École nationale supérieure
d'électronique, informatique,
télécommunications, mathématiques et
mécanique de Bordeaux

Lecture de code-barres par lancers aléatoires de rayons

Département Télécom
TS225 : Projet Images
Rapport

Élèves :
Elisa Chien
Alban Oberti
Tierno-Alpha Tall
David Yan

Encadrant :
Marc DONIAS

Table des matières

1 Introduction

Le projet présenté dans ce rapport s'intéresse à la lecture automatique des codes-barres à partir d'images. L'objectif principal est de développer un algorithme capable de détecter et de lire un code-barres en utilisant des techniques de traitement d'image.

D'abord, nous détaillerons les algorithmes utilisés pour la détection et la lecture du code-barres. Ensuite, nous expliquerons l'implémentation de ces algorithmes, puis présenterons les résultats obtenus illustrés par des exemples d'images traitées. Enfin, nous conclurons sur les performances globales.

Note importante

Dans notre approche, nous avons considéré dès le départ la détection des régions d'intérêts comme préalable à la lecture des codes barres. Pour cette raison, nous avons appelé la phase de repérage des codes barres *"Phase 1"*, et la phase de lecture *"Phase 2"*.

2 Algorithmes

2.1 Phase 1 : Détection du code barre

Dans cette première phase, nous devons détecter des zones susceptibles de contenir un code barre et tracer un rayon qui permet la lecture du code barre.

Segmentation en régions d'intérêt

Les régions d'intérêt correspondent aux zones où il y a des codes-barres. Le code-barre présente une géométrie particulière avec de grandes bandes noires, très fines et parallèles entre elles. Cette uniformité dans la répartition des bandes et l'alternance de bandes noires et blanches se traduit par une configuration spécifique du vecteur de gradient de l'intensité de l'image.

L'algorithme vise à mettre en évidence cette configuration. Pour cela, nous allons calculer le tenseur de structure local. Ce tenseur nous permettra de calculer une mesure de cohérence, qui nous permettra d'évaluer si une certaine région ou structure dans l'image présente une uniformité ou une régularité dans ses caractéristiques. Elle nous permettra de détecter la géométrie particulière du code-barre.

$$T(x,y) = \begin{bmatrix} T_{xx}(x,y) & T_{xy}(x,y) \\ T_{xy}(x,y) & T_{yy}(x,y) \end{bmatrix}$$

FIGURE 1 – Tenseur de structure local

Cette matrice permet de mettre en évidence des zones de concentrations importantes de transitions entre pixels sombres et clairs ; autrement dit l'alternance des barres dans un code barre.

$$T_{ab}(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} W(u,v) I_a(x+u, y+v) I_b(x+u, y+v)\, du\, dv$$

où `W(x,y)` est la fonction bidimensionnelle de pondération du voisinage au point (`x`, `y`) et `I(x,y)` le vecteur gradient d'intensité.

FIGURE 2 – Coefficient du tenseur de structure local

Pour calculer la mesure de cohérence, nous allons devoir d'abord calculer les vecteurs gradient d'intensité de l'image, ainsi que la fonction de pondération du voisinage pour pouvoir former le tenseur de structure local. Ensuite, la mesure de cohérence se calcule avec les coefficients du tenseur. Pour avoir de meilleur résultat, nous avons normalisé les vecteurs gradient d'intensité de l'image. Enfin, pour obtenir les régions d'intérêts, nous effectuons un seuillage sur la mesure de cohérence. Cette dernière permet de garder

$$D(x,y) = \sqrt{\frac{\left(T_{xx}(x,y) - T_{yy}(x,y)\right)^2 + 4\left(T_{xy}(x,y)\right)^2}{T_{xx}(x,y) + T_{yy}(x,y)}}$$

FIGURE 3 – Mesure de cohérence

seulement les régions structurées (et donc probables de contenir un code barres) de manière ordonnées parmi les zones d'intérêt candidates.

Tracé d'un segment traversant le code barre

Après avoir obtenu les différentes régions d'intérêt, nous les labélisons pour pouvoir effectuer les manipuler. Ce processus nous permet d'obtenir un nuage de points. On peut dès cette étape tracer des rayons qui le traversent en son centre, de façon aléatoire ou régulière. En outre, une méthode plus astucieuse consiste en l'extraction d'informations sur la forme de du nuage de points à partir de sa matrice de covariance. En diagonalisant

$$\Sigma = \begin{pmatrix} \mathrm{Var}(X) & \mathrm{Cov}(X,Y) \\ \mathrm{Cov}(Y,X) & \mathrm{Var}(Y) \end{pmatrix}$$

FIGURE 4 – Matrice de covariance Σ

celle-ci, on obtient : λ_1 caractérise la dispersion des points du nuage autour de l'axe définit

$$\mathbf{D} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \text{et} \quad (u_1, u_2) \quad \text{les vecteurs propres associés}$$

FIGURE 5 – Diagonalisation de Σ

par u_1, et λ_1 la dispersion autour de l'axe défini par u_2. En effet, les valeurs propres sont homogènes à une variance, et par conséquent leurs racines carrées permettent de mesurer approximativement la largeur du nuage. En particulier, on utilise la valeur propre la plus grande λ_{max} pour déterminer la largeur du code barre, et le vecteur propre u_min, associé à la valeur propre minimale λ_{min} pour calculer son orientation.

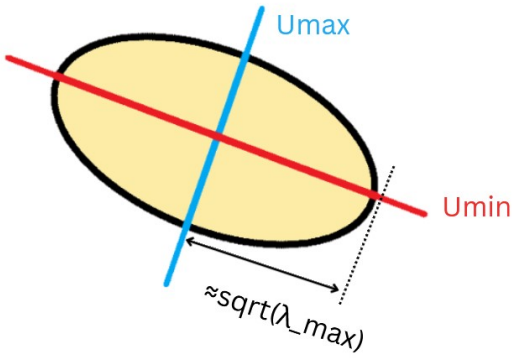


FIGURE 6 – Schéma d'un nuage de points et des axes correspondants

C'est en utilisant ces informations qu'on est capable de tracer un rayon qui traverse le code barre dans sa largeur, en suivant le procédé ci-dessous.

Algorithm 1 Calcul de l'axe et du rayon

procedure CALCUL DU RAYON POUR UN NUAGE DONNÉ
    Entrées : k ▷ facteur d'ajustement
    Calculer les valeurs propres λ_1, λ_2 et les vecteurs propres associés u_1 et u_2
    Comparer les valeurs propres pour déterminer λ_{min} et λ_{max}
    Extraire le vecteur propre u_{min} associé à la valeur propre minimale λ_{min}
    Calculer la longueur approximative du nuage de points : $r \leftarrow k \cdot \sqrt{\lambda_{\max}}$
    Calculer $ray \leftarrow r \times \frac{u_{min}}{\|u_{min}\|_2}$
    return ray
end procedure

2.2 Phase 2 : Lecture du code barre

La phase précédente a permis de récupérer un ensemble de couple de points représentant des rayons. Pour chaque couple de points, nous déterminons d'une part les 13 valeurs du code barre, et d'autre part, nous vérifions s'il est valide ou non.

Seuillage à l'aide d'Otsu

La méthode d'Otsu est une technique de seuillage de Nobuyuki Otsu utilisée en traitement d'images pour la segmentation binaire. Elle vise à séparer les pixels d'une image en deux classes distinctes : l'avant-plan et l'arrière-plan. Pour ce faire, on construit l'histogramme des niveaux de gris de l'image et évalue la dispersion des pixels pour chaque seuil possible. Le seuil optimal est celui qui maximise la variance inter-classe, assurant ainsi une séparation claire entre les deux classes. En pratique, le seuil calculé se situe entre les deux pics de l'histogramme.

Détermination des limites du code-barres

La première étape de cette phase est de réduire le segment initial, de sorte que les deux nouveaux points correspondent à la première barre noire du code barre. Pour cela, nous devons tout d'abord opérer un seuillage de l'image grâce au seuil donné par l'algorithme d'Otsu.

Cet ensemble de points va alors être binarisé grâce au seuil. C'est à dire que les points de l'image ayant une valeur inférieure au seuil vont être associés à 0 et ceux ayant une valeur supérieure au seuil vont être associés à 1.

Les points déterminant le début et la fin du code-barres sont ensuite déterminés en prenant le premier et dernier point de valeur 0.

Finalement, un dernier échantillonnage suivi d'une binarisation sur le même seuil permet de trouver la liste binaire à analyser. Cette liste devant être compatible avec les hypotheses de données auxquelles nous allons la comparer, la binarisation ce fera dans le sens contraire de la première. C'est à dire que les points de valeurs inférieures au seuil vont valoir 1 tandis que les autres vont valoir 0. Le nombre de point échantillonnés est calculé en prenant $N = u * 95$ avec u le plus petit multiple de 95 tel que u*95>L avec L la longueur du segment du code barre

Analyse du code barre binaire

La prochaine étape consiste à séparer les différents blocs contenu dans la liste binaire obtenu précédemment en ne gardant que ceux représentant des nombres Pour décoder chaque blocs, ils font être comparé avec des données de décodage en adaptant celle-ci à la taille du bloc Le choix se fera en prenant la plus petit norme de la différence entre le bloc et les données comparés.

Le résultat de cette analyse se traduit par une liste de 12 chiffres et d'une liste de 6 lettres (A ou B) déterminésde de la même façon que pour les chiffres avec le tableau précédent. Cette dernière permet de déterminer le premier chiffre du code-barres

	Élément A	**Élément B**	**Élément C**
0	BBBNBNN	BNBBNNN	NNNBBNB
1	BBNNBBN	BNNBBNN	NNBBNNB
2	BBNBBNN	BBNNBNN	NNBNNBB
3	BNNNNBN	BNBBBNN	NBBBNNB
4	BNBBBNN	BBNNNBN	NBNNBB
5	BNNBBBN	BNNNBBN	NBBNNNB
6	BBNBNNN	BBBNBNN	NBNBBNB
7	BNNNBNN	BBNBBBN	NBBBNNB
8	BNNBNNN	BBBBBNN	NBBNBBB
9	BBBNBNN	BBNBNNN	NNNBNBB

TABLE 1 – Tableau des éléments et des séquences associées

Détermination du premier chiffre du code barre

À partir de la séquence AB obtenue, nous pouvons déterminer le premier chiffre. Il suffit ainsi de lire le tableau de correspondance suivant :

Séquence	**1er Chiffre**	**Séquence**	**1er Chiffre**
AAAAAA	0	ABBAAB	5
AABABB	1	ABBBAA	6
AABBAB	2	ABABAB	7
AABBBA	3	ABABBA	8
ABAABB	4	ABBABA	9

TABLE 2 – Tableau des familles d'appartenance de la séquence

Vérification de la clef de contrôle

Pour déterminer la clé de contrôle, il faut additionner les chiffres en positions paires à trois fois la somme des chiffres en positions impaires jusqu'au 12^{eme} chiffre. La clé est alors valable si le chiffre des unités est égal au complément à 10 du 13^{eme} chiffre du code.

3 Implémentation

3.1 Phase 1 : Détection du code barre

Image

À partir d'une image en couleur RGB, nous la convertissons en YCbCr et ne gardons que le canal de luminance. En effet, un code-barre est une alternance de bandes noires et blanches, donc seul ce contraste d'intensité nous intéresse.
On ajoute d'ailleurs un bruit blanc gaussien centré de variance σ^2_{bruit} dès le chargement de l'image. Cela améliore la capacité de détection.

Mesure de cohérence

Pour le calcul de la mesure de cohérence, nous devons tout d'abord calculer les vecteurs gradients d'intensité. Pour cela, nous les calculons au moyen des filtres de Canny, ce qui correspond à appliquer un filtre convolutif gaussien avec un écart type σ_g. Cet écart type doit être relativement faible pour trouver les vecteurs de transition correspondant aux barres blanches et noires.

Ensuite, nous calculons la fonction de pondération du voisinage à l'aide d'un filtre passe-bas gaussien, avec un écart type σ_t. Cet écart type doit être proportionnel à la taille du code-barre par rapport à la taille de l'image pour trouver des clusters de vecteurs gradient d'intensité.

Labélisation

La labélisation des objets détectés dans l'image binaire est réalisée à l'aide de la fonction `measure.label` de la bibliothèque `skimage`. Cette fonction attribue un label unique à chaque blob extrait par le biais des étapes précédentes, permettant ainsi de les distinguer et de les analyser individuellement. À l'aide de celle-ci, nous obtenons une image labélisée accompagnée du nombre total d'objets détectés. Finalement, on obtient le nuage de points correspondant à toutes les régions potentiellement intéressantes qui ont été détectées.

De plus, pour éviter d'avoir des zones d'intérêt parasites, on rajoute sur l'image d'origine, avant le traitement d'image, un bruit blanc gaussien pour réduire la qualité de l'image. En effet, ce bruit va perturber les zones de faux positifs sans trop perturber le code-barre puisque sa structure est très contrastée.

Classe Blob

Une fois les différentes régions extraites de l'image, il nous fallait être en capacité de tracer des rayons pour la lecture. C'est dans cet esprit que nous avons implémenté une classe *Blob*. Celle-ci fournit une interface qui permet à l'utilisateur d'obtenir toutes les données utiles relatives à un blob sans qu'il ait à se soucier de l'implémentation. Ceci nous permet de produire un code plus synthétique et de fluidifier l'échange entre les deux équipes de projet.

Une instance de *Blob* contient une batterie d'attributs nécessaires au calcul des éléments essentiels pour le tracé : on peut citer le barycentre, la matrice de covariance, ses valeurs

et vecteurs propres, etc. Ces attributs sont accompagnés de méthodes qui implémentent plusieurs approches pour le tracé d'un rayon sur un blob donné. On peut par exemple tracer des rayons aléatoires qui passent par le barycentre du blob, ou un rayon de taille adéquate aligné sur l'axe du code-barre tout en étant capable d'ajuster sa longueur pour tenir compte de la déformation induite par le processus d'extraction des zones d'intérêt.

On peut aussi, à travers cette interface, aisément ajouter des méthodes de calcul qui permettraient d'affiner la détection : par exemple, pour utiliser d'autres approches pour le calcul du rayon, ou l'ajout de filtres sur la forme du blob pour quantifier une probabilité qu'il soit ou non un code-barre. Elle présente aussi l'avantage d'être simple d'utilisation : l'utilisateur fournit la taille d'une image et un nuage de points en son sein en entrée, et peut ensuite directement obtenir les données qui lui sont utiles.

Complexité temporelle

Nous nous sommes efforcés de produire un code relativement efficace en termes de complexité temporelle.
Initialement, celui-ci se terminait en une trentaine de secondes. Nous avons utilisé, dès que possible, des calculs matriciels plutôt que des boucles imbriquées, et fait un usage extensif des fonctionnalités de *numpy*, qui fournit des fonctions efficaces pour les calculs d'algèbre linéaire. Le changement le plus impactant a été l'ajout de la fonction *fftconvolve* de *scipy*, qui tire parti de l'algorithme de la Transformée de Fourier Rapide pour effectuer des convolutions en un temps record.
Ces améliorations nous ont permis d'atteindre une exécution de l'ordre d'une seconde.

3.2 Phase 2 : Lecture du code barre

Contrairement à la phase précédente, nous avons implémenté la phase 2 entièrement en programmation impérative. Chaque étape clé de l'algorithme est implémentée par une fonction spécifique, tout en respectant les principes de modularité et de réutilisabilité du code.

Les fonctions réutilisées sont :
- `distance` : Calcule la distance entre deux points.
- `echantillonnage` : Échantillonne un segment pour un nombre de points donné en choisissant le plus proche voisins avec np.around.
- `norme_binaire` : Calcule la norme binaire entre les blocs binaires du code-barres et les séquences à comparer. Cette norme se traduit dans l'implémentation par une comparaison bit à bit.

Les fonctions implémentées sont :
- **Seuillage à l'aide d'Otsu** : La fonction `otsu` permet de seuiller une image en niveaux de gris en utilisant la méthode d'Otsu. D'abord, l'algorithme crée un histogramme de l'image. Ensuite, il parcourt tous les seuils possibles (de 0 à 255 dans notre cas), puis met à jour les valeurs des probabilités de classe et des moyennes de classe. Dès lors, il calcule la variance inter-classe et vérifie itérativement s'il est maximal. En effet, minimiser la variance intra-classe revient à maximiser la variance inter-classe.

— **Détermination des limites du code-barres** : La fonction `find_lim` permet de déterminer les limites du code barre à partir d'un segment donné. La fonction `find_u` permet de déterminer le facteur u pour échantillonner le second segment de code barre.
— **Analyse du code barre binaire** : La fonction `separate` détermine les douze blocs de données (ignorant les gardes) à partir du segment seuillé et les placent dans un tableau, facilitant alors le déchiffrement. La fonction `compare` va alors utiliser la fonction `norme_binaire` pour comparer chaque bloc binaire du tableau sortant de `separate` et permettre de récupérer les 12 chiffres décodés et une liste contenant la séquence AB.
— **Détermination du premier chiffre du code barre** : La fonction `first_one` permet de déterminer le premier chiffre du code barre à partir de la séquence AB.
— **Vérification de la clef de contrôle** : La fonction `clef_controle` permet de vérifier la validité du code barre en calculant la clé de contrôle. Elle est calculée par le produit scalaire avec le vecteur [1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3]. De plus, nous avons utilisé un modulo 10 sur le complément à 10 pour prendre en compte que le complément à 10 de 0 est 0.

4 Résultats

4.1 Phase 1 : Détection du code barre

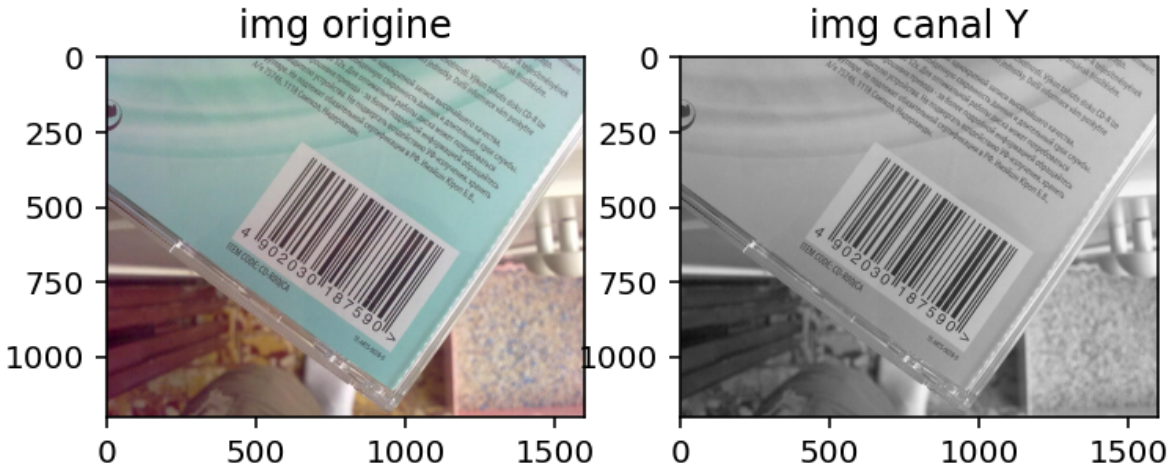


FIGURE 7 – Image originale et canal de Luminance

Nous convertissons l'image couleur RGB en image canal Y pour ne garder que l'intensité du contraste. Cela nous permet de bien distinguer les bandes noires et blanches.

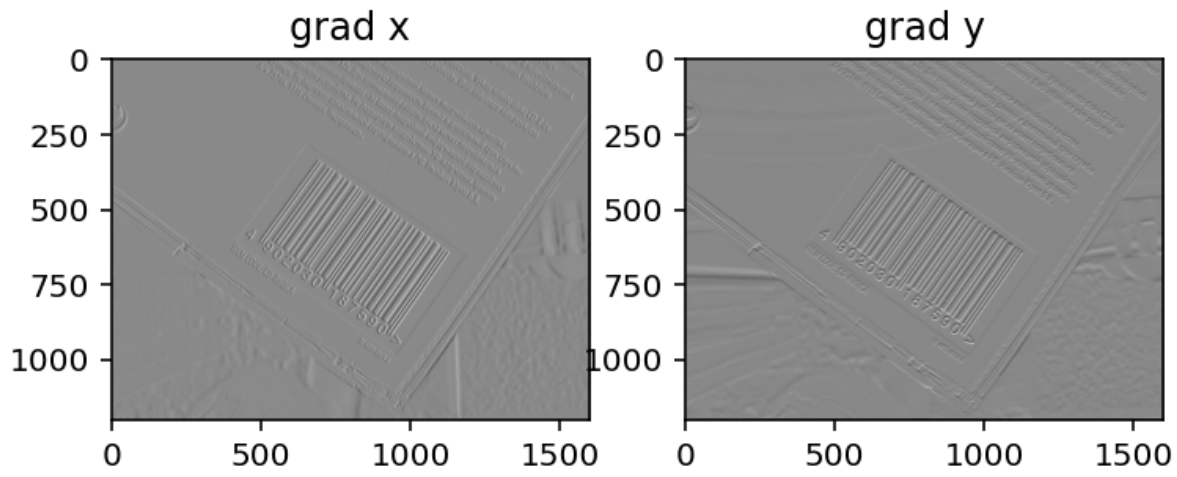


FIGURE 8 – Gradients en X et en Y

À partir du canal de luminance, nous calculons le vecteur gradient selon x et selon y. Pour vérifier qu'il s'agit du gradient selon x, nous observons dans l'image à gauche que l'on constate bien la variation horizontale d'intensité à la coordonnée (1200,1000), donc nous avons bien un gradient selon x. Et pour l'image de droite, nous observons une variation verticale à la coordonnée (0,750), donc nous avons un gradient selon y.

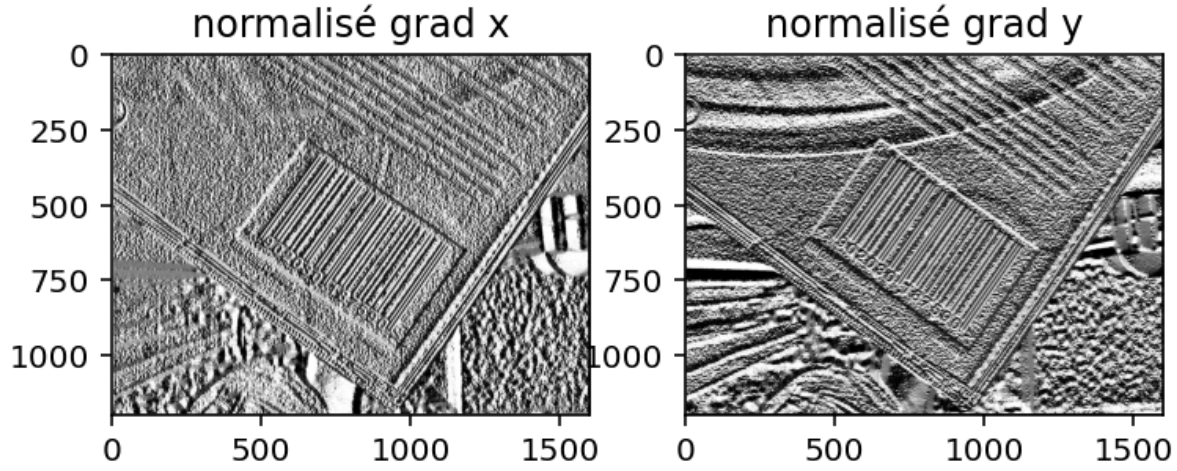


FIGURE 9 – Normalisation des gradients

La normalisation permet d'accentuer les transitions de pixels pour faciliter la détection des codes-barres et permet d'avoir un tenseur de structure local bien contrasté.

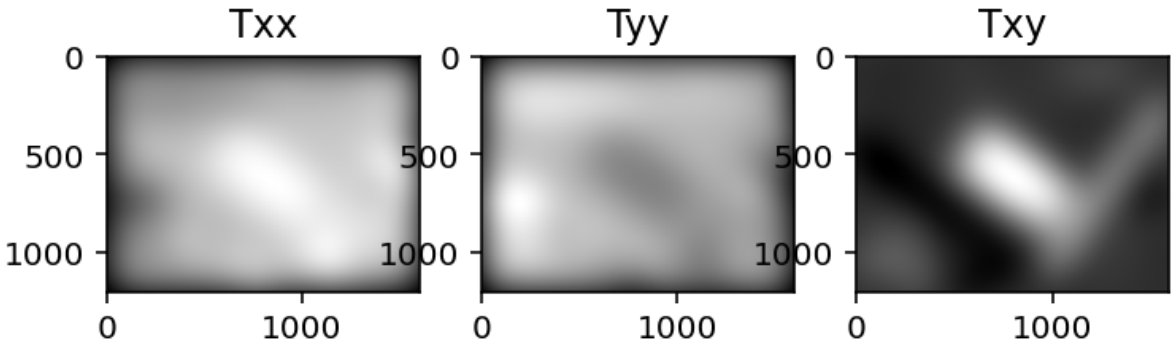


FIGURE 10 – Visualisation des composantes du tenseur de structure local

En observant le tenseur T_{xy}, on peut voir la zone du code-barre très contrastée par rapport au reste de l'image.

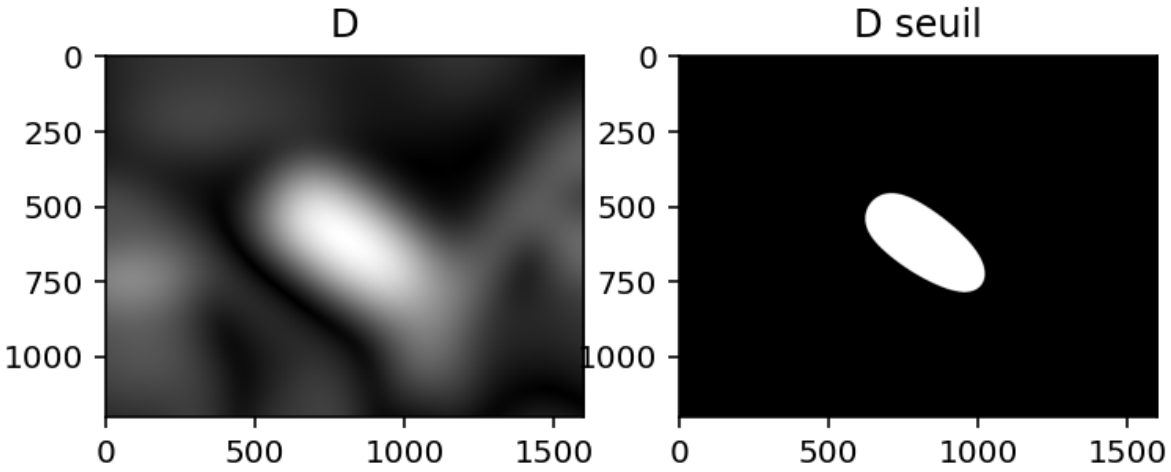


FIGURE 11 – Mesure de cohérence et seuillage

La mesure de cohérence seuillée met en évidence une région d'intérêt, qui correspond à la zone où le code-barre se situe. Nous effectuons un seuillage pour mettre en évidence la zone d'intérêt et ainsi faciliter la labélisation.

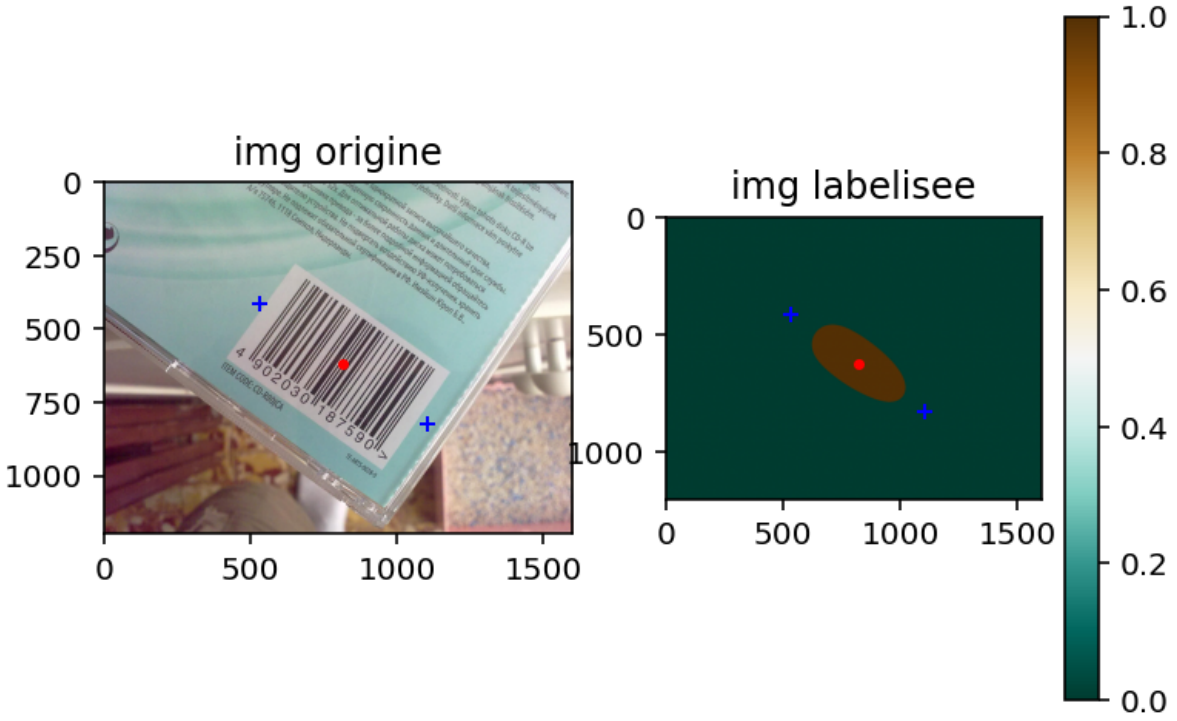


FIGURE 12 – Labélisation et tracé de rayon

En choisissant des paramètres adéquats :

$$\sigma_g = 2, \quad \sigma_t = 50, \quad \text{seuil} = 0.7, \quad \sigma_{\text{bruit}} = 2$$

On est obtient un tracé de rayon pertinent.
Ces paramètres sont différents pour chaque image et doivent respecter les conditions citées en 3.1

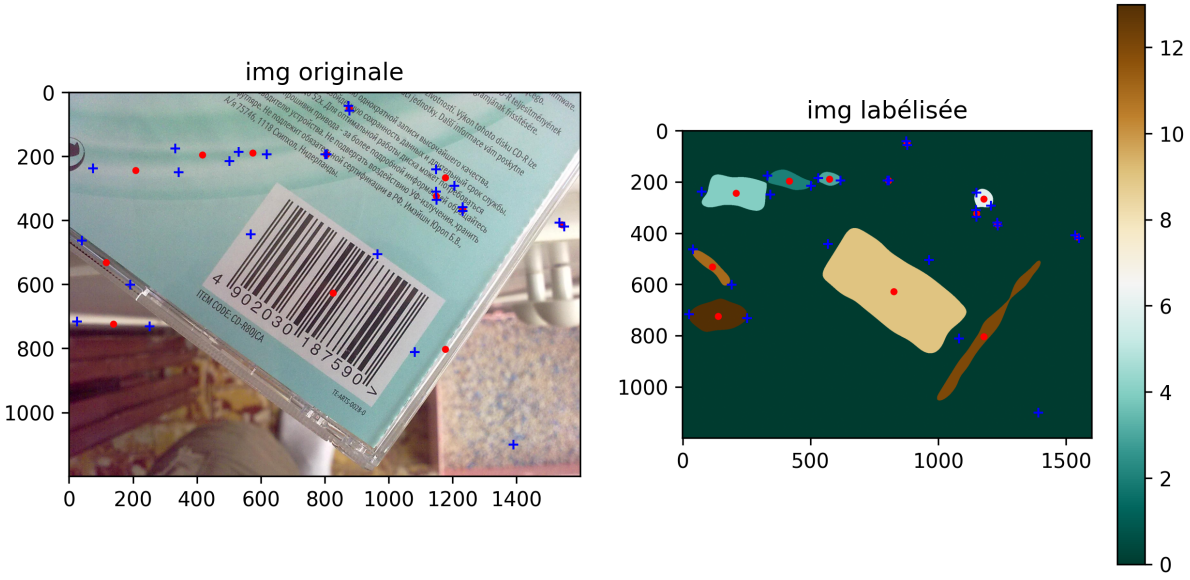


FIGURE 13 – Détection de régions parasites

Si les paramètres de détection ne sont pas bien choisis, des régions parasites font surface. Ici :

$$\sigma_g = 5, \quad \sigma_t = 30, \quad \text{seuil} = 0.7, \quad \sigma_{\text{bruit}} = 2$$

4.2 Phase 2 : Lecture du code barre

Seuillage à l'aide d'Otsu

Prenons l'image suivante :

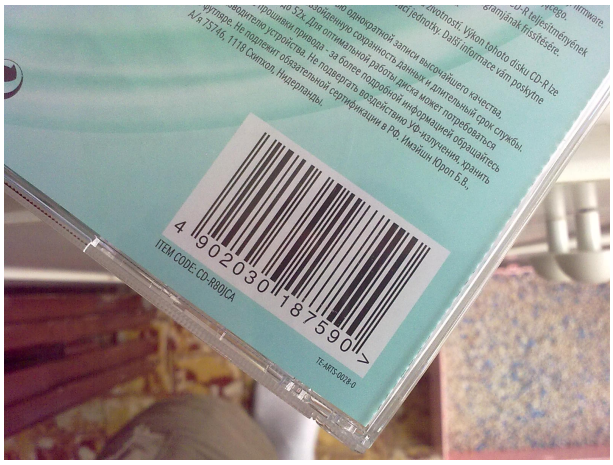


FIGURE 14 – Image d'un code barre

Avant de pouvoir lire un code barre, nous devons opérer un seuillage de l'image. Pour cela, nous utilisons la méthode d'Otsu. Pour rappel, cette méthode repose sur le calcul d'un seuil optimal pour binariser une image en niveaux de gris. Ce seuil est déterminé grâce à l'histogramme de l'image et sépare les pixels en deux classes distinctes : les pixels noirs et les pixels blancs.

Pour l'image en figure 14, nous obtenons l'histogramme suivant :

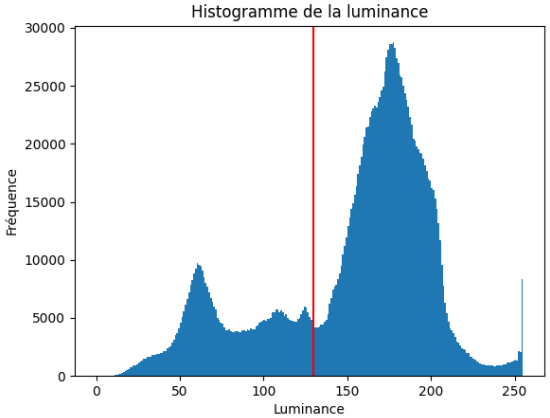


FIGURE 15 – Histogramme de la luminance de l'image

Grâce à ce seuil dénoté par la barre rouge, nous obtenons ainsi l'image seuillée :

FIGURE 16 – Image seuillée

Détermination des limites du code-barres

Une fois le seuil déterminé, nous pouvons déterminer les limites du code barre. La première étape consiste à un échantillonnage sur un segment puis une binarisation selon un seuil. Ce segment est visble en vert sur la figure suivante.

Nous recevons en argument deux points, qui définissent un segment sur l'image seuillée, comme représenté sur la figure 17. Nous souhaitons alors réduire ce segment pour ne garder que les deux points correspondant à la première barre noire du code barre.

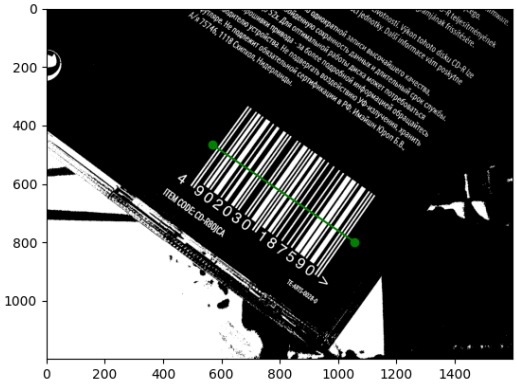


FIGURE 17 – Segment défini par le deux points en arguments

Ensuite, la recherche des limites du code-barres amènent à un nouveau segment entre les deux limites. Nous obtenons alors le nouveau segment entre les points en rouge sur l'image seuillée, comme illustré sur la figure 18 :

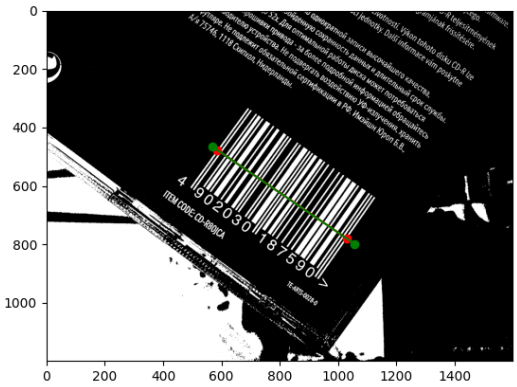


FIGURE 18 – Nouveau segment produit par l'échantillonnage et la binarisation

Celle-ci montre l'efficacité de la recherche de limites qui correspondent bien à celles observables sur l'image ci-dessus.

Analyse du code barre binaire

On en déduit alors une liste de valeurs binaires. Ceci nous permet ainsi de détecter les zones noires et blanches du code-barres. La figure 19 illustre cette détection, où les zones noires sont représentées en rouge et les zones blanches en bleu.

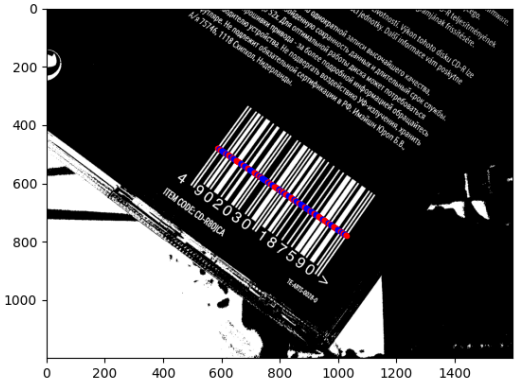


FIGURE 19 – Détection du noir et blanc

Nous notons ici que le décalage apparent sur l'image n'est que le résultat du dé-zoom de l'image. À petite échelle, on voit clairement que les zones sont bien détectées (cf Figure 20).

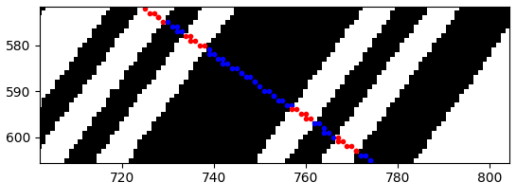


FIGURE 20 – Zoom de la détection

Détermination du premier chiffre du code barre

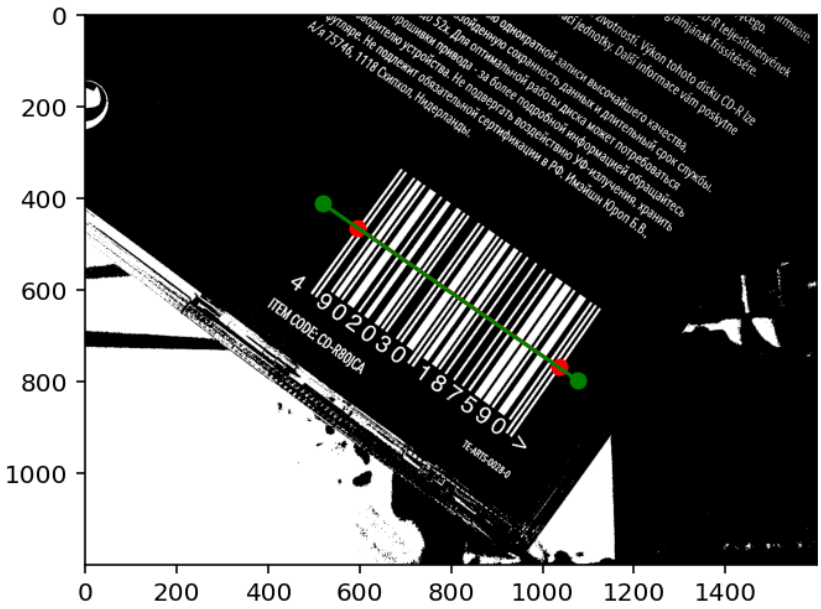
À l'aide de la séquence AB obtenue, nous pouvons ainsi déterminer le premier chiffre du code barre en se référant au tableau 2. En l'occurrence, nous obtenons le chiffre **4** comme premier chiffre.

Vérification de la clef de contrôle

Nous obtenons alors pour cette image le code **4902030187590**, ce qui correspond bien au code barre présent sur l'image (cf Figure 14). De manière générale, il faut tout de même vérifier la validité du code barre. Pour cela, nous suivons l'algorithme de vérification de code barre grâce à la clé de contrôle.

En l'occurrence, le complément à 10 du dernier chiffre du code barre vaut 0, tandis que le chiffre des unités de la somme des chiffres de rang impair et trois fois la somme des chiffres de rang pair vaut 0. Le code barre est donc valide.

4.3 Unification des algorithmes




```
Reloaded modules: phase1, common, Blob, barcode_detection
1 objects detected in img
Axe: [1076.60581107  797.9800507 ][519.61548512 411.36423286]
Code barre valide :  [4. 9. 0. 2. 0. 3. 0. 1. 8. 7. 5. 9. 0.]
Temps d'exécution :  0.3939390000887215
Temps d'exécution global: 1.98s
```

FIGURE 21 – Lecture d'un code barre combinant les deux phases

En figure 21 on observe la sortie obtenue en appliquant l'algorithme de détection qui trace le rayon en vert, utilisé finalement par le programme de lecture. Le code barre détecté est valide, et correspond parfaitement à celui dans l'image.
En combinant ces programmes, on est en mesure, avec des paramètres adéquats à l'image d'effectuer une lecture correcte en un temps de l'ordre de deux secondes.

5 Conclusion

Ce projet a consisté à développer un système automatique de lecture de codes-barres à partir d'images. Dans la phase 2, nous avons implémenté un algorithme de détection reposant sur la mesure de cohérence calculée à partir du tenseur de structure local. Après seuillage et labélisation, un rayon traversant les régions d'intérêt a été tracé pour isoler les zones contenant un code-barres.

Dans la phase 2, la lecture des codes-barres a été réalisée à l'aide de l'échantillonnage et de la binarisation du segment détecté. La méthode d'Otsu a permis de segmenter efficacement les niveaux de gris, et un tableau de correspondance a été utilisé pour décoder les chiffres et valider la clé de contrôle.

L'unification des deux phases a permis de traiter des images très rapidement avec une bonne précision. Les optimisations, notamment l'utilisation de calculs matriciels et de la transformée de Fourier rapide, ont significativement amélioré la performance.

En résumé, nous avons conçu un système complet, performant et adaptable pour la lecture de codes-barres sur des images complexes.

6 Bilan de l'organisation

6.1 Séance 1

	Tâches entreprises	Temps passé
Elisa	Prise de connaissance du sujet et commencement de la méthode d'Otsu	2h
Alban	Lecture et analyse du sujet et début de l'implémentation d'algorithme d'échantillonnage et de binarisation	2h
Tierno	découverte de la problématique, tests sur l'implémentation	2h
David	Lecture du sujet et implémentation des différents filtres	2h

TABLE 3 – Organisation de la séance 1

6.2 Séance 2

	Tâches entreprises	Temps passé
Elisa	Finition de la méthode d'Otsu	2h
Alban	Complétion de l'échantillonnage et de la binarisation, début du calcul des limites du code-barres et de la recherche de u	2h
Tierno	Optimisations et corrections	3h
David	Premier prototype de la détection de code barre	2h

TABLE 4 – Organisation de la séance 2

6.3 Séance 3

	Tâches entreprises	Temps passé
Elisa	Débuggage et mise en commun de toutes les fonctions de la phase 2	3h
Alban	Débuggage des fonctions de décodage des blocs fait chez soi et mise en commun du travail phase 2	3h
Tierno	labélisation et extraction des régions d'intérêt	3h
David	Implémentation du calcul des couples de points	3h

TABLE 5 – Organisation de la séance 3

6.4 Séance 4

	Tâches entreprises	Temps passé
Elisa	Dernier débuggage et test sur des exemples pour vérifier l'efficacité du code	3h
Alban	Dernier débuggage et test sur des exemples pour vérifier l'efficacité du code	3h
Tierno	Factorisations, implémentation de la classe Blob (en partie en travail peronnel)	6h
David	Mise en commun avec la 2e phase et derniers débuggages	3h

TABLE 6 – Organisation de la séance 4

6.5 Séance 5

	Tâches entreprises	Temps passé
Elisa	Rédaction du rapport	3h
Alban	Rédaction du rapport	3h
Tierno	Corrections dans l'algorithme de tracé de rayons	3h
David	Rédaction du rapport	3h

TABLE 7 – Organisation de la séance 5

7 Annexes

7.1 Phase 1 : Détection du code barre

7.1.1 Détection des codes barres et tracé de rayons

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from skimage import io, color, measure
from scipy import signal
from scipy.signal import fftconvolve
from Blob import Blob
from time import time
from math import floor, sqrt
import os
from common import *
start_time = time()
# ~~~~~~ Fonctions préliminaires ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

def G_2D(sigma):
    x = range(floor(-3*sigma), floor(3*sigma+1))
    X, Y = np.meshgrid(x, x)
    return np.exp(-1/2*(X**2/(sigma**2)+(Y**2/sigma**2)))


def G_x_prime(sigma):  # derive d'une gaussienne
    P = range(floor(-3*sigma), floor(3*sigma+1))
    X, Y = np.meshgrid(P, P)
    return (-X/(2*np.pi*sigma**4)*np.exp(-(X**2+Y**2)/(2*sigma**2)))


def G_y_prime(sigma):  # derive d'une gaussienne
    P = range(floor(-3*sigma), floor(3*sigma+1))
    X, Y = np.meshgrid(P, P)
    return (-Y/(2*np.pi*sigma**4)*np.exp(-(X**2+Y**2)/(2*sigma**2)))

def D(X, Y, Z): return np.sqrt((X-Y)**2+4*Z**2)/(X+Y)

def barcode_detection(img,sigma_g,sigma_t,seuil,sigma_bruit=2,affichage=
    False):
    # ~~~~~~ Préparation de l'image ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    # %%
    img_code_barre = plt.imread(img)
    # ~~~~ transformation de rgb en ycrcb  ~~~~~~~~~~~~~~~~~~~
    img_code_barre_YCbCr = color.rgb2ycbcr(img_code_barre)
    Y_code_barre = img_code_barre_YCbCr[:, :, 0]
    Cb_code_barre = img_code_barre_YCbCr[:, :, 1]
    Cr_code_barre = img_code_barre_YCbCr[:, :, 2]
    Y_code_barre += np.random.randn(len(Y_code_barre),len(Y_code_barre[0]))
    *sigma_bruit
    # ~~~~~~~~~~~~~~~~~~~ Gradients ~~~~~~~~~~~~~~~~~~~
    gauss_x_prime = G_x_prime(sigma_g)
    gauss_y_prime = G_y_prime(sigma_g)
    h, w, c = np.shape(img_code_barre_YCbCr)
    x = np.linspace(0, h, h)
    y = np.linspace(0, w, w)
```

```
    X, Y = np.meshgrid(y, x)
    I_x = fftconvolve(Y_code_barre, gauss_x_prime, mode='same')
    I_y = fftconvolve(Y_code_barre, gauss_y_prime, mode='same')
    # ~~~~~ Normalisation ~~~~~~~~~~~~~~~~~~~
    delta_I = np.stack((I_x, I_y), axis=-1)
    N_delta_I = np.linalg.norm(delta_I, ord=2, axis=-1)
    N_I_x = I_x/N_delta_I
    N_I_y = I_y/N_delta_I

    # ~~~~ tenseur de structure local T ~~~~~~~~~~~~~~~~~~~
    gauss2D = G_2D(sigma_t)
    Txx = fftconvolve(N_I_x * N_I_x, gauss2D, mode='same')
    Tyy = fftconvolve(N_I_y * N_I_y, gauss2D, mode='same')
    Txy = fftconvolve(N_I_x * N_I_y, gauss2D, mode='same')
    T = np.block([[Txx, Txy], [Txy, Tyy]])
    # ~~~~ Mesure de cohérence ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    D_res = D(Txx, Tyy, Txy)
    # ~~~~ Seuillage ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    D_seuil = (D_res > seuil)
    # ~~~~ Labelisation ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    D_labeled,num_labels=measure.label(D_seuil,return_num=True)
    blobs=measure.regionprops(D_labeled)
    print(f"{num_labels} objects detected in img")
    coords=[x.coords for x in blobs]
    # ~~~~ Extraction de l'axe ~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    # En utilisant la méthode vectorielle
    Blobs=[Blob(pixels=x.coords,imsize=[h,w]) for x in blobs]
    axis=[b.calc_axis_ray(6) for b in Blobs]

    # Methode des points extrêmes
    # Blobs=[Blob(pixels=x.coords,imsize=[h,w]) for x in blobs]
    # axis=[b.calc_axis_extr() for b in Blobs]
    #=====================================================================
    # affichage basique des rayons obtenus
    plt.figure()
    plt.subplot(1, 2, 1)
    plt.imshow(img_code_barre)
    for blob in Blobs:
        # assert(blob.axis!=None)
        x=[u[0] for u in blob.axis]
        y=[u[1] for u in blob.axis]
        p=blob.barycentre
        plt.plot(p[1],p[0],"or",markersize=2.5)
        plt.plot(x,y,"+b",markersize=5)
    plt.title("img origine")
    plt.subplot(1, 2, 2)
    plt.imshow(D_labeled, cmap=cm.BrBG_r)
    for blob in Blobs:
        # assert(blob.axis!=None)
        x=[u[0] for u in blob.axis]
        y=[u[1] for u in blob.axis]
        p=blob.barycentre
        plt.plot(p[1],p[0],"or",markersize=2.5)
        plt.plot(x,y,"+b",markersize=5)
    plt.title("img labelisee")
    plt.colorbar()
```

```
107     plt.show()
108     if affichage:
109         # ~~~~~ Affichage détaillé ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
110 
111         plt.figure(1)
112         plt.subplot(1, 2, 1)
113         plt.imshow(img_code_barre)
114         plt.title("img origine")
115         plt.subplot(1, 2, 2)
116         plt.imshow(img_code_barre_YCbCr[:, :, 0], cmap='gray')
117         plt.title("img canal Y")
118 
119         plt.figure(2)
120         plt.subplot(1, 2, 1)
121         plt.imshow(I_x, cmap='gray')
122         plt.title("grad x")
123         plt.subplot(1, 2, 2)
124         plt.imshow(I_y, cmap='gray')
125         plt.title("grad y")
126 
127         plt.figure(3)
128         plt.subplot(1, 2, 1)
129         plt.imshow(N_I_x, cmap='gray')
130         plt.title("normalisé grad x")
131         plt.subplot(1, 2, 2)
132         plt.imshow(N_I_y, cmap='gray')
133         plt.title("normalisé grad y")
134 
135         plt.figure(4)
136         plt.subplot(1, 3, 1)
137         plt.imshow(Txx, cmap='gray')
138         plt.title("Txx")
139         plt.subplot(1, 3, 2)
140         plt.imshow(Tyy, cmap='gray')
141         plt.title("Tyy")
142         plt.subplot(1, 3, 3)
143         plt.imshow(Txy, cmap='gray')
144         plt.title("Txy")
145 
146         plt.figure(5)
147         plt.subplot(1, 2, 1)
148         plt.imshow(D_res, cmap='gray')
149         plt.title("D")
150         plt.subplot(1, 2, 2)
151         plt.imshow(D_seuil, cmap='gray')
152         plt.title("D seuil")
153         plt.show()
154     return Blobs
155 # ~~~~ TEST DE L'EXTRACTION ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
156 # %%
157 img="img/barcode0.jpg"
158 # Pour le bruit, à regler à la main (2 est un bon point de départ)
159 sigma_bruit = 3
160 
161 # Pour le gradient, relativement faible pour trouver les vecteurs de
        transition correspondant aux barres
162 sigma_g = 2
```

```

# Pour le tenseur, relativement élevé pour trouver des clusters de vecteurs gradient
sigma_t = 100

seuil = 0.7
u=barcode_detection(img,sigma_g,sigma_t,seuil=0.5,sigma_bruit=2)
# u=barcode_detection("img/code_barre_prof.jpg",1,15,0.7,2,affichage=False)
# u=barcode_detection("img/barcode0.jpg",sigma_g=2,sigma_t=50,seuil=0.7, sigma_bruit=2)
# u=barcode_detection("img/b1.jpg",2,50,0.7,2,affichage=False)
for blob in u:
    print("Barycentre: ",end="")
    print(blob.barycentre)
    print("Axe: ",end="")
    print(blob.axis)
    print("Valeurs propres: ",end="")
    print(blob.valeurs_propres)
    o=blob.axis

print(f"Code exécuté en {round(time()-start_time,3)} s")
```

7.1.2 Classe Blob

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from math import floor, sqrt
from common import *

class Blob:
    def __init__(self, pixels=None,imsize=None):
        # Arguments indispensables
        self.pixels = pixels
        self.imsize=imsize # (h,w)
        # Calculés ultérieurement
        self.X = None
        self.Y = None
        self.barycentre = None
        self.valeurs_propres = None
        self.vecteurs_propres = None
        self.axis = None
        # Pour d'éventuelles applications futures
        self.area = None
        self.longueur = None
    def calc_XY(self):
        self.X,self.Y=self.pixels[:,0],self.pixels[:,1]
        return self.X,self.Y
    def calc_braycentre(self):
        self.barycentre=np.mean(self.pixels,axis=0)
        return self.barycentre
    def calc_vpropres(self):
        self.valeurs_propres,self.vecteurs_propres=np.linalg.eig(np.cov(self.X,self.Y))
        return self.valeurs_propres,self.vecteurs_propres

```

```
32      def calc_area(self): # pour éventuellement appliquer un critère de sé
    lection surla taille du bousin
33          self.area=self.pixels.size # not the real area, but a good measure
    of how large it is
34          return self.area
35
36      def calc_all(self):
37          try:
38              self.calc_XY()
39              self.calc_braycentre()
40              self.calc_vpropres()
41              return True
42          except Exception as e:
43              print(e)
44              return False
45
46      def calc_axis_0(self):
47          self.calc_all()
48          if self.axis==None:
49              if self.valeurs_propres[0]<self.valeurs_propres[1]:
50                  p1 = floor(self.barycentre[1]+self.valeurs_propres[0]/2*
    self.vecteurs_propres[0][0]), floor(self.barycentre[0]+self.
    valeurs_propres[0]/2*self.vecteurs_propres[0][1])
51                  p2 = floor(self.barycentre[1]-self.valeurs_propres[0]/2*
    self.vecteurs_propres[0][0]), floor(self.barycentre[0]-self.
    valeurs_propres[0]/2*self.vecteurs_propres[0][1])
52                  self.axis=[p1,p2]
53              else:
54                  p1 = floor(self.barycentre[1]+self.valeurs_propres[1]/2*
    self.vecteurs_propres[1][0]), floor(self.barycentre[0]+self.
    valeurs_propres[1]/2*self.vecteurs_propres[1][1])
55                  p2 = floor(self.barycentre[1]-self.valeurs_propres[1]/2*
    self.vecteurs_propres[1][0]), floor(self.barycentre[0]-self.
    valeurs_propres[1]/2*self.vecteurs_propres[1][1])
56                  self.axis=[p1,p2]
57          return self.axis
58
59      def calc_axis_extr(self):
60          # méthode des points extrêmes
61          self.calc_all()
62          if self.axis==None:
63              max_l=max(self.imsize)
64              if self.valeurs_propres[0]<self.valeurs_propres[1]:
65                  vecteurs_propres_norm=self.vecteurs_propres[0]/np.linalg.
    norm(self.vecteurs_propres[0])
66                  p1 = floor(self.barycentre[1]+max_l*vecteurs_propres_norm
    [0]), floor(self.barycentre[0]+max_l*vecteurs_propres_norm[1])
67                  p2 = floor(self.barycentre[1]-max_l*vecteurs_propres_norm
    [0]), floor(self.barycentre[0]-max_l*vecteurs_propres_norm[1])
68                  p1=bornage(self.imsize[1],self.imsize[0],p1)
69                  p2=bornage(self.imsize[1],self.imsize[0],p2)
70                  self.axis=[p1,p2]
71              else:
72                  vecteurs_propres_norm=self.vecteurs_propres[1]/np.linalg.
    norm(self.vecteurs_propres[1])
73                  p1 = floor(self.barycentre[1]+max_l*vecteurs_propres_norm
    [0]), floor(self.barycentre[0]+max_l*vecteurs_propres_norm[1])
```

```
74                  p2 = floor(self.barycentre[1]-max_l*vecteurs_propres_norm
    [0]), floor(self.barycentre[0]-max_l*vecteurs_propres_norm[1])
75                  p1=bornage(self.imsize[1],self.imsize[0],p1)
76                  p2=bornage(self.imsize[1],self.imsize[0],p2)
77                  self.axis=[p1,p2]
78          return self.axis
79
80      def calc_axis_ray(self,len_adjust=6):
81          # len adjust: facteur d'ajustement pour la longueur finale de l'axe
    . initialement à 1.
82          self.calc_all()
83          if self.valeurs_propres[0]>self.valeurs_propres[1]: #on détecte la
    valeur propre la plus grande
84              imax=0
85          else:
86              imax=1
87          axe=np.abs(self.vecteurs_propres[:,(imax+1)%2]) #on extrait le
    vecteur propre correspondant à la valeur minimale
88          # abs pour etre sur d'extraire le bon angle
89          axe_norm=axe/np.linalg.norm(axe,2)
90          alpha=np.arccos(axe_norm[0]) # on extrait l'angle avec l'
    horizontale
91          r=len_adjust*np.sqrt(self.valeurs_propres[imax])
92          self.axis = ray_center(self.barycentre[::-1],r , alpha)
93          return self.axis
94
95      def draw_random_rays(self,length,n):
96          self.calc_all()
97          return [random_ray_center_2(self.imsize[0], self.imsize[1], length,
     self.barycentre) for i in range(n)]
```

7.1.3 Fonctions utilitaires

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  def bornage(h, w, p): # à voir si une accélération est possible
5      p=list(p)
6      if p[0] < 0:
7          p[0] = 0
8      if p[0] > h:
9          p[0] = h-1
10     if p[1] < 0:
11         p[1] = 0
12     if p[1] > w:
13         p[1] = w-1
14     return p
15
16 # ~~~~ Fonctions tracé de rayons ~~~~~~~~~~~~~~~~~~~~~~~~~~~~
17 # %%
18 # ajouter des paramètres pour avoir un tirage gaussien (ou autre)
19
20 def ray_center(center,length,angle):
21     """
22     Parameters
23     ----------
```

```
    center : np.array([x0,y0])
    length : float
    angle : float
    crée un rayon de centre center, de longueur length et d'angle alpha (par rapport à l'horizontale)
    """
    offset = np.array([np.cos(angle), np.sin(angle)])*length/2
    x1 = center+offset
    x2 = center-offset
    return np.array([x1,x2])

def random_ray_center(h, w, length):
    # méthode: centre, angle, longueur
    angle = np.random.uniform(0, 2*np.pi)
    r = length/2
    center = np.array([np.random.randint(0, h), np.random.randint(0, w)])
    offset = np.array([np.cos(angle), np.sin(angle)])*r
    x1 = center+offset
    x2 = center-offset
    return np.int32([bornage(h, w, x1), bornage(h, w, x2)])

def random_ray_center_2(h, w, length,center):
    # méthode: centre, angle, longueur
    angle = np.random.uniform(0, 2*np.pi)
    r = length/2
    offset = np.array([np.cos(angle), np.sin(angle)])*r
    x1 = center+offset
    x2 = center-offset
    return np.int32([bornage(h, w, x1), bornage(h, w, x2)])

def random_ray(h, w, length):
    # méthode: extrémité1, angle, longueur
    angle = np.random.uniform(0, 2*np.pi)

    x1 = np.array([np.random.randint(0, h), np.random.randint(0, w)])
    offset = np.array([np.cos(angle), np.sin(angle)])*length
    x2 = x1+offset

    return np.int32([bornage(h, w, x1), bornage(h, w, x2)])

def plot_ray(ray):
    # for debugging
    plt.figure()
    plt.plot(ray[:,0],ray[:,1])
    k=5
    plt.plot(k * np.array([1, 1, -1, -1, 1]), k * np.array([1, -1, -1, 1, 1]))
    plt.grid()
    return

def get_angle(ray):
    ray=np.array(ray)
    print(ray)
    axe=np.abs(ray[:,1]-ray[:,0]) #abs pour être sûr d'extraire l'angle avec l'horizontale
    axe_norm=axe/np.linalg.norm(axe,2)
    alpha=np.arccos(axe_norm[0]) # on extrait l'angle avec l'horizontale
```

```
78      return alpha
```

7.2 Phase 2 : Lecture du code barre

```
1  def otsu(img, bins=255):
2    luminance = None
3
4    # Si l'image est en couleur (3 dimensions)
5    if len(img.shape) == 3 and img.shape[2] > 1:
6      # Calcul de la luminance
7      luminance = np.array([[(img[i][j][0] + img[i][j][1] + img[i][j][2])//3
       for j in range(img.shape[1])] for i in range(img.shape[0])])
8      luminance = luminance.ravel()
9    else:
10     luminance = img.ravel()
11
12   # Création de l'histogramme
13   histogram, bin_edges = np.histogram(luminance.ravel(), range=(0, 255),
       bins=bins)
14
15   # Moyenner l'histogramme
16   histogram = histogram/sum(histogram)
17
18   # Création d'un dico pour associer chaque valeur de luminance à sa fré
       quence
19   histogram_dic = {int(bin_edges[i]): histogram[i] for i in range(len(
       histogram))}
20
21   # Initialisation des moyennes et poids initiaux
22   n = len(histogram_dic)
23   sumB = 0
24   wB = 0
25   maximum = 0.0
26   sum1 = sum(i * histogram_dic[i] for i in range(n))
27   total = sum(histogram_dic.values())
28   level = 0
29   for k in range(1, n):
30     wF = total - wB
31     if wB > 0 and wF > 0:
32       mF = (sum1 - sumB) / wF
33       val = wB * wF * (sumB / wB - mF) * (sumB / wB - mF)
34
35       if val >= maximum:
36         level = k
37         maximum = val
38
39     wB += histogram_dic[k]
40     sumB += (k-1) * histogram_dic[k] # A vérifier si c'est k ou k-1
41
42   return level
43
44 def distance(x1,y1,x2,y2):
45   """ Calcule la distance entre deux points """
46   return np.sqrt((x2-x1)**2+(y2-y1)**2)
47
48 def echantillonnage(x1,y1,x2,y2,Nb_points):
```

```
49    """On échantillone sur le segment (x1,y1)->(x2,y2)"""
50    """On va choisir le plus proche voisin"""
51    X=np.around(np.linspace(np.floor(x1),np.ceil(x2),Nb_points)).astype(int)
52    Y=np.around(np.linspace(np.floor(y1),np.ceil(y2),Nb_points)).astype(int)
53    return X,Y
54
55
56  def find_lim(x1,y1,x2,y2,img,seuil):
57      """Récupération des points de départ et d'arrivée pour le segment 2"""
58      X,Y=echantillonnage(x1,y1,x2,y2,np.ceil(distance(x1, y1, x2, y2)).
        astype(int))
59      valeurs_img=np.zeros((len(X), 1))
60
61      for i in range(0,len(X)):
62          valeurs_img[i]=(img[Y[i], X[i]]) >= seuil
63      i1=0
64      i2=0
65
66      for i in range(0,len(valeurs_img)):
67          if valeurs_img[i]==0:
68              i1=i
69              break
70      for i in range(len(valeurs_img)-1,1,-1):
71          if valeurs_img[i]==0:
72              i2=i
73              break
74      return X[i1],Y[i1],X[i2],Y[i2] # xd,yd,xa,ya
75
76
77  def find_u(xd,yd,xa,ya,img,seuil):
78      """On va prendre le multiple u et le signal binaire à analyser"""
79      nb_p=np.ceil(distance(xd,yd,xa,ya)).astype(int) # Nombre de points L'
80      Nb_points=0 # u * 95
81      u=0
82      while (Nb_points<=nb_p): # On cherche le plus petit u tq u * 95 > nb_p
83          Nb_points+=95
84          u+=1
85
86      X,Y=echantillonnage(xd,yd,xa,ya,Nb_points)
87      valeurs_img=np.zeros((len(X), 1))
88
89      for i in range(0,len(X)):
90          valeurs_img[i]=(img[Y[i], X[i]]) <= seuil
91
92
93      return valeurs_img,u #Echantillonnage et binarisation
94
95
96  def separate(segment_seuille,u):
97      L=np.zeros((12,u*7))
98      start=3*u # Détermine les "gardes" (offset)
99      for i in range(0,12):
100         if (i==6):
101             start=start+5*u
102         segment_seuille_temp = segment_seuille[start+i*7*u : start+(i+1)*7*
        u]
103
```

```
104         for j in range(len(L[i])):
105             L[i,j] = segment_seuille_temp[j,0]
106 
107     return L
108 
109 def norme_binaire(liste_binaire,chaine_binaire,u):
110     sum=0
111     for k in range(0,7):
112         for i in range(0,u):
113             sum+=(liste_binaire[k*u+i]!=int(chaine_binaire[k]))
114     return sum
115 
116 def compare(region_chiffres_bin,L_the,u):
117     normes_codes=len(region_chiffres_bin[0])
118     decode=np.zeros((12))
119     r=""
120     for i in range(0,6):
121         r_b=r
122         for j in range(0,len(L_the[0])):
123             if (normes_codes>=norme_binaire(region_chiffres_bin[i],L_the
    [0][j],u)):
124                 normes_codes=norme_binaire(region_chiffres_bin[i],L_the[0][
    j],u)
125                 decode[i]=int(j)
126                 r=r_b+"A"
127             if (normes_codes>=norme_binaire(region_chiffres_bin[i],L_the
    [1][j],u)):
128                 normes_codes=norme_binaire(region_chiffres_bin[i],L_the[1][
    j],u)
129                 decode[i]=int(j)
130                 r=r_b+"B"
131         normes_codes=len(region_chiffres_bin[0])
132     for i in range(6,len(region_chiffres_bin)):
133         for j in range(0,len(L_the[0])):
134             if (normes_codes>=norme_binaire(region_chiffres_bin[i],L_the
    [2][j],u)):
135                 normes_codes=norme_binaire(region_chiffres_bin[i],L_the[2][
    j],u)
136                 decode[i]=int(j)
137         normes_codes=len(region_chiffres_bin[0])
138     return decode,r
139 
140 
141 def first_one(decode,r,tab):
142     """ Retourne le premier chiffre """
143     for i in range(0,len(tab)):
144         if (r==tab[i]):
145             return i
146     return None
147 
148 def clef_controle(decode):
149     # Complément à 10 du dernier chiffre du code barre
150     complement = (10 - decode[-1]) % 10
151 
152     L = [1, 3, 1, 3, 1, 3, 1, 3, 1, 3, 1, 3]
153     clef = np.sum(decode[0:-1] * L) % 10
154     return clef == complement
```

```
155
156 def phase1(x, y, img, seuil):
157     starttime = time.perf_counter()
158     x1 = x[0]
159     y1 = y[0]
160     x2 = x[1]
161     y2 = y[1]
162     codage_chiffres_bin = [["0001101", "0011001", "0010011", "0111101", "
    0100011", "0110001", "0101111", "0111011", "0110111", "0001011"],
163                            ["0100111", "0110011", "0011011", "0100001", "
    0011101", "0111001", "0000101", "0010001", "0001001", "0010111"],
164                            ["1110010", "1100110", "1101100", "1000010", "
    1011100", "1001110", "1010000", "1000100", "1001000", "1110100"]]
165     codage_premier_chiffre = ["AAAAAA","AABABB","AABBAB","AABBBA","ABAABB",
    "ABBAAB","ABBBAA","ABABAB","ABABBA","ABBABA"]
166     Nb=np.ceil(distance(x1, y1, x2, y2)).astype(int) # Nombre de points
167
168     # Binarisation
169     xd,yd,xa,ya = find_lim(x1,y1,x2,y2,img,seuil)
170
171     # Echantillonage + Binarisation de l'image après seuillage
172     segment_seuillage, u = find_u(xd,yd,xa,ya,img,seuil)
173     regions_chiffres_bin = separate(segment_seuillage,u)
174
175     # Décodage des regions binaires
176     regions_chiffres, sequence_AB = compare(regions_chiffres_bin,
    codage_chiffres_bin,u)
177
178     # Ajout du premier chiffre
179     premier_chiffre = first_one(regions_chiffres,sequence_AB,
    codage_premier_chiffre)
180
181     if (premier_chiffre == None):
182         return None
183
184     code_barre = np.append(premier_chiffre, regions_chiffres)
185
186     # Vérification de la clé de contrôle
187     if clef_controle(code_barre):
188         print("Code barre valide : ", code_barre)
189         endtime = time.perf_counter() - starttime
190         print("Temps d'exécution : ", endtime)
191         return code_barre
192     else:
193         print("Code barre invalide : ", code_barre)
194         return None
```

7.3 Main

```
1 from phase1 import *
2 from barcode_detection import *
3 start=time()
4 # Détection des barres
5 img_name = 'img/barcode0.jpg'
6
7 # Lecture du code barre
```

```
img = cv2.imread(img_name, cv2.IMREAD_GRAYSCALE)

# Seuillage avec la méthode d'Otsu
seuil = otsu(img)

# Extraction des points d'intérêt
Blobs = barcode_detection(img_name, sigma_g=2, sigma_t=50, seuil=0.7,
    sigma_bruit=2)
# Lecture à partir des points trouvés
for blob in Blobs:
    # altérer la méthode de calcul de l'axe si besoin (blob.calc_axis_...)
    # blob.calc_axis_ray(len_adjust=5)
    x = blob.axis[:,0].astype(int).tolist()[::-1]
    y = blob.axis[:,1].astype(int).tolist()[::-1]
    print("Axe: ", end="")
    print(blob.axis[0], end=""); print(blob.axis[1])
    res=phase1(x, y, img, seuil)

duration=time()-start
print("Temps d'exécution global: {}s".format(round(duration,2)))
```