(relx=0.5, rely=0.95, anchor="s")

        # Ajouter un label en bas a droite
        footer_label = tk.Label(self, text="Projet Image TS225 2024/2025", font
            =("Arial", 14), fg="white")
        footer_label.place(relx=1.0, rely=1.0, anchor="se", x=0, y=0)  #
            Decalage vers l'interieur

        footer_label2 = tk.Label(self, text="Developpe par: Mehdi, Salma, Arif
            et Louriz",
                                 font=("Arial", 14), fg="white")
        footer_label2.place(relx=0.0, rely=1.0, anchor="sw", x=0, y=0)  #
            Decalage leger

        # Bouton pour quitter l'application
        self.quit_button = tk.Button(
            self,
            text="Quitter",
            command=self.quit_app,
            fg="red",           # Couleur du texte
            bg="red",             # Couleur de fond

            font=("Arial", 12, "bold"), # Police
            relief="raised",      # Effet 3D
            bd=2,                 # Bordure
            padx=10,              # Padding horizontal
            pady=5                # Padding vertical
        )
        self.quit_button.place(relx=0.5, rely=0.975, anchor="center")  #
            Centrage en bas

    def load_image(self):
        """Charger une image depuis le fichier."""
        try:
            file_path = filedialog.askopenfilename(
                title="Selectionner une image",
                filetypes=[("Images", "*.png;*.jpg;*.jpeg")]
            )
            if not file_path:
                self.feedback.config(text="Aucune image selectionnee.")
                return

            self.image_path = file_path
            self.image = Image.open(file_path)
            self.display_image(self.image)
            self.feedback.config(text="Image chargee avec succes.")
        except Exception as e:
            messagebox.showerror("Erreur", f"Impossible de charger l'image : {
                str(e)}")
            self.feedback.config(text="Erreur lors du chargement.")

    def display_image(self, img):
        """Afficher une image redimensionnee."""
        img = img.resize((800, 500))
        img = ImageTk.PhotoImage(img)
        self.image_label.config(image=img)
        self.image_label.image = img
```

```
def segment_image(self):
    """Segmentation reelle avec extraction depuis le fichier segmentation.
       """
    try:
        # Mise a jour du feedback pour informer l'utilisateur
        self.feedback.config(text="Segmentation en cours...")
        self.update_idletasks()

        # Verifier si une image est chargee
        if not self.image_path:
            messagebox.showerror("Erreur", "Aucune image chargee !")
            self.feedback.config(text="Erreur : Chargez une image.")
            return

        # Appel de la fonction de segmentation
        mask, region = segmentation(self.image_path)

        # Affichage du resultat
        plt.figure(figsize=(8, 4))
        plt.subplot(1, 2, 1)
        plt.imshow(region, cmap='gray')
        plt.title("Region detectee")
        plt.axis("off")

        plt.subplot(1, 2, 2)
        plt.imshow(mask, cmap='gray')
        plt.title("Masque final")
        plt.axis("off")

        plt.tight_layout()
        plt.show()

        # Mise a jour du feedback
        self.feedback.config(text="Segmentation terminee.")
    except Exception as e:
        # Gestion des erreurs avec message d'alerte
        messagebox.showerror("Erreur", f"Erreur lors de la segmentation : {
            str(e)}")
        self.feedback.config(text="Erreur lors de la segmentation.")


def extract_signature(self):
    """Extraction des signatures avec un rayon manuel ou aleatoire."""
    try:
        # Feedback initial
        self.feedback.config(text="Extraction en cours...")
        self.update_idletasks()

        # Verifier si une image est chargee
        if not self.image_path:
            messagebox.showerror("Erreur", "Aucune image chargee !")
            self.feedback.config(text="Erreur : Chargez une image.")
            return

        # Verifier si la segmentation a ete realisee (avec des coins
            detectes)
        if not hasattr(self, 'detected_region') or self.detected_region is
            None:
            messagebox.showerror("Erreur", "Aucune region detectee. Lancez
                la segmentation d'abord.")
```

```
                self.feedback.config(text="Erreur : Pas de region detectee.")
                return

            # Recuperer les coins detectes pour la zone d'interet
            C1, C2, C3, C4 = self.detected_region  # Coins detectes apres
                segmentation

            # Verifier le mode choisi (manuel ou aleatoire)
            if self.mode_var.get() == "manuel":
                messagebox.showinfo("Instruction", "Cliquez sur deux points pour
                    definir un rayon.")
                self.feedback.config(text="Attente de la selection manuelle...")
                self.points = []  # Reinitialiser les points pour la selection
                    manuelle
                self.canvas.bind("<Button-1>", self.on_click_manual)
                return

            elif self.mode_var.get() == "aleatoire":
                # Generer un rayon aleatoire avec la fonction lancer_aleatoire
                p1, p2 = lancer_aleatoire(C1, C2, C3, C4)
                points = (p1, p2)

            # Appeler la fonction d'extraction pour obtenir la signature binaire
            self.binary_signature = extraction(self.image_path, points[0],
                points[1])

            # Verifier si l'extraction a reussi
            if self.binary_signature is None:
                raise ValueError("Signature non extraite ou invalide.")

            # Afficher la signature binaire extraite
            plt.figure()
            plt.step(range(len(self.binary_signature)), self.binary_signature,
                where='mid')
            plt.title("Signature binaire extraite")
            plt.xlabel("Position")
            plt.ylabel("Valeur (0 ou 1)")
            plt.grid(True)
            plt.show()

            # Mise a jour du feedback
            self.feedback.config(text="Extraction terminee avec succes.")

        except Exception as e:
            # Gestion des erreurs
            messagebox.showerror("Erreur", f"Erreur lors de l'extraction : {str(
                e)}")
            self.feedback.config(text="Erreur lors de l'extraction.")


    def decode_barcode(self):
        """Decodage en utilisant la fonction du fichier fonctions.py."""
        try:
            # Feedback pour l'utilisateur
            self.feedback.config(text="Decodage en cours...")
            self.update_idletasks()

            # Verifier si une signature a ete extraite
            if self.binary_signature is None:
                messagebox.showerror("Erreur", "Aucune signature disponible pour
                    le decodage.")
                self.feedback.config(text="Erreur : Aucune signature detectee.")
```

```
                return

            # Appeler la fonction de decodage
            self.decoded_barcode = decode_ean13_signature(self.binary_signature)

            # Mise a jour du feedback avec le code-barres detecte
            self.feedback.config(text=f"Code-barres detecte : {self.
                decoded_barcode}")
            messagebox.showinfo("Decodage Reussi", f"Code-barres : {self.
                decoded_barcode}")

        except Exception as e:
            # Gestion des erreurs
            messagebox.showerror("Erreur", f"Erreur lors du decodage : {str(e)}"
                )
            self.feedback.config(text="Erreur lors du decodage.")

    def verify_database(self):
        """Verifier dans la base de donnees."""
        try:
            self.feedback.config(text="Verification dans la base...")
            self.update_idletasks()

            # Verifier si un code-barres a ete decode
            if not self.decoded_barcode:
                messagebox.showwarning("Attention", "Aucun code-barres decode.
                    Veuillez lancer le decodage d'abord.")
                self.feedback.config(text="Erreur : Pas de code-barres decode.")
                return

            # Charger la base de donnees (fichier texte contenant une liste de
                codes-barres valides)
            database_path = filedialog.askopenfilename(
                title="Charger la base de donnees",
                filetypes=[("Fichiers texte", "*.txt")]
            )

            if not database_path:
                self.feedback.config(text="Erreur : Aucune base de donnees
                    selectionnee.")
                return

            # Lire la base de donnees
            with open(database_path, 'r') as file:
                database = [line.strip() for line in file.readlines()]

            # Verifier si le code-barres est dans la base
            if self.decoded_barcode in database:
                messagebox.showinfo("Resultat", f"Produit trouve : {self.
                    decoded_barcode}")
                self.feedback.config(text="Produit trouve dans la base.")
            else:
                messagebox.showwarning("Resultat", f"Produit non trouve : {self.
                    decoded_barcode}")
                self.feedback.config(text="Produit non trouve dans la base.")

        except Exception as e:
            # Gestion des erreurs
            messagebox.showerror("Erreur", f"Erreur lors de la verification : {
                str(e)}")
            self.feedback.config(text="Erreur lors de la verification.")
```

```
    def reset_app(self):
        """Reinitialiser l'application."""
        self.image = None
        self.image_path = ""
        self.binary_signature = None
        self.decoded_barcode = None
        self.image_label.config(image="")
        self.feedback.config(text="Reinitialise.")

    def quit_app(self):
        """Quitter l'application."""
        self.feedback.config(text="Fermeture de l'application...")
        self.update_idletasks()
        self.destroy()  # Ferme la fenetre principale


if __name__ == "__main__":
    app = BarcodeApp()
    app.mainloop()
```

Listing 3 – app.py

7.4 Anciennes versions des codes avant regroupement dans fonctions.py

Cette section présente les versions initiales des différents modules avant leur intégration dans un fichier centralisé (fonctions.py). Ces codes avaient été développés séparément pour faciliter les tests et les modifications avant d'être unifiés. Les tests réalisés sur chaque module séparé ont permis d'identifier et de corriger les erreurs. Cela a facilité la validation progressive des fonctionnalités.

7.4.1 code de la segmentation

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 13 11:25:26 2024

code segmentation

@author: Mehdi & Arif
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import convolve2d
from skimage import color, io
from skimage.morphology import closing, opening, square
from skimage.measure import label, regionprops

################################################################################
#                                PARAMETRES                                    #
################################################################################

# Fichier image
image_path = 'images/lait.png'

# Parametres bruit
sigma_noise = 0.02  # ecart-type du bruit gaussien

# Parametres du filtre gaussien
```

```
sigma_G = 1.8  # ecart-type pour le calcul du gradient gaussien

# Parametres du tenseur de structure
sigma_T = 18   # ecart-type pour le lissage dans le tenseur

# Parametres de segmentation
seuil_coherence = 0.3  # Seuil pour la coherence

##############################################################################
#                              CHARGEMENT IMAGE                              #
##############################################################################

img = io.imread(image_path)

# Supprimer le canal alpha si present
if img.shape[-1] == 4:
    img = img[..., :3]

# Conversion en niveaux de gris
I = color.rgb2gray(img)



##############################################################################
#                            AJOUT DE BRUIT                                  #
##############################################################################

bruit = np.random.normal(0, sigma_noise, I.shape)
I_bruite = np.clip(I + bruit, 0, 1)

##############################################################################
#                      CALCUL DES GRADIENTS GAUSSIENS                        #
##############################################################################

# Generer les filtres derives de la gaussienne
size = int(3 * sigma_G)
x, y = np.meshgrid(np.arange(-size, size+1), np.arange(-size, size+1))

G_x = -(x / (2 * np.pi * sigma_G**4)) * np.exp(-(x**2 + y**2) / (2*sigma_G**2))
G_y = -(y / (2 * np.pi * sigma_G**4)) * np.exp(-(x**2 + y**2) / (2*sigma_G**2))

# Appliquer les filtres pour calculer les gradients
I_x = convolve2d(I_bruite, G_x, mode='same', boundary='symm')
I_y = convolve2d(I_bruite, G_y, mode='same', boundary='symm')

# Normaliser les gradients
norme = np.sqrt(I_x**2 + I_y**2) + 1e-8
I_x /= norme
I_y /= norme

##############################################################################
#                              TENSEUR DE STRUCTURE                          #
##############################################################################

# Calcul du filtre gaussien pour le lissage
size_T = int(2 * sigma_T)
x_T, y_T = np.meshgrid(np.arange(-size_T, size_T+1), np.arange(-size_T, size_T
    +1))
G = (1 / (2 * np.pi * sigma_T**2)) * np.exp(-(x_T**2 + y_T**2)/(2*sigma_T**2))

# Calcul des composantes du tenseur
T_xx = convolve2d(I_x**2, G, mode='same', boundary='symm')
```

```
T_xy = convolve2d(I_x * I_y, G, mode='same', boundary='symm')
T_yy = convolve2d(I_y**2, G, mode='same', boundary='symm')

################################################################################
#                        MESURE DE COHERENCE ET SEGMENTATION                   #
################################################################################

# Calcul de la coherence D
D1 = 1 - np.sqrt((T_xx - T_yy)**2 + 4*(T_xy**2)) / (T_xx + T_yy + 1e-15)

# Segmentation par seuil
M = (D1 > seuil_coherence).astype(int)

################################################################################
#                        NETTOYAGE MORPHOLOGIQUE                               #
################################################################################

# Nettoyage avec ouverture et fermeture morphologiques
M_clean = closing(M, square(3))  # Connexion des barres
M_clean = opening(M_clean, square(2))  # Suppression du bruit



################################################################################
#           EXTRACTION DE LA PLUS GRANDE REGION CONNEXE (CODE-BARRE)           #
################################################################################

labels = label(M_clean)

if labels.max() > 0:
    # Identifier la plus grande region
    regions = regionprops(labels)
    largest_region = max(regions, key=lambda r: r.area)

    # Creer le masque final base sur cette region
    M_final = np.zeros_like(M_clean)
    M_final[labels == largest_region.label] = 1
else:
    M_final = np.zeros_like(M_clean)

################################################################################
#          ANALYSE GEOMETRIQUE DE LA REGION IDENTIFIEE                         #
################################################################################

Y, X = np.mgrid[0:M.shape[0], 0:M.shape[1]]

if np.sum(M) > 0:
    # Calcul du barycentre
    O_x = np.sum(X * M) / np.sum(M)
    O_y = np.sum(Y * M) / np.sum(M)

    # Centrage des points
    X_c = X - O_x
    Y_c = Y - O_y

    # Matrice de covariance
    C_xx = np.sum(X_c * X_c * M) / np.sum(M)
    C_xy = np.sum(X_c * Y_c * M) / np.sum(M)
    C_yy = np.sum(Y_c * Y_c * M) / np.sum(M)

    C = np.array([[C_xx, C_xy], [C_xy, C_yy]])
```

```
    # Calcul des vecteurs propres
    vals, vecs = np.linalg.eig(C)
    idx = np.argsort(vals)[::-1]
    vecs = vecs[:, idx]

    # Calcul des coins du rectangle oriente
    u1, u2 = vecs[:, 0], vecs[:, 1]
    alpha = X_c * u1[0] + Y_c * u1[1]
    beta = X_c * u2[0] + Y_c * u2[1]

    alpha_min, alpha_max = np.min(alpha[M==0]), np.max(alpha[M==0])
    beta_min, beta_max = np.min(beta[M==0]), np.max(beta[M==0])

    def corner(a, b):
        return (O_x + a * u1[0] + b * u2[0], O_y + a * u1[1] + b * u2[1])

    C1 = corner(alpha_min, beta_min)
    C2 = corner(alpha_max, beta_min)
    C3 = corner(alpha_max, beta_max)
    C4 = corner(alpha_min, beta_max)
else:
    C1 = C2 = C3 = C4 = (0, 0)



################################################################################
#                                  AFFICHAGE                                   #
################################################################################

plt.figure(figsize=(15, 5))

# Image originale
plt.subplot(1, 2, 1)
plt.imshow(img, cmap='gray')
plt.title('Image originale')
plt.axis('off')

# Masque final
plt.subplot(1, 2, 2)
plt.imshow(M_final, cmap='gray')
plt.title('Code-barres detecte')
plt.axis('off')

# Rectangle oriente
#plt.subplot(1, 3, 3)
#plt.imshow(M_final, cmap='gray')

# Tracer le rectangle rouge
rect_x = [C1[0], C2[0], C3[0], C4[0], C1[0]]
rect_y = [C1[1], C2[1], C3[1], C4[1], C1[1]]
#plt.plot(rect_x, rect_y, 'r-', linewidth=2)

# Ajout des droites horizontales (2 lignes)
for _ in range(2):
    # Generer une position verticale (y) aleatoire entre les bords verticaux
    t = np.random.uniform(0, 1)  # Coefficient pour interpoler sur l'horizontale

    # Calculer les points de depart et d'arrivee sur l'horizontale
    x_start = C1[0] * (1 - t) + C4[0] * t  # Interpolation sur la gauche (C1 ->
        C4)
    y_start = C1[1] * (1 - t) + C4[1] * t
```

```
            x_end = C2[0] * (1 - t) + C3[0] * t    # Interpolation sur la droite (C2 ->
                C3)
            y_end = C2[1] * (1 - t) + C3[1] * t

            # Tracer la ligne horizontale bleue
           # plt.plot([x_start, x_end], [y_start, y_end], 'b--', linewidth=1.5)

# Ajout des droites selon les diagonales (2 lignes)
# Diagonale 1 (C1 -> C3)
#plt.plot([C1[0], C3[0]], [C1[1], C3[1]], 'g--', linewidth=1.5)

# Diagonale 2 (C2 -> C4)
#plt.plot([C2[0], C4[0]], [C2[1], C4[1]], 'g--', linewidth=1.5)

# Parametres de l'affichage
#plt.title('Lancers aleatoirs des rayons')
#plt.axis('off')
```

Listing 4 – segmentation.py

7.4.2 code des lancers aléatoires des rayons

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 13 10:25:11 2024

code lancer aleatoire

@author: Arif
"""
import numpy as np
import matplotlib.pyplot as plt
from skimage import color, io
from skimage.morphology import closing, opening, square
from skimage.measure import label, regionprops
import cv2
from scipy.signal import convolve2d

def point_aleatoire_segment(P1, P2):
    t = np.random.random()
    return (P1[0] + t * (P2[0] - P1[0]), P1[1] + t * (P2[1] - P1[1]))

def generer_rayon_dans_rectangle(C1, C2, C3, C4):
    direction = np.array([C2[0] - C1[0], C2[1] - C1[1]])
    direction = direction / np.linalg.norm(direction)
    angle_max = np.pi/6  # 30 degrs

    if np.random.random() < 0.5:
        p1 = point_aleatoire_segment(C1, C4)
        p2 = point_aleatoire_segment(C2, C3)
    else:
        p1 = point_aleatoire_segment(C2, C3)
        p2 = point_aleatoire_segment(C1, C4)

    return p1, p2

def lancer_rayons(image_path, nb_rayons=100):
    # Chargement image
    img = io.imread(image_path)
    if img.shape[-1] == 4:
        img = img[..., :3]
    I = color.rgb2gray(img)

    # Calcul des gradients
    sigma_G = 1.8
    size = int(3 * sigma_G)
    x, y = np.meshgrid(np.arange(-size, size+1), np.arange(-size, size+1))

    G_x = -(x / (2 * np.pi * sigma_G**4)) * np.exp(-(x**2 + y**2) / (2*sigma_G
        **2))
    G_y = -(y / (2 * np.pi * sigma_G**4)) * np.exp(-(x**2 + y**2) / (2*sigma_G
        **2))

    I_x = convolve2d(I, G_x, mode='same', boundary='symm')
    I_y = convolve2d(I, G_y, mode='same', boundary='symm')

    norme = np.sqrt(I_x**2 + I_y**2) + 1e-8
    I_x /= norme
    I_y /= norme

    # Tenseur de structure
```

```
sigma_T = 19
size_T = int(2 * sigma_T)
x_T, y_T = np.meshgrid(np.arange(-size_T, size_T+1), np.arange(-size_T,
    size_T+1))
G = (1 / (2 * np.pi * sigma_T**2)) * np.exp(-(x_T**2 + y_T**2)/(2*sigma_T
    **2))

T_xx = convolve2d(I_x**2, G, mode='same', boundary='symm')
T_xy = convolve2d(I_x*I_y, G, mode='same', boundary='symm')
T_yy = convolve2d(I_y**2, G, mode='same', boundary='symm')

# Mesure de coh rence
D1 = 1 - np.sqrt((T_xx - T_yy)**2 + 4*(T_xy**2)) / (T_xx + T_yy + 1e-15)

# Segmentation et nettoyage
M = (D1 > 0.29).astype(int)
M_clean = closing(M, square(3))
M_clean = opening(M_clean, square(2))

# Extraction composante principale
labels = label(M_clean)
if labels.max() > 0:
    regions = regionprops(labels)
    largest_region = max(regions, key=lambda r: r.area)
    M_final = np.zeros_like(M_clean)
    M_final[labels == largest_region.label] = 1
else:
    M_final = np.zeros_like(M_clean)

# Calcul rectangle orient
Y, X = np.mgrid[0:M.shape[0], 0:M.shape[1]]

if np.sum(M_final) > 0:
    O_x = np.sum(X * M_final) / np.sum(M_final)
    O_y = np.sum(Y * M_final) / np.sum(M_final)

    X_c = X - O_x
    Y_c = Y - O_y

    Gx = convolve2d(M_final, [[-1, 0, 1]], mode='same')
    Gy = convolve2d(M_final, [[-1], [0], [1]], mode='same')
    poids = np.sqrt(Gx**2 + Gy**2)

    C_xx = np.sum(X_c * X_c * poids) / np.sum(poids)
    C_xy = np.sum(X_c * Y_c * poids) / np.sum(poids)
    C_yy = np.sum(Y_c * Y_c * poids) / np.sum(poids)
    C = np.array([[C_xx, C_xy], [C_xy, C_yy]])

    vals, vecs = np.linalg.eig(C)
    idx = np.argsort(vals)[::-1]
    vals = vals[idx]
    vecs = vecs[:, idx]
    u1 = vecs[:, 0]
    u2 = vecs[:, 1]

    alpha = X_c*u1[0] + Y_c*u1[1]
    beta = X_c*u2[0] + Y_c*u2[1]

    alpha_min = np.min(alpha[M_final==0])
    alpha_max = np.max(alpha[M_final==0])
    beta_min = np.min(beta[M_final==0])
    beta_max = np.max(beta[M_final==0])
```

```
        def corner(a, b):
            return (O_x + a*u1[0] + b*u2[0],
                    O_y + a*u1[1] + b*u2[1])

        C1 = corner(alpha_min, beta_min)
        C2 = corner(alpha_max, beta_min)
        C3 = corner(alpha_max, beta_max)
        C4 = corner(alpha_min, beta_max)
    else:
        return

    # G n ration des rayons
    rayons = []
    for _ in range(nb_rayons):
        p1, p2 = generer_rayon_dans_rectangle(C1, C2, C3, C4)
        rayons.append((p1, p2))

    # Visualisation
    plt.figure(figsize=(15,5))

    # Image originale avec rayons
    plt.subplot(1,3,1)
    plt.imshow(img)
    plt.title('Image originale')
    for p1, p2 in rayons:
        plt.plot([p1[0], p2[0]], [p1[1], p2[1]], 'y-', alpha=0.2)
    plt.axis('off')

    # R gion finale
    plt.subplot(1,3,2)
    plt.imshow(M_final, cmap='gray')
    plt.title('R gion finale (code-barres)')
    plt.axis('off')

    # Rectangle englobant
    plt.subplot(1,3,3)
    plt.imshow(M_final, cmap='gray')
    plt.title('Rectangle englobant orient  sur le masque')
    rect_x = [C1[0], C2[0], C3[0], C4[0], C1[0]]
    rect_y = [C1[1], C2[1], C3[1], C4[1], C1[1]]
    plt.plot(rect_x, rect_y, 'r-', linewidth=2)

    for p1, p2 in rayons:
        plt.plot([p1[0], p2[0]], [p1[1], p2[1]], 'y-', alpha=0.2)
    plt.axis('off')

    plt.tight_layout()
    plt.show()

# Example usage
if __name__ == "__main__":
    lancer_rayons("image.jpg", nb_rayons=100)
```

Listing 5 – lancers$_a$*leatoires.py*

7.4.3 code de l'extraction

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 3 11:25:26 2024

code extraction

@author: Mehdi & Arif
"""
```

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
import os
import tkinter as tk
from tkinter import messagebox
from PIL import Image, ImageTk
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

def cv2_to_imageTk(cv2_image):
    if len(cv2_image.shape) == 2:  # grayscale
        cv2_image_rgb = cv2.cvtColor(cv2_image, cv2.COLOR_GRAY2RGB)
    else:
        cv2_image_rgb = cv2.cvtColor(cv2_image, cv2.COLOR_BGR2RGB)
    pil_image = Image.fromarray(cv2_image_rgb)
    image_tk = ImageTk.PhotoImage(image=pil_image)
    return image_tk

class ImageApp(tk.Tk):
    def __init__(self, image_path):
        super().__init__()
        self.title("Application de Traitement d'Image")
        self.points = []
        self.image_path = image_path
        self.load_image()
        self.setup_gui()

    def load_image(self):
        if not os.path.exists(self.image_path):
            messagebox.showerror("Erreur", f"L'image {self.image_path} n'existe
                pas!")
            self.destroy()
        else:
            self.cv_image = cv2.imread(self.image_path, cv2.IMREAD_GRAYSCALE)
            if self.cv_image is None:
                messagebox.showerror("Erreur", "Impossible de charger l'image!")
                self.destroy()
            else:
                self.cv_image_color = cv2.cvtColor(self.cv_image, cv2.
                    COLOR_GRAY2BGR)
                self.photo_image = cv2_to_imageTk(self.cv_image_color)

    def setup_gui(self):
        self.canvas = tk.Canvas(self, width=self.cv_image_color.shape[1], height
            =self.cv_image_color.shape[0])
        self.canvas.pack()
```

```
        self.canvas_image = self.canvas.create_image(0, 0, anchor=tk.NW, image=
            self.photo_image)
        self.canvas.bind("<Button-1>", self.on_click)
        self.status_label = tk.Label(self, text="Selectionnez deux points sur l'
            image pour definir le rayon")
        self.status_label.pack()
        self.protocol("WM_DELETE_WINDOW", self.on_closing)

    def on_click(self, event):
        x, y = event.x, event.y
        self.points.append((x, y))
        cv2.circle(self.cv_image_color, (x, y), 3, (0, 0, 255), -1)
        self.photo_image = cv2_to_imageTk(self.cv_image_color)
        self.canvas.itemconfig(self.canvas_image, image=self.photo_image)
        if len(self.points) == 1:
            self.status_label.config(text="Premier point selectionne.
                Selectionnez le second point.")
        elif len(self.points) == 2:
            self.status_label.config(text="Deux points selectionnes. Traitement
                en cours...")
            self.canvas.unbind("<Button-1>")
            self.after(100, self.process_image)

    def process_image(self):
        p1, p2 = self.points
        # 1. Calculer la longueur du rayon
        longueur_rayon = np.sqrt((p2[0] - p1[0])**2 + (p2[1] - p1[1])**2)
        print(f"Longueur du rayon: {longueur_rayon:.2f} pixels")

        # 2. Extraire la premi re signature
        nb_points = max(int(longueur_rayon), 95)
        t = np.linspace(0, 1, nb_points)
        x = np.array([p1[0] + (p2[0] - p1[0])*t_i for t_i in t], dtype=np.
            float32)
        y = np.array([p1[1] + (p2[1] - p1[1])*t_i for t_i in t], dtype=np.
            float32)
        signature = cv2.remap(self.cv_image, x.reshape(-1,1), y.reshape(-1,1),
            cv2.INTER_LINEAR)[:, 0]

        # 3. Calculer et appliquer le seuil d'Otsu
        hist, bins = np.histogram(signature, bins=256, range=(0, 256))
        total_pixels = signature.size
        sum_total = sum(i * h for i, h in enumerate(hist))

        max_criteria = -1
        threshold = 0
        criteria_values = []  # Pour stocker les valeurs du crit re

        for k in range(256):
            w_b = sum(hist[:k])
            w_f = total_pixels - w_b

            if w_b == 0 or w_f == 0:
                criteria_values.append(0)
                continue

            sum_b = sum(i * hist[i] for i in range(k))
            m_b = sum_b / w_b if w_b != 0 else 0
            m_f = (sum_total - sum_b) / w_f if w_f != 0 else 0

            criteria = w_b * w_f * (m_b - m_f) ** 2
            criteria_values.append(criteria)
```

```
            if criteria > max_criteria:
                max_criteria = criteria
                threshold = k

        print(f"Seuil d'Otsu calcule : {threshold}")

        # 4. Binariser la premi re signature
        binary_signature = (signature > threshold).astype(np.uint8)

        # 5. Trouver les limites utiles
        non_zero = np.nonzero(binary_signature)[0]
        if len(non_zero) > 0:
            start_idx = non_zero[0]
            end_idx = non_zero[-1]
            print(f"Limites utiles : {start_idx}    {end_idx}")

            # 6. Calculer les coordonnees des points du rayon utile
            t_start = start_idx / nb_points
            t_end = end_idx / nb_points

            useful_p1 = (
                p1[0] + (p2[0] - p1[0]) * t_start,
                p1[1] + (p2[1] - p1[1]) * t_start
            )
            useful_p2 = (
                p1[0] + (p2[0] - p1[0]) * t_end,
                p1[1] + (p2[1] - p1[1]) * t_end
            )

            # 7. Calculer l'unite de base u
            useful_length = np.sqrt((useful_p2[0] - useful_p1[0])**2 +
                                    (useful_p2[1] - useful_p1[1])**2)
            u = max(1, int(useful_length / 95))
            print(f"Unite de base u calculee : {u}")

            # 8. Extraire la nouvelle signature le long du rayon utile
            nb_points_final = 95 * u
            t = np.linspace(0, 1, nb_points_final)
            x = np.array([useful_p1[0] + (useful_p2[0] - useful_p1[0])*t_i for
                t_i in t], dtype=np.float32)
            y = np.array([useful_p1[1] + (useful_p2[1] - useful_p1[1])*t_i for
                t_i in t], dtype=np.float32)
            final_signature = cv2.remap(self.cv_image, x.reshape(-1,1), y.
                reshape(-1,1), cv2.INTER_LINEAR)[:, 0]

            # 9. Binariser la signature finale
            final_binary_signature = (final_signature > threshold).astype(np.
                uint8)

            # Affichage des resultats
            self.display_plots(signature, binary_signature, final_signature,
                final_binary_signature, nb_points_final, criteria_values)
            len_sin=len(final_binary_signature)
            print(f"Taille signature : {len_sin}")
            print(final_binary_signature)

            # Afficher l'image avec les rayons
            cv2.line(self.cv_image_color, p1, p2, (0, 255, 0), 1)  # Rayon
                initial en vert
            cv2.line(self.cv_image_color,
                     (int(useful_p1[0]), int(useful_p1[1])),
```

```
                    (int(useful_p2[0]), int(useful_p2[1])),
                    (255, 0, 0), 2)  # Rayon utile en bleu

            self.photo_image = cv2_to_imageTk(self.cv_image_color)
            self.canvas.itemconfig(self.canvas_image, image=self.photo_image)
            self.status_label.config(text="Traitement termine.")
        else:
            print("Aucune region utile trouvee dans la signature")
            messagebox.showinfo("Information", "Aucune region utile trouvee dans
                la signature")
            self.status_label.config(text="Aucune region utile trouvee dans la
                signature.")

    def display_plots(self, signature, binary_signature, final_signature,
        final_binary_signature, nb_points_final, criteria_values):
        plot_window = tk.Toplevel(self)
        plot_window.title("Resultats")
        fig = plt.Figure(figsize=(8, 12))  # Ajuster la taille pour 5 subplots
        ax1 = fig.add_subplot(511)
        ax1.plot(signature)
        ax1.set_title('Premi re signature')
        ax1.grid(True)

        ax2 = fig.add_subplot(512)
        ax2.step(range(len(binary_signature)), binary_signature, where='mid')
        ax2.set_title('Premi re signature binarisee')
        ax2.grid(True)

        ax3 = fig.add_subplot(513)
        ax3.plot(final_signature)
        ax3.set_title(f'Signature finale ({nb_points_final} points)')
        ax3.grid(True)

        ax4 = fig.add_subplot(514)
        ax4.step(range(len(final_binary_signature)), final_binary_signature,
            where='mid')
        ax4.set_title('Signature finale binarisee')
        ax4.grid(True)

        ax5 = fig.add_subplot(515)
        ax5.plot(range(256), criteria_values)
        ax5.set_title('Crit re d\'Otsu en fonction du seuil')
        ax5.set_xlabel('Seuil')
        ax5.set_ylabel('Crit re')
        ax5.grid(True)

        fig.tight_layout()
        canvas = FigureCanvasTkAgg(fig, master=plot_window)
        canvas.draw()
        canvas.get_tk_widget().pack()

    def on_closing(self):
        plt.close('all')
        self.destroy()

if __name__ == "__main__":
    chemin_image = 'images/barpreview.png'
    app = ImageApp(chemin_image)
    app.mainloop()
```

Listing 6 – extraction.py

7.4.4 code du décodage

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Fri Dec 7 13:05:36 2024

code decodage

@author: Mehdi
"""

def decode_ean13_signature(binary_signature):
    """
    Decoder un code-barres EAN-13 a partir de sa signature binaire.

    Parametres:
    - binary_signature: liste d'entiers (0 ou 1), representant la signature
        binaire du code-barres (95 bits)

    Retourne:
    - code_barres: cha ne de caracteres representant le code EAN-13 decode
    """
    # Verifier la longueur de la signature
    if len(binary_signature) != 95:
        raise ValueError("La signature binaire doit contenir exactement 95 bits.
            ")

    # Motifs de garde
    guard_left = [1, 0, 1]
    guard_center = [0, 1, 0, 1, 0]
    guard_right = [1, 0, 1]

    # Verifier les motifs de garde
    if binary_signature[0:3] != guard_left:
        raise ValueError("Motif de garde gauche incorrect.")
    if binary_signature[45:50] != guard_center:
        raise ValueError("Motif de garde central incorrect.")
    if binary_signature[92:95] != guard_right:
        raise ValueError("Motif de garde droit incorrect.")

    # Tables de codage pour les chiffres
    code_L = {
        '0001101': '0',
        '0011001': '1',
        '0010011': '2',
        '0111101': '3',
        '0100011': '4',
        '0110001': '5',
        '0101111': '6',
        '0111011': '7',
        '0110111': '8',
        '0001011': '9',
    }

    code_G = {
        '0100111': '0',
        '0110011': '1',
        '0011011': '2',
        '0100001': '3',
        '0011101': '4',
        '0111001': '5',
```

```
    '0000101': '6',
    '0010001': '7',
    '0001001': '8',
    '0010111': '9',
}

code_R = {
    '1110010': '0',
    '1100110': '1',
    '1101100': '2',
    '1000010': '3',
    '1011100': '4',
    '1001110': '5',
    '1010000': '6',
    '1000100': '7',
    '1001000': '8',
    '1110100': '9',
}

# Table de parite pour determiner le premier chiffre
parity_table = {
    'LLLLLL': '0',
    'LLGLGG': '1',
    'LLGGLG': '2',
    'LLGGGL': '3',
    'LGLLGG': '4',
    'LGGLLG': '5',
    'LGGGLL': '6',
    'LGLGLG': '7',
    'LGLGGL': '8',
    'LGGLGL': '9',
}

# Decodage des 6 chiffres de gauche
left_digits = []
parity_pattern = ''
for i in range(6):
    start = 3 + i * 7
    end = start + 7
    pattern_bits = binary_signature[start:end]
    pattern = ''.join(map(str, pattern_bits))

    if pattern in code_L:
        digit = code_L[pattern]
        parity_pattern += 'L'
    elif pattern in code_G:
        digit = code_G[pattern]
        parity_pattern += 'G'
    else:
        raise ValueError(f"Motif inconnu dans la partie gauche : {pattern}")
    left_digits.append(digit)

# Determiner le premier chiffre a partir du motif de parite
if parity_pattern in parity_table:
    first_digit = parity_table[parity_pattern]
else:
    raise ValueError(f"Motif de parite inconnu : {parity_pattern}")

# Decodage des 6 chiffres de droite
right_digits = []
for i in range(6):
    start = 50 + i * 7
```

```
            end = start + 7
            pattern_bits = binary_signature[start:end]
            pattern = ''.join(map(str, pattern_bits))

            if pattern in code_R:
                digit = code_R[pattern]
            else:
                raise ValueError(f"Motif inconnu dans la partie droite : {pattern}")
            right_digits.append(digit)

        # Construire le code-barres complet
        code_barres = first_digit + ''.join(left_digits) + ''.join(right_digits)

        # Verifier la cle de controle
        digits = list(map(int, code_barres))
        if len(digits) != 13:
            raise ValueError("Le code-barres doit contenir 13 chiffres.")

        # Calcul de la cle de controle selon la norme EAN-13
        # etape 1 : Somme des chiffres en positions paires (2e, 4e, ..., 12e)
        sum_even = sum(digits[i] for i in range(1, 12, 2))

        # etape 2 : Multiplier la somme par 3
        sum_even *= 3

        # etape 3 : Somme des chiffres en positions impaires (1ere, 3e, ..., 11e)
        sum_odd = sum(digits[i] for i in range(0, 12, 2))

        # etape 4 : Calculer le total
        total = sum_even + sum_odd

        # etape 5 : Calculer la cle de controle
        check_digit = (10 - (total % 10)) % 10

        # Verifier si la cle de controle est correcte
        if check_digit != digits[-1]:
            raise ValueError(f"Cle de controle invalide : attendu {check_digit},
                obtenu {digits[-1]}")

        return code_barres

    # Exemple d'utilisation
    if __name__ == "__main__":

        # Exemple de signature binaire pour le code EAN-13 "4006381333931"
        binary_signature_str = (
            '101'        # Garde gauche
            '0001101'    # Chiffre 0, encodage L
            '0100111'    # Chiffre 0, encodage G
            '0101111'    # Chiffre 6, encodage L
            '0111101'    # Chiffre 3, encodage L
            '0001001'    # Chiffre 8, encodage G
            '0110011'    # Chiffre 1, encodage G
            '01010'      # Garde centrale
            '1000010'    # Chiffre 3, encodage R
            '1000010'    # Chiffre 3, encodage R
            '1000010'    # Chiffre 3, encodage R
            '1110100'    # Chiffre 9, encodage R
            '1000010'    # Chiffre 3, encodage R
            '1100110'    # Chiffre 1, encodage R
            '101'        # Garde droite
        )
```

```
print("Code EAN-13 a decoder : 4006381333931")

print(f"Signature EAN-13 a decoder : {binary_signature_str}")

# Convertir la cha ne binaire en liste d'entiers
binary_signature = [int(bit) for bit in binary_signature_str]

try:
    code_barres = decode_ean13_signature(binary_signature)
    print(f"Code-barres EAN-13 decode : {code_barres}")
except ValueError as e:
    print(f"Erreur : {e}")
```

Listing 7 – decoder.py