INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

MATLAB

Rapport de projet :

Application de décodage de codes-barres

Auteurs :

METRAS Gaël, BARGES Tanguy,

JAMADI Nassime, GAUDARD Gaëtan

Année 2019 - 2020

Table des matières

INTRODUCTION

D'abord breveté en 1952 par deux étudiants américains, Norman Joseph Woodland et Bernard Silver, le code-barres est utilisé dès 1973 pour la mise au point du Code Universel des Produits (CUP). L'invention du CUP fait suite à une demande générale d'un comité de fabricants et de distributeurs aux États-Unis pour automatiser les caisses et la gestion des stocks. Le CUP permet l'identification numérique des produits. Il est encodé sous forme de code-barres et est composé, au début, de 12 chiffres. Il se diffuse un an plus tard en Europe sous le nom de EAN 13, qui est composé non plus de 12 chiffres mais de 13.

Aujourd'hui, les codes-barres sont ancrés dans notre quotidien et permettent de donner aux consommateurs de précieuses informations, notamment dans le secteur de l'agroalimentaire. En effet, il est aujourd'hui possible de connaître le taux de gras, de sucre, de sel, le nombre de calories, ainsi que les additifs utilisés dans la réalisation de denrées alimentaires. C'est ainsi que Yuka, application française cofondée en 2016 par trois français : François Martin, Benoît Martin et Julie Chapon, permet à ses utilisateurs de connaître la composition des aliments qu'ils achètent via le scan de leur code-barres grâce à leur smartphone. L'application donne ensuite au produit scanné une note sur 100 en fonction de plusieurs critères, tels que le taux de sucre, de gras, de fibres, de protéines, ou encore le nombre d'additifs utilisés. L'application utilisait la base de données d'Open Food Facts pour connaître la composition des produits scannés, avant de se construire sa propre base de données. Open Food Facts se présente comme "une base de données sur les produits alimentaires, faite par tout le monde, pour tout le monde". La note 0 indiquant un produit dangereux pour la santé en cas de consommation régulière et la note 100 un produit très sain pour son consommateur. Grâce à ses aspects pratiques, Yuka a connu un franc succès dès son lancement. En 2017, elle comptabilise 100 000 utilisateurs et en dénombre en Octobre 2019, 12 millions dont 10 millions en France et 2 millions dans différents pays. Sa diffusion, très rapide en France et maintenant dans le monde, montre un réel attrait des consommateurs pour connaître la composition des produits qu'ils achètent.

Conscient d'une méfiance grandissante des consommateurs envers la composition et la traçabilité des produits alimentaires issus de l'industrie agroalimentaire, une société de grande distribution nous a chargé de créer une application permettant aux consommateurs de décrypter aisément la composition de leurs produits. Après avoir étudié le fonctionnement d'un code barre et quelques méthodes pour le décoder, nous avons mis au point notre propre application de décodage avec Matlab. Enfin, nous avons analysé les coûts et avantages pour la société de mise en place d'une telle application auprès des consommateurs.

I. Fonctionnement des codes-barres

Les codes-barres rencontrés dans la grande distribution sont en réalité des CUP encodés numériquement sous forme de codes-barres et composés de 8, 12 ou 13 chiffres en fonction des pays et des normes. En France, nous utilisons principalement un CUP de 13 chiffres, dénommé EAN 13. Ce code apporte des informations sur le produit. Les deux ou trois premiers chiffres correspondent au pays de provenance. Les 4 ou 5 suivants correspondent au numéro de membre de l'entreprise participant au système EAN. Les 4 ou 5 suivants correspondent au numéro d'article. Le treizième est une clé de contrôle calculée en fonction des douze précédents.

Figure 1 : Signification des chiffres du code-barres. Ref[3]

Notre application se concentrera uniquement sur le décryptage des EAN13 car ce sont les plus répandus dans l'industrie agroalimentaire. La lecture du code-barres par un scanner permet de retrouver les 13 chiffres propres au produit scanné. Sur le site e-commerce-nation (Ref[9]), on apprend qu'aujourd'hui la lecture optique du code-barres n'enregistre pas seulement la transaction. Elle déclenche aussi une incroyable cascade d'opérations, depuis l'enregistrement des données d'achats des clients jusqu'à la gestion des stocks et les commandes de réapprovisionnement automatique.

Pour le décodage des EAN13, on peut exploiter les propriétés et normes suivantes :

- La forme générale :

Les codes EAN13 sont constitués de 30 barres noires et 29 blanches.

- Les motifs de garde :

Un motif de garde est répété au centre (noté M dans la suite) à gauche (noté S) et à droite (noté E) du code-barres. Les barres de ces motifs sont souvent plus grandes que les autres, mais ce n'est pas systématique. La largeur de ces barres est une dimension de référence : c'est la largeur élémentaire. Le motif centrale M est constitué de 5 largeurs élémentaires (blanc-noir-blanc-noir-blanc) tandis que les motifs S et E aux extrémités sont constitués de seulement 3 largeurs élémentaires (noir-blanc-noir)

- L'encodage des chiffres :

Les chiffres sont encodés en binaire avec 3 tables différentes, notées L, G et R dans la suite. Un chiffre est composé de 7 largeurs élémentaires, répartis dans 4 barres (dans l'ordre noir-blanc-noir-blanc pour la table R, et dans l'ordre blanc-noir-blanc-noir pour les tables L et G). Une largeur élémentaire noir correspond à un bit de valeur 1, tandis qu'une largeur élémentaire blanche correspond à un bit de valeur 0.

On peut séparer les 13 chiffres du code en 3 groupes : le premier chiffre qui n'est pas codé avec la largeur des barres, les 6 chiffres entre S et M qui sont codées avec les tables L et G, et enfin les 6 derniers chiffres, codés avec la table R. Le premier chiffre détermine quelles tables sont utilisées (entre L et G) pour encoder les 6 chiffres suivants (entre S et M) :

Premier chiffre	Encodage chiffres de gauche
0	LLLLLL
1	LLGLGG
2	LLGGLG
3	LLGGGL
4	LGLLGG
5	LGGLLG
6	LGGGLL
7	LGLGLG
8	LGLGGL
9	LLGLGL

Figure 2 : Tables de décodage utilisées en fonction du premier chiffre. Ref[4]

Le premier chiffre n'étant pas codé, il faut déterminer avec quelles tables sont encodés les 6 chiffres entre S et M pour retrouver la valeur de ce premier chiffre.

Les tables de décodage sont présentées ci-dessous :

a)

Chiffre	L	G	R
0	0001101	0100111	1110010
1	0011001	0110011	1100110
2	0010011	0011011	1101100
3	0111101	0100001	1000010
4	0100011	0011101	1011100
5	0110001	0111001	1001110
6	0101111	0000101	1010000
7	0111011	0010001	1000100
8	0110111	0001001	1001000
9	0001011	0010111	1110100

b)

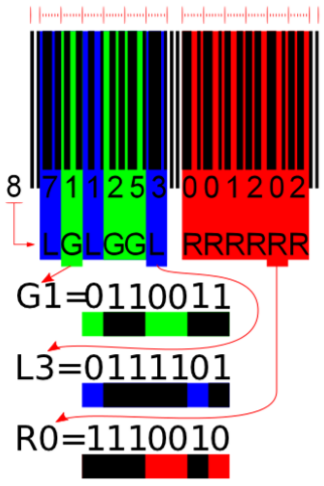

Figure 3 : a) Tables permettant de déchiffrer les barres.
b) Schéma permettant de comprendre comment utiliser la table. Ref[4]

On constate ici que les tables G et R sont symétriques. Donc si on lit un chiffre de type R à l'envers, on aura un code de type G et inversement. En revanche, si on lit un chiffre de type L à l'envers, on ne trouvera pas de correspondance et le code sera invalide. Cela permet de savoir si le code est à l'endroit ou pas.

- Le chiffre de contrôle :

Le 13ème chiffre (codé avec la table R) est un chiffre de contrôle. Il existe une relation entre ce chiffre et les 12 autres (d'indice allant de 1 à 12) :

$$(10 - (3 \times Paire + Impaire)\%10)\%10$$
$$\Leftrightarrow \big(-(3 \times Paire + Impaire)\big)\%10$$

Où %10 signifie le reste de la division euclidienne par 10. $Impaire$ est la somme des chiffres d'indice impaire $Paire$ est la somme des chiffres d'indice paire (voir la figure 4).

Pour mieux comprendre cette relation, on se propose de déterminer le chiffre de contrôle des 12 chiffres suivant :

Figure 4 : Code d'exemple pour le calcul du chiffre de contrôle. Inspiré de Ref[2]

On a :

$Impaire = 2 + 0 + 8 + 6 + 4 + 2 = 22$

$Paire = 1 + 9 + 7 + 5 + 3 + 1 = 26$

$$(-(3 \times Paire + Impaire))\%10 = \big(-(3 \times 26 + 22)\big)\%10 = (100)\%10 = 0$$

0 est donc le chiffre de contrôle de ce code.

II. Phase de prétraitement, les bases du traitement d'image

Après de nombreuses recherches documentaires concernant le décryptage des codes-barres, il apparaît plusieurs étapes de traitement d'image qui reviennent régulièrement. Il faut d'abord transformer l'image en niveau de gris, éventuellement la filtrer, puis la binariser avec un niveau de seuillage. Ces étapes sont résumées dans le schéma suivant :

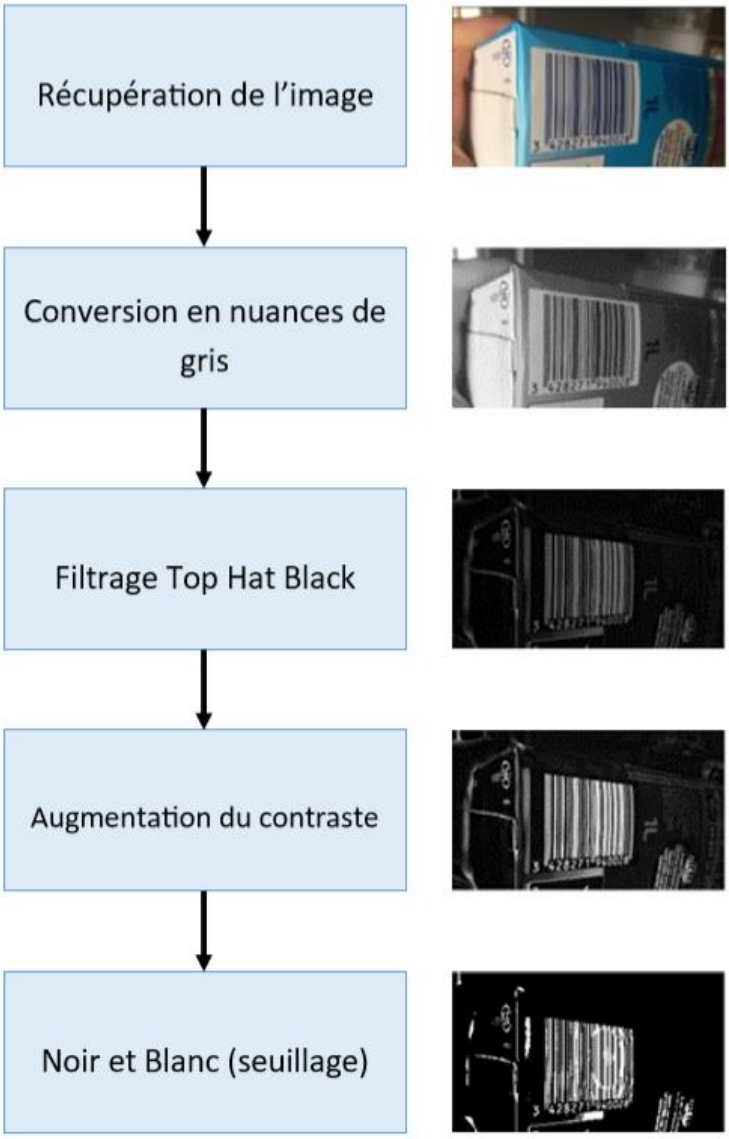

Figure 5 : Protocole de prétraitement de l'image

II.1. Récupération de l'image et niveaux de gris

Ces étapes se font simplement avec les fonctions imread() et rgb2gray() de Matlab. La première nous donne une matrice de pixels de la taille de l'image. Chaque pixel est constitué de 3 valeurs correspondant à l'intensité (entre 0 et 255) du vert, du rouge et du bleu. La seconde fonction réalise une moyenne pondérée de ces trois valeurs pour obtenir un niveau de gris entre 0 (noir) et 255 (blanc). Les coefficients de pondération prennent en compte les différences de sensibilité de l'œil humain en fonction de la couleur observée : l'intensité lumineuse perçue est différente de la simple somme des intensités des différentes couleurs.

II.2. Filtrage de l'image et augmentation du contraste

Le filtre Top-Hat-Black est un filtre morphologique qui permet de mettre en évidence certains éléments noirs de l'image (les barres du code-barres). Pour cela, on utilise un élément structurant (créé avec la fonction strel()). Cet élément doit être plus gros que la taille des barres du code-barres. C'est une sorte de référence qui va permettre de créer une image sans les objets qu'on désire mettre en évidence. Puis, si on fait la différence entre l'image de départ et l'image sans les objets, ces derniers seront mis en exergue sur un fond relativement homogène. Cette étape, réalisée avec la fonction imbothat(image, élément structurant), permet de faciliter le seuillage.

On peut également, augmenter le contraste de l'image en niveau de gris grâce à imadjust(). L'augmentation du contraste se traduit par un étalage des couleurs sur toute la plage disponible. Les histogrammes suivants qui représentent le nombre de pixel en fonction de leur couleur, illustre cette augmentation de contraste :

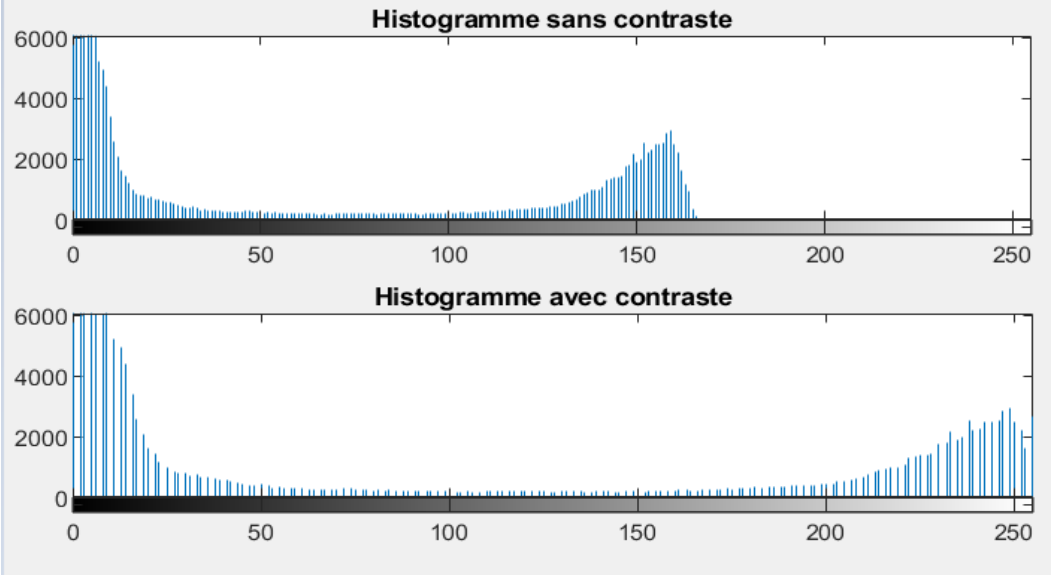

Figure 6 : Histogrammes mettant en évidence le rehaussement du contraste

L'augmentation du contraste facilite son seuillage.

II.3. Conversion en noir et blanc

La binarisation de l'image semble assez simple à réaliser, mais c'est en réalité une étape décisive, puisqu'on enlève une grande quantité d'information. Le but est d'obtenir une image en noir et blanc. On choisit donc une valeur de seuil (entre 0 et 255). Les pixels au-dessus du seuil seront blanc, et ceux en dessous seront noir. Cependant, il existe différents algorithmes qui permettent de déterminer la valeur du seuil. La méthode d'Otsu est la plus répandue : il s'agit de faire une moyenne des valeurs des pixels. C'est une méthode globale puisqu'on considère tous les pixels de l'image pour calculer le seuil. On peut aussi utiliser un seuillage adaptatif. C'est à dire que la valeur du seuil ne sera pas la même dans toutes les zones de l'image. Les seuils sont calculés grâce à une moyenne locale. Ces deux méthodes de seuillage sont implémentées dans la fonction imbinarize(image, *méthode*), où *méthode* peut être *global* ou *adaptive*. Après quelques tests sur différents codes-barres, nous avons choisi la méthode globale, mais ce n'est pas forcément la plus adaptée dans certaines situations où des parties différentes du code-barres sont éclairées différemment.

III. Extraction du code-barres

On dispose à ce stade d'une image binarisée. On cherche maintenant à extraire le code-barres de cette image, c'est à dire trouver sa localisation et son orientation. Pour ce faire, nous avons mis au point quelques algorithmes permettant de filtrer certaines régions de l'image selon différents critères. Ces étapes sont résumées dans le schéma ci-dessous, dont les photos sont issues de notre code Matlab :

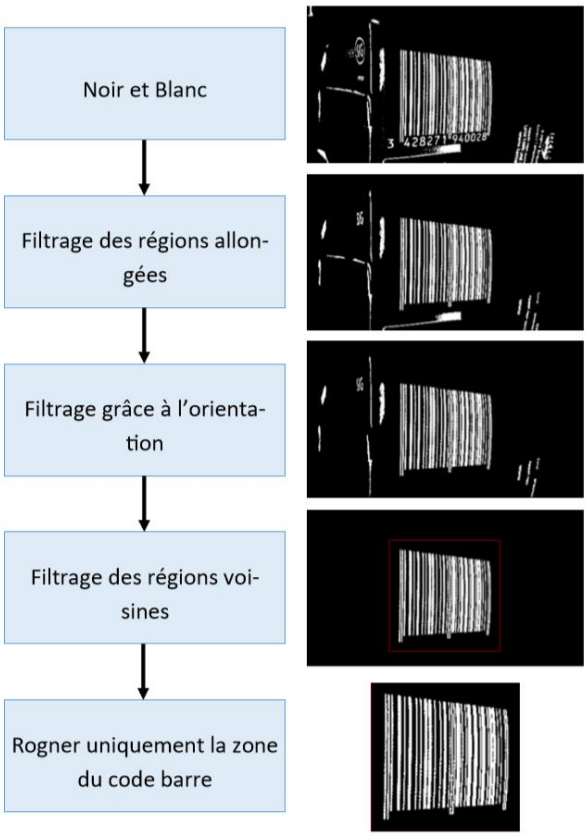

Figure 7 : Protocole d'extraction du code-barres

Pour l'ensemble des étapes de mise en évidence du code-barres, nous avons utilisé la fonction regionprops(image, *propriétés*) de Matlab. Cette fonction indique quelques propriétés, à spécifier en argument, des différentes régions de l'image binarisée. Ces régions sont des ensembles de pixels blancs connexes. Ces propriétés sont regroupées dans une structure. La longueur de cette structure, c'est à dire son nombre d'éléments, correspond au nombre total de régions dans l'image. Chaque élément est constitué de plusieurs variables qui sont les propriétés de la région, comme par exemple les coordonnées de son barycentre, son orientation…

III.1. Recherche des régions allongées

Cette phase de traitement permet de conserver uniquement les régions allongées. Pour déterminer si une région est allongée ou non, on utilise les propriétés *MajorAxisLength* et *MinorAxisLength*, qui sont les longueurs des axes principaux de chaque région. Ces axes sont représentés en bleu sur la figure suivante, extraite de la documentation Matlab, pour une région de 4 pixels :

Figure 8 : Axes principaux d'une région de 4 pixels. Ref[5]

Pour notre code, on considère qu'une région est allongée si le rapport des longueurs de ces axes est supérieur à 5. Les autres régions sont éliminées.

La limite de ce filtre est qu'il supprime définitivement certaines régions. Si les barres sont bien séparées, cela ne pose pas de problème, mais si le seuillage n'était pas optimal, alors certaines barres du code peuvent être supprimées. Cela peut ne pas poser de problème pour la suite puisqu'une fois avoir trouvé la zone où se situe le code barre, on repart de l'image binarisée initiale qui n'a subi aucun filtre de régions. En revanche, il sera impossible de trouver la zone où se situe le code-barres, si les barres sont toutes rattachées et ne forment qu'une seule région, comme sur la figure suivante :

Figure 9 : Exemple de codes-barres ne pouvant être décodés

III.2. Recherche des régions de même orientation

Une autre propriété intéressante de la fonction regionprops() est l'orientation. Il s'agit de l'angle entre l'axe principale de la région et l'axe horizontale. Cet angle est illustré sur la figure suivante :

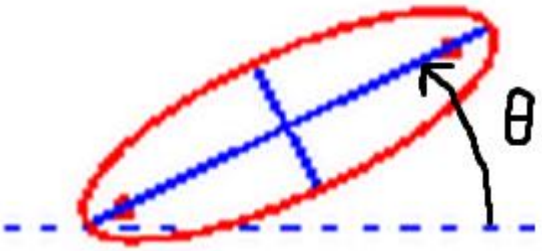

Figure 10 : Orientation d'une région. Inspiré de Ref[5]

On considère que deux régions ont la même orientation si l'angle entre leur axe principal est inférieur à 5 degrés, valeur fixée arbitrairement. Pour chaque région, on regarde combien de régions ont la même orientation. On garde le groupe de régions le plus nombreux. Les autres sont éliminées.

Là encore, ce filtrage montre certaines limites. En effet, si le groupe de régions dont l'orientation apparaît le plus souvent, ne correspond pas aux barres du code-barres, le code sera effacé et ne pourra pas être décodé. Mais c'est très rarement le cas : il faudrait une image ou il y a beaucoup de régions allongées, autres que les barres du code-barres. La seconde limite est due au seuil de 5 degrés. Par exemple, si le code-barres est très déformé, le filtre ne va conserver que la moitié du code, puisqu'il y aura un écart d'orientation de plus de 5 degrés entre les barres aux extrémités du code-barres. La figure suivante illustre ce phénomène.

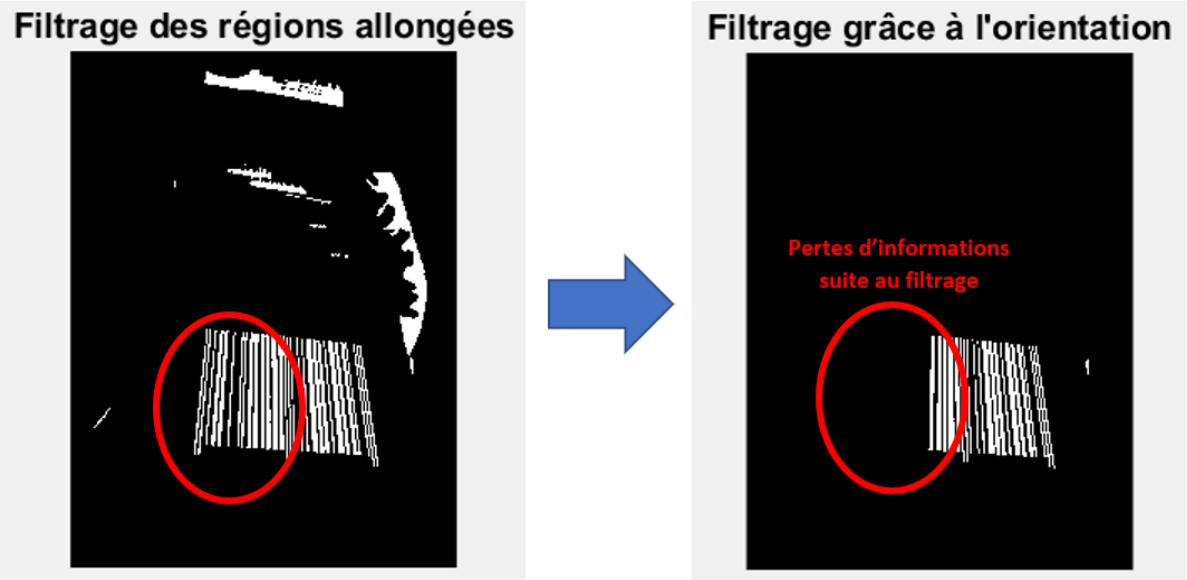

Figure 11 : Exemple de disparition de lignes du code-barres lors de la sélection par l'orientation

Dans le code final, nous avons adopté une valeur de 10° qui permet un tri efficace tout en conservant l'essentiel des codes même vus de biais comme celui de la figure 10 ci-dessus que nous avons tout de même pu décoder.

III.3. Recherche des régions voisines

A ce stade, il peut rester des éléments résiduels autour du code barre comme la barre d'un "L" dans le nom du produit ou une arête de l'emballage. Une caractéristique essentielle du code-barres est la très grande proximité de ses éléments qui nous permet de filtrer une partie des éléments résiduels. Pour ce faire, on utilise les coordonnées du 'Centroid', le barycentre des régions restantes. Pour chacune d'entre elles, on évalue le total T des distances avec ses cinq plus proches voisins. On sélectionne le plus petit T nommé Tmin et on élimine toutes les régions i pour lesquelles Ti>2.25xTmin.

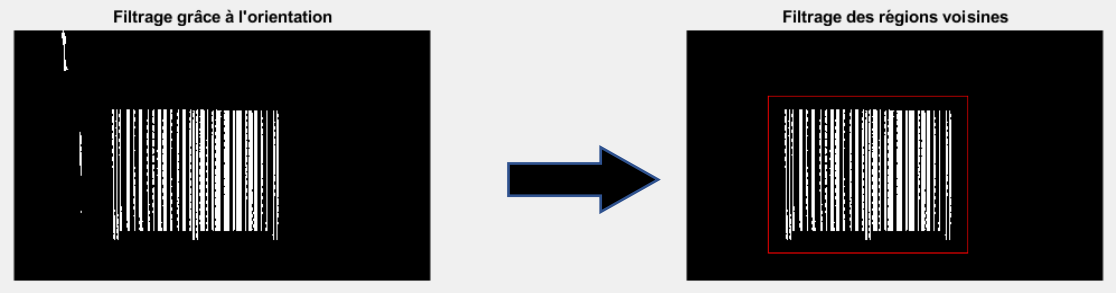

Figure 12 : Exemple de filtrage de régions isolées

Sur la figure 12 ci-dessus, on remarque la disparition des trois régions à gauche de l'image lors du passage par le filtre sur les régions voisines.

Dans ce filtre, deux valeurs sont définies expérimentalement, l'une influençant l'autre. D'abord le nombre de voisins. S'il existe un groupe de quelques tâches résiduelles proches et qu'on ne considère que les tâches de ce groupe, elles auront toutes des voisins proches et un faible T : il faut donc considérer un nombre significatif de voisins. D'un autre côté, plus le nombre de voisins considérés est grand, plus l'écart entre le T d'une barre au milieu du code et d'une barre du bord sera important : la limite entre le T d'une barre extrême du code et d'une tâche extérieure au code s'amenuise. Une fois le nombre de voisins fixé expérimentalement, il faut déterminer le coefficient de sélection, fixé ici à 2.25. Il s'agit du nombre le plus petit (on veut un filtre sélectif) permettant de conserver les barres extrêmes du code lors de nos essais successifs.

III.4. Sélection des régions par leur taille

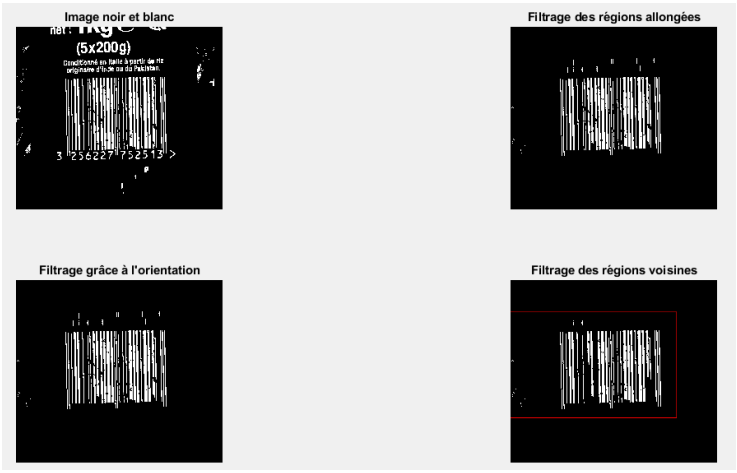

Figure 13 : Code barre présentant des défauts ponctuels après application de tous les filtres

Sur l'image en bas à droite (figure 13), on remarque à gauche du code-barres tout un ensemble de petits points. Ils sont dus à une illustration imprimée à côté du code et légèrement pixelisée. Les contours de ces pixels, très nombreux, sont orientés aléatoirement et forment des régions dont certaines arrivent à passer tous les filtres précédents. Comme ils sont peu nombreux à cette étape, on peut les séparer du code grâce à leur taille. Dans regionprops(), on utilise 'MajorAxisLength', la plus grande dimension des régions restantes. On considère la médiane de ces valeurs, elle correspond généralement à la dimension d'une barre du code : on peut supprimer toutes les barres largement inférieures ainsi que trois fois plus grandes. Les valeurs choisies pour sélectionner ou non une barre

sont expérimentales : on veut garder toutes les barres (coupées en deux ou trois morceaux parfois) du code tout en supprimant les résidus de pixels.

On remarquera qu'il s'agit de l'un des rares défauts d'une image de bonne résolution. On travaille ici sur une image de plus de 2M pixels, bien centrée sur le code. Avec une image de qualité inférieure, les défauts d'impression n'auraient pas été visibles et on aurait eu moins de régions à filtrer donc moins de chances que certaines passent tous les filtres. Bien sûr, nous avons pensé à homogénéiser les images pour que les petits défauts soient éliminés dès le départ mais on prend le risque de ne plus pouvoir déchiffrer des images de faible contraste.

III.5. Récupération de la zone où se situe le code barre

Nous avons réalisé une fonction permettant de trouver la boite englobant les régions restantes après ces phases de filtrage. Cette boîte est définie dans le repère absolue par le vecteur [Xmin Ymin Xlength Ylength]. Ce dernier est utilisé par la fonction imcrop() qui se charge de rogner l'image.

La boîte englobante possède une marge proportionnelle à la taille des régions restantes qui permet d'inclure une barre du bord qui aurait été éliminée avec les filtres précédents.

De plus, on cherche l'orientation moyenne des régions restantes, afin de remettre les barres à la verticale. Puis, on utilise la fonction imrotate() de Matlab qui tourne le code-barres. Avant d'utiliser la propriété *orientation* de la fonction regionprops(), nous avons essayé d'utiliser le spectre dans le domaine de Fourier. On distingue très bien l'orientation du code-barres dans ce domaine :

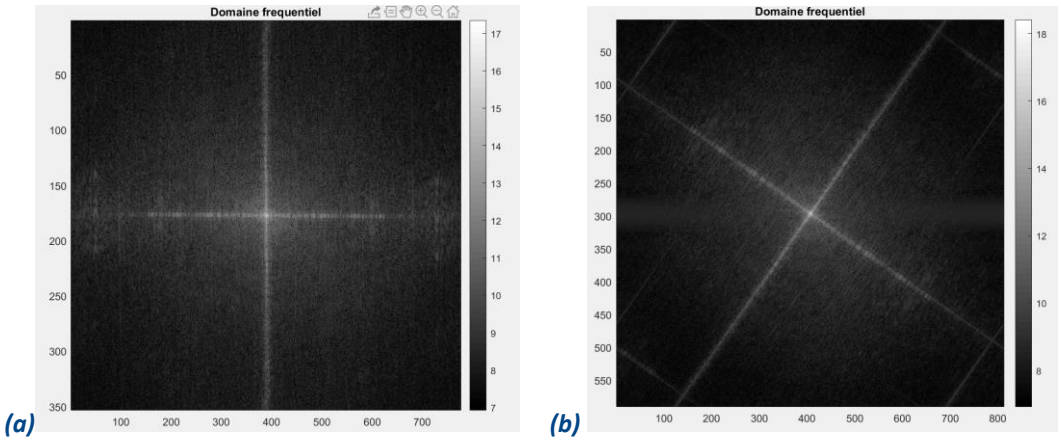

Figure 14: Domaine fréquentiel d'une image avec un code-barres droit (a) et incliné (b)

Nous avons aussi tenté de trouver l'orientation des barres avec une méthode plus fastidieuse : en partant d'un pixel au hasard en haut de l'image et en étudiant ses voisins de même couleur pour descendre, pixel par pixel, dans une même barre du code. Ce chemin de pixel suivant l'orientation d'une barre nous permettait de remonter à l'angle formé avec la verticale.

Cependant, ces deux méthodes n'ont pas été très concluantes : elles fonctionnaient mais présentait une incertitude sur l'angle entre 2 et 3°. La simplicité qu'offrait la fonction regionprops() et sa grande précision sur les angles nous ont poussé à l'adopter définitivement.

IV. Du décodage à la création de l'interface

IV.1. Phase de décodage

Nous arrivons à la partie cruciale : le code-barres est bien cadré, il faut le décoder. On repart de la dernière image filtrée, bien recadrée par notre boîte autour du code.

Pour décoder, on fonctionne ligne par ligne sur l'image. Pour chaque ligne, on l'extrait et on compte le nombre de barres verticales (alternance de pixels blanc-noir). Pour la déchiffrer, il faut savoir quelle est la taille des barres élémentaires afin de la comparer avec les barres qu'on essaie de lire. Connaissant ces tailles de référence, il ne reste qu'à étudier la composition de la ligne extraite : on parcourt les pixels entre les barres de début et de fin. Comme vu précédemment, chaque série de quatre barres définit un chiffre et une lettre, nous devons donc comparer les groupes de quatre barres avec les références des tables. Une fois la série de chiffres obtenue, on vérifie qu'elle correspond bien à un code-barres valide avec le chiffre de contrôle. Si le code n'est pas valide, on passe à la ligne suivante jusqu'à trouver le code… ou arriver en bas de l'image. Dans ce cas on retourne l'image en effectuant une rotation à 180° : elle pourrait être à l'envers. On essaie à nouveau le décodage.

Cette méthode de base a été complétée tout au long du projet lorsque nous avons identifié des problèmes récurrents. D'abord, les codes sur des surfaces courbées nous ont posé un problème. En effet, sur la surface présentée ci-contre, les barres du milieu apparaissent plus larges que les barres des côtés car la surface est bombée. Pour nous affranchir de cette limite, on considère à chaque ligne les trois groupes de barres élémentaires, sur les côtés et au centre, puis on effectue une interpolation polynomiale (polyfit()) nous indiquant la taille d'une barre élémentaire en fonction de la position sur la ligne. Ainsi, l'image ci-contre a pu être décodée.

Figure 15 : Exemple de code-barres imprimé sur une surface courbe

Un second écueil est engendré par les photos prises de biais. Pour le contourner, nous avons essayé, dans notre fonction Redresse.m, de projeter les codes-barres pour corriger leur distorsion. Pour ce faire, nous recherchons à l'aide des barycentres des régions, les deux barres extrêmes du code. Ensuite, grâce aux propriétés de ces régions limites, nous déterminons la position des quatre coins du code que nous projetons sur un rectangle pour revenir à une vue de face du code. Pour cela, nous utilisons fitgeotrans() en mode projectif (un mode linéaire ne fonctionne que si l'angle de projection est faible). Ci-contre, nous représentons en bleu claire l'espace de départ qui sera projeté dans un rectangle, le résultat de la projection se trouve à droite et rend le code parfaitement lisible.

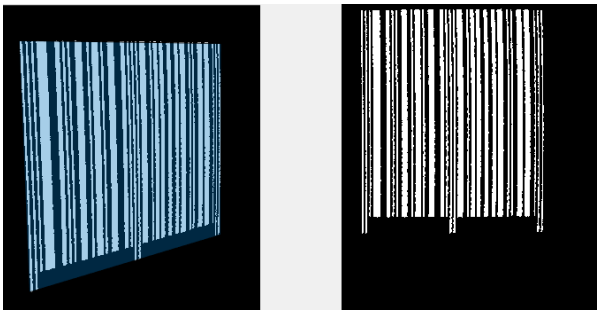

Figure 16 : Code-barres penché, avant et après binarisation et redressement

Cette fonction nous permet d'élargir largement le spectre des images décodées par notre algorithme. Cependant, elle est limitée. En effet, à partir d'un angle limite (lié à la résolution spatiale de l'image), deux barres successives se trouvent confondues : même en projetant, il est impossible de décoder l'image. Une seconde limite à la projection est liée à la détection des angles du code-barres. Il arrive que le pré-traitement entraîne la disparition de certaines portions de barres ou qu'une barre se retrouve scindée en deux morceaux (si le code est rayé comme sur la figure 17). Dans ce cas, on ne trouvera pas les angles du code et la projection ne permettra aucune amélioration.

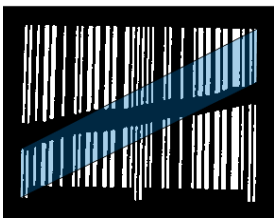

Figure 17 : Exemple de code-barres rayé

Un dernier écueil est celui de la rayure transversale. Dans le cas d'une rayure noire sur le code-barres (coup de crayon, tâche), les barres se retrouvent liées et le traitement sur les régions ne fonctionne plus. Ce n'est que peu préoccupant car ce genre de rayures est rare. En revanche, nous avons souvent trouvé des codes rayés par le frottement. Dans ce cas, la rayure est blanche et coupe les barres en deux. Lorsqu'on lit le code-barres ligne par ligne, la rayure change la valeur d'un pixel noir qui devient blanc et la ligne devient indéchiffrable. Dans le cas où la rayure couvre toutes les lignes, alors le code est indéchiffrable. Pour pallier les rayures, nous avons créé (à partir d'une idée proposée sur le forum d'échanges Matlab Ref[5]) une fonction définissant la couleur des colonnes. Pour chaque colonne, on compte le nombre de pixels blancs et noirs. La couleur possédant une majorité de pixels l'emporte et la colonne devient unie. De ce fait, une petite rayure blanche se verra absorbée par les colonnes majoritairement noires. On travaille sur des images en couleurs inversées, c'est pour cela que les tâches sont noires sur l'exemple ci-dessous (figure 18) mais l'idée reste la même ! Mais une fois encore, la solution n'est pas garantie dans tous les cas. Il arrive que le code-barres soit mal cadré, lorsqu'il reste quelques régions en dehors du code même après filtrage. Dans ce cas, le code n'occupant pas la majeure partie de l'espace, le raisonnement colonne par colonne n'est plus valide et le code devient uni.

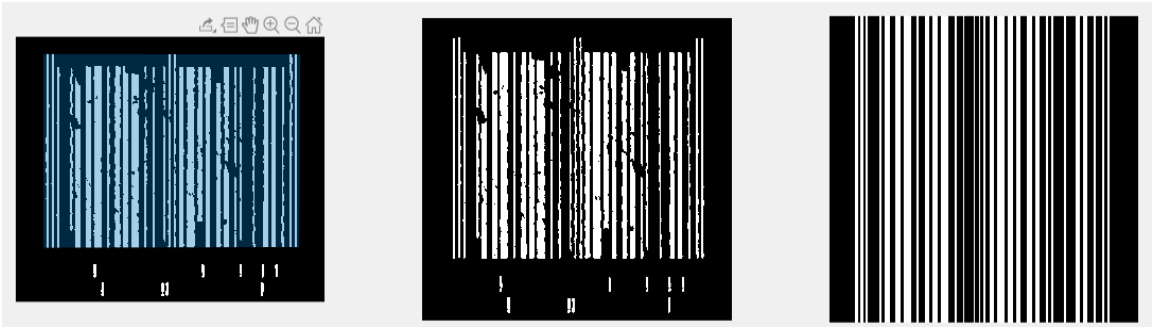

Figure 18 : Exemple d'application du moyennage sur les colonnes

Pour conclure cette partie, nous présentons notre solution finale pour décoder, bien qu'elle ne soit pas fonctionnelle dans tous les cas, bien sûr. Après le traitement de l'image avec les différents filtres évoqués précédemment, nous essayons de décoder l'image en utilisant seulement l'interpolation polynomiale pour nous affranchir des rayons de courbure des codes. Si le déchiffrage est impossible, alors on projette l'image pour redresser une potentielle vue de biais et on effectue l'homogénéisation des colonnes couleur par couleur. Nous essayons la projection sur deux images différentes, cadrées autour du code-barres grâce à la boîte définie en fin de filtrage. La première est l'image finale ayant subie tous les filtres, qui risque d'avoir coupé une barre en deux. La seconde est l'image filtrée uniquement sur l'allongement des formes qui a conservé toutes les barres mais parfois aussi des régions en trop qui peuvent être prises comme références pour les angles du code-barres. Pour déchiffrer le code-barre, on réalise donc trois essais sur trois images différentes. En effet, on observe expérimentalement que le déchiffrage du code ne fonctionne pas toujours avec la même image parmi

les trois. On optimise nos chances de lire le code-barres mais dans le même temps, on augmente le temps nécessaire à déchiffrer s'il faut essayer trois images au lieu d'une.

Le schéma suivant permet de clarifier l'algorithmie du décodage une fois la zone du code-barres obtenue :

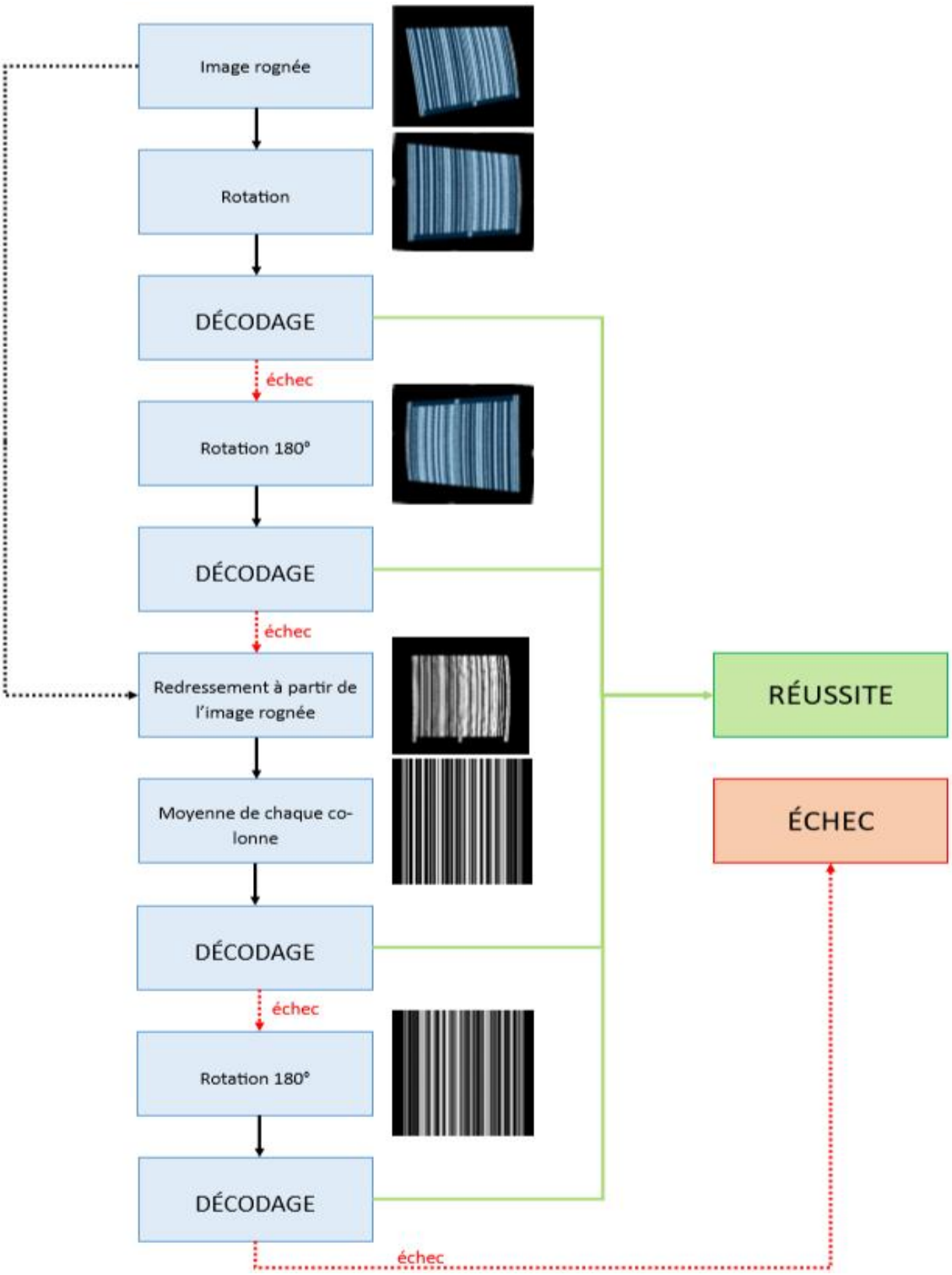

Figure 19 : Schéma du processus de décodage

IV.2. Limites de notre code

En commençant notre projet, nous espérions pouvoir décoder n'importe quel code. Le décodage d'un code, même bien photographié, s'est avéré être plus difficile que prévu, et les conditions particulières qui faisaient que notre code ne fonctionnait pas étaient nombreuses. Nous nous sommes donc attaqués à chaque problème pas à pas.

Le principal problème que nous avons rencontré est dû à la résolution de l'image. En effet, si la résolution est trop faible, la largeur de chaque barre en nombre de pixels peut être trop approximative. Si la largeur d'une barre sur les 57 barres blanches et noires n'a pas le bon rapport avec la largeur de référence des barres de contrôle, le programme ne fonctionne pas. Il est donc important d'avoir une bonne résolution. Pour indiquer la résolution minimale nécessaire au déchiffrage d'un code, nous avons réalisé une série d'essais sur une photo parfaitement centrée sur le code-barres dans un éclairage uniforme. Jusqu'à des dimensions de 200x148p, la lecture du code est possible. A 150x111p, le code annoncé par le programme n'est pas le bon. En dessous, aucune lecture n'est possible, les barres ne sont même plus séparées. Ainsi, dans des conditions optimales, en cadrant parfaitement la photo, il est possible de déchiffrer une image à partir d'environ 30 000 pixels. Cette limite basse augmente nettement lorsque la photo est prise avec un angle, ce qui rapproche visuellement les barres...

L'acquisition peut aussi être d'une qualité moindre lorsqu'elle est floue. Après avoir construit l'image en nuances de gris, on obtient, entre deux barres noires successives, un dégradé, sans discontinuité. Selon la valeur de seuillage, il est possible de changer une partie de ce dégradé en blanc et une autre en noir alors que dans l'idéal, tout l'intervalle entre deux barres noires devrait être blanc. Ainsi, pour un seuil trop bas, nous aurons des barres blanches trop larges et des barres noires trop fines et pour un seuil trop élevé, nous obtiendrons des barres blanches trop fines et des barres noires trop larges. Dans les deux cas, il sera difficile pour l'algorithme de décoder l'image, pour les mêmes raisons qu'au paragraphe précédent. Le seuillage adaptatif apporte une solution convenable au problème en changeant le seuil en fonction de chaque zone de l'image. Cette méthode est efficace seulement si le flou est léger (manque de luminosité par exemple). Il faut tout de même que la photo soit prise dans les meilleures conditions. La mise au point doit nécessairement être effectuée sur le code-barres.

Nous devons aussi nous intéresser à la profondeur de champ. Dans certains cas, lorsque le code barre est pris de côté, la mise au point s'effectue sur une partie du code-barres, et certaines parties du code deviennent floues. Si la profondeur de champ n'est pas suffisante pour une photographie avec une vue en biais, le code peut être indéchiffrable. Cette limite est récurrente avec les appareils associés à nos smartphones qui ont souvent une profondeur de champ très limitée lorsque la distance focale est courte. Ainsi, s'approcher le plus possible d'un code-barres n'est pas toujours la meilleure solution car les bords de l'image peuvent devenir flous.

Pour le moment, nous n'avons pas abordé la question de l'éclairage. Il constitue le quatrième critère d'acquisition déterminant conditionnant la réussite de la lecture du code, avec la résolution de l'image, l'angle de prise de vue, et le cadrage. Nous avons manqué de temps pour explorer en profondeur les questions d'éclairage, mais notons quelques limites à notre code. D'abord la question du flash, comme sur l'image ci-contre, qui couvre une partie des barres. S'il est aussi intense que sur cette image, alors le code devient illisible car fond et barres adoptent la même intensité lumineuse. Sur l'image en noir et blanc, on remarque une zone noire qui correspond au flash. Si elle recouvre une partie même mineure du code, il reste déchiffrable. Dans des conditions d'éclairage moins extrêmes, on rencontre le second cas illustré ci-contre (à gauche). Le code est éclairé avec deux intensités différentes du fait

de la présence d'une ombre. Dans ce cas, le seuil utilisé doit conserver toutes les barres et supprimer le fond, qu'il soit très lumineux ou non. Un seuil commun à tous les pixels de l'image est utilisé et permet de récupérer une image nette du code. Des conditions d'éclairage différentes sur un code-barres n'empêchent pas la lecture, tant que la luminosité n'est pas trop intense. En revanche, on remarque une zone blanche dans le coin haut-gauche, celle-ci touche les autres régions blanches et empêche le décodage. Elle est présente même dans le cas d'un seuil adaptatif car dans cette région, quelques points de l'emballage sont très lumineux, d'autres beaucoup moins. C'est donc une limite dans les conditions d'éclairage pour les surfaces vernies et particulièrement réfléchissantes : une acquisition en plein soleil ou avec un flash peut, bien souvent, devenir indéchiffrable.

IV.3. L'interface

La finalité de notre projet étant d'offrir à l'utilisateur un service rapide et pratique pour consulter les produits qu'il achète, nous avons mis en place une interface Matlab dont le visuel est le suivant :

Figure 20 : Capture d'écran de l'interface conçue pour l'utilisateur

Cette application permet d'utiliser des images enregistrées sur l'appareil (Load image) ou d'utiliser la webcam (Start camera) pour prendre un cliché du produit. En effet, le code que nous avons produit est fonctionnel mais ne déchiffre pas toutes les images. Pour l'utilisateur de notre code, il faudrait donc prendre plusieurs photos pour être sûr d'obtenir des résultats satisfaisants. Pour pallier cet aspect peu pratique, nous avons décidé de mettre en œuvre une solution par webcam afin de prendre plusieurs images jusqu'à la réussite du déchiffrage. Lorsqu'une image a été chargée, elle s'affiche au centre de l'application. Une fois déchiffrée, le code-barres s'affiche au-dessous et l'application récupère les informations correspondantes sur Open Food Facts qu'elle affiche à droite (nom et photo du produit). Une petite case à cocher permet d'ouvrir la page Open Food Facts pour obtenir toutes les informations liées au produit.

Cependant, bien que l'application remplisse tout à fait les objectifs que nous nous étions fixés, elle est très lente, ce qui peut être un inconvénient. Notre code n'est pas optimal, l'acquisition d'images par webcam peut entraîner un ralentissement important de la machine. Il serait important, avant de partager cette application, de travailler sur son efficacité, peut-être en utilisant un autre environnement de développement (notre code sans interface visuelle est bien plus rapide !).

V. Etude économique

V.1. Application à faible coût pour la société

L'application développée a pour but d'établir un lien de confiance entre le consommateur et la société de grande distribution.

Pour réaliser une application fonctionnelle représentant un faible investissement pour la société, nous estimons qu'il faut une petite équipe d'ingénieurs pour la développer. Celle-ci doit pouvoir être développée en 1 mois et composée d'une interface avec l'utilisateur fonctionnelle. Pour déterminer le coût de développement d'une application, on peut se concentrer sur plusieurs facteurs :

- Quelle·s plateforme·s va·ont accueillir notre application ?
- Quel·s support·s matériel·s sera·ont compatible·s avec notre application ?
- L'application sera-t-elle gratuite ? Si oui comportera-t-elle des publicités ?
- Comment sera promue l'application ?

En fonction de ces critères on peut déterminer le coût de développement de notre application.

Pour minimiser le coût de notre application, nous considérons qu'elle sera développée uniquement sur Android et ne sera donc pas disponible aux utilisateurs IOS. En effet, les utilisateurs Android étant plus nombreux que ceux présents sur IOS, on estime préférable de développer l'application uniquement sur Android pour réduire le coût de mise en service de l'application. De plus, l'application ne sera pas disponible sur tablette Android pour minimiser les coûts liés au développement de l'interface graphique. Effectivement pour implémenter une application sur une tablette et sur un smartphone il faut prévoir deux interfaces graphiques différentes, les surfaces d'affichages n'étant pas du même ordre de grandeur. Comme on recherche un faible coût de développement de l'application, une équipe minimale de 3-4 ingénieurs est donc suffisante à sa mise en service sur smartphone.

Ainsi, en admettant qu'une équipe composée de 4 ingénieurs puisse développer ce type d'application en 1 mois on estime le coût de développement à :

- 12 000 € pour le salaire moyen de 4 ingénieurs compétents dans le domaine
- 25 € pour les frais de mise en service sur Google Play

Ensuite pour populariser l'utilisation de notre application il faut compter des frais de promotion. Pour limiter les coûts de promotion de notre application on peut imaginer qu'elle sera promue par l'intermédiaire des réseaux sociaux, de spots publicitaires diffusés à la radio et sur les bus, abribus. On peut chiffrer les coûts de promotion dans ces conditions à :

- 70-110 € HT pour l'affichage temporaire d'une pub présente sur les grands axes routiers à destination des automobilistes. ref[12]
- 55-90 € pour l'affichage temporaire d'une pub à l'arrière d'un bus. ref[12]
- 250 € pour une annonce publicitaire de 30 secondes diffusée entre 7h et 9h pour une radio parisienne et seulement 70 € si elle est diffusée entre 19h et 20h. ref[11]
- 70 € pour la même annonce publicitaire diffusée entre 7h et 9h dans une radio régionale et seulement 20€ entre 19h et 20h. ref[11]
- Il faut compter environ 1500€ /mois pour une campagne de pub sur 3 réseaux sociaux comprenant 45 messages sociaux, 10 images personnalisés, 4 campagnes publicitaires. ref[14]

Cela représente un total de 2150€ pour une campagne de promotion d'un mois.

Ainsi en réduisant au maximum le coût de développement de notre application nous arrivons à un total de 14 175€ à débourser pour la société de grande surface. Cela représente un faible coût d'investissement pour une société de grande surface puisqu'en moyenne le chiffre d'affaire d'une caisse d'un magasin bien situé est compris entre 10 000 et 15 000€ par jour.

Une fois mise à disposition des consommateurs, l'application sera fonctionnelle mais présentée avec une interface graphique simple et très épurée. On peut alors prévoir que l'application sera peu utilisée car ne présentant aucun avantage pratique ou esthétique en comparaison à ses concurrents directs, à savoir Yuka, Kwalito ou encore BuyOrNot. L'application pourrait même ternir l'image de marque de la société de grande distribution si elle paraît trop austère. En effet, l'interface graphique étant simpliste et basique elle renvoie une image d'application low cost. Cet aspect est renforcé par l'absence d'adaptation aux services IOS. Si tous les consommateurs possesseurs d'Iphones ne peuvent pas télécharger l'application, ils ne pourront que recommander d'autres services. Il paraît alors compliqué, pour la société, de réussir à séduire une large part de ses consommateurs en limitant son application au strict nécessaire. D'autres investissements semblent cruciaux.

V.2. Application complète à coût élevé

Même si notre projet rentre plutôt dans le cadre d'une application simplifiée à faible coût comme vue ci-dessus, nous pouvons tout de même nous intéresser à ce que les leaders de la grande distribution attendent en général de ce genre d'application.

On peut aussi considérer que l'équipe d'ingénieurs ne s'occupe ici que du traitement d'image et du décodage. Une équipe de développeurs pourrait, s'occuper de créer l'application avec peut-être les conseils d'un designer. Ils pourraient faire l'application sous Android et sur IOS pour cibler la majorité des consommateurs. Évidemment, l'application ainsi créée serait beaucoup plus propre d'un point de vue design et plus ergonomique. L'application pourrait coûter beaucoup plus cher (30 000 à 50 000 € voire plus) mais l'utilisateur se laisserait plus facilement séduire par l'expérience. On pourrait aussi mettre en place un système de conseils et de redirection vers des articles de meilleure qualité, moins cher ou même temporairement en soldes suivant le désir du client (nécessité dès lors de créer un compte utilisateur ou tout du moins des paramètres variables). Le coût peut encore augmenter et atteindre la centaine de milliers d'euros pour créer ce système personnalisable dont les services sont d'une qualité bien supérieure à ceux offerts par l'application de base. Un utilisateur satisfait de l'application aura tendance à plus consommer dans les magasins de cette marque de grande distribution. Les retombées économiques de l'application pourront alors très largement compenser le coût de l'implémentation et de l'entretien de l'application.

Finalement, ce qui définit le coût de l'application est vraiment le cahier des charges donné. Une grande société de distribution aura tendance à donner un cahier des charges très complet, quitte à devoir investir beaucoup d'argent, car l'image de la marque et donc les retombées économiques en seront d'autant plus grandes. Actuellement, de nombreuses sociétés de la grande distribution travaillent sur des applications similaires. On peut citer notamment SuperU et Intermarché qui ont chacun développé leur propre application scanner de produit. Il faut toutefois se rappeler que l'application aura des concurrents comme Yuka par exemple, et que si notre application n'apporte pas quelque chose en plus (comme des promotions ou une véritable personnalisation des notations et recommandations par exemple), ils préféreront la plupart du temps l'utilisation de Yuka qui fonctionne dans tous les magasins et qui est indépendante de toutes les entreprises de l'agroalimentaire.

CONCLUSION

Au travers de ce projet, nous avons découvert plus en détails ce qu'était un code-barres et pris conscience de l'importance de ce système de référencement dans le monde. Au cœur des échanges économiques, il est à l'origine d'une distribution facilitée des produits grâce à ses aspects pratiques permettant le suivi et le contrôle. Ainsi, l'application que nous avons développée permet d'avoir un accès direct et rapide aux informations de la quasi-totalité des produits alimentaires en France. Cependant, mettre en œuvre une application de scan de codes-barres n'est pas simple. Le décodeur que nous avons créé n'est pas parfaitement robuste puisqu'il présente plusieurs limites. Si nous avions eu plus de temps, nous aurions pu améliorer les algorithmes et réfléchir à d'autres méthodes pour localiser un code-barres dans une image. Bien que notre code ne soit pas très pratique puisqu'il ne fonctionne qu'avec Matlab, ce projet nous a permis de découvrir les bases du traitement d'images et de développer nos connaissances en Matlab.

De plus, une étude des coûts nous a permis d'estimer les hypothétiques retombées économiques de la mise en place d'une application de scan de codes-barres par une société de grande distribution. Pour cela, nous avons découvert les techniques de marketing et de communication en usage mais aussi les stratégies de promotion de mise en service d'une application. Il apparaît alors évident que fournir une application permettant aux consommateurs d'avoir une connaissance approfondie de la composition et de la traçabilité des produits d'une société de grande distribution contribue à son image de marque.

Enfin, nous avons renforcé notre capacité à travailler en équipe, à distance, en utilisant des logiciels de partage de fichiers tels que les drives ou GitHub. Première expérience d'un projet d'une telle ampleur réalisé entièrement à distance, le développement de cette application nous aura beaucoup appris, au-delà du code.

Investissement de chacun :

Noms	METRAS Gaël	BARGES Tanguy	GAUDARD Gaëtan	JAMADI Nassime
Bonus/Malus	0	0	0	0

BIBLIOGRAPHIE

- **Recherche sur les codes-barres :**

1. WIKIPEDIA , *Code-barres EAN - Wikipédia* **[en ligne]**. (modifié en 2019), Disponible sur < https://fr.wikipedia.org/wiki/Code-barres_EAN#Description_sommaire > (Consulté le 28/04/2020)
2. DEMONTE J.B. , *Barcode-Coder* **[en ligne]**. (mis en ligne en mai 2009, modifié en fév. 2019), Disponible sur < https://barcode-coder.com/fr/specification-ean-13-102.html > (Consulté le 26/04/2020)
3. GOMARO s.a. , *Le code EAN ou UPC ou GTIN – info. Gomaro s.a.* **[en ligne]**. Disponible sur : < http://www.gomaro.ch/codeean.htm > (Consulté le 28/05/2020)

- **Documentation technique pour la mise en œuvre de l'application :**

4. ABECASSIS F. et MARQUEGNIES J. , *Traitement d'image - Rapport de projet* **[en ligne]**. EPITA, janvier 2012. Disponible sur < http://felix.abecassis.me/wp-content/uploads/2012/06/tirf.pdf> (Consulté le 27/03/2020)
5. Documentation MATLAB, *Mathworks – Editeur de MATLAB et Simulink* **[en ligne].** Disponible sur < https://fr.mathworks.com/ >
6. Chaine YouTube MATLAB, *Image Processing Made Easy - MATLAB Video* **[en ligne].** [5/09/2014] [38'39''] Disponible sur < https://www.youtube.com/watch?v=1-jURfDzP1s > (Consulté le 26/04/2020)
7. Chaine YouTube CHAUSSARD John, *Traitement d'images - Morphologie - TopHat pour segmenter une image* **[en ligne].** [5/12/2019] [12'13''] Disponible sur < https://www.youtube.com/watch?time_continue=731&v=mhNNQG2COhI&feature=emb_logo > (Consulté le 28/04/2020)
8. Chaine YouTube AGRAWAL Rashi, *Segmentation using Threshold value- Adaptive and Otsu's Method* **[en ligne].** [26/02/2015] [5'04''] Disponible sur < https://www.youtube.com/watch?v=i1wAoIDar48 > (Consulté le 27/04/2020)

- **Etude de l'impact économique de l'application :**

9. ECOMMERCE-NATION, *GTIN/EAN, quels sont les bénéfices et usages de ces codes ?* **[en ligne]**. (modifié le 27/08/2019) Disponible sur < https://www.ecommerce-nation.fr/gtin-ean-benefices-et-usages-de-ces-codes/ > (Consulté le 30/05/2020)
10. StarOfService, *Quel est le tarif d'un consultant marketing ?* **[en ligne]**. Disponible sur < https://www.starofservice.com/cost-guides/combien-coutent-services-consultant-marketing > (Consulté le 23/05/2020)
11. StarOfService, *Quel est le tarif d'achat d'un espace publicitaire ?* **[en ligne].** Disponible sur < https://www.starofservice.com/cost-guides/combien-coute-achat-espace-publicitaire > (Consulté le 23/05/2020)
12. ROGER Yann - ADINTIME, *Panneaux publicitaires : les types de campagnes d'affichage* **[en ligne].** (mis en ligne le 9/08/2019,) Disponible sur < https://adintime.com/fr/blog/panneaux-publicitaires-les-types-de-campagnes-daffichages-n35 > (Consulté le 23/05/2020)
13. FYGOSTUDIO, *Coût d'une publicité TV* **[en ligne]**. Disponible sur < https://fygostudio.com/prix-publicite-tv/ > (Consulté le 23/05/2020)
14. METADOSI, *Coûts de marketing des réseaux sociaux* **[en ligne]**. Disponible sur < https://www.metadosi.fr/prix-media-sociaux/ > (Consulté le 28/05/2020)