Filière Télécommunications 2ème année

Rapport de projet

Projet Image | Lecture de code-barres par traitement d'images numériques

Auteurs :
Paul Gressier
Louis Gallot-Lavallée

15 janvier 2017

Table des matières

1 Introduction

Aujourd'hui la plupart des marchandises sont dotées d'un identificateur numérique et graphique qui prend la forme d'un code barres. Les codes barres sont représentés par une série de lignes parallèles d'épaisseur variable. Ils sont utilisés dans de nombreux domaines, on peut citer par exemple le suivi des articles dans les magasins à rayons, le suivi des envois postaux ou encore par exemple le suivi des bagages sur les lignes aériennes. Nous allons dans ce projet nous focaliser sur les codes barres unidimensionnels du type EAN 13, qui sont les plus utilisés en Europe. Étant massivement utilisé, il est très intéressant de comprendre comment fonctionnent les codes barres et de mettre au point nos propres fonctions de décodage.
Ainsi nous allons dans un premier temps étudier le principe théorique des codes barres EAN 13, dans un second temps nous allons nous pencher sur les fonctions de décodage d'un code barre, nous allons ensuite mettre au point une technique pour gérer les codes barres orientés et détecter automatiquement la zone d'intérêt et enfin nous analyserons les performances de notre système.

2 Principe théorique des codes barres

Le code EAN 13 que nous allons étudier est le code barre de référence en Europe. Analysons ses grands principes.

2.1 Organisation spatiale d'un code barre EAN 13

Le code barre EAN 13 est composé de 13 chiffres dont le premier chiffre se déduit des six suivants. Ce code prend la forme de ligne parallèle d'épaisseur variable noir et blanche. Le code est décomposé en 5 parties distinctes visible figure 2.1.

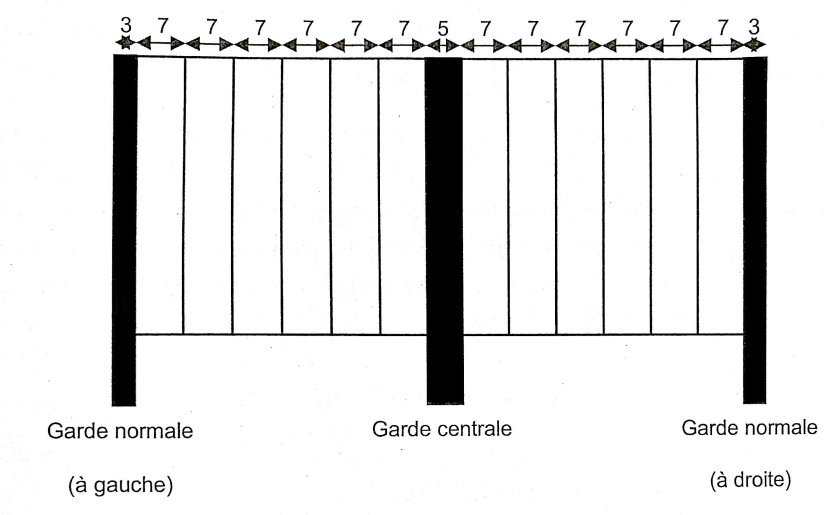

FIGURE 2.1 – organisation spatiale d'un code barre EAN 13

Sur la figure 2.1 on peut voir une première région, dite de garde de largeur égale à 3 fois l'unité de base, ensuite une région qui code 6 chiffres avec pour chaque chiffre une largeur égale à 7 fois l'unité de base. Une autre région de garde est situé au milieu du code, c'est la garde centrale, juste après celle-ci, il y a de nouveau une zone qui code 6 chiffres et enfin pour finir le code une garde similaire à la première garde.

2.2 Codage des chiffres

Les chiffres présents dans les deux zones de 6 chiffres sont codés par un code graphique binaire de 7 unités de base. Un chiffre est donc codé par une suite de 7 barres blanches (B) et Noir (N). De plus chaque chiffre appartient à une famille d'éléments A,B ou C. Le codage pour chaque chiffre de 1 à 9 suivants la famille A,B ou C est visible figure 2.2.

	Elément A	Elément B	Elément C
0	BBBNNBN	BNBBNNN	NNNBBNB
1	BBNNBBN	BNNBBNN	NNBBNNB
2	BBNBBNN	BBNNBNN	NNBNNBB
3	BNNNNBN	BNBBBBN	NBBBBNB
4	BNBBBNN	BBNNNBN	NBNNNBB
5	BNNBBBN	BNNNBBN	NBBNNNB
6	BNBNNNN	BBBBNBN	NBNBBBB
7	BNNNBNN	BBNBBBN	NBBBNBB
8	BNNBNNN	BBBNBBN	NBBNBBB
9	BBBNBNN	BBNBNNN	NNNBNBB

FIGURE 2.2 – Table de codage des chiffres EAN 13

Concernant le premier chiffre qui est déduit des 6 suivants, on le décoda grâce aux familles vues précédemment comme on peut le voir figure **??**

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

FIGURE 2.3 – Table de codage du premier chiffre EAN 13

On a donc nos 13 chiffres qui composent le code EAN 13.

Il existe une clé de contrôle pour le code barre EAN 13. Cette clé de contrôle sommaire a pour but de vérifier le décodage. Elle est localisée au niveau du dernier chiffre. En effet, le complément à dix du dernier chiffre doit être égal au chiffre des unités du nombre calculé en additionnant la somme des chiffres de rangs impairs à trois fois la somme des chiffres de rang pairs.

3 Décodage d'un code barre

Dans un premier temps nous avons implémenté une technique simple nécessitant un code-barres ayant les bords parallèles à ceux de l'image. De plus un opérateur sera nécessaire pour sélectionner une zone d'intérêt.

3.1 Zone d'intérêt

Pour procéder à l'analyse d'un code-barres il est nécessaire de trouver la zone d'intérêt contenant le code-barres. Pour cela un opérateur trace une ligne sur l'image et le programme travaillera à partir de ces données. L'opérateur étant un humain il lui est facile de trouver la zone d'intérêt cependant il lui est difficile de définir la zone au pixel près. Or l'algorithme demandant une zone exacte, ce sera donc une étape de pré-traitement, redéfinir la zone d'intérêt pour qu'elle soit exacte au pixel près.

Pour la suite nous définirons $x_{min} = min(x_1, x_2)$, $x_{max} = max(x_1, x_2)$, $y_{min} = min(y_1, y_2)$ et $y_{max} = max(y_1, y_2)$ avec les valeurs entrées par l'opérateur comme indiqué sur la FIGURE 3.1

FIGURE 3.1 – Saisie opérateur

La zone est tout d'abord définie en hauteur. L'algorithme effectue une comparaison simple entre la ligne sélectionnée par l'opérateur et la ligne supérieure à l'aide de la formule suivante :

$$1 - \epsilon < \left| \frac{\sum_{x=x_{min}}^{x_{max}} I(x, y)}{\sum_{x=x_{min}}^{x_{max}} I(x, y_{min})} \right| < 1 + \epsilon \tag{3.1}$$

Explication de la formule : une somme des valeurs des pixels sur la ligne testée est effectuée, une même somme est faite sur la ligne sélectionnée par l'opérateur. Tant que le rapport des sommes est inférieur à une différence de ϵ la ligne est comptée comme suffisamment similaire pour être prise en compte. Si le rapport diffère d'un seuil fixé à ϵ alors la ligne est rejetée et le programme s'arrête à cette ligne.
La valeur de ϵ détermine la sensibilité de la sélection.

Cette opération est effectuée vers le haut de l'image à partir de la ligne sélectionnée par l'opérateur et vers le bas de l'image. Sont ainsi trouvé les bornes supérieure et inférieure de la zone d'intérêt.

Par la suite la zone d'intérêt est ramenée a une signature monodimensionnel par une somme moyennée verticale.

L'intérêt d'effectuer ces tâches plutôt que de travailler seulement sur la sélection de l'opérateur est que certains cas pouvant poser problème sont éliminés. Par exemple si l'opérateur sélectionne une zone plus éclairée avec un peu de reflet, les pixels peuvent apparaître gris et poser problème pour la sélection du seuil plus tard.

Une fois la signature obtenue il est nécessaire de détecter les bornes droite et gauche du code-barres. Pour cela la signature est binarisée. Une fois binarisées les bornes correspondront à la 1ere bande noire et à la dernière bande noire. Le seuil permettant la binarisation est déterminé par la méthode d'Otsu.

La méthode d'Otsu permet de détecter automatiquement un seuil dans une image en niveau de gris. L'algorithme suppose qu'il n'existe que 2 types de pixels, les pixels d'arrières plans et les pixels de 1er plan. Cette méthode est particulièrement adaptée à notre cas, car en effet nos images sont majoritairement composées d'un fond uniforme et du code barre en noir.

3.2 Identification des chiffres

Comme présenté sur la FIGURE 2.2 nos chiffres sont encodés en 7 valeurs. Or il est très rare que les bandes fassent exactement 1 pixel de large. Il faut donc procéder à un changement de base. Comme expliqué dans la 1ere partie le code-barres est composé de 95 unités de bases. Il suffit donc de diviser la longueur de notre signature par 95 pour

obtenir le nombre de pixels qui représente une unité de base. Il est très rare que ce nombre soit entier. Il est donc important d'en tenir compte lors du changement de base. Nous avons implémenté ces fonctions dans *resize* et *resize*2 dans notre code MATLAB.

Une fois notre table de codage des chiffres changée de base, on effectue un calcul de corrélation entre les morceaux analysés et toutes les valeurs possibles dans la table de codage. Le numéro le plus probable est celui ayant la plus forte corrélation. Il est important de retenir la famille du code sélectionné, en effet il est nécessaire de s'en souvenir pour pouvoir décoder le 13ème chiffre comme indiqué en FIGURE 2.3.
Finalement il est nécessaire de vérifier que notre code décodé est valide à l'aide de la clef de contrôle comme expliqué dans la 1ère partie.

4 Gestion des codes barres orientés et décodage automatique

Nous avons mis en place les fonctions de décodage d'un code barre. Cependant, il y a des limitations sur nos fonctions. En effet, nous ne gérons pas les codes orientés, ni la détection automatique du code barre. Nos fonctions ne sont donc pas pratiques car des contraintes pour le décodage sont présentes.

4.1 Gestion des codes barres orientés

Les photos de codes barres ne sont pas toujours nivelées. En effet, il arrive régulièrement que le code barre soit orienté dans la photo. Une amélioration possible de nos fonctions est donc la prise en charge de cette orientation pour remettre droit la photo pour ensuite pouvoir utiliser nos fonctions déjà implémentées.

La première étape consiste en la détection de l'angle du code barre. Pour trouver l'angle, nous allons détecter les lignes du code barre. En effet, une des particularités du code barre est son grand nombre de barres parallèles. Nous allons nous servir de cette spécificité pour trouver l'angle. Pour ce faire, nous allons utiliser la transformée de Hough qui est une technique de reconnaissance de formes inventée en 1962 par Paul Hough.

La transformée de Hough consiste à changer l'espace de représentation des points d'intérêt. L'espace cartésien est ainsi remplacé par l'espace polaire (ρ, θ). Son principe découle de l'équation paramétrique d'une droite en coordonnées polaires :

$$\rho = xcos(\theta) + ysin(\theta) \tag{4.1}$$

Un point initial (x, y) forme ainsi une sinusoïde dans l'espace (ρ, θ). On accumule ensuite l'ensemble des sinusoïdes issues de la transformation de tous les points d'intérêts, ce qui forme des pics. Ces pics correspondent alors aux coordonnées polaires des lignes recherchées comme on peut le voir figure 4.1

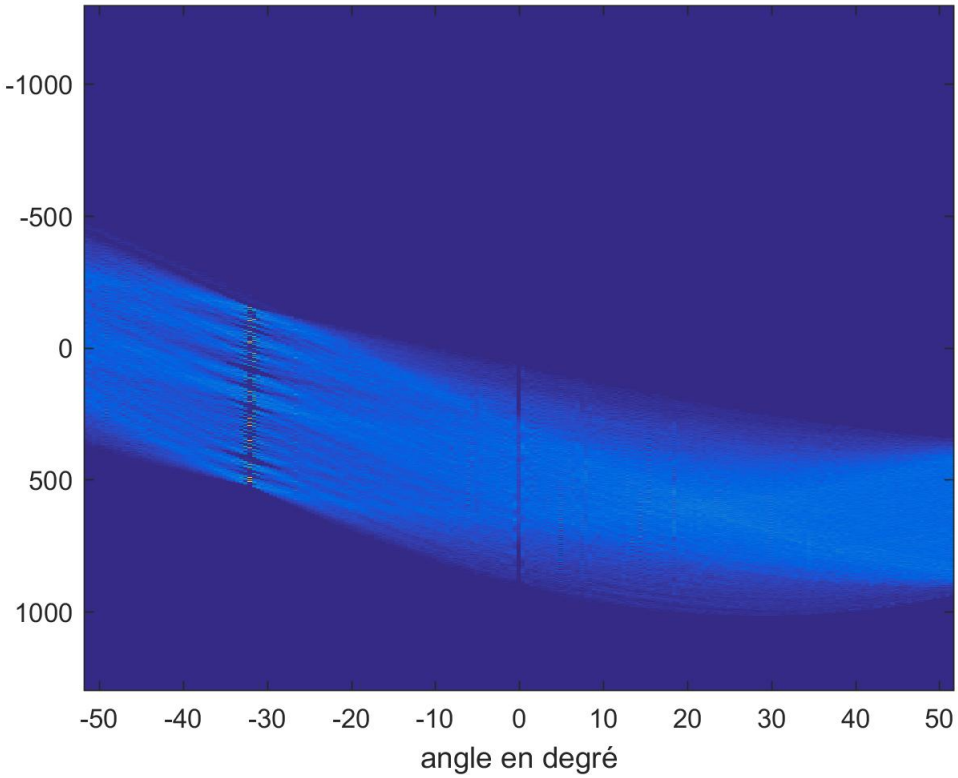

FIGURE 4.1 – Transformée de Hough

L'implémentation de la détection de l'angle est situé dans la fonction Matlab *detect_angle* et prend comme argument le code barre mais aussi x et y, les deux points utilisées pour la recherche de la zone utile. A noter que pour faciliter la tâche de la détection de l'angle nous réduisons le domaine de recherche à 40 dès le départ, en fonction de l'inclinaison de la ligne d'initialisation entre x et y.

Maintenant que nous avons obtenu l'angle de rotation à appliquer, il faut faire la rotation de l'image pour remettre droit le code barre et ainsi avoir la bonne orientation pour l'image. La matrice de rotation sera

$$\begin{pmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{pmatrix} \tag{4.2}$$

avec θ l'angle de rotation
Pour faciliter l'explication de la rotation nous allons dire que l'image 1 est l'image de départ qui est orienté et l'image 2 est l'image que l'on veut obtenir et pour lequel le code barre sera droit. Le principe de cette rotation est le suivant : nous devons tout d'abord détecter la taille de la nouvelle image. Pour ce faire nous allons sélectionner les 4 pixels

dans les corners pour l'image 1 et leur appliquer la matrice de rotation. Nous pouvons ainsi en déduire la nouvelle dimension de l'image qui va subir la rotation : l'image 2, on créer donc une matrice de cette taille. Nous devons ensuite "remplir" cette matrice. Pour ce faire, nous allons partir de cette matrice qui correspond à l'image 2 et allons appliquer la matrice de rotation inverse à tous ses points. Ensuite nous allons utiliser une fonction Matlab *interp*2 qui permet d'aller interpoler le bon pixel indiqué par la rotation inverse et le mettre dans notre nouvelle matrice. On obtient ainsi notre image 2 reconstruite à partir de l'image d'origine et de l'angle indiqué. Nous avons alors une image ou le code barre est bien droit.

La dernière étape de la rotation consiste à recalculer les points x et y qui eux aussi doivent subir une rotation. Pour rappel, ces deux points sont donnés par l'opérateur et permettent de localiser la zone d'intérêt. On va donc appliquer aux deux points x et y la matrice de rotation. On obtient alors nos nouveaux points qui vont servir pour le décodage explicité dans la partie précédente.

Une fois le rotation terminée et les nouveaux points x et y calculés, nous réutilisons toutes les fonctions décrites dans la partie précédente et décodons exactement le code barre de la même manière que précédemment.

On peut voir figure 4.2 un exemple de code barre orienté et figure 4.3 le résultat de notre rotation.

FIGURE 4.2 – Code barre orienté

FIGURE 4.3 – Code barre remis droit

4.2 Décodage automatique

Une des grandes limitations de notre système est qu'il y a besoin d'un opérateur qui clique sur 2 points du code barre pour aider à la détection de la zone d'intérêt. C'est pourquoi, pour finir notre lecteur de code barre, il est logique pour se rapprocher du fonctionnement réel et de réaliser la détection automatique de la zone d'intérêt et ainsi éviter qu'un opérateur sélectionne cette zone.

Pour ce faire, nous avons créé une fonction matlab *detect_zone*. Cette fonction matlab prend en entrée l'image ou il y a le code barre et renvoie les lignes qui sont censées entourer le code barre. Le but final est d'extraire des lignes, 2 points qui sont adaptés pour le décodage.
La première étape pour le décodage consiste à mettre en évidence le code barre. Pour ce faire on va chercher les bords des objets de l'image à l'aide de la fonction *edge*. Le résultat de cette fonction est visible figure 4.4

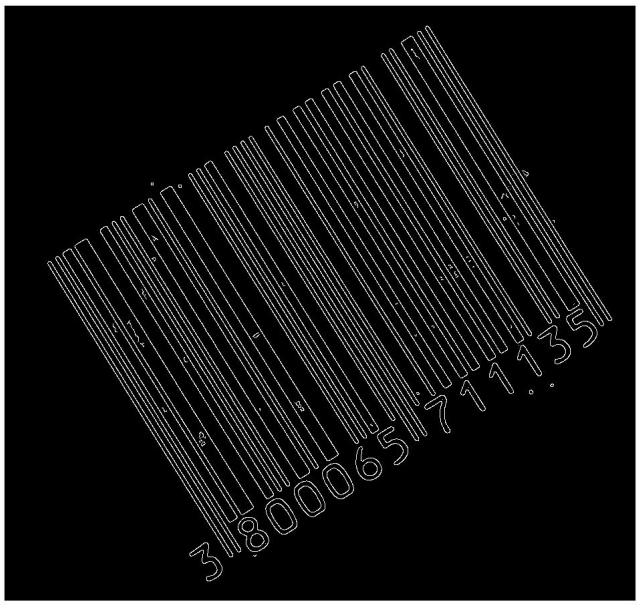

FIGURE 4.4 – Résultat de la fonction edge

Ensuite, pour "sélectionner" la zone du code barre on utilise la fonction *strel* qui construit un élément structurant, dans notre cas rectangle. On utilise ensuite la fonction *imclose* qui va ensuite "dilater" les éléments structurant et ainsi recréer une image visible figure 4.5

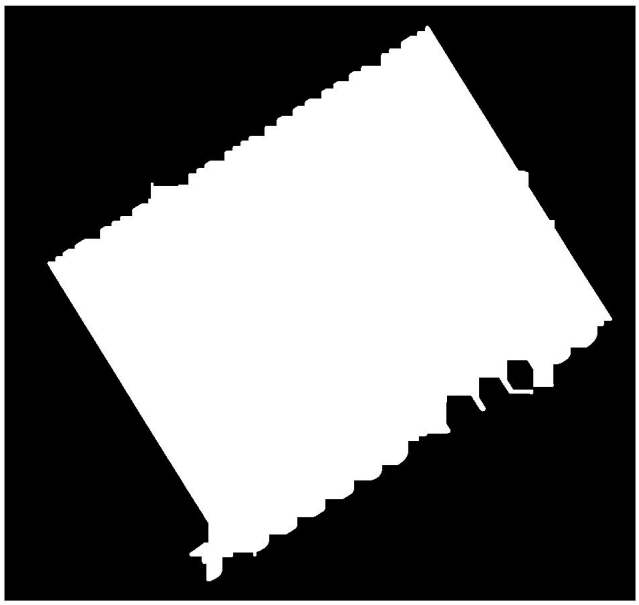

FIGURE 4.5 – Résultat de la fonction strel et imclose

Pour finir, nous allons de la même manière que pour détecter l'inclinaison du code barre, utiliser la transformée de Hough pour tracer les lignes visibles figure 4.6. Nous allons effectuer cette transformée sur le contour obtenu avec la fonction Matlab *imcontour*.

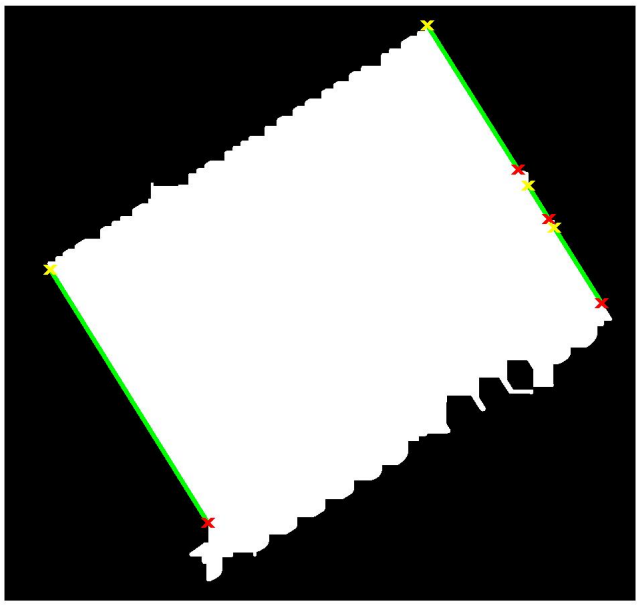

FIGURE 4.6 – Tracer des lignes selon la transformée de Hough

Il suffit ensuite de tester nos fonctions de décodage classique en prenant comme points de référence le milieu de chaque ligne. Si cela marche pas, on continue le processus en prenant un autre point et si le décodage à marché, on stop le processus.
Ce décodage automatique marche dans la plupart des cas ou notre décodage classique marchait, le problème majeur est le temps de calcul, en effet, ce programme teste tous les milieux des lignes, il est donc possible que selon la disposition des lignes, le temps de traitement soit relativement long.

5 Analyse des performances

Notre programme MATLAB fonctionne correctement dans la majorité des cas en détection manuelle et en détection automatique. Certains cas sont néanmoins aux limites des hypothèses posées pour le fonctionnement de notre code. Par exemple pour la FIGURE 5.1 il est nécessaire d'effectuer une transformation supplémentaire, en effet une simple rotation ne suffit pas pour ramener les bords du code-barres parallèle à l'image. Notre programme ne peut donc pas gérer ce genre de situation.

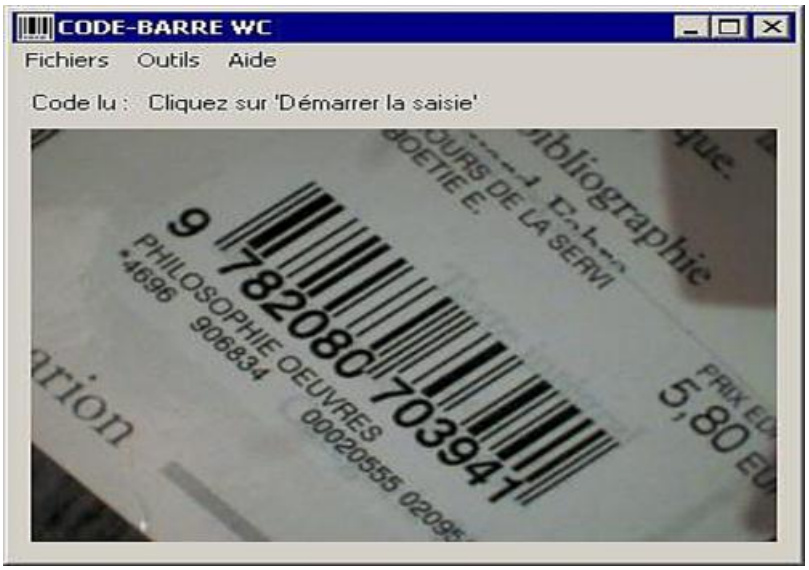

FIGURE 5.1 – Code-barres n ° 14

Des cas présentant des images binaires peuvent aussi poser problème. En effet la méthode d'Otsu suppose que les pixels sont répartis selon des gaussiennes or dans le cas d'une image binaire il n'y a plus que 2 classes de pixels : les tous noires et les tous blancs. A ce moment le seuil d'Otsu est biaisé. pour cela il est possible d'ajouter du bruit pour rendre l'image moins parfaite et correspondre à un cas plus "réel".

Nous supposons aussi dans le cas de la détection automatique que le fond est homo-

gène. Sinon cela poserait un problème lors de la détection automatique.

Lors de la détection automatique il se peut aussi que le programme détecte un code valide alors que celui-ci n'est pas représenté dans le code barre. Pour améliorer cet effet de bord nous pourrions implémenter un calcul total de tous les codes valides et sélectionner le code le plus redondant, au lieu de s'arrêter au 1er code valide trouvé.

Lors de la rotation automatique si le code est trop incliné la rotation peut-être effectuée dans le mauvais sens et le code se retrouve à l'envers. Implémenter une 2ème rotation de 180 en cas d'échec pourrait pallier a ce problème.

L'image doit posséder aussi une résolution standard approchant de la HD. Cela pour permettre la sélection du contour lors de la détection automatique qui est faite sur un nombre fixe de pixels. Dans la pratique cela ne pose pas de problème car le capteur ne change pas pour un détecteur précis de code-barres, il n'y a pas besoin de s'adapter à des changements d'objectif, ni a des résolutions différentes.

Dans les cas où le code-barres est lui-même faux, le programme réagit bien en retournant que le code est faux. C'est le cas du code présenté en FIGURE 5.2

FIGURE 5.2 – Code-barres n ° 15

6 Conclusion

Le but de ce projet a été de découvrir dans des cas concrets le fonctionnement des codes barres et de mettre en place un détecteur permettant de les décrypter.
Dans un premier temps nous avons implémenté pas à pas les fonctionnalités de bases nécessaires au décodeur avant de les regrouper pour obtenir un décodeur fonctionnel mais simpliste et ayant beaucoup d'hypothèses à vérifier pour fonctionner. Puis nous avons implémenté une fonctionnalité qui permet de réaliser une rotation en 2 dimensions afin d'obtenir un code barre droit. Pour finir, nous avons ajouté dans le projet l'auto-détection de la zone à analyser pour permettre d'utiliser notre décodeur sans opérateur, même si cela implique de nouvelle hypothèse de fonctionnement.
Néanmoins,les décodeurs actuels sont bien plus avancés et réalisent plus de fonctionnalités. Il nous serait possible d'améliorer notre projet en supprimant par exemple des hypothèses liées à notre détection automatique, de réduire le temps de calcul nécessaire au décodage, ou encore de gérer les déformations tridimensionnelles des images de code-barres.

6.1 Bilan organisationnel

Le travail a toujours été effectué a deux, sauf pour le rapport où nous nous sommes divisé les tâches. Nous avons surtout travaillé lors des séances mais également à cotés pour par exemple la détection automatique ainsi que pour le rapport.

Tâches réalisées et temps estimé :
- Délimitation de la zone d'intérêt (1h30)
- Estimation de la signature (2h)
- Création d'un imresize custom (2h)
- Identification des chiffres (3h)
- Gestion des codes barres orientés (3h)
- Détection automatique de la zone d'intérêt (4h)