ENSEIRB-MATMECA

LECTURE DE CODE-BARRES PAR LANCERS ALEATOIRES DE RAYONS
RAPPORT

PROJET TS225

Élèves :
Said Oubari
Zineb Mountich
Mayssen Mahmoud
Oussama Raji

Enseignants :
Marc Donias

7 janvier 2025

Table des matières

1 Introduction

Ce projet a pour objectif de développer, sous Python, un détecteur et lecteur de codes-barres basé sur le traitement d'images numériques. Le projet s'inscrit dans un contexte où les codes-barres jouent un rôle essentiel dans divers domaines, notamment la gestion des marchandises, le suivi logistique, et la dématérialisation de documents administratifs. L'approche retenue combine des concepts fondamentaux de traitement d'images, tels que la segmentation et l'analyse des gradients, à des méthodes algorithmiques pour identifier et décoder efficacement les codes-barres.

Dans un premier temps, l'algorithme est conçu pour détecter et lire des codes-barres bien orientés au sein d'une image. Par la suite, une méthode avancée permettant de gérer les orientations quelconques des codes-barres sera mise en œuvre. Enfin, des optimisations seront envisagées pour réduire le temps de calcul et augmenter la robustesse de la solution face aux contraintes réelles, comme les variations d'éclairage, la qualité des images ou les bruits visuels.

2 Phase 1 : Segmentation en régions d'intérêt

2.1 Calcul des Gradients de l'Image

Nous avons calculé les gradients de l'image avec la fonction **compute_gradients**. Cette étape permet de détecter les variations d'intensité dans les directions horizontale et verticale. Pour ce faire, nous avons utilisé les filtres dérivés d'une gaussienne, paramétrée par un écart-type σ_g, sous les noms de G_x et G_y. Ces filtres sont ensuite appliqués à l'image par convolution pour obtenir les gradients I_x et I_y, qui représentent respectivement les variations d'intensité dans les directions horizontale et verticale.

2.2 Construction du Tenseur

Avec la fonction **compute_structure_tensor**, nous avons construit le tenseur de structure local. Ce tenseur est obtenu en calculant les produits des gradients I_x et I_y dans les directions horizontale et verticale, suivis d'un lissage à l'aide d'un filtre gaussien paramétré par un écart-type σ_t. Plus précisément, nous avons calculé I_x^2, I_xI_y et I_y^2, puis appliqué le filtre gaussien à chacun de ces produits. Ainsi, nous avons pu capturer les relations entre les gradients dans un voisinage.

2.3 Calcul de la Cohérence

À partir du tenseur, nous avons calculé la cohérence locale grâce à la fonction **compute_coherence**. Cette cohérence, basée sur les valeurs propres du tenseur, mesure l'alignement des gradients. Une valeur élevée révèle une forte organisation directionnelle, typique des lignes parallèles d'un code-barres.

2.4 Segmentation des Régions Pertinentes

Pour isoler les régions pertinentes, nous avons utilisé la fonction **segment_regions**. Cette étape consiste à binariser les résultats de cohérence en appliquant un seuil, par

exemple le 97,5e percentile. Plus précisément, nous avons d'abord calculé ce seuil en utilisant la fonction `np.percentile`, qui permet de déterminer la valeur du percentile sur la matrice de cohérence. Ensuite, nous avons généré une image binaire où seules les zones ayant une cohérence supérieure à ce seuil sont conservées. Le résultat est une **matrice de cohérence**. Seules les régions avec un fort alignement des gradients, typiques des structures verticales d'un code-barres EAN13, sont extraites.

2.5 Algorithme de traitement des régions

L'algorithme suivant permet de générer des segments traversant les régions identifiées comme pouvant contenir des codes-barres. Ces segments sont obtenus par calcul du barycentre, de la matrice de covariance, et des valeurs propres associées.

Algorithm 1 Génération de segments à partir des régions

Require: Une image segmentée *segmented*
Ensure: Liste des segments $(x, y) \rightarrow (x_1, y_1)$
labeled $\leftarrow$ label(*segmented*)
results $\leftarrow$ []
for each région *region* dans regionprops(*labeled*) **do**
    coords $\leftarrow$ *region.coords*
    $x\_bar, y\_bar \leftarrow$ barycentre(coords)
    Calcul de la matrice de covariance *cov_matrix*
    Calcul des valeurs propres et vecteurs propres : λ_1, λ_2, vecteurs
    $L \leftarrow 2 \cdot \sqrt{\max(\lambda_1, \lambda_2)}$
    $x_1, y_1, x_2, y_2 \leftarrow$ extrémités du segment selon les vecteurs propres
    Ajouter les segments après rotations angulaires par $\alpha \in \{0, \pi/20, \dots\}$
end for
return *results*

L'algorithme commence par segmenter l'image donnée en identifiant les régions connectées à l'aide de la fonction `label`. Ces régions correspondent aux zones suspectes où un code-barres pourrait être présent.

Pour la suite pour chaque région on identifiée :

— Le barycentre $(x_{\text{bar}}, y_{\text{bar}})$ est calculé comme la moyenne des coordonnées des pixels de la région.
— Une matrice de covariance est construite pour capturer la distribution des pixels autour du barycentre.

Les valeurs propres de la matrice de covariance donnent les axes principaux de la région :

— La plus grande valeur propre correspond à l'axe principal de la région.
— Le vecteur propre associé est utilisé pour calculer l'orientation de l'axe.

À partir de l'orientation principale, des segments sont créés avec les étapes suivantes :

— Détermination des points d'extrémité du segment selon la direction principale.
— Rotation de ces segments autour du barycentre avec plusieurs angles α pour couvrir d'autres orientations possibles.
— Ajout des segments résultants à la liste `results`.

Cette méthode permet de générer plusieurs segments orientés à partir des régions segmentées. Ces segments traversent les régions avec des orientations calculées à partir des propriétés géométriques des zones,maximisant ainsi les chances de détecter correctement le(s) code(s)-barre(s).

2.6 Résultats intermédiaires

2.6.1 Traitement de l'image

Avant d'aborder l'étape de mesure de cohérence, il est crucial de travailler sur une image bruitée pour simuler des conditions d'acquisition réalistes. Cette étape permet de tester la robustesse des algorithmes développés face à des perturbations dues à l'environnement, tout en conservant les caractéristiques essentielles du code-barres.

Pour ce faire, un bruit gaussien est ajouté à l'image initiale. L'ajout de ce bruit permet non seulement de masquer certains parasites inutiles, mais également de s'assurer que les segments critiques, comme les lignes du code-barres, restent identifiables pour les prochaines étapes de traitement.

FIGURE 1 – Image originelle bruitée avant le calcul de la mesure de cohérence.

L'image ci-dessus constitue la base sur laquelle les gradients sont calculés, suivis par la construction du tenseur de structure. Ces étapes mènent finalement à l'estimation de la cohérence locale, qui permet de segmenter les régions probables contenant des codes-barres.

2.6.2 Mesure de la cohérence

Après avoir obtenu les gradients locaux et calculé le tenseur de structure, la prochaine étape consiste à estimer la mesure de la cohérence. Cette mesure permet d'identifier les

régions où les gradients sont fortement alignés, une caractéristique typique des codes-barres.

La mesure de cohérence est calculée à partir des valeurs propres du tenseur de structure. Elle varie entre 0 (absence de structure alignée) et 1 (structure hautement cohérente). Cette information est essentielle pour isoler les régions potentielles contenant un code-barres.

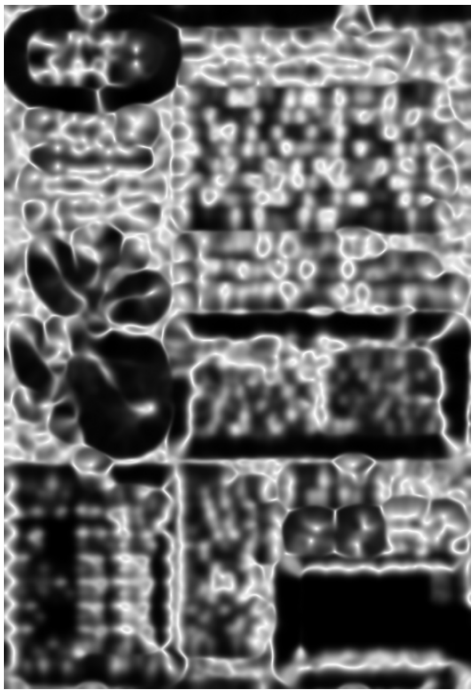

FIGURE 2 – Carte de la mesure de cohérence.

Comme illustré dans la figure 2, la mesure de cohérence met en évidence des zones structurées dans l'image. Ces zones seront ensuite segmentées et analysées pour déterminer les régions d'intérêt contenant des motifs caractéristiques des codes-barres.

2.7 Segmentation et binarisation

Une fois la mesure de cohérence calculée, l'étape suivante consiste à binariser l'image pour isoler les régions d'intérêt. Cette segmentation met en évidence les zones potentiellement pertinentes pour la détection des codes-barres.

Les régions binarisées sont ensuite segmentées pour détecter les structures rectangulaires caractéristiques des codes-barres.

La figure 3 montre le résultat de cette étape. Les segments en rouge représentent les structures détectées qui possèdent des propriétés géométriques spécifiques, comme une forte cohérence et une orientation parallèle, typiques des lignes de codes-barres. Ces résultats seront utilisés pour effectuer des analyses plus détaillées dans les prochaines étapes.

FIGURE 3 – Image binarisée et segmentée.

2.8 Conclusion

La phase de segmentation en régions d'intérêt constitue une étape essentielle dans le processus de détection des codes-barres. En commençant par le calcul des gradients, suivi de la construction du tenseur de structure et de l'estimation de la cohérence locale, nous avons pu extraire les zones où les gradients présentent une forte organisation directionnelle. Ces zones, mises en évidence par la segmentation et la binarisation, sont les candidats probables pour contenir des structures de type code-barres.

Grâce à ces traitements, nous avons obtenu des segments robustes qui facilitent l'identification précise des motifs caractéristiques. Ces résultats intermédiaires préparent ainsi le terrain pour les phases suivantes, où les segments seront analysés en détail pour extraire et décoder les informations des codes-barres EAN-13.

3 Phase 2 : Processus de Décodage EAN-13

3.1 Objectifs spécifiques liés au décodage

L'objectif principal du décodage des codes-barres est de convertir des informations visuelles sous forme de barres noires et blanches en données numériques exploitables, tout en respectant les spécifications de la norme EAN-13. Cette étape cruciale vise à garantir la fiabilité et l'automatisation des processus industriels et commerciaux, tels que l'identification de produits, la gestion des stocks et les transactions en caisse. Plus spécifiquement, il s'agit de :

— Développer un algorithme robuste capable de traiter des images de qualité variable (contraste, bruit, orientation).

— Assurer une extraction précise des bits codés à partir des motifs visuels, incluant la validation des motifs de garde et des chiffres encodés.
— Implémenter une vérification systématique de la validité des données extraites via la clé de contrôle.
— Réduire les interventions manuelles en automatisant les étapes critiques du processus, tout en maintenant une grande flexibilité pour des conditions réelles variées.

3.2 Méthodologie de Décodage Implémentée

3.2.1 Sélection des points

La sélection des points est une étape essentielle pour définir la ligne à analyser dans l'image du code-barres. Une interface utilisateur a été développée pour permettre une sélection manuelle et précise :

— L'utilisateur clique sur deux points : le début et la fin de la ligne passant à travers le code-barres.
— Les points sélectionnés sont visualisés sur l'image grâce à des cercles et une ligne tracée entre eux pour validation visuelle.
— Une réinitialisation est possible en cas d'erreur, garantissant une sélection correcte avant de passer aux étapes suivantes.

Cette méthode manuelle offre une flexibilité initiale pour s'assurer que la ligne choisie traverse bien le code-barres, même dans des images présentant des angles ou des distorsions.

FIGURE 4 – selection manuelle des points dans le code barre.

3.2.2 Prétraitement de l'image

Le prétraitement de l'image est une étape clé pour améliorer la qualité et la lisibilité du code-barres avant l'extraction du signal. Les étapes incluent :

— **Conversion en niveaux de gris :** Si l'image est en couleur, elle est convertie en niveaux de gris pour simplifier le traitement.
— **Amélioration du contraste :** Une méthode d'égalisation adaptative comme CLAHE (Contrast Limited Adaptive Histogram Equalization) est appliquée pour uniformiser les niveaux de contraste dans l'image.
— **Binarisation :** Un seuillage global, souvent avec la méthode d'Otsu, est utilisé pour convertir l'image en une représentation binaire (noir et blanc).
— **Réduction du bruit :** Un filtre gaussien est appliqué pour éliminer les petites variations indésirables et améliorer la continuité des lignes du code-barres.

Ces étapes garantissent que le signal extrait est suffisamment propre pour permettre une analyse précise.

FIGURE 5 – code barre avant et aprés traitement.

3.2.3 Extraction du signal

L'extraction du signal constitue une étape cruciale pour obtenir les données nécessaires au décodage. Les étapes incluent :

— **Projection d'une ligne horizontale :** À partir des points sélectionnés, une ligne horizontale est projetée sur l'image binarisée.
— **Échantillonnage le long de la ligne :** Les pixels traversés par la ligne sont échantillonnés pour créer un signal brut représentant les variations d'intensité le long de la ligne.
— **Suppression des marges blanches :** Les portions inutiles du signal (zones blanches à gauche et à droite) sont éliminées pour isoler la partie utile correspondant au code-barres.
— **Normalisation :** Le signal est ajusté pour s'assurer qu'il est aligné avec les unités de base définies par la norme EAN-13.

Ces étapes permettent de préparer un signal propre et exploitable pour les étapes suivantes de transformation et de décodage.

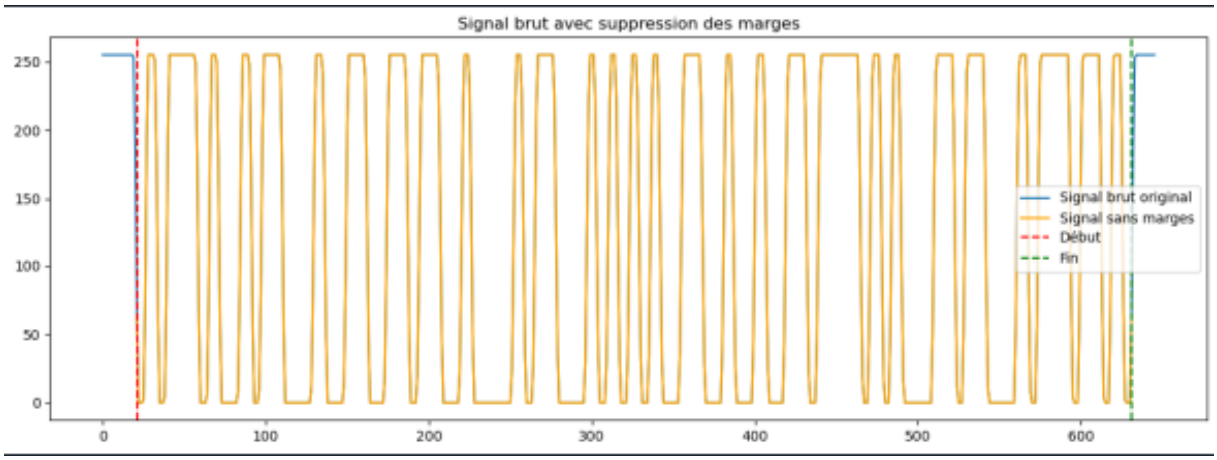

FIGURE 6 – Extraction de signal.

3.2.4 Transformation en binaire et extraction des bits

Pour traduire les variations d'intensité du signal en une séquence binaire exploitable, plusieurs étapes méthodiques sont suivies :

- **Lissage du signal :** Un filtre gaussien est appliqué sur le signal brut pour réduire les fluctuations dues au bruit tout en préservant les transitions nettes entre les barres noires et les espaces blancs.
- **Détermination du seuil :** Un seuil adaptatif, calculé en fonction de la moyenne ou de l'écart-type des valeurs du signal lissé, est utilisé pour distinguer les segments noirs (1) des segments blancs (0).
- **Segmentation en unités :** Le signal est divisé en 95 segments correspondant aux unités de base du code-barres EAN-13. Chaque unité est analysée pour déterminer si elle représente un "1" ou un "0".
- **Extraction des bits centraux :** Pour garantir une interprétation précise, seuls les centres des segments sont évalués, minimisant ainsi les erreurs dues aux transitions floues ou aux déformations des barres.

Ces étapes permettent d'obtenir une séquence binaire fiable, essentielle pour les validations ultérieures et le décodage des chiffres.

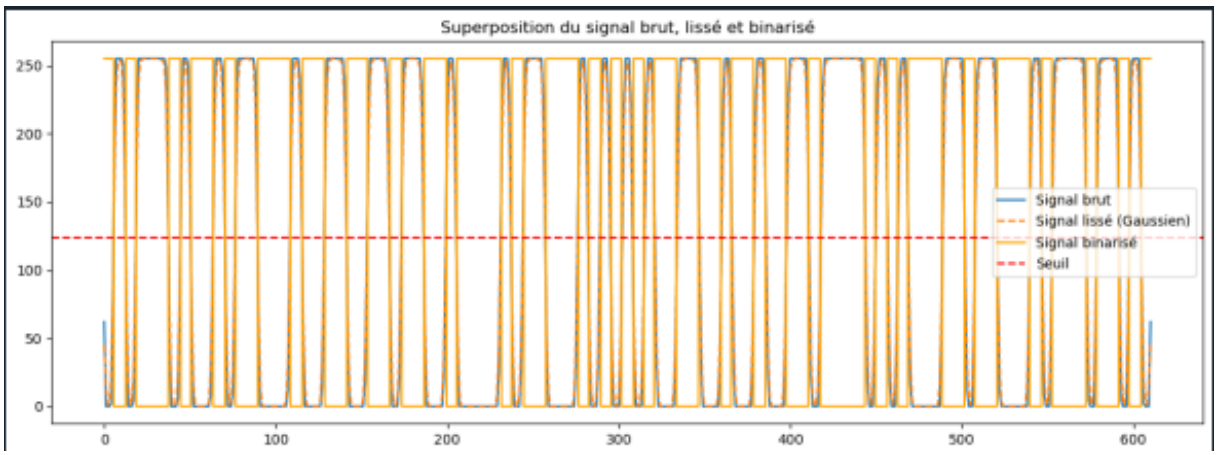

FIGURE 7 – Binarisation du signal.

3.2.5 Validation des motifs de garde

Les motifs de garde sont indispensables pour identifier avec précision les différentes sections du code-barres et structurer correctement son décodage. Leur validation s'effectue selon les étapes suivantes :

- **Motif de garde gauche :** Les trois premiers bits extraits doivent correspondre au motif spécifique [1, 0, 1], indiquant le début correct du code-barres.
- **Motif de garde central :** Les cinq bits situés au centre du code-barres sont comparés au motif attendu [0, 1, 0, 1, 0], servant de séparation entre les deux moitiés du code.
- **Motif de garde droit :** Les trois derniers bits du code-barres doivent également correspondre au motif [1, 0, 1], confirmant la fin du code.

Toute divergence par rapport à ces motifs entraîne un rejet du code-barres, garantissant que les étapes ultérieures de lecture et de décodage reposent sur une structure conforme à la norme EAN-13.

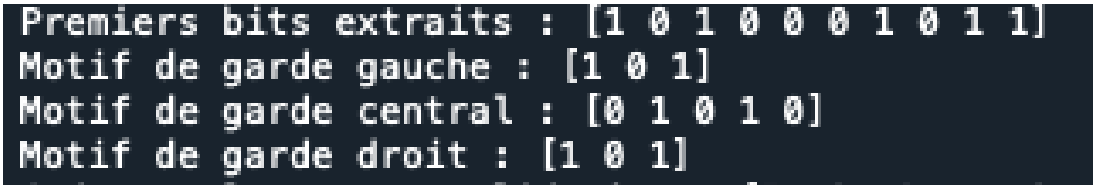

FIGURE 8 – Validation des motifs de garde et central

3.2.6 Décodage des chiffres

Le processus de décodage des chiffres repose sur l'analyse des segments binaires extraits et inclut un mécanisme de recherche avec la distance de Hamming pour corriger les éventuelles erreurs. Les étapes principales sont les suivantes :

- **Segmentation des bits :** La séquence binaire complète est divisée en segments de 7 bits, chaque segment représentant un chiffre unique du code-barres EAN-13.
- **Décodage des segments de gauche :** Les six premiers segments (après les bits de garde gauche) sont comparés aux motifs des familles `A` et `B`. Si un segment ne correspond pas directement à un motif attendu :
  - La distance de Hamming entre le segment et chaque motif valide est calculée.
  - Le motif avec la distance de Hamming minimale (inférieure à un seuil défini) est sélectionné comme candidat probable.
- **Identification du premier chiffre :** Les motifs des six premiers segments de gauche permettent de déterminer la famille d'encodage (`A` ou `B`) et, par conséquent, d'extraire le premier chiffre.
- **Décodage des segments de droite :** Les six segments de droite sont décodés à l'aide des motifs de la famille `C`. De manière similaire, si une correspondance exacte n'est pas trouvée, la distance de Hamming est utilisée pour sélectionner le motif le plus proche.
- **Validation et assemblage :** Une fois tous les chiffres identifiés, ils sont combinés avec le premier chiffre pour former le code complet. Ce code est ensuite soumis à une validation par la clé de contrôle.

L'utilisation de la distance de Hamming permet de corriger des erreurs mineures dans les segments binaires, augmentant ainsi la robustesse et la précision du décodage.

3.2.7 Validation finale avec la clé de contrôle

La clé de contrôle, dernier chiffre du code-barres EAN-13, est utilisée pour valider l'intégrité des données décodées. Le processus inclut un mécanisme de recherche et de vérification basé sur les étapes suivantes :

- **Calcul de la somme pondérée :** La somme des 12 premiers chiffres est calculée en appliquant un poids alterné de 1 et 3 à chaque chiffre. Par exemple, le premier chiffre est multiplié par 1, le deuxième par 3, le troisième par 1, et ainsi de suite.
- **Détermination de la clé attendue :** Le complément à 10 du reste de la division de la somme pondérée par 10 donne la clé attendue. Si le reste est nul, la clé est égale à 0.
- **Comparaison directe :** La clé calculée est comparée au dernier chiffre extrait du code-barres. Si elles correspondent, le code est validé.
- **Gestion des erreurs avec recherche :** En cas d'échec de la validation directe, une recherche basée sur la distance de Hamming est effectuée pour identifier d'éventuelles erreurs dans les segments binaires :
  - Les motifs binaires des chiffres sont modifiés un par un, en testant les corrections possibles.
  - Pour chaque modification, la somme pondérée et la clé correspondante sont recalculées.
  - Si une configuration valide est trouvée (clé calculée = clé attendue), le code-barres est corrigé et validé.

Ce processus assure une robustesse accrue en détectant et en corrigeant les erreurs mineures dans les données extraites, tout en maintenant une vérification stricte de l'intégrité des informations.

Algorithm 2 Décodage du code-barres EAN-13

Require: Une séquence binaire *bits* de longueur 95
Ensure: Code EAN-13 ou `None` en cas d'échec

Diviser *bits* en segments de 7 bits pour les chiffres
$left\_digits \leftarrow []$, $right\_digits \leftarrow []$, $errors \leftarrow []$
for each i dans $\{0, 1, \ldots, 5\}$ (partie gauche) **do**
    $pattern \leftarrow bits[3 + 7i : 3 + 7(i + 1)]$
    if *pattern* correspond à un motif des familles `A` ou `B` **then**
        Ajouter le chiffre correspondant à $left\_digits$
    else
        Calcul de la distance de Hamming pour trouver le motif le plus proche
        Ajouter le chiffre corrigé à $left\_digits$
        Enregistrer l'erreur dans *errors*
    end if
end for
Identifier le premier chiffre à partir de la séquence d'encodage (`A/B`)
for each i dans $\{0, 1, \ldots, 5\}$ (partie droite) **do**
    $pattern \leftarrow bits[50 + 7i : 50 + 7(i + 1)]$
    if *pattern* correspond à un motif de la famille `C` **then**
        Ajouter le chiffre correspondant à $right\_digits$
    else
        Calcul de la distance de Hamming pour trouver le motif le plus proche
        Ajouter le chiffre corrigé à $right\_digits$
        Enregistrer l'erreur dans *errors*
    end if
end for
Assembler *code* avec le premier chiffre, $left\_digits$ et $right\_digits$
if La clé de contrôle est valide **then**
    return *code*
else
    Appliquer des corrections sur *errors*, recalculer et valider
    if Une configuration valide est trouvée **then**
        return *code*
    else
        return `None`
    end if
end if

3.3 Résultats et Analyse

3.3.1 Résultats intermédiaires

FIGURE 9 – Image 1 du code-barres.

```
Premiers bits extraits : [1 0 1 0 0 0 1 0 1 1]
Motif de garde gauche : [1 0 1]
Motif de garde central : [0 1 0 1 0]
Motif de garde droit : [1 0 1]
Code complet avant validation : [5, 9, 0, 1, 2, 3, 4, 1, 2, 3, 4, 5, 7]
Code EAN-13 : 5901234123457
```

FIGURE 10 – Résultat du décodage de l'image 1.

FIGURE 11 – Image 2 du code-barres.

```
Premiers bits extraits : [1 0 1 0 1 1 1 0 1 1]
Motif de garde gauche : [1 0 1]
Motif de garde central : [0 1 0 1 0]
Motif de garde droit : [1 0 1]
Code complet avant validation : [9, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 7]
Code EAN-13 : 9781234567897
```

FIGURE 12 – Résultat du décodage de l'image 2.

FIGURE 13 – Image 3 du code-barres.

```
Premiers bits extraits : [1 0 1 0 1 1 0 1 1 1]
Motif de garde gauche : [1 0 1]
Motif de garde central : [0 1 0 1 0]
Motif de garde droit : [1 0 1]
Code complet avant validation : [3, 8, 0, 0, 0, 6, 5, 7, 1, 1, 1, 3, 5]
Code EAN-13 : 3800065711135
```

FIGURE 14 – Résultat du décodage de l'image 3.

FIGURE 15 – Image 4 du code-barres.

```
Premiers bits extraits : [1 0 1 0 0 0 1 1 0 1]
Motif de garde gauche : [1 0 1]
Motif de garde central : [0 1 0 1 0]
Motif de garde droit : [1 0 1]
Code complet avant validation : [5, 0, 0, 0, 2, 1, 8, 0, 0, 5, 6, 8, 7]
Code EAN-13 : 5000218005687
```

FIGURE 16 – Résultat du décodage de l'image 4.

FIGURE 17 – Image 5 du code-barres.

```
Premiers bits extraits : [1 0 1 0 1 0 1 1 1 1]
Motif de garde gauche : [1 0 1]
Motif de garde central : [0 0 0 0 0]
Motif de garde droit : [1 0 1]
Motifs de garde invalides
Échec de la lecture du code-barres
```

FIGURE 18 – Résultat du décodage de l'image 5.

3.3.2 Analyse des performances

L'algorithme a été testé sur cinq images présentant des qualités différentes, telles que des variations de contraste, de bruit ou d'orientation. Les résultats obtenus montrent que :

- Quatre des cinq images ont été correctement décodées grâce à la robustesse des étapes de prétraitement, d'extraction du signal et de validation.
- L'image présentant une très faible qualité (inclinaison importante et bruit élevé) n'a pas pu être décodée, illustrant les limites actuelles de l'algorithme dans des conditions extrêmes.

Les détails des images testées sont les suivants :

Image	Qualité	Description	Statut de décodage
Image 1 et 2	Bonne	Contraste net, peu de bruit	Décodée avec succès
Image 3	Moyenne	Légères déformations et bruit	Décodée avec succès
Image 4	Faible	Faible contraste, bruit élevé	Décodée avec succès
Image 5	Très faible	Orientation inclinée et bruit	Non décodée

TABLE 1 – Performances de décodage pour des images de qualité différente.

Les résultats montrent que l'algorithme est globalement performant, mais son efficacité diminue dans des scénarios extrêmes, tels que des images inclinées combinées à un bruit significatif. Ces cas mettent en évidence des pistes d'amélioration pour renforcer la robustesse du système.

3.4 Limitations des approches choisies

Les résultats des tests ont mis en évidence plusieurs limitations de la méthode utilisée :

— Une dépendance au bon choix des points pour l'extraction du signal. Bien que la sélection manuelle des points permette une certaine flexibilité, elle peut introduire des erreurs si les points ne sont pas correctement positionnés.

— Une performance réduite pour les images présentant un bruit important ou un contraste très faible. Cela a été particulièrement observé dans le cas de l'image 5, où l'inclinaison et le bruit élevé ont empêché le décodage.

— Une précision diminuée en cas de distorsion importante ou d'obstruction partielle du code-barres. Même si l'algorithme a pu gérer certaines déformations mineures (par exemple, l'image 3 et 4), les scénarios plus extrêmes restent problématiques.

Ces limitations soulignent la nécessité d'améliorer la robustesse de l'algorithme pour traiter des images dans des conditions plus variées et complexes.

3.5 Conclusion

Les tests effectués sur différentes images ont démontré que l'algorithme de décodage est capable de traiter efficacement des images avec un contraste net ou des déformations mineures. Cependant, des limitations ont été observées dans le cas d'une inclinaison importante et d'un bruit élevé, comme pour l'image 5, où le décodage n'a pas pu être réalisé.

Ces résultats soulignent la robustesse des étapes de prétraitement et de validation, tout en mettant en évidence des pistes d'amélioration nécessaires pour gérer des scénarios plus complexes, tels que le bruit extrême et les distorsions géométriques.

4 Conclusion générale

Ce projet de développement en Python pour la lecture des codes-barres a représenté un ensemble de défis stimulants, allant de la détection des zones probables contenant des codes-barres, basée sur le calcul des tenseurs de structure et la segmentation, jusqu'au décodage des codes-barres en utilisant des algorithmes robustes et précis.

Dans un premier temps, nous avons mis en œuvre une méthodologie pour isoler les régions d'intérêt dans une image. Cette étape a nécessité l'exploitation des propriétés géométriques des codes-barres, telles que leur structure parallèle et leur alignement directionnel, capturées à travers la mesure de cohérence. Le calcul des gradients, la construction du tenseur de structure et l'application de seuils de binarisation adaptés ont permis de segmenter efficacement l'image, en réduisant le bruit et en se concentrant sur les zones pertinentes.

Dans un second temps, les segments détectés ont été analysés en détail pour extraire les motifs caractéristiques des codes-barres. Les algorithmes développés ont utilisé des concepts tels que les barycentres, les matrices de covariance, les valeurs propres et les vecteurs propres pour générer des segments précis, maximisant les chances de décodage réussi. Ces segments ont ensuite été transformés en séquences binaires, qui ont été interprétées pour reconstruire le code EAN-13.

Enfin, le décodage des codes-barres a requis une validation par le biais de la clé de contrôle intégrée, garantissant ainsi la fiabilité des résultats obtenus. Malgré les défis posés

par le bruit, les variations d'éclairage ou encore l'orientation des codes-barres, les solutions mises en place ont démontré une grande robustesse et adaptabilité.

En somme, ce travail a démontré l'importance d'une approche méthodique et progressive pour relever des défis complexes, et constitue une base solide pour de futures explorations dans le domaine de la vision par ordinateur.

Répartition du Travail

Élèves	Phase Réalisée
Said Oubari et Mayssen Mahmoud	Phase 1 : Segmentation en régions d'intérêt .
Zineb Mountich et Oussama Raji	Phase 2 : Processus de Décodage EAN-13 .

TABLE 2 – Répartition du travail dans le groupe.

5 Annexe

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter
from scipy import ndimage
from scipy import signal
from skimage.measure import label, regionprops

def compute_gradients(image, sigma_g):
    P = int(6 * sigma_g + 1)
    x = np.arange(-P // 2 + 1, P // 2 + 1)
    y = np.arange(-P // 2 + 1, P // 2 + 1)
    X, Y = np.meshgrid(x, y)

    # D riv es gaussiennes
    Gx = -X * np.exp(-(X**2 + Y**2) / (2 * sigma_g**2)) / (2 * np.pi *
        sigma_g**4)
    Gy = -Y * np.exp(-(X**2 + Y**2) / (2 * sigma_g**2)) / (2 * np.pi *
        sigma_g**4)

    Ix = ndimage.convolve(image, Gx)
    Iy = ndimage.convolve(image, Gy)

    return Ix, Iy

def compute_structure_tensor(Ix, Iy, sigma_t):
    IxIx = gaussian_filter(Ix * Ix, sigma=sigma_t)
    IxIy = gaussian_filter(Ix * Iy, sigma=sigma_t)
    IyIy = gaussian_filter(Iy * Iy, sigma=sigma_t)

    return IxIx, IxIy, IyIy

def compute_coherence(IxIx, IxIy, IyIy):
    T_xx = IxIx
    T_xy = IxIy
    T_yy = IyIy
```

```
34
35      # Calcul des valeurs propres
36      c = np.sqrt((T_xx - T_yy)**2 + 4 * (T_xy**2))
37      d = T_xx + T_yy
38      coherence = c / (d + 1e-10)  #  viter  division par z ro
39
40      return coherence
41
42  def segment_regions(coherence, percentile_threshold):
43      threshold = np.percentile(coherence, percentile_threshold)
44      segmented = coherence > threshold
45      return segmented
46
47  def regions_with_covariance(segmented):
48      labeled = label(segmented)
49
50      results = []
51      for region in regionprops(labeled):
52          # R cup rer les coordonn es des pixels de la r gion
53          coords = region.coords
54          x_coords = coords[:, 1]
55          y_coords = coords[:, 0]
56
57          # Calcul du barycentre
58          x_bar = np.mean(x_coords)
59          y_bar = np.mean(y_coords)
60
61          # Calcul de la matrice de covariance
62          x_diff = x_coords - x_bar
63          y_diff = y_coords - y_bar
64
65          cov_matrix = np.array([
66              [np.sum(x_diff**2), np.sum(x_diff * y_diff)],
67              [np.sum(x_diff * y_diff), np.sum(y_diff**2)]
68          ])/len(coords)
69
70          # Calcul des valeurs propres
71          eigenvalues, eigenvectors = np.linalg.eig(cov_matrix)
72
73
74          max_index = np.argmax( eigenvalues)
75          vector= eigenvectors[:,  max_index]
76
77          vx,vy=   vector
78
79          # V rifier le rapport des valeurs propres
80          lamda1 = eigenvalues[0]
81          lamda2 = eigenvalues[1]
82          val = max(lamda1,lamda2)
83
84
85          theta = np.arctan2(vy, vx)
86          L = 2 * np.sqrt(val)
87          x1 = x_bar + np.cos(theta)*L
88          y1 = y_bar + np.sin(theta)*L
89
90          x2 = x_bar - L * np.cos(theta)
```

```
        y2 = y_bar - L * np.sin(theta)


        alphas = [0, np.pi / 20, np.pi / 21, np.pi/22, np.pi / 23]

        for alpha in alphas:
            # Calcul des nouvelles extr mit s avec rotation d'angle alpha
            cos_alpha = np.cos(alpha)
            sin_alpha = np.sin(alpha)

            # Rotation du point 1
            x1_rot = x_bar + cos_alpha * (x1 - x_bar) - sin_alpha * (y1 - y_bar)
            y1_rot = y_bar + sin_alpha * (x1 - x_bar) + cos_alpha * (y1 - y_bar)

            # Rotation du point 2
            x2_rot = x_bar + cos_alpha * (x2 - x_bar) - sin_alpha * (y2 - y_bar)
            y2_rot = y_bar + sin_alpha * (x2 - x_bar) + cos_alpha * (y2 - y_bar)

            results.append(((x1_rot, y1_rot), (x2_rot, y2_rot)))

    return results



# Charger l'image
I_rgb = plt.imread("bar.jpg")
R = I_rgb[:,:,0].astype('float64')
G = I_rgb[:,:,1].astype('float64')
B = I_rgb[:,:,2].astype('float64')
I = (R +G +B)/3

bruit = np.random.normal(0, 2, I.shape)
image = I + bruit


# Calcul des gradients
sigma_g = 2
sigma_t = 5
Ix, Iy = compute_gradients(image, sigma_g)

# Calcul du tenseur de structure
IxIx, IxIy, IyIy = compute_structure_tensor(Ix, Iy, sigma_t)

# Calcul de la mesure de coh rence
coherence = compute_coherence(IxIx, IxIy, IyIy)

# Segmentation des r gions probables
segmented = segment_regions(coherence, percentile_threshold=97.5)


result = regions_with_covariance(segmented)

```

```
print(result)


inverted_image = np.ones_like(image)
inverted_image[segmented] = 0



# Affichage des r sultats
plt.figure()
plt.imshow(I, cmap='gray')
plt.axis('off')

plt.figure(figsize=(15, 10))

plt.subplot(1, 2, 1)
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(inverted_image, cmap='gray')
plt.axis('off')

# Ajouter les segments sur l'image
for (point1, point2) in result:
    x1, y1 = point1
    x2, y2 = point2
    plt.plot([x1, x2], [y1, y2], color='red', linewidth=2)  # Tracer le segment

# Afficher la figure

plt.tight_layout()

plt.figure(figsize=(15, 10))
plt.imshow(1 - coherence, cmap='gray')
plt.axis('off')

plt.show()
```

Listing 1 – Code python : Extraction des segments traversant le code-barres

```
import numpy as np
import cv2
import matplotlib.pyplot as plt
from scipy.ndimage import gaussian_filter1d

class PointSelector:
    def __init__(self):
        self.points = []
        self.window_name = "S lectionnez deux points sur le code-barres"

    def mouse_callback(self, event, x, y, flags, param):
        if event == cv2.EVENT_LBUTTONDOWN:
            if len(self.points) < 2:
                self.points.append((x, y))
```

```
                cv2.circle(self.image, (x, y), 3, (0, 255, 0), -1)
                if len(self.points) == 2:
                    cv2.line(self.image, self.points[0], self.points[1],
                        (0, 255, 0), 2)
                cv2.imshow(self.window_name, self.image)

    def get_points(self, image):
        self.image = image.copy()
        self.points = []

        cv2.namedWindow(self.window_name)
        cv2.setMouseCallback(self.window_name, self.mouse_callback)
        print("S lectionnez deux points et appuyez sur 'Entr e' pour
            confirmer")

        while True:
            cv2.imshow(self.window_name, self.image)
            key = cv2.waitKey(1) & 0xFF
            if key == 13 and len(self.points) == 2:  # Entr e
                break
            elif key == ord('r'):  # R initialiser
                self.image = image.copy()
                self.points = []

        cv2.destroyAllWindows()
        return self.points

class BarcodeReader:
    def __init__(self, debug=True):
        self.debug = debug
        self.encoding_patterns = {
            'A': {
                '0001101': 0, '0011001': 1, '0010011': 2, '0111101': 3,
                '0100011': 4, '0110001': 5, '0101111': 6, '0111011': 7,
                '0110111': 8, '0001011': 9
            },
            'B': {
                '0100111': 0, '0110011': 1, '0011011': 2, '0100001': 3,
                '0011101': 4, '0111001': 5, '0000101': 6, '0010001': 7,
                '0001001': 8, '0010111': 9
            },
            'C': {
                '1110010': 0, '1100110': 1, '1101100': 2, '1000010': 3,
                '1011100': 4, '1001110': 5, '1010000': 6, '1000100': 7,
                '1001000': 8, '1110100': 9
            }
        }
        self.first_digit_encoding = {
            'AAAAAA': 0, 'AABABB': 1, 'AABBAB': 2, 'AABBBA': 3, 'ABAABB'
                : 4,
            'ABBAAB': 5, 'ABBBAA': 6, 'ABABAB': 7, 'ABABBA': 8, 'ABBABA'
                : 9
        }

    def preprocess_image(self, image):
        """Pr traitement pour am liorer le contraste et binariser l'
            image."""
```

```
        #  tape  1 : Convertir en niveaux de gris
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) if len(image.
            shape) == 3 else image

        #  tape  2 :  galisation  adaptative (CLAHE) pour am liorer le
            contraste
        clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8, 8))
        enhanced = clahe.apply(gray)

        #  tape  3 : Binarisation avec seuillage global (m thode Otsu)
        _, binary = cv2.threshold(enhanced, 0, 255, cv2.THRESH_BINARY +
            cv2.THRESH_OTSU)

        # Optionnel : Lissage pour  liminer  le bruit
        binary = cv2.GaussianBlur(binary, (3, 3), 0)

        # Visualisation pour le d bogage
        if self.debug:
            plt.figure(figsize=(10, 5))
            plt.subplot(1, 2, 1)
            plt.title("Image am lior e (contraste)")
            plt.imshow(enhanced, cmap='gray')
            plt.axis('off')
            plt.subplot(1, 2, 2)
            plt.title("Image binaris e")
            plt.imshow(binary, cmap='gray')
            plt.axis('off')
            plt.show()

        return binary

    def extract_signal_along_horizontal_line(self, image, point1, point2
        ):
        """Extrait le signal le long d'une ligne horizontale projet e
            sur le code-barres."""
        # Pr traitement avanc  de l'image
        binary = self.preprocess_image(image)

        # Projeter point2 sur une ligne horizontale passant par point1
        point_proj = (point2[0], point1[1])  # Nouvelle coordonn e
            projet e

        # Extraire le signal entre point1 et point_proj
        distance = abs(point_proj[0] - point1[0])
        num_points = int(distance)

        #  G n rer les coordonn es
        x = np.linspace(point1[0], point_proj[0], num_points)
        y = np.full_like(x, point1[1])  # y constant

        coordinates = np.column_stack((x, y)).astype(np.float32)

        # Remap pour extraire le signal
        signal = cv2.remap(
            binary,
            coordinates[:, 0].reshape(1, -1),
            coordinates[:, 1].reshape(1, -1),
```

```
            cv2.INTER_LINEAR
        )[0]

        # Supprimer les marges blanches
        threshold = np.mean(signal) + 0.1 * np.std(signal)  # Seuil adaptatif
        start_index = 0
        while start_index < len(signal) and signal[start_index] > threshold:
            start_index += 1

        end_index = len(signal) - 1
        while end_index > start_index and signal[end_index] > threshold:
            end_index -= 1

        trimmed_signal = signal[start_index:end_index+1]

        if self.debug:
            print(f"Signal ajust  : D but  l'indice {start_index}, fin  l'indice {end_index}")
            plt.figure(figsize=(15, 5))
            plt.plot(signal, label="Signal brut original")
            plt.plot(range(start_index, end_index+1), trimmed_signal, label="Signal sans marges", color='orange')
            plt.axvline(x=start_index, color='red', linestyle='--', label="D but")
            plt.axvline(x=end_index, color='green', linestyle='--', label="Fin")
            plt.legend()
            plt.title("Signal brut avec suppression des marges")
            plt.show()

        return trimmed_signal, distance

    def extract_signal_and_invert(self, signal):
        """Convertit le signal en binaire et applique une inversion avec contr le du lissage."""
        # Lissage du signal avec un filtre gaussien
        smoothed_signal = gaussian_filter1d(signal, sigma=0.8)

        # Seuillage manuel bas  sur la moyenne et l' cart -type
        threshold = np.mean(smoothed_signal)
        binary = (smoothed_signal < threshold).astype(int)  # Noir -> 1, Blanc -> 0

        if self.debug:
            plt.figure(figsize=(15, 5))
            plt.plot(signal, label="Signal brut")
            plt.plot(smoothed_signal, label="Signal liss  (Gaussien)", linestyle='--')
            plt.step(range(len(binary)), binary * 255, label="Signal binaris ", color='orange', where='mid')
            plt.axhline(y=threshold, color='red', linestyle='--', label="Seuil")
            plt.legend()
            plt.title("Superposition du signal brut, liss  et binaris ")
```

```
163             plt.show()
164
165         return binary
166
167     def extract_bits_center(self, binary_sequence, total_pixels):
168         """Extrait les bits en choisissant le centre de chaque unit .
                """
169         unit_width = total_pixels / 95
170         bits = []
171
172         for i in range(95):
173             start = int(i * unit_width)
174             end = int((i + 1) * unit_width)
175             segment = binary_sequence[start:end]
176             bit = 1 if np.mean(segment) > 0.5 else 0
177             bits.append(bit)
178
179         return np.array(bits)
180
181     def validate_guard_patterns(self, bits):
182         """V rifie les motifs de garde."""
183         if len(bits) != 95:
184             return False
185         print("Motif de garde gauche :", bits[:3])
186         print("Motif de garde central :", bits[45:50])
187         print("Motif de garde droit :", bits[-3:])
188
189         return (
190             list(bits[:3]) == [1, 0, 1]
191             and list(bits[45:50]) == [0, 1, 0, 1, 0]
192             and list(bits[-3:]) == [1, 0, 1]
193         )
194     def decode_ean13(self, bits):
195         """D code les bits en code EAN-13 avec recalcul en cas d' chec
                de la somme de contr le."""
196         if len(bits) != 95:
197             if self.debug:
198                 print(f"Erreur : Longueur de bits incorrecte ({len(bits)
                        } au lieu de 95)")
199             return None
200         def find_closest_pattern(pattern, encoding_dict, max_distance=1,
                context=None):
201             closest_match = None
202             closest_distance = max_distance + 1  # Initialise avec une
                    distance invalide
203             closest_digit = None
204             best_score = float('inf')  # Le score le plus faible est
                    pr  f  r
205
206             for valid_pattern, digit in encoding_dict.items():
207                 distance = sum(c1 != c2 for c1, c2 in zip(pattern,
                        valid_pattern))
208                 if distance > max_distance:
209                     continue
210
211                 score = distance
212                 if context:
```

```
                    # Ajuster le score en fonction du contexte
                    if 'last_digit' in context and abs(digit - context['last_digit']) > 1:
                        score += 1
                    if 'sequence_encoding' in context and digit not in context['sequence_encoding']:
                        score += 1

                if score < best_score:
                    best_score = score
                    closest_match = valid_pattern
                    closest_distance = distance
                    closest_digit = digit

            # V rifier si aucun motif valide n'a   t   trouv
            if closest_digit is None:
                return None, None, None, None

            return closest_digit, closest_match, closest_distance


        try:
            left_digits = []
            left_encoding = []
            errors = []

            # D codage partie gauche
            for i in range(6):
                pattern = ''.join(map(str, bits[3 + i * 7:3 + (i + 1) * 7]))
                found = False
                for encoding in ['A', 'B']:
                    if pattern in self.encoding_patterns[encoding]:
                        left_digits.append(self.encoding_patterns[encoding][pattern])
                        left_encoding.append(encoding)
                        found = True
                        break
                if not found:
                    digit, match, distance = find_closest_pattern(pattern, self.encoding_patterns['A'])
                    if digit is not None:
                        left_digits.append(digit)
                        left_encoding.append('A')  # Suppose 'A' encoding for approximation
                        errors.append((3 + i * 7, match, distance))
                    else:
                        return None

            encoding_sequence = ''.join(left_encoding)
            if encoding_sequence not in self.first_digit_encoding:
                return None
            first_digit = self.first_digit_encoding[encoding_sequence]

            # D codage partie droite
            right_digits = []
            for i in range(6):
```

```
264                 pattern = ''.join(map(str, bits[50 + i * 7:50 + (i + 1)
                        * 7]))
265                 if pattern in self.encoding_patterns['C']:
266                     right_digits.append(self.encoding_patterns['C'][
                            pattern])
267                 else:
268                     digit, match, distance = find_closest_pattern(
                            pattern, self.encoding_patterns['C'])
269                     if digit is not None:
270                         right_digits.append(digit)
271                         errors.append((50 + i * 7, match, distance))
272                     else:
273                         return None
274 
275             # Assembler le code complet
276             complete_code = [first_digit] + left_digits + right_digits
277             if self.debug:
278                 print(f"Code complet avant validation : {complete_code}"
                        )
279 
280             # Validation de la somme de contr le
281             if self.validate_checksum(complete_code):
282                 return complete_code
283 
284             # Tentative de recalcul avec motifs alternatifs
285             for error_index, (bit_start, alt_match, distance) in
                    enumerate(errors):
286                 for other_match, alt_digit in self.encoding_patterns['C'
                        ].items():
287                     if other_match == alt_match:
288                         continue  #  viter  de r utiliser le m me
                                motif
289                     # Modifiez les bits concern s
290                     new_bits = bits.copy()
291                     new_bits[bit_start:bit_start + 7] = list(map(int,
                            other_match))
292                     # Recalculer
293                     new_digits = self.decode_ean13(new_bits)
294                     if new_digits:
295                         return new_digits
296 
297             if self.debug:
298                 print("Aucune alternative valide trouv e apr s
                        recalcul")
299             return None
300 
301         except Exception as e:
302             if self.debug:
303                 print(f"Erreur lors du d codage avec recalcul : {e}")
304             return None
305 
306 
307     def validate_checksum(self, digits):
308         """Valide la somme de contr le EAN-13."""
309         if len(digits) != 13:
310             return False
311 
```

```
312         weighted_sum = sum(digits[i] * (3 if i % 2 else 1) for i in
                range(12))
313         check_digit = (10 - (weighted_sum % 10)) % 10
314         return check_digit == digits[-1]
315
316     def process_barcode(self, image, point1, point2):
317         """Processus complet de lecture du code-barres."""
318         try:
319             signal, _ = self.extract_signal_along_horizontal_line(image,
                    point1, point2)
320             binary = self.extract_signal_and_invert(signal)
321             bits = self.extract_bits_center(binary, len(binary))
322
323             print("Premiers bits extraits :", bits[:10])
324
325             if not self.validate_guard_patterns(bits):
326                 print("Motifs de garde invalides")
327                 return None
328
329
330             digits = self.decode_ean13(bits)
331             return digits
332
333         except Exception as e:
334             print(f"Erreur : {e}")
335             return None
336
337 def main():
338     image = cv2.imread("ee.jpeg")
339     if image is None:
340         print("Impossible de charger l'image")
341         return
342
343     selector = PointSelector()
344     points = selector.get_points(image)
345
346     if not points:
347         print("S lection annul e")
348         return
349
350     reader = BarcodeReader(debug=True)
351     digits = reader.process_barcode(image, points[0], points[1])
352
353     if digits:
354         print(f"Code EAN-13 : {''.join(map(str, digits))}")
355     else:
356         print(" chec  de la lecture du code-barres")
357
358 if __name__ == "__main__":
359     main()
```

Listing 2 – Code Python : Décodage de Code-Barres EAN-13